ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 335E

ANALYSIS OF ALGORITHMS I
PROJECT REPORT

PROJECT NO : 3

DATE : 05.01.2021

STUDENT:

NAME : ABDULKADİR

SURNAME : PAZAR

NUMBER : 150180028

AUTUMN 2020

1 Complexity

Height of the tree is O(log n) Proof:

Claim: subtree rooted at x has at least $2^{black\_height(x)} - 1$ nodes. Proof by induction:

If x is a leaf, (height of 0) the subtree indeed contains $2^0 - 1 = 0$ nodes.

For the inductive step, we consider an internal node x with two children. Each child has black height of $black\_height(x)$ or $black\_height(x) - 1$ whether their color is red or black respectively. We can conclude that subtrees of each child has at least $2^{black\_height(x)-1} - 1$ internal nodes. In conclusion, The subtree rooted at x contains at least $(2^{black\_height(x)-1} - 1) + (2^{black\_height(x)-1} - 1) + 1 = 2^{black\_height(x)} - 1$ internal nodes.

To prove height of the tree is O(log n), let h be height of the tree. Because of the property 4, at least half the nodes on any simple path from the root to a leaf not including the root, must be black. So, the $blackheight \geq h/2$. So,

$$n \geq 2^{h/2} - 1$$

$$log(n+1) \geq h/2$$

$$2log(n+1) \geq h$$

Asymptotic upper bound for insertion in average case is O(log n) Proof:

Insertion consists of two functions, insert() and insertfix().

During insert() function, we traverse down the tree iteratively and place newly added node to a leaf according to alphabetic order. Since the height of the tree is O(log n), this operation takes O(log n) time.

During insertfix() function, we enter the while loop again only if case 1 is executed and every loop we move 2 levels up in the tree. Since the height of the tree is O(log n), this operation takes O(log n) time.

In total since both insert() and insertfix() functions take O(log n) time, insertion takes O(log n) time.

Asymptotic upper bound for insertion in worst case is O(log n) Proof:

Since this tree is a red-black tree, the worst-case height and average case height is both O(log n). Consequently worst case run time of insertion is also O(log n).

Asymptotic upper bound for search in average case is O(log n) Proof:

Time complexity of the search depends on height of the tree since we look for the name we are searching and move left or right depending on alphabetic order. Since the height of the tree is O(log n), time complexity of the search is O(log n).

Asymptotic upper bound for search in worst case is O(log n) Proof: Since this tree is a red-black tree, the worst-case height and average case height is both O(log n). Consequently worst case run time of search is also O(log n).

2 RBT vs. BST

Red-Black Trees are self balancing whereas Binary Search Trees do not have this property. Because of this property of RB Trees, search operation is guaranteed to take logarithmic time. Binary search trees can be deep enough to make searches take linear time. Because RB Trees are self balancing, they always have O(lg n) height.

3 Augmenting Data Structures

I would keep numPG, numSG, numSF, numPF and numC as variables in the nodes created from reading the file. When the position is read from file depending on the player's position I would set that variable to 1 and other variables to a sentinel value 0. Then I would modify insert(), rotate_left() and rotate_right() functions and write getnthPG(), getnthSG(), getnthSF(), getnthPF(), getnthC() methods. The pseudocode of these methods are given below. Since insert(), rotate_left() and rotate_right() are modified, each node will hold the information of how many PG, SG, SF, PF, C exists in their subtrees.

3.1 Pseudocode

3.1.1 getnthPG()

getnthPG(root,n)
numPG = 0
if left[root] exists
then numPG < − numPG[left[root]]
if position[root] = PG
then numPG < − numPG + 1
if numPG = n and position[root] = PG
then return name[root]
else if n ≤ numPG
then return getnthPG(left[root], n)
else
then return getnthPG(right[root], n - numPG)

3.1.2 getnthSG()

getnthSG(root,n)
numSG = 0
if left[root] exists
then numSG <− numSG[left[root]]
if position[root] = SG
then numSG <− numSG + 1
if numSG = n and position[root] = SG
then return name[root]
else if n ≤ numSG
then return getnthSG(left[root], n)
else
then return getnthSG(right[root], n - numSG)

3.1.3 getnthSF()

getnthSF(root,n)
numSF = 0
if left[root] exists
then numSF <− numSF[left[root]]
if position[root] = SF
then numSF <− numSF + 1
if numSF = n and position[root] = SF
then return name[root]
else if n ≤ numSF
then return getnthSF(left[root], n)
else
then return getnthSF(right[root], n - numSF)

3.1.4 getnthPF()

getnthPF(root,n)
numPF = 0
if left[root] exists
then numPF <− numPF[left[root]]
if position[root] = PF
then numPF <− numPF + 1
if numPF = n and position[root] = PF
then return name[root]
else if n ≤ numPF
then return getnthPF(left[root], n)
else
then return getnthPF(right[root], n - numPF)

3.1.5 getnthC()

getnthC(root,n)
numC = 0
if left[root] exists
then numC <− numC[left[root]]
if position[root] = C
then numC <− numC + 1
if numC = n and position[root] = C
then return name[root]
else if n ≤ numC
then return getnthC(left[root], n)
else
then return getnthC(right[root], n - numC)