Problem 1516: Move Sub-Tree of N-Ary Tree

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given the

root

of an

N-ary tree

of unique values, and two nodes of the tree

p

and

q

.

You should move the subtree of the node

p

to become a direct child of node

q

. If

p

is already a direct child of

q

, do not change anything. Node

p

must be

the last child in the children list of node

q

.

Return

the root of the tree

after adjusting it.

There are 3 cases for nodes

p

and

q

:

Node

q

is in the sub-tree of node

p

.

Node

p

is in the sub-tree of node

q

.

Neither node

p

is in the sub-tree of node

q

nor node

q

is in the sub-tree of node

p

.

In cases 2 and 3, you just need to move

p

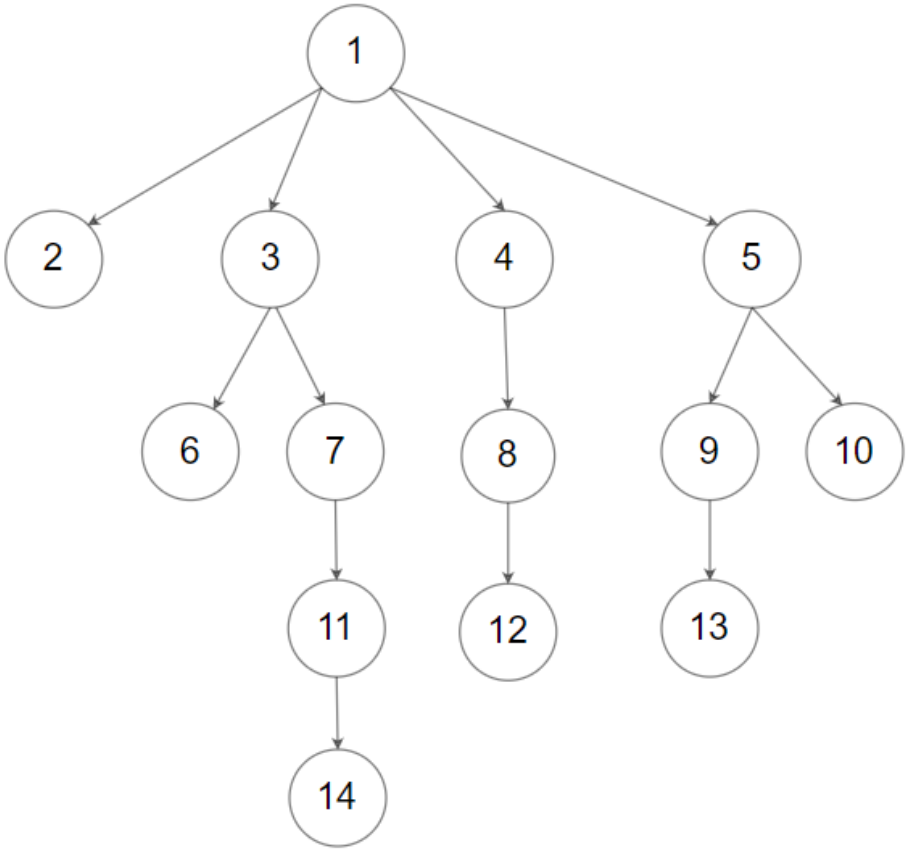
(with its sub-tree) to be a child of

q

, but in case 1 the tree may be disconnected, thus you need to reconnect the tree again.

Please read the examples carefully before solving this problem.

Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

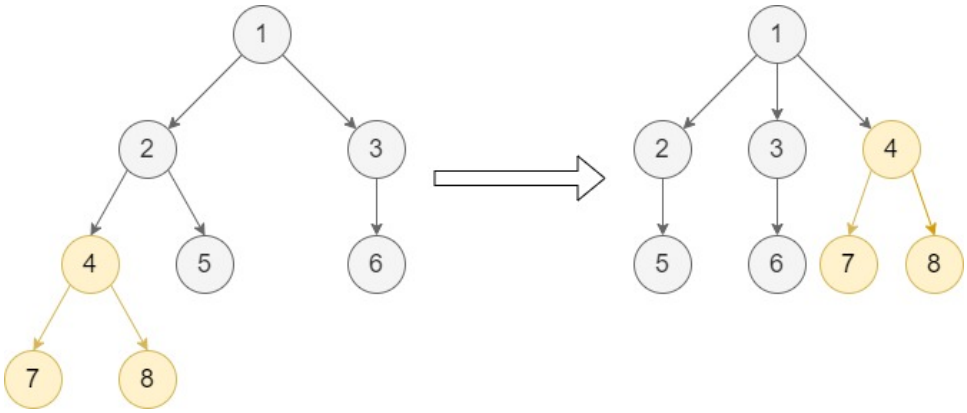


For example, the above tree is serialized as

[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

.

Example 1:



Input:

root = [1,null,2,3,null,4,5,null,6,null,7,8], p = 4, q = 1

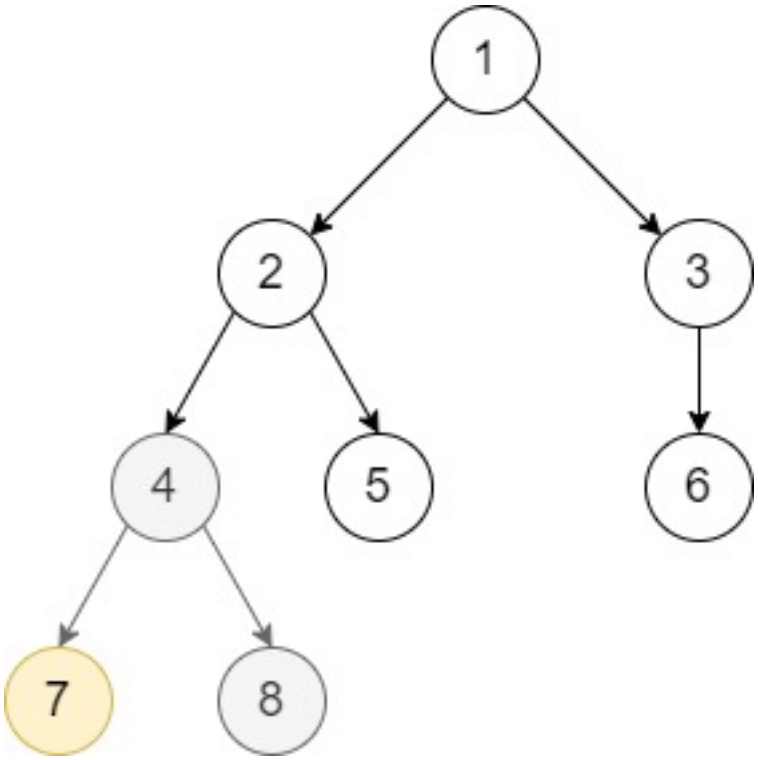
Output:

[1,null,2,3,4,null,5,null,6,null,7,8]

Explanation:

This example follows the second case as node p is in the sub-tree of node q. We move node p with its sub-tree to be a direct child of node q. Notice that node 4 is the last child of node 1.

Example 2:



Input:

root = [1,null,2,3,null,4,5,null,6,null,7,8], p = 7, q = 4

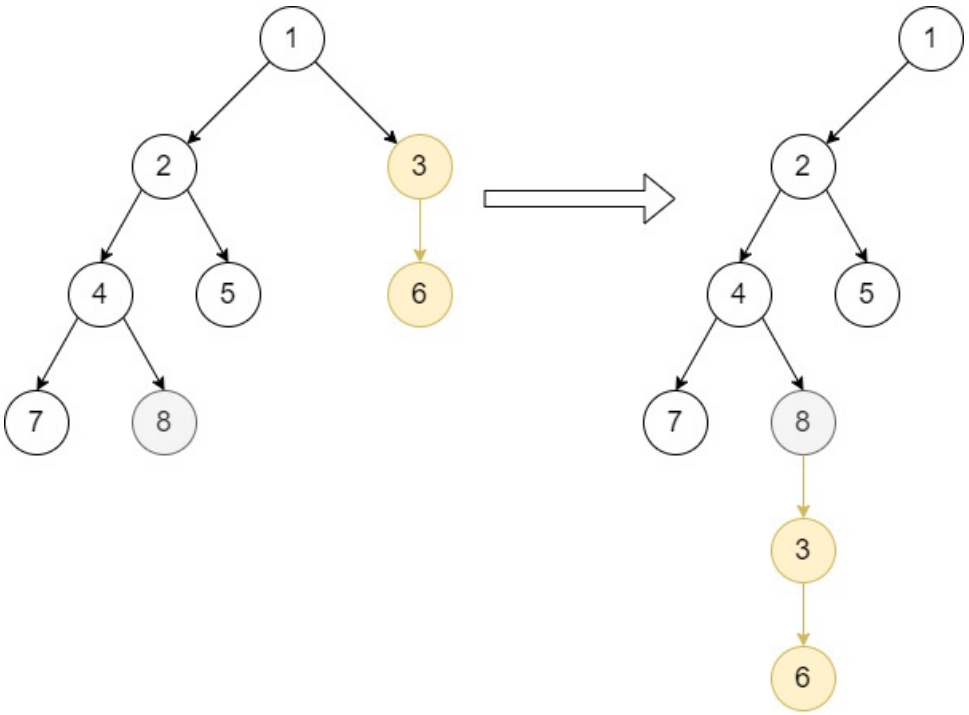
Output:

[1,null,2,3,null,4,5,null,6,null,7,8]

Explanation:

Node 7 is already a direct child of node 4. We don't change anything.

Example 3:



Input:

root = [1,null,2,3,null,4,5,null,6,null,7,8], p = 3, q = 8

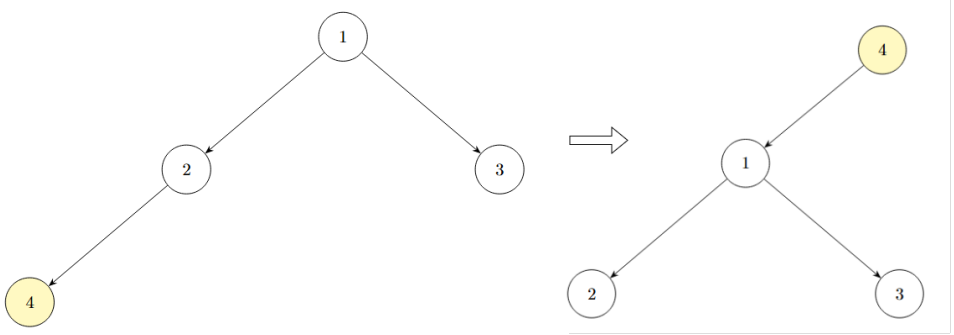
Output:

[1,null,2,null,4,5,null,7,8,null,null,null,3,null,6]

Explanation:

This example follows case 3 because node p is not in the sub-tree of node q and vice-versa. We can move node 3 with its sub-tree and make it as node 8's child.

Example 4:



Input:

root = [1,null,2,3,null,4], p = 1, q = 4

Output:

[4,null,1,null,2,3]

Explanation:

This example follows case 1 because node q is in the sub-tree of node p. Disconnect 4 with its parent and move node 1 with its sub-tree and make it as node 4's child.

Constraints:

The total number of nodes is between

[2, 1000]

.

Each node has a

unique

value.

p != null

q != null

p

and

q

are two different nodes (i.e.

p != q

).

Code Snippets

C++:

```
/*
// Definition for a Node.
class Node {
public:
int val;
vector<Node*> children;

Node() {}

Node(int _val) {
val = _val;
}

Node(int _val, vector<Node*> _children) {
val = _val;
children = _children;
}
};
*/

class Solution {
public:
Node* moveSubTree(Node* root, Node* p, Node* q) {

}
};
```

Java:

```
/*
// Definition for a Node.
class Node {
public int val;
```

```
public List<Node> children;


public Node() {
children = new ArrayList<Node>();
}

public Node(int _val) {
val = _val;
children = new ArrayList<Node>();
}

public Node(int _val,ArrayList<Node> _children) {
val = _val;
children = _children;
}
};
*/

class Solution {
public Node moveSubTree(Node root, Node p, Node q) {

}
}
```

Python3:

```
"""
# Definition for a Node.
class Node:
def __init__(self, val: Optional[int] = None, children:
Optional[List['Node']] = None):
self.val = val
self.children = children if children is not None else []
"""

class Solution:
def moveSubTree(self, root: 'Node', p: 'Node', q: 'Node') -> 'Node':
```

Python:

```
"""
# Definition for a Node.
class Node(object):
def __init__(self, val=None, children=None):
self.val = val
self.children = children if children is not None else []
"""

class Solution(object):
def moveSubTree(self, root, p, q):
"""
:type root: Node
:type p: Node
:type q: Node
:rtype: Node
"""
```

JavaScript:

```
/**
* // Definition for a Node.
* function Node(val, children) {
* this.val = val === undefined ? 0 : val;
* this.children = children === undefined ? [] : children;
* };
*/

/**
* @param {Node} root
* @param {Node} p
* @param {Node} q
* @return {Node}
*/
var moveSubTree = function(root, p, q) {

};
```

TypeScript:

```
/**
* Definition for _Node.
* class _Node {
* val: number
```

```
* children: _Node[]
*
* constructor(val?: number, children?: _Node[]) {
* this.val = (val===undefined ? 0 : val)
* this.children = (children===undefined ? [] : children)
* }
* }
*/

function moveSubTree(root: _Node | null, p: _Node | null, q: _Node | null):
_Node | null {

};
```

C#:

```
/*
// Definition for a Node.
public class Node {
public int val;
public IList<Node> children;

public Node() {
val = 0;
children = new List<Node>();
}

public Node(int _val) {
val = _val;
children = new List<Node>();
}

public Node(int _val, List<Node> _children) {
val = _val;
children = _children;
}
}
*/

public class Solution {
public Node MoveSubTree(Node root, Node p, Node q) {
```

```
}
}
```

Go:

```
/**
* Definition for a Node.
* type Node struct {
* Val int
* Children []*Node
* }
*/

func moveSubTree(root *Node, p *Node, q *Node) *Node {

}
```

Kotlin:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var children: List<Node?> = listOf()
* }
*/

class Solution {
fun moveSubTree(root: Node?, p: Node?, q: Node?): Node? {

}
}
```

Swift:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var children: [Node]
* public init(_ val: Int) {
* self.val = val
* self.children = []
```

```
* }
* }
*/

class Solution {
func moveSubTree(_ root: Node?, _ p: Node?, _ q: Node?) -> Node? {

}
}
```

Ruby:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :children
# def initialize(val=0, children=[])
# @val = val
# @children = children
# end
# end

# @param {TreeNode} root
# @param {TreeNode} p
# @param {TreeNode} q
# @return {Integer}
def move_sub_tree(root, p, q)

end
```

PHP:

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $children = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->children = array();
* }
* }
*/
```

```
class Solution {

/**
* @param Node $root
* @param Node $p
* @param Node $q
* @return Node
*/
function moveSubTree($root, $p, $q) {

}
}
```

Scala:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var children: List[Node] = List()
* }
*/

object Solution {
def moveSubTree(root: Node, p: Node, q: Node): Node = {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Move Sub-Tree of N-Ary Tree
* Difficulty: Hard
* Tags: tree, search
*
* Approach: DFS or BFS traversal
```

```
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
class Node {
public:
int val;
vector<Node*> children;

Node() {}

Node(int _val) {
val = _val;
}

Node(int _val, vector<Node*> _children) {
val = _val;
children = _children;
}
};
*/

class Solution {
public:
Node* moveSubTree(Node* root, Node* p, Node* q) {

}
};
```

Java Solution:

```
/**
* Problem: Move Sub-Tree of N-Ary Tree
* Difficulty: Hard
* Tags: tree, search
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
```

```
*/

/*
// Definition for a Node.
class Node {
public int val;
public List<Node> children;


public Node() {
children = new ArrayList<Node>();
}

public Node(int _val) {
val = _val;
children = new ArrayList<Node>();
}

public Node(int _val,ArrayList<Node> _children) {
val = _val;
children = _children;
}
};
*/

class Solution {
public Node moveSubTree(Node root, Node p, Node q) {

}
}
```

Python3 Solution:

```
"""
Problem: Move Sub-Tree of N-Ary Tree
Difficulty: Hard
Tags: tree, search

Approach: DFS or BFS traversal
Time Complexity: O(n) where n is number of nodes
Space Complexity: O(h) for recursion stack where h is height
```

```
"""

"""
# Definition for a Node.
class Node:
def __init__(self, val: Optional[int] = None, children:
Optional[List['Node']] = None):
self.val = val
self.children = children if children is not None else []
"""

class Solution:
def moveSubTree(self, root: 'Node', p: 'Node', q: 'Node') -> 'Node':
# TODO: Implement optimized solution
pass
```

Python Solution:

```
"""
# Definition for a Node.
class Node(object):
def __init__(self, val=None, children=None):
self.val = val
self.children = children if children is not None else []
"""

class Solution(object):
def moveSubTree(self, root, p, q):
"""
:type root: Node
:type p: Node
:type q: Node
:rtype: Node
"""
```

JavaScript Solution:

```
/**
* Problem: Move Sub-Tree of N-Ary Tree
* Difficulty: Hard
* Tags: tree, search
```

```
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* // Definition for a Node.
* function Node(val, children) {
* this.val = val === undefined ? 0 : val;
* this.children = children === undefined ? [] : children;
* };
*/

/**
* @param {Node} root
* @param {Node} p
* @param {Node} q
* @return {Node}
*/
var moveSubTree = function(root, p, q) {

};
```

TypeScript Solution:

```
/**
* Problem: Move Sub-Tree of N-Ary Tree
* Difficulty: Hard
* Tags: tree, search
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for _Node.
* class _Node {
* val: number
* children: _Node[]
```

```
*
* constructor(val?: number, children?: _Node[]) {
* this.val = (val===undefined ? 0 : val)
* this.children = (children===undefined ? [] : children)
* }
* }
*/

function moveSubTree(root: _Node | null, p: _Node | null, q: _Node | null):
_Node | null {

};
```

C# Solution:

```
/*
* Problem: Move Sub-Tree of N-Ary Tree
* Difficulty: Hard
* Tags: tree, search
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
public class Node {
public int val;
public IList<Node> children;

public Node() {
val = 0;
children = new List<Node>();
}

public Node(int _val) {
val = _val;
children = new List<Node>();
}
```

```
public Node(int _val, List<Node> _children) {
val = _val;
children = _children;
}
}
*/

public class Solution {
public Node MoveSubTree(Node root, Node p, Node q) {

}
}
```

Go Solution:

```
// Problem: Move Sub-Tree of N-Ary Tree
// Difficulty: Hard
// Tags: tree, search
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

/**
* Definition for a Node.
* type Node struct {
* Val int
* Children []*Node
* }
*/

func moveSubTree(root *Node, p *Node, q *Node) *Node {

}
```

Kotlin Solution:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var children: List<Node?> = listOf()
```

```
* }
*/

class Solution {
fun moveSubTree(root: Node?, p: Node?, q: Node?): Node? {

}
}
```

Swift Solution:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var children: [Node]
* public init(_ val: Int) {
* self.val = val
* self.children = []
* }
* }
*/

class Solution {
func moveSubTree(_ root: Node?, _ p: Node?, _ q: Node?) -> Node? {

}
}
```

Ruby Solution:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :children
# def initialize(val=0, children=[])
# @val = val
# @children = children
# end
# end

# @param {TreeNode} root
```

```
# @param {TreeNode} p
# @param {TreeNode} q
# @return {Integer}
def move_sub_tree(root, p, q)

end
```

PHP Solution:

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $children = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->children = array();
* }
* }
*/

class Solution {

/**
* @param Node $root
* @param Node $p
* @param Node $q
* @return Node
*/
function moveSubTree($root, $p, $q) {

}
}
```

Scala Solution:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var children: List[Node] = List()
```

```
* }
*/

object Solution {
def moveSubTree(root: Node, p: Node, q: Node): Node = {

}
}
```