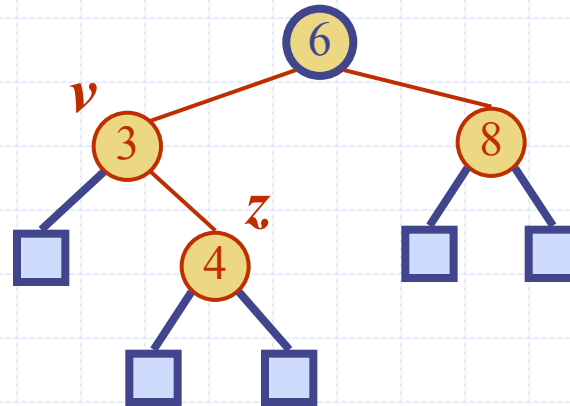


Splay Trees



Splay Trees are Binary Search Trees (§ 9.3)

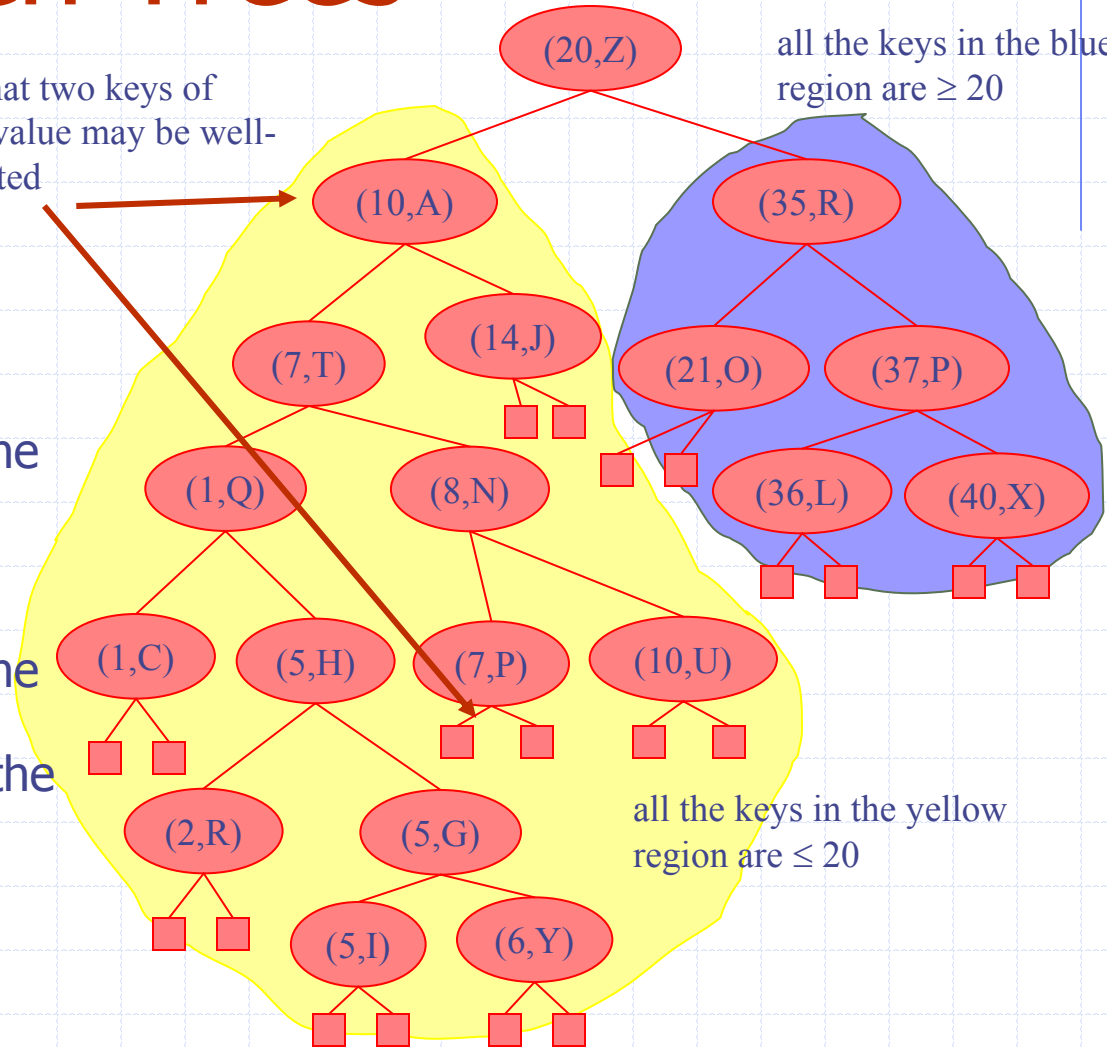
note that two keys of equal value may be well-separated

all the keys in the blue region are ≥ 20

◆ BST Rules:

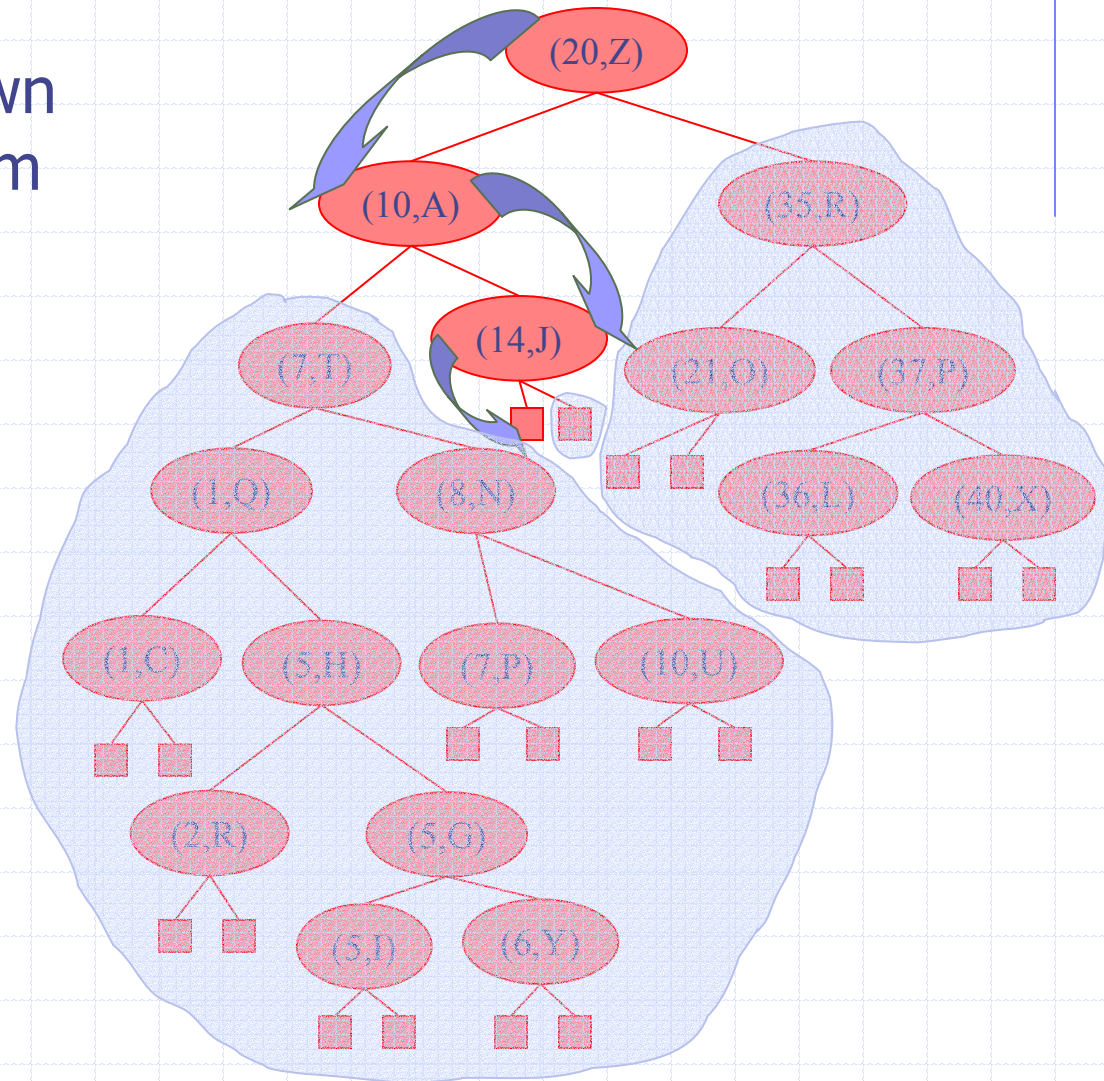
- entries stored only at internal nodes
- keys stored at nodes in the left subtree of v are less than or equal to the key stored at v
- keys stored at nodes in the right subtree of v are greater than or equal to the key stored at v

◆ An inorder traversal will return the keys in order



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

- ◆ Search proceeds down the tree to find item or an external node.
- ◆ Example: Search for time with key 11.



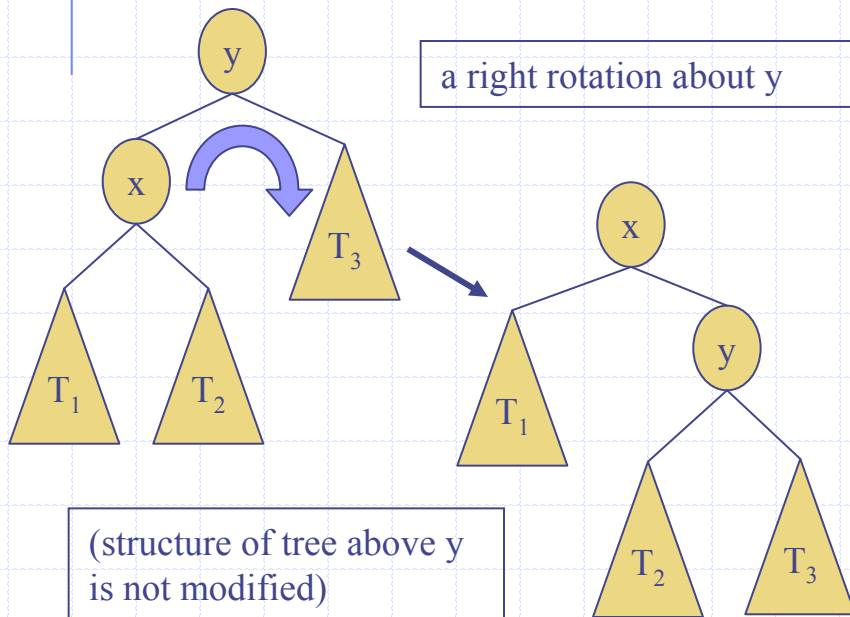
Splay Trees do Rotations after Every Operation (Even Search)

◆ new operation: *splay*

- splaying moves a node to the root using rotations

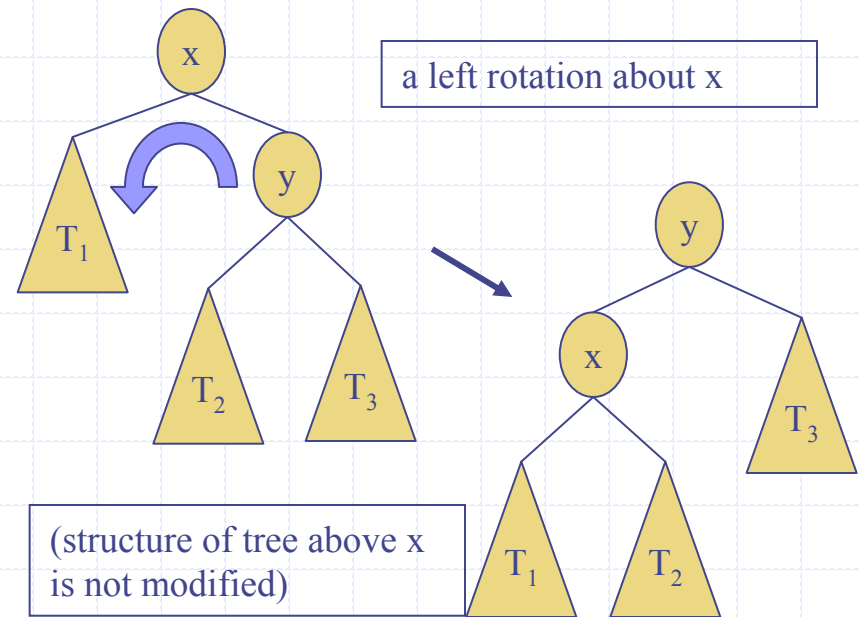
■ right rotation

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



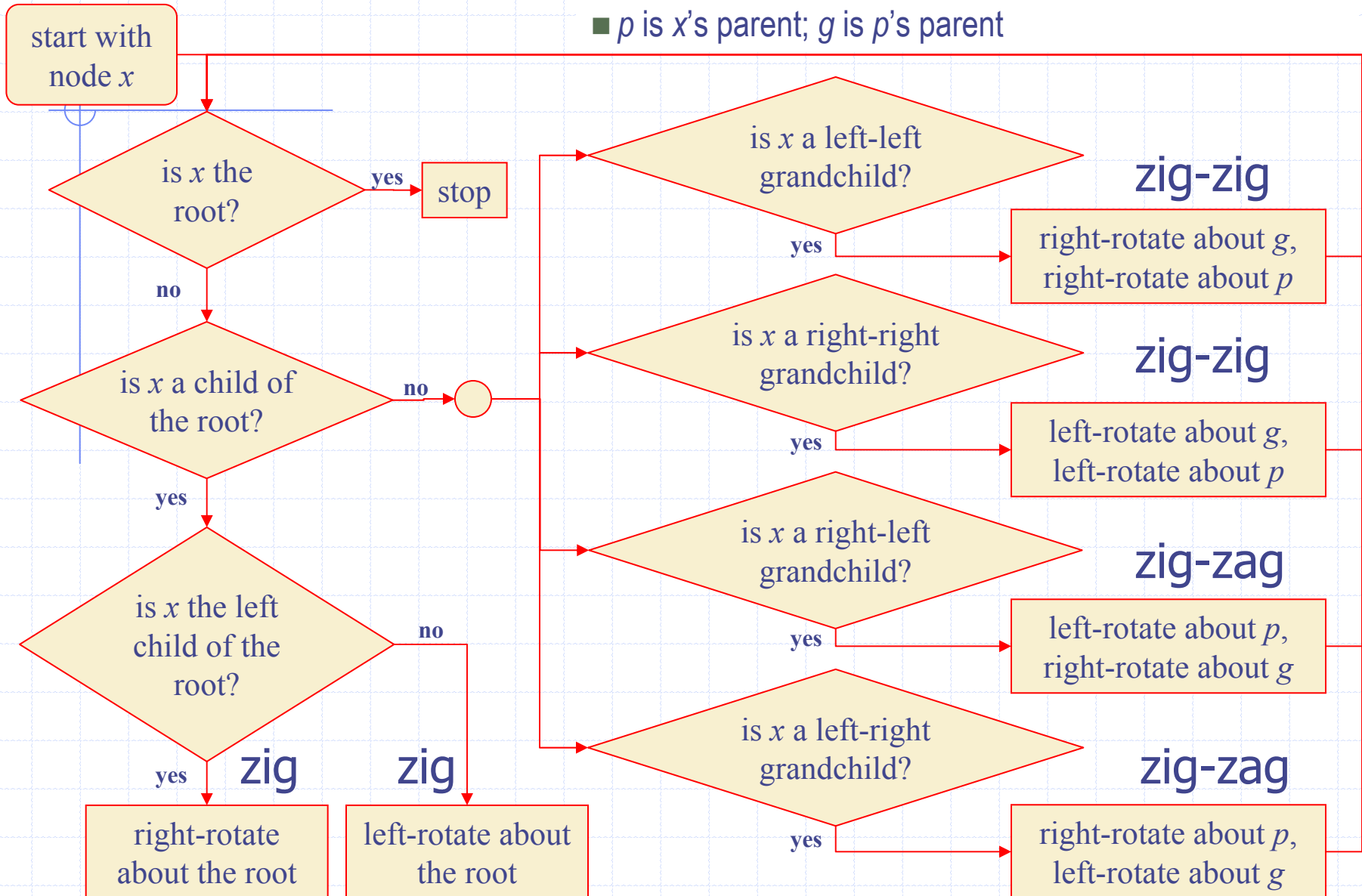
■ left rotation

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

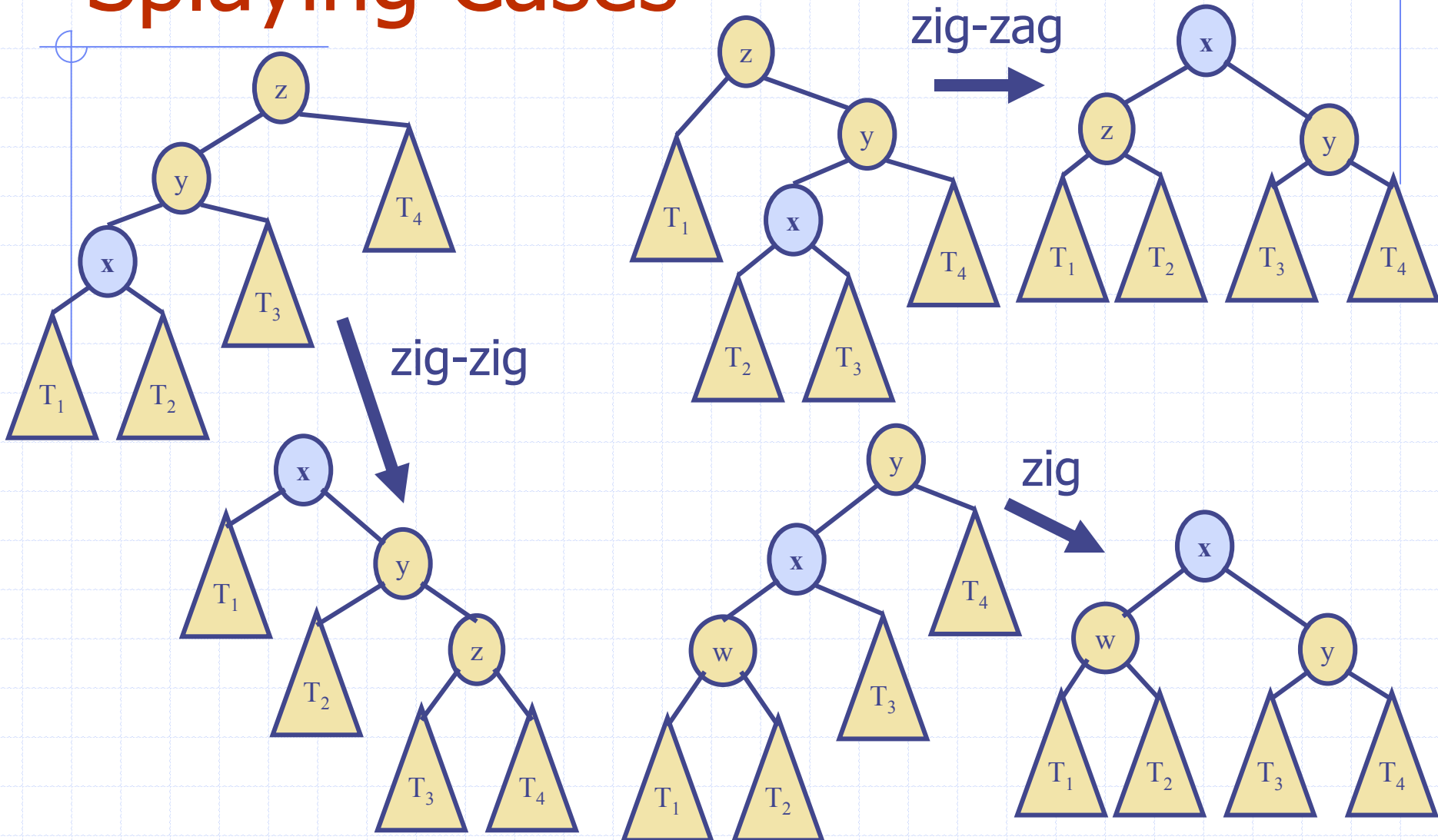


Splaying:

- “ 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



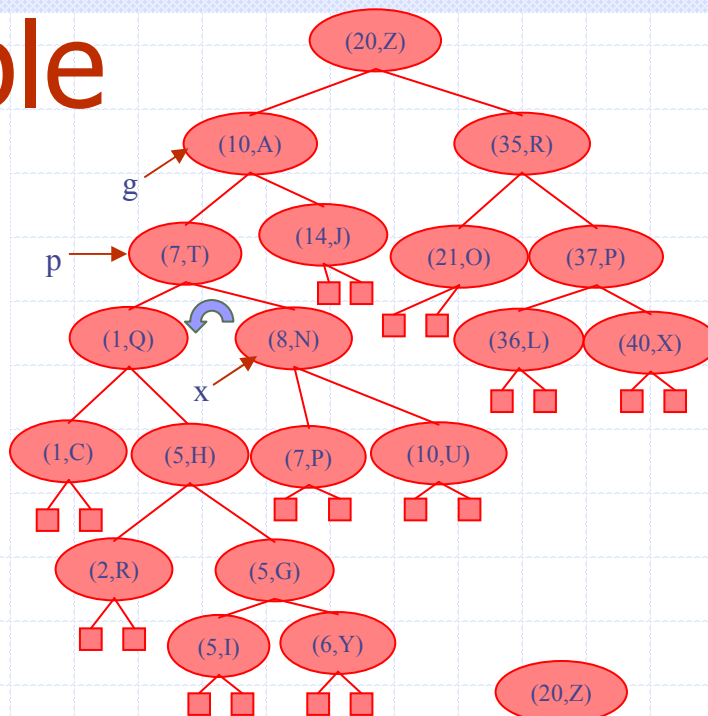
Visualizing the Splaying Cases



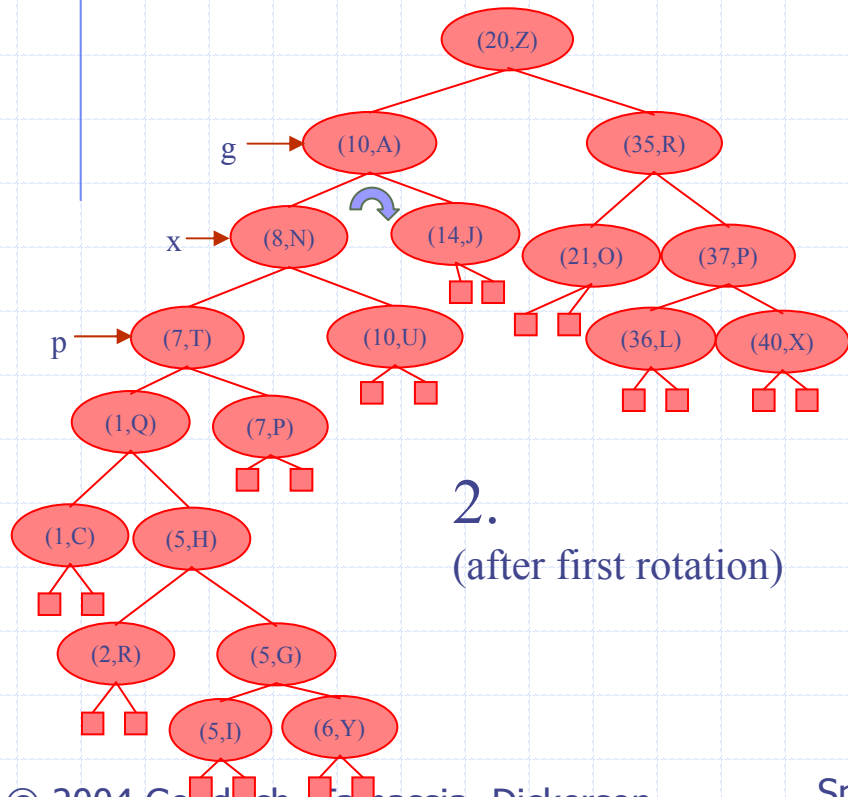
Splaying Example

◆ let $x = (8,N)$

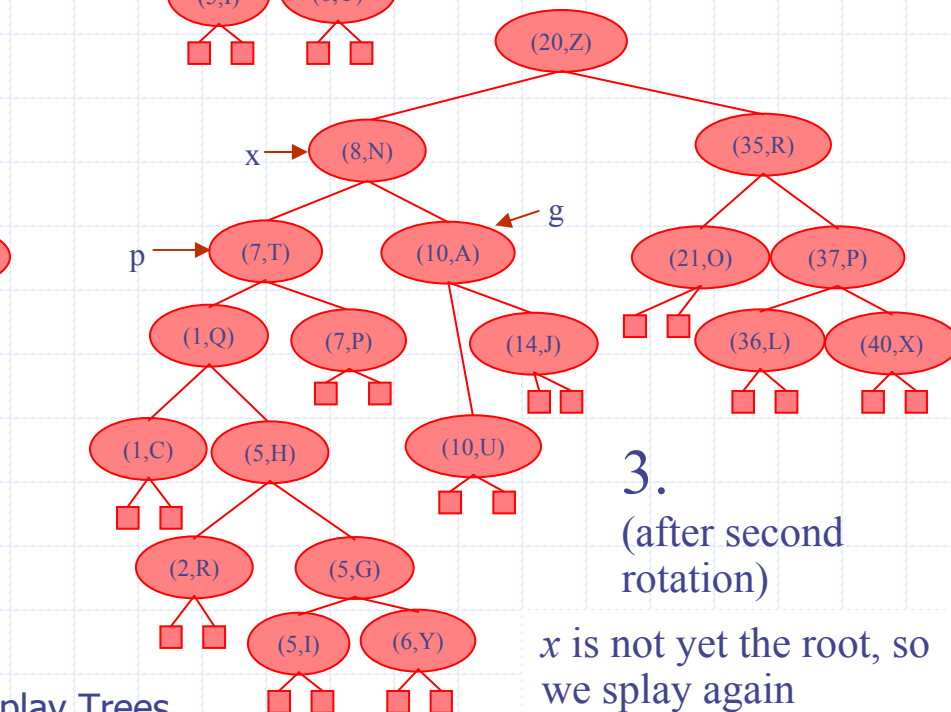
- x is the right child of its parent, which is the left child of the grandparent
- left-rotate around p , then right-rotate around g



1.
(before rotating)



2.
(after first rotation)

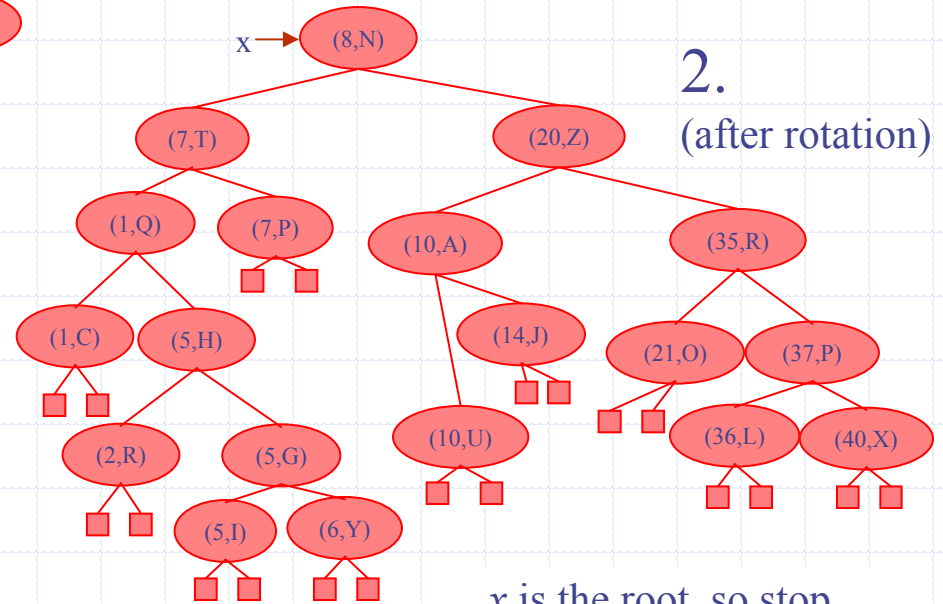
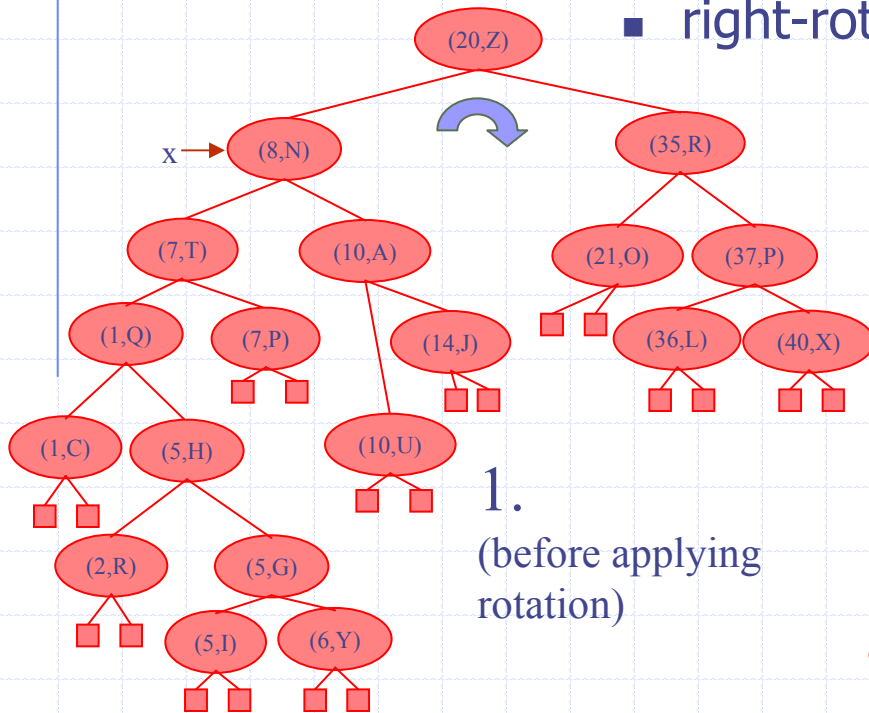


3.
(after second rotation)

x is not yet the root, so we splay again

Splaying Example, Continued

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

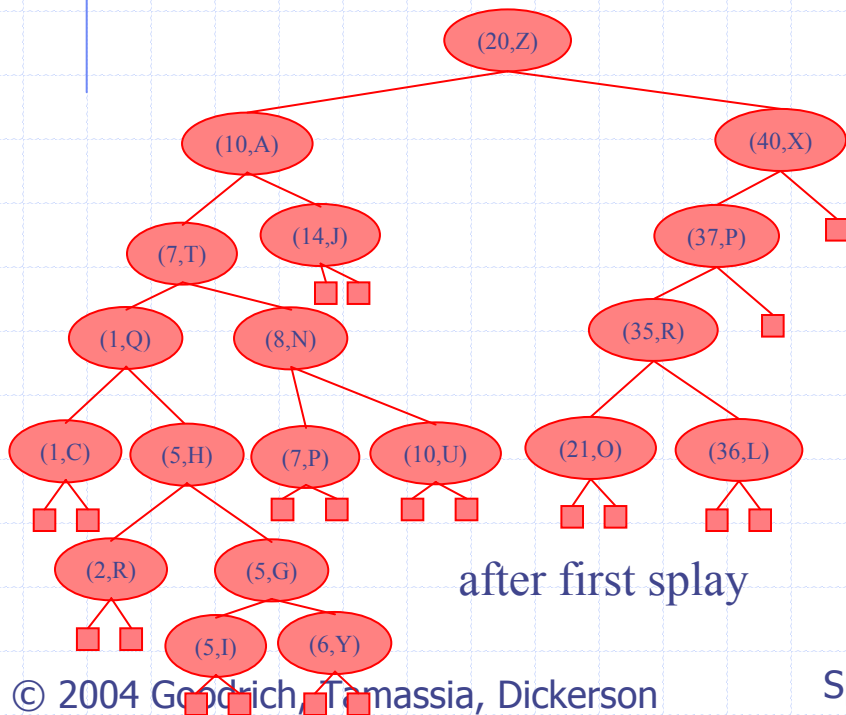
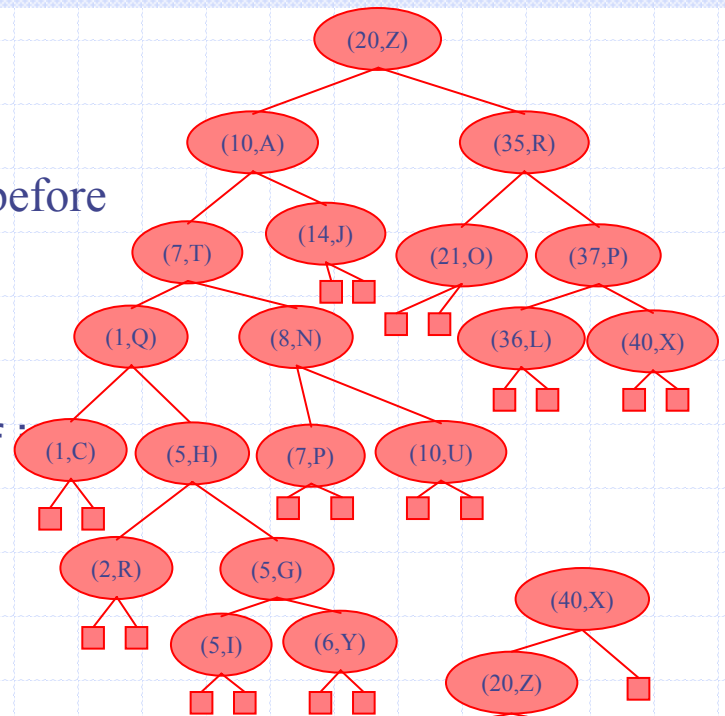


x is the root, so stop

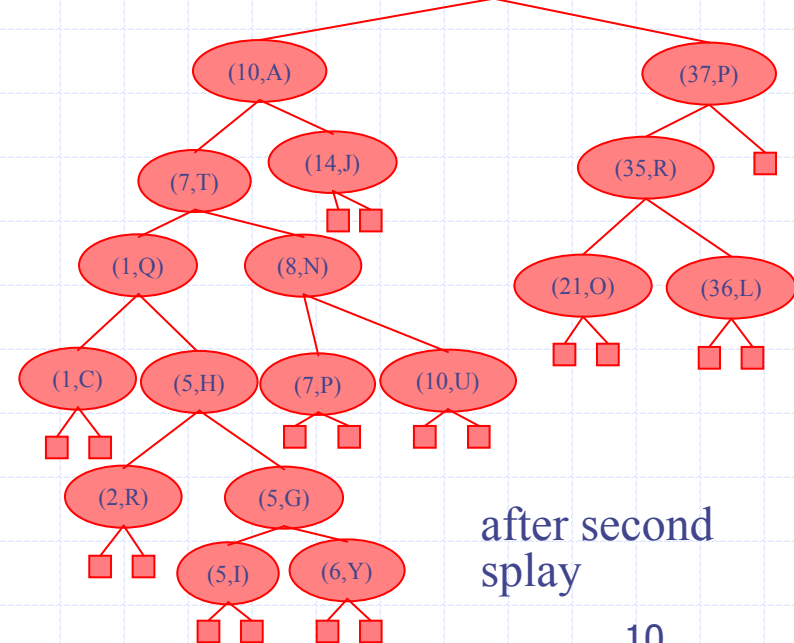
Example Result of Splaying

- 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

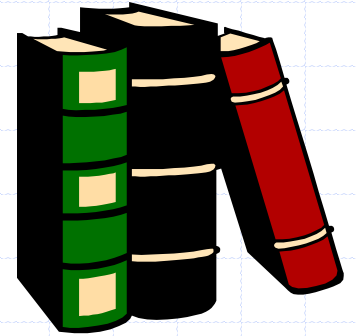
before



after first splay



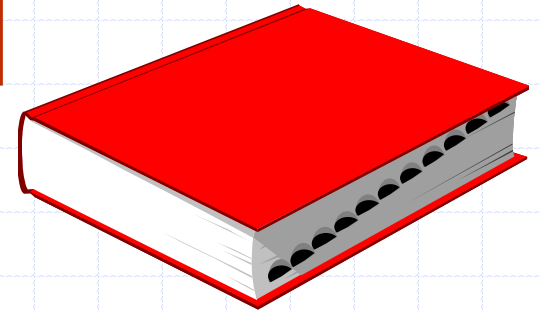
after second splay



Splay Tree 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 $O(1)$

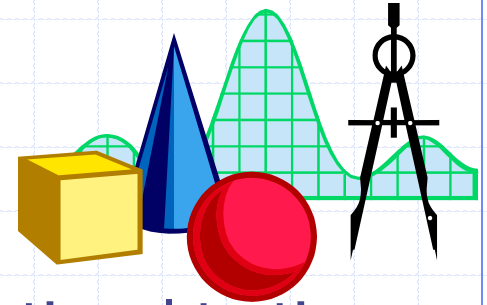
Splay Trees & Ordered Dictionaries



- ◆ which nodes are splayed after each operation?

| method | splay node |
|-------------|---|
| find(k) | if key found, use that node if key not found, use parent of ending external node |
| insert(k,v) | use the new node containing the entry inserted |
| remove(k) | 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) |

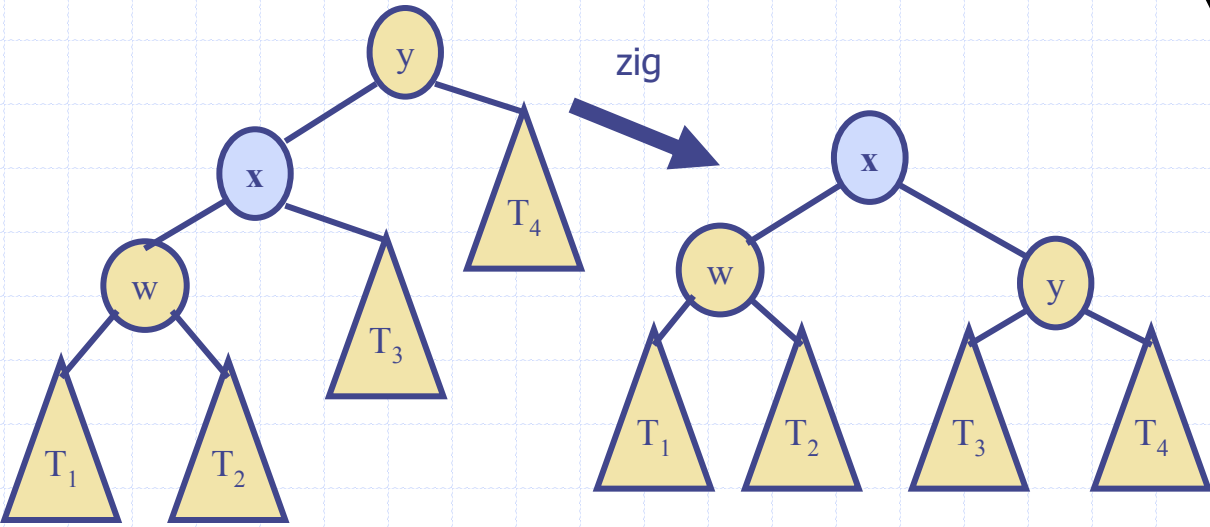
Amortized Analysis of Splay Trees



- ◆ Running time of each operation is proportional to time for splaying.
- ◆ Define $\text{rank}(v)$ 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 $\text{rank}(v)$ cyber-dollars at each node v of the splay tree (just for the sake of analysis).

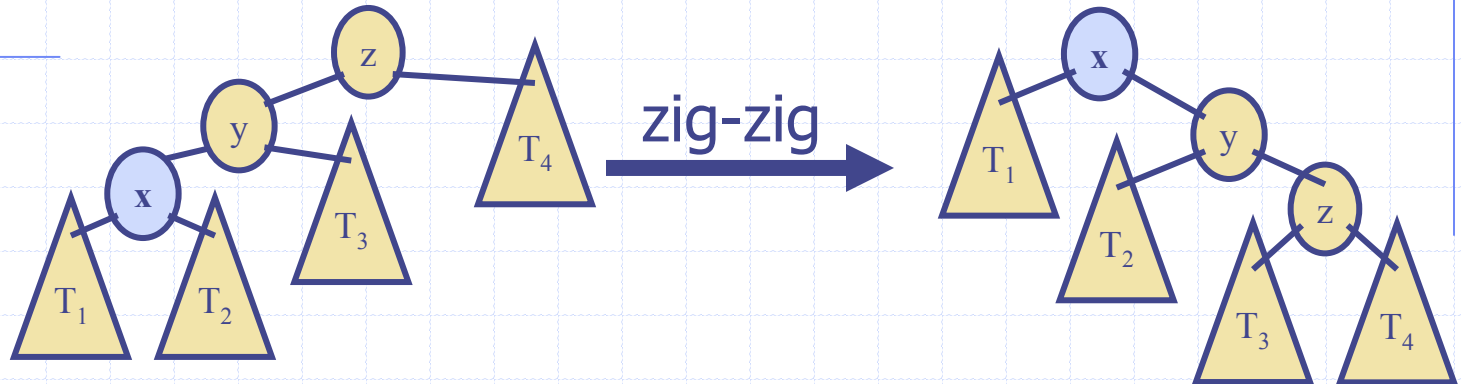


Cost per zig

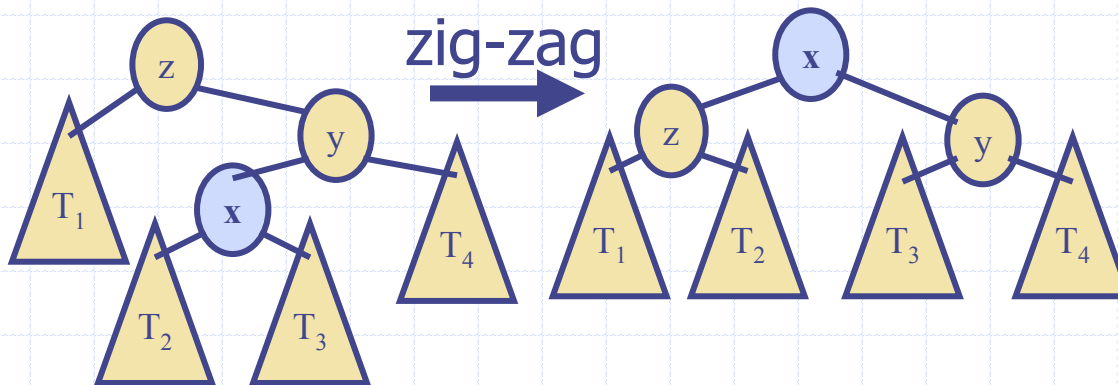


- ◆ Doing a zig at x costs at most $\text{rank}'(x) - \text{rank}(x)$:
 - $\text{cost} = \text{rank}'(x) + \text{rank}'(y) - \text{rank}(y) - \text{rank}(x) \leq \text{rank}'(x) - \text{rank}(x)$.

Cost per zig-zig and zig-zag



- ◆ Doing a zig-zig or zig-zag at x costs at most $3(\text{rank}'(x) - \text{rank}(x)) - 2$.
 - Proof: See Proposition 9.2, Page 440.





Cost of Splaying

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

$$\begin{aligned}\text{cost} &\leq \sum_{i=1}^{d/2} \text{cost}_i \\ &\leq \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\ &= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2 \\ &\leq 3(\text{rank}(r) - \text{rank}(x)) - d + 2.\end{aligned}$$

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 $\text{rank}(x)$.
- ◆ 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 Proposition 9.4 and 9.5.)