**Name : Anjali**

**UID : 22BCS10665  
Batch : 22BCS\_IOT-614(B)**

**Assignment – 6**

**Problem-1 : Implement Queue using Stack**  
**Code:**

class MyQueue {

private:

stack<int> s1, s2;

void transfer() {

while (!s1.empty()) {

s2.push(s1.top());

s1.pop();

}

}

public:

MyQueue() {}

void push(int x) {

s1.push(x);

}

int pop() {

if (s2.empty()) {

transfer();

}

int front = s2.top();

s2.pop();

return front;

}

int peek() {

if (s2.empty()) {

transfer();

}

return s2.top();

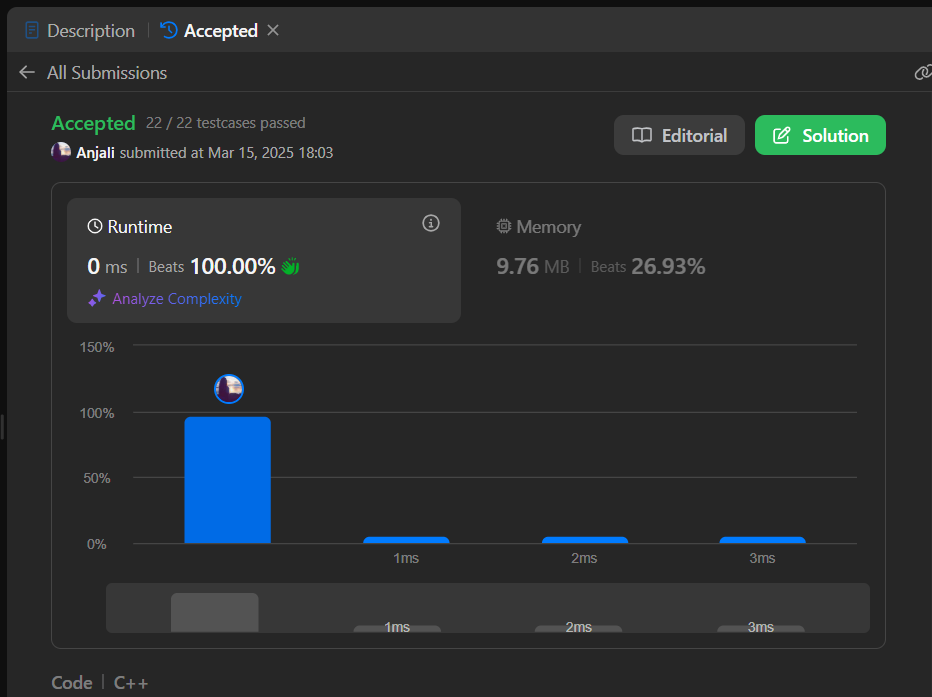
}

bool empty() {

return s1.empty() && s2.empty();

}

};

**Output SS:**

**Problem-2 : Implement Stack using Queue**  
**Code:**

class MyStack {

private:

queue<int> q1, q2;

public:

MyStack() {}

void push(int x) {

q2.push(x);

while (!q1.empty()) {

q2.push(q1.front());

q1.pop();

}

swap(q1, q2);

}

int pop() {

int top = q1.front();

q1.pop();

return top;

}

int top() {

return q1.front();

}

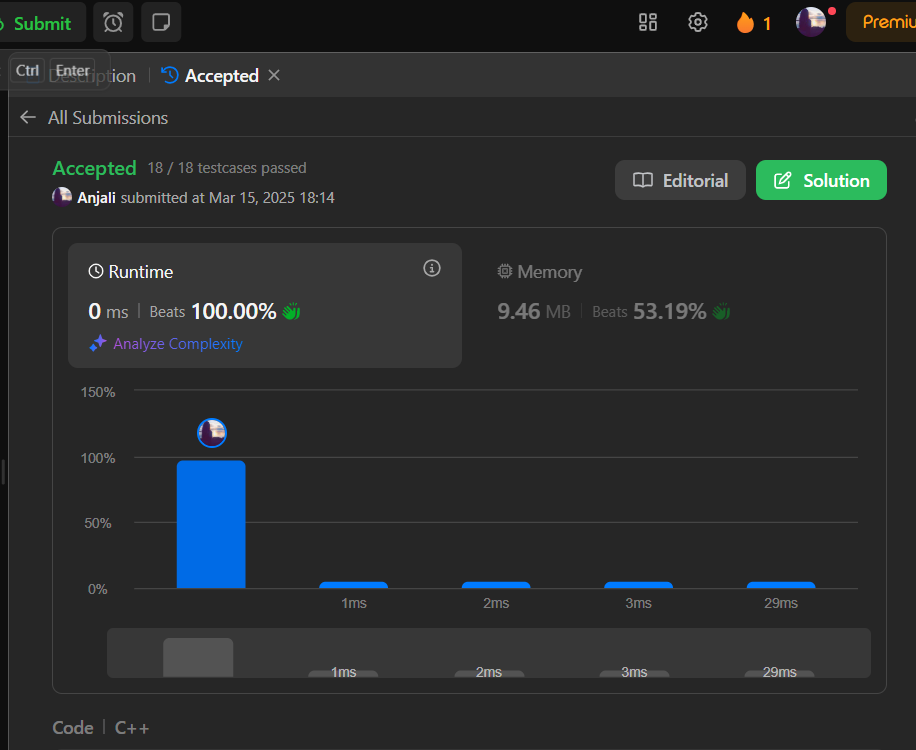
bool empty() {

return q1.empty();

}

};

**Output SS:**



**Problem-3 : Implement Trie using an Array**  
**Code:**

class Trie {

private:

    struct TrieNode {

        unordered\_map<char, TrieNode\*> children;

        bool isEndOfWord = false;

    };

    TrieNode\* root;

public:

    Trie() {

        root = new TrieNode();

    }

    void insert(string word) {

        TrieNode\* node = root;

        for (char c : word) {

            if (!node->children.count(c)) {

                node->children[c] = new TrieNode();

            }

            node = node->children[c];

        }

        node->isEndOfWord = true;

    }

    bool search(string word) {

        TrieNode\* node = root;

        for (char c : word) {

            if (!node->children.count(c)) {

                return false;

            }

            node = node->children[c];

        }

        return node->isEndOfWord;

    }

    bool startsWith(string prefix) {

        TrieNode\* node = root;

        for (char c : prefix) {

            if (!node->children.count(c)) {

                return false;

            }

            node = node->children[c];

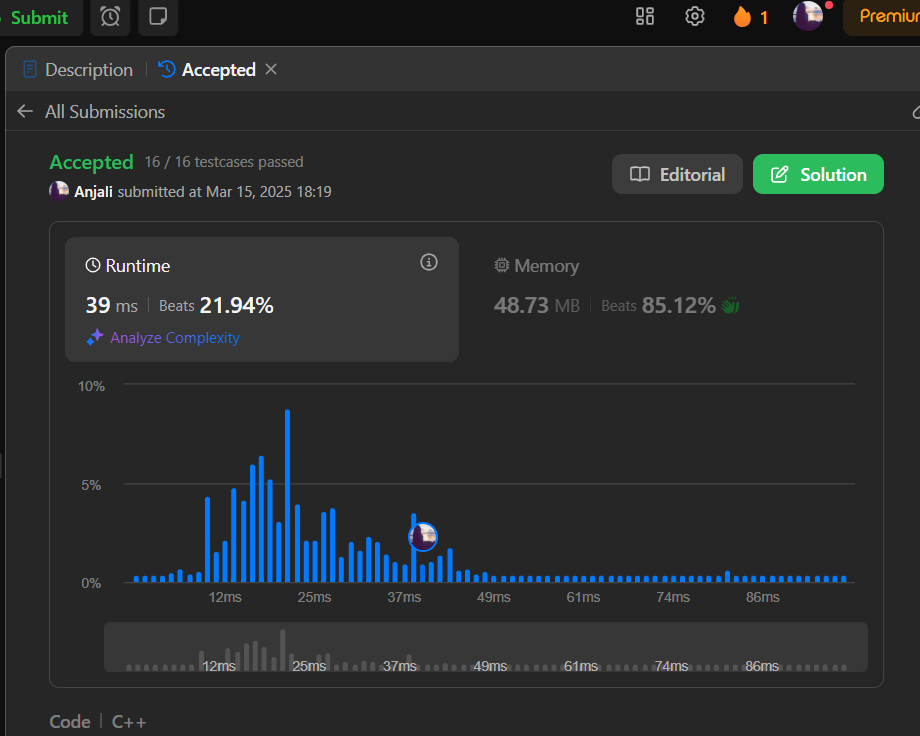
        }

        return true;

    }

};

**Output SS:**



**Problem-4 : Implement Deque using Stack**  
**Code:**

#include <iostream>

#include <stack>

using namespace std;

class DequeUsingStacks {

private:

stack<int> frontStack, backStack;

void transfer(stack<int>& from, stack<int>& to) {

while (!from.empty()) {

to.push(from.top());

from.pop();

}

}

public:

void pushFront(int x) {

frontStack.push(x);

}

void pushBack(int x) {

backStack.push(x);

}

int popFront() {

if (frontStack.empty()) {

transfer(backStack, frontStack);

}

if (frontStack.empty()) throw runtime\_error("Deque is empty");

int val = frontStack.top();

frontStack.pop();

return val;

}

int popBack() {

if (backStack.empty()) {

transfer(frontStack, backStack);

}

if (backStack.empty()) throw runtime\_error("Deque is empty");

int val = backStack.top();

backStack.pop();

return val;

}

bool empty() {

return frontStack.empty() && backStack.empty();

}

};

int main() {

DequeUsingStacks dq;

dq.pushFront(1);

dq.pushBack(2);

dq.pushFront(3);

cout << dq.popFront() << endl;

cout << dq.popBack() << endl;

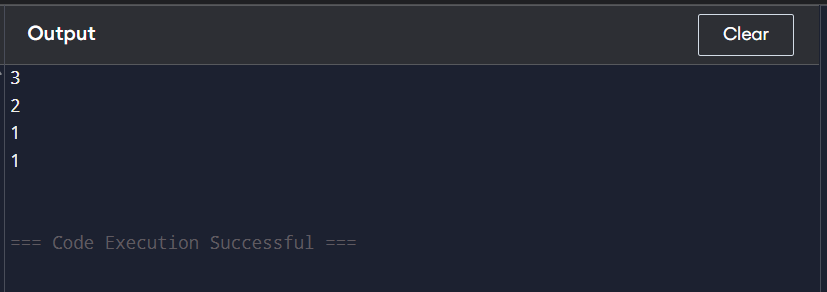
cout << dq.popFront() << endl;

cout << dq.empty() << endl;

return 0;

}

**Output SS:**



**Problem-5 : Implement Min Stack using Two Stacks**  
**Code:**

#include <iostream>

#include <stack>

using namespace std;

class MinStack {

private:

stack<int> mainStack;

stack<int> minStack;

public:

void push(int x) {

mainStack.push(x);

if (minStack.empty() || x <= minStack.top()) {

minStack.push(x);

}

}

void pop() {

if (!mainStack.empty()) {

if (mainStack.top() == minStack.top()) {

minStack.pop();

}

mainStack.pop();

}

}

int top() {

return mainStack.top();

}

int getMin() {

return minStack.top();

}

};

int main() {

MinStack minStack;

minStack.push(5);

minStack.push(2);

minStack.push(8);

cout << "Minimum: " << minStack.getMin() << endl;

minStack.pop();

cout << "Top: " << minStack.top() << endl;

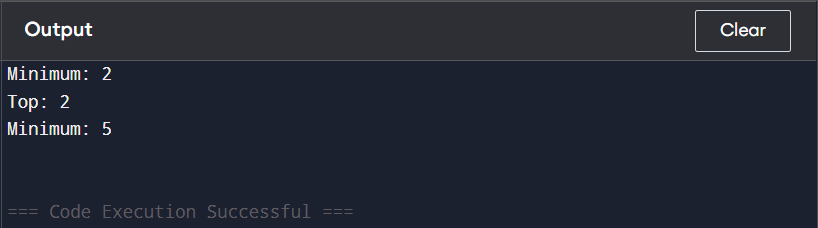
minStack.pop();

cout << "Minimum: " << minStack.getMin() << endl;

return 0;

}

**Output SS:**



**Problem-6 : Implement Max Stack using Two Stacks**  
**Code:**

#include <iostream>

#include <stack>

using namespace std;

class MaxStack {

private:

stack<int> mainStack;

stack<int> maxStack;

public:

void push(int x) {

mainStack.push(x);

if (maxStack.empty() || x >= maxStack.top()) {

maxStack.push(x);

}

}

int pop() {

if (mainStack.empty()) return -1;

int top = mainStack.top();

mainStack.pop();

if (top == maxStack.top()) {

maxStack.pop();

}

return top;

}

int top() {

return mainStack.empty() ? -1 : mainStack.top();

}

int getMax() {

return maxStack.empty() ? -1 : maxStack.top();

}

bool empty() {

return mainStack.empty();

}

};

int main() {

MaxStack s;

s.push(5);

s.push(1);

s.push(10);

cout << "Max: " << s.getMax() << endl;

cout << "Pop: " << s.pop() << endl;

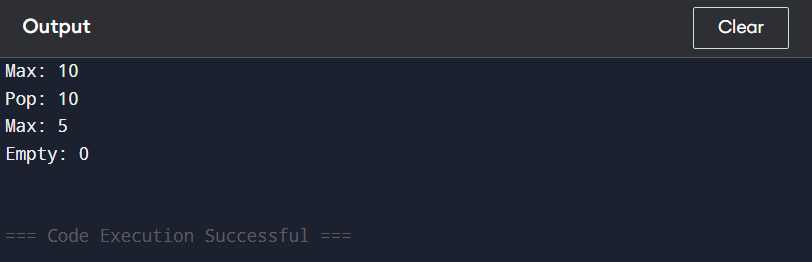
cout << "Max: " << s.getMax() << endl;

cout << "Empty: " << s.empty() << endl;

return 0;

}

**Output SS:**



**Problem-7 : Implement Priority Queue using Stack**  
**Code:**

#include <bits/stdc++.h>

using namespace std;

class PriorityQueueStack {

private:

stack<pair<int, int>> s;

int order = 0;

public:

void push(int x) {

s.push({x, order++});

}

int pop() {

if (s.empty()) return -1;

stack<pair<int, int>> temp;

pair<int, int> maxElem = s.top();

s.pop();

while (!s.empty()) {

if (s.top().first > maxElem.first || (s.top().first == maxElem.first && s.top().second < maxElem.second)) {

temp.push(maxElem);

maxElem = s.top();

} else {

temp.push(s.top());

}

s.pop();

}

while (!temp.empty()) {

s.push(temp.top());

temp.pop();

}

return maxElem.first;

}

bool empty() {

return s.empty();

}

};

int main() {

PriorityQueueStack pq;

pq.push(3);

pq.push(1);

pq.push(4);

pq.push(2);

cout << pq.pop() << endl;

cout << pq.pop() << endl;

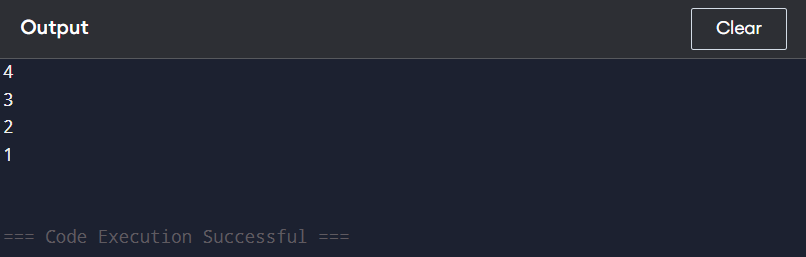
cout << pq.pop() << endl;

cout << pq.pop() << endl;

return 0;

}

**Output SS:**



**Problem-8 : Implement BST (Inorder Traversal) using Stack (Iterative DFS)**  
**Code:**

#include <iostream>

#include <stack>

using namespace std;

struct TreeNode {

int val;

TreeNode \*left, \*right;

TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

void inorderTraversal(TreeNode\* root) {

stack<TreeNode\*> s;

TreeNode\* curr = root;

while (curr != NULL || !s.empty()) {

while (curr != NULL) {

s.push(curr);

curr = curr->left;

}

curr = s.top();

s.pop();

cout << curr->val << " ";

curr = curr->right;

}

}

int main() {

TreeNode\* root = new TreeNode(5);

root->left = new TreeNode(3);

root->right = new TreeNode(7);

root->left->left = new TreeNode(2);

root->left->right = new TreeNode(4);

root->right->left = new TreeNode(6);

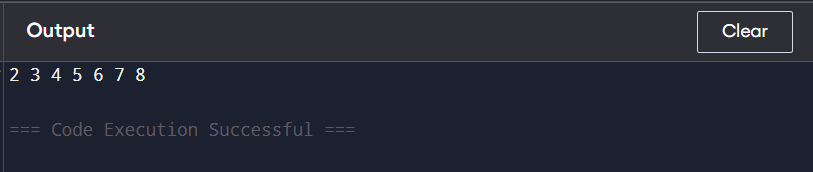
root->right->right = new TreeNode(8);

inorderTraversal(root);

return 0;

}

**Output SS:**



**Problem-9 : Implement Graph DFS using Stack (Iterative DFS)**  
**Code:**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int v, int w) {

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

}

void DFS(int start) {

vector<bool> visited(V, false);

stack<int> s;

s.push(start);

while (!s.empty()) {

int v = s.top();

s.pop();

if (!visited[v]) {

cout << v << " ";

visited[v] = true;

}

for (auto it = adj[v].rbegin(); it != adj[v].rend(); ++it) {

if (!visited[\*it]) {

s.push(\*it);

}

}

}

}

};

int main() {

Graph g(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

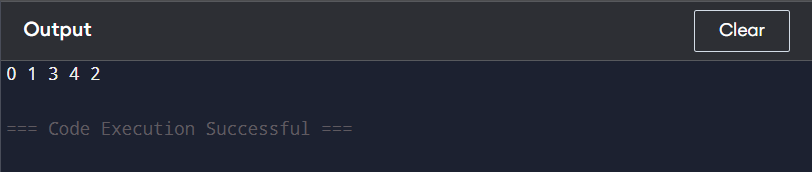
g.addEdge(1, 4);

g.DFS(0);

return 0;

}

**Output SS:**



**Problem-10 : Implement Stack using Queue**

**Code:**

#include <iostream>

#include <queue>

using namespace std;

class MyStack {

private:

queue<int> q;

public:

void push(int x) {

q.push(x);

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

q.push(q.front());

q.pop();

}

}

int pop() {

int top = q.front();

q.pop();

return top;

}

int top() {

return q.front();

}

bool empty() {

return q.empty();

}

};

int main() {

MyStack s;

s.push(1);

s.push(2);

cout << s.top() << endl;

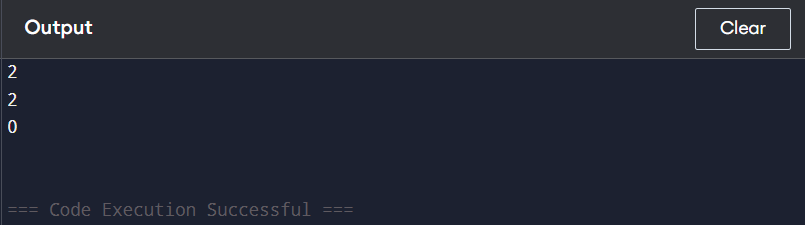
cout << s.pop() << endl;

cout << s.empty() << endl;

return 0;

}

**Output SS:**



**Problem-11 : Implement Deque using Queue**  
**Code:**

#include <iostream>

#include <queue>

using namespace std;

class MyDeque {

private:

queue<int> q1, q2;

public:

void pushFront(int x) {

q2.push(x);

while (!q1.empty()) {

q2.push(q1.front());

q1.pop();

}

swap(q1, q2);

}

void pushBack(int x) {

q1.push(x);

}

int popFront() {

if (q1.empty()) return -1;

int front = q1.front();

q1.pop();

return front;

}

int popBack() {

if (q1.empty()) return -1;

while (q1.size() > 1) {

q2.push(q1.front());

q1.pop();

}

int back = q1.front();

q1.pop();

swap(q1, q2);

return back;

}

bool empty() {

return q1.empty();

}

};

int main() {

MyDeque dq;

dq.pushFront(1);

dq.pushBack(2);

dq.pushFront(3);

cout << dq.popFront() << endl;

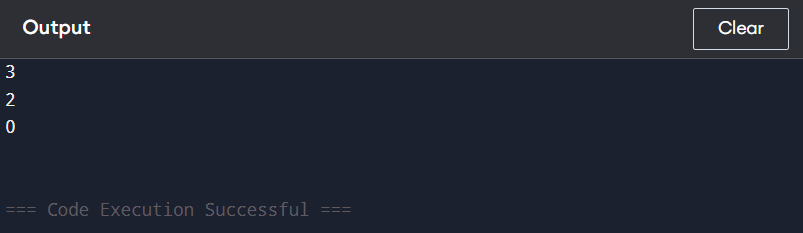
cout << dq.popBack() << endl;

cout << dq.empty() << endl;

return 0;

}

**Output SS:**



**Problem-12 : Implement Circular Queue using Queue**  
**Code:**

#include <iostream>

#include <queue>

using namespace std;

class CircularQueue {

private:

queue<int> q;

int maxSize;

public:

CircularQueue(int k) {

maxSize = k;

}

bool enQueue(int value) {

if (q.size() == maxSize) return false;

q.push(value);

return true;

}

bool deQueue() {

if (q.empty()) return false;

q.pop();

return true;

}

int Front() {

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

}

int Rear() {

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

}

bool isEmpty() {

return q.empty();

}

bool isFull() {

return q.size() == maxSize;

}

};

int main() {

CircularQueue cq(3);

cout << cq.enQueue(1) << endl;

cout << cq.enQueue(2) << endl;

cout << cq.enQueue(3) << endl;

cout << cq.enQueue(4) << endl;

cout << cq.Rear() << endl;

cout << cq.isFull() << endl;

cout << cq.deQueue() << endl;

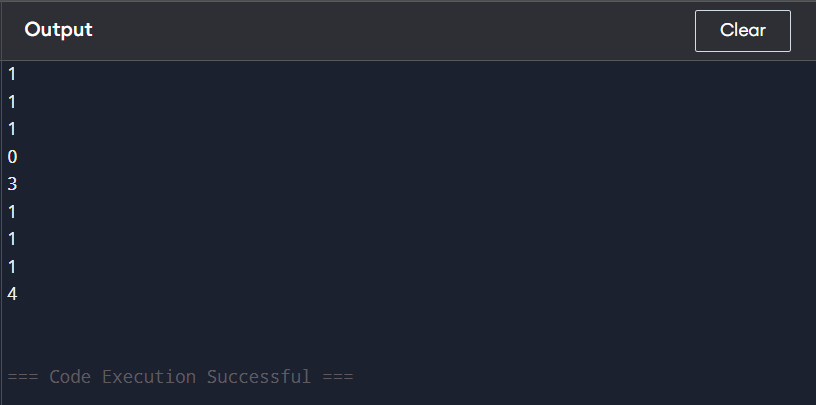
cout << cq.enQueue(4) << endl;

cout << cq.Rear() << endl;

return 0;

}

**Output SS:**



**Problem-13 :  Implement BST Level Order Traversal using Queue (BFS)**  
**Code:**

#include <iostream>

#include <queue>

using namespace std;

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

void levelOrder(TreeNode\* root) {

if (!root) return;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* node = q.front();

q.pop();

cout << node->val << " ";

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

}

int main() {

TreeNode\* root = new TreeNode(10);

root->left = new TreeNode(5);

root->right = new TreeNode(15);

root->left->left = new TreeNode(2);

root->left->right = new TreeNode(7);

root->right->left = new TreeNode(12);

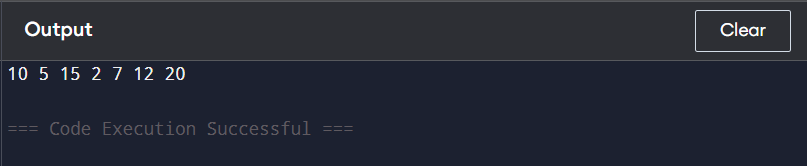
root->right->right = new TreeNode(20);

levelOrder(root);

return 0;

}

**Output SS:**



**Problem-14 : Implement Graph BFS using Queue**  
**Code:**

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int u, int v) {

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

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

}

void BFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : adj[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

};

int main() {

Graph g(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

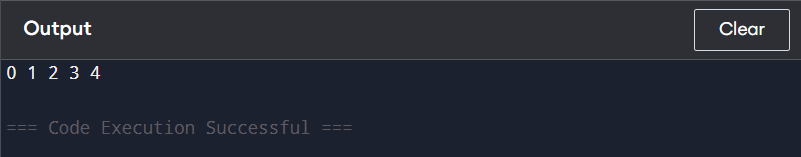
g.addEdge(1, 4);

g.BFS(0);

return 0;

}

**Output SS:**



**Problem-15 : Implement Circular Queue using an Array**  
**Code:**

#include <iostream>

using namespace std;

class CircularQueue {

private:

int \*arr;

int front, rear, size, capacity;

public:

CircularQueue(int c) {

capacity = c;

arr = new int[capacity];

front = rear = -1;

size = 0;

}

bool isFull() {

return size == capacity;

}

bool isEmpty() {

return size == 0;

}

void enqueue(int value) {

if (isFull()) return;

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

rear = (rear + 1) % capacity;

arr[rear] = value;

size++;

}

int dequeue() {

if (isEmpty()) return -1;

int data = arr[front];

front = (front + 1) % capacity;

size--;

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

return data;

}

int frontElement() {

return isEmpty() ? -1 : arr[front];

}

int rearElement() {

return isEmpty() ? -1 : arr[rear];

}

};

int main() {

CircularQueue q(5);

q.enqueue(1);

q.enqueue(2);

q.enqueue(3);

cout << q.dequeue() << endl;

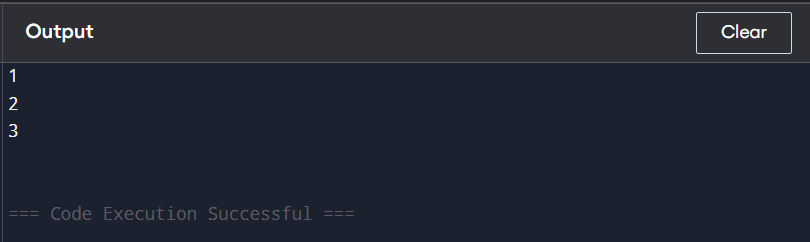
cout << q.frontElement() << endl;

cout << q.rearElement() << endl;

return 0;

}

**Output SS:**



**Problem-16 : Implement Deque using an Array**  
**Code:**

#include <iostream>

using namespace std;

class Deque {

private:

int\* arr;

int front, rear, size, capacity;

public:

Deque(int cap) {

capacity = cap;

arr = new int[cap];

front = -1;

rear = 0;

size = 0;

}

bool isFull() {

return size == capacity;

}

bool isEmpty() {

return size == 0;

}

void insertFront(int value) {

if (isFull()) return;

if (front == -1) {

front = rear = 0;

} else {

front = (front - 1 + capacity) % capacity;

}

arr[front] = value;

size++;

}

void insertRear(int value) {

if (isFull()) return;

if (front == -1) {

front = rear = 0;

} else {

rear = (rear + 1) % capacity;

}

arr[rear] = value;

size++;

}

void deleteFront() {

if (isEmpty()) return;

if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % capacity;

}

size--;

}

void deleteRear() {

if (isEmpty()) return;

if (front == rear) {

front = rear = -1;

} else {

rear = (rear - 1 + capacity) % capacity;

}

size--;

}

int getFront() {

return isEmpty() ? -1 : arr[front];

}

int getRear() {

return isEmpty() ? -1 : arr[rear];

}

};

int main() {

Deque dq(5);

dq.insertRear(10);

dq.insertFront(20);

dq.insertRear(30);

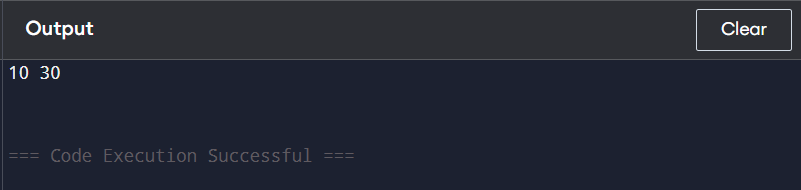
dq.deleteFront();

cout << dq.getFront() << " " << dq.getRear() << endl;

return 0;

}

**Output SS:**



**Problem-17 : Implement Two Stacks in One Array**  
**Code:**

#include <iostream>

using namespace std;

class TwoStacks {

int\* arr;

int size;

int top1, top2;

public:

TwoStacks(int n) {

size = n;

arr = new int[n];

top1 = -1;

top2 = n;

}

void push1(int x) {

if (top1 < top2 - 1) {

arr[++top1] = x;

}

}

void push2(int x) {

if (top1 < top2 - 1) {

arr[--top2] = x;

}

}

int pop1() {

if (top1 >= 0) {

return arr[top1--];

}

return -1;

}

int pop2() {

if (top2 < size) {

return arr[top2++];

}

return -1;

}

};

int main() {

TwoStacks ts(10);

ts.push1(5);

ts.push2(10);

ts.push1(15);

ts.push2(20);

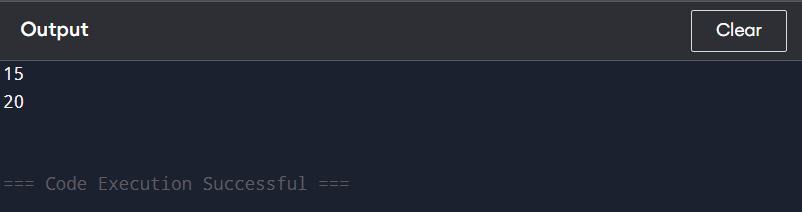
cout << ts.pop1() << endl;

cout << ts.pop2() << endl;

return 0;

}

**Output SS:**



**Problem-18 : Implement k Stacks in a Single Array**  
**Code:**

#include <iostream>

using namespace std;

class KStacks {

int \*arr, \*top, \*next;

int n, k, freeTop;

public:

KStacks(int k, int n) {

this->k = k;

this->n = n;

arr = new int[n];

top = new int[k];

next = new int[n];

freeTop = 0;

for (int i = 0; i < k; i++) top[i] = -1;

for (int i = 0; i < n - 1; i++) next[i] = i + 1;

next[n - 1] = -1;

}

void push(int item, int sn) {

if (freeTop == -1) return;

int i = freeTop;

freeTop = next[i];

arr[i] = item;

next[i] = top[sn];

top[sn] = i;

}

int pop(int sn) {

if (top[sn] == -1) return -1;

int i = top[sn];

top[sn] = next[i];

next[i] = freeTop;

freeTop = i;

return arr[i];

}

};

int main() {

KStacks ks(3, 10);

ks.push(15, 2);

ks.push(45, 2);

ks.push(17, 1);

ks.push(49, 1);

ks.push(39, 1);

ks.push(11, 0);

ks.push(9, 0);

ks.push(7, 0);

cout << ks.pop(2) << endl;

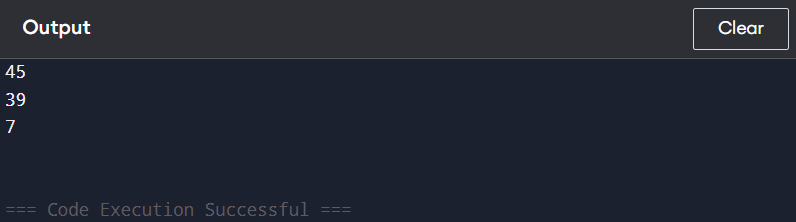
cout << ks.pop(1) << endl;

cout << ks.pop(0) << endl;

return 0;

}

**Output SS:**



**Problem-19 : Implement k Queues in a Single Array**  
**Code:**

#include <iostream>

using namespace std;

class KQueues {

int \*arr, \*front, \*rear, \*next;

int n, k, freeSpot;

public:

KQueues(int k, int n) {

this->k = k;

this->n = n;

arr = new int[n];

front = new int[k];

rear = new int[k];

next = new int[n];

freeSpot = 0;

for (int i = 0; i < k; i++) front[i] = -1;

for (int i = 0; i < n - 1; i++) next[i] = i + 1;

next[n - 1] = -1;

}

void enqueue(int data, int qn) {

if (freeSpot == -1) return;

int i = freeSpot;

freeSpot = next[i];

if (front[qn] == -1) front[qn] = i;

else next[rear[qn]] = i;

next[i] = -1;

rear[qn] = i;

arr[i] = data;

}

int dequeue(int qn) {

if (front[qn] == -1) return -1;

int i = front[qn];

front[qn] = next[i];

next[i] = freeSpot;

freeSpot = i;

return arr[i];

}

};

int main() {

KQueues kq(3, 10);

kq.enqueue(10, 0);

kq.enqueue(20, 1);

kq.enqueue(30, 2);

cout << kq.dequeue(0) << endl;

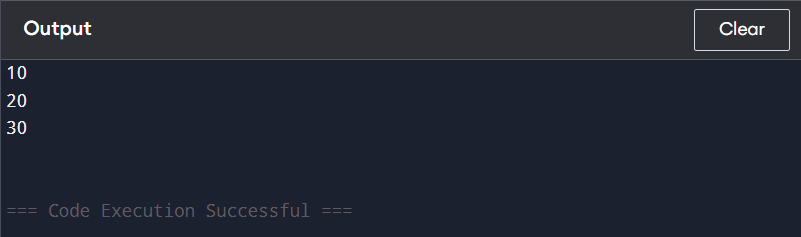
cout << kq.dequeue(1) << endl;

cout << kq.dequeue(2) << endl;

return 0;

}

**Output SS:**



**Problem-20 : Implement Min Heap using an Array**  
**Code:**

#include <iostream>

#include <vector>

using namespace std;

class MinHeap {

private:

vector<int> heap;

void heapifyUp(int index) {

while (index > 0 && heap[index] < heap[(index - 1) / 2]) {

swap(heap[index], heap[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

void heapifyDown(int index) {

int size = heap.size();

while (2 \* index + 1 < size) {

int smallest = 2 \* index + 1;

if (smallest + 1 < size && heap[smallest + 1] < heap[smallest]) {

smallest++;

}

if (heap[index] < heap[smallest]) break;

swap(heap[index], heap[smallest]);

index = smallest;

}

}

public:

void insert(int val) {

heap.push\_back(val);

heapifyUp(heap.size() - 1);

}

int extractMin() {

if (heap.empty()) return -1;

int minVal = heap[0];

heap[0] = heap.back();

heap.pop\_back();

heapifyDown(0);

return minVal;

}

void display() {

for (int num : heap) cout << num << " ";

cout << endl;

}

};

int main() {

MinHeap mh;

mh.insert(10);

mh.insert(5);

mh.insert(20);

mh.insert(2);

mh.display();

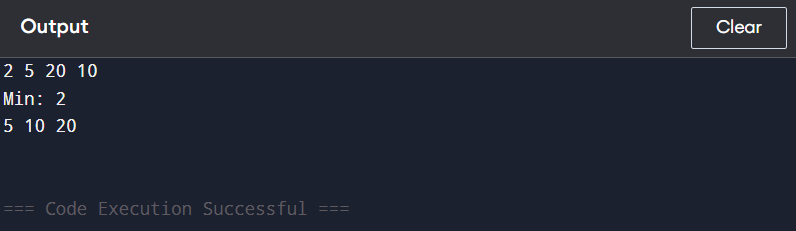
cout << "Min: " << mh.extractMin() << endl;

mh.display();

return 0;

}

**Output SS:**



**Problem-21 : Implement Max Heap using an Array**  
**Code:**

#include <iostream>

using namespace std;

class MaxHeap {

private:

int \*heap;

int capacity;

int size;

void heapifyUp(int index) {

while (index > 0) {

int parent = (index - 1) / 2;

if (heap[parent] < heap[index]) {

swap(heap[parent], heap[index]);

index = parent;

} else {

break;

}

}

}

void heapifyDown(int index) {

while (2 \* index + 1 < size) {

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int largest = left;

if (right < size && heap[right] > heap[left]) {

largest = right;

}

if (heap[index] < heap[largest]) {

swap(heap[index], heap[largest]);

index = largest;

} else {

break;

}

}

}

public:

MaxHeap(int cap) {

capacity = cap;

heap = new int[capacity];

size = 0;

}

void insert(int val) {

if (size == capacity) return;

heap[size] = val;

heapifyUp(size);

size++;

}

int extractMax() {

if (size == 0) return -1;

int maxVal = heap[0];

heap[0] = heap[size - 1];

size--;

heapifyDown(0);

return maxVal;

}

void display() {

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

cout << heap[i] << " ";

}

cout << endl;

}

};

int main() {

MaxHeap h(10);

h.insert(15);

h.insert(10);

h.insert(20);

h.insert(5);

h.display();

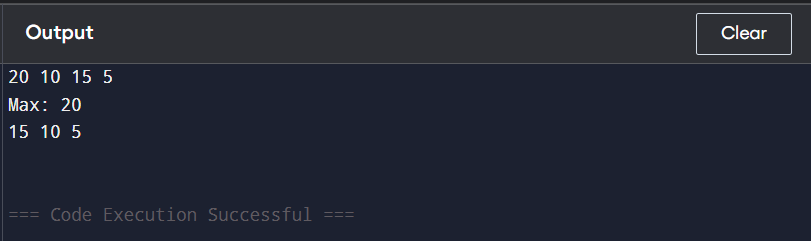
cout << "Max: " << h.extractMax() << endl;

h.display();

return 0;

}

**Output SS:**



**Problem-22 : Implement Hash Table using an Array (Linear Probing & Chaining)**  
**Code:**

#include <iostream>

#include <list>

using namespace std;

class HashTable {

private:

static const int size = 10;

list<int> table[size];

int hashFunction(int key) {

return key % size;

}

public:

void insertChaining(int key) {

int index = hashFunction(key);

table[index].push\_back(key);

}

bool searchChaining(int key) {

int index = hashFunction(key);

for (int val : table[index]) {

if (val == key) return true;

}

return false;

}

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

int hashTable[size] = {0};

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

int index = hashFunction(arr[i]);

while (hashTable[index] != 0) {

index = (index + 1) % size;

}

hashTable[index] = arr[i];

}

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

cout << hashTable[i] << " ";

}

cout << endl;

}

};

int main() {

HashTable ht;

ht.insertChaining(15);

ht.insertChaining(25);

cout << ht.searchChaining(15) << endl;

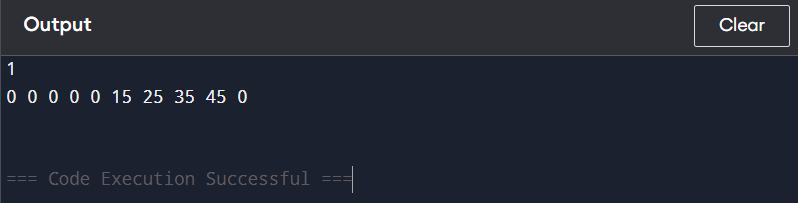
int arr[] = {15, 25, 35, 45};

ht.insertLinearProbing(arr, 4);

return 0;

}

**Output SS:**



**Problem-23 : Implement Trie using an Array**  
**Code:**

#include <iostream>

using namespace std;

class Trie {

private:

struct TrieNode {

TrieNode\* children[26];

bool isEndOfWord;

TrieNode() {

isEndOfWord = false;

for (int i = 0; i < 26; i++) children[i] = nullptr;

}

};

TrieNode\* root;

public:

Trie() {

root = new TrieNode();

}

void insert(string word) {

TrieNode\* node = root;

for (char c : word) {

int index = c - 'a';

if (!node->children[index]) node->children[index] = new TrieNode();

node = node->children[index];

}

node->isEndOfWord = true;

}

bool search(string word) {

TrieNode\* node = root;

for (char c : word) {

int index = c - 'a';

if (!node->children[index]) return false;

node = node->children[index];

}

return node->isEndOfWord;

}

bool startsWith(string prefix) {

TrieNode\* node = root;

for (char c : prefix) {

int index = c - 'a';

if (!node->children[index]) return false;

node = node->children[index];

}

return true;

}

};

int main() {

Trie trie;

trie.insert("apple");

cout << trie.search("apple") << endl;

cout << trie.search("app") << endl;

cout << trie.startsWith("app") << endl;

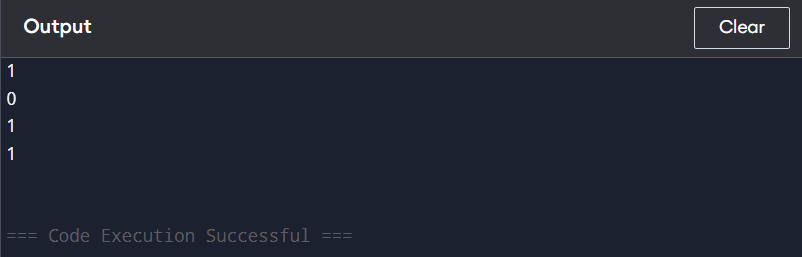
trie.insert("app");

cout << trie.search("app") << endl;

return 0;

}

**Output SS:**



**Problem-24 : Implement Graph using Adjacency Matrix (2D Array)**  
**Code:**

#include <iostream>

using namespace std;

class Graph {

private:

int vertices;

int\*\* adjMatrix;

public:

Graph(int v) {

vertices = v;

adjMatrix = new int\*[v];

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

adjMatrix[i] = new int[v];

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

adjMatrix[i][j] = 0;

}

}

}

void addEdge(int i, int j) {

adjMatrix[i][j] = 1;

adjMatrix[j][i] = 1;

}

void display() {

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

for (int j = 0; j < vertices; j++) {

cout << adjMatrix[i][j] << " ";

}

cout << endl;

}

}

~Graph() {

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

delete[] adjMatrix[i];

}

delete[] adjMatrix;

}

};

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

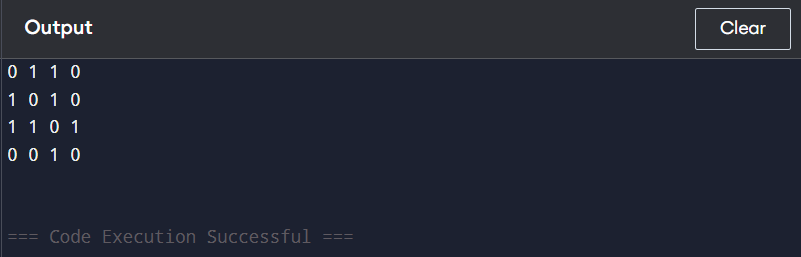
g.addEdge(2, 3);

g.display();

return 0;

}

**Output SS:**



**Problem-25 : Implement Stack using Linked List**

**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

class Stack {

private:

Node\* top;

public:

Stack() : top(nullptr) {}

void push(int x) {

Node\* newNode = new Node(x);

newNode->next = top;

top = newNode;

}

int pop() {

if (!top) return -1;

int val = top->data;

Node\* temp = top;

top = top->next;

delete temp;

return val;

}

int peek() {

return top ? top->data : -1;

}

bool empty() {

return top == nullptr;

}

};

int main() {

Stack s;

s.push(10);

s.push(20);

s.push(30);

cout << s.pop() << endl;

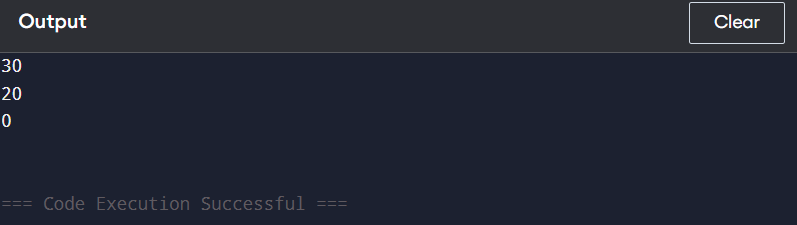
cout << s.peek() << endl;

cout << s.empty() << endl;

return 0;

}

**Output SS:**



**Problem-26 : Implement Queue using Linked List**  
**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

class Queue {

private:

Node\* front;

Node\* rear;

public:

Queue() : front(nullptr), rear(nullptr) {}

void enqueue(int val) {

Node\* newNode = new Node(val);

if (!rear) {

front = rear = newNode;

return;

}

rear->next = newNode;

rear = newNode;

}

int dequeue() {

if (!front) return -1;

Node\* temp = front;

front = front->next;

if (!front) rear = nullptr;

int val = temp->data;

delete temp;

return val;

}

bool isEmpty() {

return front == nullptr;

}

int peek() {

return front ? front->data : -1;

}

};

int main() {

Queue q;

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

cout << q.dequeue() << endl;

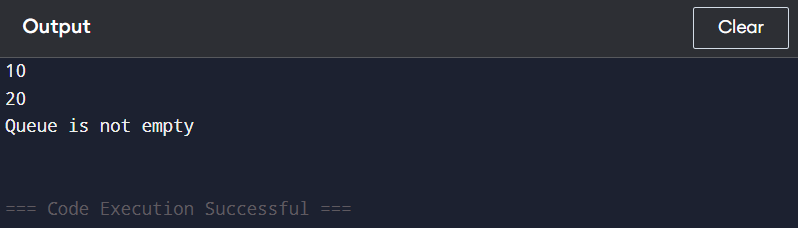
cout << q.peek() << endl;

cout << (q.isEmpty() ? "Queue is empty" : "Queue is not empty") << endl;

return 0;

}

**Output SS:**



**Problem-27 : Implement Deque using Doubly Linked List**  
**Code:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

class Deque {

private:

Node\* front;

Node\* rear;

public:

Deque() : front(nullptr), rear(nullptr) {}

void pushFront(int val) {

Node\* newNode = new Node(val);

if (!front) {

front = rear = newNode;

} else {

newNode->next = front;

front->prev = newNode;

front = newNode;

}

}

void pushBack(int val) {

Node\* newNode = new Node(val);

if (!rear) {

front = rear = newNode;

} else {

rear->next = newNode;

newNode->prev = rear;

rear = newNode;

}

}

void popFront() {

if (!front) return;

Node\* temp = front;

front = front->next;

if (front) front->prev = nullptr;

else rear = nullptr;

delete temp;

}

void popBack() {

if (!rear) return;

Node\* temp = rear;

rear = rear->prev;

if (rear) rear->next = nullptr;

else front = nullptr;

delete temp;

}

int getFront() {

return front ? front->data : -1;

}

int getBack() {

return rear ? rear->data : -1;

}

bool isEmpty() {

return front == nullptr;

}

};

int main() {

Deque dq;

dq.pushFront(10);

dq.pushBack(20);

cout << dq.getFront() << " " << dq.getBack() << endl;

dq.popFront();

cout << dq.getFront() << " " << dq.isEmpty() << endl;

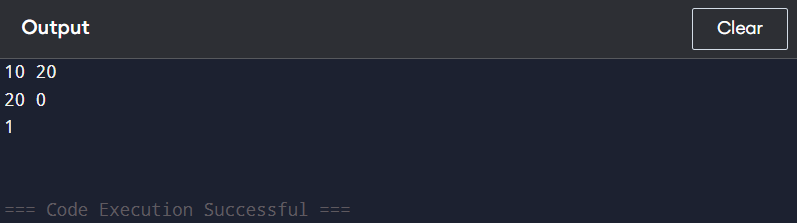
dq.popBack();

cout << dq.isEmpty() << endl;

return 0;

}

**Output SS:**



**Problem-28 : Implement Circular Queue using Linked List**  
**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

class CircularQueue {

private:

Node\* rear;

int size;

public:

CircularQueue() : rear(nullptr), size(0) {}

void enqueue(int val) {

Node\* newNode = new Node(val);

if (!rear) {

rear = newNode;

rear->next = rear;

} else {

newNode->next = rear->next;

rear->next = newNode;

rear = newNode;

}

size++;

}

void dequeue() {

if (!rear) return;

Node\* front = rear->next;

if (rear == front) rear = nullptr;

else rear->next = front->next;

delete front;

size--;

}

int front() {

return rear ? rear->next->data : -1;

}

bool isEmpty() {

return rear == nullptr;

}

void display() {

if (!rear) return;

Node\* temp = rear->next;

do {

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

temp = temp->next;

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

cout << endl;

}

};

int main() {

CircularQueue q;

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.display();

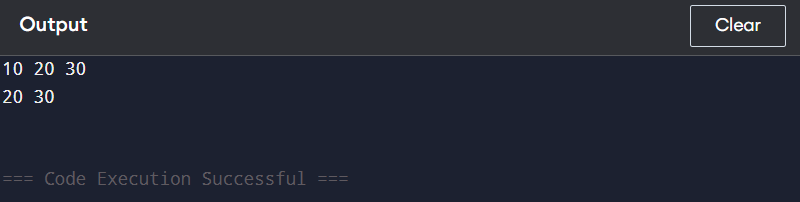
q.dequeue();

q.display();

return 0;

}

**Output SS:**



**Problem-29 : Implement Min Stack using Linked List**  
**Code:**

#include <iostream>

using namespace std;

class MinStack {

private:

struct Node {

int val, minVal;

Node\* next;

Node(int x, int minVal, Node\* next) : val(x), minVal(minVal), next(next) {}

};

Node\* head;

public:

MinStack() : head(nullptr) {}

void push(int val) {

if (!head) head = new Node(val, val, nullptr);

else head = new Node(val, min(val, head->minVal), head);

}

void pop() {

if (head) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

int top() {

return head ? head->val : -1;

}

int getMin() {

return head ? head->minVal : -1;

}

};

int main() {

MinStack s;

s.push(5);

s.push(2);

s.push(8);

cout << s.getMin() << endl;

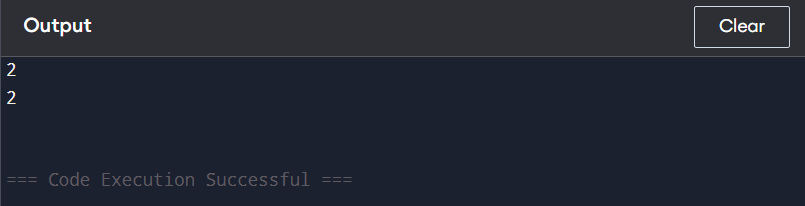
s.pop();

cout << s.getMin() << endl;

return 0;

}

**Output SS:**



**Problem-30 : Implement Hash Table using Linked List (Chaining Method)**  
**Code:**

#include <iostream>

#include <list>

using namespace std;

class HashTable {

private:

static const int TABLE\_SIZE = 10;

list<int> table[TABLE\_SIZE];

int hashFunction(int key) {

return key % TABLE\_SIZE;

}

public:

void insert(int key) {

int index = hashFunction(key);

table[index].push\_back(key);

}

void remove(int key) {

int index = hashFunction(key);

table[index].remove(key);

}

bool search(int key) {

int index = hashFunction(key);

for (int value : table[index]) {

if (value == key) return true;

}

return false;

}

void display() {

for (int i = 0; i < TABLE\_SIZE; i++) {

cout << i << ": ";

for (int value : table[i]) {

cout << value << " -> ";

}

cout << "NULL" << endl;

}

}

};

int main() {

HashTable ht;

ht.insert(10);

ht.insert(20);

ht.insert(30);

ht.insert(15);

ht.insert(25);

ht.display();

cout << (ht.search(20) ? "Found" : "Not Found") << endl;

ht.remove(20);

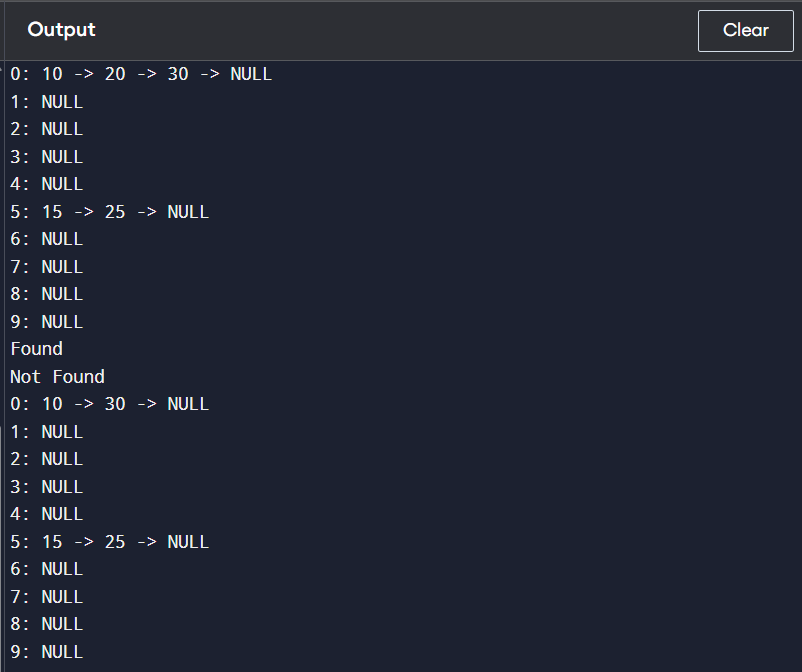
cout << (ht.search(20) ? "Found" : "Not Found") << endl;

ht.display();

return 0;

}

**Output SS:**



**Problem-31 : Implement BST using Linked List (Flattened Representation)**  
**Code:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

class BST {

public:

Node\* root;

BST() : root(nullptr) {}

Node\* insert(Node\* node, int val) {

if (!node) return new Node(val);

if (val < node->data) node->left = insert(node->left, val);

else node->right = insert(node->right, val);

return node;

}

void flatten(Node\* root, Node\*& prev) {

if (!root) return;

flatten(root->right, prev);

flatten(root->left, prev);

root->right = prev;

root->left = nullptr;

prev = root;

}

void inorder(Node\* node) {

if (!node) return;

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

};

int main() {

BST tree;

tree.root = tree.insert(tree.root, 5);

tree.insert(tree.root, 3);

tree.insert(tree.root, 7);

tree.insert(tree.root, 2);

tree.insert(tree.root, 4);

tree.insert(tree.root, 6);

tree.insert(tree.root, 8);

Node\* prev = nullptr;

tree.flatten(tree.root, prev);

Node\* temp = tree.root;

while (temp) {

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

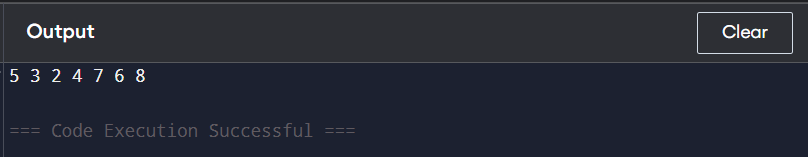
temp = temp->right;

}

return 0;

}

**Output SS:**



**Problem-32 : Implement Graph using Linked List (Adjacency List)**  
**Code:**

#include <iostream>

#include <vector>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int u, int v) {

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

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

}

void printGraph() {

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

cout << i << " -> ";

for (int v : adj[i]) {

cout << v << " ";

}

cout << endl;

}

}

};

int main() {

Graph g(5);

g.addEdge(0, 1);

g.addEdge(0, 4);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 3);

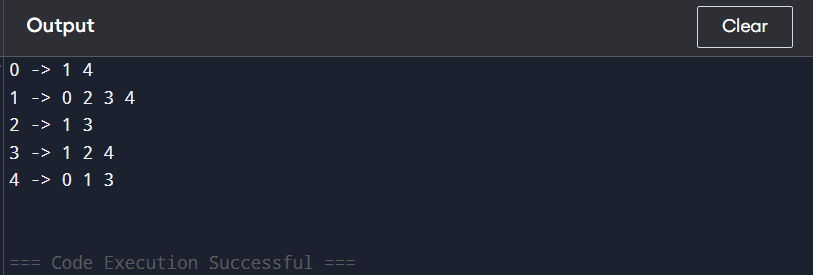
g.addEdge(3, 4);

g.printGraph();

return 0;

}

**Output SS:**



**Problem-33 : Implement Priority Queue using Heap**  
**Code:**

#include <iostream>

#include <vector>

using namespace std;

class PriorityQueue {

private:

vector<int> heap;

void heapifyUp(int index) {

while (index > 0 && heap[(index - 1) / 2] < heap[index]) {

swap(heap[index], heap[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

void heapifyDown(int index) {

int largest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if (left < heap.size() && heap[left] > heap[largest])

largest = left;

if (right < heap.size() && heap[right] > heap[largest])

largest = right;

if (largest != index) {

swap(heap[index], heap[largest]);

heapifyDown(largest);

}

}

public:

void push(int value) {

heap.push\_back(value);

heapifyUp(heap.size() - 1);

}

int pop() {

if (heap.empty()) return -1;

int top = heap[0];

heap[0] = heap.back();

heap.pop\_back();

heapifyDown(0);

return top;

}

int top() {

return heap.empty() ? -1 : heap[0];

}

bool empty() {

return heap.empty();

}

};

int main() {

PriorityQueue pq;

pq.push(10);

pq.push(20);

pq.push(15);

cout << pq.top() << endl;

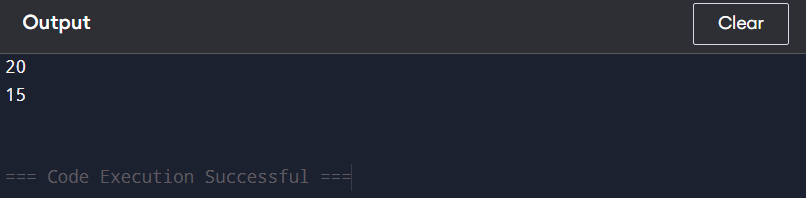
pq.pop();

cout << pq.top() << endl;

return 0;

}

**Output SS:**



**Problem-34 : Implement Min Heap using Max Heap**

**Code:**

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

class MinHeapUsingMaxHeap {

private:

priority\_queue<int> maxHeap;

public:

void push(int val) {

maxHeap.push(-val);

}

void pop() {

maxHeap.pop();

}

int top() {

return -maxHeap.top();

}

bool empty() {

return maxHeap.empty();

}

};

int main() {

MinHeapUsingMaxHeap minHeap;

minHeap.push(10);

minHeap.push(5);

minHeap.push(20);

minHeap.push(1);

while (!minHeap.empty()) {

cout << minHeap.top() << " ";

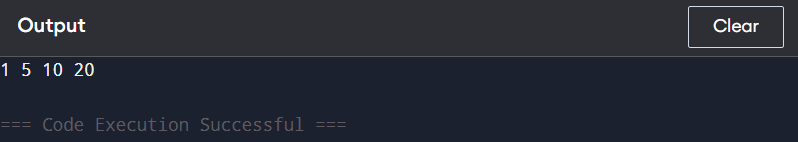
minHeap.pop();

}

return 0;

}

**Output SS:**



**Problem-35 : Implement Max Heap using Min Heap**  
**Code:**

#include <iostream>

#include <queue>

using namespace std;

class MaxHeap {

private:

priority\_queue<int, vector<int>, greater<int>> minHeap;

public:

void push(int val) {

minHeap.push(-val);

}

int pop() {

int top = -minHeap.top();

minHeap.pop();

return top;

}

int top() {

return -minHeap.top();

}

bool empty() {

return minHeap.empty();

}

};

int main() {

MaxHeap maxHeap;

maxHeap.push(10);

maxHeap.push(20);

maxHeap.push(5);

cout << maxHeap.top() << endl;

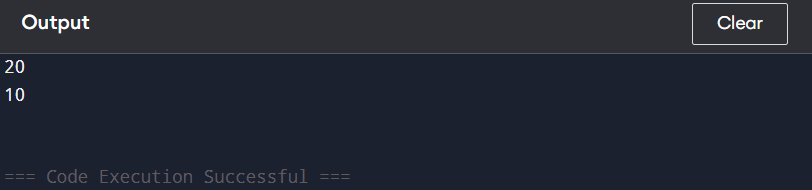
maxHeap.pop();

cout << maxHeap.top() << endl;

return 0;

}

**Output SS:**



**Problem-36 : Implement Median Finder using Two Heaps (Min Heap + Max Heap)**  
**Code:**

#include <bits/stdc++.h>

using namespace std;

class MedianFinder {

private:

priority\_queue<int> maxHeap;

priority\_queue<int, vector<int>, greater<int>> minHeap;

public:

void addNum(int num) {

if (maxHeap.empty() || num <= maxHeap.top()) maxHeap.push(num);

else minHeap.push(num);

if (maxHeap.size() > minHeap.size() + 1) {

minHeap.push(maxHeap.top());

maxHeap.pop();

} else if (minHeap.size() > maxHeap.size()) {

maxHeap.push(minHeap.top());

minHeap.pop();

}

}

double findMedian() {

if (maxHeap.size() > minHeap.size()) return maxHeap.top();

return (maxHeap.top() + minHeap.top()) / 2.0;

}

};

int main() {

MedianFinder mf;

mf.addNum(1);

mf.addNum(2);

cout << mf.findMedian() << endl;

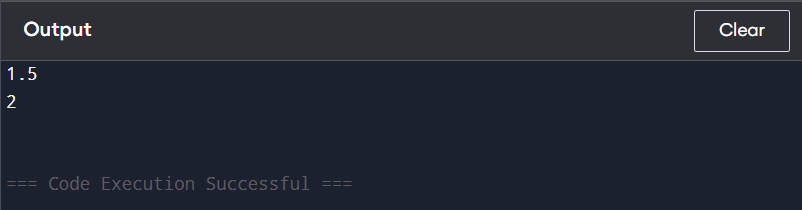
mf.addNum(3);

cout << mf.findMedian() << endl;

return 0;

}

**Output SS:**



**Problem-37 : Implement Kth Largest Element Finder using Heap**  
**Code:**

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

class KthLargest {

private:

priority\_queue<int, vector<int>, greater<int>> minHeap;

int k;

public:

KthLargest(int k, vector<int>& nums) {

this->k = k;

for (int num : nums) {

add(num);

}

}

int add(int val) {

minHeap.push(val);

if (minHeap.size() > k) {

minHeap.pop();

}

return minHeap.top();

}

};

int main() {

vector<int> nums = {4, 5, 8, 2};

KthLargest kthLargest(3, nums);

cout << kthLargest.add(3) << endl;

cout << kthLargest.add(5) << endl;

cout << kthLargest.add(10) << endl;

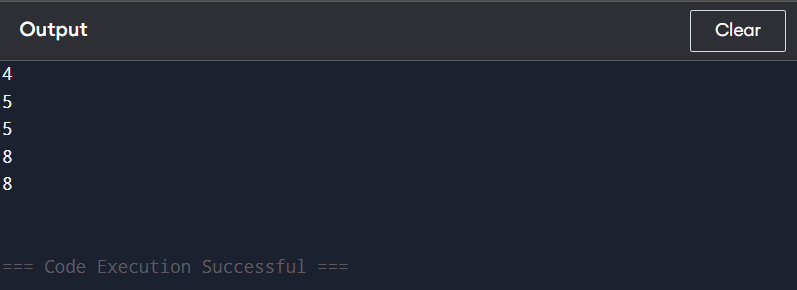
cout << kthLargest.add(9) << endl;

cout << kthLargest.add(4) << endl;

return 0;

}

**Output SS:**



**Problem-38 : Implement BST using Linked List**  
**Code:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(NULL), right(NULL) {}

};

class BST {

public:

Node\* root;

BST() : root(NULL) {}

Node\* insert(Node\* node, int val) {

if (!node) return new Node(val);

if (val < node->data) node->left = insert(node->left, val);

else node->right = insert(node->right, val);

return node;

}

void insert(int val) {

root = insert(root, val);

}

void inorder(Node\* node) {

if (!node) return;

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

void inorder() {

inorder(root);

cout << endl;

}

};

int main() {

BST tree;

tree.insert(5);

tree.insert(3);

tree.insert(7);

tree.insert(2);

tree.insert(4);

tree.insert(6);

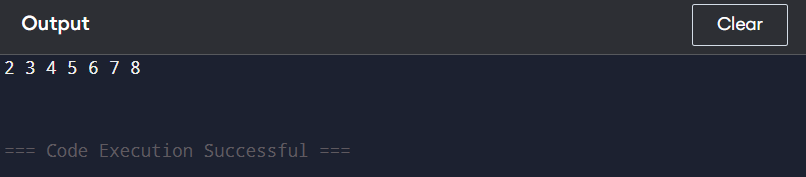
tree.insert(8);

tree.inorder();

return 0;

}

**Output SS:**



**Problem-39 : Implement AVL Tree using BST**  
**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int key;

Node \*left, \*right;

int height;

Node(int k) : key(k), left(nullptr), right(nullptr), height(1) {}

};

class AVLTree {

public:

Node\* insert(Node\* root, int key) {

if (!root) return new Node(key);

if (key < root->key) root->left = insert(root->left, key);

else if (key > root->key) root->right = insert(root->right, key);

else return root;

root->height = 1 + max(height(root->left), height(root->right));

return balance(root);

}

Node\* deleteNode(Node\* root, int key) {

if (!root) return root;

if (key < root->key) root->left = deleteNode(root->left, key);

else if (key > root->key) root->right = deleteNode(root->right, key);

else {

if (!root->left || !root->right) {

Node\* temp = root->left ? root->left : root->right;

delete root;

return temp;

} else {

Node\* temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

}

if (!root) return root;

root->height = 1 + max(height(root->left), height(root->right));

return balance(root);

}

void inorder(Node\* root) {

if (root) {

inorder(root->left);

cout << root->key << " ";

inorder(root->right);

}

}

private:

int height(Node\* node) { return node ? node->height : 0; }

int getBalance(Node\* node) { return node ? height(node->left) - height(node->right) : 0; }

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

y->left = x->right;

x->right = y;

y->height = 1 + max(height(y->left), height(y->right));

x->height = 1 + max(height(x->left), height(x->right));

return x;

}

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

x->right = y->left;

y->left = x;

x->height = 1 + max(height(x->left), height(x->right));

y->height = 1 + max(height(y->left), height(y->right));

return y;

}

Node\* balance(Node\* node) {

int balanceFactor = getBalance(node);

if (balanceFactor > 1 && getBalance(node->left) >= 0) return rotateRight(node);

if (balanceFactor > 1 && getBalance(node->left) < 0) {

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

return rotateRight(node);

}

if (balanceFactor < -1 && getBalance(node->right) <= 0) return rotateLeft(node);

if (balanceFactor < -1 && getBalance(node->right) > 0) {

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

return rotateLeft(node);

}

return node;

}

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current->left) current = current->left;

return current;

}

};

int main() {

AVLTree tree;

Node\* root = nullptr;

root = tree.insert(root, 10);

root = tree.insert(root, 20);

root = tree.insert(root, 30);

root = tree.insert(root, 40);

root = tree.insert(root, 50);

root = tree.insert(root, 25);

cout << "Inorder traversal: ";

tree.inorder(root);

cout << endl;

root = tree.deleteNode(root, 30);

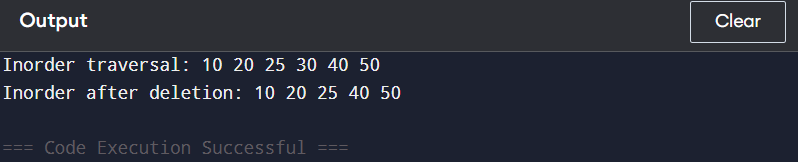
cout << "Inorder after deletion: ";

tree.inorder(root);

return 0;

}

**Output SS:**



**Problem-40 : Implement Trie using HashMap**  
**Code:**

#include <bits/stdc++.h>

using namespace std;

class Trie {

struct TrieNode {

unordered\_map<char, TrieNode\*> children;

bool isEndOfWord = false;

};

TrieNode\* root;

public:

Trie() { root = new TrieNode(); }

void insert(string word) {

TrieNode\* node = root;

for (char c : word) {

if (!node->children.count(c)) node->children[c] = new TrieNode();

node = node->children[c];

}

node->isEndOfWord = true;

}

bool search(string word) {

TrieNode\* node = root;

for (char c : word) {

if (!node->children.count(c)) return false;

node = node->children[c];

}

return node->isEndOfWord;

}

bool startsWith(string prefix) {

TrieNode\* node = root;

for (char c : prefix) {

if (!node->children.count(c)) return false;

node = node->children[c];

}

return true;

}

};

int main() {

Trie trie;

trie.insert("apple");

cout << trie.search("apple") << "\n";

cout << trie.search("app") << "\n";

cout << trie.startsWith("app") << "\n";

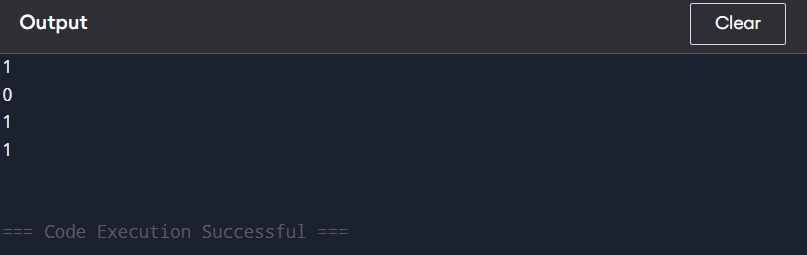
trie.insert("app");

cout << trie.search("app") << "\n";

return 0;

}

**Output SS:**



**Problem-41 : Implement Heap using BST**  
**Code:**

#include <iostream>

using namespace std;

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

class HeapBST {

public:

TreeNode\* root;

HeapBST() : root(NULL) {}

TreeNode\* insert(TreeNode\* node, int val) {

if (!node) return new TreeNode(val);

if (val > node->val) swap(val, node->val);

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

return node;

}

void insert(int val) {

root = insert(root, val);

}

void inorder(TreeNode\* node) {

if (!node) return;

inorder(node->left);

cout << node->val << " ";

inorder(node->right);

}

void print() {

inorder(root);

cout << endl;

}

};

int main() {

HeapBST heap;

heap.insert(10);

heap.insert(20);

heap.insert(15);

heap.insert(30);

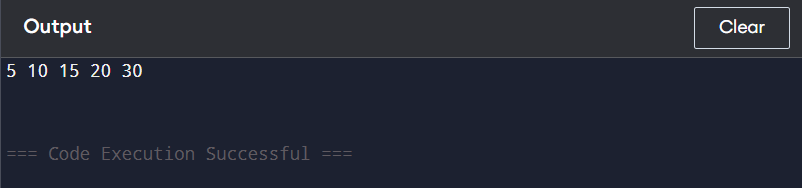
heap.insert(5);

heap.print();

return 0;

}

**Output SS:**



**Problem-42 : Implement Segment Tree using an Array**  
**Code:**

#include <iostream>

#include <vector>

using namespace std;

class SegmentTree {

private:

vector<int> tree;

int n;

void build(vector<int>& arr, int node, int start, int end) {

if (start == end) {

tree[node] = arr[start];

} else {

int mid = (start + end) / 2;

build(arr, 2 \* node + 1, start, mid);

build(arr, 2 \* node + 2, mid + 1, end);

tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2];

}

}

void update(int node, int start, int end, int idx, int val) {

if (start == end) {

tree[node] = val;

} else {

int mid = (start + end) / 2;

if (idx <= mid) update(2 \* node + 1, start, mid, idx, val);

else update(2 \* node + 2, mid + 1, end, idx, val);

tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2];

}

}

int query(int node, int start, int end, int l, int r) {

if (r < start || l > end) return 0;

if (l <= start && end <= r) return tree[node];

int mid = (start + end) / 2;

return query(2 \* node + 1, start, mid, l, r) + query(2 \* node + 2, mid + 1, end, l, r);

}

public:

SegmentTree(vector<int>& arr) {

n = arr.size();

tree.resize(4 \* n);

build(arr, 0, 0, n - 1);

}

void update(int idx, int val) {

update(0, 0, n - 1, idx, val);

}

int query(int l, int r) {

return query(0, 0, n - 1, l, r);

}

};

int main() {

vector<int> arr = {1, 3, 5, 7, 9, 11};

SegmentTree segTree(arr);

cout << segTree.query(1, 3) << endl;

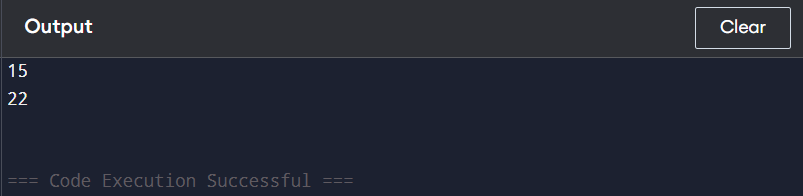
segTree.update(1, 10);

cout << segTree.query(1, 3) << endl;

return 0;

}

**Output SS:**



**Problem-43 : Implement Interval Tree using BST**

**Code:**

#include <iostream>

using namespace std;

struct Interval {

int low, high;

};

struct Node {

Interval \*i;

int max;

Node \*left, \*right;

};

Node\* newNode(Interval i) {

Node\* temp = new Node;

temp->i = new Interval(i);

temp->max = i.high;

temp->left = temp->right = nullptr;

return temp;

}

Node\* insert(Node\* root, Interval i) {

if (!root) return newNode(i);

int l = root->i->low;

if (i.low < l)

root->left = insert(root->left, i);

else

root->right = insert(root->right, i);

if (root->max < i.high)

root->max = i.high;

return root;

}

bool doOverlap(Interval i1, Interval i2) {

return (i1.low <= i2.high && i2.low <= i1.high);

}

Interval\* searchOverlap(Node\* root, Interval i) {

if (!root) return nullptr;

if (doOverlap(\*(root->i), i)) return root->i;

if (root->left && root->left->max >= i.low)

return searchOverlap(root->left, i);

return searchOverlap(root->right, i);

}

int main() {

Node\* root = nullptr;

Interval intervals[] = {{15, 20}, {10, 30}, {17, 19}, {5, 20}, {12, 15}, {30, 40}};

for (auto i : intervals)

root = insert(root, i);

Interval x = {14, 16};

Interval\* res = searchOverlap(root, x);

if (res)

cout << "Overlaps with: [" << res->low << ", " << res->high << "]\n";

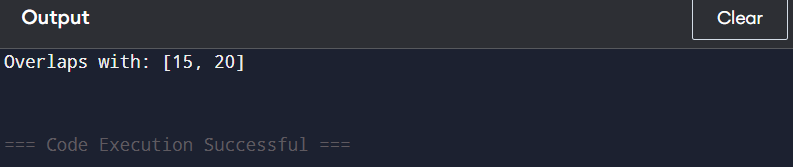
else

cout << "No Overlapping Interval\n";

return 0;

}

**Output SS:**



**Problem-44 : Implement LRU Cache using Hash Table + Doubly Linked List**

**Code:**

#include <bits/stdc++.h>

using namespace std;

class LRUCache {

private:

int capacity;

list<pair<int, int>> dll;

unordered\_map<int, list<pair<int, int>>::iterator> cache;

public:

LRUCache(int cap) { capacity = cap; }

int get(int key) {

if (cache.find(key) == cache.end()) return -1;

dll.splice(dll.begin(), dll, cache[key]);

return cache[key]->second;

}

void put(int key, int value) {

if (cache.find(key) != cache.end()) {

dll.splice(dll.begin(), dll, cache[key]);

cache[key]->second = value;

return;

}

if (dll.size() == capacity) {

cache.erase(dll.back().first);

dll.pop\_back();

}

dll.push\_front({key, value});

cache[key] = dll.begin();

}

};

int main() {

LRUCache lru(2);

lru.put(1, 1);

lru.put(2, 2);

cout << lru.get(1) << endl;

lru.put(3, 3);

cout << lru.get(2) << endl;

lru.put(4, 4);

cout << lru.get(1) << endl;

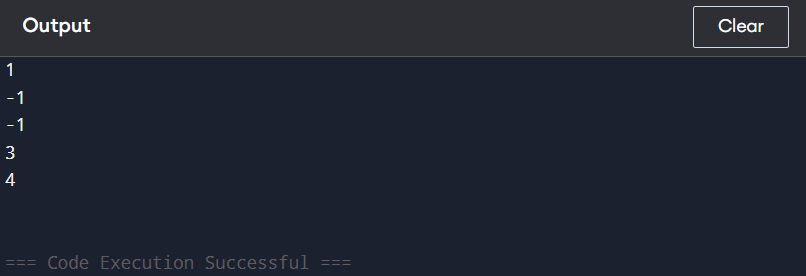
cout << lru.get(3) << endl;

cout << lru.get(4) << endl;

return 0;

}

**Output SS:**



**Problem-45 : Implement LFU Cache using Hash Table + Min Heap**

**Code:**

#include <bits/stdc++.h>

using namespace std;

class LFUCache {

public:

int capacity, minFreq;

unordered\_map<int, pair<int, int>> keyValue;

unordered\_map<int, list<int>> freqList;

unordered\_map<int, list<int>::iterator> keyIterator;

unordered\_map<int, int> keyFreq;

LFUCache(int capacity) {

this->capacity = capacity;

minFreq = 0;

}

void updateFreq(int key) {

int freq = keyFreq[key];

freqList[freq].erase(keyIterator[key]);

if (freqList[freq].empty() && freq == minFreq) minFreq++;

keyFreq[key]++;

freqList[freq + 1].push\_front(key);

keyIterator[key] = freqList[freq + 1].begin();

}

int get(int key) {

if (keyValue.find(key) == keyValue.end()) return -1;

updateFreq(key);

return keyValue[key].first;

}

void put(int key, int value) {

if (capacity == 0) return;

if (keyValue.find(key) != keyValue.end()) {

keyValue[key].first = value;

updateFreq(key);

return;

}

if (keyValue.size() >= capacity) {

int evictKey = freqList[minFreq].back();

freqList[minFreq].pop\_back();

keyValue.erase(evictKey);

keyFreq.erase(evictKey);

keyIterator.erase(evictKey);

}

keyValue[key] = {value, 1};

keyFreq[key] = 1;

freqList[1].push\_front(key);

keyIterator[key] = freqList[1].begin();

minFreq = 1;

}

};

int main() {

LFUCache cache(2);

cache.put(1, 1);

cache.put(2, 2);

cout << cache.get(1) << endl;

cache.put(3, 3);

cout << cache.get(2) << endl;

cout << cache.get(3) << endl;

cache.put(4, 4);

cout << cache.get(1) << endl;

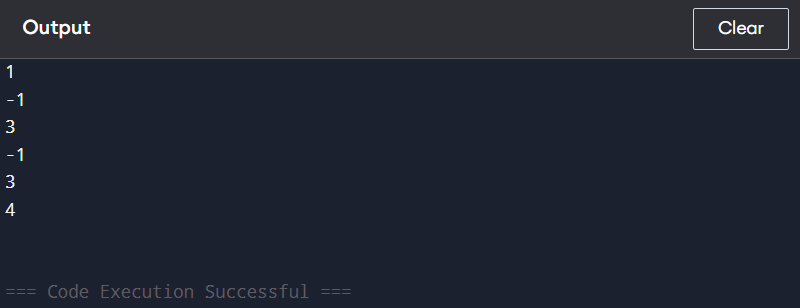
cout << cache.get(3) << endl;

cout << cache.get(4) << endl;

return 0;

}

**Output SS:**



**Problem-46 :  Implement Disjoint Set (Union-Find) using Hash Table**

**Code:**

#include <iostream>

#include <unordered\_map>

using namespace std;

class DisjointSet {

private:

unordered\_map<int, int> parent;

unordered\_map<int, int> rank;

public:

void makeSet(int x) {

parent[x] = x;

rank[x] = 1;

}

int find(int x) {

if (parent[x] != x)

parent[x] = find(parent[x]);

return parent[x];

}

void unite(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY])

parent[rootY] = rootX;

else if (rank[rootX] < rank[rootY])

parent[rootX] = rootY;

else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

};

int main() {

DisjointSet ds;

ds.makeSet(1);

ds.makeSet(2);

ds.makeSet(3);

ds.unite(1, 2);

cout << (ds.find(1) == ds.find(2)) << endl;

cout << (ds.find(2) == ds.find(3)) << endl;

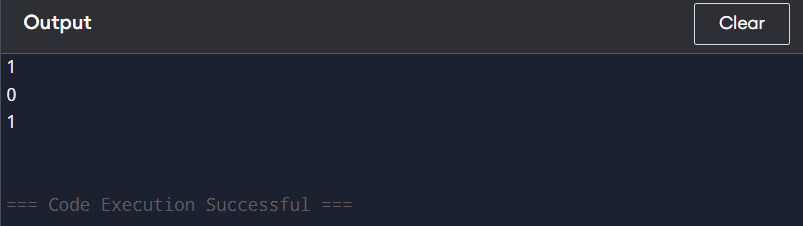
ds.unite(2, 3);

cout << (ds.find(1) == ds.find(3)) << endl;

return 0;

}

**Output SS:**



**Problem-47 : Implement Trie using Hash Table**

**Code:**

#include <iostream>

#include <unordered\_map>

using namespace std;

class Trie {

private:

struct TrieNode {

unordered\_map<char, TrieNode\*> children;

bool isEndOfWord = false;

};

TrieNode\* root;

public:

Trie() {

root = new TrieNode();

}

void insert(string word) {

TrieNode\* node = root;

for (char c : word) {

if (!node->children.count(c)) {

node->children[c] = new TrieNode();

}

node = node->children[c];

}

node->isEndOfWord = true;

}

bool search(string word) {

TrieNode\* node = root;

for (char c : word) {

if (!node->children.count(c)) {

return false;

}

node = node->children[c];

}

return node->isEndOfWord;

}

bool startsWith(string prefix) {

TrieNode\* node = root;

for (char c : prefix) {

if (!node->children.count(c)) {

return false;

}

node = node->children[c];

}

return true;

}

};

int main() {

Trie trie;

trie.insert("apple");

cout << trie.search("apple") << endl;

cout << trie.search("app") << endl;

cout << trie.startsWith("app") << endl;

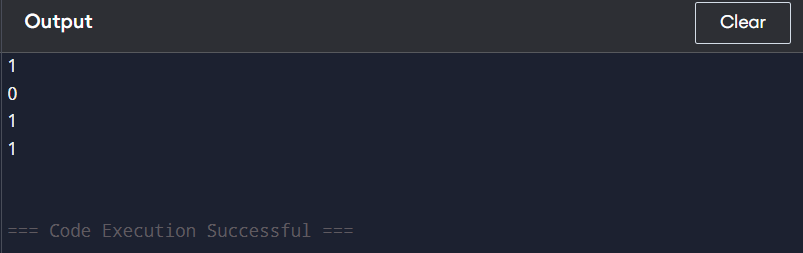
trie.insert("app");

cout << trie.search("app") << endl;

return 0;

}

**Output SS:**



**Problem-48 : Implement Graph using Hash Table (Adjacency List Representation)**

**Code:**

#include <iostream>

#include <unordered\_map>

#include <list>

using namespace std;

class Graph {

public:

unordered\_map<int, list<int>> adj;

void addEdge(int u, int v) {

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

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

}

void printGraph() {

for (auto& node : adj) {

cout << node.first << " -> ";

for (int neighbor : node.second) {

cout << neighbor << " ";

}

cout << endl;

}

}

};

int main() {

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

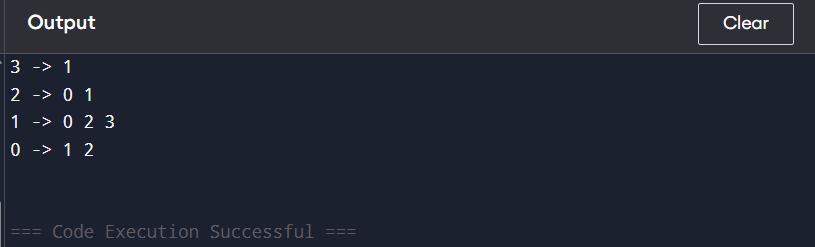
g.addEdge(1, 3);

g.printGraph();

return 0;

}

**Output SS:**



**Problem-49 : Implement Graph using Adjacency List (HashMap or Array of Lists)**

**Code:**

#include <iostream>

#include <unordered\_map>

#include <list>

using namespace std;

class Graph {

public:

unordered\_map<int, list<int>> adjList;

void addEdge(int u, int v) {

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

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

}

void printGraph() {

for (auto &pair : adjList) {

cout << pair.first << " -> ";

for (int vertex : pair.second) {

cout << vertex << " ";

}

cout << endl;

}

}

};

int main() {

Graph g;

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.addEdge(3, 4);

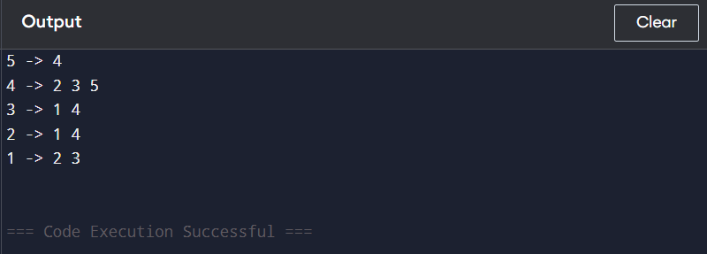
g.addEdge(4, 5);

g.printGraph();

return 0;

}

**Output SS:**



**Problem-50 : Implement Graph using Adjacency Matrix (2D Array)**

**Code:**

#include <iostream>

using namespace std;

class Graph {

private:

int vertices;

int\*\* adjMatrix;

public:

Graph(int v) {

vertices = v;

adjMatrix = new int\*[vertices];

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

adjMatrix[i] = new int[vertices];

for (int j = 0; j < vertices; j++) {

adjMatrix[i][j] = 0;

}

}

}

void addEdge(int u, int v) {

adjMatrix[u][v] = 1;

adjMatrix[v][u] = 1;

}

void display() {

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

for (int j = 0; j < vertices; j++) {

cout << adjMatrix[i][j] << " ";

}

cout << endl;

}

}

~Graph() {

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

delete[] adjMatrix[i];

}

delete[] adjMatrix;

}

};

int main() {

int v = 4;

Graph g(v);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

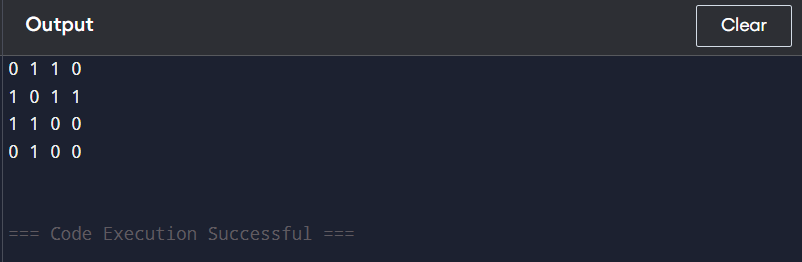
g.addEdge(1, 3);

g.display();

return 0;

}

**Output SS:**



**Problem-51 :  Implement BFS using Graph + Queue**

**Code:**

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int u, int v) {

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

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

}

void BFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : adj[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

};

int main() {

Graph g(6);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

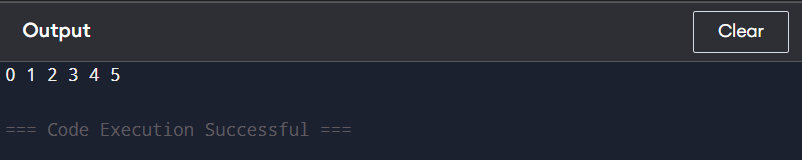
g.addEdge(2, 5);

g.BFS(0);

return 0;

}

**Output SS:**



**Problem-52 : Implement DFS using Graph + Stack**

**Code:**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int v, int w) {

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

}

void DFS(int start) {

vector<bool> visited(V, false);

stack<int> s;

s.push(start);

while (!s.empty()) {

int v = s.top();

s.pop();

if (!visited[v]) {

cout << v << " ";

visited[v] = true;

}

for (auto it = adj[v].rbegin(); it != adj[v].rend(); ++it) {

if (!visited[\*it]) {

s.push(\*it);

}

}

}

}

};

int main() {

Graph g(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

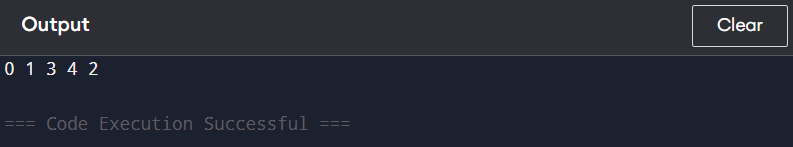
g.addEdge(2, 4);

g.DFS(0);

return 0;

}

**Output SS:**



**Problem-53 : Implement Topological Sorting using Graph + Stack (DFS)**

**Code:**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

void topologicalSortUtil(int v, vector<bool>& visited, stack<int>& Stack) {

visited[v] = true;

for (int i : adj[v]) {

if (!visited[i]) {

topologicalSortUtil(i, visited, Stack);

}

}

Stack.push(v);

}

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int v, int w) {

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

}

void topologicalSort() {

stack<int> Stack;

vector<bool> visited(V, false);

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

if (!visited[i]) {

topologicalSortUtil(i, visited, Stack);

}

}

while (!Stack.empty()) {

cout << Stack.top() << " ";

Stack.pop();

}

}

};

int main() {

Graph g(6);

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

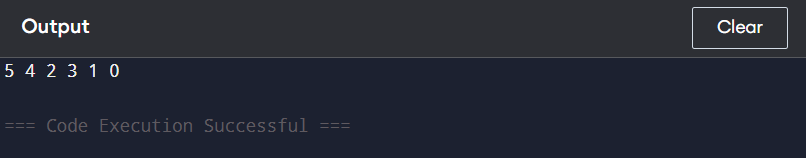
g.addEdge(3, 1);

g.topologicalSort();

return 0;

}

**Output SS:**



**Problem-54 : Implement Kruskal’s Algorithm using Graph + Disjoint Set**

**Code:**

#include <bits/stdc++.h>

using namespace std;

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

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

}

int find(int x) {

if (parent[x] != x) parent[x] = find(parent[x]);

return parent[x];

}

void unite(int x, int y) {

int rootX = find(x), rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY]) parent[rootY] = rootX;

else if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;

else { parent[rootY] = rootX; rank[rootX]++; }

}

}

};

struct Edge {

int u, v, weight;

bool operator<(const Edge& other) const {

return weight < other.weight;

}

};

class Graph {

public:

int V;

vector<Edge> edges;

Graph(int V) : V(V) {}

void addEdge(int u, int v, int weight) {

edges.push\_back({u, v, weight});

}

int kruskalMST() {

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

DisjointSet ds(V);

int mstWeight = 0;

for (Edge& e : edges) {

if (ds.find(e.u) != ds.find(e.v)) {

ds.unite(e.u, e.v);

mstWeight += e.weight;

}

}

return mstWeight;

}

};

int main() {

int V = 4;

Graph g(V);

g.addEdge(0, 1, 10);

g.addEdge(0, 2, 6);

g.addEdge(0, 3, 5);

g.addEdge(1, 3, 15);

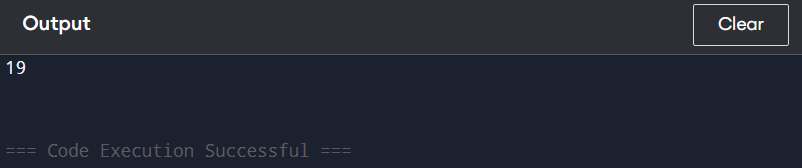
g.addEdge(2, 3, 4);

cout << g.kruskalMST() << endl;

return 0;

}

**Output SS:**



**Problem-55 : Implement Dijkstra’s Algorithm using Graph + Priority Queue (Heap)**

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

typedef pair<int, int> pii;

void dijkstra(int V, vector<vector<pii>>& adj, int src) {

vector<int> dist(V, INT\_MAX);

priority\_queue<pii, vector<pii>, greater<pii>> pq;

dist[src] = 0;

pq.push({0, src});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto &[v, weight] : adj[u]) {

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.push({dist[v], v});

}

}

}

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

cout << "Vertex " << i << " -> Distance: " << dist[i] << endl;

}

}

int main() {

int V = 5;

vector<vector<pii>> adj(V);

adj[0].push\_back({1, 10});

adj[0].push\_back({4, 3});

adj[1].push\_back({2, 2});

adj[1].push\_back({4, 4});

adj[2].push\_back({3, 9});

adj[3].push\_back({2, 7});

adj[4].push\_back({1, 1});

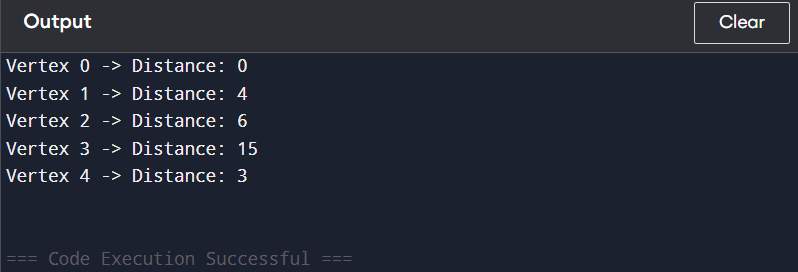
adj[4].push\_back({2, 8});

dijkstra(V, adj, 0);

return 0;

}

**Output SS:**



**Problem-56 : Implement Circular Buffer using an Array**

**Code:**

#include <iostream>

using namespace std;

class CircularBuffer {

private:

int \*buffer;

int head, tail, size, capacity;

public:

CircularBuffer(int cap) {

capacity = cap;

buffer = new int[capacity];

head = tail = size = 0;

}

bool isFull() {

return size == capacity;

}

bool isEmpty() {

return size == 0;

}

void enqueue(int value) {

if (isFull()) return;

buffer[tail] = value;

tail = (tail + 1) % capacity;

size++;

}

int dequeue() {

if (isEmpty()) return -1;

int value = buffer[head];

head = (head + 1) % capacity;

size--;

return value;

}

int front() {

return isEmpty() ? -1 : buffer[head];

}

~CircularBuffer() {

delete[] buffer;

}

};

int main() {

CircularBuffer cb(5);

cb.enqueue(1);

cb.enqueue(2);

cb.enqueue(3);

cb.enqueue(4);

cb.enqueue(5);

cout << cb.dequeue() << endl;

cout << cb.front() << endl;

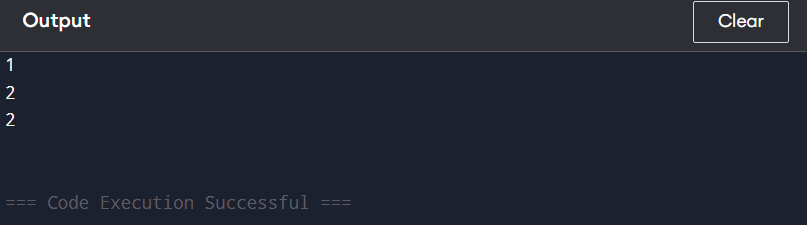
cb.enqueue(6);

cout << cb.front() << endl;

return 0;

}

**Output SS:**



**Problem-57 : Implement Sparse Table for Range Queries using an Array**

**Code:**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

class SparseTable {

vector<vector<int>> st;

vector<int> log;

public:

SparseTable(vector<int>& arr) {

int n = arr.size();

int k = log2(n) + 1;

st.assign(n, vector<int>(k));

log.assign(n + 1, 0);

for (int i = 2; i <= n; i++)

log[i] = log[i / 2] + 1;

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

st[i][0] = arr[i];

for (int j = 1; (1 << j) <= n; j++)

for (int i = 0; i + (1 << j) <= n; i++)

st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);

}

int query(int L, int R) {

int j = log[R - L + 1];

return min(st[L][j], st[R - (1 << j) + 1][j]);

}

};

int main() {

vector<int> arr = {1, 3, 2, 7, 9, 11, 3, 5, 6};

SparseTable st(arr);

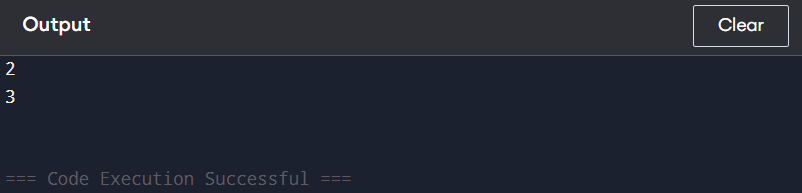
cout << st.query(1, 4) << endl;

cout << st.query(3, 7) << endl;

return 0;

}

**Output SS:**



**Problem-58 : Implement Sliding Window Maximum using Deque**

**Code:**

#include <iostream>

#include <vector>

#include <deque>

using namespace std;

vector<int> maxSlidingWindow(vector<int>& nums, int k) {

vector<int> result;

deque<int> dq;

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

if (!dq.empty() && dq.front() == i - k) dq.pop\_front();

while (!dq.empty() && nums[dq.back()] <= nums[i]) dq.pop\_back();

dq.push\_back(i);

if (i >= k - 1) result.push\_back(nums[dq.front()]);

}

return result;

}

int main() {

vector<int> nums = {1,3,-1,-3,5,3,6,7};

int k = 3;

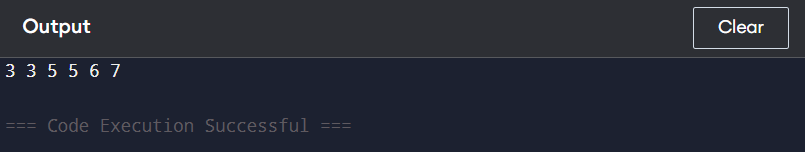
vector<int> result = maxSlidingWindow(nums, k);

for (int num : result) cout << num << " ";

return 0;

}

**Output SS:**



**Problem-59 : Implement Graph Cycle Detection using Union-Find**

**Code:**

#include <iostream>

#include <vector>

using namespace std;

class Graph {

int V;

vector<pair<int, int>> edges;

vector<int> parent, rank;

public:

Graph(int V) : V(V) {

parent.resize(V);

rank.resize(V, 0);

for (int i = 0; i < V; i++) parent[i] = i;

}

void addEdge(int u, int v) {

edges.push\_back({u, v});

}

int find(int x) {

if (parent[x] != x) parent[x] = find(parent[x]);

return parent[x];

}

bool unionSet(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX == rootY) return true;

if (rank[rootX] > rank[rootY]) parent[rootY] = rootX;

else if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;

else { parent[rootY] = rootX; rank[rootX]++; }

return false;

}

bool hasCycle() {

for (auto &[u, v] : edges) {

if (unionSet(u, v)) return true;

}

return false;

}

};

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

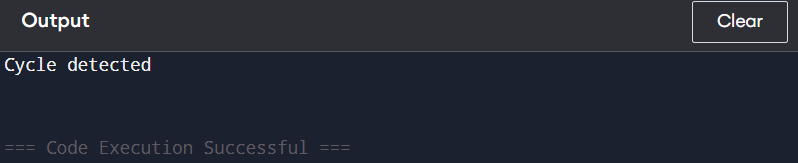
g.addEdge(3, 0);

cout << (g.hasCycle() ? "Cycle detected" : "No cycle detected") << endl;

return 0;

}

**Output SS:**



**Problem-60 : Implement A Pathfinding using Graph + Priority Queue**

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

struct Edge {

int to, weight;

Edge(int t, int w) : to(t), weight(w) {}

};

vector<int> dijkstra(int n, vector<vector<Edge>>& graph, int start) {

vector<int> dist(n, numeric\_limits<int>::max());

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

dist[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int d = pq.top().first, u = pq.top().second;

pq.pop();

if (d > dist[u]) continue;

for (Edge& e : graph[u]) {

int v = e.to, w = e.weight;

if (dist[u] + w < dist[v]) {

dist[v] = dist[u] + w;

pq.push({dist[v], v});

}

}

}

return dist;

}

int main() {

int n = 5;

vector<vector<Edge>> graph(n);

graph[0].push\_back(Edge(1, 2));

graph[0].push\_back(Edge(3, 6));

graph[1].push\_back(Edge(2, 3));

graph[1].push\_back(Edge(3, 8));

graph[1].push\_back(Edge(4, 5));

graph[2].push\_back(Edge(4, 7));

graph[3].push\_back(Edge(4, 9));

int start = 0;

vector<int> distances = dijkstra(n, graph, start);

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

cout << "Distance from " << start << " to " << i << " is " << distances[i] << endl;

}

return 0;

}

**Output SS:**

