Algorithms HW 3 Report

İsmail ÇETİN
150 150 065
2-Ulg

Part I:

2. Compare RB Tree with Standard Binary Tree:

Red-Black trees are a special kind of binary search tree in which the node of trees contain extra data which is interpreted as the color of node. Each node is either black or red. Every node has 2 children at most for both RB trees and binary search trees.

When we use Standard Black Trees, searching in a tree is O(n), along with insertion and deletion. This is not the case with RB trees because the height is balanced and time complexity is reduced to O(log n) for search, again along with insertion and deletion.

The worst case for BST happens when the given input is sorted and it is similar to linked list in that case. However as mentioned, RB trees are self balancing and prevent nodes from forming one-way structure and this mechanism balances the tree.

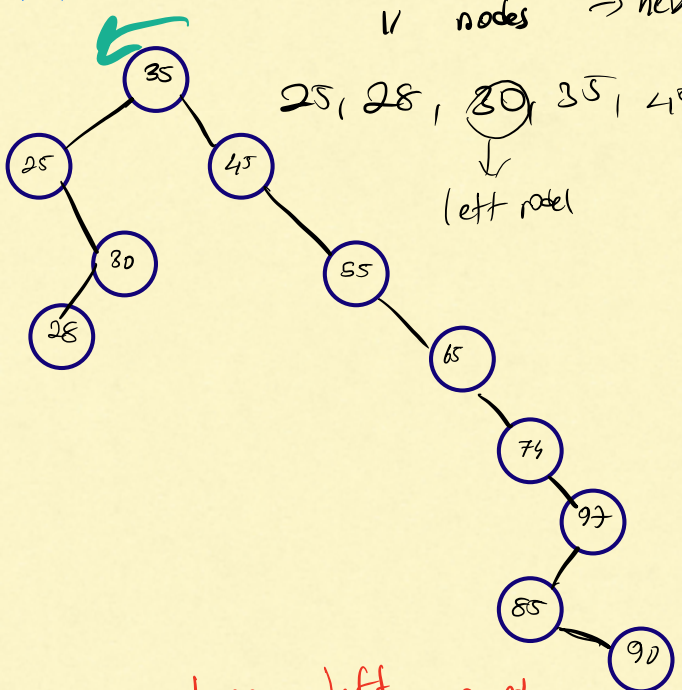
To summarize, RB tree is a more neat and balanced type of binary search tree.

1. Balance The given tree with left and right rotations. and obtain shortest tree.

11 nodes → height will be 3.

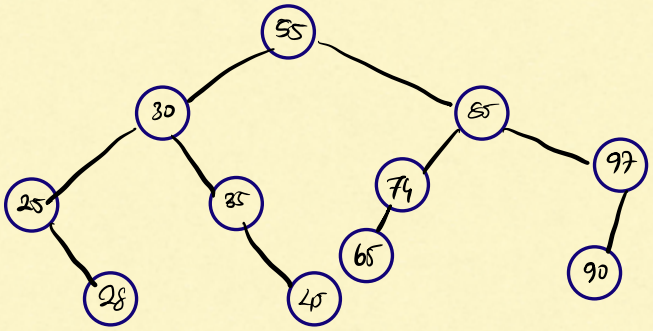
25, 28, 30, 35, 45, 55, 65, 74, 85, 90, 97

left node | root | right node

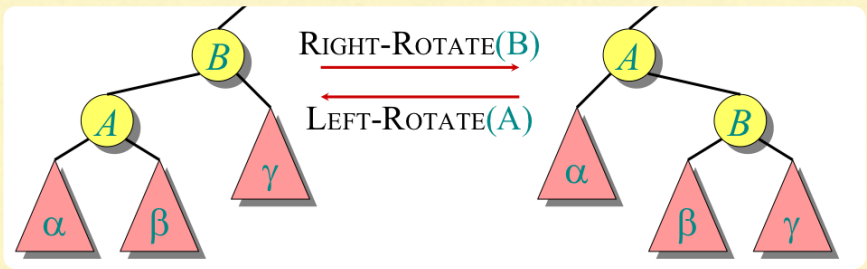


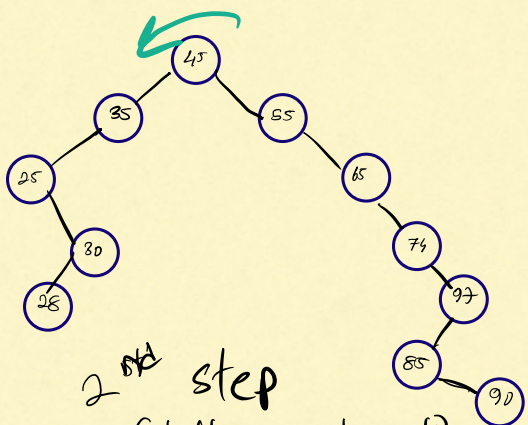
turn left around root

Ideal tree

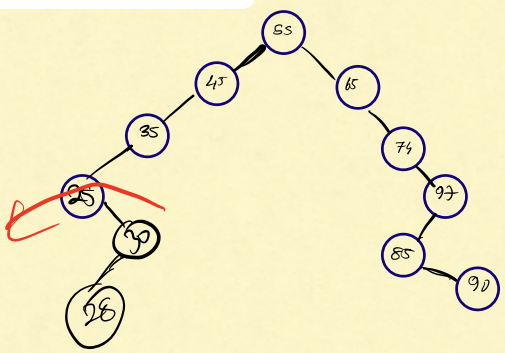


First bring 55 to root and continue from that point. To do it we need 2 left rotation

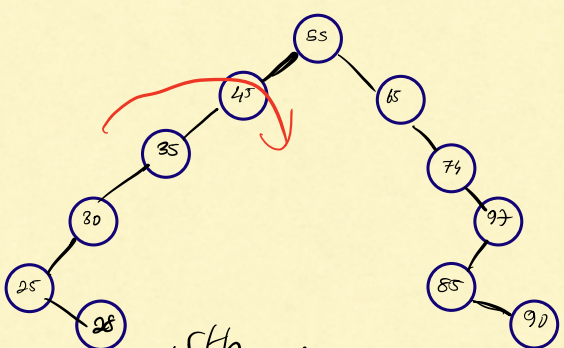




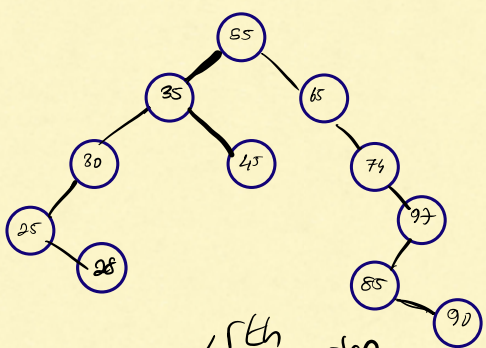
2nd step
(left around root)



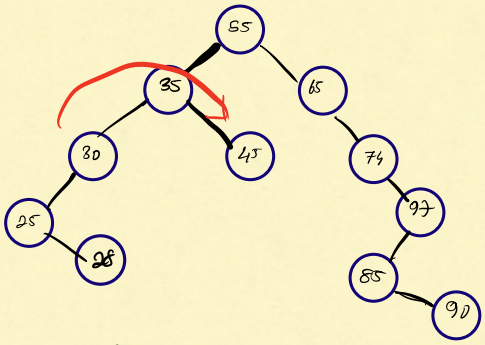
3rd step
(25-30 left)



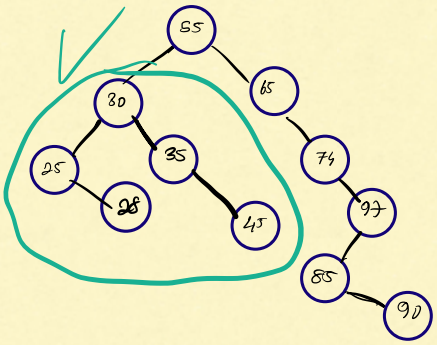
4rth step
(35-45 right)



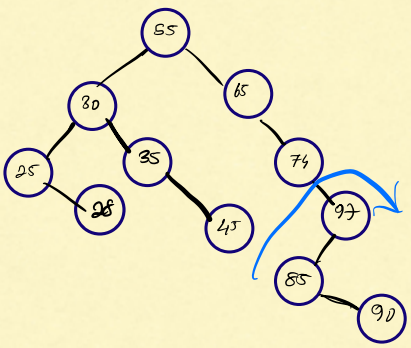
4rth step
(35-45 right)



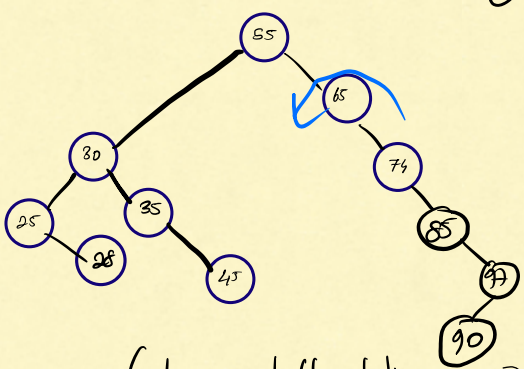
(30 - 35 right)



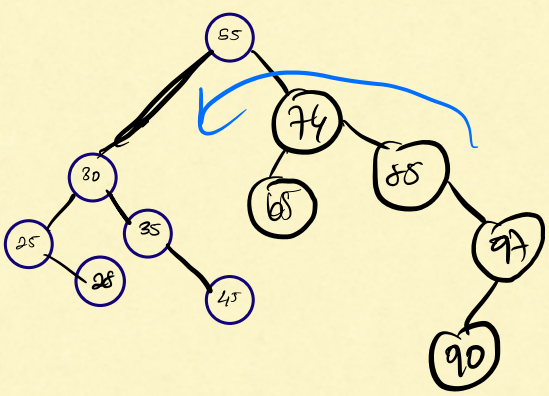
left subtree is ok



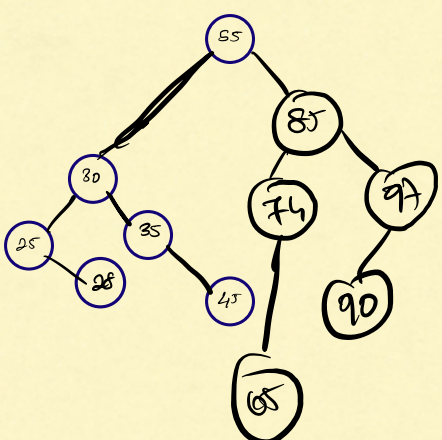
(85 - 97 right rotate)



(two left rotation
1st round 65 - 74
then 74-85)



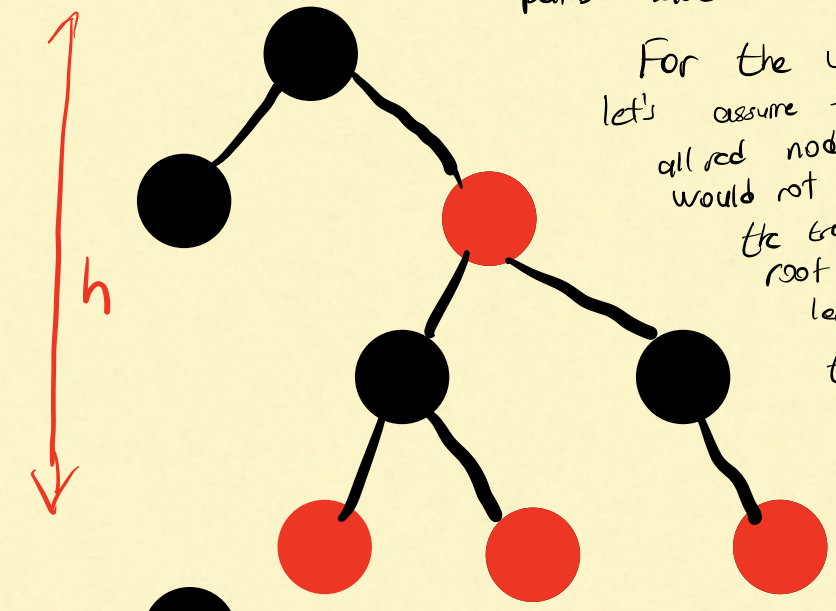
then 74 - 85



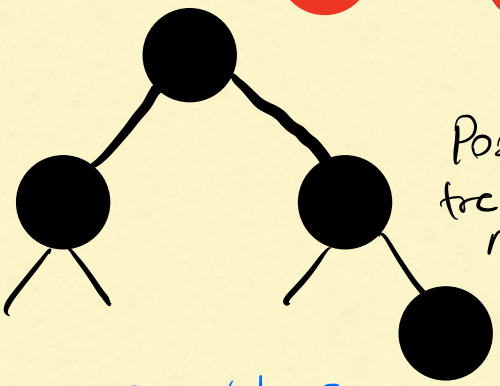
(last step)

3. Insertion Worst Case

In essence, the time complexity of search, insertion and deletion depends on the height of a tree. In RB-tree data structure all simple paths have same black-height.

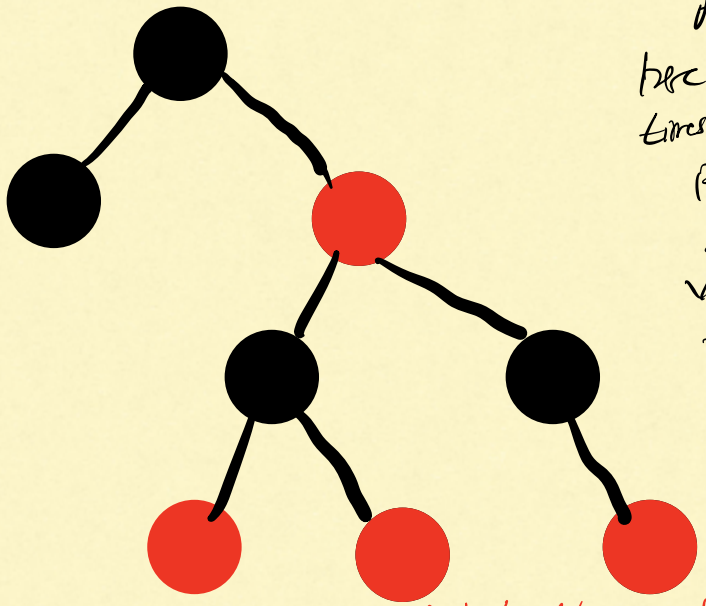


For the worst case of RB tree let's assume the following logic. Imagine all red nodes are removed. This would not change the black height of the tree. Also black height of the root (or black depth of the leaves) is now height of the tree. Thus we now have a complete binary tree where height is $O(\log n)$. Now populate some path with red nodes as many as possible. In the worst case, the longest path would be 2 times of shortest path, because red nodes can't have red children



Populate the tree with as many red nodes as possible

Complete Binary Tree



one path is populated the most possible number of red nodes.

As can be seen from here longest path is 2 times the length of shortest path.

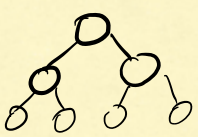
So $h \leq 2 \cdot \log(n+1)$

Worst case of RB tree therefore is:

$O(2\log(n+1)) \in O(\log n)$

for both insertion deletion and search.

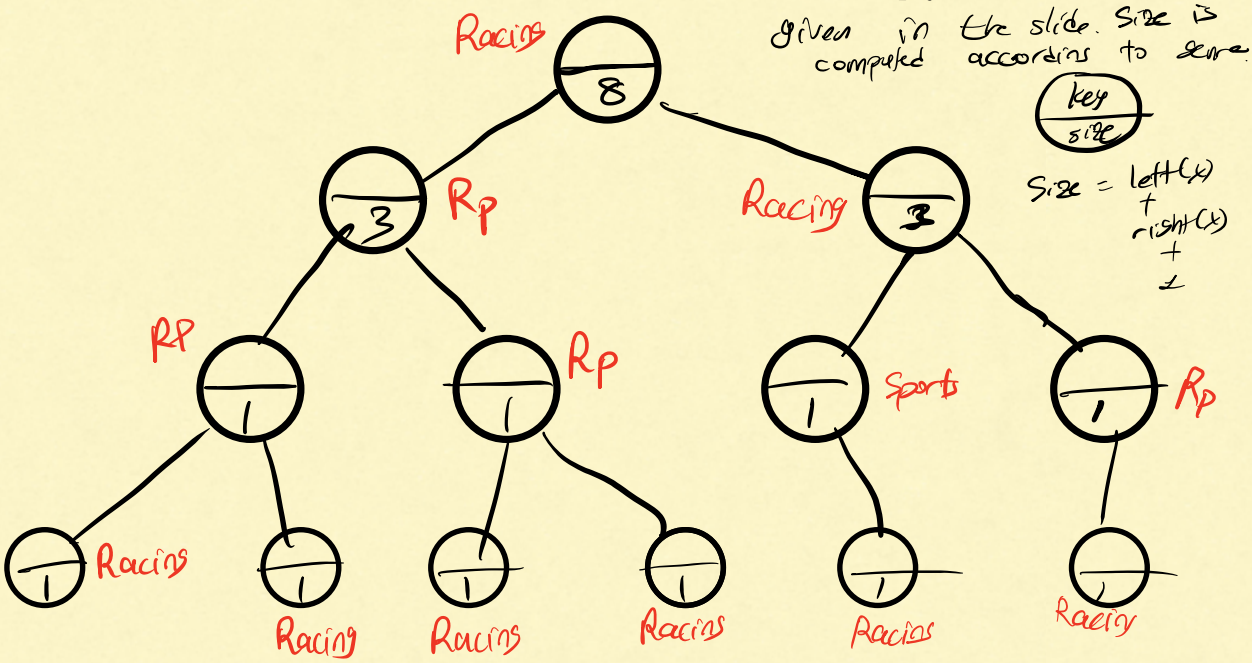
The best case of RB tree occurs when the tree is fully balanced. In this case the height is $h \geq \log n$ so $\Omega(\log n)$. Because both best and worst case is bounded by $\log n$ we can easily say that $\Theta(\log n) \Rightarrow$ average.



Augmenting Structures

We are given genre, and number i. We need to find ith sports, action, inside the genre.

From my vantage point, every genre can be considered seperately within the tree. Add a new attribute to node called size as given in the slide. Size is computed according to genre.



key
size

Size = left(x) + right(x) + 1

We need to write OS-Select and OS-Rank functions.

ith smallest element rooted at x.

We need to write rank finder. Rank is the number of children in left subtree with the genre same as our genre.

```
rank_finder (x, genre)
    if x == NIL          // if reach leaf
        return 0         // return
                         count = rank
    if x.left != NIL     // if left node exists
        if x.genre = genre and genre is the same as our genre
            count += 1   // increment rank
        count += rank_finder (x.left)
    if x.right != NIL    // if right exists
        count += rank_finder (x.right)
    return count         // return rank
```

OS-Select with Genre : (x, i, genre) (x: node, i: ith smallest)

if (x.genre == genre) ; if nodes genre is the same we're looking for.
  k ← rank-finder (x, genre) +1 ; find its rank
  if i == k then return x ; check if it is our node
  if i < k ; if node branch accordingly
    then return OS-Select with Genre (left[x], i)
    else return OS-Select " (right[x], i-k)

else
  if x == NIL return NIL ; to prevent dangling pointer
  if OS-Select with Genre (x.left, i) ; go to left
    then OS-Select with Genre (x.left, i)
  if OS-Select with Genre (x.right, i)
    then OS-Select with Genre (x.right, i)