Self-Balancing Trees

Thomas Schwarz

Self-Balancing Trees

- Binary search trees are unbalanced
- Heaps are ideally balanced but do not support searches
- Self-balancing trees:
  - Create search trees that are almost balanced
  - Fundamental Idea:
    - When a tree becomes too unbalanced after insertion or deletion
      - Restructure in a very limited way

AVL Trees

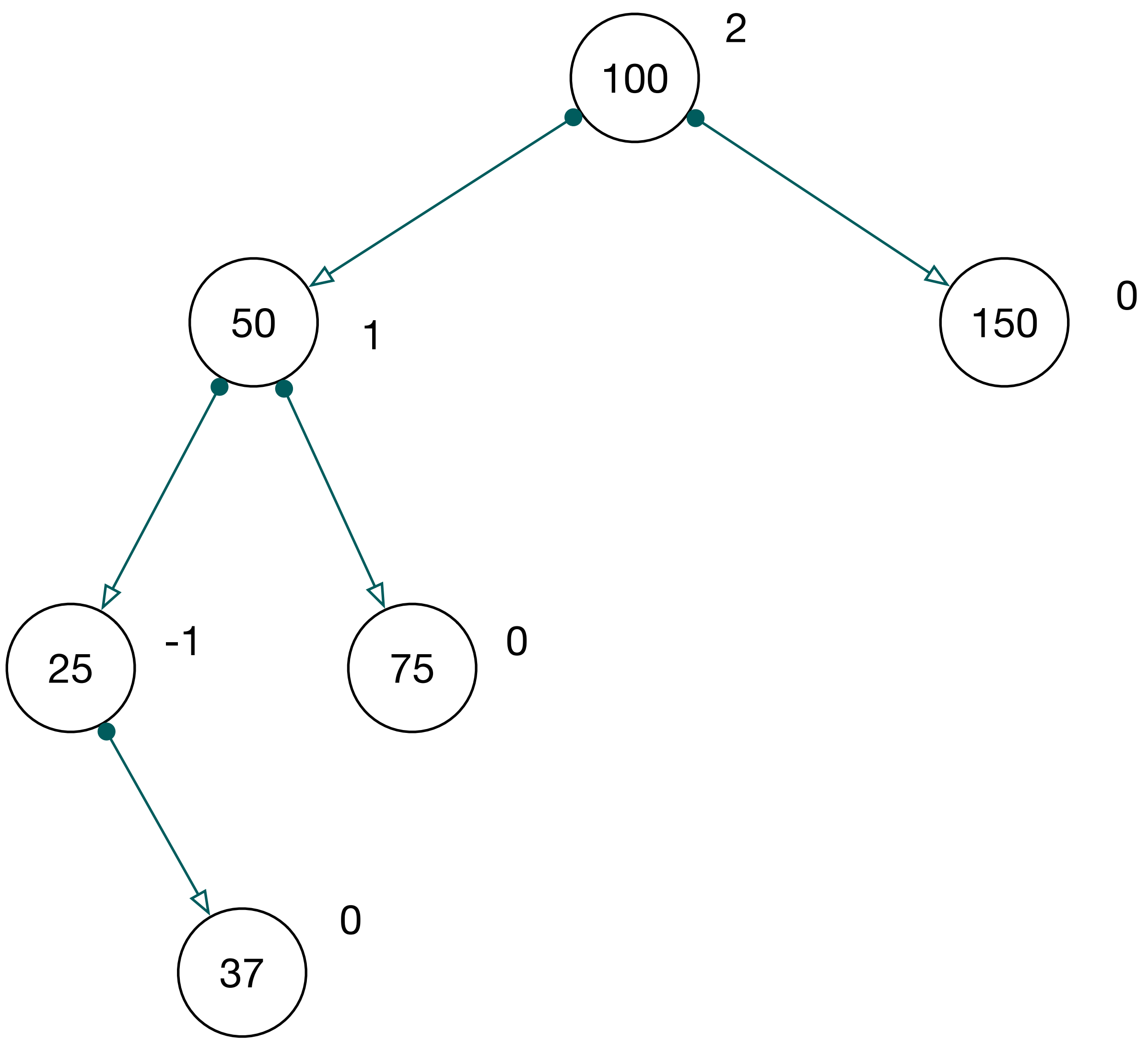
Thomas Schwarz

AVL Trees

- Georgy Adelson-Velsky & Evgenii Landis 1962
- First self-balancing binary search tree
  - For all nodes: Define a balance factor:
    - Height : Maximum of depth of leaves
    - Height of left sub-tree minus height of right sub-tree
      - Empty tree has height 0

AVL Trees

- Example for balancing
  - Heights
    - 100: 3
    - 50: 2
    - 25: 1
    - 75,37,150: 0



AVL Trees

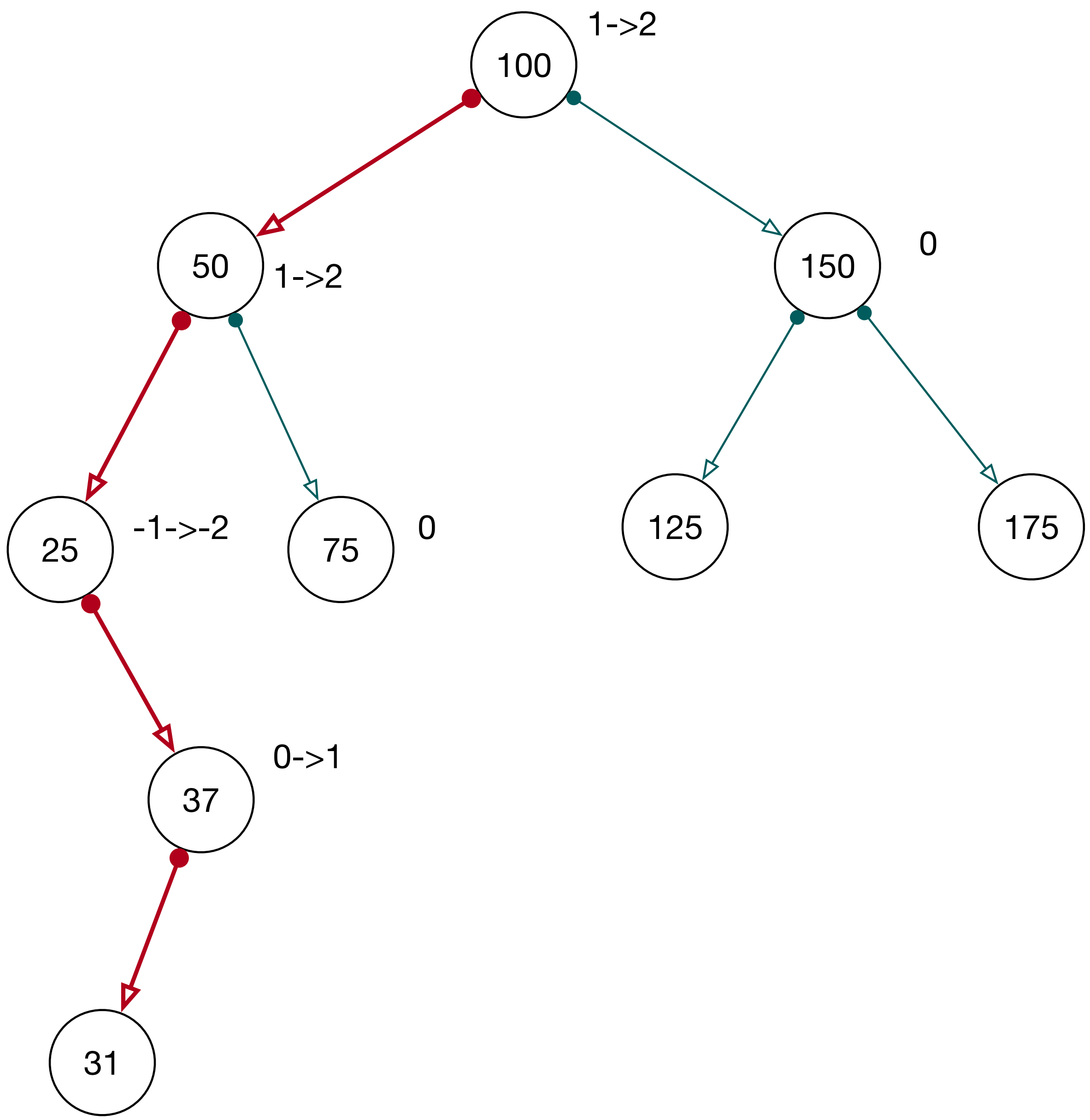
- AVL insight:
  - Keeping all balances equal to zero is impossible
    - But we can keep them in $\{-1,0,1\}$.
  - We do so by special operations on the nodes that have become unbalanced

AVL Trees

- AVL insertion:
  - Normal binary search tree insertion
    - Start at the root and compare values
      - Accordingly, move to the left or the right child
      - Insert where the corresponding child does not exist
    - Balancing condition can only be violated along this path

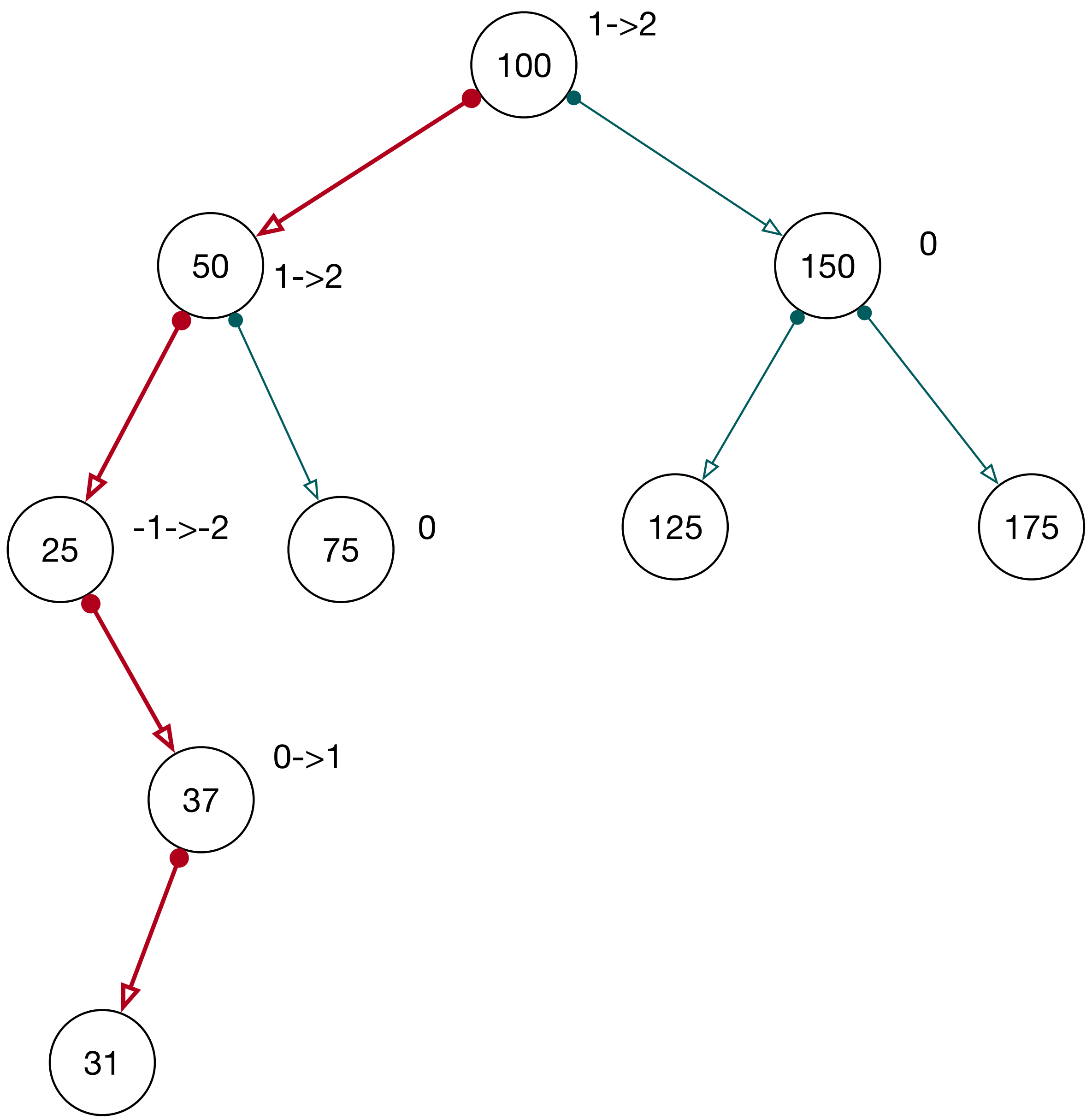
AVL Trees

- AVL Insertion: After inserting 37



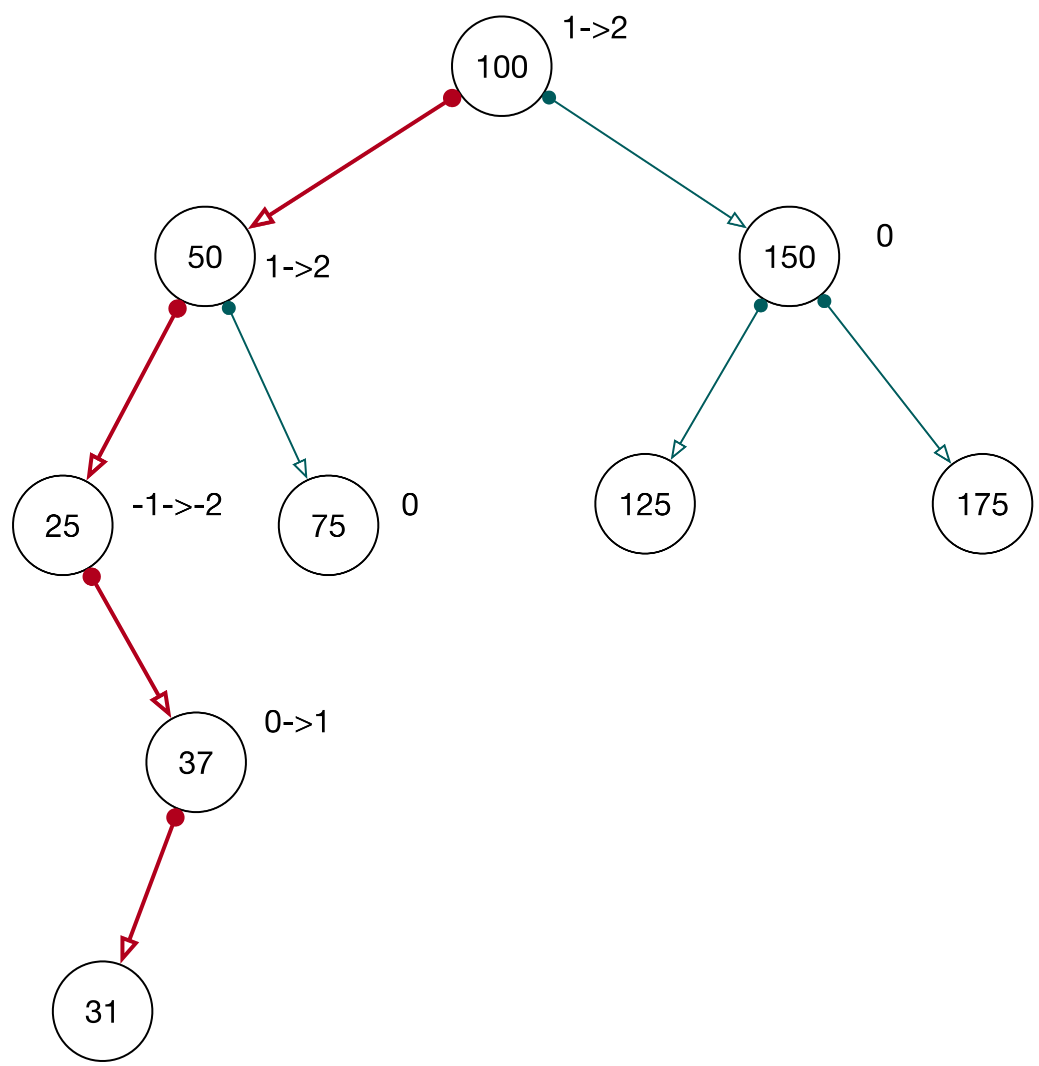
AVL Trees

- AVL Insertion: Balances change only on the insertion path



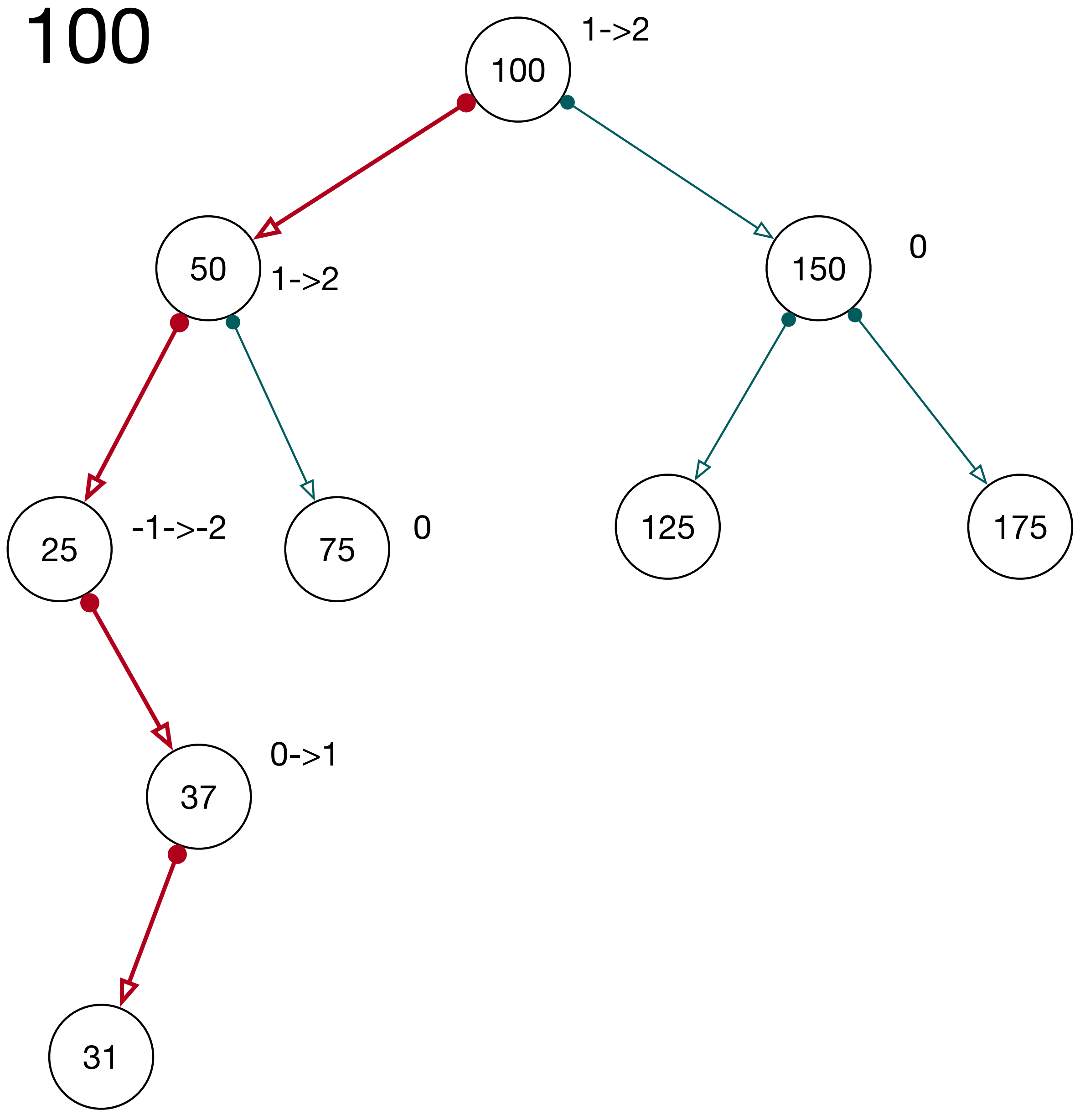
AVL Trees

- When pathing through node 100 (or 50):
  - Cannot decide if balance is becomes bad



AVL Trees

- Therefore: Push nodes on a stack:
  - 37, 25, 50, 100



AVL Trees

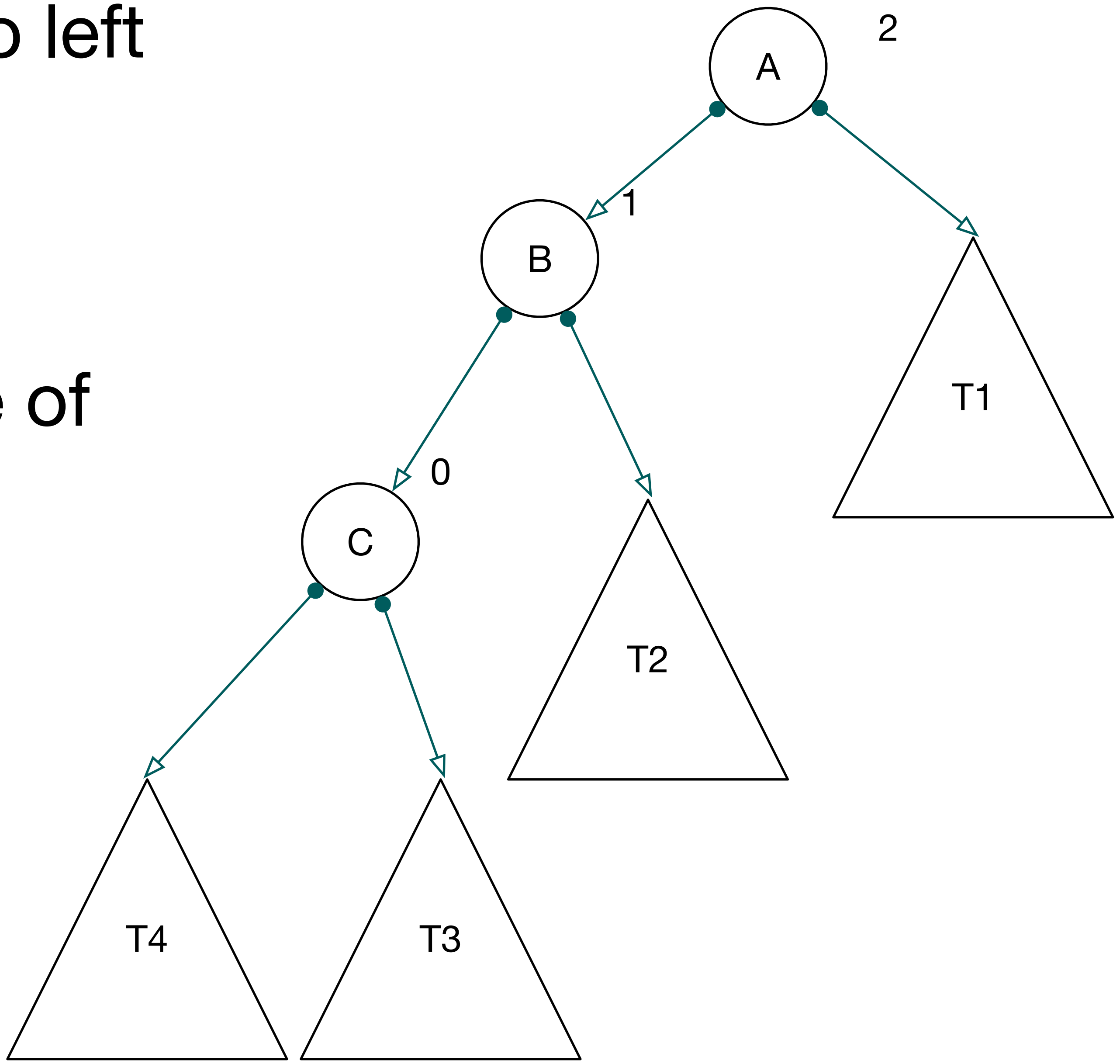
- The balancing repair uses "rotations"
  - We take two or three nodes, reorder them and their sub-trees
  - Have to make many case distinctions

AVL Trees

- How can we obtain an unbalance?
  - Only by inserting into a left or right child
    - Assume balance in a node is 1
      - Left sub-tree has larger height
    - Now we insert into the left sub-tree

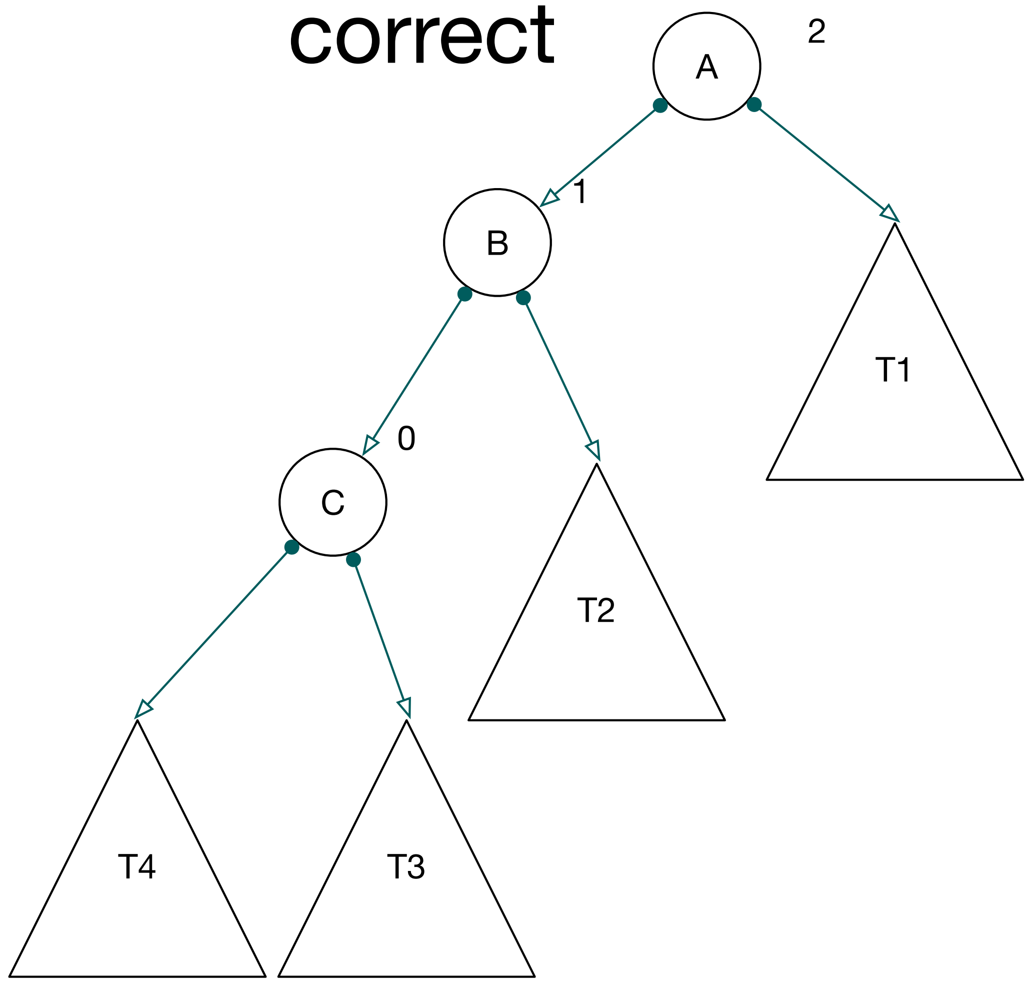
AVL Trees

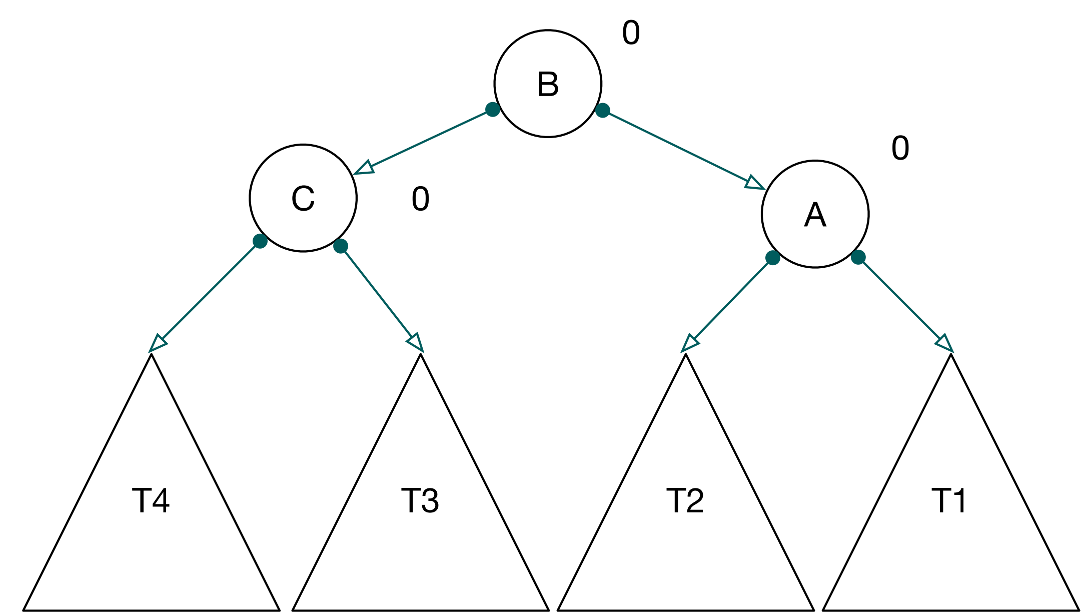
- Case 1: A has balance 2, because of insertion into left child
  - B has balance of 1
  - C can have a balance of -1, 0, or 1



AVL Trees

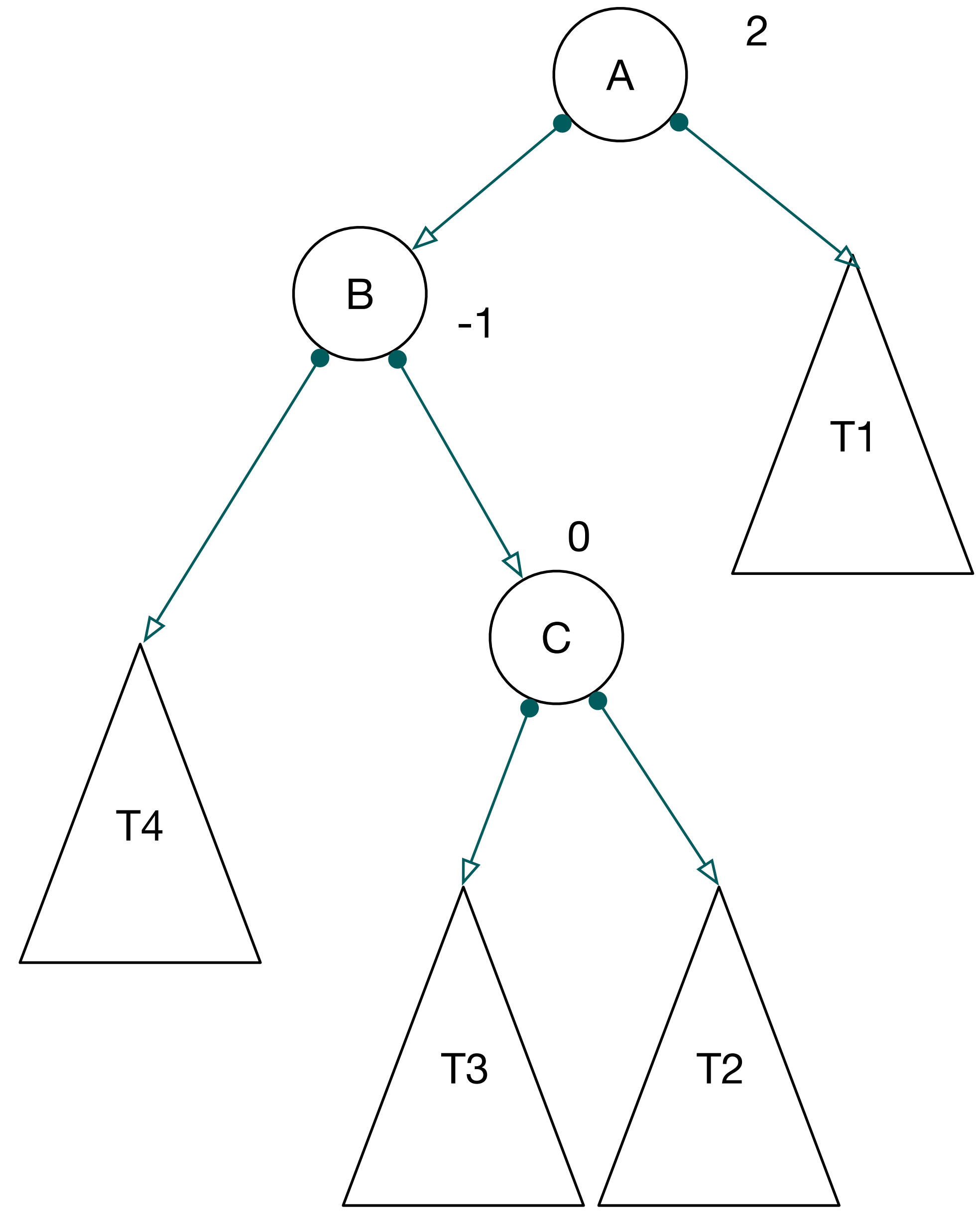
- Right rotation:
  - Check that it is well ordered and that balances are correct





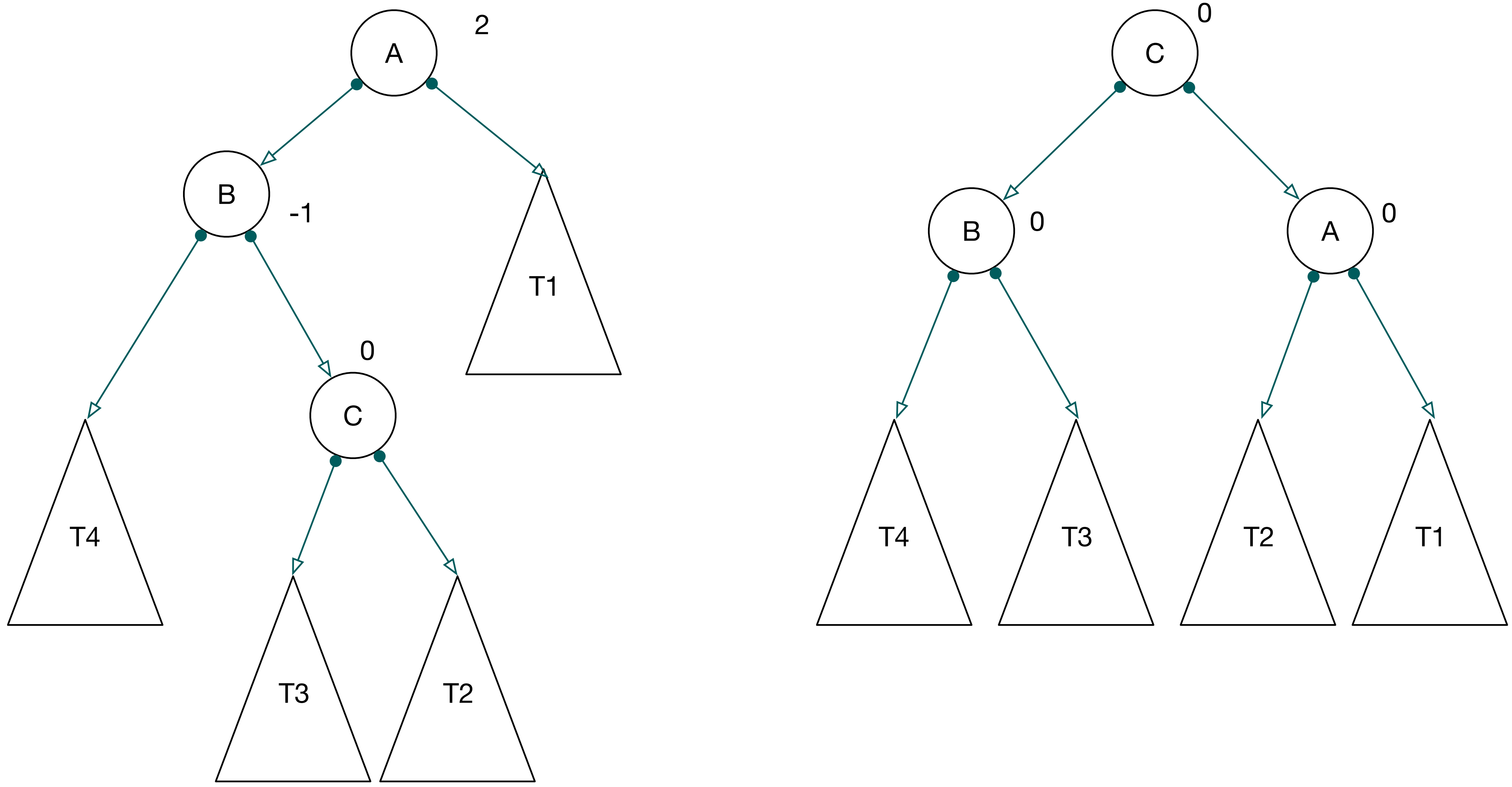
AVL Trees

- Case 2:
  - Subtree in B has increased height
  - Inserted into subtree rooted in C
  - Balance in C is 0, -1, 1



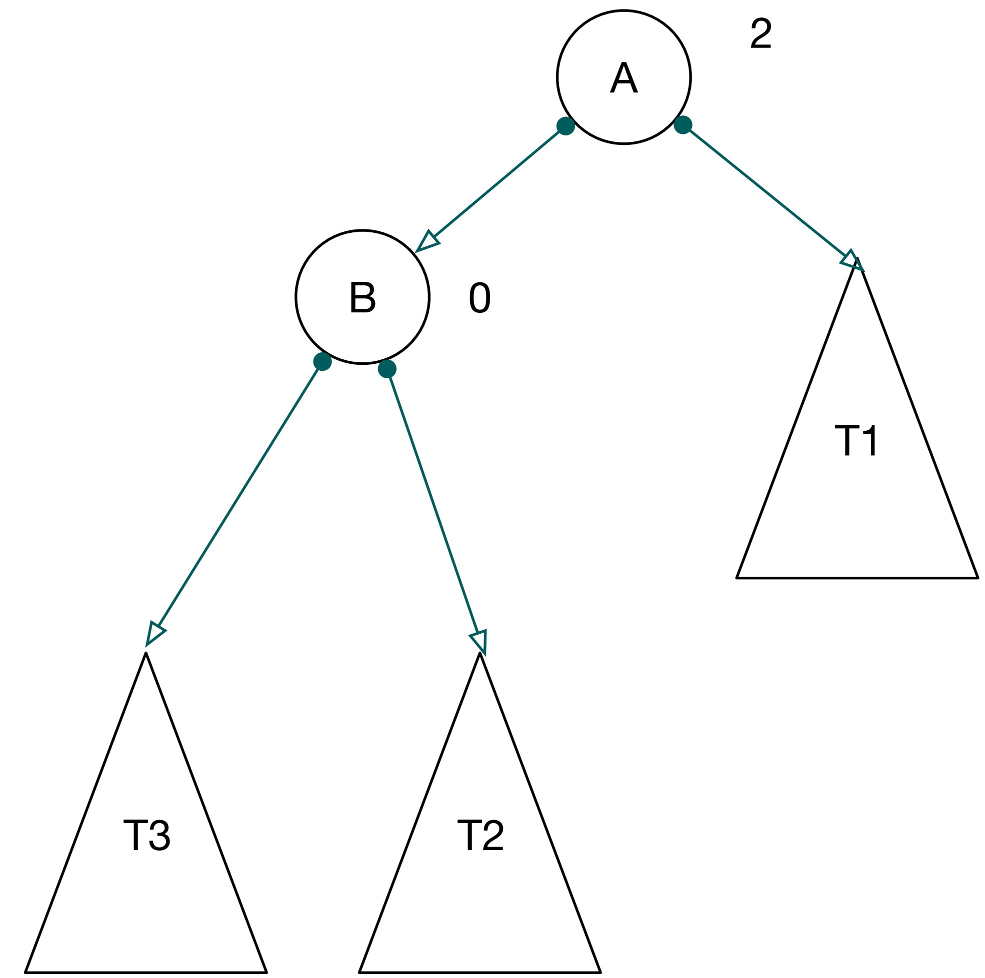
AVL Trees

- Double rotate (A with B and B with C)



AVL Trees

- Can the sub-tree in B have balance 0?
  - NO!
    - If T3 changed height, height in B would not have changed
      - Either balance in B would have been set to 2 or both T3 and T2 have same height
    - If T2 changed height, height of B would not have changed



AVL Tree

- Analogous operations if the right sub-tree increased in height

AVL Tree

- After insertion and a rotation, the new top node has always balance 0
- The new sub-tree has not changed height compared to before insertion
- This means, only one rotation is ever necessary!

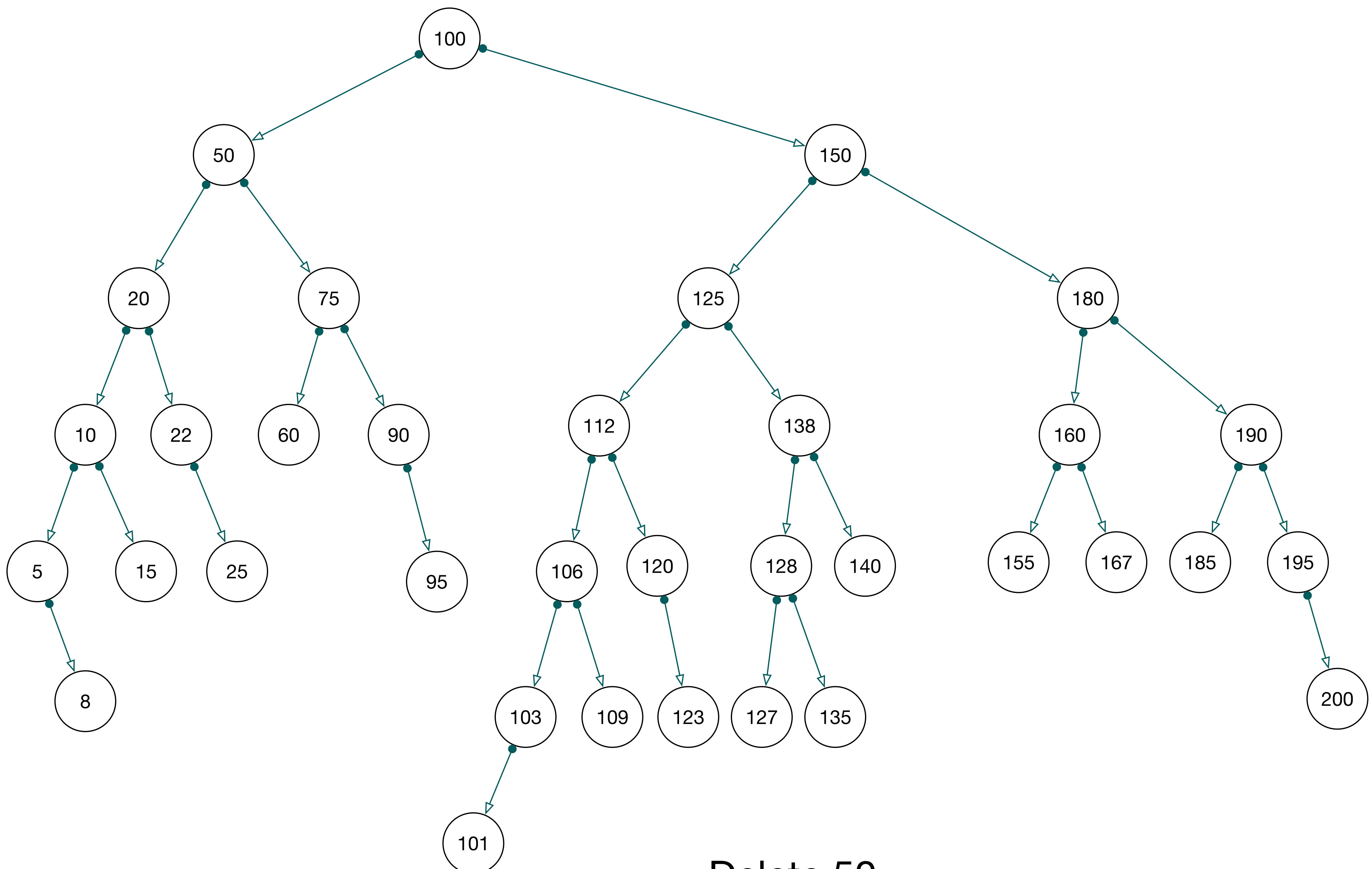
AVL Tree

- Deletions:
  - Do the normal deletion from the tree
    - Remainder:
      - We first find the node to be deleted.
        - If the node has no or only one child, we can delete it.
        - Otherwise find the in-order successor
          - Go right than left-left-left-...
          - Swap contents and then delete successor

AVL Tree

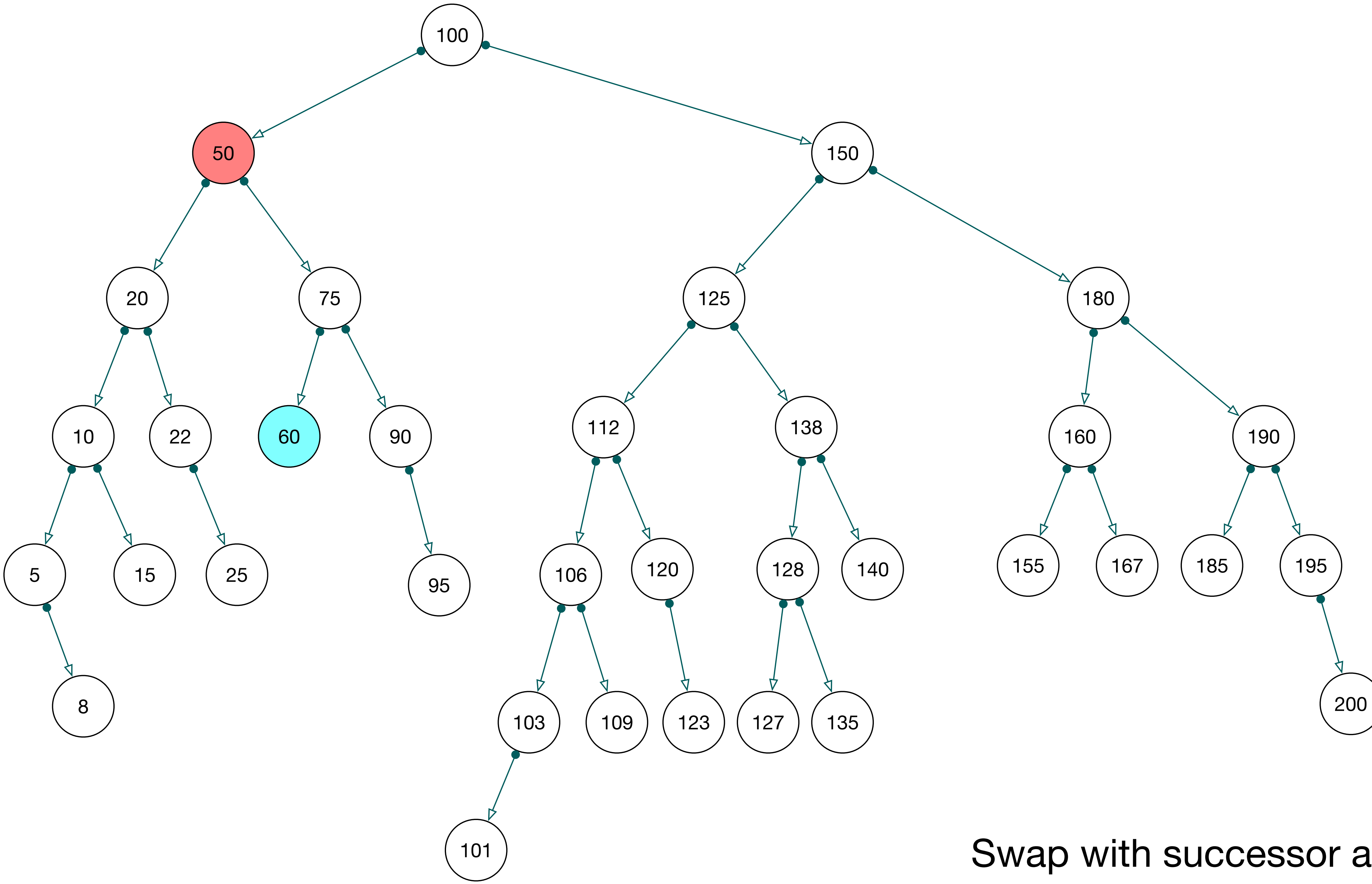
- Once we delete a node:
  - Go back on the path to the node
  - Use the same rotations in order to balance the node
  - But now, balancing can change the height of a subtree before deletion and after deletion cum rotate
  - So, we cannot stop after a single rotate but need to go up all the way to the root to insure balances

AVL Tree



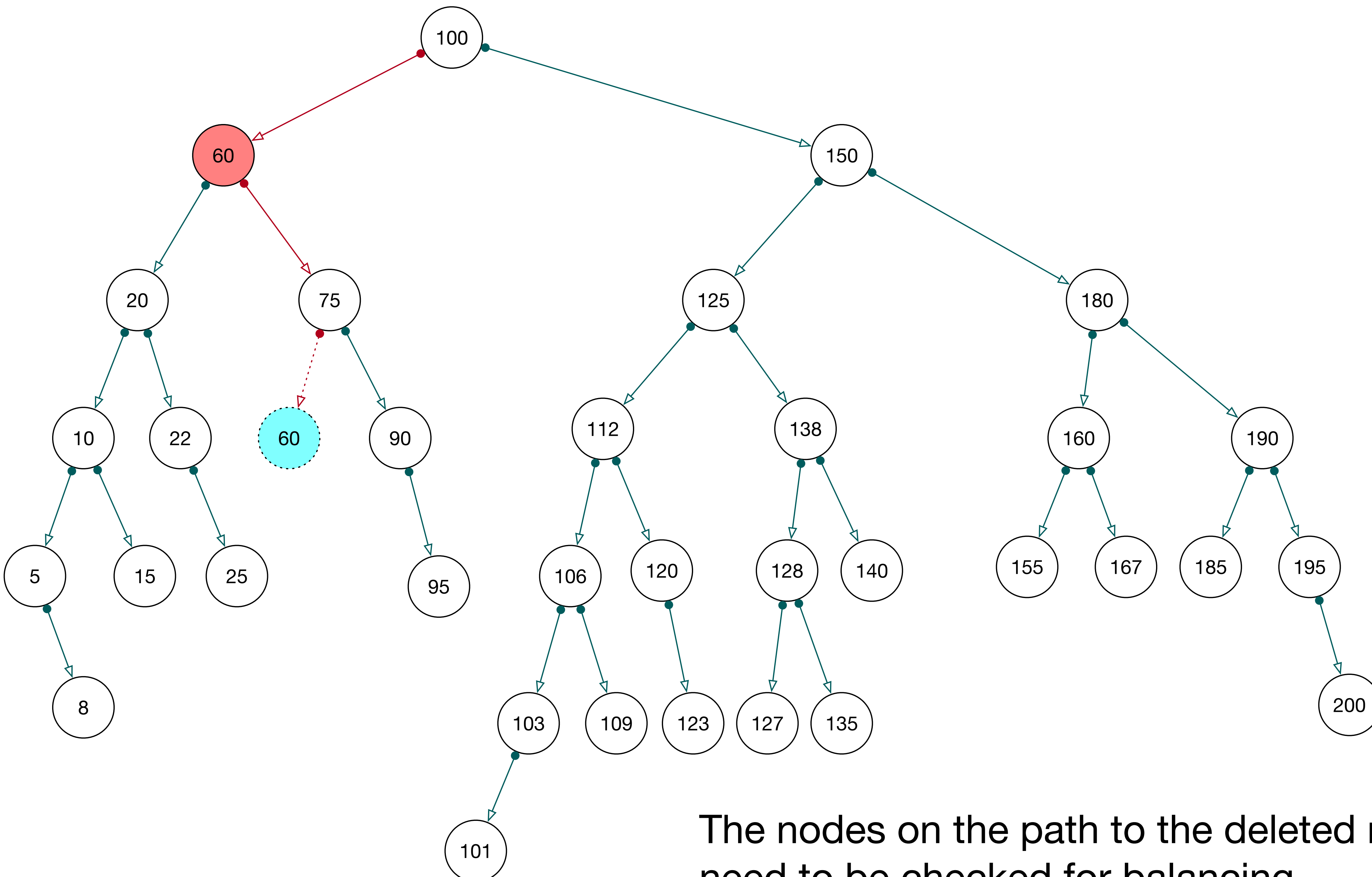
Delete 50

AVL Tree



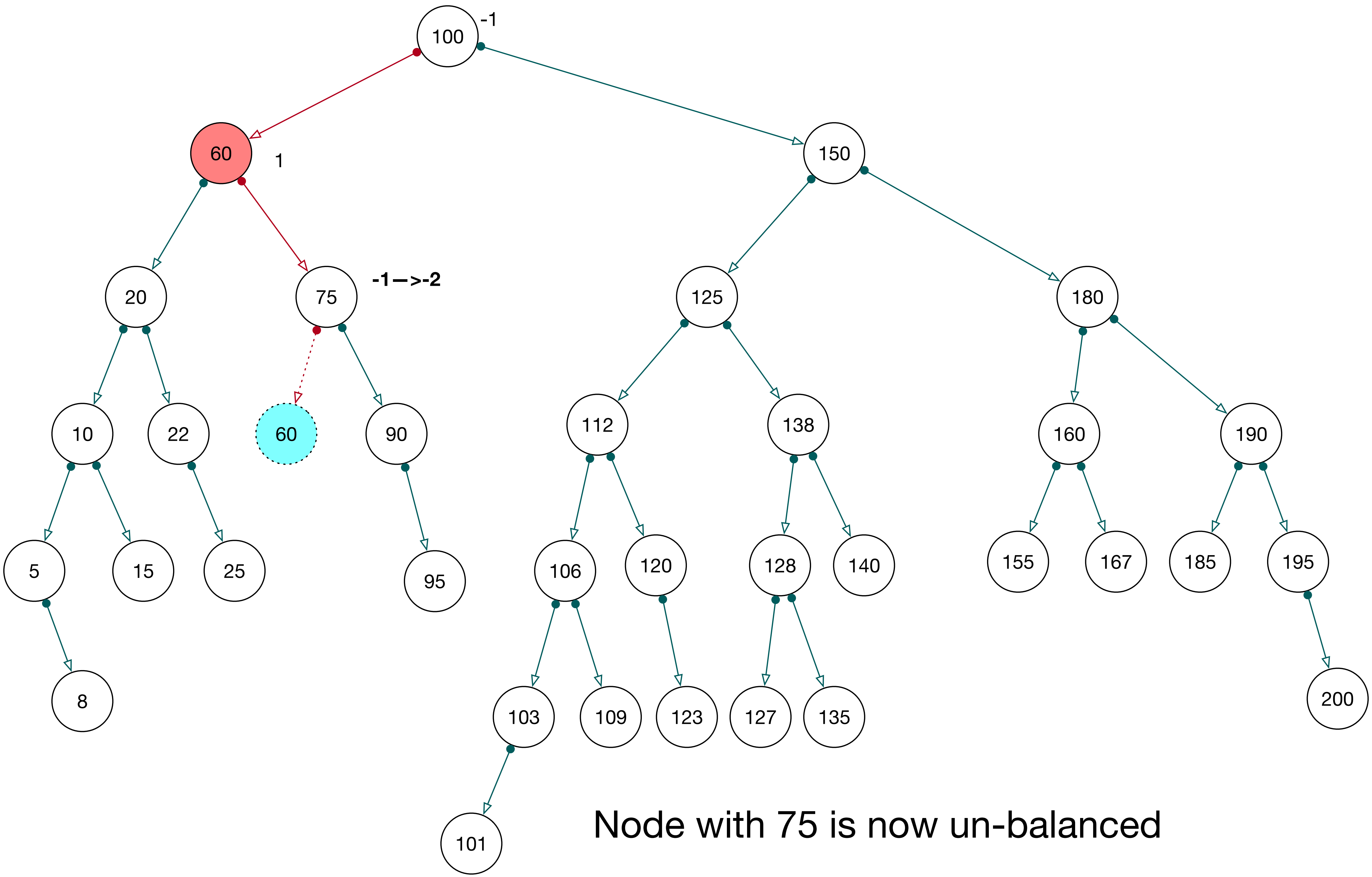
Swap with successor and delete

AVL Tree



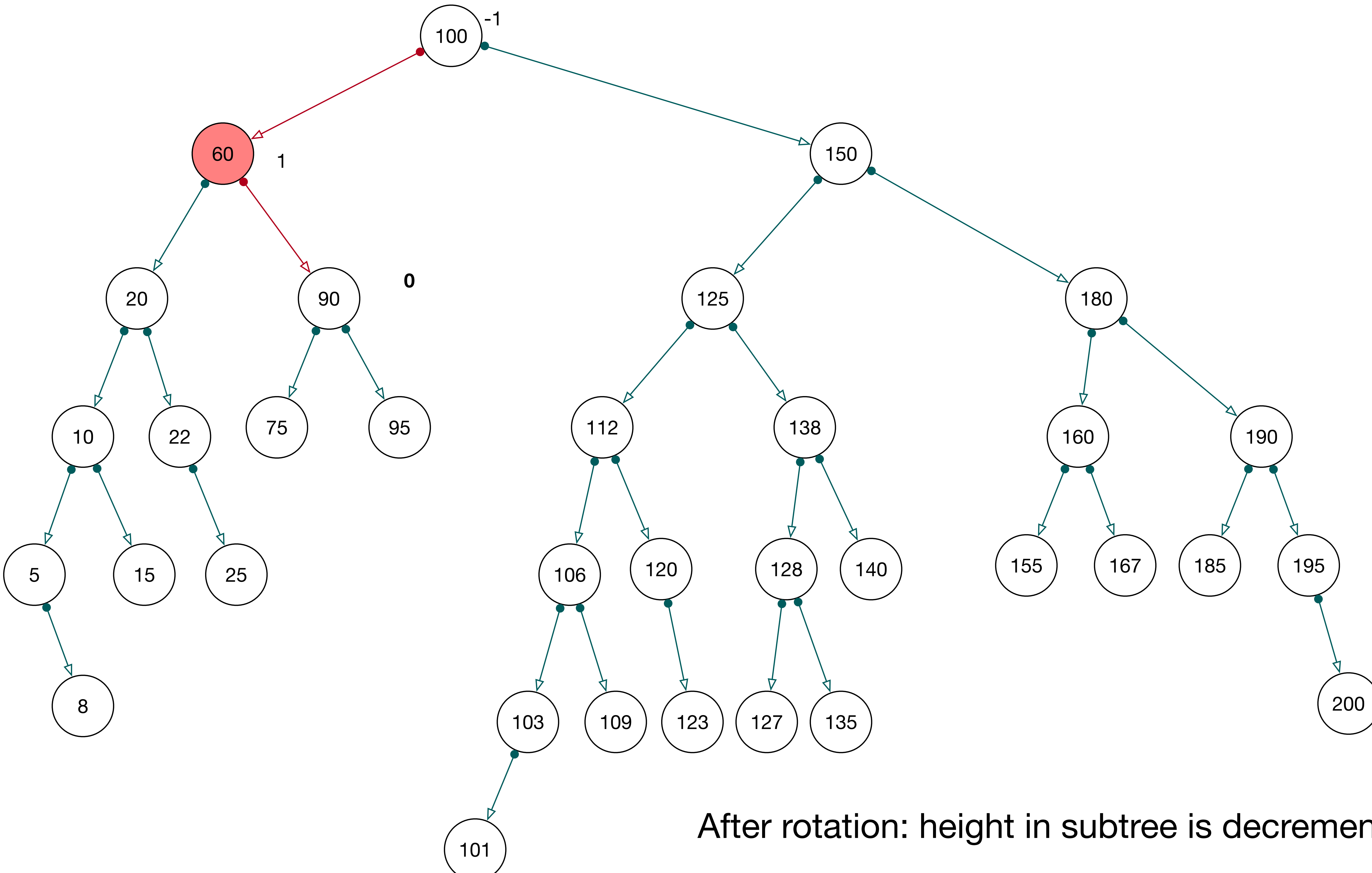
The nodes on the path to the deleted record need to be checked for balancing

AVL Tree



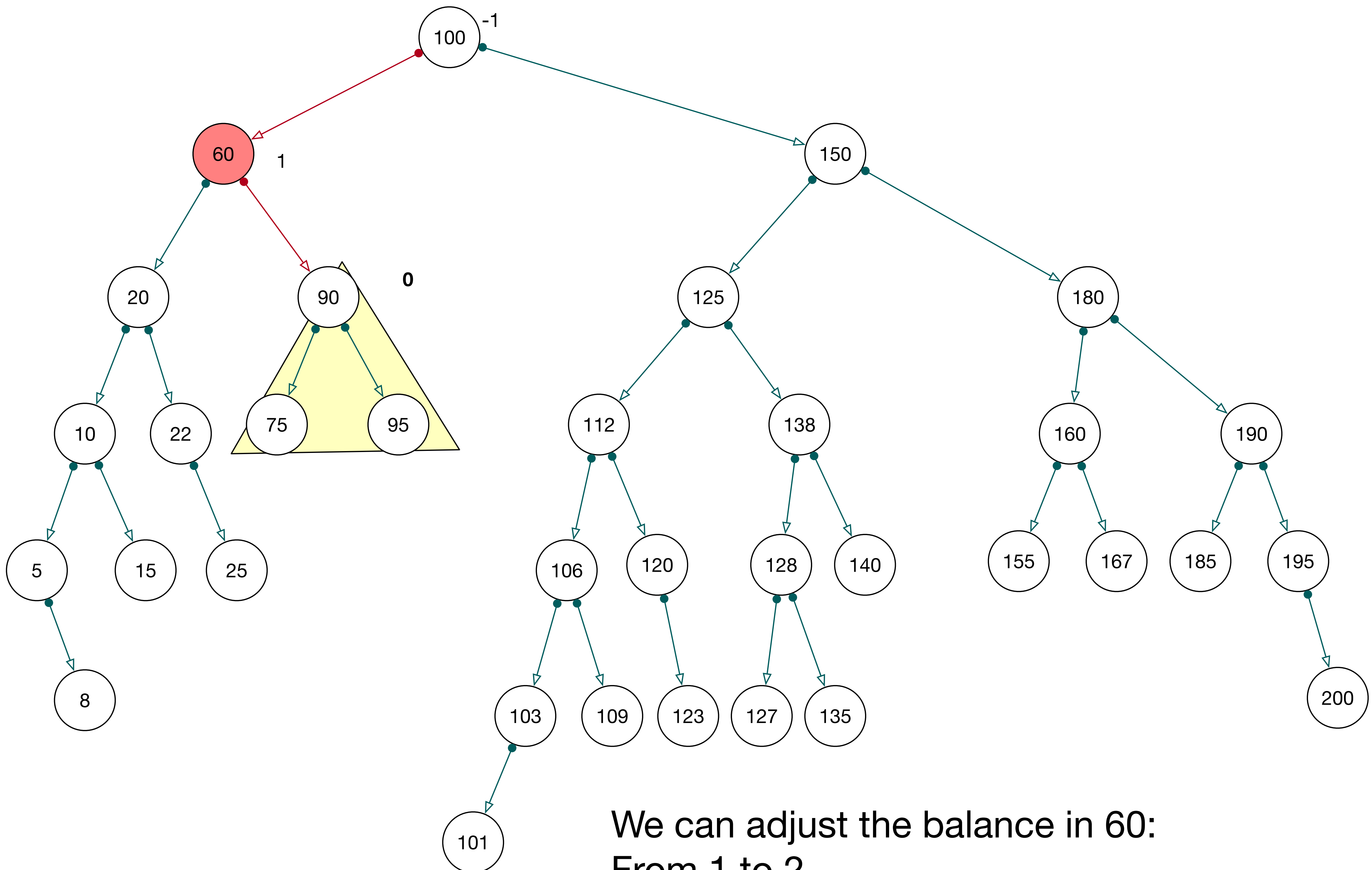
Node with 75 is now un-balanced

AVL Tree



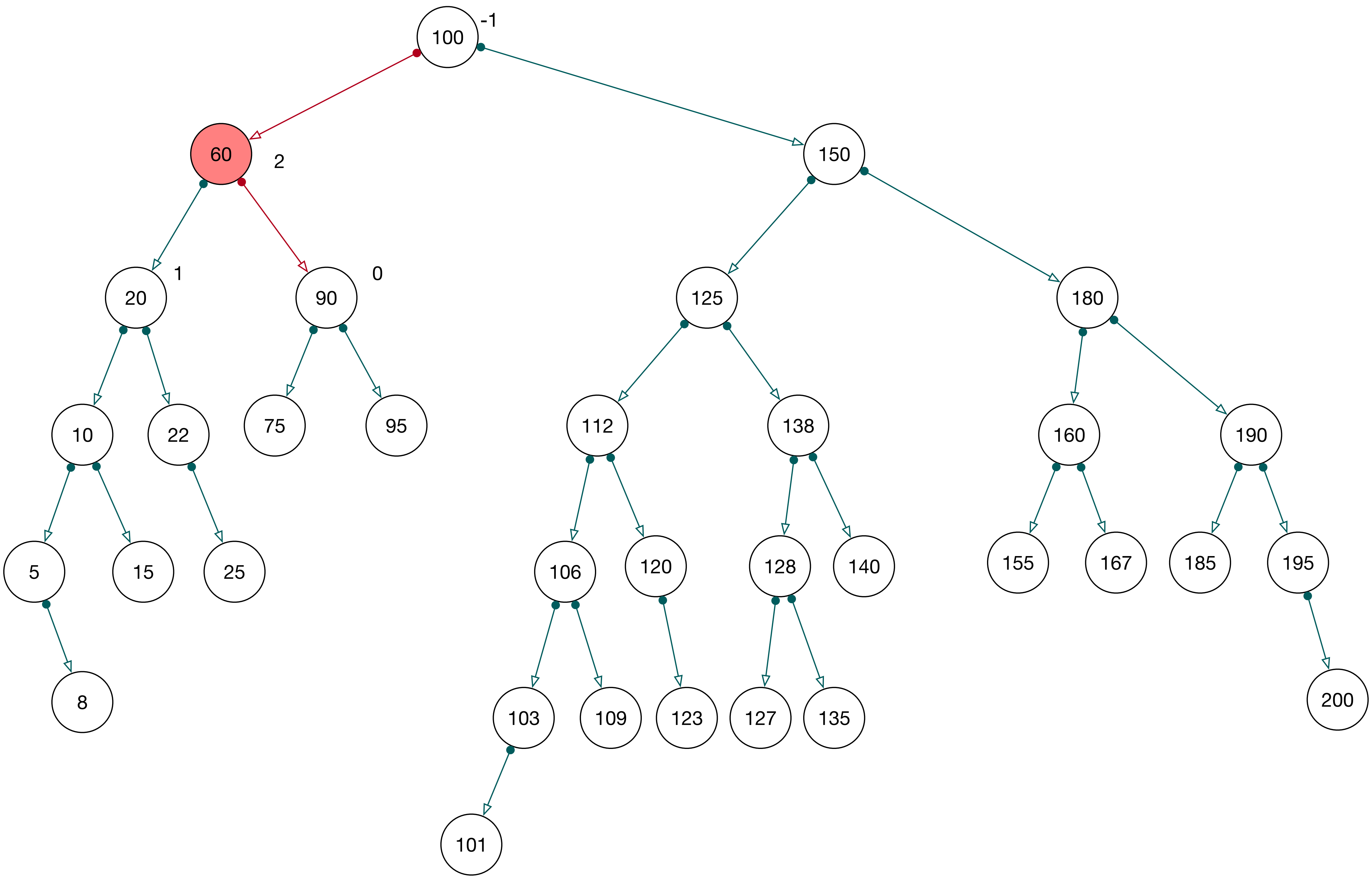
After rotation: height in subtree is decremented

AVL Tree

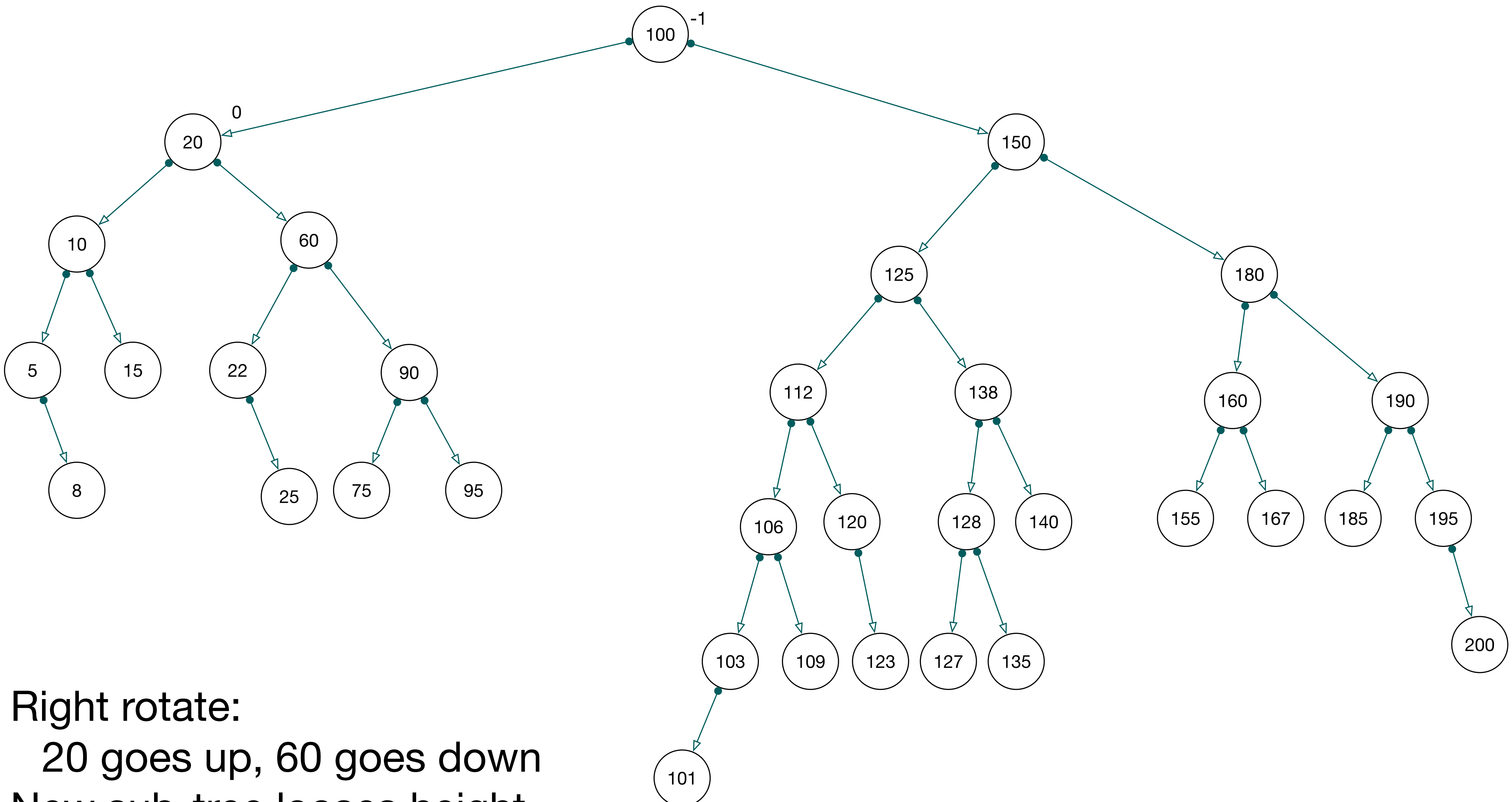


We can adjust the balance in 60:
From 1 to 2

AVL Tree

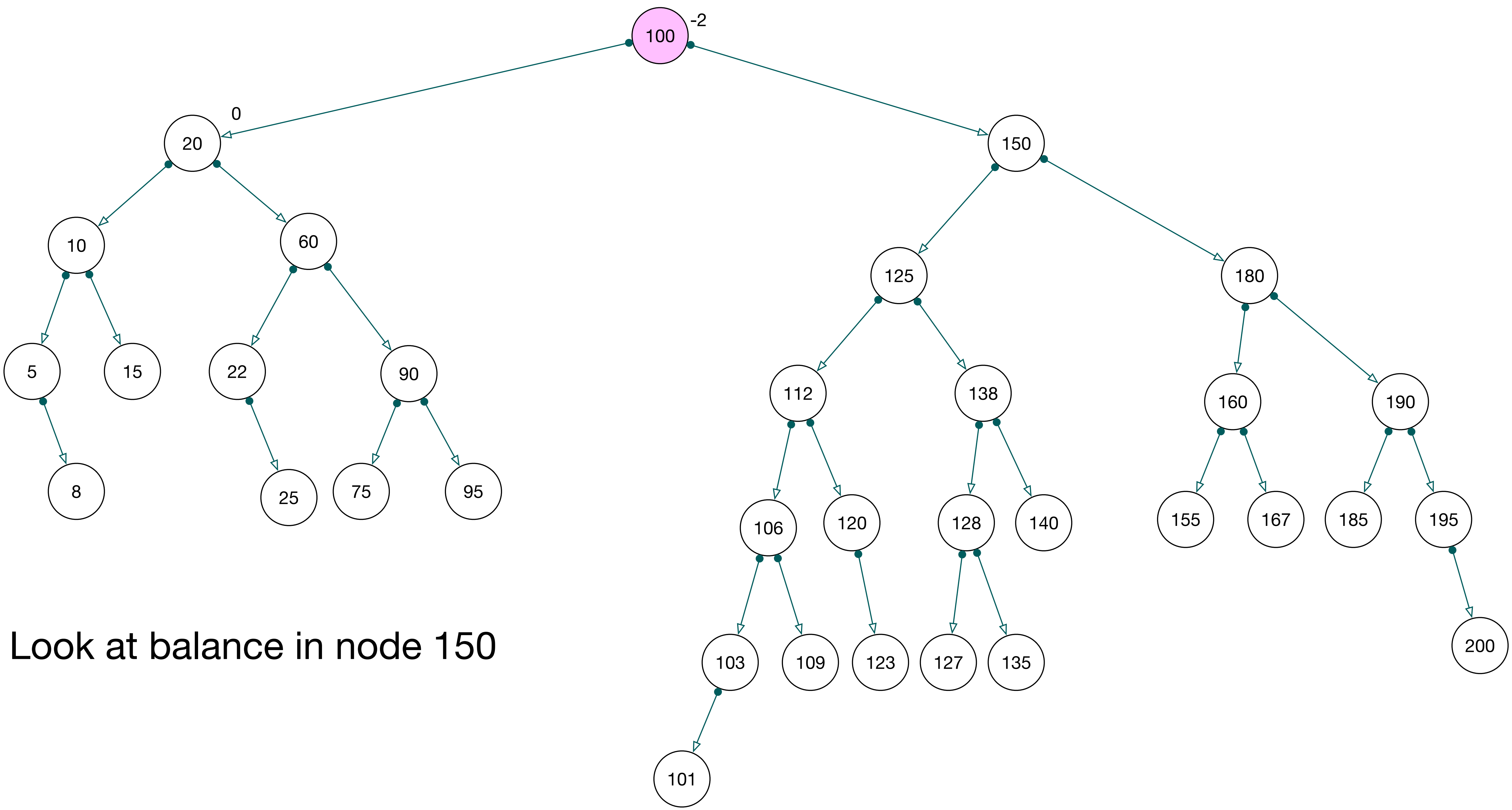

100
-1
60
2
150
20
1
90
0
125
180
10
22
75
95
112
138
160
190
5
15
25
106
120
128
140
155
167
185
195
8
103
109
123
127
135
200
101

AVL Tree



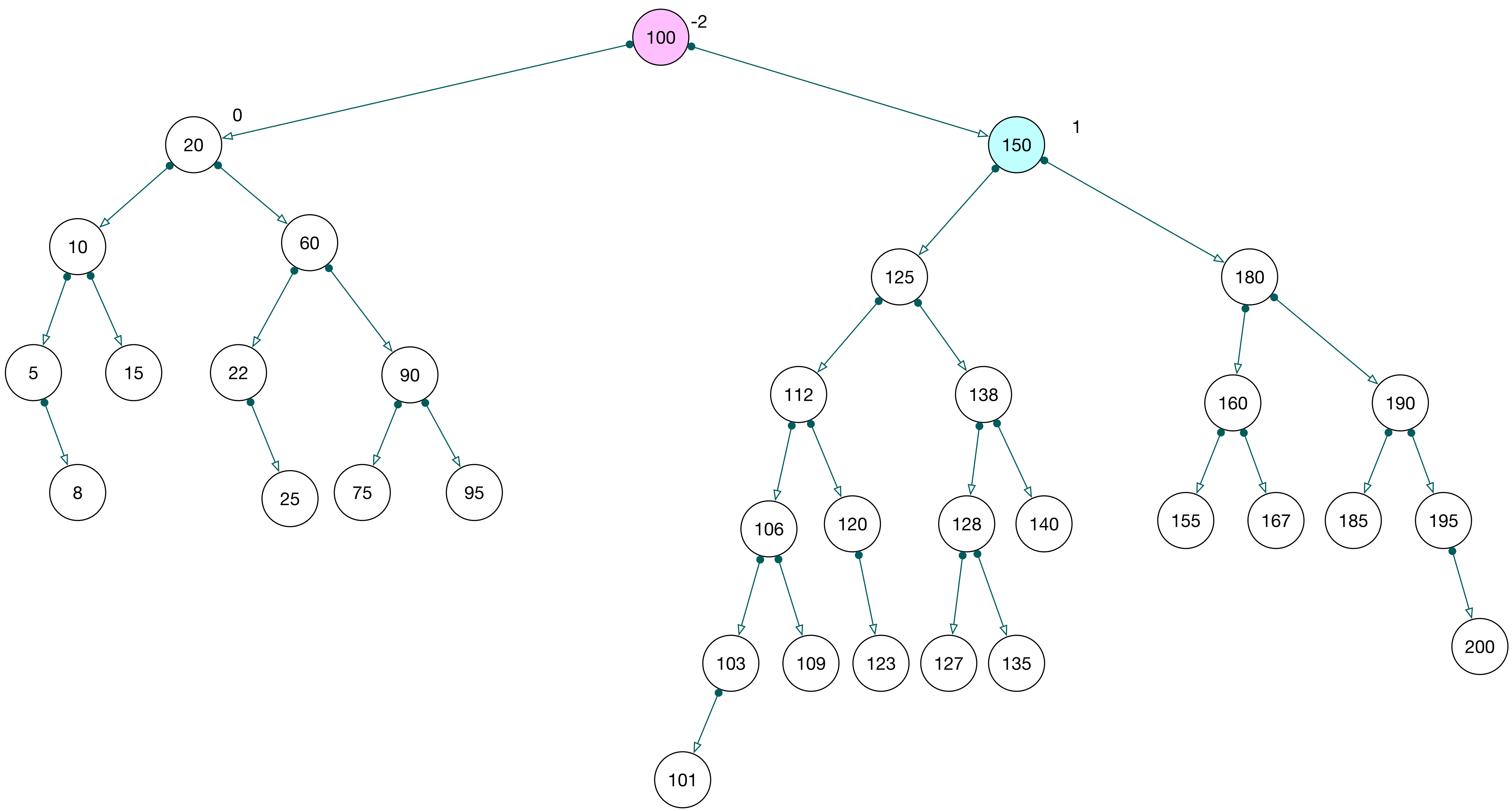
Right rotate:
 20 goes up, 60 goes down
New sub-tree looses height
Need to adjust balance in root

AVL Tree



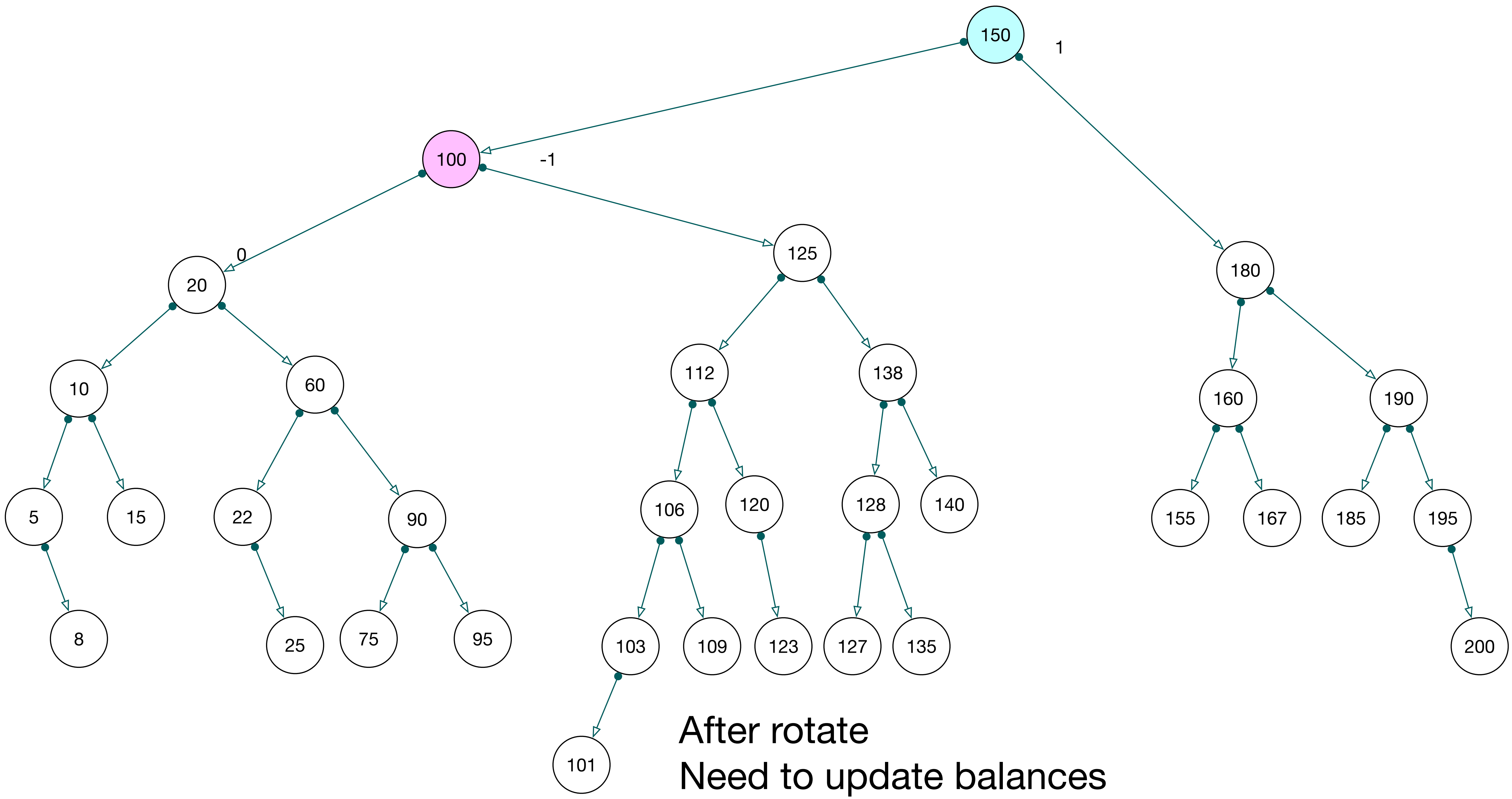
Look at balance in node 150

AVL Tree



100
-2
20
0
150
1
10
60
125
180
5
15
22
90
112
138
160
190
8
25
75
95
106
120
128
140
155
167
185
195
103
109
123
127
135
200
101

AVL Tree



After rotate
Need to update balances

AVL Tree

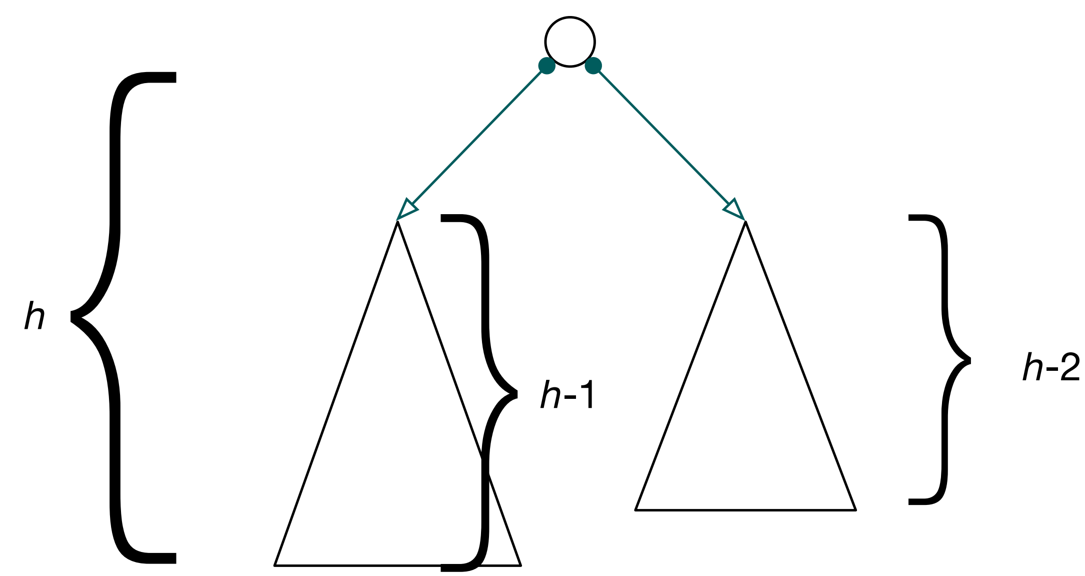
- We can update balances based on
  - type of rotation
  - the balances of the trees

AVL Tree

- Performance:
  - We now: maximum number of nodes in a tree of height *h* is
    - $1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$
  - What is the minimum number of nodes in a tree of height *h*?
  - Call this number n_h

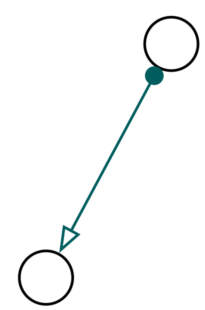
AVL Tree

- What is the minimum number of nodes in a tree of height *h*
  - At the root, one subtree has height one less than the other:
    - $h_n = 1 + h_{n-1} + h_{n-2}$



AVL Tree

- What is the minimum number of nodes in a tree of height h?
  - For $h = 1$



  - $n_0 = 1 \;\; n_1 = 2$

AVL Tree

- Recursion:
  - $n_h = 1 + n_{h-1} + n_{h-2} \qquad n_0 = 1 \qquad n_1 = 2$
  - Can be solved via the Fibonacci series:
    - $(n_h + 1) = (n_{h-1} + 1) + (n_{h-2} + 1)$
- Can be solved exactly or approximately
  - $n_h \approx 1 + \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+3}$

AVL Tree

- Reversely:
  - Sparsest ALV tree with n nodes has height $\approx 1.44 \log_2(n+1) - 1.33$
  - Fullest AVL tree with n nodes has height $\log_2(n+1) - 1$

AVL Tree

- Insertion:
  - Proportional to height of tree
- Deletion:
  - Proportional to height of tree