Beyza Ozan
ID: 150180056

BLG 335E – Analysis of Algorithms I

Homework 3

Part 2 - Report

Insertion Operation - Worst Case :

Assume that the element is inserted at position maximum h value for worst case .(Maximum h is $2.log(n+1)$).

The tree should be rearranged from the leaf to the root. Each rotating or recoloring operation's time complexity is O(1) and operations will be performed $h = 2.log(n+1)$ times.

Therefore, the worst case of insertion is => $2.log(n+1) * 2.log(n+1) * O(1) = O(logn)$

The time complexity of search operation of Red-Black Tree is **O(logn).**

Insertion Operation - Average Case :

The difference between average case and worst case in the tree is height of the nodes. Since the height of nodes in red-black trees is the same for both two cases, we can say complexity of average case is same as worst case. Hence, the complexity is **O(logn).**

Search Operation - Worst Case :

Theorem : A red-black tree with n keys has height is $h \leq 2lg(n+1)$.

Proof by induction : Assume that red nodes have black parents. A black parent can have 0, 1 or 2 red children.
A black parent with 0 red children -> 2 connections
A black parent with 1 red children -> 3 connections
A black parent with 3 red children -> 4 connections

Therefore, the height h of this tree is 2-3-4.

The number of leaves in each tree is n + 1. The maximum number of red nodes is h/2 so, $h/2 \leq H$.

$2^H \leq n+1$ $\qquad$ $h/2 \leq H \leq log(n+1)$ $\qquad$ $h \leq 2 * log(n+1)$

Hence, worst case of search operation is **O(logn).**

Search Operation - Average Case :

The difference between average case and worst case in the tree is height of the nodes. Since the height of nodes in red-black trees is the same for both two cases, we can say complexity of average case is same as worst case like insert operation. Hence, the complexity is **O(logn).**

RBT vs BST :

Red-Black Trees are a special case for binary search trees. Both binary search trees and red-black trees maintain the binary search tree property. The main difference is that a red-black tree is a **self-balancing tree**, while a binary search tree is not.

Red-Black Tree contains a few extra lines of code that describe a red and black node, as well as a few more operations in insertion and deletion. However, binary search tree can cause searches to take linear time whereas a red-black tree guarantees a search operation takes logarithmic time because it is self-balancing.

Augmenting Data Structures :

Same logic is used for all five positions. Thus, just point_guard pseudocode is shown below.

In order to find the ith point guard player , we use **inorder traverse** in binary search tree. First we assign player to the root of the tree and counter to zero. During the traversal if we find the player with the position point_guard, we increment counter by one. When the counter is equal to i, we return the name of the player which is ith point guard.

```
in_order(player, i, counter)
    if player = NULL
        return
    if player.left
        in_order(player.left, i, counter)
    if player.position = point_guard
        counter ++
        if counter = i
            return player.name
    if player.right
        in_order(player.right, i, counter)
```