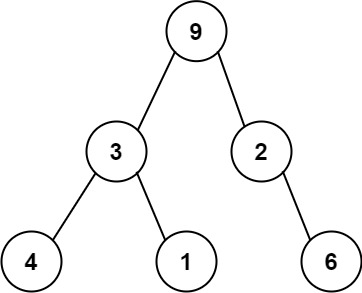
<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/>

One way to serialize a binary tree is to use **preorder traversal**. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as '#'.



For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where '#' represents a null node.

Given a string of comma-separated values preorder, return true if it is a correct preorder traversal serialization of a binary tree.

It is **guaranteed** that each comma-separated value in the string must be either an integer or a character '#' representing null pointer.

You may assume that the input format is always valid.

For example, it could never contain two consecutive commas, such as "1,,3".

**Note:**You are not allowed to reconstruct the tree.

**Example 1:**

Input: preorder = "9,3,4,#,#,1,#,#,2,#,6,#,#"

Output: true

**Example 2:**

Input: preorder = "1,#"

Output: false

**Example 3:**

Input: preorder = "9,#,#,1"

Output: false

**Constraints:**

1 <= preorder.length <= 10^4

preorder consist of integers in the range [0, 100] and '#' separated by commas ','.

**Attempt 1: 2023-06-19**

**Wrong Solution**

Input: "1,#,#,#,#"

java.util.EmptyStackException

  at line 101, java.base/java.util.Stack.peek

  at line 83, java.base/java.util.Stack.pop

  at line 9, Solution.isValidSerialization

  at line 54, \_\_DriverSolution\_\_.\_\_helper\_\_

  at line 84, \_\_Driver\_\_.main

=====================================================================================

class Solution {

    public boolean isValidSerialization(String preorder) {

        Stack<String> stack = new Stack<String>();

        String[] strs = preorder.split(",");

        for(int i = 0; i < strs.length; i++) {

            String cur = strs[i];

            while(cur.equals("#") && !stack.isEmpty() && stack.peek().equals("#")) {

                stack.pop(); // pop '#' at stack peek

                stack.pop(); // pop leaf node => this line encounter empty stack exception

            }

            stack.push(cur);

        }

        return stack.size() == 1 && stack.peek().equals("#");

    }

}

**Solution 1: Stack (60 min, very tricky, need remember)**

class Solution {

    public boolean isValidSerialization(String preorder) {

        Stack<String> stack = new Stack<String>();

        String[] strs = preorder.split(",");

        for(int i = 0; i < strs.length; i++) {

            String cur = strs[i];

            while(cur.equals("#") && !stack.isEmpty() && stack.peek().equals("#")) {

                stack.pop(); // pop '#' at stack peek

                // Test out by input: "1,#,#,#,#"

                if(stack.isEmpty()) {

                    return false;

                }

                stack.pop(); // pop leaf node

            }

            stack.push(cur);

        }

        return stack.size() == 1 && stack.peek().equals("#");

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Steps simulation**

"9,3,4,#,#,1,#,#,2,#,6,#,#"

on stack

-> 9,3,4,# (encounter 1st #, stack peek not #, push # on stack)

-> 9,3,# (encounter 2nd #, stack peek is #, pop 4,#, but also push current # back)

-> 9,3,#,1,# (encounter 3rd #, stack peek not #, push # on stack)

-> 9,3,#,# (encounter 4th #, stack peek is #, pop 1,#, but also push current # back)

-> 9,# (stack peek is #, pop #,3, but also push back one #)

-> 9,#,2,#,6,# (encounter 6th #, stack peek not #, push # on stack)

-> 9,#,2,#,# (encounter 7th #, stack peek is #, pop #,6, but also push current # back)

-> 9,#,# (stack peek is #, pop #,9, but also push back one #)

-> #

**Refer to**

<https://algo.monster/liteproblems/331>

**Problem Description**

The given problem revolves around the concept of serialization of a binary tree using preorder traversal. Preorder traversal means to visit the root node first, then the left subtree, and finally the right subtree. When serializing a tree, each non-null node is represented by its integer value, and null nodes are represented by a sentinel value '#'.

For instance, a binary tree may be serialized into a string representing the order in which nodes are visited. Notably, null child pointers are also serialized, which creates a unique representation for the structure of the original tree. The challenge here is to determine if a given string preorder, which contains comma-separated integer values and '#' symbols, is a valid serialization of a binary tree, without actually reconstructing the tree.

A valid preorder serialization must satisfy the requirements of a binary tree structure. Each node should have two children (which can be null), and the string should represent a tree where every node has been visited correctly in preorder, including null children.

**Intuition**

The intuition behind the solution is to simulate the traversal of a binary tree using a stack to keep track of the nodes being visited. As we iterate through the preorder string, each value can be seen as an action in this simulated traversal.

**When we hit a non-null node (an integer), we push it onto the stack since it represents a node that could have left and right children. Conversely, when encountering a null node (represented by '#'), we consider it as closing off a path in the tree (i.e., marking a leaf node).**

**However, a valid serialized binary tree cannot have two consecutive null nodes without them being the right and left children of some node before them. So if we have two null nodes on the top of the stack, there must be an integer just below them representing their parent node. We perform a check, and if that pattern is found, we remove (pop) the two nulls and the integer, simulating that we visited a complete subtree. We then replace them with a single null value on the stack to indicate that the entire subtree is now closed off and treated as a leaf.**

We repeat this process as we move through the string. **If the serialization is correct, we should end up with one single null value in the stack, which signifies the end of the traversal of a well-formed tree.** On the contrary, if we're left with a different pattern, the serialization is deemed incorrect.

**Solution Approach**

The provided Python code uses a stack to simulate the traversal of a binary tree during deserialization. The algorithm employs a for loop to iterate over nodes represented in the preorder serialization string split by commas. It pushes each value onto the stack, which not only represents inserting nodes but also helps to track the tree structure.

During this simulated traversal, the code looks for a specific pattern in the stack. This pattern comprises two consecutive '#' characters, representing null nodes or leaf nodes, followed by an integer, which would represent their parent node in a binary tree.

When this pattern is detected (stk[-1] == stk[-2] == "#" and stk[-3] != "#") it indicates that we've completed the visit to a subtree - specifically, the left and right children are both null, and we have their parent node just before these nulls.

At this point, the algorithm removes the three entries (stk = stk[:-3]) and replaces them with a single '#' to represent the whole subtree as a null or leaf for any higher-level nodes that might be present in the stack. This action effectively rolls up the null children into their parent, making it into a new leaf node.

Ultimately, if the preorder string represents a valid serialization of a binary tree, we'll end up with exactly one element in the stack after processing the entire input (len(stk) == 1). This remaining element must be the sentinel value '#' indicating that all nodes have been accounted for, and the full tree has been traversed (stk[0] == "#") without reconstructing it. If these conditions are met, the function returns True. Otherwise, if the stack does not adhere to this pattern, the function returns False, signaling that the given preorder serialization is not valid.

The solution is elegant as it simulates traversal without the overhead of tree construction and cleverly handles the serialization pattern-check using a stack that reflects the current state of the traversal process.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach.

Suppose we have a given preorder serialization string of a binary tree: preorder = "9,3,4,#,#,1,#,#,2,#,6,#,#". This serialization suggests that we have a tree with the root node value 9, a left child 3, which itself has a left child 4 that has no children (indicated by two consecutive # symbols), a right child 1 with no children, and finally a right child of the root, 2, which has a left child 6 with no children.

Initialize an empty stack stk.

Split the preorder serialization by commas and iterate through the values:

For the first value 9, push it onto stk: stk = [9].

The next value 3 goes onto the stack: stk = [9, 3].

Then, 4 is pushed: stk = [9, 3, 4].

A '#' is encountered, indicating a null child: stk = [9, 3, 4, '#'].

**Another '#' is encountered, so now we have two null children, which means 4 is a leaf node. We pop three times, and push a '#': stk = [9, 3, '#'].**

We encounter 1 and push it onto the stack: stk = [9, 3, '#', 1].

Again, two '#' symbols follow, indicating that 1 is a leaf node. Pop three and push '#': stk = [9, '#', '#'].

At this point, we have two '#' characters at the top of the stack, which suggests that the left and right children of 9 have been completely visited. We pop three and replace them with a '#': stk = ['#'].

Now, 2 enters the stack: stk = ['#', 2]. This is not correct as we have finished the tree rooted at 9 and should not add more nodes at the same level.

As we continue, we encounter 6 and the subsequent '#' symbols indicating its children, which after the process will result in a stk = ['#', '#' ], which is not a valid serialization as we are left with two sentinel values.

The correct result, in this case, should be the function returning False, which indicates that preorder = "9,3,4,#,#,1,#,#,2,#,6,#,#" is not a valid preorder serialization of a binary tree. If the serialization was valid, after processing all elements, the stack would have ended up with exactly one '#', reflecting the traversal of the entire tree.

**Solution Implementation**

class Solution {

// Method to validate if the preorder serialization of a binary tree is correct

public boolean isValidSerialization(String preorder) {

// Use a stack represented by a list to keep track of tree nodes.

List<String> stack = new ArrayList<>();

// Split the input string by commas to work with each node/leaf individually.

String[] nodes = preorder.split(",");

for (String node : nodes) {

// Push the current node onto the stack.

stack.add(node);

// Check the last three elements in the stack if they form a pattern of two

// consecutive '#' symbols which denote null children followed by a numeric node.

while (stack.size() >= 3 && stack.get(stack.size() - 1).equals("#")

&& stack.get(stack.size() - 2).equals("#") && !stack.get(stack.size() - 3).equals("#")) {

// Since the last two '#' symbols represent the null children of the previous numeric

// node, we can remove them all mimicking the null leaves falling off,

// which means they don't take up space in serialization.

stack.remove(stack.size() - 1); // Remove last null child

stack.remove(stack.size() - 1); // Remove second last null child

stack.remove(stack.size() - 1); // Remove parent node

// After removing a parent node and its two null children,

// we add one '#' to the stack to indicate that the subtree has been fully traversed.

stack.add("#");

}

}

// After processing all nodes, a valid serialization will end up with only one element in the stack,

// which must be '#', representing that all nodes have been matched with their children.

return stack.size() == 1 && stack.get(0).equals("#");

}

}

**Refer to**

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/solutions/1427004/python-simple-stack-explained/>

Similar to Problem **297**: Serialize and Deserialize Binary Tree, but here we do not really need to reconstruct our tree, and using stack is enough. The trick is to add elements one by one and when we see num, #, #, we replace it with #. If we get just one # in the end, return True, else: False. Let us look at the example 9,3,4,#,#,1,#,#,2,#,6,#,#. Let us go through steps:

We add elements until we have

9, 3, 4, #, #. It means now that

4 is leaf, so let us remove it: we have

9, 3, #.

Add elements, so we have

9, 3, #, 1, #, #. We have leaf

1, remove it:

9, 3, #, #. Now, we have

3 as leaf as well: remove it:

9, #.

Add elements

9, #, 2, #, 6, #, # ->

9, #, 2, #, # -> 9, #, # -> #.

**Complexity**

It is O(n) for time and O(h) for space, where h is the height of our binary tree.

**Code**

class Solution:

    def isValidSerialization(self, preorder):

        stack = []

        for elem in preorder.split(","):

            stack.append(elem)

            while len(stack) > 2 and stack[-2:] == ["#"]\*2 and stack[-3] != "#":

                stack.pop(-3)

                stack.pop(-2)

        return stack == ["#"]

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/solutions/78566/java-intuitive-22ms-solution-with-stack/>

See detailed comments below. Time complexity is O(n), space is also O(n) for the stack.

public class Solution {

    public boolean isValidSerialization(String preorder) {

        // using a stack, scan left to right

        // case 1: we see a number, just push it to the stack

        // case 2: we see #, check if the top of stack is also #

        // if so, pop #, pop the number in a while loop, until top of stack is not #

        // if not, push it to stack

        // in the end, check if stack size is 1, and stack top is #

        if (preorder == null) {

            return false;

        }

        Stack<String> st = new Stack<>();

        String[] strs = preorder.split(",");

        for (int pos = 0; pos < strs.length; pos++) {

            String curr = strs[pos];

            while (curr.equals("#") && !st.isEmpty() && st.peek().equals(curr)) {

                st.pop();

                if (st.isEmpty()) {

                    return false;

                }

                st.pop();

            }

            st.push(curr);

        }

        return st.size() == 1 && st.peek().equals("#");

    }

}

**Solution 2: Indegree and Outdegree (60 min, very tricky, need remember)**

class Solution {

    // All non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root, to uniform root node behavior, pre set 'diff = 1' which regarding root node

    // also has 1 indegree

    // All null node provides 0 outdegree and 1 indegree (0 child and 1 parent).

    public boolean isValidSerialization(String preorder) {

        String[] nodes = preorder.split(",");

        // Define 'diff = outdegree - indegree', the first(root)

        // node to start the whole thing. And it need 1 indegree,

        // so set the diff = 1 from the beginning to prepare minus

        // 1 because of root node consumes 1 indegree, equation as

        // 'diff(1) - 1 = outdegree - (indegree + 1)'

        int diff = 1;

        for(String node : nodes) {

            // As pre-define diff as 1, including root node, all nodes

            // can treat as decrease diff by 1 because of consuming 1

            // indegree

            diff -= 1;

            // If a serialization is correct, diff should never be negative

            if(diff < 0) {

                return false;

            }

            if(!node.equals("#")) {

                diff += 2;

            }

        }

        // If a serialization is correct, diff will be zero when finished.

        return diff == 0;

    }

}

**Refer to**

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/solutions/78551/7-lines-easy-java-solution/>

Some used stack. Some used the depth of a stack. Here I use a different perspective. In a binary tree, if we consider null as leaves, then

all non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root

all null node provides 0 outdegree and 1 indegree (0 child and 1 parent).

Suppose we try to build this tree. During building, we record the difference between out degree and in degree diff = outdegree - indegree. When the next node comes, we then decrease diff by 1, because the node provides an in degree. If the node is not null, we increase diff by 2,because it provides two out degrees. If a serialization is correct, diff should never be negative and diff will be zero when finished.

public boolean isValidSerialization(String preorder) {

    String[] nodes = preorder.split(",");

    int diff = 1;

    for (String node: nodes) {

        if (--diff < 0) return false;

        if (!node.equals("#")) diff += 2;

    }

    return diff == 0;

}

**Refer to**

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/solutions/78552/java-counting-indegree-and-outdegree-simple-clear/>

**Why total degree should never exceed 0 ?**

Since this is a preorder serialization, degrees are calculated in a top-down fashion, and, tree is a structure that each node has only one indegree and at most two outdegree. Positive degree means there are more indegree than outdegree, which violates the definition.

public boolean isValidSerialization(String preorder) {

    String[] strs = preorder.split(",");

    int degree = -1;        // root has no indegree, for compensate init with -1

    for (String str: strs) {

        degree++;            // all nodes have 1 indegree (root compensated)

        if (degree > 0) {    // total degree should never exceeds 0

            return false;

        }

        if (!str.equals("#")) {// only non-leaf node has 2 outdegree

            degree -= 2;

        }

    }

    return degree == 0;

}

**Solution 3: Full Tree leaves node number = non-leaves node number + 1 (60 min, very tricky, need remember)**

class Solution {

    public boolean isValidSerialization(String preorder) {

        int nonLeaves = 0;

        int leaves = 0;

        String[] nodes = preorder.split(",");

        int i = 0;

        for(; i < nodes.length; i++) {

            // The serialization is valid if and only if the prefix

            // is the entire sequence, which means when not condition

            // "nonLeaves + 1 == leaves" reached before go through the

            // whole string, it cannot convert to a binary tree

            // Test case: "#,#,3,5,#"

            if(nonLeaves + 1 == leaves) {

                return false;

            }

            if(nodes[i].equals("#")) {

                leaves++;

            } else {

                nonLeaves++;

            }

        }

        return nonLeaves + 1 == leaves && i == nodes.length;

    }

}

**Refer to**

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/solutions/78551/7-lines-easy-java-solution/comments/83320>

If we treat null's as leaves, then the binary tree will always be **full**. A full binary tree has a good property that # of leaves = # of nonleaves + 1. Since we are given a pre-order serialization, we just need to find the **shortest** prefix of the serialization sequence satisfying the property above. If such prefix does not exist, then the serialization is definitely invalid; otherwise, the serialization is valid **if and only if** the prefix is the **entire** sequence.

// Java Code

public boolean isValidSerialization(String preorder) {

    int nonleaves = 0, leaves = 0, i = 0;

    String[] nodes = preorder.split(",");

    for (i=0; i<nodes.length && nonleaves + 1 != leaves; i++)

        if (nodes[i].equals("#")) leaves++;

        else nonleaves++;

    return nonleaves + 1 == leaves && i == nodes.length;

}

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/solutions/78551/7-lines-easy-java-solution/comments/286305>

another perspective of this problem is - for any binary tree/subtree, **the # of non-null nodes = # of null nodes - 1** we can still use the same idea, counting difference.

if node not null, diff++

if node is null, diff--, when we see diff == 0, it means we should have finished traversing the tree. if there are more nodes after then it's invalid.

  public boolean isValidSerialization1(String preorder) {

        String[] nodes = preorder.split(",");

        int diff = 1, i = 0;

        for (; i < nodes.length; i++) {

            if (nodes[i].equals("#")) {

                --diff;

                if (diff == 0) break;

            } else {

                ++diff;

            }

        }

        return diff == 0 && i == nodes.length - 1 && nodes[i].equals("#");

    }

**Refer to**

[L297.Serialize and Deserialize Binary Tree (Ref.L449)](note://BF7420C763B944DA841E1B945D602EFC)