Binary Search Trees:
Splay Trees

Splay Trees

- A **splay tree** is a binary search tree with the additional property that recently accessed elements are quick to access again by splaying these to the root.
- It is said to be an efficient binary search tree because it performs basic operations such as search, insertion, and deletion operations in $O(\log n)$ amortized time.
- For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

Splay Trees

- All normal operations on a binary search tree are combined with one basic operation, called *splaying*.
- In a splay tree, search, insert and deletion are first done as BST, then followed by some rotation or splaying to bring the element to the root.

- **Why ?**
  - Since the most frequently accessed node is always moved closer to the starting point of the search (or the root node), those nodes are therefore located faster.
  - A simple idea behind it is that if an element is accessed, it is likely that it will be accessed again.

Motivation for Splay Trees

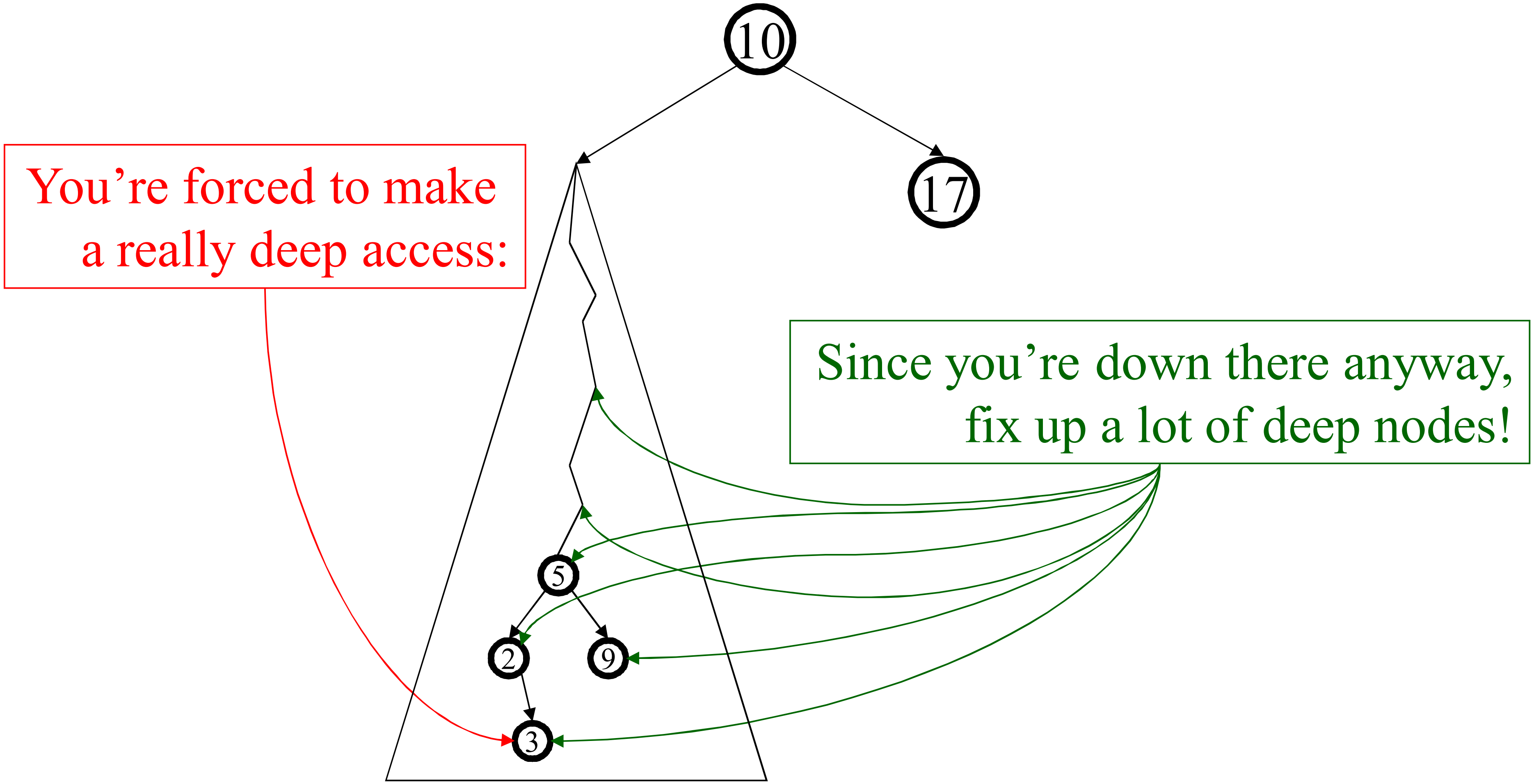
Problems with other balanced Trees

- AVL Tree:
  - extra storage/complexity for height fields
  - ugly delete code
- Red-Black Tree
  - Complex coding

Solution: splay trees

- amortized time for all operations is $O(\log n)$
- worst case time is $O(n)$
- insert/find always rotates node *to the root*!

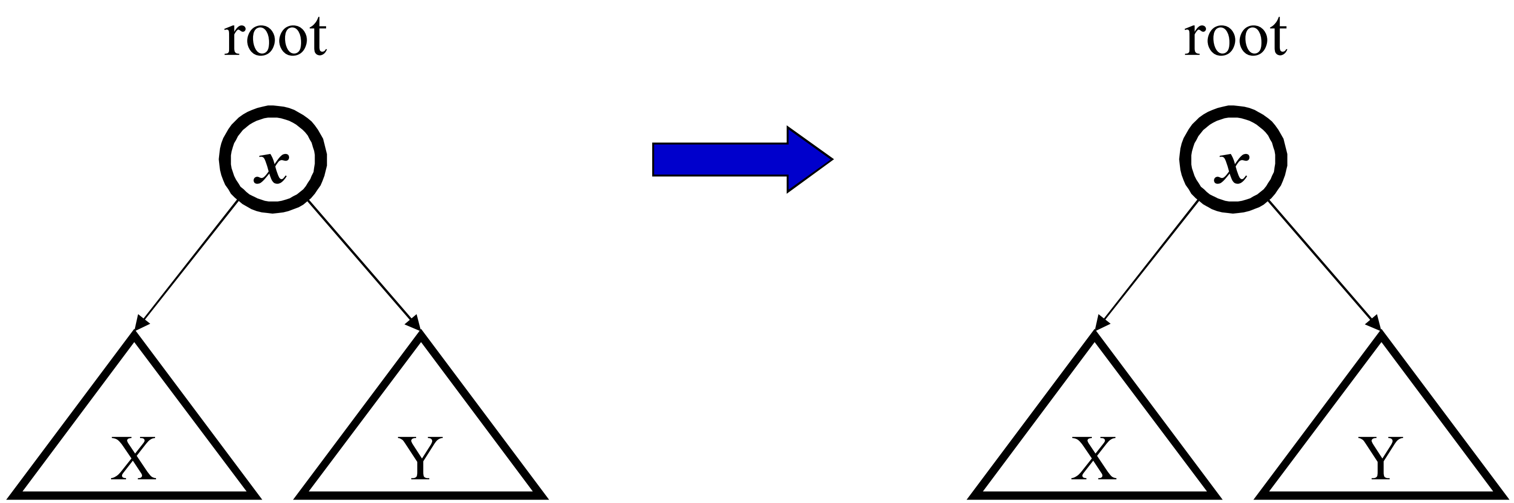
Splay Tree Idea



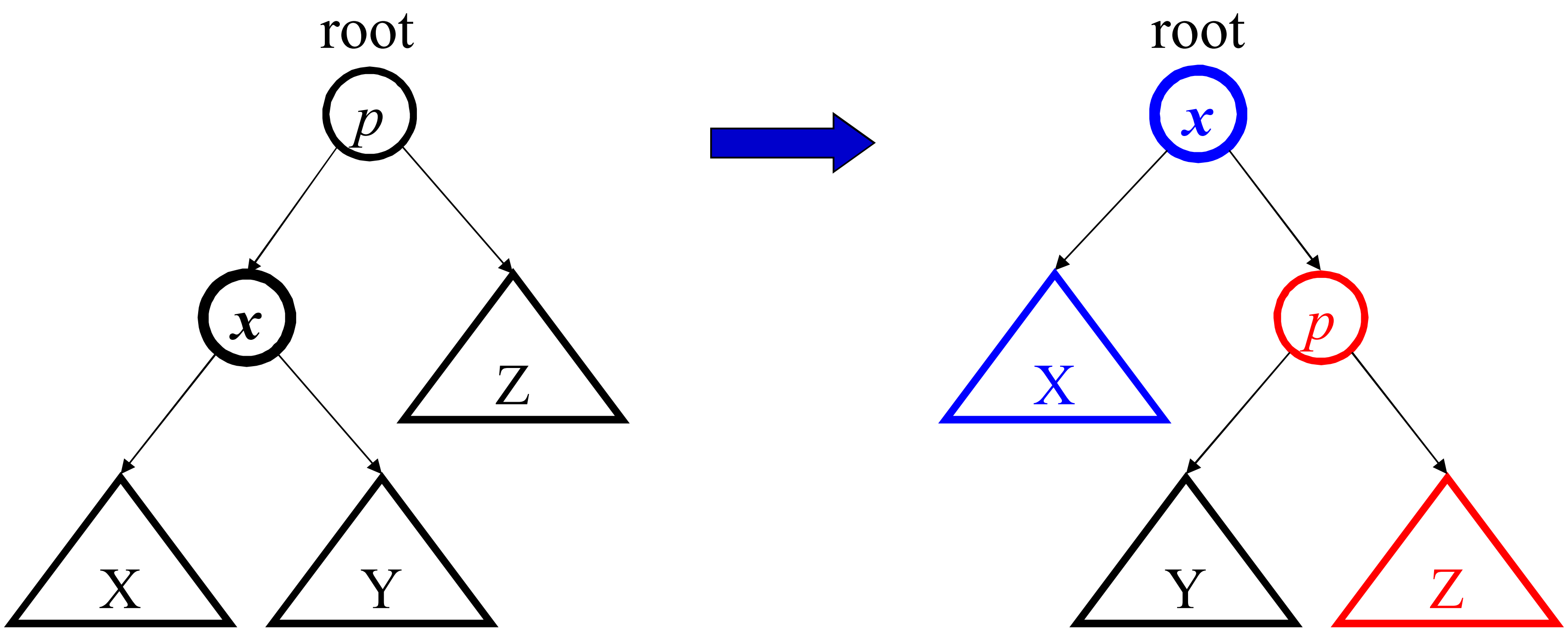
Splaying Cases

- When we access a node (x), splaying is performed on x to move it to the root.
- Splaying ensures that the recently accessed nodes are kept closer to the root and the tree remains roughly balanced.
- Depending on the node x being accessed, there are three cases:
  - x is the root
  - x is a child of the root
  - x has both parent (p) and grandparent (g)
    - Zig-zi**g** pattern: $g \rightarrow p \rightarrow x$ is left-left or right-right
    - Zig-z**a**g pattern: $g \rightarrow p \rightarrow x$ is left-right or right-left

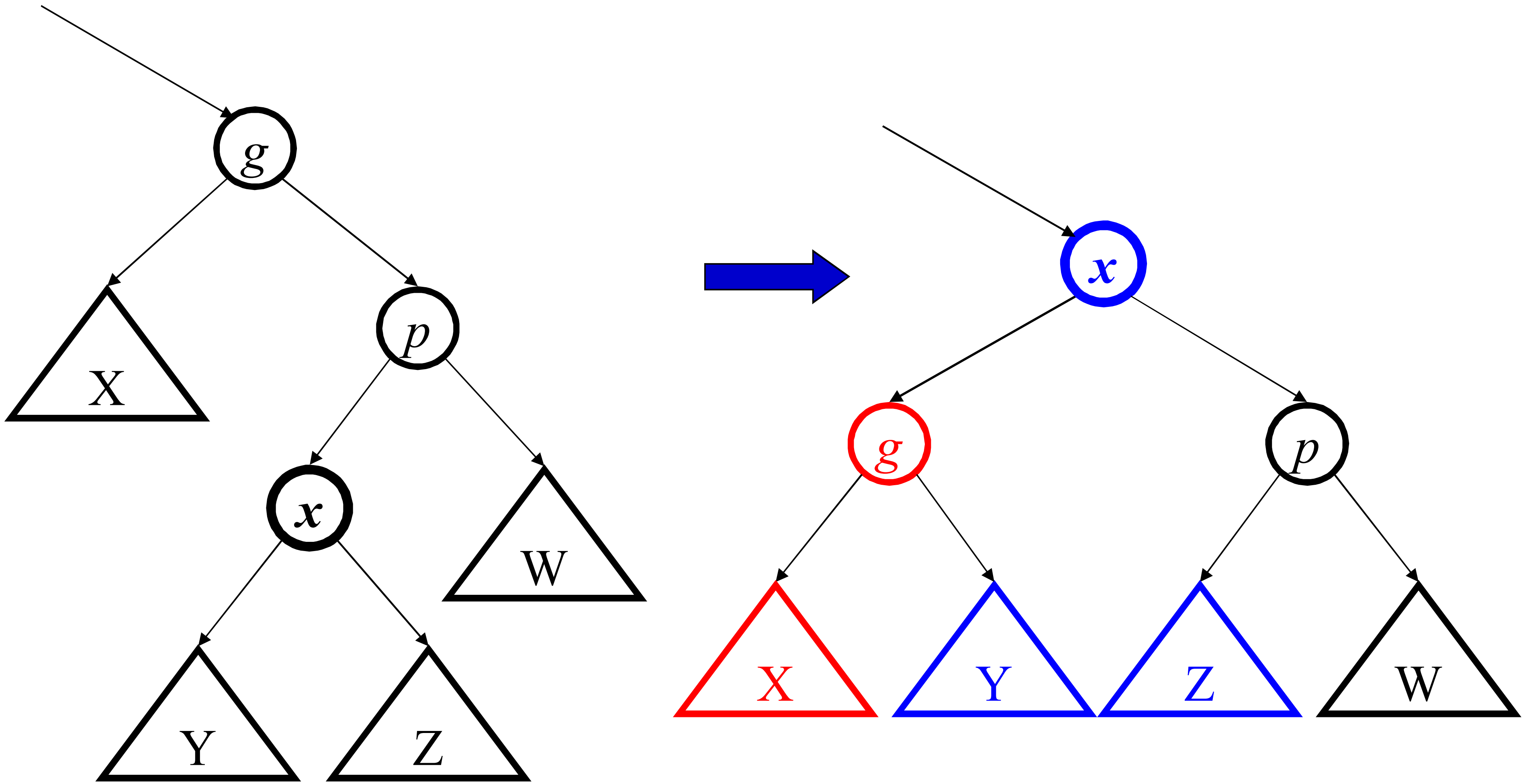
Access root:
Do nothing (that was easy!)



Access child of root:
Zig (AVL single rotation)

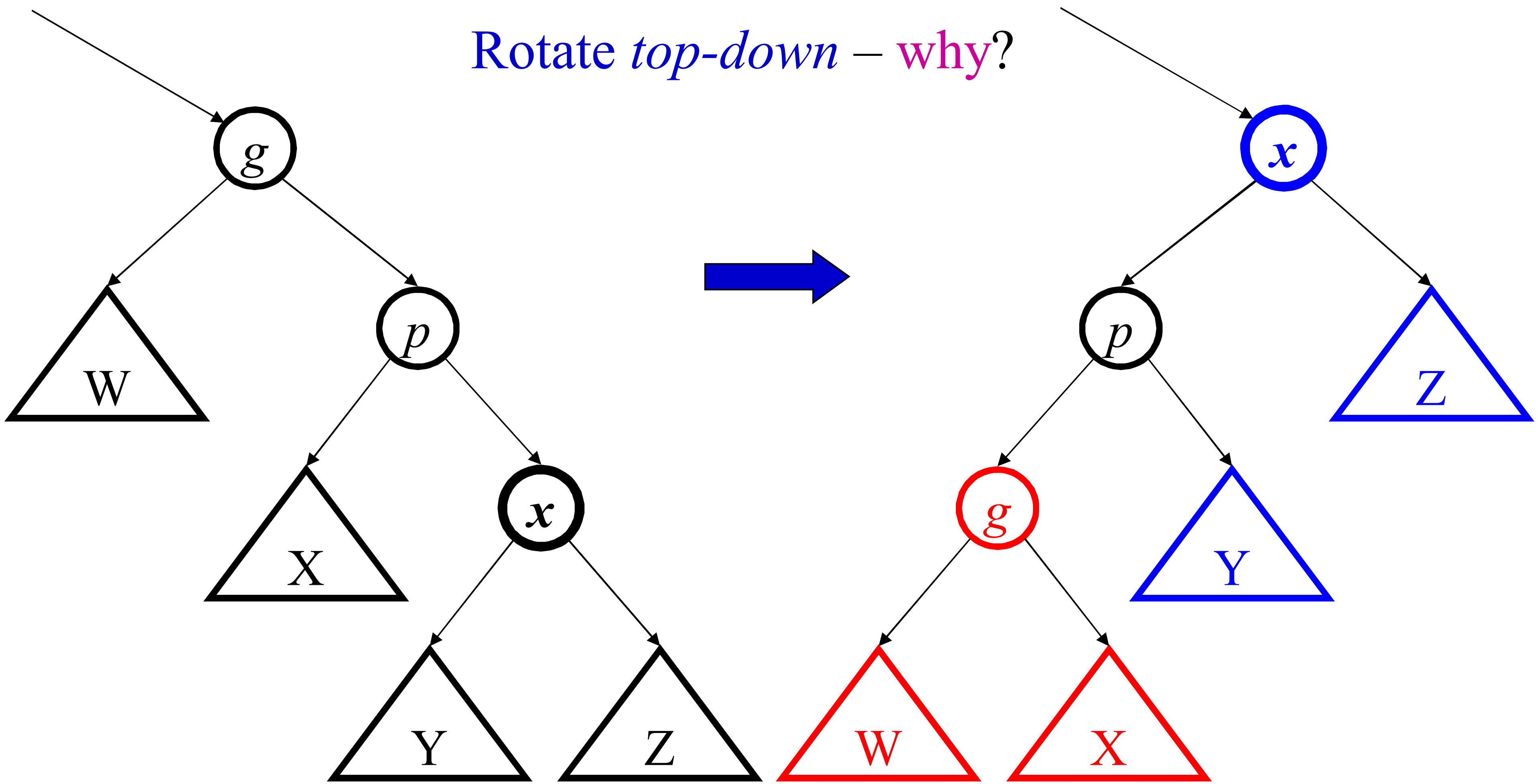


Access (LR, RL) grandchild:
Zig-Zag (AVL double rotation)

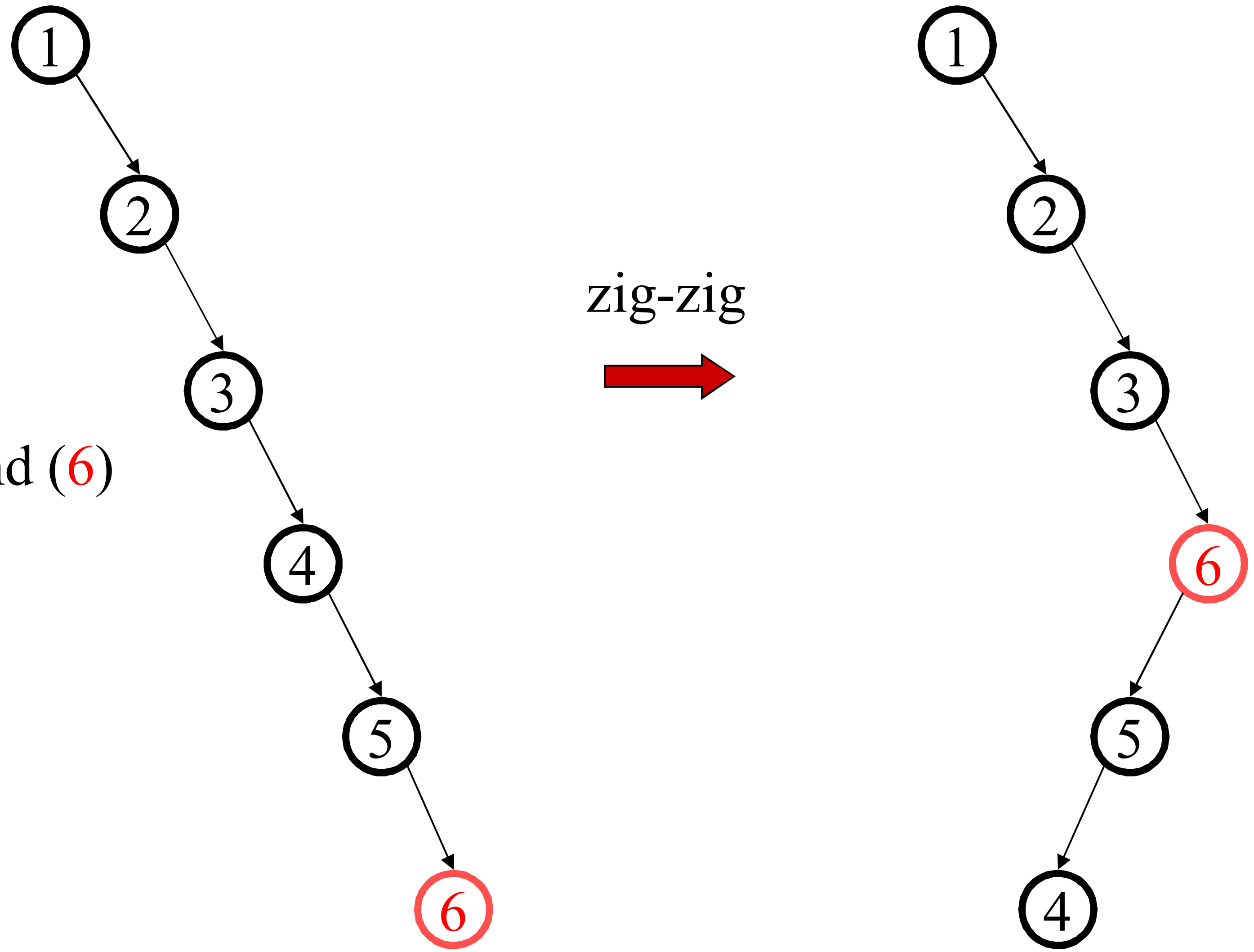


Access (LL, RR) grandchild: Zig-Zig

Rotate *top-down* – why?

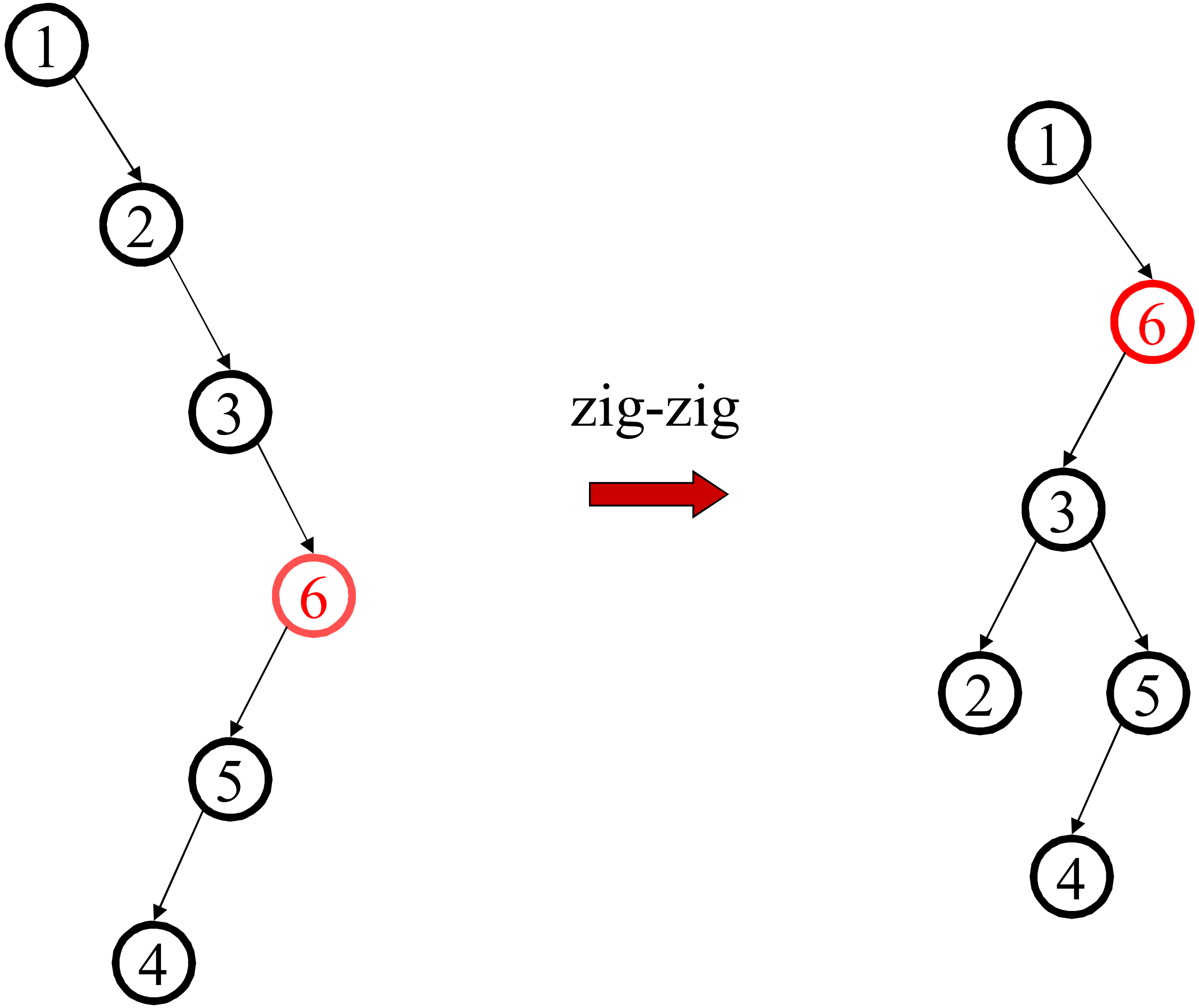


Splaying Example: Find(6)

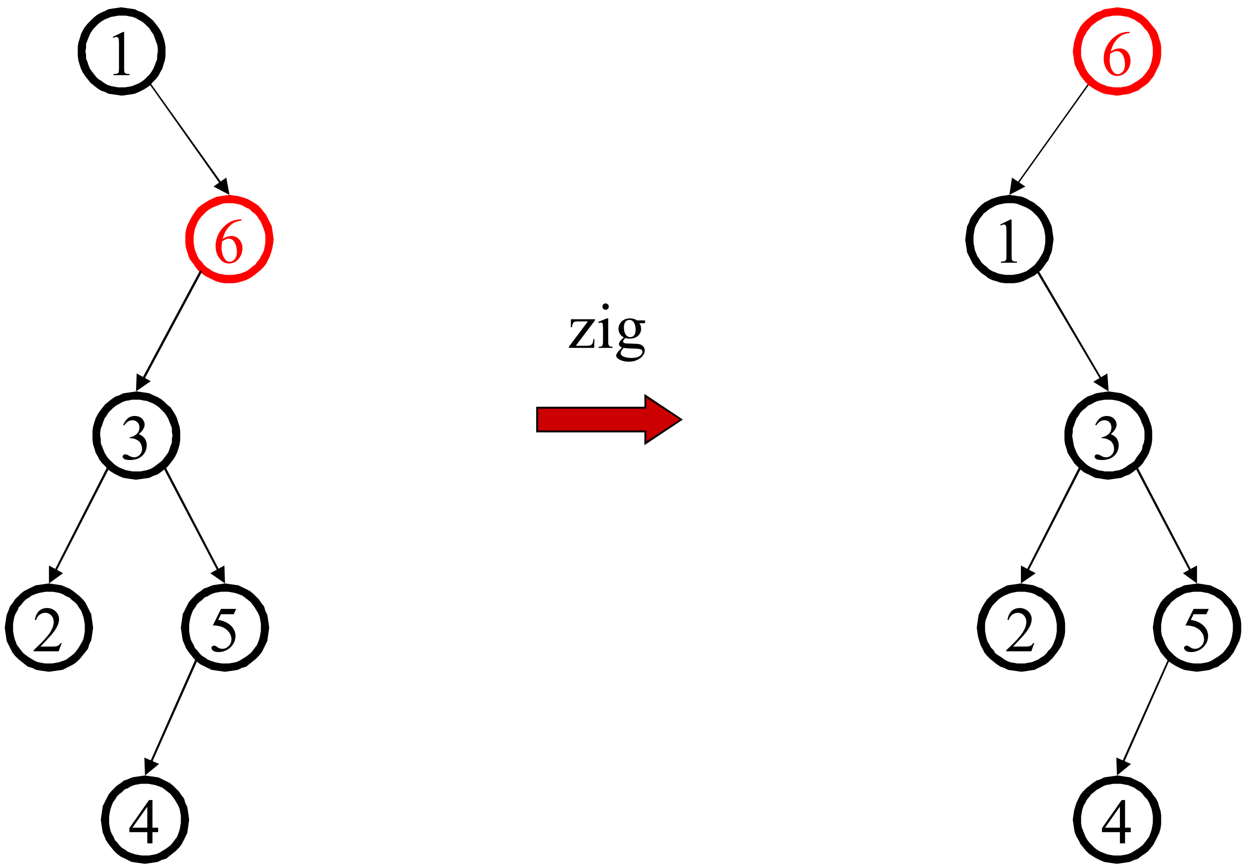


Find (6)

… still splaying …

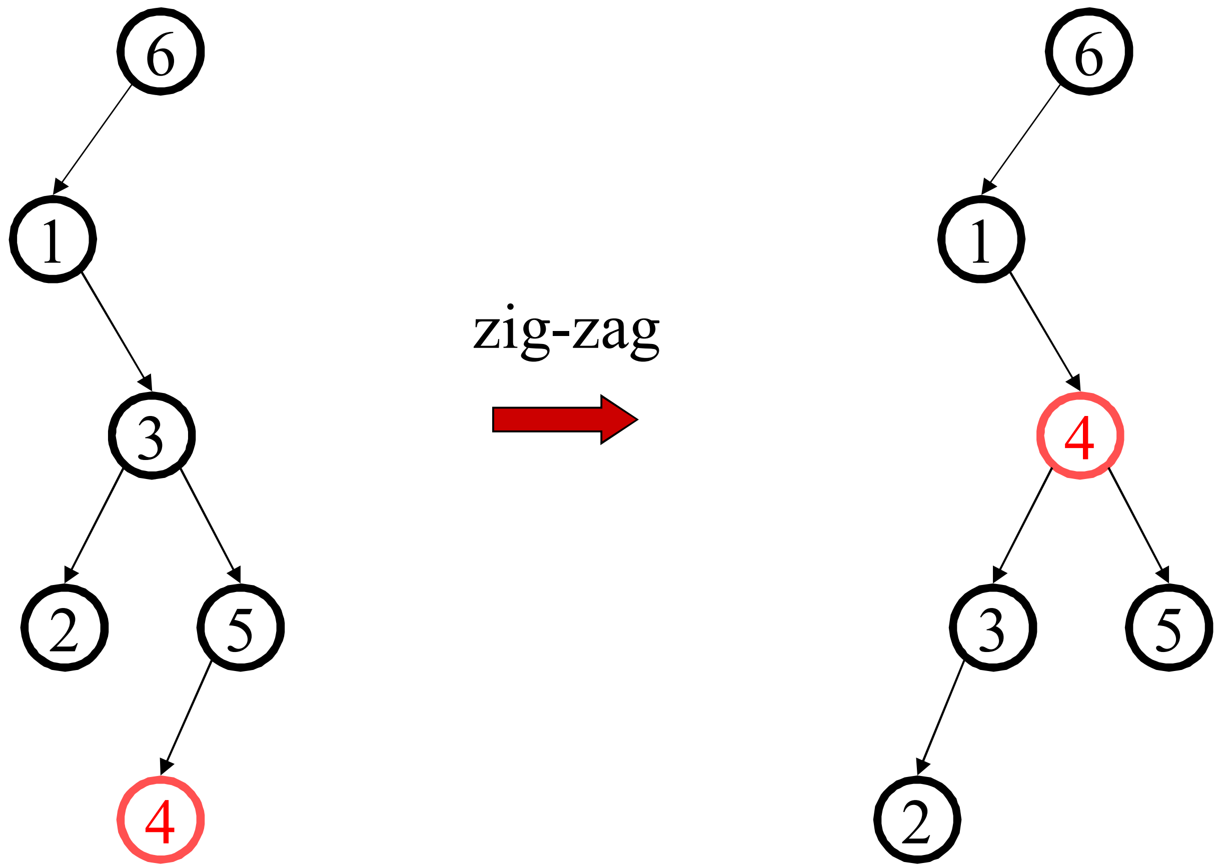


… 6 splayed out!

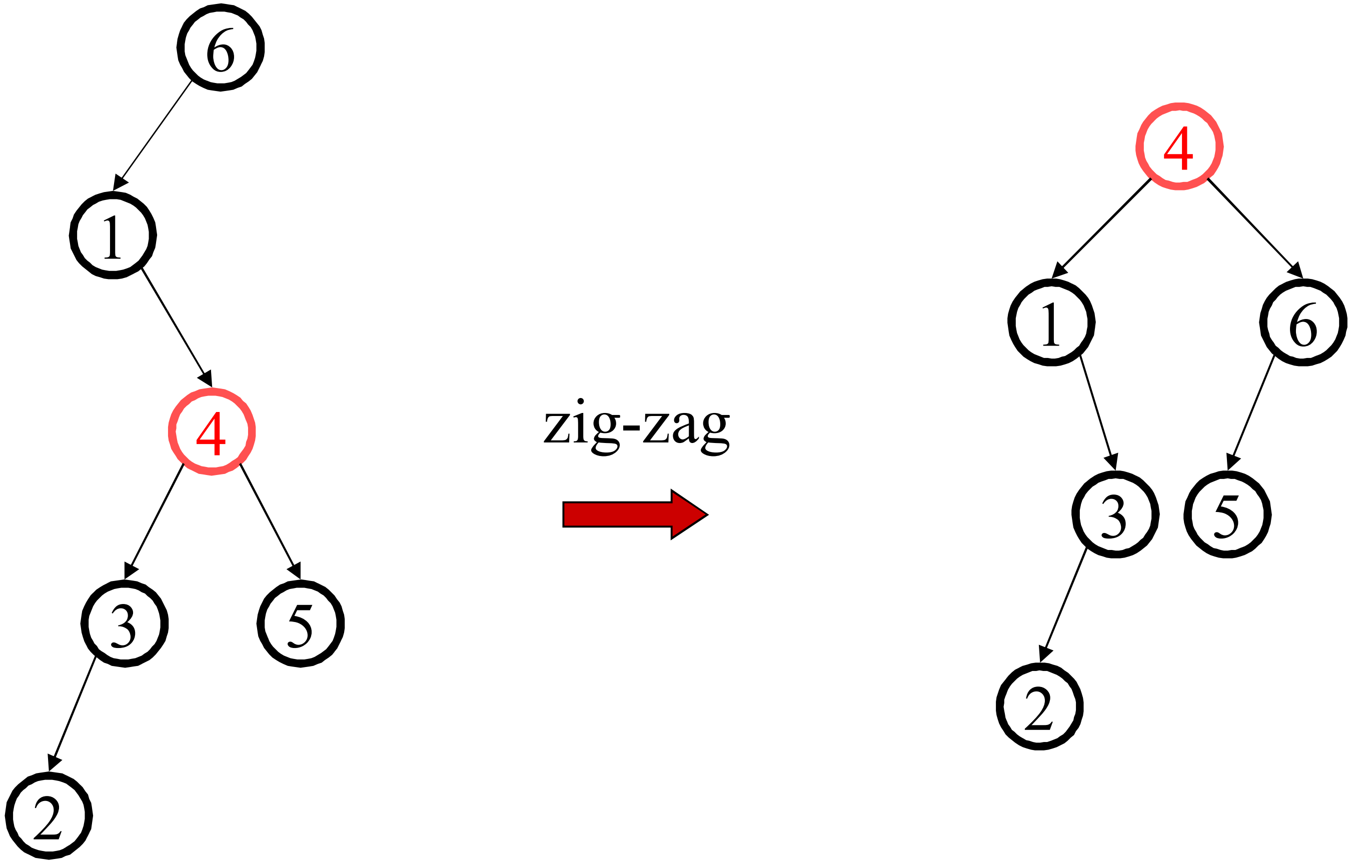


Splaying Example: Find(4)

Find(4)



… 4 splayed out!



Why Splaying Helps

- If a node x on the access path is at depth d before the splay, it's at about depth $d/2$ after the splay
  - 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

- Splaying gets amortized $O(\log n)$ performance

Splay Operation: Find

Find:

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

Splay Operation: Insert

To insert a value x into a splay tree:

- Insert x as with a normal binary search tree.
- Perform splay operation on x.
- As a result, the node x becomes the root of the tree.

Alternatively:

- Use the 'split' operation to split the tree at the value of x to two sub-trees: S and T.
- Create a new tree in which x is the root, S is its left sub-tree and T its right sub-tree.

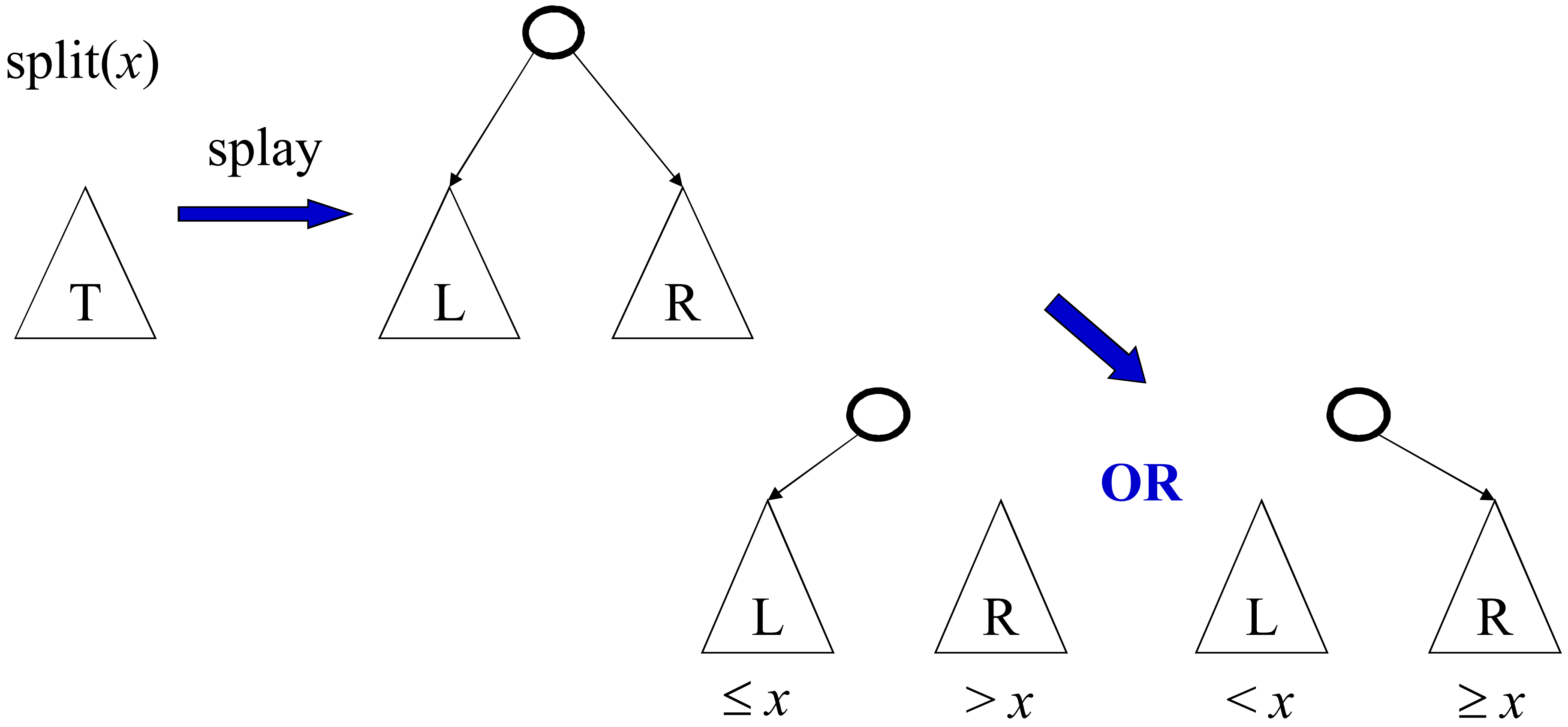
Splitting in Splay Trees

- Split(T, x) creates two BSTs L and R:
  - all elements of T are in either L or R ($T = L \cup R$)
  - all elements in L are $\leq x$
  - all elements in R are $\geq x$
  - L and R share no elements ($L \cap R = \emptyset$)

How can we split in splay trees?

- We have the splay operation
- We can find x or the parent of x where x should be
- We can splay it to the root
- Now, what's true about the left subtree of the root?
- And the right subtree?

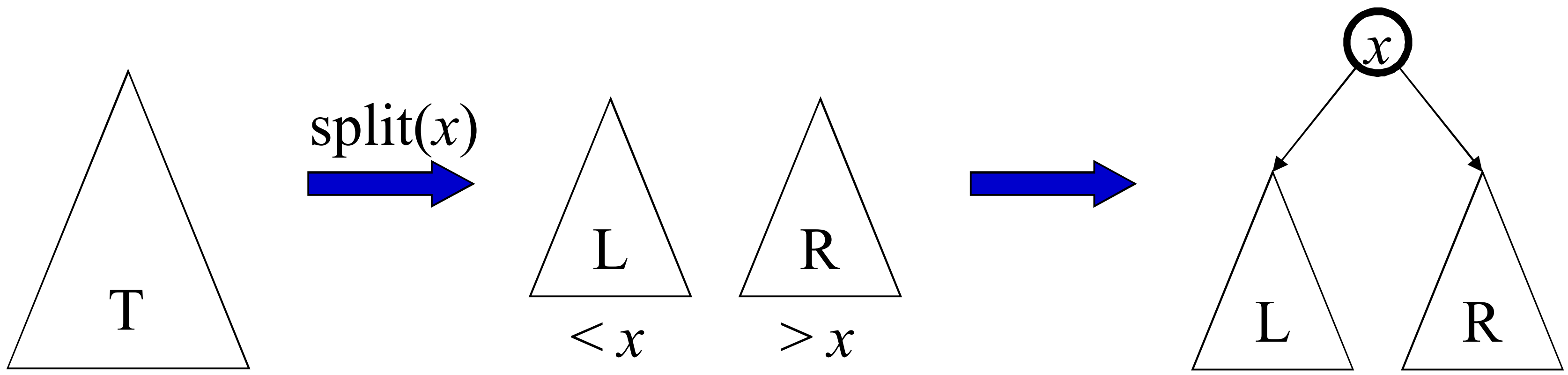
Splitting in Splay Trees



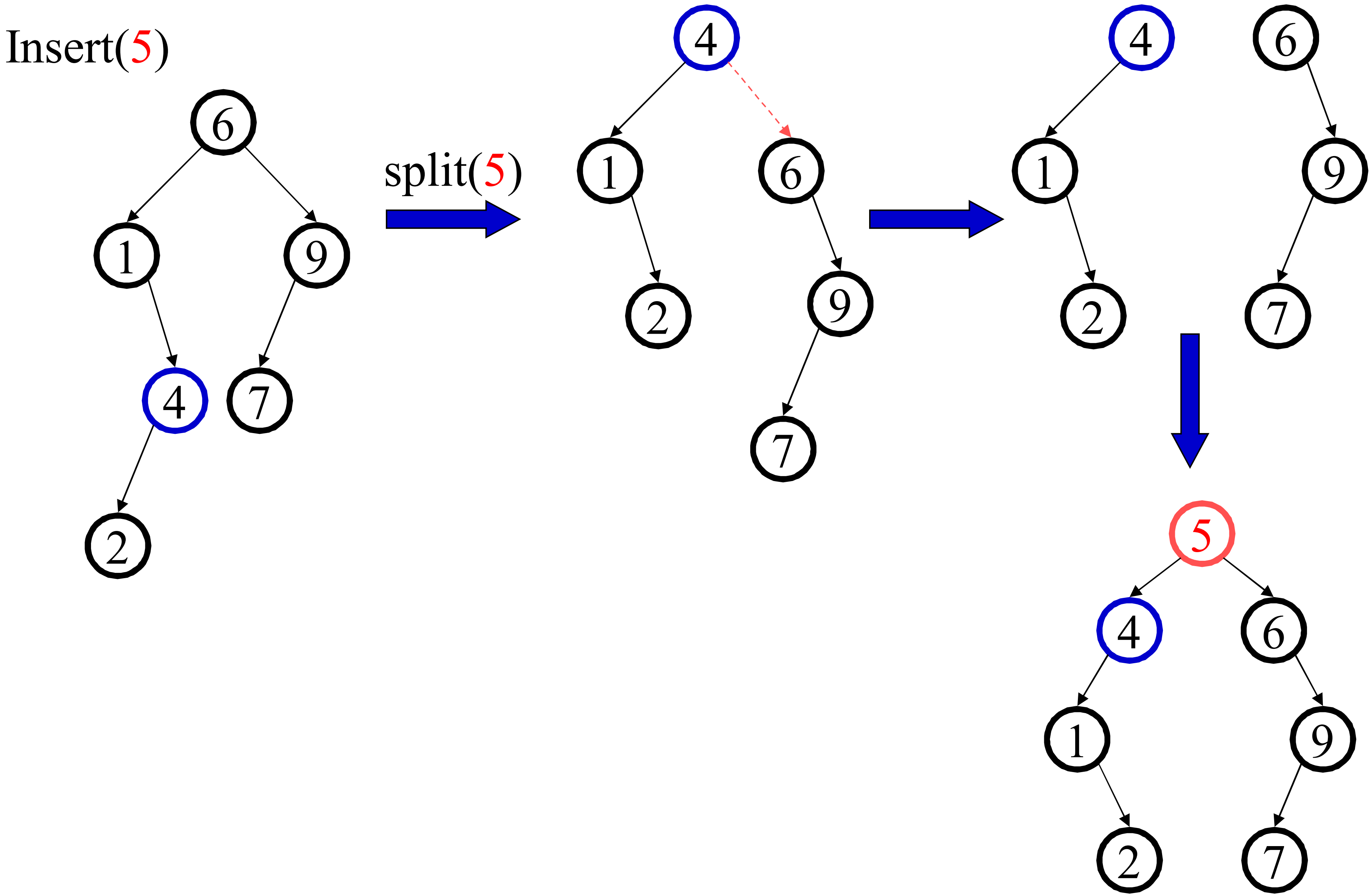
Back to Insert

To insert a value x into a splay tree:

- Use the 'split' operation to split the tree at the value of x to two sub-trees: S and T.
- Create a new tree in which x is the root, S is its left sub-tree and T its right sub-tree.



Insert Example



Splay Operation: Delete

To delete a value x from a splay tree:

Use the same method as with a binary search tree:

- If x has two children:
  - Swap its value with that of either its in-order predecessor or its in-order successor
  - Remove that node instead.

    In this way, deletion is reduced to the problem of removing a node with 0 or 1 children.
- Now, splay the parent of the removed node to the top of the tree.

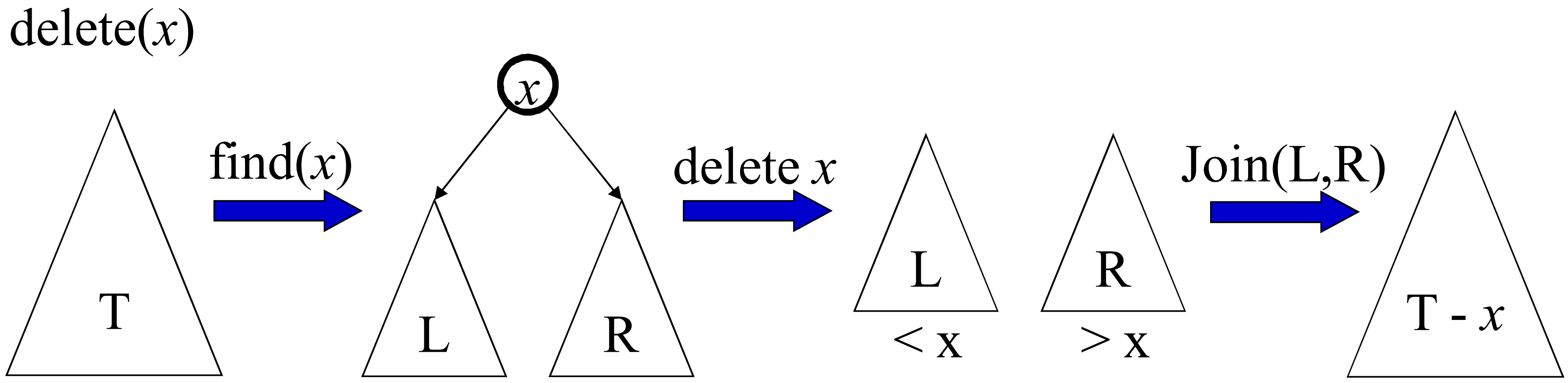
Alternatively:

- Splay the node x, i.e. bring it to the root and then deleted it.
- Join the two sub-trees using a 'join' operation.

Splay Operation: Delete

To delete a value *x* from a splay tree:

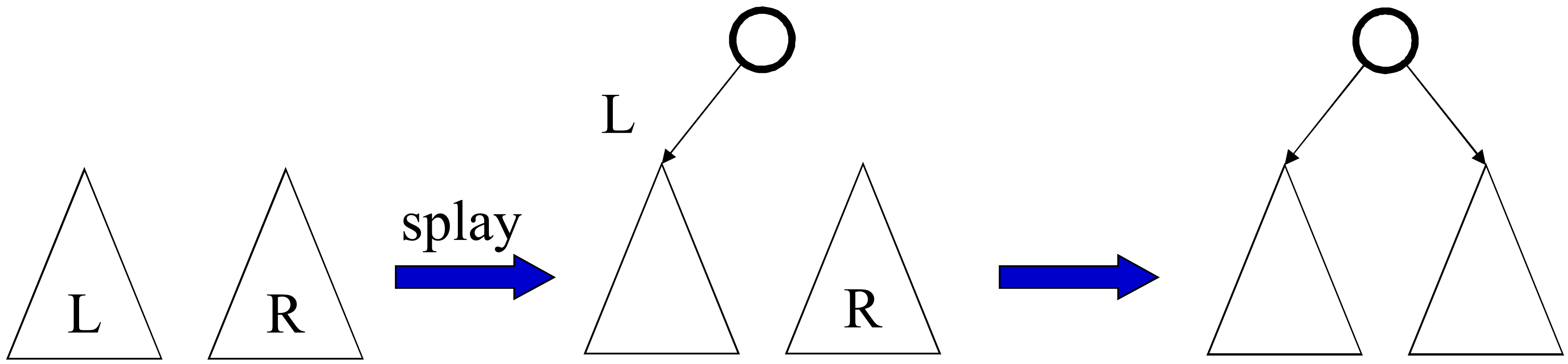
- Splay the node *x*, i.e. bring it to the root and then deleted it.
- Join the two sub-trees using a 'join' operation.



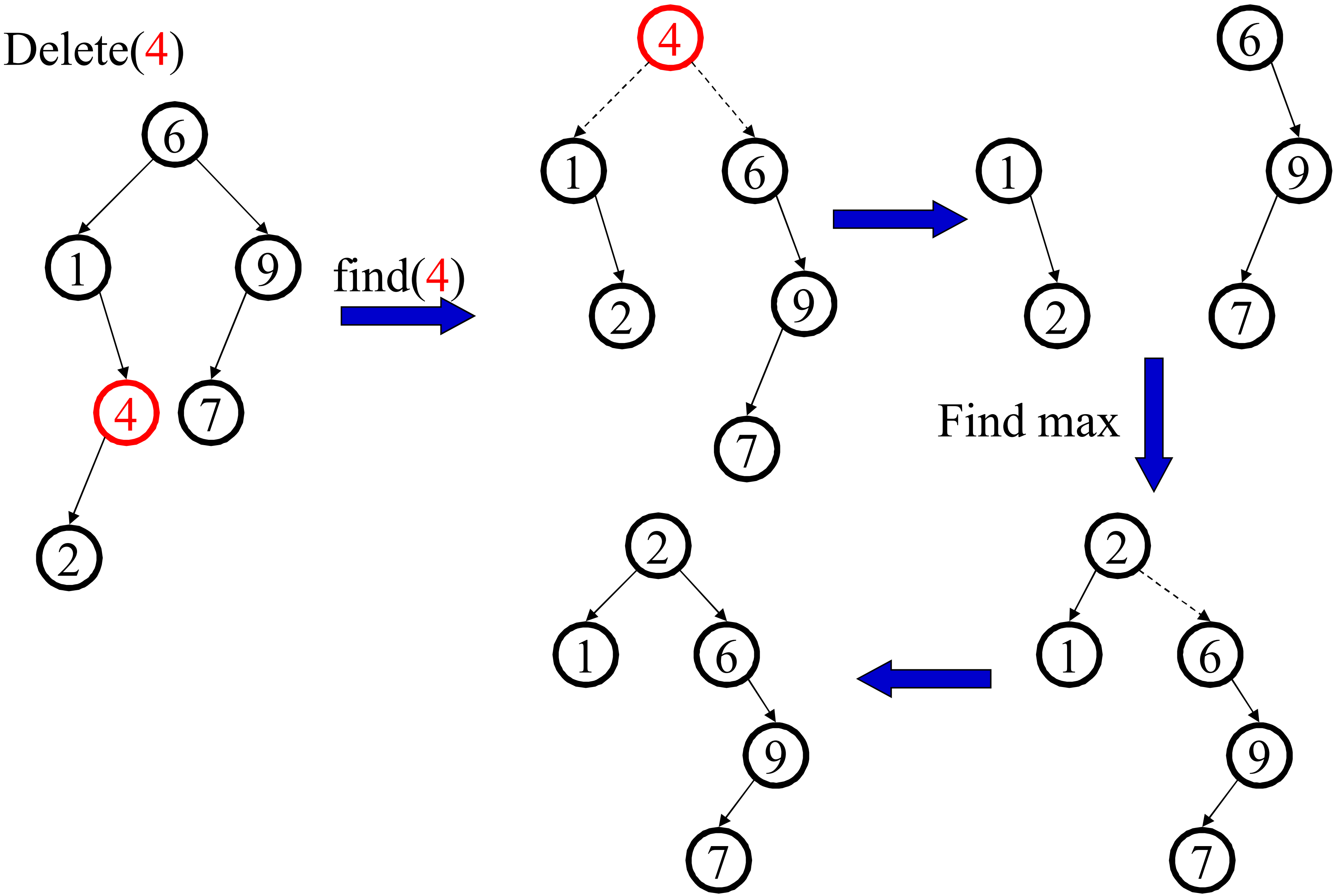
Joining in Splay Trees

Join(L, R): given two trees such that L < R, merge them.

Splay on the maximum element in L, then attach R.



Delete Example



Splay Tree Summary

- Can be shown that any m consecutive operations starting from an empty tree take at most $O(m \log n)$ time

  → All splay tree operations run in amortized $O(\log n)$ time
- Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.
- A splay tree can change even with read-only operations like search
- 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)