Lecture 23: Balanced Binary Search Trees

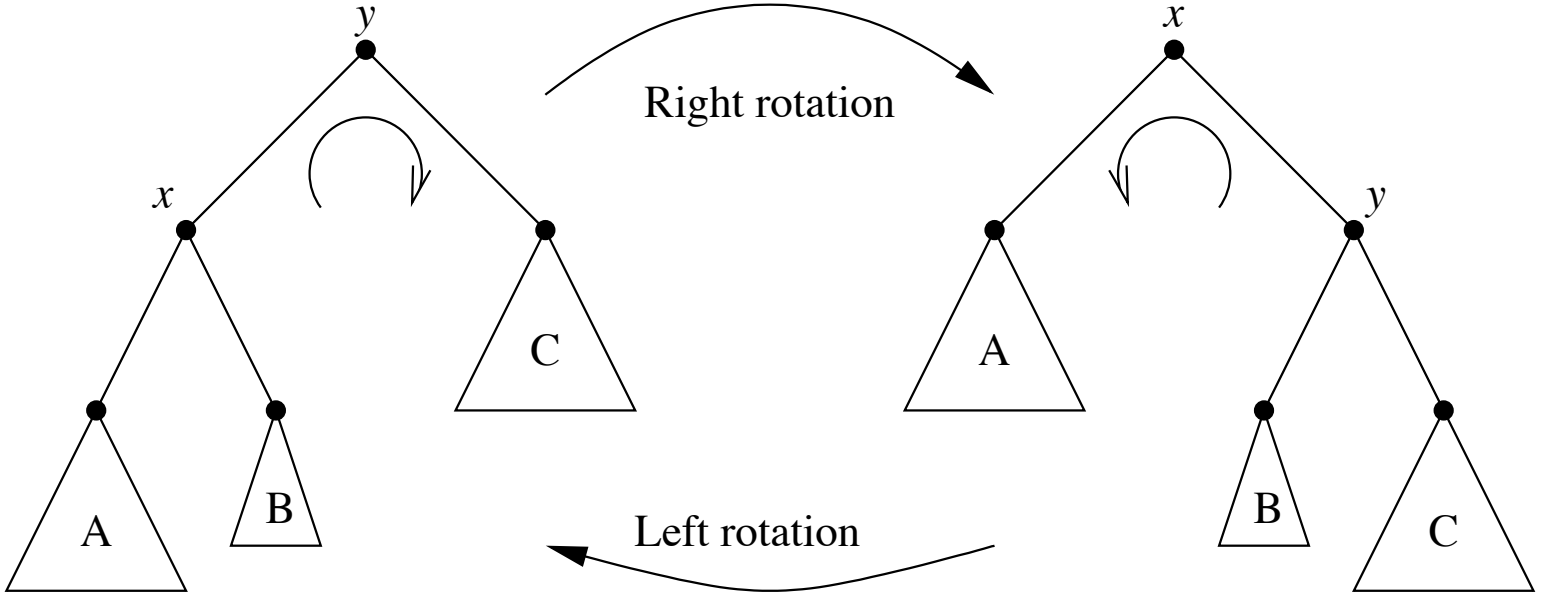
CS 62

Fall 2018

Alexandra Papoutsaki & William Devanny

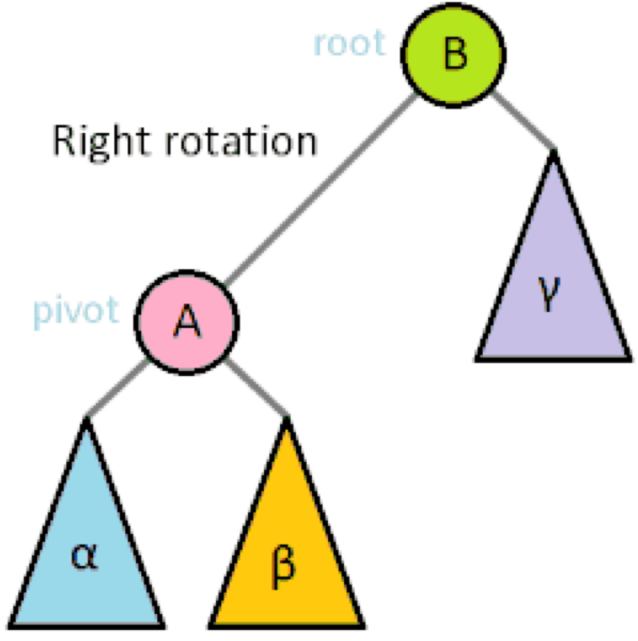
Rotating Binary Trees

Key idea: Rotate node higher in tree while keeping it in order.



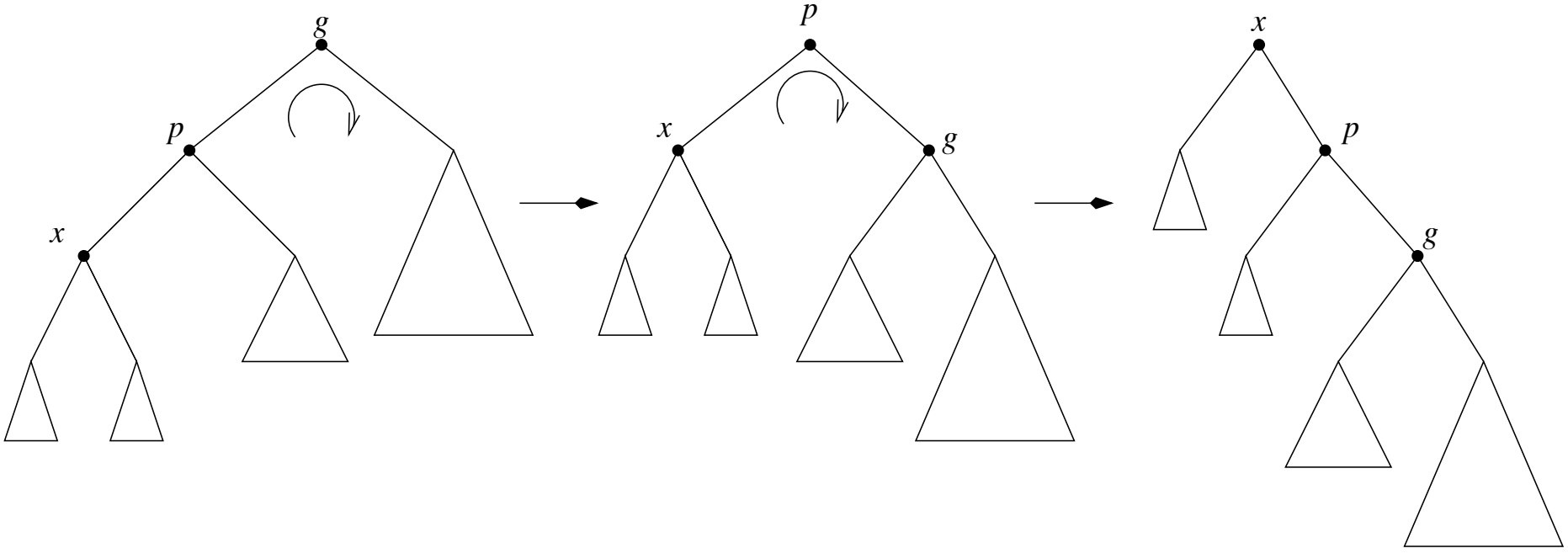
Rotating Trees

- Rotate A to root (Right rotation)
  - All nodes in subtrees α go up one level, all in γ go down one level, all in β stay same.
- Rotate B to root (Left Rotation)
  - All nodes in subtrees γ go up one level, all in α go down one level, all in β stay same.
- See code in BinaryTree.java



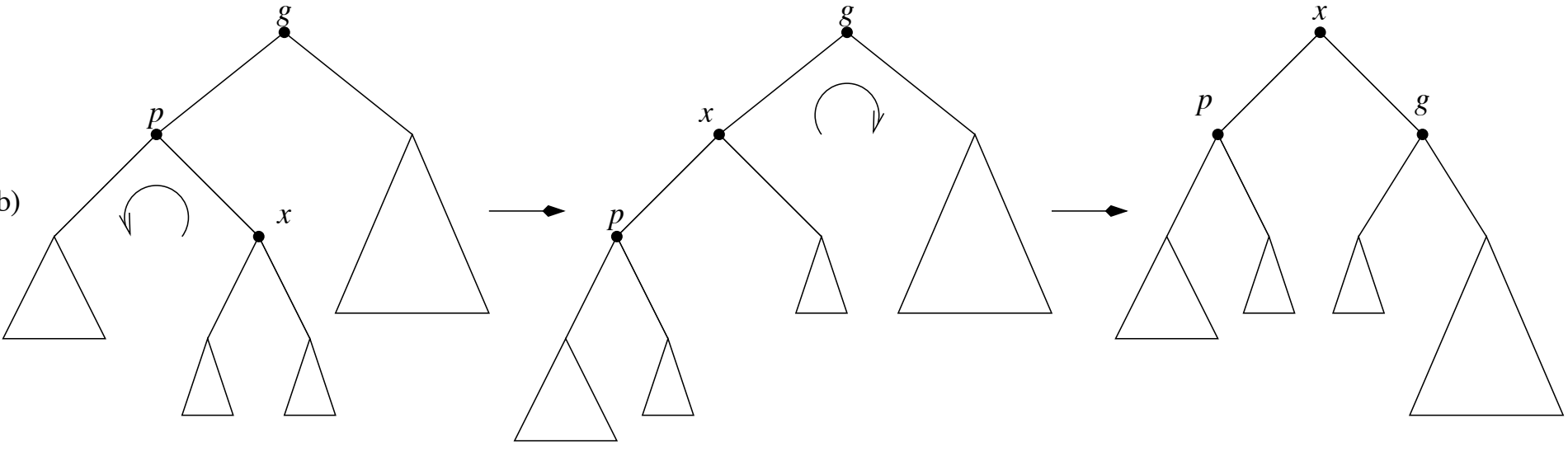
Shifting elements toward root

- Move x up two levels w/ two rotations
- If x is left child of a left child…



Shifting elements toward root

- If x is a right child of a left child…



Symmetric if interchangeable left and right

Splay Trees

- Self-adjusting Binary Search Trees
- Fast access to elements accessed recently (locality)

- Every time `contains`, `add` or `remove` an element **x**, move it to the root by a series of rotations (splay x)

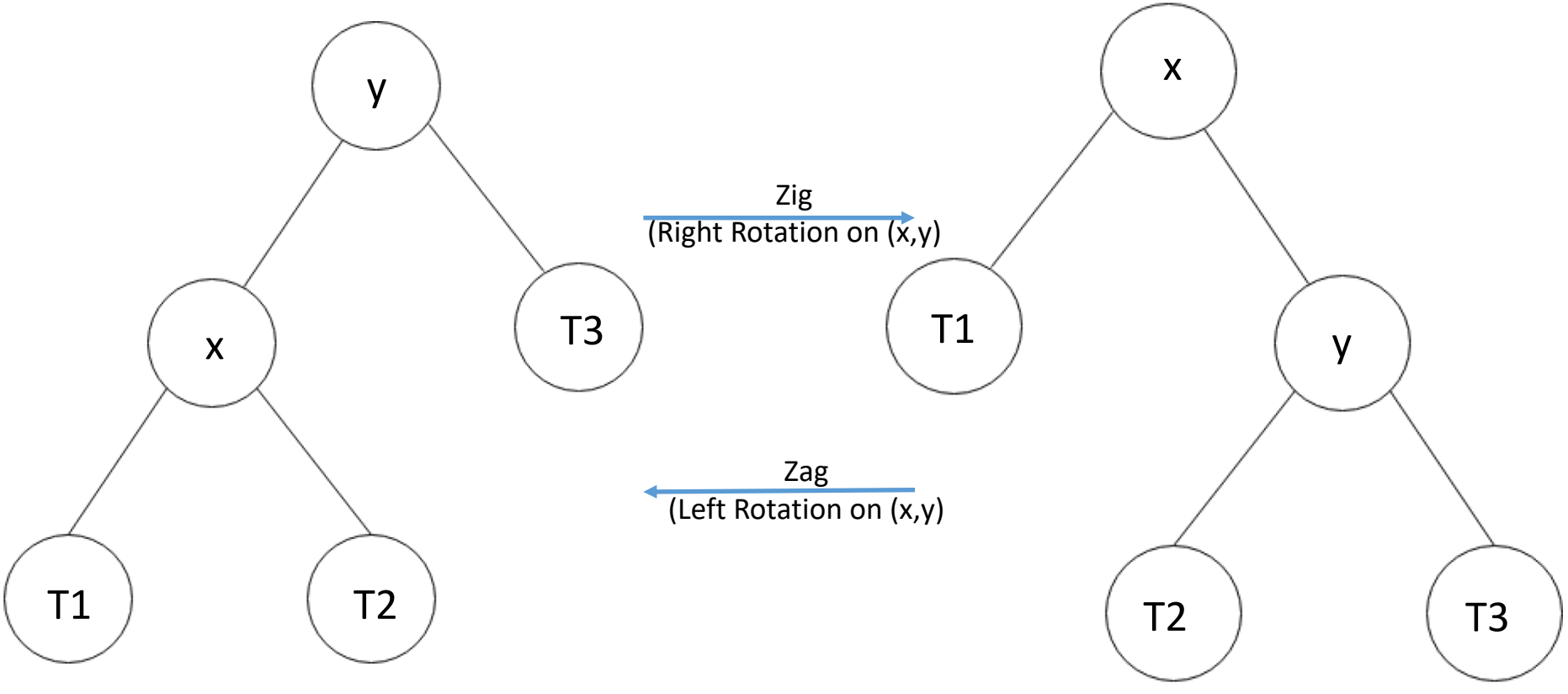
- Splay trees are balanced
  - But are not strictly balanced, unlike AVL trees
  - On average height is $O(\log n)$
  - Worst case height is $O(n)$
  - Average and worst case amortized cost is $O(\log n)$ for all operations

- Popular: caches, garbage collectors, routing

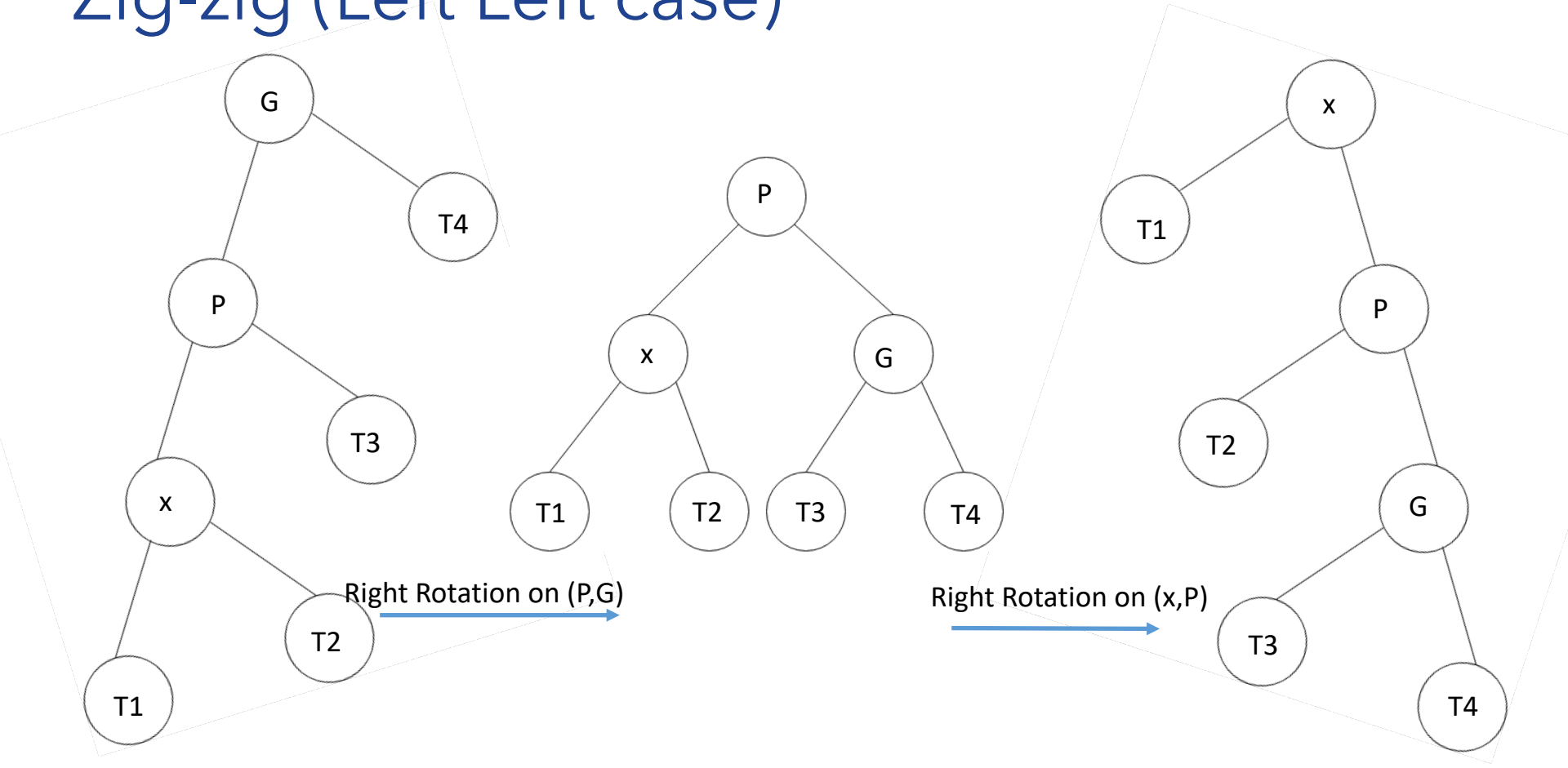
Splay operations

- Zig
  - Zag
- Zig-zig
  - Zag-zag
- Zig-zag
  - Zag-zig

Zig or Zag: node is child of root

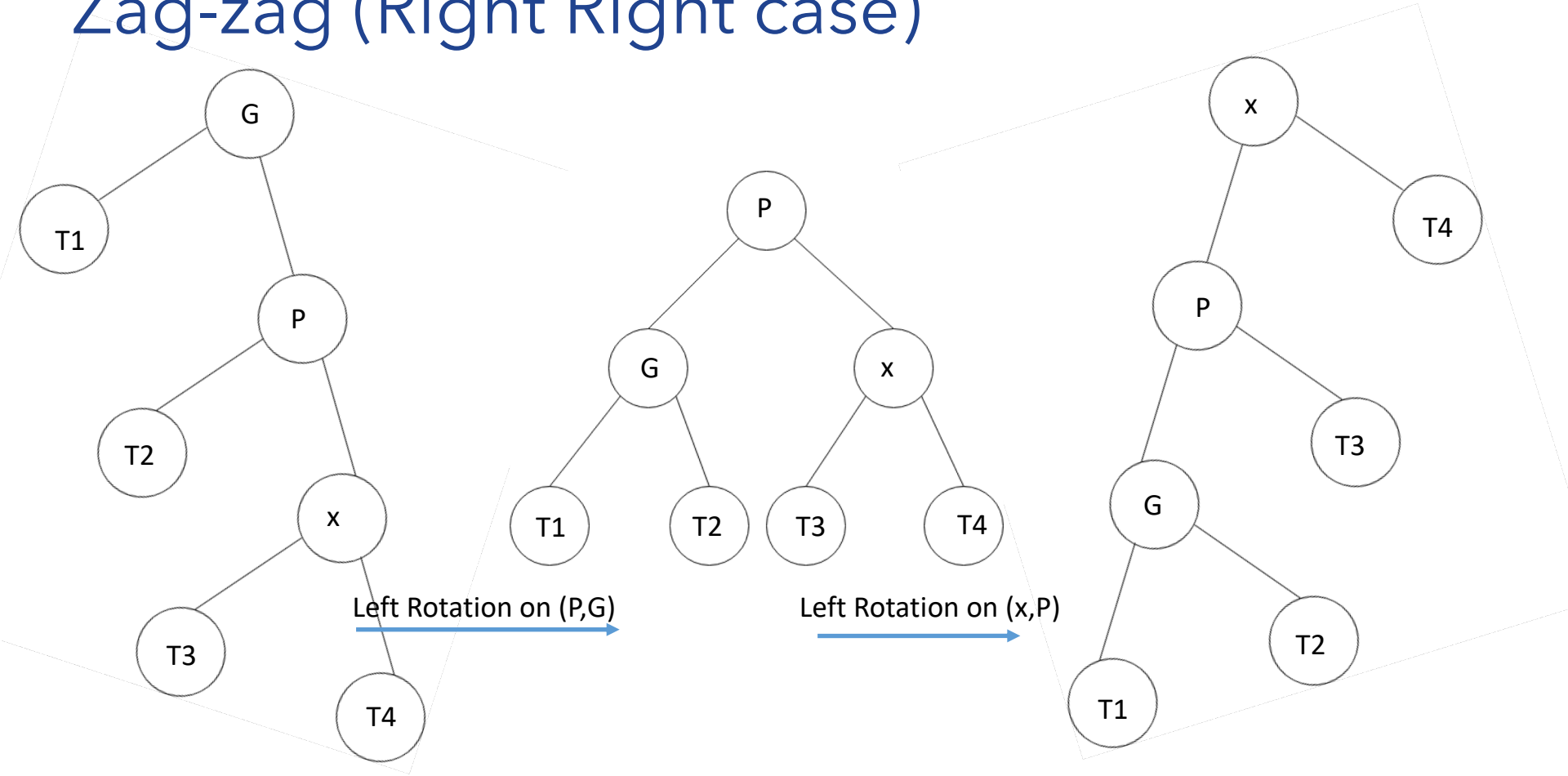


Zig-zig (Left Left case)



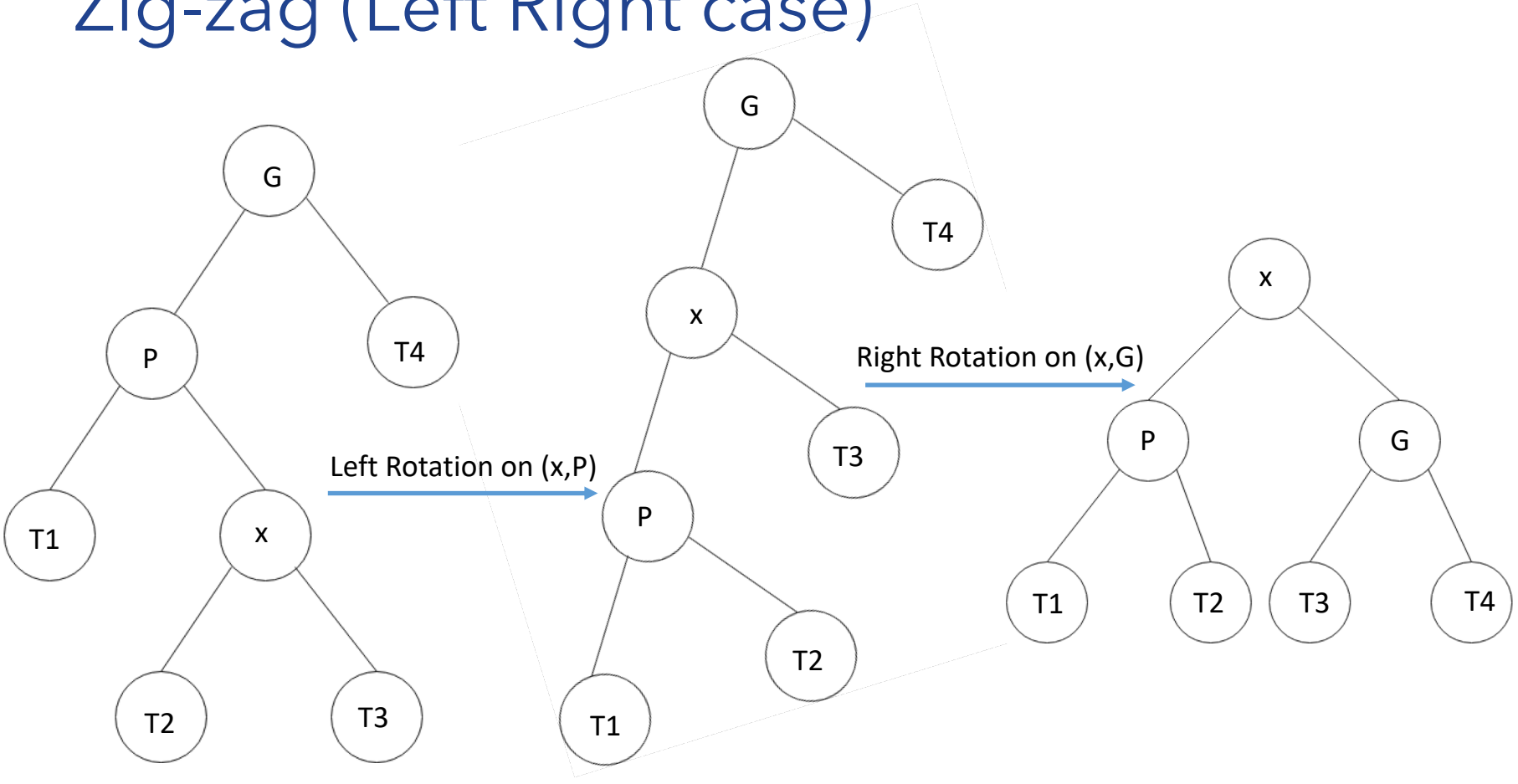
Node **x** has both parent **P** and grandparent **G**. Both **x** and **P** are the left children of their parents.

Zag-zag (Right Right case)



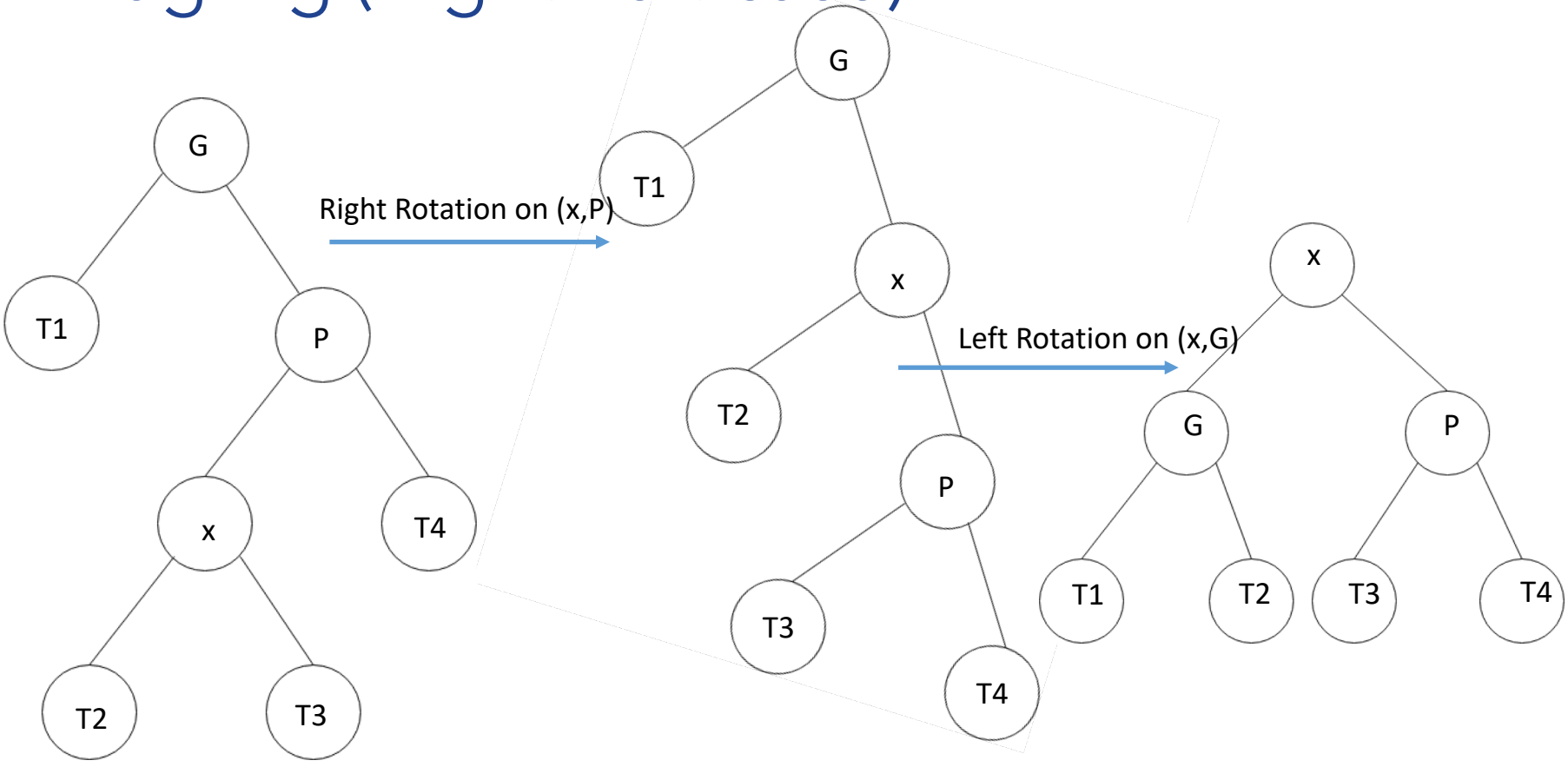
Node **x** has both parent **P** and grandparent **G**. Both **x** and **P** are the right children of their parents.

Zig-zag (Left Right case)



Node **x** has both parent **P** and grandparent **G**. **x** is right child and **P** is left child of their parents.

Zag-zig (Right Left case)



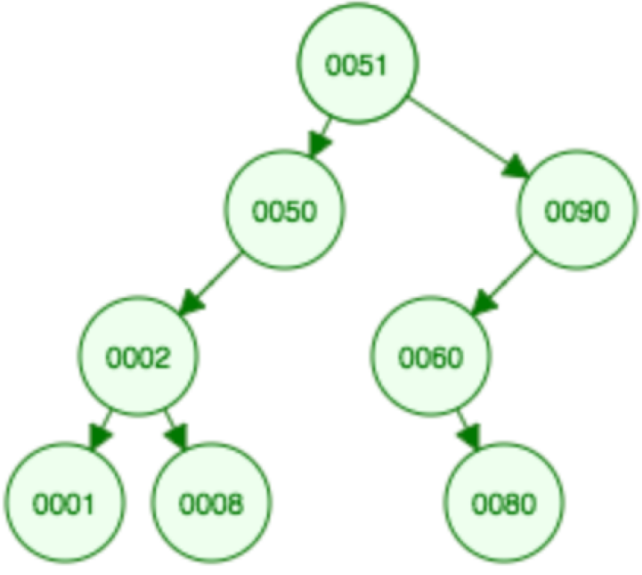
Node **x** has both parent **P** and grandparent **G**. **x** is left child and **P** is right child of their parents.

Operation sketches

- **`contains`**: Use `locate` method of BSTs. Splay the located node
- **`add`**: Use `add` method of BSTs and then splay node
- **`remove`**: Use `contains` to splay the node to be removed to the root. Delete it. You now have two independent trees. Find predecessor in left tree and splay it. Make root of right subtree its right child.
- https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

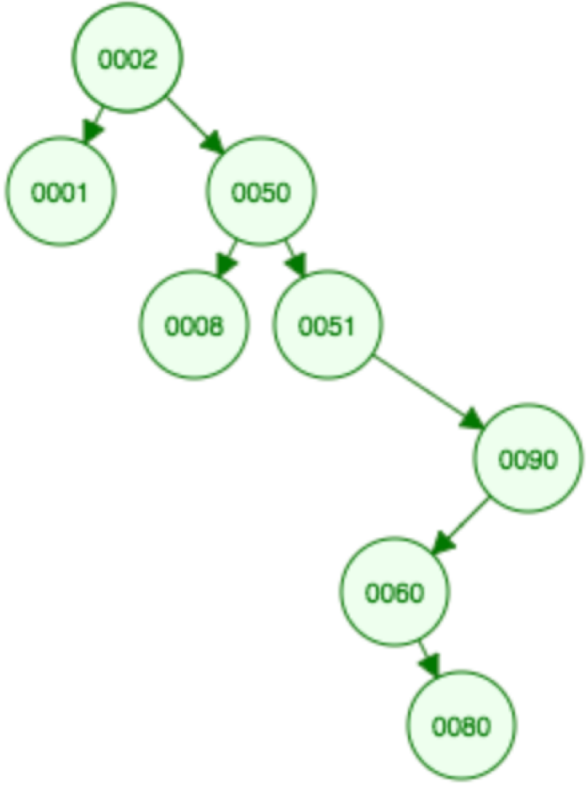
Practice time

- add : 8 1 80 50 2 60 90 51



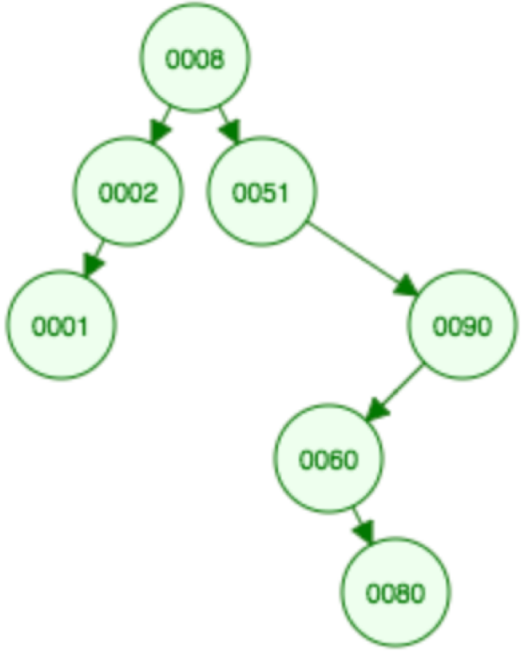
Practice time

- `contains`: 2



Practice time

- `remove`: 50



Why splay trees?

- In some applications, 80% of the time is spent approximately on 20% of all items
- Splay trees bring the most frequently-accessed items closer to the root, significantly reducing the search time (think of binary search on a BST) → locality of reference
- *Performance analysis of BSTs in system software*. Pfaff (2014) https://dl.acm.org/citation.cfm?id=1005742

- *Self-Adjusting Binary Search Trees*. Sleator and Tarjan (1985)

1999 ACM Paris Kanellakis Award