Recursion

Binary Search Tree

Problem Description

Stack

Tree

The challenge is to determine whether a given array of unique integers represents the correct preorder traversal of a Binary Search Tree (BST). Preorder traversal of a BST means visiting the nodes in the order: root, left, right. To qualify as a valid BST traversal, the sequence must reflect the BST's structure, where for any node, all the values in the left subtree are less than the node's value, and all the values in the right subtree are greater.

Binary Tree

Monotonic Stack

Intuition

Medium

greater than the root. The provided solution uses a stack and a variable that keeps track of the last node value we've visited so far when moving towards the right in the tree. Starting with the first value in the preorder traversal, which would be the root of the BST, we consider the

For a preorder traversal of a binary search tree, the order of elements would reflect the root being visited first, then the left subtree,

and then the right subtree. In a BST, the left subtree contains nodes that are less than the root, and the right subtree contains nodes

following: 1. BST Property: As we iterate through the preorder list, if we encounter a value that is less than the last visited node when turning right, we know that the tree's sequential structure is incorrect.

- 2. Preorder Traversal Structure: A stack is used to keep track of the traversal path; when we go down left we push onto the stack, and when we turn right we pop from the stack. The top of the stack always contains the parent node we would have to compare
- 3. Switching from Left to Right: Each time we pop from the stack, we update the last value, which symbolizes that all future node values should be larger than this if we are considering the right subtree now.
- 4. Finally, if we complete the iteration without finding any contradictions to the properties above, we return True indicating that the sequence can indeed be the preorder traversal of a BST.
- Throughout the stack manipulation, we are implicitly maintaining the invariant of the BST that nodes to the left are smaller than root, and nodes to the right are greater.

Solution Approach The implementation of the solution follows a straightforward algorithm that keeps track of the necessary properties of a BST during

1. Initialization: A stack stk is created to keep track of ancestors as we traverse the preorder array. A variable last is initialized to

the traversal sequence:

class Solution:

9

11

with when going right.

negative infinity, representing the minimum constraint of the current subtree node values. 2. **Traversal**: We iterate through each value x in the preorder array.

last value after turning right in the tree, which violates the BST property, hence we return False. 4. Updating Stack: While the stack is not empty and the last element (top of the stack) is less than the current value x, we are

3. Validity Check: For every x, we check if x < last. If this condition is met, it means we have encountered a value smaller than the

moving from the left subtree to the right subtree. We pop values from the stack as we are effectively traveling up the tree, and

Here is the highlighted implementation of the approach:

stk.append(x)

stack. The stack now looks like [5].

stack. So, the final stack is [6].

4. The preorder is possible for a BST, so we return True.

right. Throughout every step, the BST property was maintained properly.

Initialize the last processed value to negative infinity

If the current value is less than the last processed value, the

last processed value and pop from the stack. This means we are

backtracking to a node which is a parent of the previous nodes

If we have completed the loop without returning False, the sequence

sequence does not satisfy the binary search tree property

While there are values in the stack and the last value

in the stack is less than the current value, update the

Iterate over each node value in the preorder sequence

is a valid preorder traversal of a binary search tree

return True

def verifyPreorder(self, preorder: List[int]) -> bool:

- update the last value. The last value now becomes the right boundary for all the subsequent nodes in the subtree. 5. Processing Current Value: After performing the necessary pops (if any), we append the current value x to the stack. This push
- operation represents the traversal down the left subtree. 6. Termination: If the entire preorder traversal is processed without returning False, it implies that the sequence did not violate any BST properties, and we return True.
- subtree property holds at every step. The simplicity of the solution comes from understanding how the preorder traversal works in conjunction with the BST properties.

The algorithm uses a stack to model the ancestors in the preorder traversal and a variable last to ensure that the BST's right

stk = []# Stack for keeping track of the ancestor nodes # Variable to hold the last popped node value as a bound last = -inffor x in preorder: # If the current node's value is less than last, it violates the BST rule if x < last:</pre> return False while stk and stk[-1] < x: last = stk.pop() # Update 'last' each time we turn right (encounter a greater value than the top of the stack)

If all nodes processed without violating BST properties, the sequence is valid

Push the current value onto the stack

```
This approach effectively simulates the traversal of a BST while ensuring that both the left and right subtree properties hold
throughout the process.
Example Walkthrough
Let's consider a small example to illustrate the solution approach with the following preorder traversal list [5, 2, 1, 3, 6].
```

1. Initialize the stack stk to an empty list, and last to negative infinity, which will represent the minimum value of the current

2. Traverse the preorder list: a. Begin with the first element x = 5. Since x is not less than last (which is -inf at this point), we continue and push 5 onto the

b. Next element x = 2. It is not less than last, so we push 2 onto the stack. The stack becomes [5, 2].

c. For element x = 1, again x is not less than last, so we push 1 to the stack. The stack is now [5, 2, 1].

subtree's nodes.

popping because the top of the stack (2) is still less than 3, update last to 2, and finally push 3 to the stack. Now, the stack is [5, 3].

e. For the last element x = 6, the top of the stack is 3, which is less than x, so we pop 3 from the stack and the new last is now

3. The stack is [5] and since 5 is less than x, we pop again and update last to 5. Now the stack is empty, and we push 6 to the

d. Now x = 3. It's greater than the top of the stack (which is 1), so we pop 1 from the stack, and update last to 1. We continue

3. Since we have placed all elements onto the stack according to the rules, and have never encountered an element less than last during the process, the iteration completes successfully.

The pattern here shows that the stack is used to maintain a path of ancestors while iterating, and last serves as a lower bound for

the nodes that can come after turning right, ensuring that subsequent nodes are always greater than any node's value after turning

- **Python Solution**
 - from math import inf # Import 'inf' to use for comparison class Solution: def verify_preorder(self, preorder: List[int]) -> bool: # Initialize an empty stack to keep track of the nodes

```
last_processed_value = stack.pop()
24
25
               # Push the current value to the stack to represent the path taken
26
                stack.append(value)
27
```

return True

return true;

stack = []

10

12

13

14

15 16

17

18

19

20

21

22

23

28

29

30

27

28

29

31

30 }

C++ Solution

last_processed_value = -inf

return False

if value < last_processed_value:</pre>

while stack and stack[-1] < value:

for value in preorder:

```
31
Java Solution
   class Solution {
       public boolean verifyPreorder(int[] preorder) {
           // Stack to keep track of the ancestors in the traversal
           Deque<Integer> stack = new ArrayDeque<>();
           // The last processed node value from the traversal
           int lastProcessedValue = Integer.MIN_VALUE;
           // Iterate over each value in the preorder sequence
9
10
           for (int value : preorder) {
               // If we find a value less than the last processed value, the sequence is not a valid preorder traversal
11
               if (value < lastProcessedValue) {</pre>
12
13
                    return false;
14
15
16
               // Pop elements from the stack until the current value is greater than the stack's top value.
17
               // This ensures ancestors are properly processed for the BST structure.
               while (!stack.isEmpty() && stack.peek() < value) {</pre>
18
                    // Update the last processed value with the last ancestor for future comparisons
20
                    lastProcessedValue = stack.pop();
21
22
23
               // Push the current value to the stack, as it is now the current node being processed
24
               stack.push(value);
26
```

// If the entire sequence is processed without any violations of BST properties, return true

```
// A stack to hold the nodes
           stack<int> nodeStack;
           // Variable to store the last value that was popped from the stack
           int lastPopped = INT_MIN;
           // Iterate over each element in the given preorder vector
           for (int value : preorder) {
               // If current value is less than the last popped value, the preorder sequence is invalid
               if (value < lastPopped) return false;</pre>
18
19
20
               // While the stack isn't empty and the top element is less than the current value,
               // pop from the stack and update the lastPopped value.
21
               while (!nodeStack.empty() && nodeStack.top() < value) {</pre>
                    lastPopped = nodeStack.top();
24
                    nodeStack.pop();
25
26
27
               // Push the current value onto the stack
28
               nodeStack.push(value);
29
30
31
           // If the loop completes without returning false, the preorder sequence is valid
32
           return true;
33
34 };
35
Typescript Solution
 1 // Required for the typing generosity of TypeScript
 2 // Integer array type alias
   type IntArray = number[];
   // Stack type alias leveraging Array type for stack operations
   type Stack = number[];
   // Initialize a stack to hold the nodes
   let nodeStack: Stack = [];
```

25 // While the stack isn't empty and the top element is less than the current value, // pop from the stack and update the lastPopped value. 26 while (nodeStack.length > 0 && nodeStack[nodeStack.length - 1] < value) {</pre> 27 28 lastPopped = nodeStack.pop()!; 29

return true;

nodeStack = [];

// Variable to store the last value that was popped from the stack

// Reset stack and last popped for each validation run

// Iterate over each element in the given preorder array

// Function to verify if a given array of integers is a valid preorder traversal of a BST

// If the loop completes without returning false, the preorder sequence is valid

// Since TypeScript doesn't have IntArray and Stack as inbuilt types, we're creating aliases

// If current value is less than the last popped value, the preorder sequence is invalid

let lastPopped: number = Number.MIN_SAFE_INTEGER;

lastPopped = Number.MIN_SAFE_INTEGER;

for (let value of preorder)

nodeStack.push(value);

function verifyPreorder(preorder: IntArray): boolean {

if (value < lastPopped) return false;</pre>

// Push the current value onto the stack

10

13

16

17

19

20

21

23

24

30

31

32

33

34

35

36

38

37 }

1 #include <vector> 2 #include <stack> #include <climits> // for INT_MIN class Solution { 6 public: // Function to verify if a given vector of integers is a valid preorder traversal of a BST bool verifyPreorder(vector<int>& preorder) { 11 12 13 14 15 16

// to enhance readability and maintain a level of abstraction in our code, similar to the original C++ version. 41 Time and Space Complexity The time complexity of the given code is O(n), where n is the number of nodes in the preorder traversal list. This is because the code iterates through each element of the preorder list exactly once. During each iteration, both the stack push (stk.append(x)) and pop (stk.pop()) operations are executed at most once per element, which are 0(1) operations, therefore not increasing the overall time complexity beyond O(n).

The space complexity of the code is O(n), due to the stack stk that is used to store elements. In the worst-case scenario, the stack could store all the elements of the preorder traversal if they appear in strictly increasing order. However, due to the nature of binary search trees, if the input is a valid BST preorder traversal, the space complexity can be reduced on average, but in the worst case, it still remains O(n).