DSA EXPERIMENTS

1.Write a C++ program for Stack Implementation Push, Pop and Display Operation.

->Aim:- que

**Algorithm:**

**Push Operation:**

1. Check if the stack is full (i.e., top == size - 1)
2. If full, display "Stack Overflow"
3. Else, increment top and add the element at stack[top]

**Pop Operation:**

1. Check if the stack is empty (i.e., top == -1)
2. If empty, display "Stack Underflow"
3. Else, remove and return the element at stack[top] and decrement top

**Display Operation:**

1. If top == -1, display "Stack is empty"
2. Else, print all elements from stack[top] to stack[0]

Program #include <iostream>

using namespace std;

#define SIZE 100

class Stack {

private:

int arr[SIZE];

int top;

public:

Stack() {

top = -1;

}

void push(int value) {

if (top == SIZE - 1) {

cout << "Stack Overflow\n";

} else {

arr[++top] = value;

cout << value << " pushed into stack.\n";

}

}

void pop() {

if (top == -1) {

cout << "Stack Underflow\n";

} else {

cout << arr[top--] << " popped from stack.\n";

}

}

void display() {

if (top == -1) {

cout << "Stack is empty\n";

} else {

cout << "Stack elements:\n";

for (int i = top; i >= 0; i--) {

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

}

cout << endl;

}

}

};

int main() {

Stack s;

int choice, value;

do {

cout << "\n1.Push\n2.Pop\n3.Display\n4.Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to push: ";

cin >> value;

s.push(value);

break;

case 2:

s.pop();

break;

case 3:

s.display();

break;

case 4:

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

break;

default:

cout << "Invalid choice!\n";

}

} while (choice != 4);

return 0;

}

**2.Write a C++ Program for Queue Implementation Enquene, Dequeue and Display Operation.**

**->Algorithm:**

**Enqueue Operation:**

1. **Check if the queue is full (rear == SIZE - 1)**
2. **If full, display "Queue Overflow"**
3. **Else, increment rear and insert element at queue[rear]**
4. **If front == -1, set front = 0**

**Dequeue Operation:**

1. **Check if the queue is empty (front == -1 or front > rear)**
2. **If empty, display "Queue Underflow"**
3. **Else, display the dequeued element and increment front**

**Display Operation:**

1. **Check if the queue is empty (front == -1 or front > rear)**
2. **If empty, display "Queue is empty"**
3. **Else, print elements from queue[front] to queue[rear]**

**CODE-**

**#include <iostream>**

**using namespace std;**

**#define SIZE 100**

**class Queue {**

**private:**

**int queue[SIZE];**

**int front, rear;**

**public:**

**Queue() {**

**front = rear = -1;**

**}**

**void enqueue(int value) {**

**if (rear == SIZE - 1) {**

**cout << "Queue Overflow\n";**

**return;**

**}**

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

**queue[++rear] = value;**

**cout << value << " enqueued to the queue.\n";**

**}**

**void dequeue() {**

**if (front == -1 || front > rear) {**

**cout << "Queue Underflow\n";**

**return;**

**}**

**cout << queue[front++] << " dequeued from the queue.\n";**

**if (front > rear) {**

**front = rear = -1;**

**}**

**}**

**void display() {**

**if (front == -1 || front > rear) {**

**cout << "Queue is empty\n";**

**return;**

**}**

**cout << "Queue elements: ";**

**for (int i = front; i <= rear; i++) {**

**cout << queue[i] << " ";**

**}**

**cout << endl;**

**}**

**};**

**int main() {**

**Queue q;**

**int choice, value;**

**do {**

**cout << "\n1.Enqueue\n2.Dequeue\n3.Display\n4.Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**cout << "Enter the value to enqueue: ";**

**cin >> value;**

**q.enqueue(value);**

**break;**

**case 2:**

**q.dequeue();**

**break;**

**case 3:**

**q.display();**

**break;**

**case 4:**

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

**break;**

**default:**

**cout << "Invalid choice!\n";**

**}**

**} while (choice != 4);**

**return 0;**

**}**

**3: Singly Linked List in C++ (Create, Insert, Delete, Display)**

**->2. Algorithm:**

**Create Operation:**

1. **Create a new node.**
2. **Allocate memory and input data.**
3. **Set next pointer of the new node to NULL.**
4. **If head is NULL, assign the new node to head.**
5. **Else, traverse to the last node and add the new node.**

**Insert Operation (at Position):**

1. **Create a new node with input data.**
2. **If inserting at position 1:**
   * **Point new node to current head.**
   * **Set head to new node.**
3. **Else:**
   * **Traverse to (position-1) node.**
   * **Point new node to its next.**
   * **Set (position-1) node’s next to new node.**

**Delete Operation (at Position):**

1. **If the list is empty, print "List is empty".**
2. **If position is 1:**
   * **Point temp to head.**
   * **Set head to head's next node.**
   * **Delete temp.**
3. **Else:**
   * **Traverse to (position-1) node.**
   * **Store its next node in a temp pointer.**
   * **Point (position-1)'s next to temp’s next.**
   * **Delete temp.**

**Display Operation:**

1. **Traverse the list from head to NULL.**
2. **Print each node's data.**

**#include <iostream>**

**using namespace std;**

**class Node {**

**public:**

**int data;**

**Node\* next;**

**};**

**class SinglyLinkedList {**

**private:**

**Node\* head;**

**public:**

**SinglyLinkedList() {**

**head = nullptr;**

**}**

**void create(int value) {**

**Node\* newNode = new Node();**

**newNode->data = value;**

**newNode->next = nullptr;**

**if (head == nullptr) {**

**head = newNode;**

**} else {**

**Node\* temp = head;**

**while (temp->next != nullptr)**

**temp = temp->next;**

**temp->next = newNode;**

**}**

**cout << value << " added to the list.\n";**

**}**

**void insert(int pos, int value) {**

**Node\* newNode = new Node();**

**newNode->data = value;**

**if (pos == 1) {**

**newNode->next = head;**

**head = newNode;**

**} else {**

**Node\* temp = head;**

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

**temp = temp->next;**

**}**

**if (temp == nullptr) {**

**cout << "Position out of bounds\n";**

**return;**

**}**

**newNode->next = temp->next;**

**temp->next = newNode;**

**}**

**cout << value << " inserted at position " << pos << ".\n";**

**}**

**void remove(int pos) {**

**if (head == nullptr) {**

**cout << "List is empty\n";**

**return;**

**}**

**Node\* temp;**

**if (pos == 1) {**

**temp = head;**

**head = head->next;**

**delete temp;**

**cout << "Node at position 1 deleted.\n";**

**} else {**

**Node\* prev = head;**

**for (int i = 1; prev != nullptr && i < pos - 1; i++) {**

**prev = prev->next;**

**}**

**if (prev == nullptr || prev->next == nullptr) {**

**cout << "Position out of bounds\n";**

**return;**

**}**

**temp = prev->next;**

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

**delete temp;**

**cout << "Node at position " << pos << " deleted.\n";**

**}**

**}**

**void display() {**

**if (head == nullptr) {**

**cout << "List is empty\n";**

**return;**

**}**

**Node\* temp = head;**

**cout << "Linked List: ";**

**while (temp != nullptr) {**

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

**temp = temp->next;**

**}**

**cout << "NULL\n";**

**}**

**};**

**int main() {**

**SinglyLinkedList list;**

**int choice, val, pos;**

**do {**

**cout << "\n1.Create\n2.Insert\n3.Delete\n4.Display\n5.Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**cout << "Enter value to add: ";**

**cin >> val;**

**list.create(val);**

**break;**

**case 2:**

**cout << "Enter position and value to insert: ";**

**cin >> pos >> val;**

**list.insert(pos, val);**

**break;**

**case 3:**

**cout << "Enter position to delete: ";**

**cin >> pos;**

**list.remove(pos);**

**break;**

**case 4:**

**list.display();**

**break;**

**case 5:**

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

**break;**

**default:**

**cout << "Invalid choice!\n";**

**}**

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

**return 0;**

**}**

**Bubble Sort Algorithm Implementation in C++**

**1. Aim:**

**To implement the Bubble Sort algorithm in C++ to sort an array of elements in ascending order.**

**2. Algorithm:**

**Bubble Sort Algorithm:**

1. **Start with the first element in the array.**
2. **Compare the current element with the next element.**
3. **If the current element is greater than the next element, swap them.**
4. **Repeat steps 2-3 for each pair of adjacent elements in the array.**
5. **After each pass, the largest unsorted element moves to its correct position (like a bubble rising to the top).**
6. **Repeat the process for (n-1) passes where n is the number of elements in the array.**
7. **After all passes, the array will be sorted.**

**Time Complexity:**

* **Best Case: O(n) [when the array is already sorted]**
* **Average Case: O(n²)**
* **Worst Case: O(n²)**

**3. Program:**

**cpp**

**CopyEdit**

**#include <iostream>**

**using namespace std;**

**class BubbleSort {**

**public:**

**void sort(int arr[], int n) {**

**bool swapped;**

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

**swapped = false;**

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

**if (arr[j] > arr[j + 1]) {**

**// Swap elements**

**int temp = arr[j];**

**arr[j] = arr[j + 1];**

**arr[j + 1] = temp;**

**swapped = true;**

**}**

**}**

**// If no elements were swapped, break early (optimization)**

**if (!swapped) break;**

**}**

**}**

**void display(int arr[], int n) {**

**cout << "Sorted Array: ";**

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

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

**}**

**cout << endl;**

**}**

**};**

**int main() {**

**BubbleSort sorter;**

**int n;**

**cout << "Enter the number of elements: ";**

**cin >> n;**

**int arr[n];**

**cout << "Enter " << n << " elements:\n";**

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

**cin >> arr[i];**

**}**

**sorter.sort(arr, n);**

**sorter.display(arr, n);**

**return 0;**

**}**

**: Singly Linked List – Create, Search, and Display**

**1. Aim:**

**To implement a Singly Linked List (SLL) in C++ that allows the creation of nodes, searching for a particular element, and displaying the entire list.**

**2. Algorithm:**

**Create Operation:**

1. **Input the value to be inserted.**
2. **Create a new node and assign the value to it.**
3. **If the list is empty (i.e., head is NULL), make the new node the head.**
4. **Otherwise, traverse the list to the last node.**
5. **Set the next pointer of the last node to the new node.**

**Search Operation:**

1. **Input the value to be searched.**
2. **Start from the head node.**
3. **Traverse each node and compare its data with the target value.**
4. **If found, print the position.**
5. **If the traversal completes without finding the value, print "Not found".**

**Display Operation:**

1. **If the list is empty, print "List is empty".**
2. **Start from the head and traverse the list.**
3. **Print the data of each node followed by an arrow (->).**
4. **Stop at the last node and print NULL.**

**3. Program:**

**cpp**

**CopyEdit**

**#include <iostream>**

**using namespace std;**

**class Node {**

**public:**

**int data;**

**Node\* next;**

**};**

**class SinglyLinkedList {**

**private:**

**Node\* head;**

**public:**

**SinglyLinkedList() {**

**head = nullptr;**

**}**

**void create(int value) {**

**Node\* newNode = new Node();**

**newNode->data = value;**

**newNode->next = nullptr;**

**if (head == nullptr) {**

**head = newNode;**

**} else {**

**Node\* temp = head;**

**while (temp->next != nullptr)**

**temp = temp->next;**

**temp->next = newNode;**

**}**

**cout << value << " added to the list.\n";**

**}**

**void search(int key) {**

**Node\* temp = head;**

**int position = 1;**

**bool found = false;**

**while (temp != nullptr) {**

**if (temp->data == key) {**

**cout << "Element " << key << " found at position " << position << ".\n";**

**found = true;**

**break;**

**}**

**temp = temp->next;**

**position++;**

**}**

**if (!found)**

**cout << "Element " << key << " not found in the list.\n";**

**}**

**void display() {**

**if (head == nullptr) {**

**cout << "List is empty.\n";**

**return;**

**}**

**Node\* temp = head;**

**cout << "Linked List: ";**

**while (temp != nullptr) {**

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

**temp = temp->next;**

**}**

**cout << "NULL\n";**

**}**

**};**

**int main() {**

**SinglyLinkedList list;**

**int choice, val;**

**do {**

**cout << "\n1. Create\n2. Search\n3. Display\n4. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**cout << "Enter value to insert: ";**

**cin >> val;**

**list.create(val);**

**break;**

**case 2:**

**cout << "Enter value to search: ";**

**cin >> val;**

**list.search(val);**

**break;**

**case 3:**

**list.display();**

**break;**

**case 4:**

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

**break;**

**default:**

**cout << "Invalid choice.\n";**

**}**

**} while (choice != 4);**

**return 0;**

**}**

**6: Tree Traversal – Inorder, Preorder, and Postorder in C++**

**1. Aim:**

To implement a binary tree in C++ and perform **Inorder**, **Preorder**, and **Postorder** traversal operations.

**2. Algorithm:**

**Tree Node Structure:**

* Each node contains:
  + Data
  + Pointer to left child
  + Pointer to right child

**Inorder Traversal (Left → Root → Right):**

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

**Preorder Traversal (Root → Left → Right):**

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.

**Postorder Traversal (Left → Right → Root):**

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.

**3. Program:**

cpp

CopyEdit

#include <iostream>

using namespace std;

// Definition of a binary tree node

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = right = nullptr;

}

};

// Class to handle tree creation and traversal

class BinaryTree {

public:

Node\* root;

BinaryTree() {

root = nullptr;

}

Node\* createNode(int data) {

return new Node(data);

}

void inorder(Node\* node) {

if (node == nullptr) return;

inorder(node->left);

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

inorder(node->right);

}

void preorder(Node\* node) {

if (node == nullptr) return;

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

preorder(node->left);

preorder(node->right);

}

void postorder(Node\* node) {

if (node == nullptr) return;

postorder(node->left);

postorder(node->right);

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

}

};

int main() {

BinaryTree tree;

// Manually constructing the tree:

// 1

// / \

// 2 3

// / \ / \

// 4 5 6 7

tree.root = tree.createNode(1);

tree.root->left = tree.createNode(2);

tree.root->right = tree.createNode(3);

tree.root->left->left = tree.createNode(4);

tree.root->left->right = tree.createNode(5);

tree.root->right->left = tree.createNode(6);

tree.root->right->right = tree.createNode(7);

cout << "Inorder Traversal: ";

tree.inorder(tree.root);

cout << "\n";

cout << "Preorder Traversal: ";

tree.preorder(tree.root);

cout << "\n";

cout << "Postorder Traversal: ";

tree.postorder(tree.root);

cout << "\n";

return 0;

}

**7: Linear Search and Binary Search in C++**

**1. Aim:**

To implement **Linear Search** and **Binary Search** algorithms in C++ to search for an element in an array.

**2. Algorithm:**

**🔍 Linear Search:**

1. Start from the first element of the array.
2. Compare each element with the target value.
3. If a match is found, return the index (or position).
4. If end of array is reached without a match, return -1 or "not found".

**Time Complexity:**

* Best Case: O(1)
* Worst Case: O(n)

**🔍 Binary Search:**

1. Array must be sorted.
2. Initialize two pointers: low = 0, high = n-1.
3. Find the middle index: mid = (low + high) / 2.
4. If arr[mid] == key, return mid.
5. If arr[mid] > key, search the left half.
6. If arr[mid] < key, search the right half.
7. Repeat until low > high.

**Time Complexity:**

* Best Case: O(1)
* Worst Case: O(log n)

**3. Program:**

cpp

CopyEdit

#include <iostream>

using namespace std;

class SearchAlgorithms {

public:

// Linear Search

void linearSearch(int arr[], int n, int key) {

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

if (arr[i] == key) {

cout << "Linear Search: Element found at position " << (i + 1) << ".\n";

return;

}

}

cout << "Linear Search: Element not found.\n";

}

// Binary Search

void binarySearch(int arr[], int n, int key) {

int low = 0, high = n - 1, mid;

while (low <= high) {

mid = (low + high) / 2;

if (arr[mid] == key) {

cout << "Binary Search: Element found at position " << (mid + 1) << ".\n";

return;

} else if (arr[mid] < key) {

low = mid + 1;

} else {

high = mid - 1;

}

}

cout << "Binary Search: Element not found.\n";

}

};

int main() {

SearchAlgorithms search;

int n, key;

cout << "Enter number of elements: ";

cin >> n;

int arr[n];

cout << "Enter " << n << " sorted elements:\n";

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

cin >> arr[i];

}

cout << "Enter the element to search: ";

cin >> key;

search.linearSearch(arr, n, key);

search.binarySearch(arr, n, key);

return 0;

}

**8: Adjacency Matrix Implementation in C++**

**1. Aim:**

To represent a **graph using an adjacency matrix** in C++ and display the matrix to understand the connections between vertices.

**2. Algorithm:**

**Adjacency Matrix Basics:**

* A 2D array is used to represent a graph.
* For a graph with **V** vertices, create a matrix of size V x V.
* If there's an edge between vertex i and vertex j, set matrix[i][j] = 1.
* If there's **no edge**, set matrix[i][j] = 0.
* For **undirected graphs**, the matrix is symmetric: matrix[i][j] = matrix[j][i].

**Steps:**

1. Input the number of vertices (V) and number of edges (E).
2. Initialize a V x V matrix with all zeros.
3. For each edge, input the source and destination.
4. Set matrix[src][dest] = 1.
5. If undirected, also set matrix[dest][src] = 1.
6. Display the matrix.

**3. Program:**

cpp

CopyEdit

#include <iostream>

using namespace std;

class Graph {

private:

int vertices;

int adjMatrix[20][20]; // Assuming max 20 vertices

public:

Graph(int v) {

vertices = v;

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

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

adjMatrix[i][j] = 0;

}

void addEdge(int src, int dest, bool isDirected = false) {

adjMatrix[src][dest] = 1;

if (!isDirected) {

adjMatrix[dest][src] = 1;

}

}

void display() {

cout << "\nAdjacency Matrix:\n";

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

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

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

}

cout << "\n";

}

}

};

int main() {

int v, e;

cout << "Enter number of vertices: ";

cin >> v;

Graph g(v);

cout << "Enter number of edges: ";

cin >> e;

cout << "Enter edges (source and destination indices starting from 0):\n";

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

int src, dest;

cin >> src >> dest;

g.addEdge(src, dest); // Undirected by default

}

g.display();

return 0;

}

**9: Dijkstra's Shortest Path Algorithm in C++**

**1. Aim:**

To implement **Dijkstra’s Shortest Path Algorithm** in C++ to find the shortest path between a source node and all other nodes in a weighted graph.

**2. Algorithm:**

**Dijkstra’s Algorithm:**

1. **Initialization:**
   * Mark all distances as infinite (INF), except for the source node which has a distance of 0.
   * Create a priority queue (min-heap) to store the current node with its distance from the source.
2. **Steps:**
   * Choose the node with the smallest distance from the priority queue.
   * For each neighboring node, calculate the potential shorter path distance.
   * If a shorter path is found, update the distance and add the neighbor to the priority queue.
   * Repeat until all nodes have been visited.
3. **Termination:**
   * Once all nodes have the shortest distance computed, the algorithm terminates.

**Time Complexity:** O(V^2) for an adjacency matrix, where V is the number of vertices. Using a priority queue (min-heap), this can be optimized to O((V + E) log V), where E is the number of edges.

**3. Program:**

cpp

CopyEdit

#include <iostream>

#include <climits>

#include <vector>

#include <queue>

using namespace std;

class Graph {

private:

int V; // Number of vertices

vector<pair<int, int>> \*adjList; // Adjacency list (vertex, weight)

public:

Graph(int v) {

V = v;

adjList = new vector<pair<int, int>>[V];

}

// Add an edge to the graph

void addEdge(int src, int dest, int weight) {

adjList[src].push\_back(make\_pair(dest, weight));

adjList[dest].push\_back(make\_pair(src, weight)); // For undirected graph

}

// Dijkstra's Algorithm to find the shortest path

void dijkstra(int source) {

// Priority queue to store (distance, vertex)

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

// Vector to store the shortest distance to each vertex

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

dist[source] = 0;

// Push the source node into the priority queue

pq.push(make\_pair(0, source));

while (!pq.empty()) {

// Get the vertex with the minimum distance

int u = pq.top().second;

pq.pop();

// Traverse through all adjacent vertices

for (auto &neighbor : adjList[u]) {

int v = neighbor.first;

int weight = neighbor.second;

// If a shorter path is found

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

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

pq.push(make\_pair(dist[v], v));

}

}

}

// Display the shortest distances

cout << "Shortest distances from source vertex " << source << ":\n";

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

if (dist[i] == INT\_MAX) {

cout << "Vertex " << i << ": No path\n";

} else {

cout << "Vertex " << i << ": " << dist[i] << "\n";

}

}

}

};

int main() {

int V, E;

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

cin >> V;

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

cin >> E;

Graph g(V);

cout << "Enter " << E << " edges (source, destination, weight):\n";

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

int src, dest, weight;

cin >> src >> dest >> weight;

g.addEdge(src, dest, weight);

}

int source;

cout << "Enter the source vertex: ";

cin >> source;

g.dijkstra(source);

return 0;

}

**10: Hash Table Implementation in C++**

**1. Aim:**

To implement a **Hash Table** in C++ and demonstrate operations like insertion, deletion, and searching.

**2. Algorithm:**

**Hash Table Operations:**

1. **Insertion:**
   * Calculate the hash index for the key using a hash function.
   * If the index is empty, insert the key-value pair.
   * If there is a collision (i.e., the index is already occupied), handle it using **separate chaining** (linked list).
2. **Search:**
   * Calculate the hash index for the key.
   * If the key is found at the index, return the value.
   * If there is a collision, traverse the linked list at that index.
3. **Deletion:**
   * Calculate the hash index for the key.
   * If the key is found at the index, remove it from the linked list.

**Steps:**

1. Define a hash table with a fixed size.
2. Implement the hash function to map keys to indices.
3. Implement operations: **Insert**, **Search**, and **Delete**.
4. Use **separate chaining** to resolve collisions (using linked lists at each index).

**Hash Function:**

* A simple hash function: index = key % table\_size.

**3. Program:**

cpp

CopyEdit

#include <iostream>

#include <list>

using namespace std;

class HashTable {

private:

int size;

list<pair<int, string>> \*table;

public:

// Constructor to initialize the table with a given size

HashTable(int s) {

size = s;

table = new list<pair<int, string>>[size];

}

// Hash function to map values to key

int hashFunction(int key) {

return key % size;

}

// Insert key-value pair into the hash table

void insert(int key, string value) {

int index = hashFunction(key);

table[index].push\_back(make\_pair(key, value)); // Adding to the linked list

cout << "Inserted (" << key << ", " << value << ") at index " << index << endl;

}

// Search for a key in the hash table

void search(int key) {

int index = hashFunction(key);

bool found = false;

for (auto &x : table[index]) {

if (x.first == key) {

cout << "Found (" << key << ", " << x.second << ") at index " << index << endl;

found = true;

break;

}

}

if (!found) {

cout << "Key " << key << " not found.\n";

}

}

// Delete a key-value pair from the hash table

void remove(int key) {

int index = hashFunction(key);

bool found = false;

for (auto it = table[index].begin(); it != table[index].end(); ++it) {

if (it->first == key) {

table[index].erase(it);

cout << "Deleted key " << key << " from index " << index << endl;

found = true;

break;

}

}

if (!found) {

cout << "Key " << key << " not found.\n";

}

}

// Display the hash table

void display() {

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

cout << "Index " << i << ": ";

for (auto &x : table[i]) {

cout << "(" << x.first << ", " << x.second << ") ";

}

cout << endl;

}

}

};

int main() {

HashTable ht(7); // Create a hash table with 7 buckets

// Insert some key-value pairs

ht.insert(10, "Apple");

ht.insert(20, "Banana");

ht.insert(30, "Cherry");

ht.insert(17, "Date");

// Display the hash table

ht.display();

// Search for keys

ht.search(10);

ht.search(25);

// Remove a key

ht.remove(20);

// Display the hash table after deletion

ht.display();

return 0;

}