Name: Varun Pius Rodrigues

UFID: 61965993

GatorLink: varunpius@ufl.edu

Advanced Data Structures

Project: Red Black Tree Implementation

Number of Classes: 4

bbst.java:

This is the main class. This includes only the main method. This method is responsible for picking the arguments, which would be the filename and commands and would execute it, by calling the appropriate methods. The program is invoked by calling the *make* command. The makefile contains all the necessary values. This would compile the file and generate the necessary .java files for executing. Once that is done, we call the program as follows:

java bbst filename

The program would pick the file named filename and would execute it. Following this, we start giving the commands necessary. The program would execute the necessary commands appropriately.

Node.java:

This class is used to create an object of type Node. It contains the basic Node properties constituting a tree, namely:

- ID
- Counter
- Color
- Left Node pointer
- Right Node Pointer
- Parent Node Pointer

Every input ID-counter pair is mapped into a Node and this node will be used to build a tree. The Node class contains 2 constructors:

Default Constructor: Will be used to construct a node of 0 value as ID and Counter and null pointers to the left, right and parent node.

Parameterized Constructor: This will be used to construct a node with the ID and Counter value. We will assign it color accordingly later in the Red Black tree methods.

TreeFunc.java:

This class contains methods dealing with generic Tree functions. Converting an input array to binary tree, and displaying the tree are some of the functions.

Methods:

Node sortedArrayToBST: This will convert the input array received to a Binary Search Tree. The general idea will be to use the middle element of the array as the root. All nodes to the left of the root will be lesser than the root and all nodes to the right of the root will be greater than the

root. Since the input data sorted beforehand, this method works as selecting the middle node will result in a balanced tree. This process in repeated recursively in the respective halves till the tree is complete.

void setParent: this is used to update every node with its parent. Every node will have its parent node pointer set to null prior to calling this method. After this method is called, all nodes will have their parent nodes except the root, which will still have its parent node set to null.

void preorder: This method is used for preorder traversal of the tree.

void inOrder: This method is used for inorder traversal of tree.

int height: this method is used to determine the number of levels in the given tree.

void printGivenLevel: this method is used to print all nodes at the given height/level.

void printLevelorder: this method is used to print all the nodes in Level order fashion.

void convertRedNode: This method is used to convert the given binary tree into a Red Black Tree. The easiest way to do this is convert all nodes in the last level to red. In this way, if one of the subtree has a level more than the other, we will still have equal number of black nodes on the way from root to exterior nodes.

RedBlackFunc.java:

This class implements all the functions required for the Red Black Tree operations. These methods form the core of the project. All the primary commands to be executed in the project are implemented here.

Methods:

void rotateLeft(Node): This is used to rotate the node to the left. This is important because we have to preserve the Red-Black property in the Red-Black Tree. Every insertion/deletion disrupts this and hence, we rectify this using the rotate methods.

void rotateRight(Node): This is used to rotate the node to the right. This is important because we have to preserve the Red-Black property in the Red-Black Tree. Every insertion/deletion disrupts this and hence, we rectify this using the rotate methods.

Node minNode(Node): This is used to find the value of the minimum node with respect to the current node. Usually the minimum node will be the leftmost node from the current node, as we follow the left nodes.

Node maxNode(Node): This is used to find the value of the maximum node with respect to the current node. Usually the maximum node will be the rightmost node from the current node, as we follow the right nodes.

Node insert(Node): This is used to insert a new incoming node. Inserting a new node effectively disturbs the Red-Black property of the tree. We need to remedy this. This is done through insertFixUp.

void insertFixUp(Node z): We need to now rectify the situation when insertion takes place. This is done through the insertFixUp. Here we will perform a series of operation to preserve the Red-Black property.

void delete(Node): this is used to delete the node. Deleting the node will result in disturbance to the Red-Black property of the tree. We will solve this through deleteFixUp. Also, we will come across situations when the nodes necessary for deleteFixUp will be null. We will remedy this with the help of transplant.

void transplant(Node, Node): Here, we take in two parameters as input and preserves the Red-Black property.

void deleteFixUp(Node): We need to now rectify the situation when deletion takes place. This is done through the DeleteFixUp. Here we will perform a series of operation to preserve the Red-Black property.

Node searchNode(int): This is used to identify the node whose ID has been specified. Again, we also perform additional operation here to identify the nearest/closest ID node, which will help us in the *next* and *previous* operation.

int countRange(Node x, int low, int high): this method is used to return the sum of all counters for IDs within the range mentioned.

Node modifyCount(int nos, int ID, char): Here we perform the operation for the increase and decrease. We will evaluate the flag, and identify whether this is reduce or increase operation. Also, when further condition are need to be evaluated like less than zero conditions and accordingly delete the node, or check if node exists and if not then insert the node in case of increase operation takes place here.

Node next(int): This takes the ID as input and will find the node which will be a successor to this ID. Sometimes, it may happen the mentioned ID may not be present in the tree. In this case, we need to return the node that has ID next bigger than the ID mentioned. In case the node with given ID exists, we call the successor method and return the successor, else we will perform few more case check and return the node with next biggest ID.

Node successor(Node): It will find the successor to the node given as input.

Node previous(int): This takes the ID as input and will find the node, which will be a predecessor to this ID. Sometimes, it may happen the mentioned ID may not be present in the tree. In this case, we need to return the node that has ID next smaller than the ID mentioned. In case the node with given ID exists, we call the predecessor method and return the predecessor, else we will perform few more case check and return the node with next smallest ID.

Node predecessor(Node): It will find the predecessor to the node given as input.

Tests:

Dataset	**Time taken to perform:**	**Result:**
10^7 ID-Counter pairs	25 seconds	Pass
10^8 ID-Counter pairs	5 minutes	Pass

Tests performed on:

Compiler : Java JDK 8

- Windows 10
    - javac
    - IntelliJ
- Linux (Ubuntu)
    - Javac

Conclusion:

The tree is being constructed in O(n). The increase and the reduce operation takes O(log n). Count operation takes O(n) time. InRange takes O(log n + s), where s is the number of IDs in the range. Next and previous both takes O(log n) time.

References:

- Dr. Sahni's Lectures and Presentations for Advanced Data Structures
- Introduction to Algorithms by T. Cormen and C. Leiserson