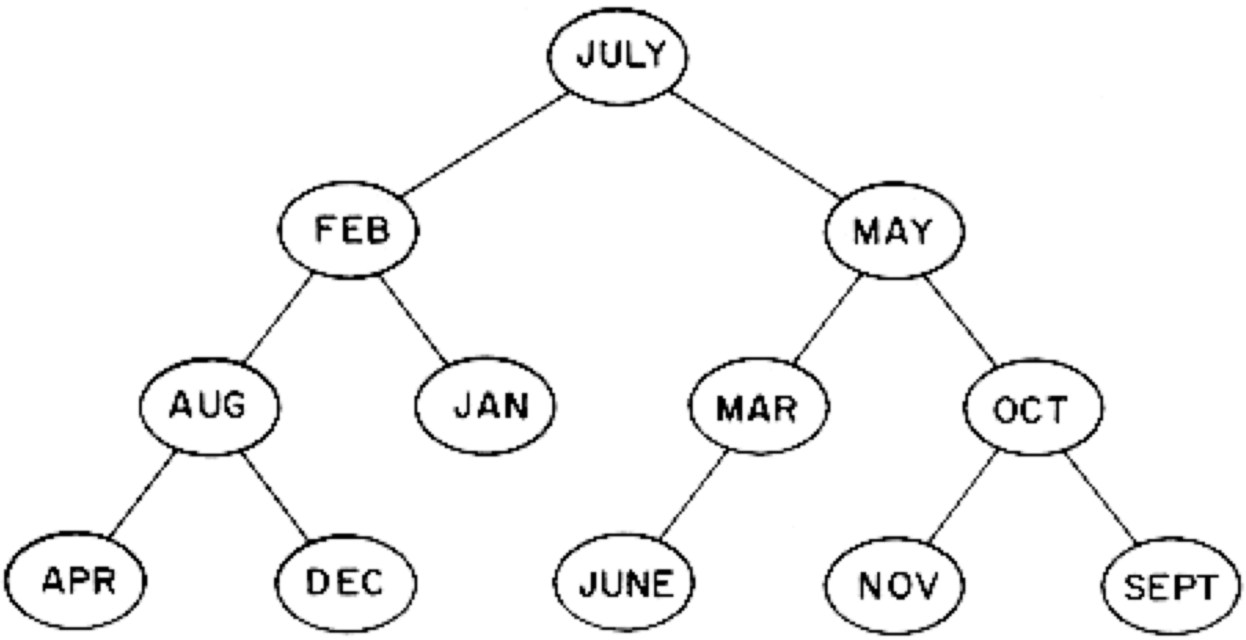
Binary Search Tree and AVL Tree



Is this a binary search tree?

- Is this the only BST with the months of the year?

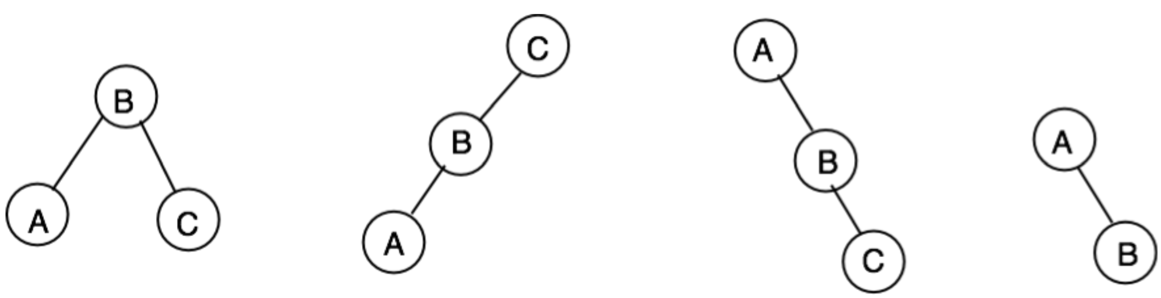
- Can you draw three binary search trees with the keys 12, 41, 9, 55, 36?

AVL trees

(Inventors G.M. Adelson-Velsky and E.M. Landis)

An AVL tree satisfies the height-balance property:

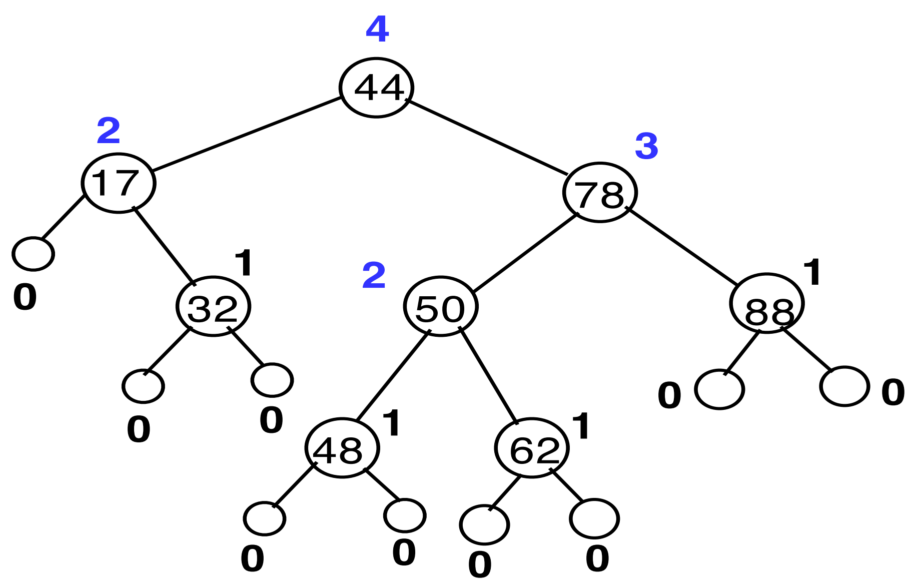
For any tree rooted at an internal node n, the heights of the left and right subtrees differ by at most 1.

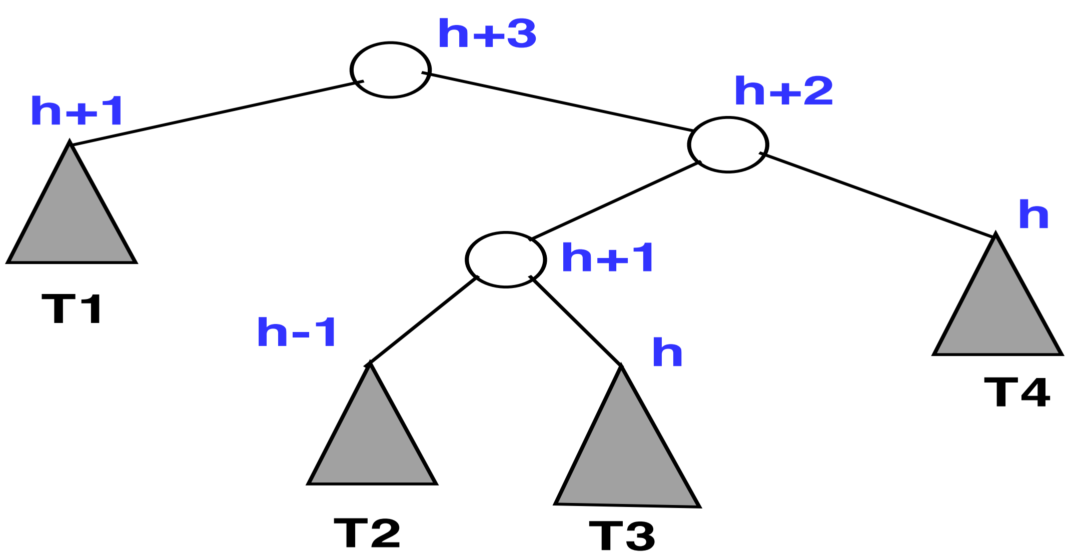


(a) (b) (c) (d)

(a)(d) satisfy height-balance property, (b) and (c) don't

Examples of AVL Trees





Valid AVL trees - satisfy height-balance property

Height of an AVL tree

Is **h= O(log n)** for AVL trees too? **Yes! Why?**

Let n(h) be the smallest number of internal (i.e. non-leaf) nodes of height h in an AVL tree. Then

$n(1)=1, n(2)=2$, and

$n(h)=n(h-1)+n(h-2)+1$

So, $n(h)>2.n(h-2)$, and $n(h)>2^i.n(h-2i)$.

Substitute $i=\frac{h-1}{2}$. This leads to

$n(h)>2^{h-1/2}.n(h-(h-1))=2^{h-1/2}$

Therefore, $\log n(h)>\frac{h-1}{2}$

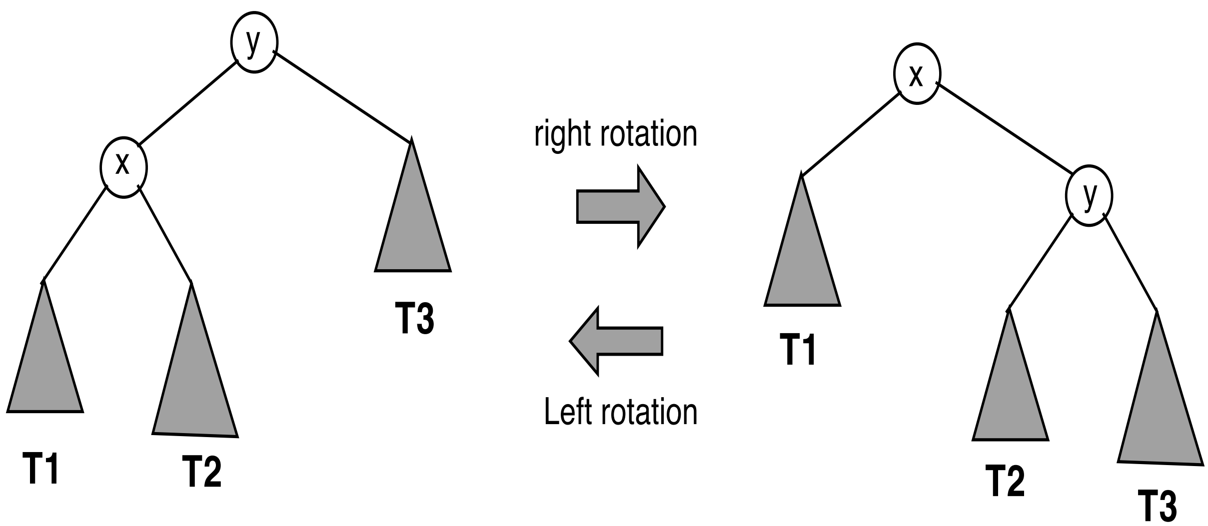
So $h<1+2\log n(h)$, and the number of leaves ≤ 1+number of internal nodes

.

Insertion algorithms for AVL Trees

1. First insert the key w into the BST. If it maintains the height-balance property then fine. Otherwise we need to rebalance it.

3. Re-balance the tree by performing appropriate **rotations** as shown below to repair the tree.



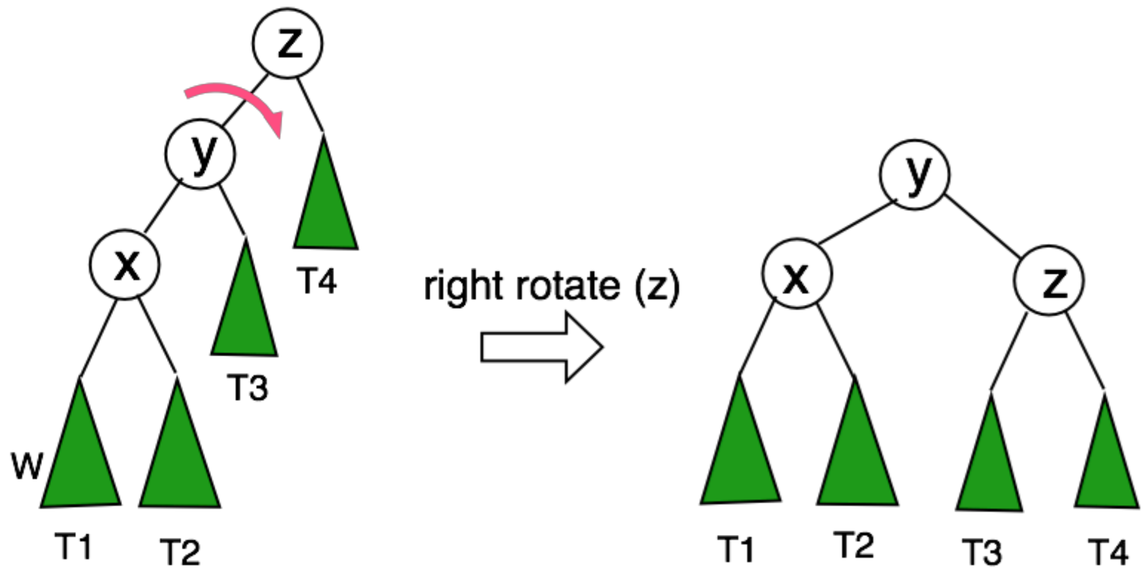
Rotation Operations for rebalancing

Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z, and x be the grandchild of z that comes on the path from w to z.

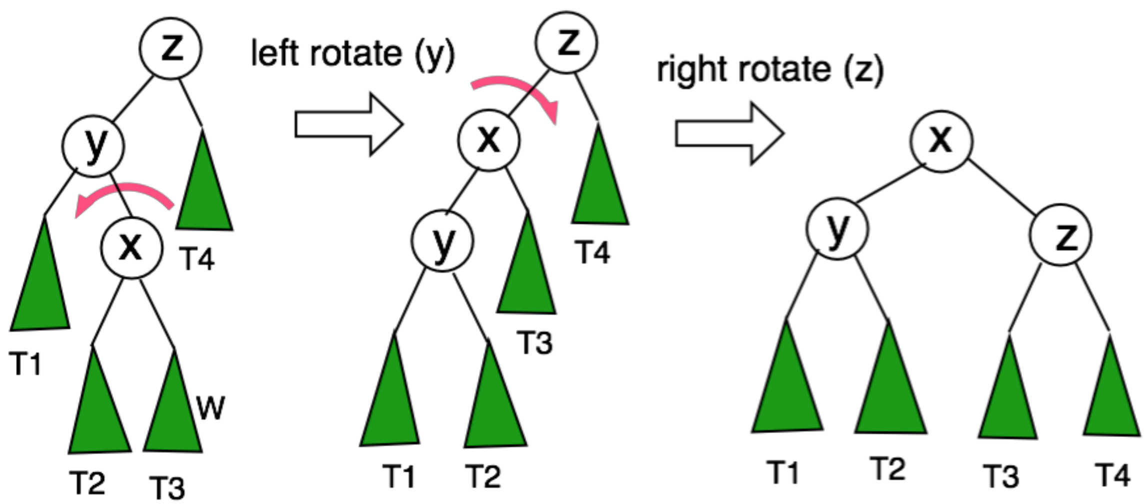
There can be four possible cases that need to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

Four different cases

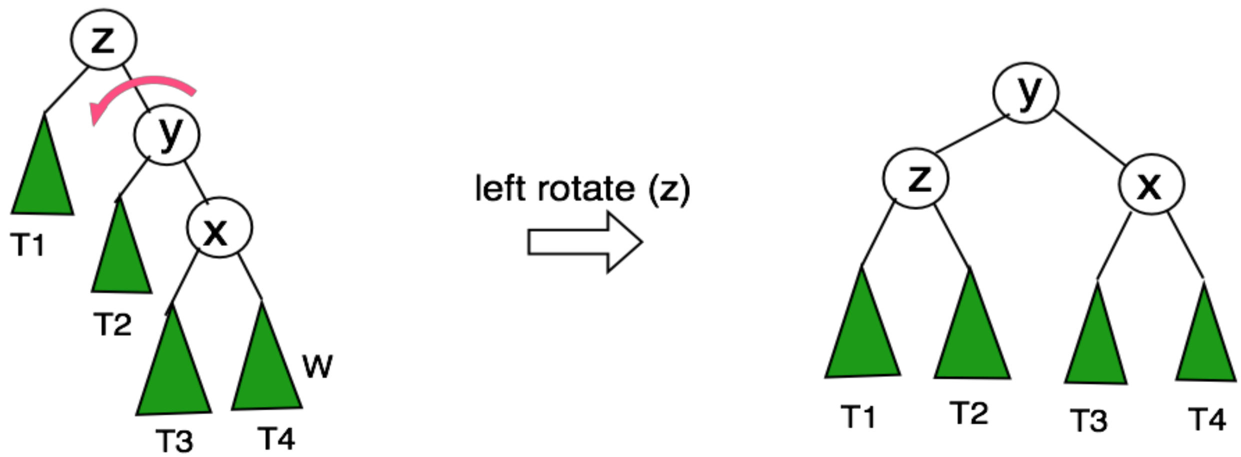
1. y = left child of z, x = left child of y (Right rotation)
2. y = left child of z, x = right child of y (Left Right)
3. y = right child of z, x = right child of y (Left rotation)
4. y = right child of z, x = left child of y (Right Left)



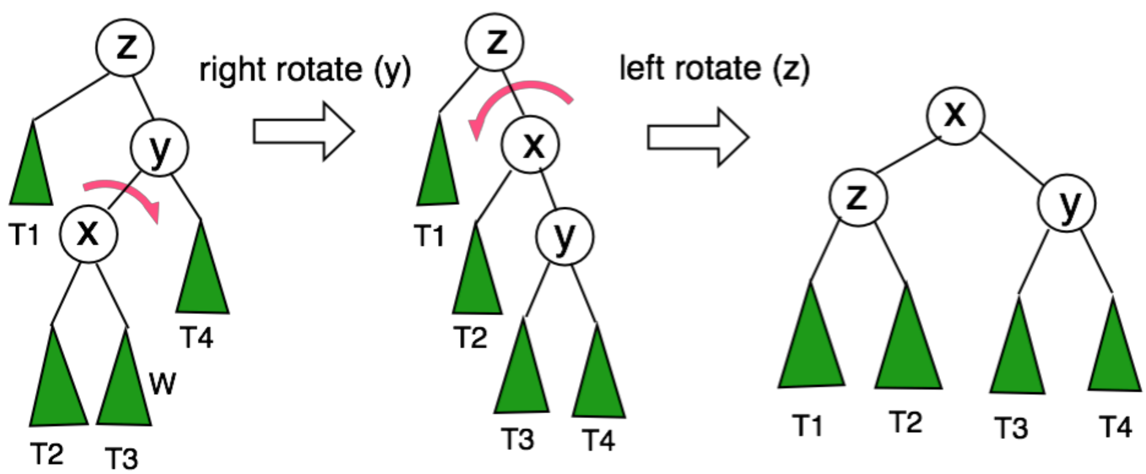
y = left child of z, x = left child of y (Right rotation)



y = left child of z, x = right child of y (Left Right)

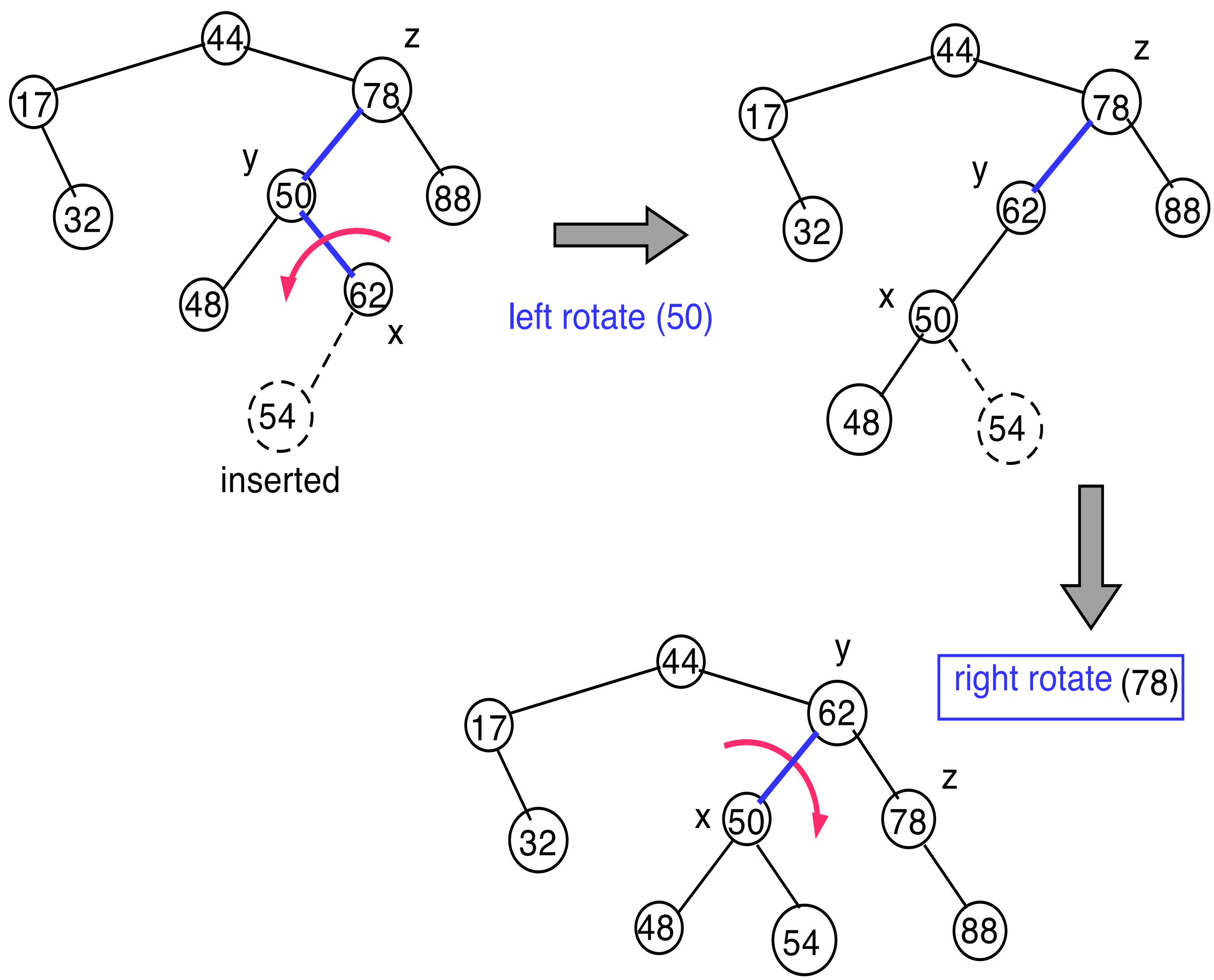


y = right child of z, x = right child of y (Left rotation)



y = right child of z, x = left child of y (Right Left)

Here is an example



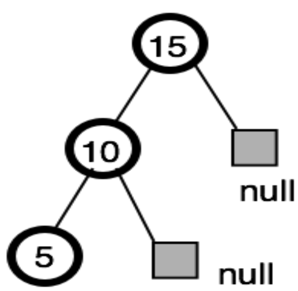
For the delete operation, follow the usual delete algorithms for BST and then, if needed, restore the height-balance property.

Red-Black trees

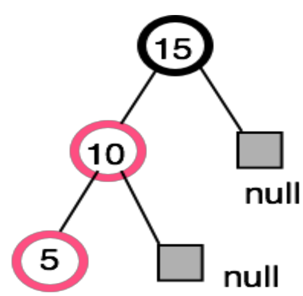
Another approach to keeping a binary search tree balanced. Contains two types of nodes: **red** and **black.** By definition, all ***null entries*** (non-existent children) are **black.** Maintains **three invariants:**

1. The **root** is always black.
2. A **red** node always has **black** children.
3. The number of black nodes in any path from the root to a leaf (a. k. a. **black depth**) is the same.

The following are NOT red-black trees. Why?

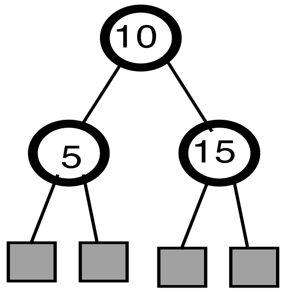


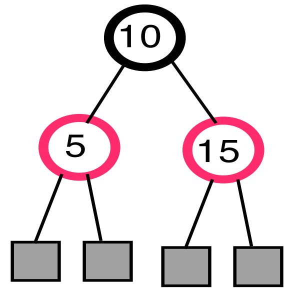
Violates property 3



Violates property 2

The following ARE valid red black trees.





1. Why do we need to learn about this?

Because $h \leq 2 \log (n+1)$: It guarantees search, insert, delete in O(log n) time.

2. But isn't that true for AVL trees too?

Yes, but in AVL trees, there are more *rotations* on an average (up to O(log n) in the worst case) during insert operations. In contrast Red-Black trees need at most one rotation per insertion and at most two rotations for deletion, so they work better with frequent insert and delete operations.

3. Search in Red-black trees

For AVL trees, $h \leq 1.44 \log n$

For RB trees, $h \leq 2 \log(n)$

So lookup is faster in AVL trees, particularly for large n, but it comes at the expense of slower insertion and deletion times.

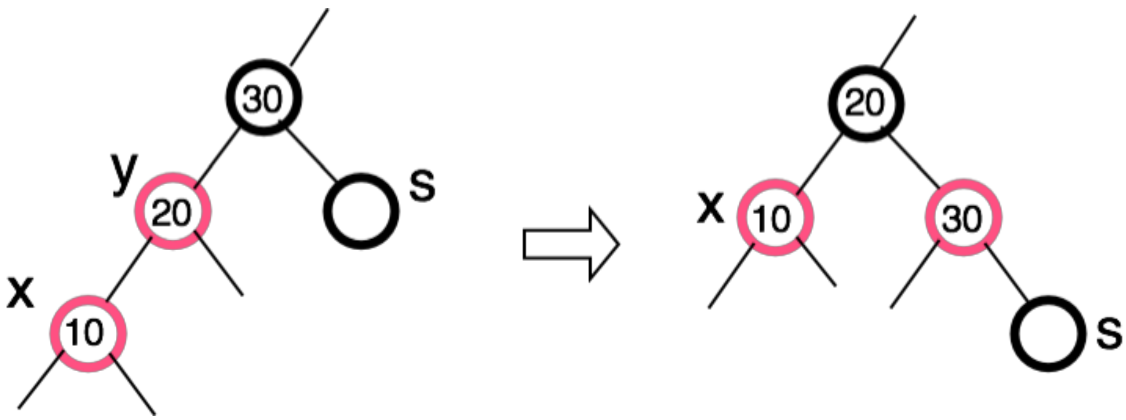
3. The TreeMap class

TreeMap class implements the Map interface using a Red-Black tree. The binary tree sorts the keys in the ascending order. The HashMap class also implements a Map interface, but is an unordered collection of key-value pairs.

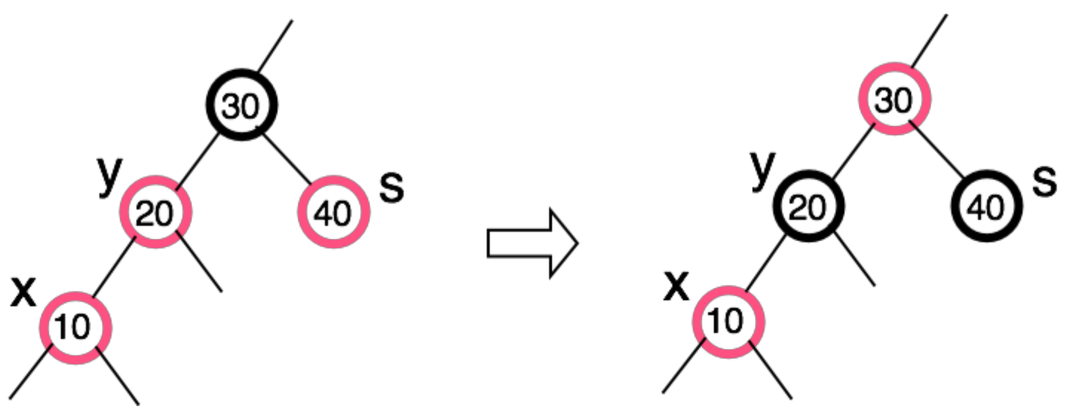
Inserting a node

Follow the normal rules of insertion in a BST. Let x = node to be inserted, and y = parent of x. Make the new node x a **red node.** Two sample cases:

Case 1. The sibling s of y is black



Case 2. The sibling s of y is red



If 30 is the root, then color it black.