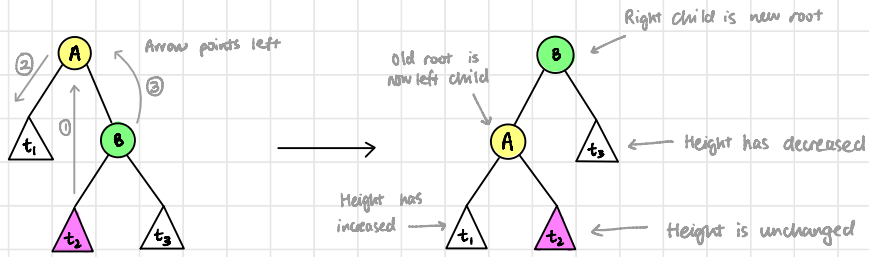
Tutorial 5

Recap: Tree rotations.

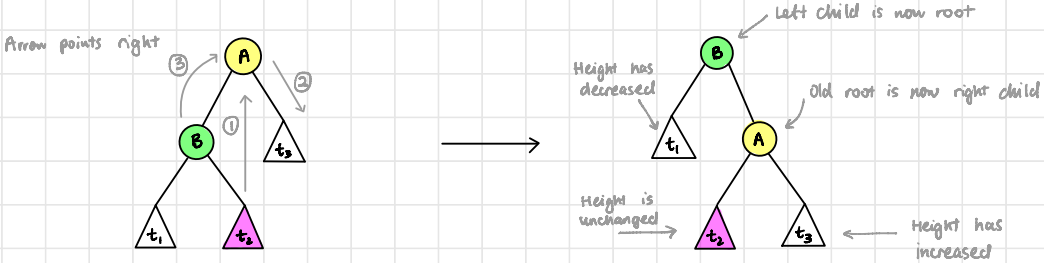
We have seen that balanced BSTs have a number of benefits over other BSTs: e.g. they guarantee $O(\log n)$ insertion and searches. Unfortunately not every BST is balanced. However, it is possible to rebalance trees, or better yet, make them automatically balance themselves! The basic operation required is rotation.

A rotation is a shape-changing operation on a tree that brings the root of a tree down one level and one of its children up one level to replace it. The use of this is to decrease the overall height of the tree by moving tall subtrees up and smaller subtrees down.

Left rotation: bring **right** subtree up

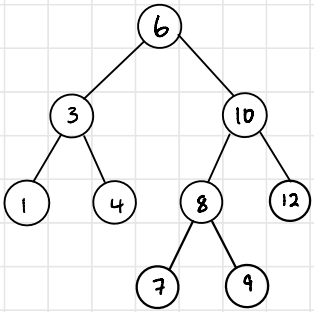


Right rotation: bring **left** subtree up

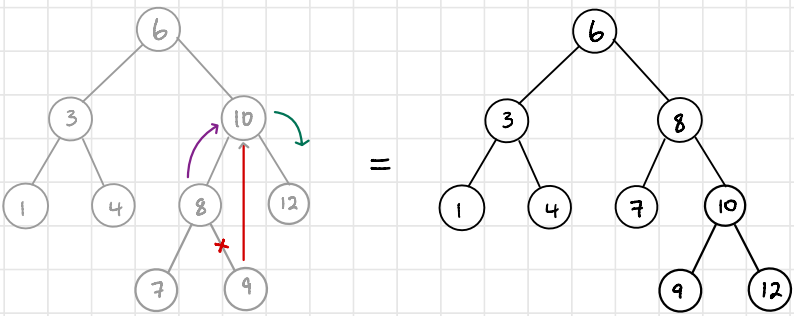


In code, this operation just involves pointer swaps, so is $O(1)$ complexity.

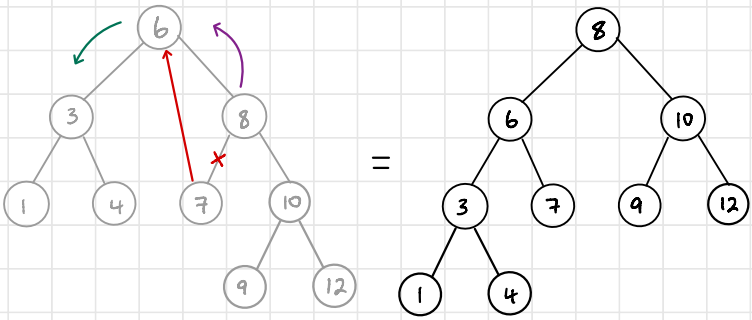
1.



Rotate right @ 10:



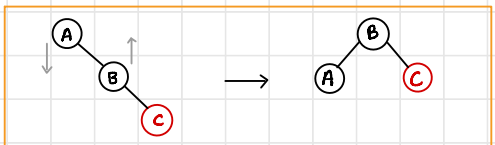
Rotate left @ 6:



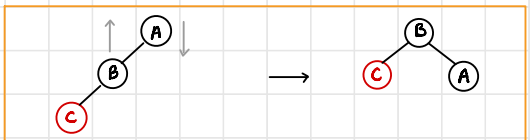
Recap: AVL trees

One of the most useful applications of rotations are AVL trees, which use them to automatically ensure a BST is balanced during each insertion. The basic principle is that imbalances occur during insertion ⇒ correct them during insertion as well. In particular, you can fix each imbalance caused by an insert into a previously balanced tree with at most 2 rotations!

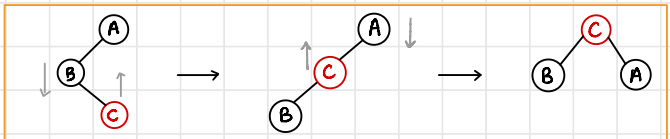
The only types of imbalances that need to be considered are



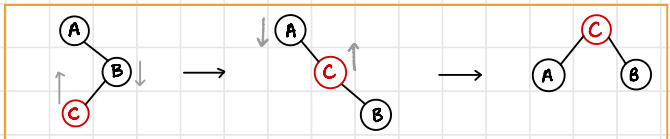
Fix by rotating B left
(1 rotation)



Fix by rotating B right
(1 rotation)



Fix by rotating B left, then rotating A right
(2 rotations)



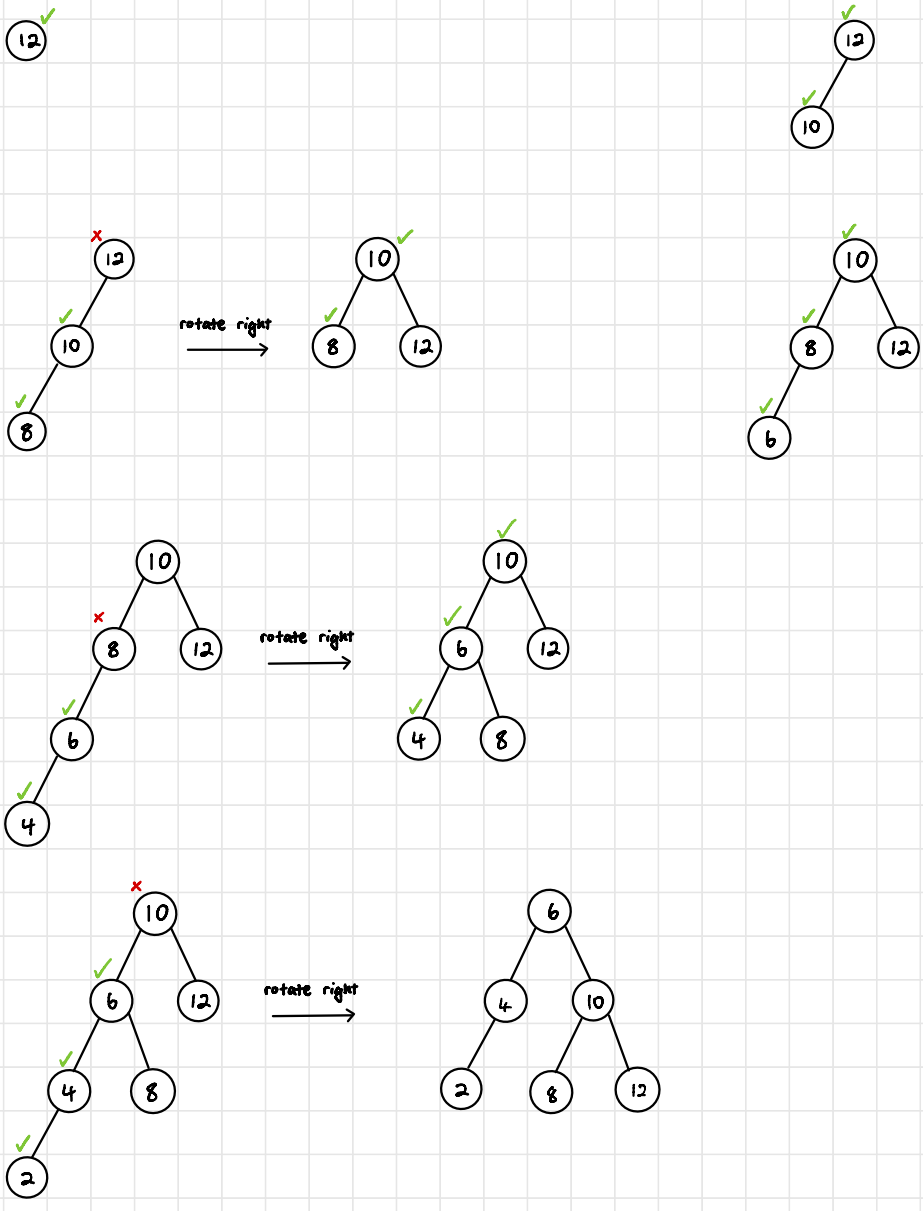
Fix by rotating B right, then rotating A left
(2 rotations)

With that, AVL insertion is just

1. Insert the node like usual
2. Check tree heights along the path you went down (i.e. is $|\text{left height} - \text{right height}| \leq 1$?)
3. If heights are imbalanced, perform one of the 4 sequences of rotation(s).

Complexity: $O(n \log n)$ if height has to be computed every time,
$O(\log n)$ if not (e.g. height stored in each node)

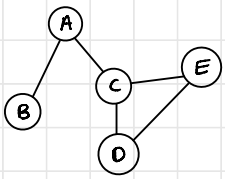
4. Insert 12 10 8 6 4 2



Recap: Graph basics

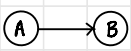
Not all data is ordered, and it also might be related to more than 2 other pieces of data. Clearly a tree is inadequate for such situations, but a graph is exactly the data structure to use.

A graph is simply a set of vertices and edges representing something like



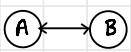
$$V = \{A, B, C, D, E\}$$
$$E = \{(A,B), (A,C), (C,D), (C,E), (D,E)\}$$

The edges can have an associated direction and/or an associated weight:



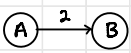
Directed

Can go A→B, but not B→A



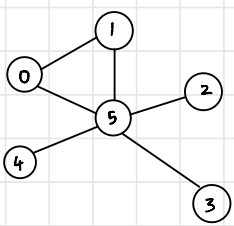
Bidirectional

Can go A→B or B→A



Weighted

5.

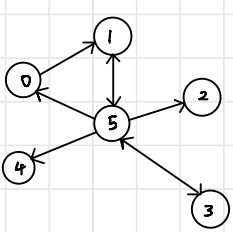


Adjacency matrix (matrix of incidences)

	0	1	2	3	4	5
0	0	1	0	0	0	1
1	1	0	0	0	0	1
2	0	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	1	1	1	1	1	0

How is the adj. matrix of a directed graph different to that of an undirected graph?

A: in an undirected graph, it is symmetric.



Adjacency list (list of edge endpoints)