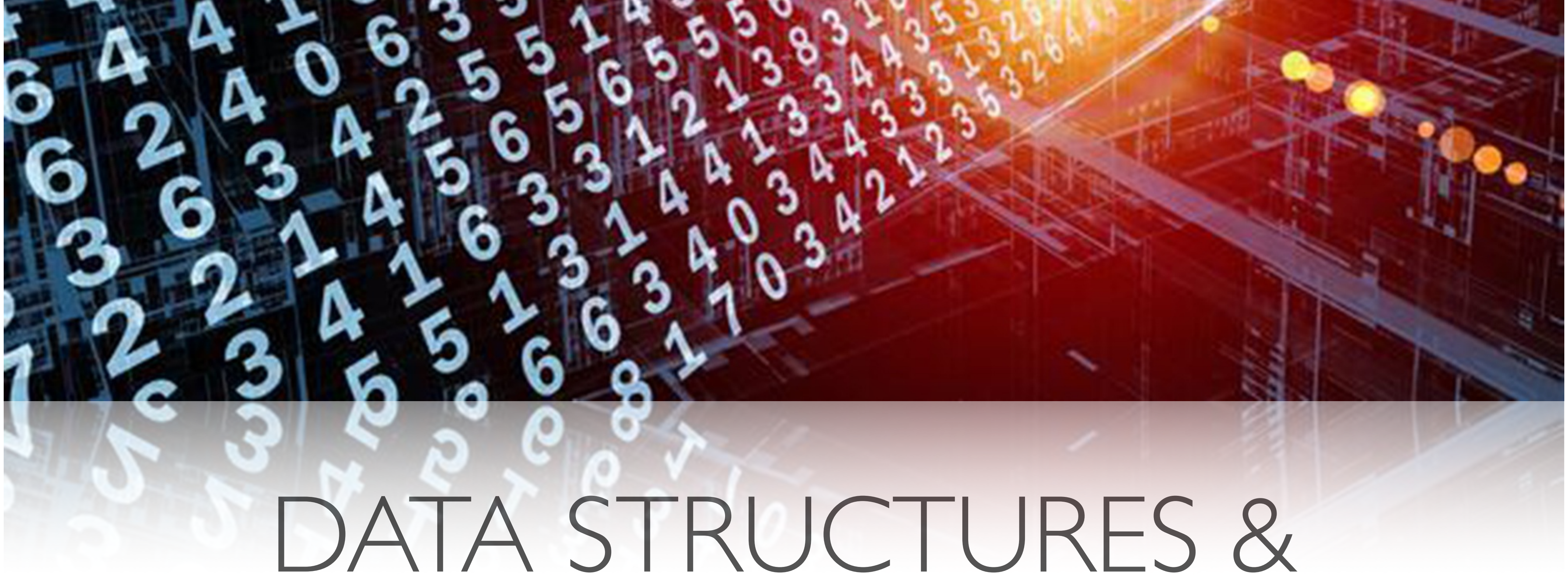


DATA STRUCTURES & ALGORITHMS
COMP H3025

Lecture 7: AVL Trees

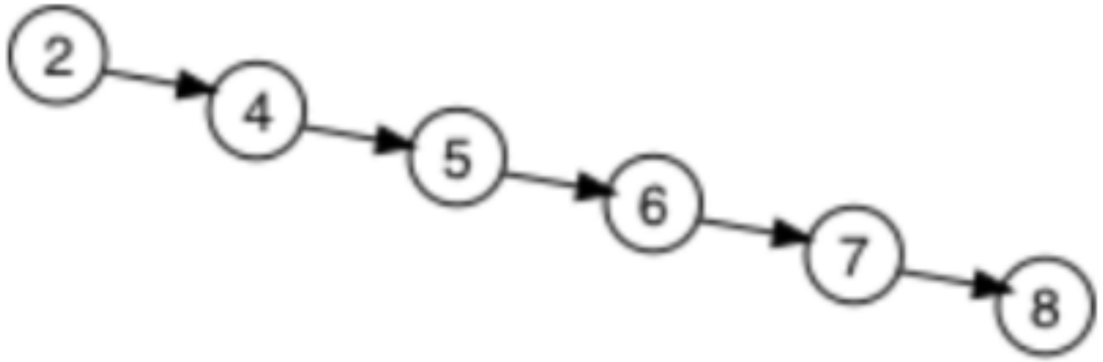
Lecturer: Stephen Sheridan

AVL TREES

- A binary search tree is designed to optimise the cost of insertion and retrieval. However, after a series of inserts and removals they can degenerate to singly linked lists.

- To see how this can happen, consider creating a binary search tree with an ordered list of values. Creating a binary search tree with the list: 2,4,5,6,7,8 gives the following tree.

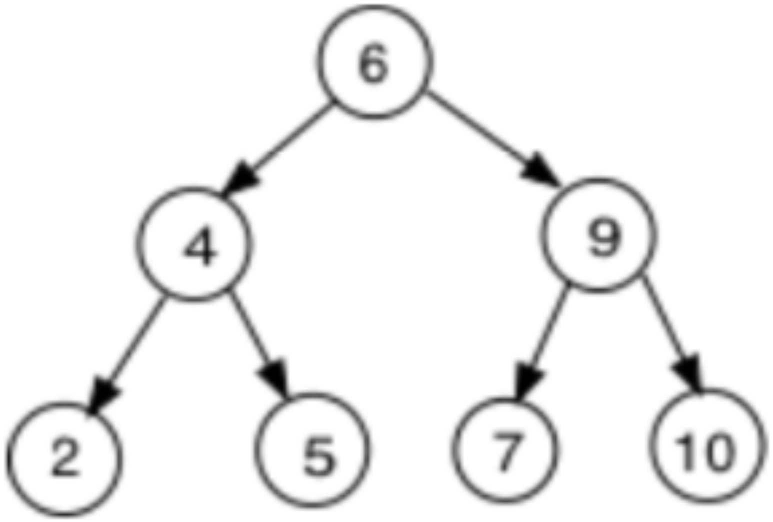
- This tree is a singly linked list where the cost of retrieval is dependant on the size of the list.



AVL TREES

- The opposite of this type of degenerate binary search tree is what is termed a complete binary tree.

- A complete binary tree is one where the nodes are equitably distributed between the two sub-trees of a given node. Such a tree would always have minimal height. The tree below is a complete binary search tree.



AVL TREES

- It can be shown that the average search cost of a binary search tree is approximately 1.39 times the cost of searching a completely balanced tree.

- This ideal model is not feasible for random sets of data. To address this problem two Russian computer scientists Adelson-Velski and Landis (1962) defined what became known as an **AVL tree**.

- This type of tree is a *self- balancing binary search tree* such that the number of left nodes and the number of right nodes of a given root node do not differ by **more than 1**.

- Each node of an AVL binary search tree has an associated **balance factor** that is **left high**, **equal** or **right high** according, respectively, as the left sub-tree has height greater than, equal to, or less than that of the right sub-tree.

AVL TREES

- We begin our study of AVL binary search trees by testing some trees to determine if they satisfy the definition of an AVL binary search tree.

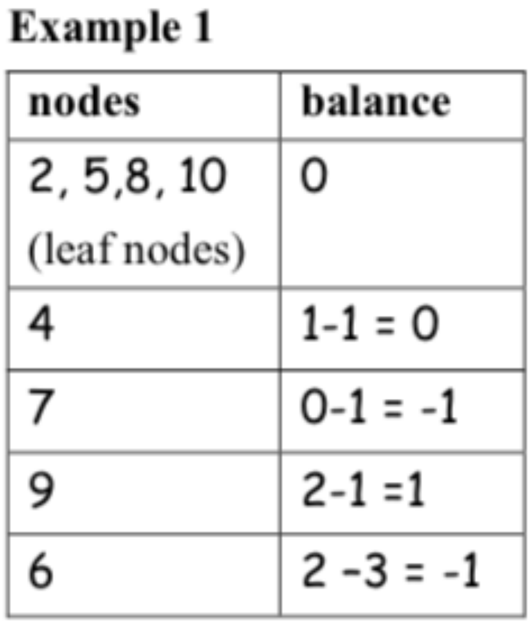
- All leaf nodes have a balance factor of 0 and the accompanying table, for each tree, lists the balance factor of each node in the tree.

- The balance factor is always equal to the difference in height between the left sub-tree and the right sub-tree of a given node.

- This value will always be 1, 0, -1, if the tree is an AVL tree. This value is got by subtracting the height of the right sub-tree from the height of the left sub- tree. The height of a node is measured starting with its deepest leaf node, tracing its path up the tree.

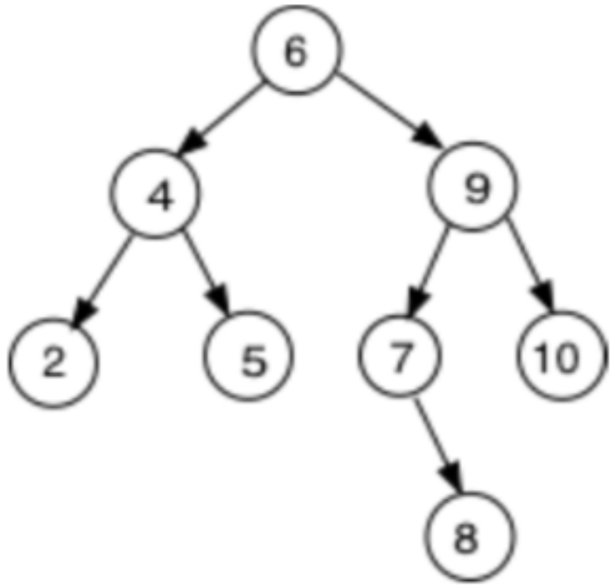
AVL TREES

- This tree satisfies the conditions for an AVL tree because each node has a balance factor of 1, 0, or -1. Node 4 has a balance factor of 0 because the heights of its left sub-tree and right sub-tree are both 1. Node 7 has a balance factor of -1 because the height of its left sub-tree is 0 and that of its right sub-tree is 1. 6 has a balance factor of -1 because the height of its left sub-tree is 2 and that of its right sub-tree is 3.



Example 1

nodes	**balance**
2, 5,8, 10 (leaf nodes)	0
4	1-1 = 0
7	0-1 = -1
9	2-1 =1
6	2 -3 = -1

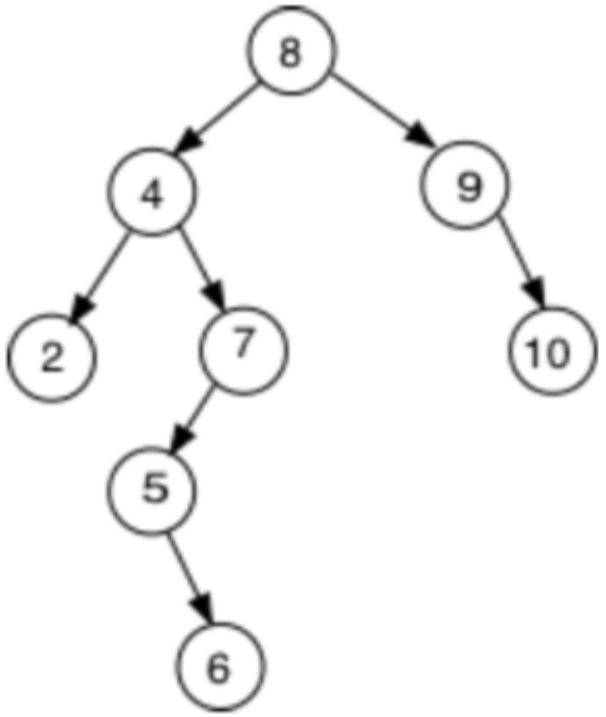


AVL TREES

- This tree fails the test because not all nodes have a balance factor of 1, 0, or -1. The first offending node is 7 because the height of the left sub-tree is 2 and that of the right sub-tree is 0.

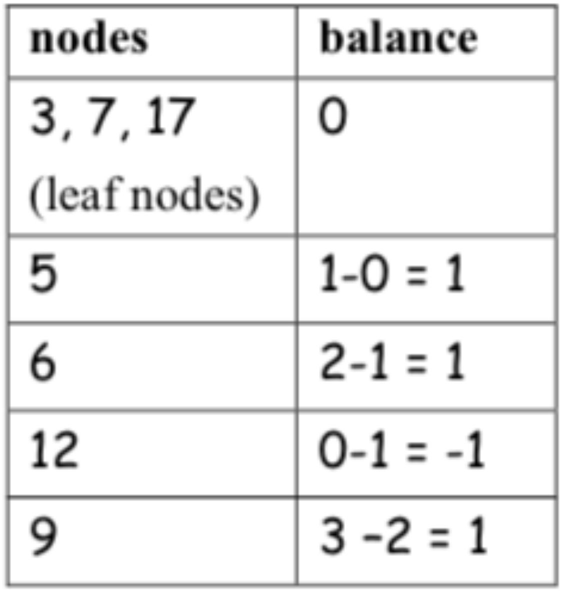
Example 2

nodes	**balance**
2, 6, 10 (leaf nodes)	0
5	0-1 = -1
7	2-0 = 2
4	1-3 =-2
9	0 -1 = -1
8	4-2 = 2

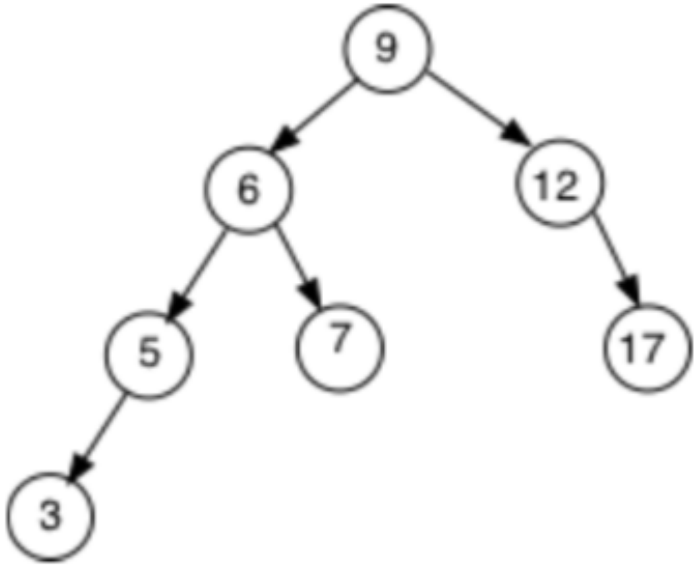


AVL TREES - INSERTING AN ELEMENT

- Given an AVL tree we want to insert a new element in the tree while keeping its AVL property invariant. Inserting a new element proceeds as for insertion in an ordinary binary search tree. This can cause problems because the tree now may become unbalanced and, hence, may need to re-balance itself. Given the following AVL binary search tree to start with.

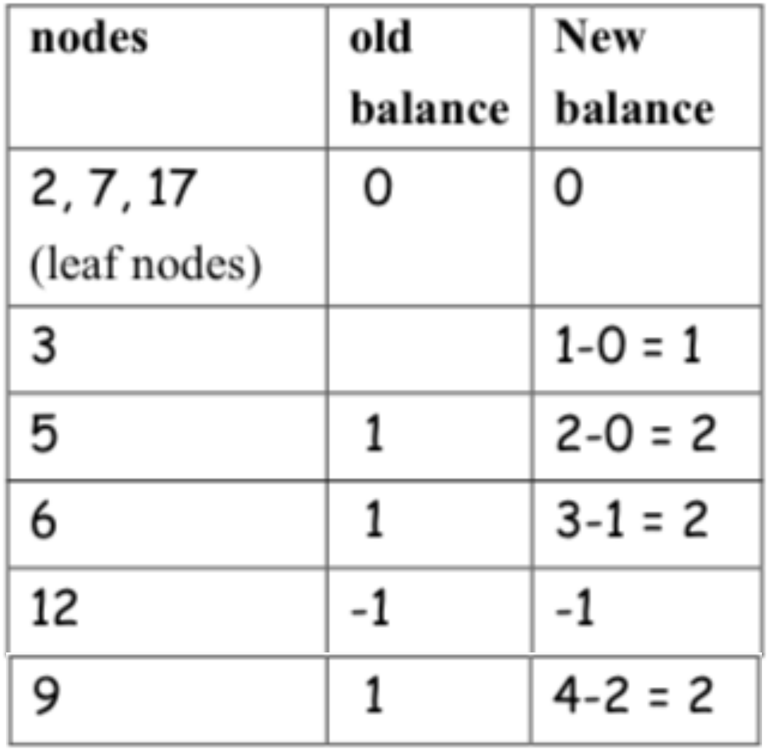


nodes	**balance**
3, 7, 17 (leaf nodes)	0
5	1-0 = 1
6	2-1 = 1
12	0-1 = -1
9	3 -2 = 1

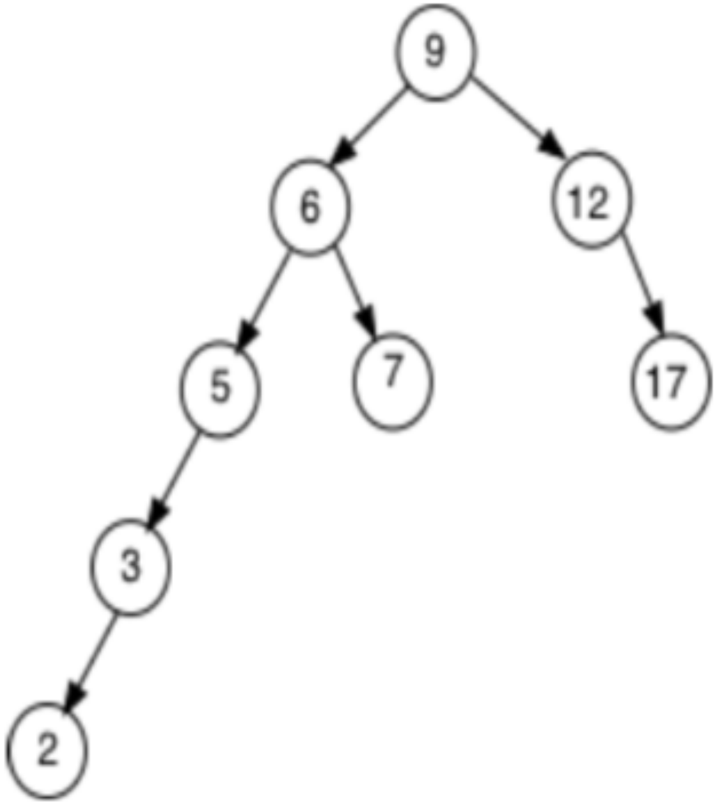


AVL TREES - INSERTING AN ELEMENT

- Inserting 2 in the tree gives the unbalanced tree.

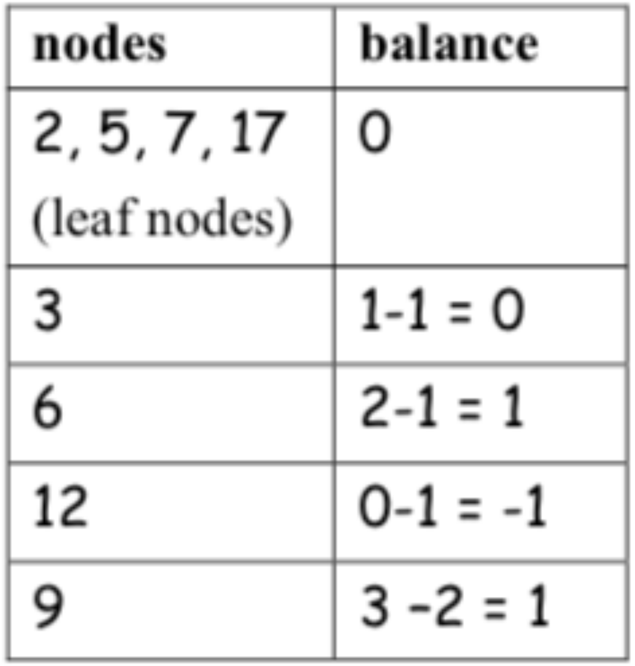


nodes	old balance	New balance
2, 7, 17 (leaf nodes)	0	0
3		1-0 = 1
5	1	2-0 = 2
6	1	3-1 = 2
12	-1	-1
9	1	4-2 = 2

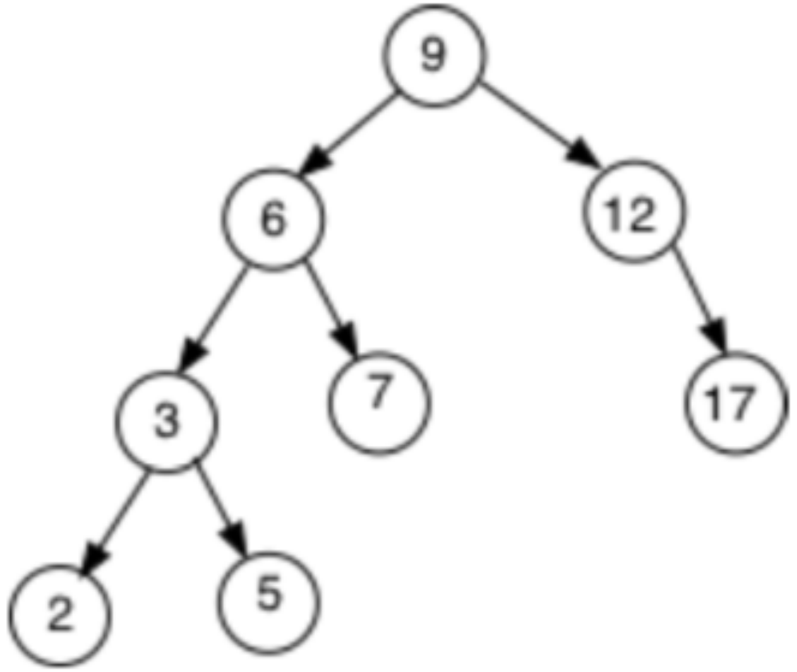


AVL TREES - INSERTING AN ELEMENT

- The first node ascending from the root node 2 that violates the AVL property is 5. Hence, the tree must be re-balanced. To do this we re-balance the tree by placing the grandparent as the right child of its child node 3, and then connecting 3 to its own grandparent 6. Now each node in the tree has a balance factor satisfying the AVL property and the new tree remains a binary search tree.



nodes	balance
2, 5, 7, 17 (leaf nodes)	0
3	1-1 = 0
6	2-1 = 1
12	0-1 = -1
9	3 -2 = 1



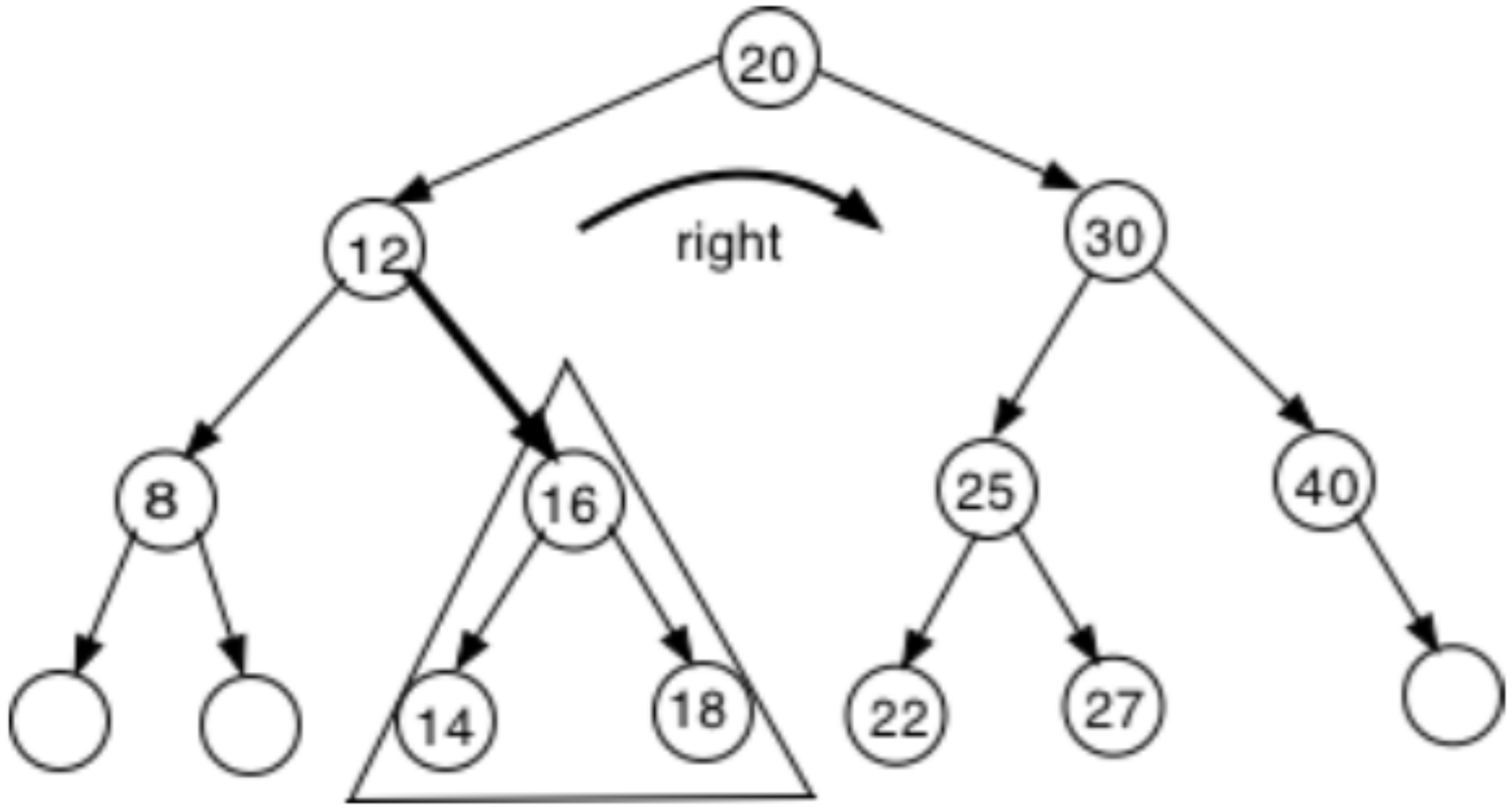
AVL TREES - INSERTING AN ELEMENT

- The task of re-balancing binary search trees keeping their ordering invariant is done by using different rotations of the nodes of the tree. There are two basic rotations used called **left rotation** and **right rotation**. Both of these are single rotations.

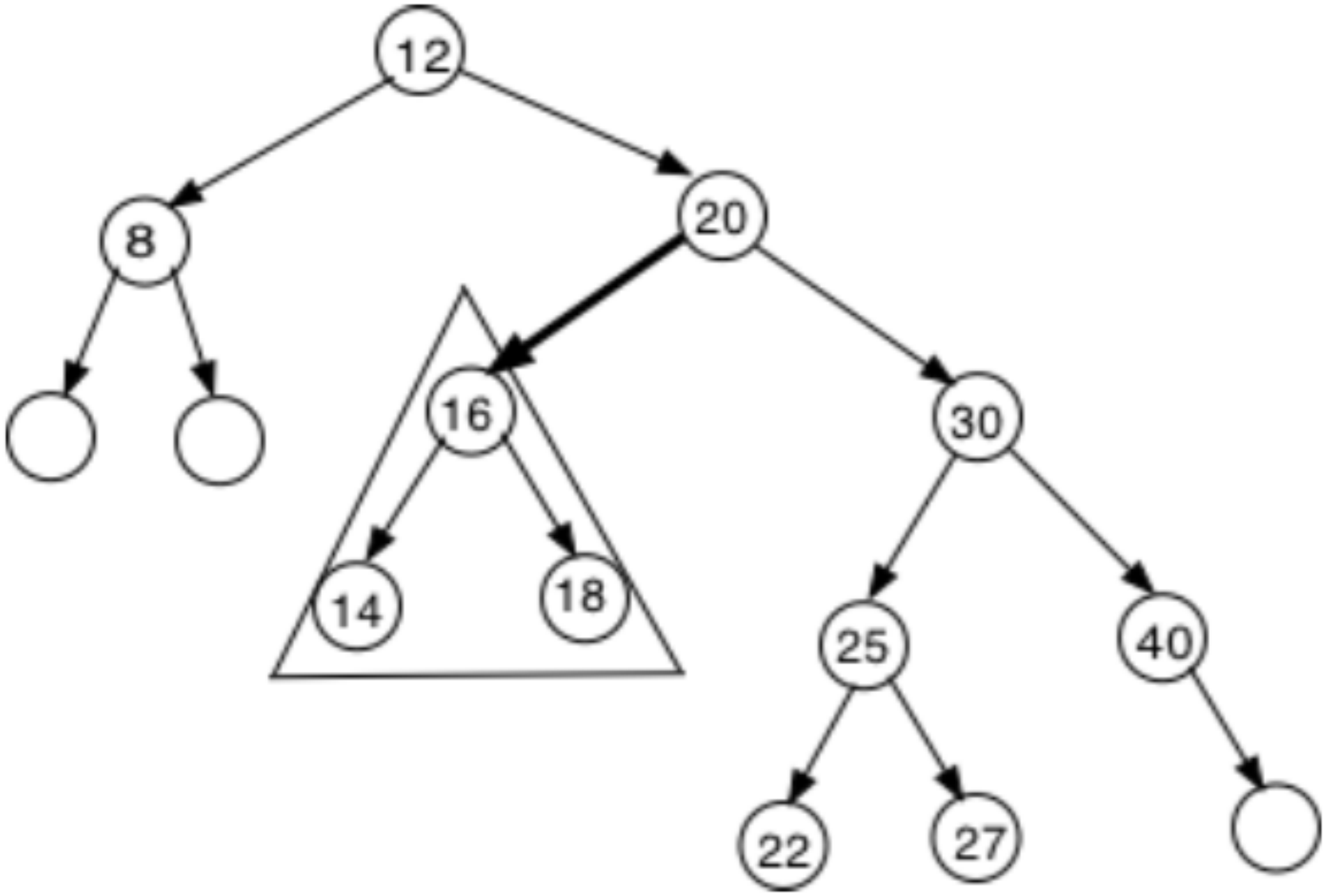
- To describe the mechanics of both rotations two examples are given. The tree on the next slide is a binary search tree. The plan is to perform a right rotation about the root 20 keeping invariant the ordering for a binary search tree. This rotation will make 12 the new root.

- The triangle enclosing nodes 14, 16 and 18 has to shift to the right side of the new root. This means that the block of nodes will be placed as the left child of 20 in the new tree. This shift is illustrated in the second diagram. The rotation only requires a change of pointers – the left pointer of 20 points to node 16 and the right pointer of node 12 points to node 20. This is called a right rotation.

AVL TREES - INSERTING AN ELEMENT



AVL TREES - INSERTING AN ELEMENT

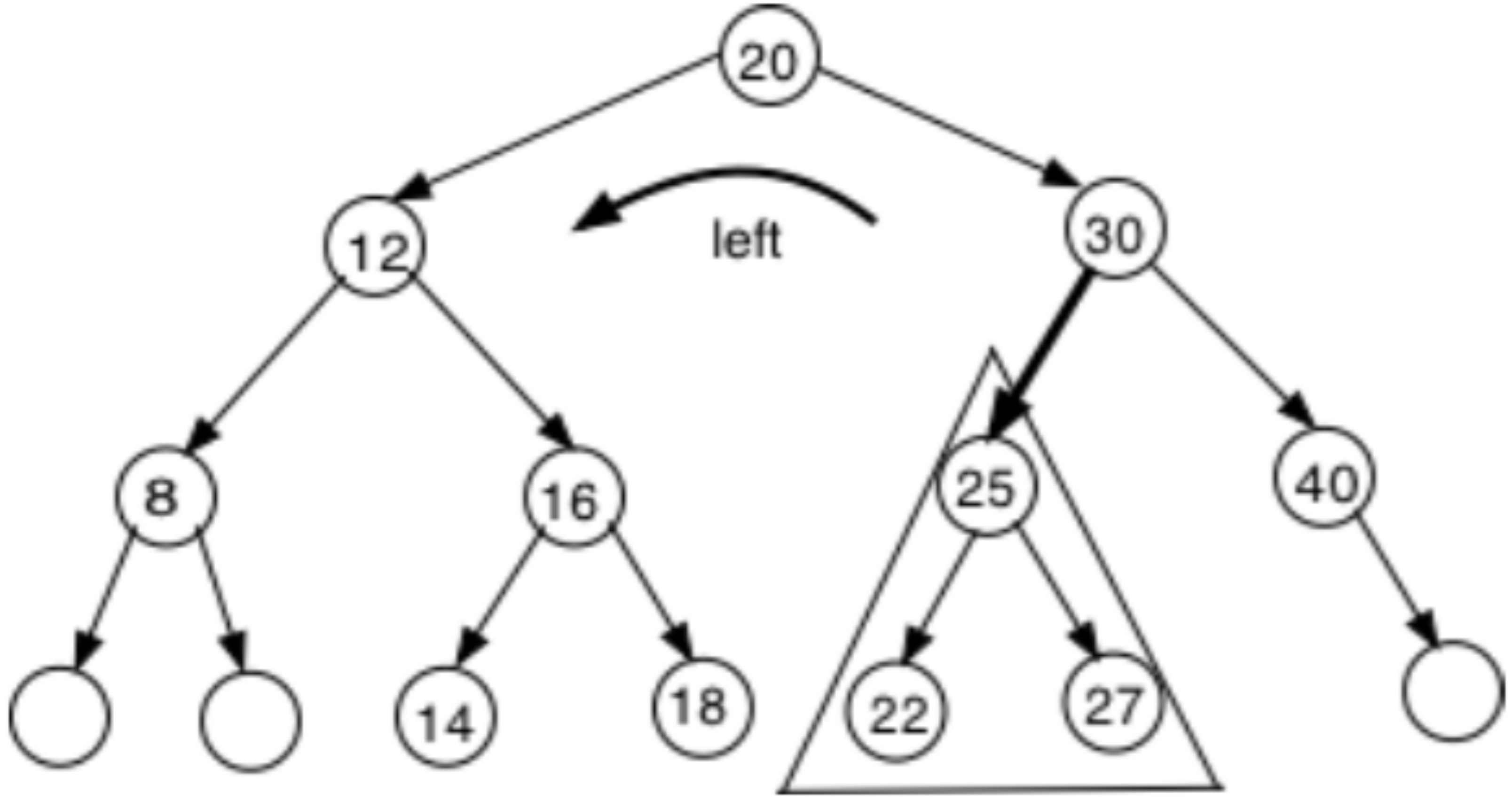


AVL TREES - INSERTING AN ELEMENT

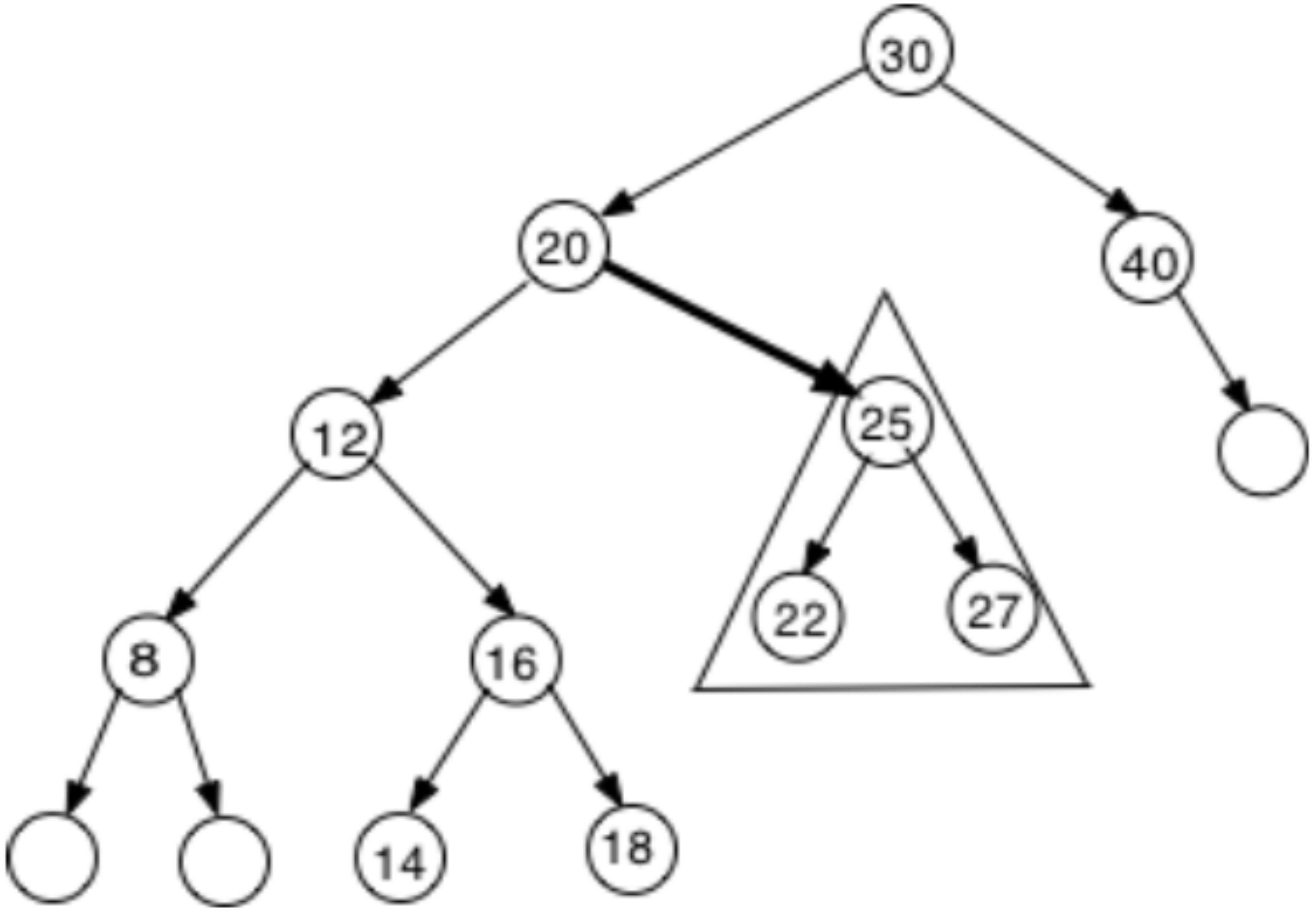
- The second rotation is called a left rotation. It is the mirror image of the right rotation described above.

- The pair of diagrams, given next, show the state of a tree before a left rotation and after it has taken place. To keep the ordering the right pointer of node 20 points to node 25 and the left pointer of node 30 points to node 20.

AVL TREES - INSERTING AN ELEMENT



AVL TREES - INSERTING AN ELEMENT



AVL TREES - INSERTING AN ELEMENT

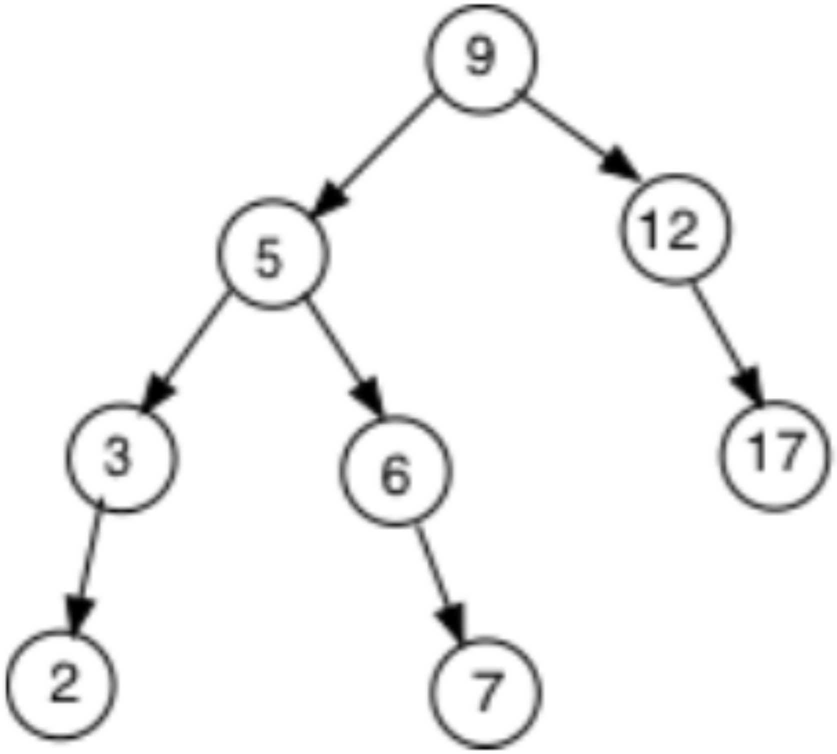
- A left or right rotation can be used to re-balance a binary search tree keeping its AVL property invariant.

- The first example we looked at illustrated a right rotation necessary to balance the tree when the number 2 was inserted in the tree. We now give an example of a **left rotation**.

AVL TREES - INSERTING AN ELEMENT

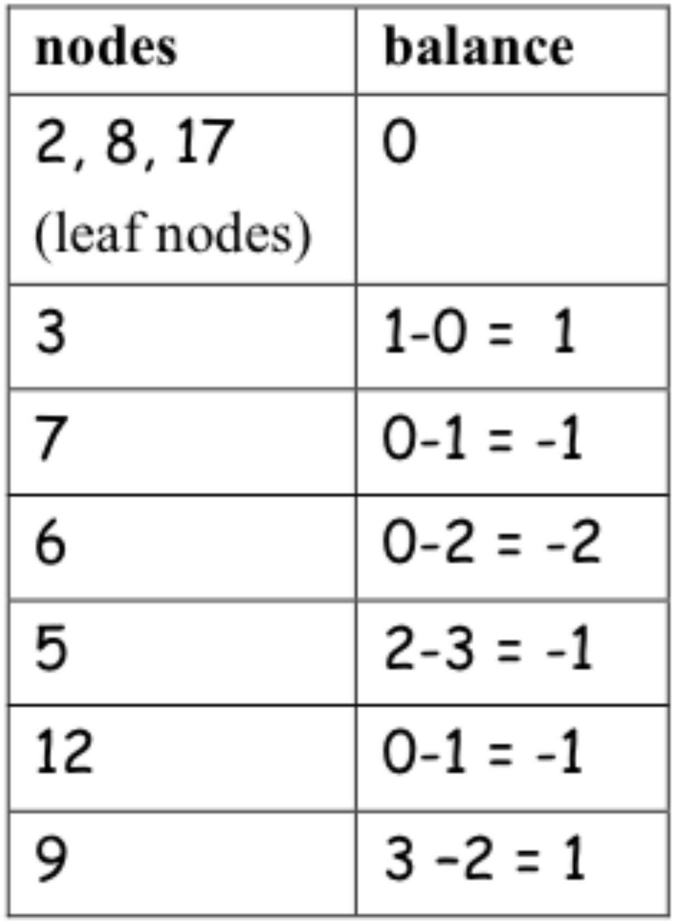
- Given below is a balanced AVL tree.

nodes	balance
2, 7, 17 (leaf nodes)	0
3	1-0 = 1
6	0-1 = -1
5	2-2 = 0
12	0-1 = -1
9	3 -2 = 1

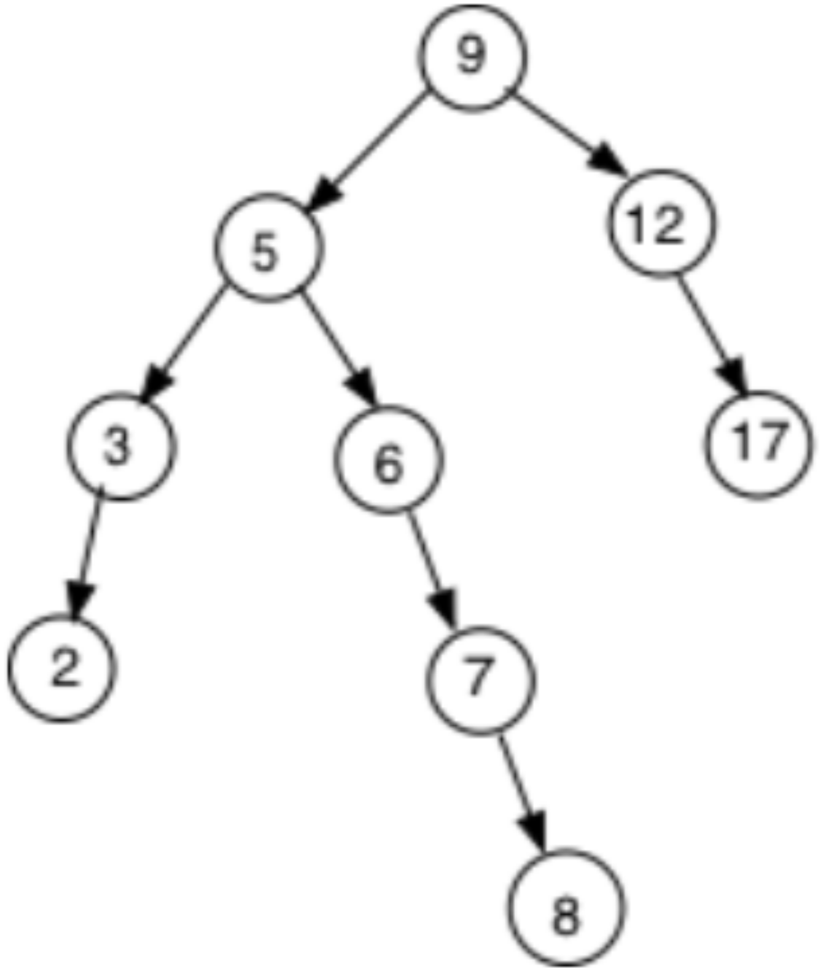


AVL TREES - INSERTING AN ELEMENT

- Inserting the number 8 to the right of 7 gives an unbalanced tree.



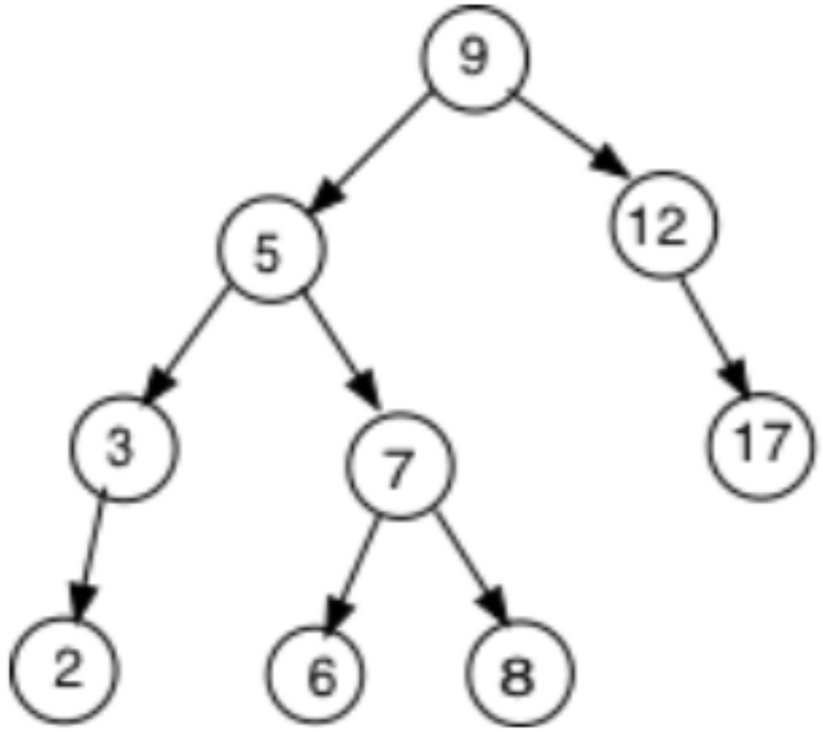
nodes	balance
2, 8, 17 (leaf nodes)	0
3	1-0 = 1
7	0-1 = -1
6	0-2 = -2
5	2-3 = -1
12	0-1 = -1
9	3 -2 = 1



AVL TREES - INSERTING AN ELEMENT

- Node 6 is the grandparent of node 8 and to re-balance the tree we place 6 as the left child of its child node 7 and 7 becomes the root node of the sub-tree. This re-balances the tree as the table below shows.

nodes	balance
2, 6, 8, 17 (leaf nodes)	0
3	1-0 = 1
7	1-1 = 0
5	2-2 = 0
12	0-1 = -1
9	3 -2 = 1

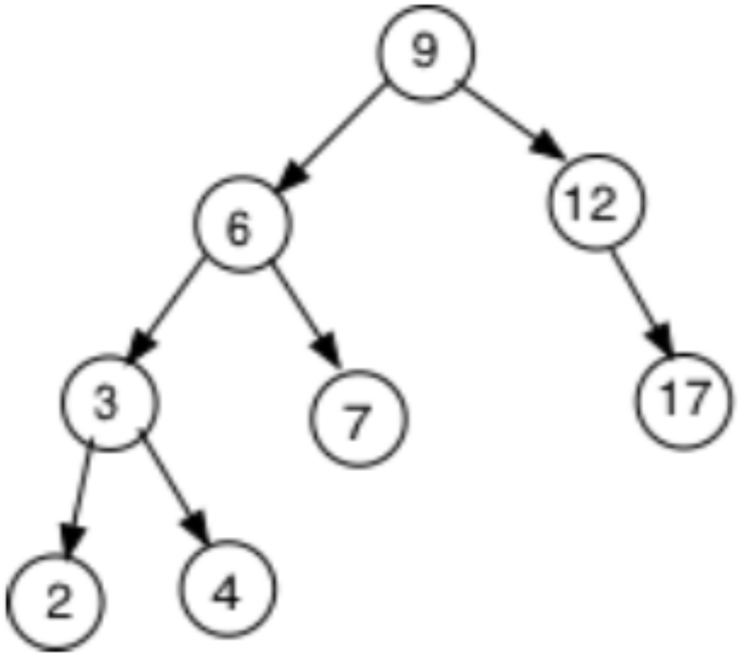


AVL TREES - INSERTING AN ELEMENT

- Sometimes when a new value is inserted the task of re-balancing the tree requires more than a single left or right rotation. The example, below, illustrates what is called a left-right rotation.

- Given the balanced AVL tree, below.

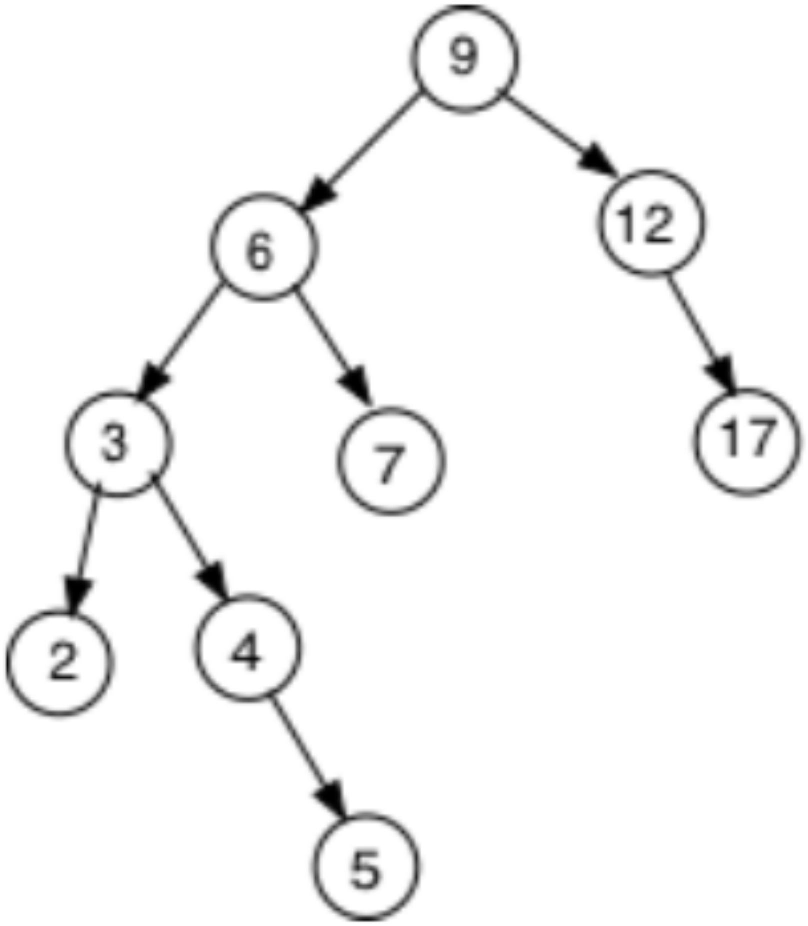
nodes	**balance**
2, 4, 7, 17 (leaf nodes)	0
3	1-1 = 0
6	2-1 = 1
12	0-1 = -1
9	3 -2 = 1



AVL TREES - INSERTING AN ELEMENT

- Inserting 5 to the right of 4 unbalances the tree as the table shows.

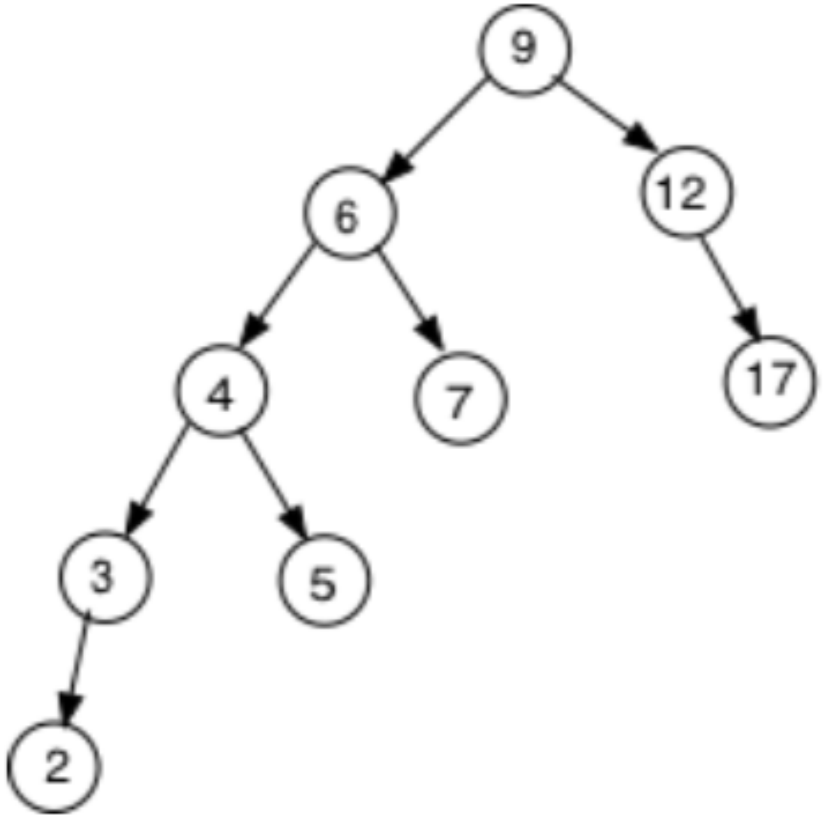
nodes	balance
2, 5, 7, 17 (leaf nodes)	0
4	0-1 = -1
3	1-2 = -1
6	3-1 = 2
12	0-1 = -1
9	4-2 = 2



AVL TREES - INSERTING AN ELEMENT

- Re-balancing the tree requires two steps: a left rotation followed by a right rotation. The left rotation gives an intermediate tree that is still unbalanced.

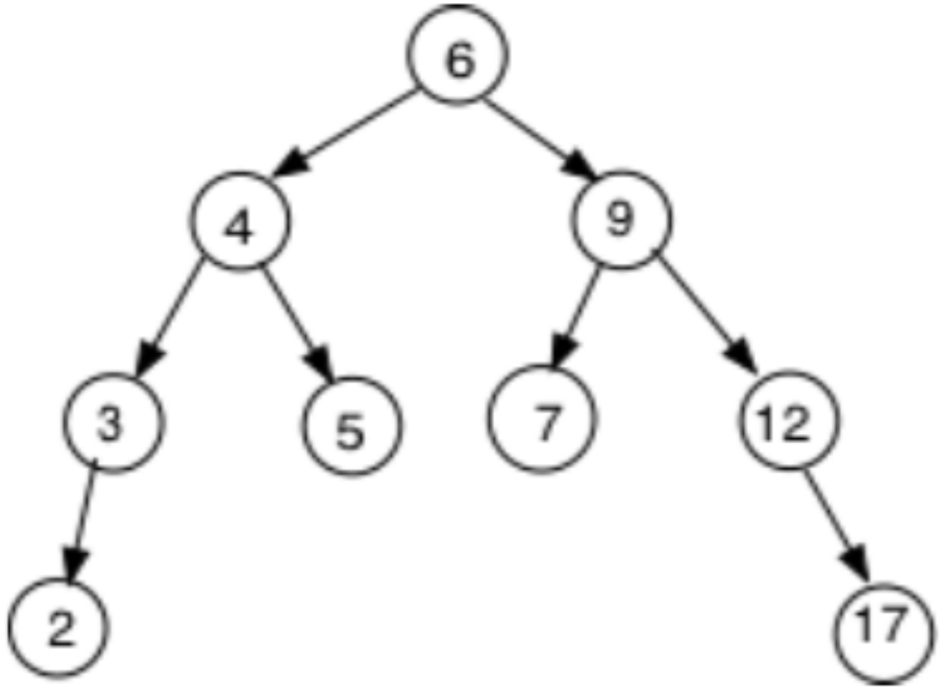
nodes	balance
2, 5, 7, 17 (leaf nodes)	0
3	1-0 = 1
4	2-1 = 1
6	3-1 = 2
12	0-1 = -1
9	4-2 = 2



AVL TREES - INSERTING AN ELEMENT

- Taking this intermediate tree and rotating right around node 6 gives a balanced tree. Node 7 becomes a child of its original grandparent.

nodes	balance
2, 5, 7, 17 (leaf nodes)	0
3	1-0 = 1
4	2-1 = 1
6	3-3 = 0
9	1-2 = -1
12	0-1 = -1



AVL TREES - INSERTING AN ELEMENT

- The fourth, and final case, involves what is called a right-left rotation. This is a mirror image of a left-right rotation.

- Deleting a node is as for binary search trees, except that depth re-calculations and sub-tree re- balancing along the path from the deletion point to the root must be carried out. This entails more work than insertion because the sub-tree re-balancing must proceed right up to the root.

TODO - WEEK 6

- Create an AVL binary search tree on paper using the list: 4,1,6,9,12,5,15,18,19,2,3,7,8,10.