CSCE 3110
Data Structures & Algorithms

- Splay Trees
- Reading: Weiss, chap. 4

Content

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

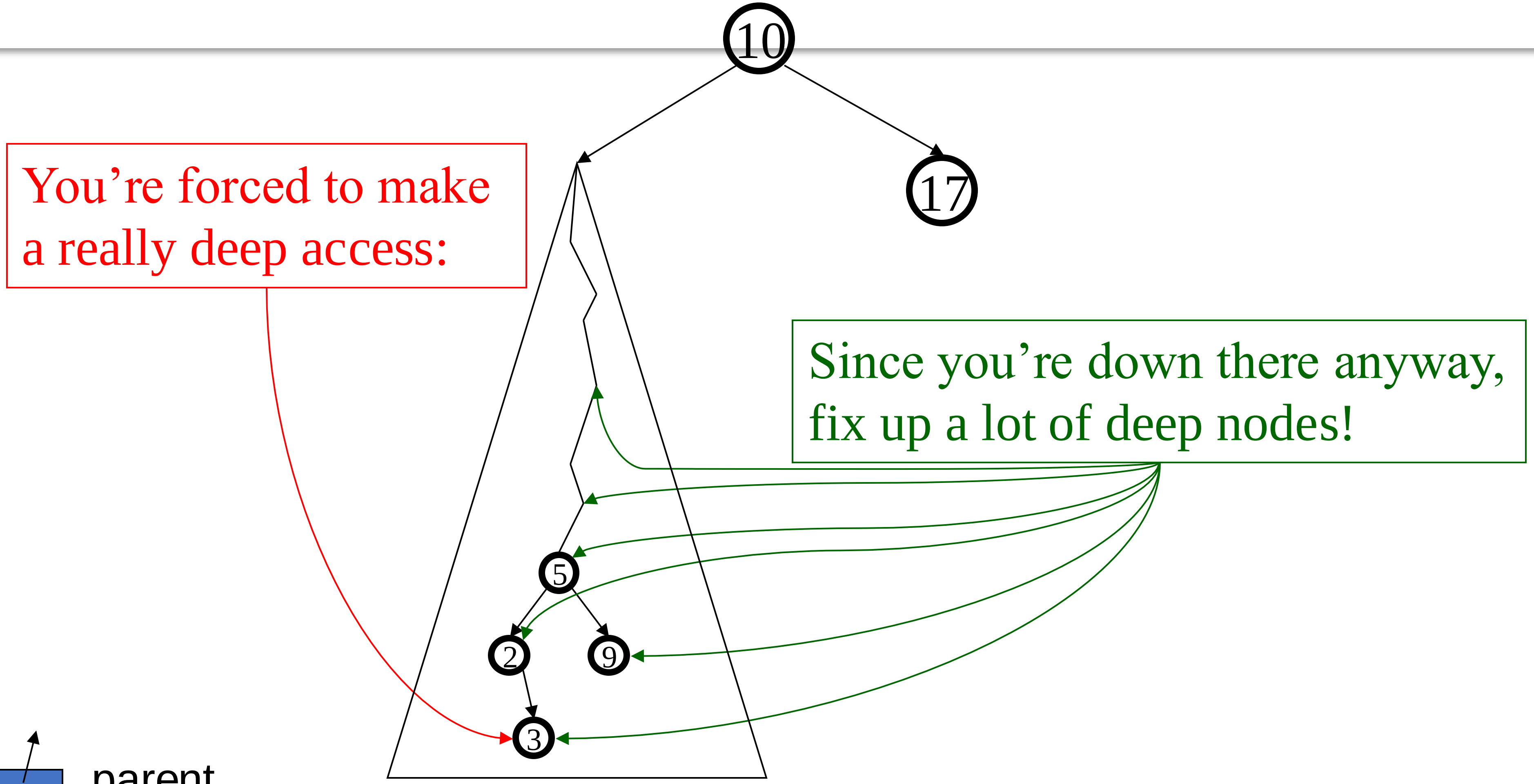
Self adjusting Trees

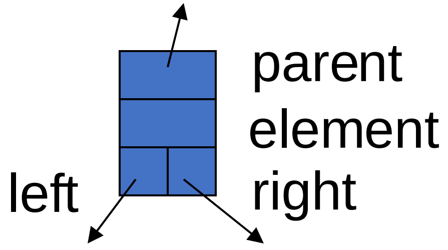
- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed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

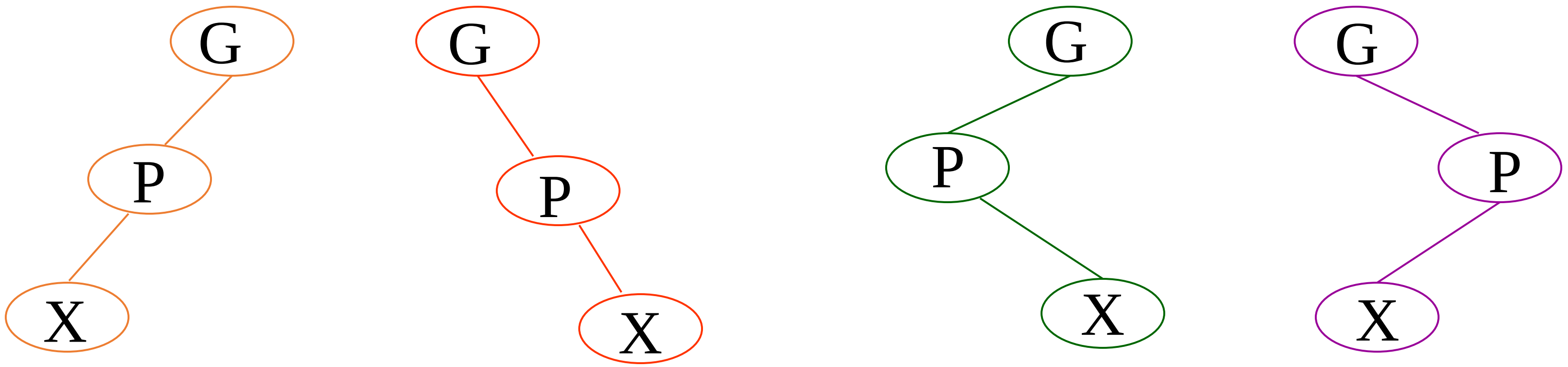
Splaying Cases

Node being accessed (x) is:

- Root
- Child of root
- Has both parent (p) and grandparent (g)

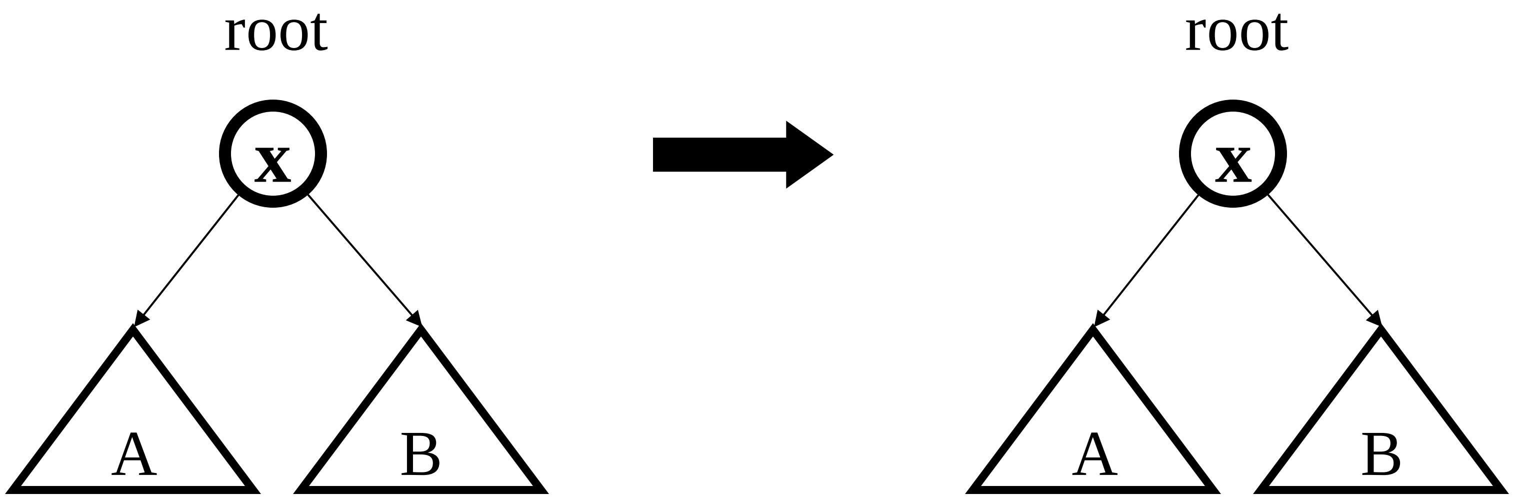
  Zig-zig pattern: g → p → x is left-left or right-right

  Zig-zag pattern: g → p → x is left-right or right-left

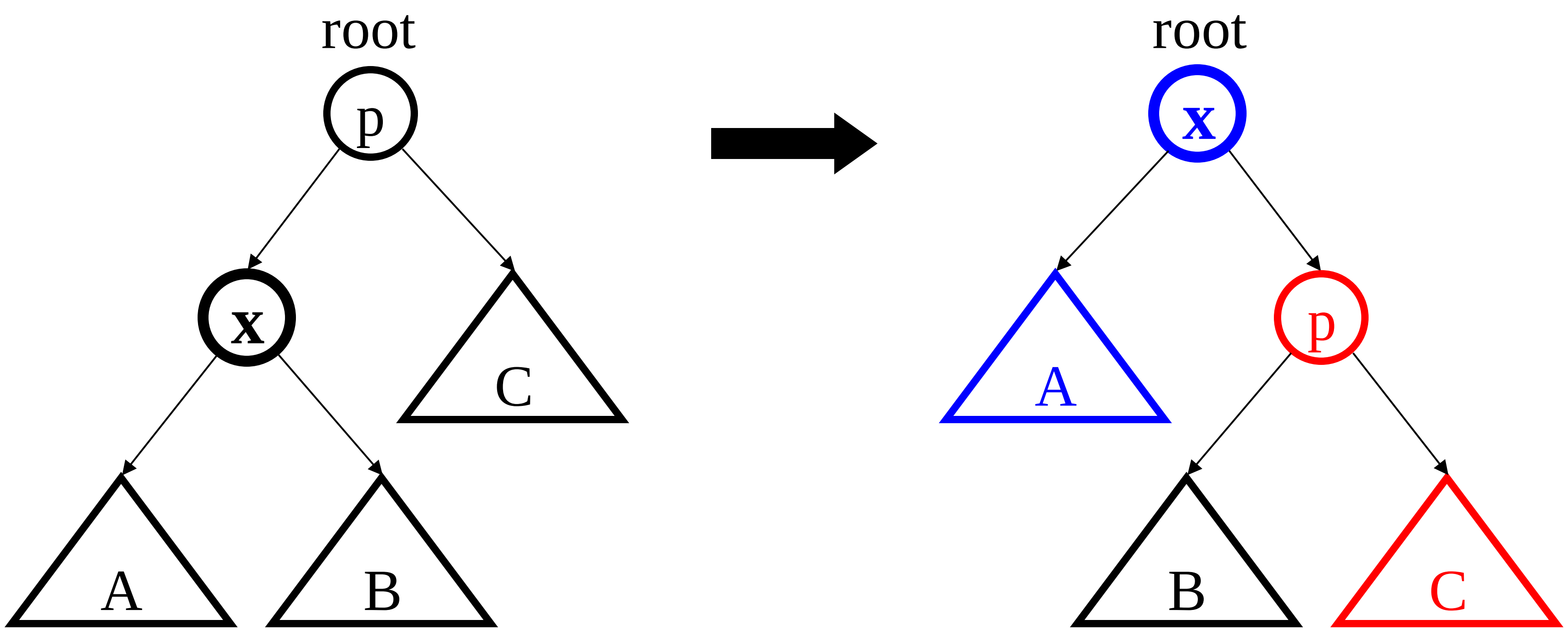


Access root:

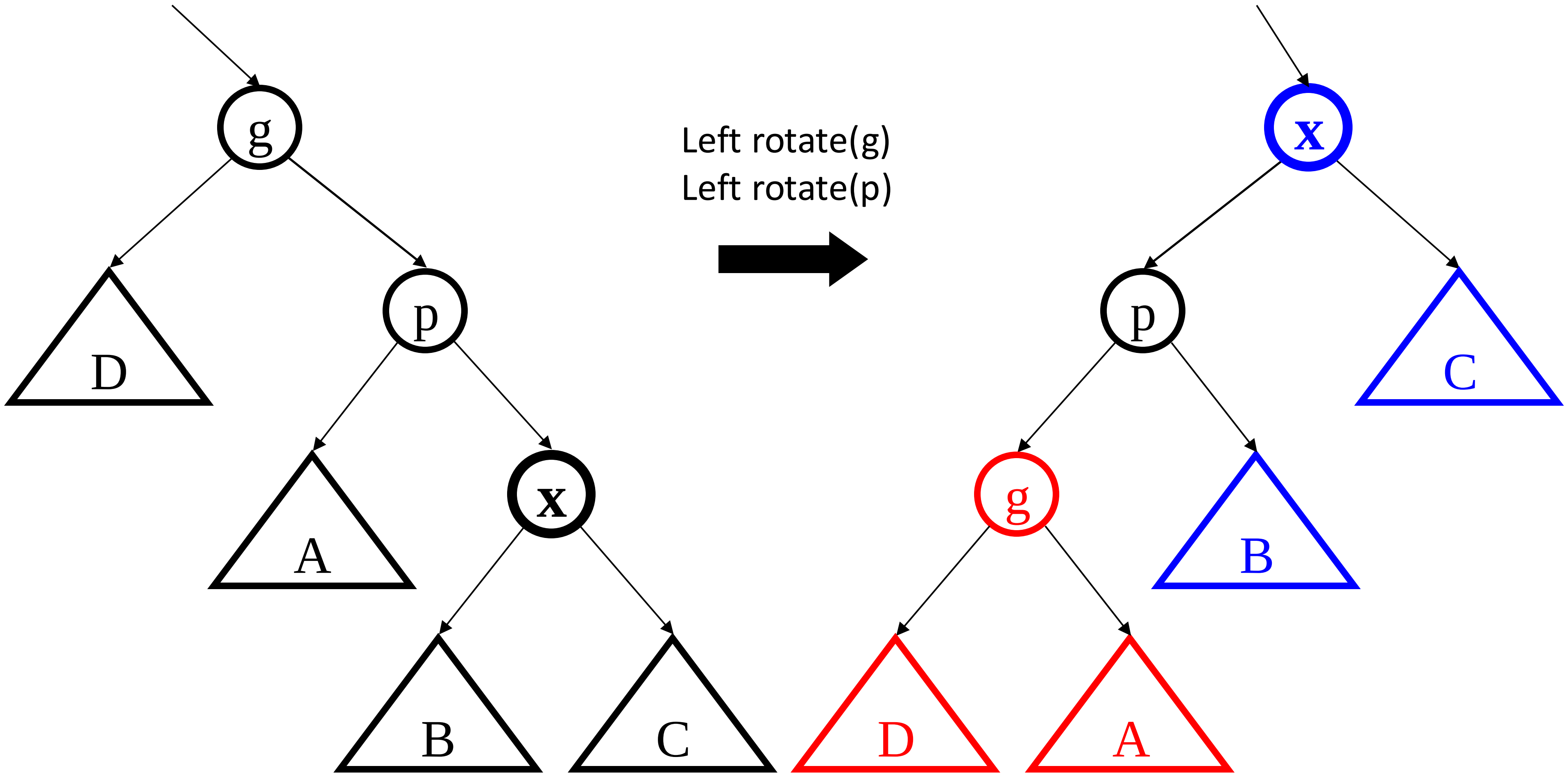
Do nothing (that was easy!)



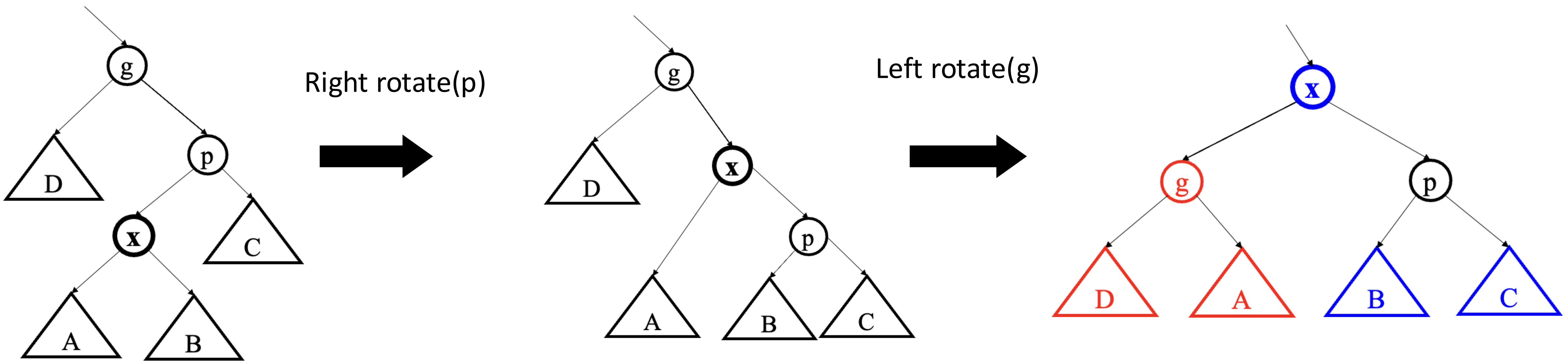
Access child of root:
Zig (AVL single rotation)



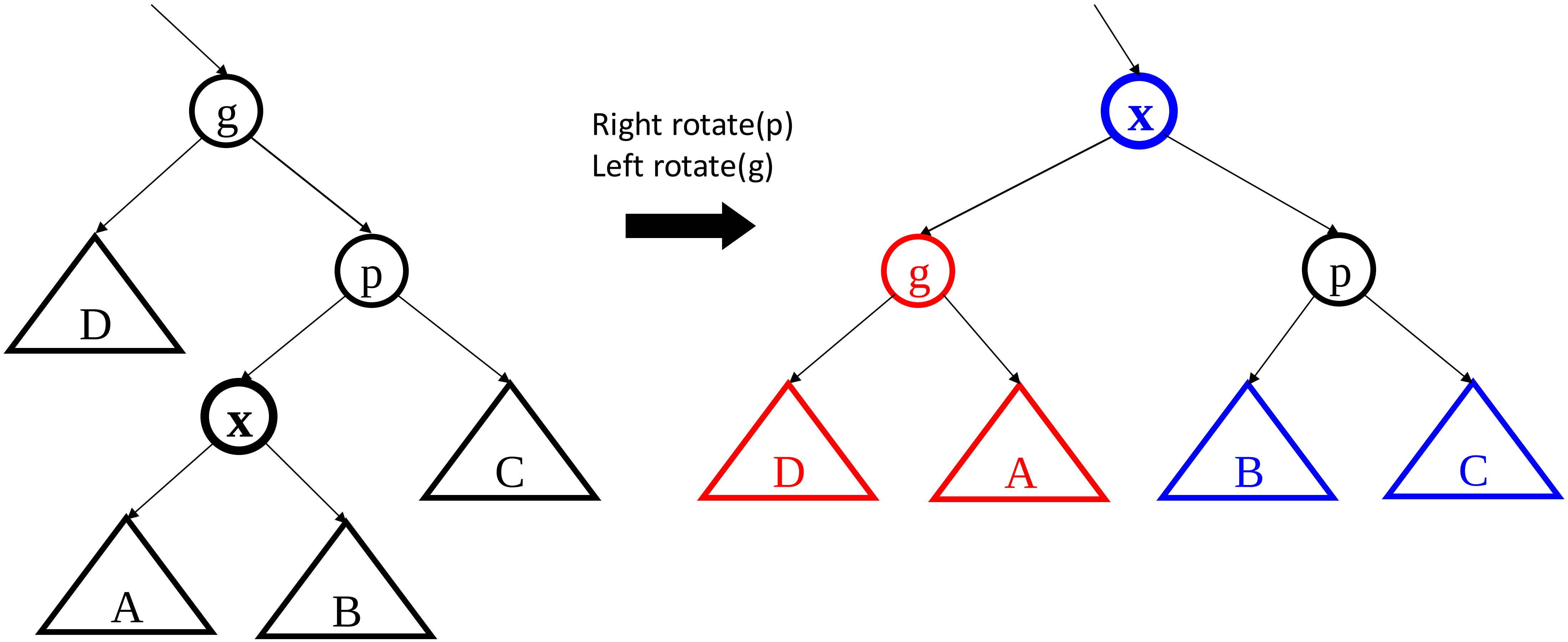
Access (LL, RR) grandchild: Zig-Zig



Access (LR, RL) grandchild: Zig-Zag



Access (LR, RL) grandchild: Zig-Zag

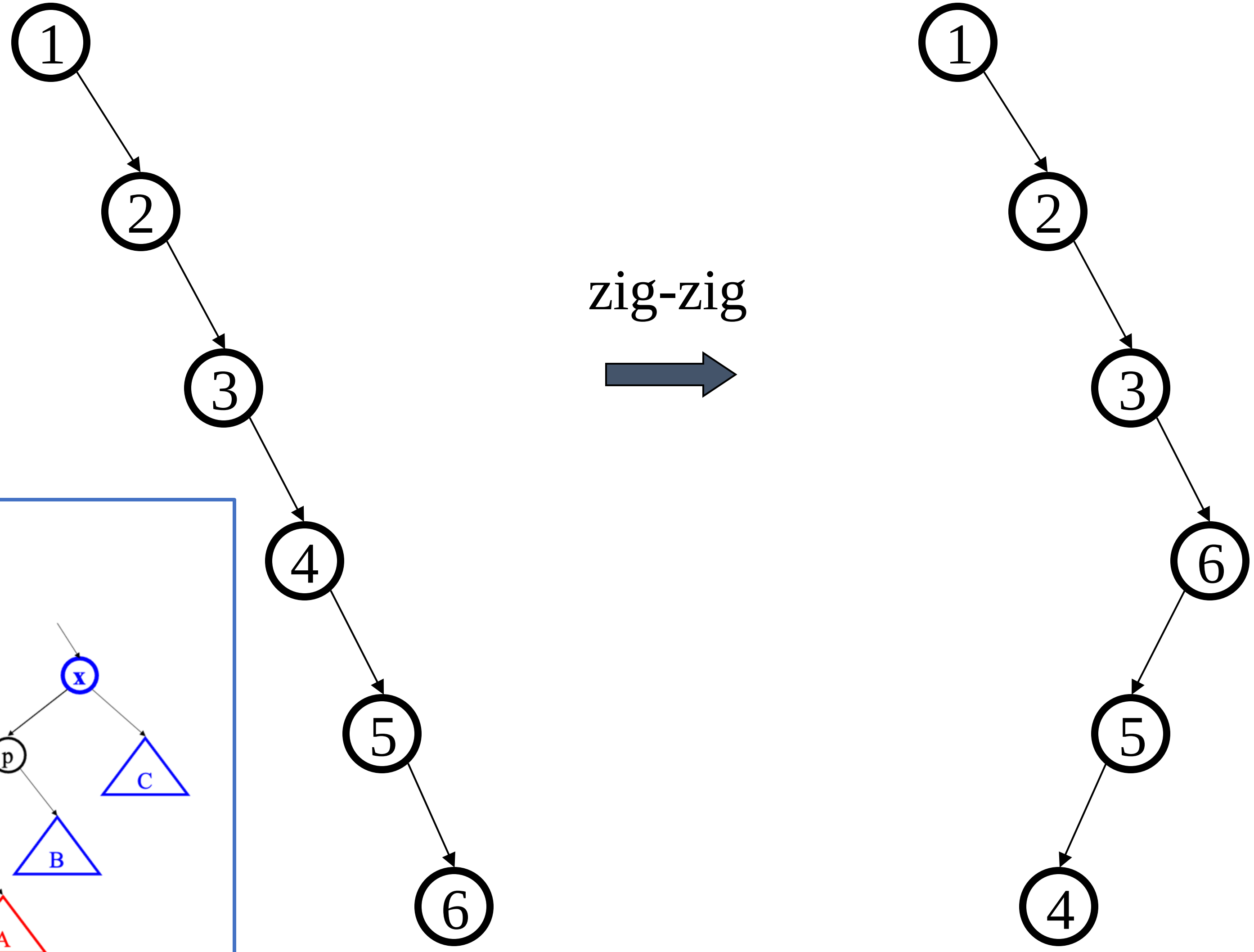


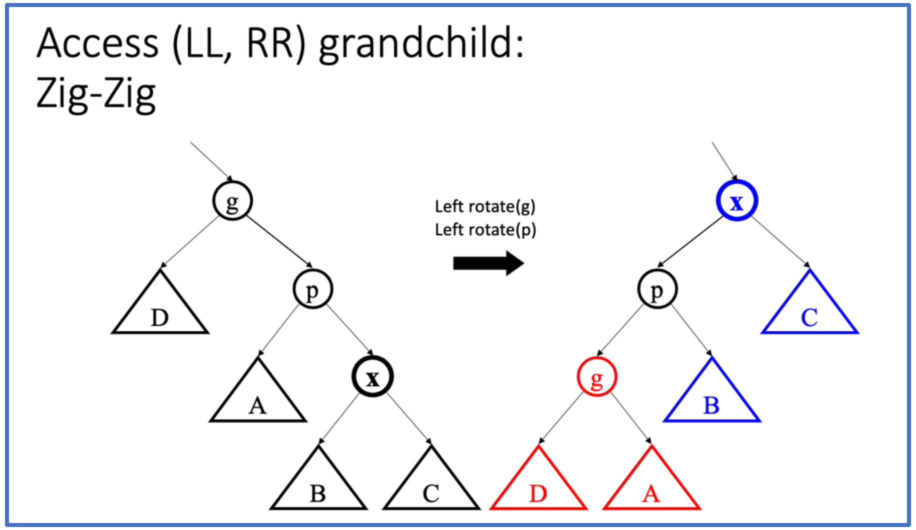
Splay Operations: Find

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

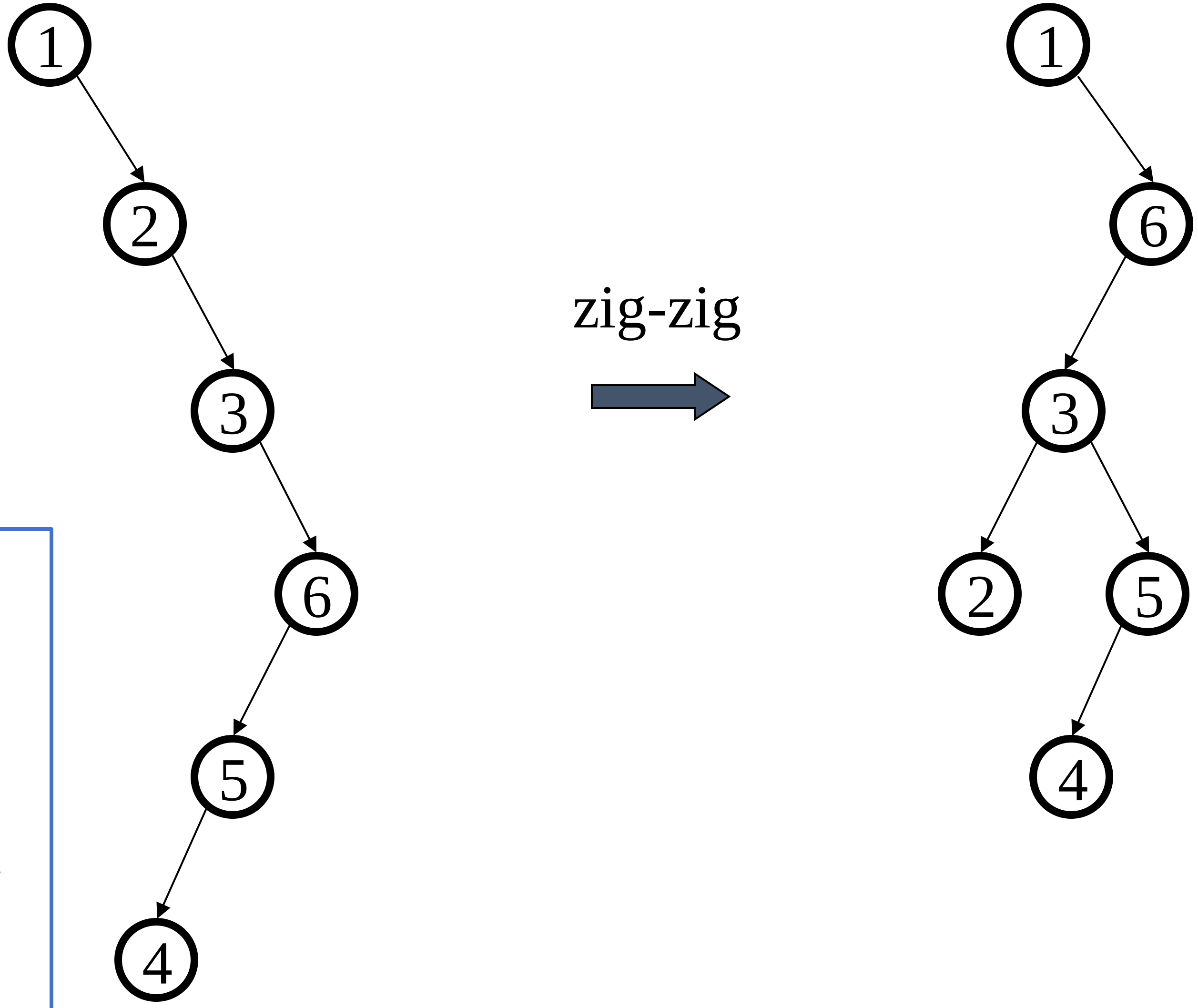
Splaying Example: Find(6)

Find(6)

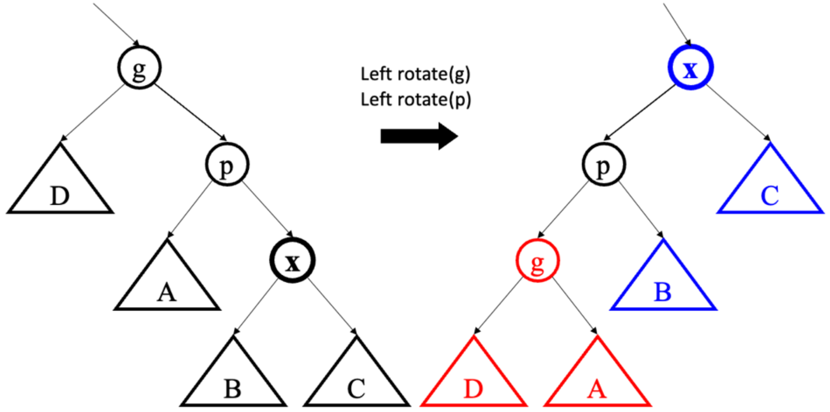




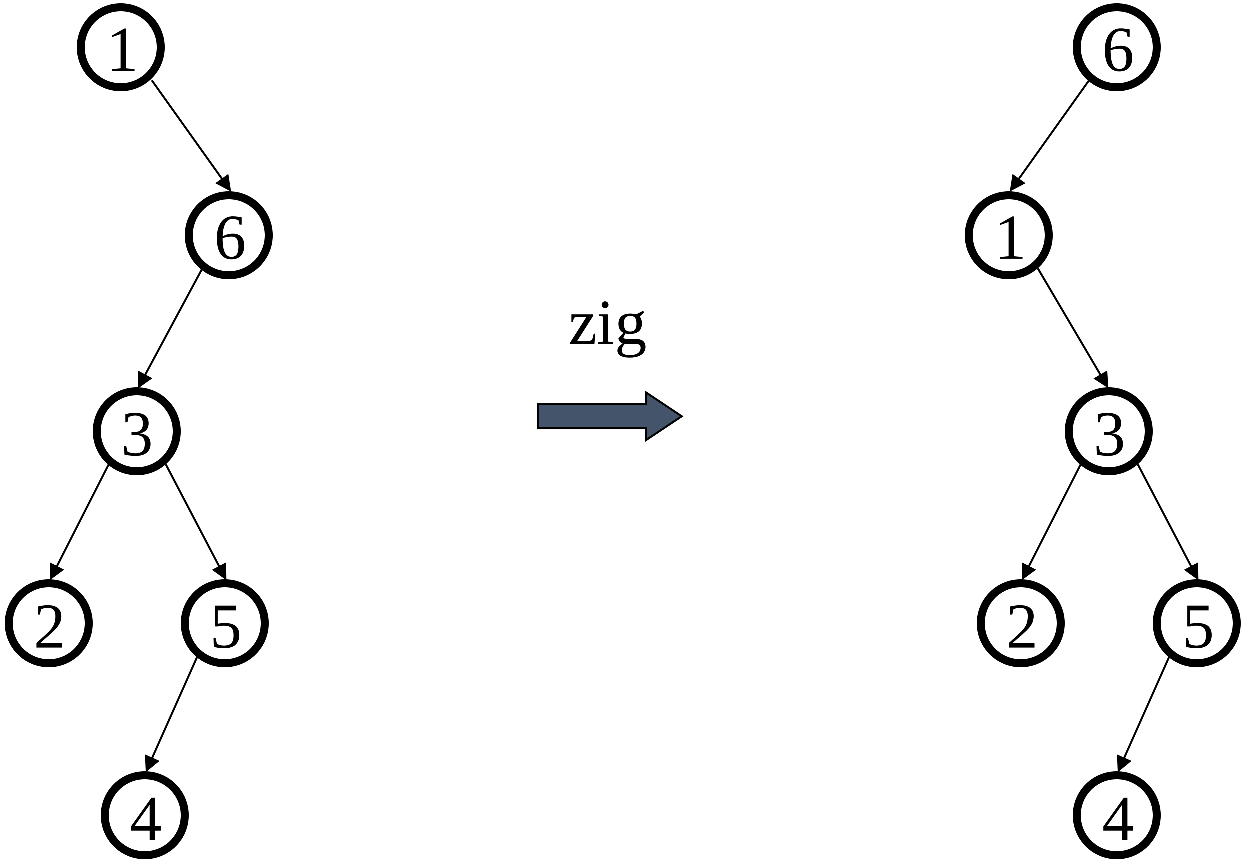
… still splaying …



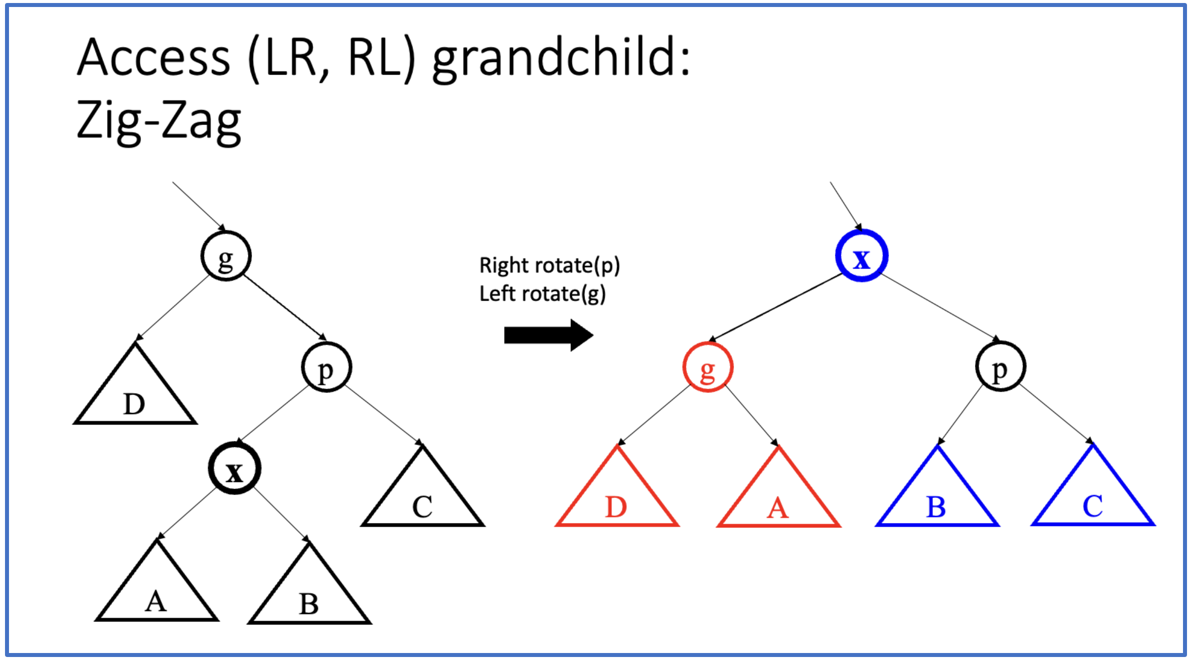
Access (LL, RR) grandchild:
Zig-Zig

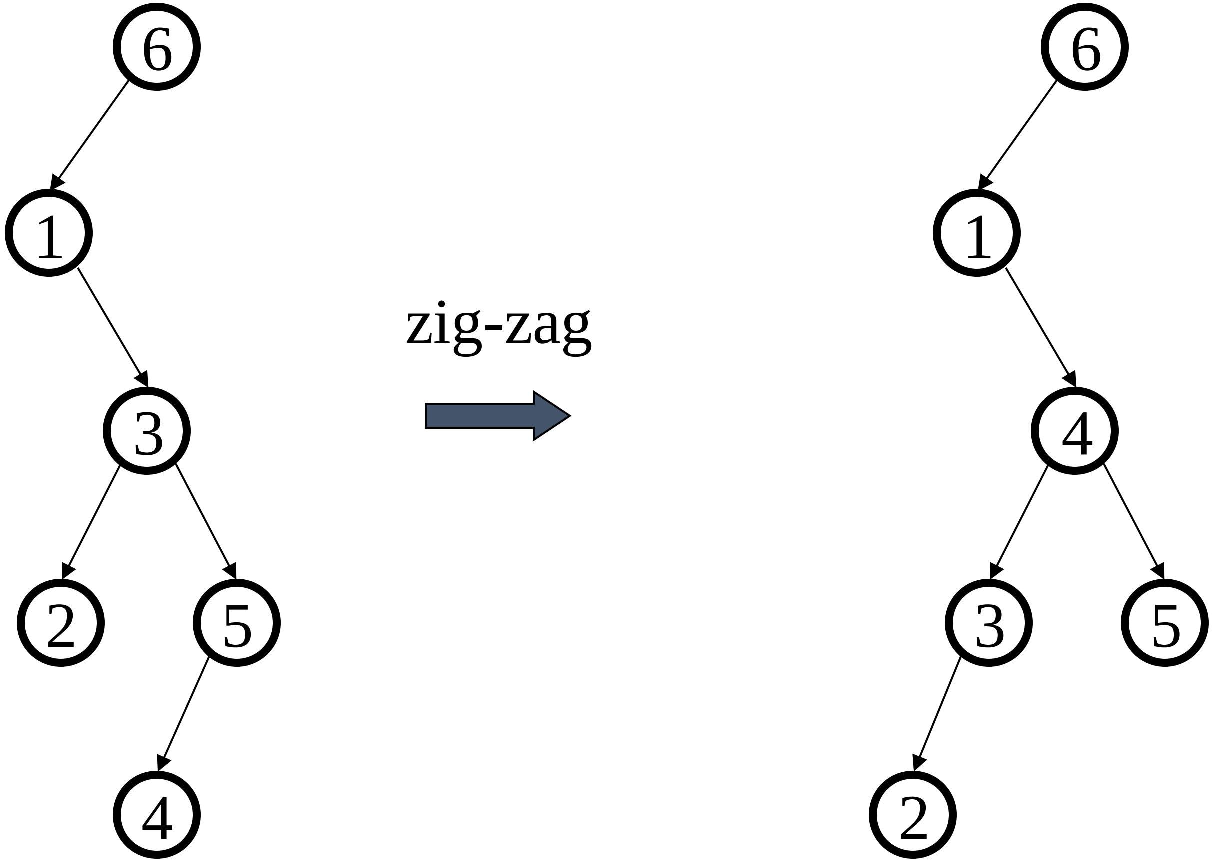


… 6 splayed out!



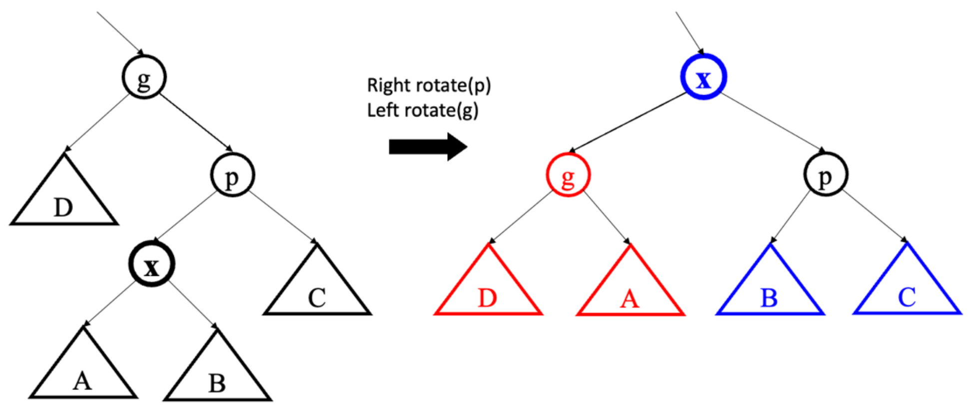
Find (4)
Splay it Again, Same!

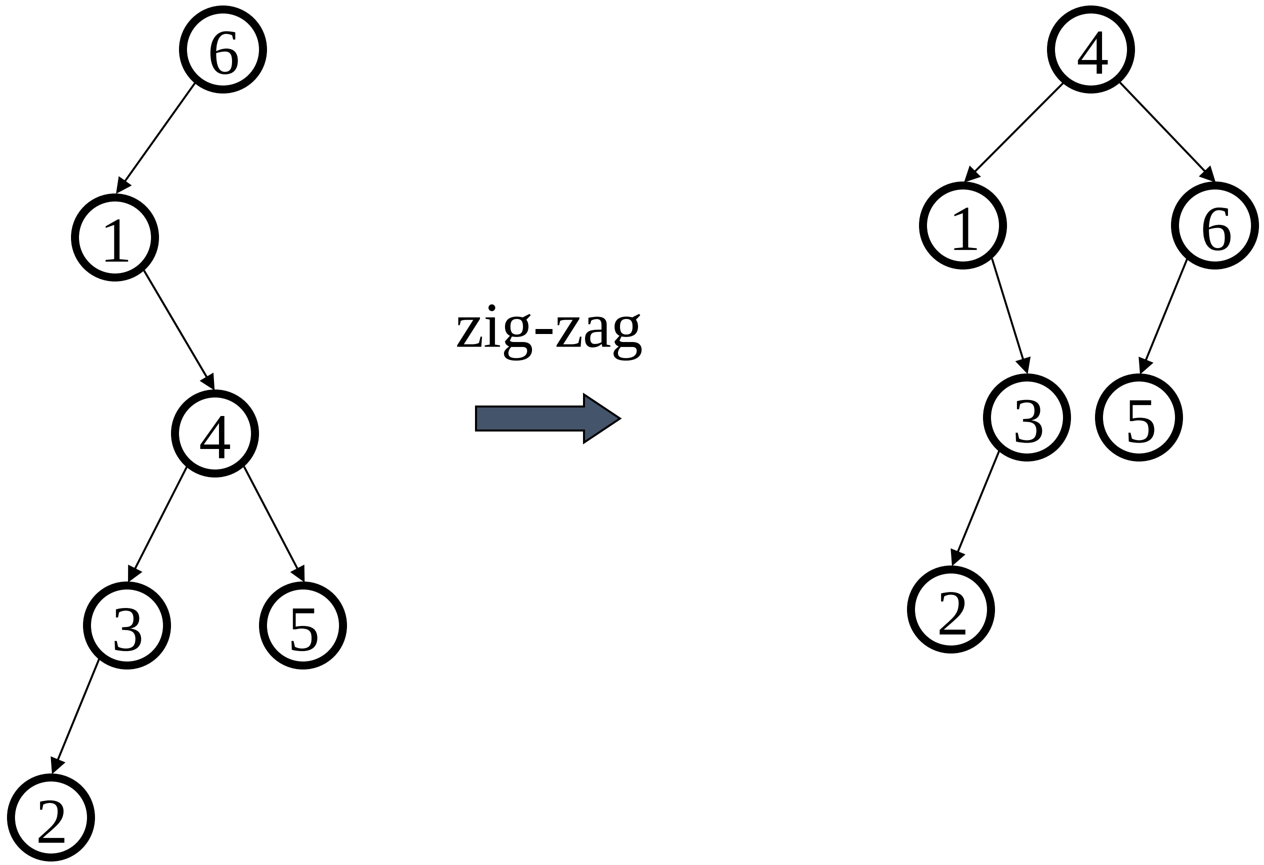




… 4 splayed out!

Access (LR, RL) grandchild:
Zig-Zag





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	......

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

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

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

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

$$\le 3$$

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

$$\boxed{S_n = \frac{a(1 - r^n)}{1 - r}}$$

Splay tree algorithm analysis

- Worst case time is $O(n)$
- Amortized time for each operation is $O(\log n)$
  - while individual operations might occasionally be expensive (up to O(n)), the **average cost** of operations, when considering a sequence of M operations, is much lower.
  - Specifically, a sequence of M operations on an n-node splay tree will take O(Mlogn) time. This ensures that the average time per operation is O(logn).

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
    - The **root** (which remains at depth 0),
    - The **child of the root** (which stays close to the top),
    - The **node being splayed** (which becomes the new root).
- Overall, nodes which are below nodes on the access path tend to move closer to the root
- By reducing the depth of frequently accessed nodes, splaying keeps the tree relatively balanced, leading to faster operations in the long run. This is the reason why the **amortized time complexity** of operations in a splay tree is O(logn), even though the worst-case time for a single operation can be O(n).

Splay Tree Insert

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

Insertion (Insert x):

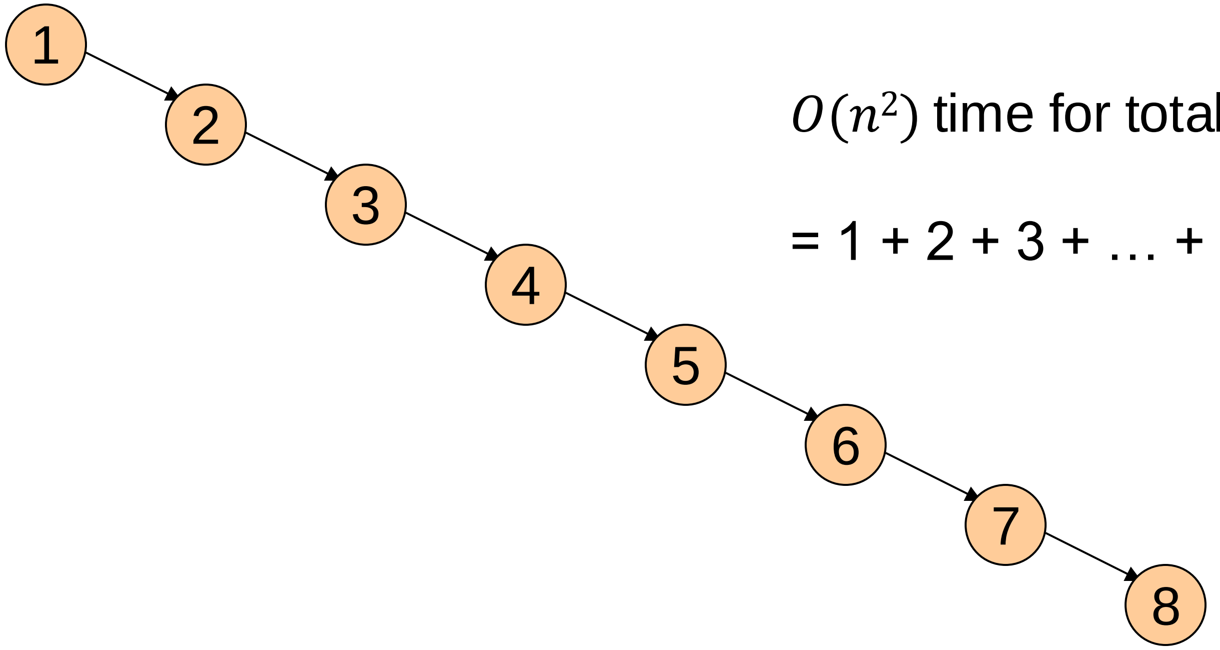
1. **Insert as Normal**:
   - You first insert the element x as you would in a regular binary search tree (BST). This involves finding the correct location for x and placing it there.
2. **Splay x to Root**:
   - After inserting x, you perform a **splay operation** to move x to the root of the tree. This step helps in maintaining the self-adjusting property of splay trees, which improves access times for frequently accessed elements.

Splay Tree Delete

- 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

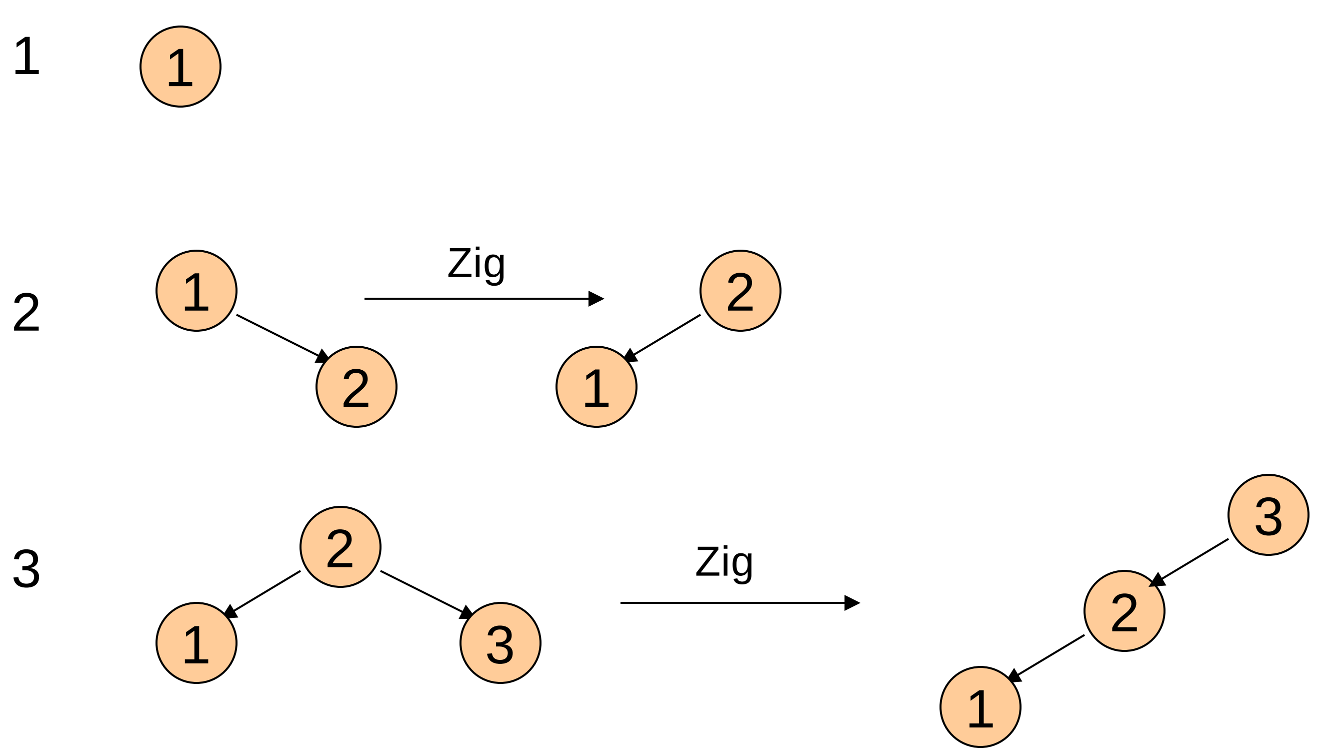
- Inserting in order 1, 2, 3, …, 8
- Without self-adjustment



$O(n^2)$ time for total n inserts

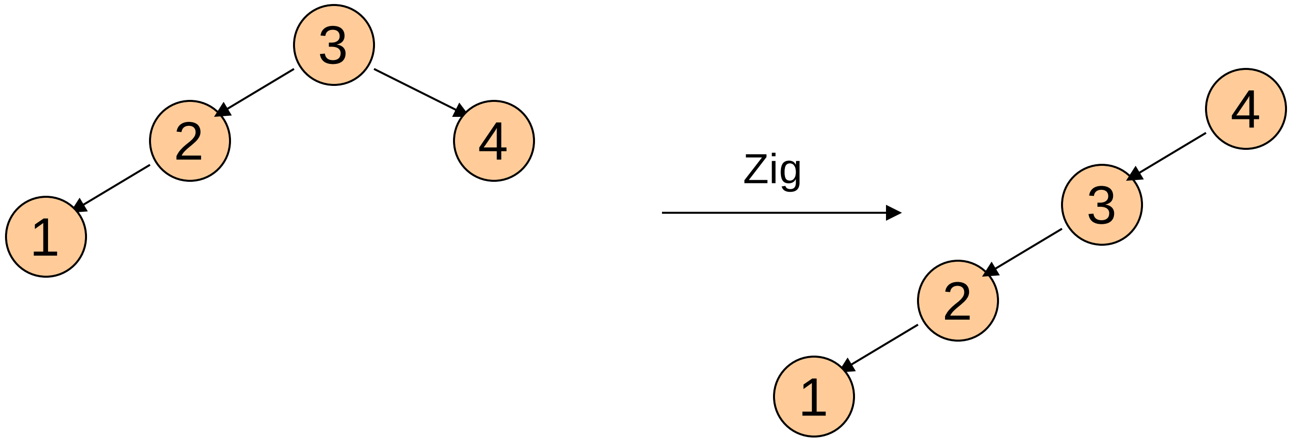
= 1 + 2 + 3 + … + (n-1)

With Self-Adjustment



With Self-Adjustment

4



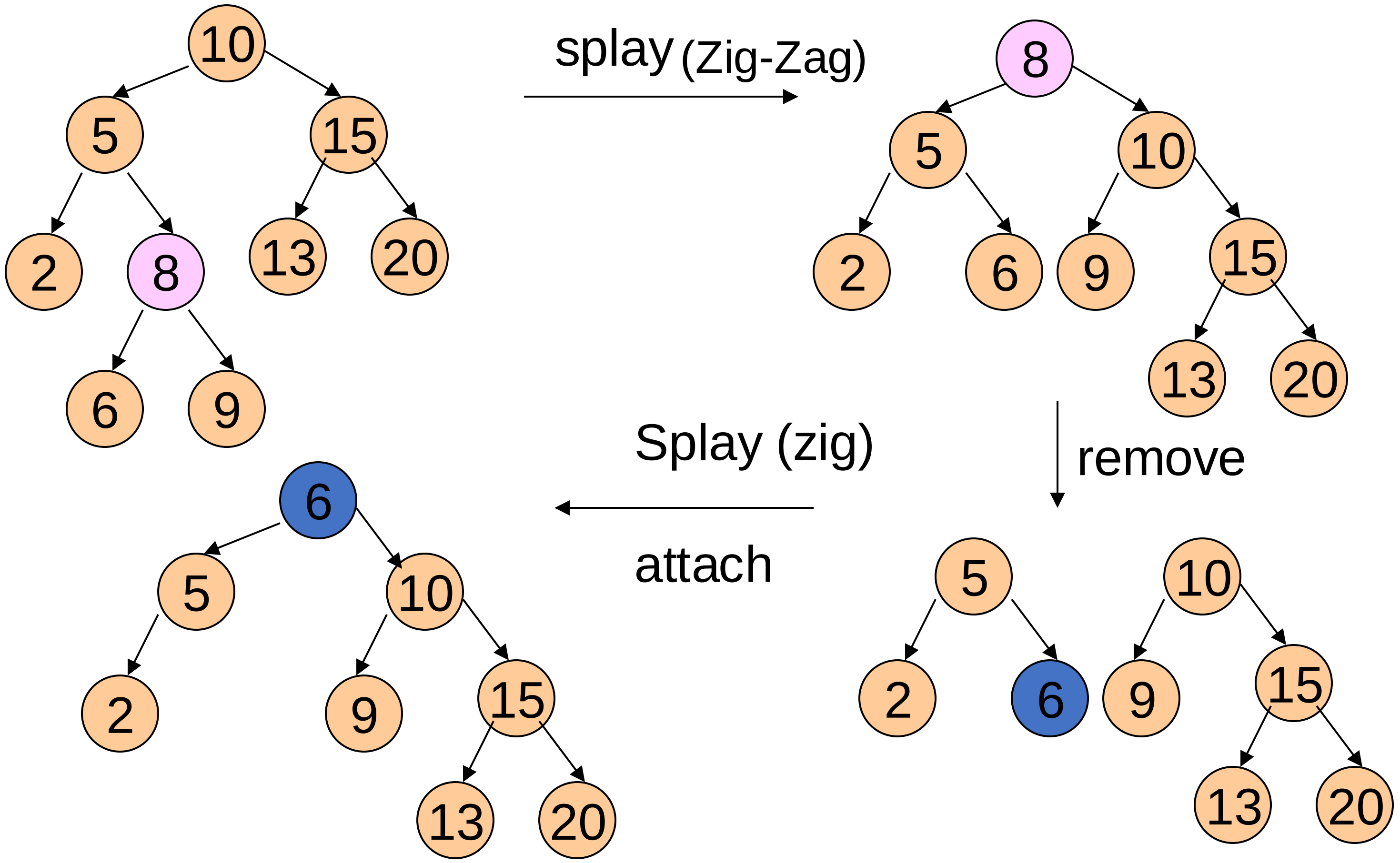
Each Insert takes $O(1)$ time, therefore $O(n)$ time for n inserts!!

Time Complexity:

- **Each Insert Takes $O(1)$**: The self-adjustment ensures that the tree remains relatively balanced, so each insert operation only takes constant time $O(1)$ to maintain balance.
- **Total Time for n Inserts is $O(n)$**: Since each insertion takes $O(1)$, the total time for n insertions is $O(n)$, which is much more efficient than the $O(n^2)$ time in an unbalanced tree.

Example Deletion

- 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.



Summary of Binary 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 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)