21CO118

Sonshree Piwalatkar

SE COMP B

4th Semester

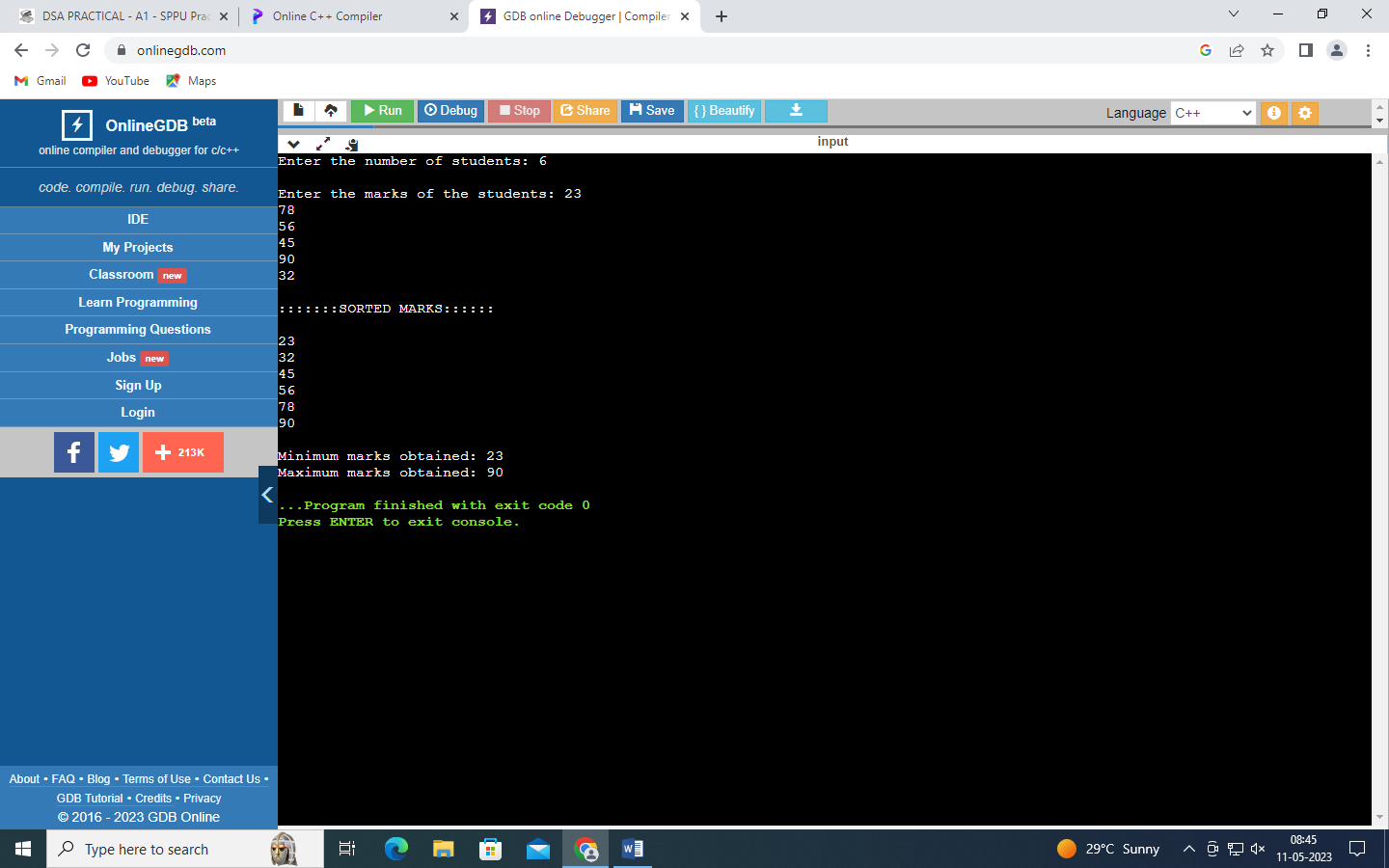
**DATA STRUCTURE AND ALGORITHMS**

Practical No-A1

Q1)Read the marks obtained by students of second year in an online examination of  
particular subject. Find out maximum and minimum marks obtained in that subject. Use  
heap data structure. Analyze the algorithm.

CODE->

|  |
| --- |
| #include <iostream> |
|  | using namespace std; |
|  |  |
|  | class Heap{ |
|  | public: |
|  | void maxHeapify (int [], int, int); |
|  | void buildMaxHeap (int [], int); |
|  | void heapsort (int [], int); |
|  | void accept (); |
|  | void display (int [],int); |
|  | }; |
|  |  |
|  | void Heap::maxHeapify (int marks[], int i, int n) { //reheapdown - deleting element from top location |
|  | int l, r, largest; |
|  | l = 2 \* i; |
|  | r = (2 \* i + 1); |
|  |  |
|  | largest = ((l <= n) && marks[l] > marks[i]) ? l : i; |
|  |  |
|  | if ((r <= n) && (marks[r] > marks[largest])) |
|  | largest=r; |
|  |  |
|  | if (largest != i) { |
|  | swap(marks[largest], marks[i]); |
|  | maxHeapify (marks, largest,n); |
|  | } |
|  | } |
|  |  |
|  | void Heap::buildMaxHeap (int marks[], int n) { |
|  | for (int k = n / 2; k >= 1; k--) |
|  | maxHeapify (marks, k, n); |
|  | } |
|  |  |
|  | void Heap::heapsort (int marks[], int n) { |
|  | buildMaxHeap (marks,n); |
|  | for (int i = n; i >= 2; i--) { |
|  | swap (marks[i], marks[1]); |
|  | maxHeapify (marks, 1, i - 1); |
|  | } |
|  | } |
|  |  |
|  | void Heap::accept (){ |
|  | int n; |
|  | cout << "Enter the number of students: "; |
|  | cin >> n; |
|  | int marks[n]; |
|  | cout << "\nEnter the marks of the students: "; |
|  | for (int i = 1; i <= n; i++) |
|  | cin >> marks[i]; |
|  |  |
|  | heapsort (marks, n); |
|  | display (marks, n); |
|  | } |
|  |  |
|  | void Heap::display (int marks[],int n) { |
|  | cout << "\n:::::::SORTED MARKS::::::\n\n"; |
|  |  |
|  | for (int i = 1; i <= n; i++) |
|  | cout << marks[i] << endl; |
|  |  |
|  | cout << "\nMinimum marks obtained: " << marks[1]; |
|  | cout << "\nMaximum marks obtained: " << marks[n]; |
|  | } |
|  |  |
|  | int main () { |
|  | Heap h; |
|  | h.accept (); |
|  | return 0; |
|  | }  OUTPUT: |



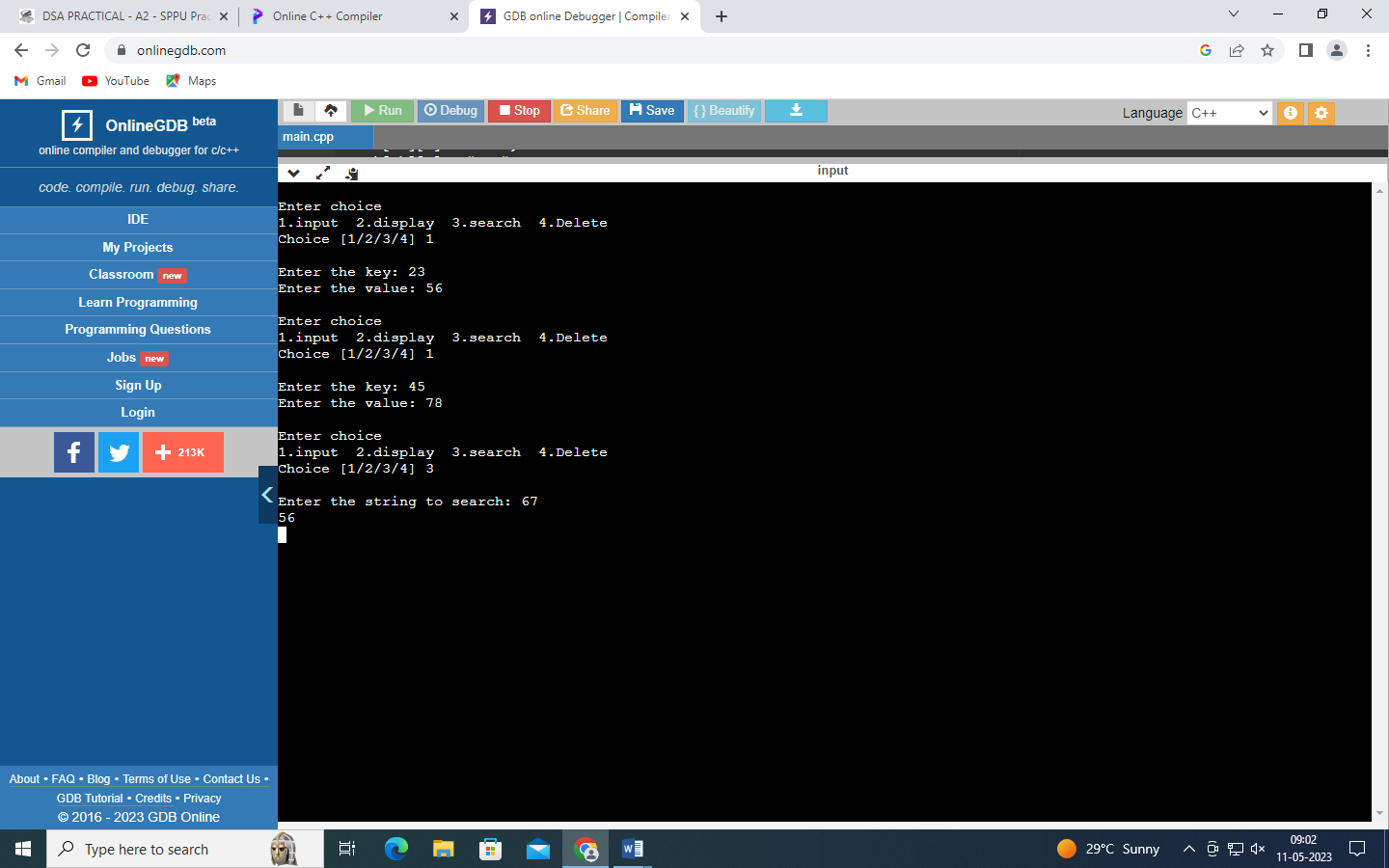
Practical A2

Q2)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)

CODE->

|  |
| --- |
| #include <iostream> |
|  | #include <cstring> |
|  | #define size 7 |
|  | using namespace std; |
|  |  |
|  | class Hashtable { |
|  | public: |
|  | char key[size], value[size], k[size]; |
|  | string h[size][2]; |
|  | int count, ch; |
|  | void input(); |
|  | void display(); |
|  | void hashf(char[]); |
|  | void linearp(string key, int); |
|  | void search(); |
|  | void Delete(); |
|  | Hashtable() { |
|  | for (int i = 0; i < size; i++) |
|  | for (int j = 0; j < 2; j++) |
|  | h[i][j] = "---"; |
|  |  |
|  | count = 0; |
|  | } |
|  | }; |
|  |  |
|  | void Hashtable ::input() { |
|  | cout << "\nEnter the key: "; |
|  | cin >> key; |
|  | cout << "Enter the value: "; |
|  | cin >> value; |
|  | hashf(key); |
|  | } |
|  |  |
|  | void Hashtable ::hashf(char key[]) { |
|  | int sum = 0; |
|  | for (int i = 0; i < strlen(key); i++) |
|  | sum = sum + int(key[i]); |
|  |  |
|  | ch = sum % size; |
|  | linearp(key, ch); |
|  | } |
|  |  |
|  | void Hashtable ::linearp(string key, int ch) { |
|  | if (count == size) |
|  | cout << "Table is Full"; |
|  |  |
|  | else { |
|  | while (h[ch][0] != "---" && count != size) |
|  | ch = ++ch % size; |
|  |  |
|  | h[ch][0] = key; |
|  | h[ch][1] = value; |
|  | count++; |
|  | } |
|  | } |
|  |  |
|  | void Hashtable ::display() { |
|  | cout << "\n\t[Key]\t[Value]\n"; |
|  | for (int i = 0; i < size; i++) { |
|  | for (int j = 0; j < 2; j++) |
|  | cout << "\t" << h[i][j]; |
|  |  |
|  | cout << "\n"; |
|  | } |
|  | } |
|  |  |
|  | void Hashtable ::search() { |
|  | cout << "\nEnter the string to search: "; |
|  | cin >> k; |
|  | int sum = 0; |
|  | for (int i = 0; i < strlen(k); i++) |
|  | sum = sum + int(k[i]); |
|  |  |
|  | ch = sum % size; |
|  | if (count == size) |
|  | cout << "\nSearch is not found \n"; |
|  |  |
|  | else { |
|  | while (h[ch][0] != k && count != size) |
|  | ch = ++ch % size; |
|  |  |
|  | count++; |
|  | if (h[ch][0] == k) |
|  | cout << "String '" << k << "' found at index " << ch << "\n"; |
|  | } |
|  | } |
|  |  |
|  | void Hashtable ::Delete() { |
|  | cout << "\nEnter the string to delete: "; |
|  | cin >> k; |
|  | int sum = 0; |
|  | for (int i = 0; i < strlen(k); i++) |
|  | sum = sum + int(k[i]); |
|  |  |
|  | ch = sum % size; |
|  | if (count == size) |
|  | cout << "Search is not found \n"; |
|  |  |
|  | else |
|  | while (h[ch][0] != k && count != size) |
|  | ch = ++ch % size; |
|  |  |
|  | h[ch][0] = "---"; |
|  | h[ch][1] = "---"; |
|  | cout << "String '" << k << "' deleted from index " << ch << "\n"; |
|  | } |
|  |  |
|  | int main() { |
|  | int ch; |
|  | Hashtable h1; |
|  | do { |
|  | cout << "\nEnter choice\n1.input 2.display 3.search 4.Delete\nChoice [1/2/3/4] "; |
|  | cin >> ch; |
|  | switch (ch) { |
|  | case 1: |
|  | h1.input(); |
|  | break; |
|  | case 2: |
|  | h1.display(); |
|  | break; |
|  | case 3: |
|  | h1.search(); |
|  | break; |
|  | case 4: |
|  | h1.Delete(); |
|  | break; |
|  | } |
|  | } while (ch < 5); |
|  | return 0; |
|  | } |

OUTPUT:



PRACTICAL B5

Q3) A book consists of chapters, chapters consist of sections and sections consist of  
subsections. Construct a tree and print the nodes. Find the time and space requirements  
of your method.

CODE->

struct node // Node Declaration

{

string label;

//char label[10];

int ch\_count;

struct node \*child[10];

} \* root;

class GT // Class Declaration

{

public:

void create\_tree();

void display(node \*r1);

GT()

{

root = NULL;

}

};

void GT::create\_tree()

{

int tbooks, tchapters, i, j, k;

root = new node;

cout << "Enter name of book : ";

cin.get();

getline(cin, root->label);

cout << "Enter number of chapters in book : ";

cin >> tchapters;

root->ch\_count = tchapters;

for (i = 0; i < tchapters; i++)

{

root->child[i] = new node;

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

cin.get();

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

cout << "Enter number of sections in Chapter : " << root->child[i]->label << " : ";

cin >> root->child[i]->ch\_count;

for (j = 0; j < root->child[i]->ch\_count; j++)

{

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

cout << "Enter Name of Section " << j + 1 << " : ";

cin.get();

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

}

}

}

void GT::display(node \*r1)

{

int i, j, k, tchapters;

if (r1 != NULL)

{

cout << "\n-----Book Hierarchy---";

cout << "\n Book title : " << r1->label;

tchapters = r1->ch\_count;

for (i = 0; i < tchapters; i++)

{

cout << "\nChapter " << i + 1;

cout << " : " << r1->child[i]->label;

cout << "\nSections : ";

for (j = 0; j < r1->child[i]->ch\_count; j++)

{

cout << "\n"<< r1->child[i]->child[j]->label;

}

}

}

cout << 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;

switch (choice)

{

case 1:

gt.create\_tree();

case 2:

gt.display(root);

break;

case 3:

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

exit(1);

default:

cout << "Wrong choice!!!" << endl;

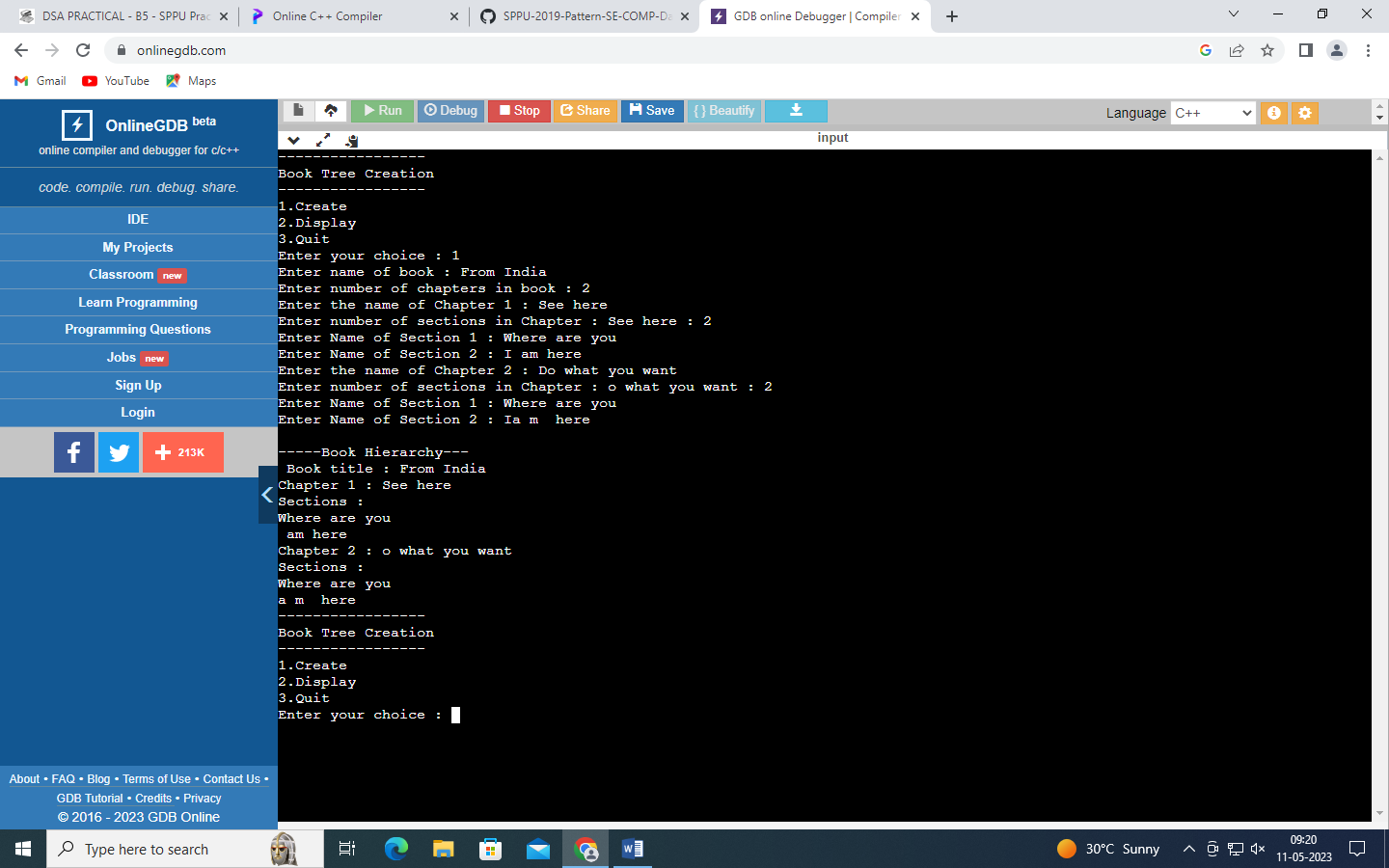
}

}

return 0;

}

OUTPUT:



Practical no.-B6

Q4) 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

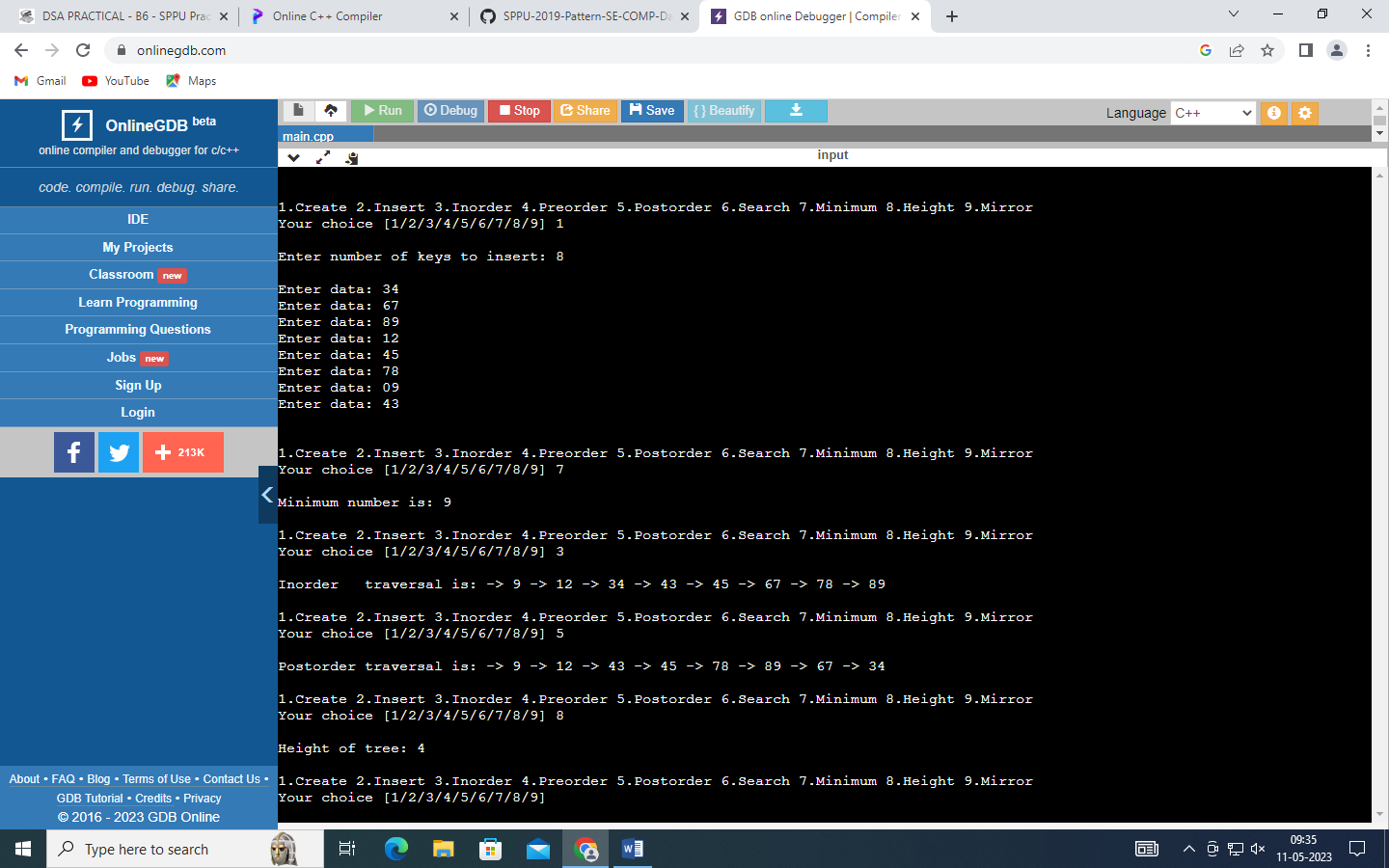
node

v. Search a value

Code:

|  |
| --- |
| #include<iostream> |
|  | using namespace std; |
|  |  |
|  | class node { |
|  | public: |
|  | int data; |
|  | node \*left; |
|  | node \*right; |
|  | }; |
|  |  |
|  | class Bst { |
|  | public: |
|  | node \*root; |
|  | Bst () { |
|  | root = NULL; |
|  | } |
|  | void create (); |
|  | void insert (); |
|  | void postorder (node\*); |
|  | void inorder (node\*); |
|  | void preorder (node\*); |
|  | void search (int key); |
|  | void minimum (); |
|  | int height (node\*); |
|  | void mirror (node\*); |
|  | }; |
|  |  |
|  | void Bst::create () { |
|  | int ans; |
|  | cout << "\nEnter number of keys to insert: "; |
|  | cin >> ans; |
|  | cout << '\n'; |
|  | while (ans--) |
|  | insert(); |
|  | } |
|  |  |
|  | void Bst::inorder (node \*root) { |
|  | if (root != NULL) { |
|  | inorder (root -> left); |
|  | cout << " -> " << root -> data; |
|  | inorder (root -> right); |
|  | } |
|  | } |
|  |  |
|  | void Bst::preorder (node \*root) { |
|  | if (root != NULL) { |
|  | cout << " -> " << root -> data; |
|  | preorder (root -> left); |
|  | preorder (root -> right); |
|  | } |
|  | } |
|  |  |
|  | void Bst::postorder (node \*root) { |
|  | if (root != NULL) { |
|  | postorder (root -> left); |
|  | postorder (root -> right); |
|  | cout << " -> " << root -> data; |
|  | } |
|  | } |
|  |  |
|  | void Bst::insert () { |
|  | node \*curr,\*temp; |
|  | cout << "Enter data: "; |
|  | curr = new node; |
|  | cin >> curr -> data; |
|  | curr -> left = curr -> right = NULL; |
|  |  |
|  | if (root == NULL) |
|  | root = curr; |
|  |  |
|  | else { |
|  | temp = root; |
|  | while (1) { |
|  | if (curr -> data <= temp -> data) { |
|  | if (temp -> left == NULL) { |
|  | temp -> left = curr; |
|  | break; |
|  | } |
|  | else |
|  | temp = temp -> left; |
|  | } |
|  | else { |
|  | if (temp -> right == NULL) { |
|  | temp -> right = curr; |
|  | break; |
|  | } |
|  | else |
|  | temp = temp -> right; |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void Bst::search (int key) { |
|  | node \*curr; |
|  | curr = root; |
|  |  |
|  | while (curr != NULL) { |
|  | if (curr -> data == key) { |
|  | cout << key << " found"; |
|  | break; |
|  | } |
|  | else { |
|  | if (key<curr -> data) |
|  | curr = curr -> left; |
|  | else |
|  | curr = curr -> right; |
|  | } |
|  | } |
|  | if (curr == NULL) |
|  | cout << key << " not found"; |
|  | } |
|  |  |
|  | void Bst::minimum () { |
|  | node \*temp = root; |
|  | int min; |
|  | while (temp -> left != NULL) { |
|  | min = temp -> data; |
|  | temp = temp -> left; |
|  | if (temp -> data<min) |
|  | min = temp -> data; |
|  | else |
|  | temp = temp -> left; |
|  | } |
|  | cout << "\nMinimum number is: " << min; |
|  | } |
|  |  |
|  | int Bst::height (node \*root) { |
|  | if (root == NULL) |
|  | return 0; |
|  |  |
|  | else { |
|  | if (height (root -> right) > height (root -> left)) |
|  | return (1 + height (root -> right)); |
|  |  |
|  | else |
|  | return (1 + height (root -> left)); |
|  | } |
|  | } |
|  |  |
|  | void Bst::mirror (node \*root) { |
|  | if (root == NULL) |
|  | return; |
|  |  |
|  | else { |
|  | mirror(root -> left); |
|  | mirror(root -> right); |
|  | swap(root -> left, root -> right); |
|  | } |
|  | } |
|  |  |
|  | int main () { |
|  | Bst b; |
|  | int key,ch; |
|  | do { |
|  | cout << "\n\n1.Create 2.Insert 3.Inorder 4.Preorder 5.Postorder 6.Search 7.Minimum 8.Height 9.Mirror\n"; |
|  | cout << "Your choice [1/2/3/4/5/6/7/8/9] "; |
|  | cin >> ch; |
|  | switch (ch) { |
|  | case 1: |
|  | b.create (); |
|  | break; |
|  | case 2: |
|  | cout << '\n'; |
|  | b.insert (); |
|  | break; |
|  | case 3: |
|  | cout << "\nInorder traversal is:"; |
|  | b.inorder (b.root); |
|  | break; |
|  | case 4: |
|  | cout << "\nPreorder traversal is:"; |
|  | b.preorder (b.root); |
|  | break; |
|  | case 5: |
|  | cout << "\nPostorder traversal is:"; |
|  | b.postorder (b.root); |
|  | break; |
|  | case 6: |
|  | cout << "\nEnter search key: "; |
|  | cin >> key; |
|  | b.search (key); |
|  | break; |
|  | case 7: |
|  | b.minimum (); |
|  | break; |
|  | case 8: |
|  | cout << "\nHeight of tree: " << b.height (b.root); |
|  | break; |
|  | case 9: |
|  | b.mirror (b.root); |
|  | cout << "\nTree is now mirrored!!!" |
|  | << "\nInorder traversal is:"; |
|  | b.inorder (b.root); |
|  | cout << "\nPreorder traversal is:"; |
|  | b.preorder (b.root); |
|  | cout << "\nPostorder traversal is:"; |
|  | b.postorder (b.root); |
|  | break; |
|  | } |
|  | }while (ch < 10); |
|  | return 0; |
|  | } |

OUTPUT:



Experiment B-11

Q5.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.

CODE:

#include <iostream>

using namespace std;

class node {

public:

string word;

string meaning;

node\* left = NULL;

node\* right = NULL;

node (string x, string y) {

word = x;

meaning = y;

left = NULL;

right = NULL;

}

friend class Dictionary;

};

class Dictionary {

public:

node\* root, \*q; //q is parent here

Dictionary () {

root = NULL;

q = NULL;

}

void insert (node\*, string, string);

void display\_asc (node \*);

void display\_desc (node \*);

void comparisons (node\*, string);

void updateWord (node\*, string);

void deleteWord (node\*, string);

node\* min\_node (node \*);

};

void Dictionary::insert (node\* p, string key, string keyMeaning) {

if (key < p -> word) {

if (p -> left != NULL)

insert (p -> left, key, keyMeaning);

else

p -> left = new node (key, keyMeaning);

}

else if (key > p -> word) {

if (p -> right != NULL)

insert (p -> right, key, keyMeaning);

else

p -> right = new node (key, keyMeaning);

}

}

void Dictionary::display\_asc (node \*p){ //inorder

if (p -> left != NULL)

display\_asc (p -> left);

cout << "\n" << p -> word << " \t" << p -> meaning;

if (p -> right != NULL)

display\_asc (p -> right);

}

void Dictionary::display\_desc (node \*p) {

if (p -> right != NULL)

display\_desc (p -> right);

cout << "\n" << p -> word << " \t" << p -> meaning;

if (p -> left != NULL)

display\_desc (p -> left);

}

void Dictionary::comparisons (node\* p, string key) {

static int count = 0;

while (p != NULL) {

if (key < p -> word) {

count++;

p = p -> left;

}

else if (key > p -> word) {

count++;

p = p -> right;

}

else if (key == p -> word) {

count++;

cout << "Number of comparisons to find the word: " << count;

return ;

}

}

cout << "\nWord not found!";

}

void Dictionary::deleteWord (node\* p, string key) {

node \*s;

while (p != NULL) { //searching for word

if (key < p -> word) {

q = p;

p = p -> left;

}

else if (key > p -> word) {

q = p;

p = p -> right;

}

else if (key == p -> word) { //word found

if (p -> left == NULL && p -> right == NULL) { //no child

if (q -> left == p) {

delete p;

q -> left = NULL;

return;

}

if (q -> right == p) {

delete p;

q -> right = NULL;

return;

}

}

if (p -> right != NULL && p -> left == NULL) { //right child only

if (q -> right == p) {

q -> right = p -> right;

delete p;

return;

}

else if (q -> left == p) {

q -> left = p -> right;

delete p;

return;

}

}

else if (p -> left != NULL && p -> right == NULL) { //left child only

if (q -> right == p) {

q -> right = p -> left;

delete p;

return;

}

else if (q -> left == p) {

q -> left = p -> left;

delete p;

return;

}

}

else if (p -> left != NULL && p -> right != NULL) {

s = min\_node (p -> right);

p -> word = s -> word;

p -> meaning = s -> meaning;

deleteWord (s, s -> word);

return;

}

}

}

cout << "\nWord NOT found!";

}

void Dictionary::updateWord (node\* p, string key) {

while (p != NULL) {

if (key < p -> word)

p = p -> left;

else if (key > p -> word)

p = p -> right;

else if (key == p -> word) {

cout << "\nEnter its new meaning: ";

cin >> p -> meaning;

return;

}

}

cout << "\nWord not found!";

}

node\* Dictionary::min\_node (node \*p) {

while (p -> left != NULL) {

q = p;

p = p -> left;

}

return p;

}

int main () {

int choice, n;

string newWord, searchWord, newMeaning;

Dictionary d1;

do {

cout << "\n\nDICTIONARY: "

<< "\n\n1. Insert new words"

<< "\n2. Display the dictionary in ascending order"

<< "\n3. Display the dictionary in descending order"

<< "\n4. Search and update a word"

<< "\n5. Delete a word"

<< "\n6. Comparisons"

<< "\n\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "\nEnter the number of words to insert: ";

cin >> n;

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

cout << "\nEnter the word to be inserted: ";

cin >> newWord;

cout << "\nEnter its meaning: ";

cin >> newMeaning;

if (d1.root == NULL)

d1. root = new node (newWord, newMeaning);

else

d1.insert (d1.root, newWord, newMeaning);

}

break;

case 2:

d1.display\_asc (d1.root);

break;

case 3:

d1.display\_desc (d1.root);

break;

case 4:

cout << "\nEnter the word to search: ";

cin >> searchWord;

d1.updateWord (d1.root, searchWord);

break;

case 5:

cout << "\nEnter the word to delete: ";

cin >> searchWord;

d1.deleteWord (d1.root, searchWord);

break;

case 6:

cout << "\nEnter the word to find comparisons: ";

cin >> searchWord;

d1.comparisons (d1.root, searchWord);

}

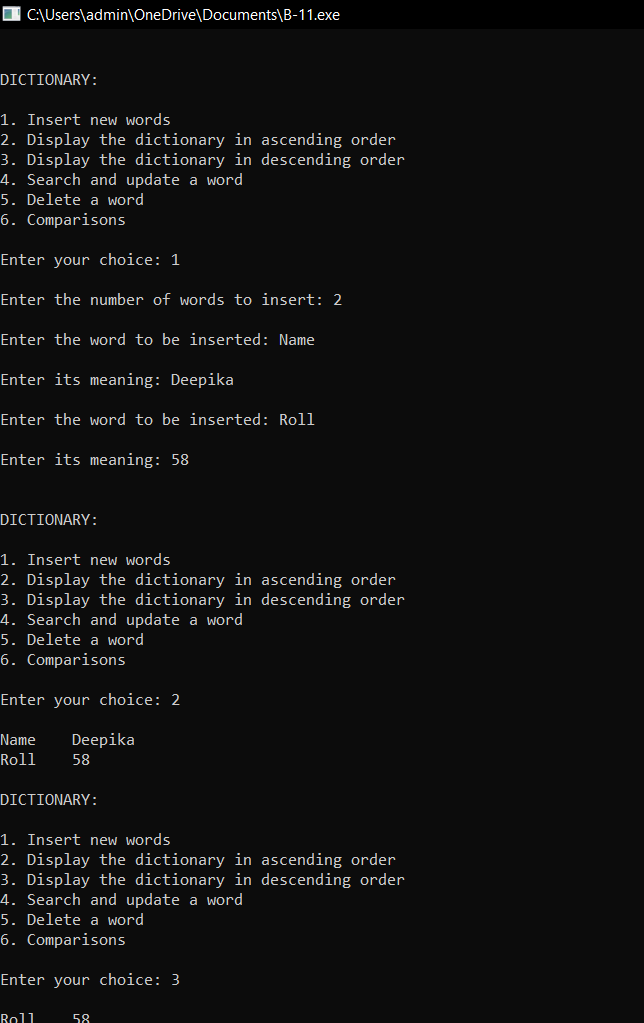
} while (choice < 7);

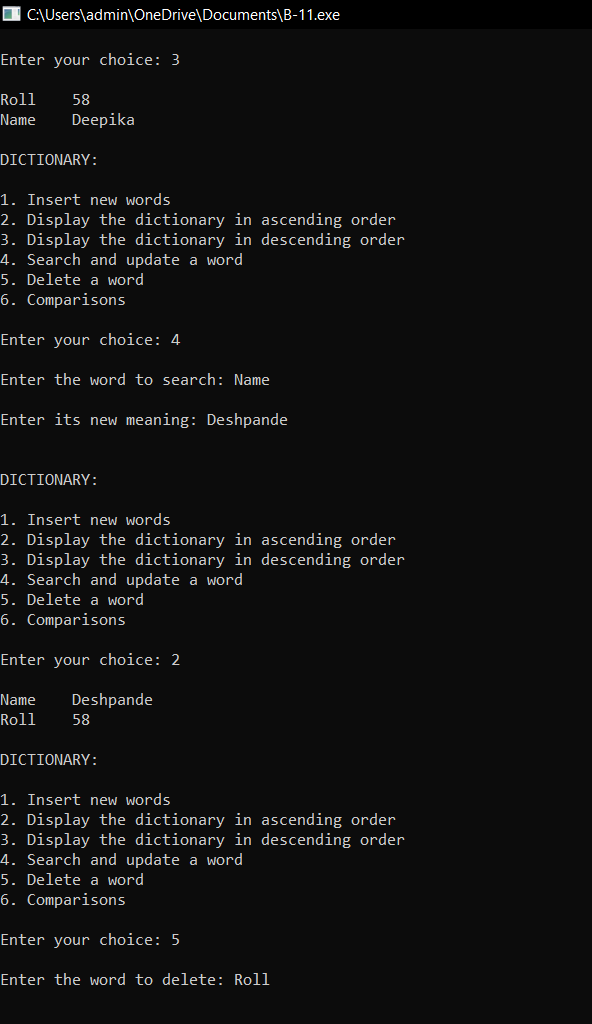
return 0;

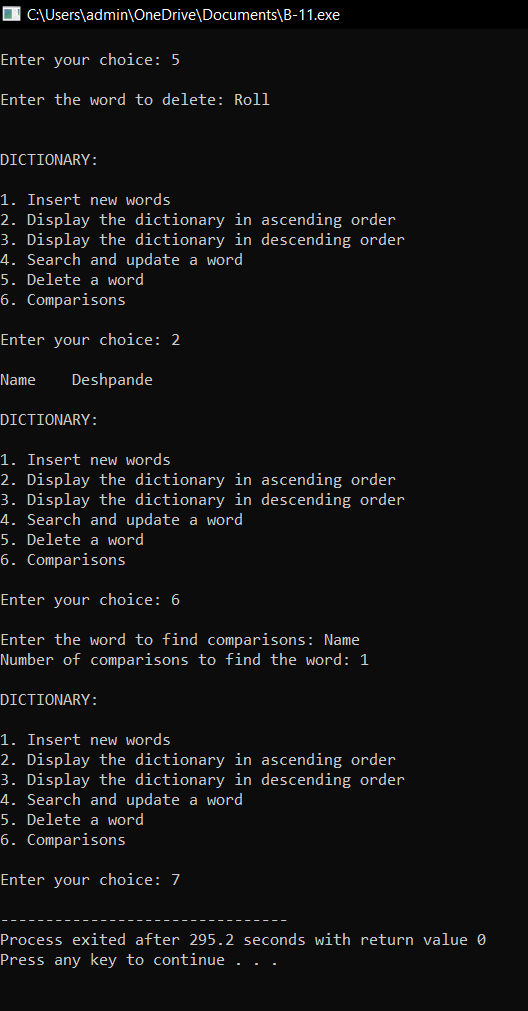
}

OUTPUT

|  |
| --- |
|  |







Practical No.-C 14

Q5) 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.

Code:

#include <iostream>

#include <queue>

using namespace std;

int adj\_mat[50][50] = {0, 0};

int visited[50] = {0};

void dfs(int s, int n, string arr[])

{

visited[s] = 1;

cout << arr[s] << " ";

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

{

if (adj\_mat[s][i] && !visited[i])

dfs(i, n, arr);

}

}

void bfs(int s, int n, string arr[])

{

bool visited[n];

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

visited[i] = false;

int v;

queue<int> bfsq;

if (!visited[s])

{

cout << arr[s] << " ";

bfsq.push(s);

visited[s] = true;

while (!bfsq.empty())

{

v = bfsq.front();

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

{

if (adj\_mat[v][i] && !visited[i])

{

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

visited[i] = true;

bfsq.push(i);

}

}

bfsq.pop();

}

}

}

int main()

{

cout << "Enter no. of cities: ";

int n, u;

cin >> n;

string cities[n];

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

{

cout << "Enter city #" << i << " (Airport Code): ";

cin >> cities[i];

}

cout << "\nYour cities are: " << endl;

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

cout << "city #" << i << ": " << cities[i] << endl;

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

{

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

{

cout << "Enter distance between " << cities[i] << " and " << cities[j] << " : ";

cin >> adj\_mat[i][j];

adj\_mat[j][i] = adj\_mat[i][j];

}

}

cout << endl;

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

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

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

{

cout << "\n"

<< cities[i];

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

cout << "\t" << adj\_mat[i][j] << "\t";

cout << endl;

}

cout << "Enter Starting Vertex: ";

cin >> u;

cout << "DFS: ";

dfs(u, n, cities);

cout << endl;

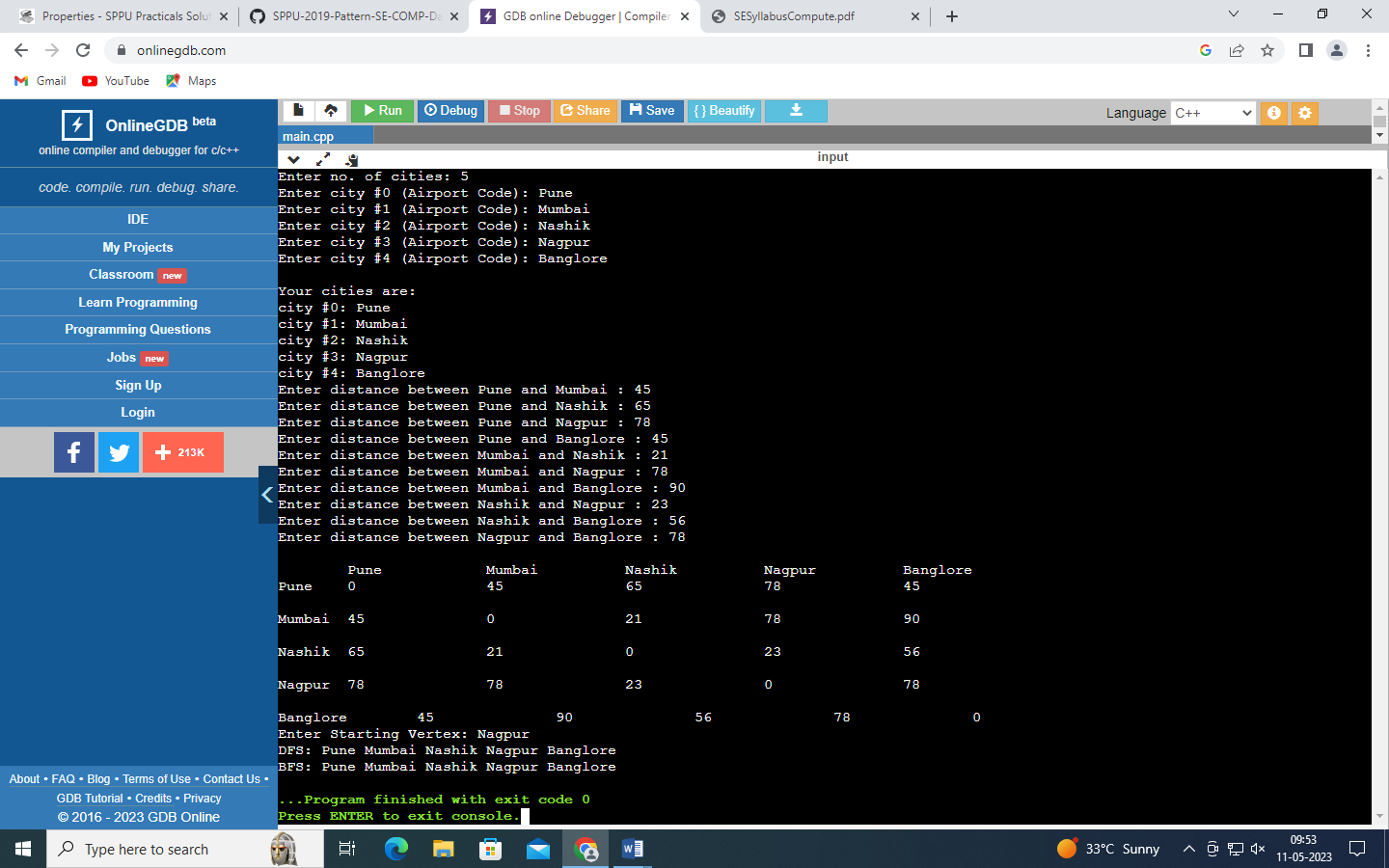
cout << "BFS: ";

bfs(u, n, cities);

return 0;

}

OUTPUT:



Experiment C-15

Q7.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.

CODE:

#include <iostream>

#include <limits.h>

using namespace std;

class Office {

int n;

int adjacent[10][10];

string office[10];

public:

void input ();

void display ();

void Prims ();

};

void Office::input () {

cout << "\nEnter no. of offices: ";

cin >> n;

cout << "\nEnter the names of offices: ";

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

cin >> office[i];

cout << "\nEnter the cost to connect the offices: \n";

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

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

if (i == j) {

adjacent[i][j] = 0;

continue;

}

cout << "Enter the cost to connect " << office[i] <<" and " << office[j]<< " : ";

cin >> adjacent[i][j];

adjacent[j][i] = adjacent[i][j];

}

}

void Office::display () {

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

cout << "\n";

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

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

}

}

}

void Office::Prims () {

int visit[n], minCost = 0, count = n - 1, minIndex, cost = 0;

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

visit[i] = 0;

cout << "\n\nShortest path: ";

visit[0]=1;

cout << office[0] << " -> ";

while (count--) {

minCost = INT\_MAX;

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

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

if (visit[i] == 1 && adjacent[i][j] != 0 && adjacent[i][j] < minCost && visit[j] == 0) {

minCost = adjacent[i][j];

minIndex = j;

}

}

}

visit[minIndex]=1;

cout << office[minIndex] << " -> ";

cost = cost + minCost;

}

cout << "End";

cout << "\nMinimum cost: "<<cost;

}

int main () {

Office o1;

int choice;

do {

cout << "\n\nMINIMUM SPANNING TREE"

<< "\n1. Input data"

<< "\n2. Display data"

<< "\n3. Calculate minimum cost"

<< "\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

o1.input ();

break;

case 2:

o1.display ();

break;

case 3:

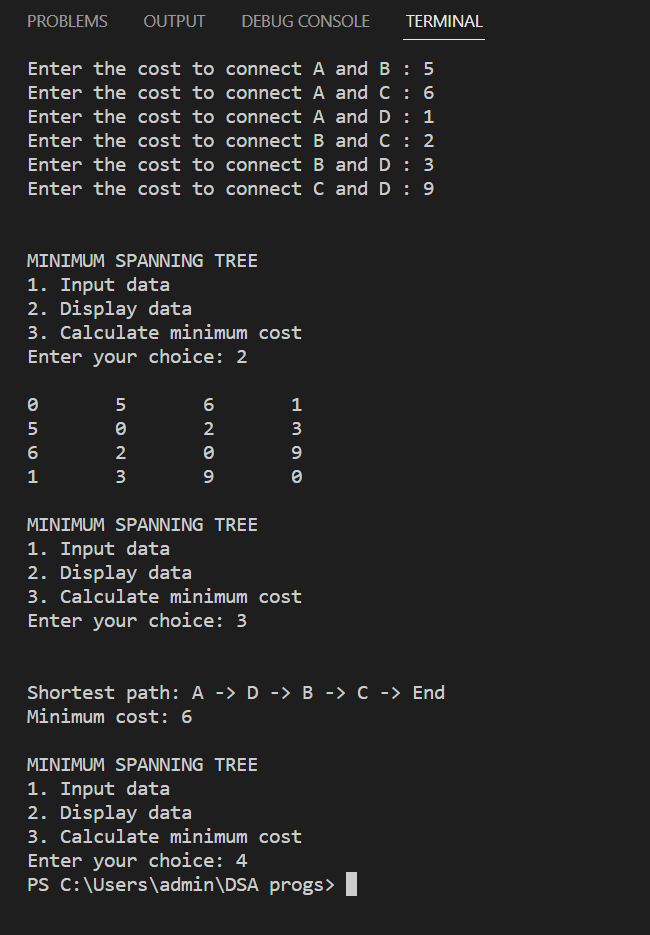
o1.Prims ();

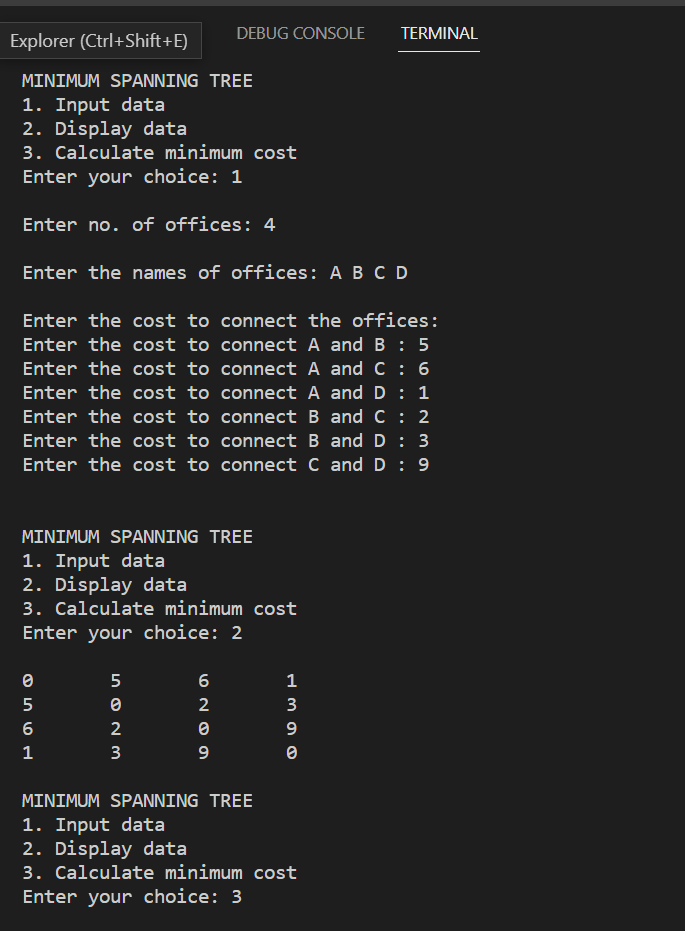
break; }

} while (choice != 4);

return 0;}

OUTPUT:





PRACTICAL No-D 18

AIM:

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;

#define SIZE 15

class OBST {

int prob[SIZE] = {}; //Probabilities with which we search for an element

int keys[SIZE] = {}; //Elements from which OBST is to be built

int weight[SIZE][SIZE] = {}; //Weight weight[i][j]’ of keys tree having root ’root[i][j]’

int cost[SIZE][SIZE] = {}; //Cost ‘cost[i][j] of keys tree having root ‘root[i][j]

int root[SIZE][SIZE] = {}; //represents root

int n; // number of nodes

public:

void get\_data();

int Min\_Value(int, int);

void build\_OBST();

void build\_tree();

void print(int [][SIZE], int);

};

/\* This function accepts the input data \*/

void OBST::get\_data() {

int i;

cout << "\nOptimal Binary Search Tree \n\nEnter the number of nodes: ";

cin >> n;

cout << "\nEnter " << n << " nodes: ";

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

cin >> keys[i];

cout << "\nEnter " << n << " probabilities: ";

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

cin >> prob[i];

}

/\* This function returns keys value in the range ‘r[i][j-1]’ to ‘r[i+1][j]’so

that the cost ‘cost[i][k-1]+cost[k][j]’is minimum \*/

int OBST::Min\_Value(int i, int j) {

int l, k;

int minimum = INT\_MAX;

for (l = root[i][j - 1]; l <= root[i + 1][j]; l++) {

if ((cost[i][l - 1] + cost[l][j]) < minimum) {

minimum = cost[i][l - 1] + cost[l][j];

k = l;

}

}

return k;

}

/\* This function builds the table from all the given probabilities It

basically computes cost,root,weight values \*/

void OBST::build\_OBST() {

int i, j, k, l;

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

//initialize

weight[i][i] = root[i][i] = cost[i][i] = 0;

//Optimal trees with one node

weight[i][i + 1] = cost[i][i + 1] = prob[i + 1];

root[i][i + 1] = i + 1;

}

weight[n][n] = root[n][n] = cost[n][n] = 0;

//Find optimal trees with ‘m’ nodes

for (l = 2; l <= n; l++) {

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

j = i + l;

weight[i][j] = weight[i][j - 1] + prob[j];

k = Min\_Value(i, j);

cost[i][j] = weight[i][j] + cost[i][k - 1] + cost[k][j];

root[i][j] = k;

}

}

cout << "\nCost are: \n";

print(cost, n);

cout << "\nRoot are: \n";

print(root, n);

}

/\* This function builds the tree from the tables made by the OBST function \*/

void OBST::build\_tree() {

int i, j, k;

int queue[20], front = -1, rear = -1;

cout << "\nThe Optimal Binary Search Tree For the Given Nodes Is…\n";

cout << "\nThe Root of this OBST is:: " << keys[root[0][n]];

cout << "\nThe Cost of this OBST is:: " << cost[0][n];

cout << "\n\n\tNODE\tLEFT CHILD\tRIGHT CHILD";

cout << "\n";

queue[++rear] = 0;

queue[++rear] = n;

while (front != rear) {

i = queue[++front];

j = queue[++front];

k = root[i][j];

cout << "\n\t" << keys[k];

if (root[i][k - 1] != 0) {

cout << "\t\t" << keys[root[i][k - 1]];

queue[++rear] = i;

queue[++rear] = k - 1;

}

else

cout << "\t\t";

if (root[k][j] != 0) {

cout << "\t" << keys[root[k][j]];

queue[++rear] = k;

queue[++rear] = j;

}

else

cout << "\t";

}

cout << "\n";

}

void OBST::print(int arr[][SIZE], int n) {

int i, j;

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

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

cout << arr[i][j] << '\t';

cout << '\n';

}

}

int main() {

OBST obj;

obj.get\_data();

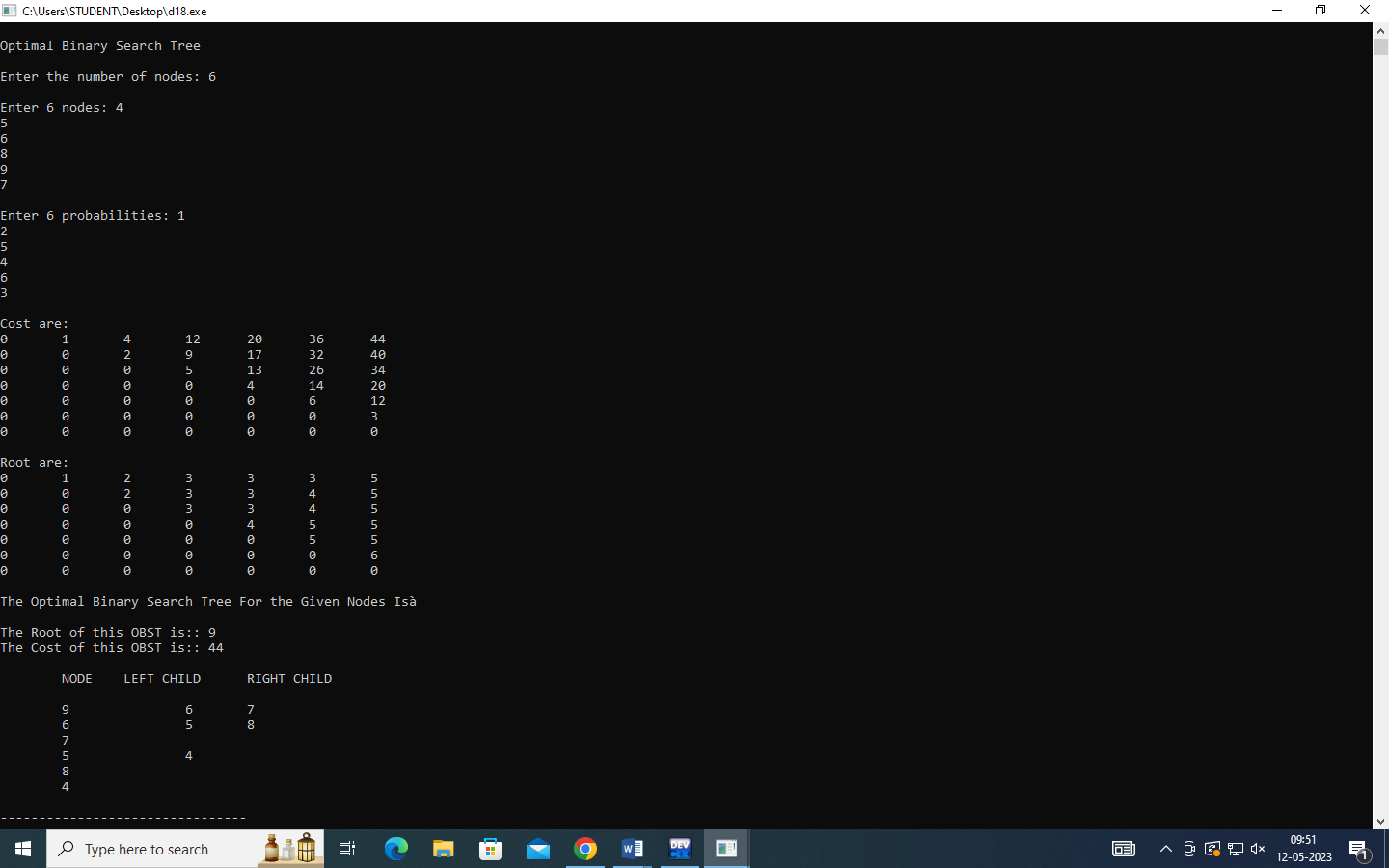
obj.build\_OBST();

obj.build\_tree();

return 0;

}

OUTPUT:



PRACTICAL NO-D19

|  |  |
| --- | --- |
|  | AIM :  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>  using namespace std;  struct AVLnode {  public:  int cWord;  string cMean;  AVLnode \*left,\*right;  int iHt;  };  class AVLtree {  public:  AVLnode \*Root;  AVLtree () {  Root = NULL;  }  AVLnode\* insert (AVLnode\*, int, string);  AVLnode\* deletE (AVLnode\*, int);  AVLnode\* LL (AVLnode\*);  AVLnode\* RR (AVLnode\*);  AVLnode\* LR (AVLnode\*);  AVLnode\* RL (AVLnode\*);  int height (AVLnode\*);  int bFactor (AVLnode\*);  void inOrder (AVLnode\*);  void preOrder (AVLnode\*);  };  AVLnode\* AVLtree::insert (AVLnode \*root, int nWord, string nMean) {  if (root == NULL) {  root = new AVLnode;  root -> left = root -> right = NULL;  root -> cWord = nWord;  root -> cMean = nMean;  root -> iHt = 0;  }  else if (root -> cWord != nWord) {  if (root -> cWord > nWord)  root -> left = insert (root -> left, nWord, nMean);  else  root -> right = insert (root -> right, nWord, nMean);  }  else  cout << "\nRedundant AVLnode\n";  root -> iHt = max(height(root -> left), height(root -> right)) + 1;  if (bFactor (root) == 2) {  if (root -> left -> cWord > nWord)  root = RR (root);  else  root = LR (root);  }  if (bFactor (root) == -2) {  if (root -> right -> cWord > nWord)  root = RL (root);  else  root = LL (root);  }  return root;  }  AVLnode \*AVLtree::deletE (AVLnode \*curr, int x) {  AVLnode \*temp;  if (curr == NULL) {  cout << "\nWord not present!\n";  return curr;  }  else if (x > curr -> cWord)  curr -> right = deletE (curr -> right, x);  else if (x < curr -> cWord)  curr -> left = deletE (curr -> left, x);  else if (curr -> right == NULL || curr -> left == NULL) {  curr = curr -> left ? curr -> left : curr -> right;  cout << "\nWord deleted Successfully!\n";  }  else {  temp = curr -> right;  while (temp -> left)  temp = temp -> left;  curr -> cWord = temp -> cWord;  curr -> right = deletE (curr -> right, temp -> cWord);  }  if (curr == NULL) return curr;  curr -> iHt = max(height(curr -> left), height(curr -> right)) + 1;  if (bFactor (curr) == 2) {  if (bFactor (curr -> left) >= 0)  curr = RR (curr);  else  curr = LR (curr);  }  if (bFactor (curr) == -2) {  if (bFactor (curr -> right) <= 0)  curr = LL (curr);  else  curr = RL (curr);  }  return (curr);  }  int AVLtree::height (AVLnode\* curr) {  if (curr == NULL)  return -1;  else  return curr -> iHt;  }  int AVLtree::bFactor (AVLnode\* curr) {  int lh = 0, rh = 0;  if (curr == NULL)  return 0;  else  return height(curr -> left) - height(curr -> right);  }  AVLnode\* AVLtree::RR (AVLnode\* curr) {  AVLnode\* temp = curr -> left;  curr -> left = temp -> right;  temp -> right = curr;  curr -> iHt = max(height(curr -> left), height(curr -> right)) + 1;  temp -> iHt = max(height(temp -> left), height(temp -> right)) + 1;  return temp;  }  AVLnode\* AVLtree::LL (AVLnode\* curr) {  AVLnode\* temp = curr -> right;  curr -> right = temp -> left;  temp -> left = curr;  curr -> iHt = max(height(curr -> left), height(curr -> right)) + 1;  temp -> iHt = max(height(temp -> left), height(temp -> right)) + 1;  return temp;  }  AVLnode\* AVLtree::RL (AVLnode\* curr) {  curr -> right = RR (curr -> right);  return LL (curr);  }  AVLnode\* AVLtree::LR (AVLnode\* curr) {  curr -> left = LL (curr -> left);  return RR (curr);  }  void AVLtree::inOrder (AVLnode\* curr) {  if (curr != NULL) {  inOrder (curr -> left);  cout << "\n\t" << curr -> cWord << "\t" << curr -> cMean;  inOrder (curr -> right);  }  }  void AVLtree::preOrder (AVLnode\* curr) {  if (curr != NULL) {  cout << "\n\t" << curr -> cWord << "\t" << curr -> cMean;  preOrder (curr -> left);  preOrder (curr -> right);  }  }  int main () {  int ch;  AVLtree avl;  AVLnode \*temp = NULL;  int word;  string mean;  cout << "\n--------------------------------------";  cout << "\n\tAVL TREE IMPLEMENTATION";  cout << "\n--------------------------------------";  do {  cout << "\n\t\tMENU";  cout << "\n1.Insert 2.Inorder 3.Delete 4.Exit";  cout << "\n--------------------------------";  cout << "\nEnter your choice: ";  cin >> ch;  switch (ch) {  case 1:  cout << "\nEnter Word: ";  cin >> word;  cout << "\nEnter Meaning: ";  cin >> mean;  avl.Root = avl.insert (avl.Root, word, mean);  break;  case 2:  cout << "\nInorder Traversal:\n\tWORD\tMEANING";  avl.inOrder (avl.Root);  cout << "\n\nPreorder Traversal:\n\tWORD\tMEANING";  avl.preOrder (avl.Root);  cout << '\n';  break;  case 3:  cout << "\nEnter the word to be deleted : ";  cin >> word;  avl.Root = avl.deletE (avl.Root, word);  break;  case 4:  exit (0);  }  } while (ch != 4);  return 0;  } |

Experiment E-22

Q10.Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure. Analyze the algorithm.

Code:

|  |
| --- |
| #include<iostream> |
|  |

|  |
| --- |
| using namespace std; |
|  |

|  |
| --- |
| class Heap |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int n; |
|  |

|  |
| --- |
| int \*minheap,\*maxheap; |
|  |

|  |
| --- |
| public: |
|  |

|  |
| --- |
| void get(); |
|  |

|  |
| --- |
| void displayMin(){cout<<"Minimum marks are :"<<minheap[0]<<endl;} |
|  |

|  |
| --- |
| void displayMax(){cout<<"Maximum marks are :"<<maxheap[0]<<endl;} |
|  |

|  |
| --- |
| void upAdjust(bool,int); |
|  |

|  |
| --- |
| }; |
|  |

|  |
| --- |
| void Heap::get() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| cout<<"Enter number of students."<<endl; |
|  |

|  |
| --- |
| cin>>n; |
|  |

|  |
| --- |
| int k; |
|  |

|  |
| --- |
| minheap=new int[n]; |
|  |

|  |
| --- |
| maxheap=new int[n]; |
|  |

|  |
| --- |
| cout<<"Enter marks of students."<<endl; |
|  |

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

|  |
| --- |
| { |
|  |

|  |
| --- |
| cin>>k; |
|  |

|  |
| --- |
| minheap[i]=k; |
|  |

|  |
| --- |
| upAdjust(0,i); |
|  |

|  |
| --- |
| maxheap[i]=k; |
|  |

|  |
| --- |
| upAdjust(1,i); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| void Heap::upAdjust(bool m,int l) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int s; |
|  |

|  |
| --- |
| if(!m) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| while(minheap[(l-1)/2]<minheap[l]) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| s=minheap[l]; |
|  |

|  |
| --- |
| minheap[l]=minheap[(l-1)/2]; |
|  |

|  |
| --- |
| minheap[(l-1)/2]=s; |
|  |

|  |
| --- |
| l=(l-1)/2; |
|  |

|  |
| --- |
| if(l==-1) |
|  |

|  |
| --- |
| break; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| while(maxheap[(l-1)/2]>maxheap[l]) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| s=maxheap[l]; |
|  |

|  |
| --- |
| maxheap[l]=maxheap[(l-1)/2]; |
|  |

|  |
| --- |
| maxheap[(l-1)/2]=s; |
|  |

|  |
| --- |
| l=(l-1)/2; |
|  |

|  |
| --- |
| if(l==-1) |
|  |

|  |
| --- |
| break; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| main() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| Heap H; |
|  |

|  |
| --- |
| H.get(); |
|  |

|  |
| --- |
| H.displayMin(); |
|  |

|  |
| --- |
| H.displayMax(); |
|  |

|  |
| --- |
| return(0); |
|  |

|  |
| --- |
| } |
|  |
| OUTPUT:  Screenshot (7) |

PRACTICAL NO-F23

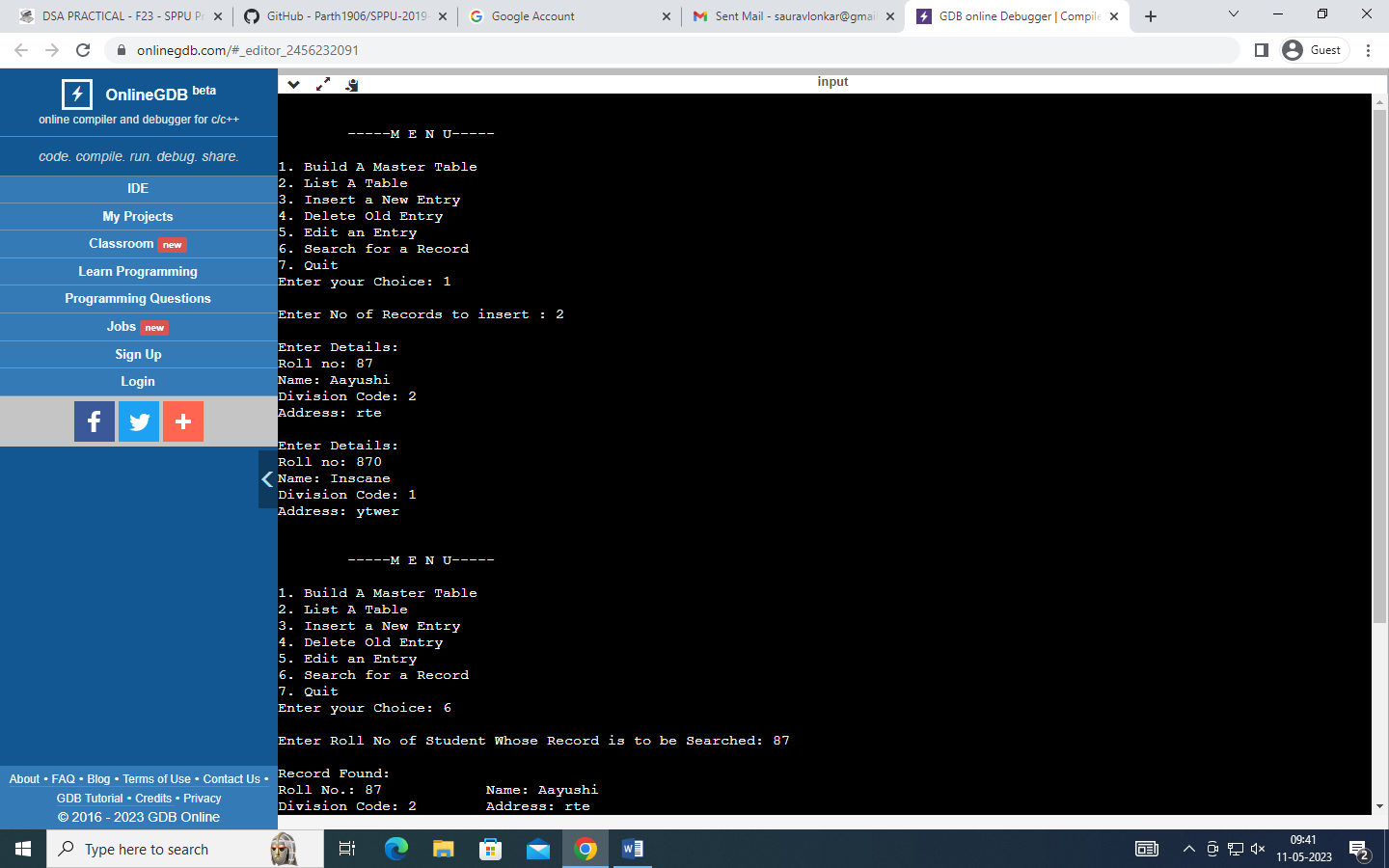
AIM:

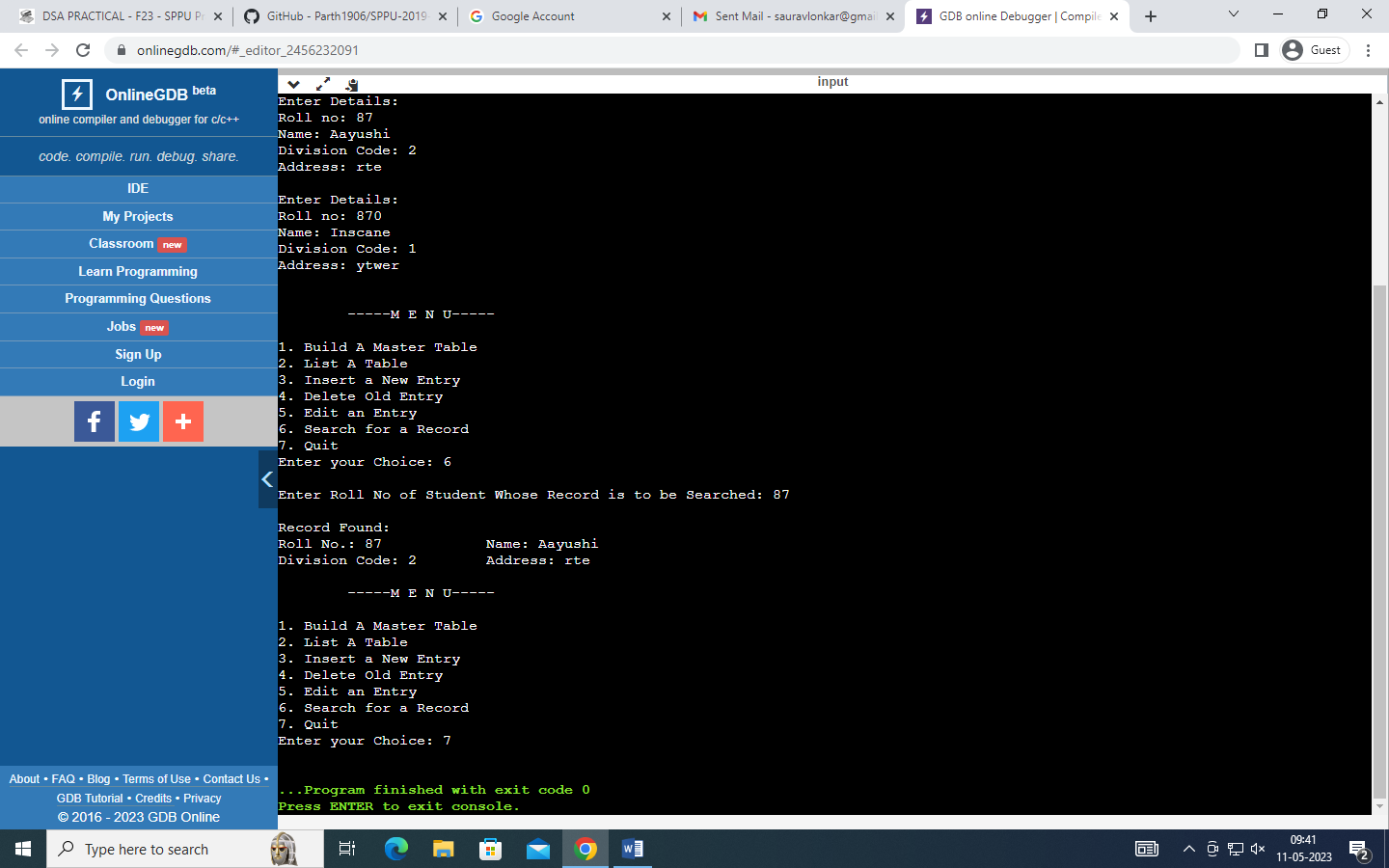
Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays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

CODE:

|  |
| --- |
| #include <iostream> |
|  | #include <fstream> |
|  | using namespace std; |
|  |  |
|  | class Record { |
|  | int rollno; |
|  | string name; |
|  | int division; |
|  | string address; |
|  |  |
|  | public: |
|  | Record(); |
|  | int getRollno(); |
|  | void getData(); |
|  | void putData(); |
|  | }; |
|  |  |
|  | Record::Record() { |
|  | rollno = 0; |
|  | name = ' '; |
|  | address = ' '; |
|  | division = 0; |
|  | } |
|  |  |
|  | int Record::getRollno() { |
|  | return (rollno); |
|  | } |
|  |  |
|  | void Record::getData() { |
|  | cout << "\nEnter Details: "; |
|  | cout << "\nRoll no: "; |
|  | cin >> rollno; |
|  | cout << "Name: "; |
|  | cin >> name; |
|  | cout << "Division Code: "; |
|  | cin >> division; |
|  | cout << "Address: "; |
|  | cin >> address; |
|  | } |
|  |  |
|  | void Record::putData() { |
|  | cout << "\nRoll No.: "; |
|  | cout << rollno; |
|  | cout << "\t\tName: "; |
|  | cout << name; |
|  | cout << "\nDivision Code: "; |
|  | cout << division; |
|  | cout << "\tAddress: "; |
|  | cout << address; |
|  | } |
|  |  |
|  | class File { |
|  | ifstream fin; |
|  | ofstream fout; |
|  | fstream fs; |
|  |  |
|  | public: |
|  | void insert(); |
|  | void display(); |
|  | void search(int); |
|  | int Delete(int); |
|  | int edit(int); |
|  | }; |
|  |  |
|  | void File::insert() { |
|  | Record r; |
|  | r.getData(); |
|  | fout.open("StudentDB", ios::ate | ios::app); |
|  | fout.write((char \*)&r, sizeof(r)); |
|  | fout.close(); |
|  | } |
|  |  |
|  | void File::display() { |
|  | Record r; |
|  | fin.open("StudentDB"); |
|  | fin.seekg(0, ios::beg); |
|  | while (fin.read((char \*)&r, sizeof(r))) |
|  | r.putData(); |
|  |  |
|  | fin.close(); |
|  | } |
|  |  |
|  | void File::search(int rollno) { |
|  | Record r; |
|  | int flag = 0; |
|  | fin.open("StudentDB"); |
|  | fin.seekg(0, ios::beg); |
|  | while (fin.read((char \*)&r, sizeof(r))) { |
|  | if (r.getRollno() == rollno) { |
|  | flag = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | fin.close(); |
|  | if (flag == 1) { |
|  | cout << "\nRecord Found:"; |
|  | r.putData(); |
|  | } |
|  | else |
|  | cout << "\nRecord not Found "; |
|  | } |
|  |  |
|  | int File::Delete(int rollno) { |
|  | Record r; |
|  | int flag = 0; |
|  | fin.open("StudentDB"); |
|  | fout.open("Temp", ios::ate | ios::app); |
|  | fin.seekg(0, ios::beg); |
|  | while (fin.read((char \*)&r, sizeof(r))) { |
|  | if (r.getRollno() == rollno) { |
|  | flag = 1; |
|  | } |
|  | else { |
|  | fout.write((char \*)&r, sizeof(r)); |
|  | } |
|  | } |
|  | fin.close(); |
|  | fout.close(); |
|  | remove("StudentDB"); |
|  | rename("Temp", "StudentDB"); |
|  | return (flag); |
|  | } |
|  |  |
|  | int File::edit(int rollno) { |
|  | Record r; |
|  | int flag = 0; |
|  | fs.open("StudentDB"); |
|  | fs.seekg(0, ios::beg); |
|  | while (fs.read((char \*)&r, sizeof(r))) { |
|  | if (r.getRollno() == rollno) { |
|  | flag = 1; |
|  | cout << "\nEnter New Details: "; |
|  | r.getData(); |
|  | fs.seekp((int)fs.tellg() - sizeof(r), ios::beg); |
|  | fs.write((char \*)&r, sizeof(r)); |
|  | } |
|  | } |
|  | fs.close(); |
|  | return (flag); |
|  | } |
|  |  |
|  | int main() { |
|  | File f; |
|  | int ch, n, i, flag = 0; |
|  | do { |
|  | cout << "\n\n\t-----M E N U-----"; |
|  | cout << "\n\n1. Build A Master Table"; |
|  | cout << "\n2. List A Table"; |
|  | cout << "\n3. Insert a New Entry"; |
|  | cout << "\n4. Delete Old Entry"; |
|  | cout << "\n5. Edit an Entry"; |
|  | cout << "\n6. Search for a Record"; |
|  | cout << "\n7. Quit"; |
|  | cout << "\nEnter your Choice: "; |
|  | cin >> ch; |
|  | switch (ch) { |
|  | case 1: |
|  | if (flag == 0) { |
|  | cout << "\nEnter No of Records to insert : "; |
|  | cin >> n; |
|  | for (i = 0; i < n; i++) { |
|  | f.insert(); |
|  | } |
|  | flag = 1; |
|  | } |
|  | else { |
|  | cout << "\nSorry.. Table is Already build... \n If want to add record please select Insert a New Entry in option....."; |
|  | } |
|  | break; |
|  | case 2: |
|  | f.display(); |
|  | break; |
|  | case 3: |
|  | f.insert(); |
|  | break; |
|  | case 4: |
|  | cout << "\nEnter Roll No of Student Whose Record is to be Deleted: "; |
|  | cin >> n; |
|  | i = f.Delete(n); |
|  | if (i == 1) |
|  | cout << "\nRecord Deleted Successfully"; |
|  | else |
|  | cout << "\nRecord not Found"; |
|  | break; |
|  | case 5: |
|  | cout << "\nEnter Roll No of Student Whose Record is to be Edit: "; |
|  | cin >> n; |
|  | i = f.edit(n); |
|  | if (i == 1) |
|  | cout << "\nRecord Modified Successfully"; |
|  | else |
|  | cout << "\nRecord not Found"; |
|  | break; |
|  | case 6: |
|  | cout << "\nEnter Roll No of Student Whose Record is to be Searched: "; |
|  | cin >> n; |
|  | f.search(n); |
|  | break; |
|  | case 7: |
|  | break; |
|  | default: |
|  | cout << "\nEnter Valid Choice....."; |
|  | } |
|  | } while (ch != 7); |
|  | return (0); |
|  | } |

OUTPUT :





PRACTICAL NO-F24

AIM:

Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.

CODE:

|  |
| --- |
| #include <iostream> |
|  | #include <fstream> |
|  | using namespace std; |
|  |  |
|  | class Record { |
|  | int id; |
|  | string name; |
|  | int salary; |
|  | string designation; |
|  |  |
|  | public: |
|  | Record(); |
|  | int getIdno(); |
|  | void getData(); |
|  | void putData(); |
|  | }; |
|  |  |
|  | Record::Record() { |
|  | id = 0; |
|  | name = ' '; |
|  | designation = ' '; |
|  | salary = 0; |
|  | } |
|  |  |
|  | int Record::getIdno() { |
|  | return (id); |
|  | } |
|  |  |
|  | void Record::getData() { |
|  | cout << "\nEnter Details: "; |
|  | cout << "\nId no: "; |
|  | cin >> id; |
|  | cout << "Name: "; |
|  | cin >> name; |
|  | cout << "Salary: "; |
|  | cin >> salary; |
|  | cout << "Designation: "; |
|  | cin >> designation; |
|  | } |
|  |  |
|  | void Record::putData() { |
|  | cout << "\nId No.: "; |
|  | cout << id; |
|  | cout << "\t\tName: "; |
|  | cout << name; |
|  | cout << "\nSalary: "; |
|  | cout << salary; |
|  | cout << "\tDesignation: "; |
|  | cout << designation; |
|  | } |
|  |  |
|  | class File { |
|  | ifstream fin; |
|  | ofstream fout; |
|  | fstream fs; |
|  |  |
|  | public: |
|  | void insert(); |
|  | void display(); |
|  | void search(int); |
|  | int Delete(int); |
|  | int edit(int); |
|  | }; |
|  |  |
|  | void File::insert() { |
|  | Record r; |
|  | r.getData(); |
|  | fout.open("EmployeeDB", ios::ate | ios::app); |
|  | fout.write((char \*)&r, sizeof(r)); |
|  | fout.close(); |
|  | } |
|  |  |
|  | void File::display() { |
|  | Record r; |
|  | fin.open("EmployeeDB"); |
|  | fin.seekg(0, ios::beg); |
|  | while (fin.read((char \*)&r, sizeof(r))) |
|  | r.putData(); |
|  |  |
|  | fin.close(); |
|  | } |
|  |  |
|  | void File::search(int id) { |
|  | Record r; |
|  | int flag = 0; |
|  | fin.open("EmployeeDB"); |
|  | fin.seekg(0, ios::beg); |
|  | while (fin.read((char \*)&r, sizeof(r))) { |
|  | if (r.getIdno() == id) { |
|  | flag = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | fin.close(); |
|  | if (flag == 1) { |
|  | cout << "\nRecord Found:"; |
|  | r.putData(); |
|  | } |
|  | else |
|  | cout << "\nRecord not Found "; |
|  | } |
|  |  |
|  | int File::Delete(int id) { |
|  | Record r; |
|  | int flag = 0; |
|  | fin.open("EmployeeDB"); |
|  | fout.open("Temp", ios::ate | ios::app); |
|  | fin.seekg(0, ios::beg); |
|  | while (fin.read((char \*)&r, sizeof(r))) { |
|  | if (r.getIdno() == id) { |
|  | flag = 1; |
|  | } |
|  | else { |
|  | fout.write((char \*)&r, sizeof(r)); |
|  | } |
|  | } |
|  | fin.close(); |
|  | fout.close(); |
|  | remove("EmployeeDB"); |
|  | rename("Temp", "EmployeeDB"); |
|  | return (flag); |
|  | } |
|  |  |
|  | int File::edit(int id) { |
|  | Record r; |
|  | int flag = 0; |
|  | fs.open("EmployeeDB"); |
|  | fs.seekg(0, ios::beg); |
|  | while (fs.read((char \*)&r, sizeof(r))) { |
|  | if (r.getIdno() == id) { |
|  | flag = 1; |
|  | cout << "\nEnter New Details: "; |
|  | r.getData(); |
|  | fs.seekp((int)fs.tellg() - sizeof(r), ios::beg); |
|  | fs.write((char \*)&r, sizeof(r)); |
|  | } |
|  | } |
|  | fs.close(); |
|  | return (flag); |
|  | } |
|  |  |
|  | int main() { |
|  | File f; |
|  | int ch, n, i, flag = 0; |
|  | do { |
|  | cout << "\n\n\t-----M E N U-----"; |
|  | cout << "\n\n1. Build A Master Table"; |
|  | cout << "\n2. List A Table"; |
|  | cout << "\n3. Insert a New Entry"; |
|  | cout << "\n4. Delete Old Entry"; |
|  | cout << "\n5. Edit an Entry"; |
|  | cout << "\n6. Search for a Record"; |
|  | cout << "\n7. Quit"; |
|  | cout << "\nEnter your Choice: "; |
|  | cin >> ch; |
|  | switch (ch) { |
|  | case 1: |
|  | if (flag == 0) { |
|  | cout << "\nEnter No of Records to insert : "; |
|  | cin >> n; |
|  | for (i = 0; i < n; i++) { |
|  | f.insert(); |
|  | } |
|  | flag = 1; |
|  | } |
|  | else { |
|  | cout << "\nSorry.. Table is Already build... \n If want to add record please select Insert a New Entry in option....."; |
|  | } |
|  | break; |
|  | case 2: |
|  | f.display(); |
|  | break; |
|  | case 3: |
|  | f.insert(); |
|  | break; |
|  | case 4: |
|  | cout << "\nEnter Id No of Employee Whose Record is to be Deleted: "; |
|  | cin >> n; |
|  | i = f.Delete(n); |
|  | if (i == 1) |
|  | cout << "\nRecord Deleted Successfully"; |
|  | else |
|  | cout << "\nRecord not Found"; |
|  | break; |
|  | case 5: |
|  | cout << "\nEnter Id No of Employee Whose Record is to be Edit: "; |
|  | cin >> n; |
|  | i = f.edit(n); |
|  | if (i == 1) |
|  | cout << "\nRecord Modified Successfully"; |
|  | else |
|  | cout << "\nRecord not Found"; |
|  | break; |
|  | case 6: |
|  | cout << "\nEnter Id No of Employee Whose Record is to be Searched: "; |
|  | cin >> n; |
|  | f.search(n); |
|  | break; |
|  | case 7: |
|  | break; |
|  | default: |
|  | cout << "\nEnter Valid Choice....."; |
|  | } |
|  | } while (ch != 7); |
|  | return (0); |
|  | } |

OUTPUT :

