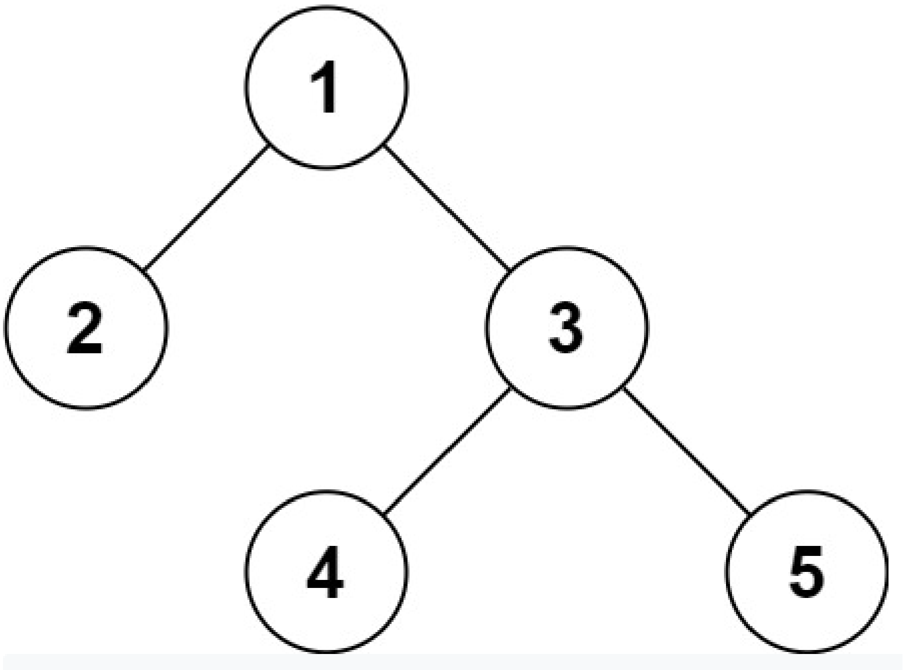
https://leetcode.com/problems/serialize-and-deserialize-binary-tree/

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 <= Node.val <= 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

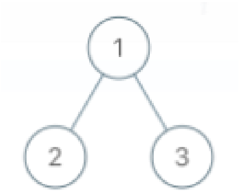
-The input [1,null,2,3] represents the *serialized* format of a binary tree using **level order traversal**, **where null signifies a path terminator where no node exists below**.

Examples:

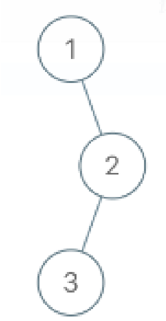
1. `[]`

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

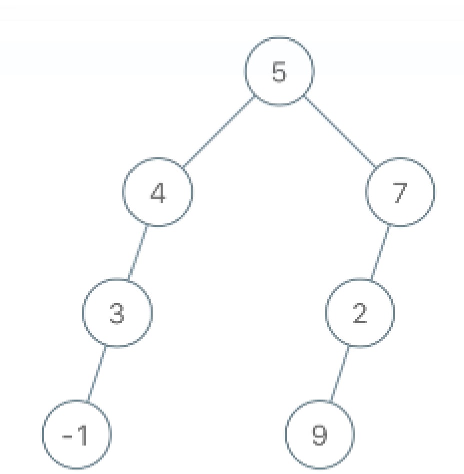
2. `[1,2,3]`



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



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



Attempt 1: 2022-11-01

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

Two key points:

1. Use preorder traversal (root -> left -> right) and mark NULL as "#"

2. Deserialize with Queue based on preorder sequence (root -> left -> right)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
    private String NA = "X";
    private String spliter = ",";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }
    /**
     e.g
               5                                        5
             /   \                                    /     \
            3     6   => if considering NULL(x) =>  3       6
           /  \    \                               / \     /  \
          2    4    7                             2   4    x   7
                                                 / \ / \       / \
                                                x  x x  x      x  x

       preorder serialize into string: 5,3,2,X,X,4,X,X,6,X,7,X,X,
    */
    // Style 1
    private void serializeHelper(TreeNode root, StringBuilder sb) {
        // Base case: Handle NULL
        if(root == null) {
```

```
            sb.append(NA).append(spliter);
            return;
        }
        // Preorder traversal
        sb.append(root.val).append(spliter);
        serializeHelper(root.left, sb);
        serializeHelper(root.right, sb);
    }

    // Style 2
    //private void serializeHelper(TreeNode root, StringBuilder sb) {
    //    if(root == null) {
    //        sb.append(NA).append(spliter);
    //    } else {
    //        sb.append(root.val).append(spliter);
    //        serializeHelper(root.left, sb);
    //        serializeHelper(root.right, sb);
    //    }
    //}

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Queue<String> q = new LinkedList<String>();
        q.addAll(Arrays.asList(data.split(spliter)));
        return buildTree(q);
    }

    // Decode preorder traversal (5,3,2,X,X,4,X,X,6,X,7,X,X,) into tree
    private TreeNode buildTree(Queue<String> q) {
        String rootVal = q.poll();
        if(rootVal.equals(NA)) {
            return null;
        }
        TreeNode root = new TreeNode(Integer.valueOf(rootVal));
        // Based on preorder, first build left subtree, then right subtree,
        // and on each recursion Queue will pop out one element, since Queue
        // is a object and no backtrack here, the number of elements on
        // Queue will keep decreasing
        root.left = buildTree(q);
        root.right = buildTree(q);
        return root;
    }
```

```
}
// Your Codec object will be instantiated and called as such:
// Codec ser = new Codec();
// Codec deser = new Codec();
// TreeNode ans = deser.deserialize(ser.serialize(root));

Time Complexity: O(N), where N <= 10^4 is number of nodes in the Binary Tree.
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

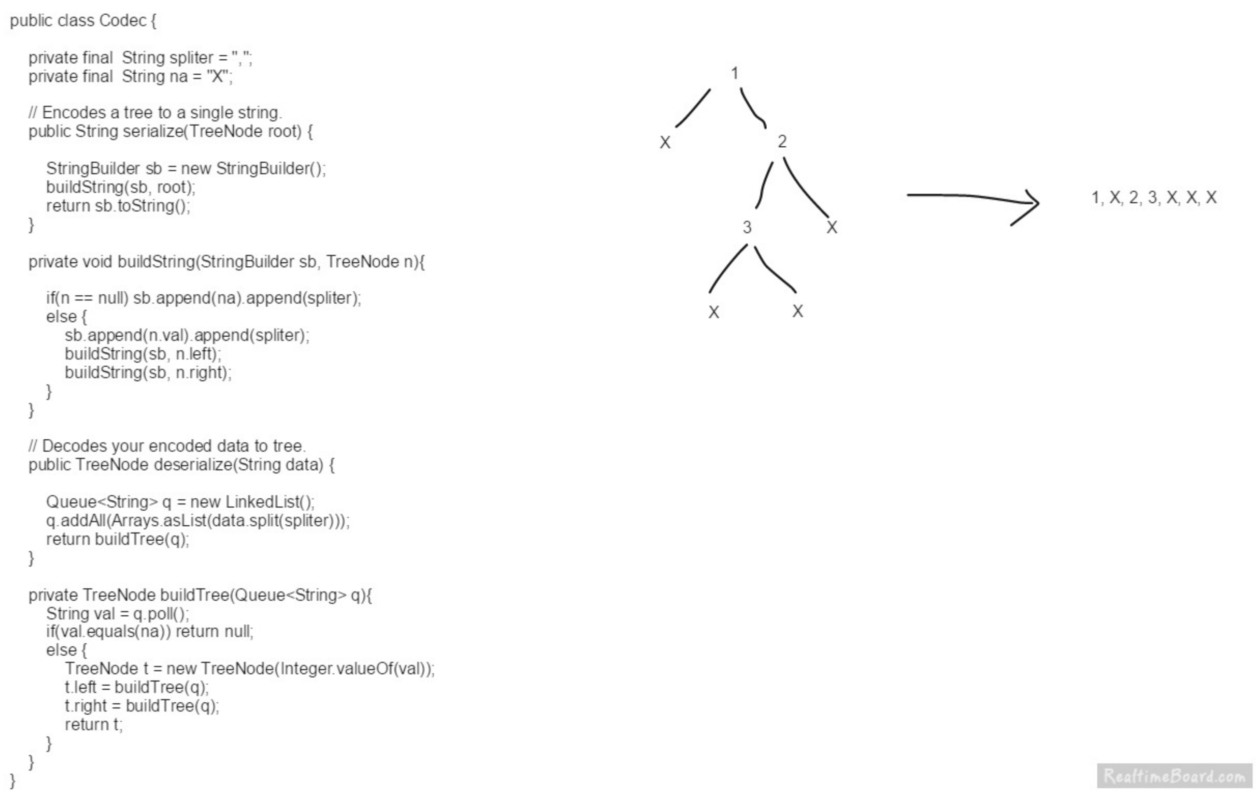
Difference between reconstruct the tree #105 **preorder/postorder + inorder** and this problem which just uses **preorder**

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



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

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
    private String NA = "X";
    private String spliter = ",";

    /**
     e.g
                5                                         5
              /   \                                     /     \
             3     6    => if considering NULL(x) =>  3        6
           /  \     \                               / \      /   \
```

```
          2    4    7                             2   4    x   7
                                                 / \ / \       / \
                                                x  x x  x      x  x

       level order serialize into string: 5,3,6,2,4,X,7,X,X,X,X,X,X,
    */
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if(root == null) {
            return "";
        }
        StringBuilder sb = new StringBuilder();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while(!q.isEmpty()) {
            TreeNode node = q.poll();
            if(node == null) {
                sb.append(NA).append(spliter);
                // Must terminate early since node already NULL,
                // if not skip following statement then NullPointerException
                // will happen because of 'node.val' not exist
                continue;
            }
            sb.append(node.val).append(spliter);
            // Add left and right child (even if it is NULL) on queue
            q.offer(node.left);
            q.offer(node.right);
        }
        return sb.toString();
    }

    // Decodes your encoded data to tree.
    // Decode level order traversal (5,3,6,2,4,X,7,X,X,X,X,X,X,) into tree
    public TreeNode deserialize(String data) {
        if(data == "") {
            return null;
        }
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        String[] values = data.split(spliter);
        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
        q.offer(root);
        for(int i = 1; i < values.length; i++) {
```

```
                TreeNode node = q.poll();
                if(!values[i].equals(NA)) {
                    TreeNode leftNode = new TreeNode(Integer.parseInt(values[i]));
                    node.left = leftNode;
                    q.offer(leftNode);
                }
                i++;
                if(!values[i].equals(NA)) {
                    TreeNode rightNode = new TreeNode(Integer.parseInt(values[i]));
                    node.right = rightNode;
                    q.offer(rightNode);
                }
            }
            return root;
        }
}
// Your Codec object will be instantiated and called as such:
// Codec ser = new Codec();
// Codec deser = new Codec();
// TreeNode ans = deser.deserialize(ser.serialize(root));

Time Complexity: O(N), where N <= 10^4 is number of nodes in the Binary Tree.
Space Complexity: O(N)
```

Style 2: Without "continue" statement in Serialize method

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

        private String NA = "X";
    private String spliter = ",";

```

```
    /**
     e.g
               5                                        5
             /   \                                   /     \
            3     6   => if considering NULL(x) =>  3       6
           /  \    \                               / \     /  \
          2    4    7                             2   4    x   7
                                                 / \ / \       / \
                                                x  x x  x      x  x

       level order serialize into string: 5,3,6,2,4,X,7,X,X,X,X,X,X,
    */
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if(root == null) {
            return "";
        }
        StringBuilder sb = new StringBuilder();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(root);
        while(!q.isEmpty()) {
            TreeNode node = q.poll();
            if(node == null) {
                sb.append(NA).append(spliter);
                // No need continue here, simply add else branch for handling not null nodes,
                // null nodes will be auto terminate processing here
                //continue;
            } else {
                sb.append(node.val).append(spliter);
                // Add left and right child (even if it is NULL) on queue
                q.offer(node.left);
                q.offer(node.right);
            }
        }
        return sb.toString();
    }

    // Decodes your encoded data to tree.
    // Decode level order traversal (5,3,6,2,4,X,7,X,X,X,X,X,X,) into tree
    public TreeNode deserialize(String data) {
        if(data == "") {
```

```
            return null;
        }
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        String[] values = data.split(spliter);
        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
        q.offer(root);
        for(int i = 1; i < values.length; i++) {
            TreeNode node = q.poll();
            if(!values[i].equals(NA)) {
                TreeNode leftNode = new TreeNode(Integer.parseInt(values[i]));
                node.left = leftNode;
                q.offer(leftNode);
            }
            i++;
            if(!values[i].equals(NA)) {
                TreeNode rightNode = new TreeNode(Integer.parseInt(values[i]));
                node.right = rightNode;
                q.offer(rightNode);
            }
        }
        return root;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec ser = new Codec();
// Codec deser = new Codec();
// 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.

```
public class Codec {
    public String serialize(TreeNode root) {
```

```java
        if (root == null) return "";
        Queue<TreeNode> q = new LinkedList<>();
        StringBuilder res = new StringBuilder();
        q.add(root);
        while (!q.isEmpty()) {
            TreeNode node = q.poll();
            if (node == null) {
                res.append("n ");
                continue;
            }
            res.append(node.val + " ");
            q.add(node.left);
            q.add(node.right);
        }
        return res.toString();
    }
    public TreeNode deserialize(String data) {
        if (data == "") return null;
        Queue<TreeNode> q = new LinkedList<>();
        String[] values = data.split(" ");
        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
        q.add(root);
        for (int i = 1; i < values.length; i++) {
            TreeNode parent = q.poll();
            if (!values[i].equals("n")) {
                TreeNode left = new TreeNode(Integer.parseInt(values[i]));
                parent.left = left;
                q.add(left);
            }
            if (!values[++i].equals("n")) {
                TreeNode right = new TreeNode(Integer.parseInt(values[i]));
                parent.right = right;
                q.add(right);
            }
        }
        return root;
    }
}
```

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 *index + 1 and right child at index 2 * 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

```
// 5,3,6,2,4,X,7,X,X,X,X,X,X,
// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    if(data == "") {
        return null;
    }
    //Queue<TreeNode> q = new LinkedList<TreeNode>();
    String[] values = data.split(spliter);
    TreeNode root = new TreeNode(Integer.parseInt(values[0]));
    //int index = 0;
    //q.offer(root);
    int len = values.length;
    for(int i = 0; i < values.length; i++) {
        int leftIndex = i * 2 + 1;
        int rightIndex = i * 2 + 2;
        if(leftIndex < len && !values[leftIndex].equals(NA)) {
            root.left = new TreeNode(Integer.parseInt(values[leftIndex]));
        }
        if(rightIndex < len && !values[rightIndex].equals(NA)) {
            root.right = new TreeNode(Integer.parseInt(values[rightIndex]));
        }
        // Not able to update both left / right subtree node for next iteration
        // in one for loop
    }
    return root;
}
```

Refer to QWEN

Q1: Leetcode 297.Serialize and Deserialize Binary Tree java solution ?

LeetCode 297 requires designing an algorithm to serialize and deserialize a binary tree. The key is to use a consistent traversal method for both processes. Here are two common approaches:

Approach 1: DFS (Preorder Traversal) with StringBuilder

```java
import java.util.*;

public class Codec {
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    private void serializeHelper(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("null,");
            return;
        }
        sb.append(node.val).append(",");
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Queue<String> nodes = new LinkedList<>(Arrays.asList(data.split(",")));
        return deserializeHelper(nodes);
    }

    private TreeNode deserializeHelper(Queue<String> nodes) {
        String val = nodes.poll();
        if (val.equals("null")) {
            return null;
        }
        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.left = deserializeHelper(nodes);
        node.right = deserializeHelper(nodes);
        return node;
```

```
    }
}
```

Approach 2: BFS (Level Order Traversal)

```
import java.util.*;

public class Codec {
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if (root == null) return "";

        StringBuilder sb = new StringBuilder();
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if (node == null) {
                sb.append("null,");
            } else {
                sb.append(node.val).append(",");
                queue.offer(node.left);
                queue.offer(node.right);
            }
        }
        return sb.toString();
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if (data.isEmpty()) return null;

        String[] nodes = data.split(",");
        TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        int i = 1;
```

```java
        while (!queue.isEmpty() && i < nodes.length) {
            TreeNode node = queue.poll();

            // Left child
            if (!nodes[i].equals("null")) {
                node.left = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(node.left);
            }
            i++;

            // Right child
            if (i < nodes.length && !nodes[i].equals("null")) {
                node.right = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(node.right);
            }
            i++;
        }
        return root;
    }
}
```

Key Points:

- **DFS Approach**: Uses preorder traversal (root, left, right). Each null node is represented as "null". The deserialization follows the same preorder sequence using a queue to track nodes .
- **BFS Approach**: Uses level-order traversal, representing the tree level by level. Null children are also serialized to maintain structural information .
- **Time Complexity**: Both approaches are O(N) where N is the number of nodes.
- **Space Complexity**: O(N) for storing the serialized string and recursion/queue space.

The DFS approach is generally more space-efficient for skewed trees, while BFS maintains the level structure explicitly. Both solutions ensure that the deserialized tree is identical to the original tree structure.

Q2: Can you elaborate why in "Approach 1: DFS (Preorder Traversal) with StringBuilder" for serializeHelper() method we choose void return pattern ?

Great question! The choice to use a `void` return type with a `StringBuilder` parameter in the `serializeHelper()` method is primarily driven by **performance optimization** and **memory efficiency**.

Why Not Return String?

Let's consider what would happen if we used a String-returning approach:

```java
// INEFFICIENT APPROACH
private String serializeHelper(TreeNode node) {
    if (node == null) {
        return "null,";
    }
    // This creates multiple intermediate String objects!
    return node.val + "," + serializeHelper(node.left) + serializeHelper(node.right);
}
```

The Problem with String Concatenation

In Java, **Strings are immutable**. Every time you concatenate strings using `+`, Java creates a **new String object**.

For a tree with N nodes, this leads to:

1. **O(N²) time complexity** in the worst case (for skewed trees)
2. **Excessive memory allocation** - each concatenation creates temporary objects
3. **Frequent garbage collection** due to many short-lived String objects

For example, with a right-skewed tree of 4 nodes [1,2,3,4]:

- `"1," + "2," + "3," + "4," + "null,"`
- Creates intermediate strings: `"1,"`, `"1,2,"`, `"1,2,3,"`, `"1,2,3,4,"`, etc.

Why StringBuilder with Void Works Better

```java
private void serializeHelper(TreeNode node, StringBuilder sb) {
    if (node == null) {
        sb.append("null,");
        return;
    }
    sb.append(node.val).append(",");
    serializeHelper(node.left, sb);
    serializeHelper(node.right, sb);
}
```

Advantages:

1. **Single mutable buffer**: Only one `StringBuilder` object is created and passed by reference
2. **O(1) append operations** (amortized): `StringBuilder.append()` modifies the existing buffer
3. **O(N) total time complexity**: Each node is visited once, and each append is O(1)

4. **Minimal memory overhead**: No intermediate String objects are created
5. **Cache-friendly**: Sequential memory access pattern

Memory Flow Comparison:

String-returning approach:

```
serializeHelper(1) → "1," + serializeHelper(2)
serializeHelper(2) → "2," + serializeHelper(3)
serializeHelper(3) → "3," + serializeHelper(4)
serializeHelper(4) → "4," + "null,"
// Now unwind: "4,null," → "3,4,null," → "2,3,4,null," → "1,2,3,4,null,"
// Creates 4 intermediate strings!
```

StringBuilder approach:

```
Single StringBuilder: initially empty
Visit 1: append "1," → ["1,"]
Visit 2: append "2," → ["1,2,"]
Visit 3: append "3," → ["1,2,3,"]
Visit 4: append "4," → ["1,2,3,4,"]
Visit null: append "null," → ["1,2,3,4,null,"]
// Only one buffer modified in-place!
```

Alternative: StringBuilder Return Pattern

You could also return StringBuilder, but it's unnecessary:

```
private StringBuilder serializeHelper(TreeNode node, StringBuilder sb) {
    if (node == null) {
        sb.append("null,");
        return sb;
    }
    sb.append(node.val).append(",");
    serializeHelper(node.left, sb);
    serializeHelper(node.right, sb);
    return sb; // Unnecessary since sb is modified in-place
```

```
}
```

Since `StringBuilder` is **mutable and passed by reference**, any modifications made to it inside the method are visible to the caller. Therefore, returning it provides no additional benefit and adds unnecessary complexity.

Summary

The `void` return pattern with `StringBuilder` parameter is chosen because:

- **Performance**: O(N) time vs O(N²) with String concatenation
- **Memory efficiency**: Single buffer vs multiple intermediate String objects
- **Simplicity**: No need to return the buffer since it's modified in-place
- **Best practice**: This is the standard idiom for efficient string building in recursive algorithms

This pattern is commonly used in tree serialization, path building, and other recursive string construction problems where performance matters.

Q3: I have a tree like below, can you simulate the process how the two approaches work ?

```
               5                                        5
             /   \                                   /     \
            3     6   => if considering NULL(x) =>  3       6
           /  \    \                               / \     /  \
          2    4    7                             2   4    x   7
                                                 / \ / \       / \
                                                x  x x  x      x  x
```

I'll simulate both approaches step by step for your tree:

Approach 1: Value Return (String Concatenation)

Let's trace the recursive calls:

serializeHelper(5)

- Calls `serializeHelper(3)` for left subtree
- Calls `serializeHelper(6)` for right subtree
- Returns `"5," + result_left + result_right`

serializeHelper(3)

- Calls `serializeHelper(2)` for left
- Calls `serializeHelper(4)` for right

- Returns `"3," + result_2 + result_4`

serializeHelper(2)

- Calls `serializeHelper(null)` → returns `"null,"`
- Calls `serializeHelper(null)` → returns `"null,"`
- Returns `"2,null,null,"`

serializeHelper(4)

- Calls `serializeHelper(null)` → returns `"null,"`
- Calls `serializeHelper(null)` → returns `"null,"`
- Returns `"4,null,null,"`

Now back to **serializeHelper(3)**:

- Returns `"3," + "2,null,null," + "4,null,null,"` = `"3,2,null,null,4,null,null,"`

serializeHelper(6)

- Calls `serializeHelper(null)` for left → returns `"null,"`
- Calls `serializeHelper(7)` for right

serializeHelper(7)

- Calls `serializeHelper(null)` → returns `"null,"`
- Calls `serializeHelper(null)` → returns `"null,"`
- Returns `"7,null,null,"`

Back to **serializeHelper(6)**:

- Returns `"6," + "null," + "7,null,null,"` = `"6,null,7,null,null,"`

Finally, **serializeHelper(5)**:

- Returns `"5," + "3,2,null,null,4,null,null," + "6,null,7,null,null,"`
- Final result: `"5,3,2,null,null,4,null,null,6,null,7,null,null,"`

Key Issue: At each return step, Java creates new String objects:

- `"2,null,null,"` (new object)
- `"4,null,null,"` (new object)
- `"3,2,null,null,4,null,null,"` (new object combining previous two)
- And so on...

This creates O(N²) intermediate strings.

Approach 2: Void Return (StringBuilder)

We start with an empty `StringBuilder sb = new StringBuilder()`

serializeHelper(5, sb)

- `sb.append("5,")` → `sb = "5,"`
- Calls `serializeHelper(3, sb)`
- Calls `serializeHelper(6, sb)`

serializeHelper(3, sb)

- `sb.append("3,")` → `sb = "5,3,"`
- Calls `serializeHelper(2, sb)`

- Calls `serializeHelper(4, sb)`

serializeHelper(2, sb)

- `sb.append("2,")` → `sb = "5,3,2,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,null,"`
- Returns (void)

serializeHelper(4, sb)

- `sb.append("4,")` → `sb = "5,3,2,null,null,4,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,null,4,null,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,null,4,null,null,"`
- Returns (void)

Back to **serializeHelper(3, sb)** - already done, returns

serializeHelper(6, sb)

- `sb.append("6,")` → `sb = "5,3,2,null,null,4,null,null,6,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,null,4,null,null,6,null,"`
- Calls `serializeHelper(7, sb)`

serializeHelper(7, sb)

- `sb.append("7,")` → `sb = "5,3,2,null,null,4,null,null,6,null,7,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,null,4,null,null,6,null,7,null,"`
- Calls `serializeHelper(null, sb)` → `sb.append("null,")` → `sb = "5,3,2,null,null,4,null,null,6,null,7,null,null,"`
- Returns (void)

Final result: `"5,3,2,null,null,4,null,null,6,null,7,null,null,"`

Key Differences:

1. **Memory Usage**:
   - Value return: Creates multiple intermediate String objects
   - Void return: Modifies single StringBuilder in-place
2. **Performance**:
   - Value return: $O(N^2)$ time due to string concatenation
   - Void return: $O(N)$ time with $O(1)$ append operations
3. **Object Creation**:
   - Value return: New String at every return statement
   - Void return: Only one StringBuilder object throughout

Both approaches produce the same final serialized string, but the StringBuilder approach is much more efficient for larger trees. This serialization format using preorder traversal with null markers ensures the tree can be uniquely reconstructed during deserialization .

Q4: So from your simulation, do you mean below solution

```java
// Value return approach
private String serializeHelper(TreeNode node) {
    if (node == null) {
        return "null,";
    }
    // This creates multiple intermediate String objects!
    return node.val + "," + serializeHelper(node.left) + serializeHelper(node.right);
}
```

equal to below ?

```java
// Value return approach
private String serializeHelper(TreeNode node) {
    if (node == null) {
        return "null,";
    }
    // This creates multiple intermediate String objects!
    String a = node.val;
    String b = serializeHelper(node.left);
    String c = serializeHelper(node.right);
    return a + "," + b + c;
}
```

Yes, **exactly!** Both implementations are functionally equivalent and will produce the same result. The second version just makes the intermediate string creation more explicit.

Let me break down what happens in both:

Original version:

```java
return node.val + "," + serializeHelper(node.left) + serializeHelper(node.right);
```

Explicit version:

```java
String a = String.valueOf(node.val);  // node.val is int, so it gets converted to String
String b = serializeHelper(node.left);
String c = serializeHelper(node.right);
return a + "," + b + c;
```

What actually happens under the hood:

In Java, when you use the `+` operator with strings, the compiler typically converts it to use `StringBuilder` internally, but **only for that single expression**. So your original line:

```java
return node.val + "," + serializeHelper(node.left) + serializeHelper(node.right);
```

Gets compiled to something like:

```java
StringBuilder temp = new StringBuilder();
temp.append(node.val);
temp.append(",");
temp.append(serializeHelper(node.left));  // This returns a String
temp.append(serializeHelper(node.right)); // This returns a String
return temp.toString();
```

However, **the key inefficiency remains**: each recursive call to `serializeHelper()` returns a **new String object**, and when you concatenate these returned strings with other strings, you're still creating new intermediate String objects at each level of recursion.

The real problem isn't the `+` operators themselves, but the recursive string returns.

Whether you write it as:

- `return node.val + "," + left + right;`
- or explicitly assign to variables first

You still have the same fundamental issue: **each recursive call returns a complete string**, which then needs to be concatenated with other strings, creating new objects at every level.

Example with your tree:

With the explicit version for node `3`:

```java
String a = "3";
String b = "2,null,null,";  // This is a complete String object from serializeHelper(2)
String c = "4,null,null,";  // This is a complete String object from serializeHelper(4)
return "3" + "," + "2,null,null," + "4,null,null,";
```

This still creates a new String `"3,2,null,null,4,null,null,"` by concatenating three existing String objects.
Whereas with the StringBuilder approach, there's only **one** StringBuilder that gets passed around and modified in-place - no intermediate String objects are created during the recursion.
So yes, both versions you showed are equivalent in terms of performance and behavior - they both suffer from the same O(N²) time complexity issue due to recursive string returns and concatenation.

Q5: In BFS solution deserialize() method, why we need variable 'i' ?

Great question! In the BFS deserialize() method, the variable `i` is needed because we're working with a **flat array of serialized values** and need to track our current position in that array as we reconstruct the tree level by level.
Let me walk through why `i` is essential:

The Problem Setup

After serialization using BFS, we get a flat array like:

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

This represents the tree level by level:

- Level 0: [1]
- Level 1: [2, 3]
- Level 2: [null, 4, null, null]
- Level 3: [5, 6]

Why We Need Index `i`

When deserializing with BFS, we use a **queue to track nodes that need children**, but we also need to know **which serialized values correspond to the next children**.
Here's what happens step by step:

```java
String[] nodes = data.split(",");
// nodes = ["1", "2", "3", "null", "4", "null", "null", "5", "6"]

TreeNode root = new TreeNode(Integer.parseInt(nodes[0])); // nodes[0] = "1"
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);

int i = 1; // Start from index 1 because index 0 is the root
```

Iteration 1:

- Process node `1` from queue
- Its left child = `nodes[i]` = `nodes[1]` = `"2"` → create node, `i` becomes 2
- Its right child = `nodes[i]` = `nodes[2]` = `"3"` → create node, `i` becomes 3

Iteration 2:

- Process node `2` from queue
- Its left child = `nodes[i]` = `nodes[3]` = `"null"` → no node, `i` becomes 4
- Its right child = `nodes[i]` = `nodes[4]` = `"4"` → create node, `i` becomes 5

Iteration 3:

- Process node `3` from queue
- Its left child = `nodes[i]` = `nodes[5]` = `"null"` → no node, `i` becomes 6
- Its right child = `nodes[i]` = `nodes[6]` = `"null"` → no node, `i` becomes 7

And so on...

Why Can't We Just Use the Queue Size?

The queue only tells us **how many nodes are waiting for children**, but it doesn't tell us **where we are in the serialized array**.

- The queue manages the **tree structure** (which nodes need children)
- The index `i` manages the **serialization order** (which values come next)

Alternative Without Explicit `i`

You could avoid the explicit index by using a `Queue<String>` instead:

```java
public TreeNode deserialize(String data) {
    if (data.isEmpty()) return null;

    String[] nodesArray = data.split(",");
    Queue<String> nodes = new LinkedList<>(Arrays.asList(nodesArray));
```

```java

    TreeNode root = new TreeNode(Integer.parseInt(nodes.poll())); // poll() gets and removes first element
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty() && !nodes.isEmpty()) {
        TreeNode node = queue.poll();

        // Left child
        String leftVal = nodes.poll(); // Automatically advances to next element
        if (!leftVal.equals("null")) {
            node.left = new TreeNode(Integer.parseInt(leftVal));
            queue.offer(node.left);
        }

        // Right child
        if (!nodes.isEmpty()) {
            String rightVal = nodes.poll(); // Automatically advances to next element
            if (!rightVal.equals("null")) {
                node.right = new TreeNode(Integer.parseInt(rightVal));
                queue.offer(node.right);
            }
        }
    }
    return root;
}
```

This approach uses `nodes.poll()` instead of an index `i`, which automatically advances through the serialized values. Both approaches are valid, but the explicit index `i` is sometimes preferred for clarity or when you need more control over array access.

The key insight is that **we need some mechanism to track our position in the serialized data**, whether it's an explicit index `i` or using a queue that automatically advances.

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)

DFS return logic