**BFS**

Code:

#include <stdio.h>

#include <stdlib.h>

struct btnode

{

int value;

struct btnode \*left, \*right;

};

typedef struct btnode node;

/\* function declarations \*/

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

void bfs\_traverse(node \*);

/\*global declarations \*/

node \*root = NULL;

int val, front = 0, rear = -1, i;

int queue[20];

void main()

{

node \*new = NULL ;

int num = 1;

printf("Enter the elements of the tree(enter 0 to exit)\n");

while (1)

{

scanf("%d", &num);

if (num == 0)

break;

new = malloc(sizeof(node));

new->left = new->right = NULL;

new->value = num;

if (root == NULL)

root = new;

else

{

insert(new, root);

}

}

printf("elements in a tree as per BFS traversal are\n");

queue[++rear] = root->value;

bfs\_traverse(root);

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

printf("%d -> ", queue[i]);

printf("%d\n", root->right->right->right->value);

}

/\* inserting nodes of a tree \*/

void insert(node \* new , node \*root)

{

if (new->value>root->value)

{

if (root->right == NULL)

root->right = new;

else

insert (new, root->right);

}

if (new->value < root->value)

{

if (root->left == NULL)

root->left = new;

else

insert (new, root->left);

}

}

/\* displaying elements using BFS traversal \*/

void bfs\_traverse(node \*root)

{

val = root->value;

if ((front <= rear)&&(root->value == queue[front]))

{

if (root->left != NULL)

queue[++rear] = root->left->value;

if (root->right != NULL || root->right == NULL)

queue[++rear] = root->right->value;

front++;

}

if (root->left != NULL)

{

bfs\_traverse(root->left);

}

if (root->right != NULL)

{

bfs\_traverse(root->right);

    }

}

A screenshot of a computer

Description automatically generated

**DFS**

Code:

#include <stdio.h>

#include <stdlib.h>

/\* ---------------------------------- Data Structures To represent a Graph ---------------------------------- \*/

// Structure to Represent a Node of Adjacency List

struct node

{

// Vertex Number will represent the Vertex which is connected from the Node

// To which the adjacency list is associated

int vertexNumber;

// This pointer will be used to point to the next vertex in the adjacency list

// NULL will denote that this vertex was the last connected vertex

// with the vertex associated with the adjacency list

struct node \*pointerToNextVertex;

};

// Structure to Represent the Graph in C

struct Graph

{

// Total Number of Vertices will be needed so that we can know upto which node we have to traverse

int numberOfVertices;

// A boolean flag will be maintained to know whether we have already visited the node or not

int \*visitedRecord;

// Adjacency Lists is a 2 dimensioanl array

// It will be used to maintain the adjacency list for each vertex of the graph

// For example: adjacencyLists[1] will denote the adjacency list of 1st vertex of graph

// And it will contain all nodes connected to the 1st vertex of the graph

struct node \*\*adjacencyLists;

};

/\* ---------------------------------- Required Helper Functions ---------------------------------- \*/

// Function to create a Node

// It will be used to create the node for which the structure is defined above

// Parameters: An integer Vertex, to represent the vertex number

struct node \*createNodeForList(int v)

{

// Use malloc to dynamically allocate Memory

struct node \*newNode = malloc(sizeof(struct node));

// Allocate the vertex Number

// Means the "v" vertex is connected to the vertex whose adjacency list contains this entire node

newNode->vertexNumber = v;

// Assign the next vertex to NULL

// COnsider it is the last connected vertex

newNode->pointerToNextVertex = NULL;

return newNode;

}

// Function to add Edge in the Graph

// Parameters: A pointer to Graph Structure and Edge (Source to Destination)

void addEdgeToGraph(struct Graph \*graph, int source, int destination)

{

// Create New Node it is required before adding Edge to Graph

// Provide Destination as the vertexNumber because the source will be used to access the required node

// We will add this node to the adjacency list of source Node

struct node \*newNode = createNodeForList(destination);

// Now provide the pointer stored for the adjacency list of source node to the newNode

newNode->pointerToNextVertex = graph->adjacencyLists[source];

// And assign the new Node as the starting point of our adjacency list of the source node

// In this way we are adding the newNode at the starting of the adjacency list

// because traversing to the end and then inserting will be an overhead

graph->adjacencyLists[source] = newNode;

// In the similar way we have to add edge from destination to source

// Because we are working with undirected graphs

newNode = createNodeForList(source);

newNode->pointerToNextVertex = graph->adjacencyLists[destination];

graph->adjacencyLists[destination] = newNode;

}

// Function to Create Graph

// Parameters: Number of Vertices

struct Graph \*createGraph(int vertices)

{

// Declare iterator variable for loop purpose

int i;

// Create graph Structure

struct Graph \*graph = malloc(sizeof(struct Graph));

// Set the number of vertices

graph->numberOfVertices = vertices;

// Create the total "vertices" adjacency lists, means the array of adjacency lists

// Because still if we don't have the connection with any particular vertex

// The empty adjacency list is necessary

graph->adjacencyLists = malloc(vertices \* sizeof(struct node \*));

// Create the array visitedRecord to store the information about which nodes have been visited till now.

graph->visitedRecord = malloc(vertices \* sizeof(int));

// Initialize the visited record with 0, because at the time of the creation of a graph

// None of the vertex is visited

// And initialize the all adjacencyLists with NULL, because we don't have any edges while creating the graph

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

{

graph->adjacencyLists[i] = NULL;

graph->visitedRecord[i] = 0;

}

// Return the graph structure to use further

return graph;

}

/\* ---------------------------------- Depth First Search Algorithm ---------------------------------- \*/

// Function to apply Depth First Search on graph

// Parameter: Graph and starting vertex Number, because it is necessary to define a starting point

void depthFirstSearch(struct Graph \*graph, int vertexNumber)

{

struct node \*adjList = graph->adjacencyLists[vertexNumber];

struct node \*temp = adjList;

graph->visitedRecord[vertexNumber] = 1;

printf("%d ", vertexNumber);

while (temp != NULL)

{

int connectedVertex = temp->vertexNumber;

if (graph->visitedRecord[connectedVertex] == 0)

{

depthFirstSearch(graph, connectedVertex);

}

temp = temp->pointerToNextVertex;

}

}

int main()

{

int numberOfVertices, numberOfEdges, i;

int source, destination;

int startingVertex;

printf("Enter Number of Vertices and Edges in the Graph: ");

scanf("%d%d", &numberOfVertices, &numberOfEdges);

struct Graph \*graph = createGraph(numberOfVertices);

printf("Add %d Edges of the Graph(Vertex numbering should be from 0 to %d)\n", numberOfEdges, numberOfVertices - 1);

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

{

scanf("%d%d", &source, &destination);

addEdgeToGraph(graph, source, destination);

}

printf("Enter Starting Vertex for DFS Traversal: ");

scanf("%d", &startingVertex);

if (startingVertex < numberOfVertices)

{

printf("DFS Traversal: ");

depthFirstSearch(graph, startingVertex);

}

    return 0;

}

A screenshot of a computer

Description automatically generated