Burak Atlamaz

150180067

1) Complexity

If $n_{black}$ is the number of the black nodes and $n_{red}$ in the number of red nodes in red-black tree, the depth of the tree that includes only black nodes (assume that we remove all the red nodes) is O(log $n_{black}$). All the leaves have same depth due to every path from root to leaves have same number of black nodes.

Insertion Worst Case: The distance between root and leaves must be maximum. To increase this distance all of the black nodes have red left_child and right_child if we consider $n_{black}$ is constant.

For example: 1 -> 2 -> 4 -> 8 -> 16

All leaves are black so $n_{red} < n_{black}$

Even if $n_{red} = n_{black}$, total depth of the three is O(log $2.n_{black}$) = O (log $n_{black}$) = O(log n)

Adding node cost O(1) and fixing the double reds cost O(1). In worst case double red cascade in all nodes from root to leaf so add a node fixing all the nodes cost

O(1).O(log n) +O(1).O(log n) = O(log n)

As can be seen above, upper bound for insertion worst case is O(log n)

Insertion Average Case: The distance between root and leaves must be average. $0 < n_{red} < n_{black}$ so depth of three is between O (log $n_{black}$) - is O(log $2.n_{black}$) . Both equations are equal to O(log n) so depth of the tree is O(log n)

Adding node cost O(1) and fixing the double reds cost O(1). In average case double red cascade in some nodes from root to leaf. Total cost is

O(1).O(log n) + O(1).O(log n). k = O(log n)

->k is constant between 0-1. k is the possibility of have double red nodes and for each case total cost is O(log n) so upper bound for average case is O(log n)

Search Worst Case: The node that searched must be in leaves so the program search the node from root to the leaves which is equal to depth. The red-black tree must have maximum depth which is equal to O(log n). Returning the node which is searched cost O(1) so total cost is

O(1) + O(log n) = O(log n)

Search Average Case: The node that searched is between root and leaves. Average depth is O(log n) and in some point the node will be found. If reaching the leaves takes k steps and the program finds the node in a steps and returning the node which is searched cost O(1), total cost is

$\frac{a}{k}$.O(log n) + O(1) = O(log n)

2) RBT vs BST

In all situations, Red-Black Tre has the same number of black nodes from root to the leaves. In Binary Search Tree does not have that condition. Because of that Red-Black Tree is more balanced.

If the key datas of the nodes are already sorted, while adding them in Binay Search Tree acts like an array; if new nodes are all smaller the old ones right childs are NULL, else if new nodes are all bigger the old ones left childs are NULL and the shape of the tree is becomes like a line. Searching in the tree cost O(n) because of the line has n element and in every process only skip one of this element, which is bigger than O(log n). Insertion and deleting nodes are also cost O(n).

But in Red-Black Tree all the leaves has same depth and it makes positive difference. Both search, insertion and delete cost O(log n) in worst case. In conclusion, using Red-Black Tree can be minimize the running time.

3) Augmenting Data Structures

We have to sort each position on their own. So key point cannot be names. Adding extra attribute which holds number of node will be the key point

Pseudecode:

Struct

```
struct node {
    string names[5];
    int stats[5][3];    //reb, assist, point
    int key;            //which node
    node* parent, * left, * right;
};
```

RB-INSERT

```
RB-INSERT(T, names, position, key)
    x = TREE-INSERT(T, names, position, key)//Insert by key number, only the name of pos, not all names
    color[x] ← RED ▷ only RB property 3 can be violated
    while (x != root[T[k]) and color[p[x]] = RED
        do if p[x] = left[p[p[x]]
            then y ← right[p[p[x]] ▷ y = aunt / uncle of x
            if color[y] = RED
                then ⟨Case 1⟩
            else if x = right[p[x]]
                then ⟨Case 2⟩ ▷ Case 2 falls into Case 3
                 ⟨Case 3⟩
        else ⟨“then” clause with “left”and “right” swapped⟩
    color[root[T]] ← BLACK
```

RETURN-NAMES

```
RETURN-NAMES(T, names, key)
    traverse = root
    while(traverse && key[traverse] != key) //Search for i'th node
        if key[traverse] < key  //key is bigger
            left[traverse]
        else                    //key is smaller
            right[traverse]
    if traverse                 //key found, return names
        string names[5] = names[traverse]
        return names
    else                        //key cannot found
        print NO_NAMES
        return NULL
```