Elementary maths for GMT

Algorithm analysis

Trees

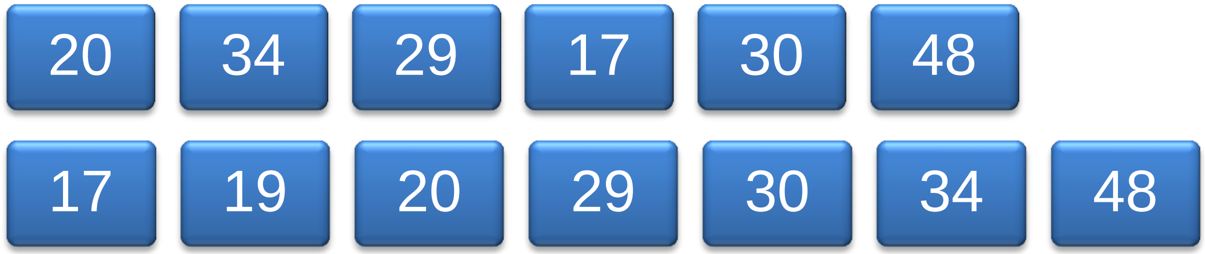
Part I: Binary Search Trees

Goal

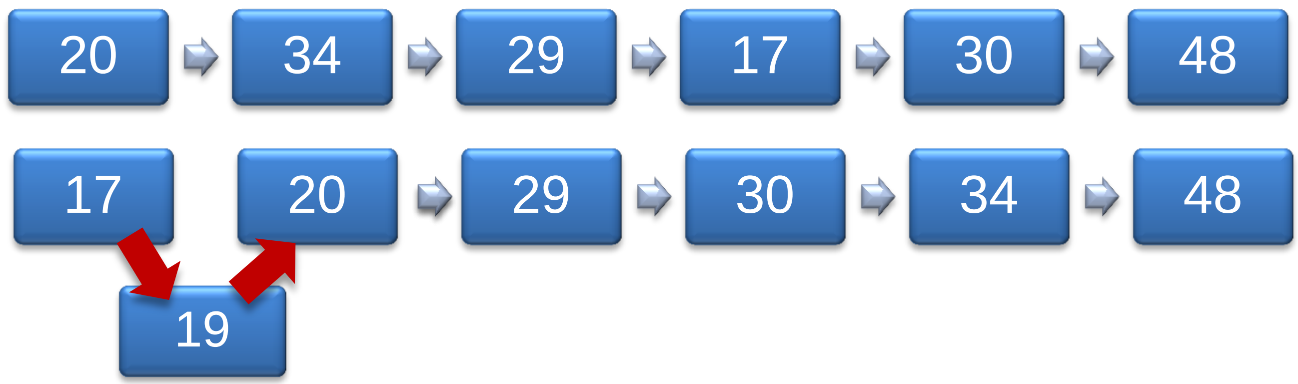
- Analyzing data structures
- Example: binary search trees
- Overview
  - Definition
  - Properties
  - Operations
- Analyzing properties and running times of operations

Storing and modifying data

- Array
  - fast searching, slow insertion



- Linked list
  - slow searching, fast insertion

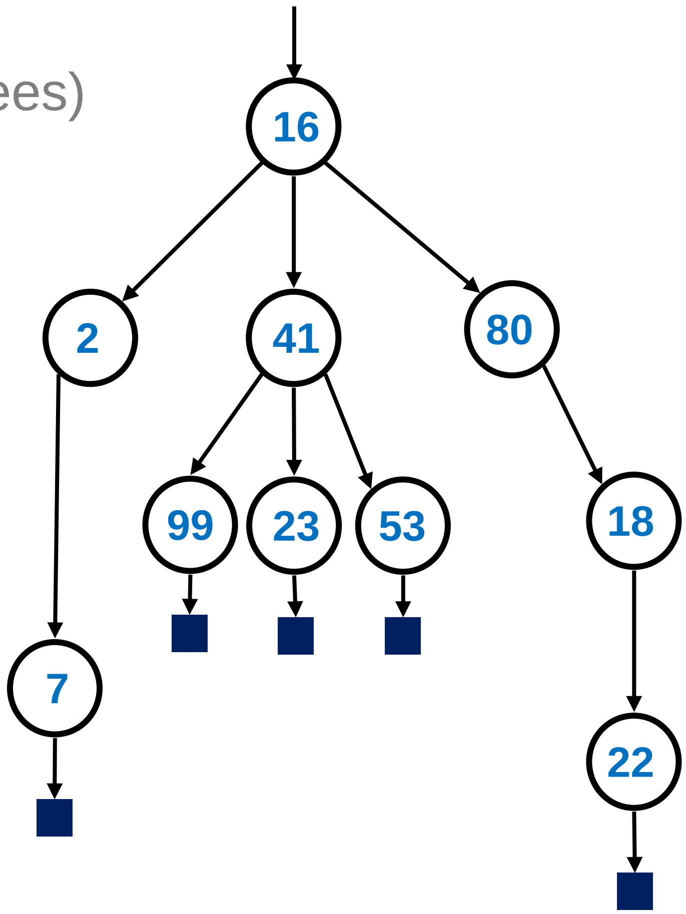


Data structures for maintaining sets

	Search	Insert
Unsorted array	$\Theta(n)$	$\Theta(1)$
Sorted array	$\Theta(\log n)$	$\Theta(n)$
Unsorted list	$\Theta(n)$	$\Theta(1)$
Sorted list	$\Theta(n)$	$\Theta(n)$
Balanced search tree	$\Theta(\log n)$	$\Theta(\log n)$

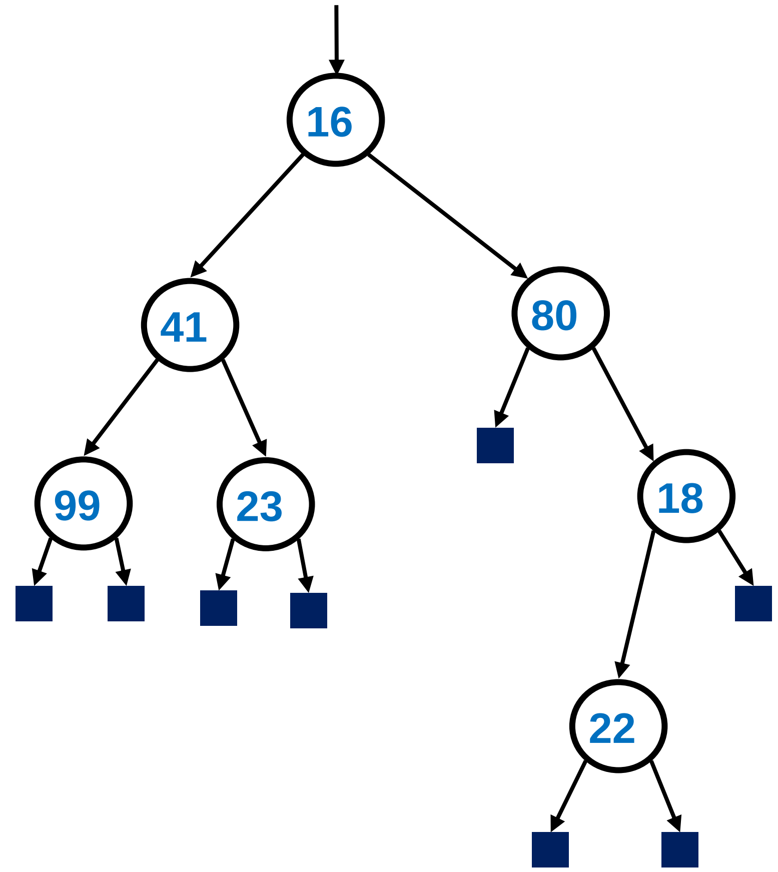
Trees

- Each of the **n nodes** contains
  - data (number, object, etc.)
  - pointers to its children (themselves trees)
- Primitives operations
  - Accessing data: $O(1)$ time
  - Traversing link: $O(1)$ time



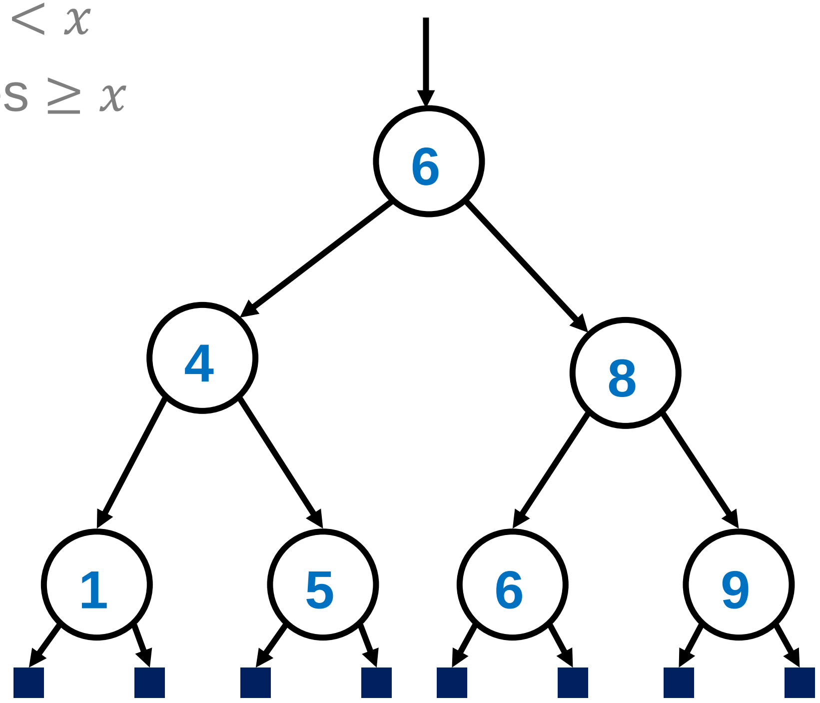
Binary trees

- Every node has only 2 children
  - children can be *dummies*



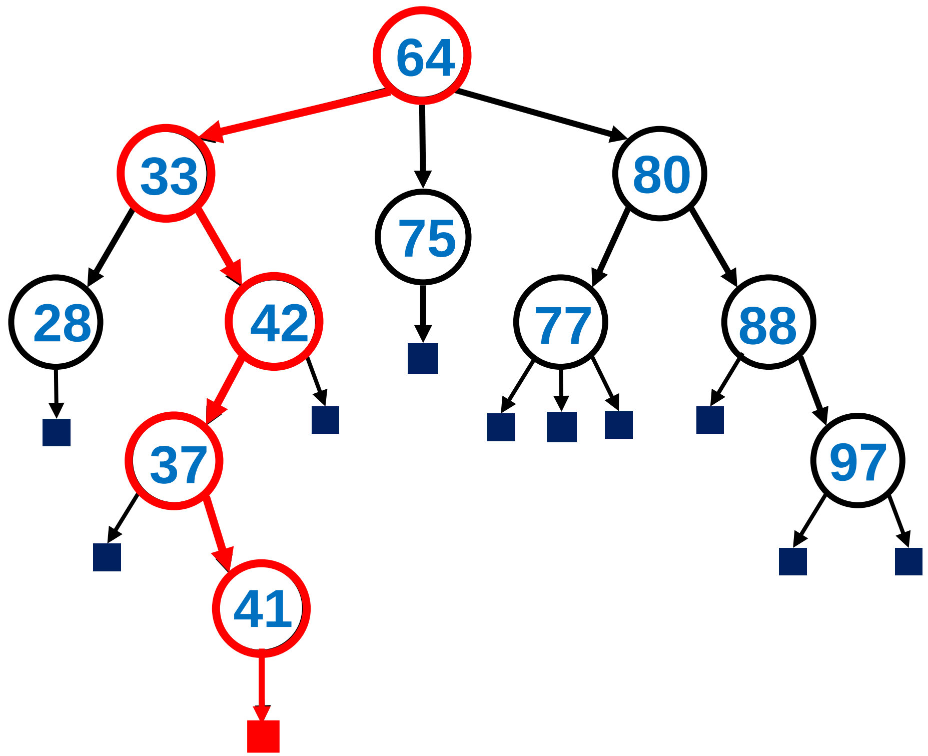
Binary search trees

- Binary trees with "comparable" values
- For a node with value *x*:
  - Left sub-tree contains values $< x$
  - Right sub-tree contains values $\geq x$



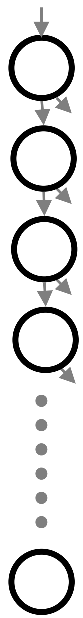
Tree property - Height

- The **height h** of a tree is the length of the longest path
- Property of the height: $0 \leq h \leq n - 1$
- Example
  - height = 4



Binary tree property - Height

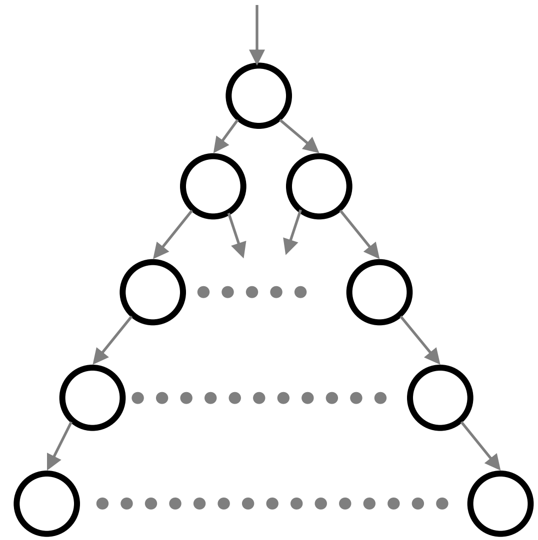
max



0	1
1	1
2	1
3	1
…	…
h	1

$$n - 1$$

min

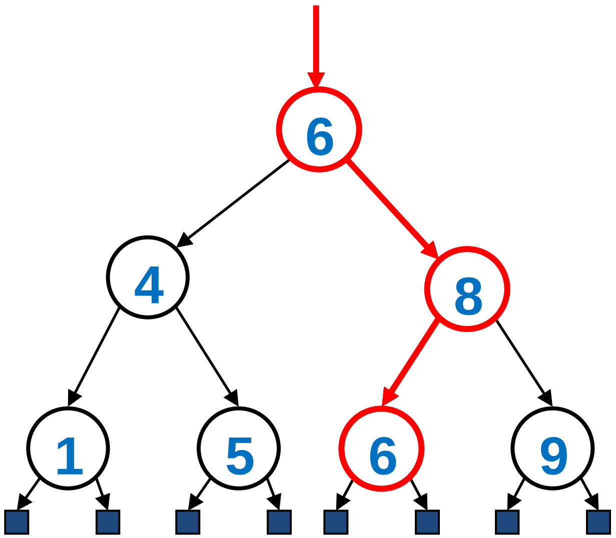


0	1
1	2
2	4
…	…
h	2^h

$$\lfloor {}^2\log n \rfloor$$

Searching for an element

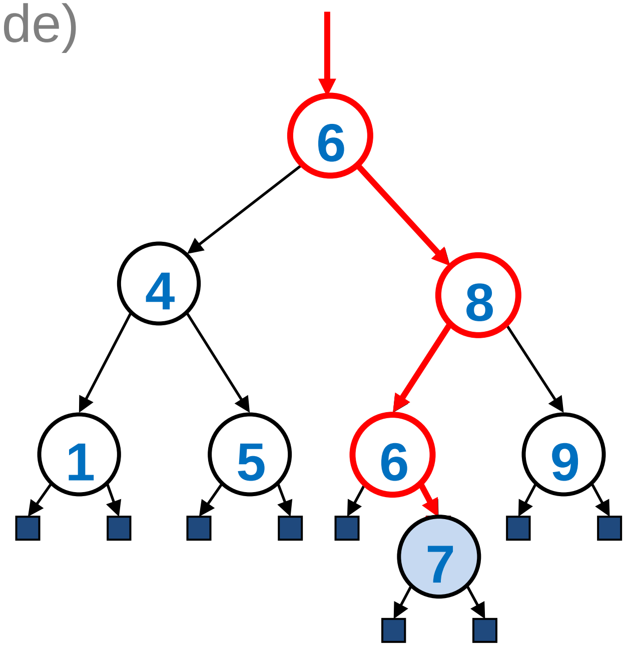
- Example in a binary search tree: searching for 7
  - Start at root
  - At every node:
    - Check if you found it
    - Otherwise choose left or right child according to value in the current node
  - Until you find the value or you are at a leaf node



- Running time is $O(h)$

Inserting an element

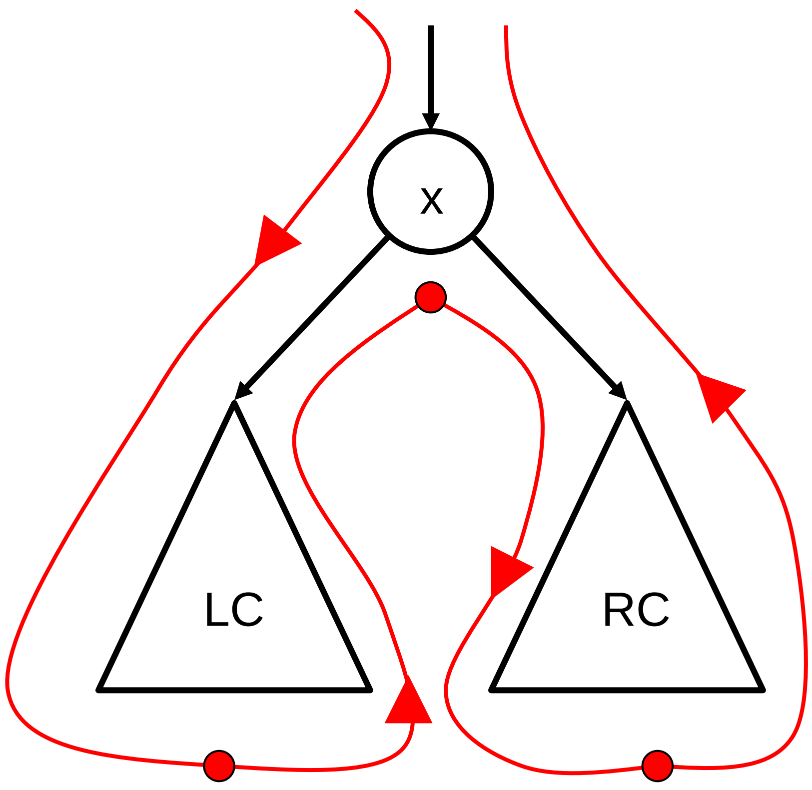
- Example in a binary search tree: inserting 7
  - First search for the value 7 (previous slide)
  - If already present, then nothing to do
  - Else replace the dummy node

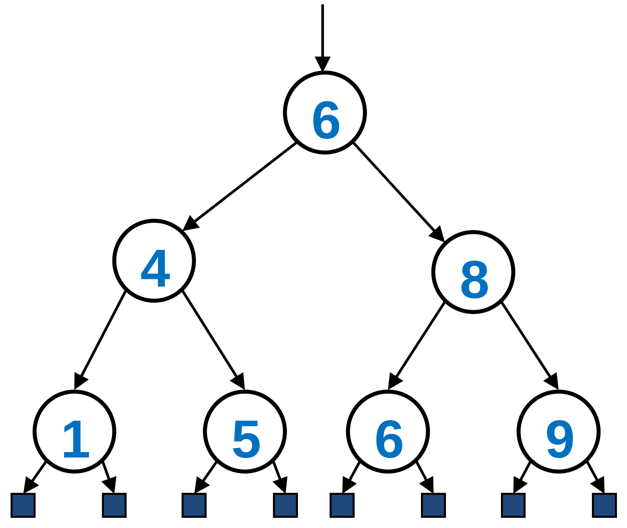


- Running time is $O(h)$

In-order tree traversal

- Visit the nodes sequentially
  - Running time $O(n)$

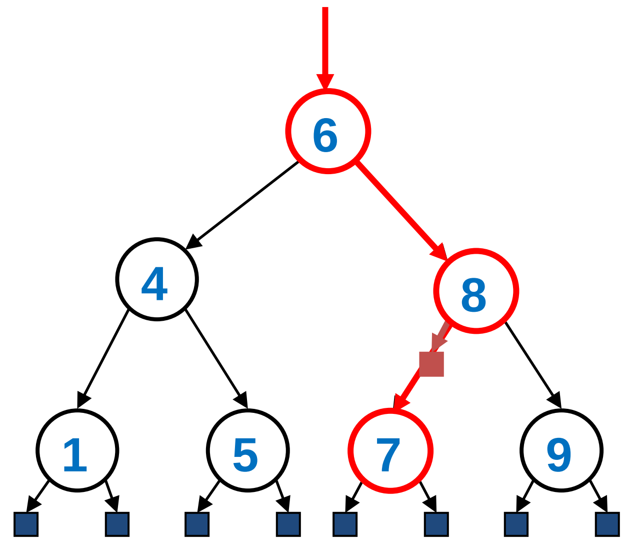




- Example when storing value x in-between visiting the children
  - {1, 4, 5, 6, 6, 8, 9}

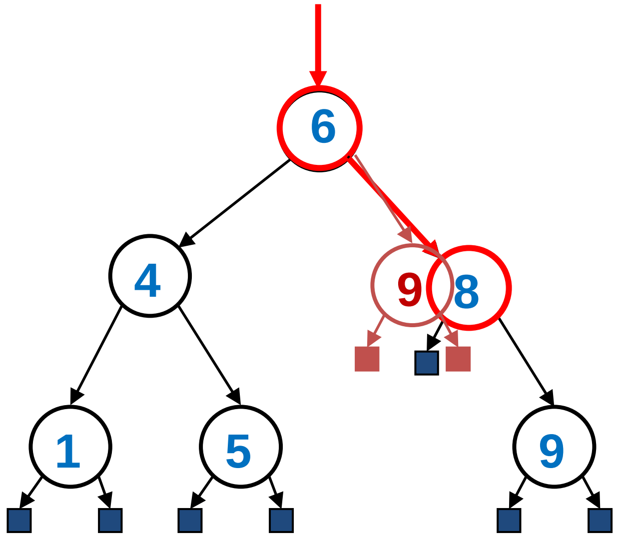
Removing an element (1 / 3)

- Example in a binary search tree: removing 7
  - First search for the value 7
  - If node has at least one dummy node as a child, delete node and attach other child to parent



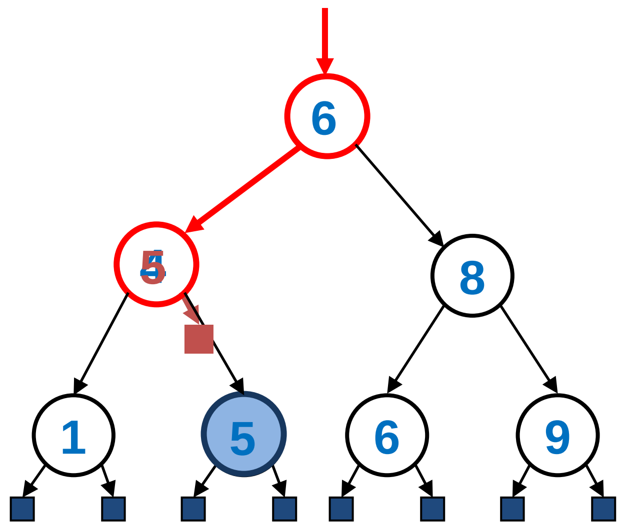
Removing an element (2 / 3)

- Example in a binary search tree: removing 8
  - Search for 8
  - If left (resp. right) child is a dummy node, attach right (resp. left) child to parent



Removing an element (3 / 3)

- Example in a binary search tree: removing 4
  - Search for 4
  - Find in-order successor (here 5)
    - it will always exists and its left child will always be a dummy node
  - Replace the node to remove with the successor node
  - Remove successor in the previously described way



- Running time to find the in-order successor is $O(h)$

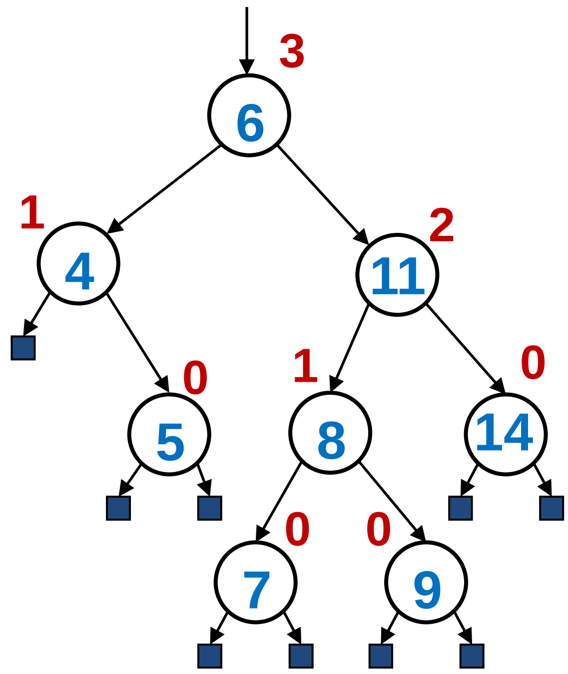
Summary on binary search trees

Parameter / Operation	Property / Time
Height h	$\lfloor {}^2\log n \rfloor \le h \le n - 1$
Accessing data, traversing a link	$O(1)$
In-order traversal	$O(n)$
Search, insertion and removal	$O(h)$

Part II: AVL trees

AVL tree: a balanced binary tree

- An **AVL tree** (Adelson-Velskii Landis) is a binary search tree where for every internal node *v*, the heights of the children of *v* can differ at most by 1
- Example where the heights are shown next to the nodes

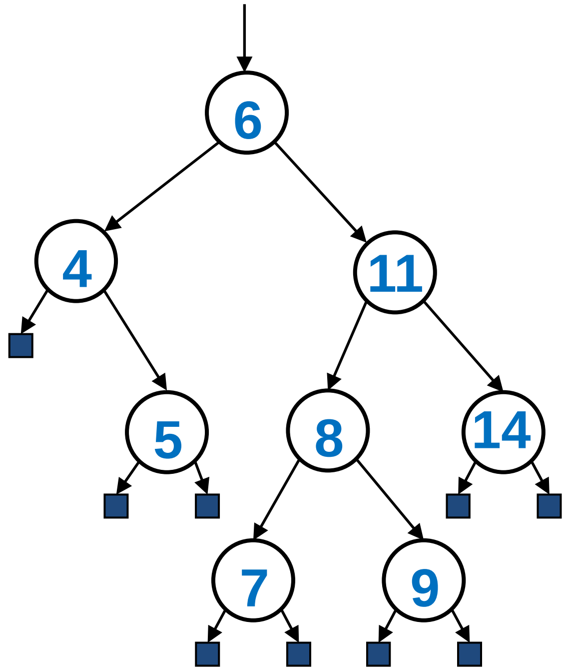


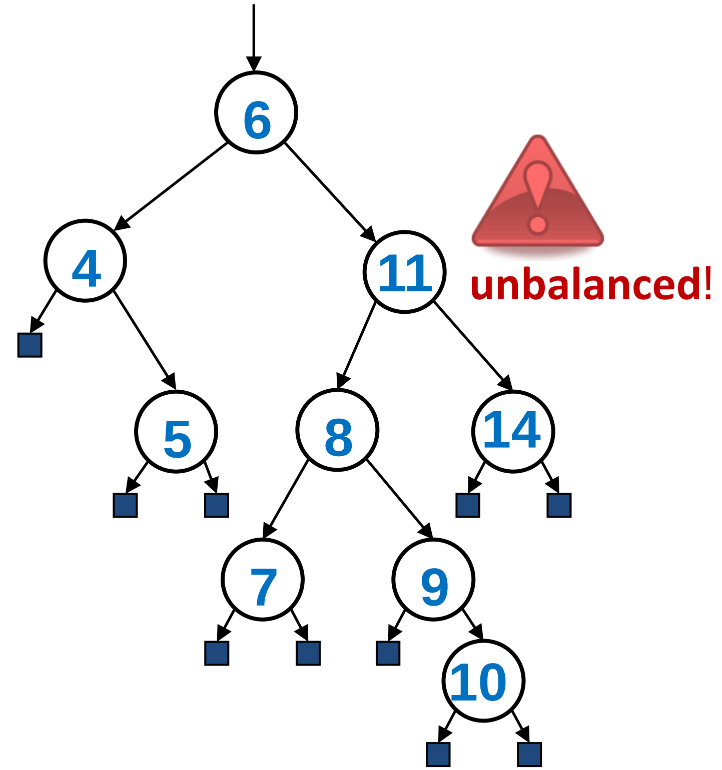
Height of an AVL tree

- **Property**: the height of an AVL tree storing *n* keys is $O(log\ n)$
- **Proof**: let $N(h)$ be the minimum number of internal nodes of an AVL tree of height *h*
  - $N(0) = 1$ and $N(1) = 2$
  - For $h > 1$, an AVL tree of height *h* contains at least a root node, one AVL sub-tree of height *h – 1*, and one AVL sub-tree of height *h – 2*, so $N(h) = 1 + N(h-1) + N(h-2)$
  - Since $N(h-1) > N(h-2)$, we have $N(h) > 2\,N(h-2)$, and so $N(h) > 2\,N(h-2)$, $N(h) > 4\,N(h-4), N(h) > 8\,N(h-6)$, ...
  - So $N(h) > 2^i N(h-2i)$
  - If we choose $i = \frac{h-1}{2}$: $N(h) > 2^{\frac{h-1}{2}} N\left(h - 2(\frac{h-1}{2})\right) = 2^{\frac{h-1}{2}} N(1) = 2^{\frac{h+1}{2}}$, then $h < 2\log(N(h)) - 1$
  - So the height of an AVL tree is $O(\log n)$

Insertion in an AVL tree

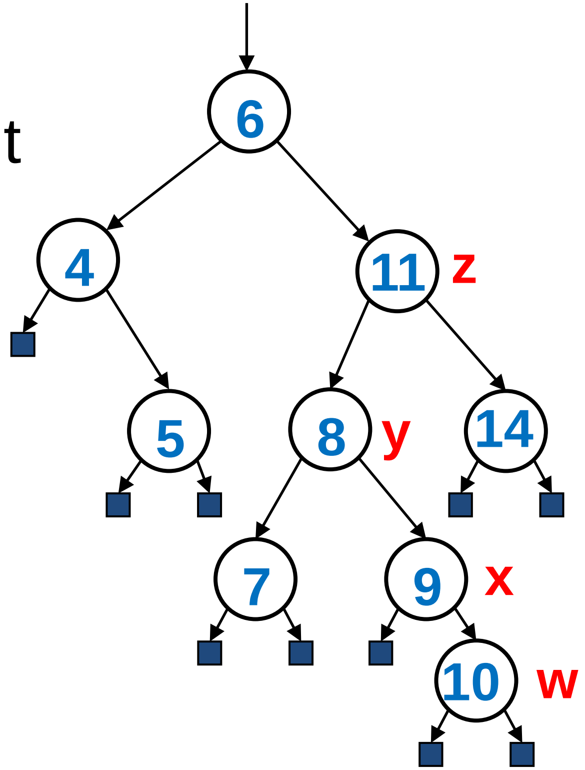
- Insertion is as in a binary search tree: always done by expanding a node
- Example: insert 10 in the following AVL tree





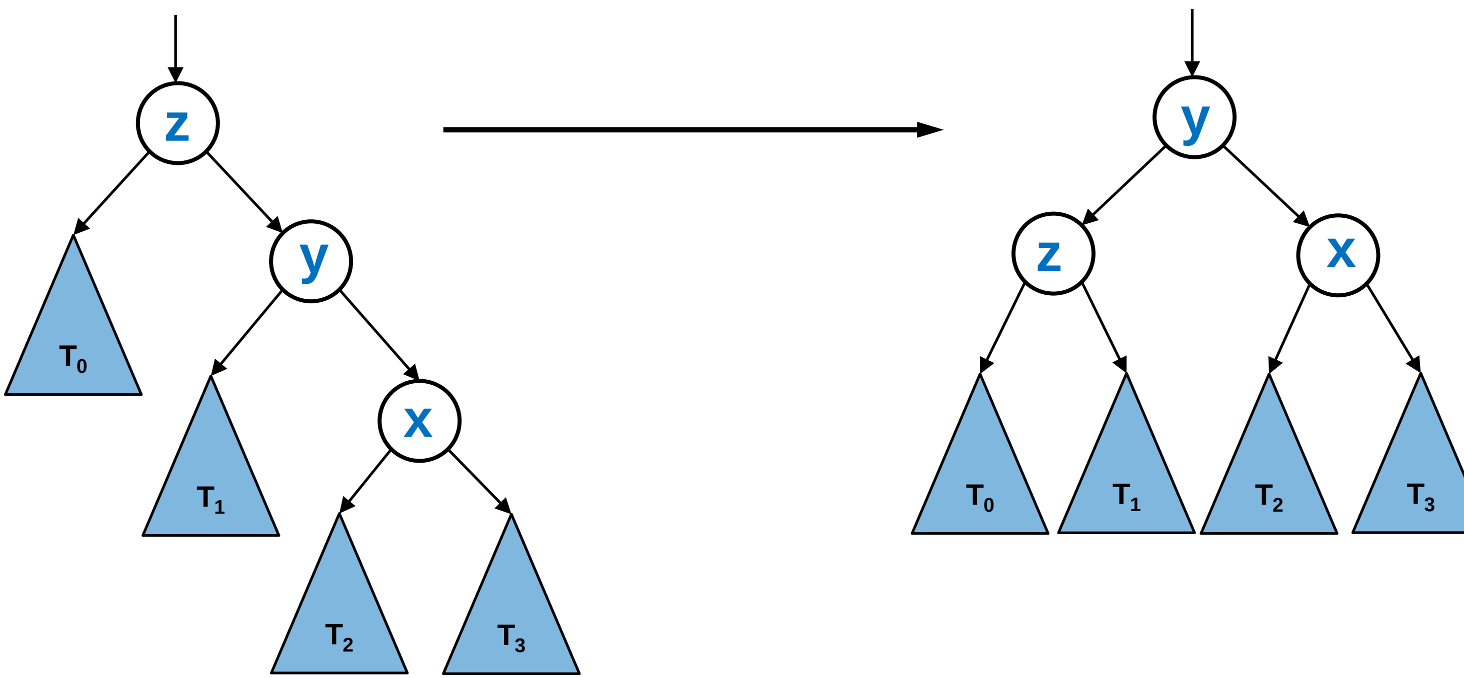
Unbalanced after insertion

- Let **w** be the inserted node (here 10)
- Let **z** be the first unbalanced ancestor of **w** (here 11)
- Let **y** be the child of **z** with higher height (must be an ancestor of **w**) (here 8)
- Let **x** be the child of **y** with higher height (must be an ancestor of **w**, or **w** itself) (here 9)



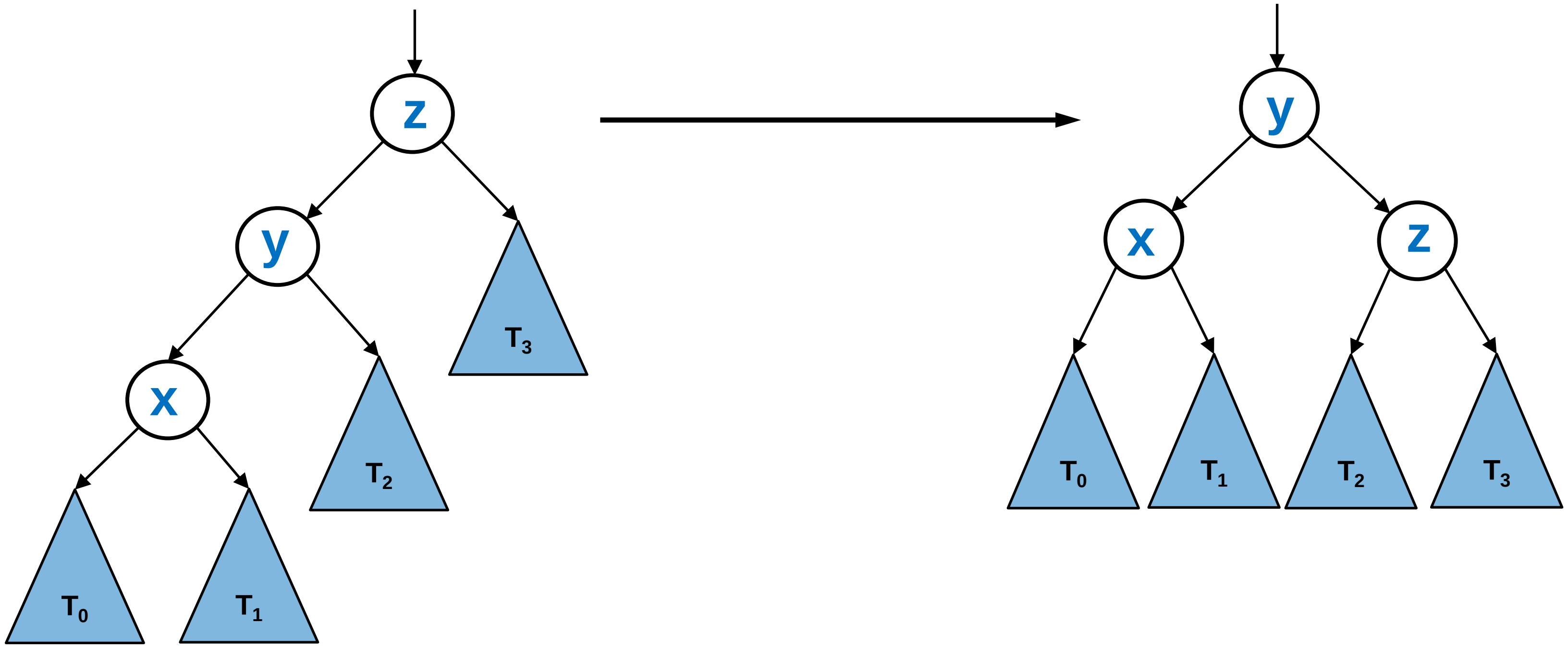
Tri-node restructuring

- Case 1: single rotation
- Perform the rotations needed to make **y** the top most node of the **z**-**y**-**x** sub-tree



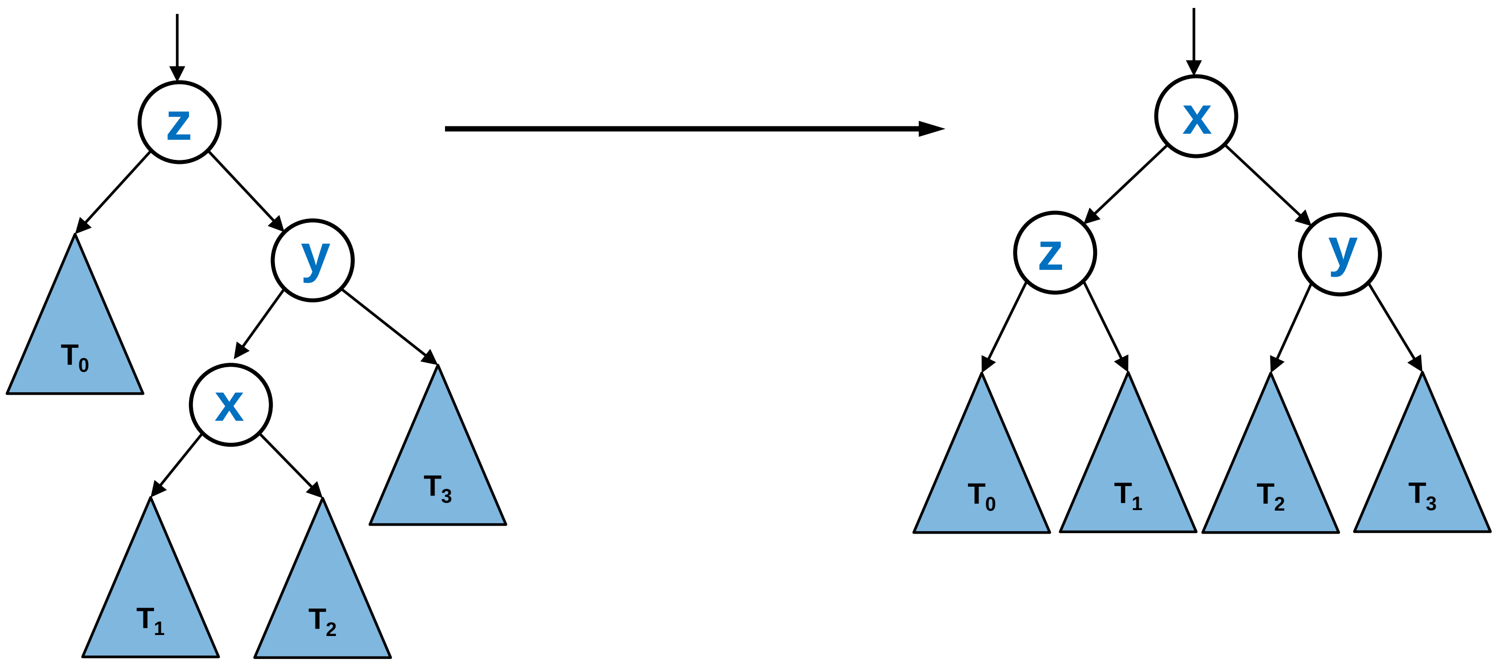
Tri-node restructuring

- Symmetric case



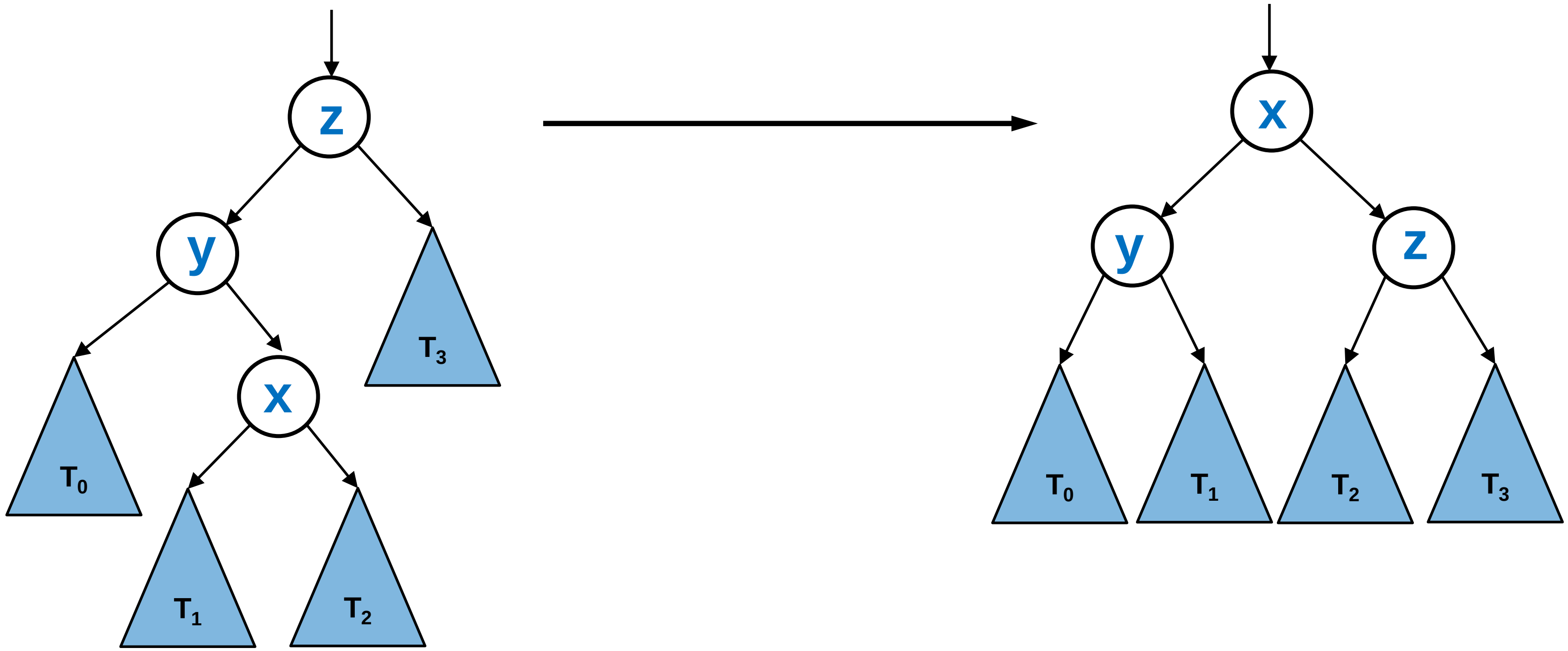
Tri-node restructuring

- Case 2: double rotation

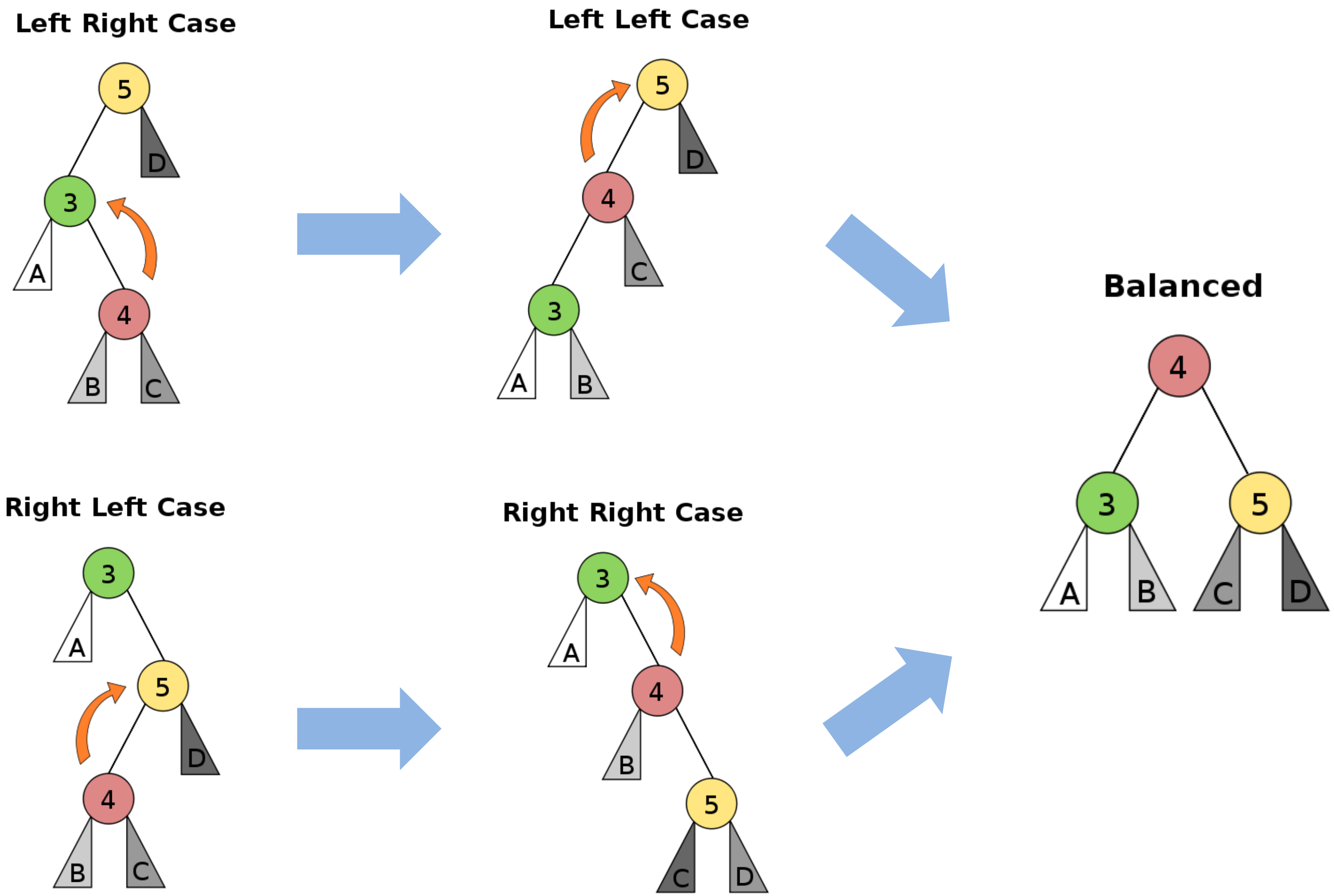


Tri-node restructuring

- Symmetric case

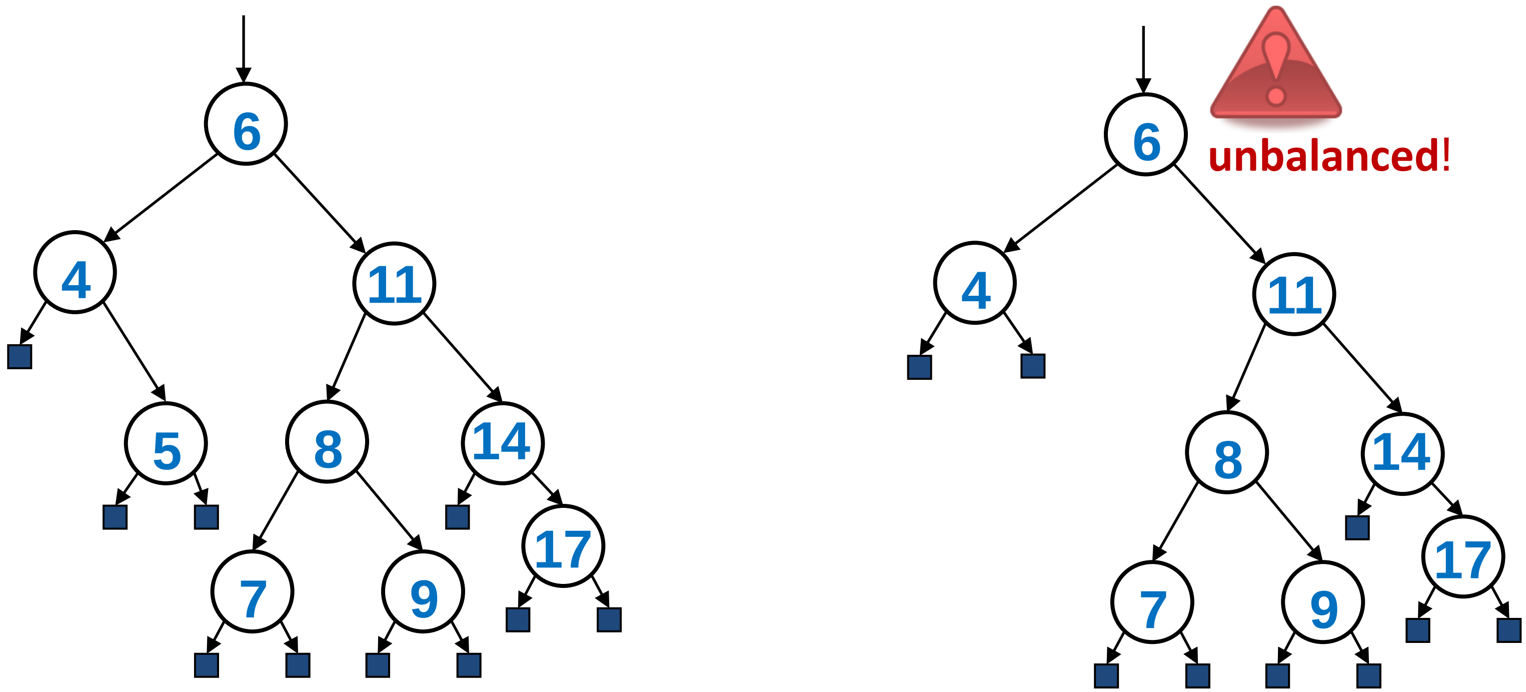


Tri-node restructuring - Summary



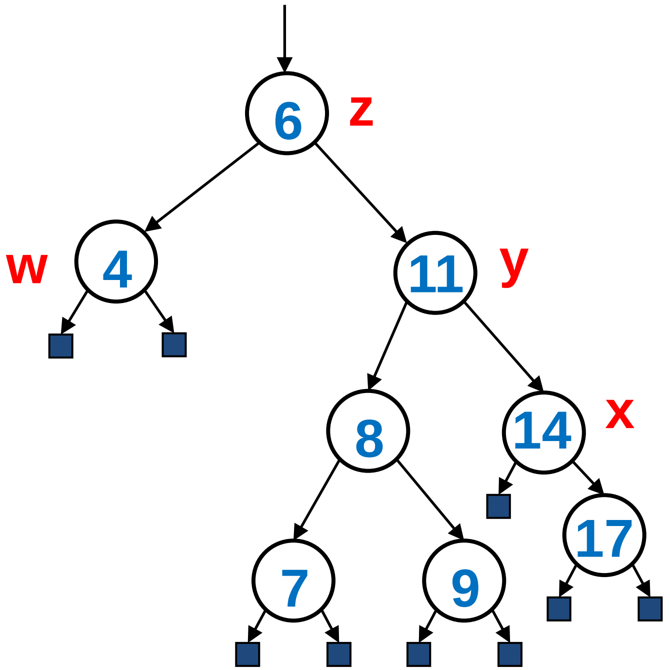
Removal in an AVL tree

- Removal begins as in a binary search tree, which means the node removed will become an *empty* node
- Example: remove 5 in the following AVL tree



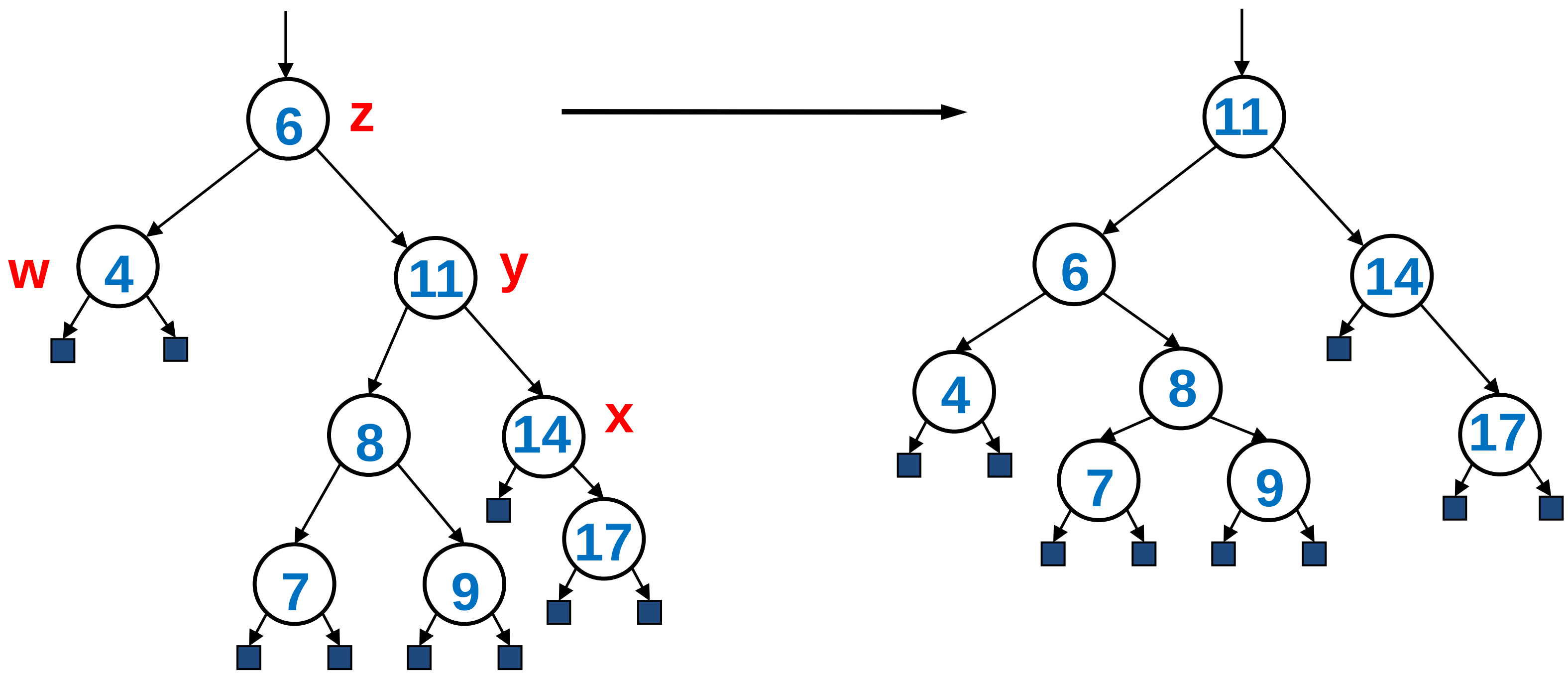
Unbalanced after removal

- Let **w** be the parent of the removed node (here 4)
- Let **z** be the first unbalanced ancestor of **w** (here 6)
- Let **y** be the child of **z** with higher height
  (is now **not** an ancestor of **w**) (here 11)
- Let **x** be
  - the child of **y** with higher height if heights are different, or
  - the child of **y** on the same side as **y** if heights are equal
    (here 14)



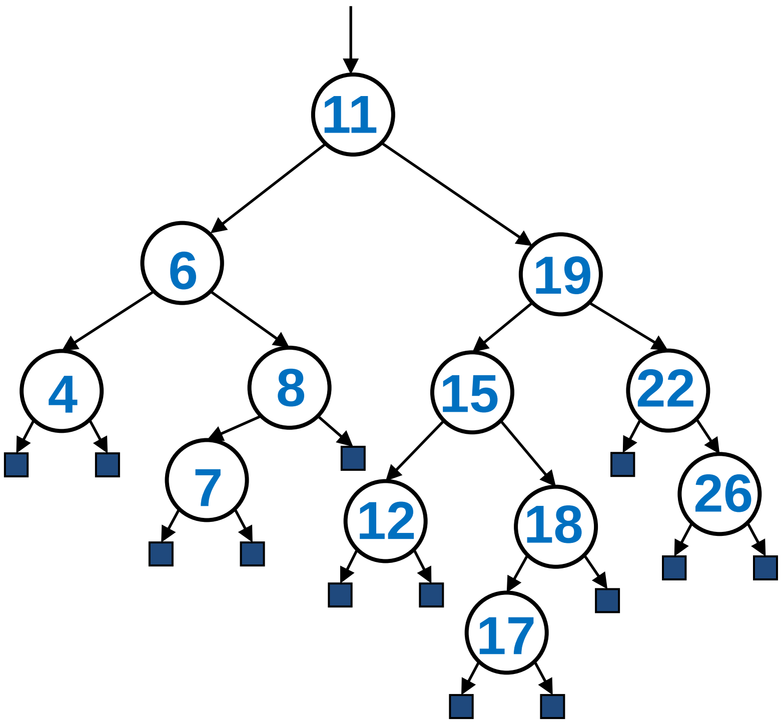
Rebalancing after a removal

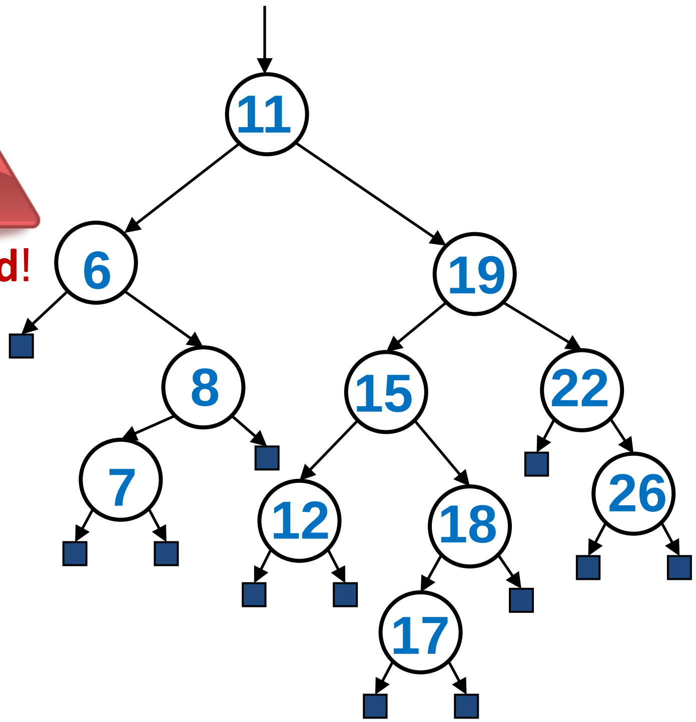
- Performs rotations to make **y** the top most of the **z-y-x** tree
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root is reached



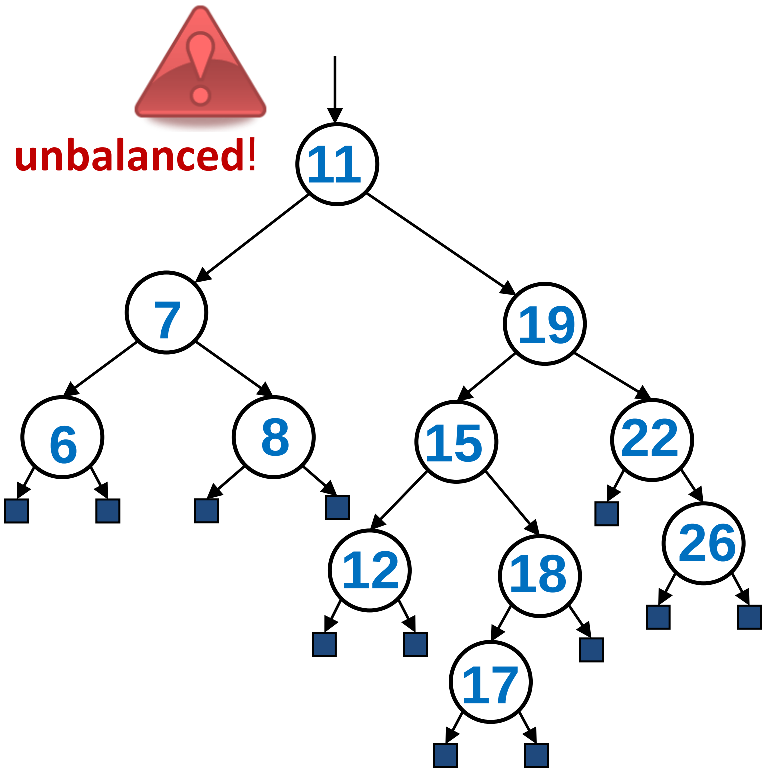
Repeated rebalancing

- Example: remove 4





Repeated balancing



Running times for AVL trees

- Finding a value takes $O(\log n)$ time
  - because height of a tree is always $O(\log n)$
- Traversal of the whole set takes $O(n)$ time
- Insertion takes $O(\log n)$ time
  - Initial find takes $O(\log n)$ time
  - 0 or 1 rebalancing of the tree, maintaining height takes $O(\log n)$ time
- Removal takes $O(\log n)$ time
  - Initial find takes $O(\log n)$ time
  - 0 or more rebalancing of the tree, maintaining height takes $O(\log n)$ time

AVL trees vs. hash tables

- In an AVL tree, insert/delete/search is $O(\log n)$ time, in a hash table they take $O(1)$ time in practice

- In an AVL tree, searching for the smallest value $\geq x$ takes $O(\log n)$ time, in a hash table it takes a linear time

- Enumerating the set in order takes $O(n)$ time in an AVL tree, in a hash table it cannot be done quickly: $O(n \log n)$

- Finding the number of values between given *x* and *y* takes $O(\log n)$ time with a simple variation of an AVL tree, in a hash table it takes linear time

➢ An AVL tree is more versatile than a hash table

Other trees

- **BB[α]-tree** are not height-balanced but weight-balanced. Height is also $O(\log n)$
- **Red-black** trees are balanced with a different scheme and also have height $O(\log n)$
- For background storage, **B-trees** exist and have a degree higher than two (more than 2 children)
- For 2- and higher-dimensional data, various trees exist
  - Kd-trees
  - Quadtrees and octrees
  - BSP-trees
  - Range trees
  - R-trees