DHA Suffa University

Department of Computer Science

CS 2001L – Data Structures and Algorithms Lab

Fall 2019

Lab 13-Graphs

Objective:

Learn about concepts and implementation of:

- Graphs
- Adjacency matrix and adjacency list
- Breadth First and Depth First Search Algorithm

Graph:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph −

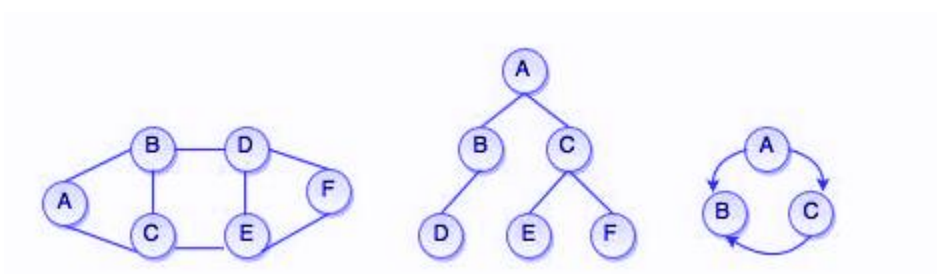


Figure 13.1 – Graphs

Directed Graph

- If a graph contains ordered pair of vertices, is said to be a Directed Graph.
- If an edge is represented using a pair of vertices (V_1, V_2), the edge is said to be directed from V_1 to V_2.
- The first element of the pair V_1 is called the start vertex and the second element of the pair V_2 is called the end vertex.

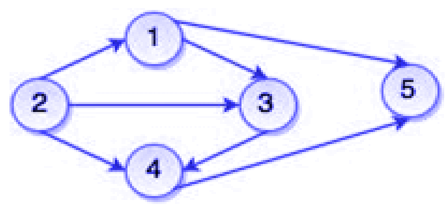


Figure 13.2 – Directed Graph

Set of Vertices V = {1, 2, 3, 4, 5}

Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

Undirected Graph

- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.

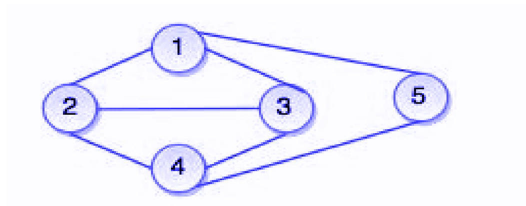


Figure 13.3 –Undirected Graph

Set of Vertices V = {1, 2, 3, 4, 5}

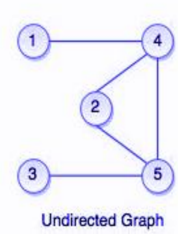
Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3,1),(3,2),(3, 4), (4,2),(4,3),(4,5), (5,1),(5,4)}In an undirected graph, the nodes are connected by undirected arcs.

It is an edge that has no arrow. Both the ends of an undirected arc are equivalent, there is no head or tail.

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex**: Each node of the graph is represented as a vertex. In figure 13.6 the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge**: Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency**: Two node or vertices are adjacent if they are connected to each other through an edge. In figure 13.6, B is adjacent to A, C is adjacent to B, and so on.



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Figure 13.4 – Adjacency Matrix of undirected Graph

- The above graph represents undirected graph with the adjacency matrix representation. It shows adjacency matrix of undirected graph is symmetric. If there is an edge (2, 4), there is also an edge (4, 2).

- Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicated a directed edge from vertex i to vertex j

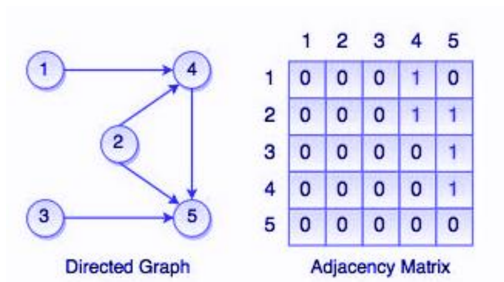


Figure 13.5 – Adjacency Matrix of Directed Graph

- **Path:** Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

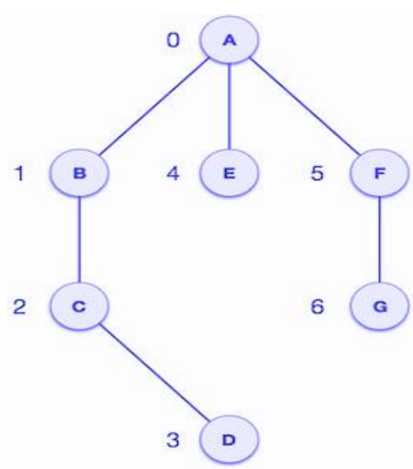


Figure 13.6 – Representation of path of a Graph

Implementation of Adjacency Matrix:

Graph.h

```
class Graph
{
    private:
        int** adjMat;

    public:
        Graph();
        ~Graph();
        void add_vertices(int v);
        void add_edge(int u, int v);
        void displayAdjMatrix(int v);
};
```

Graph.cpp

```
#include "Graph.h"
#include <iostream>
Graph::Graph(){}
Graph::~Graph(){}
void Graph :: add_vertices(int v){
    adjMat = new int*[v];
    int i,j;
    for(i = 0;i<v;++i)
    {
        adjMat[i] = new int[v];
    }

    for(i = 0 ; i < v ; ++i)
    {
        for(j = 0; j < v ; ++j)
            adjMat[i][j] = 0;
    }

}
void Graph::add_edge(int u, int v){
    adjMat[u][v] = 1;

}
```

```
void Graph::displayAdjMatrix(int v){

    int i, j;
    for(i = 0; i < v; i++) {
        for(j = 0; j < v; j++) {
            std::cout << adjMat[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Driver.cpp

```
#include <iostream>
#include "Graph.h"

int main(){

    Graph g;
    g.add_vertices(5);
    g.add_edge(0, 1);
    g.add_edge(0, 2);
    g.add_edge(1, 3);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(3, 4);
    g.add_edge(4, 4);

    g.displayAdjMatrix(5);

}
```

LAB TASK: Add function countIndegree () and countOutdegree () in class Graph to count the indegree and outdegree of a vertex .

Implementation of Adjacency List:

Adj.h

```
struct Adj
{
    int data;
    Adj* next;

    Adj(int data);
    ~Adj();
};
```

Adj.cpp

```
#include "Adj.h"
#include <iostream>
Adj::Adj(int data):data(data),next(NULL){}
Adj::~Adj(){}
```

Node.h

```
#include "Adj.h"
struct Node{

        int data;
        Node *next;
        Adj* n;
        bool is_visited;

        Node(int data);
        ~Node();
};
```

Node.cpp

```
#include "Node.h"
#include <iostream>

Node::Node(int data):data(data),next(NULL),n(NULL),is_visited(false){}
Node::~Node(){};
```

Graph.h

```
#include "Node.h"
#include <iostream>
#include <queue>
class Graph{
    private:
        Node *head;

    public:
        Graph();
        void insertNodes(int vertices);
        void addEdge(int u,int v);
        void printGraph();
        void BFS(std::queue<Node*> q);
        Node* getHeadVal();
};
```

Graph.cpp

```
#include "Graph.h"
#include <iostream>
Graph::Graph():head(NULL){}
void Graph::insertNodes(int vertices)
{
    Node *current  = NULL;
    for(int i = 0;i<vertices; ++i)
    {
        Node *temp = new Node(i);

        if(head == NULL)
        {
            head = temp;
            current = head;

        }
        else
        {
            current->next = temp;
            current = temp;
        }
    }
}
void Graph::addEdge(int u,int v){
    Node *current = head;
    Adj *c = NULL;

    while(current!=NULL)
    {
        if(current->data==u)break;
        current = current->next;
    }

    if(current->n == NULL)
        current->n = new Adj(v);
    else
    {
        c = current->n;
        while(c->next!=NULL)
            c = c->next;

        c->next = new Adj(v);
    }
}
```

```cpp
void Graph::printGraph(){


    Node *current = head;
    Adj *c = NULL;

    while(current!=NULL)
    {
        c = current->n;
        std::cout<<current->data<<" -> ";
        while(c!=NULL)
        {
            std::cout<<c->data;
            c = c->next;
            if(c!=NULL) std::cout<<" - ";
        }

        current = current->next;
        std::cout<<std::endl;
    }
}
```

Main.cpp

```cpp
#include "Graph.h"
#include <queue>
#include <iostream>

int main()
{
    Graph graph ;

    graph.insertNodes(4);
    graph.addEdge(0,1);
    graph.addEdge(0,2);
    graph.addEdge(0,3);
    graph.addEdge(1,3);
    graph.addEdge(2,3);
    graph.addEdge(3,3);

    graph.printGraph();


    return 0;
}
```

BREADTH FIRST SEARCH:

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search.

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

Implementation:

Include these functions in file Graph.cpp

```
Node* Graph::getHeadVal(){
     head->is_visited = true;
     return head;
}

void Graph::BFS(std::queue<Node*> q){


    if(q.empty()) return;

    Node *current = q.front();

    Adj *c = current->n;
    q.pop();

    std::cout<<current->data<<" - ";
    while(c!=NULL)
    {
        Node *temp = head;
        while(temp !=NULL)
        {
            if(temp->data == c->data)break;
            temp = temp->next;
        }
        if(temp->is_visited !=true)
        {
            q.push(temp);
            temp ->is_visited = true;
        }

        c = c->next;
    }

    BFS(q);
}
```

Driver.cpp

```
#include "Graph.h"
#include <queue>
#include <iostream>

int main()
{
    Graph graph ;

    graph.insertNodes(4);
    graph.addEdge(0,1);
    graph.addEdge(0,2);
    graph.addEdge(0,3);
    graph.addEdge(1,3);
    graph.addEdge(2,3);
    graph.addEdge(3,3);

    graph.printGraph();

    std::queue<Node*> myqueue;

    myqueue.push(graph.getHeadVal());

    std::cout<<"Traversal : BFS "<<std::endl;

    graph.BFS(myqueue);

    return 0;
}
```

Assignment:

Q1) In class graph of adjacency list add function countIndegree () and countOutdegree () to count the indegree and outdegree of a vertex.

Q2) Traverse graph using Depth First Search Algorithm.

Submission Guidelines

- **Write C++ code , separate function for each operation.**
- **Place your file in a folder named with your rollNo (cs172xxx) where xxx is your 3 digit rollno.**
- **Upload it on LMS.**

DHA SUFFA UNIVERSITY

Final Date Sheet

End Semester Examinations - Fall 2019

Department of Computer Science

Timing	0900 to 1200	0900 to 1200	0900 to 1200	0900 to 1200	0900 to 1200	0900 to 1200	0900 to 1200	0900 to 1200	0900 to 1200
Class/Day/Date	Monday 13-01-2020	Tuesday 14-01-2020	Wednesday 15-01-2020	Thursday 16-01-2020	Friday 17-01-2020	Monday 20-01-2020	Tuesday 21-01-2020	Wednesday 22-01-2020	Thursday 23-01-2020
BSCS -1	Calculus & Analytical Geometry BS-1301 Iftikhar Ahmed		Islamic Studies HU-2201 Dr. Waqas Ethics HU-1603 M. Mustafa Raza			Programming Fundamentals CS-1001 Twaha Ahmed Minai [Online]		Introduction to Information & Communication Technology CS-1201 Dr. Shama Siddiqui	English Composition & Comprehension HU-1002 Bushra Zaidi
BSCS -2	Multivariate Calculus BS-2301 Maraj Ahmed			Basic Electronics BS-1102 Engr. Junaid Ahmed / Muhammad Owais		Object Oriented Programming CS-1002 Syed Muhammad Farooq Shibli		Digital Logic Design CS-1101 Engr. Asif Gulraiz	Communication Skills HU-1004 Syeda Bushra Zaidi
BSCS -3		Linear Algebra BS-1302 Moheez Ur Rahim	Pakistan Studies HU-2101 Muhammad Mustafa Raza			Data Structures & Algorithms CS-2001 Aneela Nargis	Discrete Structures CS-2002 Tehniat Mirza [Online]	Computer Organization & Assembly Language CS-2101 Syed Muhammad Farooq Shibli	Technical & Business Writing HU-2009 Kamran Ali
BSCS -4	Theory of Computing CS-208 Tehniat Mirza [Online]			Database Systems CS-204 Khubaib Ahmed Qureshi [Online]	Business Law MGM-454 Shaham Mahmood [Online]	Operating Systems CS-206 Syed Nabeel Shahab		Probability & Statistics MT-206 Sajdah Hassan	
BSCS -5		Data Communication and Computer Networks CS-307 Asif Rafiq [Online]	Design & Analysis of Algorithm Bilal Hayat Butt [Online]		Entrepreneurship MGM-552 Eram Abbasi Financial Accounting AFN-331 Saqib Ghias		Compiler Construction CS-313 Raazia Sosan	Differential Equations MT-203 Muhammad Ashhad Shahid	Introduction to Software Engineering CS-305 Asad Ur Rehman / Conrad Walter D Silva
BSCS -6	Computer Architecture CS-309 Muhammad Azmi Umer [Online]			Machine Learning CS-421 Khubaib Ahmed Qureshi [Online]	Introduction to Supply Chain Management SCM-610 Farhat Tariq Umar Introduction to Psychology SSH-314 Razi Sultan Siddiqui	Artificial Intelligence CS-306 Muhammad Azmi Umer [Online]	Mobile Software Engineering CS-426 Mr. Noushad Tabani [Online] [FF-146]	Numerical Computing MT-301 Moheez Ur Rahim	Information Security CS-308 Asif Rafiq [Online]
BSCS -7	Software Quality Assurance CS-431 Ubaid Aftab Chawla	Social Network Analysis CS-430 Bilal Hayat Butt [Online]	Human Computer Interaction CS-402 Sulaman Ahmed Naz [Online]	Software Project Management CS-481 Asad Ur Rehman	Organizational Behavior MGM-555 Sadia Mahboob Principles of Management MGM-351 Tatheer Yawer	Software Design and Architecture CS-453 Ashar Ali	Big Data Analytics CS-429 Zahid Riaz		Bio Informatics CS-493 Twaha Ahmed Minai [Online]
BSCS -8	Deep Learning CS-492 Muhammad Sufiyan		Professional Issues in IT CS-404 Conrad Walter D Silva	Web Engineering CS-492 Syed Nabeel Shahab / Syed Muhammad Farooq Shibli	E-Business ICT-543 Naila Shahzada E-Business ICT-543 Shaham Mahmood [Online]		High Performance Computing using CUDA CS-320 Raazia Sosan		

13/12/19
TimeTable Coordinator

13/12/19
Dr. M. Mobeen Movania (HoD-CS)

13/12
Dean(EAS)

DHA Suffa University

Department of Computer Science

CS 2001L – Data Structures and Algorithms Lab

Fall 2019

Lab 09 – Trees

Objective:

Learn about implementation of following concepts:

- General Tree
- Binary Tree
- Traversing Binary Trees and General Trees (In order, Pre order, Post order, level order)
- Huffman Coding Algorithm

General Tree:

A general tree is a tree in which each node can have an unlimited outdegree. Each node may have as many children as is necessary to satisfy its requirements.

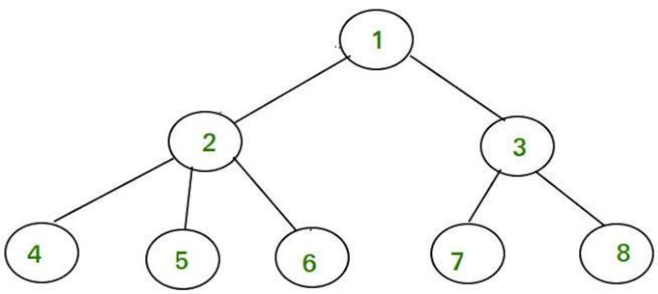


Figure 9.1 – General Tree

Implementation of General Tree:

Node.h

```
#include<iostream>
#include <vector>
struct Node
{
  int data;
  std::vector<Node *> child;
  Node(int data);
  ~Node();
};
```

Node.cpp

```
#include"Node.h"
Node::Node(int data):data(data){}
Node::~Node(){}
```

GeneralTree.h

```
#include"Node.h"
#include<queue>
class GeneralTree{

    public:
        GeneralTree();
        ~GeneralTree();
        Node* insertNode(int data);
        void levelOrderTraversal(Node *root);
};
```

GeneralTree.cpp

```
#include "GeneralTree.h"
GeneralTree:: GeneralTree(){}
GeneralTree:: ~GeneralTree(){}
Node* GeneralTree:: insertNode(int data)
{
    Node *temp = new Node(data);
    return temp;

}

  void GeneralTree:: levelOrderTraversal(Node *root){
      if (root==NULL)
          return;

      std::queue<Node *> q;
      q.push(root);
      while (!q.empty())
      {
          int n = q.size();
          while (n > 0)
          {
              Node * p = q.front();
              q.pop();
              std::cout << p->data << " ";
```

```
                for (int i=0; i<p->child.size(); i++)
                {
                     q.push(p->child[i]);
                }
                n--;
            }

            std::cout << std::endl;
    }
}
```

Driver.cpp

```
#include"GeneralTree.h"
int main()
{
    GeneralTree b;
    Node *root = b.insertNode(7);
    (root->child).push_back(b.insertNode(21));
    (root->child).push_back(b.insertNode(23));
    (root->child).push_back(b.insertNode(25));
    (root->child).push_back(b.insertNode(27));
    (root->child[0]->child).push_back(b.insertNode(33));
    (root->child[0]->child).push_back(b.insertNode(44));
    (root->child[0]->child).push_back(b.insertNode(55));
    (root->child[2]->child).push_back(b.insertNode(66));
    (root->child[2]->child[0]->child).push_back(b.insertNode(3));
    (root->child[3]->child).push_back(b.insertNode(7));
    (root->child[3]->child).push_back(b.insertNode(8));
    (root->child[3]->child).push_back(b.insertNode(9));

    b.levelOrderTraversal(root);
    std::cout<<std::endl;
}
```

Binary Tree

A binary tree **T** is defined as a finite set of elements, called nodes, such that:

a) T is empty (called the null tree or empty tree), or

b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

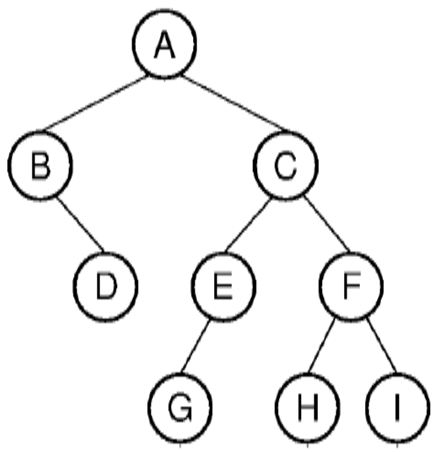


Figure 9.2 – Binary Tree

Implementation of binary tree:

Node.h

```
#include<iostream>
struct Node
{
  int data;
  Node *left;
  Node *right;
  Node(int data);
  ~Node();
};
```

Node.cpp

```
#include"Node.h"
Node::Node(int data):data(data), left(nullptr), right(nullptr){}
Node::~Node(){}
```

BinaryTree.h

```
#include"Node.h"
class BinaryTree{

    public:
        BinaryTree();
        ~BinaryTree();
        Node* insertNode(int data);
        void inorderTraversal(Node *root);
        void preorderTraversal(Node *root);
        void postorderTraversal(Node *root);

};
```

BinaryTree.cpp

```
#include"BinaryTree.h"
BinaryTree:: BinaryTree(){}
BinaryTree:: ~BinaryTree(){}
Node* BinaryTree:: insertNode(int data)
{
    Node *temp = new Node(data);
    return temp;

}
void BinaryTree:: inorderTraversal(Node *root)
{
    if(root == NULL)
        return;
    inorderTraversal(root->left);
    std::cout<<root->data<<" ";
    inorderTraversal(root->right);

}


 void BinaryTree:: preorderTraversal(Node *root)
 {
     if(root == NULL)
         return;
     std::cout<<root->data<<" ";
     preorderTraversal(root->left);
     preorderTraversal(root->right);

 }
```

```
void BinaryTree:: postorderTraversal(Node *root)
{
    if(root == NULL)
        return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    std::cout<<root->data<<" ";

}
```

Driver.cpp

```
#include"BinaryTree.h"
int main()
{
    BinaryTree b;
    Node *root = b.insertNode(7);
    root->left  = b.insertNode(9);
    root->right =b.insertNode(10);
    root->left->left  = b.insertNode(4);
/*           7
        /       \
      9           10
    /   \        /  \
   4     NULL  NULL  NULL
  /  \
NULL NULL
*/
    b.inorderTraversal(root);
    std::cout<<std::endl;
    b.preorderTraversal(root);
    std::cout<<std::endl;
    b.postorderTraversal(root);

}
```

Priority queue:

The standard interface to the FIFO queue simply consists of a enqueue() function, which adds new elements, and a dequeue() function, which always removes the oldest element.

This data structure is relatively easy to implement. However, there are many times when a more sophisticated form of queue is needed.

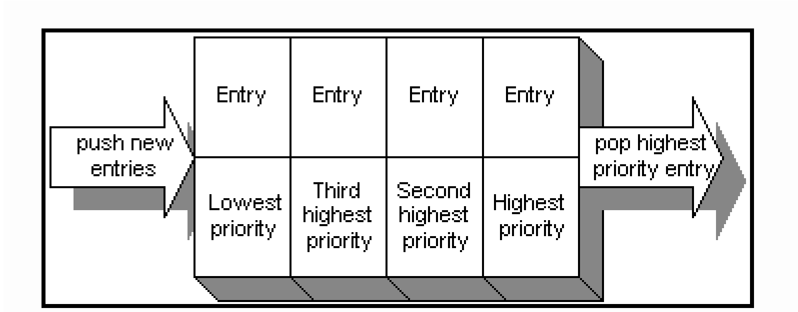


Figure 9.3 – Priority Queue

Figure 9.3 shows a representation of a priority queue. This type of queue assigns a priority to every element that it stores. New elements are added to the queue using the push() function, just as with a FIFO queue. This queue also has a pop() function, which differs from the FIFO pop() in one key area. When you call pop() for the priority queue, you don't get the oldest element in the queue. Instead, you get the element with the highest priority.

The priority queue obviously fits in well with certain types of tasks. For example, the scheduler in an operating system might use a priority queue to track processes running in the operating system.

Operations on Priority Queue :

push(): This function is used to insert a new data into the queue.

pop(): This function removes the element with the highest priority form the queue.

peek() / top(): This function is used to get the highest priority element in the queue without removing it from the queue.

Huffman Coding:

Huffman Coding (also known as Huffman Encoding) is an algorithm for doing data compression and it forms the basic idea behind file compression. The technique works by creating a binary tree of nodes. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the character itself, the weight (frequency of appearance) of the character. Internal nodes contain character weight and links to two child nodes. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has n leaf nodes and n-1 internal nodes. It is recommended that Huffman tree should discard unused characters in the text to produce the most optimal code lengths. We will use priority queue for building Huffman tree where the node with lowest frequency is given highest priority. Below are the complete steps.

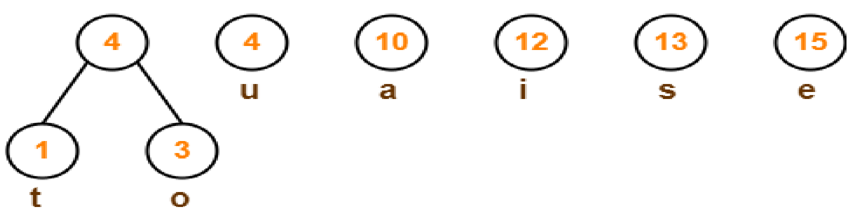
- Create a leaf node for each character and add them to the priority queue.
- While there is more than one node in the queue:
    - Remove the two nodes of highest priority (lowest frequency) from the queue
    - Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes frequencies.
    - Add the new node to the priority queue.
- The remaining node is the root node and the tree is complete.

Example:

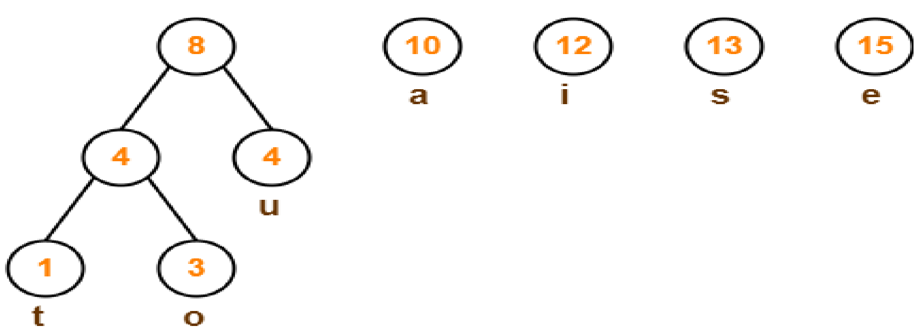
Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Step-01:

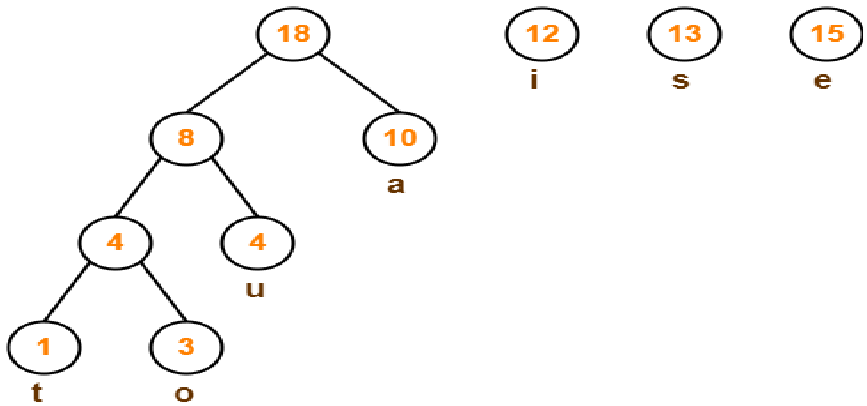
Step-02:



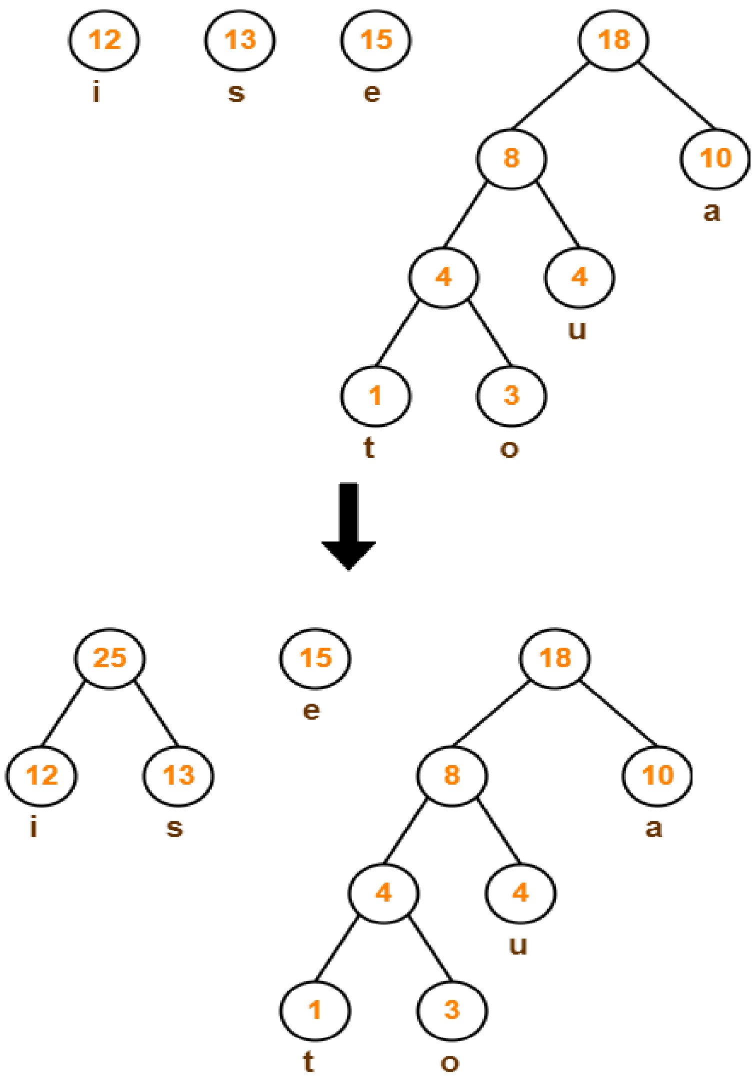
Step-03:



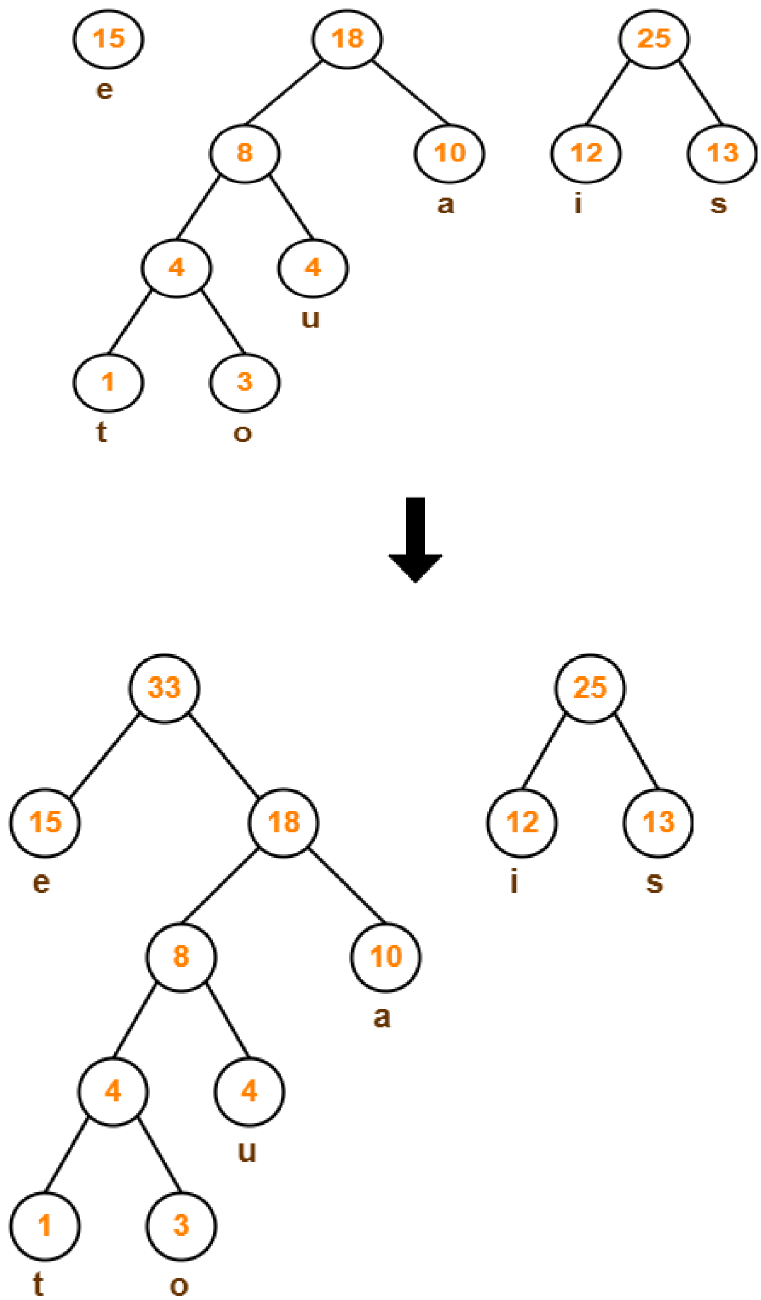
Step-04:



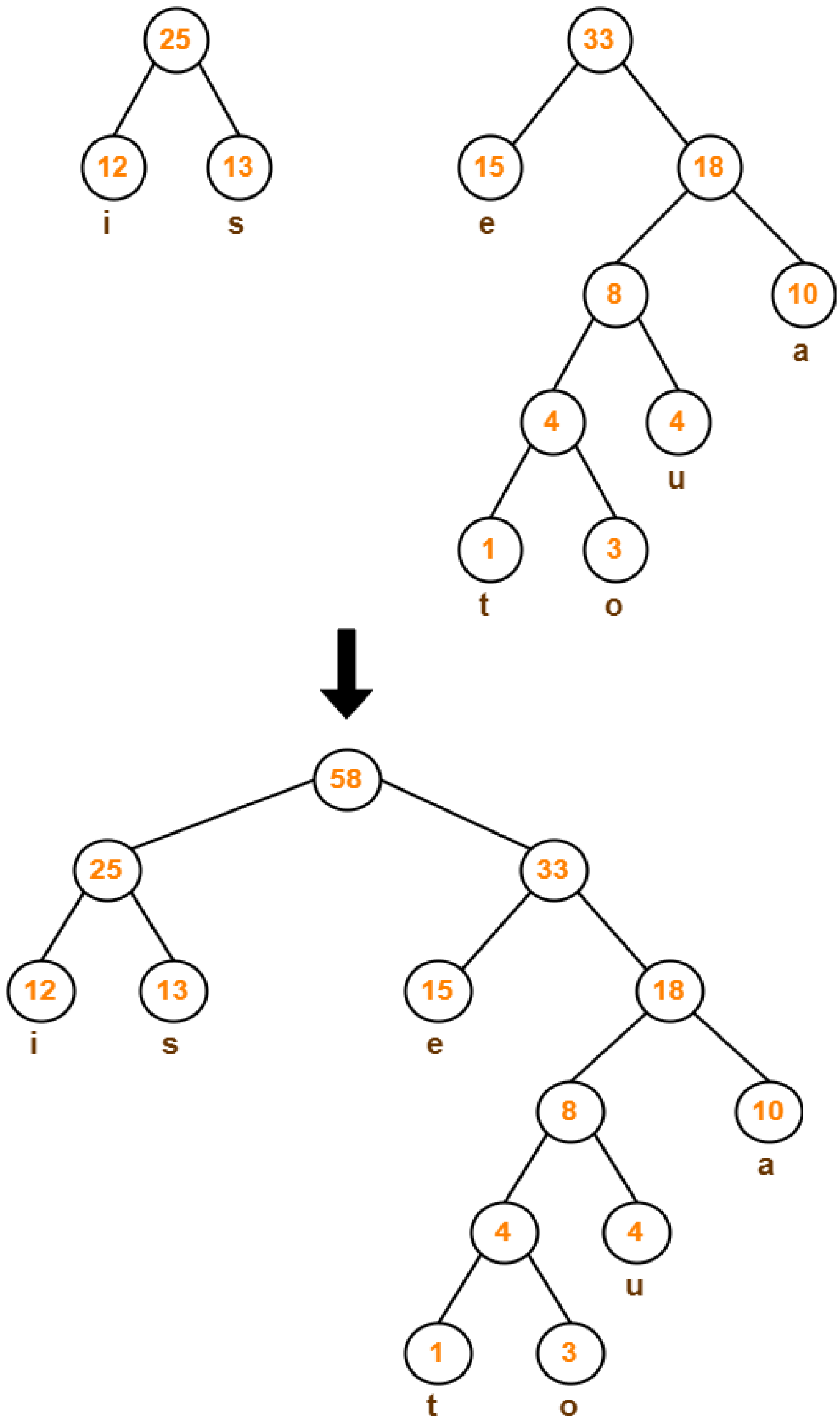
Step-05:



Step-06:


15
e
18
8
10
a
25
12
i
13
s
4
4
u
1
t
3
o
33
15
e
18
25
12
i
13
s
8
10
a
4
4
u
1
t
3
o

Step-07:


25
12
13
i
s
33
15
18
e
8
10
a
4
4
u
1
3
t
o
58
25
33
12
13
i
s
15
18
e
8
10
a
4
4
u
1
3
t
o

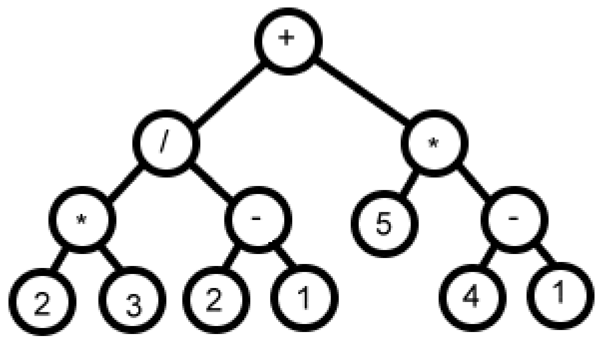
Implementation:

```
#include <iostream>
#include <fstream>
#include <string>
#include <queue>
#include <algorithm>

using namespace std;

struct node {
    int frequency;
    unsigned char value;
    node *left;
    node *right;

    node( unsigned char c, int i ) {
        value = c;
        frequency = i;
        left = NULL;
        right = NULL;
    }

    node( node* c0, node *c1 ) {
        value = 0;
        frequency = c0->frequency + c1->frequency;
        left = c0;
        right = c1;
    }


   bool operator<( const node &a ) const {
       return frequency >a.frequency;
   }

   void traverse(string code="") const {

   if ( left ) {
       left->traverse( code + '0' );
       right->traverse( code + '1' );
   }
   else {
       cout <<" " <<value <<"    ";
       cout <<frequency;
       cout <<"     " <<code <<endl;
     }
  }
};
```

```
void count_chars( int *counts )
{
    for ( int i = 0 ; i <256 ; i++ )
        counts[ i ] = 0;
    ifstream file( "input.txt" );
    if ( !file ) {
        cout <<"Couldn't open the input file!\n";
        exit(0);
    }
   else
    {
        for ( ; ; ) {
        unsigned char c;
        file>> c;
        if ( file )
            counts[ c ]++;
        else
            break;
        }
    }
}

 int main()
 {
     int counts[ 256 ];
     count_chars( counts );
     priority_queue < node > q;

     for ( int i = 0 ; i <256 ; i++ )
         if ( counts[ i ] )
             q.push( node( i, counts[ i ] ) );

     while ( q.size() >1 ) {
         node *child0 = new node( q.top() );
         q.pop();
         node *child1 = new node( q.top() );
         q.pop();
         q.push( node( child0, child1 ) );
     }

     cout <<"CHAR  FREQUENCY  HUFFMAN-CODE" <<endl;
     q.top().traverse();
     return 0;
 }
```

Assignment:

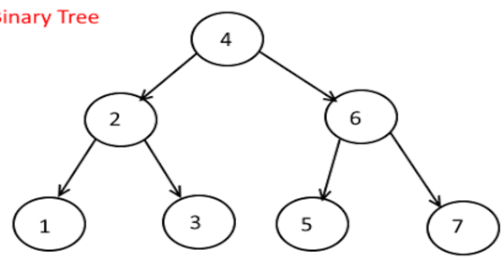
Q.1) Evaluate the expression tree consisting of basic binary operators i.e., + , – ,* and / and some integers, the expression tree for evaluation is given below.



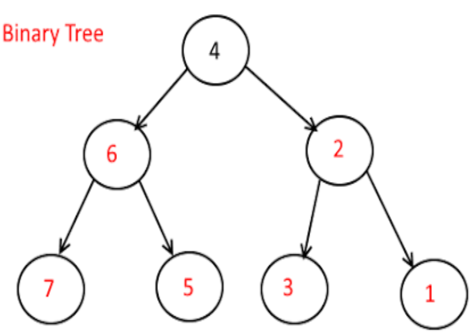
Your program should print the resultant value after evaluation as output.

Q.2) Convert the following binary Tree into its mirror tree. Write a separate function mirror() for the conversion.

Given Binary Tree



MIrror Binary Tree



Submission Guidelines

- **Write C++ code , separate function for each operation.**
- **Place your file in a folder named with your rollNo (cs172xxx) where xxx is your 3 digit rollno.**
- **Upload it on LMS.**