Introduction

This is my overview of the implementation details for my Red-Black Tree data structure. It includes explanations of the algorithms used and the rationale behind design choices, and performance analysis.

- Implementation Details
  - Overview of Red-Black Tree Properties
    - A Red-Black Tree is a self-balancing binary search tree with the following key properties:
      - Each node is either red or black.
      - The root is always black.
      - All leaves (null nodes) are black.
      - Red nodes cannot have red children (no two consecutive red nodes).
      - Every path from a node to its descendant NIL nodes has the same number of black nodes.
  - TreeNode Class
    - The TreeNode class represents each node in the Red-Black Tree. Each node contains:
    - Data value
    - Pointers to its parent, left child, and right child
    - A color attribute (red or black)
  - RedBlackTree Class
    - The RedBlackTree class manages the Red-Black Tree and includes methods for insertion, deletion, searching, and maintaining tree properties.
  - Constructor
    - The constructor initializes the tree with a root node and creates a nullNode to represent NIL nodes, ensuring all leaves are black.
  - Insertion
    - The insertion method adds a new node to the tree while maintaining Red-Black properties through a series of rotations and color changes. New nodes are initially inserted as red. If necessary, the insertFixup method adjusts the tree to preserve its properties.
  - Deletion

        - The deletion method removes a node from the tree. If the removed node was black, the deleteFixup method is called to maintain the tree's properties through rotations and color adjustments.
    - Search
        - The search method finds a node with a given value, traversing the tree from the root to the leaves. If the node is found, it is returned; otherwise, the search continues until a NIL node is reached.
    - Rotations
        - Tree rotations (left and right) are essential for maintaining the Red-Black Tree properties during insertion and deletion. Rotations rearrange the nodes and update their parent-child relationships without violating the binary search tree properties.
- Performance Analysis
    - The Red-Black Tree ensures balanced operations, leading to efficient performance in terms of insertion, deletion, and search.
- Insertion and Deletion Runtime
    - Red-Black Tree: The insertion and deletion operations have a worst-case time complexity of O(log n), where n is the number of nodes in the tree. This efficiency is due to the tree's ability to maintain a balanced structure, ensuring that the height of the tree remains logarithmic relative to the number of nodes.
- Binary Search Tree (BST): In a regular binary search tree, the time complexity for insertion and deletion can degrade to O(n) in the worst case. This occurs when the tree becomes unbalanced, essentially forming a linked list. In such a scenario, each operation might need to traverse all nodes.

Comparison with Binary Search Tree

Insertion:

Red-Black Tree: O(log n)

Binary Search Tree: O(log n) on average, O(n) in the worst case

Deletion:

Red-Black Tree: O(log n)

Binary Search Tree: O(log n) on average, O(n) in the worst case

The Red-Black Tree's ability to remain balanced through rotations and color changes makes it significantly more efficient for insertion and deletion operations compared to an unbalanced binary search tree. This ensures predictable and optimal performance, making Red-Black Trees suitable for applications requiring frequent insertions and deletions.

Conclusion

This Red-Black Tree implementation provides a balanced binary search tree with efficient insertion, deletion, and search operations. The use of rotations and color adjustments ensures that the tree remains balanced, providing good performance for a wide range of applications. The performance analysis highlights the advantages of Red-Black Trees over regular binary search trees, particularly in maintaining optimal time complexity for key operations.