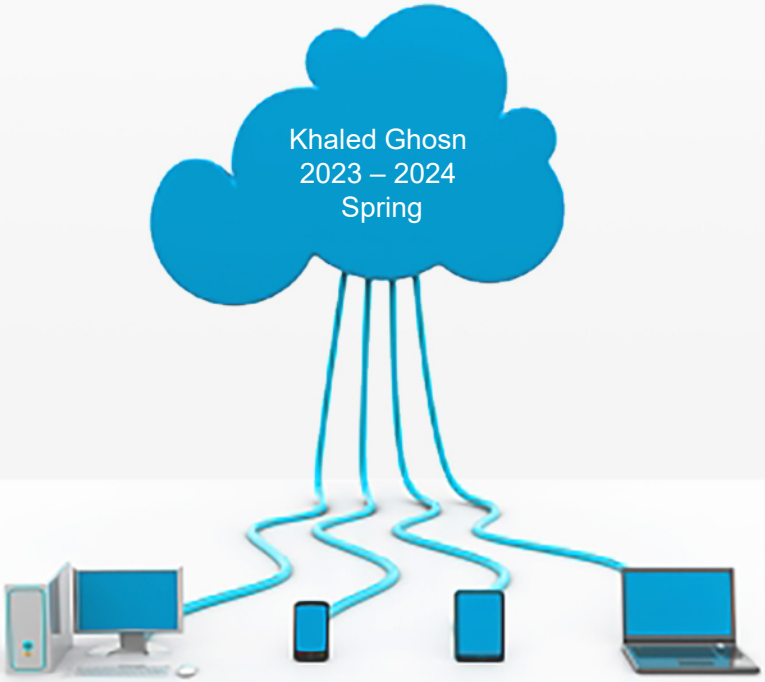
ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON
AUL

# Balanced Search Trees

**AVL, Splay, (2,4), Red-Black**



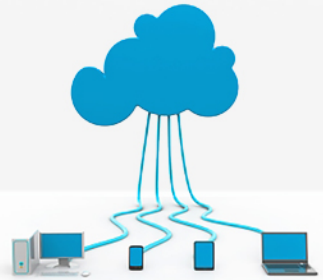
Khaled Ghosn
2023 – 2024
Spring

# Balanced Search Trees

## Overview

A full tree (proper binary tree / 2-Tree) is a tree in which every node other than the leaves has only two children

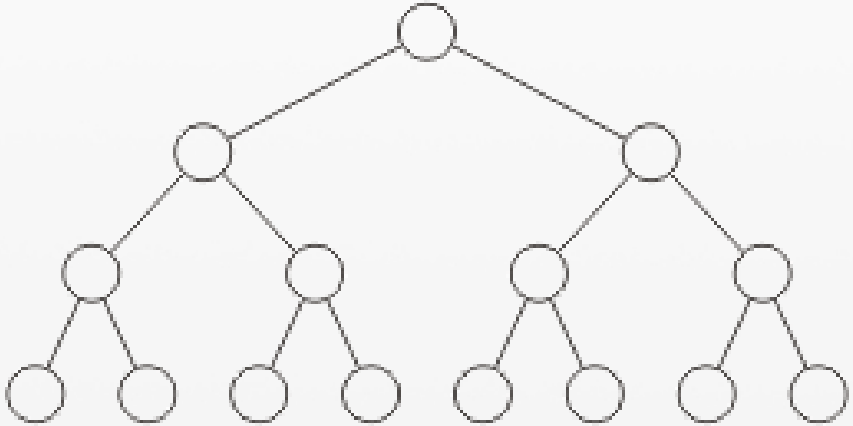
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible



# Balanced Search Trees

## Introduction

For a perfect tree, all nodes have the same number of descendants on each side

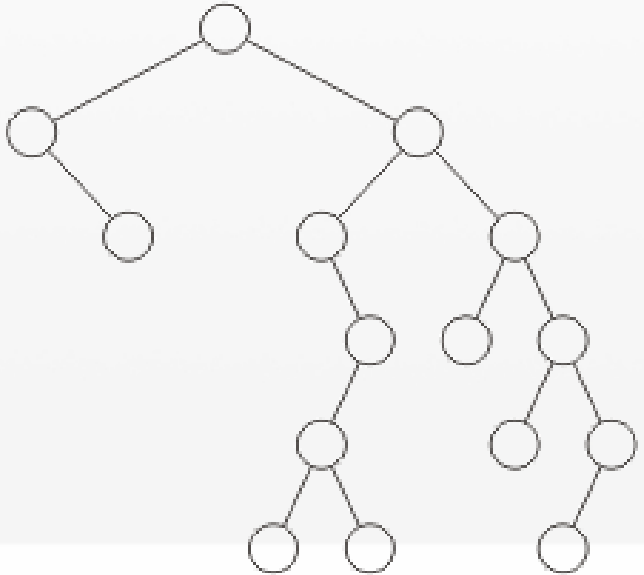


Perfect binary trees are balanced while linked lists are not

# Balanced Search Trees

## Introduction

This binary tree would also probably not be considered to be "balanced" at the root node

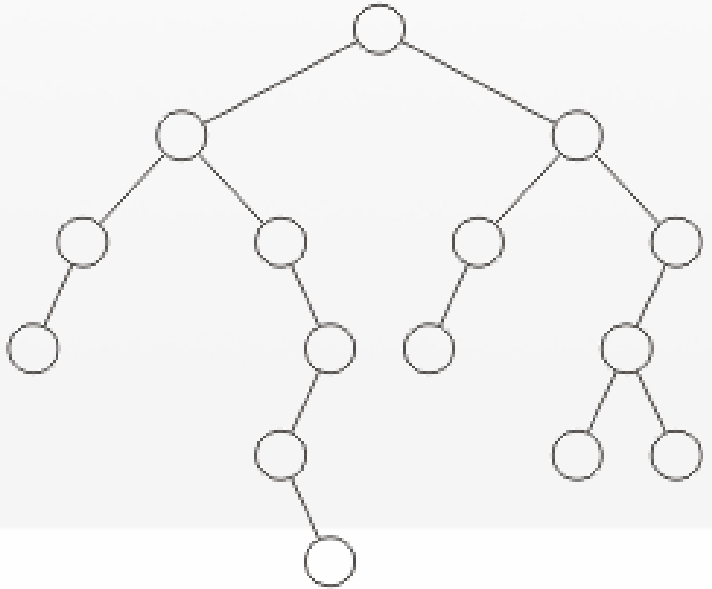


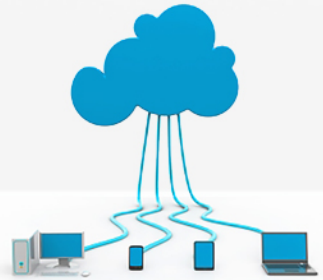
# Balanced Search Trees

## Introduction

How about this example?

- The root seems balanced, but what about the left sub-tree?





# Balanced Search Trees

## Balanced Tree

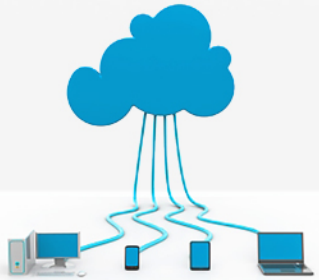
- ✓ The *height* of a tree is the length of the longest path from the tree's root to one of its leaves
  - As usual, the height of an empty sub-tree is –1
  - A tree with a single node has height 0

- ✓ Balanced may be defined by:
  - *Height balancing:* comparing the heights of the two sub trees

  Or:
  - *Null-path-length balancing:* comparing the null-path-length of each of the two sub-trees (the length to the closest null sub-tree / empty node)
  - *Weight balancing:* comparing the number of null sub-trees in each of the two sub trees

- ✓ Balance factor of a node=
  Height of its left sub-tree - Height of its right sub-tree

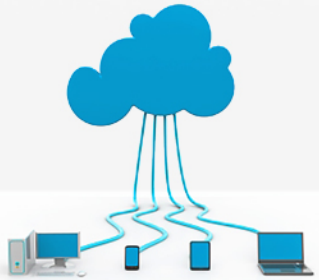
- ✓ Balanced Trees provide stronger performance guarantees



# Balanced Search Trees

## Balanced Tree

- ✓ The primary operation to rebalance a binary search tree is known as a Rotation.
- ✓ During a rotation, we "rotate" a child to be above its parent
- ✓ A rotation allows the shape of a tree to be modified while maintaining the search-tree property



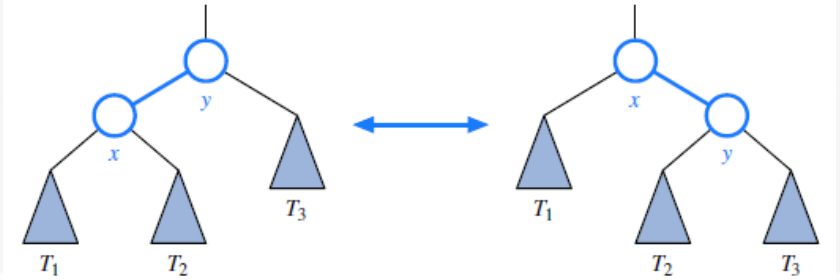
# Balanced Search Trees

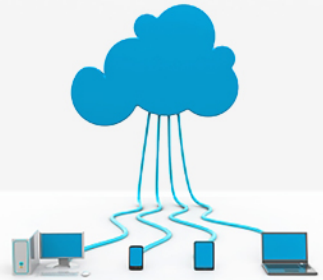
## Single Rotations

- ✓ Position X is left child of Position Y:
  - therefore the key of x is less than the key of y

  => then y becomes the right child of x after the rotation (and vice versa) + relink the subtree of entries

- ✓ Because a single rotation modifies a constant number of parent-child relationships, it can be implemented in O(1) time



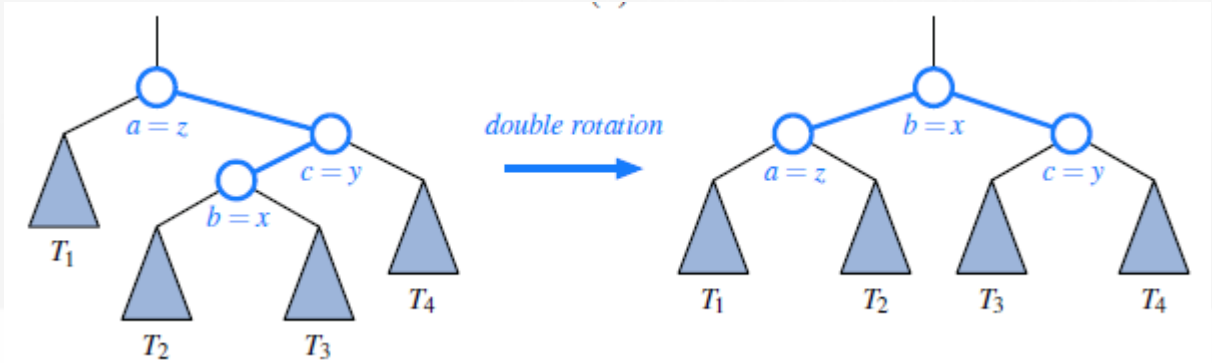



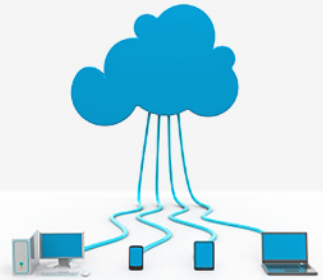
# Balanced Search Trees

## Double Rotations

- ✓ The double rotation arises when:
  - position x has the middle of the three relevant keys
  - and is first rotated above its parent
  - and then above what was originally its grandparent

- ✓ the trinode restructuring is completed with O(1) running time

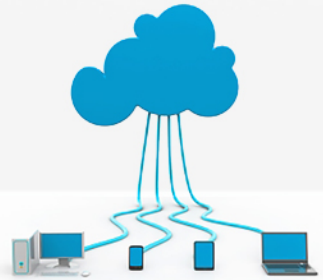


# Balanced Search Trees

- ✓ AVL trees
- ✓ Splay Trees
- ✓ (2, 4) Trees
- ✓ Red-Black Trees

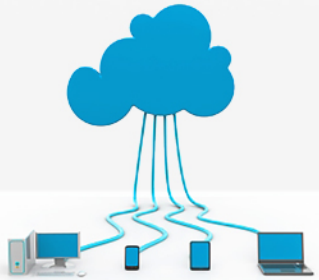
(B+ trees: balanced trees, but not binary trees)



# AVL Tree

## Introduction

- ✓ Discovered by 2 Russian researchers: *Adelson-Velskii* and *Landis*
  - published in 1968
- ✓ The AVL tree was the first balanced binary search tree
  - illustrates most of the ideas that are used in other schemes
- ✓ It is a binary search tree that has an additional balance condition
  - balance is defined by comparing the height of the two sub-trees
  - any balance condition must be easy to maintain and ensures that the depth of the tree is $O(\log N)$



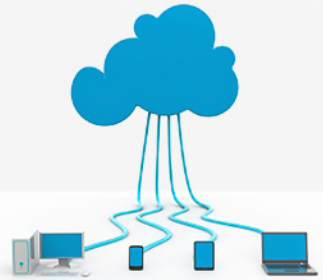
# AVL

## Definition

- ✓ in other words; Is a binary search tree with the additional balance property that:
  for any node in the tree:
  *the height of the left and right sub-trees can differ by at most 1*

- ✓ in other words; is a self-balancing Binary Search Tree where the difference between heights of left and right sub-trees cannot be more than one for all nodes

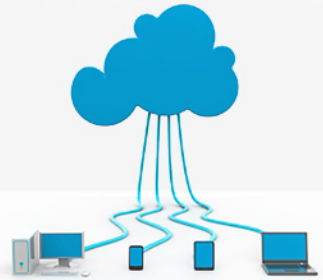
- ✓ in other words; A binary search tree is said to be AVL balanced if:
  - the difference in the heights between the left and right sub-trees is at most 1 (-1, 0, or 1), and
  - both sub-trees are themselves AVL trees



# AVL

## Definition

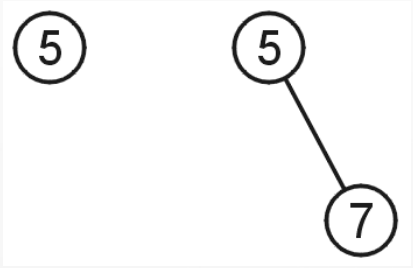
- ✓ A Node is balanced if:
  the height of its left sub-tree is -plus or minus- one the height of its right sub-tree
- ✓ Any complete binary search tree is an AVL tree

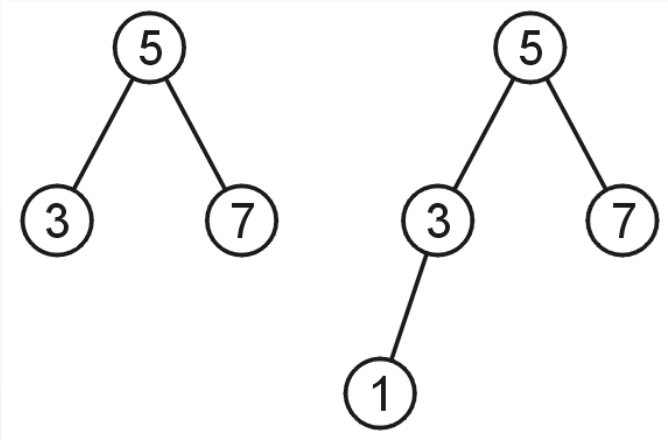


# AVL Tree

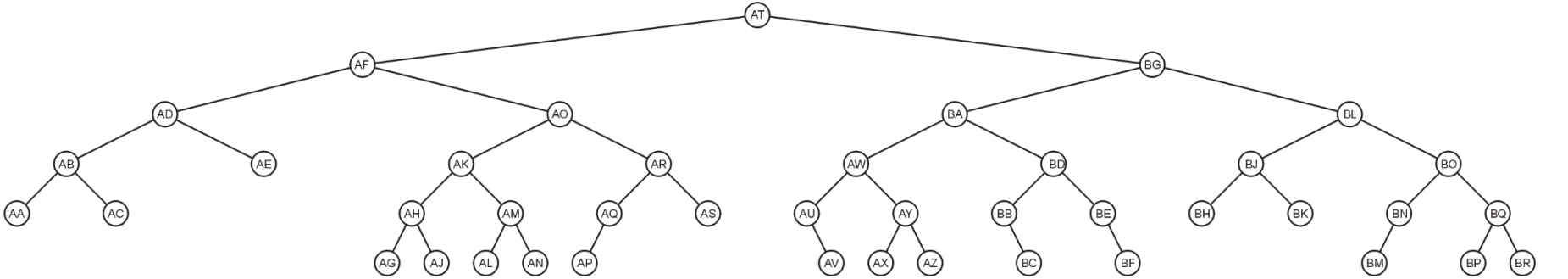
## Examples

AVL trees with 1, 2, 3, and 4 nodes:




Here is a larger AVL tree (42 nodes):

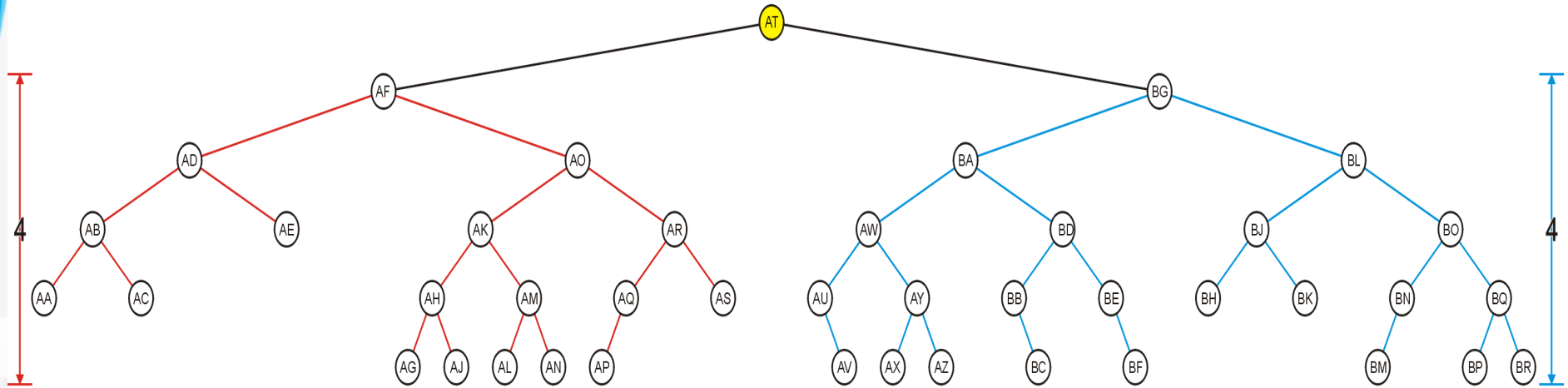


# AVL Tree

## Examples

The root node is AVL-balanced:

- Both sub-trees are of height 4

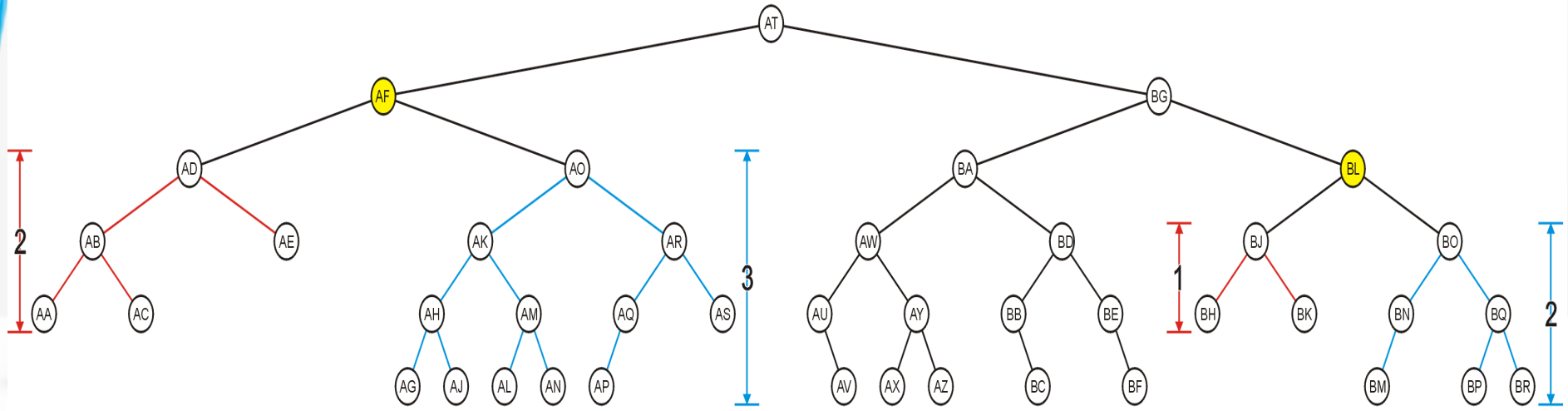


# AVL Tree

## Examples

All other nodes (e.g., AF and BL) are AVL balanced

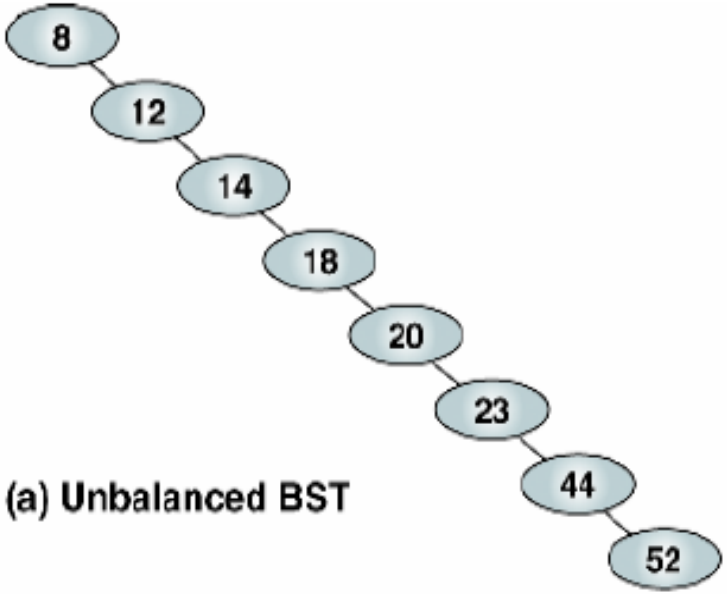
- The sub-trees differ in height by at most one



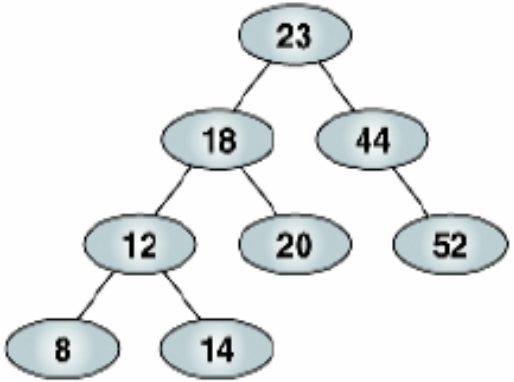
# AVL Tree

## Examples

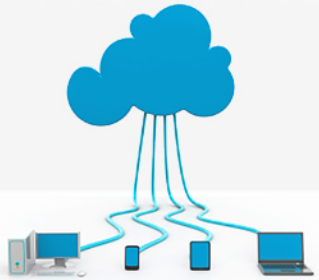
Both trees are BST, but the first tree (a) is not an AVL tree




(a) Unbalanced BST



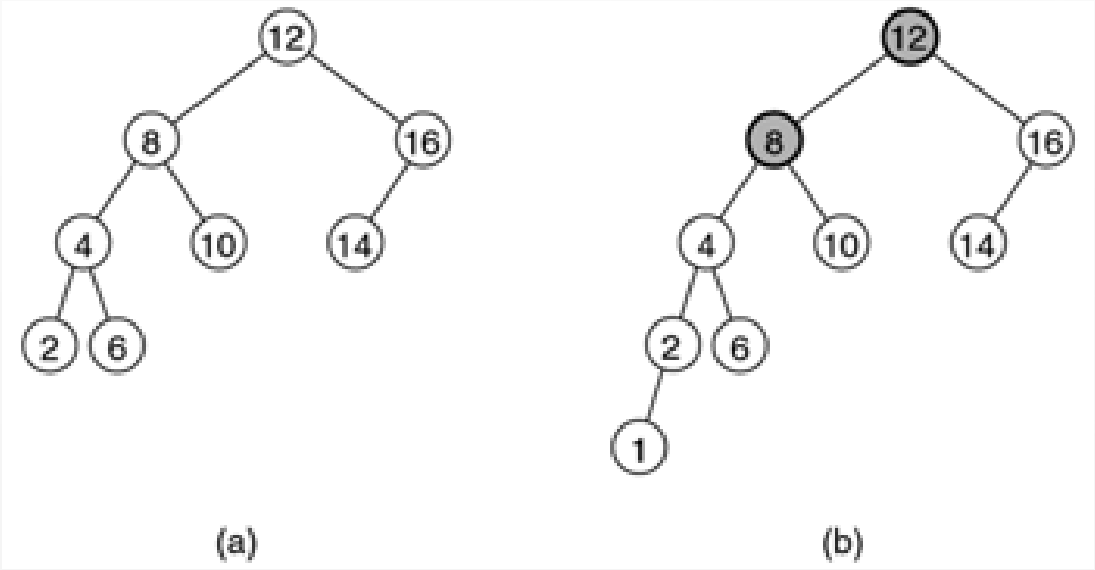

(b) AVL tree

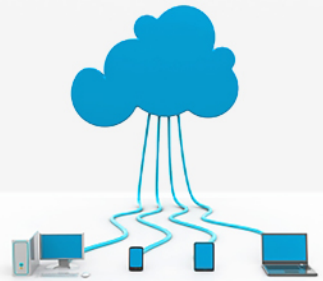


# AVL Tree

## Examples

An AVL Tree (a) becomes unbalanced after inserting new node (b)



# SPLAY

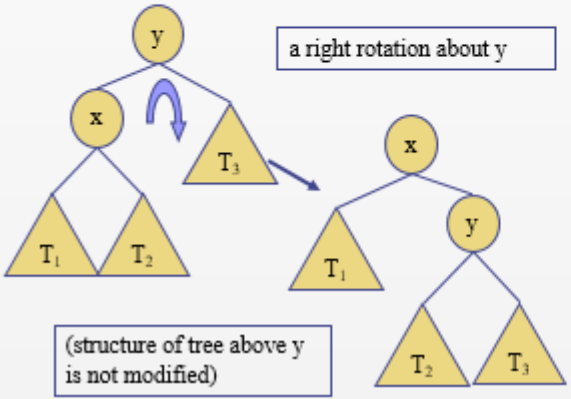
## Definition

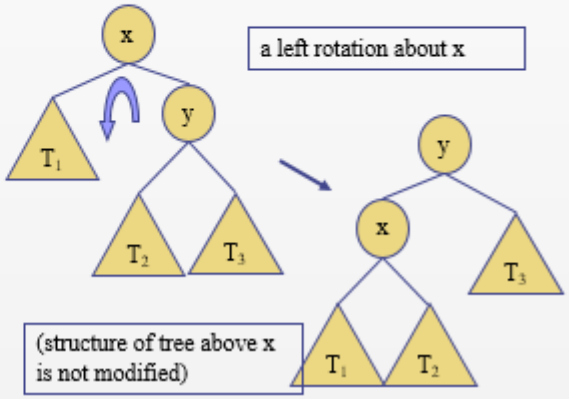
✓ Splay Trees do Rotations after Every Operation (Even Search)

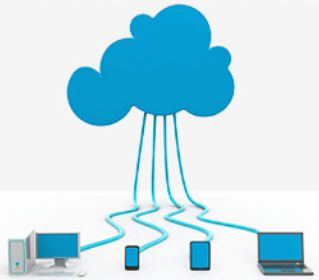
- new operation: ***splay***
  - splaying moves a node to the root using rotations

- **right rotation**
  - makes the left child $x$ of a node $y$ into $y$'s parent; $y$ becomes the right child of $x$

- **left rotation**
  - makes the right child $y$ of a node $x$ into $x$'s parent; $x$ becomes the left child of $y$



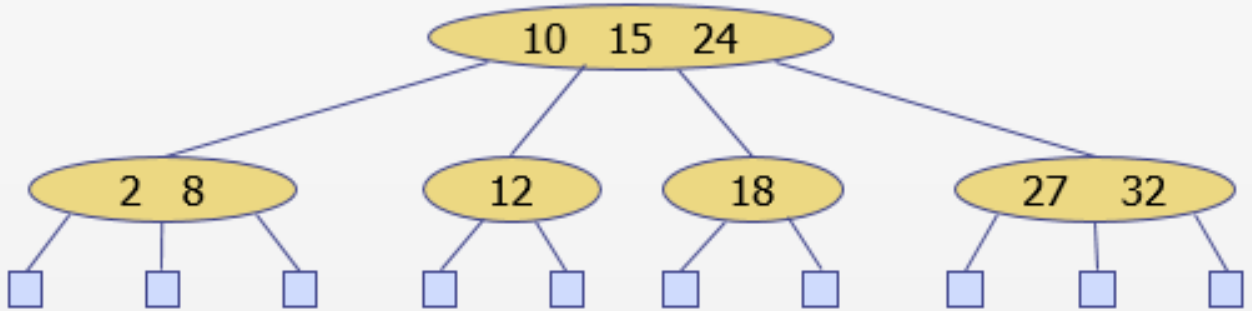



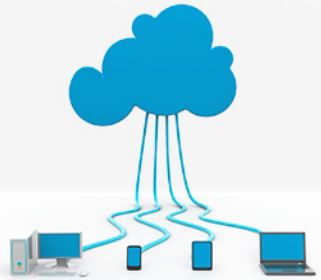



# (2, 4)

## Definition

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

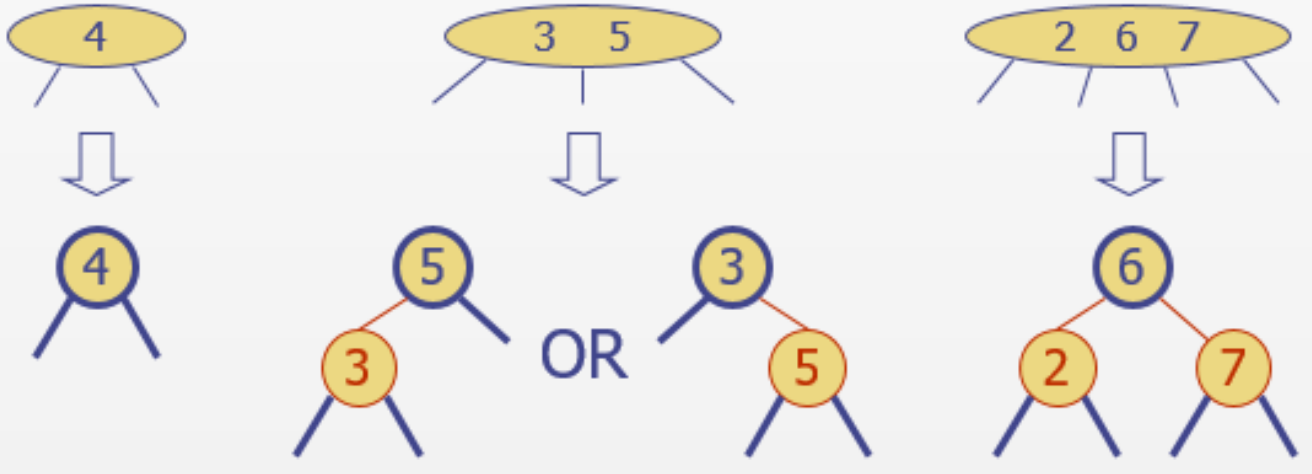


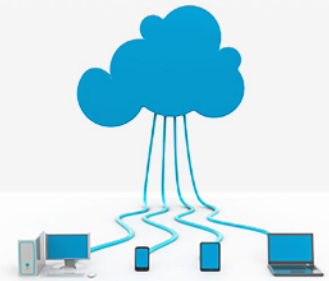



# Red Black

## Definition

- A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are colored red or **black**
- In comparison with its associated (2,4) tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type

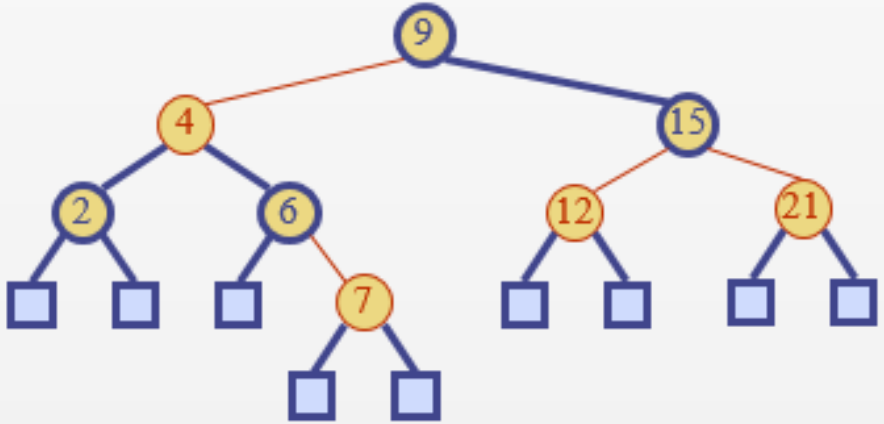


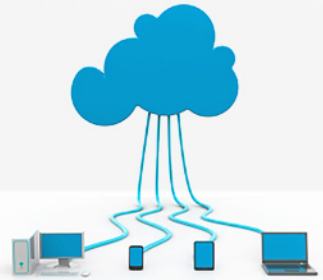



# Red Black

## Definition

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth

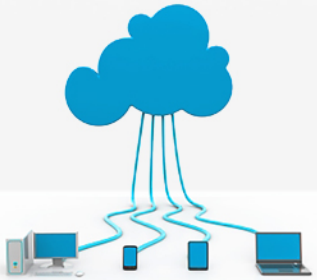


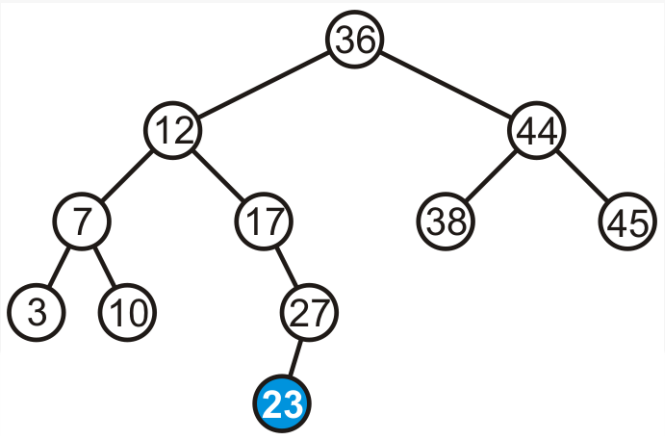



# AVL Tree

## Maintaining Balance

Suppose we insert 23 into our initial tree

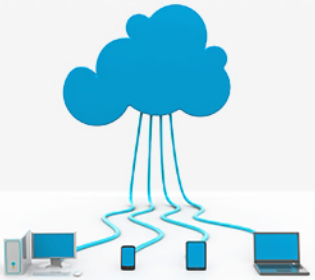


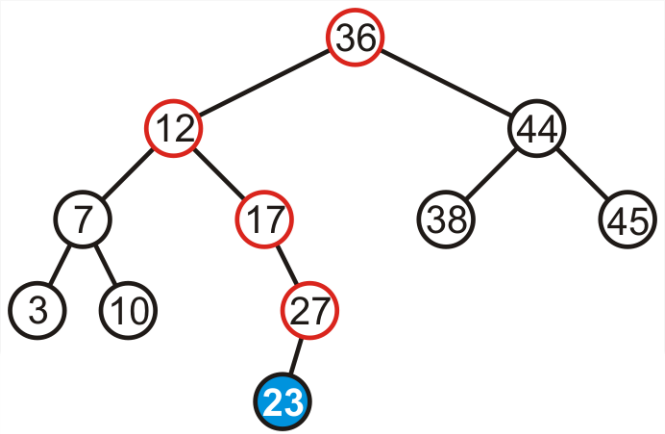


# AVL Tree

## Maintaining Balance

The heights of each of the sub-trees from here to the root are increased by one

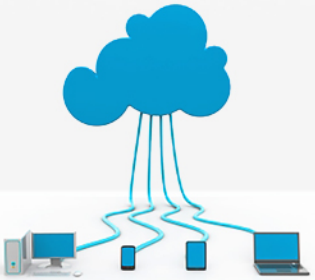


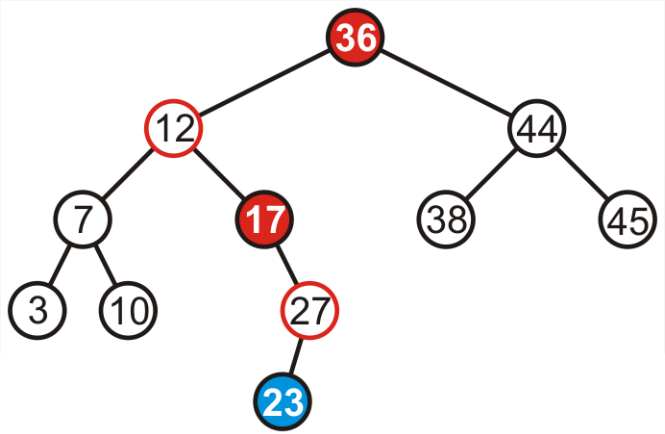


# AVL Tree

## Maintaining Balance

However, only two of the nodes are unbalanced:  17 and 36



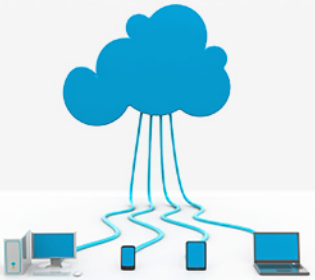


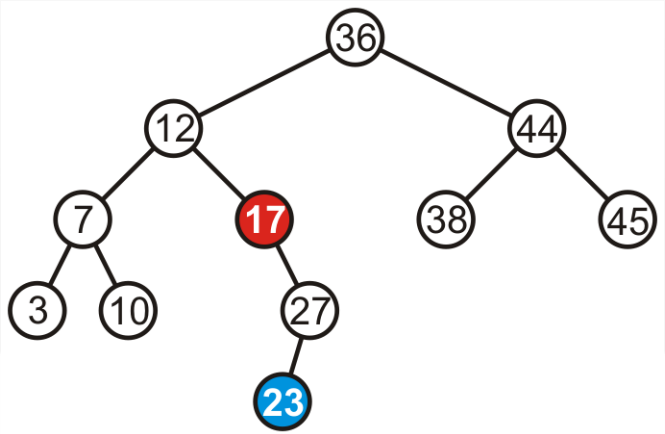
# AVL Tree

## Maintaining Balance

However, only two of the nodes are unbalanced: 17 and 36

- We only have to fix the imbalance at the lowest node

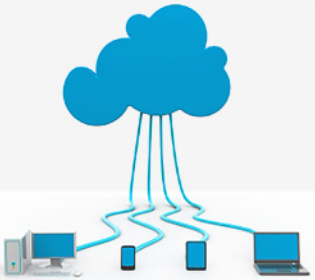


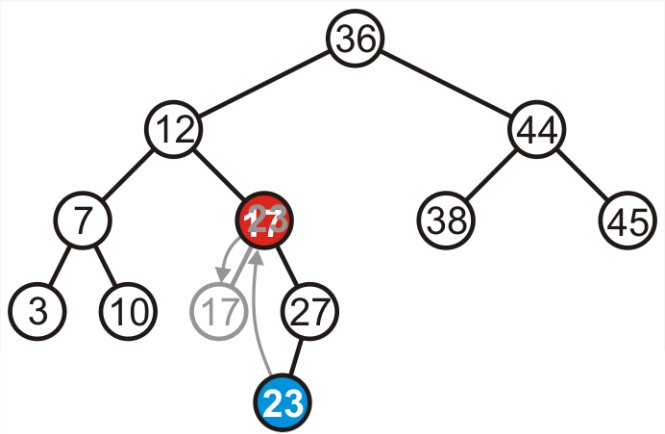


# AVL Tree

## Maintaining Balance

We can promote 23 to where 17 is, and make 17 the left child of 23



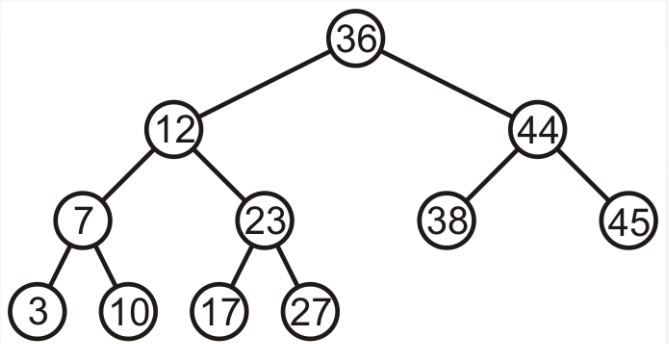


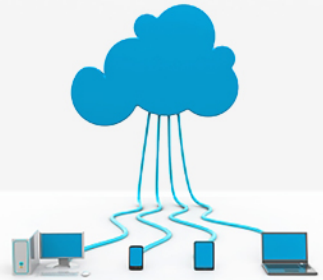
# AVL Tree

## Maintaining Balance

Thus, that node is no longer unbalanced

Incidentally, neither is the root now balanced again, too



# AVL Tree

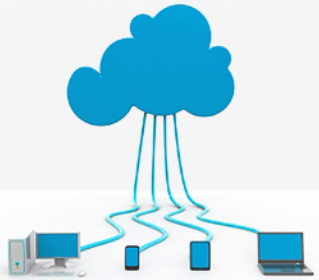
## Maintaining Balance

Only insert and erase may change the height

- ✓ This is the only place we need to update the height
- ✓ These algorithms are already recursive

To maintain AVL balance, observe that:

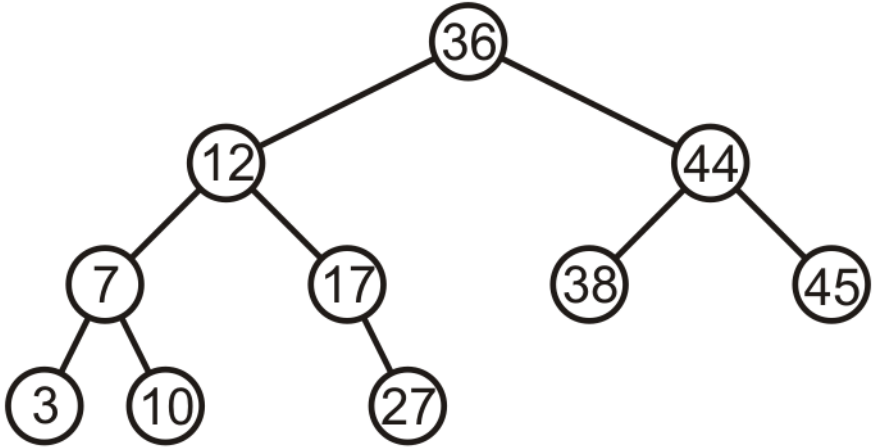
- ✓ Inserting a node can increase the height of a tree by at most 1
- ✓ Removing a node can decrease the height of a tree by at most 1

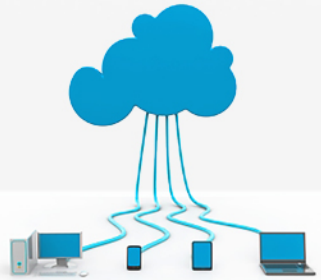


# AVL Tree

## Maintaining Balance

Consider this AVL tree



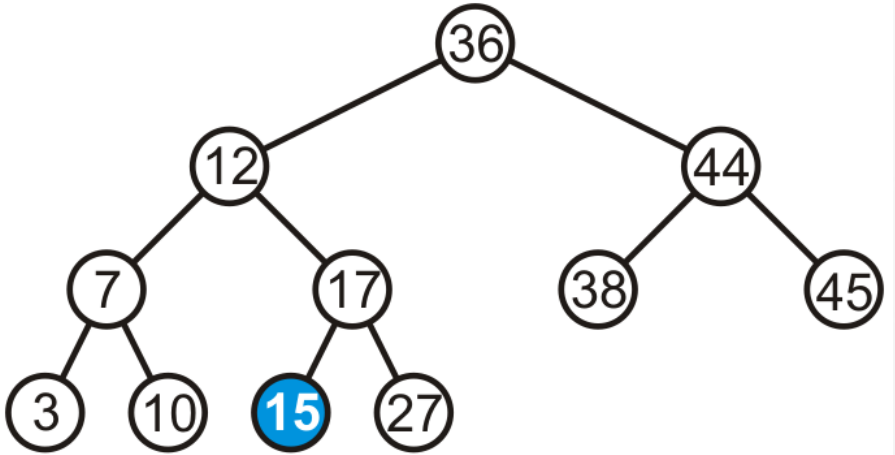



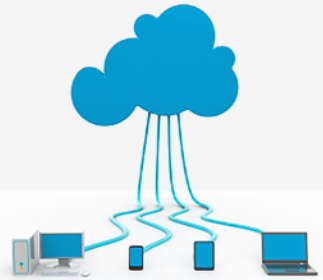
# AVL Tree

## Maintaining Balance

Consider inserting 15 into this tree

- In this case, the heights of none of the trees change

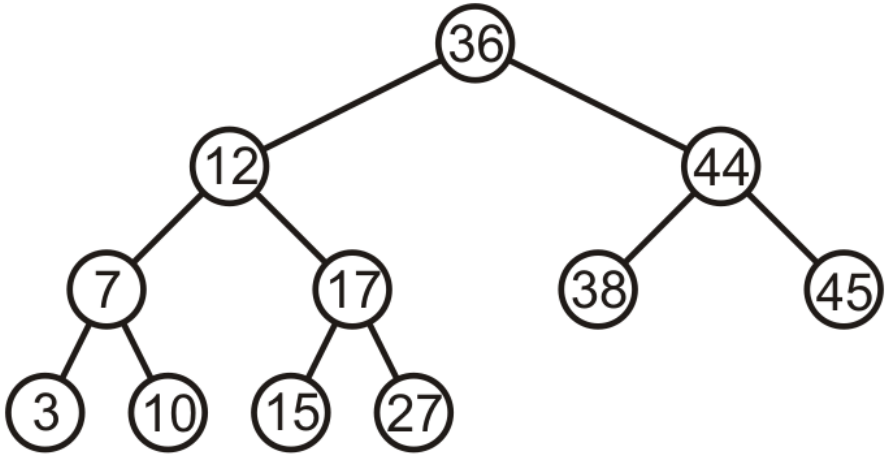


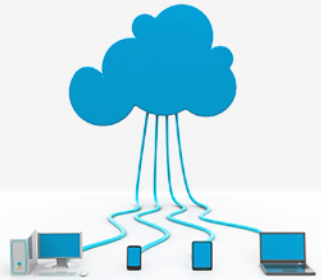



# AVL Tree

## Maintaining Balance

The tree remains balanced



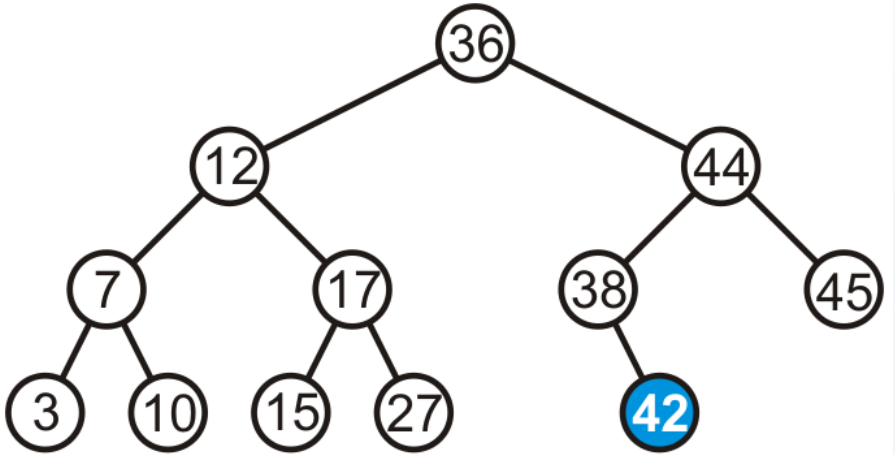



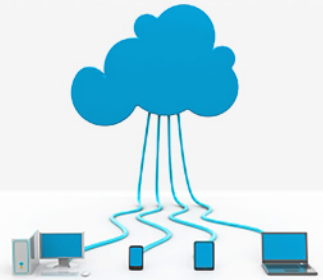
# AVL Tree

## Maintaining Balance

Consider inserting 42 into this tree

- In this case, the heights of none of the trees change



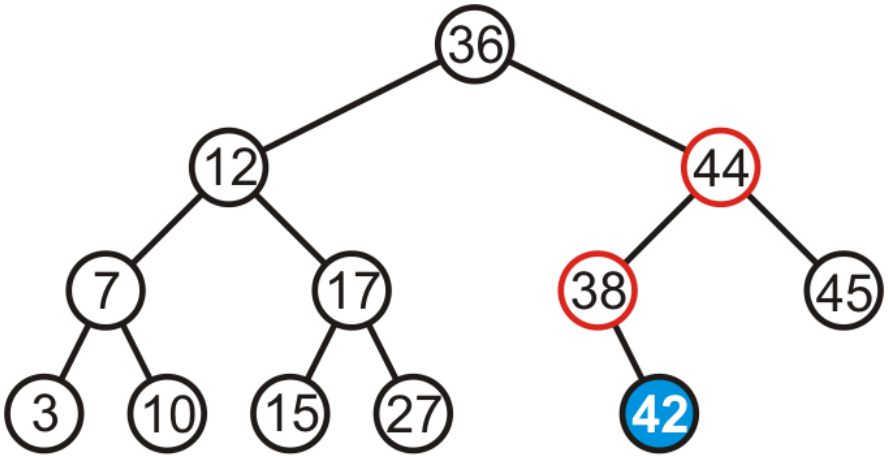



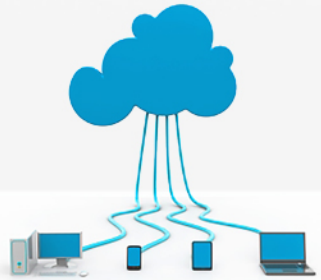
# AVL Tree

## Maintaining Balance

Consider inserting 42 into this tree

- Now we see the heights of two sub-trees have increased by one
- The tree is still balanced



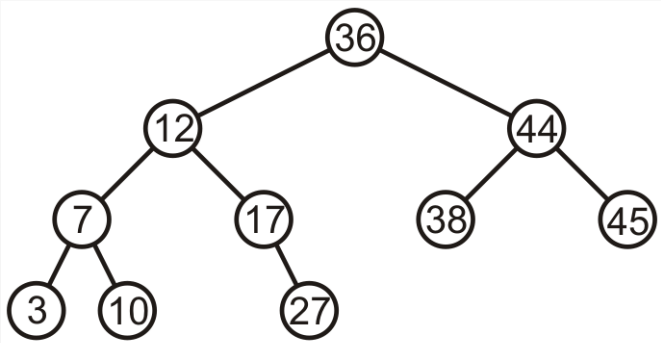



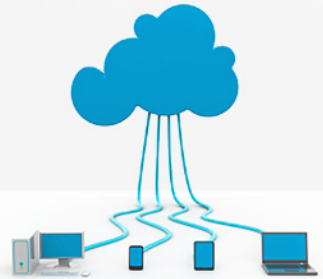
# AVL Tree

## Maintaining Balance

If a tree is AVL balanced, for an insertion to cause an imbalance:

- ✓ The heights of the sub-trees must differ by 1
- ✓ The insertion must increase the height of the deeper sub-tree by 1

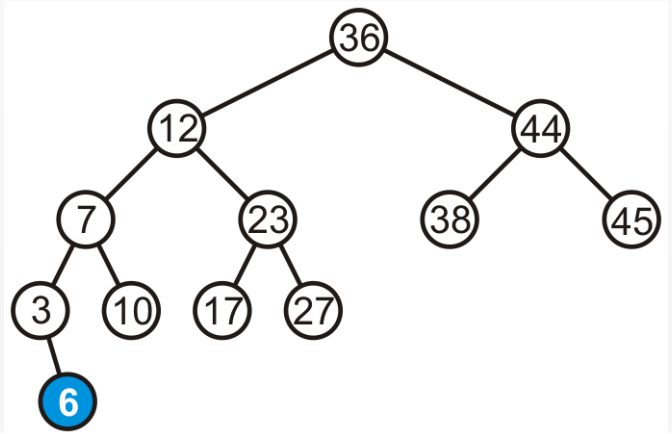


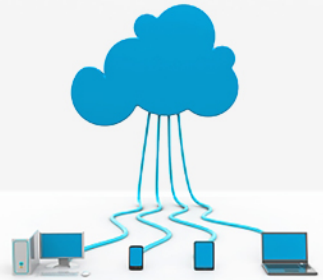



# AVL Tree

## Case 1:

Consider adding 6:

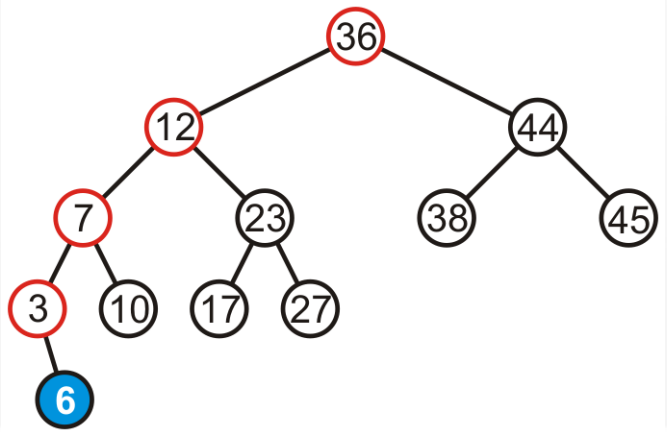


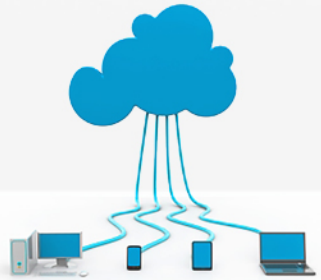



# AVL Tree

## Case 1:

The height of each of the trees in the path back to the root are increased by one



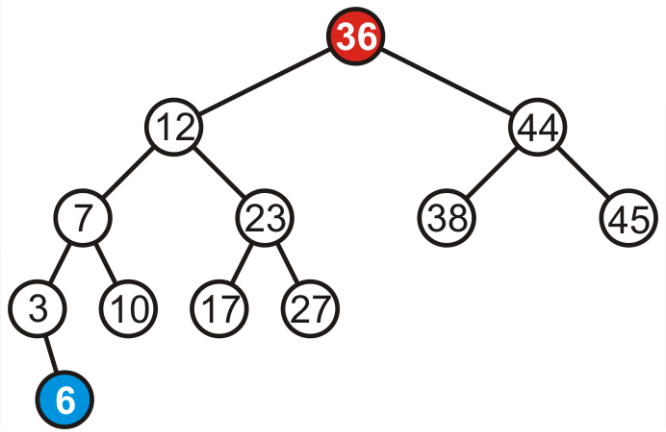



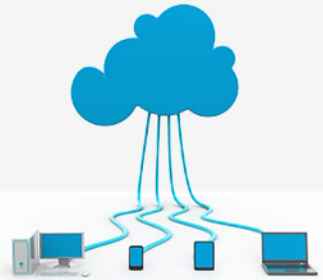
# AVL Tree

## Case 1:

The height of each of the trees in the path back to the root are increased by one

However, only the root node is no unbalanced



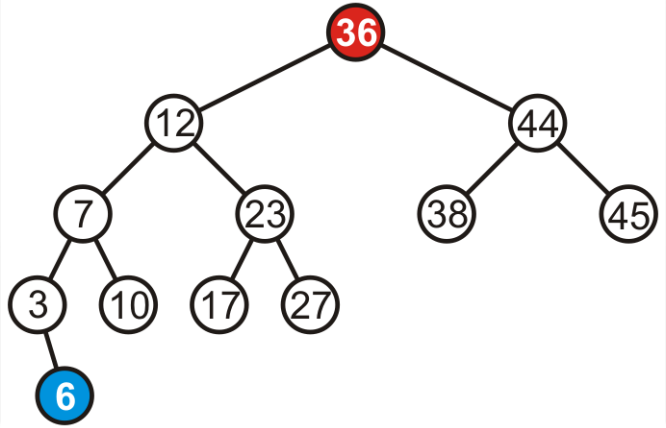
# AVL Tree

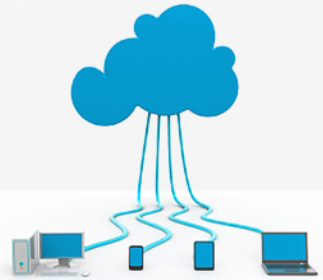
## Case 1:

The height of each of the trees in the path back to the root are increased by one

However, only the root node is no unbalanced

To fix this, we will look at the general case…



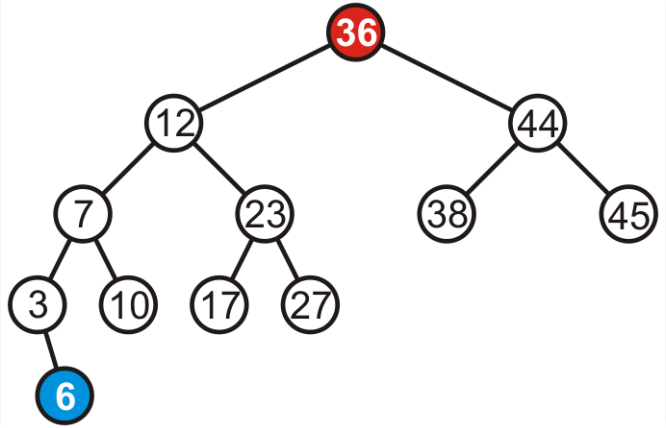
# AVL Tree

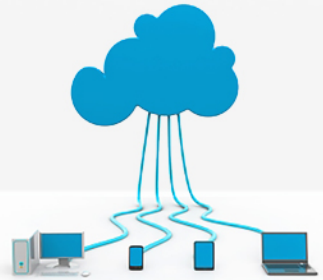
## Case 1:

The height of each of the trees in the path back to the root are increased by one

However, only the root node is no unbalanced

To fix this, we will look at the general case…

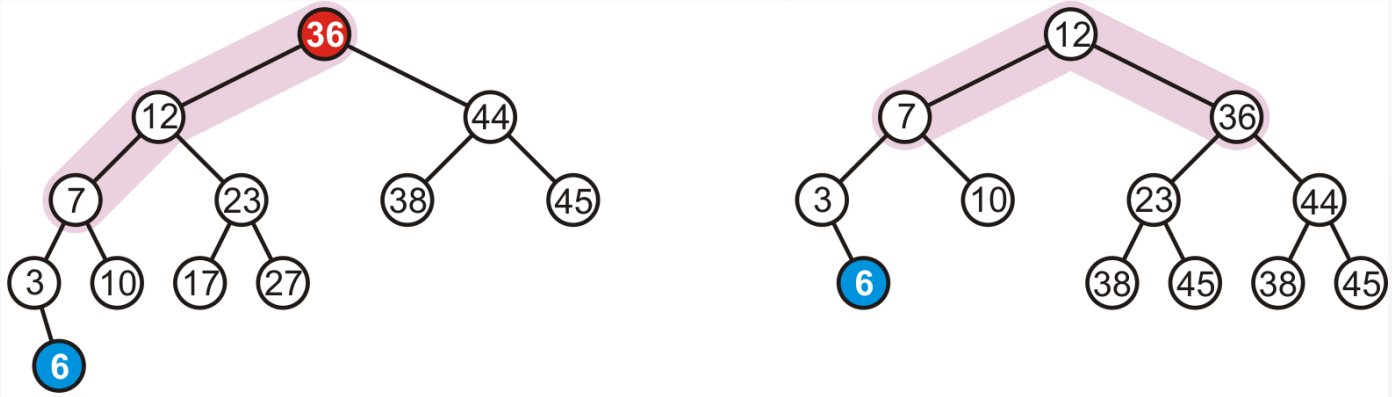


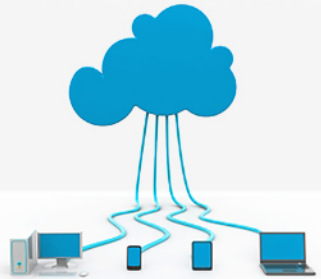



# AVL Tree

## Case 1:

In our example case, the correction

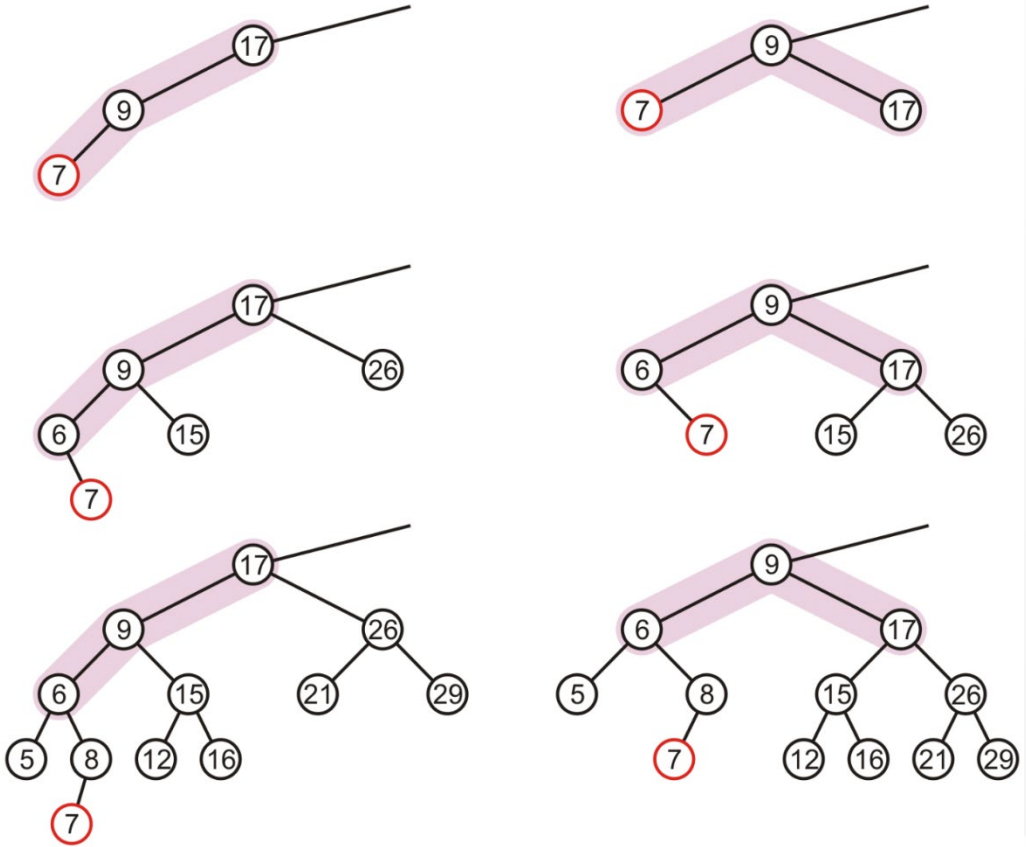


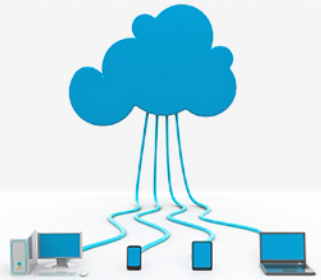



# AVL Tree

## Case 1:

In our three sample cases with $h = -1, 0,$ and $1$, the node is now balanced and the same height as the tree before the insertion

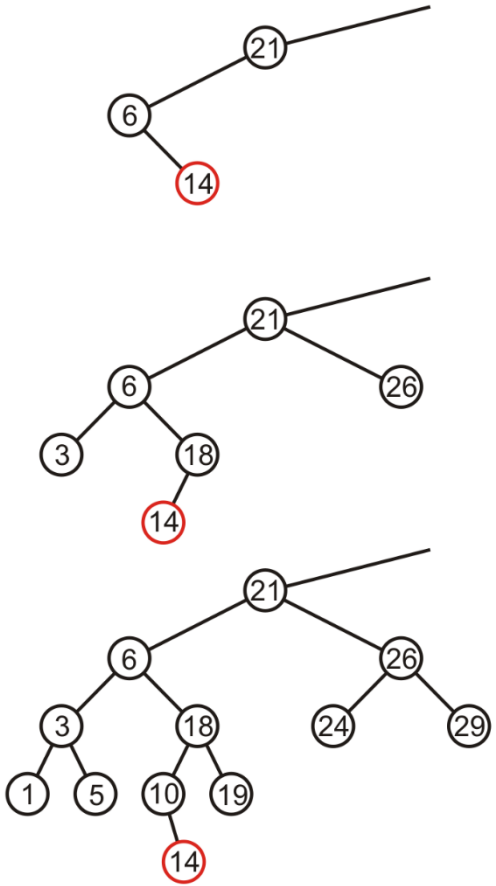


# AVL Tree

## Case 2:

Here are examples of when the insertion of 14 may cause this situation when $h = -1$, $0$, and $1$

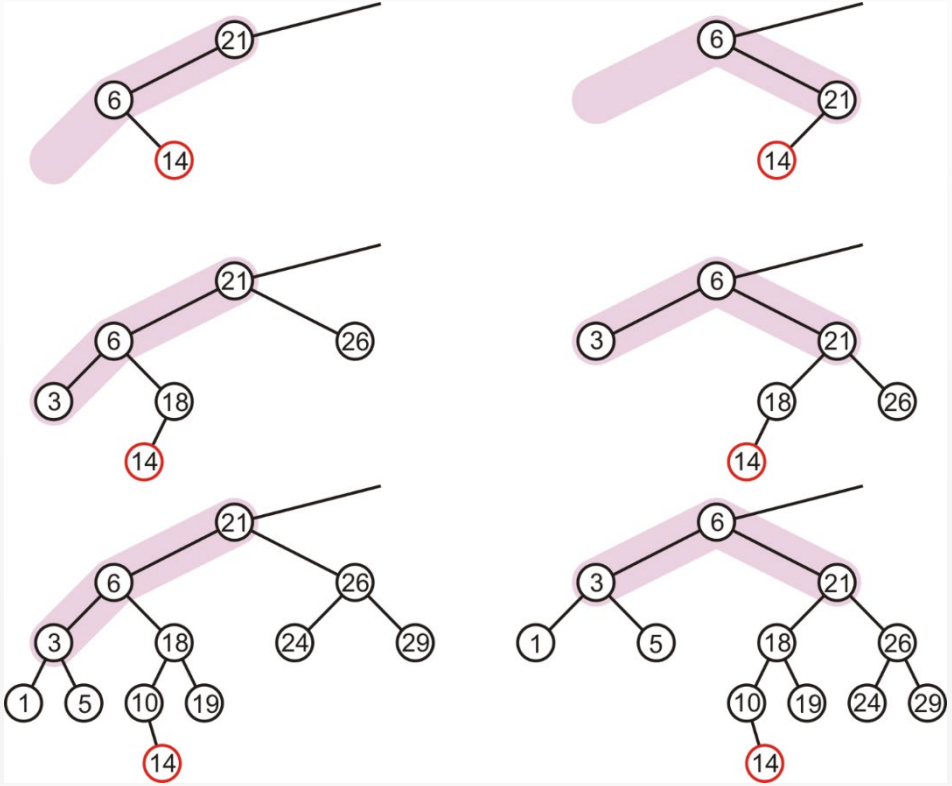


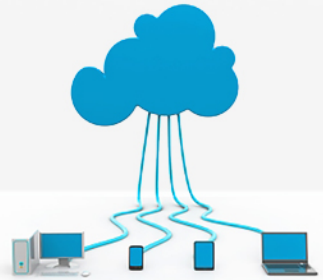

# AVL Tree

## Case 2:

In our three sample cases with $h = -1, 0,$ and $1$, doing the same thing as before results in a tree that is still unbalanced…

- The imbalance is just shifted to the other side

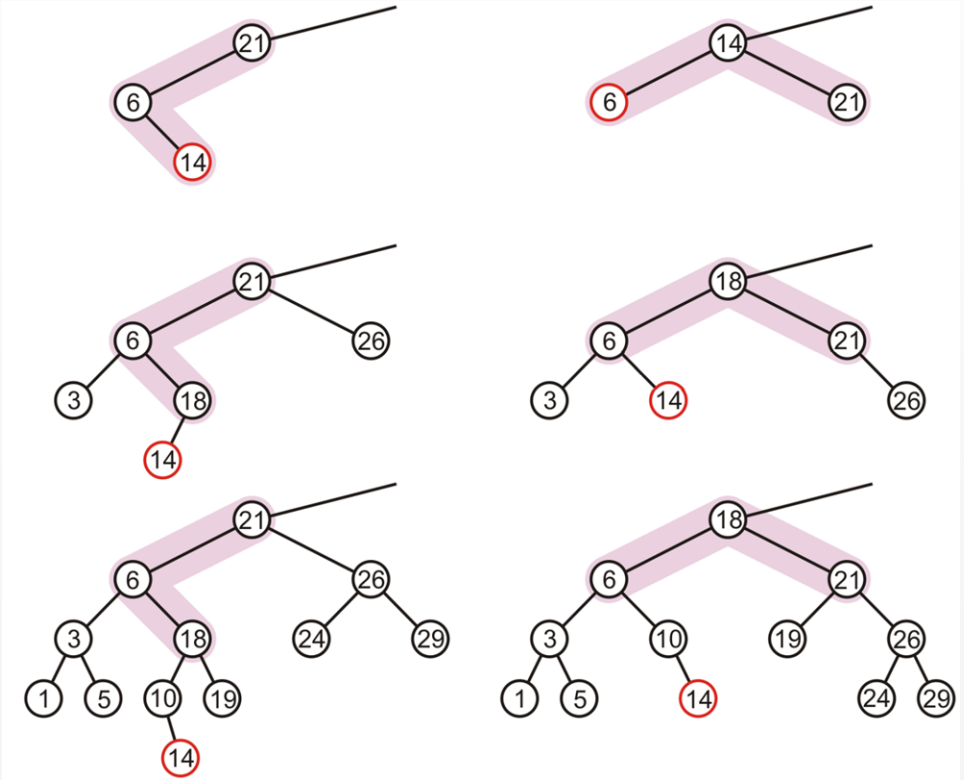


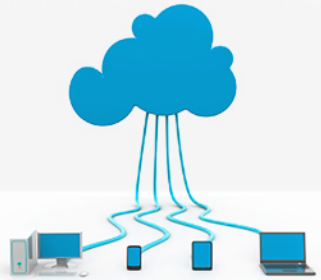



# AVL Tree

## Case 2:

In our three sample cases with $h = -1$, $0$, and $1$, the node is now balanced and the same height as the tree before the insertion



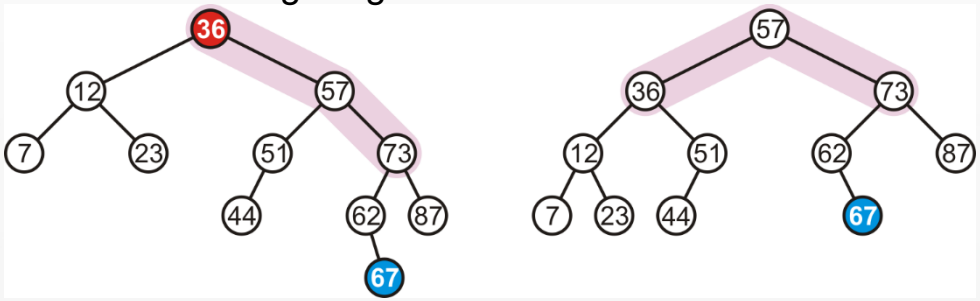



# AVL Tree

## Summary

There are two symmetric cases to those we have examined:

- Insertions into the right-right sub-tree




- Insertions into either the right-left sub-tree