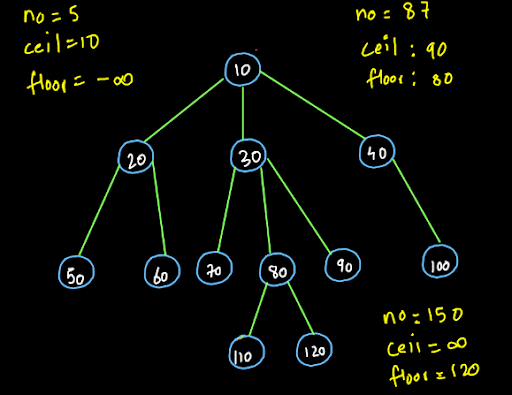
**1. Problem Explanation:**

****

We are given a generic tree and we receive an input element. We have to find the ceil and floor of that value in the given generic tree.

Ceil of a value:

The ceil of a value is the just next greater value for the element. For example : in the above generic tree, the ceiling of number 87 is 90. So, ceil can also be called the minimum value out of the larger values for any number.

Floor of a value:

Floor of a value means the just smaller element for the given value. For example: in the above generic tree the floor value of 87 is 80. So, floor can also be called the maximum value out of the smaller values for a particular number.

As demonstrated in the diagram, if we enter a value which is smaller than all the values in the tree, then we can get the ceil as the smallest value of the tree but we cannot get a floor value as no value in the tree is smaller than the input value. So, the floor becomes minus infinity in this case.

Similarly, if we enter a value which is greater than all the elements of the tree then we will not get the ceil value. The ceil value in such a case will be infinity.

You may refer to the question video if you still have any doubts about the question. We suggest you try to solve this problem on your own first and then move to the solution.

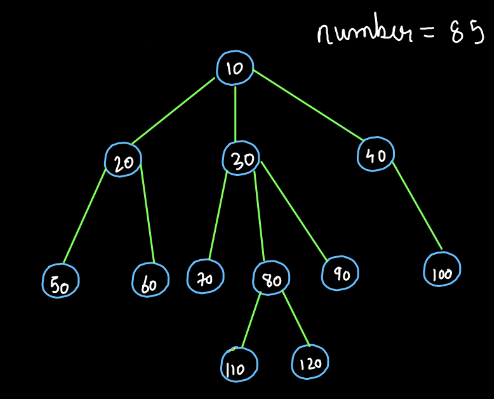
Algorithm:

1• We will again follow the "traverse and change" approach that we followed in the previous question Predecessor and successor of an element We will traverse through all the nodes and if the data inside the node is greater than the input value, we will store the value inside the ceil. This will be a potential ceil value. Similarly, if the data inside the node is less than the input value, we will store it in the floor variable. This will also be a potential floor value. 2• If we find some other node where the value of the node is smaller than the current ceil value then we will change the ceil value to it. Similarly if we find any other node where the value is larger than the current floor value, we will store it in floor. 3• This process will continue till we traverse the entire tree and then, we will print the values of the ceil and floor.

**2. Approach :**

Traverse and Change:

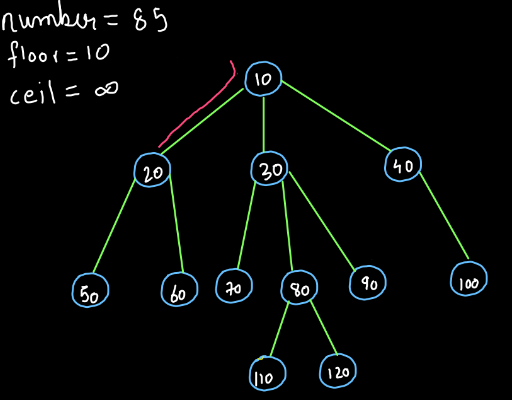
Have a look at the diagram given below:

****

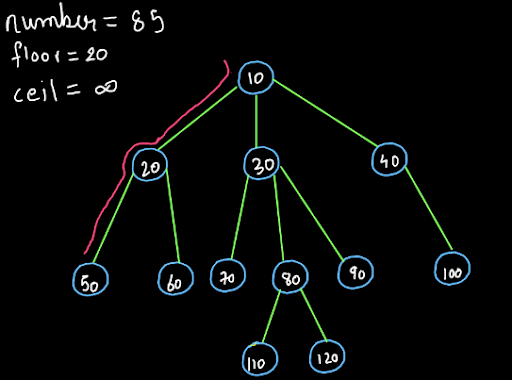
We are given a generic tree and the element that we are given as input is 85. We have to find the ceil and floor value for this element. We will trace the Euler path and follow the traverse and change algorithm. Before that we also have to do one thing:

We know that if the value that we receive as the input is greater than all the values then its ceil will be infinity and if the input value is smaller than all the values in the tree then the floor will be minus infinity. Also, ceil is the minimum value in the larger values and floor is the maximum value in the smaller values. So, to find ceil we will have to find the minimum and for finding floor, we will have to find the maximum. We know that minus infinity is the identity for finding maximum and infinity is the identity to find the minimum. Therefore, we will keep ceil as infinity and floor as minus infinity before we begin with the procedure

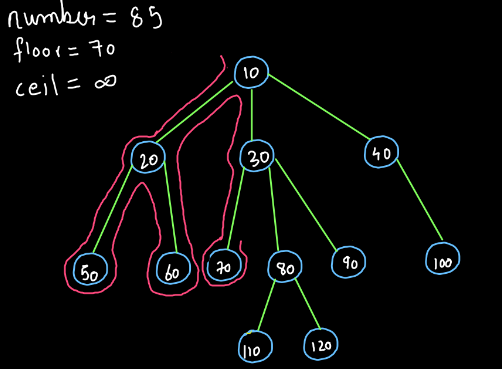
Now, we start from the root node. Since the value at the root node is 10, it is smaller than 85. Hence it is a potential floor. Since it is not greater than 85, ceil will remain infinity.

****

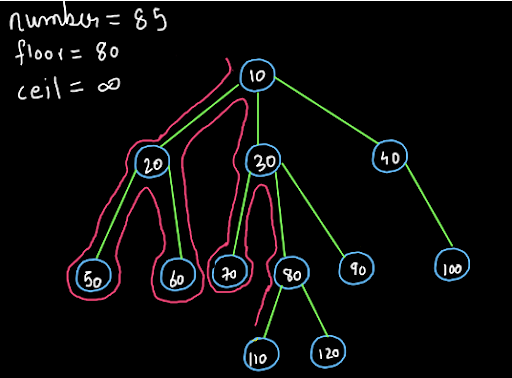
Now we move to the next element 20, it is also smaller than 85. Also, it is greater than the previous floor value i.e. 10. So, we will update the floor value to 20 and ceil will remain infinity and we move forward in the Euler Path.

****

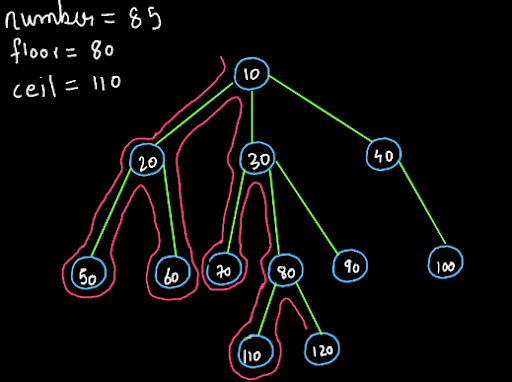
Similarly, we can keep on moving forward and updating the value floor until an element greater than 85 appears. We request you trace the Euler path further till node (70). You will have a floor value of 70 till here.

****

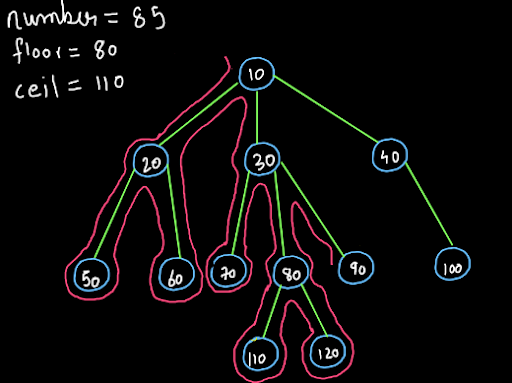
Now we move forward in the Euler path and we get a value 80. This value is still smaller than the input value 85 and greater than the previous floor value. So, the value of the floor will get updated and it becomes 80.

****

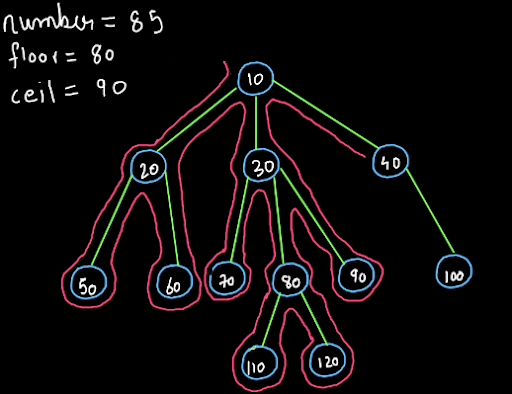
Now, we reach node (110). This value is greater than 85 so the floor will not be updated. However, the value stored in ceil (i.e. infinity) is greater than this value. So, ceil will get updated and we get ceil=110.

****

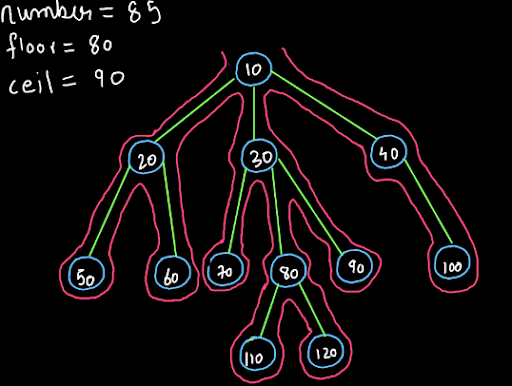
Now, we will move to the next node i.e. node (120). This value is greater than 85 so the floor won't be updated. Also, this value is greater than the current value of ceil. So, ceil will also not be updated and we will move to the next node.

****

Now we reach the node (90). This value is greater than 85 so the floor will not be updated. But, this value is greater than 85 and also less than the previous value of ceil. So, the ceil will be updated.

****

Dear reader, we request you to trace the remainder of the Euler path yourself. The values currently stored in the ceil and floor are the final ceil and floor values and the diagram will look like this:

****

So my friend, I hope that you got the entire procedure. If you still have any doubts regarding the procedure, you may refer to the solution video to clear all your doubts and understand the procedure as well as the code.

Recursion:

We will not discuss recursion in this question as we discussed in the previous question only that recursion is used only for traversing the nodes and we are not using it to solve the complete problem. You may refer to the solution video to understand the code given below and the role of recursion in it.

**3. Code Implementation :**

ConsoleCpp

#include<bits/stdc++.h>

#include<iostream>

using namespace std;

struct Node {

int data;

vector<Node\*>children;

};

Node\* newNode(int key) {

Node\* temp = new Node;

temp->data = key;

return temp;

}

Node\* construct(int arr[], int n ) {

Node\* root = NULL;

stack<Node\*>st;

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

if (arr[i] == -1) {

st.pop();

} else {

Node\* t = newNode(arr[i]);

if (st.size() > 0) {

st.top()->children.push\_back(t);

} else {

root = t;

}

st.push(t);

}

}

return root;

}

int c = INT\_MAX;

int flloor = INT\_MIN;

void cnf(Node\* node, int data) {

if (node->data > data)

{

if (node->data < c)

{

c = node->data;

}

}

else if (node->data < data)

{

if (node->data > flloor)

{

flloor = node->data;

}

}

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

{

cnf(child, data);

}

}

int main() {

int n;

cin >> n;

int arr[n];

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

cin >> arr[i];

}

int data;

cin >> data;

Node\* root = construct(arr, n);

cnf(root, data);

cout << "CEIL = " << c << endl;

cout << "FLOOR = " << flloor << endl;

}

We hope that you got the code and the role of recursion in this solution too. Now, let's analyze the time and space complexity:

**4. Time and Space Complexity Analysis:**

Time Complexity

The time complexity of this solution is O(n) as we are traversing all the nodes of the tree.

Space Complexity

The space complexity of this solution is O(1). Again like almost every previous question, if we consider the recursion space the time complexity becomes O(logn) as the maximum height of the stack can be equal to the height of the tree i.e. O(logn). So dear reader, we hope that you have understood the entire solution. If you still have any doubts about it, you may refer to the complete solution video to clear all your doubts. With this we have completed this question.