# CSCE 2110
# Foundations of Data Structures

Splay Tree

# 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

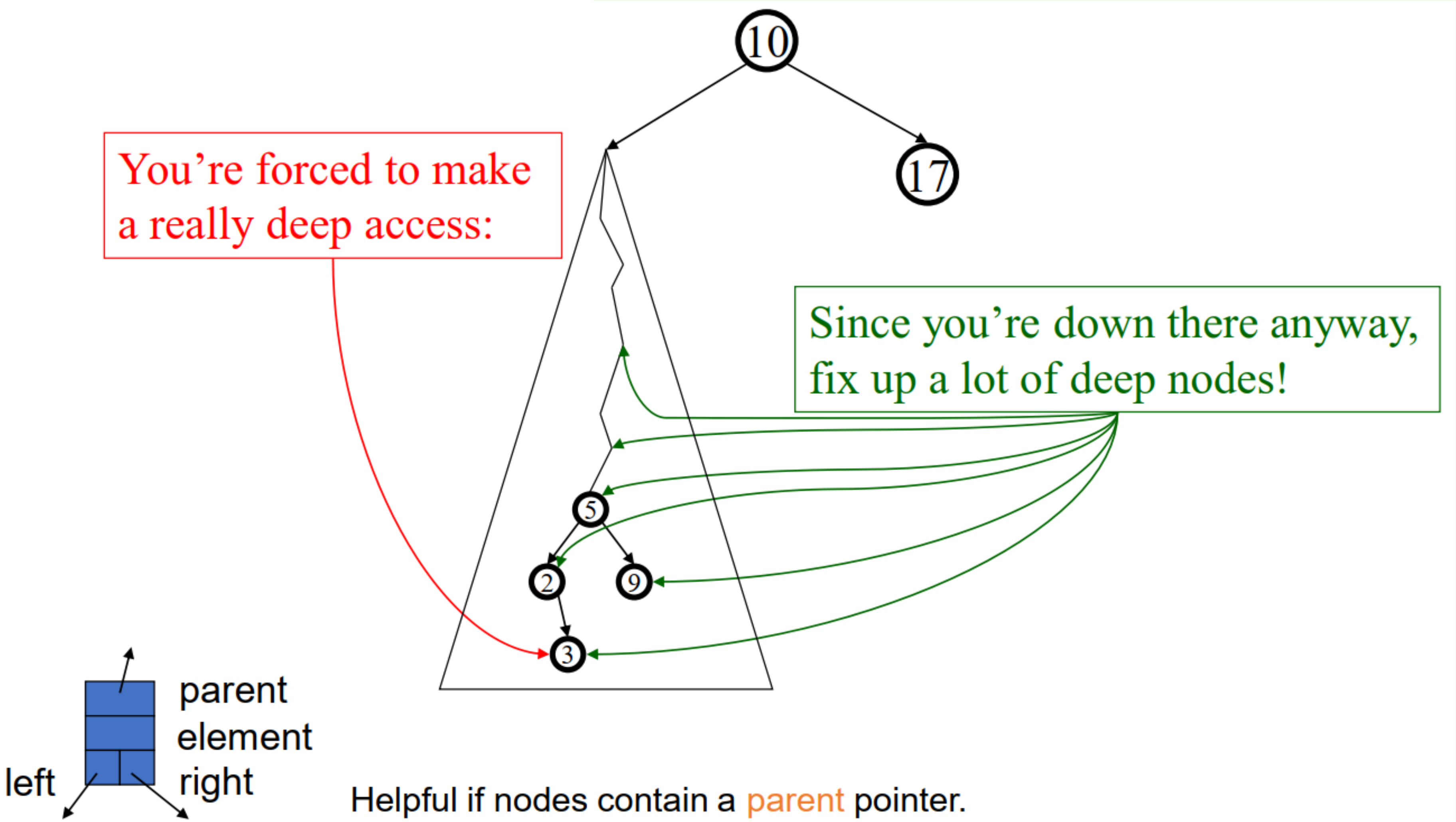
- Self-adjusting trees get reorganized over time as nodes are accessed
  - Tree adjusts after insert, delete, or find

# 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

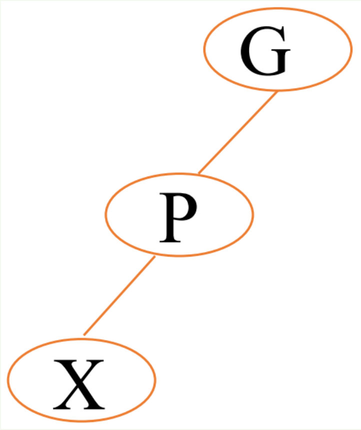


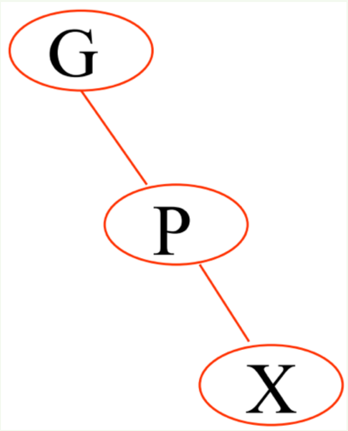

Helpful if nodes contain a parent pointer.

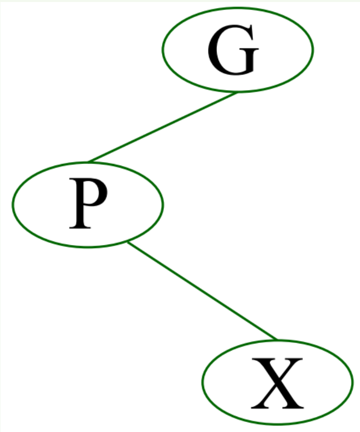
# Splaying Cases

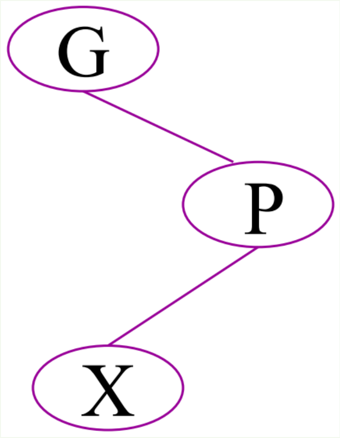
Node being accessed (x) is:

- Root
- 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



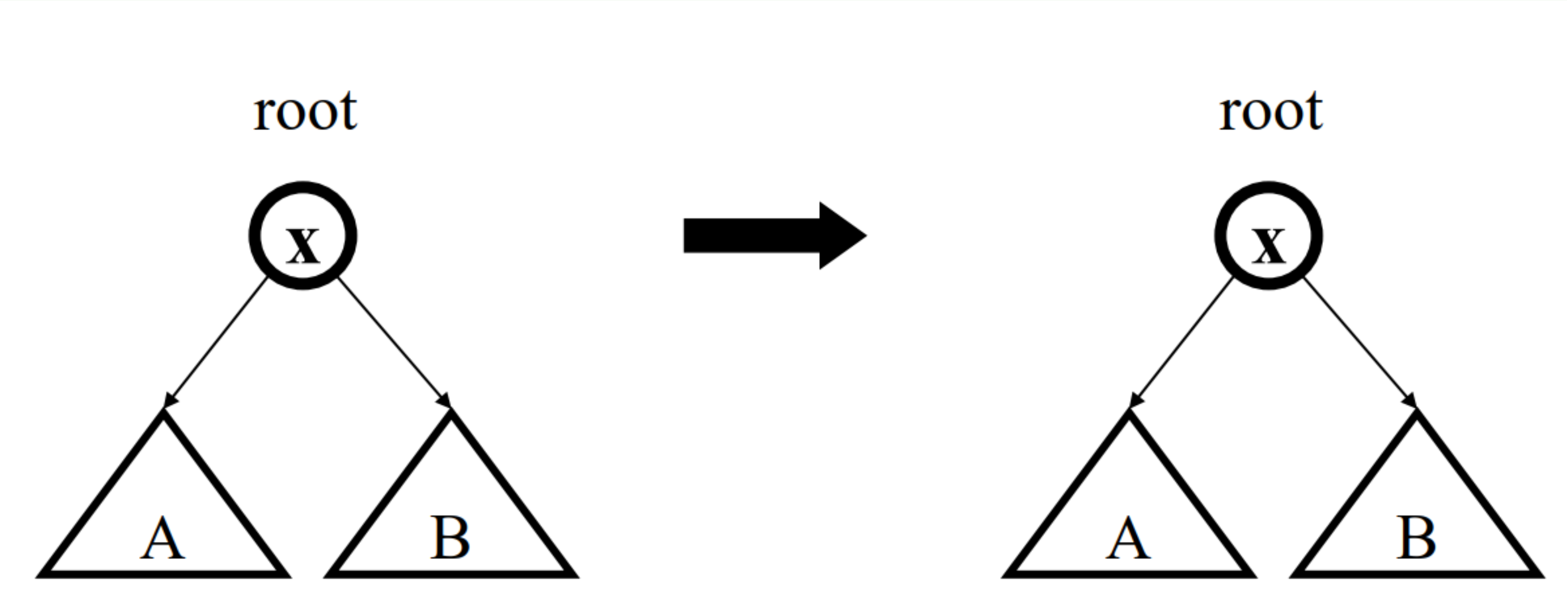






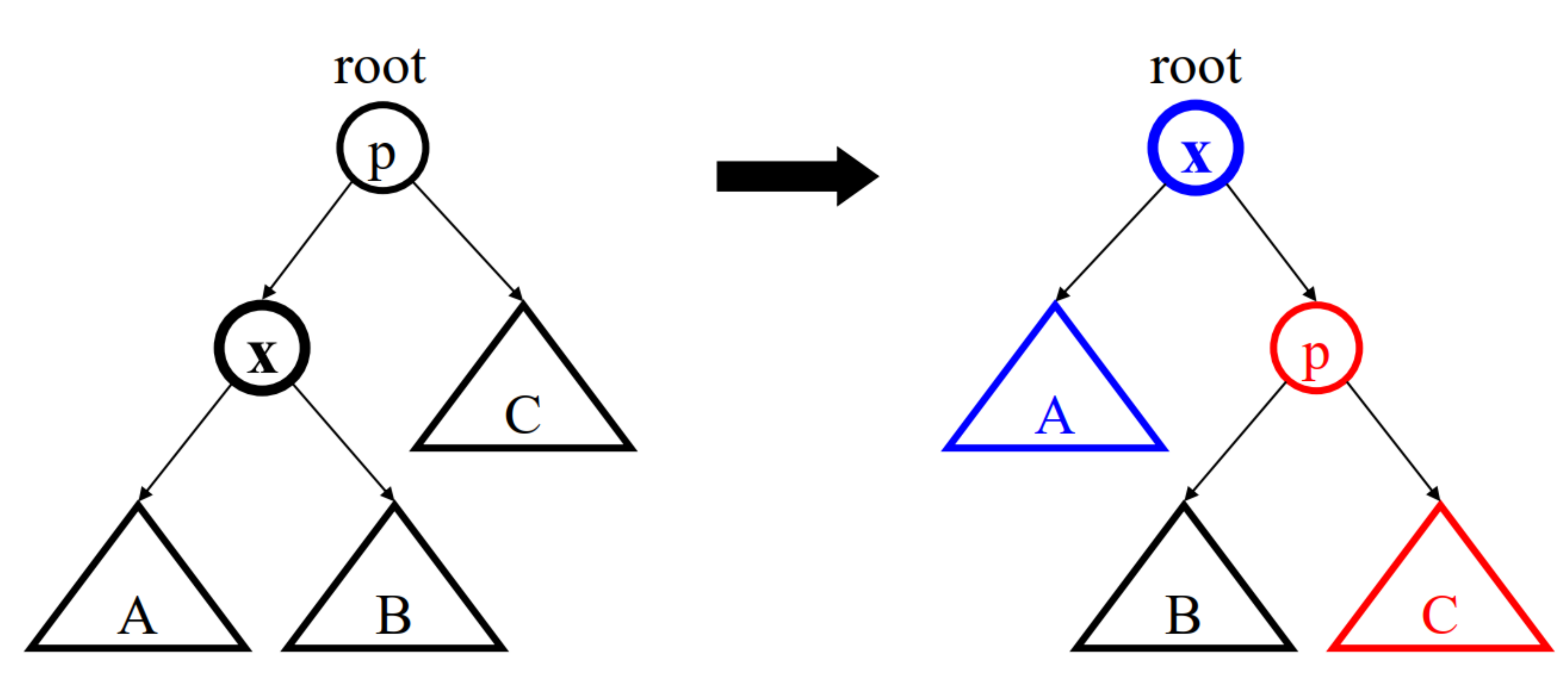
# Access Root

Do nothing (that was easy!)



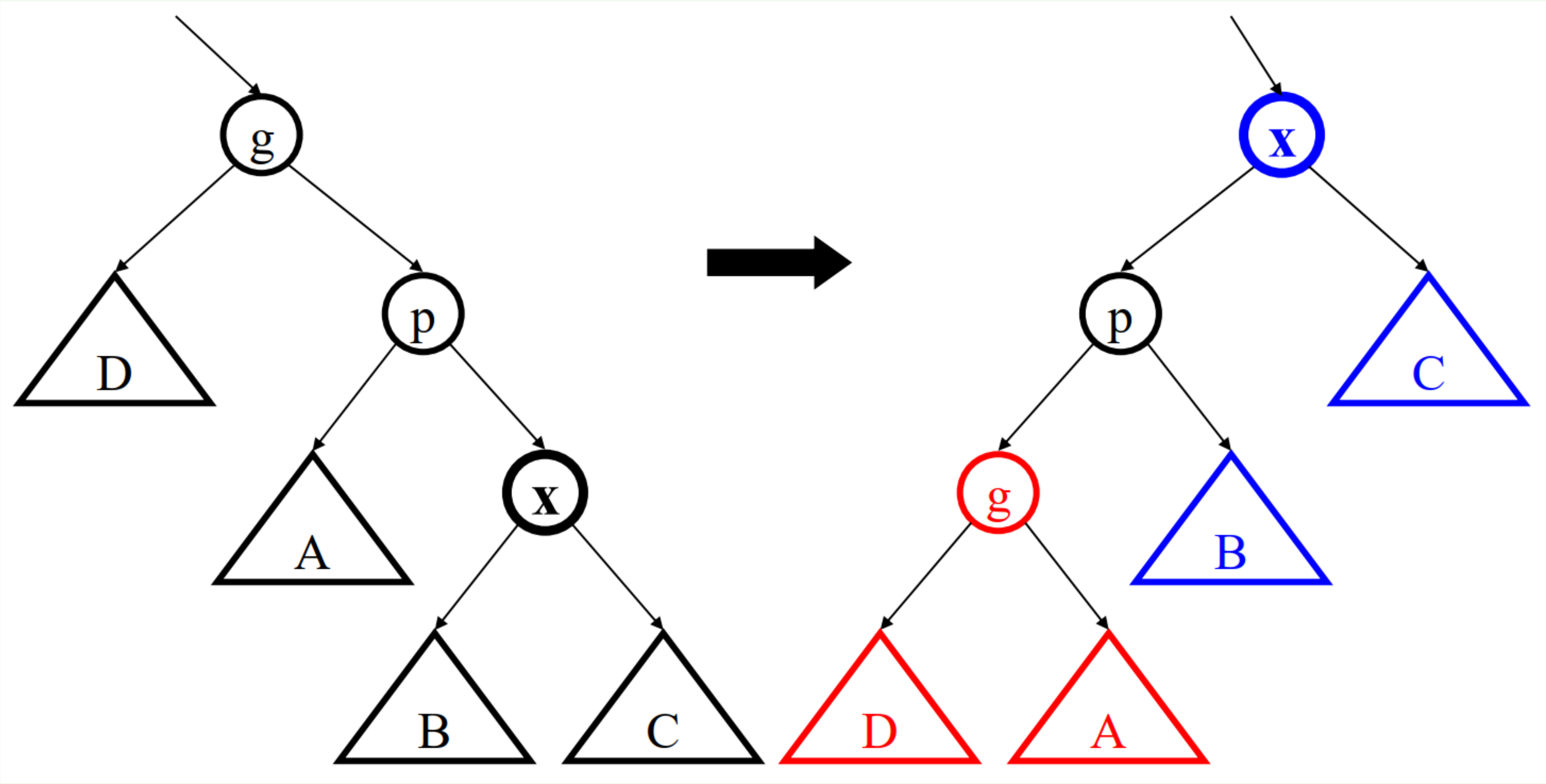
# Access Child of Root

Zig (AVL single rotation)



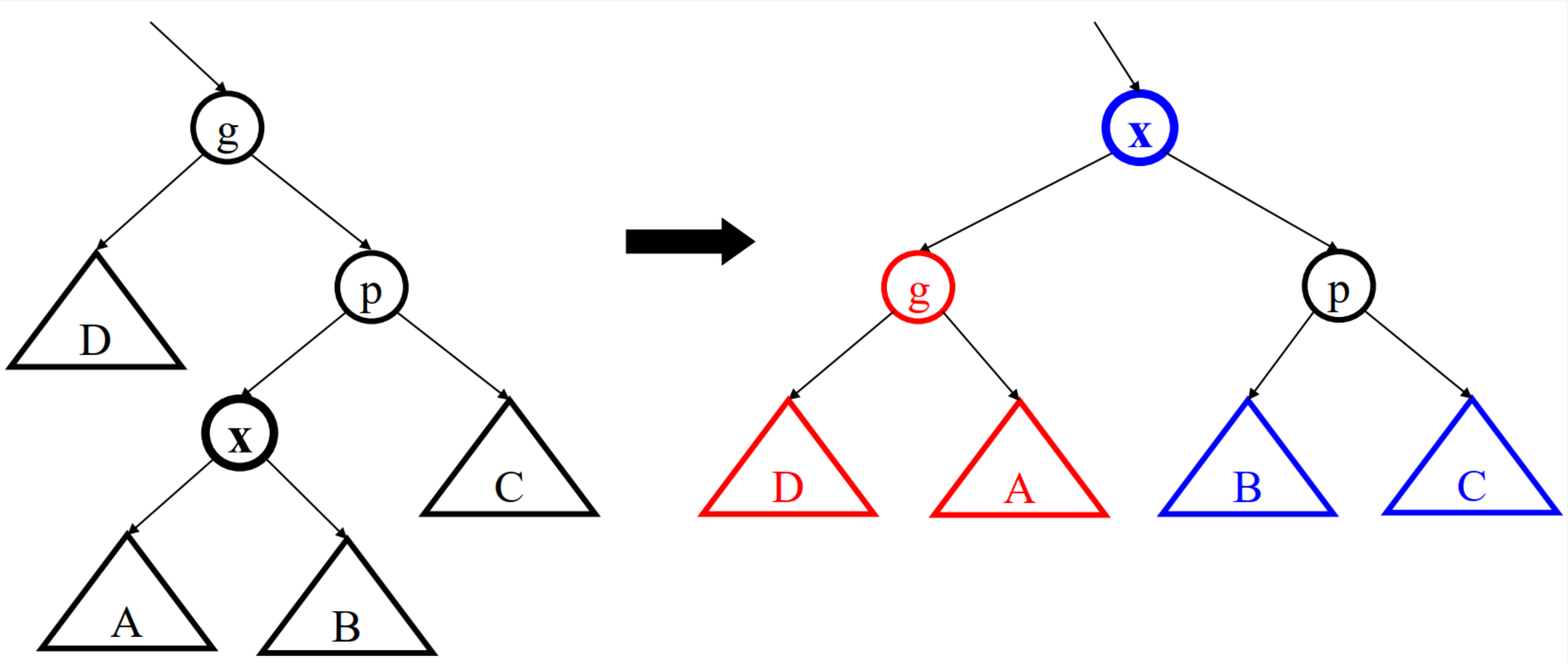
# Access (LL, RR) Grandchild

Zig-Zag



# Access (LR, RL) Grandchild

Zig-Zag

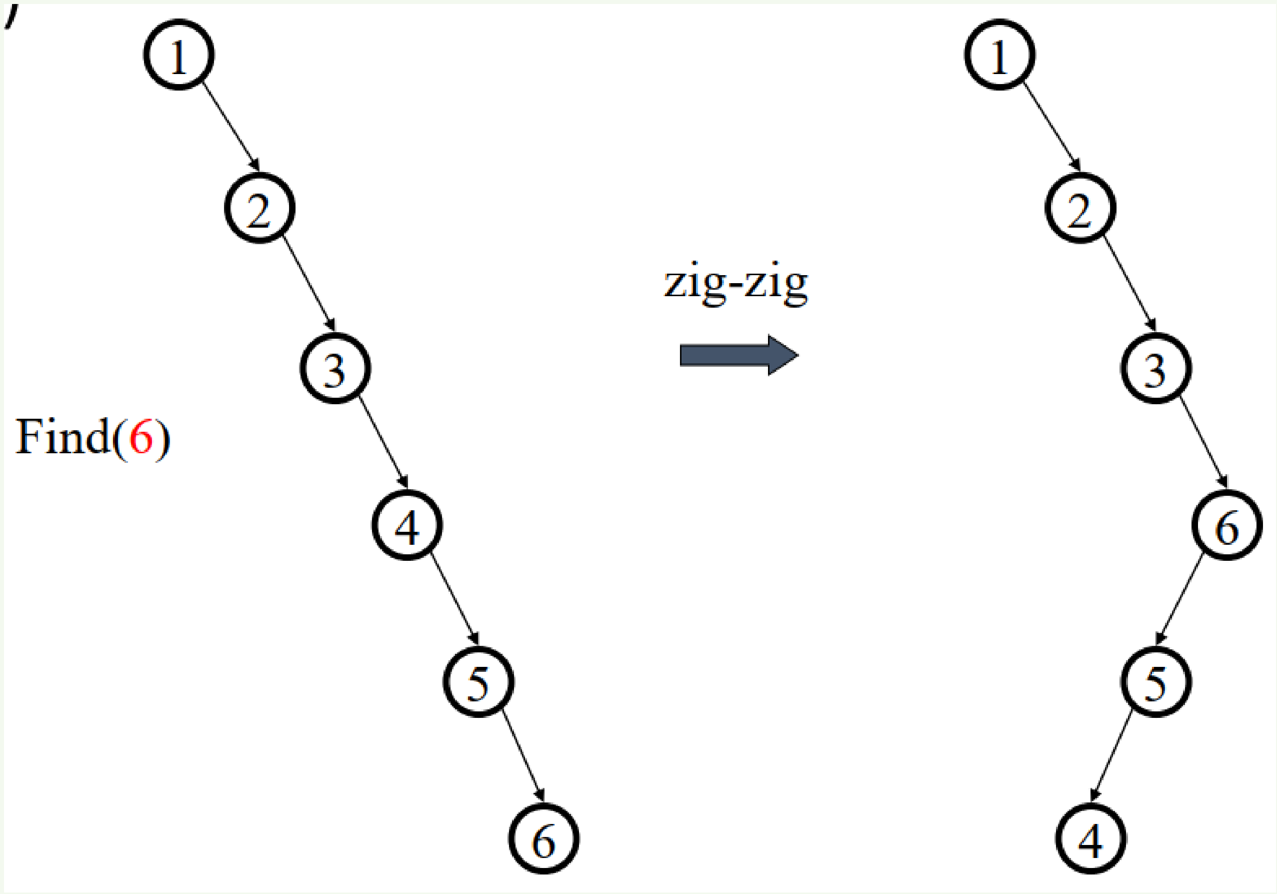


# Splay Operations: Find

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

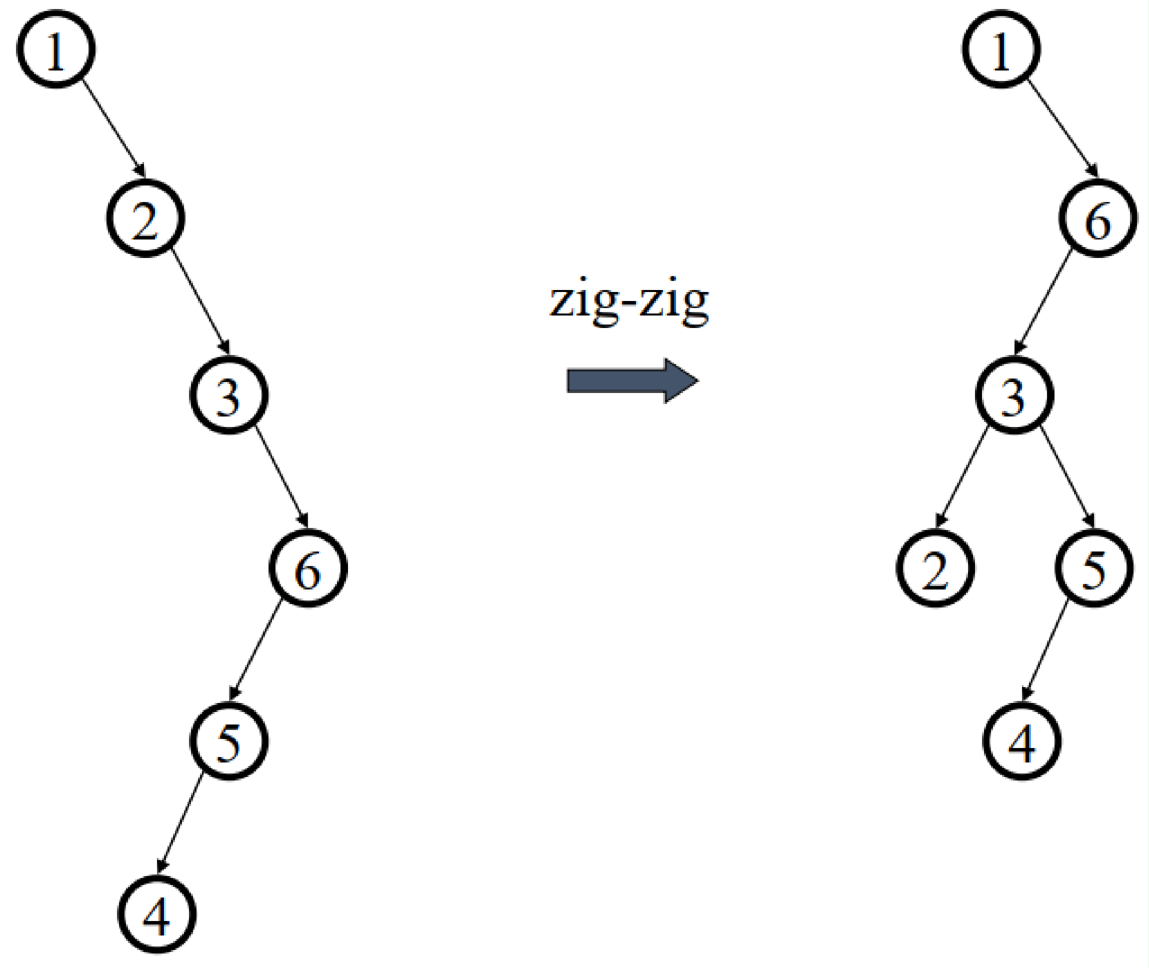
# Splaying Example

Find(6)



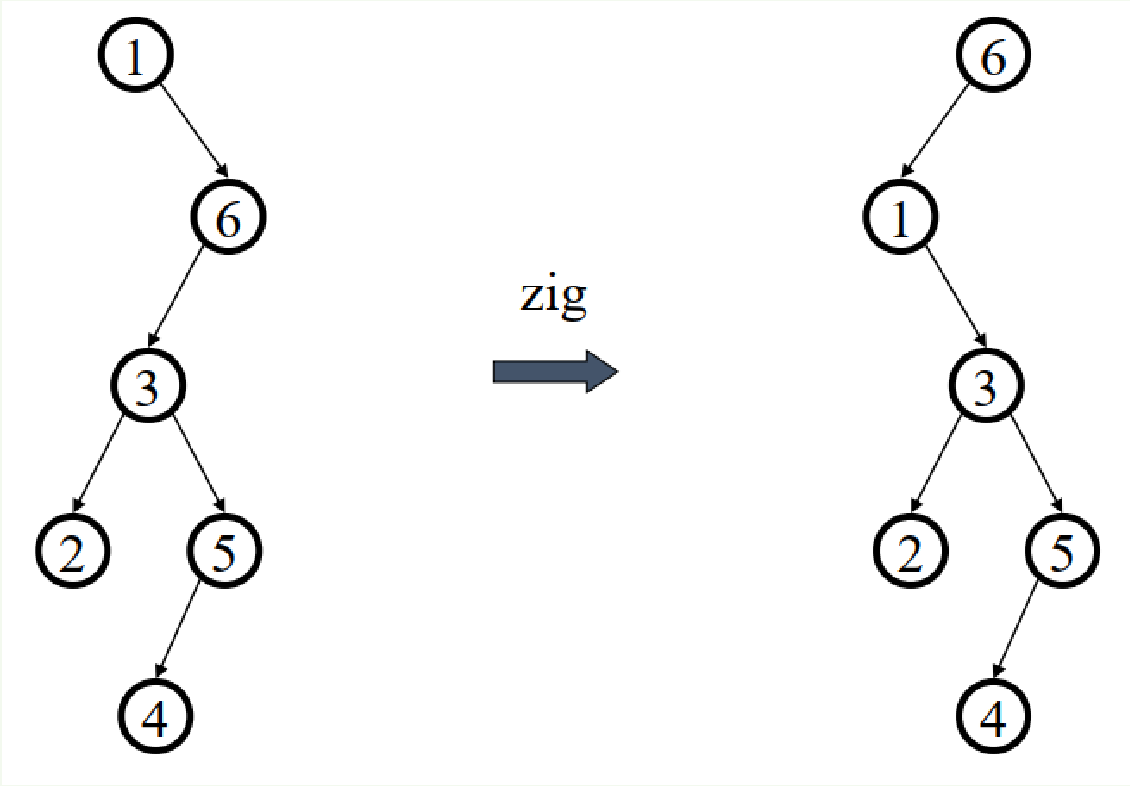
# Splaying Example

… still splaying …



# Splaying Example

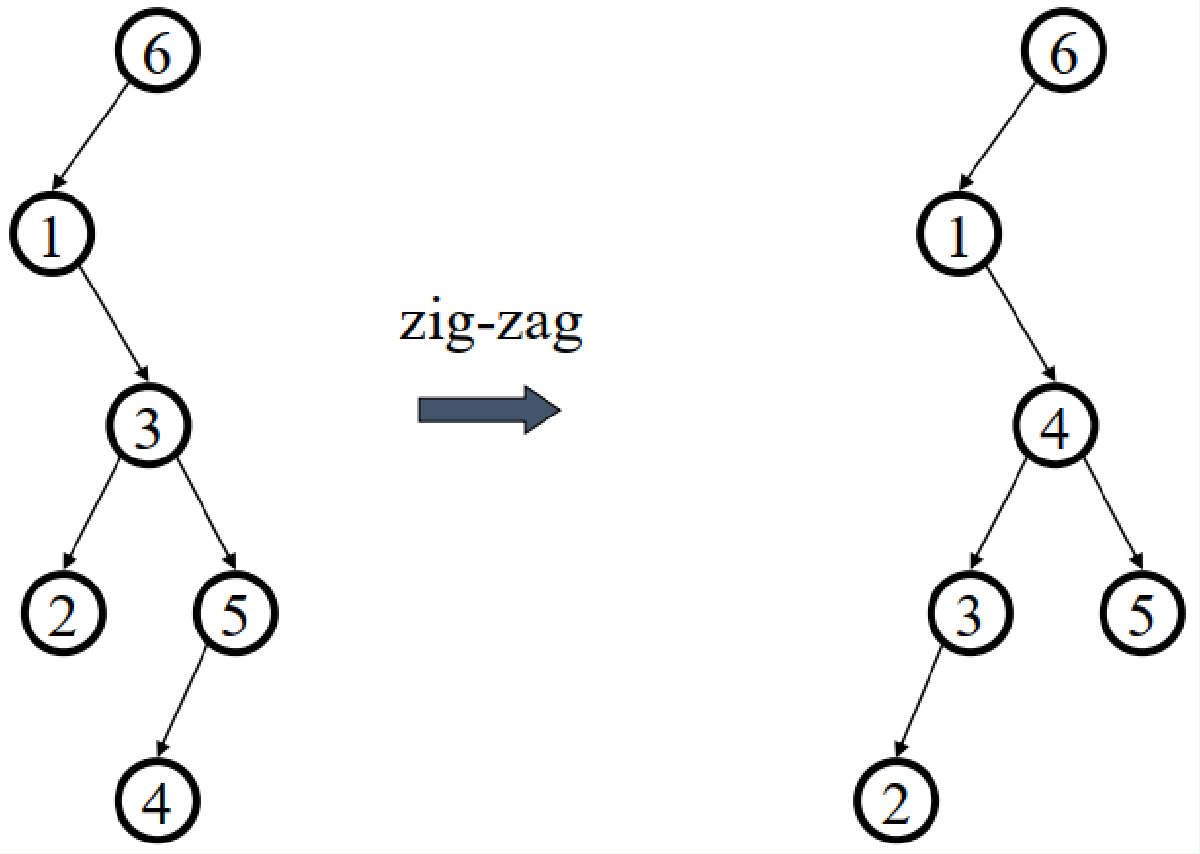
... 6 splayed out!



# Splaying Example

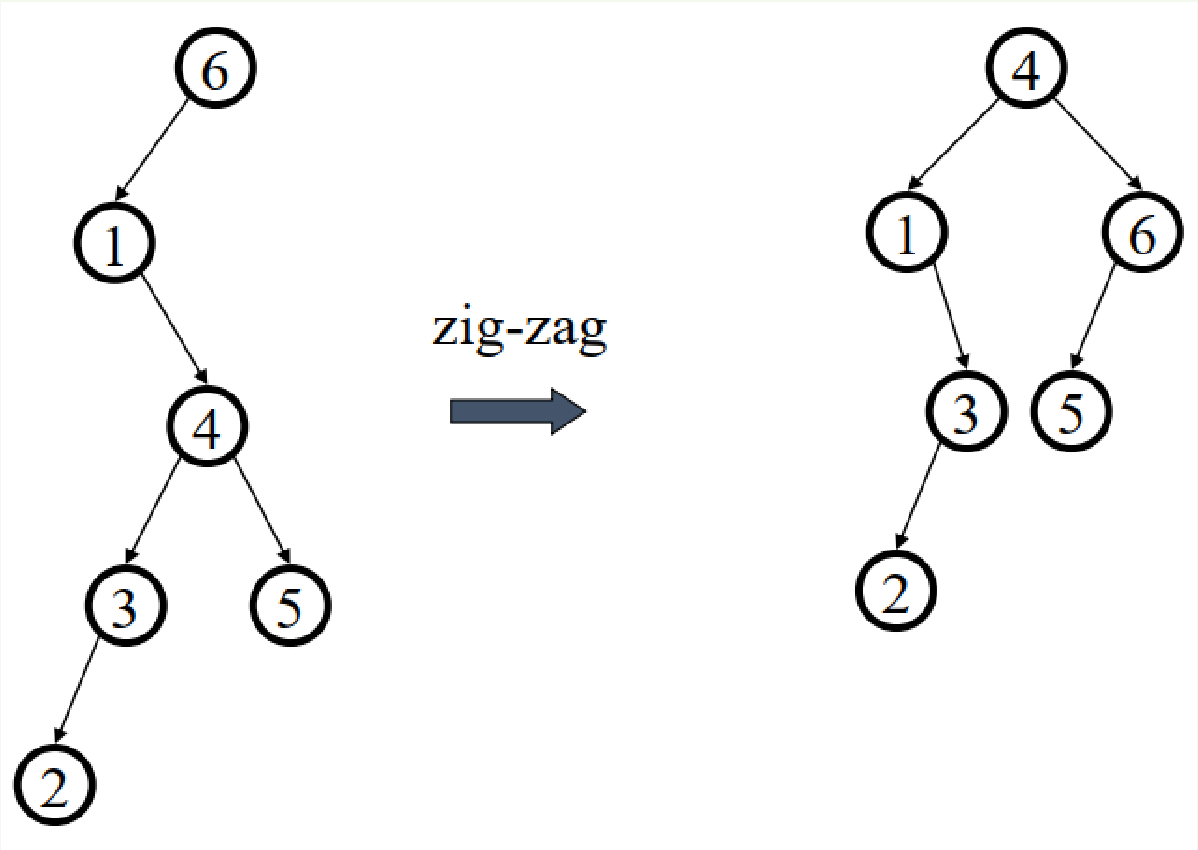
Find (4)

Splay it Again!



# Splaying Example

… 4 splayed out!



# Analyzing Calls to a Data Structure

- Some algorithms involve repeated calls to one or more data structures
- Example:
  - repeatedly insert keys into a dynamic array
  - repeatedly remove the smallest key from the heap
- When analyzing the running time of the overall algorithm, need to sum up the time spent in all the calls to the data structure
- When different calls take different times, how can we accurately calculate the total time?

# Amortized Analysis

- Purpose is to accurately compute the total time spent in executing a sequence of operations on a data structure
- Three different approaches:
  - aggregate method: brute force
  - accounting method: assign costs to each operation so that it is easy to sum them up while still ensuring that result is accurate
  - potential method: a more sophisticated version of the accounting method
- In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time which is lower than the worst- case time of a particular expensive operation.

# Dynamic Array Insertion

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Item No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ...... |
| Table Size | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 | ...... |
| Cost | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9 | 1 | ...... |

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1 \ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1\ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$\le \frac{[n + 2n]}{n}$$

$$\le 3$$

$$\text{Amortized Cost} = O(1)$$

# Splay Tree Algorithm Analysis

- Worst case time is O(n)

- Amortized time for all operations is O(log n)
  - a sequence of M operations on an n-node splay tree takes O(M log n) time.
  - Maybe not now, but soon, and for the rest of the operations

# Why Splaying Helps

- If a node on the access path is at depth $d$ before the splay, it's final depth ≤ 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

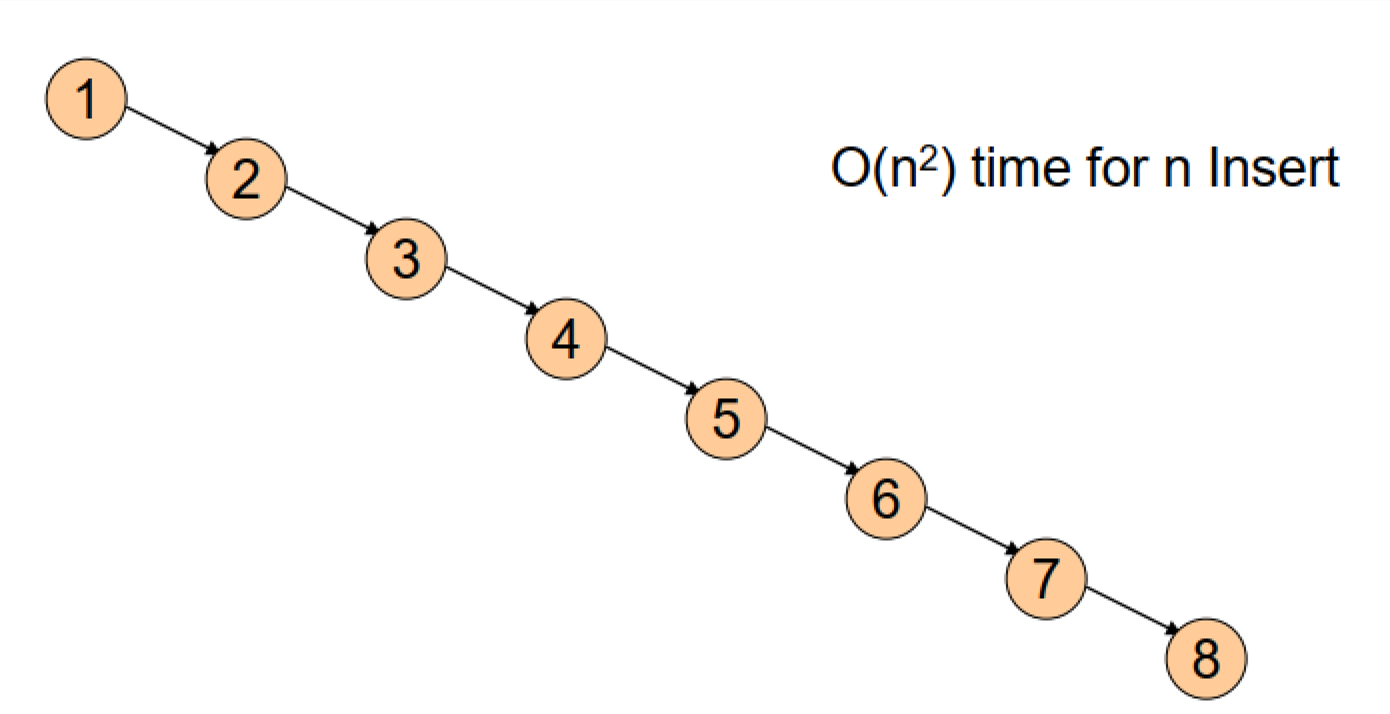
# Splay Tree Insert and Delete

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

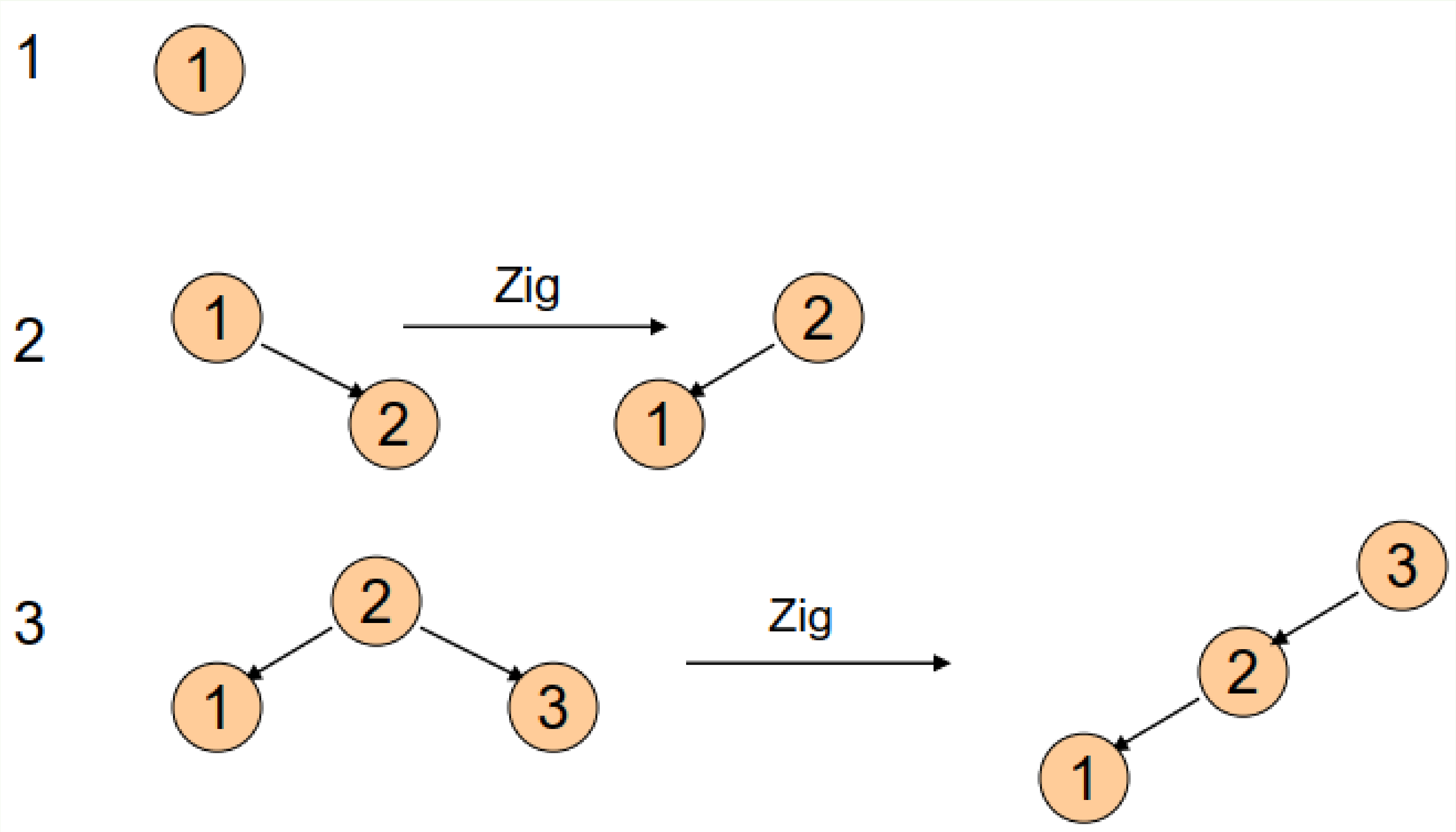
- Delete x
  - Find x
  - Splay x to root and remove it
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.

# Example Insert

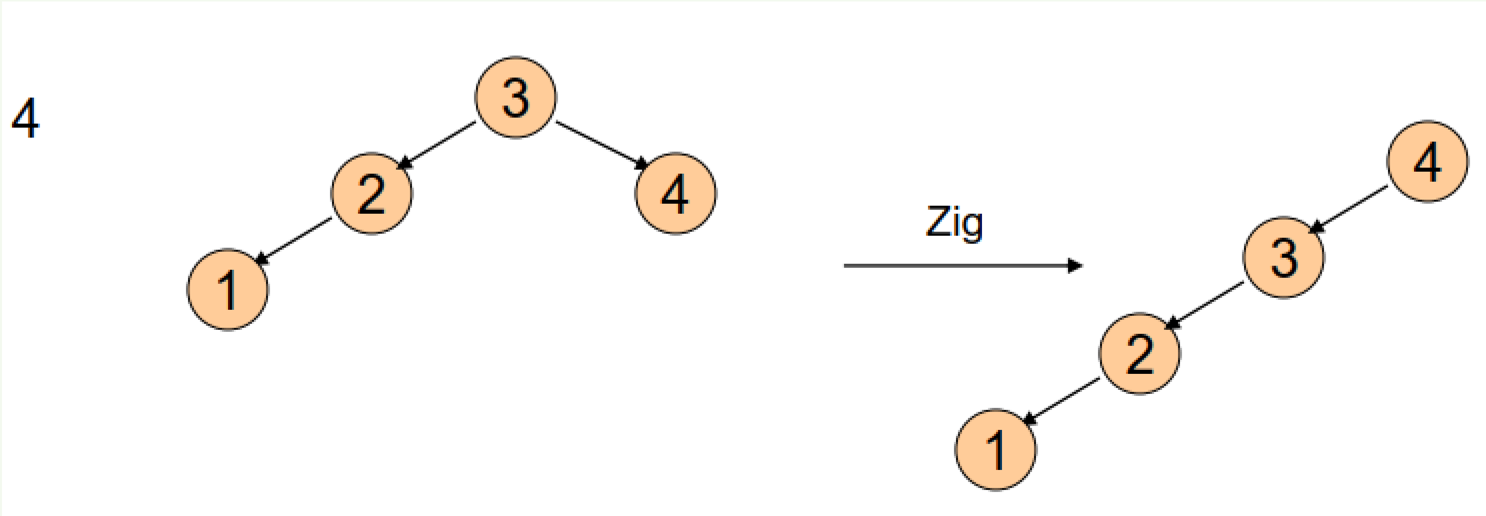
- Inserting in order 1, 2, 3, …, 8
- Without self-adjustment



# With Self-Adjustment

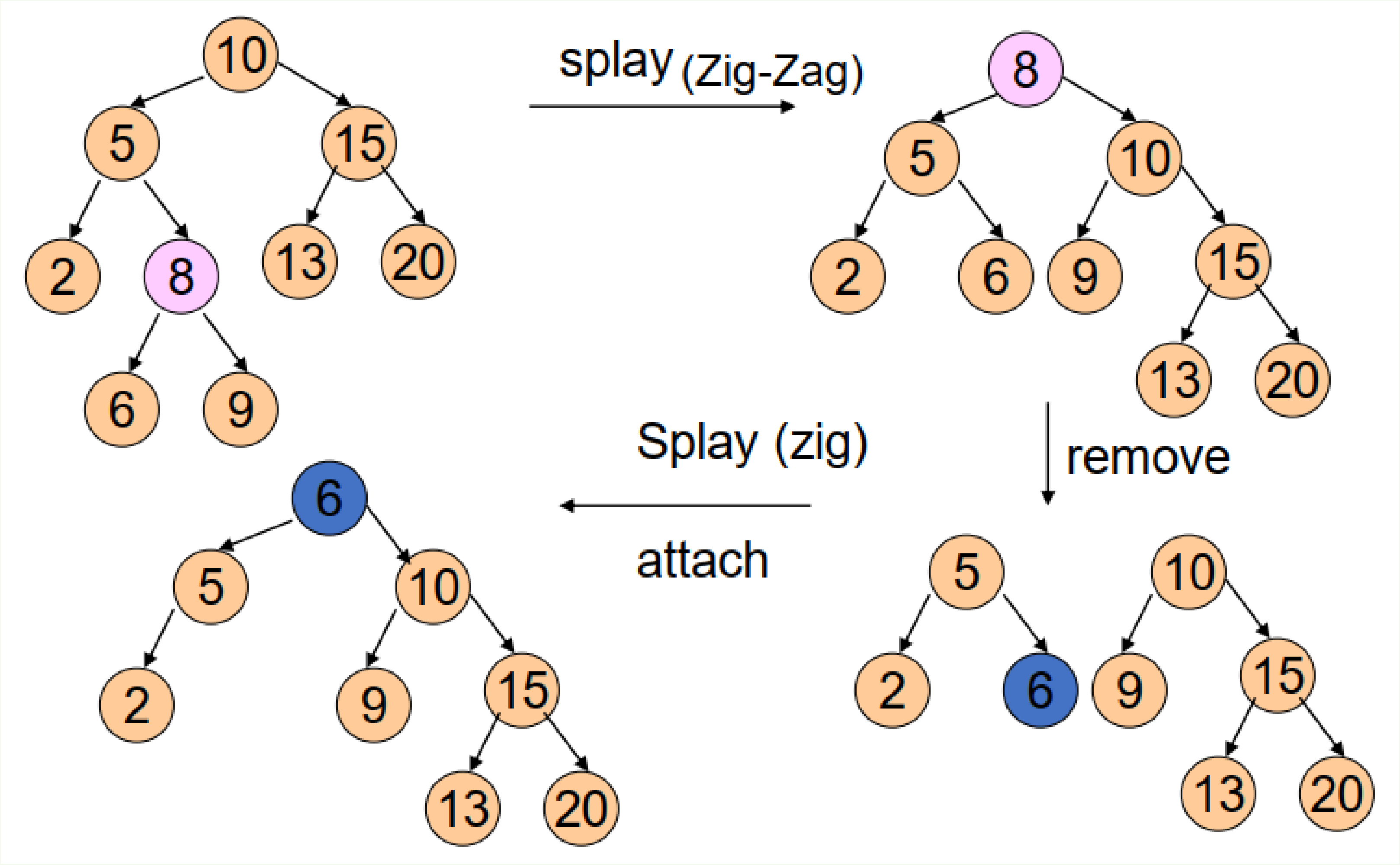


# With Self-Adjustment




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)