BLG335E Algorithm Analysis Assignment3 Report

Mihriban Nur Koçak - 150180090

07.01.2022

Question 1.1

To make the height of the given Binary Search Tree shortest, we need to transform it to be as balanced as possible. To be able to decide what to do, first we should obtain the increasing order of the nodes by applying inorder traversal:

25-28-30-35-45-**55**-65-74-85-90-97

As can be seen, the middle value is 55 so first we need to transform tree to set 55 as the root by performing following operations:

- LeftRotate(35)
- LeftRotate(45)

25-28-30-35-45-55-65-74-**85**-90-97

Then, as can be seen, the middle value of the second half is 85 so we need to transform tree to set 85 as the root's right child by performing following operations:

- RightRotate(97)
- LeftRotate(74)
- LeftRotate(65)

25-28-**30**-35-45-55-65-74-85-90-97

Then, as can be seen, the middle value of the first half is 30 so we need to transform tree to set 30 as the root's left child by performing following operations:

- LeftRotate(25)
- RightRotate(35)
- RightRotate(45)

The steps are visualized in Figure4 and the finalized tree can be seen in Figure. As can be seen, the height of the resulting tree is 3 which is minimum possible height.

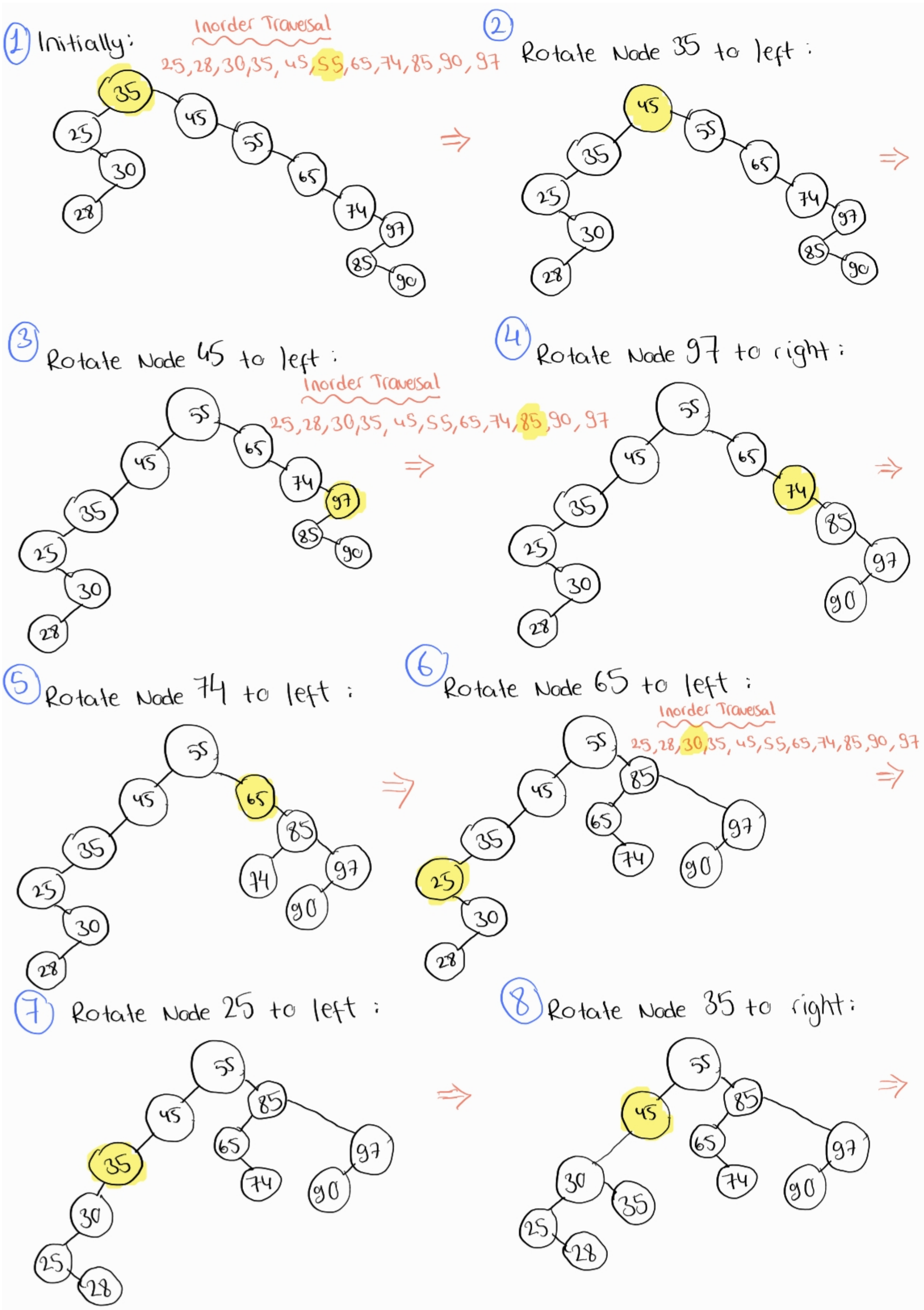


Figure 1: Transformation steps

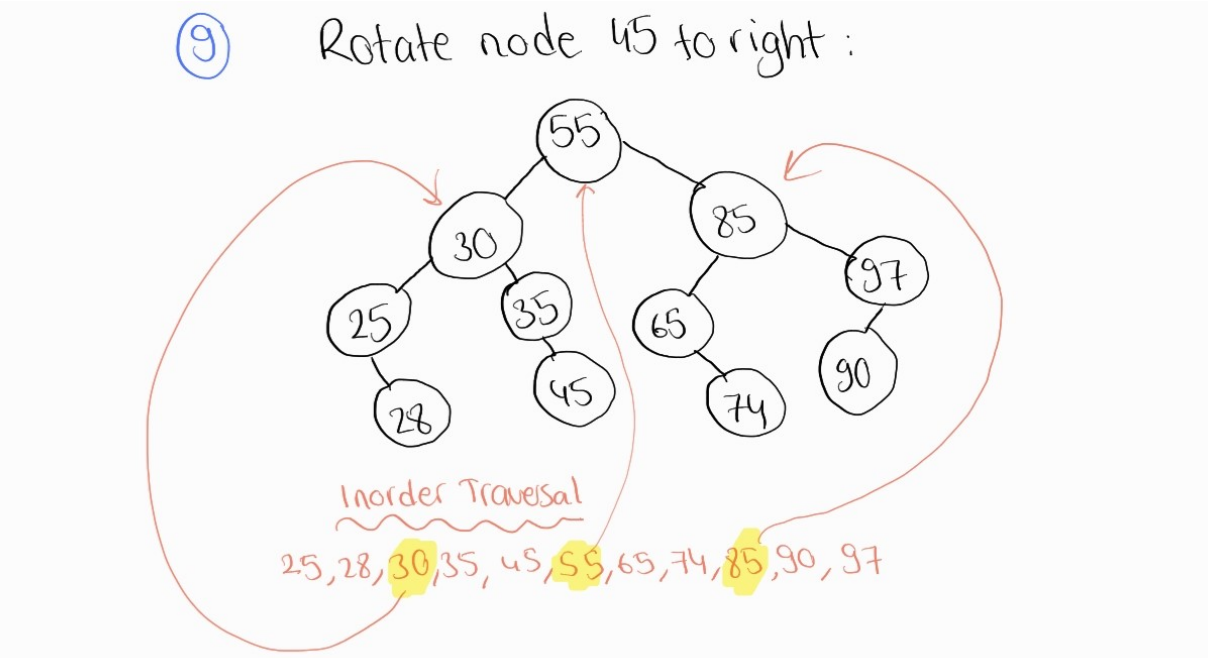


Figure 2: Result

Question 1.2

A Red Black Tree is a special kind of the Standard Binary Search Tree where each node has the additional color property which is red or black and other properties:

- The root node is black
- The children of a red node are black.
- Count NILL as black node
- Path from a node to NILL has same amount of black nodes for all paths for all nodes.

While red black tree performs fixup operation after every iteration, standard binary search tree doesn't do anything. Thanks to this fixup operation, red black tree protects its **self-balanced** property. Therefore, while red black tree is a self-balanced tree, standard binary search tree is not so that node chains are possible for binary search tree according to insertion order. Self-balanced means being balanced independent of the insertion order. As a result, while the worst and base case running times of red black tree are same, the worst and base case running times of standard binary search tree are different.

Question 2

Search Operation

Worst case:

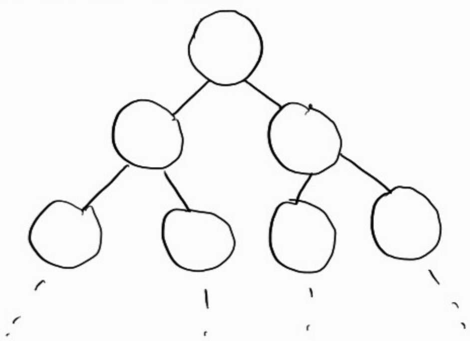
While searching in a red-black tree, worst case means searching until the leaves. This corresponds going from root to one of the leaf. At every step, the comparison operation is done and cursor goes left or right. Since constant operation is done every step

$$T(n) = c\log n = O(\log n)$$

n: number of nodes

$\log n$: height of a binary search tree

Average case:



level	# nodes in this level
0	2^0
1	2^1
2	2^2

We have n nodes, to find the average cost we need to sum costs of all individual nodes and divide this sum by n.

$$T(n) = \frac{\left(\sum_{i=0}^{\log n} 2^i \cdot i\right) + \overset{\text{for root}}{1}}{n} = \frac{O(n\log n)}{n} = O(\log n)$$

height   level   # nodes in a level

Insertion Operation

Insertion consist of two steps. First one is finding the appopriate place which costs same as searching. Second one is fix-up which is identical operation of red black tree.

Since we need to find suitable place for the node to insert, we need to go until a leaf

So that first part costs : $T_1(n) = \log n$ height of the binary search tree

Now I will explain worst and avarage case scenerios for fix-up part

Worst case

For fix-up there are three cases:

Case 1 $\Rightarrow$ while loop continues

Case 2 and 3 $\Rightarrow$ rotate operations are done while loop ends

Since we are trying to find worst case, we shuld consider the case which while loop iterates until the root. This corresponds occurence of case 1 at every loop iteration until the root.

$$T_2(n) = \log n$$ height of tree

If we add costs of two parts

$$\underbrace{T_1(n)}_{\text{inserting}} + \underbrace{T_2(n)}_{\text{fix-up}} = 2 \log n = O(\log n)$$

Avarage case :

As I have explained at worst case part, for fix-up

$$T_{worst}(n) = \log n$$

Then, the best case scenerio corresponds skipping while loop part even getting in. So for fix-up

$$T_{best}(n) = \underbrace{c}_{\text{constant}}$$

Since occurence time and order of cases in fix-up is differs from node to node, we need to consider expected value

$$E[T(n)] = \sum_{i=1}^{\log n} i \cdot \frac{1}{\log n}$$

→ all costs have same probability of occurence

possible costs are from 1 to logn

$$E[T(n)] = \frac{\log(n) \cdot (\log(n)+1)}{2} \cdot \frac{1}{\log(n)}$$

$$= \frac{\log(n)+1}{2} = O(\log n)$$

If we add costs of two parts

$$\underbrace{O(\log n)}_{\text{inserting}} + \underbrace{O(\log n)}_{\text{fix-up}} = O(\log n)$$

Question 3

First of all, to be able to find the name of the ith Sports, ith Action, ith RP, ith Racing, and ith Strategy publisher, some additional attributes should be added to Video Game node which I have implemented. In addition to current attributes, the node should contain **genre**, **Number of nodes whose genre is sports in the subtree rooted at this node**, **Number of nodes whose genre is action in the subtree rooted at this node**, **Number of nodes whose genre is role-playing in the subtree rooted at this node**, **Number of nodes whose genre is racing in the subtree rooted at this node**, **Number of nodes whose genre is strategy in the subtree rooted at this node**. Also in case of the possibility that one publisher can have games with different genres, we need following variables: **isGenreSports**,**isGenreAction**,**isGenreRp**,**isGenreRacing**,**isGenreStrategy**.

- string genre
- int sizeOfSubtreeSports
- int sizeOfSubtreeAction
- int sizeOfSubtreeRp
- int sizeOfSubtreeRacing
- int sizeOfSubtreeStrategy
- bool isGenreSports
- bool isGenreAction
- bool isGenreRp
- bool isGenreRacing
- bool isGenreStrategy

Before calling one of the functions to find ith genre, sizeOfSubtrees of all genres for each node should be calculated by traversing.

NILL node's all sizeOfSubtree attributes are initialized as 0 as constant while they are constructed.

Pseudocode to decide sizeOfSubtree for each genre using post-order traversal since sizeOfSubtree data are calculated according to children's sizeOfSubtree data:

```
calculateSubtrees():
    while traversing using the post-order traversal method:
        sizeOfSubtreeSports = node.leftChild.sizeOfSubtreeSports + node.rightChild.sizeOfSubtreeSports
        sizeOfSubtreeAction = node.leftChild.sizeOfSubtreeAction + node.rightChild.sizeOfSubtreeAction
        sizeOfSubtreeRp = node.leftChild.sizeOfSubtreeRp + node.rightChild.sizeOfSubtreeRp
        sizeOfSubtreeRacing = node.leftChild.sizeOfSubtreeRacing + node.rightChild.sizeOfSubtreeRacing
        sizeOfSubtreeStrategy = node.leftChild.sizeOfSubtreeStrategy + node.rightChild.sizeOfSubtreeStrategy
        if(node.isGenreSport)
            then sizeOfSubtreeSports++ #if publisher has sports game
        if(node.isGenreAction)
            then sizeOfSubtreeAction++ #if publisher has action game
        if(node.isGenreRp)
            then sizeOfSubtreeRp++ #if publisher has rp game
        if(node.isGenreRacing)
            then sizeOfSubtreeRacing++ #if publisher has racing game
        if(node.isGenreStrategy)
            then sizeOfSubtreeStrategy++ #if publisher has strategy game
```

Figure 3: Pseudocode of Calculate Subtrees

After sizeOfSubtrees are calculated, we can continue with selection. For selection, the rank which shows node's order is used.
To find the name of the ith Sports, following function can be used:

```
selectSports(node,i):
    if(node == NILL) #it is possible to reach to NILL in case of ith data doesnt exists
        then print("DOESNT'T EXIST")
            return
    else
        then
            if(node.isGenreSport)
                #if node's genre is sport which means it has leftchild's +1th rank
                then rank = node.leftChild.sizeOfSubtreeSports + 1
                    if(i == rank)
                        then return node.publisherName
                    else if(i < rank)
                        then return selectSports(node.leftChild,i)
                    else
                        then return selectSports(node.rightChild, i - rank)
            else
                #if node's genre is not sport which means its rank is not different than leftchild's
                then rank = node.leftChild.sizeOfSubtreeSports
                    if(i <= rank)
                        then return selectSports(node.leftChild,i) #I am not sport, go and search in my leftsubtree
                    else
                        then return selectSports(node.rightChild, i - rank)
```

Figure 4: Pseudocode of Calculate Select Sports

At the innermost else parts, we are doing recursive call for (i - rank), not for (i), because if the order which we are searching is larger than current node's order(rank), we should look into the (i-rank)th smallest value between values larger than current node's value.

The implementation is same for the other genres, just the sizeOfSubtree and isGenre variables should be replaced according to genre.