# COMP90038 Algorithms and Complexity

## Balanced Trees

Douglas Pires

Lecture 15

Semester 1, 2021

# Approaches to Balanced Binary Search Trees

To optimise the performance of BST search, it is important to keep trees (reasonably) balanced.

Instance simplification approaches: Self-balancing trees

- AVL trees
- Red-black trees
- Splay trees

Representational changes:

- 2–3 trees
- 2–3–4 trees
- B-trees

# AVL Trees

Named after Adelson-Velsky and Landis.

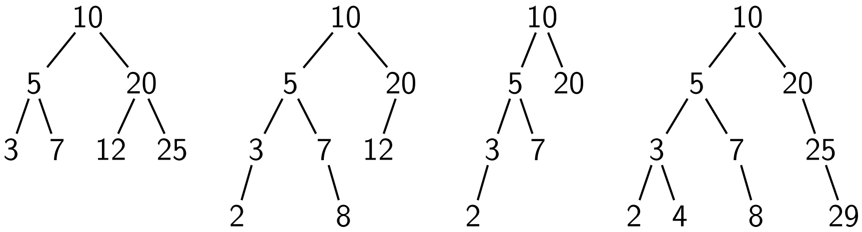
Recall that we defined the height of the empty tree as -1.

For a binary (sub-) tree, let the balance factor be the difference between the height of its left sub-tree and that of its right sub-tree.

An AVL tree is a BST in which the balance factor is -1, 0, or 1, for every sub-tree.

# AVL Trees: Examples and Counter-Examples

Which of these are AVL trees?



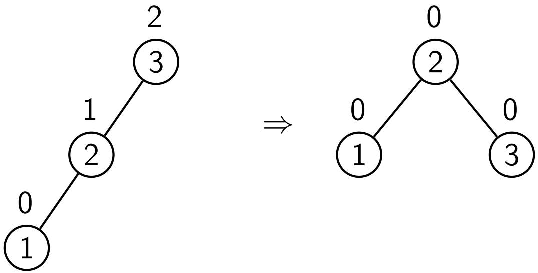
# Building an AVL Tree

As with standard BSTs, insertion of a new node always takes place at the fringe of the tree.

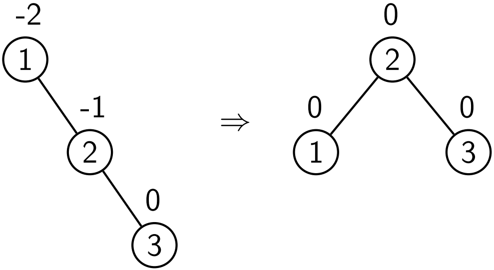
If insertion of the new node makes the AVL tree unbalanced (some nodes get balance factors of 2 or -2), transform the tree to regain its balance.

Regaining balance can be achieved with one or two simple, local transformations, so-called rotations.

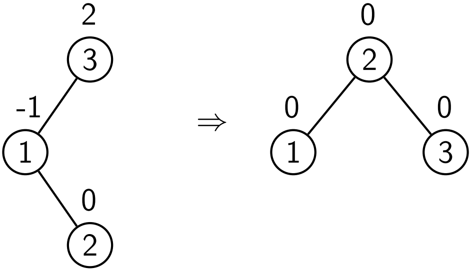
# AVL Trees: R-Rotation



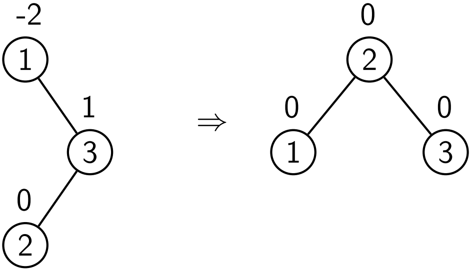
# AVL Trees: L-Rotation



# AVL Trees: LR-Rotation

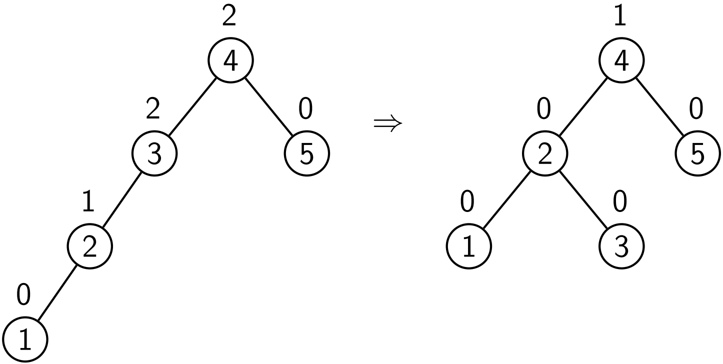


# AVL Trees: RL-Rotation



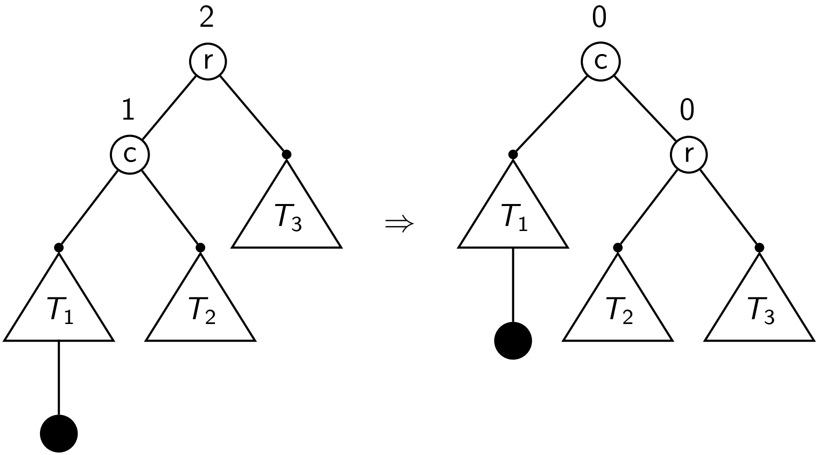
# AVL Trees: Where to Perform the Rotation

Along an unbalanced path, we may have several nodes with balance factor 2 (or -2):



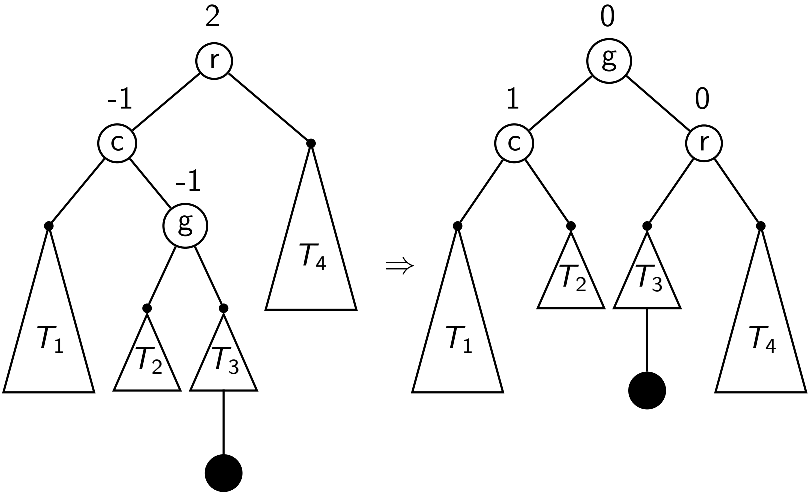

It is always the lowest unbalanced subtree that is re-balanced.

# AVL Trees: The Single Rotation, Generally




This shows an R-rotation; an L-rotation is similar.

# AVL Trees: The Double Rotation, Generally




This shows an LR-rotation; an RL-rotation is similar.

# Properties of AVL Trees

The notion of "balance" that is implied by the AVL condition is sufficient to guarantee that the depth of an AVL tree with $n$ nodes is $\Theta(\log n)$.

For random data, the depth is very close to $\log_2 n$, the optimum.

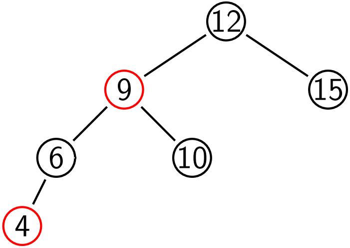
In the worst case, search will need at most 45% more comparisons than with a perfectly balanced BST.

Deletion is harder to implement than insertion, but also $\Theta(\log n)$.

# Other Kinds of Balanced Trees

A red-black tree is a BSTs with a slightly different concept of "balanced". Its nodes are coloured red or black, so that

1. No red node has a red child.
2. Every path from the root to the fringe has the same number of black nodes.




A worst-case red-black tree (the longest path is twice as long as the shortest path).

A splay tree is a BST which is not only self-adjusting, but also adaptive. Frequently accessed items are brought closer to the root, so their access becomes cheaper.

# 2–3 Trees

2–3 trees and 2–3–4 trees are search trees that allow more than one item to be stored in a tree node.

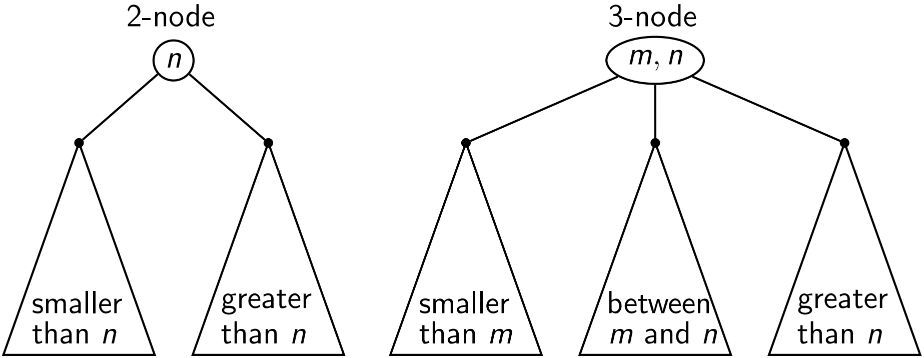
As with BSTs, a node that holds a single item has (at most) two children.

A node that holds two items (a so-called 3-node) has (at most) three children.

And for 2–3–4 trees, a node that holds three items (a 4-node) has (at most) four.

This allows for a simple way of keeping search trees balanced.

# 2-Nodes and 3-Nodes



# Insertion in a 2–3 Tree

To insert a key $k$, pretend that we are searching for $k$.

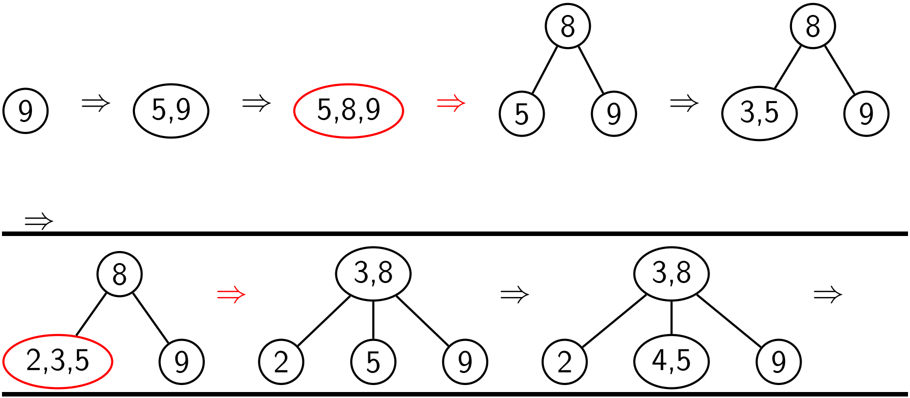
This will take us to a leaf node in the tree, where $k$ should now be inserted; if the node we find there is a 2-node, $k$ can be inserted without further ado.

Otherwise we had a 3-node, and the two inhabitants, together with $k$, momentarily form a node with three elements; in sorted order, call them $k_1$, $k_2$, and $k_3$.

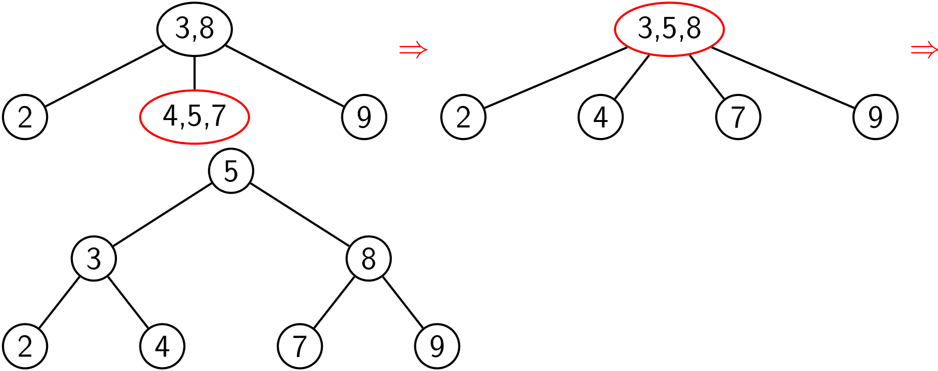
We now split the node, so that $k_1$ and $k_3$ form their own individual 2-nodes. The middle key, $k_2$ is promoted to the parent node.

The promotion may cause the parent node to overflow, in which case it gets split the same way. The only time the tree's height changes is when the root overflows.

# Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



# Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



# Exercise: 2–3 Tree Construction

Build the 2–3 tree that results from inserting these keys, in the given order, into an initially empty tree:

C, O, M, P, U, T, I, N, G

# 2–3 Tree Analysis

Worst case search time results when all nodes are 2-nodes.
The relation between the number $n$ of nodes and the height $h$ is:

$$n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$$

That is, $\log_2(n+1) = h + 1$.

In the best case, all nodes are 3-nodes:

$$n = 2 + 2 \times 3 + 2 \times 3^2 + \ldots + 2 \times 3^h = 3^{h+1} - 1$$

That is, $\log_3(n+1) = h + 1$.

Hence we have $\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$.

Useful formula: $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$, for $a \neq 1$.

# Coming Soon to a Theatre Near You

How to buy time, by spending a bit of space.