**Bharati Vidyapeeth’s College of Engineering for Women, Pune**

**Department of Computer Engineering**

**Practical Assignment Codes**

**Sub – Data Structure & Algorithm Laboratory**

**Academic Year – 2020-21, Sem – II**

**Name:- Ziya Fatima Sayed**

**Class:- SE Computer Science**

**Roll number:- 2242**

**ExamSeat Number:- S190344267**

**Practical 1:** Consider telephone book database of N clients. Make use of a hash table

implementation to quickly look up the client‘s telephone number. Use two collision handling

techniques and compare them using the number of comparisons required to find a set of

telephone numbers.

# Program to implement Hashing with Linear Probing from Record import

Record.

class hashTable:

# initialize hash Table

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

self.size = int(input("Enter the Size of the hash table : "))

# initialize table with all elements 0

self.table = list(None for i in range(self.size))

self.elementCount = 0

self.comparisons = 0

# method that checks if the hash table is full or not

def isFull(self):

if self.elementCount == self.size:

return True

else:

return False

# method that returns position for a given element

def hashFunction(self, element):

return element % self.size

# method that inserts element into the hash table

def insert(self, record):

# checking if the table is full

if self.isFull():

print("Hash Table Full")

return False

isStored = False

position = self.hashFunction(record.get\_number())

# checking if the position is empty

if self.table[position] == None:

self.table[position] = record

print("Phone number of " + record.get\_name() + " is at

position " + str(position))

isStored = True

self.elementCount += 1

# collision occured hence we do linear probing

else:

print("Collision has occured for " + record.get\_name() + "'s

phone number at position " + str(position) + " finding new Position.")

while self.table[position] != None:

position += 1

if position >= self.size:

position = 0

self.table[position] = record

print("Phone number of " + record.get\_name() + " is at

position " + str(position))

isStored = True

self.elementCount += 1

return isStored

# method that searches for an element in the table

# returns position of element if found

# else returns False

def search(self, record):

found = False

position = self.hashFunction(record.get\_number())

self.comparisons += 1

if(self.table[position] != None):

if(self.table[position].get\_name() == record.get\_name() and

self.table[position].get\_number() == record.get\_number()):

isFound = True

print("Phone number found at position {}

".format(position) + " and total comparisons are " + str(1))

return position

# if the element is not found at the position returned hash

function

else:

position += 1

if position >= self.size-1:

position = 0

while self.table[position] != None or self.comparisons <=

self.size:

if(self.table[position].get\_name() ==

record.get\_name() and self.table[position].get\_number() ==

record.get\_number()):

isFound = True

#i=0

i = self.comparisons + 1

print("Phone number found at position {}

".format(position) + " and total comparisons are " + str(i) )

return position

position += 1

#print(position)

if position >= self.size-1:

position = 0

#print(position)

self.comparisons += 1

#print(self.comparisons)

if isFound == False:

print("Record not found")

return false

# method to display the hash table

def display(self):

print("\n")

for i in range(self.size):

print("Hash Value: "+str(i) + "\t\t" + str(self.table[i]))

print("The number of phonebook records in the Table are : " +

str(self.elementCount))

# Program to implement Hashing with double hashing table from Record

import Record.

class doubleHashTable:

# initialize hash Table

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

self.size = int(input(&quot;Enter the Size of the hash table : &quot;))

# initialize table with all elements 0

self.table = list(None for i in range(self.size))

self.elementCount = 0

self.comparisons = 0

# method that checks if the hash table is full or not

def isFull(self):

if self.elementCount == self.size:

return True

else:

return False

# First hash function

def h1(self, element):

return element % self.size

# Second hash function

def h2(self, element):

return 5-(element % 5)

# method to resolve collision by double hashing method

def doubleHashing(self, record):

posFound = False

# limit variable is used to restrict the function from going into

infinite loop

# limit is useful when the table is 80% full

limit = self.size

i = 1

# start a loop to find the position

while i &lt;= limit:

# calculate new position by quadratic probing

newPosition = (self.h1(record.get\_number()) +

i\*self.h2(record.get\_number())) % self.size

# if newPosition is empty then break out of loop and return

new Position

if self.table[newPosition] == None:

posFound = True

break

else:

# as the position is not empty increase i

i += 1

return posFound, newPosition

# method that inserts element inside the hash table

def insert(self, record):

# checking if the table is full

if self.isFull():

print(&quot;Hash Table Full&quot;)

return False

posFound = False

position = self.h1(record.get\_number())

# checking if the position is empty

if self.table[position] == None:

# empty position found , store the element and print the

message

self.table[position] = record

print(&quot;Phone number of &quot; + record.get\_name() + &quot; is at

position &quot; + str(position))

isStored = True

self.elementCount += 1

# If collision occured

else:

print(&quot;Collision has occured for &quot; + record.get\_name() +

&quot;&#39;s

phone number at position &quot; + str(position) + &quot; finding new

Position.&quot;)

while not posFound:

posFound, position = self.doubleHashing(record)

if posFound:

self.table[position] = record

#print(self.table[position])

self.elementCount += 1

#print(position)

#print(posFound)

print(&quot;Phone number of &quot; + record.get\_name() + &quot; is at

position &quot; + str(position))

return posFound

# searches for an element in the table and returns position of

element if found else returns False

def search(self, record):

found = False

position = self.h1(record.get\_number())

self.comparisons += 1

if(self.table[position] != None):

if(self.table[position].get\_name() == record.get\_name()):

print(&quot;Phone number found at position {}&quot;.format(position)

+ &quot; and total comparisons are &quot; + str(1))

return position

# if the element is not found at the position returned hash function

# then we search an element using double hashing

else:

limit = self.size

i = 1

newPosition = position

# start a loop to find the position

while i &lt;= limit:

# calculate new position by double Hashing

position = (self.h1(record.get\_number()) +

i\*self.h2(record.get\_number())) % self.size

self.comparisons += 1

# if element at newPosition is equal to the required

element

if(self.table[position] != None):

if self.table[position].get\_name() == record.get\_name():

found = True

break

elif self.table[position].get\_name() == None:

found = False

break

else:

# as the position is not empty increase i

i += 1

if found:

print(&quot;Phone number found at position {}&quot;.format(position)

+ &quot; and total comparisons are &quot; + str(i+1))

#return position

else:

print(&quot;Record not Found&quot;)

return found

# method to display the hash table

def display(self):

print(&quot;\n&quot;)

for i in range(self.size):

print(&quot;Hash Value: &quot;+str(i) + &quot;\t\t&quot; +

str(self.table[i]))

print(&quot;The number of phonebook records in the Table are : &quot; +

str(self.elementCount))

**Practical 2:** To create ADT that implement the "set" concept.

a. Add (new Element) -Place a value into the set

b. Remove (element) Remove the value

c. Contains (element) Return true if the element is in the collection

d. Size () Return number of values in collection Iterator () Return an iterator used to loop over

the collection

e. The intersection of two sets

f. Union of two sets

g. Difference between two sets

h. Subset

class Set :

# Creates an empty set instance.

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

self.\_s = []

for i in range(initElementsCount) :

e = int(input(&quot;Enter Element {}: &quot;.format(i+1)))

self.add(e)

def get\_set(self):

return self.\_s

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

string = &quot;\n{ &quot;

for i in range(len(self.get\_set())):

string = string + str(self.get\_set()[i])

if i != len(self.get\_set())-1:

string = string + &quot; , &quot;

string = string + &quot; }\n&quot;

return string

# Returns the number of items in the set.

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

return len( self.\_s )

# Determines if an element is in the set.

def \_\_contains\_\_( self, e ):

return e in self.\_s

# Determines if the set is empty.

def isEmpty( self ):

return len(self.\_s) == 0

# Adds a new unique element to the set.

def add( self, e ):

if e not in self :

self.\_s.append( e )

# Removes an e from the set.

def remove( self, e ):

if e in self.get\_set():

self.get\_set().remove(e)

# Determines if this set is equal to setB.

def \_\_eq\_\_( self, setB ):

if len( self ) != len( setB ) :

return False

else :

return self.isSubsetOf( setB )

# Determines if this set is a subset of setB.

def isSubsetOf( self, setB ):

for e in setB.get\_set() :

if e not in self.get\_set() :

return False

return True

# Determines if this set is a proper subset of setB.

def isProperSubset( self, setB ):

if self.isSubsetOf(setB) and not setB.isSubsetOf(self):

return True

return False

# Creates a new set from the union of this set and setB.

def union( self, setB ):

newSet = self

for e in setB :

if e not in self.get\_set() :

newSet.add(e)

return newSet

# Creates a new set from the intersection: self set and setB.

def intersect( self, setB ):

newSet = Set(0)

for i in range(len(self.get\_set())) :

for j in range(len(setB.get\_set())) :

if self.get\_set()[i] == setB.get\_set()[j] :

newSet.add(self.get\_set()[i])

return newSet

# Creates a new set from the difference: self set and setB.

def difference( self, setB ):

newSet = Set(0)

for e in self.get\_set() :

if e not in setB.get\_set():

newSet.add(e)

return newSet

# Creates the iterator for traversing the list of items

def \_\_iter\_\_( self ):

return iter(self.\_s)

**Practical 3:** Beginning with an empty binary search tree, Construct a binary search tree by

inserting the values in the order given. After constructing a binary tree -

i. Insert new node, ii. Find the number of nodes in the 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;

struct bstnode

{

int data;

bstnode \*left,\*right;

};

bstnode\*insert(bstnode\*t,int x)

{

bstnode\*p,\*q,\*r;

r=new bstnode;

r->data=x;

r->left=NULL;

r->right=NULL;

if(t==NULL)

return(r);

p=t;

while(p!=NULL)

{

q=p;

if(x>p->data)

p=p->right;

else

p=p->left;

}

if(x>q->data)

q->right=r;

else

q->left=r;

return(t);

}

bstnode\*create()

{

int n,x,i;

bstnode\*root;

root=NULL;

cout<<"enter the no of nodes:";

cin>>n;

cout<<"enter the values:";

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

{

cin>>x;

root=insert(root,x);

}

return (root);

}

bstnode\*find\_min(bstnode\*t)

{

while(t->left!=NULL)

{

t=t->left;

}

return (t);

}

void search(bstnode\*root,int x)

{

int f=0;

while(root!=NULL)

{

if(root->data==x)

{

cout<<"element is found";

f=1;

}

if(x>root->data)

root=root->right;

else

root=root->left;

}

if(f==0)

cout<<"element is not present.";

}

void postorder(bstnode\*t)

{

if(t!=NULL)

{

postorder(t->left);

postorder(t->right);

cout<<t->data<<" ";

}

}

void inorder(bstnode\*t)

{

if(t!=NULL)

{

inorder(t->left);

cout<<t->data<<" ";

inorder(t->right);

}

}

void preorder(bstnode \*t)

{

if(t!=NULL)

{

cout<<t->data<<" ";

preorder(t->left);

preorder(t->right);

}

}

int depth(bstnode \*t)

{

int lh,rh;

if(t==NULL)

return 0;

else

{

lh=depth(t->left);

rh=depth(t->right);

if(lh<rh)

return(rh+1);

else

return(lh+1);

}

}

bstnode \*swap(bstnode \*t)

{

bstnode \*p;

if(t!=NULL)

{

p=t->left;

t->left=swap(t->right);

t->right=swap(p);

}

return t;

}

int main()

{

int ch,a,g,q,height;

bstnode \*b,\*c;

do

{

cout<<"\n1.create\n2.insert any node\n3.find minimum node from

tree\n4.search any element from tree\n5.display by inorder\n6.display by

preorder\n7.display by postorder\n8.depth\n9.no of nodes in longest

path\n10.swapping\n11.exit\nenter your choice:";

cin>>ch;

switch(ch)

{

case 1:

b=create();

break;

case 2:

cout<<"enter the value you want to insert:";

cin>>a;

b=insert(b,a);

break;

case 3:

c=find\_min(b);

cout<<"minimum value is:"<<c->data;

break;

case 4:

cout<<"enter the data which you want to search:";

cin>>g;

search(b,g);

break;

case 5:

cout<<"\nDisplay by inorder is: ";

inorder(b);

break;

case 6:

cout<<"\nDisplay by preorder is: ";

preorder(b);

break;

case 7:

cout<<"\nDisplay by postorder is: ";

postorder(b);

break;

case 8:

height=depth(b);

cout<<"\ndepth is: "<<height;

break;

case 9:

height=depth(b);

cout<<"\nno of nodes in longest path are:

"<<height;

break;

case 10:

b=swap(b);

inorder(b);

break;

case 11:

cout<<"Exit";

break;

}

}while(ch!=11);

return 0;

}

**Practical 4:** Convert the given binary tree into a threaded binary tree. Analyze the time and

space complexity of the algorithm.

#include<iostream>

using namespace std;

struct TBTnode

{

int data;

int lbit;

int rbit;

TBTnode \*left,\*right;

};

class Tree

{

public:

int n=0;

TBTnode\* in[20];

TBTnode \*create();

TBTnode \*insert(TBTnode \*T,int x);

void Inorder(TBTnode \*T);

void inorder(TBTnode \*T);

void Threding(TBTnode \*T);

};

TBTnode\* Tree::insert(TBTnode \*T,int x)

{

TBTnode \*p,\*q,\*r;

r=new TBTnode;

r->data=x;

r->left=NULL;

r->right=NULL;

if(T==NULL)

return r;

p=T;

while(p!=NULL)

{

q=p;

if(x>p->data)

p=p->right;

else

p=p->left;

}

if(x>q->data)

q->right=r;

else

q->left=r;

return T;

}

TBTnode\* Tree::create()

{

int n,x,i;

TBTnode \*root;

root=NULL;

cout<<"Enter number of node= ";

cin>>n;

cout<<"Enter tree value="<<endl;

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

{

cin>>x;

root=insert(root,x);

}

return (root);

}

void Tree::inorder(TBTnode \*T)

{

if(T!=NULL)

{

inorder(T->left);

in[n++]=T;

if(T->left==NULL)

T->lbit=0;

else

T->lbit=1;

if(T->right==NULL)

T->rbit=0;

else

T->rbit=1;

inorder(T->right);

}

}

void Tree::Inorder(TBTnode \*T)

{

if(T!=NULL)

{

Inorder(T->left);

cout<<T->data<<" ";

Inorder(T->right);

}

}

void Tree::Threding(TBTnode \*T)

{

int i;

TBTnode \*root,\*q;

root=new TBTnode;

root->data=00;

root->lbit=root->rbit=1;

root->left=T;

root->right=root;

q=in[0];

q->left=root;

cout<<"Thread to left of "<<q->data<<" is "<<q->left->data<<endl;

if(q->rbit==0)

{

q->right=in[1];

cout<<"Thread to right of "<<q->data<<" is "<<q->right->data<<endl;

}

q=in[n-1];

if(q->lbit==0)

{

q->left=in[n-2];

cout<<"Thread to left of "<<q->data<<" is "<<q->left->data<<endl;

}

q->right=root;

cout<<"Thread to right of "<<q->data<<" is "<<q->right->data<<endl;

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

{

q=in[i];

if(q->lbit==0)

{

q->left=in[i-1];

cout<<"Thread to left of "<<q->data<<" is "<<q->left->data<<endl;

}

if(q->rbit==0)

{

q->right=in[i+1];

cout<<"Thread to right of "<<q->data<<" is "<<q->right->data<<endl;

}

}

}

int main()

{

Tree s;

TBTnode \*h;

h=s.create();

cout<<"INORDER IS:"<<endl;

s.Inorder(h);

s.inorder(h);

cout<<"CONVERSION OF BINARY SEARCH TREE INTO THREADING TREE "<<endl;

s.Threding(h);

return 0;

}

**Practical 5:** Construct an expression tree from the given prefix expression eg. +--a\*bc/def and

traverse it using post-order traversal (non-recursive) and then delete the entire tree.

﻿#include<iostream>

#include<ctype.h>

#include<cstring>

using namespace std;

struct node

{

char data;

node \*link;

};

struct dnode

{

char data;

dnode \*left;

dnode \*right;

};

struct dsnode

{

dnode \*data;

dsnode \*link;

};

class Stack

{

node \*top;

public:

Stack()

{

top=new node;

top=NULL;

}

void push(char x)

{

node \*temp;

temp=new node;

temp->data=x;

if(top==NULL)

{

top=temp;

top->link=NULL;

}

else

{

temp->link=top;

top=temp;

}

}

char pop()

{

char x;

x=top->data;

top=top->link;

return x;

}

char tos()

{

return top->data;

}

int empty()

{

if(top==NULL)

return 1;

return 0;

}

int precedence(char x)

{

if(x=='(')

return 0;

if(x=='+'||x=='-')

return 1;

if(x=='\*'||x=='/'||x=='%')

return 2;

return 3;

}

void infixtopostfix(char infix[50], char postfix[50])

{

char x;

int i, j;

char token;

j=0;

for(i=0; infix[i]!='\0'; i++)

{

token=infix[i];

if(isalnum(token))

postfix[j++]=token;

else

if(token=='(')

push('(');

else

if(token==')')

while((x=pop())!='(')

postfix[j++]=x;

else

{

while(precedence(token)<=precedence(tos())&&!empty())

{

x=pop();

postfix[j++]=x;

}

push(token);

}

}

while(!empty())

{

x=pop();

postfix[j++]=x;

}

postfix[j]='\0';

}

};

class DStack

{

dsnode \*top;

public:

DStack()

{

top=new dsnode;

top=NULL;

}

void push(dnode \*x)

{

dsnode \*temp;

temp=new dsnode;

temp->data=x;

if(top==NULL)

{

top=temp;

top->link=NULL;

}

else

{

temp->link=top;

top=temp;

}

}

dnode\* pop()

{

dnode \*x;

x=top->data;

top=top->link;

return x;

}

};

class TreeMaker

{

DStack \*s;

dnode \*root;

char ex[50];

public:

TreeMaker(char a[50])

{

s=new DStack();

strcpy(ex, a);

root=NULL;

}

void creator()

{

for(int i=0; ex[i]!='\0'; i++)

{

dnode \*temp=new dnode;

temp->data=ex[i];

temp->right=NULL;

temp->left=NULL;

if(isalnum(ex[i]))

{

s->push(temp);

}

else

{

dnode \*opl, \*opr;

opr=s->pop();

opl=s->pop();

temp->left=opl;

temp->right=opr;

s->push(temp);

}

}

root=s->pop();

cout<<"\nINORDER : ";

inorder(root);

cout<<"\nPOSTORDER :";

postorder(root);

}

void inorder(dnode \*temp)

{

if(temp->left != NULL)

inorder(temp->left);

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

if(temp->right != NULL)

inorder(temp->right);

}

void postorder(dnode \*temp)

{

if(temp->left != NULL)

postorder(temp->left);

if(temp->right != NULL)

postorder(temp->right);

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

}

};

int main()

{

Stack \*s=new Stack();

char a[50], b[50], sb[52]="(", lb[2]=")";

cout<<"\nEnter the infix expression : ";

cin>>a;

strcat(sb, a);

strcat(sb, lb);

strcpy(a, sb);

s->infixtopostfix(a, b);

cout<<"POSTFIX : "<<b;

TreeMaker \*h=new TreeMaker(b);

h->creator();

return 0;

}

**Practical 6:** 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 the airport name or the 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.

#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<<"\n enter no. of cities(max.20)";

cin>>n;

cout<<"\n enter name of cities:\n";

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

cin>>v[i];

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

{

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

{ cout<<"\n if path is present between city "<<v[i]<<" and

"<<v[j]<<" then press y otherwise n\n";

cin>>ch;

if(ch=='y')

{

cout<<"\n enter time required to reach city "<<v[j]<<" from

"<<v[i]<<" in minutes\n";

cin>>m[i][j];

}

else if(ch=='n')

{ m[i][j]=0; }

else

{ cout<<"\n unknown entry\n"; }

}

}

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<<"\n adjacency list is";

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

{

if(head[i]==NULL)

{ cout<<"\n adjacency list not present";

break; }

else

{

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

temp=head[i]->next;

while(temp!=NULL)

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

temp=temp->next; }

}

}

cout<<"\n path and time required to reach cities is";

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

{

if(head[i]==NULL)

{ cout<<"\n adjacency 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\n enter the choice:\n";

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";

cin>>m;

switch(m)

{ case 1: a.getgraph();

break;

case 2: a.displaym();

break;

case 3: a.displaya();

break;

case 4: exit(0);

default: cout<<"\n unknown choice";

}

}

return 0;

}

**Practical 7:** 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.

#define infinity 9999

#define MAX 20

#include<iostream>

#include<stdlib.h>

using namespace std;

class graph

{

int G[MAX][MAX];

int n;

public:

graph()

{

n=0;

}

void readgraph();

void printgraph();

int prims(graph&);

};

void graph::readgraph()

{

int i,j;

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

cin>>n;

cout<<"\n Enter the adjacency martrix";

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

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

cin>>G[i][j];

}

void graph::printgraph()

{

int i,j;

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

{

cout<<"\n";

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

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

}

}

int graph::prims(graph &spanning)

{

int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,j;

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

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

{

if(G[i][j]==0)

cost[i][j]=infinity;

else

cost[i][j]=G[i][j];

spanning.G[i][j]=0;

}

distance[0]=0;visited[0]=1;

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

{

distance[i]=cost[0][i];

from[i]=0;

visited[i]=0;

}

min\_cost=0;

no\_of\_edges=n-1;

while(no\_of\_edges>0)

{

min\_distance=infinity;

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

if(visited[i]==0 && distance[i]<min\_distance)

{

v=i;

min\_distance=distance[i];

}

u=from[v];

spanning.G[u][v]=distance[v];

spanning.G[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

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

if(visited[i]==0 && cost[i][v]<distance[i])

{

distance[i]=cost[i][v];

from[i]=v;

}

min\_cost=min\_cost+cost[u][v];

}

spanning.n=n;

return(min\_cost);

}

int main()

{

graph g1,spanning;

g1.readgraph();

int total\_cost=g1.prims(spanning);

cout<<"\nspanning tree matrix:\n";

spanning.printgraph();

cout<<"\n total cost of spanning tree="<<total\_cost;

}

**Practical 8:** Given sequence k = k1 <k2 < … <kn of n sorted keys, with a search probability pi

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

probability for each key?

/\* This program is to implement optimal binary search tree\*/

#include<iostream>

using namespace std;

#define SIZE 10

class OBST

{

int p[SIZE]; // Probabilities with which we search for an element

int q[SIZE];//Probabilities that an element is not found

int a[SIZE];//Elements from which OBST is to be built

int w[SIZE][SIZE];//Weight ‘w[i][j]’ of a tree having root

//’r[i][j]’

int c[SIZE][SIZE];//Cost ‘c[i][j] of a tree having root ‘r[i][j]

int r[SIZE][SIZE];//represents root

int n; // number of nodes

public:

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

void get\_data()

{

int i;

cout<<"\n Optimal Binary Search Tree \n";

cout<<"\n Enter the number of nodes";

cin>>n;

cout<<"\n Enter the data as…\n";

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

{

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

cin>>a[i];

}

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

{

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

cin>>p[i];

}

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

{

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

cin>>q[i];

}

}

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

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

int Min\_Value(int i,int j)

{

int m,k;

int minimum=32000;

for(m=r[i][j-1];m<=r[i+1][j];m++)

{

if((c[i][m-1]+c[m][j])<minimum)

{

minimum=c[i][m-1]+c[m][j];

k=m;

}

}

return k;

}

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

basically computes C,r,W values \*/

void build\_OBST()

{

int i,j,k,l,m;

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

{

//initialize

w[i][i]=q[i];

r[i][i]=c[i][i]=0;

//Optimal trees with one node

w[i][i+1]=q[i]+q[i+1]+p[i+1];

r[i][i+1]=i+1;

c[i][i+1]=q[i]+q[i+1]+p[i+1];

}

w[n][n]=q[n];

r[n][n]=c[n][n]=0;

//Find optimal trees with ‘m’ nodes

for(m=2;m<=n;m++)

{

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

{

j=i+m;

w[i][j]=w[i][j-1]+p[j]+q[j];

k=Min\_Value(i,j);

c[i][j]=w[i][j]+c[i][k-1]+c[k][j];

r[i][j]=k;

}

}

}

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

\*/

void build\_tree()

{

int i,j,k;

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

cout<<"The Optimal Binary Search Tree For the Given Node Is…\n";

cout<<"\n The Root of this OBST is ::"<<r[0][n];

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

cout<<"\n\n\t NODE \t LEFT CHILD \t RIGHT CHILD ";

cout<<"\n";

queue[++rear]=0;

queue[++rear]=n;

while(front!=rear)

{

i=queue[++front];

j=queue[++front];

k=r[i][j];

cout<<"\n\t"<<k;

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

{

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

queue[++rear]=i;

queue[++rear]=k-1;

}

else

cout<<"\t\t";

if(r[k][j]!=0)

{

cout<<"\t"<<r[k][j];

queue[++rear]=k;

queue[++rear]=j;

}

else

cout<"\t";

}//end of while

cout<<"\n";

}

};//end of the class

/\*This is the main function \*/

int main()

{

OBST obj;

obj.get\_data();

obj.build\_OBST();

obj.build\_tree();

return 0;

}

**Practical 9:** A Dictionary stores keywords and their 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 the Height balance tree and find the complexity for finding a

keyword

#include <iostream>

#include <string.h>

#include <queue>

using namespace std;

class node

{

char key[10];

char mean[30];

int BF;

node \*left,\*right;

public:

node()

{

strcpy(key," ");

strcpy(mean," ");

BF=0;

left=NULL;

right=NULL;

}

friend class avl;

};

class avl

{

node\* root;

public:

avl()

{

root=NULL;

}

node\* insert(node\*,char[],char[]);

void display();

void insert();

node\* LL(node\*);

node\* LR(node\*);

node\* RR(node\*);

node\* RL(node\*);

int bal\_fact(node\*);

int get\_height(node\*);

void levelwisedisplay();

node\* delete\_(node\*,char[]);

void delete\_();

node\* FindMin(node\*);

void update(node\*);

void update();

void display\_ascending(node\*);

void display\_descending(node\*);

void search();

node\*find(node\*,char[]);

};

void avl :: insert()

{

char data[10],meaning[30];

cout<<"\n Enter the keyword to be added in the dictionary ";

cin>>data;

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

cin>>meaning;

root= insert(root,data,meaning);

}

node\* avl :: insert(node\*root,char data[10],char meaning[30])

{

int bal;

if(root==NULL)

{

root=new node;

strcpy(root->key,data);

strcpy(root->mean,meaning);

root->left=NULL;

root->right=NULL;

root->BF=0;

}

else

{

if(strcmp(data,root->key)<0)

{

cout<<root->key;

root->left=insert(root->left,data,meaning);

bal=bal\_fact(root);

cout<<"\n node "<<root->key<<" BF "<<bal;

if(bal==2)

{

if(strcmp(data,root->left->key)<0)

root=LL(root);

else

root=LR(root);

}

}

else

{

root->right=insert(root->right,data,meaning);

bal=bal\_fact(root);

cout<<"\n node "<<root->key<<" BF "<<bal;

if(bal==-2)

{

if(strcmp(data,root->right->key)>0)

root=RR(root);

else

root=RL(root);

}

}

root->BF=get\_height(root);

}

return (root);

}

int avl :: get\_height(node\*temp)

{

int hl,hr;

if(temp==NULL) //to check if node is null

return 0;

else{

hl=get\_height(temp->left); //for finding height of left

subtree

hr=get\_height(temp->right);//for finding height of right

subtree

if(hl>hr)

return hl+1; //returing height of left subtree with

adding the height by 1 for root

else

return hr+1;//returing height of right subtree with

adding the height by 1 for root

}

}

node \* avl::LL(node\* root)

{

node \*temp;

temp=root->left;

root->left=temp->right;

temp->right=root;

temp->BF=get\_height(temp);

root->BF=get\_height(root);

cout<<"\n LL rotation done ";

return (temp);

}

node \* avl::RR(node\* root)

{

node \*temp;

temp=root->right;

root->right=temp->left;

temp->left=root;

temp->BF=get\_height(temp);

root->BF=get\_height(root);

cout<<"\n RR rotation done ";

return (temp);

}

node \*avl :: RL(node \*root)

{

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

root=RR(root);

return root;

}

node\* avl :: LR(node\* root)

{

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

root=LL(root);

return root;

}

int avl :: bal\_fact(node\* temp)

{

int hr,hl;

hl=get\_height(temp->left);

hr=get\_height(temp->right);

return hl-hr;

}

void avl :: display()

{

int ch=0;

cout<<root->key;

cout<<"\nEnter your choice \n1.For Ascending display \n2.For Descending

display ";

cin>>ch;

if(ch==1)

display\_ascending(root);

else

display\_descending(root);

}

void avl :: display\_ascending(node\* root)

{

if(root!=NULL) //to check if node is null

{

display\_ascending(root->left);

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

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

display\_ascending(root->right);

}

}

void avl :: display\_descending(node\* root)

{

if(root!=NULL) //to check if node is null

{

display\_descending(root->right);

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

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

display\_descending(root->left);

}

}

node\* avl :: FindMin(node\* root){

while(root->left != NULL) root = root->left;

return root;

}

node\* avl ::delete\_(node\*root,char data[10])

{

node \*temp;

if(root == NULL) return root;

else if(strcmp(data , root->key)<0)

root->left = delete\_(root->left,data);

else if(strcmp(data , root->key)>0)

root->right = delete\_(root->right, data);

else {

// Case 1: No Child

if(root->left == NULL && root->right == NULL){

delete root;

root = NULL;

// Case 2: one child

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

temp = root;

root = root->right;

delete temp;

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

temp = root;

root = root->left;

delete temp;

} else{

temp = FindMin(root->right);

strcpy(root->key , temp->key);

root->right = delete\_(root->right, temp->key);

}

}

return root;

}

void avl :: delete\_()

{

char data[10];

cout<<"\n Enter the keyword to be deleted from the dictionary

";

cin>>data;

delete\_(root,data);

}

void avl :: update()

{

update(root);

}

void avl :: update(node\* root)

{

char key[10],meaning[30];

cout<<"\n Enter the keyword to be updated ";

cin>>key;

find(root,key);

cout<<"\n Enter the new meaning of the word ";

cin>>meaning;

strcpy(root->mean,meaning);

}

node\* avl::find(node\*root,char key[10]){

int flag=0;

if(root!=NULL&&flag==0){

if(strcmp(key , root->key)<0){

return find(root->left,key);

}if(strcmp(key , root->key)>0){

return find(root->right,key);

}else if(strcmp(key , root->key)==0){

flag=1;

}

}if(flag==1){

cout<<"DATA FOUND"<<endl;

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

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

}if(flag==0){

cout<<"DATA NOT FOUND";

}

return root;

}

void avl::search(){

avl t;

char d[10];

cout<<"ENTER DATA TO FIND";

cin>>d;

t.find(root,d);

}

int main() {

avl \*a;

a =new avl();

char ans;

int ch;

do{

cout<<"\n======MENU======\nEnter your choice ";

cout<<"\n1.Insert \n2.Display \n3.Delete \n4.Update\n5.search:";

cin>>ch;

switch(ch)

{

case 1: a->insert();

break;

case 2: a->display();

break;

case 3: a->delete\_();

break;

case 4: a->update();

break;

case 5:a->search();

break;

}

cout<<"\nDo you want to continue ?? ";

cin>>ans;

}while(ans=='y');

return 0;

}

**Practical 10:** Read the marks obtained by students of the second year in an online examination

of a particular subject =. Find out maximum and minimum marks obtained in that subject. Use

heap data structure. Analyze the algorithm

/\* Program for Heap Data Structure in C++ \*/

#include<iostream>

using namespace std;

void createmax(int[]);

void display(int[]);

void createmin(int[]);

void downadj1(int[],int);

void downadj2(int[],int);

int main()

{

int heap[30],n,i,l,temp;

cout<<"\n Enter the total no. of students ";

cin>>n;

heap[0]=n;

cout<<"\n Enter the online exanimation marks for students";

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

cin>>heap[i];

display(heap);

createmax(heap);

cout<<"\n Maximum marks = "<<heap[1];

createmin(heap);

cout<<"\n MInimum marks = "<<heap[1]<<endl;

return 0;

}

void createmax(int heap[])

{

int i,n;

n=heap[0];

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

downadj1(heap,i);

}

void display(int heap[])

{

int i, n;

n=heap[0];

cout<<"\n List of the marks = \t ";

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

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

}

void downadj1(int heap[],int i)

{

int j,temp,n,flag=1;

n=heap[0];

while(2\*i<=n&&flag==1)

{

j=2\*i;

if(j+1<=n&&heap[j+1]>heap[j])

j=j+1;

if(heap[i]>heap[j])

flag=0;

else

{

temp=heap[i];

heap[i]=heap[j];

heap[j]=temp;

i=j;

}

}

}

void createmin(int heap[])

{

int i,n;

n=heap[0];

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

downadj2(heap,i);

}

void downadj2(int heap[],int i)

{

int j,temp,n,flag=1;

n=heap[0];

while(2\*i<=n&&flag==1)

{

j=2\*i;

if(j+1<=n&&heap[j+1]<heap[j])

j=j+1;

if(heap[i]<heap[j])

flag=0;

else

{

temp=heap[i];

heap[i]=heap[j];

heap[j]=temp;

i=j;

}

}

}

**Practical 11:** Department maintains student information. The file contains roll number, name,

division and address. Allow user to add, delete information of the student. Display

information of a particular employee. If the record of the student does not exist an appropriate

message is displayed. If it is, then the system displays the student details. Use the sequential

file to main the data.

#include<iostream>

#include<fstream>

#include<string.h>

using namespace std;

class student

{

typedef struct stud

{

int roll;

char name[10];

char div;

char add[10];

}stud;

stud rec;

public:

void create();

void display();

int search();

void Delete();

};

void student::create()

{

char ans;

ofstream fout;

fout.open("stud.txt",ios::out|ios::binary);

do

{

cout<<"\n\tEnter Roll No of Student : ";

cin>>rec.roll;

cout<<"\n\tEnter a Name of Student : ";

cin>>rec.name;

cout<<"\n\tEnter a Division of Student : ";

cin>>rec.div;

cout<<"\n\tEnter a Address of Student : ";

cin>>rec.add;

fout.write((char \*)&rec,sizeof(stud))<<flush;

cout<<"\n\tDo You Want to Add More Records: ";

cin>>ans;

}while(ans=='y'||ans=='Y');

fout.close();

}

void student::display()

{

ifstream fin;

fin.open("stud.txt",ios::in|ios::binary);

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

cout<<"\n\tThe Content of File are:\n";

cout<<"\n\tRoll\tName\tDiv\tAddress";

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

{

if(rec.roll!=-1)

cout<<"\n\t"<<rec.roll<<"\t"<<rec.name<<"\t"<<rec.div<<"\t"<<rec.add;

}

fin.close();

}

int student::search()

{

int r,i=0;

ifstream fin;

fin.open("stud.txt",ios::in|ios::binary);

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

cout<<"\n\tEnter a Roll No: ";

cin>>r;

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

{

if(rec.roll==r)

{

cout<<"\n\tRecord Found...\n";

cout<<"\n\tRoll\tName\tDiv\tAddress";

cout<<"\n\t"<<rec.roll<<"\t"<<rec.name<<"\t"<<rec.div<<"\t"<<rec.add;

return i;

}

i++;

}

return -1;

fin.close();

}

void student::Delete()

{

int pos;

pos=search();

fstream f;

f.open("stud.txt",ios::in|ios::out|ios::binary);

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

if(pos==-1)

{

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

return;

}

int offset=pos\*sizeof(stud);

f.seekp(offset);

rec.roll=-1;

strcpy(rec.name,"NULL");

rec.div='N';

strcpy(rec.add,"NULL");

f.write((char \*)&rec,sizeof(stud));

f.seekg(0);

f.close();

cout<<"\n\tRecord Deleted";

}

int main()

{

student obj;

int ch,key;

char ans;

do

{

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

cout<<"\n\t1. Create\n\t2. Display\n\t3. Delete\n\t4.

Search\n\t5. Exit";

cout<<"\n\t..... Enter Your Choice: ";

cin>>ch;

switch(ch)

{

case 1: obj.create();

break;

case 2: obj.display();

break;

case 3: obj.Delete();

break;

case 4: key=obj.search();

if(key==-1)

cout<<"\n\tRecord Not Found...\n";

break;

case 5:

break;

}

cout<<"\n\t..... Do You Want to Continue in Main Menu: ";

cin>>ans;

}while(ans=='y'||ans=='Y');

return 1;

}