GALGOTIAS UNIVERSITY

LAB MANUAL

Data Structures [R1UC308B]

Name of School: SCSE

Department: CSE

Year: 2025-2026

GALGOTIAS UNIVERSITY
(Under the Uttar Pradesh Private Universities Act No. 12 of 2019)

SUBJECT	Data Structures	**PROGRAMME**	B.Tech CSE
SUBJECT CODE	R1UC308B	**SEMESTER**	3
CREDITS	1	**DURATION OF SEMESTER**	13 Weeks
PRE-REQUISITE SUBJECTS	C / C++	**SESSION DURATION**	2Hrs Per Week

Vision

"To be recognized globally as a premier School of Computing Science and Engineering for imparting quality and value-based education within a multi-disciplinary and collaborative research-based environment."

Mission

The mission of the school is to:

M1: Develop a strong foundation in fundamentals of computing science and engineering with responsiveness towards emerging technologies.

M2: Establish state-of-the-art facilities and adopt education 4.0 practices to analyze, develop, test and deploy sustainable ethical IT solutions by involving multiple stakeholders.

M3: Foster multidisciplinary collaborative research in association with academia and industry through focused research groups, Centre of Excellence, and Industry Oriented R&D Labs.

PROGRAM EDUCATIONAL OBJECTIVES

The Graduates of Computer Science and Engineering shall:

PEO1: Graduates will possess the knowledge and skills to analyze, design, and implement efficient data structures and algorithms using Java, preparing them for advanced studies or professional careers in software development.

PEO2: Graduates will be proficient in identifying problems, formulating solutions, and implementing them using data structures in Java, enabling them to address real-world challenges in diverse domains.

PEO3: Graduates will demonstrate the capability to stay current with technological advancements, adapt to new tools and programming languages, and continuously improve their skills to remain competitive in the dynamic field of software engineering.

PROGRAMME SPECIFIC OUTCOME (PSO):

The students of Computer Science and Engineering shall:

PSO1: Have the ability to work with emerging technologies in computing requisite to Industry 4.0.

PSO2: Demonstrate Engineering Practice learned through industry internship and research project to solve live problems in various domains.

Program Outcomes (POs)

PO1: Engineering Knowledge: Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization as specified in WK1 to WK4 respectively to develop to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions with consideration for sustainable development. (WK1 to WK4)

PO3: Design/Development of Solutions: Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required. (WK5)

PO4: Conduct Investigations of Complex Problems: Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions. (WK8).

PO5: Engineering Tool Usage: Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems. (WK2 and WK6)

PO6: The Engineer and The World: Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment. (WK1, WK5, and WK7).

PO7: Ethics: Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws. (WK9)

PO8: Individual and Collaborative Team work: Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.

PO9: Communication: Communicate effectively and inclusively within the engineering community and society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations considering cultural, language, and learning differences

PO10: Project Management and Finance: Apply knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.

PO11: Life-Long Learning: Recognize the need for, and have the preparation and ability for i) independent and life-long learning ii) adaptability to new and emerging technologies and iii) critical thinking in the broadest context of technological change. (WK8)

Knowledge and Attitude Profile (WK)

WK1: A systematic, theory-based understanding of the natural sciences applicable to the discipline and awareness of relevant social sciences.

WK2: Conceptually-based mathematics, numerical analysis, data analysis, statistics and formal aspects of computer and information science to support detailed analysis and modelling applicable to the discipline.

WK3: A systematic, theory-based formulation of engineering fundamentals required in the engineering discipline.

WK4: Engineering specialist knowledge that provides theoretical frameworks and bodies of knowledge for the accepted practice areas in the engineering discipline; much is at the forefront of the discipline.

WK5: Knowledge, including efficient resource use, environmental impacts, whole-life cost,re- use of resources, net zero carbon, and similar concepts, that supports engineering design and operations in a practice area.

WK6: Knowledge of engineering practice (technology) in the practice areas in theengineering discipline.

WK7: Knowledge of the role of engineering insociety and identified issues in engineering practice in the discipline, such as the professional responsibility of an engineer topublic safety and sustainable development.

WK8: Engagement with selected knowledge in the current research literature of the discipline, awareness of the power of critical thinking and creative approaches to evaluate emerging issues.

WK9: Ethics, inclusive behavior and conduct. Knowledge of professional ethics, responsibilities, and norms of engineering practice. Awareness of the need for diversity by reason of ethnicity, gender, age, physical ability etc. with mutual understanding.

COURSE OUTCOMES (COs)

After the completion of the course, the student will be able to:

CO No.	Course Outcomes
CO1	Understand and analyze the fundamentals of data structures, including arrays, linked lists, stacks, queues, trees, and graphs, and apply appropriate representations in problem-solving. *(Blooms Level: Understand & Apply)*
CO2	Implement searching and sorting algorithms efficiently and evaluate their performance using time and space complexity analysis. *(Blooms Level: Apply & Analyze)*
CO3	Develop recursive and iterative solutions for real-world problems, comparing trade-offs in terms of efficiency and readability. *(Blooms Level: Apply & Analyze)*
CO4	Apply various data structures and algorithmic techniques (e.g., trees, graphs, heaps) to solve problems in structured and modular programming environments. *(Blooms Level: Apply & Create)*

BLOOM'S LEVEL OF THE COURSE OUTCOMES

INTEGRATED

CO No.	Remember **BTL1**	Understand **BTL2**	Apply **BTL3**	Analyse **BTL4**	Evaluate **BTL5**	Create **BTL6**
CO1			√			
CO2				√		
CO3				√		
CO4						√

PROGRAM OUTCOMES (POs): AS DEFINED BY CONCERNED THE APEX BODIES

COURSE ARTICULATIONMATRIX

COs#/POs	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**		**PSO1**	**PSO2**
CO1	3	2	1	-	-	-	-	-	-	-	-	-	-	-
CO2	3	3	2	-	2	-	-	-	-	-	-	-	-	-
CO3	3	3	1	2	1	-	-	-	-	-	1	-	-	-
CO4	3	3	3	2	3	-	-	-	-	-	2	-	-	-

Note: 1-Low, 2-Medium, 3-High

COURSE ASSESSMENT

The course assessment patterns are the assessment tools used both in formative and summative examinations.

Assessment Tools	CIE			Total Marks		Final Marks
	LAB @ (Work + Record)	MTE	LAB EXAM*	CIE	SEE	CIE * 0.5 + SEE * 0.5
Integrated	**25**	**50**	**25**	**100**	**100**	**100**

@ Lab Work – 15 marks + Lab record – 10 marks

***** Passing criteria – 30% of marks to be secured in the lab exam conducted by two examiners (one internal + one external).

S.No.	Experiments
1	Write a Program to insert and delete an element in 1-D arrays
2	Write a Program to implement reverse an arrays
3	Write a Program to implement addition and multiplication of two 2D arrays
4	Write a Program to find max and min of arrays
5	Write a program to find factorial of any number using iteration and recursion.
6	Write a program to find fibonacci series till a number using iteration and recursion.
7	Write a program to implement Tower of Hanoi problem.
8	You are given an integer n. You have to print all numbers from 1 to n using recursion only.
9	Write a program to implement linear search in 2D-matrix
10	Write a program to implement Binary search in an array.
11	Given sorted array **arr** with possibly some duplicates, the task is to find the first and last occurrences of an element **x** in the given array.
12	Write a program to implement Bubble Sort
13	Write a program to implement Selection Sort
14	Write a program to implement Insertion Sort
15	Write a program to implement Quick Sort
16	Write a program to implement Merge Sort
17	Write a menu driven program to implement Single Linked List for operations : Insert (start, end and at position), Delete (start, end and at position), Search, and Display
18	Write a program to implement doubly linked list and display data of all nodes.
19	Write a program to implement circular linked list and display data of all nodes.
20	Write a program to implement Stack operations push, pop, isEmpty and isFull using array.
21	Write a program to implement Circular Queue operations push, pop, isEmpty and isFull using array.
22	Write a program to implement in-order, pre-order and post-order traversal of binary tree.
23	Write a program to implement BST. Write a function to insert, delete and search from BST.
24	Write a Program to implement BFS of graph.
25	Write a Program to implement DFS of graph.

Experiment 1

AIM:

Write a Program to insert and delete an element in an array.

OBJECTIVES:

- Understand array-based memory allocation.
- Learn basic operations like insertion and deletion.
- Develop logic for shifting elements in arrays.

ALGORITHM:

Insertion:

1. Read the array and its size.
2. Read the element to insert and its position.
3. Shift elements from the position to the right.
4. Insert the element.
5. Print the updated array.

Deletion:

1. Read the element position to delete.
2. Shift elements from the next position to the left.
3. Reduce array size.
4. Print the updated array.

C LANGUAGE CODE:

```
#include <stdio.h>

void insertElement(int arr[], int *n, int pos, int value) {
    for (int i = *n; i > pos; i--)
        arr[i] = arr[i - 1];
    arr[pos] = value;
    (*n)++;
}

void deleteElement(int arr[], int *n, int pos) {
    for (int i = pos; i < *n - 1; i++)
        arr[i] = arr[i + 1];
    (*n)--;
}

int main() {
    int arr[100], n, choice, pos, value;

    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("1. Insert\n2. Delete\nEnter your choice: ");
    scanf("%d", &choice);

    if (choice == 1) {
        printf("Enter position and value to insert: ");
        scanf("%d%d", &pos, &value);
        insertElement(arr, &n, pos, value);
    } else if (choice == 2) {
        printf("Enter position to delete: ");
        scanf("%d", &pos);
        deleteElement(arr, &n, pos);
    }

    printf("Updated array:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the time complexity of insertion and deletion in arrays?
2. How do you handle insertion if the array is full?
3. Can you insert at the end or beginning efficiently?

Experiment 2

AIM:

Write a Program to implement reverse an array.

OBJECTIVES:

- Learn in-place array manipulation.
- Understand two-pointer technique.
- Practice with for/while loops.

ALGORITHM:

1. Read the array and its size.
2. Initialize two pointers: one at start, one at end.
3. Swap elements at both pointers.
4. Move pointers toward each other until they meet.

C LANGUAGE CODE:

```
#include <stdio.h>
void reverseArray(int arr[], int n) {
    int start = 0, end = n - 1, temp;
    while (start < end) {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

int main() {
    int arr[100], n;

    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    reverseArray(arr, n);

    printf("Reversed array:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the time complexity of reversing an array?
2. Can you reverse an array without using a loop?
3. How do you reverse a portion of an array?

Experiment 3

AIM:

Write a Program to implement addition and multiplication of two matrices.

OBJECTIVES:

- Understand matrix representation using 2D arrays.
- Learn nested loop operations.
- Apply concepts of matrix addition and multiplication.

ALGORITHM:

Addition:

1. Read matrix dimensions and elements.
2. Sum corresponding elements of both matrices.
3. Store result in a third matrix.

Multiplication:

1. Check if multiplication is possible: col1 == row2.
2. Multiply row of Matrix A with column of Matrix B.
3. Store the result in Matrix C.

C LANGUAGE CODE:

```
#include <stdio.h>

void addMatrices(int a[10][10], int b[10][10], int res[10][10], int r, int c)
{
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            res[i][j] = a[i][j] + b[i][j];
}


void multiplyMatrices(int a[10][10], int b[10][10], int res[10][10], int r1,
int c1, int c2) {
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c2; j++) {
            res[i][j] = 0;
            for (int k = 0; k < c1; k++)
                res[i][j] += a[i][k] * b[k][j];
        }
}
int main() {
    int a[10][10], b[10][10], res[10][10], r1, c1, r2, c2, choice;
```

```
    printf("Enter rows and columns of Matrix A: ");
    scanf("%d%d", &r1, &c1);
    printf("Enter elements of Matrix A:\n");
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c1; j++)
            scanf("%d", &a[i][j]);
    printf("Enter rows and columns of Matrix B: ");
    scanf("%d%d", &r2, &c2);
    printf("Enter elements of Matrix B:\n");
    for (int i = 0; i < r2; i++)
        for (int j = 0; j < c2; j++)
            scanf("%d", &b[i][j]);

    printf("1. Addition\n2. Multiplication\nEnter your choice: ");
    scanf("%d", &choice);

    if (choice == 1 && r1 == r2 && c1 == c2) {
        addMatrices(a, b, res, r1, c1);
        printf("Resultant Matrix:\n");
        for (int i = 0; i < r1; i++) {
            for (int j = 0; j < c1; j++)
                printf("%d ", res[i][j]);
            printf("\n");
        }
    } else if (choice == 2 && c1 == r2) {
        multiplyMatrices(a, b, res, r1, c1, c2);
        printf("Resultant Matrix:\n");
        for (int i = 0; i < r1; i++) {
            for (int j = 0; j < c2; j++)
                printf("%d ", res[i][j]);
            printf("\n");
        }
    } else {
        printf("Invalid operation or dimension mismatch.\n");
    }
    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What are the dimension conditions for matrix addition and multiplication?
2. What is the time complexity of matrix multiplication?
3. Can matrix multiplication be done in less than $O(n^3)$?

Experiment 4

AIM:

Write a program to find the maximum and minimum elements in an array.

OBJECTIVES:

- Understand array traversal techniques.
- Learn how to compare elements in a single pass.
- Implement efficient logic to find max and min values.

ALGORITHM:

1. Start with first element as both max and min.
2. Traverse the array from the second element.
3. For each element, compare it with current max and min.
4. Update max or min accordingly.
5. Print the final max and min values.

C LANGUAGE CODE:

```
#include <stdio.h>
int main() {
    int arr[100], n, i, max, min;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    max = min = arr[0];

    for(i = 1; i < n; i++) {
        if(arr[i] > max)
            max = arr[i];
        if(arr[i] < min)
            min = arr[i];
    }
    printf("Maximum: %d\n", max);
    printf("Minimum: %d\n", min);

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the time complexity of this algorithm?
2. Can this be done in a single pass efficiently?
3. What happens if all elements are the same?

Experiment 5

AIM:

Write a program to find the factorial of any number using both iteration and recursion.

OBJECTIVES:

- Understand recursion and its base condition.
- Compare iterative and recursive approaches.
- Learn stack behavior in recursion.

ALGORITHM (Iterative):

1. Set `fact = 1`.
2. Loop from 1 to `n` and multiply `fact = fact * i`.
3. Return `fact`.

ALGORITHM (Recursive):

1. If `n == 0` or `n == 1`, return 1.
2. Else return `n * factorial(n - 1)`.

C LANGUAGE CODE:

```
#include <stdio.h>
// Iterative
int factorial_iterative(int n) {
    int fact = 1;
    for(int i = 1; i <= n; i++)
        fact *= i;
    return fact;
}
// Recursive
int factorial_recursive(int n) {
    if(n <= 1)
        return 1;
    else
        return n * factorial_recursive(n - 1);
}
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    printf("Iterative Factorial: %d\n", factorial_iterative(num));
    printf("Recursive Factorial: %d\n", factorial_recursive(num));

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. Which method is more memory efficient?
2. What is the base case in recursion?
3. Can you trace the recursive stack?

Experiment 6

AIM :

Write a program to find the Fibonacci series till a number using iteration and recursion.

OBJECTIVES:

- Learn the definition of Fibonacci numbers.
- Compare iterative vs. recursive implementation.
- Understand exponential vs. linear time solutions.

ALGORITHM (Iterative):

1. Initialize `a = 0`, `b = 1`.
2. Print `a` and `b`.
3. Loop till `n`, update `c = a + b`; `a = b`; `b = c`.

ALGORITHM (Recursive):

1. If `n == 0` return 0; if `n == 1` return 1.
2. Else return `fibonacci(n-1) + fibonacci(n-2)`.

C LANGUAGE CODE:

```
#include <stdio.h>
// Recursive
int fibonacci_recursive(int n) {
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
}
// Iterative
void fibonacci_iterative(int n) {
    int a = 0, b = 1, c;
    printf("%d %d ", a, b);
    for(int i = 2; i < n; i++) {
        c = a + b;
        printf("%d ", c);
        a = b;
        b = c;
    }
}

int main() {
    int num;
    printf("Enter number of terms: ");
    scanf("%d", &num);

    printf("Fibonacci (Iterative): ");
```

```
    fibonacci_iterative(num);

    printf("\nFibonacci (Recursive): ");
    for(int i = 0; i < num; i++)
        printf("%d ", fibonacci_recursive(i));

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the time complexity of the recursive Fibonacci function?
2. How can you optimize it using Dynamic Programming?
3. Compare memory usage in both approaches.

Experiment 7

AIM:

Write a program to implement the Tower of Hanoi problem using recursion.

OBJECTIVES:

- Understand recursive problem-solving.
- Explore divide and conquer strategy.
- Apply recursion to a classic algorithmic puzzle.

ALGORITHM:

1. If `n == 1`, move disk from source to destination.
2. Move `n-1` disks from source to auxiliary.
3. Move nth disk from source to destination.
4. Move `n-1` disks from auxiliary to destination.

C LANGUAGE CODE:

```
#include <stdio.h>

void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }
    towerOfHanoi(n - 1, source, destination, auxiliary);
    printf("Move disk %d from %c to %c\n", n, source, destination);
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int n;
    printf("Enter number of disks: ");
    scanf("%d", &n);
    towerOfHanoi(n, 'A', 'B', 'C');
    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the time complexity of Tower of Hanoi?
2. How many total moves are required for `n` disks?
3. Can this be solved iteratively?

Experiment 8

AIM:

You are given an integer n. You have to print all numbers from 1 to n using recursion only.

OBJECTIVES:

- Practice basic recursion.
- Understand stack-based execution of function calls.
- Observe order of execution in recursive calls.

ALGORITHM:

1. Define a recursive function `print(n)`.
2. If `n == 0`, return.
3. Call `print(n-1)`, then print `n`.

C LANGUAGE CODE:

```
#include <stdio.h>

void printNumbers(int n) {
    if (n == 0)
        return;
    printNumbers(n - 1);
    printf("%d ", n);
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    printNumbers(num);
    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. Can this be done in reverse order using recursion?
2. What is the base case in this recursion?
3. What will happen if there is no base case?

Experiment 9

AIM:

Write a program to implement linear search in a 2D matrix.

OBJECTIVES:

- Understand how 2D arrays are traversed.
- Apply linear search logic to a matrix.
- Handle user-defined matrix sizes and elements.

ALGORITHM:

1. Loop through each row and column.
2. Check if current element equals the target.
3. If found, print position and exit.

C LANGUAGE CODE:

```
#include <stdio.h>

int main() {
    int mat[10][10], rows, cols, i, j, key, found = 0;
    printf("Enter rows and columns: ");
    scanf("%d%d", &rows, &cols);

    printf("Enter matrix elements:\n");
    for(i = 0; i < rows; i++)
        for(j = 0; j < cols; j++)
            scanf("%d", &mat[i][j]);

    printf("Enter element to search: ");
    scanf("%d", &key);

    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            if(mat[i][j] == key) {
                printf("Element found at [%d][%d]\n", i, j);
                found = 1;
                break;
            }
        }
        if(found) break;
    }

    if(!found)
        printf("Element not found.\n");

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the time complexity of linear search in a matrix?
2. Can we use binary search here? When?
3. How do we improve search in a sorted matrix?

Experiment 10

AIM:

Write a program to implement binary search in an array.

OBJECTIVES:

- Learn divide-and-conquer strategy.
- Understand the pre-condition of a sorted array.
- Implement efficient search techniques.

ALGORITHM:

1. Set `low = 0`, `high = n - 1`.
2. While `low <= high`, calculate `mid`.
3. If `arr[mid] == key`, return index.
4. If `arr[mid] < key`, search right half.
5. Else, search left half.

C LANGUAGE CODE:

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

int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1, mid;
    while(low <= high) {
        mid = (low + high) / 2;
        if(arr[mid] == key)
            return mid;
        else if(arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int arr[100], n, key, i, index;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter sorted array:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Enter key to search: ");
    scanf("%d", &key);
```

```
    index = binarySearch(arr, n, key);
    if(index != -1)
        printf("Element found at index %d\n", index);
    else
        printf("Element not found\n");

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What happens if the array is not sorted?
2. What is the time complexity of binary search?
3. Can we implement binary search using recursion?

Experiment 11

AIM:

Given a sorted array `arr` with possibly some duplicates, write a program to find the first and last occurrences of an element `x` in the given array.

OBJECTIVES:

- Understand binary search variants.
- Learn how to deal with duplicate elements in sorted arrays.
- Practice optimized searching in logarithmic time.

ALGORITHM:

1. Use binary search to find the first occurrence:
    - If `arr[mid] == x`, check if it is the first occurrence.
    - Else search left half.
2. Use binary search again to find the last occurrence similarly.

C LANGUAGE CODE:

//Linear Search

```
void findFirstAndLast(int arr[], int n, int x) {

    int first = -1, last = -1;

    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            if (first == -1)
                first = i;
            last = i;
        }
    }

    if (first == -1)
        printf("Element %d not found in the array.\n", x);
    else
        printf("First Occurrence of %d = %d\n
               Last Occurrence of %d = %d\n", x, first, x, last);
}

#include <stdio.h>
//BINARY SEARCH
int firstOccurrence(int arr[], int n, int x) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == x) {
```

```
            result = mid;
            high = mid - 1;
        } else if (arr[mid] < x) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}

int lastOccurrence(int arr[], int n, int x) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == x) {
            result = mid;
            low = mid + 1;
        } else if (arr[mid] < x) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}

int main() {
    int arr[100], n, x, i;
    printf("Enter size of array: ");
    scanf("%d", &n);
    printf("Enter sorted array:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter element to find: ");
    scanf("%d", &x);

    int first = firstOccurrence(arr, n, x);
    int last = lastOccurrence(arr, n, x);

    if (first != -1)
        printf("First Occurrence = %d, Last Occurrence = %d\n", first, last);
    else
        printf("Element not found.\n");

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. Why do we need two binary searches for this problem?
2. What is the time complexity of this solution?
3. How would this change for an unsorted array?

Experiment 12

AIM:

Write a program to implement Bubble Sort.

OBJECTIVES:

- Learn basic sorting techniques.
- Understand swapping and comparisons.
- Analyze time complexity of simple sorting algorithms.

ALGORITHM:

1. Repeat for `n-1` passes:
2. For each pass, compare adjacent elements.
3. Swap if the left element is greater than the right.

C LANGUAGE CODE:

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int arr[100], n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter array elements:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    bubbleSort(arr, n);

    printf("Sorted array:\n");
    for(i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. What is the best and worst case time complexity of Bubble Sort?
2. Can Bubble Sort be optimized? How?
3. How many swaps and comparisons happen in Bubble Sort?

Experiment 13

AIM:

Write a program to implement Selection Sort.

OBJECTIVES:

- Practice comparison-based sorting.
- Understand the concept of selecting the minimum element.
- Compare Selection Sort with Bubble Sort.

ALGORITHM:

1. For each index `i` from `0 to n-1`:
2. Find the index of the minimum element in the unsorted part.
3. Swap it with element at index `i`.

C LANGUAGE CODE:

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
int main() {
    int arr[100], n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter array elements:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    selectionSort(arr, n);

    printf("Sorted array:\n");
    for(i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

POST-EXPERIMENT QUESTIONS:

1. How does Selection Sort work?
2. Is it stable? Can it be made stable?
3. Compare its performance with Insertion Sort.

Experiment 14

Aim:

To implement the Insertion Sort algorithm in C to sort a given array in ascending order.

Objectives:

- Understand the concept of comparison-based sorting.
- Implement the insertion sort algorithm.
- Analyze the time and space complexity of insertion sort.
- Practice array manipulation in C language.

Algorithm: Insertion Sort

1. Start from the second element (index 1) of the array.
2. Store the current element in a variable called `key`.
3. Compare `key` with elements before it.
4. Shift all elements greater than `key` one position to the right.
5. Insert the `key` at its correct position.
6. Repeat steps 2–5 for all remaining elements.
7. Display the sorted array.

C Program:

```
#include <stdio.h>

// Function to perform insertion sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}

// 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[] = {9, 5, 1, 4, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    insertionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Post-Experiment Questions:

1. What is the best-case and worst-case time complexity of insertion sort?
2. Is insertion sort a stable sorting algorithm? Justify.
3. How does insertion sort compare with selection and bubble sort?
4. Can insertion sort be used for linked lists? Why or why not?
5. Modify the program to sort the array in descending order.

Experiment: 15

AIM:

Write a program to implement Quick Sort.

Objectives:

- Understand the divide-and-conquer technique.
- Learn how to partition an array around a pivot.
- Analyze the time and space complexity of Quick Sort.

Algorithm:

1. Choose a pivot element from the array.
2. Partition the array so that elements less than the pivot are on the left and greater on the right.
3. Recursively apply the same logic to the left and right subarrays.
4. Stop when the array is sorted.

C Program:

```
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    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);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted Array: ");
    printArray(arr, n);
    return 0;
}
```

Post-Experiment Questions:

1. What is the best, average, and worst-case time complexity of Quick Sort?
2. How does pivot selection affect performance?
3. Can Quick Sort be implemented iteratively?
4. What is the space complexity of Quick Sort?

Experiment 16

AIM:

Write a program to implement Merge Sort.

Objectives:

- Understand the concept of recursion in sorting.
- Learn how to divide an array and merge sorted subarrays.
- Analyze the stable nature of Merge Sort.

Algorithm:

1. Divide the array into two halves.
2. Recursively sort both halves.
3. Merge the two sorted halves into a single sorted array.
4. Repeat until the entire array is sorted.

C Program:

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
```

```
        merge(arr, l, m, r);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, n);
    mergeSort(arr, 0, n - 1);
    printf("Sorted Array: ");
    printArray(arr, n);
    return 0;
}
```

Post-Experiment Questions:

1. What is the time complexity of Merge Sort in all cases?
2. Compare Merge Sort with Quick Sort.
3. Is Merge Sort stable? Why?
4. Can Merge Sort be implemented without recursion?

Experiment 17

AIM:

Write a menu driven program to implement Single Linked List for operations:
Insert (start, end and at position), Delete (start, end and at position), Search, and Display.

Objectives:

- Understand the structure and use of singly linked lists.
- Perform insertion, deletion, and traversal operations.
- Implement dynamic memory management.

Algorithm:

1. Create a node structure with data and next pointer.
2. For insertion:
    - At start: Update head to point to new node.
    - At end: Traverse till last node and link new node.
    - At position: Traverse to position and adjust pointers.
3. For deletion:
    - At start: Update head to point to next node and free memory.
    - At end: Traverse and unlink last node.
    - At position: Traverse and adjust links.
4. For search: Traverse each node and compare values.
5. For display: Traverse and print each node's data.

C Language Code:

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

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

struct Node* head = NULL;

// Function declarations
void insertAtEnd(int);
void insertAtStart(int);
void insertAtPosition(int, int);
void deleteAtStart();
void deleteAtEnd();
void deleteAtPosition(int);
void search(int);
void display();

int main() {
```

```
    int choice, data, pos;
    while (1) {
        printf("\nMenu:\n1.Insert Start\n2.Insert End\n3.Insert At
Position\n4.Delete Start\n5.Delete End\n6.Delete At
Position\n7.Search\n8.Display\n9.Exit\nChoice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter data: "); scanf("%d", &data);
insertAtStart(data); break;
            case 2: printf("Enter data: "); scanf("%d", &data);
insertAtEnd(data); break;
            case 3: printf("Enter data and position: "); scanf("%d %d",
&data, &pos); insertAtPosition(data, pos); break;
            case 4: deleteAtStart(); break;
            case 5: deleteAtEnd(); break;
            case 6: printf("Enter position: "); scanf("%d", &pos);
deleteAtPosition(pos); break;
            case 7: printf("Enter data to search: "); scanf("%d", &data);
search(data); break;
            case 8: display(); break;
            case 9: exit(0);
        }
    }
    return 0;
}
```

Post-Experiment Questions:

1. What is the difference between singly and doubly linked list?
2. What is the time complexity of insertion at the beginning in a singly linked list?
3. What happens if you delete a node without updating the previous node's pointer?

Experiment 18

AIM:

Write a program to implement doubly linked list and display data of all nodes.

Objectives:

- Understand the concept of doubly linked lists.
- Perform insertion and deletion operations.
- Traverse in both forward and backward directions.

Algorithm:

1. Create a doubly linked node with `prev`, `data`, and `next`.
2. For insertion:
    - At start: Create node and update head and links.
    - At end: Traverse and link new node.
3. For deletion: Update both `prev` and `next` pointers.
4. For display: Traverse and print nodes using `next`.

C Language Code:

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

struct Node {
    int data;
    struct Node *prev, *next;
};

struct Node* head = NULL;

void insertEnd(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data; newNode->next = NULL;
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        return;
    }
    struct Node* temp = head;
    while (temp->next) temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void display() {
    struct Node* temp = head;
    while (temp) {
        printf("%d <-> ", temp->data);
```

```
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Post-Experiment Questions:

1. What are the advantages of doubly linked list over singly linked list?
2. Why is memory usage higher in doubly linked list?
3. Can you reverse a doubly linked list? How?

Experiment 19

AIM:

Write a program to implement circular linked list and display data of all nodes.

Objectives:

- Understand circular linkage in linked lists.
- Handle corner cases like one-node list.
- Implement circular traversal logic.

Algorithm:

1. Create a node with `data` and `next`.
2. Make the `next` of last node point to head.
3. For display, loop until `temp != head` again.

C Language Code:

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

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

struct Node* head = NULL;

void insert(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        head->next = head;
        return;
    }
    struct Node* temp = head;
    while (temp->next != head) temp = temp->next;
    temp->next = newNode;
    newNode->next = head;
}

void display() {
    struct Node* temp = head;
    if (head == NULL) return;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
```

```
    printf("(head)\n");
}
```

Post-Experiment Questions:

1. What is the difference between circular and linear linked list?
2. How do you check if a linked list is circular?
3. Can circular lists be used in real-world applications?

Experiment 20

AIM

Write a program to implement Stack operations push, pop, isEmpty and isFull using array.

Objectives

- Understand the basic operations of stack.
- Implement stack using arrays.
- Analyze time and space complexity of stack operations.

Algorithm

1. Initialize top as -1
2. **PUSH**: If top == SIZE-1 → Stack is Full. Else, increment top and insert element.
3. **POP**: If top == -1 → Stack is Empty. Else, return element at top and decrement top.
4. **isEmpty**: return (top == -1)
5. **isFull**: return (top == SIZE-1)

C Program

```
#include <stdio.h>
#define SIZE 100
int stack[SIZE], top = -1;

void push(int val) {
    if (top == SIZE - 1)
        printf("Stack Overflow\n");
    else
        stack[++top] = val;
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else
        return stack[top--];
}

int isEmpty() { return top == -1; }
int isFull() { return top == SIZE - 1; }
```

Post-Experiment Questions

1. What is the time complexity of push and pop operations?
2. What are the limitations of stack using array?
3. How is stack used in expression evaluation?

Experiment 21

AIM

Write a program to implement Circular Queue operations: push, pop, isEmpty, and isFull using array.

Objectives

- Understand the concept of circular queues.
- Learn to manage front and rear pointers efficiently.
- Implement enqueue and dequeue operations with wrap-around behavior.

Algorithm

1. **Initialization**: front = -1, rear = -1
2. **Enqueue (push)**:
    - If (front == 0 && rear == SIZE-1) or (rear == (front-1) % SIZE): Queue is Full
    - Else if (front == -1): front = rear = 0
    - Else if (rear == SIZE-1 && front != 0): rear = 0
    - Else: rear++
    - Insert element at queue[rear]
3. **Dequeue (pop)**:
    - If front == -1: Queue is Empty
    - Save queue[front]
    - If front == rear: front = rear = -1
    - Else if front == SIZE-1: front = 0
    - Else: front++
4. **isEmpty**: return (front == -1)
5. **isFull**: return ((front == 0 && rear == SIZE - 1) || (rear == (front - 1) % SIZE))

C Program

```
#include <stdio.h>
#define SIZE 5
int queue[SIZE], front = -1, rear = -1;

void enqueue(int val) {
    if ((front == 0 && rear == SIZE - 1) || (rear == (front - 1 + SIZE) %
SIZE)) {
        printf("Queue is Full\n");
        return;
    }
    if (front == -1)
        front = rear = 0;
    else if (rear == SIZE - 1 && front != 0)
        rear = 0;
    else
        rear++;
```

```
    queue[rear] = val;
}

int dequeue() {
    if (front == -1) {
        printf("Queue is Empty\n");
        return -1;
    }
    int data = queue[front];
    if (front == rear)
        front = rear = -1;
    else if (front == SIZE - 1)
        front = 0;
    else
        front++;
    return data;
}

int isEmpty() { return (front == -1); }

int isFull() {
    return ((front == 0 && rear == SIZE - 1) || (rear == (front - 1 + SIZE) %
SIZE));
}
```

Post-Experiment Questions

1. How does a circular queue resolve the limitations of a linear queue?
2. What is the time complexity of enqueue and dequeue?
3. What conditions identify a full circular queue?

Experiment 22

Aim:

Write a program to implement in-order, pre-order, and post-order traversal of a binary tree.

Objectives:

- Understand recursive tree traversal techniques.
- Learn in-order, pre-order, and post-order strategies.
- Apply recursion in binary tree operations.

Algorithm:

Let each node have structure:

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

1. **Pre-order (Root, Left, Right):**
    - Visit root.
    - Traverse left subtree.
    - Traverse right subtree.
2. **In-order (Left, Root, Right):**
    - Traverse left subtree.
    - Visit root.
    - Traverse right subtree.
3. **Post-order (Left, Right, Root):**
    - Traverse left subtree.
    - Traverse right subtree.
    - Visit root.

C Code:

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

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;
}

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

// Inorder traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(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 = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Preorder: ");
    preorder(root);
    printf("\nInorder: ");
    inorder(root);
    printf("\nPostorder: ");
    postorder(root);

    return 0;
}
```

Post-Experiment Questions:

1. What is the time complexity of each traversal method?
2. What traversal would you use to get a sorted sequence from BST?
3. Can you write iterative versions of the traversals?

Experiment 23

Aim:

Write a program to implement BST and functions to insert, delete, and search elements.

Objectives:

- Learn BST insertion, deletion, and search.
- Understand recursive implementation.
- Practice dynamic memory allocation.

Algorithm:

Insert:

- If root is NULL, create a new node.
- If key < root → insert in left subtree.
- If key > root → insert in right subtree.

Search:

- If root is NULL → not found.
- If root->data == key → found.
- Recurse in left/right depending on key.

Delete:

- Case 1: Node with 0 children (leaf) → delete.
- Case 2: Node with 1 child → replace with child.
- Case 3: Node with 2 children → find inorder successor, replace data, delete successor.

C Code:

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

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 in 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;
}

// Search in BST
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Find min value node
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Delete node
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 1 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 2 children
        struct Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
```

```
    }
    return root;
}

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

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal: ");
    inorder(root);

    printf("\nDeleting 20\n");
    root = deleteNode(root, 20);
    inorder(root);

    printf("\nSearching 40: ");
    struct Node* found = search(root, 40);
    if (found)
        printf("Found\n");
    else
        printf("Not Found\n");

    return 0;
}
```

Post-Experiment Questions:

1. What is the time complexity for insert, delete, and search in BST?
2. How would the BST behave if all inserted elements are sorted?
3. What is the difference between BST and binary tree?

Experiment 24

Aim:

Write a program to implement **BFS** traversal of a graph.

Objectives:

- Understand how BFS explores vertices layer-wise.
- Learn to use a queue for graph traversal.
- Apply visited array to avoid repetition.

Algorithm:

1. Initialize a queue and visited array.
2. Start from the given node, mark it visited and enqueue it.
3. While queue is not empty:
    - Dequeue front node.
    - For each unvisited adjacent node:
        - Mark visited and enqueue it.

C Code:

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

#define MAX 100

int adj[MAX][MAX], visited[MAX];
int queue[MAX], front = -1, rear = -1, n;

void enqueue(int v) {
    if (rear == MAX - 1) return;
    if (front == -1) front = 0;
    queue[++rear] = v;
}

int dequeue() {
    if (front == -1 || front > rear) return -1;
    return queue[front++];
}

void bfs(int start) {
    enqueue(start);
    visited[start] = 1;

    while (front <= rear) {
        int v = dequeue();
        printf("%d ", v);
        for (int i = 0; i < n; i++) {
```

```
            if (adj[v][i] && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}

int main() {
    int edges, u, v, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter number of edges: ");
    scanf("%d", &edges);

    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &u, &v);
        adj[u][v] = adj[v][u] = 1;
    }

    printf("Enter start vertex: ");
    scanf("%d", &start);

    printf("BFS traversal: ");
    bfs(start);

    return 0;
}
```

Post-Experiment Questions:

1. What is the time and space complexity of BFS?
2. Can BFS be used to detect cycles in an undirected graph?
3. How can you modify BFS for shortest path in unweighted graphs?

Experiment 25

Aim:

Write a program to implement **DFS** traversal of a graph.

Objectives:

- Learn DFS traversal using recursion.
- Understand stack-based exploration in graph traversal.
- Recognize DFS's role in cycle detection and connectivity.

Algorithm:

1. Start from the given node.
2. Mark node as visited.
3. Recursively visit all unvisited adjacent nodes.

C Code:

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

#define MAX 100

int adj[MAX][MAX], visited[MAX], n;

void dfs(int v) {
    visited[v] = 1;
    printf("%d ", v);
    for (int i = 0; i < n; i++) {
        if (adj[v][i] && !visited[i]) {
            dfs(i);
        }
    }
}

int main() {
    int edges, u, v, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter number of edges: ");
    scanf("%d", &edges);

    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &u, &v);
        adj[u][v] = adj[v][u] = 1;
    }
```

```
    printf("Enter start vertex: ");
    scanf("%d", &start);

    printf("DFS traversal: ");
    dfs(start);

    return 0;
}
```

Post-Experiment Questions:

1. What is the difference between DFS and BFS?
2. When would DFS be more suitable than BFS?
3. How can DFS be used to detect cycles in a graph?
