Documentation for BSTIterator Class

- The BSTIterator class is designed to simulate an in-order traversal of a Binary Search Tree (BST) using an iterator. It provides methods to efficiently traverse the tree in a sorted order (left-root-right) with an average time complexity of O(1) per operation and O(h) space complexity, where h is the height of the tree.

Key Features:

1. **In-Order Traversal:** The iterator allows for in-order traversal of the BST, which means nodes are visited in ascending order.

2. **Efficiency:** The next() and hasNext() methods are optimized to run in average O(1) time complexity.

3. **Space Complexity:** The iterator uses O(h) memory, where h is the height of the BST, to maintain the state of the traversal.

Class Initialization

Constructor (BSTIterator(TreeNode root)):

- Initializes the iterator with the root node of the BST.
- Internally, it prepares the stack by pushing all the left children of the root onto the stack. This preparation ensures that the smallest element of the BST is at the top of the stack, ready to be accessed by the first call to next().

Methods

1. next() Method:

- ***Description:*** Returns the next smallest element in the in-order traversal of the BST.
- Functionality:
  - Pops the top element from the stack, which represents the current smallest unvisited node.
  - If this node has a right child, the method pushes the right child and all its left descendants onto the stack. This ensures that the next call to next() retrieves the next smallest element.
- ***Time Complexity:*** Average O(1). In the worst-case scenario (when traversing the height of the tree to push nodes), it can be O(h). However, amortized over all operations, the time complexity remains O(1).

2. hasNext() Method:

- ***Description:*** Checks if there are more elements to traverse in the BST.
- ***Functionality:***
  - Returns True if the stack is not empty, indicating that there are more nodes to be visited in the in-order traversal.
  - Returns False if the stack is empty, indicating that the traversal is complete.
- ***Time Complexity:*** O(1), as it only checks the length of the stack.

Example Usage

1. Initialization:

- A BSTIterator object is created using the root of a BST.
- ***Example:*** BSTIterator bSTIterator = new BSTIterator(root);
- This prepares the iterator for traversal by initializing the internal stack with all leftmost nodes from the root.

2. **Traversing the BST:**
   - ***next():***
     - Each call to next() moves the iterator to the next smallest element and returns that element.
     - *Example:* bSTIterator.next(); // returns the next smallest element
   - ***hasNext():***
     - Before each call to next(), hasNext() can be used to check if there are more elements to traverse.
     - *Example:* bSTIterator.hasNext(); // returns True or False depending on the state of the traversal

Internal Working

- The BSTIterator maintains a stack to simulate the recursive in-order traversal of a BST.
- When initialized, the constructor pushes all leftmost nodes of the root to the stack.
- The next() method then pops the top node (the current smallest element), processes it, and pushes all left descendants of the node's right child if one exists.
- The stack ensures that the next smallest node is always at the top, providing an efficient mechanism for in-order traversal.

Constraints

- **Number of Nodes:** The number of nodes in the BST is in the range $[1, 10^5]$.
- **Node Values:** The value of each node is within the range $[0, 10^6]$.
- **Call Limits:** The methods next() and hasNext() are called at most 10^5 times.

Follow-Up Considerations

- The BSTIterator is designed to achieve average O(1) time complexity for next() and hasNext() while maintaining O(h) space complexity.
- This design ensures efficient traversal of even large BSTs without excessive memory usage.

Conclusion

- The BSTIterator class provides an efficient and easy-to-use interface for in-order traversal of a BST. By leveraging a stack-based approach, it maintains both time and space efficiency, making it well-suited for scenarios involving large binary search trees and frequent traversal operations.