**Practical 2**

//

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

#include <iostream>

#include <list>

#include <string>

#define SIZE 10

using namespace std;

class HashTable {

private:

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

int getHash(string key) {

int hash = 0;

for (char c : key) {

hash += c;

}

return hash % SIZE;

}

public:

HashTable() {

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

}

void insert(string key, string value) {

int index = getHash(key);

table[index].push\_back(make\_pair(key, value));

}

string search(string key) {

int index = getHash(key);

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

if (it->first == key) {

return it->second;

}

}

return "";

}

void remove(string key) {

int index = getHash(key);

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

if (it->first == key) {

table[index].erase(it);

break;

}

}

}

void display() {

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

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

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

cout << "Key: " << it->first << ", Value: " << it->second << endl;

}

}

}

};

int main() {

HashTable ht;

string key, value;

int choice;

do {

cout << "Menu:\n";

cout << "1. Insert\n";

cout << "2. Search\n";

cout << "3. Remove\n";

cout << "4. Display\n";

cout << "5. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter key: ";

cin >> key;

cout << "Enter value: ";

cin >> value;

if(ht.search(key) != "") {

cout << "Value already Present" << endl;

}else {

ht.insert(key, value);

}

break;

case 2:

cout << "Enter key: ";

cin >> key;

value = ht.search(key);

if (value.empty()) {

cout << "Key not found\n";

} else {

cout << "Value: " << value << endl;

}

break;

case 3:

cout << "Enter key: ";

cin >> key;

ht.remove(key);

break;

case 4:

ht.display();

break;

case 5:

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

break;

default:

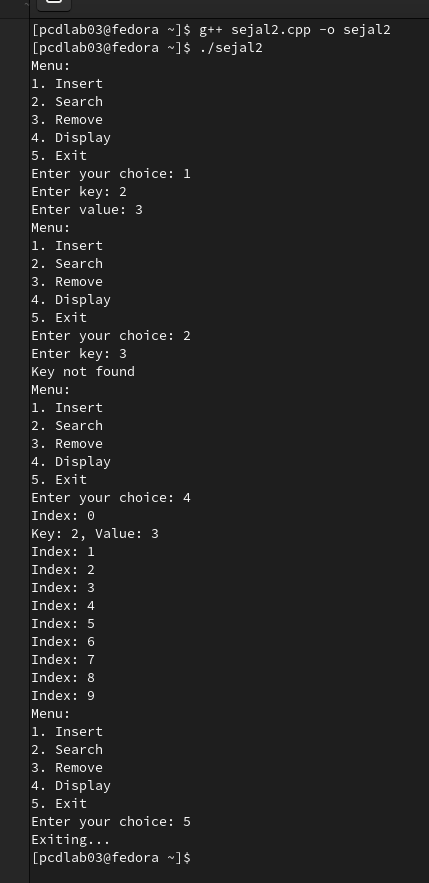
cout << "Invalid choice\n";

}

} while (choice != 5);

return 0;

}



**Practical 3**

//A book consists of chapters, chapters consist of sections and sections consist ofsubsections. Construct a tree and print the nodes. Find the time and space requirementsof your method.

#include <iostream>

using namespace std;

struct Node

{

string label;

int n;

struct Node \*child[10];

} \*root;

class GT

{

public:

GT() {

root = NULL;

}

void createTree()

{

root = new Node;

cout << "Enter the name of the book :";

getline(cin, root->label);

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

cin >> root->n;

cin.ignore();

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

root->child[i] = new Node;

cout << "Enter the name of chapter " << i + 1 << " : ";

getline(cin, root->child[i]->label);

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

cin >> root->child[i]->n;

cin.ignore();

for(int j = 0; j < root->child[i]->n; j++) {

root->child[i]->child[j] = new Node;

cout << " Enter the name of section " << j + 1 << " : ";

getline(cin, root->child[i]->child[j]->label);

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

cin >> root->child[i]->child[j]->n;

cin.ignore();

for(int k = 0; k < root->child[i]->child[j]->n; k++) {

root->child[i]->child[j]->child[k] = new Node;

cout << "Enter the name of the subsection " << k + 1 << ":";

getline(cin,root->child[i]->child[j]->child[k]->label);

}

}

}

}

void display(struct Node\*root) {

if(root) {

cout << "BOOK Name -> " << root->label << endl;

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

cout << "\tChapter -> " << root->child[i]->label << endl;

for(int j = 0; j < root->child[i]->n; j++) {

cout << "\t\t Section -> " << root->child[i]->child[j]->label << endl;

for(int k = 0; k < root->child[i]->child[j]->n;k++) {

cout << "\t\t\tSub-section -> " << root->child[i]->child[j]->child[k]->label << endl;

}

}

}

}

}

};

int main()

{

int choice;

GT gt;

while (1)

{

cout << "---------------------" << endl;

cout << " BOOK TREE CREATION " << endl;

cout << "----------------------" << endl;

cout << "1.Create" << endl;

cout << "2.Display" << endl;

cout << "3.Quit" << endl;

cout << "Enter your choice: ";

cin >> choice;

cin.ignore();

switch (choice)

{

case 1:

gt.createTree();

break;

case 2:

gt.display(root);

break;

case 3:

cout << " Thanks for using this program !";

exit(1);

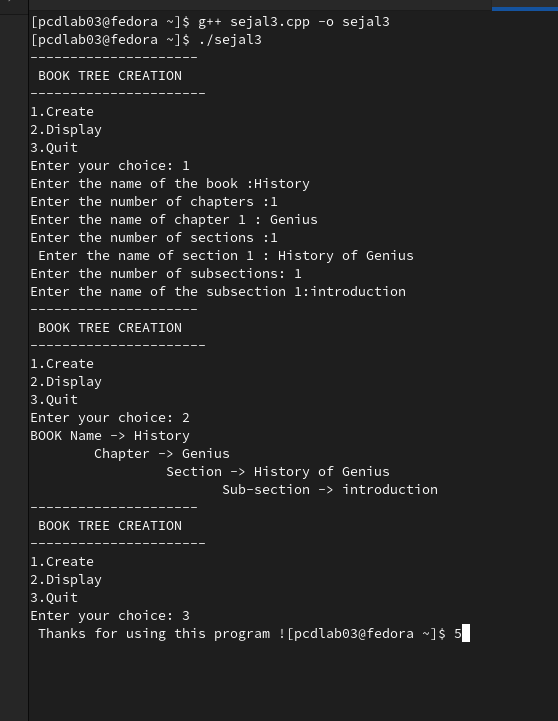
default:

cout << "Wrong Choice !!! " << endl;

}

}

return 0;

}

**Practical 4**

//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>

using namespace std;

class Node

{

public:

int data;

Node \*left, \*right;

Node(int data, Node \*left, Node \*right)

{

this->data = data;

this->left = left;

this->right = right;

}

};

class BST

{

public:

Node \*root;

BST()

{

root = NULL;

}

Node \*insert(Node \*root, int x)

{

Node \*temp = new Node(x, NULL, NULL);

if (!root)

{

root = temp;

}

else if (root->data > x)

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

else if (root->data < x)

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

return root;

}

void search(Node \*root, int x)

{

if (root)

{

if (root->data == x)

{

cout << "Entry found :) " << endl;

return;

}

else if (x < root->data)

search(root->left, x);

else if (x > root->data)

search(root->right, x);

else

{

return;

}

}

}

void min(Node \*root)

{

Node \*p = root;

while (p->left)

p = p->left;

cout << p->data << endl;

}

void inorder(Node \*root)

{

if (root)

{

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

}

void preorder(Node \*root)

{

if (root)

{

cout << root->data << " ";

preorder(root->left);

preorder(root->right);

}

}

void postorder(Node \*root)

{

if (root)

{

postorder(root->left);

postorder(root->right);

cout << root->data << " ";

}

}

int height(Node \*root) {

if (root == NULL) {

return 0;

}

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

}

void swapChildren(Node \*root) {

if (root != NULL) {

swap(root->left, root->right);

swapChildren(root->left);

swapChildren(root->right);

}

}

};

int main()

{

BST tree;

int ch = 5;

do

{

cout << "Choose Operation: " << endl;

cout << "1. Insert" << endl;

cout << "2. Traversal" << endl;

cout << "3. Search" << endl;

cout << "4. Min" << endl;

cout << "5. Longest Path from root" << endl;

cout << "6. Swap nodes" << endl;

cout << "7. Exit " << endl;

cin >> ch;

switch (ch)

{

case 1:

[&]()

{

int n;

cout << "Enter the number of Elements you want to insert: ";

cin >> n;

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

{

int ip;

cin >> ip;

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

}

cout << "Insertion Complete ... " << endl

<< endl;

}();

break;

case 2:

[&]()

{

cout << "Traversals ..." << endl;

cout << "PreOrder" << endl;

tree.preorder(tree.root);

cout << endl;

cout << "InOrder" << endl;

tree.inorder(tree.root);

cout << endl;

cout << "PostOrder" << endl;

tree.postorder(tree.root);

cout << endl;

}();

break;

case 3:

[&]() {

int ip;

cout << "Enter element to search : ";

cin >> ip;

tree.search(tree.root, ip);

}();

break;

case 4:

[&]() {

cout << "Min Element : ";

tree.min(tree.root);

}();

break;

case 5:

[&]() {

cout << "Longest path : ";

cout << tree.height(tree.root) << endl;

}();

break;

case 6:

[&]() {

cout << "Swap nodes: ";

tree.swapChildren(tree.root);

tree.preorder(tree.root);

}();

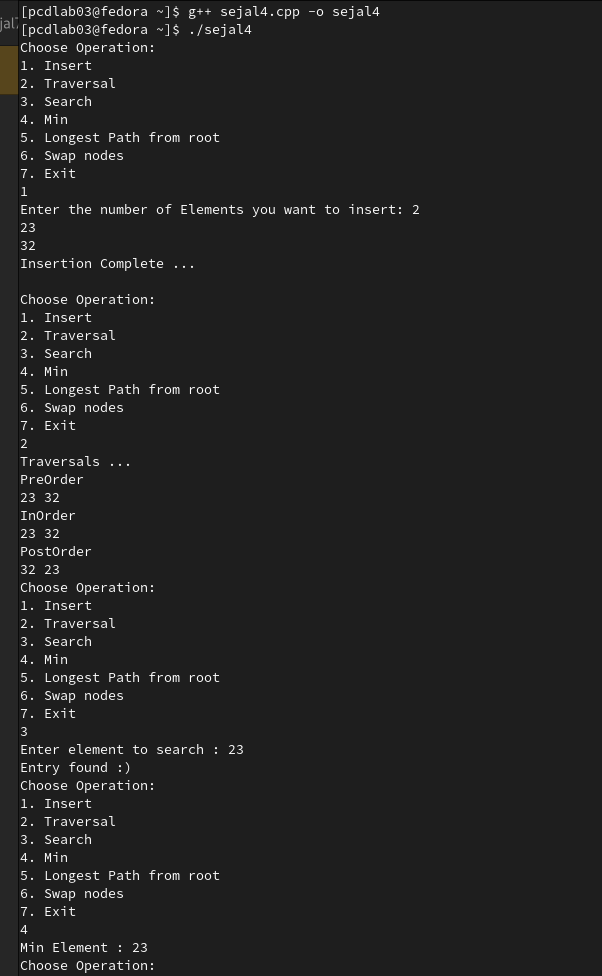
break;

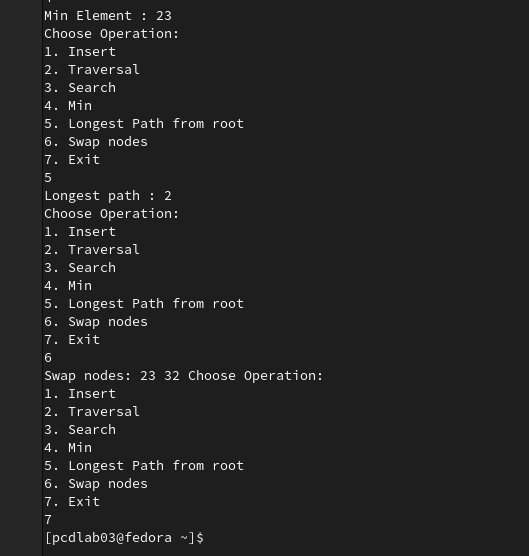
}

} while (ch != 7);

return 0;

}





**Practical 5**

//Title:Convert Binary Tree to Threaded Binary Tree

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

bool rightThread;

Node(int val) {

data = val;

left = right = nullptr;

rightThread = false;

}

};

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

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

Node\* curr = root;

while (curr) {

if (key < curr->data) {

if (!curr->left) {

curr->left = new Node(key);

return root;

}

curr = curr->left;

} else {

if (!curr->right) {

curr->right = new Node(key);

return root;

}

curr = curr->right;

}

}

return root;

}

// Function to convert binary tree to threaded binary tree

void convertToThreadedBinaryTree(Node\* root) {

if (!root) return;

Node\* prev = nullptr;

Node\* curr = root;

while (curr) {

if (!curr->left) {

if (prev) {

prev->right = curr;

prev->rightThread = true;

}

prev = curr;

curr = curr->right;

} else {

Node\* pre = curr->left;

while (pre->right && pre->right != curr) {

pre = pre->right;

}

if (!pre->right) {

pre->right = curr;

pre->rightThread = false;

curr = curr->left;

} else {

pre->right = nullptr;

pre->rightThread = true;

if (prev) {

prev->right = curr;

prev->rightThread = true;

}

prev = curr;

curr = curr->right;

}

}

}

}

// Inorder traversal of threaded binary tree

void inorder(Node\* root) {

Node\* curr = root;

while (curr) {

while (curr->left) {

curr = curr->left;

}

cout << curr->data << " ";

while (curr->rightThread) {

curr = curr->right;

cout << curr->data << " ";

}

curr = curr->right;

}

cout << endl;

}

int main() {

Node\* root = nullptr;

root = insert(root, 20);

root = insert(root, 10);

root = insert(root, 30);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 25);

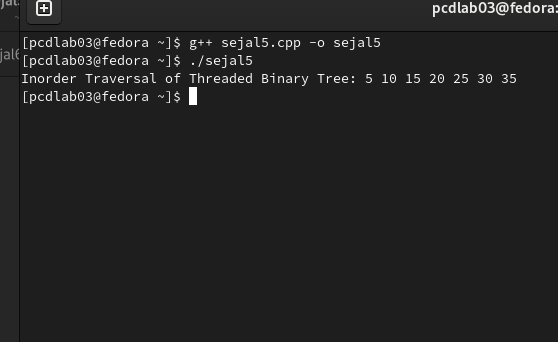
root = insert(root, 35);

convertToThreadedBinaryTree(root);

cout << "Inorder Traversal of Threaded Binary Tree: ";

inorder(root);

return 0;

}

**//practical 6**

//Title:Graph representation using Matrix and perform DFS & BFS

#include <iostream>

#include <vector>

#include <queue>

#include <string> // Include string header for string usage

using namespace std;

class Graph {

private:

vector<vector<int>> adjMatrix;

vector<vector<int>> adjList;

vector<string> landmarks;

int numLandmarks;

public:

Graph(int n) {

numLandmarks = n;

adjMatrix.resize(n, vector<int>(n, 0));

adjList.resize(n);

landmarks.resize(n);

}

void inputLandmarks() {

cout << "Enter names of " << numLandmarks << " landmarks:\n";

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

cout << "Landmark " << i + 1 << ": ";

cin >> landmarks[i];

}

}

void addConnections() {

int edges;

cout << "Enter number of connections (edges): ";

cin >> edges;

cout << "Enter connections between landmarks (use index starting from 0):\n";

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

int u, v;

cout << "Enter edge (landmark1 landmark2): ";

cin >> u >> v;

adjMatrix[u][v] = adjMatrix[v][u] = 1;

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

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

}

}

void displayAdjMatrix() {

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

for (const auto& lm : landmarks) cout << lm << "\t";

cout << "\n";

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

cout << landmarks[i] << "\t";

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

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

}

cout << "\n";

}

}

void DFSUtil(int node, vector<bool>& visited) {

cout << landmarks[node] << " ";

visited[node] = true;

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

if (adjMatrix[node][i] == 1 && !visited[i]) {

DFSUtil(i, visited);

}

}

}

void DFS(int start) {

cout << "\nDFS Traversal: ";

vector<bool> visited(numLandmarks, false);

DFSUtil(start, visited);

cout << endl;

}

void BFS(int start) {

cout << "\nBFS Traversal: ";

vector<bool> visited(numLandmarks, false);

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty()) {

int node = q.front();

q.pop();

cout << landmarks[node] << " ";

for (int neighbor : adjList[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

cout << endl;

}

};

int main() {

int n, start;

cout << "Enter number of landmarks: ";

cin >> n;

Graph g(n);

g.inputLandmarks();

g.addConnections();

g.displayAdjMatrix();

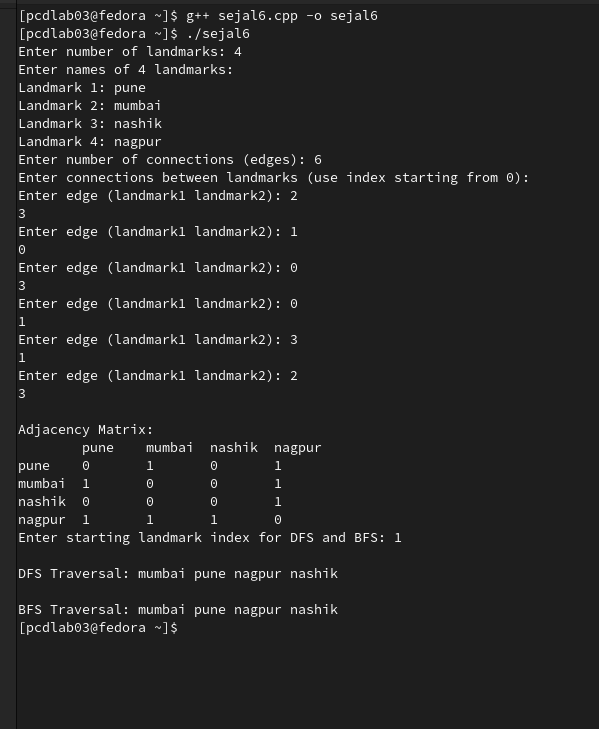
cout << "Enter starting landmark index for DFS and BFS: ";

cin >> start;

g.DFS(start);

g.BFS(start);

return 0;

}

**//practical 7**

//Title:Graph Representation using Matrix of Flights between Cities

#include <iostream>

#include <vector>

#include <string> // Include the string header for string usage

using namespace std;

class FlightGraph {

private:

vector<vector<int>> adjMatrix;

vector<string> cities;

int numCities;

public:

FlightGraph(int n) {

numCities = n;

adjMatrix.resize(n, vector<int>(n, 0));

cities.resize(n);

}

void inputCities() {

cout << "Enter names of " << numCities << " cities:\n";

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

cout << "City " << i + 1 << ": ";

cin >> cities[i];

}

}

void addFlightPaths() {

int time;

cout << "Enter travel time in hour between cities (enter 0 if no direct flight):\n";

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

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

if (i != j) {

cout << "Time from " << cities[i] << " to " << cities[j] << ": ";

cin >> time;

adjMatrix[i][j] = time;

}

}

}

}

void displayAdjMatrix() {

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

cout << "\t";

for (const auto& city : cities) {

cout << city << "\t";

}

cout << "\n";

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

cout << cities[i] << "\t";

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

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

}

cout << endl;

}

}

};

int main() {

int n;

cout << "Enter number of cities: ";

cin >> n;

FlightGraph graph(n);

graph.inputCities();

graph.addFlightPaths();

graph.displayAdjMatrix();

return 0;

}

**// Practical 8**

//Given sequence k = k1 <k2 < … <kn of n sorted keys, with a search probability pi for eachkey ki . Build the Binary search tree that has the least search cost given the accessprobability for each key?

#include <iostream>

#include <vector>

using namespace std;

float sum(const vector<float>& p, int i, int j) {

float s = 0;

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

s += p[k];

return s;

}

float optimalBST(const vector<float>& p, int n) {

vector<vector<float>> cost(n, vector<float>(n, 0));

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

cost[i][i] = p[i];

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

{

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

int j = i + L - 1;

cost[i][j] = 1e9;

for (int r = i; r <= j; r++) {

float left = (r > i) ? cost[i][r - 1] : 0;

float right = (r < j) ? cost[r + 1][j] : 0;

float total = left + right + sum(p, i, j);

if (total < cost[i][j])

cost[i][j] = total;

}

}

}

return cost[0][n - 1];

}

int main() {

int n;

cout << "Enter number of keys: ";

cin >> n;

vector<int> keys(n);

vector<float> probs(n);

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

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

cin >> keys[i];

cout << "Enter search probabilities:\n";

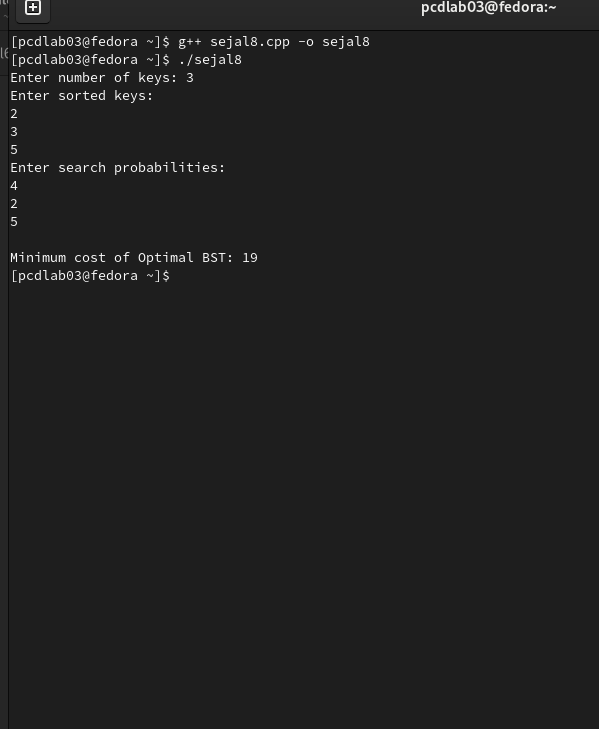
for (int i = 0; i < n; i++)

cin >> probs[i];

float minCost = optimalBST(probs, n);

cout << "\nMinimum cost of Optimal BST: " << minCost << endl;

return 0;

}

//practical 9

TITLE: 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 <string>

using namespace std;

class dict {

private:

struct Node {

string s1; // keyword

string s2; // meaning

Node\* left;

Node\* right;

};

Node\* root;

public:

dict() {

root = NULL;

}

void input();

void create\_root(Node\* &tree, Node\* node);

void check\_same(Node\* tree, Node\* node, bool &exists);

void input\_display();

void display(Node\* tree);

void input\_remove();

Node\* remove(Node\* tree, const string &s);

Node\* findmin(Node\* tree);

void input\_find();

Node\* find(Node\* tree, const string &s);

void input\_update();

Node\* update(Node\* tree, const string &s);

};

void dict::input() {

Node\* node = new Node;

cout << "\nEnter the keyword:\n";

cin >> node->s1;

cout << "Enter the meaning of the keyword:\n";

cin.ignore();

getline(cin, node->s2);

create\_root(root, node);

}

void dict::create\_root(Node\* &tree, Node\* node) {

if (tree == NULL) {

tree = node;

tree->left = NULL;

tree->right = NULL;

cout << "\nRoot node created successfully" << endl;

return;

}

if (node->s1 < tree->s1) {

create\_root(tree->left, node);

} else if (node->s1 > tree->s1) {

create\_root(tree->right, node);

} else {

cout << "The word already exists in the dictionary.\n";

delete node; // Avoid memory leak

}

}

void dict::input\_display() {

if (root != NULL) {

cout << "The words entered in the dictionary are:\n\n";

display(root);

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

void dict::display(Node\* tree) {

if (tree != NULL) {

display(tree->left);

cout << tree->s1 << " = " << tree->s2 << "\n";

display(tree->right);

}

}

void dict::input\_remove() {

if (root != NULL) {

string s;

cout << "\nEnter a keyword to be deleted:\n";

cin >> s;

root = remove(root, s);

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

dict::Node\* dict::remove(Node\* tree, const string &s) {

if (tree == NULL) {

cout << "\nWord not found.\n";

return tree;

}

if (s < tree->s1) {

tree->left = remove(tree->left, s);

} else if (s > tree->s1) {

tree->right = remove(tree->right, s);

} else {

// Node with only one child or no child

if (tree->left == NULL) {

Node\* temp = tree->right;

delete tree;

return temp;

} else if (tree->right == NULL) {

Node\* temp = tree->left;

delete tree;

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

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

tree->s1 = temp->s1;

tree->s2 = temp->s2;

tree->right = remove(tree->right, temp->s1);

}

return tree;

}

dict::Node\* dict::findmin(Node\* tree) {

while (tree && tree->left != NULL) {

tree = tree->left;

}

return tree;

}

void dict::input\_find() {

if (root != NULL) {

string s;

cout << "\nEnter the keyword to be searched:\n";

cin >> s;

find(root, s);

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

dict::Node\* dict::find(Node\* tree, const string &s) {

if (tree == NULL) {

cout << "\nWord not found.\n";

return NULL;

}

if (s == tree->s1) {

cout << "\nWord found: " << tree->s1 << " = " << tree->s2 << "\n";

return tree;

} else if (s < tree->s1) {

return find(tree->left, s);

} else {

return find(tree->right, s);

}

}

void dict::input\_update() {

// Implementation for updating a word's meaning can be added here

}

int main() {

dict d;

int ch;

do {

cout << "\n1. Input\n2. Display\n3. Remove\n4. Find\n5. Update\n6. Exit\n";

cout << "Enter your choice: ";

cin >> ch;

switch (ch) {

case 1:

d.input();

break;

case 2:

d.input\_display();

break;

case 3:

d.input\_remove();

break;

case 4:

d.input\_find();

break;

case 5:

d.input\_update();

break;

case 6:

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

break;

default:

cout << "\nPlease enter a valid option!\n";

break;

}

} while (ch != 6);

return 0;

}

**//practical 10**

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

#include <iostream>

using namespace std;

// Function to heapify a subtree rooted at index i

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

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child index

int right = 2 \* i + 2; // Right child index

// If left child exists and is greater than root

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

largest = left;

// If right child exists and is greater than current largest

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

largest = right;

// If largest is not root

if (largest != i) {

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

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Heap Sort function

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

// Build max heap

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

heapify(arr, n, i);

// Extract elements from heap one by one

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

// Move current root to the end

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

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Display array function

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

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

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

cout << "\n";

}

// Main function to test heap sort

int main() {

int arr[] = {15, 3, 17, 10, 84, 19, 6, 22, 9};

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

cout << "Original array:\n";

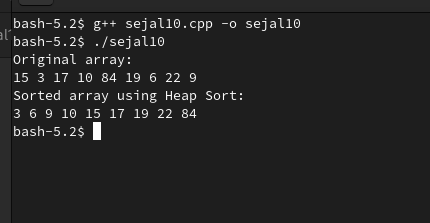
display(arr, n);

heapSort(arr, n);

cout << "Sorted array using Heap Sort:\n";

display(arr, n);

return 0;

}

**//practical 11**

//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 ofstudent does not exist an appropriate message is displayed. If it is, then the system displays the studentdetails. Use sequential file to main the data.

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

struct Student {

string roll;

string name;

string division;

string address;

};

void addStudent() {

Student s;

cout << "Enter Roll Number: ";

cin >> s.roll;

cin.ignore();

cout << "Enter Name: ";

getline(cin, s.name);

cout << "Enter Division: ";

getline(cin, s.division);

cout << "Enter Address: ";

getline(cin, s.address);

ofstream outFile("students.txt", ios::app);

outFile << s.roll << "," << s.name << "," << s.division << "," << s.address << "\n";

outFile.close();

cout << "Student added successfully.\n\n";

}

void deleteStudent() {

string rollToDelete;

bool found = false;

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

cin >> rollToDelete;

ifstream inFile("students.txt");

ofstream tempFile("temp.txt");

string line;

while (getline(inFile, line)) {

if (line.substr(0, line.find(',')) != rollToDelete) {

tempFile << line << "\n";

} else {

found = true;

}

}

inFile.close();

tempFile.close();

remove("students.txt");

rename("temp.txt", "students.txt");

if (found)

cout << "Student deleted successfully.\n\n";

else

cout << "Student record not found.\n\n";

}

void displayStudent() {

string rollToSearch;

bool found = false;

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

cin >> rollToSearch;

ifstream inFile("students.txt");

string line;

while (getline(inFile, line)) {

if (line.substr(0, line.find(',')) == rollToSearch) {

found = true;

size\_t pos1 = line.find(',');

size\_t pos2 = line.find(',', pos1 + 1);

size\_t pos3 = line.find(',', pos2 + 1);

cout << "\nStudent Found:\n";

cout << "Roll No: " << line.substr(0, pos1) << "\n";

cout << "Name: " << line.substr(pos1 + 1, pos2 - pos1 - 1) << "\n";

cout << "Division: " << line.substr(pos2 + 1, pos3 - pos2 - 1) << "\n";

cout << "Address: " << line.substr(pos3 + 1) << "\n\n";

break;

}

}

inFile.close();

if (!found)

cout << "Student record not found.\n\n";

}

int main() {

int choice;

do {

cout << "----- Student Information System -----\n";

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

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

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

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

addStudent();

break;

case 2:

deleteStudent();

break;

case 3:

displayStudent();

break;

case 4:

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

break;

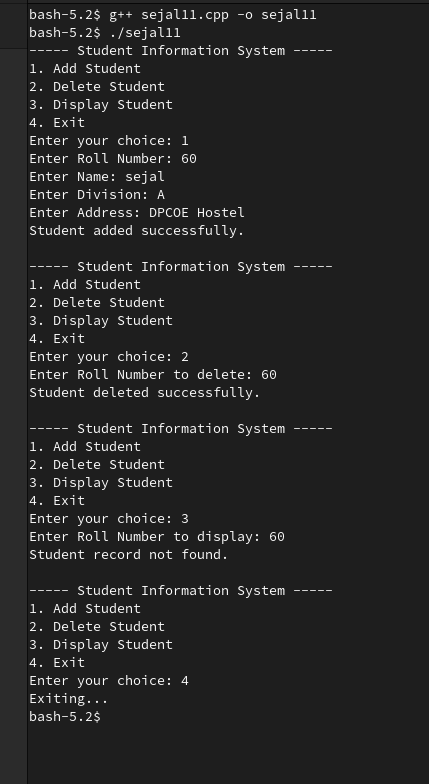
default:

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

}

} while (choice != 4);

return 0;

}