Project Overview

This project implements an ordered symbol table using a red–black binary search tree (BST). The red–black tree is a balanced BST that ensures logarithmic performance for search, insertion, and deletion operations. The tree maintains its balance through a set of rules enforced by helper methods that perform rotations and color flips. Alongside the core Java implementation, there is also a Python version provided for testing the same public interface in a simpler, sorted-list based structure.

Core Functionality and Public Methods

1. Initialization and Basic Properties

- **Constructor (RedBlackBST)**
  Creates an empty red–black tree ready for operations. This method sets up the internal state but does not insert any key–value pairs.
- **isEmpty()**
  Checks whether the symbol table is empty. It returns a boolean value indicating if the tree has any key–value pairs.
- **size()**
  Returns the total number of key–value pairs stored in the tree. It uses an internal count that reflects the current number of nodes.

2. Search and Retrieval Methods

- **get(Key key)**
  Retrieves the value associated with a given key. The method performs a standard binary search by comparing the target key to keys in the tree and navigating left or right accordingly. If the key is not found, it returns a null value.
- **contains(Key key)**
  Determines whether a specific key exists in the tree. It uses the get method internally to check for the key’s presence.

3. Insertion and Update

- **put(Key key, Value val)**
  Inserts a new key–value pair into the symbol table or updates the value for an existing key.

When a key already exists, its associated value is overwritten. If the provided value is null, this method instead removes the key from the tree. After insertion, the tree rebalances itself to maintain the red–black properties by performing necessary rotations and color flips.

4. Deletion Methods

- **delete(Key key)**
  Removes a specified key (and its associated value) from the tree. It first checks if the key exists and then deletes it while ensuring the tree remains balanced. The deletion process involves navigating the tree, adjusting links, and potentially rebalancing through rotations and color flips.

- **deleteMin()**
  Removes the smallest key (the minimum element) from the tree. Before removal, it may temporarily change the color of the root if both children are black, then performs a series of moves and rebalancing to correctly delete the minimum key and maintain the tree structure.

- **deleteMax()**
  Removes the largest key (the maximum element) from the tree. Similar to deleteMin(), it makes adjustments to the root's color and rebalances the tree after removing the largest element.

5. Order-Related Operations

- **min() and max()**
  These methods return the smallest and largest keys in the symbol table, respectively. They work by navigating as far left (for min) or as far right (for max) from the root as possible.

- **floor(Key key)**
  Finds the largest key in the tree that is less than or equal to a given key. This is useful for range queries and for cases where an exact match might not exist.

- **ceiling(Key key)**
  Returns the smallest key in the tree that is greater than or equal to a given key, which is complementary to the floor operation.

- **select(int rank)**
  Returns the key that has exactly a specified number of keys smaller than it. In other words, it retrieves the key at the given order (rank) in the sorted order of keys.

- **rank(Key key)**
  Determines the number of keys in the tree that are strictly less than the given key. This gives an idea of where a particular key falls in the overall sorted order.

- **keys()**
  Provides an iterable collection of all keys in the tree, arranged in ascending order. This allows for easy iteration over the entire set of keys.

- **keys(Key lo, Key hi)**
  Returns all keys that fall within a specified range (inclusive of the provided endpoints). This method is particularly useful for range queries.

- **size(Key lo, Key hi)**
  Counts the number of keys that lie within a given range. It builds on the keys in the specified range and returns their total count.

6. Utility Methods

- **height()**
  Returns the height of the tree, which is a measure of the longest path from the root to a leaf node. This is primarily used for debugging or analyzing the tree's structure.

- **check()** (Internal)
  Although not public, this method runs a series of integrity checks on the tree. It ensures that the tree follows the binary search tree order, that subtree sizes are consistent, and that the red–black tree properties (like no red right links and balanced black heights) are maintained.

- **Internal Helper Methods (rotateLeft, rotateRight, flipColors, moveRedLeft, moveRedRight, balance)**
  These methods are used internally during insertion and deletion to maintain the tree's red–black properties.

  - **rotateLeft and rotateRight:** These perform tree rotations to correct any right-leaning or unbalanced red links.
  - **flipColors:** This method changes the colors of a node and its children to enforce the red–black tree invariants.
  - **moveRedLeft and moveRedRight:** Used during deletion, these methods prepare a node so that one of its children can be safely removed.
  - **balance:** After insertion or deletion, this method ensures that the tree remains balanced by applying rotations and color flips as needed.

Testing

The project includes comprehensive testing of every public method. In the Java version, a separate test class uses if–else conditions to verify that each method behaves as expected. The tests cover:

- Basic operations on an empty tree (checking emptiness and size).
- Insertion of key–value pairs and updating the size.
- Retrieval of values and checking for key existence.
- Order-related operations (finding minimum, maximum, floor, ceiling, select, and rank).
- Range queries through keys and size range methods.
- Structural integrity through the height method.
- Deletion methods, including deletion of the smallest, largest, and a specific middle key.

Similarly, the Python version offers a parallel set of tests that simulate the same public interface. Although the Python version does not implement a self–balancing tree, it uses a sorted list to replicate ordered behavior and validate each method's functionality.