Problem 431: Encode N-ary Tree to Binary Tree

Problem Information

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

Problem Description

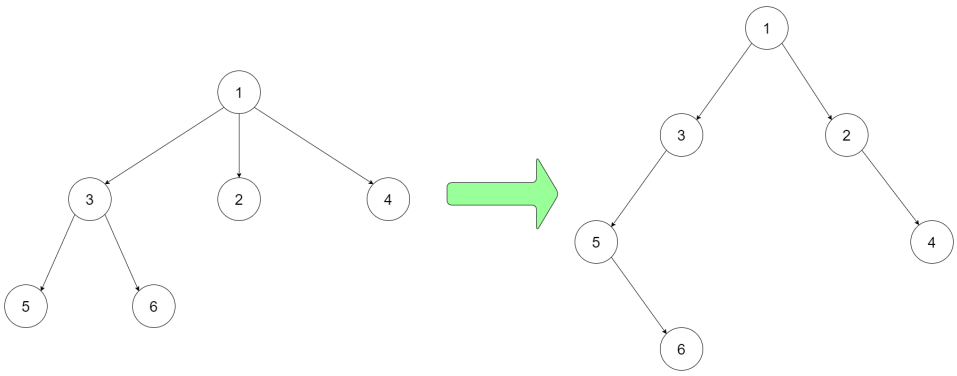
Design an algorithm to encode an N-ary tree into a binary tree and decode the binary tree to get the original N-ary tree. An N-ary tree is a rooted tree in which each node has no more than N children. Similarly, a binary tree is a rooted tree in which each node has no more than 2 children. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that an N-ary tree can be encoded to a binary tree and this binary tree can be decoded to the original N-nary tree structure.

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

For example, you may encode the following

3-ary

tree to a binary tree in this way:



Input:

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

Note that the above is just an example which

might or might not

work. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:

Input:

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

Output:

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

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:

[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 3:

Input:

root = []

Output:

[]

Constraints:

The number of nodes in the tree is in the range

[0, 10

4

]

.

0 <= Node.val <= 10

4

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

1000

Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

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;
}
};
*/

/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/

class Codec {
public:
// Encodes an n-ary tree to a binary tree.
TreeNode* encode(Node* root) {

}

// Decodes your binary tree to an n-ary tree.
Node* decode(TreeNode* root) {

}
};

// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.decode(codec.encode(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;
}
};
*/

/**
* Definition for a binary tree node.
* public class TreeNode {
* int val;
* TreeNode left;
* TreeNode right;
* TreeNode(int x) { val = x; }
* }
*/

class Codec {
// Encodes an n-ary tree to a binary tree.
public TreeNode encode(Node root) {

}

// Decodes your binary tree to an n-ary tree.
public Node decode(TreeNode root) {

}
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.decode(codec.encode(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
"""

"""
# Definition for a binary tree node.
class TreeNode:
def __init__(self, x):
self.val = x
self.left = None
self.right = None
"""

class Codec:
# Encodes an n-ary tree to a binary tree.
def encode(self, root: 'Optional[Node]') -> Optional[TreeNode]:


# Decodes your binary tree to an n-ary tree.
def decode(self, data: Optional[TreeNode]) -> 'Optional[Node]':


# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.decode(codec.encode(root))
```

Python:

```
"""
# Definition for a Node.
class Node(object):
def __init__(self, val=None, children=None):
self.val = val
self.children = children
"""

"""
# Definition for a binary tree node.
```

```
class TreeNode(object):
def __init__(self, x):
self.val = x
self.left = None
self.right = None
"""

class Codec:
def encode(self, root):
"""Encodes an n-ary tree to a binary tree.
:type root: Node
:rtype: TreeNode
"""


def decode(self, data):
"""Decodes your binary tree to an n-ary tree.
:type data: TreeNode
:rtype: Node
"""


# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.decode(codec.encode(root))
```

JavaScript:

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

/**
* Definition for a binary tree node.
* function TreeNode(val) {
* this.val = val;
* this.left = this.right = null;
* }
```

```
*/

class Codec {
constructor() {
}

/**
* @param {_Node|null} root
* @return {TreeNode|null}
*/
// Encodes an n-ary tree to a binary tree.
encode = function(root) {

};

/**
* @param {TreeNode|null} root
* @return {_Node|null}
*/
// Decodes your binary tree to an n-ary tree.
decode = function(root) {

};
}

/*
* Your Codec object will be instantiated and called as such:
* codec = Codec()
* codec.decode(codec.encode(root))
*/
```

TypeScript:

```
/**
* Definition for _Node.
* class _Node {
* val: number
* children: _Node[]
*
* constructor(v: number) {
* this.val = v;
* this.children = [];
```

```
* }
* }
*/


/**
* Definition for a binary tree node.
* class TreeNode {
* val: number
* left: TreeNode | null
* right: TreeNode | null
* constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
{
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
* }
*/

class Codec {
constructor() {

}

// Encodes a tree to a binary tree.
serialize(root: _Node | null): TreeNode | null {

};

// Decodes your encoded data to tree.
deserialize(root: TreeNode | null): _Node | null {

};
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));
```

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;
}
}
*/

/**
* Definition for a binary tree node.
* public class TreeNode {
* public int val;
* public TreeNode left;
* public TreeNode right;
* public TreeNode(int x) { val = x; }
* }
*/

public class Codec {
// Encodes an n-ary tree to a binary tree.
public TreeNode encode(Node root) {

}

// Decodes your binary tree to an n-ary tree.
public Node decode(TreeNode root) {

}
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
```

```
// codec.decode(codec.encode(root));
```

Go:

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

/**
* Definition for a binary tree node.
* type TreeNode struct {
* Val int
* Left *TreeNode
* Right *TreeNode
* }
*/

type Codec struct {

}

func Constructor() *Codec {

}

func (this *Codec) encode(root *Node) *TreeNode {

}

func (this *Codec) decode(root *TreeNode) *Node {

}

/**
* Your Codec object will be instantiated and called as such:
* obj := Constructor();
* bst := obj.encode(root);
* ans := obj.decode(bst);
```

```
*/
```

Kotlin:

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

/**
* Example:
* var ti = TreeNode(5)
* var v = ti.`val`
* Definition for a binary tree node.
* class TreeNode(var `val`: Int) {
* var left: TreeNode? = null
* var right: TreeNode? = null
* }
*/

class Codec {
// Encodes a tree to a single string.
fun encode(root: Node?): TreeNode? {

}

// Decodes your encoded data to tree.
fun decode(root: TreeNode?): Node? {

}
}

/**
* Your Codec object will be instantiated and called as such:
* var obj = Codec()
* var data = obj.encode(root)
* var ans = obj.decode(data)
*/
```

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 = []
* }
* }
*/

/**
* Definition for a binary tree node.
* public class TreeNode {
* public var val: Int
* public var left: TreeNode?
* public var right: TreeNode?
* public init(_ val: Int) {
* self.val = val
* self.left = nil
* self.right = nil
* }
* }
*/

class Codec {
func encode(_ root: Node?) -> TreeNode? {

}

func decode(_ root: TreeNode?) -> Node? {

}
}

/**
* Your Codec object will be instantiated and called as such:
* let obj = Codec()
* let ret_1: TreeNode? = obj.encode(root)
* let ret_2: Node? = obj.decode(root)
```

```
*/
```

Ruby:

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

# Definition for a binary tree node.
# class TreeNode
# attr_accessor :val, :left, :right
# def initialize(val)
# @val = val
# @left, @right = nil, nil
# end
# end

class Codec
# Encodes an n-ary tree to a binary tree.
# @param {Node} root
# @return {TreeNode}
def encode(root)

end

# Decodes your binary tree to an n-ary tree.
# @param {TreeNode} root
# @return {Node}
def decode(root)

end
end

# Your Codec object will be instantiated and called as such:
# obj = Codec.new()
# data = obj.encode(root)
# ans = obj.decode(data)
```

PHP:

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

/**
* Definition for a binary tree node.
* class TreeNode {
* public $val = null;
* public $left = null;
* public $right = null;
* function __construct($val = 0, $left = null, $right = null) {
* $this->val = $val;
* $this->left = $left;
* $this->right = $right;
* }
* }
*/

class Codec {
/**
* @param Node $root
* @return TreeNode
*/
function encode($root) {

}

/**
* @param TreeNode $root
* @return Node
*/
function decode($root) {
```

```
}
}

/**
* Your Codec object will be instantiated and called as such:
* $obj = Codec();
* $ret_1 = $obj->encode($root);
* $ret_2 = $obj->decode($root);
*/
```

Scala:

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

/**
* Definition for a binary tree node.
* class TreeNode(var _value: Int) {
* var value: Int = _value
* var left: TreeNode = null
* var right: TreeNode = null
* }
*/

class Codec {
// Encodes an n-ary tree to a binary tree.
def encode(root: Node): TreeNode = {

}

// Decodes your binary tree to an n-ary tree.
def decode(root: TreeNode): Node = {

}
}
```

```
/**
* Your Codec object will be instantiated and called as such:
* var obj = new Codec()
* var data = obj.encode(root)
* var ans = obj.decode(data)
*/
```

Solutions

C++ Solution:

```
/*
* Problem: Encode N-ary Tree to Binary 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;
}
};
*/
```

```
/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/

class Codec {
public:
// Encodes an n-ary tree to a binary tree.
TreeNode* encode(Node* root) {

}

// Decodes your binary tree to an n-ary tree.
Node* decode(TreeNode* root) {

}
};

// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.decode(codec.encode(root));
```

Java Solution:

```
/**
* Problem: Encode N-ary Tree to Binary 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() {
// TODO: Implement optimized solution
return 0;
}

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

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

/**
* Definition for a binary tree node.
* public class TreeNode {
* int val;
* TreeNode left;
* TreeNode right;
* TreeNode(int x) { val = x; }
* }
*/

class Codec {
// Encodes an n-ary tree to a binary tree.
public TreeNode encode(Node root) {

}

// Decodes your binary tree to an n-ary tree.
public Node decode(TreeNode root) {

}
}
```

```
// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.decode(codec.encode(root));
```

Python3 Solution:

```
"""
Problem: Encode N-ary Tree to Binary 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
"""

"""
# Definition for a binary tree node.
class TreeNode:
def __init__(self, x):
self.val = x
self.left = None
self.right = None
"""

class Codec:
# Encodes an n-ary tree to a binary tree.
def encode(self, root: 'Optional[Node]') -> Optional[TreeNode]:
# 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
"""

"""
# Definition for a binary tree node.
class TreeNode(object):
def __init__(self, x):
self.val = x
self.left = None
self.right = None
"""

class Codec:
def encode(self, root):
"""Encodes an n-ary tree to a binary tree.
:type root: Node
:rtype: TreeNode
"""



def decode(self, data):
"""Decodes your binary tree to an n-ary tree.
:type data: TreeNode
:rtype: Node
"""



# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.decode(codec.encode(root))
```

JavaScript Solution:

```
/**
* Problem: Encode N-ary Tree to Binary 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;
* this.children = children;
* };
*/

/**
* Definition for a binary tree node.
* function TreeNode(val) {
* this.val = val;
* this.left = this.right = null;
* }
*/

class Codec {
constructor() {
}

/**
* @param {_Node|null} root
* @return {TreeNode|null}
*/
// Encodes an n-ary tree to a binary tree.
encode = function(root) {

};

/**
* @param {TreeNode|null} root
* @return {_Node|null}
*/
// Decodes your binary tree to an n-ary tree.
decode = function(root) {
```

```
};
}

/*
* Your Codec object will be instantiated and called as such:
* codec = Codec()
* codec.decode(codec.encode(root))
*/
```

TypeScript Solution:

```
/**
* Problem: Encode N-ary Tree to Binary 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(v: number) {
* this.val = v;
* this.children = [];
* }
* }
*/



/**
* Definition for a binary tree node.
* class TreeNode {
* val: number
* left: TreeNode | null
```

```
* right: TreeNode | null
* constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
{
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
* }
*/

class Codec {
constructor() {

}

// Encodes a tree to a binary tree.
serialize(root: _Node | null): TreeNode | null {

};

// Decodes your encoded data to tree.
deserialize(root: TreeNode | null): _Node | null {

};
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));
```

C# Solution:

```
/*
* Problem: Encode N-ary Tree to Binary 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() {}

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

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

/**
* Definition for a binary tree node.
* public class TreeNode {
* public int val;
* public TreeNode left;
* public TreeNode right;
* public TreeNode(int x) { val = x; }
* }
*/

public class Codec {
// Encodes an n-ary tree to a binary tree.
public TreeNode encode(Node root) {

}

// Decodes your binary tree to an n-ary tree.
public Node decode(TreeNode root) {

}
}
```

```
// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.decode(codec.encode(root));
```

Go Solution:

```
// Problem: Encode N-ary Tree to Binary 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
* }
*/

/**
* Definition for a binary tree node.
* type TreeNode struct {
* Val int
* Left *TreeNode
* Right *TreeNode
* }
*/

type Codec struct {

}

func Constructor() *Codec {

}

func (this *Codec) encode(root *Node) *TreeNode {
```

```
}

func (this *Codec) decode(root *TreeNode) *Node {

}

/**
* Your Codec object will be instantiated and called as such:
* obj := Constructor();
* bst := obj.encode(root);
* ans := obj.decode(bst);
*/
```

Kotlin Solution:

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

/**
* Example:
* var ti = TreeNode(5)
* var v = ti.`val`
* Definition for a binary tree node.
* class TreeNode(var `val`: Int) {
* var left: TreeNode? = null
* var right: TreeNode? = null
* }
*/

class Codec {
// Encodes a tree to a single string.
fun encode(root: Node?): TreeNode? {

}

// Decodes your encoded data to tree.
fun decode(root: TreeNode?): Node? {
```

```
}
}

/**
* Your Codec object will be instantiated and called as such:
* var obj = Codec()
* var data = obj.encode(root)
* var ans = obj.decode(data)
*/
```

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 = []
* }
* }
*/

/**
* Definition for a binary tree node.
* public class TreeNode {
* public var val: Int
* public var left: TreeNode?
* public var right: TreeNode?
* public init(_ val: Int) {
* self.val = val
* self.left = nil
* self.right = nil
* }
* }
*/

class Codec {
func encode(_ root: Node?) -> TreeNode? {
```

```
}

func decode(_ root: TreeNode?) -> Node? {

}
}

/**
* Your Codec object will be instantiated and called as such:
* let obj = Codec()
* let ret_1: TreeNode? = obj.encode(root)
* let ret_2: Node? = obj.decode(root)
*/
```

Ruby Solution:

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

# Definition for a binary tree node.
# class TreeNode
# attr_accessor :val, :left, :right
# def initialize(val)
# @val = val
# @left, @right = nil, nil
# end
# end

class Codec
# Encodes an n-ary tree to a binary tree.
# @param {Node} root
# @return {TreeNode}
def encode(root)
```

```
end

# Decodes your binary tree to an n-ary tree.
# @param {TreeNode} root
# @return {Node}
def decode(root)

end
end

# Your Codec object will be instantiated and called as such:
# obj = Codec.new()
# data = obj.encode(root)
# ans = obj.decode(data)
```

PHP Solution:

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

/**
* Definition for a binary tree node.
* class TreeNode {
* public $val = null;
* public $left = null;
* public $right = null;
* function __construct($val = 0, $left = null, $right = null) {
* $this->val = $val;
* $this->left = $left;
* $this->right = $right;
* }
* }
```

```
*/

class Codec {
/**
* @param Node $root
* @return TreeNode
*/
function encode($root) {

}

/**
* @param TreeNode $root
* @return Node
*/
function decode($root) {

}
}

/**
* Your Codec object will be instantiated and called as such:
* $obj = Codec();
* $ret_1 = $obj->encode($root);
* $ret_2 = $obj->decode($root);
*/
```

Scala Solution:

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

/**
* Definition for a binary tree node.
* class TreeNode(var _value: Int) {
* var value: Int = _value
```

```
* var left: TreeNode = null
* var right: TreeNode = null
* }
*/

class Codec {
// Encodes an n-ary tree to a binary tree.
def encode(root: Node): TreeNode = {

}

// Decodes your binary tree to an n-ary tree.
def decode(root: TreeNode): Node = {

}
}

/**
* Your Codec object will be instantiated and called as such:
* var obj = new Codec()
* var data = obj.encode(root)
* var ans = obj.decode(data)
*/
```