1516. Move Sub-Tree of N-Ary Tree

Problem Explanation

This problem involves dealing with an N-ary tree where each node can have N children. We have the root of the tree and two nodes, `p` and `q`. The task is to move the subtree of node `p`, under node `q`. Node `p` should become the last child of node `q`.

There are three scenarios to consider:

1. Node `q` is in the subtree of node `p`.
2. Node `p` is in the subtree of node `q`.
3. Neither node `p` nor `q` lie in each other's subtree.

For the second and third cases, the problem is straightforward - move `p` (with its subtree) to be a child of `q`. But in the first case, doing so may disconnect the tree. Therefore, we must reconnect the tree again. In this case, we first move `p` with all of its subtree except `q` and add it as a child to `q`. Then we see if the tree is disconnected and reconnect it accordingly.

Walkthrough

Let's take an example where our input tree is `[1,null,2,3,null,4,5,null,6,null,7,8]`, `p` equals `2` and `q` equals `7`. This example falls under the first case where `q` is in the subtree of `p`.

First step: We move `p` (with all of its subtree except `q`) and add it as a child to `q`. This gives us a tree `[1,null,7,3,null,2,null,6,null,4,5,null,null,8]`.

Then we notice that the tree is disconnected so we need to reconnect `q` to replace `p`.

Approach

1. The first step in the solution is checking if `p` already is a child of `q`. In this case, we don't have to do anything.
2. We add a dummy node to handle the case where `p` is the root of the tree.
3. We get the parent of `p` and remove it from its children list.
4. Add `p` as a child to `q`.
5. We check if `q` is in the subtree of `p`. If it is, we have to also update `q`'s parent to replace `q` with `p`.

Python Solution

```
python
class Solution:
    def moveSubTree(self, root, p, q):
        if p in q.children:
            return root

        dummy = Node(0, [root])

        pParent = self.getParent(dummy, p)
        pParent.children.remove(p)

        q.children.append(p)

        qParent = self.getParent(p, q)
        if qParent:
            qParent.children.remove(q)
            pParent.children.append(q)

        return dummy.children[0]

    def getParent(self, root, node):
        stack = [(root, None)]
        while stack:
            curr, parent = stack.pop()
            if curr == node:
                return parent
            for child in curr.children:
                stack.append((child, curr))
        return None
```

Java Solution

```
java
public class Solution {
    public Node moveSubTree(Node root, Node p, Node q) {
        if (q.children.contains(p)) {
            return root;
        }

        Node dummy = new Node(0, Arrays.asList(root));

        Node pParent = getParent(dummy, p);
        pParent.children.remove(p);

        q.children.add(p);

        Node qParent = getParent(p, q);
        if (qParent != null) {
            qParent.children.remove(q);
            pParent.children.add(q);
        }

        return dummy.children.get(0);
    }
    private Node getParent(Node root, Node target) {
        Deque<Pair<Node, Node>> stack = new ArrayDeque<>();
        stack.push(new Pair(root, null));
        while (!stack.isEmpty()) {
            Pair<Node, Node> curr = stack.pop();
            if (curr.getKey() == target) {
                return curr.getValue();
            }
            for (Node child: curr.getKey().children) {
                stack.push(new Pair(child, curr.getKey()));
            }
        }
        return null;
    }
}
```

C# Solution

```
csharp
public class Solution {
    public Node MoveSubTree(Node root, Node p, Node q) {
        if (q.children.Contains(p)) {
            return root;
        }

        Node dummy = new Node(0, new List<Node> {root});

        Node pParent = GetParent(dummy, p);
        pParent.children.Remove(p);

        q.children.Add(p);

        Node qParent = GetParent(p, q);
        if (qParent != null) {
            qParent.children.Remove(q);
            pParent.children.Add(q);
        }

        return dummy.children[0];
    }

    private Node GetParent(Node root, Node target) {
        Stack<Tuple<Node, Node>> stack = new Stack<Tuple<Node, Node>>();
        stack.Push(new Tuple<Node, Node>(root, null));
        while (stack.Count > 0) {
            Tuple<Node, Node> curr = stack.Pop();
            if (curr.Item1 == target) {
                return curr.Item2;
            }
            foreach (Node child in curr.Item1.children) {
                stack.Push(new Tuple<Node, Node>(child, curr.Item1));
            }
        }
        return null;
    }
}
```

JavaScript Solution

```
javascript
class Solution {
    moveSubTree(root, p, q){
        if(q.children.includes(p)){
            return root;
        }
        let dummy = {val: 0, children: [root]};
        let pParent = this.getParent(dummy, p);
        pParent.children = pParent.children.filter(child => child !== p);
        q.children.push(p);
        let qParent = this.getParent(p, q);
        if(qParent){
            qParent.children = qParent.children.filter(child => child !== q);
            pParent.children.push(q);
        }
        return dummy.children[0];
    }
    getParent(root, target){
        let stack = [[root, null]];
        while(stack.length){
            let [node, parent] = stack.pop();
            if(node === target){
                return parent;
            }
            for(let child of node.children){
                stack.push([child, node]);
            }
        }
        return null;
    }
}
```

C++ Solution

```
cpp
class Solution{
public:
    Node* moveSubTree(Node* root, Node* p, Node* q){
        if(find(q->children.begin(), q->children.end(), p) != q->children.end()){
            return root;
        }
        Node* dummy = new Node(0, {root});
        Node* pParent = getParent(dummy, p);
        pParent->children.erase(remove(pParent->children.begin(), pParent->children.end(), p), pParent->children.end(
        q->children.push_back(p);
        Node* qParent = getParent(p, q);
        if(qParent){
            qParent->children.erase(remove(qParent->children.begin(), qParent->children.end(), q), qParent->children.
            pParent->children.push_back(q);
        }
        return dummy->children[0];
    }
private:
    Node* getParent(Node* root, Node* target){
        stack<pair<Node*, Node*>> stk;
        stk.push({root, nullptr});
        while(!stk.empty()){
            auto [node, parent] = stk.top();
            stk.pop();
            if(node == target){
                return parent;
            }
            for(auto& child : node->children){
                stk.push({child, node});
            }
        }
        return nullptr;
    }
};
```

In this C++ solution, we start by checking if `p` is already a child of `q`. If it is, we simply return the root as no moving is needed.

We then create a dummy node and add `root` to it. We find the parent of `p` and remove `p` from its children. We push `p` into `q`'s children.

Finally, we find if `q` is present in the subtree of `p`. If it is, we find the parent of `q` and remove `q` from its children and add `q` to `pParent's` children.

This way, we have moved the subtree rooted at `p` to become the last child of `q`.

The method `getParent` finds the parent of the target node by DFS (Depth-First Search) using a stack, which stores pairs of current nodes and their parents. If it finds the target node in the stack, it pops and returns the parent. If it doesn't find the target, it returns `nullptr`.

It returns the new `root`.