

BLG 335E - Analysis of Algorithms I**Homework 3****Complexity Analysis****Search Operation:****Worst-Case Analysis:**

The search operation is based on comparison of the key values. It iterates over the depths of the tree until it reaches to intended node or NIL node. Therefore, the worst-case of search operation is searching for a node that is **not exist** on a tree. In this case, function must traverse all depths and returns false.

We know that balanced Red-Black Tree with N nodes has $\log(N)$ depth but for the worst case, we have to count NIL node as a depth. Therefore, for the worst case it must perform $\log(N + 1)$ operations and its complexity is **$O(\log N)$** .

Average-Case Analysis:

To calculate the average case, we have to calculate all search complexities of each possible input. There are two conditions theses are finding node or not. To calculate the case that finds node, we have to think about all nodes of tree and for not finding case, we can think like searching for NIL node.

We know that searching a node that is placed d^{th} depth has $O(d)$ time complexity. If we consider that we have a tree with N (with NIL $N+1$) nodes, there will be $\log n$ level of depth (root's depth is taken as 0) and each depth, there will be 2^d nodes. If we calculate each node's searching complexity:

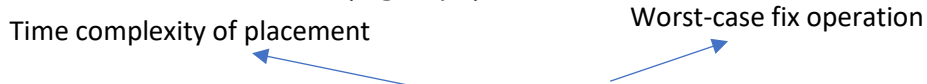
$$\begin{aligned}
 & \begin{array}{ccc} \text{Time complexity} & \text{Number of nodes per depth} & \text{Complexity of any given node} \\ \swarrow & \nearrow & \nearrow \end{array} \\
 T_{average} &= \frac{\sum_{i=0}^{\log(N)} \log(i) * 2^{\log(i)}}{N + 1} \approx \frac{\sum_{i=1}^{N+1} \log(i)}{N + 1} \\
 &\approx \frac{\log(1) + \log(2) + \dots + \log(N + 1)}{N + 1} = \frac{\log((N + 1)!)}{N + 1} \\
 &\approx \frac{(N + 1) * \log(N + 1)}{N + 1} = \log(N + 1)
 \end{aligned}$$

As we can observe from calculations, average case of insertion operation to Red-Black Tree that has N nodes is **$O(\log N)$** .

Insertion Operation:


Worst-Case Analysis:

In Red-Black Trees, the insertion operation consists of two phases that are insertion and fix-up. Actually, the insertion part is always same; it iterates to proper leaf of the tree and connects new node to leaf. It always costs $O(\log N)$ for iteration and $O(1)$ for placement in balanced trees. But the fix-up part changes according to condition of the tree. The while loop in the fix-up code, only iterates when case 1 (uncle of node is red) occurs. For other cases, black is assigned to node's parent and then while loop terminates. For the worst-case, the case 1 can be repeat and it iterates from leaf to root. In case 1, node is assigned to its grandparent, therefore the worst-case complexity is $O(\log(n)/2)$. The total complexity is:


$$T(n) = O(\log n) + O(\log(n)/2) \approx \mathbf{O(\log n)}$$

Average-Case Analysis:

We know that insertion operation guarantees that new node is placed to leaf of the tree. Therefore, the insertion phase always works with $O(\log n)$ time complexity. The fix-up phase complexity can change between $\mathbf{O(1)}$ and $\mathbf{O(\log n)}$. If the second or third case occurs, the while loop directly terminates as I stated in previous part. But for the worst case, it can iterate to root. Therefore, average case formula is:


$$T(n) = \frac{\sum_{i=1}^{\log(n)} \log(n) + i}{\log(n)} \approx \frac{\log(n)^2 + \log(\log(n) !)}{\log(n)}$$
$$\approx \frac{\log(n)^2 + \log(n) * \log(\log n)}{\log(n)} \approx \mathbf{\log(n)}$$

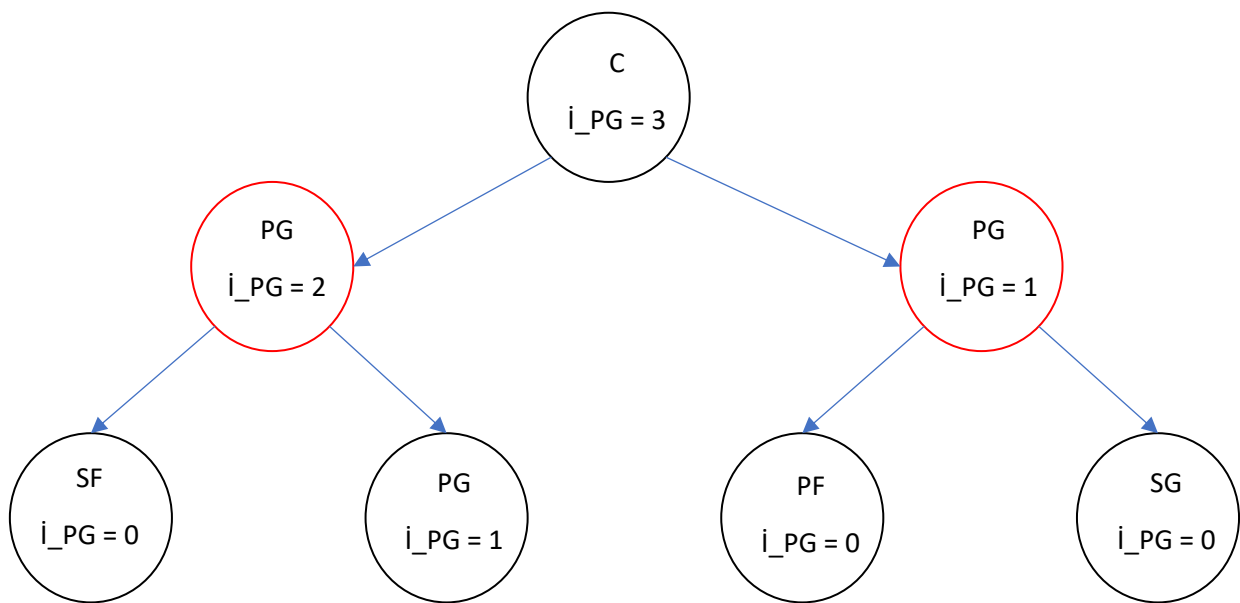
Red-Black Tree Binary Search Tree Comparison

Red-Black Tree is based on Binary Search Tree structure principles but it has more facilities. For average cases, both works with $\mathbf{O(\log N)}$ time complexity for insertion and search operations. But for they differ, when worst-case conditions occur. Binary Search Tree has not **balancing** property and when elements that will be inserted are in sorted order, algorithm adds new elements to same direction (all left or all right child). This condition makes tree some linked-list form. Therefore, its search and insertion operations work with $\mathbf{O(N)}$ time complexity in worst-case conditions. On the other hand, Red-Black Tree has balancing mechanism that tries to keep depth of a tree as $\log N$. This mechanism allows us to keep complexities of tree operations such as insertion and search same as average case. Red-Black Tree has more complex function in terms of implementation but it is more **reliable** and **faster** than Binary Search Tree.

Augmenting Data Structures

To augment a data structure, we must determine our needs, then we must find a way that solves our problem and compatible with our existing data structure.

For given problem, we are expected to keep indexes of given players' position according to their key values. We can calculate number of players in every sub-tree according to their positions. We need to add new parameters that are called i_PG , i_SG , i_SF , i_PF , i_C and they keep number of players that are placed in node's sub-tree according to their positions. To make understanding process easy, I only think about **PG** position and I drew the illustration given below. Implementation of the other positions same as PG examples.



The search process is based on these stored parameters that keep number of players that are placed in sub-tree for given position. We can analyse search function with two parts that are current node's position is **same** with node that is searched or **not**.

If current node's position is intended position, we can say that current node's index is equal to left node's index+1. If searched index is same with current index, it directly returns current node. Also, based on this index value, we can determine our intended node is placed left or right of current node. If we search for higher index value, we branch to right child or vice versa.

If current node's position is not intended position, we can say that current node's index is equal to left node's index. Because, current node is not counted as this position's player. Based on this index value, if we search for smaller or equal index value, we branch to left child; else it branched to right child.

My pseudo-code is given below.

```

PG-Search (node, i):
    if node == NIL: //if it is NIL
        return None //this index does not exist
    if(node.PG): //if current node is PG
        current_index = node.left.i_PG +1 //current index left index+1
        if current_index == i: //if current index is equal to searched index
            return node //returns current node
        else if current_index > i: // if current index is bigger
            return PG-Search(node.left,i) //it searches left subtree
        else: // if current index is smaller
            //it searches right subtree by subtracting current index
            return PG-Search(node.right, i-current_index)
    else: //if current node is not PG
        current_index = node.left.i_PG //current index is equal to left index
        if current_index >= i: //if current index is bigger or equal
            return PG-Search(node.left,i) //it searches left subtree
        else: //if current index is smaller
            //it searches right subtree by subtracting current index
            return PG-Search(node.right,i-current_index)

```

Maintenance is important issue in augmented data structures. Red-Black Tree has four different operations that can damage my indexes, these are left rotation, right rotation, insert and delete.

While **inserting** new node to tree, we know our new node's position. Therefore, while iterating to new node's position, we can **increment** related position index in node's that are parents of new node. In similar way, we can decrement parent's node's position value when we delete a node.

Left and right rotation changes node's order and it can cause a problem with index numbers. Therefore, we have update nodes' indexes after rotation operation for each position value. At the end of the rotation codes, we have to add these operations. If we consider two nodes are rotated, after rotation operation we can directly assign **lower nodes' index value to upper node's index** value. For lower node, we have to **sum its child's index values**. Also, **only** for node's position we have to **add 1** to index value.