ECS 32B - Trees: Advanced

Aaron Kaloti

UC Davis - Summer Session #2 2020

Overview

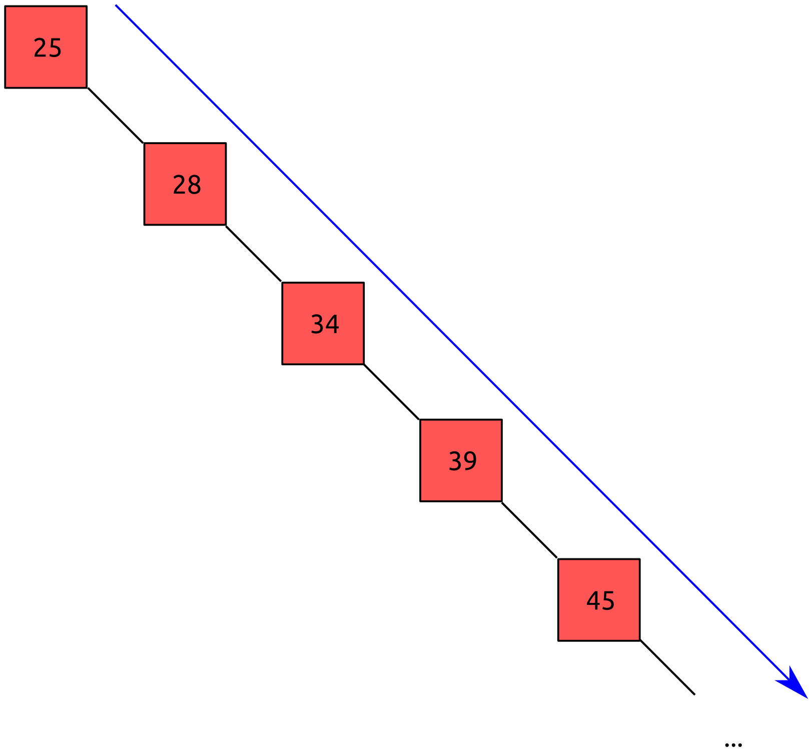
- Self-balancing binary search trees.
  - Motivation.
  - AVL trees.
  - Splay trees.

Self-Balancing Binary Search Tree

- After certain operations (e.g. insertion, deletion), reorients itself to achive some goal (usu.) faster future operations.

Motivation

- Binary search tree has linear worst-case time complexity for find/insert/delete.



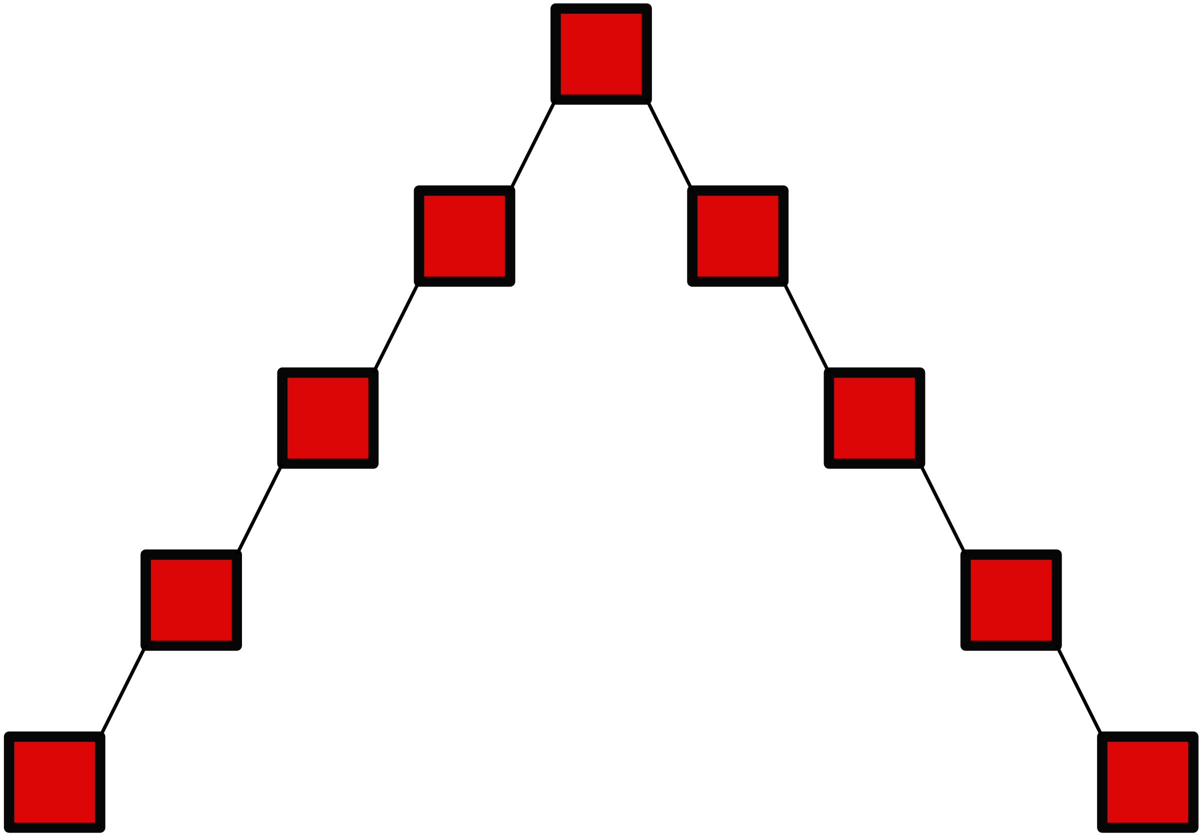
- Self-balancing binary search trees can usually achieve logarithmic time, i.e. $\Theta(\lg n)$.

Balance Condition[1]

- What balance condition can we force a binary search tree (BST) to maintain?

Example (of a Bad Idea)

- Require left/right subtrees of root to have same height.
- Doesn't force BST to be shallow.

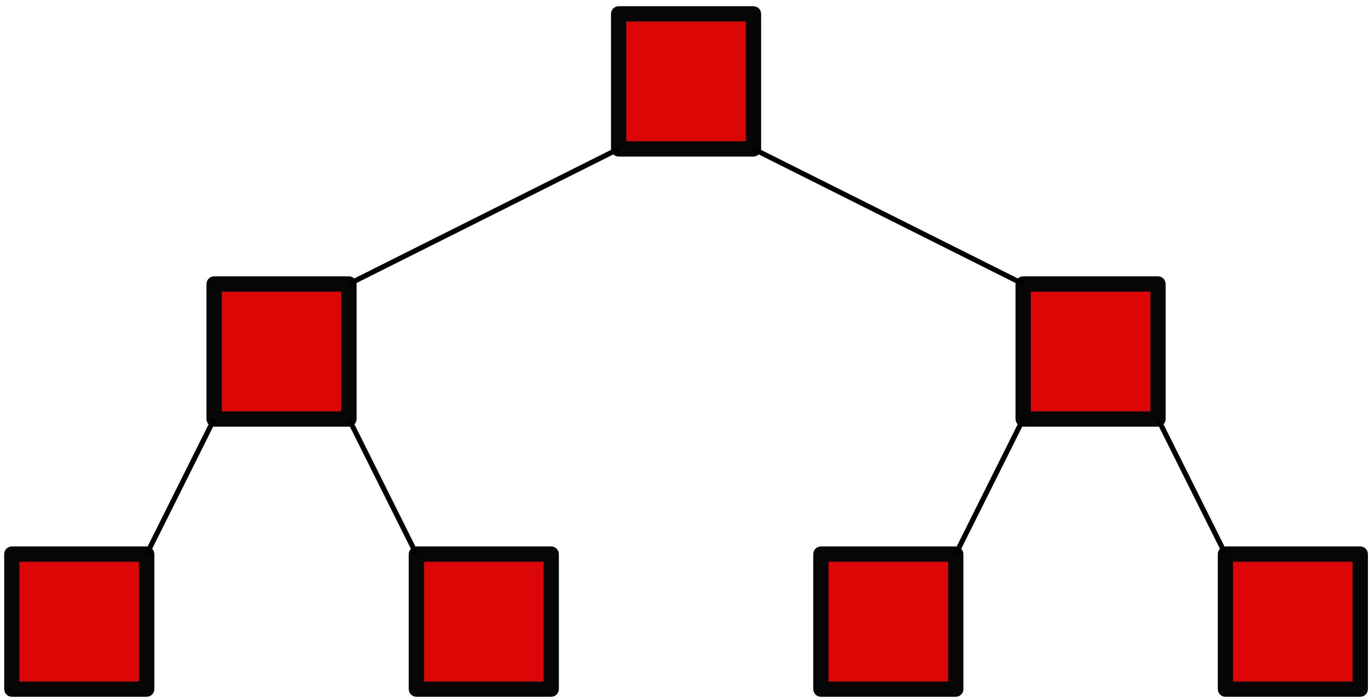


1. This slide and the next one were not in this slide deck during the 08/27 lecture but were inserted before the balance factor slides shown during that lecture.

Balance Condition

Example (of Another Bad Idea)

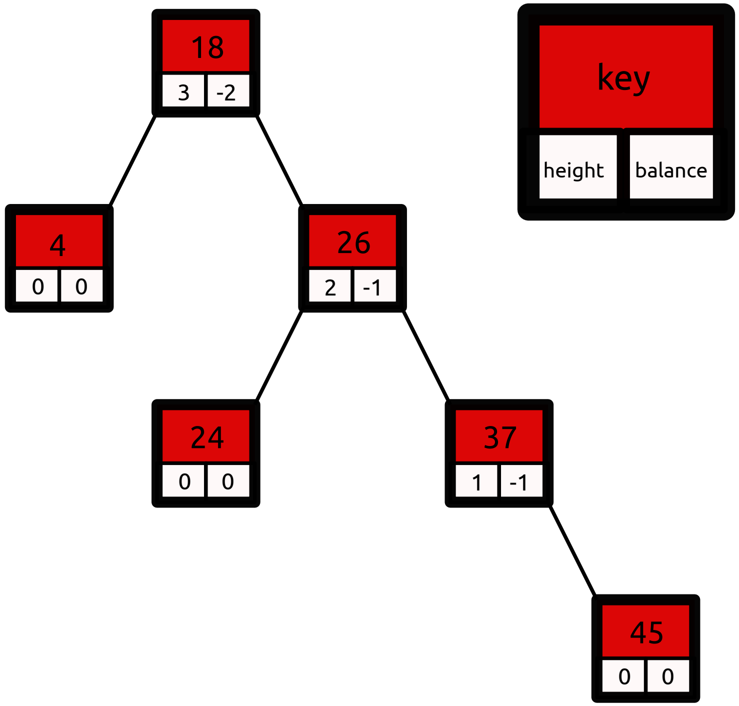
- Force every node to have left/right subtrees of same height.
- Too rigid: only BSTs with $2^k - 1$ nodes could obey, where $k \in \mathbf{N}_1$ (positive natural numbers).



Balance Factor

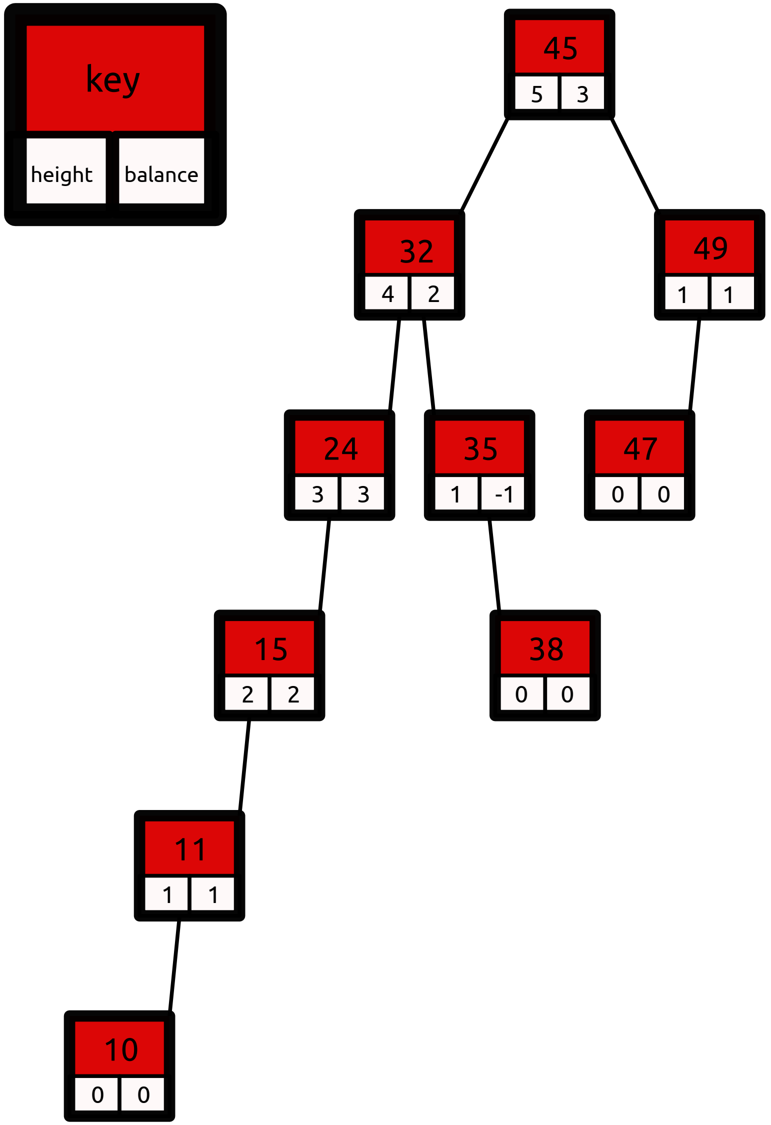
- Can give each node a balance factor ⇒ $\mathit{height}(\text{leftSubTree}) - \mathit{height}(\text{rightSubTree})$.
  - Subtree is *left-heavy* if root's balance factor is positive; *right-heavy* if negative.
  - Height of empty/nonexistent subtree is -1.
- $\mathit{height}(\mathit{node}) = 1 + \mathit{max}(\mathit{height}(\text{leftSubTree}), \mathit{height}(\text{rightSubTree}))$.

Example: Right-Heavy Tree



Balance Factor

Example: Left-Heavy Tree

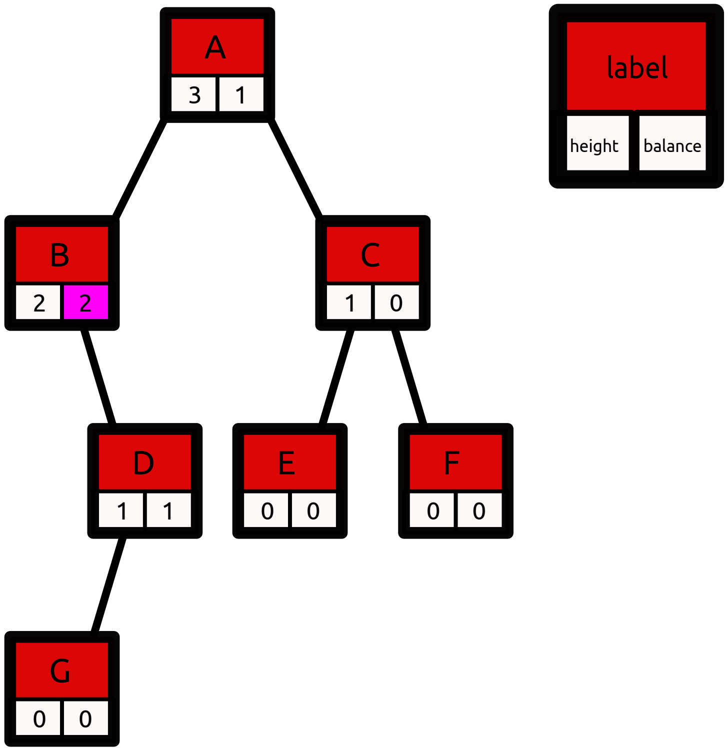


AVL Tree

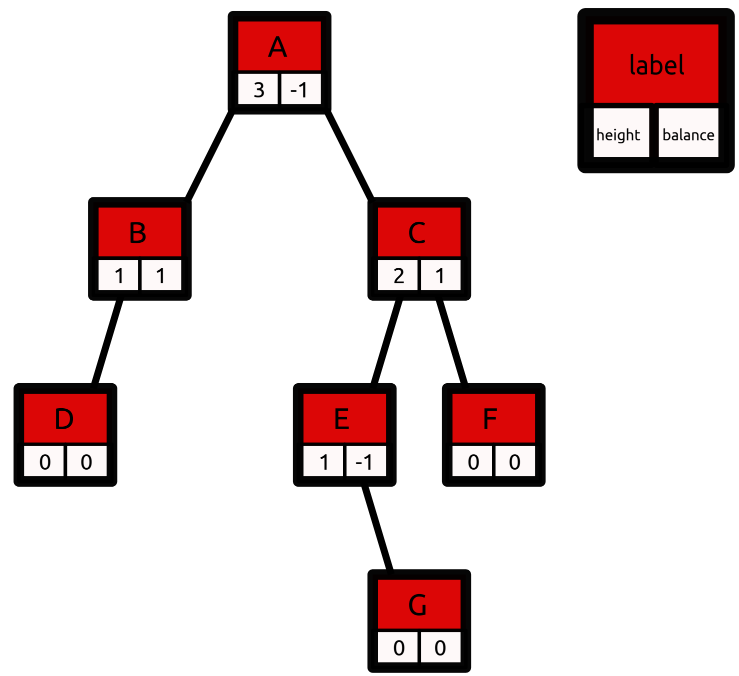
Description

- An AVL tree is a binary search tree in which each node has a balance factor of -1, 0, or 1.
- Through some math, we[1] can prove that at any time, the height of an AVL tree is at most around $1.44 \lg n$, where n is the number of nodes. Thus, searching takes $\Theta(\lg n)$ in the worst case.

Example: *Not* an AVL Tree



Example: *Is* an AVL Tree



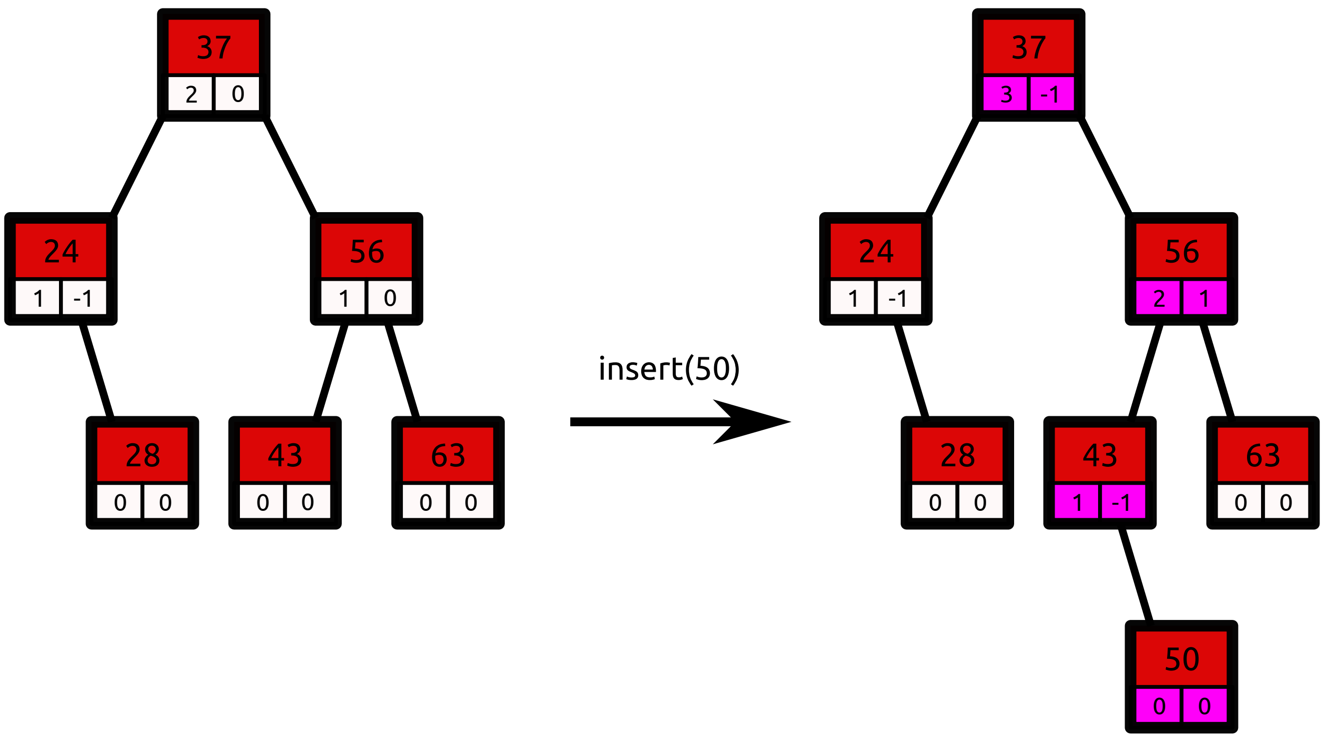
1. By "we", I mean section 7.16 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

AVL Tree

Insert: Simple Balance Update (1 of 2)

- Start with same procedure as with BST. Inserted node is a leaf, i.e. height and balance factor of 0.
- Update parent:
  - New node is left child ⇒ increment parent's balance factor.
  - Else ⇒ decrement.
- Update parent's parent, etc. Recursively apply until reach root.
- Adjust heights too.

Example

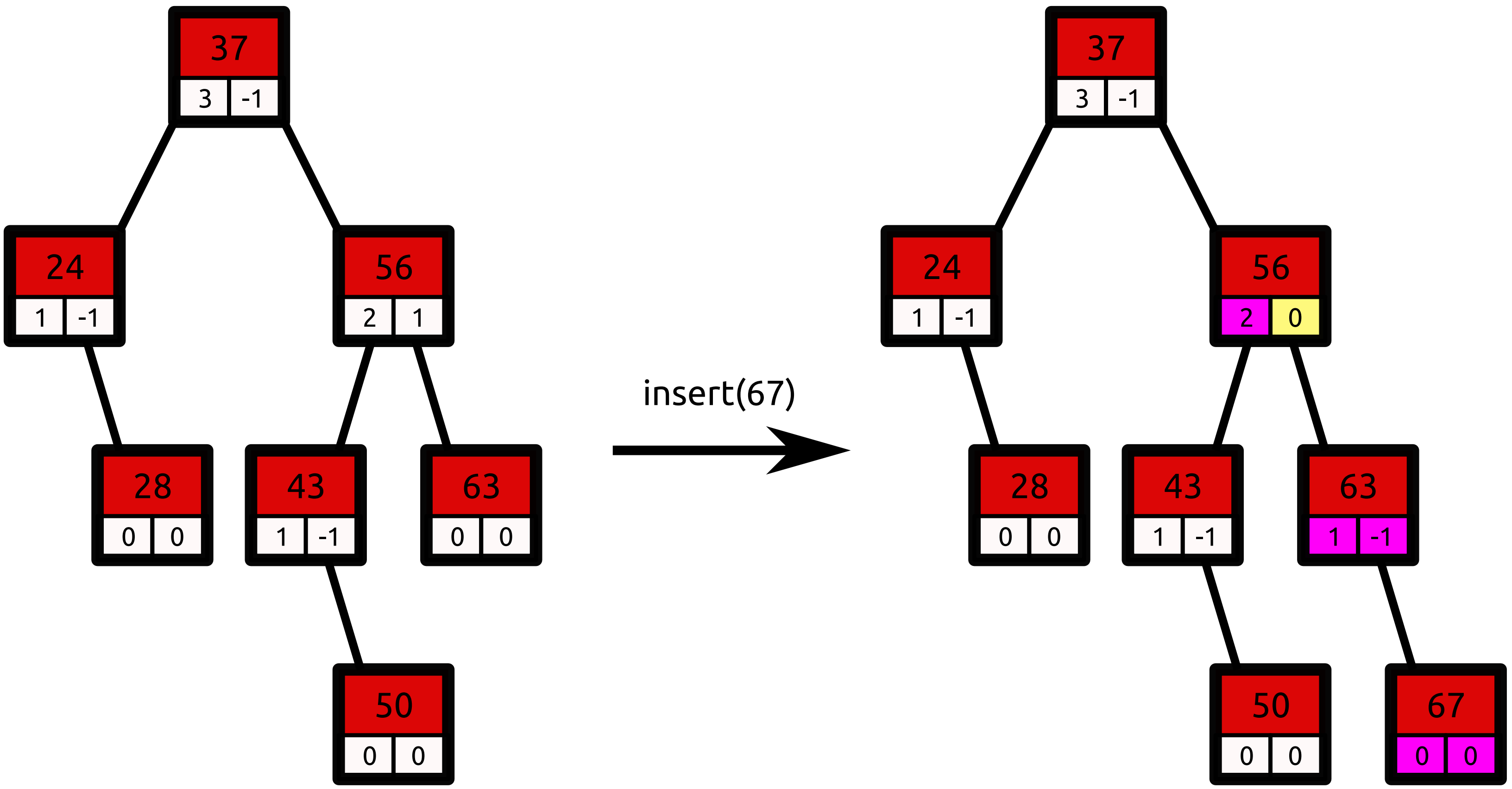


AVL Tree

Insert: Simple Balance Update (2 of 2)

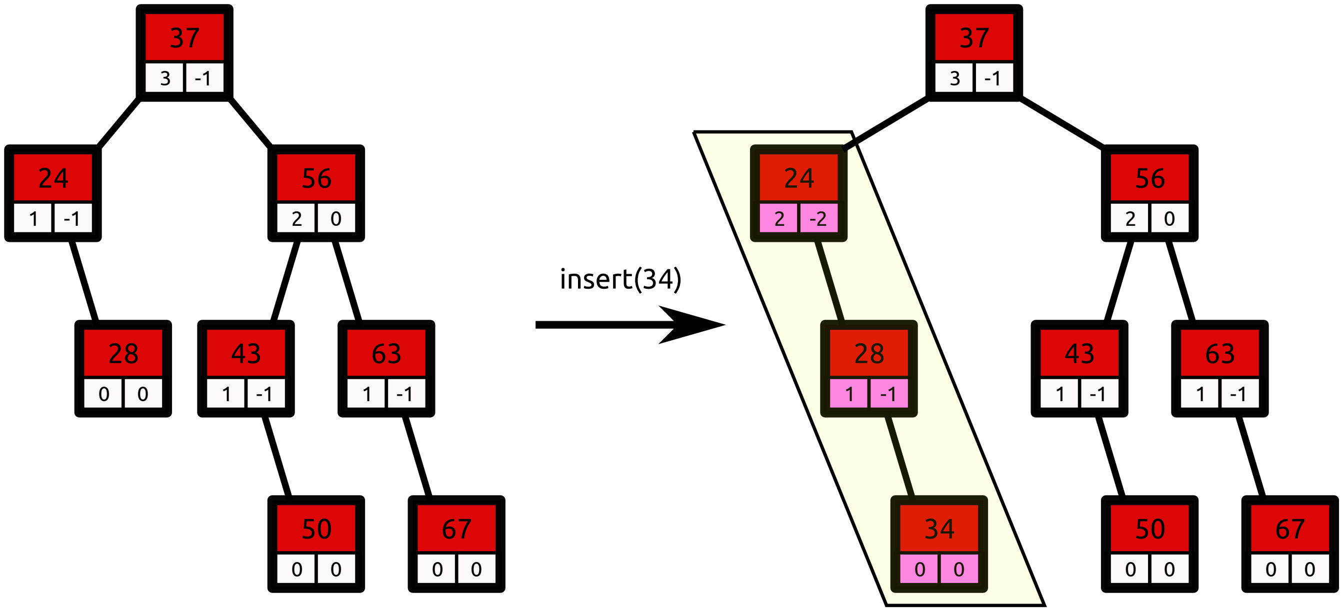
- If adjust parent's balance factor to zero, *don't adjust ancestors*.

Example



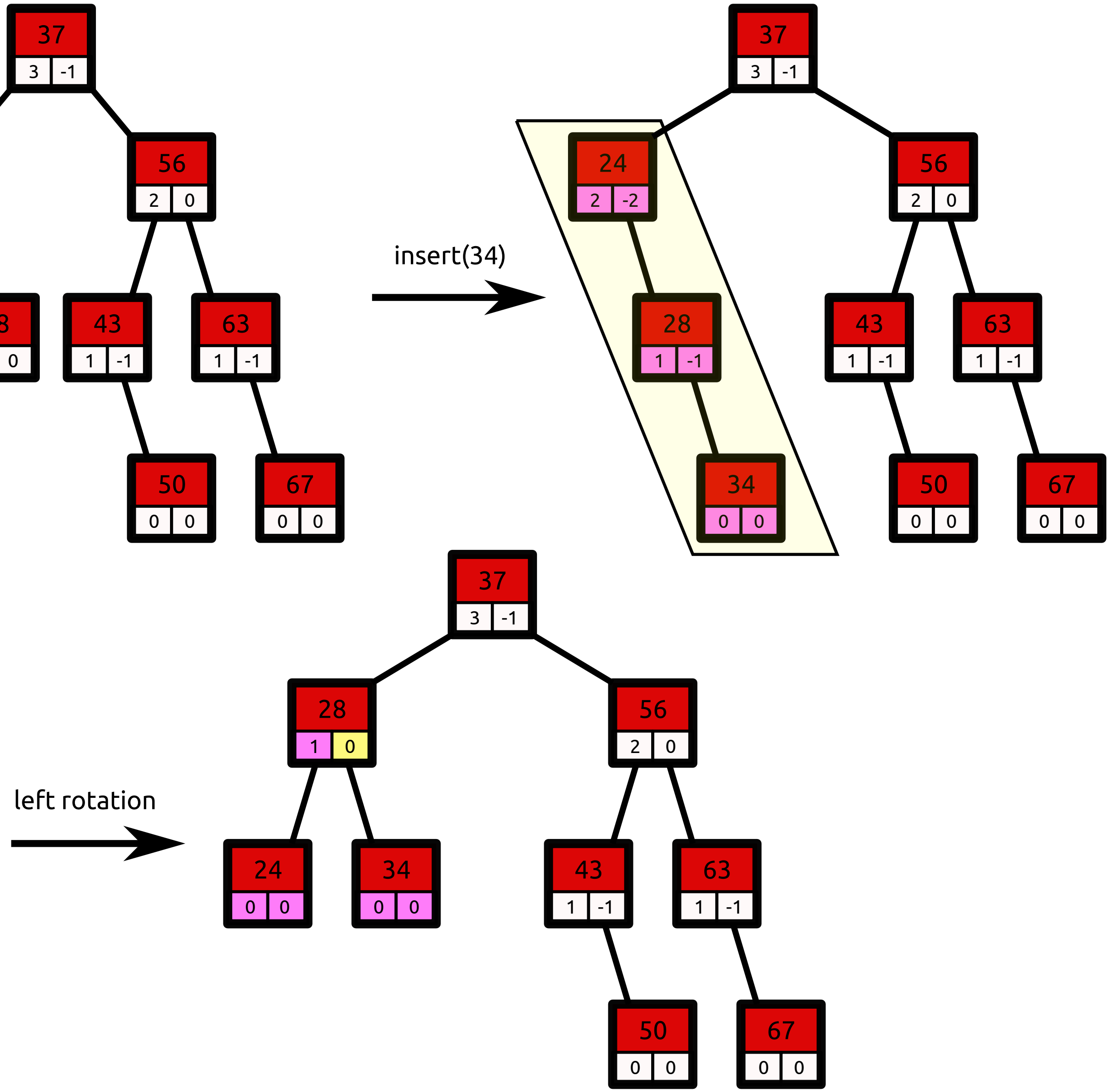
AVL Tree

Insert: When Rebalancing is Required



AVL Tree

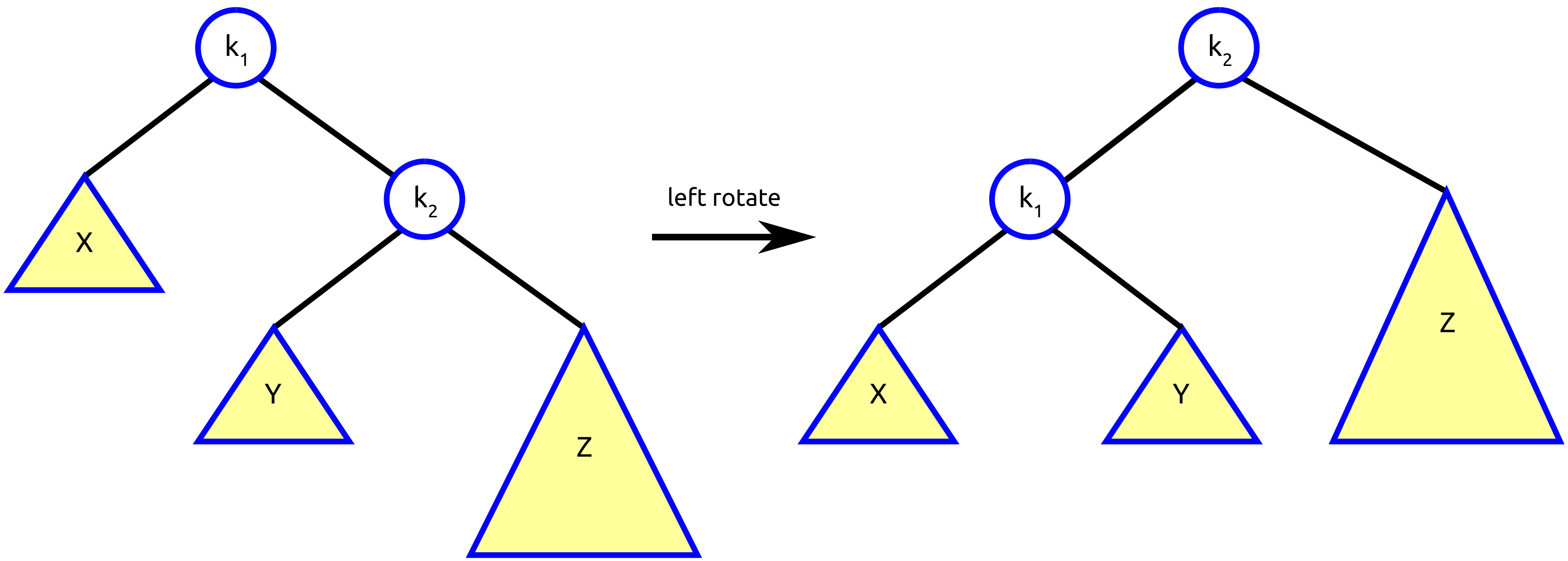
Insert: When Rebalancing is Required



AVL Tree

Left Rotation

- Subtrees of k_1 differ by more than 1 in height.
  - e.g. just inserted into Z.



- k_2 will have balance 0 after.
- Previous example is a left rotation with empty X and Y.
- Takes constant time (adjust some references).

AVL Tree

- Never need to do more than one rebalancing[1] per insertion.
- k_1 is node with bad balance factor.

Four Reasons a Rebalance is Needed

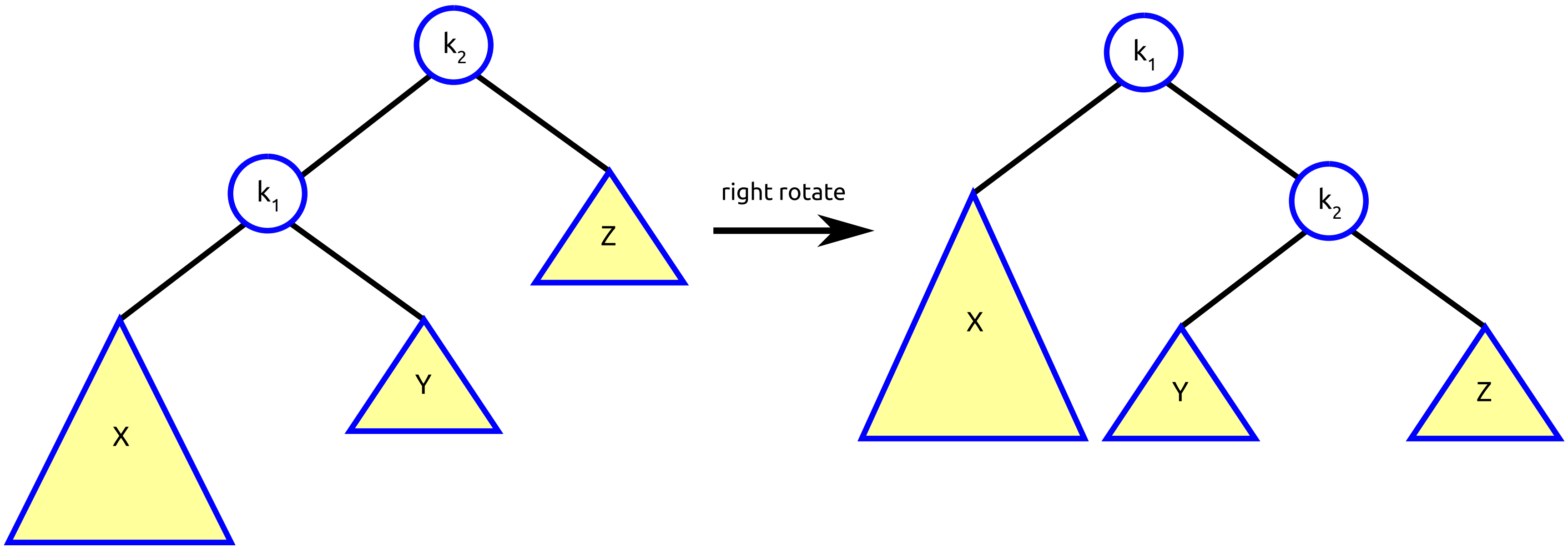
1. An insertion into the left subtree of the left child of k_1.
2. An insertion into the right subtree of the left child of k_1.
3. An insertion into the left subtree of the right child of k_1.
4. An insertion into the right subtree of the right child of k_1.
    - Resolved by left rotation in previous slide.

1. I say "rebalancing" instead of rotation because, as you'll see soon, some insertions (i.e. ones that lead to cases #2 or #3 above) necessitate *double rotations*.

AVL Tree

Right Rotation

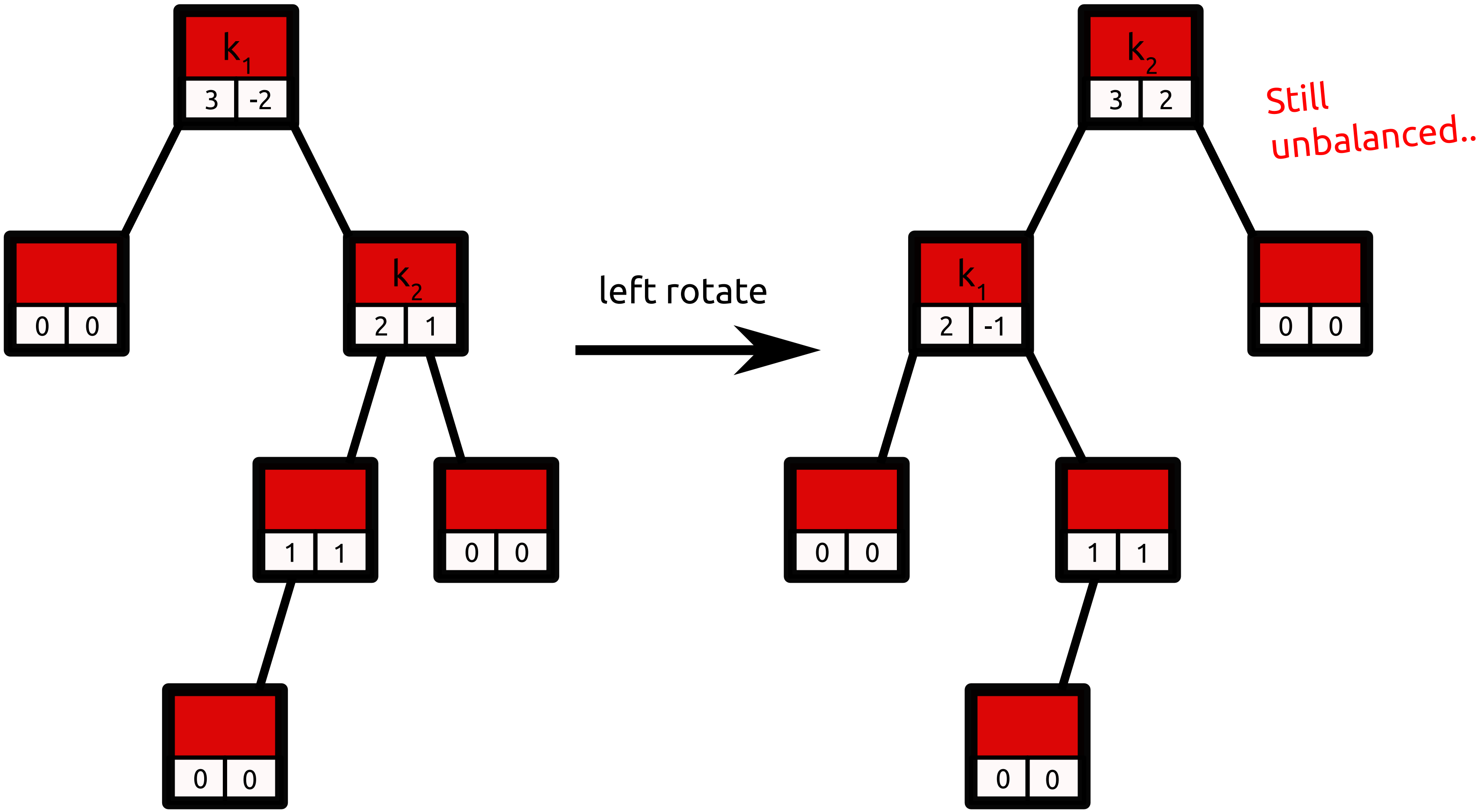
- Resolves case #1.



AVL Tree

Insert: When a Single Rotation is Insufficient

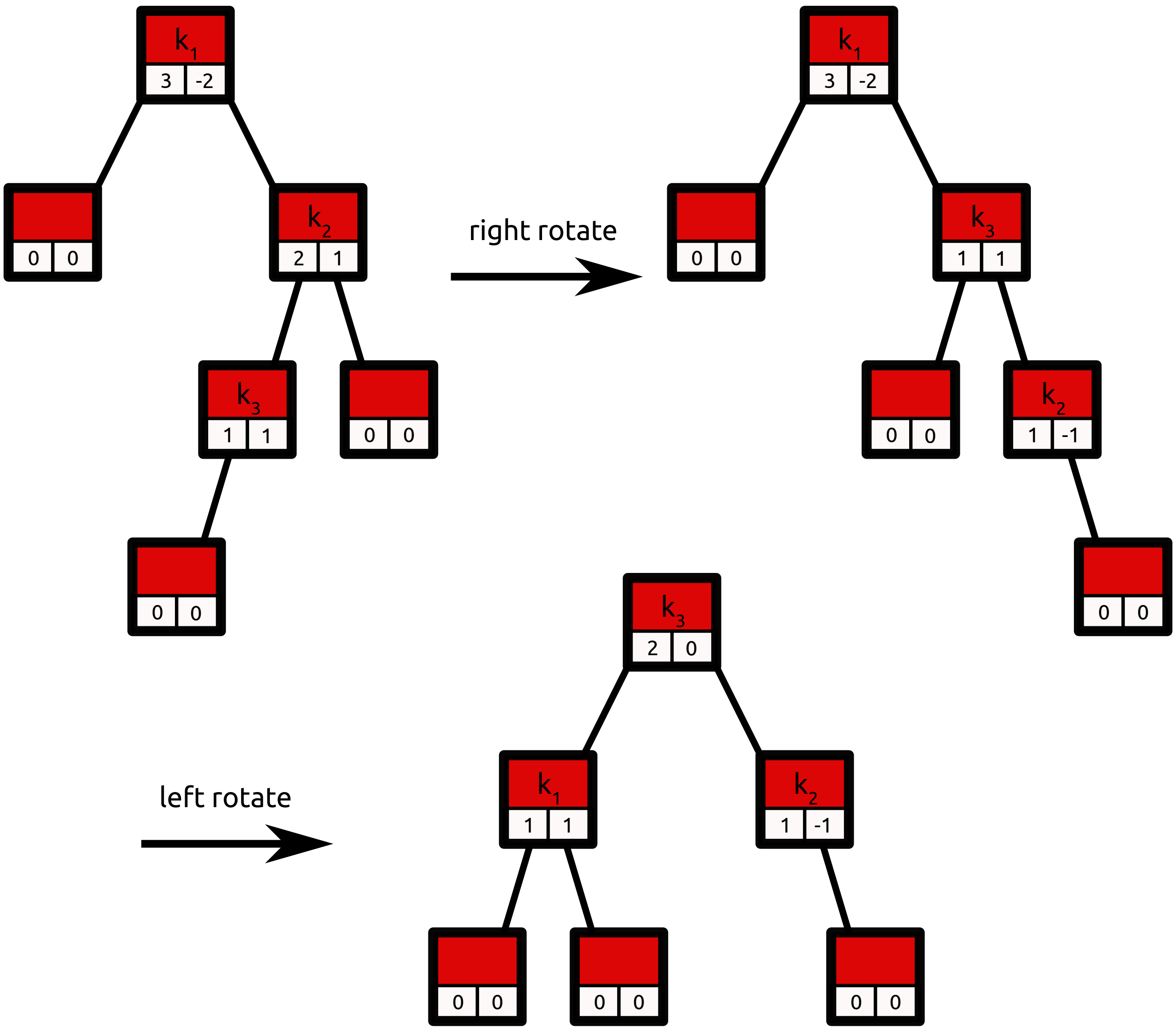
- Corresponds to case #3.



AVL Tree

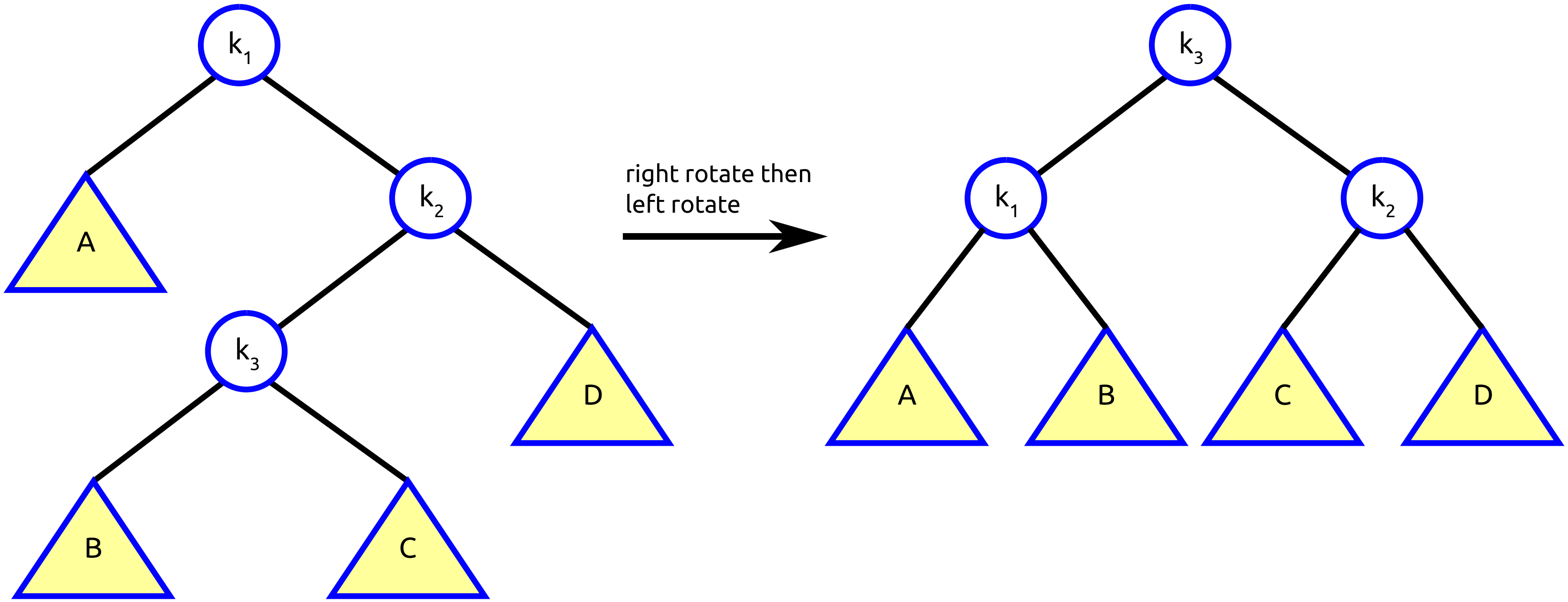
Right-left Double Rotation

- Solve case #3 with right rotation *followed by* left rotation.



AVL Tree

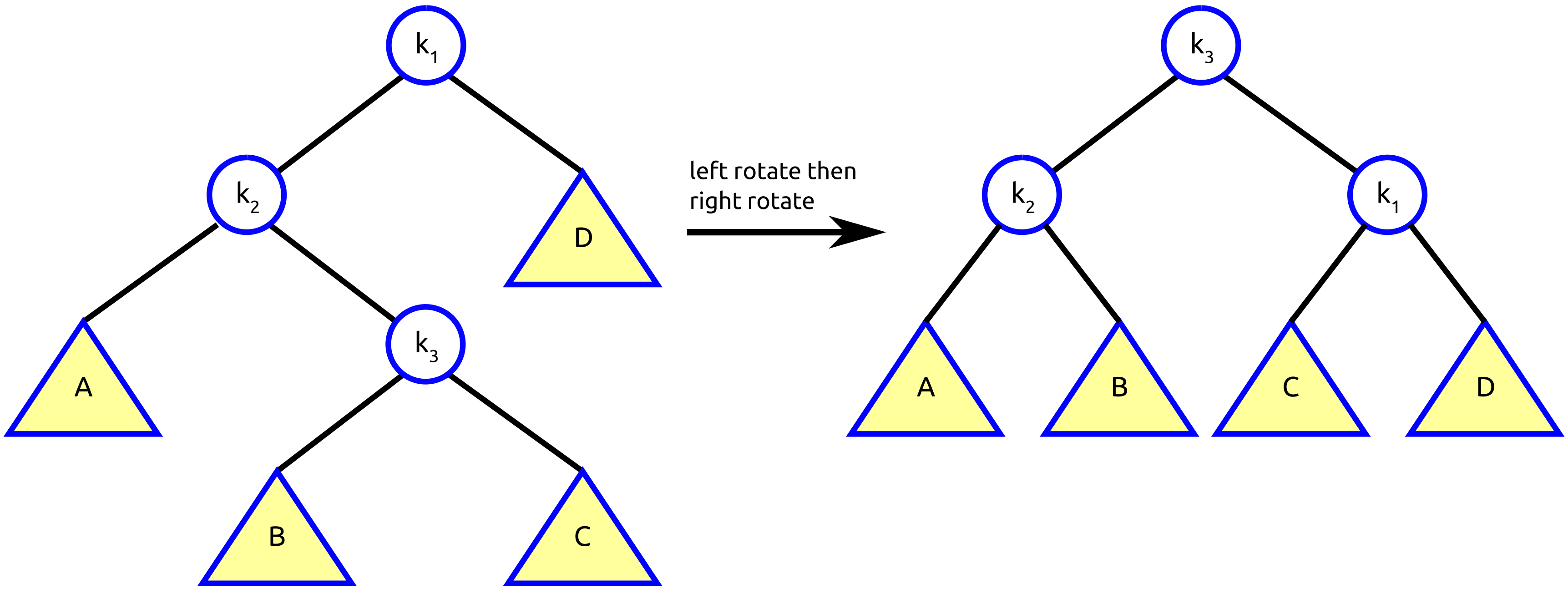
Right-left Double Rotation: Generic View



AVL Tree

Left-right Double Rotation: Generic View

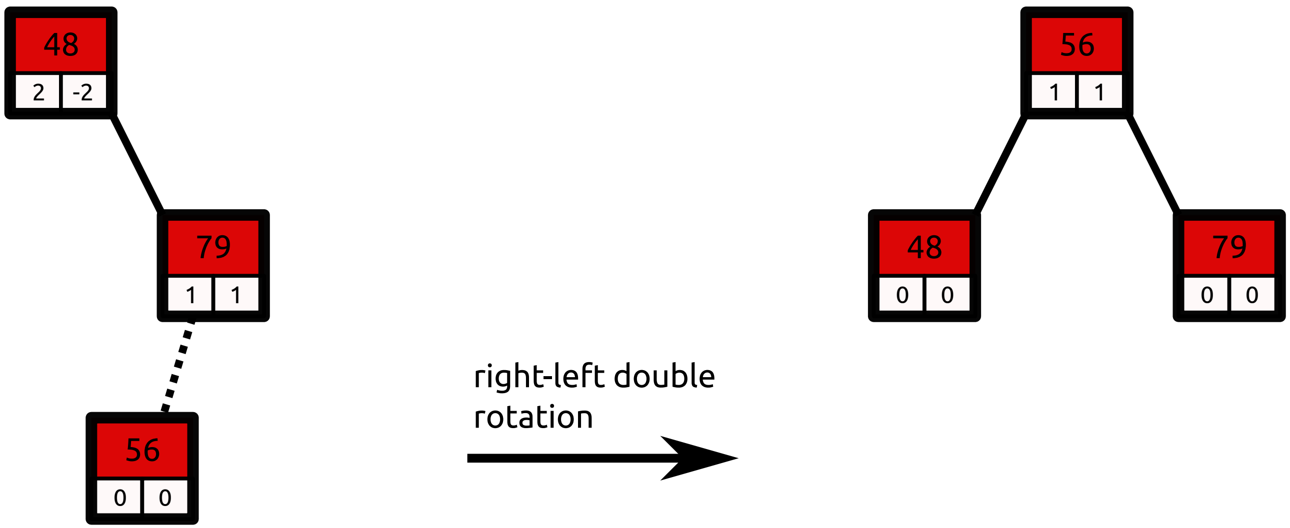
- Solve case #2 with left rotation followed by right rotation.



AVL Tree

Examples of Insertions

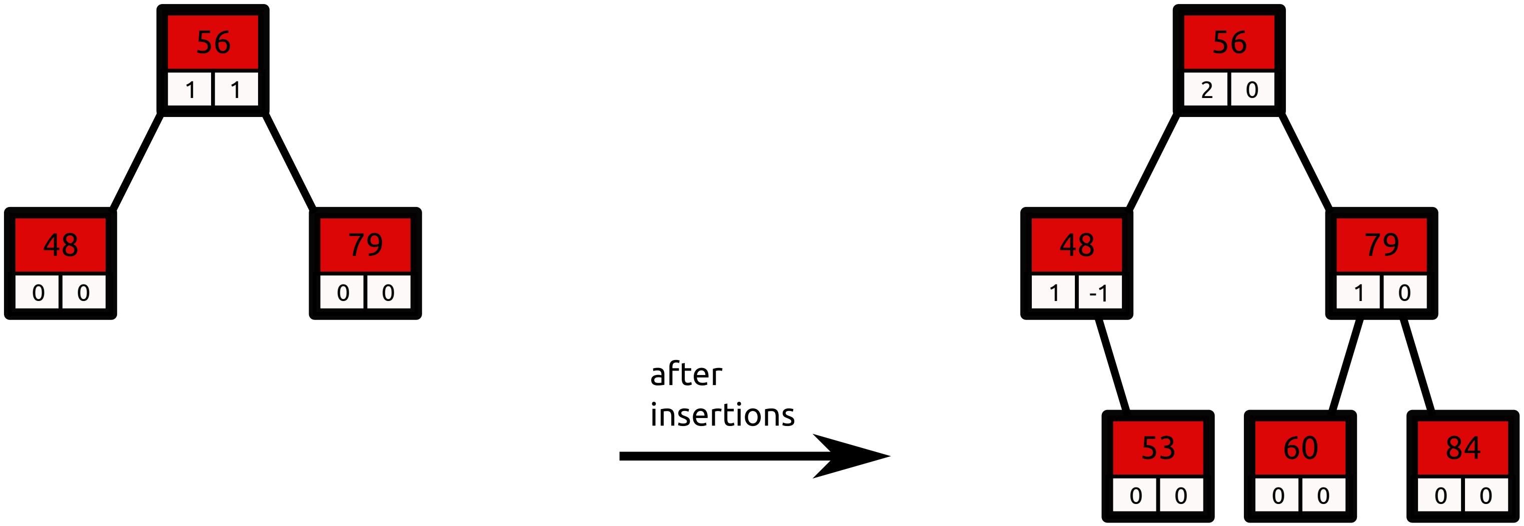
insert(56):



AVL Tree

Examples of Insertions

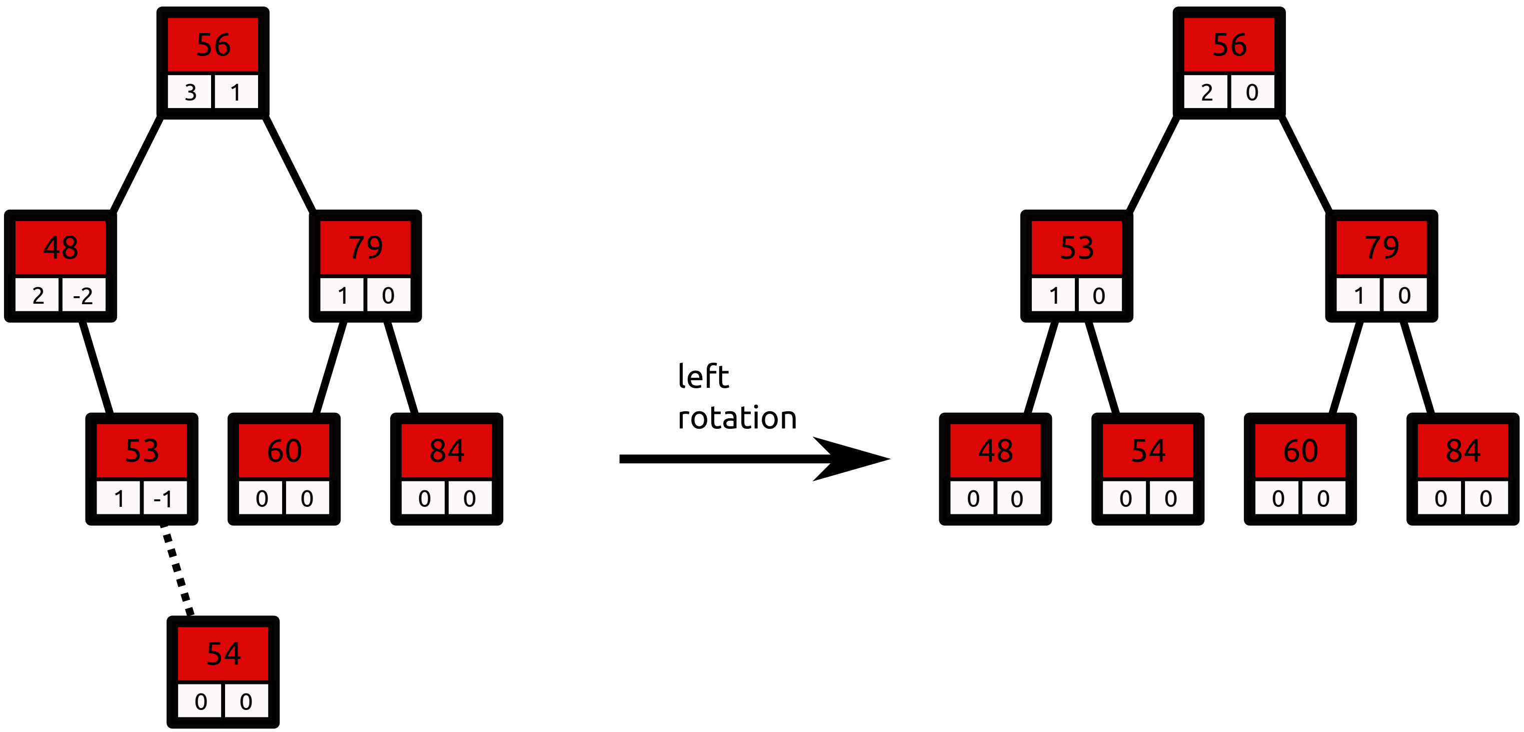
insert(53), insert(60), insert(84):



AVL Tree

Examples of Insertions

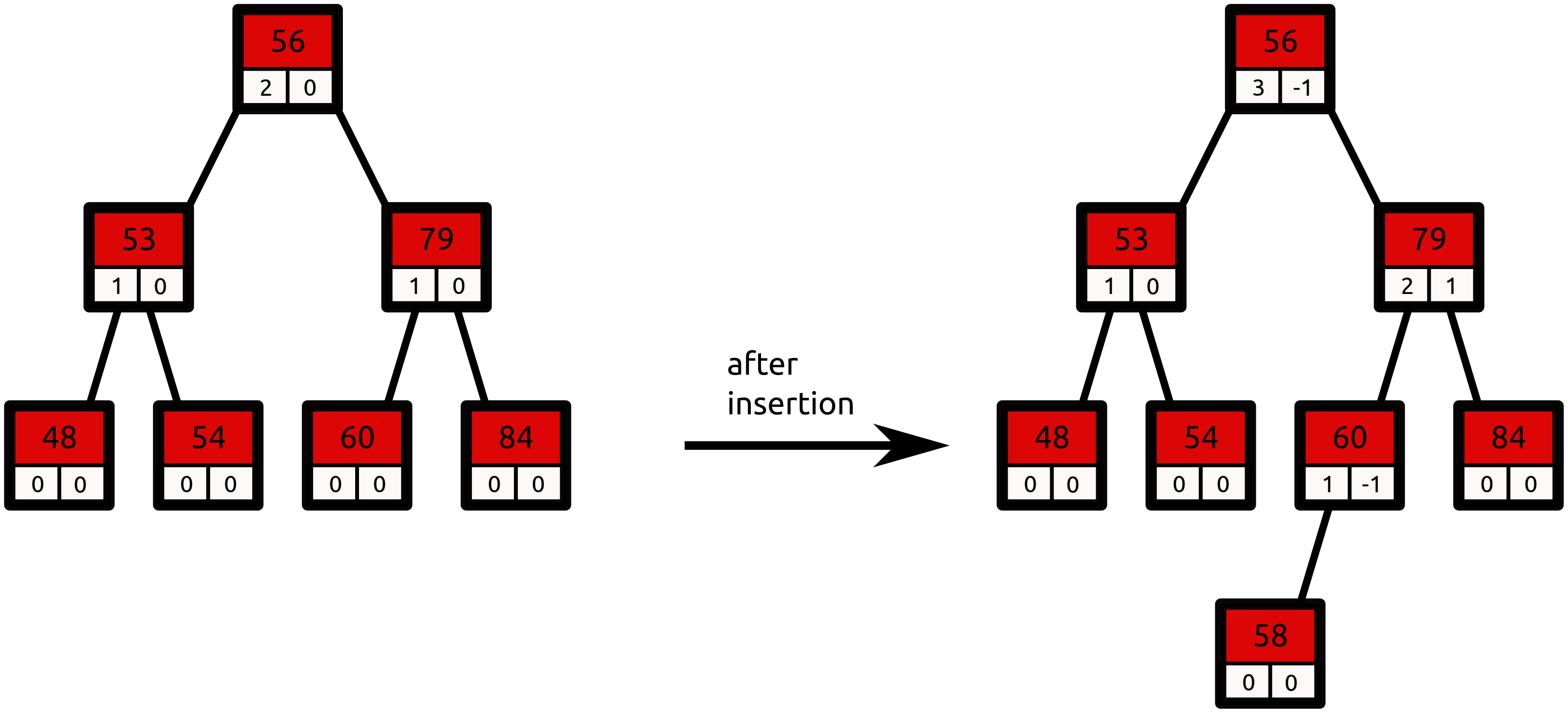
insert(54):



AVL Tree

Examples of Insertions

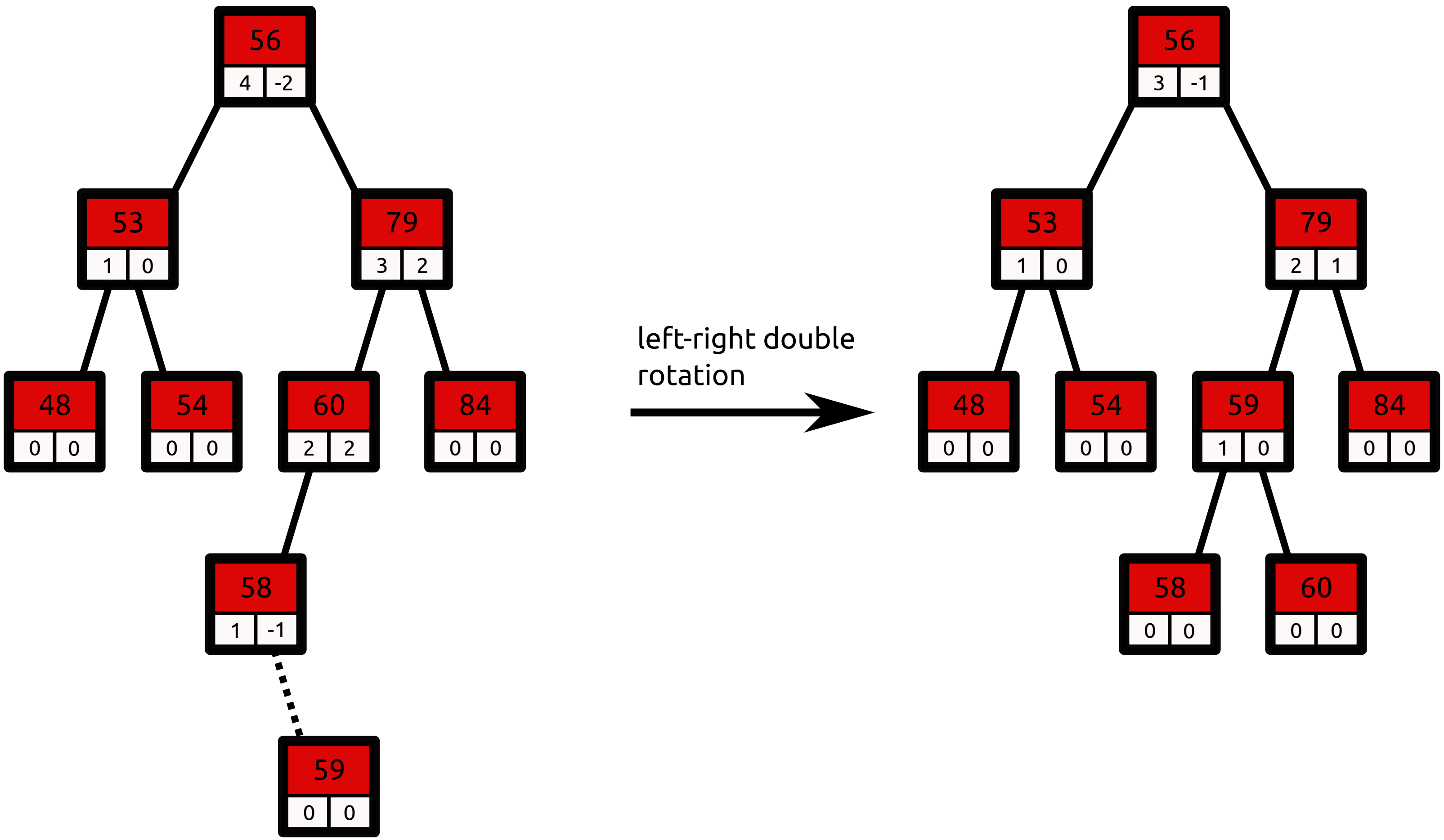
insert(58):



AVL Tree

Examples of Insertions

insert(59):



AVL Tree

Notes on Textbook[1] Implementation

- The book has a `BinarySearchTree` class implementation that we didn't talk about.
- Their `AVLTree` class subclasses it / inherits from it. We probably will not talk about inheritance in this course[2], and you will not be expected to understand inheritance unless I teach it, but feel free to ask questions about it.
- Their `AVLTree` does not implement deletion.

1. *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

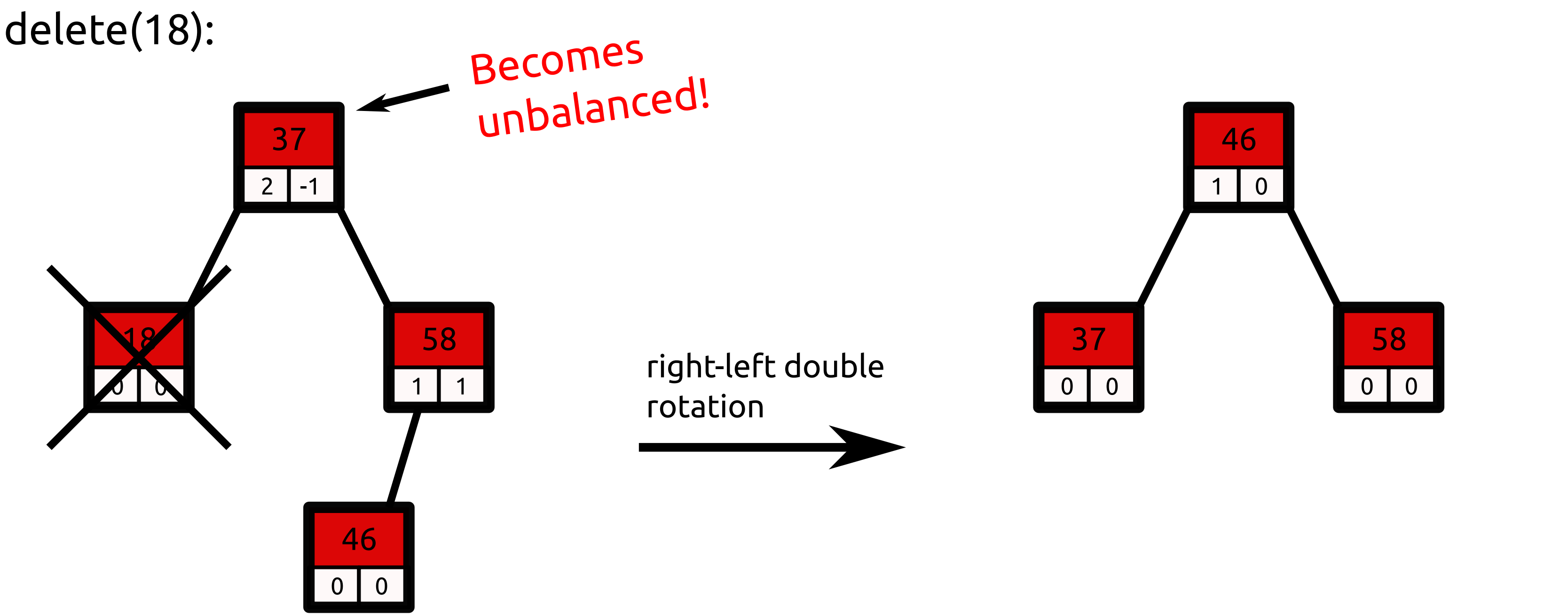
2. We will talk about inheritance in ECS 34, but even then, inheritance is a concept that is fading in popularity. One new popular language, Golang, does not even support inheritance (at least, not in the traditional sense).

AVL Tree

Deletion

- Start with same procedure as BST.
- Going from replacement node (if any) up to root of tree, rebalance any unbalanced node.

Example #1

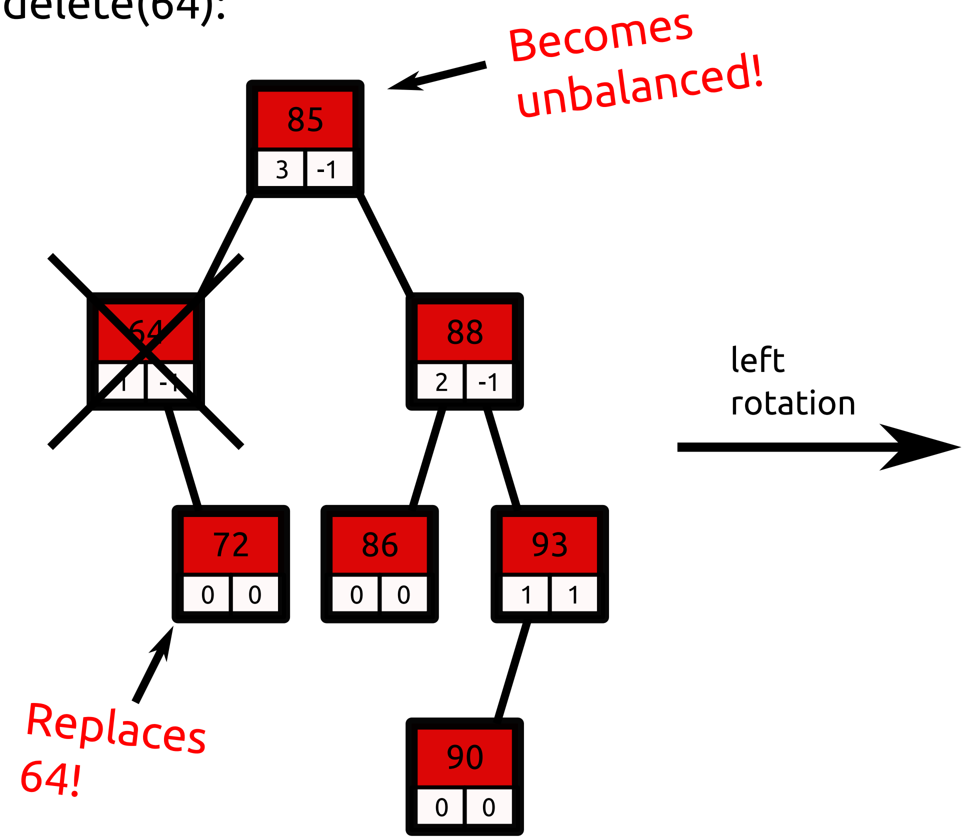


AVL Tree

Deletion

Example #2

delete(64):



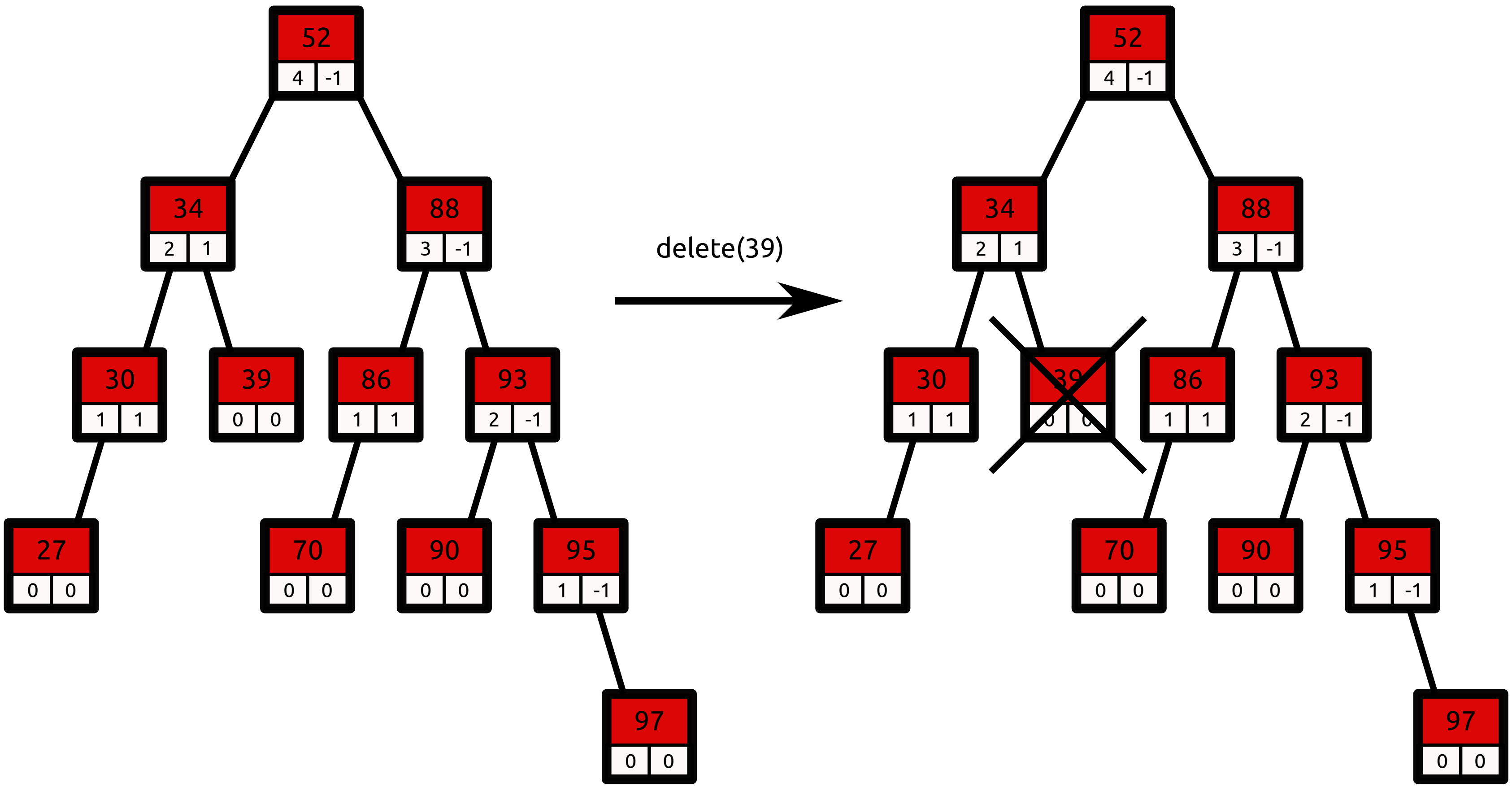
- 93 (instead of 86) is involved in rotation because higher height.

AVL Tree

Deletion

- May have to rebalance at multiple nodes.

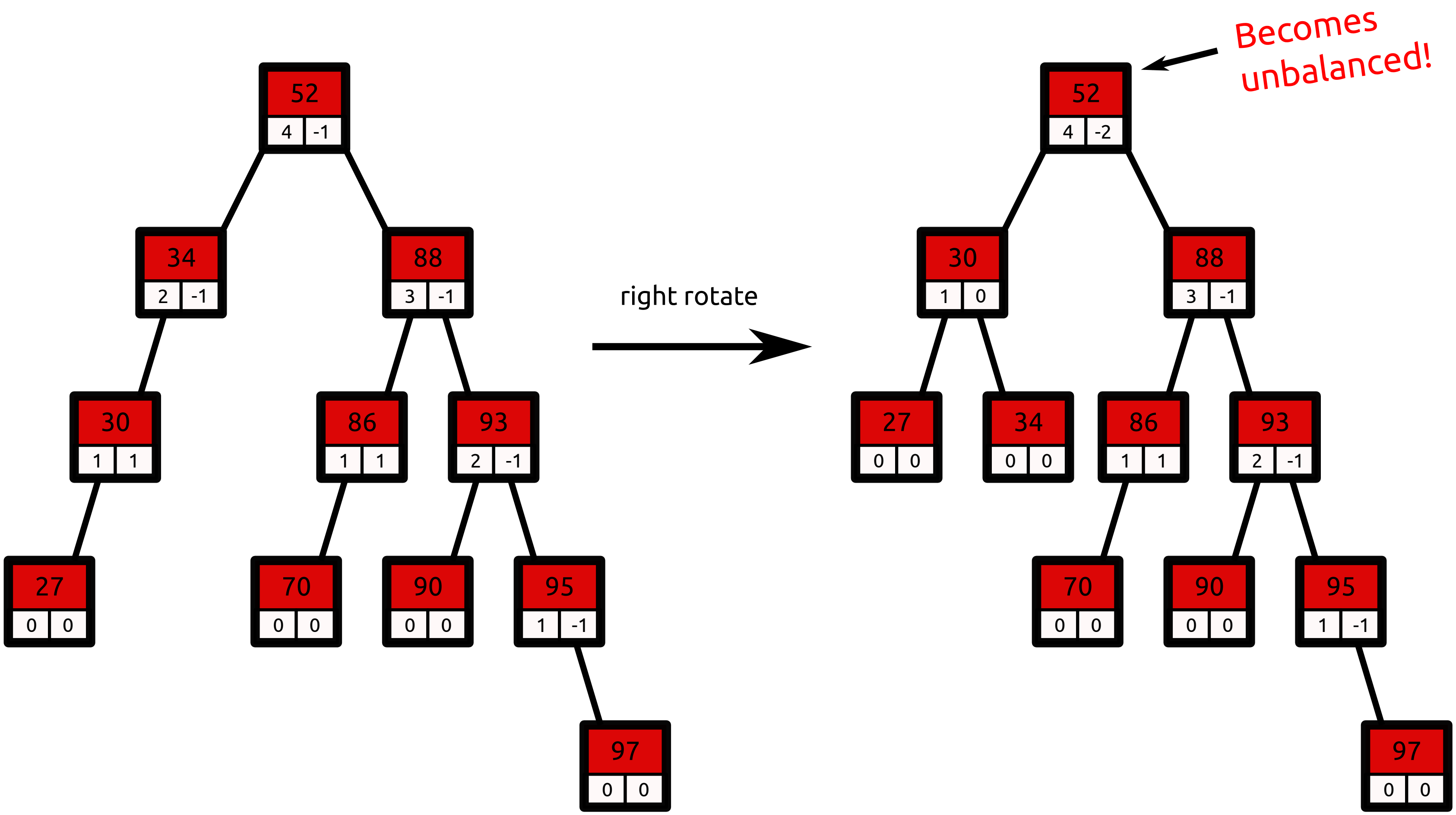
Example



AVL Tree

Deletion

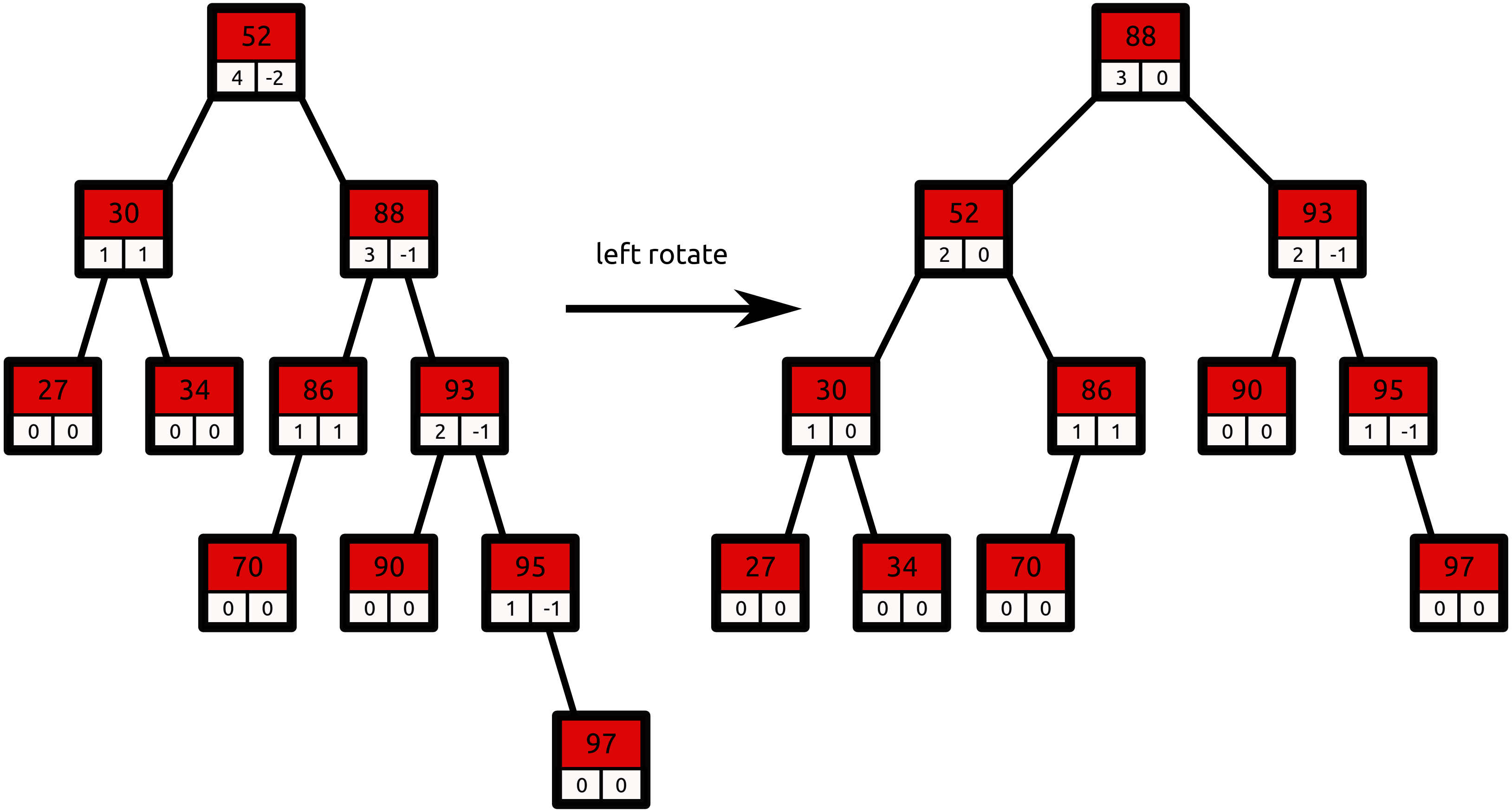
Example



AVL Tree

Deletion

Example



Splay Tree

- *I'll show you how the tree works and then explain the use of it / why one might use it over an AVL tree.*

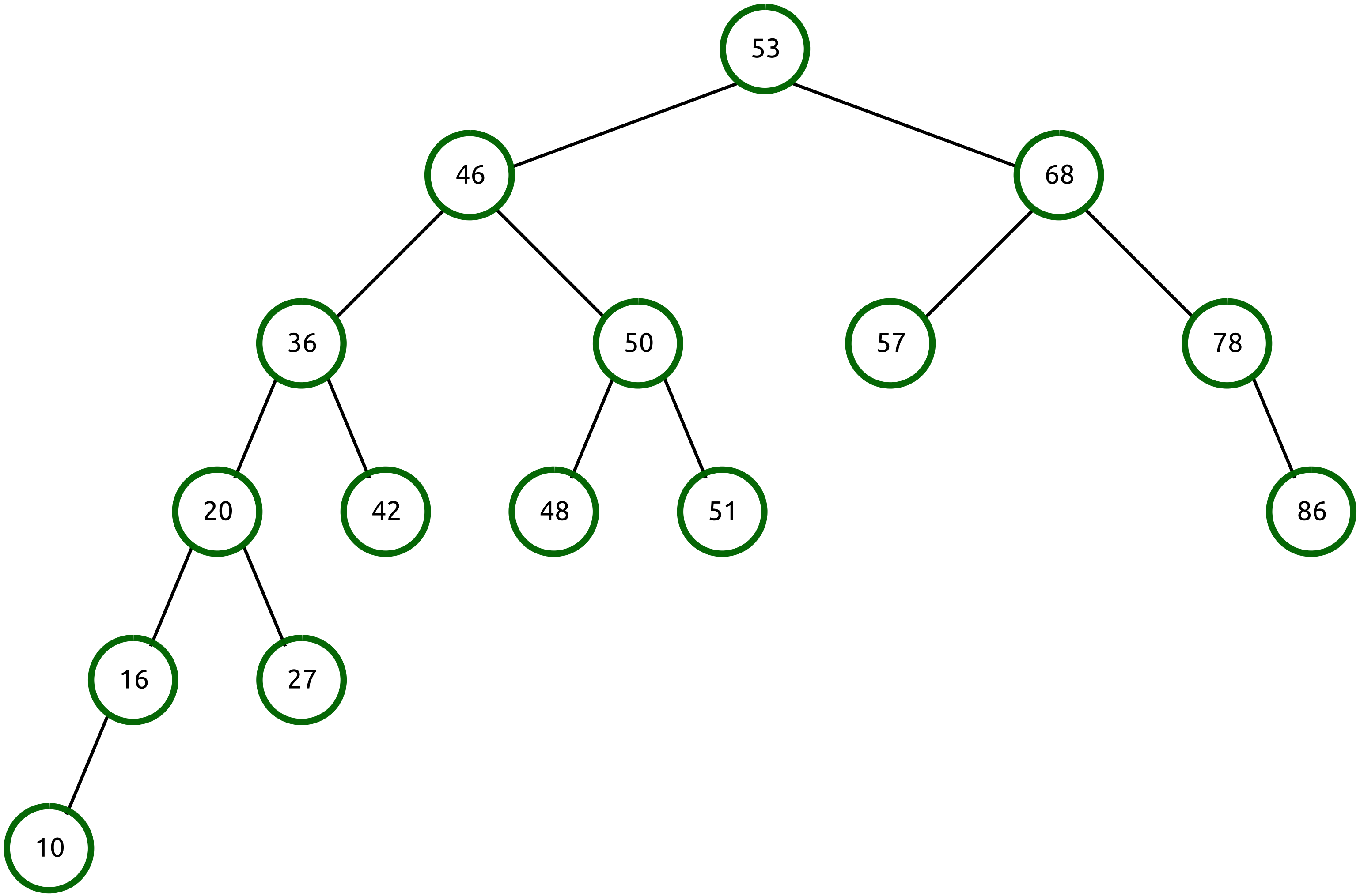
Quick Description

- Self-balancing, but less rigidly so.
- No balance factors.
- When a node is *accessed* (whether by find, insert, or delete), the node is pushed[1] to the top.

1. Some might say the node is "splayed", but I'm not actually sure that's a real verb.

Splay Tree

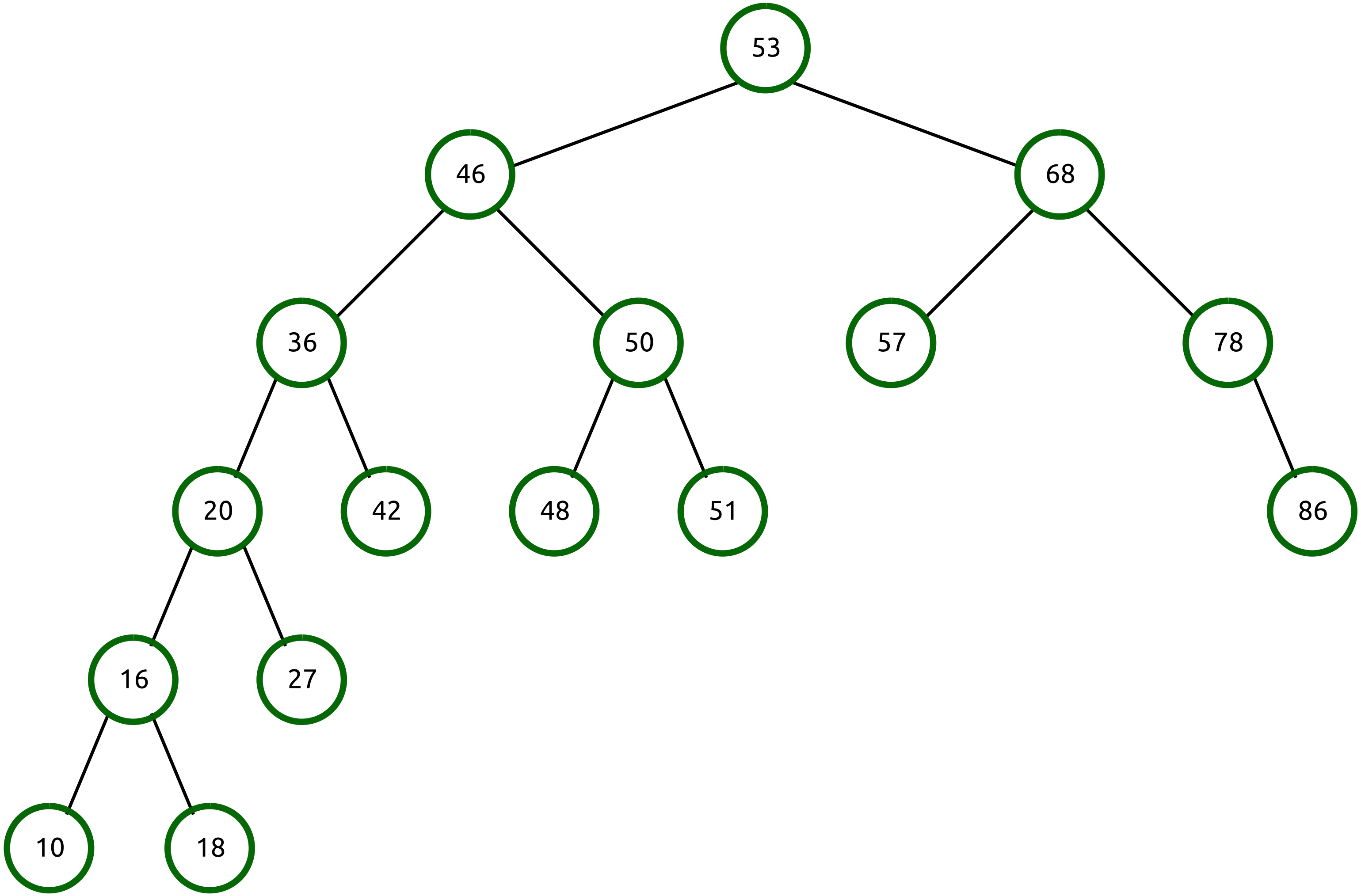
Example #1



- Let's insert 18.

Splay Tree

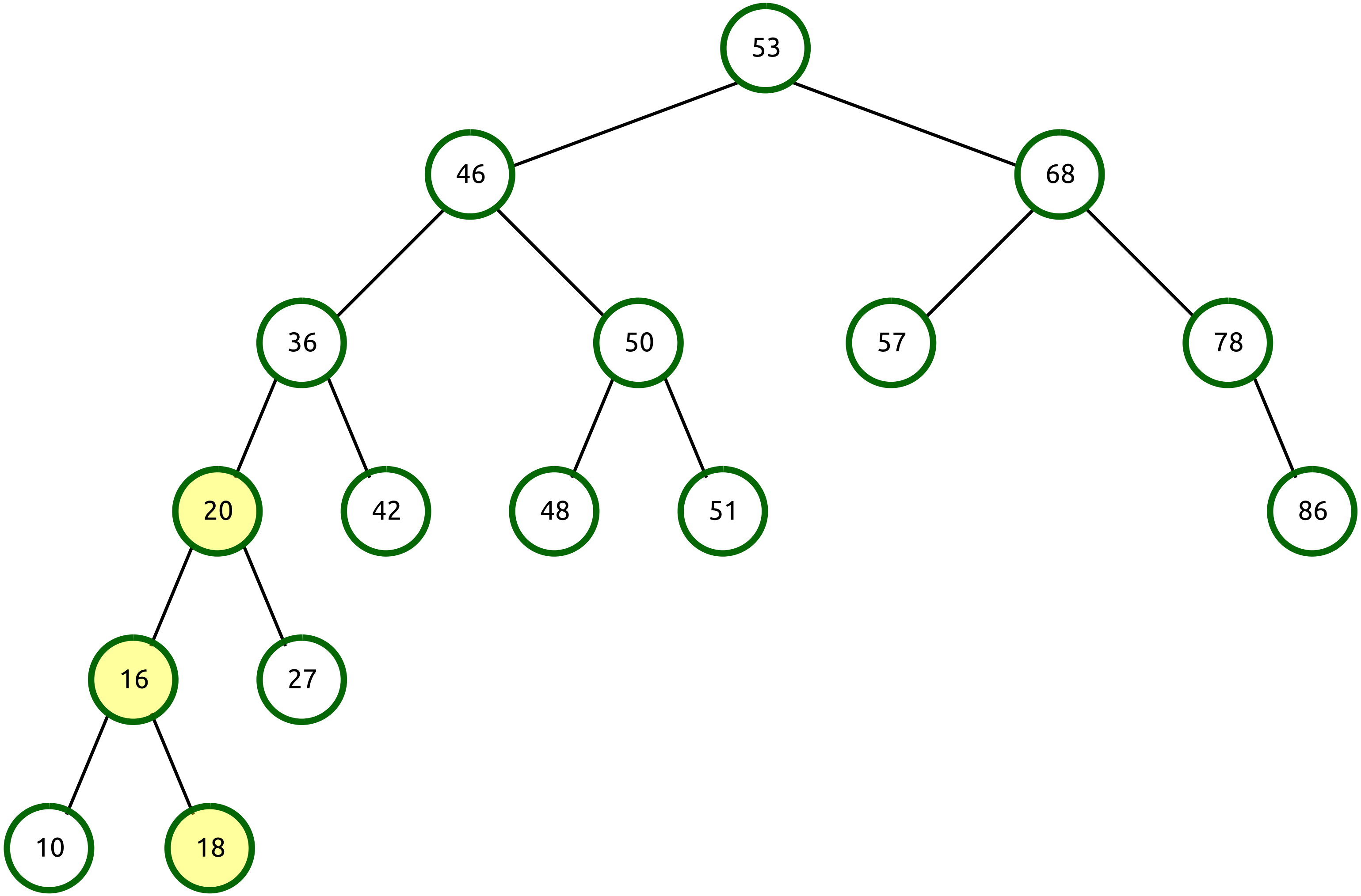
Example #1



- Must splay 18 to the root.

Splay Tree

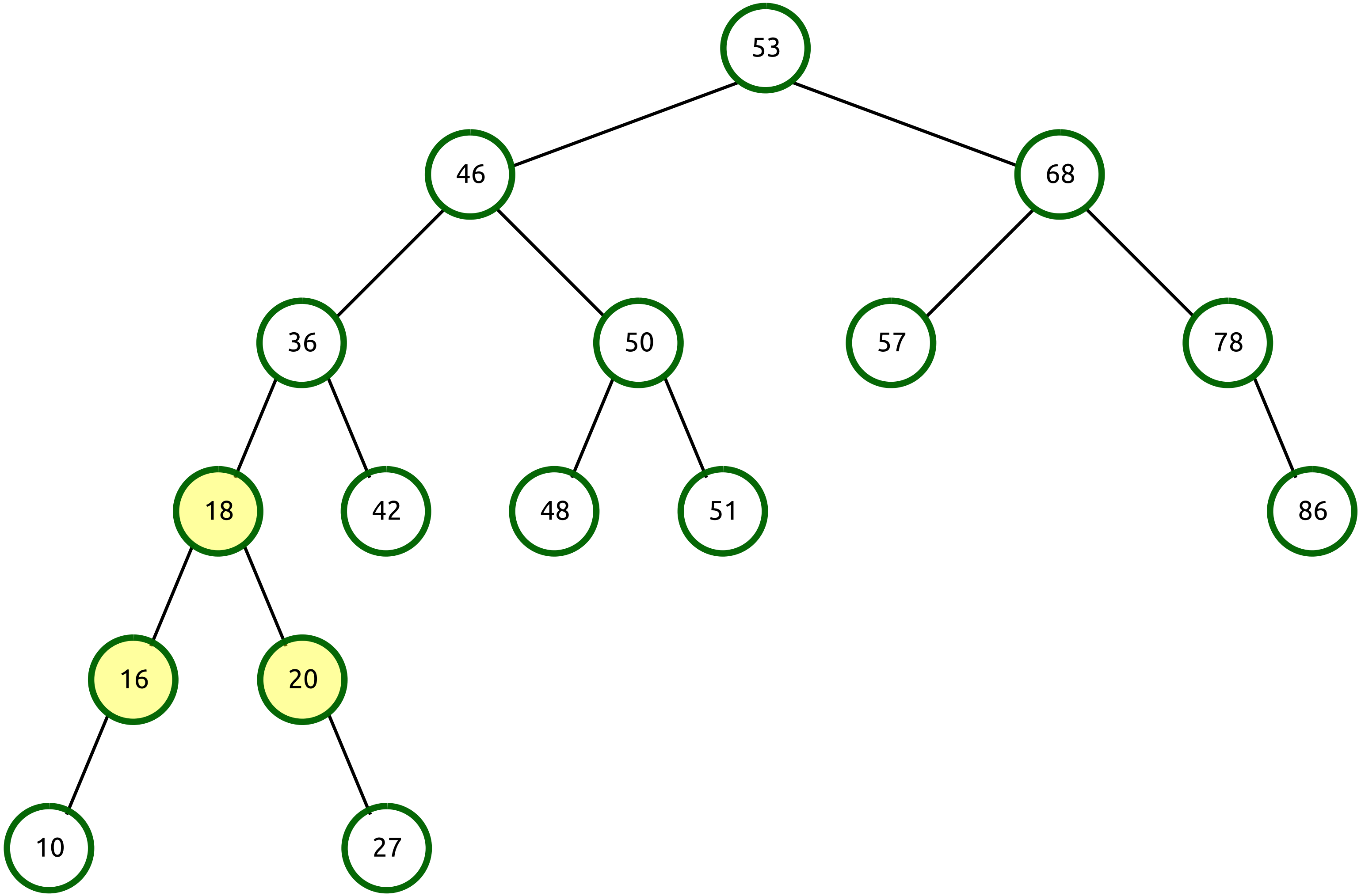
Example #1



- Do a zig-zag! (i.e. an AVL tree-style left rotation and right rotation)

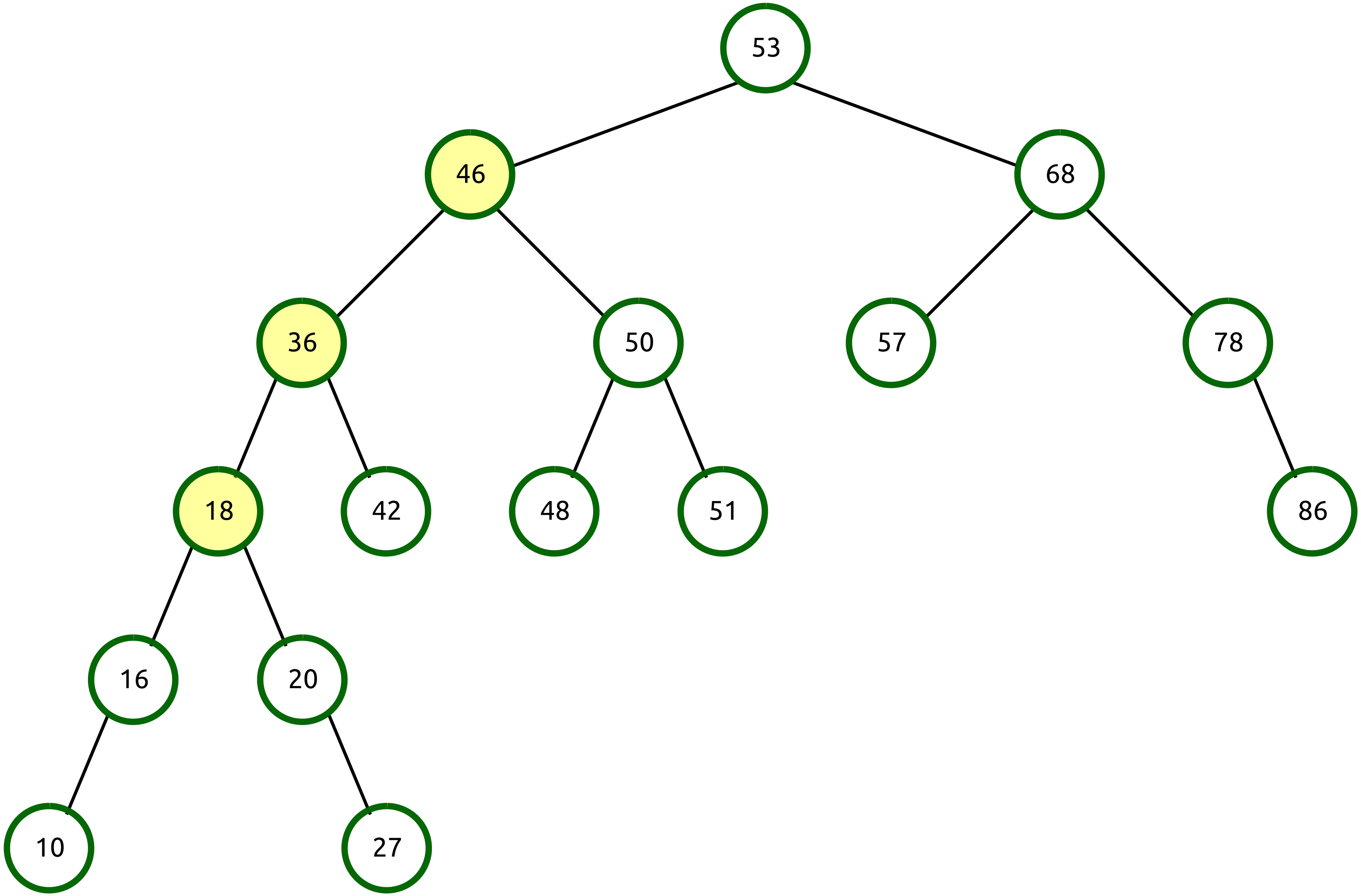
Splay Tree

Example #1



Splay Tree

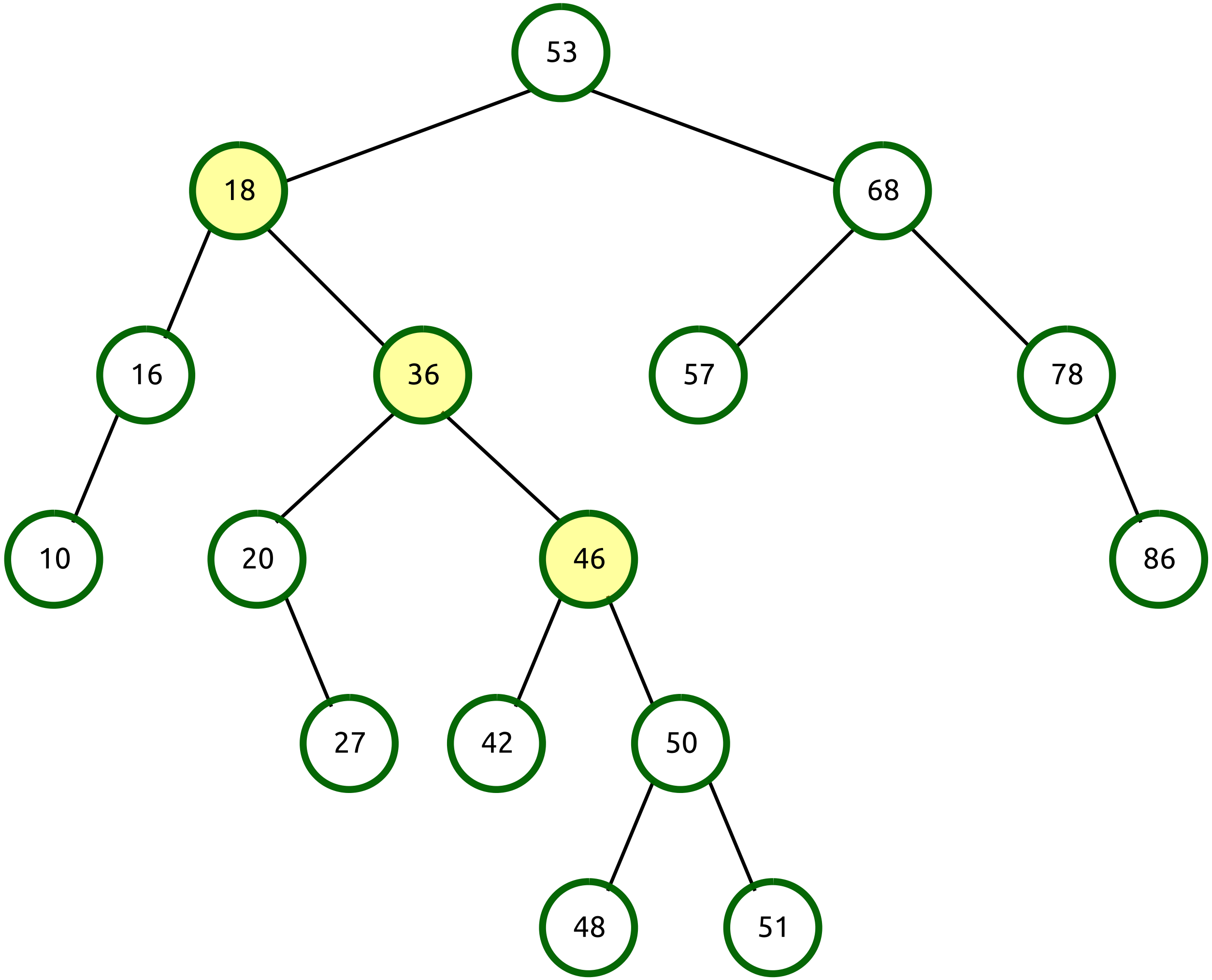
Example #1



- Now do a zig-zig! (not similar to anything AVL tree does)

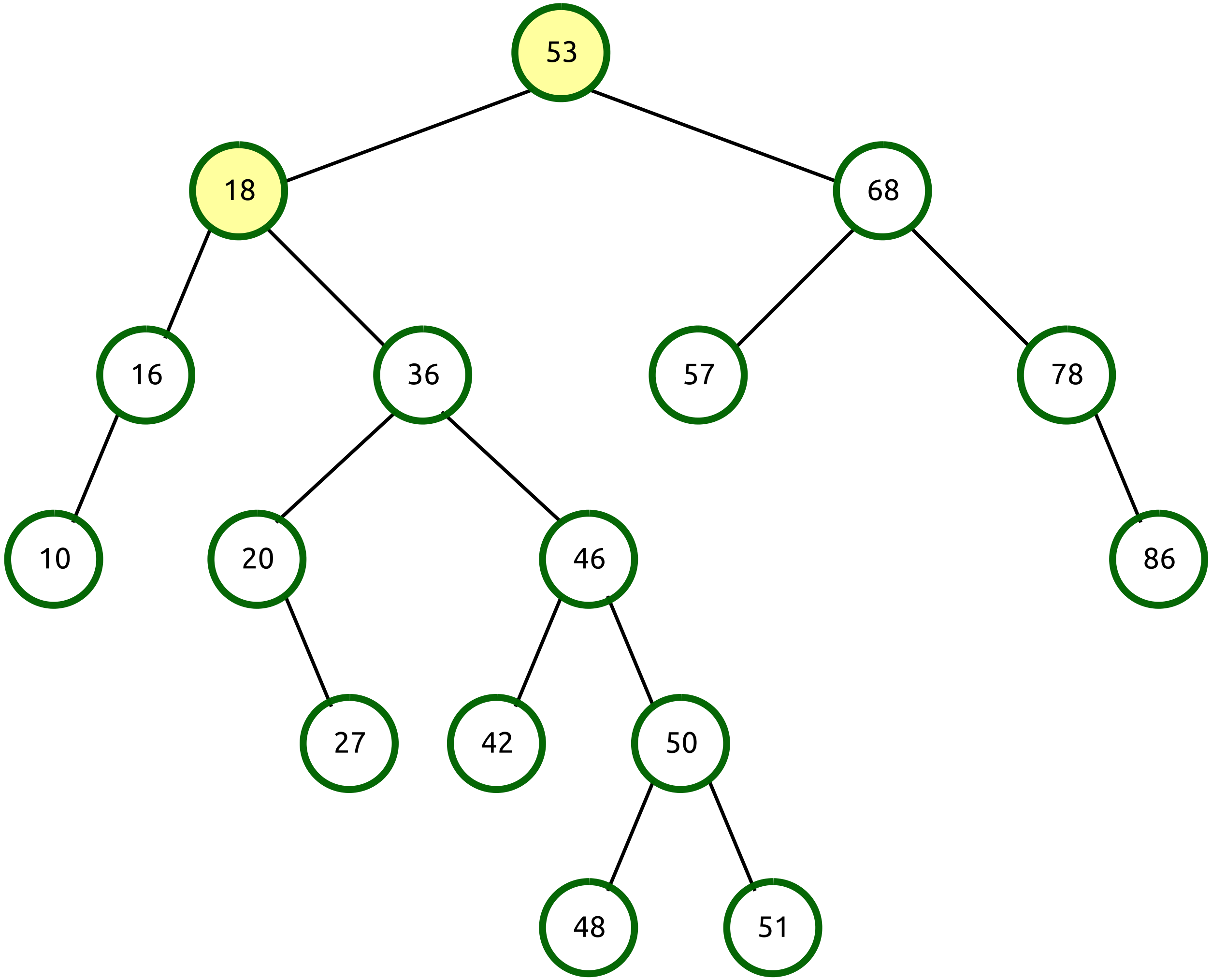
Splay Tree

Example #1



Splay Tree

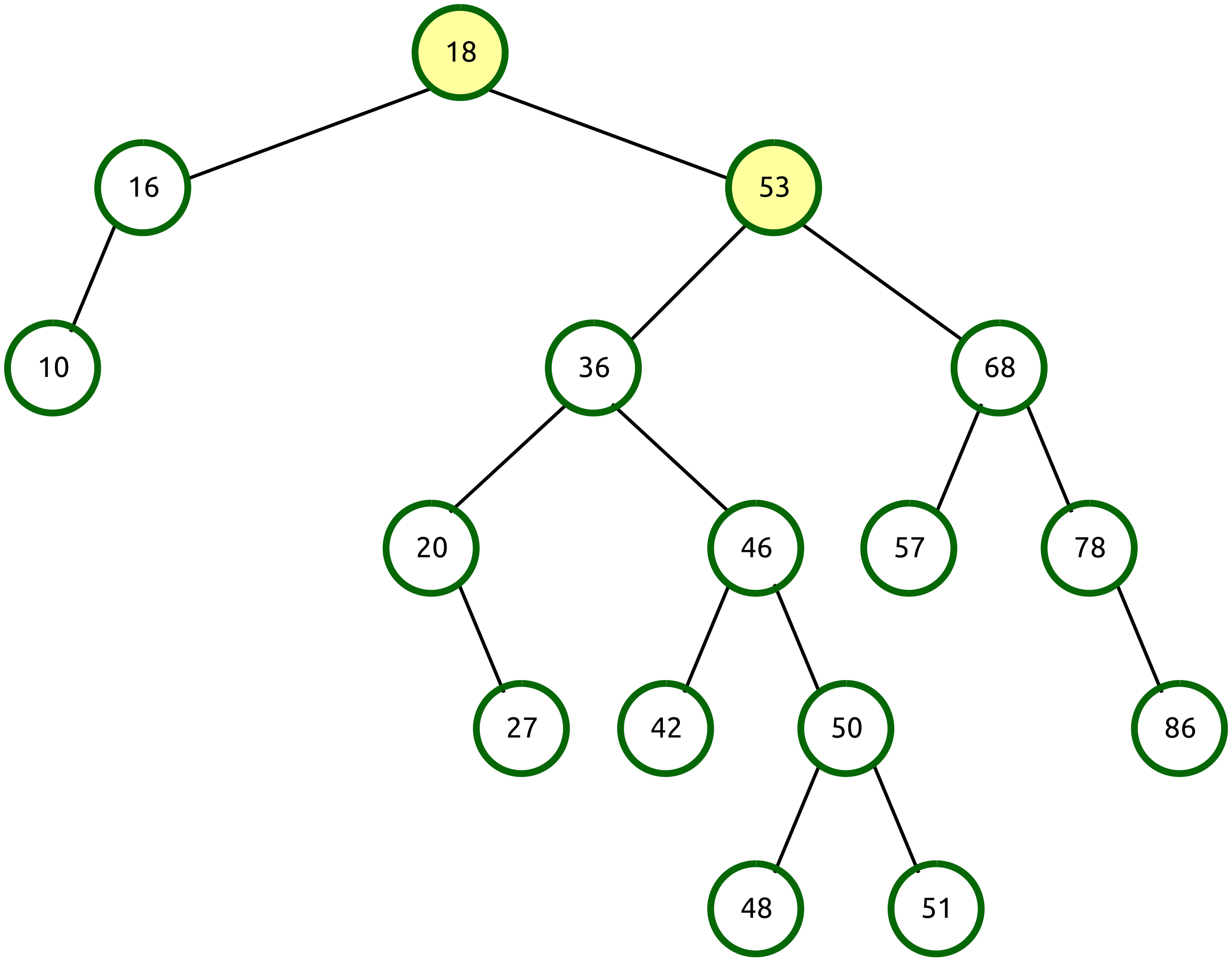
Example #1



- Then a small pivot (a zig?).

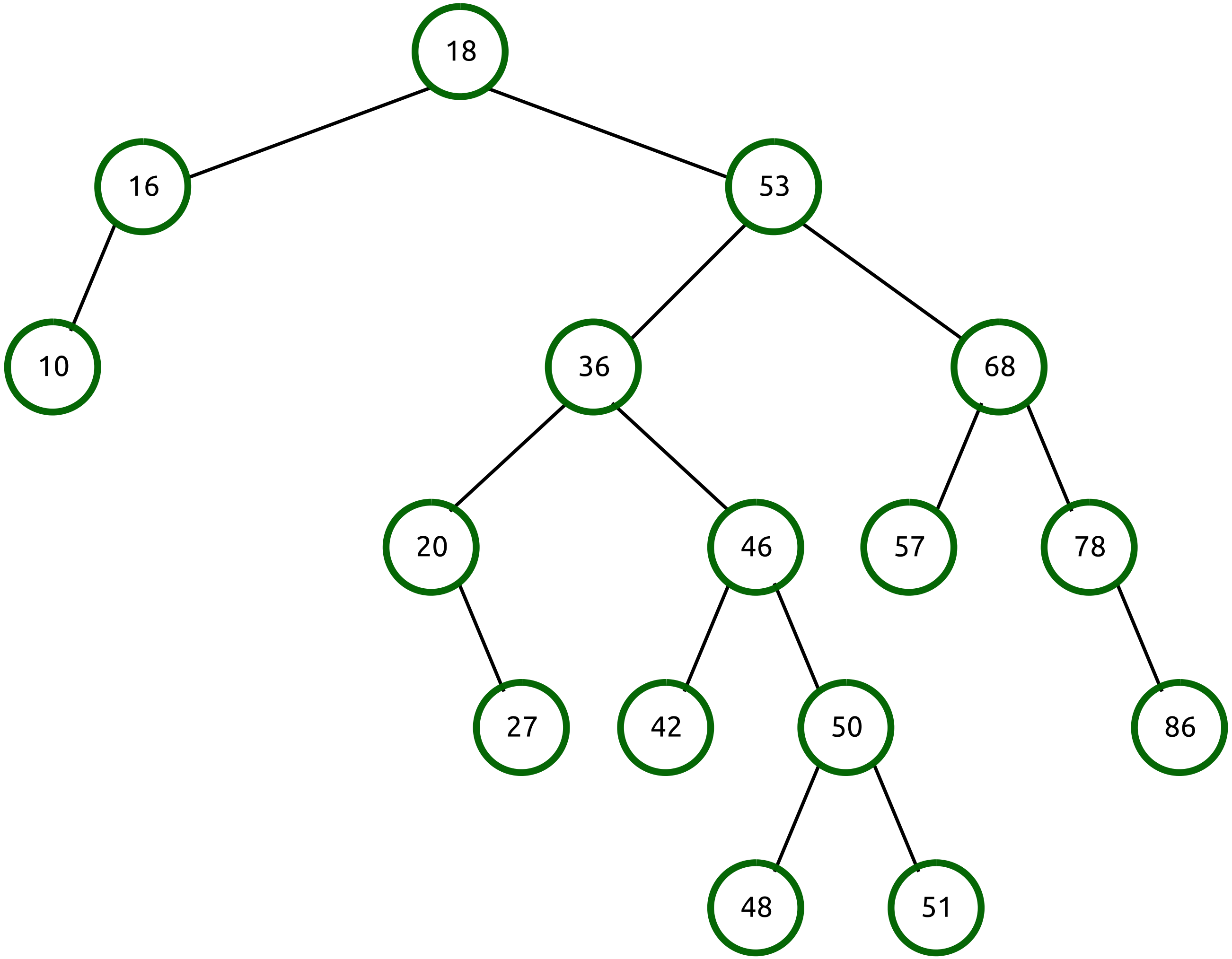
Splay Tree

Example #1



Splay Tree

Example #1



- Done.

Splay Tree

When Splay Operation Occurs

- Push node to the root when:
  - *Insert* the node, i.e. insert the node following normal BST rules, then push to top.
  - *Find* the node.
    - Unlike previous data structures, a find operation modifies the structure.
  - *Delete* the node.

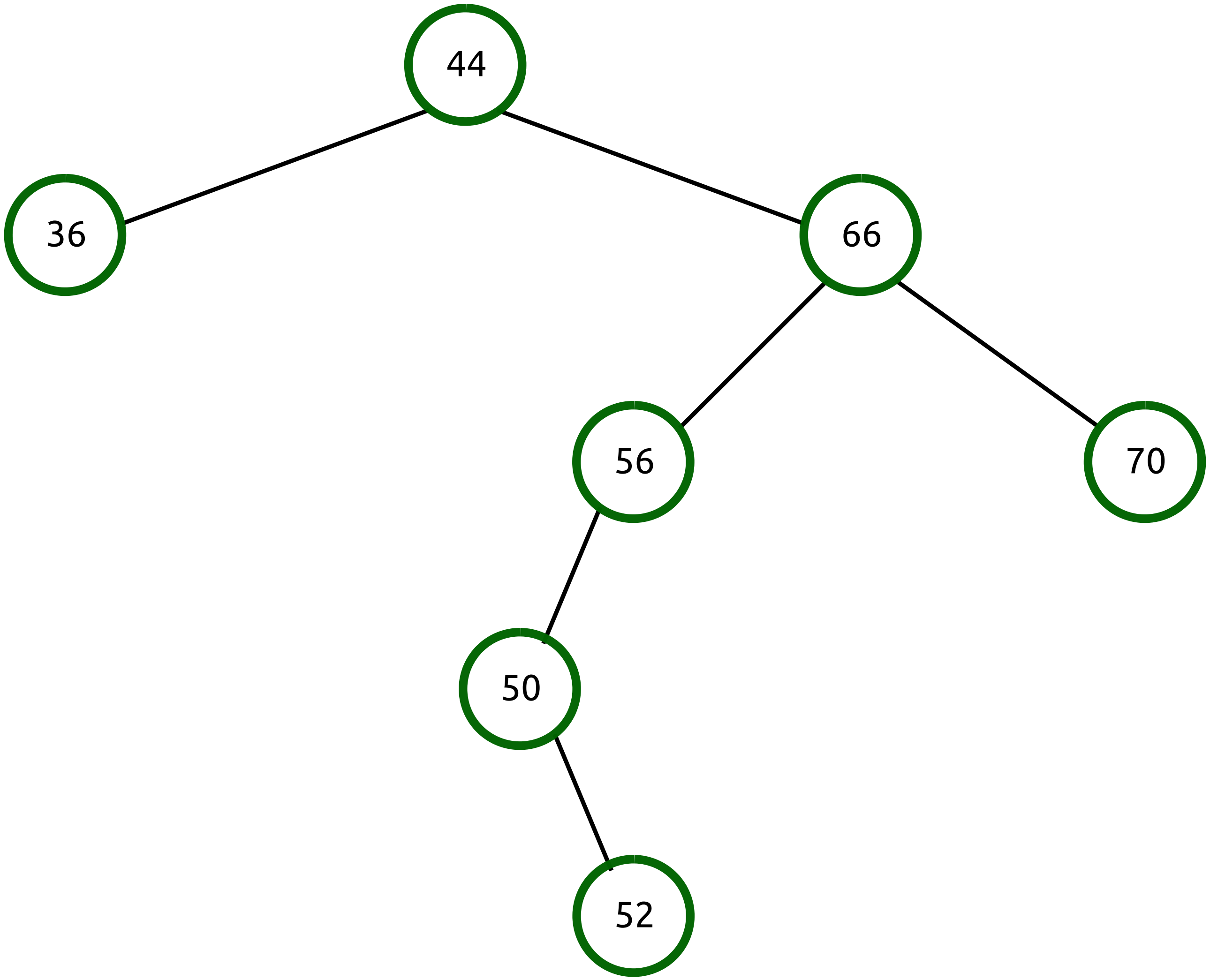
Deletion

1. Push the node-to-delete to the top.
   - Implementation: perform a find operation on it.
2. Delete it (now the root).
3. Find largest in left subtree[1]. Push[2] that element to root of left subtree.
4. Make it root of entire tree.

1. Could instead do smallest in right subtree.
2. This differs from BST which *replaces* instead of *pushes*.

Splay Tree

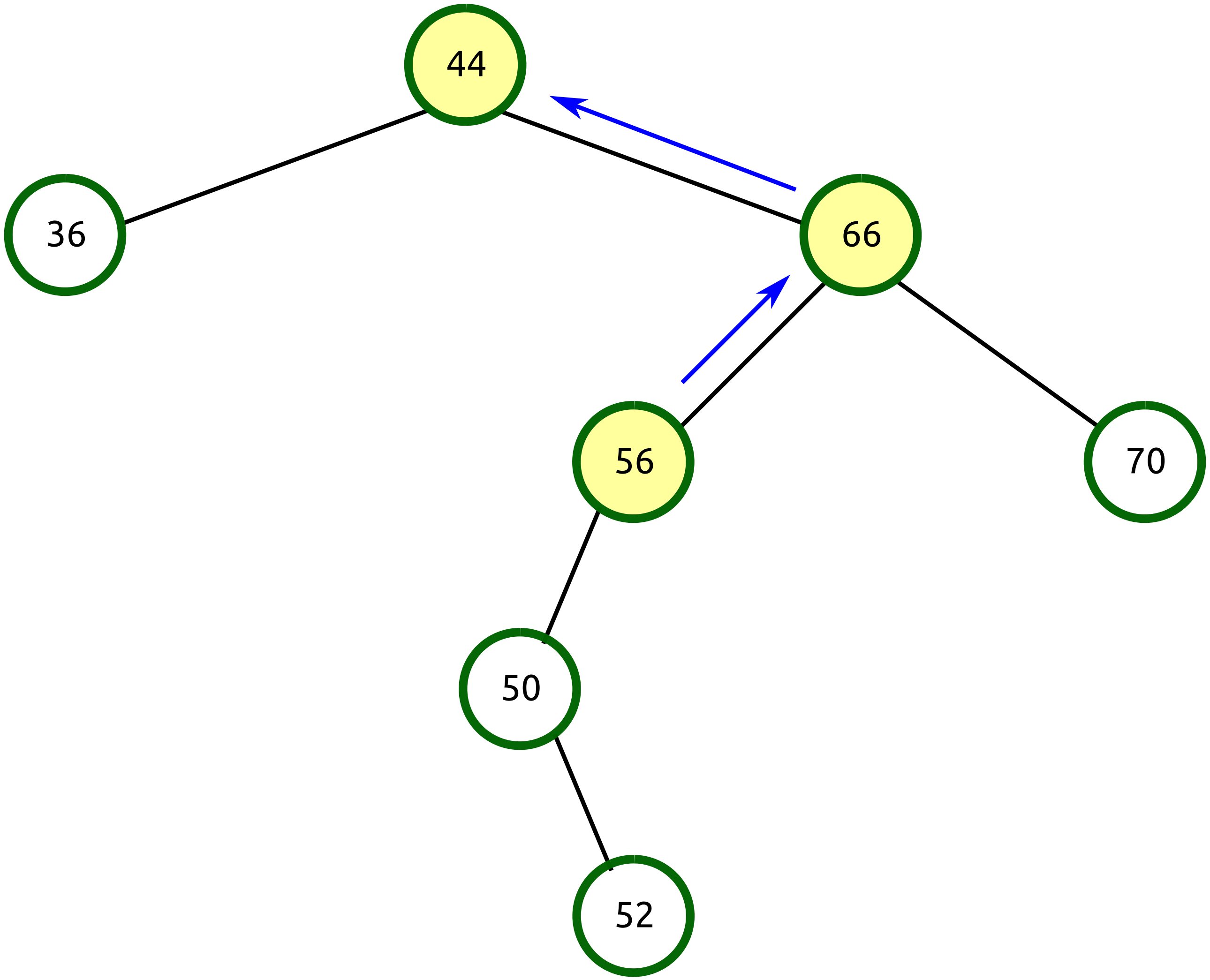
Example #2



- Goal: delete 56.

Splay Tree

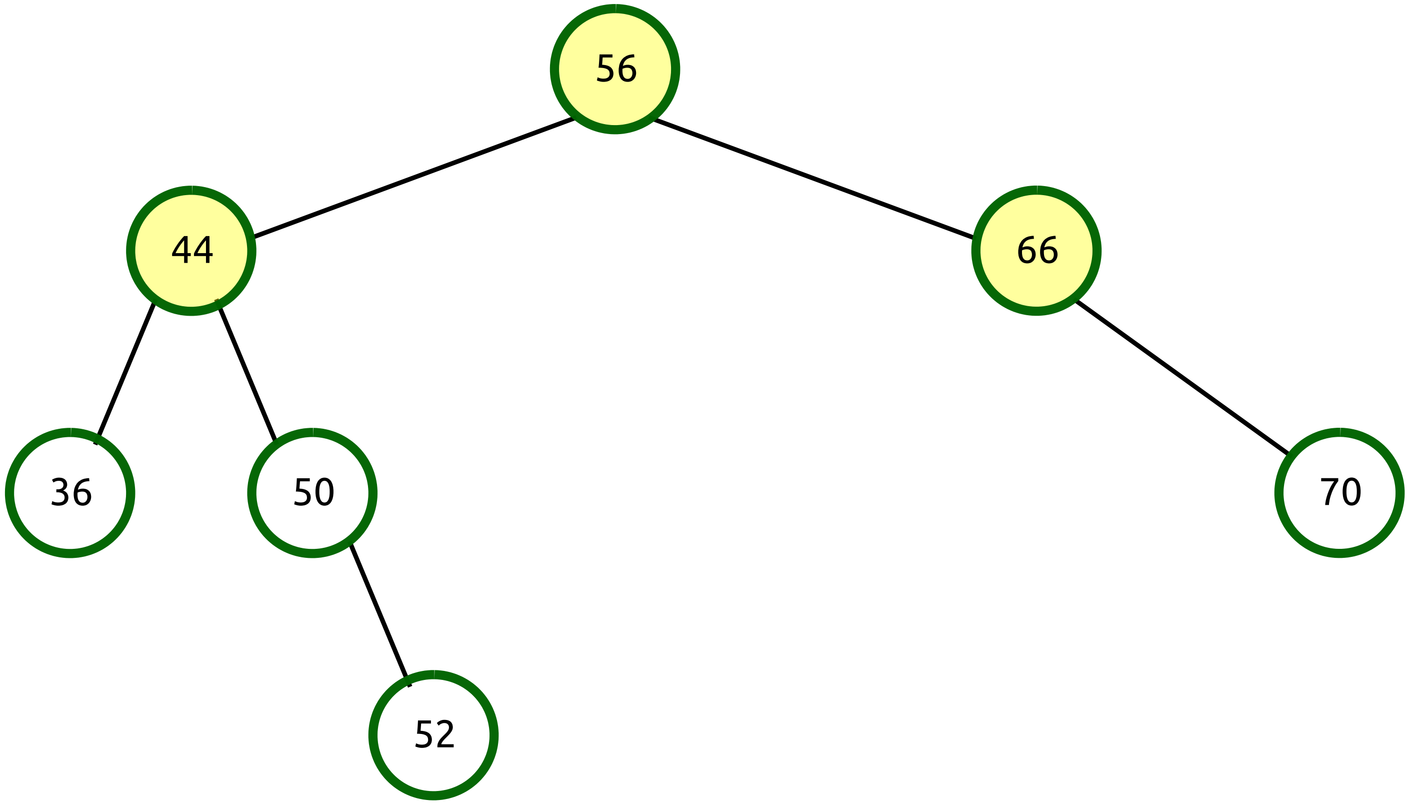
Example #2



- Need to splay 56 to the top: zig-zag!

Splay Tree

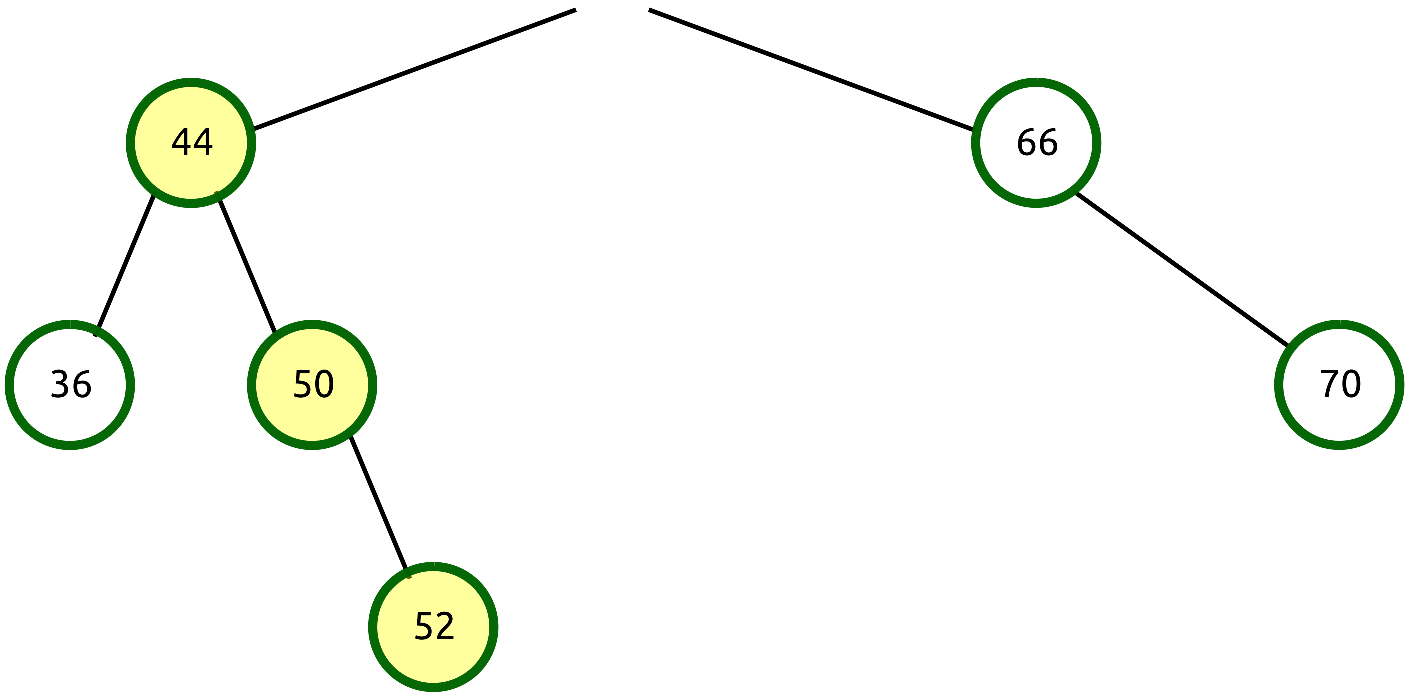
Example #2



- Now, need to remove 56.

Splay Tree

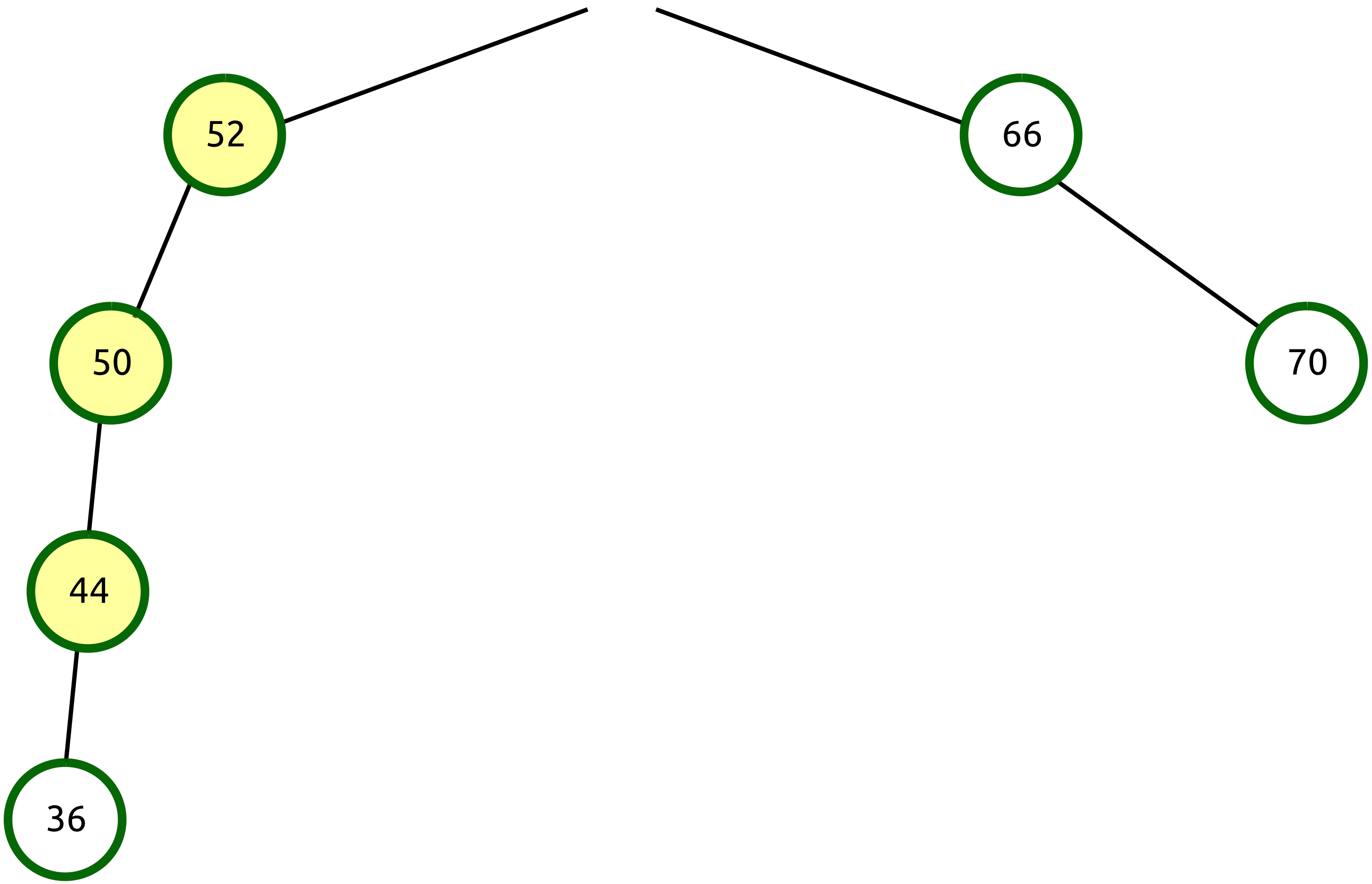
Example #2



- Next, splay highest element in left subtree to top of that subtree: zig-zig!

Splay Tree

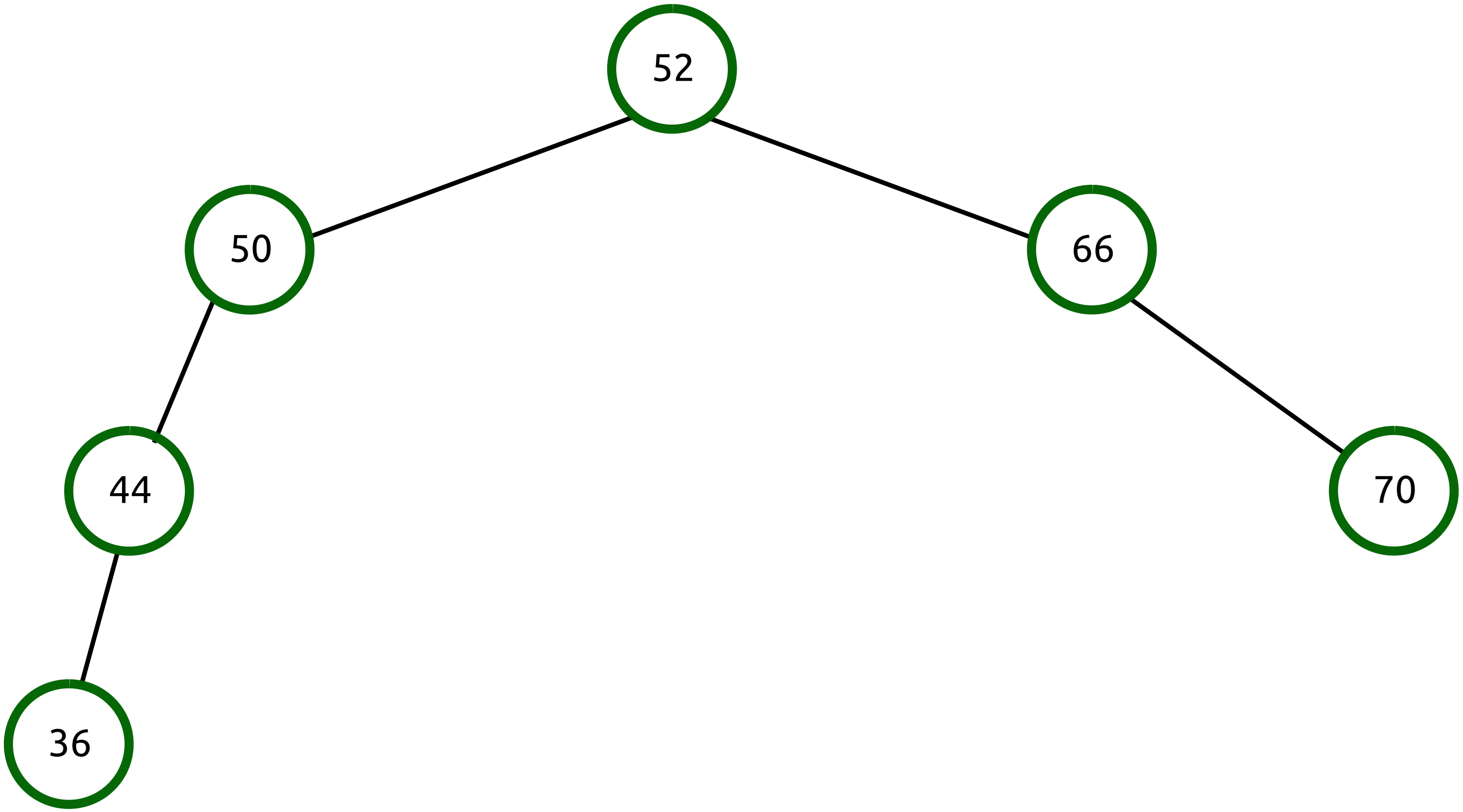
Example #2



- Now, can combine the trees.

Splay Tree

Example #2



- Done.

Splay Tree

Reasons to Use

- Are situations in which data that was recently accessed is more likely to be accessed.

Worst-Case Time Complexity

- Single operation (find/insert/delete): $\Theta(n)$ in worst case.
- m operations (any combination of find/insert/delete): $\Theta(m \lg n)$ time in worst case $\Rightarrow$ single operation takes ***amortized*** $\Theta(\lg n)$ time in worst case.
    - Splay operation on a deep node will bring it and its surrounding nodes up $\Rightarrow$ worst-case scenario *provably* can't happen consistently, unlike normal BST.

Compared to AVL Tree

- AVL tree is preferable for *consistency*.
- Splay tree is easier to program[1].
- Splay tree takes less storage; no height info.

1. This is according to Weiss' book.

Other BST Operations

- Find, insert, and delete are not all.
- Other operations (that rely on the values being sorted in the BST):
  - Find min.
  - Find max.
  - Print tree's values in sorted order in linear time. (in order traversal)
  - etc.
- Next slide deck: hash tables, which sacrifice some of these operations for speed.

References / Further Reading

- Additional topics that we will skip:
  - Red-black trees.
  - B trees.
- Chapter 7 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.
- *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss (Fourth Edition).
  - Chapter 4: AVL trees, splay trees, B trees.
  - Chapter 11: amortized analysis.
- *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition).
  - Chapter 13 is about red-black trees.
  - Chapter 17 is about amortized analysis.
- Chapter 10 of *Data Structures and Algorithms with Python* by Kent D. Lee and Steve Hubbard.
  - I believe this book is still legally, freely available.