**1. Problem Discussion :**

In this problem you are given a partially written BinaryTree class. You are given an element. You are required to complete the body of find and nodeToRoot function. The functions are expected to find -> return true or false depending on if the data is found in a binary tree. nodeToRoot -> returns the path from node (corresponding to data) to root in form of an arraylist (root being the last element)

If you are finding it difficult to understand then we recommend you to watch the question video of this topic. I am sure that question will get much clearer once you watch it.

**2. Approach :**

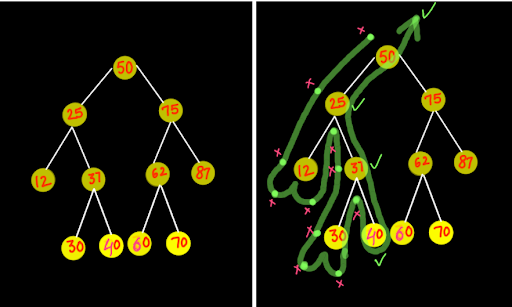
We can easily code both the functions of this problem using Recursion. Both the functions are quite similar. If you remember that Recursion has two stages, High level thinking and Low level thinking. High level thinking has three parts:

1• Expectation 2• Faith 3• How to meet Expectation with Faith

With high level thinking, we can figure out our basic code. Then coming to low level thinking. This helps us to identify our base case. And with both high and low level thinking, we will finally get our code.

Let's start with the "find" function first:

Consider that we need to find 40 in the given binary tree given below:

****

Developing high-level thinking for the find function:

● Expectation: Our expectation from find(50, 40) is that it will check whether 40 is present in our tree or not. If it's present then functions return true else false. ● Faith: Our faith is, find(25, 40) and find(75, 40) works fine and know exactly how to find an element in their respective subtrees. ● How to meet Expectation with Faith: If either of the left or right subtree returns true then it means that 40 is present in the given binary tree. And if both returns false, then 40 is present in neither the left nor right subtree. Therefore, there is only one possibility left to check that is at root (50) itself. So this will be our self work which is to check the data of the current node.

Concluding Final Steps:

It is important that we first check if the given element is present at the current node or not because if it is present then we can return true from this point only without making any further calls. Otherwise we first recursively call the left child of the node keeping the faith that find(node.left, element) works perfectly. Then we check whether this call returns true or not. If it does then we return true for this function also from here only as there is no need to check for element further. Else we recursively call the right child of the node keeping the faith that find(node.left, element) works perfectly. Then we check whether this call returns true or not. If it does then we return true for this function also from here only as there is no need to check for element further. And if control comes out of all the above conditions then it means the element was not present in the given tree. So at last we return false.

Talking of low level thinking, we try to identify our base case.

After doing so many problems it is quite easy to guess that the base case hits when node becomes null. So, in this case if a call is made to a null node then we return false.

Moving to the "node to root path" function:

This function is completely based on the find function discussed above. Only a few additional steps and changes and we are done. Here we not only traverse the tree and check for every node whether the current node is the desired node or not but also keep the track of the path back to the root. How? For this we primarily change the return type of the function to ArrayList< Integer> which contains the path. Why? We do this, so that after we are done checking for every node whether the current node is the desired node or not and we find an equal element then we return an ArrayList having the node value in it. And if we reach the end of the branch then we return an empty ArrayList. If in any case (either from node.left or node.right call), we get a non-empty ArrayList then we add the current node to the list and return immediately without considering the following branches as the search there would be redundant. So in the end, we will have a path from the desired node, way back to the root node. But it's also important to take care of the case when an element is not present in the tree. In such a case, after all lines are executed and control reaches the end of the function, at this point we return a new Arraylist.

**3. Code Implementation :**

ConsoleCpp

#include <iostream>

#include <string.h>

#include <vector>

using namespace std;

class Node {

public:

int data;

Node\* left = nullptr;

Node\* right = nullptr;

Node(int data)

{

this->data = data;

}

};

class Pair {

public:

Node\* node = nullptr;

int state = 0;

Pair(Node\* node, int state)

{

this->node = node;

this->state = state;

}

};

int idx = 0;

Node\* constructTree(vector<int>& arr) {

if (idx == arr.size() || arr[idx] == -1) {

idx++;

return nullptr;

}

Node\* node = new Node(arr[idx++]);

node->left = constructTree(arr);

node->right = constructTree(arr);

return node;

}

void display(Node\* node) {

if (node == nullptr)

return;

string str = "";

str += node->left != nullptr ? to\_string(node->left->data) : ".";

str += " <- " + to\_string(node->data) + " -> ";

str += node->right != nullptr ? to\_string(node->right->data) : ".";

cout << str << endl;

display(node->left);

display(node->right);

}

bool find(Node\* node, int data) {

if (node == nullptr)

return false;

if (node->data == data)

return true;

return find(node->left, data) || find(node->right, data);

}

bool rootToNodePath(Node\* node, int data, vector<Node\*>& path)

{

if (node == nullptr)

return false;

if (node->data == data)

{

path.push\_back(node);

return true;

}

bool res = rootToNodePath(node->left, data, path) || rootToNodePath(node->right, data, path);

if (res)

path.push\_back(node);

return res;

}

vector<int> nodeToRootPath(Node\* node, int data)

{

if (node == nullptr) {

vector<int> arr;

return arr;

}

if (node->data == data) {

vector<int> list;

list.push\_back(node->data);

return list;

}

vector<int> llist = nodeToRootPath(node->left, data);

if (llist.size() > 0) {

llist.push\_back(node->data);

return llist;

}

vector<int> rlist = nodeToRootPath(node->right, data);

if (rlist.size() > 0) {

rlist.push\_back(node->data);

return rlist;

}

vector<int> arr;

return arr;

}

int main() {

int n;

cin >> n;

vector<int> arr(n, 0);

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

string temp;

cin >> temp;

if (temp == "n") {

arr[i] = -1;

} else {

arr[i] = stoi(temp);

}

}

Node\* root = constructTree(arr);

int data;

cin >> data;

bool found = find(root, data);

found == 1 ? cout << "true" << endl : cout << "false" << endl;

vector<int> path = nodeToRootPath(root, data);

cout << "[";

for (int i = 0; i < path.size(); i++) {

cout << path[i];

if (i != path.size() - 1) {

cout << ", ";

}

}

cout << "]" << endl;

}

**4. Complexities :**

Time Complexity:

O(n) The time complexity for the function is linear as tree traversal is involved which is linear in terms of time complexity.

Space Complexity:

O(logn) The space complexity for the function is proportional to the height of the tree due to the recursion stack.