

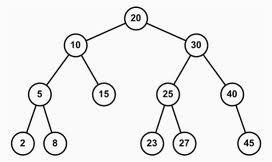
MAYANK GUPTA 20BCE1538

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Programme | : | **B.Tech** | Semester | : | **Fall 2021-22** |
| Course | : | **Data structures and Algorithms** | Code | : | **CSE2003** |
| Faculty | : | **Dr.V.Vani** | Slot | : | **L27+L28** |

AIM Depth First Search and Breadth First Search

**AIM**

1. Consider the below tree with 12 nodes, start from the root node and explore all the nodes in each level and reach the node 45.



#include <iostream> #include <queue>

template <typename T> class BinaryTreeNode {

public:

T data; BinaryTreeNode<T> \*left; BinaryTreeNode<T> \*right;

BinaryTreeNode(T data) { this->data = data;

left = NULL; right = NULL;

}

BinaryTreeNode() { if (left) delete left;

if (right) delete right;

}

};

using namespace std;

bool searchInBST(BinaryTreeNode<int> \*root , int k) {

// Write your code here if(root==NULL){

return false;

}

if(root->data==k){ return true;

}

else if(root->data>k){

return searchInBST(root->left ,k);

}

else if(root->data<k){

return searchInBST(root->right ,k);

}

}

BinaryTreeNode<int> \*takeInput() { int rootData;

cin >> rootData;

if (rootData == -1) { return NULL;

}

BinaryTreeNode<int> \*root = new BinaryTreeNode<int>(rootData); queue<BinaryTreeNode<int> \*> q;

q.push(root);

while (!q.empty()) {

BinaryTreeNode<int> \*currentNode = q.front(); q.pop();

int leftChild, rightChild; cin >> leftChild;

if (leftChild != -1) {

BinaryTreeNode<int> \*leftNode = new BinaryTreeNode<int>(leftChild); currentNode->left = leftNode;

q.push(leftNode);

}

cin >> rightChild;

if (rightChild != -1) { BinaryTreeNode<int> \*rightNode =

new BinaryTreeNode<int>(rightChild); currentNode->right = rightNode; q.push(rightNode);

}

}

return root;

}

void printLevelWise(BinaryTreeNode<int> \*root) {

// Write your code here queue<BinaryTreeNode<int>\*> pendingNode; pendingNode.push(root); while(!pendingNode.empty()){

BinaryTreeNode<int> \*front=pendingNode.front(); pendingNode.pop();

cout<<front->data<<":"; if(front->left!=NULL){

cout<<"L:"<<front->left->data; pendingNode.push(front->left);

}

else{

cout<<"L:"<<-1;

}

cout<<",";

if(front->right!=NULL){ cout<<"R:"<<front->right->data; pendingNode.push(front->right);

}

else{

cout<<"R:"<<-1;

}

cout<<endl;

}

}

int main() {

BinaryTreeNode<int> \*root = takeInput(); int k;

cin >> k;

cout << ((searchInBST(root, k)) ? "true" : "false")<<endl; printLevelWise(root);

if(searchInBST(root,k)){ printLevelWise(root);

}

else{

cout<<"Element not present";

}

delete root;

}

## BFS algorithm

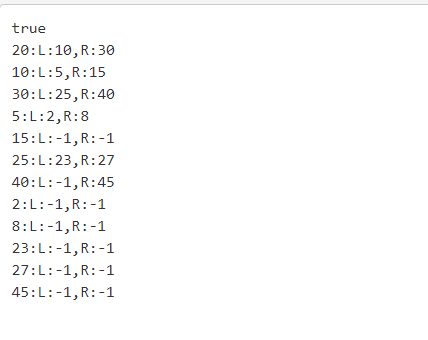
A standard BFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

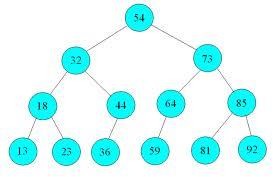
The algorithm works as follows:

* Start by putting any one of the graph's vertices at the back of a queue.
* Take the front item of the queue and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
* Keep repeating steps 2 and 3 until the queue is empty.
* The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node



**AIM**

1. Consider the below tree with 13 nodes, start from the root node and explore all the nodes and reach the node 92 using DFS.



#include <iostream> #include <queue>

template <typename T> class BinaryTreeNode {

public:

T data; BinaryTreeNode<T> \*left; BinaryTreeNode<T> \*right;

BinaryTreeNode(T data) { this->data = data;

left = NULL; right = NULL;

}

BinaryTreeNode() { if (left) delete left;

if (right) delete right;

}

};

using namespace std;

bool searchInBST(BinaryTreeNode<int> \*root , int k) {

// Write your code here if(root==NULL){

return false;

}

if(root->data==k){ return true;

}

else if(root->data>k){

return searchInBST(root->left ,k);

}

else if(root->data<k){

return searchInBST(root->right ,k);

}

}

BinaryTreeNode<int> \*takeInput() { int rootData;

cin >> rootData;

if (rootData == -1) { return NULL;

}

BinaryTreeNode<int> \*root = new BinaryTreeNode<int>(rootData); queue<BinaryTreeNode<int> \*> q;

q.push(root);

while (!q.empty()) {

BinaryTreeNode<int> \*currentNode = q.front(); q.pop();

int leftChild, rightChild; cin >> leftChild;

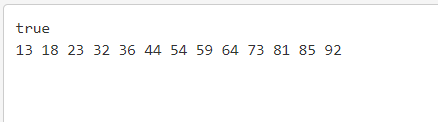
if (leftChild != -1) {

BinaryTreeNode<int> \*leftNode = new BinaryTreeNode<int>(leftChild); currentNode->left = leftNode;

q.push(leftNode);

}

cin >> rightChild;



if (rightChild != -1) { BinaryTreeNode<int> \*rightNode =

new BinaryTreeNode<int>(rightChild); currentNode->right = rightNode; q.push(rightNode);

}

}

return root;

}

void inorder(BinaryTreeNode<int> \*tree)

{

if (tree != NULL)

{

inorder(tree->left);

cout << tree->data << " "; inorder(tree->right);

}

}

int main() {

BinaryTreeNode<int> \*root = takeInput(); int k;

cin >> k;

cout << ((searchInBST(root, k)) ? "true" : "false")<<endl; if(searchInBST(root,k)){

inorder(root);

}

else{

cout<<"Element not present";

}

delete root;

}

## Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

* Start by putting any one of the graph's vertices on top of a stack.
* Take the top item of the stack and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
* Keep repeating steps 2 and 3 until the stack is empty.