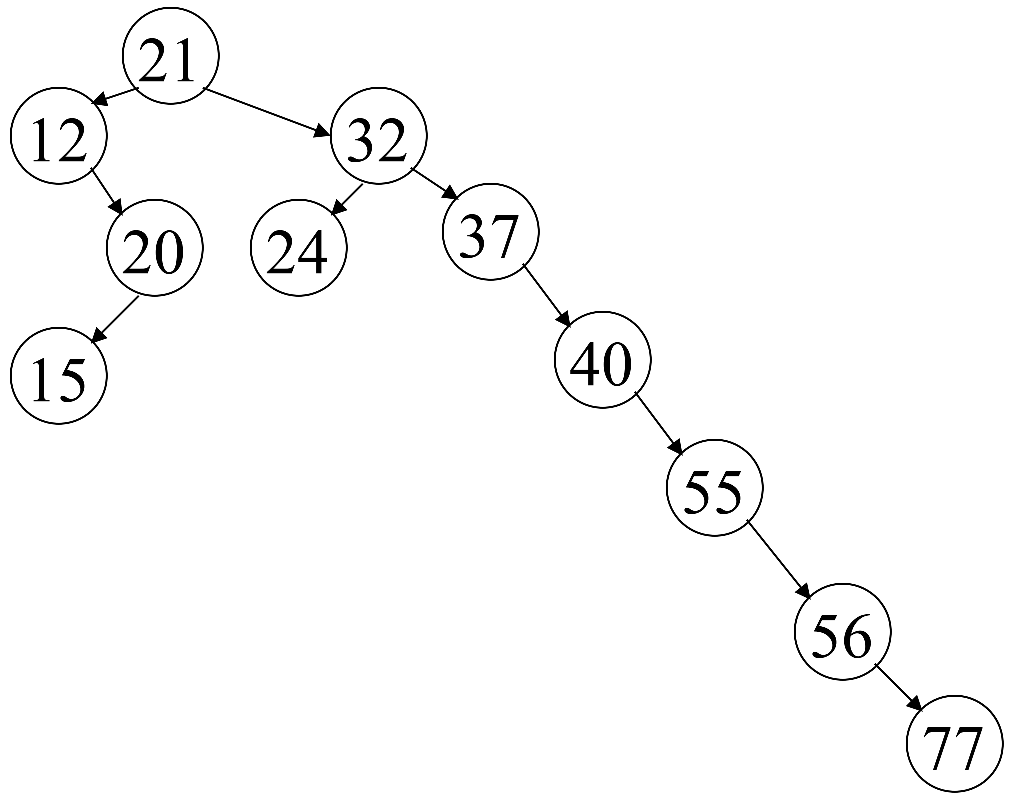
Splay Trees

Splay Trees

Problems with BSTs

- Because the shape of a BST is determined by the order that data is inserted, we run the risk of trees that are essentially lists



BST Sequence of Operations

- Worst case for a single BST operation is *O*(N)
- Not so bad if this happens only occasionally
- BUT...its not uncommon for an entire sequence of "bad" operations to occur. In this case, a sequence of M operations take *O*(M * N) time and the time for the sequence of operations becomes noticeable.

Splay Tree Sequence of Operations

- Splay trees guarantee that a sequence of M operations takes at most *O*(M * lg N) time.
- We say that the splay tree has ***amortized*** running time of *O*(lg N) cost per operation. Over a long sequence of operations, some may take more than lg N time, some will take less.

Splay Tree Sequence of Operations (cont.)

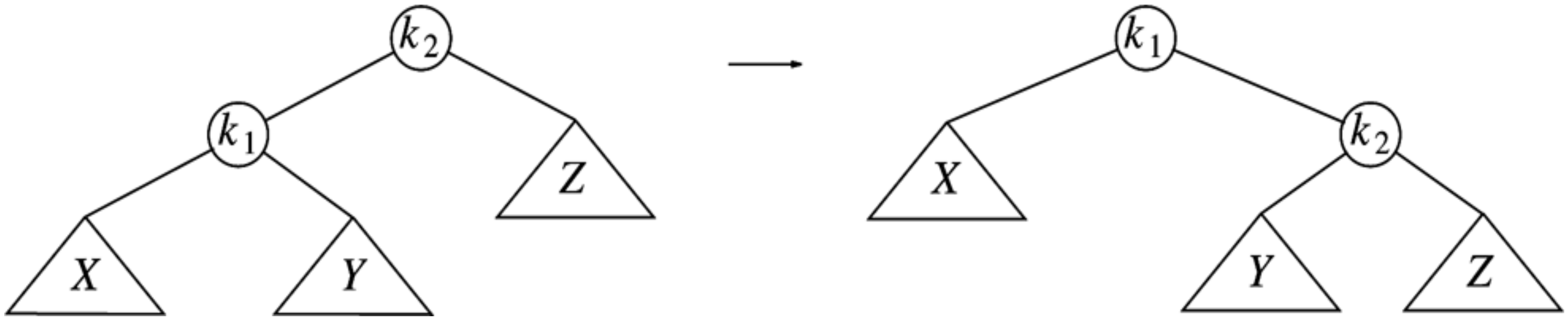
- Does not preclude the possibility that any particular operation is still *O*(N) in the worst case.
  - Therefore, amortized O(lg N) not as good as worst case O(lg N)
  - But, the effect is the same – there is no "bad" sequence of operations or bad input sequences.
- If any particular operation is *O*(N) and we still want amortized *O*(lg N) performance, then whenever a node is accessed, it must be moved. Otherwise its access time is always *O*(N).

Splay Trees

- The basic idea of the splay tree is that every time a node is accessed, it is pushed to the root by a series of ***tree rotations.*** This series of tree rotations is knowing as “splaying”.

- If the node being “splayed” is deep, many nodes on the path to that node are also deep and by restructuring the tree, we make access to all of those nodes cheaper in the future.

Basic "Single" Rotation in a BST



Rotating k_1 around k_2

Assuming that the tree on the left is a BST, how can we verify that the tree on the right is still a valid BST?

Note that the rotation can be performed in either direction.

Under the Hood

how rotation *really* works

In the previous slide, rotating k_1 around k_2 is really nothing more than performing these 2 relinking statements:

k_2.left = k_1.right; and

k_1.right = k_2;

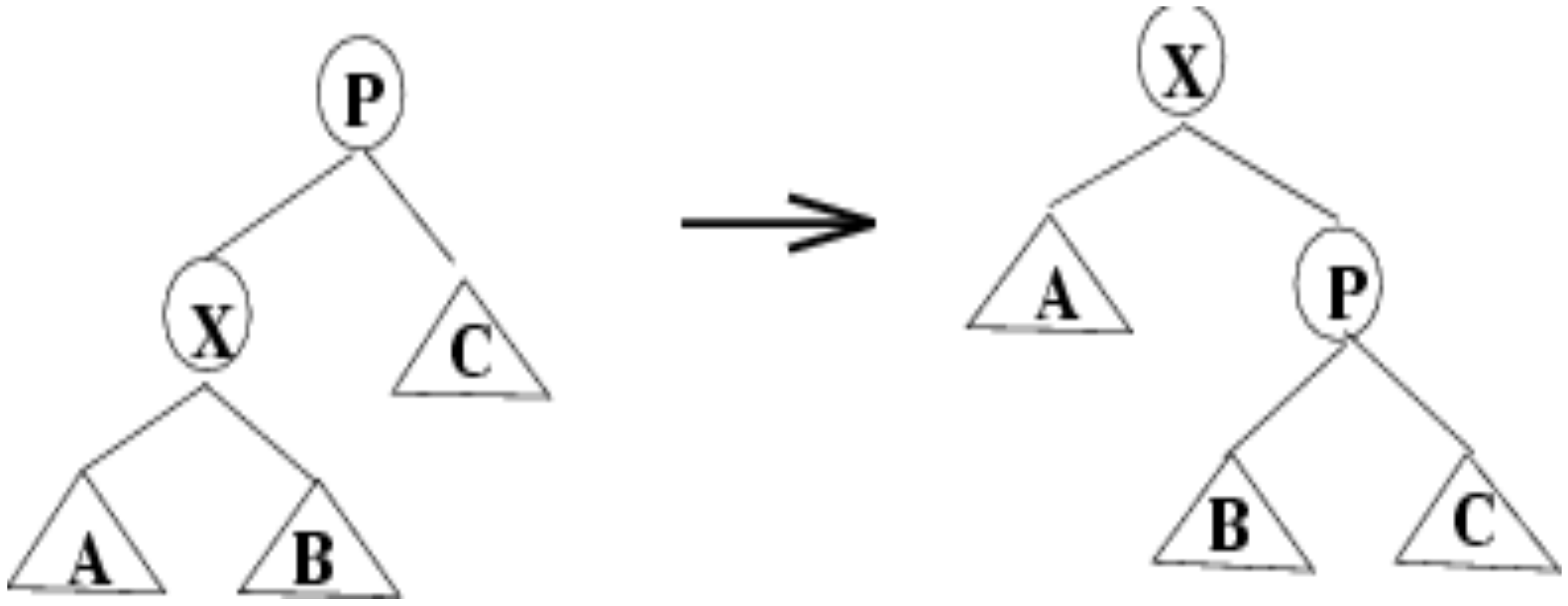
Now, k_2 is the parent of k_1, and the diagram on the right just shows the nodes in their proper perspective.

You should work out the code to do all of the double rotations in the splay tree section (zig-zig, zig-zag) of the text and the slides that follow

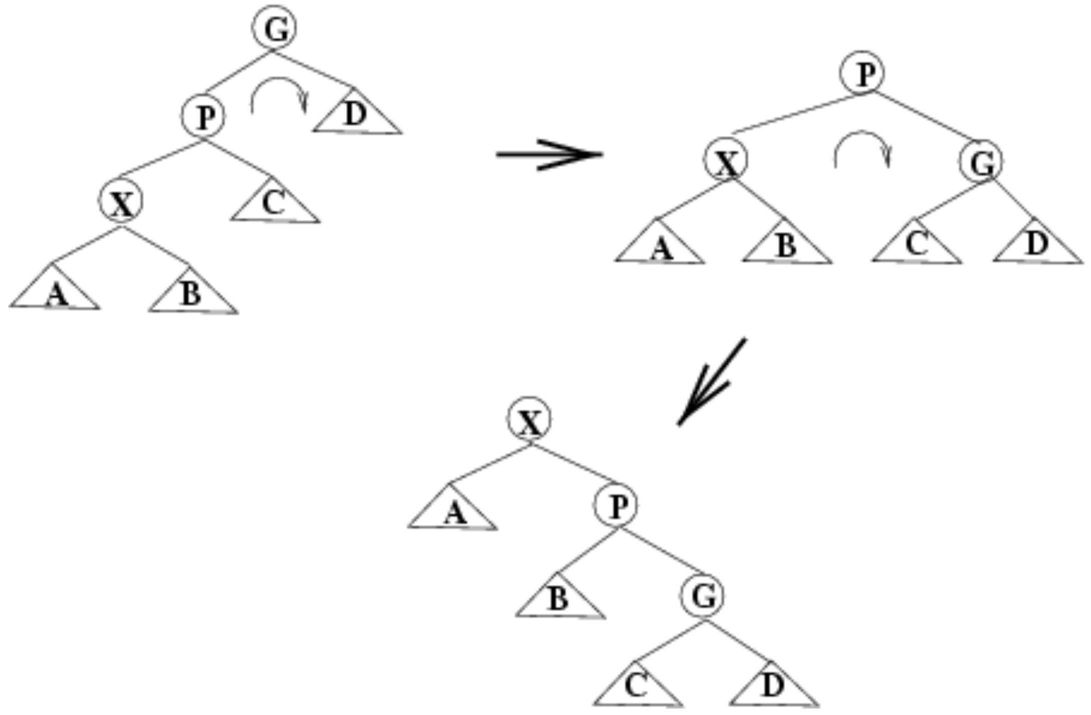
Splay Operation

- To "splay node x", traverse up the tree from node x to root, rotating along the way until x is the root. For each rotation:
  - If x is the root, do nothing.
  - If x has no grandparent, rotate x about its parent.
  - If x has a grandparent,
    - if x and its parent are both left children or both right children, rotate the parent about the grandparent, then rotate x about its parent.
    - if x and its parent are opposite type children (one left and the other right), rotate x about its parent, then rotate x about its new parent (former grandparent).

Node has no grandparent - Zig

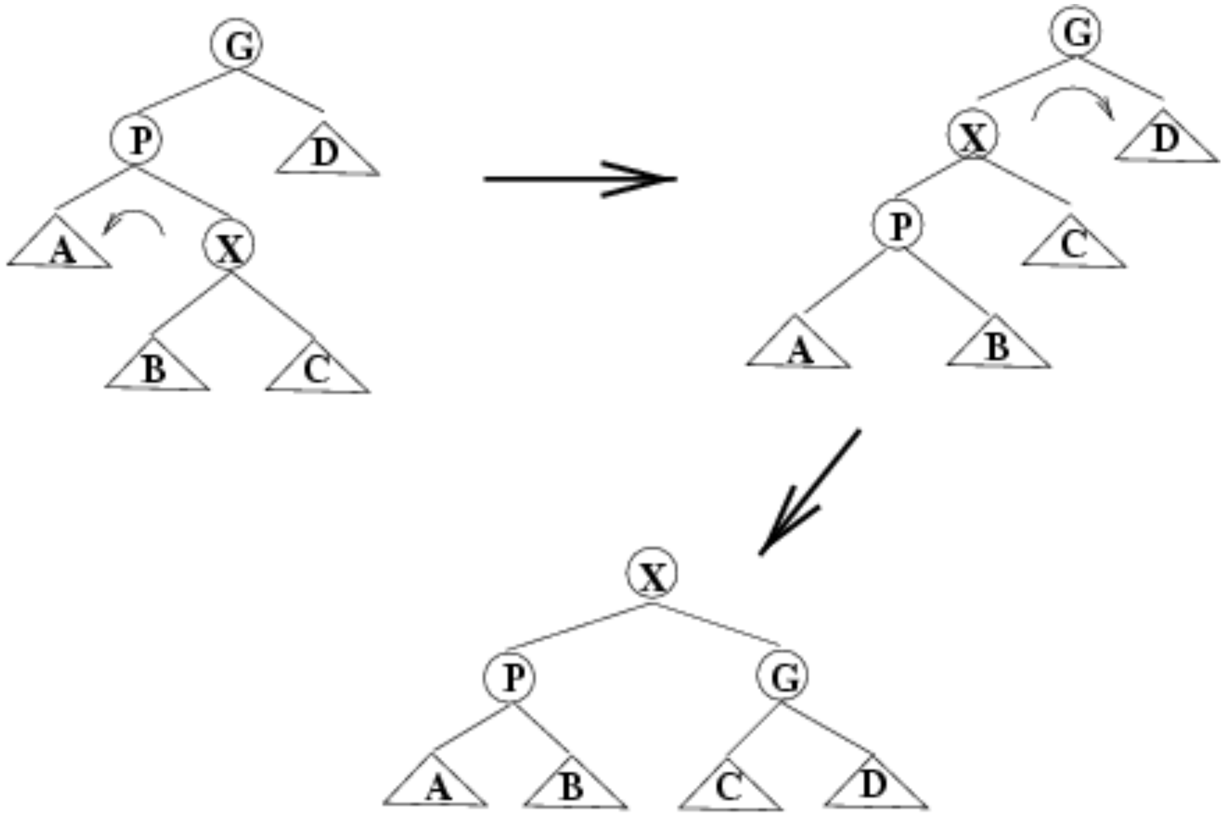


Node and Parent are Same Side (both left/right children) -- Zig-Zig



Rotate P around G, then X around P

Node and Parent are Different Sides (one is left, one is right child) -- Zig-Zag



Rotate X around P, then X around G

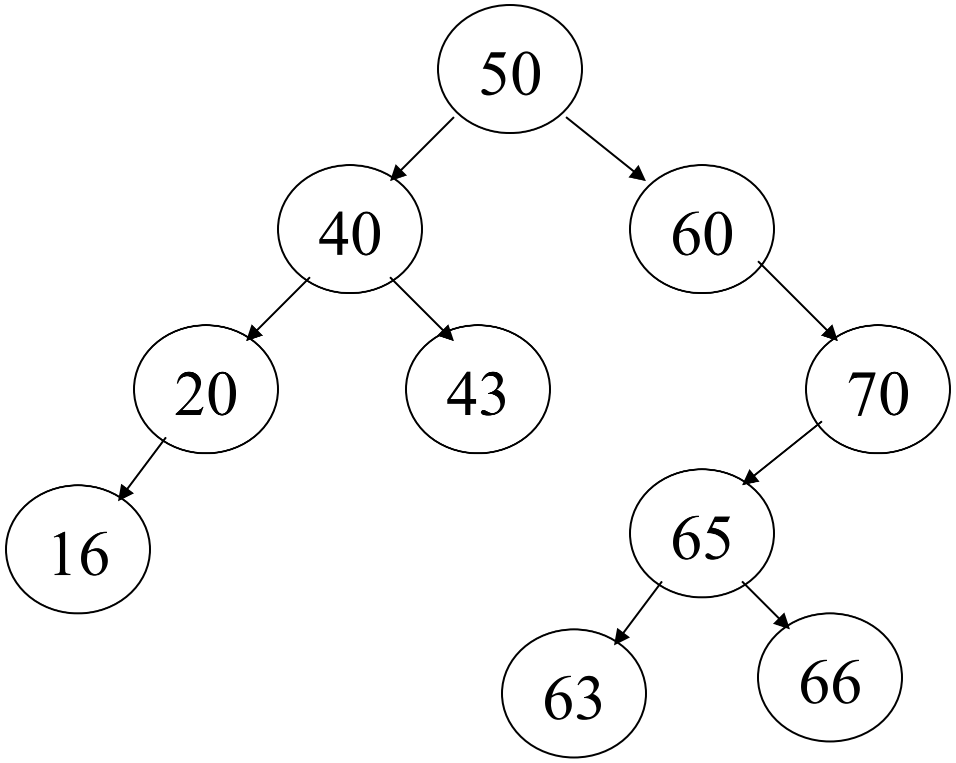
Operations in Splay Trees

- insert
  - first insert as in binary search tree
  - then splay inserted node
  - if there is a duplicate, the node holding the duplicate element is splayed
- find/contains
  - search for node
  - if found, splay it; otherwise splay last node accessed on the search path

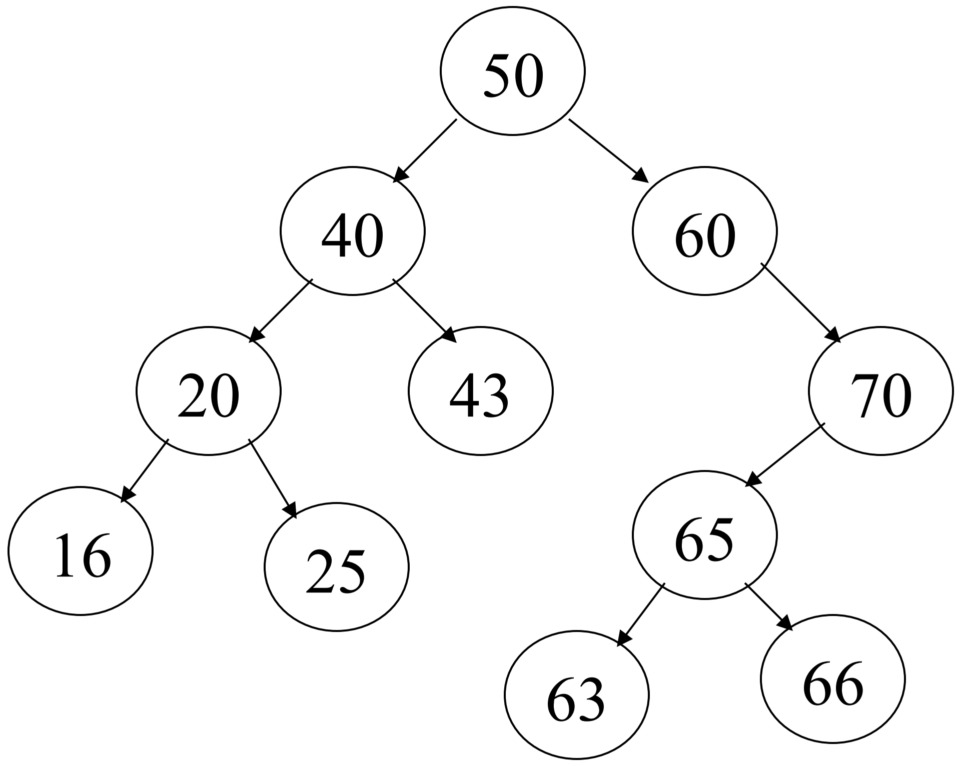
Operations on Splay Trees (cont)

- remove
  - splay element to be removed
    - if the element to be deleted is not in the tree, the node last visited on the search path is splayed
  - disconnect left and right subtrees from root
  - do one or both of:
    - splay max item in T_L (then T_L has no right child)
    - splay min item in T_R (then T_R has no left child)
  - connect other subtree to empty child of root

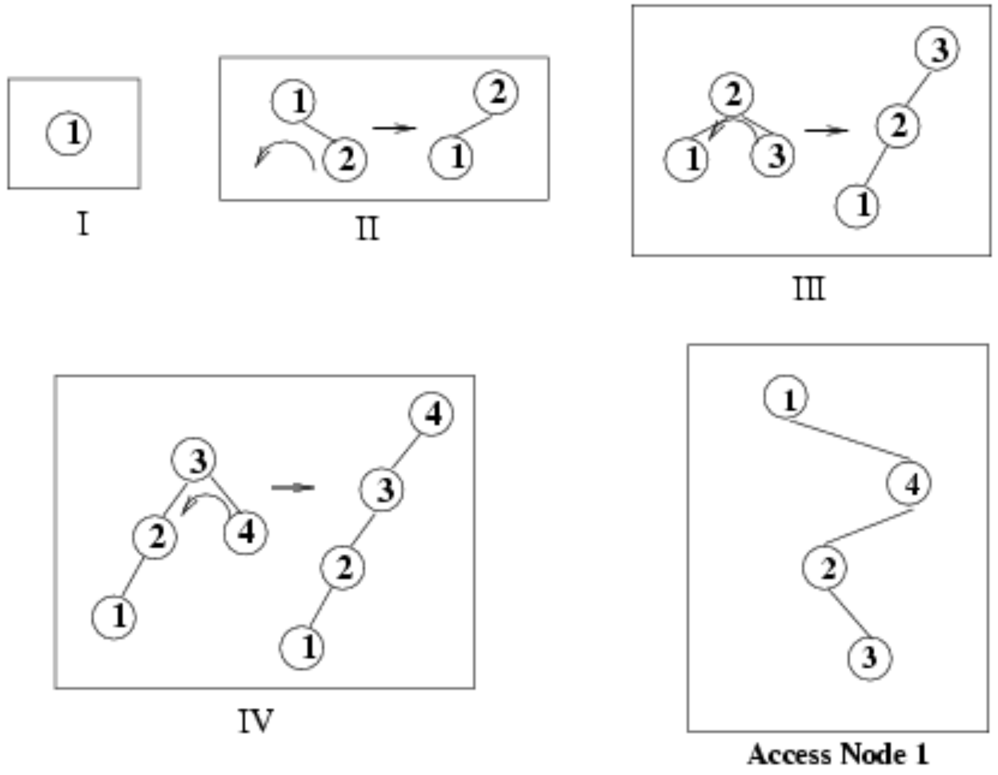
Exercise - find(65)



Exercise - remove(25)

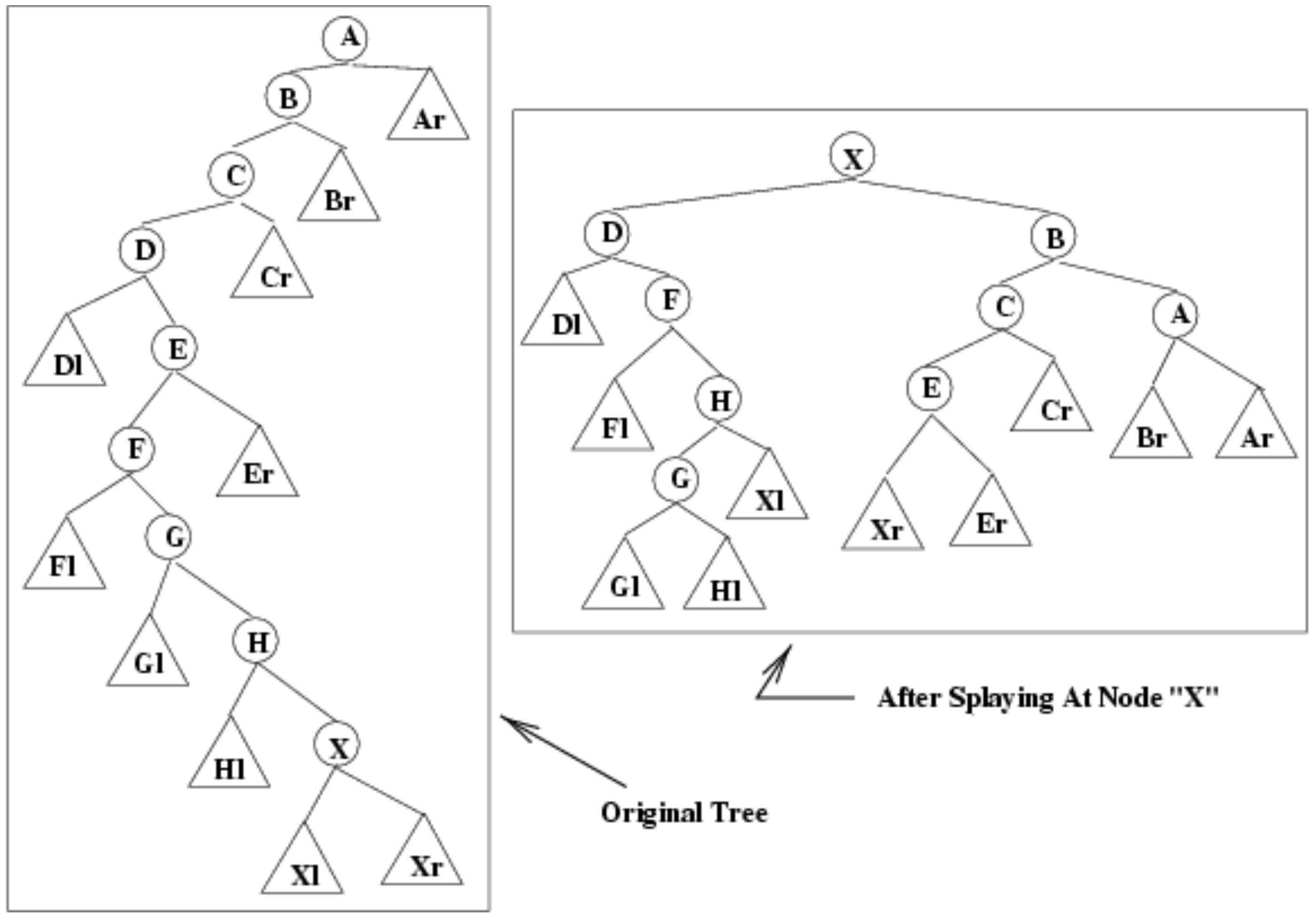

$$\begin{tikzcd}
 & & & 50 \arrow[dl] \arrow[dr] & & \\
 & & 40 \arrow[dl] \arrow[d] & & 60 \arrow[dr] & \\
 & 20 \arrow[dl] \arrow[d] & 43 & & & 70 \arrow[dl] \\
16 & 25 & & & 65 \arrow[dl] \arrow[d] & \\
 & & & 63 & 66 &
\end{tikzcd}$$

Insertion in order into a Splay Tree



In a BST, building a tree from N sorted elements was *O*(N^2). What is the performance of building a splay tree from N sorted elements?

An extreme example of splaying



Splay Tree Code

- The splaying operation is performed “up the tree” from the node to the root.
- How do we traverse “up” the tree?
- How do we know if X and P are both left/right children or are different sided children?
- How do we know if X has a grandparent?
- What disadvantages are there to this technique?

Top-Down Splay Trees

- Rather than write code that traverses both up and down the tree, "top-down" splay trees only traverse down the tree. On the way down, rotations are performed and the tree is split into three parts depending on the access path (zig, zig-zig, zig-zag) taken
  - X, the node currently being accessed
  - Left – all nodes less than X
  - Right – all nodes greater than X
- As we traverse down the tree, X, Left, and Right are reassembled
- This method is faster in practice, uses only *O*(1) extra space and still retains *O*(lg N) amortized running time.