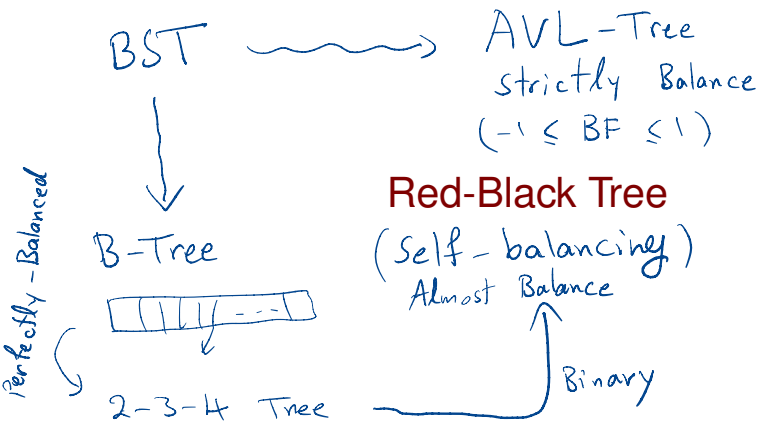
Data Structures and Algorithms

Mohammad GANJTABESH

Department of Computer Science,
School of Mathematics, Statistics and Computer Science,
University of Tehran,
Tehran, Iran.

mgtabesh@ut.ac.ir

Dept. of Computer Science
University of Tehran



Data Structures and Algorithms

Undergraduate course

Mohammad GANJTABESH
mgtabesh@ut.ac.ir

Dept. of Computer Science
University of Tehran

Motivations

2-3-4 Tree —Binary→ Red-Black tree

search : $O(\log n)$

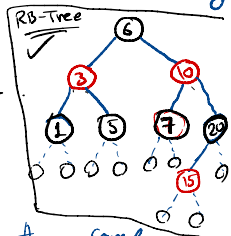
$2 \log n$ | $\approx \log n$

insertion : $O(\log n)$

deletion : $O(\log n)$

Basic Definitions/Rules

Red-Black tree is a binary search tree with the following rules:

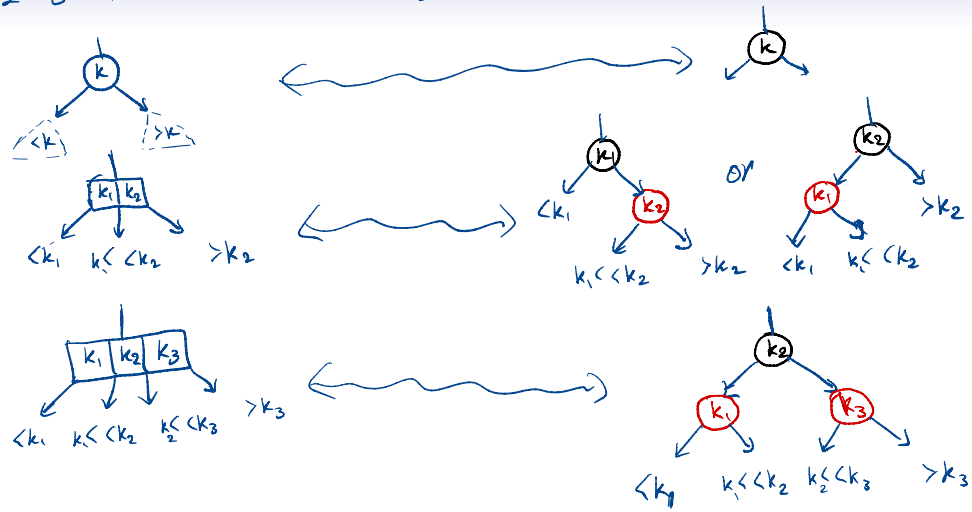


- Every node is either RED or Black
- Root is always Black.
- New insertions are always RED
- Every path from the root to leaf has the same number of Black nodes.
- No path can have two consecutive RED nodes.
- Nulls are Black.

---

* Fixing the tree if the above rules are violated.
* Fixing is done by Rotation and Color flips.

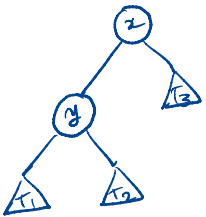
2-3-4 Tree vs Red-Black Tree



Rotations and Color Flippings

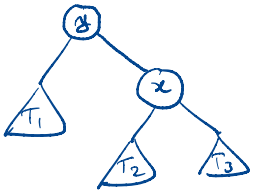
(exactly the same as AVL-tree)

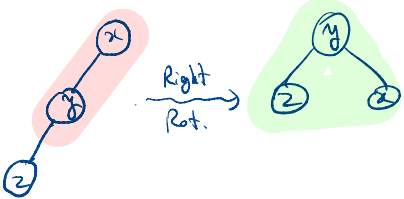
new operation

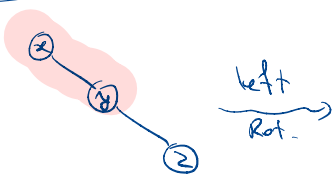


Left-Rotation

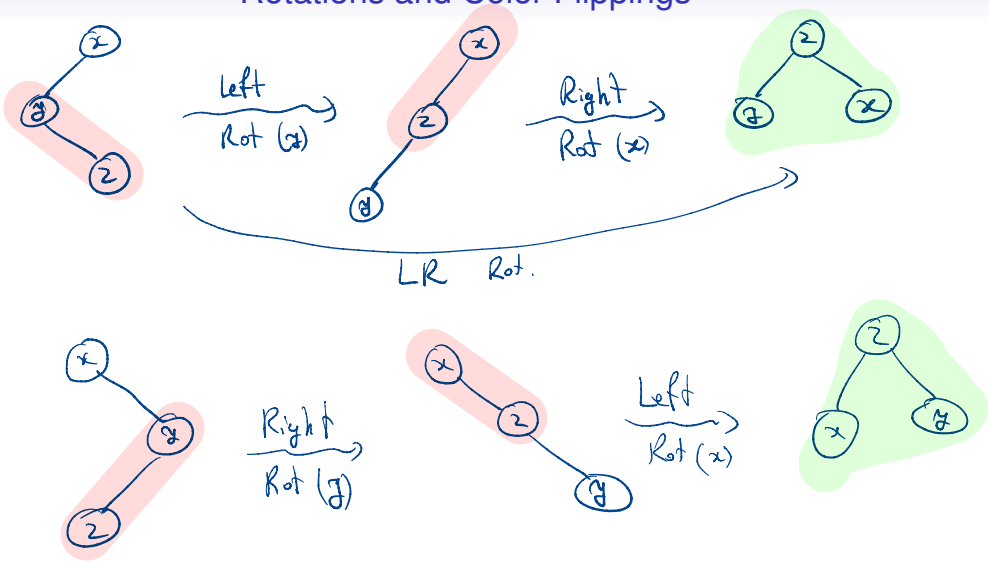
Right Rotation





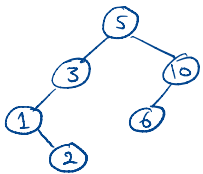


Rotations and Color Flippings



Red-Black Tree Operations : Insert

(1) Insert a new node as we done in BST
(new node is always RED)



k=2

- if the node is root, flip its color.

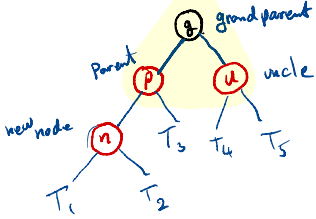
- else, check the parent's color :
  - if the color is Black, done.

  - else (if the parent's color is Red),
    check the color of the node's uncle!

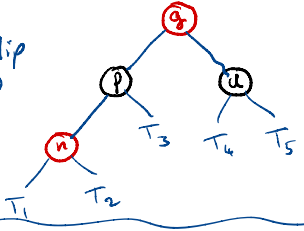
We have several cases

Red-Black Tree Operations : Insert

- Case 1 : if the node's uncle is Red.

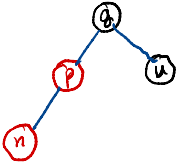


color flip

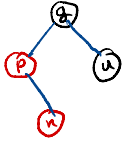


repeat this process for grand parent.

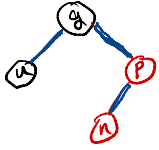
- Case 2 : if the node's uncle is Black :



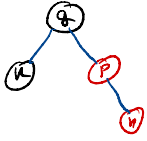
2.1



2.2



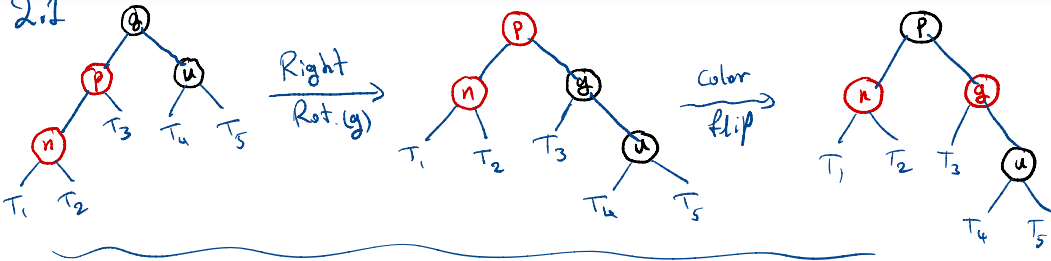
2.3



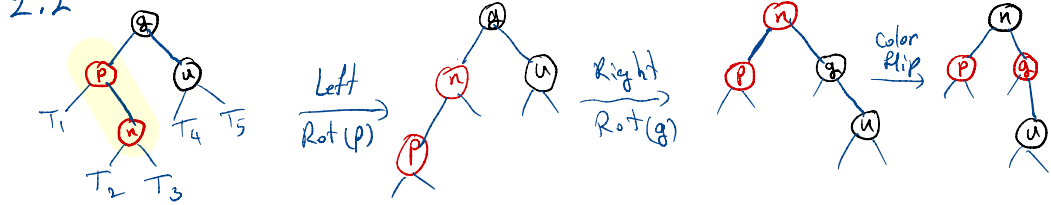
2.4

Red-Black Tree Operations : Insert

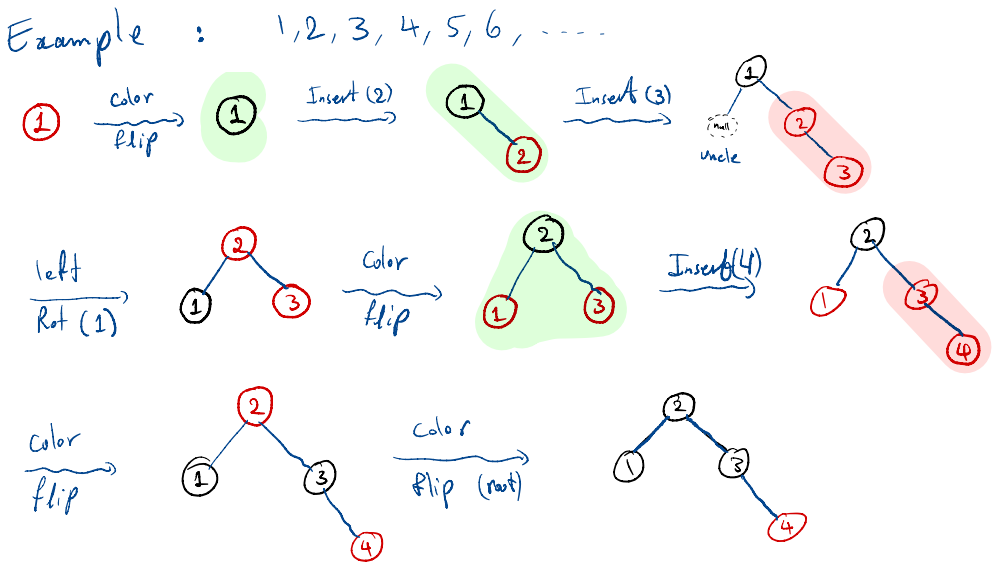
2.1



2.2

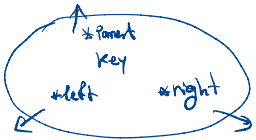


Red-Black Tree Operations : Insert



Red-Black Tree Implementation

```
struct Node {
    int Key;
    Node *left, *right, *parent;
    bool isBlack;
    bool isLeftChild;
    Node (int k) {
        Key ← k;
        left ← right ← parent ← null;
        isBlack ← false;
        isLeftChild ← false;
    }
}
```




```
class Red-Black-Tree {
    Node * root;
    int size;
    Red-Black-Tree () {
        root ← null;
        size ← 0;
    }
    // methods
}
```

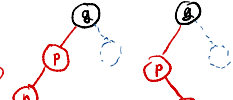
Red-Black Tree Implementation


```
public insert (int k){
    Node *node ← new Node(k);
    if (root = null){
        root ← node;
        root.isBlack ← true;
        size ← size +1 ;
        return ;
    }
    insert (root , node);
    size ← size +1 ;
}
```

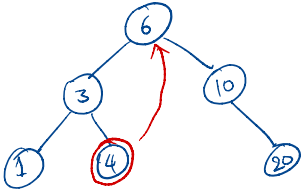
```
private insert (Node * parent , Node *
    if(node.key > parent.key){
        if(parent.right = null){
            parent.right ← node;
            node.parent ← parent;
        }
        else{
            insert (parent.right , node);
        }
    }
    else { if (parent.left = null)
            parent.left ← node;
            node.parent ← parent;
            node.isLeftChild ← true;
        }
        else{
            insert (parent.left , node);
        }
    }
    Fix (node);
}
```

Red-Black Tree Implementation

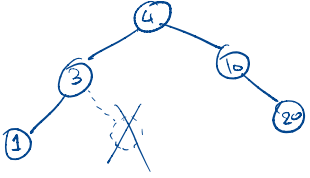
```
Fix (Node * node){
    if (node = root) return;
    if ( node.isBlack = false and node.parent.isBlack = false)
        Fix_Tree (node);
    Fix (node.parent);
}

Fix_Tree (Node * node){
    if (node.parent.isLeftChild) {
        if(node.parent.parent.right = null or node.parent.parent.right.isBlack)
            Rotation (node);
        if(node.parent.parent.right != null) {node.parent.parent.right.isBlack ← true}
        node.parent.parent.isBlack ← false;
    } node.parent.isBlack ← true;
    else{
```


delete :

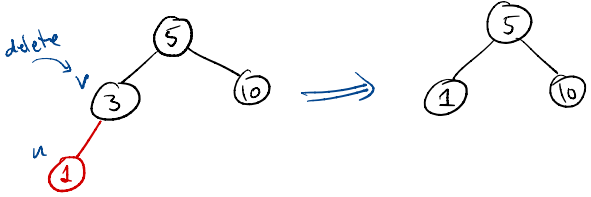


Delete(6)



Let v be the node to be deleted and u be the node to be replaced.

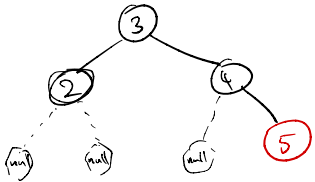
(1) if either v or u is **Red** :



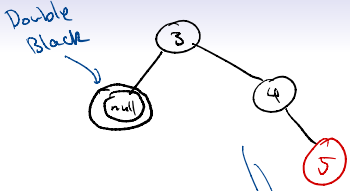
(2) if both u and v are **Black**.

cause Double Black

(2)



Delete (2)



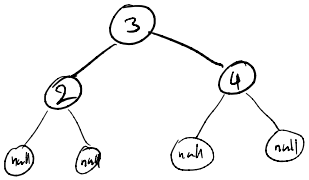
case 1) if sibling node is black and at least one of its children is Red

⟹ rotation and recoloring.

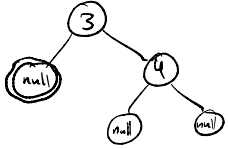
rotation

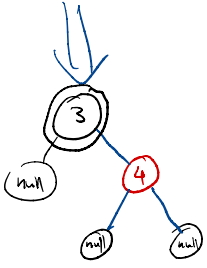
★ Consider four cases for Rotation.

case 2) if the sibling is Black and both of its children are Black. ⟹ perform recoloring and check the parent.

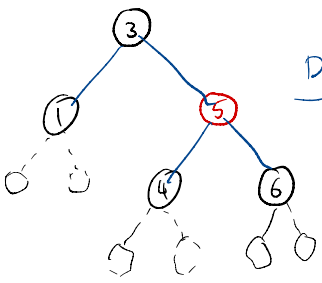


Delete (2)

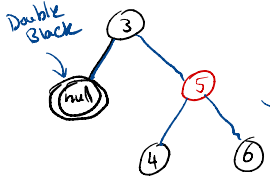




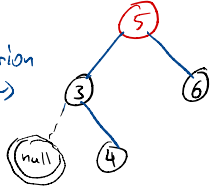
Case 3) if sibling is red ⟹ perform Rotation to move the sibling up and recolor the sibling and parent.



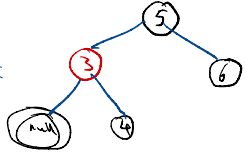
Delete (1)



Rotation



recolor



check the previous cases (1, 2)