SGOU
Higher Education for All

Data Structures with C

COURSE CODE: M25CA01DC
Master of Computer Applications
Discipline Core Course
Self Learning Material

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Data Structures With C

Course Code: M25CA01DC

Semester - I

Discipline Core Course
Postgraduate Programme
Master of Computer Applications
Self Learning Material

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU
OPEN UNIVERSITY

DATA STRUCTURES WITH C

Course Code: M25CA01DC
Semester- I
Discipline Core Course
Master of Computer Applications

Scan this QR Code for reading the SLM on a digital device.

Edition
April 2026

ISBN 978-81-686027-4-8

9 788168 602748

www.sgou.ac.m

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear Learner,

It gives me immense pleasure and a deep sense of pride to warmly welcome you to Sreenarayanaguru Open University—a vibrant and progressive institution committed to transforming lives through inclusive, flexible, and high-quality education.

Established in September 2020 as a forward-looking initiative of the Government of Kerala, the University stands as a beacon of opportunity for learners seeking to advance their academic and professional aspirations through the open and distance learning mode. Guided by our foundational principle that "access and quality define equity," we are steadfast in our mission to democratize education while upholding uncompromising academic standards.

Our university is inspired by the timeless vision and philosophy of Sree Narayana Guru, whose ideals of knowledge, equality, and social transformation continue to guide our academic journey. His enduring legacy instils in us the responsibility to create an educational environment that empowers individuals, nurtures critical thinking, and contributes meaningfully to society.

Understanding the dynamic needs of contemporary learners, we have adopted a robust and learner-centric blended learning model, seamlessly integrating Self-Learning Materials, Academic Counselling, and Advanced Digital Learning Platforms. This holistic approach ensures flexibility without sacrificing academic depth, enabling you to learn at your own pace while remaining meaningfully connected to a vibrant academic ecosystem.

The Master of Computer Applications (MCA) programme you are embarking upon is carefully designed to position you at the forefront of the digital revolution. It uniquely blends strong theoretical foundations with practical, industry-oriented competencies. The curriculum emphasizes algorithmic thinking, system design, programming, database management, networking, and emerging technologies such as artificial intelligence, data science, and cloud computing.
What sets this programme apart is its forward-thinking design, which offers:

- Opportunities for skill enhancement aligned with industry needs
- Multidisciplinary learning pathways for broader intellectual development
- Exposure to innovative and emerging technology domains
- Multiple specialization options tailored to evolving career landscapes
- A strong focus on employability, entrepreneurship, and global career readiness
-

Our Self-Learning Materials are meticulously developed by experts, enriched with contemporary case studies, real-world applications, and practical insights to ensure clarity, engagement, and relevance. We are committed to equipping you not just with knowledge, but with the confidence and competence to excel in a competitive global environment.

At Sreenarayanaguru Open University, you are never alone in your learning journey. Our dedicated learner support system is designed to provide continuous academic guidance, timely assistance, and effective grievance redressal. We encourage you to actively engage with us, share your concerns, and make the most of the resources and support available to you.

As you begin this important phase of your academic journey, I urge you to embrace learning with curiosity, discipline, and determination. The world of technology is ever-evolving, and your willingness to adapt, innovate, and grow will define your success.

Remember, this is not just a programme—it is a pathway to transforming your future.
I wish you a fulfilling learning experience and a successful, inspiring career ahead.

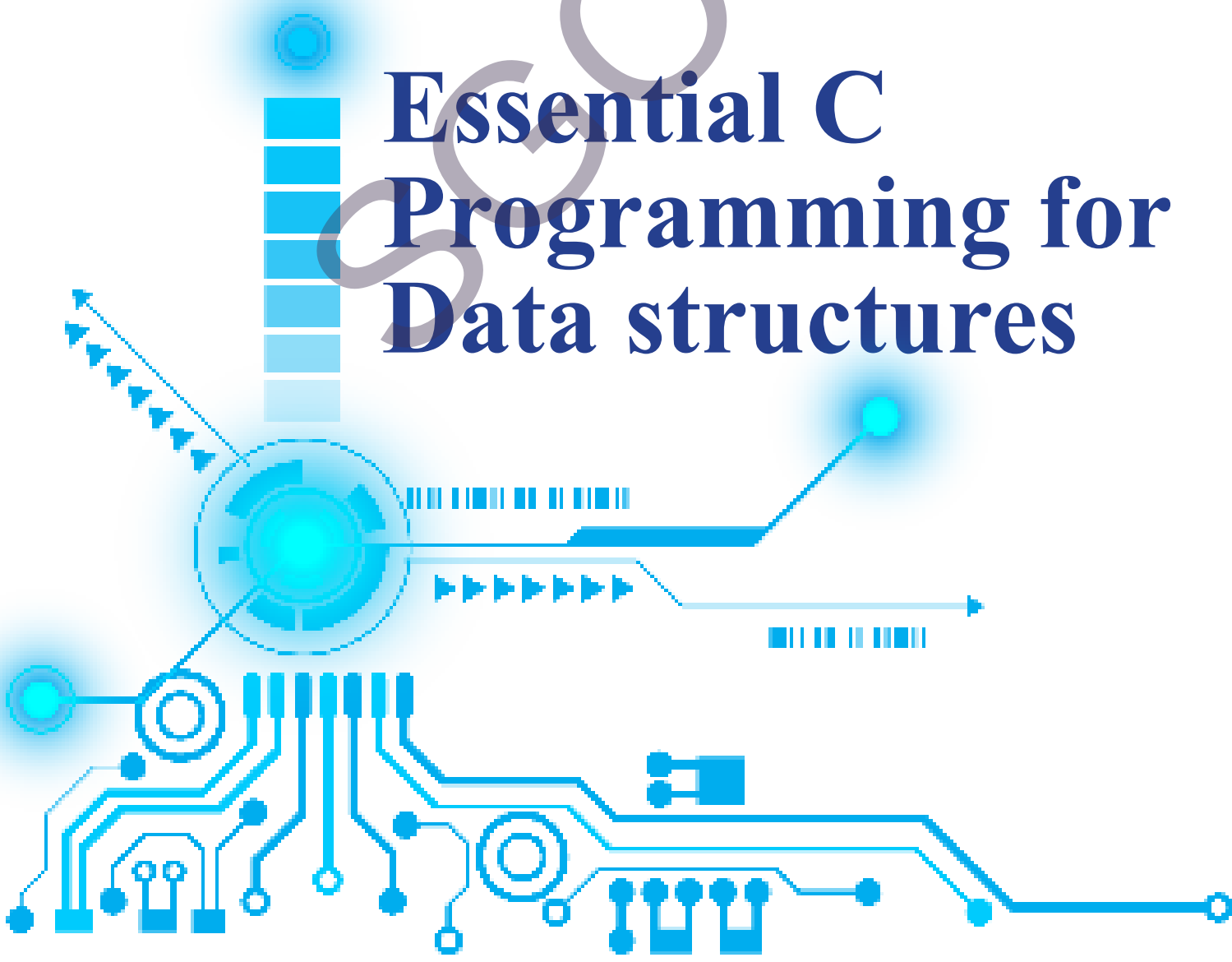
Warm regards,

Prof. (Dr.) Jagathy Raj V. P.
Vice Chancellor

Contents

BLOCK 1

Essential C Programming for Data structures



1 UNIT

Introduction to C Programming

Learning Outcomes

By the end of this unit, students will be able to:

- identify the different building blocks in C
- differentiate between variables and constants based on their declaration and purpose
- demonstrate the use of various operators in simple C programs
- analyze appropriate data types while writing C programs

Background

Computers are part of our daily life, from mobile phones and ATMs to traffic lights and medical devices. All these systems work using programs based on a set of basic rules. Computer languages are a way for humans to communicate with computers. They provide a set of instructions that a computer can understand and execute. Just as we use different languages to talk to each other, computers use programming languages to perform tasks. These languages are essential for writing computer programs, which tell a computer what to do step by step. In computer programming, writing instructions in a specific language allows us to solve problems, create applications, and control hardware. Without programming languages, we would not be able to interact with computers in a meaningful way.

Before starting a program, it is important to understand the terms source code and object code. Source code is the set of human-readable instructions written by a programmer using a programming language like C. It is what you type into a text editor before it is processed. This source code cannot be directly understood by the

computer. The compiler translates the source code into object code, which is in machine language (binary format) that the computer can understand and execute. This process is necessary because computers can only work with binary instructions, not human-readable code.

Keywords

Variables, Constants, Operators, size of , comma, Operator precedence.

Discussion

C is one of the oldest and most widely used programming languages in the world. It was developed in the early 1970s by Dennis Ritchie at Bell Labs. C was created to write the UNIX operating system. Over the years, it has become the foundation for many other languages such as C++, Java, and Python. C is often called a middle-level language because it combines features of both high-level languages and low-level languages. This means it is both human-readable and capable of directly interacting with the hardware, making programs fast and efficient.

1.1.1 Key Features of C Programming

1. Mid-level Language

C is often referred to as a mid-level programming language. This means it bridges the gap between low-level assembly languages and high-level languages.

2. Portability

C programs are highly portable. A program written in C on one machine can be compiled and run on another machine with little to no modification. This is because C compilers are available for almost all operating systems and hardware platforms.

3. Structured Language

C is a structured programming language. This means it encourages the use of functions, loops (for, while), and conditional statements (if, else, switch) to create clear, organized, and modular code, reducing the complexity of large programs.

4. Rich Set of Libraries

C has a rich set of built-in functions and libraries, such as the Standard Input/Output library (stdio.h), which provides functions for file handling and console I/O. These libraries simplify common programming tasks.

5. Speed of Execution

Being a compiled language, C code is translated directly into machine code. This eliminates the need for an interpreter, resulting in very fast execution. This makes C suitable for performance-critical applications, such as game engines and operating systems.

6. Dynamic Memory Allocation

C provides functions like malloc() and calloc() for dynamic memory allocation. This allows programmers to allocate memory during runtime, providing flexibility in handling data structures of varying sizes, such as linked lists and trees.

7. Extensibility

C is highly extensible. New libraries and functions can be easily created and added to the language. It is often used as a base language to develop other programming languages and software, serving as a "building block" for more complex systems.

8. Case-Sensitive

Case sensitivity in C is a fundamental rule that dictates how the compiler interprets identifiers. It means how the language distinguishes between uppercase and lowercase letters. Consequently, an identifier written with different capitalization is treated as a completely separate and unique entity. This strict rule requires you to be consistent with the capitalization of every identifier throughout your program. It's a powerful feature that provides more freedom in naming conventions but also requires precision and attention to detail to avoid common typographical errors.

1.1.2 Structure of the C program

The structure of a C program is as follows:

1. Header files - In C programming, the first component of a program is often the inclusion of header files. A header file is a file with the extension **.h** that contains C function declarations and macro definitions to be shared among multiple source files. All lines starting with the # symbols are processed by the *preprocessor*. A preprocessor is a program which is invoked by the compiler before actual compilation begins. For example, in the statement #include <stdio.h>, the preprocessor inserts the preprocessed code from the **stdio.h** file into our program. These .h files are called *header files* in C.

2. main()- In a C program, the main() function serves as the entry point where program execution begins, starting from the first statement within it. The empty parentheses () after main indicate that the function does not accept any parameters. The keyword **int** placed before main specifies that the function returns an integer value, which is generally used to convey the program's termination status to the operating system. By convention, returning zero usually signifies successful execution, while returning a non-zero value indicates an error or abnormal termination. Thus, the main() function not only initiates the program's execution but also communicates its outcome upon completion.

3. Body part - In a C program, the body of the main() function is the section enclosed within curly brackets {} where all the instructions to be executed are written. This is the part of the program that actually performs the required tasks, which may include operations such as calculations, data manipulation, searching, sorting, displaying messages, or any other logic needed. The opening curly bracket { indicates the start of the function's body, and the closing curly bracket } marks its end. Every function in C, including main(), must have its body enclosed within these brackets to define its scope and structure clearly.

In C programming, scanf() and printf() are standard input/output functions provided by the stdio.h header file, used to read and display data.

- printf() is used for output. It displays text or data stored within the variables to the screen. It uses format specifiers (like %d for integers, %c for characters, %f for floats, %s for strings) to define the type of data to print. Its full form is print formatted, meaning it allows both plain text and variables to be displayed in a controlled (formatted) way.
- scanf() is used for input. It reads values entered by the user from standard input (keyboard) and stores them in variables. It also uses format specifiers and requires the address of variables (using & for most data types) so that the function can store the input directly into memory.

```
Syntax:
printf("control string", argument1, argument2, ...);
scanf("format_specifiers", &variable1, &variable2, ...);
```

Example : To print a message :Welcome to C programming".

```
#include <stdio.h>  // Header file for input-output functions
int main()
{
  printf("Welcome to C programming"); // Prints the message on the screen
   return 0; // Indicates successful program execution
}
```

1.1.3 Keywords in C

In the C programming language, *keywords* are special reserved words that have pre-defined meanings and purposes. They form the vocabulary of the language and are used to write programs in a structured and meaningful way. Keywords cannot be used as identifiers such as variable names, function names, or labels, because the compiler treats them as commands or instructions. The standard keywords in C programming are shown in the figure below:

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

Fig. 1.1.1 The Standard 32 Keywords in C

1.1.4 Data Types in C

In C programming, every variable must be declared with a specific **data type**, which determines the kind of data it can hold, how much memory it occupies, and the range of values it can store. A data type also defines the operations that can be performed on that variable and how the stored data will be interpreted by the compiler. For example, an **int** variable is used to store whole numbers without any fractional part, while a **char** variable stores single characters such as letters or symbols, represented internally by their ASCII values. Similarly, a **float** variable is designed to hold decimal numbers with single precision, whereas a **double** variable holds decimal numbers with double precision, providing greater accuracy and a wider range. Choosing the correct data type is crucial for efficient memory usage and accurate program execution. If the wrong data type is used, it can lead to memory wastage, data loss, or unexpected results during program execution. In short, the data type is like a blueprint that tells the compiler both the nature of the data and how it should be processed.

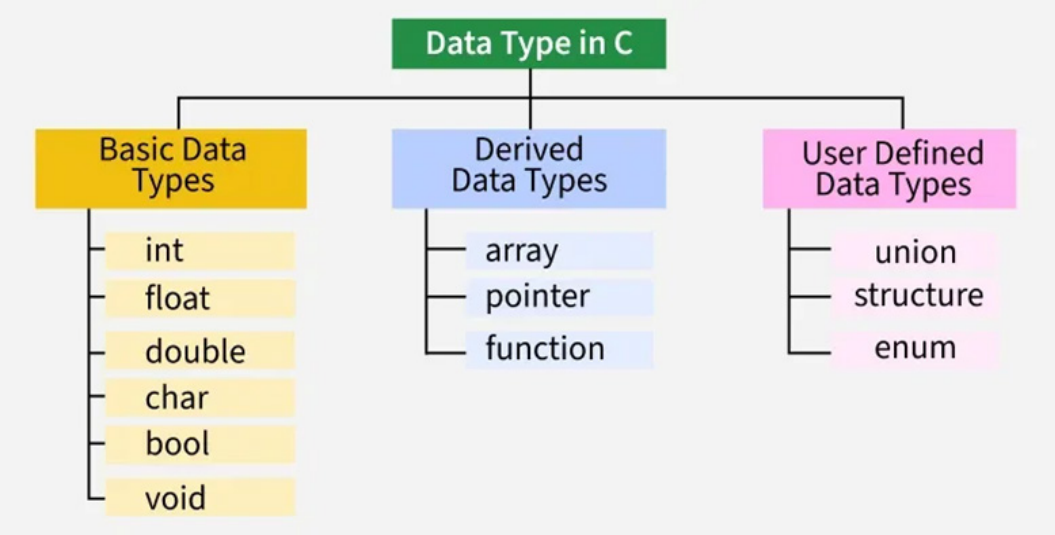


Fig.1.1.2 Data types in C

1. Integer Data Type

The integer (int) data type is used to store whole numbers, meaning numbers without any fractional or decimal part. This includes positive numbers, negative numbers, and zero. The size of an int typically depends on the system and compiler, but on most modern systems it occupies 4 bytes (32 bits), allowing it to store values in the range -2,147,483,648 to 2,147,483,647 for a signed integer.

One of the strengths of the int data type in C is that it can represent numbers in different number systems:

- **Decimal (Base 10)** – The standard form we use in everyday life.
- **Octal (Base 8)** – Represented in C by prefixing the number with 0.
- **Hexadecimal (Base 16)** – Represented in C by prefixing the number with 0x or 0X.

C also supports signed and unsigned integers:

- **Signed int** (default) can store both negative and positive numbers.
- **Unsigned int** can store only non-negative numbers, but its range is doubled on the positive side.

In C programming, the control string for the integer data type refers to the format specifier used inside functions like printf() and scanf() to tell the compiler how to interpret and display or read integer values. For a signed integer, the control string is %d when printing or reading values, while for an unsigned integer, it is %u. These specifiers can also include field width, flags, and length modifiers for more control over the output. For example, %5d prints the integer in a field of 5 spaces (right-aligned), and %05d pads it with leading zeros. Length modifiers like %ld (long int) or %hd (short int) are used for different integer sizes. When using scanf() for integers, the same control strings (%d, %u, %ld, etc.) are used, but the address-of operator (&) must be applied to

the variable so the function can store the input directly into memory.

```
#include <stdio.h>
int main() {
    int age = 25;            // signed integer
    unsigned int rollNo = 101; // unsigned integer

   printf("Age: %d\n", age);
    printf("Roll Number: %u\n", rollNo);
     return 0;
}
```

2. Character Data Type

In C programming, the character (char) data type is used to store a single character, such as a letter, digit, or special symbol. It is the most basic data type for handling textual data and typically occupies 1 byte (8 bits) of memory in almost all compilers. Since a byte can represent 256 different values, a char can store either signed values in the range -128 to 127 or unsigned values in the range 0 to 255, depending on how it is declared. Internally, characters are stored as their corresponding ASCII values, meaning a char is essentially stored as an integer code representing the character. The format specifier %c is used in printf() or scanf() to display or read character values.

```
Example:
#include <stdio.h>
int main()
 {
    char letter = 'A';  // storing a single character
    printf("The character is: %c\n", letter);
   printf("The ASCII value is: %d\n", letter); // shows numeric representation
    return 0;
}
```

3. Float Data Type

In C programming, the float data type is used to store single-precision floating-point numbers, which are numbers that have a fractional part or can be represented in scientific (exponential) notation. A float typically requires 4 bytes (32 bits) of memory and follows the IEEE 754 standard for single-precision representation. This allows it to store values in the approximate range 1.2×10^{-38} to 3.4×10^{38}, with a precision of about 6 to 7 decimal places. Floating-point numbers are used when more precision is required than integers can provide, especially for scientific calculations, measurements, or financial data where decimals are necessary. The format specifier %f is used in printf() to display a float value, and the same %f is used in scanf() to read a float value from the user (with the address-of operator &). Floats can also be displayed in exponential form

using %e or %E.

```
#include <stdio.h>
int main() {
    float price = 99.75;          // float variable
    printf("Price: %f\n", price); // default decimal format
    printf("Price (2 decimals): %.2f\n", price); // fixed precision
    printf("Price (exponential): %e\n", price);  // exponential format
    return 0;
}
```

4. Double Data Type

In C programming, the double data type is used to store double-precision floating-point numbers, which means it can represent decimal numbers with significantly greater precision and range than the float type. A double typically occupies 8 bytes (64 bits) of memory and also follows the IEEE 754 standard, allowing it to store values approximately in the range 1.7×10^{-308} to 1.7×10^{308}. It provides about 15 to 17 significant decimal digits of precision, making it ideal for scientific, engineering, and financial calculations where high accuracy is required. The format specifier %lf is used with scanf() to read a double value, but with printf(), both float and double use %f for output (although %lf can also be used for clarity in some compilers). Doubles can also be displayed in exponential notation using %e or %E, and precision can be controlled using format modifiers like %.10lf for fixed decimal places.

```
#include <stdio.h>
int main() {
    double distance = 1234567.891234567;
    printf("Distance: %lf\n", distance);        // default precision
    printf("Distance (10 decimals): %.10lf\n", distance); // high precision
    printf("Distance (exponential): %e\n", distance);     // exponential format
    return 0;
}
```

5. Void Data Type

In C programming, the void data type is a special type that signifies the absence of a value or no specific data type. You cannot declare normal variables of type void because it cannot store any data, but it plays an important role in three main situations: as a function return type when a function does not return any value (e.g., void display(void)),

as a function parameter type to explicitly indicate that no arguments are passed, and as a generic pointer type (void*) that can hold the address of any data type but must be typecast before use. Void functions are often used for tasks like printing messages or performing actions without producing a result, while void pointers are useful in creating flexible and reusable code that works with different data types. For example, a void pointer can store the address of an integer, float, or any other type, making void an essential tool for writing generic and type-independent programs in C.

Table 1.1.1: Datatypes in C

Data Type	Size (in Bytes)
short	2
int	4
long	4 or 8
long long	8
float	4
double	8
long double	10, 12, or 16
char	1
void	0

1.1.5 Variables in C

A **variable** in C is a named memory location used to store data during program execution. It allows a program to store, retrieve, and manipulate values dynamically. The value of a variable can change during the program's lifetime, hence the name “variable”.

In C, variable names follow certain rules:

1. Can contain letters (A–Z, a–z), digits (0–9), and underscores (_).
2. Must start with a letter or underscore (cannot start with a digit).
3. Case-sensitive (Count and count are different variables).
4. Cannot use C keywords (e.g., int, while, return).
5. No spaces or special symbols allowed except the underscore (_).

Example: marks, total_amount, _value, Age1

```
Declaring Variables
data_type variable_name;

Example:
int age;
float salary;
char grade;
Variable Initialization examples:
int age = 25;
float price = 99.50;
char grade = 'A';
```

1.1.6 Constants in C

In C programming, **constants** are fixed values that do not change during the execution of a program. Unlike variables, whose values can be modified, constants remain the same once they are defined. Constants make programs more reliable, easier to understand, and maintainable, as they represent values that have a fixed meaning in the program's logic. The following are some different types of constants in C:

1. Integer Constants – Whole numbers without a fractional part. Example: 10, -25, 0.

2. Floating-point Constants – Numbers with decimal or exponential notation. Example: 3.14, -0.75, 2.5e3.

3. Character Constants – A single character enclosed in single quotes. Example: 'A', '9', '\n'.

4. String Constants – A sequence of characters enclosed in double quotes. Example: "Hello", "C Programming".

5. Enumeration Constants – User-defined constants created using the enum keyword. Example: enum week {MON, TUE, WED, THU, FRI, SAT, SUN};

C provides two main ways to define constants:

- **Using #define Preprocessor Directive**

```
#define PI 3.14159
#define MAX 100
```

- **Using const Keyword**

```
const int MAX = 100;
const float PI = 3.14159;
```

1.1.7 Operators in C

In C programming, operators are special symbols that perform specific operations on one or more operands (variables or values). They are the building blocks for creating expressions and making decisions in a program. In C programming, operators can be classified into three types based on the number of operands they work on. Unary operators operate on a single operand and are used to perform operations such as increment (++) and decrement (--). Binary operators work on two operands and include operations like addition (+), subtraction (-), and multiplication (*). Ternary operators operate on three operands, with the most common example being the conditional operator (? :), which is used to evaluate a condition and return one of two possible values based on whether the condition is true or false. In C language, there are various built-in operators, which can be broadly classified into seven types based on their functionality.

1. Arithmetic Operators

Arithmetic Operators in C are fundamental tools used to perform mathematical calculations on numerical data, either with constants, variables, or expressions. They work on operands and produce a single numeric result. These operators can handle both integer and floating-point data types, and their behavior may vary slightly depending on the type of operands.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Fig. 1.1.3 Arithmetic Operators in C

2. Assignment Operators

Assignment operators in C are used to assign values to variables. The most common assignment operator is the simple equal sign (=), which stores the value on its right-hand side into the variable on its left-hand side. However, C also provides several *compound assignment operators* that combine an arithmetic or bitwise operation with

assignment, allowing for shorter and more efficient code. For example, the operator += adds the value on the right to the existing variable and then assigns the result back to that variable. This means that x += 5; is equivalent to x = x + 5.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Fig.1.1.4 Assignment operators in C

3. Comparison operators or Relational operators

Comparison operators in C are used to evaluate the relationship between two values or expressions. They help determine whether one value is equal to, greater than, or less than another, among other comparisons. This is an essential feature in programming because many logical decisions and control flow structures, such as if, while, and for statements, rely on these evaluations. When a comparison is performed, the result is a Boolean value represented numerically in C: 1 indicates true (the condition is satisfied), and 0 indicates false (the condition is not satisfied). These results can then be used to decide the next course of action in a program.

Operator	Name	Example	Description
==	Equal to	x == y	Returns 1 if the values are equal
!=	Not equal	x != y	Returns 1 if the values are not equal
>	Greater than	x > y	Returns 1 if the first value is greater than the second value
<	Less than	x < y	Returns 1 if the first value is less than the second value
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value

Fig.1.1.5 Relational operators in C

4. Logical Operators

Logical operators in C are used to combine or modify conditions when making decisions in a program. They allow you to test multiple expressions at once and determine whether the final result is true (1) or false (0). This is particularly useful in scenarios where a single condition is not enough to decide the flow of execution.

Operator	Name	Example	Description
&&	AND	x < 5 && x < 10	Returns 1 if both statements are true
||	OR	x < 5 || x < 4	Returns 1 if one of the statements is true
!	NOT	!(x < 5 && x < 10)	Reverse the result, returns 0 if the result is 1

Fig.1.1.6 Logical operators in C

5. Bitwise Operators

Bitwise operators in C work directly on the binary representation of integers (bit by bit). Instead of working on whole numbers like arithmetic operators, they manipulate individual bits (0s and 1s) in the number's binary form. These operators are fast and memory-efficient because they use low-level machine instructions. They are often used in systems programming, device drivers, cryptography, and network protocols.

Table 1.1.2: Bitwise operators in C

Operator	Name	Description	Example	Result (Binary)	Result (Decimal)
&	AND	Sets each bit to 1 if both corresponding bits are 1.	5 & 3	1	1
|	OR	Sets each bit to 1 if at least one of the corresponding bits is 1.	5 | 3	111	7
^	XOR	Sets each bit to 1 if only one of the corresponding bits is 1.	5 ^ 3	110	6
~	NOT	Inverts all the bits of the operand.	~5	11111010	-6
<<	Left Shift	Shifts bits to the left, filling new bits with zeros.	5 << 1	1010	10
>>	Right Shift	Shifts bits to the right. Behavior depends on the sign.	10 >> 1	101	5

6. Conditional Operator (?:)

The conditional operator is also called the ternary operator because it requires three operands. It is a shorthand way of writing an if-else statement.

Syntax:

condition ? expression_if_true : expression_if_false;

Example:

```
max = (a > b) ? a : b;   // Check if a > b
```

7. Other Operators

Apart from arithmetic, relational, logical, bitwise, and assignment operators, C also provides some additional operators for specific purposes. The most commonly used are

sizeof and the comma operator.

- sizeof Operator- The sizeof operator is widely used in C. It is a compile-time unary operator used to determine the size (in bytes) of its operand. The result returned by sizeof is of an unsigned integral type, typically size_t.

Syntax:

sizeof (operand)

- Comma Operator (,)- The comma operator is a binary operator in C. It evaluates the first operand, discards the result, then evaluates the second operand and returns the value and type of the second operand. It has the lowest precedence among all C operators.

Syntax:

operand1 , operand2

	Operator Precedence
1	! Logical not (Highest)
2	( ) Parenthesis
3	*, /, %
4	+, -
5	>, >=, <, <=
6	==, !=
7	&& (AND)
8	|| (OR)
9	= (Lowest)

Fig. 1.1.7 Operator Precedence in C

Example:

```
//Write a C program that reads two integers from the user and prints their sum.

#include <stdio.h>

int main() {
   int num1, num2, sum;
   printf("Enter first number: ");
   scanf("%d", &num1);
   printf("Enter second number: ");
   scanf("%d", &num2);

   sum = num1 + num2;
   printf("Sum = %d\n", sum);
   return 0;
}
```

```
// To read the side of a square and print its area and perimeter.

#include <stdio.h>
int main()
{
   float side, area, perimeter;

   printf("Enter the side of the square: ");
   scanf("%f", &side);

   area = side * side;
   perimeter = 4 * side;

   printf("Area of Square = %.2f\n", area);
   printf("Perimeter of Square = %.2f\n", perimeter);

   return 0;
}
```


1.1.8 Comments in C

In C, comments are portions of text in the code that the compiler ignores. They are used to explain the code, make it more readable, and help in documentation. C supports two types of comments:

- **Single-line comments -** Start with // and continue until the end of the line.
- **Multi-line comments**- Start with /* and end with */. It is used for longer explanations or temporarily disabling blocks of code.

Summarised Overview

The C programming language is a powerful and widely used mid-level programming language developed by Dennis Ritchie at Bell Labs for developing the UNIX operating system. It combines the efficiency of low-level programming with the simplicity of high-level languages, making it suitable for system programming, application development, and embedded systems. C is known for its portability, fast execution, structured programming approach, rich libraries, and support for dynamic memory allocation. A C program mainly consists of header files, the main() function, and program statements using input/output functions like printf() and scanf(). The language provides different data types such as int, char, float, double, and void, along with variables and constants for storing data. C also supports various operators including arithmetic, relational, logical, assignment, bitwise, and conditional operators, enabling efficient problem solving and program development.

Assignments

1. Explain the key features of the C programming language?
2. What are the different data types available in C?
3. Discuss the difference between variables and constants in C programming.
4. Explain how operators are classified in C. Discuss each category with suitable examples.
5. What are the rules for naming variables in C?
6. Write a C program to read the length and breadth of a rectangle and print the area and perimeter of the rectangle.

Reference

1. Rajaram, M. (2024). *Computer Programming with C*. Pearson India.
2. Kanetkar, Y. (2021). *Let Us C* (18th ed.). BPB Publications.
3. Balagurusamy, E. (2017). *Programming in ANSI C* (7th ed.). McGraw-Hill Education.
4. Gottfried, B. S. (2017). *Programming with C* (3rd ed.). McGraw-Hill Education.

Suggested Reading

1. Robinson, V., & Chandran, K. S. (2025). *C Programming: Mastering the Basics (2024 Scheme)*. Code Academy.

2. Venugopal, K. R., & Prasad, S. R. (2019). *Mastering C* (3rd ed.). Tata McGraw-Hill Education.

3. Schildt, H. (2018). *C: The complete reference* (4th ed.). McGraw-Hill Education.

4. King, K. N. (2008). *C programming: A modern approach* (2nd ed.). W. W. Norton & Company.

5. https://www.geeksforgeeks.org/c/operator-precedence-and-associativity-in-c/

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT

Control Structures and Arrays

Learning Outcomes

- recall different types of control structures (sequence, selection, iteration) and identify their syntax in C programming.
- explain the working of loops (for, while, do-while) and classify problems that can be solved using iterative statements.
- apply array concepts to solve problems involving storage and retrieval of multiple values efficiently.
- differentiate between built-in and user-defined functions and illustrate their usage in modular programming.
- construct string-handling programs by using standard library functions to manipulate text data.

Background

Programming requires clear ways to control the flow of execution in order to solve problems efficiently. Control structures provide this capability by allowing decisions, repetitions, and branching within a program. Loops are a type of control structure that enable repeated execution of a block of code until a condition is met. Arrays are used to store multiple values of the same type in a single variable, making data management and processing much easier. Functions promote modularity by breaking large problems into smaller, reusable blocks of code. They help improve readability, maintainability, and reduce redundancy in programs. Strings are a special kind of array that deal with characters and are essential for handling text-based data. Together, arrays and strings form the foundation of data storage

and manipulation in many applications. Understanding loops, arrays, functions, and strings equips students to design structured, efficient, and logical solutions. Mastering these concepts lays the groundwork for advanced programming and problem-solving in computer science. Control structures, loops, arrays, functions, and strings form the foundation of structured programming. Control structures help in decision-making and guiding the flow of execution, while loops allow repetition of tasks until certain conditions are met. Arrays and strings provide ways to store and manage collections of data, whether numbers or characters, making data handling more efficient. Functions promote modular programming by breaking complex problems into smaller reusable blocks, improving readability and reusability of code.

Keywords

Control Structures, if statement, if-else, Nested if-else, switch-case, Looping, Arrays, Strings, User defined Functions

Discussion

1.2.1 Control Structures

In our daily lives, we frequently encounter situations that require making choices from the options available. For example, when preparing a cup of tea, you may first check whether there is milk in the refrigerator. If milk is not available, it goes onto your shopping list; if it is, you proceed to make the tea. This is a simple example of decision-making, where an action is determined by a specific condition. In the same way, programming uses *selection* to decide between different courses of action based on given criteria. This section discusses the selection control structures provided in the C programming language.

1.2.2 The simple if statement

The if statement is a fundamental tool for decision-making in programming. It is used to control the flow of execution depending on whether a certain condition is met. An if statement generally contains a condition that is evaluated before the block of code runs. The statements inside the block are executed only when the condition is true. The basic syntax of the if statement is presented below:

The syntax of the if statement for single-line statement is :

```
if(condition)
        Statement ;
```

The syntax of if statement for multi-line statements is :

```
if(condition)
{
        Statement1 ;
        Statement2 ;
            .
            .
            .
            .
       StatementN ;
}
```

If we want to display the positive difference between two numbers, num1 and num2, we need to adjust our method. The program shown below performs the calculation by subtracting the smaller number from the larger one, making sure the result is always positive. The choice of operation depends on the values of num1 and num2.

```
/* program to print the positive difference between two numbers*/
            #include <stdio.h>
            #include <conio.h>
            void main()
            {
       int num1, num2, diff;
       printf("Enter the first number\t:\t");
       scanf("%d",&num1);
       printf("Enter the second number\t:\t");
       scanf("%d", & num2);
       if(num1 > num2)
              diff = num1 - num2
 if(num2 > num1)
          diff = num2 - num1
    printf("The positive difference between %d and %d is %d",num1, num2, diff);
    getch();
                  }
```

Output:

```
Enter the first number  :  5
Enter the second number  :  9
The positive difference between 5 and 9 is 4
```

In the program above, two integer variables, num1 and num2, are declared, and their values are entered by the user. The program then determines which number is larger and calculates the difference between them.

Let's consider another program:

```
/* program to check voting eligibility*/
#include <stdio.h>
#include <conio.h>
void main()
{
int age;
printf("Enter your age\t:\t");
scanf("%d", &age);
if(age >= 18)
printf("Eligible for voting");
if (age<18)
printf("Not eligible for voting");
getch();
}
```

Output:

```
Enter your age  :      45
Eligible for voting

Enter your age  :      12
Not eligible for voting
```

1.2.3 The if-else statement

The if-else statement is an extension of the if statement that provides two alternative execution paths. The condition decides which path will be followed. The general syntax

of the if-else statement is shown below:

The syntax of if statement for single-line statement is :

```
if(condition)
                    Statement ;
else
                    Statement ;
```

The syntax of if statement or multi-line statements is :

```
if(condition)
  {
                    Statement1 ;
                    Statement2 ;
                        .
                        .
                        .
                    StatementN ;
  }
  else
  {
                    Statement1 ;
                    Statement2 ;
                        .
                        .
                        .
                    StatementN ;
  }
```

Let us consider the flowchart to understand program flow in if-else statement

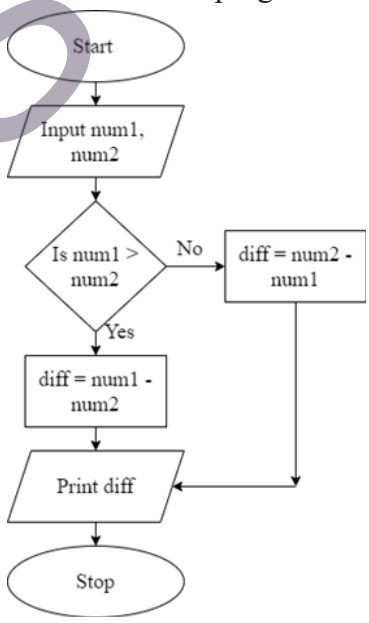


Figure 1.2.1 if-else flowchart

Let us modify the above program with an if-else statement

```
                        /* program to check voting eligibility*/
                        #include <stdio.h>
                        #include <conio.h>
                        void main()
                        {
int age;
printf("Enter your age\t:\t");
scanf("%d", &age);
if(age >= 18)
        printf("Eligible for voting");
else
        printf("Not eligible for voting");
getch();
                                }
```

1.2.4 Nested if-else statement

Nested if-else statements are applied when decisions must be made at multiple levels. In this approach, one if-else block is placed inside another, which is known as nesting. The following program demonstrates how a nested if-else structure works in practice.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num=1;
if(num<10)
{
if(num==1)
{
printf("The value is:%d\n",num);
}
else
{
printf("The value is greater than 1");
}
 }
 else
 {
printf("The value is greater than 10");
 }
getch();
}
```

```
Output
The value is: 1
```

The program above shows how nested if-else statements can be used to check whether a number is less than or greater than 10 and display the result. At the beginning, the variable num is assigned the value 1. The outer if condition tests if the number is less than 10. When this condition is true, the program moves to the inner block. Inside this block, another condition verifies whether num is equal to 1. If the condition is true, the inner if block executes; otherwise, the else block runs. Since the condition is met in this case, the if block executes and displays the output.

1.2.5 The if-else-if ladder

Sometimes a program requires checking several conditions that can lead to different results. In such scenarios, the if-else-if structure is helpful as it allows multiple conditions to be connected in sequence. The general syntax of the if-else-if construct is shown below:

```
if( condition 1)
{
statements;
}
else if( condition 2)
{
statements;
}
else if(condition N)
{
statements;
}
else
{
statements;
}
```

The number of else if statements used depends on how many conditions must be evaluated. If the first condition is false, the program checks the next one, and this process continues until a true condition is found. Once a condition evaluates to true, its corresponding block of code executes, and the entire if-else-if structure terminates.

To illustrate the concept more clearly, let's examine a sample C program.

```
#include<stdio.h>
int main()
{
    int marks=83;
    if(marks>75){
        printf("First class");
    }
    else if(marks>65){
        printf("Second class");
    }
    else if(marks>55){
        printf("Third class");
    }
    else{
        printf("Fourth class");
    }
    return 0;
}
```

In this program, a variable called mark is initialized, and multiple conditions are tested using the else-if ladder. The value of mark is first compared with the initial condition; if it is true, the related message is displayed. If the condition is false, the program checks the next one in sequence. This process continues until a matching condition is found. If none of the conditions are true, control shifts to the final else statement, which is then executed.

1.2.6 The switch-case statement

In everyday life, we often come across situations where we must choose from several alternatives rather than just two. For example, selecting a school, booking a hotel, or deciding on a career may involve many options. In the same way, C programming sometimes requires decisions that go beyond simple yes or no choices. To handle such situations effectively, C provides a special control structure.

The switch statement, also called the switch case default construct, allows a program to select one action from multiple possibilities. The general syntax of the switch case statement is shown below.

```
switch( expression )
{
case value-1:
statement(s);
break;
case value-2:
statement(s);
break;
case value-n:
statement(s);
break;
default:
statement(s);
break;
}
```

Rules for using the switch statement in C:

- The expression in a switch must be of integer or character type.
- Case labels (e.g., 1, 2, n) represent different options. Each label must be unique, since duplicate values will cause errors during execution.
- Every case label should end with a colon (:), and each is associated with its own block of code.
- A break statement is usually placed at the end of each case to stop the program from executing the following cases. Without it, execution continues until the end of the switch block.
- The default case is optional. It executes only when none of the case labels match the expression. If all possible cases are already handled, the default case can be left out.

The control flow that happens in the switch case is shown below:

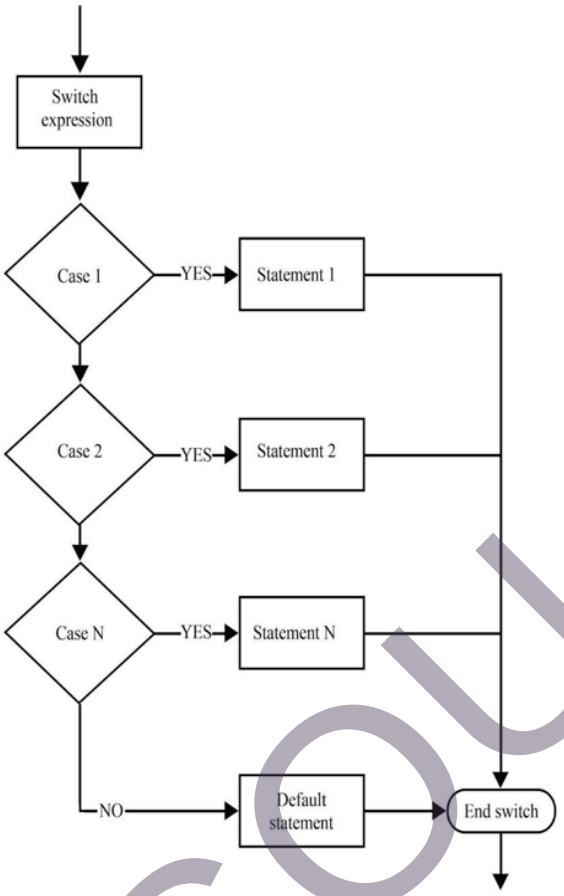


Fig. 1.2.2 Flowchart representation of Switch Statement

Let us understand the switch-case using an example

```
//Print day in a week
#include <stdio.h>
#include <conio.h>
void  main() {
int day_no;
printf("Enter a Day Number\t:\t");
scanf("%d", &day_no);
switch (day_no) {
case 1:
printf("Sunday");
break;
case 2:
printf("Monday");
break;
case 3:
```

```
printf("Tuesday");
break;
case 4:
printf("Wednesday");
break;
case 5:
printf("Thursday");
break;
case 6:
printf("Friday");
break;
case 7:
printf("Saturday");
break;
default:
printf("Enter a valid day number from 1 to 7");
break
}
getch();
}
```

Output

```
Enter a Day Number      :       1
Sunday

Enter a Day Number      :       12
Enter a valid day number from 1 to 7
```

The switch statement checks the value stored in the variable day_no and executes the block of code that matches the corresponding case. In this example, since day_no is assigned the value 1, the program runs the code under case 1. If the input does not match any of the given cases, the default case is executed. After running the matching case (or the default), the control exits the switch block, and the program displays the output successfully.

1.2.7 Looping

In many situations, certain actions need to be performed repeatedly. Take the example of hammering a nail: each time you strike, you check, “Is the nail in?” If the answer is no, you hit it again. This continues until the answer becomes yes, after which you stop. Such a process of repeating actions is called a loop or iteration. In programming, loops allow repetitive tasks to be handled efficiently without writing the same instructions again and again.

To understand this concept better, let's look at an example program.

```
                        / A simple program to display the first five even numbers /

#include <stdio.h>
#include <conio.h>

void main()
{
   printf("0\n");
   printf("2\n");
   printf("4\n");
   printf("6\n");
   printf("8\n");
}
```

This program uses a series of printf statements to display the first five even numbers starting from 0. Each number is printed on a new line.

Suppose we need to print 1,000 or even 100,000 even numbers. Writing thousands of separate printf statements would be both inefficient and impractical. A far better approach is to use a loop (iteration), which can handle the task efficiently with just a few lines of code.

A more efficient method can be achieved using a simple algorithm:

1. Declare a variable, for example, count.
2. Initialize count to 0.
3. Check whether count is divisible by 2 (i.e., remainder equals 0).
4. If the condition is true, display the value of count.
5. Increase count by 1.
6. Repeat steps 3 to 5 until count reaches 100,000.

In programming, looping structures are used to repeatedly execute a set of instructions based on a specific condition. The statements continue to run as long as the condition remains true, and the condition is usually controlled by a variable called the loop control variable. When the condition becomes false, the loop terminates. It is important to design loops carefully so that the condition eventually fails; otherwise, the program may enter an infinite loop and even crash.

In C, the three main types of loops are: for, while, and do-while.

1.2.8 The “for” loop

The for loop is used to execute a set of instructions repeatedly for a defined range or sequence of values. Each value in that range is handled one by one during the loop. These values may be numbers or elements from data types such as strings or arrays.

During each iteration, the control variable checks if all the values in the range have been processed. The loop continues to run until every element has been covered. After the loop finishes, control shifts to the statement that follows it. In most cases, the total number of iterations in a for loop is known in advance. A flowchart (see Fig. 1.2.3) visually explains how the for loop works.

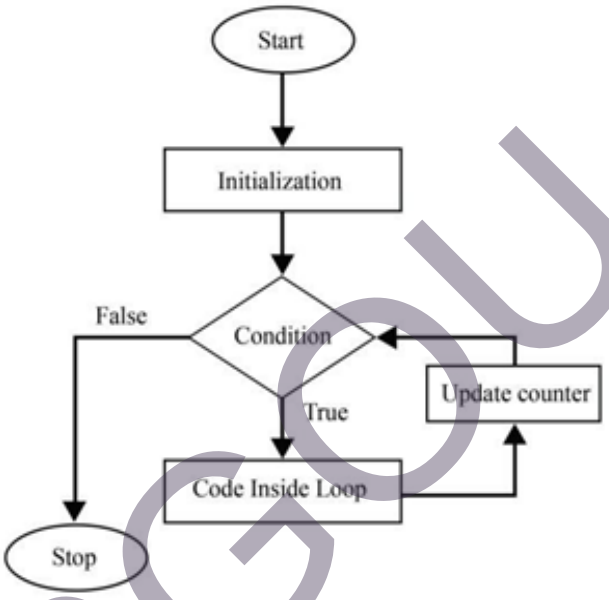


Fig. 1.2.3 Flowchart of a Loop Execution

The syntax of for loop is

```
for (initializationStatement; testExpression; updateStatement)
{
   // statements inside the body of the loop
}
```

- The initialStatement in a for loop is executed just once at the beginning.
- The testExpression is a condition that evaluates to either true or false; it checks the loop counter against a specified limit after every iteration and stops the loop when it evaluates to false.
- The updateStatement adjusts the counter value either incrementing or decrementing it by a defined amount after each loop cycle.

Let us consider a C program to understand the for loop better

```
#include<stdio.h>
int main()
{
int number;
for(number=1;number<=10;number
++) //for loop to print 1-10 nos
        {
 printf("%d\n",number);
//to print the number
        }
 return 0;
}
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

The program above displays numbers from 1 to 10 using a for loop. Let's break down how it works. An integer variable is declared to store the values. In the initialization part of the loop, the variable number is set to 1. The loop condition specifies how long the loop should run, and the increment section increases the value after each iteration. Inside the loop body, the current value of number is printed on a new line. With each repetition, the variable increases by one, and this continues until the value reaches 10.

1.2.9 The "while" loops

We already discussed that a for loop is used when the number of repetitions is known in advance. But what if we are unsure how many times a set of statements should run?

Consider a music streaming app: when the repeat option is turned on, a song keeps playing until you choose to stop or skip it. Similarly, a while loop executes a block of code repeatedly as long as its condition remains true. The condition is checked before each iteration, including the very first one. If the condition is false right from the start, the loop's body will not run even once.

The condition is evaluated before every cycle, and the loop continues as long as it remains true. Once it becomes false, the loop stops, and the control moves to the next instruction after the loop.

It is important that the logic inside the while loop eventually makes the condition false. Otherwise, the loop will continue forever, which causes a logical error in the program.

The syntax of while loop is:

```
                    while(condition)
                    {
statement(s);
                    }
```

Let us consider a C program

```
#include<stdio.h>
#include<conio.h>
int main()
{
int num=1;          // initializing the variable
while(num<=10)  //while loop with condition
{
printf("%d ",num);
num++;
//incrementing operation
}
return 0;
}
```

Output

1 2 3 4 5 6 7 8 9 10

1.2.10 The "do-while" loop

Earlier, we saw that a for loop is used when the number of repetitions is known in advance. But what if we don't know how many times a set of statements should be repeated?

Take the example of music streaming apps: when the repeat option is enabled, a song keeps playing until you choose to stop it or switch to another one. Similarly, a while loop repeatedly executes a block of code as long as a given condition remains true. This condition is checked before each execution, even before the very first run. If the condition is false from the beginning, the loop body will not execute at all.

The loop continues running until the condition becomes false, at which point control moves to the statement immediately following the loop. It is important to ensure that the logic inside the loop eventually makes the condition false. Otherwise, the loop will run endlessly, creating an infinite loop, which is a logical error in the program, as shown in the example program.

The syntax of the do-while loop is:

```
do
{
        statement(s);
    } while( condition );
```

Let us consider a C program to understand the concept clearly

```
/* program to print the first n even numbers*/

#include<stdio.h>
int main()
{
 int num=1, limit;   //initializing the variable
 printf("Enter the limit: ");
 scanf("%d",&limit);
 do //do-while loop
          {
  if(num%2==0)
                  {
  printf("%d ",num);
                  }
  num++;   //incrementing operation
  }while(num<=limit);
    return 0;
}
```

Out Put Enter the limit: 10 2 4 6 8 10

In the above program, we display the first n odd numbers up to the limit provided by the user. But how is this sequence generated?

First, a variable num is initialized with the value 1, and another variable limit is declared to store the user's input. The scanf() function is used to read this input and assign it to limit.

A do while loop is then used to control the process. Inside the loop, the program checks if num % 2 is not equal to zero, which means the number is odd. If the condition is true, the value of num is printed. After each iteration, num is incremented by 1.

This cycle continues until num reaches the specified limit. At that point, the loop ends, and the statement after the loop, return 0 in this case, gets executed.

1.2.11 Break, continue and go to

1. Break

The break statement is used to terminate a loop instantly. When the program encounters a break within a loop or a block of code, execution of that block stops, and control shifts

to the statement that follows it. This brings the loop to an end. The break statement is also widely used in switch-case structures to exit from a particular case.

Syntax: break;

Let us understand the control flow of break command using a flow chart (refer Fig 1.2.4)

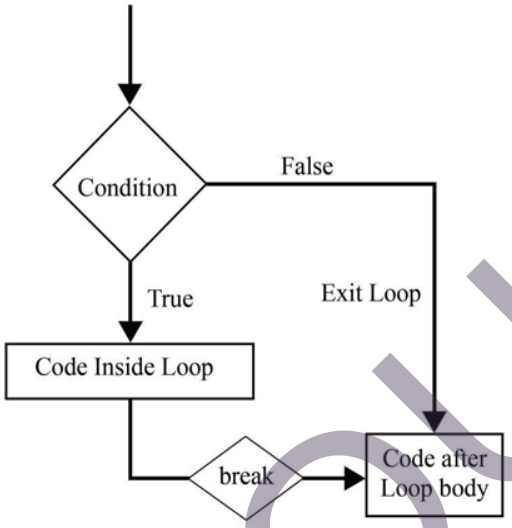


Fig. 1.2.4 Flowchart for Break in Loop

Let us consider a sample program to implement the break in while loop.

```
#include <stdio.h>
int main()
{
int num =0;
while(num<=100)
    {
printf("value of variable num is: %d\n", num);
if (num >= 3)
       {
break;
       }
num++;
    }
 if(num <=3 )
 printf("break command executed");
 return 0;
}
```

Output:

value of variable num is: 0

value of variable num is: 1

value of variable num is: 2

break command executed

2. Continue statement

The continue statement is used within loops. When executed, it bypasses the remaining statements in the current iteration and immediately transfers control back to the start of the loop for the next cycle.

Syntax: continue;

Let us consider a program

```
#include<stdio.h>
int main()
{
int nb = 7;
while (nb > 0)
{
 nb--;
 if (nb == 5)
  continue;
  printf("%d ", nb);
   }

  }
```

Output 6 4 3 2 1 0

In the above output, you could notice that the value 5 is skipped.

3. C - Goto command

When a goto statement is used in a C program, the control transfers directly to the labeled statement mentioned in the goto.

Syntax of the goto statement is.

```
goto label_name;
 ..
statement(s)
 ..
label_name: C-statement(s);
```

The flowchart of the goto statement is shown below as Fig. 1.2.5

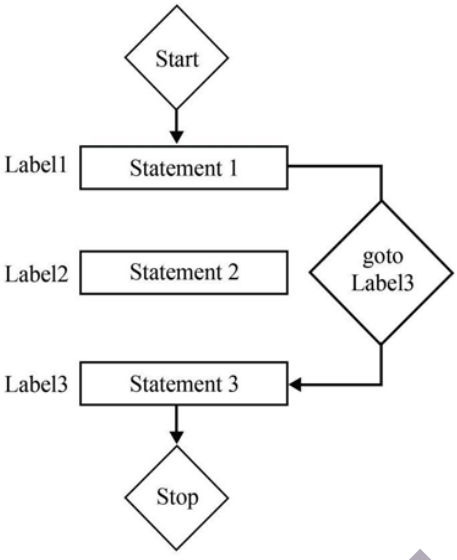


Fig. 1.2.5 Flowchart for goto Statement

Let us understand the goto statement through a c program.

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int sum=0;
  for(int i = 0; i<=10; i++){
  sum = sum+i;
  if(i==5){
  goto addition;
  }
  }
  addition:
  printf("%d", sum);
  getch();
 }
```

Output: 15

The goto statement is an unconditional jump that disrupts the normal sequence of program execution. It allows control to move directly from the current point in the code to a specified label elsewhere in the program. After the jump, execution continues from that new location without returning to the original position.

However, the use of goto is generally discouraged because it:

- May introduce unexpected errors in the program.
- Reduces the readability of code.
- Makes program logic more difficult to understand.

- Complicates the debugging process.
- Can often be replaced with structured statements like break or continue.

1.2.12 Nested loops

A loop inside a loop is called a nested loop. Let's consider a clock which is a perfect example of a nested loop.

Fig. 1.2.6 A clock shown as an example for a nested loop.

Inside a clock, when the seconds hand completes 60 rounds, the minute hand begins its movement. After the minute hand finishes 60 rounds, the hour hand starts moving. In this context, the seconds loop is called the inner loop, while the hours loop is referred to as the outer loop.

Let us consider an example to understand the concept

```
#include <stdio.h>
int main()
{
int i, j;
int table = 2;
int max = 5;
for (i = 1; i <= table; i++)
{ // outer loop
  for (j = 0; j <= max; j++)
{ // inner loop
    printf("%d x %d = %d\n", i, j, i*j);
 }
    printf("\n"); /* blank line between
tables */
}
}
```

```
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

The above program shows nesting of for loops. It can be extended to any level. Similarly you can do nesting of any loops.

1.2.13 Arrays

In C programming, arrays are among the most essential and powerful tools for organizing collections of data. They allow efficient handling of information, whether it's a list of integers, a sequence of characters, or even more complex data structures.

An array in C is a group of variables stored in consecutive memory locations, which can be accessed using a common name along with an index. The elements of an array may belong to basic types such as int, float, char, double, or even user-defined types like structures.

1.2.13.1 C Array Declaration Syntax

In C programming, an array is a collection of elements of the same data type stored in consecutive memory locations. When declaring an array, you need to specify the data type, the array name, and the size (i.e., the number of elements it can store).

General syntax:

```
data_type array_name[array_size];
```

- data_type – defines the type of elements the array will hold (e.g., int, float, char).
- array_name – the identifier used to reference the array.
- array_size – the total number of elements the array can contain (must be a positive integer).

Array Elements in Memory

For example:

int array[5];

This statement instructs the compiler to reserve memory space for 5 integers.

- On most systems, an int occupies 2 bytes (though on some systems it may be 4 bytes).
- Hence, 5 integers × 2 bytes = 10 bytes of memory allocated.

Since no initial values are provided, the array contains garbage values (unpredictable leftover data in memory). Regardless of their values, the elements are stored sequentially in memory, meaning each element is placed right next to the previous one.

Examples of Array Declarations

int numbers[5]; // Array of 5 integers

float prices[10]; // Array of 10 floating-point values

char name[20]; // Character array (string) with space for 20 characters

Note: The array size must be a constant defined at compile time. If the size is not mentioned in the declaration, it should be determined from the initialization.

Let us consider a C program to understand it

```
/* program to find the average of given five numbers*/
#include<stdio.h>
void main()
{
int i, avg, num[5] = { 10,20,32,50,26},sum=0;
for ( i = 0 ; i <5 ; i++ )
sum = sum + num[ i ] ; /* read data from an array*/
avg = sum / 5 ;
printf ( "Average marks = %d\n", avg ) ;
}
```

We use the variable i as a subscript to refer to different elements of the array in the above code. This attribute may have several values, allowing it to apply to the array's various elements in turn. The ability to interpret subscripts with variables is what makes arrays so useful.

Let us see how to enter elements into arrays

```
printf ( "Enter any 5 numbers " ) ;
for ( i = 0 ; i <5 ; i++ )
{
scanf ( "%d", &num[ i ] ) ;
}
```

In the given code example, the for loop repeatedly prompts the user to enter numbers, doing this a total of five times. On the first iteration, when i = 0, the scanf() function stores the entered value in the first element of the array, num[0]. This continues until i reaches the value 5, thereby filling all five elements of the array.

When using scanf(), we apply the address-of operator (&) to the array element num[i], just as we do with ordinary variables (e.g., &rate). This is necessary because scanf() requires the address of the variable (or array element) where the input will be stored, not its current value.

In the next version of the code, the for loop works similarly, but this time, the body of the loop adds each entered number to a variable named sum, which keeps a

running total. Once all five numbers are added, the program divides the total by 5 (the number of students) to calculate the average.

```
for ( i = 0 ; i <5 ; i++ )
sum = sum + num[ i ] ; /* read data from an array*/
avg = sum / 5 ;
printf ( "Average marks = %d\n", avg ) ;
```

1.2.14 Types of Arrays

Arrays are a fundamental data structure in C programming that stores multiple elements of the same type in a contiguous memory block. Based on their dimensions, arrays in C can be categorized into One-dimensional and Multidimensional arrays. Below, we'll explore these types in detail.

1.2.14.1 One Dimensional Array in C

A one-dimensional array is the simplest form of an array, storing elements in a single linear sequence. It can be visualized as a row of elements.

Declaration

data_type array_name[size];

Example: int arr[5]; // Declares an array of size 5

Initialization

int arr[5] = {1, 2, 3, 4, 5};

Accessing Elements

printf("%d", arr[2]); // Accesses the third element, which is 3

Examples for array usage:

1. Storing a list of numbers, such as marks or salaries.
2. Simple linear data handling.

1.2.14.2 Multidimensional Array

Multidimensional arrays are essentially arrays that contain other arrays, making it possible to store data in a table-like or matrix structure. They build upon the idea of one-dimensional arrays by introducing additional dimensions.

A basic example of declaring a multidimensional array is shown below:

```
Datatype arrayName[size1][size2]...[sizeN];
```

The two-dimensional array is the most basic type of multidimensional array. A two-dimensional array is essentially a list of one-dimensional arrays. The following is the syntax for a two-dimensional array:

```
Datatype arrayName[size1][size2]
```

1.2.14.3 2-Dimensional Array in C

A 2D array organizes data in rows and columns, making it ideal for matrix representation. Fig 1.2.7 illustrates the representation of the following 2-D array.

e.g.: int a[2][3]= {{1,2,3},{4,5,6}};

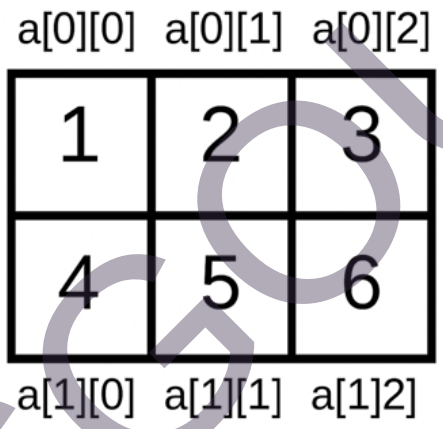


Fig. 1.2.7 2-D Array representation

Declaration

```
data_type array_name[rows][columns];
```

Example

```
int arr[3][4]; // Declares a 2D array with 3 rows and 4 columns
```

Initialization

```
int arr[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Key Points to Remember When Initializing a 2D Array:

- The total elements in the array should not be more than rows × columns.
- If some elements are not specified during initialization, they are automatically set to 0.
- For clarity and readability, it is recommended to use nested braces while assigning values.

Examples of 2D Array in C: Printing a Matrix

```
#include <stdio.h>
 int main() {
 int arr[2][3] = {
 {1, 2, 3},
 {4, 5, 6}
};
 for (int i = 0; i < 2; i++) {
 for (int j = 0; j < 3; j++) {
 printf("%d ", arr[i][j]);
}
  printf("\n");
}
  return 0;
}
```

Output

```
1 2 3
4 5 6
```


1.2.15 Advantages and Disadvantages of Arrays in C

1.2.15.1 Advantages of Arrays in C

- Optimized Memory Utilization: Arrays store elements in consecutive memory locations, minimizing memory overhead.
- Direct Access to Data: Any element can be quickly retrieved using its index.
- Simplified Coding: Loops can be used to process multiple elements, reducing redundant lines of code.

- Contiguous Storage: Since data is stored continuously, access and manipulation become faster.
- Best for Fixed-Size Collections: Arrays are highly effective when the number of elements is predetermined.

1.2.15.2 Disadvantages of Arrays in C

- Fixed Size: The size of an array is specified during its declaration and cannot be changed later, which reduces flexibility.
- Same Data Type: Arrays can hold only elements of a single data type.
- No Automatic Bounds Checking: Accessing elements outside the defined index range may cause errors or unpredictable behavior.
- Insertion and Deletion Overhead: Adding or removing elements is inefficient because it often requires shifting existing elements.
- Memory Usage: If not managed carefully, large arrays may occupy excessive memory.

1.2.16 Strings

In C programming, arrays allow us to group together elements of the same data type. For instance, a set of integers can be stored in an integer array, while a sequence of characters can be stored in a character array. Character arrays are often referred to as strings, and they are widely used in programming to handle text, such as words and sentences.

A string constant in C is essentially a one-dimensional array of characters that ends with a special symbol called the null character ('\0'). For example:

char name[] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' };

In this case, each character takes up 1 byte of memory, and the final character in the array is '\0'. This null character is crucial because it signals the end of the string. It is different from '0':

- The ASCII value of '\0' is 0.
- The ASCII value of '0' (the digit zero) is 48.

The elements of a character array are stored in contiguous memory locations, just like other arrays. See Fig. 1.2.8, The terminating '\0' is essential, as it allows C string-handling functions to correctly identify where the string finishes. Without the null character, the collection of characters is not considered a proper string but merely an array of characters.

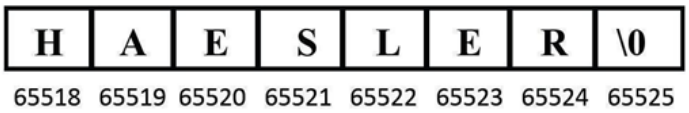


Fig. 1.2.8 String as an Array

In many cases, we work with strings, and C provides a simpler way to initialize them. Instead of listing each character separately, a string can be directly assigned within double quotes.

For example, the string "HAESLER" can be initialized as:

char name[] = "HAESLER";

In this form of declaration, the null terminator (\0) does not need to be explicitly added. The C compiler automatically appends it at the end of the string.

```
/* Program to demonstrate printing of a string */
# include <stdio.h>
int main( )
{
char name[ ] = "Hellions" ;
int i = 0 ;
while ( i <= 7 )
{
printf ( "%c", name[ i ] ) ;
i++ ;
}
printf ( "\n" ) ;
return 0 ;
}
output
Hellions
```


Here in the above program, a character array is initialized, and then printed out the elements of this array iterating through a while loop.

1.2.17 Functions

In C programming, a function is a self-contained block of code designed to perform a particular task. It is written inside curly braces {} and may take inputs (parameters), process them, and optionally return a result.

Functions are highly valuable because they encourage code reusability—you write the logic once and reuse it multiple times by calling the function with different inputs. This makes programs more efficient, organized, and modular.

Instead of duplicating the same logic throughout a program, you can place it in a function and call it whenever required. This not only reduces redundancy but also improves clarity.

Importance of Functions in C

Functions provide several advantages to programmers:

- Reusability – avoid rewriting the same code repeatedly.
- Simplification – break large programs into smaller, manageable units.
- Modularity – focus on one task at a time, making the program well-structured.
- Abstraction – hide internal implementation details from the main program.
- Maintainability – make the code easier to read, debug, and update.

Using functions is similar to building a machine from smaller parts—each part has a specific role, and together they complete a larger task.

Basic Syntax of Functions

The basic syntax of functions in C programming is:

```
return_type function_name(arg1, arg2, … argn)
{
Body of the function //Statements to be processed
}
```

- return_type: Defines the type of value that the function will produce (e.g., int, float). If the function does not return any value, the keyword void is used.
- function_name: The identifier assigned to the function. It must follow the naming conventions of the C language.
- arg1, arg2, ..., argn: Represent the parameters or inputs provided to the function. A function may have no parameters or accept one or more parameters.

The combination of the function name and its parameter list is known as the function signature.

In C programming, functions are generally categorized into two types:

- Built-in functions – predefined functions available in C libraries.
- User-defined functions – functions created by the programmer to perform specific tasks.

1.2.17.1 Built-In Functions

In C programming, built-in functions (also called standard library functions) are predefined functions made available to programmers by default.

These functions work much like departments in an office. For example, if there is an attestation section, anyone needing attestation will directly approach that section without confusion. Similarly, built-in functions are organized to perform specific tasks, and they are grouped together in header files.

Two commonly used built-in functions are printf() and scanf(), which belong to the stdio (standard input/output) library. To use a library function in a program, the corresponding header file must be included at the beginning of the program.

For example, since printf() and scanf() are part of the stdio.h file, this header must be linked before using them. The definitions and properties of these functions are stored inside their respective header files.

Syntax:

```
#include <filename.h>
```

Example:

```
#include <stdio.h>
```

1.2.17.2 User-Defined Functions

In C programming, users can create their own functions to carry out specific tasks. These functions are code blocks written by the programmer to handle particular operations.

Every function generally involves inputs and outputs. It contains a set of statements or instructions that process the input values and produce the desired result. Similar to a complete program, both the input and output can take different forms, such as integers, floating-point numbers, characters, or arrays.

Before writing a user-defined function, it is important to understand some of its fundamental features and characteristics.

A user-defined function has three phases

- function declaration
- function definition

- function call

1. Function Declaration

Just as a variable must be declared before use in a program, a function declaration is required before calling a function.

A function declaration (also called a function prototype) tells the compiler three important details:

- The function name
- The number and type of input parameters (parameter names are optional here)
- The return type of the function

Understanding Parameters

The term parameter refers to the variables specified in a function declaration or definition. Parameters act as placeholders for the values passed to the function during execution.

To illustrate, consider the admission process in a school. The office section handles the processing and prepares a list of admitted students, which is then forwarded to the respective department. In this case, the name list is like a parameter—it carries the required data from one section to another. Similarly, in programming, parameters transfer values into a function.

Example: Function to Add Two Integers

```
int add(int x, int y);
```

This declaration defines a function named add, which accepts two integer parameters (x and y) and returns an integer result.

It can also be written without parameter names:

```
int add(int, int);
```

General Syntax

```
return_type function_name(parameter_list);
```

2. Function Definition

Once a function is declared, its definition must be provided. A function definition contains the actual block of code that performs a specific task.

A function definition consists of two main parts:

- Function Header – includes the function name, return type, and parameter list.
- Function Body – contains the set of instructions that carry out the required operations.

Example:

```
int add(int a, int b)   // Function name: add, Parameters: a and b, Return type: int
{
   int c;
   c = a + b;
   return c;   // Returns the result stored in c
}
```

In this example, the function add() takes two integer inputs (a and b), adds them together, stores the result in the integer variable c, and returns c as the output.

Note: The return type of a function specifies the data type of the value that the function returns.

3. Function Call

After declaring and defining a function, it must be called in order to execute it. A function call is the process of invoking the function to perform its task.

Example:

```
add(5, 7);
```

Here, the values 5 and 7 are passed to the function add(). Inside the function definition, these values are assigned to the parameters a and b. The function then executes, adds the numbers, and returns the result.

Flow of working of a function

```
#include<stdio.h>
int add(int,int);
int add(int a, int b)
{
int c;
c=a+b;
return(c );
}
void main()
```

```
{
int sum;
sum = add(5,7);
printf("Sum is %d",sum);
}
```

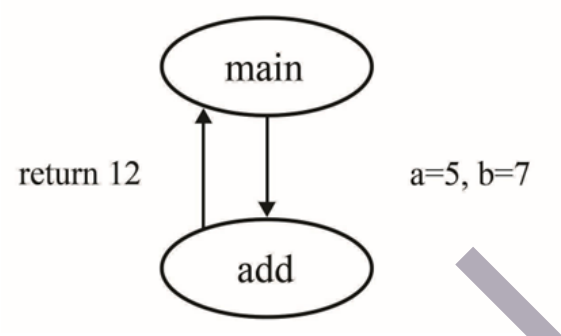


Fig. 1.2.9 Working of function

Function Implementation and Flow

How does a function work? What are its main components, and how do they interact? These aspects can be better understood with the help of Fig 1.2.9.

When a C program begins execution, the compiler starts with the main() function. The flow continues until a function call is encountered (Step 1). At this stage, control shifts to the corresponding function definition (Step 2). The values passed in the function call are assigned to the parameters of the function definition, and the code within the function executes.

After the function finishes its task and reaches the return statement (Step 3), control is handed back to the calling function.

A function may be defined either before or after main(), as long as it is properly declared to ensure correct execution.

1.2.17.3 Examples of functions

The following program illustrates the implementation of function

Example 1: Program to implement a function for addition of 3 intcalculator

```
#include<stdio.h>
int add (int, int ,int);                    //function declaration
int add (int a, int b,int c)                //function definition
{
int d;
d=a+b+c;
return(d);
}
void main ()
{
int w,x,y,z;
printf (“Enter three numbers”);
scanf("%d%d%d",&w,&x,&y);
z=add(w,x,y);                               //function call
printf("\nThe sum is %d",z);
}
```

Output

Enter three numbers 1 2 4
The sum is 7

The above program defines the function add() which adds three integers and returns an integer value to the calling function.

Example 2: Program to implement calculator

```
#include<stdio.h>
float add(float,float);                          //function declaration
float sub(float,float);
float mul(float,float);
float div(float,float);
float add(float num_1, float num_2)              //function definition
{
float ans;
ans= num_1 + num_2;
return(ans);
}
float sub(float num_1, float num_2)
{
float ans2= num_1 -  num_2;
return(ans2);
}
float mul(float num_1, float num_2)
{
```

```
float ans3= num_1 * num_2;
return(ans3);
}
float div(float num_1, float num_2)
{
float ans4= num_1 / num_2;
return(ans4);
}
void main()                    //Program execution begins here
{
float b,c;
float a,s,m,d;
printf("Calculator\n----------\n");
printf("Enter two numbers to perform operations\n");
scanf("%f %f",&b,&c);
a=add(b,c);
printf("The result of addition is %f\n",a);
s=sub(b,c);
printf("The result of subtraction  is %f\n",s);
m=mul(b,c);
printf("The result of multiplication  is %f\n",m);
d=div(b,c);
printf("The result of division  is %f\n",d);
}
```

The code defines an operation for addition, subtraction, multiplication and division on two variables. The functions send two float values to the function definition and execution occurs there. The results are sent back to the called functions.

Output

```
Calculator
----------
Enter two numbers to perform operations
2 3
The result of addition is 5.000000
The result of subtraction  is -1.000000
The result of multiplication  is 6.000000
The result of division  is 0.666667
```

Concept of functions permits to divide a program into simple and smaller tasks. It makes the code to be called many times and implements the code reusability. For example, a function to find the sum of marks can also be used by a program to find the grade of students.

Advantages of using functions

- Complexity of the program gets reduced by division of work.
- Dividing a program into subprograms enables easier error handling.
- A large program being divided into subprograms makes it easy to update.
- Code reusability – Same code segment can be used in different programs.

1.2.18 Nested Functions

Functions defined within other functions are called nested functions.

Syntax:

```
fun1()
    {
        fun2();
    }
```

The concept of nested functions can be explained through an example. Previously, we discussed the function 'add' to sum three integers (Example 1). A program that finds out the average of three numbers has to perform the same sequence of steps additionally and a division operation.

Therefore, the function to find the average can use the function 'add' to calculate the average.

```
int avg()
{
        int k, result;
        k=add();
        result=k/3;
        return(result);
    }
```

The function average() makes a call to the function add() and stores the returned sum in the variable k. This value of k is then divided by 3, and the result is stored in the variable result as the average.

In this way, the add() function is reused within average() to calculate the average of the numbers.

Example: Program to find the average of 5 marks

```
#include<stdio.h>
float avg();
int add();
float avg()                                                    //Nested Function
{
int sum=add();                                    //function call within a function
float avrg=sum/5;
return(avrg);
}
int add()
{
int i,a[5],mark=0;
printf("Compute average marks of 5 subjects\n----------------------------------
\n");
printf("Enter marks of 5 subjects\n");
for(i=0;i<5;i++)
{
scanf("%d",&a[i]);
mark=mark+a[i];
}
printf("Total mark is %d\n",mark);
return(mark);
}
void main()
{
 float result;
result=avg();
printf ("The average of marks is %f", result);
}
```

Output

```
Compute average marks of 5 subjects
-----------------------------------
Enter marks of 5 subjects
20 30 89 90 50
Total mark is 279
The average of marks is 55.000000
```

Summarised Overview

In the C programming language, conditional statements, looping constructs, arrays, functions, and strings are fundamental building blocks of program design. Conditional statements such as if, if-else, nested if, and switch allow decisions to be made based on certain conditions, enabling programs to follow different execution paths. They provide flexibility by controlling which part of the code should run depending on the given situation. Looping statements like for, while, and do-while are used to repeat a block of code until a condition is satisfied, making it easier to handle repetitive tasks such as traversing data, generating sequences, or performing repeated calculations. Arrays in C are collections of elements of the same data type stored in contiguous memory locations, allowing multiple values to be grouped together under a single name. They are especially useful for storing large sets of related data, such as lists of numbers, characters, or marks of students, and can be accessed easily using index values. Functions in C provide modularity by allowing specific tasks to be grouped into reusable code blocks, reducing duplication and improving readability. A function typically involves a declaration, definition, and call, and it may return a value or simply perform an operation. Strings in C are character arrays ending with a null character ('\0') and are widely used for processing text. Standard library functions like strlen(), strcpy(), strcat(), and strcmp() simplify string operations. Together, conditional statements, loops, arrays, functions, and strings create a strong foundation for problem-solving, data handling, and structured programming in C.

Assignments

1. Write an algorithm and C program that takes three numbers as input and determines the largest, justifying the role of conditional statements in your solution.

2. Analyze how nested loops can be used to generate a multiplication table up to 10 × 10. Implement the program and explain the working of both inner and outer loops.

3. Design a program using arrays to store the marks of 10 students in 3

subjects, calculate the average marks of each student, and identify the student with the highest average.

4. Develop a C program that uses functions to compute the factorial of a number. Discuss how modularity and reusability of functions improve the program design.

5. Write a program to count the frequency of each character in a given string. Analyze how arrays and string manipulation functions are used together in the solution.

6. Create a program that accepts a list of integers and sorts them in ascending order using any sorting algorithm (e.g., bubble sort). Compare the time complexity of your algorithm with at least one other sorting technique.

Reference

1. Robinson, V., & Chandran, K. S. (2025). *C Programming: Mastering the Basics (2024 Scheme)*. Code Academy.
2. Venugopal, K. R., & Prasad, S. R. (2019). *Mastering C* (3rd ed.). Tata McGraw-Hill Education.
3. Schildt, H. (2018). *C: The complete reference* (4th ed.). McGraw-Hill Education.
4. King, K. N. (2008). *C programming: A modern approach* (2nd ed.). W. W. Norton & Company.
5. https://www.geeksforgeeks.org/c/operator-precedence-and-associativity-in-c/

Suggested Reading

1. Rajaram, M. (2024). *Computer Programming with C*. Pearson India.
2. Kanetkar, Y. (2021). *Let Us C* (18th ed.). BPB Publications.
3. Balagurusamy, E. (2017). *Programming in ANSI C* (7th ed.). McGraw-Hill Education.
4. Gottfried, B. S. (2017). *Programming with C* (3rd ed.). McGraw-Hill Education.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3
UNIT

Pointers and Structures

Learning Outcomes

After completing this unit, students will be able to:

- summarize how pointers work in C and their significance in memory management.
- discuss the structure and usage of user-defined data types
- implement pointer declarations and initializations for single and multidimensional arrays in C.
- operate on structure members using pointers within C programs.
- examine between normal variables, pointer variables, and structure pointers based on their behavior and usage.

Background

To effectively study the topics of Pointers and Structures in C programming, a learner should possess foundational knowledge of C syntax, including data types, variables, and operators. It is essential to be comfortable with control flow constructs such as if, else, for, and while loops, as well as having a solid grasp of how arrays are declared, stored, and accessed in memory. Additionally, learners should understand how functions work and how arguments are passed by value. A basic awareness of how memory is managed in C, particularly the concept of variable addresses and the use of the address-of operator (&), is crucial. This background will help learners comprehend how pointers reference memory locations and how structures group related data, forming the foundation for more complex programming tasks like dynamic memory management and data structure implementation.

Keywords

Dereferencing, Self-Referential Structure, Array Pointer, Arrow Operator, Dynamic Memory Allocation

Discussion

A pointer is a variable that holds the memory address of another variable instead of storing the actual data itself. Rather than containing a value, it points to the location in memory where the data is stored. In C programming, pointers are essential for managing memory at a low level. When you access a pointer directly, it shows the memory address it refers to, not the data stored at that location.

```
#include <stdio.h>
int main()
 {
     int var = 10;            // Normal Variable
     int* ptr = &var;         // Pointer Variable ptr that stores address of var
    printf("%d", ptr);        // Directly accessing ptr will give us an address
     return 0;
}
```

Output

0x7fffa0757dd4

The memory address is represented as a hexadecimal number, beginning with 0x as in Fig 1.3.1.

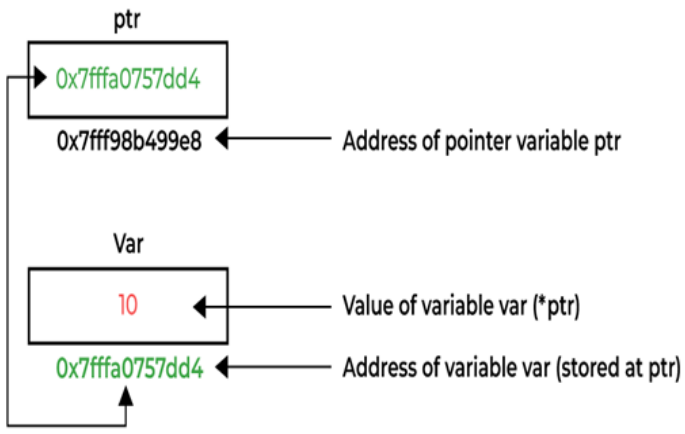


Fig 1.3.1 C Pointers

1.3.1 Creation of Pointers

A pointer is declared in a similar way to a regular variable, but with an asterisk (*) before its name to signify that it is a pointer.

Syntax

```
data_type *pointer_name;
```

Here, data_type specifies the type of variable the pointer will refer to. For instance, a pointer to an int can only store the address of an integer variable, while a pointer to a float holds the address of a floating-point variable.

Example:

```
int *ptr;
```

This line declares ptr as a pointer to an integer, meaning it is designed to hold the memory address of an int variable. It is interpreted as "a pointer to an integer."

1.3.1.1 Initialise the Pointers

Pointer initialization is the act of assigning a memory address to a pointer variable. In C, this is typically done using the address-of operator (&), which retrieves the memory address of a variable. This address can then be stored in the pointer. You can also declare and initialize a pointer in a single statement, a practice referred to as pointer definition.

```
#include <stdio.h>
int main()
{
int num;                    //declaration of integer variable
 int *pNum;                 //declaration of integer pointer
pNum=& num;                 //assigning address of num
\num=100;
printf("Using variable num:\n");
printf("value of num: %d\naddress of num: %u\n",num,&num);
printf("Using pointer variable:\n");
 printf("value of num: %d\naddress of num: %u\n",*pNum,pNum);
return 0;
}
```

Output

```
Using variable num:
value of num: 100
address of num: 2764564284
Using pointer variable:
value of num: 100
address of num: 2764564284
```

1.3.2 Dereference Operator

Dereferencing is the process of accessing the value stored at the memory address that a pointer refers to. In C, this is done using the asterisk (*) operator, commonly known as the dereference operator. When applied to a pointer variable, it retrieves the actual data from the memory location the pointer is pointing to. Essentially, dereferencing a pointer returns the value of the variable it points to.

1.3.2.1 How to dereference a pointer

To dereference a pointer in C, the asterisk (*) symbol is placed before the pointer variable. This allows access to the value stored at the memory address the pointer is referencing.

```
int main()
{
    int x = 19,deref;
    int *ptr;          // creating a pointer
    ptr = &x;          // Stores address of x in ptr
    deref = *ptr;      // Put Value at ptr in deref
    printf("%d",deref);
}
```

Output

19

Steps to dereference the pointer

1. An integer variable x was declared and initialized with the value 19.
2. Another integer variable named deref was also declared.
3. A pointer variable ptr was created to store the address of x.
4. The address-of operator (&) was used to assign the memory address of x to ptr.
5. The pointer ptr was dereferenced using the asterisk (*) operator to access the value stored at that memory address.
6. The retrieved value was then assigned to the variable deref.

1.3.2.2 Need to dereference a pointer

Dereferencing a pointer allows direct access to the value stored at the memory address held by the pointer. This helps minimize both memory usage and program complexity.

When an operation is performed on a dereferenced pointer, it directly modifies the value of the original variable associated with that address. As a result, the program runs more efficiently, since there is no need to work on or update the original variable separately.

1.3.2.3 Manipulating Value by Dereferencing Pointer

The dereference operator enables indirect modification of a variable's value through its pointer.

```
#include <stdio.h>
int main ()
{
int a = 10;
int *b = &a;
float x = 10.5;
float *y = &x;
*b = 100;
*y = 100.50;
printf ("Address of 'a': %d Value of 'a': %d\n", b, *b);
printf ("Address of 'x': %d Value of 'x': %f\n", y, *y);
return 0;
}
```

Output

```
Address of 'a': 6422028 Value of 'a': 100
Address of 'x': 6422024 Value of 'x': 100.500000
```

In this example, the values of "a" and "x" are modified using their corresponding dereferenced pointers.

1.3.3 Pointer to an Array

An array pointer is a type of pointer that points to the whole array rather than just the first element. It handles the array as a single entity instead of as a collection of separate elements.

Syntax : `type (*ptr)[size];`

Where:

- type specifies the data type stored in the array.
- ptr is the name of the pointer.
- size represents the number of elements in the array the pointer refers to.

Example : int (*ptr)[10];

Here, ptr is a pointer to an array of 10 integers.

Since array subscripts ([]) have higher precedence than the dereference operator (*), the pointer name and * must be enclosed in parentheses to ensure correct interpretation.

1.3.3.1 Access Array Using Array Pointer

In the code shown below, the pointer ptr is of the type "pointer to an array of 10 integers."

```
#include <stdio.h>
int main()
{

    int arr[3] = { 5, 10, 15 };
    int n = sizeof(arr)     //sizeof(arr[0]);
    int (*ptr)[3];          // Declare pointer variable

  // Assign address of val[0] to ptr.
  // We can use ptr=&val[0];(both are same)
  ptr = &arr;
  for (int i = 0; i < n; i++)
  printf("%d ", (*ptr)[i]);
  return 0;

}
```

Output
5 10 15

1.3.3.2 Pointer to Multidimensional Arrays

1. **Pointers to 2D Arrays**

When declaring a pointer to a 2D array, it is necessary to specify both the number of rows and columns in the pointer's definition.

Syntax: `type *(ptr)[row][cols]`

```
#include <stdio.h>
int main()
{
 int arr[2][3] = {{1, 2, 3},
{4, 5, 6}};    // pointer to above array
 int (*ptr)[2][3] = &arr;    // Traversing the array using ptr
 {
 for (int j = 0; j < 3; j++)
 {
  printf("%d ", (*ptr)[i][j]);
  }
  printf("\n");
  }
return 0;
}
```

Output

1 2 3
4 5 6

2. Pointers to 3D Arrays

To declare a pointer to a two-dimensional array, you must include both the row and column dimensions in the pointer definition.

Syntax:

```
type *(ptr)[depth][row][cols]
```

```
#include <stdio.h>
int main()
{
 int arr[2][3][2] = {{{1, 2}, {3, 4}, {5, 6}},
{{7, 8}, {9, 10}, {11, 12}}};
int (*ptr)[2][3][2] = &arr;   // Pointer to the 3D array
  // Traversing the 3D array using the pointer
  for (int i = 0; i < 2; i++)
{
for (int j = 0; j < 3; j++)
 {
  for (int k = 0; k < 2; k++)
  {
   printf("%d ", (*ptr)[i][j][k]);
   }
   printf("\n");
```

```
  }
  printf("\n");
  }
 return 0;
 }
```

Output

```
1 2
3 4
5 6
7 8
9 10
11 12
```

1.3.4 Structures

In C, a structure is a user-defined data type that allows grouping of variables of different data types under a single name. It is defined using the struct keyword. The variables within a structure are known as its members, and they can be of any valid data type. Structures are commonly used in building complex data structures like linked lists and trees. They are also useful for modeling real-world entities in software applications, such as representing students and faculty members in a college management system.

```
#include <stdio.h>                  // Defining a structure
struct A
 {
        int x;
};

int main()
 {
struct A a;                 // Creating a structure variable
a.x = 11;                   // Initializing member
          printf("%d", a.x);
          return 0;
}
```

In this example, a structure named A is defined with an integer member x. A variable a of type struct A is then declared, and its member x is assigned the value 11 using the dot (.) operator. Finally, the value of a.x is displayed on the screen.

1.3.4.1 Syntax of Structure

There are two steps of creating a structure in C:

1. Structure Definition
2. Creating Structure Variables

1. Structure Definition

A structure is declared using the struct keyword, followed by the structure name and its members. This declaration acts as a structure template or prototype, and it does not allocate any memory at this stage.

Syntax:

```
struct structure_name
 {
        data_type1 member1;
        data_type2 member2;
        …
}
```

structure_name refers to the name given to the structure. member1, member2, ... are the identifiers used for the structure's members. data_type1, data_type2, ... indicate the data types assigned to each member of the structure.

2. Creating Structure Variable

Once a structure is defined, a variable of that structure type must be created in order to use it. This process is similar to declaring variables of any other data type.

Syntax:

```
struct strcuture_name var;
```

We can also declare structure variables with structure definition.

Syntax:

```
struct structure_name
{
...
 }var1, var2....;
```

```
#include <stdio.h>
struct Student       // Structure declaration
 {

    int id;
    char name[50];
    float marks;
};
int main()
{
      struct Student s1;            // Structure variable declaration
      s1.id = 101;                    // Assigning values to structure members
      strcpy(s1.name, "Alice");   // Copying string into the 'name' member
      s1.marks = 87.5;
     // Displaying structure member values
     printf("Student ID: %d\n", s1.id);
     printf("Student Name: %s\n", s1.name);
     printf("Student Marks: %.2f\n", s1.marks);
     return 0;
}
```

Output

Student ID: 101
Student Name: Alice
Student Marks: 87.50

1.3.5 Access Structure Members

To access or modify the members of a structure, the dot operator (.) is used when working directly with a structure variable.

Syntax:

```
structure_name.member1;
structure_name.member2;
```

```
#include <stdio.h>
struct Point
 {
    int x;
    int y;
};
int main()
{
        // Declare and initialize multiple structure variables
          struct Point p1 = {5, 10};
          struct Point p2 = {3, 8};
          struct Point p3 = {7, 12};
       // Access and print the values of structure members
         printf("Coordinates of p1: (%d, %d)\n", p1.x, p1.y);
         printf("Coordinates of p2: (%d, %d)\n", p2.x, p2.y);
        printf("Coordinates of p3: (%d, %d)\n", p3.x, p3.y);
        return 0;
}
```

Output

```
Coordinates of p1: (5, 10)
Coordinates of p2: (3, 8)
Coordinates of p3: (7, 12)
```

However, if you are using a pointer to a structure, the arrow operator (->) is used to access the members.

Syntax:

```
structure_ptr->member1;
structure_ptr->member2;
```

```
#include <stdio.h>
#include <string.h>
struct Student
 {
   int id;
   char name[50];
   float marks;
};

int main()
 {
```

```
struct Student s1;
    struct Student *ptr;     // Declare a pointer to the structure
    ptr = &s1;               // Assign the address of s1 to the pointer

    // Assign values to structure members using pointer and arrow operator
    ptr->id = 102;
    strcpy(ptr->name, "Bob");
    ptr->marks = 91.5;

  // Display structure member values using the pointer
    printf("Student ID: %d\n", ptr->id);
    printf("Student Name: %s\n", ptr->name);
    printf("Student Marks: %.2f\n", ptr->marks);

    return 0;
}
```

Output

```
Student ID: 102
Student Name: Bob
Student Marks: 91.50
```

1.3.6 Self Referential Structure

A self-referential structure in C is a struct data type in which one or more members are pointers to variables of the same structure type. These user-defined types are highly valuable in C programming and play a crucial role in creating complex and dynamic data structures like linked lists, trees, and graphs.

Syntax:

```
strut typename
{
      type var1;
      type var2;
     ...
    ...
     struct typename *var3;
}
```

```
#include <stdio.h>
 struct mystruct
 {
int a;
 struct mystruct *b;
 };
 int main()
{
struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30,
NULL};
     struct mystruct * p1, *p2, *p3;
     p1 = &x;
     p2 = &y;
     p3 = &z;
     x.b = p2;
     y.b = p3;
printf("Address of x: %d a: %d Address of next: %d\n", p1, x.a,
x.b);
printf("Address of y: %d a: %d Address of next: %d\n", p2, y.a,
y.b);
printf("Address of z: %d a: %d Address of next: %d\n", p3, z.a,
z.b);
return 0;
}
```

Output

```
Address of x: 659042000 a: 10 Address of next: 659042016
Address of y: 659042016 a: 20 Address of next: 659042032
Address of z: 659042032 a: 30 Address of next: 0
```

1.3.6.1 Purpose of Self-Referential Structures

Self-referential structures are a fundamental concept in programming, mainly used for creating recursive and flexible data structures.

- **Recursive Data Models**: They make it possible to design data structures that reference themselves, allowing for the construction of linked lists, trees, graphs, and stacks that can grow or shrink as needed.
- **Dynamic Memory Handling:** These structures support the allocation of memory at runtime, enabling the management of data when its size isn't known in advance.
- **Optimized Memory Use:** Memory is allocated only when required, making this approach efficient for structures that frequently change in size.

1.3.6.2 Applications of Self-Referential Structures

In C, a self-referential structure includes a pointer to a structure of the same type. This technique is commonly used in the following data structures:

- **Linked Lists:** In singly linked lists, each node contains a pointer to the next node. Doubly linked lists add an additional pointer to the previous node.
- **Binary Trees:** Each tree node holds pointers to both left and right child nodes.
- **Graphs:** Graph nodes can store references to adjacent nodes, forming interconnected networks.
- **Stacks and Queues:** These can be efficiently implemented using linked lists, allowing dynamic push and pop operations without predefined limits.

Summarised Overview

The unit provides an in-depth explanation of pointers and structures in C programming. It starts by defining pointers as variables that store memory addresses of other variables and explains how to declare, initialize, and dereference them to access or modify data indirectly. It covers pointers to arrays, including single-dimensional, multidimensional (2D and 3D) arrays, demonstrating how to declare and traverse them using pointers. The discussion then shifts to structures, a user-defined data type grouping variables of different types under one name. It explains structure declaration, variable creation, and member access using dot and arrow operators. Finally, the unit explores self-referential structures, where a structure contains pointers to its own type, essential for building dynamic data structures like linked lists, trees, and graphs, highlighting their role in efficient memory management and flexible data modeling.

Assignments

1. Write a C program to declare an integer variable and a pointer to it. Initialize the pointer and use it to modify the value of the integer variable. Print the value and address before and after modification.

2. Create a C program that declares a pointer to a one-dimensional array of 5 integers. Initialize the array and use the pointer to print all array elements.

3. Write a C program to define a structure Student with members id, name, and marks. Create a structure variable and assign values to each member. Then print the details using both the dot operator and a pointer with the arrow operator.

4. Implement a C program that declares a pointer to a 2D array (3x3), initializes the array, and uses the pointer to print all elements of the 2D array.

5. Design a self-referential structure for a singly linked list node containing an integer data member and a pointer to the next node. Write a program to create three nodes, link them, and print their data and addresses.

Reference

1. Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed.). Prentice Hall.
2. Deitel, P. J., & Deitel, H. M. (2017). *C how to program* (7th ed.). Pearson.
3. Gaddis, T. (2015). *Starting out with C* (4th ed.). Pearson.
4. Balagurusamy, E. (2017). *Programming in ANSI C* (7th ed.). McGraw-Hill Education.
5. Kanetkar, Y. (2018). *Let us C* (15th ed.). BPB Publications.

Suggested Reading

1. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.

2. Deitel, P. J., & Deitel, H. M. (2017). *C How to Program* (7th ed.). Pearson.

3. Gaddis, T. (2015). *Starting Out with C* (4th ed.). Pearson.

4. Balagurusamy, E. (2017). *Programming in ANSI C* (7th ed.). McGraw-Hill Education.

5. Yashavant Kanetkar. (2018). *Let Us C* (15th ed.). BPB Publications.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

Dynamic Memory Allocation

Learning Outcomes

At the end of this unit, the learner will be able to:

- define static and dynamic memory allocation.
- list the functions used for dynamic memory allocation in C.
- differentiate between static and dynamic memory allocation with examples.
- identify the correct syntax of malloc(), calloc(), realloc(), and free() used in dynamic memory management

Background

In many programming situations, we already know how much memory a variable or array will need. For example, if we declare int marks[50];, we are reserving space for exactly 50 integers. This kind of memory assignment is called static memory allocation, and it happens before the program begins running at compile time. This method works well when the size of the data is fixed and known in advance.

However, real-world programs are not always so predictable. Suppose we are writing a program to store student names entered by the user. We cannot always predict how many names the user will input. Using static memory for such problems can either lead to memory wastage (allocating more than needed) or memory shortage (allocating less than needed).

This is where dynamic memory allocation becomes important. In C programming, dynamic memory allocation allows a program to request memory from the system while it is running at runtime. This helps programmers manage memory more efficiently, especially when working with large datasets, linked lists, or other dynamic data structures.

Keywords

Dynamic memory allocation, malloc, calloc(), realloc(), free()

Discussion

1.4.1 Memory Allocation

Memory allocation is a fundamental concept in programming that deals with the process of assigning memory space to variables and data structures during the execution of a program. In the C programming language, memory allocation can be categorized into two main types: static memory allocation and dynamic memory allocation. Static memory allocation occurs at compile time, where the size and location of memory are fixed and cannot be altered during program execution. In contrast, dynamic memory allocation takes place at runtime, allowing programs to request and release memory as needed, based on the program's current requirements.

Understanding the distinction between these two types of memory allocation is essential for efficient memory management. It enables programmers to utilize system resources effectively, reduce memory wastage, and develop optimized programs that can adapt to varying data sizes and user inputs.

1.4.1.1 Static Memory Allocation

In static memory allocation, the memory is assigned to variables at compile time, that is, before the program starts running. The size and memory location of the variables are fixed and determined by the compiler. This means that once the memory is allocated, the program cannot change the size of that memory during execution.

This method is suitable when the size of the data is known in advance and does not change. It is commonly used when we declare arrays or variables with a fixed size. Since the memory is already allocated at compile time, it is easy for the program to access and manage it. However, it also has limitations.

- If the allocated memory is too large, the extra memory is wasted.
- If the allocated memory is too small, the program may run into memory-related errors because it cannot store all the required data.

Example of static memory allocation:

```
int arr[10];
```

In this example, the compiler allocates space for an integer array of size 10. Whether we use all 10 elements or not, the memory is reserved and cannot be changed while the program runs.

1.4.1.2 Dynamic Memory Allocation

Dynamic memory allocation allows the program to request memory during runtime, that is, while the program is actually running. This is done using functions provided in the C Standard Library such as malloc(), calloc(), and realloc(). Memory allocated dynamically is taken from a special memory area called the heap.

The key advantage of dynamic memory allocation is flexibility. The program can request exactly the amount of memory it needs based on the user's input or other runtime conditions. If more memory is needed later, the program can ask for it. If the memory is no longer needed, it can be released using the free() function. This approach helps in the efficient use of memory resources.

Each of these library functions malloc(), calloc(), realloc(), and free() plays a specific role in managing dynamic memory and will be discussed in detail in the following sections.

Dynamic memory allocation is very useful in the following situations:

- When the size of the data is not known in advance.
- When the data size may change during program execution.
- When we want to avoid wasting memory by allocating only what is required.

1.4.1.3 Static v/s Dynamic Memory Allocation

Table 1.4.1 Comparison between static and Dynamic memory allocation

Feature	Static	Dynamic
Time of Allocation	Compile time	Runtime
Memory Size	Fixed	Can be changed during execution
Memory Area	Stack	Heap
Flexibility	Low – size cannot be changed	High – size can be increased or decreased
Wastage Risk	High – may waste memory if not fully used	Low – only allocate as needed
Functions Used	No special function needed	Uses malloc(), calloc(), realloc(), free()
Suitable For	Known and fixed data size	Unknown or varying data size

1.4.2 Allocating Dynamic Memory

In C programming, dynamic memory is allocated during runtime using functions provided in the C Standard Library (<stdlib.h>). These functions help in requesting memory from the heap, a special area in memory meant for dynamic allocation. The key functions used for this purpose are:

- malloc()
- calloc()
- realloc()
- free()

1.4.2.1 malloc() – Memory Allocation

The malloc() function stands for memory allocation. It is used in C programming to dynamically allocate a single continuous block of memory while the program is running. The size of the memory block is specified in bytes.

When memory is allocated using malloc(), it is not initialized. This means that the newly allocated memory will contain unknown or garbage values, as it may still hold data from previous operations in the memory.

Syntax:

```
pointer_variable = (data_type *) malloc(size_in_bytes);
```

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
int *arr;
int i;

arr = (int *)malloc(5 * sizeof(int));

if(arr == NULL) {
printf("Memory allocation failed\n");
return 1;
}

printf("Enter 5 integers:\n");

for(i = 0; i < 5; i++) {
scanf("%d", &arr[i]);
}
```

```
printf("The entered numbers are:\n");

for(i = 0; i < 5; i++) {
printf("%d ", arr[i]);
}

free(arr);
arr = NULL;

return 0;
}
```

Output:

```
Enter 5 integers:
10
20
30
40
50

The entered numbers are:
10 20 30 40 50
```

The statement malloc(5 * sizeof(int)) dynamically allocates memory from the heap to store five integer values. The function sizeof(int) returns the number of bytes required to store an integer. On most systems, an integer occupies 4 bytes, so the total memory allocated is 5 × 4 = 20 bytes.

The pointer arr stores the address of the first byte of the allocated memory block. Using this pointer, the program stores values entered by the user in the allocated memory. Since malloc() does not initialize the allocated memory, the memory locations initially contain garbage values. Therefore, the program assigns values to the allocated memory before using it.

After the memory is no longer required, the free() function is used to release the allocated memory. The pointer is then set to NULL to avoid a dangling pointer.

1.4.2.2 calloc() – Contiguous Allocation

The calloc() function stands for contiguous allocation. It is used to allocate multiple blocks of memory, each of the same size. Unlike malloc(), it initializes all allocated memory blocks to zero.

Syntax:

```
pointer_variable = (data_type *) calloc(number_of_elements, size_of_each_
element);
```

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    arr = (int *)calloc(5, sizeof(int));  // Allocates memory for 5 integers

    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Values in calloc allocated memory:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);  // Will print 0 for all elements
    }

    free(arr);  // Free the allocated memory
    arr = NULL;
    return 0;
}
```

Output:

```
0 0 0 0 0
```


In the given program, the calloc() function is used to allocate memory dynamically for storing 5 integers. The expression calloc(5, sizeof(int)) requests memory for 5 elements, each of size equal to an int. On most systems, an int typically occupies 4 bytes, so the total memory allocated would be 5 × 4 = 20 bytes. One of the key differences between calloc() and malloc() is that calloc() automatically initializes all the allocated memory to zero. This means that each element in the allocated array will have an initial value of 0.

1.4.2.3 realloc() – Reallocation of Memory

In dynamic memory allocation, we sometimes find that the memory allocated using malloc() or calloc() is either too much or not sufficient for our needs. Instead of allocat-

ing a new memory block and copying the old data manually, C provides a convenient function called realloc() to resize an existing memory block.

The realloc() function is used to increase or decrease the size of a previously allocated memory block without losing the existing data (up to the new size).

Syntax:

```
pointer_variable = (data_type *) realloc(previous_pointer, new_size);
```

Where,

- previous_pointer: This is the pointer to a memory block previously allocated using malloc(), calloc(), or even realloc().
- new_size: This is the new size in bytes that you want for the memory block.
- data_type *: This is a typecast used to convert the void* pointer returned by realloc() into the desired data type.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *arr;
   int i;

   // Initial allocation using malloc
   arr = (int *)malloc(3 * sizeof(int));
   if (arr == NULL) {
      printf("Memory not allocated.\n");
      return 1;
   }

   // Assign values to the initial memory
   for (i = 0; i < 3; i++) {
      arr[i] = i + 1;
   }

   // Display original values
   printf("Original values:\n");
   for (i = 0; i < 3; i++) {
      printf("%d ", arr[i]);
   }
```

```
    // Reallocate memory to hold 5 integers instead of 3
    arr = (int *)realloc(arr, 5 * sizeof(int));
    if (arr == NULL) {
        printf("\nMemory reallocation failed.\n");
        return 1;
    }

    // Assign values to the newly added memory
    arr[3] = 4;
    arr[4] = 5;

    // Display values after reallocation
    printf("\nValues after realloc:\n");
    for (i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    // Free the memory
    free(arr);
    arr = NULL;

    return 0;
}
```

Output:

```
Original values:
1 2 3
Values after realloc:
1 2 3 4 5
```


Initially, memory is allocated for 3 integers using malloc(), and the values 1, 2, and 3 are stored in it. Later, the program needs space for 5 integers. Instead of creating a new memory block and copying the values manually, the realloc() function is used. This function increases the size of the previously allocated memory from 3 integers to 5. The original values (1, 2, and 3) remain unchanged. The program can now directly store two more values in the extra space. If there is enough space next to the original memory block, realloc() simply extends it. Otherwise, it creates a new block, copies the old values into it, and returns a pointer to the new block. In the end, free() is used to release the memory and avoid memory leaks.

1.4.2.4 free() - Freeing Memory

When the dynamically allocated memory is no longer needed, it should be released using the free() function. This prevents memory leaks, which happen when memory stays allocated even though it is no longer used by the program.

Syntax:

```
free(ptr);
```

Here, ptr must be a pointer returned earlier by malloc(), calloc(), or realloc().

Example:

```
int *arr;
arr = (int *)malloc(10 * sizeof(int));
free(arr);
arr = NULL;
```

After calling free(), the memory pointed to by arr is returned to the system and can be reused.

Summarised Overview

In C programming, memory allocation is the process of assigning space in memory to variables and data structures, and it can be done either statically or dynamically. Static memory allocation occurs at compile time, where the size and location of memory are fixed and cannot be altered during execution. This is efficient when the data size is known in advance but lacks flexibility and may lead to memory wastage. Dynamic memory allocation, on the other hand, happens at runtime using functions from <stdlib.h> such as malloc(), calloc(), and realloc(), which allocate memory from the heap. malloc() allocates a block of uninitialized memory, while calloc() allocates and initializes memory to zero. If the allocated size needs to change, realloc() adjusts the size of an existing block without losing its stored data. Once the memory is no longer needed, the free() function should be used to release it back to the system, preventing memory leaks. Dynamic allocation is especially useful when the size of data is unknown or variable during program execution. By understanding and applying these concepts correctly, programmers can optimize memory usage, avoid wastage, and ensure that programs adapt efficiently to changing data requirements.

Assignments

1. Explain static memory allocation in C with an example. Discuss its advantages and limitations.

2. Differentiate between static and dynamic memory allocation in C. Present your answer in a tabular format.

3. Describe the working of the malloc() function in C. Provide a suitable example and explain the output.

4. Explain the purpose of the calloc() function in C. How does it differ from malloc()? Illustrate with an example.

5. What is the role of the realloc() function in C? Explain how it helps in resizing memory blocks with an example.

Reference

1. https://nptel.ac.in/courses/106102064

2. https://www.geeksforgeeks.org/c/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Suggested Reading

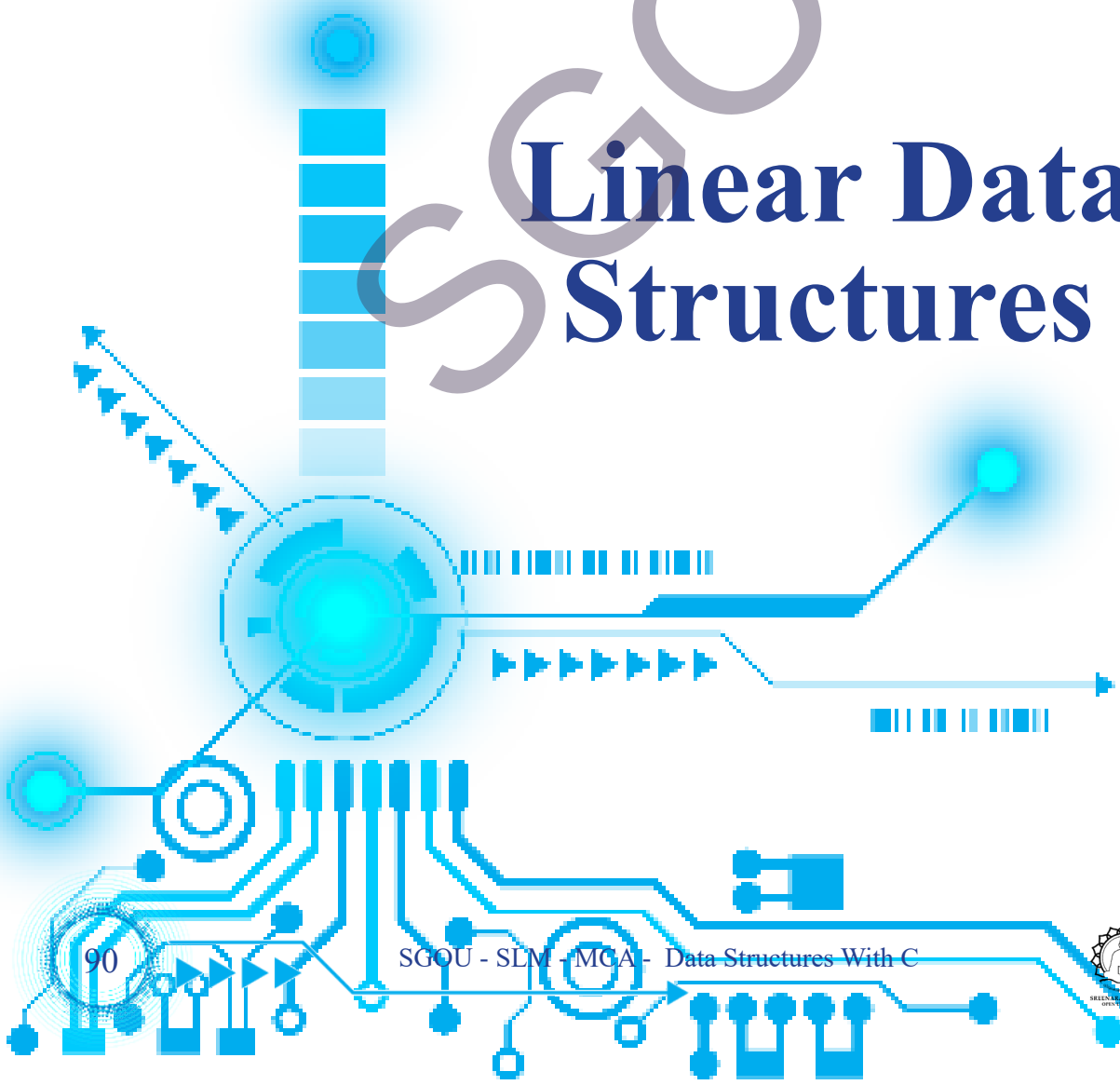
1. Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2022.
2. Samanta, D. "Classic data structures." Terminology 2 (2001): 1.
3. Levitin, Anany. Introduction to design and analysis of algorithms, 2/E. Pearson Education India, 2008.Tam vid nos Catorev iverfeste ia? P.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 2

Linear Data Structures



1
UNIT

Linked List

Learning Outcomes

After completing this unit, students will be able to:

- explain the need for linked lists and their memory representation using self-referential structures.
- differentiate between singly, doubly, and circular linked lists with examples and applications.
- implement linked list operations (creation, insertion, deletion, and traversal) in C using dynamic memory allocation.
- apply linked list concepts in solving real-world problems such as stacks, queues, graph representations, file management, and dynamic memory allocation.

Background

Data storage and organization have been central challenges in computer science since the earliest days of programming. While arrays provided one of the first structured ways to store sequential data, they came with limitations such as fixed size and costly insertion or deletion operations in the middle of the sequence. As applications grew more complex, handling dynamic datasets like contact lists, transaction histories, or sensor readings here arose a need for a more flexible data structure.

This led to the development of linked lists, a fundamental concept in dynamic memory management. Unlike arrays, linked lists store elements as individual nodes that can be scattered across memory. Each node contains data and a reference (pointer) to the next node, allowing easy insertion and deletion without shifting large blocks of data.

Among the different types, the singly linked list is the simplest. It is characterized by nodes linked in one direction from the head node to the last node, which points to NULL. This design makes singly linked lists memory-efficient and relatively easy to implement. They became widely adopted in operating systems, compilers, and embedded systems where efficient memory allocation is essential.

Over time, singly linked lists have become a standard topic in programming courses, especially in the C language, because C offers low-level control over pointers and memory allocation. Learning this data structure not only builds problem-solving skills but also strengthens a programmer's understanding of dynamic memory, pointer manipulation, and algorithmic thinking.

Keywords

Linked List, Node, Pointer, Singly Linked List, Doubly Linked List, Circular Linked List, Traversal, Insertion, Deletion.

Discussion

We have to discuss in detail the types of linked lists and how data is stored in a linked list.
Suppose we want to store the names of students in a memory. There are two available ways for us to maintain that list in the memory. Namely:

- An Array
- A Linked List

You have already learned about arrays. An Array is a static data structure where elements are stored in contiguous memory locations, and there is a limit on the number of items to be stored.

However, a linked list is a dynamic data structure where there is no limit to the number of items. Unlike arrays, elements in the linked list are scattered in the memory, but each data is linked with pointers (pointer points to the address of the memory location where the data resides). The linked list grows when a new data item is added and shrinks when it is removed from the list. We will see the different types of linked lists in this unit.

2.1.1 Need for linked list

When we work with data in programming, we often need to store and manage collections of items. In many programs, arrays are commonly used for this purpose. Arrays allow us to store multiple values in a single variable and access them easily by index numbers. However, arrays come with some limitations. One of the biggest problems with arrays is that their size is fixed. This means that once we declare the size of an array, we cannot change it during the execution of the program. If we allocate more space than needed, it wastes memory. If we allocate less space, we cannot store all the required data.

Another problem with arrays is that inserting or deleting data can be difficult. If we want to add or remove an element in the middle of an array, we have to shift the other elements, which takes extra time and makes the program slower. This becomes a major issue when dealing with large amounts of data.

To solve these problems, linked lists are used. A linked list is a data structure that can easily grow or shrink in size while the program is running. In a linked list, each element is stored in a separate node, and these nodes are connected using pointers. This makes it very easy to insert or delete elements without shifting other data. Memory is allocated only when required, so there is no memory wastage.

For example, think about a music playlist app. In such an app, users may add or remove songs at any time. If we use an array, it will be difficult to manage the changing number of songs, especially if the playlist grows beyond the initial size. But if we use a linked list, we can easily add or remove songs as needed without worrying about the size limit or memory wastage.

Linked lists also allow faster insertion and deletion of elements compared to arrays. This makes them suitable for applications where data changes frequently. Another real-life example is a web browser's history feature. Browsers use linked lists to keep track of visited pages, allowing users to easily go back and forward between pages.

In summary, we need linked lists because they solve the major drawbacks of arrays. They provide flexible memory usage, easy insertion and deletion, and efficient management of changing data sizes. Linked lists are very useful in many real-life applications where data is frequently modified.

2.1.2 Memory Representation of Linked List

Let's understand how a linked list works with a simple example of storing the names of cities visited during a trip.

Imagine you visited four cities in this order: Chennai, Bangalore, Hyderabad, and Mumbai. You want to store these city names in memory. For this, we create four separate nodes, with each node storing one city name.

Each node has two parts:

- A data part to store the city name.
- A link part to store the address of the next node (the next city in the trip).

Let's say we store the first city, "Chennai," in the first node. The second node stores "Bangalore," the third stores "Hyderabad," and the fourth stores "Mumbai." These nodes do not have to be stored one after another in memory. They may be scattered in memory at addresses like 200, 205, 208, and 215.

Although these nodes are scattered, we can connect them using the link part. The first node, which stores "Chennai," also stores the address of the second node (205) in its link part. The second node contains "Bangalore" and the address of the third node (208). The third node stores "Hyderabad" and points to the fourth node (215). Finally, the last node, which stores "Mumbai," points to NULL in its link part, showing that it is the last node of the list.

To start accessing the linked list, we use a special pointer called START, which holds the address of the first node (200 in this case). From the first node, we can follow the links to move to each next node until we reach the last node where the link part has NULL. This process of moving from node to node is called traversing the linked list.

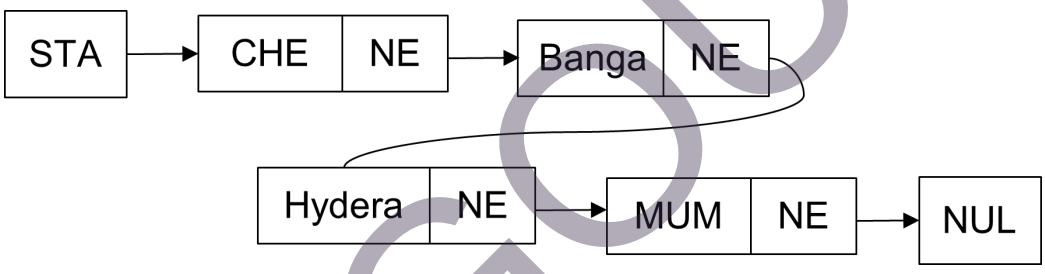


Fig. 2.1.1 Linked list

Explanation of the Diagram

- START points to the first node containing "Chennai."
- "Chennai" points to "Bangalore," which points to "Hyderabad."
- "Hyderabad" points to "Mumbai."
- "Mumbai" points to NULL, indicating it is the last node.

In a linked list, the nodes can be scattered anywhere in memory, but they are connected through their link parts. This allows us to efficiently store, add, or remove items like cities in our trip example, without worrying about memory location order.

In real life this is similar to a treasure hunt where each clue (node) tells you where to find the next clue by giving you the address or direction. You keep moving from one clue to the next until you reach the last one.

2.1.2.1 Implementation of LinkedList

How to create a node of a linked list in C? Linked list is a collection of nodes, where each node contains two portions, one is actual data and other is a pointer to the next node.

So to implement a linked list in C, we need a Structure. We have already learned that structure is a user defined datatype. The member of a structure can be a pointer and it may be a pointer to the same structure. We call such a structure a self-referential structure.

How a Node is declared in C?

For creating a node of Linked list, self-referential structure is used.

Self-referential structure is a structure which contains pointers pointing to a structure of the same type.

For example,

```
struct sample{
        int a; char b;
        struct sample *self
}
```

Here 'sample' is a structure having a pointer self which points to the struct sample itself.So it is referencing itself. Hence it is called self-referencing structure.

Node

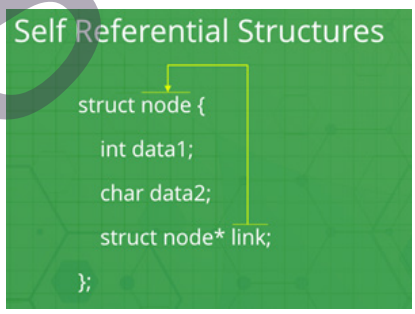


Fig. 2.1.2 Self Referential Structures

We have already seen that a node is a combination of two different types of data.It consists of data as well as link parts. To combine two different types into a single type, we use structure. Nodes in the linked list can be designed using structure as follows:

```
struct node
 {
int data1;
char data2;
struct node *link;
}
```

The structure struct node consists of three different types of data. One is an integer data and second is a character data. Data could be anything and any number of data is also possible. And data type could be anything like char, float or any other data type. Third element, link, is a pointer to the same structure type node. What is struct node *link?

Link is a pointer to some other node and as we know node is nothing but structure only. So the link must be a pointer to the struct node. Hence called self-referential structure.

2.1.2.2 Algorithm for Creation of a complete Linked list

The creation of a linked list can be viewed as a repeated insertion operation at the tail of the linked list.

Step 1: Create a node and collect the address of that node.

Step 2: Store data in the Data part of the node and NULL in the link part of the node

Step 3: Store the address of that node in the START, if that node is the first node.

Step 4:If it is not the first node, store the address of the new node in the link part of the previous node.

Step 5: Repeat the steps 1 to 4 till the user request stops.

2.1.2.3 Steps to create a Linked list

1. Linked list is empty

START

2. The first node is created with address 100. Data and link are filled

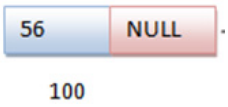


Fig. 2.1.3 single node

3. Address of the first node, here=100, is stored in the START

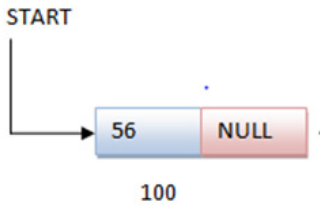


Fig. 2.1.4 Start with value 100

4. Second node is created with address 102.Data and link are filled

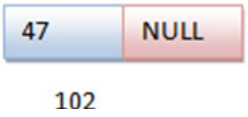


Fig. 2.1.5 Second node

5. Address of the Second node is stored in the link part of the first node.

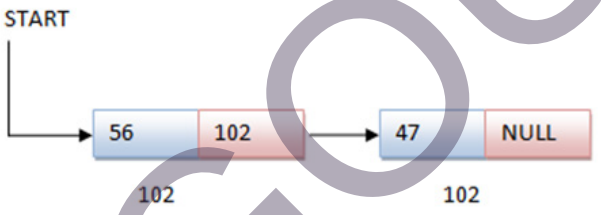


Fig. 2.1.6 Second node is stored in first node

6. Third node is created with address 104 and Data and Link are filled.

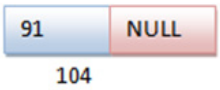


Fig. 2.1.7 Third node

7. The address of the third node is stored in the link part of the second node.

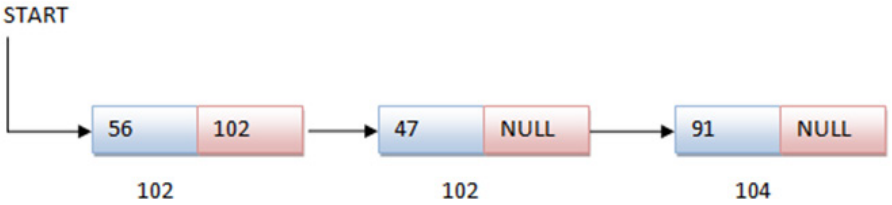


Fig. 2.1.8 Creation of a Linked list

Stacks and queues can be implemented using Linked list too,which results in dynamic stacks and queues. Here we discuss the creation of a singly linked list. In the next Unit we will discuss the insertion and deletion of nodes in singly linked list.

2.1.2.4 Creating a node in C

```
#include <stdio.h>
#include<stdlib.h> struct node{
int data;

struct node *link;

};

int main( ) {

struct node*start=NULL;   //start is a special pointer that point to first node. Here initialize to NULL

start = (node *) malloc(sizeof( node)); //malloc is a function for creating a node. The address of the   first node is stored in the startpointer

start →data=56;     // since it is a pointer the data is accessed using an arrow pointer. Initialize data by 56

start→link =NULL;  //link part is filled with NULL

printf("%d",start→data);   //print the data on first node using the access pointer

return 0;

}
```

2.1.3 Types of Linked List

Linked lists are divided mainly into three types: Singly Linked List, Doubly Linked List, and Circular Linked List. Each type has a different way of storing and connecting data.

2.1.3.1 Singly Linked List

In a singly linked list, each node has two parts. The first part stores the data and the second part stores the address of the next node. This type of linked list connects the nodes in only one direction, starting from the first node and ending at the last node. The last node does not point to any other node; instead, it stores a special value called NULL, which indicates the end of the list. In this type of list, we can only move forward through the nodes.

For example, consider storing the numbers 10, 20, 30, and 40. In this singly linked list, each number is stored in a separate node, and each node points to the next one.

A real-life example of a singly linked list is a to-do list in a mobile app where you can only move forward from one task to the next.

The figure below shows how the singly linked list works.

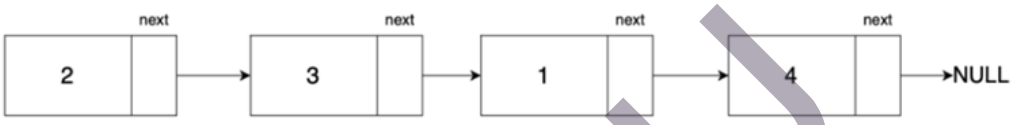


Fig.2.1.9 Singly linked list

Advantages of Singly Linked List:

- Simple and easy to implement.
- Requires less memory compared to doubly linked lists.
- Efficient in adding or deleting nodes at the beginning.

Disadvantages of Singly Linked List:

- Cannot move backward; only forward movement is possible.
- Traversing takes time as we must follow nodes one by one.
- Cannot directly access a node; searching is slow.

2.1.3.2 Doubly Linked List

A doubly linked list has three parts in each node. The first part stores the address of the previous node, the second part stores the data, and the third part stores the address of the next node. This type of linked list allows movement in both directions because each node points to both its previous and next nodes. The first node points to NULL in its previous link, and the last node points to NULL in its next link.

For example, we can store the numbers 5, 15, 25, and 35 in a doubly linked list. Here, you can easily move forward or backward between the numbers.

A common real-life example of a doubly linked list is the web browser history, where you can move to the next page or go back to the previous page.

The structure of a doubly linked list is shown below.

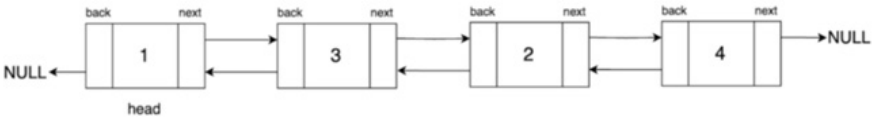


Fig.2.1.10 Doubly linked list

Advantages of Doubly Linked List:

- Allows movement in both directions.
- Easier to delete nodes from the middle of the list.
- More flexible for complex operations.

Disadvantages of Doubly Linked List:

- Requires extra memory to store the previous link.
- More complex to implement than singly linked lists.
- Slightly slower due to extra pointer operations.

2.1.3.3 Circular Linked List

In a circular linked list, the last node does not point to NULL. Instead, it points back to the first node, forming a circular connection among the nodes. This allows continuous movement through the nodes in a loop.

For example, if we want to store the numbers 1, 2, 3, and 4 in a circular linked list, the nodes will be connected in such a way that after reaching the last node containing 4, we will move back to the first node containing 1.

A good real-life example of a circular linked list is a music playlist with repeat mode, where the last song automatically connects back to the first song for continuous play.

The diagram below explains the circular linked list.

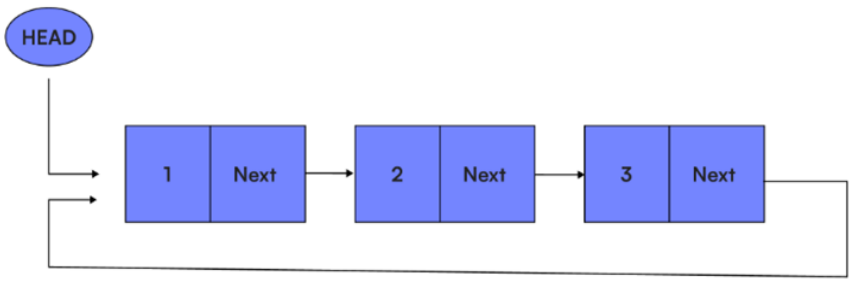


Fig.2.1.11 Circular linked list

Advantages of Circular Linked List:

- No need to check for NULL; traversal can continue endlessly.
- Suitable for applications that require looping, like scheduling systems.
- Efficient in situations requiring a circular path.

Disadvantages of Circular Linked List:

- Difficult to know where the list starts and ends.
- Risk of infinite loops if not handled carefully.
- More complex to implement compared to singly linked lists.

2.1.4 Operations

A linked list is a dynamic data structure in which elements, called nodes, are connected using links (pointers). Unlike arrays, linked lists do not store elements in contiguous memory locations. Because of this flexible structure, performing operations such as insertion and deletion becomes easier and more efficient.

The two most common operations performed on a linked list are insertion and deletion. Insertion refers to the process of adding a new node into the linked list, while deletion refers to removing an existing node from the list. These operations can be performed at different positions in the list depending on the requirement.

Insertion can occur at the beginning (first position), where a new node is added before the current first node. It can also occur in the middle position, where a node is inserted between two existing nodes by updating the links. Another possibility is insertion at the end (last position), where the new node is added after the last node of the list.

Similarly, deletion operations remove nodes from different positions in the list. A node can be removed from the beginning, where the first node is deleted and the start pointer is updated to the next node. Deletion can also occur in the middle, where a specific node between two nodes is removed by adjusting the links. Finally, deletion at the end removes the last node of the list by updating the link of the second-last node to NULL.

These operations are fundamental in linked lists because they allow the structure to grow or shrink dynamically without rearranging other elements in memory. This makes linked lists very useful in applications such as dynamic memory management, implementation of stacks and queues, and real-time data processing systems.

2.1.5 Insertion in a linked list

Insertion of an item in a linked list is the process of placing the node containing the item in a particular position .To insert the nodes into linked list, following three things should be done

1. Allocating anode.
2. Assigning the data.
3. Adjusting the pointers.

As in the case of arrays, nodes can be inserted anywhere in a linked list –at the beginning, at the end or in between any nodes. i.e., inserting a new node into the linked list has the following three instances.

A. Insertion at the beginning of the Linkedlist.

B. Insertion at the end of the Linkedlist.

C. Insertion at the specified position within the list.

2.1.5.1 Inserting a node at the beginning.

In order to insert a new node at the beginning of the linked list, the following steps are to be followed.

If the linked list is empty or the node to be inserted appears before the starting node, then insert that node at the beginning of the linked list.

When a new node is created, the node can be inserted at the beginning of the list by copying the content of Start into the link part of the new node and the address of the new node into Start.

An algorithm for inserting the new node at the beginning of the linked list.

Step 1: Create a new node with given value in data field of the new node

Step 2: Check whether list is Empty (Start==NULL),

Step 3: A new node is created and the address is stored in Start pointer and Link of new node is assigned as NULL

link [new_node]==NULL

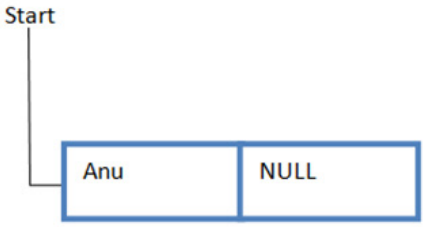


Fig. 2.1.12 single node

Step 4: If the list is not empty,

then set link[new node]= Start.

Start=new_node

Which means store the address of Start pointer (here 102) to the Link part of the new node and replace Start with the address of the new node (i.e. 100,the address of new node).

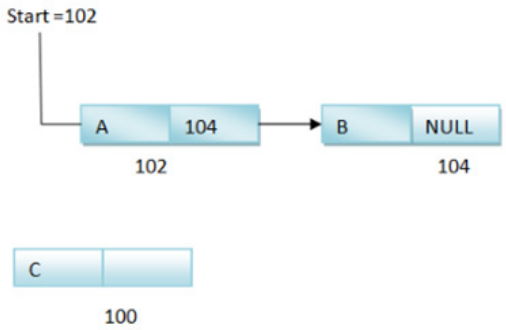


Fig. 2.1.13 (a) Linked list before adding the new node C

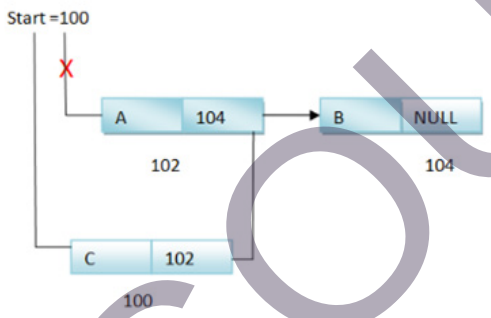


Fig. 2.1.13 (b) Inserting a node at the beginning of the linked list

2.1.5.2 Inserting at End of the List

Similarly, if the node to be inserted appears after the last node in the linked list then insert that node at the end of the linked list. For inserting a node at the end of the list, we have to copy the address of the new node to the link part of the last node. Assign NULL in the link part of the newly added node.

We can use the following steps to insert a new node at the end of the single linked list

Step 1: Create a node new_node with value in the data field (here, 25) and the link of new_node is NULL.

data[new_node]=25.

link[new_node]=NULL.

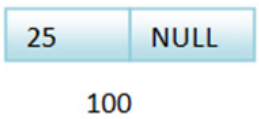


Fig. 2.1.14 (a) A node termed new_node is created with address100

Step 2: Check whether list is Empty (ie,Start==NULL)

Step 3: If the list is empty then, set Start=new_node

Start=100 //Assigning address of new node to the start.

Step 4: If it is not Empty then, a node pointer temp may be defined and initialize it with Start.

Step 5: Keep moving the temp to its following node till it reaches to the end node in the list (finding link of End node =NULL)

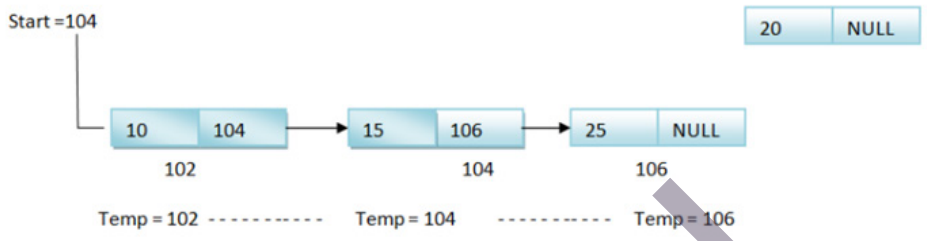


Fig. 2.1.14 (b) Temp traversed from first node to last node

Step 6: Make the link field of the last node to point to the new_node and make the link field of new_node =NULL.

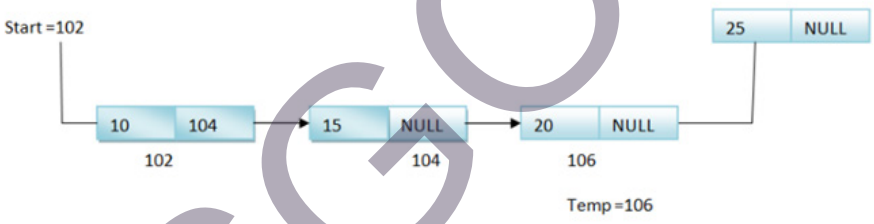


Fig. 2.1.14 (c) Inserting a node at the end of the linked list

2.1.5.3 Inserting at Specific Location in the list (in between two nodes)

The following steps are used to insert a new node in a specific location in the linked list. Here we are trying to insert a node in the third position in the linked list. A series of operations to be performed to insert at the third position in a linked list. Let temp, pre-node and post-node are pointers of Node type structure. Assume POS is a variable which contains the value of the position where the node is to be inserted. The following steps can be developed for the insertion operation.

Step 1: Create a node and the address is stored in temp.

Step 2: Store the data and link part of this node using temp.

Step 3: By traversing the list,identify the location where the new node is to be inserted. Also obtain the address of the nodes at position POS-1 and POS in the pointers pre-node and post-node respectively(POS1 and POS are the positions of pre-node and

post-node).

Step 4: Copy the content of the temp (address of the new node) into the link part of the node at position (POS-1), which can be accessed using prenode

Step 5: The content of postnode is copied (address of the node at position POS) to the link part of the new node which is pointed to by temp.

1. Linked list having three nodes.A new node is going to be inserted at the 3rd position.

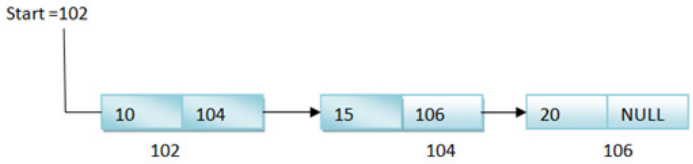


Fig. 2.1.15 (a) Linked list having three nodes

2. A new node is created and the address is stored in temp.

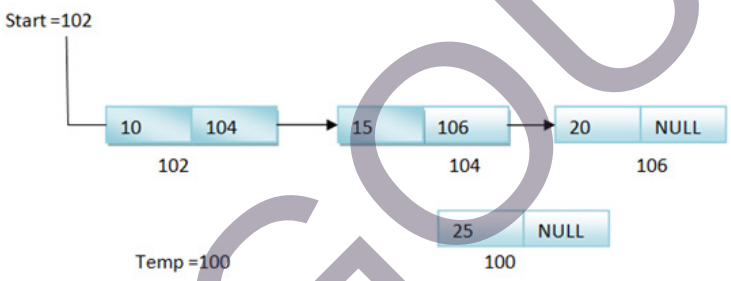


Fig. 2.1.15 (b) new node is created and the address is stored in temp

3. Address of the second node is copied into the prenode and the address of the third node is copied into the postnode.

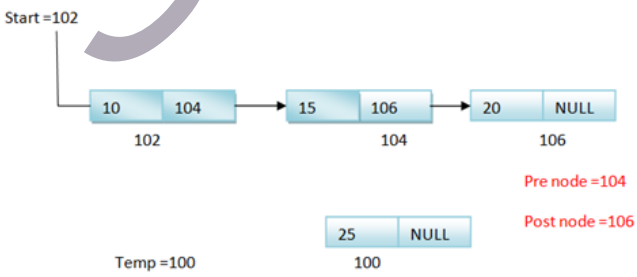


Fig. 2.1.15 (c) Address copied into the pre node and into the post node.

4. Address of the new node (from temp) is copied into the link part of the second node pointed to by the prenode and the new node is linked.

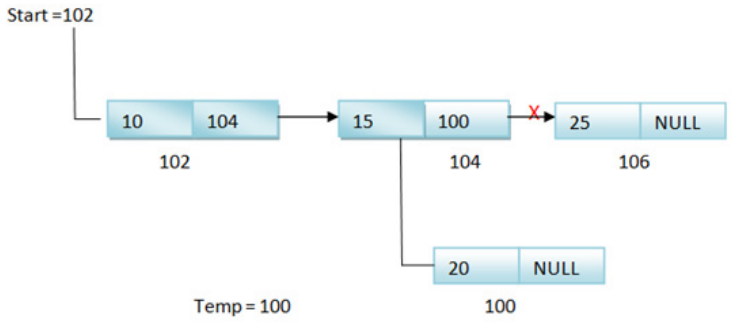


Fig. 2.1.15 (d) Linking of new node with pre node

5. Address of the fourth node available in the postnode is copied into the link list part of the new node.

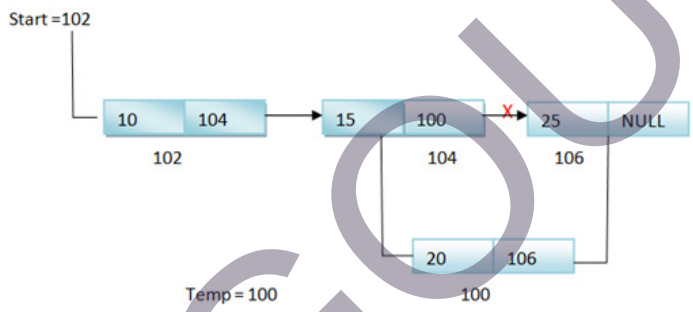


Fig. 2.1.15 (e) Linking of new node with post node

6. Linked list after the insertion of a new node at the third position.

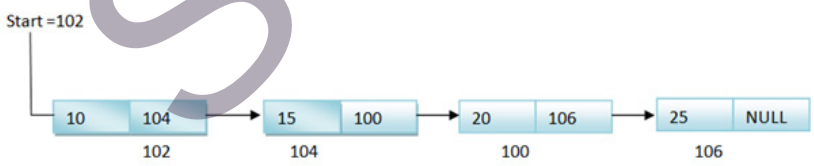


Fig. 2.1.15 (f) Inserting a node from a specific location of the linked list

2.1.6 Deletion from a linked list

Deleting an item from a linked list is the process of removing a node from the linked list. The position of the node to be removed will be given. Instead of this, if the data item is given, the node containing that item is to be searched and its position is to be noted so that we can apply the steps for removal operation. Any node, whether first, last or node at a specified position, can be removed from the list. To delete the first node, we have to copy the content in the link part of the first node into Start. The last node can be

deleted by assigning the NULL value to the link part of the second last node.

To remove a node from a specified position certain steps are to be performed which will be explained further in this unit.

2.1.6.1 Deleting from the beginning of the list

We can use the following steps to delete a node from the beginning of the single linked list.

Step 1: Check whether list is empty (Start ==NULL)

Step 2: If it is empty then, display 'List is Empty!!! Can't delete the node' and terminate the function.

Step 3: If it is not Empty then, Check whether list is having only one node

Step 4: If it is true then set Start=NULL(Setting Empty list conditions) and stop.

Step 5: If the list contains more than one node, then copy the content in the link part of the first node into Start.

1. List contains only one node.

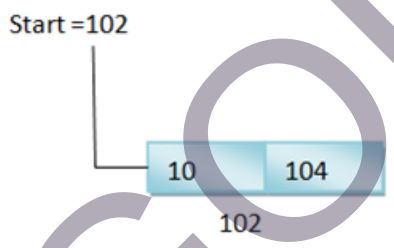


Fig. 2.1.16 (a) Just set Start = NULL

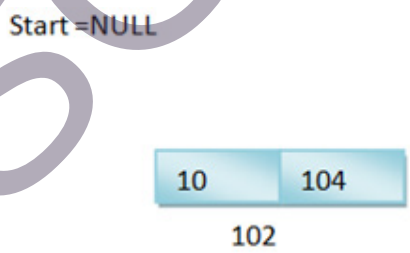


Fig. 2.1.16 (b) The node is removed from the list

2. List contains more than one node

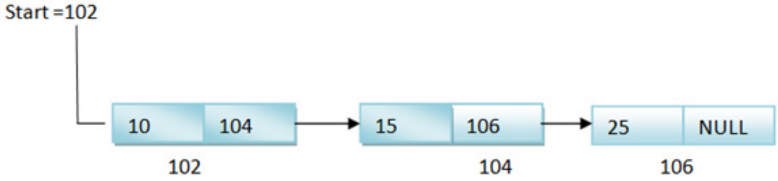


Fig. 2.1.16 (c)Copy the content in the link part of the first node into Start.

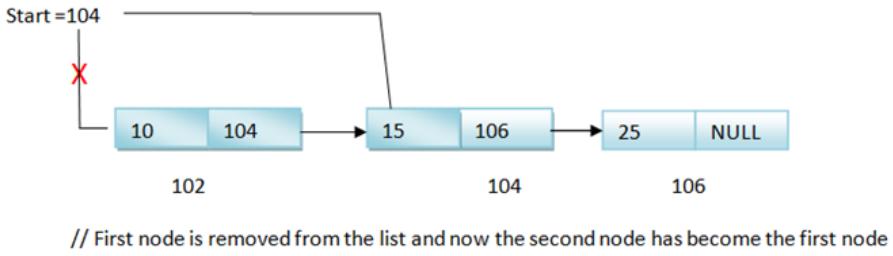


Fig. 2.1.16 (d) Deleting a node from the beginning of the linked list

The value of Start is now 104

2.1.6.2 Deleting from End of the List

We can use these steps to delete a node from the end of the singly linked list

Step 1: Check whether the list is empty (Start==NULL).

Step 2: If it is empty then, display 'Empty list..!!!

Cannot Delete and terminate the function

Step 3: If it is not Empty then, Check whether list is having only one node

Step 4: If it is true then set Start=NULL(Setting Empty list conditions) and stop.

Step 5: If the list has more than one node, then traverse until the last node is reached. (Keep the address of the previous node in pre node while traversing).

Step 6: Set NULL value in the link field of the pre node.Then the last node gets deleted from the list.

1. List contains only one node.

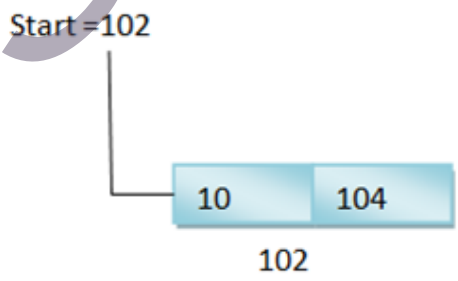


Fig. 2.1.17(a) First node

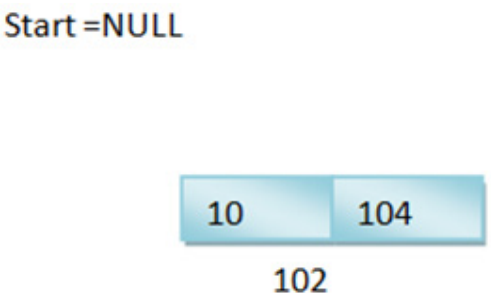


Fig. 2.1.17 (b) Just set Start = NULL

The node is removed from the list

2. List contains more than one node

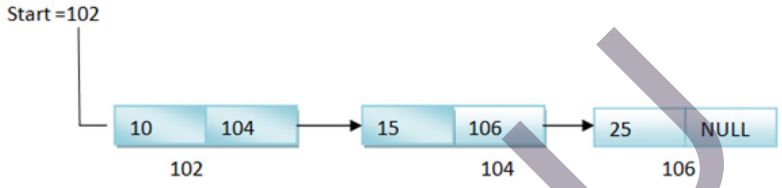


Fig. 2.1.17(c) Traverse until the last node is reached

(Keep the address of the previous node in pre node while traversing).

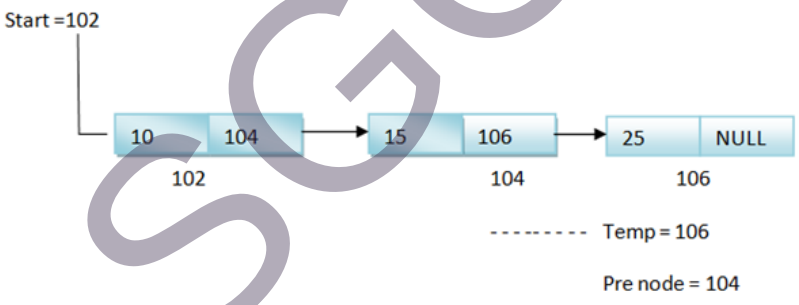


Fig. 2.1.17 (d) Set NULL value in the link field of the pre node

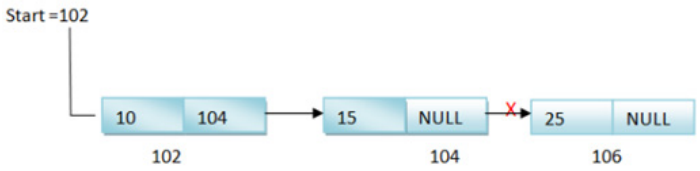


Fig. 2.1.17 (e) Deleting a node from the end of the linked list

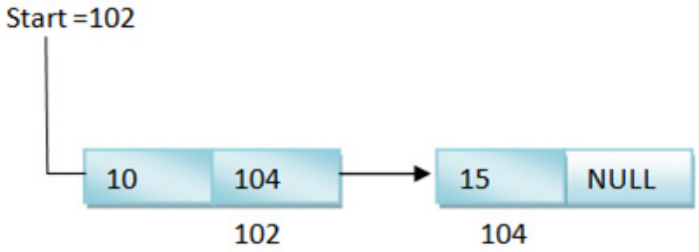


Fig. 2.1.17 (f) Linked list after the last node is deleted

2.1.6.3 Delete a node in specific location from the list

To remove a node from a specified position these all steps are to be performed. Below figure describes the procedure for the removal of the third node from the linked list having four nodes initially.it is assumed that prenode and postnode are pointers of Node type structure. Consider POS as a variable where the variable contains the position of the node to be removed. Following steps are involved in the deletion operation

Step 1: Obtain the address of the nodes at the position, POS1 and POS+1 in the pointers pre–node and post-node respectively, with the help of a traversal operation.

Step 2: Copy the content of postnode (address of the node at position POS+1) into the link part of the node at position (POS1), which can be accessed using prenode.

Step 3: Free the node at position POS.

Linked list having four nodes

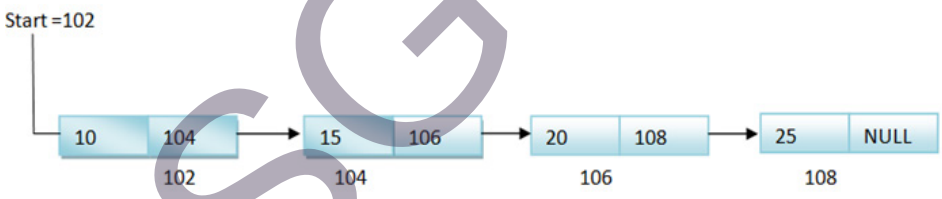


Fig. 2.1.18 (a) Linked list having four nodes

Address of the second node is stored in the prenode and the address of the fourth node is in postnode

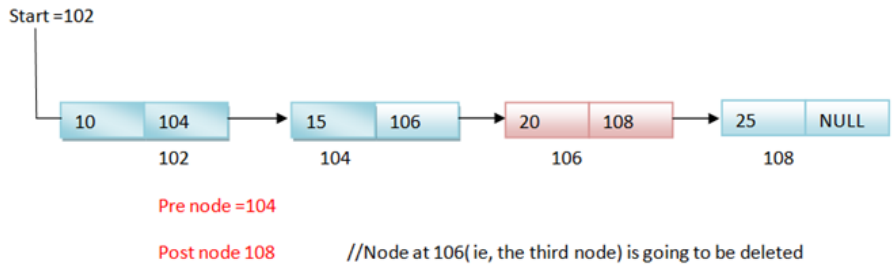


Fig. 2.1.18 (b) Third node is going to be deleted

Address of the fourth node available in post-node is copied into the link part of the second node pointed to by the prenode. Thus the third node is removed from the list.

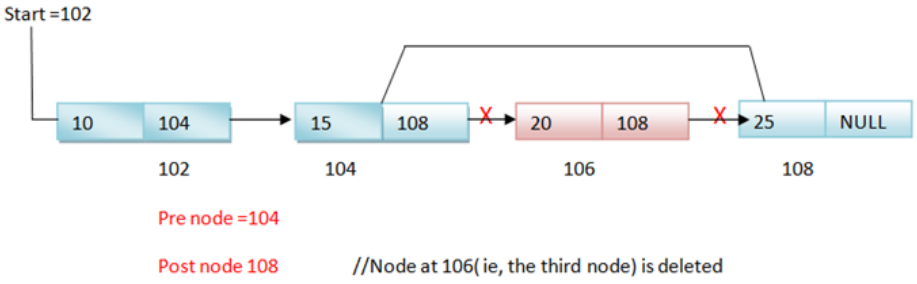


Fig. 2.1.18 (c) Third node is deleted

Linked list after the deletion of the third node.

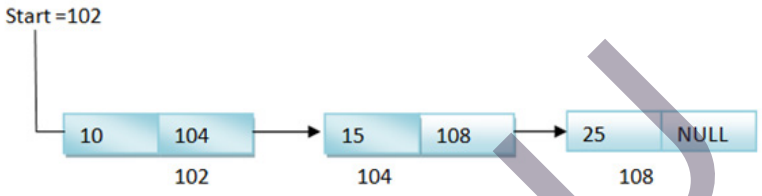


Fig. 2.1.18 (d) Deleting a node from a specific location of the linked list

If we apply the first two steps in the linked list, even after deleting the third node from the second node, the presence of the third node will still be there in the memory, pointing to the fourth node. So it should be freed using the memory deallocation facility provided by the programming language.

While implementing operations in the linked list, the temporary pointers like temp, prenode, postnode etc should also be freed after the operations.

2.1.7 Traversing a linked list

Traversing a singly linked list means visiting each node of a linked list until the end node is reached. In the case of a linked list, traversal can begin from the first node. The 'Start' pointer gives the address of the first node so that we can access the data part using the arrow operator. From the link part of the first node, we get the address of the second node. We can access the data and link part of the next node with this address. This method is continued until a NULL pointer is found in the link of a node.

For the sake of simplicity we will consider an example. We assume that we already have a linked list and we need to traverse this list. Not only do we need to traverse this list, we also have to calculate the total number of nodes in this list by traversing this linked list. The figure below derives the steps required for traversal operation in a linked list. It is assumed that Temp is a temporary pointer of 'node' type and Val is a variable to store the data read from a node.

1. Linked List having three nodes

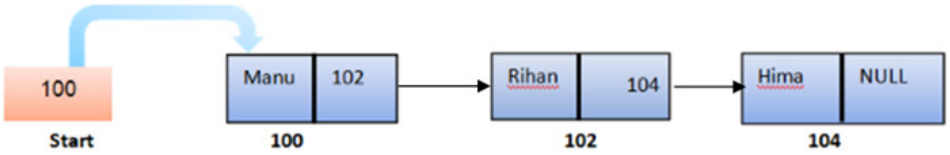


Fig. 2.1.19 (a) Linked List having three nodes.

2. The content of the start is copied into Temp (Because if we change the value of start, the address of the first node will be lost when traversing). Now, Temp can point to the first node.

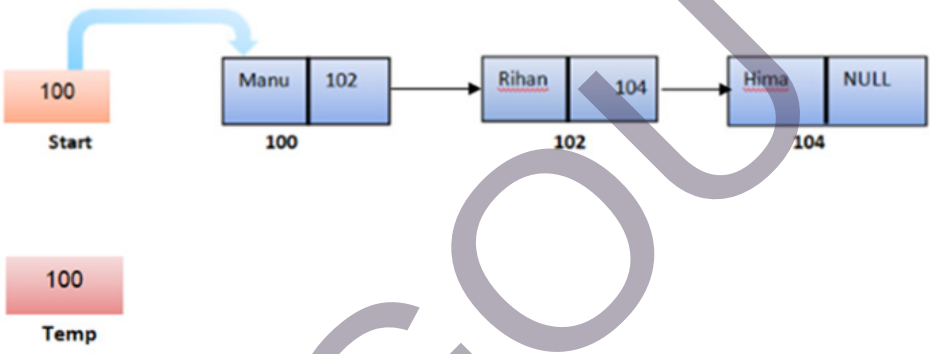


Fig. 2.1.19 (b) The content of the start is copied into Temp

3. First node is accessed using Temp and Data is retrieved to Val. Thereafter, Temp is updated by the address of the second node.

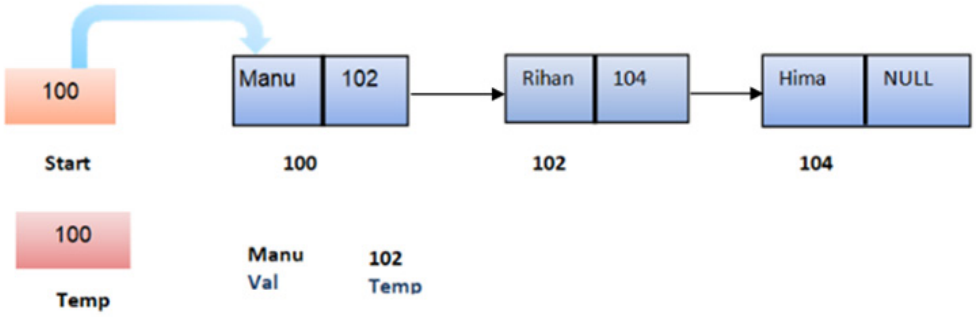


Fig. 2.1.19 (c) Temp is updated by the address of the second node

4. Second node is accessed using Temp and data is retrieved. Thereafter Temp is updated by the address of the third node.

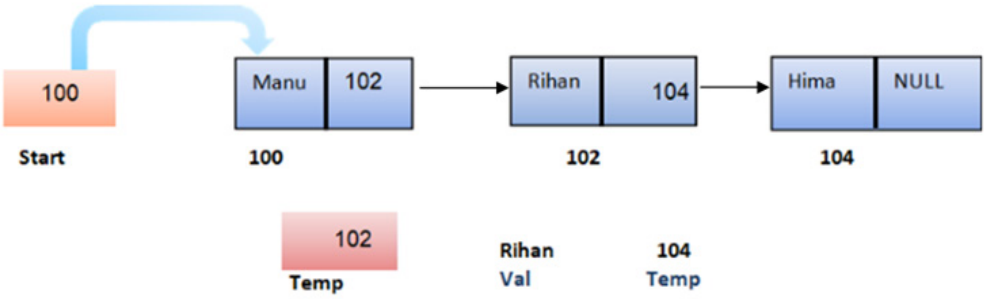


Fig. 2.1.19 (d) Temp is updated by the address of the third node.

5. Third node is accessed using value in Temp and data is retrieved to Val. Thereafter Temp is updated by the link part of the third node.Since it is NULL, traversal ends here.

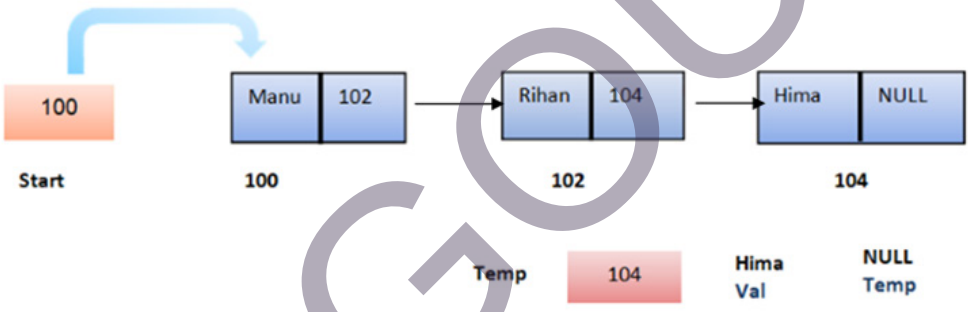


Fig. 2.1.19 (e) Showing traversal operation in a linked list

Algorithm for Traversal operation in linked list

Step 1: The address of the first node is taken from Start and stored in Temp.

Step 2: Using the stored address in Temp, find the data of the first or next node and store it in Val.

Step 3: The value of the link part of this node is found (or the address of the next node) and store it in Temp

Step 4: Repeat step 2 until content of Temp is not NULL; else stop.

The steps explained above are needed in the creation of a linked list if Start is not NULL. In that situation, the last node's address is necessary for storing the address of the new node in its link. The traversal operation begins from the first node and the address is stored in a temporary pointer variable (Temp in the figure). It will update by copying the content of the link to the node pointed by Temp. Traversal will continue until the link of a node pointed by Temp shows NULL value.

2.1.8 Applications

1. **Dynamic Memory Allocation-** Linked lists are widely used for dynamic memory allocation where memory is allocated or freed at runtime based on the program's needs. Unlike arrays, linked lists do not require a fixed size.

2. **Implementation of Stacks and Queues-** Stacks and queues, which are important data structures, are often implemented using linked lists to allow easy insertion and deletion of elements.

3. **Graph Representations-** Linked lists are used to represent graphs through adjacency lists. Each node in the graph can point to a linked list of its connected vertices.

4. **File and Directory Management-** Operating systems use linked lists to manage files and directories, especially when files are stored in scattered memory locations. Linked lists link file blocks together for easy access.

5. **Music and Image Playlists -** Circular linked lists are used in media players and image viewers where playlists or slideshows need to repeat continuously.

6. **Undo and Redo Operations -** Doubly linked lists are used in text editors, graphic editors, and other software to implement undo and redo functionalities by moving back and forth between previous and next states.

7. **Polynomial Arithmetic and Large Number Calculations -** Linked lists are useful for performing operations on large numbers and polynomials where terms or digits are processed dynamically.

Summarised Overview

A Linked List (LL) is a linear data structure consisting of nodes, where each node stores data and a pointer (link) to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, allowing efficient memory utilization and dynamic resizing. In SLL, traversal is possible only in the forward direction, starting from the head node until the last node, which points to NULL. Basic operations include insertion, deletion, traversal, and searching, each involving the adjustment of pointers rather than shifting elements. This makes SLL advantageous for applications requiring frequent insertions and deletions, although it has higher access time compared to arrays due to sequential access.

Assignments

1. Draw a linked list diagram for the following sequence: 5 → 15 → 25 → 35 → NULL.
2. Explain the differences between arrays and singly linked lists in terms of memory allocation and access speed.
3. Write pseudocode to traverse and print all elements in a singly linked list.
4. Modify the given linked list so that it becomes: START → [10 | Next] → [25 | Next] → [35 | NULL]. Draw the updated diagram.
5. Discuss one real-life scenario where a linked list would be preferred over an array.

Reference

1. Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed.). Prentice Hall.
2. Deitel, P. J., & Deitel, H. M. (2016). *C: How to program* (8th ed.). Pearson.
3. King, K. N. (2008). *C programming: A modern approach* (2nd ed.). W. W. Norton & Company.
4. Kochan, S. G. (2014). *Programming in C* (4th ed.). Addison-Wesley.
5. Perry, G., & Miller, D. (2013). *C programming absolute beginner's guide* (3rd ed.). Que Publishing.

Suggested Reading

1. Prata, S. (2014). *C primer plus* (6th ed.). Addison-Wesley.

2. Griffiths, D. (2015). *Head first C: A brain-friendly guide*. O'Reilly Media.

3. Oualline, S. (2003). *Practical C programming* (3rd ed.). O'Reilly Media.

4. Summit, S. (2013). *C programming FAQs: Frequently asked questions*. Addison-Wesley.

5. Oualline, S. (2000). *Programming in C in 7 days*. Sams Publishing

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT

Stacks And Queues

Learning Outcomes

After Completing this unit, the learner will be able to:

- describe the logical principles of stacks and queues
- demonstrate the working of stack and queue operations through step-by-step examples
- implement a circular queue using appropriate programming constructs

Background

Order and sequence are essential in many everyday situations—whether it's stacking plates in a kitchen or waiting in line at a billing counter. These simple, familiar actions mirror the fundamental principles of certain core data structures in computer science. Stacks operate on the principle of last-in, first-out (LIFO), much like the way plates are stacked, and are widely used in scenarios such as undo operations in text editors, expression evaluation, and navigating backward in web browsers. Queues, following the first-in, first-out (FIFO) principle, resemble the orderly progression of people in a queue and are applied in printer scheduling, call center systems, and process management in operating systems.

By understanding the mechanisms behind stacks and queues, we gain insight into how computers manage tasks and solve problems in an organized and efficient manner. This topic provides the foundation for exploring the logical structures and practical uses of stacks and queues, which support countless technologies in our daily lives.

Keywords

Push, Pop, Enqueue, Dequeue

Discussion

2.2.1 What is a Stack?

Stacks can be thought of as a pile of books, where you can only access the topmost book, or a stack of trays in a cafeteria. It is a linear data structure that follows the Last-In, First-Out (LIFO) principle, which means that the most recently added element is the first one to be removed. It is an Abstract Data Type (ADT) commonly supported by many programming languages due to its simplicity and usefulness. The term "stack" is inspired by real-world stacks, such as a pile of plates, where the last plate placed on top is the first to be taken off.

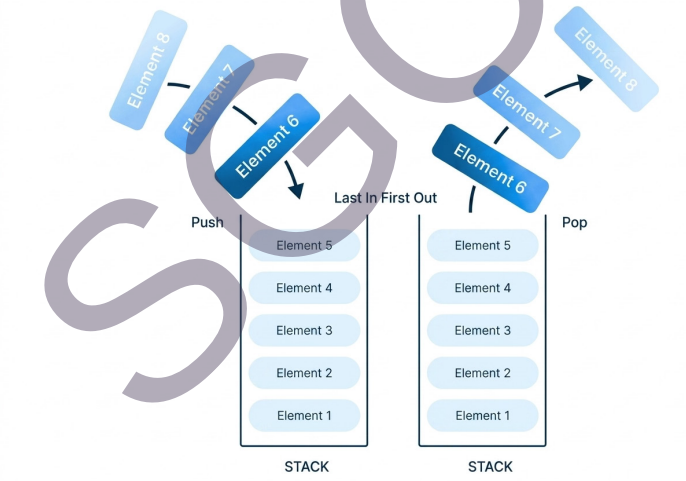


Fig. 2.2.1 Stack representation

An Abstract Data Type (ADT) describes what a data structure does, not how it is implemented.

In the case of a stack, the ADT defines the set of values it can store, the operations allowed, and the constraints (e.g., LIFO order), without tying it to arrays, linked lists, or any other concrete structure.

2.2.1.1 Mathematical Model of Stack ADT

We can model a stack formally as:

$$S = (D, S0, F)$$

Where:

D → Domain of elements (type of items the stack can hold, e.g., integers, characters, objects).

S0 → Initial state (usually an empty stack []).

F → Set of operations that can be performed on the stack.

The state of a stack at any given time is a sequence of elements:

$$S = [a_1, a_2, a_3, \ldots, a_n]$$

Here, a_n is the top element of the stack.

2.2.1.2 Basic Operations on Stack

To work with a stack, a defined set of basic operations is provided through its Abstract Data Type (ADT). These operations enable inserting, removing, and inspecting elements, as well as checking the status of the stack. This unit focuses on implementing a stack using an array. The basic operations that can be performed on a stack are given below:

1. Push – Add an element to the top of the stack.
2. Pop – Remove the top element from the stack.
3. Peek / Top – Returns the top element without removing it.
4. isEmpty – Check if the stack has no elements.
5. isFull – Check if the stack has reached its maximum size (in fixed-size stacks).

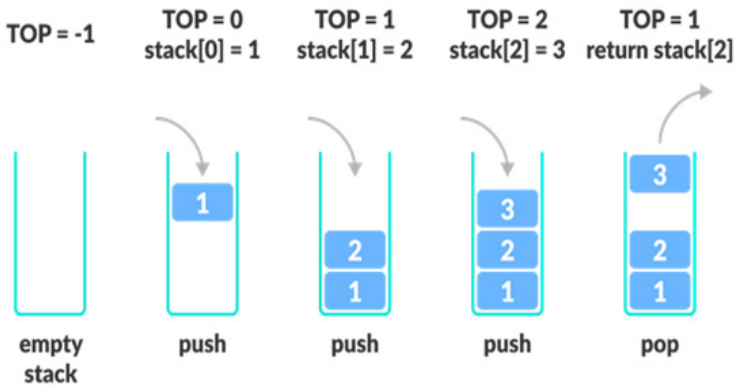


Fig. 2.2.2 Working of stack operations

Figure 2.2.2 illustrates the sequential functioning of stack operations. The TOP variable acts as a pointer to manage the elements within the stack. Initially, TOP is set to -1, indicating that the stack is empty. Each time an item is pushed onto the stack, TOP is incremented by one, and the new item is stored at this updated position. This continues as TOP takes the values 0, 1, and 2, representing the successful addition of three elements.

To remove an element, a POP operation is performed. This operation accesses the element at the TOP position, removes it, and then decrements TOP by one (for example, from 2 to 1). In this way, the most recently inserted element, Item3, is removed first. Repeated POP operations will remove all elements in reverse order of insertion.

Before carrying out a PUSH, it is essential to ensure the stack is not full, and before a POP, it must be confirmed that the stack is not empty.

Formal Definition of Stack Operations

Let Stack(D) represent the set of all possible stacks whose elements belong to domain D.

1. create( )

   Input: None

   Output: An empty stack

   Postcondition: S = [ ] (empty sequence)

2. push(S, x)

   Input: Stack S, element $x \in D$

   Output: Modified stack S'

Postcondition: S' = [a_1, a_2, …, a_n, x]
(element x is added to the top)

3. pop(S)

   Precondition: $S \neq [\ ]$ (stack is not empty)

   Output: A pair (top element, new stack)

Postcondition: S' = [a_1, a_2, …, $a_{(n-1)}$]
(last element is removed)

4. peek(S)

   Precondition: $S \neq [\ ]$ (stack is not empty)

   Output: a_n (top element) without removing it

5. isEmpty(S)

   Output: true if S = [ ], otherwise false

To implement stack, we need to maintain reference to the top item.

1. **Push Operation: push( )**

The push( ) operation is used to add elements to the top of the stack. Here's a simple and easy-to-follow algorithm that explains how the push( ) operation works.

Algorithm

1. Check if the stack is already full.
2. If it is full, display an error message and stop the operation. This is actually the isFull( ) operation.
3. If there's space, increase the top pointer to the next available position.
4. Insert the new element at the position pointed to by top.
5. Indicate that the push operation was successful.

2. **Pop operation: pop( )**

The pop( ) operation is used to remove the top element from the stack. Below is a clear and straightforward algorithm that illustrates how the pop( ) operation works.

Algorithm

1. Check if the stack is empty.
2. If it is empty, display an error message and stop the operation. In fact, this is the isEmpty( ) operation.
3. If there are elements in the stack, access the element at the position pointed to by top.
4. Optionally, store or display the removed element.
5. Decrease the top pointer by 1 to remove the element logically from the stack.
6. Indicate that the pop operation was successful.

3. **Peek operation: peek( )**

This operation returns the top element of the stack without removing it.

2.2.1.3 Implementation of Stack using Array

Algorithm

1. Define an array to serve as the underlying storage for the stack.

2. Use a variable (top) to keep track of the index of the topmost element in the stack.
3. Implement the push operation to insert elements at the position indicated by top.

```
The push( ) function in C
void push(int val) {
if (top == SIZE - 1)
 printf("Stack Overflow\n");  // Stack is full
else
stack[++top] = val;         // Move top and insert the value
}
```

Implement the peek operation to access the top element without removing it.

```
The peek( ) function in C
void peek( ) {
 if (isEmpty( ))
printf("Stack is empty\n");
else
 printf("Top element is: %d\n", stack[top]);
}
```

This function checks if the stack is empty using isEmpty( ). If not, it displays the element at the top of the stack. It does not modify the stack.

Ensure that all operations properly handle overflow (stack full) and underflow (stack empty) conditions.

C program to implement Stack

```
#include <stdio.h>
#define SIZE 5  // Define the size of the stack

int stack[SIZE]; // Stack array
int top = -1;     // Top is initialized to -1, indicating an empty stack

// Function to check if stack is empty
int isEmpty() {
   return top == -1;
}
```

```
// Function to check if stack is full
int isFull() {
    return top == SIZE - 1;
}

// Push operation
void push(int val) {
    if (isFull())
        printf("Stack Overflow\n");
    else {
        top++;
        stack[top] = val;
        printf("%d pushed to stack\n", val);
    }
}

// Pop operation
void pop() {
    if (isEmpty())
        printf("Stack Underflow\n");
    else {
        printf("%d popped from stack\n", stack[top]);
        top--;
    }
}

// Peek operation
void peek() {
    if (isEmpty())
        printf("Stack is empty\n");
    else

        printf("Top element is: %d\n", stack[top]);
}

// Display stack elements
void display() {
    if (isEmpty())
        printf("Stack is empty\n");
    else {
        printf("Stack elements: ");
        for (int i = 0; i <= top; i++)
            printf("%d ", stack[i]);
        printf("\n");
```

```
    }
}
// Main function with menu
int main() {
    int choice, value;

    while (1) {
        printf("\n--- Stack Menu ---\n");
        printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            pop();
            break;
        case 3:
            peek();
            break;
        case 4:
            display();
            break;
        case 5:
            return 0;
        default:
            printf("Invalid choice\n");
        }
    }
}
```

Stack variants include the double-ended stack (deque) for insertion and deletion at both ends, multi-stack implementation for storing multiple stacks in one array, circular stack for reusing storage in a wraparound manner, and thread-safe stack for synchronized access in multi-threaded environments.

2.2.1.4 Limitations of Stack

1. **Restricted Access:** Only the top element can be accessed or modified; other elements are not directly reachable.
2. **Fixed Size in Static Implementation:** In array-based stacks, the maximum size must be predefined, leading to possible overflow.
3. **Overflow and Underflow Conditions:** Overflow occurs when pushing onto a full stack, and underflow occurs when popping from an empty stack.
4. **Not Suitable for Random Access:** Unlike arrays, stacks cannot retrieve elements at arbitrary positions efficiently.
5. **Limited Flexibility:** Pure LIFO order may not suit all data processing needs.

2.2.2 Queues

Queues are linear data structures that represent a collection of elements arranged in a sequential order. They follow the First-In, First-Out (FIFO) principle, where the first element added to the queue is the first one to be removed. A queue operates much like a real-world line of people waiting at a ticket counter, those who arrive first are served first.

It is an Abstract Data Type (ADT) widely used in various applications such as scheduling tasks, managing printer queues, and in operating systems for process management. The structure is simple yet powerful, and it can be implemented using arrays or linked lists.

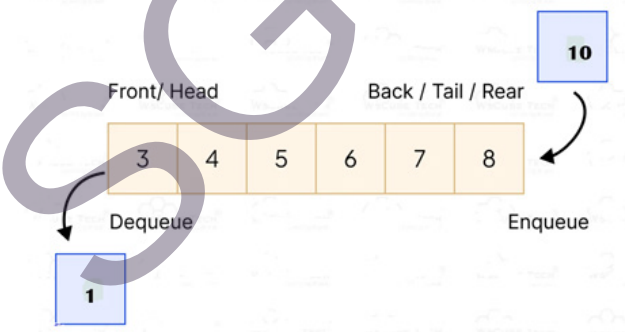


Fig.2.2.2 Queue representation

2.2.2.1 Mathematical Model of Queue ADT

We can formally define a queue as:

$$Q = (D, Q0, F)$$

Where:

D → Domain of elements (type of items the queue can hold, e.g., integers, characters, objects).

Q0 → Initial state (usually an empty queue []).

F → Set of operations that can be performed on the queue.

At any given time, the state of the queue can be represented as:

$$Q = [a_1, a_2, a_3, \ldots, a_n]$$

Here:

a_1 is the **front** element (first to be removed).

a_n is the **rear** element (last inserted).

2.2.2.2 Basic Operations on Queue

The basic operations that can be performed on a queue are:

1. Enqueue – Add an element to the rear of the queue.
2. Dequeue – Remove the element from the front of the queue.

Formal definition of Queue operations

1. **create( )**

   Input: None

   Output: An empty queue

   Postcondition: Q = [ ]

2. **enqueue(Q, x)**

   Input: Queue Q, element x ∈ D

   Output: Modified queue Q'

   Postcondition: Q' = [a_1, a_2, …, a_n, x]

3. **dequeue(Q)**

   Precondition: Q ≠ [ ] (queue is not empty)

   Output: Pair (front element, new queue)

   Postcondition: Q' = [a_2, a_3, …, a_n]

4. **peek(Q) / front(Q)**

   Precondition: Q ≠ [ ]

   Output: The element a1 (without removing it)

5. **isEmpty(Q)**

   Output: true if Q = [ ], otherwise false

To implement a queue, we need to maintain two references, the front and rear elements.

1. **Enqueue Operation: enqueue( )**

The enqueue( ) operation is used to add elements to the rear of the queue. Below is a simple algorithm that explains how the enqueue( ) operation works.

Algorithm

1. Initialize the queue:

   Initially: front = -1, rear = -1

2. Check if the queue is already full.
3. If it is full, display an error message and stop the operation.
4. If the queue is empty (front == -1), set front = 0.
5. Increment rear to point to the next available position.
6. Insert the new element at queue[rear].
7. Indicate that the enqueue operation was successful.

The enqueue( ) function in C

```
void enqueue(int value) {
if (rear == SIZE - 1) {
printf("Queue Overflow! Cannot insert %d\n",
value);
return;
}
if (front == -1) {
front = 0;
}
rear++;
queue[rear] = value;
printf("Enqueued %d successfully.\n", value);
}
```

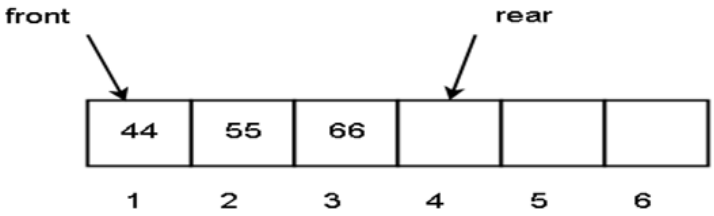


Fig.2.2.3 Queue before enqueue

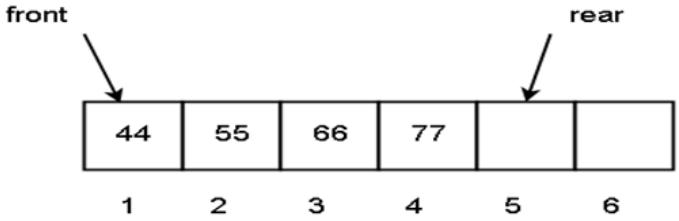


Fig.2.2.4 Queue after enqueue

2. Dequeue Operation: dequeue( )

The dequeue( ) operation is used to remove an element from the front of the queue. Here is the algorithm:

Algorithm

1. Check if the queue is empty.
2. If it is empty, display an error message and stop the operation. This is the isEmpty( ) check.
3. Access and display the element at queue[front].
4. If front == rear, reset both front and rear to -1 (queue becomes empty).
5. Otherwise, increment the front to remove the element logically.
6. Indicate that the dequeue operation was successful.

The dequeue( ) function in C

```
void dequeue( ) {
   if (front == -1) {
printf("Queue Underflow! Cannot remove element.\n");
return;
}
printf("Dequeued element: %d\n", queue[front]);
if (front == rear) {
front = rear = -1;  // Queue becomes empty
   }
```

```
else {
front++;  // Move front to next element
}
 printf("Dequeue operation successful.\n");
}
```

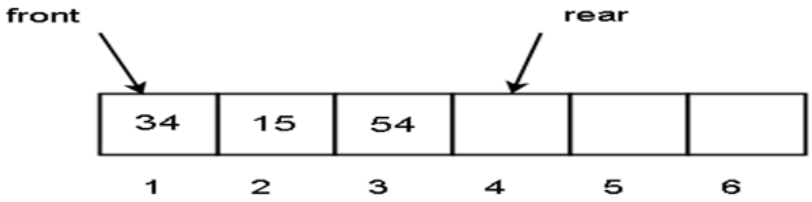


Fig.2.2.5 Queue before dequeue

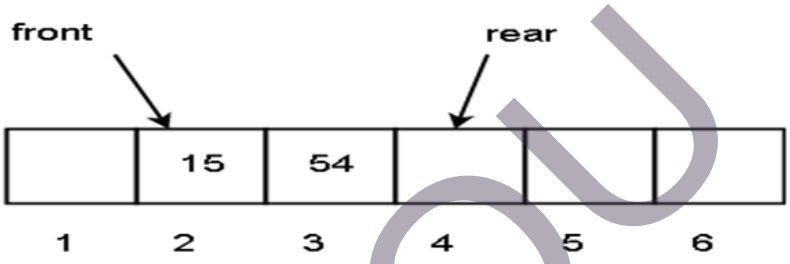


Fig. 2.2.6 Queue after dequeue

Now, you try to write a C program to implement the Queue data structure.

Note: This linear array implementation does not reuse freed space. To overcome this limitation, a Circular Queue is used.

2.2.3 Circular Queue

A circular queue is a type of queue that, despite being a linear data structure, operates in a circular manner. It still adheres to the FIFO (First-In, First-Out) principle, but when it reaches the end of its allocated space, it wraps around to the beginning, effectively using the available memory in a loop.

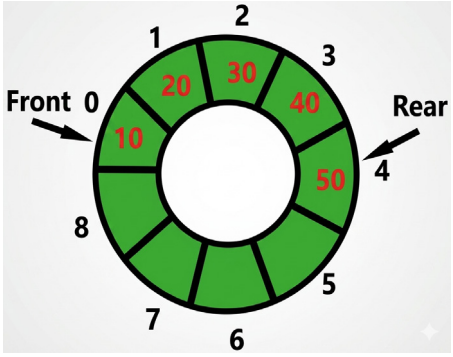


Fig. 2.2.7 Circular Queue

The circular queue structure effectively addresses the issue of unused space that occurs in a simple linear queue implemented using an array. In a linear array-based queue, the structure may appear full even when some positions are vacant. This happens when the rear pointer reaches the end of the array, despite some elements having been removed from the front, leaving empty slots at the beginning.

Figure 2.2.8 illustrates a graphical representation of a circular queue containing six elements, where the rear is at position 5 and the front at position 0. A circular queue operates on the FIFO (First In, First Out) principle, similar to a linear queue. However, unlike linear queues, the circular queue connects the last position back to the first, allowing continuous use of space.

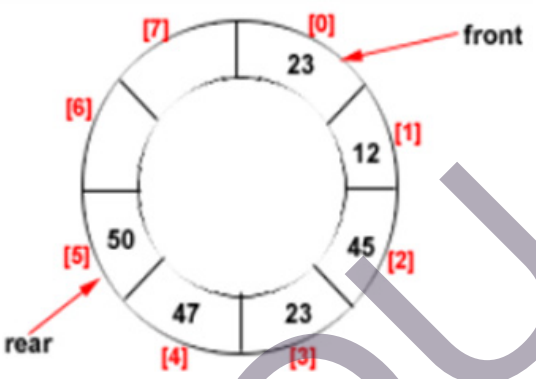


Fig. 2.2.8 Circular Queue having Rear = 5 and Front = 0

In the circular queue, there are two pointers, front and rear. The front pointer is used to get the front (first) element from the circular queue, and the rear pointer is used to get the rear (last) element from the circular queue. Thus, the queue has insertion and deletion operations; they are enqueue and dequeue.

The enqueue operation is used to insert the new value into the circular queue. The new element is always inserted from the rear end. This operator works as follows:

Step1: Check if the queue is full

Step2: For inserting the first element, set the value of the front to 0

Step3: Circularly increase the rear index value by 1

Step4: Add the new element in the position pointed to by the rear pointer.

The dequeue operation is used to delete an element from the circular queue. The deletion in a queue always takes place from the front end. This operator works as follows;

Step1: Check if the queue is empty

Step2: Return the value pointed by the front pointer

Step3: Circularly increase the front index value by 1

Step4: For the last element, reset the values of the front pointer and a rear pointer to -1

Enqueue and dequeue operations are based on the queue data insertion and deletion algorithms in the circular queue.

Below are the two algorithms used for inserting and deleting elements in a circular queue:

1. Circular Queue Insertion Algorithm

In this algorithm, the variables involved are CQueue, Rear, Front, N, and Item. CQueue refers to the circular queue used to store the data.Rear indicates the position where the new data item will be inserted.Front indicates the position from where a data item will be deleted.N is the maximum capacity of the circular queue.Item is the new data element to be added to the queue.

- Insert_CircularQ(CQueue, Rear, Front, N, Item)

  (Initially, set Rear = 0 and Front = 0)

Step 1: If Front = 0 and Rear = 0, then set Front = 1 and proceed to Step 4.

Step 2: If Front = 1 and Rear = N, or if Front = Rear + 1, print "Circular Queue Overflow" and return.

Step 3: If Rear = N, set Rear = 1 and go to Step 5.

Step 4: Increment Rear by 1 → Rear = Rear + 1.

Step 5: Assign the new item → CQueue[Rear] = Item.

Step 6: Return.

2. Circular Queue Deletion Algorithm

- Once the circular queue is populated with elements, the deletion operation can be performed. This process also uses CQueue, Rear, Front, N, and Item. CQueue is the circular queue containing data. Front is the location from which an element is to be removed. Rear is the location where new elements are inserted. Item will hold the deleted data from the front of the queue.

- Delete_CircularQ(CQueue, Front, Rear, Item)
  (Initially, set Front = 1)

Step 1: If Front = 0, print "Circular Queue Underflow" and return.

Step 2: Assign the front element → Item = CQueue[Front].

Step 3: If Front = N, set Front = 1 and return.

Step 4: If Front = Rear, set both Front and Rear to 0 and return.

Step 5: Increment Front by 1 → Front = Front + 1.

Step 6: Return.

The insertion and deletion procedures are often illustrated with a graphical representation of a circular queue. For example, consider a circular queue with five slots (N = 5). This visual helps in understanding how data elements are added and removed in a circular manner.

1. Initially, Rear = 0, Front = 0.

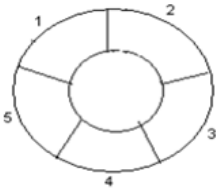


Fig. 2.2.9: illustration of step1

2. Insert 10, Rear = 1, Front = 1.

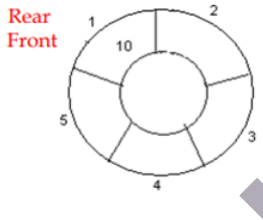


Fig. 2.2.10 illustration of step2

3. Insert 50, Rear = 2, Front = 1.

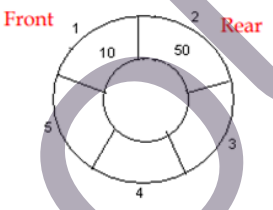


Fig. 2.2.11 illustration of step3

4. Insert 20, Rear = 3, Front = 1.

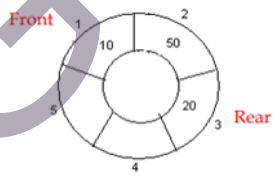


Fig. 2.2.12 illustration of step4

5. Insert 70, Rear = 4, Front = 1

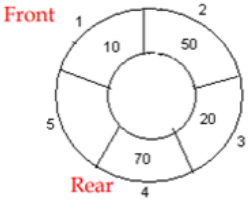


Fig. 2.2.13 illustration of step5

6. Delete Front(10), Rear = 4, Front = 2.

Fig. 2.2.14 illustration of step6

7. Insert 100, Rear = 5, Front = 2.

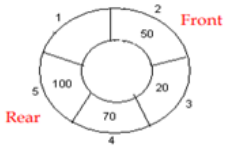


Fig. 2.2.15 illustration of step7

8. Insert 40, Rear = 1, Front = 2.

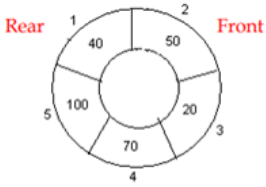


Fig. 2.2.16 illustration of step8

9. Insert 60, Rear = 1, Front = 2.

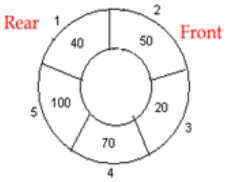


Fig. 2.2.17 illustration of step9

As Front = Rear + 1, so Queue overflow.

10. Delete Front (50), Rear = 1, Front = 3

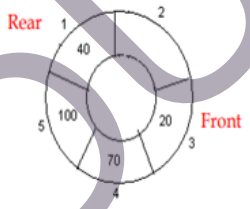


Fig. 2.2.18 illustration of step10

11. Delete Front(20), Rear = 1, Front = 4.

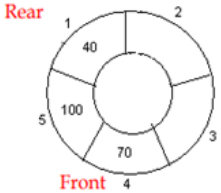


Fig. 2.2.19 illustration of step11

12. Delete Front(70), Rear = 1, Front = 5.

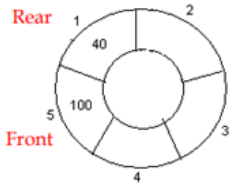


Fig. 2.2.20 illustration of step12

A traffic light sequence on a large roundabout is an example of a circular queue. It changes the signals circularly in a sequence with an equal interval of time. Some other real-time examples are print spooler of the operating system, bottle-capping systems in cold drink factories, a routine of human life and days in a week.

C program to implement Circular queue

```
#include <stdio.h>
#define SIZE 5

int queue[SIZE];
int front = -1, rear = -1;

// Check if the queue is full
int isFull() {
    return (front == 0 && rear == SIZE - 1) || (front == rear + 1);
}

// Check if the queue is empty
int isEmpty() {
    return front == -1;
}

// Enqueue function
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow! Cannot insert %d\n", value);
        return;
    }
    if (front == -1) {
        front = rear = 0;
    } else if (rear == SIZE - 1 && front != 0) {
        rear = 0;
    } else {
        rear++;
    }
    queue[rear] = value;
    printf("Enqueued %d successfully.\n", value);
}

// Dequeue function
void dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow! Cannot remove element.\n");
        return;
    }
    printf("Dequeued element: %d\n", queue[front]);
    if (front == rear) {
        front = rear = -1; // Queue becomes empty
    } else if (front == SIZE - 1) {
```

```
    front = 0;
    } else {
        front++;
    }
}

// Display function
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue elements: ");
    if (rear >= front) {
        for (int i = front; i <= rear; i++)
            printf("%d ", queue[i]);
    } else {
        for (int i = front; i < SIZE; i++)
            printf("%d ", queue[i]);
        for (int i = 0; i <= rear; i++)
            printf("%d ", queue[i]);
    }
    printf("\n");
}

// Main function (Menu-driven)
int main() {
    int choice, value;

    while (1) {
        printf("\n--- Circular Queue Menu ---\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value to enqueue: ");
            scanf("%d", &value);
            enqueue(value);
            break;
        case 2:
```

```
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice! Please try again.\n");
        }
    }
}
```

Summarised Overview

Stacks and queues are fundamental abstract data types (ADTs) that provide specific rules for data access. A stack operates on a Last-In, First-Out (LIFO) principle, like a pile of books, and its primary operations are push (adding an element to the top) and pop (removing the top element). A queue, in contrast, follows a First-In, First-Out (FIFO) principle, similar to a waiting line, with enqueue (adding to the rear) and dequeue (removing from the front) as its main operations. A circular queue is a variation of a regular queue that uses a fixed-size array and wraps around to the beginning when the end is reached, efficiently managing space and preventing the issue of unused gaps that can occur in a simple array-based queue. Both stacks and queues require careful handling of overflow (when the structure is full) and underflow (when it's empty) conditions to function correctly.

Assignments

1. Write a C program to implement a stack using arrays with push, pop, peek, and display functions.
2. Write a C program to implement a simple queue using arrays with enqueue, dequeue, and display functions.
3. Write a C program to implement a circular queue using arrays, including overflow and underflow checks.
4. Simulate a situation where a circular queue can avoid wastage of space compared to a linear queue.
5. Write a menu-driven program that allows the user to select stack, queue, or circular queue operations dynamically.

Reference

1. Rao, G. A. V. (2021). *Data structures and algorithms using C and C++*. PHI Learning.
2. Narayan, S. (2021). *Data structures and algorithms made easy: Data structure and algorithmic puzzles* (5th ed.). CareerMonk Publications.
3. Tiwari, K. (2021). *Data structures, algorithms and programming techniques in C*. BPB Publications.
4. Thareja, R. (2019). *Data structures using C* (2nd ed.). Oxford University Press.
5. Lipschutz, **S.** (2014). *Data Structures (Schaum's Outlines Series)*. McGraw-Hill Education.

Suggested Reading

1. Narayan, S. (2021). *Data structures and algorithms made easy: Data structure and algorithmic puzzles* (5th ed.). Career Monk Publications.

2. Lipschutz, S. (2014). *Data structures* (Schaum's outlines series, Revised ed.). McGraw-Hill Education.

3. Kruse, R. L., Leung, B. P., & Tondo, **C. L.** (1999). *Data structures and program design in C* (2nd ed.). Pearson Education.

4. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3
UNIT

Applications of Stack and Queue

Learning Outcomes

After completion of this unit, the learner will be able to:

- summarize the concepts and operations of stack and queue
- identify infix , prefix, postfix notations and its procedures
- explain how to convert infix, prefix, and postfix expressions
- describe how stacks are used in expression evaluation, recursion, and reversing data
- define queues applications such as CPU scheduling, printer spooling, and multithreading

Background

Stacks and queues are fundamental linear data structures that play a crucial role in computer science and software development. A stack follows the Last In First Out (LIFO) principle, where the last element added is the first to be removed. In contrast, a queue operates on the First In First Out (FIFO) principle, where the first element added is the first to be removed. These abstract data types are not only essential in algorithm design but also help solve a wide range of real-world computing problems efficiently.

One of the key applications of stacks lies in handling arithmetic expressions, especially when dealing with infix, prefix, and postfix notations. Stacks are used to convert expressions from one form to another and evaluate them effectively. Moreover, stacks are vital in recursion, as they maintain the function call history in programming languages. They are also used for reversing data, like strings

or arrays, which demonstrates their importance in algorithmic manipulation and control flow.

Queues, on the other hand, are commonly applied in scenarios involving task scheduling and data flow management. For instance, CPU scheduling and printer spooling rely on queues to maintain a proper execution sequence. In multithreading environments, queues manage tasks and resources between threads, ensuring efficient execution and avoiding conflicts. Understanding these applications provides students with a strong foundation in designing systems that are both robust and logically structured using these data structures.

Keywords

Infix, Prefix, Postfix, Expression Evaluation, Recursion, Reversing Data, CPU Scheduling, Printer Spooling, Multithreading

Discussion

The unit "Applications of Stack and Queue" explores how two fundamental data structures stack and queue are used to solve real-world problems in computer science. These structures form the backbone of many important processes such as expression handling, recursion, memory management, task scheduling, and data processing. By understanding how to apply stacks in operations like expression conversion and evaluation, and how queues are used in CPU scheduling, printer spooling, and multithreading, learners gain practical insights into designing efficient algorithms and managing system processes effectively. This unit builds a strong conceptual and application-oriented foundation that is essential for advanced topics in programming and system design.

2.3.1 Infix, Prefix and Postfix notations

In the field of computer science and mathematics, infix, prefix, and postfix notations are different ways of writing arithmetic expressions. While humans commonly use infix notation (e.g., A + B), it is not the most efficient for computers to evaluate due to the need for parentheses and operator precedence rules. To simplify expression evaluation, prefix (Polish) and postfix (Reverse Polish) notations are used, where operators are placed before or after operands, respectively. Understanding these notations is essential for learning how expressions are processed in compilers, calculators, and various algorithmic applications. This topic focuses on understanding these notations and learning how to convert expressions between them using stack-based algorithms, which

is a fundamental concept in compiler design and expression evaluation.

2.3.1.1 Infix Notation

Infix expressions are mathematical statements in which the operator is positioned between the operands. This notation is the standard convention employed by humans for expressing arithmetic operations.

In general, the expression "A + B" is an infix form where the addition operator "+" is placed between the two operands, A and B (fig 3.2.1). For example, the expression "5 + 2" is an infix form where the addition operator "+" is placed between the two operands, 5 and 2.

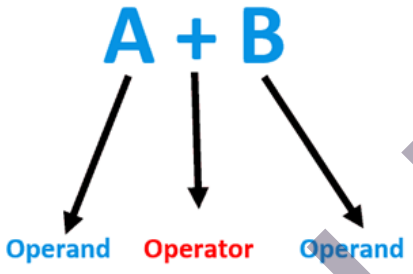


Fig 2.3.1: Infix Notation

While infix notation is intuitive and easily comprehensible for human readers, it presents challenges for computational evaluation. This complexity arises because the evaluation must respect the precedence and associativity rules of operators, and parentheses are utilized to explicitly define or alter the default order of operations. Infix expressions are evaluated according to operator precedence rules, which decide the sequence in which operations are carried out. i.e., multiplication and division take priority over addition and subtraction.

So, in the expression "3 + 2 * 4," the multiplication happens first, followed by the addition. Consider another example, in the expression "(5 + 2) * 3", the parentheses show that the addition must be carried out first, before multiplying by 3. Below is a table 2.3.1 that summarizes the precedence levels for common mathematical operators.

Table 2.3.1: Operator precedence levels

Operator	Order of Precedence
Parentheses ( )	1
Exponents ^	2
Division /, Multiplication *	3
Addition +, Subtraction -	4

Advantages of Infix Notation

Infix notation is commonly used in mathematics and programming because it clearly represents the order of operations, making expressions easier to read and understand. Some of the advantages of the infix notation are:

- Easy and natural for humans to read and write because it matches everyday math expressions.
- Intuitive to understand since operators are placed between operands (e.g., 2 + 3).
- Widely used and accepted in most mathematical texts and programming languages.

Disadvantages of Infix Notation

However, this notation has certain drawbacks. Complex expressions can become ambiguous, parentheses are often required to clarify the order of operations, and it is not directly suitable for computer-based evaluation, which makes processing expressions more challenging. Some of the disadvantages of the infix notation are:

- Can be complicated for computers to evaluate directly because of the need to consider operator precedence and associativity.
- Requires extra rules or symbols (like parentheses) to specify the order of operations clearly.
- Parsing infix expressions for computation is more complex compared to other notations like postfix or prefix.

2.3.1.2 Prefix Notation

Prefix expressions, commonly referred to as Polish notation, represent a form of mathematical notation in which the operator is positioned before its operands. This contrasts with the conventional infix notation, where the operator is situated between the operands.

In prefix notation, the operator precedes the operands; in general, the infix expression "A + B" is expressed as "+ A B" in prefix form (fig 3.2.2). For example, the infix expression "5 + 2" is expressed as "+ 5 2" in prefix form.

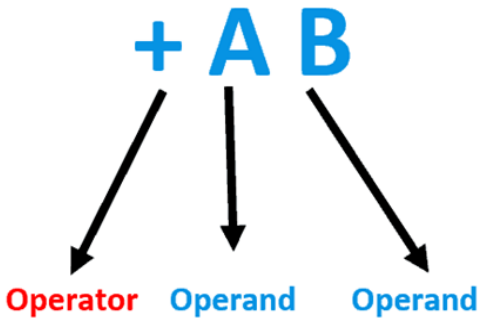


Fig 2.3.2: Prefix Notation

Advantages of Prefix Notation

The key advantages of Prefix Notation are:

- Eliminates the need for parentheses, as the order of operations is inherently clear from the position of operators and operands.
- Simplifies expression parsing and evaluation for computers, enabling straightforward and efficient processing.
- Reduces ambiguity in complex expressions since operator precedence is explicitly defined by notation structure.

Disadvantages of Prefix Notation

The main disadvantages of Prefix Notation are:

- Less intuitive and harder for humans to read and write compared to infix notation, which is more natural in everyday use.
- Requires familiarity and practice to interpret expressions correctly, making it less user-friendly for beginners.
- Rarely used outside of specific computer science or mathematical contexts, limiting its general applicability.

2.3.1.3 Postfix Notation

Postfix expressions, also referred to as Reverse Polish Notation (RPN), are a mathematical notation in which the operator is placed after its operands. This differs from the traditional infix notation, where the operator appears between the operands.

In postfix notation, the operands come first, followed by the operator. In general, the infix expression "A+ B" is represented as "AB +" in postfix form (fig 3.2.3). For example, the infix expression "5 + 2" is represented as "5 2 +" in postfix form.

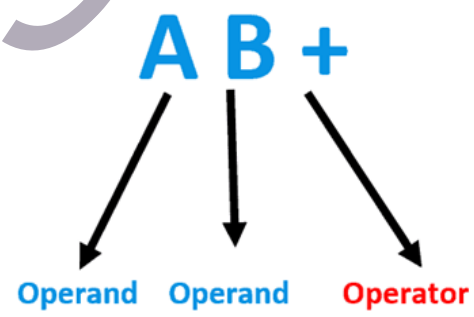


Fig 2.3.3: Postfix Notation

Advantages of Postfix Notation

Some of the advantages of Postfix Notation are:

- **No need for parentheses:** The order of operations is unambiguous, so parentheses are unnecessary.
- **Easy for computers to evaluate:** Postfix expressions can be efficiently processed using stacks, simplifying parsing and computation.
- **Eliminates ambiguity:** Operator precedence and associativity are inherently handled by the position of operators and operands.
- **Simplifies expression evaluation algorithms:** Postfix notation is widely used in compilers and calculators due to its straightforward evaluation process.

Disadvantages of Postfix Notation

Some of the key disadvantages of Postfix Notation are:

- **Less intuitive for humans:** It's not as natural or easy to read as infix notation, which most people learn first.
- **Requires learning curve:** People need practice to become comfortable reading and writing postfix expressions.
- **Rarely used in everyday math:** Mainly used in computing contexts rather than general mathematical writing or communication.

Table 2.3.2 shows further examples of infix expressions alongside their corresponding prefix and postfix forms. Make sure you grasp how these different notations represent the same order of operations.

Table 2.3.2: Examples of infix, prefix and postfix expressions

Infix expression	Prefix expression	Postfix expression
A + B * C	A * B C	A B C * +
(8 + 5) * 4	* + 8 5 4	8 5 + 4 *
A + B + C + D	+ + + + A B C D	A B + C + D +
(3 - 5) / (4 + 7)	/ - 3 5 + 4 7	3 5 - 4 7 + /

2.3.2 Conversions of Notations

Expressions in mathematics and computer programming can be represented in different forms, such as infix, prefix, and postfix notations. Since each notation follows different rules for arranging operators and operands, converting between these forms becomes

essential. Converting notations enables clearer interpretation and easier evaluation of expressions, especially in computers and calculators where the order of operations must be strictly followed. Learning how to convert these notations is important for understanding how expressions are processed internally and for designing efficient algorithms.

2.3.2.1 Infix to Postfix Expression

To convert infix expression to postfix expression, use the stack data structure. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence. Below are the steps to implement the above idea:

Step 1: Initialize an empty stack for operators and an empty list (or string) for the output postfix expression.

Step 2: Scan the infix expression from left to right:

- If the scanned character is an operand (like a variable or number), add it directly to the postfix output.
- If the scanned character is an opening parenthesis '(', push it onto the stack.
- If the scanned character is a closing parenthesis ')', pop from the stack and add to the postfix output until an opening parenthesis '(' is encountered on the stack. Remove the '(' from the stack but don't add it to the postfix output.
- If the scanned character is an operator (like +, -, *, /, ^):
  - ♦ While the stack is not empty, and the operator at the top of the stack has higher or equal precedence than the scanned operator, pop the operator from the stack and add it to the postfix output.
  - ♦ Push the scanned operator onto the stack.

Step 3: After the entire infix expression is scanned, pop all remaining operators from the stack and add them to the postfix output.

Example 1: Convert the below infix expression into a postfix expression.

A + B * C

Step	Character Scanned	Action	Stack	Postfix Output
1	A	Operand → Add to postfix	(empty)	A
2	+	Operator → Stack empty, push '+'	+	A
3	B	Operand → Add to postfix	+	A B
4	*	Operator → '*' *precedence (2)* > '+' *precedence (1), push* ' *'	+ *	A B

5	C	Operand → Add to postfix	+ *	A B C
End	-	End of expression → Pop all from stack and add to postfix	(empty)	A B C * +

Final Postfix expression: A B C * +

Activity 1 : Convert Infix to Postfix Expression

1. A*(B+C)/D
2. a+b*(c^d-e)^(f+g*h)-i
3. 7-2*3
4. 6+(4*1)/7

2.3.2.2 Infix to Prefix Expression

Begin by reversing the infix expression and swapping every '(' with ')' and vice versa. Then, apply a stack-based conversion process, following operator precedence and associativity, to transform the updated expression into postfix form. Lastly, reverse the postfix result to produce the prefix notation. Below are the steps for the conversion of infix to prefix expression:

Step 1: Reverse the infix expression

- Reverse the string from left to right.
- Swap each opening parenthesis '(' with a closing parenthesis ') ' and vice versa.

Step 2: Convert the modified expression to postfix

- Use the stack-based infix-to-postfix algorithm, respecting operator precedence and associativity.

Step 3: Reverse the postfix expression

- The reversed postfix expression is the **prefix notation**.

Example 2: Convert the below infix expression into prefix expression.

(A + B) * (C - D)

Step 1: Reverse and swap parentheses.

Original: (A + B) * (C - D)

Reversed:)D - C(*)B + A(

Swap parentheses: (C - D) * (A + B)

Step 2: Convert to postfix.

We now convert (C - D) * (A + B) → Postfix using stack rules.

Symbol Read	Action / Rule Applied	Stack Content	Postfix Output
(	Push to stack	(	
C	Operand → Add to output	(	C
-	Push to stack	(-	C
D	Operand → Add to output	(-	C D
)	Pop until (		C D -
*	Push to stack	*	C D -
(	Push to stack	* (	C D -
A	Operand → Add to output	* (	C D - A
+	Push to stack	* (+	C D - A
B	Operand → Add to output	* (+	C D - A B
)	Pop until (	*	C D - A B +
End	Pop remaining stack		C D - A B + *

Postfix Result: C D - A B + *

Step 3: Reverse Postfix to get Prefix

Final Prefix expression: * + A B - C D

Activity 2: Infix to Prefix Expression

1. A + B * C
2. (X - Y) / (M + N)
3. P * (Q + R) - S / T
4. (3 + 2) * (8 - 6) / 7

2.3.2.3 Postfix to Infix Expression

Postfix to infix conversion involves transforming expressions where operators follow their operands (postfix notation) into standard mathematical expressions with operators

placed between operands (infix notation). This conversion improves readability and understanding. Below are the steps to implement the above idea:

Step 1: Initialize an empty stack for storing intermediate infix expressions.

Step 2: Scan the postfix expression from left to right.

Step 3: If the symbol is an operand, push it onto the stack.

Step 4: If the symbol is an operator:

- Pop the **top two elements** from the stack.

  (Let the first popped element be operand2 and the second popped element be operand1).

- Form a string: **(operand1 operator operand2).**
- Push this new string back onto the stack.

Step 5: Repeat until all symbols are processed.

Step 6: The final element in the stack is the required infix expression.

Example 3: Convert the below postfix expression to infix expression.

ABC*+

Step	Symbol Read	Action Taken	Stack Content
1	A	Operand → Push	A
2	B	Operand → Push	A, B
3	C	Operand → Push	A, B, C
4	*	Operator → Pop C, B → Form (B*C) → Push	A, (B*C)
5	+	Operator → Pop (B*C), A → Form (A+(B*C)) → Push	(A+(B*C))

Final Infix Expression: (A + (B * C))

Activity 3: Postfix to Infix Expression

1. abc+*d/
2. 93-42+*
3. AB+CD - / EF * -
4. PQ + RS -*T /

2.3.2.4 Prefix to Infix Expression

The goal is to rearrange the prefix expression into infix form, where operators are placed between operands while preserving the original order of operations. Computers usually do the computation in either prefix or postfix (usually postfix). But for humans, it's easier to understand an Infix expression rather than a prefix. Hence conversion is needed for human understanding. Below are the steps to implement the above idea:

Step 1: Initialize an empty stack to store intermediate infix expressions.

Step 2: Scan the prefix expression from right to left (reverse order).

Step 3: If the symbol is an operand, push it onto the stack.

Step 4: If the symbol is an operator:

- Pop the top two elements from the stack.
  (Let the first popped element be operand1 and the second popped element be operand2.)
- Form a string: **(operand1 operator operand2).**
- Push this new string back onto the stack.

Step 5: Repeat until all symbols are processed.

Step 6: The final element in the stack is the required infix expression.

Example 4: Convert the below prefix expression to infix expression

+ A * B C

Step	Symbol Read	Action Taken	Stack Content
1	C	Operand → Push	C
2	B	Operand → Push	C, B
3	*	Operator → Pop B, C → Form (B*C) → Push	(B*C)
4	A	Operand → Push	(B*C), A
5	+	Operator → Pop A, (B*C) → Form (A+(B*C)) → Push	(A+(B*C))

Final Infix Expression: (A + (B * C))

Activity 4: Prefix to Infix Expression

1. -+ABC
2. *+UV+XY
3. –A/B/*CDE
4. +*PQ / RS

2.3.2.5 Postfix to Prefix Expression

The goal here is to take a postfix expression and rearrange it so the operators come before operands without changing the computation order. Below are the steps to implement the above idea:

Step 1: Initialize an empty stack to store intermediate prefix expressions.

Step 2: Scan the postfix expression from left to right.

Step 3: If the symbol is an operand, push it onto the stack.

Step 4: If the symbol is an operator:

- Pop the top two elements from the stack.
  (Let the first popped element be operand2, second popped element be operand1).
- Form a new string: **operator operand1 operand2.**
- Push this new string back onto the stack.

Step 5: Repeat until all symbols are processed.

Step 6: The final element in the stack is the prefix expression.

Example 5: Convert the given Postfix to Prefix Expression.

ABC*+

Step	Symbol Read	Action Taken	Stack Content
1	A	Operand → Push	A
2	B	Operand → Push	A B
3	C	Operand → Push	A B C
4	*	Operator → Pop C, B → Form *BC → Push	A *BC
5	+	Operator → Pop *BC, A → Form +A*BC → Push	+A*BC

Final Prefix Expression: + A * B C

Activity 5: Postfix to Prefix Expression.

1. abc+*d/
2. 93-42+*
3. AB+CD - / EF * -
4. PQ + RS -*T /

2.3.2.6 Prefix to Postfix Expression

Prefix to Postfix conversion refers to the process of changing an expression written in prefix form, where the operator precedes the operands, into postfix form, where the operator follows the operands. This transformation is useful because postfix expressions can be evaluated easily using stack-based techniques without the need for parentheses or precedence rules. Understanding the conversion process helps in developing efficient algorithms for parsing and evaluating mathematical expressions. Below are the steps to implement the above idea:

Step 1: Initialize an empty stack to store intermediate postfix expressions.

Step 2: Scan the prefix expression from right to left.

Step 3: If the symbol is an operand, push it onto the stack.

Step 4: If the symbol is an operator:

- Pop the top two elements from the stack.
  (Let the first popped element be operand1 and the second be operand2.)
- Form a string: **operand1 operand2 operator.**
- Push this string back onto the stack.

Step 5: Repeat until all symbols are processed.

Step 6: The final element in the stack is the postfix expression.

Example 6: Convert the given Prefix to Postfix Expression

+ A * B C

Step	Symbol Read	Action Taken	Stack Content
1	C	Operand → Push	C
2	B	Operand → Push	C, B
3	*	Operator → Pop B, C → Form BC* → Push	BC*
4	A	Operand → Push	BC*, A
5	+	Operator → Pop A, BC* → Form A BC* + → Push	ABC*+

Final Postfix Expression: ABC*+

Acitivity 6: Prefix to Postfix Expression

1. -+ABC
2. *+UV+XY
3. –A/B/*CDE
4. +*PQ / RS

Infix, prefix, and postfix notations provide different ways to represent and evaluate mathematical expressions, each with its own advantages and applications. While infix notation is the most natural and widely used by humans, prefix and postfix notations are more efficient for computer processing as they eliminate the need for parentheses and complex precedence rules. The ability to convert between these notations is essential in fields such as compiler design, expression evaluation, and stack-based algorithms. A clear understanding of these conversions not only enhances problem-solving skills but also forms the foundation for learning advanced concepts in data structures and programming.

2.3.3 Applications of Stack

The stack is a fundamental linear data structure that follows the Last-In-First-Out (LIFO) principle, making it highly useful in various programming and computational tasks. In this topic, we explore key applications of stacks such as expression conversion and evaluation involving infix, prefix, and postfix notations, as well as recursion handling, where function calls are managed using an internal stack. Stacks also play a vital role in tasks like reversing data, including strings or arrays, and managing memory allocation in program execution. These applications highlight the versatility and importance of stacks in building efficient and well-structured programs.

2.3.3.1 Expression evaluation

Expression evaluation is the process of computing the result of a mathematical expression containing operands (numbers or variables) and operators (such as +, -, *, /). Expressions can be written in different notations:

- Infix Notation: Operator between operands (e.g., A + B)
- Prefix Notation: Operator before operands (e.g., + A B)
- Postfix Notation: Operator after operands (e.g., A B +)

Among these, infix notation is most familiar but difficult for computers to evaluate directly because of operator precedence and parentheses. Postfix and prefix notations, on the other hand, can be easily evaluated using stacks. Stacks follow the Last-In-First-Out (LIFO) principle, which makes them perfectly suited for problems where you need to keep track of recent items such as operators or operands and resolve them in the

correct order. Some key rules are:

- When evaluating postfix expressions, operands are pushed onto the stack until an operator is found.
- Operators pop the correct number of operands from the stack, apply the operation, and push the result back.
- This mechanism naturally respects the correct order of operations without the need for parentheses or precedence rules during evaluation.

Basic methods for the expression evaluation using stack are:

a. Postfix Expression Evaluation Algorithm

Postfix expressions (also called Reverse Polish Notation) are evaluated from left to right. The steps for postfix expression evaluation algorithm as follows:

1. Initialize an empty stack.
2. Scan the postfix expression from left to right.
3. For each token:
   - If it is an operand (number/variable), push it onto the stack.
   - If it is an operator:
     - Pop the required number of operands from the stack (for binary operators, pop two operands).
     - Apply the operator to the operands in the correct order.
     - Push the result back onto the stack.
4. After scanning the entire expression, the stack contains the final result.

Example 7: Evaluate the postfix expression **5 6 2 + * 12 4 / -**

Token	Action	Stack
5	Push operand	5
6	Push operand	5, 6
2	Push operand	5, 6, 2
+	Pop 2 and 6, add (6 + 2 = 8)	5, 8
*	Pop 8 and 5, multiply (5 * 8=40)	40
12	Push operand	40, 12
4	Push operand	40, 12, 4
/	Pop 4 and 12, divide (12/4=3)	40, 3
-	Pop 3 and 40, subtract (40 - 3=37)	37

b. Infix to Postfix Conversion Using Stack

Before evaluating an infix expression, it can be converted to postfix form using a stack to

handle operator precedence and parentheses. (Already explained in the section 2.3.2.1). Advantages of Using Stack for Expression Evaluation are:

- Handles operator precedence and associativity naturally.
- Simplifies evaluation of expressions with parentheses.
- Efficient and easy to implement.
- Works well with postfix and prefix expressions which eliminate ambiguity of parentheses.
- Enables compilers and interpreters to evaluate expressions correctly.

2.3.3.2 Recursion

Recursion is a method in programming where a function calls itself either directly or indirectly to solve a problem. It works by breaking a larger problem into smaller sub-problems until it reaches a base case, which is the stopping condition. Behind the scenes, recursion relies heavily on the stack data structure, specifically the call stack, to manage the sequence of function calls.

When a function is called recursively, the system pushes a stack frame onto the call stack. This frame stores:

- Parameters passed to the function
- Local variables
- Return address (location in the program to resume after the call finishes)

Each new recursive call pushes another stack frame, preserving the current state before diving deeper into the problem. Once the base case is reached, functions begin to return in reverse order, popping stack frames one by one. This Last-In-First-Out (LIFO) nature of stacks perfectly matches the way recursion unwinds. However, if recursion goes too deep without a base case, the stack can run out of memory, causing a stack overflow. This is why designing a proper base case and controlling recursion depth is important.

Example 8: Factorial Calculation Using Recursion

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0)
        return 1;                          // Base case
    else
        return n * factorial(n - 1);       // Recursive call
}

int main() {
    int num = 5;
    printf("Factorial of %d = %d", num, factorial(num));
    return 0;
}
```

Detailed step-by-step of stack behavior for factorial(5) using recursion explained below.

Call	Action on Stack
factorial(5)	Push frame for n=5
factorial(4)	Push frame for n=4
factorial(3)	Push frame for n=3
factorial(2)	Push frame for n=2
factorial(1)	Push frame for n=1
factorial(0)	Push frame for n=0 → Base case returns 1

Unwinding the Stack:

- factorial(0) returns 1 → Pop frame
- factorial(1) returns 1 × 1 = 1 → Pop frame
- factorial(2) returns 2 × 1 = 2 → Pop frame
- factorial(3) returns 3 × 2 = 6 → Pop frame
- factorial(4) returns 4 × 6 = 24 → Pop frame
- factorial(5) returns 5 × 24 = 120 → Pop frame

Final Output: Factorial of 5 = 120

Recursion works only because of the stack's ability to store and restore the state of each function call. Each recursive call pushes a new frame with its own variables, and results are returned by popping frames in reverse order. This makes recursion elegant and manageable but also potentially risky if the stack size limit is exceeded due to too many recursive calls.

2.3.3.3 Reversing data

Reversing data means changing the order of elements so that the last element becomes the first, and the first becomes the last. This process naturally matches the Last-In-First-Out (LIFO) property of a stack, making it one of the simplest and most efficient tools for reversal tasks. The basic rules are as follows:

- Push each element of the data (string, array, list, etc.) onto the stack.
- Since a stack stores items in LIFO order, popping elements one by one will return them in reverse order of insertion.
- Store or display these popped elements to get the reversed data.

Stacks are especially useful for reversing:

- Strings
- Arrays
- Linked lists
- Sequences read from files or streams

Example 9: Reversing a String Using Stack

```
#include <stdio.h>
#include <string.h>

#define MAX 100

char stack[MAX];
int top = -1;

void push(char c) {
    stack[++top] = c;
}

char pop() {
    return stack[top--];
}

int main() {
    char str[MAX] = "STACK";
    int i;

    // Push all characters of the string onto the stack
    for (i = 0; i < strlen(str); i++) {
        push(str[i]);
    }

    // Pop and print characters in reverse order
    printf("Reversed string: ");
    while (top != -1) {
        printf("%c", pop());
    }

    return 0;
}
```

Output:
Reversed string: KCATS

Example 10: Reversing an Array Using Stack

When elements of an array are pushed into a stack, they are stored in the order of insertion. Later, when the elements are popped from the stack, they come out in the reverse order, thereby reversing the original array. The algorithmic steps as follows:

1. Create an empty stack.
2. Traverse the array and push all elements into the stack.
3. Once all elements are pushed, pop elements from the stack one by one.
4. Store the popped elements back into the array.
5. The array will now be reversed.

Suppose the array is: A = [10, 20, 30, 40, 50]

Step 1: Push elements into stack → [10, 20, 30, 40, 50]
Step 2: Pop elements and put back into array → [50, 40, 30, 20, 10]

Final reversed array = [50, 40, 30, 20, 10]

```
#include <stdio.h>
#define MAX 100

int stack[MAX], top = -1;

void push(int x) {
   stack[++top] = x;
}

int pop() {
   return stack[top--];
}

int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr) / sizeof(arr[0]);

   // Push all elements into stack
   for(int i = 0; i < n; i++) {
      push(arr[i]);
   }

   // Pop from stack and put back in array
   for(int i = 0; i < n; i++) {
      arr[i] = pop();
```

```
}

    printf("Reversed Array: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:

Reversed Array: 50 40 30 20 10

The stack's LIFO property makes it a perfect choice for reversing data structures. By pushing all elements onto the stack and popping them back, we can reverse the order easily without complex algorithms. This method is simple, efficient, and widely used in applications like undo operations, backtracking, and text editors.

2.3.4 Applications of Queue

A queue is a fundamental linear data structure that operates on the First-In-First-Out (FIFO) principle, meaning the element that enters first is processed first. It consists of two main operations: enqueue for inserting elements at the rear end and dequeue for removing elements from the front end. This structure ensures an orderly and fair processing of data, making it highly suitable for situations where tasks must be handled in the exact order they arrive.

In computer science and real-world systems, queues have a wide range of applications due to their simple yet effective data handling mechanism. They are used in operating system process scheduling, print spooling, data buffering, network traffic management, and customer service systems. By preserving the sequence of tasks, queues help maintain efficiency, avoid conflicts, and ensure that resources are allocated in an organized manner.

2.3.4.1 CPU scheduling

In a multitasking operating system, multiple processes compete for CPU time. CPU Scheduling is the method used by the operating system to decide the order in which processes are executed. A queue is the ideal data structure for this purpose because it follows the First-In-First-Out (FIFO) principle or can be adapted for priority-based and time-sharing scheduling.

When a process is ready to be executed but the CPU is busy, it is placed into a ready queue. This queue holds all processes in the "ready" state, waiting for CPU allocation. The CPU scheduler selects processes from the queue according to the chosen scheduling algorithm. Depending on the policy, the queue can behave differently as a simple FIFO

queue, a priority queue, or even a circular queue (for round robin scheduling).

How Queue Works in CPU Scheduling

In CPU scheduling, a queue is used to manage processes that are ready for execution but are waiting for the CPU to become available. These processes are placed in a ready queue, where they wait in an organized manner until the scheduler selects one based on the chosen scheduling algorithm. The queue ensures that processes are handled in a systematic way whether it follows First-In-First-Out (FIFO) for fair processing, a priority queue for urgent tasks, or a circular queue for time-sharing in round robin scheduling. This structured approach helps the operating system efficiently allocate CPU time and maintain smooth multitasking. Some important key points are:

1. **Process Arrival:** When a process arrives, it is inserted into the ready queue.
2. **Waiting for Execution:** Processes wait in the queue until the CPU becomes available.
3. **Scheduling Decision:** The CPU scheduler picks the next process from the queue based on the scheduling algorithm:
    - FCFS (First Come First Served): FIFO queue.
    - SJF (Shortest Job First): Priority queue with shortest burst time first.
    - Priority Scheduling: Priority queue based on assigned priority values.
    - Round Robin: Circular queue with a fixed time slice.
4. **Execution:** The selected process is executed by the CPU.
5. **Completion or Return:** After execution, the process either terminates or is placed back into the queue (if incomplete in time-slicing).

Example 11: Round Robin Scheduling with a Circular Queue

Let's say we have three processes:

P1: Burst time = 5 units

P2: Burst time = 3 units

P3: Burst time = 6 units

Time Quantum: 2 units

Step-by-step Execution:

1. Enqueue P1, P2, P3 in the ready queue.
2. P1 runs for 2 units → Remaining = 3 → Enqueue back.
3. P2 runs for 2 units → Remaining = 1 → Enqueue back.

4. P3 runs for 2 units → Remaining = 4 → Enqueue back.
5. Continue the cycle until all processes are completed. This ensures fairness, as each process gets equal CPU time in turns.

Some advantages of Using Queue in CPU Scheduling are:

- **Fairness:** Ensures every process gets a chance for execution.
- **Orderly Execution:** Maintains arrival order or priority.
- **Flexibility**: Supports various scheduling algorithms.
- **Efficient Resource Management:** Minimizes idle CPU time.

Queues are central to CPU scheduling in operating systems, acting as the storage structure for processes waiting for execution. By adapting the queue type (FIFO, priority, circular), the OS can implement different scheduling strategies to balance performance, fairness, and response time. Without queues, managing process execution in a multitasking environment would be inefficient and prone to conflicts.

2.3.4.2 Printer spooling

Printer spooling is a process in computing where print jobs are managed through a temporary storage area called the "spool." This method allows users to send documents to the printer without having to wait for the printing to finish. Spooling is especially useful in multi-user environments such as offices, schools, or print shops where several people might send print commands simultaneously. Instead of sending each job directly to the printer, which can only handle one task at a time, the system queues these jobs in a spool. The printer then processes each job one-by-one in the order they were received.

The concept of a *queue* plays a central role in printer spooling. A queue is a First-In, First-Out (FIFO) data structure where elements are added at the rear (enqueue) and removed from the front (dequeue). In the context of spooling, each print job is enqueued as soon as a user sends it. The printer works like a consumer that dequeues the job at the front and begins printing it. Once finished, it moves to the next job in the queue. This process continues until the queue is empty. This method ensures fair access to printing resources and prevents confusion or overlap in output.

To relate this to a real-world scenario, imagine a coffee shop where customers place their orders at the counter. The barista prepares one drink at a time, in the exact order that customers placed their orders. If ten customers arrive, the first customer to place an order gets their coffee first. Similarly, in printer spooling, the first print job gets printed before the second, and so on. This avoids conflict, maintains order, and ensures no job is skipped.

Internally, printer spooling can be implemented using a queue data structure. Operating systems like Windows use print spooler services to manage this queue. Each job contains metadata such as the user ID, number of pages, file format, and priority. Advanced spooling systems may allow reordering, pausing, or deleting jobs from the queue. The

implementation might involve a software thread that continuously monitors the spool and sends jobs to the printer once it is available.

2.3.4.3 Multithreading

Multithreading is a programming and operating system concept where multiple threads (lightweight sub-processes) are executed concurrently within a single process. Threads share the same memory space but run independently. This allows programs to perform multiple tasks simultaneously, making them faster and more responsive. For example, in a web browser, one thread might load a webpage while another plays a video, and a third handles user inputs all running concurrently.

In multithreaded applications, threads often need to communicate or coordinate with each other. This is where queues come in. A queue acts as a buffer or bridge between two threads, usually a producer thread that generates data or tasks, and a consumer thread that processes them. The producer places data into the queue (enqueue), and the consumer removes data (dequeue) when it is ready to process it. This model ensures decoupling, meaning the producer and consumer do not need to run at the same speed or time.

Producer-Consumer Problem

A classic problem in multithreading is the Producer-Consumer problem, which is solved efficiently using queues. Suppose a producer thread keeps generating tasks (like downloading files), and a consumer thread processes them (like saving to disk). If the consumer is busy, the queue holds the incoming tasks temporarily. This prevents data loss and keeps both threads functioning independently. It also avoids race conditions and deadlocks by controlling access to shared resources through thread-safe queues.

Real-Life Analogy

Think of a newspaper factory. Reporters (producers) write news articles and submit them to an editorial board (queue). Editors (consumers) pick up articles from the tray and edit them before sending them to print. Reporters and editors work at their own pace, but the queue ensures smooth flow and communication between them.

Applications of Multithreading with Queues

- **Web Servers:** Handle multiple HTTP requests at once.
- **Chat Applications:** Simultaneously receive and send messages.
- **Background Tasks:** Run tasks like file scanning or system updates without freezing the UI.
- **Download Managers:** Queue multiple downloads while processing them concurrently.
- **Real-Time Processing Systems:** Process sensor data, log messages, or stream audio/video.

In conclusion, stacks and queues are fundamental data structures that play a vital role in solving a wide range of computational problems efficiently. Stacks, with their Last-In-First-Out (LIFO) property, are essential for tasks such as expression evaluation, handling different notations and their conversions, managing recursive function calls, and reversing data. Queues, with their First-In-First-Out (FIFO) property, are equally important in applications like CPU scheduling, printer spooling, and managing tasks in multithreaded environments, ensuring fairness and order in processing. Understanding these applications not only enhances problem-solving skills but also builds a strong foundation for designing efficient algorithms and systems in both academic and real-world contexts.

Summarised Overview

The unit explores the applications of stack data structure, focusing on its use in expression notations such as infix, prefix, and postfix, and the conversions between these forms using stacks to handle operator precedence and associativity. Stacks are also crucial in expression evaluation, where they efficiently process postfix and prefix expressions, and in recursion, where the call stack manages multiple function calls by storing parameters and return addresses. Additionally, stacks are used to reverse data like strings and arrays by leveraging their Last In First Out (LIFO) behavior. The unit further covers the applications of queue, which operates on a First In First Out (FIFO) basis, ensuring ordered and fair processing. Queues are fundamental in CPU scheduling, where processes wait in a ready queue for CPU time, supporting various scheduling algorithms like FCFS and Round Robin. They also manage tasks in printer spooling by lining up print jobs and in multithreading by organizing threads for efficient execution. Together, stacks and queues form essential building blocks in designing efficient algorithms and system processes.

Assignments

1. Explain in detail the steps involved in converting an infix expression to postfix and prefix forms. Illustrate each conversion with a suitable example and stack operations trace.

2. Describe the algorithm for evaluating a postfix expression using a stack. Apply the algorithm to evaluate the expression 6 2 / 3 – 4 2 * + showing the stack content after each step.

3. Discuss how recursion is implemented internally using a stack. Explain with the help of a recursive function (e.g., Fibonacci) showing how activation records are pushed and popped during execution.

4. Write and explain an algorithm to reverse a string using a stack. Provide a dry run of the algorithm for the string "DATASTRUCTURES", showing intermediate stack states.

5. Explain how queues are used in CPU scheduling, particularly in the Round Robin algorithm.

Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2017). *Data structures and algorithms in Java* (6th ed.). Wiley.

3. Weiss, M. A. (2018). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.

4. Sahni, S. (2004). *Data structures, algorithms, and applications in C++* (2nd ed.). University Press.

5. Lafore, R. (2017). *Data structures and algorithms in Java* (2nd ed.). Pearson Education.

Suggested Reading

1. GeeksforGeeks — Data Structures: Stack and Queue https://www.geeksforgeeks.org/

2. TutorialsPoint — Stack and Queue Data Structures https://www.tutorialspoint.com/data_structures_algorithms/

3. Programiz — Stack and Queue Data Structures https://www.programiz.com/dsa/

4. StudyTonight — Stack and Queue Concepts and Applications https://www.studytonight.com/data-structures/

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

Searching and Sorting Algorithms

Learning Outcomes

After completing this unit, students will be able to:

- explain the step-by-step procedure of each searching and sorting
- use the appropriate sorting or searching technique based on the nature of data
- implement searching algorithms in a programming language
- analyze the time complexity of each algorithm in best, average, and worst-case scenarios

Background

Before studying searching and sorting algorithms, learners are expected to have a solid understanding of basic programming concepts. This includes familiarity with variables, data types, operators, and control structures like loops and conditional statements, which are essential for implementing algorithmic logic. Students should also be comfortable working with one-dimensional arrays, particularly in terms of indexing and traversing them. Knowledge of functions, including how to define and call them with parameters and return values, is important for writing modular and reusable code. A foundation in logical reasoning and mathematical operations, such as using comparison operators and performing arithmetic calculations (especially for determining midpoints in Binary Search), is also necessary. Additionally, prior experience in problem-solving and algorithmic thinking enables learners to break down problems into smaller steps and understand how different algorithms work. Familiarity with flowcharts or pseudocode can further enhance their ability to visualize the steps of an algorithm before coding.

Keywords

Sequential Search, Algorithm,Time Complexity, Divide-and-Conquer, In -Place Sorting

Discussion

2.4.1 Searching

Searching refers to the process of locating a specific item within a group of elements. It is a fundamental concept in computer science and also applies to many everyday scenarios. Whether it is finding a book arranged by title on a shelf, spotting a word in an alphabetically ordered dictionary, or choosing an outfit from a wardrobe, each situation involves different search methods. A typical example is typing a query into an online store's search bar to locate a product among thousands. The technique used in each case depends on how the data is arranged and how quickly the result needs to be obtained. Efficient searching minimizes time and effort, improving access to information. A search is considered successful when the desired item is found; otherwise, it is considered unsuccessful. In computing, common search algorithms include linear search and binary search.

2.4.2 Linear Search

Linear search, also referred to as Sequential search, is the most basic searching technique where each item in a list is examined in order until the target item is located or the end of the list is reached as in Fig 4.1.1. The process begins at the first element and compares each entry with the value being searched for. If a match is found, the index (position) of that element is returned. If the entire list is checked without finding the target, the algorithm returns -1 or a message indicating the item is not found.

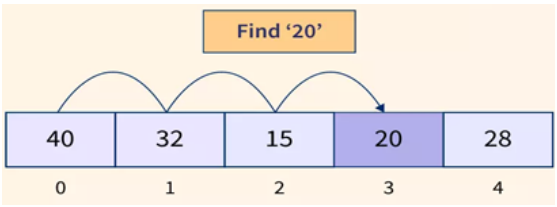


Fig 4.1.1 Linear Search

2.4.2.1 Algorithm

The process begins by examining the first index of the array.

Step 1 – Begin at index 0 and compare the target key with the element at that index.

Step 2 – If the element matches the key, return the index where it was found.

Step 3 – If there is no match, move to the next element in the array and compare again.

Step 4 – Continue this comparison until a match is found, then return the corresponding index.

Step 5 – If the end of the array is reached without finding the key, display a message indicating the element is not in the array and terminate the program.

Example

Step-by-Step Linear Search for Key Element 47

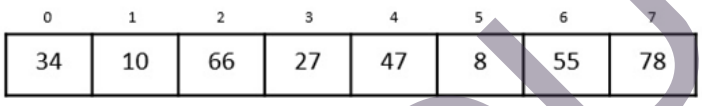


Fig. 4.1.2 (a) Linear Search for key element 47

Step 1: Linear search begins at index 0. Compare the target element (key) with the value stored at index 0, which is 34.

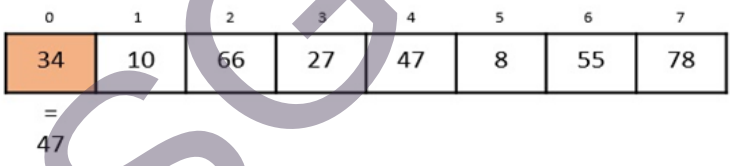


Fig. 4.1.2 (b) Linear Search for key element 47

However, 47 is not equal to 34, so the search proceeds to the next element.

Step 2: The key is now compared with the value at index 1 of the array.

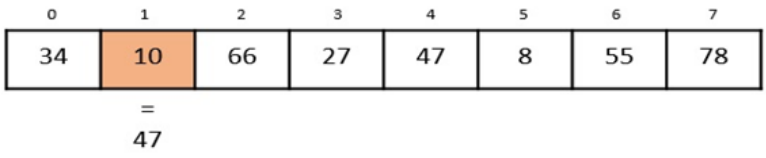


Fig. 4.1.2 (c) Linear Search for key element 47

47 still does not match 10, so the algorithm continues to the next iteration.

Step 3: The following element, 66, is compared with 47. Since they do not match, the search proceeds to check the remaining elements.

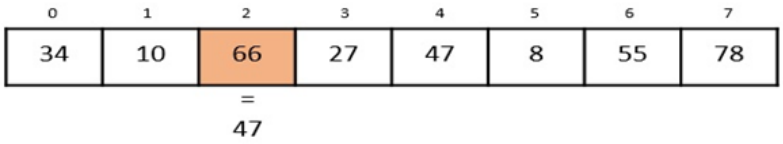


Fig. 4.1.2 (d) Linear Search for key element 47

Step 4: The element at index 3, which is 27, is now compared with the key value, 47. Since they are not equal, the algorithm continues to the next element in the array.

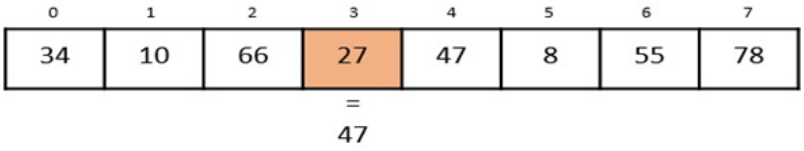


Fig. 4.1.2 (e) Linear Search for key element 47

Step 5: The element at index 4, which is 47, is compared with the key value 47. Since they match, the search is successful, and the index where the key is found,index 4, is returned.

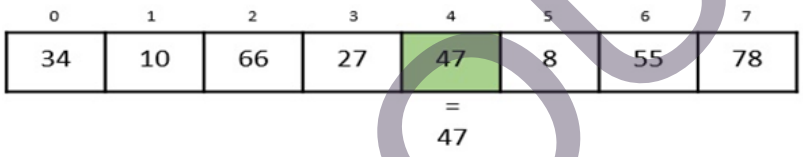


Fig. 4.1.2 (f) Linear Search for key element 47

The result obtained is: Element located at index 4.

2.4.2.2 Implementation

```
#include <stdio.h>
void linear_search(int a[], int n, int key)
{
int i, count = 0;
for(i = 0; i < n; i++)
{
if(a[i] == key)     // compares each element of the array
{
printf("The element is found at %d position\n", i+1);
count = count + 1;
                }
        }
if(count == 0)      // for unsuccessful search
printf("The element is not present in the array\n");
}
```

```
int main()
{
int i, n, key;
n = 6;
int a[10] = {12, 44, 32, 18, 4, 10};
key = 18;
linear_search(a, n, key);
key = 23;
linear_search(a, n, key);
return 0;
}
```

Output

The element is found at 4 position
The element is not present in the array

2.4.2.3 Time Complexity

In a linear search, each element is checked one after another until the target is found or the list ends. The time complexity of this method varies based on the position of the item being searched for. It can be analyzed under the following three scenarios:

a. Best Case – O(1)

If the target element is located at the very beginning of the array, only one comparison is needed to find it.

Example: Searching for 15 in the array {15, 20, 32, 45, 23, 50, 40}
Time complexity: O(1)

b. Worst Case – O(n)

If the desired element is either at the very end of the array or not present at all, every element must be checked, requiring n comparisons.

Example: Searching for 40 or a non-existent value like 60 in the same array.
Time complexity: O(n)

c. Average Case – O(n)

In the average scenario, we assume the target element has an equal chance of being at any position in the array.

- 1st position → 1 comparison
- 2nd position → 2 comparisons

...

- nth position → n comparisons

The total number of comparisons becomes:
$1 + 2 + 3 + ... + n = n(n + 1)/2$

Average comparisons = $(n + 1)/2$

Therefore, the average-case time complexity is:
$T(n) = O((n + 1)/2) \approx O(n)$

2.4.3 Binary Search

Have you ever looked up a word in a traditional dictionary? Think about how you usually do it. You do not start from the first page and check every word one by one. Instead, you open the dictionary near the middle and examine the words on that page. If the word you need comes before those, you flip back; if it comes after, you flip forward. You continue this process, gradually narrowing the range until you find the exact word.

This approach mirrors the concept behind Binary Search, one of the most efficient search techniques in computer science. Binary search searches for a specific key by comparing it with the middle element of the array as in Fig 4.1.2. If the middle element matches the key, its index is returned. If the key is smaller than the middle element, the search continues in the left half of the array. If the key is larger, the right half is searched instead. This process is repeated recursively until either the element is found or the search range becomes empty.

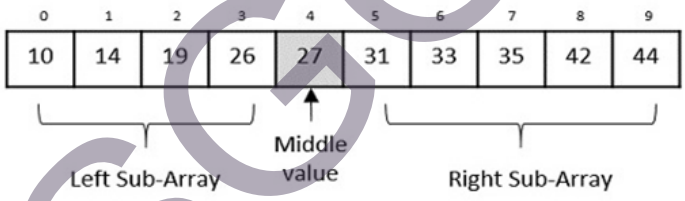


Fig. 4.1.3 Binary Search

2.4.3.1 Algorithm

The Binary Search algorithm is a method of searching that works by dividing the data into intervals. It only functions correctly on sorted arrays, as it relies on comparing the key value with the middle element to decide which half of the array to continue searching.

Step 1 – Identify the middle element of the array and compare it with the target key. If they are equal, return the index of the middle element.

Step 2 – If there is no match, determine whether the key is greater or smaller than the middle element.

Step 3 – If the key is greater, continue the search in the right half of the array. If the key is smaller, shift the search to the left half.

Step 4 – Repeat the above steps until the sub-array is reduced to a single element.

Step 5 – If the key is not found by the time the sub-array size becomes zero, the algorithm concludes that the search has failed.

Example

Here is a sorted array, and we need to find the position of the value 31 using the binary search technique.

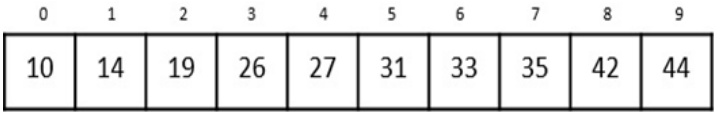


Fig. 4.1.3 (a) binary search for element 31

To begin, we calculate the middle index of the array using the formula:

mid = low + (high - low) / 2

In this case: mid = 0 + (9 - 0) / 2 = 4 (taking the integer part of 4.5)

So, the middle element is located at index 4.

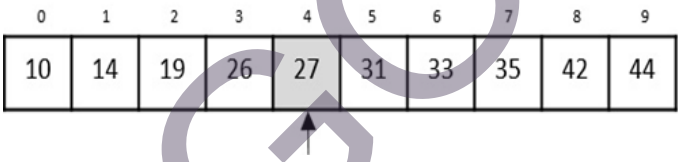


Fig. 4.1.3 (b) binary search for element 31

Next, we compare the element at index 4 with the target value, which is 31. The value at index 4 is 27, which is not a match. Since 31 is greater than 27 and the array is sorted, we can conclude that the target must lie in the right half of the array.

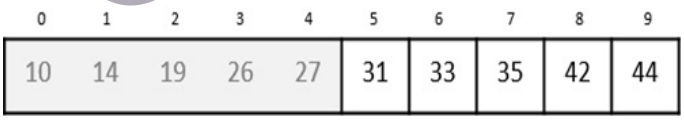


Fig. 4.1.3 (c) binary search for element 31

We update the low pointer to mid + 1 and recalculate the middle index:

low = mid + 1
mid = low + (high - low) / 2

This gives us a new middle index of 7. We then compare the value at index 7 with our target value, 31.

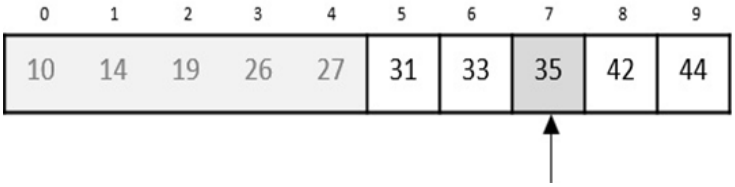


Fig. 4.1.3 (d) binary search for element 31

The value at index 7 does not match the target; in fact, it is less than the value we are searching for. Therefore, the target must lie in the lower section beyond this point (to the left of this index).

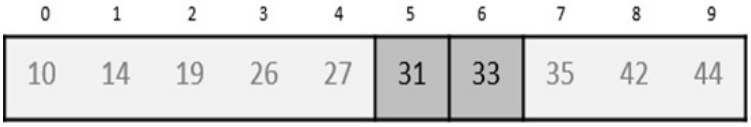


Fig. 4.1.3 (e) binary search for element 31

So, we recalculate the middle index once more, and this time the mid value comes out to be 5.

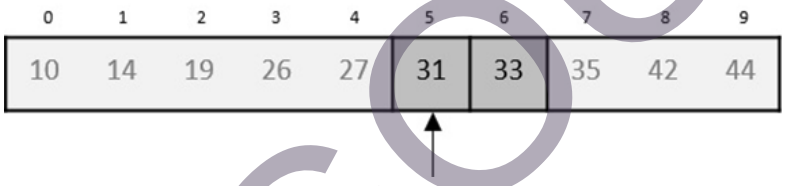


Fig. 4.1.3 (f) binary search for element 31

We then compare the element at index 5 with our target value and find that they are equal, a successful match.

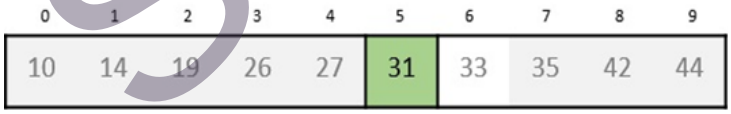


Fig. 4.1.3 (g) binary search for element 31

Therefore, we conclude that the target value 31 is located at index 5 in the array.

2.4.3.2 Implementation

```
#include<stdio.h>
void binary_search(int a[], int low, int high, int key)
{
int mid;
mid = (low + high) / 2;
if (low <= high)
        {
if (a[mid] == key)
printf("Element found at index: %d\n", mid);
else if(key < a[mid])
binary_search(a, low, mid-1, key);
else if (a[mid] < key)
binary_search(a, mid+1, high, key);
        }
        else if (low > high)
        printf("Unsuccessful Search\n");
}
int main()
{
int i, n, low, high, key;
n = 5;
low = 0;
high = n-1;
int a[10] = {12, 14, 18, 22, 39};
key = 22;
binary_search(a, low, high, key);
key = 23;
binary_search(a, low, high, key);
return 0;
}
```

Output

Element found at index: 3
Unsuccessful Search

2.4.3.3 Time Complexity

The time complexity of binary search is usually described through its best, average, and worst-case scenarios:

- Best Case: O(1)
  This happens when the target element is found on the very first

comparison, which occurs if the target is the middle element of the initial search range.

- Average Case: O(log n)
  On average, the number of comparisons needed grows logarithmically with the size of the array (n), since binary search continually divides the search space in half.

- Worst Case: O(log n)

  The worst case arises when the target is either not in the array or is found only after the search space has been narrowed down to a single element. In this scenario, the number of comparisons still scales logarithmically with the input size.

2.4.4 Sorting

Take a look at the image given in Fig 4.1.3. The boys are organising books on a shelf, which is a real-life example of sorting. The way they arrange the books can differ.They might sort them alphabetically by the author's name, by category like Science, Arts, or History, by size, color, or even by the date of publication. Regardless of the approach, the primary purpose is to arrange the books in an orderly and logical manner to make finding and using them more convenient.In a similar way, sorting plays an important role in computers and digital systems.

Fig. 4.1.4 Organising Books

Sorting involves arranging data in a specific order or sequence. A sorting algorithm determines how this arrangement is achieved. The most commonly used sorting orders are numerical (based on numbers) and lexicographical (similar to alphabetical order, as in a dictionary). Sorting plays a key role in improving the efficiency of data handling, making tasks such as searching, displaying, and analyzing information much easier and more effective.

Some sorting algorithms require a small amount of extra memory to perform comparisons and temporarily hold data elements. If the sorting is done directly within the original data structure, usually an array it is referred to as In-Place Sorting. An example of this

is bubble sort, which does not need significant additional space.

Certain sorting algorithms need extra memory that is equal to or greater than the size of the data set being sorted. These are known as Not-in-Place Sorting algorithms. Merge sort is a typical example, as it utilizes additional space while sorting.

2.4.5 Bubble Sort

Imagine a line of children, and the teacher wants to arrange them from shortest to tallest. The teacher starts by comparing the first two children in the line. If the child on the left is taller than the one on the right, they switch places. The teacher then moves to the next pair and continues this comparison and swapping process along the line. After one complete pass, the tallest child ends up at the far end of the line, in their correct position. The teacher then repeats the same process, this time excluding the last child, since they are already correctly placed. This continues until all the children are in proper height order.

This scenario mirrors how the Bubble Sort algorithm functions. In Bubble Sort, adjacent elements in a list are compared and swapped if they are out of order. This process is repeated multiple times until the entire list is sorted. It is called "Bubble Sort" because, with each pass, the largest (or smallest, depending on sorting order) element moves or "bubbles" to its correct position at the end of the list. Bubble Sort is a basic, comparison-based algorithm known for its simplicity, where sorting is done through repeated comparisons and swaps until no further changes are needed.

2.4.5.1 Algorithm

Assume we have an array called list containing n elements, and a swap function is used to exchange the values of two elements in the array.

Step 1 – Start by comparing the first element with the one that follows it.

Step 2 – If the first element is larger than the next, swap their positions; if not, move to the next pair of elements.

Step 3 – Continue this process until you reach the end of the array.

Step 4 – After one full pass, check whether the array is fully sorted. If it is not, repeat the comparison and swapping process again—this time moving from the end of the array back toward the beginning.

Step 5 – Once no more swaps are needed and the elements are in order, the array is considered sorted

Example

Consider an unsorted array

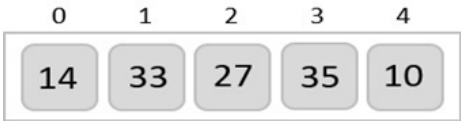


Fig. 4.1.5 (a) Bubble Sort

Bubble sort begins by taking the first two elements and comparing them to see which one is larger.

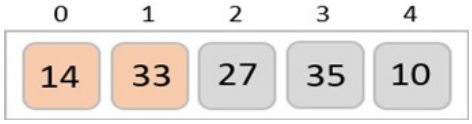


Fig. 4.1.5 (b) Bubble Sort

Here, 33 is greater than 14, so they are already in the correct order. We then move on to compare 33 with 27.

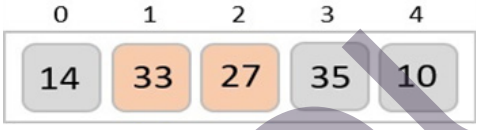


Fig. 4.1.5 (c) Bubble Sort

Since 27 is less than 33, the two elements are out of order and need to be swapped.

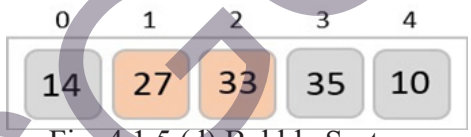


Fig. 4.1.5 (d) Bubble Sort

Next, we compare 33 with 35 and observe that they are already in the correct order.

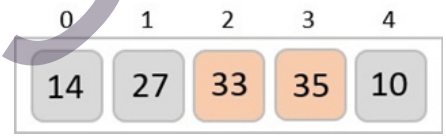


Fig. 4.1.5 (e) Bubble Sort

After that, we compare the next pair of elements: 35 and 10.

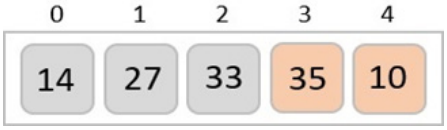


Fig. 4.1.5. (f) Bubble Sort

Since 10 is less than 35, they are not in the correct order and need to be swapped. At this point, we have reached the end of the array. After completing one full pass, the array will appear as follows:

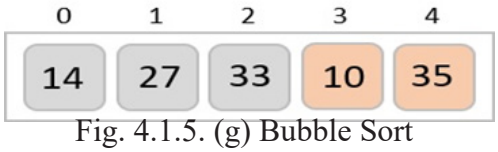


Fig. 4.1.5. (g) Bubble Sort

To be specific, we are now demonstrating how the array appears after each pass. Following the second iteration, the array should look like this:

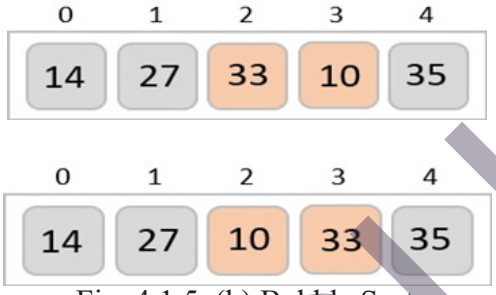


Fig. 4.1.5. (h) Bubble Sort

Observe that with each pass, at least one element settles into its correct position at the end of the array.

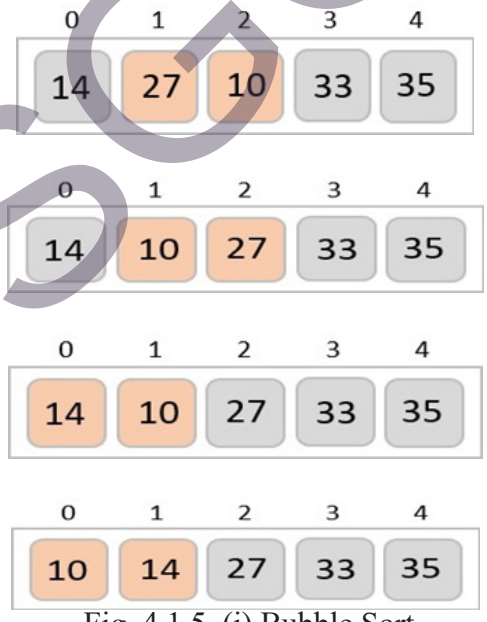


Fig. 4.1.5. (i) Bubble Sort

When no swaps are needed during a pass, bubble sort recognizes that the array is fully sorted.

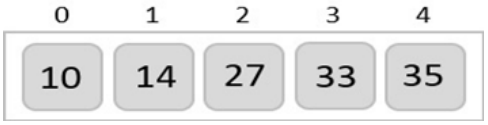


Fig. 4.1.5. (j) Bubble Sort

2.4.5.2 Implementation

```
#include <stdio.h>
void bubbleSort(int array[], int size)
{
      for(int i = 0; i<size; i++)
     {
int swaps = 0;                    //flag to detect any swap is there or not
for(int j = 0; j<size-i-1; j++)
 {
                if(array[j] > array[j+1])    //when the current item is bigger than
next
                {
int temp;
temp = array[j];
array[j] = array[j+1];
array[j+1] = temp;
swaps = 1; //set swap flag
              }
        }
       if(!swaps)
        break;                  // No swap in this pass, so array is sorted
     }
}
int main()
{
int n;
n = 5;
int arr[5] = {67, 44, 82, 17, 20};          //initialize an array
printf("Array before Sorting: ");
for(int i = 0; i<n; i++)
printf("%d ",arr[i]);
printf("\n");
bubbleSort(arr, n);
printf("Array after Sorting: ");
for(int i = 0; i<n; i++)
```

```
printf("%d ", arr[i]);
printf("\n");
}
```

Output

Array before Sorting: 67 44 82 17 20
Array after Sorting: 17 20 44 67 82

2.4.5.3 Time Complexity

- Best Case (O(n)): When the list is already sorted, the algorithm only needs to verify this with one comparison per element.
- Average Case ($O(n^2)$): When the elements are in random order, the algorithm typically performs many comparisons and shifts.
- Worst Case ($O(n^2)$): When the list is sorted in reverse order, every element must be compared and moved, resulting in the maximum number of operations.

2.4.6 Insertion Sort

Imagine you are sorting playing cards in your hand. You start by treating the first card as already sorted. Then, you pick up the next card and place it in its correct position relative to the cards already in your hand, either to the left or right depending on its value. You repeat this process with each remaining card until all the cards are arranged in order.This everyday scenario closely demonstrates how the Insertion Sort algorithm works.

Insertion Sort is a comparison-based, in-place sorting algorithm. It works by maintaining a sublist that is always sorted, usually beginning with the lower part of the array. As each new element is considered, the algorithm searches for its correct position within the sorted sublist and inserts it accordingly. This is done by scanning the array sequentially, shifting unsorted elements as needed, and placing the new item in its appropriate spot within the same array. While simple and efficient for small datasets, Insertion Sort is not well-suited for handling large amounts of data.

2.4.6.1 Algorithm

Step 1 – If the element is the first in the array, it is already considered sorted. Return 1.

Step 2 – Select the next element in the array.

Step 3 – Compare this element with all the elements in the sorted portion of the list.

Step 4 – Shift every element in the sorted sub-list that is larger than the current value one position to the right.

Step 5 – Insert the current value into its correct position.

Step 6 – Repeat the process until the entire array is sorted.

Example

Consider the following unsorted array

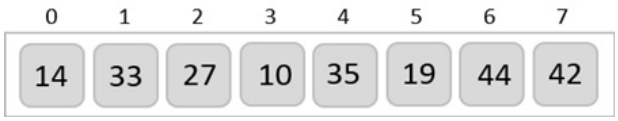


Fig. 4.1.6 (a) Inertion sort

Insertion sort begins by comparing the first two elements of the array.

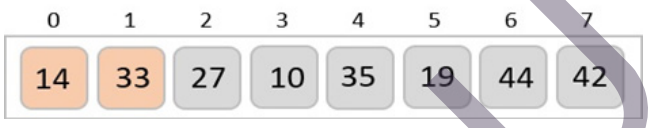


Fig. 4.1.6 (b) Inertion sort

It determines that 14 and 33 are already in the correct ascending order. At this point, 14 is considered part of the sorted sub-list.

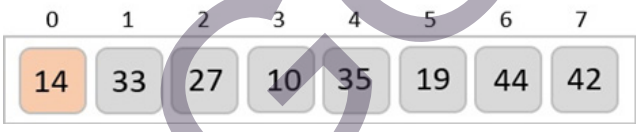


Fig. 4.1.6 (c) Inertion sort

Insertion sort proceeds to the next step and compares 27 with 33.

Fig. 4.1.6 (d) Inertion sort

It finds that 33 is out of place and swaps it with 27. The algorithm then checks 27 against the elements in the sorted sub-list. Since the sub-list currently contains only one element, 14, and 27 is greater than 14, the sorted order is maintained after the swap.

Fig. 4.1.6 (e) Inertion sort

At this point, 14 and 27 are part of the sorted sub-list. The algorithm then compares 10 with 33 and finds that they are not in the correct order.

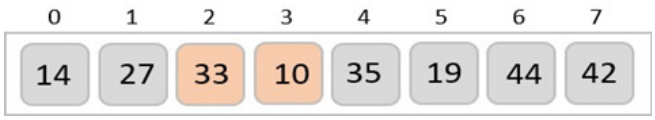


Fig. 4.1.6 (f) Inertion sort

So, the two values are swapped.

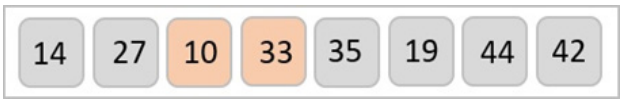


Fig. 4.1.6 (g) Inertion sort

However, this swap causes 27 and 10 to become unsorted.

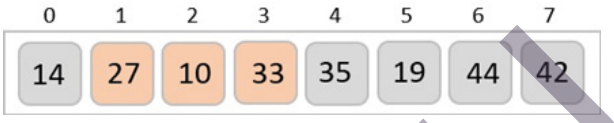


Fig. 4.1.6 (h) Inertion sort

Hence, we swap them too.

Fig. 4.1.6 (i) Inertion sort

We then notice that 14 and 10 are also not in the correct order.

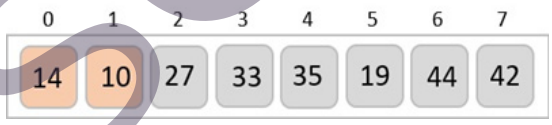


Fig. 4.1.6 (j) Inertion sort

Swap them again

Fig. 4.1.6 (k) Inertion sort

After the third iteration, the sorted sub-list contains four elements.

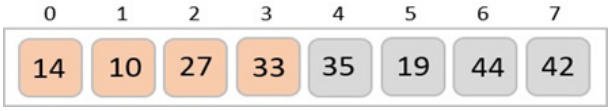


Fig. 4.1.6 (l) Inertion sort

This process continues until every element from the unsorted portion has been included in the sorted sub-list.

2.4.6.2 Implementation

```
#include <stdio.h>
void insertionSort(int array[], int size)
{
        int key, j;
      for(int i = 1; i<size; i++)
      {
          key = array[i];            //take value
          j = i;
         while(j > 0 && array[j-1]>key)
          {
                  array[j] = array[j-1];
                  j--;
         }
       array[j] = key;       //insert in right place
   }
}
int main()
{
 int n;

 n = 5;
int arr[5] = {67, 44, 82, 17, 20};   // initialize the array
printf("Array before Sorting: ");
for(int i = 0; i<n; i++)
printf("%d ",arr[i]);
printf("\n");
insertionSort(arr, n);
printf("Array after Sorting: ");
for(int i = 0; i<n; i++)
printf("%d ", arr[i]);
printf("\n");
}
```

Output

Array before Sorting: 67 44 82 17 20
Array after Sorting: 17 20 44 67 82

2.4.6.3 Time Complexity

- Best Case – O(n)
  This occurs when the array is already sorted. Each element is compared once with the previous one, and no shifting is required, only n−1 comparisons in total.

- Average Case – $O(n^2)$
  In this scenario, the elements are in random order. On average, each new element may need to be compared with about half of the already sorted elements and shifted accordingly, resulting in quadratic time.

- Worst Case – $O(n^2)$
  Happens when the array is sorted in reverse order. Each element has to be compared and shifted past all previously sorted elements, leading to the maximum number of comparisons and shifts

2.4.7 Merge Sort

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

Divide: The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

Conquer: The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

The breaking down and building up of the array to sort the array is done recursively.

2.4.7.1 Algorithm

Step 1: If the list contains only one element, it is already sorted, no further action is needed, so return it as is.

Step 2: Recursively split the list into two halves until each sub-list has only one element.

Step 3: Combine the smaller sub-lists back together in sorted order to form larger sorted lists.

Example

Consider the following unsorted array

Fig.4.1.7 (a)

Merge sort begins by repeatedly dividing the entire array into equal halves until individual elements (atomic values) are reached. In this example, an array containing 8 elements is initially split into two sub-arrays, each containing 4 elements.

Fig.4.1.7 (b)

This process maintains the original order of elements. Next, each of the two sub-arrays is further divided into halves.

Fig.4.1.7 (c)

We continue splitting these sub-arrays until we reach individual elements,atomic values,that can no longer be divided.

Fig.4.1.7 (d)

Now, we begin merging the elements in the same order in which they were originally divided. Take note of the color codes assigned to each list for clarity.

We start by comparing the elements in each pair of sub-lists and merging them in sorted order into new lists. For instance, 14 and 33 are already in the correct order, so they are combined as is. When comparing 27 and 10, we place 10 first, followed by 27 in the new list. Similarly, 19 and 35 are reordered, while 42 and 44 remain in their original sequence.

Fig.4.1.7 (e)

In the following step of the merging phase, we compare sub-lists containing two elements each and merge them into new lists of four elements, ensuring all values are arranged in sorted order.

Fig.4.1.7 (f)

Once all the merging steps are complete, the list is fully sorted and represents the final result.

2.4.7.2 Implementation

```
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high)
{
int l1, l2, i;
for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++)
        {
              if(a[l1] <= a[l2])
              b[i] = a[l1++];
         else
             b[i] = a[l2++];
        }
while(l1 <= mid)
b[i++] = a[l1++];
while(l2 <= high)
b[i++] = a[l2++];
for(i = low; i <= high; i++)
a[i] = b[i];
}
void sort(int low, int high)
{
     int mid;
     if(low < high)
     {
           mid = (low + high) / 2;
           sort(low, mid);
           sort(mid+1, high);
           merging(low, mid, high);
      }
     else
     {

return;
     }
}
int main()
{
int i;
printf("Array before sorting\n");
for(i = 0; i <= max; i++)
printf("%d ", a[i]);
```

```
sort(0, max);
printf("\nArray after sorting\n");
for(i = 0; i <= max; i++)
printf("%d ", a[i]);
}
```

Output

```
Array before sorting
10 14 19 26 27 31 33 35 42 44 0
Array after sorting
0 10 14 19 26 27 31 33 35 42 44
```

2.4.7.3 Time Complexity

- Best Case: O(n log n) — The array is divided and merged efficiently regardless of initial order.
- Average Case: O(n log n) — Typical scenario where elements are in random order.
- Worst Case: O(n log n) — Even in the least favorable arrangement, the algorithm maintains the same time complexity.

Summarised Overview

The Searching and Sorting Algorithms covers fundamental techniques used to organize and retrieve data efficiently. Linear Search is a simple method that checks each element one by one until the target is found or the list ends, making it suitable for unsorted data but less efficient for large datasets. Binary Search, on the other hand, is much faster but requires the array to be sorted; it repeatedly divides the search range in half to locate the target. In terms of sorting, Bubble Sort compares and swaps adjacent elements through multiple passes to gradually "bubble" the largest values to the end. Insertion Sort builds a sorted list by inserting each new element into its correct position, mimicking the way we sort playing cards. Merge Sort is a more advanced, divide-and-conquer algorithm that splits the array into smaller parts, sorts them, and then merges them back together in order. While simpler algorithms like Bubble and Insertion Sort

are easy to implement, they are inefficient for large datasets, unlike Merge Sort, which offers consistent performance with a time complexity of O(n log n).

Assignments

1. Explain the working principle of Linear Search with a suitable example. What is its time complexity in best, average, and worst-case scenarios?

2. Write a C program to implement Binary Search on a sorted array. Include user input for array elements and the key to be searched.

3. Compare Bubble Sort, Insertion Sort, and Merge Sort in terms of:
   - Algorithm type (comparison/in-place)
   - Time complexity (best, worst, average case)
   - Space complexity

4. Write a function in C to perform Insertion Sort, and test it with an array input from the user. Display the sorted result.

5. What is the role of the Divide and Conquer strategy in Merge Sort? Illustrate the process with a diagram or example.

Reference

1. Lafore, R. (2002). *Data structures and algorithms in Java* (2nd ed.). Sams Publishing.

2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.

3. Sinha, P. (2017). *Data structures, algorithms, and applications in C++* (2nd ed.). Wiley.

4. Skiena, S. S. (2008). *The algorithm design manual* (2nd ed.). Springer

Suggested Reading

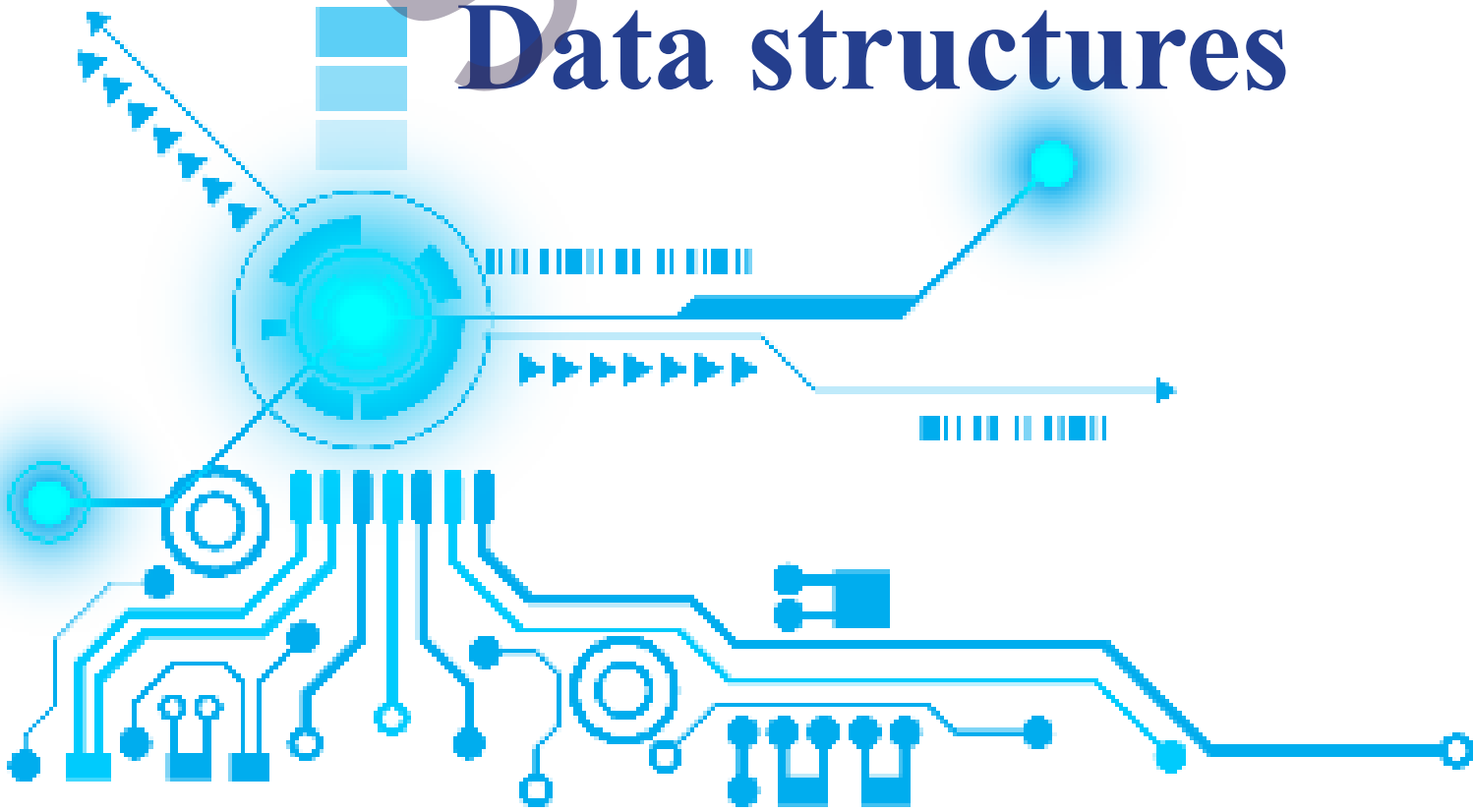
1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
2. Drozdek, A. (2012). *Data structures and algorithms in C* (2nd ed.). Cengage Learning.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
4. Knuth, D. E. (1998). *The art of computer programming, volume 3: Sorting and searching* (2nd ed.). Addison-Wesley
5. Weiss, M. A. (2014). *Data structures and algorithm analysis in C* (4th ed.). Pearson.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 3

Non-linear Data structures



1 UNIT

Trees

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the basic structure and terminologies of a tree
- differentiate between various types of binary trees
- understand and apply binary tree traversal techniques
- implement BFS and DFS traversals in binary trees
- identify real-world applications of tree data structures

Background

Trees are one of the most important non-linear data structures used in computer science to represent hierarchical relationships between elements. Unlike arrays or linked lists, which are linear, trees store data in a branching structure, making them ideal for representing data like organizational charts, file systems, expression parsing, and decision-making processes. Understanding tree terminologies, different types of binary trees, and traversal methods is essential for designing efficient algorithms for searching, sorting, and manipulating hierarchical data.

Keywords

Tree, Node, Root, Parent, Child, Leaf, Edge, Height, Depth, In-order Traversal, Pre-order Traversal, Post-order Traversal

3.1.1 Tree Terminologies

Trees are hierarchical data structures consisting of nodes connected by edges. Each tree has a root node, which serves as the starting point, and every other node is connected by edges forming a parent-child relationship. Key terminologies include root, leaf, sibling, and subtree, which help in understanding the structure and relationships within a tree. The following figure 1.3.2 shows a tree structure. Each line connecting two nodes denotes a relation, namely parent-child relation between two nodes.

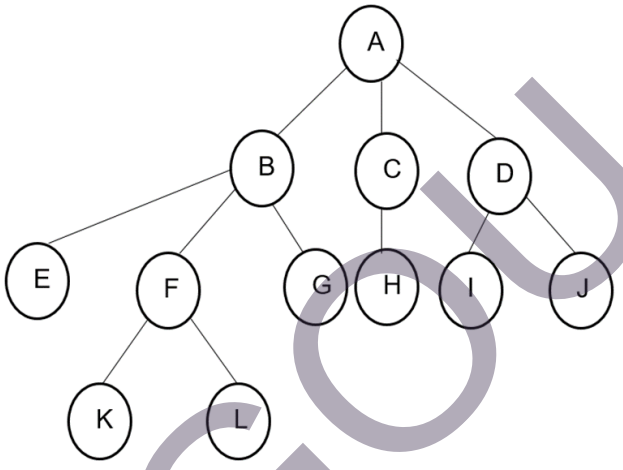


Fig. 3.1.1 Structure of Tree

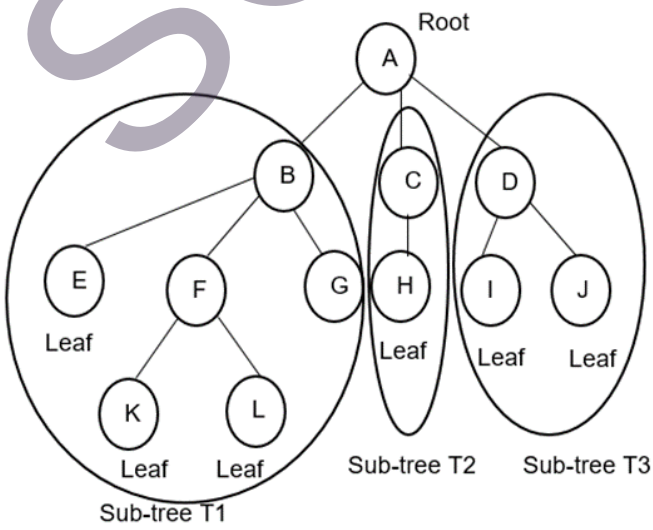


Fig. 3.1.2 The Tree and its related terms

From figure 3.1.2, the following components of a tree can be identified.

1. Node

A node is a fundamental unit in the tree that holds data. Each circle (labeled A to L) represents a node. Every node may act as a root, internal, or leaf node depending on its position in the tree. Example: Nodes A, B, C, D, ..., L are all nodes.

2. Root

The root is the topmost node in the tree. It has no parent and serves as the origin of all paths. Example: A is the root, since all other nodes are connected through it.

3. Parent

A parent node is a node that has one or more children. It is directly connected to lower-level nodes.

Examples :

- A is the parent of B, C, and D.
- B is the parent of E, F, and G.
- F is the parent of K and L.

4. Child

A child node is one that has a parent above it. It directly descends from another node.

Examples :

- B is a child of A
- E is a child of B
- L is a child of F

5. Siblings

Siblings are nodes that share the same parent.

Examples :

- B, C, D are siblings (all children of A)
- E, F, G are siblings (children of B)
- K and L are siblings (children of F)

6. Leaf Node (External Node)

A leaf node is a node with no children. It represents the end of a branch in the tree.

Leaf Nodes: E, G, H, I, J, K, L; These nodes have no further descendants.

7. Internal Node

An internal node has at least one child. It's between the root and the leaves.

Internal Nodes: A, B, C, D, F; They are not leaf nodes and are not the final destination in their path.

8. Edge

An edge is a connection between a parent and its child node. It represents the link or relationship.

Examples of edges:

A–B, A–C, B–E, F–K, F–L, etc.

There are n - 1 edges in a tree with n nodes.

The above tree has 12 nodes → 11 edges.

9. Path

A path is a sequence of nodes connected by edges, starting from one node and ending at another. Examples:

- Path from A to K: A → B → F → K
- Path from A to H: A → C → H

Path length = number of edges traversed.

10. Level

The level of a node refers to its distance from the root, starting at 0 for the root.

Level-wise breakdown:

- Level 0: A
- Level 1: B, C, D
- Level 2: E, F, G, H, I, J
- Level 3: K, L

11. Depth

Depth is the number of edges from the root to a given node. It is the same as the level of the node.

Examples :

- Depth of E = 2
- Depth of K = 3

12. Height of a Node

The height of a node is the number of edges on the longest path from that node down to a leaf.

Examples :

- Height of F = 1 (to K or L)
- Height of B = 2 (via F → K)
- Height of A = 3 (via B → F → K)

13. Height of the Tree

The height of a tree is the height of the root node. It is the longest path from root to any leaf.

Longest path : A → B → F → K = 3 edges

So, height = 3

14. Subtree

A subtree consists of a node and all its descendants.

- Subtree rooted at B includes: B, E, F, G, K, L
- Subtree rooted at D includes: D, I, J

Each child node forms its own subtree.

15. Degree of a Node

The degree of a node is the number of children it has.

Examples:

- Degree of A = 3 (B, C, D)
- Degree of B = 3 (E, F, G)
- Degree of F = 2 (K, L)
- Degree of E = 0 (leaf)

3.1.2 Binary Trees

In a normal tree, each node can have any number of children. However, in a binary tree, each node can have at most two children, or it can be an empty tree. A binary tree is a finite set of nodes that either has no nodes or consists of a root node and two separate binary trees known as the left subtree and the right subtree. Figure 3.1.3 illustrates the generic structure of a binary tree.

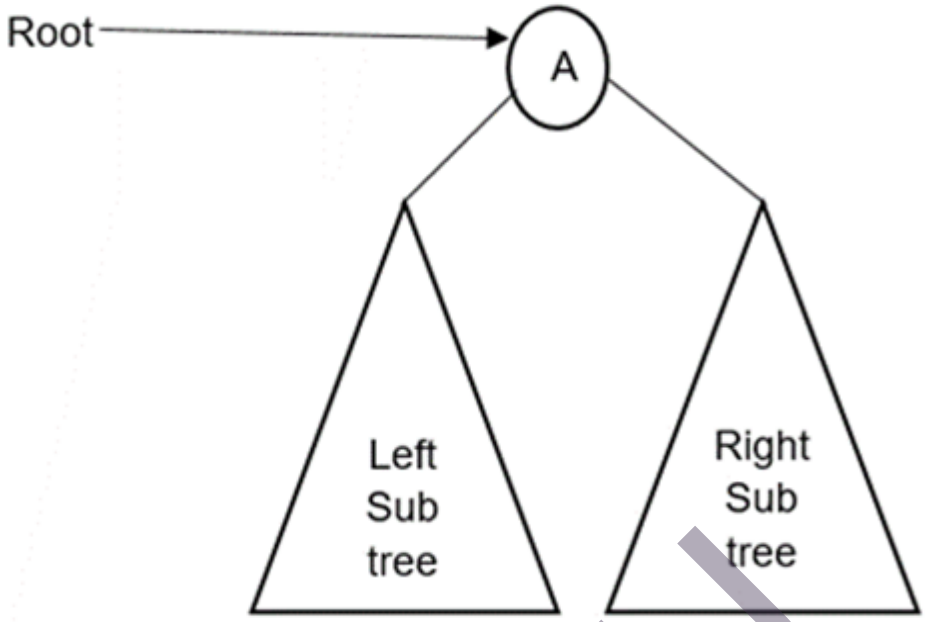


Fig. 3.1.3 Generic Binary Tree

A non-empty binary tree consists of the following:

- A node called the root node
- A left sub-tree
- A right subtree

Binary tree are classified into four type:

- Complete binary tree
- Full binary tree
- Degenerate binary tree
- Skewed binary tree

3.1.2.1 Complete Binary Trees

In the case of a complete binary tree, all the nodes are filled except the last one. The nodes must be as left as possible in the last level. The following figure 3.1.4 is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

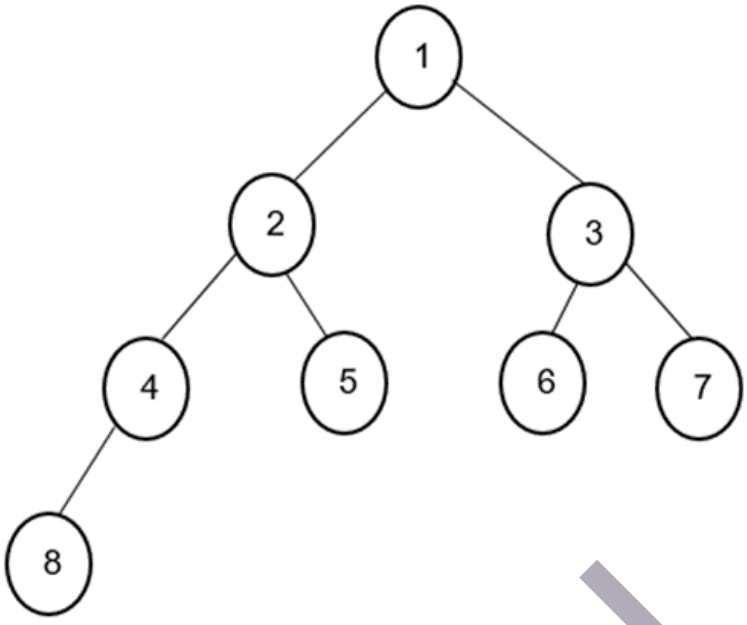
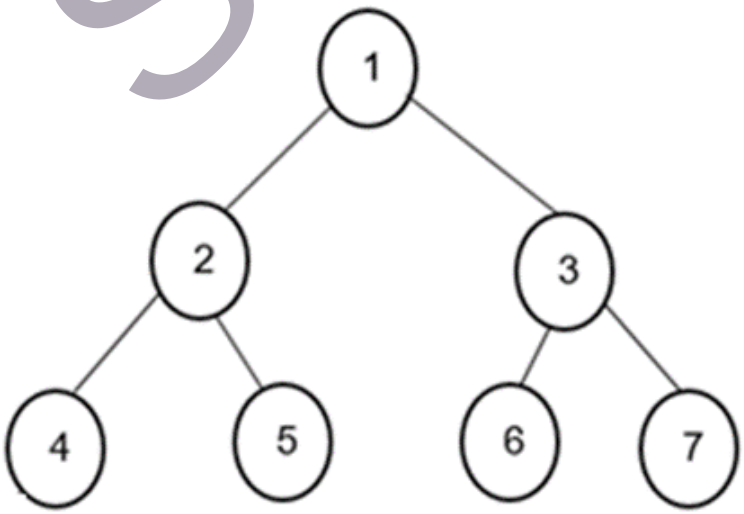


Fig. 3.1.4 Complete Binary Tree

3.1.2.2 Full Binary Trees

A full binary tree contains the maximum allowed number of nodes at all levels. This means that each node has exactly zero or two children. The leaf nodes at the last level will have zero children and the other non-leaf nodes will have exactly two children as shown in the following figure 3.1.5 a and b.

(a)



(b)

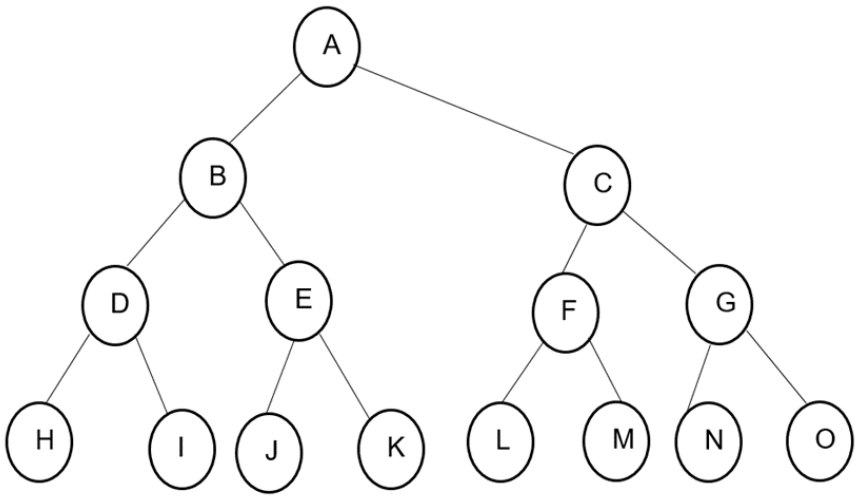


Fig. 3.1.5 Full Binary Trees

A complete binary tree is a full tree except at the last level where all nodes must appear as far left as possible.

3.1.2.3 Degenerate Binary Trees

A degenerate binary tree is a binary tree where each internal node has only one child. This results in a structure that resembles a linked list rather than a typical tree, with all nodes effectively forming a single path. The term "degenerate" implies that the tree structure has lost its typical branching characteristics and has become a linear structure.

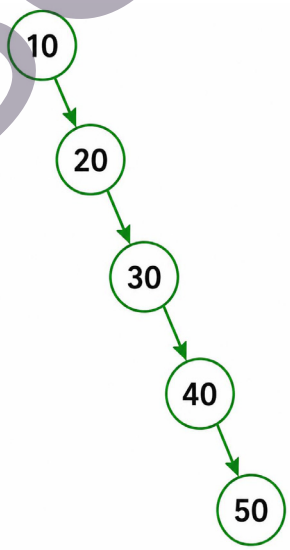


Fig. 3.1.6 Degenerate Binary Trees

3.1.2.4 Skewed Binary Tree

A skewed binary tree is a type of degenerate binary tree where each node has only one child, forming a linear structure. Depending on whether the child is to the left or right, it can be classified into two types:

- Left Skewed Binary Tree
- Right Skewed Binary Tree

These trees are inefficient for searching and inserting operations because they behave like a linked list rather than a balanced tree.

1. Left Skewed Binary Tree

A Left Skewed Tree is one in which every node has only a left child, and no right child.

Example :

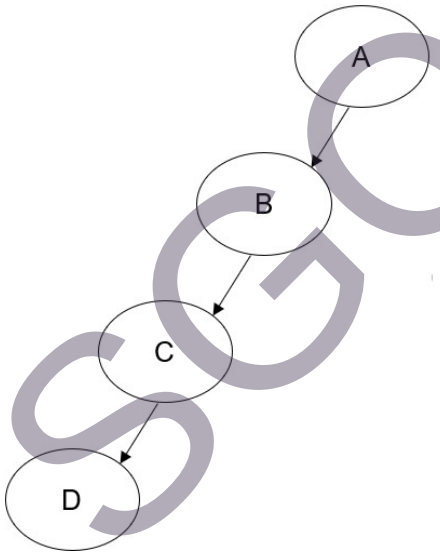


Fig. 3.1.7 Left Skewed Binary Tree

- Nodes A → B → C → D form a straight line leaning left.
- This happens if data is inserted in decreasing order in a BST

2. Right Skewed Binary Tree

A Right Skewed Tree is one in which every node has only a right child, and no left child.

Example:

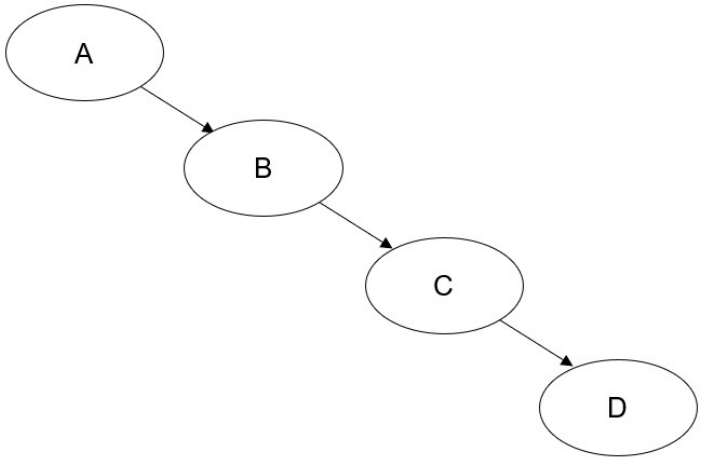


Fig. 3.1.8 Right Skewed Binary Tree

- Nodes A → B → C → D form a straight line leaning right.
- This happens if data is inserted in increasing order in a BST.

Structure of a Node

Each node comprises the following:

- It contains some information.
- It has an edge to a left child node.
- It has an edge to a right child node.

The above components of a node can be comfortably represented by a linked list as shown in Figure 3.1.10. Thus, a node consists of:

a. pointer that points towards the right node(Right Child Address)
b. pointer that points towards the left node(Left Child Address)
c. data element.

Left Child Address	Data	Right Child Address

Fig. 3.1.9 Node of a binary Tree

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
int data;
struct node *leftChild;
struct node *rightChild;
};
```

In a tree, all nodes share a common construct.

3.1.3 Tree Traversal

Tree traversal is the process of visiting every node in a tree data structure exactly once in a specific order. It is a fundamental concept used in many applications like searching, expression evaluation, and hierarchical data management. The two major categories of traversal are Breadth-First Search (BFS) and Depth-First Search (DFS). Each follows a unique strategy to explore the tree.

3.1.3.1 Breadth-First Search(BFS)

Breadth-First Search, also known as level-order traversal, explores the tree level by level. Starting from the root, BFS visits all nodes at the current depth before moving on to nodes at the next depth level. BFS is typically implemented using a queue data structure.

The steps for BFS traversal are as follows:

- Enqueue the root node.
- While the queue is not empty:
  - Dequeue a node from the front of the queue.
  - Visit the dequeued node.
  - Enqueue all the children (or left and right child in binary trees) of the visited node.

This process ensures that nodes are visited in order of their distance from the root, making BFS ideal for finding the shortest path in unweighted trees or graphs.

Example :

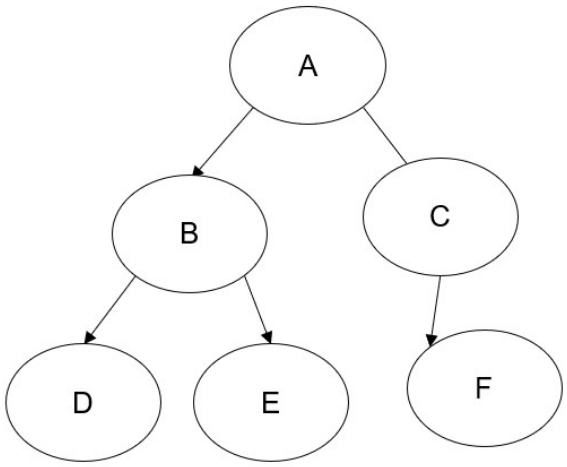


Fig 3.1.10 An unweighted tree

The BFS traversal for this tree would be:

A → B → C → D → E → F

BFS is especially useful in scenarios where we want to find the shortest path in an unweighted tree or graph because it explores all paths level-wise before going deeper.

3.1.3.2 Depth-First Search (DFS)

Imagine you are organizing a series of meetings with three colleagues: Alice, Bob, and Carol. Each meeting must be held in a specific order, but the sequence can vary depending on your priorities. Here are all the possible orders in which you could meet them:

Alice, Bob, Carol

Alice, Carol, Bob

Bob, Alice, Carol

Bob, Carol, Alice

Carol, Alice, Bob

Carol, Bob, Alice

Now, suppose you want to structure your meetings so that Bob is always met before Carol. This restriction narrows down your options to the following three sequences:

A. Alice, Bob, Carol

B. Bob, Alice, Carol

C. Bob, Carol, Alice

This scenario is analogous to traversing a binary tree, where each person represents a node and the order of meetings reflects the traversal path. In a binary tree, each node can have up to two children, often referred to as the left and right child. The way you visit the nodes, whether you prioritize the left child, the right child, or the parent node determines the type of traversal.

Let's map the meeting sequences to the three primary types of binary tree traversal:

- Sequence A (Alice, Bob, Carol) is similar to inorder traversal: you visit the left child (Alice), then the parent (Bob), and finally the right child (Carol).
- Sequence B (Bob, Alice, Carol) resembles preorder traversal: you visit the parent (Bob) first, then the left child (Alice), and finally the right child (Carol).
- Sequence C (Bob, Carol, Alice) is like postorder traversal: you visit the left child (Bob), then the right child (Carol), and finally the parent (Alice).

A binary tree is traversed for many purposes such as for searching a particular node, for processing some or all nodes of the tree and the like. (In the case of a binary tree, every node can have a maximum of 2 children.) There are three types of tree traversals. Sequence A in the above example is similar to inorder traversal where, left subtree traversal is followed by root followed by right subtree traversal. Sequence B in the above example is similar to a preorder traversal where, root followed by the left and right subtree is traversed. Sequence C in the above example is similar to postorder traversal where, left subtree traversal is followed by right subtree traversal that is followed by root visit.

During tree traversal, all the nodes of a tree are visited and their values may be printed. The reasons for binary tree traversal includes searching a particular node, for processing some or all nodes, etc. The following operations are possible on a node of a binary tree:

1. Process the visited node – V.
2. Visit its left child – L.
3. Visit its right child – R.

Any combinations of the above three operations can be done for tree traversal. The tree traversals have been named as preorder, inorder, and postorder according to the operation visit node (V).

1.Inorder Traversal

In the meeting analogy, this is like meeting Alice first (left), then Bob (parent), and finally Carol (right). Let us go into the details of inorder traversal.

L-V-R : Inorder traversal, that is, travel the left sub-tree (L), process the visited node

(V) and travel the right subtree (R).

Algorithm : An algorithm for inorder travel (L-V-R) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
inorderTravel(Tree){
   if (Tree == NULL) return
else
   {
   inorderTravel (leftChild (Tree));
   process DATA (Tree);
   inorderTravel (rightChild (Tree));
   }

   }
```

Explanation : In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree i.e, call inorderTravel (leftChild (Tree));
2. Visit the root i.e; process DATA (Tree);
3. Traverse the right subtree, i.e., call inorderTravel (rightChild (Tree));

Illustration: Figure 3.1.10 shows the illustration of the Inorder traversal of the binary tree in figure 3.1.11

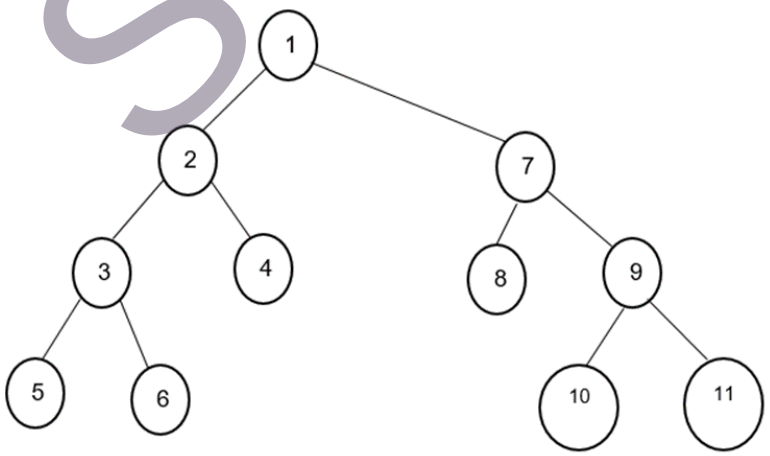


Fig. 3.1.11 Binary Tree

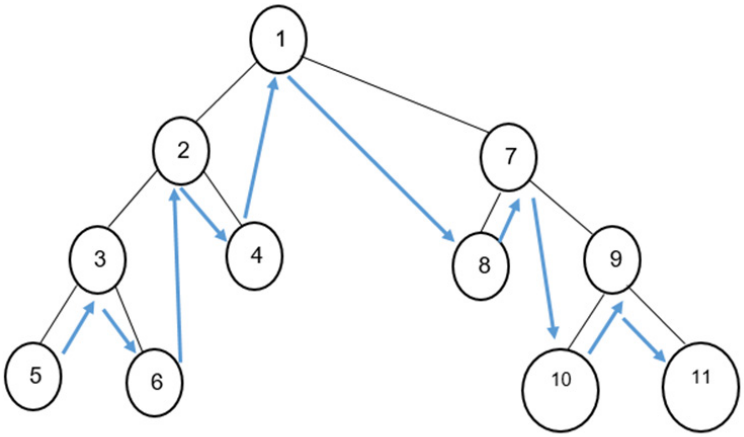


Fig. 3.1.12 Inorder traversal

In case of figure 3.1.11, the Inorder traversal will give the following result.

5 3 6 2 4 1 8 7 10 9 11

Explanation : In the case of Inorder traversal, L is followed by V that is followed by R. So, in figure 3.1.13, first 5(L) is visited followed by 3(V) and then 6(R). This is followed by 2(V) which is followed by 4(R). Then 1(V) is visited, followed by 8(L) and then 7(V). This is followed by 10(L), 9(V) and 11(R).

2. Preorder Traversal

In the travel path example discussed above, the sequence B (Bob, Alice, Carol) is similar to a preorder traversal where the root followed by the left and right subtree are traversed. Let us now go into the details of preorder traversal.

V-L-R : Preorder traversal, that is, process the visited node (V), travel the left sub-tree (L) and travel the right subtree (R)

Algorithm: An algorithm for preorder travel (V-L-R) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

```
preorderTravel(Tree){
if (Tree == NULL) return
else
    {
      process DATA (Tree);
      preorderTravel (leftChild (Tree));
      preorderTravel (rightChild (Tree));
    }
}
```

Explanation: In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Visit the root i.e; process DATA (Tree);
2. Traverse the left subtree i.e, call preorderTravel (leftChild (Tree));
3. Traverse the right subtree, i.e., call preorderTravel (rightChild (Tree));

Illustration:

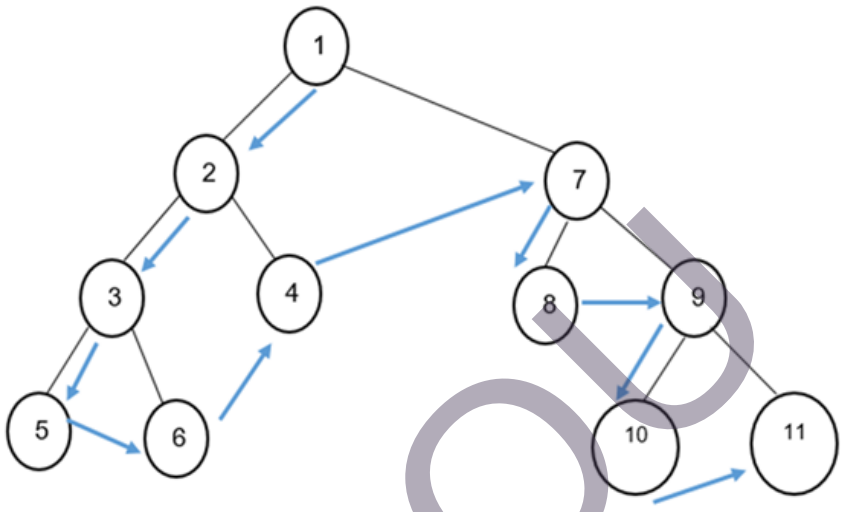


Fig. 3.1.13 Preorder traversal

In case of figure 3.1.12, the Preorder traversal will give the following result.

1 2 3 5 6 4 7 8 9 10 11

Explanation : In the case of preorder traversal, node(V) is visited followed by L and R. So, in figure 3.1.11, 1(V) is visited followed by 2(L), 3(L), 5(L). then this is followed by 6 (R), 4(R). This is followed by 7(V) that is followed by 8(L). This is followed by 9 (V), 10(L), and 11(R).

3. Postorder Traversal

In the travel path example the sequence C (Bob, Carol, Alice) is similar to postorder traversal where, left subtree traversal is followed by right subtree traversal that is followed by root visit.

Let us now go into the details of postorder traversal.

L-R-V : Postorder traversal, that is, travel the left sub-tree (L), travel the right sub-tree (R) and process the visited node (V)

Algorithm : An algorithm for postorder travel (L-R-V) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
PostOrderTravel(Tree){
if (Tree == NULL) return
else
   {
      postOrderTravel (leftChild (Tree));
      postOrderTravel (rightChild (Tree));
      process DATA (Tree);
   }
}
```

Explanation: In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree i.e, call postorderTravel (leftChild (Tree));
2. Traverse the right subtree, i.e., call postorderTravel (rightChild (Tree));
3. Visit the root i.e; process DATA (Tree);

Illustration :

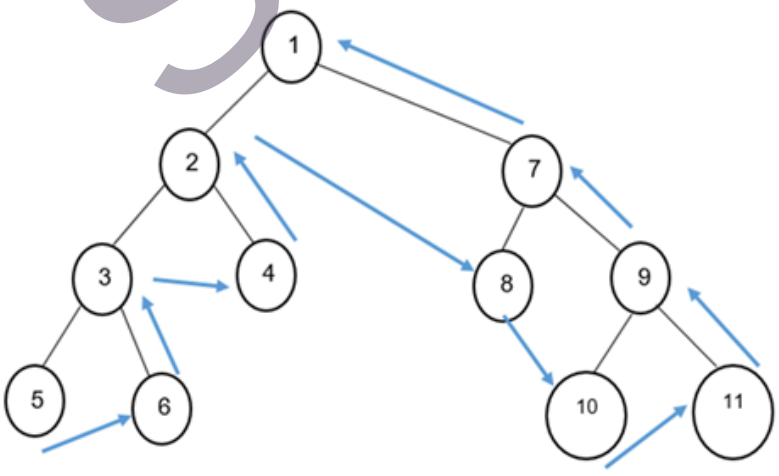


Fig. 3.1.14 Postorder traversal

In case of figure 3.1.13, the Postorder traversal will give the following result.

5 6 3 4 2 8 10 11 9 7 1

Explanation : In the case of Postorder traversal, 5(L) is followed by 6(R) which is followed by 3(V). This is followed by 4(R) which is followed by 2(V). This is followed by 8(L), 10(L) and 11(R). This is followed by 9(V), 7(V), and 1(V).

Complexity of Tree traversal

Tree traversal algorithms such as preorder, inorder, and postorder traversal are used to visit every node in a tree in a systematic order. During traversal, each node is processed exactly once. Therefore, if a tree contains n nodes, the traversal algorithm performs n operations, one for each node. Because the number of operations increases directly with the number of nodes, the time complexity of tree traversal is (O(n)). This means that regardless of whether the traversal is preorder, inorder, or postorder, the algorithm must visit every node in the tree to complete the traversal.

The space complexity of these traversals mainly depends on the height of the tree, because recursive traversal methods use the system call stack to keep track of function calls. If the height of the tree is represented by h, then the space complexity is (O(h)). In a balanced tree, the height is relatively small (approximately (log,n)), so the memory required is also small. However, in a skewed tree where nodes are arranged like a linked list, the height can become equal to the number of nodes, resulting in (O(n)) space usage. Thus, while the time complexity remains linear for all traversals, the memory usage depends on the structure and height of the tree.

Summarised Overview

The tree data structures, which are hierarchical, non-linear data structures composed of nodes and edges. It defines key terminologies such as root, parent, child, and leaf nodes, and explains how to determine a node's level, depth, and height. The text then shifts focus to binary trees, a specific type where each node has at most two children. It categorizes these into complete, full, and degenerate binary trees, illustrating each with examples. Finally, the document explains two primary methods for visiting every node in a tree, known as traversal: Breadth-First Search (BFS), which explores level by level, and Depth-First Search (DFS), which explores down a path before backtracking. Three types of DFS. in-order, pre-order, and post-order are described with their respective recursive algorithms and examples.

Assignments

1. Define the following tree terminologies with examples: root, leaf, internal node, height, depth, and degree.
2. Differentiate between a binary tree and a general tree.
3. Explain with diagrams the differences between full binary tree and complete binary tree.
4. What is a degenerate tree? Give a real-life analogy.
5. Define a skewed binary tree. How does it affect traversal time complexity?
6. Write and explain the algorithm for BFS traversal of a binary tree.
7. Write and explain the algorithm for DFS (in-order) traversal of a binary tree.
8. Write and explain the algorithm for DFS (pre-order) traversal of a binary tree.
9. Write and explain the algorithm for DFS (post-order) traversal of a binary tree.
10. Draw a binary tree of your own and perform all three DFS traversals on it, showing the output order for each.
11. Create a binary tree from the given node sequence and illustrate BFS traversal.

Reference

1. Kernighan, B. W., & Ritchie, D. M. (1988). The C programming language (2nd ed.). Prentice Hall.

2. Tanenbaum, A. S., Langsam, Y., & Augenstein, M. J. (2015). Data structures using C (2nd ed.). Pearson Education.

3. Weiss, M. A. (2006). Data structures and algorithm analysis in C (2nd ed.). Addison-Wesley.

4. Lipschutz, S. (2014). Data structures with C (Schaum's outlines series). McGraw-Hill Education.

5. Deshpande, A., & Kakde, O. G. (2008). C and data structures. Dreamtech Press.

Suggested Reading

1. Kruse, R. L., Tondo, C. L., & Leung, B. P. (1997). Data structures and program design in C (2nd ed.). Prentice Hall.

2. Kanetkar, Y. P. (2017). Data structures through C (2nd ed.). BPB Publications.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT Binary Search Trees

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the process of insertion and deletion in Binary Search Trees
- describe how AVL Trees maintain balance using rotations
- apply algorithms for insertion and deletion in BST, AVL Trees
- summarize the concept of m-way search tree
- discuss the operations of B- Trees, and B+ Trees

Background

Search trees are an essential data structure in computer science, used to store and organize data in a way that enables efficient searching, insertion, and deletion operations. Among them, the Binary Search Tree (BST) is one of the most fundamental types, where each node follows the property that the left child contains values less than the parent, and the right child contains values greater than the parent. This simple ordering principle allows for quick lookups, making BSTs ideal for implementing dictionaries, databases, and dynamic sets. However, without careful structuring, a BST can become unbalanced, leading to inefficient performance similar to a linked list.

To overcome the inefficiency caused by unbalanced BSTs, balanced search trees are introduced. These are specialized forms of search trees that maintain their height close to the minimum possible, ensuring that operations like search, insertion, and deletion consistently work in logarithmic time. The AVL Tree, one of the earliest self-balancing binary search trees, maintains a strict height balance by ensuring that the height difference between left and right subtrees of any node is at most

one. This balancing mechanism guarantees fast performance even after multiple dynamic updates, making AVL trees suitable for real-time applications where consistent speed is crucial.

While BSTs and AVL trees work well for in-memory data structures, large-scale storage systems such as databases often require structures optimized for disk access. This is where M-way search trees, including B-Trees and B+ Trees, play a key role. Unlike binary trees, these structures allow each node to have multiple children, significantly reducing the height of the tree and minimizing disk reads during operations. B-Trees are widely used in database indexing, while B+ Trees, a variation that stores all data in leaf nodes and maintains linked leaves for range queries, are particularly effective in file systems and database management systems. Together, these tree structures provide the foundation for efficient data organization in both memory-constrained and large-scale storage environments.

Keywords

Balanced Search Trees, AVL Tree, M-way Search Trees, B- Tree, B+ Tree

Discussion

Efficient storage and retrieval of data is a core requirement in computer science, and search trees are among the most widely used structures to achieve this goal. A Binary Search Tree (BST) organizes data so that searching, insertion, and deletion can be performed quickly by following a simple ordering rule. However, to maintain efficiency as data changes, trees must remain balanced, leading to the use of Balanced Search Trees like the AVL Tree, which automatically adjusts its structure. In large-scale systems such as databases and file systems, M-way search trees like B-Trees and B+ Trees are preferred because they minimize disk access and handle large volumes of data effectively. This unit explores the principles, operations, and practical applications of these tree structures, highlighting their importance in both memory-based and storage-based data management.

3.2.1 Overview of Binary Search Trees

A Binary Search Tree (BST) is an organized form of a binary tree that is widely used in computer science to store, manage, and retrieve data efficiently. It adheres to a specific ordering property: for any given node, all the values in its left subtree are less than the node's key, and all the values in its right subtree are greater than the node's key (fig 3.2.1). This rule is applied recursively to every node in the tree, ensuring that the entire

structure remains sorted. The hierarchical nature of a BST allows for quick searching, insertion, and deletion operations, typically achieving time complexities close to O(log n) in balanced cases. By maintaining this ordered structure, a BST not only supports fast access to individual elements but also facilitates operations like finding the minimum or maximum value, performing range queries, and traversing the elements in sorted order using inorder traversal. However, the efficiency of a BST can degrade to O(n) in the worst case if it becomes unbalanced, which is why balanced variants like AVL trees and Red-Black trees are often used in practice.

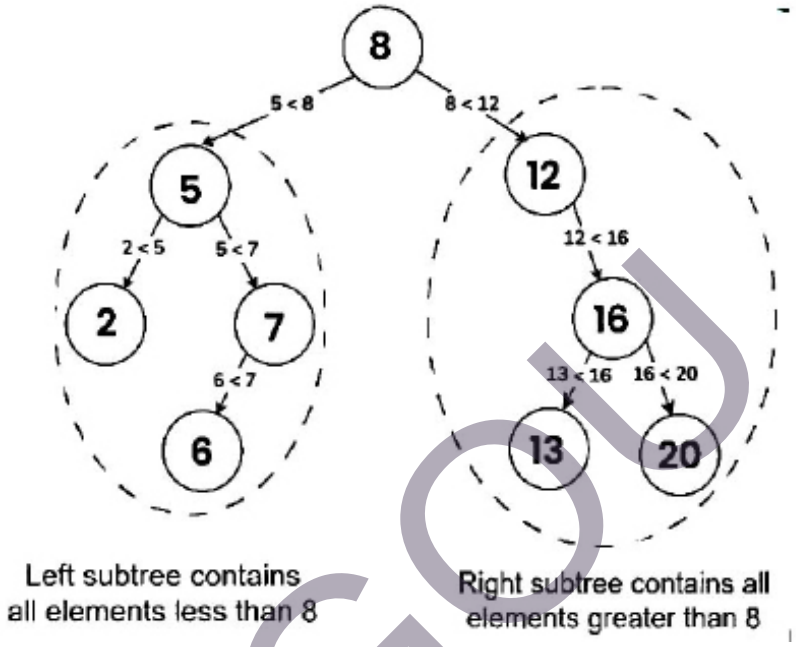


Fig. 3.2.1 Binary Search Tree

Some basic rules of a Binary Search Tree (BST) are:

- The left subtree of a node contains only keys smaller than the node's key.
- The right subtree of a node contains only keys larger than the node's key.
- Both the left and right subtrees must also be valid BSTs.
- Duplicate keys are generally not allowed, though some variations handle them differently.

3.2.1.1 Binary Search Tree Properties

A Binary Search Tree has distinct characteristics that enable fast search, insertion, and deletion operations.

1. **Node Components:** Each node in a BST consists of the following;
    - Data: The value stored in the node.
    - Left Child: A reference or pointer to the node's left child.
    - Right Child: A reference or pointer to the node's right child.

2. **BinaryStructureTree :** A BST is a specialized form of a binary tree, where each node can have up to two children: one on the left and one on the right.

3. **Node Ordering Property :** For any given node,

- All keys in the left subtree are less than the node's key.
- All keys in the right subtree are greater than the node's key.

Example 1 : Consider the fig 3.2.2 BST, for node 10; left subtree values {5, 3, 8} are less than 10, right subtree values {16, 15, 20} are greater than 10. This property holds true for every node.

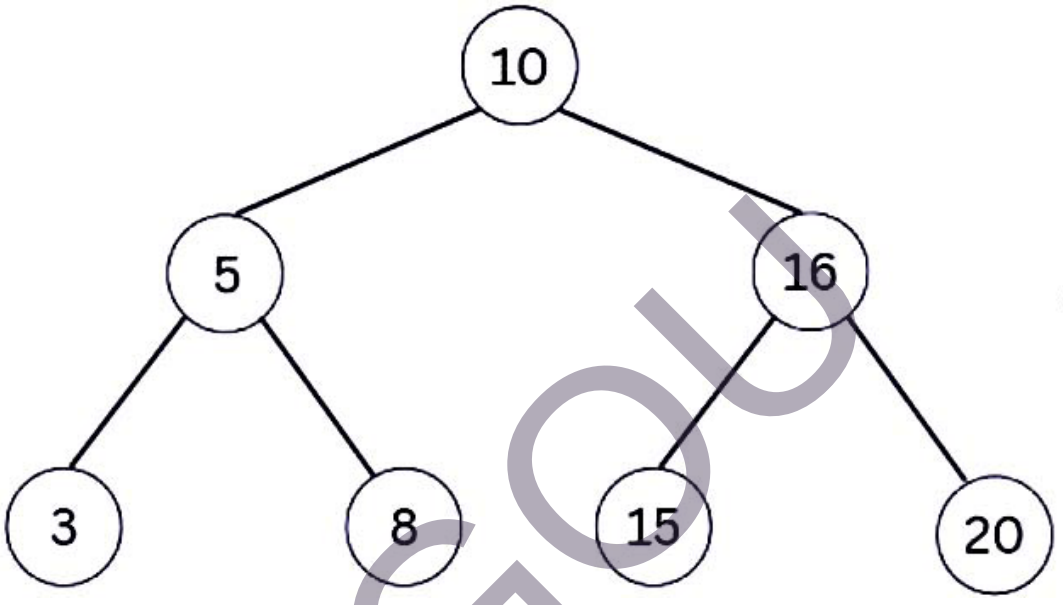


Fig. 3.2.2 BST example

4. **Recursive Structure:** Both the left and right subtrees of a node must themselves be valid BSTs.

Example 2: In the above tree;

- Left subtree rooted at 5 is a BST because $3 < 5 < 8$.
- Right subtree rooted at 16 is a BST because $15 < 16 < 20$.

5. **No Duplicate Keys (in standard BSTs):** Typically, a BST does not allow duplicate values to avoid ambiguity in searching and ordering.

Example 3: Inserting another 5 into the above BST is not allowed in the standard implementation. Some BSTs store duplicates in the left or right subtree by a specific rule, or maintain a count field in each node.

6. Sorted Order Traversal (Inorder Traversal) : An inorder traversal (Left → Node → Right) of a BST yields the keys in ascending order.

Example 4: Inorder of the above BST = 3,5,8,10,15,16,20 (sorted sequence).

7. Time Complexity (When Balanced): Operations like search, insertion, and deletion generally take O(log n) time when the tree is balanced.

Example 5: Searching for 8 in the above tree: 10 → 5 → 8 → Found (3 comparisons).

3.2.1.2 Applications of BST

Binary Search Trees (BSTs) play a vital role in computer science and programming because of their efficient way of storing, searching, and managing data. A BST organizes elements in a hierarchical structure where each node has a value greater than all values in its left subtree and smaller than those in its right subtree. This property makes operations such as searching, insertion, and deletion faster compared to linear data structures. Due to their efficiency, BSTs are widely applied in scenarios such as database indexing, expression evaluation, sorting, symbol table construction, and efficient storage of dynamic sets of data. Some of them are listed below.

- **Database Indexing :** Used to store and retrieve records efficiently.
- **Dictionary Implementation :** Enables quick search, insertion, and deletion of words.
- **Symbol Table in Compilers :** Stores identifiers for quick lookup during compilation.
- **File and Folder Management :** Helps in organizing directories for quick access.
- **Range Searching :** Efficiently finds all elements within a given range.
- **Auto-complete Features :** Supports predictive text search by maintaining sorted data.
- **Sorting :** Inorder traversal of BST produces elements in ascending order.

3.2.1.3 Advantages of BST

Binary Search Trees (BSTs) offer several advantages that make them one of the most important data structures in computer science. Some of the advantages are:

- **Efficient Searching :** Provides O(log n) search time in balanced trees.
- **Dynamic Data Handling :** Allows insertion and deletion at any time without reallocation like arrays.
- **Sorted Data Storage :** Maintains elements in a sorted manner naturally.
- **Supports Range Queries :** Quickly finds all elements within a given range.
- **Flexible Traversals :** Can be traversed in different orders for various purposes.

3.2.1.4 Disadvantages of BST

While Binary Search Trees (BSTs) are powerful and efficient in many cases, they also come with certain limitations. Some of the disadvantages are:

- **Performance Degradation if Unbalanced:** In worst cases (skewed tree), time complexity becomes O(n).
- **No Automatic Balancing:** Standard BSTs do not self-balance (requires AVL, Red-Black Tree for that).
- **Duplicate Handling:** Managing duplicate values requires special rules or extra space.
- **Extra Memory Usage:** Requires pointers for left and right child, increasing memory usage compared to arrays.

3.2.2 Operations of BST

Binary Search Trees (BSTs) provide an efficient way to store and manage data in a sorted, hierarchical structure. To fully utilize a BST, it is important to understand the key operations that can be performed on it, including searching for a specific value, inserting new elements, deleting existing elements, and traversing the tree in different orders. These operations leverage the BST's ordering property to ensure that data can be accessed, added, or removed quickly, making BSTs a fundamental structure for dynamic datasets and applications such as databases, dictionaries, and file systems.

3.2.2.1 Binary Search Trees - Searching

Searching in a Binary Search Tree (BST) is one of the most fundamental operations, taking advantage of the tree's ordered structure to locate a specific element efficiently. In a BST, the left subtree of a node contains smaller values and the right subtree contains larger values, which allows the search process to systematically eliminate half of the remaining elements at each step. This makes searching in a BST faster than in an unsorted binary tree or a linear data structure, especially when the tree is balanced. The operation is widely used in applications such as databases, dictionaries, and symbol tables where quick data retrieval is essential.

Steps for Searching in a BST

1. Begin the search from the root node of the BST.
2. Compare the key you want to find with the key of the current node.
3. If the key matches the current node's key, the element is found.
4. If the key is smaller than the current node's key, move to the left subtree.
5. If the key is larger than the current node's key, move to the right subtree.
6. Repeat this process until the key is found or a NULL node is reached, indicating the element is not present in the tree.

Source code

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

// Search function
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->key == key)
        return root;
    if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}
```

Example 6: Consider the fig 3.2.3 BST,

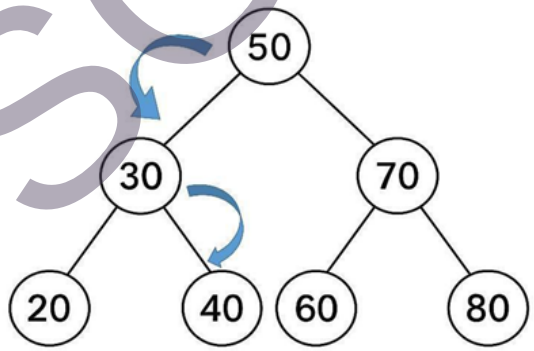


Fig. 3.2.3 BST searching

Search for 40 :

1. Start at root (50). Since 40 < 50, move to the left subtree.
2. At node 30. Since 40 > 30, move to the right subtree.
3. At node 40, match found. Element *40 is present in the BST.*

3.2.2.2 Binary Search Trees - Traversal

Traversal in a Binary Search Tree (BST) refers to the process of visiting all the nodes in a specific order to access or display the data. Different traversal methods, such as inorder, preorder, postorder, and level-order, serve different purposes: inorder traversal produces elements in sorted order, preorder is useful for copying the tree, postorder is helpful for deleting the tree, and level-order visits nodes level by level. Understanding these traversal techniques is essential for efficiently performing operations like searching, printing, or processing data stored in a BST.

Traversal in BST:

1. **Inorder (LNR):** Visit the left subtree first, then the node, and finally the right subtree; this produces the elements in sorted order.

2. **Preorder (NLR):** Visit the node first, then the left subtree, and finally the right subtree; this is useful for copying the tree.

3. **Postorder (LRN):** Visit the left subtree first, then the right subtree, and finally the node; this is helpful for deleting the tree.

Source code

```
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->key);
    }
}
```

Example 7: Consider the example 6, then traversal be

1. Inorder (LNR → Left, Node, Right): 20, 30, 40, 50, 60, 70, 80
2. Preorder (NLR → Node, Left, Right): 50, 30, 20, 40, 70, 60, 80
3. Postorder (LRN → Left, Right, Node): 20, 40, 30, 60, 80, 70, 50

3.2.2.3 Binary Search Trees – Insertion

Insertion in a Binary Search Tree (BST) involves placing a new node into the tree in a way that maintains its hierarchical ordering. By comparing the new key with existing nodes and moving left or right accordingly, the BST ensures that smaller values are always in the left subtree and larger values in the right. This careful placement allows the tree to remain organized, supporting fast searching, deletion, and traversal operations, and is fundamental for efficiently managing dynamic datasets.

Steps for Insertion in a BST:

1. If the tree is empty, create a new node and make it the root.
2. Compare the key to be inserted with the key of the current node.
3. If the key is smaller than the current node's key, move to the left subtree.
4. If the key is larger than the current node's key, move to the right subtree.
5. Repeat this process until a suitable NULL position is found.
6. Insert the new node at that position.

Source code

```
struct Node* createNode(int key) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int key) {
    if (root == NULL) return createNode(key);
    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
     root->right = insert(root->right, key);
  return root;
}
```

Example 8 : Consider the example 6, insert 65 into this BST by the following steps (fig 3.2.4).

1. Start at the root (*50*). Since 65 > 50, move to the right subtree.
2. At node (*70*). Since 65 < 70, move to the left subtree.
3. At node (*60*). Since 65 > 60, move to the right.
4. The right child of 60 is empty → insert *65* here.

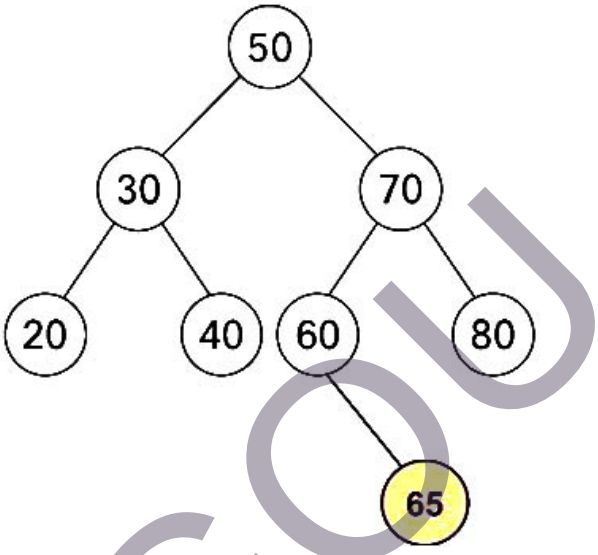


Fig. 3.2.4 BST insertion

3.2.2.4 Binary Search Trees - Deletion

Deletion in a Binary Search Tree (BST) is the process of removing a node while preserving the tree's ordered structure. Depending on whether the node has no children, one child, or two children, different strategies are applied to ensure that the BST property is maintained. Proper deletion is essential for keeping the tree organized and efficient for subsequent operations such as searching, insertion, and traversal, making it a critical operation in dynamic data management.

Steps for Deletion in a BST:

1. Locate the node that needs to be deleted.
2. **Case 1 :** If the node has no children, simply delete it.
3. **Case 2 :** If the node has one child, replace the node with its child.
4. **Case 3 :** If the node has two children, first find its inorder successor (the smallest value in the right subtree). Replace the key of the node to be deleted with the key of its inorder successor. Finally, delete the inorder successor node from its original position.

Source code

```
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) return root;
    if (key < root ->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
   free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        // Node with two children
        struct Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

Example 9 : Consider example 6, delete 70. Node 70 has two children (60 and 80), so we apply Case 3 (fig 3.2.5).

Step 1: Find the *inorder successor* of 70 → that is *80* (smallest in right subtree).

Step 2: Replace 70 with 80.

Step 3: Delete the duplicate 80 node.

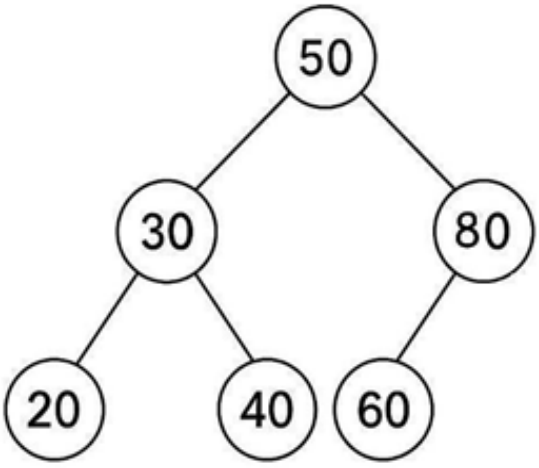


Fig. 3.2.5 BST deletion

3.2.3 Balanced Search Trees

Balanced Search Trees are specialized forms of binary search trees designed to maintain a roughly equal height across their subtrees. By keeping the tree balanced, these structures ensure that operations such as searching, insertion, and deletion remain efficient, typically in O(log n) time. This prevents performance degradation that can occur in unbalanced BSTs, where the tree may become skewed like a linked list. Balanced trees, such as AVL Trees and Red-Black Trees, are widely used in applications like databases, file systems, and memory indexing where consistent and fast access to data is critical. A balanced binary tree, also referred to as a height-balanced binary tree. Following are the conditions for a height-balanced binary tree (fig 3.2.6):

- difference between the left and the right subtree for any node is not more than one.
- the left subtree is balanced.
- the right subtree is balanced.

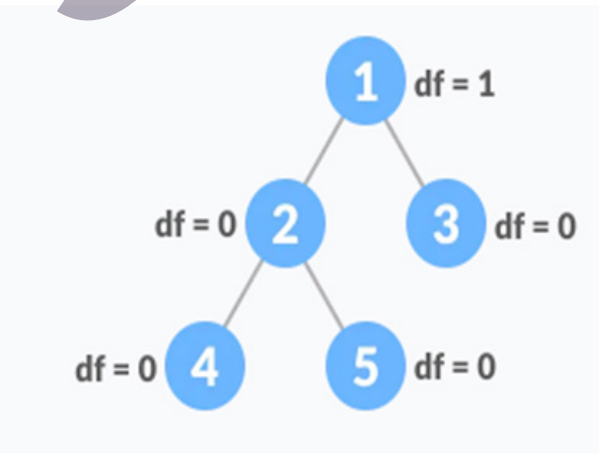


Fig. 3.2.6 Balanced Binary Tree with depth at each level

3.2.3.1 Advantages of Balanced Binary Tree

Balancing a binary tree is essential for maintaining its efficiency and performance. When a binary tree remains balanced, it keeps its height minimal, ensuring that operations like searching, inserting, and deleting nodes can be performed quickly and effectively. Without balancing, the tree can become skewed, turning into a linear structure that slows down these operations. Balanced binary trees, such as AVL trees and Red-Black trees, are widely used in applications that require fast data access and manipulation. Understanding the advantages of tree balancing helps highlight its importance in optimizing time compexity in computer programs.

1. **Faster Searching:** A balanced binary tree keeps its height low, ensuring that search operations take O(log n) time instead of O(n), as in an unbalanced tree.

2. **Efficient Insertions and Deletions:** Operations like inserting or deleting nodes are faster in a balanced tree because fewer nodes need to be visited or adjusted.

3. **Prevents Skewed Structure:** Balancing avoids the formation of skewed trees (like linked lists), which can degrade performance and increase time complexity and recursion depth.

4. **Improved Performance in Large Data Sets:** Balanced trees handle large volumes of data more effectively, making them ideal for applications like databases, search engines, and file systems.

5. **Predictable Time Complexity:** Balancing ensures consistent logarithmic time complexity, which improves the reliability of programs that depend on tree operations.

6. **Supports Real-Time Applications:** Because of predictable and fast response times, balanced trees are useful in real-time systems such as navigation, robotics, or mobile applications.

3.2.3.2 Disadvantages of Balanced Binary Tree

Balanced Binary Trees, such as AVL trees and Red-Black trees, are designed to keep data well-structured and operations efficient. However, maintaining balance comes with certain drawbacks. The process of rebalancing requires additional time and complex rotations during insertion and deletion, which increases overhead. They also consume extra memory for storing balance factors or additional information, making them less space-efficient compared to simple binary search trees.

1. More complex to implement than a standard BST.

2. Insertion and deletion may require *rebalancing,* which adds overhead.

3. Requires extra memory to store balance information (like height or color).

3.2.3.3 Applications of Balanced Binary Tree

Balanced Binary Trees, such as AVL trees and Red-Black trees, are widely used in computer science because they maintain data in a sorted and well-structured manner. Their ability to keep operations like searching, insertion, and deletion efficient makes them highly useful in real-world applications. They are commonly applied in database indexing, memory management, file systems, and implementing dynamic sets where quick access and updates are required.

- **AVL Trees :** Used where frequent searches are needed with consistent speed.
- **Red-Black Trees :** Used in operating systems, language libraries, and database indexing.
- **B-Trees and B+ Trees :** Balanced multiway search trees suitable for **disk-based storage** in databases and file systems.

3.2.4 AVL Tree

An *AVL Tree* is a type of self-balancing binary search tree named after its inventors *Adelson-Velsky and Landis*. In an AVL tree, the heights of the left and right subtrees of every node differ by at most one, ensuring that the tree remains balanced. This balance property allows *search, insertion, and deletion* operations to be performed efficiently in *O(log n)* time. AVL trees are widely used in applications that require fast and predictable search performance, such as databases, indexing systems, and memory management.

AVL trees are a type of self-balancing binary search tree that maintain their structure through a strict balancing condition. Unlike regular binary search trees, which can become skewed and inefficient over time, AVL trees automatically adjust themselves after each insertion or deletion to preserve balance. This ensures that the height of the tree remains logarithmic in relation to the number of nodes, allowing for consistently efficient performance in search, insert, and delete operations. The key properties of AVL trees make them highly suitable for applications where data access speed and structure balance are critical.

- An AVL tree is a self-balancing binary search tree where the height difference between the left and right subtrees of any node is at most one.
- Every node in an AVL tree maintains a balance factor calculated as the height of the left subtree minus the height of the right subtree, which must be -1, 0, or +1.
- The AVL tree maintains the binary search tree property, meaning the left child contains values less than the parent node and the right child contains values greater than the parent.
- AVL trees automatically perform rotations (single or double) during insertion and deletion operations to maintain balance.

3.2.4.1 Balance Factor

The balance factor is a key concept in AVL trees used to determine whether a node is balanced. It is defined as the difference between the heights of the left and right subtrees of a node.

Balance Factor (BF) = Height of Left Subtree − Height of Right Subtree

Possible Values of Balance Factor :

- **+1** : Left subtree is one level taller than the right.
- **0** : Left and right subtrees are of equal height.
- **–1** : Right subtree is one level taller than the left.

If the balance factor of any node becomes less than -1 or greater than +1, the tree is considered unbalanced, and rotations (single or double) must be performed to restore balance.

Example 10 : To find a balance factor, consider the below figure 3.2.7.

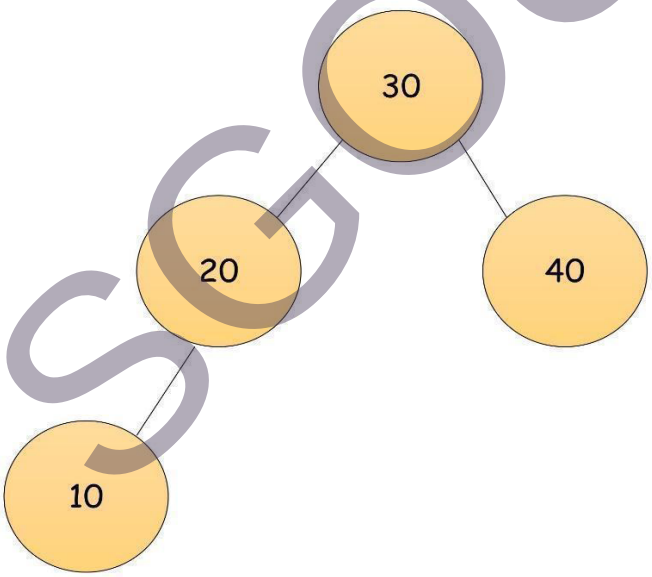


Fig. 3.2.7 Example diagram to find balance factor

By the equation,

Balance Factor (BF) = Height(Left Subtree) - Height(Right Subtree)

- Balance Factor of node 40 = 0 - 0 = *0* (Balanced)
- Balance Factor of node 30 = 2 - 1 = *+1* (Balanced)
- Balance Factor of node 20 = 1 - 0 = *+1* (Balanced)
- Balance Factor of node 10 = 0 - 0 = *0* (Balanced)

3.2.4.2 Rotations of AVL

In AVL trees, maintaining balance is crucial to ensure optimal performance in search, insertion, and deletion operations. Whenever an insertion or deletion causes the balance factor of any node to become greater than +1 or less than -1, the tree becomes unbalanced. To restore balance, rotations are performed. These rotations are specific tree restructuring operations that rearrange nodes without violating the binary search tree properties. Depending on the type and direction of imbalance, AVL trees use single or double rotations, making them a fundamental part of keeping the tree height-balanced. There are usually four cases of rotation in the balancing algorithm of AVL trees: LL, RR, LR, RL.

a. LL Rotation

LL rotation (Fig 3.2.8) is performed when the node is inserted into the right subtree leading to an unbalanced tree. This is a single left rotation to make the tree balanced again.

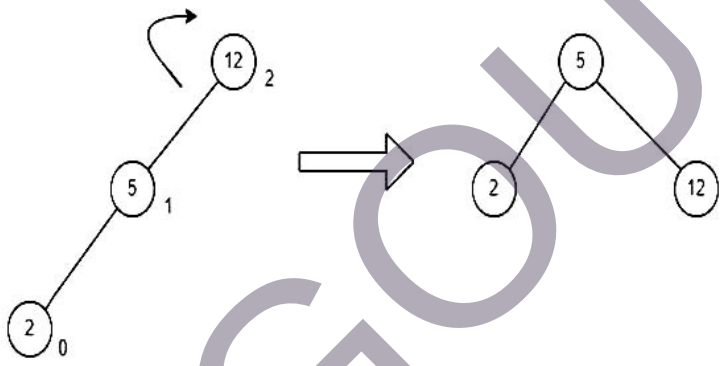


Fig. 3.2.8 LL Rotation

The node where the unbalance occurs becomes the left child and the newly added node becomes the right child with the middle node as the parent node.

b. RR Rotation

RR rotation (Fig 3.2.9) is performed when the node is inserted into the left subtree leading to an unbalanced tree. This is a single right rotation to make the tree balanced again.

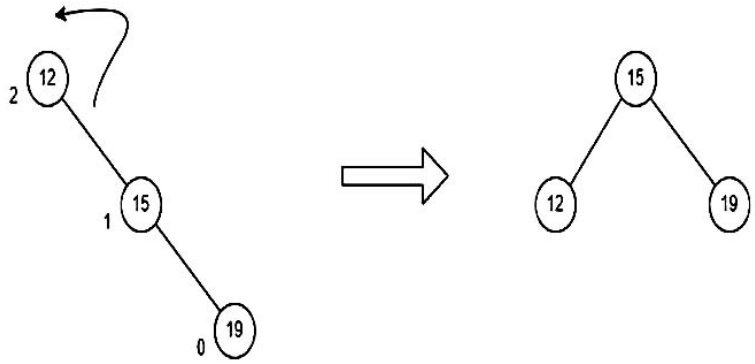


Fig. 3.2.9 RR Rotation

The node where the unbalance occurs becomes the right child and the newly added node becomes the left child with the middle node as the parent node.

c. LR Rotation

LR rotation (Fig 3.2.10) is the extended version of the previous single rotations, also called a double rotation. It is performed when a node is inserted into the right subtree of the left subtree. The LR rotation is a combination of the left rotation followed by the right rotation. There are multiple steps to be followed to carry this out.

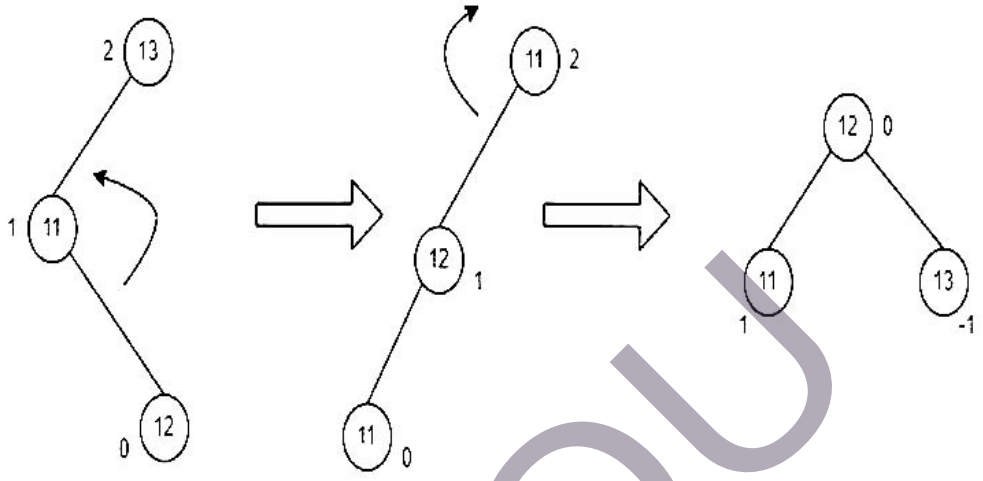


Fig. 3.2.10 LR Rotation

d. RL Rotation

RL rotation (Fig 3.2.11) is also the extended version of the previous single rotations, hence it is called a double rotation and it is performed if a node is inserted into the left subtree of the right subtree. The RL rotation is a combination of the right rotation followed by the left rotation. There are multiple steps to be followed to carry this out.

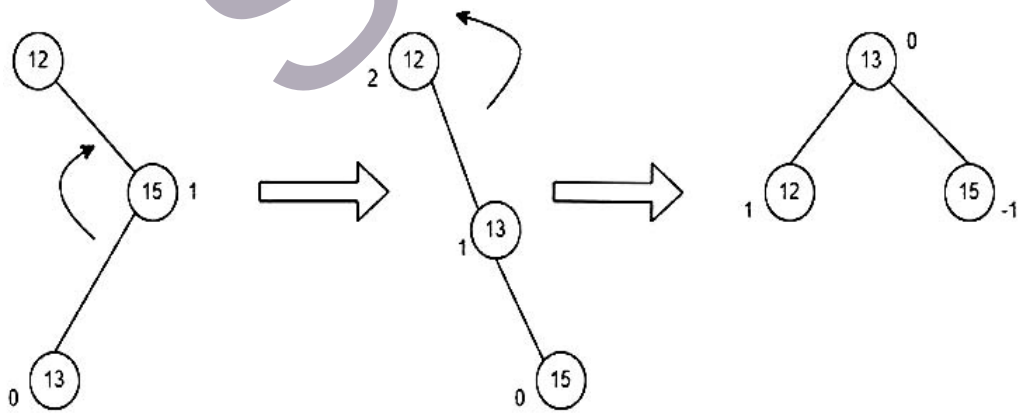


Fig. 3.2.11 RL Rotation

3.2.4.3 Basic Operations of AVL Trees

The basic operations performed on the AVL Tree structures include all the operations performed on a binary search tree, since the AVL Tree at its core is actually just a binary search tree holding all its properties. Therefore, basic operations performed on an AVL Tree are Insertion and Deletion.

a. Insertion Operation

The data is inserted into the AVL Tree by following the Binary Search Tree property of insertion, i.e. the left subtree must contain elements less than the root value and right subtree must contain all the greater elements. However, in AVL Trees, after the insertion of each element, the balance factor of the tree is checked; if it does not exceed 1, the tree is left as it is. But if the balance factor exceeds 1, a balancing algorithm is applied to readjust the tree such that balance factor becomes less than or equal to 1 again.

Algorithm: The following steps are involved in performing the insertion operation of an AVL Tree.

Step 1: Create a node

Step 2: Check if the tree is empty

Step 3: If the tree is empty, the new node created will become the root node of the AVL Tree.

Step 4: If the tree is not empty, we perform the Binary Search Tree insertion operation and check the balancing factor of the node in the tree.

Step 5: Suppose the balancing factor exceeds 1, we apply suitable rotations on the said node.

Example 11 : Let us understand the insertion operation by constructing an example AVL tree with 1 to 7 integers. Starting with the first element 1 (Fig 3.2.12), we create a node and measure the balance, i.e., 0.

Fig. 3.2.12 Insert the node 1

Since both the binary search property and the balance factor are satisfied, we insert another element into the tree (Fig 3.2.13).

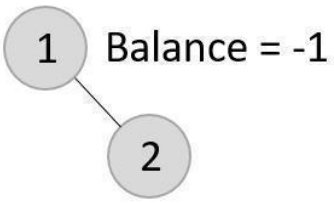


Fig. 3.2.13 Insert the node 2 and find balance factor

The balance factor for the two nodes are calculated and is found to be -1 (Height of left subtree is 0 and height of the right subtree is 1). Since it does not exceed 1, we add another element to the tree (Fig 3.2.14).

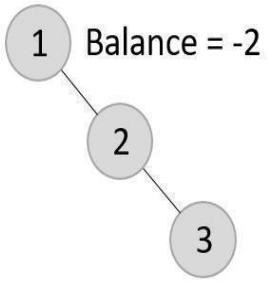


Fig. 3.2.14 Insert the node 3 and find balance factor

Now, after adding the third element, the balance factor exceeds 1 and becomes 2. Therefore, rotations are applied. In this case, the RR rotation is applied since the imbalance occurs at two right nodes (Fig 3.2.15).

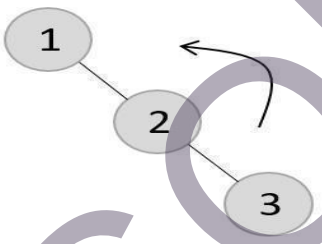


Fig. 3.2.15 RR rotation

The tree is rearranged as shown in Fig. 3.2.16.

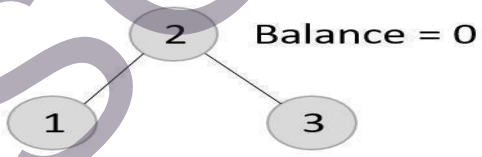


Fig 3.2.16 Rearranged nodes after RR rotation

Similarly, the next elements are inserted and rearranged using these rotations. After rearrangement, we achieve the tree as shown in Fig 3.2.17.

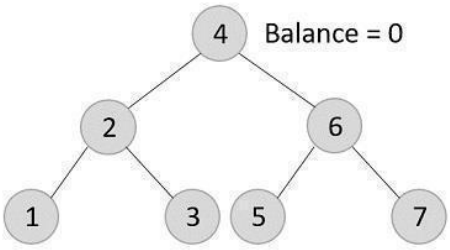


Fig. 3.2.17 Final result

b. Deletion Operation

Deletion in AVL trees involves removing a node while maintaining the tree's balanced structure. Similar to deletion in a standard binary search tree, the node is removed based on its position (leaf, one child, or two children). However, in an AVL tree, the deletion may disturb the height balance of the tree, causing some nodes to become unbalanced. To fix this, balance factors are updated as the algorithm moves up the tree, and rotations (single or double) are applied wherever necessary. This ensures that the AVL tree continues to provide efficient operations with O(log n) time complexity. Deletion in the AVL Trees take place in three different scenarios.

- **Scenario 1 (Deletion of a leaf node) :** If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However, the balance factor may get disturbed, so rotations are applied to restore it.
- **Scenario 2 (Deletion of a node with one child) :** If the node to be deleted has one child, replace the node pointer with its child. Then delete the child node. If the balance factor is disturbed, rotations are applied.
- **Scenario 3 (Deletion of a node with two child nodes) :** If the node to be deleted has two child nodes, find the inorder successor of that node and replace its value with the inorder successor value. Then try to delete the inorder successor node. If the balance factor exceeds 1 after deletion, apply balance algorithms.

Example 12: Using the fig 3.2.17 tree structure, let us perform deletion in three scenarios.

Scenario 1: Deleting element 7 from the tree above figure 3.2.17. Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree fig 3.2.18.

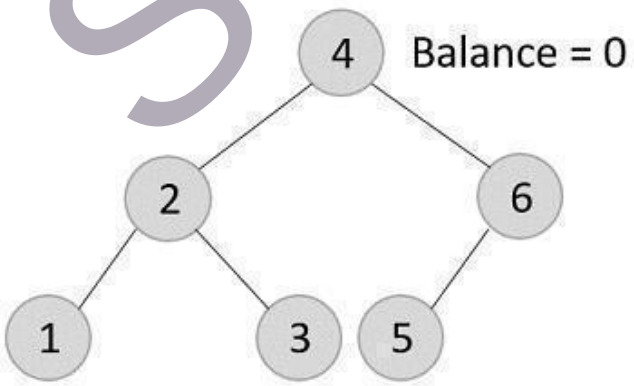


Fig. 3.2.18 Result of scenario 1

Scenario 2: Deleting element 6 from the output tree achieved. However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node (node 5) fig 3.2.19.

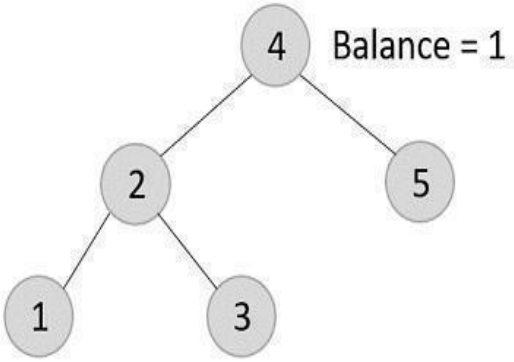


Fig 3.2.19 Result of scenario 2

The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is. If we delete element 5 further, we would have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4 in fig 3.2.20.

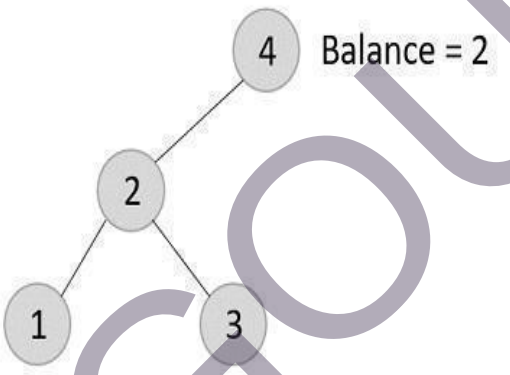


Fig. 3.2.20 Delete element 5

The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here) fig 3.2.21.

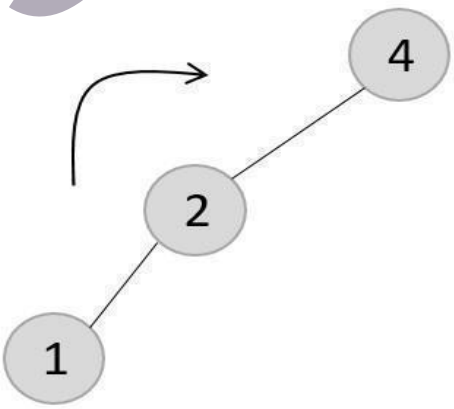


Fig. 3.2.21 LL rotation

Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4 in fig 3.2.22.

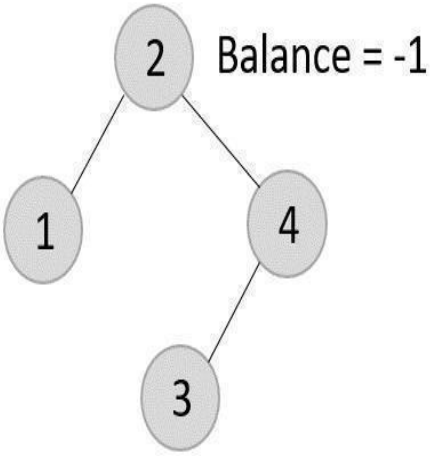


Fig. 3.2.22 Final Result

Scenario 3: Deleting element 2 from the remaining tree. As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor in fig 3.2.23.

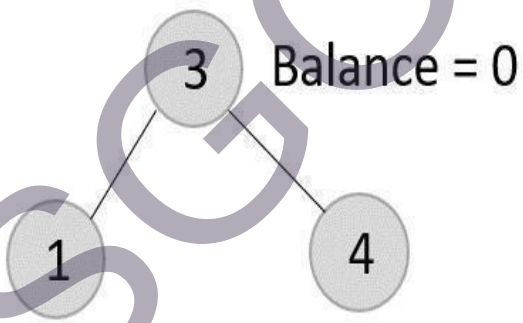


Fig. 3.2.23 Final result of scenario 3

The balance of the tree still remains 1, therefore we leave the tree as it is without performing any rotations.

3.2.4.4 Applications of AVL Trees

AVL trees are widely used in areas where fast and efficient data access is critical, due to their self-balancing nature and guaranteed logarithmic time operations. By maintaining a balanced structure at all times, AVL trees ensure consistent performance for search, insert, and delete operations, even with large and dynamic datasets. Their ability to quickly adjust to changes makes them suitable for real-time applications, database indexing, memory management, and any system where maintaining sorted data with optimal access speed is essential.

1. **Memory Management:** Operating systems and compilers use AVL trees for dynamic memory allocation. They help in tracking free and used memory blocks efficiently by maintaining a balanced tree of memory segments.

2. **Search Engines:** AVL trees can be used to implement search dictionaries or inverted indexes, allowing quick word lookups or phrase matching.

3. **Routing Tables in Networks:** AVL trees are used in networking for managing routing tables, where fast lookup of routes and paths is essential.

4. **Gaming Applications:** Games use AVL trees in decision-making logic (e.g., AI pathfinding) where frequent searches and updates are needed.

5. **Symbol Tables in Compilers:** Compilers use AVL trees to manage symbol tables, allowing quick insertions, deletions, and lookups of variables and function names.

6. **Autocompletion and Spell Checkers:** AVL trees help in storing dictionaries or word lists where fast prefix searching and retrieval is important.

7. **Real-Time Systems:** In real-time systems where guaranteed performance is required, AVL trees ensure operations complete in predictable time (O(log n)).

3.2.5 M-way Search Trees

An m-way search tree is a type of self-balanced, multi-way tree that generalizes binary search trees to improve search efficiency. In a tree of order *m*, each node can hold up to m - 1 elements and have up to m children . These trees are designed to optimize operations like search, insertion, and deletion, achieving a best-case time complexity of $O(\log_m(n))$. A binary search tree is simply a special case of an m-way tree where m = 2, meaning each node has at most two children as in fig 3.2.24.

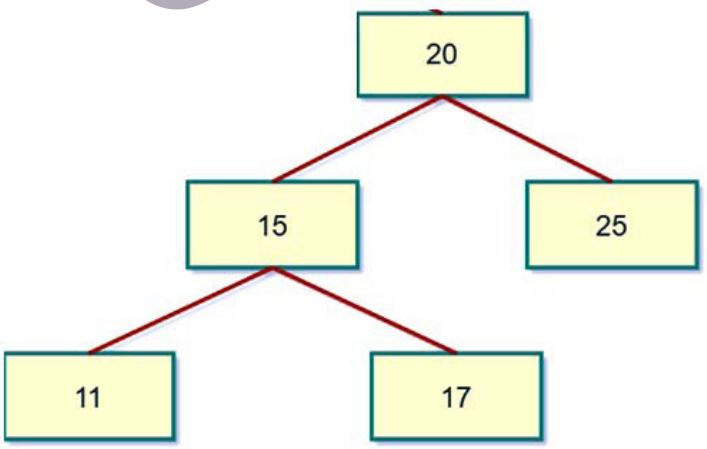


Fig. 3.2.24 2-Way Search Tree

3.2.5.1 Properties of M-way Search Trees

M-way search trees have several key properties that define their structure and behavior:

1. **Children :** Each node can have up to m children, making it a multi-way tree.

2. **Elements per Node :** A node can contain a maximum of m - 1 elements.

3. **Tree Height :** The height of the tree ranges from a minimum of $\log_m(n + 1)$ (when the tree is balanced) to a maximum of n (in the worst-case unbalanced scenario).

4. **Search Efficiency** : In the best case, searching takes $O(\log_m(n))$ time. However, in the worst case, it can degrade to O(n), resembling a linear search.

5. **Balancing :** To maintain optimal search time, balancing techniques are applied. For instance, B-Trees, a type of m-way search tree, are self-balancing and ensure that every node (except the root) has at least $\lceil m/2 \rceil$ children.

6. **Element Ordering :** Within a node, elements are kept in ascending order. The left subtree contains values less than the leftmost key, the right subtree contains values greater than the rightmost key, and any middle subtree lies between the keys it is adjacent to holding values greater than the left key and less than the right key.

3.2.5.2 Searching Algorithm

To locate a specific element in the tree, follow these steps:

- Start at the first element of the root node.
- Compare the current value with the target value:
  - If they match, the element has been found.
  - If not, proceed to the next step.
- If the target value is less than the current value:
  - Check whether a left subtree exists.
    - If it does, move to the first element of the left subtree and repeat the process from step 2.
    - If no left subtree exists, the element is not present in the tree.
- If the target value is greater than the current value:
  - Check if there is a next element within the current node.
    - If there is, and it is less than or equal to the target value, shift focus to this next element and return to step 2.
    - If it is greater than the target value, check for a right subtree linked to the current element:

- If such a right subtree exists, move into it and repeat from step 2.
- If not, the element does not exist in the tree.
    - If no next element exists:
        - Check for a right subtree linked to the current element.
        - If it exists, move into it and return to step 2.
        - If not, the target element is not in the tree.

Example 13 : 3-way search tree, m = 3 (fig 3.2.25).

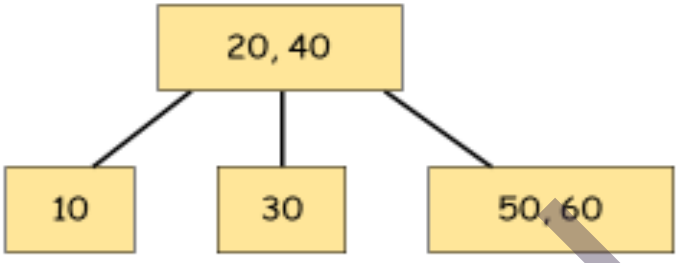


Fig. 3.2.25 3-Way Search Tree

Search for 30 (fig 3.2.26),

1. Start at node [20, 40].
2. 30 is greater than 20 and less than 40 then go to the middle child.
3. Visit node [30].
4. 30 matches a key in the node.

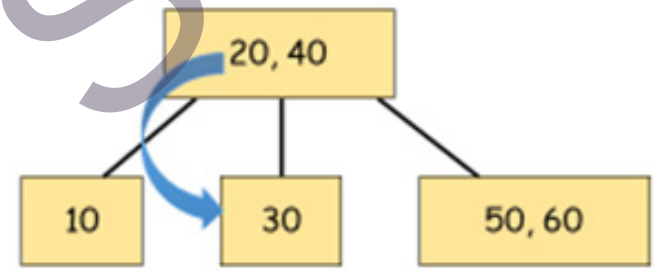


Fig 3.2.26 Search for 30 in 3-Way Search Tree

Search for 25 (fig 3.2.27)

1. Start at node [20, 40].
2. 25 is greater than 20 and less than 40 then go to the middle child.
3. Visit node [30].

4. 25 is less than 30 then move to the left child (null).

5. Child is null.

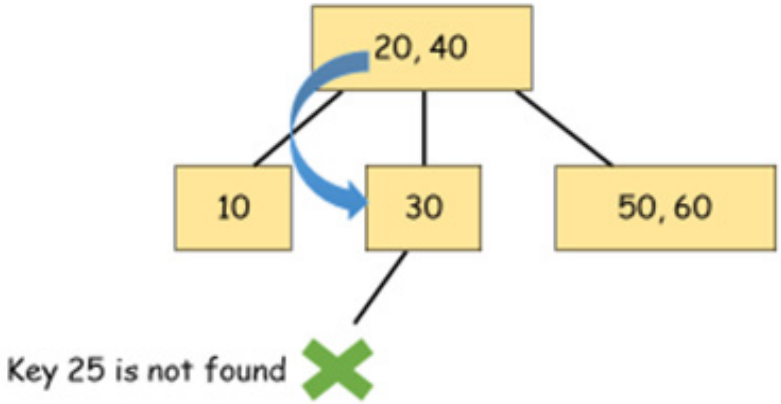


Fig. 3.2.27 Search for 25 in 3-Way Search Tree

3.2.6 B- Tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree. B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of a huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, etc.

Following are the features of a B-Tree:

- All leaves are at the same level.
- A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
- Every node except root must contain at least t-1 keys. Root may contain a minimum 1 key.
- All nodes (including root) may contain at most 2t – 1 keys.

- Number of children of a node is equal to the number of keys in it plus 1.
- All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.
- B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

It is also known as a height-balanced m-way tree. The following fig 3.2.28 shows a B tree. Here, the root node contains more than 1 key value (Here, we have 3 pointers and 2 key values). Also, the root node has more than two children.

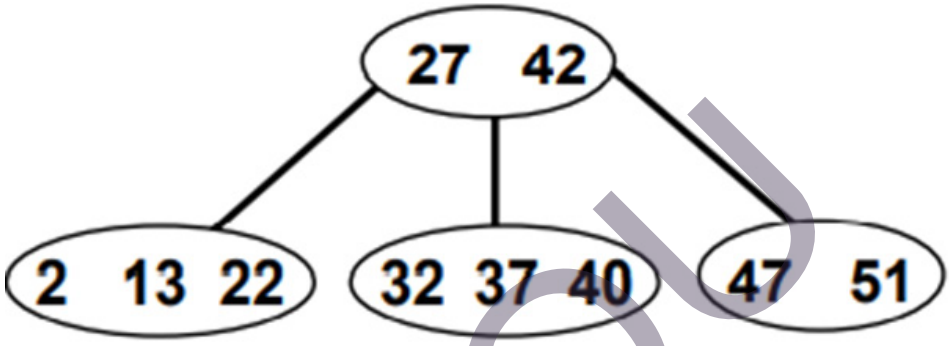


Fig. 3.2.28 B –Tree

The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity.

3.2.6.1 Operations on B-Tree

B-Trees support various operations that make them highly efficient for managing large datasets. Below are the key operations in table 3.2.1.

Table 3.2.1 Basic key operations and time complexity of B-Tree

Sl. No.	Operation	Time Complexity
1	Search	O(log n)
2	Insert	O(log n)
3	Delete	O(log n)

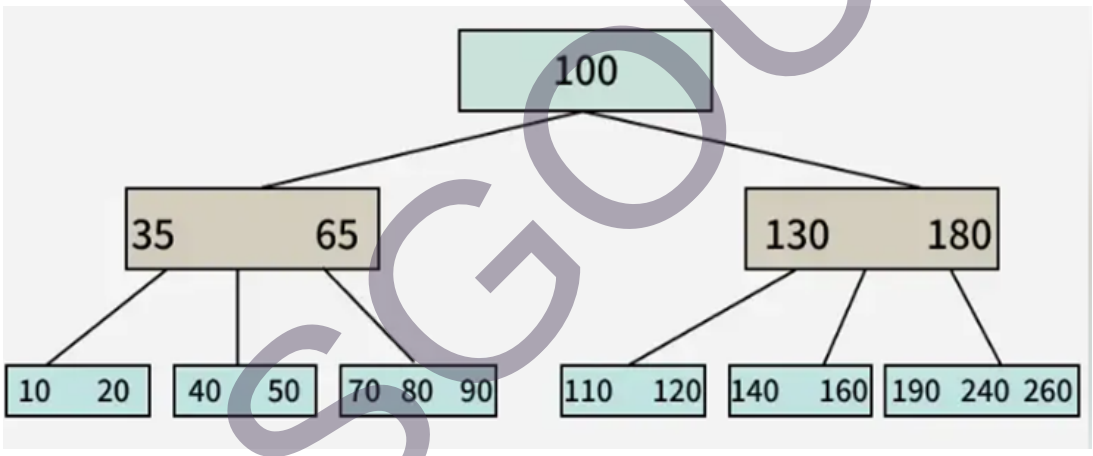
a. Search Operation in B-Tree

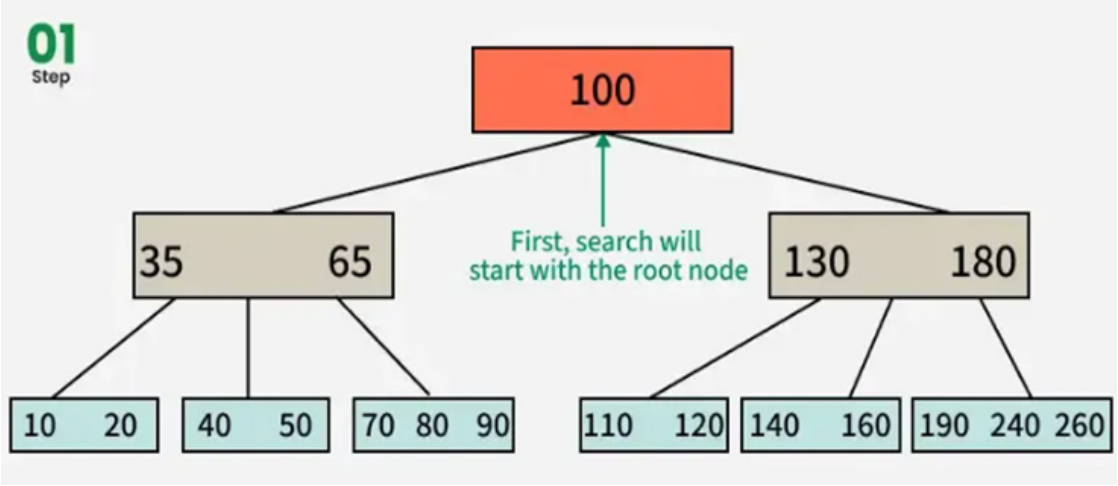
Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

- Start from the root and recursively traverse down.
- For every visited non-leaf node
  - If the current node contains k, return the node.
  - Otherwise, determine the appropriate child to traverse. This is the child just before the first key greater than k.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

Searching for a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

Example 14 : Search 120 in the given B-Tree in fig 3.2.29.





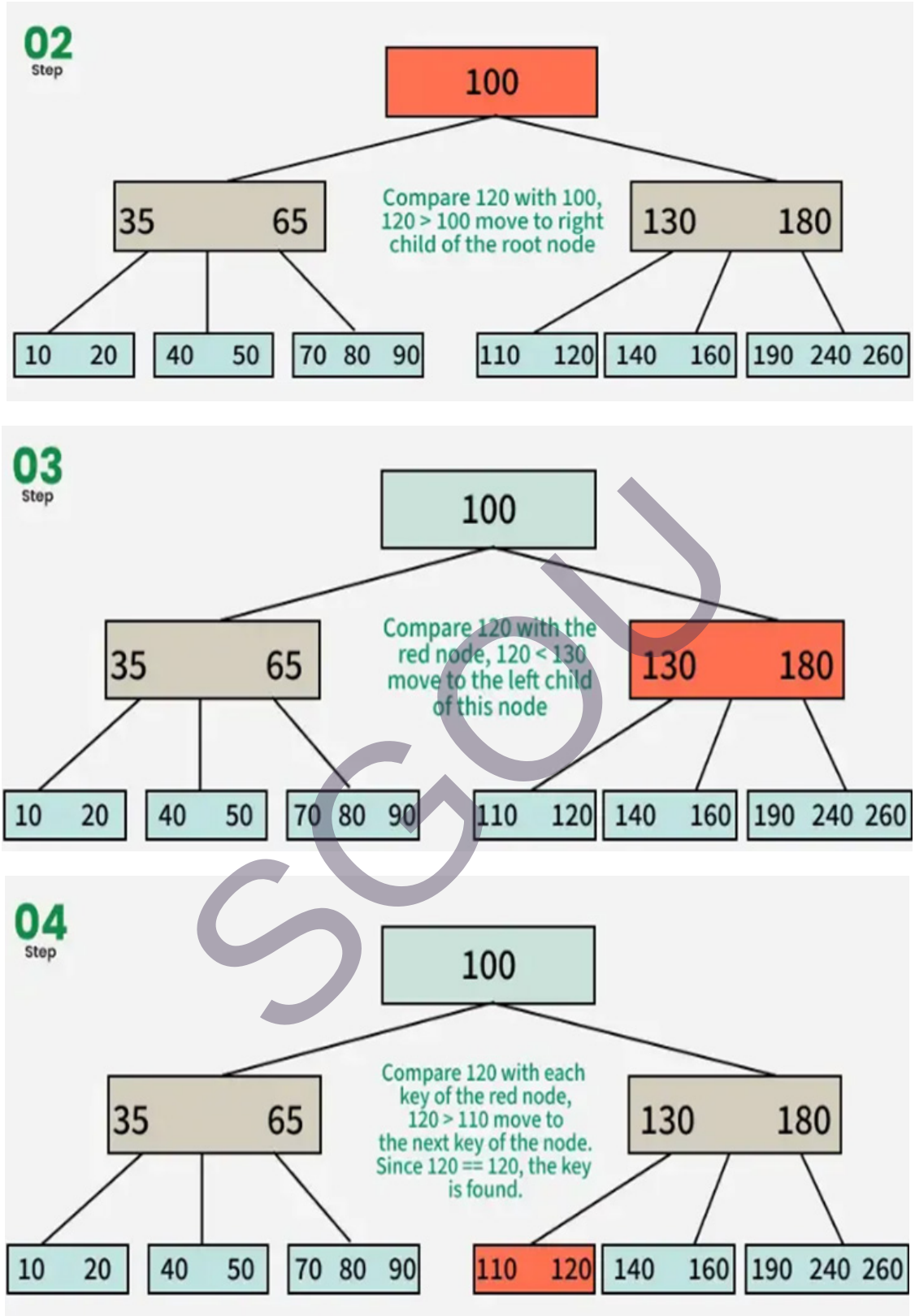


Fig. 3.2.29 Search operation of B –Tree

The key 120 is located in the leaf node containing 110 and 120. The search process is complete.

b. Insertion Operation

The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure.

Step1: Calculate the maximum (m−1) and, minimum (⌈m2⌉−1) number of keys a node can hold, where m is denoted by the order of the B Tree.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys (m − 1) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil$) − 1 = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

Step 2: The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children in fig 3.2.30.

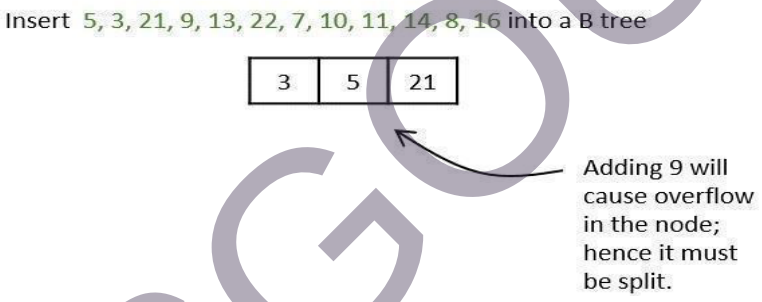


Fig. 3.2.30 Inserting operation

Step 3 : All the leaf nodes must be on the same level in fig 3.2.31.

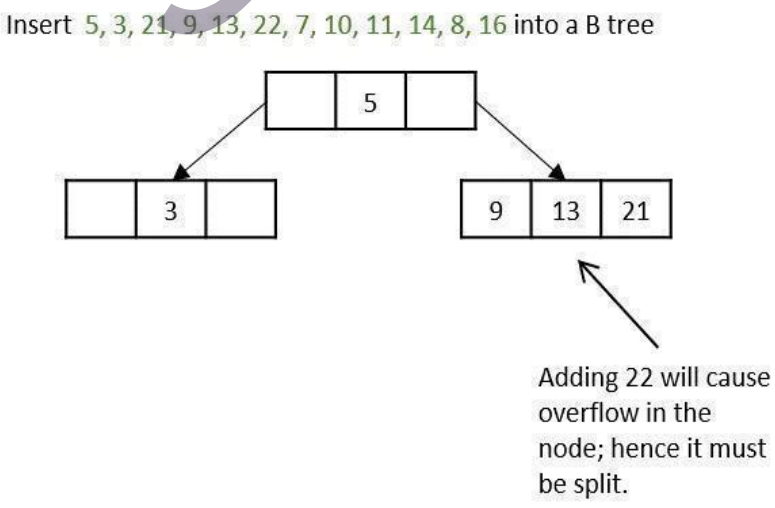


Fig. 3.2.31 Overflow occur in insert operation

Step 4 : The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued in fig 3.2.32.

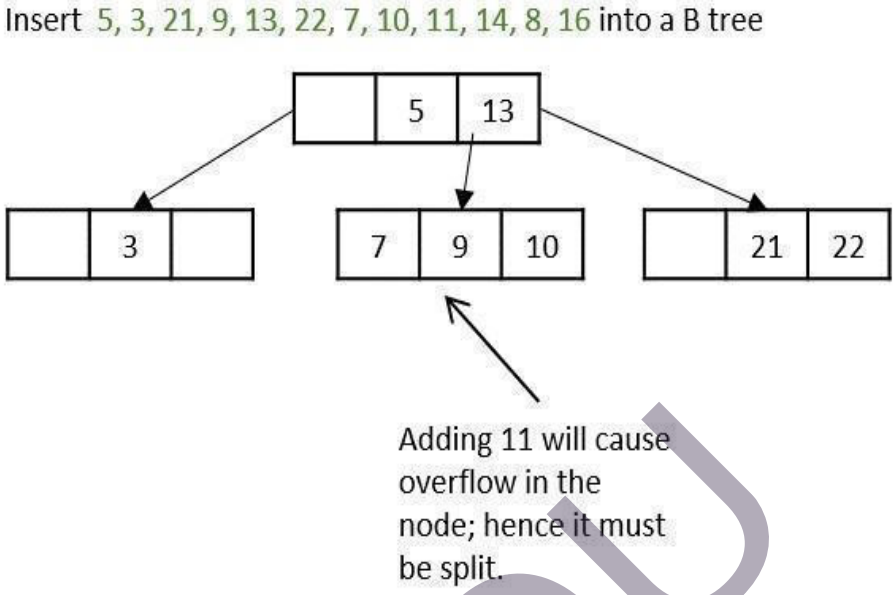


Fig. 3.2.32 Overflow occur while adding 11

Another problem occurs during the insertion of 11, so the node is split and median is shifted to the parent.

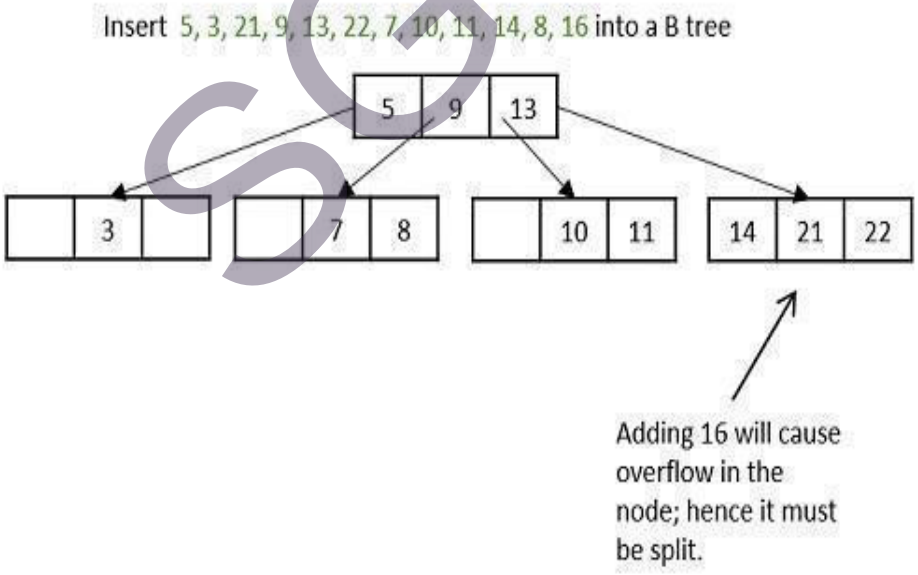


Fig. 3.2.33 Overflow occur in insertion of 16

While inserting 16, even if the node is split in two parts, the parent node also overflows (fig 3.2.33) as it reached the maximum keys. Hence, the parent node is split first, and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

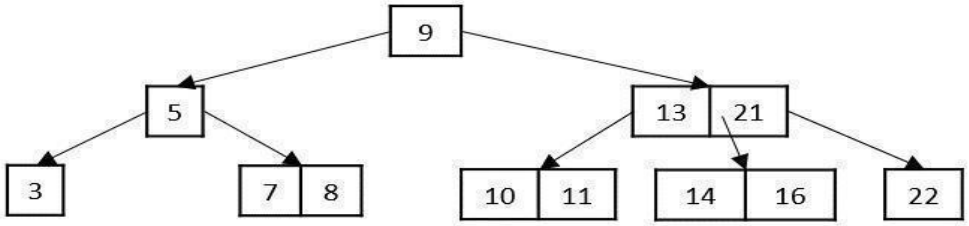


Fig. 3.2.34 Final Result

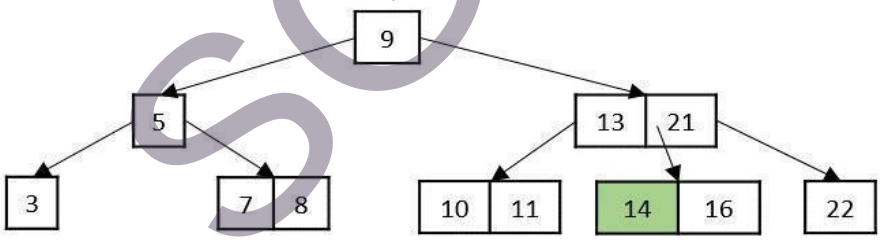
The final B tree after inserting all the elements is achieved in fig 3.2.34.

c. Deletion Operation

The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows:

Case 1 : If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node in fig 3.2.35.

Delete key 14



Delete key 14

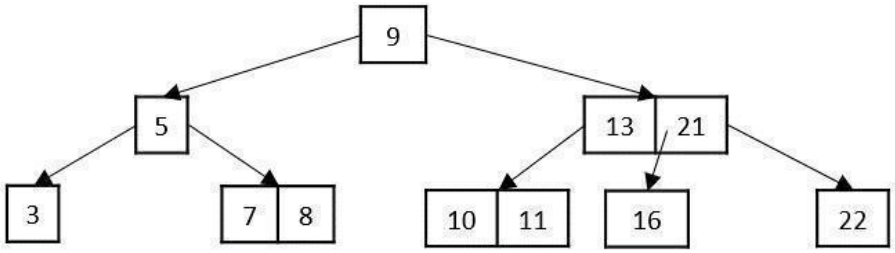


Fig. 3.2.35 Delete key 14

Case 2: If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them in fig 3.2.36.

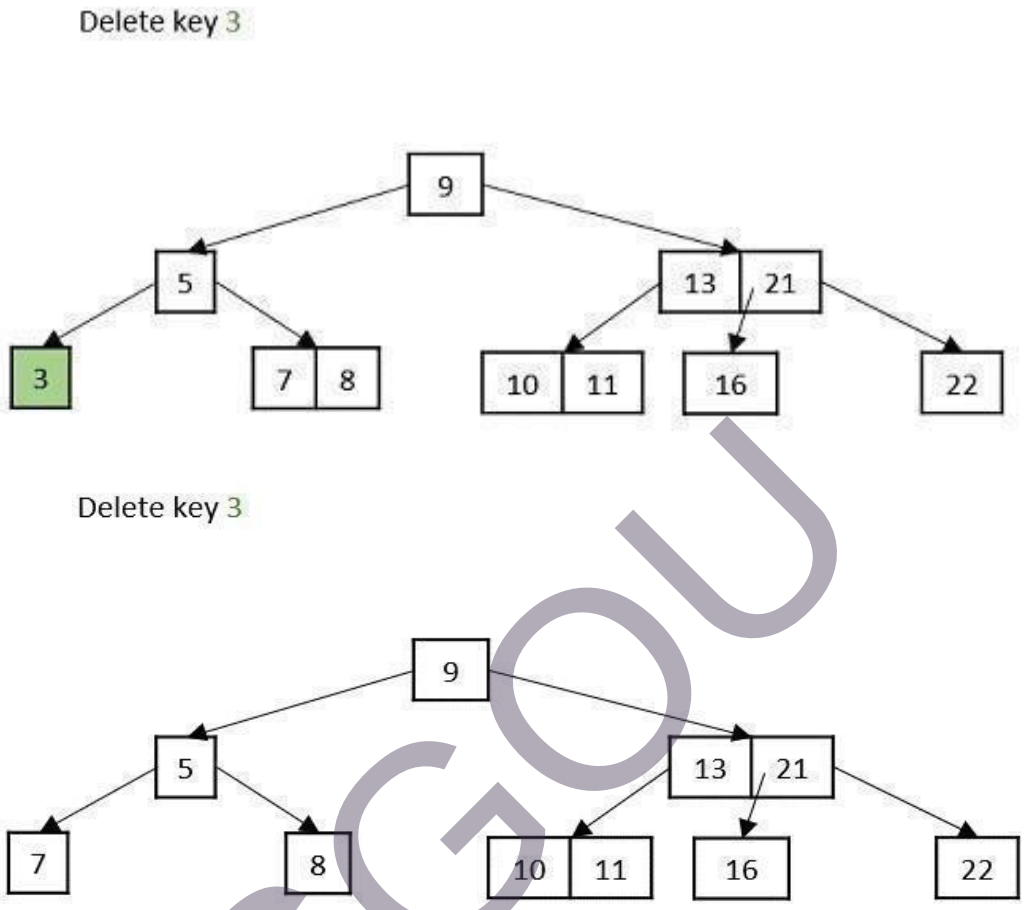
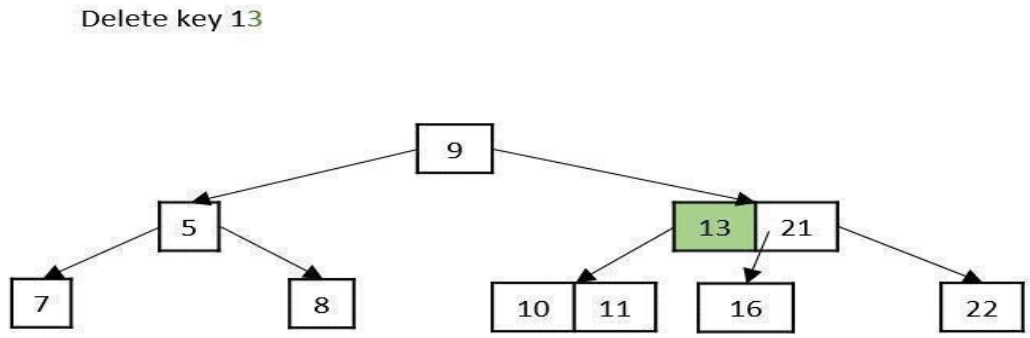


Fig. 3.2.36 Delete key 3

Case 3: If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have a minimum number of keys, they're merged together in fig 3.2.37.



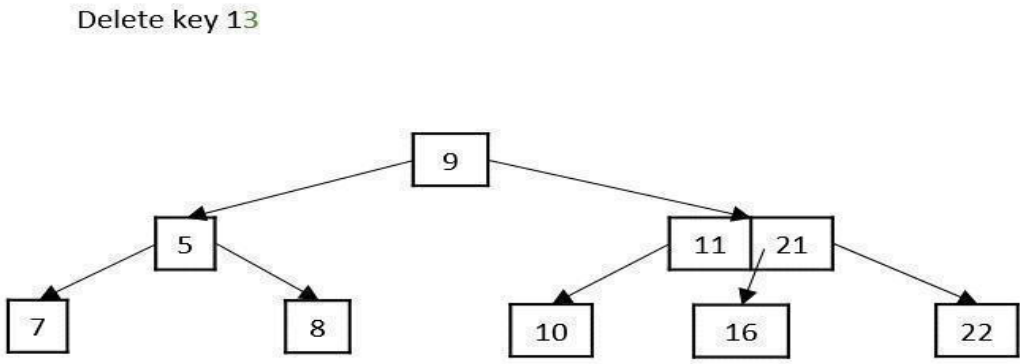


Fig. 3.2.37 Delete key 13

Case 4: If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have a minimum number of keys, merge the children. Then merge its sibling with its parent in fig 3.2.38.

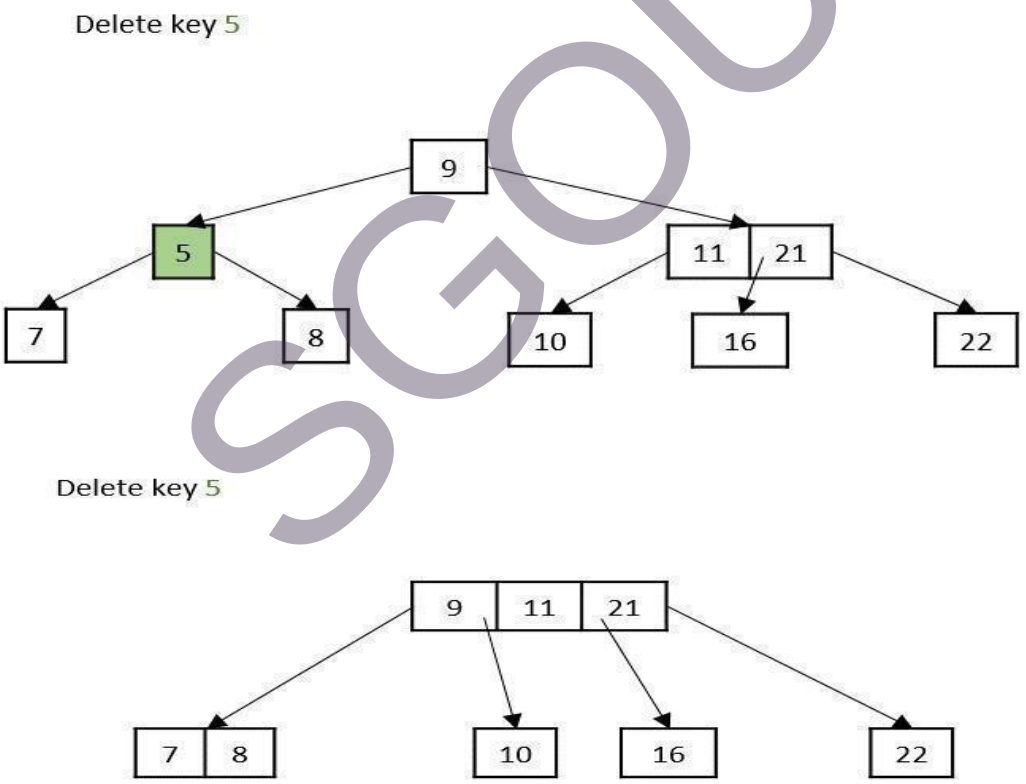


Fig. 3.2.38 Delete key 5

3.2.6.2 Applications of B-Trees

B-Trees are powerful data structures that provide efficient and balanced storage for large volumes of sorted data. Due to their ability to minimize disk access and maintain sorted order, B-Trees are widely used in systems that handle extensive read/write operations. Their structure allows fast search, insertion, and deletion, even with millions of records, making them ideal for use in databases, file systems, and indexing mechanisms. As a result, B-Trees have become a fundamental component in both traditional and modern computing environments where performance and scalability are essential.

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

3.2.6.3 Advantages and Disadvantages of B-Trees

B-Trees are widely used in computer science for managing and organizing large sets of sorted data efficiently. Their balanced structure ensures that operations such as search, insert, and delete are performed in logarithmic time, making them suitable for systems that rely on fast data access and minimal disk reads. However, like any data structure, B-Trees have their own strengths and weaknesses. Understanding the advantages and disadvantages of B-Trees is essential for determining their suitability in various applications such as databases, file systems, and indexing systems.

Some of the advantages of B-Trees are :

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Some of the disadvantages of B-Trees are :

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- For small datasets, the search time in a B-Tree might be slower compared to a binary search tree, as each node may contain multiple keys.

In summary, B-Trees are a smart and efficient way to handle large amounts of data. Their balanced structure and ability to store multiple keys in one node make searching, adding, and deleting data fast and reliable. B-Trees are especially useful for systems like databases and file storage, where quick access to data is important.

3.2.7 B+ Tree

The B+ trees (fig 3.2.39) are extensions of B trees designed to make the insertion, deletion and searching operations more efficient. The properties of B+ trees are similar to the properties of B trees, except that the B trees can store keys and records in all internal nodes and leaf nodes while B+ trees store records in leaf nodes and keys in internal nodes. One profound property of the B+ tree is that all the leaf nodes are connected to each other in a single linked list format and a data pointer is available to point to the data present in the disk file. This helps fetch the records in equal numbers of disk access.

Since the size of main memory is limited, B+ trees act as the data storage for the records that couldn't be stored in the main memory. For this, the internal nodes are stored in the main memory and the leaf nodes are stored in the secondary memory storage.

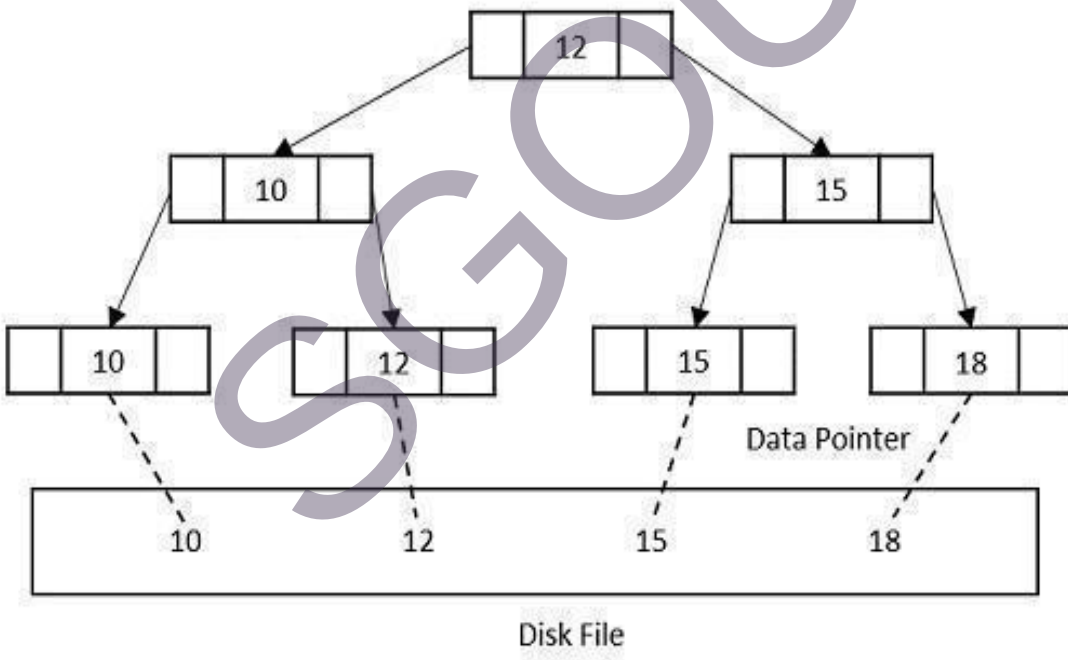


Fig. 3.2.39 Structure of B+ Tree

Properties of B+ trees

- Every node in a B+ Tree, except root, will hold a maximum of m children and (m-1) keys, and a minimum of ⌈m2⌉ children and ⌈m−12⌉ keys, since the order of the tree is m.
- The root node must have no less than two children and at least one search key.

- All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.
- A B+ tree always maintains sorted data.

3.2.7.1 Basic Operations of B+ Trees

The operations supported in B+ trees are Insertion, deletion and searching with the time complexity of O(log n) for every operation. They are almost similar to the B tree operations as the base idea to store data in both data structures is the same. However, the difference occurs as the data is stored only in the leaf nodes of a B+ trees, unlike B trees.

a. Insertion operation

The insertion to a B+ tree starts at a leaf node.

Step 1 : Calculate the maximum and minimum number of keys to be added onto the B+ tree node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Order = 4
Maximum Children (m) = 4
Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2
Maximum Keys (m - 1) = 3
Minimum Keys ($\lceil \frac{m-1}{2} \rceil$) = 1

Step 2 : Insert the elements one by one accordingly into a leaf node until it exceeds the maximum key number in fig 3.2.40.

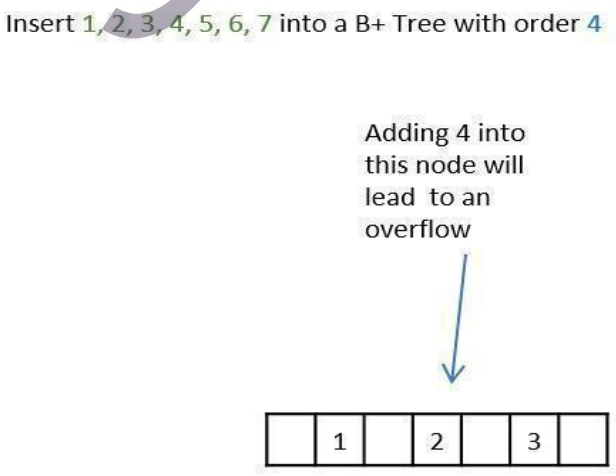


Fig. 3.2.40 Insertion of B+ Tree

Step 3 : The node is split into half where the left child consists of minimum number of keys and the remaining keys are stored in the right child in fig 3.2.41.

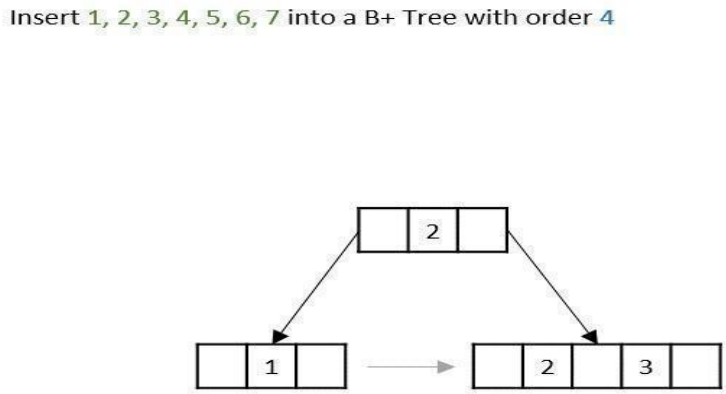


Fig. 3.2.41 Insertion of nodes

Step 4 : But if the internal node also exceeds the maximum key property, the node is split in half where the left child consists of the minimum keys and remaining keys are stored in the right child. However, the smallest number in the right child is made the parent in fig 3.2.42.

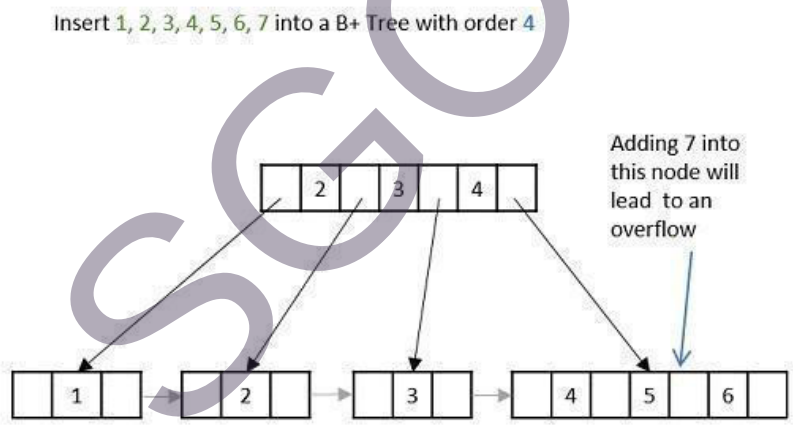


Fig. 3.2.42 Insertion of node 7

Step 5 : If both the leaf node and internal node have the maximum keys, both of them are split in the similar manner and the smallest key in the right child is added to the parent node in fig 3.2.43.

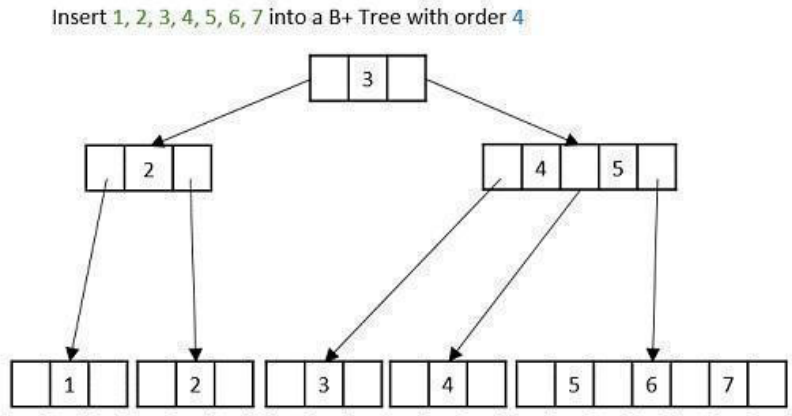


Fig. 3.2.43 Final result

b. Deletion

Deletion in a B+ Tree is the process of removing a key while preserving the balanced structure and search properties of the tree. Since all data entries are stored at the leaf level, deletion always begins from the leaves. After removing a key, the tree must be checked to ensure that every node has at least the minimum number of keys (at least ⌈m/2⌉ keys in a B+ Tree of order m) required. If this condition is violated, adjustments such as borrowing a key from a sibling or merging nodes are performed. These steps ensure that the B+ Tree remains efficient for searching, inserting, and managing data even after deletions.

Steps of Deletion

1. Locate the key in the leaf node.
2. Delete the key from the leaf.
3. Check whether the node still satisfies the minimum occupancy requirement.
4. If no underflow occurs, the process ends.
5. If underflow occurs, check the sibling nodes:
   - **Borrowing :** If a sibling has extra keys, redistribute the keys by borrowing from the sibling and update the parent.
   - **Merging :** If borrowing is not possible, merge the underflowing node with its sibling and adjust the parent node.
6. If merging causes the parent to underflow, repeat the same process upward.
7. If the root has only one child left, make the child the new root, reducing the height of the tree.

Cases of Deletion

Case 1: No Underflow

If the node has enough keys after deletion, no further action is required.

Example 15: Consider fig 3.2.44,

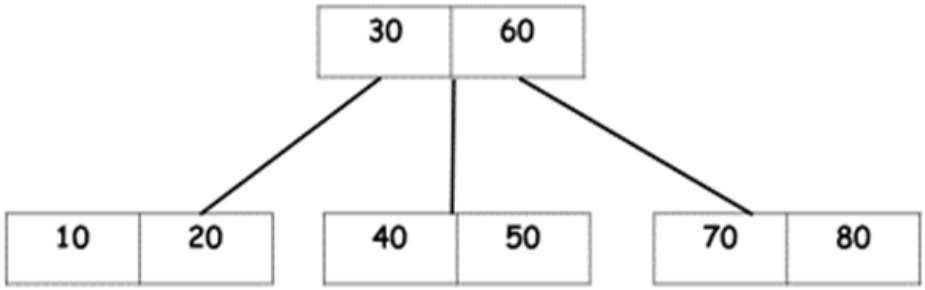


Fig. 3.2.44 No underflow condition

Delete 50 from [40 50] becomes [40], which still satisfies the minimum requirement. Then the result is in fig 3.2.45.

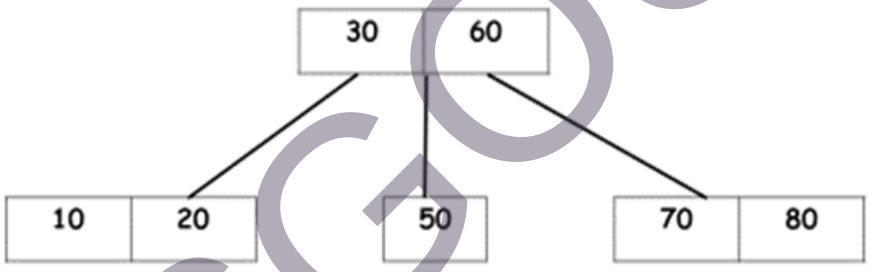


Fig. 3.2.45 Result of no underflow condition

Case 2: Underflow with Borrowing

If a node underflows but its sibling has extra keys, borrowing is performed.

Example 16: Consider fig 3.2.46,

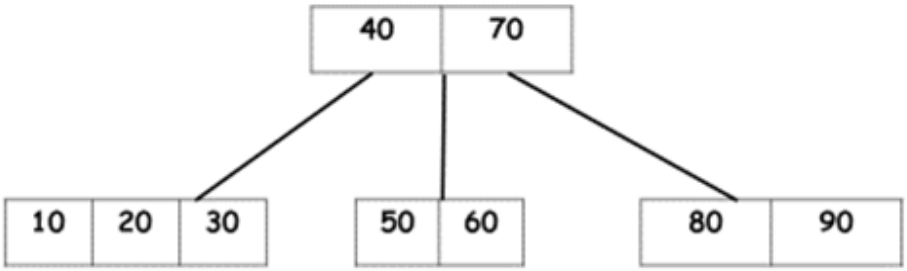


Fig. 3.2.46 Underflow with Borrowing

Delete 30 from [10 20 30] becomes [10 20] (underflow). Borrow key 40 from parent via sibling [50 60]. Then the result is in fig 3.2.47.

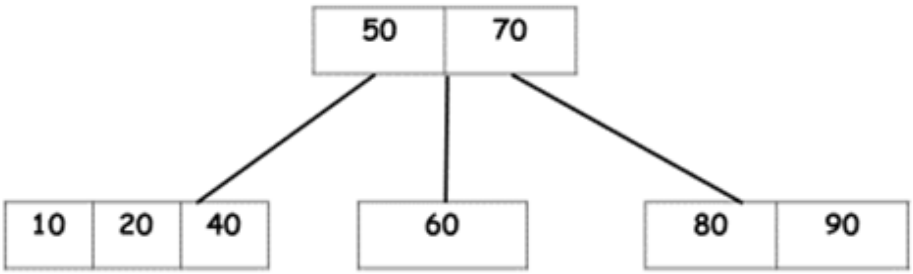


Fig. 3.2.47 Result of underflow with Borrowing

Case 3: Underflow with Merging

If borrowing is not possible, merge with a sibling.

Example 17: Consider the fig 3.2.48,

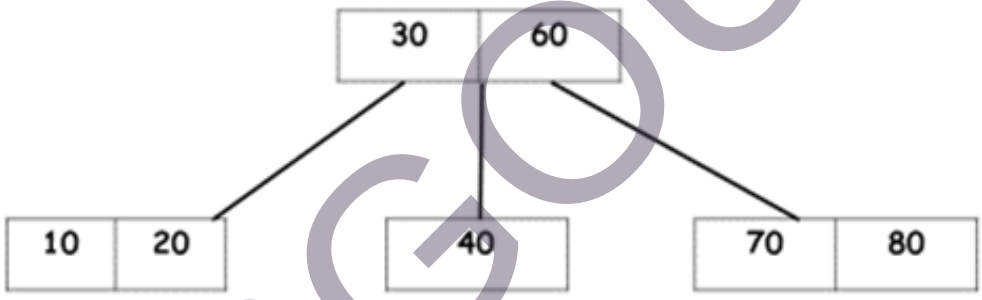


Fig. 3.2.48 Underflow with Merging

Delete 40 from [40] becomes empty (underflow). Merge with left sibling [10 20]. Then the parent updated by removing 30.Then result is in fig 3.2.49.

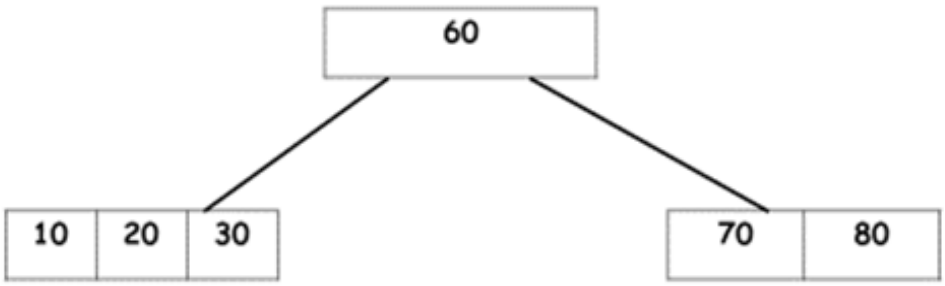


Fig. 3.2.49 Result of underflow with Merging

Case 4: Propagation to Root

If merging reduces the parent keys below minimum, underflow may propagate up.

Example 18 : Consider the fig 3.2.50,

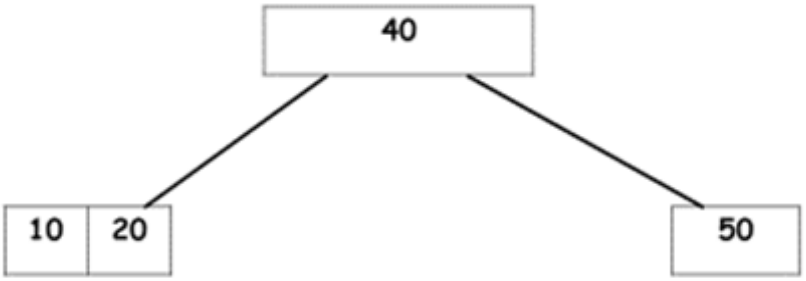


Fig. 3.2.50 Propagation to Root

Delete 50, right child becomes empty. Merge with left child then the parent becomes empty. Root replaced with [10 20]. (Tree height reduced). The result is in fig 3.2.51.

Fig. 3.2.51 Result of Propagation to Root

Deletion in B+ Trees is handled by deleting the key, then checking for underflow. If underflow occurs, borrowing or merging is applied, and the process may propagate upwards until the tree is balanced.

3.2.7.2 Applications of B+ Trees

B+ Trees are widely used in systems that manage large and sorted datasets, thanks to their efficient and scalable structure. By storing all data values at the leaf level and maintaining a linked list of leaves, B+ Trees support fast and smooth traversal for both point queries and range queries. This makes them ideal for use in databases, file systems, and search engines where high-performance data access is required.

1. **Database Indexing :** B+ Trees are extensively used in relational databases (e.g., MySQL, Oracle) to organize and access large datasets efficiently.

2. **File Systems :** Many modern file systems (e.g., NTFS, ReiserFS) use B+ Trees to manage directories and metadata due to their fast sequential and random access.

3. **Data Warehousing and OLAP Systems :** B+ Trees support range-based queries and ordered traversals, making them ideal for analytical workloads.

4. **Search Engines :** Used in inverted indexes for keyword searches where large volumes of data need quick lookup.

5. **Operating Systems :** B+ Trees are used in memory and storage management for indexing and accessing data blocks.

6. **Key-Value Stores :** Many NoSQL databases and distributed storage systems use B+ Trees to handle sorted keys and efficient access.

3.2.7.3 Advantages of B+ Trees

The design of B+ Trees offers several advantages over traditional tree structures. Their shallow height and uniform data access from leaf nodes ensure minimal disk I/O and consistent performance. The linked structure of leaf nodes also enables fast sequential access, making B+ Trees especially effective for range queries and full-table scans. These strengths make B+ Trees a preferred choice in data-intensive applications.

1. **Efficient Range Queries :** All data is stored in linked leaf nodes, allowing fast and easy traversal for range-based and sequential queries.

2. **Better Disk Access :** Internal nodes store only keys (no data), making nodes smaller and reducing the number of disk accesses.

3. **All Values at Leaf Level :** Uniform access time for all records since data is retrieved only from the leaf level.

4. **Linked Leaves :** The leaf nodes are linked as a linked list, which makes full table scans and ordered access very efficient.

5. **Shallower Trees:** Compared to B-Trees, B+ Trees are often shallower, meaning fewer I/O operations are needed.

3.2.7.4 Disadvantages of B+ Trees

Despite their benefits, B+ Trees also have some drawbacks. Because data is only stored at the leaf level, even exact-match queries must traverse to the bottom of the tree, which can slightly increase access time. Additionally, maintaining internal index nodes and the linked list of leaves adds implementation complexity and memory overhead. These trade-offs should be considered when deciding to use B+ Trees in a system.

1. **Slower Point Queries :** Since all data is stored only in the leaves, even exact-match queries must reach the leaf level.

2. **More Storage for Internal Nodes :** Additional space is needed for internal nodes to maintain multiple keys and pointers.

3. **Complex Implementation :** The logic for insertion, deletion, and re-balancing is more complex compared to simpler trees.

4. **Overhead of Leaf Links :** Maintaining the linked structure of leaves introduces additional memory and update overhead.

Balanced binary trees are essential data structures that maintain efficient performance by ensuring their height remains logarithmic with respect to the number of nodes. Among them, AVL trees are height-balanced binary search trees that use rotations to maintain balance during insertions and deletions. B-Trees are multi-way balanced search trees designed for efficient disk storage and are commonly used in databases and file systems. B+ Trees, an enhancement of B-Trees, store data only in leaf nodes and maintain a linked list for fast sequential access, making them ideal for range queries. Each of these trees offers specific advantages depending on the use case, and their properties and operations such as searching, insertion, and deletion are optimized to support fast and reliable data management in large-scale applications.

Summarised Overview

This unit deals with tree-based data structures that play a crucial role in efficient data storage, retrieval, and manipulation. It begins with Binary Search Trees (BSTs), where students learn how insertion and deletion operations are performed while maintaining the sorted property of the tree. The discussion highlights how unbalanced trees can affect performance, leading to slower search operations. To overcome this drawback, the unit introduces Balanced Binary Search Trees, focusing on the AVL Tree, which uses rotations to maintain height balance and ensure logarithmic time complexity for operations. The unit then progresses to M-way Search Trees, which allow multiple branches at each node and are suitable for handling large datasets. Among these, B Trees and B+ Trees are studied in detail, as they form the backbone of indexing in databases and file systems. Learners explore their insertion, deletion, and search operations, along with their applications in real-world scenarios where quick access to large volumes of data is required. By the end of this unit, students gain a strong understanding of how different types of search trees are designed, balanced, and applied to achieve efficient performance in computing systems.

Assignments

1. Explain the process of insertion and deletion in a Binary Search Tree with suitable examples and diagrams. Discuss how the height of a BST affects search efficiency.

2. Describe how an AVL Tree maintains balance using rotations. Illustrate the different types of rotations (LL, RR, LR, RL) with examples of insertion that trigger each case.

3. Compare Binary Search Trees and AVL Trees in terms of structure, performance, and applications. Provide scenarios where AVL Trees are preferred over BSTs.

4. Explain the structure and properties of M-way Search Trees. Using an example, demonstrate the insertion process in a B-Tree of order 3.

5. Discuss the similarities and differences between B Trees and B+ Trees. Explain the deletion process in a B+ Tree with the help of an example and diagram.

Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.

2. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

3. Johnson, R. (2025). B-Tree Algorithms and Applications: Definitive Reference for Developers and Engineers. (E-book).

4. Weiss, M. A. (2022). Data structures & algorithm analysis in C (4th ed.). Pearson.

Suggested Reading

1. GeeksforGeeks — Tree Data Structures https://www.geeksforgeeks.org/data-structures/binary-search-tree/

2. Programiz – Data Structures Tutorial https://www.programiz.com/dsa

3. W3Schools – DSA Binary Trees https://www.w3schools.com/dsa/dsa_data_binarytrees.php

4. Simplilearn – B Tree in Data Structure https://www.simplilearn.com/tutorials/data-structure-tutorial/b-tree-in-data-structure

5. Wikipedia – B+ Tree and AVL Tree https://en.wikipedia.org/wiki/AVL_tree

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3
UNIT

Spanning Tree

Learning Outcomes

After completion of this unit, the learner will be able to:

- understand the concepts of spanning trees and minimum spanning trees (MST)
- identify real-world situations where spanning trees and shortest path algorithms optimize networks and resources
- describe the working of Kruskal's and Prim's algorithms
- apply Kruskal's and Prim's algorithms to solve graph problems
- analyze and compare algorithms based on efficiency, graph type, and applications

Background

Do you understand the concept of a graph? Do you know what a graph represents? Are you familiar with basic data structures like arrays, linked lists, or priority queues? If yes, you are ready to explore how these ideas can solve real-world problems. Graphs are used to model networks, such as cities connected by roads, computers in a network, or even social connections. Programming languages help us implement algorithms to process these graphs efficiently, while data structures allow us to store and manage the data effectively. This chapter introduces Minimum Spanning Tree (MST) and shortest path algorithms, which help find the most efficient paths or connections in a network. For instance, if a company wants to connect several offices using the least amount of cable, MST algorithms like Kruskal's or Prim's can determine the cheapest way to connect all locations. By studying this chapter, you will learn to apply programming and data structures to solve practical optimization problems, save resources, and improve decision-making in computer networks, transportation, and other real-life systems.

Keywords

Directed Graph, Spanning Tree, Kruskal's Algorithm, Prim's Algorithm, Shortest Path, Optimization,Network Routing, Path, Cycle.

Discussion

3.3.1 Spanning Tree

A spanning tree is a special subgraph of a connected, undirected graph G that includes all the vertices of G while using the minimum number of edges possible to keep them connected. In a graph with n vertices, a spanning tree will always have exactly n−1 edges, because this is the smallest number of edges needed to connect all vertices without forming any cycles. If G is disconnected, each connected component has a spanning tree and together they form a spanning forest. It is acyclic (it contains no loops) and connected (there is a path between any two vertices). The concept is important in network design, as it ensures the creation of a minimal connection structure without redundancy, saving resources like cable length or communication channels. In real-life applications, spanning trees are used in network routing, electrical grid design, and clustering in machine learning to ensure minimal and efficient connectivity.

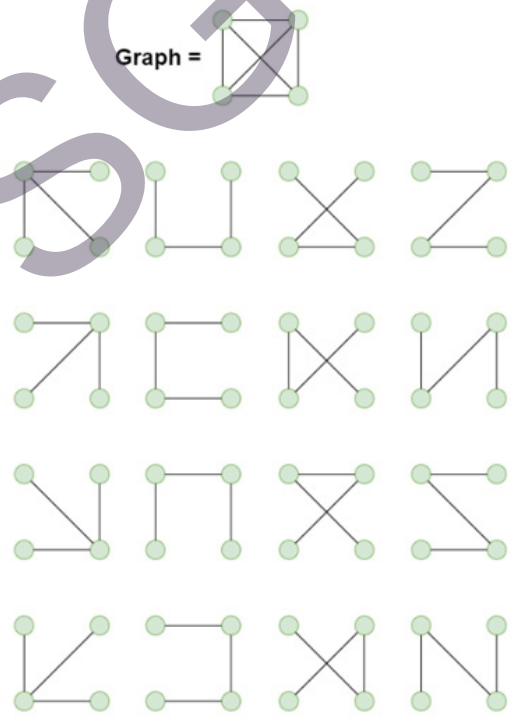


Fig. 3.3.1 All possible Spanning Trees of the Graph

Properties of a Spanning Tree

1. A spanning tree does not exist for a **disconnected graph**.
2. For a **connected graph** having N vertices, the number of edges in its spanning tree will be $N - 1$.
3. A spanning tree **does not contain any cycle**.
4. For a **complete graph**, we can construct a spanning tree by removing $E - N + 1$ edges, where:
   - E = number of edges
   - N = number of vertices
5. **Cayley's Formula**:
   The number of spanning trees possible in a complete graph with N vertices is:

$$N^{N-2}$$

Example: If $N = 4$,

$$\text{Number of spanning trees} = 4^{4-2} = 4^2 = 16$$

Fig. 3.3.2 Properties of a Spanning Trees

3.3.2 Minimum Cost Spanning Tree

A minimum cost spanning tree (MCST), also known as a minimum spanning tree (MST), is a subset of a graph's edges that connects all the vertices together, without any cycles and with the minimum possible total edge weight. In simpler terms, it's the cheapest way to connect all the nodes in a network. The "cost" of the tree is the sum of the weights of all its edges. An MCST is used in various applications, such as designing networks for telecommunications, transportation, and computer systems, where the goal is to connect all locations with the lowest possible cost.

Properties

- **Spanning :** The tree must include all vertices of the original graph.
- **Tree :** It must be acyclic (contain no cycles) and connected.
- **Minimum Cost :** The sum of the weights of the edges in the tree is as small as possible.
- **Unique or Multiple :** A graph can have a unique MCST, or multiple MCSTs if there are edges with the same weight.

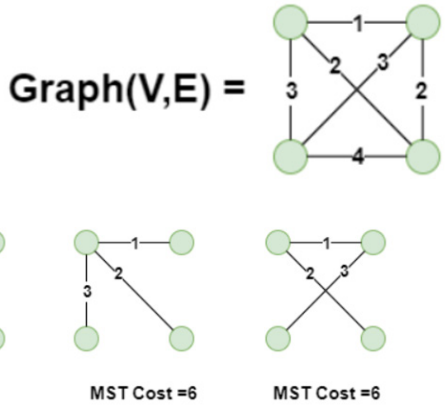


Fig. 3.3.3 All possible MST's of the graph

3.3.3 Minimum Spanning Tree Algorithms

A Minimum Spanning Tree (MST) is a spanning tree of a connected, weighted, undirected graph that has the smallest possible total edge weight among all possible spanning trees. Several algorithms exist to find an MST efficiently. Two of the most popular algorithms are Kruskal's Algorithm and Prim's Algorithm. Both aim to achieve the same goal but follow different approaches.

3.3.3.1 Kruskal's Algorithm

It is a greedy algorithm used to find a minimum spanning tree (MST) for a weighted, connected, and undirected graph. Kruskal works by sorting all the edges in non-decreasing order of their weights. It then iterates through this sorted list, adding an edge to the MST if and only if it does not form a cycle with the edges already chosen. This process continues until a total of V−1 edges (where V is the number of vertices) have been selected, ensuring that all vertices are connected with the minimum possible total weight. The use of a disjoint set data structure is key to efficiently checking for cycles and makes the algorithm highly effective.

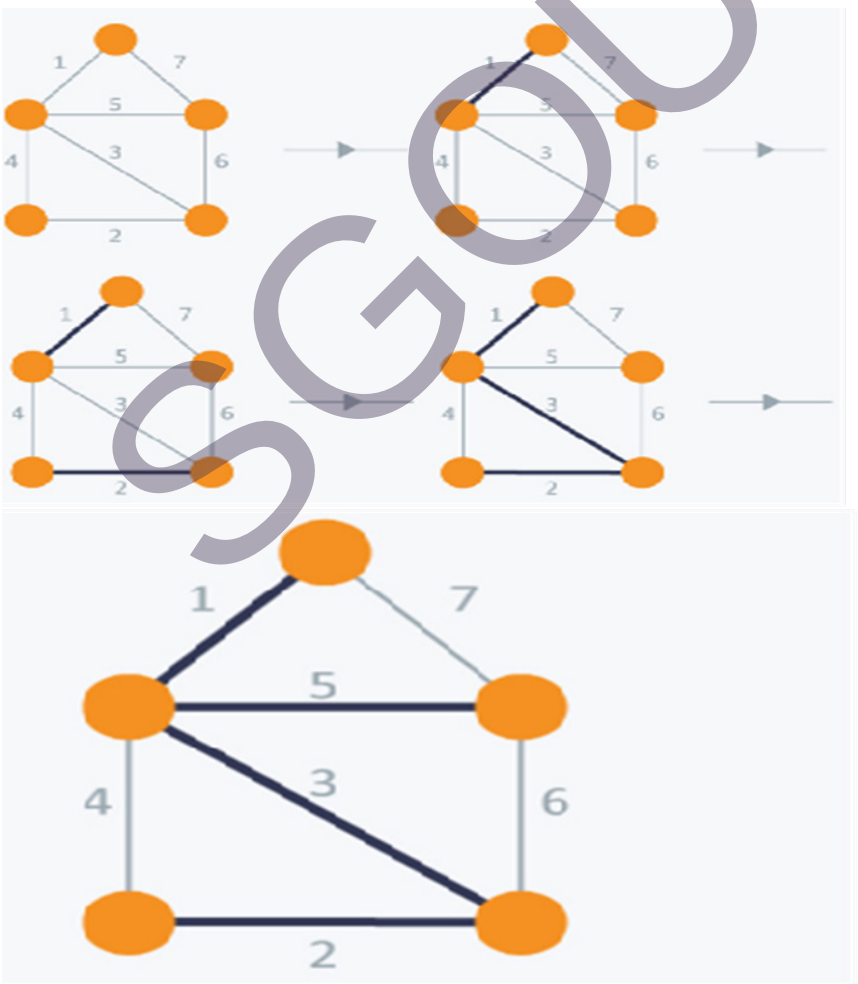


Fig. 3.3.4 Spanning tree

In Kruskal's algorithm, always pick the edge with the smallest weight first.

1. **Step 1 :** Start with the edge of weight 1.
2. **Step 2 :** Pick the next smallest edge, weight 2. These two edges are disjoint so far.
3. **Step 3 :** Select the next smallest edge, weight 3, which connects the two disjoint components.
4. **Step 4 :** Skip the edge with weight 4, as it would create a cycle.
5. **Step 5 :** Choose the edge with weight 5 instead.
6. **Step 6 :** Ignore the remaining edges (weights 6 and 7) since they would also form cycles.

Result: The Minimum Spanning Tree (MST) has a total cost of 11 (1 + 2 + 3 + 5).

3.3.3.2 Prim'S Algorithm

Prim's Algorithm is a Greedy Algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted, undirected graph. Like Kruskal's Algorithm, its goal is to connect all vertices together using the smallest possible total edge weight without creating any cycles

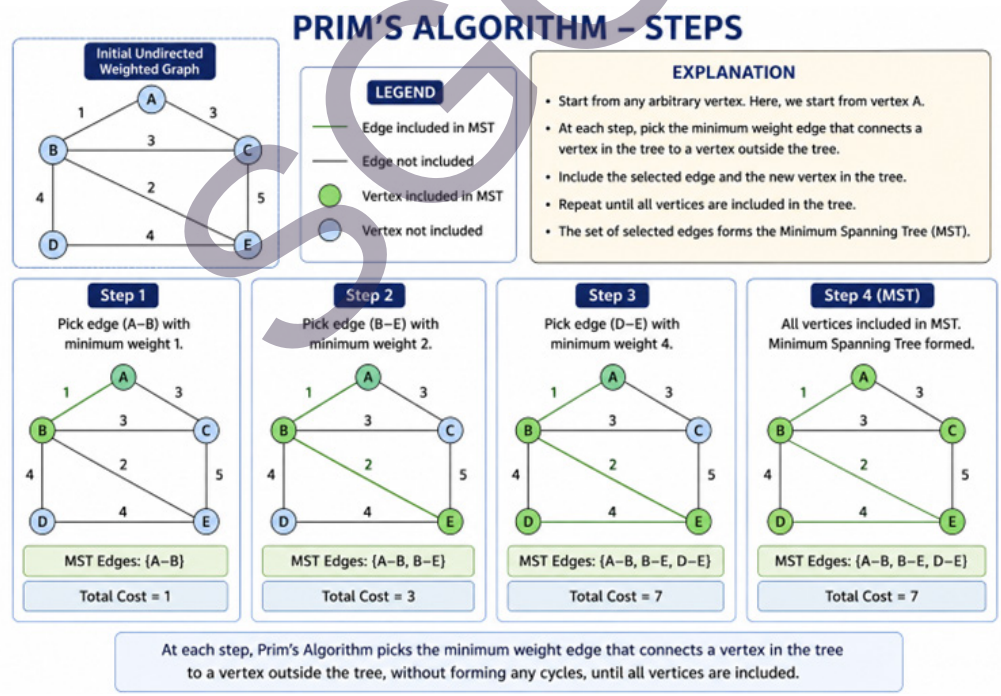


Fig. 3.3.5 Spanning tree

In Prim's Algorithm, the process begins from any arbitrary node, which is then marked. At each step, a new vertex adjacent to the marked set is chosen using the cheapest available edge.

1. **Step 1:** Start with the edge of weight 1 and mark its connected vertex.

2. **Step 2:** Among the available edges (weights 2, 3, and 4), select the edge with weight 2 and mark its vertex.

3. **Step 3:** Now, the available edges have weights 3, 4, and 5. Since the edge with weight 3 would form a cycle, choose the edge with weight 4 instead.

Result : The Minimum Spanning Tree (MST) is obtained with a total cost of 7 (1 + 2 + 4).

3.3.3.3 Comparison of Kruskal's and Prim's Algoritham

Table 3.3.1 Comparison between Kruskal's and Prim's Algorithm

Kruskal's Algorithm	**Prim's Algorithm**
Works by sorting all edges in the graph in ascending order of their weights and adding them one by one to the MST, skipping those that create a cycle.	Works by starting from an arbitrary vertex and adding the smallest edge that connects a visited vertex to an unvisited vertex.
Edge-based approach — focuses on choosing the smallest edges first.	Vertex-based approach — focuses on expanding the MST from one vertex outward.
Uses Disjoint Set Union (Union-Find) to check and avoid cycles.	Avoids cycles naturally by marking visited vertices and only connecting unvisited ones.
Best suited for sparse graphs (fewer edges).	Best suited for dense graphs (many edges).
Starts with the smallest edge in the graph.	Starts with an arbitrary vertex in the graph.
Works well with edge list representation.	Works well with adjacency matrix or adjacency list representation.
Time complexity: O(E log E) due to edge sorting (same as O(E log V) for connected graphs).	Time complexity: O(E log V) using min-heap and adjacency list.
Commonly used in clustering problems and when the edge list is already available or sorted.	Commonly used in network design where you expand from a starting point.

3.3.4 Applications of a Spanning Tree

1. Path Finding Algorithms

Several well-known path finding algorithms, such as Dijkstra's Algorithm and A* Search Algorithm, make use of spanning trees as an intermediate step in their execution. In these algorithms, the spanning tree helps to ensure that all vertices are connected while avoiding unnecessary cycles, thus reducing complexity. For example, when determining the shortest route in a map navigation system, the algorithm internally builds a spanning tree to explore paths efficiently. This structure ensures that every location is reachable in the least costly manner without redundant loops, which greatly improves performance and accuracy.

2. Building Telecommunication Networks

In designing large-scale telecommunication networks such as telephone lines, broadband connections, or optical fiber layouts, spanning trees are used to minimize the cost of connecting all stations. Each network node represents a location or hub, and the goal is to connect them with the least number of cables while maintaining full connectivity. By using a spanning tree, network engineers can ensure that every node is connected with minimal total cable length, which saves cost and simplifies maintenance. Additionally, the absence of cycles reduces the risk of signal interference and data transmission conflicts.

3. Image Segmentation

In computer vision, spanning trees are used for image segmentation, a process of dividing an image into distinct, meaningful regions. The image is represented as a graph, where each pixel or group of pixels is a vertex, and the edges represent the similarity between them. A spanning tree can be constructed to group similar regions while avoiding redundant links, which allows efficient separation of different objects in the image. This method is especially useful in medical imaging, satellite imagery, and facial recognition systems, where accurate and efficient segmentation is critical.

4. Computer Network Routing Protocols

Spanning trees play a crucial role in computer network routing protocols, such as the Spanning Tree Protocol (STP) used in Ethernet networks. STP ensures that there are no loops in the network topology, which could otherwise lead to broadcast storms and network congestion. By creating a loop-free logical topology from a physically redundant network, STP uses the concept of a spanning tree to determine the best path for data packets. This not only ensures efficient routing but also provides fault tolerance—if one link fails, the protocol can recalculate the tree and reroute traffic through alternative paths without disrupting connectivity.

5. Power Distribution Networks

In designing electrical power distribution systems, spanning trees are used to create an optimal wiring plan that connects all substations and distribution points without forming cycles. This ensures minimal wiring cost, reduces transmission losses, and simplifies fault detection. The absence of loops in a spanning tree makes it easier to

isolate faults and restore services quickly in case of power outages.

3.3.4 Shortest Path Algorithms

In graph theory, the shortest path problem is about finding a path between two vertices such that the sum of the edge weights is minimized. The goal of shortest path algorithms is to compute the minimum cost path from one node to another. The shortest path problem can be classified into different types depending on the number of sources and destinations. The most common type is the Single Source Shortest Path (SSSP) problem, where the objective is to determine the minimum cost or distance from a single starting vertex to all other vertices in the graph. This type is very practical in scenarios such as finding the shortest driving routes from a particular city to multiple cities, and is efficiently solved by algorithms like Dijkstra's Algorithm (for graphs with non-negative weights) and Bellman–Ford Algorithm (for graphs with negative weights). The second type is the Single Destination Shortest Path (SDSP) problem, in which the goal is to compute the shortest paths from every vertex in the graph to a specific destination vertex. This is useful, for instance, when many delivery trucks need to reach a central warehouse from different locations. Finally, the All Pairs Shortest Path (APSP) problem aims to find the shortest paths between every pair of vertices in the graph. This type is essential for network routing, airline scheduling, and distance matrix computations, where information about the minimum cost between all nodes is required. A widely used algorithm for this category is the Floyd–Warshall Algorithm, which applies dynamic programming to compute shortest paths between all pairs efficiently. Together, these three problem types provide a foundation for solving real-world optimization problems in navigation, networking, logistics etc.

Dijkstra's algorithm is a greedy algorithm used to find the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It works by maintaining a set of vertices whose shortest distance from the source is already known and repeatedly selecting the unvisited vertex with the smallest tentative distance. From this vertex, it updates the distances of its neighboring vertices if a shorter path is found. This process continues until all vertices have been visited, ensuring that the shortest path to every node is determined. For example, consider a graph with edges A→B(4), A→C(5), B→D(2), C→D(3). Starting from A, the algorithm sets the initial distances as A=0, B=∞, C=∞, D=∞. After visiting A, the distances update to B=4, C=5, D=∞. Next, B is selected (since 4 is the smallest), and relaxing B→D gives D=6. Then C is processed, and relaxing C→D updates D to 5 (since 5 < 6). Finally, the shortest paths are obtained as A=0, B=4, C=5, and D=5. This shows how Dijkstra efficiently computes the shortest routes, making it highly useful in applications such as GPS navigation, traffic routing, and communication networks.

The Bellman–Ford Algorithm is a shortest path algorithm that finds the minimum distance from a single source vertex to all other vertices in a weighted graph, even when edges have negative weights. Unlike Dijkstra's algorithm, which fails with negative values, Bellman–Ford works by repeatedly applying the relaxation technique: it iteratively updates the distance to each vertex through all edges, and after at most V–1 iterations (where V is the number of vertices), it guarantees the shortest paths. An extra iteration is used to check for negative weight cycles—if distances still decrease,

the graph contains such a cycle. For example, in a graph with edges A→B(4), A→C(5), B→D(–2), and C→D(3), starting from A, the shortest distances found are A=0, B=4, C=5, and D=2. Though its time complexity is higher (O(V·E)), making it slower than Dijkstra, it is more versatile and widely used in finance (currency arbitrage detection), network routing, and transportation problems where negative weights may occur.

The Floyd–Warshall Algorithm is a dynamic programming approach used to find the shortest paths between all pairs of vertices in a weighted graph, handling both positive and negative edge weights (but not negative weight cycles). Unlike Dijkstra or Bellman–Ford, which focus on a single source, Floyd–Warshall systematically updates a distance matrix by considering whether including an intermediate vertex can provide a shorter path between two vertices. For example, in the graph with edges A→B(4), A→C(5), B→D(2), C→D(3), we first initialize the distance matrix with direct edge weights: A→B=4, A→C=5, B→D=2, C→D=3, and all other pairs as ∞ (except self-distances=0). Then, the algorithm iteratively checks each vertex as an intermediate point: when considering B, we find a shorter path A→D via B with cost 6 (A→B→D). When considering C, we update A→D again to 5 (A→C→D), which is the true shortest distance. At the end, the shortest path results are A=0, B=4, C=5, and D=5, consistent with Dijkstra's output but now extended for all pairs of vertices simultaneously. Due to its simplicity and systematic nature, Floyd–Warshall is widely applied in network routing, airline scheduling, and distance matrix computation.

Summarised Overview

A spanning tree is a subgraph of a connected, undirected graph that connects all vertices with exactly n−1 edges, ensuring connectivity without cycles; if the graph is disconnected, spanning forests are formed. A minimum cost spanning tree (MST) is a spanning tree with the smallest total edge weight, useful in network design, routing, and resource optimization. Two main algorithms to find MSTs are Kruskal's Algorithm (edge-based, sorts edges and adds the smallest non-cyclic ones; efficient for sparse graphs) and Prim's Algorithm (vertex-based, expands outward from a chosen node using the cheapest connecting edge; efficient for dense graphs).Shortest path algorithms focus on minimizing path costs: Dijkstra's Algorithm (greedy, works with non-negative weights), Bellman–Ford Algorithm (handles negative weights, detects negative cycles), and Floyd–Warshall Algorithm (dynamic programming, finds shortest paths between all vertex pairs). Together, these algorithms are widely applied in telecommunication networks, transportation systems, image segmentation, computer network protocols (like STP), and power distribution, as they ensure efficient connectivity, cost minimization, and reliability.

Assignments

1. Apply Kruskal's Algorithm to the following graph and find its MST. Show each step clearly. Vertices = {A, B, C, D, E} Edges = { (A–B, 1), (A–C, 3), (B–C, 2), (B–D, 4), (C–E, 5), (D–E, 7) } Apply Prim's Algorithm to the same graph (start from vertex A) and find the MST. Compare the steps with Kruskal's Algorithm.

2. Compare Kruskal's Algorithm, and Prim's Algorithm.

3. Discuss about shortest path algorithms.

4. Explain the applications of a Spanning Tree in real-world scenarios?

Reference

1. Kumar, A., & Yadav, J. (2024). Minimum spanning tree clustering approach for effective feature partitioning in multi-view ensemble learning. *Knowledge and Information Systems, 66*(3), 6785–6813.

2. Zhao, Y., & Zhang, L. (2023). Innovative method to solve the minimum spanning tree problem: The LC-MST method. *Computers, Materials & Continua, 74*(3), 2811–2825.

3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

4. Kleinberg, J., & Tardos, É. (2006). *Algorithm design*. Pearson Education.

5. Chao, K. M. (2003). *Spanning trees and optimization problems*. Springer

Suggested Reading

1. https://www.geeksforgeeks.org/dsa/spanning-tree
2. https://doi.org/10.1007/s10107-023-01791-6
3. *Minimum Spanning Tree vs Shortest Path Tree*. https://www.baeldung.com/cs/minimum-spanning-vs-shortest-path-trees
4. https://doi.org/10.32604/cmc.2023.023929

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4
UNIT

Graphs

Learning Outcomes

After completion of this unit, the learner will be able to:

- familiarize with the fundamental terminology and concepts related to graphs
- understand different methods of representing graphs in memory and their characteristics
- explore the working principles and algorithms of Depth First Search (DFS) and Breadth First Search (BFS)
- compare DFS and BFS in terms of approach, efficiency, and use cases
- recognize the practical applications of graphs in solving real-world problems

Background

Graphs are essential in data structures because they let us represent relationships in a way that other structures like arrays or linked lists cannot. Many real-world problems are not just about storing data but about showing how different pieces of information connect to each other.

Think of social media networks—users are not isolated; they're connected through friendships, followers, and interactions. Road maps work the same way: cities (nodes) are linked by roads (edges). Even complex systems like airline routes, recommendation engines, or dependency charts in project planning rely on these interconnected relationships.

Without graphs, representing such connections would require messy, repetitive, and inefficient data models. Graphs offer a clean, logical, and efficient way to handle these scenarios, enabling algorithms to quickly find shortest paths, detect cycles, or analyze connectivity.

They turn scattered information into a web of meaning—helping computers "understand" not just the data, but how that data is related.

Keywords

Weighted graph, self loop, degree, incidence, walk, path, articulation point

Discussion

3.4.1 Definition of Graph

A graph G is defined as a non-empty finite set of vertices V(G) and a finite set of edges E(G). The vertex set is represented as V(G) = (v0, v1, …, vn), also called the set of vertices or nodes, and the edge set is represented as E(G) = (e1, e2, …, en), known as the set of edges. Each edge connects a pair of vertices. If e = (vi, vj) is an edge where vi and vj belong to V(G), then vi and vj are said to lie on the edge e, and e is said to be incident with both vi and vj.

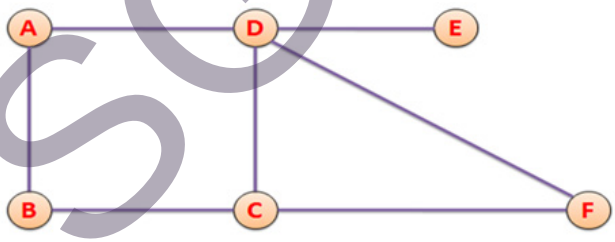


Fig. 3.4.1 Example of simple Graph.

Consider a simple graph G with vertices A, B, C, D, E, and F as shown in Figure 3.4.1. We can see that there are 6 vertices or nodes and 7 edges. That is,

V (G) = { A, B, C, D, E, F }

E (G) = { (A, B), (A, D), (B, C), (D, C), (C, F), (D, E), (D, F) }

3.4.2 Types of Graphs

Different types of graphs are used for different purposes, depending on the nature of the data and the relationships they represent. Understanding these types is essential for designing efficient algorithms and solving complex problems in various domains.

Broadly, graphs can be classified into two main categories: Undirected Graphs and Directed Graphs.

Undirected Graph

In an undirected graph, the pair of vertices connected by an edge is unordered. This means that if there is an edge between vertices v1 and v2, it can be written as either (v1, v2) or (v2, v1).

Example : Consider a graph with 5 vertices and 5 edges:

V(G) = {A, B, C, D, E} E(G) = {(A, B), (A, C), (A, D), (D, C), (C, E)}

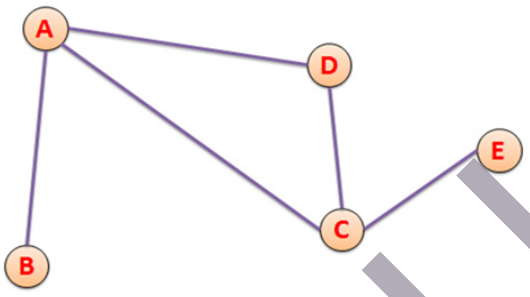


Fig. 3.4.2 Example of Undirected Graph

Here, the edge (A, B) can also be written as (B, A), and the edge (A, C) can be written as (C, A). Similarly, the remaining edges can be reversed in the same way. Since the order of vertices in each edge does not matter, this is called an **undirected graph**.

Directed Graph

In a directed graph (also called a digraph), the pair of vertices connected by an edge is ordered. This means that an edge is represented as (v1, v2), where v1 is the *tail* and v2 is the *head* of the edge.

Directed edges are shown using arrowheads to indicate direction.

Example : Consider a directed graph with 5 vertices and 6 edges:

V(G) = {A, B, C, D, E}
E(G) = {(A, D), (D, E), (E, C), (C, D), (B, C), (B, A)}

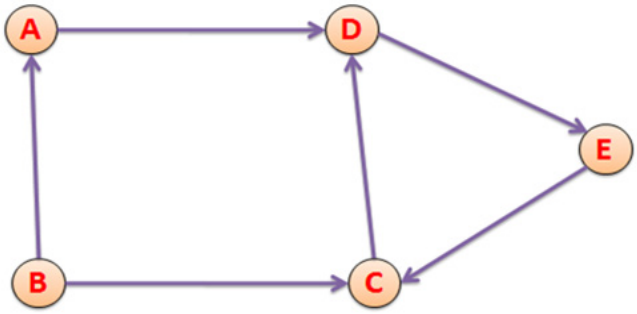


Fig. 3.4.3 Example of directed Graph

In this example, the edge (A, D) has its tail at A and head at D, and cannot be written as (D, A) because that would represent a different edge. The same rule applies to all other edges in the graph.

3.4.3 Terminologies used in Graphs

Graphs are widely used to represent complex relationships between objects, and understanding their structure requires familiarity with specific terminologies. These terms help describe the components and properties of a graph, enabling clear communication and effective problem-solving. Key graph terminologies include vertices (or nodes), edges (or arcs), degree, adjacency, path, cycle, and connectivity, among others. Each of these terms plays a vital role in defining the behavior and characteristics of different types of graphs. Gaining a strong grasp of graph terminology is fundamental for studying graph theory and applying it in areas such as computer networks, data structures, social networks, and transportation systems.

3.4.3.1 Weighted Graph

Consider a city route map which is represented by a graph. The junctions are represented using nodes and the roads connecting them are represented using edges. Take an edge connecting two nodes (vertices). We can label this edge with a value that corresponds to the distance between those nodes. We can say it as the weight of that edge. Similarly the speed limit in a road (edge in the graph) can be represented as weight.

If all the edges in a graph are labeled with some numbers or weights, then the graph is called a weighted graph. Figure shows an example for a weighted graph. Here P, Q, R, and S are the vertices of the graph. The graph consists of 4 edges and all are labeled with some weights. Edge PQ is labeled with a weight 4. Edge PR is labeled with a weight 2. Edge RS is labeled with a weight 5 and edge QS is labeled with a weight 3 in figure .

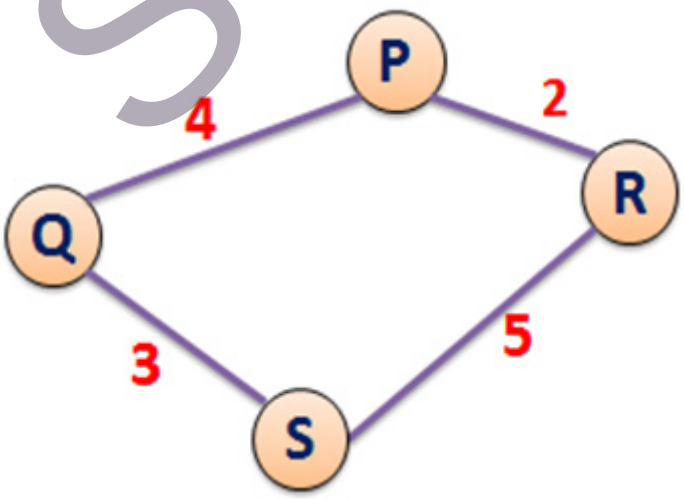


Fig. 3.4.4 An example for Weighted Graph

3.4.3.3 Parallel Edges

For a graph G, if there are more than one edges between the same pair of vertices then they are known as parallel edges. Figure 3.4.5 shows a graph with parallel edges. We can see there are two edges between the same pair of vertices v1 and v2.

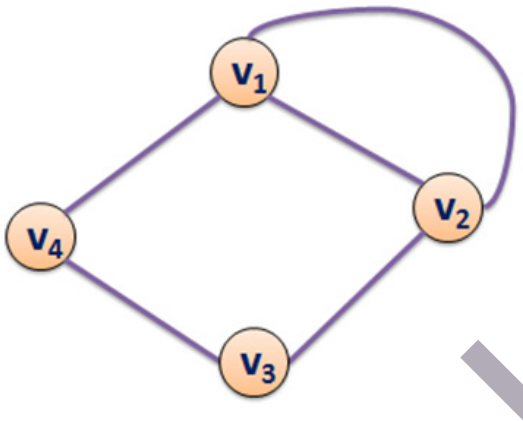


Fig. 3.4.5 An example for Parallel Edges

3.4.3.2 Self Loop

For a graph G, if there is an edge whose starting and ending vertices are the same, that is (v1, v1) then it is called a self loop. Figure 3.4.6 shows an example of a graph with a self loop. Here v1, v2, v3, and v4 are the vertices of the graph. The edges are (v1, v2), (v2, v3), (v3, v4), (v4, v1) and (v1, v1) and the edge (v1, v1) is a self loop.

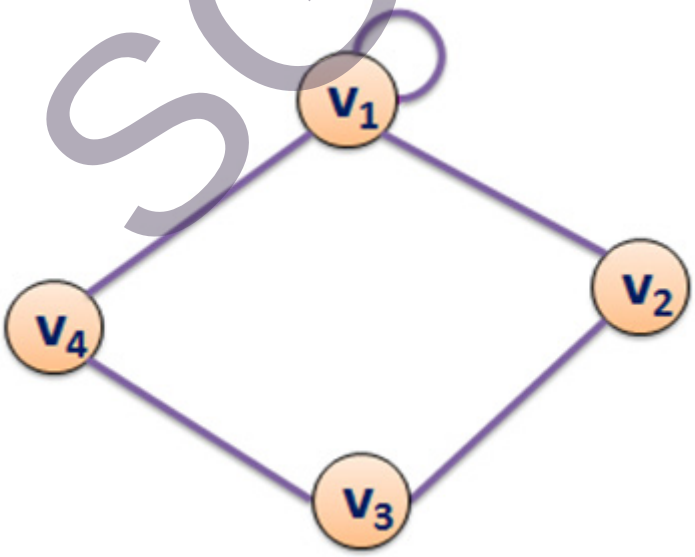


Fig. 3.4.6 An example for Self Loop

3.4.3.3 Adjacent Vertices

In a graph G, a vertex u is adjacent to another vertex v if there is an edge from u to v. In an undirected graph if (v1, v2) is an edge then v1 is adjacent to v2 and v2 is adjacent to v1. In a directed graph if (v1, v2) is an edge then v1 is adjacent to v2 and v2 is adjacent from v1. For example, Figure 3.4.7 (i) shows an undirected graph, vertex u is adjacent to vertex v and vertex v is adjacent to vertex u. Figure 3.4.7 (ii) shows a directed graph, vertex u is adjacent to vertex v and vertex v is adjacent from vertex u.

Fig. 3.4.7 An example for adjacent vertices and Incidence

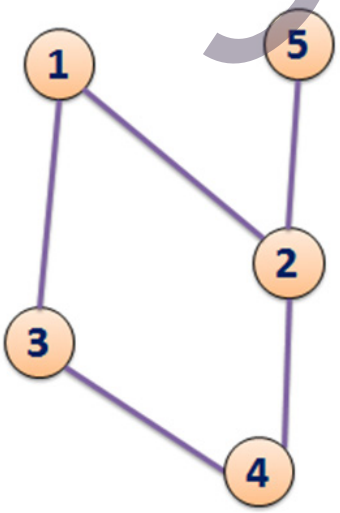
3.4.3.4 Incidence

In an undirected graph the edge (u, v) is incident on vertices u and v. In a directed graph the edge (u, v) is incident from node u and is incident to node v.

For example, Figure 3.4.7(i) shows an undirected graph; the edge (u, v) is incident on vertices u and v. Figure 3.4.7(ii) shows a directed graph; edge (u, v) is incident from node u and is incident to node v.

3.4.3.5 Degree of Vertex

The degree of a vertex is the number of edges incident to that vertex. The degree of a node in an undirected graph is the number of edges connected to that node.



Vertex	Degree of Vertex
1	2
2	3
3	2
4	2
5	1

Fig. 3.4.8 Degree of Vertex in an Undirected Graph

Take an example graph shown in Figure 3.4.8, to find the degree of a vertex. Here the undirected graph contains five vertices. We can find the degree of each vertex in Figure 3.4.8. The degree of vertex 1 is 2 because 2 edges are incident to vertex 1. The degree of vertex 2 is 3 because 3 edges are incident to vertex 2 and so on.

In a directed graph there are two types of degrees for every node; in degree and out degree.

In degree: The in degree of a vertex is the number of edges coming to that vertex or edges incident to it.

Out degree: The out degree of a vertex is the number of edges going outside from that node or the edges incident from it.

Take an example digraph shown in Fig.3.4.9, to find the degree of a vertex. Here the directed graph contains five vertices. We can find the in degree and out degree of each vertex in Figure 3.4.9.

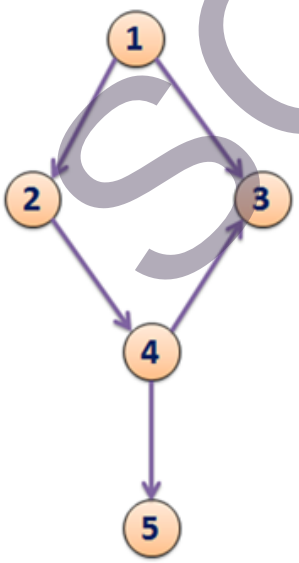
The **in degree** of vertex 1 is **0** because no edges are incident to vertex 1.

The **in degree** of vertex 2 is **1** because only 1 edge is incident to the vertex 2.

The **in degree** of vertex 3 is **2** because 2 edges are incident to vertex 3.

The **in degree** of vertex 4 is **1** because only 1 edge is incident to the vertex 4.

The **in degree** of vertex 5 is **1** because only 1 edge is incident to the vertex 5.



Vertex	In degree	Out Degree
1	0	2
2	1	1
3	2	0
4	1	1
5	1	0

Fig. 3.4.9 Degree of Vertex in a Directed Graph

The **out degree** of vertex 1 is **2** because 2 edges are incident from the vertex 1.

The **out degree** of vertex 2 is **1** because only 1 edge is going outside from the vertex 2.

The **out degree** of vertex 3 is **0** because no edges are going outside from the vertex 3.

The **out degree** of vertex 4 is **1** because only 1 edge is incident from the vertex 4.

The **out degree** of vertex 5 is **0** because no edges are going outside from the vertex 5.

3.4.3.6 Simple graph

A graph or digraph which does not have only self loop or parallel edges is called a simple graph. Figure 3.4.10 shows examples for simple graph. (i) shows a simple undirected graph. There are no self loops and parallel edges. (ii) shows a simple digraph with no self loops and parallel edges.

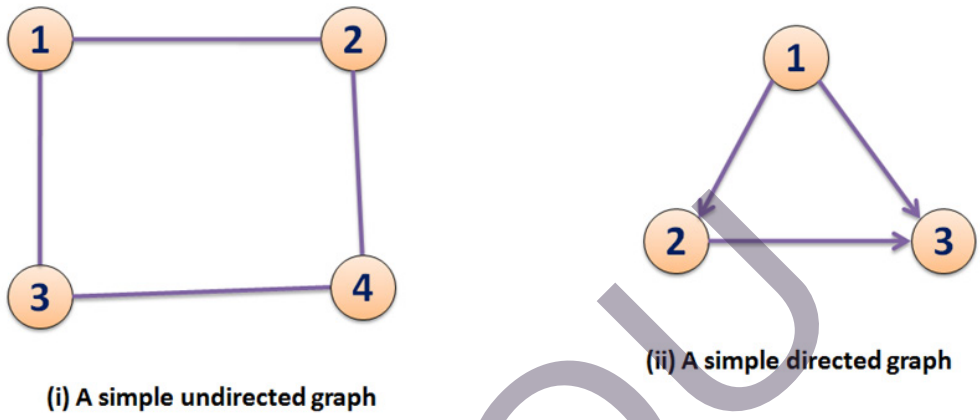


Fig. 3.4.10 Examples for Simple Graph

3.4.3.7 Multi Graph

A graph which has either a self loop or parallel edges or both is called a multi graph.

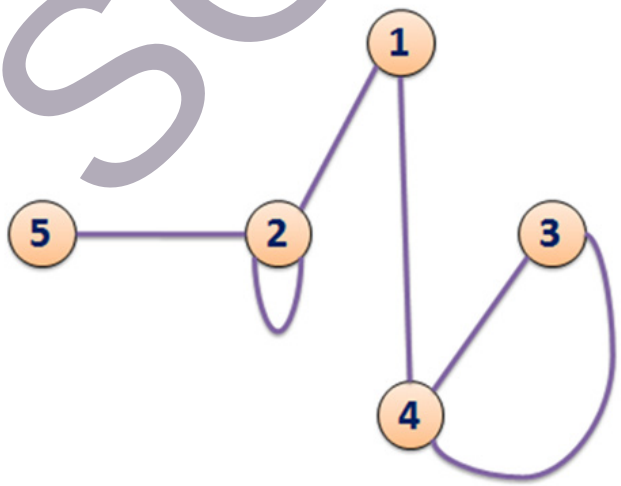


Fig. 3.4.11 Example for Multi Graph

Figure 3.4.11 shows an example for a multi graph. The graph consists of 5 vertices and 6 edges. There is a self loop in node 2. The vertices 4 and 3 are connected with 2 parallel edges. So it is a multi graph.

Maximum Edges in Graph

In a simple undirected graph there can be n(n–1)/2 maximum edges and in a simple directed graph there can be n(n–1) maximum edges. Where n is the total number of vertices in the graph.

Fig. 3.4.12 Examples for Maximum edges

For example (Figure 3.4.12 (i)), if the number of nodes of a simple undirected graph is 2, then the maximum number of edges will be 1. That is (2 x (2-1))/2 = 1.

For example (Figure 3.4.12(ii)), if the number of nodes of a simple directed graph is 2, then the maximum number of edges will be (2 x 1) = 2.

3.4.3.8 Complete Graph

A graph is said to be complete if each vertex is adjacent to every other vertex in the graph.

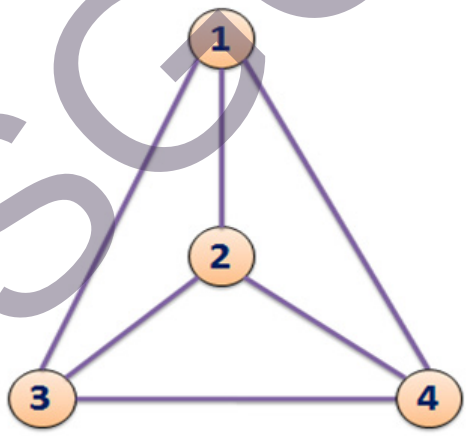


Fig. 3.4.13 Example for Complete Graph

Figure 3.4.13 shows an example for a complete graph. In figure we can see four vertices; say 1, 2, 3, and 4. Vertex 1 is adjacent to vertices 2, 3, and 4. Vertex 2 is adjacent to vertices 1, 3, and 4. Vertex 3 is adjacent to vertices 1, 2, and 4. Vertex 4 is adjacent to vertices 1, 2, and 3. So each vertex is adjacent to every other vertex in the graph. So we can call it a complete graph. The total number of edges in an undirected complete graph will be nn–12 ; where n is the total number of vertices or nodes

In Figure 3.4.13, n = 4. So the total number of edges will be (n*(n-1))/2; ie. 12/2 = 6 edges.

3.4.3.9 Regular Graph

A graph is regular if every node is adjacent to the same number of nodes. That is, each node in the graph has the same degree. Figure 3.4.14 shows an example of a regular graph. It consists of 10 nodes. Each node is adjacent to the same number of nodes. That is, each node has the same degree. Take node 1. It is adjacent to nodes 2, 3 and 6. Therefore the degree of node 1 is 3. Node 2 is adjacent to nodes 1, 4 and 7. The degree of node 2 is 3.

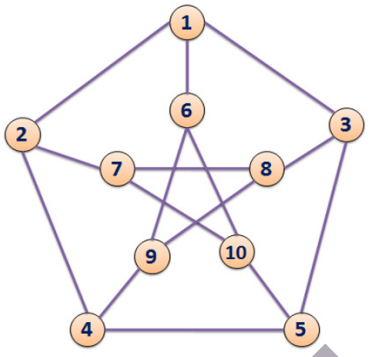


Fig. 3.4.14 Example for Regular Graph

Likewise we can see that all other nodes in the graph have the same degree. That is the degree of nodes 3, 4, 5, 6, 7, 8, 9 and 10 is equal to 3. So it is a regular graph.

3.4.3.10 Planar Graph

A graph is planar if it can be drawn in a plane without any two edges intersecting.

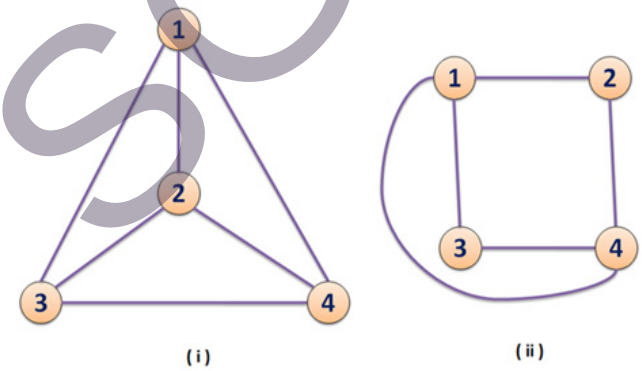


Fig.3.4.15 Examples for Planar Graphs

Figure 3.4.15 shows examples for planar graphs. In both the graphs no two edges are intersecting.

3.4.3.11 Walk & Path

In a graph ***G***. a **walk** is a finite sequence of edges of the form $\mathbf{v_0v_1}$, $\mathbf{v_1v_2}$, $\mathbf{v_2v_3}$, ……., $\mathbf{v_{n-1}v_n}$. And we can also denote this by $\mathbf{v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow ….. \rightarrow v_n}$ in which any 2 consecutive edges are adjacent or identical. Such a walk determines a sequence of

vertices $v_0, v_1, v_2, \ldots\ldots, v_n$. We can call v_0 as the initial vertex and v_n as the final vertex of the walk. That is a walk from v_0 to v_n.

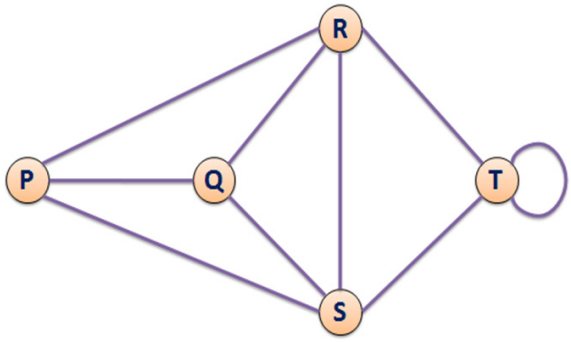


Fig. 3.4.16 Example for walk and path

The number of edges in a walk called its length. For example in Figure 3.4.16, P → Q → R → S → T → T → S → Q is a walk of length 7 from P to Q.

If all the edges in a walk are distinct then it is called a trail. If all the vertices in a trail are distinct then it is called a path. That is the vertices in a trail, v0, v1, v2, ….., vn are distinct, then the trail is a path. If v0 = vn then the path or trail is called a closed path or a closed trail.

Consider Figure 3.4.16 we can see that

P → Q → R → S → T → T → R is a trial.

P → Q → R → S → T is a path.

P → Q → R → S → T → R → P is a closed trail.

P → Q → S → T → R → P is a closed path.

3.4.3.12 Cycle

If there is a path containing one or more edges which starts from a vertex and terminates into the same vertex then the path is called a cycle. A closed path containing at least one edge is a cycle. Any loop or pair of multiple edges is a cycle.

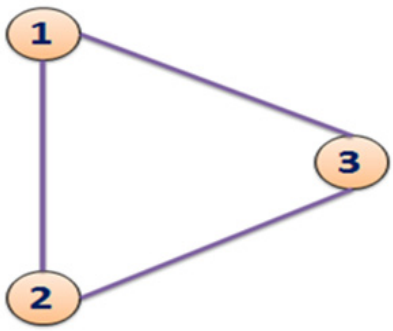


Fig. 3.4.17 Examples for a Cycle

Figure 3.4.17 shows an example for a cycle. Take the vertex 1, we can see a path **1 - 2 – 3-1**. That is a path starting from 1 and terminating at 1. So it is a cycle.

3.4.3.13 Cyclic Graph

A graph that has cycles is called a cyclic graph. In Figure 3.4.18 we can see both directed and undirected graphs with a cycle.

The 1st digraph is cyclic because it contains a cycle 1 - 2 - 3 - 1.

The 2nd undirected graph is also cyclic because it contains a cycle 1 - 2 - 3 - 4 - 1.

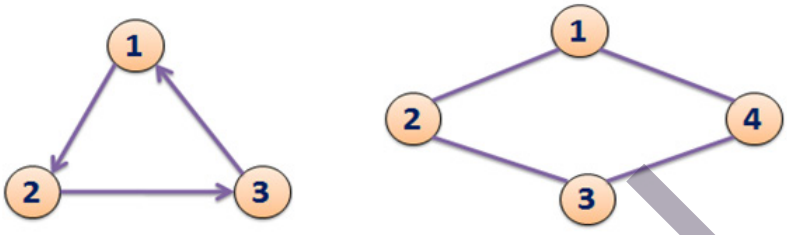


Fig. 3.4.18 Examples for Cyclic Graph

3.4.3.14 Acyclic Graph

If a graph does not have any cycle then it is called an acyclic graph. In Figure 3.4.19 we can see both directed and undirected graphs with no cycles. The first digraph is acyclic because it contains no cycle. It contains the following paths;

1 - 2 - 3, 1 - 3, 2 - 3

All the three paths are not closed. So there is no cycle. That is the digraph is acyclic.

Fig. 3.4.19 Examples for Acyclic Graph

The second undirected graph shown in Figure 3.4.19 , is also acyclic because it contains no cycle. It contains the following paths;

1 - 2 - 3, 3 - 2 - 1, 2 -1, 1 - 2, 2 - 3, 3 - 2

All the six paths are not a cycle. So the graph is acyclic.

3.4.3.15 Connected Graph

In a graph G, two vertices v1 and v2 are said to be connected if there is a path in G from v1 to v2 or v2 to v1. A graph is said to be connected if there is a path from any node of graph to any other node. That is, for every pair of distinct vertices in G, there exists a path. Figure 3.4.20 (i) shows a connected graph. Here we can see that each pair of nodes is connected. There is a path from node 1 to nodes 2, 3 and 4. Likewise all other nodes have a path to every other node. Figure 3.4.20 (ii) shows a disconnected graph. The graph is disconnected because there is no path to node 4 from the nodes 1, 2 and 3.

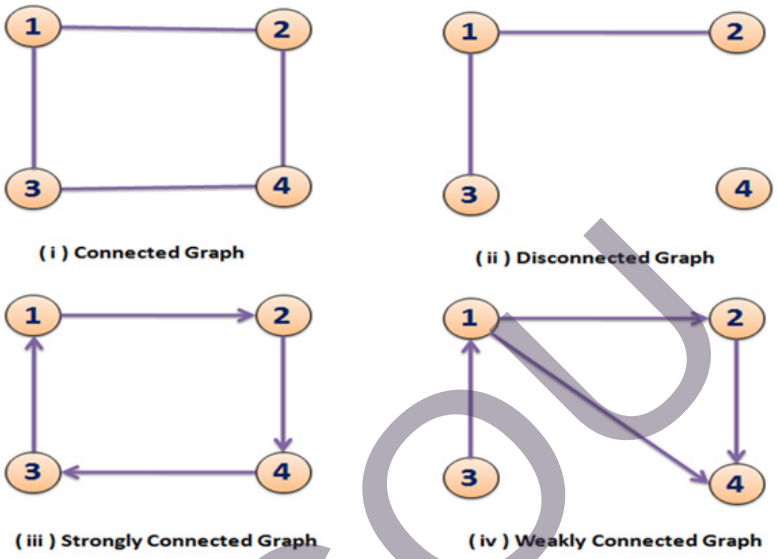


Fig. 3.4.20 Examples for Connected and Disconnected Graphs

A directed graph G is said to be strongly connected, if every pair of distinct vertices in G, there is a path. Figure 3.4.20 (iii) shows a strongly connected digraph. For example take the case of vertex 1; there is a path from 1 to 2, 1 to 3, and 1 to 4. In the case of vertex 2; there is a path from 2 to 3, 2 to 4, and 2 to 1. Likewise in the case of vertex 3 and vertex 4; there is a path to all other vertices.

A digraph is called weakly connected, if for any pair of vertices u and v, there is a path from u to v or a path from v to u. In a digraph, if we replace the directed edge with undirected edges and the resulting graph is connected then that digraph is weakly connected. Figure 3.4.20 (iv) shows a weakly connected graph. In the case of vertex 1; there is a path from 1 to vertices 2 and 4, and a path from vertex 3 to 1. If we replace all the directed edges with undirected edges here, then the resultant graph is a connected one. That is there is a path from vertex 1 to vertices 2, 3, and 4. There is a path from vertex 2 to vertices 1, 3, and 4. There is a path from vertex 3 to vertices 1, 2, and 4. There is a path from vertex 4 to vertices 1, 2, and 3. So the graph is weakly connected.

3.4.3.16 Articulation Point

If on removing a node from the graph, the graph becomes disconnected then that node is called the articulation point.

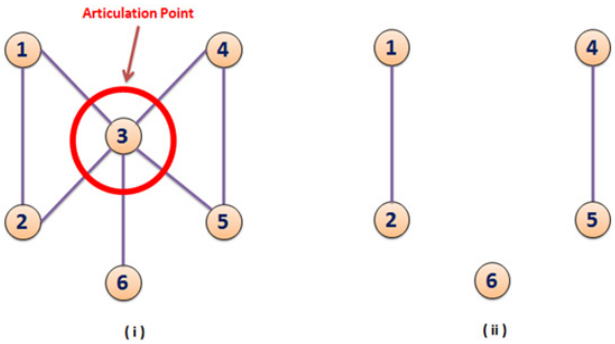


Fig. 3.4.21 Example for Articulation point in a graph

In Figure 3.4.21 (i), node 3 is an articulation point because on removing node 3, the graph becomes disconnected which is shown in Figure 3.4.21 (ii). On removing node 3 from the graph, all the 5 edges connected to node 3 are removed. That is, edges 3-1, 3-2, 3-6, 3-5 and 3-4 are removed. Then the resultant graph becomes disconnected.

3.4.3.17 Bridge

If on removing an edge from the graph, the graph becomes disconnected then that edge is called the bridge.

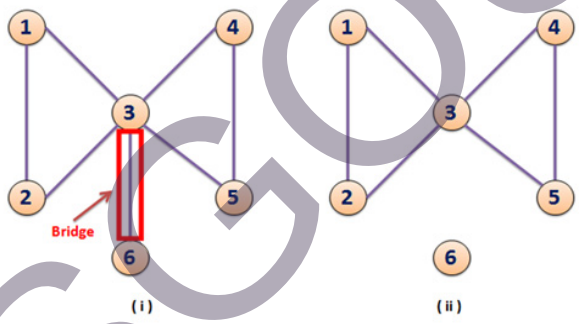


Fig. 3.4.22 Example for Bridge in a graph

In Figure 3.4.22 (i) edge 3-6 is a bridge because if this edge is removed then the graph becomes disconnected which is shown in Figure 3.4.22 (ii).

3.4.3.18 Biconnected Graph

A graph with no articulation points is called a biconnected graph.

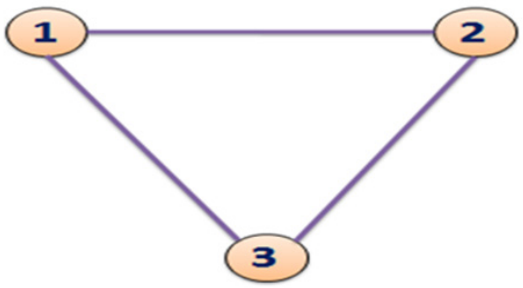


Fig.3.4.23 Example for Biconnected Graph

In Figure 3.4.23, there is no articulation point. There are three vertices and three edges in the given graph. Try to remove vertex 1 from the graph which does not make the graph disconnected.

By removing vertex 2 does not make the graph disconnected. By removing vertex 3 does not make the graph disconnected. So it is a biconnected graph.

3.4.4 Graph Traversal

Traversal is a searching technique used in graphs. For searching a particular vertex in a given graph we use traversal. So the main goal of graph traversal is to find all vertices or nodes reachable from a given set of nodes. The order of vertices visited in a search process is also decided by the traversal. It also finds the edges to be used without creating any loops. That is in graph traversal all the vertices are visited without getting into a looping path. We can follow all the edges in an undirected graph. But in a directed graph we can follow only the out edges.

Traversal in graph is different from tree traversal because;

- There is no root node in the graph, so the traversal can start from any node.
- Only those nodes are traversed which are reachable from the starting node. If we want to traverse all the reachable nodes, we again have to select another starting node for the remaining nodes.
- While traversing a graph there may be a possibility that we reach a node more than once. To ensure that each node is visited only once we have to keep the status of each node whether it has been visited or not.

The main graph traversal techniques are Breadth First Search (BFS) and Depth First Search (DFS). We can study each of them in detail in the following sessions. Breadth first search uses a queue structure for keeping the nodes and processing. In Depth first search we use a stack structure for node processing.

3.4.4.1 Breadth First Search (BFS)

BFS technique use queue for traversing all the nodes in the given graph. Here we first take any node as a starting node. That node is placed in the queue at first. So the front element of the queue is the starting node. Then we take all the adjacent nodes of the starting node. They are placed behind the starting node in the queue. After entering all the adjacent nodes of the staring node; that node is deleted from the queue. That is the first node is traversed successfully.

Then the second node in the queue becomes the front element of the queue. We take that node for processing. That is, we find all the adjacent nodes and place them in the queue if it is not present in the queue. So the front node in the queue is traversed successfully and we delete it from the queue. Then we take the next front node in the queue. Similar approach is done for all the other adjacent nodes placed in the queue. Likewise each node is processed.

Algorithm

Step 1. Define a queue for storing the nodes of the graph.

Step 2. Select any node as a starting node for traversal and insert it into the queue.

Step 3. Delete the front element from the queue and insert all its unvisited adjacent nodes into the queue at the end.

Step 4. Repeat step 3 until the queue becomes empty.

Example :

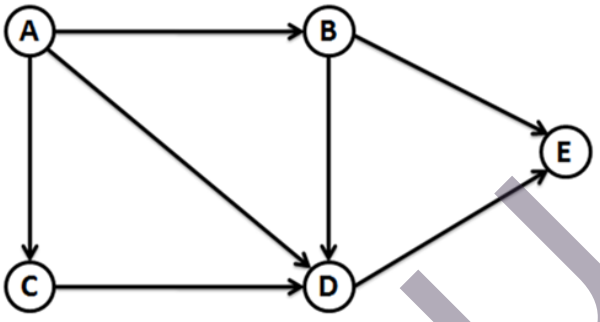


Fig.3.4.24 Example graph for BFS

Let us consider a graph shown in Figure 3.4.24 for breadth first traversing. Take the starting node as node A. The adjacent nodes of A are B, C, and D. The following steps illustrate the BFS.

1. Initially insert the starting node into the queue. That is insert node A into the queue. Here; Front = Rear = 0. The resultant graph and queue are shown in Figure 3.4.25.

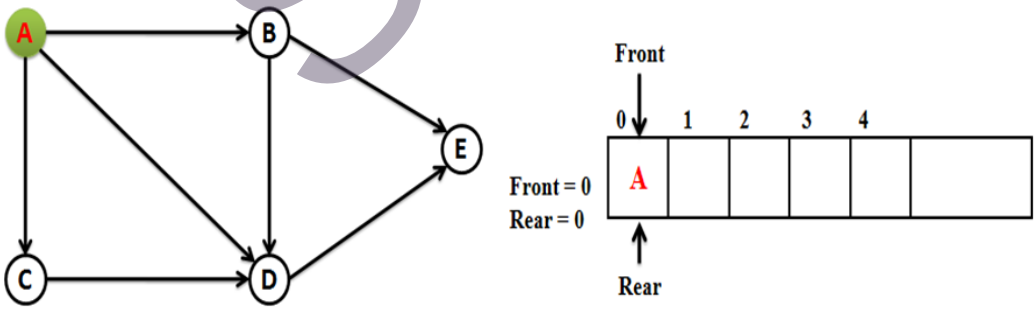


Fig. 3.4.25 Breadth First Traversal - Resultant graph, queue (1)

2. Remove the front element A from the queue and increment Front = Front + 1. That is Front becomes 1. Insert the adjacent nodes of A to the queue if it is not present in the queue. Here; B, C, and D are the adjacent nodes and insert them into the queue.

So the Front = 1 and Rear = 3. Figure 3.4.26 shows the resultant graph and queue.

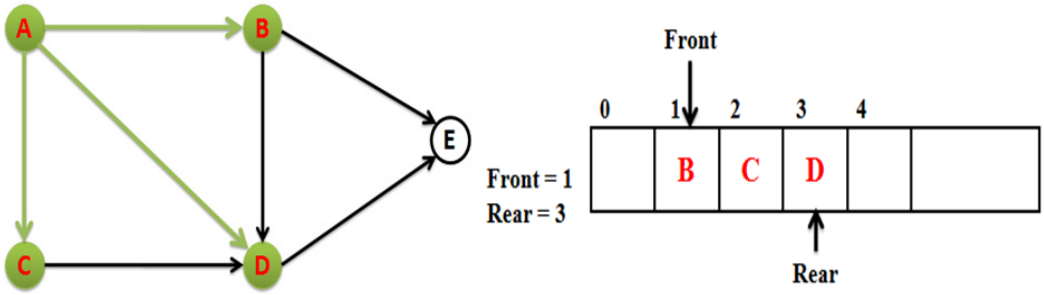


Fig. 3.4.26 Breadth First Traversal - Resultant graph, queue (2)

3. Remove the front element B from the queue and add the adjacent nodes of B to the queue, if it is not present in the queue. The adjacent nodes of B are D and E. Here D is already in the queue. E is not in the queue. So insert E to the rear position. So the Rear = 4 and Front = 2. Figure 3.4.27 shows the resultant graph and queue.

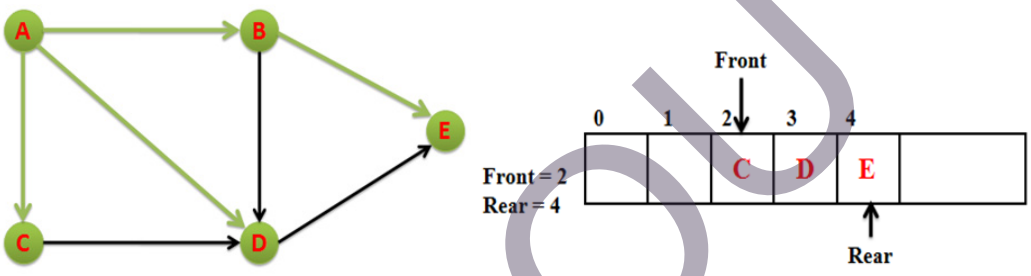


Fig. 3.4.27 Breadth First Traversal - Resultant graph, queue (3)

4. Remove the front element C from the queue and add the adjacent nodes of C to the queue, if it is not present in the queue. The adjacent node of C is D. Here D is already in the queue. So the Rear = 4 and Front = 3. Figure 3.4.28 shows the resultant graph and queue.

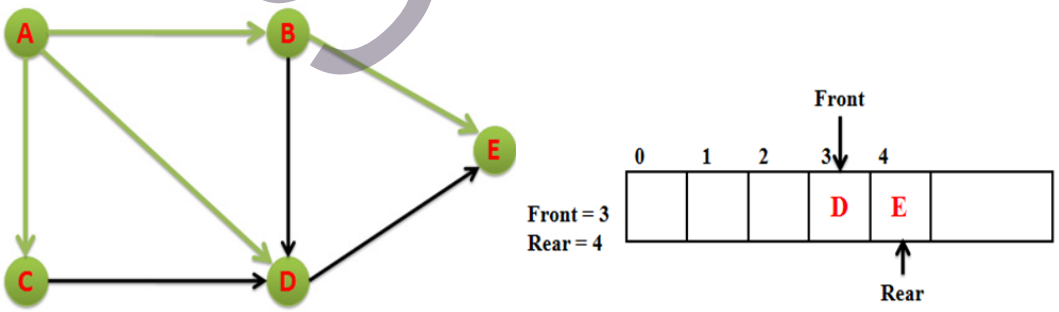


Fig. 3.4.28 Breadth First Traversal - Resultant graph, queue (4)

5. Remove the front element D from the queue and add the adjacent nodes of D to the queue, if it is not present in the queue. The adjacent node of D is E. Here E is already in the queue. So the Rear = 4 and Front = 4. Figure 3.4.29 shows the resultant graph and queue.

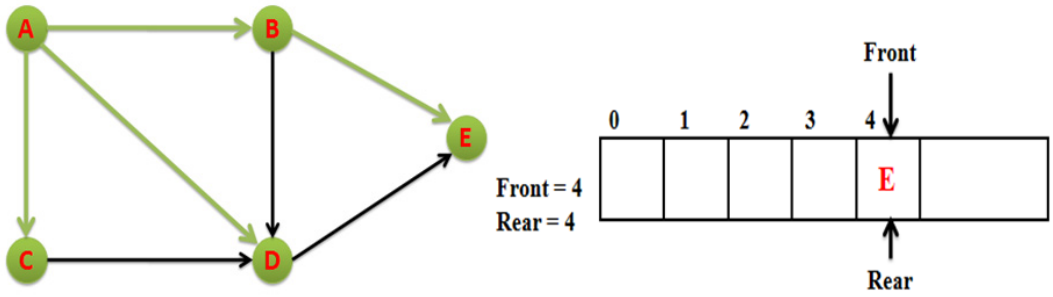


Fig. 3.4.29 Breadth First Traversal - Resultant graph, queue (5)

6. The process is repeated until Front & Rear. Remove the front element E from the queue and add the adjacent nodes of E to the queue, if it is not present in the queue. Here the node E has no adjacent nodes. So the queue becomes empty. Also Front and Rear. Here Front = 5 and Rear = 4. So we stop the traversal. Figure 3.4.30 shows the resultant graph and queue.

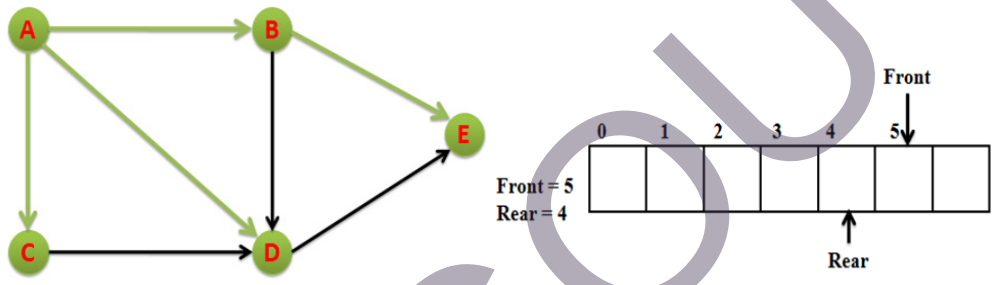


Fig. 3.4.30 Breadth First Traversal - Resultant graph, queue (6)

And finally the traversed nodes are A, B, C, D, E.

3.4.4.2 Depth First Search (DFS)

Depth First Search (DFS) starts from a source node. We can select any node from the graph as a starting or source node. If S1 is the starting node, then DFS first visits S1 and then visits any one of its adjacent nodes, say S2. Then DFS again visits an adjacent node of S2, say S3. In the next step DFS again visits any one of the adjacent nodes of S3, say S4 and so on. DFS uses a stack for processing nodes. DFS uses a backtracking method for traversing all unvisited nodes in the graph. For example if node S4 has no adjacent nodes then it backtracks the traversal to the previous node. That is S4 backtracks to S3. Then DFS looks for any other adjacent node of S3. If it exists then visit that node. If not exists then backtrack to the previous node S2. This process is repeated until all the unvisited nodes get visited.

Algorithm

Step 1. Define a stack for storing the nodes of the graph.

Step 2. Select any node as a starting node for traversal and push it into the stack

Step 3. Visit any one of the unvisited adjacent nodes of a node which is on the top of the stack and push it onto the stack

Step 4. Repeat Step 3 until there is no new node to be visited from the node which is on the top of the stack.

Step 5. If there is no unvisited adjacent node, it remains to backtrack the traversal and pop the top node from the stack.

Step 6. Again repeat the Steps 3, 4, and 5 until the stack becomes empty.

Example : Let us consider a graph shown in Figure 3.4.31 for depth first traversing. Take the starting node as node A.

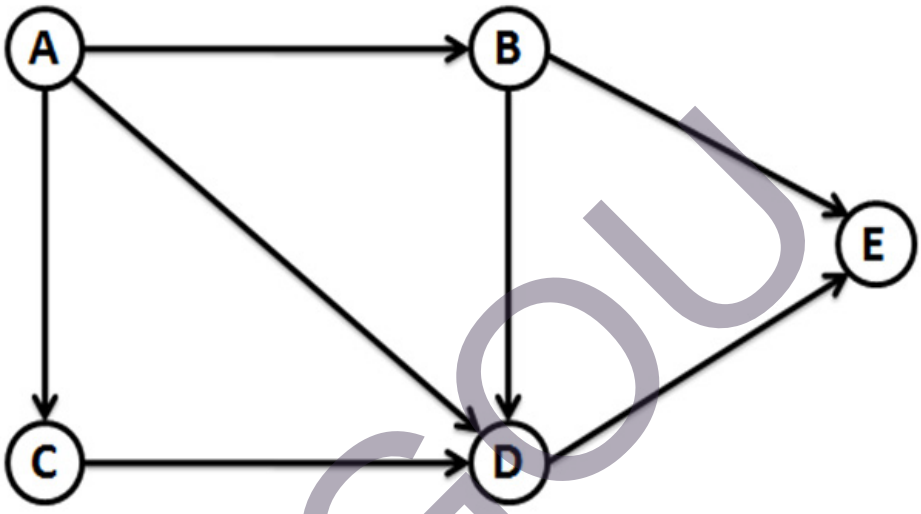


Fig. 3.4.31 Example graph for DFS

The following steps illustrate the DFS.

1. Initially insert the starting node into the stack. That is push node A into the stack. Here top = 0. Figure 3.4.32 shows the resultant graph and stack.

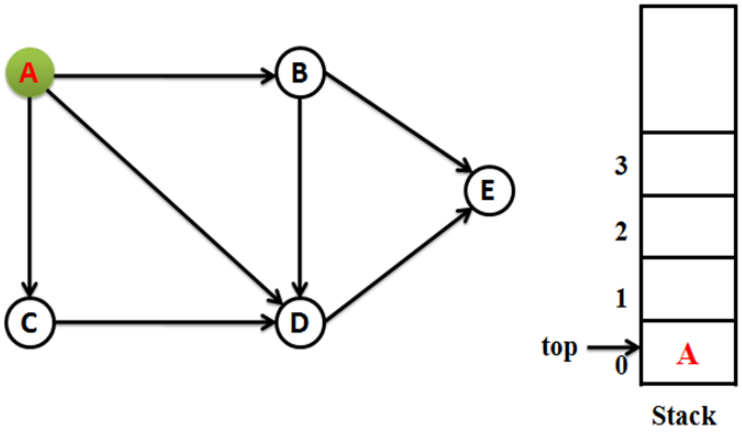


Fig. 3.4.32 Depth First Traversal - Resultant graph, stack (1)

2. Visit any one of the unvisited adjacent node of A. Here we visit the node B. Then push B into the stack. So top = 1. Figure 3.4.33 shows the resultant graph and stack.

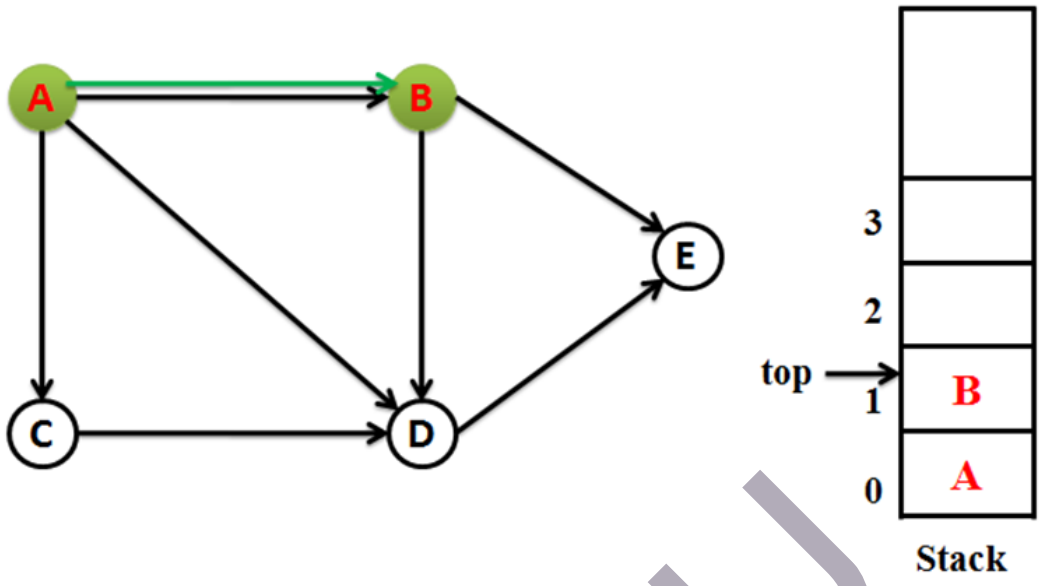


Fig. 3.4.33 Depth First Traversal - Resultant graph, stack (2)

3. Visit any one of the unvisited adjacent node of B. Here we visit the node D. Then push D into the stack. So top = 2. Figure 3.4.34 shows the resultant graph and stack.

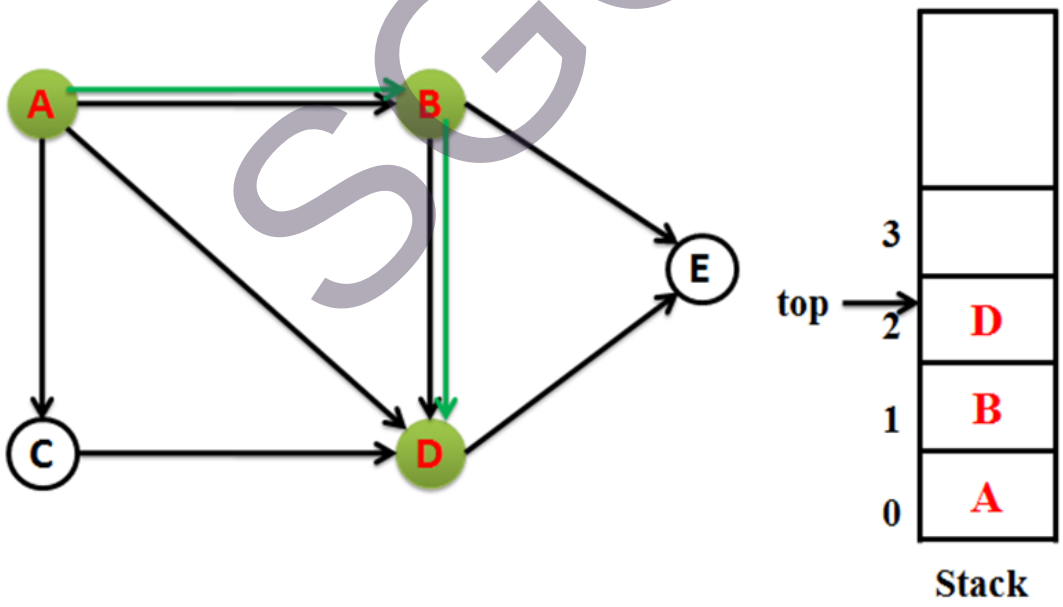


Fig. 3.4.34 Depth First Traversal - Resultant graph, stack (3)

4. Visit any one of the unvisited adjacent node of D. Here the node is E. So we visit the node E. Then push E into the stack. So top = 3. Figure 3.4.35 shows the resultant graph and stack.

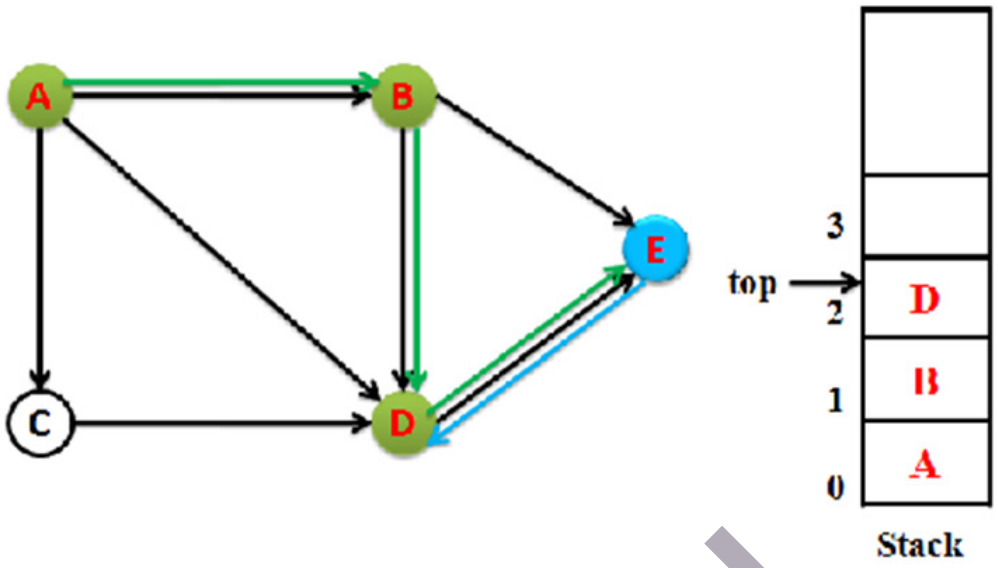


Fig. 3.4.35 Depth First Traversal - Resultant graph, stack (4)

5. Visit any one of the unvisited adjacent nodes of E. There is no adjacent nodes for node E. So we backtrack the traversal. Pop node E from the stack. So top = 2. Figure 3.4.36 shows the resultant graph and stack.

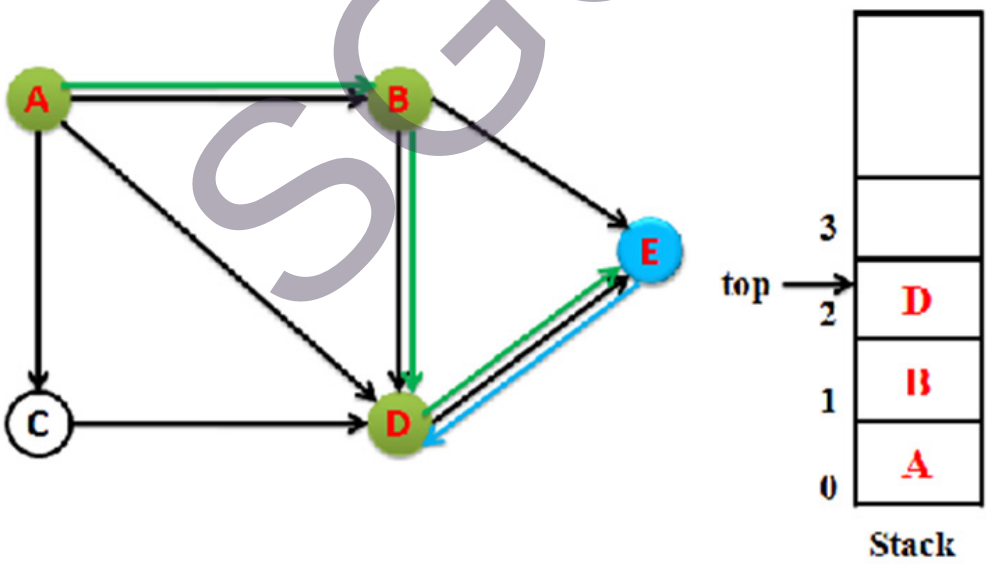


Fig. 3.4.36 Depth First Traversal - Resultant graph, stack (5)

6. Visit any one of the unvisited adjacent node of D. There is no other unvisited adjacent node remaining for node D. So we backtrack the traversal. Pop node D from the stack. So top = 1. Figure 3.4.37 shows the resultant graph and stack.

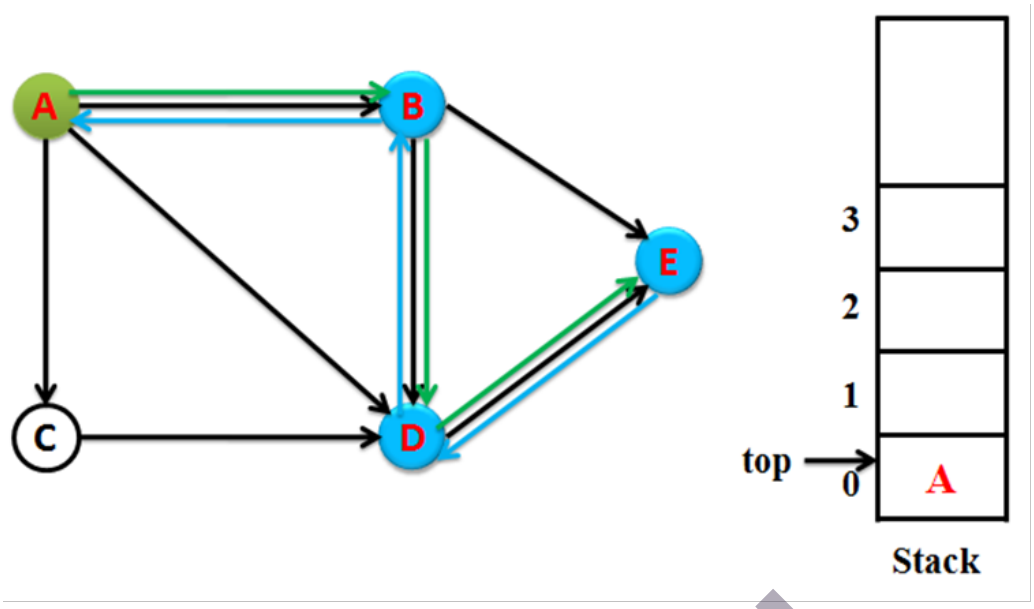


Fig. 3.4.37: Depth First Traversal - Resultant graph, stack (6)

7. Visit any one of the unvisited adjacent nodes of B. There is no other unvisited adjacent node remaining for node is B. So we backtrack the traversal. Pop node B from the stack. So top = 0. Figure 3.4.38 shows the resultant graph and stack.

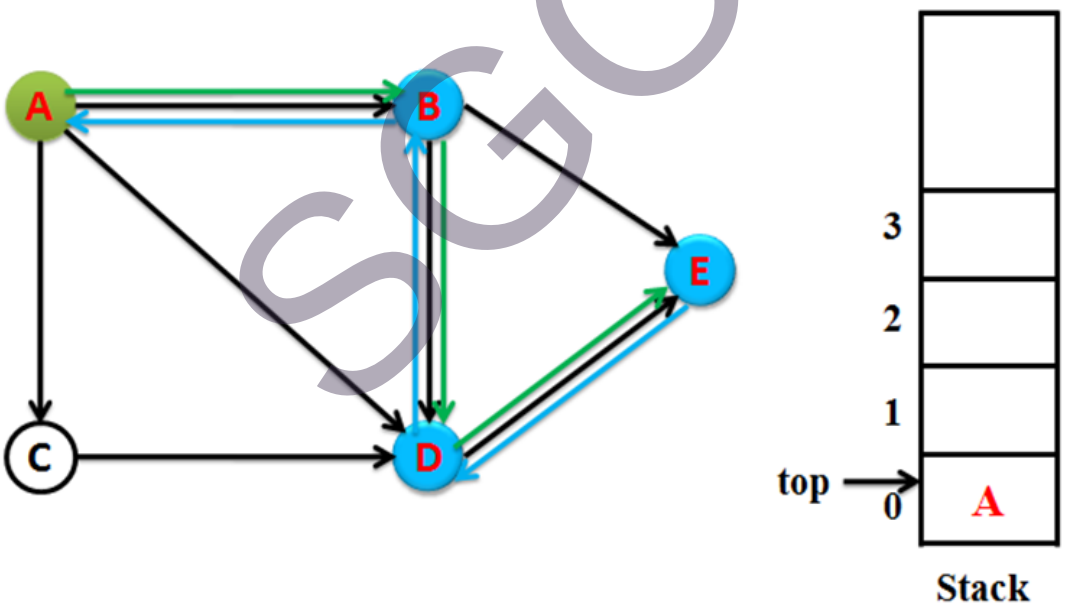


Fig.3.4.38 Depth First Traversal - Resultant graph, stack (7)

8. Visit any one of the unvisited adjacent node of A. Here only one unvisited node of A remains and that is the node C. So we visit the node C. Then push C into the stack. So top = 1. Figure 3.4.39 shows the resultant graph and stack.

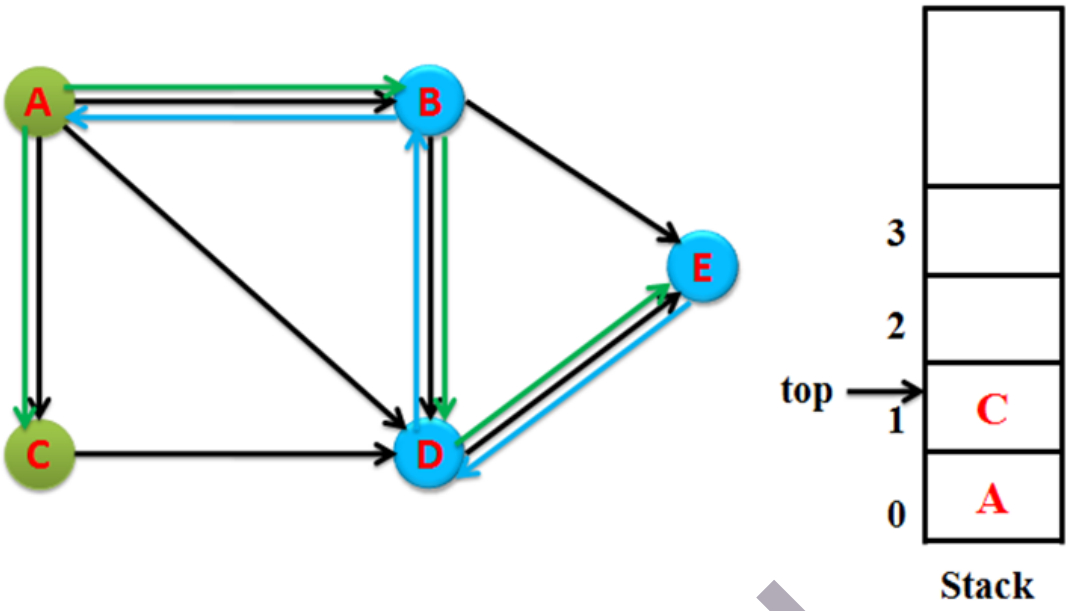


Fig.3.4.39 Depth First Traversal - Resultant graph, stack (8)

9. Visit any one of the unvisited adjacent node of C. There is no other unvisited adjacent node remains for node is C. So we backtrack the traversal. Pop node C from the stack. So top = 0. Figure 3.4.40 shows the resultant graph and stack.

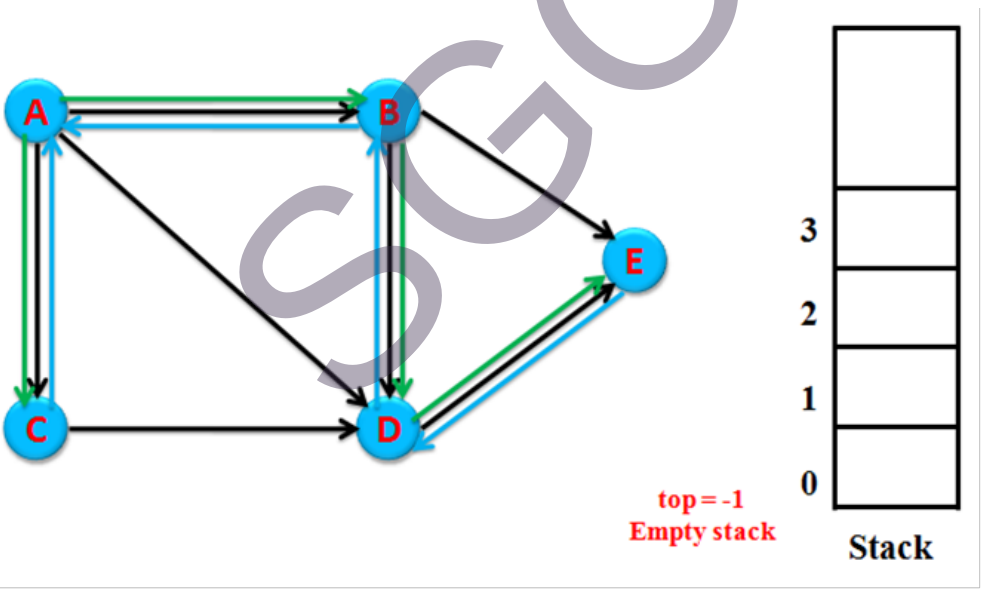


Fig. 3.4.40 Depth First Traversal - Resultant graph, stack (9)

10. Visit any one of the unvisited adjacent nodes of A. There is no other unvisited adjacent node remaining for node A. So we backtrack the traversal. Pop node A from the stack. So top = -1. That is, the stack becomes empty. So we can stop DFS. Figure 3.4.41 shows the resultant graph and stack.

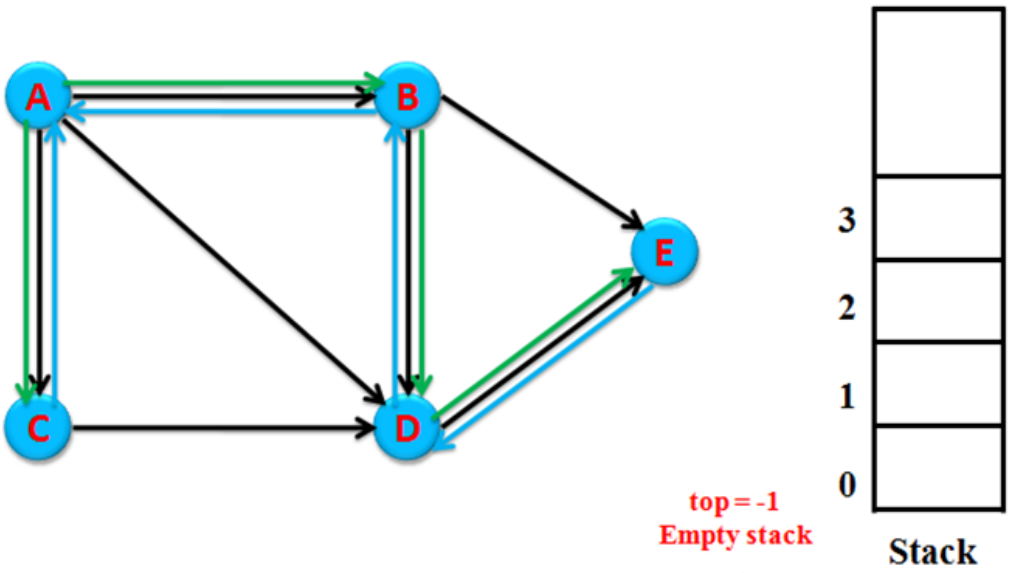


Fig. 3.4.41 Depth First Traversal - Resultant graph, stack (10)

And finally all the nodes are traversed.

3.4.5 Applications of Graph

Graphs are versatile data structures used to model relationships between entities, making them essential in various applications. Some of the important applications of graphs are traveling salesperson problem, GPS Navigation Systems, Social Networks, Knowledge Graphs, etc.

3.4.5.1 Traveling Salesman Problem (TSP)

Suppose you are an area manager of a financial firm. Your duty is to manage all the branches coming under your area. Your office is attached to the main branch. But you have to visit all the other branches periodically. Your higher authority insists that you make a quick inspection in all branches that come under you and submit the report to him in five days. What will you do first? How you will manage this situation. You have to find the shortest route to visit each branch and return back to your office without failure. A proper planning is needed for this purpose. The Traveling Salesman Problem (TSP) is also like this. TSP consists of a salesman and a set of cities. The salesman has to visit all the cities starting from a certain one (home town), by selecting shortest routes and returning back to his place where he started. So the challenge of TSP is to minimize the total length of the trip.

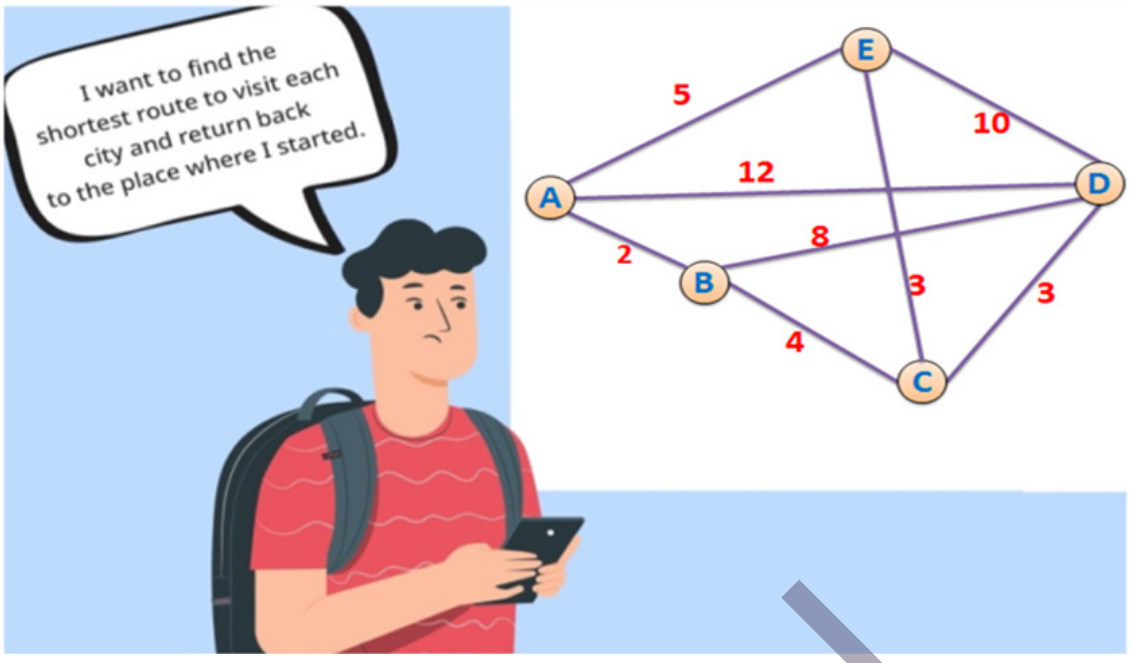


Fig. 3.4.42 Traveling Salesman Problem

In Figure 3.4.42, we can see a set of cities represented by a graph. Here each city is represented by a node. Here the problem lies in finding a minimal path passing from all nodes once. Suppose the home town is A. That is the salesman starts his journey from the city A. He must visit all the cities (B, C, D, and E) and returns back to his home town (A). So he has to find a route with minimum distance that starts from A and pass through all the cities once and return back to A. For example take the path A-B-C-D-E-A. We can represent it as Path 1 = {A, B, C, D, E, A}. And take the path A-B-C-E-D-A. We can represent it as Path 2 = {A, B, C, E, D, A}. Both the paths pass all the nodes but Path 1 has a total length of 24 and Path 2 has a total length of 31. So considering all other possible paths, the salesman has to choose a shortest route. Here Path 1 is the shortest route and it is selected.

3.4.5.2 Google Map

Suppose you are going to attend your friend's marriage at Kunnamkulam. But you have no idea about that place. Now you are staying at Wadakkanchery. What will you do? You will search that place in Google map and you can see (Figure 3.4.43) the different routes to Kunnamkulam from Wadakkanchery. It shows you the distance and approximate travel time to reach the destination.

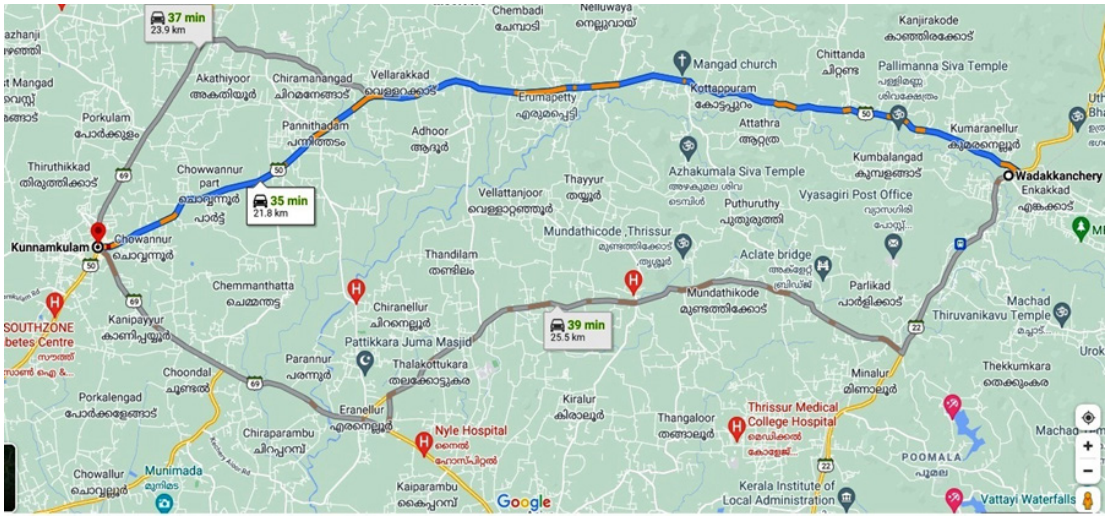


Fig. 3.4.43 Google Map

Google maps uses graphs for building transportation systems. In Google map, various locations are represented as vertices and roads connecting these locations are represented as edges. The intersection of two or more roads is also considered to be a vertex. The navigation system is based on the algorithm to calculate the shortest path from one vertex to another. In the previous example we can see three paths between Wadakkanchery and Kunnamkulam. We can represent these paths using graphs.

Figure 3.4.44 shows the graph representation. Here Wadakkanchery is the source vertex represented with vertex 1 and Kunnamkulam is the destination vertex represented with vertex 6. The interconnection of two or more roads that are coming in these different routes (paths) can be represented as vertices (vertices 2, 3, 4 and 5). Here $\mathbf{w}_{12}$ is the distance between nodes 1 and 2 and $\mathbf{w}_{26}$ is the distance between the nodes 2 and 6.

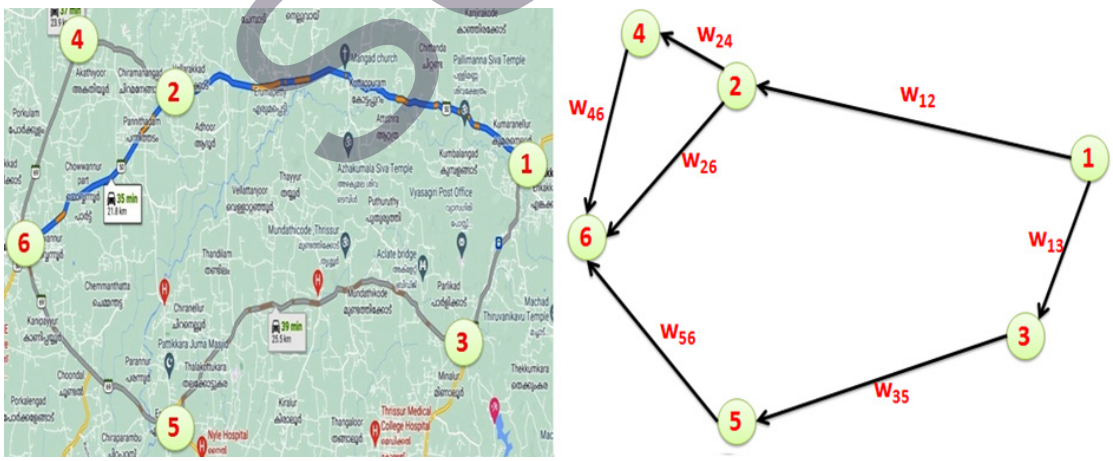


Fig. 3.4.44 Graph representation of routes in Google Map

Likewise $\mathbf{w}_{13}$, $\mathbf{w}_{35}$, $\mathbf{w}_{56}$, $\mathbf{w}_{24}$ and $\mathbf{w}_{46}$ are the distances between the corresponding nodes. Here three paths exist between node 1 and node 6. They are; Path 1= 1-2-6, Path 2= 1-2-4-6 and Path 3= 1-3-5-6. Here the shortest path is path 1. So it is recommended to the user.

3.4.5.3 GPS Navigation System

For vehicle navigation we use Global Positioning System or GPS. It is also a type of shortest path routing API and differs from Google Maps routing API because it uses single source (from one vertex to every other). That is it computes locations from where you are to any other location you might be interested in going. Here BFS is used to find all the neighboring locations. Stand alone GPS devices actually store their own map data compared to a smart phone which is cloud based. It requires a connection to download the maps as you go.

Summarised Overview

A spanning tree is a subgraph of a connected, undirected graph that connects all vertices with exactly n−1 edges, ensuring connectivity without cycles; if the graph is disconnected, spanning forests are formed. A minimum cost spanning tree (MST) is a spanning tree with the smallest total edge weight, useful in network design, routing, and resource optimization. Two main algorithms to find MSTs are Kruskal's Algorithm (edge-based, sorts edges and adds the smallest non-cyclic ones; efficient for sparse graphs) and Prim's Algorithm (vertex-based, expands outward from a chosen node using the cheapest connecting edge; efficient for dense graphs).Shortest path algorithms focus on minimizing path costs: Dijkstra's Algorithm (greedy, works with non-negative weights), Bellman–Ford Algorithm (handles negative weights, detects negative cycles), and Floyd–Warshall Algorithm (dynamic programming, finds shortest paths between all vertex pairs). Together, these algorithms are widely applied in telecommunication networks, transportation systems, image segmentation, computer network protocols (like STP), and power distribution, as they ensure efficient connectivity, cost minimization, and reliability.

Assignments

1. Apply Kruskal's Algorithm to the following graph and find its MST. Show each step clearly. Vertices = {A, B, C, D, E} Edges = { (A–B, 1), (A–C, 3), (B–C, 2), (B–D, 4), (C–E, 5), (D–E, 7) } Apply Prim's Algorithm to the same graph (start from vertex A) and find the MST. Compare the steps with Kruskal's Algorithm.

2. Compare Kruskal's Algorithm, and Prim's Algorithm.
3. Discuss about shortest path algorithms.
4. Explain the applications of a Spanning Tree in real-world scenarios?

Reference

1. Kumar, A., & Yadav, J. (2024). Minimum spanning tree clustering approach for effective feature partitioning in multi-view ensemble learning. *Knowledge and Information Systems, 66*(3), 6785–6813.
2. Zhao, Y., & Zhang, L. (2023). Innovative method to solve the minimum spanning tree problem: The LC-MST method. *Computers, Materials & Continua, 74*(3), 2811–2825.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
4. Kleinberg, J., & Tardos, É. (2006). *Algorithm design*. Pearson Education.
5. Chao, K. M. (2003). *Spanning trees and optimization problems*. Springer

Suggested Reading

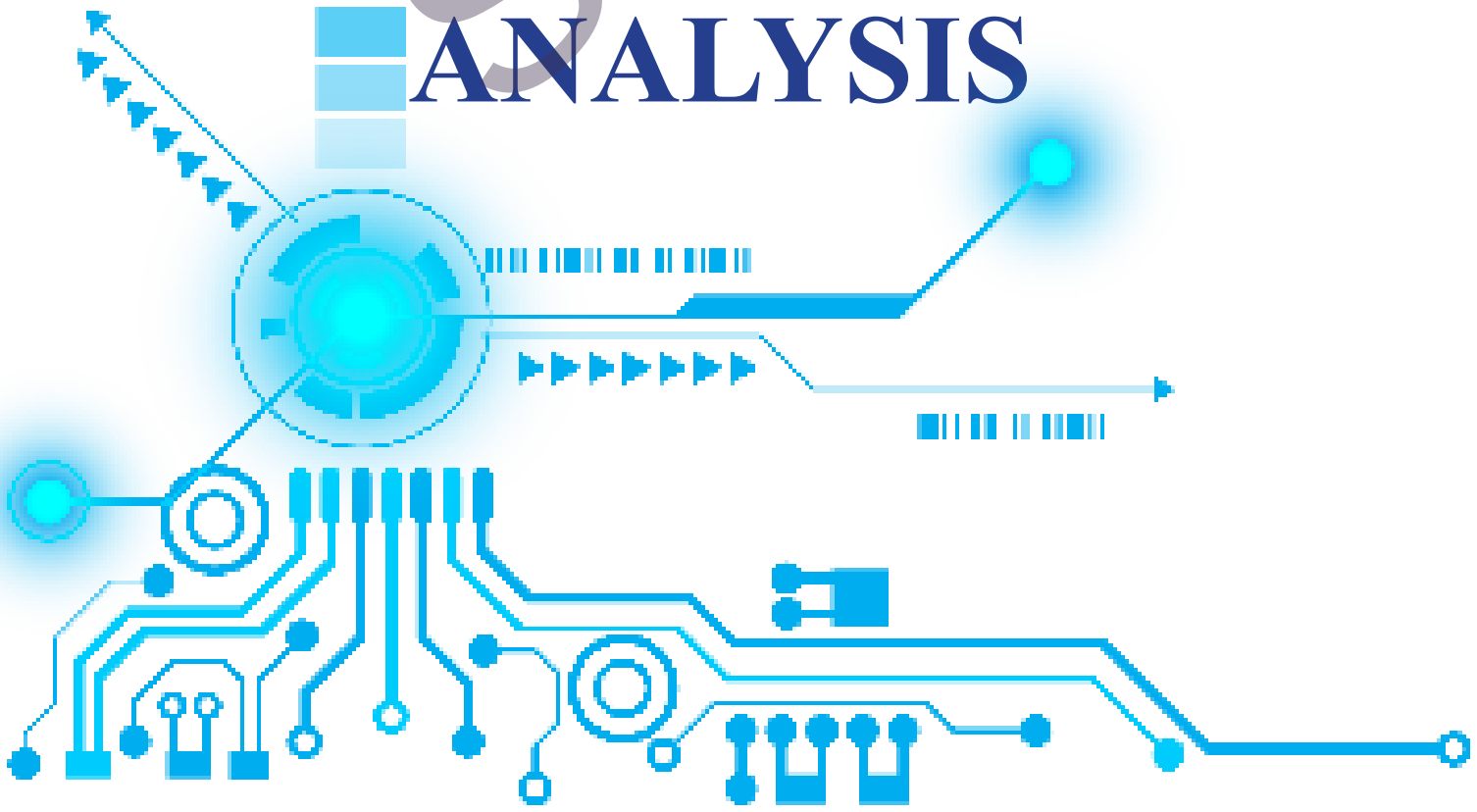
1. https://www.geeksforgeeks.org/dsa/spanning-tree
2. https://doi.org/10.1007/s10107-023-01791-6
3. *Minimum Spanning Tree vs Shortest Path Tree*. https://www.baeldung.com/cs/minimum-spanning-vs-shortest-path-trees
4. https://doi.org/10.32604/cmc.2023.023929

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 4

HASH TECHNIQUES, ALGORITHM ANALYSIS



1 UNIT

Hashing

Learning Outcomes

After completion of this unit, the learner will be able to:

- define hashing and list its main components
- state the formula for the division method in hashing
- identify examples of collisions in a hash table
- name two categories of collision resolution techniques
- recall the steps involved in hash table searching

Background

In our daily life, we often use a method to quickly find things without searching through everything. For example, in a library, books are arranged with unique numbers so that we can go directly to the right shelf. In the same way, in a mobile phone, when we search for a contact name, the phone quickly finds it without checking each contact one by one. This fast searching is possible because of a concept similar to hashing in computers.

In computing, hashing is a way to store and find data quickly by using a key (like a name or number) and a hash function to calculate the exact position of the data in a hash table. This topic will help you understand what hashing is, how different hash functions like division, multiplication, folding, and mid-square methods work, and how collisions (when two keys get the same position) are handled using techniques like chaining and open addressing. You will also learn how searching works in a hash table. By the end, you will see how this simple but powerful idea is used in many systems you already use every day.

Keywords

hashing, hash function, hash table, collision, collision resolution

Discussion

4.1.1 What is Hashing?

Hashing is a method used to quickly store and find data in a computer. It works by taking an input value, which can be of any size (such as a name, number, or word), and turning it into a smaller, fixed-size value using a special mathematical formula called a hash function.

This fixed-size value is usually a number, and it tells us the exact position (index) where the data should be stored in a data structure like a hash table. So, instead of searching through all the data one by one, the computer can go directly to the correct spot by using the result of the hash function.

4.1.2 Components of Hashing

Hashing mainly involves three important components that work together to store and retrieve data efficiently. These are:

1. Key
2. Hash Function
3. Hash Table

1. Key

A key is an input value, such as a string or an integer, that is provided to the hash function. It serves as a unique identifier and is used to determine the index or location where the corresponding data will be stored in a data structure.

2. Hash Function

The hash function is a mathematical formula that takes the input key and computes an index. This index, known as the hash index, indicates the specific position in an array called the hash table where the data should be placed.

3. Hash Table

A hash table is a data structure that stores data by associating keys with values using a hash function. It organizes data in an array format, where each value is stored at a unique index calculated by the hash function, allowing efficient access, insertion, and deletion.

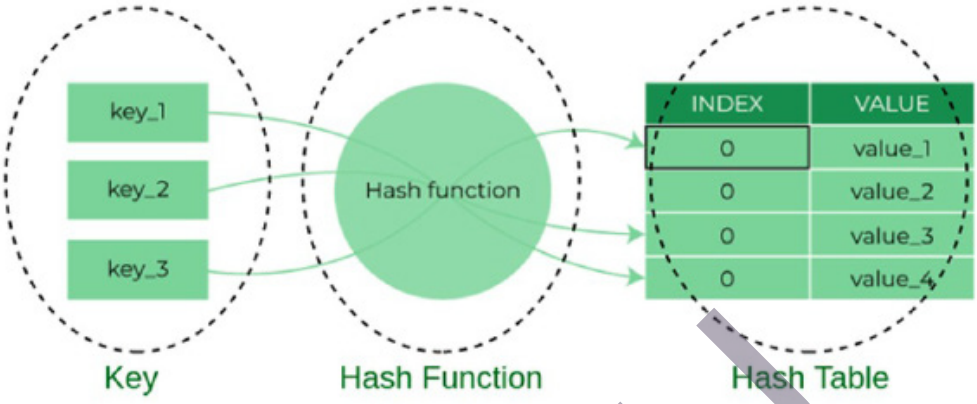


Fig. 4.1.1 Components of Hashing

4.1.2.1 Load Factor in Hash Tables

The load factor is an important concept that measures how full a hash table is. It helps in analyzing the performance and efficiency of hashing.

The load factor is represented by the Greek letter α (alpha) and is defined as:

$$\alpha = \frac{n}{m}$$

Where:

- n = number of elements stored in the hash table
- m = total number of slots (size of the hash table)

The load factor indicates how many elements are stored compared to the available table size.

For example, if a hash table has 10 slots and 5 elements stored in it:

$$\alpha = \frac{5}{10}$$

This means the table is 50% full.

The performance of hashing operations such as searching, insertion, and deletion depends heavily on the load factor. As the load factor increases, the probability of collisions also increases. A smaller load factor generally leads to better performance because there are fewer collisions in the hash table.

4.1.3 Hash Functions

A hash function is a basic mathematical formula used to convert a key (like a number or word) into an index number. This index tells us where to store the data in a hash table. The goal is to make storing and finding data fast and efficient.

Types of Hash Functions:

There are various types of hash functions designed to work with numeric or alphanumeric keys. This section highlights and explains some of the commonly used hash functions, including:

- Division Method
- Multiplication Method
- Mid-Square Method
- Folding Method

1. Division Method

The division method is one of the simplest and most widely used hash functions. In this method, the key is divided by a number (usually a prime number), and the remainder of this division is taken as the hash value or index. The formula used to calculate a hash function is given below:

$$h(k) = k \bmod m$$

- h(k) is the hash value (index),
- k is the key (input value),
- m is the size of the hash table (preferably a prime number).

Example:

Let's say the key is 29 and the table size m is 7

$$h(29) = 29 \bmod 7$$

$$= 1$$

So, the data with key 29 will be stored at index 1 in the hash table.

2. Multiplication Method

The multiplication method is another common hash function that works by multiplying the key with a constant value and using the fractional part of the result to determine the hash index. The formula used to calculate a hash function is given below:

$$hash(key) = \lfloor table\ size \times (Key \times A \mod 1) \rfloor$$

- key is the input value
- A is a constant between 0 and 1 (commonly used value: 0.618033, the fractional part of the golden ratio),
- mod 1 means only the fractional part of the multiplication is used,
- floor function takes the greatest integer less than or equal to the result.

Example:

Let us consider a key value of 123, a hash table size of 10, and a constant A = 0.618.

1. First, multiply the key with the constant:

   123×0.618=76.014123

2. Next, extract the fractional part of the result:

   Fractional part of 76.014=0.014

3. Multiply the fractional part by the table size:

   0.014×10=0.14

4. Apply the floor function to get the final hash value:

   floor(0.14)=0

So, the key 123 will be stored at index 0 in the hash table.

3. Folding Method

The folding method is a simple and commonly used hash function in which the key is divided into equal parts, and these parts are added together. The resulting sum is then reduced to an index within the hash table using the modulo operation with the table size m.

Steps in Folding Method:

1. Divide the key into parts of equal length (except possibly the last part).
2. Add all the parts together to get a total.
3. Take modulo m (size of the hash table) of the total sum to get the final hash index.

Example :

Let's say we are given the key:

Key = 123456

Hash table size: m = 10

Step 1 : Divide the key into equal parts

Split the key into 2-digit parts:

123456 → 12, 34, 56

Step 2 : Add the parts

12 + 34 + 56 = 102

Step 3 : Apply modulo operation

102 mod 10 = 2

Final Result :

The key 123456 will be stored at index 2 in the hash table.

4. Mid-Square Method

The Mid-Square Method is a hashing technique that generates a hash value by squaring the key and then extracting a fixed number of digits from the middle of the result. This method is particularly useful when the keys are uniformly distributed and of similar size.

Steps :

1. Square the key.
2. Extract the middle digits from the squared value (the number of digits to extract depends on the table size).
3. Apply modulo operation with the table size to get the index.

Example :

Let the key be:

Key = 123

Step 1: Square the key

123 × 123 = 15129

Step 2: Extract the middle digits

From 15129, the middle 3 digits are 512

Step 3: Apply modulo with table size

If the hash table size is 100, then : 512 mod 100 = 12

Final Result:

The key 123 will be stored at index 12 in the hash table using the Mid-Square Method.

4.1.4 Collision in Hashing

In hashing, a hash function is used to generate a unique index for each key so that it can be stored efficiently in a hash table. However, in some cases, two or more different keys may generate the same hash value. This situation is known as a collision.

Example of Collision :

Let's consider a simple hash function:

hash(key) = key % 10

This hash function returns the remainder when the key is divided by 10. So, it maps all keys to a range of values between 0 and 9 (assuming the hash table has 10 slots).

Now, let's take two different keys:

- Key 1: 25
  hash(25) = 25 % 10 = 5
- Key 2: 35
  hash(35) = 35 % 10 = 5

Although 25 and 35 are different keys, they both produce the same hash value 5. This means both would be placed at the same index (5) in the hash table.

4.1.5 Collision Resolution Techniques

To handle collisions, we use special strategies called collision resolution techniques. These techniques allow us to store multiple keys that hash to the same location in a way that maintains the efficiency of data storage and retrieval.

There are two main categories of Collision Resolution Techniques:

1. Chaining
2. Open Addressing

4.1.5.1 Chaining

In a hash table, each index is connected to a linked list as in Fig 4.1.2. When multiple keys hash to the same index, their corresponding values are added to the linked list at that location. This approach ensures that, even when collisions occur, all the values can still be retrieved by going through the linked list at that index.

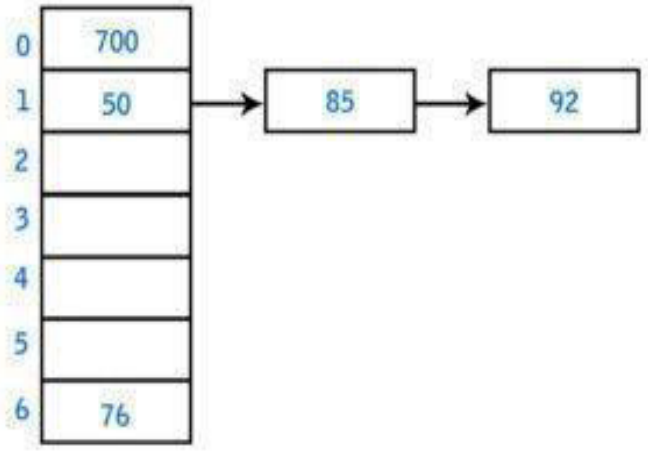


Fig. 4.1.2 Chaining

When a single slot contains multiple elements such as {50, 85, 92}, a linked list is used to store the extra items {85, 92} at that index. Using the chaining method makes inserting or deleting elements in a hash table relatively easy and efficient. However, since chaining relies on linked lists, it also shares their advantages and limitations. In some cases, dynamic arrays can be used instead of linked lists for implementing chaining.

The performance of hashing with chaining depends on the load factor (α). If the hash function distributes keys uniformly, the expected time complexity for search operations in chaining is approximately:

$$Expected\ Time \approx o(1+\alpha)$$

Where $\alpha = \frac{n}{m}$ represents the load factor of the hash table.

When the load factor is small, most operations such as insertion, deletion, and searching can be performed in constant time O(1). However, as the load factor increases, the linked lists at each index may become longer, which increases the search time.

4.1.5.2 Open Addressing

In open addressing, entries are stored directly in the array rather than using linked lists. The hash value does not directly determine the final location of an item. Instead, the algorithm starts checking from the hashed index and uses a probing sequence to find the next available empty slot. This sequence determines how the search moves through the array, often with varying gaps between checks. This method is also known as Closed hashing, and it handles collisions using three main techniques.

- Linear Probing
- Quadratic Probing
- Double-Hashing

1. Linear Probing

Linear probing involves checking the hash table in a step-by-step manner starting from the hashed index as in Fig 4.1.3. If the desired spot is already taken, the next available slot is searched. In this method, the gap between each probe is usually constant, commonly set to 1.

Formula:

index = key % hashTableSize

Linear Probing Example

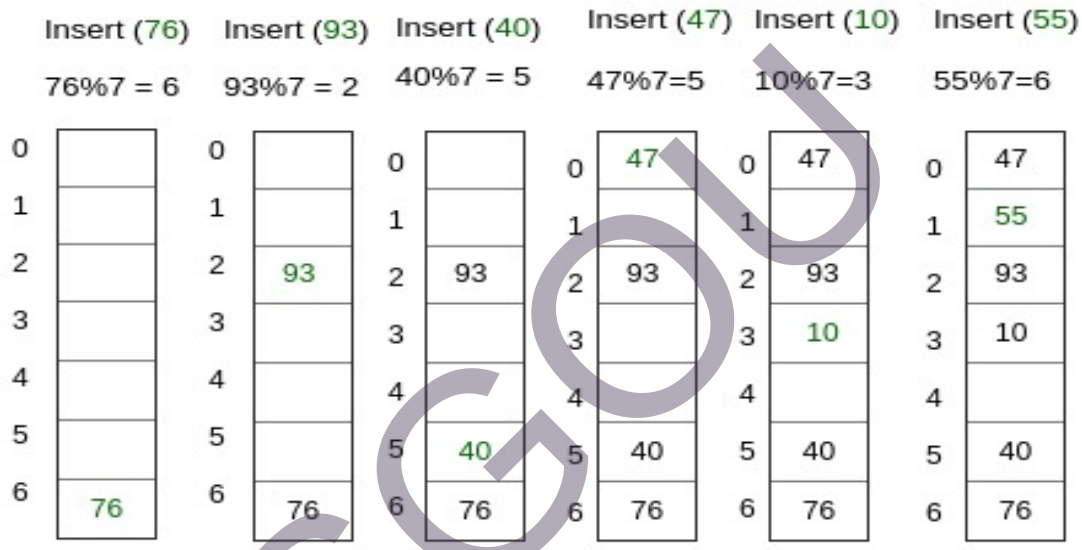


Fig 4.1.3 Linear Probing

2. Quadratic Probing

The main difference between linear and quadratic probing lies in the spacing between the slots checked during a collision. In quadratic probing, if a collision occurs, the system continues searching for an empty slot, but the distance between successive probes increases according to a polynomial pattern. Instead of checking the next consecutive slot as in linear probing, quadratic probing checks positions by adding squared values to the original hash index as shown in Fig 4.1.7.

Quadratic probing is an open-addressing collision resolution technique in which the i^{th} probe checks the slot at a distance of i^2 from the original hash location. This method helps reduce clustering problems that commonly occur in linear probing.

If *hash* (x) is the index generated using the hash function and S is the size of the hash table, then the probing sequence is calculated as follows:

Formula :

$$\text{Index} = (\text{hash}(x)+i^2)\% S$$

where:

- hash (x) : hash value of the key
- i : probe number (0, 1, 2, 3, …)
- S : size of the hash table

Working of Quadratic Probing

- First, the slot hash (x) % S is checked.
- If the slot is occupied, the next slot checked is:

  $(\text{hash}(x)+1^2) \% S$
- If that position is also occupied, the next probe becomes:

  $(\text{hash}(x)+2^2) \% S$
- If another collision occurs, the next position checked is:

  $(\text{hash}(x)+3^2) \% S$
- This process continues by increasing the square value until an empty slot is found.

Thus, the probe sequence follows the pattern:

$\text{hash}(x), \text{hash}(x) + 1^2, \text{hash}(x) + 2^2 + \text{hash}(x) + 3^2, ...$

Example :

Let us consider a hash table of size 7 and a simple hash function:

$h(x) = x \bmod 7$

Let the sequence of keys be: 22, 30, 50

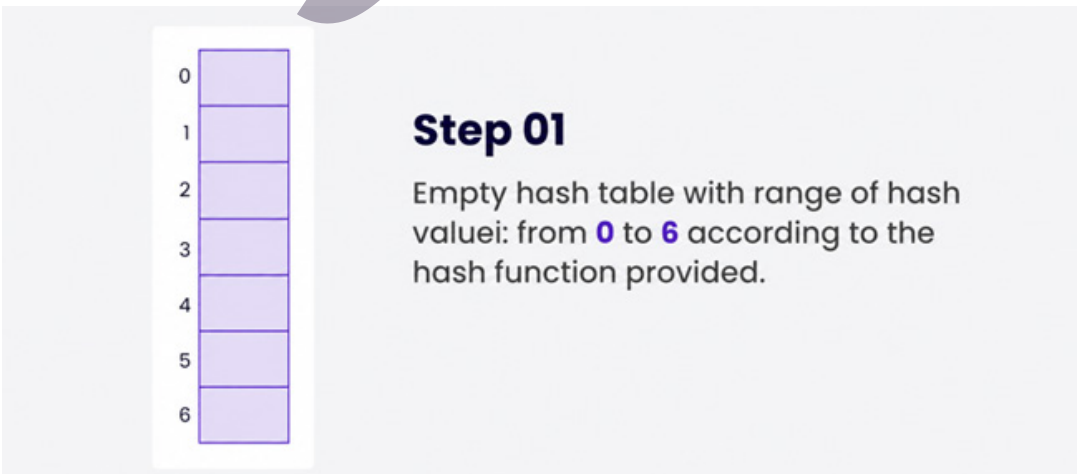


Fig. 4.1.4 empty hash table

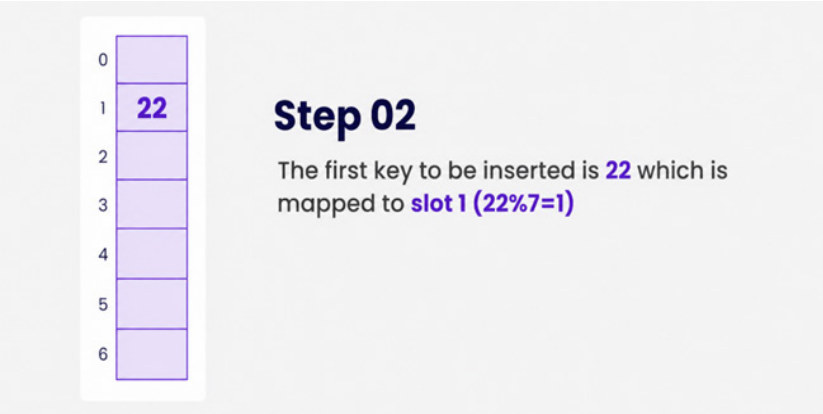


Fig. 4.1.5 insertion of 22

Fig. 4.1.6 insertion of 30

Fig. 4.1.7 insertion of 50

3. Double-Hashing

Double hashing is an open-addressing collision resolution technique in which a second hash function is used to determine the interval between probes. When a collision occurs, instead of checking the next slot sequentially, the position of the next probe is calculated using another hash value. This helps reduce clustering more effectively than linear probing and quadratic probing.

In double hashing, two hash functions are used:

- The first hash function determines the initial index.
- The second hash function determines the step size for probing.

The probing sequence is calculated using the formula:

$$(\text{First Hash (key)} + i \times \text{Second Hash (key)}) \ \% \ \text{Table Size}$$

where:

- First Hash(key) : gives the initial hash index
- SecondHash(key) : gives the probe interval or step size
- i : probe number (0, 1, 2, 3, …)
- Table Size : size of the hash table

Working of Double Hashing

In double hashing, the initial position of a key is first calculated using the primary hash function. If the calculated position is already occupied, a second hash function is used to determine the interval or step size for the next probe. The new position is then calculated using both hash functions and the probe number. If another collision occurs, the process is repeated by increasing the probe number until an empty slot is found in the hash table. This method distributes keys more uniformly and reduces clustering effectively.

Example:

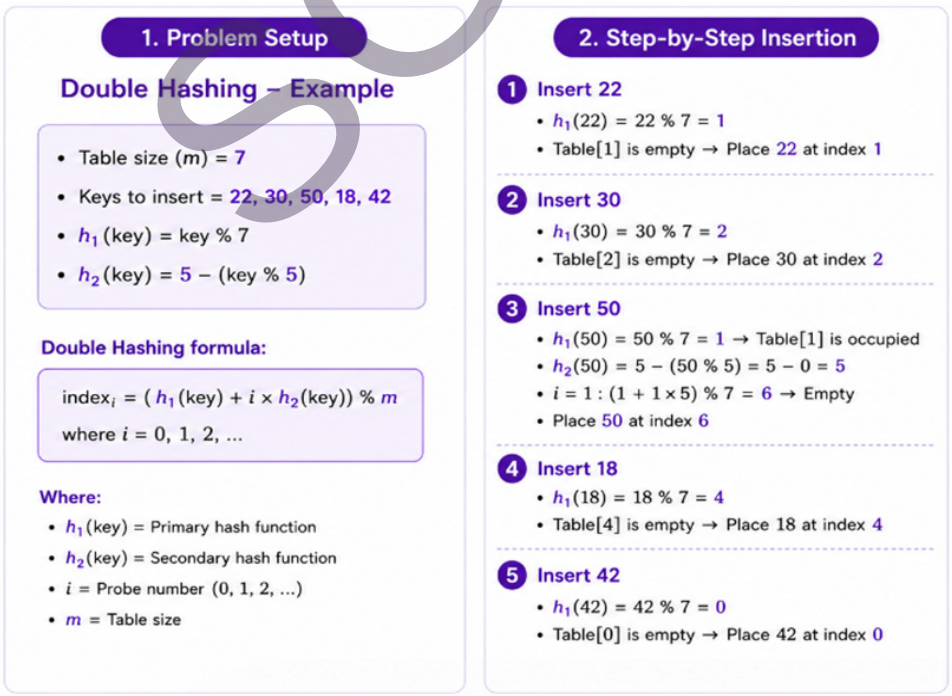


Fig. 4.1.8 Illustration of steps of double hashing

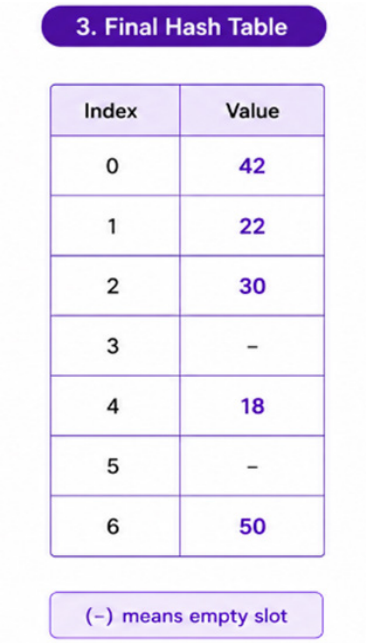


Fig. 4.1.9 Final Hash table

4.1.6 Hash table searching

Hash table searching is the process of retrieving data (a key or a value) from a hash table using its hash function. The main idea is that a hash function maps a given key to a specific index in the hash table. When we want to search for a key, we apply the same hash function and go directly to that index.

Steps Involved in Hash Table Searching:

1. Apply the hash function to the key.
2. Get the index from the hash value.
3. Check if the key is present at that index.
4. If there's a collision resolution technique used (like chaining or probing), check all possible locations.
5. If the key is found, return the data (success); if not, return "not found".

Example : With Linear Probing (Open Addressing)

Let's say we have a hash table of size 10:

Hash Function:

hash(key) = key % 10

Inserting Keys:

Let's insert these keys: 10, 20, 30, 21

1. 10 % 10 = 0 → index 0
2. 20 % 10 = 0 → index 0 (collision! Linear probing → try next index → 1)
3. 30 % 10 = 0 → index 0 (collision! next → 1 (already used) → 2)
4. 21 % 10 = 1 → index 1 (already used) → next index → 2 → already used → next → 3

So the hash table looks like this:

Index	Value
0	10
1	20
2	30
3	21

Searching for Key 30:

1. 30 % 10 = 0 → Go to index 0 → 10 ≠ 30
2. Try index 1 → 20 ≠ 30
3. Try index 2 → 30 , Found!

So key 30 is found at index 2.

What If the Key Is Not Found?

Let's search for 40:

- 40 % 10 = 0 → index 0 → check 10, 20, 30, 21 → not equal to 40
- Stop when we find an empty cell or loop back to the original position → "Key not found."

Summarised Overview

Hashing is a fast method for storing and retrieving data by using a key and a hash function to produce an index where the data is stored in a hash table. The key is a unique identifier, the hash function is a mathematical formula that converts the key into an index, and the hash table is the data structure that stores the data at that index. Common hash function methods include the division method, where the index is the remainder when the key is divided by the table size (often a prime number), the multiplication method, where the key is multiplied by a constant between 0 and 1, the fractional part is taken and multiplied by the table size, the folding method, where the key is split into equal parts, added, and then reduced using modulo, and the mid-square method, where the key is squared, the middle digits are extracted, and modulo is applied. A collision occurs when two keys produce the same index. Collisions can be resolved using chaining, which stores multiple elements in a linked list at the same index, or open addressing, which finds another slot using techniques such as linear probing, quadratic probing, or double hashing. Searching in a hash table involves applying the hash function to the key, checking the calculated index, and following the collision handling method until the key is found or an empty slot is reached.

Assignments

1. Define hashing and explain its main components — key, hash function, and hash table with suitable examples
2. Explain the working of Division Method, Multiplication Method, Folding Method, and Mid-Square Method in hashing with examples.
3. What is a collision in hashing? Explain with an example and state why collisions occur.
4. Describe chaining and open addressing as collision resolution techniques. How do they differ?
5. Explain linear probing, quadratic probing, and double hashing with suitable examples.
6. Describe the steps involved in hash table searching and explain the process using linear probing.

Reference

1. https://onlinecourses.nptel.ac.in/noc25_cs81/preview
2. https://www.geeksforgeeks.org/dsa/hashing-p-structure/

Suggested Reading

1. Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.
2. Samanta, D. "Classic data structures." *Terminology* 2 (2001): 1.
3. 3.Levitin, Anany. *Introduction to design and analysis of algorithms, 2/E*. Pearson Education India, 2008.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT

Algorithm Analysis

Learning Outcomes

After completion of this unit, the learner will be able to:

- recall and define basic asymptotic notations such as Big-O, Big-Ω, and Big-Θ
- explain the importance of time and space complexity in evaluating algorithm efficiency
- differentiate between best, worst, and average case complexities with suitable illustrations
- apply asymptotic notations to analyze the time complexity of simple algorithms in different cases

Background

In algorithm analysis, students need to study topics such as time complexity, space complexity, and asymptotic notations like Big-O, Big-Ω, and Big-Θ to measure the efficiency of algorithms. They should also learn how to analyze the best, worst, and average cases to understand performance under different conditions. Studying growth rates of functions helps compare how algorithms scale as input size increases. It is also important to practice step count analysis and recurrence relations for recursive algorithms. We need to study algorithm analysis because it helps in selecting the most efficient algorithm for solving real-world problems. It also allows programmers to optimize resources like time and memory. Without analysis, an algorithm may work correctly but perform poorly for large inputs. Overall, algorithm analysis ensures that solutions are not just correct, but also efficient and scalable.

Keywords

Time Complexity, Space Complexity, Asymptotic Notations, Big-O Notation, Best Case, Worst Case, Average Case

Discussion

4.2.1 Characteristics of algorithms

An algorithm is a finite sequence of instructions designed to perform a specific task. The main characteristics of an algorithm are as follows:

Input : An algorithm can have zero or more inputs, which are the data items provided before execution begins.

Output : The execution of an algorithm should produce at least one output, representing the result of processing the given input.

Definiteness : Every step in the algorithm must be clearly defined and free from ambiguity

Finiteness : The algorithm must complete its execution in a limited number of steps. Effectiveness: Each instruction in the algorithm must be practical and capable of performing a meaningful operation.

4.2.2 Reasons for analyzing algorithms

A single problem can have multiple possible algorithms. To determine which one is the most efficient, we need to analyze them. This analysis involves comparing the time an algorithm takes to run (time complexity) and the memory it requires (space complexity). When evaluating time complexity, we focus on the number of significant operations performed by the algorithm.

The time taken by an algorithm to finish its execution is known as its time complexity. It is a theoretical measure that estimates the growth rate of the algorithm, denoted as T(n), for large input sizes (n). Typically, time complexity is expressed using Big-O notation.

4.2.3 Asymptotic Notations

Asymptotic notations are mathematical tools used to describe the efficiency of algorithms in terms of their growth rate as the input size increases. They provide a way to express the running time or space requirement without depending on machine-specific constants or minor details. By using asymptotic notations, we can compare algorithms objectively and determine which one will perform better for large inputs.

They help in analyzing scalability, efficiency, and resource utilization. Overall, these notations are essential for understanding the long-term behavior of algorithms and for making informed decisions in algorithm design and selection.

To convey complex or lengthy information clearly and precisely, symbols or signs are used to represent quantities, concepts, or ideas in a standardized and concise form. This representation is called notation. For analyzing the performance of algorithms, we rely on asymptotic notation.

Asymptotic Notations act as mathematical tools that help us study an algorithm's running time by focusing on its growth behavior as the input size becomes very large. The main types of Asymptotic Notations are:

1. Big-O (O-notation) – Represents the asymptotic upper bound.
2. Big-Omega (Ω-notation) – Represents the asymptotic lower bound.
3. Big-Theta (Θ-notation) – Represents the asymptotic tight bound.

Omega notation gives the lower bound, representing the best-case performance. Theta notation defines the tight bound, meaning it gives both the upper and lower limits for an algorithm.

When analyzing algorithms, it is important to focus on their performance for large input sizes. An algorithm is generally considered more efficient if its worst-case runtime grows at a slower rate. Algorithms can be categorized into three groups based on their growth rates:

1. Algorithms that grow at least as fast as a given function.
2. Algorithms that do not grow faster than a certain function.
3. Algorithms that grow at exactly the same rate as a given function.

These categories are represented using Big-Omega Ω(g(n)), Big-O O(g(n)), and Big-Theta Θ(g(n)) notations, which will be explained in detail shortly.

4.2.3.1 Big-O notation

Big-O notation is a key concept in computer science used to describe the time or space complexity of algorithms. It defines the upper bound of an algorithm's performance, indicating the maximum resources it might require in terms of execution time or memory. Instead of providing exact measurements, Big-O captures the asymptotic behavior of an algorithm, showing how its growth rate changes as the input size increases. This makes it possible to compare the efficiency of different algorithms or data structures. Big-O essentially estimates the maximum time or space an algorithm may consume, typically considering the worst-case scenario. It is denoted as O(f(n)), where f(n) represents the approximate number of steps or operations needed to solve a problem of size n. By using Big-O notation, programmers can analyze scalability and make informed decisions about algorithm selection.

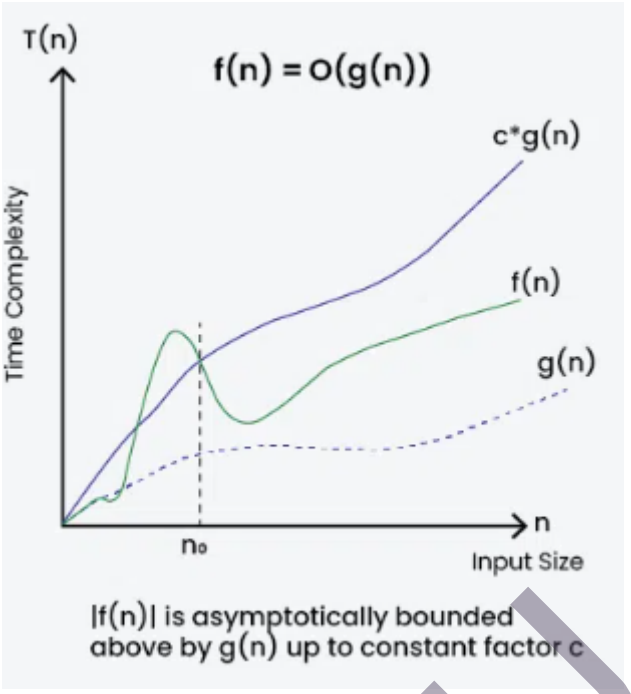


Fig. 4.2.1 Big O Analysis

Big-O Conditions

For two functions, f(n) and g(n), we say that f(n) is O(g(n)) if there exist constants c > 0 and $n_0 \geq 0$ such that for all $n \geq n_0$, the inequality $f(n) \leq c \times g(n)$ holds.

In simpler terms, f(n) is O(g(n)) when its growth rate does not exceed a constant multiple of g(n) for sufficiently large values of n, where both c and n_0 are fixed constants.

Importance of Big-O Notation

Big-O notation is a mathematical method used to establish an upper bound on the time or memory that an algorithm or data structure might need. It provides a standardized approach to assess and compare the efficiency of different algorithms and to estimate how they will perform as the size of the input increases.

Its significance lies in the following points:

- Helps assess the efficiency of algorithms in terms of time and space usage.
- Describes how the execution time or memory requirements scale with input size.
- Enables comparison of multiple algorithms to select the most suitable one for a given task.
- Assists in evaluating the scalability of algorithms for larger datasets
- Guides developers in optimizing code for better overall performance.

A Quick Method to Determine Big-O of an Expression

To quickly identify the Big-O notation of an expression:

- Disregard the lower-order terms and focus only on the term with the highest degree.
- Omit any constant multiplier attached to that highest-order term.

Examples

Example 1: For $f(n) = 3n^2 + 2n + 1000 \log n + 5000$

- After removing the lower-order terms, the dominant term is $3n^2$.
- Ignoring the constant factor 3 leaves n^2.
- Hence, the Big-O notation for this expression is $O(n^2)$.

Example 2: For $f(n) = 3n^3 + 2n^2 + 5n + 1$

- The dominant term is $3n^3$.
- The growth rate is cubic (n^3).
- Therefore, the Big-O notation is $O(n^3)$.

Properties of Big-O Notation

Big-O notation follows several key properties:

1. **Reflexivity**
   Any function is Big-O of itself.
   Example: If $f(n) = n^2$, then $f(n) = O(n^2)$.

2. **Transitivity**
   If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
   Example: For $f(n) = n^2$, $g(n) = n^3$, and $h(n) = n^4 \rightarrow f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$.

3. **Constant Factor Rule**
   Multiplying a function by a constant does not change its Big-O classification.
   Example: If $f(n) = n$ and $g(n) = n^2$, then $f(n) = O(g(n))$; hence, $2f(n) = O(g(n))$.

4. **Sum Rule**
   When adding two functions, the one with the larger growth rate dominates.
   Example: If $f(n) = n^2$ and $h(n) = n^3$, then $f(n) + h(n) = n^2 + n^3 = O(n^3)$.

5. **Product Rule**
   Multiplying two functions multiplies their growth rates.
   Example: If $f(n) = n$, $g(n) = n^2$,
   then $f(n) \times g(n) = n \times n^2 = O(n^3)$.

6. **Composition Rule**
   If f(n) = O(g(n)) and x = O(y), then f(x) = O(f(y)), provided that f is an increasing function
   Example: If $f(n) = n^2$ and $g(n) = n^3$ then $f(g(n)) = (n^3)^2 = n^6 = O(n^6)$.

Common Big-O Notations

Big-O notation is used to assess the time and memory requirements of an algorithm, with a particular focus on the worst-case scenario. It defines an upper bound on complexity, allowing us to understand how an algorithm's performance scales as the input size grows. Some common types of time complexities are listed below:

1. **Linear Time Complexity – O(n)**

   In linear time complexity, the execution time of the algorithm increases directly in proportion to the size of the input.

2. **Logarithmic Time Complexity – O(log n)**

   In logarithmic time complexity, the execution time increases in proportion to the logarithm of the input size, meaning the growth rate slows significantly as the input becomes larger.

3. **Quadratic Time Complexity – $O(n^2)$**

   Quadratic time complexity indicates that the algorithm's execution time grows in proportion to the square of the input size, often occurring in algorithms with nested loops over the data.

4. **Cubic Time Complexity – $O(n^3)$**

   Cubic time complexity signifies that the execution time of an algorithm increases in proportion to the cube of the input size, commonly seen in algorithms with three levels of nested loops.

5. **Polynomial Time Complexity – $O(n^k)$**

   Polynomial time complexity describes algorithms whose execution time can be expressed as a polynomial function of the input size n. In Big O notation, it is represented as $O(n^k)$, where k is a constant indicating the degree of the polynomial. Algorithms in this category are often seen as efficient because their growth rate remains manageable as input size increases. Examples include O(n) (linear), $O(n^2)$ (quadratic), and $O(n^3)$ (cubic) complexities.

6. **Exponential Time Complexity – $O(2^n)$**

   Exponential time complexity indicates that the algorithm's runtime doubles with each additional element in the input, leading to very rapid growth and poor scalability for large datasets.

7. **Factorial Time Complexity – O(n!)**

   Factorial time complexity describes algorithms whose execution time increases at a factorial rate relative to the input size. This type of complexity is typically found in algorithms that enumerate all possible permutations of a dataset.

4.2.3.2 Big-Theta (Θ)

In algorithm analysis, asymptotic notations help evaluate an algorithm's efficiency by describing its precise growth rate. One of these notations is Big-Theta (Θ), denoted by the Greek letter Θ.

Definition :

Let **f(n)** and **g(n)** be functions mapping natural numbers to natural numbers. We say that **f(n) is Θ(g(n))** if there exist positive constants c_1 and c_2, and a natural number n_0, such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0$$

In other words, for sufficiently large values of n, the function f(n) is bounded both above and below by constant multiples of g(n). This means f(n) grows at the same rate as g(n) asymptotically.

Mathematical Definition of Big-Theta (Θ)

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0,\ n_0 \in \mathbb{N} \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0\}$$

> **Note:** Θ(g(n)) represents a set of functions.

In simpler terms, a function $f(n)$ belongs to **Θ(g(n))** if, for all sufficiently large values of n, its growth is **bounded both above and below** by constant multiples of $g(n)$. In other words, there exist positive constants c_1 and c_2 such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0$$

Additionally, $f(n)$ must be **non-negative** for all $n \geq n_0$.

See fig.4.2.2

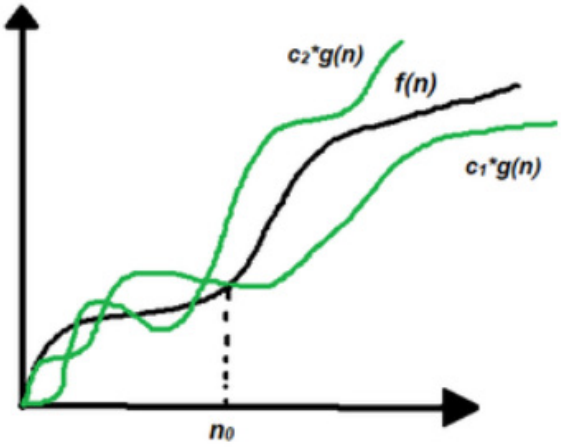


Fig. 4.2.2 Big-Theta (Θ) Graphical Representation

In simple terms, Big-Theta (Θ) notation defines both the upper and lower asymptotic bounds of a function f(n), representing the average time complexity of an algorithm. To calculate the average time complexity of a program, follow these steps:

- Divide the program into smaller, manageable sections.
- Identify all types and numbers of inputs, then determine how many operations each input requires for execution. Make sure the input cases are uniformly distributed.
- Sum all the computed operation counts and divide the total by the number of inputs.
- If the resulting function of nnn is g(n)g(n)g(n) after ignoring constant factors, it is expressed in Θ notation as:

  Θ(g(n))

In a linear search example, assume all cases occur with equal probability, including the scenario when the key is not present in the array.

This involves adding the time taken for each case (key found at positions 1, 2, 3, …, n, and the absent case), then dividing the total by n+1n + 1n+1.

Average case time complexity:

Average Case Time Complexity:

$$\frac{\sum_{i=1}^{n+1}\Theta(i)}{n+1}$$

$$\Rightarrow \frac{\Theta\big((n+1)\times(n+2)/2\big)}{n+1}$$

$$\Rightarrow \Theta\left(1+\frac{n}{2}\right)$$

$$\Rightarrow \Theta(n) \quad \text{(after removing constants)}$$

Thus, the **average-case time complexity** of a linear search is **Θ(n)**.

So, the average case complexity for linear search is Θ(n).

4.2.3.3 Big-Omega notation(Ω)

In algorithm analysis, asymptotic notations help evaluate an algorithm's efficiency in both best-case and worst-case situations. Among these, the Big-Omega notation, represented by the Greek letter (Ω).

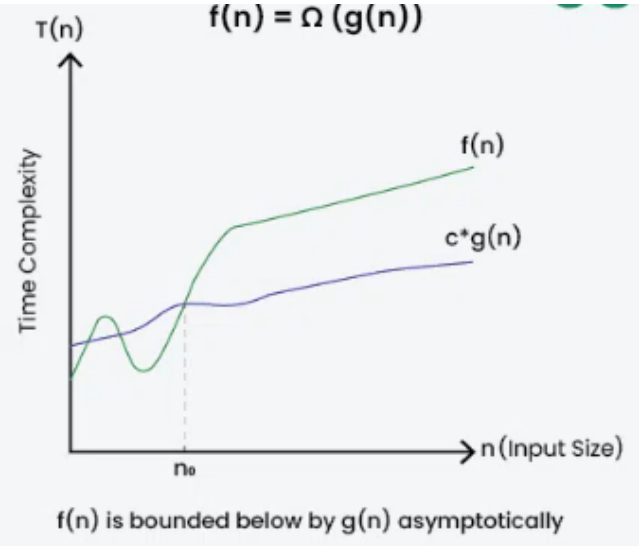


Fig. 4.2.3 Big-Omega notation(Ω)

Big-Omega (Ω) notation is used to represent the asymptotic lower bound of an algorithm's time complexity, focusing on its best-case performance. It establishes the minimum time an algorithm will take to run, based on the size of the input. This notation is expressed as Ω(f(n)), where f(n) indicates the number of steps an algorithm requires to process an input of size n. f(n) is bounded below by g(n) asymptotically.

In simpler terms, Big-Omega (Ω) is applied when we want to show that an algorithm requires at least a certain amount of time or space, regardless of other conditions. See fig. 4.2.3.

How to Identify Big-Omega (Ω) Notation?

In simple terms, Big-Omega (Ω) notation represents the asymptotic lower bound of a function f(n). It describes the minimum growth rate of the function as the input size approaches infinity.

Steps to Find Big-Omega (Ω) Notation:

1. **Divide the program into parts :** Split the algorithm into smaller sections where each section has a definite runtime complexity.

2. **Evaluate each section's complexity :** Determine the number of operations in each part (based on input size), assuming the program runs in the least possible time.

3. **Combine all complexities :** Add together the operation counts from each section and simplify the result, calling it f(n).

4. **Eliminate constants :** Discard constant factors and identify the term with the lowest order, or another function always less than f(n) as n grows infinitely.

If this smaller-order function is g(n), then the Big-Omega (Ω) of f(n) is written as Ω(g(n)).

4.2.4 Cases to consider during analysis

Let us again take our pizza baking example. Let us consider your mother and your friend's mother who are going to prepare pizza. Suppose your mother is very systematic and keeps every ingredient ready before preparing the pizza. In the case of your mother, the performance will be the best if we have all the ingredients ready and everything well arranged, as it will be easy for the preparation. Let us imagine that your friend's mother is not very systematic. In this case, even if we have all the ingredients ready and everything well arranged, she might not be able to perform well. However, there might be a cooking style that she is used to and she can prepare well. This situation might not be favorable for your mother. i. e the preparation is highly dependent on the kind of input(here ingredients) that you have provided.

In the same manner, the performance of an algorithm has significant impact based on the inputs considered when analyzing an algorithm. For example, if an input list is almost in the sorted order, some sorting algorithms will perform well, while some other sorting algorithms will perform poorly. But, the opposite would be the condition if the list is randomly arranged instead of sorted. Hence, while analyzing an algorithm, multiple input sets must be considered.

Following are the cases to be considered during analysis:

1. Best Case Input
2. Worst Case Input
3. Average Case Input

4.2.4.1 Best case Input

Consider the example of your mother preparing a pizza where all the ingredients are arranged neatly and kept in order. If she needs to find the salt, she can quickly locate it because everything is properly organized. In this situation, she will take the minimum amount of time to cook.

Similarly, take an example of an algorithm that searches for a number within a group of numbers. If the required number is found in the very first position of the list, the search takes the least possible time. Such an input is called the best-case input. In other words, best-case input refers to the input set that enables the algorithm to run most efficiently. This happens because the execution takes minimum time, as the algorithm performs the least amount of work.

4.2.4.2 Worst case Input

Consider the example of your mother preparing a pizza where the required ingredients are not arranged properly and are scattered around. If she needs to find salt, since it is not placed in order, she cannot locate it quickly and will have to search the entire kitchen for it. In such a case, the preparation will take the maximum amount of time.

Similarly, take the case of an algorithm searching for a number in a list. If the desired number happens to be the last element in the list, then the search will consume the maximum amount of time. This type of input is referred to as the worst-case input. In other words, the worst-case input denotes the set of inputs that makes the algorithm run the slowest. This occurs because such an input forces the algorithm to spend the maximum time and effort during execution.

4.2.4.3 Average case Input

Consider the example of your mother making a pizza, where certain ingredients are already available while others are missing. In such a situation, she will take longer than her normal pace since she may need to ask you to buy the missing items, organize the available ones, and so on. Hence, the preparation will take an average amount of time.

Similarly, the average case input represents the set of inputs that make an algorithm perform at its average efficiency

4.2.5 Algorithmic Complexity

Some criteria are required to evaluate the efficiency of an algorithm. Evaluating the efficiency of algorithms allows us to compare them. The two primary measures of an algorithm's efficiency are the time it takes and the space it consumes

4.2.5.1 Time complexity

The time taken to finish a task plays an important role. For example, in cooking, if all ingredients are pre-arranged and the mother follows a straightforward method, the required time will be very less. On the other hand, if the ingredients are not organized and she uses a complicated approach, the cooking will take much longer.

Time complexity refers to the duration required by an algorithm to execute, expressed as a function of the input size. It evaluates the time needed to carry out each instruction of the algorithm's code.

4.2.5.2 Space complexity

The right proportion of ingredients is essential in cooking. In the same way, the memory consumed by each variable plays a vital role in an algorithm.

The space complexity of an algorithm or program refers to the total memory space needed to handle an instance of a computational problem, expressed as a function of the input characteristics. It represents the memory an algorithm requires until its complete execution.

4.2.6 Estimating complexity

4.2.6.1 Time for an algorithm to run T(n)

Imagine a case where you are given a set of numbers that are not arranged in order. Consider the following numbers placed in a random sequence.

9 4 6 2 5 3

To arrange the above list in order, different techniques can be applied. Before analyzing the cases, we will represent the numbers using n.

9	4	6	2	5	3
n1	n2	n3	n4	n5	n6

Let us consider the following case:

Case 1:

In Insertion Sort, the key element is compared with the elements before it. If those elements are larger than the key element, they are shifted one position forward. A simple example can be used to illustrate this process:

[9 4 6 2 5 3]

Step 1:

Key = 4

The key is compared with the previous element. i.e key is compared with 9.

Since 9>4 , move the element 9 to the next position and insert 'key' to the previous position.

Result : [4 9 6 2 5 3]

Step 2:

4	9	6	2	5	3

Key = 6; 9> 6 , move 9 to the next position and insert key to the previous position

Result : [4 6 9 2 5 3]

Step 3:

4	6	9	2	5	3

Key = 2

9>2→ [4 6 2 9 5 3]
6>2 →[4 2 6 9 5 3]
4>2 →[2 4 6 9 5 3]

Result : [2 4 6 9 5 3]

Step 4:

2	4	6	9	5	3

Key = 5

9 > 5 → [2 4 6 5 9 3]

6 > 5 → [2 4 5 6 9 3]

4 > 5 ≠ →Stop

Result : [2 4 5 6 9 3]

Step 5:

2	4	5	6	9	3

Key = 2

9>3 →[2 4 5 6 3 9]

6>3 →[2 4 5 3 6 9]

5>3 →[2 4 3 5 6 9]

4>3 →[2 3 4 5 6 9]

2>3 ≠→ Stop

Result : [2 3 4 5 6 9]

Now let us take the example of insertion sort. First, we will look into the best case analysis: In insertion sort, two major operations are carried out:

1. Scanning the list and comparing each pair of elements.
2. Swapping elements whenever they are not in order.

As explained earlier, the best case means the least time the algorithm requires for 'n' inputs. This happens when the array is already arranged in sorted order. Hence, in the best case, insertion sort executes in O(n) time.

Next, let us move to the Worst and Average Case Analysis:

The worst case for insertion sort arises when the input list is in strictly decreasing order. For insertion sort with an input list arranged in descending order, the following operations are performed:

1. Scanning through the list, comparing each pair of elements → (1+2+....+n−2+n−1) scans

2. Swaps elements → (1+2+....+n−2+n−1) swaps.

   i.e it takes 2*(1+2+....+n−2+n−1) operations to perform insertion sort.

   →2*(n-1*(n-1+1))/2

   =n*n-1→ $O(n^2)$

   Therefore, the worst case complexity is $O(n^2)$.

While analyzing algorithms, the average case frequently exhibits the same complexity as the worst case. Hence, on average, insertion sort requires $O(n^2)$. We will explore the process of estimating complexity in detail shortly.

4.2.6.2 Big Oh- O-notation(Asymptotic Upper Bound) - Worst-case

O-notation expresses the worst-case complexity of an algorithm. It emphasizes how the runtime of an algorithm increases as the input size grows.

Definition

If $f(n)$ denotes the runtime of an algorithm and $g(n)$ represents a related complexity function, then $f(n)$ is $O(g(n))$ if there exist real constants c (where $c > 0$) and n_0 such that:

$f(n) \leq c \cdot g(n)$ for all input sizes n (where $n \geq n_0$).

Consider the following functions:
f(n) = 3n + 2
g(n) = n

We want to show that f(n) = O(g(n)).
f(n) = 3n + 2
Relating f(n) to g(n), $T(n) = 3n + 2 \leq 3n + n \leq 4n$
Determine constants: c = 4 and $n_0 = 2$
Thus, for $n \geq 2$, $3n + 2 \leq 4n$.

Worst-case Complexity: Big-O notation illustrates the worst-case scenario of an algorithm's execution. In this case, as n becomes very large, the linear term 3n dominates the constant 2. Hence, the complexity simplifies to O(n), meaning the runtime grows linearly with input size.

Constants: The constants c and n_0 confirm that the inequality holds for large enough n. Here, c = 4 and $n_0 = 2$ prove that 3n + 2 is bounded above by 4n.

By using Big-O notation, the time complexity is represented as: 3n + 2 = O(n)

4.2.6.3 Ω-notation(Asymptotic Lower Bound) - Best case

Ω-notation expresses the best-case complexity of an algorithm. It emphasizes how the execution time behaves in the most favorable scenario as the input size increases.

If f(n) represents the running time of an algorithm and g(n) is a function of time complexity to compare with, then f(n) = Ω(g(n)) if there exist real constants c (c > 0) and n_0 (n_0 > 0) such that f(n) ≥ c × g(n) for all n ≥ n_0.

Consider the functions:

f(n) = 3n + 2
g(n) = n

To express *f*(n) as Ω(*g*(n)), the condition *f*(n) ≥ c × *g*(n) must be satisfied for all c > 0 and n_0 ≥ 1.

f(n) ≥ c × g(n)

⇒ 3n + 2 ≥ c × n
For n ≥ 1, 3n + 2 ≥ 3n. Thus, 3n acts as the lower bound of 3n + 2.

Choosing constants:
c = 3
n_0 = 1

Therefore, for n ≥ 1, 3n + 2 ≥ 3n, which is always TRUE for c = 3 and n ≥ 1.

Best-case complexity : Big-Omega notation describes the best-case growth of an algorithm. In this example, as n becomes large, the linear term 3n overshadows the constant 2. Hence, the complexity simplifies to Ω(n), showing that runtime increases at least linearly with input size.

Role of constants : The values c and n_0 are chosen to prove that the inequality holds for sufficiently large inputs. Here, c = 3 and n_0 = 1 confirm that 3n + 2 is bounded below by 3n.

Thus, using Big-Omega notation, the time complexity can be expressed as:
3n + 2 = Ω(n)

4.2.6.4 Θ-notation(Asymptotic Tight Bounds) - Average-case

Θ-notation is applied to study the average-case complexity of an algorithm. It provides a tight bound on execution time by setting both an upper and lower limit, showing that the runtime grows asymptotically as a specific function of the input size for both the best and worst cases.

If f(n) represents the run time of an algorithm and g(n) represents a comparable function, then f(n) is Θ(g(n)) if there exist real constants c1, c2, and n0 (where c1>0, c2>0, n0>0) such that:

$c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$ for every input size n ($n \geq n0$).

Example:

$f(n) = 3n + 2$

$g(n) = n$

To show $f(n) = \Theta(g(n))$, we need constants c1, c2, and n0 such that:
$c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$

So, $c1 \cdot n \leq 3n + 2 \leq c2 \cdot n$
This inequality holds true when $c1 = 1$, $c2 = 4$, and $n \geq 2$.

In general, if constants c1, c2, and n0 exist ($c1 > 0$, $c2 > 0$, $n0 > 0$) such that:
$c1 \cdot g(n) < f(n) < c2 \cdot g(n)$ for all $n > n0$,
then $f(n)$ is $\Theta(g(n))$, meaning it is both $O(g(n))$ and $\Omega(g(n))$.

- Lower Bound: $3n + 2 \geq 3n$, hence $c1 = 3$.
- Upper Bound: $3n + 2 \leq 3n + 2n \leq 5n$, hence $c2 = 5$.

Thus, for constants $c1 = 3$, $c2 = 5$, and $n0 = 1$, the function $3n + 2$ is bounded above and below by linear functions of n.

Average-case Complexity: Big-Theta notation indicates that the runtime grows linearly with input size in the average case, regardless of minor fluctuations.

Therefore, using Big-Theta notation, we represent the time complexity as: $3n + 2 = \Theta(n)$

Following table 4.1.1 shows the main difference between the three notations.

Table 4.1.1 Difference between O, Ω and Θ notations

Big oh (O)	**Big Omega (Ω)**	**Big Theta (Θ)**
Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
Big-O is a measure of the longest amount of time it could possibly take for the algorithm to complete.	Big- Ω takes a small amount of time as compared to Big-O it could possibly take for the algorithm to complete.	Big- Θ take very short amount of time as compare to Big-O and Big- Ω it could possibly take for the algorithm to complete.

Summarised Overview

An algorithm is a finite sequence of clear and effective instructions designed to solve a specific problem, and it is defined by key characteristics such as input, output, definiteness, finiteness, and effectiveness. Since multiple algorithms may exist for the same task, analyzing them is essential to identify the most efficient one. Algorithm analysis primarily focuses on time complexity, which measures how execution time grows with input size, and space complexity, which evaluates memory usage. To express efficiency in a machine-independent way, asymptotic notations are used. The three most common notations are Big-O, which defines the upper bound and represents the worst-case scenario, Big-Omega, which defines the lower bound and represents the best-case scenario, and Big-Theta, which gives a tight bound and represents the average case. Common time complexities include constant O(1), logarithmic O(log n), linear O(n), quadratic $O(n^2)$, exponential $O(2^n)$, and factorial O(n!), each describing different growth rates. For instance, linear search has an average case of $\Theta(n)$, while insertion sort shows a best case of O(n), a worst case of $O(n^2)$, and an average case of $O(n^2)$. Considering best, worst, and average cases allows a fair comparison of algorithms. Overall, asymptotic analysis helps in predicting scalability, comparing performance, and selecting the most efficient algorithm for larger input sizes.

Assignments

1. Explain with suitable examples how the five essential characteristics of an algorithm (input, output, definiteness, finiteness, and effectiveness) ensure correctness and reliability in problem-solving.

2. Given two algorithms solving the same problem, one with a time complexity of O(n log n) and another with $O(n^2)$, analyze which algorithm would be more efficient for small inputs and for very large inputs. Justify your reasoning.

3. Compare Big-O, Big-Omega, and Big-Theta notations with respect to their purpose, bounds, and use cases. Provide one example function for each notation and explain why it fits that category.

4. Determine the Big-O notation of the following function and explain the reasoning step by step:

   $f(n) = 4n^3 + 20n^2 + 6n + 50$

5. Arrange the following complexities in increasing order of growth: O(n), $O(2^n)$, $O(n^2)$, O(log n), $O(n^3)$, O(n!). Justify your order by analyzing their scalability for large input sizes.

6. Using linear search as an example, analyze the differences between best-case, worst-case, and average-case time complexities. Which of these cases is most important for evaluating an algorithm in real-world applications, and why?

7. Analyze the time complexity of insertion sort for best-case, worst-case, and average-case scenarios. Support your answer with an explanation of the operations performed in each case.

8. Explain with examples how the following properties of Big-O notation are useful in algorithm analysis:
   a. Reflexivity
   b. Transitivity
   c. Sum Rule
   d. Product Rule

Reference

1. https://onlinecourses.nptel.ac.in/noc25_cs81/preview
2. https://www.geeksforgeeks.org/dsa/

Suggested Reading

1. Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.
2. Samanta, D. "Classic data structures." *Terminology* 2 (2001): 1.
3. Levitin, Anany. *Introduction to design and analysis of algorithms, 2/E*. Pearson Education India, 2008.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3
UNIT

Algorithm Design Approaches

Learning Outcomes

After completion of this unit, the learner will be able to:

- understand the principles of Divide and Conquer and Greedy strategies with suitable examples
- apply Binary Search, Quick Sort, and the Knapsack problem to demonstrate the working of these approaches
- compare the efficiency and limitations of Divide and Conquer and Greedy methods in solving computational problems
- analyze problem characteristics to determine whether Divide and Conquer or Greedy is the appropriate strategy
- implement and evaluate algorithmic solutions using these approaches in real-world scenarios

Background

Have you ever wondered how Google shows you the most relevant search results in a fraction of a second, or how your phone quickly finds a contact among thousands of names? Behind these everyday technologies lies the power of algorithms. An algorithm is not just a computer program; it is a step-by-step procedure designed to solve a problem or perform a task efficiently. Just as we follow a recipe to prepare a dish, computers follow algorithms to process data and produce meaningful results. To understand algorithms better, imagine a library with thousands of books. If you want to find a specific title, you could check each book one by one, but that would take hours. Instead, if the books are arranged alphabetically, you can jump to the middle shelf, compare, and then decide whether to search the left or right half. This smart way of searching is exactly how the Divide and Conquer strategy

works, and Binary Search is a classic example. Similarly, when sorting items like exam papers, efficient algorithms such as Quick Sort apply the same principle by dividing, solving, and combining.

On the other hand, many real-life decisions are made step by step by choosing the best immediate option. For example, if you are filling a backpack for a trip and want to maximize the space, you might first pack the largest useful items, then medium ones, and finally the smaller ones until it is full. This is similar to the Greedy approach, where we make the best local choice at each step. The Knapsack Problem illustrates this concept: Greedy works perfectly for the Fractional Knapsack but fails for the 0/1 Knapsack, which requires more advanced methods such as Dynamic Programming. With these algorithmic strategies, students gain the ability to choose the right problem-solving approach based on the situation. The Divide and Conquer method trains us to think recursively and design efficient searching and sorting techniques, while the Greedy method teaches us how to optimize step by step in real-world scenarios. Together, they build a strong foundation in algorithmic thinking, enabling the design of efficient, robust, and scalable solutions that are at the heart of computer science and modern technology.

Keywords

Algorithm Design, Problem-Solving Strategies, Divide and Conquer, Binary Search, Quick Sort, Recursion, Knapsack Problem, Fractional Knapsack, 0/1 Knapsack, Optimal Solution, Dynamic Programming

Discussion

4.3.1 Algorithm design approaches

In computer science, an algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output. It is a sequence of computational steps that transform the input into the output.

The study of algorithms goes beyond simply writing a piece of code. It involves a deeper understanding of how to approach a problem and formulate a step-by-step solution. Algorithm design approaches are systematic methods for creating efficient and effective algorithms. These paradigms provide a framework for thinking about problem-solving and are crucial for developing robust software solutions.

4.3.2 Common algorithm design approaches

1. Brute Force

The brute force approach is the simplest way to solve a problem. It involves trying every possible solution and selecting the one that works. Although this technique is easy to understand and implement, it can be very inefficient for large problems because it examines all possibilities. Despite this limitation, brute force is useful as a starting point and helps to compare the performance of more advanced algorithms.

2. Divide and Conquer

Divide and conquer is a powerful strategy in which a problem is broken into smaller, more manageable subproblems. Each subproblem is solved independently, and the solutions are combined to solve the original problem. This method works very well for problems that have a recursive structure. Examples include merge sort, which sorts an array by dividing it into halves and merging the sorted parts, and binary search, which efficiently finds an element in a sorted list.

3. Dynamic Programming

Dynamic programming is an optimization technique used for problems that can be broken into overlapping subproblems. Instead of solving the same subproblem multiple times, the results are stored in a table to avoid repetition. This approach is especially effective for problems such as the knapsack problem, calculating Fibonacci numbers, and certain shortest path problems in weighted graphs.

4. Greedy Algorithms

Greedy algorithms build a solution step by step by making the best choice available at each step, hoping that the sequence of local decisions leads to the best overall solution. These algorithms are usually simple and fast, but they do not always guarantee the best possible solution. Examples include Dijkstra's algorithm for finding the shortest path in a graph and the greedy method for constructing a minimum spanning tree in network design.

5. Backtracking

Backtracking is a systematic method for exploring all possible solutions by building them gradually. If a candidate solution is found to be invalid or unsuitable, it is abandoned immediately, and the algorithm returns to try other options. This method is widely used in constraint satisfaction problems such as the N-Queens problem, Sudoku, and other puzzles or combinatorial challenges.

6. Randomized Algorithms

Randomized algorithms use random choices in their logic to improve efficiency or provide probabilistic guarantees. By incorporating randomness, these algorithms can

often solve problems faster or more simply than deterministic methods. Examples include randomized quicksort, which selects a random pivot to avoid the worst-case scenario, and the Monte Carlo method, which uses random sampling to estimate values such as pi or probabilities in simulations.

Two widely used approaches, Divide and Conquer and Greedy algorithms, are explained in this unit, each with its own principles, suitable problem types, and applications

4.3.3 Divide and Conquer strategy

The Divide and Conquer strategy is a powerful algorithmic paradigm that solves problems by breaking them down into smaller, more manageable subproblems. These subproblems are then solved independently, and their solutions are combined to solve the original problem. This approach is highly effective for a wide range of problems, from sorting and searching to computational geometry. The strategy has three main steps:

Divide : Split the problem into smaller subproblems of the same type.

Conquer : Solve each subproblem recursively. If the subproblem is small enough, solve it directly.

Combine : Merge the solutions of the subproblems to form the solution of the original problem.

Binary search and Quick Sort are another well-known example of the Divide and Conquer approach.

4.3.3.1 Binary Search

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half. It is a classic example of the Divide and Conquer strategy, though it's often simpler to think of it as a form of recursive reduction.

- Divide : The algorithm's core principle is to find the middle element of the sorted array and compare it with the target value. This comparison effectively divides the array into two halves.
- Conquer :
  - If the target value matches the middle element, the search is successful.
  - If the target value is less than the middle element, the algorithm discards the right half of the array and continues the search in the left half.
  - If the target value is greater than the middle element, it discards the left half and searches in the right half.
- Combine: In the case of binary search, the "combine" step is implicit. The successful "conquer" step of finding the element is the solution. The problem is reduced at each step until the element is found or the search space is exhausted.

Algorithm (Binary Search):

1. Let low = 0 and high = n-1.
2. While low ≤ high:
   - Find mid = (low + high) / 2.
   - If arr[mid] == target, return mid.
   - If arr[mid] > target, search in the left half (high = mid - 1).
   - Else search in the right half (low = mid + 1).
3. If the element is not found, return -1.

   Example: Searching for the number 25 in the sorted array [2, 8, 15, 25, 30, 42, 50].

   1. Divide : The middle element is 25.
   2. Conquer : The target is 25, which is equal to the middle element. The search is successful, and the index is returned.

Complexity: Binary search has a time complexity of O(log n), where n is the number of elements in the array. This logarithmic complexity makes it extremely fast for large datasets. Its space complexity is O(1) for the iterative approach and O(log n) for the recursive approach due to the call stack.

4.3.3.2 Quick Sort

Quick Sort is an efficient, in-place, comparison-based sorting algorithm. It's often considered one of the fastest general-purpose sorting algorithms in practice. It is a prime example of the Divide and Conquer paradigm.

- **Divide :** The algorithm first selects an element from the array, called the pivot. It then rearranges the other elements so that all elements smaller than the pivot come before it, and all elements greater than the pivot come after it. This operation is called partitioning. The array is now divided into two sub-arrays, one with elements less than the pivot and one with elements greater than the pivot.
- **Conquer :** The two sub-arrays are then recursively sorted by calling Quick Sort on each.
- **Combine :** The "combine" step is trivial in Quick Sort. The partitioning step places the pivot in its correct sorted position. Once the two sub-arrays are sorted, the entire array is sorted. No extra work is needed to merge them.

Algorithm (Quick Sort):

1. If the array has 0 or 1 elements, return (already sorted).
2. Select a pivot element (commonly the first, last, or a random element

3. Partition the array: place elements smaller than pivot on the left and greater elements on the right.

4. Recursively apply Quick Sort on left and right subarrays.

5. Combine the results.

Example : Sorting the array [10, 7, 8, 9, 1, 5].

1. Divide : Choose the last element, 5, as the pivot. After partitioning, the array might look like [1, 5, 8, 9, 10, 7].

2. Conquer : Recursively sort the left sub-array [1] and the right sub-array [8, 9, 10, 7].

3. Combine : The sorted sub-arrays, along with the pivot, form the fully sorted array.

Complexity :

- **Worst-case time complexity :** $O(n^2)$. This occurs when the pivot selection consistently results in a highly imbalanced partition (e.g., the array is already sorted and the first or last element is chosen as the pivot).
- **Best and average-case time complexity :** O(n log n). This is because, on average, the partitioning step divides the array into roughly equal-sized sub-arrays, similar to a binary tree.
- **Space complexity :** O(log n) due to the recursive call stack. In the worst case, it can be O(n). Quick Sort is an in-place algorithm, but the recursion requires space on the stack.

Table 4.3.1 Comparison of Binary Search and Quick Sort in Divide and Conquer

Feature	**Binary Search**	**Quick Sort**
Problem Type	Searching	Sorting
Input Requirement	Sorted array	Unsorted array
Combine Step	Not required	Merging partitions
Time Complexity	O(logn)	O(nlogn)(average)
Space Complexity	O(1) (iterative)	O(logn)

4.3.4 Greedy approach

The Greedy approach is an algorithm design technique where decisions are made step by step, choosing the option that looks the best at the moment (locally optimal) with the hope of reaching a globally optimal solution.

- It builds up a solution piece by piece.
- At each step, it makes the greedy choice, the one that offers the most immediate benefit.
- Greedy algorithms do not always guarantee the optimal solution for every problem, but they work very well for problems that have the Greedy Choice Property and Optimal Substructure.
- Greedy Choice Property : A global optimum can be arrived at by choosing a local optimum.

Optimal Substructure: A problem has optimal substructure if an optimal solution of the whole problem contains optimal solutions of its subproblems.

The Knapsack Problem is one of the classical problems that can be solved using the Greedy approach under certain conditions.

4.3.4.1 Knapsack problem

The Knapsack Problem is a classic optimization problem that asks you to determine the most valuable collection of items to pack into a knapsack with a limited weight capacity. The greedy approach is a simple, intuitive method that can solve a specific variation of this problem.

Problem Definition

A thief has a knapsack (bag) with a weight capacity W. He has n items, each with:

- Weight (wi)
- Profit (pi)

The goal is to maximize the total profit without exceeding the knapsack's capacity.

There are two main types of Knapsack problems:

- 0/1 Knapsack Problem: You can either take an item entirely or leave it entirely; you cannot take a fraction of an item.
- Fractional Knapsack Problem: You are allowed to take fractions of items.

The greedy approach works perfectly for the Fractional Knapsack Problem but does not guarantee an optimal solution for the 0/1 Knapsack Problem.

- Greedy Approach for Fractional Knapsack

Choose items based on the profit-to-weight ratio (pi/wi).

- Pick the item with the highest profit-to-weight ratio first.
- If the item fits completely, put it in the knapsack.
- If not, take the fraction that fits.
- Continue until the knapsack is full.

Algorithm (Fractional Knapsack)

1. Input: weights and profits of n items, knapsack capacity W.
2. Compute ratio pi/wi for each item.
3. Sort all items in decreasing order of pi/wi.
4. Initialize total_profit = 0.
5. For each item in the sorted list:
   - If the item fits completely (wi ≤ remaining capacity):
     - Add its profit to total_profit.
     - Reduce knapsack capacity.
   - Else take fraction of item:
     - total_profit += (remaining_capacity / wi) * pi
     - Knapsack is now full → stop.
6. Output total_profit.

   Example : Knapsack capacity W=50

Table 4.3.2 Item details

Item	Weight (wi)	Profit (pi)	Ratio (pi/wi)
1	10	60	6.0
2	20	100	5.0
3	30	120	4.0

Steps:

- Sort items by ratio → Item 1 (6.0), Item 2 (5.0), Item 3 (4.0).
- Take Item 1 fully → Profit = 60, Remaining capacity = 40.
- Take Item 2 fully → Profit = 60 + 100 = 160, Remaining capacity = 20.
- Take 20/30 fraction of Item 3 → Profit = 160 + (20/30 × 120) = 160 + 80 = 240.

Maximum Profit = 240

Why Greedy Fails for the 0/1 Knapsack Problem

The greedy approach fails for the 0/1 Knapsack problem because taking the item with the highest immediate value-to-weight ratio may prevent you from taking other items that would result in a greater total value. The greedy choice is not always globally optimal in this scenario.

Example : Same knapsack (W=50) but for the 0/1 Knapsack Problem.

Table 4.3.3 Item details

Item	Value ($)	Weight (kg)	Value-to-Weight Ratio ($/kg)
A	60	10	6.0
B	100	20	5.0
C	120	30	4.0

Greedy Approach:

- Take Item A (ratio 6.0). Remaining capacity: 40. Total value: 60.
- Take Item B (ratio 5.0). Remaining capacity: 20. Total value: 160.
- Item C (ratio 4.0) does not fit (30 kg > 20 kg).
- Final greedy value: $160.

Optimal Solution:

- A better solution is to take items B and C.
- Total weight: 20+30=50 kg.
- Total value: 100+120=220.
- This is a higher value than the greedy result.

The greedy approach, in this case, made a locally optimal choice (taking item A first) that led to a suboptimal overall solution. This is because it could not accommodate the combination of items B and C, which collectively provided a higher value. The 0/1 Knapsack Problem is typically solved using dynamic programming.

Applications of Knapsack Problem

- Resource allocation (maximize profit under budget constraints).
- Cargo loading.
- Investment decisions (choosing projects with limited funds).
- Task scheduling with limited time slots.

Limitations of Greedy in Knapsack

- Greedy gives an optimal solution for Fractional Knapsack.
- For 0/1 Knapsack, Greedy does not always give the correct result → requires Dynamic Programming.

Summarised Overview

The discussion focuses on fundamental algorithm design approaches that form the backbone of problem solving in computer science. Algorithms are more than just pieces of code; they are systematic procedures for transforming input into output through a series of logical steps. Several paradigms are introduced, each offering a unique way to tackle problems efficiently. The Divide and Conquer strategy is explained as a powerful method that breaks a complex problem into smaller, manageable subproblems, solves them independently, and then combines the results, with Binary Search and Quick Sort presented as classic examples to show its effectiveness in searching and sorting tasks. The Greedy approach is then discussed as another important paradigm that builds solutions incrementally by choosing the locally optimal option at each stage, aiming for a globally optimal outcome. To demonstrate this, the Knapsack Problem is studied in detail, where Greedy gives the best solution for the Fractional Knapsack but may fail in the 0 1 Knapsack, which highlights the boundaries of this method and the need for alternative approaches such as Dynamic Programming. Together, these strategies illustrate how the choice of algorithmic approach depends on the problem structure and constraints, reinforcing the importance of selecting the right paradigm to design efficient, robust, and scalable solutions in computer science.

Assignments

1. Discuss the common algorithm design approaches.
2. What are the main steps involved in the Divide and Conquer approach?
3. Discuss the working principle of Binary Search and Quick Sort with examples.
4. Explain the working of the Greedy approach with reference to the Knapsack Problem
5. Differentiate between the Fractional Knapsack and the 0 1 Knapsack.
6. Discuss the importance of selecting the right algorithm design strategy. Use Divide and Conquer and Greedy approaches as case studies to highlight your answer.

Reference

1. Skiena, S. S. (2020). *The algorithm design manual* (3rd ed.). Springer.

2. Karumanchi, N. (2018). *Algorithm design techniques: Recursion, backtracking, greedy, divide and conquer, and dynamic programming* (18th ed.). CareerMonk Publications.

3. Kleinberg, J., & Tardos, É. (2006). *Algorithm design*. Pearson Education.

4. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). *Algorithms*. McGraw-Hill.

Suggested Reading

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

2. https://www.geeksforgeeks.org/dsa/algorithms-design-techniques/

3. https://www.numberanalytics.com/blog/greedy-algorithm-divide-and-conquer-approach

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4
UNIT

Dynamic Programming

Learning Outcomes

After completion of this unit, the learner will be able to:

- understand and apply the principles of Dynamic Programming to solve optimization problems such as Fibonacci sequence and shortest path algorithms
- analyze and implement the Floyd–Warshall algorithm for finding all-pairs shortest paths in weighted graphs
- explain the concept of Backtracking and apply it to constraint satisfaction problems
- solve and trace the N-Queens problem using backtracking techniques

Background

Algorithm design is at the heart of computer science, and different strategies are used to make problem-solving more efficient. Among these, Dynamic Programming (DP) and Backtracking are two powerful approaches that address problems in different but complementary ways.

Dynamic Programming provides a systematic method of solving problems by breaking them into smaller overlapping subproblems and storing their results to avoid redundant calculations. This makes DP especially suitable for optimization problems, such as finding the shortest path, resource allocation, and sequence alignment. Classic problems like the Fibonacci sequence, Knapsack, Longest Common Subsequence, and Matrix Chain Multiplication highlight how DP reduces exponential complexity to polynomial time.

Backtracking, on the other hand, is a trial-and-error technique that explores possible solutions by building them step by step, abandoning a path as soon as it violates the problem's constraints. It is highly effective in puzzles, combinatorial generation, and constraint-based challenges. The N-Queens problem, Sudoku solving, and graph coloring are prime examples of backtracking in action.

Together, these techniques demonstrate two different philosophies: DP focuses on reusing computed solutions to improve efficiency, while Backtracking emphasizes exploring possibilities and pruning invalid paths. Both approaches equip learners with essential tools to handle real-world challenges, from network routing to artificial intelligence, optimization, and game design.

Keywords

Nodes, Traversal, Temporary Pointer, Dynamic Memory, Adjacency List, Playlists, Undo-Redo, Polynomials

Discussion

4.4.1 Dynamic Programming

Dynamic Programming (DP) is an algorithmic problem-solving technique that provides an efficient way to tackle complex problems by decomposing them into a collection of simpler subproblems. Unlike a naive recursive approach, which repeatedly solves the same subproblems and leads to exponential time complexity, DP ensures that each subproblem is solved only once, and its solution is stored in a structured memory space such as an array, matrix, or dictionary for future use. This strategy prevents redundant calculations and significantly improves performance. DP is particularly powerful for problems that exhibit two key properties: Optimal Substructure and Overlapping Subproblems. Optimal Substructure means that the overall solution to a problem can be derived from the solutions of its smaller subproblems, making it possible to build a solution step by step. Overlapping Subproblems occur when the recursive solution revisits the same subproblems multiple times; instead of recalculating them, DP reuses the stored results. By combining these two principles, DP transforms problems with exponential complexity into polynomial-time solutions, making it an indispensable approach in data structures and algorithms for tasks such as sequence alignment, pathfinding, optimization, and resource allocation.

4.4.1.1 Steps to Solve a DP Problem

Dynamic Programming (DP) works on the idea of solving problems efficiently by breaking them into smaller subproblems, solving those subproblems once, and storing their results for reuse. The following steps define the foundation of DP:

1. Divide a Problem into Subproblems

The first step in applying DP is to express a complex problem as a combination of smaller, simpler subproblems. These subproblems are easier to solve and collectively contribute to the solution of the original problem. For example, to compute the Fibonacci number F(n), we reduce it into two smaller subproblems: F(n−1) and F(n−2) . Similarly, in the Shortest Path Problem, finding the shortest path between two cities can be broken down into finding the shortest path through intermediate cities. By dividing a problem into subproblems, we create a recursive structure where larger solutions depend on smaller ones.

2. Store (Memoize) Solutions to Subproblems

One of the main challenges in recursion is the repeated computation of the same subproblem, which leads to inefficiency. To overcome this, DP introduces memoization (top-down approach) or tabulation (bottom-up approach) to store the results of solved subproblems in a data structure such as an array, matrix, or hash table. Once a subproblem is solved, its result is saved in the memory space so that when it is needed again, the stored value can be retrieved instantly instead of recomputing it. This storage mechanism is the key factor that transforms inefficient recursive solutions into efficient DP algorithms.

3. Reuse the Stored Solutions to Avoid Recomputation

The real efficiency of DP comes from the ability to reuse previously computed subproblem results whenever required. When the algorithm encounters the same subproblem again, it directly fetches the answer from the storage instead of solving it again. For example, in the Fibonacci series, instead of recalculating F(3) multiple times, DP will compute it once and store it, ensuring that every future reference to F(3) is resolved in constant time. This eliminates redundant work and dramatically reduces the time complexity of the algorithm.

4.4.1.2 Properties of DP

DP is most effective when two properties exist in a problem: Overlapping Subproblems and Optimal Substructure.

- **Overlapping Subproblems** occur when the recursive approach solves the same subproblem multiple times. For instance, computing Fibonacci recursively generates a recursion tree where the same subproblem (like F(3)) appears several times.

- **Optimal Substructure** means that the solution to the overall problem can be constructed from solutions to its smaller subproblems. For example, the shortest path from A to C through B can be computed as the sum of the shortest path from A to B and the shortest path from Bto C.

Together, these two properties ensure that by dividing, storing, and reusing, DP provides optimal and efficient solutions to complex problems.

4.4.1.3 Approaches to Dynamic Programming

Dynamic Programming (DP) can be implemented in two main approaches: Top-Down (Memoization) and Bottom-Up (Tabulation). Both methods rely on the same principle of solving subproblems and reusing their results, but they differ in how the computation is carried out and stored.

1. Top-Down Approach (Memoization)

The Top-Down Approach, also called Memoization, begins with the original problem and recursively breaks it down into smaller subproblems. Whenever a subproblem is solved, its result is stored in a memory structure such as an array, list, or dictionary (map). If the same subproblem appears again, the algorithm retrieves the result from storage instead of recomputing it.

This approach combines the natural recursive structure of the problem with caching to avoid redundant calculations. It is easier to design because we can start from the problem definition and use recursion, making it intuitive for problems that are naturally recursive (such as Fibonacci, longest common subsequence, and knapsack).

Advantages

- Simpler to implement if the problem already has a recursive structure.
- Avoids unnecessary computations.
- Saves time compared to plain recursion.

Disadvantages

- Requires function call overhead due to recursion.
- May use more stack space, leading to possible stack overflow for very deep recursions.

2.Bottom-Up (Tabulation) Approach

The Bottom-Up approach, also known as Tabulation, is one of the two major methods to implement Dynamic Programming (DP). In this method, we iteratively construct a table (array, matrix, or higher-dimensional structure) that stores solutions to subproblems. Instead of solving problems recursively, the results are computed iteratively in a specific order, ensuring smaller subproblems are solved before larger ones. This eliminates redundant work and recursion overhead, making the approach efficient and predictable.

How it Works

1. Choosing the Data Layout

The first step is to decide how subproblems will be represented in memory. Depending on the recurrence relation, this could be a 1D array (for problems like Fibonacci), a 2D array (for Knapsack or Edit Distance), or a higher-dimensional table if the state has more variables. Each index in the table corresponds to a unique subproblem.

2. Initializing Base Cases

After deciding the data layout, the next step is to fill the base cases. These are the smallest subproblems whose answers are known directly without further computation. For example, in Fibonacci, F(0) = 0 and F(1) = 1 form the base values that allow the rest of the sequence to be calculated.

3. Deciding the Order of Computation

The key to Bottom-Up DP is choosing a correct order of evaluation. Subproblems must be solved in a sequence that ensures all required smaller states are available before computing a larger one. For Fibonacci, we compute from F(2) up to F(n), while in problems like Edit Distance, we fill the table row by row or column by column.

4. Filling the Table and Extracting the Answer

Once the base cases and computation order are set, the table is filled iteratively using the recurrence relation. Each cell holds the solution to a subproblem. The final answer is found in the last cell (or a target index) after the table is completed.

Advantages

- Eliminates recursion, so there is no risk of stack overflow.
- Generally faster in practice due to reduced function call overhead.
- Space optimization is easier using rolling arrays or compressed tables.
- Provides a predictable runtime, which is valuable in performance-sensitive systems.

Disadvantages

- Requires knowing a valid dependency order before implementation.
- May compute all subproblems, even those that are not necessary, leading to wasted effort.
- Needs extra boilerplate code for loops, array initialization, and boundary conditions.

4.4.1.4 Applications of Dynamic Programming

Dynamic Programming (DP) is widely used to solve optimization and decision-making problems where the solution can be built from smaller overlapping subproblems. Below are some of the most common applications:

1. Fibonacci Sequence

The Fibonacci sequence is the simplest and most common introductory example of DP. The recurrence relation is:

$F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1$

Using recursion without DP leads to exponential time complexity because the same subproblems are recalculated multiple times. With DP (either memoization or tabulation), we compute each Fibonacci number once, reducing complexity to O(n)

2. Knapsack Problem

The Knapsack family of problems demonstrates how DP can optimize resource allocation:

0/1 Knapsack: Each item can be chosen at most once. The DP table decides whether to include or exclude each item for every weight capacity.

Fractional Knapsack: Unlike 0/1 knapsack, items can be divided into fractions. This is not solved using DP but rather with a greedy algorithm. However, comparing the two helps in understanding how DP handles discrete optimization while greedy works on fractional cases.

The 0/1 knapsack problem is a classic DP problem solved in $O(n \cdot W)$, where is the number of items and is the capacity of the knapsack.

3. Longest Common Subsequence (LCS)

The LCS problem is used in text comparison, DNA sequence alignment, and version control systems. Given two sequences, the goal is to find the length of the longest subsequence present in both.

DP builds a 2D table where dp[i][j] represents the LCS length between the first i characters of one string and the first j characters of another. This avoids recomputation of overlapping subsequences, reducing complexity to $O(m \cdot n)$ where m and n are string lengths.

4. Matrix Chain Multiplication

The Matrix Chain Multiplication problem determines the most efficient way to multiply

a chain of matrices by minimizing the total number of scalar multiplications.

A naive recursive solution tries all possible parenthesizations, leading to exponential complexity. DP, however, stores solutions to subchains in a table, reducing the complexity to $O(n^3)$. This is a classic example of how DP optimizes computation-heavy problems.

5. Coin Change Problem

This problem asks: given coin denominations and a target sum, in how many ways can the sum be formed (or what is the minimum number of coins needed).

Using DP, we build solutions for smaller amounts and use them to construct solutions for larger amounts. The iterative DP solution has a time complexity of $O(n \cdot target)$, where n is the number of coin types.

6. Shortest Path Problems

DP is also applied in graph algorithms where we need to compute shortest paths:

Floyd-Warshall Algorithm: Computes shortest paths between all pairs of vertices using DP. The state dp[k][i][j] represents the shortest path from i to j using only the first k vertices as intermediates. Its complexity is $O(V^3)$.

Bellman-Ford Algorithm: Uses DP to compute the shortest path from a source to all other vertices in a graph, even with negative weight edges. Complexity is $O(V \cdot E)$.

7. Optimal Binary Search Tree (OBST)

The OBST problem arises in compiler design and database indexing. Given search probabilities for keys, the goal is to construct a binary search tree with minimum expected search cost.

DP computes the cost of all possible subtrees and stores them in a table to avoid recomputation, leading to a polynomial-time solution.

4.4.1.5 Advantages of Dynamic Programming

- Greatly reduces computation time compared to recursion by avoiding repeated evaluation of the same subproblems.
- Provides efficient solutions to exponential problems, converting them into polynomial-time solutions (e.g., Fibonacci, Knapsack, LCS).
- Ensures optimality by systematically exploring all subproblems and storing results.
- Widely applicable across computer science, including algorithms, operations research, AI, and bioinformatics.

4.4.1.6 Disadvantages of Dynamic Programming

- Requires extra space for memoization tables or tabulation arrays, which may be significant for large state spaces.

- Needs careful design to identify subproblems, recurrence relations, and base cases.
- Not always intuitive: converting a recursive problem into a DP formulation often requires deeper insight.
- May compute unnecessary states in tabulation, which is wasteful for sparse problems.

4.4.2 Fibonacci Sequence

The Fibonacci sequence is one of the simplest yet most powerful examples in computer science and mathematics. It is often the very first problem used to explain recursion and later to introduce dynamic programming (DP). The sequence has fascinated mathematicians, scientists, and artists for centuries because it also appears in real-world patterns. For example, the spiral shapes of sunflower seeds, pinecones, shells, and even the branching of trees follow Fibonacci numbers. In finance, traders sometimes use Fibonacci ratios in stock market analysis. In computer science, Fibonacci is a classic example to understand algorithm efficiency.

The Fibonacci sequence starts with two initial numbers:

- F(0) = 0
- F(1) = 1

From this point onward, each new number is obtained by adding the two previous numbers. Mathematically,

$F(n)=F(n-1)+F(n-2)$, for $n>1$

$F(n) = F(n-1) + F(n-2)$, for $n > 1$

$F(n)=F(n-1)+F(n-2)$, for $n>1$

This means:

- F(2) = F(1) + F(0) = 1 + 0 = 1
- F(3) = F(2) + F(1) = 1 + 1 = 2
- F(4) = F(3) + F(2) = 2 + 1 = 3
- F(5) = F(4) + F(3) = 3 + 2 = 5

So, the first few numbers of the sequence are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Each number grows larger but in a pattern that depends only on the previous two.

4.4.2.1 Fibonacci and Recursion

The Fibonacci sequence is one of the most famous mathematical series, defined as F(n)=F(n−1) + F(n−2) for n>1, with base values F(0) = 0 and F(1) =1. This produces the sequence 0, 1, 1, 2, 3, 5, 8, 13, and so on. The sequence has importance not only in mathematics but also in computer science, biology, and finance. It is often used as an introductory problem to learn about recursion because the definition itself naturally suggests a recursive solution.

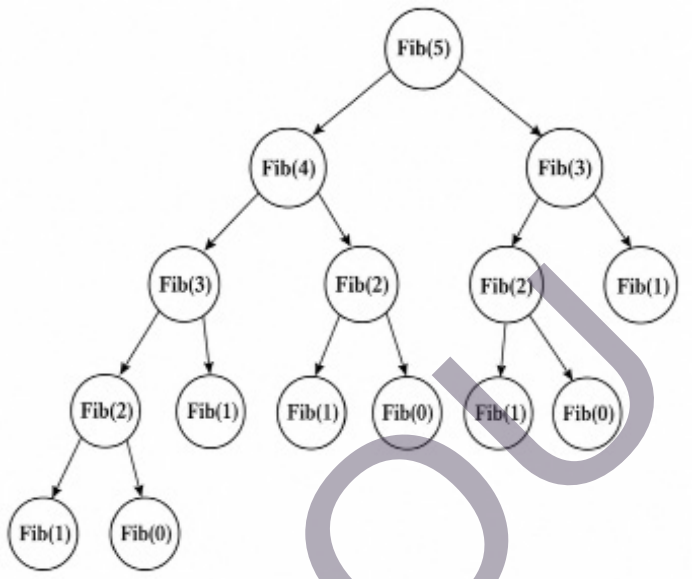


Fig. 4.4.1 Recursion Tree for fib(5)

In C programming, recursion means that a function calls itself to solve a smaller version of the same problem. For Fibonacci, the recursive function can be written such that if the value of nnn is 0 or 1, it simply returns nnn, otherwise it returns the sum of the two preceding Fibonacci numbers by making recursive calls to fib(n-1) and fib(n-2). For example, if we call fib(5), the function will break it into fib(4) + fib(3). To calculate fib(4), it again breaks into fib(3) + fib(2). Similarly, fib(3) further expands into fib(2) + fib(1). This process continues until the base cases are reached.

If we visualize this process, it forms a tree-like structure of calls. At the root we have fib(5), which branches into fib(4) and fib(3). Each of these branches again divides into two more calls. While this approach looks elegant and directly follows the mathematical definition, it has a major drawback: many calculations are repeated. For instance, to compute fib(5), the value of fib (3) is calculated multiple times and fib(2) is calculated even more frequently. As the value of n grows, these repeated calculations increase rapidly, making the program extremely slow. The time complexity of this recursive method is exponential, approximately $O(2^n)$, which means the number of operations doubles with each increase in n. For large inputs, this becomes impractical. Moreover, recursion consumes memory on the function call stack. For very large values of n, this can even cause a stack overflow error.

To overcome this problem, iterative methods or dynamic programming techniques are

used. Instead of recalculating values, the idea is to compute Fibonacci numbers step by step from the beginning and store previously computed results. In C, this can be done using a simple loop. We start with two variables, say a = 0 and b = 1, representing the first two Fibonacci numbers. Then, in each iteration, we calculate the next number as the sum of the previous two, update the variables, and continue. This method is very efficient because it computes each Fibonacci number exactly once and requires only constant space for storage. The time complexity of this iterative solution is linear, O(n), which is much faster than the exponential recursive approach.

4.4.2.2 Fibonacci series with Dynamic Programming

Dynamic Programming (DP) is an optimization technique used to solve problems that can be divided into smaller overlapping subproblems. Unlike simple recursion, where the same subproblems are solved repeatedly, DP stores the results of already solved subproblems and reuses them when needed. This avoids redundant work and makes the solution much faster. The core idea of DP can be summarized as: solve small problems first, store their answers, and then reuse those answers to build the solution for larger problems. The main idea of DP is:

1. Solve small problems first.
2. Store their answers.
3. Reuse them to solve bigger problems instead of calculating again.

For Fibonacci, we can use DP because the value of F(n) depends only on smaller values F(n-1) and F(n-2).

The Fibonacci sequence is a classic example that demonstrates the power of Dynamic Programming. In the recursive approach, to compute F(n), we call F(n−1) and F(n−2). But to compute F(n−1), the program again computes F(n−2) and F(n−3), and so on. This leads to repeated calculations of the same values. Dynamic Programming eliminates this inefficiency by storing the values of F(n−1) and F(n−2) once they are computed, and then directly reusing them whenever required.

There are two main ways to apply DP to the Fibonacci sequence: Top-Down (Memoization) and Bottom-Up (Tabulation). In the memoization approach, recursion is still used, but each computed Fibonacci number is stored in an array or dictionary. Whenever the same value is needed again, the function simply returns the stored result instead of recalculating it. This drastically reduces the number of recursive calls. On the other hand, in the tabulation approach, the problem is solved iteratively. We start with the base values F(0) = 0 and F(1) = 1, and then build up the sequence step by step until we reach F(n). This method avoids recursion completely and is often more efficient in terms of memory usage.

4.4.2.3 Methods to Solve Fibonacci with DP

The Fibonacci sequence is one of the most common examples to demonstrate how

Dynamic Programming (DP) improves efficiency compared to normal recursion. The key idea of DP is to break the problem into smaller overlapping subproblems, store their solutions, and reuse them. For Fibonacci, since each value F(n) depends on F(n-1) and F(n-2), it fits perfectly with DP.

There are three major methods to solve Fibonacci numbers using DP: Top-Down (Memoization), Bottom-Up (Tabulation), and Space-Optimized Approach. Let's explore them in detail.

1. Top-Down Approach (Memoization)

In the Top-Down approach, we use recursion with memory. Normally, recursion recalculates the same values many times, which makes it inefficient. Memoization solves this by storing already computed results in an array or dictionary. Whenever the same function is called again, instead of recalculating, we simply reuse the stored answer.

Example : To compute F(5), recursion will calculate F(4) and F(3). But F(4) again needs F(3), which means F(3) is recalculated multiple times. With memoization, once F(3) is calculated, it is stored. So, when needed again, it is fetched instantly, saving a lot of repeated work.

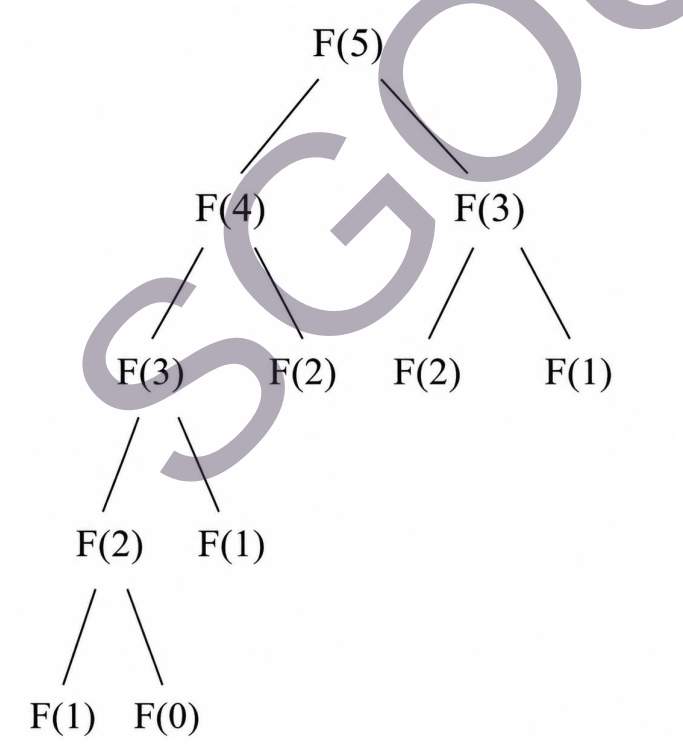


Fig. 4.4.2 Memoization Tree with Storage

(Here, F(3) and F(2) are stored after the first calculation and reused later.)

This reduces the time complexity from exponential ($O(2^n)$) to linear (O(n)), with an extra space cost of storing results.

2. Bottom-Up Approach (Tabulation)

Unlike recursion, the Bottom-Up approach builds the solution iteratively from the base cases. We already know:

- F(0) = 0
- F(1) = 1

From here, we compute higher values step by step until we reach F(n). Instead of breaking the problem down, we build it up using a table (array).

Example: Let's calculate F(6) step by step:

- F(0) = 0
- F(1) = 1
- F(2) = F(1) + F(0) = 1
- F(3) = F(2) + F(1) = 2
- F(4) = F(3) + F(2) = 3
- F(5) = F(4) + F(3) = 5
- F(6) = F(5) + F(4) = 8

So, the answer for F(6) is 8.

n	**0**	**1**	**2**	**3**	**4**	**5**	**6**
F(n)	0	1	1	2	3	5	8

This method completely avoids recursion and repeated work. Its time complexity is O(n), and the space complexity is also O(n) because we store all intermediate results.

3. Space-Optimized Approach

If we carefully observe, to calculate F(n) we only need the previous two values (F(n-1) and F(n-2)). That means we do not need to keep the entire table of values. Instead, we can store only the last two numbers and keep updating them in each step.

Example : To compute F(6):

- Start: prev2 = 0, prev1 = 1
- Next = prev1 + prev2 = 1 (F(2))
- Update: prev2 = 1, prev1 = 1
- Next = 2 (F(3)) → Update
- Next = 3 (F(4)) → Update
- Next = 5 (F(5)) → Update
- Next = 8 (F(6)) → Final Answer

Step 1: prev2=0, prev1=1 → next=1

Step 2: prev2=1, prev1=1 → next=2

Step 3: prev2=1, prev1=2 → next=3

Step 4: prev2=2, prev1=3 → next=5

Step 5: prev2=3, prev1=5 → next=8

This approach is the most efficient in terms of memory, with O(n) time and O(1) space complexity.

4.4.3 All-Pairs Shortest Path

In graph problems, sometimes we need to find the shortest path between all pairs of vertices, not just from one source. This problem is called the All-Pairs Shortest Path (APSP) problem.

For example, if we have a network of cities with roads between them, APSP tells us the shortest route between every city and every other city.

Dynamic Programming gives us an efficient solution for APSP using the Floyd–Warshall Algorithm.

4.4.3.1 Floyd–Warshall Algorithm

The Floyd–Warshall algorithm is a classical Dynamic Programming (DP) algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. Unlike Dijkstra's algorithm, which finds the shortest path from a single source to all other vertices, Floyd–Warshall is an all-pairs shortest path algorithm. This makes it especially useful in applications like network routing, urban traffic planning, and analysis of social or communication networks.

The key idea behind the algorithm is to gradually improve the estimate of the shortest distance between every pair of vertices. Instead of directly trying to compute the shortest path in one step, the algorithm incrementally considers more intermediate vertices and refines the path length.

Suppose we have a directed graph with vertices labeled 1, 2, …, n, and weighted edges (positive, negative, or zero, but without negative weight cycles).

We define a DP state:

D[k][i][j]=length of the shortest path from i to j using only the first k vertices as intermediate points.

- **Base case (k = 0) :** If no intermediate vertices are allowed, the shortest path from i to j is simply the direct edge weight (if it exists). If no edge exists, we take it as ∞ (infinity).
- **Transition (Recurrence Relation):** At each step, we decide whether to include vertex k as an intermediate point:

$$D[k][i][j]=\min(D[k-1][i][j],\ D[k-1][i][k]+D[k-1][k][j])$$

This means the shortest path from i to j using the first k vertices is either:

1. The same as the shortest path without using vertex k (previously computed), or
2. A path that goes through k, i.e., i → k → j.

Final result : After considering all vertices (k = n), the table D[n][i][j] gives the shortest distance between every pair (i, j).

Step-by-Step Working

The algorithm works in three nested loops:

1. The outer loop goes over intermediate vertices (k = 1 to n).
2. Inner two loops check every pair of vertices (i, j).
3. For each pair, update the shortest path using the recurrence relation.

This way, all possible intermediate vertices are considered systematically, and the shortest path estimates are gradually refined.

Example :

Consider a directed weighted graph with 4 vertices:

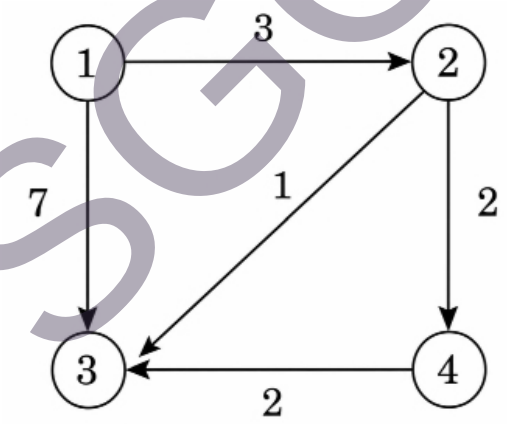


Fig. 4.4.3 Weighted graph with 4 vertices

Initial distances (D[0][i][j]):

- Direct edge weights are filled in.
- If no edge exists, distance is ∞.
- The distance from a node to itself is 0.

- As k increases (k = 1, 2, 3, 4), we update the matrix by checking if paths through intermediate vertices yield shorter results.

At the end, we obtain the shortest distance between all pairs of vertices.

Algorithm

1. Start with the adjacency matrix of the graph:
   - If there is an edge between i and j, the cost is that weight.
   - If there is no edge, the cost is infinity.
   - The diagonal (i to i) is 0.
2. Repeat for each vertex k = 1 to n:
   - For every pair (i, j), check if dist[i][j] > dist[i][k] + dist[k][j].
   - If yes, update dist[i][j].
3. After n iterations, dist[i][j] will hold the shortest path distance from i to j.

Example

Table 4.4.1 Representation of graph with 4 vertices

From → To	1	2	3	4
1	0	5	∞	10
2	∞	0	3	∞
3	∞	∞	0	1
4	∞	∞	∞	0

"∞" means no direct edge.

Iterations

- After considering vertex 2, the distance from 1 → 3 becomes 5+3 = 8.
- After considering vertex 3, the distance from 1 → 4 becomes 8+1 = 9 (better than 10).

Final result :

Table 4.4.2 Final result

From → To	1	2	3	4
1	0	5	8	9
2	∞	0	3	4
3	∞	∞	0	1
4	∞	∞	∞	0

Complexity

- Time Complexity : $O(n^3)$ because we use 3 nested loops (k, i, j).
- Space Complexity : $O(n^2)$ to store the distance matrix.

This makes Floyd Warshall good for dense graphs with small n (like ≤ 400 vertices). For very large graphs, other algorithms like Dijkstra's (repeated for each node) may be better.

Advantages

- Works for all-pairs shortest paths in one run.
- Can handle graphs with negative edge weights (unlike Dijkstra).
- Simple and systematic approach based on DP.

Limitation

- Does not work if the graph has a negative weight cycle, since the distance can become indefinitely small.
- Slower than Dijkstra's algorithm for sparse graphs when only single-source shortest paths are required.

4.4.4 Back Tracking

Backtracking is a problem-solving technique that explores all potential solutions and selects the most appropriate or desired ones. It is commonly applied to problems with multiple possible outcomes. The concept of backtracking involves discarding a solution if it proves to be unsuitable and then stepping back to explore alternative options.

The backtracking algorithm is suitable for the following types of problems:

- Problems that have several possible solutions or require discovering all valid solutions.
- Problems that can be divided into smaller, similar subproblems.
- Problems that involve specific constraints or conditions that any valid solution must meet.

4.4.4.1 Key Components of the Backtracking Algorithm

1. **Decision Tree :** Backtracking explores the solution space by constructing a decision tree, where each node represents a possible decision, and each branch reflects the outcome of that decision.

2. **Recursive Strategy :** The algorithm employs recursion to navigate through the decision tree. At every stage, it makes a decision and then recursively attempts to solve the resulting smaller subproblem.

3. **Backtracking Mechanism :** If a chosen path does not lead to a valid or optimal solution, the algorithm reverses (or "backtracks") the last decision and explores a different option. This process continues until all viable paths have been examined and a solution is identified.

4.4.4.2 How the Backtracking Algorithm Works

The backtracking algorithm searches through different possible paths to discover a sequence that leads to the solution. Along the way, it sets up checkpoints that allow it to return to a previous state if a particular path does not yield a valid solution. This process is repeated until an optimal or valid solution is found.

Steps in the Backtracking Algorithm

1. **Begin at the Starting Point :** The algorithm initiates from the starting position or the root of the decision tree. This marks the point where exploration of various paths begins.

2. **Make a Choice :** At each stage, the algorithm selects an option that progresses it forward such as choosing a direction in a maze or an option in a decision-making scenario.

3. **Validate the Path :** Once a decision is made, the algorithm checks whether the current path satisfies the problem's rules or constraints. If it doesn't, or if it leads to a dead end, the algorithm prepares to backtrack.

4. **Backtrack When Needed :** If the current path fails to produce a solution, the algorithm reverses the last choice and attempts a different alternative from the previous point.

5. **Continue Searching :** This process of decision-making, validating, and backtracking continues as the algorithm explores all potential paths until a valid solution is identified or no more paths remain.

6. **Stop Upon Finding a Solution or Exhausting All Options :** The algorithm concludes once a correct solution is discovered or when every possible path has been tried without success.

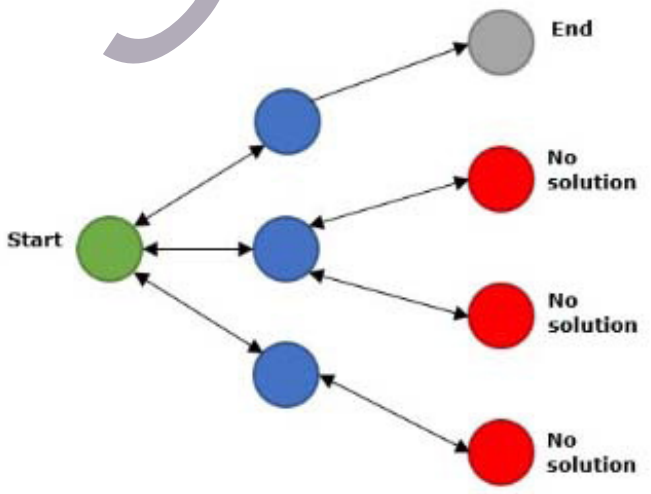


Fig. 4.4.4 Backtracking

In the figure 4.4.1, the green point represents the starting position, blue indicates intermediate steps, red marks dead ends with no feasible solution, and grey signifies the final solution point.

When the backtracking algorithm reaches the end of a path, it verifies whether the path represents a valid solution. If it does, the solution is returned. If not, the algorithm retraces its steps to the previous point to continue exploring other possible paths.

4.4.4.3 Applications of the Backtracking Algorithm

- **Puzzle Solving :** Applied in solving logic-based puzzles like Sudoku, crosswords, and the N-Queens problem.
- **Combinatorial Problems :** Used to generate all possible permutations, combinations, and subsets of a given set.
- **Constraint-Based Problems :** Useful in scenarios like graph coloring, task scheduling, and job assignments where specific rules or constraints must be followed.
- **Pathfinding :** Helps in navigating mazes or solving path-related problems, such as the Rat in a Maze.
- **Game Strategies :** Employed in strategic games to explore all possible moves, as seen in problems like the Knight's Tour.
- **Decision-Making Scenarios :** Assists in choosing the best option when multiple possibilities exist, such as in the Subset Sum problem.
- **String Generation :** Generates strings that fulfill specific conditions, for example, creating valid sequences of balanced parentheses.
- **Optimization Challenges :** Solves complex problems focused on maximizing gains or minimizing losses while adhering to constraints.

4.4.5 N-Queens Problem

In the N-Queens problem, the goal is to place N queens on an N×N chessboard so that no two queens threaten each other. When n=4n = 4, the problem is referred to as the 4-Queens problem. Similarly, when n=8n = 8, it is known as the 8-Queens problem.A queen can attack any other queen that lies in the same row, column, or diagonal. The most widely used method to solve this puzzle is the backtracking algorithm.

4.4.5.1 4-Queens Problem

Consider a 4×4 chessboard with 4 queens to be placed. The goal is to position all four queens on the board such that no two queens are in the same row, column, or diagonal.

- Explicit Constraint : Four queens must be placed on a 4×4 board, which can be done in 4^4 (i.e., 256) different ways if no restrictions are applied.
- Implicit Constraints : No two queens can share the same row, column, or diagonal.

Let{x1, x2, x3, x4} be the solution vector where the x1 column on which the queen i is placed.

As an initial step, the first queen is placed in the first row and first column.

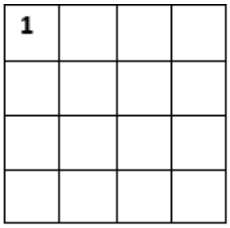


Fig. 4.4.5 First queen is placed in the first row and first column

The second queen must not be placed in the first row or the second column. It should be positioned in the second row, choosing from the second, third, or fourth columns. However, placing it in the second column would put it on the same diagonal as the first queen. Therefore, it should be placed in the third column to avoid conflict.

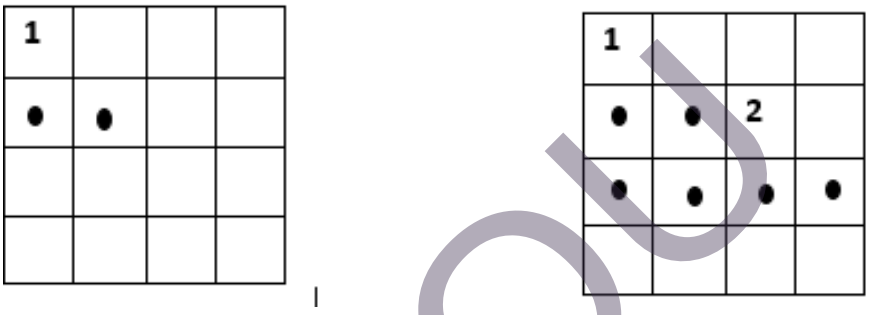


Fig. 4.4.6 The second queen is placed in the third column

Since placing the third queen in the third row does not lead to a valid position, we need to backtrack,go back to the second queen and try placing it in a different column.

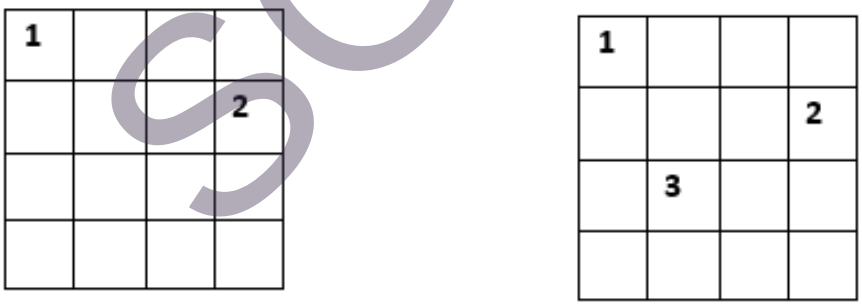


Fig. 4.4.7 Placing of 3rd queen

At this point, the fourth queen needs to be placed in the fourth row and third column, but doing so would result in a diagonal attack from the third queen. Therefore, we backtrack by removing the third queen and attempting to place it in the next column. However, no valid position is available for queen 3. So, we backtrack further,remove the second queen and try placing it in a different column. Again, no valid options remain. As a result, we go back to the first queen and move it to the next column to explore new possibilities.

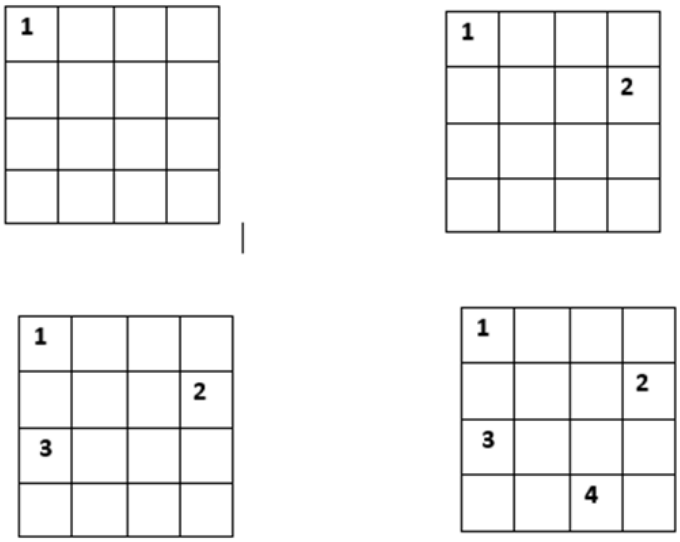


Fig. 4.4.8 Placing of 4th queen

Hence the solution of to 4-queens's problem is x1=2, x2=4, x3=1, x4=3 i.e;the first queen is positioned in the second column, the second queen is placed in the fourth column, the third queen is set in the first column, and the fourth queen is placed in the third column.

4.4.5.2 8-Queens Problem

A well-known combinatorial challenge is the 8-queens problem, which involves placing 8 queens on an 8×8 chessboard such that no two queens threaten each other meaning no two queens share the same row, column, or diagonal.

To solve this, we can apply a strategy similar to the one used for the 4-queens problem. The solution approach for the 8-queens problem is essentially an extension of the N-queens algorithm, where we set n=8n = 8.

If two queens are placed at positions (i,j) and (k,l). They are on the same diagonal only if

i-j=k-l …….(1) or

i+j=k+l .....(2).

From (1) and (2) implies j-l=i-k and j-l=k-i

Two queens lie on the same diagonal iff |j-l|=|i-k|

The solution of 8 queens problem can be obtained similar to the solution of 4 queens.

X1=3, X2 =6, X3=2, X4=7, X5=1, X6=4, X7=8, X8=5

The solution can be shown as

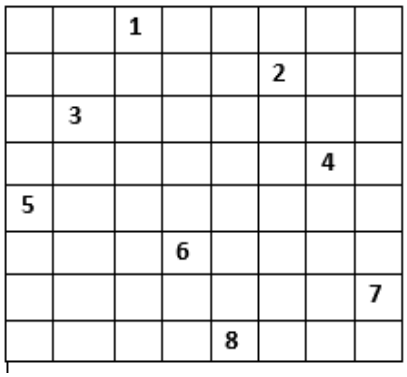


Fig. 4.4.9 8 - Queens problem

Summarised Overview

Dynamic Programming (DP) and Backtracking are two important approaches in algorithm design that help solve complex problems efficiently.

Dynamic Programming works by breaking a large problem into smaller overlapping subproblems and storing their results to avoid repeated calculations. This approach is widely used in optimization problems. For example, the Fibonacci sequence can be solved efficiently using DP instead of recursion, and the Floyd Warshall algorithm uses DP to compute the shortest paths between all pairs of vertices in a graph.

Backtracking is a step by step problem solving technique that tries possible solutions and abandons them if they do not satisfy the conditions. It is particularly useful for problems that involve searching and constraints. A classic example is the N-Queens problem, where we place queens on a chessboard such that no two attack each other. Backtracking systematically explores all possibilities and eliminates invalid positions.

Together, these methods show two different problem solving philosophies: DP focuses on reusing previous results to save time, while Backtracking explores solution paths carefully to find valid ones. Both are essential in computer science, forming the basis for solving real world problems like network routing, scheduling, resource allocation, game design, and puzzle solving.

Assignments

1. Explain the principle of Dynamic Programming with a real-life example. How does it differ from the Divide and Conquer approach?

2. Derive the recursive relation for the Fibonacci sequence. Write a short note on how Dynamic Programming reduces the time complexity of Fibonacci computation.

3. Compare Dijkstra's Algorithm and Floyd–Warshall Algorithm. In what situations is Floyd–Warshall more suitable?

4. Define Backtracking. Explain with an example how it can be used to generate all possible binary strings of length n.

5. Solve the 4-Queens problem using backtracking. Draw the state-space tree and explain how invalid states are pruned.

6. Discuss the advantages and limitations of Dynamic Programming and Backtracking. Give at least two real-world problems where each approach is useful.

7. Write an algorithm (pseudocode) for the subset sum problem using backtracking. Trace the solution for the set {3, 4, 5, 2} and target sum = 7.

8. Consider a directed weighted graph. Explain step by step how Floyd–Warshall guarantees finding the shortest path between every pair of vertices.

9. "Dynamic Programming is about memory efficiency, while Backtracking is about search efficiency." Justify this statement with examples.

Reference

1. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.

2. Bellman, R. (1957). *Dynamic programming*. Princeton University Press

3. Horowitz, E., Sahni, S., & Rajasekaran, S. (2008). *Fundamentals of computer algorithms* (2nd ed.). Universities Press.

4. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Suggested Reading

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.

2. Kleinberg, J., & Tardos, É. (2006). Algorithm design. Pearson.

3. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). Algorithms. McGraw-Hill Higher Education.

4. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2021). Data structures and algorithms in Python (2nd ed.). Wiley.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: …… **Reg. No. : .....**

**Name: ……..... **

FIRST SEMESTER MCA EXAMINATION

DISCIPLINE CORE COURSE

M25CA01DC - Data Structures with C

2026 - Admission Onwards

MODEL QUESTION PAPER- SET A

Time: 3 Hour Max Marks: 70

SECTION A

Answer any **ten** *questions of the following. Each question carries* **two** *mark.*

(10 × 2 = 20 Marks)

1. Define balance factor in an AVL tree. What is the formula to calculate the balance factor of a node?
2. What is the purpose of the malloc() and calloc() functions in C? Mention one key difference between them.
3. What is a collision in hashing? Why does it occur?
4. Write any two applications of a stack .
5. What is the conditional (ternary) operator? Write its syntax.
6. What is the difference between a complete graph and a planar graph?
7. What is a self-referential structure?
8. Why is Binary Search considered more efficient than Linear Search for large datasets?
9. Differentiate between while loop and do-while loop.
10. What is the difference between DFS and BFS traversal?
11. Name any two other sorting techniques.
12. How does Big-O notation help in comparing two algorithms with different growth rates?
13. Does an MST contain cycles? Why?
14. Compare Kruskal's Algorithm, and Prim's Algorithm.
15. Write the limitations of a stack.

SECTION B

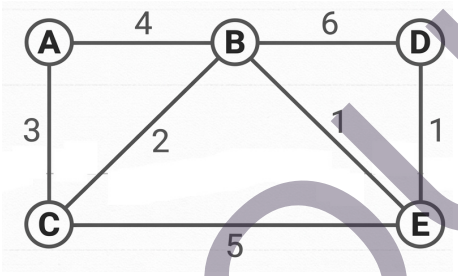
Answer any **six** *questions of the following. Each question carries* **five** *marks.*

(6×5 =30 Marks)

16. What are the rules for naming variables in C? Discuss the difference between variables and constants in C programming.

17. What is a Circular Queue? How does it overcome the limitation of a linear queue?

18. A connected weighted graph is given with vertices and edge weights. Explain how a Minimum Cost Spanning Tree is constructed step by step using either Prim's algorithm.



19. Explain the concept of dynamic programming and discuss any four of its applications.

20. Explain the types of arrays in C with examples.

21. Describe the steps involved in infix to postfix conversion with an example.

22. A graph is given in both adjacency matrix and adjacency list form. Compare these two representations and discuss their advantages and disadvantages in different scenarios.

23. Why is Ω-notation considered the best-case analysis of an algorithm? Find the Ω-notation for the following code:

```
for(int i = 0; i < n; i++) {
  if(arr[i] == key) {
     break;
  }
}
```

(Hint: Best case occurs when element is found at first position)

24. What is a self-referential structure in C? Explain its applications..

SECTION C

Answer any **two** *questions of the following. Each question carries* **ten** *marks.*

(2×10 = 20 Marks)

25. Explain the concept of searching in data structures. Describe linear search and binary search algorithms in detail with examples, algorithm steps, implementation, and time complexity analysis.

26. What is an AVL tree? Explain its rotations with examples.

27. Explain the Greedy algorithm design approach and illustrate its application using the Fractional Knapsack Problem with a suitable example. Also, justify why the greedy method fails to produce an optimal solution for the 0/1 Knapsack Problem.

28. Explain the different types of dynamic memory allocation functions in C with syntax, examples and differences.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: …… **Reg. No. : .....**

**Name: ……..... **

FIRST SEMESTER MCA EXAMINATION

DISCIPLINE CORE COURSE

M25CA01DC - Data Structures with C

2026 - Admission Onwards

MODEL QUESTION PAPER- SET B

Time: 3 Hour Max Marks: 70

SECTION A

Answer any **ten** *questions of the following. Each question carries* **two** *mark.*

(10 × 2 = 20 Marks)

1. State the properties of a Binary Search Tree.
2. Differentiate between static memory allocation and dynamic memory allocation in C.
3. How does a hash function help in locating data quickly?
4. What is infix notation?
5. Differentiate between logical AND (&&) and logical OR (||) operators.
6. Compare adjacency matrix and adjacency list representations of a graph.
7. What is a linked list? Name the types of linked list.
8. During sorting, a pivot divides the list into two unequal parts repeatedly. What impact does this have on performance?
9. What is a control structure in C? Name any two types.
10. Define the degree of a node and height of a tree with an example.
11. What is insertion sort ? Explain with an example.
12. An algorithm takes 5n+10 steps for input size n. How would its growth be expressed using asymptotic notation?

13. How many edges are in a spanning tree with n vertices?

14. What is a Minimum Cost Spanning Tree (MCST)?

15. Write about basic operations of the queue.

SECTION B

Answer any **six** *questions of the following. Each question carries* **five** *marks.*

(6×5 =30 Marks)

16. What are tokens in C?

17. State the difference between Stack and Queue based on their working principles.

18. Define a Minimum Spanning Tree (MST). Explain Kruskal's Algorithm in detail and compare it with Prim's Algorithm.

19. Describe the working principle of the Floyd–Warshall Algorithm with an example.

20. Discuss the decision-making statements in C with suitable examples.

21. How to evaluate a postfix expression? Explain the algorithm with an example.

22. Compare different forms of binary trees based on their structure and shape. How do these variations affect data storage and traversal?

23. Explain how Big-O helps in choosing the best algorithm.

24. Write about Dereference Operator.How to dereference a pointer?.

SECTION C

Answer any **two** *questions of the following. Each question carries* **ten** *marks.*

(2×10 = 20 Marks)

25. What is searching? Explain the types of searching with examples and algorithms.

26. Explain the structure and properties of a B+ Tree. Describe in detail the basic operations such as insertion and deletion, including all possible cases, with suitable examples.

27. Discuss the working principle of Binary Search and Quick Sort with examples.

28. Explain how memory is allocated, reallocated, and freed during program execution, comparing different allocation approaches with examples.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

DATA STRUCTURES WITH C

COURSE CODE: M25CA01DC

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-686027-4-8