ISE315 – Analysis of Algorithms
Fall 2021
Homework-3 Report

Aral ŞEN
150170217

a) (10 points) Write down the asymptotic upper bound for the insertion and search operations of Red-Black Tree for worst case and average case with detailed explanations.

Search ->

In a red-black tree every leaf of a subtree has the same amount of black nodes up until the root thus the complexity of search for a red-black binary tree is O($\log\ n$) in worst case. Red nodes can only be found in the space between two black nodes. If each leaf in a tree has the same depth of black nodes and there are only red nodes between the black nodes, the tree's height may be calculated as follows:

max($\log\ black\_nodes\ + \log\ red\_nodes$)

We may claim that this operation is upperbounded by 2*log n, which is O(log n), because there will always be more black nodes than red. In the worst-case scenario, we can compare each level of the tree, resulting in a log n times comparison. There will be a logn/n times comparison in the average case.The average case is same also O($\log\ n$) for RBT.

So the complexity will be: $\Theta(\log_2 n)$

Insert ->

In both the worst and best scenarios, the insertion complexity of red-black trees is O($\log\ n$). If the RBT is empty, we may simply add the new node to the tree in O(1) time; otherwise, we must ensure that the insertion meets the properties of RBT. For insertion, we have three parts. First, we search for a suitable site to put the node into the tree, which takes $\log\ n$ time. Second, coloring the new node red (all new nodes are initially red) takes O(1) time since it only involves setting a value for one node. Finally, when balancing the tree to restore the red-black property after insertion, we only go up one or two nodes in the tree depending on the scenario, which takes time equal to the depth of the tree, which in the worst case is $\log\ n$. In the best case situation, our insertion does not violate the rb tree property, but it still takes $\log\ n$ time, therefore the average time will be : $\Theta(\log_2 n)$

So the complexity will be: $\Theta(\log_2 n)$

b) (5 points) Compare Red-Black Tree with Standard Binary Search Tree in your own words.

RBT and BST maintain the binary search tree property, but a regular binary search tree is not self-balancing. For example, in BST depending on the order of insertion, if nodes are put uniformly to the tree, most likely the best case is for the tree to be balanced (O(logn)) but this is the best case, in worst BST will be O(n) like linked-list. However, Red black tree is a self-balancing tree, which means re-organize themselves after each insert operation so that their depth never passes $\log\ n$. A binary search tree can have the depth n, where all nodes have one child, that can cause delete, insertion, searching operations takes linear, O(n) time in worst-case. Red black trees guarantees (because of their self balancing properties) to all operations takes logarithmic, O(log(n)) time.

c) (15 points) Suppose that you are given the position (Point Guard PG, Shooting Guard SG, Small Forward SF, Power Forward PF or Center C) of the players. If you were to augment your Red-Black Tree with 5 new methods, ith PG, ith SG, ith SF, ith PF and ith C, that return the name of the ith Point Guard, ith Shooting Guard, ith Small Forward, ith Power Forward and ith Center, respectively, what will be your strategy? Provide a pseudocode with explanations to implement these methods but do not implement them.

For this implementation, we can use the dynamic order statistics. Instead of merely keeping track of the player's name as a key, I can also maintain track of the node's location. While searching through the tree, I can increment the counter by one each time I discover the position I'm looking for. When the number reaches a specific I th value, I can simply return the player's name.

```
fun find_pos (node, value, position):
    global count  # initialized in the main as 0
    global flag  # initialized in the main as false
    if (flag) return

    if (node != null && node.position = position) {
                count = count + 1
                if count = value {
                     print(node.data)
                     flag = true
                     count = 0
                     return
                 }  else {
                     searchPosition(node.left, value, position)
                     searchPosition(node.right, value, position)
                 }
    }
    else if (node != null && node.positon != position) {
          searchPosition(node.left, value, position)
          searchPosition(node.right, value, position)
          }
    else
         return
```