SGOU
Higher Education for All

Data Structures with C Lab

COURSE CODE: M25CA01PC
Master of Computer Applications
Practical 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 Lab

Course Code: M25CA01PC

Semester - I

Practical 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 LAB

Course Code: M25CA01PC
Semester- I
Practical Core Course
Master of Computer Applications

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

Edition
March 2026

www.sgou.ac.in

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.

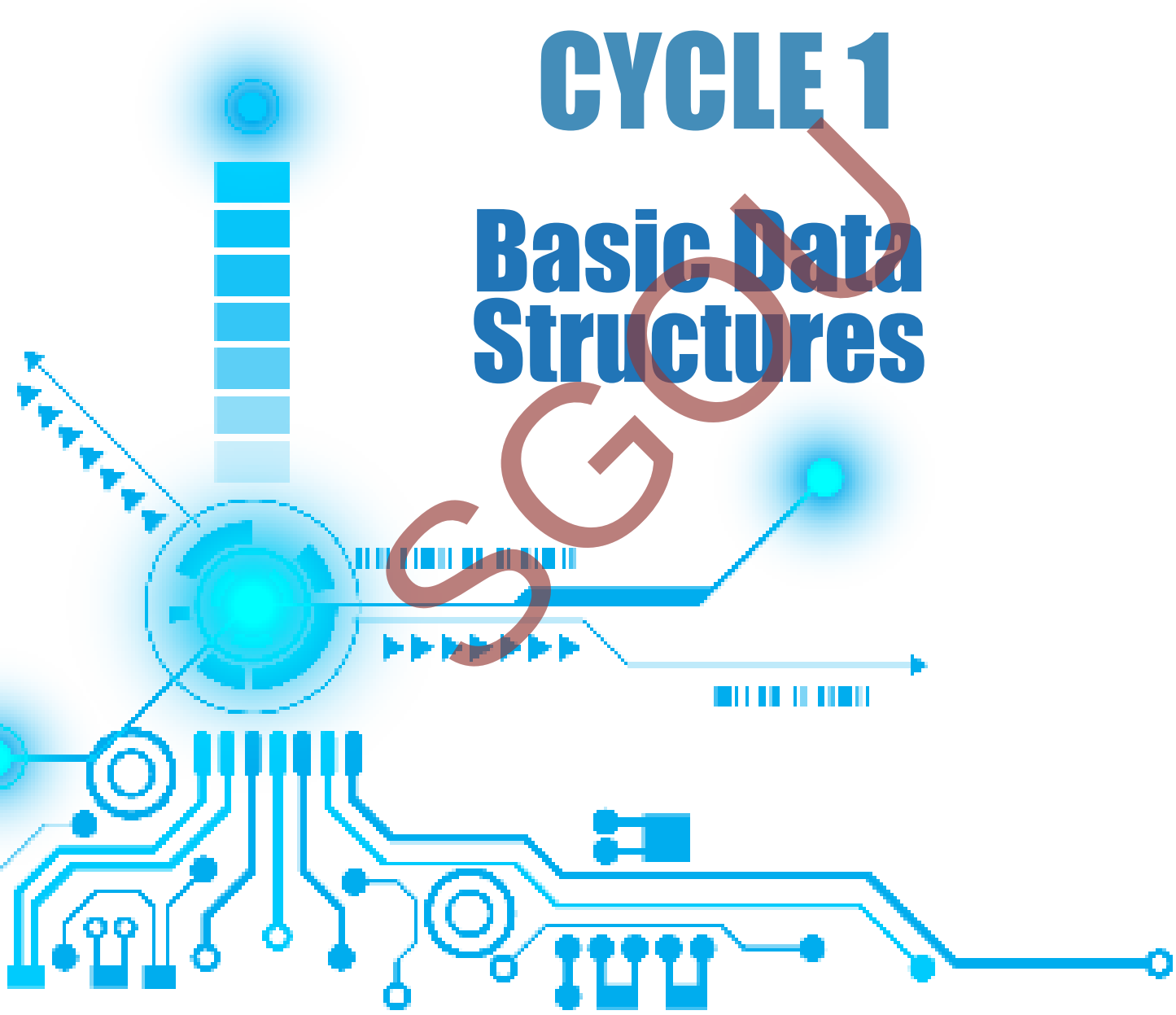
Warm regards,

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

Contents

CYCLE 1

Basic Data Structures



Introduction

Data Structures are the fundamental building blocks of efficient software development. They provide organized ways to store, manage, and manipulate data in memory, enabling algorithms to perform tasks optimally. Mastering data structures is essential for solving complex problems in programming, system design, and database management. C is widely used for teaching data structures due to its ability to directly manipulate memory using pointers, allowing students to understand underlying data representation and memory management. Knowledge of C-based data structures is critical for advanced topics such as algorithms, operating systems, databases, networking, and artificial intelligence.

This lab manual covers a complete cycle from basic C programming and function concepts to complex data structures, trees, and graphs, including implementations of Prim's and Kruskal's algorithms. Students will gain hands-on experience with arrays, linked lists, stacks, queues, trees, and graphs, along with related algorithms for searching, sorting, and optimization.

Objectives of the Lab

1. Understand the fundamental concepts of data structures in C, including arrays, linked lists, stacks, queues, trees, and graphs.
2. Implement basic operations such as insertion, deletion, traversal, searching, and sorting using C.
3. Apply pointers effectively to create dynamic data structures and manage memory efficiently.
4. Analyze the performance of different data structures in terms of time and space complexity.
5. Develop problem-solving skills by designing and implementing modular and efficient programs using data structures.

Scope of the Lab

This lab equips students with practical knowledge of data structures and their applications. Students will:

- Learn to write structured programs using C with functions, pointers, and modular design.
- Gain hands-on experience in dynamic memory allocation using malloc, calloc, and free.

- Understand algorithmic efficiency by implementing various sorting and searching techniques.
- Solve real-world problems using stacks, queues, linked lists, trees, and graphs.
- Integrate data structures with file handling for persistent data storage.

By completing this lab, students will be prepared for advanced topics in system programming, database design, and software engineering.

Expectations from Students

- Attend all lab sessions and actively participate in practical exercises.
- Complete and submit lab assignments on time to reinforce learning.
- Collaborate with peers for discussions and problem-solving but ensure submitted work is individual and original.
- Explore additional examples and test cases beyond the given exercises.
- Maintain discipline, focus, and a clean workspace.
- Document all experiments properly, including flowcharts, code, output, and observations.

Lab Guidelines

- Follow the experiment instructions and instructor guidance carefully.
- Use approved compilers such as GCC, Code::Blocks, Dev-C++, or Visual Studio Code.
- Regularly save and back up code to prevent data loss.
- Write modular, readable, and well-commented code.
- Avoid plagiarism; all code must be original or properly cited.
- Participate actively in discussions and seek help when needed.
- Ensure all experiments follow proper memory management practices to avoid leaks.

Tools and Software for C Programming

1. Compilers & IDEs

- **GCC Compiler** – standard compiler for C programs.

- **Code::Blocks** – lightweight IDE for learning and development.
- **Dev-C++** – simple IDE with integrated compiler.
- **Visual Studio Code (VS Code)** – versatile code editor with C extensions.

2. **Debugging Tools**
   - **gdb** – GNU debugger for troubleshooting C programs.
   - **Integrated IDE debuggers** – breakpoints, step execution, and variable inspection.
3. **Version Control**
   - **Git** – for tracking code changes.
   - **GitHub/GitLab/Bitbucket** – repository management and collaboration.

Steps to Configure C Programming Environment

1. **Install Compiler**
   - For Windows: Install GCC via MinGW or TDM-GCC.
   - For Linux/macOS: GCC is often pre-installed; else install via package manager.
2. **Install IDE**
   - Download and install IDE such as Code::Blocks, Dev-C++, or VS Code.
3. **Verify Installation**
   - Open terminal or command prompt and type:

```
gcc --version      # to check if the compiler is installed correctly
```

4. **Set Up Workspace**
   - Create a folder for lab exercises.
   - Organize code files with proper naming conventions

Code of Conduct

- Respect instructors, peers, and lab assistants.
- Maintain integrity while coding and documenting experiments.
- Clearly label outputs, flowcharts, and diagrams.
- Ensure proper memory management to avoid leaks and errors.
- Handle lab equipment and systems responsibly.

Experiment 1

Implementation of a simple calculator using functions in C, swapping two integer values using Pointers in C

Objectives

- Understand the concepts of functions and pointers in C.
- Demonstrate call by value and call by reference using functions.
- Apply pointers to perform operations like swapping numbers and finding the largest number.
- Analyze how pointers access and modify memory addresses and variable values.
- Evaluate and implement small programs using functions and pointers to solve practical problems.

1.1.1Theory

C is a powerful, general-purpose programming language that provides both low-level memory access and high-level structured programming features. The basics of C programming revolve around understanding its building blocks, which form the foundation for solving computational problems efficiently. Every C program begins with a structured format that typically includes header files, the main() function, and a set of statements enclosed within braces { }. Variables are used to store data, and they must be declared with appropriate data types such as int, float, char, or double, ensuring proper memory allocation. Operators, including arithmetic, relational, logical, and assignment operators, are employed to perform computations and decision-making. Control structures like if-else, switch, for, while, and do-while loops allow the execution of code in a logical sequence, enabling branching and iteration. Functions are introduced for modular programming, improving readability and reusability of code, while arrays and pointers extend the ability to handle collections of data and direct memory access.

Input and output operations in C are primarily performed using standard library functions such as printf() and scanf(). Together, these fundamental elements—data types, variables, operators, control structures, functions, arrays, and pointers—create the backbone of C programming. In the context of data structures, C is widely used because of its efficiency, portability, and fine control over memory through pointers. Before exploring advanced topics such as arrays, linked lists, and trees, it is important to build a strong foundation in the basics of C programming such as functions and pointers.

1.1.1.1 Function in C

Functions in C are self-contained blocks of code designed to perform specific tasks, making programs more modular, reusable, and easier to manage. A function is defined once and can be invoked multiple times from different parts of the program, thereby reducing redundancy and improving readability. In C, every program begins execution from the main() function, but programmers can define additional user-defined functions to handle complex problems in smaller, manageable units. Each function typically consists of a function declaration (prototype), which specifies its name, return type, and parameters; a function definition, which contains the actual logic; and a function call, which executes the code when needed. Functions can be classified into two types: standard library functions or built in functions , such as printf(), scanf(), sqrt(),strlen() etc, which are pre-defined functions and user-defined functions, which programmers create to meet specific requirements.

1.1.1.2 Pointers in C

Pointers are one of the most powerful and distinctive features of the C programming language. A pointer is a special variable that stores the memory address of another variable rather than the actual value. This ability to directly access and manipulate memory makes pointers essential for efficient programming, dynamic memory allocation, and data structure implementation.

Every variable in C is stored at a specific memory location, which is identified by a unique address. A pointer "points" to this address. For example, if int a = 10; then &a gives the address of a, and a pointer can be used to store this address. The * operator (dereference operator) is then used to access or modify the value stored at that address.

Syntax:

```
data_type *pointer_name;
```

Example:

```
#include <stdio.h>

int main() {
    int a = 10;       // normal variable
    int *ptr;         // pointer declaration
    ptr = &a;         // storing address of a in pointer
    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", &a);
    printf("Pointer stores = %p\n", ptr);
```

```
    printf("Value at pointer = %d\n", *ptr);  // dereferencing
    return 0;
}
```

OUTPUT

```
Value of a = 10
Address of a = 0x7ffc81053a94                    //Example  Memory address
Pointer stores = 0x7ffc81053a94
Value at pointer = 10
```

1.1.2 Implementation

1. Write a C program that takes an integer from the user and prints its reverse.

```
#include <stdio.h>
int main() {
    int num, rev = 0, rem;
    printf("Enter an integer: ");
    scanf("%d", &num);
    while (num != 0) {
        rem = num % 10;          // get last digit
        rev = rev * 10 + rem;    // build reverse
        num = num / 10;          // remove last digit
    }
    printf("Reversed number = %d\n", rev);
    return 0;
}
```

OUTPUT

```
Enter an integer: 786
Reversed number = 687
```

2. Write a C program that reverses a given integer using a function with arguments and return value.

```
#include <stdio.h>
// Function declaration
int reverse(int num);
int main() {
    int num, result;
    printf("Enter an integer: ");
    scanf("%d", &num);
    // Function call
    result = reverse(num);
    printf("Reversed number = %d\n", result);
    return 0;
}
// Function definition
int reverse(int num) {
    int rev = 0, rem;
    while (num != 0) {
        rem = num % 10;         // extract last digit
        rev = rev * 10 + rem;   // build reverse
        num = num / 10;         // remove last digit
    }
    return rev;   // return reversed number
}
```

OUTPUT

```
Enter an integer: 8734
Reversed number = 4378
```

3. Write a C program that implements a simple calculator using functions for addition, subtraction, multiplication, and division.

```
#include <stdio.h>

// Function declarations

int add(int a, int b);

int subtract(int a, int b);

int multiply(int a, int b);

float divide(int a, int b);

int main() {

    int choice, num1, num2;

    float result;

    printf("Simple Calculator using Functions\n");

    printf("1. Addition\n");

    printf("2. Subtraction\n");

    printf("3. Multiplication\n");

    printf("4. Division\n");

    printf("Enter your choice (1-4): ");

    scanf("%d", &choice);

    printf("Enter two numbers: ");

    scanf("%d %d", &num1, &num2);

    switch (choice) {
```

```
        case 1:
            result = add(num1, num2);
            printf("Result = %.2f\n", result);
            break;
        case 2:
            result = subtract(num1, num2);
            printf("Result = %.2f\n", result);
            break;
        case 3:
            result = multiply(num1, num2);
            printf("Result = %.2f\n", result);
            break;
        case 4:
            if (num2 != 0)
                result = divide(num1, num2);
            else {
                printf("Error: Division by zero!\n");
                return 0;
            }
            printf("Result = %.2f\n", result);
            break;
        default:
            printf("Invalid choice!\n");
    }
    return 0;
}
```

```
// Function definitions
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int multiply(int a, int b) {
    return a * b;
}
float divide(int a, int b) {
    return (float)a / b;
}
```

OUTPUT

```
Simple Calculator using Functions
    1. Addition
    2. Subtraction
    3. Multiplication
    4. Division
Enter your choice (1-4): 1
Enter two numbers: 2
5
Result = 7.00
```

4. **Write a C program that demonstrates swapping of two numbers using a function with call by value.**

```
#include <stdio.h>

// Function declaration

void swap(int x, int y);

int main() {

    int a, b;

    printf("Enter two numbers: ");

    scanf("%d %d", &a, &b);

    printf("Before swapping in main: a = %d, b = %d\n", a, b);

    swap(a, b);   // function call (call by value)

    printf("After swapping in main: a = %d, b = %d\n", a, b);

    return 0;

}

// Function definition (Call by Value)

void swap(int x, int y) {

    int temp;

    temp = x;

    x = y;

    y = temp;

    printf("Inside swap function: x = %d, y = %d\n", x, y);

}
```

OUTPUT

```
Enter two numbers: 7

8

Before swapping in main: a = 7, b = 8

Inside swap function: x = 8, y = 7

After swapping in main: a = 7, b = 8
```

5. Write a C program that demonstrates swapping of two numbers using a function and pointers.

```
#include <stdio.h>
// Function to swap two numbers using call by reference
void swap(int *a, int *b) {
    int temp;
    temp = *a;   // store the value at address a
    *a = *b;     // assign value at address b to address a
    *b = temp;   // assign stored value to address b
     printf("Inside swap function: x = %d, y = %d\n", *a, *b);
}
int main() {
    int x, y;
    printf("Enter first number: ");
    scanf("%d", &x);
    printf("Enter second number: ");
    scanf("%d", &y);
    printf("Before swapping: x = %d, y = %d\n", x, y);
    swap(&x, &y);  // pass addresses of x and y
    printf("After swapping: x = %d, y = %d\n", x, y);
    return 0;
}
```

OUTPUT

```
Enter first number: 8
Enter second number: 9
Before swapping: x = 8, y = 9
Inside swap function: x = 9, y = 8
After swapping: x = 9, y = 8
```

6. **Write a C program that finds the largest of three numbers using a function and pointers.**

```
#include <stdio.h>
// Function to find and print the largest number
void printLargest(int *a, int *b, int *c) {
    int *largest = a;  // assume first is largest
    if(*b > *largest)
        largest = b;
    if(*c > *largest)
        largest = c;
    printf("The largest number is: %d\n", *largest);
}
int main() {
    int x, y, z;
    printf("Enter three numbers: ");
    scanf("%d %d %d", &x, &y, &z);
    printLargest(&x, &y, &z);  // pass addresses
    return 0;
}
```

OUTPUT

```
Enter three numbers: 89
67
778
The largest number is: 778
```

7. **Write a C program to calculate the area and circumference of a circle using pointers**

```
#include <stdio.h>
int main() {
    float radius, area, circumference;
    float *r, *a, *c;  // pointers for radius, area, and circumference
    const float pi = 3.14159;
    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);
    // Assign addresses to pointers
    r = &radius;
    a = &area;
    c = &circumference;
    // Calculate area and circumference using pointers
    *a = 3.14159 * (*r) * (*r);
    *c = 2 * 3.14159 * (*r);
    printf("Area of the circle: %.2f\n", *a);
    printf("Circumference of the circle: %.2f\n", *c);
    return 0;
}
```

OUTPUT

```
Enter the radius of the circle: 2
Area of the circle: 12.57
Circumference of the circle: 12.57
```

1.1.3 Lab Practice Questions

1. Write a C program that reverses a given integer using a function with arguments and no return value.

2. Write a C program that calculates the area of a square and a rectangle using functions.

3. Write a C program that calculates the factorial of a given number using a function with arguments and no return value.

4. Write a C program that checks whether a given integer is a palindrome using a function with arguments and return value.

5. Write a C program that finds the smallest of three numbers using a function and pointers.

6. Write a C program to swap two numbers using pointers (without using functions).

7. Write a C program to calculate the area and perimeter of a rectangle using pointers.

8. Write a C program to calculate the area and perimeter of a triangle using pointers.

Experiment 2

Implementation and operations of Stack and Queue using Array and Linked List

Objectives

- Understand the concepts of array and linked list data structures.
- Explain the difference between array-based and linked list implementations.
- Recall the concepts of stack and queue data structures.
- Implement stack operations (push and pop) and queue operations (enqueue and dequeue) using C programming.
- Compare the efficiency of array and linked list implementations for stack and queue operations.
- Interpret the output of stack and queue programs to solve practical problems.

1.2.1Theory

Data structures in C are ways of organizing and storing data in a computer to enable efficient access, modification, and management. They are essential for designing optimized algorithms and software systems, as the choice of data structure directly affects performance in operations such as insertion, deletion, searching, and sorting. In C, data structures are implemented using primitive types, arrays, pointers, structures, and dynamic memory allocation functions like malloc() and free(). They can be broadly classified into primitive types, like int, float, and char, and non-primitive types, including arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Arrays store elements of the same type in contiguous memory locations and allow fast access but have fixed size, whereas linked lists use nodes connected via pointers, allowing dynamic sizing and efficient insertion or deletion. Stacks and queues are linear structures that follow LIFO (Last In, First Out) and FIFO (First In, First Out) principles, respectively, and are widely used in function calls, expression evaluation, and resource scheduling. Trees and graphs represent hierarchical and networked relationships, while hash tables enable fast key-based data retrieval. C is particularly suitable for implementing data structures because it allows low-level memory manipulation, efficient runtime performance, and a clear understanding of how data is managed at the hardware level.In this lab, students will learn how to create and perform operations on Stack and Queue data structures using two different approaches: arrays and linked lists. The lab demonstrates both static and dynamic memory implementations, allowing a thorough understanding of how these structures work in C programming.

1.2.1.1 Arrays and Linked Lists in C

In C programming, arrays and linked lists are fundamental data structures used to store collections of elements, but they differ significantly in structure and memory management. An array is a collection of elements of the same type stored in contiguous memory locations, allowing direct access to any element using an index, which makes searching and updating elements fast. However, arrays have a fixed size, so their capacity must be defined at compile time, and insertion or deletion of elements requires shifting existing elements, which can be inefficient. In contrast, a linked list is a dynamic data structure composed of nodes, where each node contains data and a pointer to the next node, allowing elements to be inserted or deleted easily without shifting other elements. Linked lists can grow or shrink at runtime, making them more flexible than arrays, but they require extra memory for pointers and sequential traversal is needed to access elements, making direct access slower compared to arrays. Arrays are generally preferred when the number of elements is known and memory efficiency is important, while linked lists are ideal for situations with frequent insertions and deletions or when the size of the dataset can change dynamically. Both structures form the backbone of more complex data structures like stacks, queues, graphs and trees.

Arrays

Syntax:

```
datatype arrayName[arraySize];
```

Example:

```
#include <stdio.h>
int main() {
    int marks[5]; // Array to store marks of 5 students
    // Input marks
    for(int i = 0; i < 5; i++) {
        printf("Enter mark for student %d: ", i+1);
        scanf("%d", &marks[i]);
    }
    // Display marks
    printf("Marks of students are: ");
    for(int i = 0; i < 5; i++) {
        printf("%d ", marks[i]);
```

```
    }
    return 0;
}
```

Linked List - Node Structure Syntax:

```
struct Node {
    int data;
    struct Node *next;
};
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
int main() {
    // Create nodes
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;
    // Allocate memory
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
    // Assign data
    head->data = 10;
    head->next = second;
    second->data = 20;
    second->next = third;
```

```
    third->data = 30;
    third->next = NULL;
    // Display linked list
    struct Node* temp = head;
    printf("Linked List: ");
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    return 0;
}
```

1.2.1.2 Stack Data Structure in C

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle, meaning the last element added to the stack is the first one to be removed. It can be visualized as a collection of elements stacked on top of each other, similar to a stack of plates, where you can only add or remove plates from the top.

Basic Operations on a Stack

1. **Push:** Adds an element to the top of the stack.
2. **Pop:** Removes the top element from the stack.
3. **Peek/Top:** Returns the top element without removing it.
4. **isEmpty:** Checks if the stack is empty.
5. **isFull:** Checks if the stack is full (only for array implementation).

Stack Implementation in C

Stacks can be implemented in two ways in C:

1. Using Arrays (Static Stack)
   - ♦ Uses a fixed-size array to store elements.
   - ♦ A variable, usually called top, keeps track of the index of the last inserted element.

- Simple and easy to implement.
- Size is fixed; cannot grow beyond array limit.

2. Using Linked Lists (Dynamic Stack)

- Uses nodes connected via pointers to store elements dynamically.
- The top of the stack points to the head of the linked list.
- Can grow or shrink at runtime; no size limitation.
- Requires extra memory for pointers; slightly more complex to implement.

1.2.1.3 Queue Data Structure in C

A queue is a linear data structure that follows the FIFO (First In, First Out) principle, meaning the first element added to the queue is the first one to be removed. It can be visualized as a line of people waiting for service, where the person at the front is served first and new arrivals join at the rear.

Basic Operations on a Queue

1. **Enqueue:** Adds an element to the rear of the queue.
2. **Dequeue:** Removes an element from the front of the queue.
3. **Front/Peek:** Returns the front element without removing it.
4. **isEmpty:** Checks if the queue is empty.
5. **isFull:** Checks if the queue is full (only for array implementation).

Queue Implementation in C

Queues can be implemented in two ways:

1. **Using Arrays (Static Queue)**

- Uses a fixed-size array to store elements.
- Two pointers, usually front and rear, track the positions for removal and insertion.
- Simple and easy to implement.
- Fixed size; waste of memory if many elements are dequeued.

2. **Using Linked Lists (Dynamic Queue)**

- Uses nodes connected via pointers.
- The front points to the first node, and the rear points to the last node.
- Can grow or shrink dynamically; no fixed size limitation.
- Requires extra memory for pointers; slightly more complex.

1.2.2 Implementation

1. **Write a C program to insert a given element into a specific position in an array.**

```
#include <stdio.h>
int main() {
    int arr[100], n, i, element, pos;
    // Input array size
    printf("Enter number of elements in the array: ");
    scanf("%d", &n);
    // Input array elements
    printf("Enter %d elements: ", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    // Input element and position
    printf("Enter element to insert: ");
    scanf("%d", &element);
    printf("Enter position to insert (1 to %d): ", n+1);
    scanf("%d", &pos);
    if(pos < 1 || pos > n+1) {
        printf("Invalid position!\n");
```

```
        return 0;
    }
    // Shift elements to the right
    for(i = n; i >= pos; i--) {
        arr[i] = arr[i-1];
    }
    // Insert the element
    arr[pos-1] = element;
    n++; // Increase array size
    // Display updated array
    printf("Array after insertion: ");
    for(i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

OUTPUT

```
Enter number of elements in the array: 3
Enter 3 elements: 23
45
67
Enter element to insert: 100
Enter position to insert (1 to 4): 1
Array after insertion: 100 23 45 67
```

2. Write a C program to implement a stack using an array.

```
#include <stdio.h>
#define MAX 5  // Maximum size of stack
int stack[MAX];
int top = -1;
// Function to push an element into the stack
void push(int val) {
    if(top == MAX - 1) {
        printf("Stack Overflow! Cannot insert %d\n", val);
    } else {
        top++;
        stack[top] = val;
        printf("%d pushed into the stack.\n", val);
    }
}
// Function to pop an element from the stack
void pop() {
    if(top == -1) {
        printf("Stack Underflow! No element to pop.\n");
    } else {
        printf("Popped element: %d\n", stack[top]);
        top--;
    }
}
// Function to display the stack elements
void display() {
    if(top == -1) {
        printf("Stack is empty.\n");
```

```
    } else {
        printf("Stack elements: ");
        for(int i = top; i >= 0; i--) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}
// Function to peek at the top element
void peek() {
    if(top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Top element: %d\n", stack[top]);
    }
}
int main() {
    int choice, value;
    while(1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Peek\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:
            display();
            break;
        case 4:
            peek();
            break;
        case 5:
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }
   return 0;
}
```

OUTPUT

```
Stack Operations:
1. Push
2. Pop
3. Display
4. Peek
5. Exit
Enter your choice: 1
```

```
Enter value to push: 10
10 pushed into the stack.
Stack Operations:
1. Push
2. Pop
3. Display
4. Peek
5. Exit
Enter your choice: 5
Exiting program.
```

3. Write a C program to implement a stack using a linked list.

```
#include <stdio.h>
#include <stdlib.h>
// Define node structure
struct Node {
    int data;
    struct Node* next;
};
struct Node* top = NULL;
// Function to push an element into the stack
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed into the stack.\n", value);
}
```

```
// Function to pop an element from the stack
void pop() {
    if(top == NULL) {
        printf("Stack Underflow! No element to pop.\n");
    } else {
        struct Node* temp = top;
        printf("Popped element: %d\n", top->data);
        top = top->next;
        free(temp);
    }
}
// Function to peek at the top element
void peek() {
    if(top == NULL) {
        printf("Stack is empty.\n");
    } else {
        printf("Top element: %d\n", top->data);
    }
}
// Function to display stack elements
void display() {
    if(top == NULL) {
        printf("Stack is empty.\n");
    } else {
        struct Node* temp = top;
        printf("Stack elements: ");
        while(temp != NULL) {
            printf("%d ", temp->data);
```

```
            temp = temp->next;
        }
        printf("\n");
    }
}
int main() {
    int choice, value;
    while(1) {
        printf("\nStack Operations (Linked List):\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Peek\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:
                display();
                break;
            case 4:
                peek();
                break;
            case 5:
```

```
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }
  return 0;
}
```

OUTPUT

```
Stack Operations (Linked List):
1. Push
2. Pop
3. Display
4. Peek
5. Exit
Enter your choice: 1
Enter value to push: 2555
2555 pushed into the stack.
Stack Operations (Linked List):
1. Push
2. Pop
3. Display
4. Peek
5. Exit
Enter your choice: 3
Stack elements: 2555
```

4. Write a C program to implement a queue using an array.

```
#include <stdio.h>
#define MAX 5
int queue[MAX];
int front = -1, rear = -1;
// Function to add an element to the queue
void enqueue(int val) {
    if(rear == MAX - 1) {
        printf("Queue Overflow! Cannot insert %d\n", val);
    } else {
        if(front == -1) front = 0;
        rear++;
        queue[rear] = val;
        printf("%d enqueued into the queue.\n", val);
    }
}
// Function to remove an element from the queue
void dequeue() {
    if(front == -1 || front > rear) {
        printf("Queue Underflow! No element to dequeue.\n");
    } else {
        printf("Dequeued element: %d\n", queue[front]);
        front++;
        if(front > rear) front = rear = -1; // Reset queue when empty
    }
}
// Function to display queue elements
void display() {
```

```
    if(front == -1 || front > rear) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for(int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
// Function to peek at the front element
void peek() {
    if(front == -1 || front > rear) {
        printf("Queue is empty.\n");
    } else {
        printf("Front element: %d\n", queue[front]);
    }
}
int main() {
    int choice, value;
    while(1) {
        printf("\nQueue Operations (Array):\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Peek\n5. 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:
                peek();
                break;
            case 5:
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

OUTPUT

```
Queue Operations (Array):
1. Enqueue
2. Dequeue
3. Display
4. Peek
5. Exit
Enter your choice: 1
Enter value to enqueue: 11
11 enqueued into the queue.
Queue Operations (Array):
1. Enqueue
2. Dequeue
3. Display
4. Peek
5. Exit
Enter your choice: 5
Exiting program.
```

5. Write a C program to implement a queue using a linked list.

```
#include <stdio.h>
#include <stdlib.h>
// Define node structure
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
// Function to add an element to the queue
```

```
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d enqueued into the queue.\n", value);
}
// Function to remove an element from the queue
void dequeue() {
    if(front == NULL) {
        printf("Queue Underflow! No element to dequeue.\n");
    } else {
        struct Node* temp = front;
        printf("Dequeued element: %d\n", front->data);
        front = front->next;
        free(temp);
        if(front == NULL) rear = NULL; // Reset rear if queue is empty
    }
}
// Function to peek at the front element
void peek() {
    if(front == NULL) {
        printf("Queue is empty.\n");
```

```
    } else {
        printf("Front element: %d\n", front->data);
    }
}
// Function to display queue elements
void display() {
    if(front == NULL) {
        printf("Queue is empty.\n");
    } else {
        struct Node* temp = front;
        printf("Queue elements: ");
        while(temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main() {
    int choice, value;
    while(1) {
        printf("\nQueue Operations (Linked List):\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Peek\n5. 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:
            peek();
            break;
        case 5:
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }
   return 0;
}
```

OUTPUT

```
Queue Operations (Linked List):
1. Enqueue
2. Dequeue
3. Display
4. Peek
```

```
5. Exit
Enter your choice: 1
Enter value to enqueue: 333
333 enqueued into the queue.
Queue Operations (Linked List):
1. Enqueue
2. Dequeue
3. Display
4. Peek
5. Exit
Enter your choice: 5
Exiting program.
```

1.2.3 Lab Practice Questions

1. Write a C Program to print the largest element in an array.
2. Write a C program to implement a stack using an array to store the marks of 5 students in Mathematics. The program should allow the user to:
   - Push a student's mark into the stack.
   - Pop a mark from the stack.
   - Display all marks in the stack.
   - Peek at the top mark.
   - Exit the program.
3. Write a C program to implement a stack using an array to store employee IDs of a company. The program should allow the user to:
   - Push an employee ID into the stack.
   - Pop an employee ID from the stack.
   - Display all employee IDs in the stack.
   - Peek at the top employee ID.

- Exit the program.

4. Write a C program to implement a stack to manage books in a library using linked lists. Each node stores a book code and performs the following operations: push, pop, display, and exit the program.

5. Write a C program to implement a queue using an array to manage patient IDs waiting for consultation. The program should allow the user to:

    - Enqueue: Add a new patient ID to the rear of the queue.
    - Dequeue: Remove the patient ID from the front of the queue.
    - Display: Show all patient IDs currently in the queue.
    - Peek: View the next patient ID to be attended without removing it.
    - Exit the program.

6. Write a C program to implement a queue using a linked list to maintain exam scores of students. The program should allow the user to:

    - Enqueue: Add a new student's exam score to the rear of the queue.
    - Dequeue: Remove the exam score from the front of the queue.
    - Display: Show all exam scores currently in the queue.
    - Peek: View the front exam score without removing it.
    - Exit the program.

EXPERIMENT 3

Linked List Operations - Singly, Doubly and Circular List

Objectives

- Understand the process of creating singly, doubly, and circular linked lists, including node structure and memory allocation.
- Implement insertion operations in linked lists at various positions (beginning, middle, and end) to manage dynamic data efficiently.
- Perform deletion operations to remove nodes correctly while maintaining the structure and integrity of the linked list.
- Apply traversal techniques to access, display, and verify all elements in singly, doubly, and circular linked lists.

1.3.1 Theory

In linked list implementations, the key operations include creation, insertion, deletion, and traversal, which form the foundation of working with singly, doubly, and circular linked lists. Creation involves defining a node structure with data and pointers and dynamically allocating memory to form the initial list. Insertion allows adding new nodes at different positions such as the beginning, end, or middle, depending on the requirement. Deletion deals with removing nodes by carefully updating the links to maintain the integrity of the list without breaking the chain. Traversal refers to sequentially visiting and displaying each node in the list, which may be done from head to tail in singly linked lists, in both directions in doubly linked lists, and circularly in circular linked lists where the last node points back to the head. Together, these operations demonstrate how linked lists can be efficiently managed and manipulated to store and process dynamic data.

1.3.2 Singly linked list

Node Creation:

Creation of a Linked List Node

```
struct node
        {
          int data;
          struct node *next;
        };
```

Here,

- **data** – stores the integer value or information of the node.
- ******struct node next* – acts as a pointer that stores the address of the next node in the list, thereby linking the nodes together.

Inserting Node at the Beginning of a linked list

Example

Consider a linked list containing the elements: 20 → 30 → 40 → NULL.

If we insert the value 100 at the beginning, it becomes the new head of the list.

After insertion, the updated linked list will be:

100 → 20 → 30 → 40 → NULL

Algorithm

1. Initialize a head pointer and set it to NULL.
2. Allocate memory and create a new node with the required data.
3. Link the new node so that it points to the current head.
4. Update the head pointer to make the new node the new head of the list.

Step 1: Initialize the head pointer as NULL

```
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
```

Step 2: Create a new node with the specified value

```
void addFirst(struct node **head, int val) {
   // allocate memory for the new node
   struct node *newNode = malloc(sizeof(struct node));
```

```
    newNode->data = val;
}
```

Step 3: Link the new node to the current head

```
void addFirst(struct node **head, int val) {
    // allocate memory for the new node
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    // make newNode point to the previous head
    newNode->next = *head;
}
```

Step 4: Update head to point to the new node

```
void addFirst(struct node **head, int val) {
    // allocate memory for the new node
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    // link the new node with current head
    newNode->next = *head;
    // set newNode as the new head
    *head = newNode;
}
```

Inserting a node at the end of a linked list

Example

Input:

Linked List: 10 → 20 → 30 → 40 → NULL

Element to Insert: 50

OUTPUT

Linked List: 10 → 20 → 30 → 40 → 50 → NULL

Algorithm

1. Initialize the head pointer and set it to NULL.
2. Create a new node with the given data and set its next pointer to NULL (since it will be added as the last node).
3. If the head is NULL (i.e., the linked list is empty), assign the new node as the head.
4. If the head is not NULL (i.e., the linked list already contains elements), traverse to the last node and update its next pointer to link with the new node.

Program for inserting a new node at the end of a singly linked list.

```
#include<stdio.h>
#include<stdlib.h>
// Define the structure for a node
struct node {
    int data;
    struct node *next;
};
// Function to insert a node at the end
void addLast(struct node **head, int val) {
    // Create a new node
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    newNode->next = NULL;
    // If the list is empty, the new node becomes the head
    if (*head == NULL) {
        *head = newNode;
    }
    // Otherwise, traverse to the last node and attach the new node
```

```
    else {
        struct node *lastNode = *head;
        while (lastNode->next != NULL) {
            lastNode = lastNode->next;
        }
        lastNode->next = newNode;
    }
}
// Function to display the linked list
void printList(struct node *head) {
    struct node *temp = head;
    while (temp != NULL) {
        printf("%d->", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    struct node *head = NULL;
    // Insert nodes at the end
    addLast(&head, 10);
    addLast(&head, 20);
    addLast(&head, 30);
    // Print the linked list
    printList(head);
    return 0;
}
```

Explanation

- A new node is dynamically allocated and initialized with the given value.
- If the list is empty, the new node is assigned as the head.
- If the list already has nodes, the program traverses to the last node and links the new node at the end.
- Finally, the list is printed to verify the insertion.

Algorithm: Inserting a Node at a Specific Position in a Linked List

In this operation, a new node with a given value (val) is inserted at a particular position (pos) in a singly linked list. The head pointer of the list, the desired position, and the value to be inserted are provided as inputs.

Example

- **Input:** val = 3, pos = 3
- **Output:** 1 → 2 → 3 → 4
- **Explanation:** The new node with value 3 is successfully placed at the 3rd position in the list.

Step-by-Step Implementation:

1. Start by initializing a pointer curr to the head of the list, and create a new node with the given value.
2. Move through the linked list using curr until you reach the node just before the desired position (pos - 1).
3. If curr->next is not NULL, set the next pointer of the new node to point to the node currently after curr.
4. Update the next pointer of curr so that it points to the newly created node.
5. Finally, return the head pointer of the linked list.

Program for implementing Inserting a Node at a Specific Position in a - Linked List

```
#include <stdio.h>

#include <stdlib.h>

// Structure for a node
```

```
struct Node {
    int val;
    struct Node* next;
};
// Function to create a new node
struct Node* createNode(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->val = x;
    newNode->next = NULL;
    return newNode;
}
// Function to insert a node at given position
struct Node* insertPos(struct Node* head, int pos, int val) {
    if (pos < 1)
        return head;
    // If insertion is at the beginning
    if (pos == 1) {
        struct Node* newNode = createNode(val);
        newNode->next = head;
        return newNode;
    }
    struct Node* curr = head;
    // Traverse to node just before required position
    for (int i = 1; i < pos - 1 && curr != NULL; i++) {
        curr = curr->next;
    }
```

```
    // If position is greater than number of nodes
    if (curr == NULL)
        return head;
    struct Node* newNode = createNode(val);
    // Adjust the pointers
    newNode->next = curr->next;
    curr->next = newNode;
    return head;
}
// Function to print linked list
void printList(struct Node* head) {
    struct Node* curr = head;
    while (curr != NULL) {
        printf("%d", curr->val);
        if (curr->next != NULL) {
            printf(" -> ");
        }
        curr = curr->next;
    }
    printf("\n");
}
// Driver Code
int main() {
    // Creating the list 1->2->4
    struct Node* head = createNode(1);
    head->next = createNode(2);
```

```
    head->next->next = createNode(4);
    int val = 3, pos = 3;

    head = insertPos(head, pos, val);

    printList(head);

    return 0;
}
```

OUTPUT

```
1 -> 2 -> 3 -> 4
```

Deleting a node from singly linked list

In a singly linked list, deletion refers to removing a specific node while maintaining the structure of the list. Depending on the position of the node to be deleted, deletion can happen at the beginning, at the end, or at any given position. The process involves adjusting the link (pointer) of the previous node so that it bypasses the node to be removed, and then freeing the memory of the deleted node.

Example

Linked List: 10 → 20 → 30 → NULL

Input

Element to delete: 20

OUTPUT

10 → 30 → NULL

Types of Deletion in a Linked List

Deletion in a linked list can be performed in different ways depending on the position of the node to be removed. The three common cases are:

- Deletion at the Beginning
- Deletion at a Specific Position
- Deletion at the End

1. Deletion at the Beginning of the Linked List

This operation removes the very first node of the list.

Step-by-Step Approach:

- **Check if the list is empty:** If the head is NULL, then no deletion is possible.
- **Update the head pointer:** Change the head to point to the second node (head = head->next).
- **Delete the original first node:** Free or release the memory of the old head (in languages like C where manual memory management is required).

2. Deletion at a Specific Position in the Linked List

This method removes a node from a particular position, which could be the first, middle, or last node.

Step-by-Step Approach:

- **Validate the position:** Ensure that the given position is within the valid range of the list.
- **Locate the previous node:** Traverse the list until you reach the node just before the one to be deleted (position n-1).
- **Update the link:** Set the next pointer of the previous node to skip the target node and point to the next one.
- **Delete the node:** Release the memory of the target node, making it unreferenced.

3. Deletion at the End of the Linked List

This operation deletes the last node of the list.

Step-by-Step Approach:

- **Check if the list is empty:** If the head is NULL, then there is nothing to delete.
- **If only one node exists:** Set the head to NULL to make the list empty.
- **Find the second last node:** Traverse the list until reaching the node whose next points to the last node.
- **Update the link:** Set the next pointer of this second last node to NULL, disconnecting the last node.
- **Delete the last node:** Free or deallocate its memory as needed.

1.3.3 Doubly Linked List

A doubly linked list is a variation of the linked list in which every node consists of three components: the data field, a pointer to the next node, and a pointer to the previous

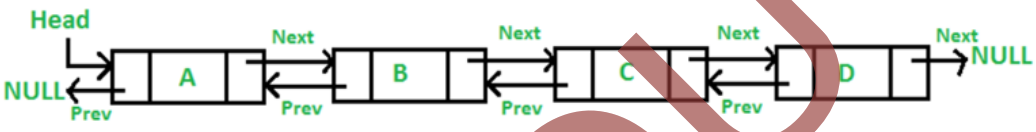
node. Unlike a singly linked list, which allows traversal only in the forward direction, the doubly linked list provides an additional pointer (prev) that makes it possible to move both forward and backward through the list. This bidirectional navigation is the key feature that differentiates it from a singly linked list.

In this discussion, we will explore how a doubly linked list is implemented in C, examine its structure, and understand the basic operations that can be carried out on it.

Representation of a Doubly Linked List in C

A doubly linked list in C is generally represented using a pointer to the head node (the first element of the list). Each node contains the following three parts:

- Data: Stores the actual information to be processed.
- Next: A pointer that refers to the next node in the sequence.
- Prev: A pointer that refers to the previous node, enabling backward traversal.



To build a doubly linked list in C, the first step is to define a node structure. This structure must include three members: the data field, a pointer to the next node, and a pointer to the previous node. With this structure defined, we can then create new nodes and link them together to form the doubly linked list. The use of structures in C allows us to group these different data elements into a single custom data type for efficient representation.

To declare a node structure for a doubly linked list, you can use the following format:

```
struct Node {
    int data;          // Stores the value
    struct Node* next;    // Pointer to the next node
    struct Node* prev;    // Pointer to the previous node
};
```

Algorithm

Basic Operations on a Doubly Linked List in C

In a doubly linked list, each node is connected in both directions, which allows insertion, deletion, and traversal operations to be performed more flexibly than in a singly linked list. The fundamental operations include insertion, deletion, and traversal.

1. Insertion in a Doubly Linked List

Insertion requires updating both the next and prev pointers. There are three main cases:

- **At the Beginning:**
  - Create a new node with the given data.
  - If the list is empty, set both next and prev to NULL and update head.
  - If not empty, link the new node's next to the current head, update head's prev, and make the new node the new head.
  - Time Complexity: O(1).
- **At the End:**
  - Create a new node.
  - If the list is empty, update head.
  - Otherwise, traverse to the last node, adjust pointers (last->next = newNode and newNode->prev = last), and mark newNode as the end.
  - Time Complexity: O(n), O(1) if a tail pointer is maintained.
- **At a Specific Position:**
  - Validate the position.
  - If at the start, follow the beginning case; if at the end, follow the end case.
  - Otherwise, traverse to the desired position, adjust next and prev pointers of the surrounding nodes, and insert the new node.
  - Time Complexity: O(n).

2. Deletion in a Doubly Linked List

Deletion also has three scenarios, where proper adjustment of next and prev links is required:

- **At the Beginning:**
  - If the list is empty, nothing to delete.
  - If only one node exists, set head to NULL.

- Otherwise, move head to the second node, update its prev to NULL, and free the old head.
- Time Complexity: O(1).

♦ **At the End:**

- If the list is empty, no deletion.
- If only one node exists, set head to NULL.
- Otherwise, traverse to the last node, unlink it from the second-last node (prev->next = NULL), and free it.
- Time Complexity: O(n), or O(1) with a tail pointer.

♦ **At a Specific Position:**

- Validate the position.
- If it is the first node, use the beginning case; if the last node, use the end case.
- Otherwise, adjust pointers of the previous and next nodes to skip the target node, then free it.
- Time Complexity: O(n).

3. Traversal in a Doubly Linked List

Traversal means visiting nodes one by one, either forward or backward.

♦ Forward Traversal: Start from head and move through next until reaching NULL.

♦ Reverse Traversal: Start from the tail and move through prev until reaching NULL.

♦ Time Complexity: O(n) in both cases.

C Program for Doubly Linked List Implementation

The following program illustrates how to implement and perform fundamental operations on a doubly linked list. It covers:

♦ Insertion: at the beginning, at the end, and at a specific position.

♦ Deletion: from the beginning, from the end, and from a specific position.

♦ Traversal: both forward and reverse directions.

This provides a complete demonstration of how to manage a doubly linked list in C.

C Program to Implement Doubly Linked List

```
#include <stdio.h>

#include <stdlib.h>

// Defining a node in the doubly linked list

typedef struct Node {

    int data;

    struct Node* next;

    struct Node* prev;

} Node;

// Function to create a new node with the given data

Node* createNode(int data) {

    Node* newNode = (Node*)malloc(sizeof(Node));

    newNode->data = data;

    newNode->next = NULL;

    newNode->prev = NULL;

    return newNode;

}

// Insert a node at the beginning

void insertAtBeginning(Node** head, int data) {

    Node* newNode = createNode(data);

    if (*head == NULL) {

        *head = newNode;

        return;

    }

    newNode->next = *head;

    (*head)->prev = newNode;

    *head = newNode;

}
```

```
// Insert a node at the end
void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
// Insert a node at a given position
void insertAtPosition(Node** head, int data, int position) {
    if (position < 1) {
        printf("Invalid position. Must be >= 1.\n");
        return;
    }
    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }
    Node* newNode = createNode(data);
    Node* temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
```

```
    }
    if (temp == NULL) {
        printf("Position exceeds list length.\n");
        return;
    }
    newNode->next = temp->next;
    newNode->prev = temp;
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
}
// Delete a node from the beginning
void deleteAtBeginning(Node** head) {
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    Node* temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }
    free(temp);
}
// Delete a node from the end
void deleteAtEnd(Node** head) {
    if (*head == NULL) {
```

```
        printf("The list is empty.\n");
        return;

    }
    Node* temp = *head;
    if (temp->next == NULL) {
        *head = NULL;
        free(temp);
        return;
    }
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->prev->next = NULL;
    free(temp);
}
// Delete a node from a specific position
void deleteAtPosition(Node** head, int position) {
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    Node* temp = *head;
    if (position == 1) {
        deleteAtBeginning(head);
        return;
    }
    for (int i = 1; temp != NULL && i < position; i++) {
        temp = temp->next;
```

```
    }
    if (temp == NULL) {
        printf("Invalid position.\n");
        return;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
}
// Print the list in forward direction
void printListForward(Node* head) {
    Node* temp = head;
    printf("Forward List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Print the list in reverse direction
void printListReverse(Node* head) {
    Node* temp = head;
    if (temp == NULL) {
        printf("The list is empty.\n");
```

```
        return;
    }

    while (temp->next != NULL) {
        temp = temp->next;
    }
    printf("Reverse List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
}
int main() {
    Node* head = NULL;
    // Demonstrating insertions
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtBeginning(&head, 5);
    insertAtPosition(&head, 15, 2); // List: 5 15 10 20
    printf("After Insertions:\n");
    printListForward(head);
    printListReverse(head);
    // Demonstrating deletions
    deleteAtBeginning(&head); // List: 15 10 20
    deleteAtEnd(&head);       // List: 15 10
    deleteAtPosition(&head, 2); // List: 15
    printf("After Deletions:\n");
    printListForward(head);
```

```
    return 0;
}
```

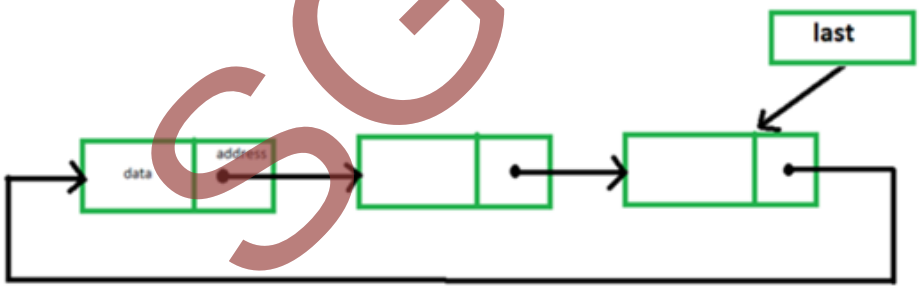
OUTPUT

```
After Insertions:
Forward List: 5 15 10 20
Reverse List: 20 10 15 5
After Deletions:
Forward List: 15
```

1.3.4 Circular List

Operations on a Circular Linked List in C

A circular linked list is a variation of a linked list in which each node connects to the next, and the last node links back to the first node, forming a circle. Unlike a regular singly linked list where the last node points to NULL, in a circular linked list the last node points to the head node. This allows traversal of the list starting from any node without reaching an end.



1. **Insertion at the beginning :** When inserting a node at the start of a circular linked list, the new node becomes the first element. The next pointer of the last node is updated to point to this new node, while the new node's next pointer is linked to the old first node.

The following code demonstrates this operation:

C Program to Insert Nodes at the Beginning of a Circular Linked List

```
#include <stdio.h>

#include <stdlib.h>
```

```
// Structure definition for a circular linked list node
struct Node {
    int data;
    struct Node* next;
};
// Pointer to keep track of the last node
struct Node* last = NULL;
// Function to insert a node at the front of the circular linked list
void insertAtFront(int value) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    // Case 1: If the list is empty
    if (last == NULL) {
        newNode->next = newNode; // Points to itself
        last = newNode;          // Last points to the new node
    }
    // Case 2: If the list already contains nodes
    else {
        newNode->next = last->next; // New node points to the first node
        last->next = newNode;       // Last node points to the new node
    }
}
// Function to display the circular linked list
```

```
void displayList() {
    // If the list is empty

    if (last == NULL) {
        printf("\nThe list is empty.\n");
        return;
    }
    // Start from the first node
    struct Node* temp = last->next;
    // Traverse until we reach the first node again
    printf("\nCircular Linked List: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != last->next);
    printf("\n");
}
// Driver function
int main() {
    // Insert elements at the beginning
    insertAtFront(10);
    insertAtFront(20);
    insertAtFront(30);
    // Display the list
    displayList();
```

```
    return 0;
}
```

OUTPUT

```
Circular Linked List:
30 20 10
```

2. Insertion at the End

Insertion at the end of a circular linked list means adding a new node as the last element. After insertion, the next pointer of the newly added node will point to the first node, while the current last node will update its next pointer to refer to this new node. The last pointer is then updated to the newly inserted node.

C program to perform insertion at the end of a circular linked list

```
#include <stdio.h>
#include <stdlib.h>
// Structure for a circular linked list node
struct node {
    int data;
    struct node* next;
};
// Pointer to the last node of the list
struct node* last = NULL;
// Function to insert a node at the end
void insertAtEnd(int value)
{
    // Create a new node
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    temp->data = value;
```

```
    // If the list is empty, initialize it
    if (last == NULL) {

        temp->next = temp;

        last = temp;

    }

    // Otherwise, adjust pointers for circular linking

    else {

        temp->next = last->next;

        last->next = temp;

        last = temp;

    }

}
// Function to display the circular linked list

void displayList()

{

    if (last == NULL) {

        printf("\nList is empty\n");

        return;

    }

    struct node* temp = last->next;

    printf("\nCircular Linked List: ");

    do {

        printf("%d ", temp->data);

        temp = temp->next;
```

```
    } while (temp != last->next);
}

// Driver code
int main()
{
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtEnd(30);
    // Print the list
    displayList();
    return 0;
}
```

OUTPUT

```
Circular Linked List:
10 20 30
```

3. Insertion After a Specific Element

This operation inserts a new node immediately after a given node in the circular linked list. If the specified element is the last node, the new node becomes the new last node, maintaining the circular property of the list.

C program to insert a node after a specific element in a circular linked list

```
#include <stdio.h>
#include <stdlib.h>
// Structure of a circular linked list node
struct node {
    int data;
```

```
    struct node* next;
};

// Global pointer to the last node
struct node* last = NULL;
// Function to insert a node at the end (helper function)
void insertAtEnd()
{
    int value;
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    printf("\nEnter data to be inserted: ");
    scanf("%d", &value);
    temp->data = value;
    // If the list is empty, initialize it
    if (last == NULL) {
        temp->next = temp;
        last = temp;
    }
    else {
        temp->next = last->next;
        last->next = temp;
        last = temp;
    }
}
// Function to insert a new node after a given element
```

```
void insertAfter()
{
    int value, newData;
    printf("\nEnter the element after which you want to insert: ");
    scanf("%d", &value);
    struct node* temp = last->next;
    // Traverse the list to find the element
    do {
        if (temp->data == value) {
            struct node* newNode = (struct node*)malloc(sizeof(struct node));
            printf("Enter data to be inserted: ");
            scanf("%d", &newData);
            newNode->data = newData;
            newNode->next = temp->next;
            temp->next = newNode;
            // Update last pointer if inserted after the last node
            if (temp == last) {
                last = newNode;
            }
            return;
        }
        temp = temp->next;
    } while (temp != last->next);
    printf("Element %d not found in the list.\n", value);
```

```
}
// Function to display the list

void displayList()

{

   if (last == NULL) {

      printf("\nList is empty.\n");

      return;

   }
   struct node* temp = last->next;

   printf("\nCircular Linked List: ");

   do {

      printf("%d ", temp->data);

      temp = temp->next;

   } while (temp != last->next);

   printf("\n");

}

// Driver Code

int main()

{

   // Initialize list with three nodes

   insertAtEnd();

   insertAtEnd();

   insertAtEnd();

   // Insert after a specific node
```

```
    insertAfter();
    // Print final list

    displayList();

    return 0;

}
```

OUTPUT

```
Enter data to be inserted: 10

Enter data to be inserted: 20

Enter data to be inserted: 30

Enter the element after which you want to insert: 20

Enter data to be inserted: 25

Circular Linked List:

10 20 25 30
```

4. Deletion of the First Node

In this operation, the first element of the circular linked list is removed. Since the list is circular, the 'next' pointer of the last node is updated to point to the second node, effectively removing the first node from the sequence.

C program to delete the first node of a circular linked list

```
#include <stdio.h>

#include <stdlib.h>

// Structure of a circular linked list node

struct node {

    int data;

    struct node* next;

};
```

```
// Pointer to the last node in the list
struct node* last = NULL;

// Function to insert a node at the end
void insertAtEnd(int value)
{
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    temp->data = value;
    // Case 1: If the list is empty
    if (last == NULL) {
        temp->next = temp;
        last = temp;
    }
    // Case 2: Add new node after last and update last
    else {
        temp->next = last->next;
        last->next = temp;
        last = temp;
    }
}
// Function to delete the first node
void deleteFirst()
{
    // If list is empty
    if (last == NULL) {
```

```
        printf("\nList is empty.\n");
        return;
    }
    struct node* temp = last->next;
    // If there is only one node
    if (last == last->next) {
        free(temp);
        last = NULL;
    }
    else {
        // Make last->next skip the first node
        last->next = temp->next;
        free(temp);
    }
}
// Function to display the list
void displayList()
{
    if (last == NULL) {
        printf("\nList is empty\n");
        return;
    }
    struct node* temp = last->next;
    printf("\nCircular Linked List: ");
```

```
    do {
        printf("%d ", temp->data);

        temp = temp->next;
    } while (temp != last->next);
    printf("\n");
}
// Driver Code
int main()
{
    // Create list with three nodes
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtEnd(30);
    printf("Before deletion:");
    displayList();
    // Delete first node
    deleteFirst();
    printf("\nAfter deletion:");
    displayList();
    return 0;
}
```

OUTPUT

```
Before deletion:

Circular Linked List: 10 20 30

After deletion:

Circular Linked List: 20 30
```

5. Deletion of the Last Node

In this operation, the last element of the circular linked list is removed. To achieve this, the second-last node is updated so that its next pointer refers back to the first node. This effectively removes the last node from the list.

C program to delete the last node of a circular linked list

```
#include <stdio.h>

#include <stdlib.h>

// Structure for a circular linked list node

struct node {

    int data;

    struct node* next;

};

// Pointer to the last node in the list

struct node* last = NULL;

// Function to insert a node at the end

void insertAtEnd(int value)

{

    struct node* temp = (struct node*)malloc(sizeof(struct node));

    temp->data = value;

    // Case 1: If list is empty
```

```
    if (last == NULL) {
        temp->next = temp;
        last = temp;
    }
    // Case 2: Add after last node and update last
    else {
        temp->next = last->next;
        last->next = temp;
        last = temp;
    }
}
// Function to delete the last node
void deleteLast()
{
    // Case 1: If list is empty
    if (last == NULL) {
        printf("\nList is empty.\n");
        return;
    }
    struct node* temp = last->next;
    // Case 2: If there is only one node
    if (last == last->next) {
        free(last);
        last = NULL;
```

```
        return;
    }
    // Case 3: Traverse to the second last node
    while (temp->next != last) {
        temp = temp->next;

    }
    // Update second last node to point to first node
    temp->next = last->next;
    free(last);
    last = temp;
}
// Function to display the list
void displayList()
{
    if (last == NULL) {
        printf("\nList is empty\n");
        return;
    }
    struct node* temp = last->next;
    printf("\nCircular Linked List: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != last->next);
```

```
    printf("\n");
}
// Driver Code
int main()
{
    // Create a list with three nodes
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtEnd(30);
    printf("Before Deletion:");
    displayList();
    // Delete last node
    deleteLast();
    printf("\nAfter Deletion:");
    displayList();
    return 0;
}
```

OUTPUT

```
Before Deletion:
Circular Linked List: 10 20 30
After Deletion:
Circular Linked List: 10 20
```

6. Deletion at a Specific Position

This operation removes a node from a given position in a circular linked list. The list is traversed until the target node is found, and then the links are adjusted so that the previous node points to the next node, effectively removing the target node from the structure.

C program to delete a node at a given position in a circular linked list

```
#include <stdio.h>
#include <stdlib.h>
// Structure for a circular linked list node
struct node {
    int data;
    struct node* next;
};
// Pointer to the last node of the list
struct node* last = NULL;
// Function to insert a node at the end
void insertAtEnd(int value)
{
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    temp->data = value;
    // Case 1: If list is empty
    if (last == NULL) {
        temp->next = temp;
        last = temp;
    }
    // Case 2: Insert after last node and update last
    else {
        temp->next = last->next;
        last->next = temp;
```

```
        last = temp;
    }
}
// Function to delete a node at a given position
void deleteAtPosition(int pos)
{
    // Case 1: If list is empty
    if (last == NULL) {
        printf("\nList is empty.\n");
        return;
    }
    struct node *temp = last->next, *prev;
    // Case 2: If only one node
    if (last == last->next) {
        if (pos == 1) {
            free(last);
            last = NULL;
        } else {
            printf("\nInvalid position.\n");
        }
        return;
    }
    // Case 3: If deleting the first node
    if (pos == 1) {
```

```
        last->next = temp->next; // last points to second node
        free(temp);
        return;
    }
    // Traverse to the node before the target position
    int i;
    for (i = 1; i < pos - 1 && temp->next != last->next; i++) {
        temp = temp->next;
    }
    // If position is out of range
    if (temp->next == last->next) {
        printf("\nInvalid position.\n");
        return;
    }
    // Adjust links to skip the target node
    prev = temp->next;
    temp->next = prev->next;
    // If the last node is being deleted, update last
    if (prev == last)
        last = temp;
    free(prev);
}
// Function to display the list
void displayList()
```

```
{
    if (last == NULL) {
        printf("\nList is empty\n");
        return;
    }
    struct node* temp = last->next;
    printf("\nCircular Linked List: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != last->next);
    printf("\n");
}
// Driver Code
int main()
{
    // Create a list
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtEnd(30);
    insertAtEnd(40);
    printf("Original List:");
    displayList();
    // Delete node at given position
```

```
    deleteAtPosition(3);
    printf("\nAfter Deletion at Position 3:");
    displayList();
    return 0;
}
```

OUTPUT

```
Original List:
Circular Linked List: 10 20 30 40
After Deletion at Position 3:
Circular Linked List: 10 20 40
```

1.3.5 Lab Practice Questions

1. Develop a C program to manage a list of employee IDs using a singly linked list.

   Perform the following operations

   a. Insert the following IDs at the beginning : 105, 102

   b. Insert the following IDs at the end : 110, 115

   c. Insert an ID 108 at position 3

   d. Delete the node at position 2

   e. Display the final list

2. Write a C program to create a singly linked list of students marks. Perform the following operations:

   a. Insert marks 45, 50 at the beginning

   b. Insert marks 65, 70 at the end

   c. Insert mark 60 at position 3

   d. Delete the first node

   e. Display the list after each operation.

3. Develop a C program to create a doubly linked list of integers & perform:

a. Insert 5, 3 at the beginning

b. Insert 15, 20 at the end

c. Insert 10 at position 3

d. Delete the first node

e. Delete the last node

f. Delete the node at position 2

g. Traverse the list in both forward & reverse direction.

4. Write a C program to implement a doubly linked list & perform the following operations:

   a. Insert the elements 20, 10 at the beginning

   b. Insert the elements 30, 40 at the end

   c. Insert the element 25 at position 3

   d. Delete a node from beginning

   e. Delete a node from position 2

   f. Display the list after each operation.

5. Write a C program to create a circular linked list using the elements:

   10, 20, 30, 40

   Then perform the following operations:

   a. Insert 5 at the beginning

   b. Insert 50 at the end

   c. Insert 25 at position 4

   d. Delete the first node

   e. Delete the last node

   f. Delete the node at position 3

   g. Display the final circular list.

6. Develop a C program to:
    1. Create a circular linked list with initial elements: 3, 6, 9
    2. Perform the following operations:
        a. Insert 1 at the beginning
        b. Insert 12 at the end
        c. Insert 7 at position 3
        d. Delete first node
        e. Delete last node
        f. Delete the node at position 2
3. Display the list after all operations.

Experiment 4

Implementation of Binary Search Tree (BST) with Inorder, Preorder, and Postorder Traversals

Objectives

- To understand the concept of Binary Search Tree (BST).
- To implement basic operations on a BST.
- To perform tree traversals: inorder, preorder, and postorder.

1.4.1 Theory

1.4.1.1 Binary Search Tree (BST)

A Binary Search Tree is a node-based binary tree data structure where each node has the following properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both left and right subtrees must also be binary search trees.

1.4.1.2 Tree Traversals

Tree traversal refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way.

1. Inorder Traversal (Left, Root, Right)

- Visit the left subtree.
- Visit the root node.
- Visit the right subtree.
- Output for BST: sorted order of elements.

2. Preorder Traversal (Root, Left, Right)

- Visit the root node.

- Visit the left subtree.
- Visit the right subtree.
- Used for copying the tree, and for prefix expression.

3. **Postorder Traversal (Left, Right, Root)**
   - Visit the left subtree.
   - Visit the right subtree.
   - Visit the root node.
   - Used to delete or free nodes in the tree

1.4.2 Implementation

1. Program to perform creation, insertion, and deletion in a Binary Search Tree (BST) and display using:
   - Inorder Traversal
   - PreorderTraversal
   - Postorder Traversal

```
#include <stdio.h>
#include <stdlib.h>
// Structure of a node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
//Create New Node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
```

```
    return newNode;
}
//Insert Node into BST
struct Node* insert(struct Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }
    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
    return root;
}
//Find Minimum Node (Used in Deletion)
struct Node* findMin(struct Node* root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}
// Delete Node from BST
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL)
        return root;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
```

```
        // Node with 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;
        }
        // Node with two children
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

//Inorder Traversal

```
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

//Preorder Traversal

```
void preorder(struct Node* root) {
```

```
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
//Postorder Traversal
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
int main() {
    struct Node* root = NULL;
    int n, i, value;
    //  Tree Creation
    printf("Enter number of nodes to create BST: ");
    scanf("%d", &n);
    printf("Enter %d node values:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &value);
        root = insert(root, value);
    }
    // Traversals after Creation
    printf("\nTraversals after creation:\n");
    printf("1. Inorder: ");
```

```
inorder(root);
printf("\n2. Preorder: ");
preorder(root);
printf("\n3. Postorder: ");
postorder(root);
// Insert a Node
printf("\n\nEnter value to insert: ");
scanf("%d", &value);
root = insert(root, value);
printf("\nTraversals after insertion of %d:\n", value);
printf("1. Inorder: ");
inorder(root);
printf("\n2. Preorder: ");
preorder(root);
printf("\n3. Postorder: ");
postorder(root);
// Delete a Node
printf("\n\nEnter value to delete: ");
scanf("%d", &value);
root = deleteNode(root, value);
printf("\nTraversals after deletion of %d:\n", value);
printf("1. Inorder: ");
inorder(root);
printf("\n2. Preorder: ");
preorder(root);
printf("\n3. Postorder: ");
postorder(root);
```

```
    return 0;
}
```

OUTPUT

```
Enter number of nodes to create BST: 5
Enter 5 node values:
50 30 70 20 40
Traversals after creation:
1. Inorder: 20 30 40 50 70
2. Preorder: 50 30 20 40 70
3. Postorder: 20 40 30 70 50
Enter value to insert: 60
Traversals after insertion of 60:
1. Inorder: 20 30 40 50 60 70
2. Preorder: 50 30 20 40 70 60
3. Postorder: 20 40 30 60 70 50
Enter value to delete: 30
Traversals after deletion of 30:
1. Inorder: 20 40 50 60 70
2. Preorder: 50 40 20 70 60
3. Postorder: 20 40 60 70 50
```


1.4.3 Lab Practice Questions

1. Write a program to create a BST using the following data:

   Input Data: 50, 30, 70, 20, 40, 60, 80

   Perform and display:

   - Inorder traversal
   - Preorder traversal

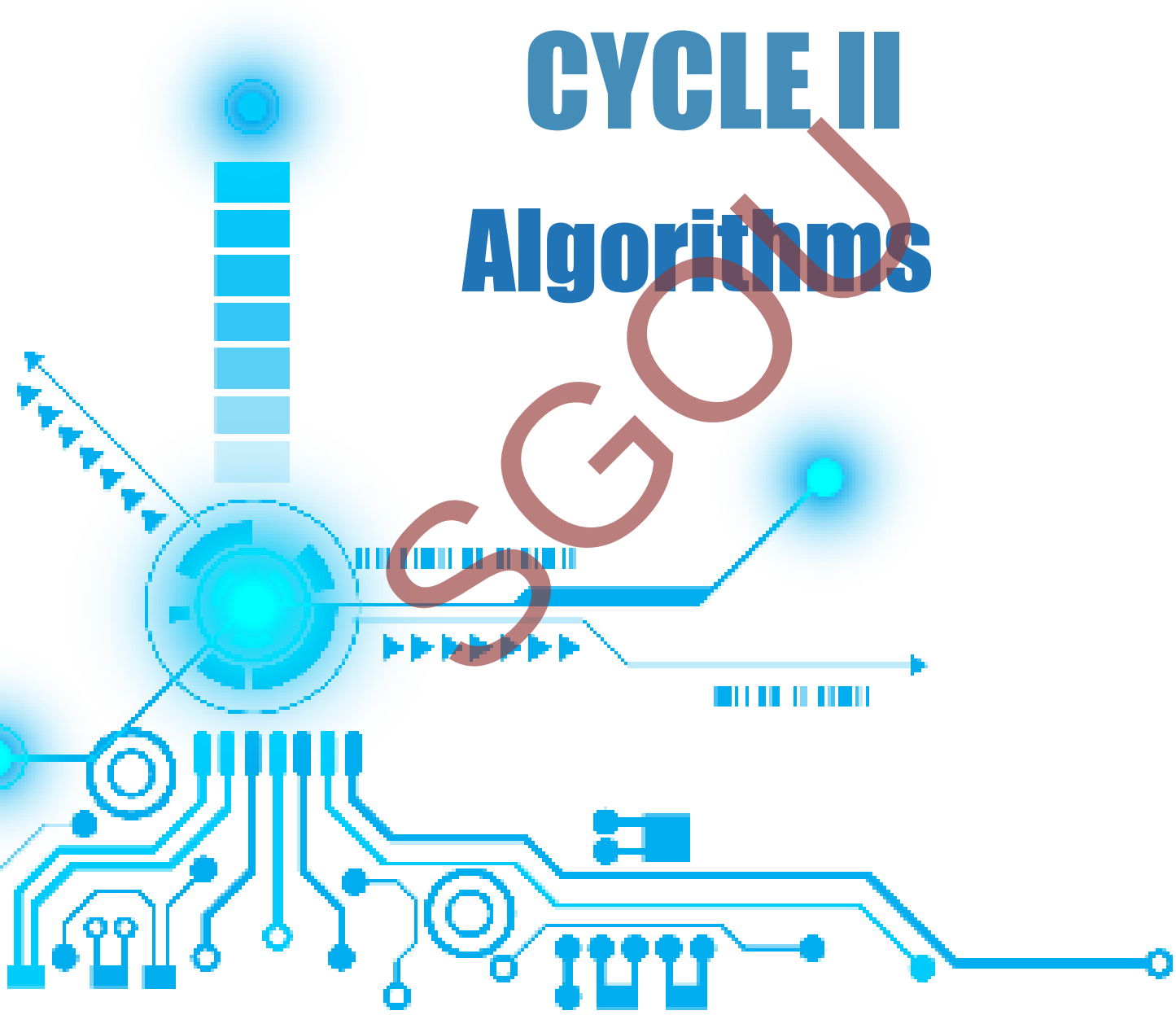
- Postorder traversal

2. Given the operations below, trace the tree and print final traversals:
    - Insert: 70, 50, 90, 30, 60, 80, 100
    - Insert: 55
    - Delete: 50
    - Insert: 65
    - Delete: 90

Then print Inorder, Preorder, Postorder Traversals.

CYCLE II

Algorithms



EXPERIMENT 1

Linear Search and Binary Search

Objectives

- To understand the concept of searching techniques in arrays and their applications.
- To implement and analyze the working of the Linear Search algorithm for finding an element in an array.
- To implement and evaluate the Binary Search algorithm on sorted arrays for efficient searching.

2.1.1 Theory

Searching is one of the most fundamental operations in computer science. It refers to the process of finding the location of a particular element (called the key) in a collection of data such as an array. There are different searching techniques, and the choice of method depends on whether the array is sorted or unsorted. Two commonly used searching algorithms are Linear Search and Binary Search.

2.1.1.1 Linear Search

Linear Search is the simplest searching technique. In this method, each element in the array is compared one by one with the key element until the element is found or the end of the array is reached.

Characteristics:

- Can be applied to unsorted arrays.
- Simple to implement but inefficient for large datasets.
- Time Complexity: O(n) in the worst case, where n is the number of elements.

Working Principle:

- Start from the first element of the array.
- Compare the current element with the key.
- If a match is found, return the position.
- If not, move to the next element.

- If the end of the array is reached without a match, report that the element is not present.

2.1.1.2 Binary Search

Binary Search is a more efficient searching technique, but it requires the array to be sorted. Instead of checking elements one by one, it divides the search interval into halves. Table 2.1.1 explained the comparison between Linear Search and Binary Search.

Characteristics:

- Requires the array to be sorted.
- Much faster than linear search for large arrays.
- Time Complexity: O(log n) in the worst case.

Working Principle:

- Start with the middle element of the array.
- If the key matches the middle element, the search is successful.
- If the key is smaller than the middle element, search in the left half of the array.
- If the key is greater, search in the right half.
- Repeat the process until the element is found or the search interval becomes empty.

Table 2.1.1 Comparison between Linear Search and Binary Search

Feature	Linear Search	Binary Search
Input Requirement	Works on both sorted & unsorted arrays	Works only on sorted arrays
Approach	Sequential comparison	Divide and conquer (halving)
Best Case	O(1) (first element is the key)	O(1) (middle element is the key)
Worst Case	O(n)	O(log n)
Efficiency	Less efficient for large datasets	Very efficient for large datasets

2.1.2 Implementation

1. Write a Python program to search for an element in an array using linear search.

Source code:

```
def linear_search(arr, key):
    for i in range(len(arr)):
        if arr[i] == key:
            return i   # return index if found
    return -1          # return -1 if not found
# Example array
arr = [10, 25, 30, 45, 50]
key = 30
result = linear_search(arr, key)
if result != -1:
    print(f"Element {key} found at index {result}")
else:
    print(f"Element {key} not found")
```

OUTPUT

```
Element 30 found at index 2
```

2. Write a Python program to search for an element in a sorted array using binary search.

Source code:

```
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1
        while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
```

```
            low = mid + 1
        else:
            high = mid - 1
    return -1
# Sorted array
arr = [5, 15, 25, 35, 45, 55]
key = 35
result = binary_search(arr, key)
if result != -1:
    print(f"Element {key} found at index {result}")
else:
    print(f"Element {key} not found")
```

OUTPUT

```
Element 35 found at index 3
```

3. Write a Python program to search for an element in a sorted array using both linear search and binary search, and compare the results.

Source Code:

```
def linear_search(arr, key):
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
```

```
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1
arr = [2, 4, 6, 8, 10, 12, 14]
key = 10
# Using Linear Search
res1 = linear_search(arr, key)
# Using Binary Search
res2 = binary_search(arr, key)
print(f"Linear Search: Element {key} found at index {res1}")
print(f"Binary Search: Element {key} found at index {res2}")
```

OUTPUT

```
Linear Search: Element 10 found at index 4
Binary Search: Element 10 found at index 4
```

2.1.3 Lab Practice Questions

1. Write a program in Python to implement linear search to find whether a given number exists in an array of 10 integers entered by the user.

2. Implement a Python program to perform binary search on a sorted array of integers. The program should print the position of the element if it is found, otherwise print "Element not found".

3. Write a Python program that asks the user to enter a sorted list of numbers and a key element, then searches for the key using both linear search and binary search, and displays the number of comparisons made in each case.

Experiment 2

Sorting Algorithms

Objectives

- Implement Quick Sort algorithm on an array.
- Implement Merge Sort algorithm on an array.
- Test sorting with different input cases.

2.2.1 Theory

Quick Sort

Quick Sort is a divide-and-conquer algorithm. It works by selecting a pivot element, partitioning the array into two parts (elements smaller than the pivot and elements greater than the pivot), and then recursively sorting the two parts. It is efficient in practice with an average time complexity of O(n log n).

Merge Sort

Merge Sort is also a divide-and-conquer algorithm. It works by dividing the array into two halves, recursively sorting them, and then merging the sorted halves into a single sorted array. Merge Sort guarantees a time complexity of O(n log n), but it requires additional memory for merging.

2.2.1.1 Algorithm: Quick Sort

1. Start with an array of n elements.
2. If the array has only one element, return (already sorted).
3. Select a pivot element from the array.
4. Partition the array so that:
    - Elements smaller than pivot are placed on the left.
    - Elements greater than pivot are placed on the right.
5. Recursively apply Quick Sort to the left part of the array.
6. Recursively apply Quick Sort to the right part of the array.
7. Continue until all subarrays are sorted.
8. End.

Program

```
#include <stdio.h>
// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choosing last element as pivot
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        // Recursively sort elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
// Function to print array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
// Main function
int main() {
    int arr[50], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    quickSort(arr, 0, n - 1);
    printf("Sorted array (Quick Sort): ");
    printArray(arr, n);
    return 0;
}
```

OUTPUT

```
Enter number of elements: 6
Enter elements: 34 7 23 32 5 62
Sorted array (Quick Sort): 5 7 23 32 34 62
```


2.2.1.2 Algorithm: Merge Sort

1. Start with an array of n elements.
2. If the array has only one element, return (already sorted).
3. Divide the array into two halves.

4. Recursively apply Merge Sort to the left half.
5. Recursively apply Merge Sort to the right half.
6. Merge the two sorted halves into a single sorted array.
7. Continue until the whole array is merged and sorted.
8. End.

Program

```
#include <stdio.h>
// Merge function
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    // Merge the arrays back into arr[]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
```

```
        k++;
    }
    // Copy remaining elements
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Merge Sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        // Merge sorted halves
        merge(arr, left, mid, right);
    }
}
// Function to print array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
```

```
        printf("%d ", arr[i]);
    printf("\n");
}
// Main function
int main() {
    int arr[50], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    mergeSort(arr, 0, n - 1);
    printf("Sorted array (Merge Sort): ");
    printArray(arr, n);
    return 0;
}
```

OUTPUT

```
Enter number of elements: 8
Enter elements: 12 11 13 5 6 7 1 10
Sorted array (Merge Sort): 1 5 6 7 10 11 12 13
```

2.2.2 Lab Practice Questions

1. Implement Quick Sort to sort an array of integers.
2. Implement Merge Sort to sort an array of integers.
3. Test both algorithms with the following input cases:
    - Already sorted array

- Reverse sorted array
- Random order array
- Array with duplicate elements

4. Compare the outputs of Quick Sort and Merge Sort for the same input.

Experiment 3

Implementation of Breadth First Search (BFS) and Depth First Search (DFS) Traversals of a Graph

Objectives

- To understand the concept of graph traversal techniques.
- To implement Breadth First Search (BFS) traversal of a graph using C programming.
- To implement Depth First Search (DFS) traversal of a graph using C programming.

2.3.1 Theory

Graphs are widely used data structures in computer science to represent relationships between objects. A graph consists of vertices (nodes) and edges (connections between nodes).

Graph traversal is the process of visiting all the vertices of a graph in a systematic manner. Two commonly used traversal algorithms are Breadth First Search (BFS) and Depth First Search (DFS).

Breadth First Search (BFS)

BFS explores the graph level by level. It starts at a source node, visits all of its neighboring nodes first, and then moves to the next level of neighbors. A queue is used to keep track of the vertices to be explored. BFS is often used to find the shortest path in unweighted graphs.

Depth First Search (DFS)

DFS explores the graph by going as deep as possible along each branch before backtracking. It starts from a source node and continues visiting nodes until it reaches a node with no unvisited neighbors. DFS can be implemented using recursion or a stack.

BFS and DFS differ in their traversal order:

- BFS visits nodes in increasing levels.
- DFS visits nodes by going deep into each path before moving to the next.

2.3.2 Procedure for BFS Traversal

1. Start with a graph G and a starting node s.
2. Create an empty queue and mark all vertices as unvisited.
3. Insert the starting vertex s into the queue and mark it as visited.
4. While the queue is not empty:
    - Remove a vertex v from the queue.
    - Visit and print v.
    - Insert all unvisited neighbors of v into the queue and mark them as visited.
5. Repeat until all reachable nodes are visited.

2.3.3 Implementation of BFS Traversal

Write a C program to implement Breadth First Search (BFS) traversal of a graph using an adjacency matrix.

```
#include <stdio.h>
#define MAX 10
int graph[MAX][MAX], visited[MAX];
int queue[MAX], front = -1, rear = -1;
int n;
void enqueue(int v) {
if(rear == MAX-1)
return;
if(front == -1)
front = 0;
queue[++rear] = v;
}
int dequeue() {
return queue[front++];
}
```

```
void bfs(int start) {
int i;
enqueue(start);
visited[start] = 1;
while(front <= rear) {
int v = dequeue();
printf("%d ", v);
for(i = 0; i < n; i++) {
if(graph[v][i] == 1 && visited[i] == 0) {
enqueue(i);
visited[i] = 1;
}
}
}
}
int main() {
int i, j, start;
printf("Enter number of vertices: ");
scanf("%d", &n);
printf("Enter adjacency matrix:\n");
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
scanf("%d", &graph[i][j]);
printf("Enter starting vertex: ");
scanf("%d", &start);
printf("BFS Traversal: ");
bfs(start);
return 0;
}
```

Input

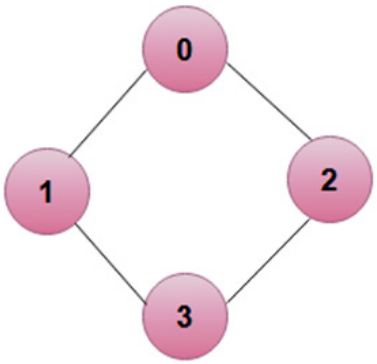


Fig. 2.3.1 Input graph of BFS traversal

OUTPUT

```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter starting vertex: 0
BFS Traversal: 0 1 2 3
```


2.3.4 Procedure for DFS Traversal (Recursive)

1. Start with a graph G and a starting node s.
2. Mark s as visited and print it.
3. For each adjacent vertex v of s:
    - If v is not visited, apply DFS recursively on v.
4. Repeat until all vertices are visited.

2.3.5 Implementation of DFS Traversal

Write a C program to implement Depth First Search (DFS) traversal of a graph using recursion.

```
#include <stdio.h>
#define MAX 10
int graph[MAX][MAX], visited[MAX];
int n;
void dfs(int v) {
int i;
printf("%d ", v);
visited[v] = 1;
for(i = 0; i < n; i++) {
if(graph[v][i] == 1 && visited[i] == 0) {
dfs(i);
}
}
}
int main() {
int i, j, start;
printf("Enter number of vertices: ");
scanf("%d", &n);
printf("Enter adjacency matrix:\n");
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
scanf("%d", &graph[i][j]);
printf("Enter starting vertex: ");
scanf("%d", &start);
printf("DFS Traversal: ");
dfs(start);
return 0;
}
```

Input

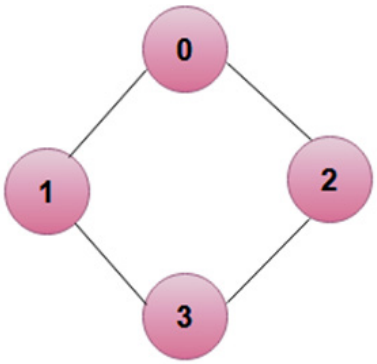


Fig. 2.3.2 Input graph of DFS traversal

OUTPUT

```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter starting vertex: 0
DFS Traversal: 0 1 3 2
```


2.3.6 Lab Practice Questions

1. Write a C program to perform Breadth First Search (BFS) traversal of a graph with 6 vertices using an adjacency matrix.

2. Write a C program to implement Depth First Search (DFS) traversal of a graph using recursion.

3. Modify the BFS program to allow the user to enter the graph dynamically using an adjacency matrix.

4. Write a program to perform DFS traversal using a stack (iterative method) instead of recursion.

5. Construct a graph with 7 vertices and perform both BFS and DFS traversals starting from vertex 0. Compare the traversal orders.

Experiment 4

Implementation of Dijkstra's Algorithm and Kruskal's Algorithm

Objectives

- To implement Dijkstra's Algorithm for finding the shortest path in a graph.
- To implement Kruskal's Algorithm for constructing the Minimum Spanning Tree (MST).

2.4.1 Theory

2.4.1.1 Dijkstra's Algorithm

Dijkstra's algorithm is a greedy technique used to determine the shortest path from a single source vertex to all other vertices in a graph.

- It works for both directed and undirected graphs.
- The graph must have non-negative edge weights.
- The algorithm keeps track of the shortest known distance to each vertex and continuously updates these values until all vertices are processed.

If negative edge weights are present, Dijkstra's Algorithm cannot be used, and algorithms like Bellman-Ford should be applied instead.

Algorithm

1. Assign a distance value of zero to the source vertex and infinity to all other vertices.
2. Mark all vertices as unvisited.
3. Select the unvisited vertex with the smallest distance value and consider it as the current vertex.
4. For each unvisited neighbor of the current vertex, calculate the new distance by adding the edge weight to the current vertex's distance.
5. If this newly calculated distance is smaller than the previously known distance, then update the distance of that neighbor.

6. After checking all neighbors, mark the current vertex as visited (it will not be processed again).

7. Repeat steps 3–6 until all vertices are visited.

8. At the end, the distance array contains the shortest distance from the source to every other vertex.

Applications of Dijkstra's Algorithm

- GPS Navigation systems (finding shortest route)
- Computer networks (routing protocols)
- Game development (pathfinding for characters)

2.4.1.2 Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph. An MST is a subset of the edges that connects all vertices together, without forming any cycles, and with the minimum possible total edge weight.

The algorithm works by sorting all edges of the graph in ascending order of their weights, and then adding them one by one to the spanning tree, ensuring that no cycle is formed. To check for cycles, a Union-Find (Disjoint Set Union) data structure is commonly used.

Algorithm

1. Sort all edges of the graph in non-decreasing order of their weights.

2. Initialize the Minimum Spanning Tree (MST) as an empty set.

3. For each edge in the sorted list, check whether adding it to the MST will create a cycle.

4. If the edge does not form a cycle, include it in the MST; otherwise, discard it.

5. Repeat step 3 until the MST contains exactly (V – 1) edges, where V is the number of vertices.

6. The edges included in the MST together represent the Minimum Spanning Tree of the graph.

Applications of Kruskal's Algorithm

- Designing communication networks (telephone, internet, or LAN cabling).
- Electrical grid design to minimize wiring cost.

- Clustering in data analysis.

2.4.2 Implementation

1. Write a program to implement Dijkstra's Algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph.

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minDistance(int dist[], int visited[]) {
    int min = INT_MAX, min_index = -1;
    for (int v = 0; v < V; v++)
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    return min_index;
}
void dijkstra(int graph[V][V], int src) {
    int dist[V], visited[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        visited[i] = 0;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, visited);
        visited[u] = 1;
        for (int v = 0; v < V; v++)
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX
```

```
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}
int main() {
    int graph[V][V] = {
        {0, 10, 0, 0, 5},
        {10, 0, 1, 0, 2},
        {0, 1, 0, 4, 9},
        {0, 0, 4, 0, 7},
        {5, 2, 9, 7, 0}
    };
    dijkstra(graph, 0);
    return 0;
}
```

OUTPUT

```
Vertex   Distance from Source
0        0
1        7
2        8
3        12
4        5
```

2. Write a program to implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) of a weighted graph.

```
#include <stdio.h>
#include <stdlib.h>
#define V 5
#define E 7
struct Edge {
    int src, dest, weight;
};
struct Subset {
    int parent, rank;
};
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```

```
int compare(const void* a, const void* b) {
    return ((struct Edge*)a)->weight - ((struct Edge*)b)->weight;
}
void Kruskal(struct Edge edges[]) {
    struct Edge result[V];
    int e = 0, i = 0;
    qsort(edges, E, sizeof(edges[0]), compare);
    struct Subset* subsets = (struct Subset*) malloc(V * sizeof(struct Subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    while (e < V - 1 && i < E) {
        struct Edge next_edge = edges[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }
    printf("Edges in the Minimum Spanning Tree:\n");
    int totalWeight = 0;
    for (i = 0; i < e; i++) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
        totalWeight += result[i].weight;
    }
    printf("Total weight of MST: %d\n", totalWeight);
```

```
}
int main() {
    struct Edge edges[E] = {
        {0, 1, 10}, {0, 4, 5}, {1, 2, 1},
        {2, 3, 4}, {3, 4, 7}, {1, 4, 2}, {2, 4, 9}
    };
    Kruskal(edges);
    return 0;
}
```

OUTPUT

```
Edges in the Minimum Spanning Tree:
1 -- 2 == 1
1 -- 4 == 2
2 -- 3 == 4
0 -- 4 == 5
Total weight of MST: 12
```

2.4.3 Lab Practice Questions

Vertices: 0, 1, 2, 3, 4

Weighted Edges:

Edge	Weight
0 – 1	10
0 – 4	5
1 – 2	1
1 – 4	2
2 – 3	4
2 – 4	9
3 – 4	7

1. Using the given graph, write a C program to implement Dijkstra's Algorithm and find the shortest distance from vertex 0 to all other vertices.

2. Trace the step-by-step working of Dijkstra's Algorithm for the given graph and show updated distances at each iteration.

3. Using the given graph, write a C program to implement Kruskal's Algorithm and construct the Minimum Spanning Tree (MST).

4. Show the step-by-step selection of edges when applying Kruskal's Algorithm on this graph.

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 LAB

COURSE CODE: M25CA01PC

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841