

Problem 590: N-ary Tree Postorder Traversal

Problem Information

Difficulty: [Easy](#)

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

Given the

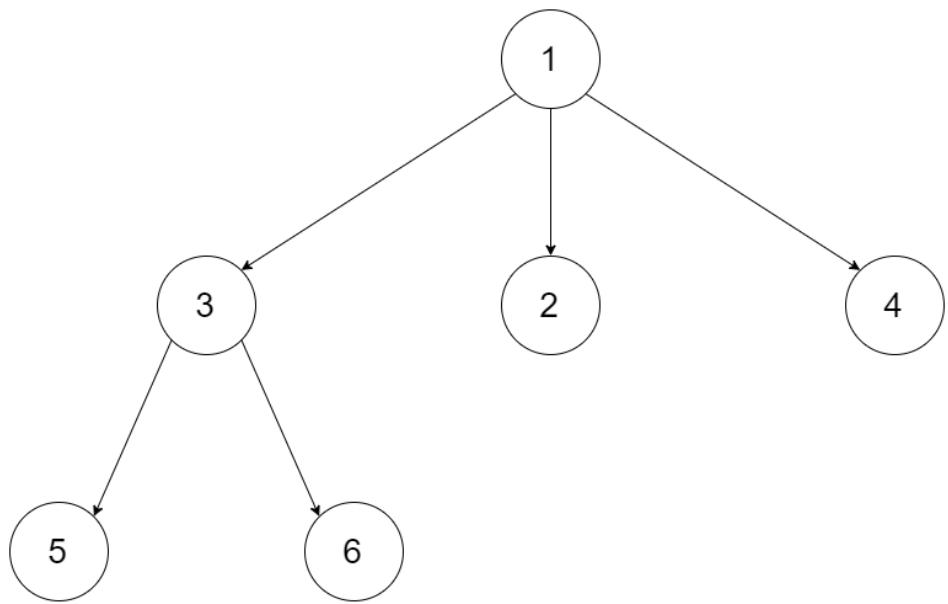
root

of an n-ary tree, return

the postorder traversal of its nodes' values

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

Example 1:



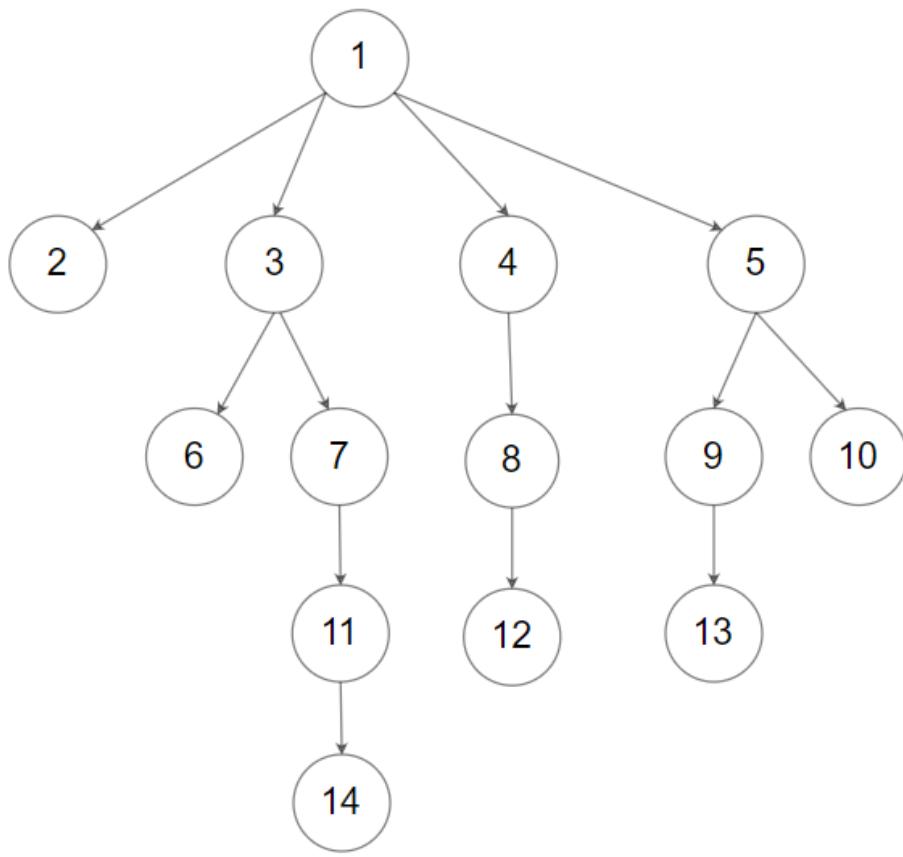
Input:

```
root = [1,null,3,2,4,null,5,6]
```

Output:

```
[5,6,3,2,4,1]
```

Example 2:



Input:

```
root = [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]
```

Output:

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

Constraints:

The number of nodes in the tree is in the range

```
[0, 10
```

4

]

$0 \leq \text{Node.val} \leq 10$

4

The height of the n-ary tree is less than or equal to

1000

.

Follow up:

Recursive solution is trivial, could you do it iteratively?

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:  
vector<int> postorder(Node* root) {  
  
}  
};
```

Java:

```
/*  
// Definition for a Node.  
class Node {  
public int val;  
public List<Node> children;  
  
public Node() {}  
  
public Node(int _val) {  
val = _val;  
}  
  
public Node(int _val, List<Node> _children) {  
val = _val;  
children = _children;  
}  
}  
*/  
  
class Solution {  
public List<Integer> postorder(Node root) {  
  
}  
}
```

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
```

```
"""
class Solution:
    def postorder(self, root: 'Node') -> List[int]:
```

Python:

```
"""
# Definition for a Node.
class Node(object):
    def __init__(self, val: Optional[int] = None, children: Optional[List['Node']] = None):
        self.val = val
        self.children = children
"""

class Solution(object):
    def postorder(self, root):
        """
:type root: Node
:rtype: List[int]
"""

"""

class Solution:
    def postorder(self, root: 'Node') -> List[int]:
```

JavaScript:

```
/**
 * // Definition for a _Node.
 * function _Node(val,children) {
 *     this.val = val;
 *     this.children = children;
 * };
 */

/**
 * @param {_Node|null} root
 * @return {number[]}
 */
var postorder = function(root) {

};
```

TypeScript:

```

/**
 * Definition for node.
 * class _Node {
 * val: number
 * children: _Node[]
 * constructor(val?: number) {
 * this.val = (val==undefined ? 0 : val)
 * this.children = []
 * }
 * }
 */
function postorder(root: _Node | null): number[] {
}

```

C#:

```

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

public Node() {}

public Node(int _val) {
val = _val;
}

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

public class Solution {
public IList<int> Postorder(Node root) {

}
}

```

C:

```
/**  
 * Definition for a Node.  
 * struct Node {  
 *     int val;  
 *     int numChildren;  
 *     struct Node** children;  
 * };  
 */  
  
/**  
 * Note: The returned array must be malloced, assume caller calls free().  
 */  
int* postorder(struct Node* root, int* returnSize) {  
  
}
```

Go:

```
/**  
 * Definition for a Node.  
 * type Node struct {  
 *     Val int  
 *     Children []*Node  
 * }  
 */  
  
func postorder(root *Node) []int {  
  
}
```

Kotlin:

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

```
}
```

```
}
```

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 postorder(_ root: Node?) -> [Int] {  
  
    }  
}
```

Ruby:

```
# Definition for a Node.  
# class Node  
# attr_accessor :val, :children  
# def initialize(val)  
#     @val = val  
#     @children = []  
# end  
# end  
  
# @param {Node} root  
# @return {List[int]}
```

```
def postorder(root)
```

```
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 $root
 * @return integer[]
 */
function postorder($root) {

}
}

```

Scala:

```

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

object Solution {
def postorder(root: Node): List[Int] = {

}
}

```

Solutions

C++ Solution:

```

/*
 * Problem: N-ary Tree Postorder Traversal
 * Difficulty: Easy
 * Tags: tree, search, stack
 *
 * 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:
    vector<int> postorder(Node* root) {

    }
};

```

Java Solution:

```

/**
 * Problem: N-ary Tree Postorder Traversal
 * Difficulty: Easy

```

```

* Tags: tree, search, stack
*
* 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() {}

public Node(int _val) {
val = _val;
}

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

}
*/
class Solution {
public List<Integer> postorder(Node root) {

}
}

```

Python3 Solution:

```

"""
Problem: N-ary Tree Postorder Traversal
Difficulty: Easy
Tags: tree, search, stack

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

"""

class Solution:

    def postorder(self, root: 'Node') -> List[int]:
        # TODO: Implement optimized solution
        pass

```

Python Solution:

```

"""

# Definition for a Node.
class Node(object):

    def __init__(self, val: Optional[int] = None, children: Optional[List['Node']] = None):
        self.val = val
        self.children = children

"""

class Solution(object):

    def postorder(self, root):

"""

:type root: Node
:rtype: List[int]
"""


```

JavaScript Solution:

```

/**
 * Problem: N-ary Tree Postorder Traversal
 * Difficulty: Easy
 * Tags: tree, search, stack

```

```

*
* 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;
* this.children = children;
* };
*/
/*
* @param {_Node|null} root
* @return {number[]}
*/
var postorder = function(root) {

};

```

TypeScript Solution:

```

/**
* Problem: N-ary Tree Postorder Traversal
* Difficulty: Easy
* Tags: tree, search, stack
*
* 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) {
* this.val = (val==undefined ? 0 : val)

```

```

        * this.children = []
        *
        *
        */
    }

    function postorder(root: _Node | null): number[] {
}

```

C# Solution:

```

/*
 * Problem: N-ary Tree Postorder Traversal
 * Difficulty: Easy
 * Tags: tree, search, stack
 *
 * 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() {}

public Node(int _val) {
val = _val;
}

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

}

public class Solution {

```

```
public IList<int> Postorder(Node root) {  
    }  
}
```

C Solution:

```
/*  
 * Problem: N-ary Tree Postorder Traversal  
 * Difficulty: Easy  
 * Tags: tree, search, stack  
 *  
 * 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.  
 * struct Node {  
 *     int val;  
 *     int numChildren;  
 *     struct Node** children;  
 * };  
 */  
  
/**  
 * Note: The returned array must be malloced, assume caller calls free().  
 */  
int* postorder(struct Node* root, int* returnSize) {  
  
}
```

Go Solution:

```
// Problem: N-ary Tree Postorder Traversal  
// Difficulty: Easy  
// Tags: tree, search, stack  
//  
// 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 postorder(root *Node) []int {
}

```

Kotlin Solution:

```

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

class Solution {
    fun postorder(root: Node?): List<Int> {
        return if (root == null) {
            emptyList()
        } else {
            root.children.map { postorder(it) }.flatten() + root.`val`
        }
    }
}

```

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 postorder(_ root: Node?) -> [Int] {

}
}

```

Ruby Solution:

```

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

# @param {Node} root
# @return {List[int]}
def postorder(root)

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 $root
* @return integer[]
*/
function postorder($root) {
}

}
```

Scala Solution:

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

object Solution {
def postorder(root: Node): List[Int] = {

}
}
```