1. Write C++ code for merge sort and analyze time complexity.

#include <iostream>

using namespace std;

// Merge two sorted subarrays into one

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1; // size of left half

int n2 = right - mid; // size of right half

int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

// Merge the temp arrays back into arr[]

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j])

arr[k++] = L[i++];

else

arr[k++] = R[j++];

}

// Copy remaining elements of L[]

while (i < n1)

arr[k++] = L[i++];

// Copy remaining elements of R[]

while (j < n2)

arr[k++] = R[j++];

}

// Recursive merge sort function

void mergeSort(int arr[], int left, int right) {

if (left >= right) return; // base case

int mid = (left + right) / 2;

mergeSort(arr, left, mid); // sort left half

mergeSort(arr, mid + 1, right); // sort right half

merge(arr, left, mid, right); // merge both halves

}

int main() {

int arr[] = {6, 3, 8, 5, 2, 7, 4, 1};

int size = sizeof(arr) / sizeof(arr[0]);

cout << "Original array: ";

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

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

cout << endl;

mergeSort(arr, 0, size - 1);

cout << "Sorted array: ";

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

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

cout << endl;

return 0;

}

1. Implement 0/1 Knapsack problem or 8 Queen’s problem

#include <iostream>

#include <vector>

using namespace std;

// Function to solve 0/1 Knapsack problem

int knapsack(int capacity, vector<int>& weights, vector<int>& values, int n) {

// Create a 2D array for dynamic programming

vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

// Build table dp[][] in bottom-up manner

for (int i = 1; i <= n; i++) {

for (int w = 1; w <= capacity; w++) {

// If current item can fit in the knapsack

if (weights[i-1] <= w) {

// Maximum of two cases:

// 1. Include the current item

// 2. Exclude the current item

dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w]);

} else {

// Item cannot be included, so just copy the value from the previous row

dp[i][w] = dp[i-1][w];

}

}

}

// Return the maximum value that can be put in a knapsack of capacity W

return dp[n][capacity];

}

int main() {

// Example usage

vector<int> values = {60, 100, 120};

vector<int> weights = {10, 20, 30};

int capacity = 50;

int n = values.size();

int max\_value = knapsack(capacity, weights, values, n);

cout << "Maximum value that can be obtained: " << max\_value << endl;

return 0;

}

1. Implement Cuckoo Hashing

#include <iostream>

using namespace std;

// Simple Cuckoo Hashing implementation

class CuckooHash {

private:

int\* table1; // First hash table

int\* table2; // Second hash table

int size; // Size of each table

int MAX\_LOOP = 100; // Maximum number of relocations

// Simple hash functions

int hash1(int key) {

return key % size;

}

int hash2(int key) {

return (key / size) % size;

}

public:

// Constructor

CuckooHash(int tableSize) {

size = tableSize;

table1 = new int[size];

table2 = new int[size];

// Initialize tables with -1 (indicating empty)

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

table1[i] = -1;

table2[i] = -1;

}

}

// Destructor

~CuckooHash() {

delete[] table1;

delete[] table2;

}

// Insert a key into the hash table

bool insert(int key) {

// If key already exists, don't insert again

if (lookup(key)) {

return true;

}

// Try to insert in table1 first

for (int loop = 0; loop < MAX\_LOOP; loop++) {

// Try table1

int pos1 = hash1(key);

if (table1[pos1] == -1) {

table1[pos1] = key;

return true;

}

// Swap with what's in table1

int temp = table1[pos1];

table1[pos1] = key;

key = temp;

// Try table2

int pos2 = hash2(key);

if (table2[pos2] == -1) {

table2[pos2] = key;

return true;

}

// Swap with what's in table2

temp = table2[pos2];

table2[pos2] = key;

key = temp;

}

// If we reach here, we couldn't find a place for the key

// Need to rehash with bigger tables, but for simplicity:

cout << "Failed to insert " << key << " after " << MAX\_LOOP << " attempts" << endl;

return false;

}

// Look up a key in the hash table

bool lookup(int key) {

int pos1 = hash1(key);

int pos2 = hash2(key);

return (table1[pos1] == key || table2[pos2] == key);

}

// Remove a key from the hash table

bool remove(int key) {

int pos1 = hash1(key);

if (table1[pos1] == key) {

table1[pos1] = -1;

return true;

}

int pos2 = hash2(key);

if (table2[pos2] == key) {

table2[pos2] = -1;

return true;

}

return false;

}

// Display the contents of both tables

void display() {

cout << "Table 1: ";

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

if (table1[i] == -1) {

cout << "- ";

} else {

cout << table1[i] << " ";

}

}

cout << endl;

cout << "Table 2: ";

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

if (table2[i] == -1) {

cout << "- ";

} else {

cout << table2[i] << " ";

}

}

cout << endl;

}

};

// Example usage

int main() {

CuckooHash hash(10);

// Insert some keys

hash.insert(20);

hash.insert(50);

hash.insert(53);

hash.insert(75);

hash.insert(100);

hash.insert(67);

cout << "After insertions:" << endl;

hash.display();

// Look up some keys

cout << "Lookup 50: " << (hash.lookup(50) ? "Found" : "Not found") << endl;

cout << "Lookup 25: " << (hash.lookup(25) ? "Found" : "Not found") << endl;

// Remove a key

hash.remove(50);

cout << "After removing 50:" << endl;

hash.display();

return 0;

}

4.Implement any real time application with the help of hashing concept and handle collision using linear probing without replacement.

#include <iostream>

#include <iomanip>

using namespace std;

struct Student {

int rollNo;

string name;

float marks;

bool isOccupied;

Student() {

rollNo = -1;

name = "";

marks = 0.0;

isOccupied = false;

}

};

class StudentHashTable {

private:

Student\* table;

int size;

int hashFunction(int key) {

return key % size;

}

public:

StudentHashTable(int s) {

size = s;

table = new Student[size];

}

~StudentHashTable() {

delete[] table;

}

void insert(int rollNo, string name, float marks) {

int index = hashFunction(rollNo);

int start = index;

while (table[index].isOccupied) {

if (table[index].rollNo == rollNo) {

cout << "Duplicate roll number. Cannot insert.\n";

return;

}

index = (index + 1) % size;

if (index == start) {

cout << "Hash table is full. Cannot insert.\n";

return;

}

}

table[index].rollNo = rollNo;

table[index].name = name;

table[index].marks = marks;

table[index].isOccupied = true;

cout << "Student inserted successfully at index " << index << ".\n";

}

void search(int rollNo) {

int index = hashFunction(rollNo);

int start = index;

while (table[index].isOccupied) {

if (table[index].rollNo == rollNo) {

cout << "Student Found:\n";

cout << "Roll No: " << table[index].rollNo << "\n";

cout << "Name: " << table[index].name << "\n";

cout << "Marks: " << table[index].marks << "\n";

return;

}

index = (index + 1) % size;

if (index == start) break;

}

cout << "Student with Roll No " << rollNo << " not found.\n";

}

void display() {

cout << "\nHash Table:\n";

cout << "----------------------------------------------------\n";

cout << setw(5) << "Index" << setw(10) << "Roll No" << setw(15) << "Name" << setw(10) << "Marks" << "\n";

cout << "----------------------------------------------------\n";

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

if (table[i].isOccupied)

cout << setw(5) << i << setw(10) << table[i].rollNo << setw(15) << table[i].name << setw(10) << table[i].marks << "\n";

else

cout << setw(5) << i << setw(10) << "-" << setw(15) << "-" << setw(10) << "-" << "\n";

}

}

};

int main() {

int tableSize;

cout << "Enter size of the hash table: ";

cin >> tableSize;

StudentHashTable ht(tableSize);

int choice, rollNo;

string name;

float marks;

do {

cout << "\n1. Insert Student\n2. Search Student\n3. Display Table\n4. Exit\n";

cout << "Enter choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter Roll No: ";

cin >> rollNo;

cout << "Enter Name: ";

cin >> name;

cout << "Enter Marks: ";

cin >> marks;

ht.insert(rollNo, name, marks);

break;

case 2:

cout << "Enter Roll No to search: ";

cin >> rollNo;

ht.search(rollNo);

break;

case 3:

ht.display();

break;

case 4:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Try again.\n";

}

} while (choice != 4);

return 0;

}

1. A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.

#include <iostream>

#include <string>

using namespace std;

// Node structure representing each keyword-meaning pair in the BST

struct Node {

string keyword;

string meaning;

Node\* left;

Node\* right;

// Constructor to initialize a new node

Node(string k, string m) {

keyword = k;

meaning = m;

left = right = NULL;

}

};

// Dictionary class which manages the BST and its operations

class Dictionary {

private:

Node\* root;

// Internal recursive insert function

Node\* insert(Node\* root, string keyword, string meaning) {

if (!root) {

return new Node(keyword, meaning); // Insert new node

}

if (keyword < root->keyword) {

root->left = insert(root->left, keyword, meaning);

} else if (keyword > root->keyword) {

root->right = insert(root->right, keyword, meaning);

} else {

cout << "❌ Keyword already exists!\n";

}

return root;

}

// Internal recursive delete function

Node\* deleteNode(Node\* root, string keyword) {

if (!root) return root;

if (keyword < root->keyword) {

root->left = deleteNode(root->left, keyword);

} else if (keyword > root->keyword) {

root->right = deleteNode(root->right, keyword);

} else {

// Node to be deleted found

// Case 1: Leaf node

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

delete root;

return NULL;

}

// Case 2: One child

if (!root->left) {

Node\* temp = root->right;

delete root;

return temp;

}

if (!root->right) {

Node\* temp = root->left;

delete root;

return temp;

}

// Case 3: Two children - find inorder successor

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

root->keyword = temp->keyword;

root->meaning = temp->meaning;

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

}

return root;

}

// Helper to find minimum node (used in delete)

Node\* findmin(Node\* root) {

while (root->left) {

root = root->left;

}

return root;

}

// Internal search

Node\* search(Node\* root, string keyword) {

if (!root || root->keyword == keyword) {

return root;

}

if (keyword < root->keyword)

return search(root->left, keyword);

else

return search(root->right, keyword);

}

// In-order traversal for ascending order display

void inorder(Node\* root) {

if (root) {

inorder(root->left);

cout << root->keyword << " : " << root->meaning << endl;

inorder(root->right);

}

}

// Reverse in-order traversal for descending order display

void reverseInorder(Node\* root) {

if (root) {

reverseInorder(root->right);

cout << root->keyword << " : " << root->meaning << endl;

reverseInorder(root->left);

}

}

// Count maximum comparisons to find a keyword (Binary Search path length)

int maxComp(Node\* root, string keyword) {

int count = 0;

while (root) {

count++;

if (root->keyword == keyword)

return count;

else if (keyword < root->keyword)

root = root->left;

else

root = root->right;

}

return count; // If not found, still return how far it searched

}

public:

// Constructor

Dictionary() {

root = NULL;

}

// Public insert interface

void insert(string keyword, string meaning) {

root = insert(root, keyword, meaning);

}

// Public delete interface

void deleteKeyword(string keyword) {

root = deleteNode(root, keyword);

}

// Update a keyword’s meaning

void update(string keyword, string newMeaning) {

Node\* temp = search(root, keyword);

if (temp) {

temp->meaning = newMeaning;

cout << "✅ Meaning updated.\n";

} else {

cout << "❌ Keyword not found!\n";

}

}

// Display all entries in ascending order

void displayAscending() {

inorder(root);

}

// Display all entries in descending order

void displayDescending() {

reverseInorder(root);

}

// Search and display a keyword’s meaning

void searchKeyword(string keyword) {

Node\* temp = search(root, keyword);

if (temp) {

cout << "✅ Found -> Keyword: " << temp->keyword << " | Meaning: " << temp->meaning << endl;

} else {

cout << "❌ Keyword not found!\n";

}

}

// Show how many comparisons it took (max path)

void findMaxComp(string keyword) {

int comp = maxComp(root, keyword);

cout << "📊 Maximum comparisons required to find \"" << keyword << "\": " << comp << endl;

}

};

// Main function for interaction

int main() {

Dictionary dict;

int choice;

string keyword, meaning;

do {

cout << "\n📘 Dictionary Menu:\n";

cout << "1. Insert\n";

cout << "2. Delete\n";

cout << "3. Update\n";

cout << "4. Display Ascending\n";

cout << "5. Display Descending\n";

cout << "6. Search\n";

cout << "7. Max Comparisons\n";

cout << "8. Exit\n";

cout << "Enter choice: ";

cin >> choice;

cin.ignore(); // Clear input buffer

switch (choice) {

case 1:

cout << "Enter Keyword: ";

getline(cin, keyword);

cout << "Enter Meaning: ";

getline(cin, meaning);

dict.insert(keyword, meaning);

break;

case 2:

cout << "Enter keyword to be deleted: ";

getline(cin, keyword);

dict.deleteKeyword(keyword);

break;

case 3:

cout << "Enter keyword to update: ";

getline(cin, keyword);

cout << "Enter new meaning: ";

getline(cin, meaning);

dict.update(keyword, meaning);

break;

case 4:

cout << "\n📚 Dictionary (Ascending Order):\n";

dict.displayAscending();

break;

case 5:

cout << "\n📚 Dictionary (Descending Order):\n";

dict.displayDescending();

break;

case 6:

cout << "Enter keyword to search: ";

getline(cin, keyword);

dict.searchKeyword(keyword);

break;

case 7:

cout << "Enter keyword to calculate max comparisons: ";

getline(cin, keyword);

dict.findMaxComp(keyword);

break;

case 8:

cout << "👋 Exiting... Thank you!\n";

break;

default:

cout << "❌ Invalid choice! Please try again.\n";

break;

}

} while (choice != 8);

return 0;

}

1. For given expression eg. a-b\*c-d/e+f construct inorder sequence and traverse it using post-order traversal(non recursive).

#include <iostream>

#include <stack>

#include <cctype>

using namespace std;

// Expression Tree Node

struct Node {

char data;

Node\* left;

Node\* right;

Node(char val) {

data = val;

left = right = nullptr;

}

};

// Step 1: Construct Expression Tree from Postfix

Node\* constructTree(const string& postfix) {

stack<Node\*> st;

for (char ch : postfix) {

if (isalnum(ch)) {

// Operand → create node and push

st.push(new Node(ch));

} else {

// Operator → pop two nodes, make them children

Node\* node = new Node(ch);

node->right = st.top(); st.pop();

node->left = st.top(); st.pop();

st.push(node);

}

}

return st.top(); // Root of expression tree

}

// Step 2: Post-order Traversal (Non-recursive)

void postOrderNonRecursive(Node\* root) {

if (!root) return;

stack<Node\*> st1, st2;

st1.push(root);

while (!st1.empty()) {

Node\* curr = st1.top();

st1.pop();

st2.push(curr);

if (curr->left) st1.push(curr->left);

if (curr->right) st1.push(curr->right);

}

cout << "Post-order Traversal (non-recursive): ";

while (!st2.empty()) {

cout << st2.top()->data << " ";

st2.pop();

}

cout << endl;

}

// Driver Code

int main() {

// Infix: a - b \* c - d / e + f

// Corresponding Postfix: a b c \* - d e / - f +

string postfix = "abc\*-de/-f+";

Node\* root = constructTree(postfix);

postOrderNonRecursive(root);

return 0;

}

7.

#include <iostream>

using namespace std;

#define INF 99999 // Infinity value for no direct path

int getMinVertex(int dist[], bool visited[], int n) {

int minVertex = -1;

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

if (!visited[i] && (minVertex == -1 || dist[i] < dist[minVertex])) {

minVertex = i;

}

}

return minVertex;

}

void dijkstra(int\*\* graph, int n, int start) {

int dist[n]; // Stores shortest distance from start

bool visited[n]; // Visited array

int path[n]; // To store shortest path (parent)

// Initialization

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

dist[i] = INF;

visited[i] = false;

path[i] = -1;

}

dist[start] = 0; // Distance to source is 0

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

int u = getMinVertex(dist, visited, n);

visited[u] = true;

// Relax all neighbors

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

if (!visited[v] && graph[u][v] != INF && dist[u] + graph[u][v] < dist[v]) {

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

path[v] = u;

}

}

}

// Print result

cout << "\nShortest distances from vertex " << start << ":\n";

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

cout << "To vertex " << i << ": " << dist[i] << " via ";

if (i == start) cout << "start";

else {

int p = i;

while (path[p] != -1) {

cout << path[p] << " <- ";

p = path[p];

}

cout << start;

}

cout << endl;

}

}

int main() {

int n;

cout << "Enter the number of vertices: ";

cin >> n;

int\*\* graph = new int\*[n];

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

graph[i] = new int[n];

}

cout << "Enter the adjacency matrix (use -1 for no direct path):\n";

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

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

int input;

cout << "Enter edge weight from vertex " << i << " to vertex " << j << ": ";

cin >> input;

if (input == -1) input = INF;

graph[i][j] = input;

}

}

int startVertex;

cout << "Enter the starting vertex: ";

cin >> startVertex;

dijkstra(graph, n, startVertex);

// Clean up memory

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

delete[] graph[i];

}

delete[] graph;

return 0;

}

Write a program to find all the Hamiltonian cycles in a connected undirected graph G(V,E) using backtracking

#include <iostream>

#include <vector>

using namespace std;

#define MAX 20

int n; // Number of vertices

int graph[MAX][MAX]; // Adjacency matrix

vector<int> path; // Stores current path

bool visited[MAX]; // Marks visited vertices

// Function to check if current vertex can be added

bool isSafe(int v, int pos) {

// Check if vertex is adjacent to previous one

if (graph[path[pos - 1]][v] == 0)

return false;

// Check if already visited

if (visited[v])

return false;

return true;

}

// Recursive function to find Hamiltonian cycles

void hamiltonianCycle(int pos) {

if (pos == n) {

// Check if there is an edge from last to first vertex

if (graph[path[pos - 1]][path[0]] == 1) {

// Valid cycle

for (int v : path)

cout << v << " ";

cout << path[0] << " (cycle)" << endl;

}

return;

}

for (int v = 1; v < n; v++) {

if (isSafe(v, pos)) {

path.push\_back(v);

visited[v] = true;

hamiltonianCycle(pos + 1);

// Backtrack

visited[v] = false;

path.pop\_back();

}

}

}

int main() {

cout << "Enter number of vertices: ";

cin >> n;

cout << "Enter adjacency matrix:\n";

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

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

cin >> graph[i][j];

// Initialize path and visited

path.push\_back(0); // Start at vertex 0

visited[0] = true;

cout << "\nHamiltonian cycles in the graph are:\n";

hamiltonianCycle(1);

return 0;

}

9. Single source shortest path algorithm – Connected Components and finding a cycle in a graph

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

#define INF 1e9

// ---------- Single Source Shortest Path using Dijkstra ----------

void dijkstra(int n, vector<vector<pair<int, int>>>& adj, int src) {

vector<int> dist(n, INF);

dist[src] = 0;

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

pq.push({0, src});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

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

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

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

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

}

}

}

cout << "\nShortest distances from vertex " << src << ":\n";

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

cout << "To " << i << " -> " << dist[i] << endl;

}

// ---------- Connected Components using DFS ----------

void dfs\_cc(int u, vector<vector<int>>& graph, vector<bool>& visited) {

visited[u] = true;

cout << u << " ";

for (int v : graph[u]) {

if (!visited[v])

dfs\_cc(v, graph, visited);

}

}

void findConnectedComponents(int n, vector<vector<int>>& graph) {

vector<bool> visited(n, false);

int count = 0;

cout << "\nConnected Components:\n";

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

if (!visited[i]) {

count++;

cout << "Component " << count << ": ";

dfs\_cc(i, graph, visited);

cout << endl;

}

}

}

// ---------- Cycle Detection in Undirected Graph ----------

bool hasCycleUtil(int u, int parent, vector<vector<int>>& graph, vector<bool>& visited) {

visited[u] = true;

for (int v : graph[u]) {

if (!visited[v]) {

if (hasCycleUtil(v, u, graph, visited))

return true;

} else if (v != parent) {

return true; // Cycle found

}

}

return false;

}

bool containsCycle(int n, vector<vector<int>>& graph) {

vector<bool> visited(n, false);

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

if (!visited[i]) {

if (hasCycleUtil(i, -1, graph, visited))

return true;

}

}

return false;

}

// ---------- Main Function ----------

int main() {

int n, e;

cout << "Enter number of vertices: ";

cin >> n;

cout << "Enter number of edges: ";

cin >> e;

vector<vector<pair<int, int>>> weightedGraph(n);

vector<vector<int>> unweightedGraph(n);

cout << "Enter edges in format (u v weight):\n";

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

int u, v, w;

cin >> u >> v >> w;

weightedGraph[u].push\_back({v, w});

weightedGraph[v].push\_back({u, w}); // undirected graph

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

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

}

// 1. Dijkstra

dijkstra(n, weightedGraph, 0);

// 2. Connected Components

findConnectedComponents(n, unweightedGraph);

// 3. Cycle Detection

if (containsCycle(n, unweightedGraph))

cout << "\nGraph contains a cycle.\n";

else

cout << "\nGraph does NOT contain a cycle.\n";

return 0;

}

10. Implementation Max or Min Heap.

#include <iostream>

#include <vector>

using namespace std;

// ---------- Max Heap ----------

class MaxHeap {

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 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 insert(int value) {

heap.push\_back(value);

heapifyUp(heap.size() - 1);

}

void deleteMax() {

if (heap.empty()) return;

cout << "Deleted max: " << heap[0] << endl;

heap[0] = heap.back();

heap.pop\_back();

heapifyDown(0);

}

void print() {

cout << "Max Heap: ";

for (int val : heap)

cout << val << " ";

cout << endl;

}

};

// ---------- Min Heap ----------

class MinHeap {

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 smallest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if (left < heap.size() && heap[left] < heap[smallest]) smallest = left;

if (right < heap.size() && heap[right] < heap[smallest]) smallest = right;

if (smallest != index) {

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

heapifyDown(smallest);

}

}

public:

void insert(int value) {

heap.push\_back(value);

heapifyUp(heap.size() - 1);

}

void deleteMin() {

if (heap.empty()) return;

cout << "Deleted min: " << heap[0] << endl;

heap[0] = heap.back();

heap.pop\_back();

heapifyDown(0);

}

void print() {

cout << "Min Heap: ";

for (int val : heap)

cout << val << " ";

cout << endl;

}

};

// ---------- Main ----------

int main() {

MaxHeap maxHeap;

MinHeap minHeap;

// Insert some values

vector<int> data = {10, 4, 15, 20, 0, 6};

for (int val : data) {

maxHeap.insert(val);

minHeap.insert(val);

}

maxHeap.print();

minHeap.print();

maxHeap.deleteMax();

minHeap.deleteMin();

maxHeap.print();

minHeap.print();

return 0;

}

11. Implement word cloud program.

import matplotlib.pyplot as plt

from wordcloud import WordCloud

from collections import Counter

import nltk

nltk.download('stopwords')

from nltk.corpus import stopwords

# Function to preprocess the text (removing common words and punctuation)

def preprocess\_text(text):

stop\_words = set(stopwords.words('english'))

words = text.split()

cleaned\_words = [word.lower() for word in words if word.lower() not in stop\_words]

return ' '.join(cleaned\_words)

# Function to generate and display the word cloud

def generate\_wordcloud(text):

# Preprocess text (remove stopwords)

cleaned\_text = preprocess\_text(text)

# Create a WordCloud object and generate the word cloud from the text

wordcloud = WordCloud(width=800, height=400, background\_color='white').generate(cleaned\_text)

# Display the generated word cloud using matplotlib

plt.figure(figsize=(10, 5))

plt.imshow(wordcloud, interpolation='bilinear')

plt.axis('off') # No axes for word cloud

plt.show()

# Example text to generate a word cloud

text = """

Python is a powerful programming language. Python can be used for web development, data analysis,

machine learning, automation, and much more. It is a favorite language for many developers and data scientists.

"""

# Generate and display word cloud from the provided text

generate\_wordcloud(text)