Splay Trees

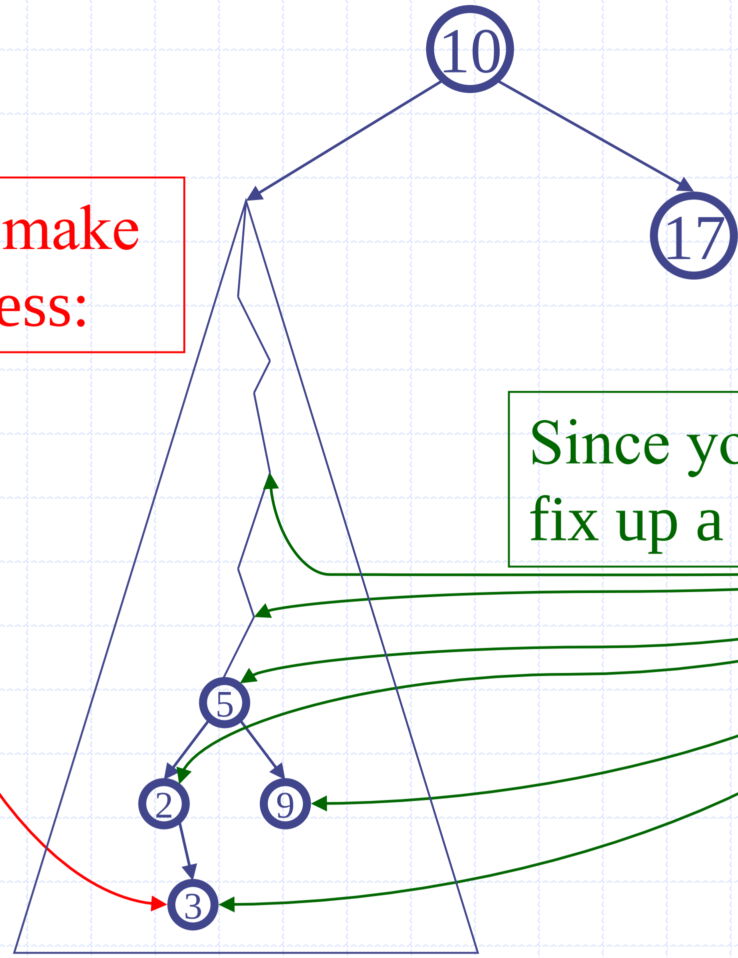
David Kaplan

Dept of Computer Science & Engineering

Motivation for Splay Trees

- Problems with AVL Trees
  - extra storage/complexity for height fields
  - ugly delete code
- Solution: splay trees
  - blind adjusting version of AVL 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



You're forced to make a really deep access:

Since you're down there anyway, fix up a lot of deep nodes!

Splaying Cases

Node being accessed (n) is:

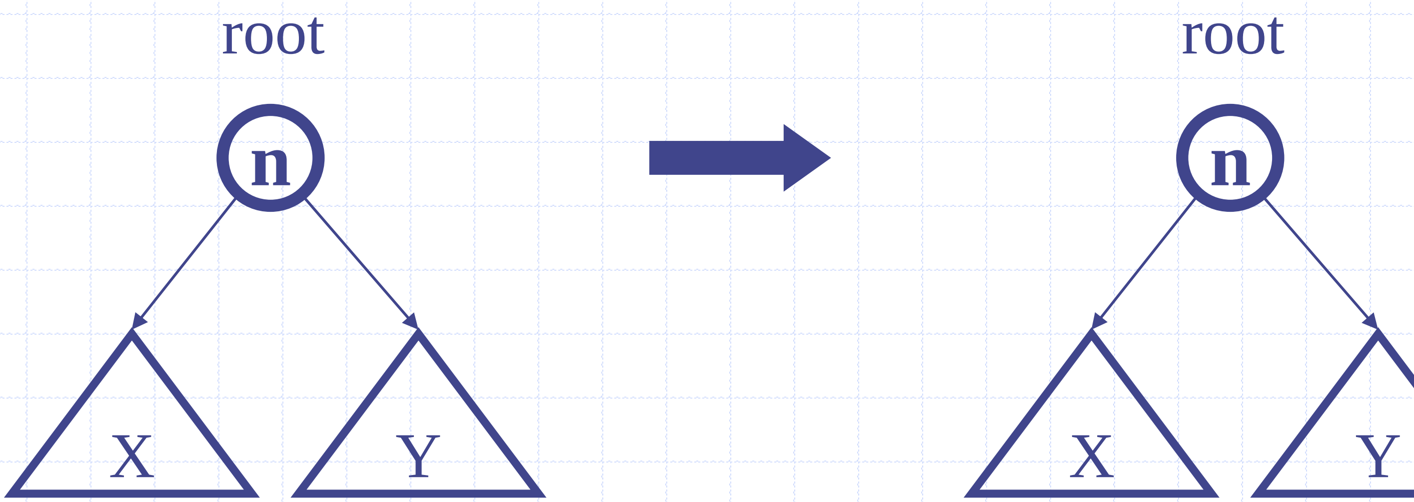
- Root
- Child of root
- Has both parent (p) and grandparent (g)

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

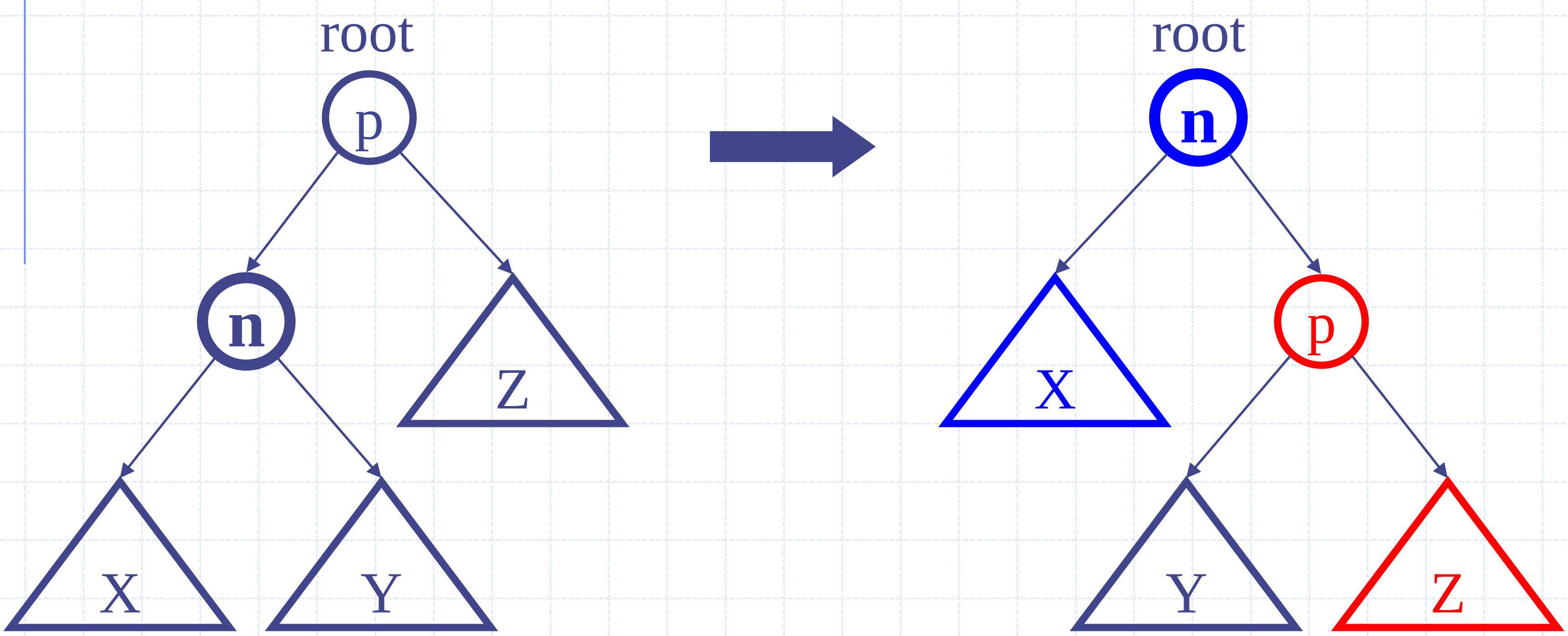
  Zig-zag pattern: g → p → n is left-right or right-left

Access root:

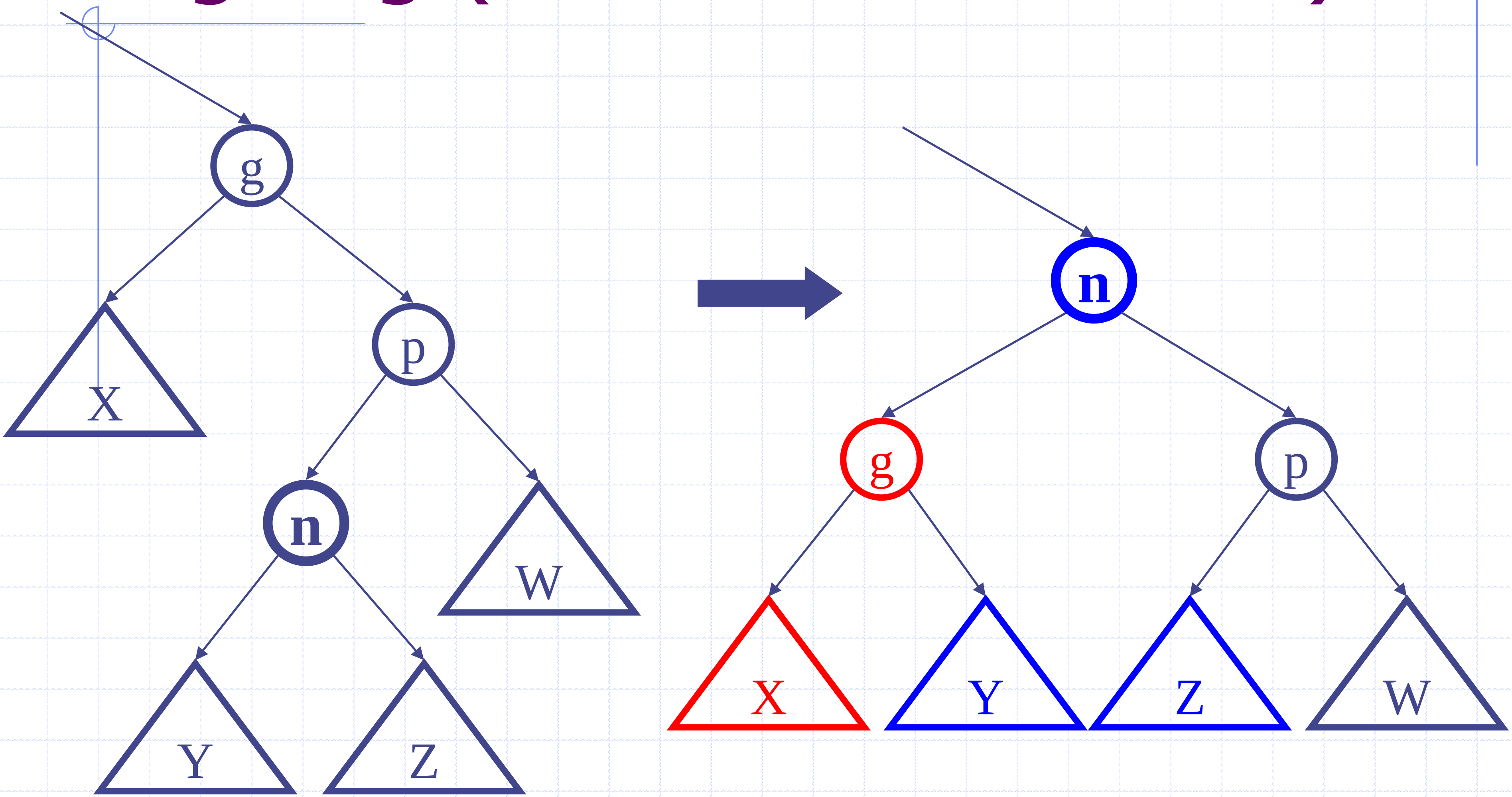
Do nothing (that was easy!)



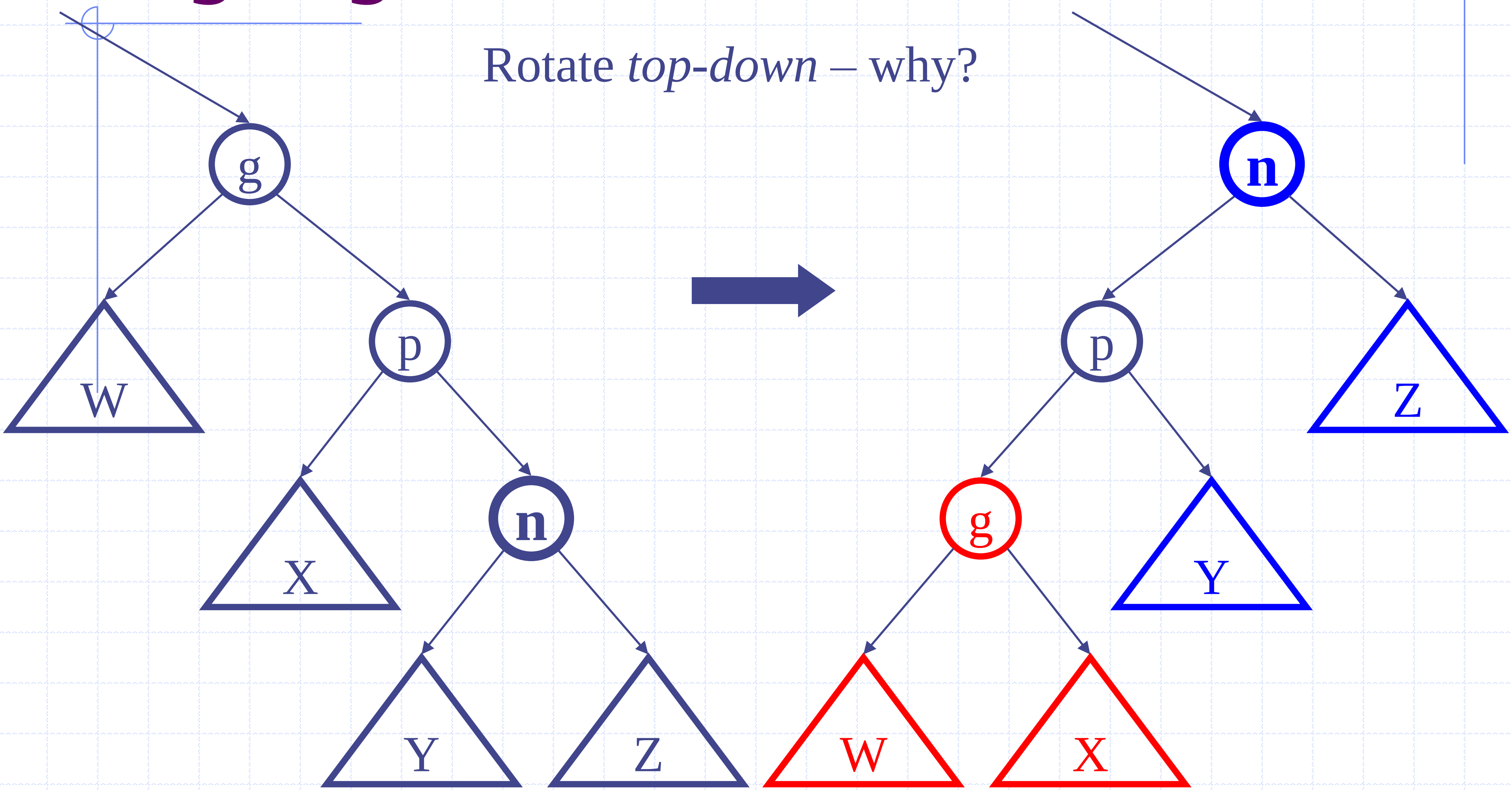
Access child of root:
Zig (AVL single rotation)



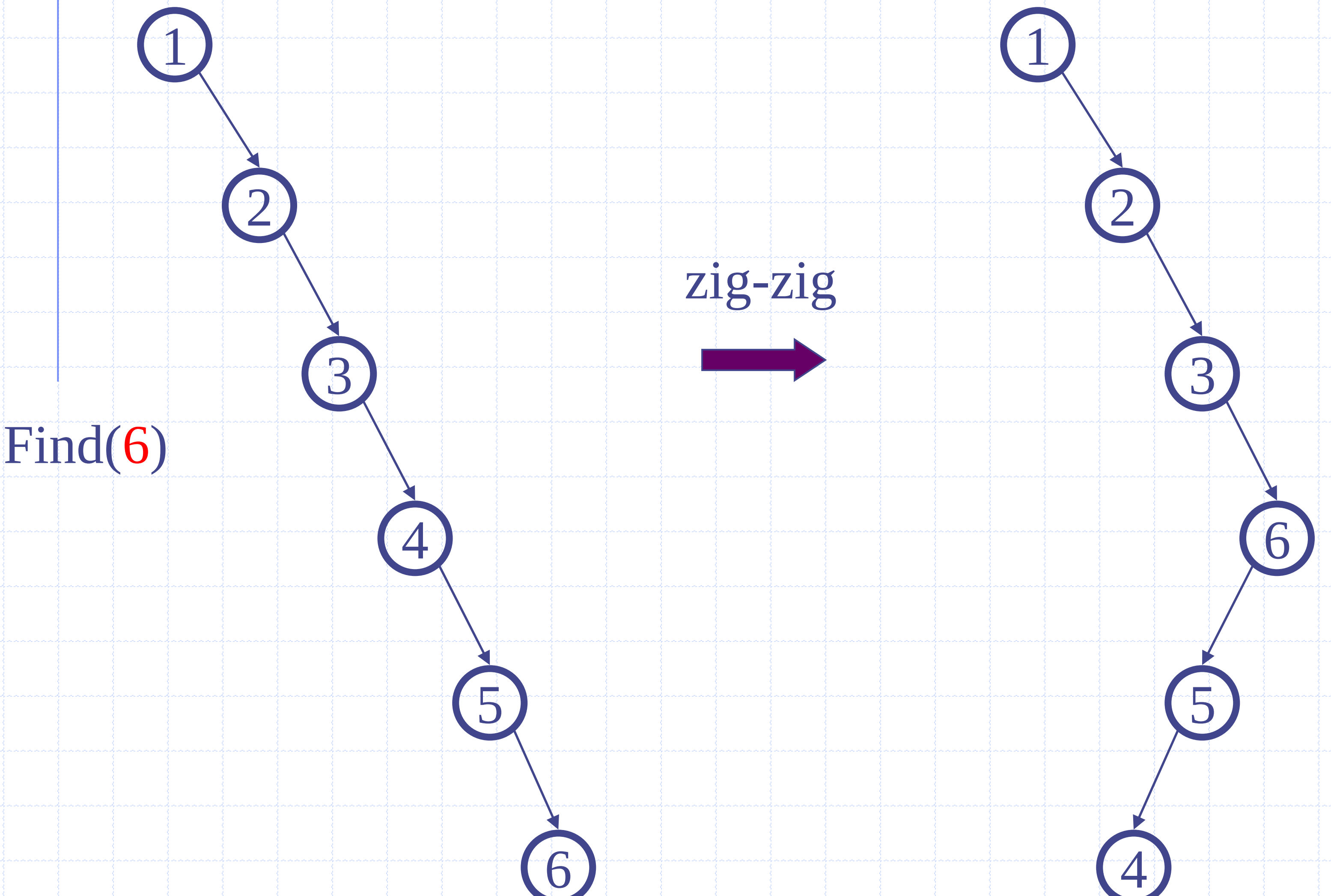
Access (LR, RL) grandchild:
Zig-Zag (AVL double rotation)



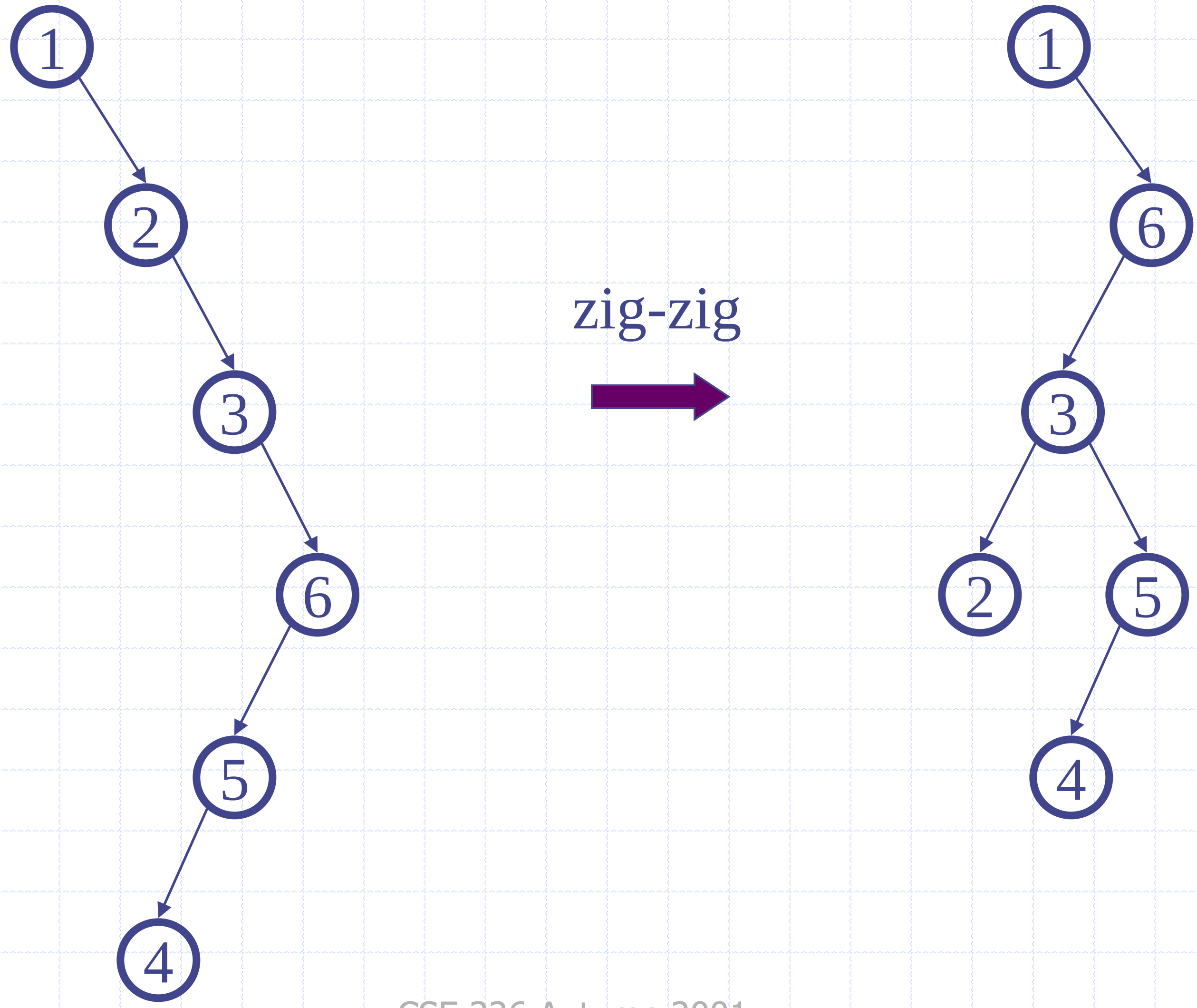
Access (LL, RR) grandchild: Zig-Zig



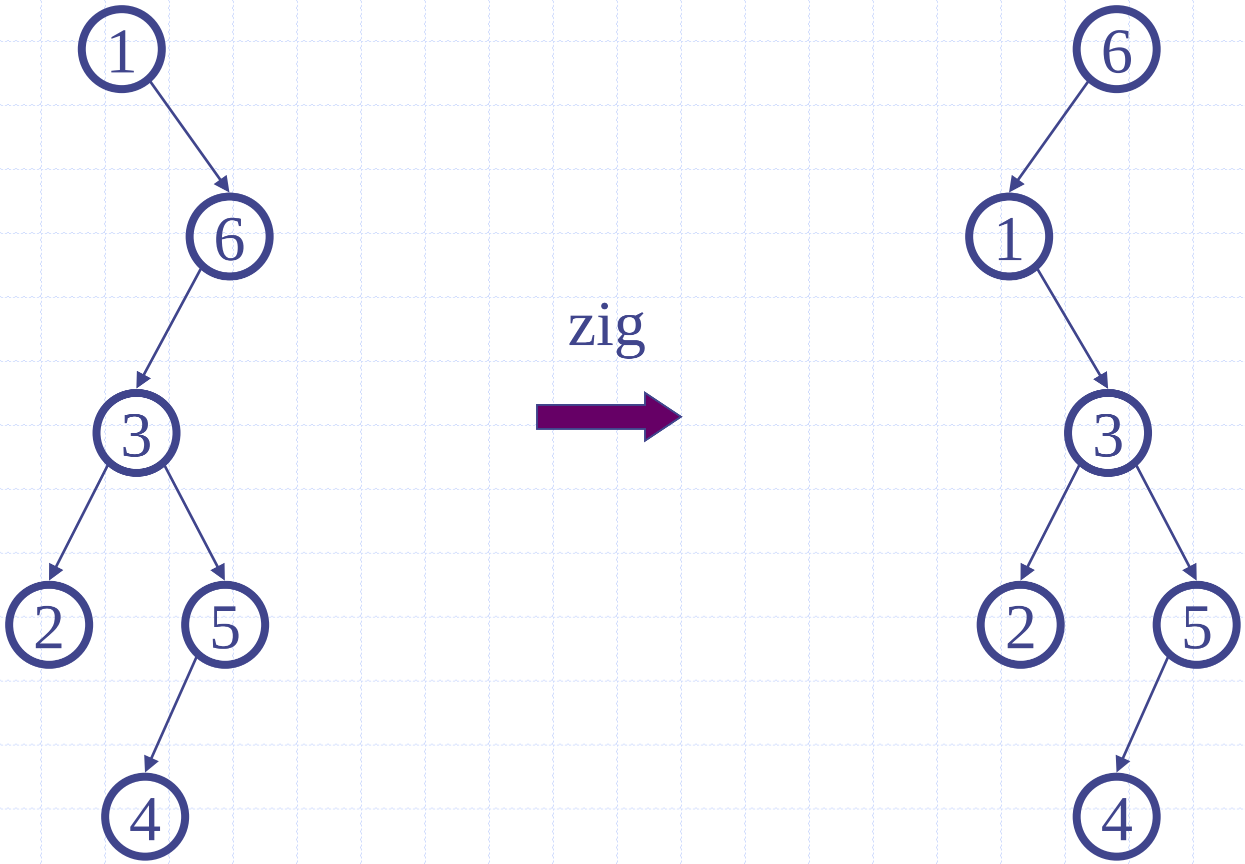
Splaying Example: Find(6)



… still splaying …

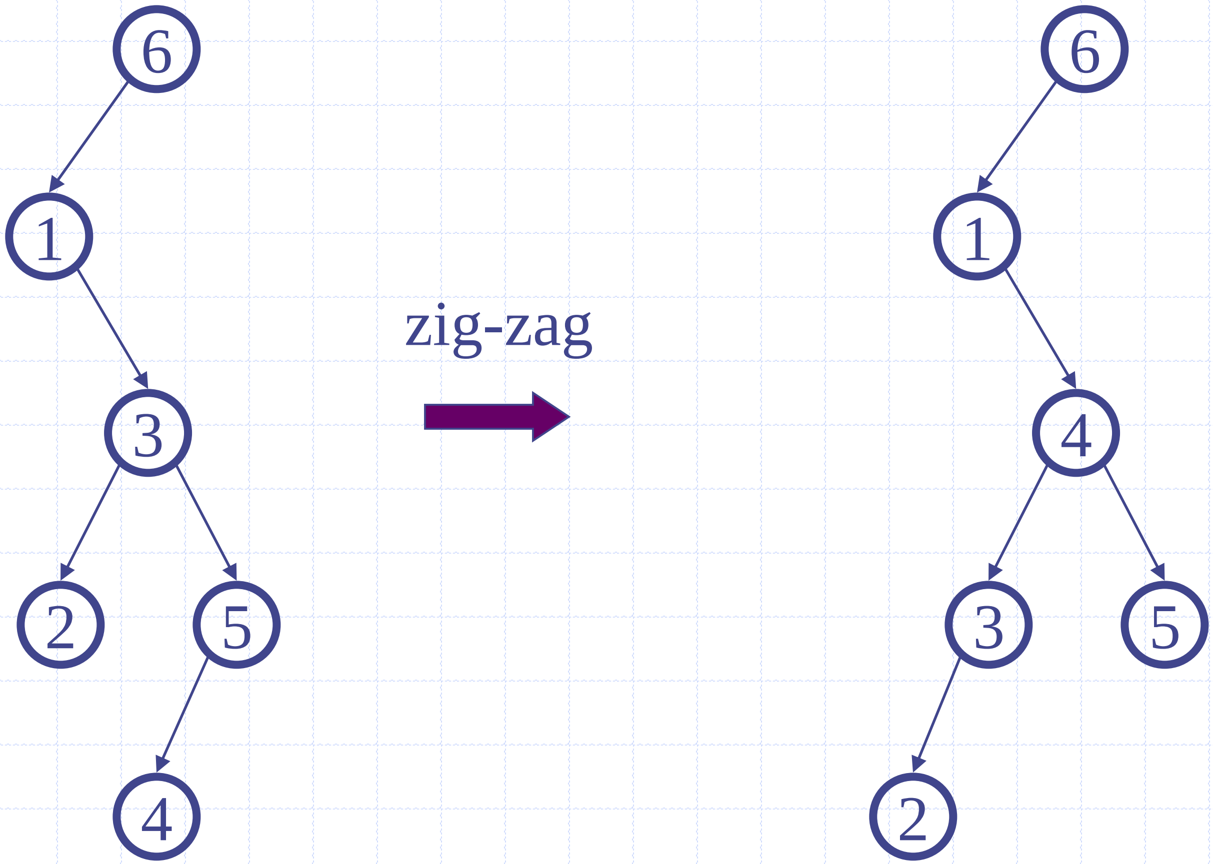


… 6 splayed out!

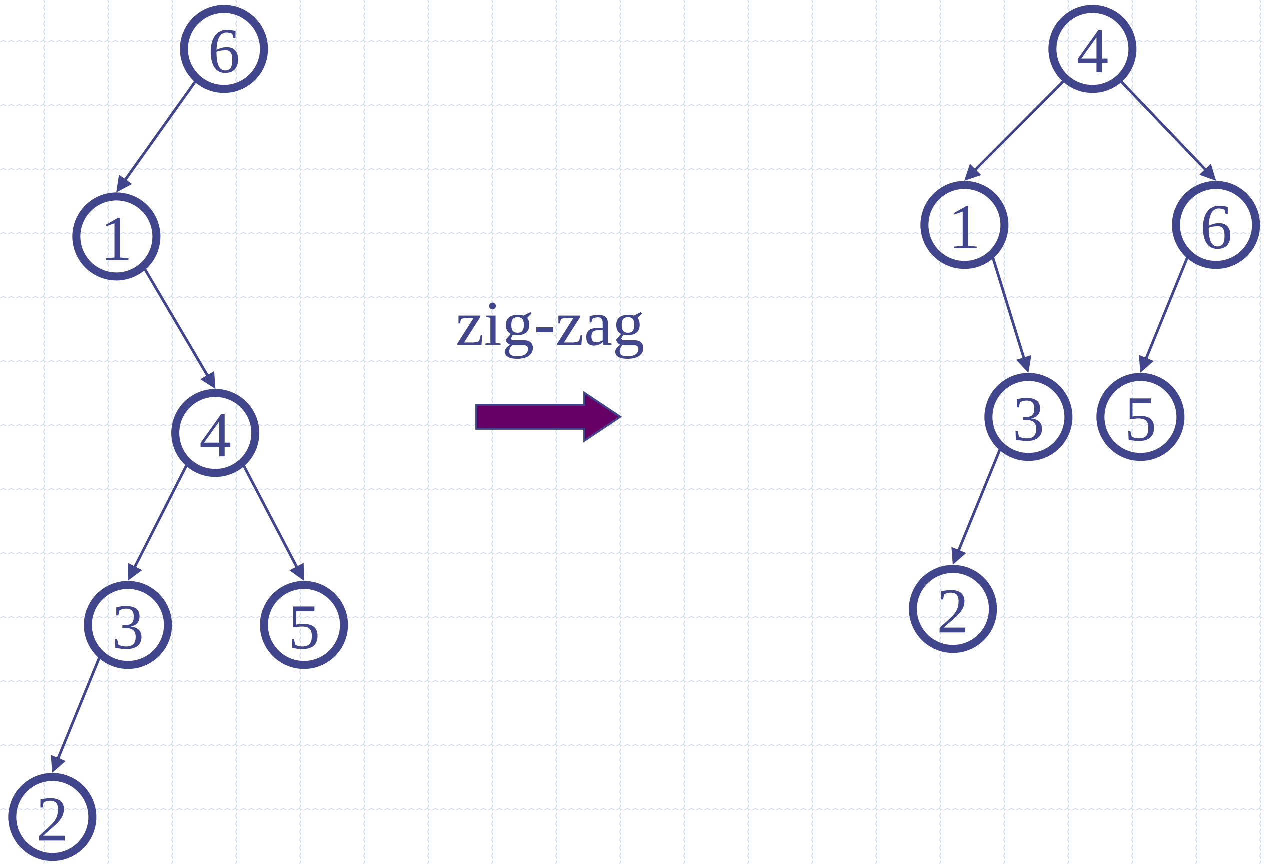


Splay it Again, Sam!
Find (4)

Find(4)



… 4 splayed out!



Why Splaying Helps

- If a node *n* 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. (Maybe not now, but soon, and for the rest of the operations.)

Splay Operations: Find

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

Splay Operations: Insert

- Ideas?
- Can we just do BST insert?

Digression: Splitting

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

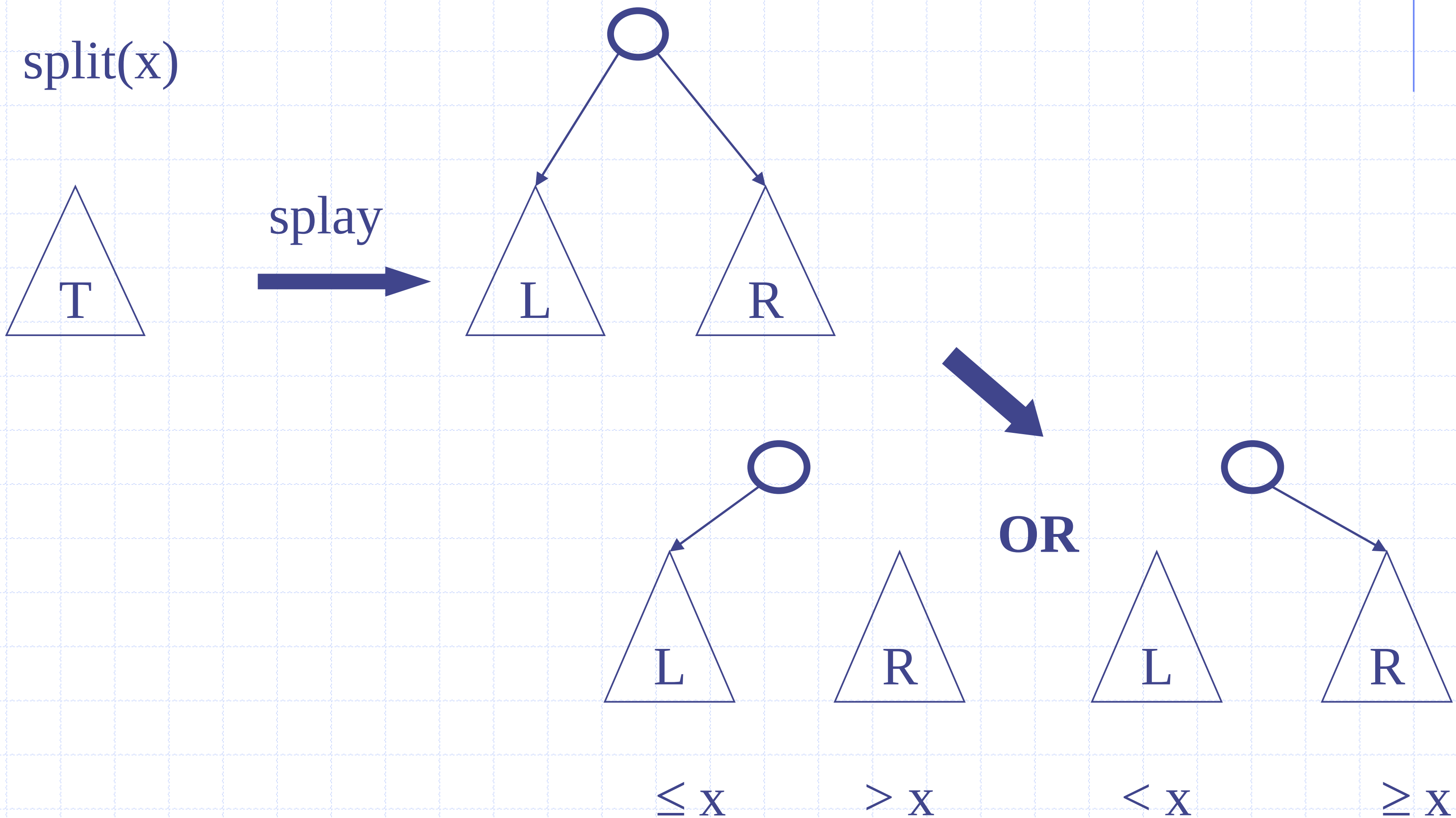
Splitting in Splay Trees

How can we split?

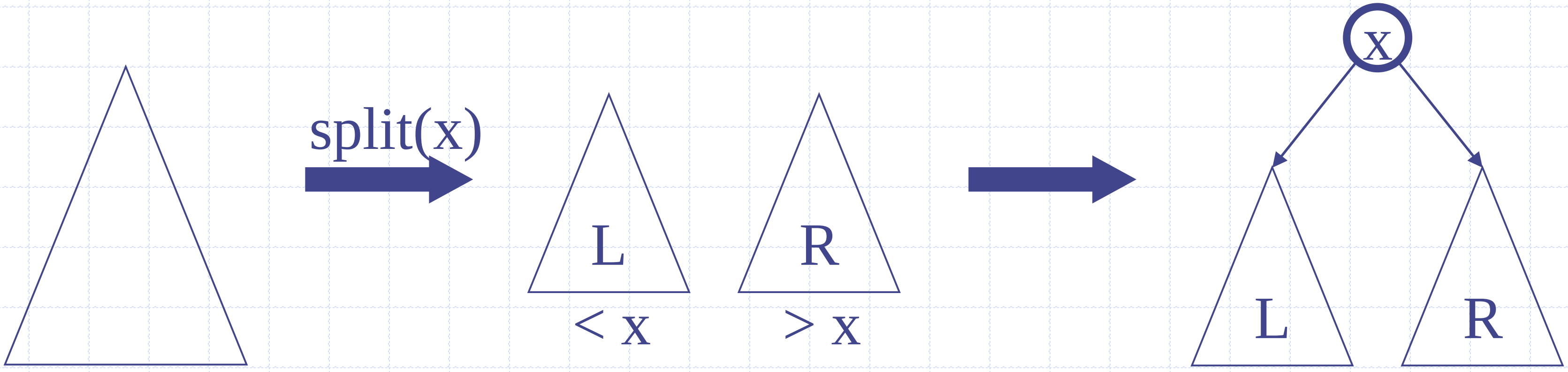
- We have the splay operation.
- We can find x or the parent of where x should be.
- We can splay it to the root.
- Now, what's true about the left subtree of the root?
- And the right?

Split

split(x)

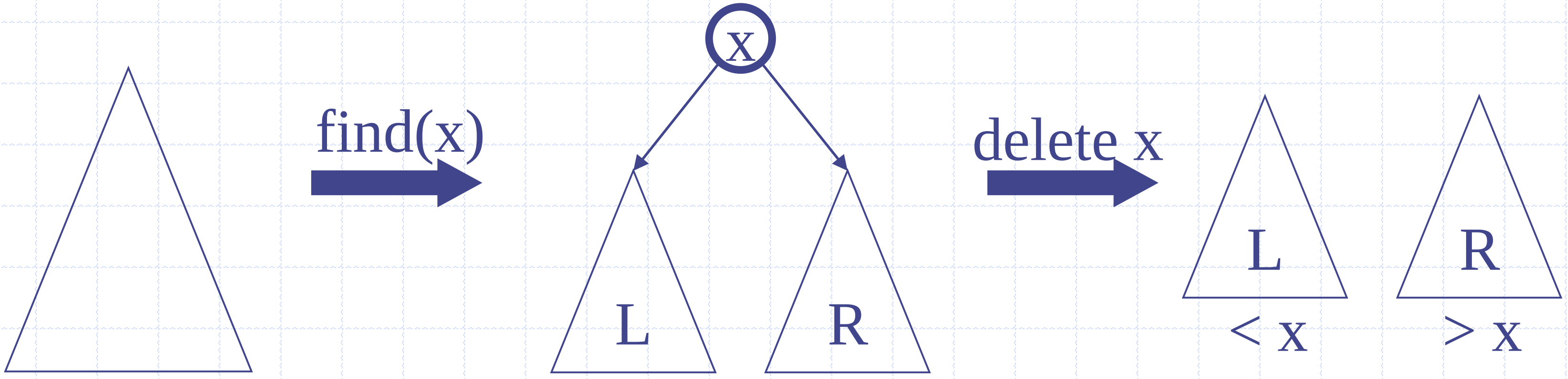


Back to Insert




```
void insert(Node *& root, Object x)
{
  Node * left, * right;
  split(root, left, right, x);
  root = new Node(x, left, right);
}
```

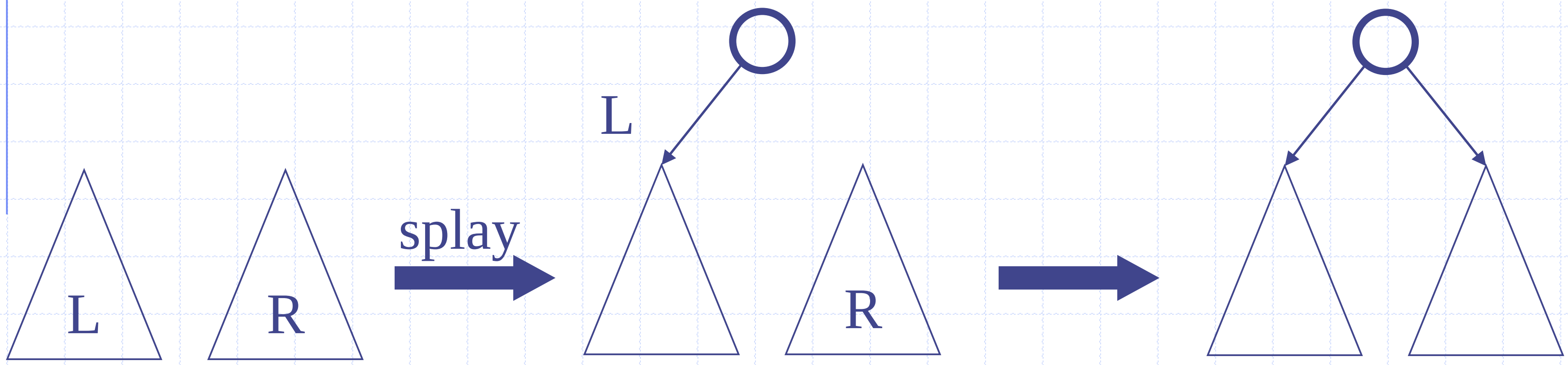
Splay Operations: Delete



Now what?

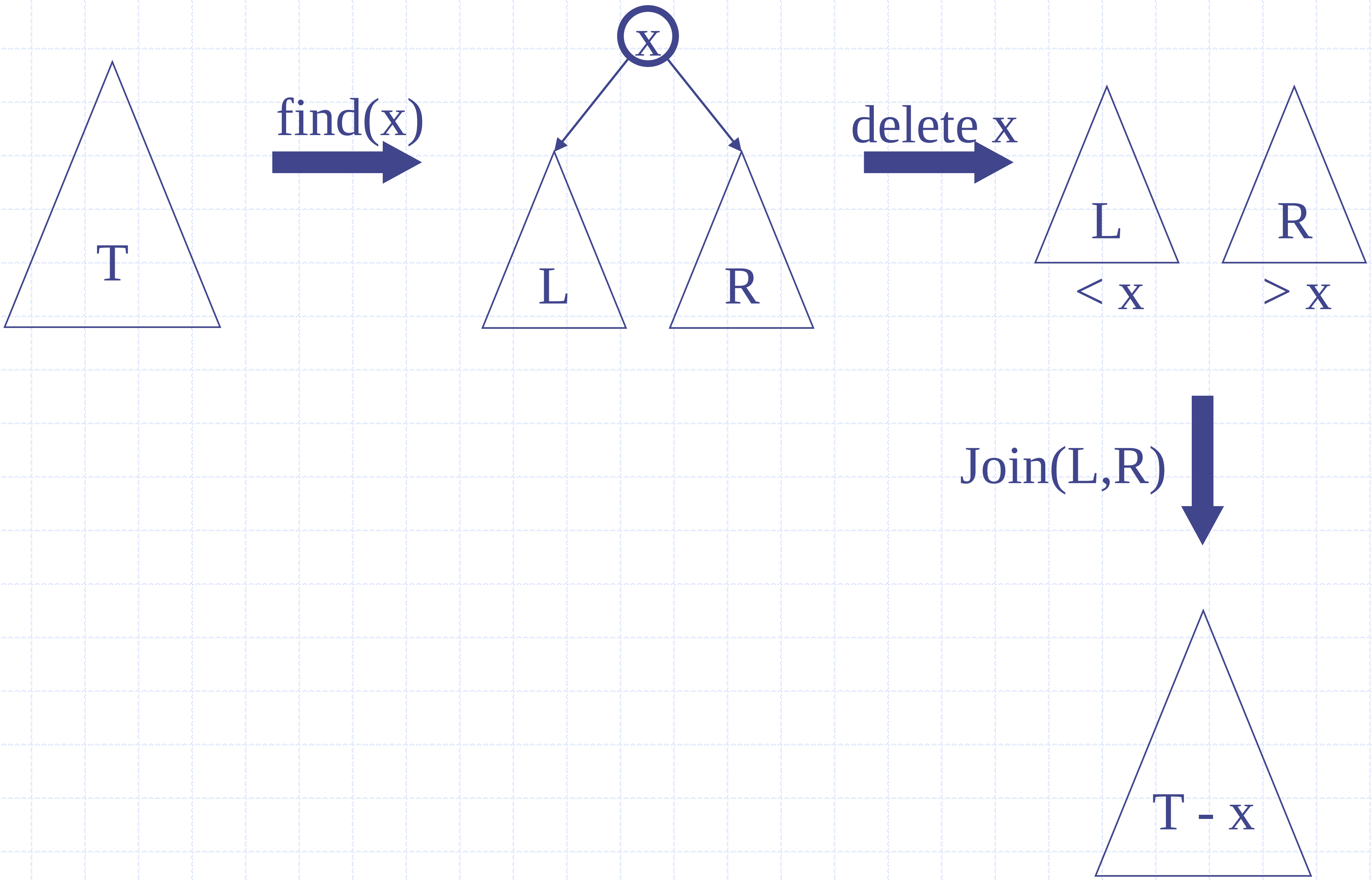
Join

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

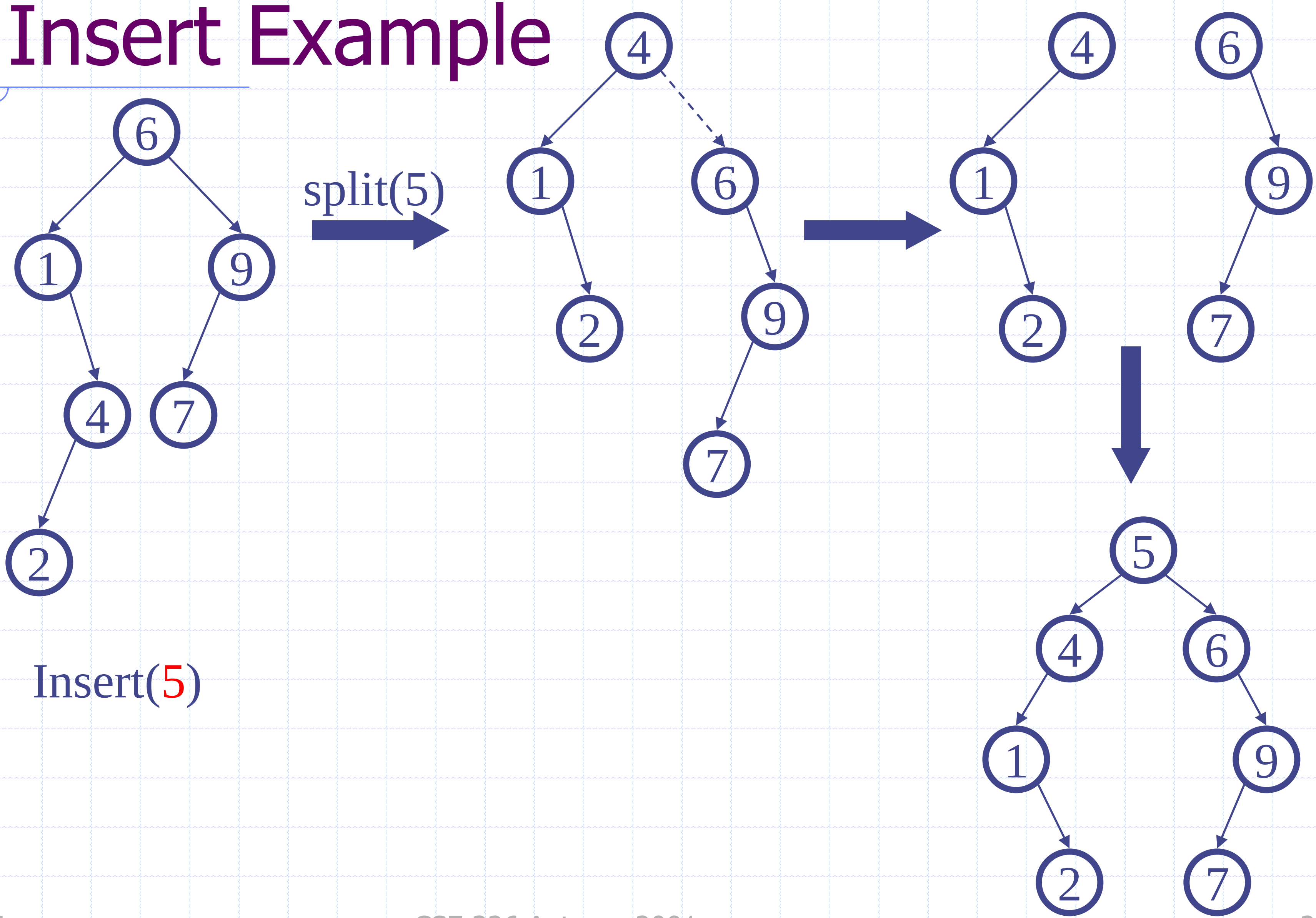


Splay on the maximum element in L, then attach R

Delete Completed

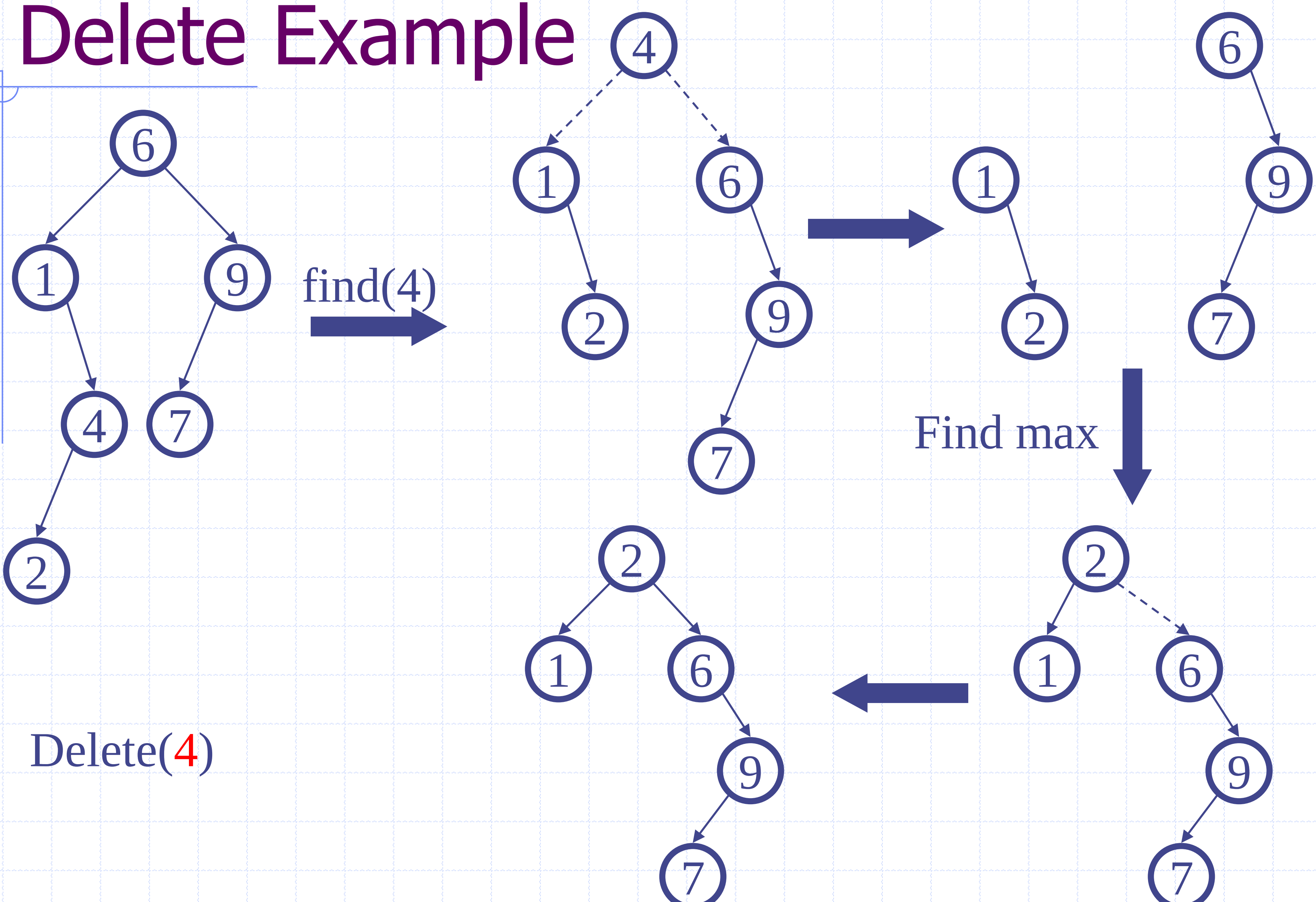


Insert Example



Insert(5)

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))

→ All splay tree operations run in amortized O(log n) time

O(N) operations can occur, but splaying makes them infrequent

Implements most-recently used (MRU) logic

- Splay tree structure is self-tuning

Splay Tree Summary (cont.)

Splaying can be done top-down; better because:

- only one pass
- no recursion or parent pointers necessary

There are alternatives to split/insert and join/delete

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)