CSCE 3110
Data Structures and Algorithms

Splay Tree

Reading: Weiss, chap. 4

Contents

- Splay tree
  - insertion
  - find
  - deletion
  - running time analysis Binary Trees

Self adjusting Trees

- Ordinary binary search trees have no balance conditions
  - What you get from insertion order is it

- Balanced trees like AVL trees enforce a balance condition when nodes change
  - Tree is always balanced after an insert or delete

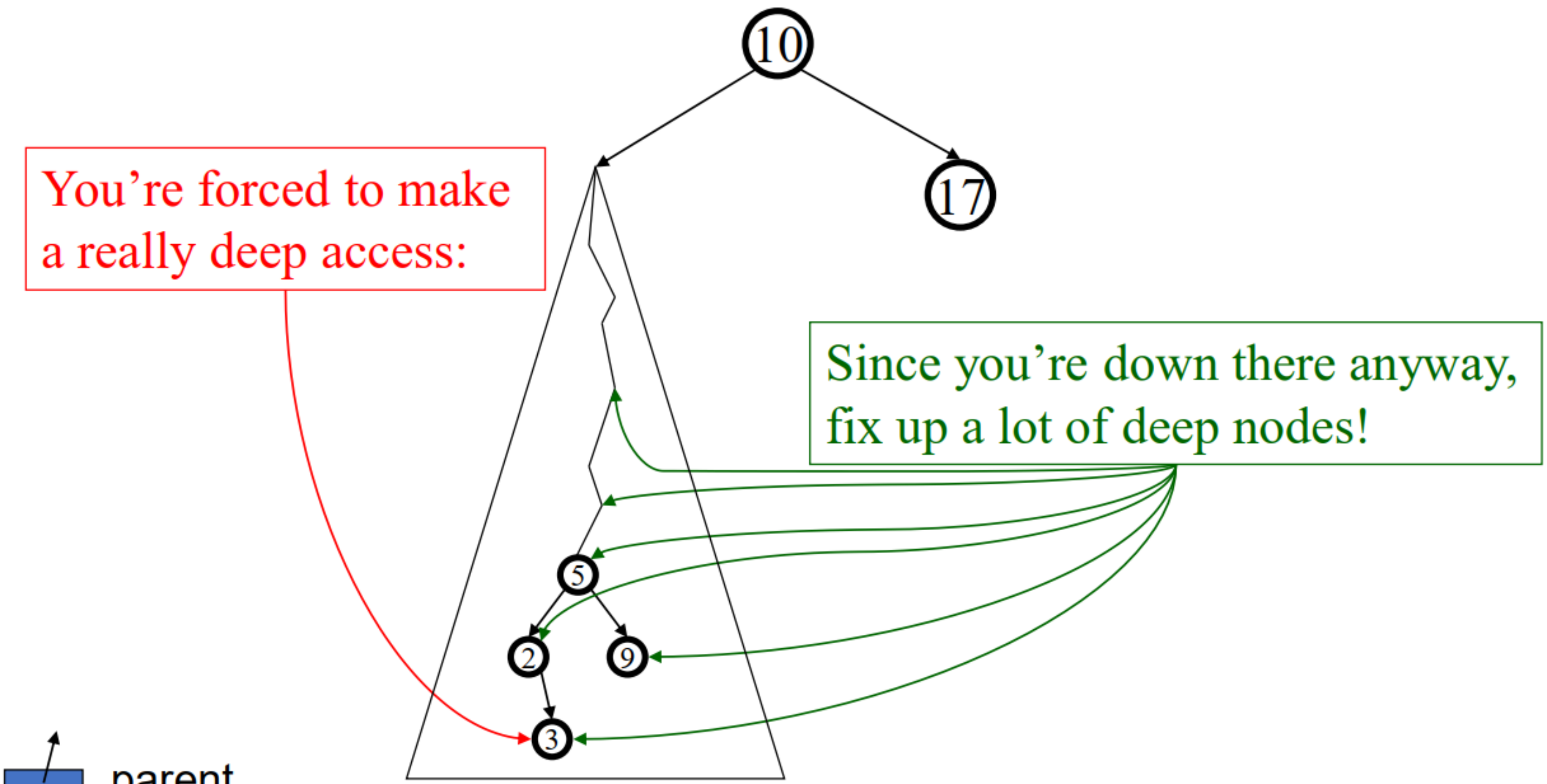
- Self-adjusting trees get reorganized over time as nodes are accessed
  - Tree adjusts after insert, delete, or find

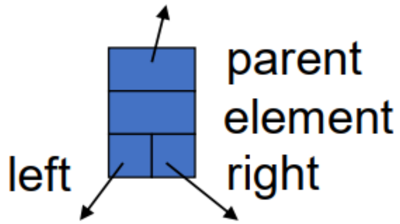
Splay Trees

- Splay trees are tree structures that:
  - Are not perfectly balanced all the time
  - Data most recently accessed is near the root. (principle of locality; 80-20 "rule")

- The procedure:
  - After node X is accessed, perform "splaying" operations to bring X to the root of the tree.
  - Do this in a way that leaves the tree more balanced as a whole

Splay Tree Idea



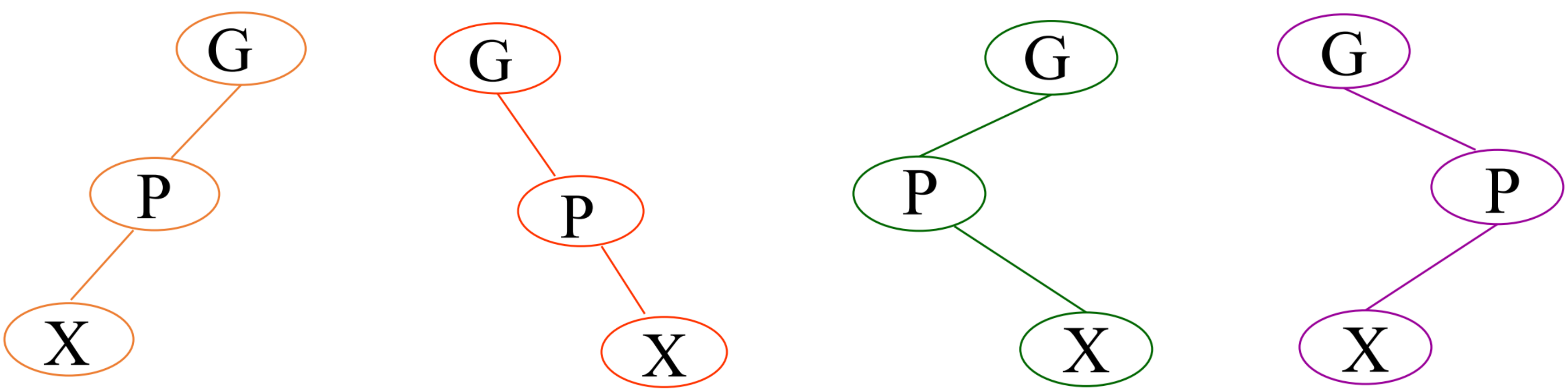


Helpful if nodes contain a parent pointer.

Splaying Cases

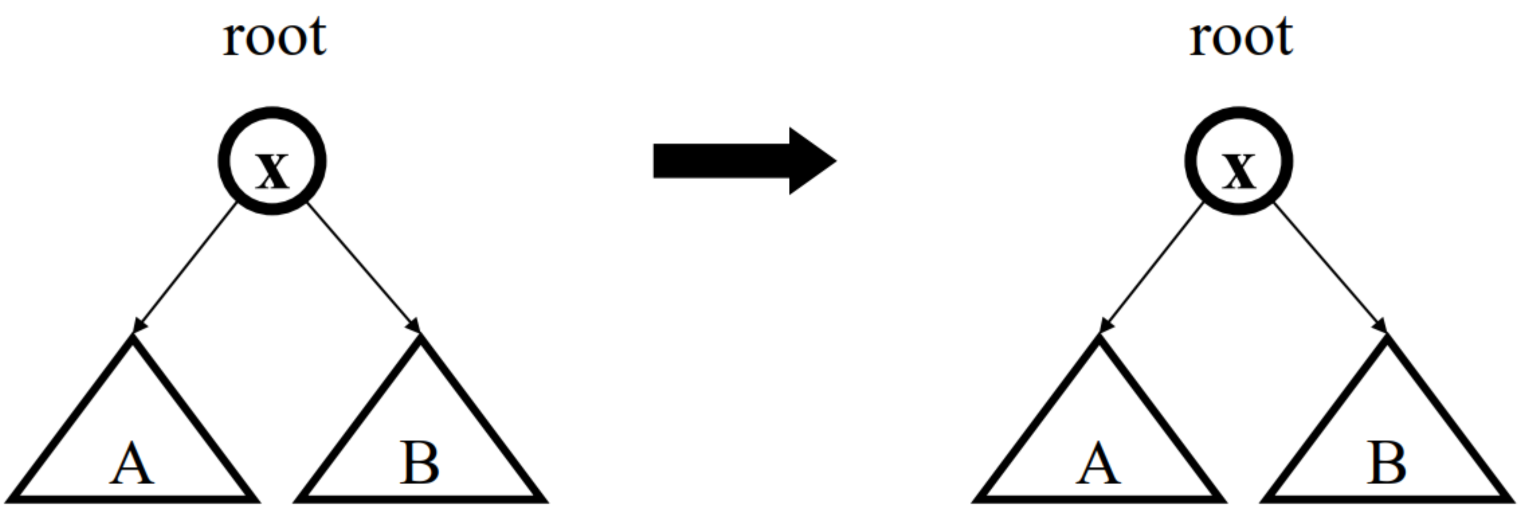
Node being accessed (x) is:

- Root
- Zig pattern: x is the child of root
- Has both parent (p) and grandparent (g)
  - Zig-zig pattern: g → p → x is left-left or right-right
  - Zig-zag pattern: g → p → x is left-right or right-left



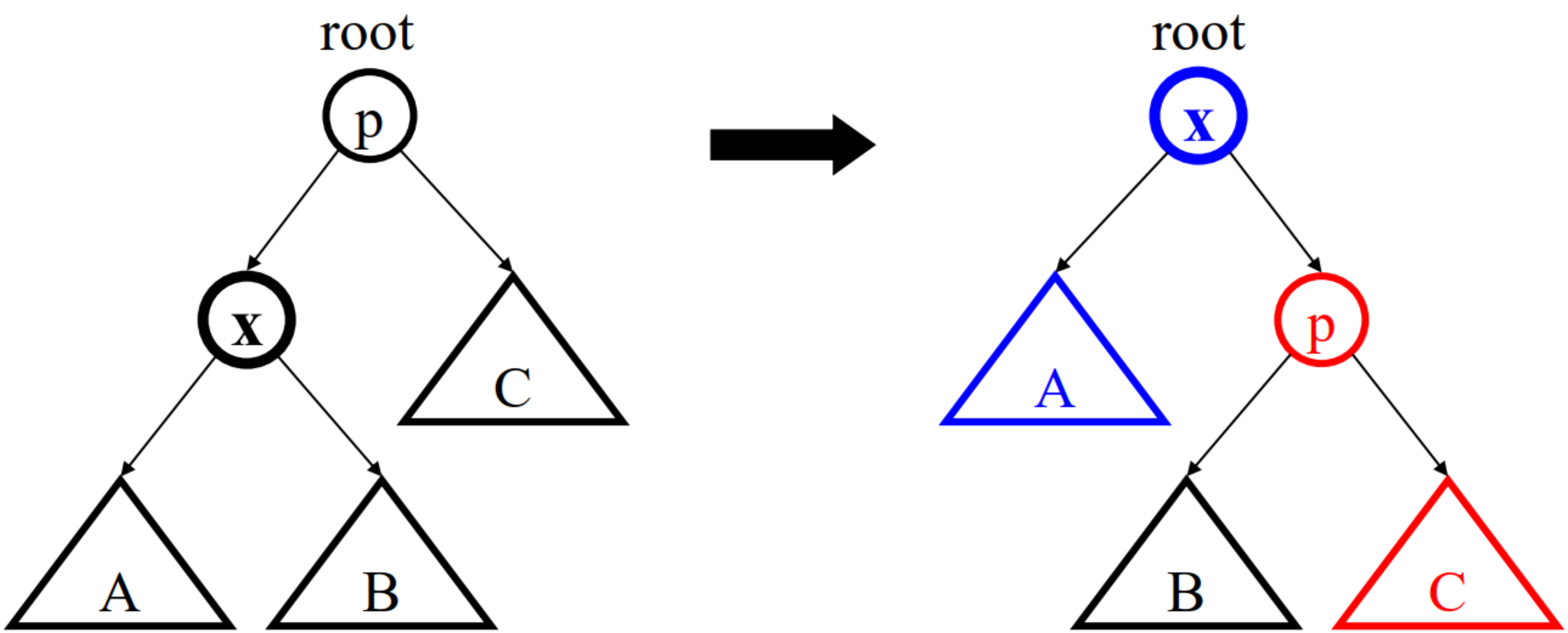
Access Root

Do nothing (that was easy!)



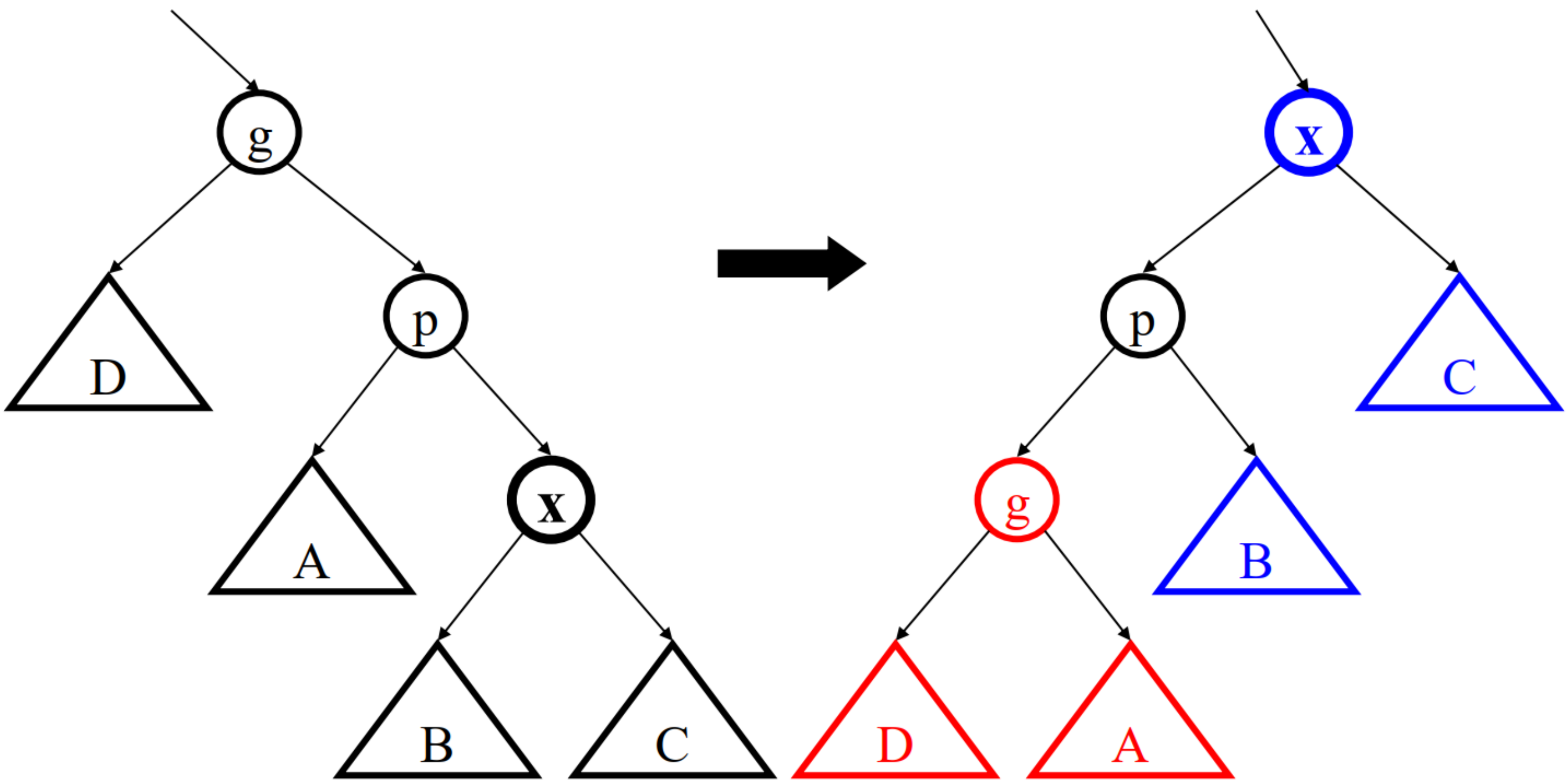
Access Child of Root

Zig (AVL single rotation)



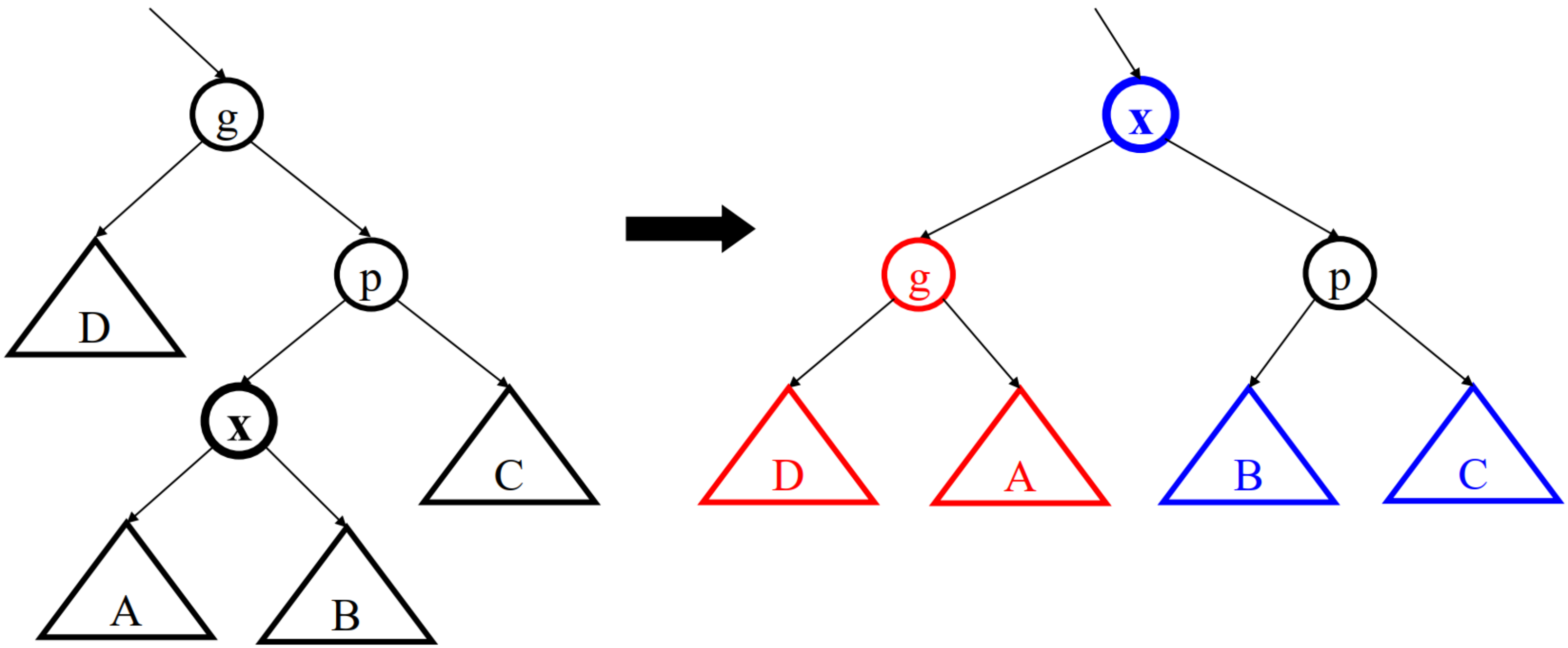
Access (LL, RR) Grandchild

Zig-Zig



Access (LR, RL) Grandchild

Zig-Zag



Summary of Zig-Zig and Zig-Zag

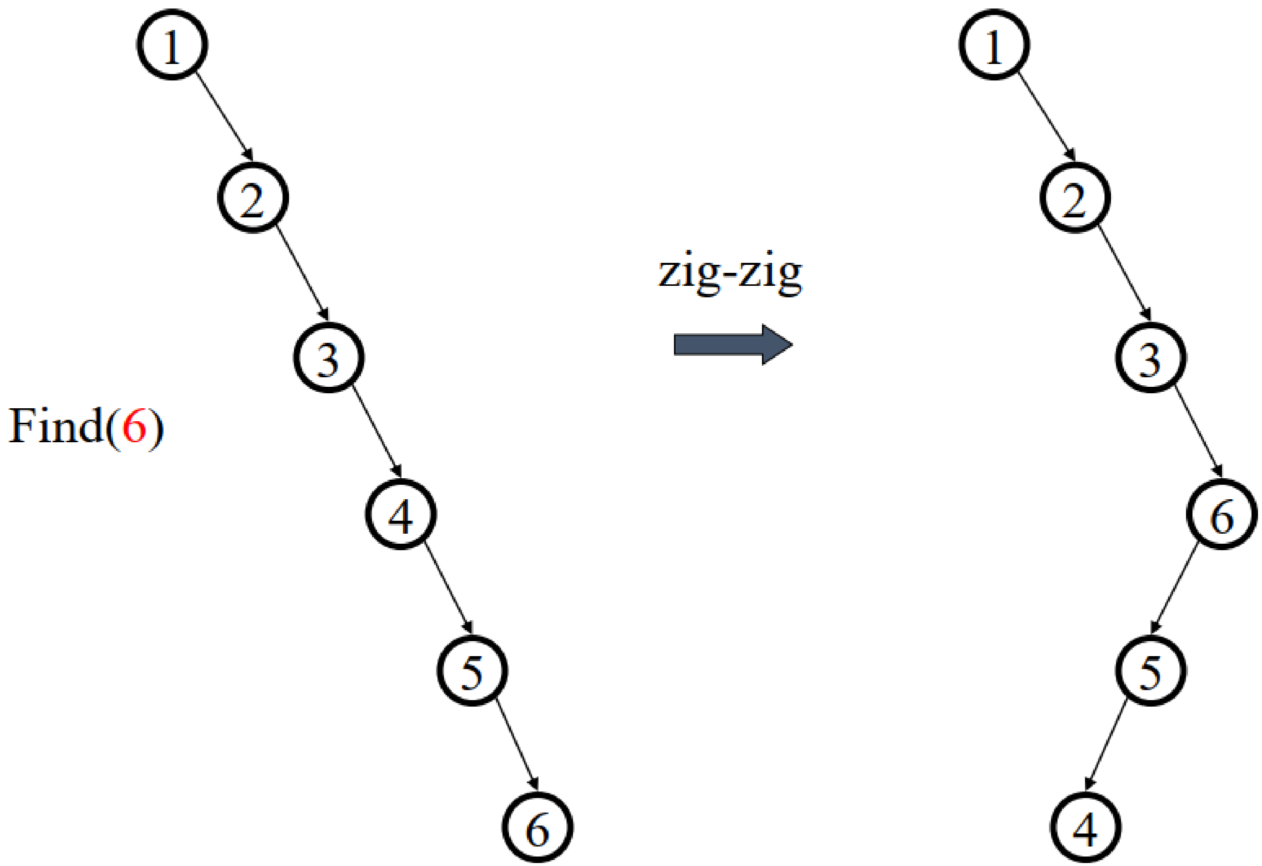
- Zig-Zig
  - If the node (e.g., x) being accessed is left-left, do two right rotations. The first rotation is on x's parent node, and the second rotation is on node x;
  - If the node (e.g., x) being accessed is right-right, do two left rotations. The first rotation is on x's parent node, and the second rotation is on node x;
- Zig-Zag
  - If the node (e.g., x) being accessed is left-right, do left rotation on node x, and then do right rotation on node x;
  - If the node (e.g., x) being accessed is right-left, do right rotation on node x, and then do left rotation on node x;

Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root Are not perfectly balanced all the time

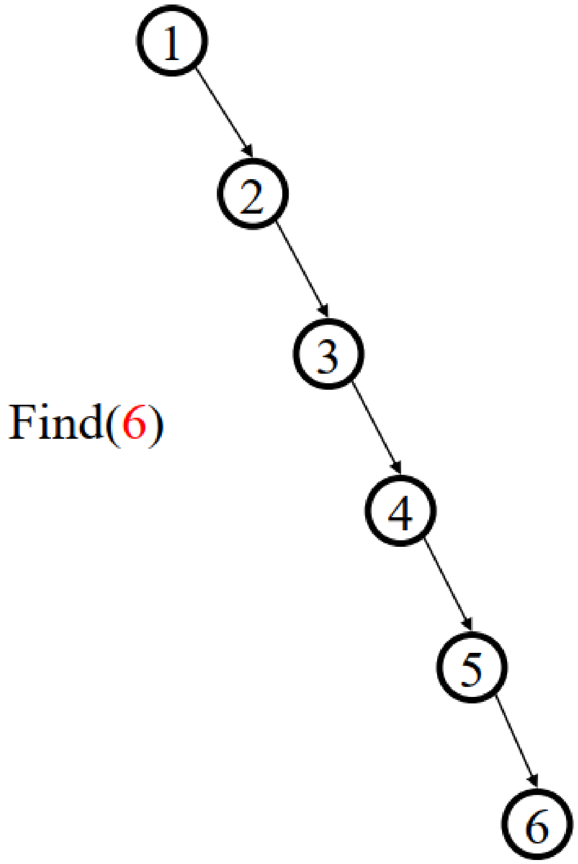
Splaying Example

Find(6)



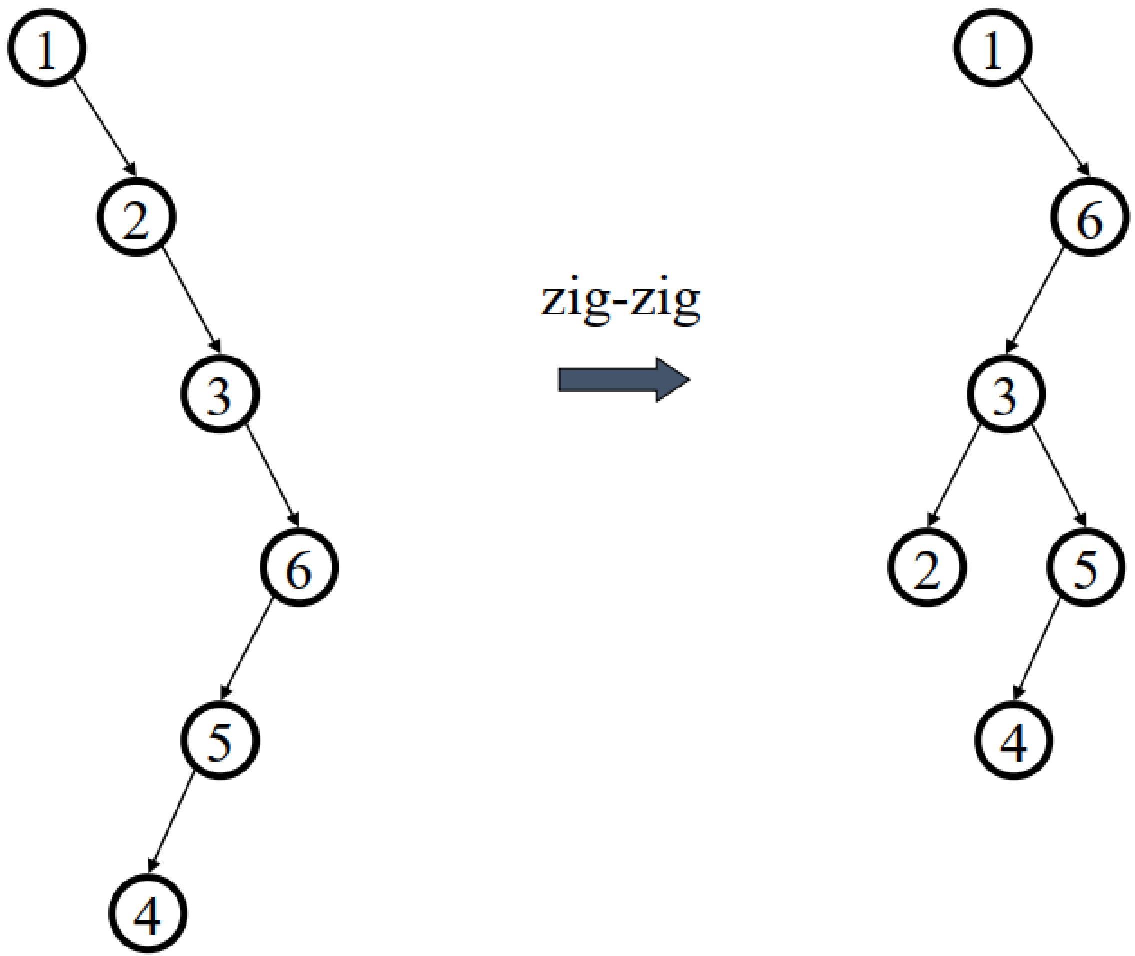
Splaying Example

Find(6)



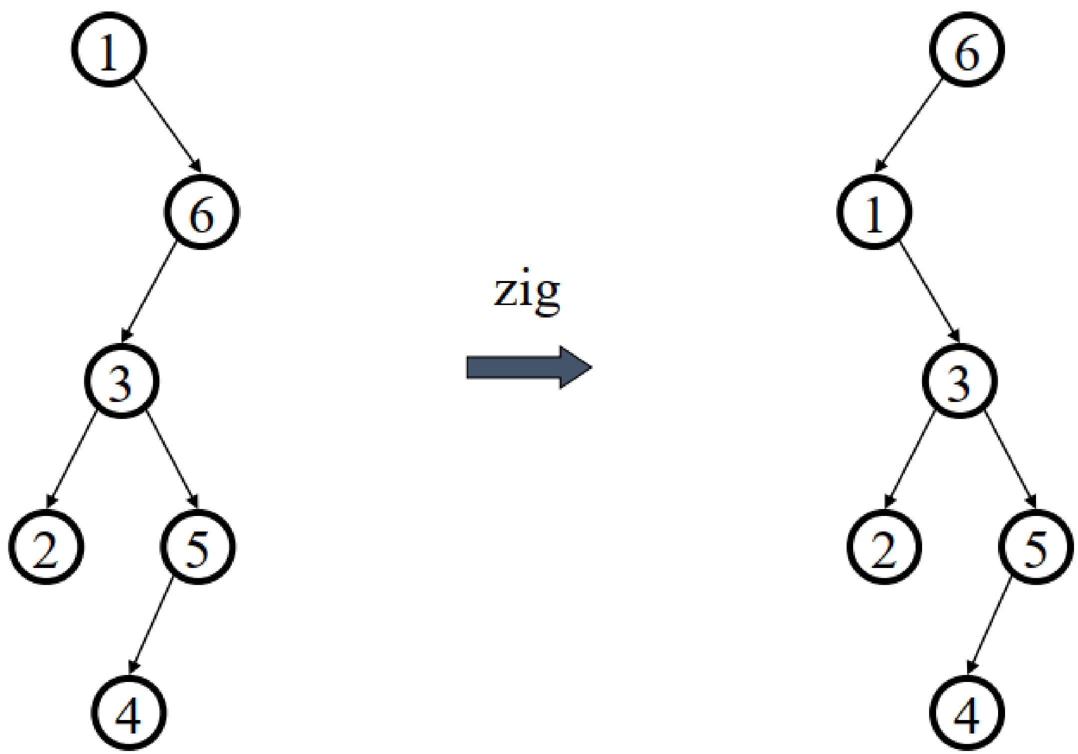
Splaying Example

… still splaying …



Splaying Example

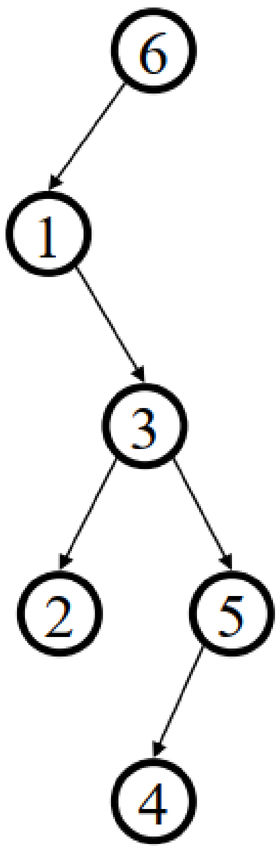
… 6 splayed out!



Splaying Example

Find (4)

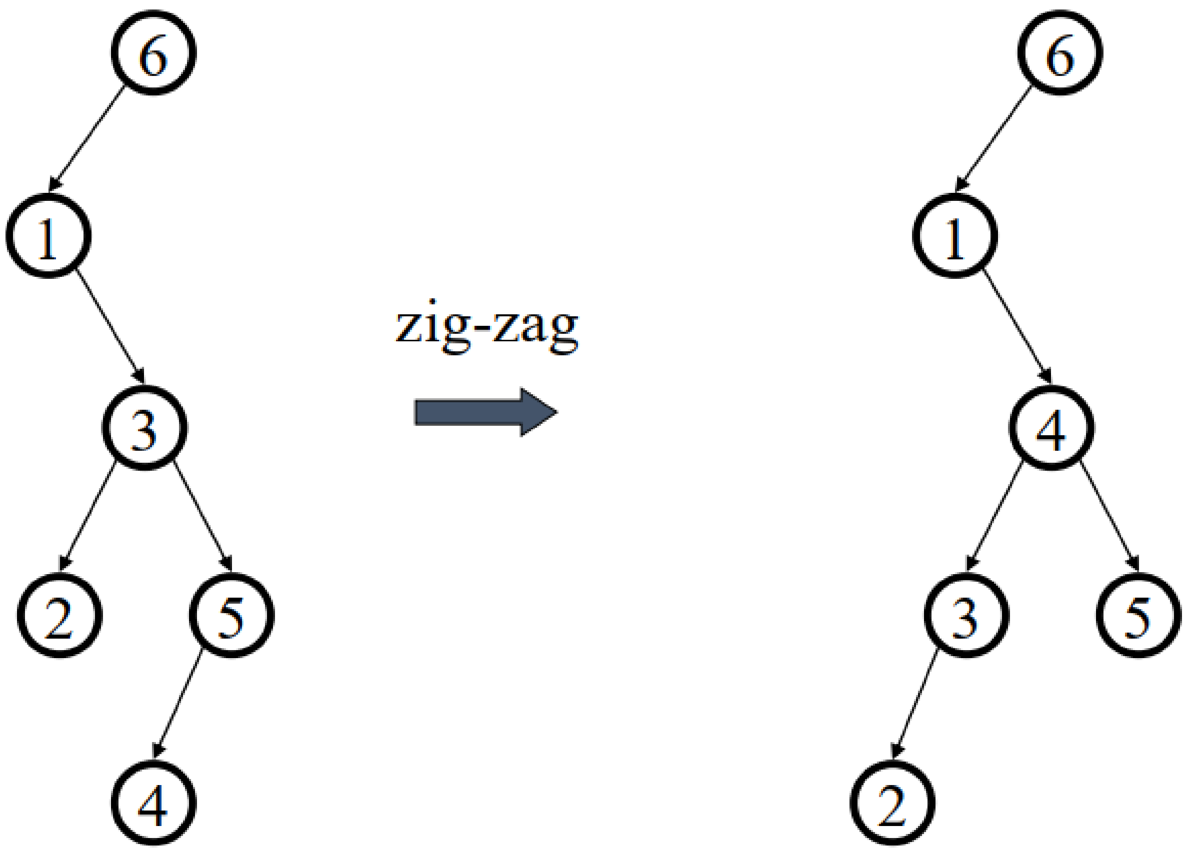
Splay it Again!



Splaying Example

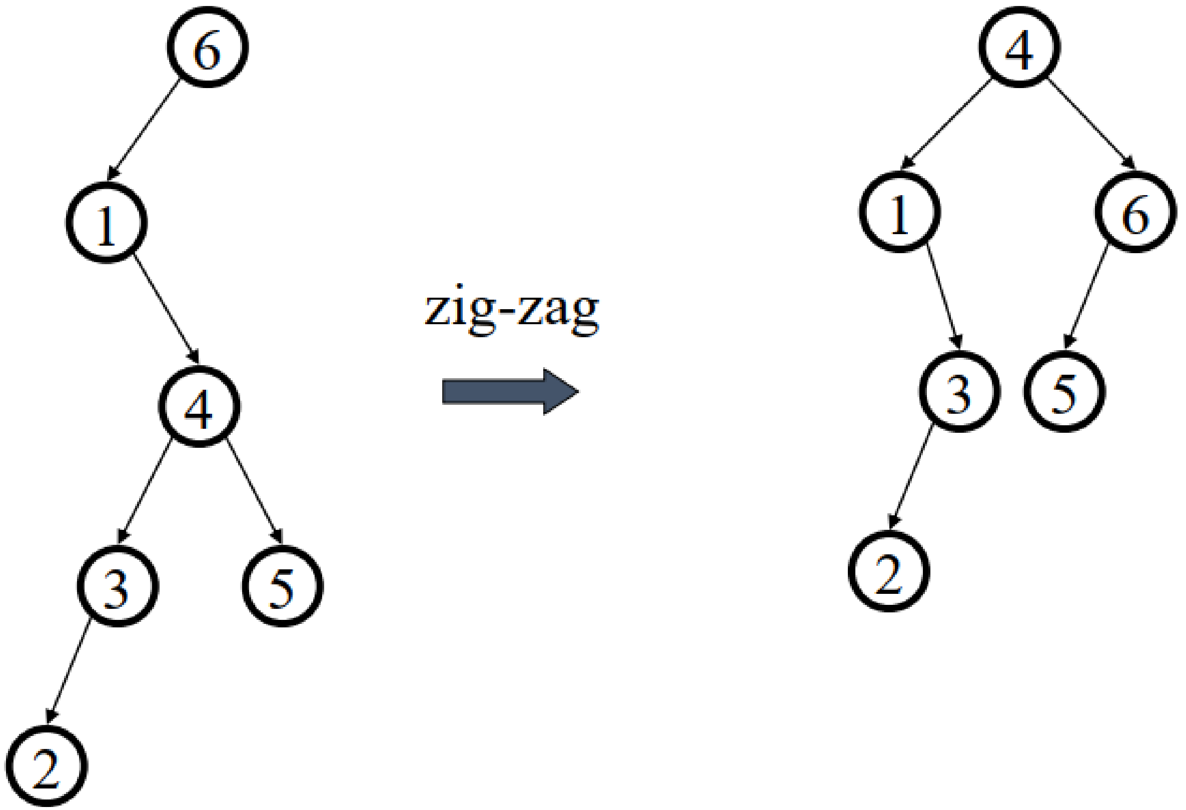
Find (4)

Splay it Again!



Splaying Example

… 4 splayed out!



Why Splaying Helps

- If a node on the access path is at depth d before the splay, it's final depth $\leq 3 + d/2$
  - Exceptions are the root, the child of the root, and the node splayed

- Overall, nodes which are below nodes on the access path tend to move closer to the root

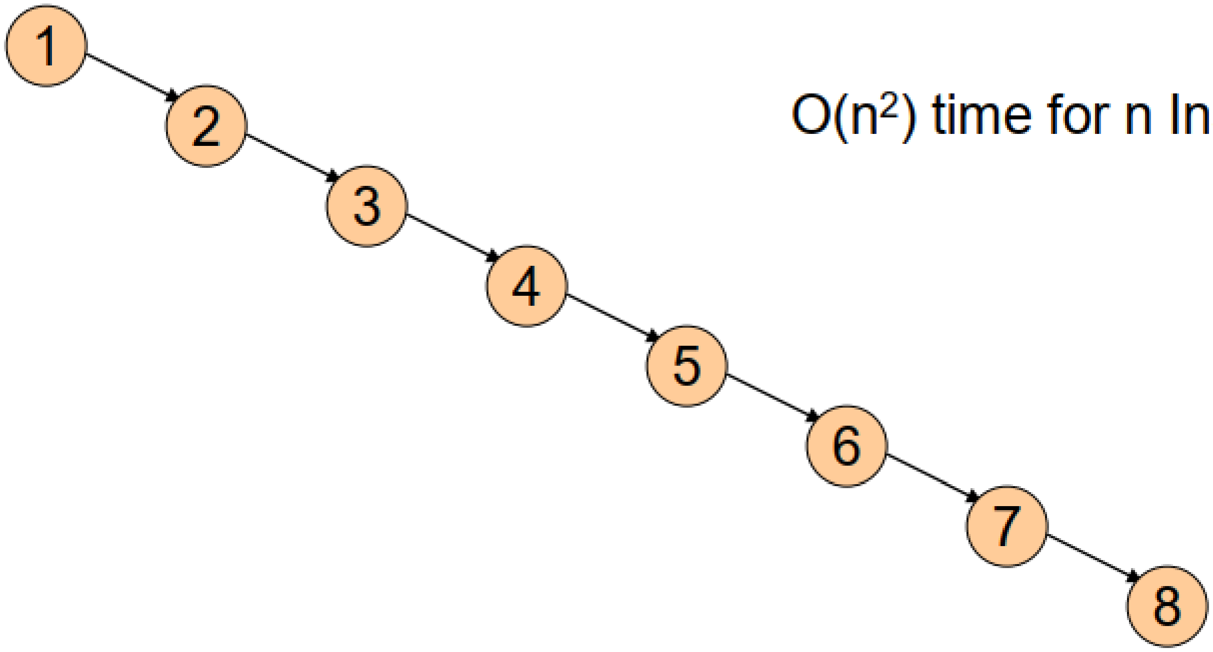
Splay Tree Insert and Delete

- Insert x
  - Insert x as normal then splay x to root.

- Delete x
  - Find x
  - Splay x to root and remove it
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.

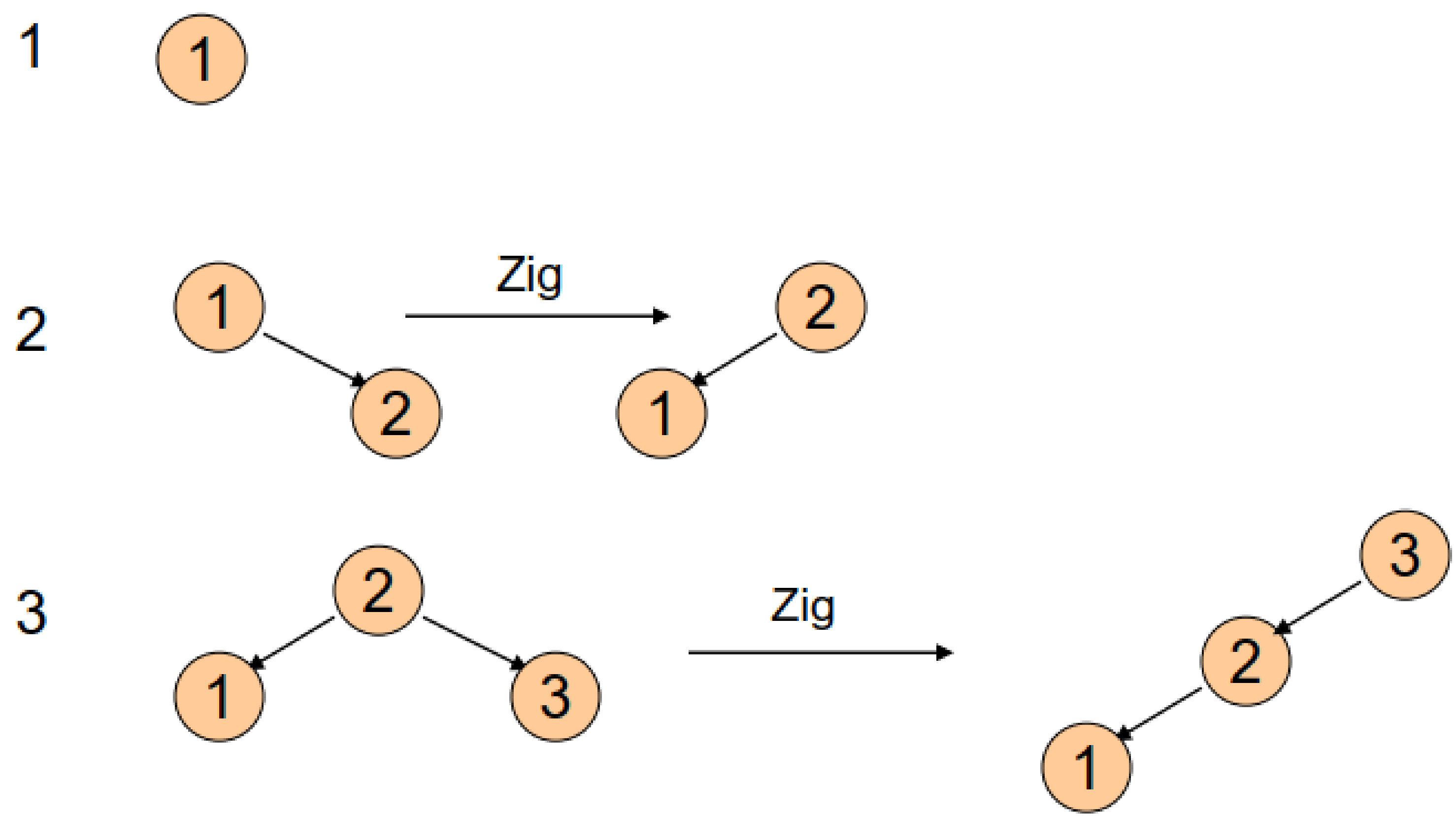
Example Insert

- Inserting in order 1, 2, 3, …, 8
- Without self-adjustment

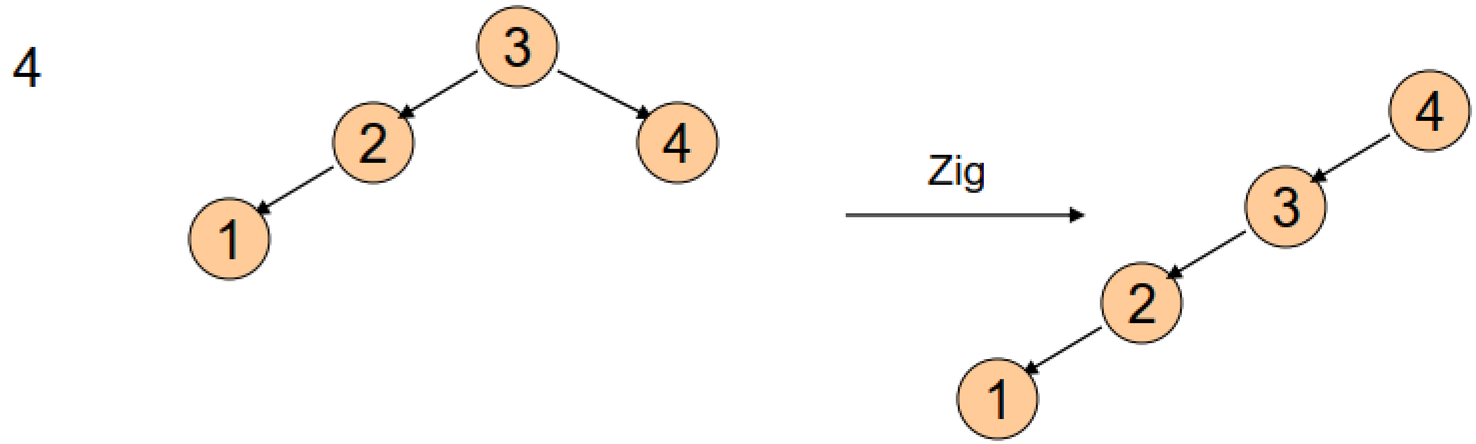


$O(n^2)$ time for n Insert

With Self-Adjustment

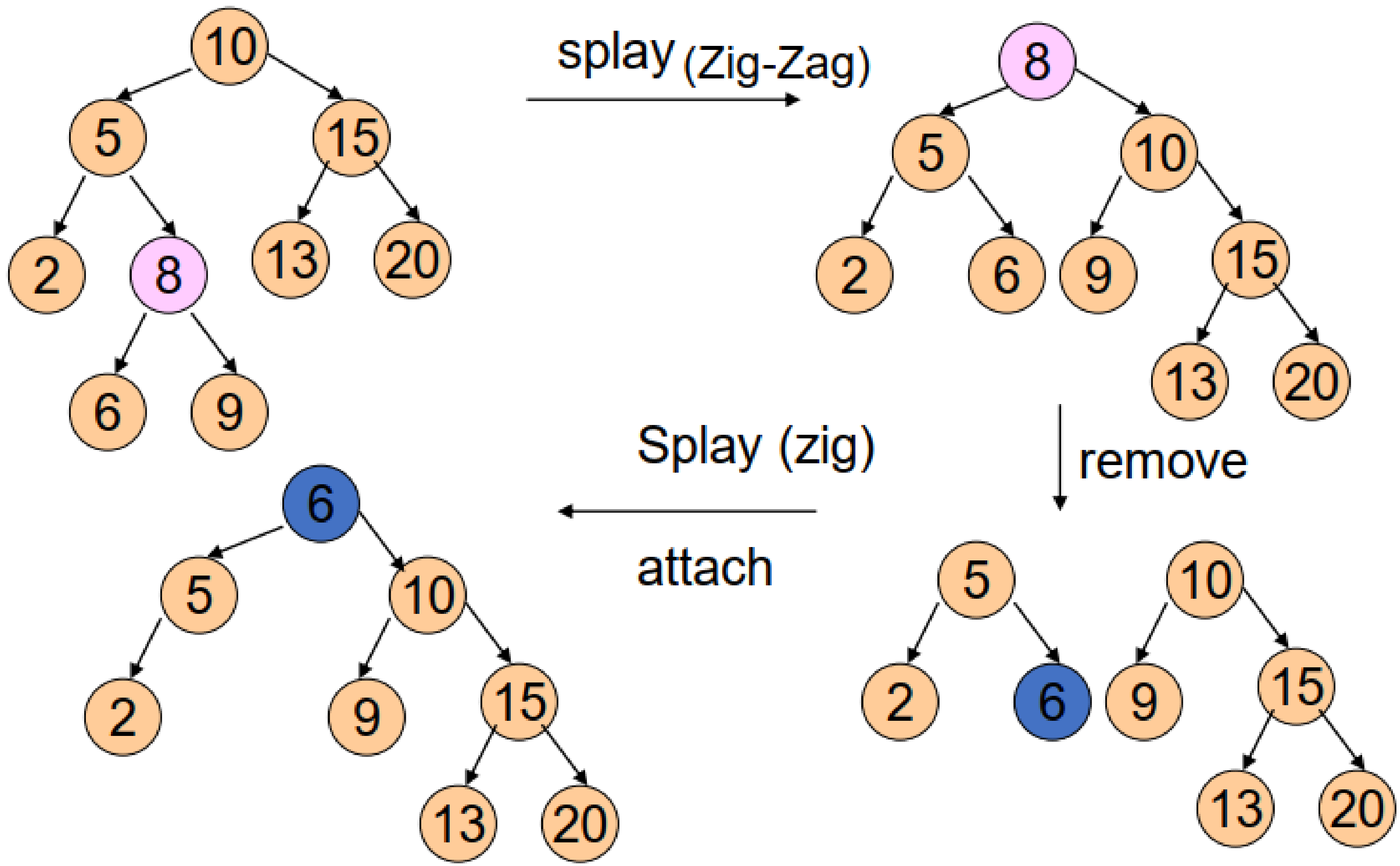


With Self-Adjustment



Each Insert takes O(1) time therefore O(n) time for n Insert!!

Example Deletion



Next Class

Priority Queues

Reading: Weiss, chap. 6