THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

Splay Trees

CS 124 / Department of Computer Science

Locality of reference

- "Locality of reference" is just a fancy way of saying that we tend to use certain stuff more frequently within a certain period of time or within a certain place.
- This is also called the "principle of locality."
- Maybe June's desk looks a mess, but the papers she's working on are right on top.
- This is the idea behind caching. Put stuff we use frequently in a handy place. Chances are good we'll use it again soon.

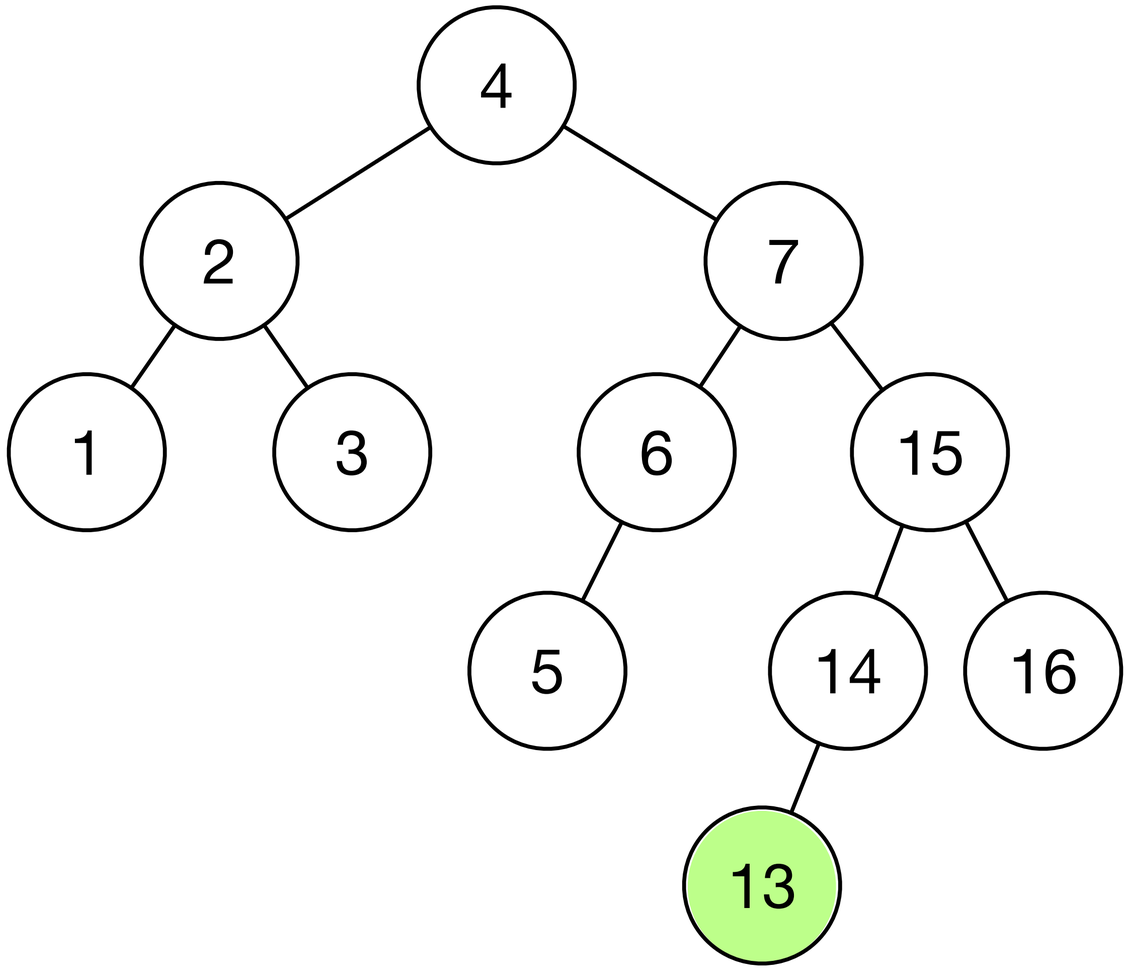
Splay Trees

- To leverage this "locality of reference", Sleator and Tarjan introduced splay trees in 1985.

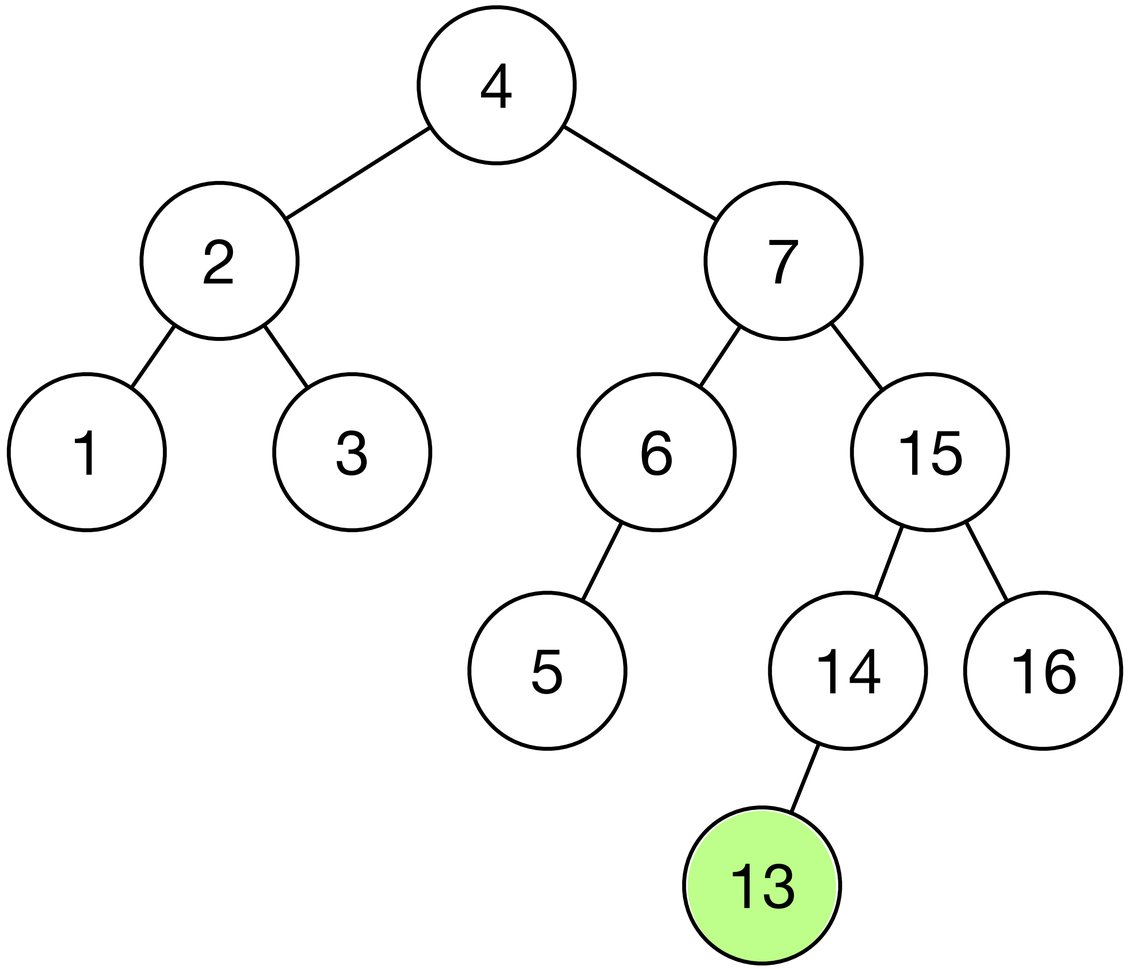
> "A splay tree is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $O(\log N)$ amortized time."*

* https://en.wikipedia.org/wiki/Splay_tree

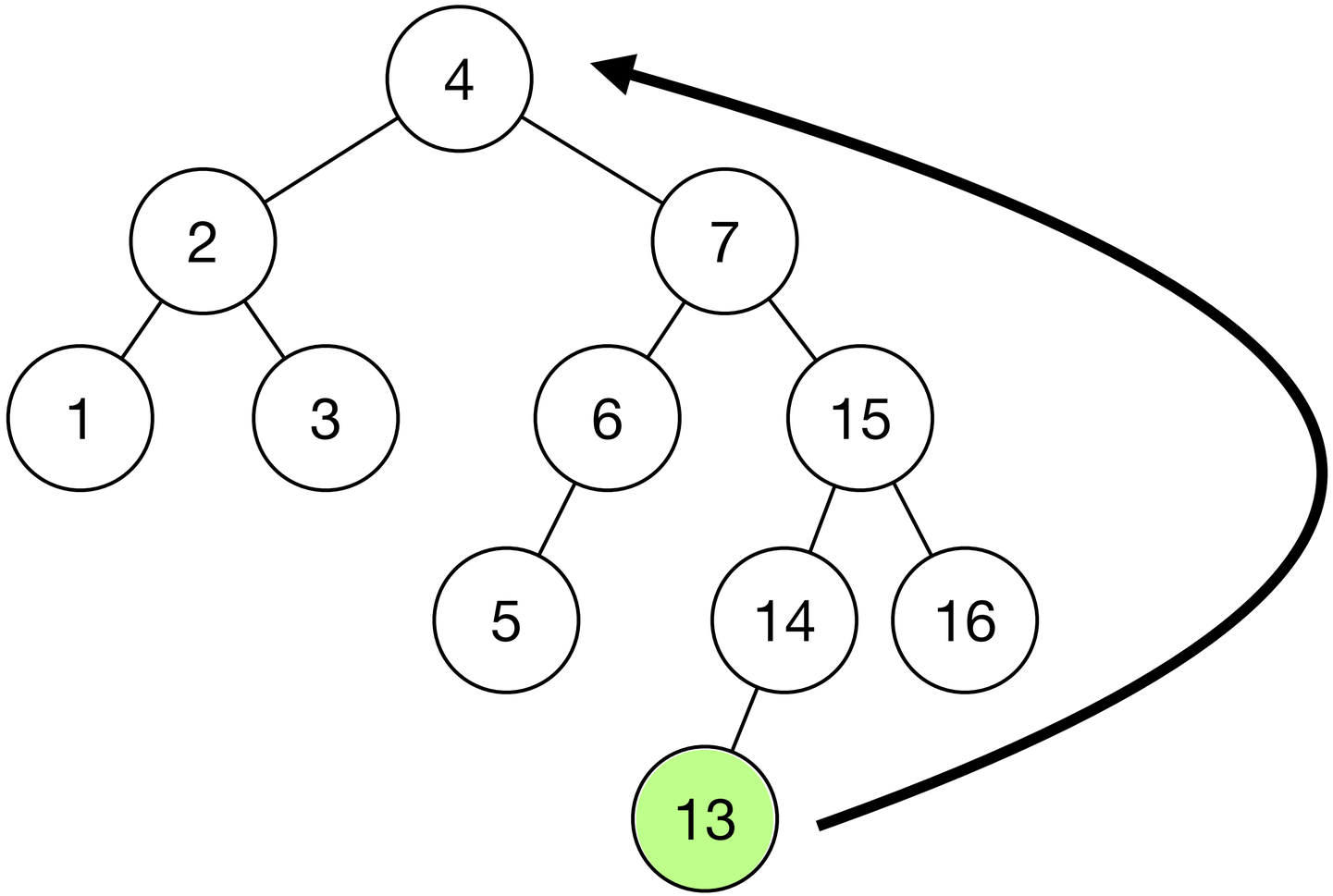
Example



Example



Example



Splay Trees

- Splay tree will move the currently accessed or inserted node to the root of the tree on each access or insertion.
- Upon deletion, splay tree will first splay the target node to the root, then delete it, then it will join the two separated subtrees.

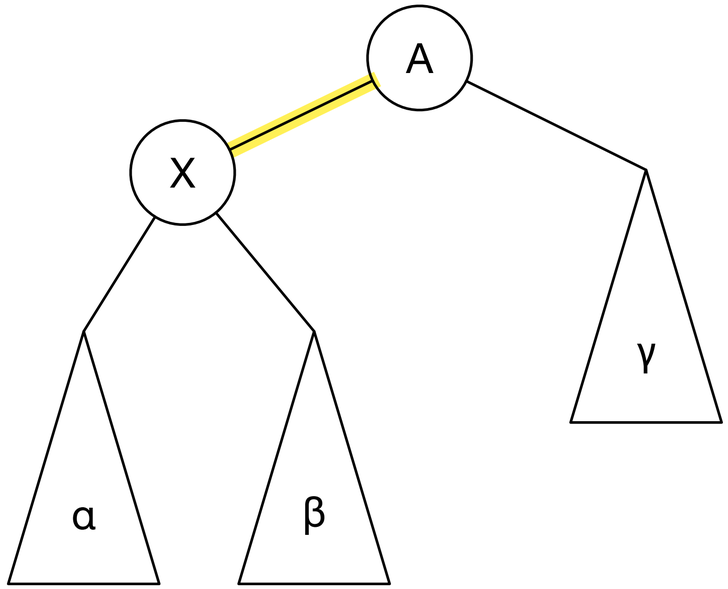
Tree Rotation

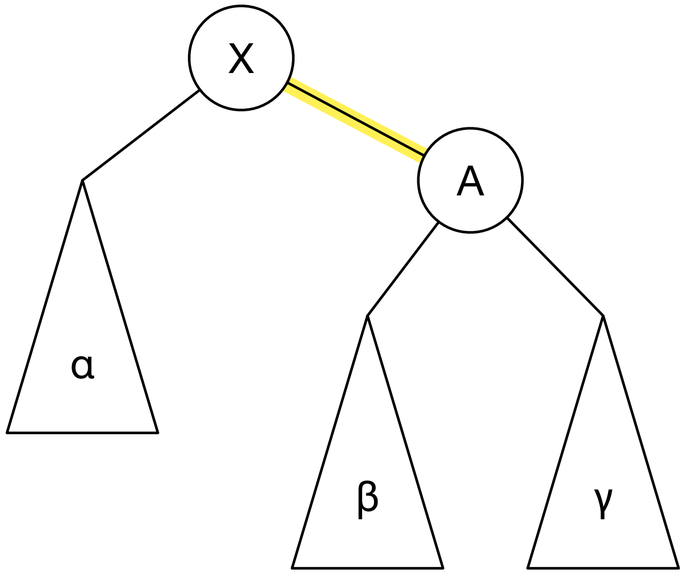
Zig, Zig-Zig, and Zig-Zag

Case	Description	Action / rotation
Zig	The parent of x is the root	Rotate the edge joining x and the root
Zig-Zig	The parent of x is not the root, and the parent of x and x are both left (or right children)	Rotate the edge joining parent and grandparent of x; then rotate the edge joining x and its parent
Zig-Zag	The parent of x is not the root, and x is a left child and its parent is a right child (or vice-versa)	Rotate the edge joining x and its parent; then *again* rotate the edge joining x and its *new* parent ("double rotation")

Case: Zig

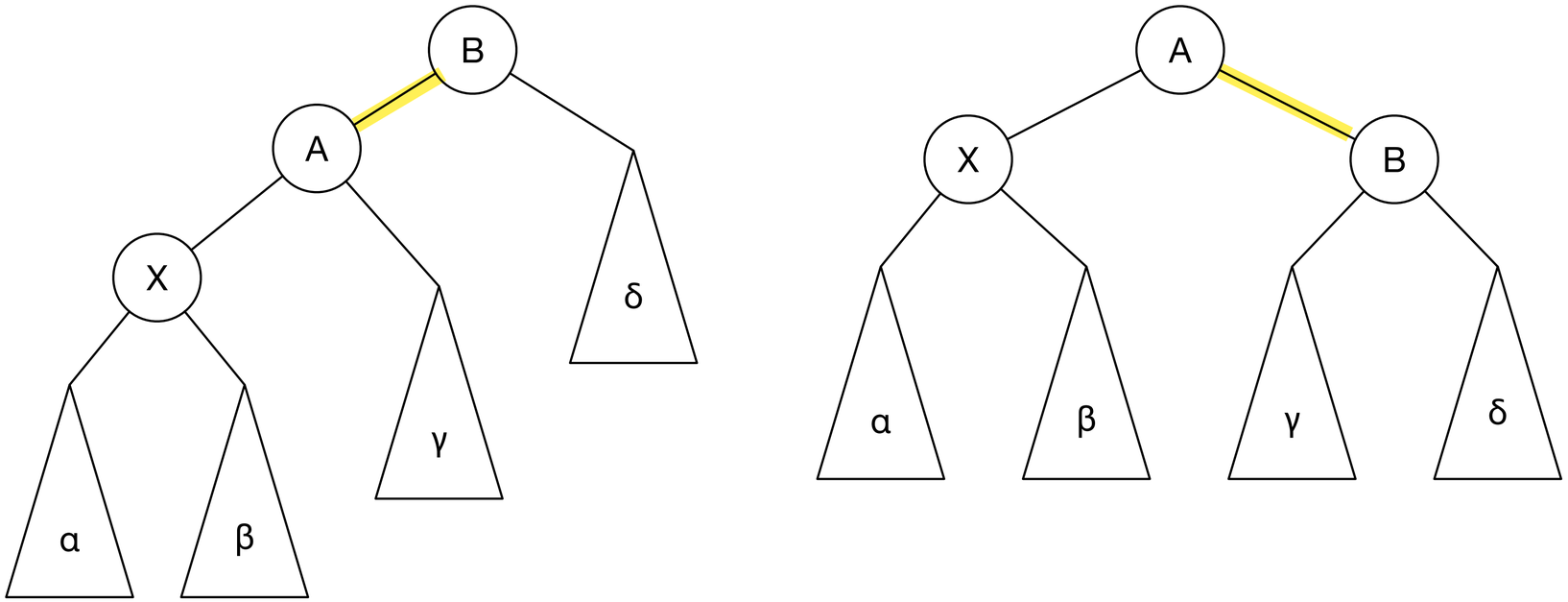
Target node is child of root





Case: Zig-Zig

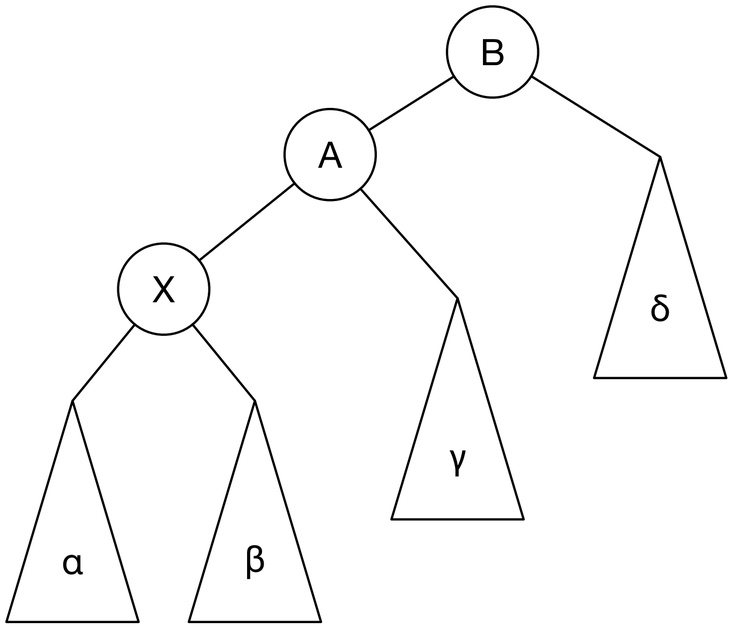
Target node is not child of root; two edges same direction

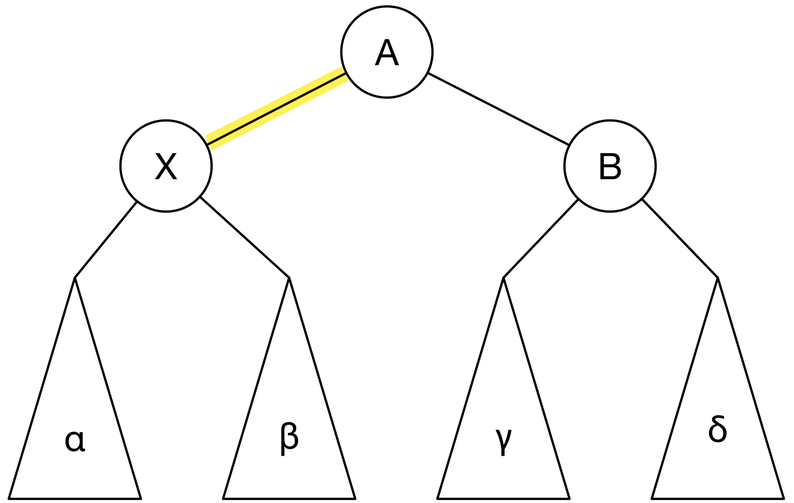


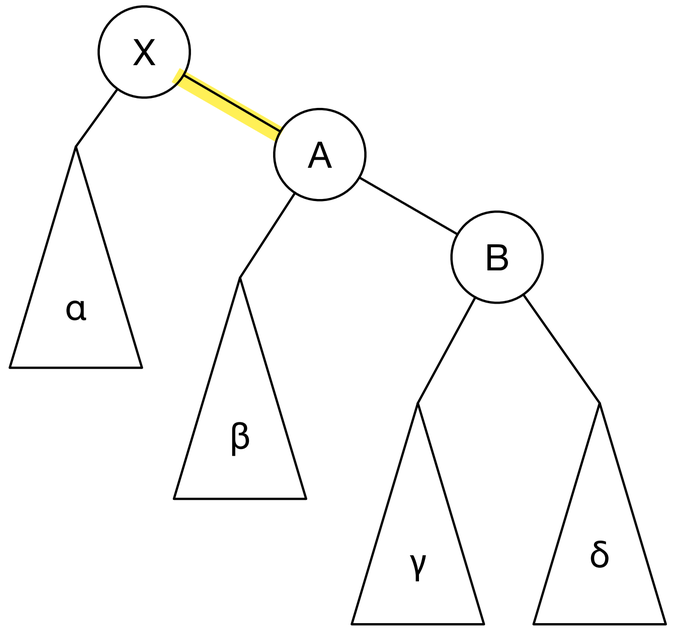
First, rotate the edge between X's parent and X's grandparent

Case: Zig-Zig

Target node is not child of root; two edges same direction



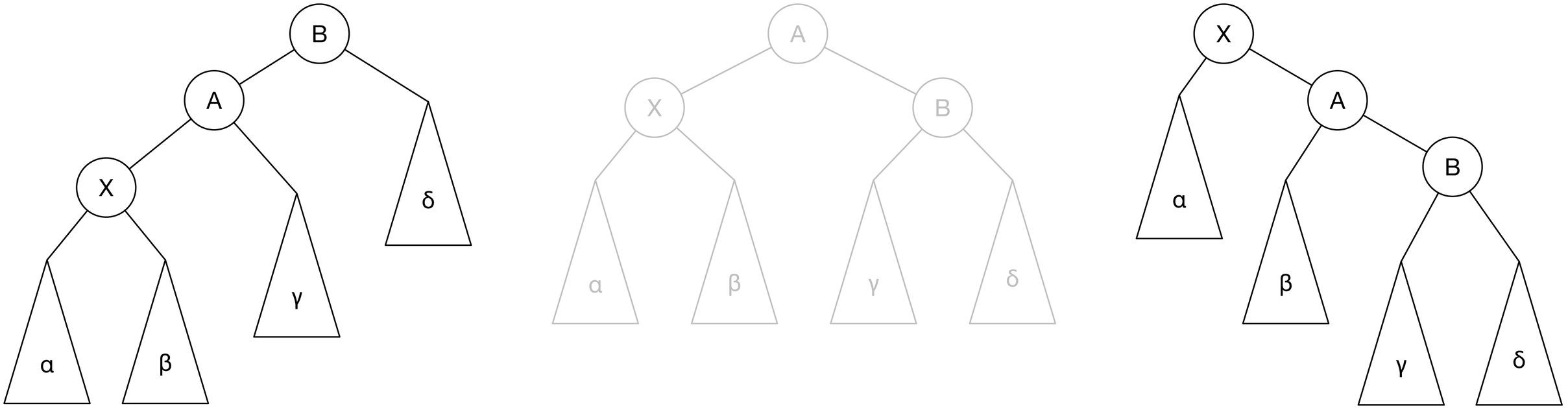




Then rotate the edge connecting X and its parent

Case: Zig-Zig

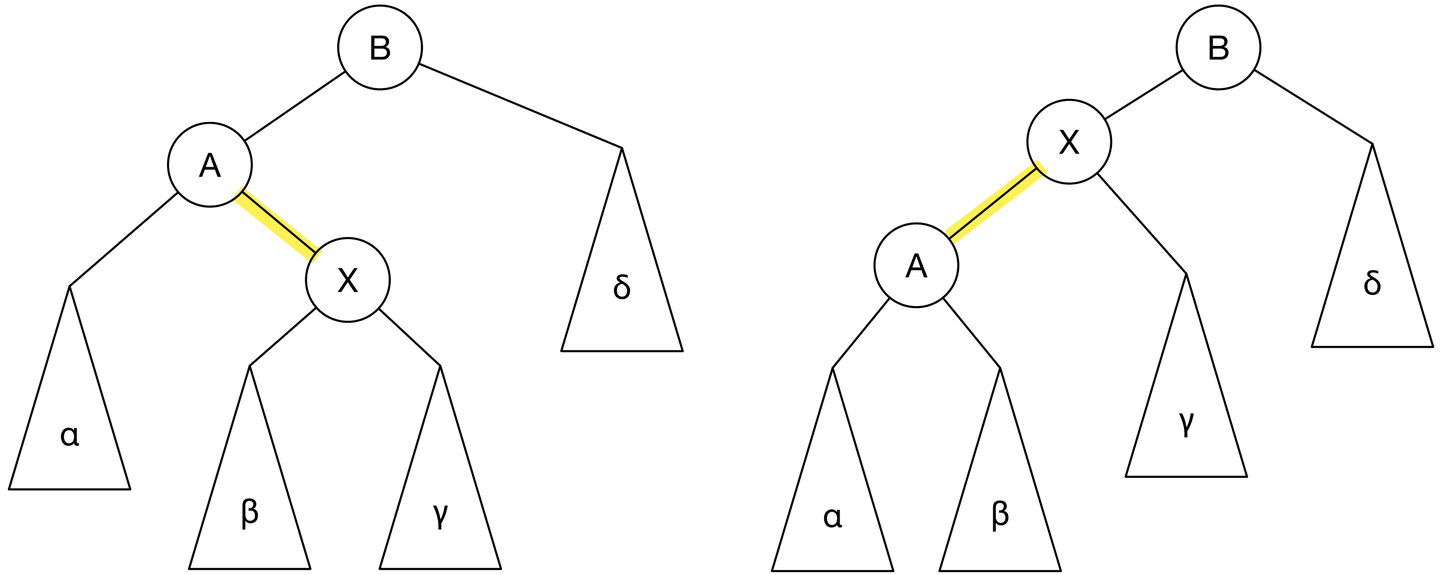
Target node is not child of root; two edges same direction



Note: Some textbooks and instructors treat this combination as a single rotation. That's OK as long as you don't get confused.

Case: Zig-Zag

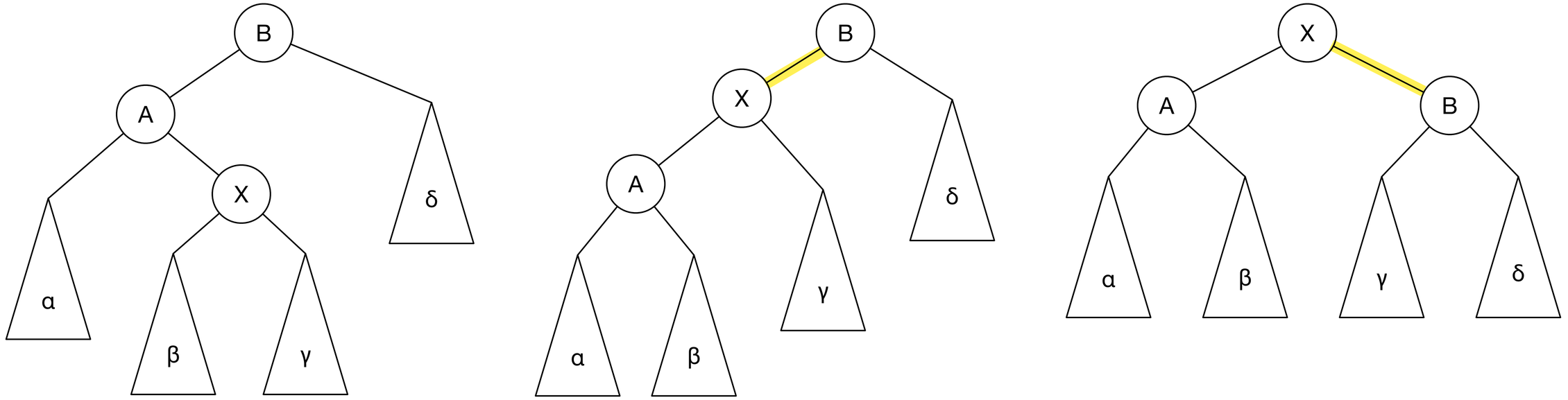
Target node is not child of root; two edges, different directions



First, rotate the edge between X and X's parent.
Notice that this will leave X with a new parent.

Case: Zig-Zag

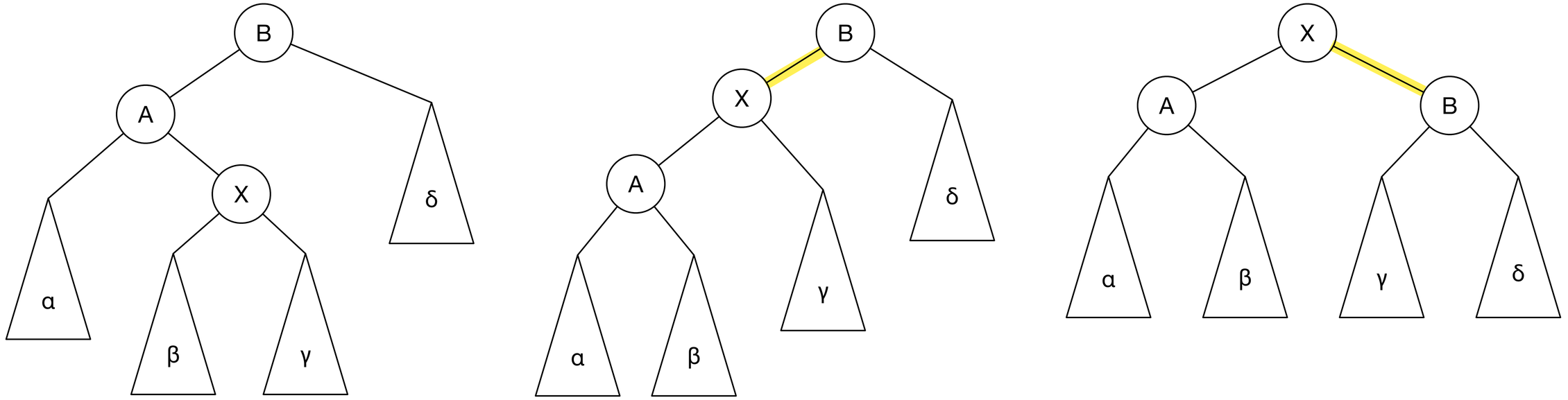
Target node is not child of root; two edges, different directions



Then rotate the edge connecting X and its new parent

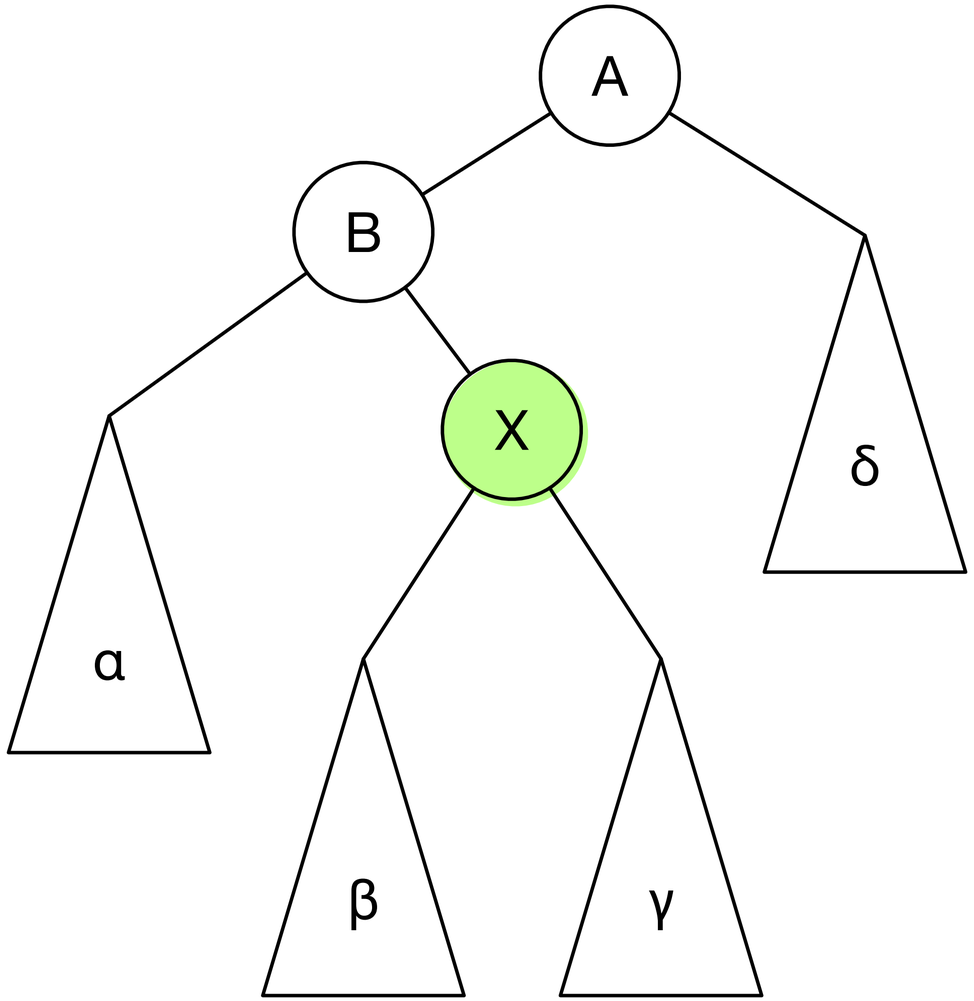
Case: Zig-Zag

Target node is not child of root; two edges, different directions

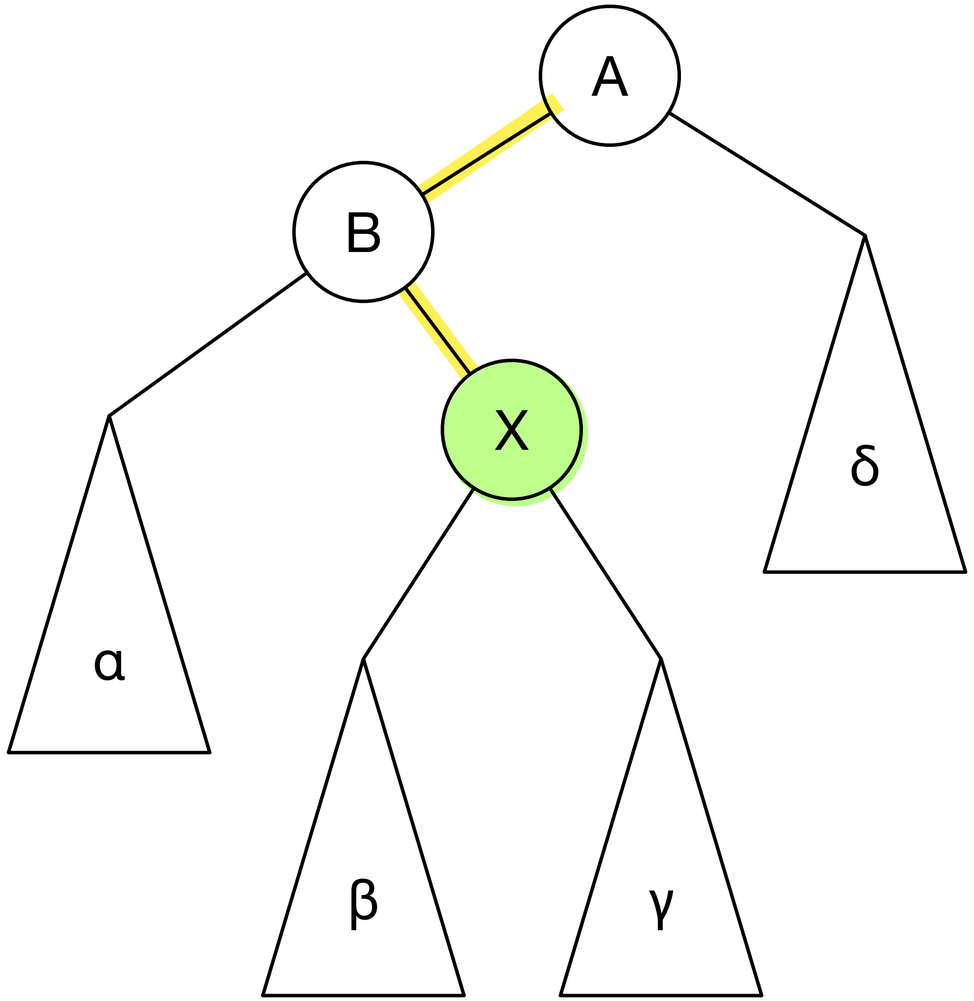


It is because we rotate the edge between X and its parent *twice* that we call this a "double rotation."

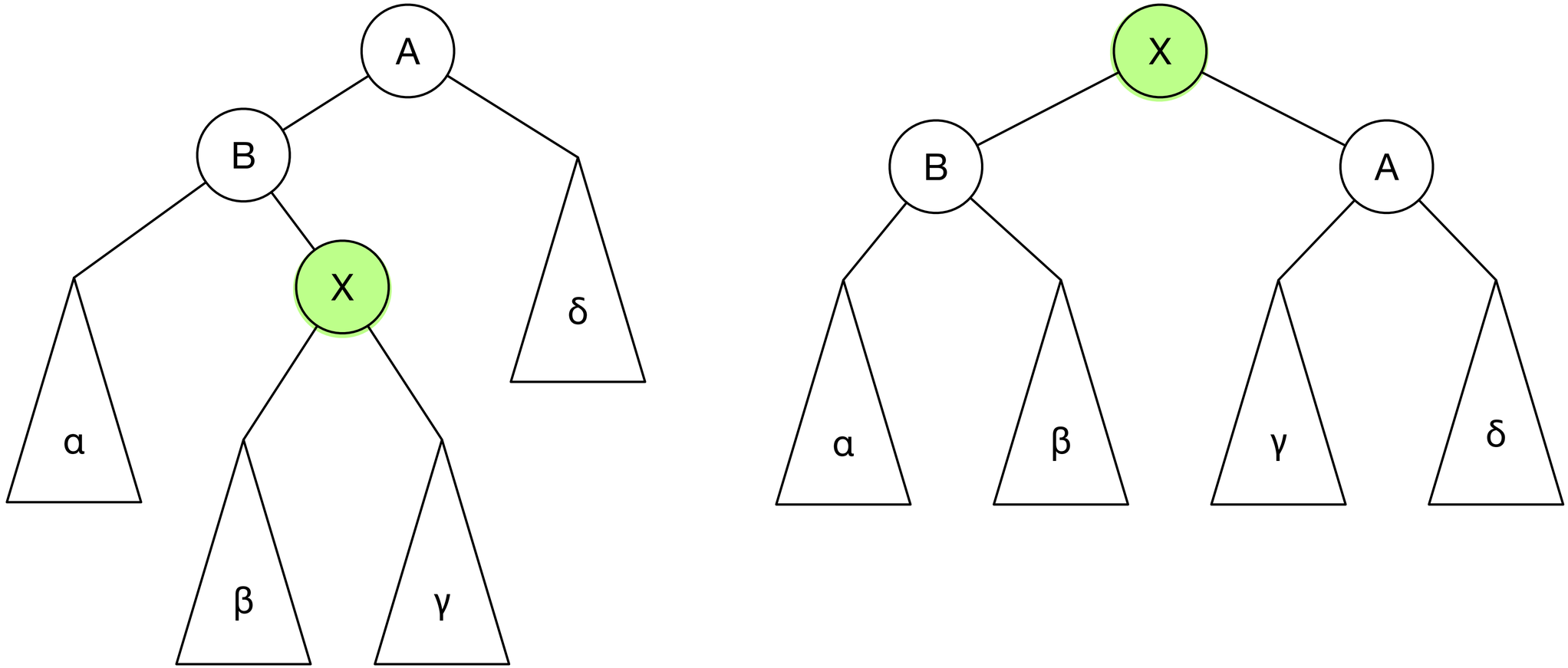
Example



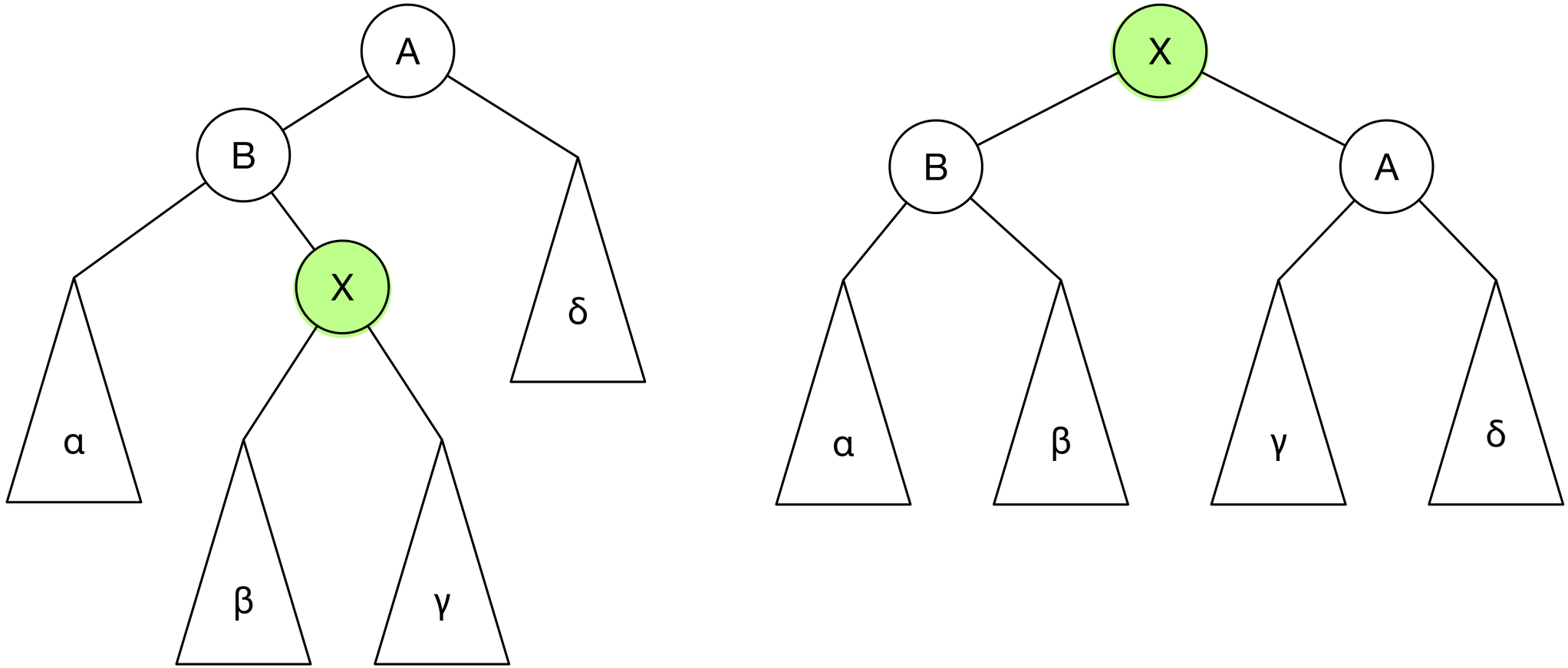
Example



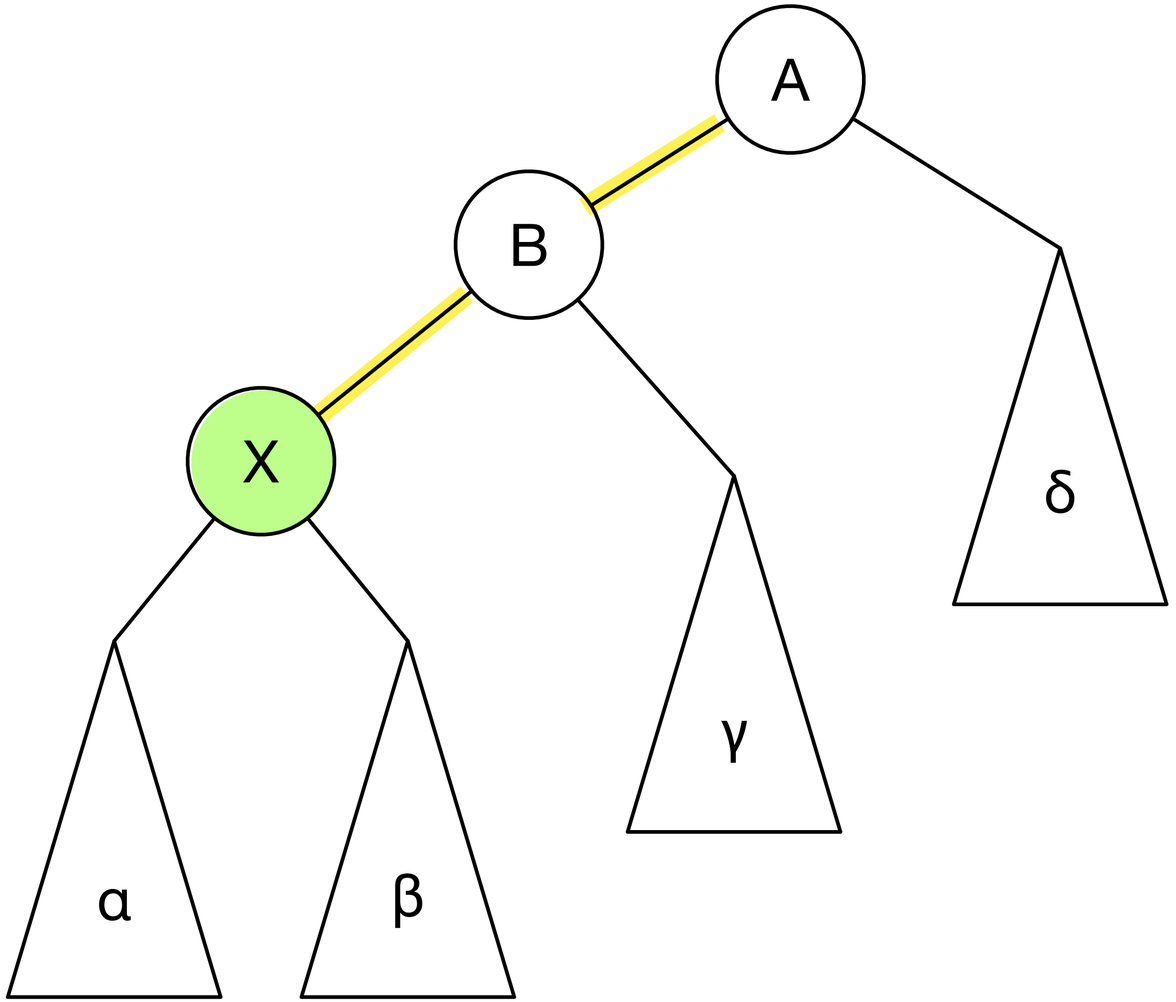
Example



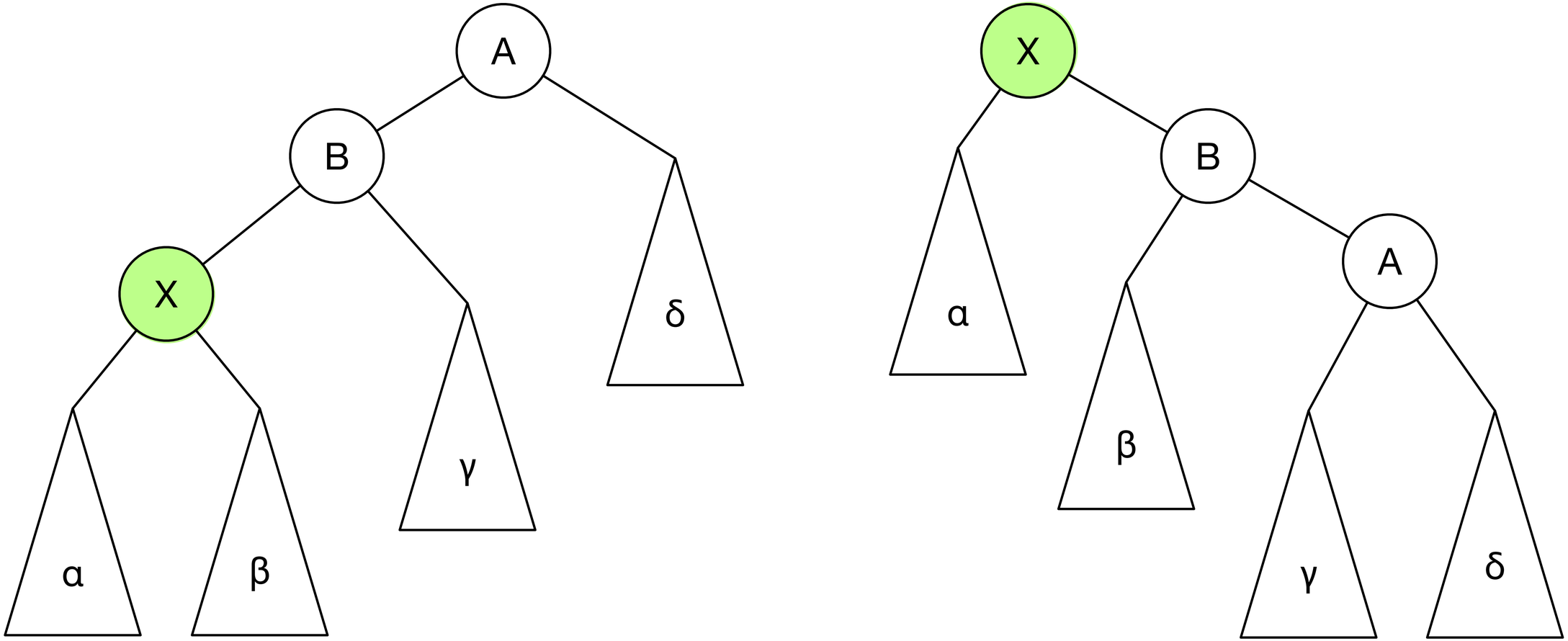
Example



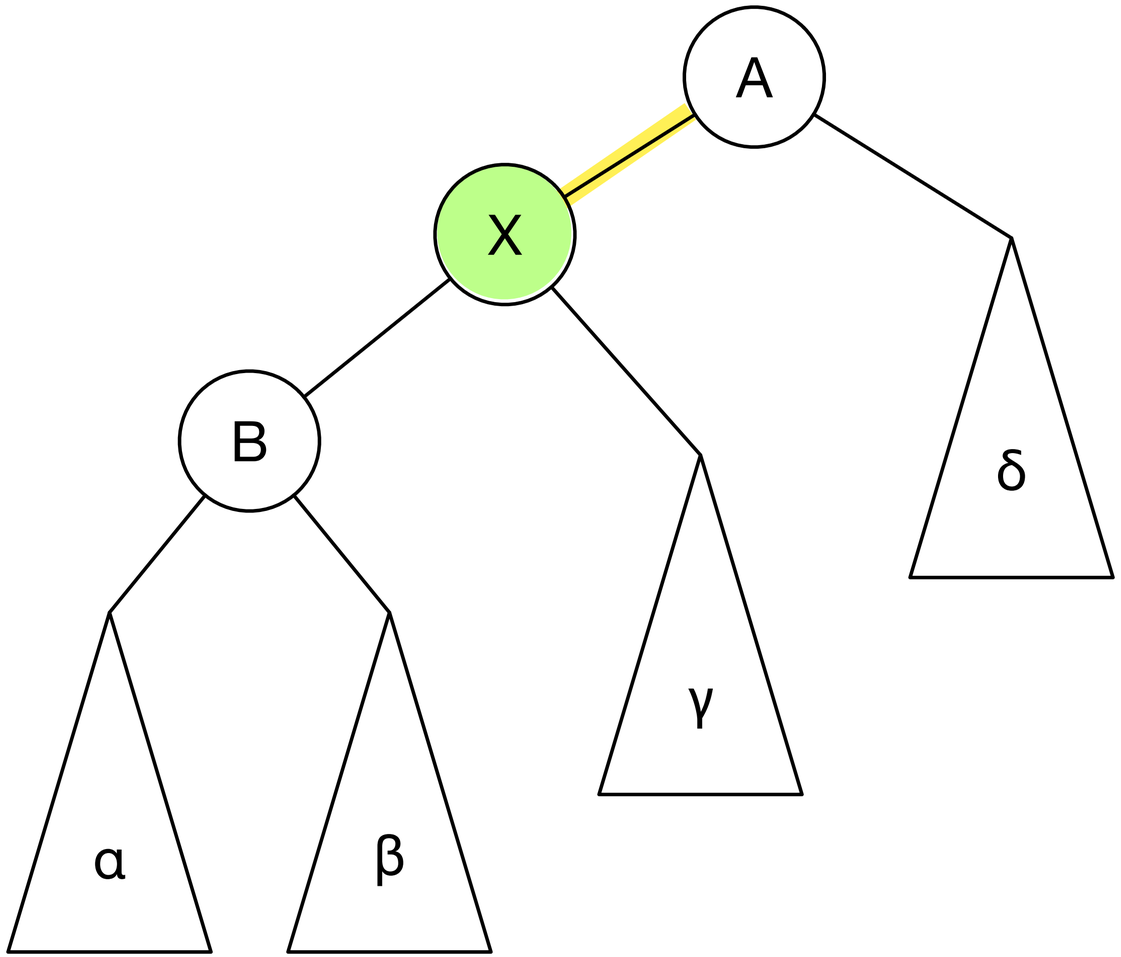
Example



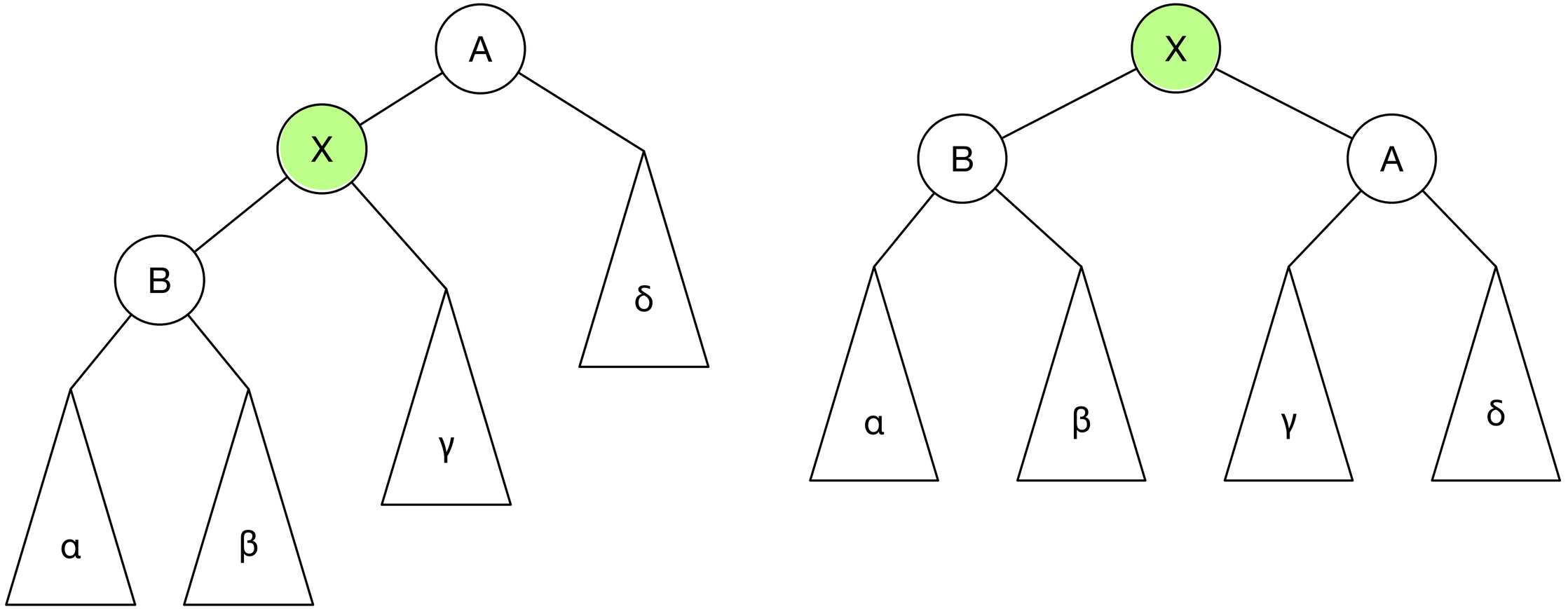
Example



Example



Example

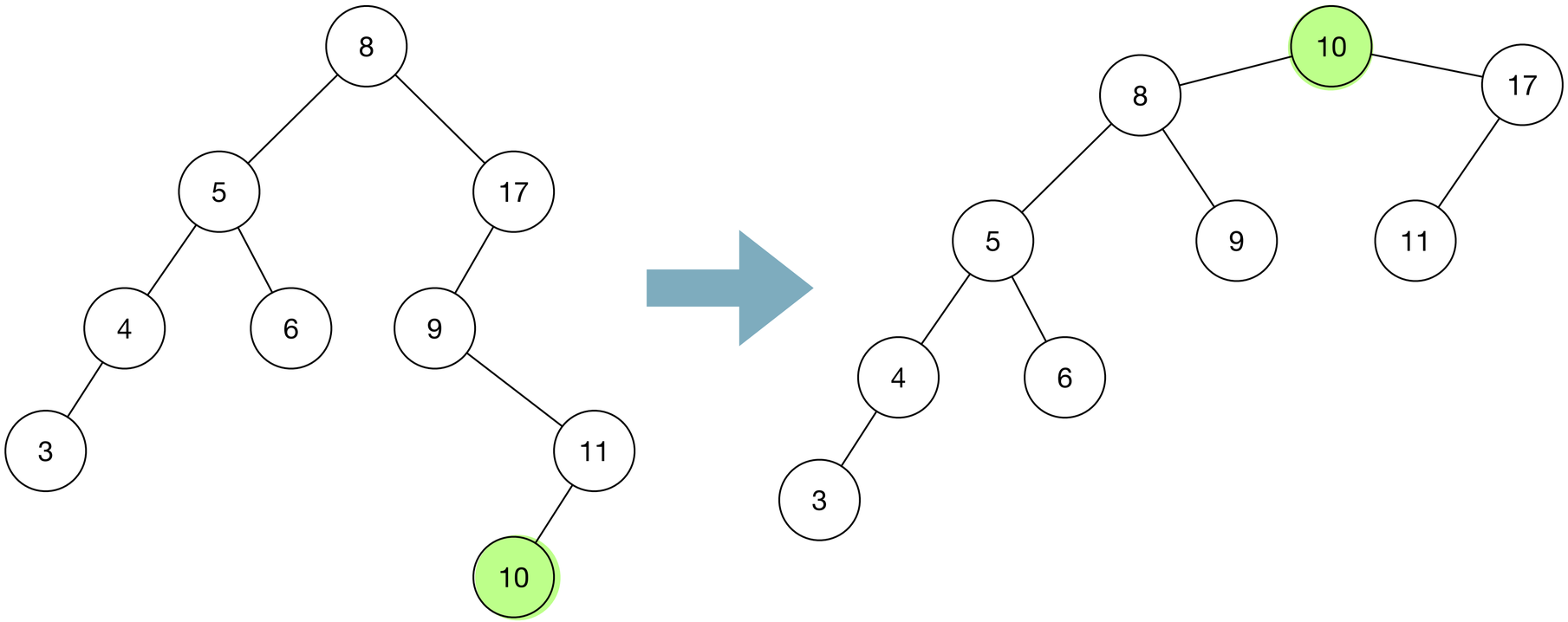


Splay Trees

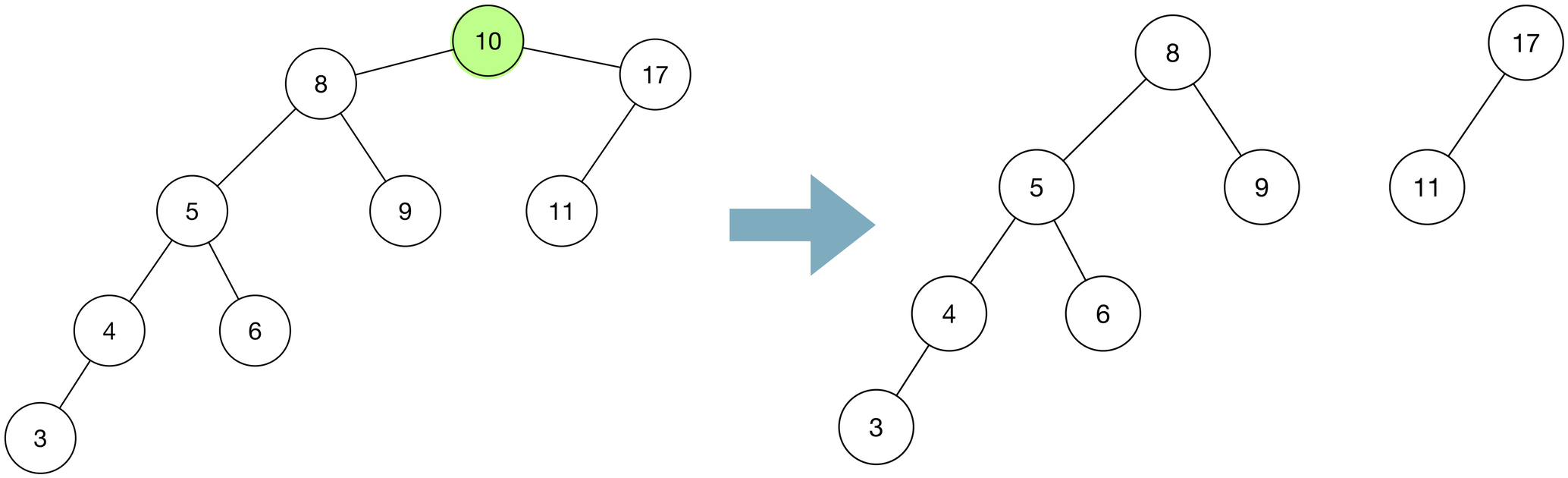
Deletion of nodes

- First, splay the target node to the root, and delete the target node.
- Three cases.
  - There are no remaining nodes.
  - There is only one subtree remaining. Make the root of this subtree the root of the tree.
  - There are two subtrees to be joined. Find the largest element in the left subtree and splay it to the root of the left subtree. Then join the right subtree as a child of the root in the left subtree.

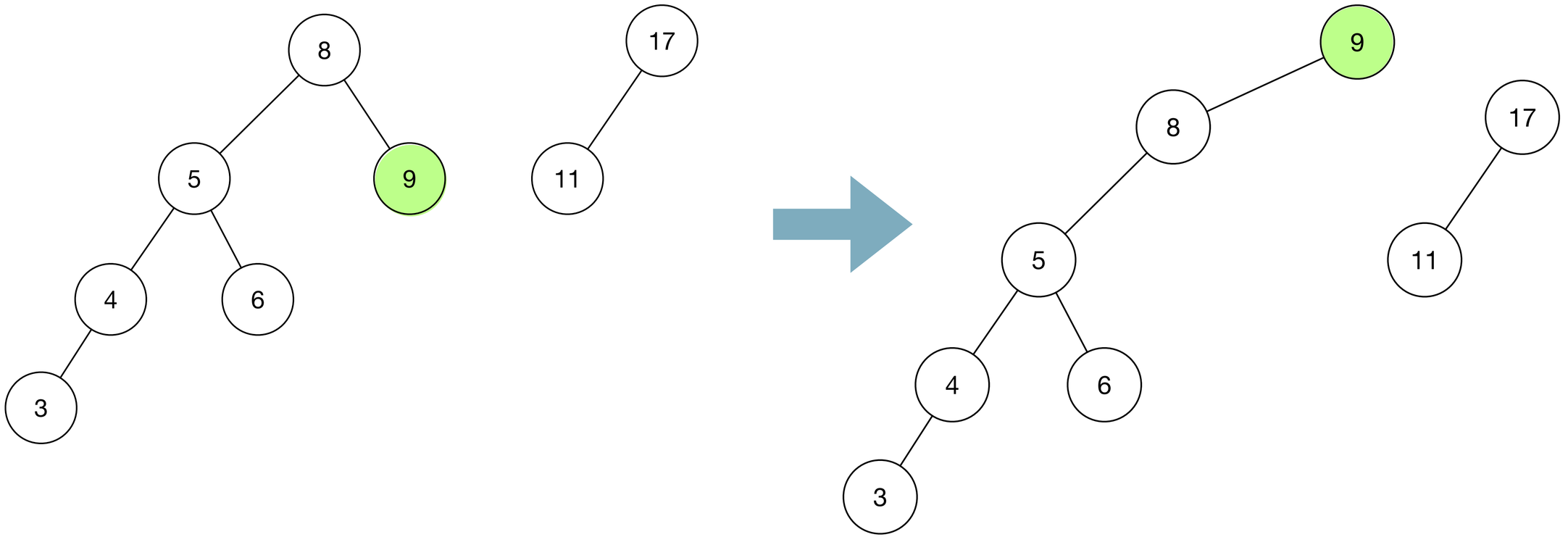
Delete 10



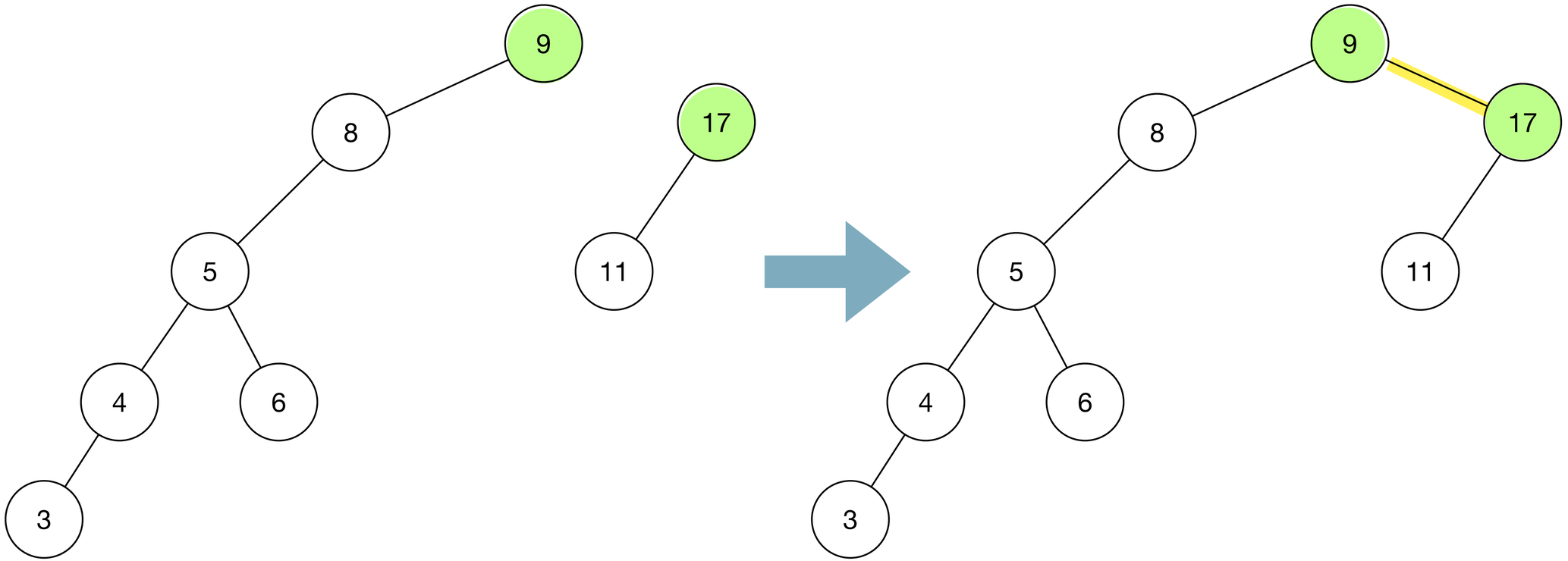
Delete 10



Delete 10



Delete 10



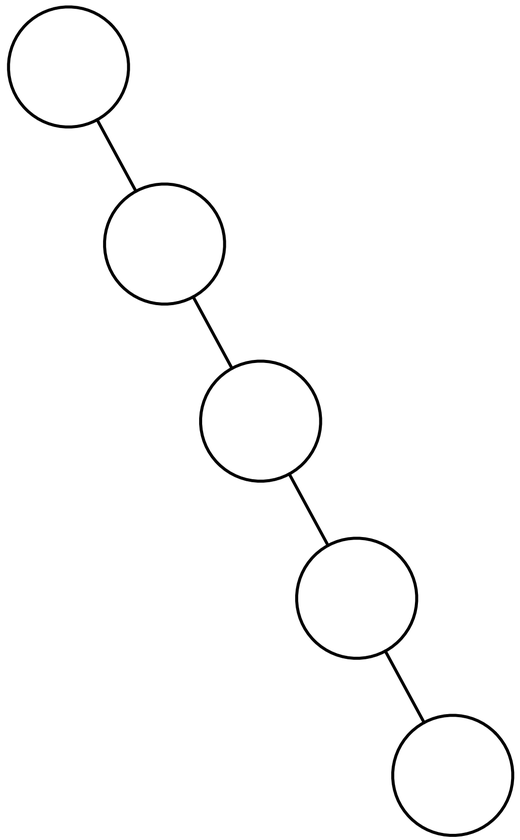
What's the downside?

- **"Cache misses."** After we move a node to the root, there's no guarantee that it will be referenced again in the near term. So in a sense, we've wasted the effort of moving it to the root. This happens with caching. I assume you've heard of cache hits and cache misses. Sometimes you guess wrong. It happens.

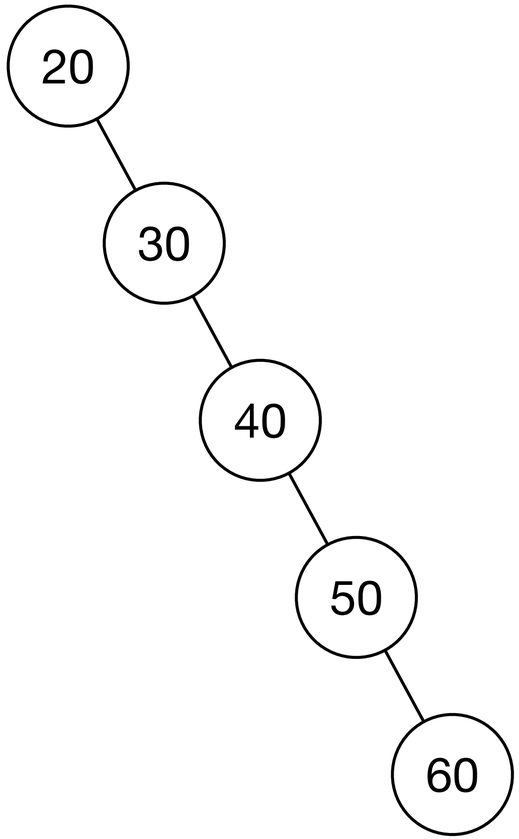
- **Unbalanced trees.** While in general, splay trees tend to be reasonably well balanced, there is nothing that prevents them from becoming unbalanced, and under certain sequences of additions, deletions, and accesses, a splay tree can become very imbalanced.

- **Extra work.** We splay on each operation, so there's computational overhead in maintaining a splay tree.

Unbalanced Trees



Unbalanced Trees



Splay Trees: Typical Behavior

- Barring pathological sequences, splay trees perform well, expecially if the principle of locality applies and frequent accesses are made to values in the root position. In this sense, a splay tree behaves like a cache.

- Here we introduce the concept of **amortization**. Notice that unless we access the root, the tree is modified with each operation. If we amortize this cost -- that is, spreading the cost across a large number of accesses -- performance approaches $O(\log N)$. Though a single access could be $O(N)$ in the worst case for *any single operation*, overall performance is much closer to $O(\log N)$.