COP 5536 Spring 2016

Advanced Data Structures

Dr. Sartaj Sahni

Project

Event Counter using Red Black Tree

Submitted By:

Name: Pramit Dutta

Email: pdutta04@ufl.edu

UFID: 7513-2433

In this project we implement an Event Counter using Red Black Tree. A Red Black Tree is basically a binary search tree with an added feature, the colour. The Colour of a node in a Red Black Tree can either be RED or BLACK. Red Black Tree ensures that no path is twice as long as any other (maintaining black height) by constraining the node colours and thereby maintains the tree is balanced. Each tree node has attributes colour, value, left, right and parent. If a child or the parent of a node does not exist, the corresponding pointer has value NULL. In this case we use a pair of ID and Count as the Node's Value. Each Tree Node represents an Event which has an ID and a Count.

Compiler Information:

Program was compiled and tested successfully on thunder.cise.ufl.edu and storm.cise.ufl.edu and on local machine with the following compiler information given below –

java version "1.8.0_60"

Java(TM) SE Runtime Environment (build 1.8.0_60-b27)

Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)

Program Structure:

The Program starts by reading inputs from the system using Scanner.

Reads the filename and then reads the number of event inputs and then stores the event details in two arrays **arr_ID** and **arr_Count**.

Global variables are maintained to store the Root, Nil/Null Nodes and the total number of inputs.

A TreeNode class is used with attributes `String color; int ID; int count;`
and parent, leftChild and rightChild(which are also of TreeNode types)

After the arrays are initialized then the Red Black Tree is built in **O(n)** time by taking the middle element of the arrays as the root of the BST and then recursively calling the left part of the array and the right part of the array to build the left and right subtrees.

After the Red Black Tree is built initially then we perform Tree Operations to Insert a Node, Delete a Node, Left Rotate a Subtree, Right Rotate a Subtree and other function to fix the tree properties and get it balanced. The helper functions like Minimum_Tree, Maximum_Tree, transplant, Successor and Predecessor aids in the functions to fix and balance the Tree and perform Insert and Delete operations. The find function searches for a Node with a given ID in the RB tree.

After the Tree is built we then perform our own defined operations increase, reduce, count, inrange, next and previous on the RB Tree and update counts, delete or insert nodes accordingly as per the rules.

The output of the Commands for these operations then redirects into an output stream and writes into an output file.

Function Prototypes:

1. **static TreeNode sortedArrayToBST(int[] arr_ID, int[] arr_Count,int start, int end, TreeNode parent,int current_level, int max_level**) – returns the root of the BST after building the Binary Search Tree(Red Black Tree) in O(n) time.

2. **static TreeNode insert(TreeNode newNode)** -- inserts the new Node into the Red Black Tree and returns the new Node

3. **static void insert_Fix(TreeNode newNode)** -- after insertion operation balances the tree and restores the properties of the Red Black Tree.

4. **static void RightRotate(TreeNode node)** – function that right rotates a subtree across a given node

5. **static void LeftRotate(TreeNode node)** – function that left rotates a subtree across a given node

6. **static TreeNode Minimum_Tree(TreeNode node)** – function that returns the node with the minimum value in a tree/subtree

7. **static TreeNode Maximum_Tree(TreeNode node)** – function that returns the node with the maximum value in a tree/subtree

8. **static void delete(TreeNode node)** – deletes the given node from the Red Black Tree

9. **static void delete_Fix(TreeNode node)** – after deletion operation balances the tree and restores the properties of the red black tree

10. **static void transPlant(TreeNode a, TreeNode b)** -- replaces one subtree as a child of its parent with another subtree

11. **static TreeNode predecessor(TreeNode node)** – returns the predecessor of a given node.

12. **static TreeNode successor(TreeNode node)** – returns the successor of a given node.

13. **static TreeNode find(int ID)** – returns the tree node with a given ID

14. **static void inorderTraversal(TreeNode node)** – traverses the Tree in order, Can be used to print the tree.

15. **static TreeNode increase(int ID, int count)** - Increases the count of the event the ID by m. If the ID is not present, insert it. Returns the Node.

16. **static TreeNode reduce(int ID, int count)** - Decrease the count of the ID by m. If the ID's count becomes less than or equal to 0, remove the ID from the counter. Returns the Node.

17. **static TreeNode count(int ID)** – Finds the Node with the given ID and returns it.

18. **static TreeNode next(int ID)** – Returns the node that is next/ just after to the Node with the given ID

19. **static TreeNode previous(int ID)** – Returns the node that is previous/ just before to the Node with the given ID

20. **static int inRange_helper(TreeNode n, int ID1, int ID2)** – helper function for in range count calculation between ID1 and ID2.

21. **static void inRange(int ID1, int ID2)** – Calculates the Count of all event's count that fall between the range of event IDs.

[int h =(int) Math.ceil(Math.log(n+1)/Math.log(2)) - calculates the height of the bbst**]**

References:

1. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

2. **Introduction to Algorithms 3rd Edition - Thomas H. Cormen, Charles E. Leiserson, R**