KALINGA UNIVERSITY

DEPARTMENT OF CS AND FACULTY OF IT

DEPARTMENT OF CS

Lab Manual



Bachelor of Technology in Computer Science and AIML

Semester-3

Data Structures using C Lab

[BTCSAIML307P]

INDEX

Experiment No.	Objective
01	Write a C program to illustrate the concept of sorting algorithms-non-recursive.
02	Write a C program to illustrate the concept of sorting algorithms-Recursive.
03	Write a C program to illustrate the concept of searching algorithms.
04	Write a C program to implement a stack using an array.
05	Write a C program to implement a queue using an array.
06	Write a C program to implement a circular queue using an array.
07	Write a C program to implement a stack using a linked list.
08	Write a C program to implement a queue using a linked list.
09	Write a C program to implement a circular queue using a linked list.
10	Write a C program to illustrate the implementation of tree structures, including binary trees, binary search trees (BSTs), and tree traversal, with insertion and deletion in BSTs.
11	Write a C program to illustrate graph implementation, BFS, DFS, the minimum-cost spanning tree, and the shortest-path algorithm.

EXPERIMENT NO. 1

OBJECTIVE:

To understand and implement non-recursive sorting algorithms in C and arrange the elements of an array in ascending order using an iterative approach.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++ / Visual Studio Code
- C Programming Environment
- Windows/Linux Operating System

THEORY:

Sorting is the process of arranging data in a specific order, typically in ascending or descending order. It is one of the most fundamental operations in computer science and is widely used in searching, data analysis, database management, and algorithm optimization.

A non-recursive sorting algorithm performs sorting using iterative constructs such as loops instead of recursive function calls. These algorithms are generally easier to understand, consume less stack memory, and are suitable for sorting small to medium-sized datasets.

One of the simplest non-recursive sorting algorithms is Bubble Sort.

Bubble Sort repeatedly compares adjacent elements in an array and swaps them if they are in the wrong order. After each pass, the largest unsorted element moves to its correct position at the end of the array, similar to a bubble rising to the surface.

Algorithm:

1. Read the number of elements.
2. Store the elements in an array.
3. Compare each pair of adjacent elements.
4. Swap them if the first element is greater than the second.
5. Repeat the process for all elements until the array is sorted.
6. Display the sorted array.

Advantages

- Simple to understand and implement.
- Does not require additional memory.

- Suitable for small datasets.

Disadvantages

- Inefficient for large datasets.
- Performs a large number of comparisons and swaps.

Time Complexity

Case	Time Complexity
Best Case	O(n) (Optimized Bubble Sort)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity

O(1) (In-place sorting)

C Program

```
#include <stdio.h>

int main()
{
   int a[100], n, i, j, temp;

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

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

   for(i = 0; i < n - 1; i++)
   {
      for(j = 0; j < n - i - 1; j++)
      {
         if(a[j] > a[j + 1])
         {
            temp = a[j];
```

```
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
    }
}

printf("Sorted Array:\n");

for(i = 0; i < n; i++)
    printf("%d ", a[i]);

return 0;
}
```

DIAGRAM:

Bubble Sort Working

Initial Array

45 12 78 25 10

Pass 1

45 12 → Swap

12 45 78 25 10

78 25 → Swap

12 45 25 78 10

78 10 → Swap

12 45 25 10 78

Pass 2

45 25 → Swap

12 25 45 10 78

45 10 → Swap

12 25 10 45 78

Pass 3

25 10 → Swap

12 10 25 45 78

Pass 4

12 10 → Swap

10 12 25 45 78

Final Sorted Array:

10 12 25 45 78

PROCEDURE:

1. Start the program.
2. Declare variables and an array.
3. Read the number of elements.
4. Input the array elements from the user.
5. Use nested for loops to compare adjacent elements.
6. Swap the elements whenever they are not in ascending order.
7. Repeat the process until all passes are completed.
8. Display the sorted array.
9. Stop the program.

OBSERVATION TABLE:

S.No.	Input Array	Sorted Output
1	45, 12, 78, 25, 10	10, 12, 25, 45, 78

S.No.	Input Array	Sorted Output
2	9, 5, 3, 7, 1	1, 3, 5, 7, 9
3	20, 15, 40, 5	5, 15, 20, 40

CALCULATIONS:

For an array containing n elements:

- Number of passes = n − 1
- Number of comparisons (Worst Case)

[

\frac{n(n-1)}{2}

]

Example:

If n = 5

- Passes = 4
- Comparisons = (5 × 4) / 2 = 10
- Worst-case swaps = 10

Time Complexity:

- Best Case: O(n) (Optimized Bubble Sort)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Space Complexity:

- O(1)

RESULT:

The program was successfully executed to sort the given array using the non-recursive Bubble Sort algorithm. The elements were arranged in ascending order, demonstrating the working of an iterative sorting technique.

PRECAUTIONS:

1. Enter the correct number of array elements.
2. Ensure that the array size does not exceed its declared limit.
3. Use valid integer inputs.
4. Verify the loop limits to avoid array index out-of-bounds errors.

5. Compile the program without syntax errors before execution.
6. Check the output for correctness after sorting.
7. Save the source code before closing the programming environment.

EXPERIMENT NO. 2

OBJECTIVE:

To understand and implement a recursive sorting algorithm in C by sorting an array of integers using Recursive Merge Sort, and to study the concept of recursion in sorting.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++ / Visual Studio Code
- C Programming Environment
- Windows/Linux Operating System

THEORY:

Sorting is the process of arranging data in a specified order, such as ascending or descending. Recursive sorting algorithms solve the sorting problem by dividing it into smaller subproblems, solving each subproblem recursively, and then combining the results.

A recursive sorting algorithm repeatedly calls itself until a base condition is reached. These algorithms are based on the Divide and Conquer technique, making them highly efficient for large datasets.

One of the most commonly used recursive sorting algorithms is Merge Sort.

Merge Sort

Merge Sort divides the array into two equal halves until each sub-array contains only one element. It then merges the sub-arrays in sorted order to produce the final sorted array.

Algorithm:

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

Advantages

- Efficient for large datasets.
- Stable sorting algorithm.
- Guaranteed time complexity of O(n log n).

- Suitable for linked lists and external sorting.

Disadvantages

- Requires additional memory for merging.
- Slightly more complex to implement than iterative sorting algorithms.

Time Complexity

Case	Time Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)

Space Complexity

O(n)

C Program

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for(i = 0; i < n1; i++)
        L[i] = arr[left + i];

    for(j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;
```

```
    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 left, int right)
{
    if(left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

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

    printf("Enter the number of elements: ");
```

```
    scanf("%d", &n);

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

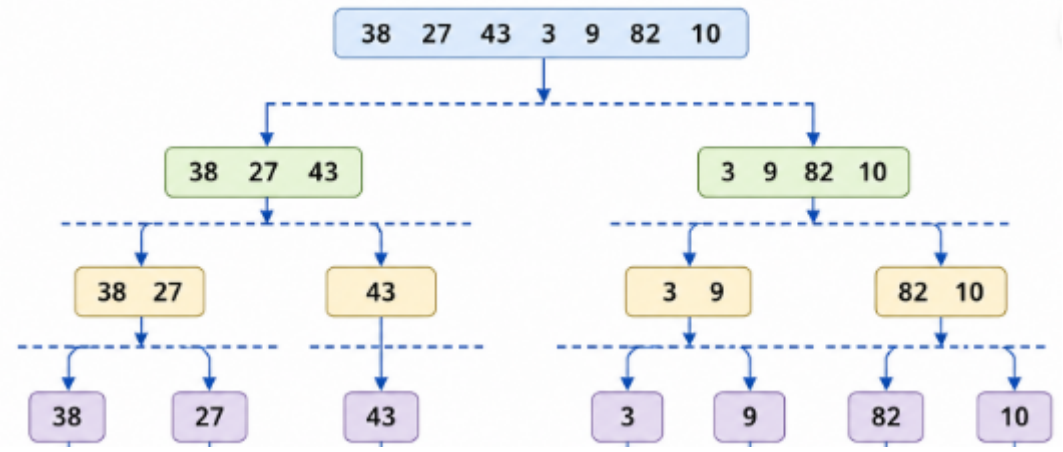
    mergeSort(arr, 0, n - 1);

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

    return 0;
}
```

DIAGRAM:

Merge Sort Working



Merge Back

38 27 → 27 38

3 9 → 3 9

82 10 → 10 82

27 38 + 43 → 27 38 43

3 9 + 10 82 → 3 9 10 82

Final Merge

27 38 43 + 3 9 10 82

↓

3 9 10 27 38 43 82

PROCEDURE:

1. Start the program.
2. Declare an array and required variables.
3. Read the number of elements.
4. Input the array elements.
5. Call the mergeSort() recursive function.
6. Divide the array recursively until each sub-array contains one element.
7. Merge the sorted sub-arrays using the merge() function.
8. Display the sorted array.
9. Stop the program.

OBSERVATION TABLE:

S.No.	Input Array	Sorted Output
1	38, 27, 43, 3, 9, 82, 10	3, 9, 10, 27, 38, 43, 82
2	50, 20, 60, 10, 40	10, 20, 40, 50, 60
3	15, 12, 18, 5	5, 12, 15, 18

CALCULATIONS:

For an array containing n elements:

- Number of recursive levels:

[
\log_2 n
]

- Number of merge operations:

Approximately n $\log_2$ n

Example:

For n = 8

- Recursive Levels = $\log_2 8 = 3$
- Time Complexity = O(n log n)

Complexity Analysis:

Case	Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)
Space Complexity	O(n)

RESULT:

The program was successfully executed to sort the given array using the Recursive Merge Sort algorithm. The output confirmed that the array elements were arranged in ascending order, demonstrating the effectiveness of recursion and the divide-and-conquer approach in sorting.

PRECAUTIONS:

1. Enter the correct number of array elements.
2. Ensure that the array size does not exceed the declared limit.
3. Verify that the recursive base condition (left < right) is implemented correctly to prevent infinite recursion.
4. Ensure the merge function correctly combines the sorted sub-arrays.
5. Compile the program without syntax errors before execution.
6. Provide valid integer inputs.
7. Verify the sorted output after execution.

EXPERIMENT NO. 3

OBJECTIVE:

To understand and implement searching algorithms in C by locating a given element in an array using Linear Search and Binary Search, and to compare their working and efficiency.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++ / Visual Studio Code
- C Programming Environment
- Windows/Linux Operating System

THEORY:

Searching is the process of finding the location of a specific element (called the key) in a collection of data. It is one of the most fundamental operations in computer science and is extensively used in databases, file systems, information retrieval, and data processing.

The two most common searching algorithms are:

1. Linear Search

Linear Search is the simplest searching technique in which each element of the array is checked sequentially until the desired element is found or the end of the array is reached.

Algorithm:

1. Start from the first element.
2. Compare each element with the search key.
3. If a match is found, return its position.
4. If the end of the array is reached without finding the key, report that the element is not found.

Advantages

- Simple to implement.
- Works on both sorted and unsorted arrays.

Disadvantages

- Inefficient for large datasets.
- Requires more comparisons.

Time Complexity

- Best Case: O(1)
- Average Case: O(n)
- Worst Case: O(n)

2. Binary Search

Binary Search is an efficient searching algorithm that works only on sorted arrays. It repeatedly divides the search interval into two halves until the required element is found.

Algorithm:

1. Set the lower and upper indices.
2. Find the middle element.
3. Compare the middle element with the search key.
4. If they are equal, return the position.
5. If the key is smaller, search the left half.
6. Otherwise, search the right half.
7. Repeat until the element is found or the search interval becomes empty.

Advantages

- Much faster than Linear Search for large datasets.
- Requires fewer comparisons.

Disadvantages

- Applicable only to sorted arrays.
- Slightly more complex to implement.

Time Complexity

Case	Linear Search	Binary Search
Best Case	O(1)	O(1)
Average Case	O(n)	O(log n)
Worst Case	O(n)	O(log n)

C Program

```
#include <stdio.h>

int linearSearch(int arr[], int n, int key)
{
   int i;
```

```
    for(i = 0; i < n; i++)
    {
        if(arr[i] == key)
            return i;
    }
    return -1;
}

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(key < arr[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }

    return -1;
}

int main()
{
    int arr[100], n, i, key, choice, result;

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

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

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

    printf("\n1. Linear Search\n2. Binary Search\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    if(choice == 1)
        result = linearSearch(arr, n, key);
    else
        result = binarySearch(arr, n, key);

    if(result == -1)
        printf("Element not found.\n");
    else
        printf("Element found at position %d", result + 1);

    return 0;
}
```

DIAGRAM:

Linear Search

Array

10 20 30 40 50

Search Key = 40

10 → No

20 → No

30 → No

40 → Found ✓

Binary Search
Sorted Array

10 20 30 40 50 60 70

Search Key = 50

Step 1

Low High
| |
10 20 30 40 50 60 70
↑
Mid=40

50 > 40

Search Right Half

40 50 60 70

↑
Mid=60

50 < 60

Search Left Half

50

Found

PROCEDURE:

1. Start the program.
2. Declare the array and required variables.
3. Read the number of elements.
4. Input the array elements.
5. Enter the element to be searched.
6. Select either Linear Search or Binary Search.
7. Execute the selected searching algorithm.
8. Display whether the element is found along with its position.
9. Stop the program.

OBSERVATION TABLE:

S.No.	Array Elements	Search Key	Algorithm	Result
1	10, 20, 30, 40, 50	40	Linear Search	Found at Position 4
2	10, 20, 30, 40, 50	30	Binary Search	Found at Position 3
3	5, 8, 12, 16, 25	18	Binary Search	Not Found

CALCULATIONS:

For an array containing n elements:

Linear Search

- Best Case Comparisons = 1
- Worst Case Comparisons = n
- Average Comparisons = n/2

Binary Search

- Maximum Comparisons = $\lceil \log_2 n \rceil$

Example:

For n = 16

- Linear Search (Worst Case) = 16 Comparisons
- Binary Search (Worst Case) = $\log_2 16 = 4$ Comparisons

Complexity Analysis

Algorithm	Best	Average	Worst	Space
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)

RESULT:

The program was successfully executed to search a given element using Linear Search and Binary Search algorithms. The position of the searched element was displayed correctly, demonstrating the working of both searching techniques and highlighting the efficiency of Binary Search for sorted arrays.

PRECAUTIONS:

1. Enter valid integer values for the array elements.
2. Ensure that the array size does not exceed the declared limit.
3. The array must be sorted before performing Binary Search.
4. Verify the search key before execution.
5. Compile the program without syntax errors.
6. Check the output to ensure the correct position of the searched element is displayed.
7. Save the source code before closing the programming environment.

EXPERIMENT NO. 4

OBJECTIVE:

To understand the concept of the Stack data structure and implement its basic operations (Push, Pop, Peek, and Display) using an array in C.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A Stack is a linear data structure that follows the LIFO (Last In, First Out) principle. The element inserted last is removed first.

The insertion operation in a stack is called Push, whereas the deletion operation is called Pop. Another important operation is Peek (Top), which returns the topmost element without removing it.

A stack can be implemented using either arrays or linked lists. In this experiment, the stack is implemented using an array with a variable called top to indicate the current topmost element.

Basic Operations

- Push: Inserts an element onto the stack.
- Pop: Removes the top element from the stack.
- Peek: Displays the top element.
- Display: Shows all the elements of the stack.

Applications of Stack

- Function call management
- Expression evaluation
- Parentheses balancing
- Undo/Redo operations
- Backtracking algorithms
- Browser history management

Time Complexity

Operation	Time Complexity
Push	O(1)
Pop	O(1)
Peek	O(1)
Display	O(n)

C PROGRAM

```
#include<stdio.h>

#define MAX 5

int stack[MAX];
int top = -1;

void push()
{
   int item;

   if(top == MAX-1)
   {
      printf("Stack Overflow\n");
      return;
   }

   printf("Enter element: ");
   scanf("%d",&item);

   stack[++top] = item;

   printf("Element inserted successfully.\n");
}
```

```
void pop()
{
   if(top==-1)
   {
      printf("Stack Underflow\n");
      return;
   }

   printf("Deleted Element = %d\n",stack[top--]);
}

void peek()
{
   if(top==-1)
      printf("Stack is Empty\n");
   else
      printf("Top Element = %d\n",stack[top]);
}

void display()
{
   int i;

   if(top==-1)
   {
      printf("Stack is Empty\n");
      return;
   }

   printf("Stack Elements:\n");

   for(i=top;i>=0;i--)
```

```
        printf("%d\n",stack[i]);
}

int main()
{
   int choice;

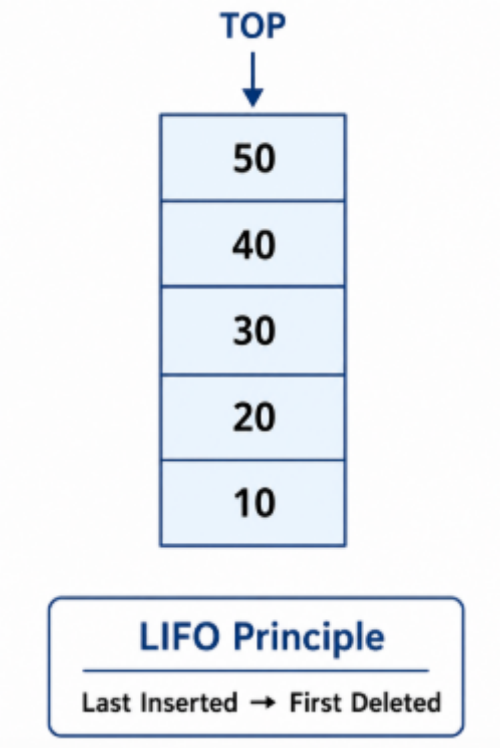
   do
   {
      printf("\n1.Push\n2.Pop\n3.Peek\n4.Display\n5.Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);

      switch(choice)
      {
         case 1: push(); break;
         case 2: pop(); break;
         case 3: peek(); break;
         case 4: display(); break;
         case 5: printf("Program Terminated");
                 break;
         default: printf("Invalid Choice");
      }

   }while(choice!=5);

   return 0;
}
```

DIAGRAM:



PROCEDURE:

1. Start the program.
2. Declare an array to represent the stack.
3. Initialize top = -1.
4. Display the menu.
5. Perform Push, Pop, Peek, or Display according to the user's choice.
6. Repeat until Exit is selected.
7. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Push	10	10 inserted
Push	20	20 inserted
Peek	-	20
Pop	-	20 deleted
Display	-	10

CALCULATIONS:

Maximum Stack Size = 5

If

Top = -1 → Stack Empty

Top = MAX-1 → Stack Full

Time Complexity:

- Push = O(1)
- Pop = O(1)
- Peek = O(1)
- Display = O(n)

Space Complexity = O(n)

RESULT:

The Stack was successfully implemented using an array. The operations Push, Pop, Peek, and Display were executed correctly, illustrating the LIFO (Last In, First Out) principle.

PRECAUTIONS:

1. Check for stack overflow before Push.
2. Check for stack underflow before Pop.
3. Initialize top = -1.
4. Do not exceed the declared array size.
5. Enter valid integer values.
6. Compile the program before execution.
7. Verify the output after each operation.

EXPERIMENT NO. 5

OBJECTIVE:

To understand the concept of the Queue data structure and implement its basic operations (Enqueue, Dequeue, Display, and Peek) using an array in C.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. The first element inserted into the queue is the first element removed.

Insertion is called Enqueue, while deletion is called Dequeue.

In an array implementation, two pointers are maintained:

- Front → First element
- Rear → Last element

Basic Operations

- Enqueue
- Dequeue
- Peek (Front)
- Display

Applications

- CPU Scheduling
- Printer Queue
- Ticket Reservation
- Breadth First Search
- Network Packet Scheduling
- Process Scheduling

Time Complexity

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)
Display	O(n)

C PROGRAM

```
#include<stdio.h>

#define MAX 5

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

void enqueue()
{
   int item;

   if(rear==MAX-1)
   {
      printf("Queue Overflow\n");
      return;
   }

   printf("Enter element: ");
   scanf("%d",&item);

   if(front==-1)
      front=0;

   queue[++rear]=item;
```

```
    printf("Element inserted successfully.\n");
}

void dequeue()
{
    if(front==-1 || front>rear)
    {
        printf("Queue Underflow\n");
        return;
    }

    printf("Deleted Element = %d\n",queue[front++]);

    if(front>rear)
    {
        front=-1;
        rear=-1;
    }
}

void peek()
{
    if(front==-1)
        printf("Queue Empty\n");
    else
        printf("Front Element = %d\n",queue[front]);
}

void display()
{
    int i;

    if(front==-1)
```

```
    {
        printf("Queue Empty\n");
        return;
    }

    printf("Queue Elements:\n");

    for(i=front;i<=rear;i++)
        printf("%d ",queue[i]);
}

int main()
{
    int choice;

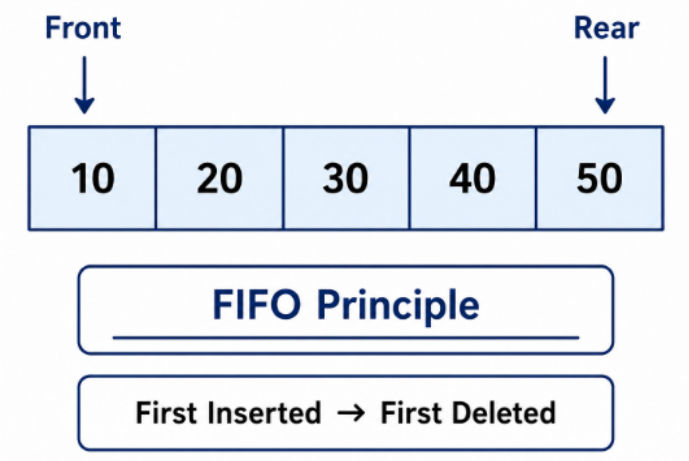
    do
    {
        printf("\n1.Enqueue\n2.Dequeue\n3.Peek\n4.Display\n5.Exit\n");
        printf("Enter Choice: ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueue(); break;
            case 2: dequeue(); break;
            case 3: peek(); break;
            case 4: display(); break;
            case 5: printf("Program Terminated");
                    break;
            default: printf("Invalid Choice");
        }

    }while(choice!=5);
```

```
    return 0;
}
```

DIAGRAM:



PROCEDURE:

1. Start the program.
2. Declare an array for the queue.
3. Initialize front = -1 and rear = -1.
4. Display the menu.
5. Perform Enqueue, Dequeue, Peek, or Display based on user input.
6. Repeat until Exit is selected.
7. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Enqueue	10	10 inserted
Enqueue	20	20 inserted
Peek	-	10
Dequeue	-	10 deleted
Display	-	20

CALCULATIONS:

Maximum Queue Size = 5

Initially:

- Front = -1
- Rear = -1

Conditions:

- Queue Empty → Front = -1
- Queue Full → Rear = MAX - 1

Time Complexity:

- Enqueue = O(1)
- Dequeue = O(1)
- Peek = O(1)
- Display = O(n)

Space Complexity = O(n)

RESULT:

The Queue was successfully implemented using an array. The operations Enqueue, Dequeue, Peek, and Display were executed correctly, demonstrating the FIFO (First In, First Out) principle.

PRECAUTIONS:

1. Check for queue overflow before Enqueue.
2. Check for queue underflow before Dequeue.
3. Initialize front and rear correctly.
4. Do not exceed the maximum queue size.
5. Enter valid integer inputs.
6. Compile the program before execution.
7. Verify the output after each queue operation.

EXPERIMENT NO. 6

OBJECTIVE:

To understand the concept of a Circular Queue and implement its basic operations (Enqueue, Dequeue, Peek, and Display) using an array in C.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A Circular Queue is a linear data structure that follows the FIFO (First In, First Out) principle. Unlike a simple queue, the last position of the queue is connected back to the first position, forming a circular structure.

In a normal queue, once the rear reaches the last index of the array, no further insertion is possible even if there are empty spaces at the beginning. A Circular Queue overcomes this limitation by reusing the vacant positions created after deletion.

The queue is maintained using two pointers:

- Front → Points to the first element.
- Rear → Points to the last element.

The rear pointer wraps around using the modulo (%) operator.

Basic Operations

- Enqueue
- Dequeue
- Peek
- Display

Advantages

- Efficient utilization of memory.
- Eliminates wastage of unused spaces.
- Suitable for CPU scheduling and buffering applications.

Applications

- CPU Scheduling

- Circular Buffer
- Printer Spooling
- Traffic Management Systems
- Multimedia Streaming

Time Complexity

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)
Display	O(n)

C PROGRAM

```
#include<stdio.h>

#define MAX 5

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

void enqueue()
{
   int item;

   if((rear + 1) % MAX == front)
   {
      printf("Circular Queue Overflow\n");
      return;
   }

   printf("Enter element: ");
   scanf("%d",&item);
```

```
    if(front == -1)
        front = 0;

    rear = (rear + 1) % MAX;
    queue[rear] = item;

    printf("Element inserted successfully.\n");
}

void dequeue()
{
    if(front == -1)
    {
        printf("Circular Queue Underflow\n");
        return;
    }

    printf("Deleted Element = %d\n",queue[front]);

    if(front == rear)
        front = rear = -1;
    else
        front = (front + 1) % MAX;
}

void peek()
{
    if(front == -1)
        printf("Queue Empty\n");
    else
        printf("Front Element = %d\n",queue[front]);
}
```

```
void display()
{
   int i;

   if(front == -1)
   {
      printf("Queue Empty\n");
      return;
   }

   printf("Queue Elements:\n");

   i = front;

   while(1)
   {
      printf("%d ",queue[i]);

      if(i == rear)
         break;

      i = (i + 1) % MAX;
   }
}

int main()
{
   int choice;

   do
   {
      printf("\n1.Enqueue\n2.Dequeue\n3.Peek\n4.Display\n5.Exit\n");
      printf("Enter Choice: ");
```

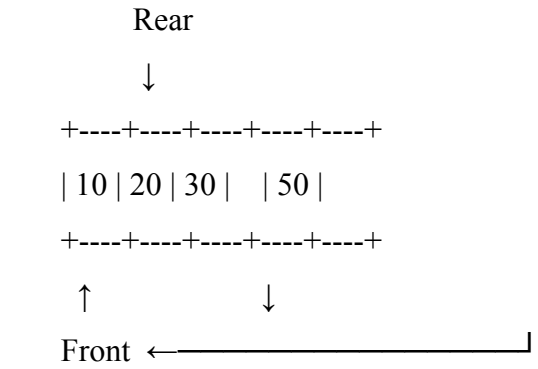
```
        scanf("%d",&choice);

        switch(choice)
        {
           case 1: enqueue(); break;
           case 2: dequeue(); break;
           case 3: peek(); break;
           case 4: display(); break;
           case 5: printf("Program Terminated");
                   break;
           default: printf("Invalid Choice");
        }

    }while(choice != 5);

    return 0;
}
```

DIAGRAM:



Circular Queue (FIFO)

PROCEDURE:

1. Start the program.
2. Declare an array and initialize front = -1 and rear = -1.
3. Display the menu.

4. Perform Enqueue, Dequeue, Peek, or Display according to the user's choice.
5. Use the modulo (%) operator to wrap the rear and front pointers.
6. Repeat until Exit is selected.
7. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Enqueue	10	10 inserted
Enqueue	20	20 inserted
Peek	-	10
Dequeue	-	10 deleted
Display	-	20

CALCULATIONS:

Maximum Queue Size = 5

Conditions:

- Queue Empty → front = -1
- Queue Full → (rear + 1) % MAX == front

Time Complexity

- Enqueue = O(1)
- Dequeue = O(1)
- Peek = O(1)
- Display = O(n)

Space Complexity = O(n)

RESULT:

The Circular Queue was successfully implemented using an array. The operations Enqueue, Dequeue, Peek, and Display were executed correctly, demonstrating efficient memory utilization and the FIFO (First In, First Out) principle.

PRECAUTIONS:

1. Check for queue overflow before insertion.

2. Check for queue underflow before deletion.
3. Use the modulo (%) operator correctly for circular movement.
4. Initialize front and rear properly.
5. Do not exceed the maximum queue size.
6. Enter valid integer values.
7. Verify the output after each operation.

EXPERIMENT NO. 7

OBJECTIVE:

To understand the implementation of a Stack using a Singly Linked List and perform the basic operations (Push, Pop, Peek, and Display) in C.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A Stack is a linear data structure that follows the LIFO (Last In, First Out) principle. Instead of using an array, a stack can also be implemented using a Linked List, where each node consists of two parts:

- Data
- Pointer to the next node

The Top pointer always points to the first node of the linked list.

Unlike array implementation, a linked-list stack does not have a fixed size. Memory is allocated dynamically during program execution using malloc().

Stack Operations

- Push
- Pop
- Peek
- Display

Advantages

- Dynamic memory allocation.
- No fixed size limitation.
- Efficient insertion and deletion.

Applications

- Expression Evaluation
- Function Calls

- Undo/Redo Operations
- Backtracking
- Browser History

Time Complexity

Operation	Time Complexity
Push	O(1)
Pop	O(1)
Peek	O(1)
Display	O(n)

C PROGRAM

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

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

struct node *top = NULL;

void push()
{
   struct node *newnode;

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

   printf("Enter element: ");
   scanf("%d",&newnode->data);

   newnode->next = top;
```

```
    top = newnode;

    printf("Element inserted successfully.\n");
}

void pop()
{
    struct node *temp;

    if(top == NULL)
    {
        printf("Stack Underflow\n");
        return;
    }

    temp = top;

    printf("Deleted Element = %d\n",temp->data);

    top = top->next;

    free(temp);
}

void peek()
{
    if(top == NULL)
        printf("Stack Empty\n");
    else
        printf("Top Element = %d\n",top->data);
}

void display()
```

```
{
   struct node *temp;

   if(top == NULL)
   {
      printf("Stack Empty\n");
      return;
   }

   temp = top;

   printf("Stack Elements:\n");

   while(temp != NULL)
   {
      printf("%d\n",temp->data);
      temp = temp->next;
   }
}

int main()
{
   int choice;

   do
   {
      printf("\n1.Push\n2.Pop\n3.Peek\n4.Display\n5.Exit\n");
      printf("Enter Choice: ");
      scanf("%d",&choice);

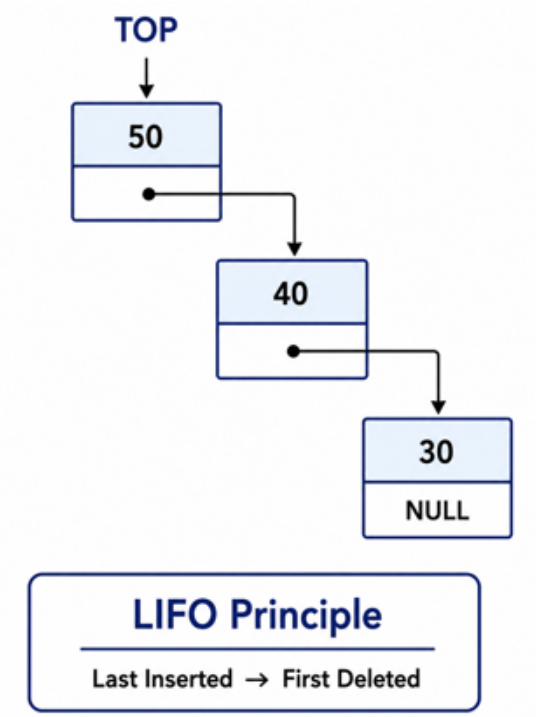
      switch(choice)
      {
         case 1: push(); break;
```

```
            case 2: pop(); break;
            case 3: peek(); break;
            case 4: display(); break;
            case 5: printf("Program Terminated");
                    break;
            default: printf("Invalid Choice");
        }

    }while(choice != 5);

    return 0;
}
```

DIAGRAM:



PROCEDURE:

1. Start the program.
2. Create a node structure containing data and next pointer.
3. Initialize top = NULL.
4. Display the menu.

5. Perform Push, Pop, Peek, or Display according to the user's choice.
6. Allocate memory dynamically using malloc() during Push.
7. Delete nodes using free() during Pop.
8. Repeat until Exit is selected.
9. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Push	10	Inserted
Push	20	Inserted
Peek	-	20
Pop	-	20 deleted
Display	-	10

CALCULATIONS:

Memory Required for One Node

= sizeof(int) + sizeof(pointer)

Time Complexity

- Push = O(1)
- Pop = O(1)
- Peek = O(1)
- Display = O(n)

Space Complexity = O(n)

RESULT:

The Stack was successfully implemented using a Singly Linked List. The operations Push, Pop, Peek, and Display were executed correctly, demonstrating the LIFO (Last In, First Out) principle and dynamic memory allocation.

PRECAUTIONS:

1. Check for stack underflow before Pop.
2. Allocate memory using malloc() before insertion.

3. Release memory using free() after deletion.
4. Initialize the top pointer to NULL.
5. Enter valid integer values.
6. Check the output after each operation.
7. Compile the program successfully before execution.

EXPERIMENT NO. 8

OBJECTIVE:

To understand the implementation of a Queue using a Singly Linked List and perform the basic operations Enqueue, Dequeue, Peek, and Display.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. The first element inserted into the queue is the first element removed.

A queue can be implemented using either an array or a linked list. In the linked list implementation, memory is allocated dynamically during runtime using malloc(), eliminating the limitation of a fixed-size array.

A queue implemented using a linked list maintains two pointers:

- Front – Points to the first node of the queue.
- Rear – Points to the last node of the queue.

Each node consists of:

- Data field
- Pointer to the next node

Queue Operations

- Enqueue: Inserts a new node at the rear of the queue.
- Dequeue: Removes the node from the front of the queue.
- Peek: Displays the front element.
- Display: Displays all elements from front to rear.

Advantages

- Dynamic memory allocation.
- No fixed size limitation.
- Efficient insertion and deletion.

- Better memory utilization.

Applications

- CPU Scheduling
- Printer Queue
- Ticket Reservation System
- Breadth First Search (BFS)
- Network Packet Scheduling
- Process Scheduling

Time Complexity

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)
Display	O(n)

C PROGRAM

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

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

struct node *front=NULL,*rear=NULL;

void enqueue()
{
   struct node *newnode;

   newnode=(struct node*)malloc(sizeof(struct node));
```

```
    printf("Enter Element: ");
    scanf("%d",&newnode->data);

    newnode->next=NULL;

    if(front==NULL)
    {
        front=rear=newnode;
    }
    else
    {
        rear->next=newnode;
        rear=newnode;
    }

    printf("Element Inserted Successfully.\n");
}

void dequeue()
{
    struct node *temp;

    if(front==NULL)
    {
        printf("Queue Underflow\n");
        return;
    }

    temp=front;

    printf("Deleted Element = %d\n",front->data);
```

```
    front=front->next;

    if(front==NULL)
        rear=NULL;

    free(temp);
}

void peek()
{
    if(front==NULL)
        printf("Queue Empty\n");
    else
        printf("Front Element = %d\n",front->data);
}

void display()
{
    struct node *temp;

    if(front==NULL)
    {
        printf("Queue Empty\n");
        return;
    }

    temp=front;

    printf("Queue Elements:\n");

    while(temp!=NULL)
    {
        printf("%d ",temp->data);
```

```
        temp=temp->next;
    }

    printf("\n");
}

int main()
{
    int choice;

    do
    {
        printf("\n1.Enqueue");
        printf("\n2.Dequeue");
        printf("\n3.Peek");
        printf("\n4.Display");
        printf("\n5.Exit");

        printf("\nEnter Choice: ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1: enqueue();
                    break;

            case 2: dequeue();
                    break;

            case 3: peek();
                    break;

            case 4: display();
```

```
            break;

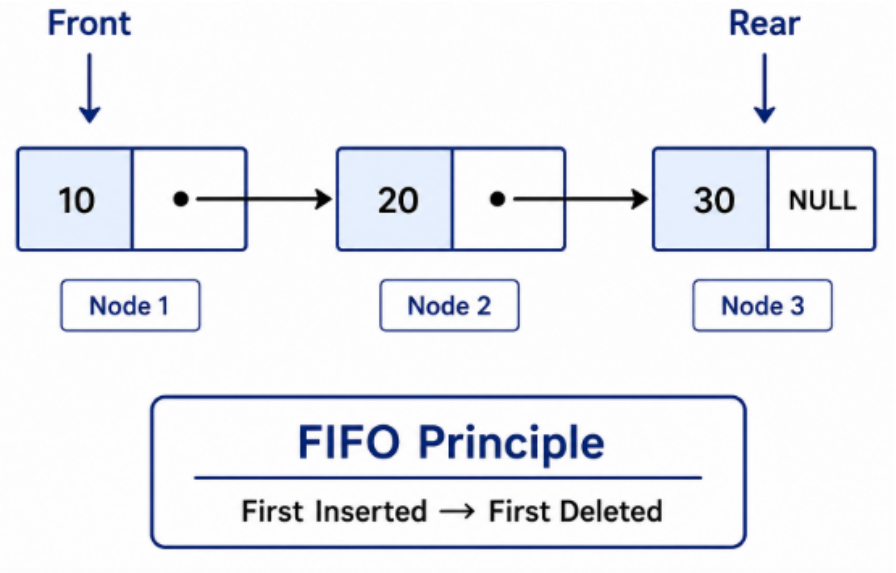
        case 5: printf("Program Terminated");
            break;

        default: printf("Invalid Choice");
    }

  }while(choice!=5);

  return 0;
}
```

DIAGRAM:



PROCEDURE:

1. Start the program.
2. Define a node containing data and next pointer.
3. Initialize front = rear = NULL.
4. Display the menu.
5. Insert elements using Enqueue.
6. Delete elements using Dequeue.
7. Display the front element using Peek.

8. Display all queue elements using Display.
9. Repeat the operations until Exit is selected.
10. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Enqueue	10	10 inserted
Enqueue	20	20 inserted
Enqueue	30	30 inserted
Peek	-	10
Dequeue	-	10 deleted
Display	-	20 30

CALCULATIONS:

For each node:

Memory Required

= sizeof(int) + sizeof(pointer)

Time Complexity

Operation	Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)
Display	O(n)

Space Complexity = O(n)

RESULT:

The Queue was successfully implemented using a Singly Linked List. The operations Enqueue, Dequeue, Peek, and Display were performed correctly, demonstrating the FIFO (First In, First Out) principle using dynamic memory allocation.

PRECAUTIONS:

1. Check for queue underflow before deletion.

2. Allocate memory using malloc() before insertion.
3. Release memory using free() after deletion.
4. Initialize front and rear pointers to NULL.
5. Update the rear pointer correctly after insertion.
6. When the queue becomes empty, set both front and rear to NULL.
7. Enter valid integer values.
8. Compile the program successfully before execution.
9. Verify the output after each queue operation.

EXPERIMENT NO. 9

OBJECTIVE:

To understand the implementation of a Circular Queue using a Singly Linked List and perform the basic operations Enqueue, Dequeue, Peek, and Display.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A Circular Queue is a linear data structure that follows the FIFO (First In, First Out) principle, where the last node is connected back to the first node, forming a circular structure.

Unlike a normal queue, the circular queue allows continuous traversal without reaching a NULL pointer. This implementation efficiently utilizes memory because there is no fixed size limitation, as memory is allocated dynamically using linked lists.

A Circular Queue implemented using a linked list maintains two pointers:

- Front – Points to the first node.
- Rear – Points to the last node.

The next pointer of the rear node always points to the front node, thereby maintaining the circular structure.

Queue Operations

- Enqueue: Inserts a node at the rear of the queue.
- Dequeue: Removes a node from the front of the queue.
- Peek: Displays the front element.
- Display: Traverses and displays all elements until the traversal reaches the front node again.

Advantages

- Dynamic memory allocation.
- Better memory utilization.

- No fixed queue size.
- Continuous circular traversal.
- Efficient insertion and deletion.

Applications

- Round Robin CPU Scheduling
- Circular Buffers
- Multimedia Streaming
- Process Scheduling
- Network Packet Scheduling
- Resource Sharing Systems

Time Complexity

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)
Display	O(n)

C PROGRAM

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

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

struct node *front = NULL;
struct node *rear = NULL;
```

```
void enqueue()
{
   struct node *newnode;

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

   printf("Enter Element: ");
   scanf("%d",&newnode->data);

   if(front == NULL)
   {
      front = rear = newnode;
      rear->next = front;
   }
   else
   {
      rear->next = newnode;
      rear = newnode;
      rear->next = front;
   }

   printf("Element Inserted Successfully.\n");
}

void dequeue()
{
   struct node *temp;

   if(front == NULL)
```

```
    {
        printf("Circular Queue Underflow\n");
        return;
    }

    if(front == rear)
    {
        printf("Deleted Element = %d\n",front->data);

        free(front);

        front = rear = NULL;
    }
    else
    {
        temp = front;

        printf("Deleted Element = %d\n",front->data);

        front = front->next;

        rear->next = front;

        free(temp);
    }
}

void peek()
{
    if(front == NULL)
```

```
        printf("Queue Empty\n");
    else
        printf("Front Element = %d\n",front->data);
}

void display()
{
    struct node *temp;

    if(front == NULL)
    {
        printf("Queue Empty\n");
        return;
    }

    temp = front;

    printf("Queue Elements:\n");

    do
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
    while(temp != front);

    printf("\n");
}

int main()
```

```
{
   int choice;

   do
   {
      printf("\n1.Enqueue");
      printf("\n2.Dequeue");
      printf("\n3.Peek");
      printf("\n4.Display");
      printf("\n5.Exit");

      printf("\nEnter Choice: ");
      scanf("%d",&choice);

      switch(choice)
      {
         case 1:
            enqueue();
            break;

         case 2:
            dequeue();
            break;

         case 3:
            peek();
            break;

         case 4:
            display();
```

```
            break;

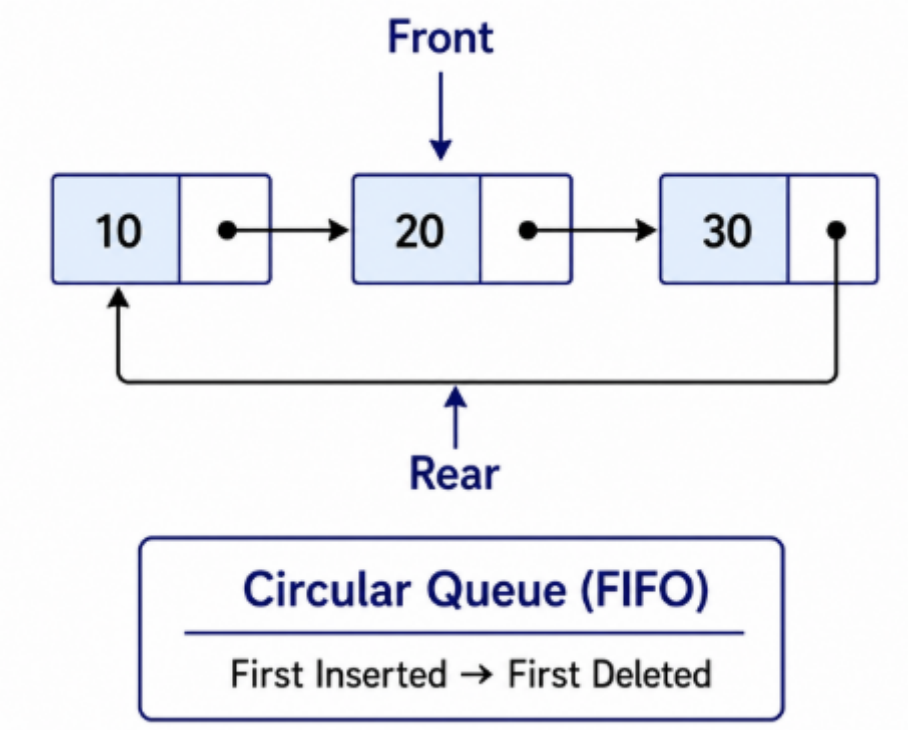
        case 5:
            printf("Program Terminated");
            break;

        default:
            printf("Invalid Choice");
    }

}while(choice != 5);

return 0;
}
```

DIAGRAM:



PROCEDURE:

1. Start the program.
2. Define a node containing data and next pointer.
3. Initialize front = rear = NULL.
4. Display the menu.
5. Insert elements using Enqueue.
6. Delete elements using Dequeue.
7. Display the front element using Peek.
8. Display all elements using Display by traversing until the front node is reached again.
9. Repeat the operations until Exit is selected.
10. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Enqueue	10	10 inserted
Enqueue	20	20 inserted
Enqueue	30	30 inserted
Peek	-	10
Dequeue	-	10 deleted
Display	-	20 30

CALCULATIONS:

Memory Required per Node:

= sizeof(int) + sizeof(pointer)

Complexity Analysis

Operation	Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)

Operation	Complexity
Display	O(n)

Space Complexity: O(n)

RESULT:

The Circular Queue was successfully implemented using a Singly Linked List. The operations Enqueue, Dequeue, Peek, and Display were executed correctly, demonstrating dynamic memory allocation and the FIFO (First In, First Out) principle with a circular linked structure.

PRECAUTIONS:

1. Initialize front and rear pointers to NULL.
2. Ensure that the rear node always points to the front node after every insertion and deletion.
3. Check for queue underflow before performing deletion.
4. Allocate memory using malloc() before inserting a node.
5. Release memory using free() after deleting a node.
6. While displaying the queue, stop traversal when the pointer reaches the front node again to avoid an infinite loop.
7. Enter valid integer values.
8. Compile the program successfully before execution.
9. Verify the output after each queue operation.

EXPERIMENT NO. 10

OBJECTIVE:

To understand the concept of Tree Data Structures and implement a Binary Search Tree (BST) in C with operations such as Insertion, Deletion, Searching, and Tree Traversals (Inorder, Preorder, and Postorder).

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A **Tree** is a hierarchical, non-linear data structure consisting of nodes connected by edges. The topmost node is called the **Root**, and each node may have one or more child nodes.

A **Binary Tree** is a tree in which each node has at most two children:

- Left Child
- Right Child

A **Binary Search Tree (BST)** is a special type of binary tree where:

- All elements in the left subtree are smaller than the root.
- All elements in the right subtree are greater than the root.
- Both left and right subtrees are themselves BSTs.

BSTs provide efficient searching, insertion, and deletion operations.

Tree Traversals

1. **Inorder Traversal (Left → Root → Right)**
   - Produces elements in sorted order.
2. **Preorder Traversal (Root → Left → Right)**
   - Used for copying trees.
3. **Postorder Traversal (Left → Right → Root)**
   - Used while deleting trees.

BST Operations

- Insertion
- Deletion
- Searching
- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Applications

- Database Indexing
- Expression Trees
- File Systems
- Decision Trees
- Dictionary Implementation
- Symbol Tables

Time Complexity

Operation	Average Case	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Traversal	O(n)	O(n)

C PROGRAM

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

struct node
{
   int data;
```

```
    struct node *left,*right;
};

struct node* createNode(int value)
{
    struct node *newnode;

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

    newnode->data=value;
    newnode->left=NULL;
    newnode->right=NULL;

    return newnode;
}

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;
}

struct node* minValue(struct node *node)
```

```
{
   while(node->left!=NULL)
      node=node->left;

   return node;
}

struct node* deleteNode(struct node *root,int key)
{
   struct node *temp;

   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
   {
      if(root->left==NULL)
      {
         temp=root->right;
         free(root);
         return temp;
      }

      else if(root->right==NULL)
```

```
        {
            temp=root->left;
            free(root);
            return temp;
        }

        temp=minValue(root->right);

        root->data=temp->data;

        root->right=deleteNode(root->right,temp->data);
    }

    return root;
}

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

void preorder(struct node *root)
{
    if(root!=NULL)
    {
```

```
        printf("%d ",root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

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

int main()
{
    struct node *root=NULL;

    root=insert(root,50);
    insert(root,30);
    insert(root,70);
    insert(root,20);
    insert(root,40);
    insert(root,60);
    insert(root,80);

    printf("Inorder Traversal : ");
    inorder(root);
```

```
    printf("\nPreorder Traversal : ");
    preorder(root);

    printf("\nPostorder Traversal : ");
    postorder(root);

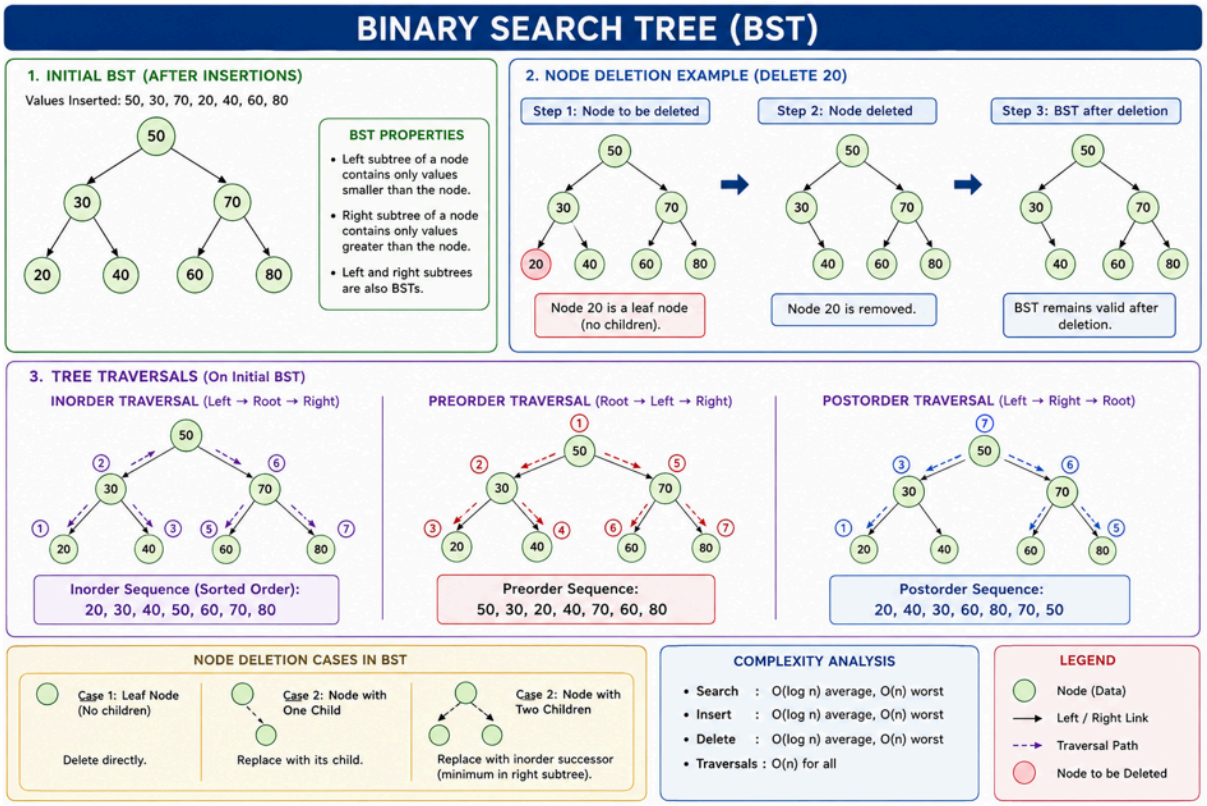
    root=deleteNode(root,20);

    printf("\n\nAfter Deletion (20)\n");

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

    return 0;
}
```

DIAGRAM:



PROCEDURE:

1. Start the program.
2. Define a node containing data, left pointer, and right pointer.
3. Create the Binary Search Tree.
4. Insert nodes into the BST.
5. Perform Inorder, Preorder, and Postorder traversals.
6. Delete the desired node from the BST.
7. Display the updated tree using Inorder Traversal.
8. Stop the program.

OBSERVATION TABLE:

Operation	Input	Output
Insert	50	Root Node
Insert	30	Left Child
Insert	70	Right Child

Operation	Input	Output
Insert	20	Left of 30
Insert	40	Right of 30
Delete	20	Node Deleted
Inorder	-	30 40 50 60 70 80

CALCULATIONS:

For a BST containing **n** nodes:

Time Complexity

Operation	Average	Worst
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Traversal	O(n)	O(n)

Space Complexity

- Tree Storage = **O(n)**
- Recursive Traversal = **O(h)**

where **h** is the height of the tree.

RESULT:

The Binary Search Tree (BST) was successfully implemented in C. The operations Insertion, Deletion, Inorder Traversal, Preorder Traversal, and Postorder Traversal were executed correctly, demonstrating the working of tree data structures and Binary Search Trees.

PRECAUTIONS:

1. Insert only valid integer values.
2. Avoid inserting duplicate values unless specifically allowed.
3. Handle memory allocation using malloc() carefully.
4. Free allocated memory whenever nodes are deleted.
5. Ensure BST properties are maintained during insertion and deletion.

6. Verify traversal outputs after every modification.
7. Compile the program without syntax errors before execution.
8. Test deletion cases for leaf nodes, nodes with one child, and nodes with two children.

EXPERIMENT NO. 11

OBJECTIVE:

To understand the implementation of **Graphs** using an adjacency matrix and perform graph traversal using **Breadth First Search (BFS)** and **Depth First Search (DFS)**. Also, to determine the **Minimum Cost Spanning Tree (MST)** using **Prim's Algorithm** and the **Shortest Path** using **Dijkstra's Algorithm**.

APPARATUS REQUIRED:

- Computer System
- GCC Compiler / Turbo C / Code::Blocks / Dev-C++
- C Programming Environment
- Windows/Linux Operating System

THEORY:

A **Graph** is a non-linear data structure consisting of a finite set of **vertices (nodes)** and **edges** connecting pairs of vertices. Graphs are widely used to model relationships among objects, such as road networks, social networks, communication systems, and computer networks.

A graph is represented as:

[
G=(V,E)
]

where:

- **V** = Set of vertices
- **E** = Set of edges

Graphs may be:

- Directed Graph
- Undirected Graph
- Weighted Graph
- Unweighted Graph

Graph Representation

The most common graph representations are:

1. Adjacency Matrix
2. Adjacency List

In this experiment, the graph is represented using an **Adjacency Matrix**, where:

- graph[i][j] = 1 indicates an edge exists.
- graph[i][j] = 0 indicates no edge.

For weighted graphs,

graph[i][j] = weight of edge

Breadth First Search (BFS)

Breadth First Search visits all neighboring vertices before moving to the next level.

It uses a **Queue**.

Algorithm

1. Start from the source vertex.
2. Mark it as visited.
3. Insert it into the queue.
4. Remove one vertex from the queue.
5. Visit all adjacent unvisited vertices.
6. Insert them into the queue.
7. Repeat until the queue becomes empty.

Applications

- Network Broadcasting
- Shortest Path (Unweighted Graph)
- Web Crawling
- GPS Navigation

Depth First Search (DFS)

Depth First Search visits a path completely before backtracking.

It uses either

- Stack

- Recursion

Algorithm

1. Visit the source node.
2. Mark it as visited.
3. Visit one adjacent node recursively.
4. Continue until no adjacent node exists.
5. Backtrack.
6. Repeat until all vertices are visited.

Applications

- Cycle Detection
- Topological Sorting
- Maze Solving
- Connected Components

Minimum Cost Spanning Tree (Prim's Algorithm)

A **Minimum Spanning Tree (MST)** is a spanning tree having the minimum total edge weight.

Prim's Algorithm constructs the MST by repeatedly selecting the minimum-weight edge connecting a visited vertex to an unvisited vertex.

Applications

- Network Design
- Telephone Networks
- Electrical Distribution
- Road Construction

Time Complexity

$O(V^2)$

Shortest Path (Dijkstra's Algorithm)

Dijkstra's Algorithm computes the shortest distance from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

Algorithm

1. Assign infinite distance to every vertex.
2. Set source distance = 0.
3. Select the unvisited vertex with the smallest distance.
4. Update distances of adjacent vertices.
5. Repeat until all vertices are visited.

Applications

- GPS Navigation
- Internet Routing
- Flight Scheduling
- Robot Navigation

Time Complexity

$O(V^2)$

Time Complexity

Algorithm	**Time Complexity**
BFS	O(V + E)
DFS	O(V + E)
Prim's Algorithm	$O(V^2)$
Dijkstra's Algorithm	$O(V^2)$

C PROGRAM

```
#include<stdio.h>
#define MAX 10
#define INF 9999

int graph[MAX][MAX];
int visited[MAX];
int queue[MAX];
int front=-1,rear=-1;
```

```
int n;

void bfs(int start)
{
   int i;

   for(i=0;i<n;i++)
      visited[i]=0;

   front=rear=-1;

   visited[start]=1;

   queue[++rear]=start;

   printf("\nBFS Traversal : ");

   while(front!=rear)
   {
      int v=queue[++front];

      printf("%d ",v);

      for(i=0;i<n;i++)
      {
         if(graph[v][i]!=0 && visited[i]==0)
         {
            visited[i]=1;
            queue[++rear]=i;
         }
```

```
        }
    }
}

void dfs(int v)
{
    int i;

    visited[v]=1;

    printf("%d ",v);

    for(i=0;i<n;i++)
    {
        if(graph[v][i]!=0 && visited[i]==0)
            dfs(i);
    }
}

void prim()
{
    int selected[MAX]={0};
    int edges=0;
    int x,y;
    int min;

    selected[0]=1;

    printf("\n\nEdges in MST\n");
```

```
    while(edges<n-1)
    {
        min=INF;

        for(int i=0;i<n;i++)
        {
            if(selected[i])
            {
                for(int j=0;j<n;j++)
                {
                    if(!selected[j] && graph[i][j])
                    {
                        if(min>graph[i][j])
                        {
                            min=graph[i][j];
                            x=i;
                            y=j;
                        }
                    }
                }
            }
        }

        printf("%d - %d = %d\n",x,y,graph[x][y]);

        selected[y]=1;

        edges++;
    }
}
```

```
void dijkstra(int source)
{
   int distance[MAX];
   int visited[MAX]={0};

   for(int i=0;i<n;i++)
      distance[i]=INF;

   distance[source]=0;

   for(int count=0;count<n-1;count++)
   {
      int min=INF,u;

      for(int i=0;i<n;i++)
      {
         if(!visited[i] && distance[i]<min)
         {
            min=distance[i];
            u=i;
         }
      }

      visited[u]=1;

      for(int v=0;v<n;v++)
      {
         if(!visited[v] && graph[u][v] &&
            distance[u]+graph[u][v]<distance[v])
```

```
            {
                distance[v]=distance[u]+graph[u][v];
            }
        }
    }

    printf("\n\nShortest Distance from Source %d\n",source);

    for(int i=0;i<n;i++)
        printf("%d -> %d = %d\n",source,i,distance[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);
```

```
    bfs(start);

    for(i=0;i<n;i++)
        visited[i]=0;

    printf("\nDFS Traversal : ");

    dfs(start);

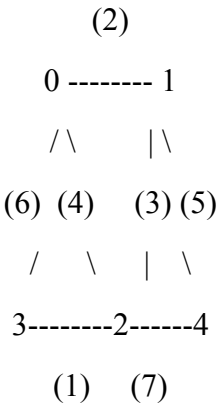
    prim();

    dijkstra(start);

    return 0;
}
```

DIAGRAMS:

Figure 1: Sample Weighted Graph



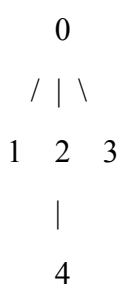
Vertices: 0, 1, 2, 3, 4

Weighted Edges:

- 0 – 1 = 2
- 0 – 2 = 4
- 0 – 3 = 6
- 1 – 2 = 3
- 1 – 4 = 5
- 2 – 3 = 1
- 2 – 4 = 7

Figure 2: Breadth First Search (BFS)

Starting Vertex = **0**



Traversal Order

0 → 1 → 2 → 3 → 4

Explanation

- Visit root (0)

- Visit all adjacent vertices
- Continue level by level

Uses **Queue (FIFO)**

Figure 3: Depth First Search (DFS)



Traversal Order

0 → 1 → 2 → 3 → 4

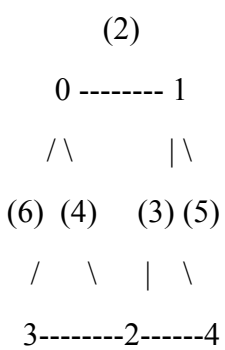
Explanation

- Visit one branch completely
- Backtrack
- Visit remaining branches

Uses **Stack / Recursion**

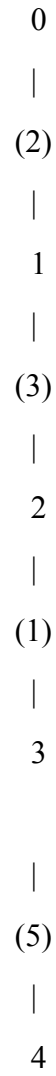
Figure 4: Minimum Spanning Tree (Prim's Algorithm)

Original Graph



(1) (7)

Minimum Spanning Tree



Selected Edges

0 – 1

1 – 2

2 – 3

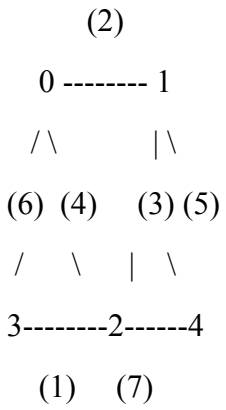
1 – 4

Minimum Cost

2 + 3 + 1 + 5 = 11

Figure 5: Dijkstra's Shortest Path

Source Vertex = **0**



Shortest Distance Table

Vertex	Distance from Source
0	0
1	2
2	4
3	5
4	7

Shortest Paths

0

↓

1

↓

2

↓

3

0

↓

1

↓

4

PROCEDURE:

Graph Creation

1. Start the program.
2. Enter the number of vertices.
3. Input the adjacency matrix representing the graph.
4. Specify the starting vertex.

BFS

5. Initialize all vertices as unvisited.
6. Insert the source vertex into the queue.
7. Visit vertices level by level.
8. Continue until the queue becomes empty.

DFS

9. Mark the source vertex as visited.
10. Visit adjacent vertices recursively.
11. Backtrack whenever necessary.
12. Continue until all vertices are visited.

Prim's Algorithm

13. Select an initial vertex.
14. Find the minimum weight edge connecting a visited vertex to an unvisited vertex.
15. Repeat until all vertices are included in the spanning tree.

Dijkstra's Algorithm

16. Assign infinity to all vertices.
17. Set source distance as zero.
18. Select the nearest unvisited vertex.
19. Update the distances of adjacent vertices.
20. Repeat until every vertex is processed.
21. Display the shortest distances.
22. Stop the program.

OBSERVATION TABLE:

Graph Input

Parameter	**Value**
Number of Vertices	5
Starting Vertex	0

BFS Traversal

Step	**Queue**	**Visited Vertex**
1	0	0
2	1,2,3	1
3	2,3	2
4	3,4	3
5	4	4

Traversal

0 → 1 → 2 → 3 → 4

DFS Traversal

Step	Visited Vertex
1	0
2	1
3	2
4	3
5	4

Traversal

0 → 1 → 2 → 3 → 4

Prim's MST

Selected Edge	Weight
0 – 1	2
1 – 2	3
2 – 3	1
1 – 4	5

Total Cost

11

Dijkstra's Algorithm

Vertex	Shortest Distance
0	0
1	2
2	4
3	5
4	7

CALCULATIONS:

Breadth First Search

Time Complexity

O(V + E)

Space Complexity

O(V)

Depth First Search

Time Complexity

O(V + E)

Space Complexity

O(V)

Prim's Algorithm

Time Complexity

$O(V^2)$

Space Complexity

O(V)

Dijkstra's Algorithm

Time Complexity

$O(V^2)$

Space Complexity

O(V)

Total Weight of Minimum Spanning Tree

Edge (0–1) = 2

Edge (1–2) = 3

Edge (2–3) = 1

Edge (1–4) = 5

Total Cost = 2 + 3 + 1 + 5 = 11

RESULT:

The graph was successfully implemented using an adjacency matrix. The **Breadth First Search (BFS)** and **Depth First Search (DFS)** traversals were performed correctly. The **Minimum Cost Spanning Tree (MST)** was obtained using **Prim's Algorithm**, and the shortest paths from the source vertex were successfully determined using **Dijkstra's Algorithm**.

PRECAUTIONS:

1. Enter the correct number of vertices before creating the graph.
2. Ensure that the adjacency matrix accurately represents the graph.
3. Use non-negative edge weights for Dijkstra's Algorithm.
4. Ensure the graph is connected while applying Prim's Algorithm.
5. Mark visited vertices correctly to avoid repeated traversal.
6. Initialize all arrays before execution.
7. Verify the adjacency matrix before running the algorithms.
8. Compile the program successfully before execution.
9. Compare the traversal output with the expected result.
10. Verify the total cost of the Minimum Spanning Tree and the shortest path distances.