Enes Bedirhan Dikmen

150180016

BLG 335E – Analysis of Algorithms I Fall2020 Homework 3

Part 1.

My code can be compiled with:

g++ -std=c++11 150180016.cpp -o 150180016

And can be run with:

./150180016 euroleague.csv

Part 2.

Complexity

Insertion

In binary search trees since we move one layer deeper with each recursion worst case time complexity is O(height of the tree). Red Black Trees are balanced trees which means before creating another layer previous layer is always filled. This reduces height and lets us to the operation with much less recursions. Using RBT's we reduce height of the tree to logn. So:

Worst case time complexity = **O(logn)**

Average case time complexity = Since there are a lot more nodes concentrated in deeper layers than upper ones we can say: **O(logn)**

Search

In search operations we don't search the tree by it's key. So tree structure doesn't help us to search faster. I basically went through nodes one by one with a pre-order traversal. This takes O(height of the tree) time and as I mentioned in Insertion part height of RBT is logn. So:

Worst case time comlexitiy: **O(logn)**

Average case time complexity: **O(logn)** (For the same reason with insertion)

RBT vs BST

In binary search trees order of insertions play a big role in shape of the tree. With same set of inputs we can obtain very different trees just by inserting in different orders. In some cases this causes tree to expand to much on one direction without filling previous layers. And this causes tree operations to take longer since they mostly depend on height of the tree. Red-Black trees are designed to solve this problem and thats why they are called balanced binary search trees. After each regular BST insertion we check the tree and make sure it is balanced. This way we can do tree operations in O(logn) which is the smallest height the tree can be.

Augmenting Data Structures

I would augment the Red-Black tree structure that I have, to be a Order Statistics Tree. Since we are ordering the players in 5 different categories(PG, SG, SF, PF, C) in each node I would keep the numbers of nodes in corresponding position for subtrees. This means each node would store 5 more variables that hold number of players in that subtree of that position. Like PG = 0, SG=3, SF=2, PF=1, C=2 for a node. These numbers can be calculated by summing child tree's numbers and adding +1 to node's own position and can be maintained easily. This way I will be able to reach any i'th smallest player in the tree within their position category.

Pseudocode:

```
OS-Select-PG(x, i):                          // i'th smallest element of the rooted at x
 k ← PG(left[x])                             // PG() returns the value of PG variable
if x.position = PG then k = k + 1            //Only if node has the position PG k increases by 1
if i = k then return x                       //Now k value is the rank of the node
if i < k then return OS-Select-PG(left[x], i)  //Function recurs with proper parameters
else return OS-Selcet-PG(right, i - k)
```

Almost same function applies for other 4 position. We just need to calculate the ranks with the correct positions variable.