Julius-Maximilians-UNIVERSITÄT WÜRZBURG
CAIDAS
WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Balanced Trees (AVL)

Prof. Dr. Goran Glavaš

20.11.2023

Content

- Balanced Binary Search Trees
- AVL Trees

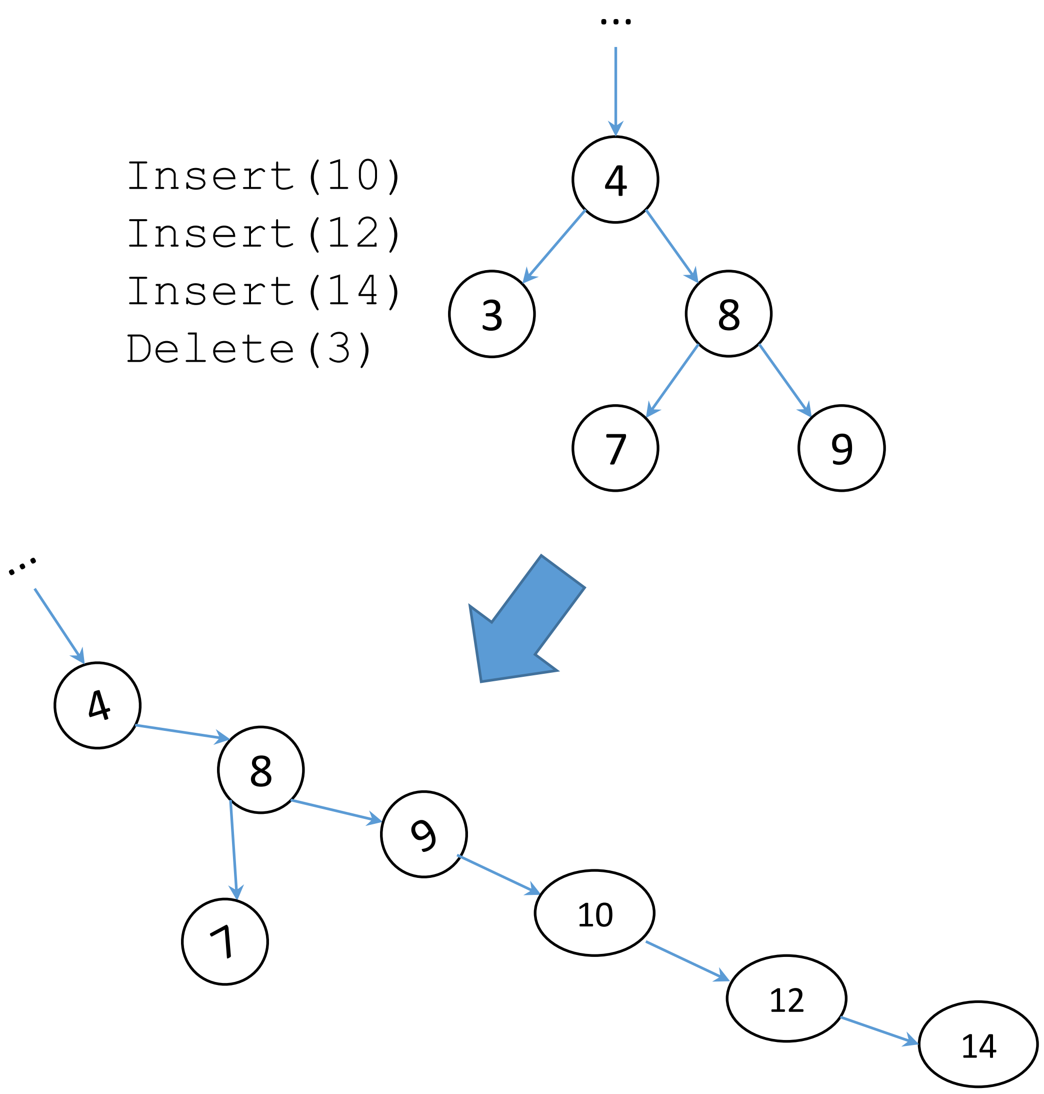
Dynamic Sets – Operations

	Runtime				
Data struct.	Search	Insert	Delete	Min/Max	Pred/Succ
Array	O(n)	O(1)	O(n)	O(n)	O(n)
Linked List	O(n)	O(1)	O(1)	O(n)	O(n)
Hash Table	O(1)	O(1)	O(1)	not possible	not possible
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)
Binary Search Tree	O(h) = O(log n)	O(h)	O(h)	O(h)	O(h)

- **Insert** & **Delete** – **change** the dynamic set (by adding or removing values)
- **Search**, **Min/Max**, and **Pred/Succ** – **query (anfragen)** the dynamic set, but do not change it

Binary Search Tree: **Height/Depth**

- The complexity of all operations on the BST is O(h)
- If the BST is **balanced** $h \approx \log_2 n$
- Frequent insertions and deletions can disturb the balance of the tree



- **The height/depth drastically increases**
  - Extreme: BST reduced to a **linked list**
  - Search efficiency gains **lost**
  - Need to **re-balance** the tree. **Q:** How?

Re-Balancing the Binary Search Tree

- The standard **binary search tree** and its insert and deletion operations provide **no guarantee** that the tree will remain **balanced**

- If BST becomes too unbalanced,

  h can become much larger than log n
  - Consequently **T(n) > O(log n)** for all operations
  - Working with tree becomes much slower

- **Solution #1**: re-balance the tree
  1. Create a **sorted array** via `inorder_walk` → **O(n)**

```
inorder_array(T)
  A.Size = T.Size
  A.Length = 0
  inorder_walk(T.root, A)
  return A
```

```
inorder_walk(x, A)
  if x != null
    inorder_walk(x.left, A)
    A.Length = A.Length + 1
    A[A.Length - 1] = x.key
    inorder_walk(x.right)
```

Re-Balancing the Binary Search Tree

- The standard **binary search tree** and its insert and deletion operations provide **no guarantee** that the tree will remain **balanced**

- If BST becomes too unbalanced,

  h can become much larger than log n
  - Consequently **T(n) > O(log n)** for all operations
  - Working with tree becomes much slower

- **Solution #1**: re-balance the tree
  1. Create a **sorted array** via `inorder_walk` → **O(n)**
  2. Create a binary tree recursively from a sorted array → **O(n)**

```
tree_from_array(A)
  T.root = null
  array_to_tree(A, 0, A.Length - 1, T.root)
  return T


array_to_tree(A, p, r, x)
  n = r - p + 1
  if n % 2 == 1
    q = p + n//2
  else
    q = p + n/2 - 1

  x = new node
  x.key = A[q]
  x.left = null
  x.right = null

  if n > 1
    l = array_to_tree(A, p, q-1, x.left)
    l.parent = x
    r = array_to_tree(A, q+1, r, x.right)
    r.parent = x

  return x
```

Binary Search Tree: **Height/Depth**

- How do we **balance out** an unbalanced binary search tree?

  1. Construct the sorted array from the BST – **O(n)**
  2. Build a new BST from the sorted array (recursively) – **O(n)**

  If n is large, re-balancing this way is expensive and cannot be done frequently

- How to **maintain** balanced BSTs?
  - Make sure that after every insert / delete, the tree is (more or less) **balanced**

Self-balancing BSTs

- We want to have a guarantee that query operations cost **O(log n)**
  - **`Search`**, **`Min/Max`**, **`Pre/Succ`**

- **Self-balancing** binary search trees
  - Number of variants, we'll see one of the two most commonly used
  - AVL trees
  - Red-black trees

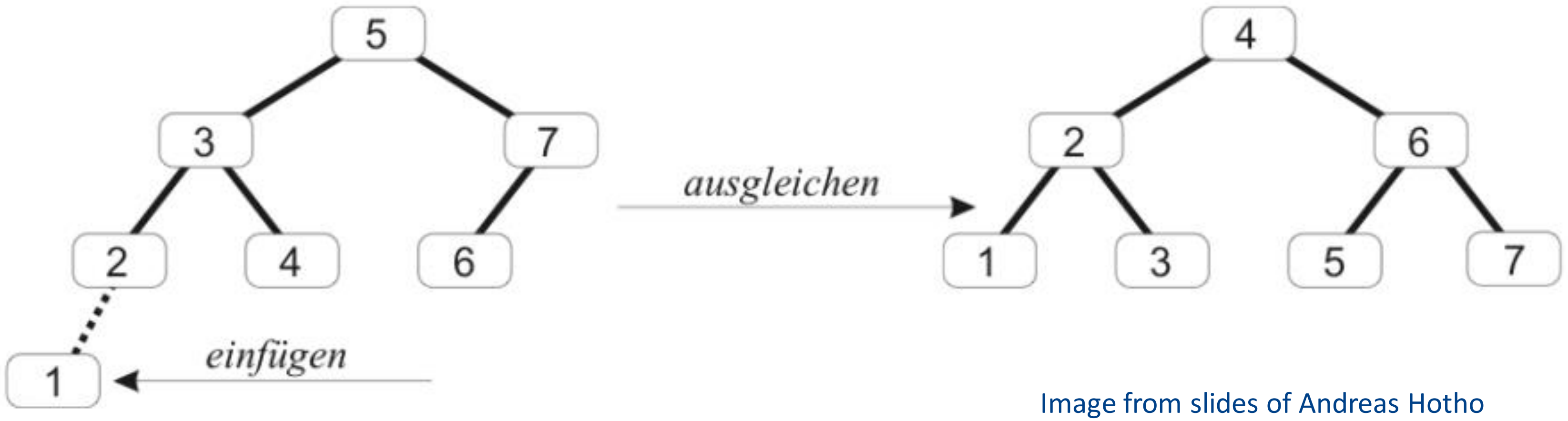


Image from slides of Andreas Hotho

Content

- Balanced Binary Search Trees
- AVL Trees
  - Insertion
  - Deletion

AVL Trees

- First self-balancing binary search tree
  - Named after the inventors: *Georgy Adelson-Velsky* and *Evgenii Landis*

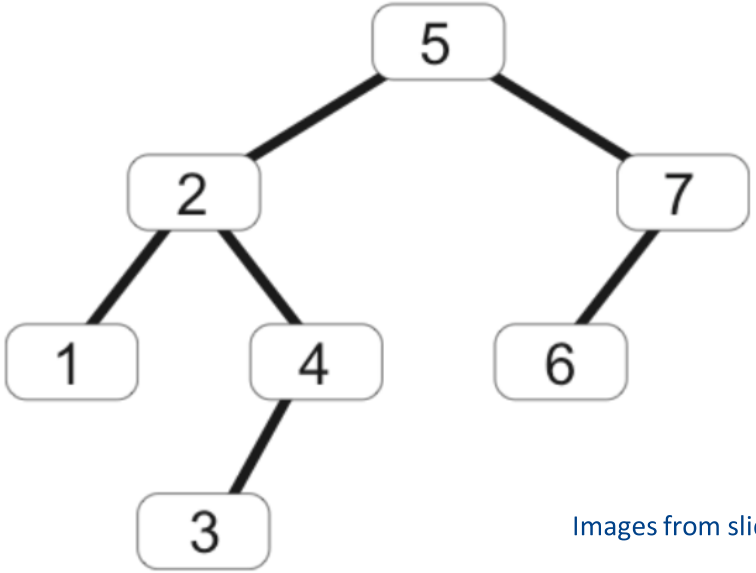
AVL tree property

Core **property** (guiding/operatring principle) of **AVL trees** is given as follows:

for any two sibling nodes x and y, the difference in their respective tree height (i.e., tree heights at which x and y appear), must not be more than 1, $|height(x) - height(y)| \leq 1$.

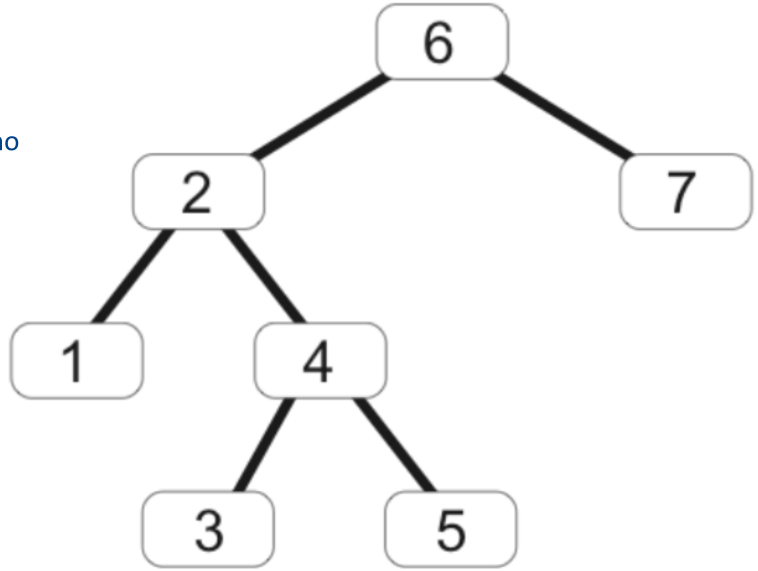
  - Put differently, for each of the non-leaf nodes, the difference in height between its left and right subtree must be at most 1.

AVL Trees



Is this an AVL tree
(satisfies the AVL property)?

Images from slides of Andreas Hotho



Is this an AVL tree
(satisfies the AVL property)?

AVL Trees

- It's still a binary search tree – just a **balanced** one
- Query operations: `Search`, `Max/Min`, `Pred/Succ`
  - Nothing changes in the algorithms for these operations
- `Insert` and `Delete` need to be modified
  - As they can violate the AVL property of the tree
- **Formalization**
  - **Balance factor** – difference between the heights of the left and right subtree

bf(x) = height(x.right) – height(x.left),

  - For any x, **bf(x)** must be in the set **{-1, 0, 1}**

AVL Trees: **Insertion**

- **Balance factor** – difference between the heights of the left and right subtree

  bf(x) = height(x.left) – height(x.right), for any x, **bf(x)** must be in the set **{-1, 0, 1}**

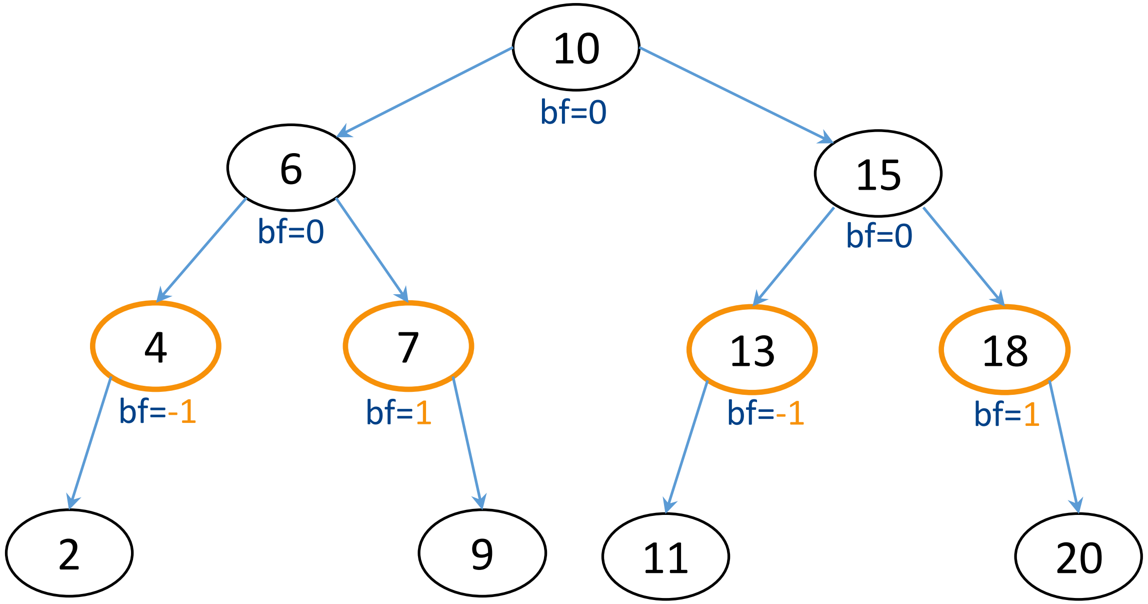
- Insertion in AVL trees:
  1. Insert the new node x as you normally would in regular BST
  2. Fix the AVL property of the nodes for which it has been violated

  - **AVL Violation**: if **bf(y)** becomes -2 or 2 for some node y
  - **Note**: when we add a new node x, **bf(y)** may change only for the nodes y that are the **ancestors of** x – nodes on the path from x to the root

AVL Trees: **Insertion**

- Let's take a look at **all** possible cases that could violate the AVL balancing property

- **Q:** when can a violation occur?
  - New level x added <u>in the path</u> of a node y for which **bf(y)** ∈ {-1, 1}



AVL Trees: **Left-left case (right rotation)**

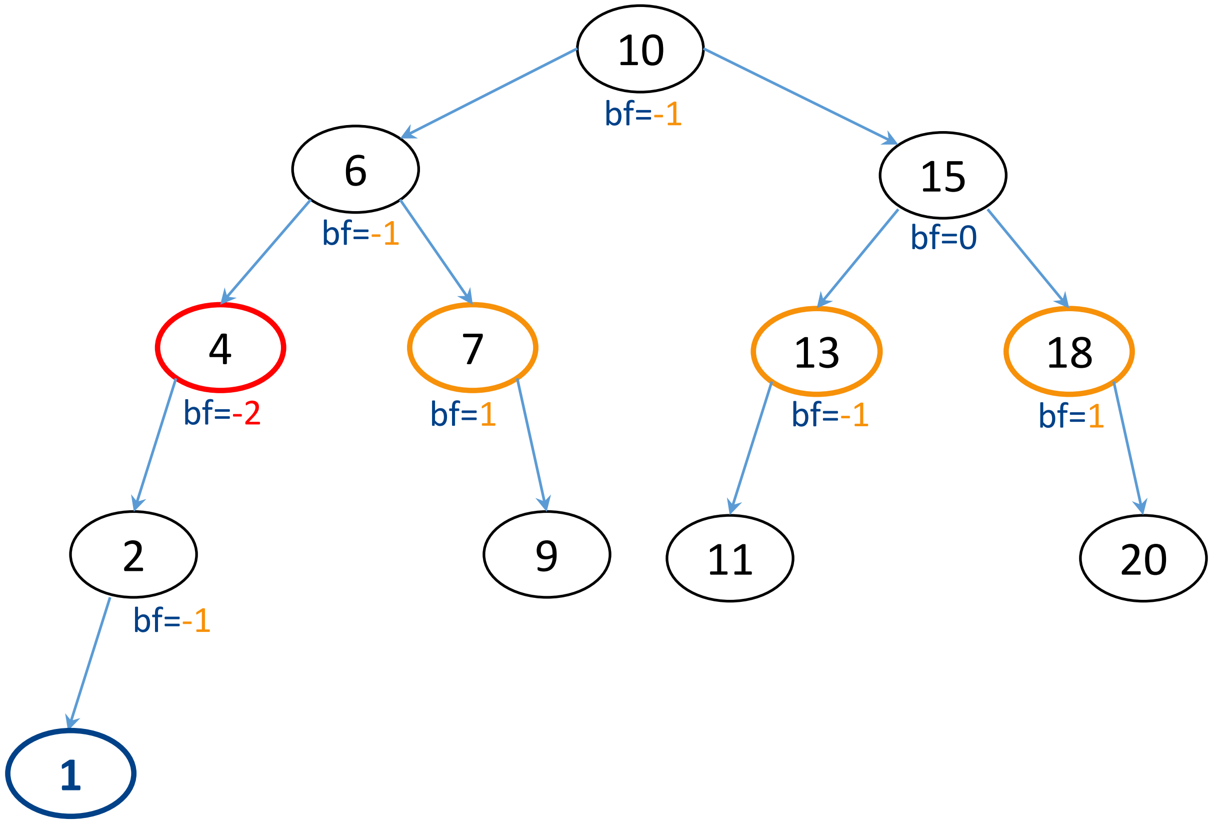
- Let's take a look at **all** possible cases that could violate the AVL balancing property

- **Case #1**: left-left

  `insert(T, 1)`

  - Node 4 (grandparent of the inserted node) violates the AVL property
  - How to restore it?

  - **Right Rotation**
    - **Rotation root (rr):** node with bf -2 (node 4)
    - **Rotation pivot**: child of **rr** with bf -1 (node 2)
    - **rr** becomes the **right** child of the **pivot**, and **pivot** goes where **rr** was



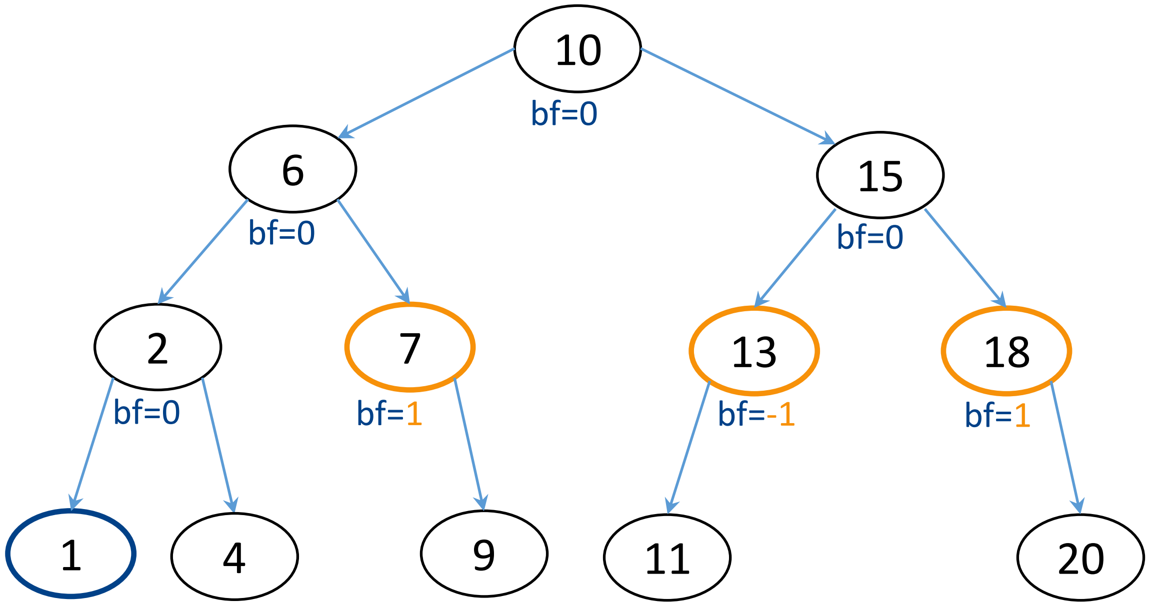
AVL Trees: **Left-left case (right rotation)**

- Let's take a look at **all** possible cases that could violate the AVL balancing property

- **Case #1**: left-left

```
insert(T, 1)
```

- **Right Rotation**
  - **Rotation root (rr):** node with bf -2 (node 4)
  - **Rotation pivot**: child of **rr** with bf -1 (node 2)
  - **rr** becomes the **right** child of the **pivot**, and **pivot** goes where **rr** was



AVL Trees: **Right-right case (left rotation)**

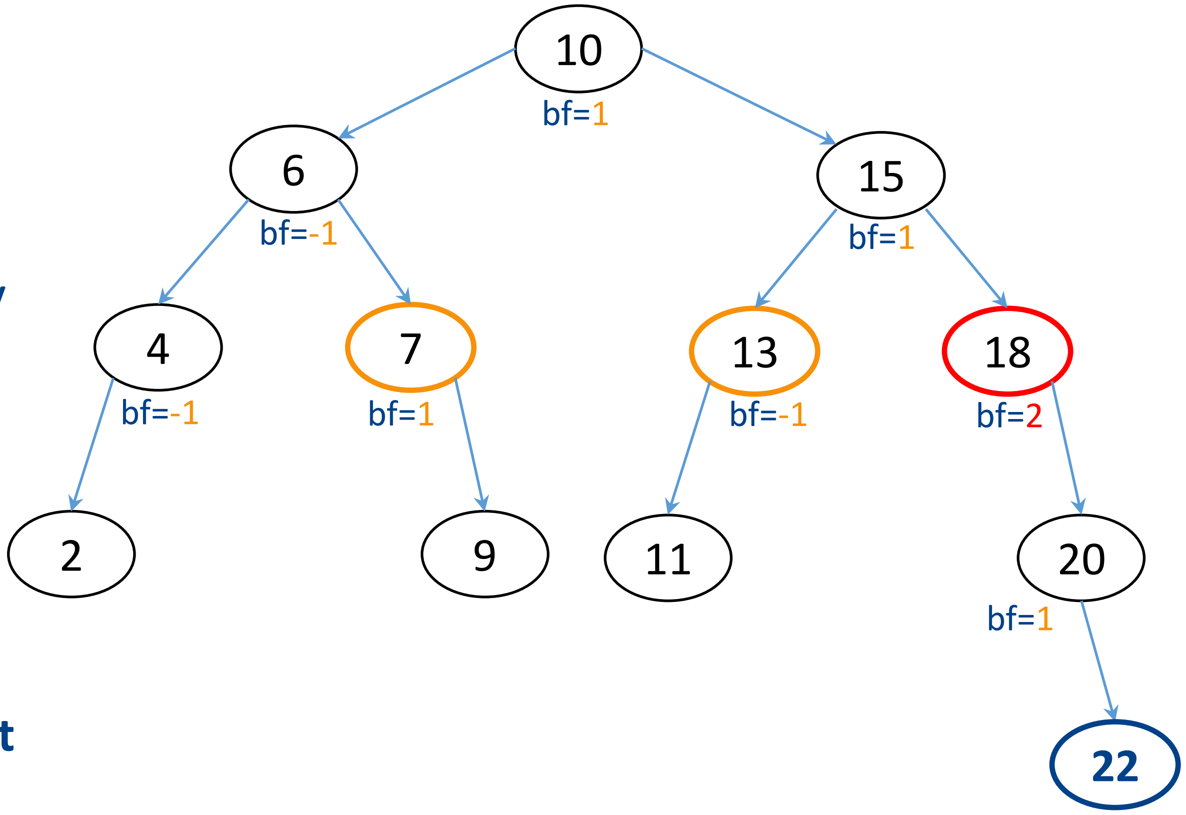
- Let's take a look at **all** possible cases that could violate the AVL balancing property

- **Case #2**: right-right

  `insert(T, 22)`

  - Node 18 (grandparent) violates AVL property
  - How to restore it?

  - **Left Rotation**
    - **Rotation root (rr):** node with bf 2 (node 22)
    - **Rotation pivot**: child of **rr** with bf -1 (node 20)
    - **rr** becomes the **left** child of the **pivot**, and **pivot** goes where **rr** was



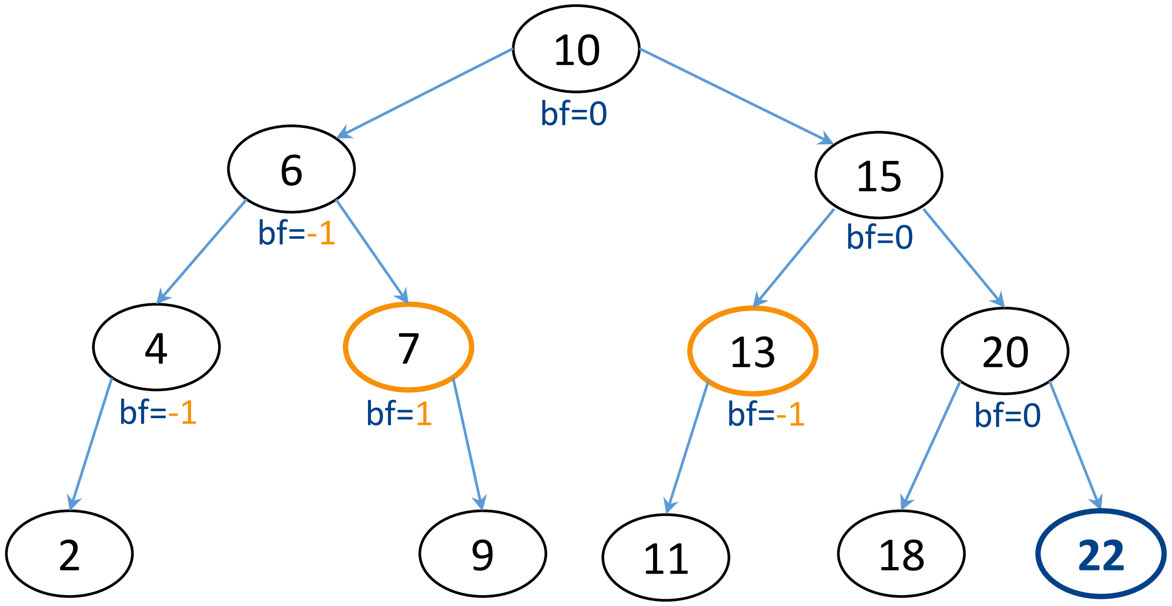
AVL Trees: **Right-right case (left rotation)**

- Let’s take a look at **all** possible cases that could violate the AVL balancing property

- **Case #2**: right-right

  `insert(T, 22)`

  - **Left Rotation**
    - **Rotation root (rr):** node with bf 2 (node 22)
    - **Rotation pivot**: child of **rr** with bf -1 (node 20)
    - **rr** becomes the **left** child of the **pivot**, and **pivot** goes where **rr** was



AVL Trees: **Right-left case (double rotation)**

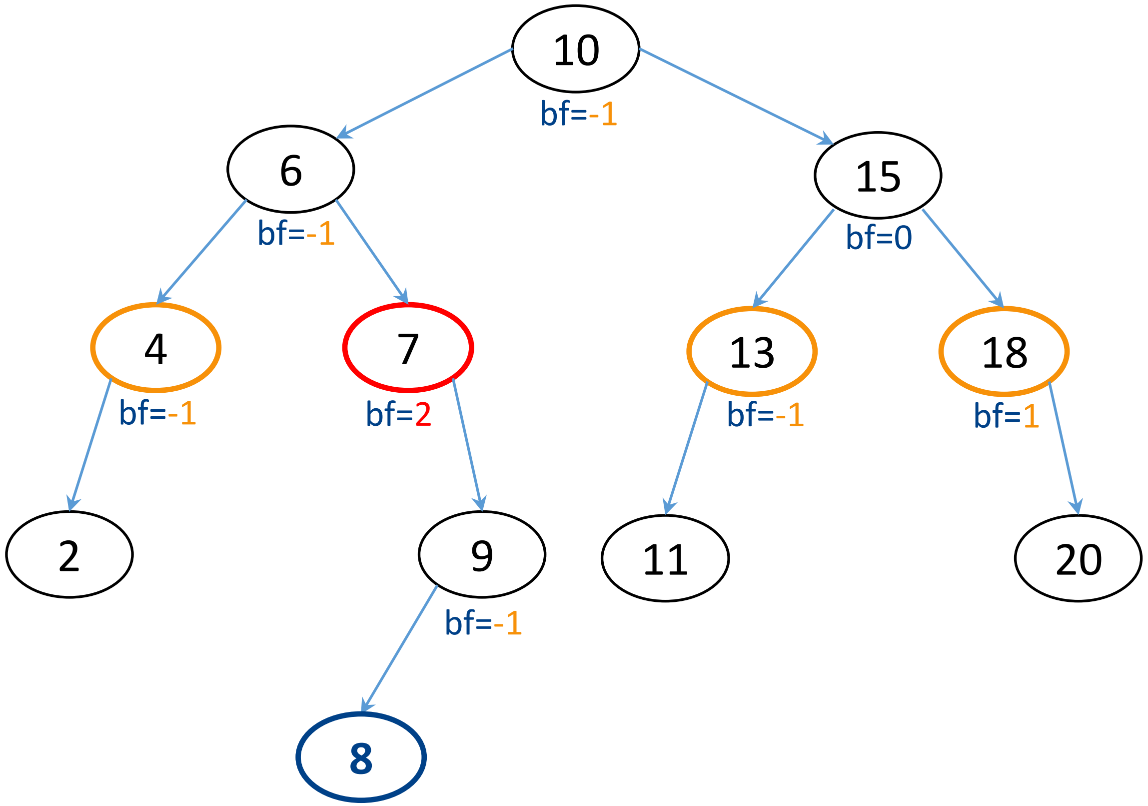
- Let's take a look at **all** possible cases that could violate the AVL balancing property

- **Case #3**: right-left

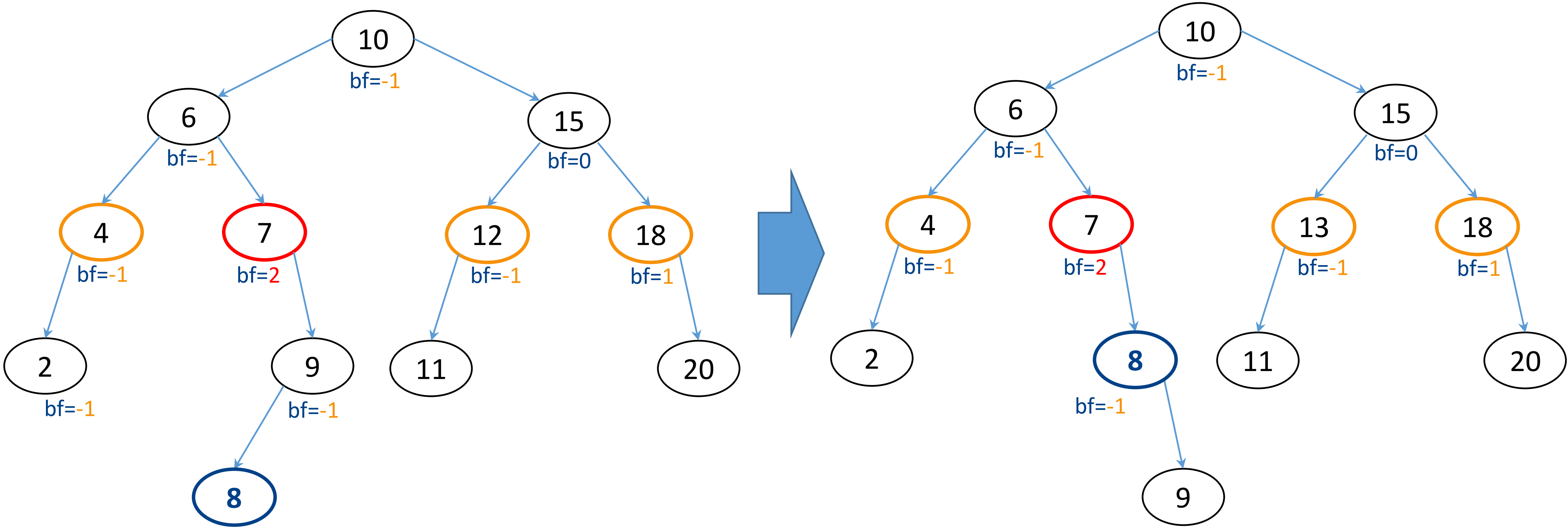
  `insert(T, 8)`

  - Node 7 (grandparent) violates AVL property
  - How to restore it?

  - **Double rotation:** right then left
    - **Rotation root (rr):** node with bf 2 (node 7)
    - **Rotation pivot**: **grandchild** of **rr** (node 8)
    - **Rotation #1: right rotation**
      - Pivot's parent becomes its right child
      - Converts this to the **right-right case**

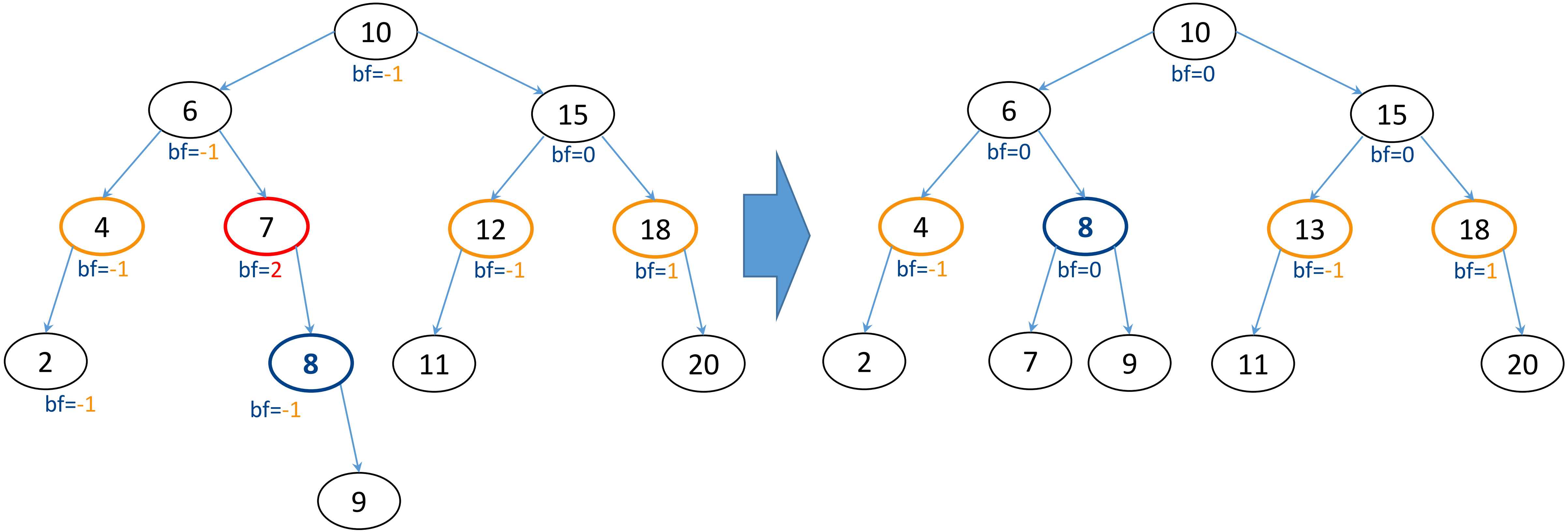


AVL Trees: **Right-left case (double rotation)**



This is now the familiar **right-right case** (**Case 2**)!
Solution: **left rotation** around the pivot

AVL Trees: **Right-left case (double rotation)**



AVL Trees: **Left-right case (double rotation)**

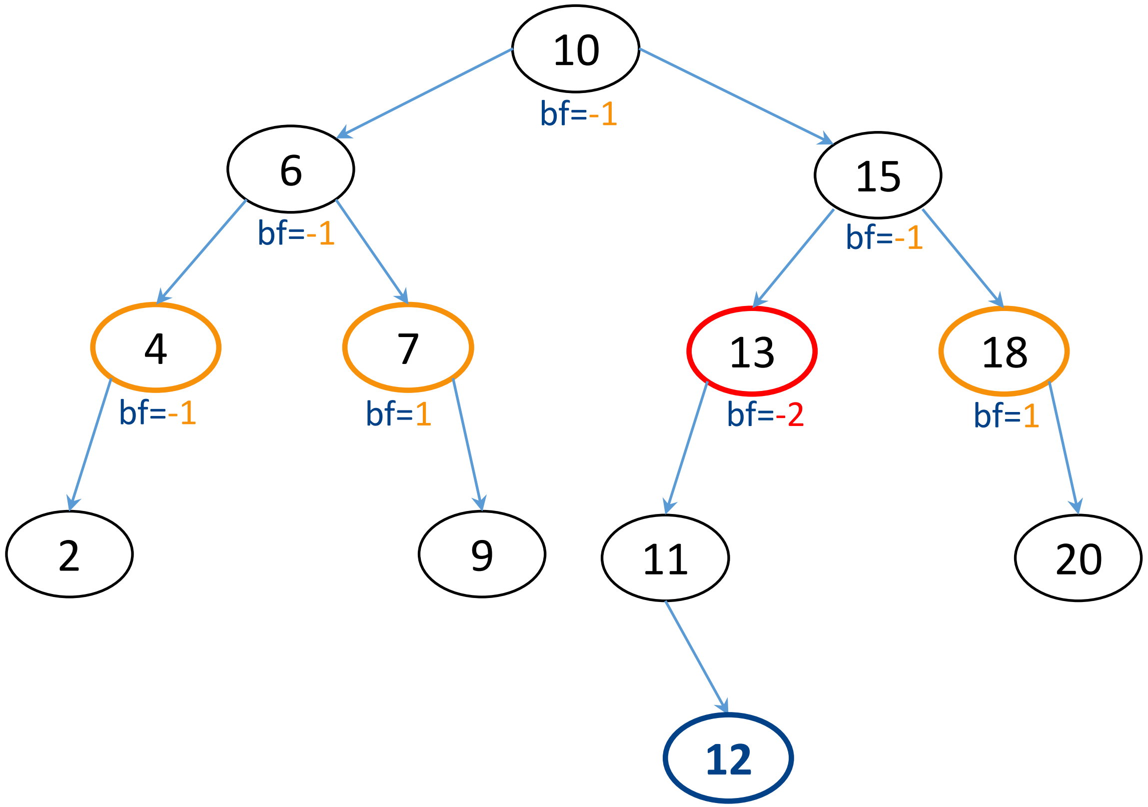
- Let's take a look at **all** possible cases that could violate the AVL balancing property

- **Case #4**: left-right

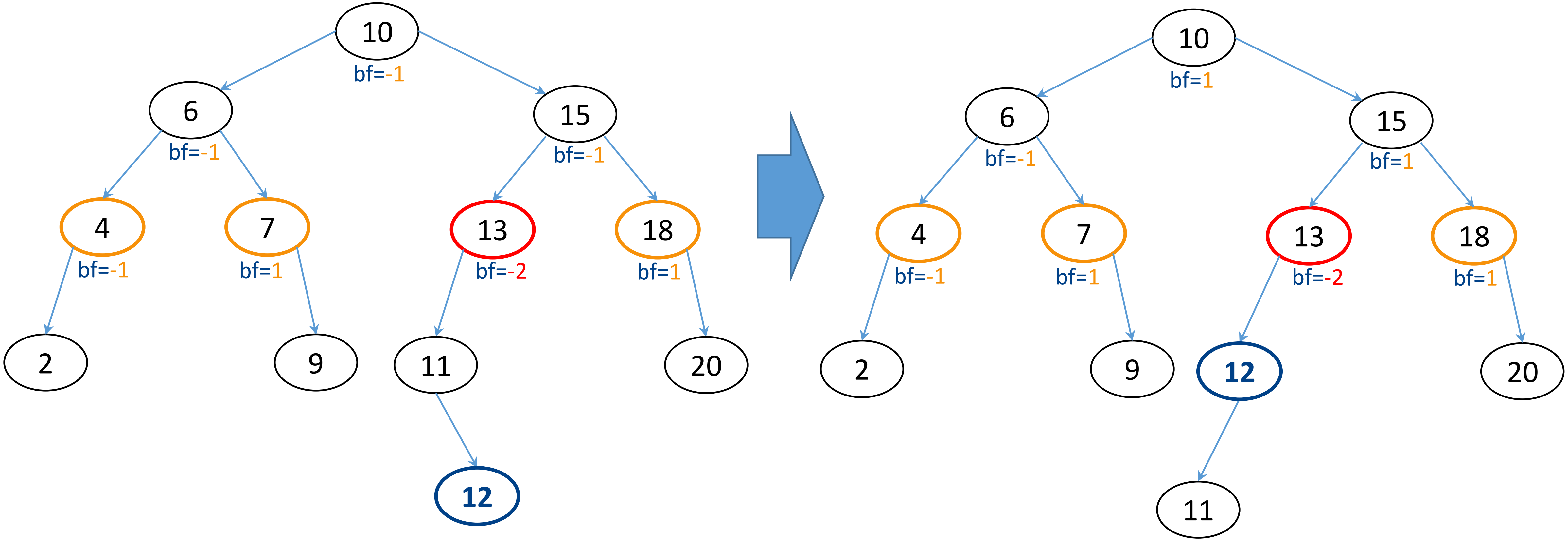
  `insert(T, 12)`

  - Node 13 (grandparent) violates AVL property
  - How to restore it?

  - **Double rotation: left** then **right**
    - **Rotation root (rr):** node with bf 2 (node 13)
    - **Pivot**: **grandchild** of **rr** (node 12)
    - **Rotation #1: left rotation**
      - Pivots parent becomes its **left** child
      - Converts this to the **left-left case**

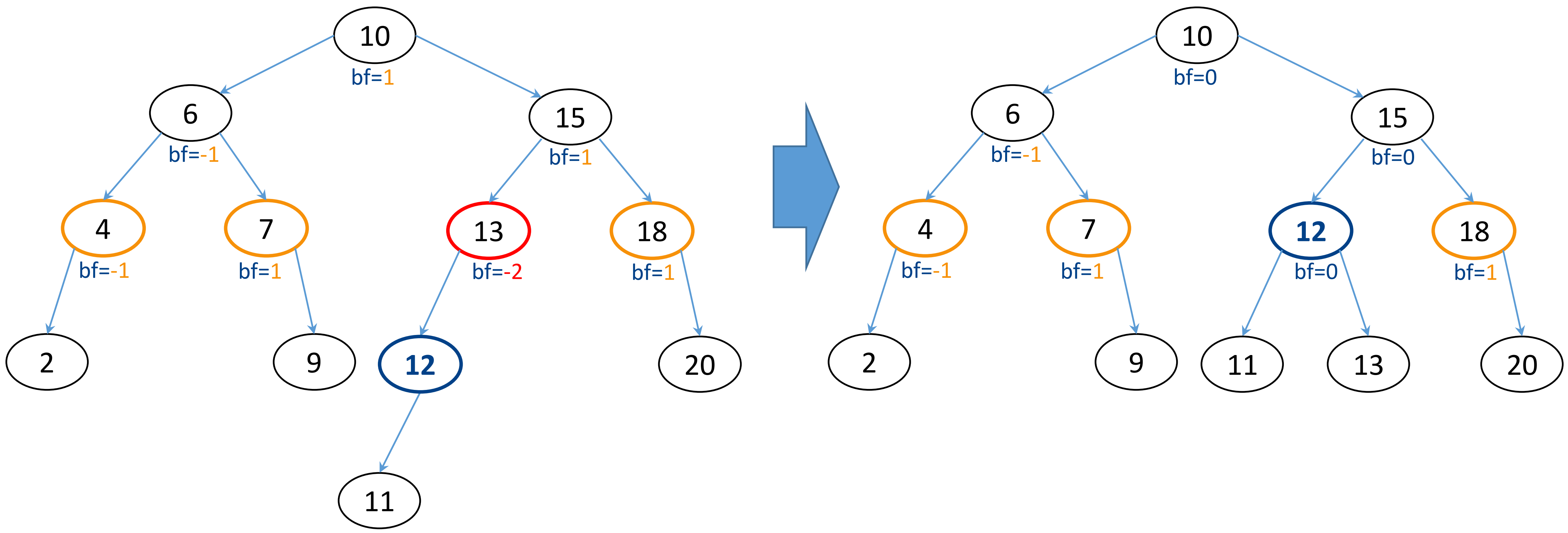


AVL Trees: **Left-right case (double rotation)**



This is now the familiar **left-left case** (**Case 1**)!
Solution: **right rotation** around the pivot

AVL Trees: **Left-right case (double rotation)**



AVL Trees: Overview of All Four Rotations

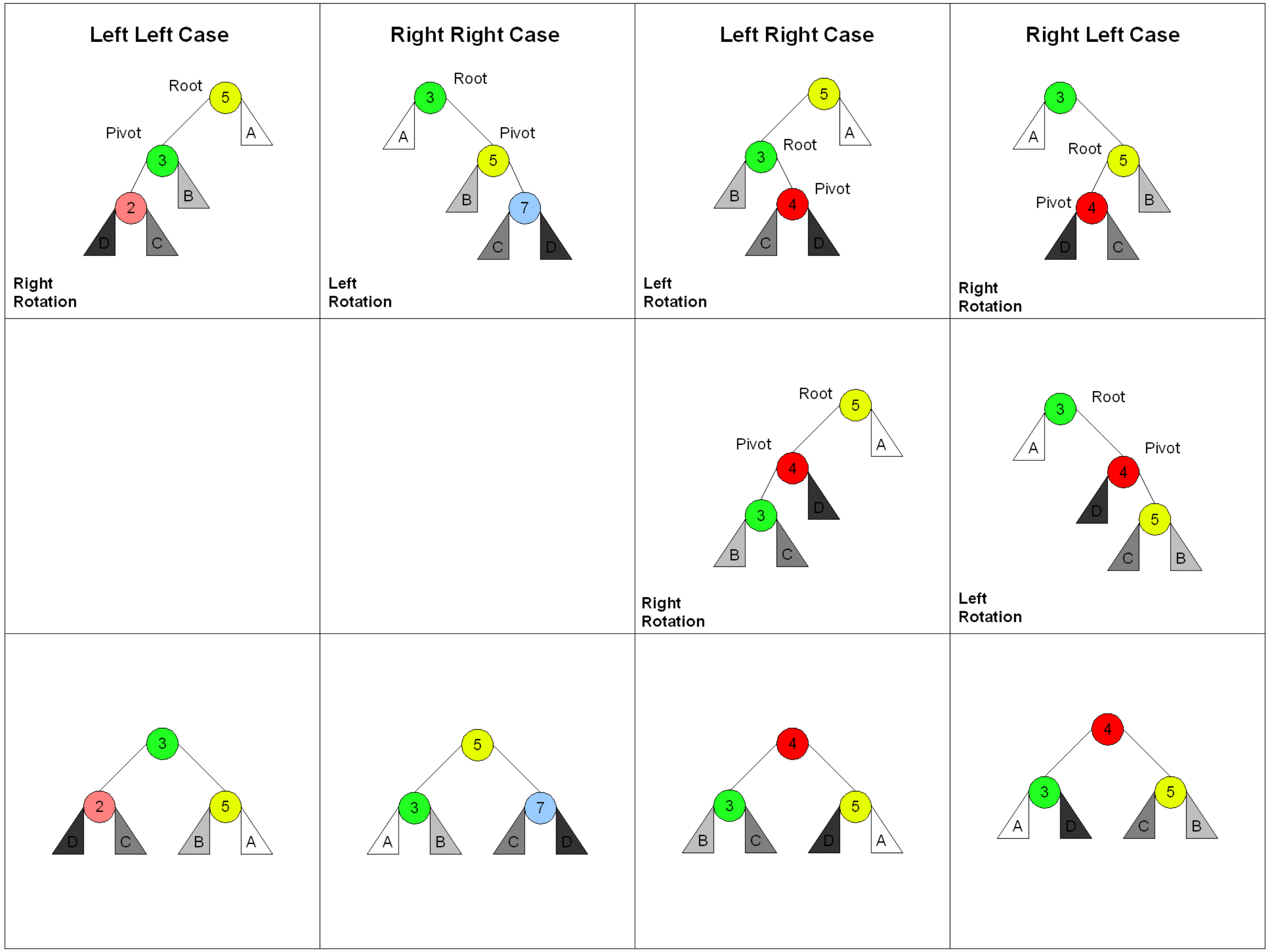


Image from https://upload.wikimedia.org/wikipedia/commons/c/c4/Tree_Rebalancing.gif

Content

- Balanced Binary Search Trees
- AVL Trees
  - Insertion
  - Deletion

AVL Tree: **Deletion**

- We want to delete a node x from a binary search tree T
  - Three cases: two simple, one more complex

1. **Node** without children (i.e., leaf node)
   - Simply set it's corresponding parent's pointer (left or right) to **`null`**

2. **Node with one child** (i.e., only one subtree, left or right)
   - „Bypass" the node x to be deleted – set the corresponding parent's pointer (left or right, depending on which child x is) to point to x's only child

- **Violation** of AVL property ? Only **if**

  (1) x was the left child and its parent y had **bf(y)** = 1 (before x's deletion) or

  (2) x was the right child and its parent y had **bf(y)** = -1 (before x's deletion)

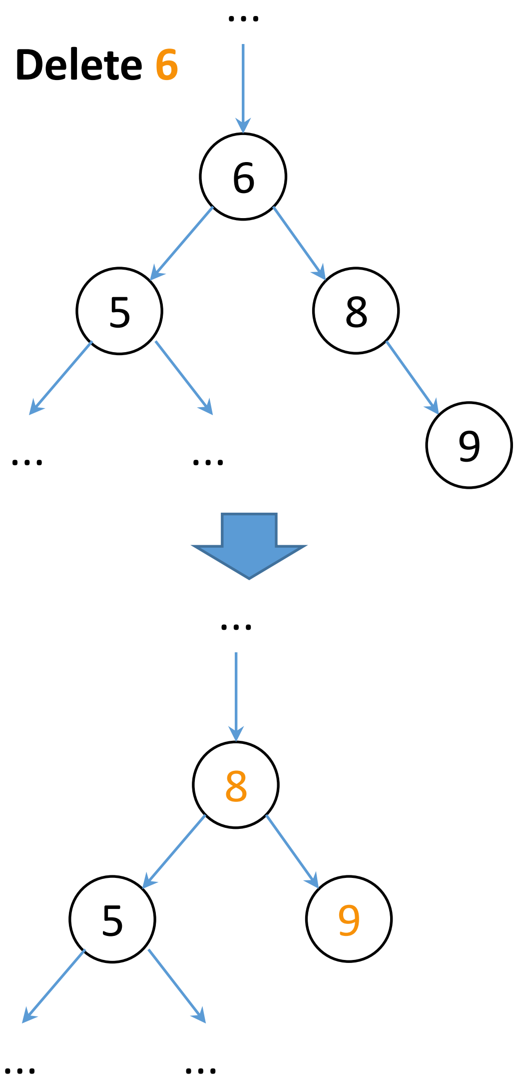
- **Solution**:
  - After deletion of x, its former parent y will have **bf** either -2 or 2 → y is the **rotation root**
  - Recognize which of 4 cases it is – apply the rotation or double rotation as with insertion

Binary Search Tree: **Deletion**

- We want to delete a node x from a binary search tree T
  - Three cases: two simple, one more complex

3. **Node with both children** (the trickiest case)
    - Find x's successor y (in x's right subtree) and place y in x's place
    - **Two subcases**, depending on whether y was direct right child of x or not

- AVL violation in **deletion case 3a**
    - Problem and solution for restoring AVL property the same as for **deletion cases 1 and 2**

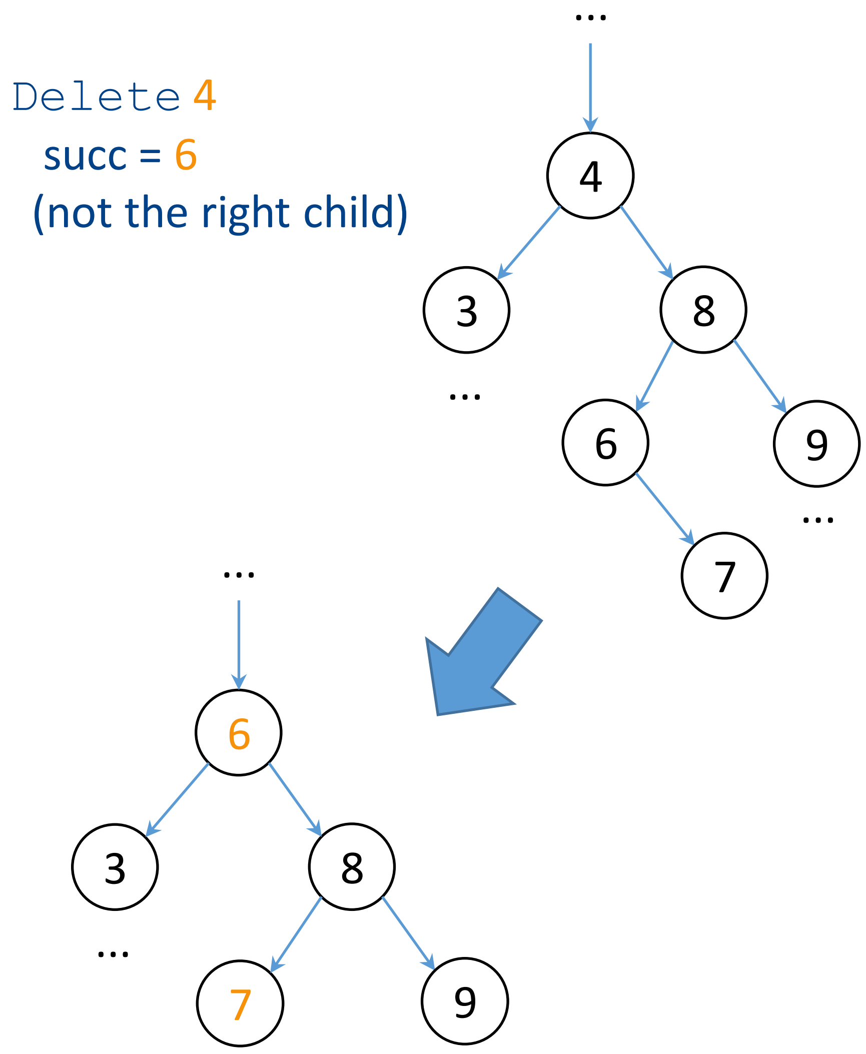


Binary Search Tree: **Deletion**

- **Deletion case #3**: delete node with two children
  - x – being removed, y – the successor

- **Subcase 3b**: successor is not the right child of x, **y ≠ x.right**
  - y has no left child (being a successor of x)
  - y may or may not have the right child
  - **Solution**:
    - We replace y with its own right child
    - Then we replace x with y

- AVL violation and solution?
  - Violation possible for **parent** of **successor** of x (or any of its ancestors)
  - We're effectively deleting the node of the successor(x) and not the node of x
  - If violation → **rotation root** is the **parent** of **successor** of x (before deletion of x)



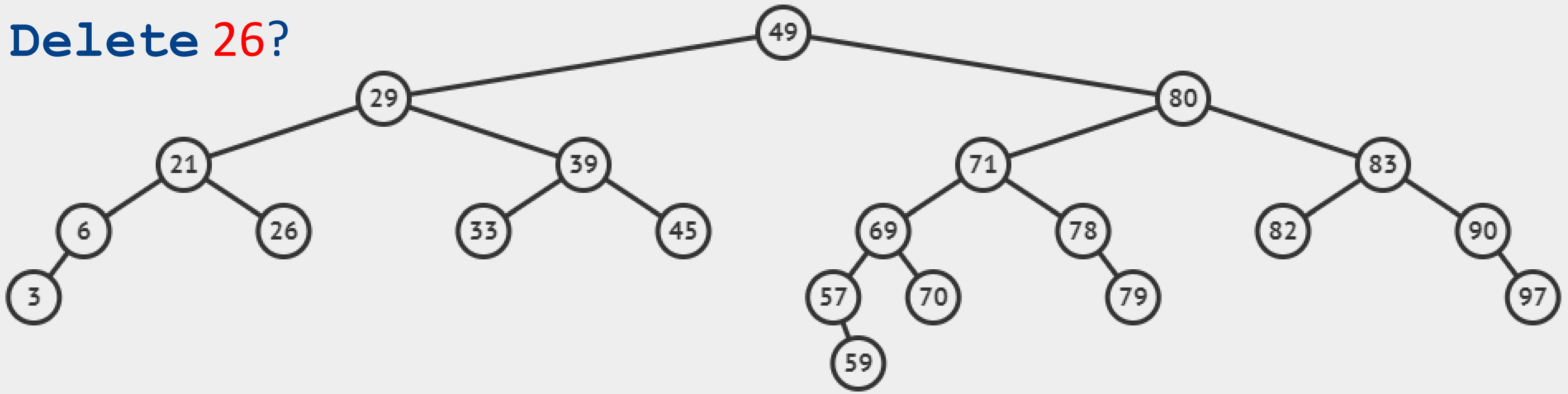
Exercise

- Write the pseudocode for AVL-insert and AVL-delete
- Do it in a **modular fashion**
  - First implement each of the **„rotation cases”**
    - Single rotation: left-left, right-right,
    - Double rotation: right-left, left-right
  - Then think of how to **recognize each case**
    - So that you can „call” the correct (single or double) rotation function
- After adjusting the closest node with **bf** 2 or -2
  - Do you need to adjust any other nodes?

Exercise

- After adjusting the closest node with **bf** 2 or -2
  - Do you need to fix the **bf** of **any other nodes** (i.e., more than one)?

- **Delete** 26?



Questions?


ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Awọn ibeere?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Frågor?
Questions?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
أسئلة؟