EVENT COUNTER

Using a red-black tree

Final project of the course “Advanced Data Structures, COP 5536”

Abhishek Chawla
abhichawla@ufl.edu
0812-1437

A. Program Structure:

The program implements an event counter using a Red-Black tree, in which each node of the tree corresponds to an event and keeps the count and ID of that event. Since the underlying structure is a Red-Black tree, self-balancing is performed after insertion and deletion. Instead of using the NULL values, I have used a sentinel (dummy) 'Null Node'; one of the main reasons to do so is to be able to represent a "Null node" as a 'Black' node, conveniently.

The primary algorithms of RB Tree deletion, insertion, re-balancing, and rotations are based on the pseudo-code given in the book "Introduction to Algorithms", by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. To acknowledge this, I have not used my regular notation of variables in the definition of these methods (instead of using pNode to denote a pointer to a node, I have stick to the single letter variables used in the book).

The Program begins with creating a simple Binary Tree satisfying properties of an RB Tree. The contents are read from the file provided in the command-line arguments.

It then reads the commands from the Standard Input and executes respective functions. It quits after the command 'quit' is encountered.

Output of these executions is displayed on the Standard Output.

I have implemented the project in C++, using the Microsoft Visual Studio 2012 IDE for development.

Each RB Tree Node, 'TreeNode' contains following objects:

1. TreeNode* left – pointer to the left child of the node.
2. TreeNode* right - pointer to the left child of the node.
3. TreeNode* parent - pointer to the parent of the node.
4. NodeColor color – color of the node.
5. int count – count of the event represented by the node.
6. int val – key of the node.

The main class, 'RBTree', contains all the methods and variables relevant to an RB Tree. Following is a list of primary methods of RB Tree, such as insertion, deletion, etc.:

1. TreeNode* GetRoot() - return the root of the tree.
2. bool isNull(TreeNode* pNode) – returns true if a node is the sentinel null node, else false.
3. void RotateLeft(TreeNode *x) - performs a right rotation on the given node. This method is used in rebalcing.
4. void RotateRight(TreeNode *x) – performs a left rotation on the given node. This method is used in rebalcing.
5. void Insert(int key, int count) – creates a new node having key and inserts it into the RB tree. Also performs rebalancing after insertion.
6. void Insert_FixUp(TreeNode *z) – called by 'Insert' method to perform rebalancing of Rb Tree after node insertion.
7. void Remove(int key) – removes a node from the RB Tree. Also performs rebalcing after removal.
8. void Remove_FixUp(TreeNode* pNode) – called by 'Remove' method to perform rebalancing of RB Tree after node removal.

9. TreeNode* TreeMinimun(TreeNode* pRoot) – returns the node with the smallest key in the tree rooted at 'pRoot'.
10. TreeNode* TreeSuccessor(TreeNode* pNode) – returns the node with the smallest key, greater than the key of 'pNode'.
11. TreeNode* FindNode(int key) – finds the node having 'key' key. Returns the sentinel null node if not found.

Following is the list of methods the project description asks to implement:

12. void Increase(int ID, int count)
13. void Reduce(int ID, int count)
14. void Count(int ID)
15. void InRange(int ID1, int ID2)
16. void Next(int ID)
17. TreeNode* Previous(int ID)

B. <u>Acknowledgments:</u>

Following sources have been used to implement this project:

1. Chapter 13 (Red-Black Trees) of Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree