Splay Trees
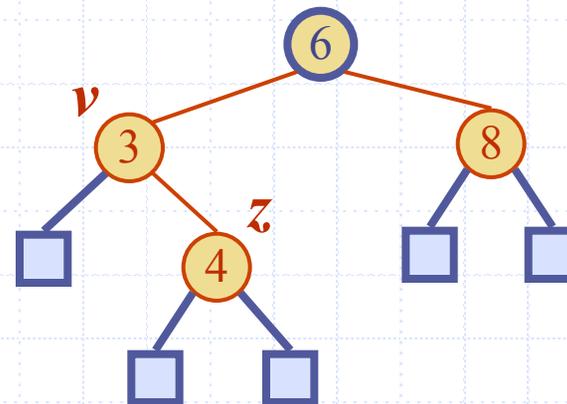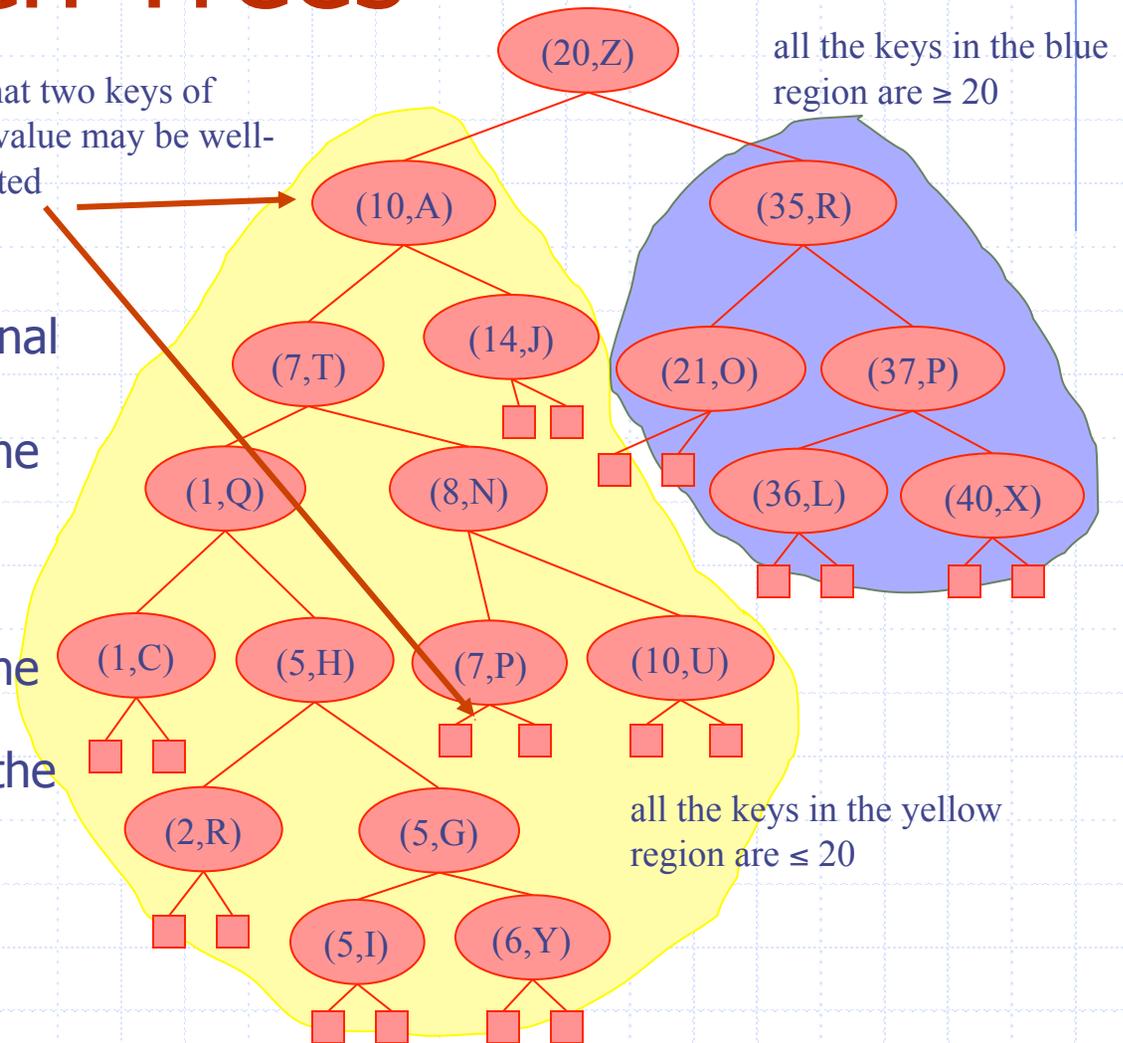
Splay Trees are Binary Search Trees

- BST Rules:
  - items stored only at internal nodes
  - keys stored at nodes in the left subtree of *v* are less than or equal to the key stored at *v*
  - keys stored at nodes in the right subtree of *v* are greater than or equal to the key stored at *v*
- An inorder traversal will return the keys in order

note that two keys of equal value may be well-separated

all the keys in the blue region are ≥ 20

all the keys in the yellow region are ≤ 20

(20,Z)

(10,A) (35,R)

(7,T) (14,J) (21,O) (37,P)

(1,Q) (8,N) (36,L) (40,X)

(1,C) (5,H) (7,P) (10,U)

(2,R) (5,G)

(5,I) (6,Y)

Splay Tree Properties

- Every access causes the deepest accessed node to be "splayed" to top
- Roughly halves (on average) the depth of most nodes on access path
- We can show M access take $O(M \log n)$ time
  - Amortization analysis shows $O(\log n)$ asymptotic performance
- Any one access could be $O(n)$ worst case
- Splaying looks a lot like AVL trees but with less algorithmic and storage overhead

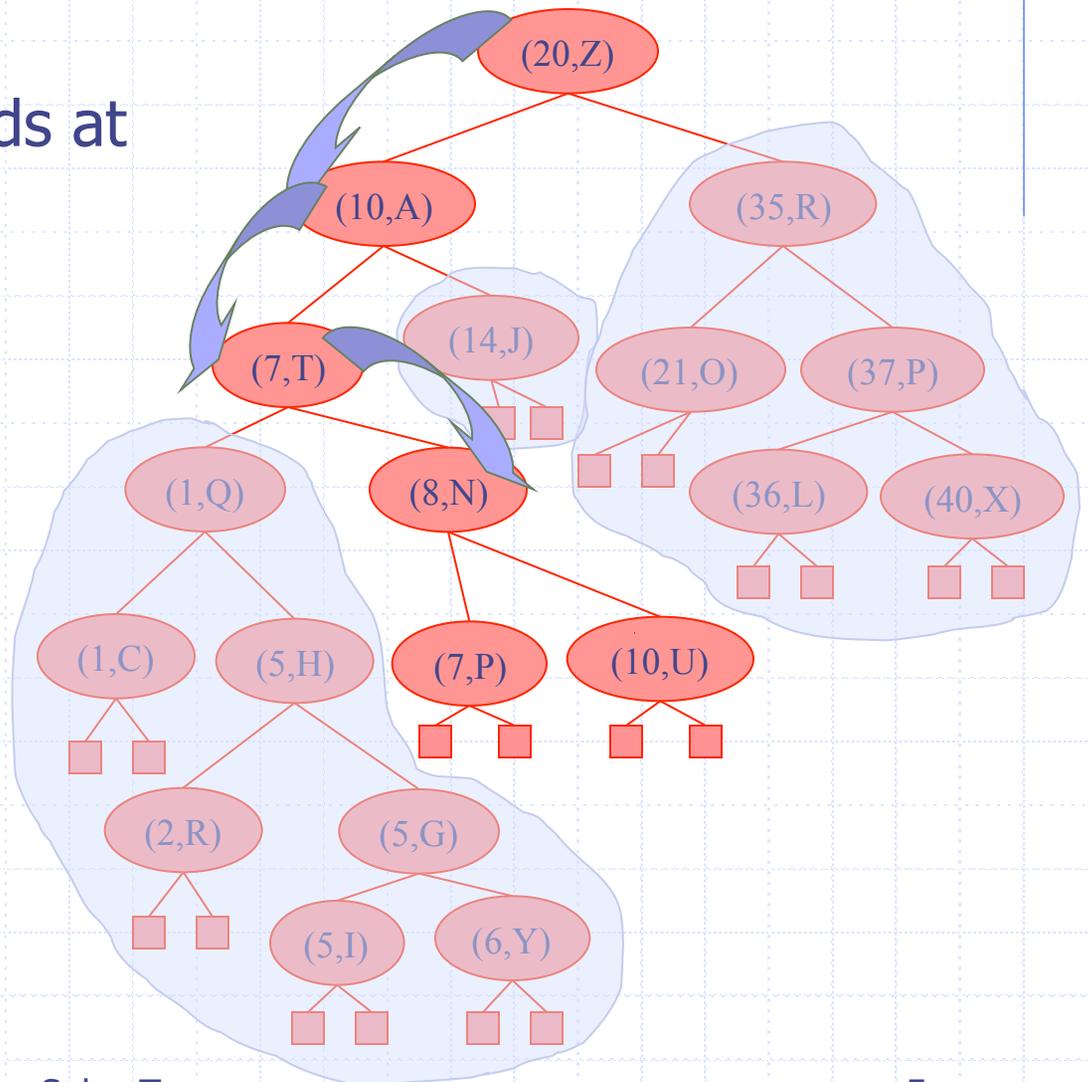# Searching in a Splay Tree: Starts the Same as in a BST

- Search proceeds down the tree to found item or an external node.
- Example: Search for time with key 11.
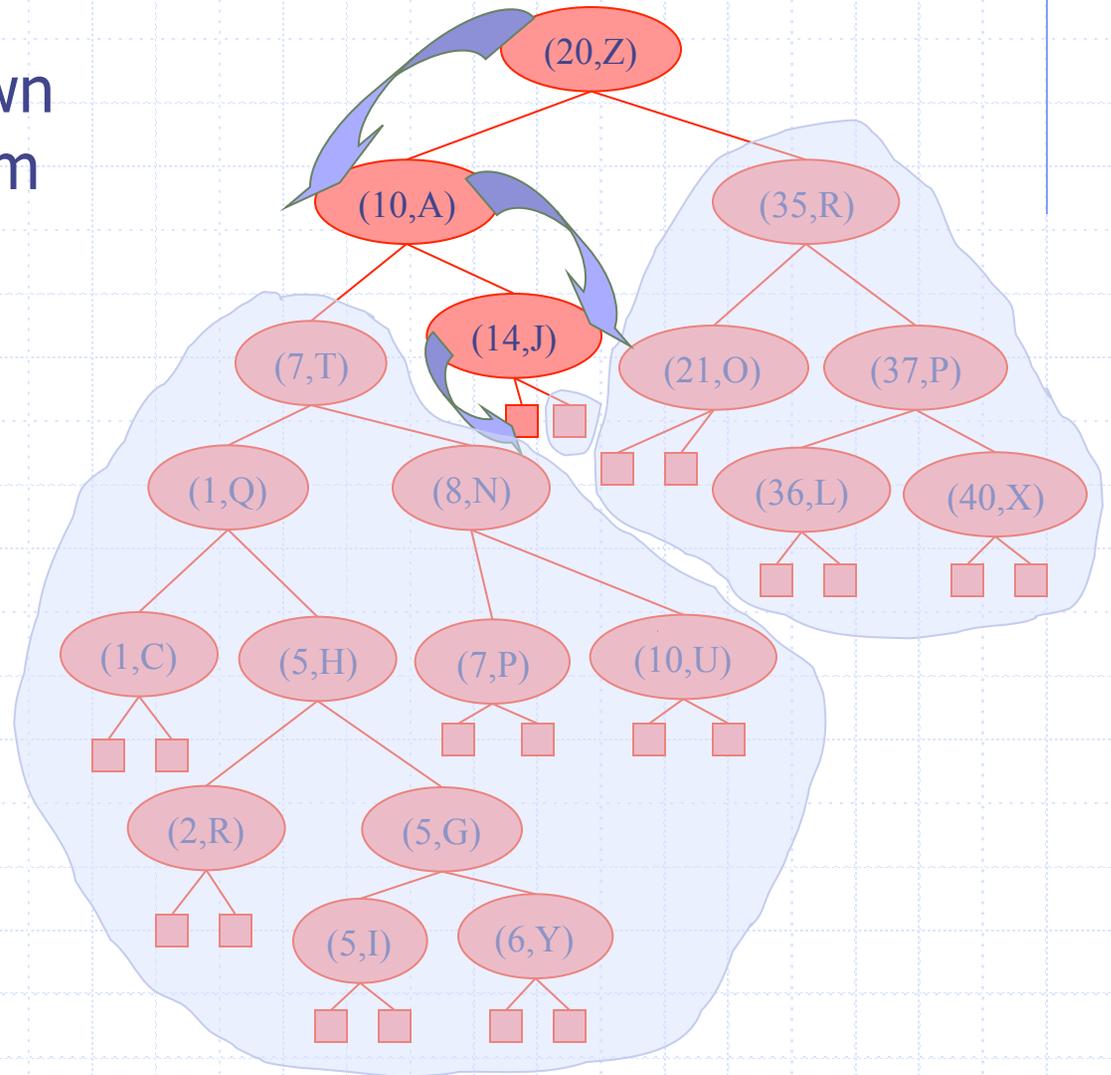
Example Searching in a BST, continued

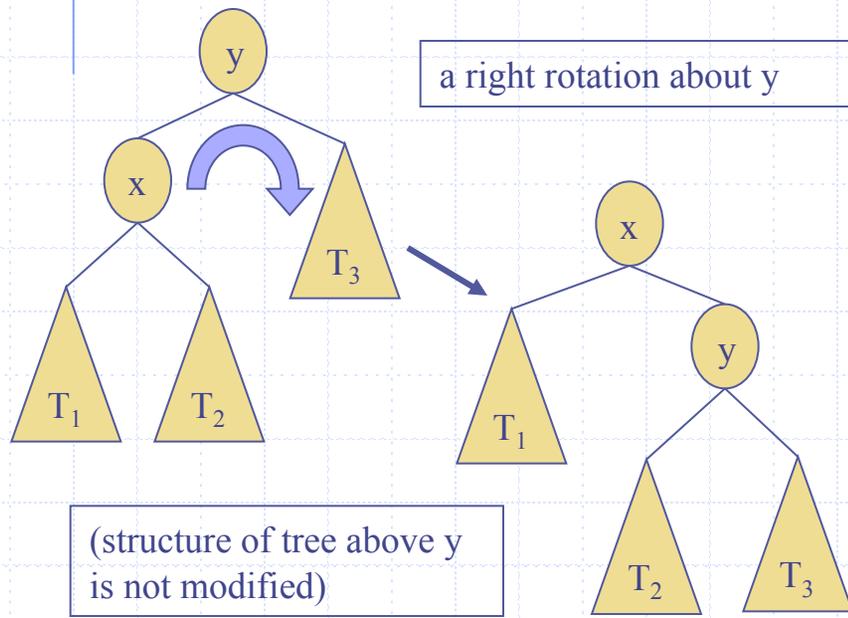◆ search for key 8, ends at an internal node.

Splay Trees do Rotations after Every Operation (Even Search)

◆ new operation: **splay**
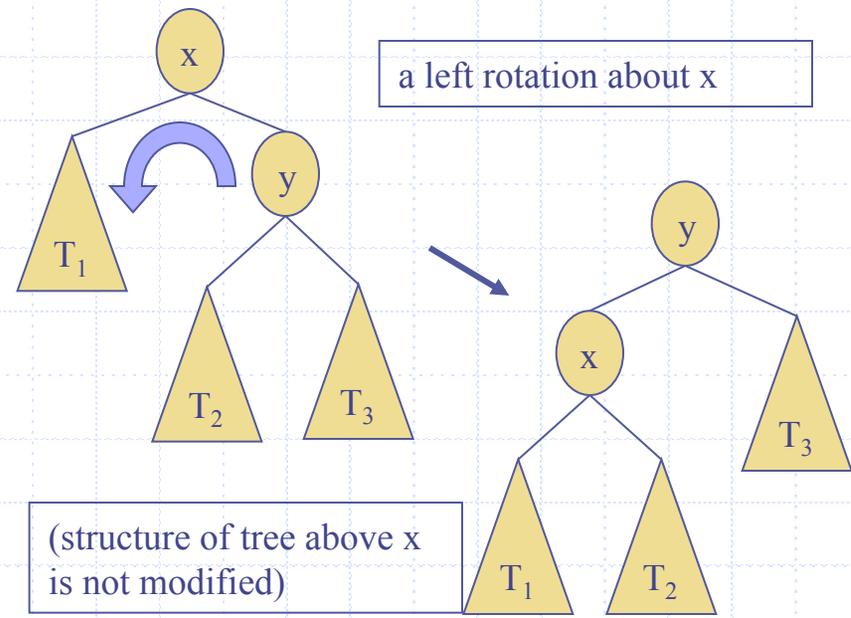- splaying moves a node to the root using rotations

■ **right rotation**
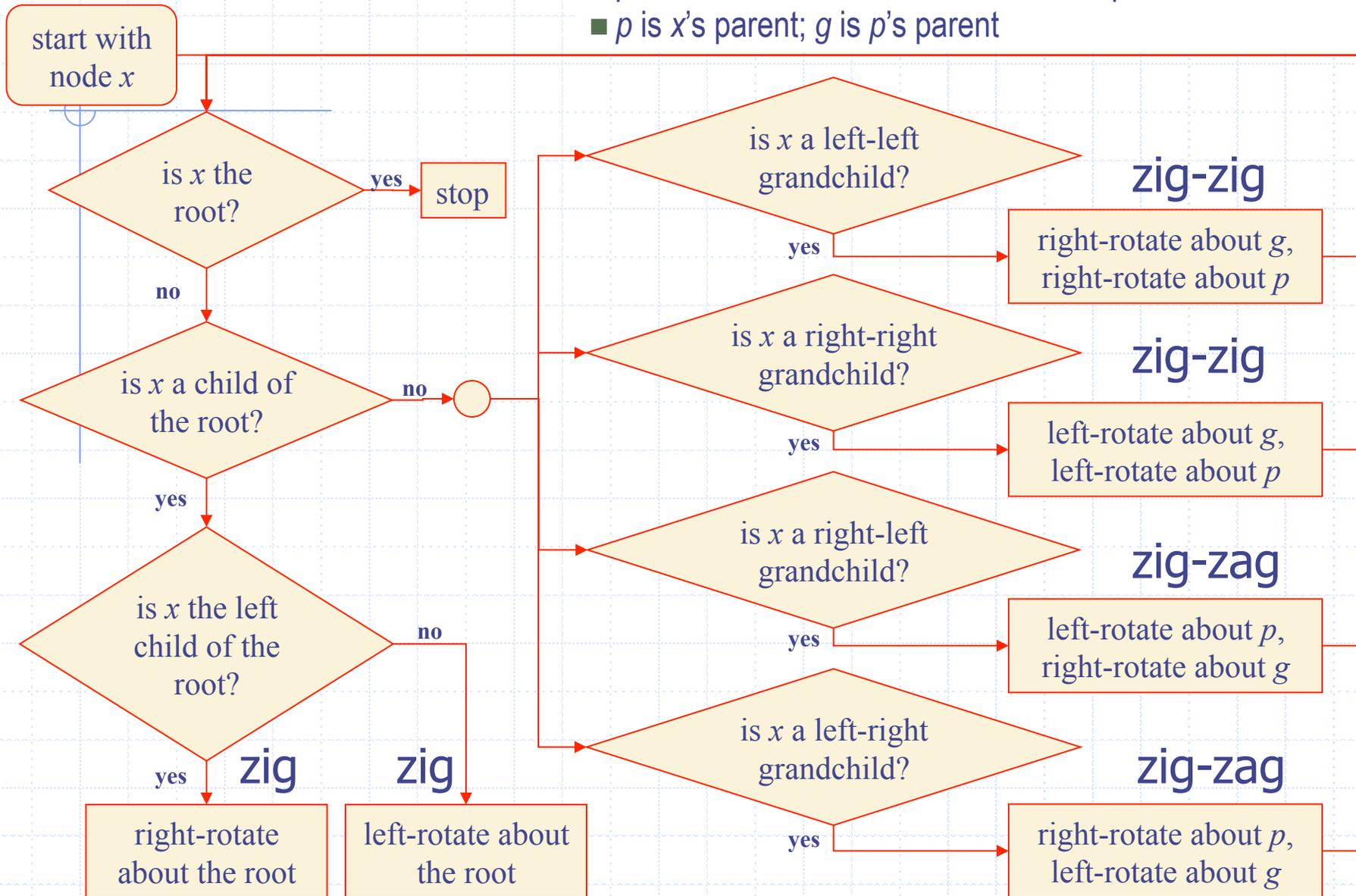- makes the left child x of a node y into y's parent; y becomes the right child of x

a right rotation about y

(structure of tree above y is not modified)

■ **left rotation**
- makes the right child y of a node x into x's parent; x becomes the left child of y

a left rotation about x

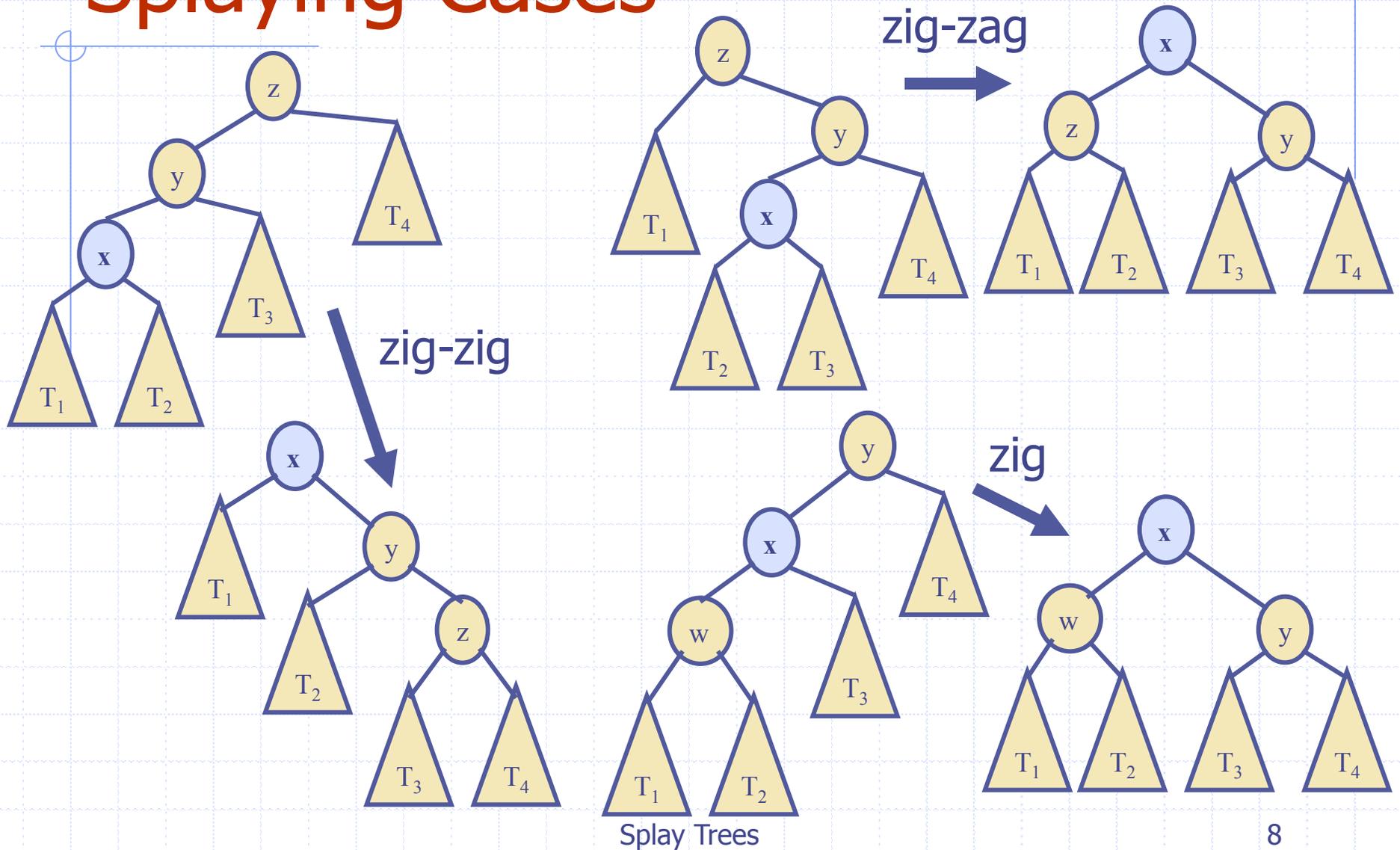(structure of tree above x is not modified)

Splaying:

- "*x* is a left-left grandchild" means *x* is a left child of its parent, which is itself a left child of its parent
- *p* is *x*'s parent; *g* is *p*'s parent

start with node *x*

is *x* the root? — **yes** → **stop**

no ↓

is *x* a child of the root? — **no** →

yes ↓

is *x* the left child of the root? — **no** →

yes → **zig** → right-rotate about the root

zig → left-rotate about the root

is *x* a left-left grandchild? — **yes** → **zig-zig** → right-rotate about *g*, right-rotate about *p*

is *x* a right-right grandchild? — **yes** → **zig-zig** → left-rotate about *g*, left-rotate about *p*

is *x* a right-left grandchild? — **yes** → **zig-zag** → left-rotate about *p*, right-rotate about *g*

is *x* a left-right grandchild? — **yes** → **zig-zag** → right-rotate about *p*, left-rotate about *g*

Splay Trees

7

Visualizing the Splaying Cases
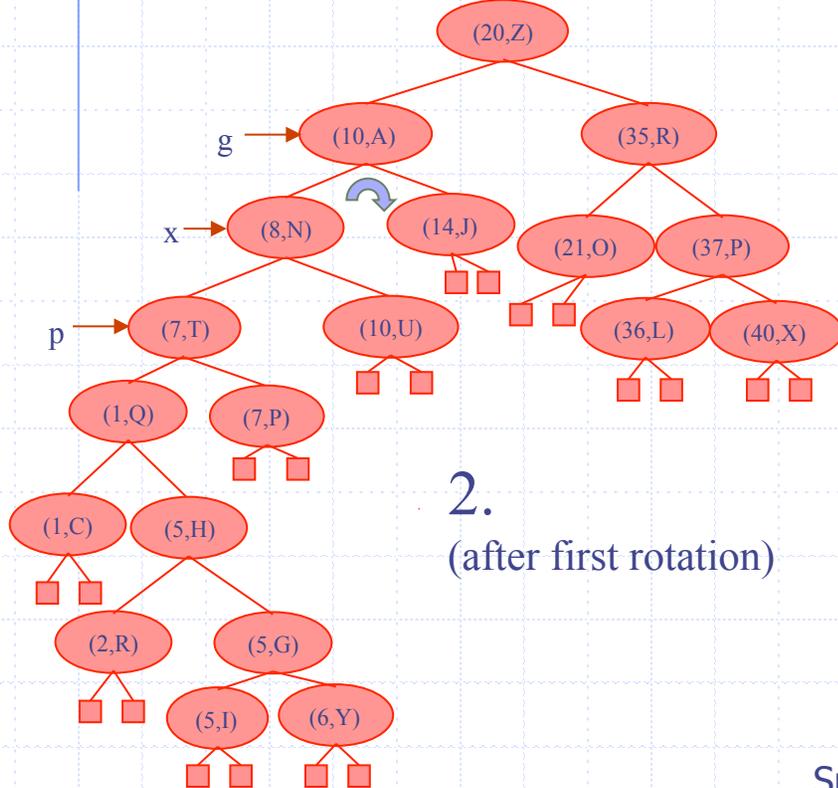

zig-zag

zig-zig
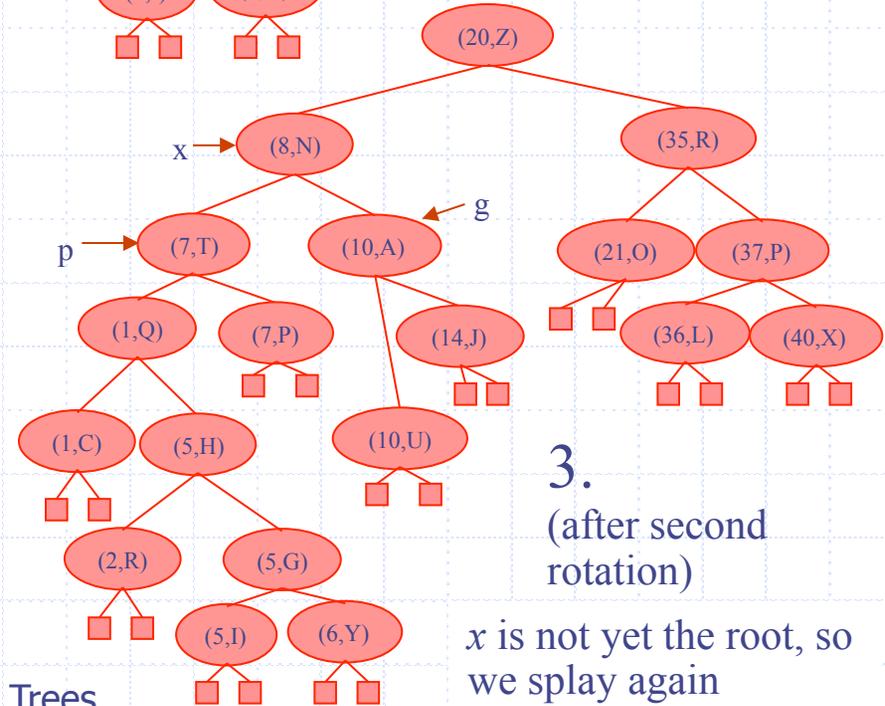
zig

Splaying Example

- let $x = (8,N)$
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate around p, then right-rotate around g
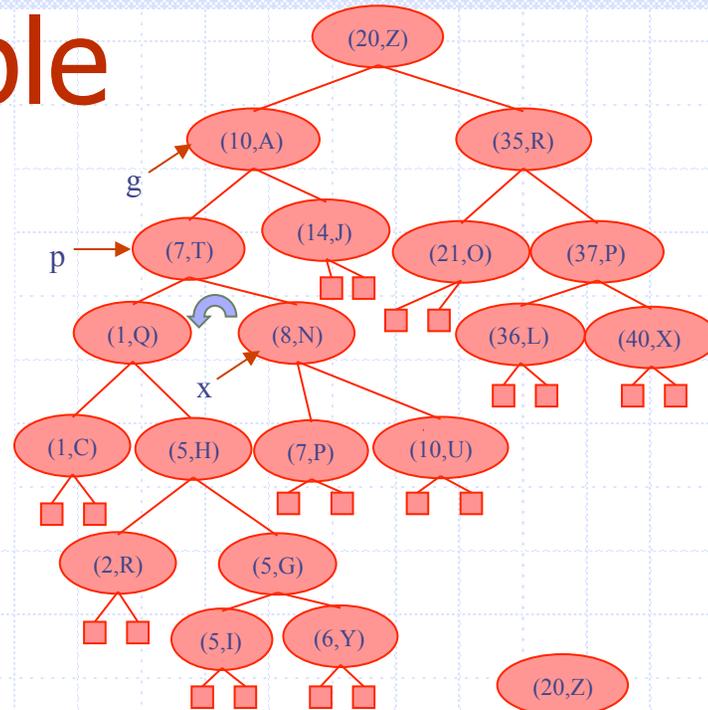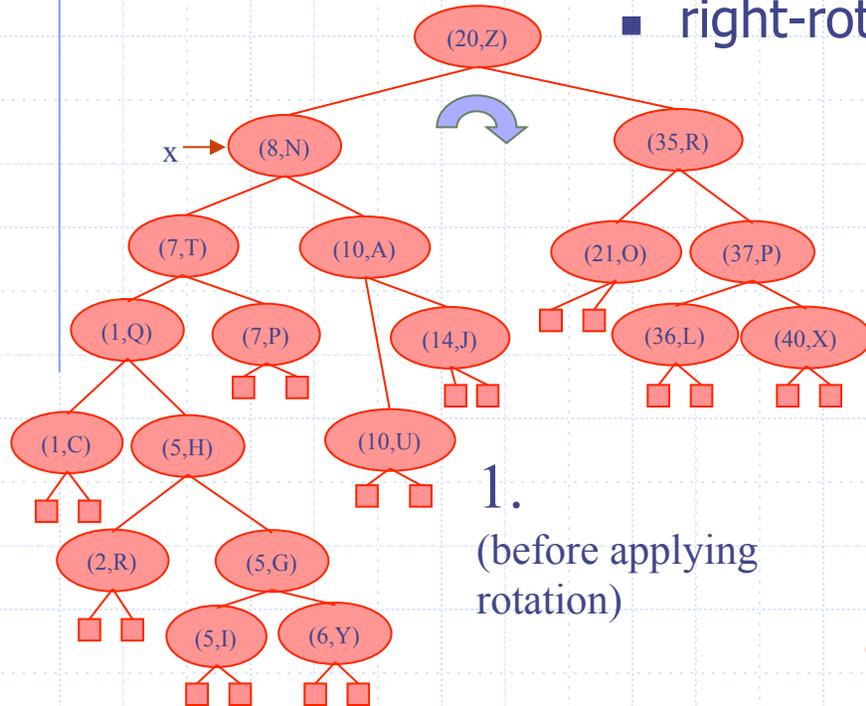

1. (before rotating)
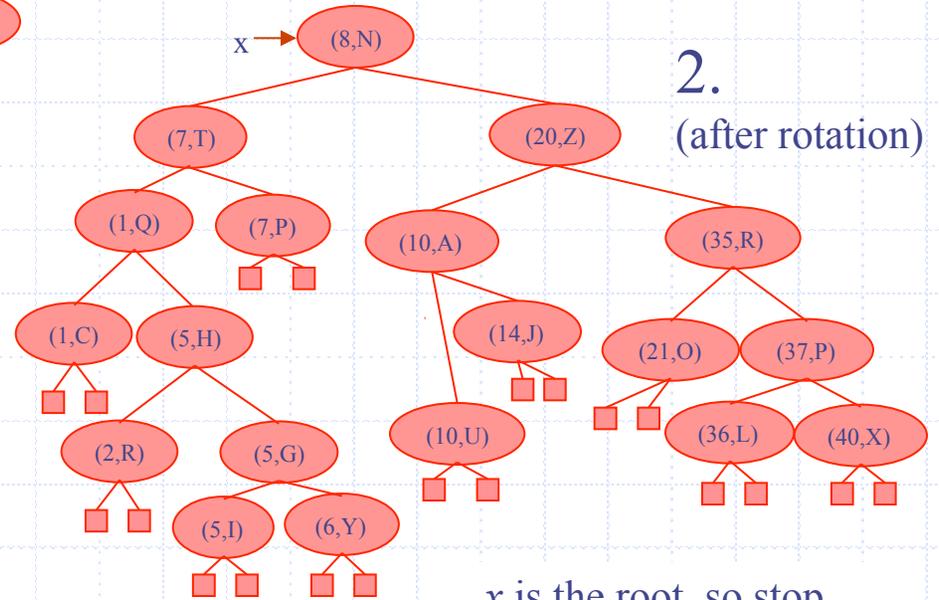
2. (after first rotation)

3. (after second rotation)

x is not yet the root, so we splay again

Splay Trees

Splaying Example, Continued

◆ now *x* is the left child of the root
  ■ right-rotate around root

1. (before applying rotation)

2. (after rotation)

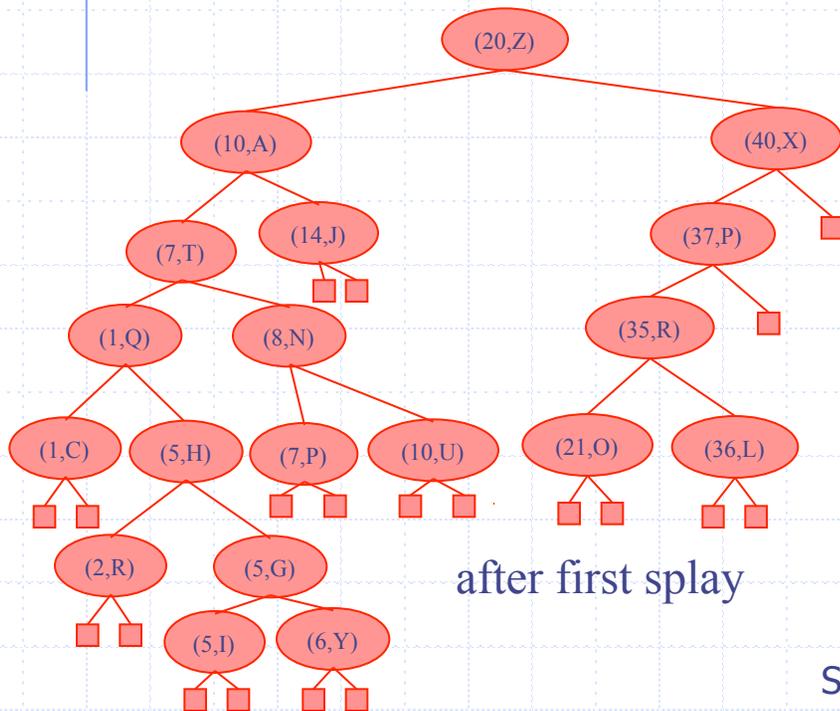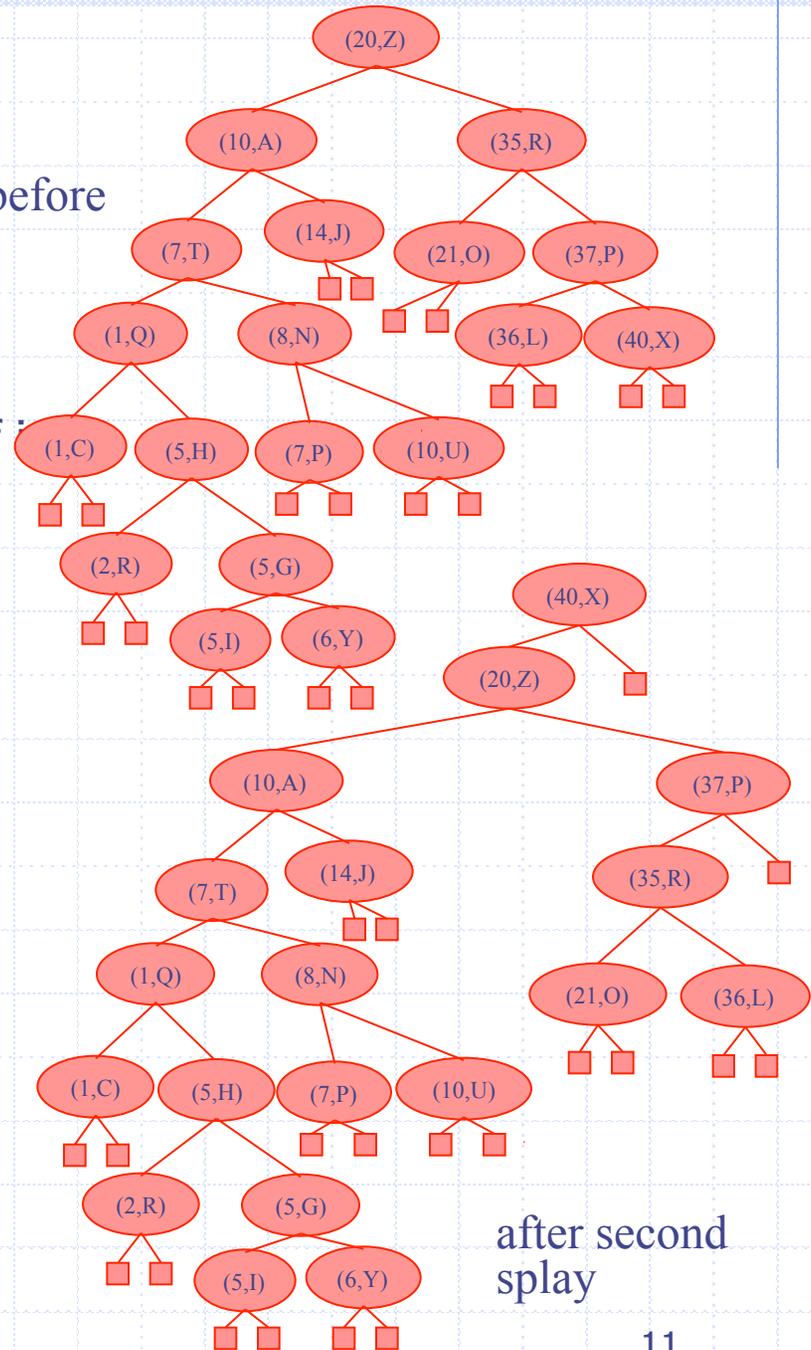*x* is the root, so stop

Example Result of Splaying

- tree might not be more balanced
- e.g. splay (40,X)
  - before, the depth of the shallowest leaf is 3 and the deepest is 7
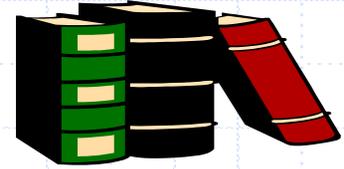 - after, the depth of shallowest leaf is 1 and deepest is 8

before

after first splay

after second splay

Splay Tree Manipulation

◆ Search: Splay found node

◆ Insert: Insert as a normal binary tree and splay

◆ Delete:
- Search (puts node at root)
- Delete yielding TL and TR
- Access largest node in TL, gives tree with no right child
- Put in TR

Splay Tree Definition

◆ a *splay tree* is a binary search tree where a node is splayed after it is accessed (for a search or update)

  ■ deepest internal node accessed is splayed

  ■ splaying costs $O(h)$, where h is height of the tree – which is still $O(n)$ worst-case

    ◆ $O(h)$ rotations, each of which is $O(1)$

Splay Trees & Ordered Dictionaries
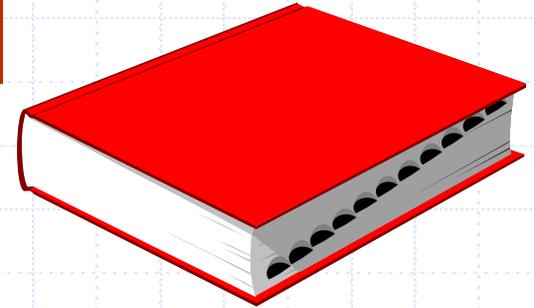
◆ which nodes are splayed after each operation?

| method	splay node
findElement	if key found, use that node if key not found, use parent of ending external node
insertElement	use the new node containing the item inserted
removeElement	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

Amortized Analysis of Splay Trees
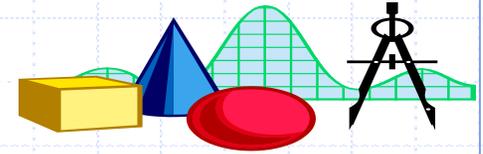
◆ Running time of each operation is proportional to time for splaying.

◆ Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.

◆ Costs: zig = $1, zig-zig = $2, zig-zag = $2.

◆ Thus, cost for playing a node at depth d = $d.

◆ Imagine that we store rank(v) cyber-dollars at each node v of the splay tree (just for the sake of analysis).

Cost per zig

zig
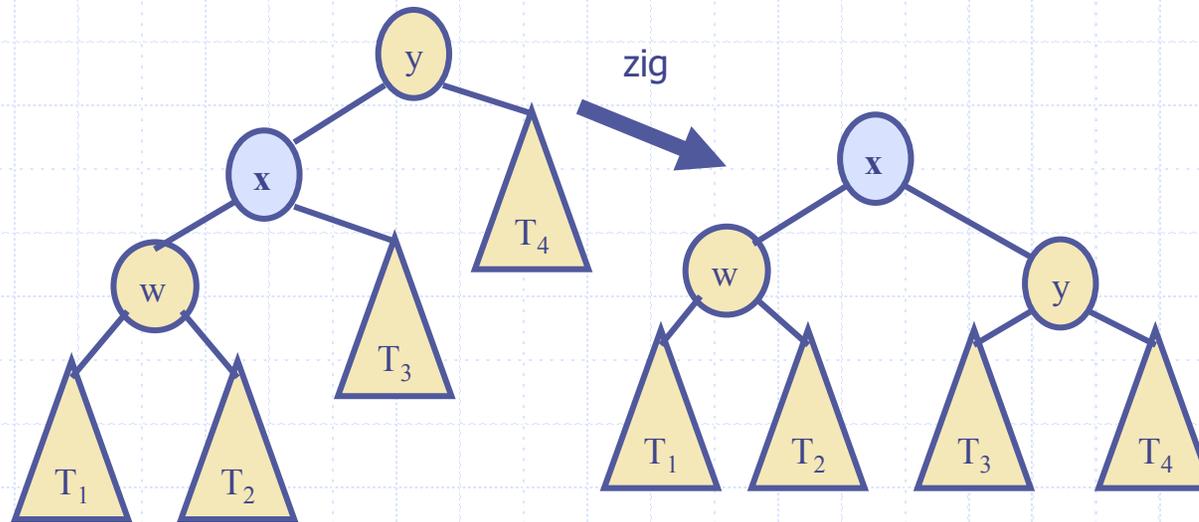
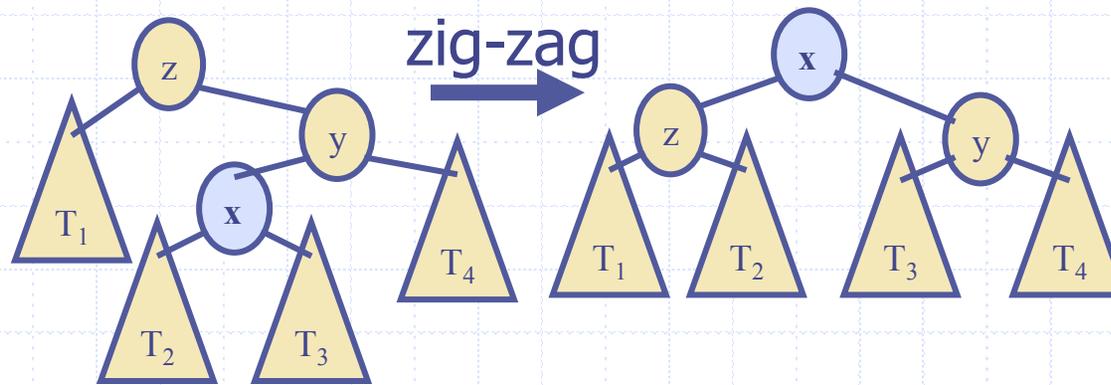◆ Doing a zig at x costs at most rank'(x) - rank(x):

- cost = rank'(x) + rank'(y) - rank(y) - rank(x)
  $\leq$ rank'(x) - rank(x).

Cost per zig-zig and zig-zag
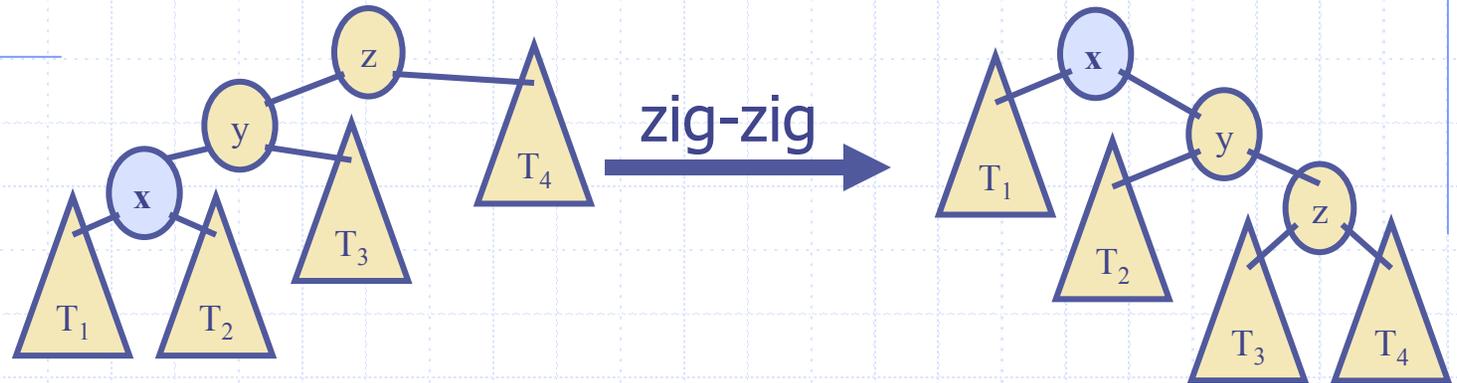
◆ Doing a zig-zig or zig-zag at x costs at most
$3(rank'(x) - rank(x)) - 2$.

- ■ Proof: See Theorem 3.9, Page 192.

Cost of Splaying

- Cost of splaying a node x at depth d of a tree rooted at r:
  - at most 3(rank(r) - rank(x)) - d + 2:
  - Proof: Splaying x takes d/2 splaying substeps:

$$\mathrm{cost} \leq \sum_{i=1}^{d/2} \mathrm{cost}_i$$

$$\leq \sum_{i=1}^{d/2} (3(\mathrm{rank}_i(x) - \mathrm{rank}_{i-1}(x)) - 2) + 2$$

$$= 3(\mathrm{rank}(r) - \mathrm{rank}_0(x)) - 2(d/d) + 2$$

$$\leq 3(\mathrm{rank}(r) - \mathrm{rank}(x)) - d + 2.$$

Performance of Splay Trees

- Recall: rank of a node is logarithm of its size.
- Thus, amortized cost of any splay operation is **O(log n).**
- In fact, the analysis goes through for any reasonable definition of rank(x).
- This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases. (See Theorems 3.10 and 3.11.)