Project Report

Ninad Mundalik
UFID 55386969

Project Title: COP 5536 Event Counter Using RBT

1) Environment Information:
   Language Used : Java (JDK1.7.0_79)
   Operating System: Windows 10 machine with 4gb ram
   IDE Used: Eclipse Mars

2) Code Structure:
   The structure of Node of red black tree is given as follows.

```
Class Node{
        Node left, right, parent;
        Color color;
        int ID;
        int count;
        boolean isNull;
}.
```

   We make use of a special Null node that just stores the color and parent information of a leaf.

   We use the enum Color to store the color of the nodes and an enum Arguments to make the code of choosing commands to run easier to read. The important functions, algorithms and prototypes are given in the section below.

3) Important Functions: The description of important functions are given as below.

i) O(n) Input:
   This is achieved by reading the event id and count pairs in an array and then recursively dividing the array in 2 parts using middle index. This index value will recursively be set as the left and right roots of subtrees of the current root. This will give us a balanced BST with all black nodes. After which we color the nodes RED that are at the maximum level. Required prototypes are given below.

   private static void initializeEventCounter(String[] args) :
   Reading of the file and creation of the sorted EventPair that has ID and count.

   private Node sortedArrayToBST(List<EventPair> allEvents, **int** start, **int** end, Node root1):
   Conversion into BST recursively by making max value in left to be root's left child and max value in right to be roots' right child respectively.

   private void findDeepestNodes(Node root, **int** level, Object[] levelInformation):

Find the nodes at the maximum level and populate list to be colored using change color function given below to make nodes Red.

private void changeColor(List<Node> listToColor, Color red):
Change color to red.

ii) Accepting and interpreting commands:
This is done with the help of the functions showMenu() which accepts commands in the required format. validateAndReturnType() is a function that returns which command we want to execute and validates if required number of arguments are present or not.

private void showMenu():
Accept Commands.

private Arguments validateAndReturnType(String[] arguments):
Validate if right number of arguments are given for the right commands.

iii) Count(*theID*):
Count is performed with the help of Search logic of a Binary search tree. We first look for the node with *theID* and if it is present, we print the output to the standard output window or print 0 if not found. The function prototypes involved are given below:

private void count(String[] arguments)
Print count of node if it exists

private Node searchID(Node root, **int** ID):
Normal BST recursive search.

iv) Inrange(ID1, ID2):
We make use of a recursive call and keep looking in the left and right subtrees to check for nodes that satisfy the conditions id1 <= id of node >= id2. We add the count to a counter when we come across such a node. This process is supported with the help of 2 functions whose prototypes are as given below. inRange just invokes the call by verifying the arguments whereas countInRange actually computes the InRange Count.

private void inRange(String[] arguments)
Print the total count of nodes in range

private void countInRange(Node curr, **int** IDLow, **int** IDHigh):
Calculate count of nodes found in the given range recursively.

v) Next(theID): We find the inorder successor to find the next node greater than *theID.* The algorithm for finding the successor is if right subtree is not null find

smallest node in right subtree or start from the root and recursively look for the next ID greater than theID. This is achieved with the help of 2 functions next() and findSuccessor where again next just validates the arguments and calls the findSuccessor function. The prototypes are as given below.

private void next(String[] arguments)
Print next node if available after validating arguments

private void findSuccessor(Node root, **int** ID)
Find the inorder successor of the given ID.

vi) Previous(theID): Finding the previous node is similar to finding the inorder successor. Difference in the algorithm is that we find the biggest node in the left subtree if its not empty otherwise we keep looking recursively in either of the subtrees. The functions involved in this are as follows.

private void previous(String[] arguments):
Print the previous node and its count if it exists or 0 0 after validating arguments.

private void findPredecessor(Node root, **int** ID)
Find in-order predecessor of the given node recursively.

vii) Increase(theID, m):
Increase first involves finding the node as in BST. This is done by the function searchID(ID). If the required node is found we just add m to its count and print the output. If the node is not found we have to insert it in the right position as a red node and then adjust the properties of the red black tree to make it valid. This involves making necessary rotations left or right and color changes if any. The functions are explained as below.

private void increase(String[] arguments): Accepts the argument and serves as the starting point for increase operation. Calls the searchID function and makes decision of whether to insert node or not.

private Node insertNode(Node parent, Node curr, EventPair eventPair): This function checks if the root is null. If so it creates a black node and the process ends there. If not it creates a red node and checks if we are inserting in the left child or not. If we went to the left side we have to check if current node is left of the parent. This will create a LL case which we tackle using a Right rotate. If a LR case is present we do a right rotate and a left rotate at the parent along with some color changes which can be found in the code and report below. We also similarly perform operations for RR case and RL case. One more case that is handled is that of a Red sibling when we change the color of red sibling and curr to black and change the parent's color to Red propagating the problem upwards.

private void rotateLeft(Node curr, **boolean** colorChangeNeeded): Rotate left centred at Node curr and change color of curr and its parent if colorChangeNeeded is true.

Private void rotateRight(Node curr, Boolean colorChangeNeeded): Rotate right centred at the Node curr and change color of curr and its parent if colorChangeNeeded is true.

viii) Reduce(theID, m):
Reduce also proceeds by finding the node with the given ID and reduces the count of the events by m. If however the value of count reduces to less than or equal to 0 it is removed using the delete routine which is described with the help of the prototypes below.

private void reduce(String[] arguments): Main reduce functions that decided whether node has to deleted or not.

private Node delete(Node curr, **int** ID): Called with root and the ID which has to be deleted.

private void delete(Node curr, **int** ID, AtomicReference<Node> rootRef): Main delete function which first checks if node to be deleted is a degree 1 or 0 node. If not it finds the inorder successor in the right subtree and replaces its ID and count with the new value. Plus it then deletes the inorder successor using the case for deleting node of degree 0 or 1.

private void deleteDegreeOneChild(Node curr, AtomicReference<Node> rootRef)
This handles the case of a degree with 0 or 1 null leaf. If red node is to be deleted it can be safely deleted as it does not disturb the red black property. If the color of the node to be deleted is black and it has a red child we replace red child as node curr and change its color to black.

All of the below cases are used to change the tree structure by identifying which node is one black node deficient after deletion and adjust it accordingly.

private void adjustRootDeficiency(Node deficientNode, AtomicReference<Node> rootRef):
If the root is deficient by one black node we set current node to be new root in root reference. Since it is the root that is deficient no more adjustments need to be done.

private void adjustDeficientRedSiblingNode(Node deficientNode, AtomicReference<Node> rootRef):
If the deficient node has a red sibling with 2 black nodes, we do a right or a left rotate depending on if the sibling is left child or right child. This is not sufficient to make it a valid red black tree so we check for further case as described below.

private void adjustDeficientBlackSiblingNode(Node deficientNode, AtomicReference<Node> rootRef):
If the deficient Node has a black parent, black sibling with both children black we change sibling's color to become red and push the problem further up and start by checking from the case when root is deficient or not.

private void adjustRedParentNode(Node deficientNode, AtomicReference<Node> rootRef):
If the parent of the deficient node is red we exchange the color of parent and sibling to maintain red black tree properties.

private void adjustRedChildBlackSibling (Node deficientNode, AtomicReference<Node> rootRef):
If one of the sibling of the deficient node has a red node we do a rotation around sibling depending on if it's a right child or left child.

private void adjustBlackSiblingRedChild(Node deficientNode, AtomicReference<Node> rootRef):
If deficient node has a black sibling and one red node we do a left or right rotation centred at the parent of deficient node. The parent takes the color of its parent. Original parent which is now the left node becomes black and the initial red child of sibling becomes black.

4) Results **:**
Tested the program on following input files :
test_100.txt, test_1000000.txt, test_10000000.txt
All methods as implemented in required time complexity.