University of North Texas

CSCE 2110 Foundations of Data Structures

Splay Tree

Contents

- Splay tree
  - insertion
  - find
  - deletion

Self adjusting Trees

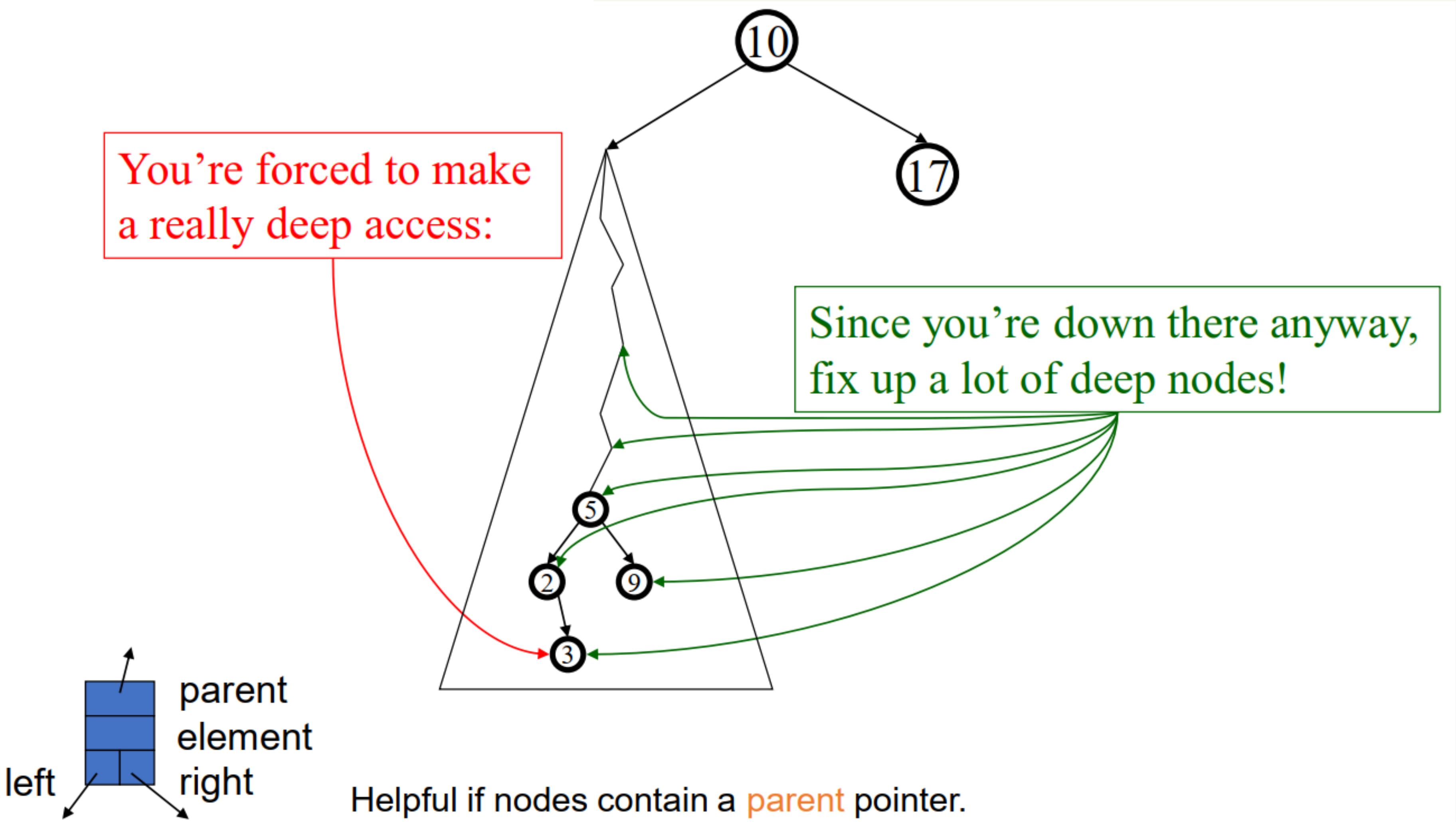
- Ordinary binary search trees have no balance conditions
  - What you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - Tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed
  - 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

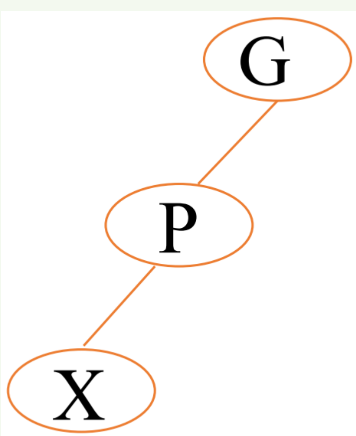


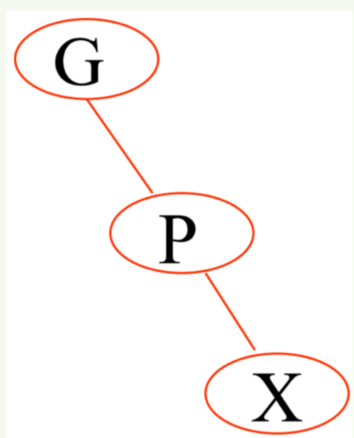
Helpful if nodes contain a parent pointer.

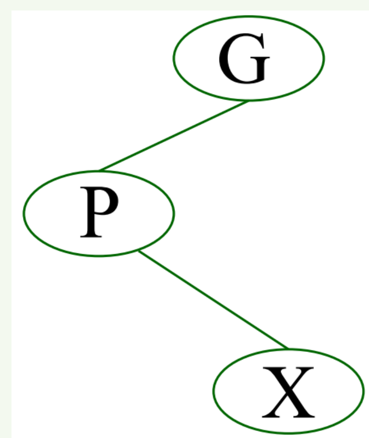
Splaying Cases

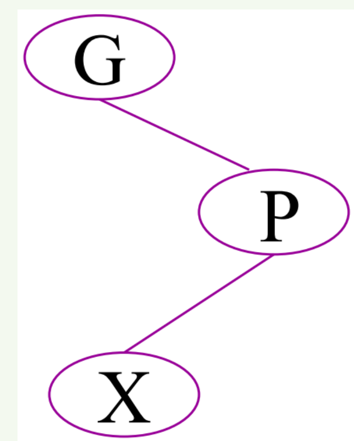
Node being accessed (x) is:

- Root (no rotation)
- Child of root (single rotation)
- Has both parent (p) and grandparent (g)
  - Zig-zig pattern: $g \rightarrow p \rightarrow x$ is left-left or right-right (double rotations)
  - Zig-zag pattern: $g \rightarrow p \rightarrow x$ is left-right or right-left (double rotations)



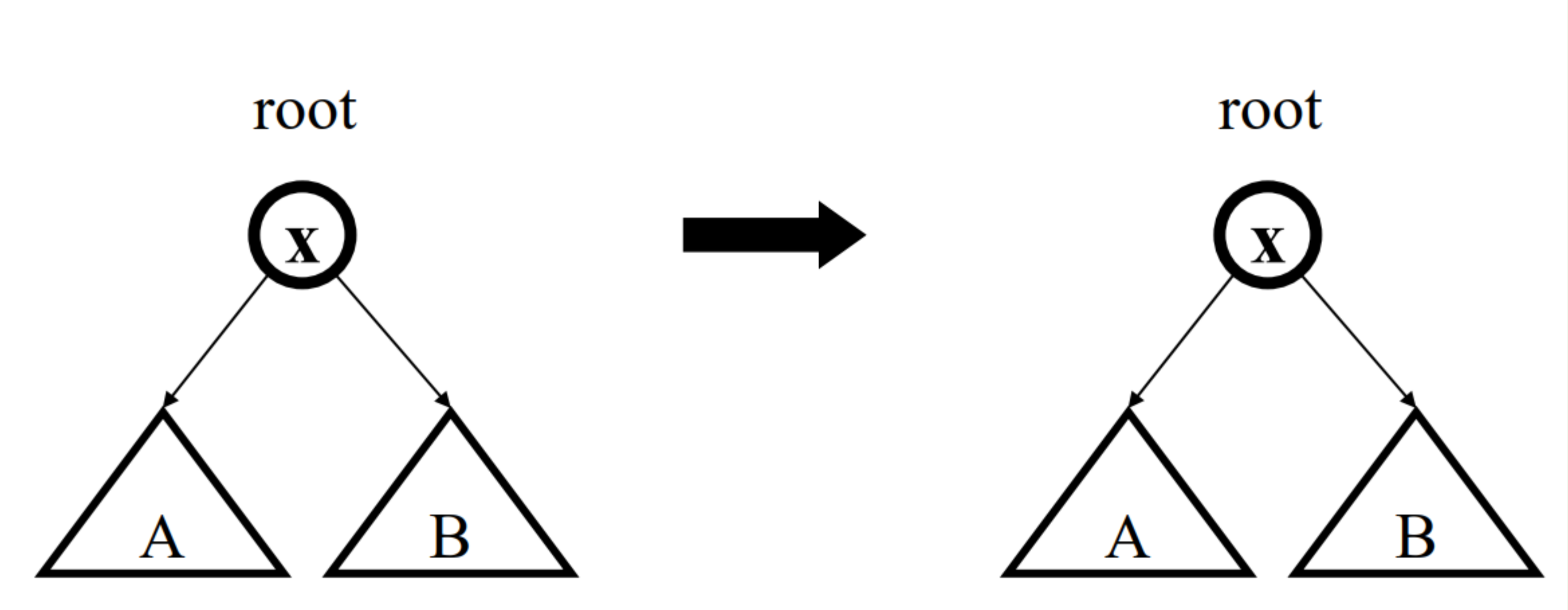






Access Root

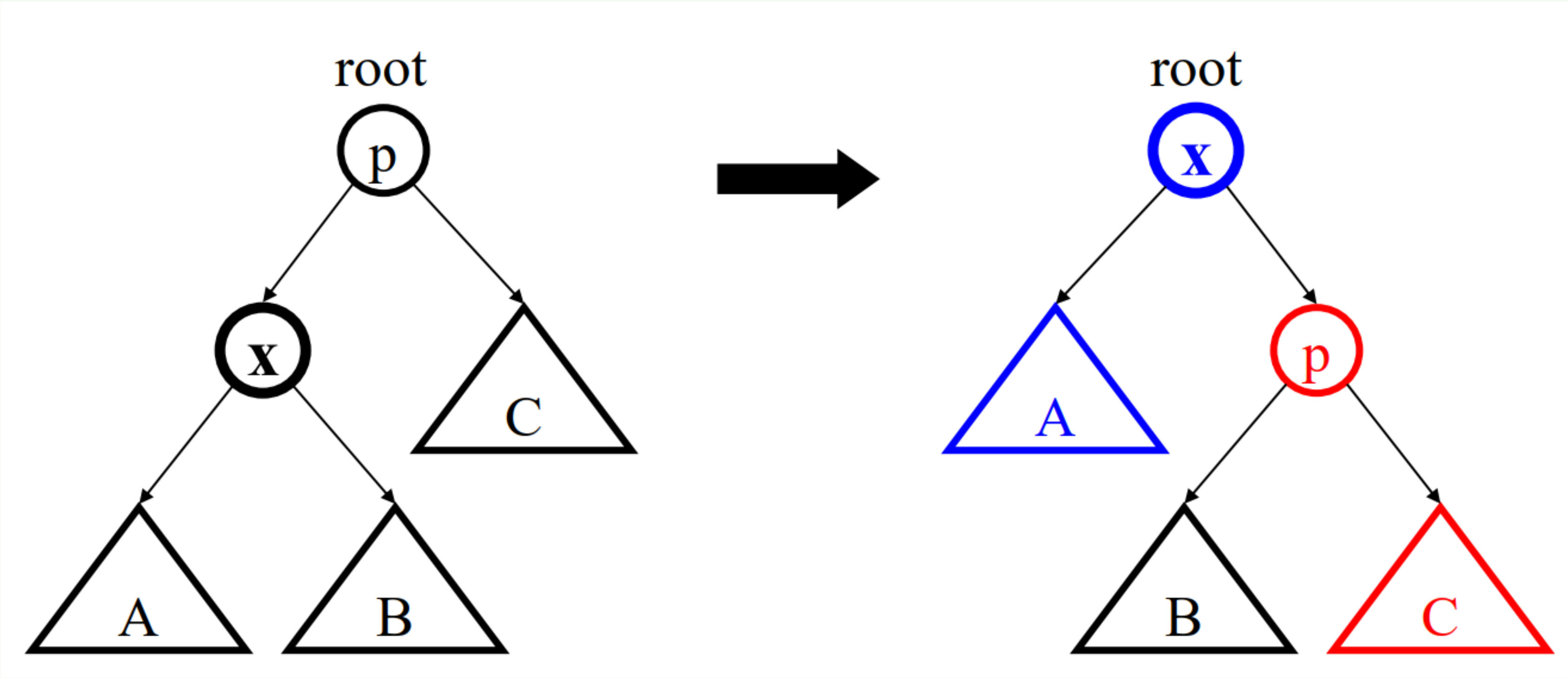
Do nothing (that was easy!)



Access Child of Root

Zig (AVL single rotation)

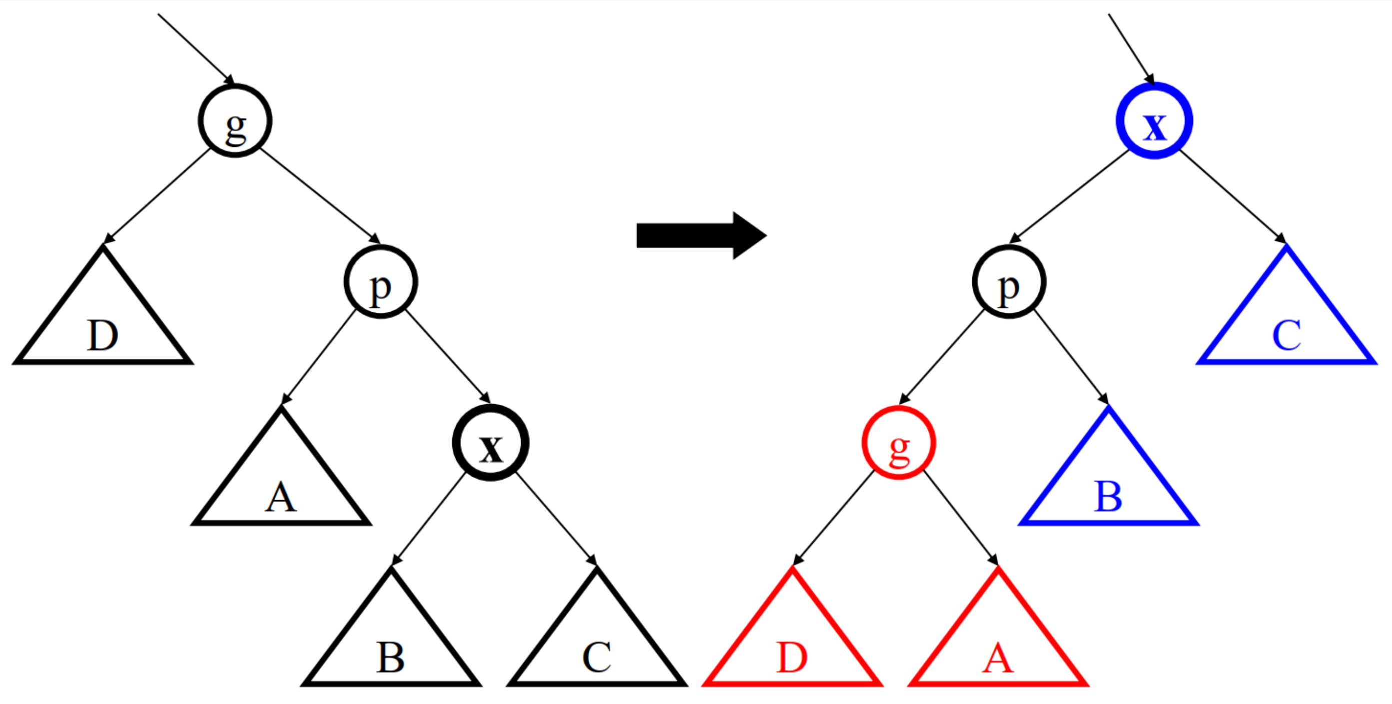
- If x is the right child: single left rotation on root node
- If x is the left child: single right rotation on root node



Access (LL, RR) Grandchild

Zig-Zig

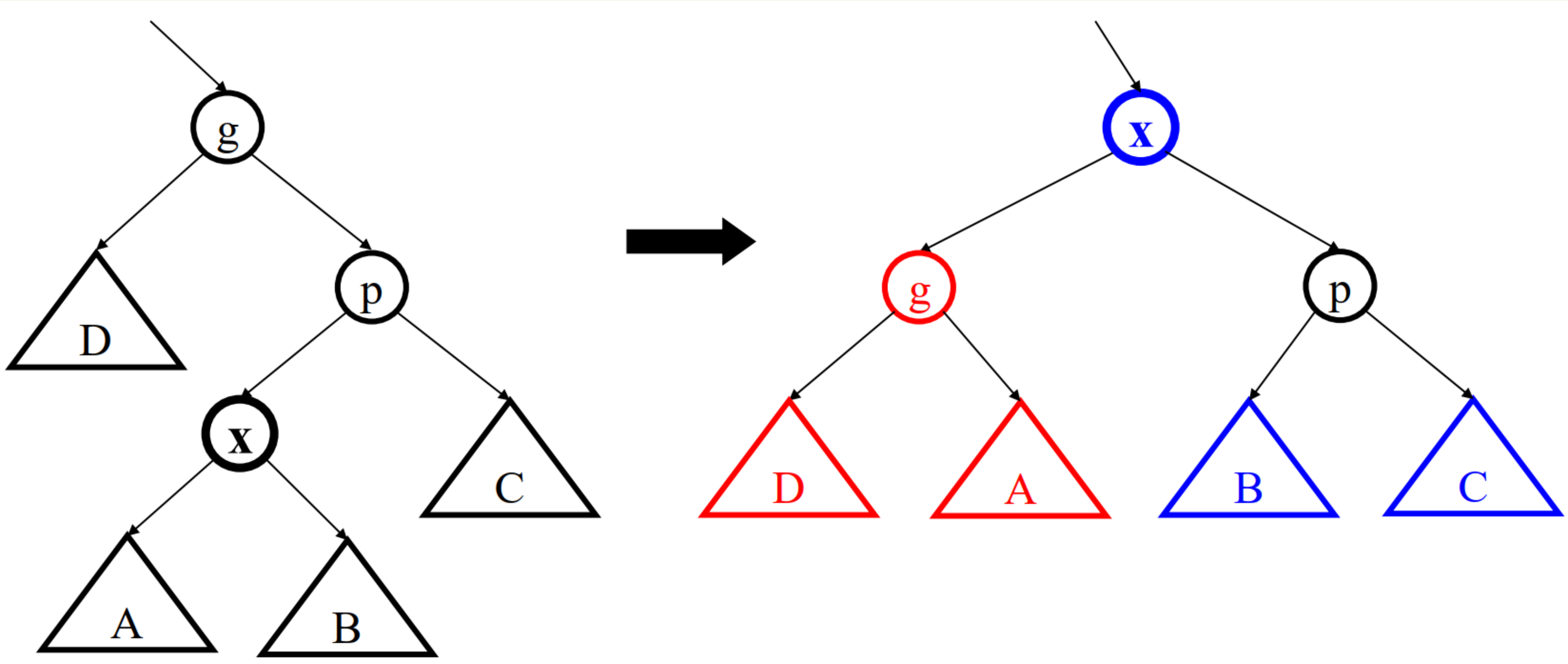
- Lef—left: two right rotations: first right rotation on **grandparent node** g, then right rotation on **parent node** p
- Right-right: two left rotations: first left rotation on **grandparent node** g, then left rotation on **parent node** p



Access (LR, RL) Grandchild

Zig-Zag

- Left-right: first left rotation on **parent node** p, then right rotation on (original) **grandparent node** g
- Right-left: first right rotation on **parent node** p, then left rotation on (original) **grandparent node** g

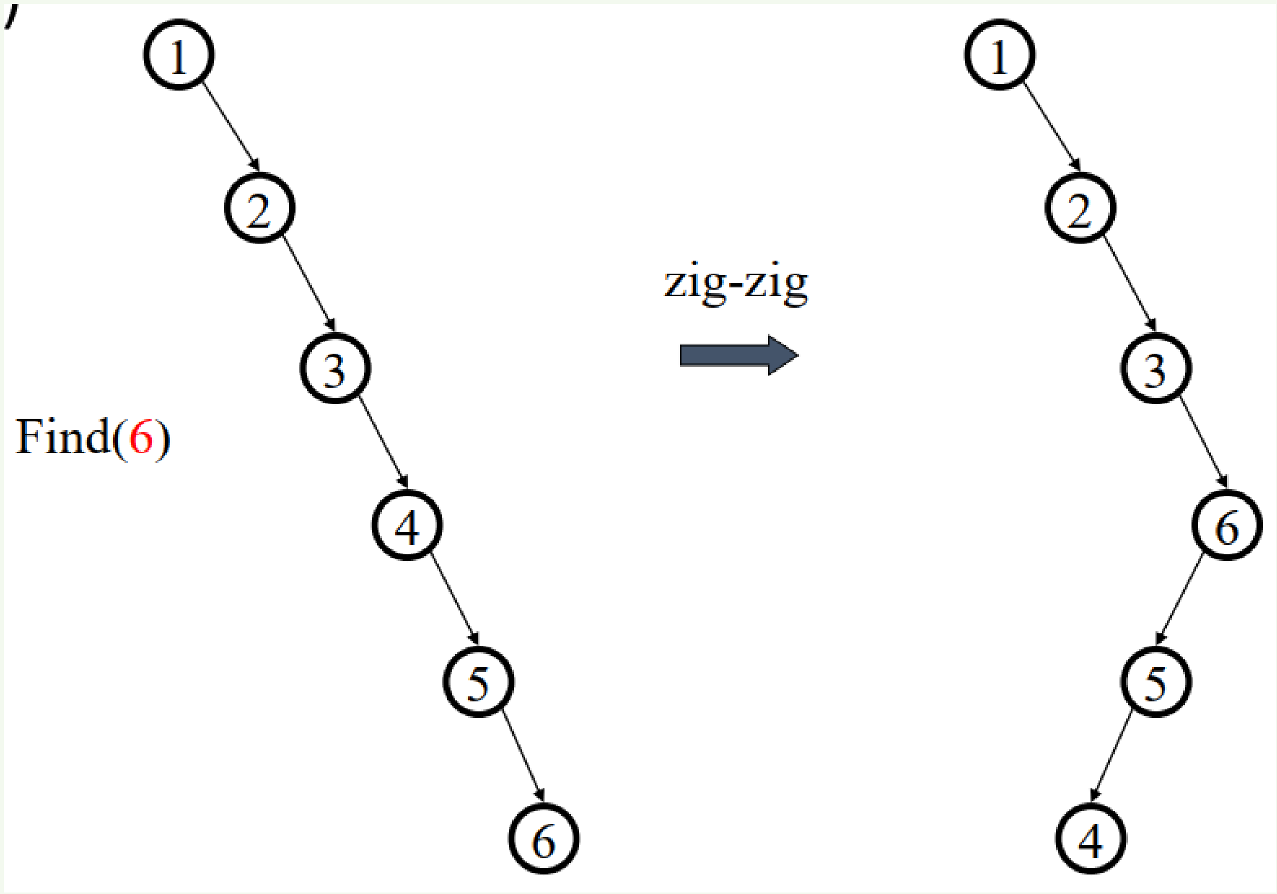


Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root

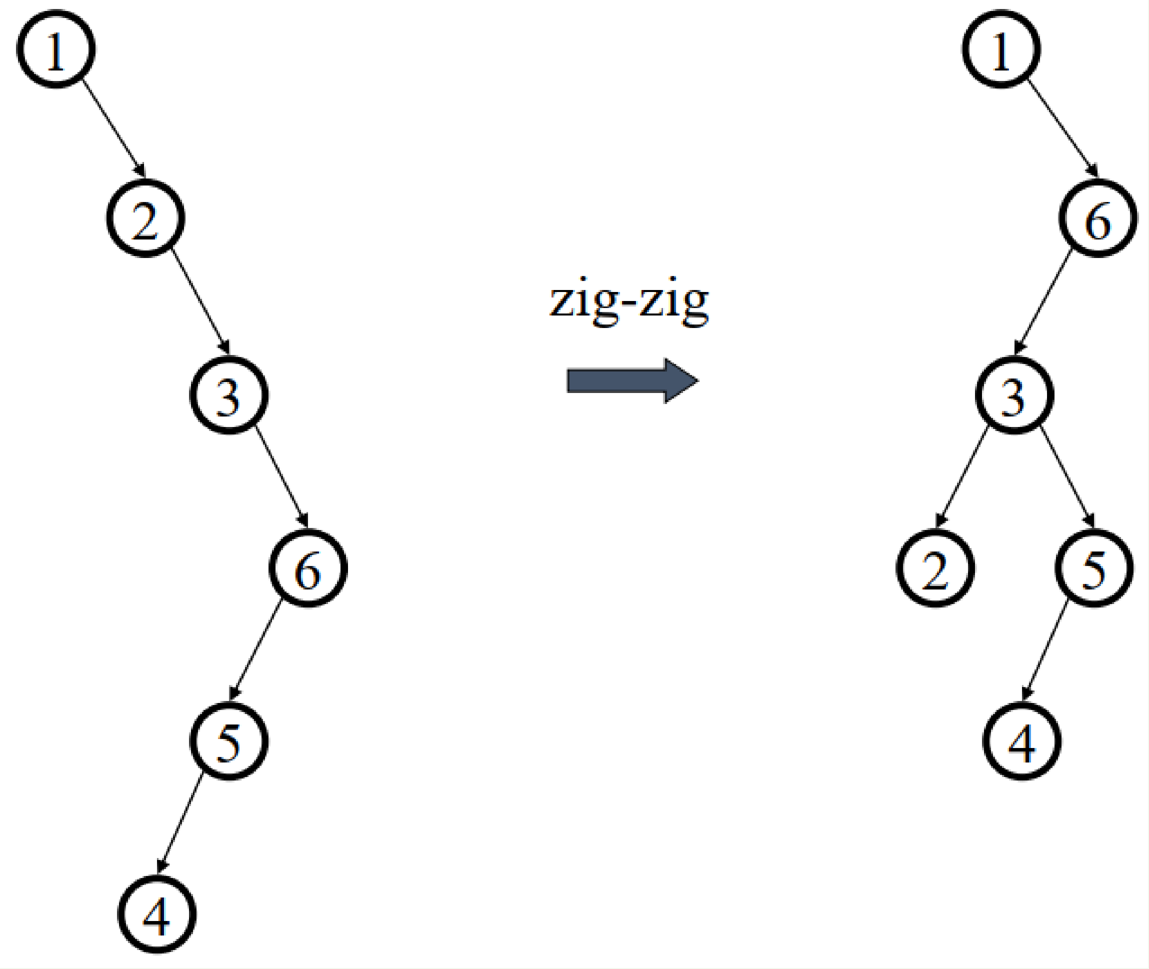
Splaying Example

Find(6)



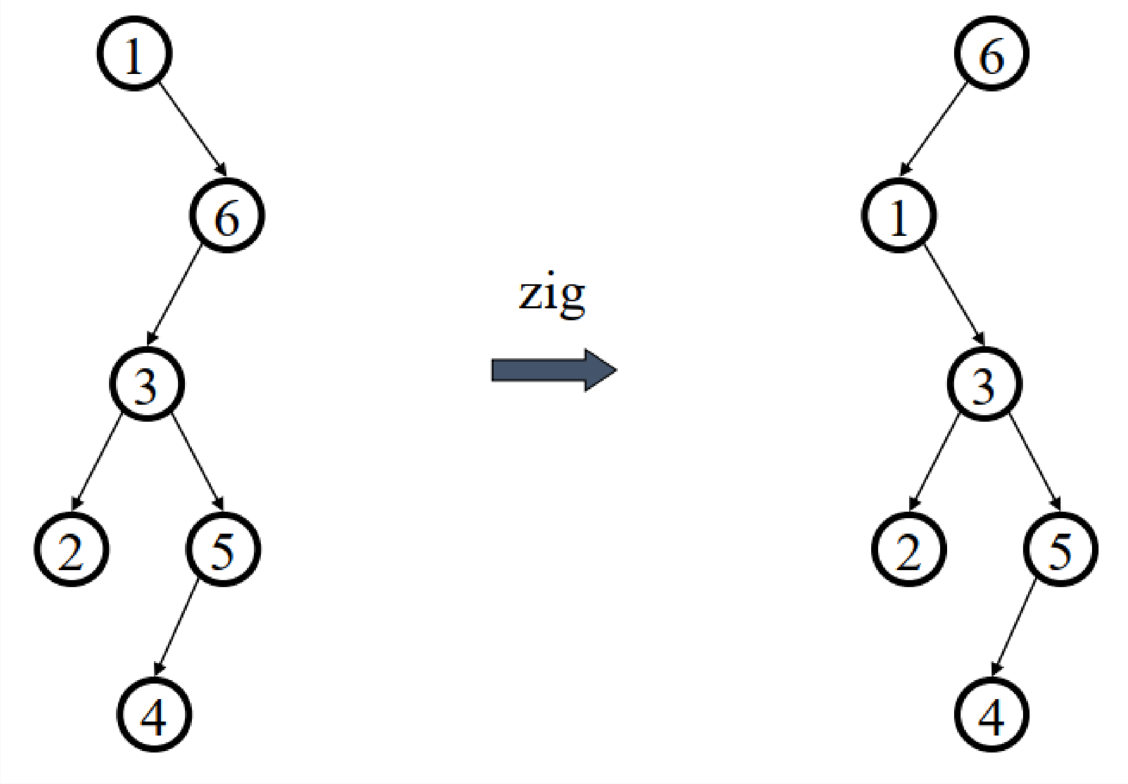
Splaying Example

… still splaying …



Splaying Example

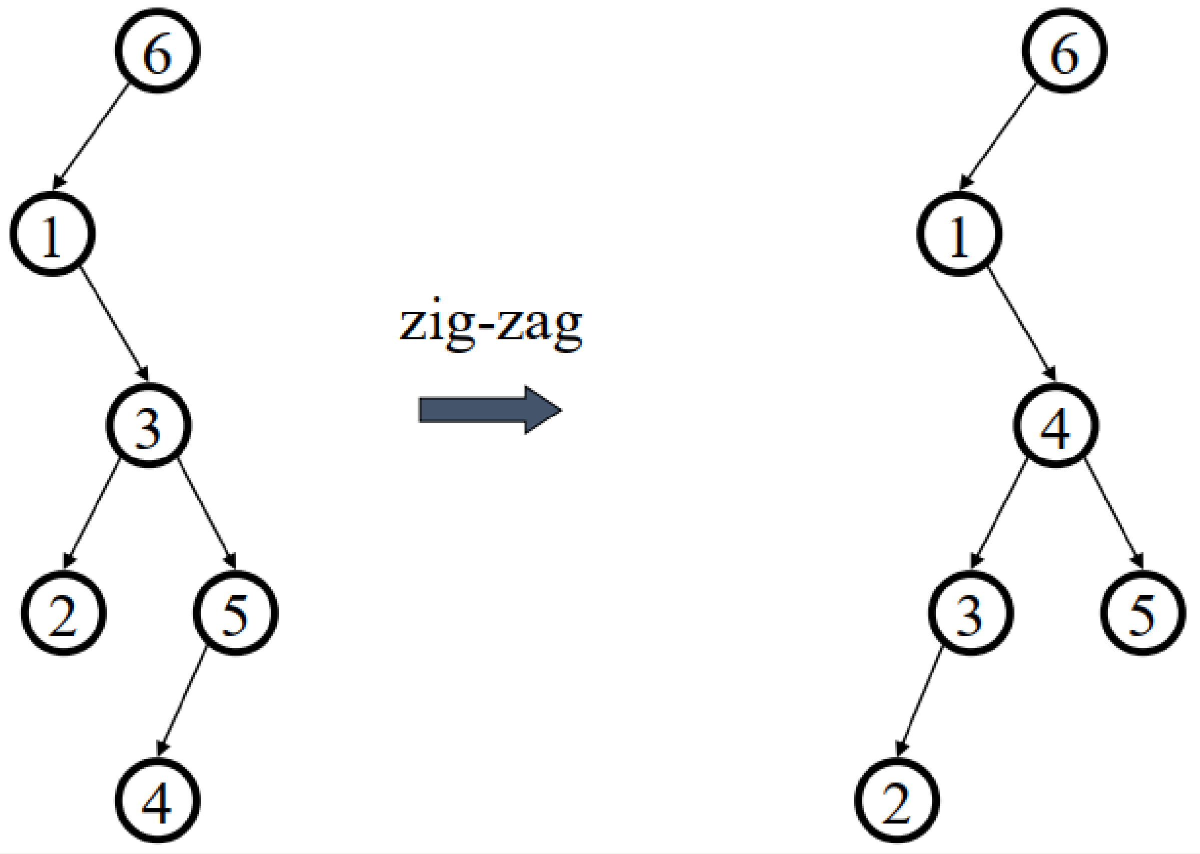
… 6 splayed out!



Splaying Example

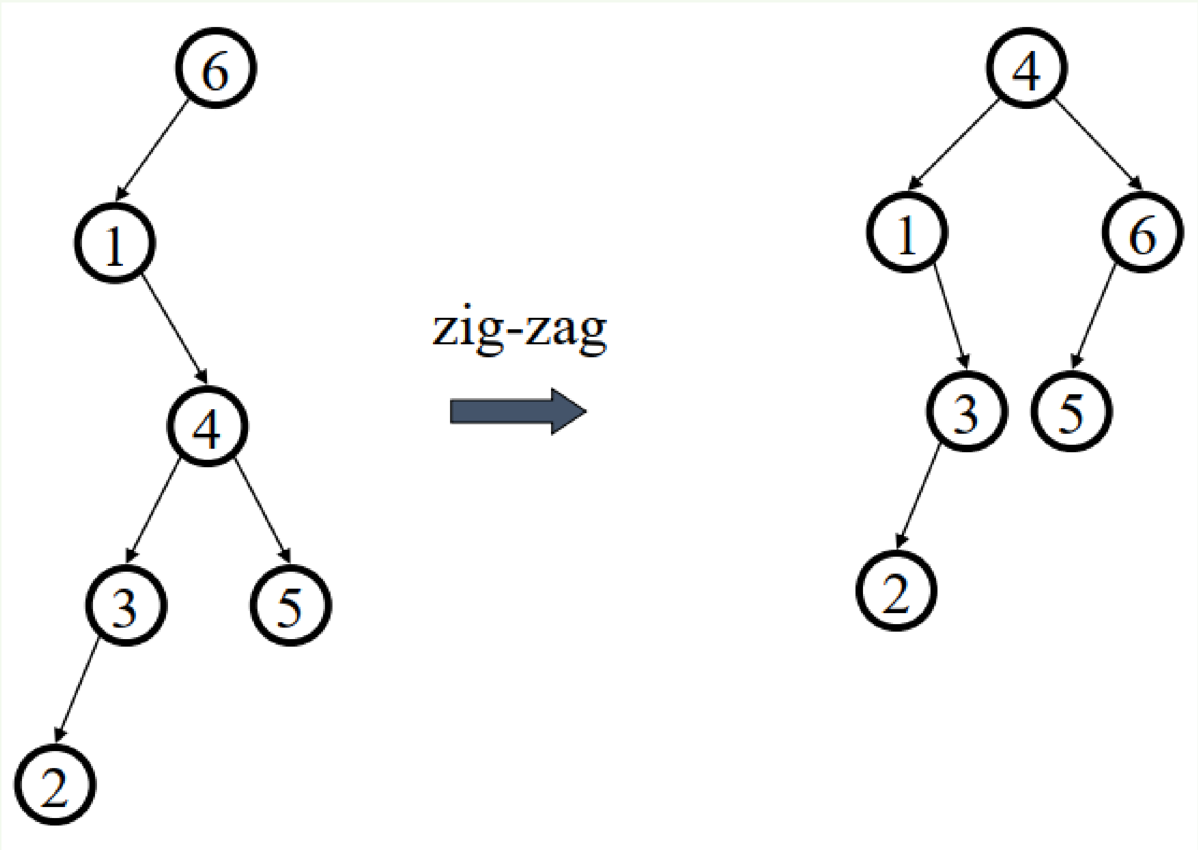
Find (4)

Splay it Again!



Splaying Example

… 4 splayed out!



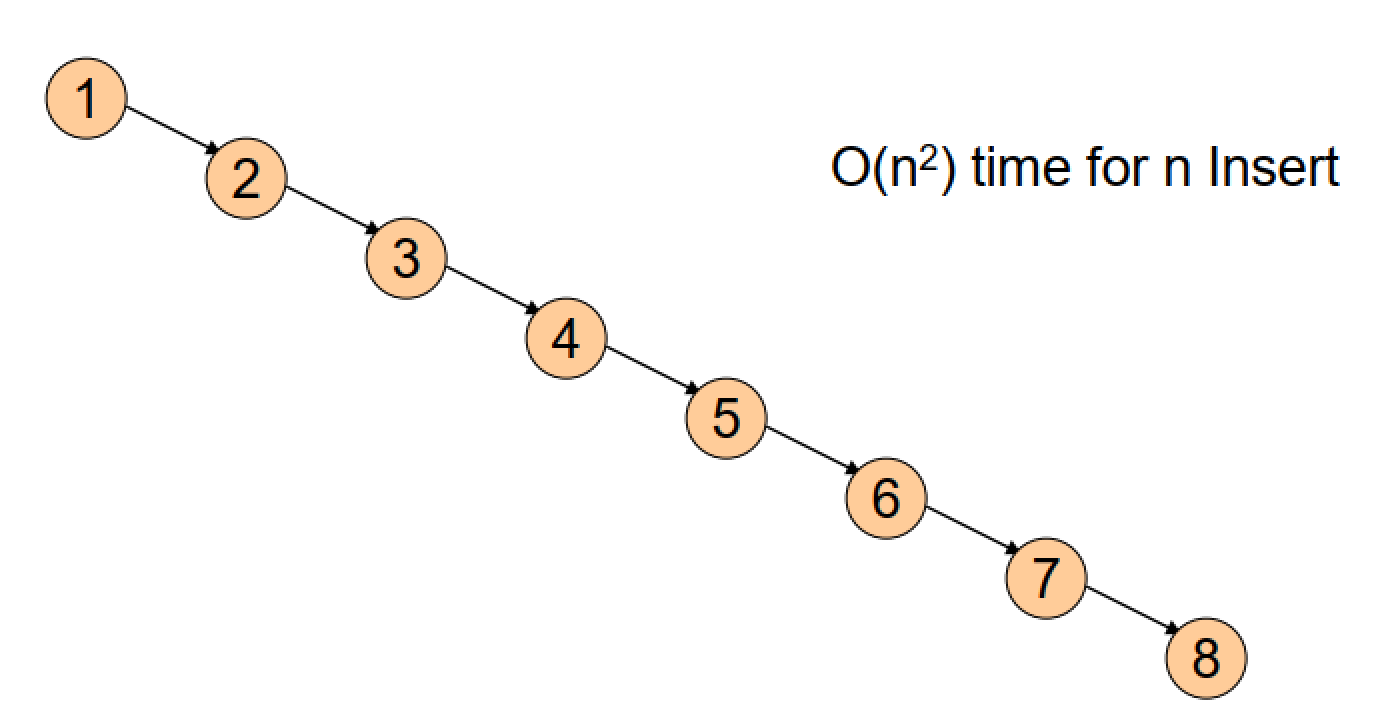
Splay Tree Insert and Delete

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

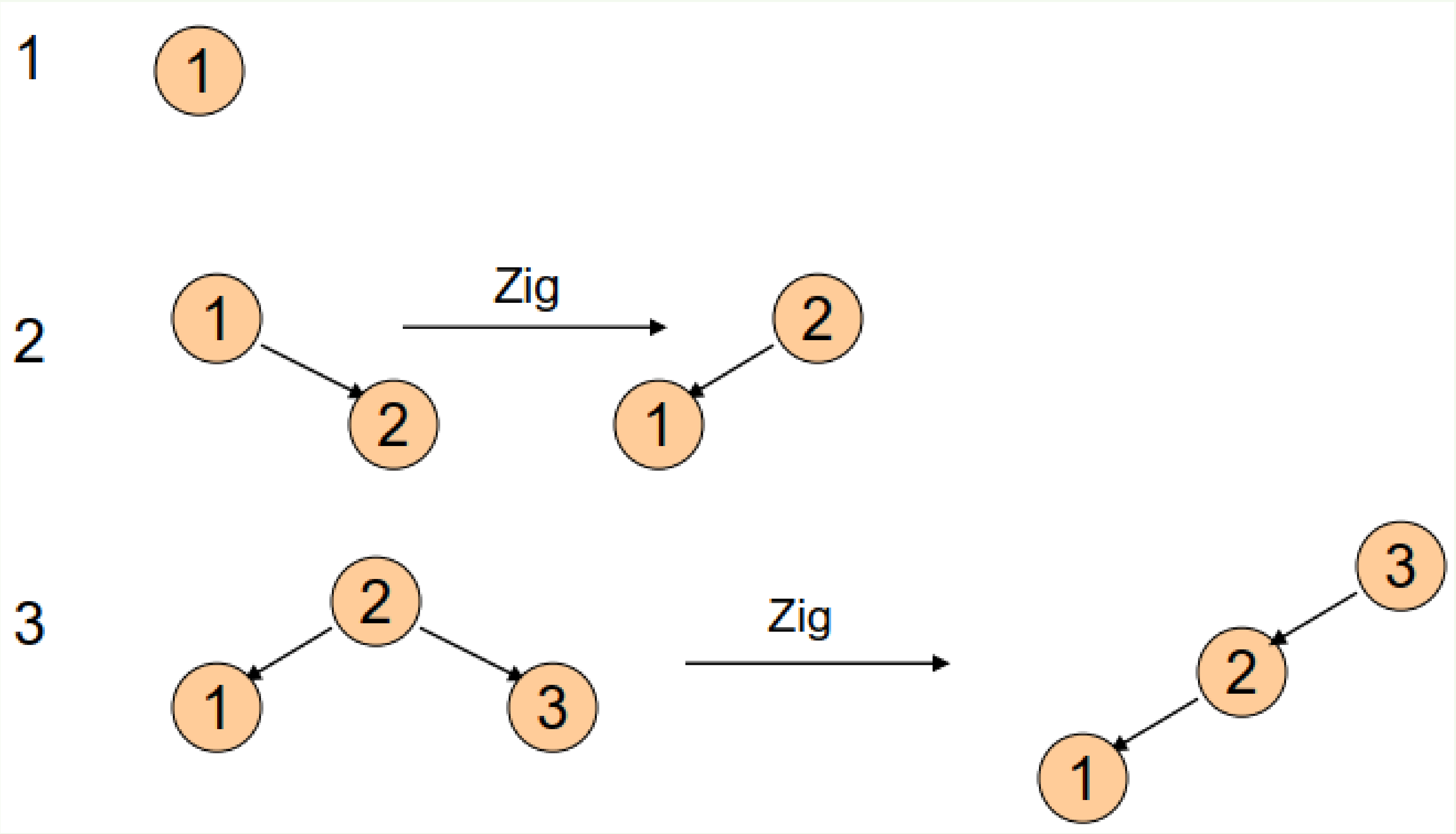
- Delete x
  - Find x
  - Splay x to root and remove it
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.

Example Insert

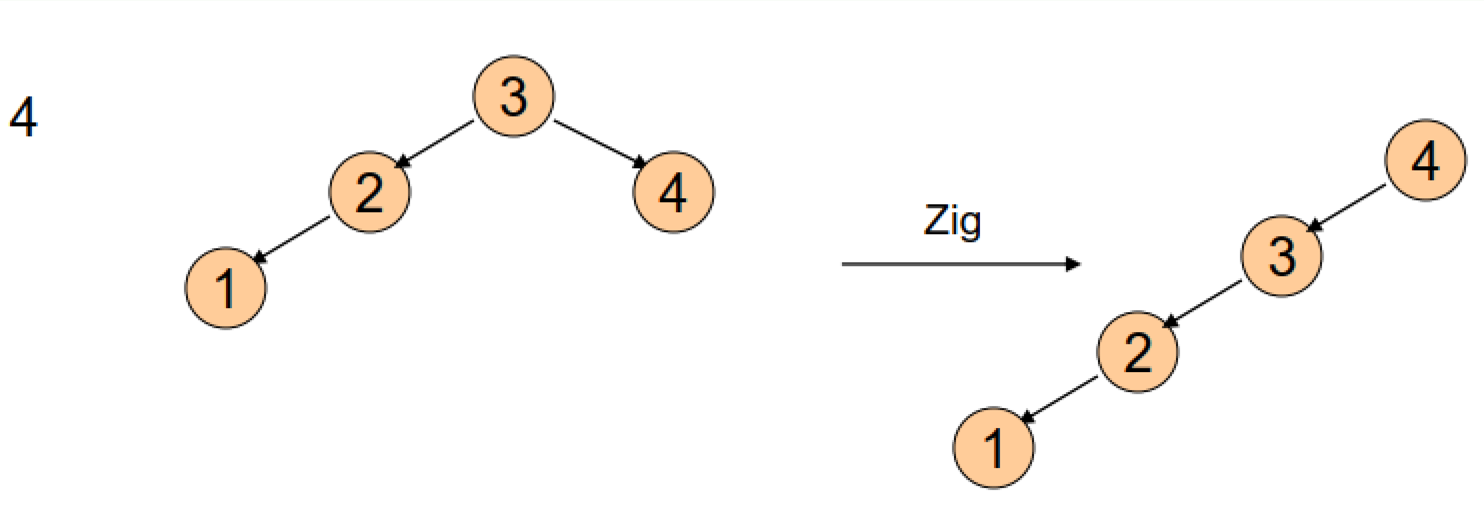
- Inserting in order 1, 2, 3, …, 8
- Without self-adjustment



With Self-Adjustment

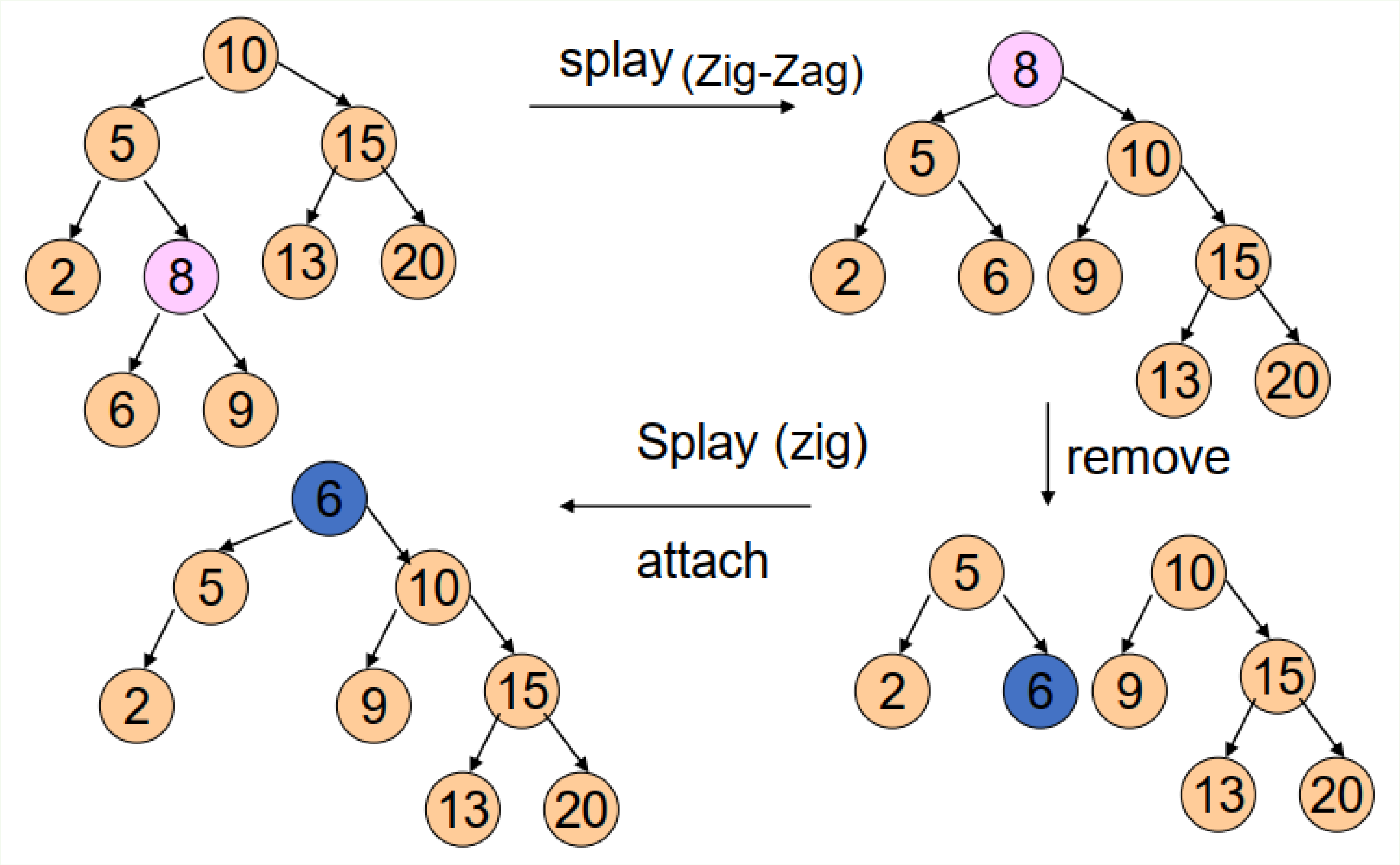


With Self-Adjustment



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

Example Deletion



Summary of Search Trees

- Problem with Binary Search Trees: Must keep tree balanced to allow fast access to stored items
- AVL trees: Insert/Delete operations keep tree balanced
- Splay trees: Repeated Find operations produce balanced trees
- Splay trees are very effective search trees
  - relatively simple: no extra fields required
  - excellent locality properties:
    - frequently accessed keys are cheap to find (near top of tree)
    - infrequently accessed keys stay out of the way (near bottom of tree)