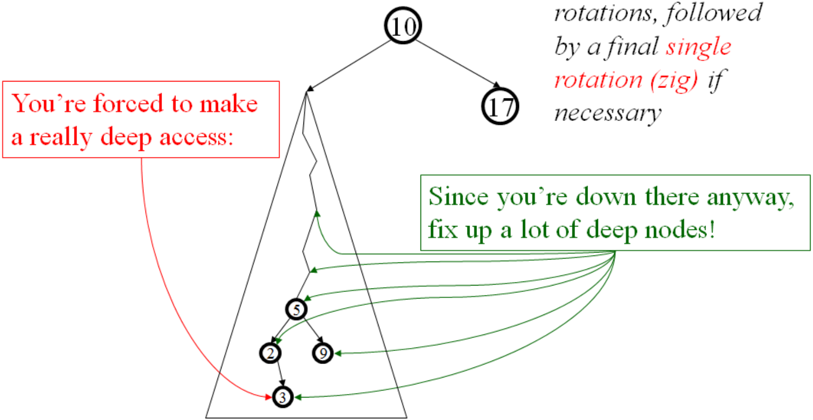
Splay Trees

Basic Idea

- Invented by Sleator and Tarjan (1985)
- Blind rebalancing – no height info kept!
- Worst-case time per operation is $O(n)$
- Worst-case *amortized time* is $O(\log n)$
- Insert/find always rotates node to the root!
- Good locality:
  - Most commonly accessed keys move high in tree – become easier and easier to find
  - Incorporate *Move-to-Top* strategy

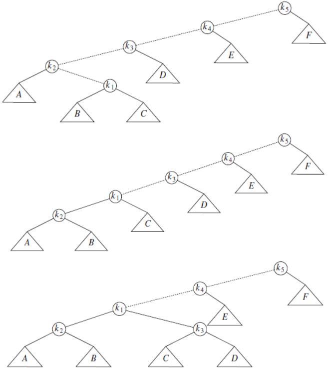
Idea

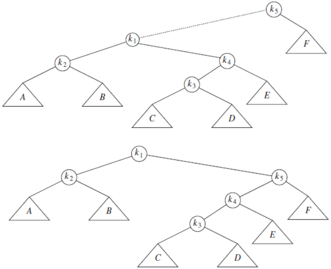
move n to root by series of zig-zag and zig-zig rotations, followed by a final single rotation (zig) if necessary



A Simple Idea (That Does Not Work)

- Perform single rotations bottom up.



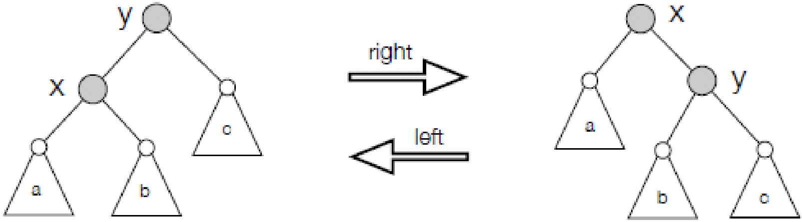


- There is a sequence of M operations requiring $\Omega(M \cdot N)$ time, so this idea is not quite good enough.

Splaying

Splay(x): do following rotations until x is the root.

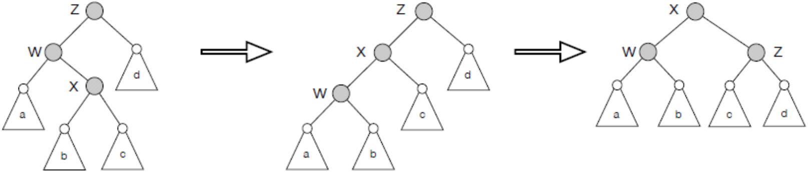
- right (or left): if x has no grandparent.



Right rotation at x (and left rotation at y)

Splaying (Cont'd)

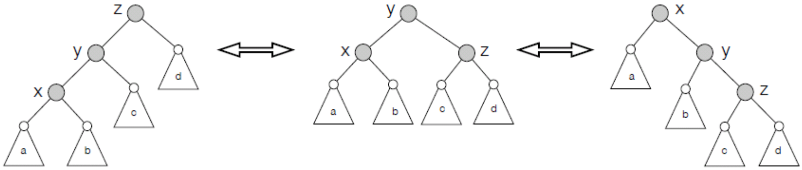
- zig-zag (or zag-zig): if one of x, $p(x)$ is a left child and the other is a right child.



zig-zag at x

Splaying (Cont'd)

- zig-zig: if x and $p(x)$ are either both left children or both right children.



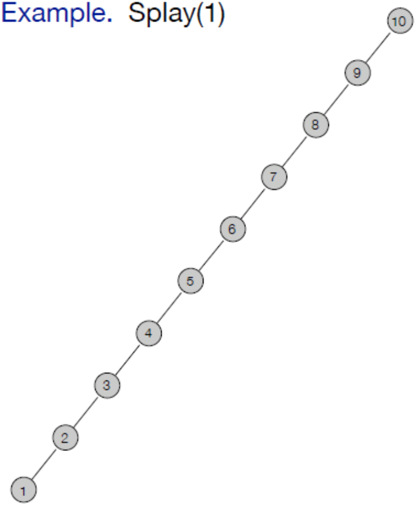
zig-zig at x

Why Splaying Helps

- Node n and its children are always helped (raised)
- Except for last step, nodes that are hurt by a *zig-zag* or *zig-zig* are later helped by a rotation higher up the tree!
- Result:
  - shallow nodes may increase depth by one or two
  - helped nodes decrease depth by a large amount
- If a node n on the access path is at depth d before the splay, it is at about depth $d/2$ after the splay
  - Exceptions are the root, the child of the root, and the node splayed

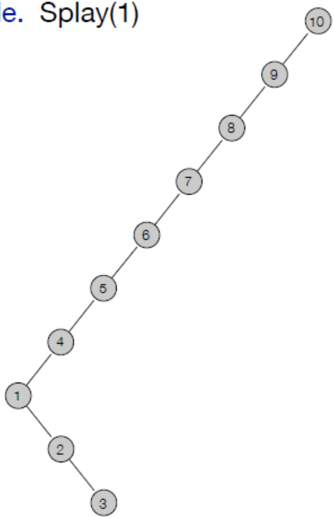
Splaying Example

- Example. Splay(1)



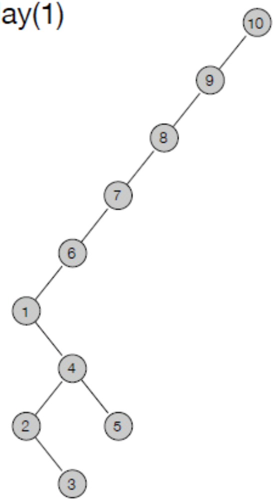
Splaying Example (Cont'd)

- Example. Splay(1)



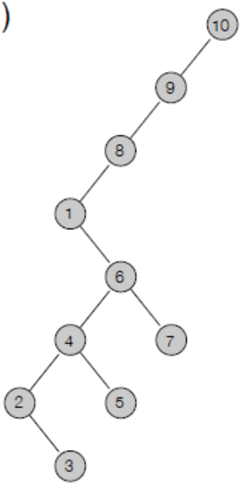
Splaying Example (Cont'd)

- Example. Splay(1)



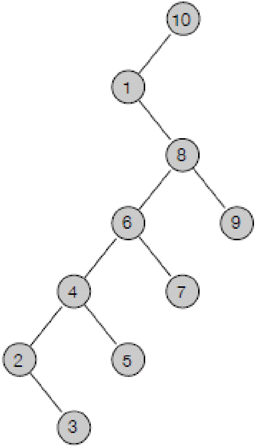
Splaying Example (Cont'd)

- Example. Splay(1)



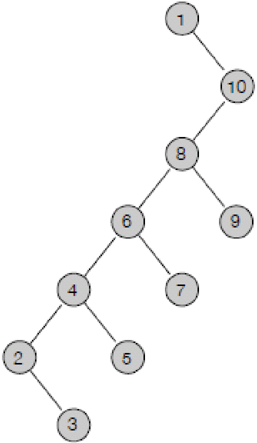
Splaying Example (Cont'd)

- Example. Splay(1)



Splaying Example (Cont'd)

- Example. Splay(1)



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 worst case time is $O(m \log n)$
  - $O(\log n)$ per access amortized time
- Suppose only k distinct items are accessed in the m accesses.
  - Time is $O(n \log n + m \log k)$
    - ★ $O(n \log n)$ – getting those k items near root
    - ★ $m \log k$ – those k items are all at the top of the tree
  - Compare with $O(m \log n)$ for AVL tree

Splay Operations: Insert

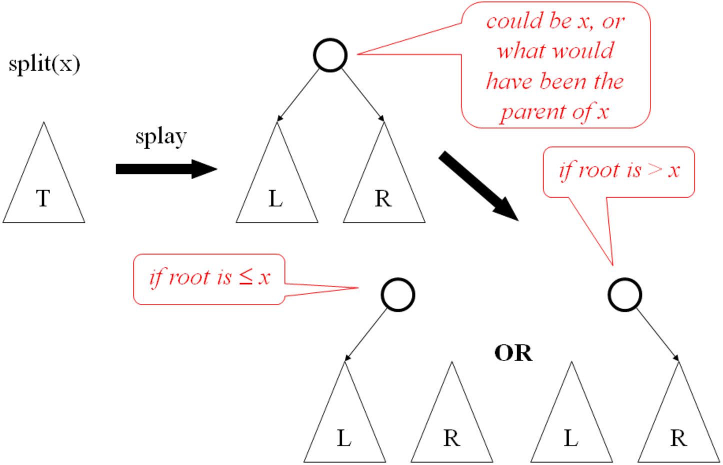
- To insert, could do an ordinary BST insert
  - but would not fix up tree
  - A BST insert followed by a find (splay)?
- Better idea: do the splay before the insert!
- How?
  Split(T, x) creates two BSTs L and R
  - All elements of T are in either L or R
  - All elements in L are $\leq x$
  - All elements in R are $> x$
  - L and R share no elements

  Insert as root, with children L and R

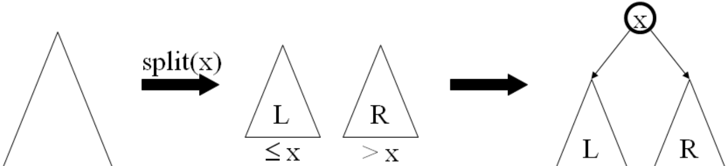
Splitting in Splay Trees

- How can we split?
  - We have the splay operation
  - We can find x or the parent of where x would be if we were to insert it as an ordinary BST
  - We can splay x or the parent to the root
  - Then break one of the links from the root to a child

Split

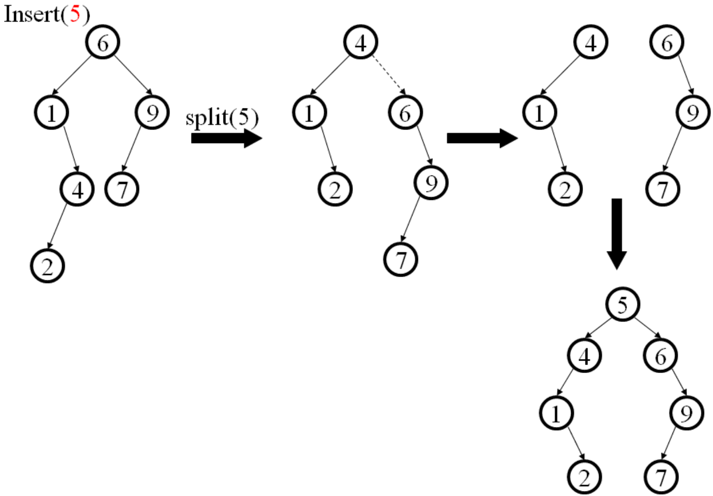

split(x)
T
splay
L
R
could be x, or what would have been the parent of x
if root is > x
if root is ≤ x
L
R
OR
L
R

Back to Insert

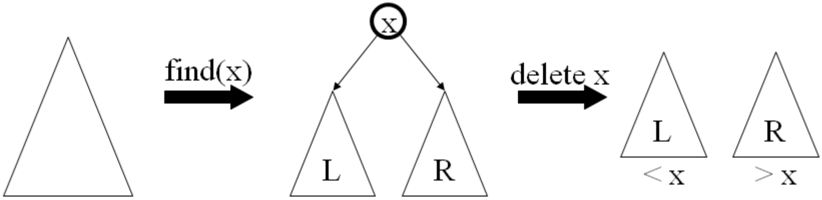



```
Insert(x):
  Split on x
  Join subtrees using x as root
```

Insert Example



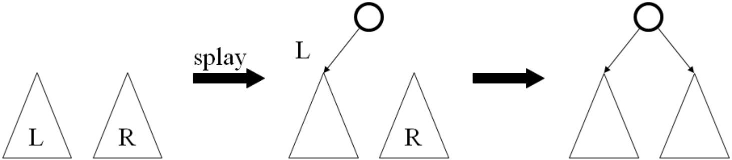
Delete



Now what?

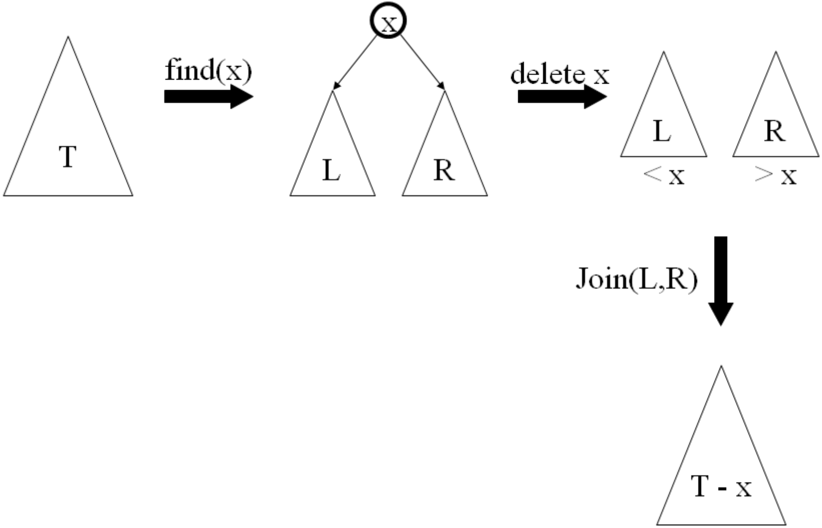
Join

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

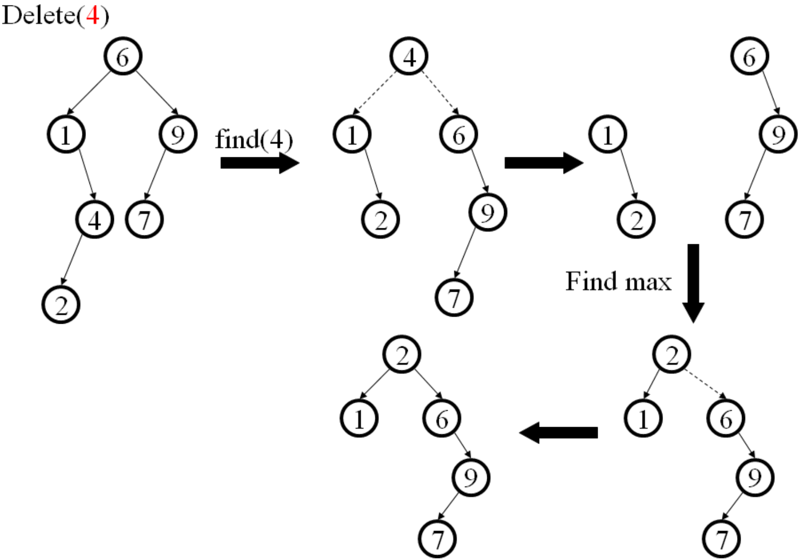


- Splay on the maximum element in L then attach R

Delete Completed

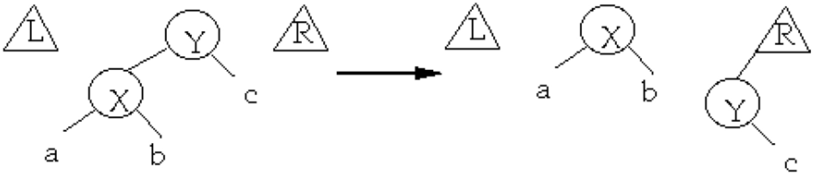


Delete Example

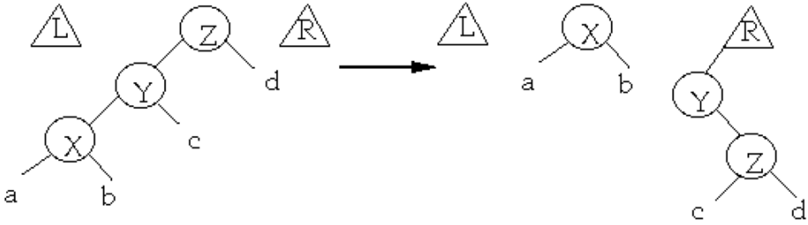


Top-down Splay

- **Case 1**: X is the node we are splaying

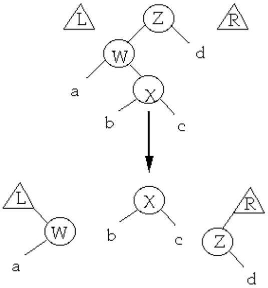


- **Case 2**: (zig-zig) The node we are splaying is in the subtree rooted at X

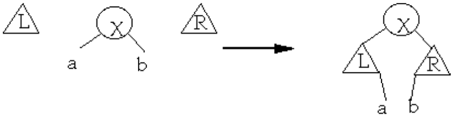


Top-down Splay (Cont'd)

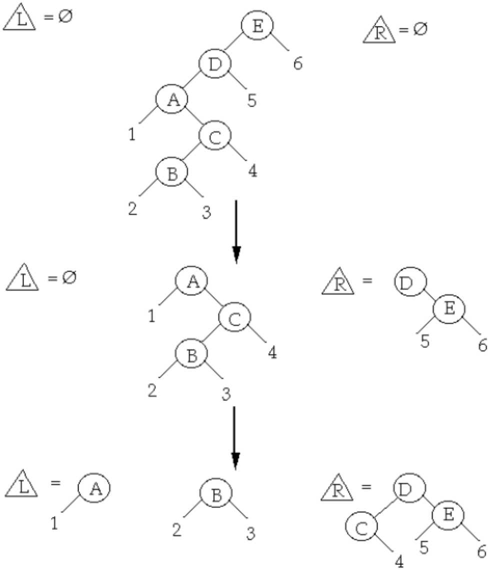
- **Case 3**: (zig-zag) The node we are splaying is in the subtree rooted at X



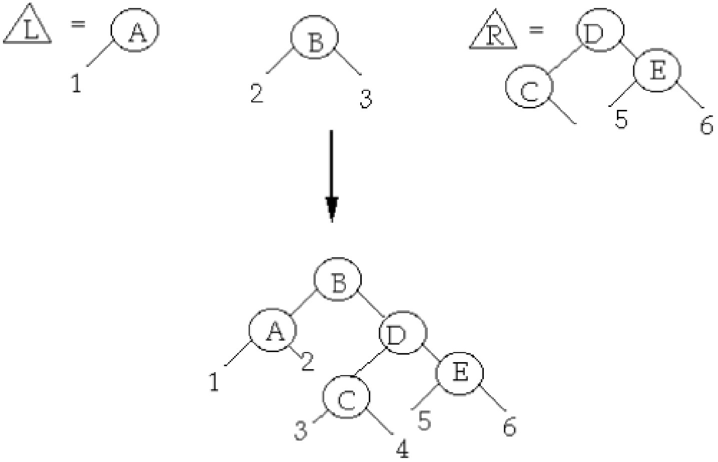
- **Case 4**: (the last step) X is the node we wish to splay



Example Splay at B



Example Splay at B (Cont'd)



Summary

- Splay trees are arguably the most practical kind of self-balancing trees
- If number of finds is much larger than n, then locality is crucial!
- Also supports efficient Split and Join operations – useful for other tasks