**SUBMITTED BY**

**NAME: Rithikesh Rathi**

**ROLL NO.: 2021UIC3514**



| **SR.NO.** | **NAME OF EXPERIMENT** | **TEACHER’S SIGNATURE** |
| --- | --- | --- |
|  | Determinant of 2D Matrix |  |
| 2. | Linear Search in 1D array |  |
| 3. | Binary Search In Sorted Array |  |
| 4. | Check if string is palindrome |  |
| 5. | Operations of singly linked list |  |
| 6. | Polynomial Addition using linked list |  |
| 7. | Operations of doubly linked list |  |
| 8. | Operations of stack using queue |  |
| 9. | Convert Infix to Postfix and evaluate |  |
| 10. | Operations of Binary Tree and BST |  |
| 11. | Operations of AVL Tree |  |
| 12. | Sorting Algorithms |  |
| 13. | Graph, DFS, BFT |  |
| 14. | Prim’s and Kruskal’s Algorithm |  |
| 15. | Dijkstra’s Algorithm |  |

**Q.1. Write a program to compute determinant of a 2-d matrix.**

#include <iostream>

using namespace std;

int main () {

    int mat[2][2];

    cout << "Enter the elements of the 2d matrix: " << endl;

    for (int i = 0; i<2; i++) {

        cout << "Row " << i << ": ";

        for (int j = 0; j<2; j++) {

            cin >> mat[i][j];

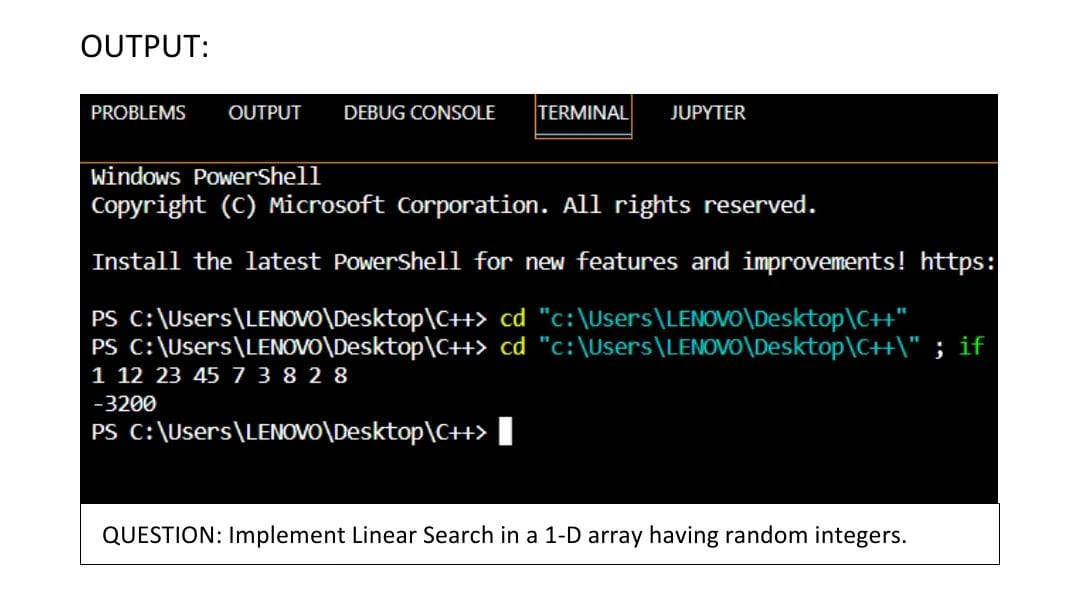
        }

    }

    cout << "The determinant of the 2d matrix is " << (mat[0][0]\*mat[1][1]) - (mat[0][1]\*mat[1][0]) << endl;

    return 0;

}



**Q.2. Implement Linear Search in a 1-D array having random integers.**

#include <iostream>

using namespace std;

int main () {

int n;

cout << "Enter the size of array: ";

cin >> n;

int arr[n];

cout << "Enter the elements of array: ";

for (int i = 0; i<n; i++) {

cin >> arr[i];

}

cout << "Enter the element to be searched: ";

int k;

cin >> k;

bool check = false;

for (int i = 0; i<k; i++) {

if (arr[i] == k) {

check = true;

cout << "The element is present at " << i << " position" << endl;

}

}

if (!check) {

cout << "The element is not present in the array." << endl;

}

return 0;

}



**Q.3. Implement Binary Search in a sorted array.**

#include <iostream>

using namespace std;

int bsearch(int\* arr, int size, int tar) {

int l = 0, h = size-1;

int mid = (l+h)/2;

while (l<=h) {

if (arr[mid] == tar) {

return mid;

}

else if (arr[mid] > tar) {

h = mid - 1;

}

else if (arr[mid] < tar) {

l = mid + 1;

}

mid = (l+h)/2;

}

return -1;

}

int main () {

int n;

cout << "Enter the size of array: ";

cin >> n;

int arr[n];

cout << "Enter the elements of array: ";

for (int i = 0; i<n; i++) {

cin >> arr[i];

}

cout << "Enter the element to be searched: ";

int k;

cin >> k;

int ans = bsearch(arr, n, k);

if (ans == -1) {

cout << "Element is not present in the array";

}

else {

cout << "Element is present at " << ans << " position";

}

return 0;

}



**Q.4. Write a program to check whether a string is palindrome.**

#include <iostream>

#include <string>

using namespace std;

int main () {

string s;

cout << "Enter a string: ";

cin >> s;

int i = 0, j = s.length() - 1;

bool check = true;

while (i <= j) {

if (s[i] != s[j]) {

check = false;

break;

}

i++, j--;

}

if (check) {

cout << "Yes, the string is palindrome.";

}

else {

cout << "No, the string is not palindrome.";

}

}



**Q.5. Implement various operations of a singly linked list.**

#include <iostream>

using namespace std;

class Node {

public:

    int data;

    Node\* next;

    Node(int data) {

        this->data = data;

        next = NULL;

    }

};

Node\* input() {

    int a;

    cin >> a;

    Node\* head = NULL;

    Node\* tail = NULL;

    while (a != -1) {

        Node \*newnode = new Node(a);

        if (head == NULL) {

            head = newnode;

            tail = newnode;

        }

        else {

            tail->next = newnode;

            tail = newnode;

        }

        cin >> a;

    }

    return head;

}

void print(Node\* head) {

    Node\* temp = head;

    while (temp != NULL) {

        cout << temp->data << ' ';

        temp = temp->next;

    }

    cout << endl;

}

Node\* insertion(Node\* head, int k, int pos) {

    if (pos == 0) {

        Node\* newnode = new Node(k);

        newnode->next = head;

        head = newnode;

        return head;

    }

    Node\* temp = head;

    for (int i = 0; i<pos-1 && temp!=NULL; i++) {

        temp = temp->next;

    }

    if (temp != NULL) {

        Node\* newnode = new Node(k);

        newnode->next = temp->next;

        temp->next = newnode;

    }

    return head;

}

Node\* deletion(Node\* head, int pos) {

    if (pos == 0 && head!= NULL) {

        head = head->next;

        return head;

    }

    Node\* temp = head;

    for (int i = 0; i<pos-1 && temp!=NULL; i++) {

        temp = temp->next;

    }

    if (temp != NULL && temp->next !=NULL) {

        temp->next = temp->next->next;

    }

    return head;

}

int main () {

    cout << "Enter the elements of linked list: ";

    Node\* head = input();

    cout << "Enter an element to be inserted, and specify the position: ";

    int e, pos;

    cin >> e >> pos;

    head = insertion(head, e, pos);

cout << "The updated linked list: ";

print(head);

    cout << "Enter the position to be deleted: ";

    int d;

    cin >> d;

    head = deletion(head, d);

    cout << "Printing the linked list: ";

    print(head);

}

**Q.6. Implement polynomial addition using singly linked list.**

#include <iostream>

#include "dsalab.h"

using namespace std;

void printpol(Node\* head) {

Node\* temp = head;

for (int i = 0; temp!=NULL; i++) {

if (i == 0) {

cout << temp->data;

}

else if (i == 1) {

cout << " + " << temp->data << "x";

}

else {

cout << " + " << temp->data << "x^" << i;

}

temp = temp->next;

}

cout << endl;

}

int main () {

cout << "Enter the coeffecients of first polynomial (starting from power of 0): ";

Node\* h1 = input();

cout << "Enter the coeffecients of second polynomial (starting from power of 0): ";

Node\* h2 = input();

cout << "First polynomial is: ";

printpol(h1);

cout << "Second polynomial is: ";

printpol(h2);

Node\* t1 = h1;

Node\* t2 = h2;

while (t1!= NULL & t2!= NULL) {

t1->data = t1->data + t2->data;

t1 = t1->next;

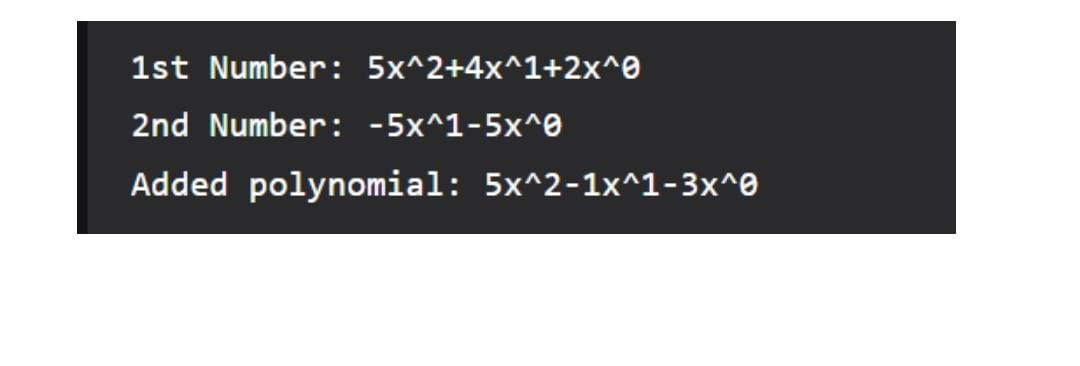
t2 = t2->next;

}

cout << "Final polynomial is: ";

printpol(h1);

}



**Q.7. Implement various operations of a doubly linked list.**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node\* prev;

Node (int data) {

this->data = data;

next = NULL;

prev = NULL;

}

};

Node\* insertion(Node\* head, int a, int pos) {

if (pos == 0) {

Node\* newnode = new Node(a);

newnode->next = head;

head->prev = newnode;

head = newnode;

return head;

}

Node\* temp = head;

for (int i = 0; i<pos-1 && temp!=NULL; i++) {

temp = temp->next;

}

if (temp != NULL) {

Node\* newnode = new Node(a);

newnode->next = temp->next;

temp->next->prev = newnode;

newnode->prev = temp;

temp->next = newnode;

}

return head;

}

Node\* deletion (Node\* head, int pos) {

if (pos == 0) {

head->next->prev = NULL;

head = head->next;

return head;

}

Node\* temp = head;

for (int i = 0; i<pos-1 && temp!=NULL; i++) {

temp = temp->next;

}

if (temp != NULL && temp->next != NULL) {

temp->next->next->prev = temp;

temp->next = temp->next->next;

}

return head;

}

Node\* input () {

int a;

cin >> a;

Node\* head = NULL;

Node\* tail = NULL;

while (a != -1) {

Node\* newnode = new Node(a);

if (head == NULL) {

head = newnode;

tail = newnode;

}

else {

newnode->prev = tail;

tail->next = newnode;

tail = newnode;

}

cin >> a;

}

return head;

}

void print (Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

int main () {

cout << "Enter the desired doubly linked list: ";

Node\* head = input();

cout << "Enter the element to be inserted, and specify the position: ";

int a, k;

cin >> a >> k;

head = insertion(head, a, k);

cout << "Updated doubly linked list is: ";

print(head);

cout << "Enter the position to be deleted: ";

int d;

cin >> d;

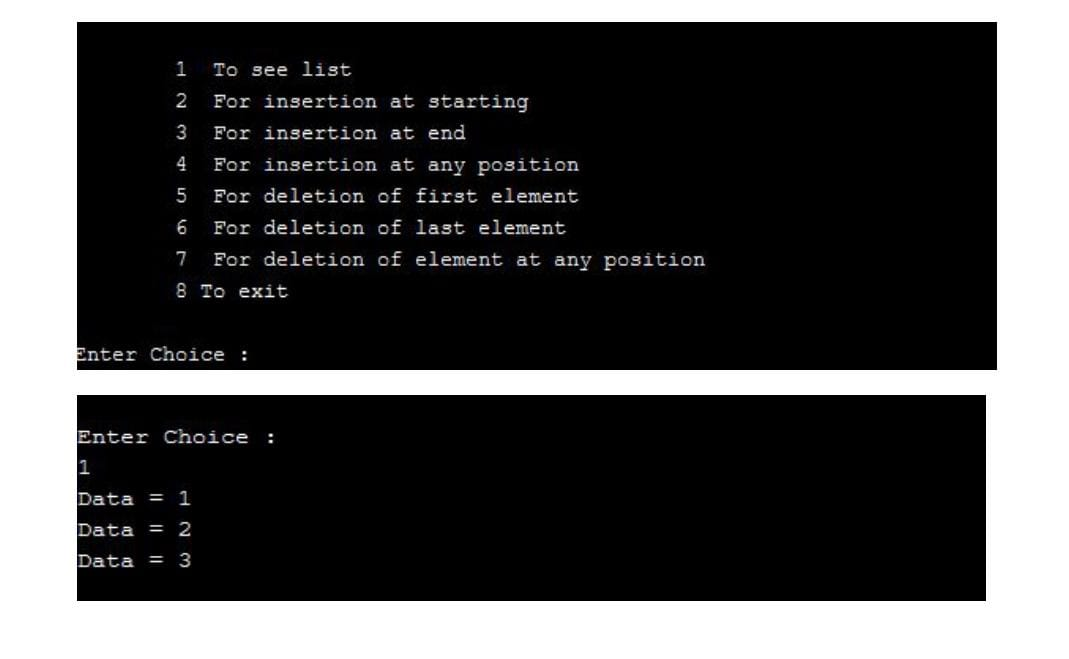
head = deletion(head, d);

cout << "Updated doubly linked list is: ";

print(head);

return 0;

}



**Q. 8. Implement various operations of Stack using queue.**

#include <iostream>

#include <queue>

using namespace std;

class Stack {

    queue<int> q;

public:

    void push(int data) {

        int s = q.size();

        q.push(data);

        for (int i = 0; i < s; i++) {

            q.push(q.front());

            q.pop();

        }

    };

    void pop() {

        if (q.empty())

            cout << "No elements\n";

        else

            q.pop();

    };

    int top() {

        return (q.empty()) ? -1 : q.front();

    };

    int size() {

        return q.size();

    };

    bool empty() {

        return (q.empty());

    };

};

int main() {

    Stack st;

    st.push(1);

    st.push(2);

    st.push(3);

    cout << "Current size: " << st.size() << endl;

    cout << "Top element: " << st.top() << endl;

    st.pop();

    cout << "New top element: " << st.top() << endl;

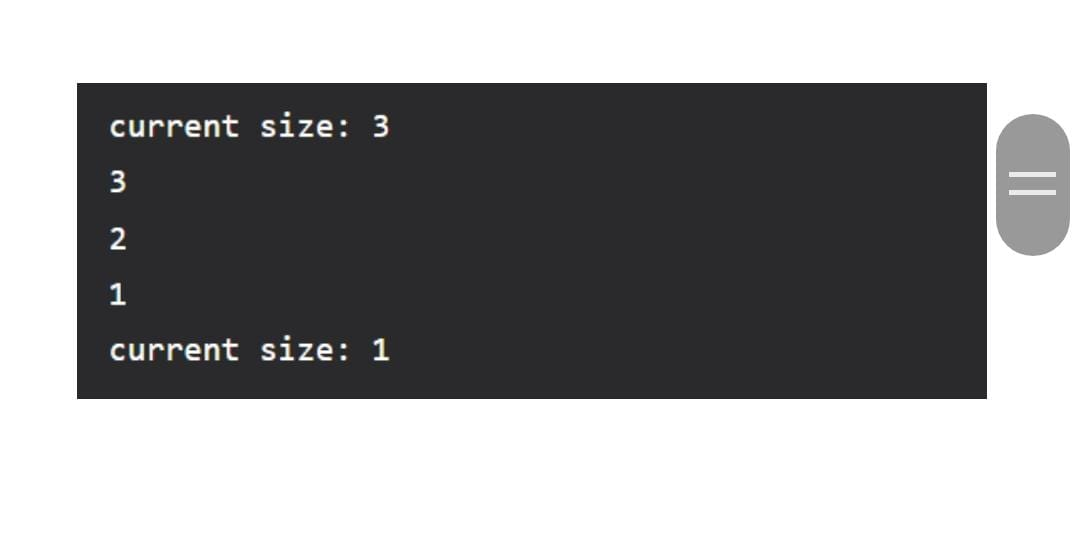
    st.pop();

    cout << "New top element: " << st.top() << endl;

    cout << "Current size: " << st.size();

    return 0;

}



**Q.9. Write program to convert an infix expression to postfix form and evaluate the postfix expression.**

#include <iostream>

#include <stack>

#include<math.h>

using namespace std;

int prec(char c) {

    if (c == '^')   return 3;

    else if (c == '\*' || c == '/')  return 2;

    else if (c == '+' || c == '-')  return 1;

    else    return -1;

}

string inf2pos (string s) {

    stack<char> st;

    string res;

    for (int i = 0; i<s.length(); i++) {

        if ((s[i] >= 'a' && s[i]<='z') || s[i]>='A' && s[i]<='Z') {

            res+=s[i];

        }

        else if (s[i] == '(')   st.push(s[i]);

        else if (s[i] == ')') {

            while (!st.empty() && st.top()!='(') {

                res+=st.top();

                st.pop();

            }

            if(!st.empty()) {

                st.pop();

            }

        }

        else {

            while(!st.empty() && prec(st.top()) > prec(s[i])) {

                res+=st.top();

                st.pop();

            }

            st.push(s[i]);

        }

    }

    while(!st.empty()) {

        res+=st.top();

        st.pop();

    }

    return res;

}

int poseva(string s) {

    stack<int> st;

    for (int i = 0; i<s.length(); i++) {

        if (s[i]>='0' && s[i]<='9') {

            st.push(s[i]-'0');

        }

        else {

            int op2 = st.top();

            st.pop();

            int op1 = st.top();

            st.pop();

            switch(s[i]) {

                case '+':

                    st.push(op1+op2);

                    break;

                case '-':

                    st.push(op1-op2);

                    break;

                case '\*':

                    st.push(op1\*op2);

                    break;

                case '/':

                    st.push(op1/op2);

                    break;

                case '^':

                    st.push(pow(op1,op2));

                    break;

            }

        }

    }

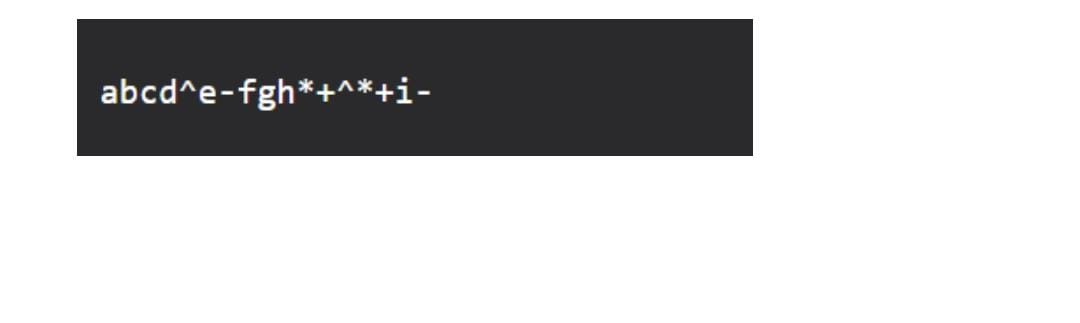
    return st.top();

}

int main () {

    cout << poseva(inf2pos("(a-b/c)\*(a/k-l"));

}



**Q.10. Implement insertion, deletion, traversal and searching operations for a Binary tree and Binary Search Tree.**

#include <iostream>

#include <queue>

using namespace std;

class btnode {

public:

    int data;

    btnode\* left;

    btnode\* right;

    btnode(int data) {

        this->data = data;

        left = NULL;

        right = NULL;

    }

};

btnode\* inputBT () {

    int a;

    cin >> a;

    if (a == -1) {

        return NULL;

    }

    btnode\* root = new btnode(a);

    queue <btnode\*> pending;

    pending.push(root);

    while (!pending.empty()) {

        btnode\* front = pending.front();

        pending.pop();

        cout << "Enter the left child of " << front->data << ": ";

        int lchild;

        cin >> lchild;

        if (lchild != -1) {

            btnode\* child = new btnode(lchild);

            front->left = child;

            pending.push(child);

        }

        cout << "Enter the right child of " << front->data << ": ";

        int rchild;

        cin >> rchild;

        if (rchild != -1) {

            btnode\* child = new btnode(rchild);

            front->left = child;

            pending.push(child);

        }

    }

    return root;

}

void inorder (btnode\* root) {

    if (root == NULL) {

        return;

    }

    inorder(root->left);

    cout << root->data;

    inorder(root->right);

}

void postorder (btnode\* root) {

    if (root == NULL) {

        return;

    }

    postorder(root->left);

    postorder(root->right);

    cout << root->data << ' ';

}

void preorder (btnode\* root) {

    if (root == NULL) {

        return;

    }

    cout << root->data << ' ';

    preorder(root->left);

    preorder(root->right);

}

bool searchbst (btnode\* root, int data) {

    if (root == NULL) {

        return false;

    }

    if (root->data == data) {

        return true;

    }

    if (root->data > data) {

        return searchbst(root->right, data);

    }

    else {

        return searchbst(root->left, data);

    }

}

void deletebst (btnode\* root, int data) {

    if (root == NULL) {

        return;

    }

    if (root->data = data) {

        delete root;

    }

    else if (root->data > data) {

        deletebst(root->right, data);

    }

    else {

        deletebst(root->left, data);

    }

}

int main () {

    cout << "Enter the root of binary tree: ";

    btnode\* root = inputBT();

    int a;

    cout << "Enter the data to be searched: ";

    cin >> a;

    if (searchbst(root,a)) {

        cout << "Yes, the data is present in the BST" << endl;

    }

    else {

        cout << "No, the data is not present in the BST" << endl;

    }

    int d;

    cout << "Enter the data to be deleted: ";

    cin >> d;

    deletebst(root, d);

    postorder(root);

    cout << endl;

    preorder(root);

    cout << endl;

    inorder(root);

    cout << endl;

}

**Q. 11. Write program to perform insertion and deletion operations in an AVL tree.**

#include<bits/stdc++.h>

using namespace std;

// Node created for AVL TREE

class Node

{

    public:

    int key;

    Node \*left;

    Node \*right;

    int height;

};

int height(Node \*N)

{

    if (N == NULL)

        return 0;

    return N->height;

}

Node\* newNode(int key)

{

    Node\* node = new Node();

    node->key = key;

    node->left = NULL;

    node->right = NULL;

    node->height = 1;

    return(node);

}

Node \*rightRotate(Node \*y)

{

    Node \*x = y->left;

    Node \*T2 = x->right;

    x->right = y;

    y->left = T2;

    y->height = max(height(y->left),

                    height(y->right)) + 1;

    x->height = max(height(x->left),

                    height(x->right)) + 1;

    return x;

}

Node \*leftRotate(Node \*x)

{

    Node \*y = x->right;

    Node \*T2 = y->left;

    y->left = x;

    x->right = T2;

    x->height = max(height(x->left),

                    height(x->right)) + 1;

    y->height = max(height(y->left),

                    height(y->right)) + 1;

    return y;

}

int getBalance(Node \*N)

{

    if (N == NULL)

        return 0;

    return height(N->left) - height(N->right);

}

Node\* insert(Node\* node, int key)

{

    if (node == NULL)

        return(newNode(key));

    if (key < node->key)

        node->left = insert(node->left, key);

    else if (key > node->key)

        node->right = insert(node->right, key);

    else // Equal keys are not allowed in BST

        return node;

    node->height = 1 + max(height(node->left),

                        height(node->right));

    int balance = getBalance(node);

    // Left Left Case

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // Right Right Case

    if (balance < -1 && key > node->right->key)

        return leftRotate(node);

    // Left Right Case

    if (balance > 1 && key > node->left->key)

    {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    if (balance < -1 && key < node->right->key)

    {

        node->right = rightRotate(node->right);

        return leftRotate(node);

    }

    return node;

}

void preOrder(Node \*root)

{

    if(root != NULL)

    {

        cout << root->key << " ";

        preOrder(root->left);

        preOrder(root->right);

    }

}

// Driver Code

int main()

{

    Node \*root = NULL;

    root = insert(root, 1);

    root = insert(root, 2);

    root = insert(root, 5);

    root = insert(root, 6);

    root = insert(root, 7);

    root = insert(root, 8);

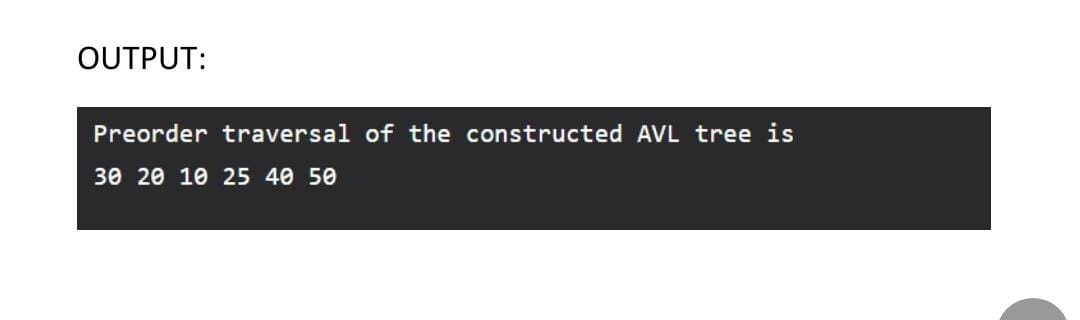
    cout << "Preorder traversal of the "

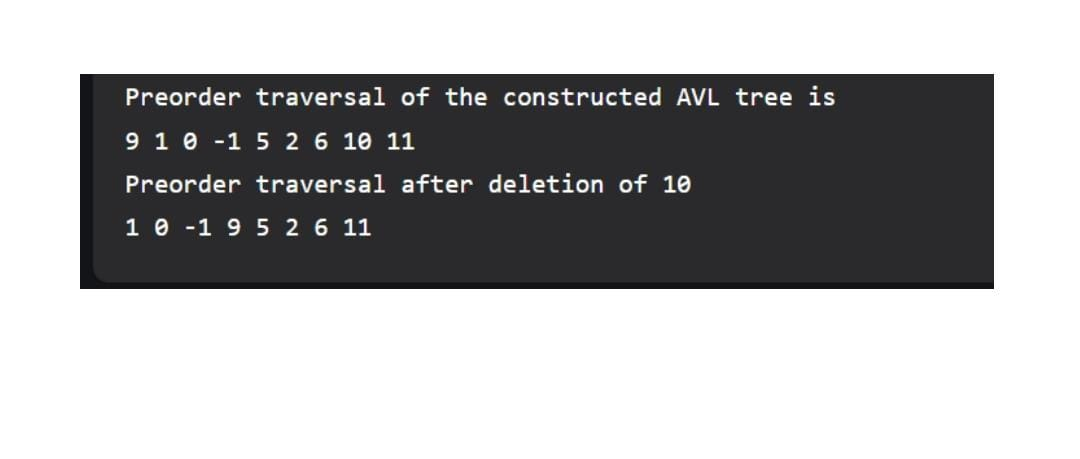
            "constructed AVL tree is \n";

    preOrder(root);

    return 0;

}





**Q. 12. Implement Sorting algorithms – Bubble, Insertion, Selection, Merge Sort, Quick Sort and Heap Sort.**

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int array[], int size) {

for (int step = 0; step < (size-1); ++step) {

int swapped = 0;

for (int i = 0; i < (size-step-1); ++i) {

if (array[i] > array[i + 1]) {

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

swapped = 1;

}

}

if (swapped == 0)

break;

}

}

void insertionSort(int array[], int size) {

for (int step = 1; step < size; step++) {

int key = array[step];

int j = step - 1;

while (key < array[j] && j >= 0) {

array[j + 1] = array[j];

--j;

}

array[j + 1] = key;

}

}

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) {

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

for (int i = n - 1; i >= 0; i--) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

void printArray(int arr[], int n) {

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

cout << "\n";

}

int main () {

int n;

cout << "Enter the size of array: ";

cin >> n;

int arr[n];

cout << "Enter the elements of array: ";

for (int i = 0; i<n; i++) {

cin >> arr[i];

}

cout << "The sorted array is: " << endl;

heapSort(arr,n);

for (int i = 0; i<n; i++) {

cout << arr[i] << ' ';

}

}

**Q. 13. Represent Graph using adjacency matrix and implement depth first traversal, breadth first**

**traversal.**

#include<bits/stdc++.h>

using namespace std;

void addEdge(vector<int> adj[],int u, int v){

adj[u].push\_back(v);

adj[v].push\_back(u);

}

void printGraph(vector<int> adj[],int V)

{

for (int v = 0; v < V; ++v) {

cout << "\n Adjacency list of vertex " << v

<< "\n head ";

for (auto x : adj[v])

cout << "-> " << x;

printf("\n");

}

}

void bfs(int node,vector<int> adj[],vector<int>&ans,vector<bool>&vis){

queue<int>q;

q.push(node);

vis[node]=1;

while(!q.empty()){

int top=q.front();

q.pop();

ans.push\_back(top);

for(auto i:adj[top]){

if(!vis[i]){

vis[i]=true;

q.push(i);

}

}

}

}

// Function to return Breadth First Traversal of given graph.

vector<int> bfsOfGraph(int V, vector<int> adj[]) {

vector<bool>vis(V,0);

vector<int>ans;

bfs(0,adj,ans,vis);

return ans;

}

void dfs(int node, vector<int>&v, map<int,bool>&visited,vector<int> adj[]){

v.push\_back(node);

visited[node]=1;

for(int j=0;j<adj[node].size();j++){

if(!visited[adj[node][j]]){

int n=adj[node][j];

dfs(n,v,visited,adj);

}

}

}

vector<int> dfsOfGraph(int V, vector<int> adj[]) {

vector<int>v;

map<int,bool>visited;

for(int i=0;i<V;i++){

if(!visited[i]){

dfs(i,v,visited,adj);

}

}

return v;

}

int main(){

int V = 5;

vector<int>adj[V];

addEdge (adj, 0, 1);

addEdge (adj, 1, 4);

addEdge (adj, 2, 3);

addEdge (adj, 3, 4);

printGraph (adj, V);

cout<<endl;

vector<int>v1=bfsOfGraph(V,adj);

cout<<"BFS Traversal of Graph is :";

for(int i=0;i<v1.size();i++){

cout<<v1[i]<<" ";

}

cout<<endl;

vector<int>v2=dfsOfGraph(V,adj);

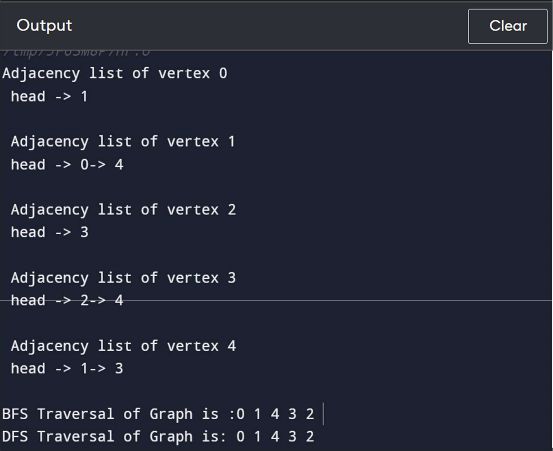
cout<<"DFS Traversal of Graph is: ";

for(int i=0;i<v2.size();i++){

cout<<v2[i]<<" ";

}

}



**Q. 14. Write program to construct Minimum Spanning Tree using Prim’s and Kruskal’s algorithm.**

#include<bits/stdc++.h>

using namespace std;

typedef pair<int, int> iPair;

struct Graph

{

int V, E;

vector< pair<int, iPair> > edges;

Graph(int V, int E)

{

this->V = V;

this->E = E;

}

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();

};

struct DisjointSets

{

int \*parent, \*rnk;

int n;

DisjointSets(int n)

{

this->n = n;

parent = new int[n+1];

rnk = new int[n+1];

for (int i = 0; i <= n; i++)

{

rnk[i] = 0;

//every element is parent of itself

parent[i] = i;

}

}

int find(int 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);

if (rnk[x] > rnk[y])

parent[y] = x;

else // If rnk[x] <= rnk[y]

parent[x] = y;

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

rnk[y]++;

}

};

int Graph::kruskalMST()

{

int mst\_wt = 0; // Initialize result

sort(edges.begin(), edges.end());

DisjointSets ds(V);

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);

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;

}

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;

}

#include <bits/stdc++.h>

using namespace std;

#define V 5

int minKey(int key[], bool mstSet[])

{

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

void printMST(int parent[], int graph[V][V])

{

cout << "Edge \tWeight\n";

for (int i = 1; i < V; i++)

cout << parent[i] << " - " << i << " \t"

<< graph[i][parent[i]] << " \n";

}

void primMST(int graph[V][V])

{

// Array to store constructed MST

int parent[V];

int key[V];

bool mstSet[V];

for (int i = 0; i < V; i++)

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

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

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++)

if (graph[u][v] && mstSet[v] == false

&& graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

int main()

{

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

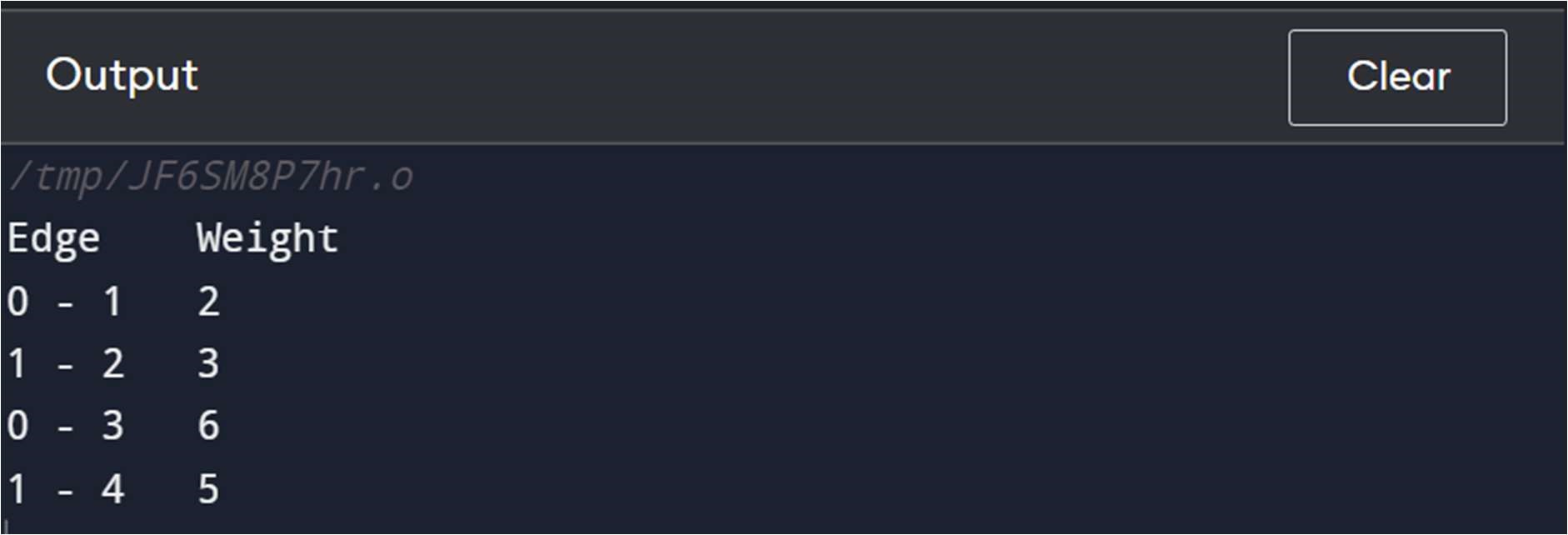
{ 0, 5, 7, 9, 0 } };

primMST(graph);

return 0;

}

OUTPUT:



**Q.15. Implement Dijkstra’s Shortest Path Algorithm.**

#include <iostream>

#include <vector>

#define INT\_MAX 10000000

using namespace std;

void DijkstrasTest();

int main() {

  DijkstrasTest();

  return 0;

}

class Node;

class Edge;

void Dijkstras();

vector<Node\*>\* AdjacentRemainingNodes(Node\* node);

Node\* ExtractSmallest(vector<Node\*>& nodes);

int Distance(Node\* node1, Node\* node2);

bool Contains(vector<Node\*>& nodes, Node\* node);

void PrintShortestRouteTo(Node\* destination);

vector<Node\*> nodes;

vector<Edge\*> edges;

class Node {

   public:

  Node(char id)

    : id(id), previous(NULL), distanceFromStart(INT\_MAX) {

    nodes.push\_back(this);

  }

   public:

  char id;

  Node\* previous;

  int distanceFromStart;

};

class Edge {

   public:

  Edge(Node\* node1, Node\* node2, int distance)

    : node1(node1), node2(node2), distance(distance) {

    edges.push\_back(this);

  }

  bool Connects(Node\* node1, Node\* node2) {

    return (

      (node1 == this->node1 &&

       node2 == this->node2) ||

      (node1 == this->node2 &&

       node2 == this->node1));

  }

   public:

  Node\* node1;

  Node\* node2;

  int distance;

};

void DijkstrasTest() {

  Node\* a = new Node('a');

  Node\* b = new Node('b');

  Node\* c = new Node('c');

  Node\* d = new Node('d');

  Node\* e = new Node('e');

  Node\* f = new Node('f');

  Node\* g = new Node('g');

  Edge\* e1 = new Edge(a, c, 1);

  Edge\* e2 = new Edge(a, d, 2);

  Edge\* e3 = new Edge(b, c, 2);

  Edge\* e4 = new Edge(c, d, 1);

  Edge\* e5 = new Edge(b, f, 3);

  Edge\* e6 = new Edge(c, e, 3);

  Edge\* e7 = new Edge(e, f, 2);

  Edge\* e8 = new Edge(d, g, 1);

  Edge\* e9 = new Edge(g, f, 1);

  a->distanceFromStart = 0;  // set start node

  Dijkstras();

  PrintShortestRouteTo(f);

}

void Dijkstras() {

  while (nodes.size() > 0) {

    Node\* smallest = ExtractSmallest(nodes);

    vector<Node\*>\* adjacentNodes =

      AdjacentRemainingNodes(smallest);

    const int size = adjacentNodes->size();

    for (int i = 0; i < size; ++i) {

      Node\* adjacent = adjacentNodes->at(i);

      int distance = Distance(smallest, adjacent) +

               smallest->distanceFromStart;

      if (distance < adjacent->distanceFromStart) {

        adjacent->distanceFromStart = distance;

        adjacent->previous = smallest;

      }

    }

    delete adjacentNodes;

  }

}

Node\* ExtractSmallest(vector<Node\*>& nodes) {

  int size = nodes.size();

  if (size == 0) return NULL;

  int smallestPosition = 0;

  Node\* smallest = nodes.at(0);

  for (int i = 1; i < size; ++i) {

    Node\* current = nodes.at(i);

    if (current->distanceFromStart <

      smallest->distanceFromStart) {

      smallest = current;

      smallestPosition = i;

    }

  }

  nodes.erase(nodes.begin() + smallestPosition);

  return smallest;

}

vector<Node\*>\* AdjacentRemainingNodes(Node\* node) {

  vector<Node\*>\* adjacentNodes = new vector<Node\*>();

  const int size = edges.size();

  for (int i = 0; i < size; ++i) {

    Edge\* edge = edges.at(i);

    Node\* adjacent = NULL;

    if (edge->node1 == node) {

      adjacent = edge->node2;

    } else if (edge->node2 == node) {

      adjacent = edge->node1;

    }

    if (adjacent && Contains(nodes, adjacent)) {

      adjacentNodes->push\_back(adjacent);

    }

  }

  return adjacentNodes;

}

int Distance(Node\* node1, Node\* node2) {

  const int size = edges.size();

  for (int i = 0; i < size; ++i) {

    Edge\* edge = edges.at(i);

    if (edge->Connects(node1, node2)) {

      return edge->distance;

    }

  }

  return -1;  // should never happen

}

bool Contains(vector<Node\*>& nodes, Node\* node) {

  const int size = nodes.size();

  for (int i = 0; i < size; ++i) {

    if (node == nodes.at(i)) {

      return true;

    }

  }

  return false;

}

void PrintShortestRouteTo(Node\* destination) {

  Node\* previous = destination;

  cout << "Distance from start: "

     << destination->distanceFromStart << endl;

  while (previous) {

    cout << previous->id << " ";

    previous = previous->previous;

  }

  cout << endl;

}

vector<Edge\*>\* AdjacentEdges(vector<Edge\*>& Edges, Node\* node);

void RemoveEdge(vector<Edge\*>& Edges, Edge\* edge);

vector<Edge\*>\* AdjacentEdges(vector<Edge\*>& edges, Node\* node) {

  vector<Edge\*>\* adjacentEdges = new vector<Edge\*>();

  const int size = edges.size();

  for (int i = 0; i < size; ++i) {

    Edge\* edge = edges.at(i);

    if (edge->node1 == node) {

      cout << "adjacent: " << edge->node2->id << endl;

      adjacentEdges->push\_back(edge);

    } else if (edge->node2 == node) {

      cout << "adjacent: " << edge->node1->id << endl;

      adjacentEdges->push\_back(edge);

    }

  }

  return adjacentEdges;

}

void RemoveEdge(vector<Edge\*>& edges, Edge\* edge) {

  vector<Edge\*>::iterator it;

  for (it = edges.begin(); it < edges.end(); ++it) {

    if (\*it == edge) {

      edges.erase(it);

      return;

    }

  }

}

