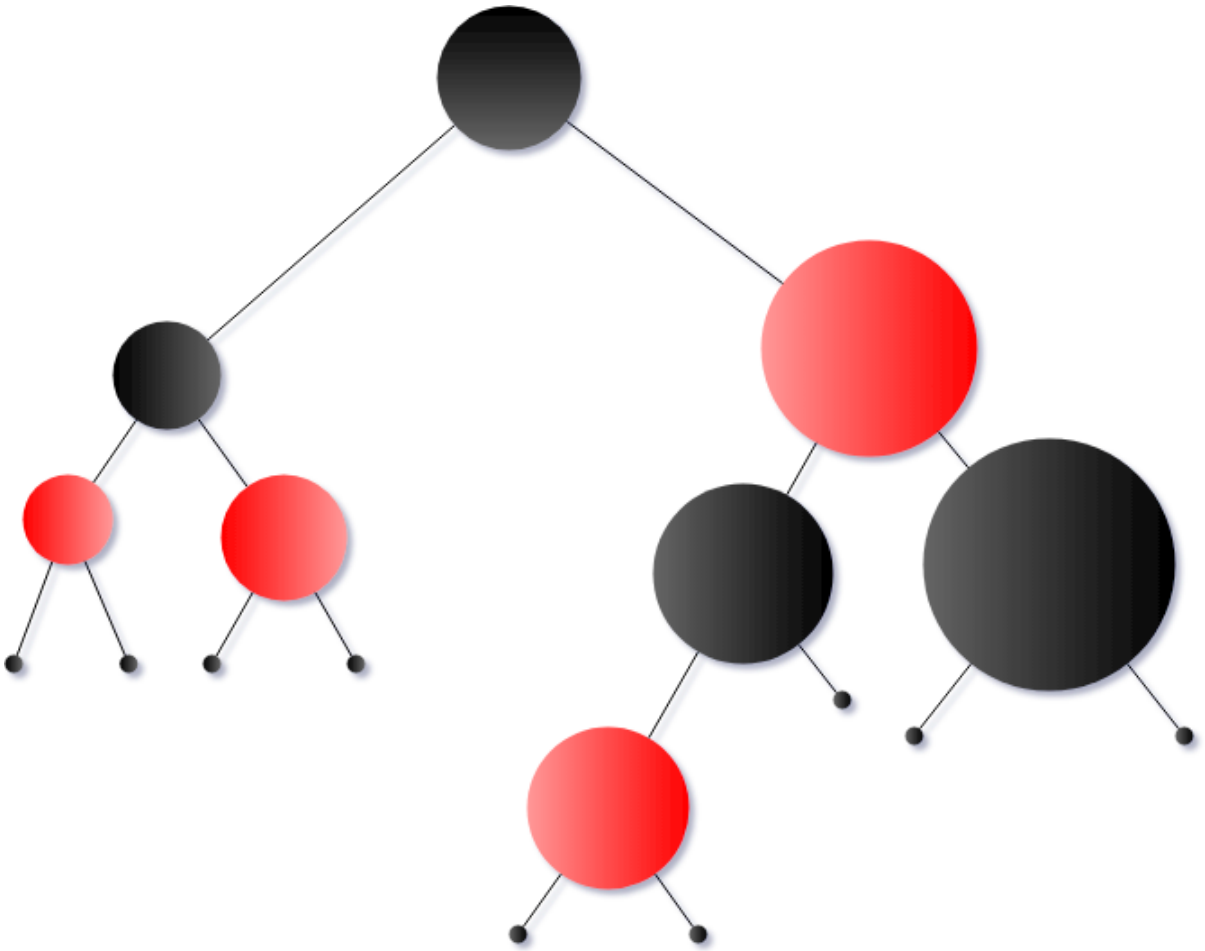

BLG 335E

Homework 3 Report

Red-Black Tree

Berke Dönmez

150170045

January 11, 2021

1 Complexity

Before starting to explain the complexities of the operations, it would be very useful to find the upper bound of the height (h) of a Red-Black Tree with n nodes. First, let's recall the properties of RBT:

- A node is either red or black.
- NULL nodes (nodes' non-existing children) are assumed to be leaves. Leaves are black. (Leaves are children of their parents as well.)
- Every simple path from a node to its descendant leaves contains the same amount of black nodes. This amount is called black height of the node (excluding itself).
- If a node is red, then its parent is black.
- Root is black.

Let's connect the children of the red nodes to their parents and erase the red nodes. If a black node has,

- no red children, in the new tree, it will have 2 (black) children.
- 1 red child, it will have 3 children.
- 2 red child, it will have 4 children.

These are all of the possible cases. Therefore, the new tree is guaranteed to be a 2-3-4 tree, in which every non-leaf node has 2, 3 or 4 children. **Because of the black height property of RBT, depth of every leaf (number of nodes in the simple path from the root to leaf) in the new tree is the same because all nodes are black in this tree.** Height of the RBT was h. Since every red node had a black parent, the longest path from the root to a leaf (there were h nodes) contained at most $\frac{h}{2}$ red nodes (rounded down, but unnecessary to consider), thus, it contained at least $\frac{h}{2}$ black nodes. Consequently, height of the 2-3-4 tree h' is at least $\frac{h}{2}$. Since every node in the 2-3-4 tree has at least 2 children, the number of leaves is at least $2^{h'}$ (there are h' levels and each level contains at least 2 times more nodes than the previous level). Now, observe that the number of leaves in 2-3-4 tree is the same as the number of leaves in RBT because leaves stayed as leaves and any internal node did not become a leaf as there were not any red leaves. It is time to calculate the number of leaves in the RBT. Every node had 2 children and there were n nodes. Therefore, there were $2 * n$ children. Now, exclude the internal nodes (non-root nodes except leaves) as we included them

in $2 * n$. There are only leaves left: $2 * n - (n - 1) = n + 1$ leaves exist. Finally, let's write the inequality of the number of leaves:

$$n + 1 \geq 2^{h'}$$

Also:

$$h' \geq \frac{h}{2}$$

As a consequence,

$$log_2(n + 1) \geq h' \geq \frac{h}{2}$$

$$h \leq 2log_2(n + 1)$$

Worst/average height of an RBT is $\mathcal{O}(log_2 n)$. Now, let's examine the operations.

1.1 Search

RBT is just a smarter Binary Search Tree. Therefore, upper bound of worst/average complexity for searching is $\mathcal{O}(height) = \mathcal{O}(logn)$ because at every step before finding what we are searching for, we go one level down (going to a child) and there are $height$ levels.

1.2 Insertion

First step of insertion is searching which leaf to insert. This step is done in $\mathcal{O}(logn)$ (searching a leaf). After we find the leaf, we insert the node. Then, in order to fix the violations, recoloring and rotations are done at most $height$ times. Both recoloring and rotation are done in $\mathcal{O}(1)$. Thus, insertion is done in $\mathcal{O}(logn)$ (worst/average).

2 Red-Black Tree vs. Binary Search Tree

Both trees' purpose is to be fast for some operations: insertion, deletion, search etc. but compared to RBT (Red-Black Tree), BST (Binary Search Tree) is a much simpler and poorer data structure since BST does not guarantee the height (total number of nodes on the simple path from the root to the deepest leaf) to be much less than the total number of nodes whereas RBT (stronger/smarter BST) guarantees. Why do we even height to be much smaller than the total number of nodes? Because, as programmers, our aim is to have data structures that do not rely on randomness, i.e. it is preferable in any cases (even in the worst case). Imagine constructing a BST with a sorted list of numbers (in descending or ascending order) in the same order the numbers ordered. At every time, the current number is going to be inserted below the deepest node. Thus if we call these numbers as $x_0, x_1, x_2, ..., x_n$, before inserting x_i, height of the tree will be i. Therefore, after inserting all the numbers, total number of

operations is $0 + 1 + 2 + ... + n = \frac{n*(n+1)}{2} \approx n^2$. Now, the following question should be asked: What was the purpose of using tree for this list? Using tree was totally unnecessary. Even after completing the construction, there would be no difference between the original list and tree. Consider the cases close to being worst, this problem also occurs in those cases. This is where the importance of RBT (or other self-balancing trees) is perfectly understood: Regardless of how elements in the input are arranged, RBT can rebalance itself in order to avoid being a bad/poor BST, having its height **asymptotically** bounded by $\mathcal{O}(logn)$ where n is the number of nodes at any moment. With this advantage, it is way better (handles the operations faster) than the poor standard BST.

3 Augmenting Red-Black Tree

(Players of a certain position is assumed to be ordered in lexicographically increasing names.) There are 5 different positions:

1. Point Guard (PG)
2. Shooting Guard (SG)
3. Small Forward (SF)
4. Power Forward (PF)
5. Center (C)

Every player has a role and it is asked to implement a method that returns the i^{th} player of a certain role. It is important to highlight that there is only one method. It takes the preferred position as an integer parameter.

3.1 First Approach

Let's add an attribute (integer) to each node. This attribute tells which role the player plays as. Since RBT is constructed with nodes' keys being the names of the players, every name in the left subtree is less than the name in the current node and name in the current node is less than every name in the right subtree. Thus, we can traverse the tree inorder and count the number of players of the target role. When we reach i, we find our target player.

Pseudocode is as follows:

```
string find_ith(Node* root, int pos, int& i)
    if(root == NULL)
        return "" //empty string
    string leftName = find_ith(root->left, pos, i)
    if(leftName != "")
        //player has been found in the left subtree
        return leftName
    if(root->pos == pos)
        i--
    if(i == 0)
        return root->name
    string rightName = find_ith(root->right, pos, i)
    if(rightName != "")
        //player has been found in the left subtree
        return rightName
    return "" //player has not been found yet
```

This approach works in $\mathcal{O}(n)$ as it can traverse all nodes in the worst case.

3.1.1 Second Approach (Better)

Another approach requires adding an integer array of size 5 as attribute to each node. Each element keeps the total number of players of the related role in the subtree (including the node itself). This array should be updated when inserting/deleting nodes. This approach is similar to previous approach but it is more efficient: Instead of recursing to the deepest nodes, just check the number of players of target role in the subtrees. The function only goes along a path from root to descendants. Pseudocode is as follows:

```
string find_ith(Node* root, int pos, int i)
    if(root == NULL)
        return ""
    int lCount = 0, rCount = 0
    if(root->left != NULL)
        lCount = root->left->count[pos]
    if(root->right != NULL)
        rCount = root->right->count[pos]
    //is current's position pos?
    bool isCurrent = root->count[pos] - (lCount+rCount) == 1
    if(leftCount >= i)
        return find_ith(root->left, pos, i)
    if(leftCount == i-1 && isCurrent)
        return root->name
    return find_ith(root->right, pos, i - lCount)
```

Here is the explanation: Let number of players (playing in `pos`) in the left subtree be l. If $l >= i$, then i^{th} player is guaranteed to be in the left subtree (there are at least i players in left subtree). Else if $l == i-1$, i^{th} player cannot be in the left subtree. Check the current player. If the current player plays in `pos`, then current player is the i^{th} player. Else, ($l < i-1$ or current player does not play in `pos`) i^{th} player is guaranteed to be in right subtree. If there is no i^{th} player, function returns empty string.

This approach works in $\mathcal{O}(logn)$ (height) since it goes one level deeper at every step.