COMP90038
Algorithms and Complexity

Lecture 15: Balanced Trees

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

On the previous lecture

- We talked about **transform-and-conquer**, a group of design techniques that:
  - **Modify** the problem to a more **amenable** form, and then
  - **Solve** it using a **known efficient** algorithm

- There are three major variations
  - In **instance simplification** we try to make the problem **easier** through some type of **pre-processing**, typically **sorting**
  - In **representation change** we use a different data structure with better properties
  - In **problem reduction** we solve the instance **as if it was a different problem**

Balanced search trees

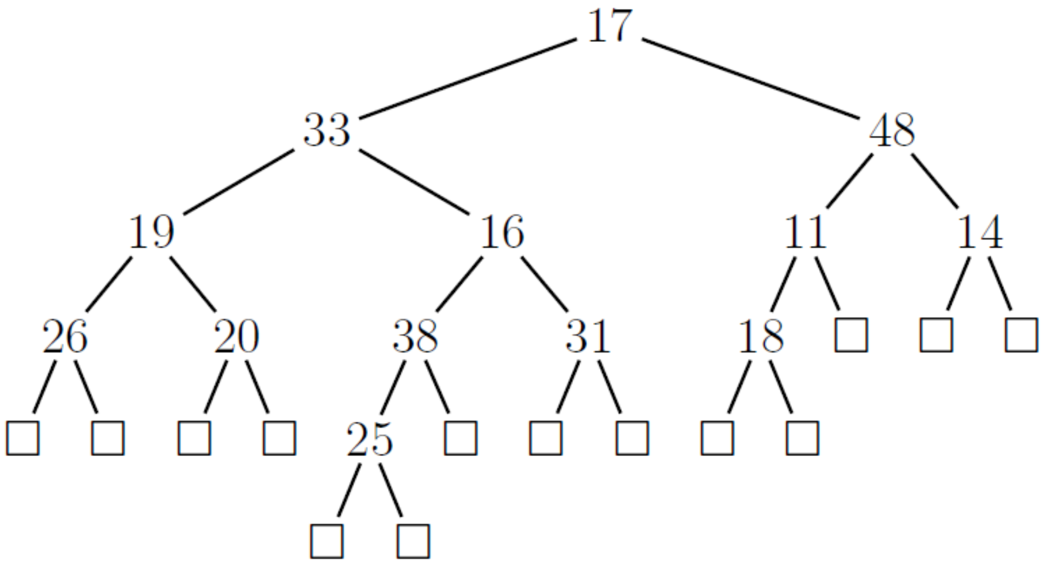
- If a BST with n elements is kept **"reasonably" balanced**, search involves $\Theta(\log n)$ comparisons in the worst case

- If the BST is **unbalanced**, search performance may degrade to be as bad as linear search

- Let's examine two approaches to **maintain** a tree balanced:
  - Instance simplification through self-balancing: **AVL trees**, Red-black trees and Splay trees
  - Representational changes: **2–3 trees**, 2–3–4 trees and B-trees

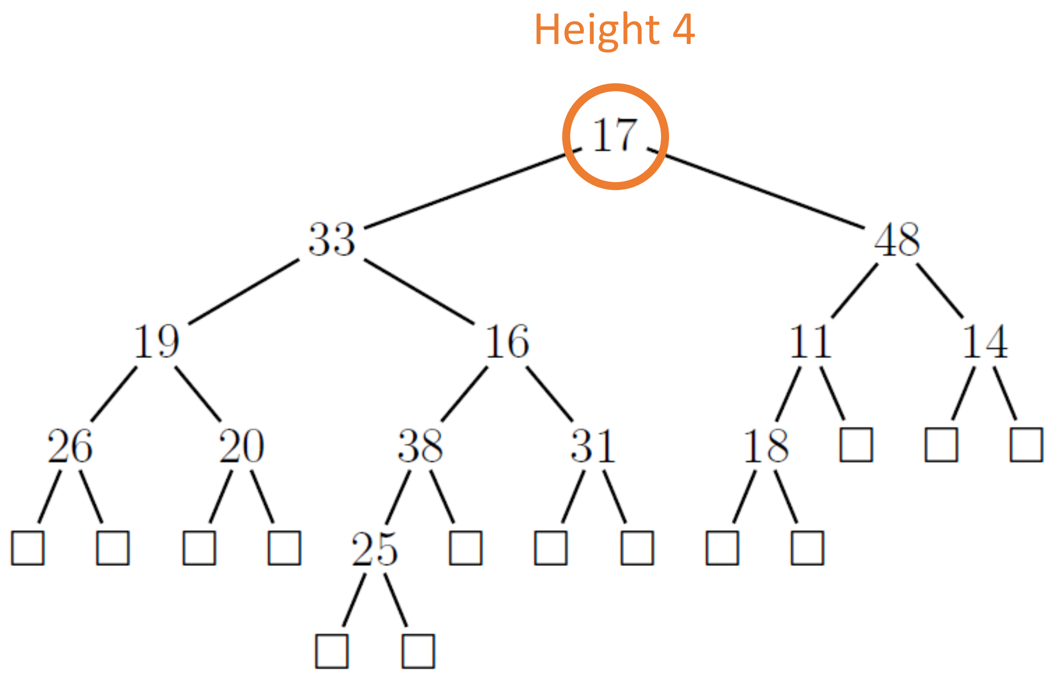
AVL Trees

- On lecture 12, you discussed the **height** of a **binary tree**



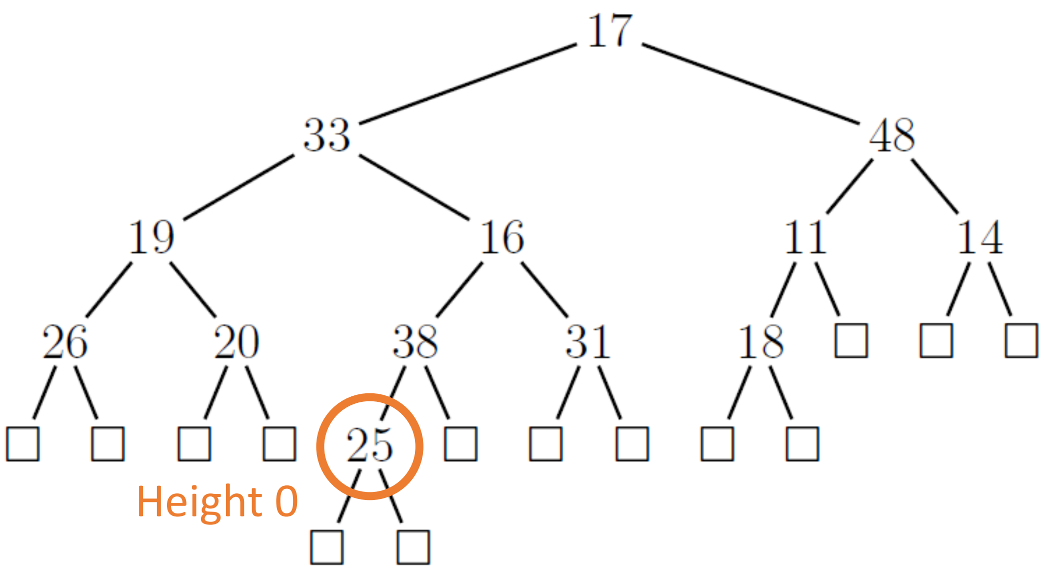
AVL Trees

- On lecture 12, you discussed the **height** of a **binary tree**
  - This **full tree** has a height of 4



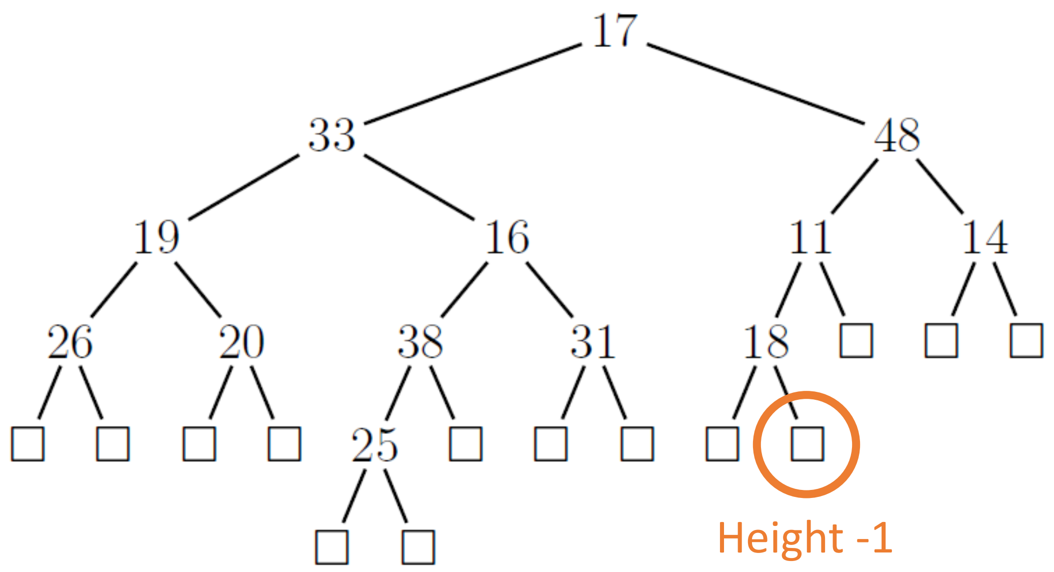
AVL Trees

- On lecture 12, you discussed the **height** of a **binary tree**
  - This **full tree** has a height of 4
  - This **sub-tree** has a height of 0



AVL Trees

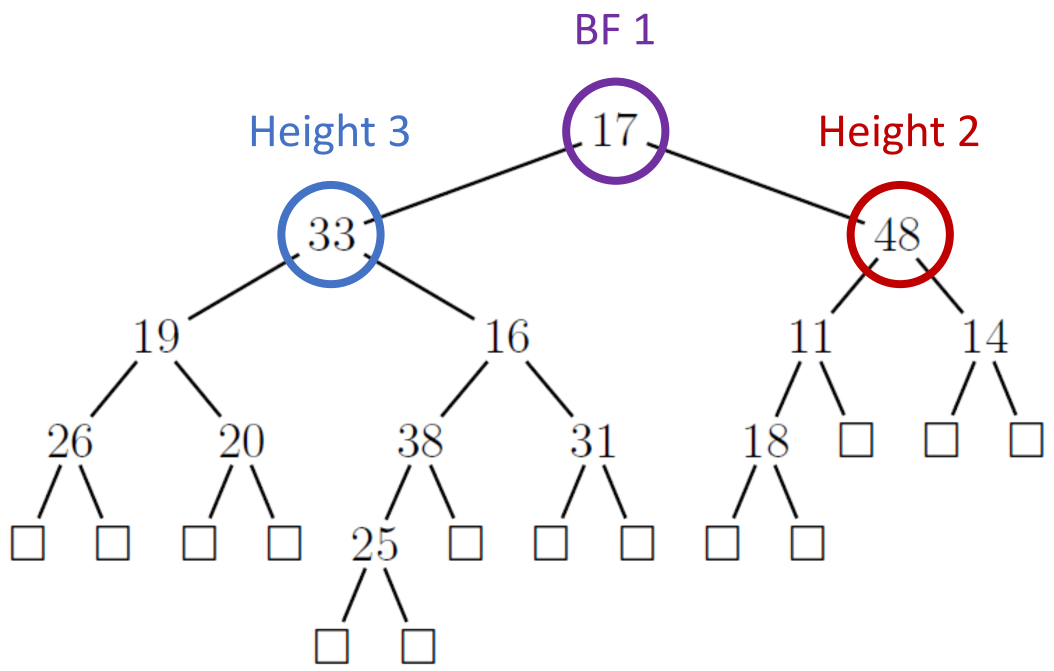
- On lecture 12, you discussed the **height** of a **binary tree**
  - This **full tree** has a height of 4
  - This **sub-tree** has a height of 0
  - An **empty tree** has a height of -1



AVL Trees

- On lecture 12, you discussed the **height** of a **binary tree**
  - This **full tree** has a height of 4
  - This **sub-tree** has a height of 0
  - An **empty tree** has a height of -1

- For any binary tree, the **balance factor (BF)** is the **difference in height** between the left and the right sub-trees

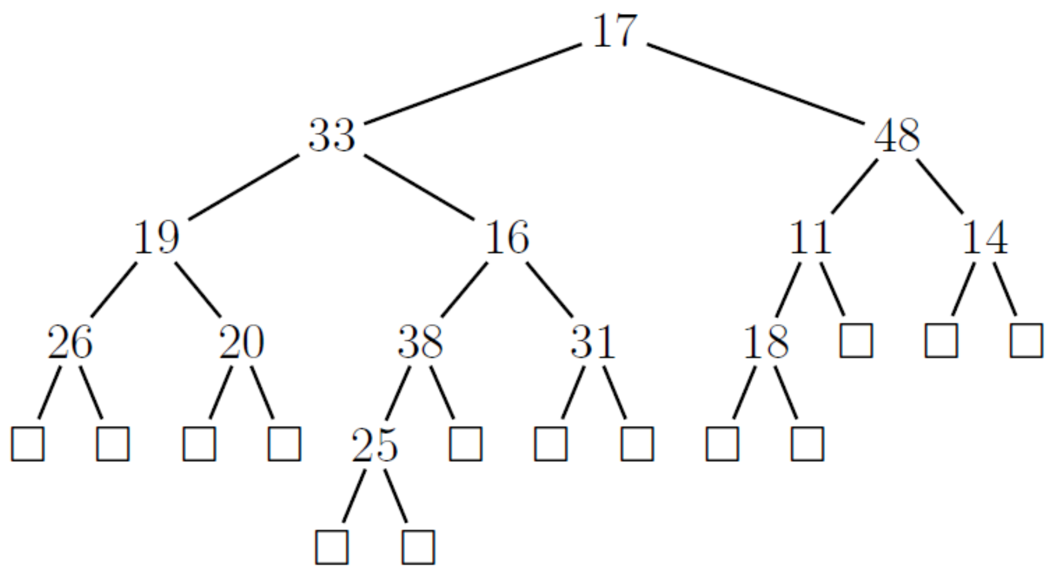


AVL Trees

- On lecture 12, you discussed the **height** of a **binary tree**
  - This **full tree** has a height of 4
  - This **sub-tree** has a height of 0
  - An **empty tree** has a height of -1

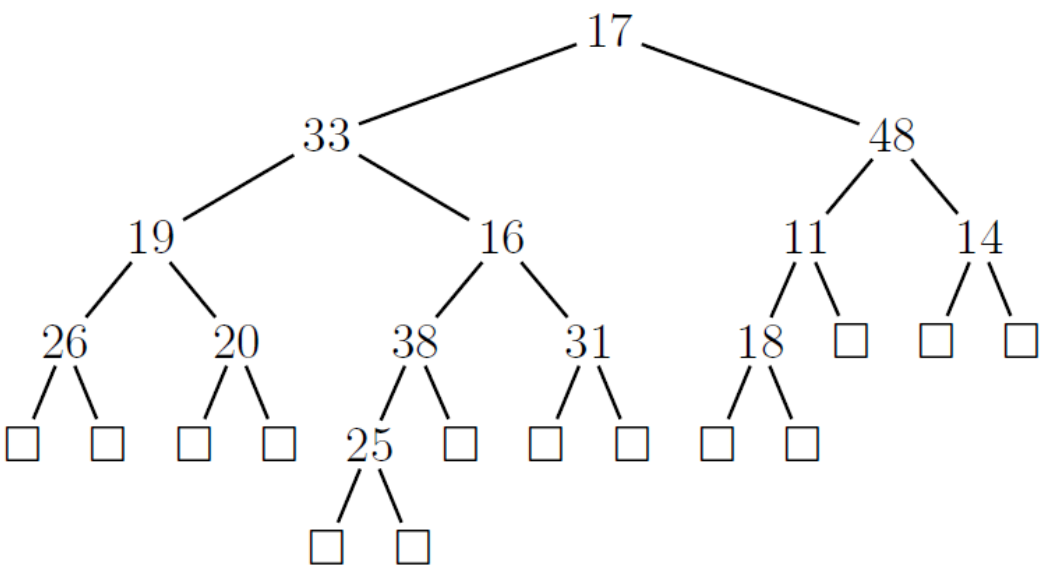
- For any binary tree, the **balance factor (BF)** is the **difference in height** between the left and the right sub-trees

- Named after Adelson-Velsky and Landis, an **AVL tree** is a BST in which the balance factor is -1, 0, or 1, for every sub-tree



AVL Trees

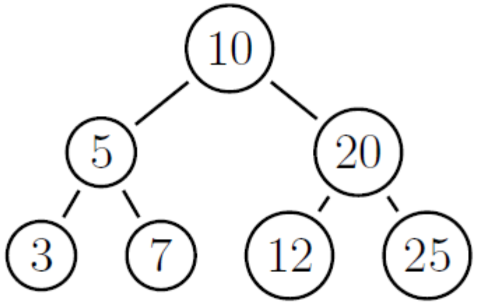
- On lecture 12, you discussed the **height** of a **binary tree**
  - This **full tree** has a height of 4
  - This **sub-tree** has a height of 0
  - An **empty tree** has a height of -1
- For any binary tree, the **balance factor (BF)** is the **difference in height** between the left and the right sub-trees
- Named after Adelson-Velsky and Landis, an **AVL tree** is a BST in which the balance factor is -1, 0, or 1, for every sub-tree
  - Is the tree on the slide an AVL tree?

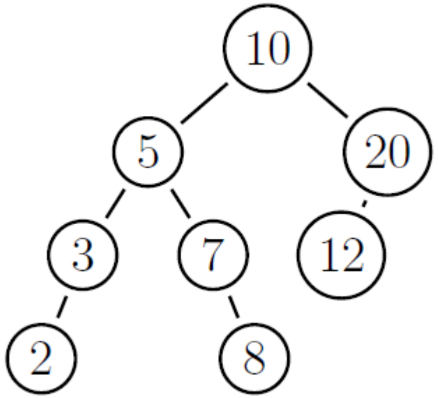


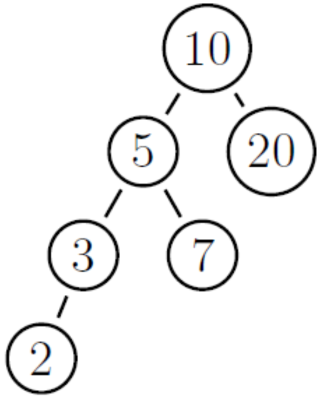
AVL Trees

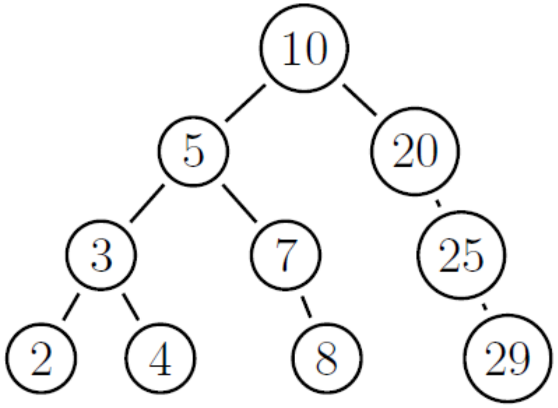
- Let's take a few **minutes** to calculate the **balance factors** for these BSTs

- Which ones are AVLs?









How to build an AVL Tree?

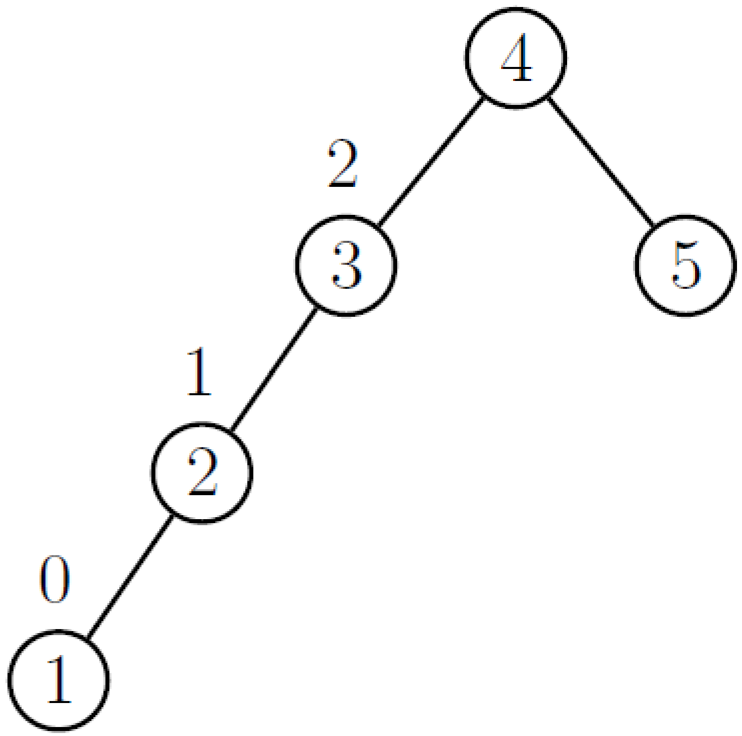
- As with standard **BSTs**, a new element k is inserted by searching for it. Once an **empty** sub-tree is found, we insert k in that position.

- If the insertion of k makes the AVL tree unbalanced, i.e., some nodes get balance factors of ± 2, then rebalance the tree.
  - Always rebalance the **lowest** unbalanced subtree.

- Rebalancing is achieved by simple, local transformations called **rotations**.
  - Rotations preserve the basic requirements for a BST.

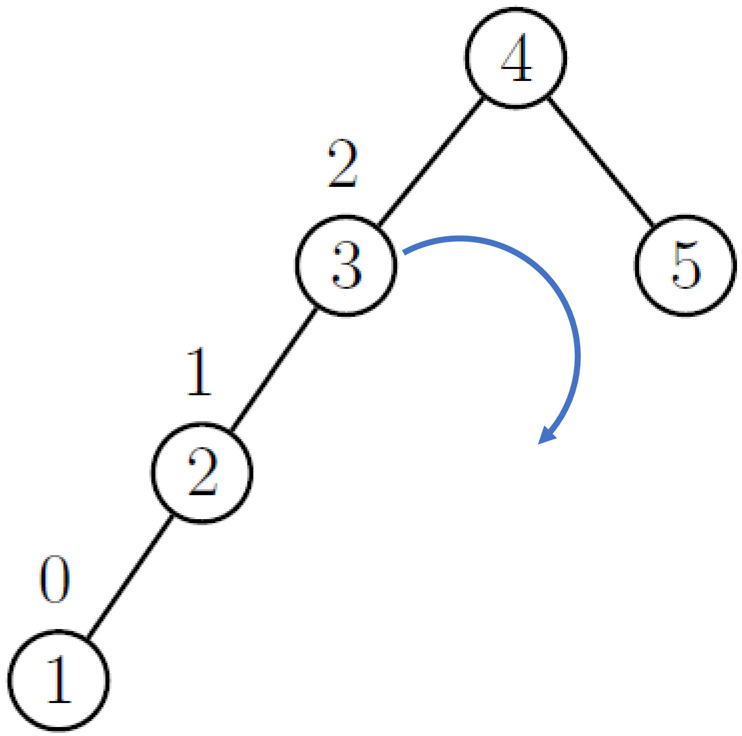
Single rotations

R-Rotation



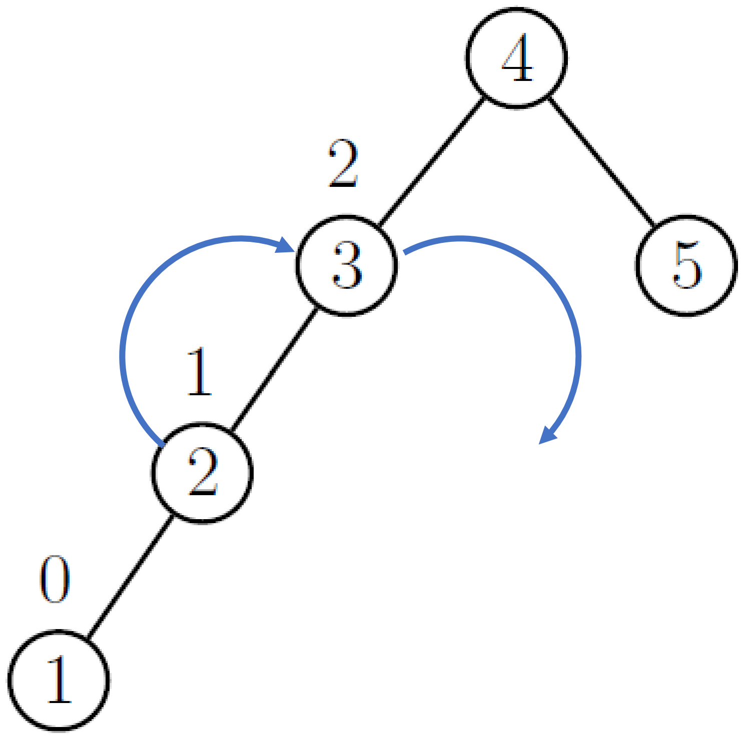
Single rotations

R-Rotation



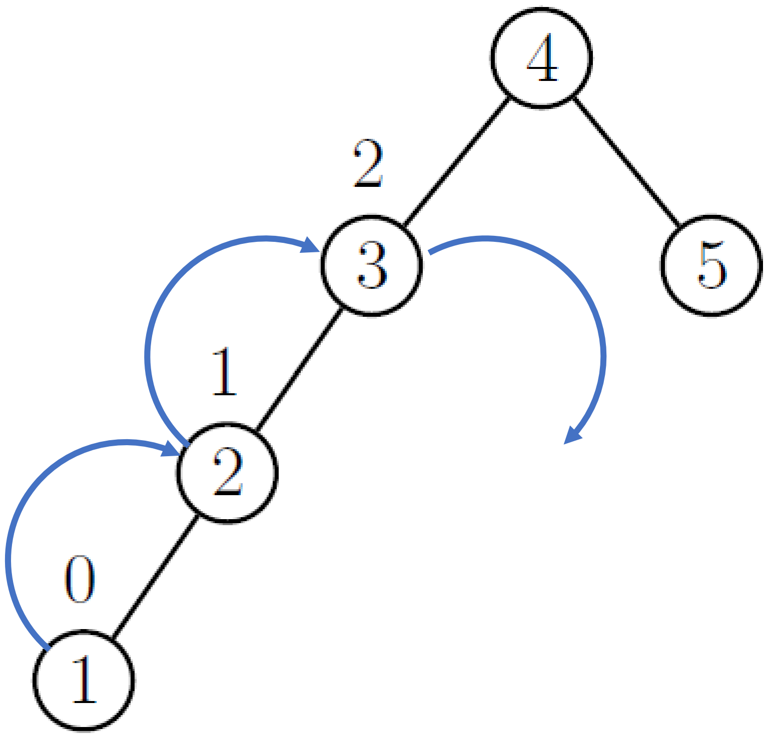
Single rotations

R-Rotation



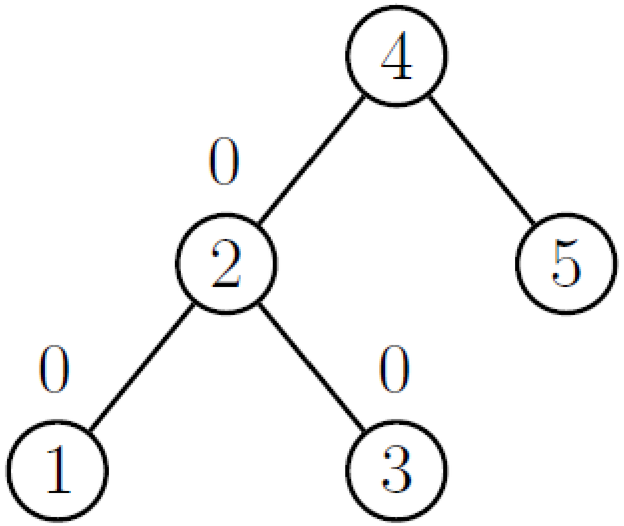
Single rotations

R-Rotation



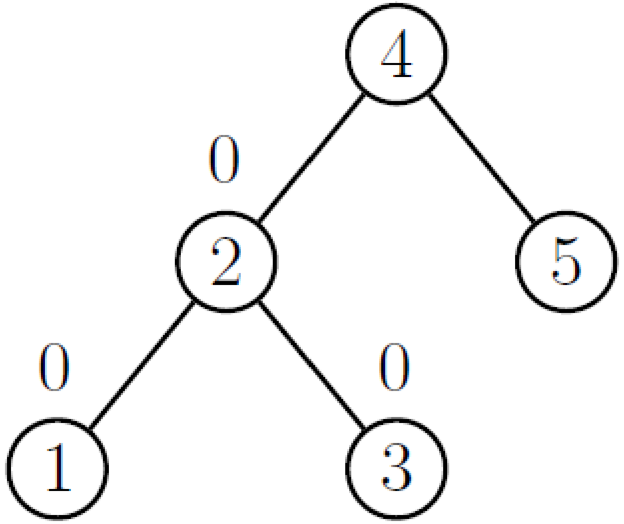
Single rotations

R-Rotation

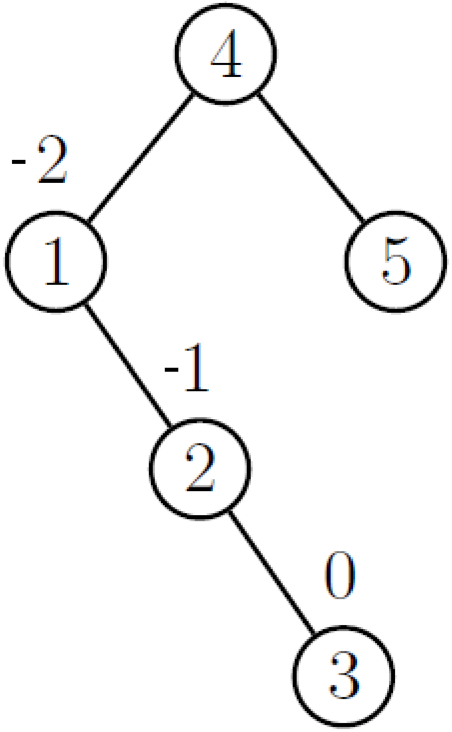


Single rotations

R-Rotation

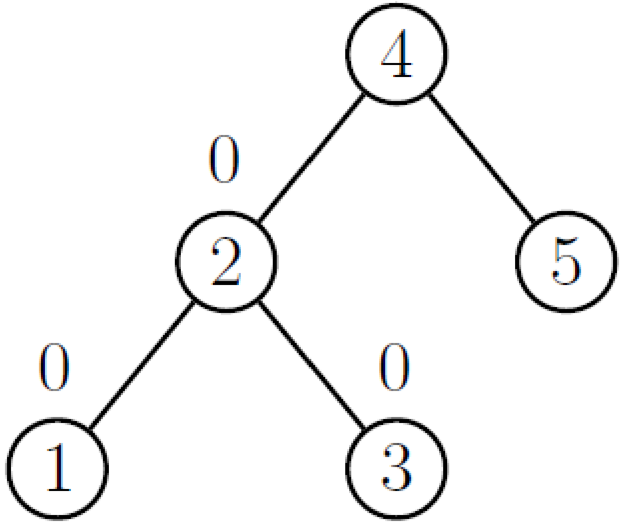


L-Rotation

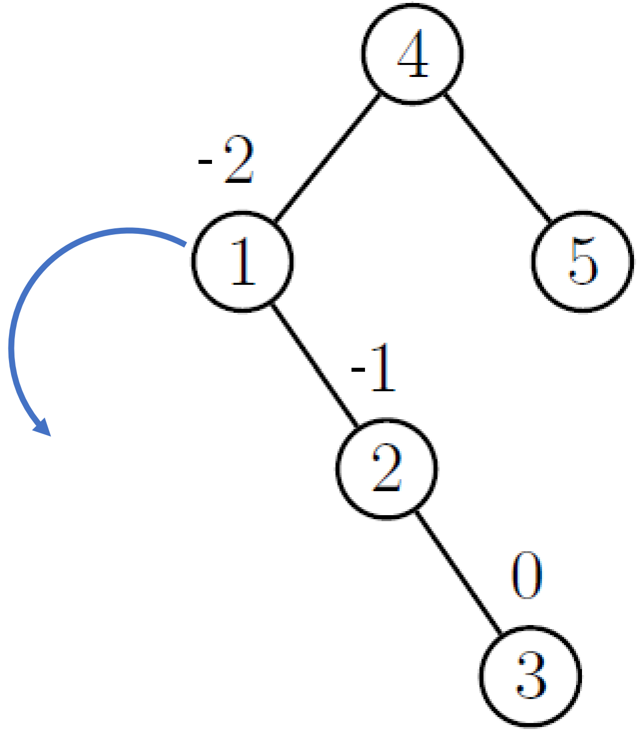


Single rotations

R-Rotation

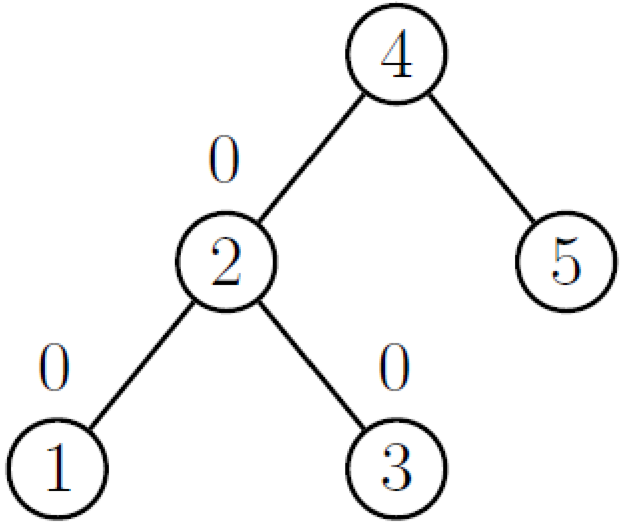


L-Rotation

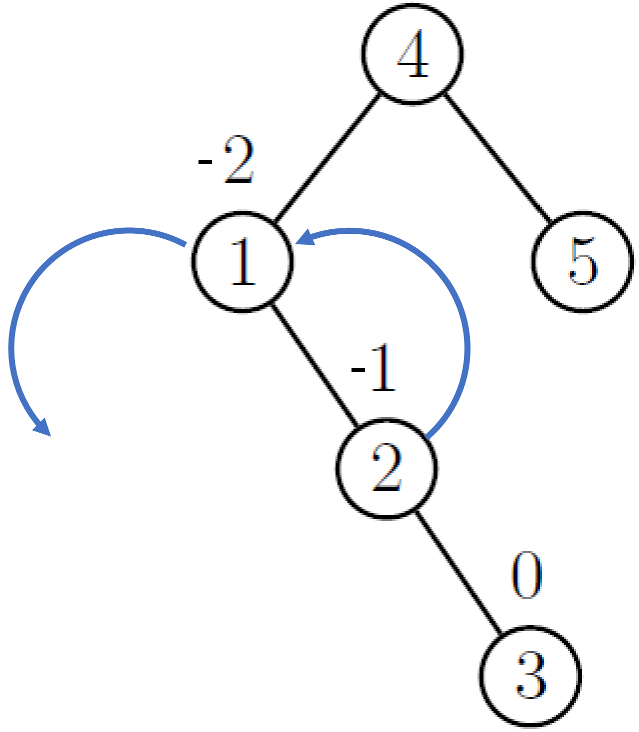


Single rotations

R-Rotation

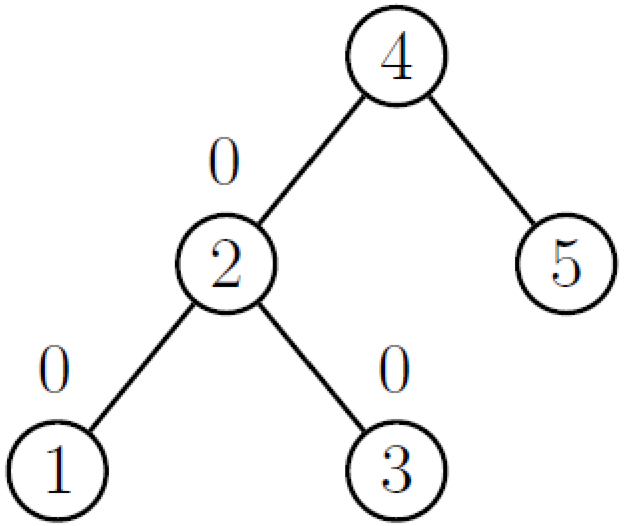


L-Rotation

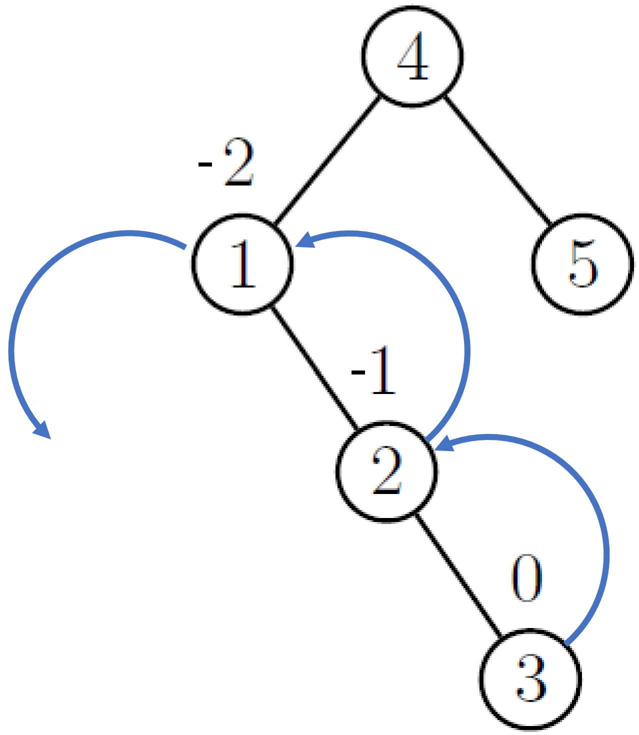


Single rotations

R-Rotation

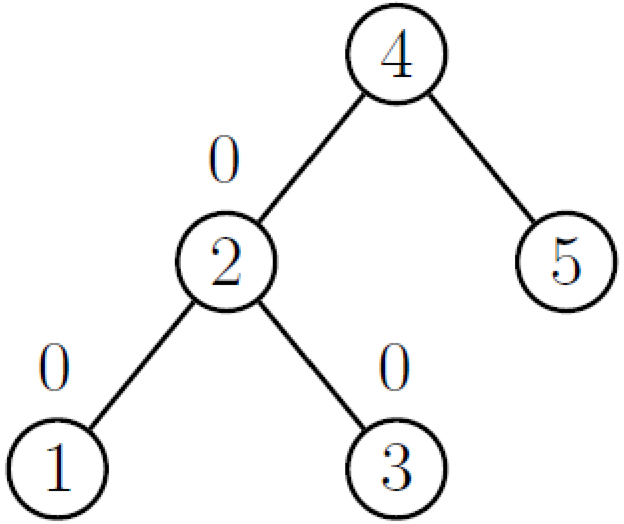


L-Rotation

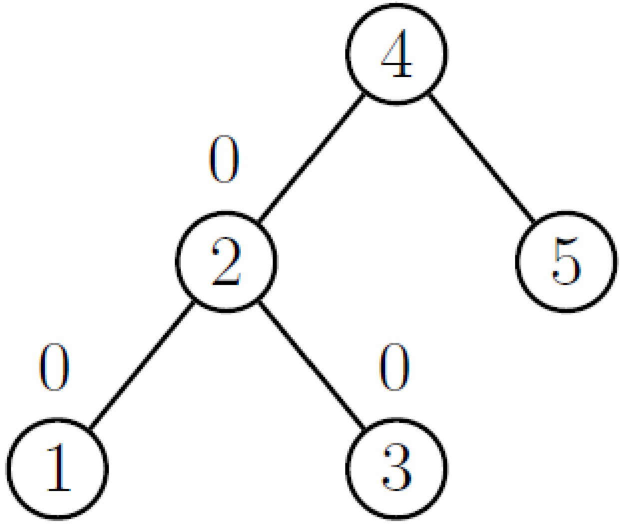


Single rotations

R-Rotation



L-Rotation



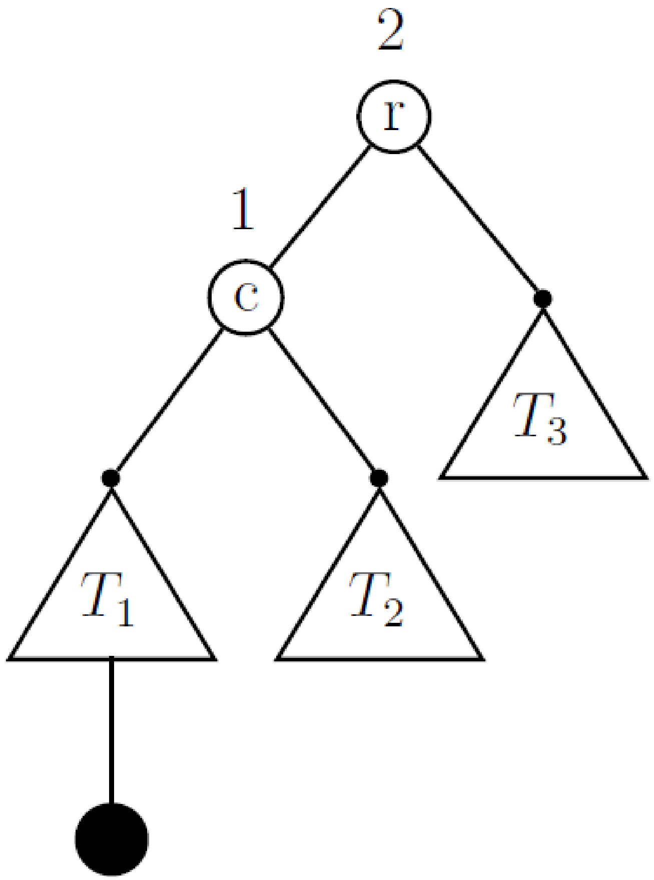
General single rotations

- It is possible that nodes at lower levels (closer to the root) have balance factors of ± 2.

General single rotations

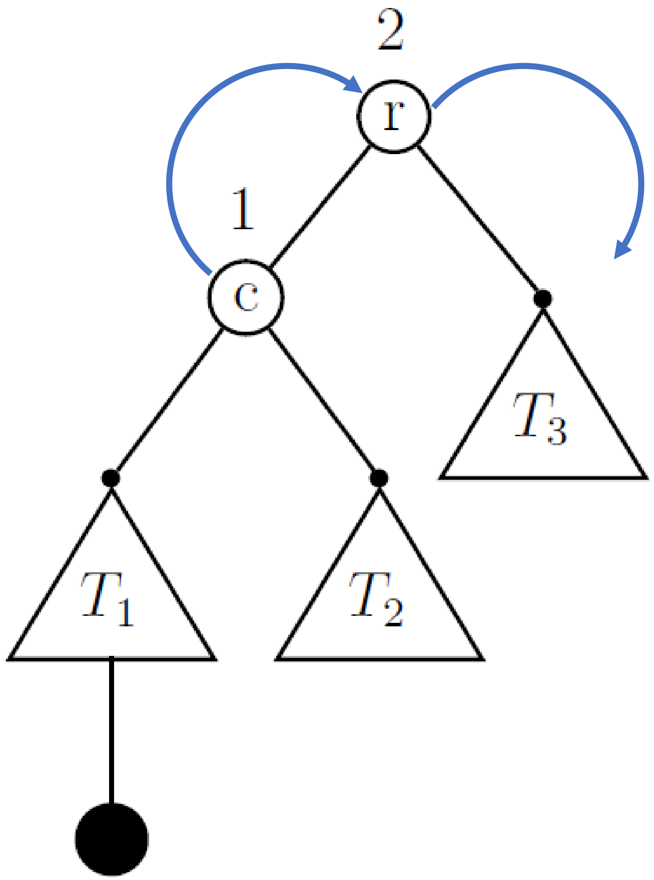
- It is possible that nodes at lower levels (closer to the root) have balance factors of ± 2.

- The diagram to the right uses **triangular** nodes to represent any sub-tree
  - The **black** node corresponds to the inserted element



General single rotations

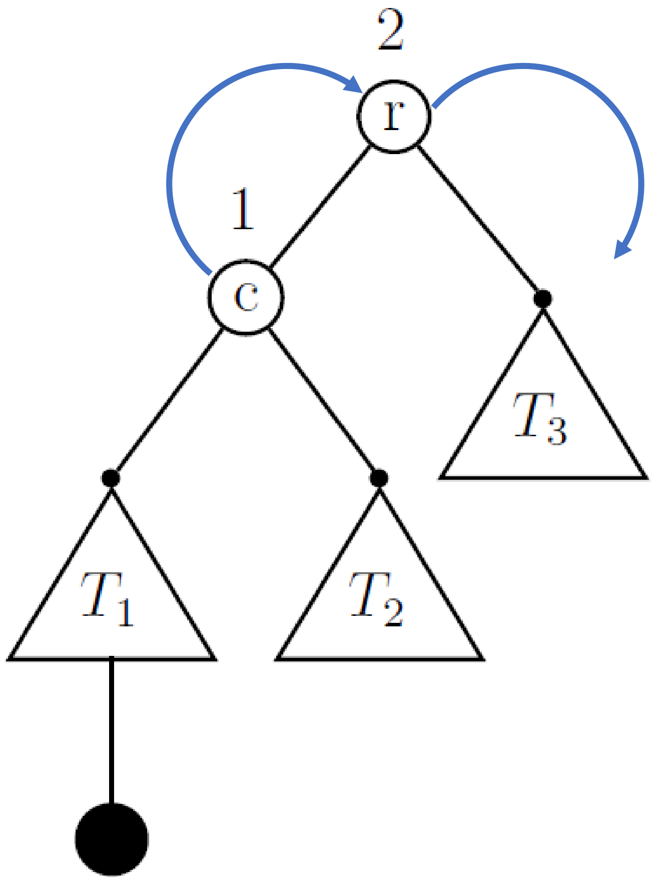
- It is possible that nodes at lower levels (closer to the root) have balance factors of ± 2.
- The diagram to the right uses **triangular** nodes to represent any sub-tree
  - The **black** node corresponds to the inserted element
  - To rebalance the tree we perform a **R-rotation**



General single rotations

- It is possible that nodes at lower levels (closer to the root) have balance factors of ± 2.

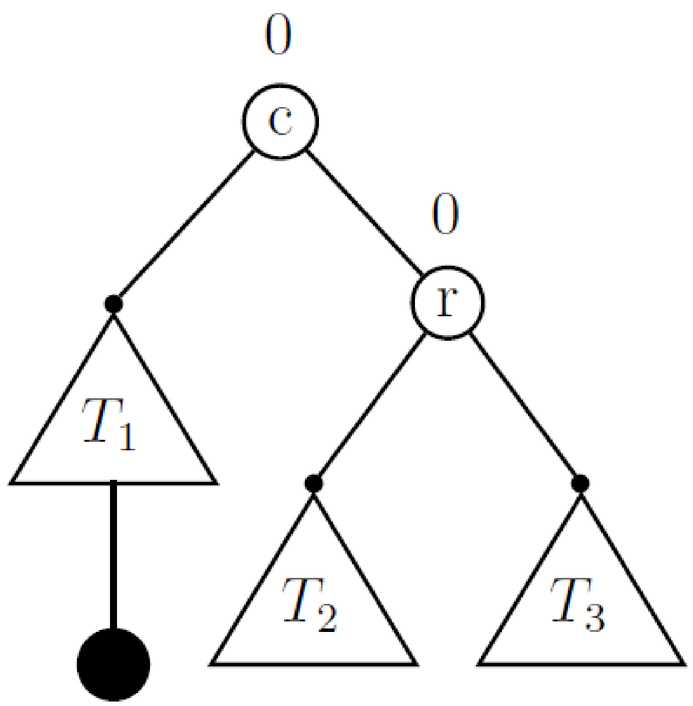
- The diagram to the right uses **triangular** nodes to represent any sub-tree
  - The **black** node corresponds to the inserted element
  - To rebalance the tree we perform a **R-rotation**
  - When r **descends** and c takes its place, T_2 can **no longer** be the **right** child of c



General single rotations

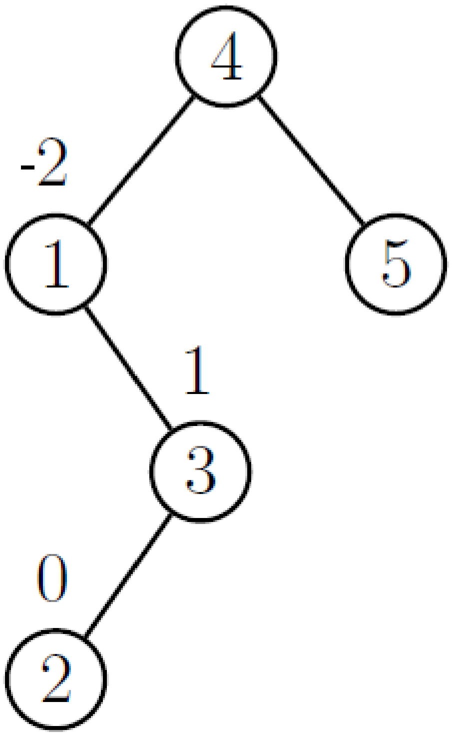
- It is possible that nodes at lower levels (closer to the root) have balance factors of ± 2.

- The diagram to the right uses **triangular** nodes to represent any sub-tree
  - The **black** node corresponds to the inserted element
  - To rebalance the tree we perform a **R-rotation**
  - When r **descends** and c takes its place, T_2 can **no longer** be the **right** child of c
  - However, all elements to the **left** of r must be **smaller**. Therefore, T_2 can **become** the **left** child of r



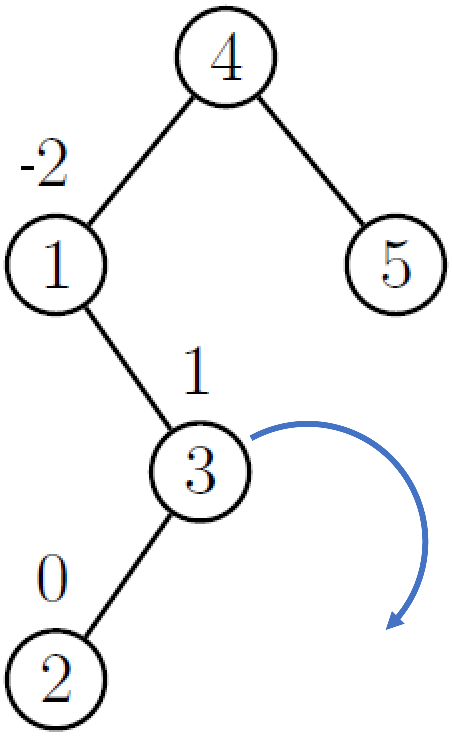
Double rotations

RL-Rotation



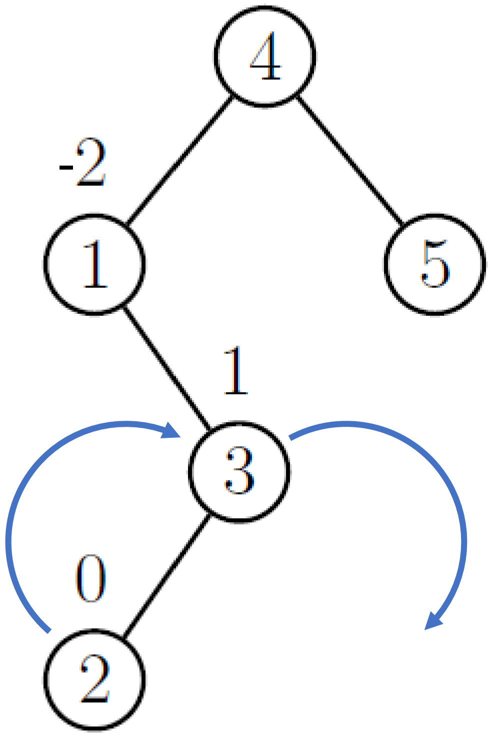
Double rotations

RL-Rotation



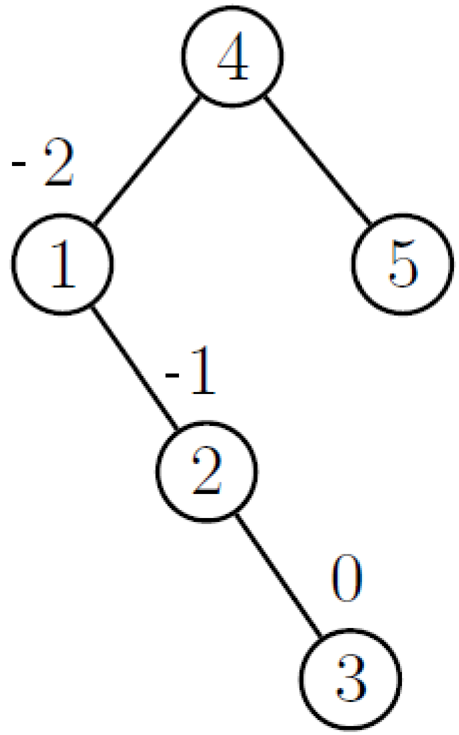
Double rotations

RL-Rotation



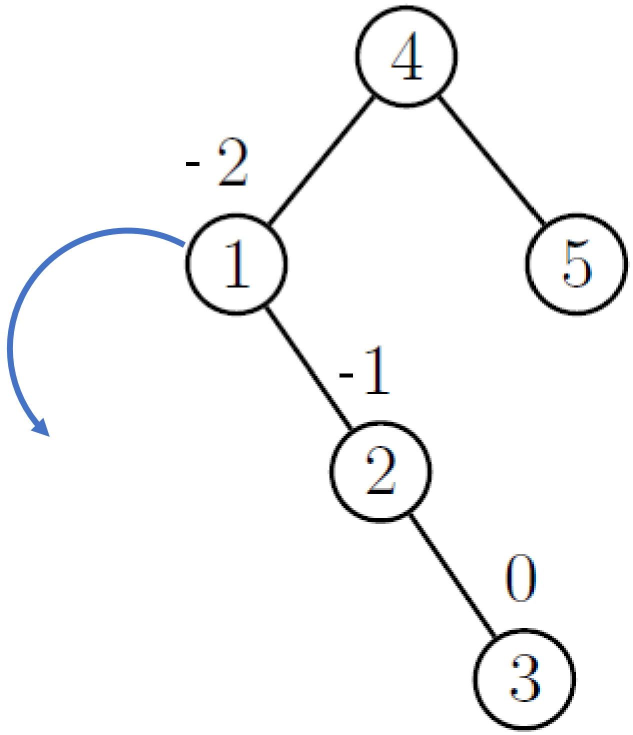
Double rotations

RL-Rotation



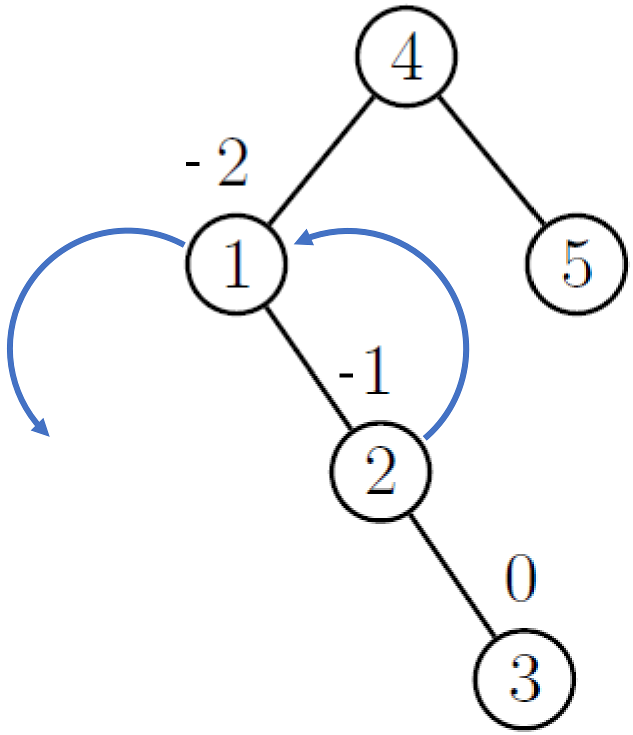
Double rotations

RL-Rotation



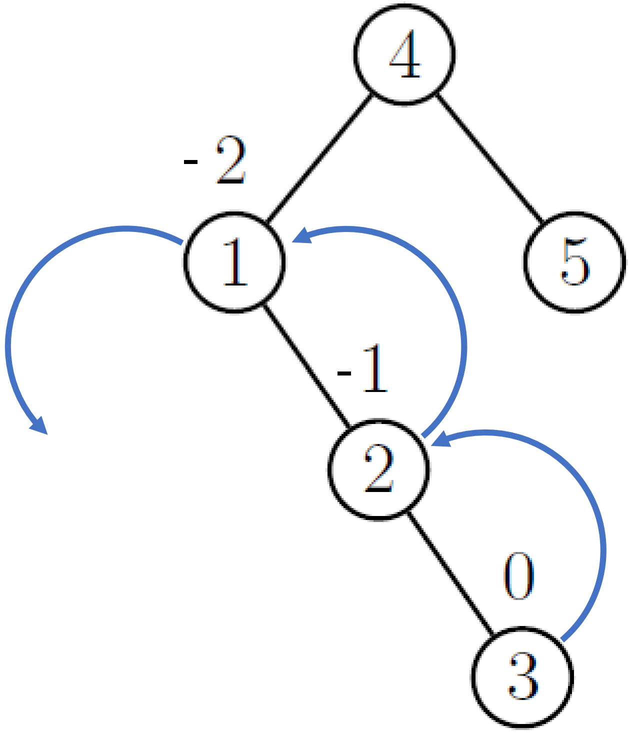
Double rotations

RL-Rotation



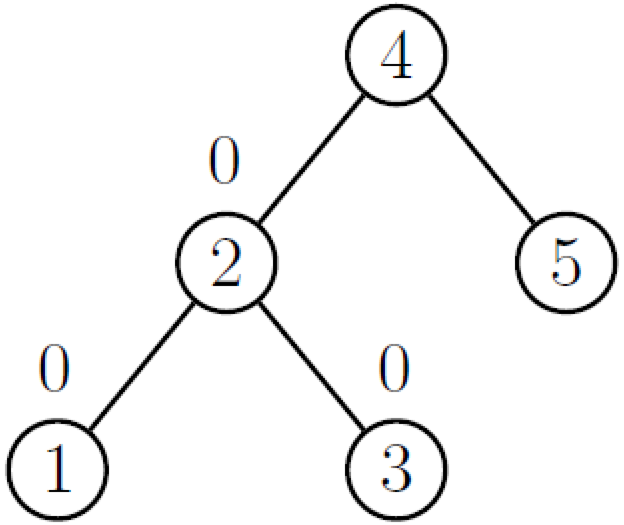
Double rotations

RL-Rotation



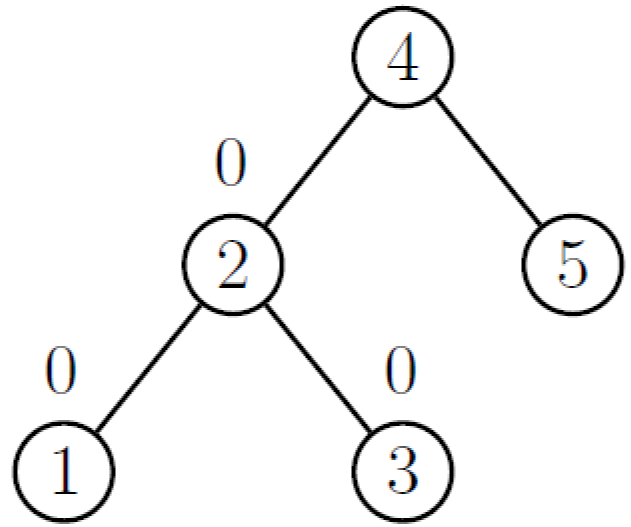
Double rotations

RL-Rotation

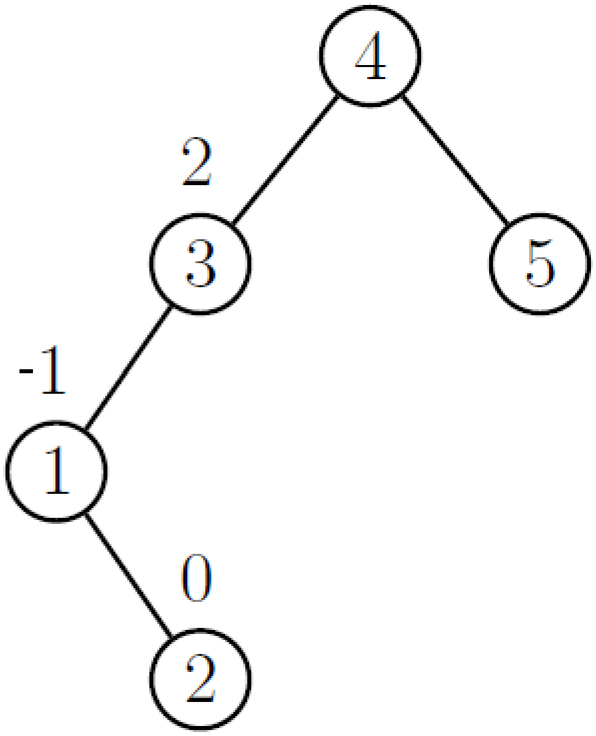


Double rotations

RL-Rotation

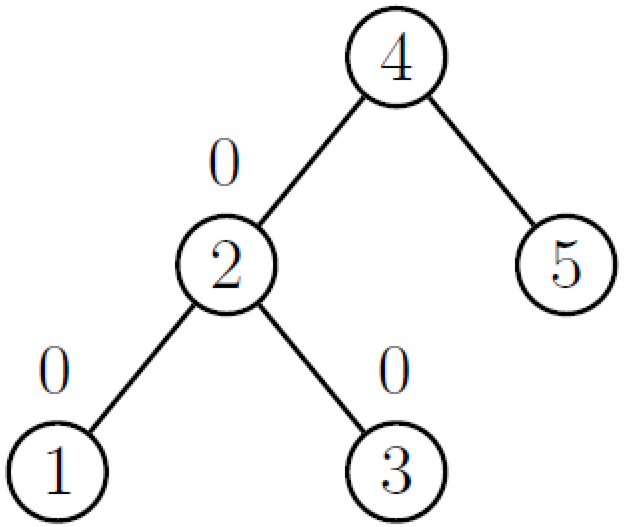


LR-Rotation

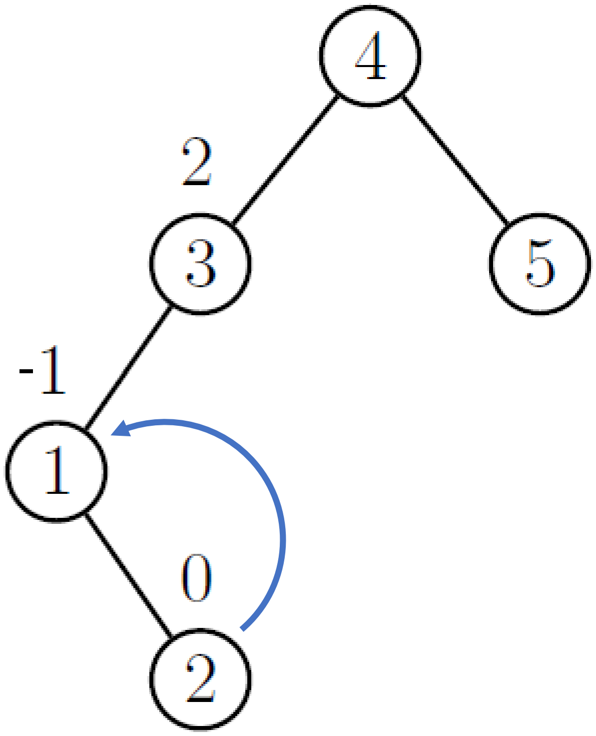


Double rotations

RL-Rotation

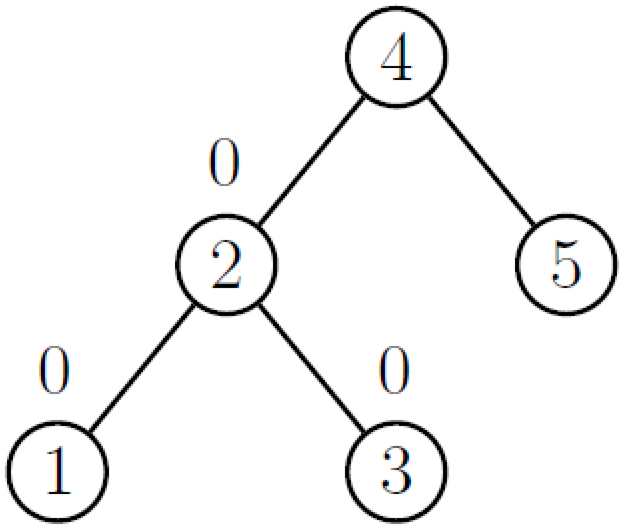


LR-Rotation

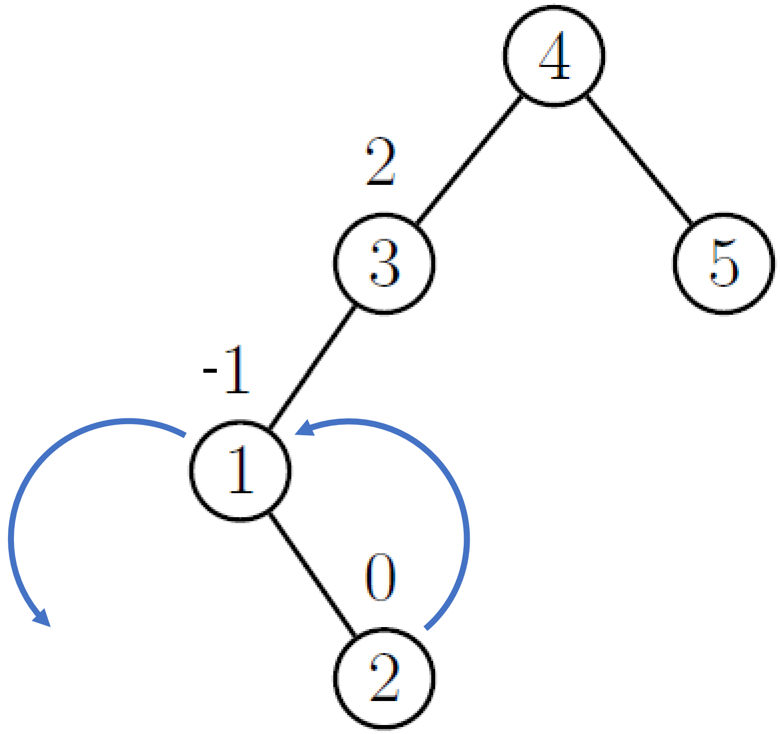


Double rotations

RL-Rotation

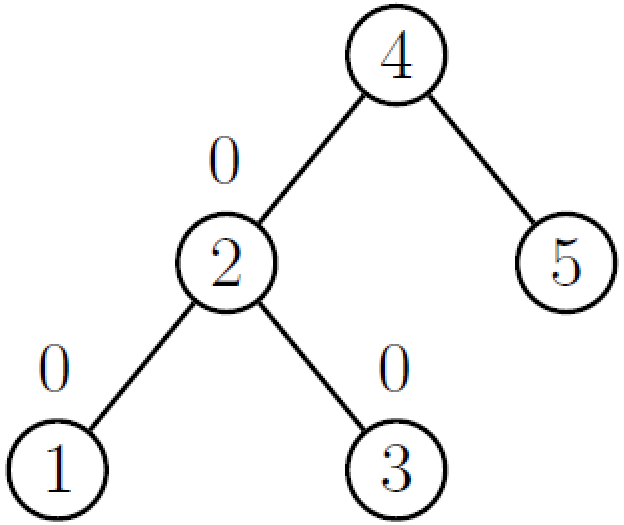


LR-Rotation

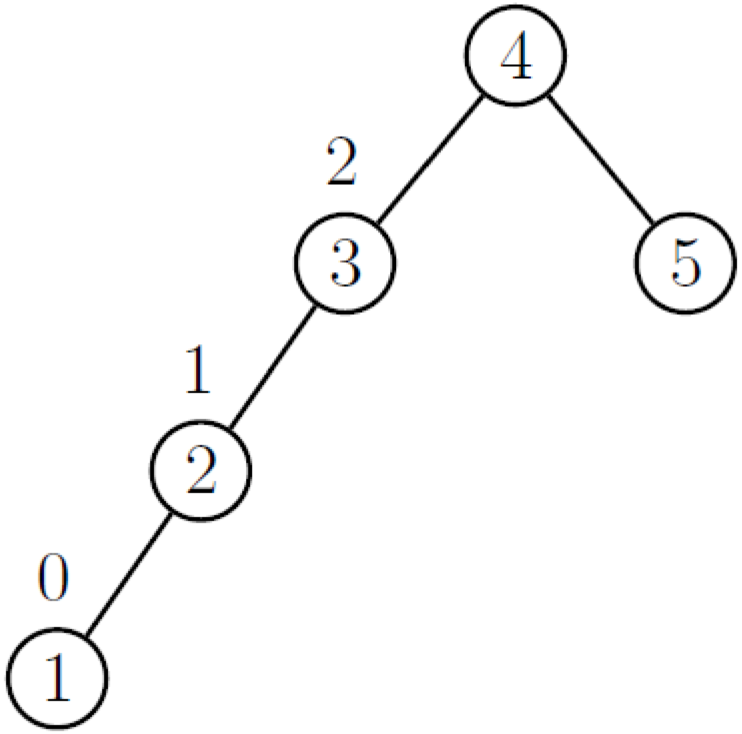


Double rotations

RL-Rotation

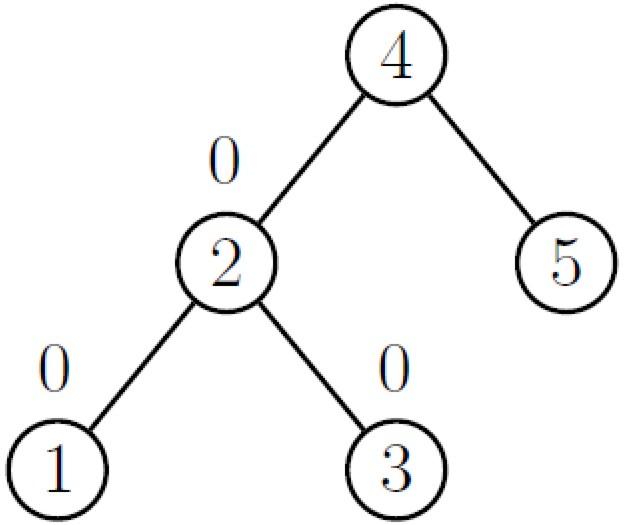


LR-Rotation

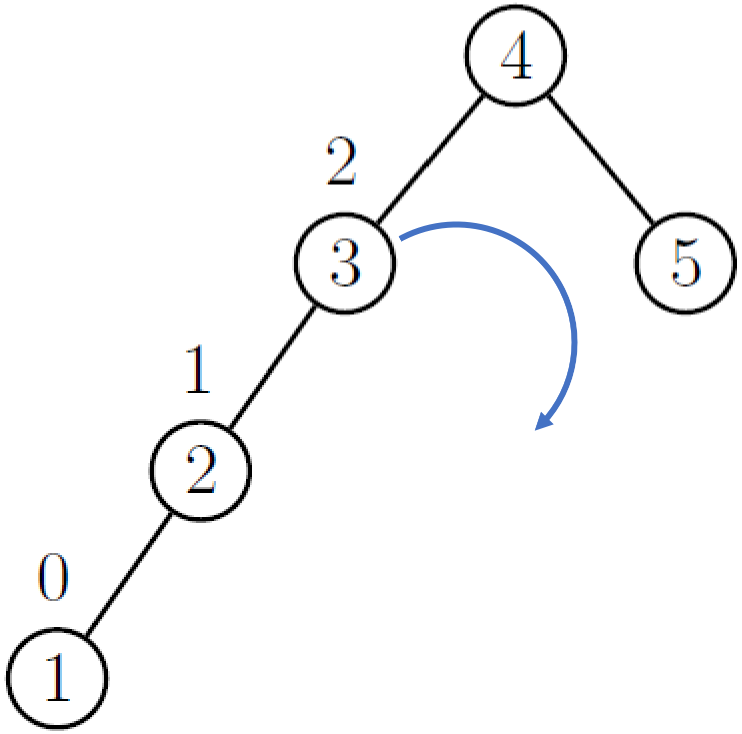


Double rotations

RL-Rotation

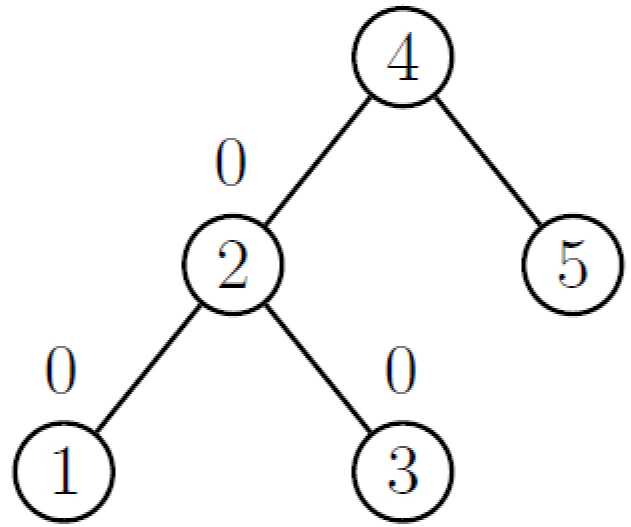


LR-Rotation

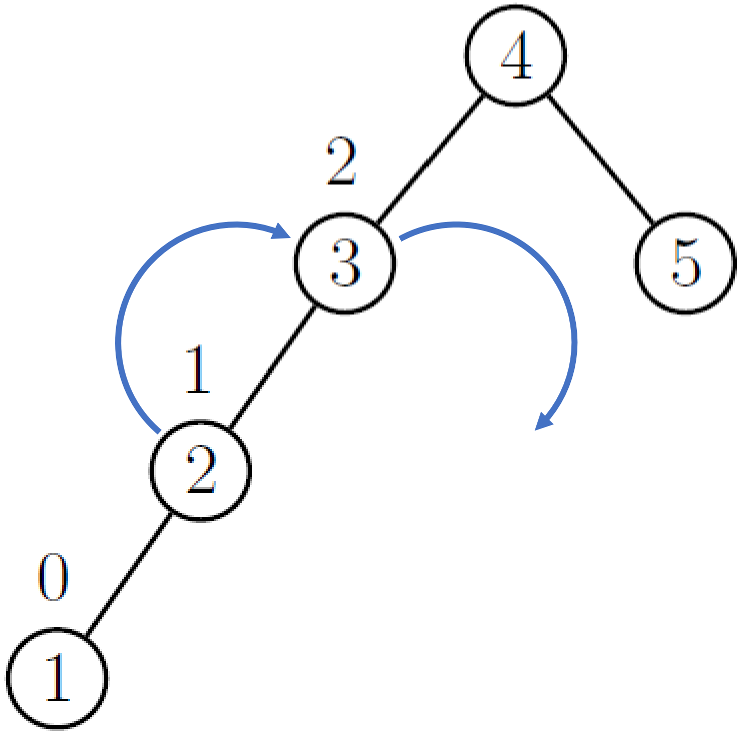


Double rotations

RL-Rotation

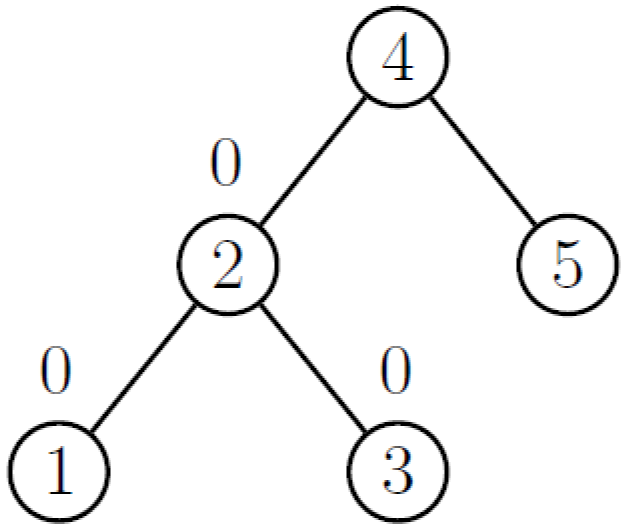


LR-Rotation

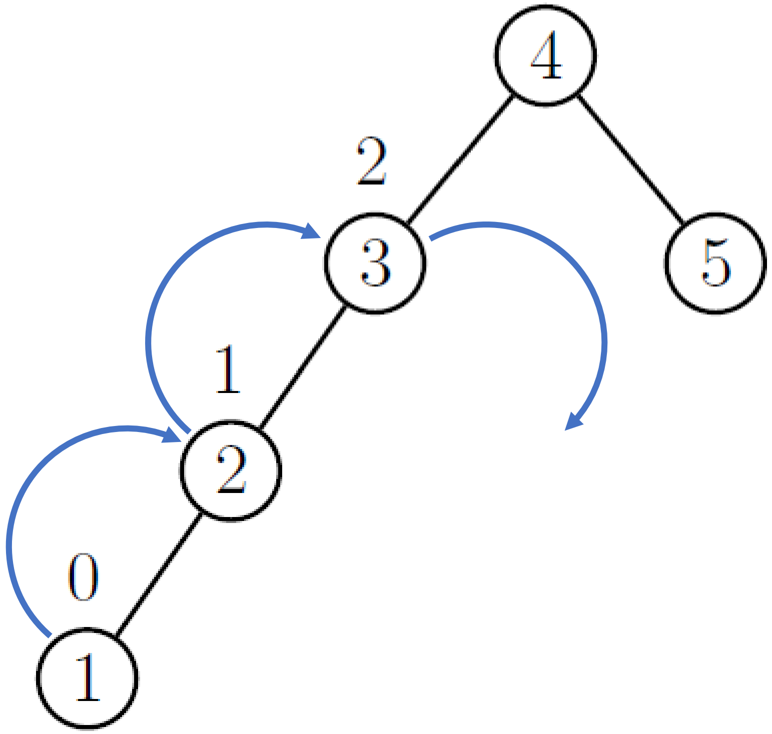


Double rotations

RL-Rotation

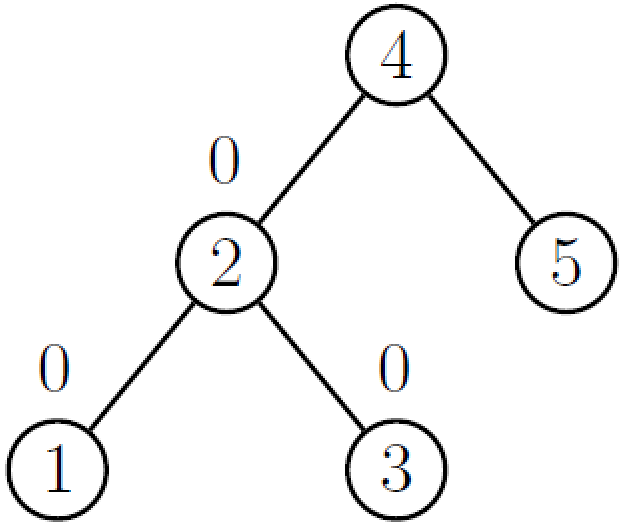


LR-Rotation

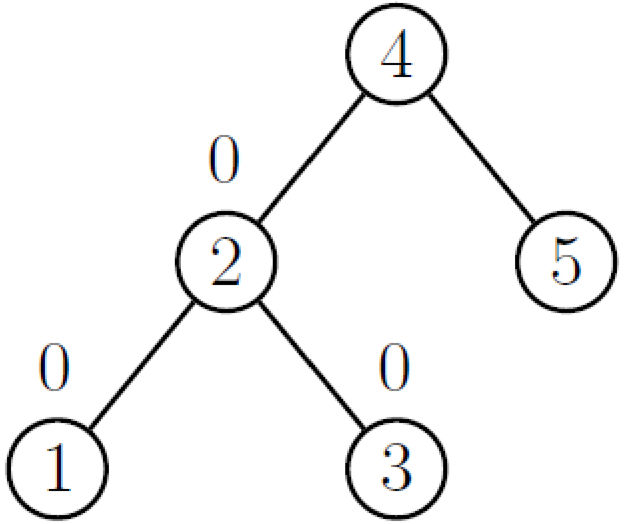


Double rotations

RL-Rotation

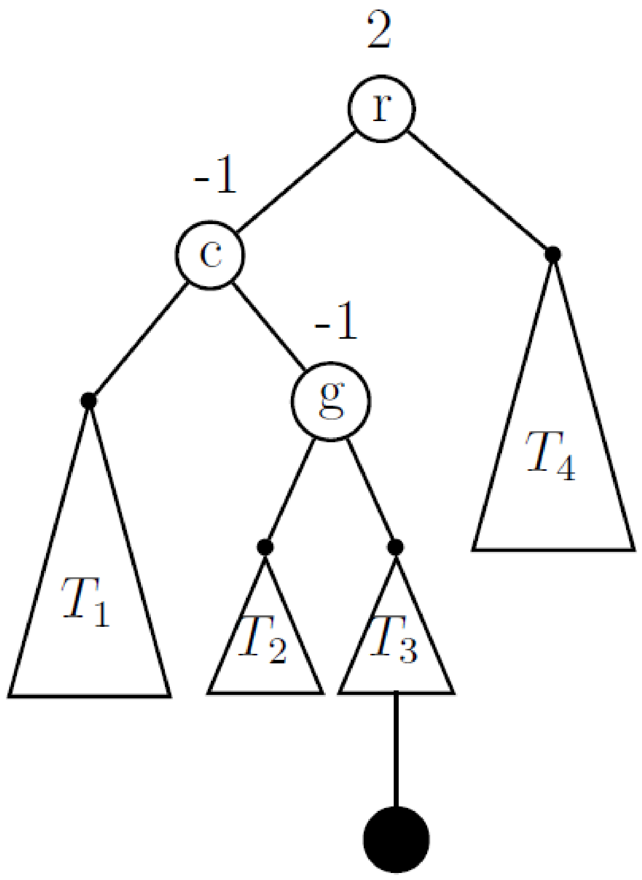


LR-Rotation



General double rotations

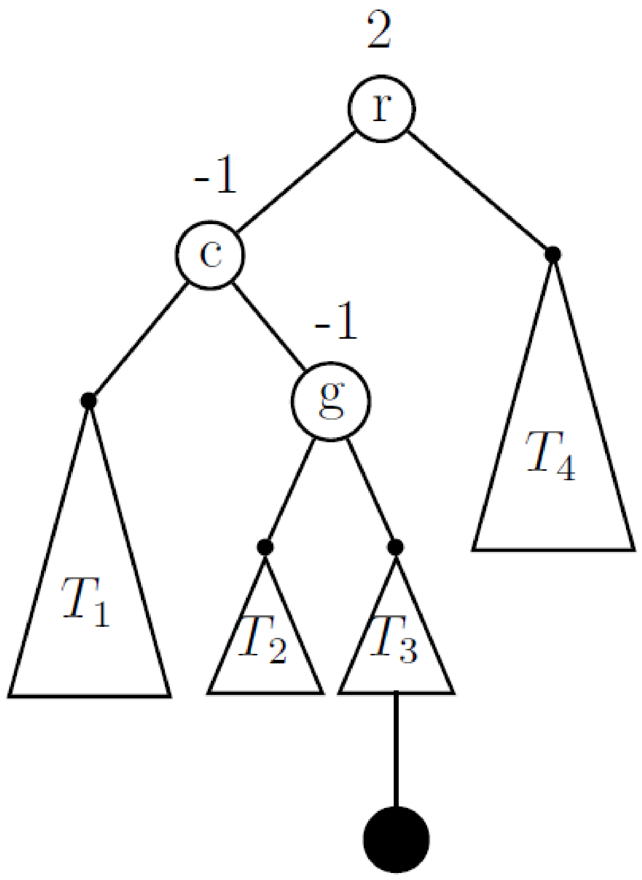
- Similarly to the general single rotation, the **black** node corresponds to a single inserted element



General double rotations

- Similarly to the general single rotation, the **black** node corresponds to a single inserted element

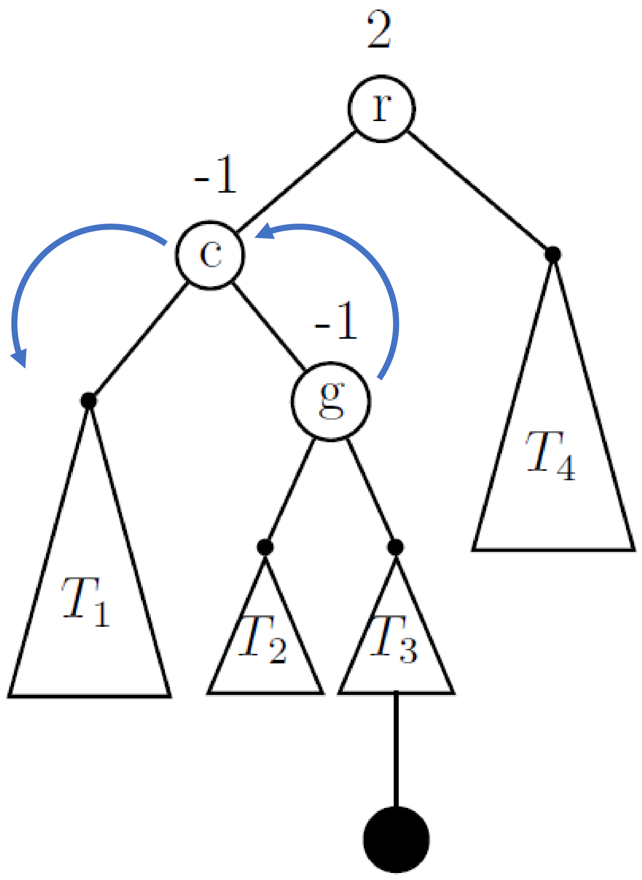
- To rebalance the tree we need to do a **LR-rotation**



General double rotations

- Similarly to the general single rotation, the **black** node corresponds to a single inserted element

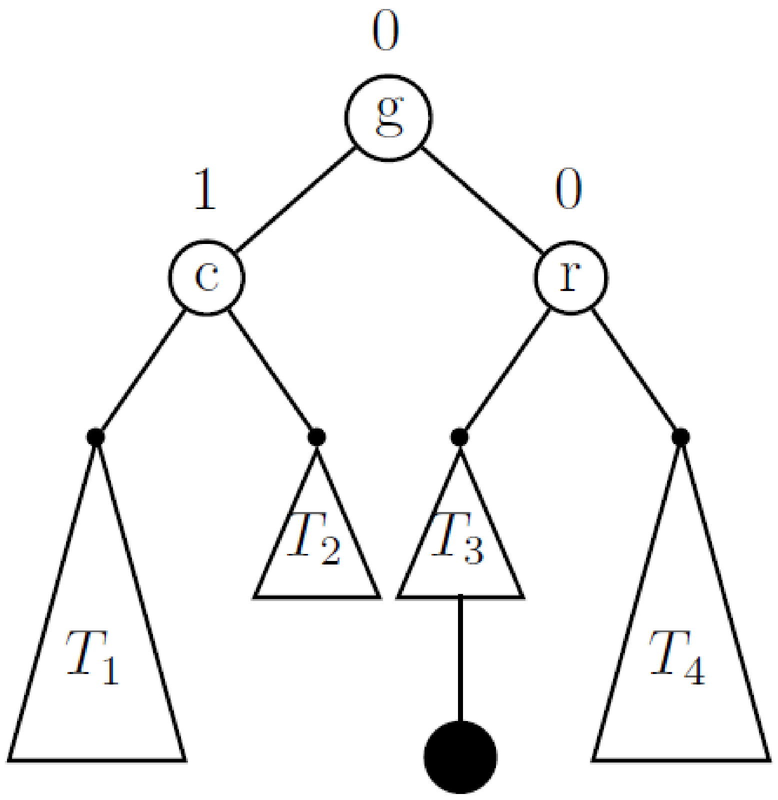
- To rebalance the tree we need to do a **LR-rotation**
  - On the **L-rotation**, T_2 cannot be any longer the **left** child of g because it must be c. However, T_2 can be the **right** child of c



General double rotations

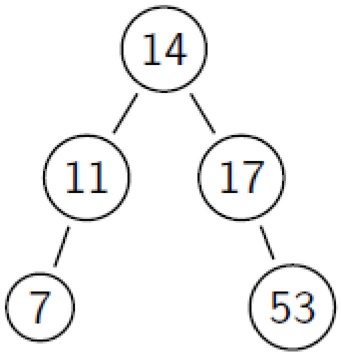
- Similarly to the general single rotation, the **black** node corresponds to a single inserted element

- To rebalance the tree we need to do a **LR-rotation**
  - On the **L-rotation**, T_2 cannot be any longer the **left** child of g because it must be c. However, T_2 can be the **right** child of c
  - On the **R-rotation**, T_3 cannot be any longer the **right** child of g, because it must be r. However, T_3 can be the **left** child of r



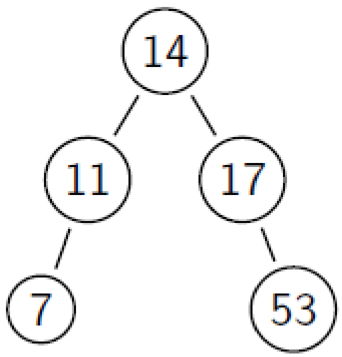
Example

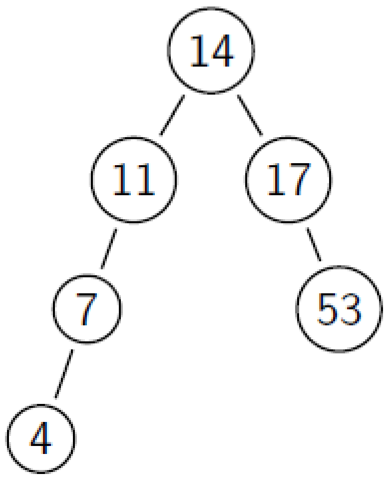
- On the tree below, insert 4



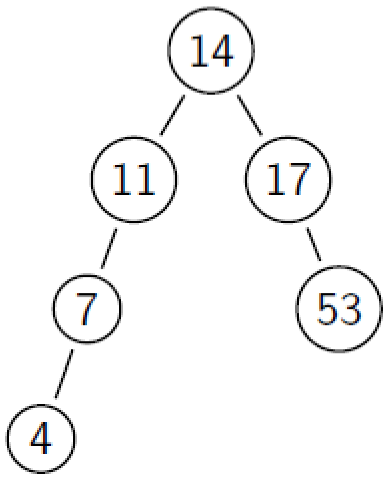
Example

- On the tree below, insert 4





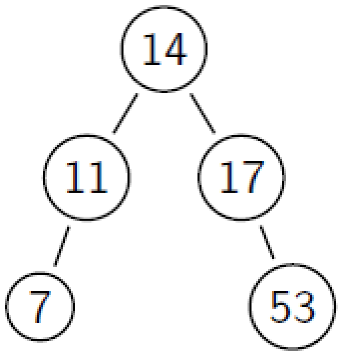
Sub-tree heights

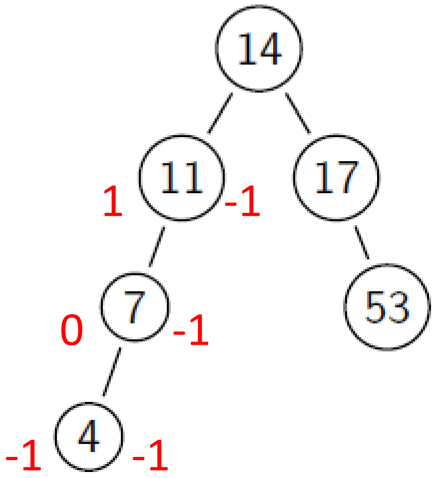


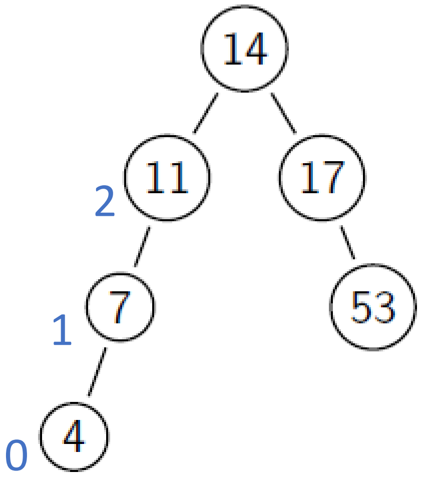
Balance Factors

Example

- On the tree below, insert 4





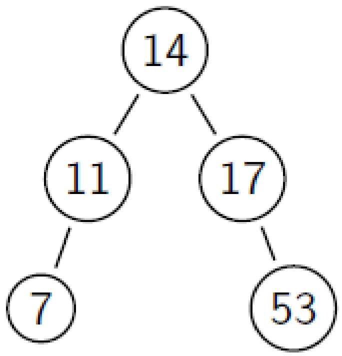


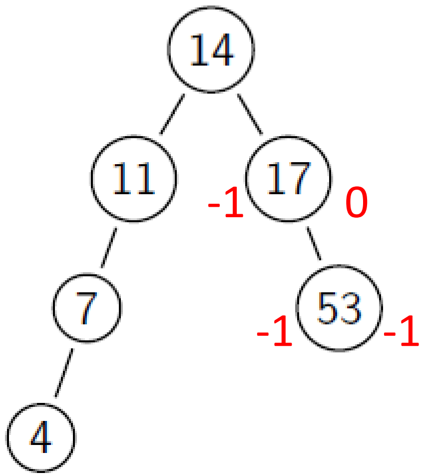
Sub-tree heights

Balance Factors

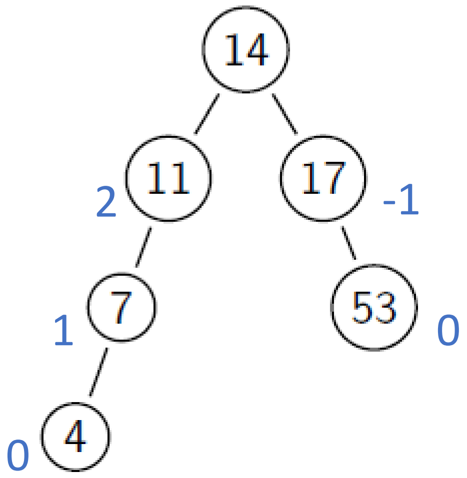
Example

- On the tree below, insert 4





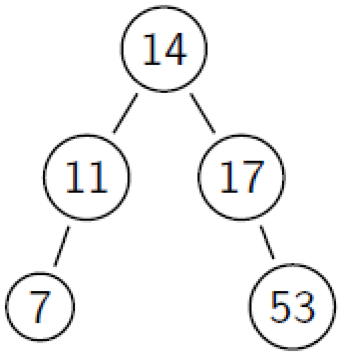
Sub-tree heights

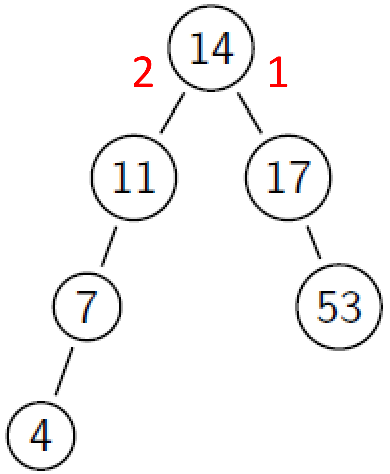


Balance Factors

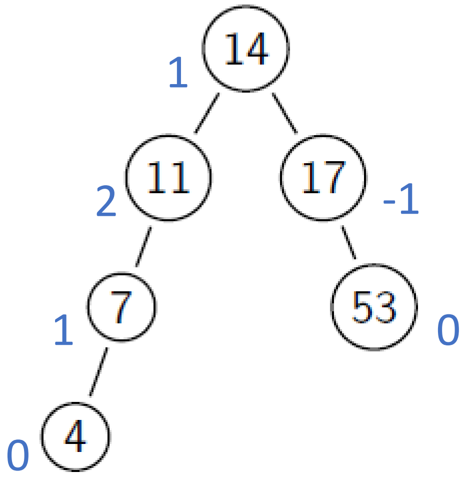
Example

- On the tree below, insert 4





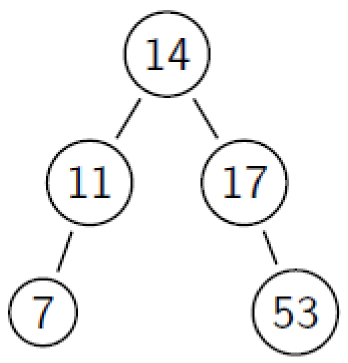
Sub-tree heights

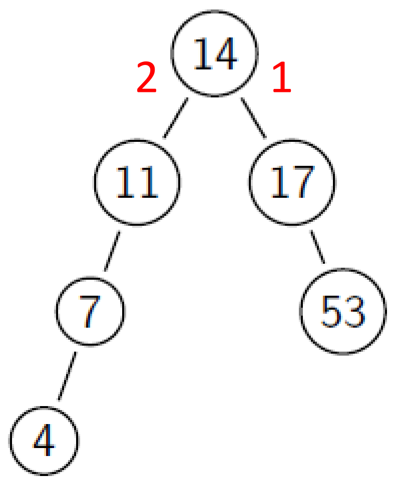


Balance Factors

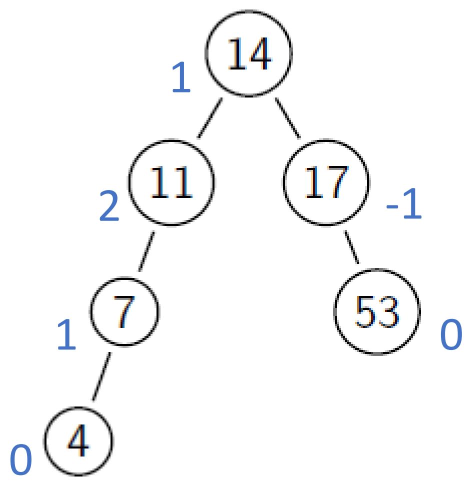
Example

- On the tree below, insert 4

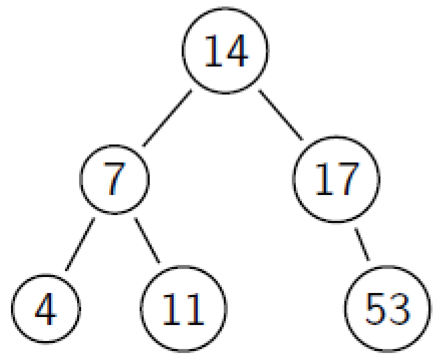




Sub-tree heights



Balance Factors



R-Rotation

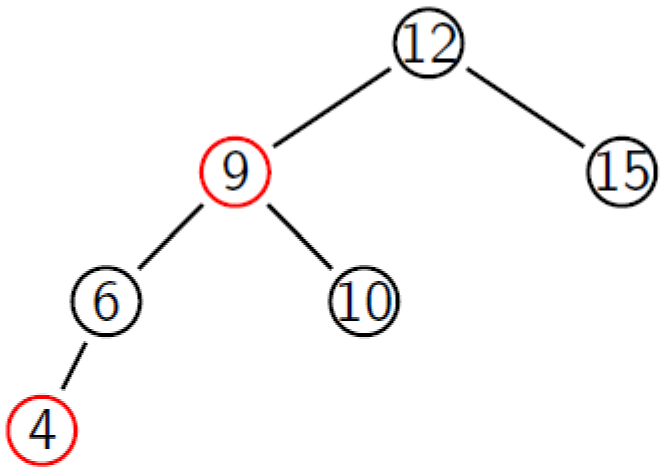
Properties of AVL Trees

- The notion of **balance** implied by the AVL condition guarantees that the depth of an AVL tree with n nodes is $\Theta(\log n)$
  - For random data, the depth is very close to the optimal $\log_2 n$
  - 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
  - No red node has a red child.
  - 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).

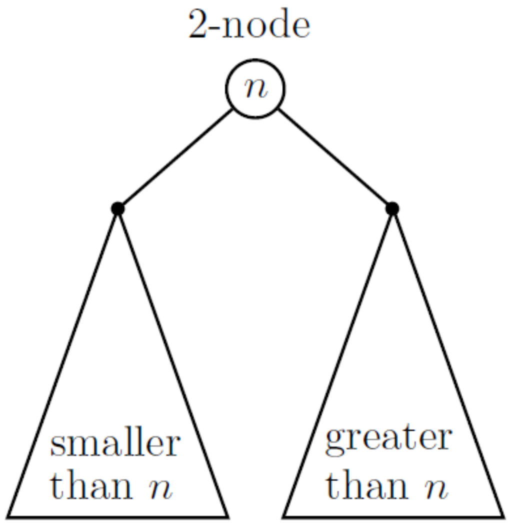
- A **splay tree** is a BST which is not only self-adjusting, but also **adaptive**. Frequently accessed items are brought closer to the root, so their access becomes cheaper.

2–3 Trees

- 2–3 trees and 2–3–4 trees **allow more than one item** to be stored in a node.

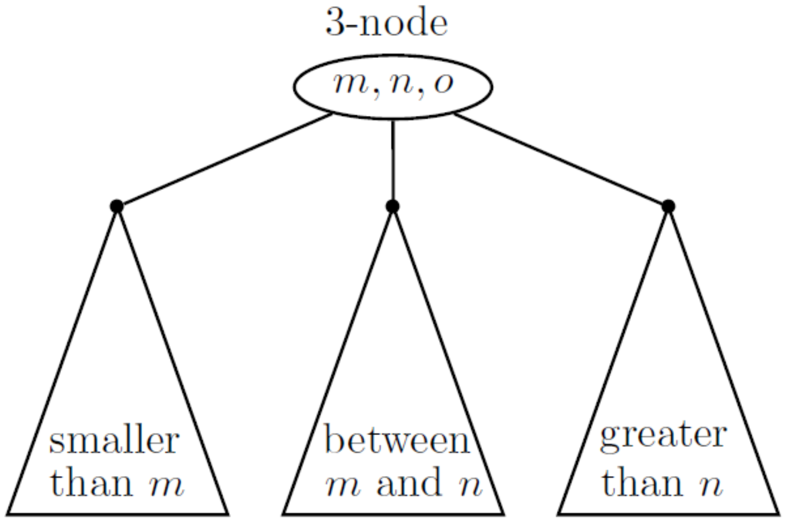
2–3 Trees

- 2–3 trees and 2–3–4 trees **allow more than one item** to be stored in a node.
  - A node with a **single** item has up to **two children**



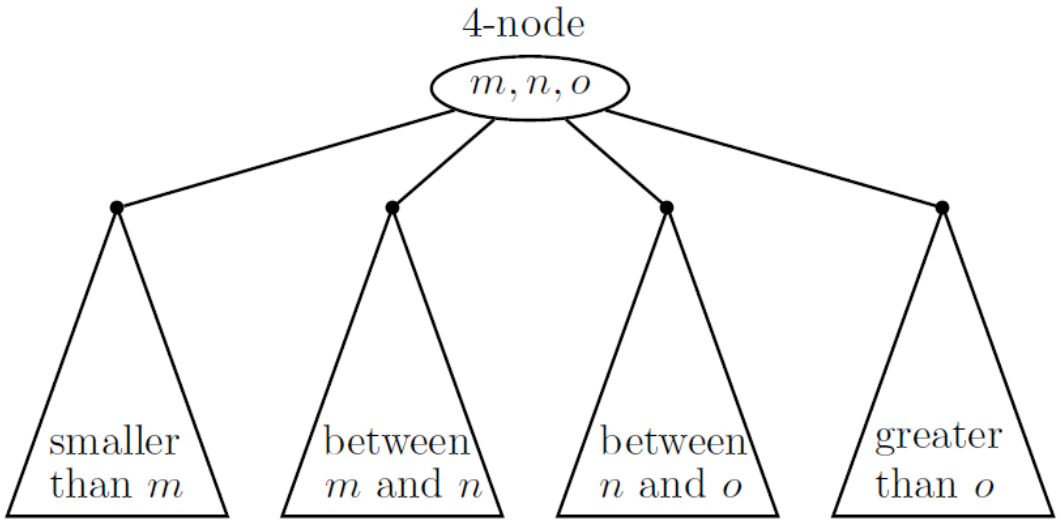
2–3 Trees

- 2–3 trees and 2–3–4 trees **allow more than one item** to be stored in a node.
  - A node with a **single** item has up to **two children**
  - A node with **two** items (**3-node**) has up to **three children**



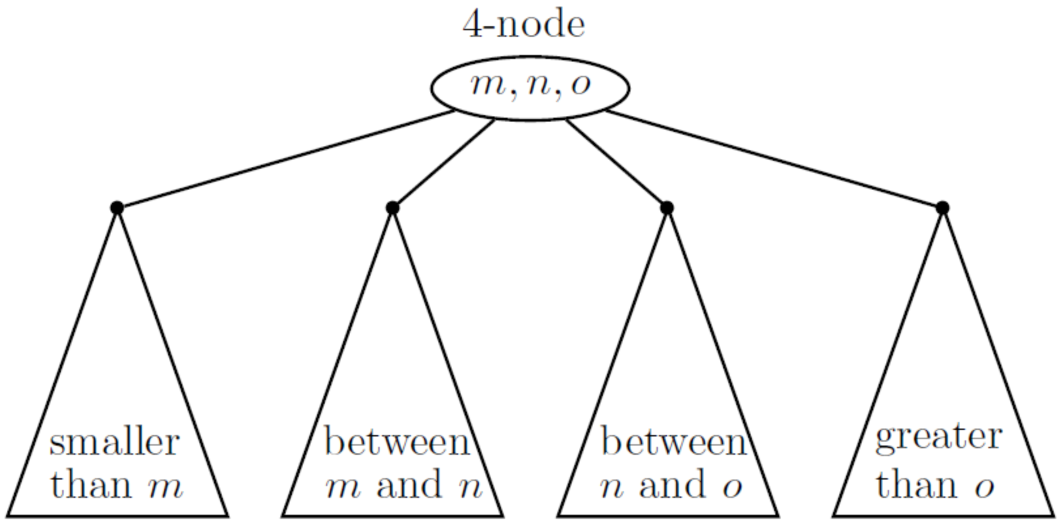
2–3 Trees

- 2–3 trees and 2–3–4 trees **allow more than one item** to be stored in a node.
  - A node with a **single** item has up to **two children**
  - A node with **two** items (**3-node**) has up to **three children**
  - And for 2–3–4 trees, a with **three** items (**4-node**) has up to **four children**



2–3 Trees

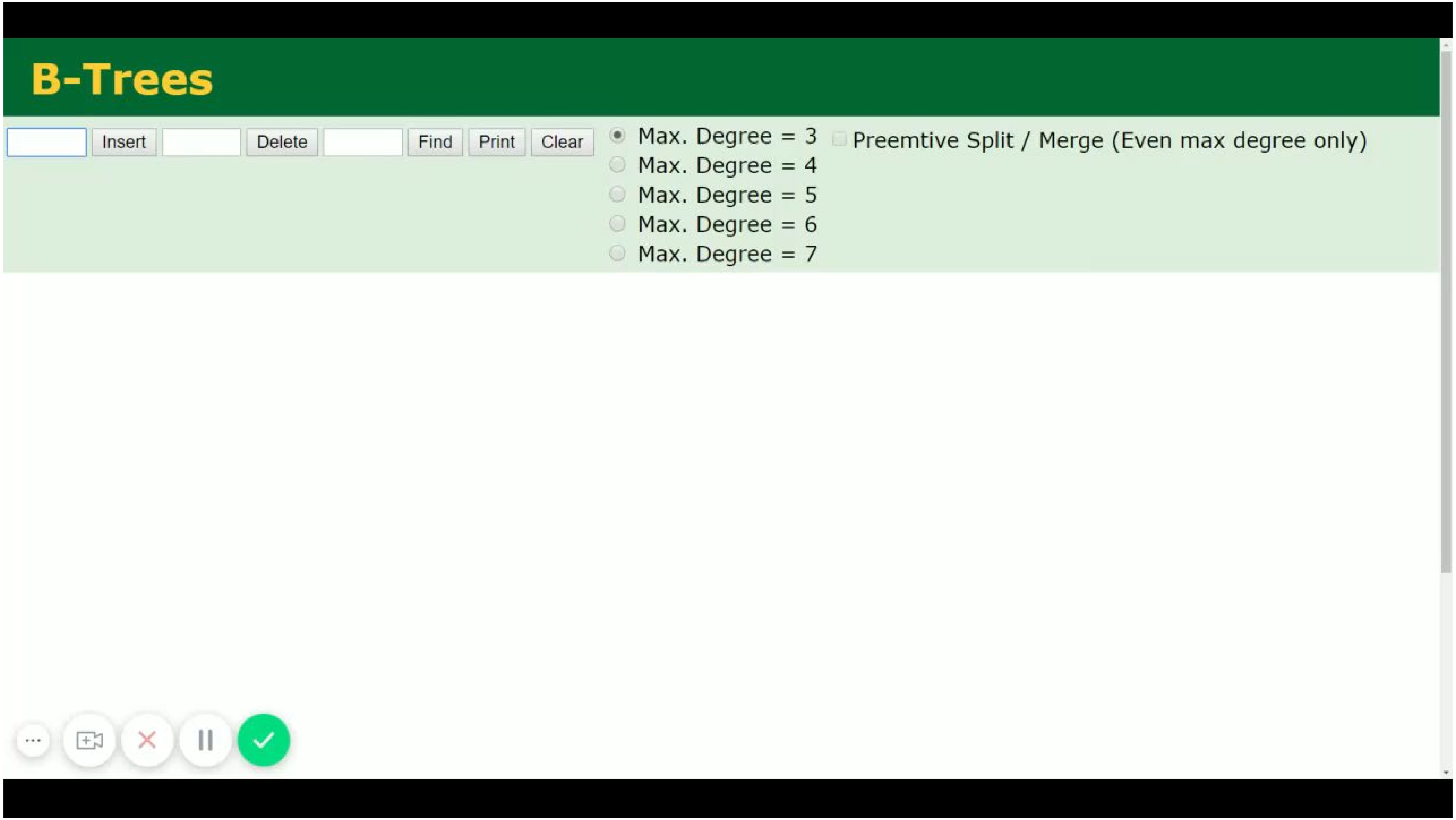
- 2–3 trees and 2–3–4 trees **allow more than one item** to be stored in a node.
  - A node with a **single** item has up to **two children**
  - A node with **two** items (**3-node**) has up to **three children**
  - And for 2–3–4 trees, a with **three** items (**4-node**) has up to **four children**
- This allows for a simple way of keeping search trees **perfectly** balanced



How to insert an element in a 2–3 Tree?

- As with other trees, a new element k is inserted by searching for it
  - However, once a **leaf node** is found, we insert k in that position
  - If the **leaf** was originally a **2-node** (one element), nothing else is done
  - If the **leaf** was originally a **3-node** (two elements), the now three elements $\{k_1, k_2, k_3\}$ are sorted
  - We **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
  - If the parent node **overflows** due to the promotion, it is split in the same way
  - The **height** of the tree only changes when the **root overflows**

- Let's observe this in detail by building a tree containing {9, 5, 8, 3, 2, 4, 7}


B-Trees
Insert
Delete
Find
Print
Clear
Max. Degree = 3
Max. Degree = 4
Max. Degree = 5
Max. Degree = 6
Max. Degree = 7
Preemtive Split / Merge (Even max degree only)

Properties of the 2–3 Tree

- **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**:

$$n = 2 + 2\times3 + 2\times3^2 + \ldots + 2\times3^h = 3^{h+1} - 1$$

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

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

- This implies that **search, insertion and deletion** are all $\Theta(\log n)$ in the **worst and average** cases.

2–3 Trees

- Let’s take a few **minutes** to build the 2–3 tree that results from inserting {C, O, M, P, U, T, I, N, G}, in the given order, into an initially empty tree.


B-Trees
Insert
Delete
Find
Print
Clear
Max. Degree = 3
Max. Degree = 4
Max. Degree = 5
Max. Degree = 6
Max. Degree = 7
Preemtive Split / Merge (Even max degree only)

Next lecture

- Input enhancement
  - Distribution counting (Levitin Section 7.1)
  - Horspool’s string search algorithm (Levitin Section 7.2)
  - Knuth-Morris-Prat string search algorithm (http://jeffe.cs.illinois.edu/teaching/algorithms/notes/07-strings.pdf)