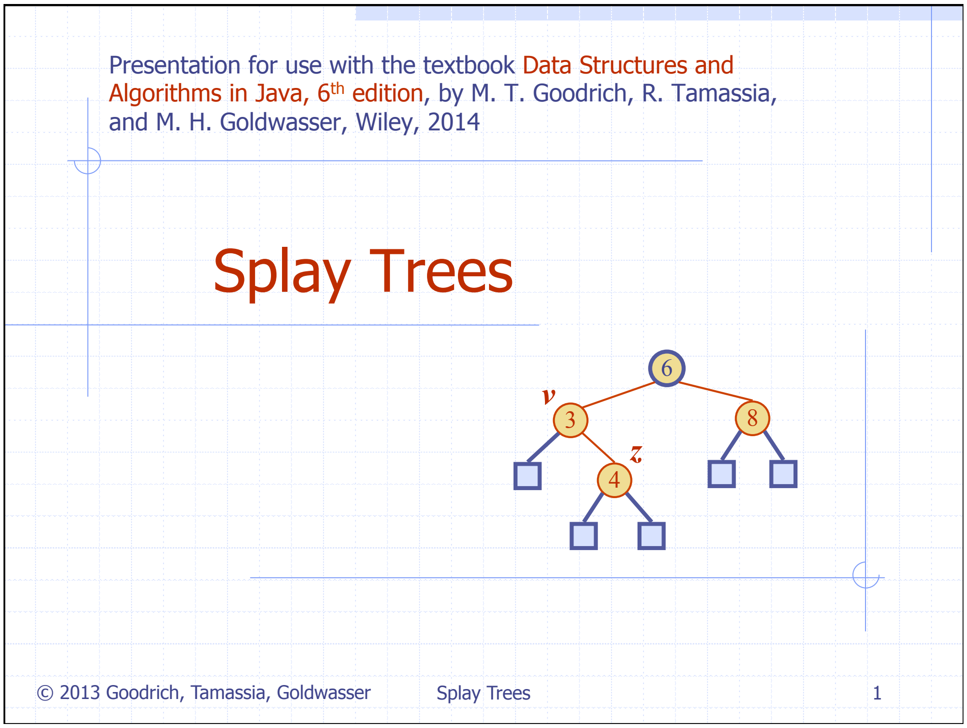
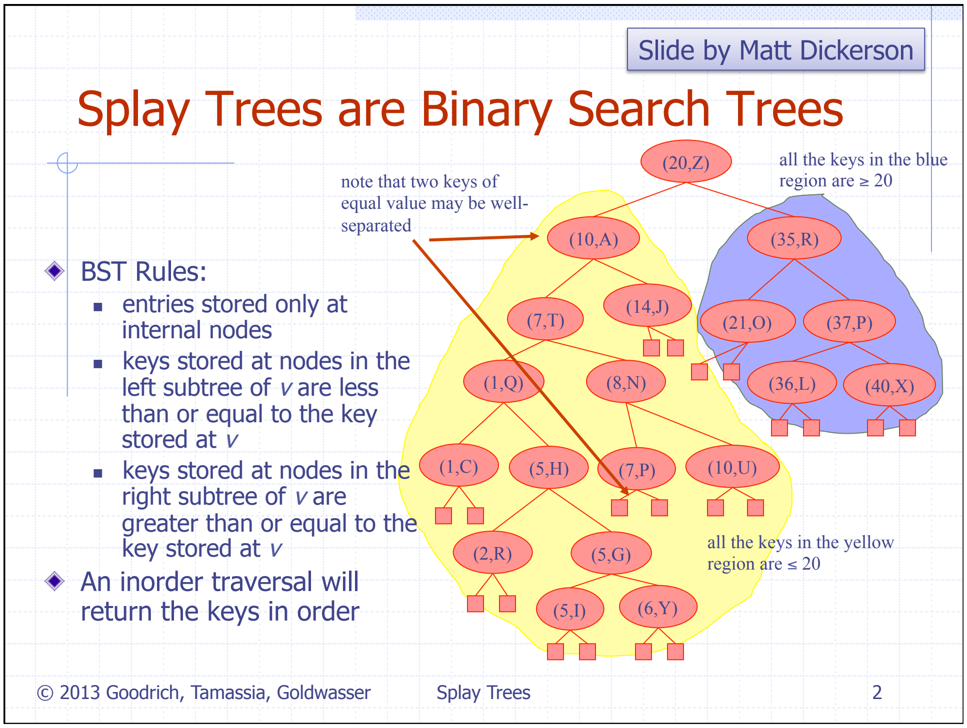


Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Splay Trees



Slide by Matt Dickerson

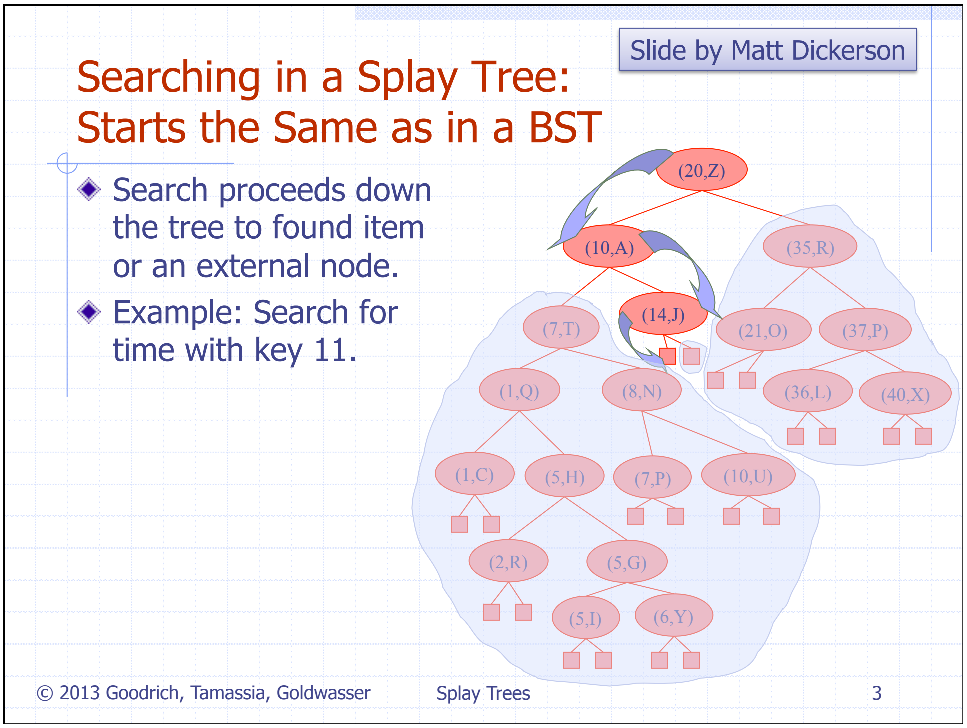
Splay Trees are Binary Search Trees

- 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

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

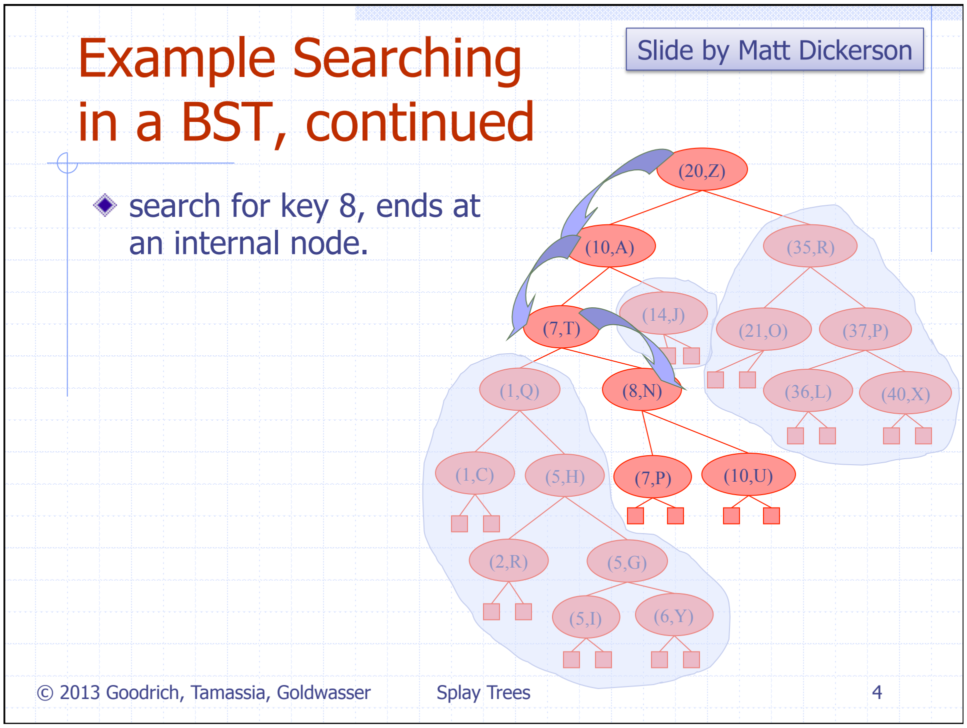


Slide by Matt Dickerson

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.

© 2013 Goodrich, Tamassia, Goldwasser Splay Trees 3

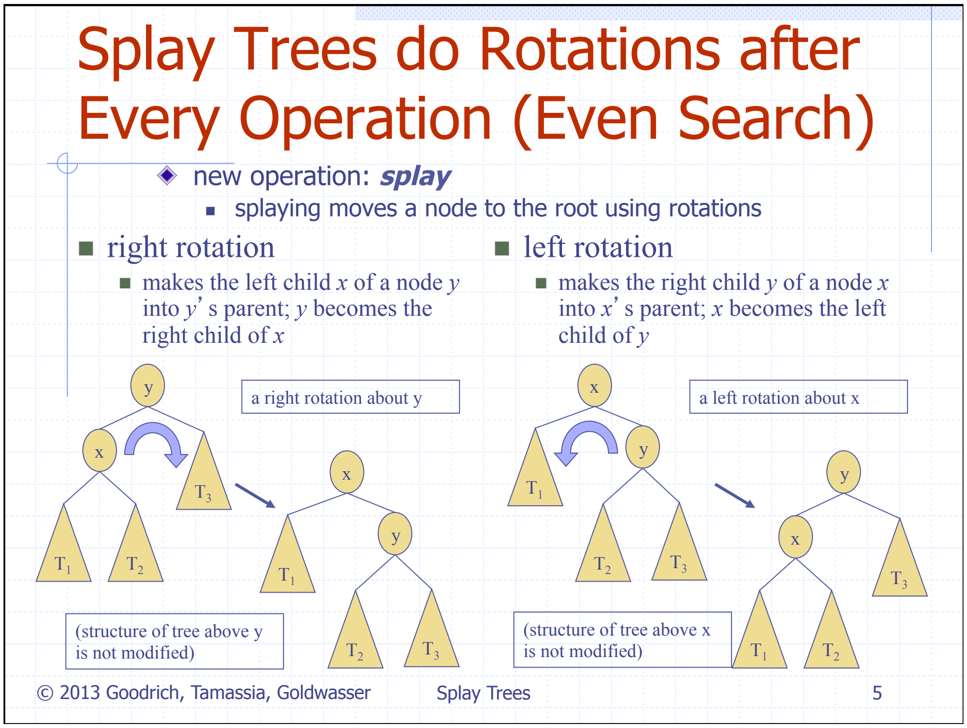


Slide by Matt Dickerson

Example Searching in a BST, continued

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

© 2013 Goodrich, Tamassia, Goldwasser Splay Trees 4



Splay Trees do Rotations after Every Operation (Even Search)

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

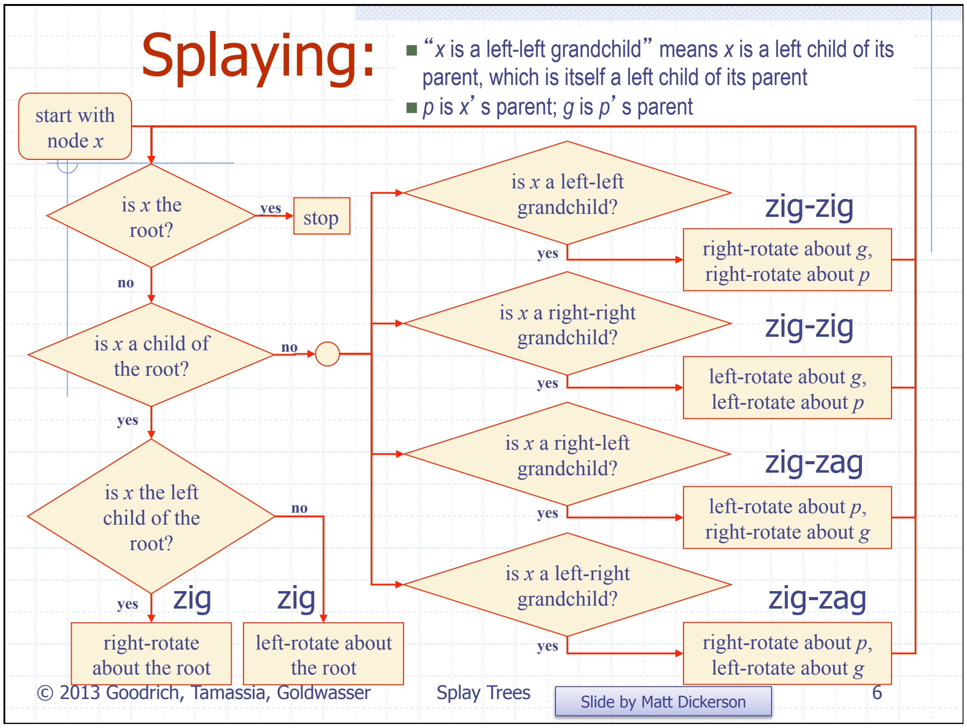
- right rotation
  - makes the left child *x* of a node *y* into *y*' s parent; *y* becomes the right child of *x*
- left rotation
  - makes the right child *y* of a node *x* into *x*' s parent; *x* becomes the left child of *y*

a right rotation about y

(structure of tree above y is not modified)

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? — yes → zig: right-rotate about the root; no → zig: left-rotate about the root

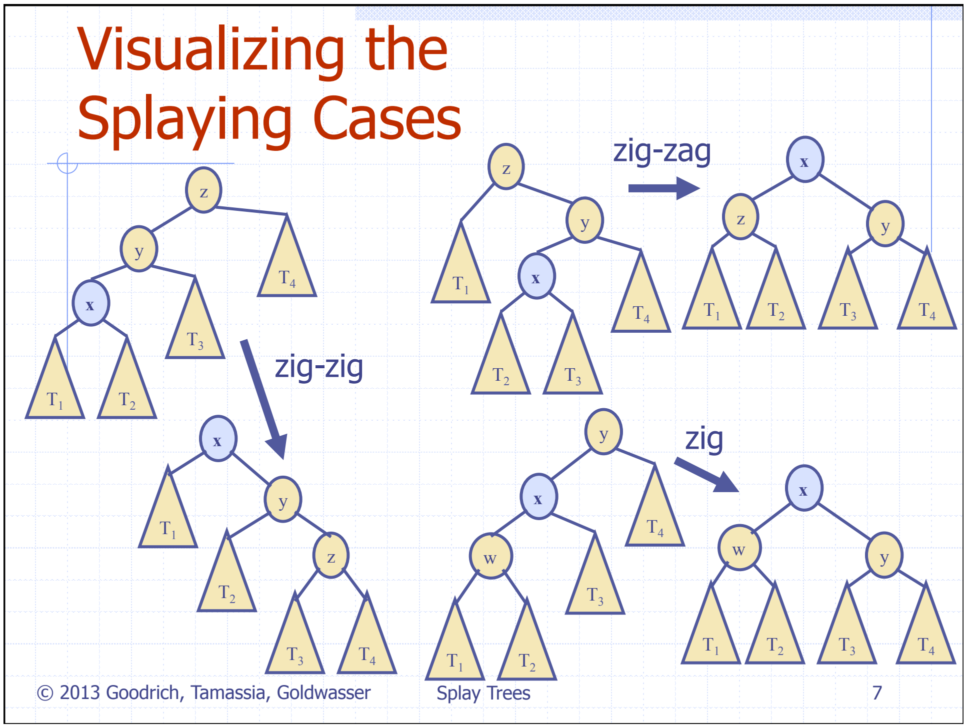
is *x* a left-left grandchild? — yes → zig-zig: right-rotate about *g*, right-rotate about *p*

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*

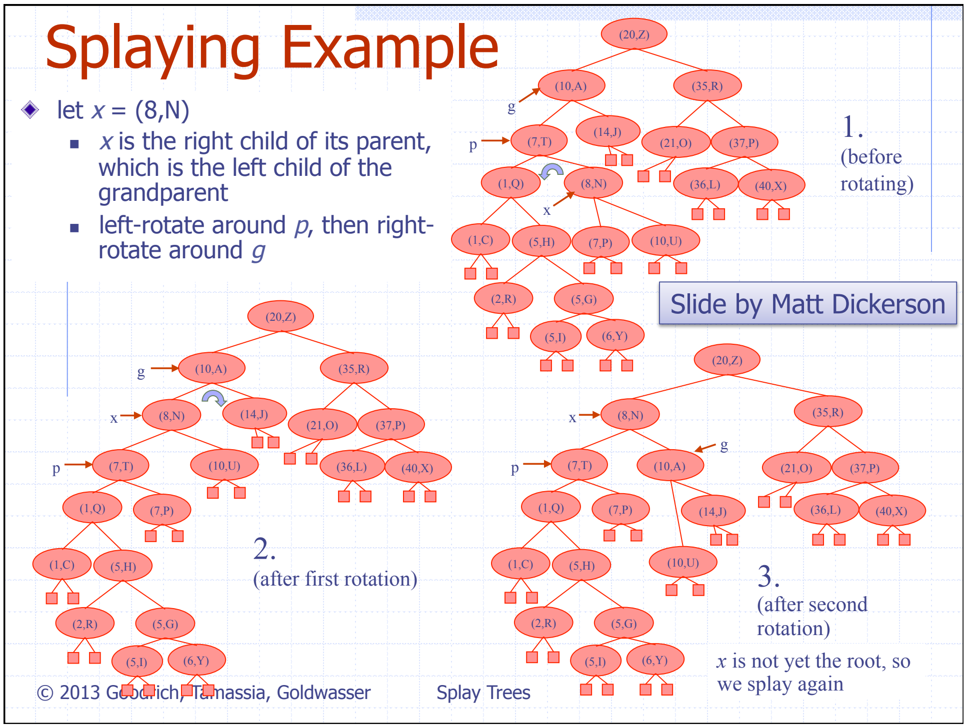
is *x* a left-right grandchild? — yes → zig-zag: right-rotate about *p*, left-rotate about *g*

Slide by Matt Dickerson



Visualizing the Splaying Cases

© 2013 Goodrich, Tamassia, Goldwasser Splay Trees 7



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)

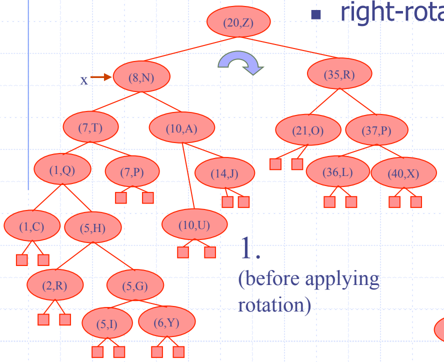
x is not yet the root, so we splay again

Slide by Matt Dickerson

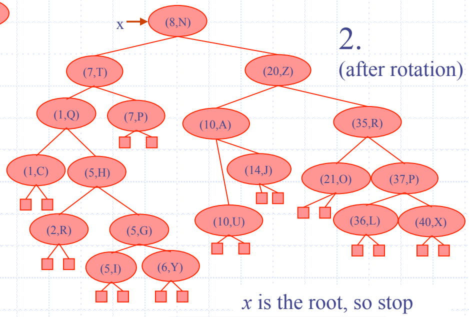
© 2013 Goodrich, Tamassia, Goldwasser Splay Trees

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

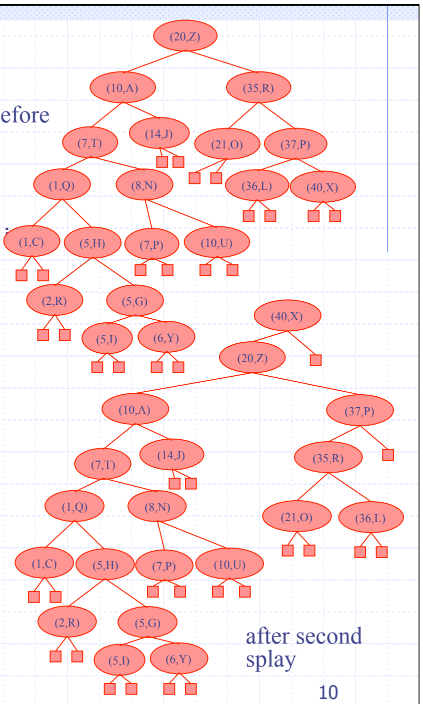
© 2013 Goodrich, Tamassia, Goldwasser Splay Trees

Slide by Matt Dickerson

Slide by Matt Dickerson

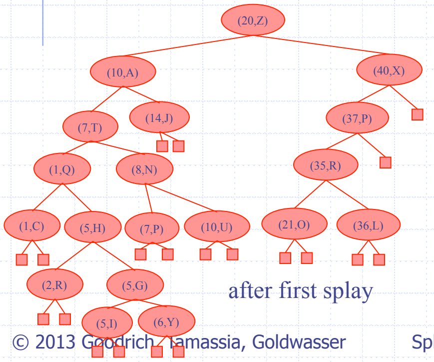
Example Result of Splaying

- 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

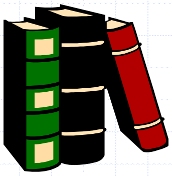
after second splay



after first splay

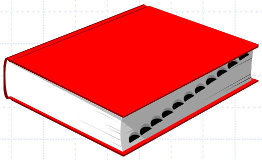
© 2013 Goodrich, Tamassia, Goldwasser Splay Trees

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)

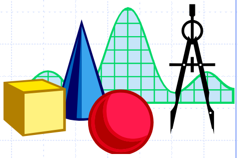
Splay Trees & Ordered Dictionaries



- which nodes are splayed after each operation?

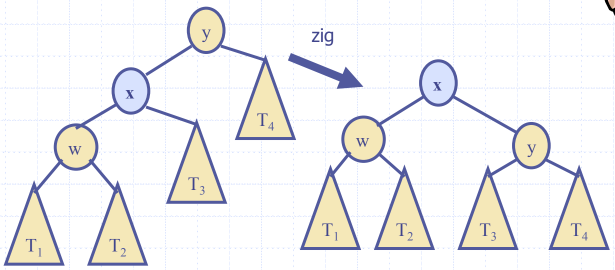
method	splay node
Search for k	if key found, use that node if key not found, use parent of ending external node
Insert (k,v)	use the new node containing the entry inserted
Remove item with key k	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

Amortized Analysis of Splay Trees

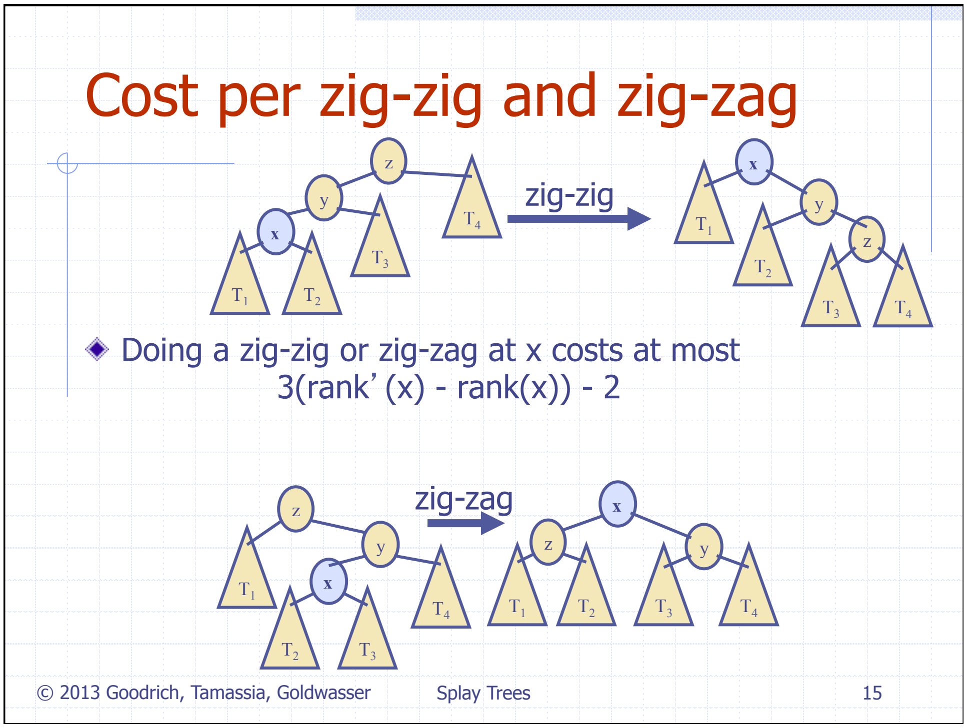


- Running time of each operation is proportional to time for splaying.
- Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.
- Costs: zig = \$1, zig-zig = \$2, zig-zag = \$2.
- Thus, cost for playing a node at depth d = \$d.
- Imagine that we store rank(v) cyber-dollars at each node v of the splay tree (just for the sake of analysis).

Cost per zig

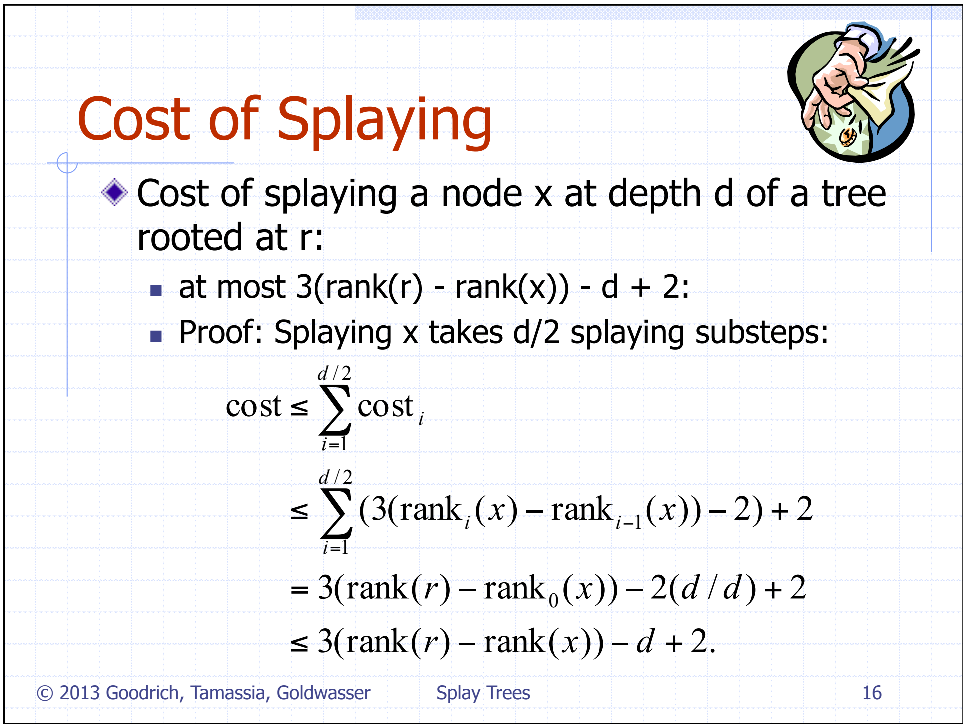


- Doing a zig at x costs at most rank'(x) - rank(x):
  - cost = rank'(x) + rank'(y) - rank(y) - rank(x)
    $\leq$ rank'(x) - rank(x).



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



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:

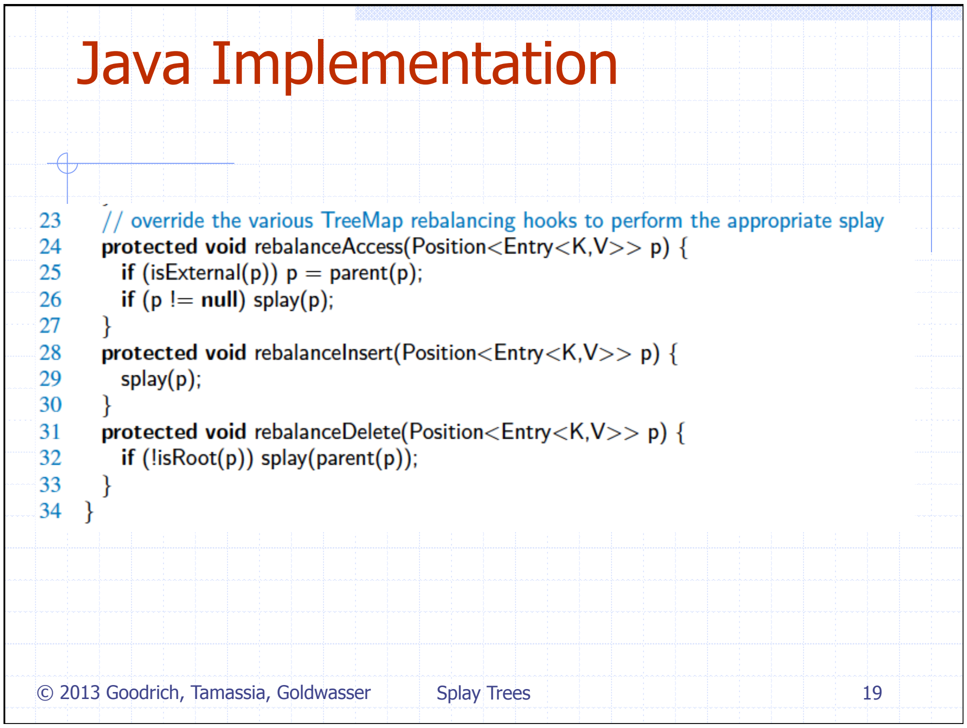
$$
\begin{aligned}
\text{cost} &\le \sum_{i=1}^{d/2} \text{cost}_i \\
&\le \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\
&= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2 \\
&\le 3(\text{rank}(r) - \text{rank}(x)) - d + 2.
\end{aligned}
$$

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 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

Java Implementation

```
/** An implementation of a sorted map using a splay tree. */
public class SplayTreeMap<K,V> extends TreeMap<K,V> {
  /** Constructs an empty map using the natural ordering of keys. */
  public SplayTreeMap() { super(); }
  /** Constructs an empty map using the given comparator to order keys. */
  public SplayTreeMap(Comparator<K> comp) { super(comp); }
  /** Utility used to rebalance after a map operation. */
  private void splay(Position<Entry<K,V>> p) {
    while (!isRoot(p)) {
      Position<Entry<K,V>> parent = parent(p);
      Position<Entry<K,V>> grand = parent(parent);
      if (grand == null)                                   // zig case
        rotate(p);
      else if ((parent == left(grand)) == (p == left(parent))) {   // zig-zig case
        rotate(parent);           // move PARENT upward
        rotate(p);                // then move p upward
      } else {                                             // zig-zag case
        rotate(p);                // move p upward
        rotate(p);                // move p upward again
      }
    }
  }
```



Java Implementation

```
  // override the various TreeMap rebalancing hooks to perform the appropriate splay
  protected void rebalanceAccess(Position<Entry<K,V>> p) {
    if (isExternal(p)) p = parent(p);
    if (p != null) splay(p);
  }
  protected void rebalanceInsert(Position<Entry<K,V>> p) {
    splay(p);
  }
  protected void rebalanceDelete(Position<Entry<K,V>> p) {
    if (!isRoot(p)) splay(parent(p));
  }
}
```