AVL Trees

Note:

Text representation of some trees are also be given in the slides, in which case the format is:

root(first[first.first, first.second], second[second.first, second.second])

Problem with BST

- Binary Search Trees are fast if they're shallow.
- Problems occur when one branch is much longer than the other.

Balance Factor

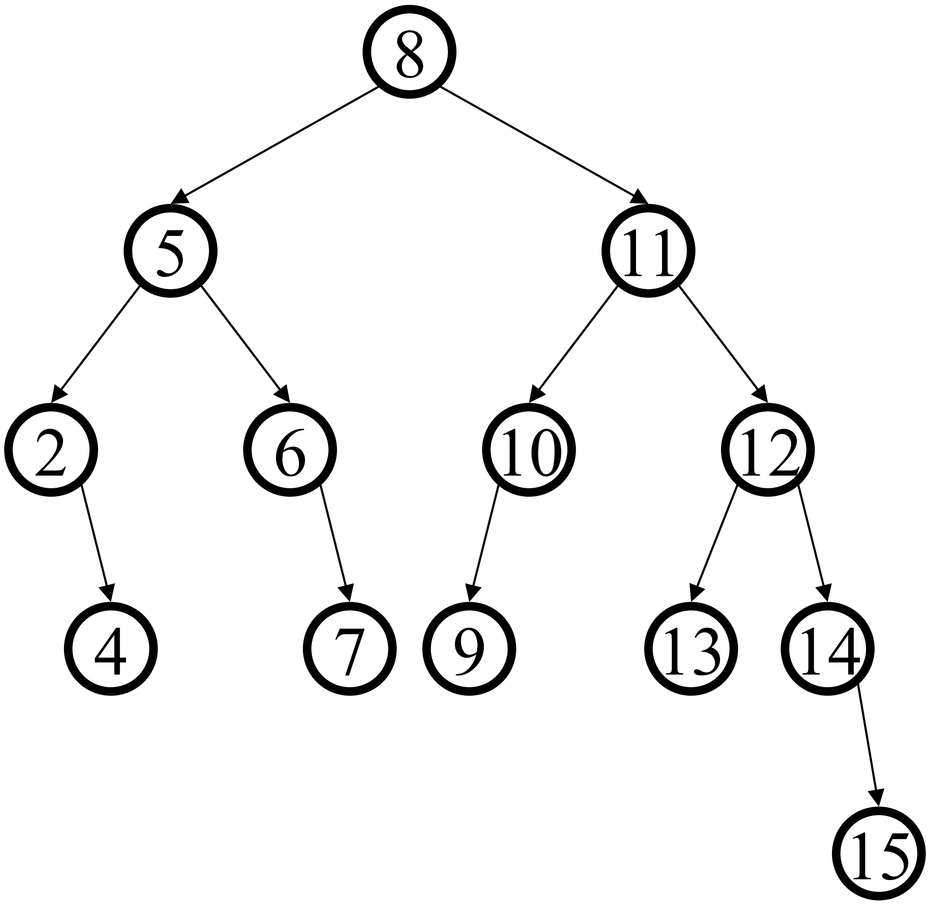
Balance Factor of a node

= height(left subtree) - height(right subtree)

AVL Tree (Adelson-Velskii Landis)

- A binary search tree
- Balance Factor of every node is **$-1 \leq b \leq 1$**
- Tree re-balances itself after every insert or delete

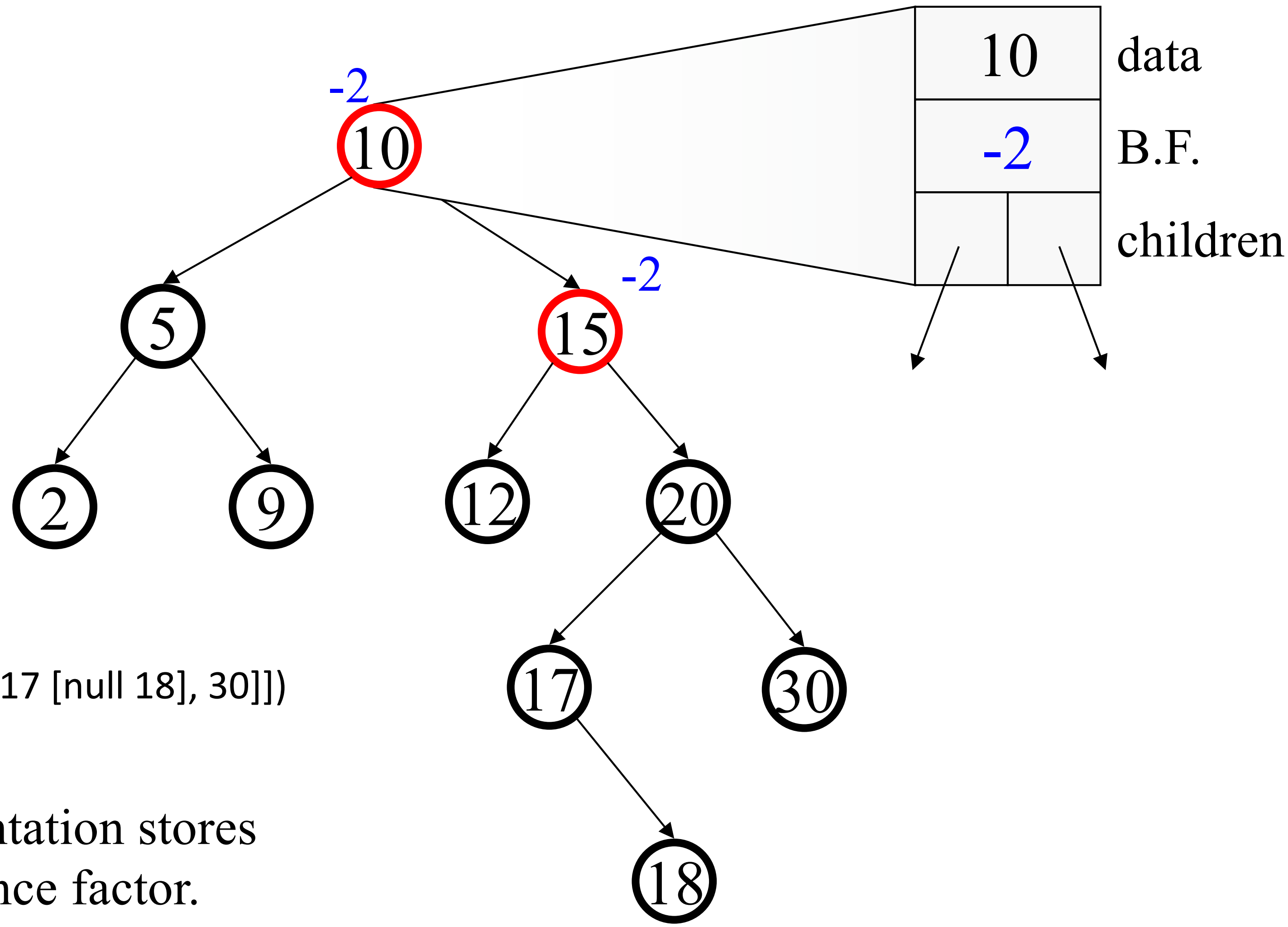
Example of an AVL tree



- 8 (5 [2 [null,4], 6 [null, 7]], 11[10 [9, null],12 [13, 14[null, 15]]])

What is the balance factor of each node in this tree?

Not An AVL Tree



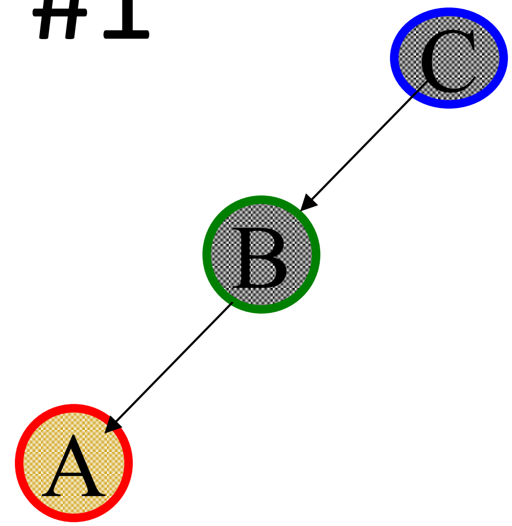
- 10 (5 [2 , 9], 15[12 , 20 [17 [null 18], 30]])

Note: Some implementation stores height instead of balance factor.

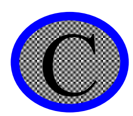
4 cases that can cause imbalance: (symmetric)

- Case 1: An insertion into the *left subtree of the left child*
- *case* 2: *Insertion into the right subtree of the left child*
- case 3:*Insertion into the left subtree of the right child*
- case 4: An insertion into the right subtree of the right child

Bad Case #1

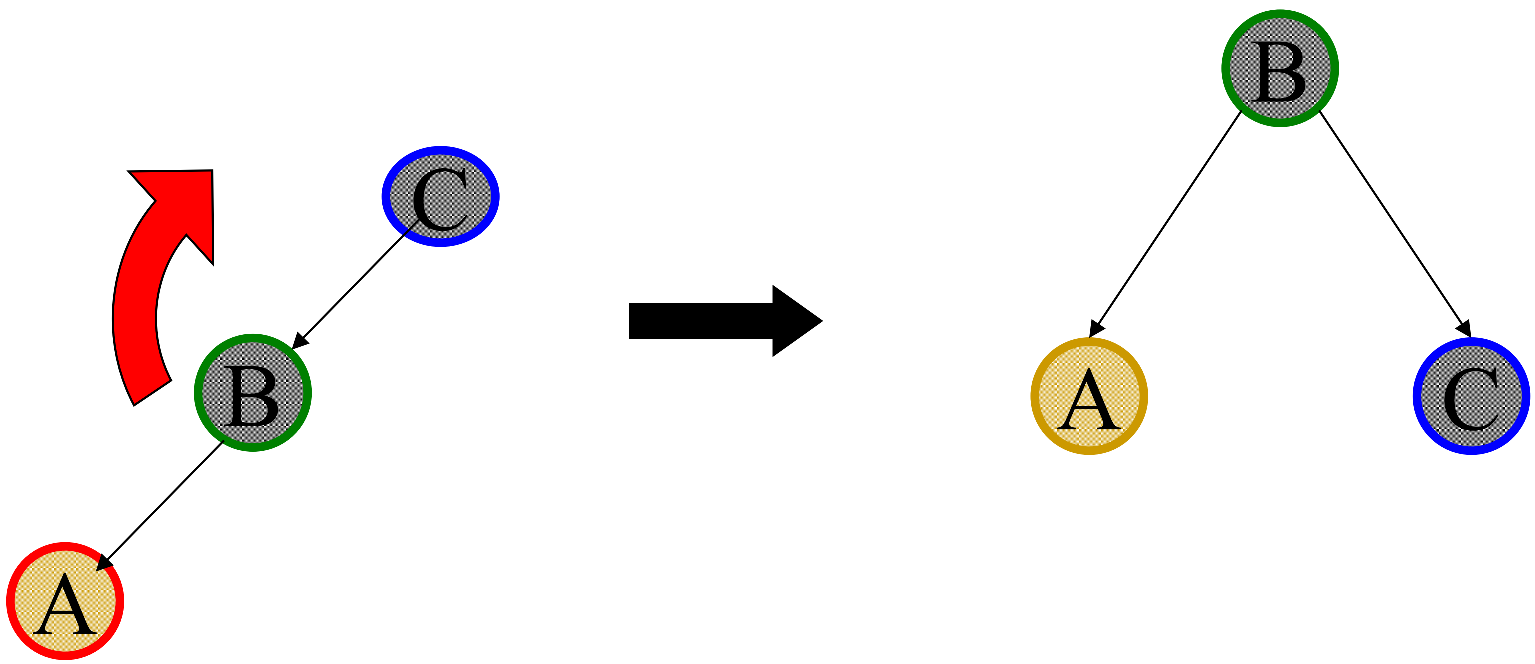


- Insert C, B, A
- C (B [A, null], null)

- An insertion into the *left subtree of the left child* of 

Single Rotation: Rotate Right

- Two much left, rotate to right!



C (B [A, null], null) -> B (A, C)

Rotation

- An operation on the binary tree that moves one node or subtree up and another down.
- Change the shape but does not change the order.
- Typically for the purpose of reducing height.

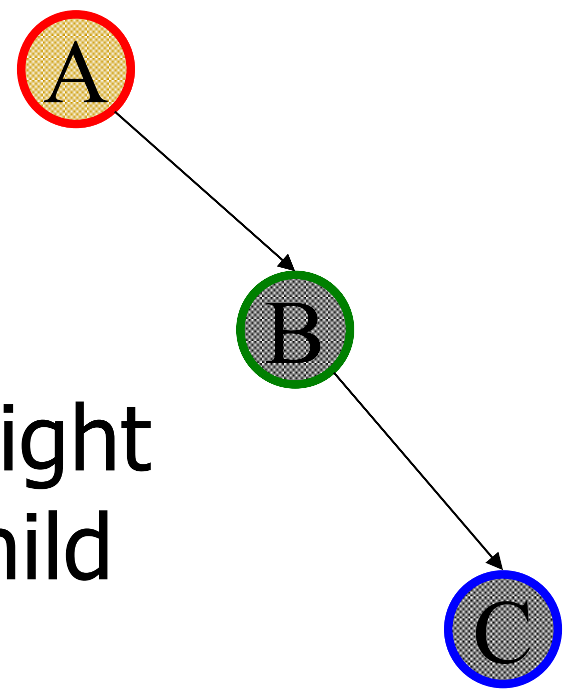
Single Right Rotation

1. Change the parent of C to point to B.
2. Set the left link of C equal to the right link of B.

   (In the example, B's right link is null)
3. Set the right link of B to point to C.

Bad Case #4

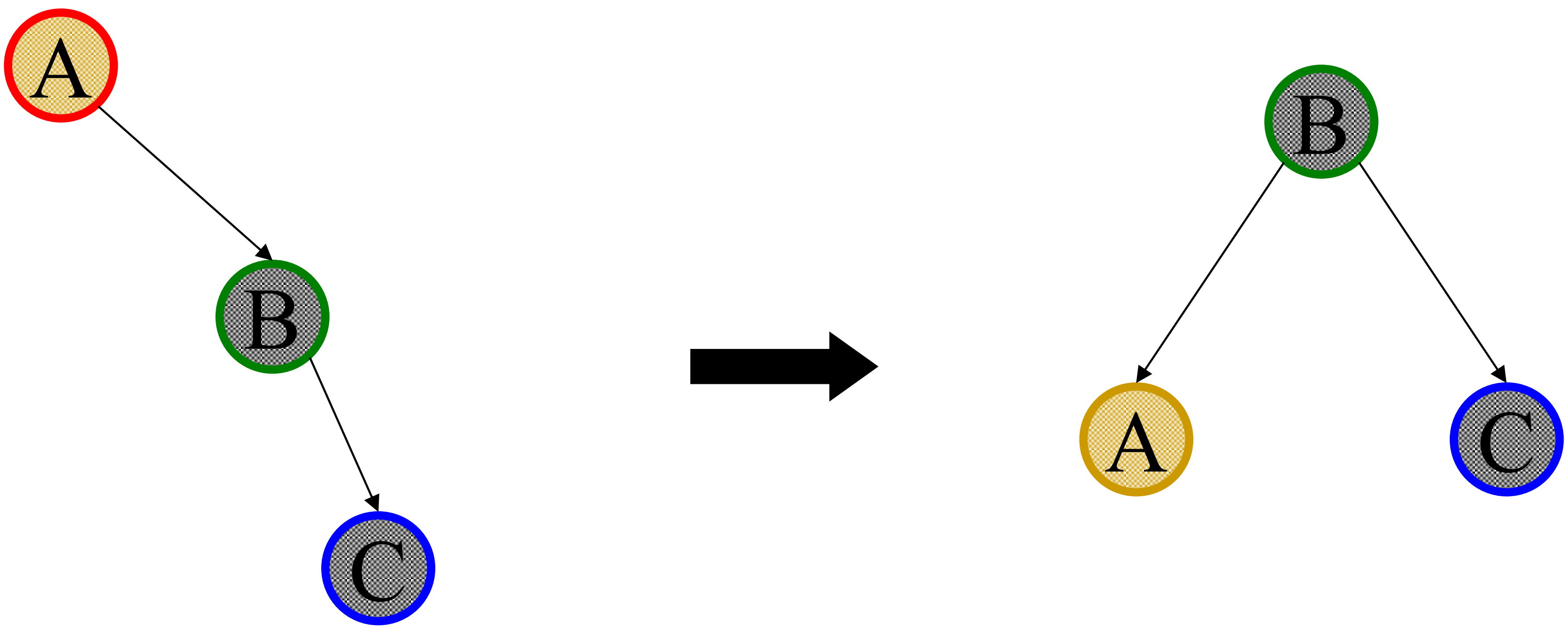
- Insert A, B, C
- A (null, B [null, C])



- An insertion into the right subtree of the right child of A

Single Rotation: Rotate Left

Two much right, rotate left!



A (null, B [null, C])   →   B (A, C)

Single Left Rotation

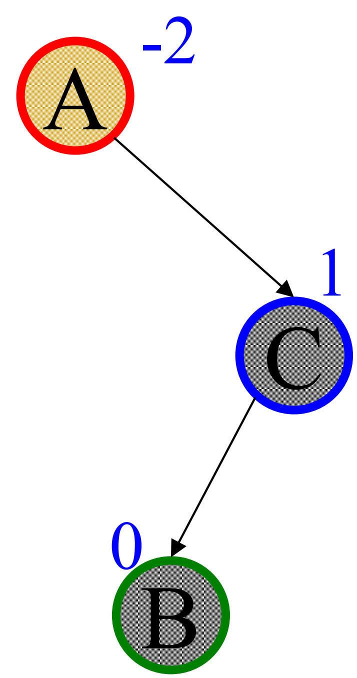
1. Change the parent of A to point to B.
2. Set the right link of A equal to the left link of B.
3. Set the left link of B to point to A.

Properties of Single Rotation

- Restores balance to a lowest point in tree where imbalance occurs.
- After rotation, height of the subtree is the same as it was before the insert that imbalanced it.
- Thus, no further rotations are needed anywhere in the tree!

Bad Case #3

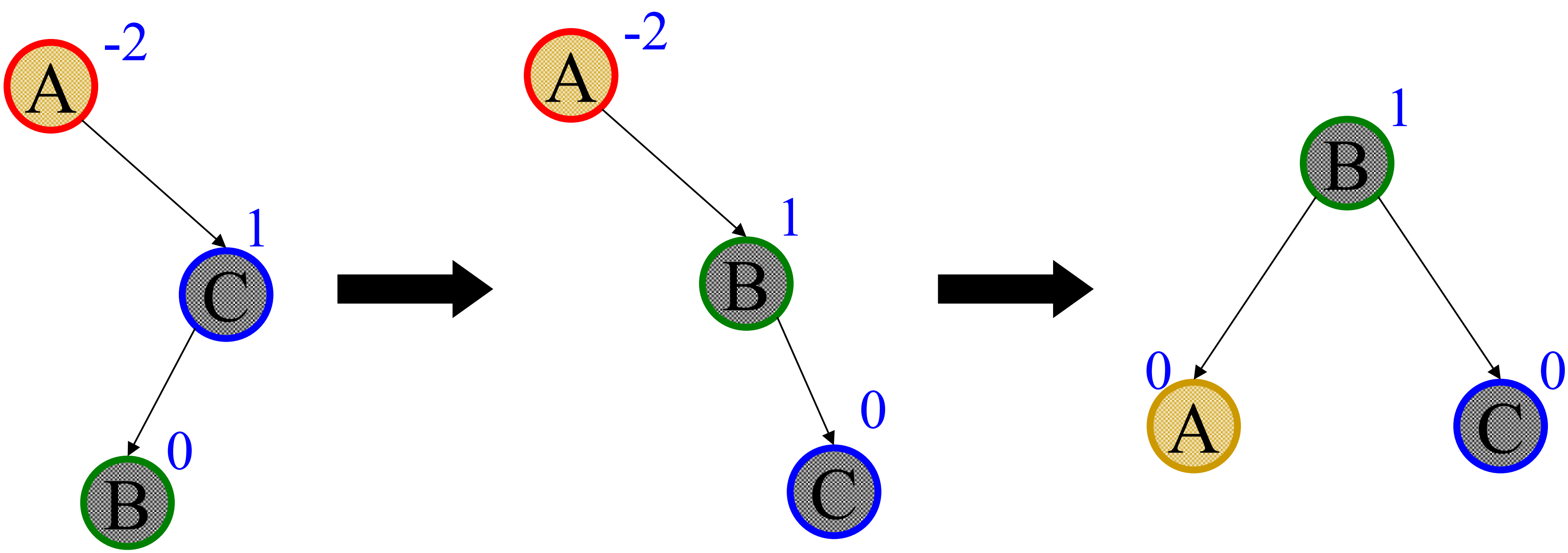
- Insert A, C, B
- A (null, C[B, null])



- *Insertion into the left subtree of the right child of A.*
- *One single rotation doesn't work for this.*

Double Rotation: Rotate-RL

Two single rotations: single right rotation to rotate up B, then single left rotation to rotate up B again.

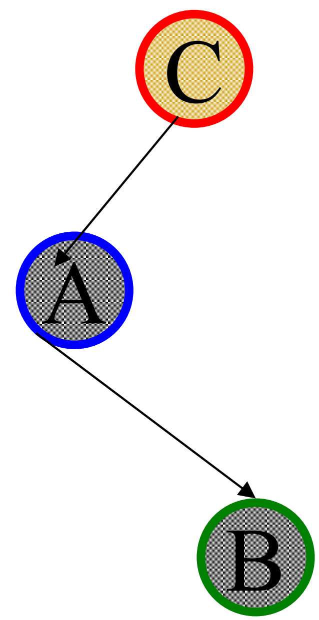


A (null, C[B, null]) → A (null, B[null, C]) → B (A, C)

Bad Case #2

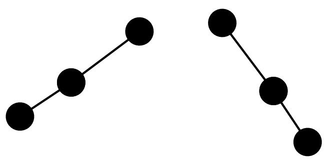
Insert C, A, B

- *Insertion into the right subtree of the left child of*
- *One single rotation doesn't it work.*
- *The solution is symmetric to bad case #3:Double rotation: Rotate-LR*
    - *Rotate B left, get case #1,*
    - *Rotate B right, get a balanced tree: B(A, C)*

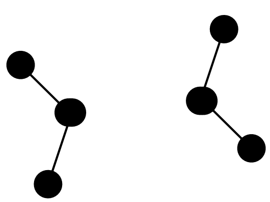


Insert Algorithm Summary

1. Find the spot for inserting the value.
2. Hang the new node.
3. Search back up looking for imbalance.
4. If there is an imbalance:



“outside”: Perform single rotation and exit.



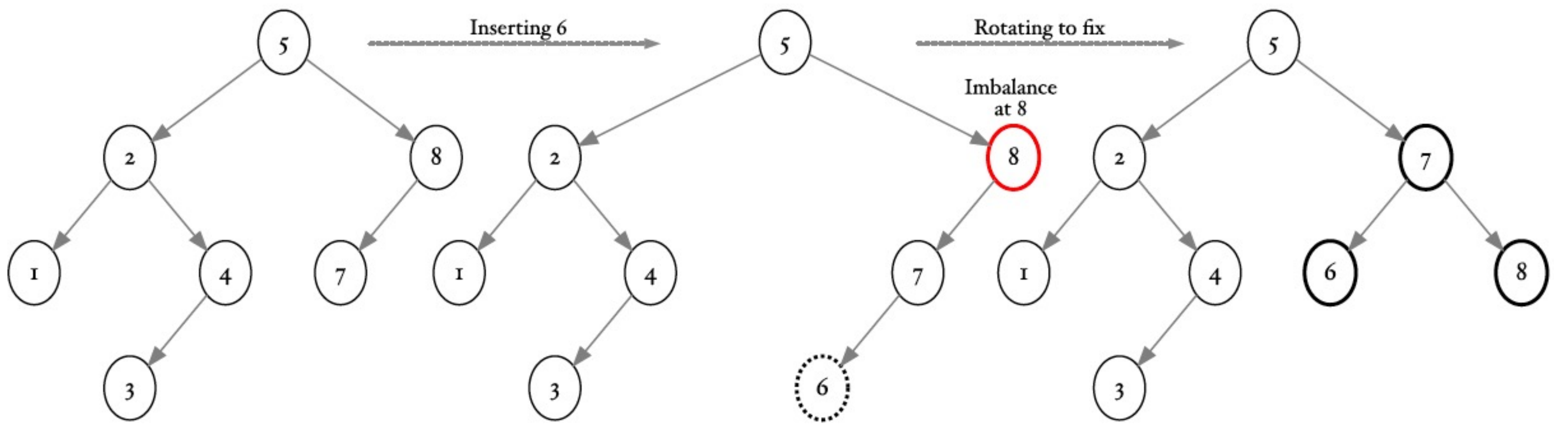
“inside”: Perform double rotation and exit.

Note: After such rotation is applied to any subtree where the balance is off, the balance of the large AVL tree will be kept.

Insertion

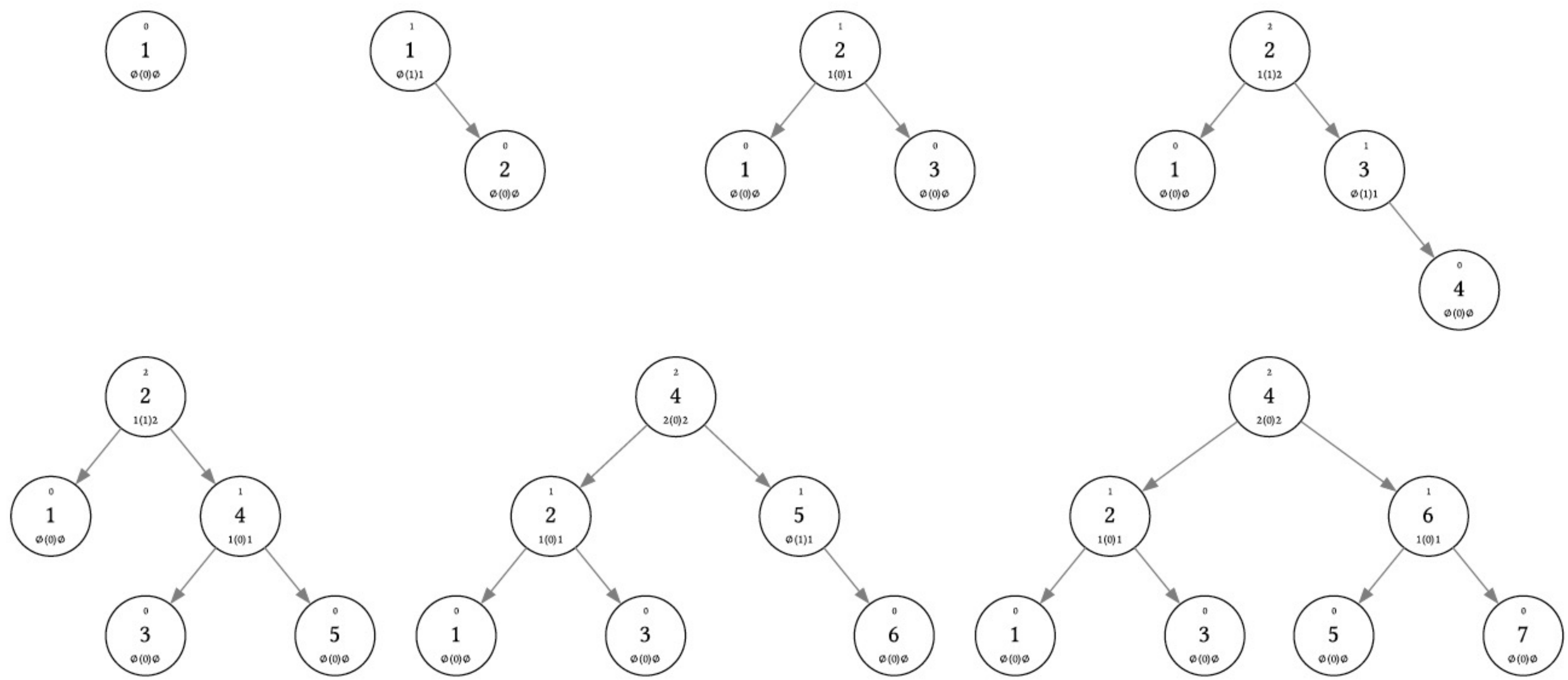
To insert into an AVL tree, there are two steps:

1. Insert the new node where it should be based on BST rules.
2. Rotate to fix any imbalances that may have occurred from that insertion.

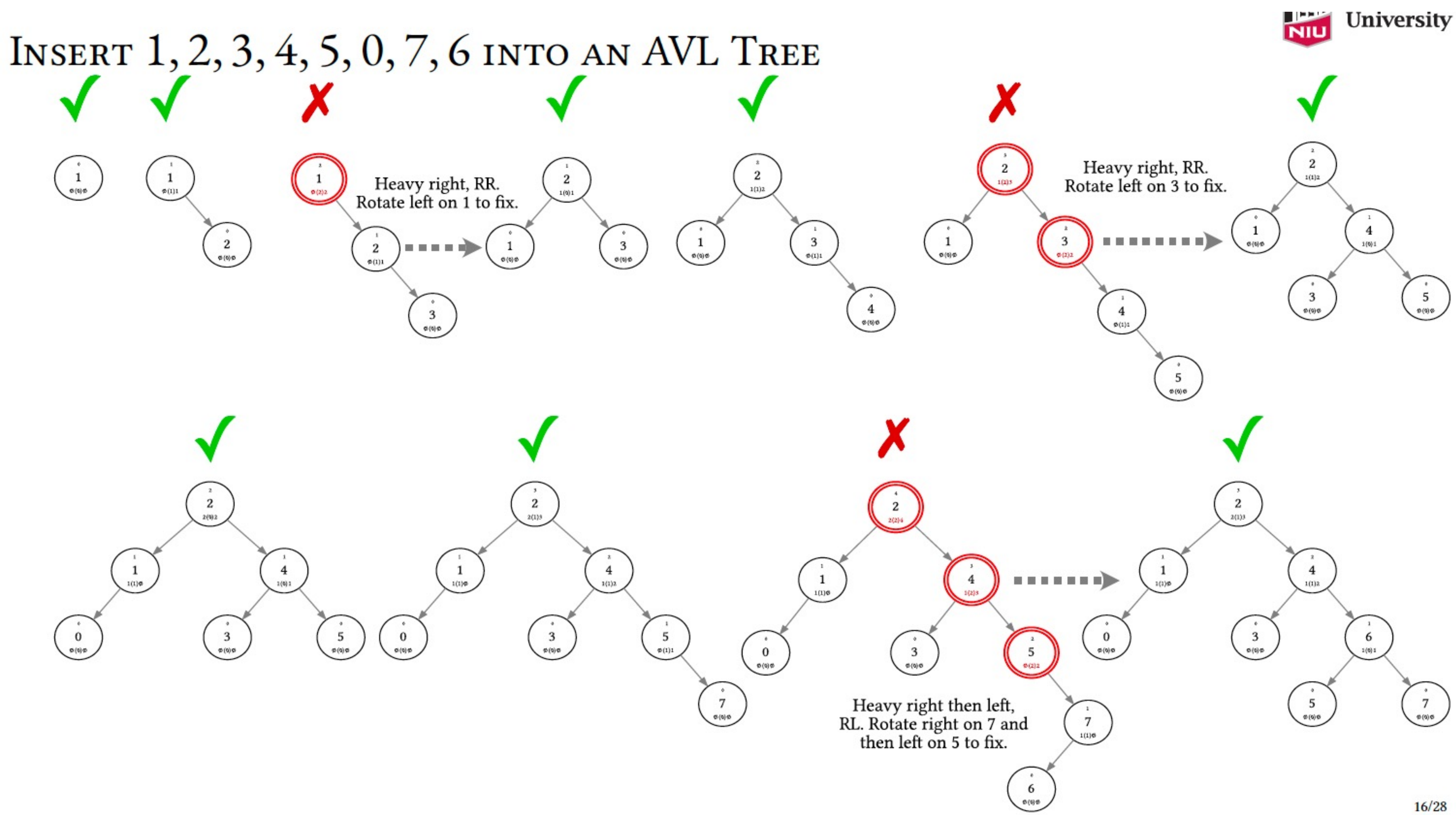


Example 1

INSERT 1, 2, 3, 4, 5, 6, 7 INTO AN AVL TREE



Example 2



INSERT 1, 2, 3, 4, 5, 0, 7, 6 INTO AN AVL TREE
University
Heavy right, RR.
Rotate left on 1 to fix.
Heavy right, RR.
Rotate left on 3 to fix.
Heavy right then left,
RL. Rotate right on 7 and
then left on 5 to fix.
16/28

Example 3

INSERTING 20, 40, 60, 10, 15

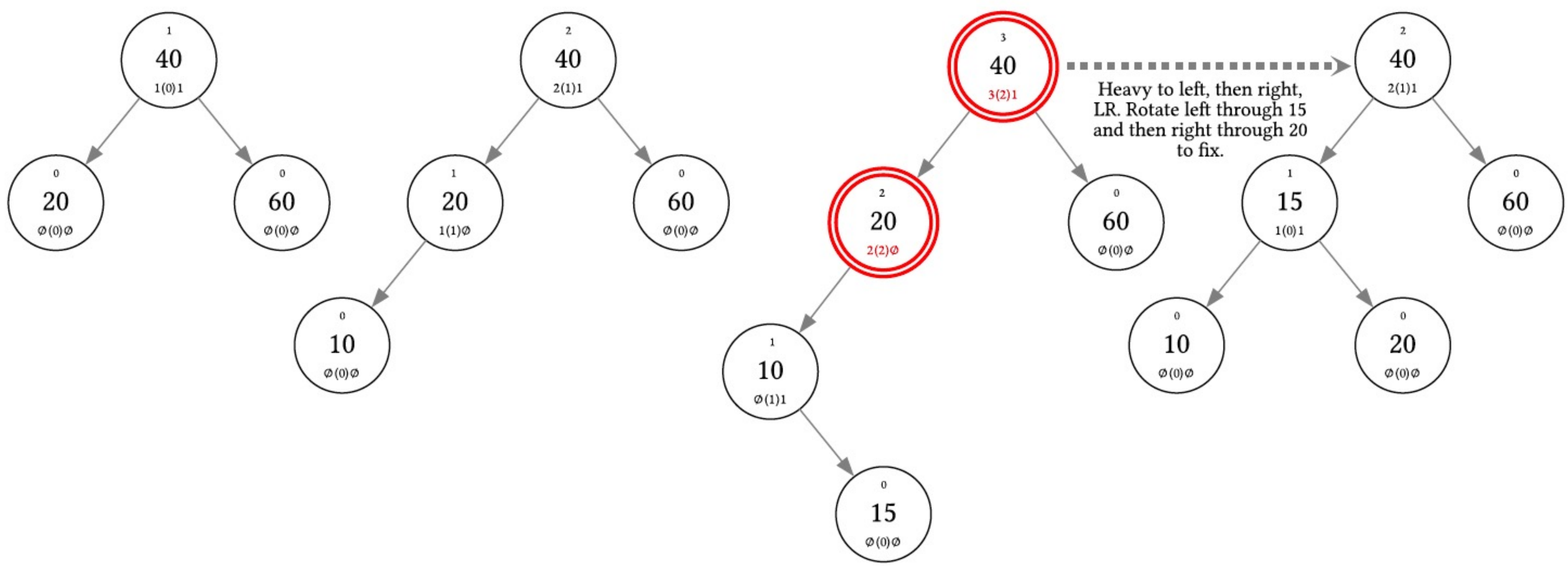
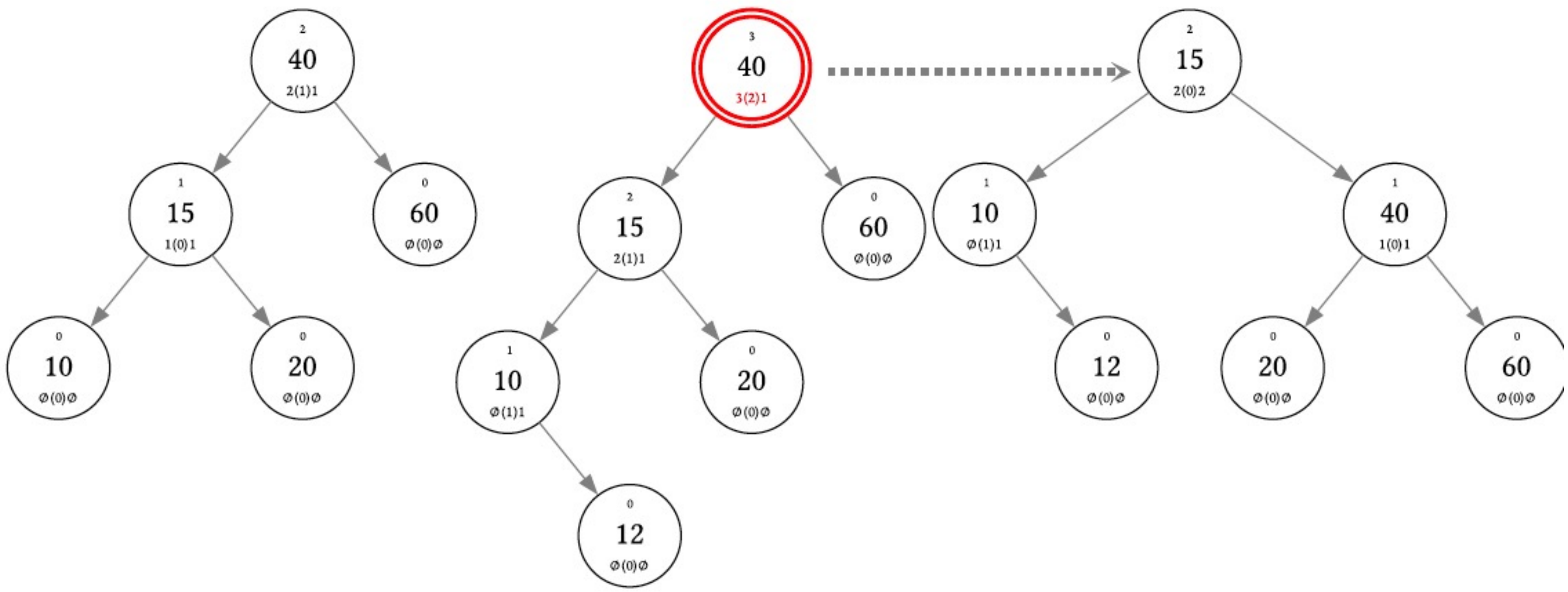


Figure 12: Example of insertion (with rotation): 20, 40, 60, 10, 15

INSERTING 20, 40, 60, 10, 15, 12



Inserting 20, 40, 60, 10, 15, 12, 11

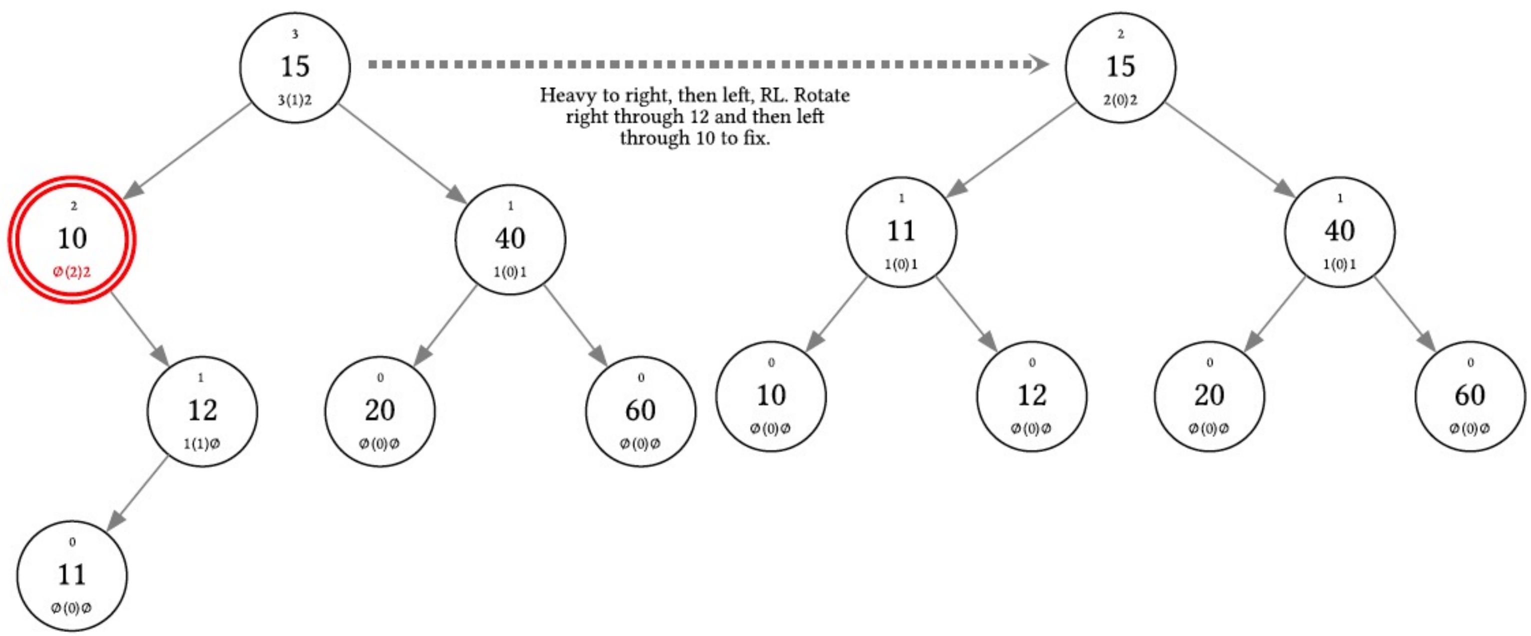
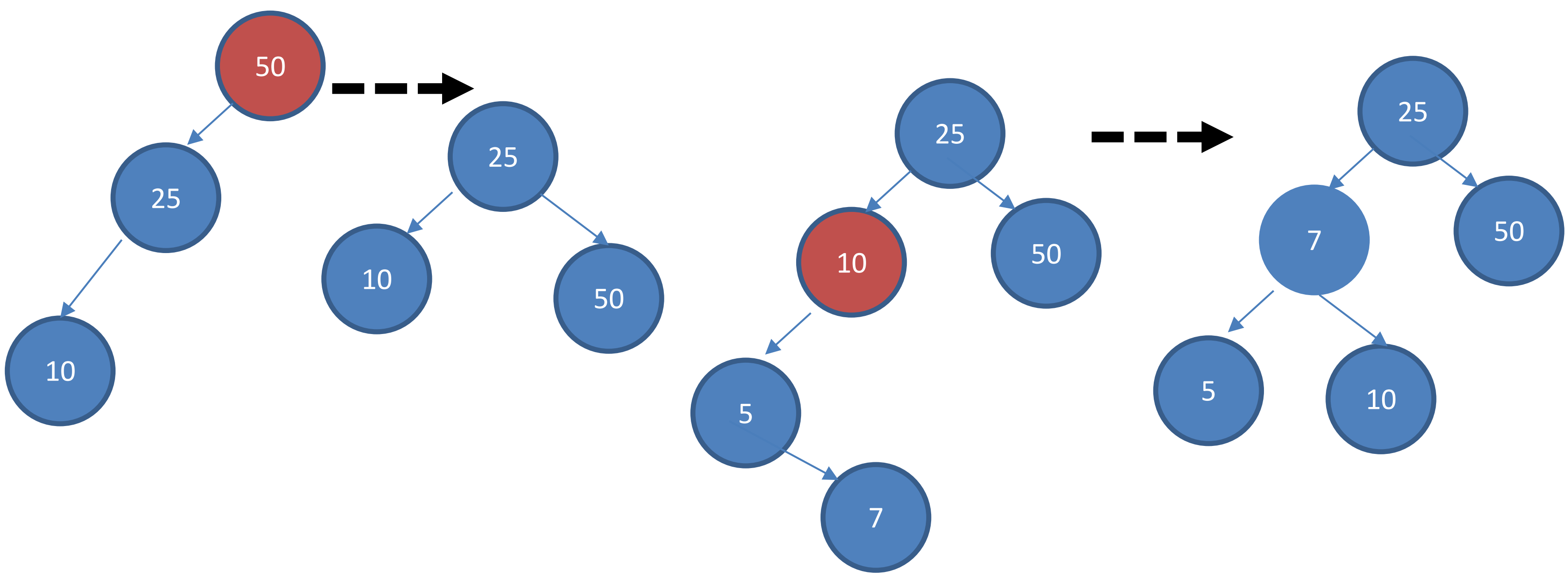


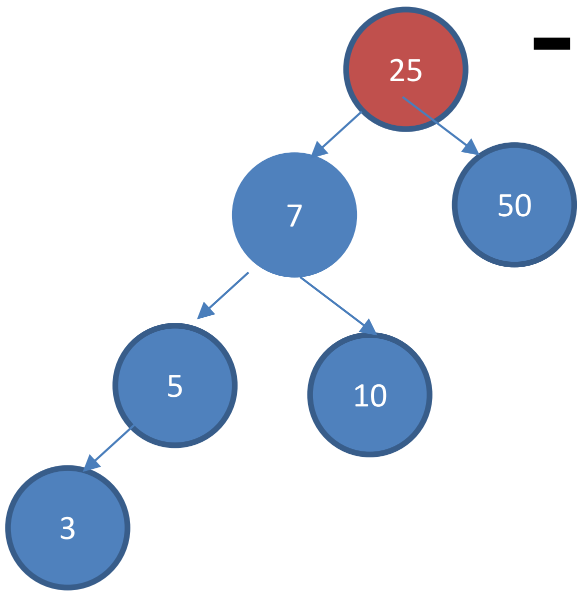
Figure 14: Example of insertion (with rotation) 20, 40, 60, 10, 15, 12, 11

Example 4

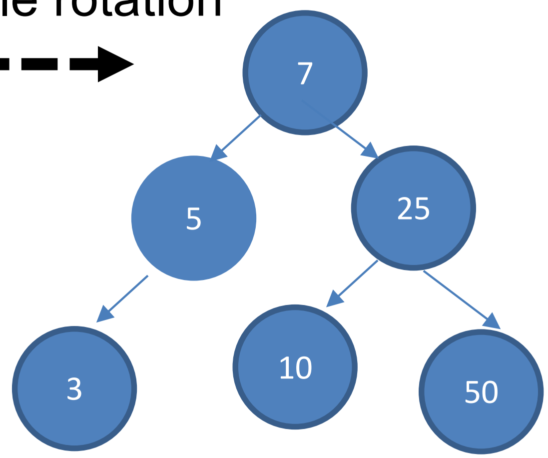
- Insert 50, 25, 10, 5, 7, 3 30, 20, 8, 15

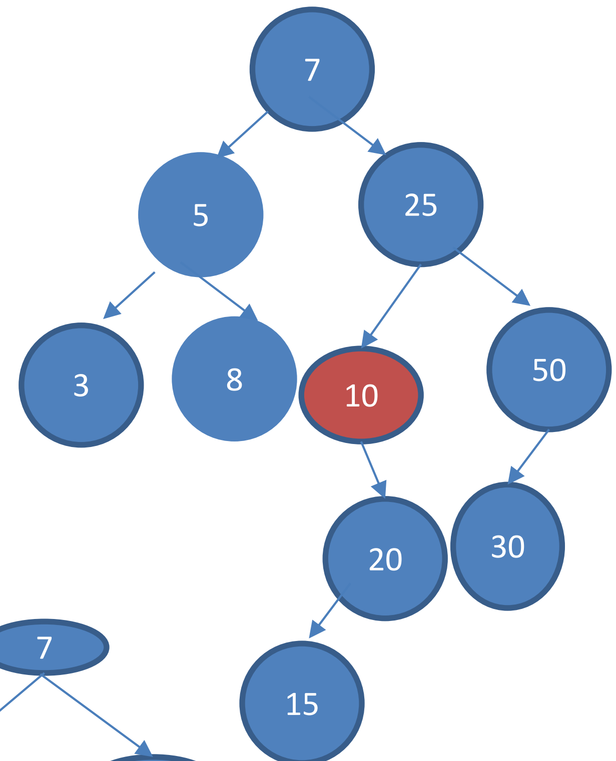


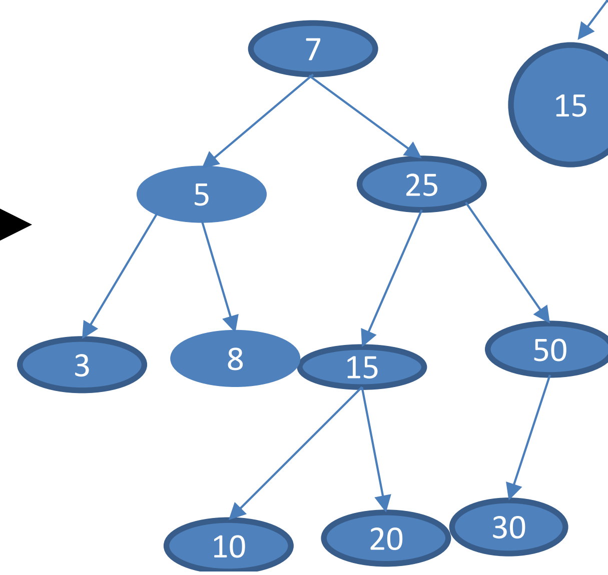
Insert 50, 25, 10, 5, 7, 3 30, 20, 8, 15



Single rotation







Student Exercise: Building an AVL tree using insertion and rotation

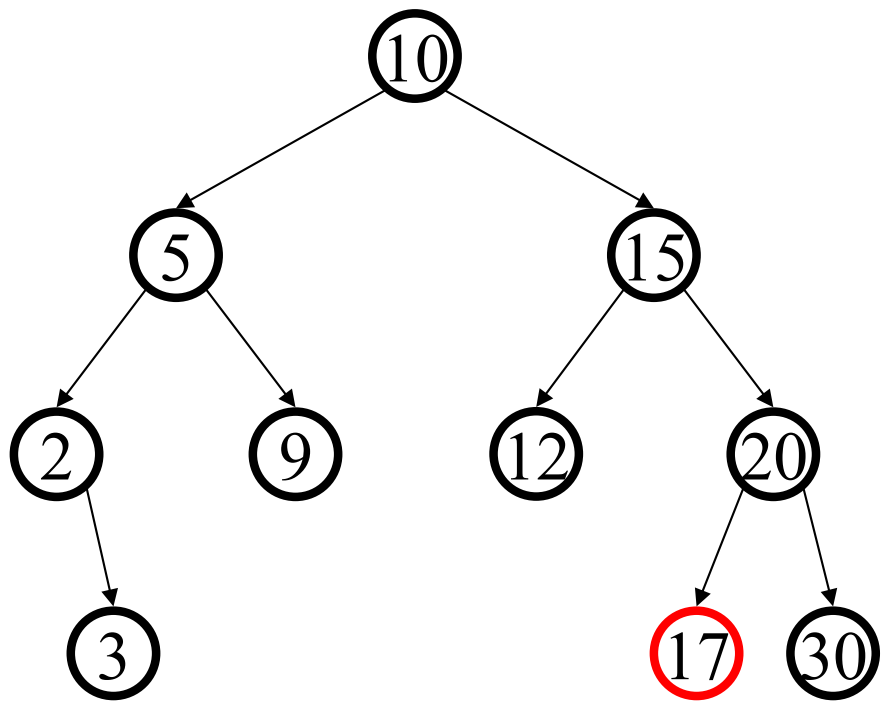
- Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 into an AVL tree
- Insert 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 into an AVL tree
- Insert 10, 2, 11, 1, 3, 9, 8, 4, 5, 7, 6 into an AVL tree

AVL Tree Deletion

- First: Normal BST deletion
  - 0 Children: just delete it
  - 1 child: delete child to parent
  - 2 children: copy predecessor/successor in your place, delete predecessor/successor
- If the resulting tree is no longer an AVL tree, check which node's balance factor(or height) have changed:
  - 0 children or 1 child: check the path from the deleted node to root.
  - 2 children: check from predecessor/successor to root.
  - Do rotation similar as insertion if needed. If for the imbalanced subnode, the left subtree's height is the same as the right subtree, can do either single or double rotation.
  - Different from insertion, the checking needs to be done all the way to root, due to the possibility of "propagation".

Deletion (A Really Easy Case)

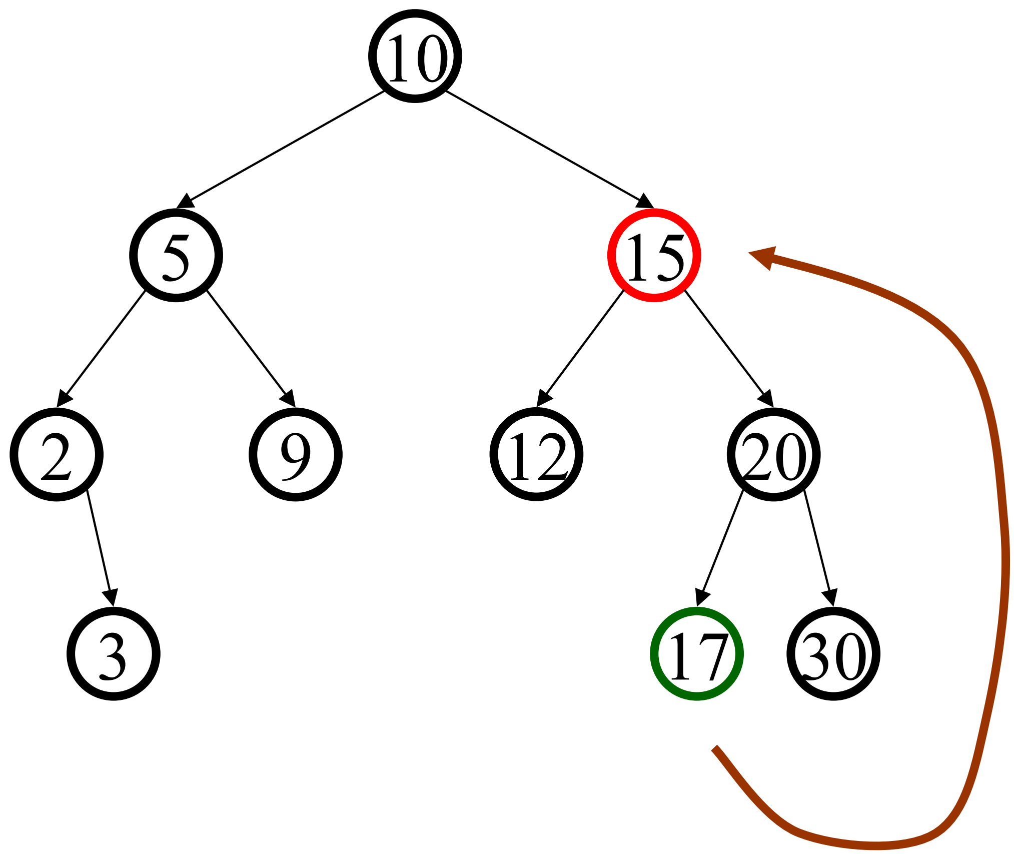
Delete(17)



10 (5 [2 [null, 3] , 9], 15[12 , 20 [17, 30]])

Deletion (A Pretty Easy Case)

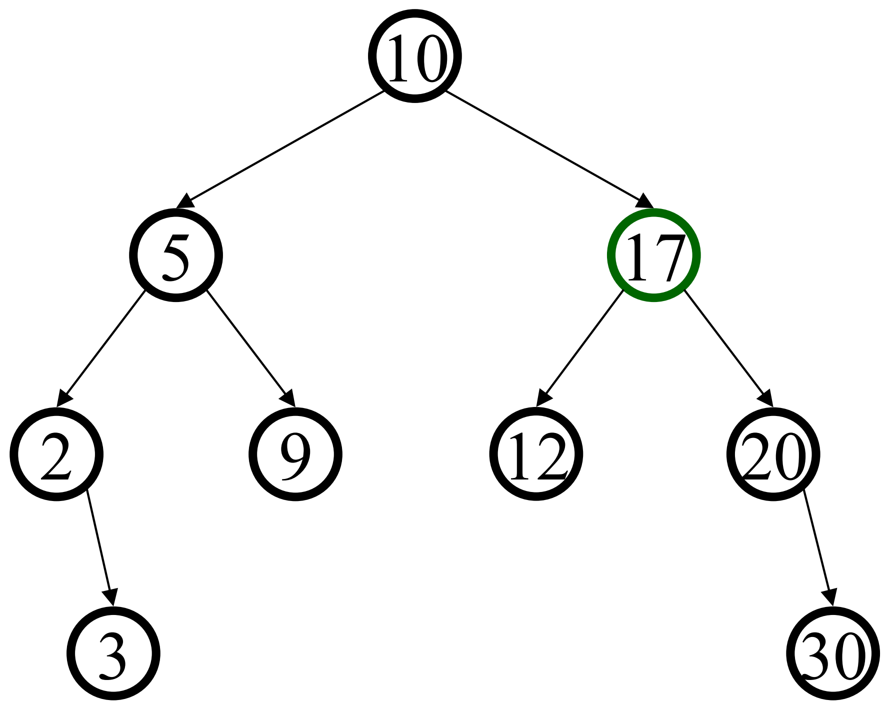
Delete(15)



10 (5 [2 [null, 3] , 9], 15[12 , 20 [17, 30]])

Deletion (Pretty Easy Case *cont.*)

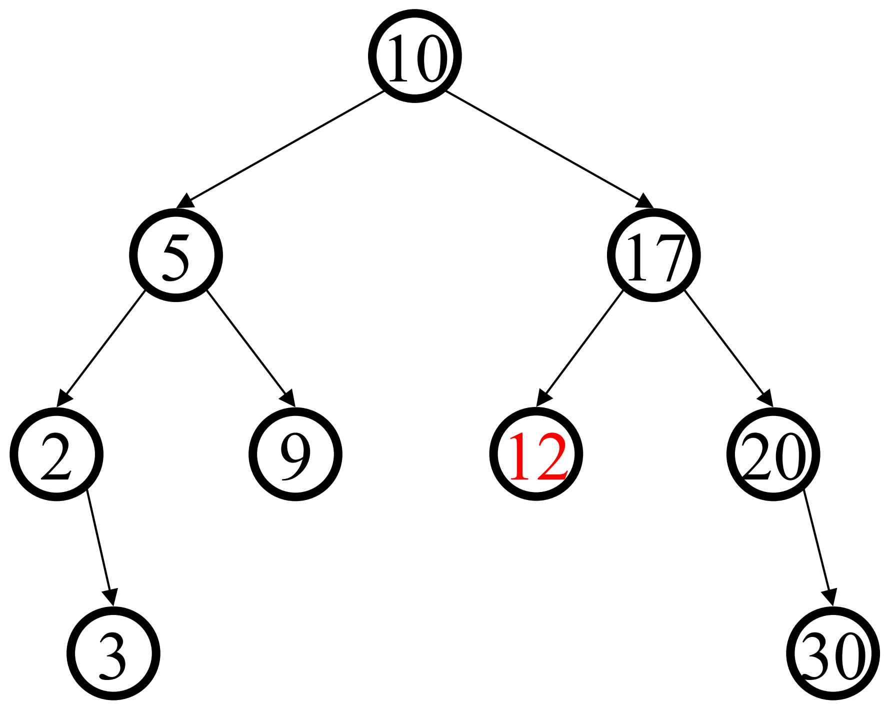
Delete(15)



10 (5 [2 [null, 3] , 9], 17[12 , 20 [null, 30]])

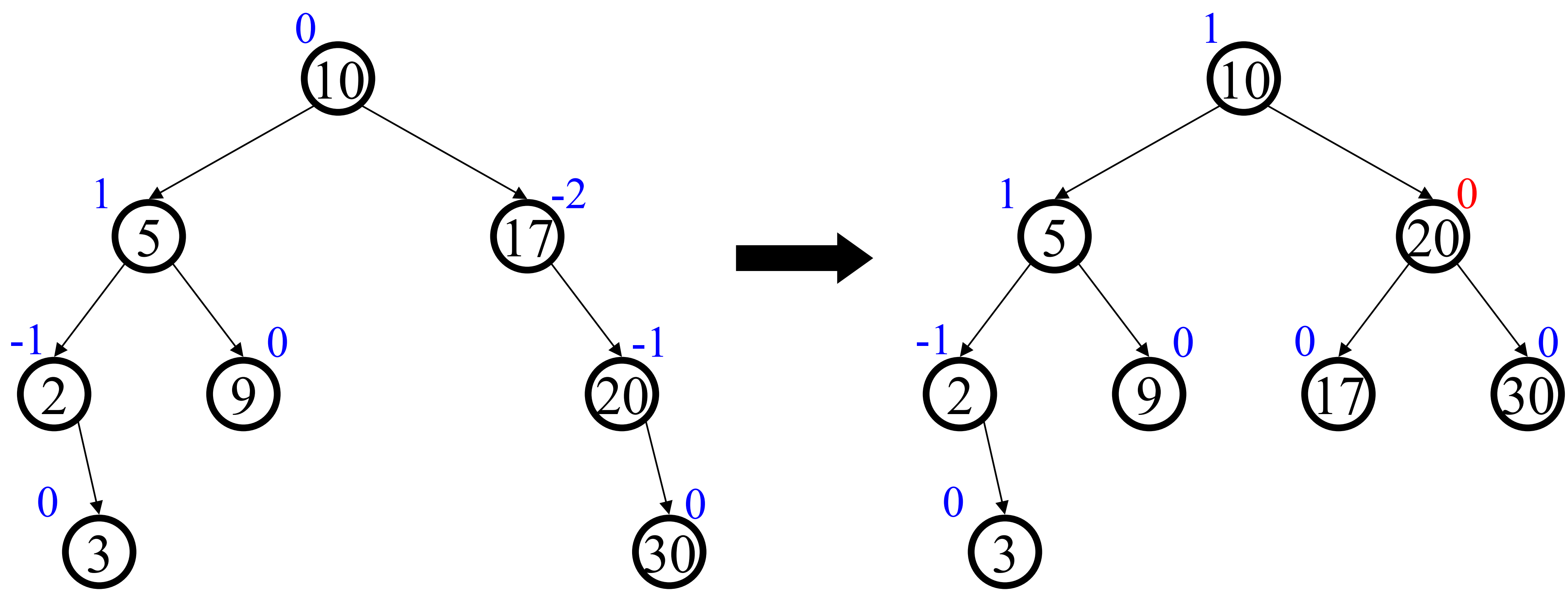
Deletion (Hard Case #1)

Delete(12)



10 (5 [2 [null, 3] , 9], 17[12 , 20 [null, 30]])

Single Rotation on Deletion

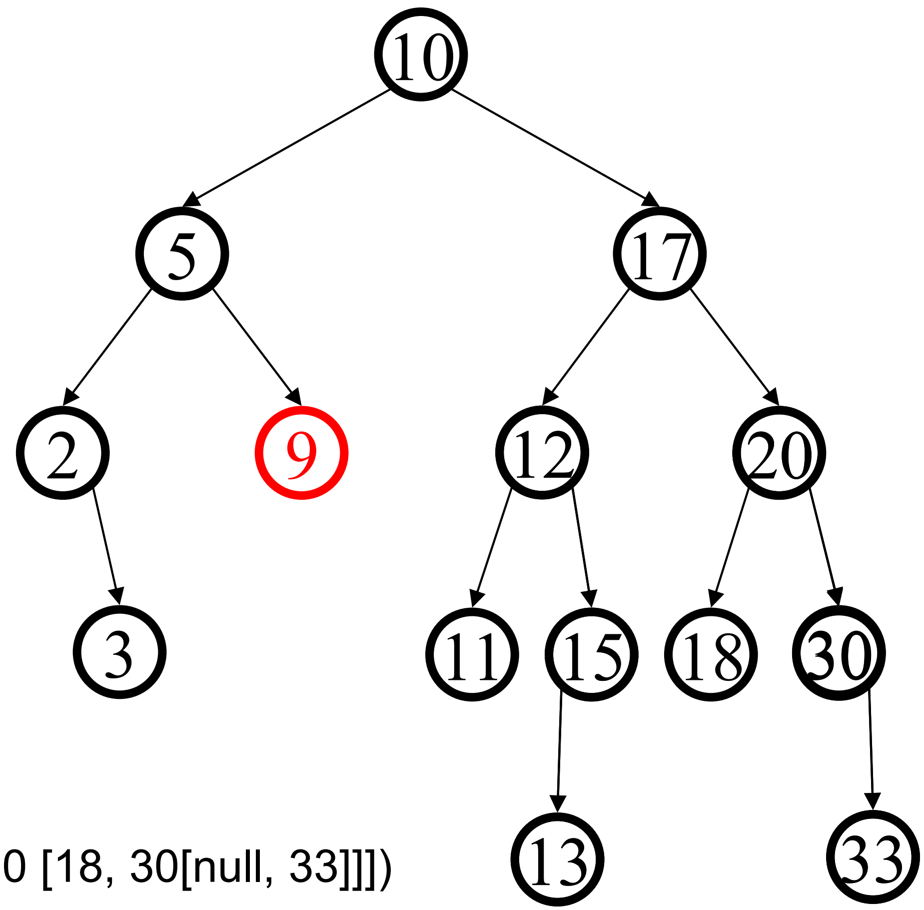


10 (5 [2 [null, 3] , 9], 17[12 , 20 [null, 30]]) →

10 (5 [2 [null, 3] , 9], 20 [17 , 30])

Deletion (Hard Case)

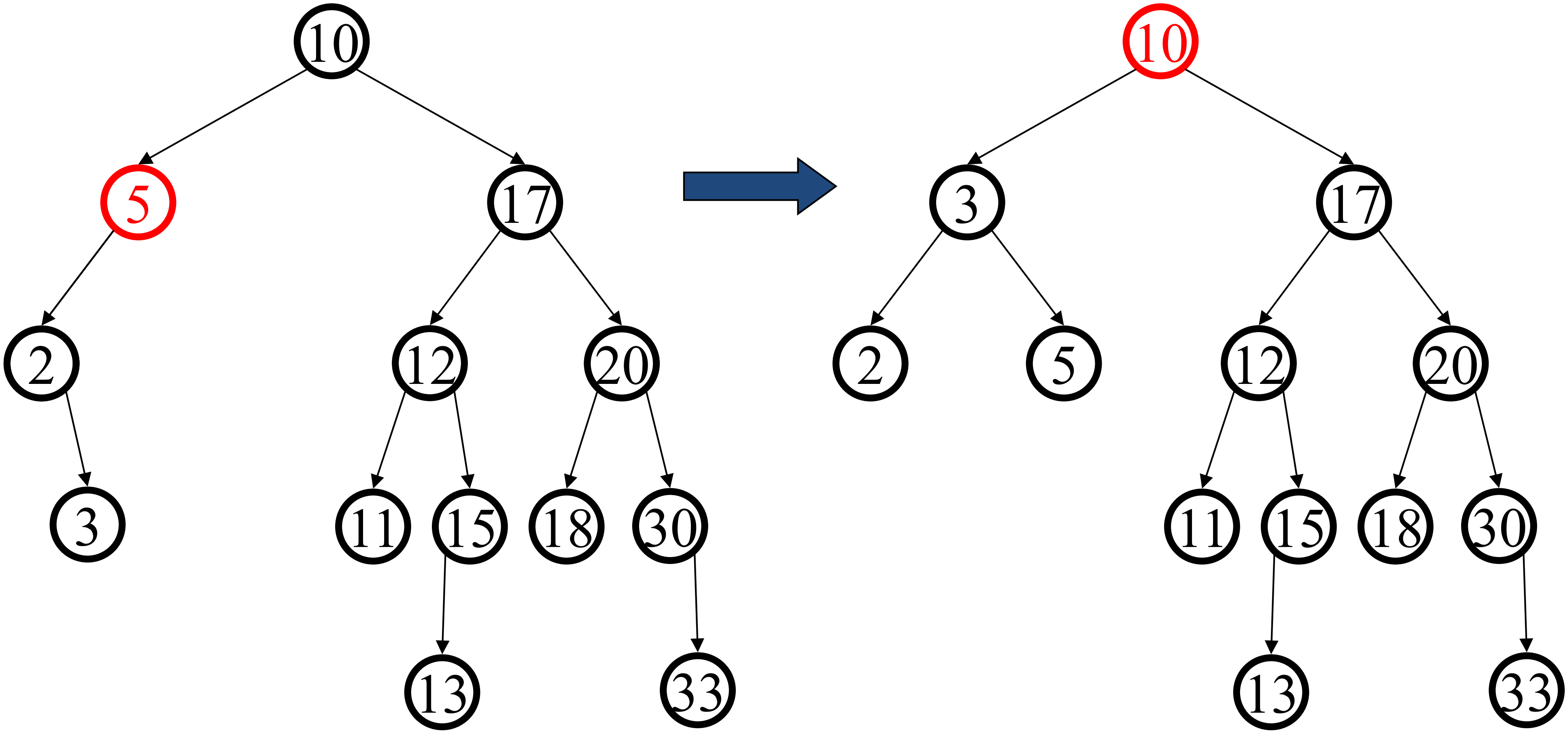
Delete(9)



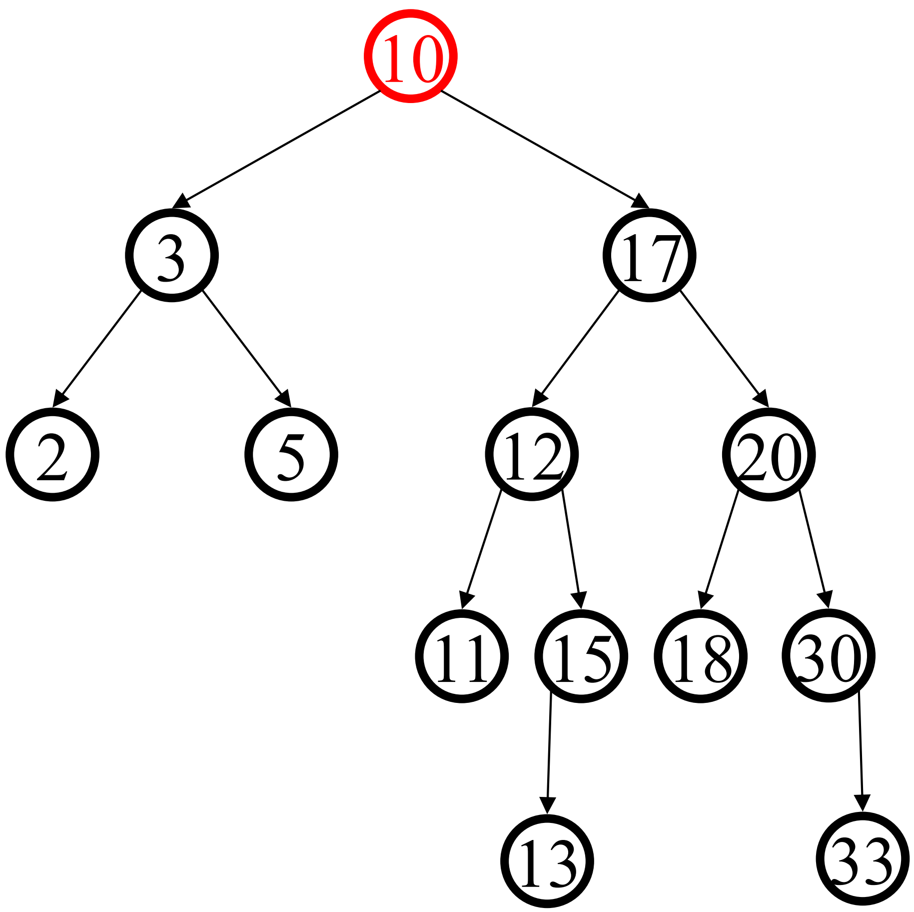
10 (5 [2 [null, 3] , 9], 17[12 [11, 15[13, null]], 20 [18, 30[null, 33]]])

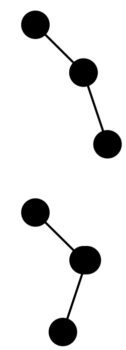
Double Rotation on Deletion

Not finished! Root's B.F. is -2.



Deletion with Propagation

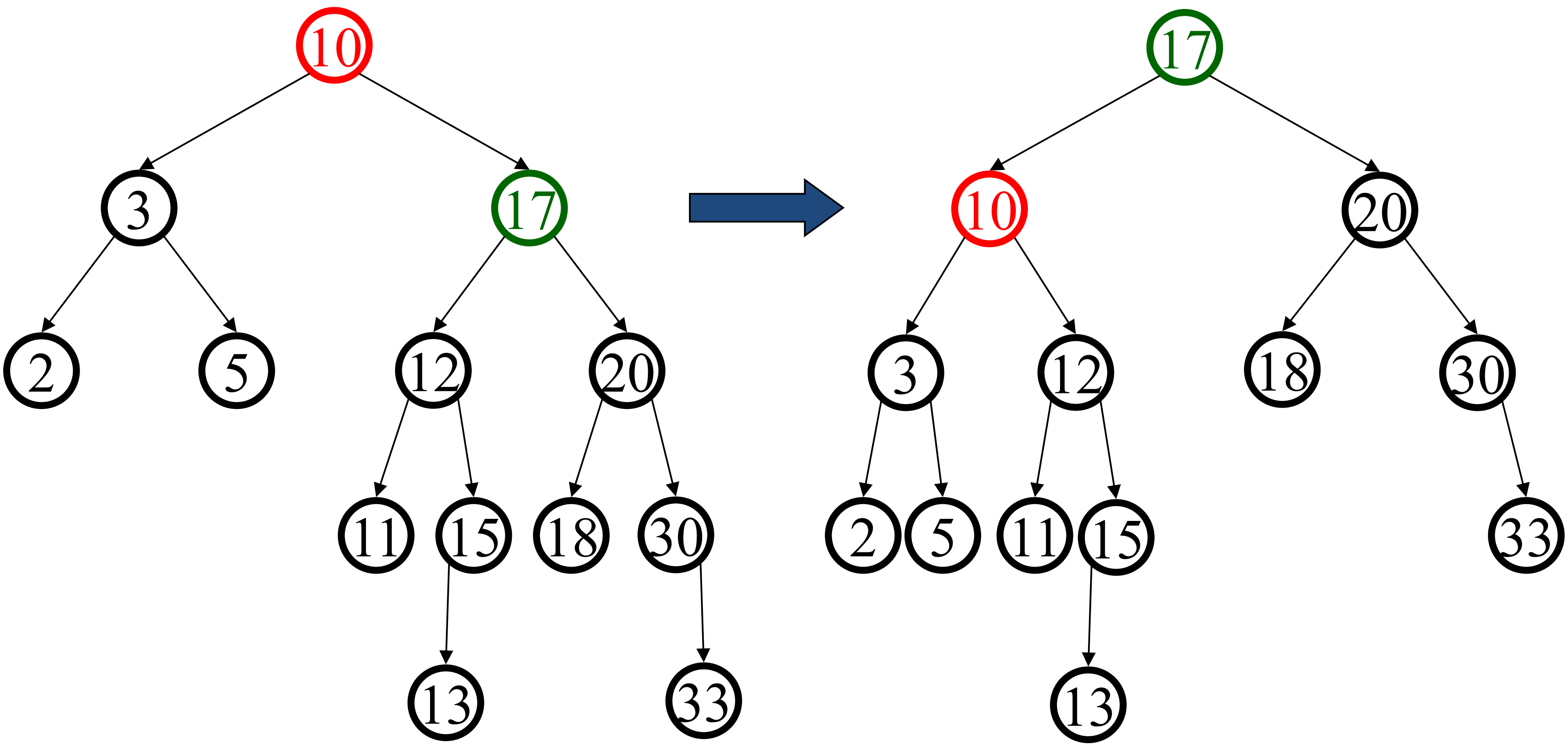




We get to choose whether to single or double rotate!

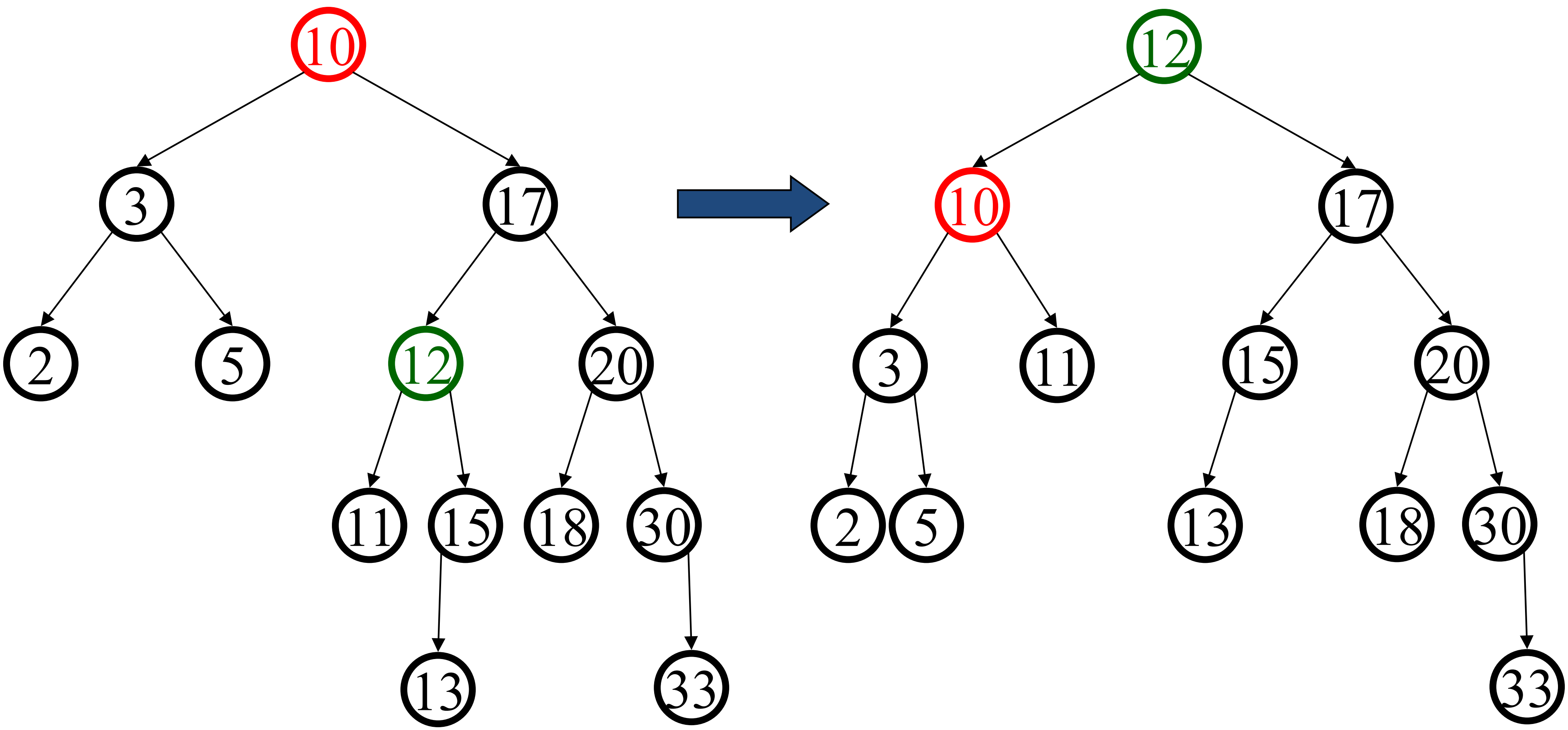
10 (3 [2 , 5], 17[12 [11, 15[13, null]], 20 [18, 30[null, 33]]])

Propagated Single Rotation



10 (3 [2 , 5], 17[12 [11, 15[13, null]], 20 [18, 30[null, 33]]]) → 17 (10 [3[2,5] , 12[11,15[13,null]]], 20[18, 30 [null, 33]])

Propagated Double Rotation



10 (3 [2 , 5], 17[12 [11, 15[13, null]], 20 [18, 30[null, 33]]]) → 12 (10 [3 [2,5], 11] , 17[15[13,null] , 20[18, 30 [null, 33]]])

Time complexity for AVL tree

- Worst case: O(lg(n)) for all operations
- Insertion: insertion + balancing
  - BST insertion itself was a O(lg(N))
  - Balancing is constant (1 or 2 rotation)
- Deletion: balancing: in the worst case, every node on the path needs rotation, another term of O(lg(N)).

Pros and Cons of AVL Trees

Pros:

- All operations guaranteed O(log N)
- The height balancing adds no more than a constant factor to the speed of insertion

Cons:

- Space consumed by height (or B.F.) field in each node
- Slower than ordinary BST on random data

Data structure visualization:

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

AVL:

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL Tree Deletion

We first do the normal BST deletion:

- 0 children: just delete it
- 1 child: delete it, connect child to parent
- 2 children: put predecessor/successor in your place, delete predecessor/successor

After deleting the node , the resulting tree might no longer be an AVL tree. As in the case of insertion into a non-AVL tree, the height at the top could decrease or increase by 1.

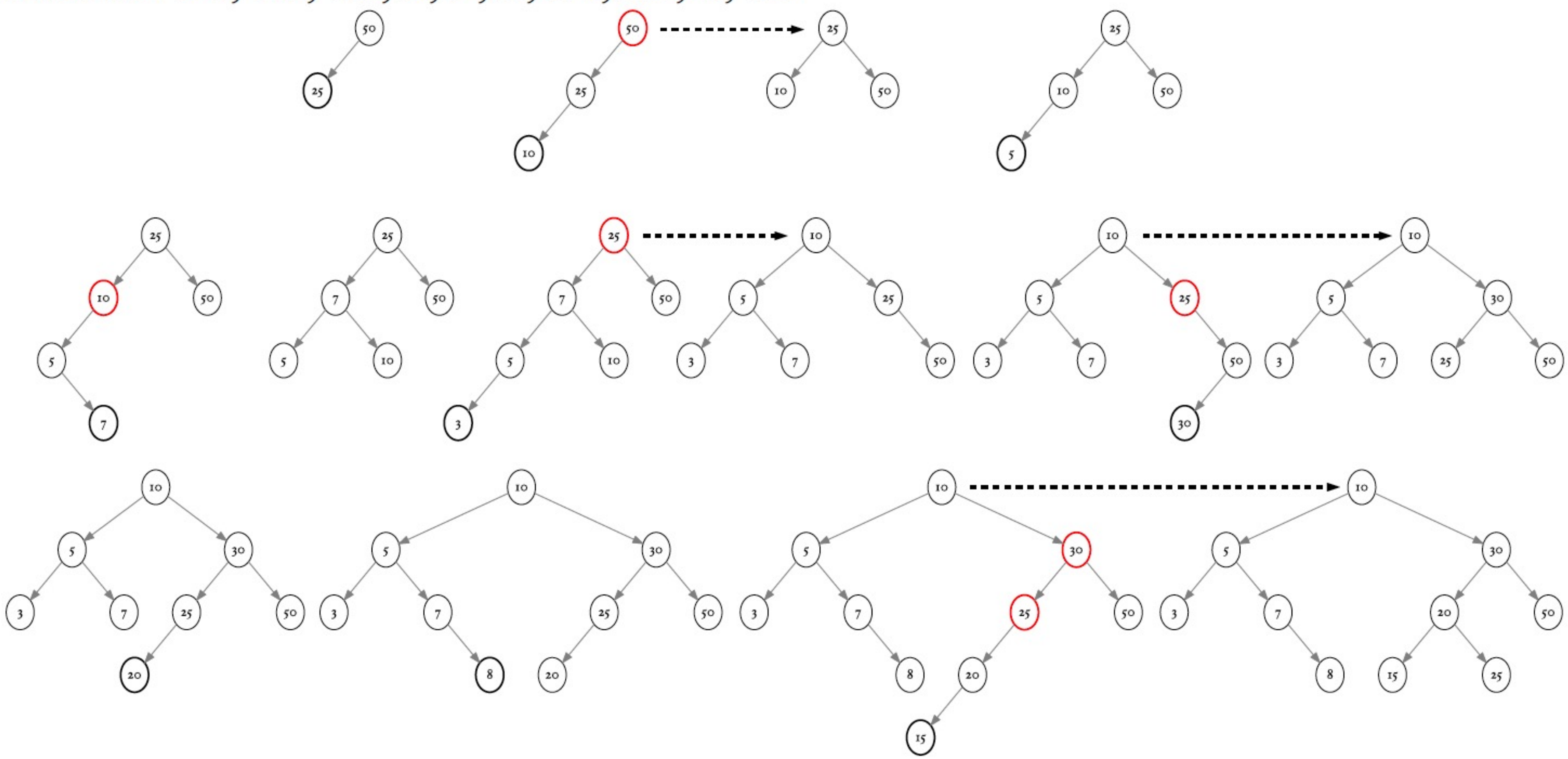
Which nodes' heights may have changed:

- 0 children: path from deleted node to root
- 1 child: path from deleted node to root
- 2 children: path from deleted successor leaf to root

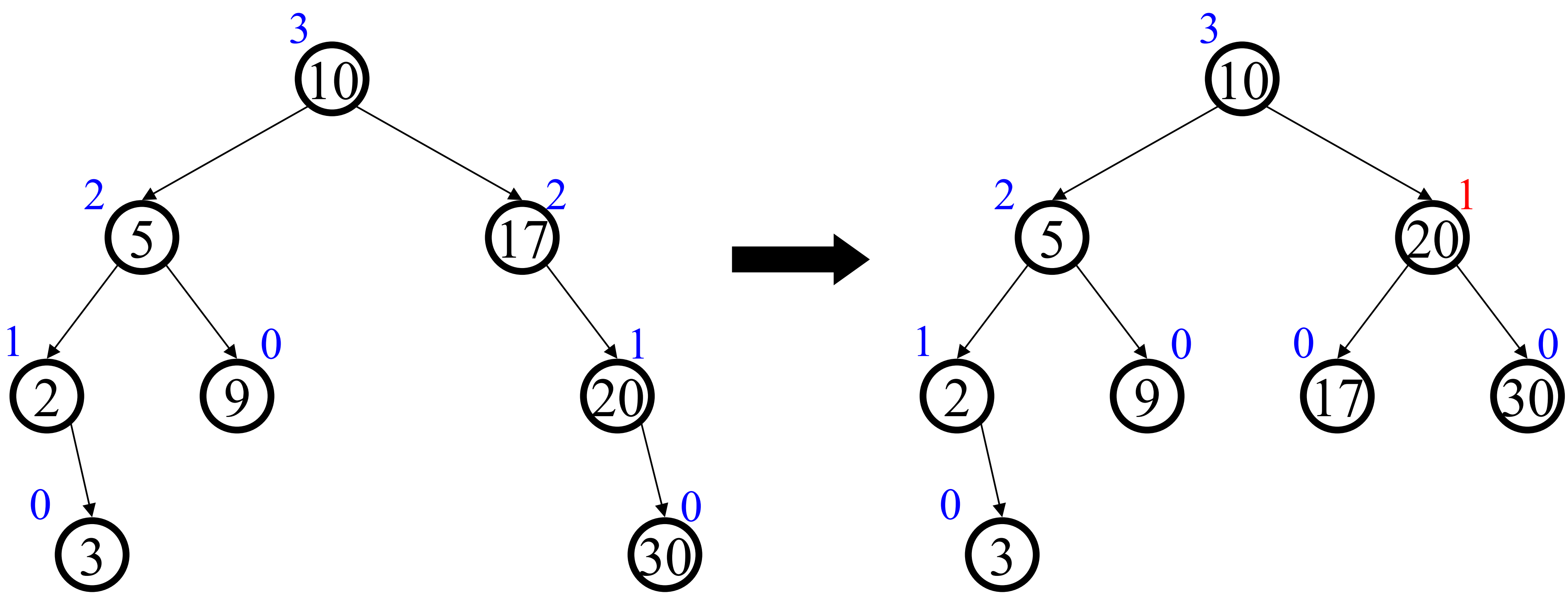
Similar rotations to insert , but in case there is an imbalance at a node and the left subtree height is equal to right subtree height, then perform either rotation (single or double rotate).

Example 4 (has some issue?)

INSERTING 50, 25, 10, 5, 7, 3, 30, 20, 8, 15



Single Rotation on Deletion



10 (5 [2 [null, 3] , 9], 17[12 , 20 [null, 30]]) →

10 (5 [2 [null, 3] , 9], 20 [17 , 30])

Numbers are height of the tree/subtree? Expect the one with root is 0.