

# Lesson 3: Trees

---

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS

DR. ANDREY TIMOFEYEV

A solid blue horizontal bar at the bottom of the slide.

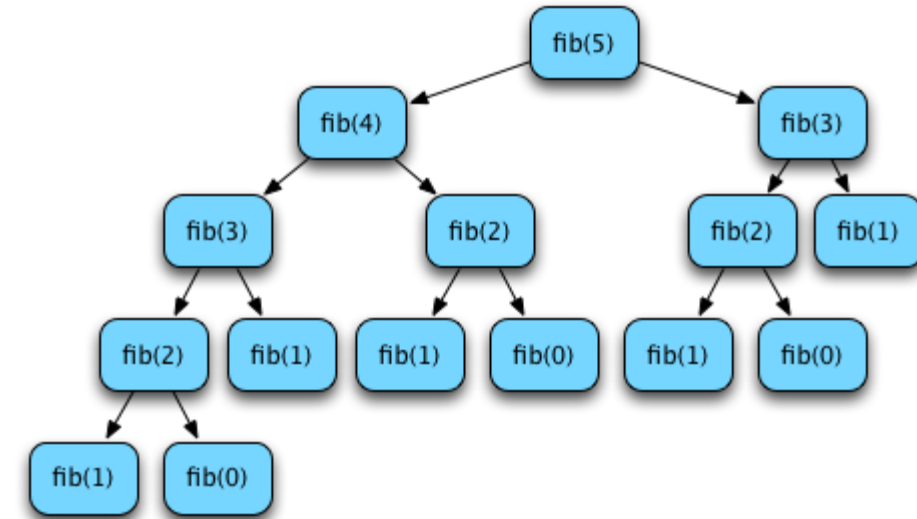
# OUTLINE

---

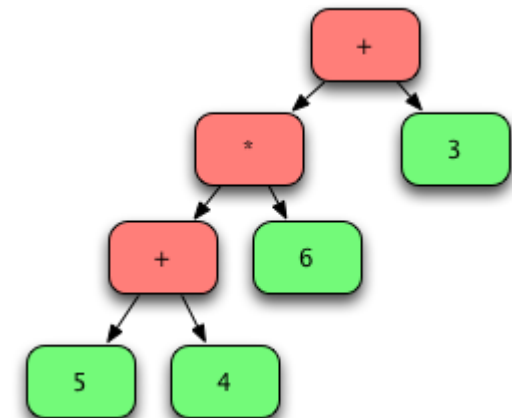
- Introduction.
- Binary search trees.
- Search space.
- AVL trees.
- Splay trees.

# INTRODUCTION

- **Trees** data structure.
  - **Non-sequential.**
  - Data stored in **nodes** with **links** to **two** or **more** other nodes.
    - Root, parents, children, siblings, subtrees.
  - **Abstract Syntax Trees (ASTs)** used by programming languages to convert source code to a tree-like structure for evaluation.
- **Binary search trees.**
  - Each node has **up to two child nodes**.
  - Values in **left subtree** < **root** value < values in **right subtree**.
  - **Left & right** subtrees are **binary search trees**.
  - **Insertion** is  $O(n \log n)$  but takes **more** space than lists.
    - If items sorted, then insertion is  $O(n^2)$ .
      - Tree -> stick -> linked list.
  - **Improved performance:** AVL trees & splay trees.



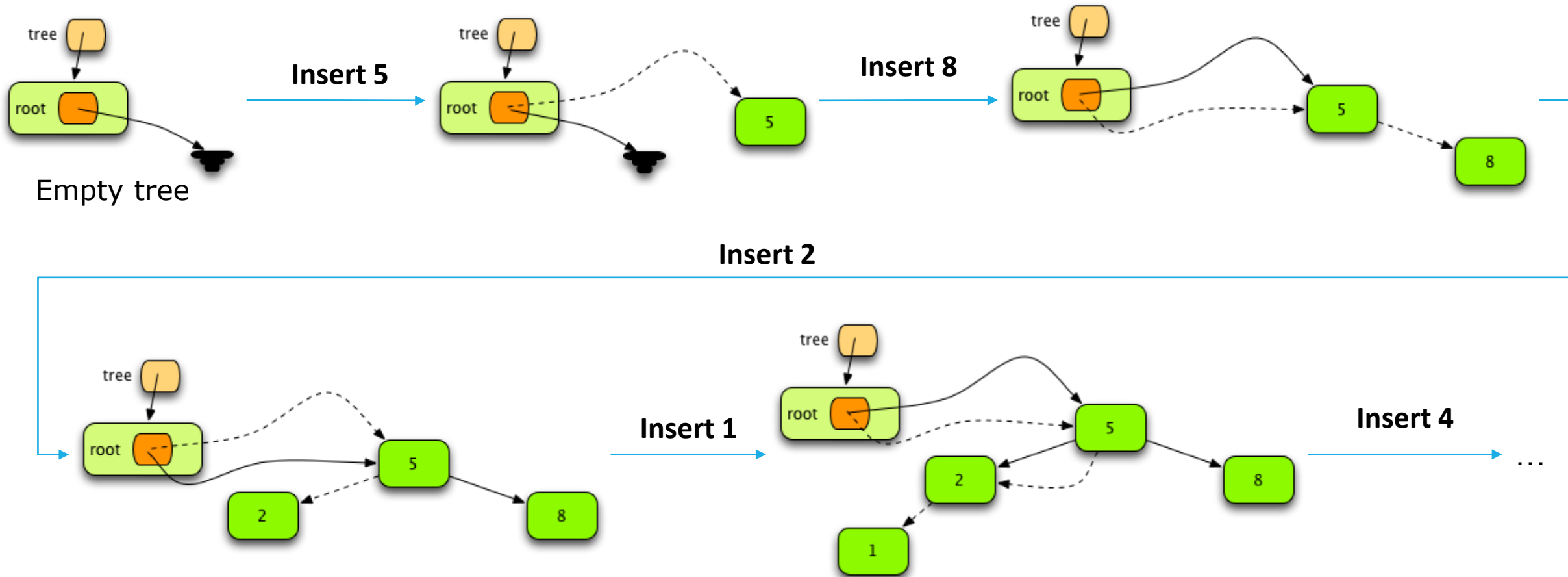
Fib(5) call Tree



Abstract syntax tree for  $(5 + 4) * 6 + 3$

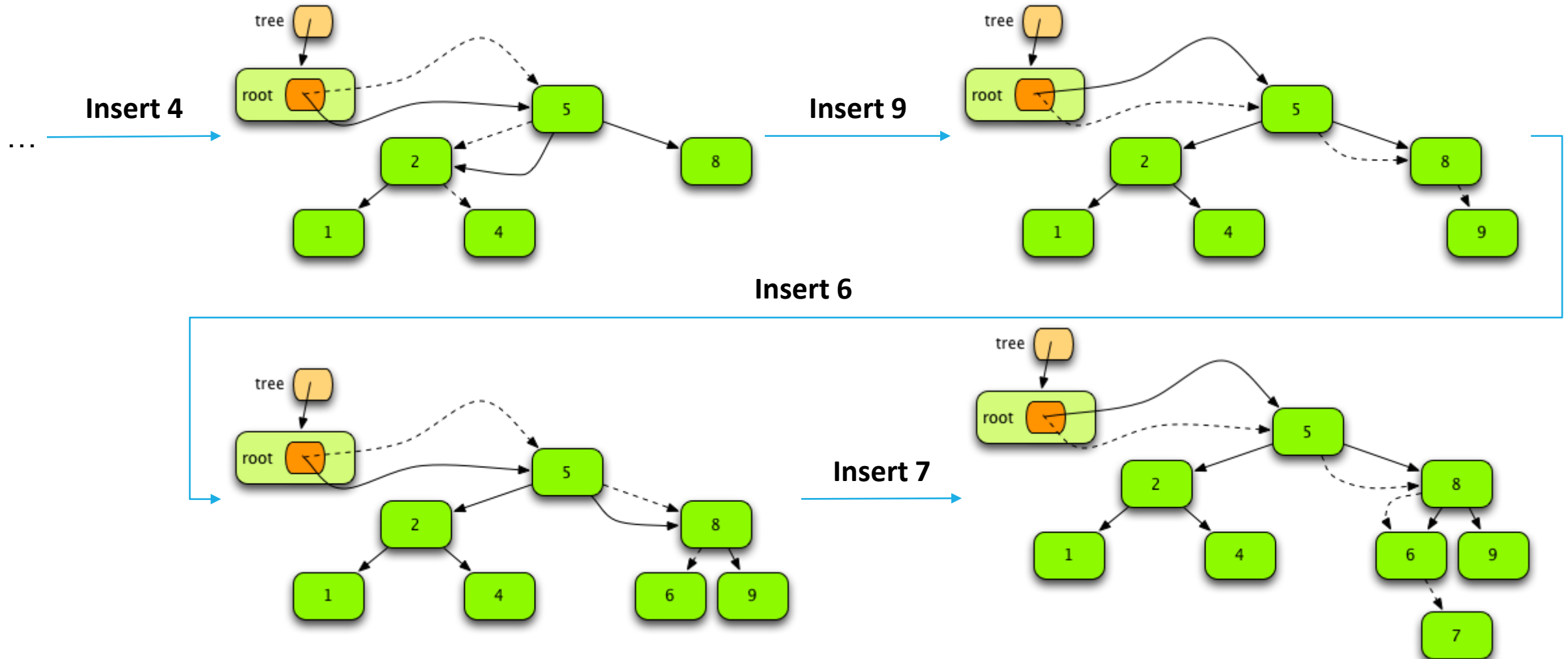
# BINARY SEARCH TREES: INSERTION (1)

- Inserting 5, 8, 2, 1, 4, 9, 6, 7 into binary search tree:



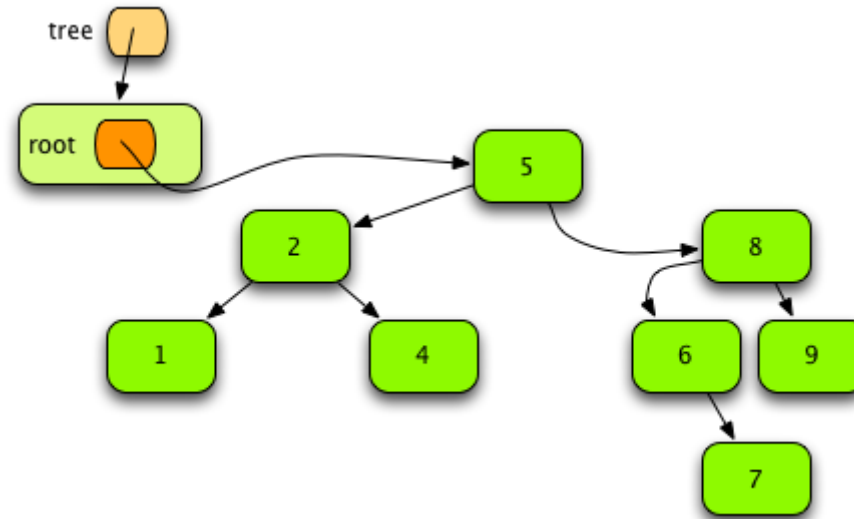
# BINARY SEARCH TREES: INSERTION (2)

- Inserting 5, 8, 2, 1, 4, 9, 6, 7 into binary search tree (cont.):



# BINARY SEARCH TREES: INSERTION (3)

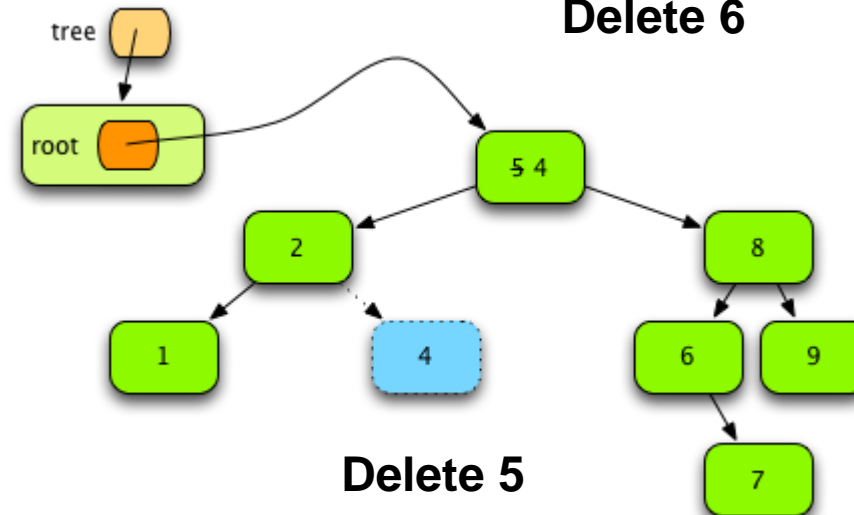
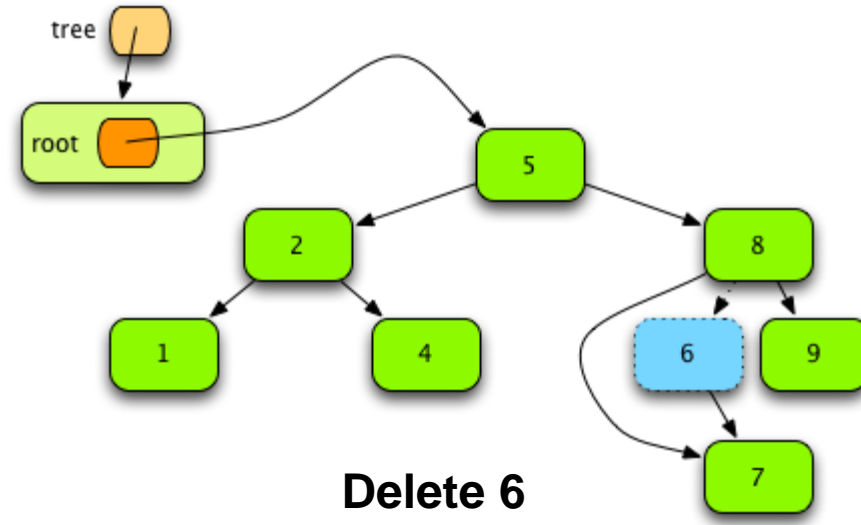
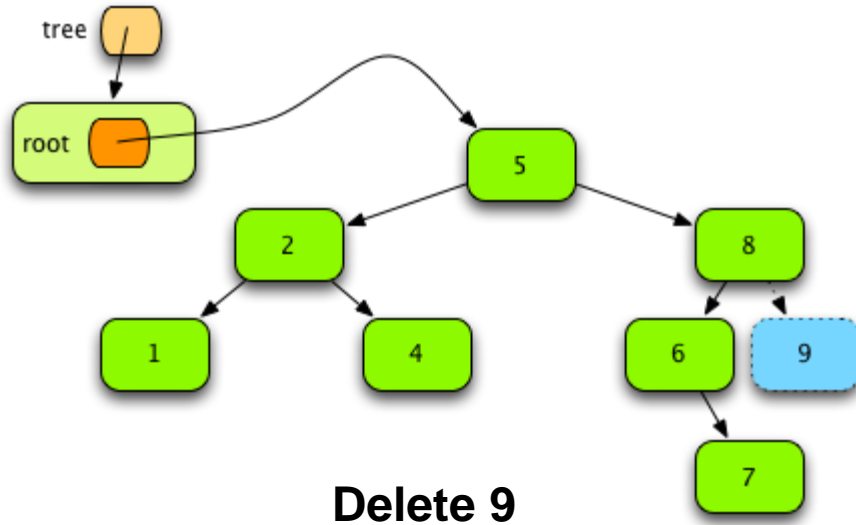
- Inserting 5, 8, 2, 1, 4, 9, 6, 7 into binary search tree (cont.):



Final tree

# BINARY SEARCH TREES: DELETION

- Deleting 9, 6, 5 from binary search tree:



# SEARCH SPACES

- Normally, **problems** consist of many different **states**.
  - **Goal** (*end state*) = finding a **solution** to the problem.
  - **Collection** of all problem **states** = **search space**.
  - A **solution** can be found by applying a **depth first search** on the problem search space.
- **Depth first search** of a problem space:
  - Check for a **solution** by making a **guess**.
  - If **guess** does **not lead** to a solution, then **backtrack**.
    - **Backtracking** – undoing unsuccessful guess and trying next guess.
  - Keep **trying** guesses and **backtracking** until solution is **found**.



# BALANCED BINARY SEARCH TREES

---

- **Unbalanced binary search trees.**

- Problem: BST operations computational complexity grows once the tree becomes unbalanced.
- Solution: **Balanced** BST structure with guaranteed  $\log n$  height,  $\Theta(n \log n)$  to build a tree,  $\Theta(n)$  for in-order traversal.
  - AVL trees & splay trees.

- **Balanced binary search trees** are used when many **insert/delete/lookup** operations & many **iterations** over items in **ascending/descending** order are required.

# AVL TREES

- **AVL trees.**

- **Binary search trees** with an additional information to maintain the **balance**.
- **Height** is  $\log n$ .
- **Lookup/insert/delete** operations are  $\Theta(\log n)$ .
- Tree is **built** in  $\Theta(n \log n)$ .
- **In-order traversing** (yields sorted items) is  $\Theta(n)$ .

- **AVL tree definitions:**

- Height = 1 + the maximum height of subtrees.
  - Height of leaf node = 1.
- Balance = (height of right subtree) – (height of left subtree).
- AVL tree = balance of every node must be in  $[-1, 0, 1]$ .

- **Balance** is maintained using **height** or **balance** of **each node** in the tree.

# AVL TREES: INSERTION (1)

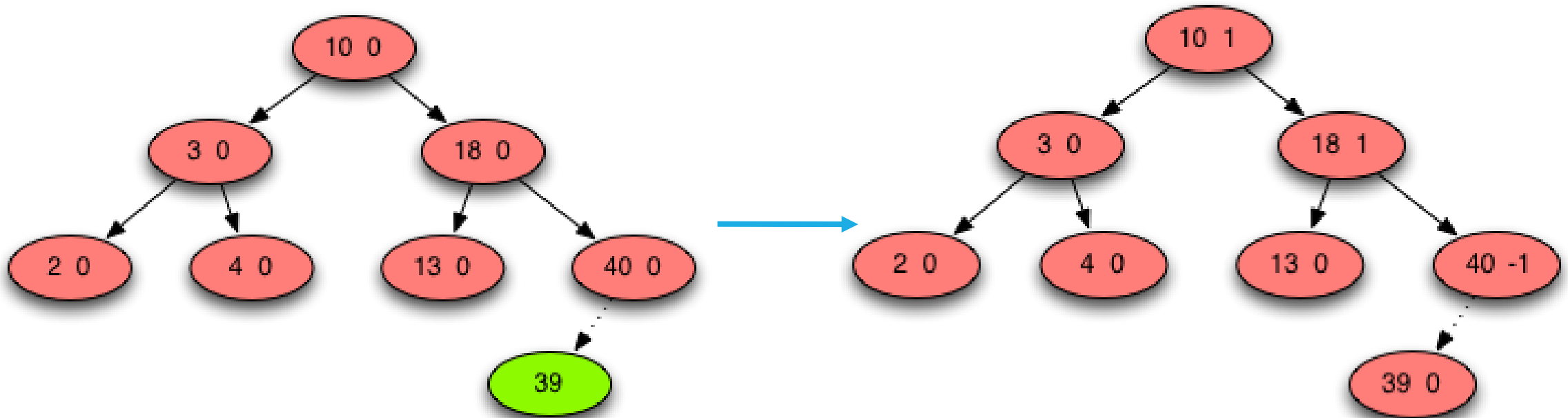
- **Inserting into AVL trees:**

- Initially, **same process** as for BST – **start** from the **root**, **find** correct **position**.
  - In addition, **keep track** of the nodes along the path.
  - Traversed nodes are pushed onto the stack (**path stack**).
- Once node is inserted, nodes **balances** are **adjusted**.
  - **Pop** nodes off the **path stack** and **adjust balances**.
  - If found node with **balance  $\neq 0$**  before adjusting – **pivot node**.
- **Based on the pivot node and location of inserted node, three cases must be considered:**
  - **Case 1:** No pivot found.
  - **Case 2:** Pivot found; no rotation needed.
  - **Case 3:** Pivot found; rotation needed.
    - Single rotation or double rotation.

# AVL TREES: INSERTION (2)

- **Case 1: No pivot found.**

- Balance of each node along the path == 0.
- Adjust balance of each node in path stack.

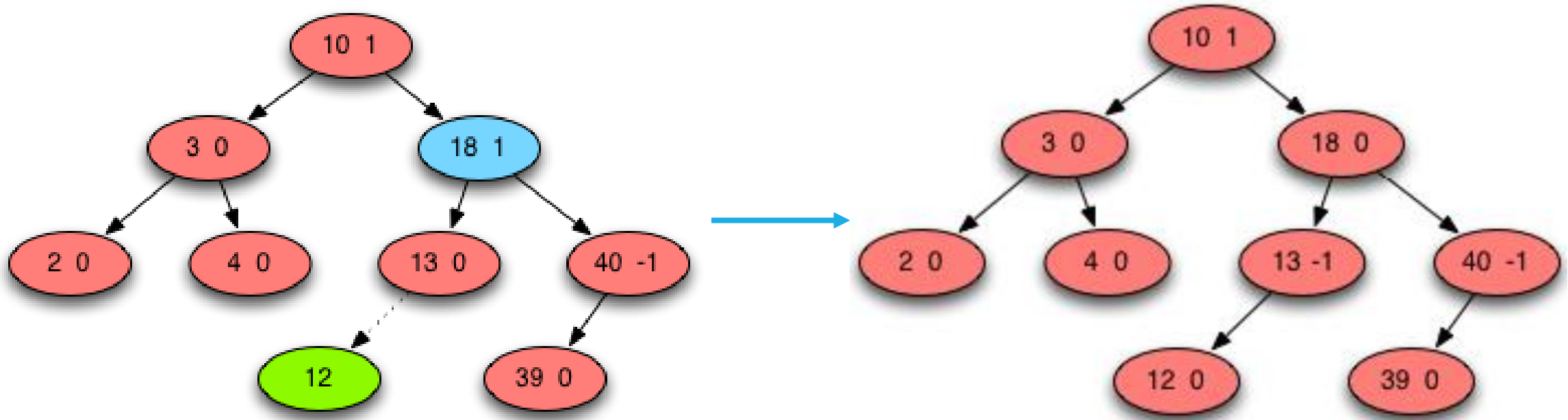


Inserting 39, no pivot

# AVL TREES: INSERTION (3)

- **Case 2: Pivot found, no rotation.**

- Subtree of the pivot node (where new node inserted) has smaller height.
- Adjust balance from new nodes up to pivot node.
- Balances of nodes above the pivot are not affected.
- Height of the subtree rooted at the pivot node is not changed by the insertion of the new node.



Inserting 12, pivot = 18

# AVL TREES: INSERTION (4)

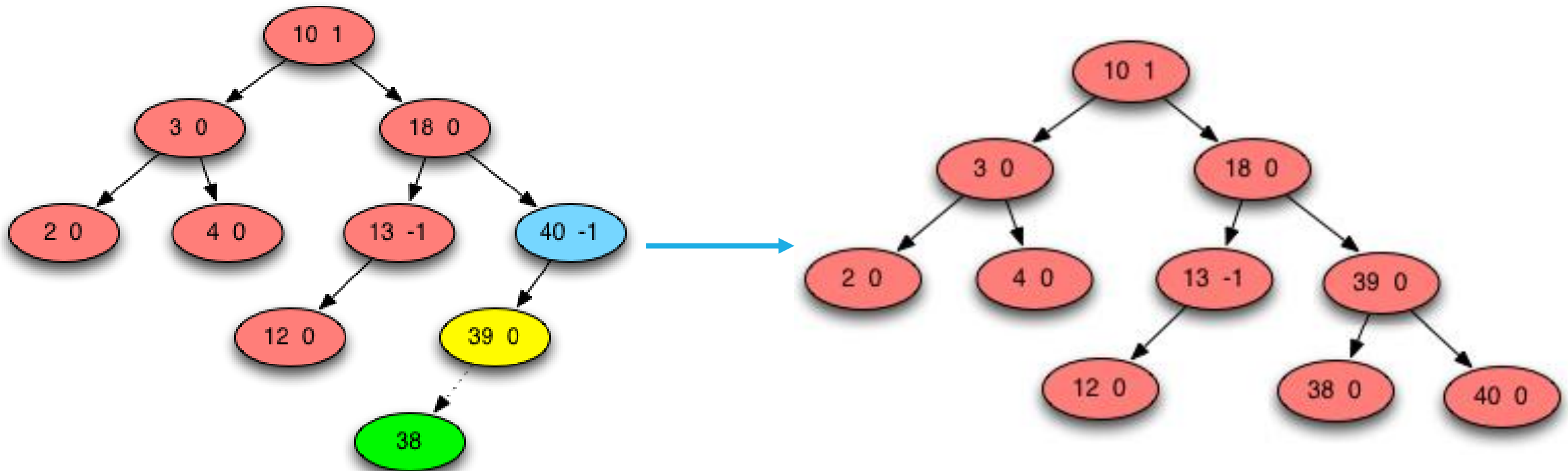
- **Case 3: Pivot found, rotation needed.**
  - Node is added to subtree of larger height.
  - Pivot balance becomes -2 or 2 -> tree is no longer AVL.
    - Single or double rotation must be performed.
  - **Bad child** – pivot child in the direction of the imbalance.
  - **Bad grandchild** – child of a bad child in the direction of the imbalance.
- **Two subcases must be considered:**
  - **Subcase A: Single rotation.**
    - Node is added to the subtree of a bad child in the **direction** of the imbalance.
    - **Rotate** at the **pivot** in the **direction opposite** to the imbalance.
  - **Subcase B: Double rotation.**
    - Node is added to the subtree of a bad child in the **direction opposite** to the imbalance.
    - **Rotate** at the **bad child** in the **direction** of the imbalance.
    - **Rotate** at the **pivot** in the **direction opposite** to the imbalance.

# AVL TREES: INSERTION (5)

- **Case 3: Pivot found, rotation needed (cont.)**

- **Subcase A: Single rotation.**

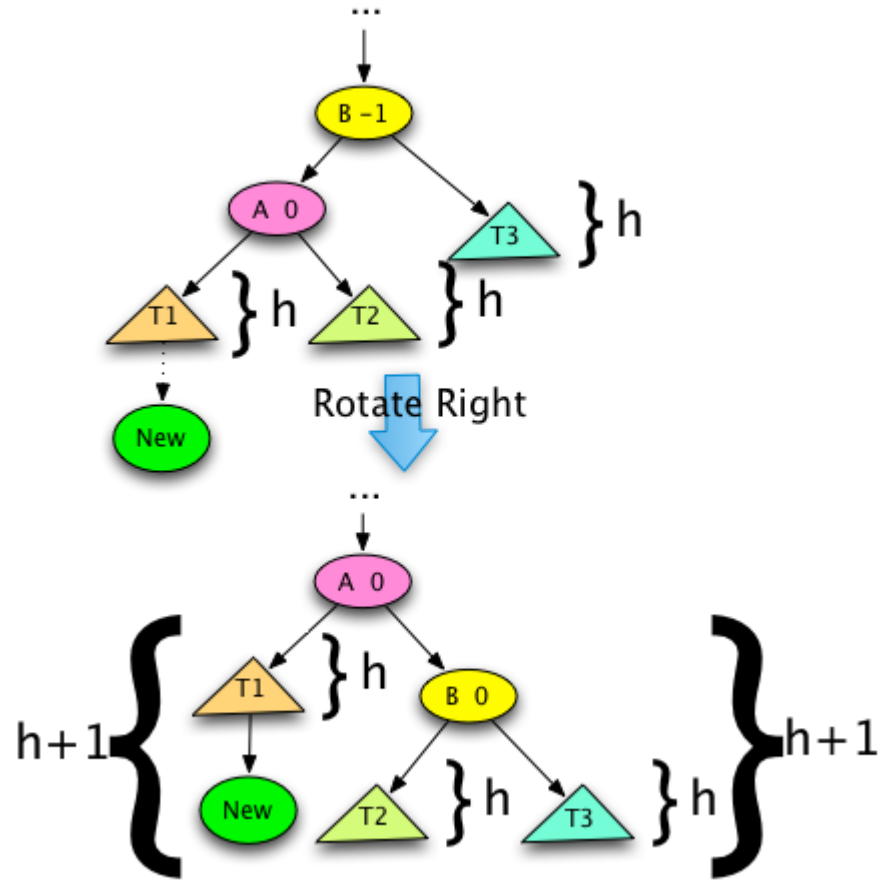
- Node is added to the subtree of a bad child in the direction of the imbalance.
- Rotate at the pivot in the direction opposite to the imbalance.



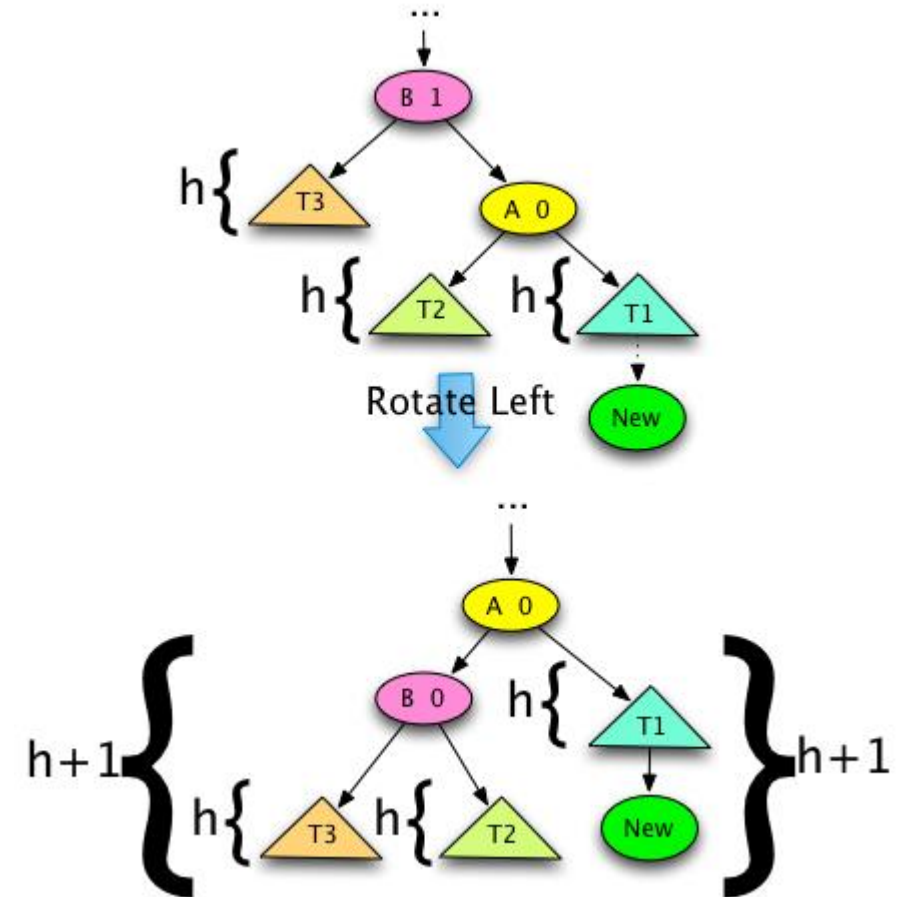
Inserting 38, pivot = 40, bad child = 39

# AVL TREES: INSERTION (6)

- Case 3: Pivot found, rotation needed (cont.)
  - Subcase A: Single rotation.



Case 3A right rotation



Case 3A left rotation

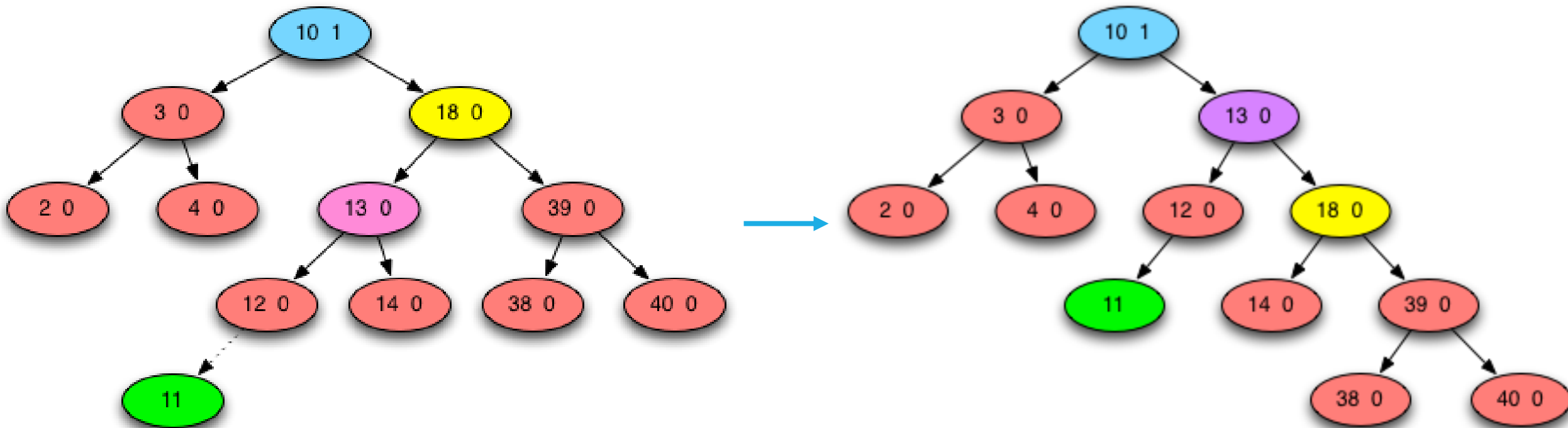


# AVL TREES: INSERTION (7)

- **Case 3: Pivot found, rotation needed (cont.)**

- **Subcase B: Double rotation.**

- Node is added to the subtree of a bad child in the direction opposite to the imbalance.
- **Rotate at the bad child in the direction of the imbalance.**
- Rotate at the pivot in the direction opposite to the imbalance.



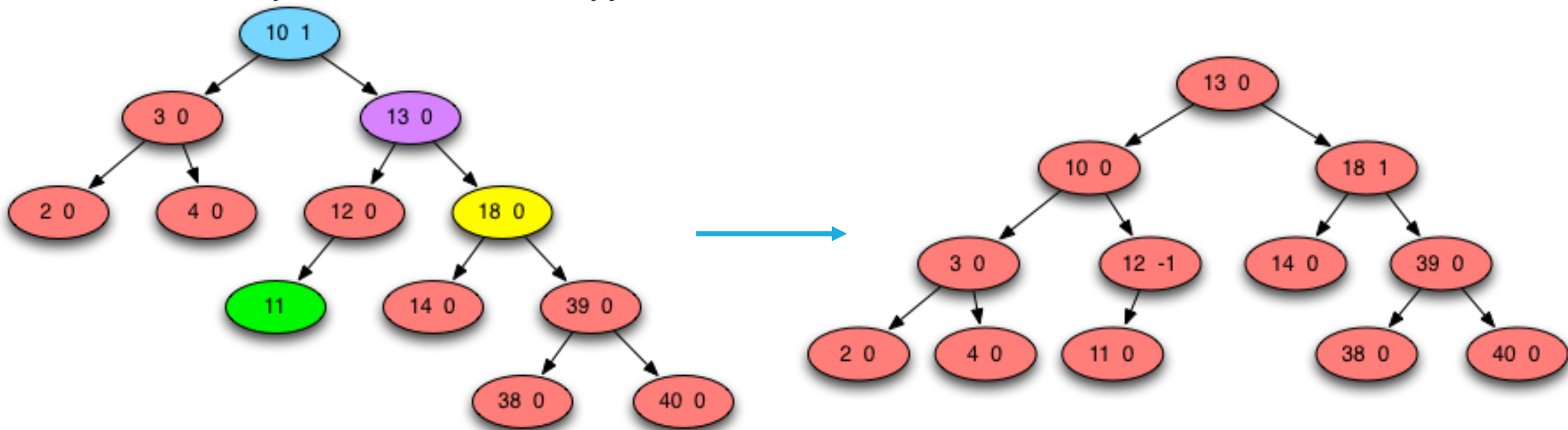
Inserting 11, pivot = 10, bad child = 18, bad grandchild = 13

# AVL TREES: INSERTION (8)

- **Case 3: Pivot found, rotation needed (cont.)**

- **Subcase B: Double rotation.**

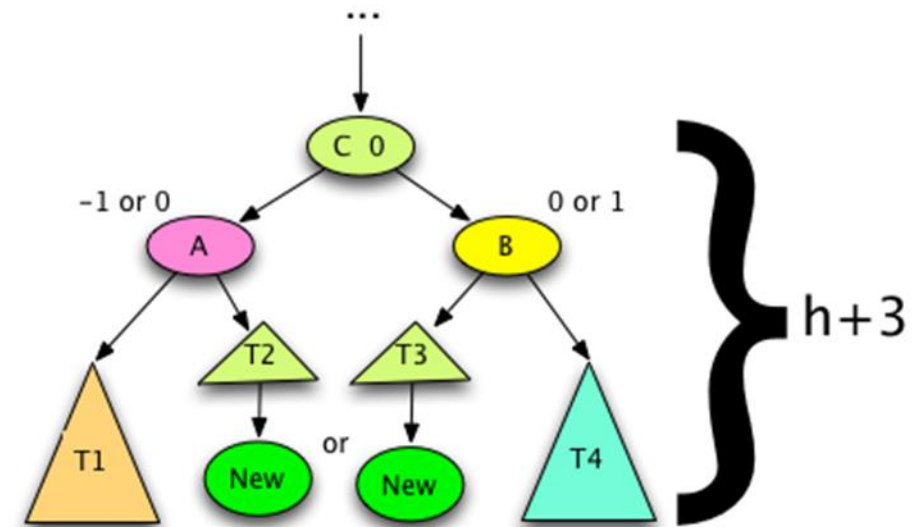
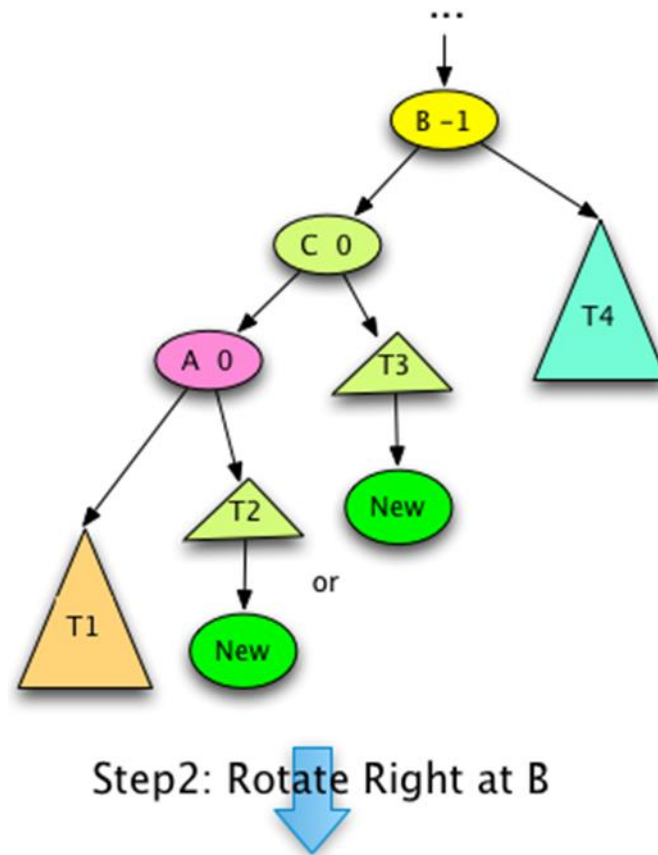
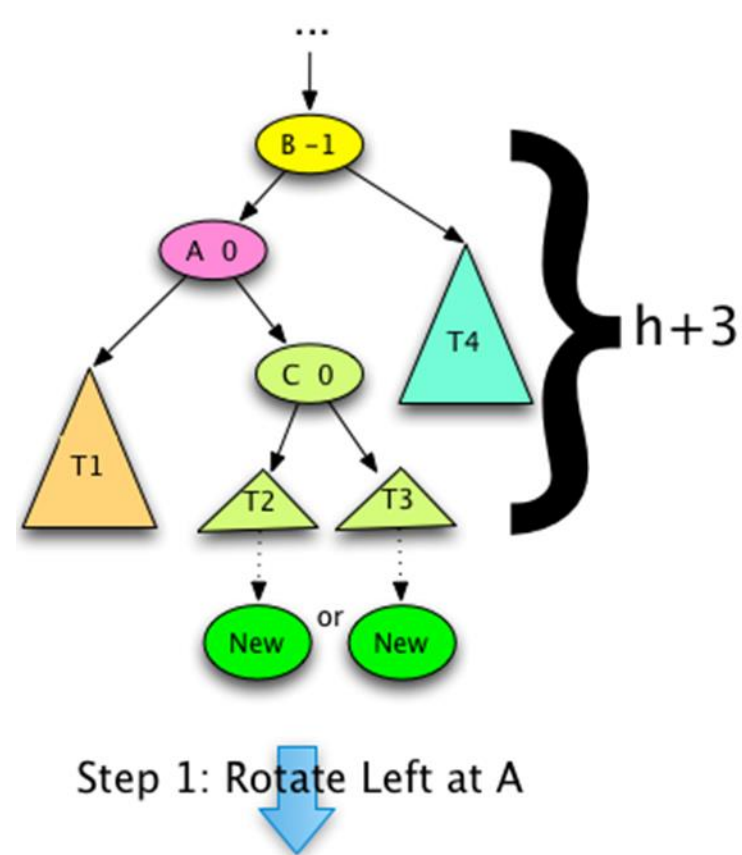
- Node is added to the subtree of a bad child in the direction opposite to the imbalance.
- Rotate at the bad child in the direction of the imbalance.
- **Rotate at the pivot in the direction opposite to the imbalance.**



Inserting 11, pivot = 10, bad child = 18, bad grandchild = 13

# AVL TREES: INSERTION (9)

- Case 3: Pivot found, rotation needed (cont.)
  - Subcase B: Double rotation.



# SPLAY TREES

- AVL tree **drawback** – node must maintain the **balance**.
  - **Extra work** to calculate + **extra space** to store.
  - **Improvement** – exploiting **spatial locality**.
- **Spatial locality**.
  - In large data sets a relatively **small subset** of data is **accessed** over a **short period** of time.
- **Splay tree** – *relatively* **balanced** binary search tree with no stored **balance**.
  - **Insert/lookup** moves the node to the root of the tree (**splaying** through **rotations**).
  - **Deletion** of item splays the parent node to the root of the tree.
- **Splay trees** performance is **same/better** than AVL trees on **random** data sets.
  - **Balanced enough** to get **amortized complexity** of  $\Theta(\log n)$  for insert/lookup/delete operations.

# SPLAY TREES: OPERATIONS & SPLAYING

- **Operations:**

- **Insertion.**

- Value splayed to the root.

- **Lookup.**

- Value found - splayed to the root.
    - Value not found - would-be parent splayed to the root.

- **Deletion.**

- Parent is splayed to the root.

- Value is splayed to the top of the tree through **rotations**:

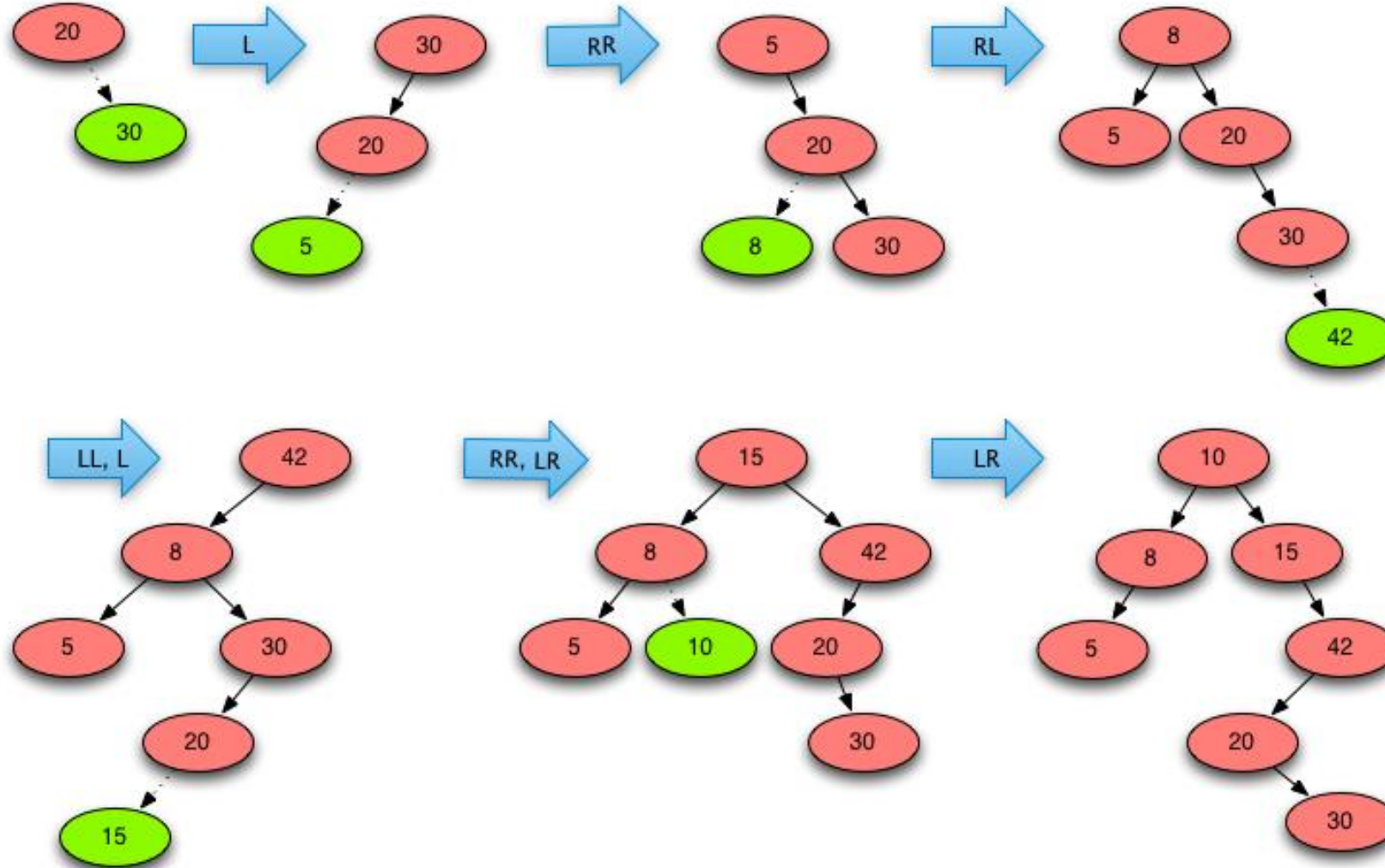
- Single rotations: *zig* or *zag*.
  - Double rotations: *zig-zig* or *zig-zag*.

- Series of **double rotations** take node to the **root** or to the **root child**.

- If at root child – additional **single rotation** to get to the root.

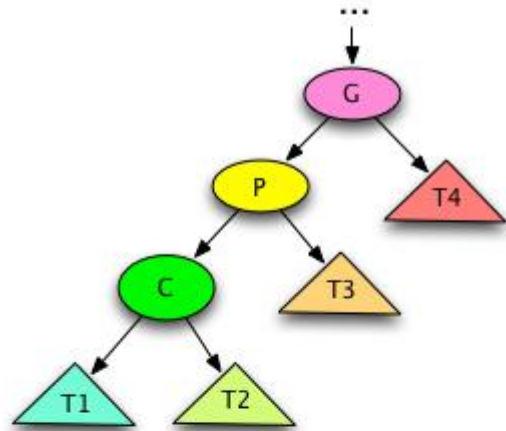
# SPLAY TREES: EXAMPLE

- Inserting values 30, 5, 8, 42, 15, 10 into Splay tree.

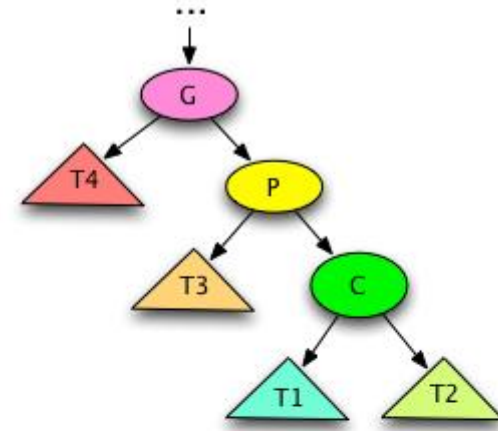
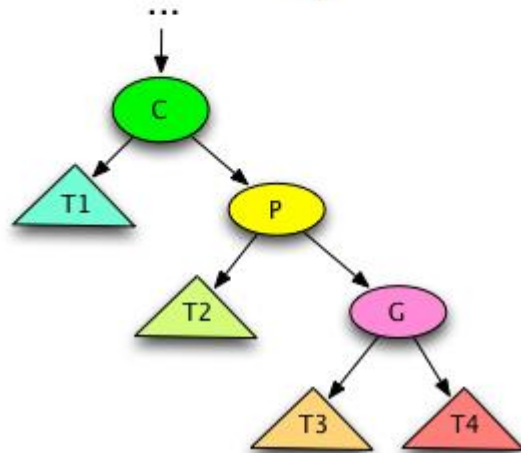


# SPLAY TREES: ROTATIONS (1)

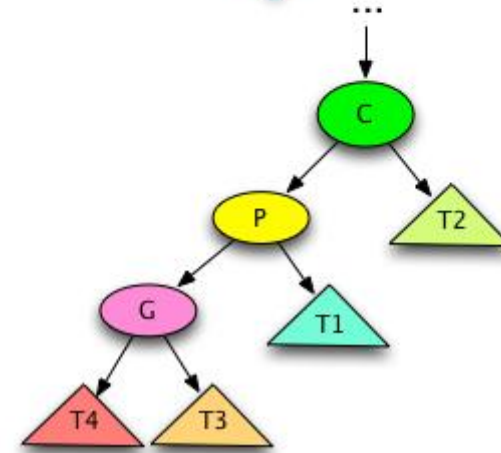
- Zig-zig rotations.



Rotate Right-Right



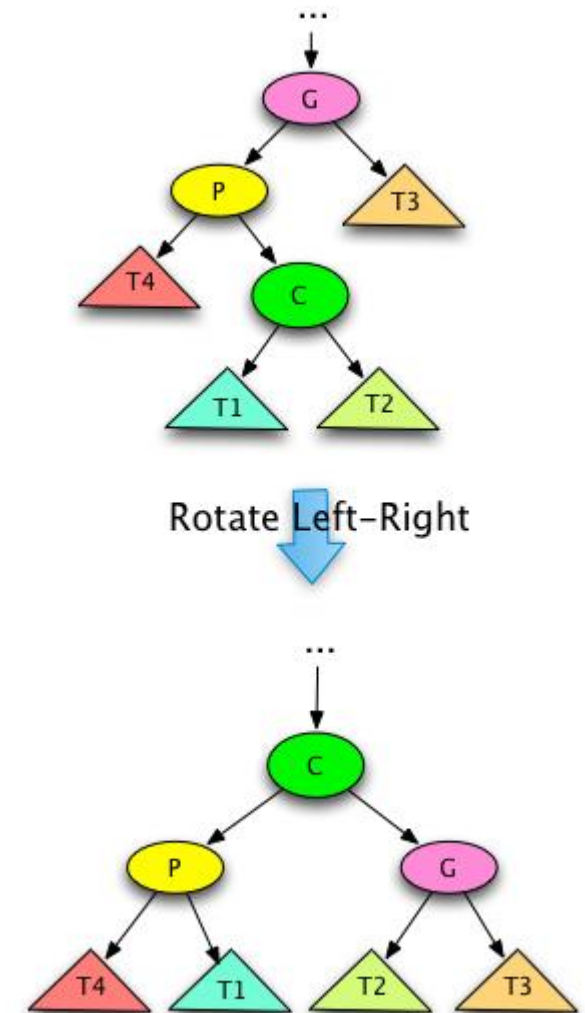
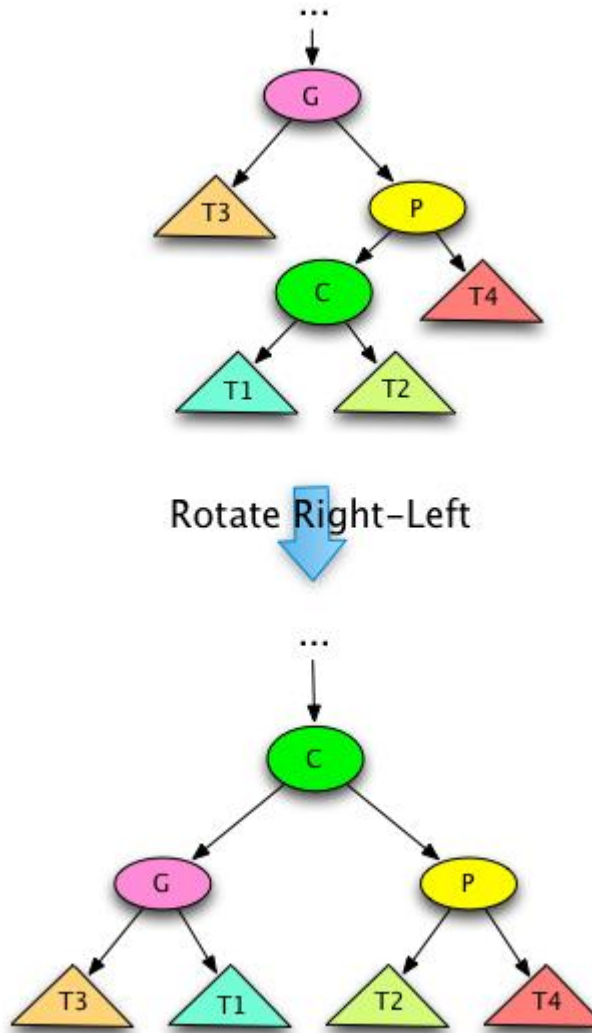
Rotate Left-Left





# SPLAY TREES: ROTATIONS (2)

- Zig-zag rotations.





# SUMMARY

---

- Trees.
- Binary search trees.
  - Properties, Insertion, deletion, traversal.
- Search spaces.
  - DFS + backtracking.
- Unbalanced vs. balanced trees.
- AVL trees.
  - Properties, insertions.
- Splay trees.
  - Spatial locality, properties, rotations, operations.