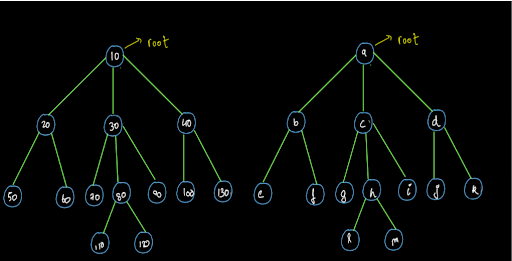
**1. Problem Discussion :**

****

In the diagram above, the data in both the trees is different, still they are of the same shape. (Note: The Node that we have created has data of integer type. Hence you will not be able to compare it with a tree which contains characters. This is shown just for the sake of an example to show that data can be different.)

You may refer to the question video if you have any doubts regarding the question. We recommend you try to solve this problem on your own first and then go to the solution.

**2. Approach :**

Time Complexity:

O(n) where n is the number of nodes in the trees

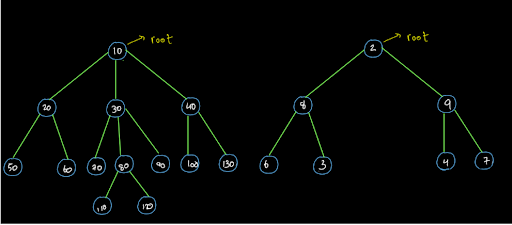
Space Complexity:

O(1)

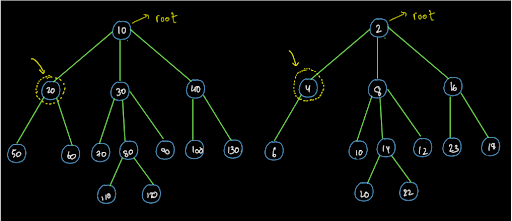
So dear reader, we hope that you have tried to solve this problem by yourself. Now let us move to the solution and understand how to solve this problem. We will solve it using recursion.

Checking the Child Nodes of the Root Node:

Consider the following example:

****

Do we even need to check by applying any procedure whether these trees are the same or not? No!!! This is because the root nodes of both these trees have different number of child nodes. So, the very first step will be to check whether the number of child nodes of the root nodes of both the trees is the same or not. But is it valid for only the root node? Well, this is valid for every node.

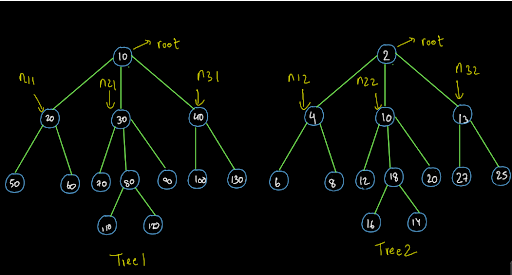
****

In the above diagram we can clearly see that the two trees are not of the same shape as one of the nodes does not have the child nodes equal to the corresponding node in the other tree. So, the initial step is to check whether the number of child nodes is equal or not.

Wait: Is this the initial step? We say that this is the only step that we need to perform at every single node. This is because the data can be different so order doesn't even matter. All that matters is that every node of both the trees should have the same shape which means that every node should have the same number of child nodes as the corresponding node in the other tree. We will just apply this procedure recursively now.

High-Level Thinking:

Let us have a look at the diagram given below:

****

We have already established that the number of child nodes of the root node should be the same for us to begin the procedure. Now, after that we can break our problem into smaller problems and solve it using recursion.

As shown above we have mentioned the 3 child nodes of tree1 as n11 (node one of tree 1), n21 (node two of tree 1) and n31 (node 3 of tree 1) respectively. Similarly we have named the corresponding nodes in the second tree as n12, n22 and n32.

The trees will be the same in shape if the sub-trees from n11 and n21 are the same and the sub-trees from n21 and n22 are the same and the sub-trees from n31 and n32 are the same. So, let us now break this procedure down to:

Expectation: We expect that the function areSimilar(node1,node2) will return true if the trees have the same shape and it will return false if they have a different shape.

Faith: We have a faith on the recursion that recursion can tell us whether the sub-trees of the same given trees are of the same shape or not.

Relation: We will first find whether the number of child nodes of the root nodes of both the trees is same or not. If it is then we will check whether the corresponding sub-trees of both the trees have the same shape or not. If we find at any node that the shape is not the same, we will return false otherwise if we complete the entire procedure and find no difference then we will return true.

Now that we have done the high level thinking, let us write the code for the above procedure.

**3. Pseudo Code**

1• Check Number of Child Nodes: For the trees to be of the same shape the number of child nodes of the root nodes of both the trees must be the same. If they are not the same, return false otherwise continue the procedure. 2• High Level Thinking: After checking the number of nodes, we must ensure that the subtree which is starting from each child node must be the same in shape as the corresponding subtree of the other tree. So, we will use our faith in recursion to achieve it by calling the function areSimilar(node1,node2) for all the children of both the trees. 3• Low Level Thinking:We don't have to find a base case as such. We will do the low level thinking for understanding the working of the code that we wrote using the high level thinking in a better way.

**4. Code Implementation :**

ConsoleCpp

#include<bits/stdc++.h>

#include<climits>

using namespace std;

class Node

{

public:

int data = 0;

vector<Node \*> children;

Node(int data)

{

this->data = data;

}

};

void display(Node \*node)

{

string s = "";

s += to\_string(node->data) + " Child: ";

for (Node \*child : node->children)

{

s += to\_string(child->data) + ", ";

}

cout << s << "." << endl;

for (Node \*child : node->children)

{

display(child);

}

}

Node \*constructor01(vector<int>arr)

{

if (arr.size() == 0)

return NULL;

vector<Node \*> stack;

stack.push\_back(new Node(arr[0]));

Node \*root = stack[0];

for (int i = 1; i < arr.size(); i++)

{

if (arr[i] != -1)

{

Node \*node = stack.back();

Node \*nnode = new Node(arr[i]);

node->children.push\_back(nnode);

stack.push\_back(nnode);

}

else

stack.pop\_back();

}

return root;

}

int sizeRec(Node \*node)

{

int size = 0;

for (Node \*child : node->children)

{

size += sizeRec(child);

}

return size + 1;

}

int maxEle(Node \*node)

{

int OverMax = node->data;

for (Node \*child : node->children)

{

OverMax = max(OverMax, maxEle(child));

}

return OverMax;

}

int height(Node \*node)

{

int ht = 0;

for (Node \*child : node->children)

{

ht = max(ht, height(child));

}

return ht + 1;

}

bool areSimilar(Node \*n1, Node \*n2)

{

if(n1->children.size() != n2->children.size())

{

return false;

}

for(int i = 0; i < n1->children.size(); i++)

{

Node \*c1 = n1->children[i];

Node \*c2 = n2->children[i];

if(areSimilar(c1,c2) == false)

{

return false;

}

}

return true;

}

void solve()

{

int n1;

cin>>n1;

vector<int>arr1(n1,0);

for(int i = 0; i < n1; i++)

{

cin>>arr1[i];

}

Node \*root1 = constructor01(arr1);

int n2;

cin>>n2;

vector<int>arr2(n2,0);

for(int i = 0; i < n2; i++)

{

cin>>arr2[i];

}

Node \*root2 = constructor01(arr2);

bool similar = areSimilar(root1, root2);

if(similar == true)

{

cout<<"true"<<endl;

}

else

{

cout<<"false"<<endl;

}

}

int main()

{

solve();

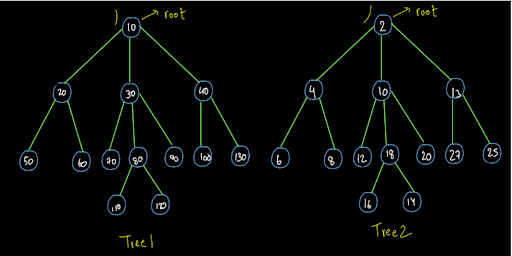
return 0;

}

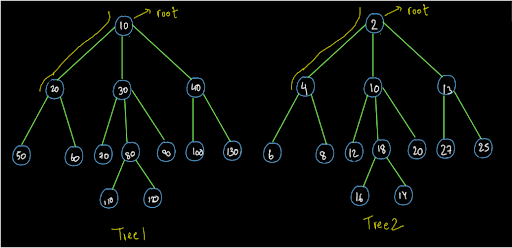
Dear reader, we hope that you got the procedure as well as the code. If you have any doubts regarding the above procedure or the code, you may refer to the solution video to understand the procedure and the code in detail. Now let us do the low-level analysis of this code and understand how this code is working.

Low-Level-Analysis:

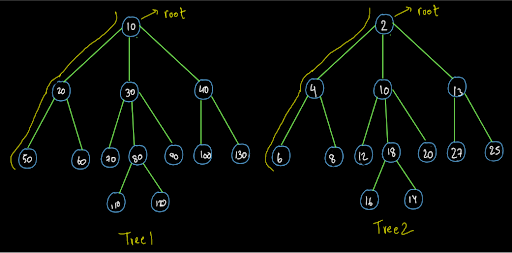
We recommend you have the code by your side to understand this low level analysis. We are initially at the root nodes of both the trees. We check whether the number of child nodes of the root nodes of both the trees is the same or not.

****

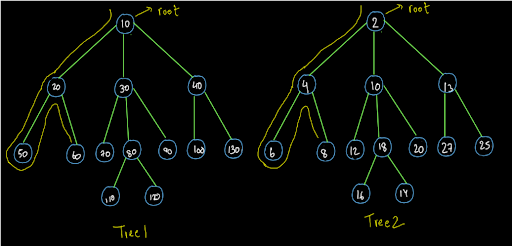
Since they were the same, we entered the loop inside the code. Here we move to the first child of both the trees.

****

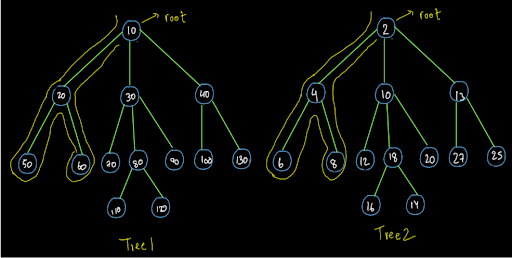
Since both of them also have the same number of nodes therefore we enter the loop in the code and we move to the first child nodes of these nodes respectively.

****

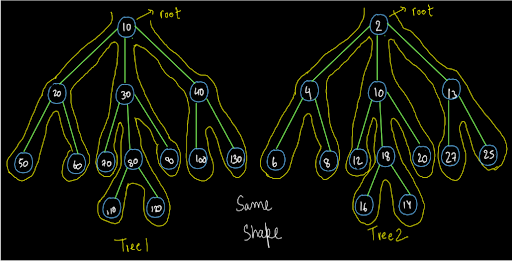
Since both of them are leaf nodes, the number of nodes i.e. 0 is the same for both of them. Now we will not enter the loop as there is no child node for these nodes. Therefore, their code is completed. So, they will wipe out from the recursion stack and we will return to node (20) and node(4) respectively. Since we are still inside the loop for both these nodes, the call will be made for their second child nodes and we will head there in our Euler paths.

****

Again, they both are leaf nodes. So, we will return to node (20) and node (4) respectively. Since all its child nodes have been traversed, we will return back to the root nodes of both the trees.

****

So, dear reader, we now request you to complete this procedure further in the same way as taught above. If you still have any doubts regarding this procedure, you may refer to the solution video to clear all your doubts. For the above trees, you should have completed the Euler path and found that both of them are same in shape.

****

Now let us analyze the time and space complexity of the above procedure.

**5. Analysis:**

Time Complexity:

O(n) The time complexity will be O(n) as we have traversed every node of both the trees if they are of the same shape.

Space Complexity:

The space complexity is O(1) as we have not used any extra data structure. As usual if we consider the recursion space then the space complexity will be O(logn) as the max height of the stack at any point will be the same as the height of the tree i.e. O(logn).

So dear reader, we hope that you got the entire procedure as well as the code and the time and space complexity analysis. If you still have any doubts then you may refer to the complete solution video to clear all your doubts. With this, we have completed this problem.