# Problem 428: Serialize and Deserialize N-ary Tree

## Problem Information

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

## Problem Description

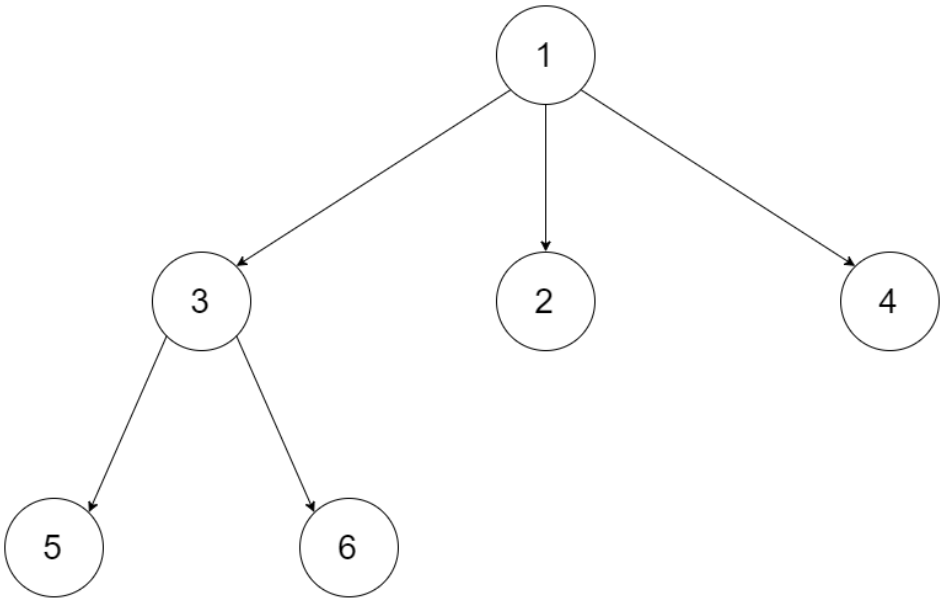
Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize an N-ary tree. An N-ary tree is a rooted tree in which each node has no more than N children. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that an N-ary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following

3-ary

tree

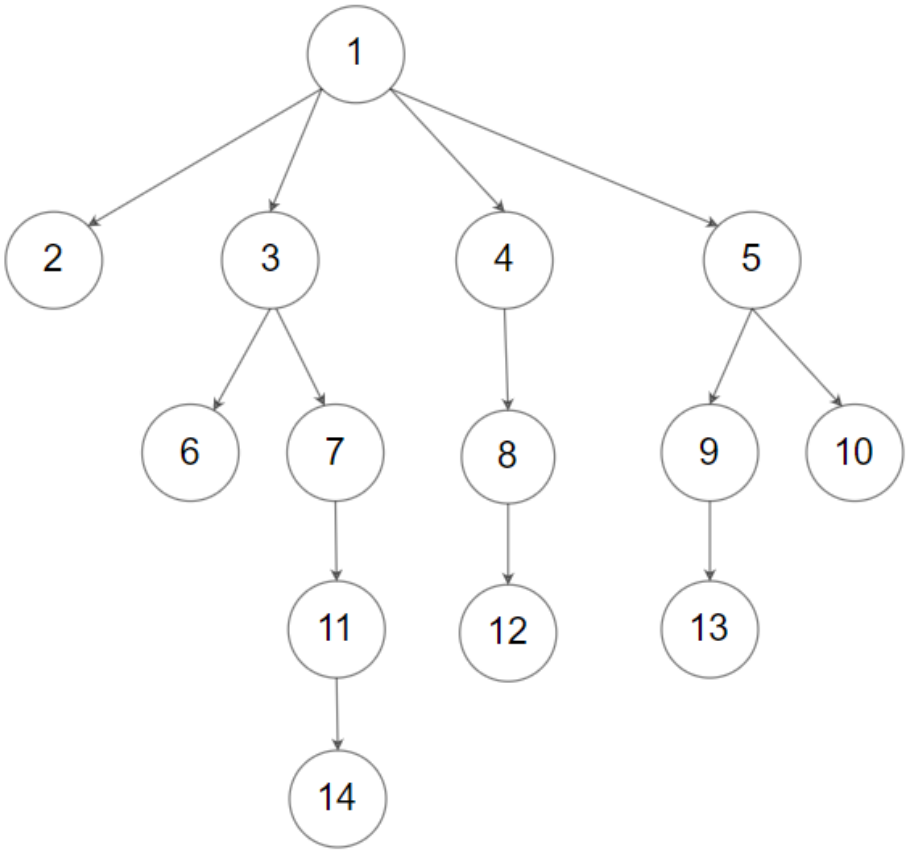



as

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

. Note that this is just an example, you do not necessarily need to follow this format.

Or you can follow LeetCode's level order traversal serialization format, where each group of children is separated by the null value.




For example, the above tree may be 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]

.

You do not necessarily need to follow the above-suggested formats, there are many more different formats that work so please be creative and come up with different approaches yourself.

Example 1:

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 2:

Input:

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

Output:

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

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;
}
};
*/

class Codec {
public:
// Encodes a tree to a single string.
string serialize(Node* root) {

}

// Decodes your encoded data to tree.
Node* deserialize(string data) {

}
};

// Your Codec object will be instantiated and called as such:
```

```
// Codec codec;
// codec.deserialize(codec.serialize(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 Codec {
// Encodes a tree to a single string.
public String serialize(Node root) {

}

// Decodes your encoded data to tree.
public Node deserialize(String data) {

}
}

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

**Python3:**

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

class Codec:
def serialize(self, root: 'Node') -> str:
"""Encodes a tree to a single string.

:type root: Node
:rtype: str
"""



def deserialize(self, data: str) -> 'Node':
"""Decodes your encoded data to tree.

:type data: str
:rtype: Node
"""



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

**Python:**

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

```
class Codec:
def serialize(self, root):
"""Encodes a tree to a single string.

:type root: Node
:rtype: str
"""


def deserialize(self, data):
"""Decodes your encoded data to tree.

:type data: str
:rtype: Node
"""


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

**JavaScript:**

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

class Codec {
constructor() {

}

/**
* @param {_Node|null} root
* @return {string}
*/
// Encodes a tree to a single string.
```

```
serialize = function(root) {

};

/**
* @param {string} data
* @return {_Node|null}
*/
// Decodes your encoded data to tree.
deserialize = function(data) {

};
}

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

**TypeScript:**

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


class Codec {
constructor() {

}

// Encodes a tree to a single string.
serialize(root: _Node | null): string {
```

```
};

// Decodes your encoded data to tree.
deserialize(data: string): _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;
}
}
*/

public class Codec {
// Encodes a tree to a single string.
public string serialize(Node root) {

}

// Decodes your encoded data to tree.
public Node deserialize(string data) {
```

```
}
}

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

**Go:**

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

type Codec struct {

}

func Constructor() *Codec {

}

func (this *Codec) serialize(root *Node) string {

}

func (this *Codec) deserialize(data string) *Node {

}

/**
* Your Codec object will be instantiated and called as such:
* obj := Constructor();
* data := obj.serialize(root);
* ans := obj.deserialize(data);
*/
```

**Kotlin:**

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

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

}

// Decodes your encoded data to tree.
fun deserialize(data: String): Node? {

}
}

/**
* Your Codec object will be instantiated and called as such:
* var obj = Codec()
* var data = obj.serialize(root)
* var ans = obj.deserialize(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 = []
* }
* }
*/

class Codec {
func serialize(_ root: Node?) -> String {
```

```
}

func deserialize(_ data: String) -> Node? {

}
}

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

**Ruby:**

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

class Codec
# Encodes a tree to a single string.
# @param {Node} root
# @return {String}
def serialize(root)

end

# Decodes your encoded data to tree.
# @param {String} data
# @return {Node}
def deserialize(data)

end
end

# Your Codec object will be instantiated and called as such:
```

```
# obj = Codec.new()
# data = obj.seralize(root)
# ans = obj.desrialize(data)
```

**PHP:**

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

class Codec {
/**
* Encodes a tree to a single string.
* @param Node $root
* @return String
*/
function serialize($root) {

}

/**
* Decodes your encoded data to tree.
* @param String $data
* @return Node
*/
function deserialize($data) {

}
}

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

```
* $ret_2 = $obj->deserialize($data);
*/
```

**Scala:**

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

class Codec {
// Encodes a tree to a single string.
def serialize(root: Node): String = {

}

// Decodes your encoded data to tree.
def deserialize(data: String): Node = {

}
}

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

## Solutions

### C++ Solution:

```
/*
* Problem: Serialize and Deserialize N-ary Tree
* Difficulty: Hard
* Tags: string, tree, search
```

```
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

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

Node() {
// TODO: Implement optimized solution
return 0;
}

Node(int _val) {
val = _val;
}

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

class Codec {
public:
// Encodes a tree to a single string.
string serialize(Node* root) {

}

// Decodes your encoded data to tree.
Node* deserialize(string data) {

}
};
```

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

**Java Solution:**

```
/**
* Problem: Serialize and Deserialize N-ary Tree
* Difficulty: Hard
* Tags: string, tree, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* 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;
}
};
*/

class Codec {
// Encodes a tree to a single string.
public String serialize(Node root) {
```

```
}

// Decodes your encoded data to tree.
public Node deserialize(String data) {

}
}

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

### Python3 Solution:

```
"""
Problem: Serialize and Deserialize N-ary Tree
Difficulty: Hard
Tags: string, tree, search

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

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

class Codec:
def serialize(self, root: 'Node') -> str:
# TODO: Implement optimized solution
pass
```

### Python Solution:

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

class Codec:
def serialize(self, root):
"""Encodes a tree to a single string.

:type root: Node
:rtype: str
"""



def deserialize(self, data):
"""Decodes your encoded data to tree.

:type data: str
:rtype: Node
"""



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

### JavaScript Solution:

```
/**
* Problem: Serialize and Deserialize N-ary Tree
* Difficulty: Hard
* Tags: string, tree, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* 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;
* };
*/

class Codec {
constructor() {

}

/**
* @param {_Node|null} root
* @return {string}
*/
// Encodes a tree to a single string.
serialize = function(root) {

};

/**
* @param {string} data
* @return {_Node|null}
*/
// Decodes your encoded data to tree.
deserialize = function(data) {

};
}

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

**TypeScript Solution:**

```
/**
* Problem: Serialize and Deserialize N-ary Tree
* Difficulty: Hard
* Tags: string, tree, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* 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 = [];
* }
* }
*/


class Codec {
constructor() {

}

// Encodes a tree to a single string.
serialize(root: _Node | null): string {

};

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

};
}

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

**C# Solution:**

```
/*
* Problem: Serialize and Deserialize N-ary Tree
* Difficulty: Hard
* Tags: string, tree, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* 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 Codec {
// Encodes a tree to a single string.
public string serialize(Node root) {

}

// Decodes your encoded data to tree.
public Node deserialize(string data) {

}
}
```

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

### Go Solution:

```
// Problem: Serialize and Deserialize N-ary Tree
// Difficulty: Hard
// Tags: string, tree, search
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

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

type Codec struct {

}

func Constructor() *Codec {

}

func (this *Codec) serialize(root *Node) string {

}

func (this *Codec) deserialize(data string) *Node {

}

/**
* Your Codec object will be instantiated and called as such:
```

```
* obj := Constructor();
* data := obj.serialize(root);
* ans := obj.deserialize(data);
*/
```

**Kotlin Solution:**

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

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

}

// Decodes your encoded data to tree.
fun deserialize(data: String): Node? {

}
}

/**
* Your Codec object will be instantiated and called as such:
* var obj = Codec()
* var data = obj.serialize(root)
* var ans = obj.deserialize(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 = []
* }
* }
*/

class Codec {
func serialize(_ root: Node?) -> String {

}

func deserialize(_ data: String) -> Node? {

}
}

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

**Ruby Solution:**

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

class Codec
# Encodes a tree to a single string.
# @param {Node} root
# @return {String}
def serialize(root)

end
```

```
# Decodes your encoded data to tree.
# @param {String} data
# @return {Node}
def deserialize(data)

end
end

# Your Codec object will be instantiated and called as such:
# obj = Codec.new()
# data = obj.seralize(root)
# ans = obj.desrialize(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();
* }
* }
*/

class Codec {
/**
* Encodes a tree to a single string.
* @param Node $root
* @return String
*/
function serialize($root) {

}

/**
* Decodes your encoded data to tree.
* @param String $data
```

```
* @return Node
*/
function deserialize($data) {

}
}

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

**Scala Solution:**

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

class Codec {
// Encodes a tree to a single string.
def serialize(root: Node): String = {

}

// Decodes your encoded data to tree.
def deserialize(data: String): Node = {

}
}

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

```
*/
```