COP 5536
Spring 2016
Advanced Data Structures

Programming Project Report

Submitted by:

Shantanu Godbole

UFID: 48104395

Problem Statement: To develop an event counter which stores the ID of an event along with a count variable denoting the count of events associated with that ID.

Problem Specifications : Based on an input of [ID count] pairs, build a Red-black tree that accommodates all the above IDs in O(n) time complexity. Further, enable the event counter to perform the following operations on the tree:

Command	**Desctiption**	**Time complexity**
Increase(*theID*, *m*)	Increase the count of *the event theID* by *m*. If *theID* is not present, insert it. Print the count of *theID* after the addition.	$O(\log n)$
Reduce(*theID*, *m)*	Decrease the count of *theID* by *m*. If *theID*'s count becomes less than or equal to *0*, remove theID from the counter. Print the count of *theID* after the deletion, or 0 if theID is removed or not present.	$O(\log n)$
Count(*theID*)	Print the count *of theID*. If not present, print 0.	$O(\log n)$
InRange(*ID1*, *ID2*)	Print the total count for *ID*s between *ID1* and *ID2* inclusively. Note, *ID1* ≤ *ID2*	$O(\log n + s)$ where s is the number of *ID*s in the range.
Next(*theID*)	Print the *ID* and the *count* of the event with the lowest *ID* that is greater that *theID*. Print "0 0", if there is no next *ID*.	$O(\log n)$
Previous*(theID)*	Print the *ID* and the *count* of the event with the greatest key that is less that *theID*. Print "0 0", if there is no previous *ID*.	$O(\log n)$

Compiler used/ Language: Java Programming Language Compiler or javac . A limitation faced because of the platform used is that while running the program for **10^8** input pairs, the program needs to be run with increased heap space, i.e., JVM arguments **–Xmx8g** to complete within 2 minutes.

Application Structure:

Classes: There are three classes that handle all the operations required to build the event counter under the given specifications. **RBTreeNode.java** specifies a node and its properties and which can be inserted in a Red-Black tree. **RBTree.java** is the class which handles all the operations internally on the red-black tree by taking inputs from **bbst.java**, which acts as the main trigger class and an interface between the user and the tree. The 3 classes have been described in depth below:

- RBTreeNode.java – Used to define basic functions and attributes required for any node to be included in a Red-Black Tree, i.e., color, left, right etc. The skeletal structure of the above class, i.e., fields, methods etc. is as follows:

- Fields
  - `private int id; // Id of the event`
  - `private int color; // Color of node (0 for red and 1 for black)`
  - `private int count; // Count of the event particular to id`
  - `private RBTreeNode left; // Reference to the left child`
  - `private RBTreeNode right; // Reference to the right child`
  - `private RBTreeNode parent;// Reference to the parent(null for root)`
  - `private int leftRight;// Field showing whether the current node is left or right (0 for right,1 for left and -1 for root)`
- Methods
  - Constructors
    - `public RBTreeNode(int id, int count) {} // Constructor with two arguments; by default sets the color as red and parent as null`
    - `public RBTreeNode() {}  // default constructor`
  - Getter/Setter methods
    - `public RBTreeNode getLeft() {}// gets left child`
    - `public void setLeft(RBTreeNode left) {}//sets left child`
    - `public RBTreeNode getRight() {}//gets right child`
    - `public void setRight(RBTreeNode right) {}//sets right child`
    - `public RBTreeNode getParent() {}//gets parent`
    - `public void setParent(RBTreeNode parent, int lr) {}//sets parent`
    - `public int getColor() {} // gets color of node(0/1)`
    - `public void setColor(int color) {} // sets color of node`
    - `public int getId() {}   // gets ID of the event`
    - `public void setId(int id) {}  // sets ID of the event`
    - `public int getCount() {} // gets count of the event counter`
    - `public void setCount(int count) {} // sets count of the event counter`

- `public void setChild(RBTreeNode node, int i) {} // set the given node as child of the current node; left child if i = 0 else right child`

- `public int getLeftRIght() {} // gets the leftRight field for the current node; 0 for left, 1 for right and -1 for parent`

- `public void setLeftRight(int lr) {} // sets the leftRight field for the current node`

- `public boolean isEqualTo(RBTreeNode test) {} // customized isEqualTo function that checks whether the id of the two nodes is equal or not`

- **RBTreeNode.java**
  - Fields:
    - `private RBTreeNode root;`
    - `private int smallest;`
    - `private int largest;`
  - Methods:
    - Constructors:
      - `public RBTree() {}`
    - Getters and Setters:
      - `private RBTreeNode getSibling(RBTreeNode node, RBTreeNode parent) {}// gets sibling of the current node`
      - `private int getColor(RBTreeNode node) {} // gets color of the given node; for null node it returns black`
      - `private void setParent(RBTreeNode node1, RBTreeNode node2, int lr) {} // sets node2 as a parent of node1`
      - `private void setLeft(RBTreeNode node1, RBTreeNode node2) {}// sets node2 as the left child of node1`
      - `private void setRight(RBTreeNode node1, RBTreeNode node2) {}// sets node2 as right child of node1`
      - `private int getCountInRange(RBTreeNode root2, int i, int j) {} // gets accumulated count of nodes within the given range; searches recursively for nodes within the given range`
    - Build/Insert/Rebalance methods:
      - `public RBTreeNode insertAsList(int[] arr, int start, int end, int maxHeight) {} // inserts the given list of numbers in a red black tree`

- `private RBTreeNode buildTreeFromList(int[] arr, int start, int end, int maxHeight) {} // builds a red-black tree using the given input array`

- `public void insert(RBTreeNode node) {} // inserts a particular node in the red-black tree`

- `private void balanceInsert(RBTreeNode root2, int colorCheck) {} // identify and manage the type of imbalance occurred in the tree ,i.e., RR,LL,LR,RL etc. after insertion and accordingly trigger rotate.`

❖ Delete/Rebalance methods
- `public void delete(RBTreeNode node){} // deletes the given node from the tree if found.`

- `private RBTreeNode bstDelete(RBTreeNode root2) {} // internally does the standard Binary Search Tree deletion operation if the node is found.`

- `private void balanceDelete(RBTreeNode parent, RBTreeNode dbbNode) {} // identifies and manages the type of imbalance in tree caused by double black node after deletion. Ultimately triggers rotate() to balance the tree.`

❖ Common methods
- `private void rotate(RBTreeNode root2, int colorCheck, String op) {} // makes the required rotation operations based on the type of imbalance received either after insertion/deletion.`

- `private RBTreeNode search(RBTreeNode root, int i, String op) {} // searches for a node with the value i in the tree. For insert operation, it simply finds the blank location where a new node with the given value i can be inserted.`

❖ Permissible operations on the tree by outside user

- `public void Increase(int i, int j) {}`
- `public void Reduce(int i, int j) {}`
- `public void Count(int i) {}`
- `public void InRange(int i, int j) {}`
- `public void Next(int i) {}`
- `public void Previous(int i) {}`

- **bbst.java**
  This class contains only 3 methods namely, main, eventOps and LogBase
  The main method forms the triggering domain of the application
  eventOps reads commands from the given commands.txt and accordingly triggers the respective operation on the tree.
  LogBase function is a helper function used to find the max height of the tree.

UML Diagrams Explanation: The following are the UML diagrams for the three classes and their interactions.