Containers:
Set and Dictionary

Jordi Cortadella and Jordi Petit
Department of Computer Science

Sets and Dictionaries

- A set: a collection of items. The typical operations are:
  - Add/remove one element
  - Does it contain an element?
  - Size?, Is it empty?
  - Visit all items

- A dictionary (map): a collection of key-value pairs. The typical operations are:
  - Put a new key-value pair
  - Remove a key-value pair with a specific key
  - Get the value associated to a key
  - Does it contain a key?
  - Visit all key-value pairs

Sets and Dictionaries

- A dictionary can be treated as a set of keys, each key having an associated value.

- We will focus on the implementation of sets.

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

key value

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
python.org	18.21.3.144
sourceforge.net	51.98.23.53

set

dictionary (map)

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Source: Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper

Possible implementations of a set

Unsorted list or vector	
Insertion	$O(n)$, if checking for duplicate keys, $O(1)$ otherwise.
Deletion	$O(n)$ since it has to find the item along the list.
Lookup	$O(n)$ since the list must be scanned.
Good for	Small sets.

Sorted vector	
Insertion	$O(n)$ in the worst case (similar to insertion sort)
Deletion	$O(n)$ since it has to sift the elements after deletion.
Lookup	$O(\log n)$ with binary search.
Good for	Read-only collections (only lookups) or very few updates.

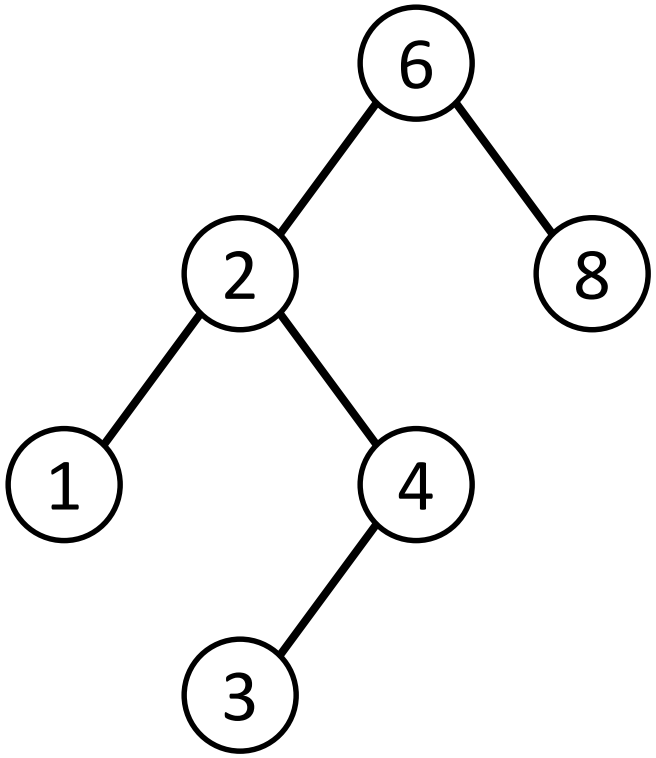
Can we have a data structure with efficient insertion/deletion/lookup operations?

Note: n is the number of items in the set.

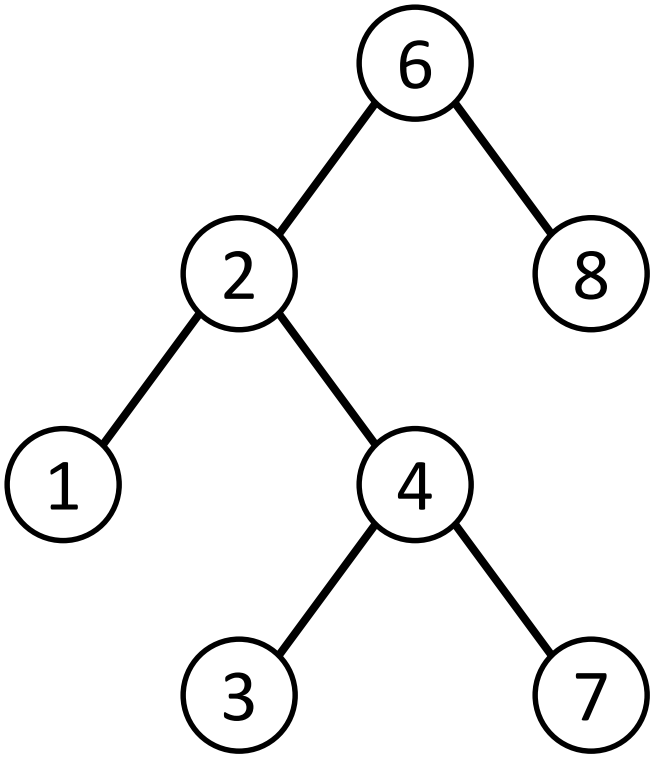
Binary Search Trees

BST property: for every node in the tree with value V:

- All values in the left subtree are smaller than V.
- All values in the right subtree are larger than V.

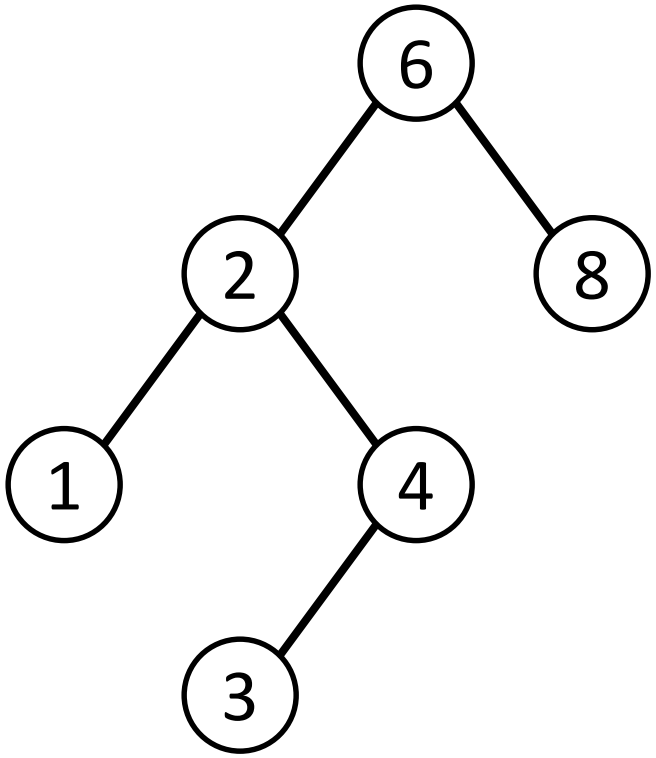


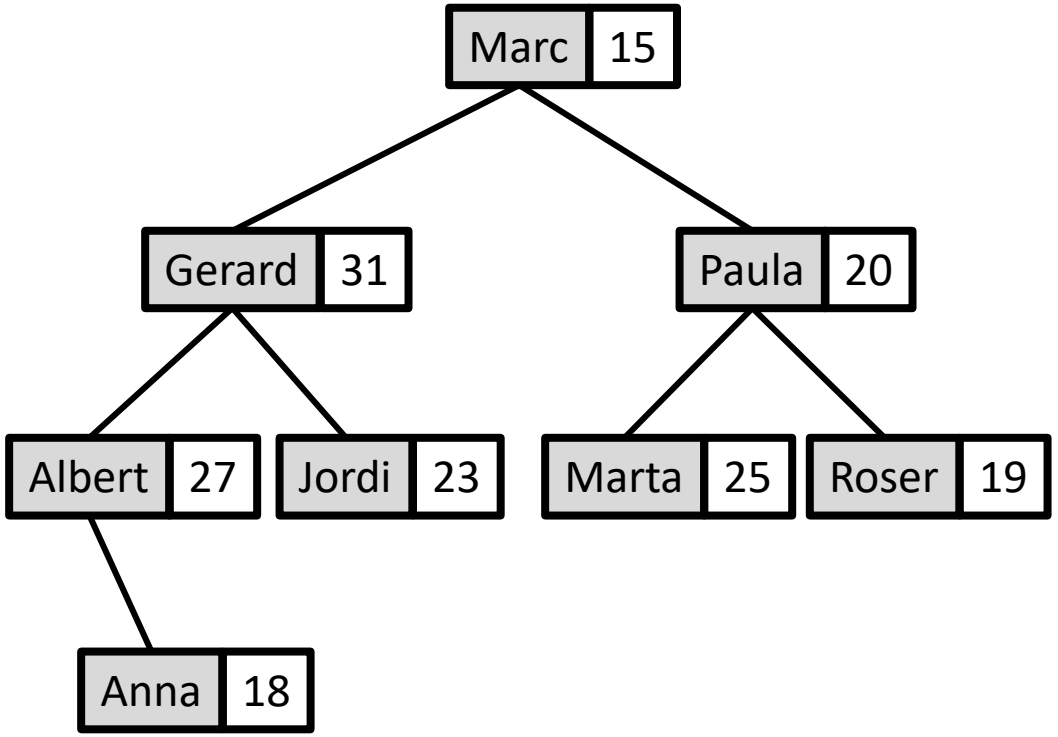
This is a binary search tree



This is ***not*** a binary search tree

Sets and dictionaries





Set

Dictionary (Key=Name, Value=Age)

Requirement: keys must be *comparable*

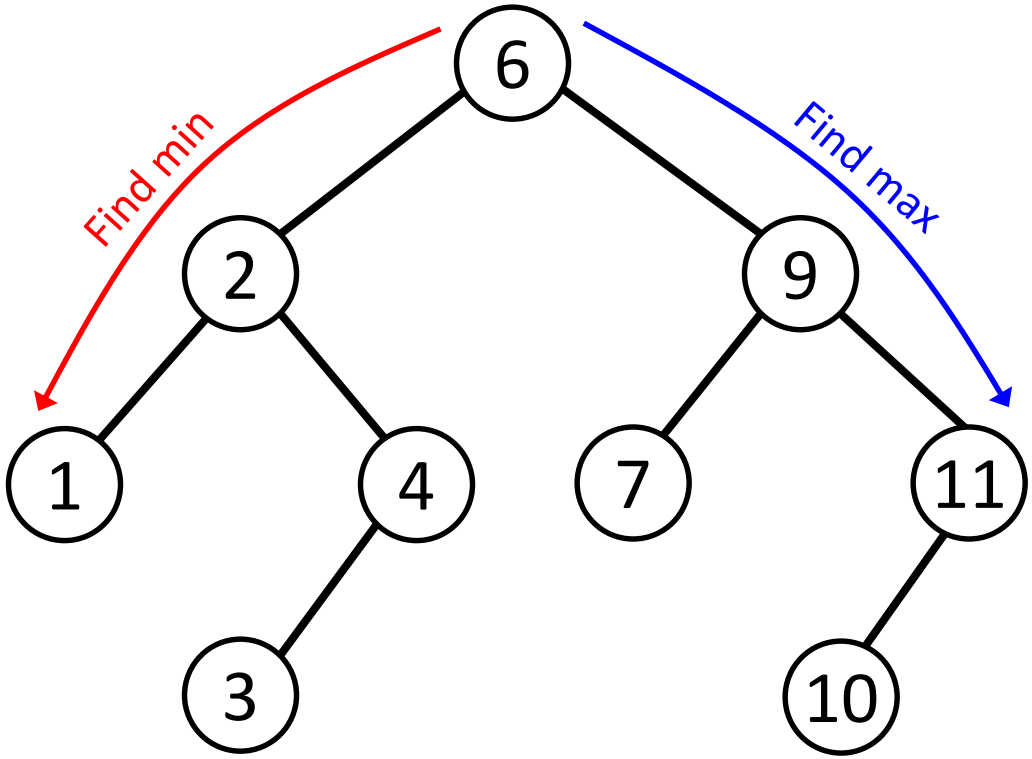
BST: public methods

```
template<typename T>
class Set {
  public:
    // Constructors, assignment and destructor
    Set();
    Set(const Set& S);
    Set& operator=(const Set& S);
    ~Set();

    // Finding elements
    const T& findMin() const;
    const T& findMax() const;
    bool contains(const T& x) const;
    int size() const;
    bool isEmpty() const;

    // Insert/remove methods
    void insert(const T& x);
    void remove(const T& x);
};
```

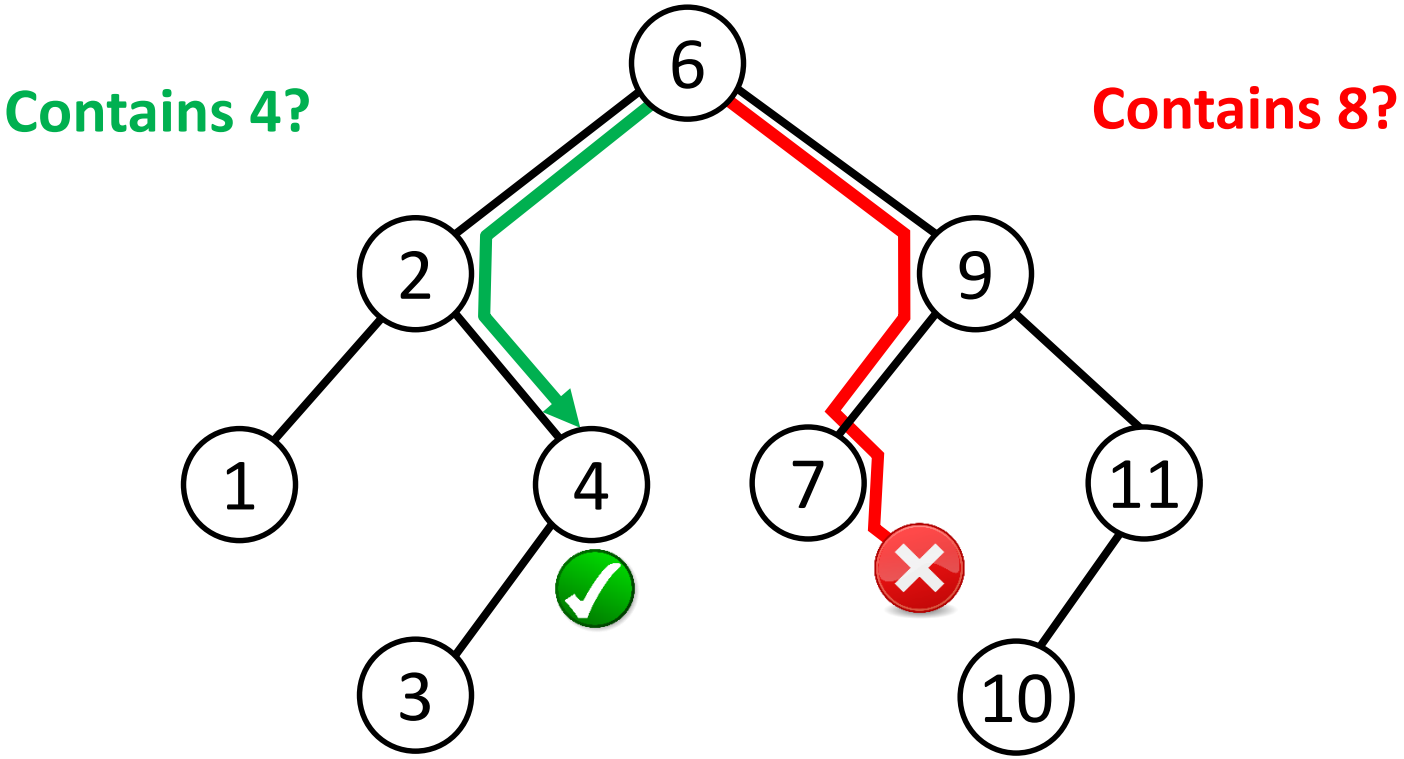
Binary Search Trees: find min/max



Find min: Go to the leftmost element.

Find max: Go to the rightmost element.

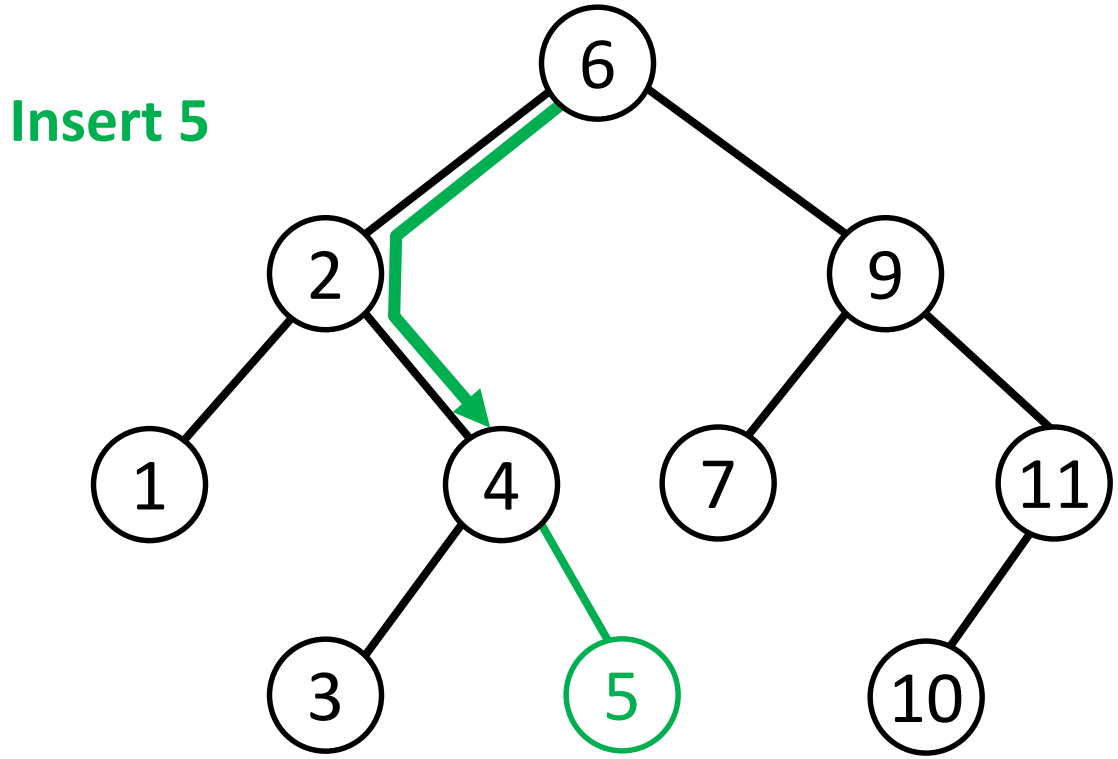
Binary Search Trees: contains



Contains:

- Move to left/right depending on the value.
- Stop when:
  - The value is found (contained)
  - No more elements exist (not contained)

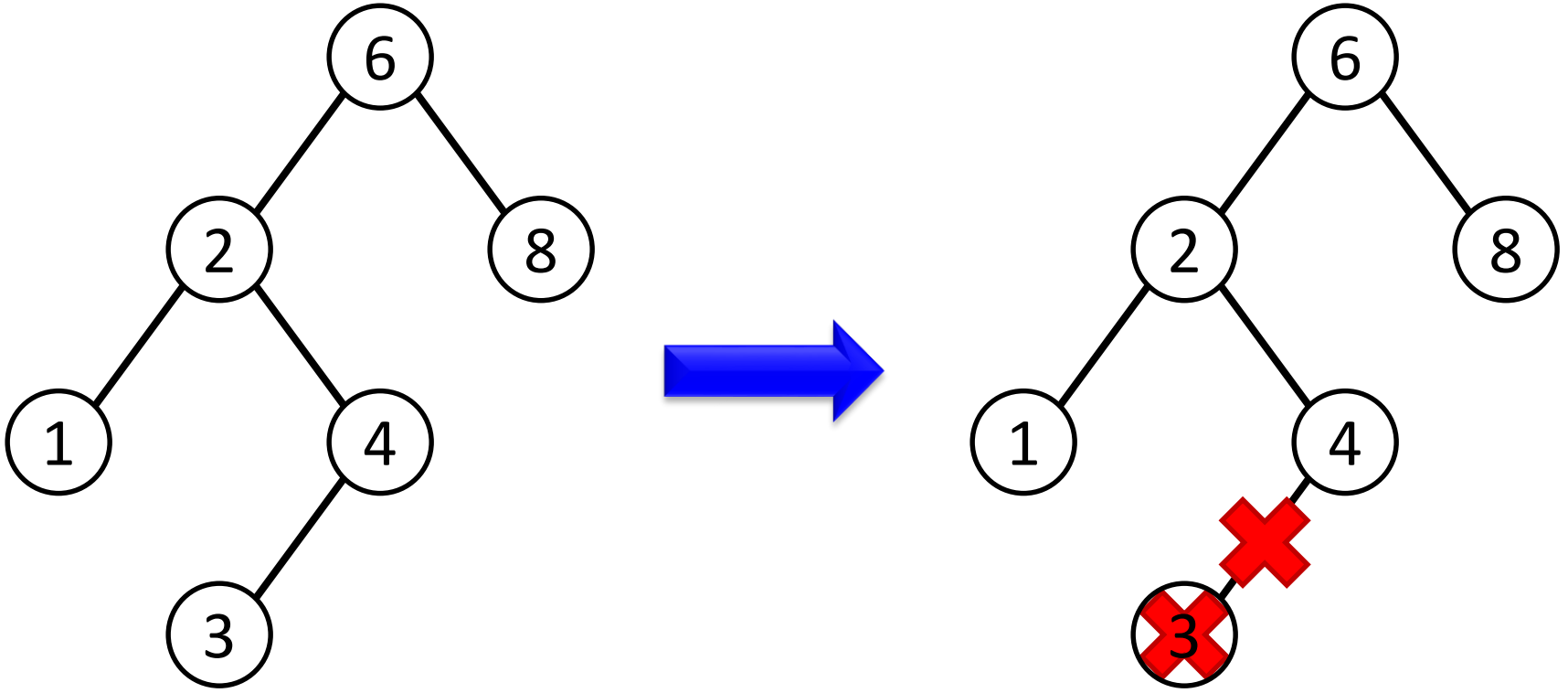
Binary Search Trees: insert



Insert:

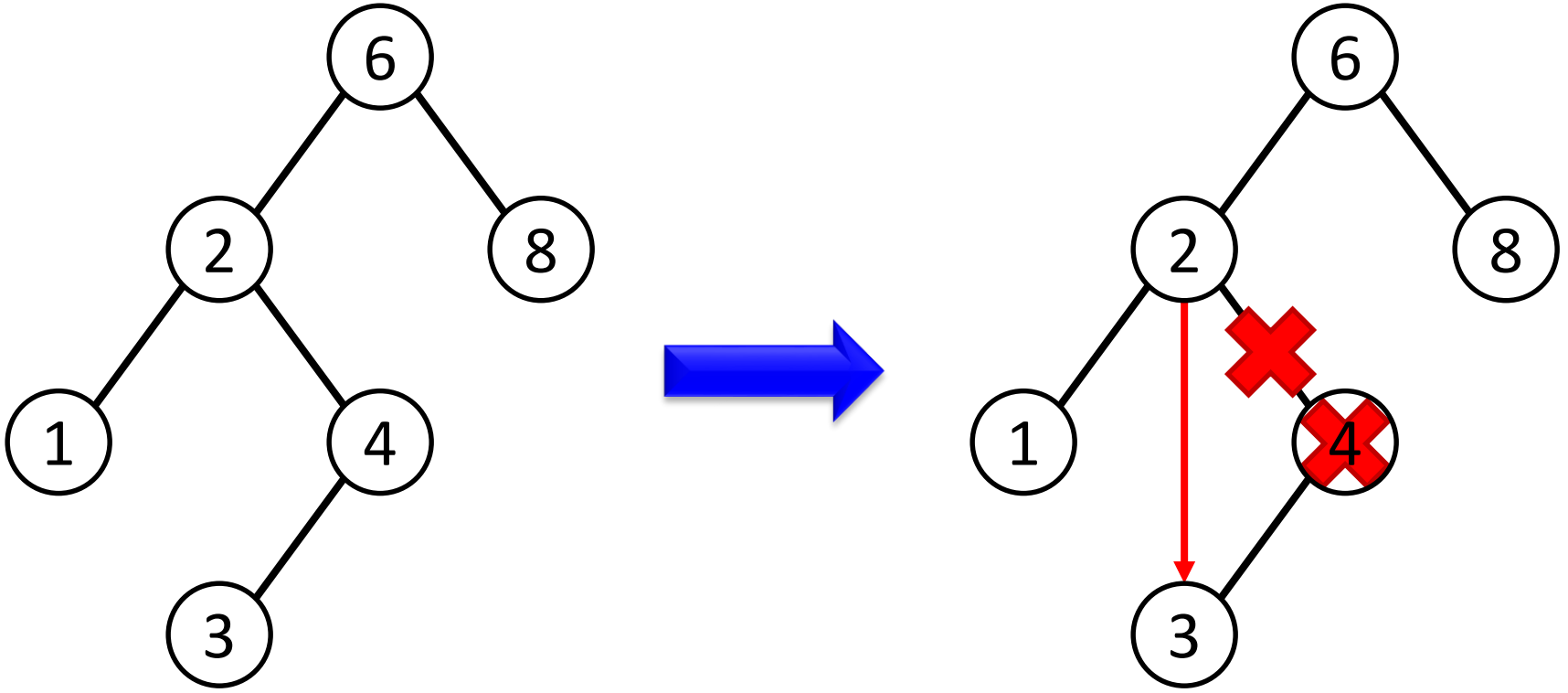
- Move to left/right depending on the value.
- Stop when the element is found (nothing to do) or a null is found.
- If not found, substitute null by the new element.

remove: simple case (no children)



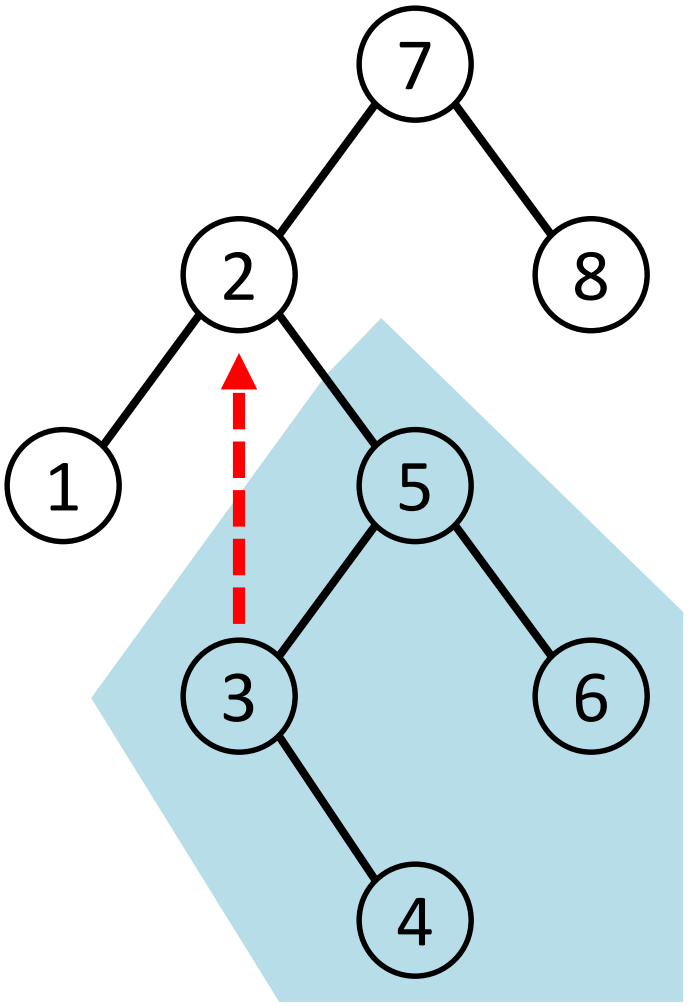
`remove(3)`

remove: simple case (one child)



`remove(4)`

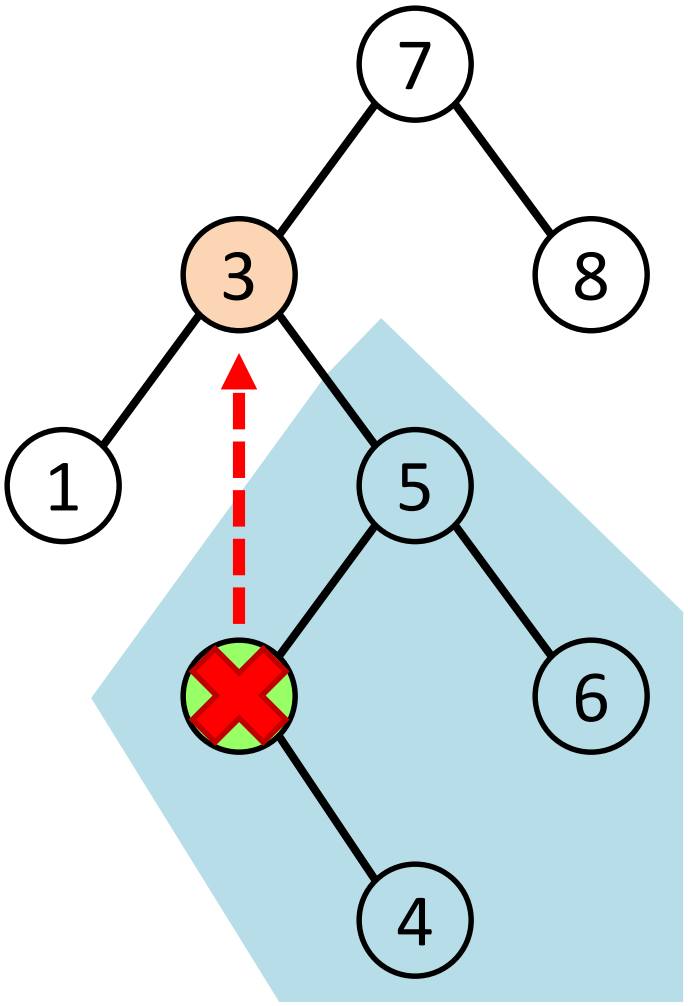
remove: complex case (two children)



`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.

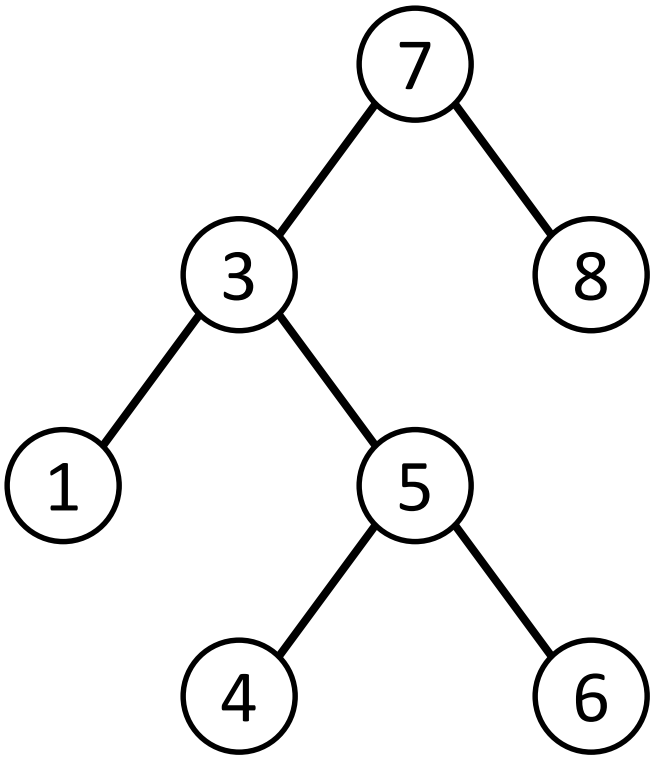
remove: complex case (two children)



`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

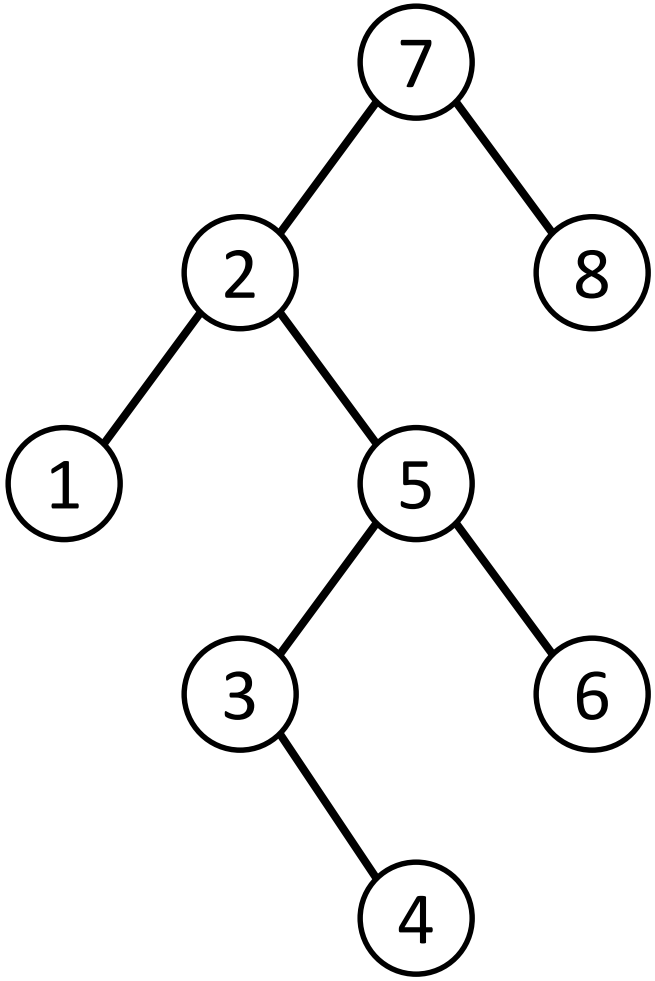
remove: complex case (two children)



1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

`remove(2)`

Visiting the items in ascending order



Question:

How can we visit the items of a BST in ascending order?

Answer:

Using an **in-order** traversal

© Dept. CS, UPC

BST: runtime analysis

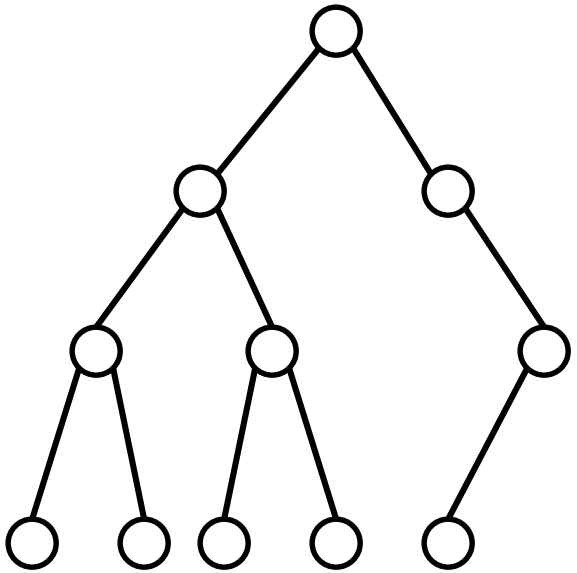
- Copying and deleting the full tree takes $\mathrm{O}(n)$.

- We are mostly interested in the runtime of the `insert/remove/contains` methods.
  - The complexity is $\mathrm{O}(d)$, where d is the depth of the node containing the required element.

- But, how large is d?

BST: runtime analysis

- Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.



$$\text{ILP} = 0 \times 1 + 1 \times 2 + 2 \times 3 + 3 \times 5 = 23$$

$$\text{Avg. IPL} = \frac{23}{11} \approx 2.09$$

BST: runtime analysis

- Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.

- $D(n)$ is the IPL of a tree with n nodes. $D(1) = 0$. The left subtree has i nodes and the right subtree has $n - i - 1$ nodes. Thus,

$$D(n) = D(i) + D(n - i - 1) + (n - 1)$$

- If all subtree sizes are equally likely, then the average value for $D(i)$ and $D(n - i - 1)$ is

$$\frac{1}{n}\sum_{j=0}^{n-1} D(j)$$

BST: runtime analysis

- Therefore,

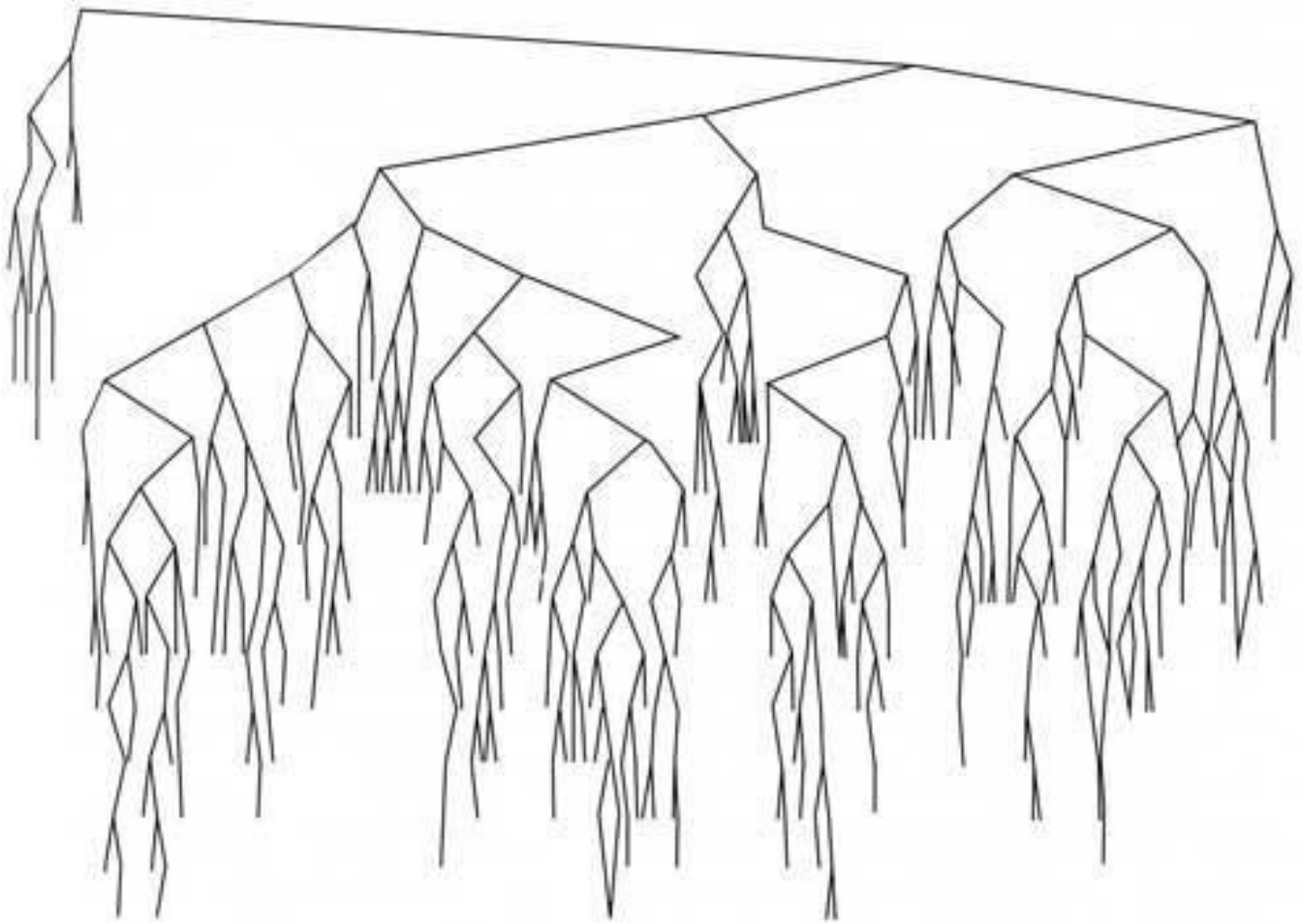
$$D(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} D(j)\right] + n - 1$$

- The previous recurrence gives: $D(n) = \mathrm{O}(n \log n)$

- The average height of nodes after n random insertions is $\mathrm{O}(\log n)$.

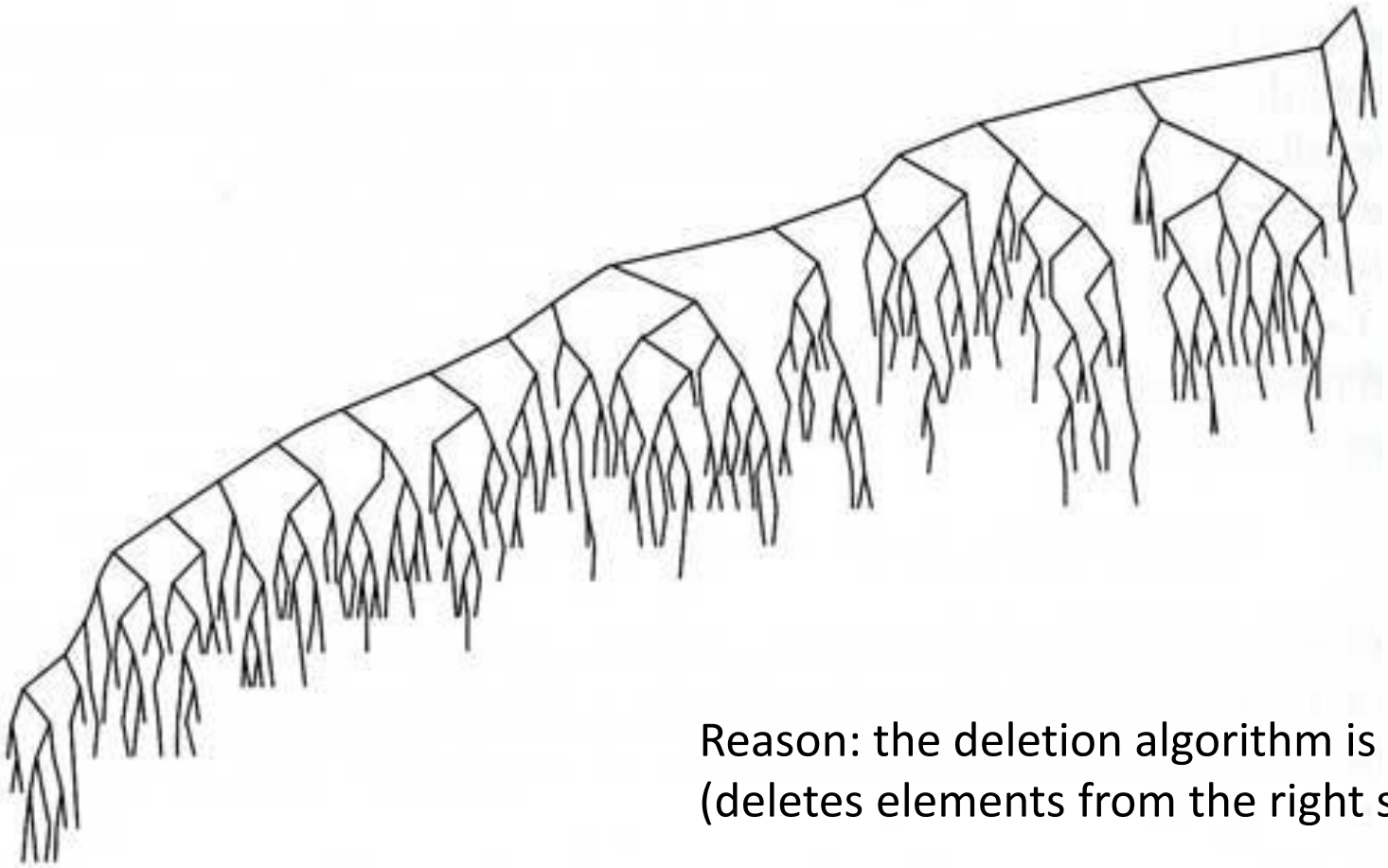
- However, the $\mathrm{O}(\log n)$ average height is not preserved when doing deletions.

Random BST



Source: Fig 4.29 of Weiss textbook

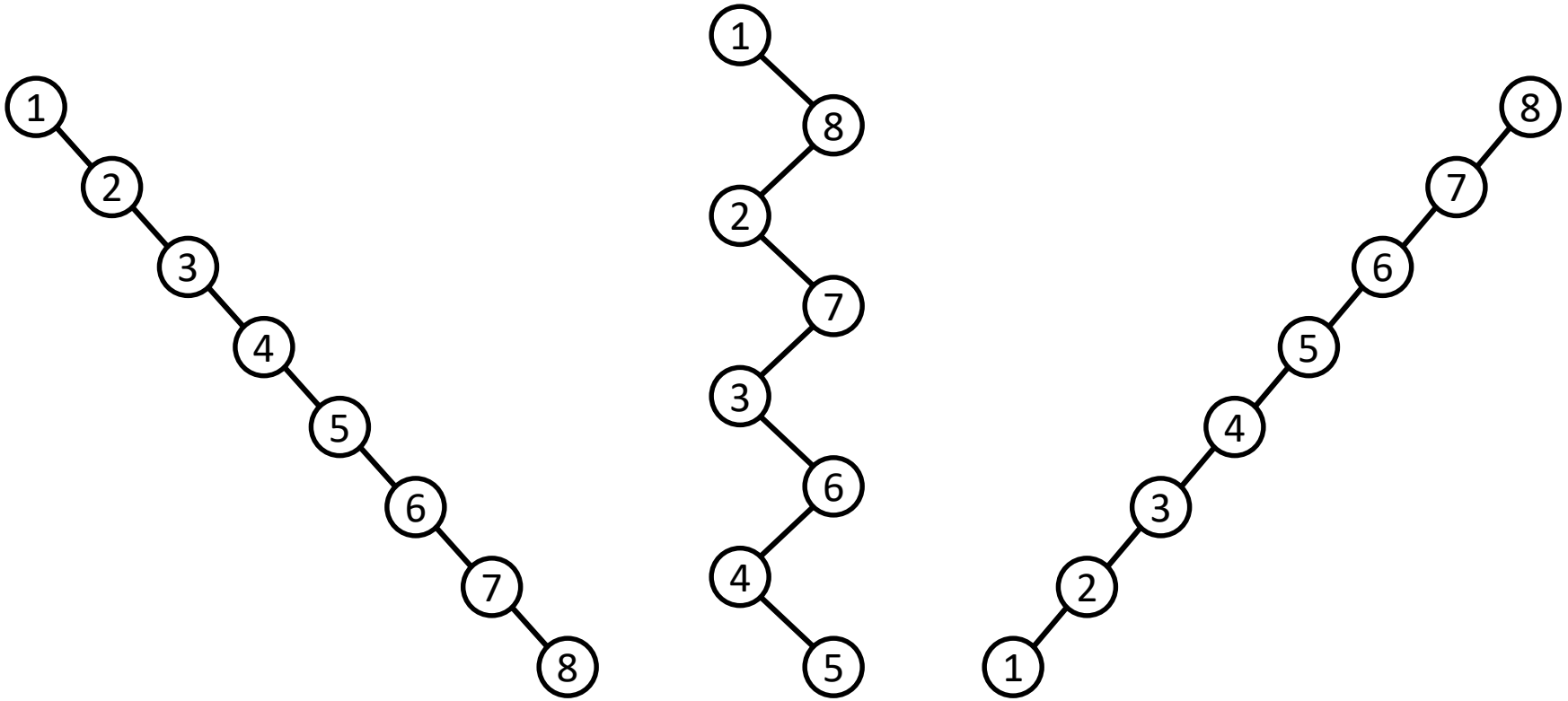
Random BST after n^2 insert/removes



Reason: the deletion algorithm is asymmetric
(deletes elements from the right subtree)

Source: Fig 4.30 of Weiss textbook

Worst-case runtime: $\mathrm{O}(n)$



Balanced trees

- The worst-case complexity for insert, remove and search operations in a BST is $\mathrm{O}(n)$, where n is the number of elements.

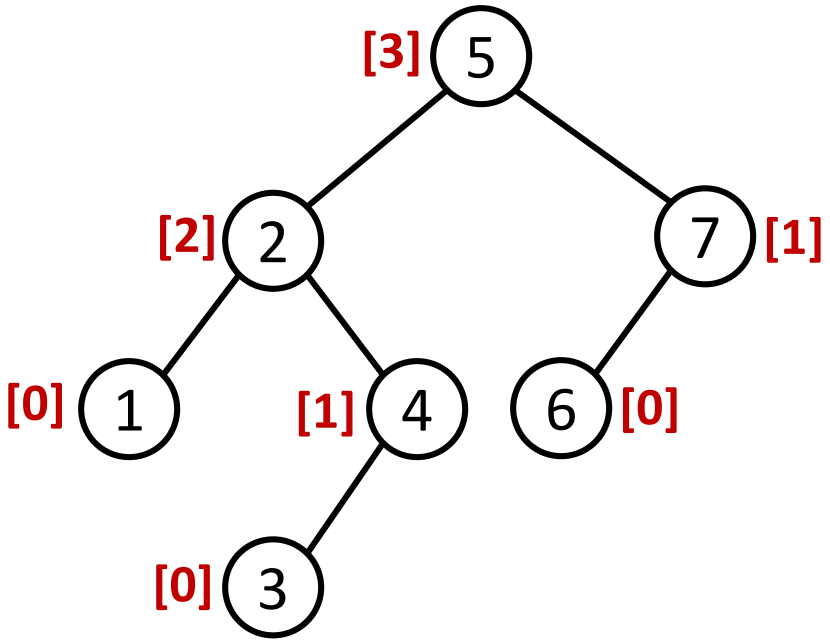
- Various representations have been proposed to keep the height of the tree as $\mathrm{O}(\log n)$:
  - AVL trees
  - Red-Black trees
  - Splay trees
  - B-trees

AVL trees

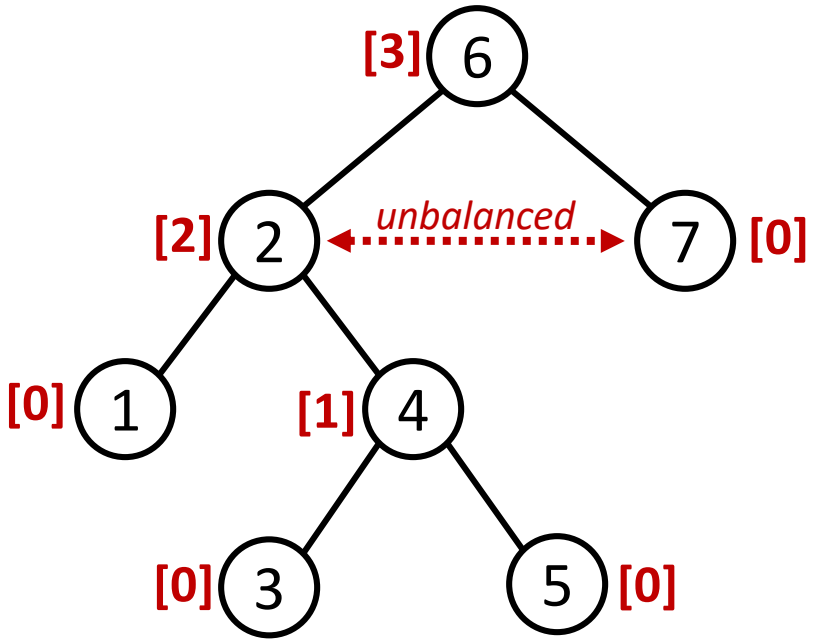
- Named after Adelson-Velsky and Landis (1962).
- Main idea: invest some additional time to balance the tree each time a new element is inserted or deleted.
- Properties:
  - The height of the tree is always $\Theta(\log n)$.
  - The time devoted to balancing is $\mathrm{O}(\log n)$.

AVL tree: definition

- An AVL tree is a BST such that, for every node, the difference between the heights of the left and right subtrees is at most 1.



AVL



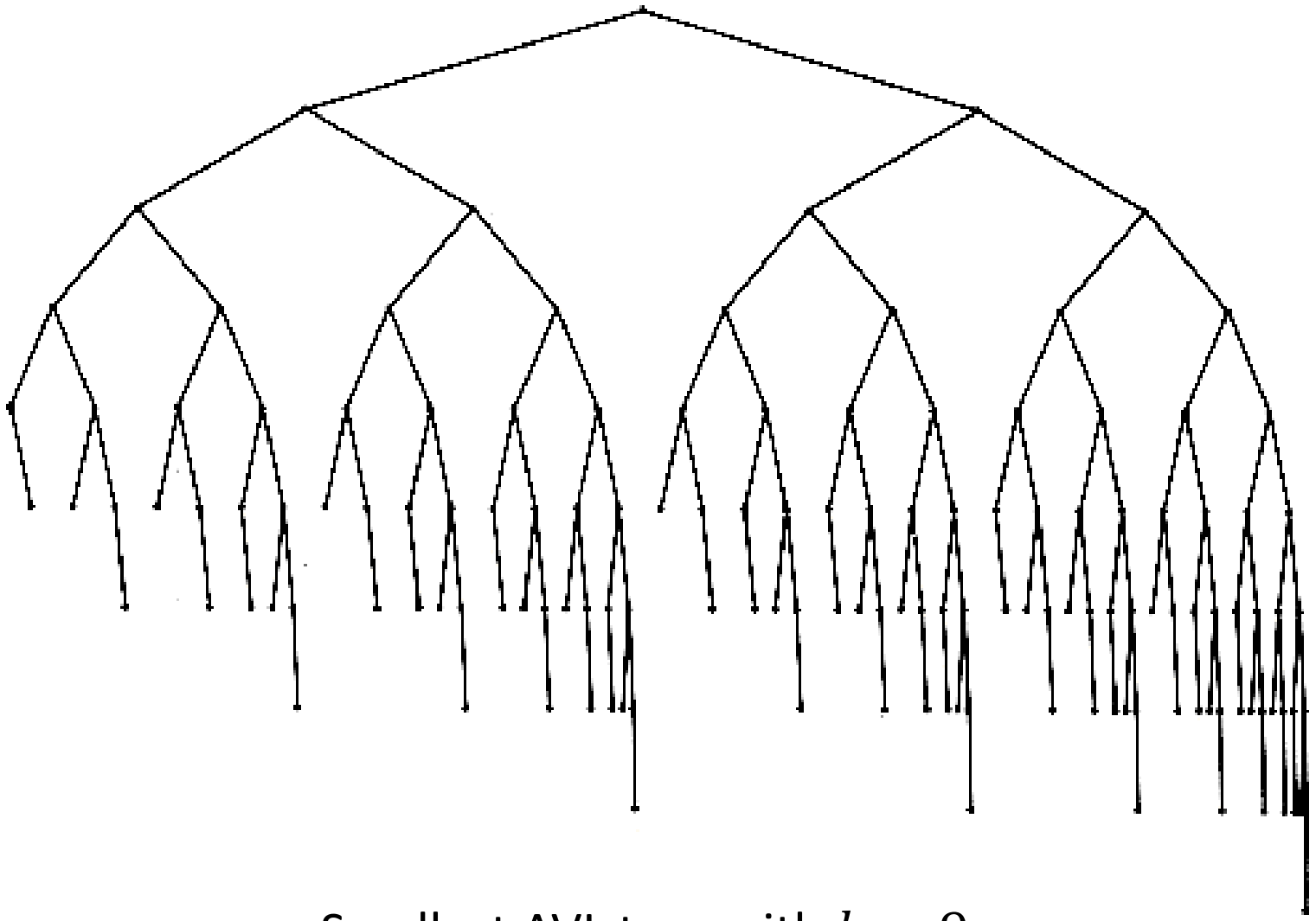
not AVL

© Dept. CS, UPC

AVL tree in action

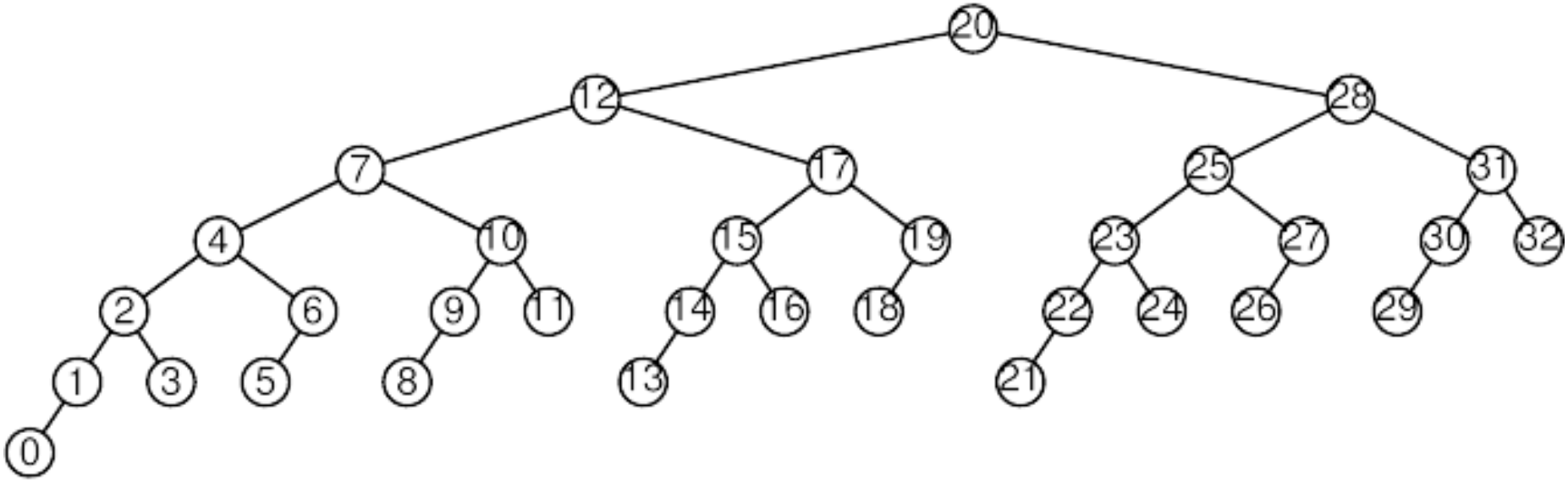
https://en.wikipedia.org/wiki/AVL_tree

AVL trees



Smallest AVL tree with $h = 9$.

AVL trees



Smallest AVL tree with $h = 6$.

The important question: what is the size of an AVL tree with height h?

Height of an AVL tree

- Theorem: The height of an AVL tree with n nodes is $\Theta(\log n)$.

- Proof in two steps:
  - The height is $\Omega(\log n)$.
  - The height is $\mathrm{O}(\log n)$.

The height is $\Omega(\log n)$

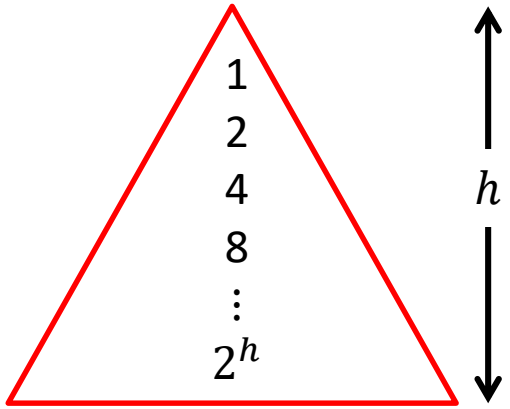
- The size n of a tree with height h is:

$$n \leq \underbrace{1 + 2 + 4 + \cdots + 2^h}_{\text{full binary tree}} = 2^{h+1} - 1$$

- Therefore,

$$\log_2(n+1) - 1 \leq h$$

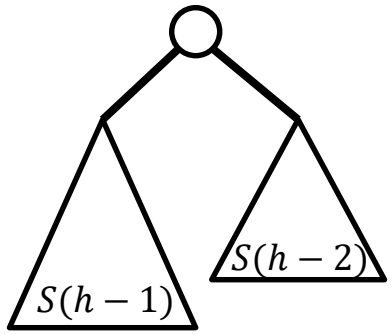
and $h = \Omega(\log n)$.



The height is $O(\log n)$

- Let $S(h)$ be the min number of nodes of an AVL tree with height h.
- One of the children (e.g., left) must have height $h-1$. The other child must have height $h-2$ (because the AVL has min size).
- Therefore,

$$S(h) = S(h-1) + S(h-2) + 1.$$



- Thus,

$$S(h) \geq 2 \cdot S(h-2).$$

- Given that $S(0) = 1$ and $S(1) = 2$, it can be easily proven, by induction, that:

$$S(h) \geq 2^{h/2}$$

- Since $n \geq S(h)$ and $S(h) \geq 2^{h/2}$, then $\log_2 n \geq h/2$:

$$h = O(\log n)\,.$$

Height of an AVL tree

- The recurrence

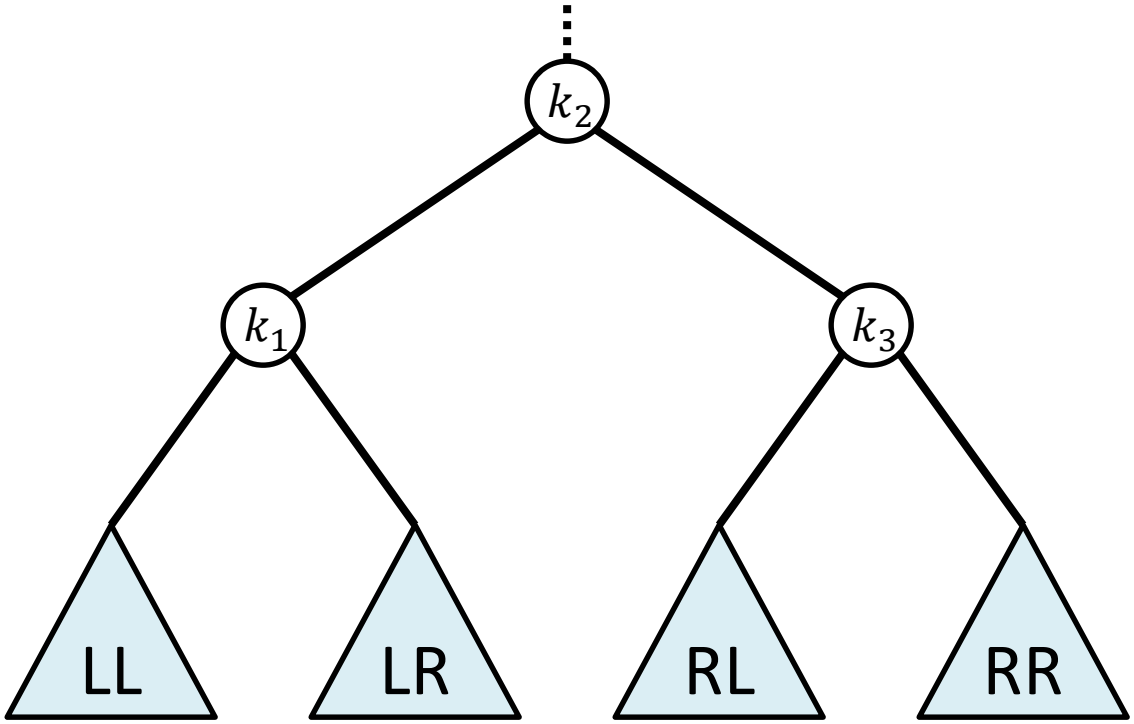
$$S(h) = S(h-1) + S(h-2) + 1$$

resembles the one of the Fibonacci numbers. A tighter bound can be obtained.

- Theorem: the height of an AVL tree with n internal nodes satisfies:

$$h < 1.44 \log_2(n+2) - 1.328$$

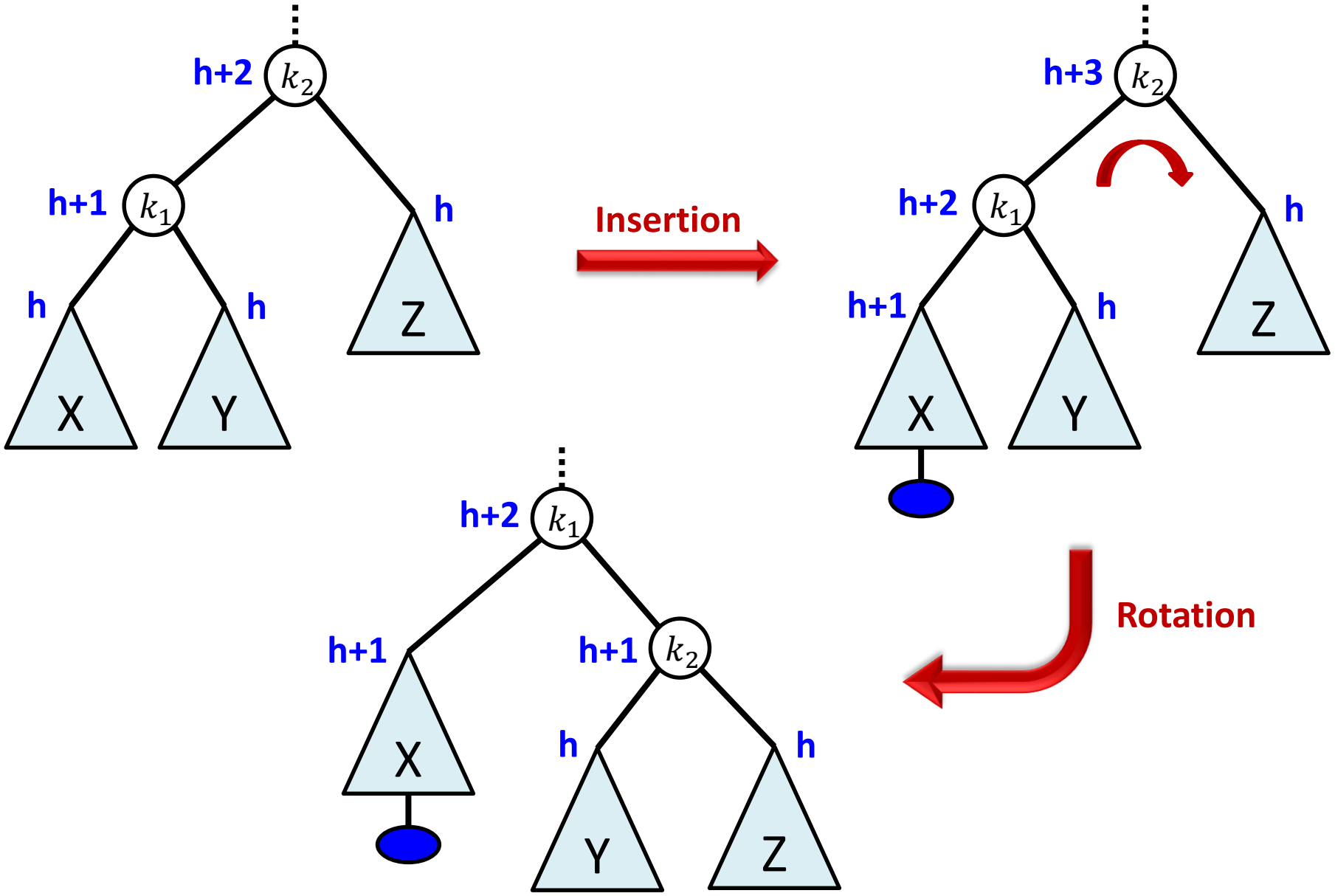
Unbalanced insertion: 4 cases



Any newly inserted item may fall into any of the four subtrees (LL, LR, RL or RR).

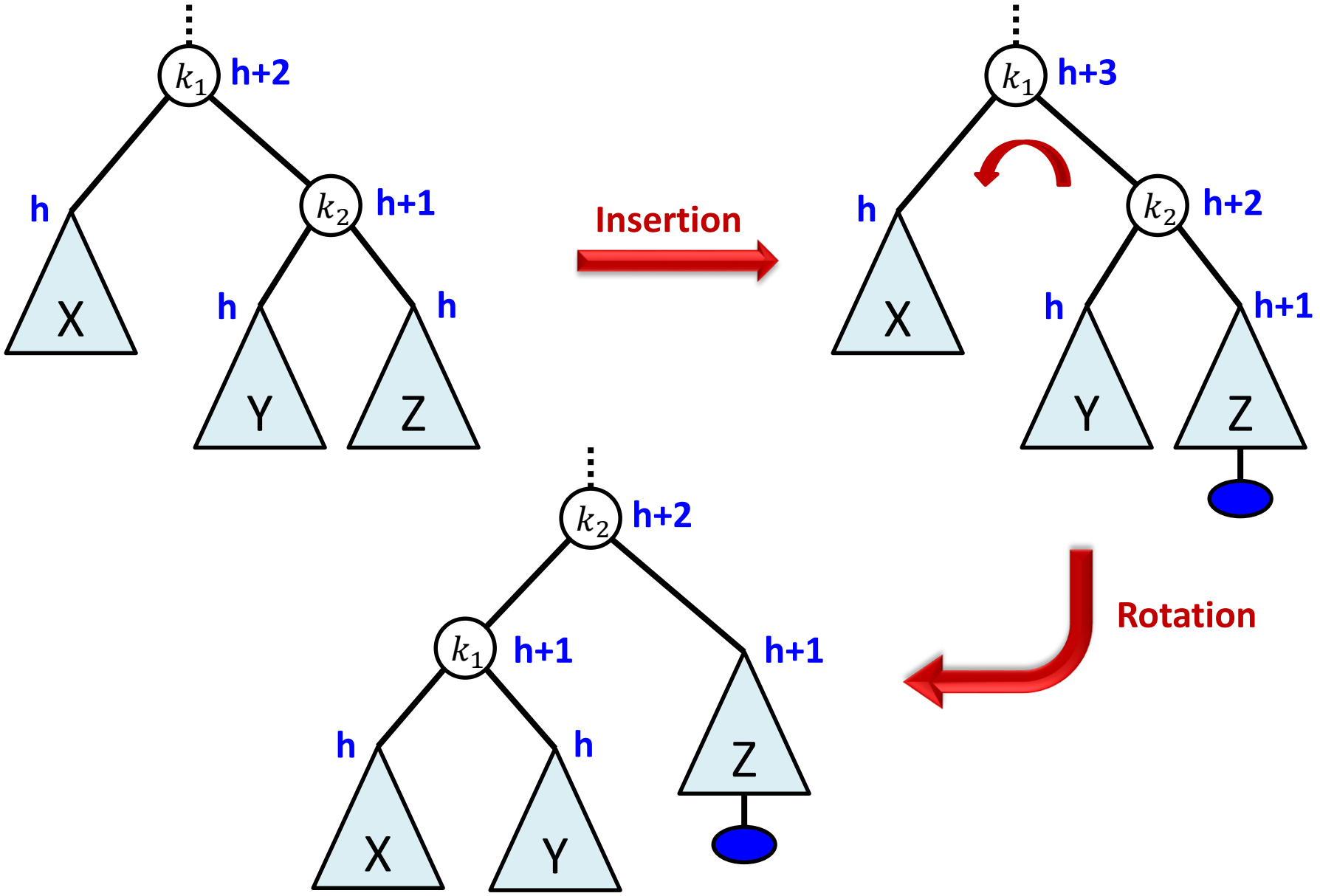
A new insertion may violate the balancing property. Re-balancing might be required.

Single rotation: the left-left case



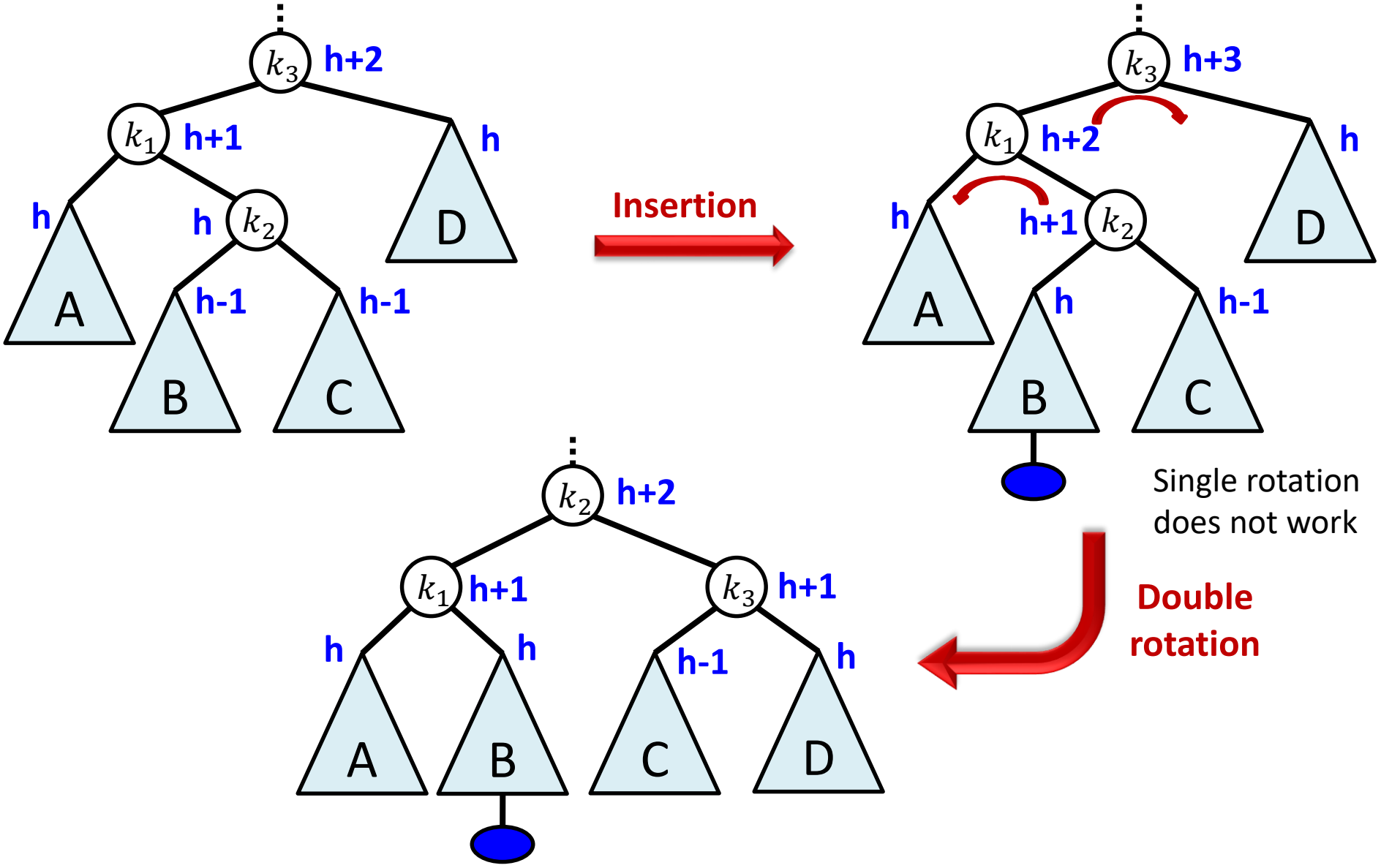
© Dept. CS, UPC

Single rotation: the right-right case

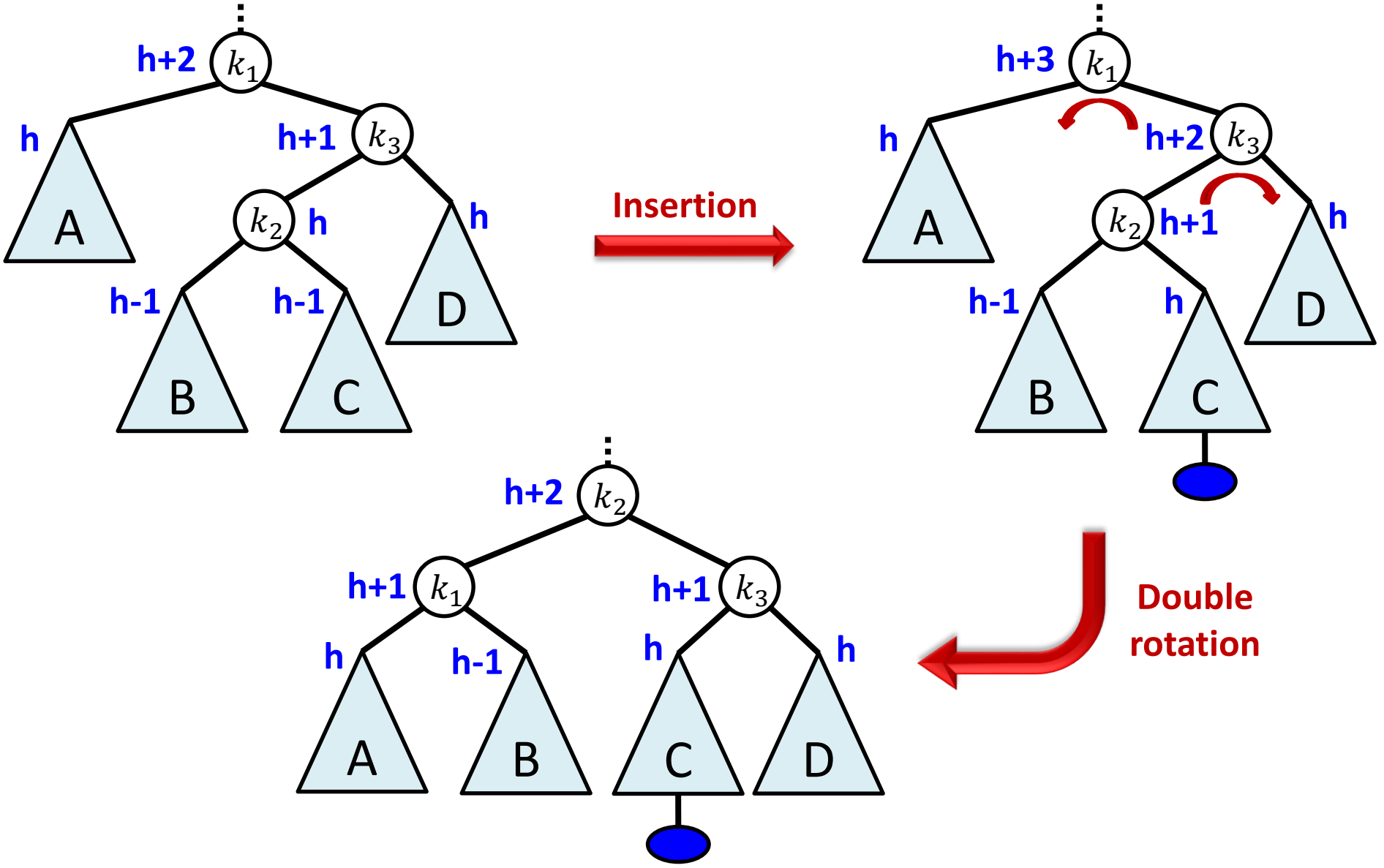


© Dept. CS, UPC

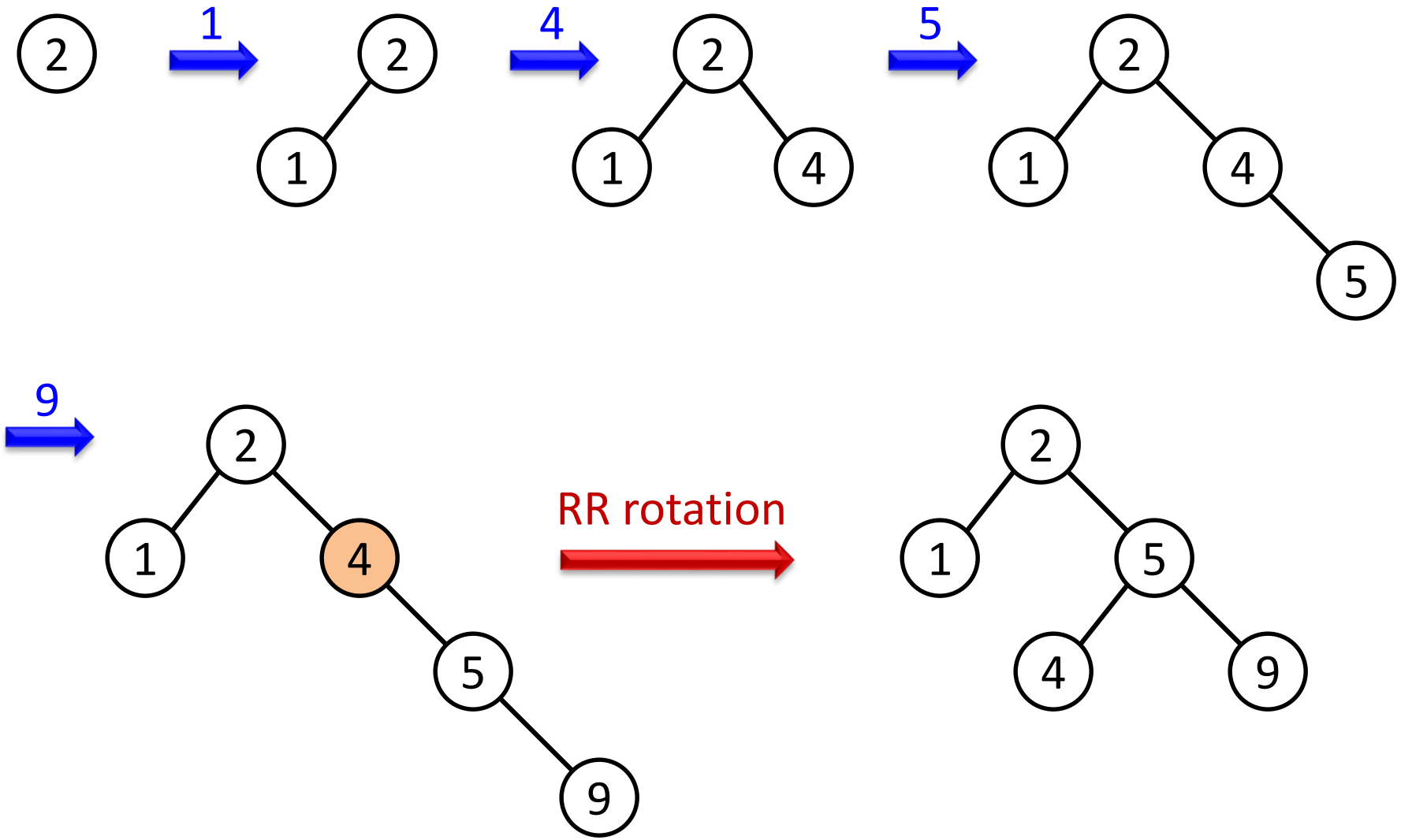
Double rotation: the left-right case



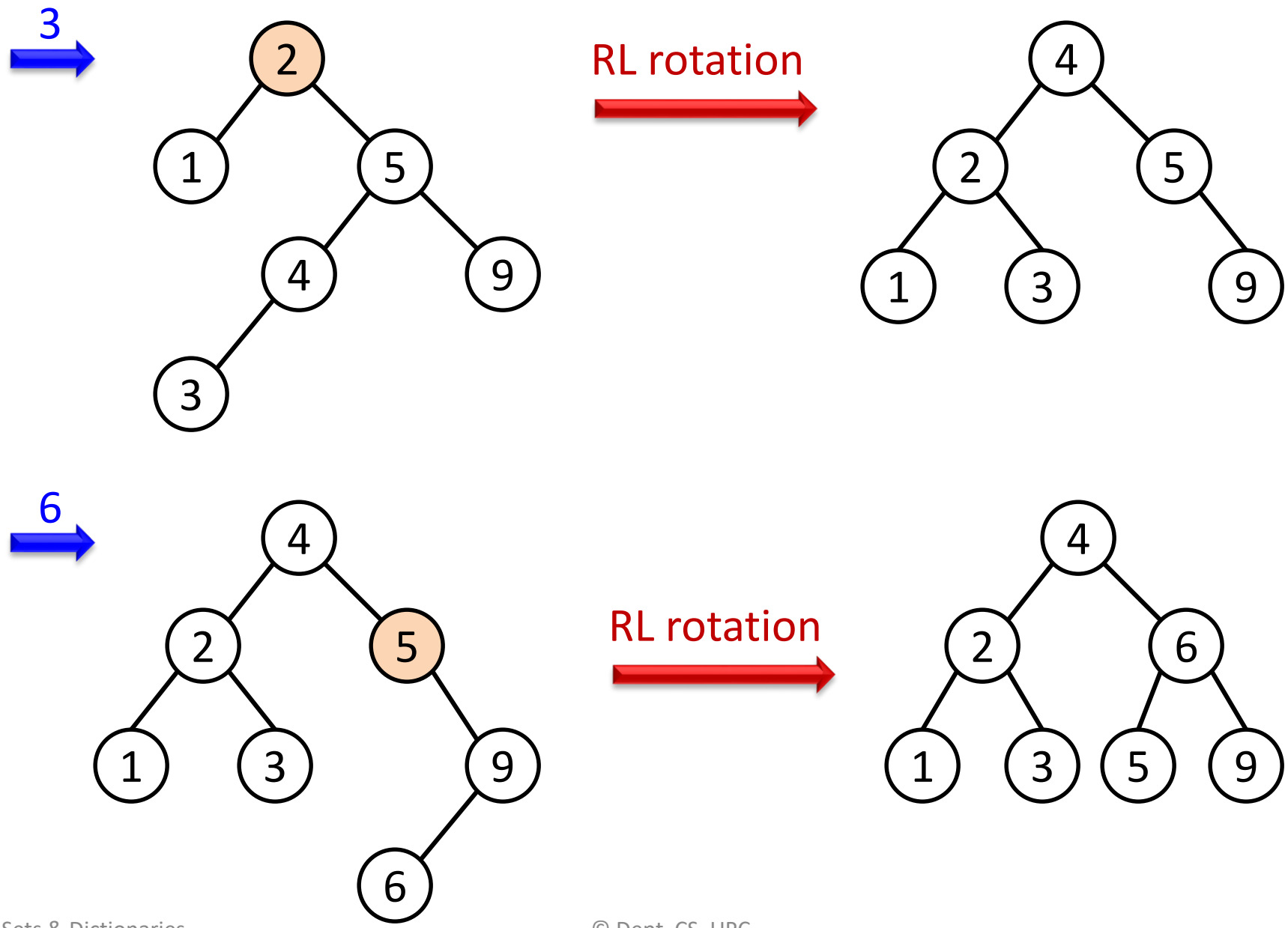
Double rotation: the right-left case



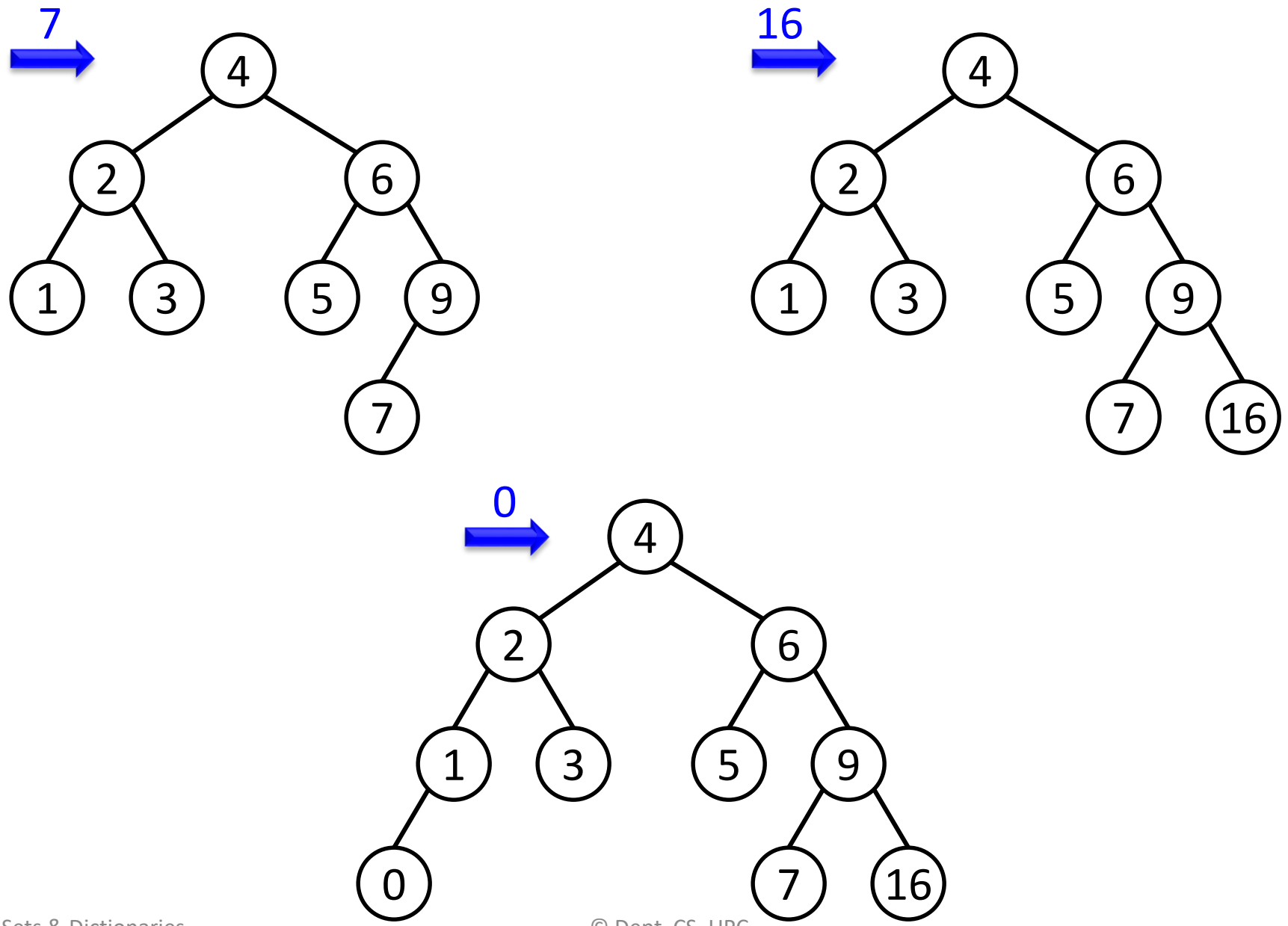
Example: insertions



Example: insertions

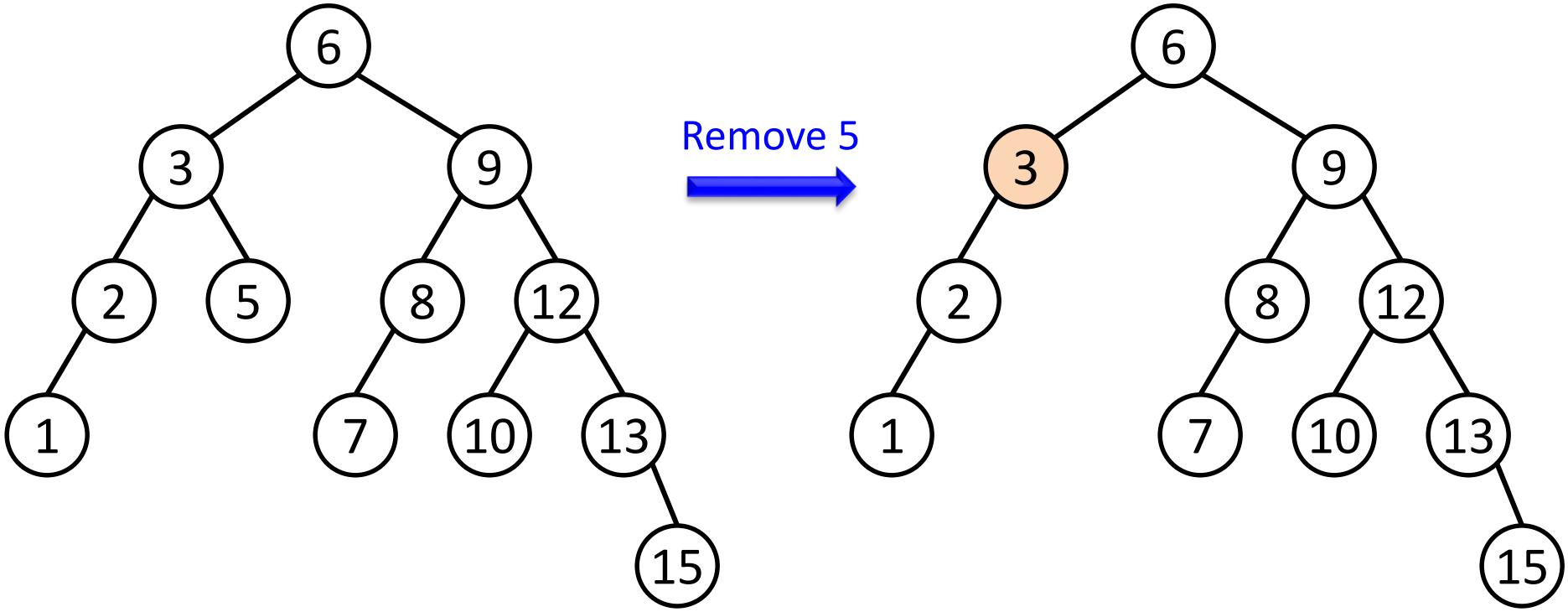


Example: insertions



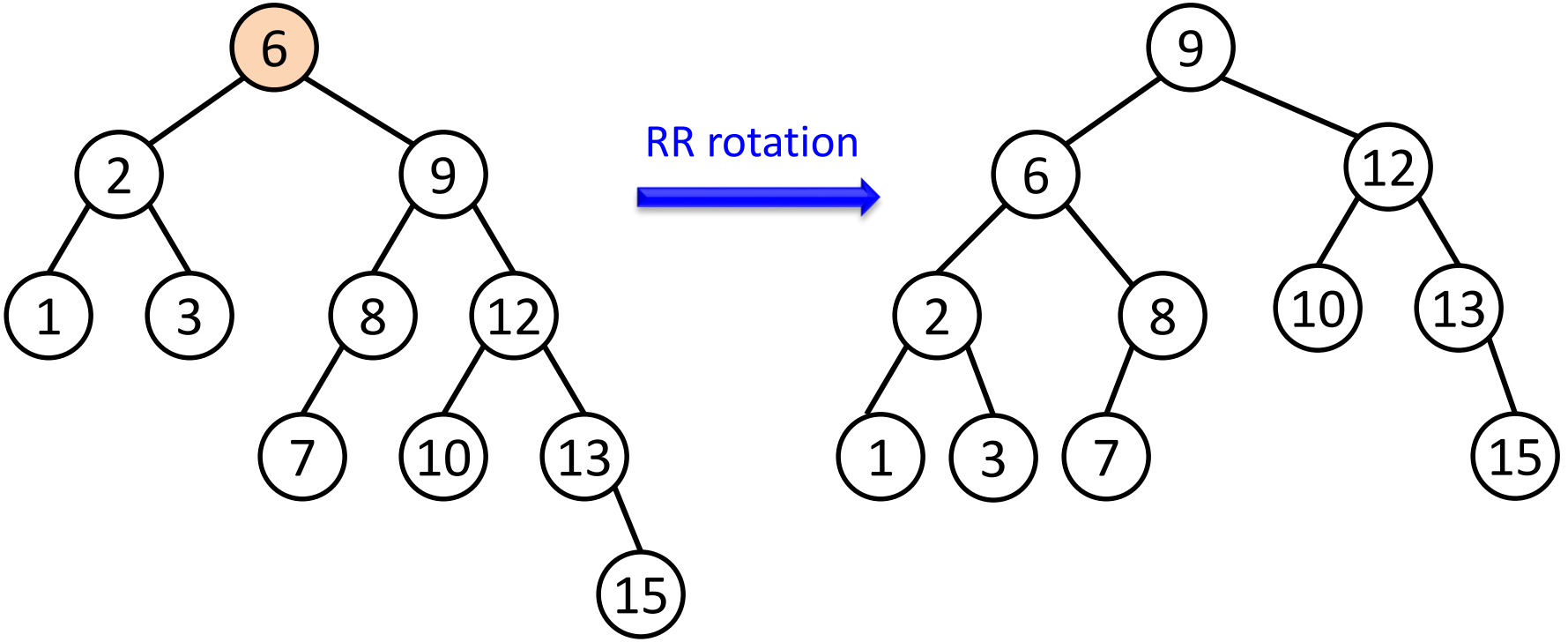
© Dept. CS, UPC

Example: deletion



Apply LL rotation on 3

Example: deletion



Implementation details

- The height must be stored at each node. Only the unbalancing factor ({-1,0,1}) is strictly required.
- The insertion/deletion operations are implemented similarly as in BSTs (recursively).
- The re-balancing of the tree is done when the recursive calls return to the ancestors (check heights and rotate if necessary).

Complexity

- Single and double rotations only need the manipulation of few pointers and the height of the nodes ($O(1)$).

- Insertion: the height of the subtree after a rotation is the same as the height before the insertion. Therefore, at most only one rotation must be applied for each insertion.

- Deletion: more complicated. More than one rotation might be required.

- Worst case for deletion: $O(\log n)$ rotations (a chain effect from leaves to root).

EXERCISES

BST

- Starting from an empty BST, depict the BST after inserting the values 32, 15, 47, 67, 78, 39, 63, 21, 12, 27.

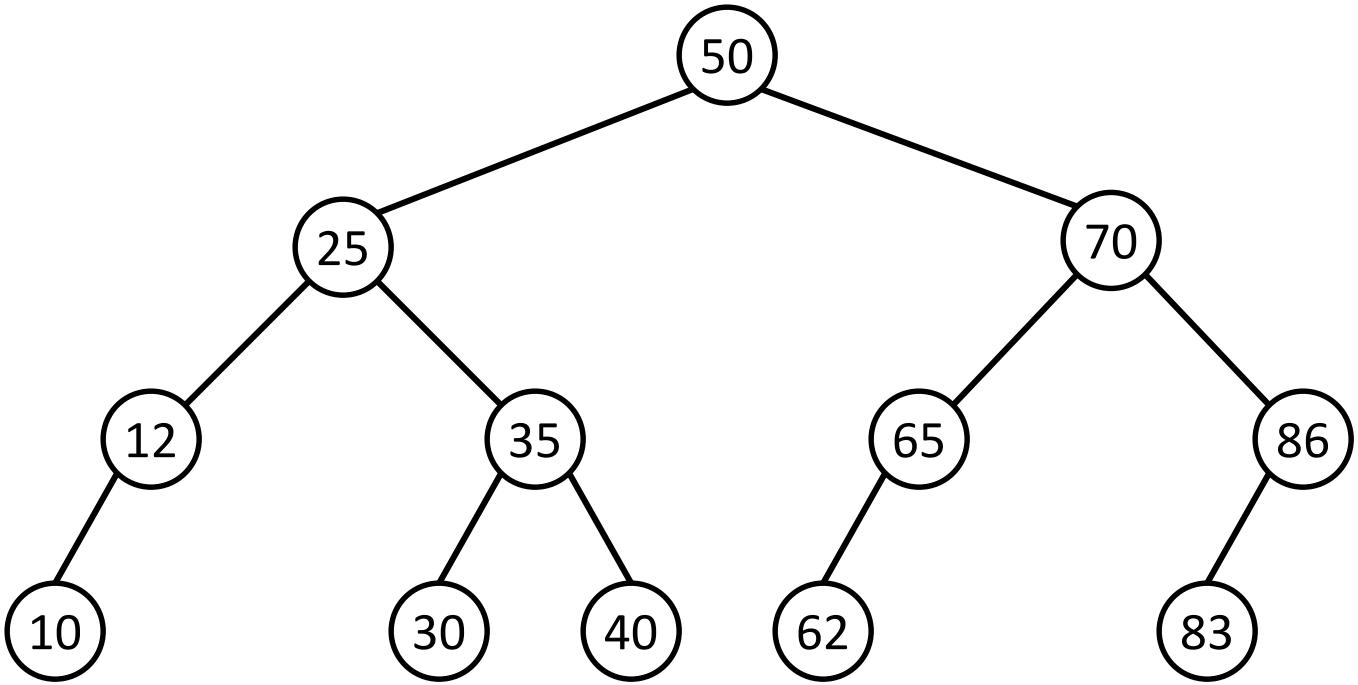
- Depict the previous BST after removing the values 63, 21, 15 and 32.

Merging BSTs

- Describe an algorithm to generate a sorted list from a BST. What is its cost?
- Describe an algorithm to create a balanced BST from a sorted list. What is its cost?
- Describe an algorithm to create a balanced BST that contains the union of the elements of two BSTs. What is its cost?

AVL

Depict the three AVL trees after sequentially inserting the values 31, 32 and 33 in the following AVL tree:



AVL

- Build an AVL tree by inserting the following values: 15, 21, 23, 11, 13, 8, 32, 33, 27. Show the tree before and after applying each rotation.

- Depict the AVL tree after removing the elements 23 and 21 (in this order). When removing an element, move up the largest element of the left subtree.