SWE210 - Data Structures and Algorithms - Fall 2025

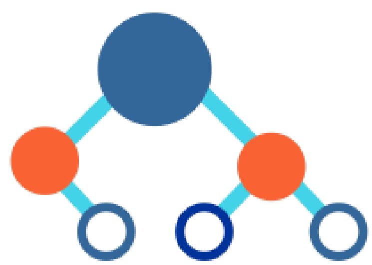
Instructor: Dr. Ashraf AbdElRaouf, Dr. Osama El-Ghonimy, and Assoc. Prof. Sarah Nabil

Eng. Yasmin Kandil

Eng. Mohamed Salah

Eng. Karim Hesham

Eng. Maya Waleed

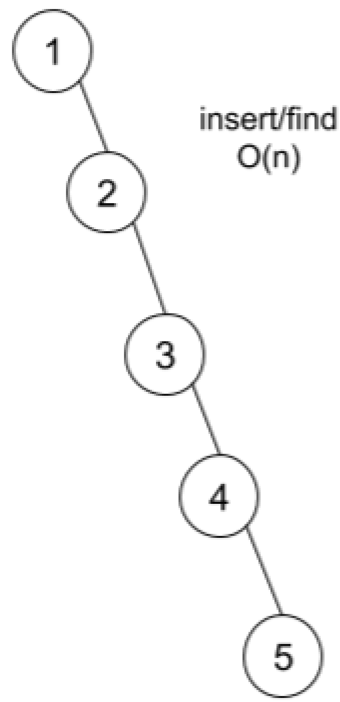


Data Structures

Lab #9

Problem with normal Binary Search Tree

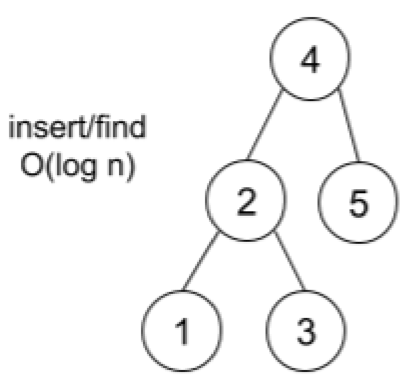
Theoretically, Searching and inserting in a BST takes O (Log n), but sometimes the BST becomes unbalanced (where one branch is a lot longer than other branches) that inserting/searching the tree takes O (n).
For Example: Inserting 1, 2, 3, 4, 5 into a tree in that order.



Now, if we want to insert or search the tree, we will have to do n operations in the worst-case.

Solution:

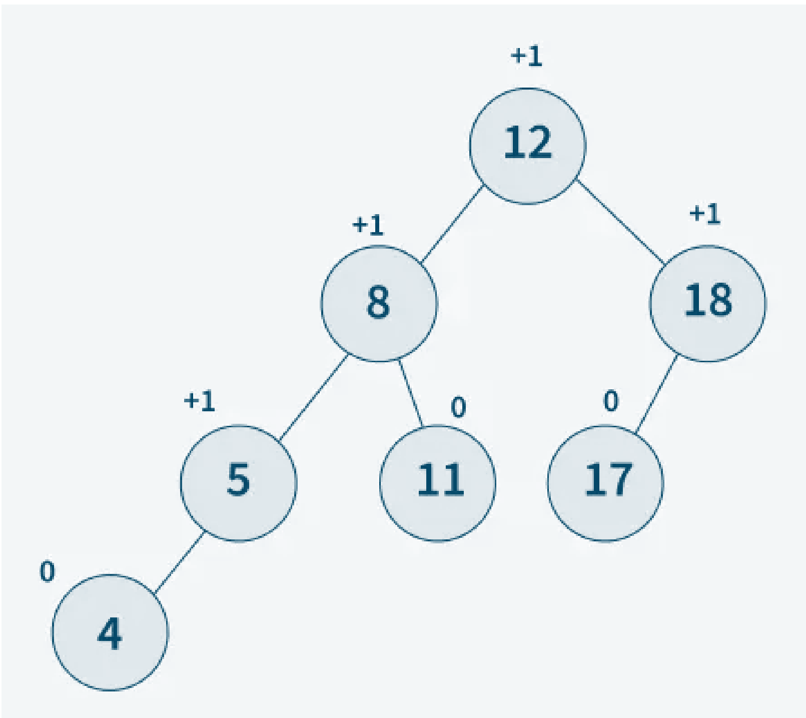
Use a **self-balancing BST**, that re-order their nodes after each insertion or deletion to maintain balance between the longest branch and the shortest branch and ensure **O (Log n)** complexity.



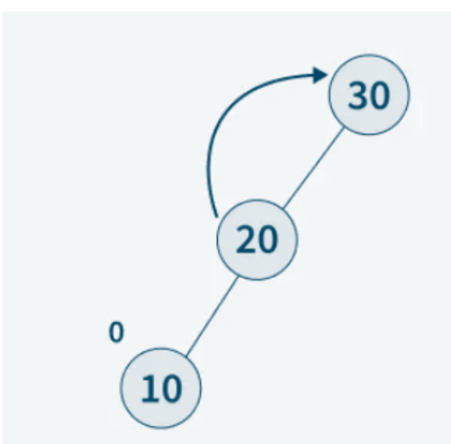
1 - AVL Tree

A self-balancing BST where the difference between heights of left and right subtrees for any node cannot be more than one.

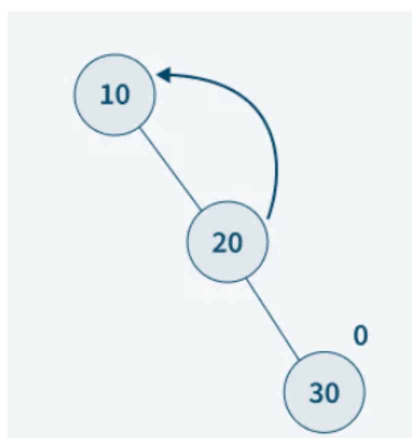
Balance Factor = left subtree height - right subtree height (accepted range [-1, 1])



Balancing the tree happens on most insertion and deletion operations through rotations of nodes.



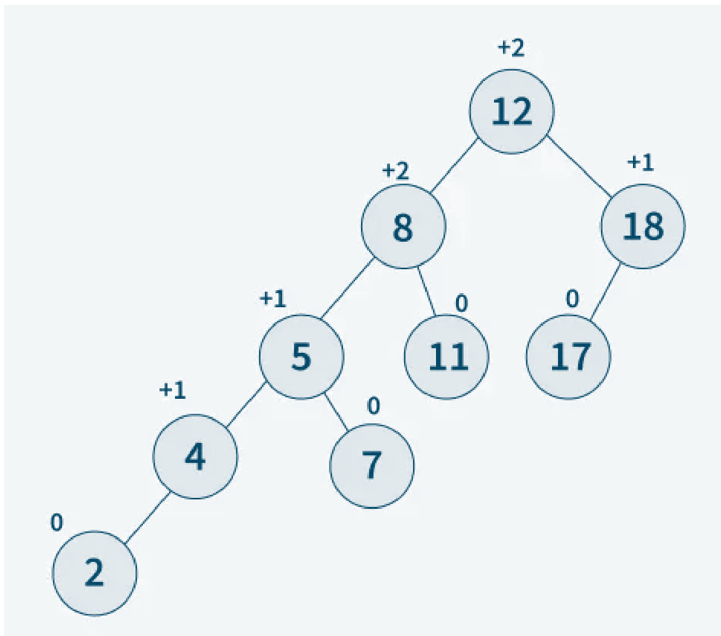
Right Rotation



Left Rotation

Exercises:

1 - Is this tree balanced or not? Why? And if it is not balanced then how to balance it?



2- Draw the following trees and show balance factors for each node before and after each rebalancing:

I. 50, 45, 75, 65, 70, 35, 25, 15, 60, 20, 41, 30, 55, 10, 80
II. 11, 22, 33, 44, 55, 66, 77, 88, 99
III. 1,2,3,4,5,6,7,8

Pros and cons of AVL:

Pros

I. AVL trees can self-balance themselves and therefore provides time complexity as **O (log n)** for **search**, **insert** and **delete**.
II. AVL are very **fast in searching** compared to other self-balancing trees.
III. AVL trees are relatively less complex to understand and implement compared to other self-balancing trees.

Cons

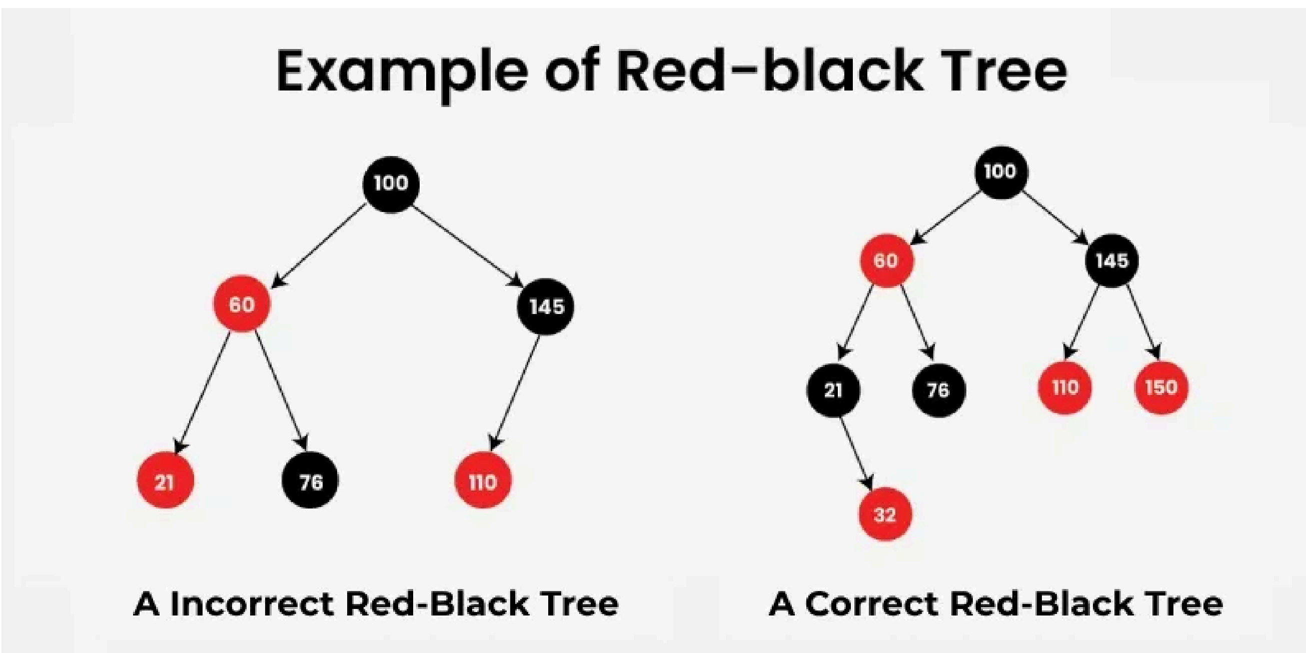
I. They are difficult to implement compared to normal BST.
II. They have a very strict balance.
III. They provide complicated insertion and removal operations as more rotations are performed.

2 – Red-Black Tree

A self-balancing BST where the height of the tree is never beyond O (Log n). Each node has an additional attribute: a color, which can be either **red** or **black**.
The colors are used to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.

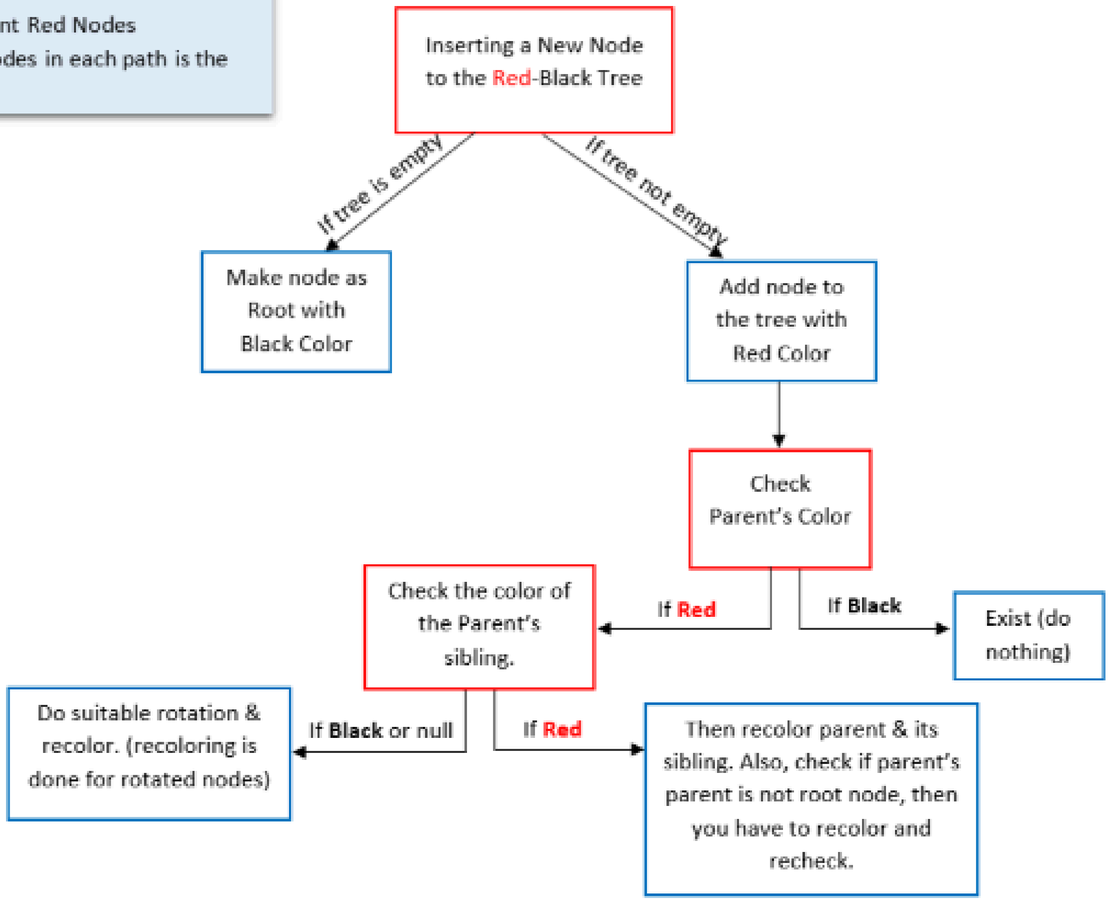
Properties of a Red-Black Tree:

1. **Root Property**: The root of the tree is always black.
2. **Red Property**: Red nodes cannot have red children.
3. **Black Property**: Every path from a node to the leaves has the same number of black nodes.



Red-Black Trees has the following properties:

- Root is black
- No two Adjacent Red Nodes
- No. of black nodes in each path is the same.



Exercises:

1. Insert the following nodes in Red-Black Tree: 53, 27, 75, 25, 70, 41, 38, 16, 59, 36.
2. Construct Red-Black Tree of the following nodes: 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70.
3. Insert the following nodes into a Red-Black Tree: 40, 20, 60, 10, 30, 50, 70, 5, 15, 25, 35

Pros and cons of Red-Black Tree:

Pros

1. Red-Black trees can self-balance themselves and therefore provides time complexity as **O (log n)** for **search**, **insert** and **delete**.
2. Red-Black trees have **Less rotations** and **less strict** on insertion and deletion compared to AVL trees.
3. Red-Black trees are **widely used** and are a popular choice for implementing various data structures, such as maps, sets, and priority queues.

Cons

1. **More complex** than other balanced trees (Compared to simpler balanced trees like AVL trees)
2. Maintaining the Red-Black Tree properties adds a **small overhead** to every insertion and deletion operation.