

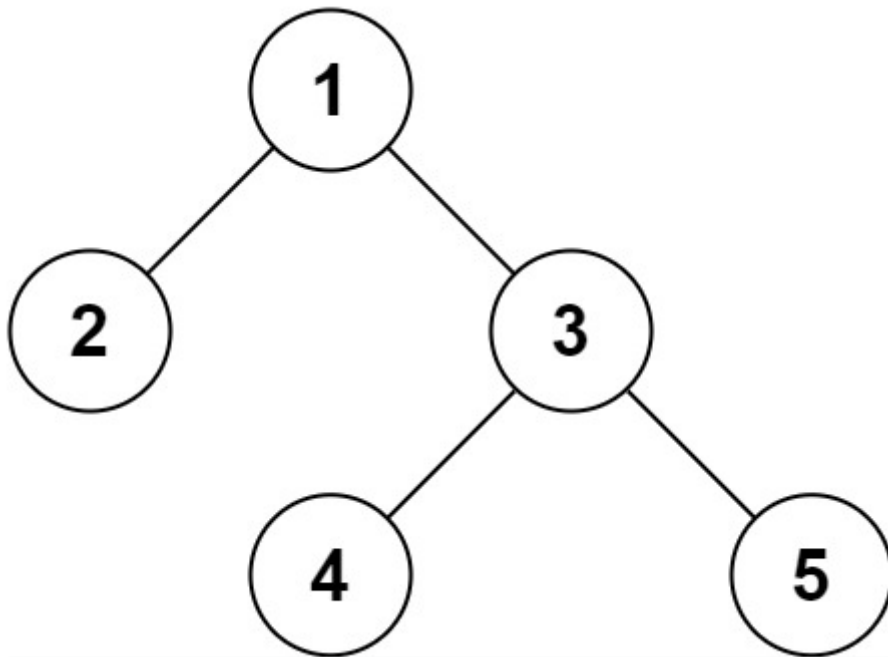
<https://leetcode.com/problems/serialize-and-deserialize-binary-tree/>

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 a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Clarification:** The input/output format is the same as [how LeetCode serializes a binary tree](#). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Example 1:**



Input: root = [1,2,3,null,null,4,5]

Output: [1,2,3,null,null,4,5]

**Example 2:**

Input: root = []

Output: []

**Constraints:**

- The number of nodes in the tree is in the range  $[0, 10^4]$ .
  - $-1000 \leq \text{Node.val} \leq 1000$
-

## What does [1,null,2,3] mean in binary tree representation?

<https://support.leetcode.com/hc/en-us/articles/360011883654-What-does-1-null-2-3-mean-in-binary-tree-representation>

-The input [1,null,2,3] represents the *serialized* format of a binary tree using **level order traversal**, where **null** signifies a path terminator where no node exists below.

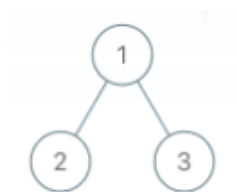
Examples:

1. `[]`

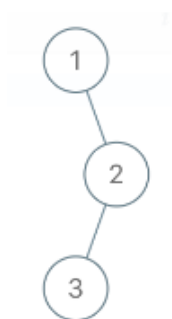
Empty tree.

The root is a reference to `NULL` (C/C++), `null` (Java/C#/Javascript), `None` (Python), or `nil` (Ruby).

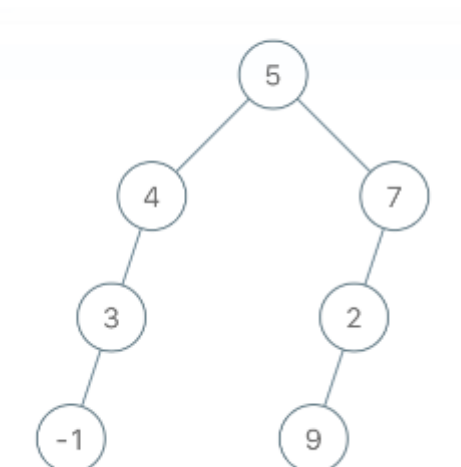
2. `[1,2,3]`



3. `[1,null,2,3]`



4. `[5,4,7,3,null,2,null,-1,null,9]`



**Solution 1: Preorder traversal for Serialize using DFS and for Deserialize using DFS (based on Queue) (30min)**

1. Use preorder traversal (root -> left -> right) and mark NULL as "#"
2. Deserialize with Queue based on preorder sequence (root -> left -> right)

```

1  /**
2   * Definition for a binary tree node.
3   * public class TreeNode {
4   *     int val;
5   *     TreeNode left;
6   *     TreeNode right;
7   *     TreeNode(int x) { val = x; }
8   * }
9   */
10 public class Codec {
11     private String NA = "X";
12     private String splitter = ",";
13
14     // Encodes a tree to a single string.
15     public String serialize(TreeNode root) {
16         StringBuilder sb = new StringBuilder();
17         serializeHelper(root, sb);
18         return sb.toString();
19     }
20     /**
21      e.g
22
23          5
24         / \
25        3  6
26       / \ \
27      2  4  7
28
29          5
30         / \
31        3  6
32       / \ / \
33      2  4 x  7
34     / \ / \
35    x x x x
36
37     => if considering NULL(x) =>
38
39          5
40         / \
41        3  6
42       / \ \
43      2  4  7
44     / \ / \
45    x x x x
46
47     => if considering NULL(x) =>
48
49          5
50         / \
51        3  6
52       / \ \
53      2  4  7
54     / \ / \
55    x x x x
56
57     => if considering NULL(x) =>
58
59          5
60         / \
61        3  6
62       / \ \
63      2  4  7
64     / \ / \
65    x x x x
66
67     => if considering NULL(x) =>
68
69          5
70         / \
71        3  6
72       / \ \
73      2  4  7
74     / \ / \
75    x x x x
76
77     => if considering NULL(x) =>
78
79          5
80         / \
81        3  6
82       / \ \
83      2  4  7
84     / \ / \
85    x x x x
86
87     => if considering NULL(x) =>
88
89          5
90         / \
91        3  6
92       / \ \
93      2  4  7
94     / \ / \
95    x x x x
96
97     => if considering NULL(x) =>
98
99          5
100         / \
101        3  6
102       / \ \
103      2  4  7
104     / \ / \
105    x x x x
106
107     => if considering NULL(x) =>
108
109          5
110         / \
111        3  6
112       / \ \
113      2  4  7
114     / \ / \
115    x x x x
116
117     => if considering NULL(x) =>
118
119          5
120         / \
121        3  6
122       / \ \
123      2  4  7
124     / \ / \
125    x x x x
126
127     => if considering NULL(x) =>
128
129          5
130         / \
131        3  6
132       / \ \
133      2  4  7
134     / \ / \
135    x x x x
136
137     => if considering NULL(x) =>
138
139          5
140         / \
141        3  6
142       / \ \
143      2  4  7
144     / \ / \
145    x x x x
146
147     => if considering NULL(x) =>
148
149          5
150         / \
151        3  6
152       / \ \
153      2  4  7
154     / \ / \
155    x x x x
156
157     => if considering NULL(x) =>
158
159          5
160         / \
161        3  6
162       / \ \
163      2  4  7
164     / \ / \
165    x x x x
166
167     => if considering NULL(x) =>
168
169          5
170         / \
171        3  6
172       / \ \
173      2  4  7
174     / \ / \
175    x x x x
176
177     => if considering NULL(x) =>
178
179          5
180         / \
181        3  6
182       / \ \
183      2  4  7
184     / \ / \
185    x x x x
186
187     => if considering NULL(x) =>
188
189          5
190         / \
191        3  6
192       / \ \
193      2  4  7
194     / \ / \
195    x x x x
196
197     => if considering NULL(x) =>
198
199          5
200         / \
201        3  6
202       / \ \
203      2  4  7
204     / \ / \
205    x x x x
206
207     => if considering NULL(x) =>
208
209          5
210         / \
211        3  6
212       / \ \
213      2  4  7
214     / \ / \
215    x x x x
216
217     => if considering NULL(x) =>
218
219          5
220         / \
221        3  6
222       / \ \
223      2  4  7
224     / \ / \
225    x x x x
226
227     => if considering NULL(x) =>
228
229          5
230         / \
231        3  6
232       / \ \
233      2  4  7
234     / \ / \
235    x x x x
236
237     => if considering NULL(x) =>
238
239          5
240         / \
241        3  6
242       / \ \
243      2  4  7
244     / \ / \
245    x x x x
246
247     => if considering NULL(x) =>
248
249          5
250         / \
251        3  6
252       / \ \
253      2  4  7
254     / \ / \
255    x x x x
256
257     => if considering NULL(x) =>
258
259          5
260         / \
261        3  6
262       / \ \
263      2  4  7
264     / \ / \
265    x x x x
266
267     => if considering NULL(x) =>
268
269          5
270         / \
271        3  6
272       / \ \
273      2  4  7
274     / \ / \
275    x x x x
276
277     => if considering NULL(x) =>
278
279          5
280         / \
281        3  6
282       / \ \
283      2  4  7
284     / \ / \
285    x x x x
286
287     => if considering NULL(x) =>
288
289          5
290         / \
291        3  6
292       / \ \
293      2  4  7
294     / \ / \
295    x x x x
296
297     => if considering NULL(x) =>
298
299          5
300         / \
301        3  6
302       / \ \
303      2  4  7
304     / \ / \
305    x x x x
306
307     => if considering NULL(x) =>
308
309          5
310         / \
311        3  6
312       / \ \
313      2  4  7
314     / \ / \
315    x x x x
316
317     => if considering NULL(x) =>
318
319          5
320         / \
321        3  6
322       / \ \
323      2  4  7
324     / \ / \
325    x x x x
326
327     => if considering NULL(x) =>
328
329          5
330         / \
331        3  6
332       / \ \
333      2  4  7
334     / \ / \
335    x x x x
336
337     => if considering NULL(x) =>
338
339          5
340         / \
341        3  6
342       / \ \
343      2  4  7
344     / \ / \
345    x x x x
346
347     => if considering NULL(x) =>
348
349          5
350         / \
351        3  6
352       / \ \
353      2  4  7
354     / \ / \
355    x x x x
356
357     => if considering NULL(x) =>
358
359          5
360         / \
361        3  6
362       / \ \
363      2  4  7
364     / \ / \
365    x x x x
366
367     => if considering NULL(x) =>
368
369          5
370         / \
371        3  6
372       / \ \
373      2  4  7
374     / \ / \
375    x x x x
376
377     => if considering NULL(x) =>
378
379          5
380         / \
381        3  6
382       / \ \
383      2  4  7
384     / \ / \
385    x x x x
386
387     => if considering NULL(x) =>
388
389          5
390         / \
391        3  6
392       / \ \
393      2  4  7
394     / \ / \
395    x x x x
396
397     => if considering NULL(x) =>
398
399          5
400         / \
401        3  6
402       / \ \
403      2  4  7
404     / \ / \
405    x x x x
406
407     => if considering NULL(x) =>
408
409          5
410         / \
411        3  6
412       / \ \
413      2  4  7
414     / \ / \
415    x x x x
416
417     => if considering NULL(x) =>
418
419          5
420         / \
421        3  6
422       / \ \
423      2  4  7
424     / \ / \
425    x x x x
426
427     => if considering NULL(x) =>
428
429          5
430         / \
431        3  6
432       / \ \
433      2  4  7
434     / \ / \
435    x x x x
436
437     => if considering NULL(x) =>
438
439          5
440         / \
441        3  6
442       / \ \
443      2  4  7
444     / \ / \
445    x x x x
446
447     => if considering NULL(x) =>
448
449          5
450         / \
451        3  6
452       / \ \
453      2  4  7
454     / \ / \
455    x x x x
456
457     => if considering NULL(x) =>
458
459          5
460         / \
461        3  6
462       / \ \
463      2  4  7
464     / \ / \
465    x x x x
466
467     => if considering NULL(x) =>
468
469          5
470         / \
471        3  6
472       / \ \
473      2  4  7
474     / \ / \
475    x x x x
476
477     => if considering NULL(x) =>
478
479          5
480         / \
481        3  6
482       / \
```

```

30     preorder serialize into string: 5,3,2,X,X,4,X,X,6,X,7,X,X,
31     */
32     // Style 1
33     private void serializeHelper(TreeNode root, StringBuilder sb) {
34         // Base case: Handle NULL
35         if(root == null) {
36             sb.append(NA).append(splitter);
37             return;
38         }
39         // Preorder traversal
40         sb.append(root.val).append(splitter);
41         serializeHelper(root.left, sb);
42         serializeHelper(root.right, sb);
43     }
44
45     // Style 2
46     //private void serializeHelper(TreeNode root, StringBuilder sb) {
47     //    if(root == null) {
48     //        sb.append(NA).append(splitter);
49     //    } else {
50     //        sb.append(root.val).append(splitter);
51     //        serializeHelper(root.left, sb);
52     //        serializeHelper(root.right, sb);
53     //    }
54     //}
55
56     // Decodes your encoded data to tree.
57     public TreeNode deserialize(String data) {
58         Queue<String> q = new LinkedList<String>();
59         q.addAll(Arrays.asList(data.split(splitter)));
60         return buildTree(q);
61     }
62
63     // Decode preorder traversal (5,3,2,X,X,4,X,X,6,X,7,X,X,) into tree
64     private TreeNode buildTree(Queue<String> q) {
65         String rootVal = q.poll();
66         if(rootVal.equals(NA)) {
67             return null;
68         }
69         TreeNode root = new TreeNode(Integer.valueOf(rootVal));

```

```

70     // Based on preorder, first build left subtree, then right subtree,
71     // and on each recursion Queue will pop out one element, since Queue
72     // is a object and no backtrack here, the number of elements on
73     // Queue will keep decreasing
74     root.left = buildTree(q);
75     root.right = buildTree(q);
76     return root;
77 }
78 }
79 // Your Codec object will be instantiated and called as such:
80 // Codec ser = new Codec();
81 // Codec deser = new Codec();
82 // TreeNode ans = deser.deserialize(ser.serialize(root));
83
84 Time Complexity:  $O(N)$ , where  $N \leq 10^4$  is number of nodes in the Binary Tree.
85 Space Complexity:  $O(N)$ 

```

**Difference between L297.Serialize and Deserialize Binary Tree (use only preorder to construct tree) and L105.Construct Binary Tree from Preorder and Inorder Traversal (use both preorder and inorder to construct tree) ?**

**Refer to**

<https://leetcode.com/problems/serialize-and-deserialize-binary-tree/discuss/74253/Easy-to-understand-Java-Solution/269310>

**Difference** between reconstruct the tree #105 **preorder/postorder + inorder** and this problem which just uses **preorder**

1. #105 preorder/postorder + inorder: **why we have to use 2 lists/traversals**

**The lists does not preserve the null, so we do not have an indicator to check if a node is in the left subtree or right subtree, so 2 traversals are needed.**

2. **But for this problem, we can preserve null, so we can reconstruct by just using 1 list, i.e. preorder list**

**Refer to**

<https://leetcode.com/problems/serialize-and-deserialize-binary-tree/discuss/74253/Easy-to-understand-Java-Solution/77362>

```

public class Codec {

    private final String splitter = ",";
    private final String na = "X";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {

        StringBuilder sb = new StringBuilder();
        buildString(sb, root);
        return sb.toString();
    }

    private void buildString(StringBuilder sb, TreeNode n){

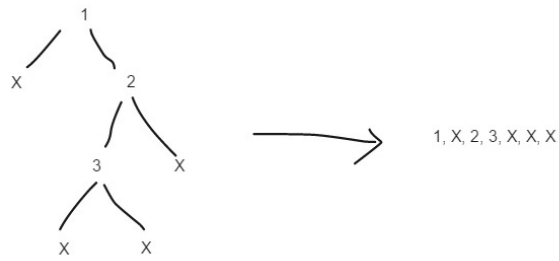
        if(n == null) sb.append(na).append(splitter);
        else {
            sb.append(n.val).append(splitter);
            buildString(sb, n.left);
            buildString(sb, n.right);
        }
    }

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

        Queue<String> q = new LinkedList();
        q.addAll(Arrays.asList(data.split(splitter)));
        return buildTree(q);
    }

    private TreeNode buildTree(Queue<String> q){
        String val = q.poll();
        if(val.equals(na)) return null;
        else {
            TreeNode t = new TreeNode(Integer.valueOf(val));
            t.left = buildTree(q);
            t.right = buildTree(q);
            return t;
        }
    }
}

```



RealtimeBoard.com

## Solution 2: Level order traversal for Serialize using BFS (Queue) and for Deserialize using BFS (Queue) (60min)

### Style 1: With "continue" statement in Serialize method

```

1  /**
2   * Definition for a binary tree node.
3   * public class TreeNode {
4   *     int val;
5   *     TreeNode left;
6   *     TreeNode right;
7   *     TreeNode(int x) { val = x; }
8   * }
9   */
10 public class Codec {
11     private String NA = "X";
12     private String splitter = ",";
13
14     /**
15     e.g
16
17         5
18         / \

```

```

18         3      6  => if considering NULL(x) => 3      6
19       /  \    \                /  \    /  \
20      2    4    7                2    4    x    7
21                               /  \ /  \    /  \
22                             x  x x  x    x  x

```

```

23
24     level order serialize into string: 5,3,6,2,4,X,7,X,X,X,X,X,X,
25 */
26 // Encodes a tree to a single string.
27 public String serialize(TreeNode root) {
28     if(root == null) {
29         return "";
30     }
31     StringBuilder sb = new StringBuilder();
32     Queue<TreeNode> q = new LinkedList<TreeNode>();
33     q.offer(root);
34     while(!q.isEmpty()) {
35         TreeNode node = q.poll();
36         if(node == null) {
37             sb.append(NA).append(splitter);
38             // Must terminate early since node already NULL,
39             // if not skip following statement then NullPointerException
40             // will happen because of 'node.val' not exist
41             continue;
42         }
43         sb.append(node.val).append(splitter);
44         // Add left and right child (even if it is NULL) on queue
45         q.offer(node.left);
46         q.offer(node.right);
47     }
48     return sb.toString();
49 }
50
51 // Decodes your encoded data to tree.
52 // Decode level order traversal (5,3,6,2,4,X,7,X,X,X,X,X,X,) into tree
53 public TreeNode deserialize(String data) {
54     if(data == "") {
55         return null;
56     }
57     Queue<TreeNode> q = new LinkedList<TreeNode>();

```

```

58     String[] values = data.split(splitter);
59     TreeNode root = new TreeNode(Integer.parseInt(values[0]));
60     q.offer(root);
61     for(int i = 1; i < values.length; i++) {
62         TreeNode node = q.poll();
63         if(!values[i].equals(NA)) {
64             TreeNode leftNode = new TreeNode(Integer.parseInt(values[i]));
65             node.left = leftNode;
66             q.offer(leftNode);
67         }
68         i++;
69         if(!values[i].equals(NA)) {
70             TreeNode rightNode = new TreeNode(Integer.parseInt(values[i]));
71             node.right = rightNode;
72             q.offer(rightNode);
73         }
74     }
75     return root;
76 }
77 }
78 // Your Codec object will be instantiated and called as such:
79 // Codec ser = new Codec();
80 // Codec deser = new Codec();
81 // TreeNode ans = deser.deserialize(ser.serialize(root));
82
83 Time Complexity: O(N), where N <= 10^4 is number of nodes in the Binary Tree.
84 Space Complexity: O(N)

```

## Style 2: Without "continue" statement in Serialize method

```

1  /**
2   * Definition for a binary tree node.
3   * public class TreeNode {
4   *     int val;
5   *     TreeNode left;
6   *     TreeNode right;
7   *     TreeNode(int x) { val = x; }
8   * }

```



```

9  */
10 public class Codec {
11
12     private String NA = "X";
13     private String splitter = ",";
14
15     /**
16     e.g
17
18         5                                5
19         /  \                            /  \
20        3    6    => if considering NULL(x) => 3    6
21       /  \  \                            /  \  /  \
22      2    4   7                            2    4    x   7
23                                           /  \ /  \
24                                          x  x x  x      x  x
25
26     level order serialize into string: 5,3,6,2,4,X,7,X,X,X,X,X,X,
27
28     */
29     // Encodes a tree to a single string.
30     public String serialize(TreeNode root) {
31         if(root == null) {
32             return "";
33         }
34         StringBuilder sb = new StringBuilder();
35         Queue<TreeNode> q = new LinkedList<TreeNode>();
36         q.offer(root);
37         while(!q.isEmpty()) {
38             TreeNode node = q.poll();
39             if(node == null) {
40                 sb.append(NA).append(splitter);
41                 // No need continue here, simply add else branch for handling not null
42                 nodes,
43
44                 // null nodes will be auto terminate processing here
45                 //continue;
46             } else {
47                 sb.append(node.val).append(splitter);
48                 // Add left and right child (even if it is NULL) on queue
49                 q.offer(node.left);
50                 q.offer(node.right);
51             }
52         }
53     }
54 }

```

```

48     }
49     return sb.toString();
50 }
51
52 // Decodes your encoded data to tree.
53 // Decode level order traversal (5,3,6,2,4,X,7,X,X,X,X,X,X,) into tree
54 public TreeNode deserialize(String data) {
55     if(data == "") {
56         return null;
57     }
58     Queue<TreeNode> q = new LinkedList<TreeNode>();
59     String[] values = data.split(splitter);
60     TreeNode root = new TreeNode(Integer.parseInt(values[0]));
61     q.offer(root);
62     for(int i = 1; i < values.length; i++) {
63         TreeNode node = q.poll();
64         if(!values[i].equals(NA)) {
65             TreeNode leftNode = new TreeNode(Integer.parseInt(values[i]));
66             node.left = leftNode;
67             q.offer(leftNode);
68         }
69         i++;
70         if(!values[i].equals(NA)) {
71             TreeNode rightNode = new TreeNode(Integer.parseInt(values[i]));
72             node.right = rightNode;
73             q.offer(rightNode);
74         }
75     }
76     return root;
77 }
78 }
79
80 // Your Codec object will be instantiated and called as such:
81 // Codec ser = new Codec();
82 // Codec deser = new Codec();
83 // TreeNode ans = deser.deserialize(ser.serialize(root));

```

**Refer to**

<https://leetcode.com/problems/serialize-and-deserialize-binary-tree/discuss/74264/Short-and-straight-forward-BFS-Java-code-with-a-queue> Here I use typical BFS method to handle a binary tree. I use string n to represent null values. The string of the binary tree in the example will be "1 2 3 n n 4 5 n n n n".

When deserialize the string, I assign left and right child for each not-null node, and add the not-null children to the queue, waiting to be handled later.

```
1 public class Codec {
2     public String serialize(TreeNode root) {
3         if (root == null) return "";
4         Queue<TreeNode> q = new LinkedList<>();
5         StringBuilder res = new StringBuilder();
6         q.add(root);
7         while (!q.isEmpty()) {
8             TreeNode node = q.poll();
9             if (node == null) {
10                 res.append("n ");
11                 continue;
12             }
13             res.append(node.val + " ");
14             q.add(node.left);
15             q.add(node.right);
16         }
17         return res.toString();
18     }
19     public TreeNode deserialize(String data) {
20         if (data == "") return null;
21         Queue<TreeNode> q = new LinkedList<>();
22         String[] values = data.split(" ");
23         TreeNode root = new TreeNode(Integer.parseInt(values[0]));
24         q.add(root);
25         for (int i = 1; i < values.length; i++) {
26             TreeNode parent = q.poll();
27             if (!values[i].equals("n")) {
28                 TreeNode left = new TreeNode(Integer.parseInt(values[i]));
29                 parent.left = left;
30                 q.add(left);
31             }
32         }
33     }
34 }
```

```

32         if (!values[++i].equals("n")) {
33             TreeNode right = new TreeNode(Integer.parseInt(values[i]));
34             parent.right = right;
35             q.add(right);
36         }
37     }
38     return root;
39 }
40 }

```

## Refer to

<https://leetcode.com/problems/serialize-and-deserialize-binary-tree/discuss/74264/Short-and-straight-forward-BFS-Java-code-with-a-queue/980762>

Q: I also think if we are using a level order traversal, the left child is at index  $2 * \text{index} + 1$  and right child at index  $2 * \text{index} + 2$  can this info be applied in deserializing, and avoid the extra space, and do it recursively ?

A: Not able to because its difficult to update 2 root in one round for next iteration

```

1    // 5,3,6,2,4,X,7,X,X,X,X,X,X,
2    // Decodes your encoded data to tree.
3    public TreeNode deserialize(String data) {
4        if(data == "") {
5            return null;
6        }
7        //Queue<TreeNode> q = new LinkedList<TreeNode>();
8        String[] values = data.split(splitter);
9        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
10       //int index = 0;
11       //q.offer(root);
12       int len = values.length;
13       for(int i = 0; i < values.length; i++) {
14           int leftIndex = i * 2 + 1;
15           int rightIndex = i * 2 + 2;
16           if(leftIndex < len && !values[leftIndex].equals(NA)) {
17               root.left = new TreeNode(Integer.parseInt(values[leftIndex]));
18           }

```

```
19         if(rightIndex < len && !values[rightIndex].equals(NA)) {
20             root.right = new TreeNode(Integer.parseInt(values[rightIndex]));
21         }
22         // Not able to update both left / right subtree node for next iteration
23         // in one for loop
24     }
25     return root;
26 }
```

---

### Video explain:

[Serialize and Deserialize Binary Tree - Preorder Traversal - Leetcode 297 - Python](#)

<https://www.youtube.com/watch?v=u4JAI2JJhI8>

### Refer to

[📄L449.Serialize and Deserialize BST \(Ref.L297\)](#)