DATA STRUCTURE AND ALGORITHM (ITCO6)

PRACTICAL FILE



**NAME:** JAHANAVI MISHRA

**ROLL NO:** 2017UIT2622

**BRANCH:** INFORMATION TECHNOLOGY

**SECTION:** 2

# SUBMITTED TO: PROF. Amarjit Malhotra

Q1 : Implement two stacks in an array.

#include<iostream>

using namespace std;

class stack

{

int \*arr;

int size;

int top1, top2;

public:

stack(int n)

{

size=n;

arr=new int[n];

top1=-1;

top2=size;

}

void push1(int x)

{

if(top1<top2-1)

{

top1++;

arr[top1]=x;

}

else

cout<<"Stack Overflow";

}

void push2(int x)

{

if(top1<top2-1)

{

top2--;

arr[top2]=x;

}

else

cout<<"stack overflow";

}

int pop1()

{

if(top1>=0)

{

int x=arr[top1];

top1--;

return x;

}

else

cout<<"Stack Underflow";

}

int pop2()

{

if(top2<size)

{

int x=arr[top2];

top2++;

return x;

}

else

cout<<"stack underflow";

}

};

int main()

{

stack s(5);

s.push1(2);

s.push1(4);

s.push1(6);

s.push2(8);

s.push2(10);

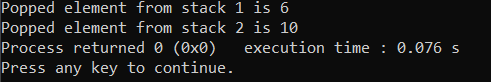
cout<<"Popped element from stack 1 is "<<s.pop1();

cout<<endl;

cout<<"Popped element from stack 2 is "<<s.pop2();

return 0;

}



Q2) Implement Queue using two stacks.

#include<iostream>

#include<stack>

using namespace std;

class queue

{

stack <int> s1,s2;

public:

void enqueue(int x)

{

while(!s1.empty())

{

s2.push(s1.top());

s1.pop();

}

s1.push(x);

while(!s2.empty())

{

s1.push(s2.top());

s2.pop();

}

}

int dequeue()

{

if(s1.empty())

{

cout<<"the queue is empty";

return -1;

}

int x=s1.top();

s1.pop();

return x;

}

};

int main()

{

queue q;

q.enqueue(1);

q.enqueue(2);

int k=q.dequeue();

if(k!=-1)

cout<<"Popped element is "<<k<<endl;

k=q.dequeue();

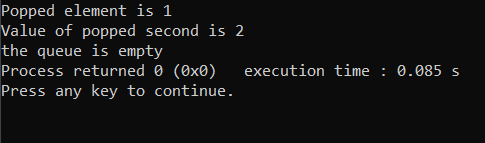
if(k!=-1)

cout<<"Value of popped second is "<<k<<endl;

k=q.dequeue();

return 0;

}



Q3) Implement sorted linked list.

#include<iostream>

using namespace std;

class node

{

public:

int data;

node\*next;

node(int d)

{

data=d;

next=NULL;

}

};

class lnklst

{

public:

node\*head;

lnklst()

{

head=NULL;

}

node\*search(int d)

{

node\*ptr=head;

while(ptr!=NULL)

{

if(ptr->data==d)

return ptr;

else if(ptr->data>d)

break;

else ptr=ptr->next;

}

cout<<"data not available"<<endl;

return NULL;

}

void insert(int d)

{

node\*n=new node(d);

if(head==NULL)

{

head=n;

head->next=NULL;

return;

}

node\*ptr=head;

node\*prev=NULL;

while(ptr!=NULL&&ptr->data<d)

{

prev=ptr;

ptr=ptr->next;

}

if(prev==NULL)

{

n->next=head;

head=n;

}

else

{

n->next=prev->next;

prev->next=n;

}

}

int del(int d)

{

int item;

if(head==NULL)

{

cout<<"empty"<<endl;

return -1;

}

node\*prev=NULL;

node\*ptr=head;

while(ptr!=NULL)

{

if(ptr->data==d)

{

if(prev==NULL)

{

item=head->data;

head=head->next;

delete ptr; //CHECK

return item;

}

else

{

item=ptr->data;

prev->next=ptr->next;

delete ptr;

return item;

}

}

if(ptr->data>d)

{

cout<<"item unavailable"<<endl;

return -1;

}

prev=ptr;

ptr=ptr->next;

}

}

void printll()

{

if(head==NULL)

return;

node\*ptr=head;

while(ptr!=NULL)

{

cout<<ptr->data<<" - >";

ptr=ptr->next;

}

}

};

int main()

{

lnklst l;

l.insert(1);

l.insert(3);

l.insert(2);

l.insert(0);

l.insert(5);

l.printll();

cout<<endl;

l.del(1);

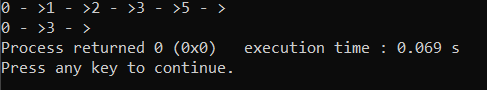
l.del(5);

l.del(2);

l.printll();

return 0;

}



Q4. Program to implement priority queue.

#include <iostream>

#include <stdio.h>

using namespace std;

class node

{

public:

    int data; //implementing priority queue as linked list

    int prior;

    node \*next;

    node(int info, int priority)

    {

        data=info;

        prior=priority;

        next=NULL;

    }

};

class pri\_queue

{

    public:

    node \* start;

    node \*\* rear;//array of rear index

    int count;//No. of elements in priority queue

    pri\_queue(int data)

    {

        start= NULL;

        rear= new node\*[data];

        for(int i=0;i<data;i++)

        {

            rear[i]=NULL;

        }

        count=0;

    }

    void insert(int data, int priority);

    int del(int prority);

    void display();

};

void pri\_queue :: insert(int data, int priority)

{

    node\*temp= new node(data,priority);//for first insertion

    count=count+1;

    if(start==NULL )

    {

        rear[priority-1]=temp;

        start=temp;

    }

    else if(rear[priority-1]==NULL)//all successive insertions

    {

        node \*tmp= NULL;

        for(int i=0; i<priority;i++)//for storing address of rear

        {

            if (rear[i]!= NULL)

            {

                tmp=rear[i];

            }

        }

        if( tmp!=NULL)

        {

            temp->next= tmp->next;

            tmp->next= temp;

            rear[priority-1]=temp;

        }

        else

        {

            temp->next= start;

            start=temp;

            rear[priority-1]= temp;

            start= temp;

        }

    }

    else

    {

        temp->next= rear[priority-1]->next;

        rear[priority-1]->next= temp;

        rear[priority-1]= temp;

    }

return;

}

int pri\_queue :: del (int priority)

{

    node \* temp2 =NULL;//deletion of an element

    temp2= start;

    start=start->next;

    temp2->next=NULL;

    int x = temp2->data;

    delete temp2;

    if(rear[priority-1]==start)//changing rear address

    {

        rear[priority-1]=NULL;

    }

    count--;

    return x;

}

void pri\_queue::display()//display the priority queue

{

    node \*ptr;

    ptr = start;

    if (start == NULL)

        cout<<"Queue is empty\n";

    else

    {          cout<<"Queue is :\n";

        cout<<"Priority       Item\n";

        while(ptr != NULL)

        {

            cout<<ptr->prior<<"                 "<<ptr->data<<endl;

            ptr = ptr->next;

                }

            }

        }

int main()

{

    pri\_queue q(5);

    q.insert(1,2);

    q.insert(5,3);

    q.display();

    q.insert(3,1);

    q.insert(4,2);

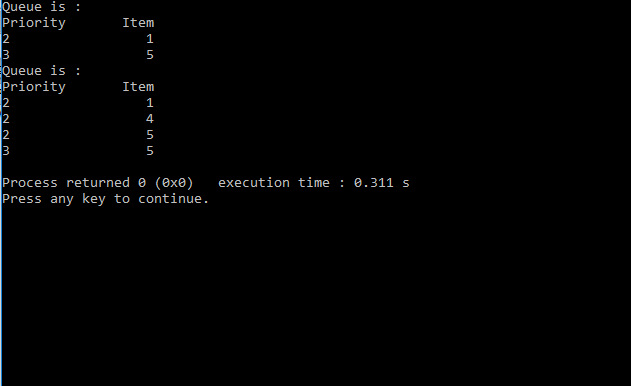
    q.insert(5,2);

    q.del(1);

    q.display();

    return 0;

}



Q5) Implement quicksort using Doubly Linked List.

#include<iostream>

#include<stack>

using namespace std;

class Node

{

public:

int data;

Node \* next;

Node \* prev;

Node(int data)

{

this -> data = data;

this -> next = this -> prev = this;

}

};

Node\* head = NULL;

Node\* takeinput()

{

cout<<"enter the elements to be sorted"<<endl;

int data;

cin >> data;

while(data != -1)

{

Node \*newNode = new Node(data);

if(head == NULL)

{

Node \* newNode = new Node(data);

head = newNode;

}

else

{

Node \* newNode = new Node(data);

Node \* tail = head -> prev;

tail -> next = newNode;

newNode -> next = head;

newNode -> prev = tail;

head -> prev = newNode;

}

cin >> data;

}

return head;

}

Node \* swapTwoNodes(Node \* start, Node\* temp1, Node\* temp2)

{

if(temp1 == temp2)

return head;

if(temp1 -> next == temp2)

{

Node \* prev = temp1 -> prev;

Node \* next = temp2 -> next;

prev -> next = temp2;

temp2 -> prev = prev;

temp2 -> next = temp1;

temp1 -> prev = temp2;

temp1 -> next = next;

next -> prev = temp1;

if (start == temp1)

return temp2;

}

if(temp2 -> next == temp1)

{

Node \* prev = temp2 -> prev;

Node \* next = temp1 -> next;

prev -> next = temp1;

temp1 -> prev = prev;

temp1 -> next = temp2;

temp2 -> prev = temp1;

temp1 -> next = next;

next -> prev = temp2;

if (start == temp2)

return temp1;

}

Node \* prev1 = temp1 -> prev;

Node \* prev2 = temp2 -> prev;

Node \* next1 = temp1 -> next;

Node \* next2 = temp2 -> next;

prev1 -> next = temp2;

temp2 -> prev = prev1;

temp2 -> next = next1;

next1 -> prev = temp2;

temp1 -> next = next2;

prev2 -> next = temp1;

temp1 -> prev = prev2;

next2 -> prev = temp1;

if (start == temp2)

{

return temp1;

}

else if(start == temp1)

{

return temp2;

}

return start;

}

void swap\_data(Node\* temp1, Node\* temp2)

{

int temp = temp1 -> data;

temp1 -> data = temp2 -> data;

temp2 -> data = temp;

}

Node\* quick(Node\* head, Node\* beg, Node\* end)

{

Node\* left = beg;

Node\* right = end;

Node\* loc = left;

while(1)

{

while(loc -> data <= right -> data && loc != right)

right = right -> prev;

if(loc == right)

break;

if(loc -> data > right -> data)

{

swap\_data(loc, right);

loc = right;

}

while(loc -> data >= left -> data && loc != left)

left = left -> next;

if(loc == left)

break;

if(loc -> data < left -> data)

{

swap\_data(loc, left);

loc = left;

}

}

return loc;

}

void quickSort(Node\* head)

{

if(head == NULL || head -> next == head)

return;

stack<Node\*> upper;

stack<Node\*> lower;

lower.push(head);

upper.push(head -> prev);

while(!upper.empty() && !lower.empty())

{

Node\* beg = lower.top();

Node\* end = upper.top();

lower.pop();

upper.pop();

Node\* loc = quick(head, beg, end);

if(beg != loc)

{

lower.push(beg);

upper.push(loc -> prev);

}

if(end != loc && loc -> next != end)

{

lower.push(loc -> next);

upper.push(end);

}

}

}

void printL(Node\* head)

{

if(head == NULL)

return;

Node\* ptr = head;

cout<< ptr -> data << " ";

ptr = ptr -> next;

while(ptr != head)

{

cout << ptr -> data << " ";

ptr = ptr -> next;

}

}

int main()

{

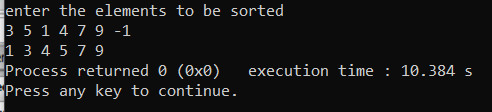
Node \* head = takeinput();

quickSort(head);

printL(head);

return 0;

}



Q6) Evaluate infix arithmetic expression.

#include<iostream>

#include<stack>

using namespace std;

int exp\_eval(string);

int operation(int,int,char);

bool isoperator(char);

bool predence(char,char);

int main()

{

string exp;

int ans;

cout<<"Enter Infix Arithmetic Expression: ";

cin>>exp;

ans= exp\_eval(exp);

cout<<"The value of the Infix Arithmetic Expression is: "<<ans<<endl;

return 0;

}

int exp\_eval(string exp)

{

stack<int> opr; //stack1

stack<char> op; //stack2

for(int i=0; i<exp.length(); i++)

{

if(isdigit(exp[i]))

{

int num=0;

while(isdigit(exp[i]))

{

num\*=10;

num+=exp[i]-'0';

i++;

}

opr.push(num);

i--;

}

else if(exp[i]=='(')

op.push(exp[i]);

else if(exp[i]==')')

{

while(op.top()!='(')

{

char c=op.top();

op.pop();

int a=opr.top();

opr.pop();

int b= opr.top();

opr.pop();

int result= operation(a,b,c);

opr.push(result);

}

op.pop();

}

else if(isoperator(exp[i]))

{

while(!op.empty() && predence(op.top(),exp[i]))

{

char c=op.top();

op.pop();

int a=opr.top();

opr.pop();

int b= opr.top();

opr.pop();

int result= operation(a,b,c);

opr.push(result);

}

op.push(exp[i]);

}

}

while(!op.empty())

{

char c=op.top();

op.pop();

int a=opr.top();

opr.pop();

int b= opr.top();

opr.pop();

int result= operation(a,b,c);

opr.push(result);

}

return opr.top();

}

int operation(int a, int b, char c)

{

if(c=='+')

return b+a;

else if(c=='-')

return b-a;

else if(c=='\*')

return b\*a;

else if(c=='/')

return b/a;

else if(c=='^')

{

int value=1;

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

value\*=b;

return value;

}

return 0;

}

bool isoperator(char c)

{

if(c=='+'||c=='-'||c=='\*'||c=='/'||c=='^')

return 1;

else return 0;

}

bool predence(char a, char b)

{

if( a=='^')

return 1;

else if(a=='/'||a=='\*'&& b!='^')

return 1;

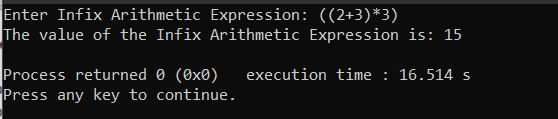
else if(a=='+'||a=='-'&& b=='+'||b=='-')

return 1;

else

return 0;

}



Q7. Implement BST and display in tree format.

#include<iostream>

#include<math.h>

#include<queue>

#include<iomanip>

using namespace std;

struct Node

{

int data;

Node\* left, \*right;

};

Node \*root=new Node;

Node\* newNode(int data) //Helper Function to create new node

{

Node\* node = new Node;

node->data = data;

node->left =NULL; node->right = NULL;

return node;

}

void print\_tree(Node \*root,int h)

{

queue<int>q;

queue<Node\*>n1;

int count=0;

Node\* temp=root;

Node\* dummy= newNode(-1);

n1.push(temp);

//Insert all data in q and put -1 when node does not exist

while(count!= (pow(2, h)-1)){

Node\* temp1= n1.front();

n1.pop();

q.push(temp1->data);

if(temp1->left== NULL)

temp1->left= dummy;

if(temp1->right== NULL)

temp1->right=dummy;

n1.push(temp1->left);

n1.push(temp1->right);

count++;

}

queue<int>q1;

q1=q;

//Actual Print Function

int n= pow(2,h-1);

int space= (2\*n-1);

int in\_sp= ((n-1)\*3) +1;

int level=0, levelcount=0;

count=0;

int prev=0,curr=0;

bool toggle=true;

// flag=0 print '\'

//and flag=1 '/' print

while(level<h)

{

int lc=levelcount;

if(lc==0){

int s= space/(pow(2,level+1));

if(q1.front()!= -1)

if(level!=0)

cout<<setw(s+1)<<"/";

else

cout<<setw(s+1)<<" ";

q1.pop();

curr=s;

toggle=true;

lc++;

}

//Printing the subsequent nodes

while(lc<pow(2,level)){

int s= space/(pow(2,level));

int k= prev-curr;

if(q1.front()== -1){

cout<<setw(2\*k)<<" ";

q1.pop();

}

else{

if(toggle){

cout<<setw(2\*k)<<"\\";

}

else

cout<<setw(2\*k)<<"/";

q1.pop();

}

lc++;

toggle = !toggle;

}

cout<<endl;

//Printing the first node of a level

if(levelcount==0){

int s= space/(pow(2,level+1));

if(q.front()!= -1)

cout<<setw(s+1)<<q.front();

else

cout<<setw(s+1)<<" ";

q.pop();

curr=s;

levelcount++;

}

//Printing the subsequent nodes

while(levelcount<pow(2,level)){

int s= space/(pow(2,level));

int k= prev-curr;

if(q.front()== -1){

cout<<setw(2\*k)<<" ";

q.pop();

}

else{

cout<<setw(2\*k)<<q.front();

q.pop();

}

levelcount++;

}

//Go to next level and line

cout<<endl;

level++;

levelcount=0;

prev=curr;

}

}

int main()

{

int height=5;

Node \*n2=new Node;

Node \*n3=new Node;

Node \*n4=new Node;

Node \*n5=new Node;

Node \*n6=new Node;

Node \*n7=new Node;

Node \*n8=new Node;

Node \*n9=new Node;

Node \*n10=new Node;

Node \*n11=new Node;

Node \*n12=new Node;

Node \*n13=new Node;

Node \*n14=new Node;

root->data=10;

root->left=n2;

root->right=n3;

n2->data=20;

n2->left=n4;

n2->right=n5;

n3->data=100;

n3->left=n6;

n3->right=n7;

n4->data=2;

n4->left=n8;

n4->right=NULL;

n5->data=70;

n5->left=NULL;

n5->right=NULL;

n6->data=68;

n6->left=NULL;

n6->right=n9;

n7->data=88;

n7->left=n10;

n7->right=n11;

n8->data=93;

n8->left=n12;

n8->right=NULL;

n9->data=7;

n9->left=NULL;

n9->right=NULL;

n10->data=110;

n10->left=NULL;

n10->right=NULL;

n11->data=5;

n11->left=NULL;

n11->right=NULL;

n12->data=46;

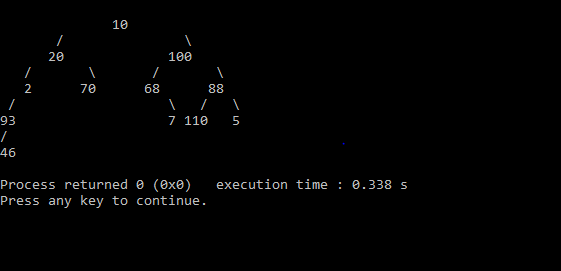
n12->left=NULL;

n12->right=NULL;

print\_tree(root, height);

return 0;

}



Q8. Implement AVL tress.

#include<stdio.h>

#include<stdlib.h>

#include<iostream>

using namespace std;

struct Node

{

int key;

struct Node \*left;

struct Node \*right;

int height;

};

int height(struct Node \*N)

{

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b)

{

return (a > b)? a : b;

}

struct Node\* newNode(int key)

{

struct Node\* node = (struct Node\*)

malloc(sizeof(struct Node));

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return(node);

}

struct Node \*rightRotate(struct Node \*y)

{

struct Node \*x = y->left;

struct Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right))+1;

x->height = max(height(x->left), height(x->right))+1;

return x;

}

struct Node \*leftRotate(struct Node \*x)

{

struct Node \*y = x->right;

struct Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right))+1;

y->height = max(height(y->left), height(y->right))+1;

return y;

}

int getBalance(struct Node \*N)

{

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

struct Node\* insert(struct Node\* node, int key)

{

if (node == NULL)

return(newNode(key));

if (key < node->key)

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

else if (key > node->key)

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

else

return node;

node->height = 1 + max(height(node->left),

height(node->right));

int balance = getBalance(node);

if (balance > 1 && key < node->left->key)

return rightRotate(node);

if (balance < -1 && key > node->right->key)

return leftRotate(node);

if (balance > 1 && key > node->left->key)

{

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

return rightRotate(node);

}

if (balance < -1 && key < node->right->key)

{

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

return leftRotate(node);

}

return node;

}

void inOrder(struct Node \*root)

{

if(root != NULL)

{

inOrder(root->left);

printf("%d ", root->key);

inOrder(root->right);

}

}

void preOrder(struct Node \*root)

{

if(root != NULL)

{

printf("%d ", root->key);

preOrder(root->left);

preOrder(root->right);

}

}

int main()

{

struct Node \*root = NULL;

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 100);

root = insert(root, 10);

root = insert(root, 60);

root = insert(root, 70);

cout<<"The preorder traversal"<<endl;

preOrder(root);

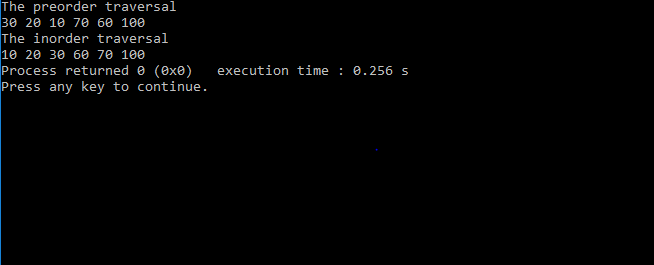
cout<<endl;

cout<<"The inorder traversal"<<endl;

inOrder(root);

return 0;

}



Q9. Implement Huffman code.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_TREE\_HT 100

struct MinHeapNode

{

char data;

unsigned freq;

struct MinHeapNode \*left, \*right;

};

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHeapNode\*\* array;

};

struct MinHeapNode\* newNode(char data, unsigned freq)

{

struct MinHeapNode\* temp

= (struct MinHeapNode\*)malloc

(sizeof(struct MinHeapNode));

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

temp->data = data;

temp->freq = freq;

return temp;

}

struct MinHeap\* createMinHeap(unsigned capacity)

{

struct MinHeap\* minHeap

= (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array

= (struct MinHeapNode\*\*)malloc(minHeap->

capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b)

{

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

void minHeapify(struct MinHeap\* minHeap, int idx)

{

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->

freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->

freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest],

&minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

int isSizeOne(struct MinHeap\* minHeap)

{

return (minHeap->size == 1);

}

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)

{

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode)

{

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

void buildMinHeap(struct MinHeap\* minHeap)

{

int n = minHeap->size - 1;

int i;

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

minHeapify(minHeap, i);

}

void printArr(int arr[], int n)

{

int i;

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

printf("%d", arr[i]);

printf("\n");

}

int isLeaf(struct MinHeapNode\* root)

{

return !(root->left) && !(root->right);

}

struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size)

{

struct MinHeap\* minHeap = createMinHeap(size);

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

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size)

{

struct MinHeapNode \*left, \*right, \*top;

struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

void printCodes(struct MinHeapNode\* root, int arr[], int top)

{

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

if (isLeaf(root)) {

printf("%c: ", root->data);

printArr(arr, top);

}

}

void HuffmanCodes(char data[], int freq[], int size)

{

struct MinHeapNode\* root

= buildHuffmanTree(data, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

int main()

{

char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

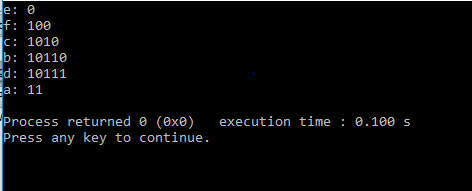
int freq[] = { 45, 3, 11, 10, 67, 15 };

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

HuffmanCodes(arr, freq, size);

return 0;

}



Q10. Implement Heapsort.

#include<iostream>

using namespace std;

void inplaceHeapSort(int input[], int n)

{

//build heap within the array

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

{

int ci = i;

while(ci > 0)

{

int pi = (ci - 1)/2;

if(input[ci] < input[pi])

{

int temp = input[ci];

input[ci] = input[pi];

input[pi] = temp;

}

else break;

ci = pi;

}

}

//down-heapifying elements

int size = n;

while(size > 1)

{

int temp = input[0];

input[0] = input[size - 1];

input[size - 1] = temp;

size--;

int pi = 0;

int li = 2\*pi + 1;

int ri = 2\*pi + 2;

while(li < size)

{

int mi = pi;

if(input[mi] > input[li])

mi = li;

if(input[mi] > input[ri] && ri < size)

mi = ri;

if(mi == pi)

break;

int t = input[mi];

input[mi] = input[pi];

input[pi] = t;

pi = mi;

li = 2\*pi + 1;

ri = 2\*pi + 2;

}

}

}

int main() {

int size;

cout<<"Enter the number of elements"<<endl;

cin >> size;

int \*input = new int[1 + size];

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

cin >> input[i];

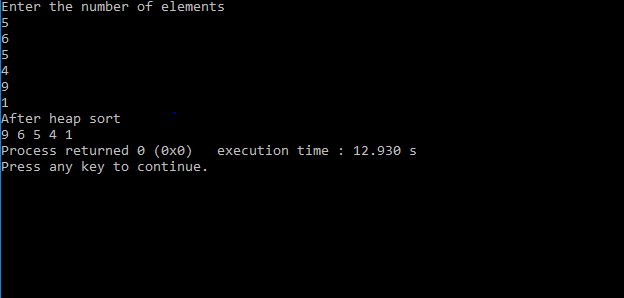
inplaceHeapSort(input, size);

cout<<"After heap sort"<<endl;

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

cout << input[i] << " ";

return 0;

}

Q11. Implement Dijkstra’s Algorithm.

#include<iostream>

#include<climits>

using namespace std;

int findMinVertex(int\* distance, bool\* visited, int n){

int minVertex = -1;

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

if(!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex])){

minVertex = i;

}

}

return minVertex;

}

void dijkstra(int\*\* edges, int n){

int\* distance = new int[n];

bool\* visited = new bool[n];

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

distance[i] = INT\_MAX;

visited[i] = false;

}

distance[0] = 0;

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

int minVertex = findMinVertex(distance, visited, n);

visited[minVertex] = true;

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

if(edges[minVertex][j] != 0 && !visited[j]){

int dist = distance[minVertex] + edges[minVertex][j];

if(dist < distance[j]){

distance[j] = dist;

}

}

}

}

cout<<"Vertex"<<" "<<"Distance"<<endl;

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

cout << i << " " << distance[i] << endl;

}

delete [] visited;

delete [] distance;

}

int main() {

int n;

int e;

cin >> n >> e;

int\*\* edges = new int\*[n];

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

edges[i] = new int[n];

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

edges[i][j] = 0;

}

}

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

int f, s, weight;

cin >> f >> s >> weight;

edges[f][s] = weight;

edges[s][f] = weight;

}

cout << endl;

dijkstra(edges, n);

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

delete [] edges[i];

}

delete [] edges;

}

