A2-imp\_func\_of\_dict

class Node:

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

self.key = key

self.value = value

self.next = None

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

return f"({self.key}, {self.value})"

class HashTable:

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

self.size = size

self.table = [None] \* size

self.replacement = replacement

def hash(self, key):

return hash(key) % self.size

def insert(self, key, value):

index = self.hash(key)

if self.table[index] is None:

self.table[index] = Node(key, value)

else:

if self.replacement:

node = self.table[index]

while node.next is not None and node.key != key:

node = node.next

if node.key == key:

node.value = value

else:

node.next = Node(key, value)

else:

node = self.table[index]

while node.next is not None:

node = node.next

node.next = Node(key, value)

def find(self, key):

index = self.hash(key)

node = self.table[index]

while node is not None:

if node.key == key:

return node.value

node = node.next

return None

def delete(self, key):

index = self.hash(key)

node = self.table[index]

prev\_node = None

while node is not None:

if node.key == key:

if prev\_node is None:

self.table[index] = node.next

else:

prev\_node.next = node.next

return

prev\_node = node

node = node.next

def display(self):

for i in range(self.size):

print(f"Bucket {i}: ", end="")

node = self.table[i]

while node is not None:

print(node, end=" -> ")

node = node.next

print("None")

# Example usage with user input

size = int(input("Enter hash table size: "))

replacement = input("Do you want to use replacement? (y/n): ").lower() == 'y'

ht = HashTable(size, replacement)

while True:

print("1. Insert")

print("2. Find")

print("3. Delete")

print("4. Display")

print("5. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

key = input("Enter key: ")

value = input("Enter value: ")

ht.insert(key, value)

print("Key-value pair inserted.")

elif choice == 2:

key = input("Enter key: ")

value = ht.find(key)

if value is not None:

print(f"Value for key {key}: {value}")

else:

print(f"Key {key} not found.")

elif choice == 3:

key = input("Enter key: ")

ht.delete(key)

print("Key-value pair deleted.")

elif choice == 4:

ht.display()

elif choice == 5:

print("Exited Successfully!")

break

else:

print("Invalid choice.")

A1-telephone\_book

class Client:

def \_\_init\_\_(self, name, phone\_number):

self.name = name

self.phone\_number = phone\_number

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.table = [[] for \_ in range(size)]

def hash\_function(self, key):

return sum(ord(char) for char in key) % self.size

def insert(self, client):

key = client.name

index = self.hash\_function(key)

self.table[index].append(client)

def search(self, key):

index = self.hash\_function(key)

comparisons = 0

for client in self.table[index]:

comparisons += 1

if client.name == key:

return comparisons, client.phone\_number

return comparisons, None

# Example usage

clients = [

Client("Jerome Smith", "1234567890"),

Client("Mike Ross", "0987654321"),

Client("Thomas Shelby", "9876543210"),

Client("Harry Lewis", "4567890123"),

]

hash\_table\_chaining = HashTable(10)

hash\_table\_probing = HashTable(10)

# Insert clients into hash tables

for client in clients:

hash\_table\_chaining.insert(client)

hash\_table\_probing.insert(client)

# Search for telephone numbers

search\_names = ["Jerome Smith", "Thomas Shelby", "Harry Lewis"]

total\_comparisons\_chaining = 0

total\_comparisons\_probing = 0

print("Separate Chaining:")

for name in search\_names:

comparisons, phone\_number = hash\_table\_chaining.search(name)

total\_comparisons\_chaining += comparisons

if phone\_number:

print(f"Found {name}: {phone\_number}")

else:

print(f"{name} not found")

print("Linear Probing:")

for name in search\_names:

comparisons, phone\_number = hash\_table\_probing.search(name)

total\_comparisons\_probing += comparisons

if phone\_number:

print(f"Found {name}: {phone\_number}")

else:

print(f"{name} not found")

print("Total comparisons (Separate Chaining):", total\_comparisons\_chaining)

print("Total comparisons (Linear Probing):", total\_comparisons\_probing)

D19-Height\_balance\_tree

#include<iostream>

using namespace std;

class node

{

public:

string key;

string meaning;

node \*left;

node \*right;

};

class AVL

{

node \*root;

public:

AVL()

{

root=NULL;

}

void create();

node\* insert(node \*cur,node \*temp);

node\* balance(node \*temp);

int dif(node \*temp);

int height(node \*temp);

node\* LL(node \*par);

node\* RR(node \*par);

node\* LR(node \*par);

node\* RL(node \*par);

void ascending(node \*temp);

node\* delete\_n(node \*root,string key1);

void deleten();

node\* extractmin(node \*t);

void descending(node \*temp);

void display();

void comparisons (node\* p, string key);

void search\_value();

};

void AVL::create()

{

char answer;

node \*temp;

do

{

temp=new node();

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

cin>>temp->key;

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

cin>>temp->meaning;

temp->left=temp->right=NULL;

root=insert(root,temp);

cout<<"\n Do you want to add another word?(y/n)";

cin>>answer;

}

while(answer=='y'||answer=='Y');

}

node\* AVL::insert(node\* cur,node\* temp)

{

if(cur==NULL)

{

return temp;

}

if(temp->key<cur->key)

{

cur->left=insert(cur->left,temp);

cur=balance(cur);

}

else if(temp->key>cur->key)

{

cur->right=insert(cur->right,temp);

cur=balance(cur);

}

return cur;

}

node\* AVL::balance(node \*temp)

{

int bal;

bal=dif(temp);

if(bal>=2)

{

if(dif(temp->left)<0)

temp=LR(temp);

else

temp=LL(temp);

}

else if(bal<=-2)

{

if(dif(temp->right)<0)

temp=RR(temp);

else

temp=RL(temp);

}

return temp;

}

int AVL::dif(node \*temp)

{

int l,r;

l=height(temp->left);

r=height(temp->right);

return(l-r);

}

int AVL::height(node \*temp)

{

if(temp==NULL)

return(-1);

else

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

}

node\* AVL::LL(node \*par)

{

node \*temp,\*temp1;

temp=par->left;

temp1=temp->right;

temp->right=par;

par->left=temp1;

return temp;

}

node\* AVL::RR(node \*par)

{

node \*temp,\*temp1;

temp=par->right;

temp1=temp->left;

temp->left=par;

par->right=temp1;

return temp;

}

node\* AVL::LR(node \*par)

{

par->left=RR(par->left);

return(LL(par));

}

node\* AVL::RL(node \*par)

{

par->right=LL(par->right);

return(RR(par));

}

void AVL::ascending(node \*temp)

{

if(temp!=NULL)

{

ascending(temp->left);

cout<<"\n\t"<<temp->key<<" : "<<temp->meaning;

ascending(temp->right);

}

}

void AVL::descending(node \*temp)

{

if(temp!=NULL)

{

descending(temp->right);

cout<<"\n\t"<<temp->key<<" : "<<temp->meaning;

descending(temp->left);

}

}

void AVL::display()

{

cout<<"\n The keywords in ascending order are : \n";

ascending(root);

cout<<"\n The keywords in descending order are : \n";

descending(root);

}

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

int count = 0;

while (p != NULL) {

if (key < p -> key) {

count++;

p = p -> left;

}

else if (key > p -> key) {

count++;

p = p -> right;

}

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

count++;

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

return ;

}

}

cout << "\nWord not found!";

}

void AVL::search\_value()

{

string key2;

cout<<"\n Enter the keyword you wish to search : ";

cin>>key2;

comparisons(root,key2);

}

node\* AVL::delete\_n(node\* cur,string key1)

{

if ( !cur)

return cur;

if ( key1 < cur->key )

cur->left = delete\_n(cur->left, key1);

else if( key1 > cur->key )

cur->right = delete\_n(cur->right, key1);

else

{

node \*l = cur->left;

node \*r = cur->right;

delete cur;

if ( !r )

return l;

node \*m=r;

while(m->left)

m=m->left;

m->right = extractmin(r);

m->left = l;

return balance(m);

}

return balance(cur);

}

node\* AVL::extractmin(node \*t)

{

if ( !t->left )

return t->right;

t->left = extractmin(t->left);

return balance(t);

}

void AVL::deleten()

{

string key;

cout<<"\n Enter the keyword to be deleted : ";

cin>>key;

root=delete\_n(root,key);

}

int main()

{

char c;

int ch;

AVL a;

do

{

cout<<"\*\*\*\*\*\*\*\*\*\*\*";

cout<<"\n 1.Insert a keyword in AVL tree.";

cout<<"\n 2.Display the AVL tree.";

cout<<"\n 3.Search a keyword";

cout<<"\n 4.Delete a keyword.";

cout<<"\n Enter your choice : ";

cin>>ch;

switch(ch)

{

case 1 : a.create();

break;

case 2 : a.display();

break;

case 3 : a.search\_value();

break;

case 4 : a.deleten();

break;

default : cout<<"\n Wrong choice ! ";

}

cout<<"\n Do you want to continue? (y/n): ";

cin>>c;

}

while(c=='y'||c=='Y');

return 0;

}

C24-emplye\_db

#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);

}

C23-std\_db

#include<iostream>

#include<fstream>

#include<string.h>

using namespace std;

class student

{

int rollno,divno;

string name,address;

public:

student()

{

rollno=0;

divno=0;

name=' ';

address=' ';

}

void putdata();

void getdata();

int getRollno();

};

void student::getdata()

{

cout<<"\nEnter the name of student-->";

cin>>name;

cout<<"\nEnter the roll no of student-->";

cin>>rollno;

cout<<"\nEnter the div no -->";

cin>>divno;

cout<<"\nEnter the home address-->";

cin>>address;

}

void student::putdata()

{

cout<<"\nThe name--> "<<name;

cout<<"\nThe Division no--> "<<divno;

cout<<"\nThe Roll No --> "<<rollno;

cout<<"\nThe address --> "<<address<<endl;

}

int student::getRollno()

{

return rollno;

}

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()

{

student s;

s.getdata();

fout.open("stuDB",ios::ate | ios::app);

fout.write((char\*)&s,sizeof(s));

fout.close();

}

void file::display()

{

student s;

fin.open("stuDB");

fin.seekg(0,ios::beg);

while(fin.read((char\*)&s,sizeof(s)))

{

s.putdata();

}

fin.close();

}

void file::search(int roll)

{

student s;

int flag=0;

fin.open("stuDB");

fin.seekg(0,ios::beg);

while(fin.read((char\*)&s,sizeof(s)))

{

if(s.getRollno()==roll)

{

flag=1;

break;

}

}

if(flag==0)

{

cout<<"\nRecord Not Found\n";

}

else if(flag==1)

{

cout<<"\nRecord Found\n";

s.putdata();

}

fin.close();

}

int file::Delete(int roll)

{

student s;

fin.open("stuDB");

fout.open("temp",ios::ate | ios::app);

fin.seekg(0,ios::beg);

int flag=0;

while(fin.read((char\*)&s,sizeof(s)))

{

if(s.getRollno()==roll)

{

flag=1;

}

else

{

fout.write((char\*)&s,sizeof(s));

}

}

fout.close();

fin.close();

remove("stuDB");

rename("temp","stuDB");

return flag;

}

int file::edit(int roll)

{

student s;

fs.open("stuDB");

fs.seekg(0,ios::beg);

int flag=0;

while(fs.read((char\*)&s,sizeof(s)))

{

if(s.getRollno()==roll)

{

flag=1;

cout<<"\nEnter new Details-->\n";

s.getdata();

fs.seekp((int)fs.tellg()-sizeof(s),ios::beg);

fs.write((char\*)&s,sizeof(s));

}

}

fs.close();

return flag;

}

int main()

{

file f;

int ch,roll;

int flag=0;

do

{

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MENU\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout<<"\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout<<"1.List Table\n2.Display\n3.Insert\n4.Search\n5.Delete\n6.Edit\nEnter your choice-->";

cin>>ch;

switch(ch)

{

case 1:

{

int n;

cout<<"\nEnter the number of records you want to insert-->";

cin>>n;

if(flag==0)

{

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

{

f.insert();

}

flag=1;

}

else if(flag==1)

{

cout<<"\nDatabase is already listed if you want to add record then plz choose option 3.\n";

}

break;

}

case 2:

{

f.display();

break;

}

case 3:

{

f.insert();

break;

}

case 4:

{

cout<<"\nEnter the roll no of student you wants to searched-->";

cin>>roll;

f.search(roll);

break;

}

case 5:

{

cout<<"\nEnter the roll no of student you wants to delete-->";

cin>>roll;

int res=f.Delete(roll);

if(res==1)

{

cout<<"\nRecord Deleted Successfully\n";

}

else if(res==0)

{

cout<<"\nRecord Not Found\n";

}

break;

}

case 6:

{

cout<<"\nEnter the roll no of student you wants to edit-->";

cin>>roll;

int res=f.edit(roll);

if(res==1)

{

cout<<"\nRecord Edited Successfully\n";

}

else if(res==0)

{

cout<<"\nRecord Not Found\n";

}

break;

}

default:

{

cout<<"\nInvalid Choice\n";

exit(0);

}

}

}while(ch!=7);

return 0;

}

C22-Marks\_obtn\_heap

#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;

}

C18-given\_seq\_obst

#include<iostream>

#include<limits.h>

#define SIZE 15

using namespace std;

class OBST

{

private:

int keys[SIZE]={};

int prob[SIZE]={};

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

int weight[SIZE][SIZE]={};

int cost[SIZE][SIZE]={};

int n;

public:

void getData();

int Min\_Value(int i,int j);

void bulid\_OBST();

void build\_Tree();

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

};

void OBST::getData()

{

cout<<"\nEnter the number of keys to be inserted-->";

cin>>n;

cout<<"\nEnter the all keys";

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

{

cin>>keys[i];

}

cout<<"\nEnter there corrrsponding probalbilities";

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

{

cin>>prob[i];

}

}

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

{

int k,l;

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;

}

void OBST::bulid\_OBST()

{

int i,j,k,l;

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

{

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

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

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

}

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

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

{

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

}

void OBST::build\_Tree()

{

int i,j,k;

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

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*OPTIMAL BINARY SEARCH TREE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout<<"The root of tree is -->"<<keys[root[0][n]];

cout<<"\nThe cost of root is -->"<<cost[0][n];

cout<<"\n\n\tRoot\tLEFT child\tRIGHT child\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)

{

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

{

cout<<"\n";

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

{

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

}

}

}

int main()

{

OBST t;

t.getData();

t.bulid\_OBST();

t.build\_Tree();

return 0;

}

C15-several\_business\_offices\_min\_cost

#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 : "

<< "\n4. Exit"

<< "\nEnter your choice : ";

cin >> choice;

switch (choice) {

case 1:

o1.input ();

break;

case 2:

o1.display ();

break;

case 3:

o1.Prims ();

break;

case 4:

cout << "Exit Successful!" ;

break;

}

} while (choice != 5);

return 0;

}

C14-flights\_path\_AB\_graph

#include<iostream>

#include<stdlib.h>

#include<string.h>

using namespace std;

struct node

{ string vertex;

int time;

node \*next;

};

class adjmatlist

{

int m[10][10],n,i,j; char ch; string v[20]; node \*head[20]; node \*temp=NULL;

public:

adjmatlist()

{

for(i=0;i<20;i++)

{

head[i]=NULL;

}

}

void getgraph();

void adjlist();

void displaym();

void displaya();

};

void adjmatlist::getgraph()

{

cout<<"\nEnter no. of cities(max. 20) : ";

cin>>n;

cout<<"\nEnter name of cities : ";

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

cin>>v[i];

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

{

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

{

cout<<"\nIf path is present between city "<<v[i]<<" and "<<v[j]<<" then press enter (Y) otherwise (N) : ";

cin>>ch;

if(ch=='Y')

{

cout<<"\nEnter time required to reach city "<<v[j]<<" from "<<v[i]<<" in minutes : ";

cin>>m[i][j];

}

else if(ch=='N')

{

m[i][j]=0;

}

else

{

cout<<"\nUnknown entry!";

}

}

}

adjlist();

}

void adjmatlist::adjlist()

{

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

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

{ node \*p=new(struct node);

p->next=NULL;

p->vertex=v[i];

head[i]=p; cout<<"\n"<<head[i]->vertex;

}

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

{

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

{

if(m[i][j]!=0)

{

node \*p=new(struct node);

p->vertex=v[j];

p->time=m[i][j];

p->next=NULL;

if(head[i]->next==NULL)

{ head[i]->next=p; }

else

{ temp=head[i];

while(temp->next!=NULL)

{

temp=temp->next;

}

temp->next=p;

}

}

}

}

}

void adjmatlist::displaym()

{

cout<<"\n";

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

{

cout<<"\t"<<v[j];

}

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

{

cout<<"\n "<<v[i];

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

{

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

}

cout<<"\n";

}

}

void adjmatlist::displaya()

{

cout<<"\nAdjacency list is : ";

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

{

if(head[i]==NULL)

{ cout<<"\nAdjacency list not present!"; break; }

else

{

cout<<"\n"<<head[i]->vertex;

temp=head[i]->next;

while(temp!=NULL)

{

cout<<"-> "<<temp->vertex;

temp=temp->next;

}

}

}

cout<<"\nPath and time required to reach cities is: ";

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

{

if(head[i]==NULL)

{

cout<<"\nAdjacency list not present!";

break;

}

else

{

temp=head[i]->next;

while(temp!=NULL)

{

cout<<"\n"<<head[i]->vertex;

cout<<"-> "<<temp->vertex<<"\n [time required: "<<temp->time<<" min ]";

temp=temp->next;

}

}

}

}

int main()

{ int m;

adjmatlist a;

while(1)

{

cout<<"\n 1.Enter graph : ";

cout<<"\n 2.Display adjacency matrix for cities : ";

cout<<"\n 3.Display adjacency list for cities : ";

cout<<"\n 4.Exit";

cout<<"\n\n Enter the choice: ";

cin>>m;

switch(m)

{

case 1:

a.getgraph();

break;

case 2:

a.displaym();

break;

case 3:

a.displaya();

break;

case 4:

cout << "Exit Successful!" ;

exit(0);

default:

cout<<"\n unknown choice";

}

}

return 0;

}

C11-dict\_key\_mean

#include <iostream>

#include<string>

using namespace std;

class Node {

public:

string key;

string meaning;

Node\* left;

Node\* right;

Node(string k, string m) {

key = k;

meaning = m;

left = NULL;

right = NULL;

}

};

class Dictionary {

private:

Node\* root;

public:

Dictionary() {

root = NULL;

}

void insert(string key, string meaning) {

root = insertNode(root, key, meaning);

cout << "Keyword inserted successfully!" << endl;

}

void remove(string key) {

root = removeNode(root, key);

cout << "Keyword removed successfully!" << endl;

}

void update(string key, string newMeaning) {

Node\* node = search(root, key);

if (node == NULL) {

cout << "Keyword not found in the dictionary!" << endl;

return;

}

node->meaning = newMeaning;

cout << "Meaning updated successfully!" << endl;

}

void displayAscending() {

cout << "Dictionary contents (ascending order):" << endl;

if (root == NULL) {

cout << "Dictionary is empty." << endl;

} else {

displayAscending(root);

}

}

void displayDescending() {

cout << "Dictionary contents (descending order):" << endl;

if (root == NULL) {

cout << "Dictionary is empty." << endl;

} else {

displayDescending(root);

}

}

int getMaxComparisons(string key) {

int comparisons = 0;

Node\* node = search(root, key);

if (node != NULL) {

comparisons = getMaxComparisons(root, key);

}

return comparisons;

}

private:

Node\* insertNode(Node\* root, string key, string meaning) {

if (root == NULL) {

return new Node(key, meaning);

}

if (key < root->key) {

root->left = insertNode(root->left, key, meaning);

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

root->right = insertNode(root->right, key, meaning);

}

return root;

}

Node\* removeNode(Node\* root, string key) {

if (root == NULL) {

return root;

}

if (key < root->key) {

root->left = removeNode(root->left, key);

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

root->right = removeNode(root->right, key);

} else {

if (root->left == NULL) {

Node\* temp = root->right;

delete root;

return temp;

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

Node\* temp = root->left;

delete root;

return temp;

}

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

root->key = temp->key;

root->meaning = temp->meaning;

root->right = removeNode(root->right, temp->key);

}

return root;

}

Node\* getMinNode(Node\* node) {

while (node->left != NULL) {

node = node->left;

}

return node;

}

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

if (root == NULL || root->key == key) {

return root;

}

if (key < root->key) {

return search(root->left, key);

}

return search(root->right, key);

}

void displayAscending(Node\* root) {

if (root != NULL) {

displayAscending(root->left);

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

displayAscending(root->right);

}

}

void displayDescending(Node\* root) {

if (root != NULL) {

displayDescending(root->right);

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

displayDescending(root->left);

}

}

int getMaxComparisons(Node\* root, string key) {

int comparisons = 1;

while (root->key != key) {

if (key < root->key) {

root = root->left;

} else {

root = root->right;

}

comparisons++;

}

return comparisons;

}

};

int main() {

Dictionary dictionary;

int choice;

string key, meaning, newMeaning;

while (true) {

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << " MENU" << endl;

cout << "1. Add a keyword" << endl;

cout << "2. Remove a keyword" << endl;

cout << "3. Update the meaning of a keyword" << endl;

cout << "4. Display the dictionary in ascending order" << endl;

cout << "5. Display the dictionary in descending order" << endl;

cout << "6. Get the maximum comparisons for a keyword" << endl;

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

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the keyword to add: ";

cin >> key;

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

cin.ignore();

getline(cin, meaning);

dictionary.insert(key, meaning);

break;

case 2:

cout << "Enter the keyword to remove: ";

cin >> key;

dictionary.remove(key);

break;

case 3:

cout << "Enter the keyword to update: ";

cin >> key;

cout << "Enter the new meaning of the keyword: ";

cin.ignore();

getline(cin, newMeaning);

dictionary.update(key, newMeaning);

break;

case 4:

dictionary.displayAscending();

break;

case 5:

dictionary.displayDescending();

break;

case 6:

cout << "Enter the keyword to get maximum comparisons: ";

cin >> key;

cout << "Maximum comparisons required: " << dictionary.getMaxComparisons(key) << endl;

break;

case 7:

exit(0);

default:

cout << "Invalid choice! Please try again." << endl;

}

cout << endl;

}

return 0;

}

B6-beg\_with\_empty\_BST

// B6

#include <iostream>

using namespace std;

class Node

{

public:

int data;

Node \*left;

Node \*right;

Node(int d)

{

data = d;

left = NULL;

right = NULL;

}

};

class BST

{

public:

Node \*insertToBst(Node \*root, int d)

{

if (root == NULL)

{

root = new Node(d);

return root;

}

if (d > root->data)

{

root->right = insertToBst(root->right, d);

return root;

}

else

{

root->left = insertToBst(root->left, d);

return root;

}

}

void create(Node \*&root)

{

int data;

cout << "Enter Element to insert in BST (if not enter -1) :" << endl;

cin >> data;

while (data != -1)

{

root = insertToBst(root, data);

cin >> data;

}

}

void inOrder(Node \*root)

{

if (root == NULL)

{

return;

}

inOrder(root->left);

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

inOrder(root->right);

}

Node \*search(Node \*root, int key)

{

if (root == NULL)

return NULL;

if (root->data == key)

return root;

if (root->data < key)

return search(root->right, key);

else

{

return search(root->left, key);

}

}

void minValue(Node \*root)

{

Node \*temp = root;

while (temp->left != NULL)

{

temp = temp->left;

}

cout << "min :" << temp->data;

}

void maxValue(Node \*root)

{

Node \*temp = root;

while (temp->right != NULL)

{

temp = temp->right;

}

cout << "max :" << temp->data;

}

Node \*mirror(Node \*root)

{

if (root == NULL)

return root;

else

{

Node \*temp;

/\* do the subtrees \*/

mirror(root->left);

mirror(root->right);

/\* swap the pointers in this node \*/

temp = root->left;

root->left = root->right;

root->right = temp;

}

}

int maxDepth(Node \*root)

{

if (root == NULL)

return 0;

else

{

int lDepth = 1 + maxDepth(root->left);

int rDepth = 1 + maxDepth(root->right);

if (lDepth > rDepth)

return (lDepth);

else

return (rDepth);

}

}

};

int main()

{

BST b;

int key;

Node \*root = NULL;

Node \*p;

int ch;

cout << "\*\*\*\*\*\*\*\*\*\*MENU\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\n1.Create\n2.Inorder\n3.Search\n4.Min\n5.Max\n6.Mirror image\n7.Height of BST\n8.Exit\n";

while (1)

{

cout << "\nEnter a choice:";

cin >> ch;

switch (ch)

{

case 1:

b.create(root);

break;

case 2:

b.inOrder(root);

break;

case 3:

cout << "\nEnter key to be searched : ";

cin >> key;

p = b.search(root, key);

if (p == NULL)

cout << "\nElement not found";

else

cout << "\nElement found :" << p->data;

break;

case 4:

b.minValue(root);

break;

case 5:

b.maxValue(root);

break;

case 6:

b.mirror(root);

break;

case 7:

cout << "Height of tree :" << b.maxDepth(root);

break;

case 8:

exit(1);

}

}

return 0;

}

B5-book\_chapters

//B5

#include <iostream>

#include <vector>

using namespace std;

class Tree

{

public:

string value;

vector<Tree \*> child;

Tree(string val)

{

value = val;

}

};

void insertchap(Tree \*head)

{

string chapname;

cout << "Enter the chapter name - " << endl;

cin >> chapname;

cout << endl;

Tree \*newchap = new Tree(chapname);

int subsect;

cout << "Enter no of subsection - " << endl;

cin >> subsect;

cout << endl;

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

{

string subname;

cout << "Enter the subsection name - ";

cin >> subname;

cout << endl;

Tree \*subsection = new Tree(subname);

newchap->child.push\_back(subsection);

}

head->child.push\_back(newchap);

}

void display(Tree \*head, string temp)

{

if (head)

{

cout << temp + head->value << endl;

temp += " ";

for (int i = 0; i < head->child.size(); i++)

{

display(head->child[i], temp);

}

}

}

int main()

{

Tree \*head = new Tree("--INDEX--");

int chapters;

cout << "Enter no of chapters:" << endl;

cin >> chapters;

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

{

insertchap(head);

}

string temp = "";

display(head, temp);

return 0;

}