Data Structures

CDT Tea

Outline

- Abstract Data Types
- Sorting
- Tree

Data Structures

- A container of stuff (data).
- Some things that you can do:
  - Add stuff to it
  - Remove stuff from it
  - Find specific stuff
  - Empty it

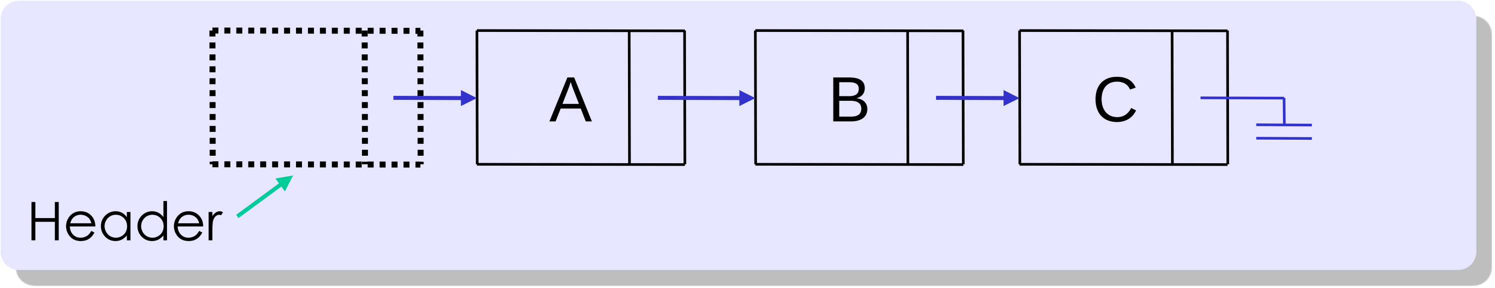
Container

Data

List

- A collection of items in which the items have a **position**.
- Can access any item by its **index**.
- Ex: array
  - Insertion/deletion is expensive.
  - Allows random access.

Linked List



- Random access is not allowed.
- Extra memory space.
- Ease of insertion/deletion.
- Variants: sorted, doubly-linked, circular

Stack

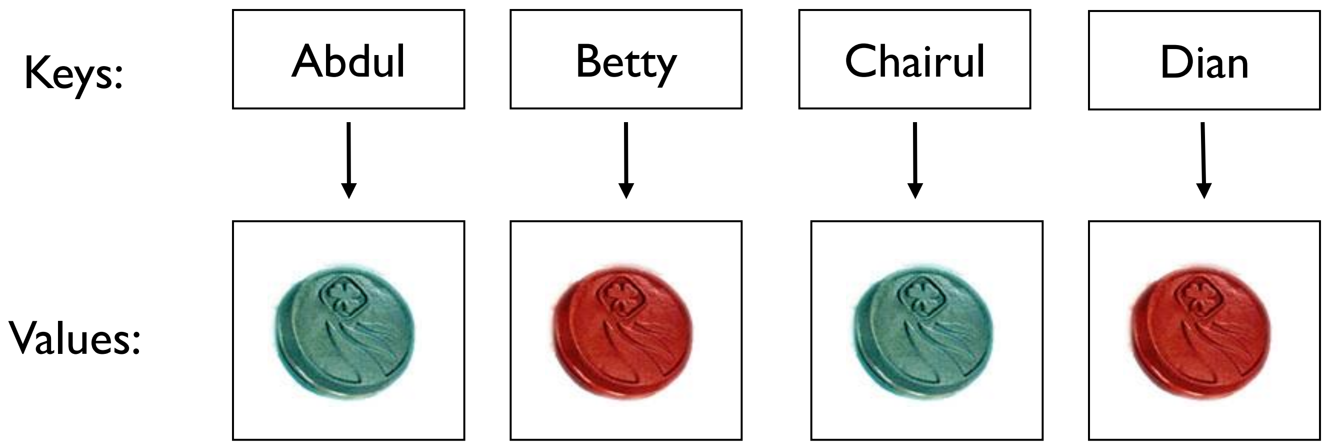
- Access is restricted to the **most recently inserted** item.
- Operations take a **constant** amount of time.

Queue

- Access is restricted to the **least recently inserted** item.
- Operations also take a **constant** amount of time.

Priority Queue

Maps

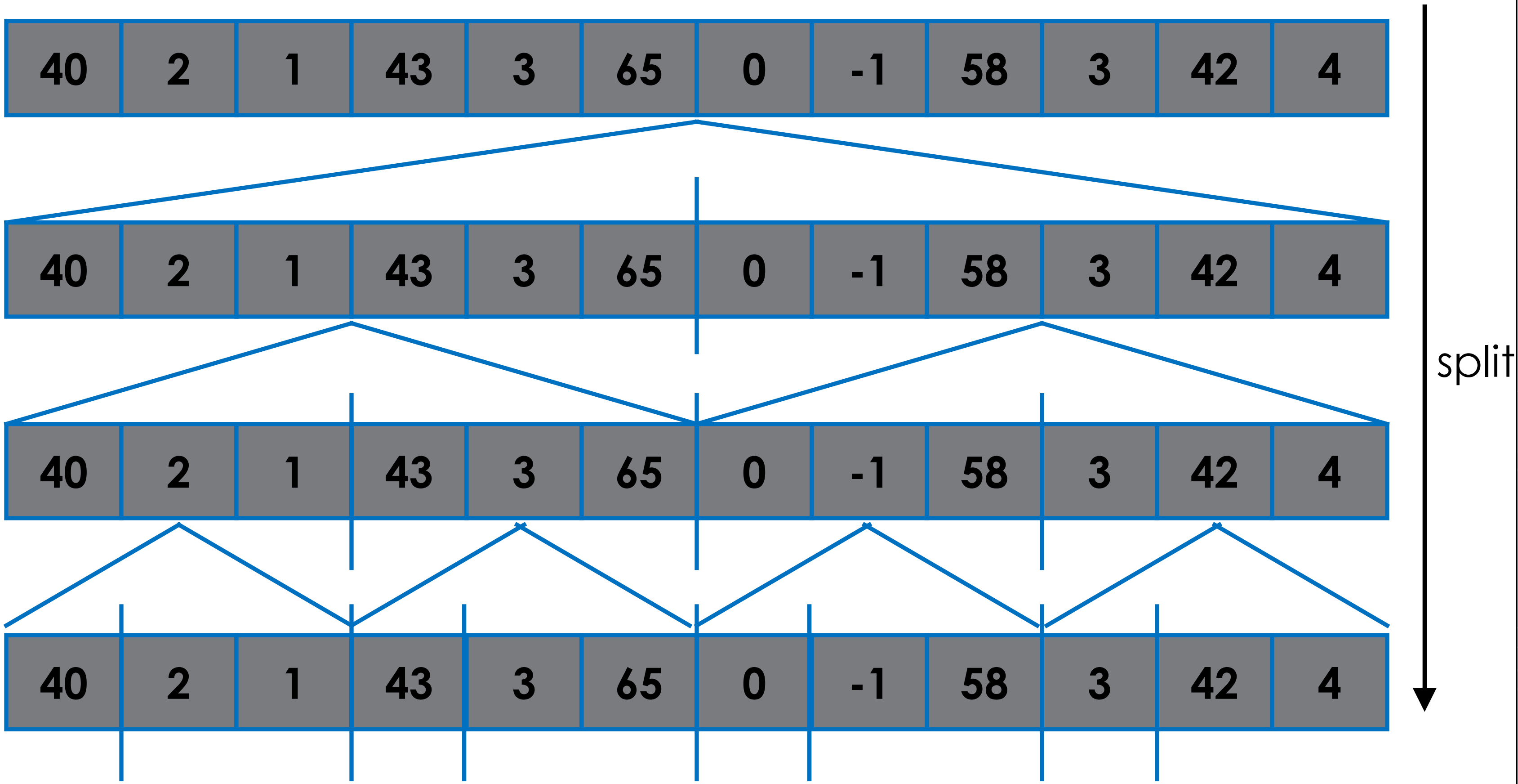


Sorting

Sorting Algorithm

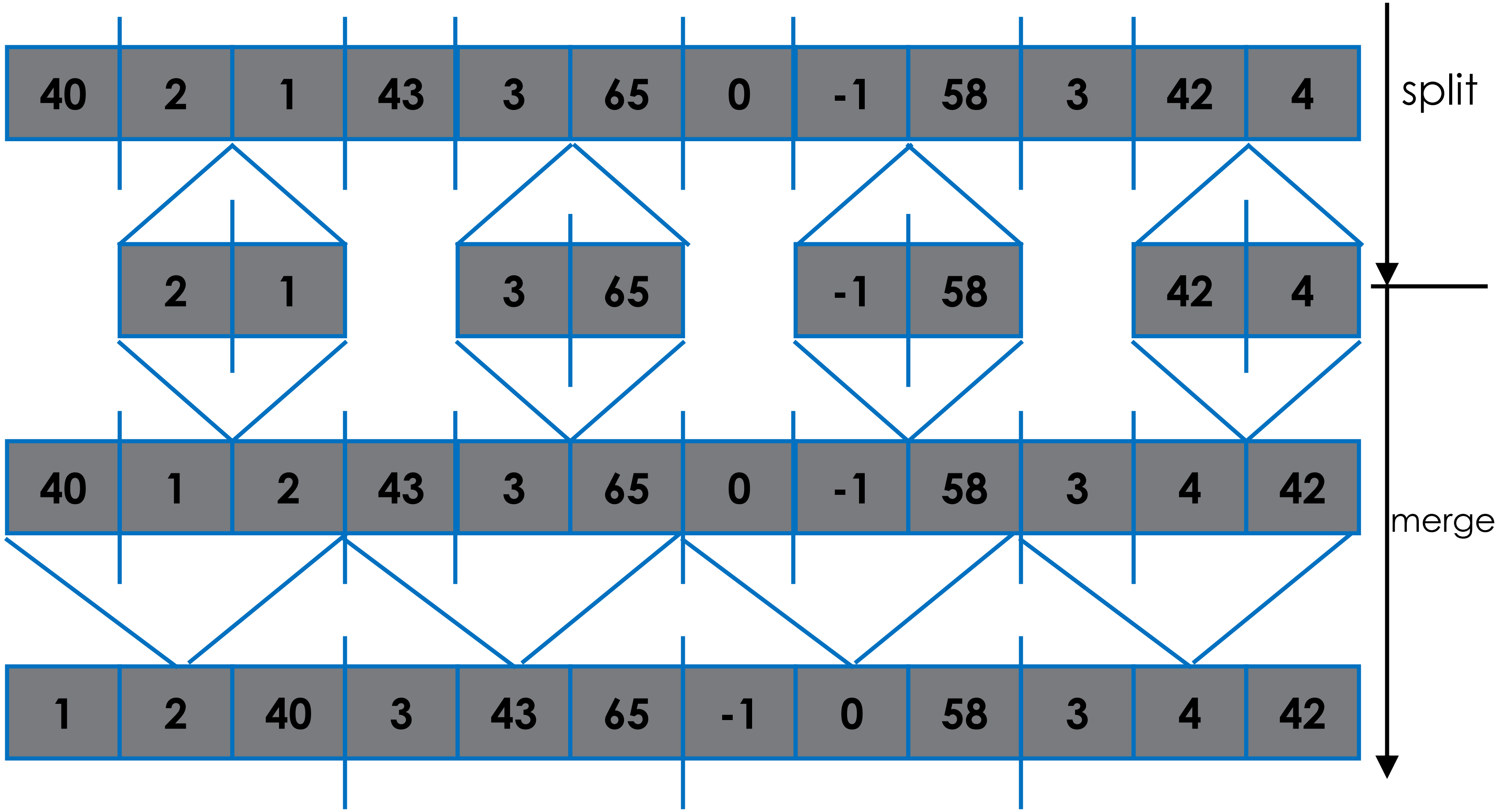
- Bubble sort, Insertion sort, Selection sort
  - Have worst case of $O(n^2)$
  - Exchanges adjacent items
    - Best worst case $\Omega(n^2)$ – **lower bound**!
- Merge sort & Quick sort
  - Divide & Conquer approach
  - Merge sort: $O(n \log n)$
  - Quick sort: $O(n \log n)$, $O(n^2)$

Merge Sort

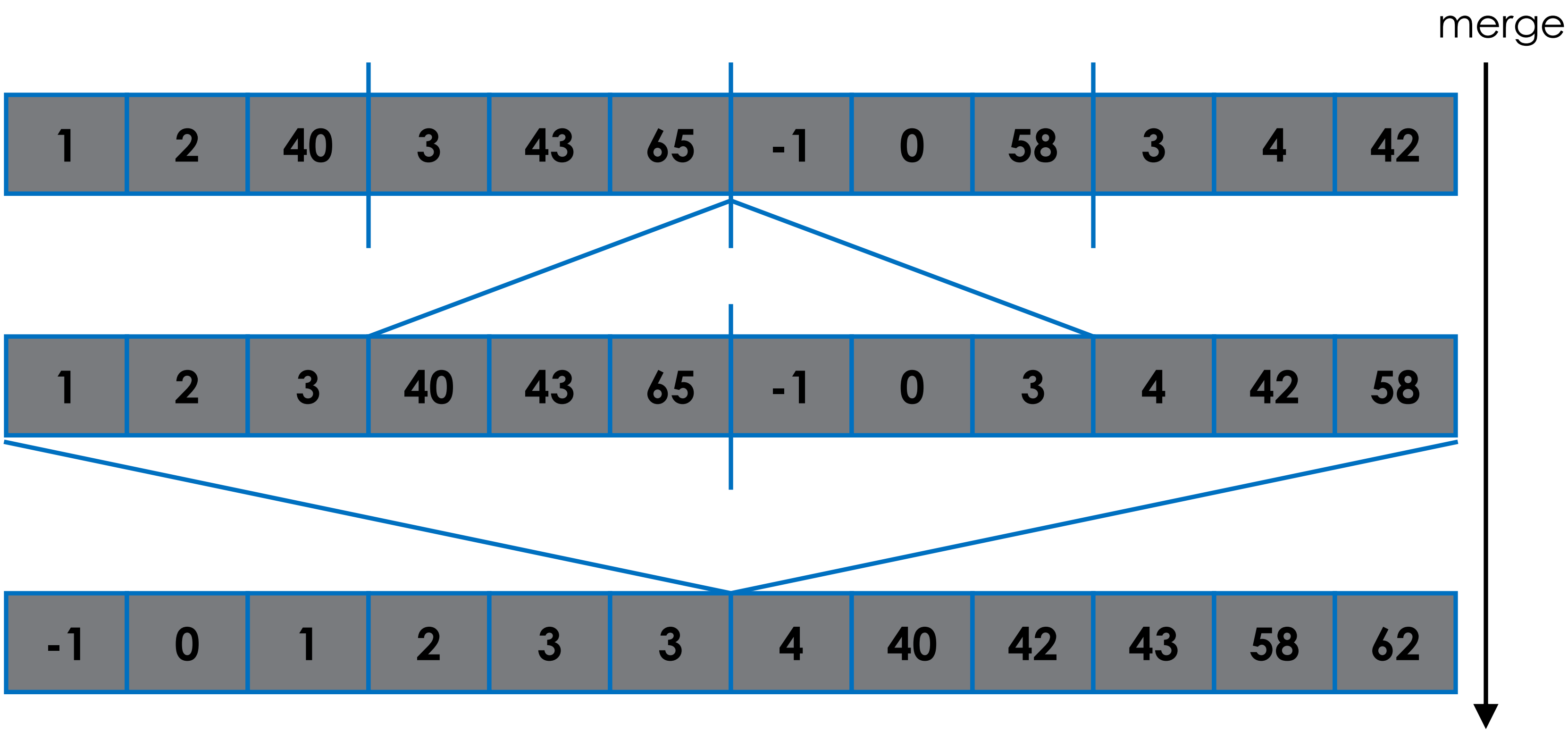


40 2 1 43 3 65 0 -1 58 3 42 4
40 2 1 43 3 65 0 -1 58 3 42 4
40 2 1 43 3 65 0 -1 58 3 42 4
40 2 1 43 3 65 0 -1 58 3 42 4
split

Merge Sort


40 2 1 43 3 65 0 -1 58 3 42 4
split
2 1
3 65
-1 58
42 4
40 1 2 43 3 65 0 -1 58 3 4 42
merge
1 2 40 3 43 65 -1 0 58 3 4 42

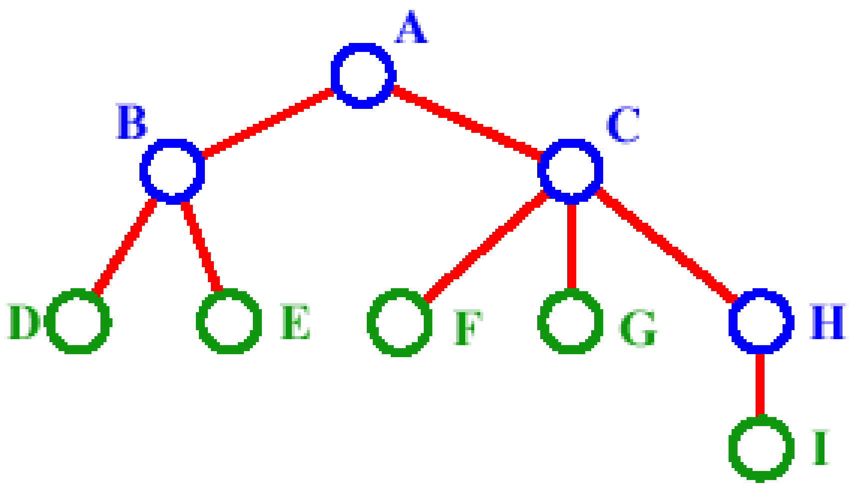
Merge Sort


merge
1 2 40 3 43 65 -1 0 58 3 4 42
1 2 3 40 43 65 -1 0 3 4 42 58
-1 0 1 2 3 3 4 40 42 43 58 62

Tree

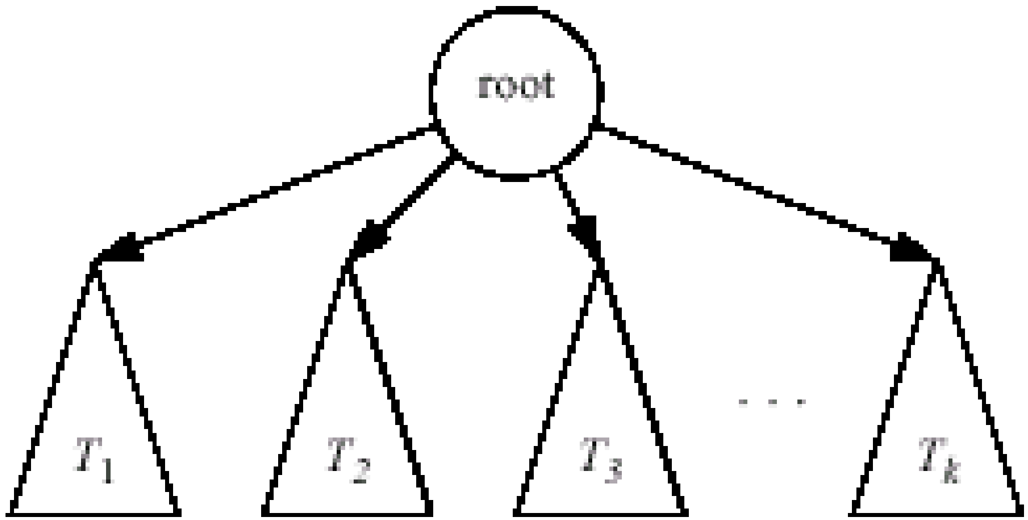
AVL Tree

Terminology



- *A* is the *root node*
- *B* is the *parent* of D and E
- C is the *sibling* of B
- *D* and *E* are the *children* of B
- *D, E, F, G, I* are *external nodes*, or *leaves*
- *A, B, C, H* are *internal nodes*
- The *depth*, *level*, or *path length* of *E* is 2
- The *height* of the tree is 3
- The *degree* of node *B* is 2

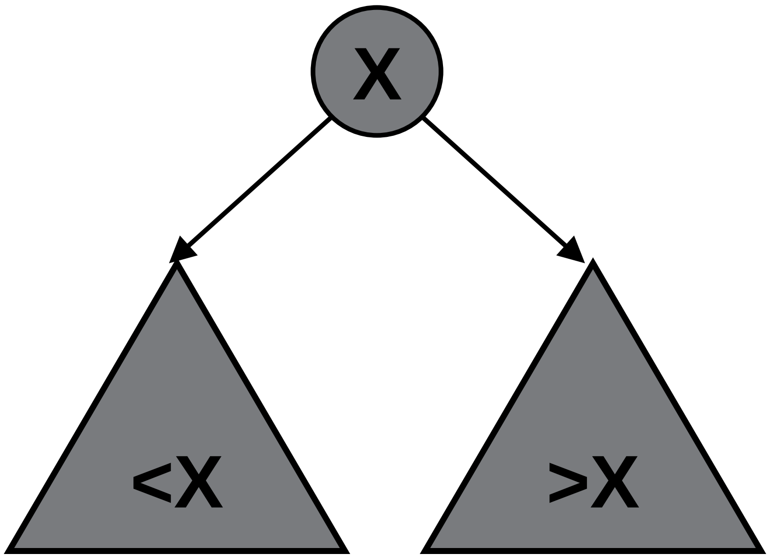
Property: |*edges*| = |*nodes*| - 1



A sub-tree is also a tree

Binary Search Tree

- Elements have keys (no duplicates allowed).
- For every node X in the tree, the values of all the keys in the left subtree are smaller than the key in X and the values of all the keys in the right subtree are larger than the key in X.
- The keys must be comparable.

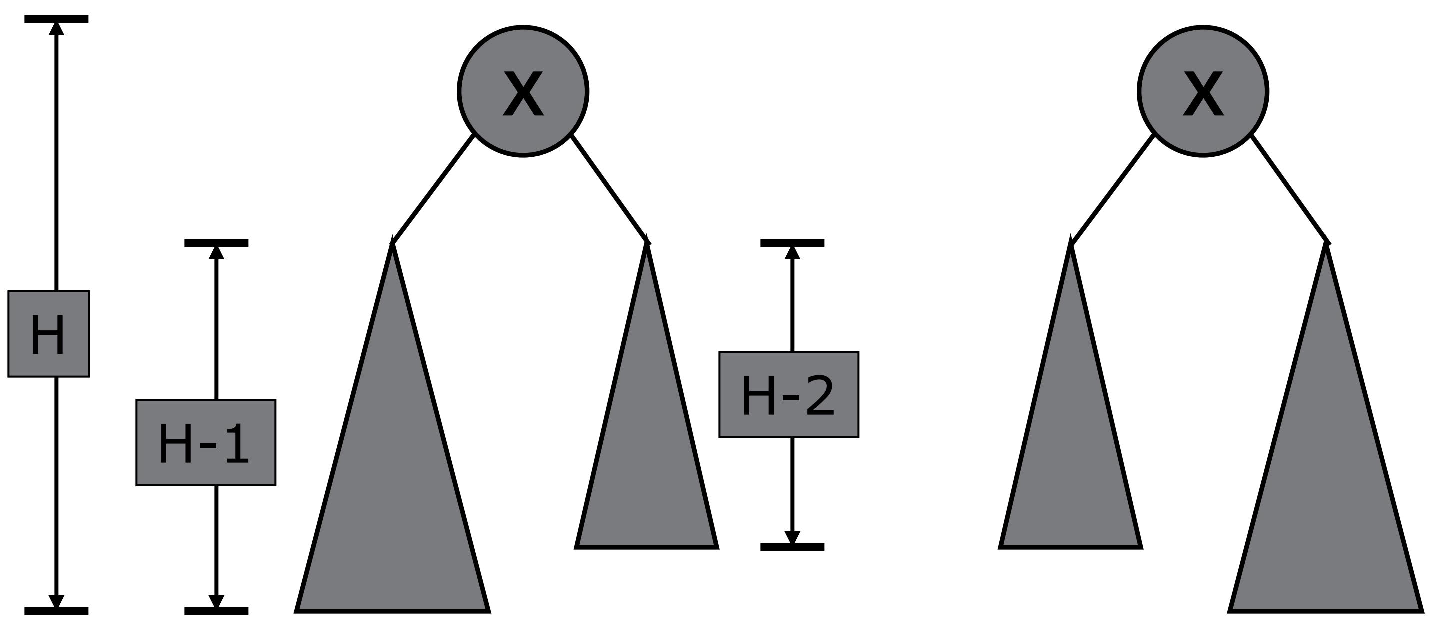


Binary Search Tree

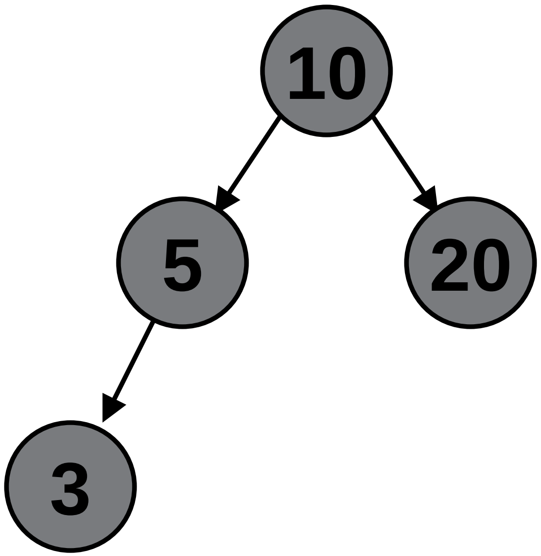
- Running time for:
  - Insert
  - Find min
  - Remove
  - Find
- Average case: O(log n) – equally balanced
- Worst case: O(n) – the height of the tree equals the number of nodes

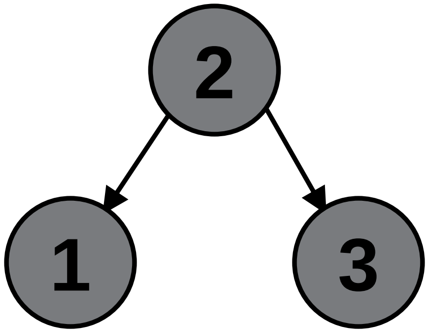
AVL Trees

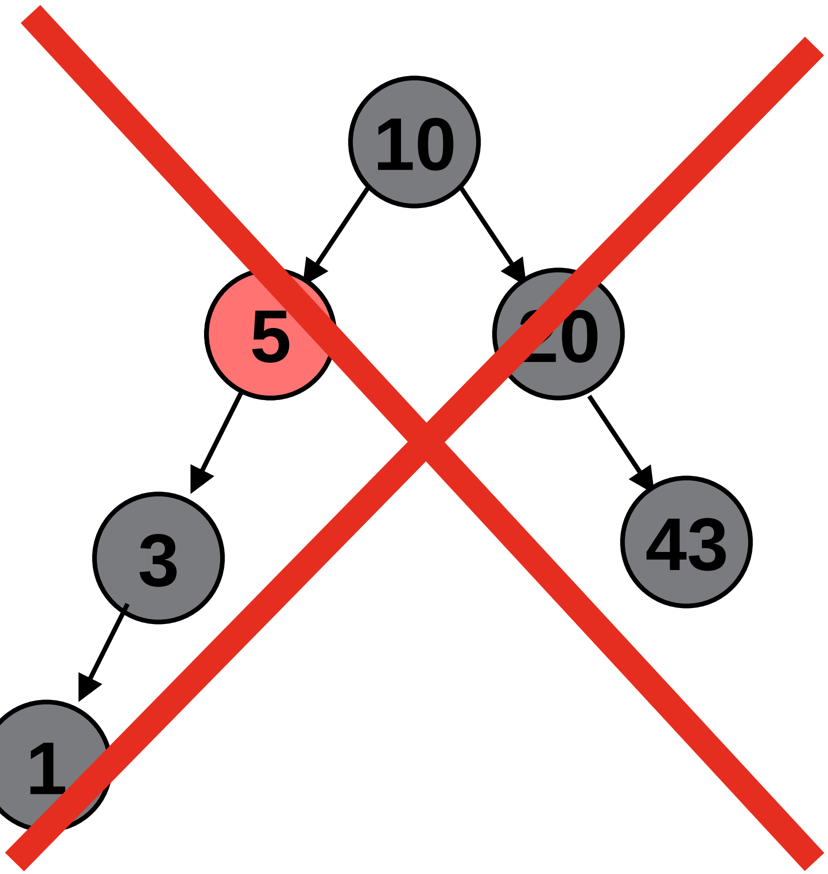
- AVL (**Adelson-Velskii & Landis**) trees maintain balance.
- For each node in tree, height of left subtree and height of right subtree differ by a maximum of 1.



AVL Trees



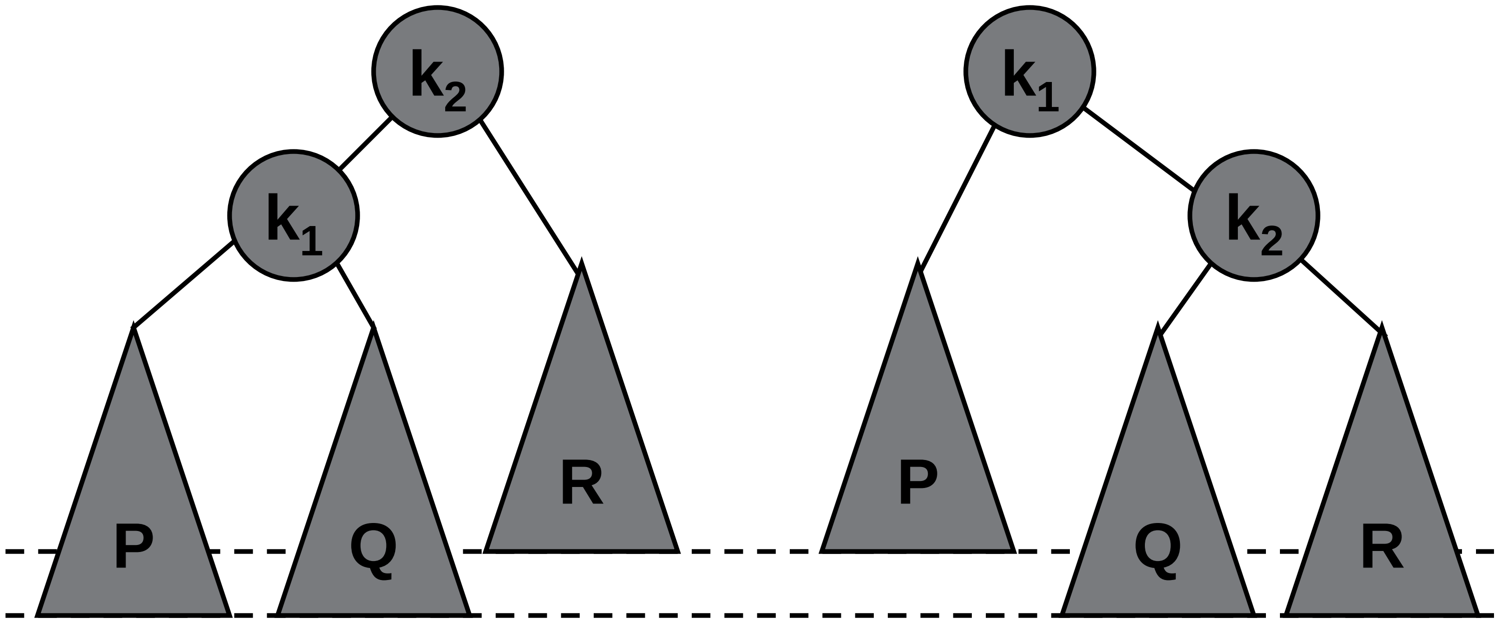




Insertion

- To ensure balance condition for AVL-tree, after insertion of a new node, we back up the path from the inserted node to root and check the balance condition for each node.
- If after insertion, the balance condition does not hold in a certain node, we do one of the following rotations:
  - Single rotation
  - Double rotation

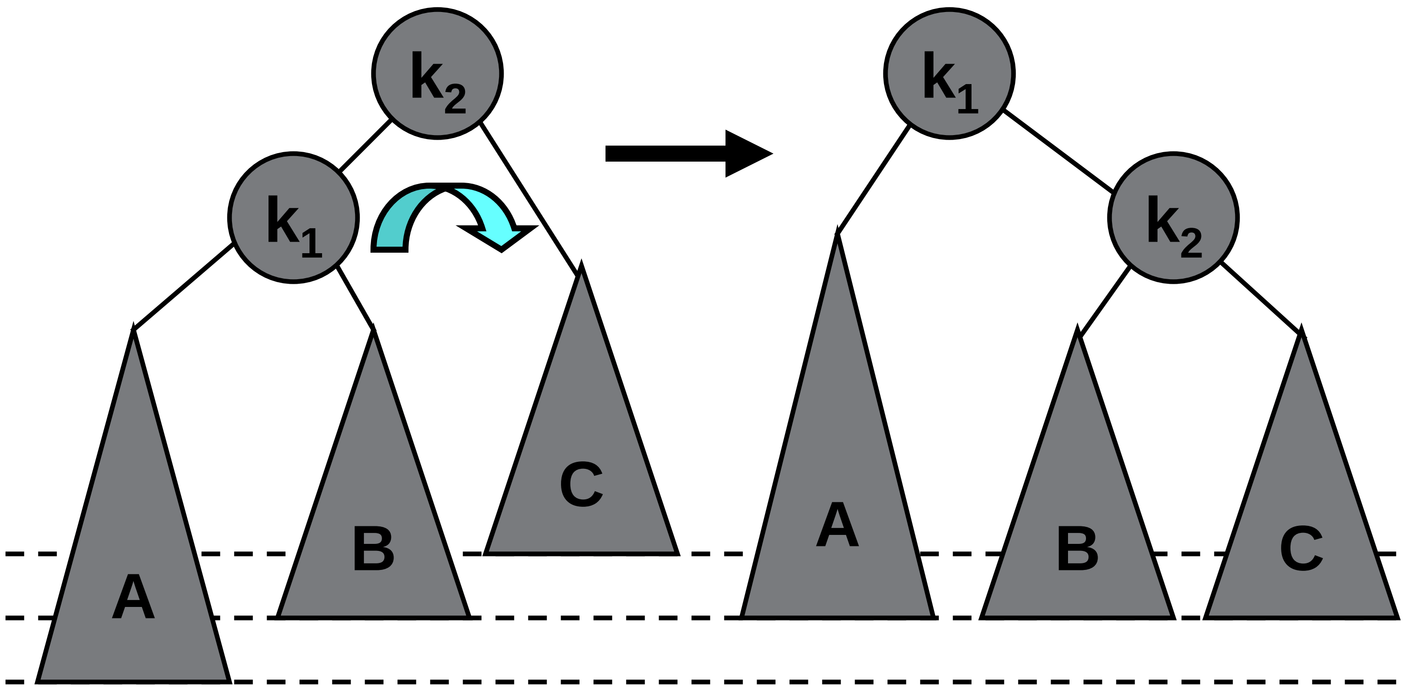
Insertion



- An insertion into the subtree:
  - P (outside) - case 1
  - Q (inside) - case 2
- An insertion into the subtree:
  - Q (inside) - case 3
  - R (outside) - case 4

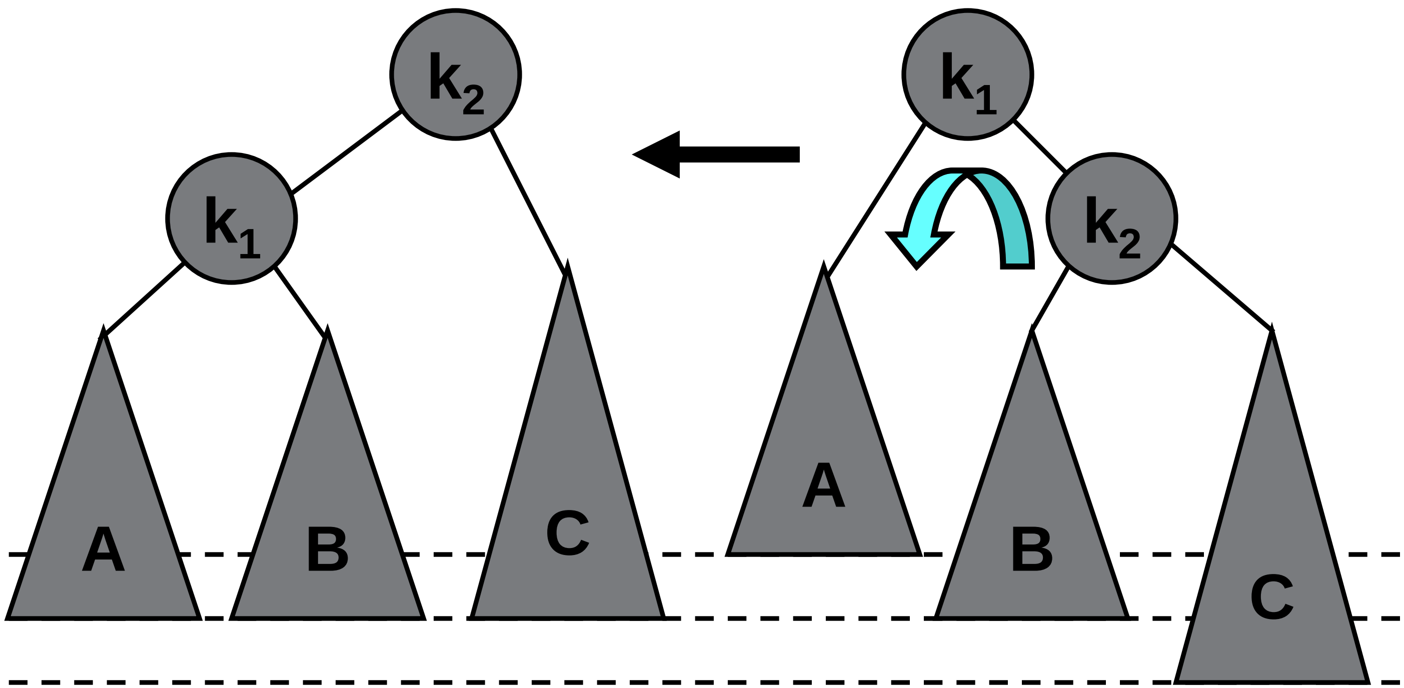
Single Rotation (case 1)

$H_A = H_B + 1$
$H_B = H_C$



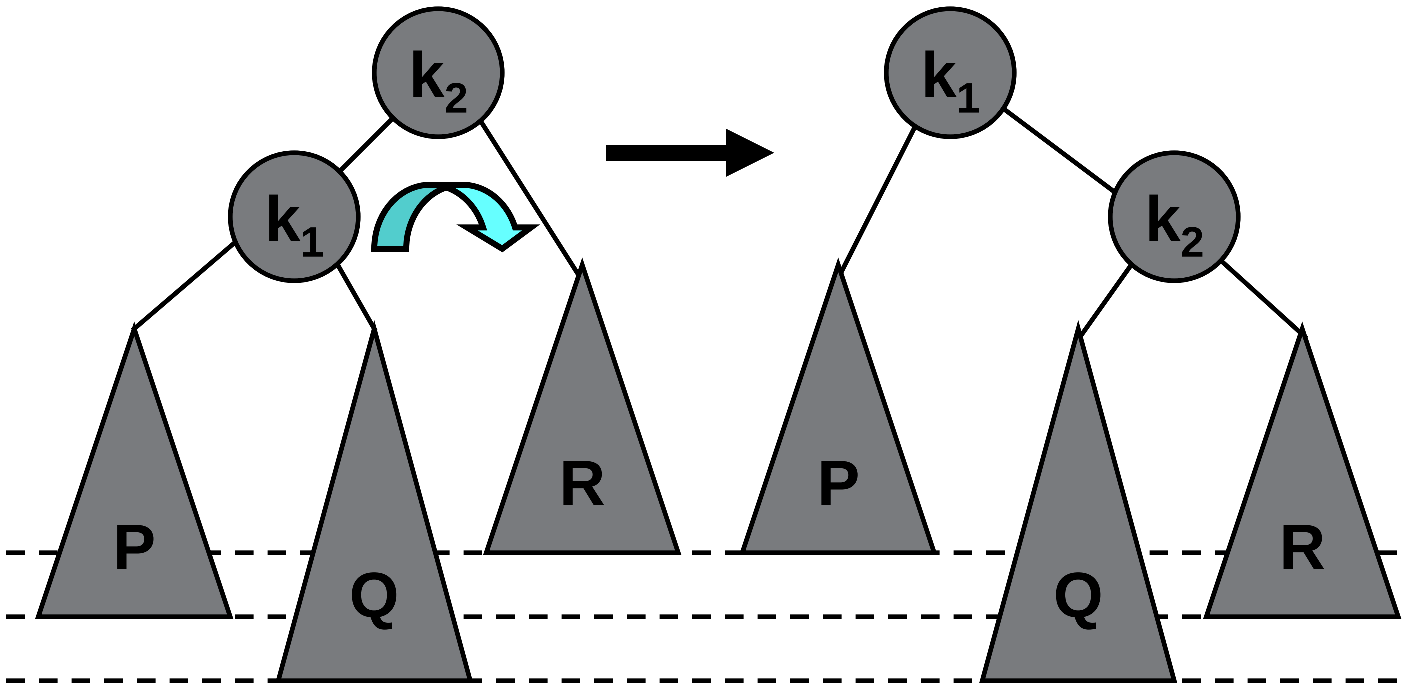
Single Rotation (case 4)

$H_A = H_B$
$H_C = H_B + 1$



Problem

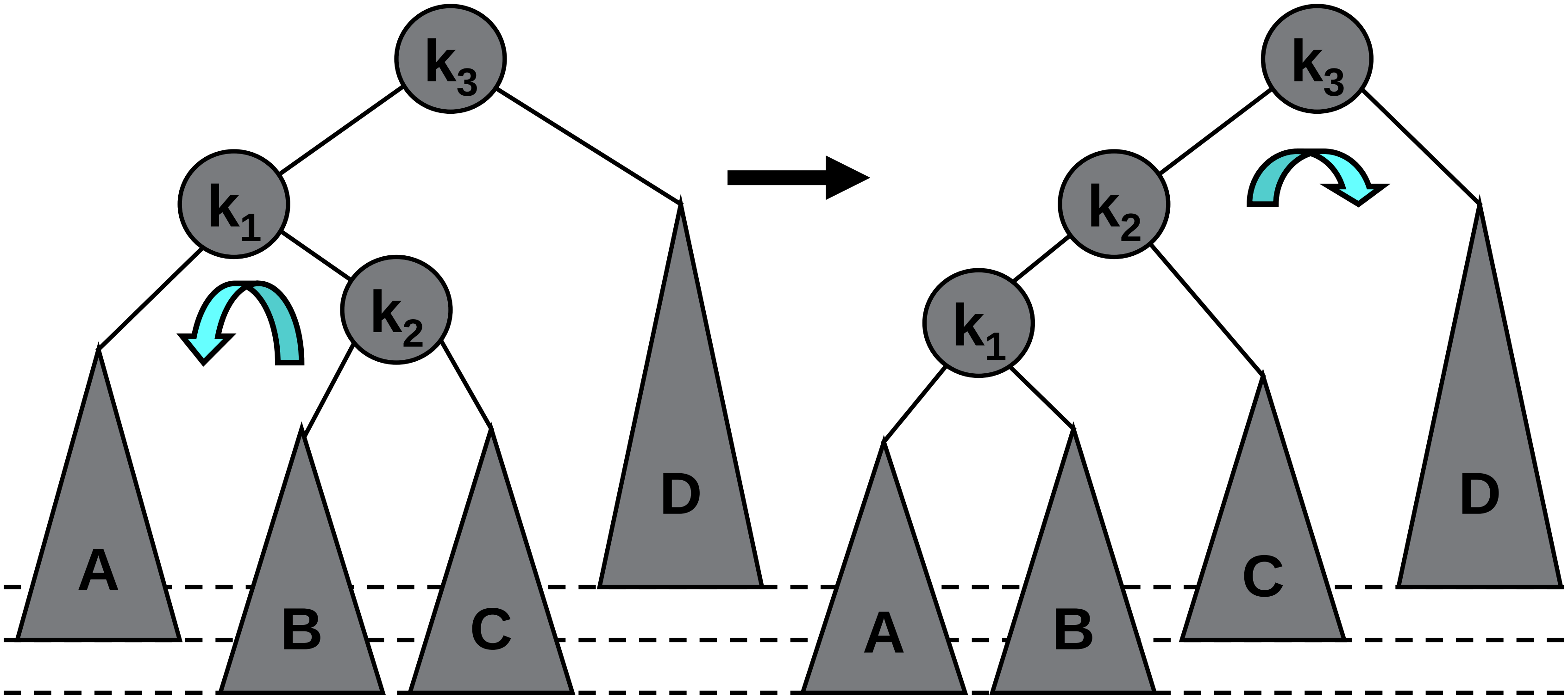
- Single rotation does not work for case 2 and 3 (*inside case*)



$H_Q = H_P + 1$

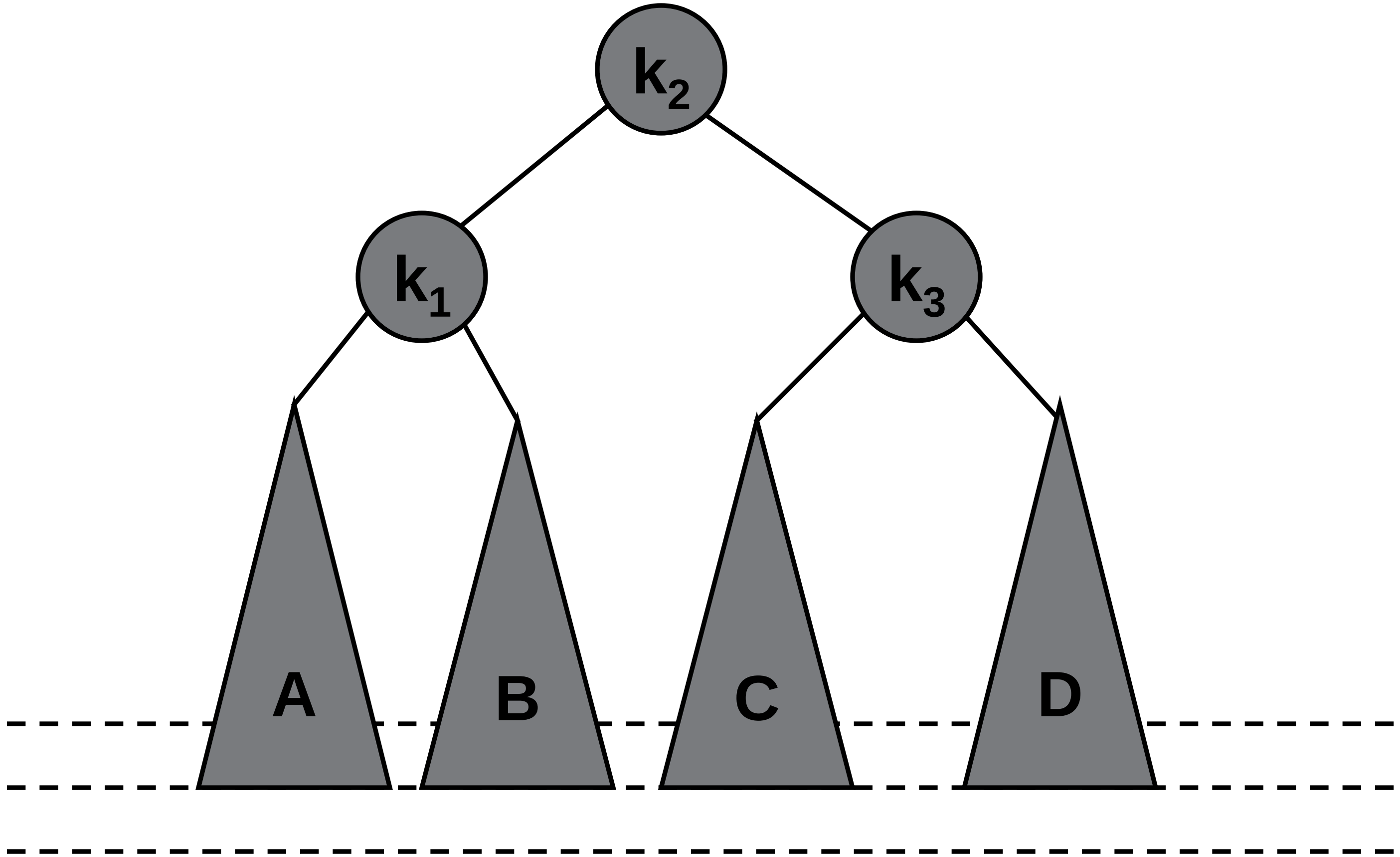
$H_P = H_R$

Double Rotation



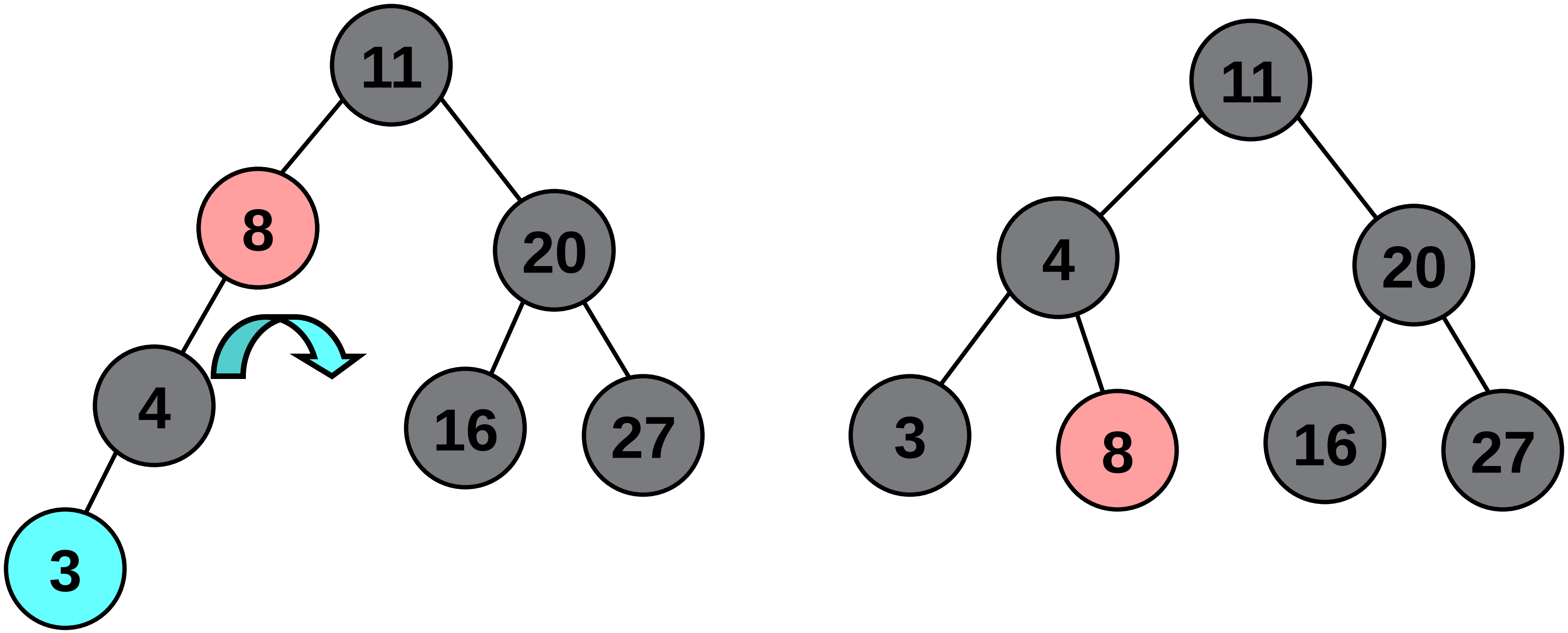
$H_A = H_B = H_C = H_D$

Double Rotation



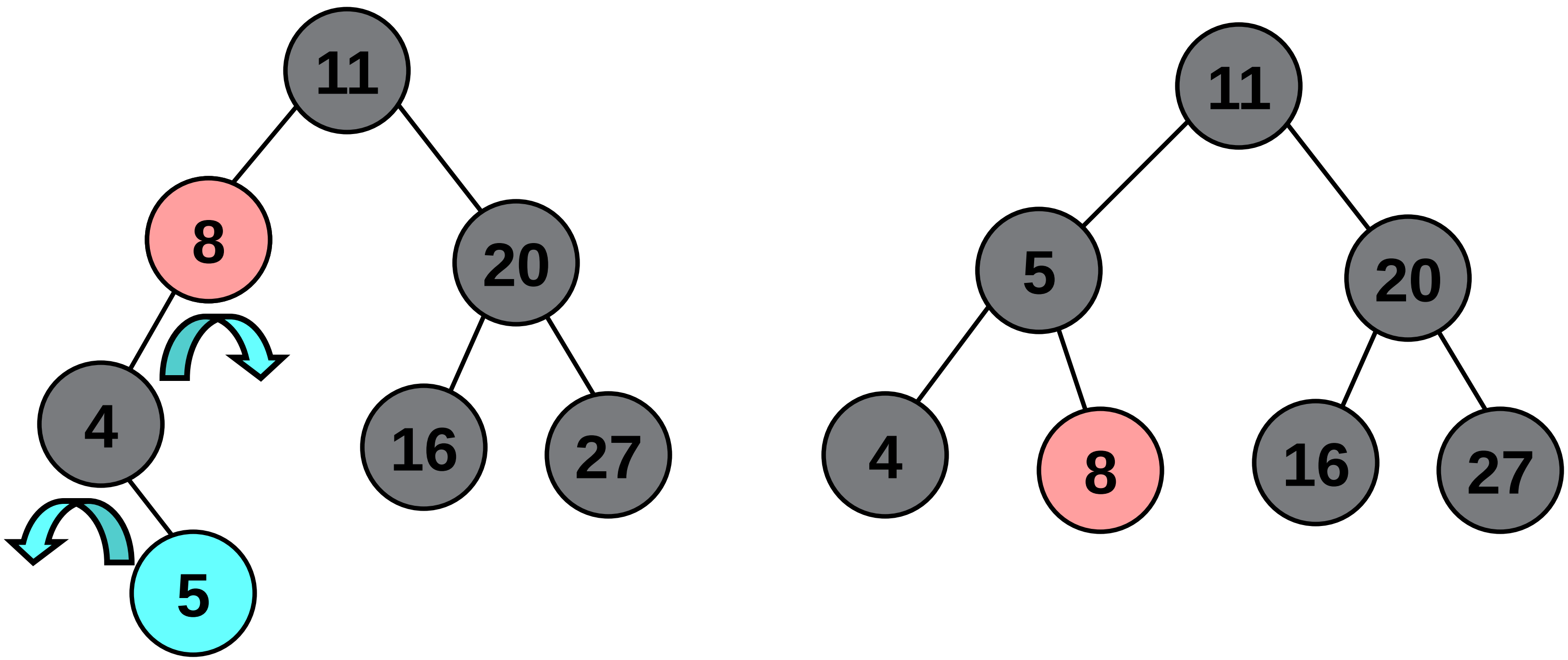
Example

- Insert 3 into the AVL tree



Example

- Insert 5 into the AVL tree



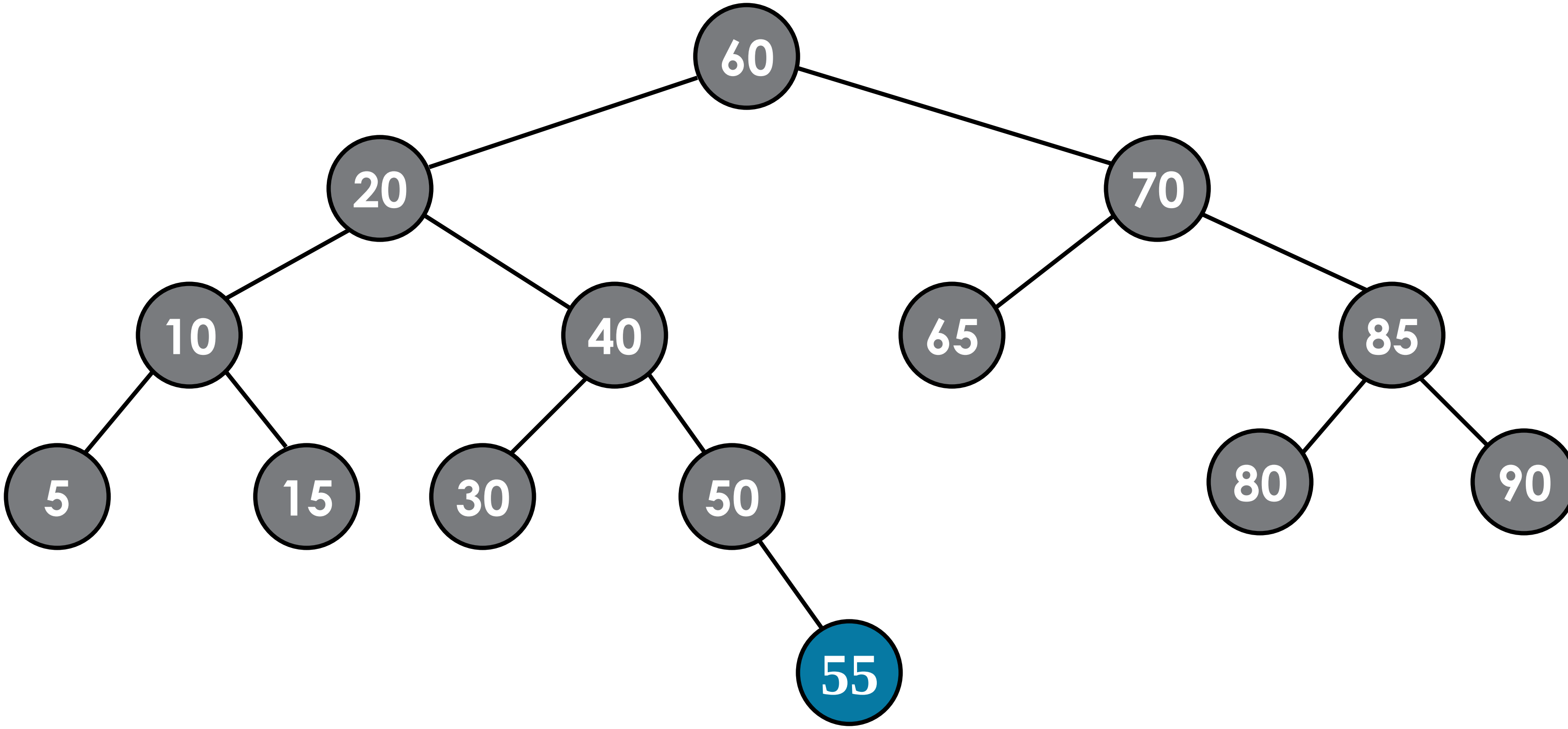
Deletion

- Removing a node from an AVL Tree is the same as removing from a binary search tree. However, it may unbalance the tree.
- Similar to insertion, starting from the removed node we check all the nodes in the path up to the root for the first unbalance node.
- Use the appropriate single or double rotation to balance the tree.
- May need to continue searching for unbalanced nodes all the way to the root.

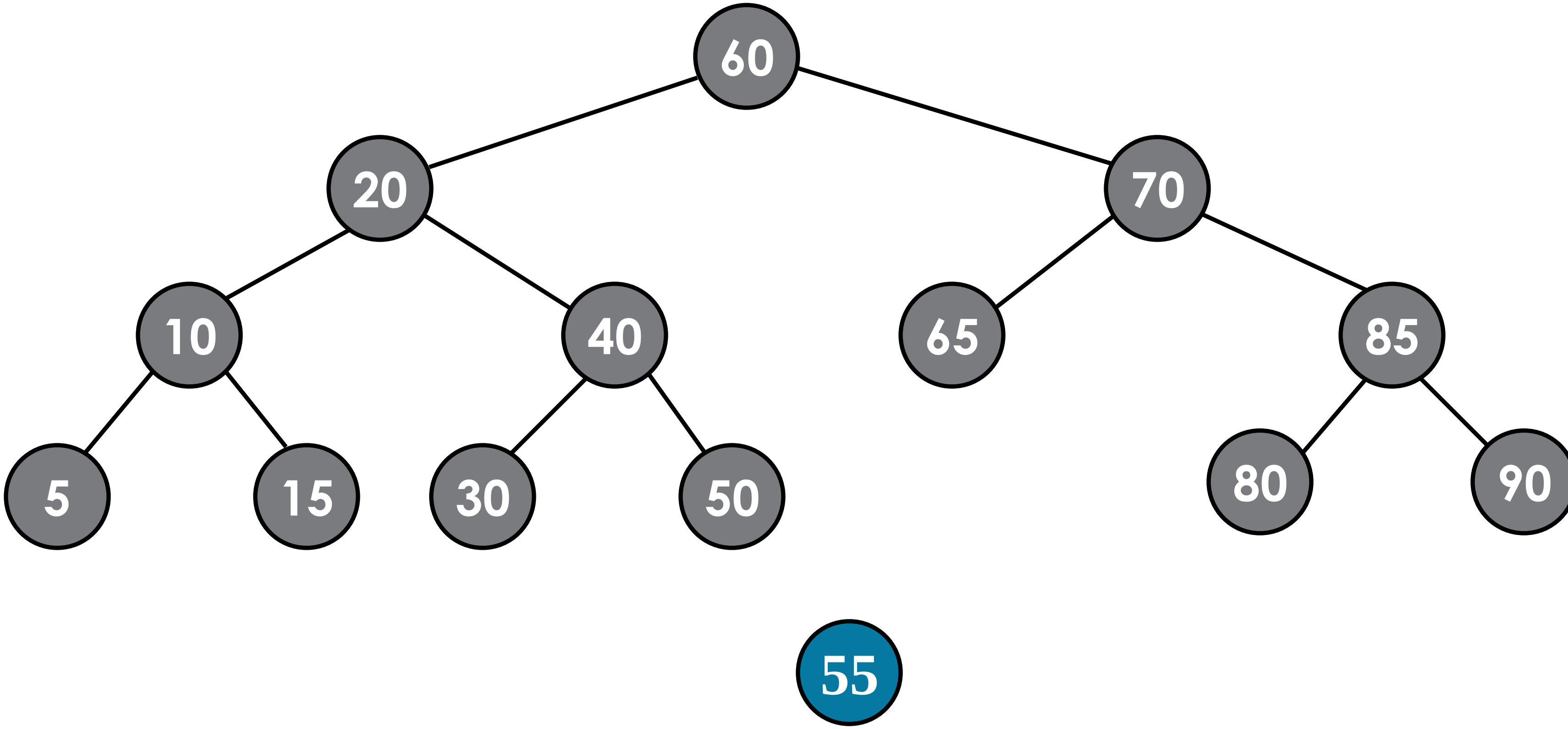
Deletion

- Deletion:
  - Case 1: if X is a leaf, delete X
  - Case 2: if X has 1 child, use it to replace X
  - Case 3: if X has 2 children, replace X with its inorder predecessor (and recursively delete it)
- Rebalancing

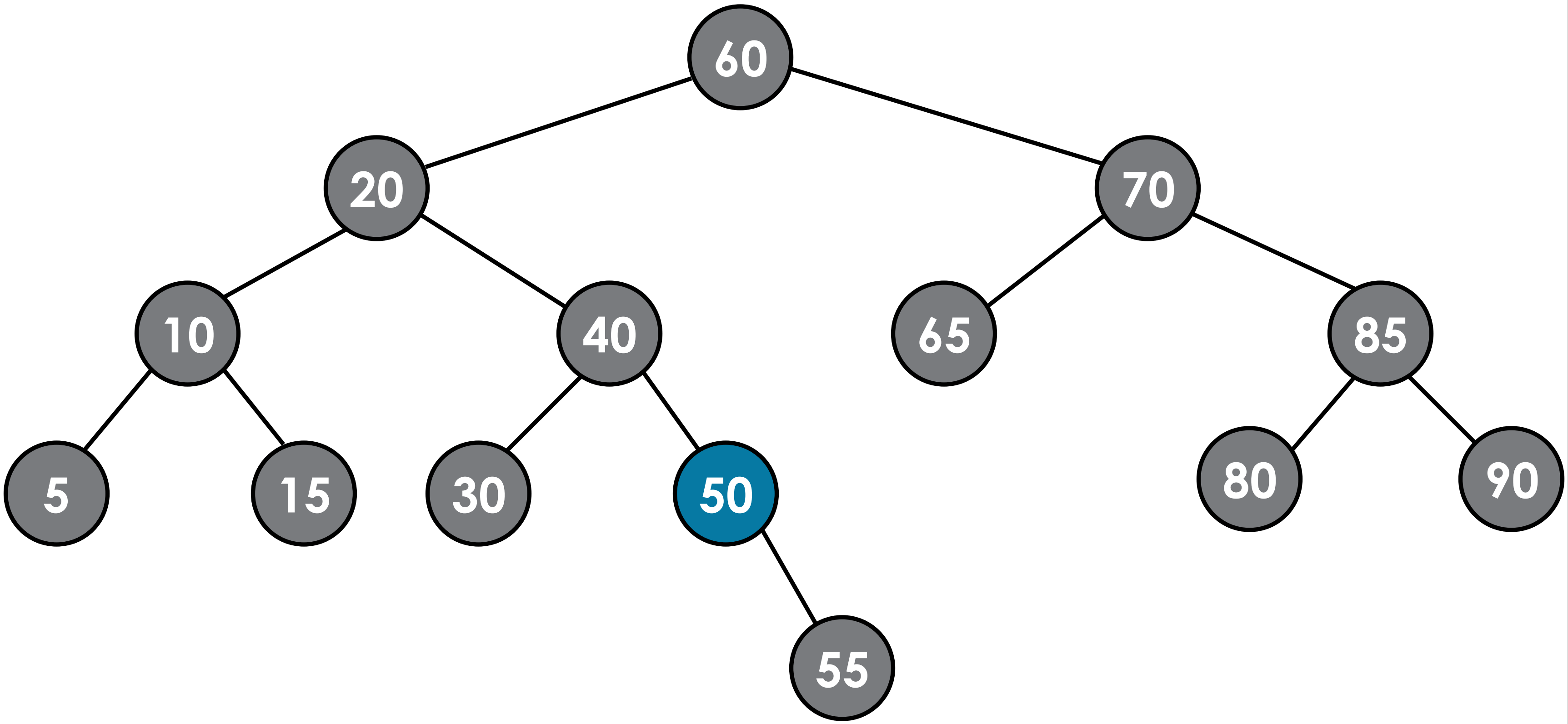
Delete 55 (case 1)



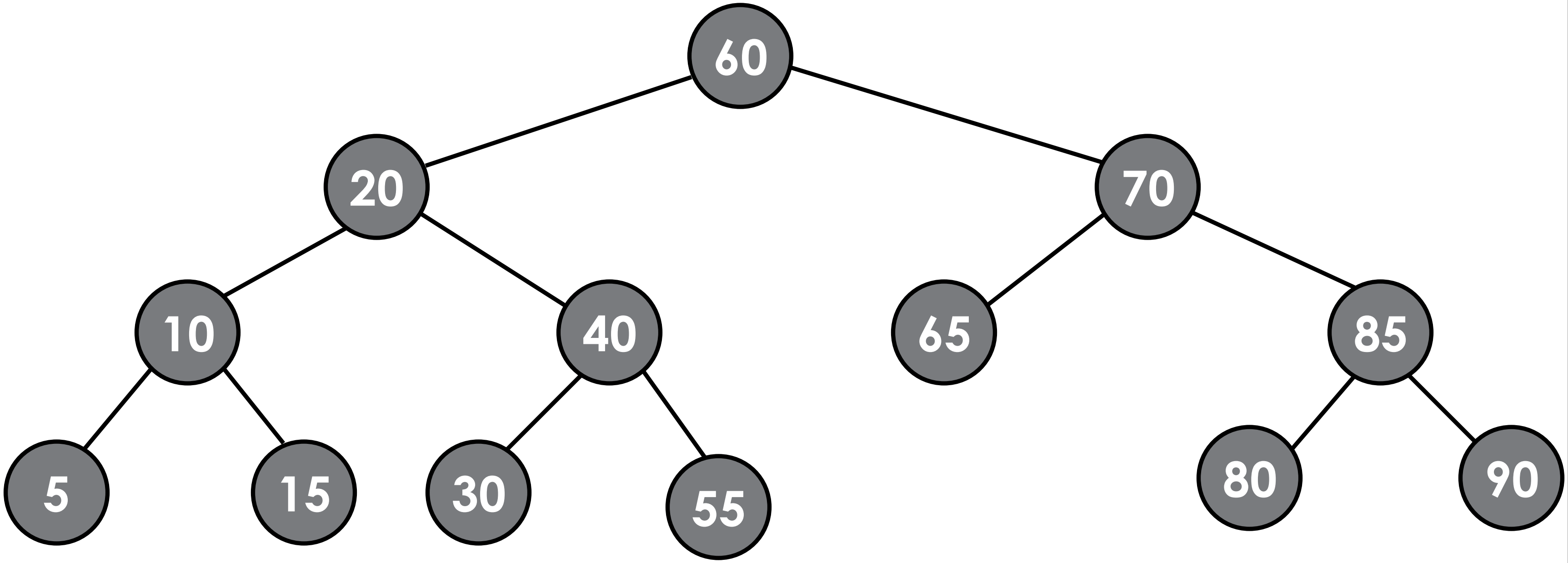
Delete 55 (case 1)



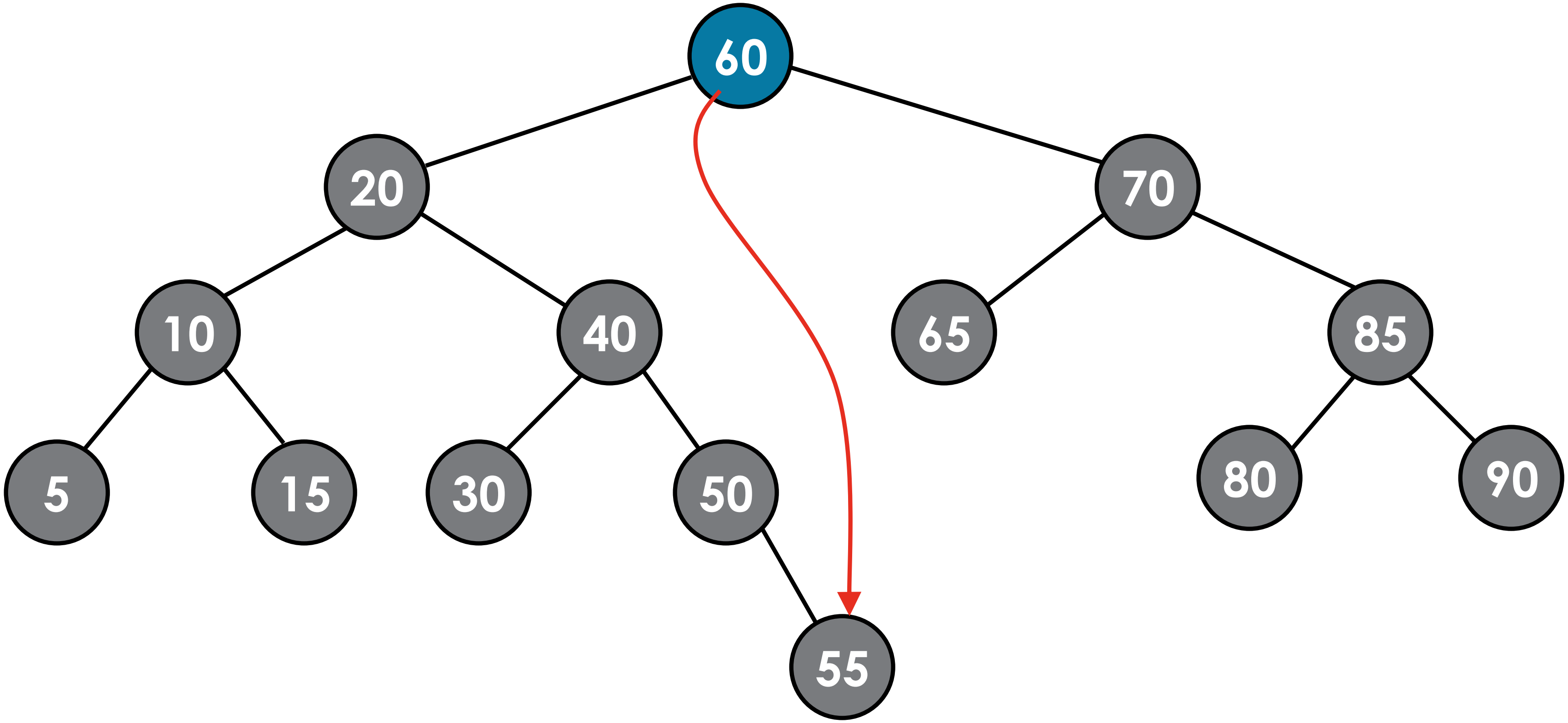
Delete 50 (case 2)



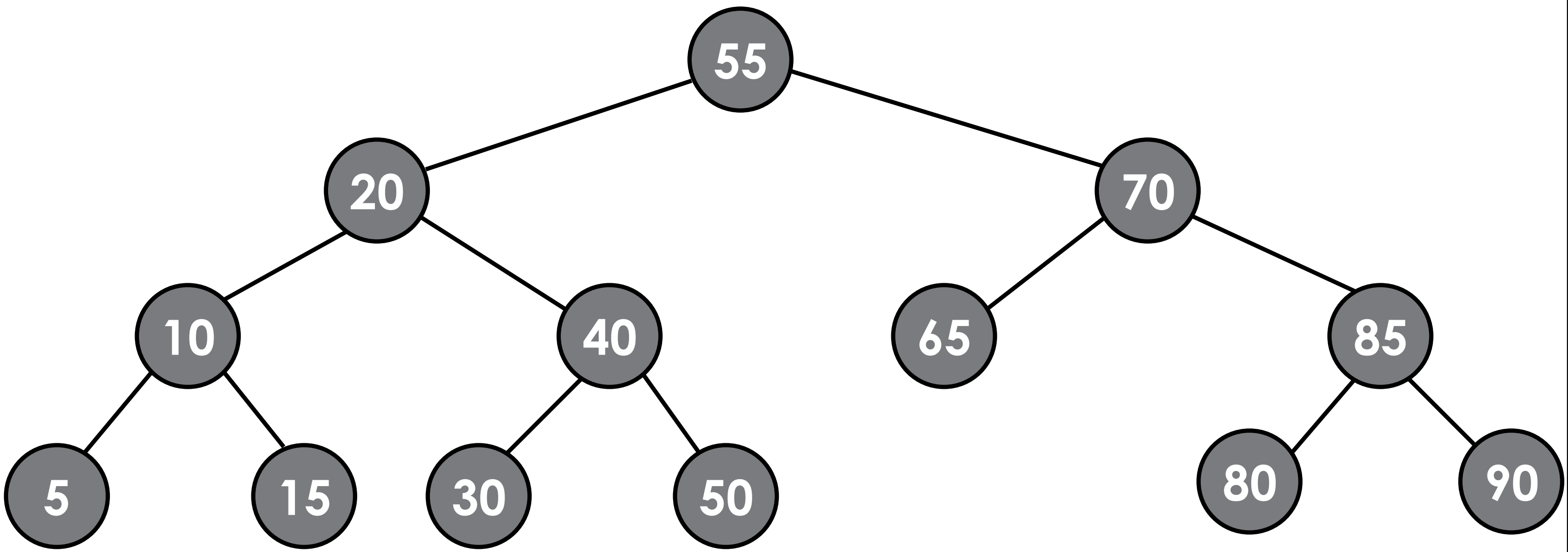
Delete 50 (case 2)



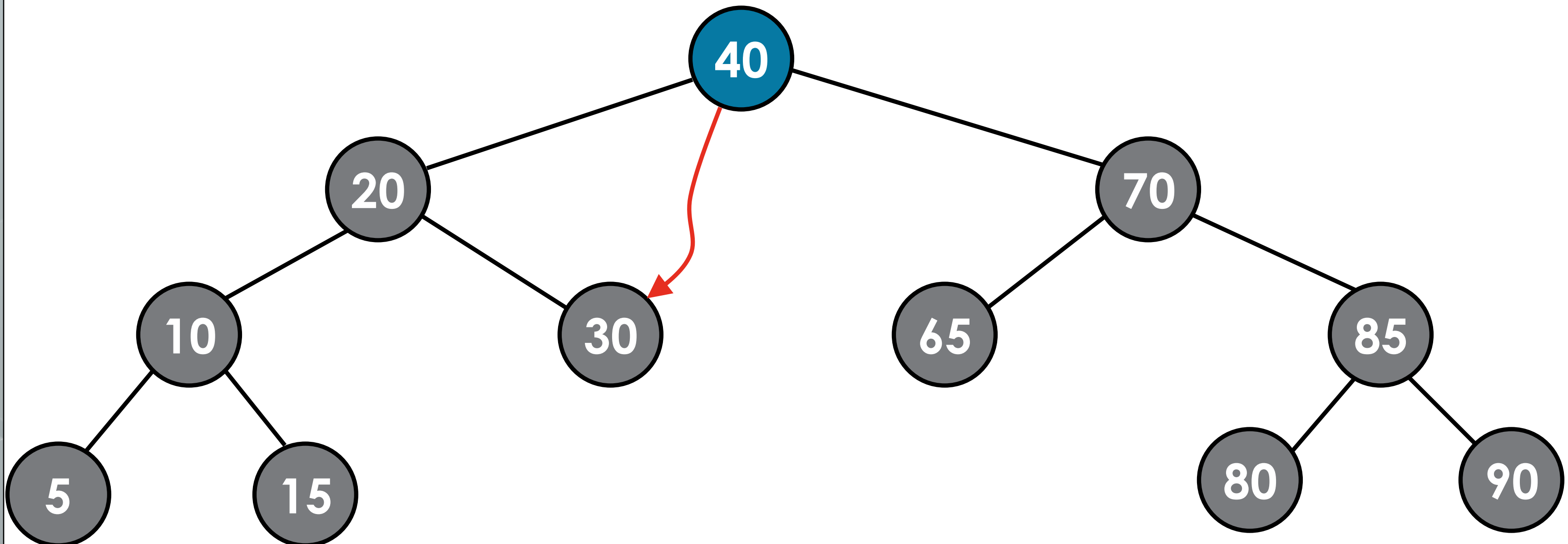
Delete 60 (case 3)



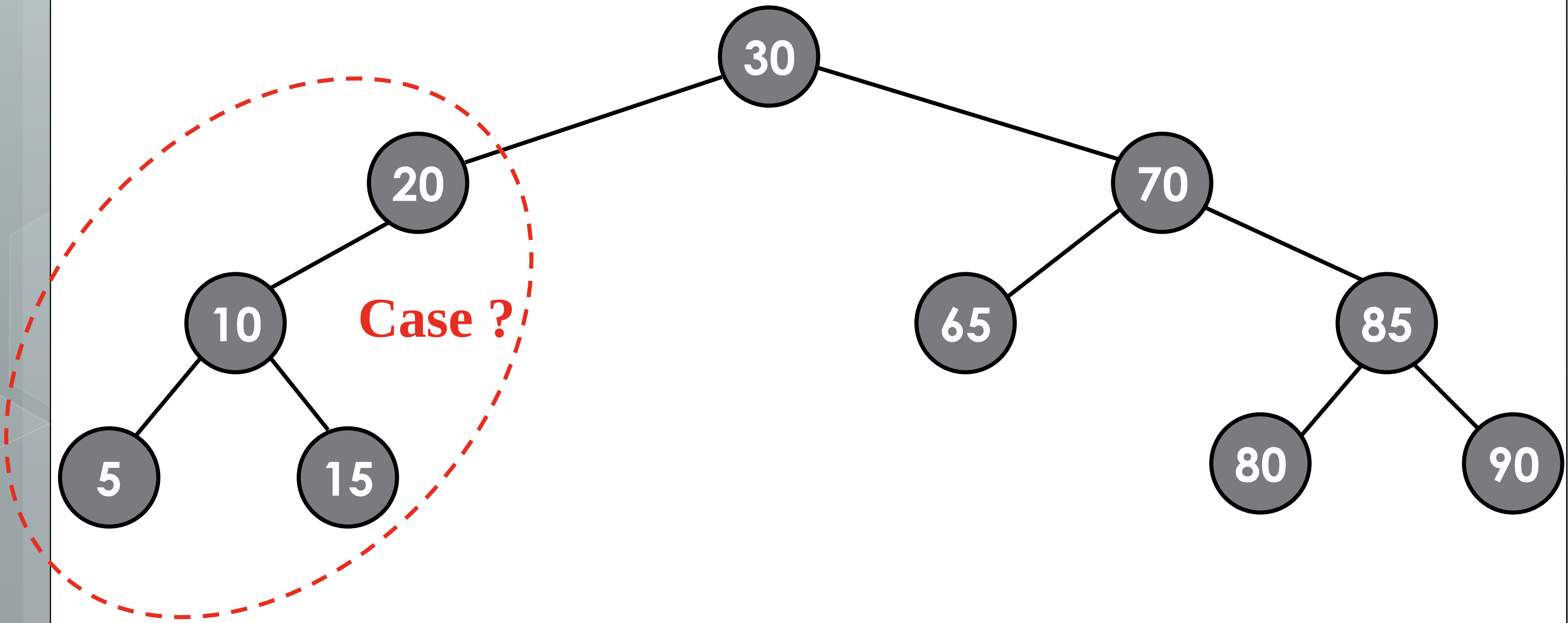
Delete 60 (case 3)



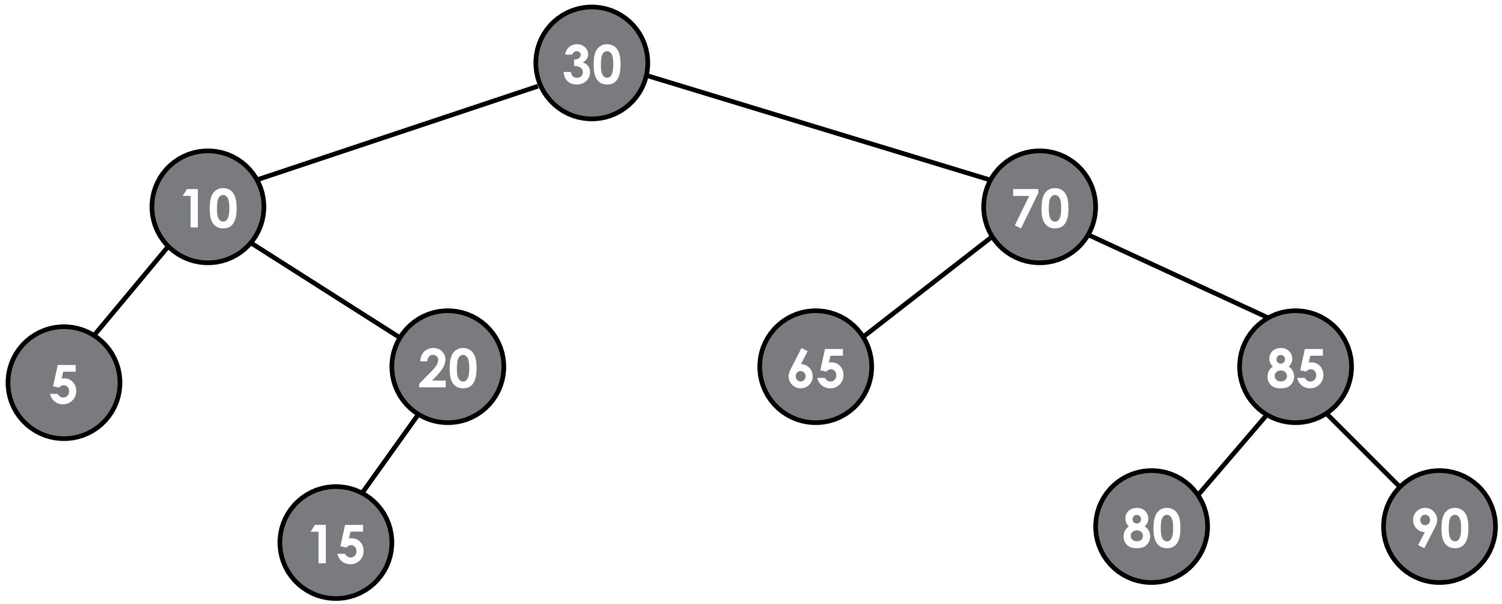
Delete 40 (case 3)



Delete 40 : Rebalancing



Delete 40: after rebalancing



Single rotation is preferred!

Analysis

- The depth of AVL Trees is at most logarithmic.
- So, all of the operations on AVL trees are also logarithmic.
- Find element, insert element, and remove element operations all have complexity O(log n) for worst case

Thank you