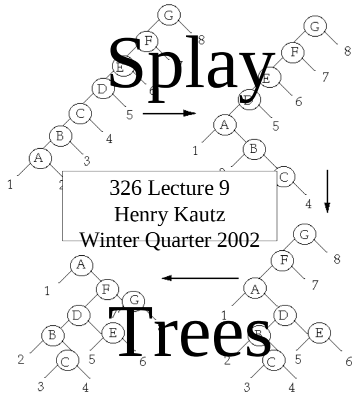
Splay Trees



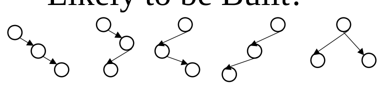
326 Lecture 9
Henry Kautz
Winter Quarter 2002

First, a Random Question

1. The average depth of a node in randomly-built binary search tree on *n* nodes is O(log n).
   - we showed this in class
2. The average height of a randomly-built binary search tree on *n* nodes is O(log n).
   - a stronger statement, still true
3. The average height of a binary tree with *n* nodes is $\theta(\sqrt{n})$

Why is this not contradictory?

Are All Binary Trees Equally Likely to be Built?



- How many ways are there to sequence the numbers 1, 2, 3?
- Which of the binary trees can be built in more than one way?

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for height info.
2. Asymptotically faster but usually slower in practice!

More Treelike Data Structures

- Today : Splay Trees
  - Fast both in amortized analysis and in practice
    - Are used in the *kernel* of NT for keep track of process information!
    - Invented by Sleator and Tarjan (1985)
  - Good "locality"
  - Details:
    - Weiss 4.5 (basic splay trees)
    - 11.5 (amortized analysis)
    - 12.1 (better "top down" implementation)
- Coming up: B-Trees

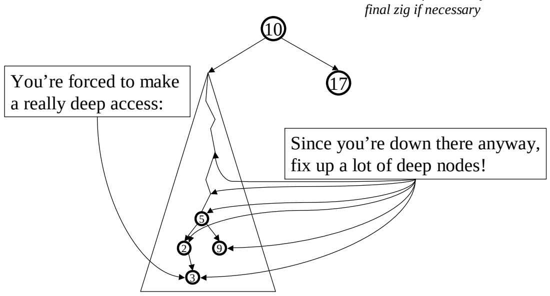
Splay Trees

"Blind" rebalancing – no height info kept

- amortized time for all operations is O(log n)
- worst case time is O(n)
- insert/find always rotates node *to the root*!
  - Good locality – most common keys move high in tree

Idea

move n to root by series of zig-zag and zig-zig rotations, followed by a final zig if necessary

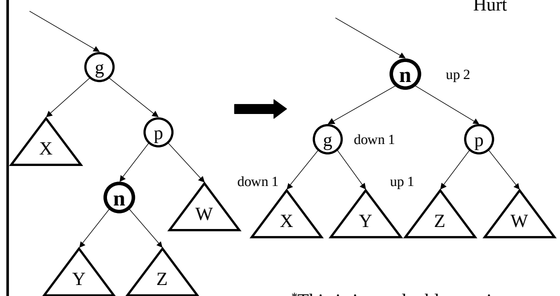


Zig-Zag*

Helped

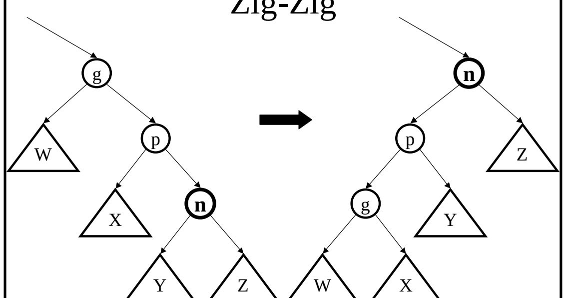
Unchanged

Hurt



*This is just a double rotation

Zig-Zig



Why Splaying Helps

- Node n and its children are always helped (raised)
- Except for final zig, nodes that are *hurt* by a zig-zag or zig-zig are later *helped* by a rotation higher up the tree!
- **Result**:
  - shallow (zig) nodes may increase depth by one or two
  - helped nodes may decrease depth by a large amount
- 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

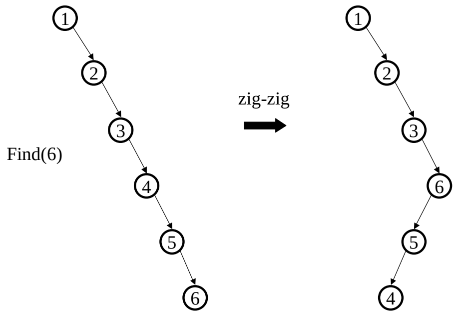
Locality

- "Locality" – if an item is accessed, it is likely to be accessed again soon
  - *Why?*
- Assume $m \geq n$ access in a tree of size n
  - Total amortized time O($m \log n$)
  - O($\log n$) per access on average
- Suppose only k distinct items are accessed in the m accesses.
  - Time is O($m \log k + n \log n$)
  - *What would an AVL tree do?*

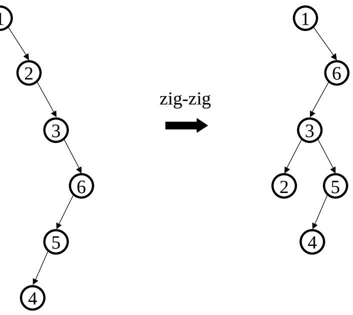
Locality

- "Locality" – if an item is accessed, it is likely to be accessed again soon
  - *Why?*
- Assume $m \geq n$ access in a tree of size n
  - Total amortized time O($m \log n$)
  - O($\log n$) per access on average
- Suppose only k distinct items are accessed in the m accesses.
  - Time is O($m \log k + n \log n$)
  - compare with O($m \log n$) for AVL tree

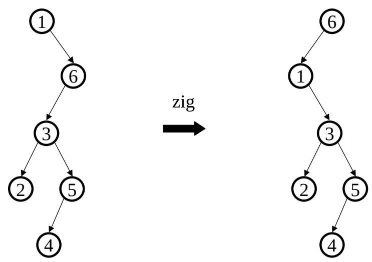
Splaying Example



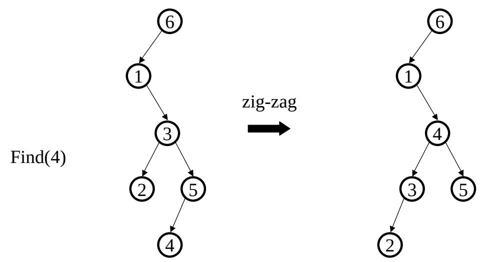
Still Splaying 6



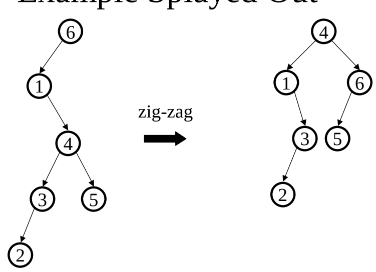
Almost There, Stay on Target



Splay Again



Example Splayed Out



Splay Operations: Insert

- To insert, could do an ordinary BST insert
  - but would not fix up tree
  - A BST insert followed by a find (splay)?
- Better idea: do the splay before the insert!
- How?

Splay Operations: Insert

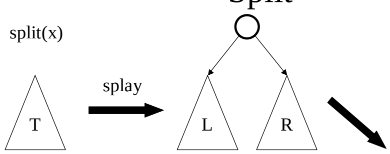
- To insert, could do an ordinary BST insert
  - but would not fix up tree
  - A BST insert followed by a find (splay)?
- Better idea: do the splay before the insert!
- How?
- Split(T, x) creates two BSTs L and R:
  - all elements of T are in either L or R (`T = L ∪ R`)
  - all elements in L are ≤ x
  - all elements in R are ≥ x
  - L and R share no elements (`L ∩ R = ∅`)

Then how do we do the insert?

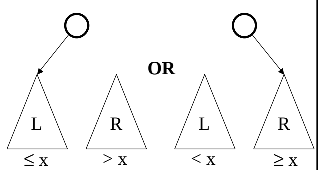
Splitting in Splay Trees

- How can we split? (SPOILERS below ^L)
  - 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?

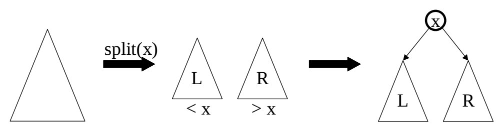
Split



To split do a find on x: if x is in T then splay it to the root, otherwise splay the last node found to the root. After splaying split the tree at the root

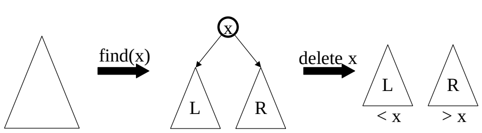


Back to Insert




```
Insert(x):
  Split on x
  Join subtrees using x as root!
```

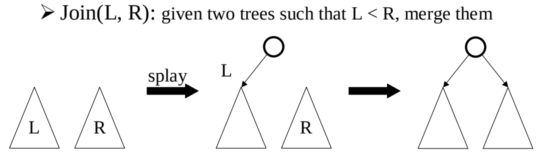
Splay Operations: Delete



Now what?

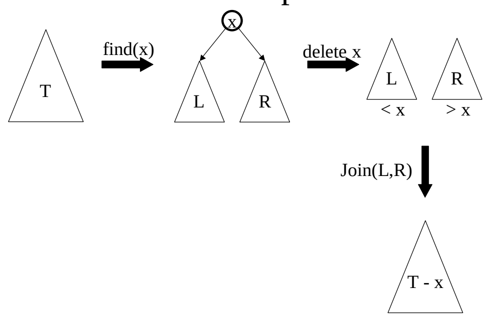
Join

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



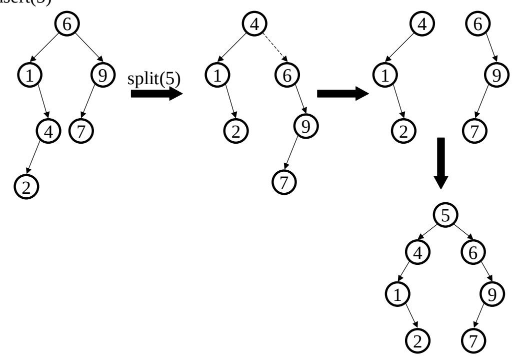
- Splay on the maximum element in L then attach R

Delete Completed



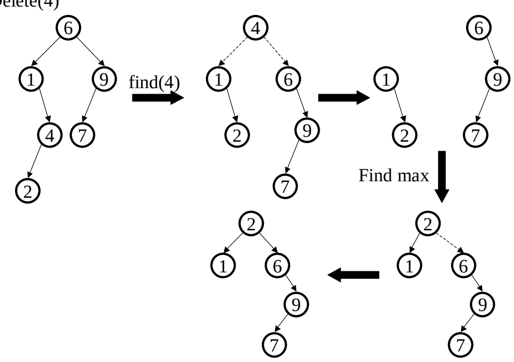
Insert Example

Insert(5)



Delete Example

Delete(4)



For Wednesday

- Read 4.7
- You should be well on your way to completing assignment 3