BLG 335E – Analysis of Algorithms I
Fall 2020-2021 Homework 3 Report
Oguzhan Karabacak 150170021

A) COMPLEXITY

a. Insertion Operation

Inserting the new player into a non-empty tree has three steps. First we create new node and assign values(name,season etc.) and red color to new node.These operations are O(1) so constant because these operations has not any recursive or loop. The second step is that we add tree using Binary Search Tree operation without balancing. The BST insert operation is O(logN) because a red black tree is balanced and logN is equal to height of tree.In the third step,we fix violated red-black properties. First step of re-balanced is restructuring which has O(1) since we change most 5 pointers to nodes. (This case is new node’s parent is not Black or new node is not root and new node’s uncle is Black). So in the worst case, restructuring is done during insert is O(1).Changing Nodes’s color is O(1) but we need to solve the Double-Red case in the entire path from the root of the tree to where the new node is added.

So , in the worst case,time-complexity of the recoloring is O(logN).

Thus, in the worst case, total time for insert is O(logN).

In the average case,the first two step is same as the worst case. However, re-balanced step is different. In Average case, we may not need to re-coloring the entire path. But since we are considering upper bound, we should use tree depth.So in the average case, total time for insert is O(logN).

b. Search Operation

The worst case in the search operation is that the player we are searching for is on the leaf of the tree. In this case, we need to check all the players in the path between root and leaf.

In this case, the time complexity is O (logN) because the length of the tree is logN and we check logN times of the if block.

In the average case, Since the Red-Black tree is balanced, the height of the tree is logN.

Since we are considering the Upper Bound case, the height of the path between the node of the data we are searching for and the root is logN.

Therefore, the average time complexity in search operation is O (logN).

B) Compare RBT vs BST

Both Binary Search Tree and Red Black Tree have binary search property. In other words,Data of node smaller than Root data are located in the left tree and data of node bigger than Root data are in the right tree. In both, node can have 2 children.

The difference between them is that RBT is a self-balancing tree, so there can be no more than 1 depth difference between any two subtree, and each node has color. But BST is not a self-balancing tree, a large depth difference may occur between two subtree, and this increases search time complexity.

For example, let's assume that we add data of 8,7,9 to both trees respectively. In BST, the number of root 8 is 7 and 9 is its children. In this case, time complexity logn is in BST.

When we add the same numbers to RBT, root becomes 8, 7 and 9 are its children. RBT's time complexity is also logn.

But suppose we add the numbers 7,8,9 respectively in the BST tree. In this case, 7 is root, 8 is root's right child, 9 is 8's right child, in this case time complexity is n. Because if the given elements are sorted as decreasing or increasing, the tree grows linearly.

But if we add the numbers 7,8,9 to the RBT tree, the RBT tree balances itself, and 8 becomes root, 7 and 9 are children of root (just like 8,7,9). In this case, even if we give decreasing or increasing sorted elements to the RBT tree, time complexity becomes logn because the tree balances itself.

C) Augmenting Data Structures

If i was to augment Red-Black Tree with 5 new methods , i^{th} Point Guard, i^{th} Shooting Guard, i^{th} Small Forward, i^{th} Power Forward and i^{th} Center. My strategy would be like this:

Since the 5 methods are similar to each other, I will explain here only through the i^{th} Center function.

Count Function

The **Count** function returns the value of how many players have **Center positions** in **the left-subtree**.

The **Count** function works like this: First we define **counter=0**, whenever we find **Center position** we add it to the value of counter.

If root is NULL, we return **counter**. If it is not Null, we check whether the root's position is **Center**. If it is **Center**, we **add 1** to the value of **counter**.

Then we use the **recursive function** to find out how **many Center positions** there are in the **left-subtree** and we give **root-> left as the root parameter**. We **add** the **return value** of this function **to the counter**.

We **repeat** the same process in the **right-subtree**. At the end of the function, we find the **total number of players with the Center position in the Root + Left-subtree + Right subtree and return this value(total number of players).**

Count Function Pseudocode:

```
Count(root)

    1. counter=0;
    2. if root==NULL
    3.         return counter
    4. if root.position == “center”
    5.         counter = counter +1
    6. counter = counter + Count(root.left)
    7. counter = counter + Count(root.right)
    8. return counter
```

find_center() function

First, we use the **Count function** to find out how many (**counter**) Center positions there are to the left of the root.

If the player in **Root** has the **Center position**, we **add 1** to the result we find, while **determining how many Center positions** are in the **left-subtree and root**.

Then we **compare counter** with the number of ***i***th ***value*** requested from us. If the **two values are equal**, that means the ***i***th **Center position** is at **root or left-subtree**.

But **root** may **not** have the **Center position**, so we **check** if the **root position** is **Center**. If the **root** position is **Center**, it means we **found** the ***i***th **Center position**, then we return this **root**.

But if **root position** is not **Center** then ***i***th **Center** is in position **left-subtree.** So we use the **recursive** function to give the **left-node** of root **as the root parameter** while the **i** parameter remains the **same.**

If **counter** and **i value** are **not equal**, we check whether the **i value** is **smaller** than the value of **counter**.

If **i value** is **smaller**, it means that the value ***i***th **Center** position we are searching for is in the **left-subtree of root**. In this case, we give the **left-node** of root as the **root**

parameter and the **i parameter** remains the same. We search for the ***i***

parameter and the **i parameter** remains the same. We search for the ***i***th **Center** position in left-subtree.

The **last possibility** is that the value of **counter** is **smaller than** the **value of i**. In this case, the ***i***th **Center** position is **in the right-subtree**.

Again, we use the **recursive function** and give the **right-node of root as the root** parameter, but since we find count of C positions on the **left-subtree**, our new **i value parameter** becomes (**i value – counter)**.

find_center() function Pseudocode:

find_center(root,i)

```
1.  counter=Count(root.left)
2.  if root.position == "center"
3.          counter=counter+1
4.  if counter == i
5.          if root.position == "center"
6.                  return root
7.          return find_center(root.left, i)
8.  else if i < counter
9.          return find_center(root.left,i)
10. else
11.         return find_center(root.right, i-counter)
```

Maybe the pseudocode is not clear enough. I implemented and this is my c++ code:

```cpp
int Count(Node* root){
   int counter=0;
   if (root==NULL) return counter;
   if (root->position == "center") counter+=1;
   counter+=Count(root->left);
   counter+=Count(root->right);
   return counter;
}

Node* find_center(Node* root,int i){
   int counter=Count(root->left);
   if (root->position=="center") counter+=1;
   if (counter == i){
      if (root->position == "center"){
         cout << root->position << endl;
         return root;
      }
      return find_center(root->left,i);
   }
   else if (i < counter) return find_center(root->left,i);

   else return find_center(root->right,i-counter);
}
```