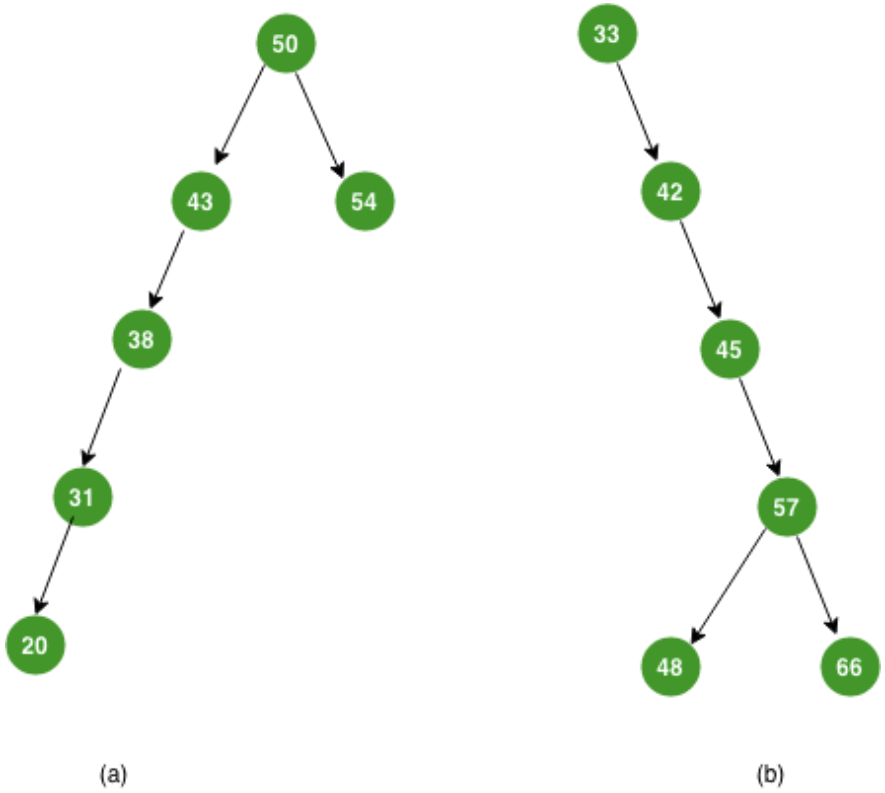
Self-balancing Binary Search Trees

Introduction

> We want to keep the height of the BST as small as possible in order to reduce the complexity of dynamic set operations

The binary search tree (https://algorithmtutor.com/Data-Structures/Tree/Binary-Search-Trees) supports dynamic set operations (Insert, Delete, Search, Maximum, Minimum, successor, Predecessor) in $O(h)$ time, where h is the height of the tree. If the height of the tree is small, these operations run fast whereas they are slow if the height of the tree is large. The problem with the ordinary binary search tree is that the height of the tree can, sometimes, be linear (n). Figure 1 shows the examples of binary search tree that are 'unbalanced' meaning that the height is large.



(

Fig 1: The examples of unbalanced binary search tree

This is equivalent to a linked list (https://algorithmtutor.com/Data-Structures/Basic/Linked-Lists/) where the dynamic set operations take $O(n)$ time. Therefore the normal binary search tree has worst-case running time of $O(n)$ for these operations. The self-balancing binary search trees keep the height as small as possible so that the height of the tree is in the order of $\log(n)$. They do this by performing transformations on the tree at key times (insertion and deletion), in order to reduce the height. Although a certain overhead is involved, it is justified in the long run by ensuring fast execution of later operations.

What is the minimum height we can obtain?

Any binary tree with height h can have at most $2^{h+1} - 1$ nodes. i.e.

$$n \leq 2^{h+1} - 1$$

And that implies,

$$h \geq \lceil \log_2(n+1) - 1 \rceil \geq \lfloor \log_2(n) \rfloor$$

That means the minimum height of the binary search tree is $\log_2(n)$ rounded down. Maintaining the height of the tree at its minimum i.e. $\lfloor \log_2(n) \rfloor$ is not always viable because of the excessive overhead it would take for the insertion operation. Therefore, most of the self-balancing BSTs keep the height within a constant factor of this bound.

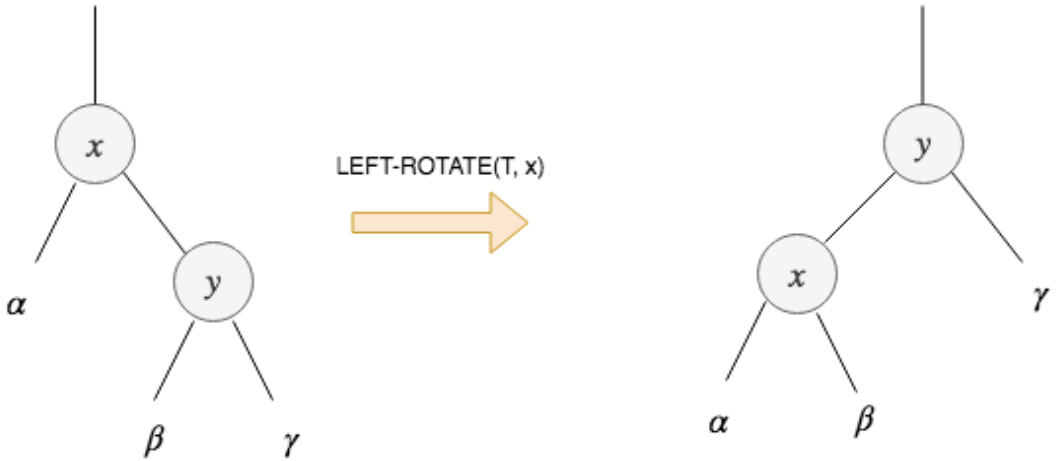
Tree rotation

> The tree rotation should not change the in-order traversal of the tree.

Tree rotation is a transformation technique which can be used to change the structure of the binary search tree *without changing the order of the elements*. It is used to decrease the height of the tree by moving down the smaller subtree and moving up the larger subtree. Many self-balancing BSTs algorithms use this technique to maintain the height of the tree after insert and delete operations. There are two types of tree rotation.

Left rotation

Figure 2 shows the left operation on a node x.



(

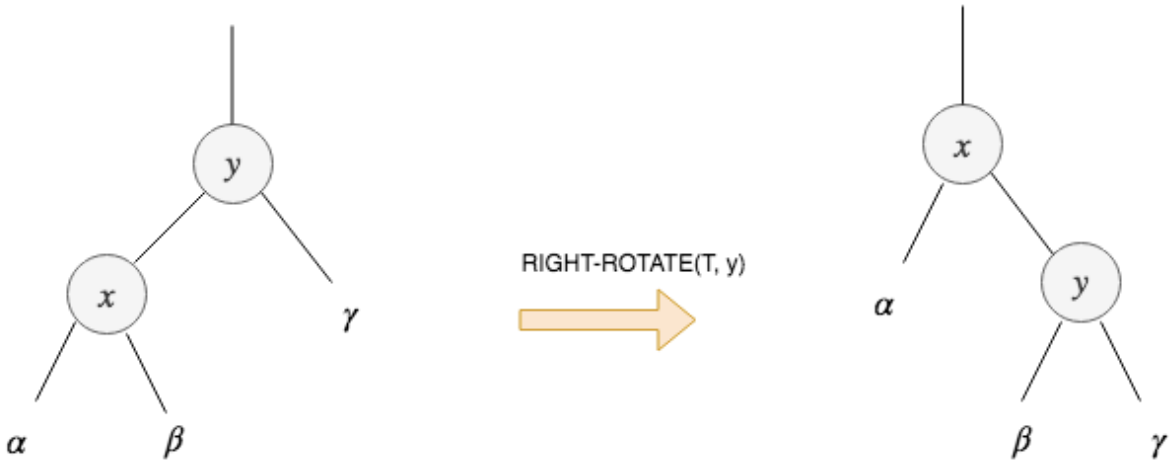
Fig 2: Illustration the left tree rotation

When we do the left rotation on a node x, we assume that its right child y is not `nil`. The left rotation makes y the new root of the subtree, with x as y's left child and y's left child as x's right child. The pseudo code for left rotation is given below.

```
LEFT-ROTATE (T, x)
    y = x.right
    x.right = y.left
    if y.left != nil
        y.left.parent = x
    y.parent = x.parent
    if x.parent == nil
        T.root = y
    elseif x == x.parent.left
        x.parent.left = y
    else x.parent.right = y
    y.left = x
    x.parent = y
```

Right rotation

Right rotation is symmetric to the left rotation. Figure 3 shows the right rotation operation.



(

Fig 3: Illustration the right tree rotation

As in the left rotation, when we do the right rotation on a node y, we assume its left child x is not `nil`. The pseudo code is given below.

```
RIGHT-ROTATE (T, x)
    y = x.left
    x.left = y.right
    if y.right != nil
        y.right.parent = x
    y.parent = x.parent
    if x.parent == nil
        T.root = y
    elseif x == x.parent.right
        x.parent.right = y
    else x.parent.left = y
    y.right = x
    x.parent = y
```

Please note that in both left and right rotations, the inorder traversal of the tree is same

Implementations

Following are the BSTs that implement the self-balancing technique.

1. AA Tree

2. AVL Tree (https://algorithmtutor.com/Data-Structures/Tree/AVL-Trees/).
3. 2-3 Tree (https://algorithmtutor.com/Data-Structures/Tree/2-3-Trees/).
4. Red-black tree (https://algorithmtutor.com/Data-Structures/Tree/Red-Black-Trees/)
5. Scapegoat tree
6. Splay tree
7. Treap

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (n.d.). Introduction to algorithms (3rd ed.). The MIT Press.

balanced tree (/tags/balanced-tree/) binary search tree (/tags/binary-search-tree/)
binary tree (/tags/binary-tree/) self-balancing tree (/tags/self-balancing-tree/)

‹ Binary Search Trees (/Data-Structures/Tree/Binary-Search-Trees/)

AVL Trees (/Data-Structures/Tree/AVL-Trees/) ›