**Practical 5**

**Group B 11**

**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>

#include <algorithm>

using namespace std;

// Structure for a node in the binary search tree

struct Node {

string keyword;

string meaning;

Node\* left;

Node\* right;

Node(string k, string m) {

keyword = k;

meaning = m;

left = right = nullptr;

}

};

// Class for Dictionary (BST implementation)

class Dictionary {

private:

Node\* root;

// Helper function to insert a node into the BST

Node\* insert(Node\* node, const string& keyword, const string& meaning) {

if (node == nullptr) {

return new Node(keyword, meaning);

}

if (keyword < node->keyword) {

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

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

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

}

return node;

}

// Helper function to delete a node from the BST

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

if (root == nullptr) return root;

if (keyword < root->keyword) {

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

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

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

} else {

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

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

root->keyword = temp->keyword;

root->meaning = temp->meaning;

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

}

return root;

}

// Helper function to find the minimum node in a tree

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current && current->left != nullptr) {

current = current->left;

}

return current;

}

// Helper function for updating a keyword's meaning

Node\* update(Node\* node, const string& keyword, const string& newMeaning) {

if (node == nullptr) return node;

if (keyword == node->keyword) {

node->meaning = newMeaning;

} else if (keyword < node->keyword) {

node->left = update(node->left, keyword, newMeaning);

} else {

node->right = update(node->right, keyword, newMeaning);

}

return node;

}

// Helper function to display the dictionary in ascending order (inorder traversal)

void inorder(Node\* node) {

if (node == nullptr) return;

inorder(node->left);

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

inorder(node->right);

}

// Helper function to display the dictionary in descending order (reverse inorder traversal)

void reverseInorder(Node\* node) {

if (node == nullptr) return;

reverseInorder(node->right);

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

reverseInorder(node->left);

}

// Helper function to calculate the maximum depth of the tree (for maximum comparisons in binary search)

int maxDepth(Node\* node) {

if (node == nullptr) return 0;

int leftDepth = maxDepth(node->left);

int rightDepth = maxDepth(node->right);

return max(leftDepth, rightDepth) + 1;

}

public:

Dictionary() {

root = nullptr;

}

// Function to insert a new keyword and its meaning

void insert(const string& keyword, const string& meaning) {

root = insert(root, keyword, meaning);

}

// Function to delete a keyword

void deleteKeyword(const string& keyword) {

root = deleteNode(root, keyword);

}

// Function to update the meaning of a keyword

void update(const string& keyword, const string& newMeaning) {

root = update(root, keyword, newMeaning);

}

// Function to display the dictionary in ascending order

void displayAscending() {

inorder(root);

}

// Function to display the dictionary in descending order

void displayDescending() {

reverseInorder(root);

}

// Function to find the maximum number of comparisons required to find a keyword

int maxComparisons() {

return maxDepth(root);

}

};

// Main function

int main() {

Dictionary dict;

while (true) {

int choice;

cout << "\nDictionary Menu: \n";

cout << "1. Add new keyword\n";

cout << "2. Delete keyword\n";

cout << "3. Update keyword meaning\n";

cout << "4. Display dictionary (ascending order)\n";

cout << "5. Display dictionary (descending order)\n";

cout << "6. Find max comparisons needed\n";

cout << "7. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

cin.ignore();

if (choice == 1) {

string keyword, meaning;

cout << "Enter keyword: ";

getline(cin, keyword);

cout << "Enter meaning: ";

getline(cin, meaning);

dict.insert(keyword, meaning);

} else if (choice == 2) {

string keyword;

cout << "Enter keyword to delete: ";

getline(cin, keyword);

dict.deleteKeyword(keyword);

} else if (choice == 3) {

string keyword, newMeaning;

cout << "Enter keyword to update: ";

getline(cin, keyword);

cout << "Enter new meaning: ";

getline(cin, newMeaning);

dict.update(keyword, newMeaning);

} else if (choice == 4) {

cout << "Dictionary in ascending order:\n";

dict.displayAscending();

} else if (choice == 5) {

cout << "Dictionary in descending order:\n";

dict.displayDescending();

} else if (choice == 6) {

cout << "Maximum comparisons required to find a keyword: " << dict.maxComparisons() << endl;

} else if (choice == 7) {

break;

} else {

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

}

}

return 0;

}

**Practical 3**

**Group B 06**

**Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree - i. Insert new node ii. Find number of nodes in longest path from root iii. Minimum data value found in the tree iv. Change a tree so that the roles of the left and right pointers are swapped at every node v. Search a value**

**#include <iostream>**

**#include <algorithm>**

**using namespace std;**

**// Structure for a node in the binary search tree**

**struct Node {**

**int data;**

**Node\* left;**

**Node\* right;**

**// Constructor**

**Node(int val) {**

**data = val;**

**left = right = nullptr;**

**}**

**};**

**// Class for Binary Search Tree (BST) implementation**

**class BinarySearchTree {**

**private:**

**Node\* root;**

**// Helper function to insert a node into the BST**

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

**if (node == nullptr) {**

**return new Node(val);**

**}**

**if (val < node->data) {**

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

**} else {**

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

**}**

**return node;**

**}**

**// Helper function to find the height (number of nodes in the longest path) of the tree**

**int height(Node\* node) {**

**if (node == nullptr) return 0;**

**int leftHeight = height(node->left);**

**int rightHeight = height(node->right);**

**return max(leftHeight, rightHeight) + 1;**

**}**

**// Helper function to find the minimum data value in the tree**

**Node\* findMin(Node\* node) {**

**while (node && node->left != nullptr) {**

**node = node->left;**

**}**

**return node;**

**}**

**// Helper function to perform a search for a value**

**bool search(Node\* node, int val) {**

**if (node == nullptr) return false;**

**if (val == node->data) return true;**

**if (val < node->data) return search(node->left, val);**

**return search(node->right, val);**

**}**

**// Helper function to swap left and right subtrees at every node**

**void swapSubtrees(Node\* node) {**

**if (node == nullptr) return;**

**swap(node->left, node->right); // Swap left and right pointers**

**swapSubtrees(node->left); // Recursively swap left and right subtrees**

**swapSubtrees(node->right); // Recursively swap left and right subtrees**

**}**

**public:**

**// Constructor**

**BinarySearchTree() {**

**root = nullptr;**

**}**

**// Function to insert a new node with the given value**

**void insert(int val) {**

**root = insert(root, val);**

**}**

**// Function to find the height of the tree (longest path from root)**

**int findHeight() {**

**return height(root);**

**}**

**// Function to find the minimum data value in the tree**

**int findMinValue() {**

**Node\* minNode = findMin(root);**

**if (minNode != nullptr) return minNode->data;**

**return -1; // If tree is empty, return -1**

**}**

**// Function to swap the left and right subtrees at every node**

**void swapTree() {**

**swapSubtrees(root);**

**}**

**// Function to search for a specific value in the tree**

**bool searchValue(int val) {**

**return search(root, val);**

**}**

**// Function to display the tree (in-order traversal)**

**void inorder(Node\* node) {**

**if (node == nullptr) return;**

**inorder(node->left);**

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

**inorder(node->right);**

**}**

**// Wrapper function for inorder traversal (starting from root)**

**void inorderTraversal() {**

**inorder(root);**

**cout << endl;**

**}**

**};**

**// Main function**

**int main() {**

**BinarySearchTree bst;**

**// Insert values in the given order**

**cout << "Inserting values into the BST: 50, 30, 20, 40, 70, 60, 80\n";**

**bst.insert(50);**

**bst.insert(30);**

**bst.insert(20);**

**bst.insert(40);**

**bst.insert(70);**

**bst.insert(60);**

**bst.insert(80);**

**cout << "In-order traversal of the tree: ";**

**bst.inorderTraversal();**

**// i. Insert a new node**

**int newValue;**

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

**cin >> newValue;**

**bst.insert(newValue);**

**cout << "In-order traversal after insertion: ";**

**bst.inorderTraversal();**

**// ii. Find number of nodes in the longest path (height of the tree)**

**cout << "The number of nodes in the longest path (height) from the root: " << bst.findHeight() << endl;**

**// iii. Find the minimum data value found in the tree**

**cout << "The minimum data value in the tree: " << bst.findMinValue() << endl;**

**// v. Search a value**

**int searchValueToFind;**

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

**cin >> searchValueToFind;**

**if (bst.searchValue(searchValueToFind)) {**

**cout << "Value " << searchValueToFind << " is found in the tree." << endl;**

**} else {**

**cout << "Value " << searchValueToFind << " is not found in the tree." << endl;**

**}**

**// iv. Swap the left and right children at every node**

**bst.swapTree();**

**cout << "In-order traversal after swapping left and right subtrees: ";**

**bst.inorderTraversal();**

**return 0;**

**}**

**Practical 4**

**Group B 10**

**Consider threading a binary tree using preorder threads rather than inorder threads. Design an algorithm for traversal without using stack and analyze its complexity.**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

bool isLeftThreaded;

bool isRightThreaded;

Node(int val) {

data = val;

left = right = nullptr;

isLeftThreaded = false;

isRightThreaded = false;

}

};

class ThreadedBinaryTree {

private:

Node\* root;

void threadTree(Node\* node, Node\*& prev) {

if (node == nullptr) return;

if (node->left == nullptr) {

node->left = prev;

node->isLeftThreaded = true;

}

if (prev != nullptr && prev->right == nullptr) {

prev->right = node;

prev->isRightThreaded = true;

}

prev = node;

if (!node->isLeftThreaded) {

threadTree(node->left, prev);

}

if (!node->isRightThreaded) {

threadTree(node->right, prev);

}

}

void preorderTraversal(Node\* node) {

while (node != nullptr) {

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

if (node->left && !node->isLeftThreaded) {

node = node->left;

} else {

node = node->right;

while (node && node->isLeftThreaded) {

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

node = node->right;

}

if (node ==nullptr) {

break;

}

}

}

}

public:

ThreadedBinaryTree() {

root = nullptr;

}

void insert(int data) {

Node\* newNode = new Node(data);

if (root == nullptr) {

root = newNode;

} else {

Node\* current = root;

Node\* parent = nullptr;

while (current != nullptr) {

parent = current;

if (data < current->data) {

current = current->left;

} else {

current = current->right;

}

}

if (data < parent->data) {

parent->left = newNode;

} else {

parent->right = newNode;

}

}

}

void threadTree() {

Node\* prev = nullptr;

threadTree(root, prev);

}

void preorderTraversal() {

cout << "Preorder Traversal (without stack) using threads: ";

preorderTraversal(root);

cout << endl;

}

};

int main() {

ThreadedBinaryTree tree;

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

tree.threadTree();

tree.preorderTraversal();

return 0;

}

**Practical 1**

**Group A 01**

**Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers**

#include <iostream>

#include <list>

#include <vector>

#include <string>

using namespace std;

// Maximum size of the hash table

#define TABLE\_SIZE 10

// Struct for a client (name and telephone number)

struct Client {

string name;

string phone;

};

// Hash function to map a string (client name) to an index in the hash table

int hashFunction(const string& key) {

int hashValue = 0;

for (char c : key) {

hashValue += c; // Adding ASCII values of the characters

}

return hashValue % TABLE\_SIZE; // Ensure index stays within the table size

}

// Separate Chaining Hash Table

class SeparateChainingHashTable {

private:

list<Client>\* table;

int totalComparisons;

public:

SeparateChainingHashTable() {

table = new list<Client>[TABLE\_SIZE];

totalComparisons = 0;

}

void insert(const string& name, const string& phone) {

int index = hashFunction(name);

table[index].push\_back({name, phone});

}

string search(const string& name) {

int index = hashFunction(name);

totalComparisons = 0; // Reset comparison counter

// Traverse the list at the hashed index and compare

for (Client& client : table[index]) {

totalComparisons++;

if (client.name == name) {

return client.phone;

}

}

return "Not Found";

}

int getComparisons() {

return totalComparisons;

}

~SeparateChainingHashTable() {

delete[] table;

}

};

// Linear Probing Hash Table

class LinearProbingHashTable {

private:

Client\* table;

int\* status; // Array to track the status of each slot (0: empty, 1: filled)

int totalComparisons;

public:

LinearProbingHashTable() {

table = new Client[TABLE\_SIZE];

status = new int[TABLE\_SIZE]{0}; // Initialize all to 0 (empty)

totalComparisons = 0;

}

void insert(const string& name, const string& phone) {

int index = hashFunction(name);

int originalIndex = index;

// Linear probing to find an empty slot

while (status[index] == 1) { // If slot is filled, move to next

index = (index + 1) % TABLE\_SIZE;

if (index == originalIndex) return; // Avoid infinite loop

}

// Insert the client into the empty slot

table[index] = {name, phone};

status[index] = 1;

}

string search(const string& name) {

int index = hashFunction(name);

int originalIndex = index;

totalComparisons = 0; // Reset comparison counter

while (status[index] != 0) { // While slot is not empty

totalComparisons++;

if (status[index] == 1 && table[index].name == name) {

return table[index].phone;

}

index = (index + 1) % TABLE\_SIZE;

if (index == originalIndex) break; // Avoid infinite loop

}

return "Not Found";

}

int getComparisons() {

return totalComparisons;

}

~LinearProbingHashTable() {

delete[] table;

delete[] status;

}

};

int main() {

// Sample telephone book database

string names[] = {"Alice", "Bob", "Charlie", "David", "Eve"};

string phones[] = {"1234", "5678", "91011", "1213", "1415"};

// Create hash tables for separate chaining and linear probing

SeparateChainingHashTable sct;

LinearProbingHashTable lpt;

// Insert clients into both hash tables

for (int i = 0; i < 5; ++i) {

sct.insert(names[i], phones[i]);

lpt.insert(names[i], phones[i]);

}

// Test search and compare number of comparisons

string searchName = "Charlie";

cout << "Search using Separate Chaining:" << endl;

string phone = sct.search(searchName);

cout << "Phone: " << phone << endl;

cout << "Comparisons made: " << sct.getComparisons() << endl;

cout << "\nSearch using Linear Probing:" << endl;

phone = lpt.search(searchName);

cout << "Phone: " << phone << endl;

cout << "Comparisons made: " << lpt.getComparisons() << endl;

return 0;

}

**Practical 2**

**Group A 02**

**Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique Standard Operations: Insert(key, value), Find(key), Delete(key)**

class Node:

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

self.next = None

class HashTable:

def \_\_init\_\_(self, capacity=10, replacement=False):

self.capacity = capacity

self.table = [None] \* capacity

self.size = 0

self.replacement = replacement

def \_hash(self, key):

return hash(key) % self.capacity

def insert(self, key, value):

index = self.\_hash(key)

new\_node = Node(key, value)

if self.table[index] is None:

self.table[index] = new\_node

self.size += 1

return

current = self.table[index]

prev = None

while current:

if current.key == key:

if self.replacement:

current.value = value

return

else:

raise KeyError(f"Key '{key}' already exists.")

prev = current

current = current.next

prev.next = new\_node

self.size += 1

def find(self, key):

index = self.\_hash(key)

current = self.table[index]

while current:

if current.key == key:

return current.value

current = current.next

return None

def delete(self, key):

index = self.\_hash(key)

current = self.table[index]

prev = None

while current:

if current.key == key:

if prev:

prev.next = current.next

else:

self.table[index] = current.next

self.size -= 1

return

prev = current

current = current.next

raise KeyError(f"Key '{key}' not found.")

def \_\_len\_\_(self):

return self.size

def \_\_str\_\_(self):

result = "{"

for i in range(self.capacity):

current = self.table[i]

while current:

result += f"'{current.key}': '{current.value}', "

current = current.next

if self.size > 0:

result = result[:-2]

result += "}"

return result

# Example Usage with Replacement

ht\_replace = HashTable(capacity=5, replacement=True)

ht\_replace.insert("apple", 1)

ht\_replace.insert("banana", 2)

ht\_replace.insert("cherry", 3)

ht\_replace.insert("date", 4)

ht\_replace.insert("elderberry", 5)

print("HashTable with replacement:", ht\_replace)

print("Find 'banana':", ht\_replace.find("banana"))

ht\_replace.insert("banana", 22) # Replace existing value

print("HashTable with replacement after banana replacement:", ht\_replace)

ht\_replace.delete("cherry")

print("HashTable with replacement after cherry deletion:", ht\_replace)

# Example Usage without Replacement

ht\_no\_replace = HashTable(capacity=5, replacement=False)

ht\_no\_replace.insert("apple", 1)

ht\_no\_replace.insert("banana", 2)

ht\_no\_replace.insert("cherry", 3)

ht\_no\_replace.insert("date", 4)

ht\_no\_replace.insert("elderberry", 5)

print("\nHashTable without replacement:", ht\_no\_replace)

print("Find 'date':", ht\_no\_replace.find("date"))

try:

ht\_no\_replace.insert("banana", 22)

except KeyError as e:

print(e)

ht\_no\_replace.delete("cherry")

print("HashTable without replacement after cherry deletion:", ht\_no\_replace)

**Practical 6**

**Group C 14**

**There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.**

class Graph:

def \_\_init\_\_(self, directed=False, use\_matrix=False):

self.directed = directed

self.use\_matrix = use\_matrix

if use\_matrix:

self.adj\_matrix = {} # Dictionary of dictionaries for matrix representation

else:

self.adj\_list = {} # Dictionary for adjacency list

self.nodes = set()

def add\_edge(self, u, v, weight=1):

self.nodes.add(u)

self.nodes.add(v)

if self.use\_matrix:

if u not in self.adj\_matrix:

self.adj\_matrix[u] = {}

self.adj\_matrix[u][v] = weight

if not self.directed:

if v not in self.adj\_matrix:

self.adj\_matrix[v] = {}

self.adj\_matrix[v][u] = weight

else:

if u not in self.adj\_list:

self.adj\_list[u] = []

self.adj\_list[u].append((v, weight))

if not self.directed:

if v not in self.adj\_list:

self.adj\_list[v] = []

self.adj\_list[v].append((u, weight))

def is\_connected(self):

if not self.nodes:

return True # Empty graph is considered connected

start\_node = next(iter(self.nodes)) # Choose any node as the starting point

visited = set()

self.\_dfs(start\_node, visited)

return len(visited) == len(self.nodes)

def \_dfs(self, node, visited):

visited.add(node)

if self.use\_matrix:

if node in self.adj\_matrix:

for neighbor in self.adj\_matrix[node]:

if neighbor not in visited:

self.\_dfs(neighbor, visited)

else:

if node in self.adj\_list:

for neighbor, \_ in self.adj\_list[node]:

if neighbor not in visited:

self.\_dfs(neighbor, visited)

def \_\_str\_\_(self):

if self.use\_matrix:

result = "Adjacency Matrix:\n"

for u in self.nodes:

result += f"{u}: {self.adj\_matrix.get(u, {})} \n"

return result

else:

result = "Adjacency List:\n"

for u in self.nodes:

result += f"{u}: {self.adj\_list.get(u, [])}\n"

return result

# Example Usage (Adjacency List)

graph\_list = Graph(directed=False, use\_matrix=False)

graph\_list.add\_edge("Mumbai", "Delhi", 140) # 140 minutes flight time

graph\_list.add\_edge("Delhi", "Bangalore", 180)

graph\_list.add\_edge("Bangalore", "Chennai", 60)

graph\_list.add\_edge("Chennai", "Mumbai", 100)

graph\_list.add\_edge("Kolkata","Delhi", 120)

print(graph\_list)

print("Is graph connected (list):", graph\_list.is\_connected())

graph\_list2 = Graph(directed=False, use\_matrix=False)

graph\_list2.add\_edge("Mumbai", "Delhi", 140)

graph\_list2.add\_edge("Bangalore", "Chennai", 60)

print("Is graph connected (list):", graph\_list2.is\_connected())

# Example Usage (Adjacency Matrix)

graph\_matrix = Graph(directed=False, use\_matrix=True)

graph\_matrix.add\_edge("Mumbai", "Delhi", 140)

graph\_matrix.add\_edge("Delhi", "Bangalore", 180)

graph\_matrix.add\_edge("Bangalore", "Chennai", 60)

graph\_matrix.add\_edge("Chennai", "Mumbai", 100)

graph\_matrix.add\_edge("Kolkata","Delhi", 120)

print(graph\_matrix)

print("Is graph connected (matrix):", graph\_matrix.is\_connected())

graph\_matrix2 = Graph(directed=False, use\_matrix=True)

graph\_matrix2.add\_edge("Mumbai", "Delhi", 140)

graph\_matrix2.add\_edge("Bangalore", "Chennai", 60)

print("Is graph connected (matrix):", graph\_matrix2.is\_connected())

**Practical 7**

**Group C 15**

**You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures**.

import heapq

def prim\_mst(graph):

start\_node = list(graph.keys())[0]

visited = {start\_node}

edges = []

min\_spanning\_tree = []

total\_cost = 0

for neighbor, cost in graph[start\_node]:

heapq.heappush(edges, (cost, start\_node, neighbor))

while edges:

cost, u, v = heapq.heappop(edges)

if v not in visited:

visited.add(v)

min\_spanning\_tree.append((u, v, cost))

total\_cost += cost

for neighbor, neighbor\_cost in graph[v]:

if neighbor not in visited:

heapq.heappush(edges, (neighbor\_cost, v, neighbor))

return min\_spanning\_tree, total\_cost

# Example Usage

graph = {

"Office A": [("Office B", 10), ("Office C", 15)],

"Office B": [("Office A", 10), ("Office C", 20), ("Office D", 5)],

"Office C": [("Office A", 15), ("Office B", 20), ("Office D", 12)],

"Office D": [("Office B", 5), ("Office C", 12)],

}

mst, cost = prim\_mst(graph)

print("Minimum Spanning Tree:", mst)

print("Total Cost:", cost)

**Practical 8**

**Group D 18**

**Given sequence k = k1 <k2 < … <kn of n sorted keys, with a search probability pi for each**

**key ki . Build the Binary search tree that has the least search cost given the access**

**probability for each key?**

#include <iostream>

#include <limits.h>

using namespace std;

struct Node

{

int key;

Node \*left, \*right;

Node(int k)

{

key = k;

left = NULL;

right = NULL;

}

};

// function to find optimal root

int findOptimalRoot(int keys[], int freq[], int start, int end)

{

int sum = 0;

for (int i = start; i <= end; i++)

{

sum += freq[i];

}

int minCost = INT\_MAX;

int cost;

for (int i = start; i <= end; i++)

{

cost = sum + ((i > start) ? findOptimalRoot(keys, freq, start, i - 1) : 0) + ((i < end) ? findOptimalRoot(keys, freq, i + 1, end) : 0);

if (cost < minCost)

{

minCost = cost;

}

}

return minCost;

}

// function to construct and optimal bst

Node \*buildOptimalBST(int keys[], int freq[], int start, int end)

{

if (start > end)

{

return NULL;

}

int minIdx;

int minCost = INT\_MAX;

int cost;

for (int i = start; i <= end; i++)

{

cost = findOptimalRoot(keys, freq, start, end);

if (cost < minCost)

{

minCost = cost;

minIdx = i;

}

}

Node \*root = new Node(keys[minIdx]);

root->left = buildOptimalBST(keys, freq, start, minIdx - 1);

root->right = buildOptimalBST(keys, freq, minIdx + 1, end);

return root;

}

void inorder(Node \*root)

{

if (root == NULL)

{

return;

}

inorder(root->left);

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

inorder(root->right);

}

int main()

{

cout<<"DSAL Practical No. 08 (D-18)"<<endl;

int k;

cout << "\nEnter number of keys: ";

cin >> k;

cout << "\nEnter keys: ";

int keys[k];

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

{

cin >> keys[i];

}

cout << "\nEnter the frequencies of keys: ";

int freq[k];

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

{

cin >> freq[i];

}

int n = sizeof(keys) / sizeof(keys[0]);

Node \*root = buildOptimalBST(keys, freq, 0, n - 1);

cout << "\n Inorder traversal of the optimal binary search tree: ";

inorder(root);

return 0;

}

**Practical 9**

**Group D 19**

**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 Height balance tree and find the complexity for finding a keyword**

#include <iostream>

#include <map>

#include <vector>

#include <algorithm>

class AVLTreeDictionary {

private:

struct TreeNode {

std::string key;

std::string value;

TreeNode\* left;

TreeNode\* right;

int height;

TreeNode(std::string k, std::string v) : key(k), value(v), left(nullptr), right(nullptr), height(1) {}

};

TreeNode\* root;

int getHeight(TreeNode\* node) {

if (node == nullptr) {

return 0;

}

return node->height;

}

int getBalanceFactor(TreeNode\* node) {

if (node == nullptr) {

return 0;

}

return getHeight(node->left) - getHeight(node->right);

}

void updateHeight(TreeNode\* node) {

if (node != nullptr) {

node->height = 1 + std::max(getHeight(node->left), getHeight(node->right));

}

}

TreeNode\* rotateRight(TreeNode\* y) {

TreeNode\* x = y->left;

TreeNode\* t2 = x->right;

x->right = y;

y->left = t2;

updateHeight(y);

updateHeight(x);

return x;

}

TreeNode\* rotateLeft(TreeNode\* x) {

TreeNode\* y = x->right;

TreeNode\* t2 = y->left;

y->left = x;

x->right = t2;

updateHeight(x);

updateHeight(y);

return y;

}

TreeNode\* balance(TreeNode\* node) {

if (node == nullptr) {

return node;

}

updateHeight(node);

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1) {

if (getBalanceFactor(node->left) < 0) {

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

}

return rotateRight(node);

}

if (balanceFactor < -1) {

if (getBalanceFactor(node->right) > 0) {

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

}

return rotateLeft(node);

}

return node;

}

TreeNode\* insert(TreeNode\* node, std::string key, std::string value) {

if (node == nullptr) {

return new TreeNode(key, value);

}

if (key < node->key) {

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

} else if (key > node->key) {

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

} else {

node->value = value;

return node;

}

return balance(node);

}

TreeNode\* deleteNode(TreeNode\* node, std::string key) {

if (node == nullptr) {

return nullptr;

}

if (key < node->key) {

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

} else if (key > node->key) {

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

} else {

if (node->left == nullptr) {

TreeNode\* temp = node->right;

delete node;

return temp;

} else if (node->right == nullptr) {

TreeNode\* temp = node->left;

delete node;

return temp;

}

TreeNode\* temp = minValueNode(node->right);

node->key = temp->key;

node->value = temp->value;

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

}

return balance(node);

}

TreeNode\* minValueNode(TreeNode\* node) {

TreeNode\* current = node;

while (current->left != nullptr) {

current = current->left;

}

return current;

}

std::string findNode(TreeNode\* node, std::string key) {

if (node == nullptr) {

return "";

}

if (key < node->key) {

return findNode(node->left, key);

} else if (key > node->key) {

return findNode(node->right, key);

} else {

return node->value;

}

}

void inorderTraversal(TreeNode\* node, std::vector<std::pair<std::string, std::string>>& result) {

if (node != nullptr) {

inorderTraversal(node->left, result);

result.push\_back({node->key, node->value});

inorderTraversal(node->right, result);

}

}

void reverseInorderTraversal(TreeNode\* node, std::vector<std::pair<std::string, std::string>>& result) {

if (node != nullptr) {

reverseInorderTraversal(node->right, result);

result.push\_back({node->key, node->value});

reverseInorderTraversal(node->left, result);

}

}

public:

AVLTreeDictionary() : root(nullptr) {}

void insert(std::string key, std::string value) {

root = insert(root, key, value);

}

void remove(std::string key) {

root = deleteNode(root, key);

}

std::string find(std::string key) {

return findNode(root, key);

}

void displayAscending() {

std::vector<std::pair<std::string, std::string>> result;

inorderTraversal(root, result);

for (const auto& pair : result) {

std::cout << pair.first << ": " << pair.second << std::endl;

}

}

void displayDescending() {

std::vector<std::pair<std::string, std::string>> result;

reverseInorderTraversal(root, result);

for (const auto& pair : result) {

std::cout << pair.first << ": " << pair.second << std::endl;

}

}

int maxComparisons() {

return getHeight(root);

}

};

int main() {

AVLTreeDictionary dictionary;

dictionary.insert("apple", "A fruit");

dictionary.insert("banana", "Another fruit");

dictionary.insert("cherry", "A small red fruit");

dictionary.insert("date", "A sweet fruit");

std::cout << "Find 'banana': " << dictionary.find("banana") << std::endl;

dictionary.insert("banana", "A yellow fruit");

std::cout << "Ascending order:\n";

dictionary.displayAscending();

std::cout << "\nDescending order:\n";

dictionary.displayDescending();

dictionary.remove("cherry");

std::cout << "\nAscending order after delete:\n";

dictionary.displayAscending();

std::cout << "\nMaximum comparisons: " << dictionary.maxComparisons() << std::endl;

return 0;

}

**Practical 10**

**Group E 21**

**Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language**

#include <iostream>

#include <vector>

#include <algorithm>

// Heap Sort Module

namespace HeapSort {

void heapify(std::vector<int>& arr, int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

}

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

largest = right;

}

if (largest != i) {

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

heapify(arr, n, largest);

}

}

void sort(std::vector<int>& arr) {

int n = arr.size();

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

heapify(arr, n, i);

}

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

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

heapify(arr, i, 0);

}

}

}

// Shell Sort Module

namespace ShellSort {

void sort(std::vector<int>& arr) {

int n = arr.size();

for (int gap = n / 2; gap > 0; gap /= 2) {

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

int temp = arr[i];

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {

arr[j] = arr[j - gap];

}

arr[j] = temp;

}

}

}

}

// Utility Module

namespace Utility {

void printArray(const std::vector<int>& arr) {

for (int val : arr) {

std::cout << val << " ";

}

std::cout << std::endl;

}

}

int main() {

std::vector<int> arr1 = {12, 11, 13, 5, 6, 7};

std::vector<int> arr2 = {64, 34, 25, 12, 22, 11, 90};

std::cout << "Original Array for Heap Sort: ";

Utility::printArray(arr1);

HeapSort::sort(arr1);

std::cout << "Sorted Array (Heap Sort): ";

Utility::printArray(arr1);

std::cout << "\nOriginal Array for Shell Sort: ";

Utility::printArray(arr2);

ShellSort::sort(arr2);

std::cout << "Sorted Array (Shell Sort): ";

Utility::printArray(arr2);

return 0;

}

**Practical 11**

**Group F 23**

**Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.**

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

using namespace std;

struct Student {

int rollNumber;

string name;

string division;

string address;

};

// Function to add student record

void addStudent(const string& filename) {

ofstream outFile(filename, ios::app); // Open file in append mode

if (!outFile) {

cerr << "Error opening file!" << endl;

return;

}

Student student;

cout << "Enter Roll Number: ";

cin >> student.rollNumber;

cin.ignore(); // Ignore newline character left by cin

cout << "Enter Name: ";

getline(cin, student.name);

cout << "Enter Division: ";

getline(cin, student.division);

cout << "Enter Address: ";

getline(cin, student.address);

outFile << student.rollNumber << endl;

outFile << student.name << endl;

outFile << student.division << endl;

outFile << student.address << endl;

outFile.close();

cout << "Student added successfully!" << endl;

}

// Function to display student record by roll number

void displayStudent(const string& filename, int rollNumber) {

ifstream inFile(filename);

if (!inFile) {

cerr << "Error opening file!" << endl;

return;

}

Student student;

bool found = false;

while (inFile >> student.rollNumber) {

inFile.ignore(); // Ignore newline after roll number

getline(inFile, student.name);

getline(inFile, student.division);

getline(inFile, student.address);

if (student.rollNumber == rollNumber) {

cout << "Student found!" << endl;

cout << "Roll Number: " << student.rollNumber << endl;

cout << "Name: " << student.name << endl;

cout << "Division: " << student.division << endl;

cout << "Address: " << student.address << endl;

found = true;

break;

}

}

if (!found) {

cout << "Student record not found!" << endl;

}

inFile.close();

}

// Function to delete student record by roll number

void deleteStudent(const string& filename, int rollNumber) {

ifstream inFile(filename);

ofstream outFile("temp.txt");

if (!inFile || !outFile) {

cerr << "Error opening file!" << endl;

return;

}

Student student;

bool found = false;

while (inFile >> student.rollNumber) {

inFile.ignore(); // Ignore newline after roll number

getline(inFile, student.name);

getline(inFile, student.division);

getline(inFile, student.address);

if (student.rollNumber == rollNumber) {

found = true;

cout << "Student record deleted!" << endl;

} else {

outFile << student.rollNumber << endl;

outFile << student.name << endl;

outFile << student.division << endl;

outFile << student.address << endl;

}

}

if (!found) {

cout << "Student record not found!" << endl;

}

inFile.close();

outFile.close();

// Remove original file and rename temp file to original file name

remove(filename.c\_str());

rename("temp.txt", filename.c\_str());

}

// Main Menu function

void menu() {

string filename = "students.txt";

int choice, rollNumber;

do {

cout << "\nStudent Information System\n";

cout << "1. Add Student\n";

cout << "2. Display Student\n";

cout << "3. Delete Student\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

addStudent(filename);

break;

case 2:

cout << "Enter Roll Number to display: ";

cin >> rollNumber;

displayStudent(filename, rollNumber);

break;

case 3:

cout << "Enter Roll Number to delete: ";

cin >> rollNumber;

deleteStudent(filename, rollNumber);

break;

case 4:

cout << "Exiting the program." << endl;

break;

default:

cout << "Invalid choice, please try again." << endl;

}

} while (choice != 4);

}

int main() {

menu(); // Display the menu and handle user operations

return 0;

}

Output:

Student Information System

1. Add Student

2. Display Student

3. Delete Student

4. Exit

**Practical 12**

**Group F 23**

**Implementation of a direct access file -Insertion and deletion of a record from a direct access file**

/\*

Implementation of a direct access file -Insertion and deletion of a record from a Direct Access File.

\*/

#include <iostream>

#include <string>

#include <fstream>

using namespace std;

class Employee {

string name;

string address;

int empID;

int loc;

Employee() {

name = address = "";

loc = empID = 0;

}

bool isEmpty() {

return empID == 0;

}

void display() {

cout<<name<<" "<<address<<" "<<empID<<" "<<loc<<endl;

}

friend class Hashtable;

friend class MyFile;

};

class Hashtable {

Employee \*employees;

int tableSize;

int hash(int empID) {

return empID % tableSize;

}

public:

Hashtable() {

cout<<"Enter max number of employees: ";

cin>>tableSize;

employees = new Employee[tableSize];

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

employees[i] = Employee();

}

}

void insertHT( Employee emp ) {

int index = hash(emp.empID);

if(employees[index].isEmpty()) {

employees[index] = emp;

}

else {

int curr = index;

while(!employees[index].isEmpty())

curr = (curr+1)%tableSize;

employees[curr] = emp;

}

}

void showHT() {

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

cout.width(7);

cout<<employees[i].empID;

cout<<"\t";

cout.width(5);

cout<<employees[i].loc<<endl;

}

}

void deleteHT(int empID) {

int index = hash(empID);

if(employees[index].empID == empID)

employees[index] = Employee();

else {

int curr = index;

while( employees[index].empID != empID)

curr = (curr+1)%tableSize;

employees[curr] = Employee();

}

}

void nukeHT() {

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

employees[i] = Employee();

}

int searchHT(int empID) {

int index = hash(empID);

if(employees[index].empID == empID) {

return employees[index].loc;

}

else {

int curr = index, passes = 0;

while( employees[index].empID != empID and passes != tableSize) {

curr = (curr+1)%tableSize;

passes++;

}

if(passes != tableSize)

return employees[curr].loc;

else

return -1;

}

}

void updateEntry(int empID, int newLoc) {

int index = hash(empID);

if(employees[index].empID == empID)

employees[index].loc = newLoc;

else {

int curr = index;

while( employees[index].empID != empID )

curr = (curr+1)%tableSize;

employees[curr].loc = newLoc;

}

}

};

class MyFile {

char filename[20];

fstream file;

Hashtable table;

Employee E;

public:

MyFile() {

cout<<"Enter file name: ";

cin>>filename;

file.open(filename, ios::out);

if(file)

cout<<"File opened successfully"<<endl;

else

cout<<"Error in file opening"<<endl;

file.close();

}

void addRecord(string n, int id, string a) {

file.open(filename, ios::app);

if(file) {

E.name = n;

E.empID = id;

E.address = a;

cout<<"Location: "<<file.tellp()<<endl;

E.loc = file.tellp();

table.insertHT(E);

file.write(reinterpret\_cast<char\*>(&E), sizeof(E));

}

else {

cout<<"Error";

}

file.close();

}

void readRecord(int empID) {

file.open(filename, ios::in);

if(!file)

cout<<"Error opening file"<<endl;

else {

int loc = table.searchHT(empID);

if(loc == -1)

cout<<"No such record exists."<<endl;

else {

file.seekg(loc, ios::beg);

file.read(reinterpret\_cast<char\*>(&E), sizeof(E));

E.display();

}

}

file.close();

}

void deleteRecord(int empID) {

int loc = table.searchHT(empID);

if(loc == -1)

cout<<"No such record exists to delete"<<endl;

else {

fstream temp;

temp.open("temp", ios::out);

file.open(filename, ios::in);

while(!file.eof()) {

file.read(reinterpret\_cast<char\*>(&E), sizeof(E));

if(E.empID == empID) {

cout<<"Deleting Record"<<endl;

}

else {

E.loc = temp.tellp();

table.updateEntry(E.empID, E.loc);

temp.write(reinterpret\_cast<char\*>(&E), sizeof(E));

}

}

file.close(); temp.close();

remove(filename);

rename("temp", filename);

table.deleteHT(empID);

cout<<"\nRecord deleted!"<<endl;

}

}

};

int main() {

MyFile File;

File.addRecord("Durvesh", 21381, "Mumbai");

File.addRecord("Shantanu", 21382, "Pune");

File.addRecord("Riya", 21383, "Nasik");

File.addRecord("Atharva", 21384, "Baner");

File.addRecord("Aarti", 21385, "Balewadi");

File.addRecord("Riddhi", 21467, "Hadapsar");

File.readRecord(21383);

File.deleteRecord(21384);

File.readRecord(21467);

File.readRecord(21384);

return 0;

}