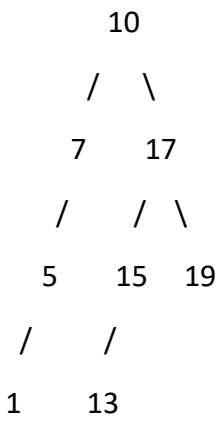
Find the Inorder Predecessor and Successor

CodeStudio

You are given a binary search tree (BST), and your task is to create a program that finds the inorder predecessor and successor for a given key in the tree. The inorder predecessor is the node with the largest value that is smaller than the given key, and the inorder successor is the node with the smallest value that is larger than the given key.

Example:



TargetNode = 13

Output:

The Inorder Predecessor of 13 is: 10

The Inorder Successor of 13 is: 15

Approach 1: Function to find the in-order predecessor and successor of a key in a BST using Inorder Traversal

- **Function Purpose:** This approach is used to find the inorder predecessor and successor of a given key in a BST.
- **Explanation:**
    1. The function performs an inorder traversal of the tree and stores the result in the 'inorderAns' vector.
    2. It then iterates through the vector to find the predecessor (the largest value less than the key) and the successor (the smallest value greater than the key).
- **Time Complexity: The time complexity of this approach is O(N), where N is the number of nodes in the tree, as it needs to visit all nodes.**
- **Space Complexity: The space complexity is O(N) due to the vector used to store the inorder traversal values.**

Approach 2: Optimized function to find the in-order predecessor and successor of a key in a BST

- **Function Purpose:** This approach is used to find the inorder predecessor and successor of a given key in a BST.
- **Explanation:**
    1. The function uses an optimized approach without storing the entire inorder traversal.
    2. It starts at the root and iteratively traverses the tree to find the predecessor and successor.
    3. The predecessor is updated while moving to the right in the tree, and the successor is updated while moving to the left.
- **Time Complexity: The time complexity of this approach is O(H), where H is the height of the tree. In the worst case, it's O(N) for a skewed tree.**
- **Space Complexity: The space complexity is O(1) as it doesn't use any additional data structures to store the traversal.**

Conclusion:

- Approach 1 uses an inorder traversal to find the predecessor and successor, and it's straightforward to implement.
- Approach 2 provides an optimized solution with constant space complexity, making it more efficient for large trees.
- Both approaches accurately find the predecessor and successor for a given key in a BST.
- The choice between the two approaches depends on the specific requirements and characteristics of the tree being processed.