BLG 335E – Analysis of Algorithms I

Homework 3

Red-Black Tree

150170092 : Barış İncesu

11 JANUARY 2020

Part 2. Report

2.1 Complexity

Whether we do the **insertion or search** function in Red-Black Trees, we should not forget that the tree we make is always balanced. we can guarantee an upper bound of ***O(logn)*** for these operations. The height of a Red-Black tree is always ***O(logn)*** where ***n*** is the number of nodes in the tree. Since the tree is always in balance, there is no change in the upper bound value in **worst case or average case**.

2.2 RBT vs BST

The most general definition that can be said for a binary search tree (BST) and a red-black tree is that both have the property of a binary search tree. The feature that distinguishes the two from each other is to always achieve a balanced structure with the extra operations performed in the red-black tree. In unstable one-way operations that may occur in BST in this way, the worst case, like a chain, search and add operations will be ***O(n),*** while the Red-black tree will always be in equilibrium, so a worst case cannot be mentioned and the result will always be ***O(logn).*** In general terms, Red black tree can also be called stabilized form of BST.

2.3 Augmenting Data Structures

I focused on the Augmenting Data Structures section in the title to implement the given situation. A search function can be written for each location, considering these researches and the structure of the tree we have, to be formed in a balanced way according to the name.

It is important to know the number of players in the appropriate position on the right and left wing of the tree, as the second and third ranking player is requested from us. Since the left wing is always bigger than the right wing, we can get as far as the i we are looking for by adding the players on the left.

```
sizePG(root->left) function:
if root->position == “PG”
then n ++;
sizePG(root->left)
sizePG(root->right)

and returned n + 1. // one for includes itself

after know the number of how many PG player in left wing of given node
we can compare position and nodes number can reach wanted player.

Search(pt,i) // return pointer to node containing the i th smallest
key of the subtree rooted at pt.

r <- sizePG(root->left)
if i = r
then return pt
elseif i < r
then return Search( pt->left, i )
else return Search( pt->right, i – r )

In this model, PG can be replaced by SG, SF, PF, C clone 4 more size
and search functions can be written.
```