Mickey Vellukunnel, UFID: 9413-9616
COP 5536 Programming Project Report
March 22, 2016

Event Counter using Red-Black Tree

Language used: Java
java version "1.8.0_65" | “1.8.0_74”
Java(TM) SE Runtime Environment (build 1.8.0_65-b17) | (build 1.8.0_74-b02)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode) | (build 25.74-b02, mixed mode)

Compiler: javac “1.8.0_65” | “1.8.0_74”

NOTE: Since this project was written using the Java language and runs with the heavy JVM, while running huge test files of the order of ~ 1GB, the program should be run with the max increase heap option set to to an appropriate heap size (8GB heap for 1GB testfiles).
i.e. java -Xmx8000m bbst test_file

Project Structure:

This project consists of two main source files.

1. bbst.java
2. RedBlackTree.java

1. **bbst.java**

This is the test file of the project which contains the main function and expects as argument to the program “bbst” a **test-input** file. This way, it supports redirected input from a file "file-name" which contains the initial sorted list.

The command line for this mode is as follows Java:
$java bbst file-name

test file's Input format:

n
ID1 count1
ID2 count2
.
.
.
IDn countn

Here the assumption is that IDi < IDi+1 where IDi and counti are positive integers and the total count fits in 4-byte integer limits.

After the input is read from the source file, we get into the **interactive part** of the program.

This then will read the commands from the standard input stream and print the output to the standard output stream.

The command and the arguments should be separated by a space, not parenthesis or commas (i.e "inrange 3 5" instead of "InRange(3, 5)"). At the end of each command, there should be an EOL character.

For each command, the specified output will be printed to the standard output stream. An EOL character will be printed at the end of each command's output.

To exit from the program, use "**quit**" command.

To run the program, simply issue the 'make' command, and then use the generated 'bbst' executable.

2. **RedBlackTree.java**

This is the actual source of the project. It contains an **augmented** Red Black tree implementation to support the Event counter operations.

A. Each node in the RedBlack tree represents an Event in the event counter and is encapsulated within the "TreeNode" class.
   Other than the standard fields in this node, I've also introduced the "subTreeCount" variable inside it, i.e. total count of all events' counts in the subtree rooted at this node. This is an augmented order statistic variable which helps to support the **inRange** operation in **O(log n)** time. This variable is updated in constant time whenever an operation which changes the tree structure

relevant to this node (i.e. the subtree below it) changes, i.e. whenever an insert, delete, increase, initialization or reduce operation takes place.
Definition: subtreeCount = leftChild.subtreeCount + rightChild.subtreeCount + this.count

B. We can initialize the RedBlack tree with a constructor that accepts an ascending sorted array of n TreeNodes (Events). This is done in in O(n) time by calling a recursive function sortedArrayToRBBST().

RedBlackTree(TreeNode arrOfTreeNodesInAscendingSortedOrder[], int totalNumberOfNodesInSortedArray);

sortedArrayToRBBST() is a recursive function which takes a sorted list and builds a RedBlack tree from it by recursively splitting the list into two at the middle, and assigning key, left and right based on the splits. This takes linear time as it visits each node only once. Also, the RedBlack tree property is maintained by coloring all the internal nodes at the last level red. This is done by checking if the current level is at the max level for that tree (comparing against log of the total number of nodes). Maintain the value of the augmented variable subTreeCount as well. Time complexity: O(n).

TreeNode **sortedArrayToRBBST**(TreeNode arr[], int start, int end, int currentHeight, int maxHeightOfTreeWithNNodes);

C. Event counter commands (/functions). The following are a list of the Event counter specific functions. These are called from the main executable program (bbst) directly if the user enters the specific commands related to them.

- Increase() increases the count of the event theID by m. If theID is not present, inserts it. Prints the count of theID after the addition. Maintains the value of the augmented variable subTreeCount as well. Time complexity: O(log n).

void **increase**(int theIDofEvent, int countIncreaseBy);

- Reduce() decreases the count of theID by m. If theID's count becomes less than or equal to 0, removes theID from the counter. Prints the count of theID after the deletion, or 0 if theID is removed or not present. Maintains the value of the augmented variable subTreeCount as well. Time complexity: O(log n).

void **reduce**(int theIDofEvent, int decreaseCountBy);

- Count() Prints the count of theID. If not present, prints 0. Time complexity: O(log n).

void **count**(int theIDofEvent);

- Next() Prints the ID and the count of the event with the lowest ID that is greater that theID. Prints “0 0”, if there is no next ID. Time complexity: O(log n).

TreeNode **next**(int theIDofEvent, boolean shouldPrint);

- Previous() prints the ID and the count of the event with the greatest key that is less that theID. Prints “0 0”, if there is no previous ID. Time complexity: O(log n).

TreeNode **previous**(int theIDofEvent, boolean shouldPrint);

- InRange() Prints the total count for IDs between ID1 and ID2 inclusively. Note, ID1 ≤ ID2. Time complexity **desired**: O(log n + s) where s is the number of IDs in the range. NOTE: Since we're using and maintaining an order statistic augmented variable "**subTreeCount**" in each Event node in this RedBlackTree implementation, this query in fact only takes **O(log n)** time to complete, regardless of the number of IDs in the specified range (’s').

void **inRange**(int ID1, int ID2);

D. Red Black Tree specific functions. The following are a list of functions specific to the Red Black tree implementation.

- Binary search tree insert. Time complexity: O(log n). Maintain the value of the augmented variable "subtreeCount" in constant time as we go down the tree and place the newly inserted node. Also, call the insert1() function to check if the newly inserted node satisfies the RedBlack tree properties, and fix if not.

void **insert**(int key, int count);

- The insert<i> methods (where 1<=i<=5) handles different cases of the RedBlack Tree insert scenarios. They all take constant time to run. Time complexity: O(1). If required, they’ll call the rotate functions to fix the structure of the tree to be balanced.

void **insert<i>**(TreeNode node);

- Binary search tree delete after finding the node with the given key. Time complexity: O(log n).
void **delete**(int key);

- The delete<i> methods (where 1<=i<=6) handles different cases of the RedBlack Tree delete scenarios. They all take constant time to run. Time complexity: O(1).

void **delete**<i>(TreeNode nodeN);

- deleteFix1() is called if the node to be deleted is black with ONE child, and the child is red, simply repaint the child black. This is a trivial case of the delete operation which is checked first.

boolean **deleteFix1**(TreeNode node);

- The left and right rotate functions are used to fix an unbalanced tree which is leaning towards the left or right during an insertion operation or a deletion. This takes constant time. Also we need to fix the augmented variable subTreeCount on rotate. Time complexity: O(1).

void **leftRotate**(TreeNode node);
void **rightRotate**(TreeNode node);

- The following functions all returns some relatives of the required nodes, i.e.
Successor() (Returns the successor of the node, i.e. the left-most child in it's right subtree),
predecessor() (Returns the predecessor of the node, i.e. the right-most child in it's left subtree),
grandparent() (Returns the grandparent of a node if available, i.e. parent of its parent),
uncle() (Returns the uncle of a node if available, i.e. sibling of its parent),
sibling() (Returns the sibling of a node if available, i.e. other child of its parent).
Successor and predecessor functions have Time complexity: O(log n). Grandparent, uncle and sibling functions have Time complexity: O(1)

TreeNode **successor**(TreeNode node);
TreeNode **predecessor**(TreeNode node);
TreeNode **grandparent**(TreeNode node);
TreeNode **uncle**(TreeNode node);
TreeNode **sibling**(TreeNode node);

E. Utility functions. These are functions which are called by the main Red Black tree or Commands functions.

- Return the node in the tree which fits the leftEnd of the inRange query. This could be the event which has ID equal to the specified leftRange or the smallest event ID greater than it.

TreeNode **getRangeLeftNode**(int ID1);

- Return the node in the tree which fits the rightEnd of the inRange query. This could be the event which has ID equal to the specified rightRange or the greatest event ID less than it.

TreeNode **getRangeRightNode**(int ID2);

- Return the Event node which is the smallest common ancestor in the tree for the given two event IDs. Used for the inRange queries.

TreeNode **leastCommonAncestor**(int leftID, int rightID);

- Create a temporary NULL sentinel node and attach it to the parent node for re-balancing purposes while deletion. This should be deleted after the complete re-balance process it complete. This is used because there is no explicit NULL leaf sentinel nodes in this implementation of the Red Black Tree. For any Event node which has child as null, it's implied to be a null sentinel leaf of color "BLACK" (isRed= false).

TreeNode **getNullLeaf**(TreeNode parent, boolean onRight);

- Removes the null leaf from the RBT and removes its parent's references to it. To be called after the rebalancing has been performed and the added helper null sentinel leaf node is no longer required.

void **cleanIfNullLeaf**(TreeNode node);

- findMix/Max. Find and return the ID of the event with the minimum/maximum ID. Time complexity: O(log n)

int **findMin**();
int **findMax**();

- Returns the subtreeCount of the node if it exists, if not, return zero.

int **getSubtreeEventCount**(TreeNode node);

- Find node with the given ID, if not found (or if tree is empty), return null. Time complexity: O(log n).

TreeNode **findNode**(int ID);

- Deletes the given node. Also fixes RedBlackTree violations if any and calls the delete2() method if further fixes are required. Time complexity: O(log n).
void **deleteNode**(TreeNode node);

- "Deletes" a node by removing all references to it and setting its parent reference to it as null. Only happens to a leaf node.

void **deleteNodeReferences**(TreeNode node);

- Copy the contents (event ID and count) of one node to another.

void **replaceNode**(TreeNode replaceeNode, TreeNode replacerNode);

- Find the log base 2 of a number using integer arithmetic.

public static int **log2**(int n);