RUSTAMJI INSTITUTE OF TECHNOLOGY

BSF ACADEMY, TEKANPUR

Practical File for

CS303 (Data Structure)

Submitted by

Kishan Kumar (0902CS231051)
B.Tech. Computer Science & Engineering 3rd Semester
(2023-2027 batch)

Subject Teacher
Dr. Jagdish Makhijani

File Checked by
Mr. Yashwant Pathak

Self-Declaration Certificate

I, **Kishan Kumar** , hereby declare that I have completed the lab work of CS303 (Data Structure) at my own effort and understanding.

I affirm that the work submitted is my own, and I take full responsibility for its authenticity and originality.

Date:

Kishan Kumar

[0902CS231051]

ENVORIONMENT USED

Hardware Configuration : <LAPTOP-3CDBOK0P(64-bit OS,x64 processor) >

C Compiler : GCC Compiler

User Interface : <VS CODE>

GROUP MEMBERS

Member-1 : **Kishan Kumar(0902cs231051)**

https://github.com/0902kishan/Data-structure-and-algorithms-.git

Member-2 : **Nikhil B(0902cs231062)**

https://github.com/29N11/Data-Structures-and-Algorithms.git

Member-3 : **Yogesh Patel (0902cs23137)**

https://github.com/Yogesh02545/Data-Structure-.git

Member-4 : **Srashti Jain(0902cs231117)**

https://github.com/Srashti-1/Data-Structure-.git

TABLE OF CONTENTS

Section-A (Linked List)

Section-B (Stack)

Section-C (Queue)

Section-D (Trees & Graphs)

S. No.	Practical Description	Page Nos.	COs
1	Implementation of Binary Search Tree.		CO-3
2	Conversion of BST PreOrder/PostOrder/InOrder.		CO-3
3	Implementation of Kruskal Algorithm		CO-4
4	Implementation of Prim Algorithm		CO-4
5	Implementation of Dijkstra Algorithm		CO-4

Section-E (Sorting & Searching)

S. No.	Practical Description	Page Nos.	COs
1	Implementation of Sorting a. Bubble b. Selection c. Insertion d. Quick e. Merge		CO-5
2	Implementation of Binary Search on a list of numbers stored in an Array		CO-5
3	Implementation of Binary Search on a list of strings stored in an Array		CO-5
4	Implementation of Linear Search on a list of strings stored in an Array OR Implementation of Binary Search on a list of strings stored in a Single Linked List		CO-5

Experiment No.: 1

Program Description:

Implementation of Linked List using array.

Solution:

```
#include <iostream>

using namespace std;


struct Node {

    int data;

    Node* next;

    Node(int d)

    {

        data = d;

        next = NULL;

    }

};


// Function to insert node at the end

Node* insertEnd(Node* root, int item)

{

    Node* temp = new Node(item);

    if (root == NULL)

        return temp;

    Node* last = root;

    while (last->next != NULL) {

        last = last->next;

    }
```

```
    last->next = temp;
    return root;
}
Node* arrayToList(int arr[], int n)
{
    Node* root = NULL;
    for (int i = 0; i < n; i++) {
        root = insertEnd(root, arr[i]);
    }
    return root;
}
void display(Node* root)
{
    while (root != NULL) {
        cout << root->data << " ";
        root = root->next;
    }
}

// Driver code
int main()
{
    int arr[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    Node* root = arrayToList(arr, n);
    display(root);
    return 0;
}
```

Output: 1 2 3 4 5

Experiment No.: 2

Program Description:

Implementation of Linked List using Pointers.

Solution:

```
#include <iostream>
using namespace std;
class node{
public:
   int data;
   node* next;
   node(int val){
   data=val;
   next=NULL;
  }
};
void insertAtTail(node* &head,int val){
node* n=new node(val);
if(head==NULL){
   return;
}
node* temp=head;
while(temp->next!=NULL)
{
  temp=temp->next;
}
temp->next=n;
}
```

```
void print(node* head){
node* temp = head;
while(temp!=NULL){
   cout<<temp->data<<" " ;
   temp=temp->next;
}
}
int main(){
   node*head = new node(1);
   insertAtTail(head,2);
   insertAtTail(head,3);
   insertAtTail(head,4);
   print(head);
   return 0;
}
```

Output:
```
Linked List:
1 2 3 4 5
Dummy pointer pointing to head of Linked List:
-1 1 2 3 4 5
```

Experiment No.: 3

Program Description:

Implementation of Doubly Linked List using Pointers.

Solution:

```
#include <iostream>
using namespace std;
class node
{
public:
    int data;
    node *next;
    node *prev;
    node(int x)
    {
        data = x;
        prev = NULL;
        next = NULL;
    }
};
void print(node *head)
{
    node *temp = head;
    while (temp != NULL)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
```

```
}
void insertatlast(node *head, int value)
{
    node *p = new node(value);
    node *temp = head;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = p;
    p->prev = temp;
    temp = p;
}
void insertathead(node *&head, int val)
{
    node *p = new node(val);
    p->next = head;
    head->prev = p;
    head = p;
}
void insertatindex(node *&head, int val, int index)
{
    node *p = head;
    node *node1 = new node(val);
    int i = 0;
    while (i < index - 1)
    {
        p = p->next;
```

```
        i++;
    }
    node1->next = p->next;
    p->next->prev = node1;
    p->next = node1;
    node1->prev = p;
}
int main()
{   node *head = new node(12);
    node *temp1 = new node(14);
    node *temp2 = new node(16);
    node *temp3 = new node(18);
    node *temp4 = new node(20);
    head->next = temp1;
    temp1->next = temp2;
    temp2->next = temp3;
    temp3->next = temp4;
    temp4->next = NULL;
    print(head);
    cout << endl;

    insertathead(head, 24);
    cout << "linkedlist after insertion " << endl;

    insertatlast(head, 50);
    cout << "linkedlist after insertion " << endl;
    print(head);
```

```
    insertatindex(head, 15, 2);
    cout << "linkedlist after insertion " << endl;
    print(head);
    return 0;
}
```

Output:
```
Forward Traversal:
1 2 3
Backward Traversal:
3 2 1
```

Experiment No.: 4

Program Description:

Implementation of Circular Single Linked List using Pointers.

Solution:

```
#include<bits/stdc++.h>

using namespace std;

// Doubly linked list node

struct node

{

    int data;

    struct node *next;

    struct node *prev;

};

// Utility function to create a node in memory

struct node* getNode()

{

    return ((struct node *)malloc(sizeof(struct node)));

}

// Function to display the list

int displayList(struct node *temp)

{

    struct node *t = temp;

    if(temp == NULL)

        return 0;
```

```
    else
    {
      cout<<"The list is: ";
       while(temp->next != t)
       {
          cout<<temp->data<<" ";
          temp = temp->next;
       }
       cout<<temp->data;
       return 1;
    }
}

// Function to convert array into list
void createList(int arr[], int n, struct node **start)
{
    // Declare newNode and temporary pointer
    struct node *newNode,*temp;
    int i;

    // Iterate the loop until array length
    for(i=0;i<n;i++)
    {
       // Create new node
       newNode = getNode();

       // Assign the array data
       newNode->data = arr[i];
```

```
        // If it is first element
        // Put that node prev and next as start
        // as it is circular
        if(i==0)
        {
            *start = newNode;
            newNode->prev = *start;
            newNode->next = *start;
        }
        else
        {
            // Find the last node
            temp = (*start)->prev;

            // Add the last node to make them
            // in circular fashion
            temp->next  =  newNode;
            newNode->next = *start;
            newNode->prev  =  temp;
            temp = *start;
            temp->prev = newNode;
        }
    }
}

// Driver Code
int main()
```

```
{
    // Array to be converted
   int arr[] = {1,2,3,4,5};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Start Pointer
    struct node *start = NULL;

// Create the List
    createList(arr, n, &start);

    // Display the list
    displayList(start);
    return 0;
}
```

Output: `List after insertion: 1`

Experiment No.: 5

Program Description:

Implementation of Circular Doubly Linked List using Pointers.

Solution:

```
#include <iostream>

using namespace std;


class node

{

public:

    int data;

    node *next;

    node(int val)

    {

        this->data = val;

        this->next = NULL;

    }

    ~node()

    {

        int value = this->data;

        if (this->next != NULL)

        {

            delete next;

            next = NULL;

        }

    }

};
```

```
// creation of circular linkedlist

void insertatnode(node *&tail, int index, int el)
{
    if (tail == NULL)
    {
        node *newNode = new node(el);
        tail = newNode;
        newNode->next = newNode;
    }
    else
    {
        node *temp = tail;
        while (temp->next != tail)
        {
            temp = temp->next;
        }
        node *curr = new node(el);
        curr->next = temp->next;
        temp->next = curr;
    }
}

//traversing linkedlist
void print(node *tail)
{
    node *temp = tail;
    do
```

```
    {
        cout << tail->data << " ";
        tail = tail->next;
    } while (tail != temp);
    cout << endl;
}
int main()
{
    node *tail = NULL;
    insertatnode(tail, 5, 3);
    print(tail);

    insertatnode(tail, 3, 4);
    print(tail);

    insertatnode(tail, 4, 9);
    print(tail);

    insertatnode(tail, 9, 7);
    print(tail);

    insertatnode(tail, 4, 5);
    print(tail);
return 0;
}
```

Output: `5 10 20 30`

Experiment No.: 1

Program Description:

Implementation of Stack using Array.

Solution:

```
#include <iostream>
#include <stack>
using namespace std;
class Stack
{
public:
   int *arr, top, n;
   bool isEmpty();
   int size();

   //where n is the size of the array
   Stack(int x)
   {
      n = x;
      arr = new int[x];
      top = -1;
   }
   Stack(){}
   // function to push the element in an stack
   void push(int element)
   {
      if (n- top > 1)
      {
```

```
            top++;
            arr[top] = element;
        }
        else
        {
            cout << "overflow condition " << endl;
        }
    }
    void pop(){
        if(top>=0){
            top--;
        }
        else{
            cout<<"underflow condition  "<<endl;
        }
    }
    //display the top element of the stack
    int peek(){
        if(top>=0){
            return arr[top];
        }
        else{
            cout<<"stack is empty "<<endl;
        }
    }
};
int main()
{
```

```
class Stack st(5);
st.push(1);
cout << "The element at top is : "<< st.peek() << endl;
st.push(2);
cout << "The element at top is : "<< st.peek() << endl;
st.push(3);
cout << "The element at top is : "<< st.peek() << endl;
st.push(4);
cout << "The element at top is : "<< st.peek() << endl;
st.push(5);
cout << "The element at top is : "<< st.peek() << endl;
st.pop();
st.pop();
cout<<"after popping of an element the element at top is : "<< st.peek() << endl;
return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Top element is : 20
Elements present in stack : 20 10
```

Experiment No.: 2

Program Description:

Implementation of Stack using Pointers.

Solution:

```
#include <bits/stdc++.h>

using namespace std;


// Class representing a node in the linked list

class Node {

public:

    int data;

    Node* next;

    Node(int new_data) {
        this->data = new_data;

        this->next = nullptr;

    }

};


// Class to implement stack using a singly linked list

class Stack {


    // head of the linked list

    Node* head;


public:

    // Constructor to initialize the stack

    Stack() { this->head = nullptr; }
```

```
    // Function to check if the stack is empty
    bool isEmpty() {

        // If head is nullptr, the stack is empty
        return head == nullptr;
    }

    // Function to push an element onto the stack
   void push(int new_data) {

        // Create a new node with given data
        Node* new_node = new Node(new_data);

        // Check if memory allocation for the new node
        // failed
        if (!new_node) {
            cout << "\nStack Overflow";
        }

        // Link the new node to the current top node
        new_node->next = head;
      // Update the top to the new node
head = new_node;
    }

    // Function to remove the top element from the stack
      void pop() {
```

```
    // Check for stack underflow
    if (this->isEmpty()) {
        cout << "\nStack Underflow" << endl;
    }
    else {
        // Assign the current top to a temporary
        // variable
        Node* temp = head;

        // Update the top to the next node
        head = head->next;

        // Deallocate the memory of the old top node
        delete temp;
    }
}

// Function to return the top element of the stack
  int peek() {

   // If stack is not empty, return the top element
    if (!isEmpty())
        return head->data;
    else {
        cout << "\nStack is empty";
        return INT_MIN;
    }
  }
```

```
    };

// Driver program to test the stack implementation
int main() {
  // Creating a stack
  Stack st;

  // Push elements onto the stac
  st.push(11);
  st.push(22);
  st.push(33);
  st.push(44);

  // Print top element of the stack
  cout << "Top element is " << st.peek() << endl;

  // removing two elemements from the top
  cout << "Removing two elements..." << endl;
  st.pop();
  st.pop();

  // Print top element of the stack
  cout << "Top element is " << st.peek() << endl;
  return 0;
}
```

Output:
```
Pushed 10 to stack
Pushed 20 to stack
Pushed 30 to stack
```

```
Top element is: 30
Elements present in stack : 30 20 10
```

Experiment No.: 3

Program Description:

Program for Tower of Hanoi using recursion.

Solution:

```
#include <bits/stdc++.h>

using namespace std;


void towerOfHanoi(int n, char from_rod, char to_rod,

                  char aux_rod)

{

    if (n == 0) {

        return;

    }

    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);

    cout << "Move disk " << n << " from rod " << from_rod

         << " to rod " << to_rod << endl;

    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);

}


// Driver code

int main()

{

    int N = 3;


    // A, B and C are names of rods

    towerOfHanoi(N, 'A', 'C', 'B');

    return 0;
```

```
}
```

Output:
```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

Experiment No.: 4

Program Description:

Program to find out factorial of given number using recursion. Also show the various states of stack using in this program.

Solution:

```
#include <iostream>

using namespace std;


// Define a function to calculate factorial

// recursively

long long factorial(int n)

{

   // Base case - If n is 0 or 1, return 1

   if (n == 0 || n == 1) {

      return 1;

   }

   // Recursive case - Return n multiplied by

   // factorial of (n-1)


   return n * factorial(n - 1);

}


int main()

{

   int num = 5;

   // printing the factorial

   cout << "Factorial of " << num << " is " << factorial(num) << endl;
```

```
    return 0;
}
```

Output: `Factorial of 5 is 120`

Experiment No.: 1

Program Description:

Implementation of Queue using Array.

Solution:

```
#include <iostream>
#define SIZE 5 // Define the maximum size of the queue
using namespace std;
class Queue {
private:
   int arr[SIZE]; // Array to store the queue
   int front;     // Index of the front element
   int rear;      // Index of the rear element


public:
   Queue() {
      front = -1;
      rear = -1;
   }
   // Function to check if the queue is empty
   bool isEmpty() {
      return (front == -1);
   }
   // Function to check if the queue is full
   bool isFull() {
      return (rear == SIZE - 1);
   }
   // Function to add an element to the queue
```

```
void enqueue(int value) {
    if (isFull()) {
        cout << "Queue is full! Cannot enqueue " << value << endl;
        return;
    }
    if (isEmpty()) {
        front = 0; // Initialize front if queue was empty
    }
    arr[++rear] = value;
    cout << "Enqueued " << value << endl;
}
// Function to remove an element from the queue
void dequeue() {
     if (isEmpty()) {
        cout << "Queue is empty! Cannot dequeue." << endl;
        return;
    }
    cout << "Dequeued " << arr[front] << endl;
    if (front == rear) {
        // Reset the queue when the last element is dequeued
        front = -1;
        rear = -1;
    } else {
        front++;
    }
}
// Function to display the elements in the queue
void display() {
```

```
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return;
        }
        cout << "Queue elements: ";
        for (int i = front; i <= rear; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    q.dequeue();
    q.display();
    q.enqueue(40);
    q.enqueue(50);
    q.enqueue(60); // Attempt to enqueue beyond capacity
    q.display();
    return 0;
}
```

Output: `After Enqueueing:`

```
Front element: 1
Rear element: 3
Queue:  1 2 3
Queue:  1 2 3 4 5

Dequeueing elements:
Dequeued element: 1
Dequeued element: 2

After Dequeueing:
Front element: 3
Rear element: 6
Queue:  3 4 5 6
```

<u>Experiment No.: 2</u>

Program Description:

Implementation of Queue using Pointers.

Solution:

```
#include <iostream>
using namespace std;

// Node structure for the queue
struct Node {
    int data;       // Value of the node
    Node* next;     // Pointer to the next node

    // Constructor to initialize a new node
    Node(int val) {
        data = val;
        next = nullptr;
    }
};

// Queue class
class Queue {
private:
    Node* front; // Pointer to the front node
    Node* rear;  // Pointer to the rear node

public:
    // Constructor to initialize the queue
```

```
Queue() {
    front = nullptr;
    rear = nullptr;
}

// Function to check if the queue is empty
bool isEmpty() {
    return front == nullptr;
}

// Function to add an element to the queue
void enqueue(int value) {
    Node* newNode = new Node(value); // Create a new node
    if (isEmpty()) {
        // If the queue is empty, both front and rear point to the new node
        front = rear = newNode;
    } else {
        // Add the new node at the rear and update the rear pointer
        rear->next = newNode;
        rear = newNode;
    }
    cout << "Enqueued: " << value << endl;
}

// Function to remove an element from the queue
void dequeue() {
            cout << "Queue is empty! Cannot dequeue." << endl;
        return;
    }
```

```
    Node* temp = front;       // Temporary pointer to the front node
    front = front->next;      // Move the front pointer to the next node
    cout << "Dequeued: " << temp->data << endl;
    delete temp;              // Delete the old front node

    if (front == nullptr) {
        // If the queue is empty after dequeue, reset rear to nullptr
        rear = nullptr;
    }
}

// Function to get the front element of the queue
int peek() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return -1;
    }
    return front->data;
}

// Function to display the elements in the queue
void display() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return;
    }
    Node* temp = front;
    cout << "Queue elements: ";
```

```
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Destructor to free memory
    ~Queue() {
        while (!isEmpty()) {
            dequeue();
        }
    }
};

int main() {
    Queue q;

    q.enqueue(5);
    q.enqueue(15);
    q.enqueue(25);
    q.display();

    cout << "Front element: " << q.peek() << endl;

    q.dequeue();
    q.display();
```

```
    q.enqueue(35);
    q.enqueue(45);
    q.display();


    q.dequeue();
    q.dequeue();
    q.dequeue();
    q.dequeue(); // Attempt to dequeue when queue is empty
    return 0;
}
```

Output:
```
Front element is: 10
Front element is: 20
Queue is empty: 1
```

Experiment No.: 3

Program Description:

Implementation of Circular Queue using Array.

Solution:

```
#include <iostream>

using namespace std;


class CircularQueue {

private:

    int *queue;      // Pointer to dynamically allocated array

    int front;      // Index of the front element

    int rear;       // Index of the rear element

    int size;       // Maximum size of the queue

    int count;       // Current number of elements in the queue


public:

    // Constructor to initialize the circular queue

    CircularQueue(int maxSize) {

        size = maxSize;

        queue = new int[size];

        front = 0;

        rear = -1;

        count = 0;

    }


    // Destructor to clean up the allocated memory

    ~CircularQueue() {
```

```
    delete[] queue;
}

// Function to check if the queue is empty
bool isEmpty() {
    return count == 0;
}

// Function to check if the queue is full
bool isFull() {
    return count == size;
}

// Function to add an element to the queue
void enqueue(int value) {
    if (isFull()) {
        cout << "Queue is full! Cannot enqueue " << value << "." << endl;
        return;
    }

    // Increment rear in a circular manner
    rear = (rear + 1) % size;
    queue[rear] = value;
    count++; // Increase the element count
    cout << "Enqueued: " << value << endl;
}

// Function to remove an element from the queue
```

```
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty! Cannot dequeue." << endl;
        return;
    }

    cout << "Dequeued: " << queue[front] << endl;

    // Move front forward in a circular manner
    front = (front + 1) % size;
    count--; // Decrease the element count
}

// Function to get the front element
int getFront() {
    if (isEmpty()) {
        cout << "Queue is empty! No front element." << endl;
        return -1;
    }
    return queue[front];
}

// Function to get the rear element
int getRear() {
    if (isEmpty()) {
        cout << "Queue is empty! No rear element." << endl;
        return -1;
    }
```

```
        return queue[rear];
    }

    // Function to display the elements of the queue
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return;
        }

        cout << "Queue elements: ";
        for (int i = 0; i < count; i++) {
            int index = (front + i) % size; // Calculate the current index
            cout << queue[index] << " ";
        }
        cout << endl;
    }
};

int main() {
    CircularQueue cq(5); // Create a circular queue with size 5

    cq.enqueue(10);
    cq.enqueue(20);
    cq.enqueue(30);
    cq.enqueue(40);
    cq.enqueue(50);
    cq.display();
```

```
    cq.dequeue();
    cq.dequeue();
    cq.display();

    cq.enqueue(60);
    cq.enqueue(70);
    cq.display();

    cout << "Front element: " << cq.getFront() << endl;
    cout << "Rear element: " << cq.getRear() << endl;

    cq.dequeue();
    cq.display();

    return 0;
}
```

Output: 10 10

```
10 20
10 30
10 40
20 40
30 40
30 50
```

Experiment No.: 1

Program Description:

Implementation of Binary Search Tree.

Solution:

```
#include <iostream>
using namespace std;

// Node structure for a Binary Search Tree
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new Node
Node* createNode(int data)
{
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

// Function to insert a node in the BST
Node* insertNode(Node* root, int data)
{
    if (root == nullptr) { // If the tree is empty, return a
```

```
                // new node
        return createNode(data);
    }

    // Otherwise, recur down the tree
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    }
    else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }

    // return the (unchanged) node pointer
    return root;
}

// Function to do inorder traversal of BST
void inorderTraversal(Node* root)
{
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}

// Function to search a given key in a given BST
Node* searchNode(Node* root, int key)
```

```
{
   // Base Cases: root is null or key is present at root
   if (root == nullptr || root->data == key) {
      return root;
   }

   // Key is greater than root's key
   if (root->data < key) {
      return searchNode(root->right, key);
   }

   // Key is smaller than root's key
   return searchNode(root->left, key);
}

// Function to find the inorder successor
Node* minValueNode(Node* node)
{
   Node* current = node;
   // loop down to find the leftmost leaf
   while (current && current->left != nullptr) {
      current = current->left;
   }
   return current;
}

// Function to delete a node
Node* deleteNode(Node* root, int data)
```

```
{
    if (root == nullptr)
        return root;

    // If the data to be deleted is smaller than the root's
    // data, then it lies in the left subtree
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    }
    // If the data to be deleted is greater than the root's
    // data, then it lies in the right subtree
    else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    }
    // if data is same as root's data, then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
```

```
        // node with two children: Get the inorder successor
        // (smallest in the right subtree)
        Node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->data = temp->data;


        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Main function to demonstrate the operations of BST
int main()
{


    Node* root = nullptr;
    // create a BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);
```

```
// Print the inorder traversal of a BST
cout << "Inorder traversal of the given Binary Search "
        "Tree is: ";
inorderTraversal(root);
cout << endl;

// delete a node in BST
root = deleteNode(root, 20);
cout << "After deletion of 20: ";
inorderTraversal(root);
cout << endl;

// Insert a node in BST
root = insertNode(root, 25);
cout << "After insertion of 25: ";
inorderTraversal(root);
cout << endl;

// Search a key in BST
Node* found = searchNode(root, 25);

// check if the key is found or not
if (found != nullptr) {
    cout << "Node 25 found in the BST." << endl;
}
else {
    cout << "Node 25 not found in the BST." << endl;
}
```

```
    return 0;
}
```

Output: Not Found
Found

Experiment No.: 2

Program Description:

Conversion of BST PreOrder/PostOrder/InOrder.

Solution:

a) Preorder

```
#include <bits/stdc++.h>
using namespace std;

// Class describing a node of tree
class Node {
public:
    int data;
    Node* left;
    Node* right;
    Node(int v)
    {
        this->data = v;
        this->left = this->right = NULL;
    }
};

// Preorder Traversal
void printPreOrder(Node* node)
{
    if (node == NULL)
        return;
```

```
    // Visit Node
    cout << node->data << " ";

    // Traverse left subtree
    printPreOrder(node->left);

    // Traverse right subtree
    printPreOrder(node->right);
}

// Driver code
int main()
{
    // Build the tree
    Node* root = new Node(100);
    root->left = new Node(20);
    root->right = new Node(200);
    root->left->left = new Node(10);
    root->left->right = new Node(30);
    root->right->left = new Node(150);
    root->right->right = new Node(300);

    // Function call
    cout << "Preorder Traversal: ";
    printPreOrder(root);
    return 0;
}
```

Output: `Preorder Traversal: 100 20 10 30 200 150 300`

b) Postorder

```
#include <bits/stdc++.h>
using namespace std;

// Class to define structure of a node
class Node {
public:
    int data;
    Node* left;
    Node* right;
    Node(int v)
    {
        this->data = v;
        this->left = this->right = NULL;
    }
};

// PostOrder Traversal
void printPostOrder(Node* node)
{
    if (node == NULL)
        return;

    // Traverse left subtree
    printPostOrder(node->left);

    // Traverse right subtree
    printPostOrder(node->right);
```

```
    // Visit node
    cout << node->data << " ";
}

// Driver code
int main()
{
    Node* root = new Node(100);
    root->left = new Node(20);
    root->right = new Node(200);
    root->left->left = new Node(10);
    root->left->right = new Node(30);
    root->right->left = new Node(150);
    root->right->right = new Node(300);

    // Function call
    cout << "PostOrder Traversal: ";
    printPostOrder(root);
    cout << "\n";

    return 0;
}
```

Output: `PostOrder Traversal: 10 30 20 150 300 200 100`

c) Inorder

```
#include <bits/stdc++.h>
using namespace std;

// Class describing a node of tree
class Node {
public:
    int data;
    Node* left;
    Node* right;
    Node(int v)
    {
        this->data = v;
        this->left = this->right = NULL;
    }
};

// Inorder Traversal
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    // Traverse left subtree
    printInorder(node->left);

    // Visit node
    cout << node->data << " ";
```

```
    // Traverse right subtree
    printInorder(node->right);
}

// Driver code
int main()
{
    // Build the tree
    Node* root = new Node(100);
    root->left = new Node(20);
    root->right = new Node(200);
    root->left->left = new Node(10);
    root->left->right = new Node(30);
    root->right->left = new Node(150);
    root->right->right = new Node(300);

    // Function call
    cout << "Inorder Traversal: ";
    printInorder(root);
    return 0;
}
```

Output:

```
Inorder Traversal: 10 20 30 100 150 200 300
```

Experiment No.: 3

Program Description:

Implementation of Kruskal Algorithm

Solution:

```
#include<bits/stdc++.h>

using namespace std;


// Creating shortcut for an integer pair

typedef pair<int, int> iPair;


// Structure to represent a graph

struct Graph

{

    int V, E;

    vector< pair<int, iPair> > edges;


    // Constructor

    Graph(int V, int E)

    {

        this->V = V;

        this->E = E;

    }


    // Utility function to add an edge

    void addEdge(int u, int v, int w)

    {

        edges.push_back({w, {u, v}});
```

```
    }

    // Function to find MST using Kruskal's
    // MST algorithm
    int kruskalMST();
};

// To represent Disjoint Sets
struct DisjointSets
{
    int *parent, *rnk;
    int n;

    // Constructor.
    DisjointSets(int n)
    {
        // Allocate memory
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];

        // Initially, all vertices are in
        // different sets and have rank 0.
        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;

            //every element is parent of itself
```

```
        parent[i] = i;
    }
}

// Find the parent of a node 'u'
// Path Compression
int find(int u)
{
    /* Make the parent of the nodes in the path
    from u--> parent[u] point to parent[u] */
    if (u != parent[u])
        parent[u] = find(parent[u]);
    return parent[u];
}

// Union by rank
void merge(int x, int y)
{
    x = find(x), y = find(y);

    /* Make tree with smaller height
    a subtree of the other tree */
    if (rnk[x] > rnk[y])
        parent[y] = x;
    else // If rnk[x] <= rnk[y]
        parent[x] = y;

    if (rnk[x] == rnk[y])
```

```
            rnk[y]++;
        }
    };

/* Functions returns weight of the MST*/

int Graph::kruskalMST()
{
    int mst_wt = 0; // Initialize result

    // Sort edges in increasing order on basis of cost
    sort(edges.begin(), edges.end());

    // Create disjoint sets
    DisjointSets ds(V);

    // Iterate through all sorted edges
    vector< pair<int, iPair> >::iterator it;
    for (it=edges.begin(); it!=edges.end(); it++)
    {
        int u = it->second.first;
        int v = it->second.second;

        int set_u = ds.find(u);
        int set_v = ds.find(v);

        // Check if the selected edge is creating
        // a cycle or not (Cycle is created if u
```

```
        // and v belong to same set)
        if (set_u != set_v)
        {
            // Current edge will be in the MST
            // so print it
            cout << u << " - " << v << endl;

            // Update MST weight
            mst_wt += it->first;

            // Merge two sets
            ds.merge(set_u, set_v);
        }
    }

    return mst_wt;
}

// Driver program to test above functions
int main()
{
    /* Let us create above shown weighted
    and undirected graph */
    int V = 9, E = 14;
    Graph g(V, E);

    // making above shown graph
    g.addEdge(0, 1, 4);
```

```
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    cout << "Edges of MST are \n";
    int mst_wt = g.kruskalMST();

    cout << "\nWeight of MST is " << mst_wt;
    return 0;
}
```

Output: Following are the edges in the constructed MST

```
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Experiment No.: 4

Program Description:

Implementation of Prim Algorithm

Solution:

```
#include <bits/stdc++.h>
using namespace std;

// Function to construct and print the MST
void primMST(vector<vector<int>> graph) {
    int v = graph.size();

    // vector to store the parent of vertex
    vector<int> parent(v);

    // vector holds the weight/ cost of the MST
    vector<int> key(v);

    // vector to represent the set of
    // vertices included in MST
    vector<bool> vis(v);

    priority_queue<pair<int, int>,
    vector<pair<int, int>>,
    greater<pair<int, int>>> pq;

    // Initialize all key vector as INFINITE
    // and vis vector as false
```

```
for (int i = 0; i < v; i++) {
    key[i] = INT_MAX;
    vis[i] = false;
}

// Always include the first vertex in MST.
// Make key 0 so that this vertex is
// picked as the first vertex.
key[0] = 0;

// First node is always the root of MST
parent[0] = -1;

// Push the source vertex to the min-heap
pq.push({0, 0});

while (!pq.empty()) {
    int node = pq.top().second;
    pq.pop();
    vis[node] = true;
    for (int i = 0; i < v; i++) {

        // If the vertex is not visited
        // and the edge weight of neighbouring
        // vertex is less than key value of
        // neighbouring vertex then update it.
        if (!vis[i] && graph[node][i] != 0
            && graph[node][i] < key[i]) {
```

```
            pq.push({graph[node][i], i});
            key[i] = graph[node][i];
            parent[i] = node;
          }
        }
    }

    // Print the edges and their
    // weights in the MST
    cout << "Edge \tWeight\n";
    for (int i = 1; i < v; i++) {
        cout << parent[i] << " - " << i
          << " \t" << graph[i][parent[i]] << " \n";
    }
}

int main() {

    // Define the adjacency matrix
  vector<vector<int>> graph = {{0, 2, 0, 6, 0},
                              {2, 0, 3, 8, 5},
                              {0, 3, 0, 0, 7},
                              {6, 8, 0, 0, 9},
                              {0, 5, 7, 9, 0}};

    // Find and print the Minimum Spanning
    // Tree using Prim's algorithm
    primMST(graph);
```

```
    return 0;
}
```

Output:

```
Edge      Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5
```

Experiment No.: 5

Program Description:

Implementation of Dijkstra Algorithm

Solution:

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>


using namespace std;


typedef pair<int, int> pii; // Pair to store (distance, node)


void dijkstra(int start, vector<vector<pii>>& graph, vector<int>& distances) {

    priority_queue<pii, vector<pii>, greater<pii>> pq; // Min-heap priority queue

    pq.push({0, start});

    distances[start] = 0;


    while (!pq.empty()) {
        int currentDistance = pq.top().first;

        int currentNode = pq.top().second;

        pq.pop();


        // Skip if the distance is outdated

        if (currentDistance > distances[currentNode]) continue;


        // Explore neighbors
```

```
        for (auto& neighbor : graph[currentNode]) {
            int neighborNode = neighbor.first;
            int edgeWeight = neighbor.second;

            // Relaxation step
            if (distances[currentNode] + edgeWeight < distances[neighborNode]) {
                distances[neighborNode] = distances[currentNode] + edgeWeight;
                pq.push({distances[neighborNode], neighborNode});
            }
        }
    }
}

int main() {
    int n, m; // Number of nodes and edges
    cout << "Enter the number of nodes and edges: ";
    cin >> n >> m;

    vector<vector<pii>> graph(n + 1); // Adjacency list (1-based indexing)

    cout << "Enter the edges (u v w) where u and v are nodes and w is the weight:\n";
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].push_back({v, w});
        graph[v].push_back({u, w}); // For undirected graph; omit this for directed
    }
```

```
    int start;
    cout << "Enter the start node: ";
    cin >> start;

    vector<int> distances(n + 1, INT_MAX); // Initialize distances to infinity
    dijkstra(start, graph, distances);

    cout << "Shortest distances from node " << start << ":\n";
    for (int i = 1; i <= n; ++i) {
        if (distances[i] == INT_MAX) {
            cout << "Node " << i << ": INF\n";
        } else {
            cout << "Node " << i << ": " << distances[i] << "\n";
        }
    }
    return 0;
}
```

Output:

```
Vertex      Distance from Source
0           0
1           4
2           12
3           19
4           21
5           11
6           9
7           8
8           14
```

Experiment No.: 1

Program Description:

Implementation of Sorting

Solution:

a. Bubble Sort

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
// Traverse through all array elements
for (int i = 0; i < n-1; i++) {
    // Flag to check if any swapping happens in the inner loop
    bool swapped = false;

    // Last i elements are already sorted, so we reduce the range
    for (int j = 0; j < n-i-1; j++) {
        // If the element is greater than the next element, swap them
        if (arr[j] > arr[j+1]) {
            // Swap arr[j] and arr[j+1]
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            swapped = true;
        }
    }

    // If no two elements were swapped by inner loop, then the array is already sorted
    if (!swapped) {
```

```
            break;
        }
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    // Perform Bubble Sort
    bubbleSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output: 1 2 4 5 8

b. Selection Sort

```
#include <iostream>
using namespace std;

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    // Traverse through all array elements
    for (int i = 0; i < n - 1; i++) {
        // Find the minimum element in the unsorted part of the array
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first element of the unsorted part
        if (min_idx != i) {
            int temp = arr[i];
            arr[i] = arr[min_idx];
            arr[min_idx] = temp;
        }
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
```

```
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    // Perform Selection Sort
    selectionSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output: `Sorted array:`
`11 12 22 25 64`

c. Insertion

```
#include <iostream>
#include <vector>

using namespace std;

void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key to one position ahead of their
current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n);
```

```
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    insertionSort(arr);

    cout << "Sorted array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output: 5 6 11 12 13

d. Quick

```
#include <iostream>
using namespace std;

// Function to partition the array into two halves based on the pivot
int partition(int arr[], int low, int high) {
    // Choose the rightmost element as the pivot
    int pivot = arr[high];

    // Pointer for the smaller element
    int i = (low - 1);

    // Traverse the array and rearrange elements based on the pivot
    for (int j = low; j < high; j++) {
        // If current element is smaller than or equal to the pivot, swap it
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    // Place the pivot element at its correct position in the array
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array into two halves and get the pivot index
```

```
        int pi = partition(arr, low, high);

        // Recursively sort the two halves
        quickSort(arr, low, pi - 1);  // Sort elements before the pivot
        quickSort(arr, pi + 1, high); // Sort elements after the pivot
    }
}
// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    // Perform Quick Sort
    quickSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output: `1 5 7 8 9 10`

e. merge sort

```
#include <iostream>
#include <vector>

using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

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

```
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<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 n;
    cout << "Enter the number of elements: ";
    cin >> n;
```

```
    vector<int> arr(n);
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }
    mergeSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output: 5 6 7 11 12 13

Experiment No.: 2

Program Description:

Implementation of Binary Search on a list of numbers stored in an Array

Solution:

```
#include <iostream>

using namespace std;


// Function to perform Binary Search
int binarySearch(int arr[], int size, int target) {
    int left = 0;

    int right = size - 1;


    // Loop until the search space is empty

    while (left <= right) {

        int mid = left + (right - left) / 2; // Find the middle element


        // Check if target is present at mid

        if (arr[mid] == target) {

            return mid; // Target found, return the index

        }


        // If target is greater, ignore the left half

        if (arr[mid] < target) {

            left = mid + 1;

        }

        // If target is smaller, ignore the right half

        else {
```

```
            right = mid - 1;
        }
    }

    // Target not found
    return -1;
}

int main() {
    // Example array (must be sorted for Binary Search)
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int size = sizeof(arr) / sizeof(arr[0]);

    int target;
    cout << "Enter the number to search for: ";
    cin >> target;

    // Perform Binary Search
    int result = binarySearch(arr, size, target);

    if (result != -1) {
        cout << "Element found at index " << result << endl;
    } else {
        cout << "Element not found!" << endl;
    }
    return 0;
}
```

Output: `Element is present at index 3`

Experiment No.: 3

Program Description:

Implementation of Binary Search on a list of strings stored in an Array

Solution:

```
#include <iostream>
#include <string>
using namespace std;

// Function to perform Binary Search on a list of strings
int binarySearch(string arr[], int size, string target) {
  int left = 0;
  int right = size - 1;

  // Loop until the search space is empty
  while (left <= right) {
    int mid = left + (right - left) / 2; // Find the middle index

    // Check if the target is present at mid
    if (arr[mid] == target) {
      return mid; // Target found, return the index
    }

    // If target is greater, ignore the left half
    if (arr[mid] < target) {
      left = mid + 1;
    }
    // If target is smaller, ignore the right half
```

```
        else {
            right = mid - 1;
        }
    }

    // Target not found
    return -1;
}

int main() {
    // Example sorted array of strings
    string arr[] = {"apple", "banana", "cherry", "date", "grape", "kiwi", "mango", "orange", 
"pear", "watermelon"};
    int size = sizeof(arr) / sizeof(arr[0]);


    string target;
    cout << "Enter the string to search for: ";
    cin >> target;

    // Perform Binary Search
    int result = binarySearch(arr, size, target);

    if (result != -1) {
        cout << "Element found at index " << result << endl;
    } else {
        cout << "Element not found!" << endl;
    }
    return 0;
```

```
}
```

Output: `Element found at index 2`

Experiment No.: 4

Program Description:

Implementation of Linear Search on a list of strings stored in an Array

Solution:

```
#include <iostream>

#include <string>

using namespace std;


// Node structure for singly linked list

struct Node {

   string data;

   Node* next;


   Node(string value) {

      data = value;

      next = nullptr;

   }

};


// Function to add a node to the linked list
void append(Node*& head, const string& value) {

   Node* newNode = new Node(value);

   if (!head) {

      head = newNode;

   } else {

      Node* temp = head;

      while (temp->next) {
```

```
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to find the middle node using slow and fast pointers
Node* findMiddle(Node* left, Node* right) {
    if (!left) return nullptr;

    Node* slow = left;
    Node* fast = left;

    while (fast != right && fast->next != right) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}

// Function to perform binary search on the linked list
Node* binarySearch(Node* head, const string& target) {
    if (!head) return nullptr;

    Node* left = head;
    Node* right = nullptr;

    while (left != right) {
```

```
        Node* mid = findMiddle(left, right);
        // Compare mid's data with the target
        if (mid->data == target) {
            return mid; // Target found
        }

        // If target is greater, search in the right half
        if (mid->data < target) {
            left = mid->next;
        }
        // If target is smaller, search in the left half
        else {
            right = mid;
        }
    }

    return nullptr; // Target not found
}

int main() {
    Node* head = nullptr;

    // Adding elements to the linked list
    append(head, "apple");
    append(head, "banana");
    append(head, "cherry");
    append(head, "date");
    append(head, "grape");
```

```
    append(head, "kiwi");
    append(head, "mango");
    append(head, "orange");
    append(head, "pear");
    append(head, "watermelon");

    string target;
    cout << "Enter the string to search for: ";
    cin >> target;

    // Perform binary search on the linked list
    Node* result = binarySearch(head, target);

    if (result != nullptr) {
        cout << "Element found: " << result->data << endl;
    } else {
        cout << "Element not found!" << endl;
    }

    return 0;
}
```

Output: `30 Found at Position: 3`