BLG335E
ANALYSIS OF ALGORITHMS I
FALL 2021

ASSIGNMENT #3 RB Tree
REPORT

ŞERİFE DAMLA KONUR
040160433

CRN : 11458
Date of Delivery : 11.01.2022

1. INTRODUCTION

In this assignment, we are expected to implement red black tree keeping the video games in VideoGames.csv.

In my project, all the codes are written in 040160433.cpp file. Also the compilation and execution commands as follows :

Compilation : g++ -std=c++11 040160433.cpp

Execution : ./a.out VideoGames.csv

```
-zsh
damlakonur@Damla-MacBook-Pro BLG335E_HW3_Fall2021 % g++ -std=c++11 040160433.cpp

damlakonur@Damla-MacBook-Pro BLG335E_HW3_Fall2021 % ./a.out VideoGames.csv
```

2. QUESTIONS

2.1 Make the height of the Binary Search Tree given in Figure 1 shortest with Left and Right rotation operations. Draw your steps and define which node is rotated in which direction. (You can use rotation operations which you desire order).

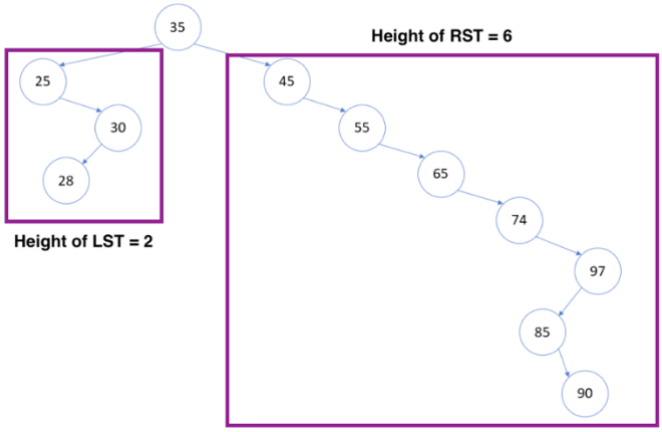
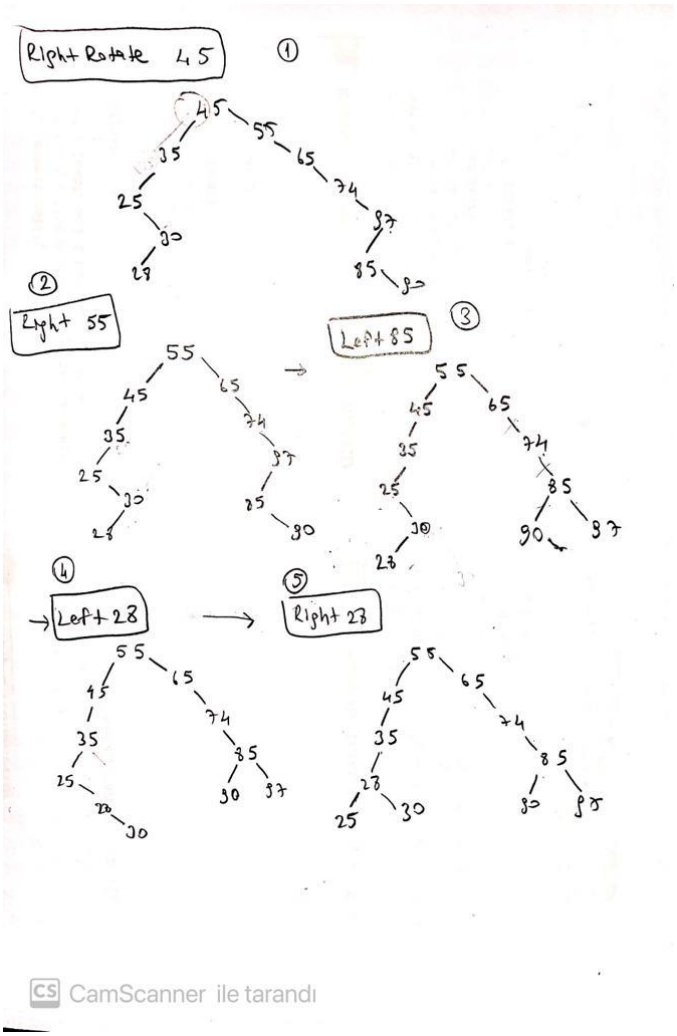


Figure 1. Binary Search Tree

As can be seen in Figure 1. The height of the left sub-tree is equal to two and height of the right sub-tree is equal to six, so the balance factor is equal to minus four. This means that the given binary search tree is in-balanced. Therefore, in order to make the height of the BST shortest we have to do some rotate operations. Also it can be seen that the BST is left in-balanced, thus we can start right rotation. The all steps as follows :



2.2 Compare Red-Black Tree with Standard Binary Search Tree in your own words.

A red–black tree is a special binary search tree where each node has a color attribute, the value of which is either red or black. The coloured nodes allow for the data structure to be self-balanced. In Standard Binary Search Trees, if the input is already sorted or reversed sorted the tree will be in a line shape. Therefore, in such worst cases, Standard Binary Search Trees are not balanced and this situation leads to spending more time while inserting, searching or deleting. On the other hand, In Red Black Trees, as the rules of red and black conditions, rotations are occurred and this situation leads to having balanced trees compared to Standard Binary Search Trees. Thus, depth or height of the tree is not high even for the worst case scenario; therefore, inserting, searching and deleting operations take less time than Standard BSTs. Even though their principles are almost identical and the time complexity of their average cases are the same, for the worst case scenario RBTs is better than Standard BST. Except this

difference, Standard Binary Search Trees and Red Black Trees have the same principles that right child is bigger than the left child and the parent node.

2.3 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.

	Time Complexity	
Operation	**Average Case**	**Worst Case**
Search	$O(logn)$	$O(logn)$
Insertion	$O(logn)$	$O(logn)$

Asymptotic upper bound for the insertion and search operations of Red Black Tree for worst case and average case are given in the above table.

Red-black trees guarantee that no other path is more than twice as long as any other, by constraining the node colors on any particular path from the root to a leaf, so that the Red Black Trees are considered balanced. Balanced means that the depth of the tree is $logn$, where n is the number of nodes. In the worst case, that is the case with the tallest tree, there must be some long path from the root to a leaf. Since the number of black nodes on that long path is limited to $logn_b$(black height of the tree), the only way to make it longer is to have lots of red nodes. Since red nodes cannot have red children, in the worst case, the number of nodes on that path must alternate red/black. Thus, that path can be only twice as long as the black depth of the tree. Therefore, the worst case height of the tree is $2logn_b$ so $logn$.

While reaching the appropriate node to insertion, asymptotic upper bound of this operation is *O*(logn). To maintain the rbt properties after the insertion operation, recoloring and rotation operations are required. These operations' asymptotic upper bound is *O*(1). In this way, for the average case upper bound is *O*(logn)***O*(1)=*O*(logn). As can be explained in previous paragraph worst case height of the tree is $logn$. So the worst case scenario is actually the average case scenario. Therefore, **asymptotic upper bound** of **insertion operation** for the worst case is $\mathbf{O}(\boldsymbol{logn})$**.**

The same principle is also valid for the search operation in RBTs. To find the any node in a tree, time is required proportional to the height of the tree. As this situation depends on the height of the tree, asymptotic upper bound of search operation for the average case is *O*(logn) as well. As mentioned above, the RBTs are balanced trees. Therefore, their worst case scenario

can be considered as average case scenario. So, the asymptotic upper bound of **search operation** for the worst case is ***O*(logn)** as well.

2.4 Suppose that you are given the genre (Sports, Action, Role-Playing (RP), Racing, Strategy) of the publishers' video games. If you were to augment your Red-Black Tree with five new methods that return the name of the ith Sports, ith Action, ith RP, ith Racing, and ith Strategy publisher, what would be your strategy? Provide pseudocode with explanations to implement these methods but do not implement them.

Since the key is Publisher name, we should start from the left of the tree. Arguments of the function; i refers to index of the desired genre, root refers to the root node of the tree, videoGamesArr is string array for holding the Publisher names of the desired genre, and size is the last index of the videoGamesArr which is zero at the beginning.

```
1 pubNameOfTheithAction(i, root, videoGamesArr, size)
2       pubNameOfTheithAction(i, root->left, videoGamesArr, size);
3       if (root->genre == 'Action')
4       then
5               videoGamesArr[size] = root->pubName;
6               size := size + 1;
7       if (size == i)
8       then
9               return videoGamesArr[size-1];
10      pubNameOfTheithAction(i, root->right, videoGamesArr, size);
```

- In the second line, in-order traversal start from root to left child.

- In the 3rd and 5th lines if the node's genre is desired genre, this node's Publisher name is added to the array and size is increased by one.

- Then check the size whether it is equal to the desired index (i) or not, if it is equal return the name of the desired genre's Publisher name.

- In the last line, in order to complete in-order traversal the function traverse the right subtree.

Same pseudocode can be used for the other genres as well.