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

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

**The encoded string should be as compact as possible.**

**Example 1:**

Input: root = [2,1,3]

Output: [2,1,3]

**Example 2:**

Input: root = []

Output: []

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* 0 <= Node.val <= 104
* The input tree is **guaranteed** to be a binary search tree.

**Attempt 1: 2022-10-29**

**Solution 1: Same as L297.Serialize and Deserialize Binary Tree, not use BST property,  preorder traversal for Serialize using DFS and for Deserialize using DFS (based on Queue) (10min)**

/\*\*

\* 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();

}

private void serializeHelper(TreeNode root, StringBuilder sb) {

if(root == null) {

sb.append(NA).append(spliter);

return;

}

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 deserializeHelper(q);

}

private TreeNode deserializeHelper(Queue<String> q) {

String rootVal = q.poll();

if(rootVal.equals(NA)) {

return null;

}

TreeNode root = new TreeNode(Integer.parseInt(rootVal));

root.left = deserializeHelper(q);

root.right = deserializeHelper(q);

return root;

}

}

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

// Codec ser = new Codec();

// Codec deser = new Codec();

// String tree = ser.serialize(root);

// TreeNode ans = deser.deserialize(tree);

// return ans;

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

Space Complexity: O(N)

**Solution 2: Same as L297.Serialize and Deserialize Binary Tree, not use BST property, level order traversal for Serialize using BFS (Queue) and for Deserialize using BFS (Queue) (10min)**

/\*\*

\* 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) {

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);

continue;

}

sb.append(node.val).append(spliter);

q.offer(node.left);

q.offer(node.right);

}

return sb.toString();

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

if(data == "") {

return null;

}

String[] values = data.split(spliter);

Queue<TreeNode> q = new LinkedList<TreeNode>();

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();

// String tree = ser.serialize(root);

// TreeNode ans = deser.deserialize(tree);

// return ans;

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

Space Complexity: O(N)

**Solution 3:  Use BST property (60min, too long to understand how to use BST attribute in deserialize function)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Codec {

// No need handling NULL with String NA = "X" like L297.Serialize and Deserialize Binary Tree,

// it will match the requirement 'The encoded string should be as compact as possible'

private String spliter = ",";

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

if(root == null) {

return "";

}

StringBuilder sb = new StringBuilder();

serializeHelper(root, sb);

return sb.toString();

}

private void serializeHelper(TreeNode root, StringBuilder sb) {

// No need handling NULL with sb.append(NA).append(spliter) like

// L297.Serialize and Deserialize Binary Tree, just return, it will match

// the requirement 'The encoded string should be as compact as possible'

if(root == null) {

return;

}

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) {

if(data == "") {

return null;

}

List<String> values = Arrays.asList(data.split(spliter));

Queue<String> q = new LinkedList<String>();

q.addAll(values);

return deserializeHelper(q, Integer.MIN\_VALUE, Integer.MAX\_VALUE);

}

private TreeNode deserializeHelper(Queue<String> q, int lower, int upper) {

if(q.isEmpty()) {

return null;

}

int currentRootVal = Integer.parseInt(q.peek());

// No need check 'currentRootVal < lower'

//if(currentRootVal < lower || currentRootVal > upper) {

if(currentRootVal > upper) {

return null;

}

q.poll();

TreeNode root = new TreeNode(currentRootVal);

root.left = deserializeHelper(q, lower, currentRootVal);

root.right = deserializeHelper(q, currentRootVal, upper);

return root;

}

}

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

// Codec ser = new Codec();

// Codec deser = new Codec();

// String tree = ser.serialize(root);

// TreeNode ans = deser.deserialize(tree);

// return ans;

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

Space Complexity: O(N)

**What's the difference between L449.Serialize and Deserialize BST with L297.Serialize and Deserialize Binary Tree ?**

**Refer to**

<https://leetcode.com/problems/serialize-and-deserialize-bst/discuss/177617/the-General-Solution-for-Serialize-and-Deserialize-BST-and-Serialize-and-Deserialize-BT/524978>

Q: Since BST is special case of Binary Tree, should not we use Binary Tree approach for Serializing and De-serializing both ?

A:  Because the problem says.. "The encoded string should be as compact as possible" . we could save the space used for saving "X" while encountering null in BST.

**Refer to**

<https://leetcode.com/problems/serialize-and-deserialize-bst/discuss/177617/the-General-Solution-for-Serialize-and-Deserialize-BST-and-Serialize-and-Deserialize-BT/214024>

Q: Can someone please explain why if (val < lower || val > upper) return null;is used?

A: That way you do not have to store null values, return null, when the range bound is out of lower or upper. It makes the string compact. Smart solution indeed. According to the Binary search rule, only the lower values than the root can be added to the left child of the tree. This is done by creating a boundary such that if the value > higher bound, do not add it to left child i.e. Val > upper=> return null condition

**Do we need to check** val < lower**?**

**Refer to**

<https://leetcode.com/problems/serialize-and-deserialize-bst/discuss/177617/the-General-Solution-for-Serialize-and-Deserialize-BST-and-Serialize-and-Deserialize-BT/733933>

I think if (val < lower || val > upper) return null; can be simplified into if (val > upper) return null;Because we are traversing the tree in preorder, each condition violation will only happen in upper bound.(You can think that in this algorithm, we will first fill all elements on the left. When the first violation happens, this value must satisfy val > upper.Then we will go back to the parent node. In the parent node's right child, the bound becomes [parent's val, upper\_bound].However, we know that we are in a BST tree, so all the remaining values in the queue must > parent's val.

Therefore, because all the remaining elements > parent's val and so on so forth. There is no need to maintain the lower bound.