Balanced Search Tree

Luciano Bononi
Dip. di Scienze dell'Informazione
Università di Bologna

bononi@cs.unibo.it

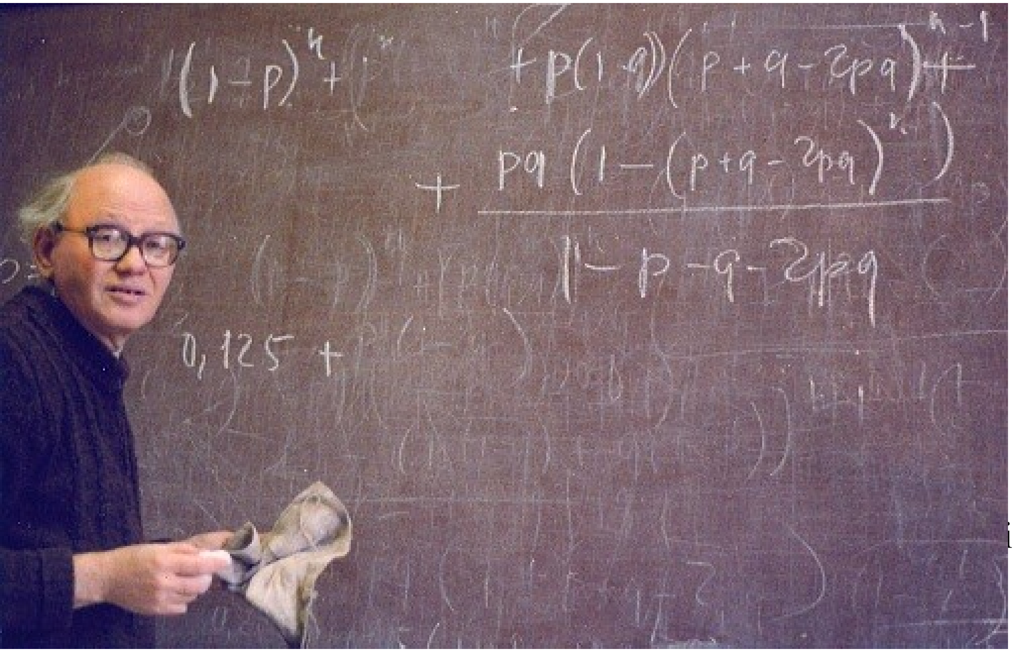
Introduction

- We have seen that in BST we can search, delet and insert nodes with given key k in O(h) where h=heigth of the tree
  - A complete binary tree with n nodes has heigth $h=\Theta(\log n)$
- However, insertion and deletion of nodes could unbalance the tree
  - question: identify a sequence of n insertions in a BST (initially empty) such that the resulting BST has heigth $\Theta(n)$
- Our aim: keep balanced a BST despite insertions and deletions

AVL tree

- an AVL tree is a search tree (almost) balanced
  - AVL tree with n nodes supports insert(), delete(), lookup() operations with cost O(log n) in the worst case
  - Adelson-Velskii, G.; E. M. Landis (1962). *"An algorithm for the organization of information"*. Proceedings of the USSR Academy of Sciences 146: 263–266

Georgy Maximovich Adelson-Velsky (1922—)
http://chessprogramming.wikispaces.com/Georgy+Adelson-Velsky



Evgenii Mikhailovich Landis (1921—1997)
http://en.wikipedia.org/wiki/Yevgeniy_Landis

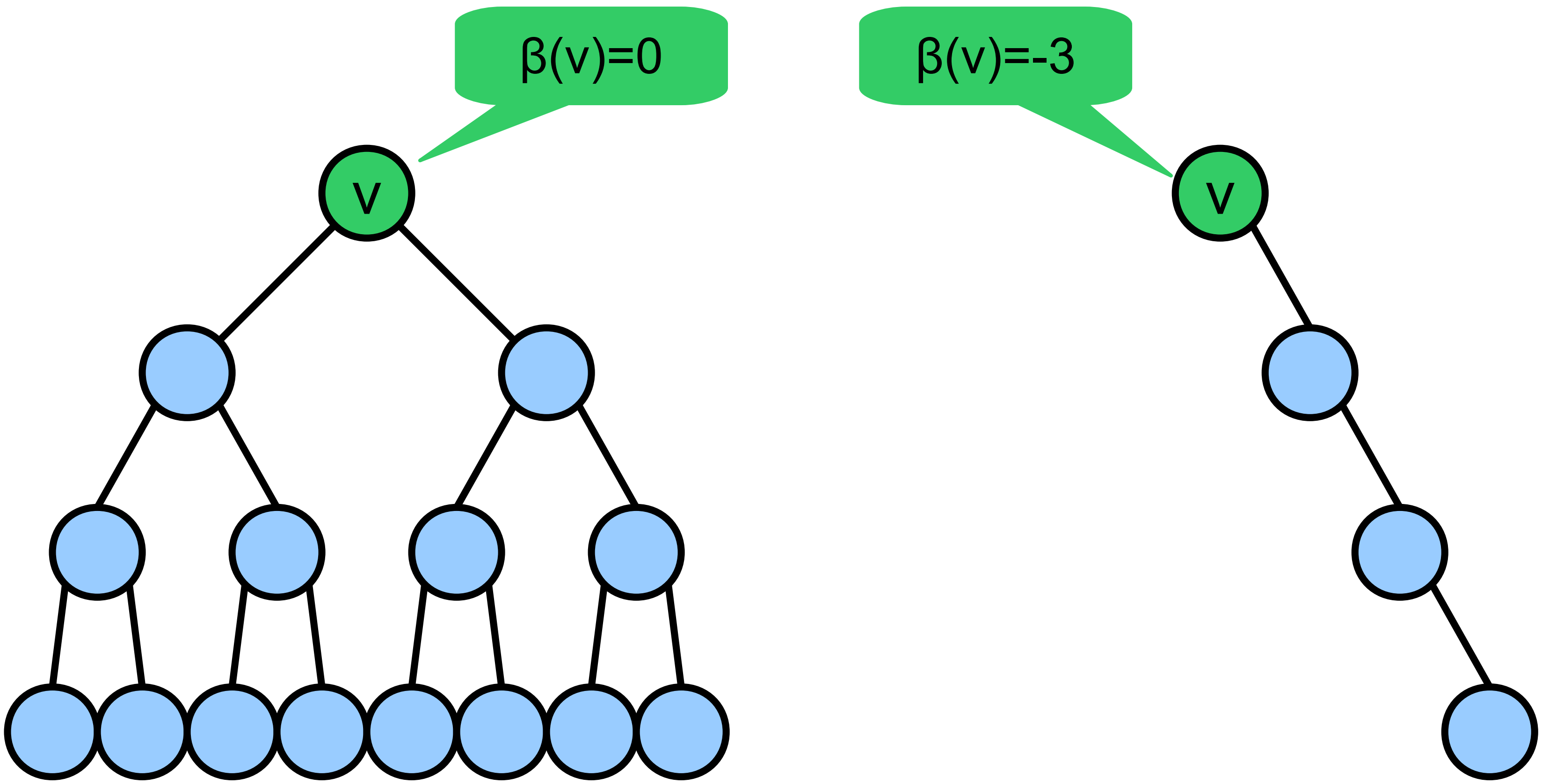
definitions

- Balancing factor
  - The balancing factor $\beta(v)$ of node v is the difference of heigth of left and right subtrees of v (in order):

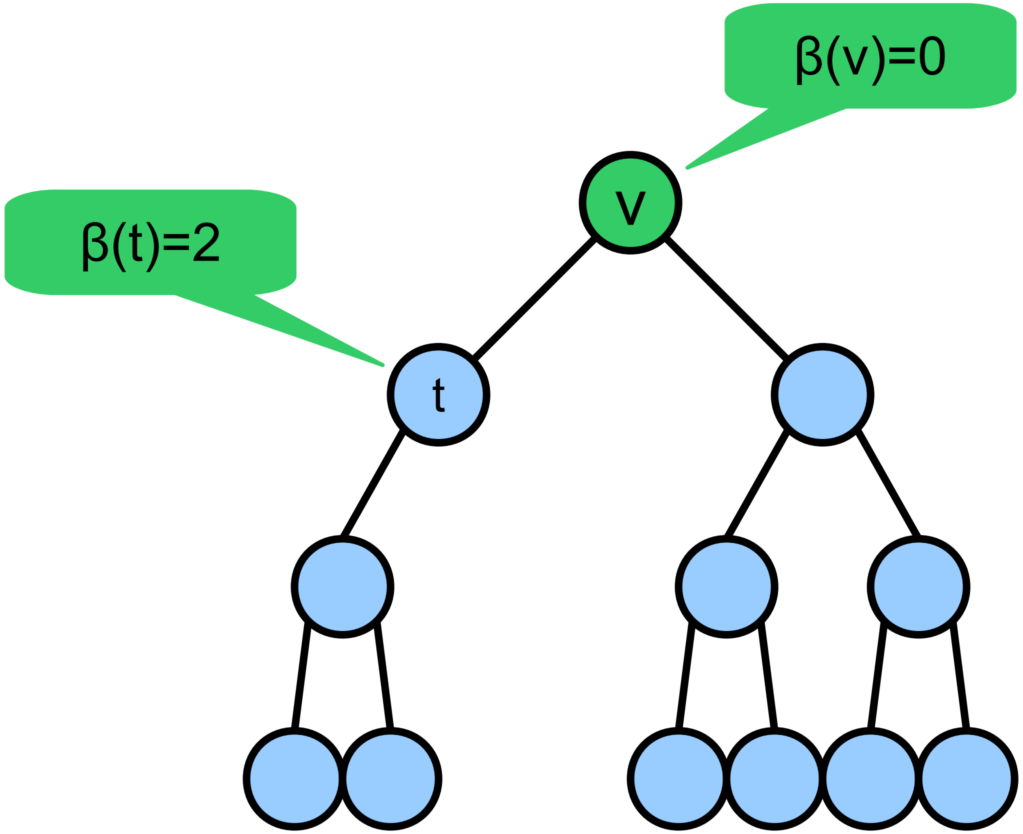
$$\beta(v) = \text{heigth}(\text{left}(v)) - \text{heigth}(\text{right}(v))$$

- Heigth balancing
  - A tree is said to be balanced in heigth if the heigth of subtrees left and right of each node v is at most 1
  - In other words a tree is balanced in heigth is for any node v, $|\beta(v)| \leq 1$
- Definition: an AVL tree is a BST balanced in heigth.

Example

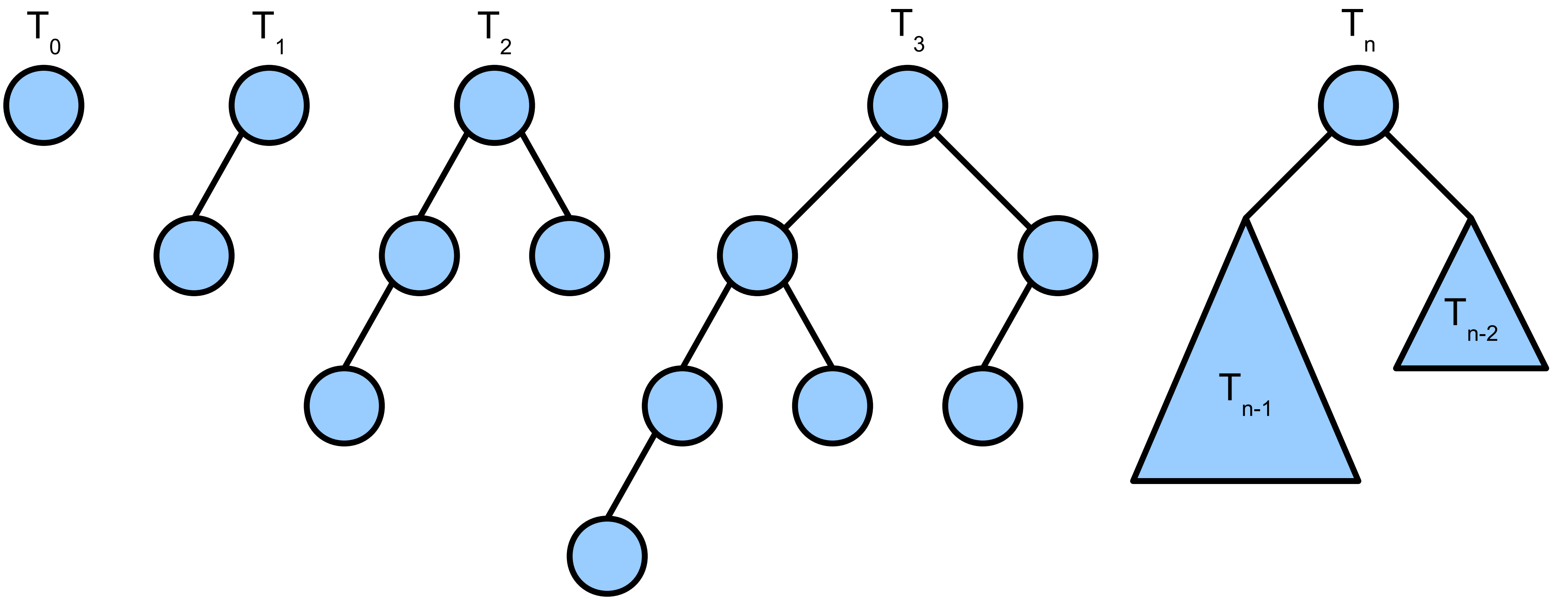


Example



Heigth of an AVL tree

- To evaluate the heigth of AVL trees, we start considering the most “unbalanced” trees we can realize.
- Fibonacci trees



Heigth of a Fibonacci tree

- Given a Fibonacci tree of heigth h, let n_h be the number of nodes.
- We get (by construction) that

$$n_h = n_{h-1} + n_{h-2} + 1$$

- We proof that

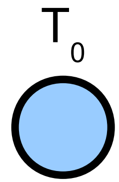
$$n_h = F_{h+3} - 1$$

where F_n is the n-th Fibonacci number.

Heigth of a Fibonacci tree

$$n_h = F_{h+3} - 1$$

- Base step: h=0
  - $n_0 = 1$
  - $F_3 = 2$



- Inductive step

$$\begin{aligned} n_h &= n_{h-1} + n_{h-2} + 1 \\ &= (F_{h+2} - 1) + (F_{h+1} - 1) + 1 \\ &= F_{h+2} + F_{h+1} - 1 \\ &= F_{h+3} - 1 \end{aligned}$$

Heigth of a Fibonacci tree

- hence: a Fibonacci tree with heigth h has $F_{h+3} - 1$ nodes
- We note that

$$F_h = \Theta(\phi^h), \phi \approx 1.618$$

hence

$$n_h = F_{h+3} - 1 = \Theta(\phi^h)$$

and we conclude that

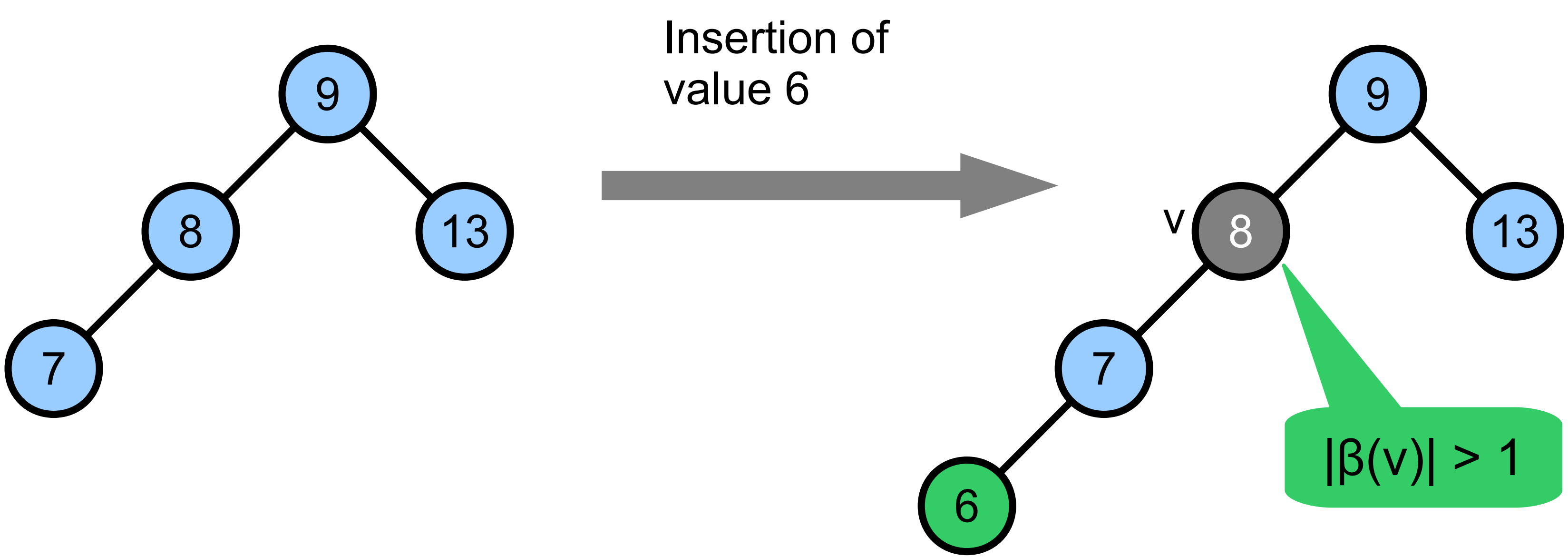
$$h = \Theta(\log n_h)$$

Conclusion

- Given that...
  - A Fibonacci tree with n nodes is the AVL tree with maximum heigth (and n nodes)
  - Heigth of a Fibonacci tree with n nodes is proportional to (log n)
- ...we conclude:
  - The heigth of a AVL tree with n nodes is O(log n)

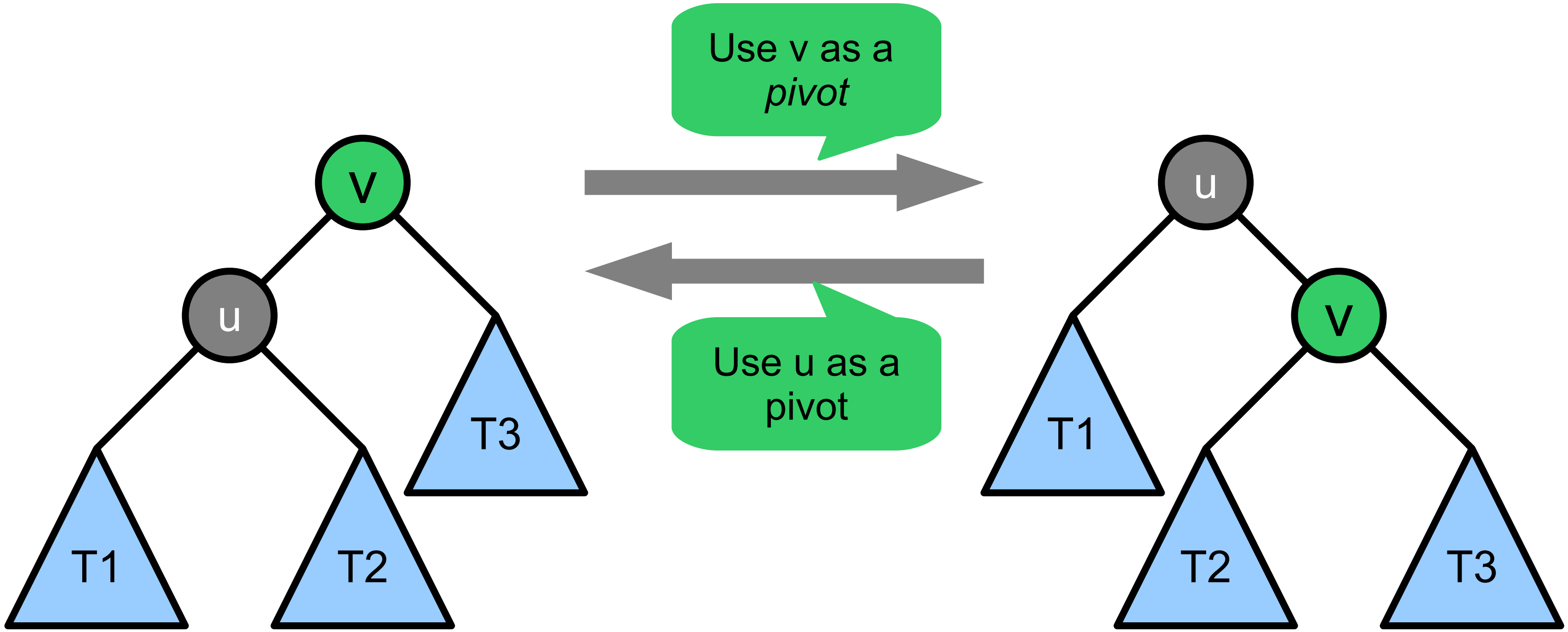
How to keep the AVL balanced?

- The search() operation in a AVL tree is made as in a generic BST (no modifications)
- Unfortunately, Insert() and delete() require to be modified to maintain the balancing of the AVL tree
- Example



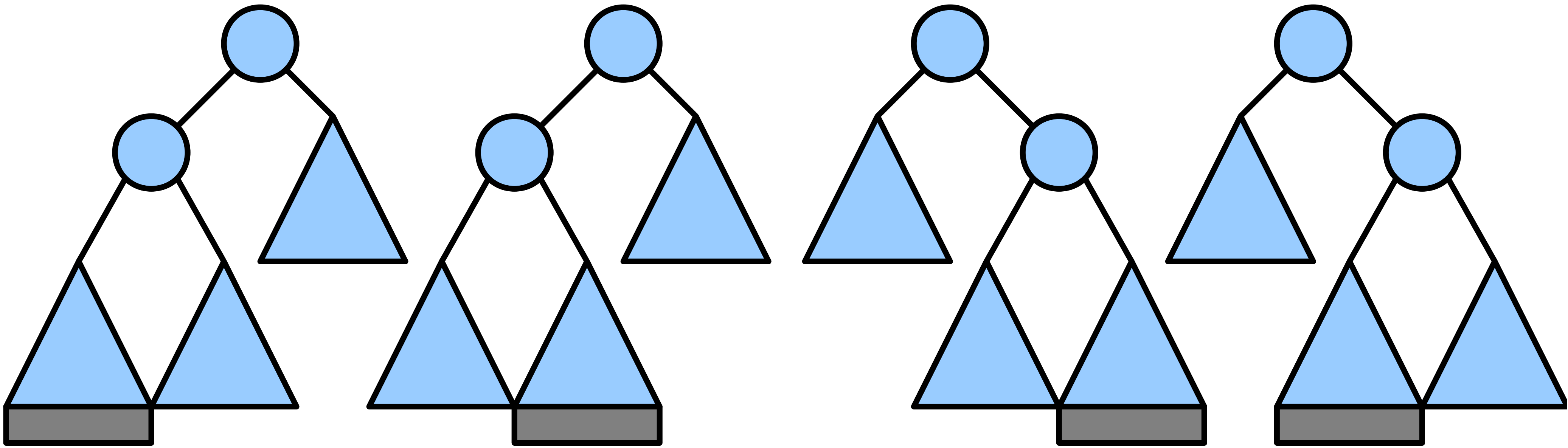
Rotation operation

- A new fundamental operation to be implemented for balancing the AVL tree is the simple rotation
  - question: proof that the simple rotation preserves the order relationship of a BST



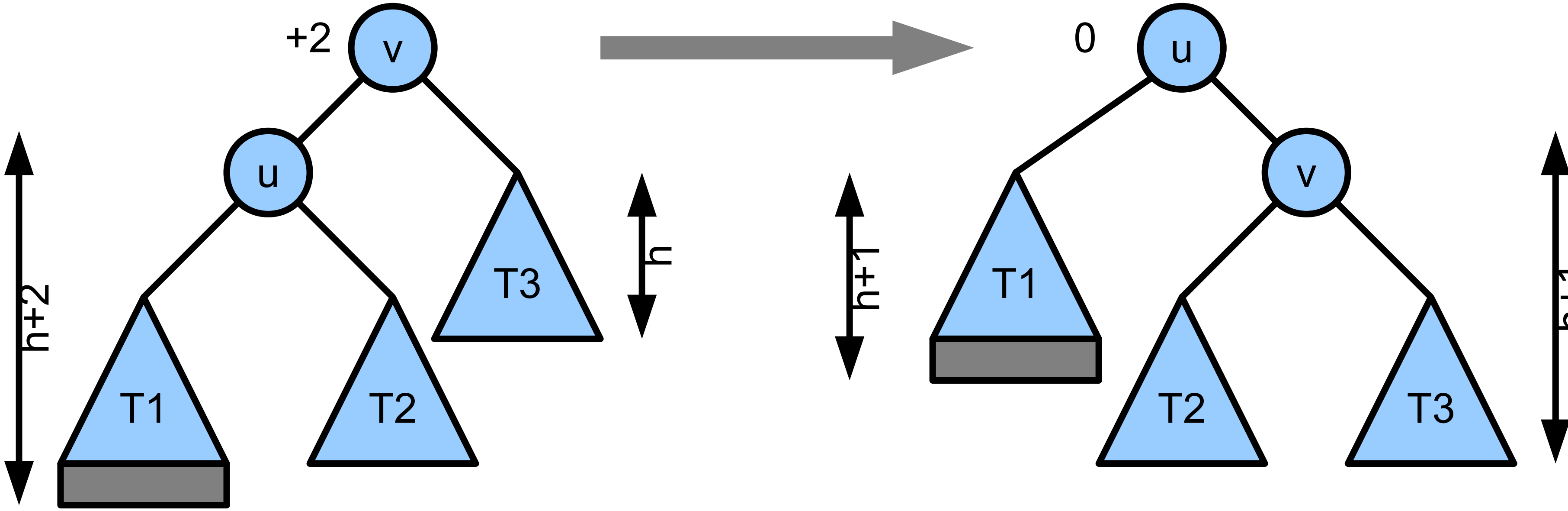
Rotations

- Let's assume that after a insert() or delete() the AVL tree is unbalanced.
- We have 4 cases (symmetry between 1-2 and 3-4)

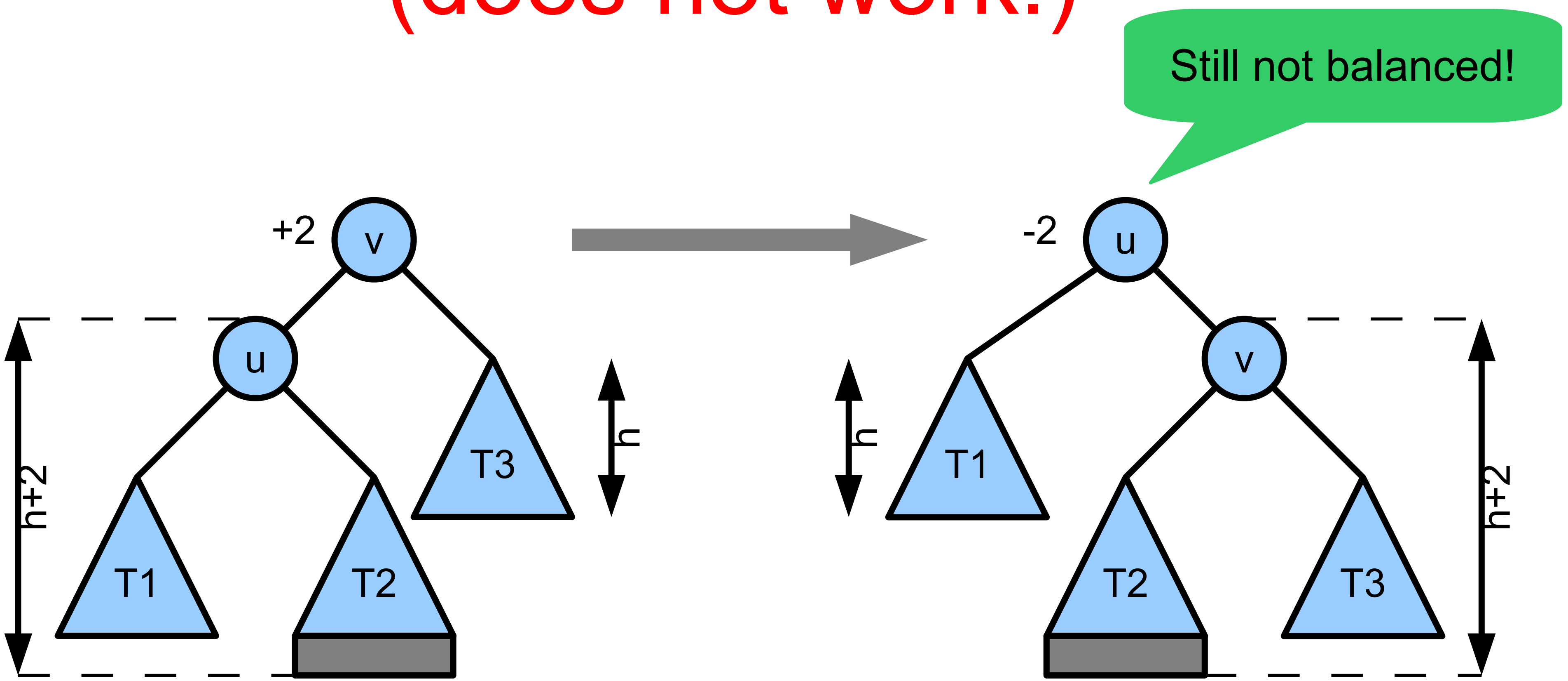


Rebalancing: rotation SS

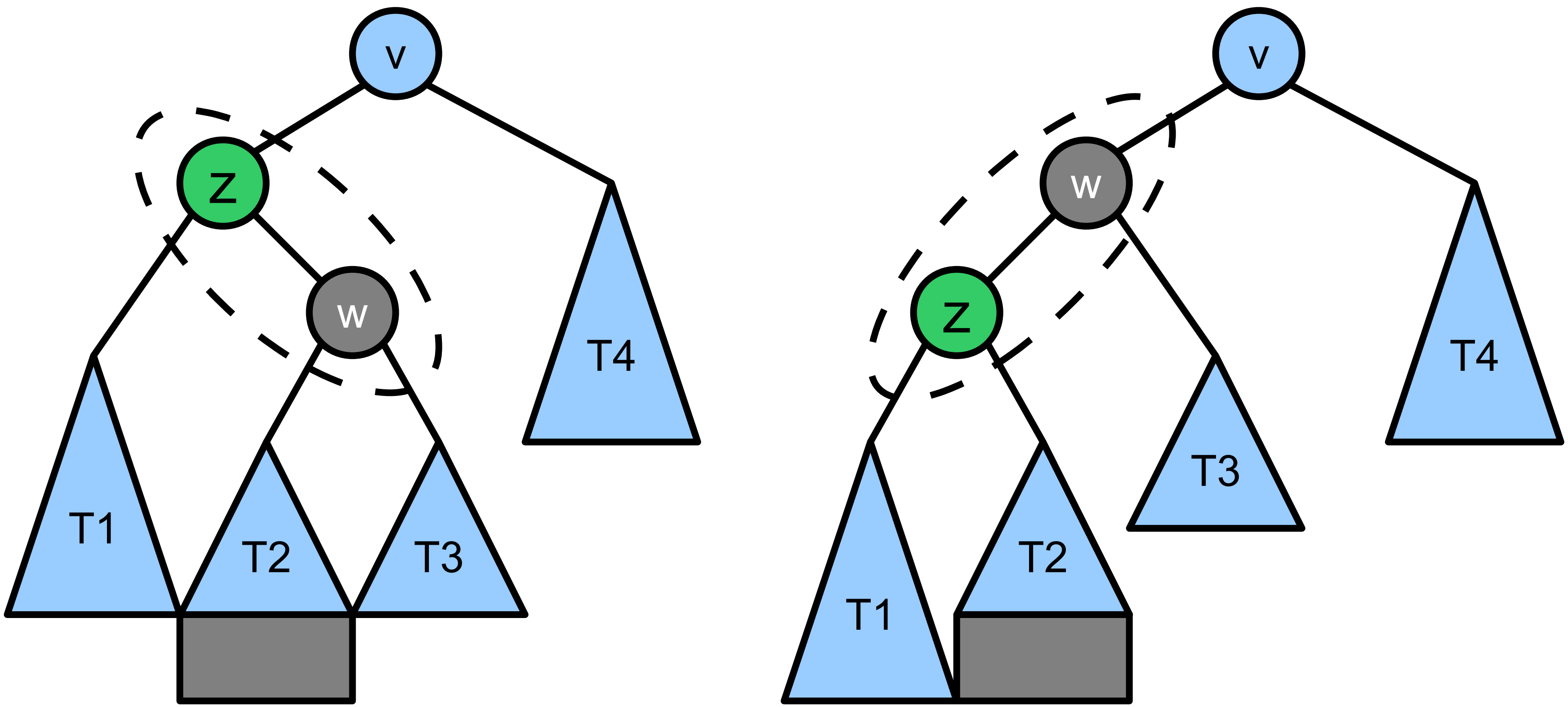
- A clockwise simple rotation of u on v
- Has cost O(1)



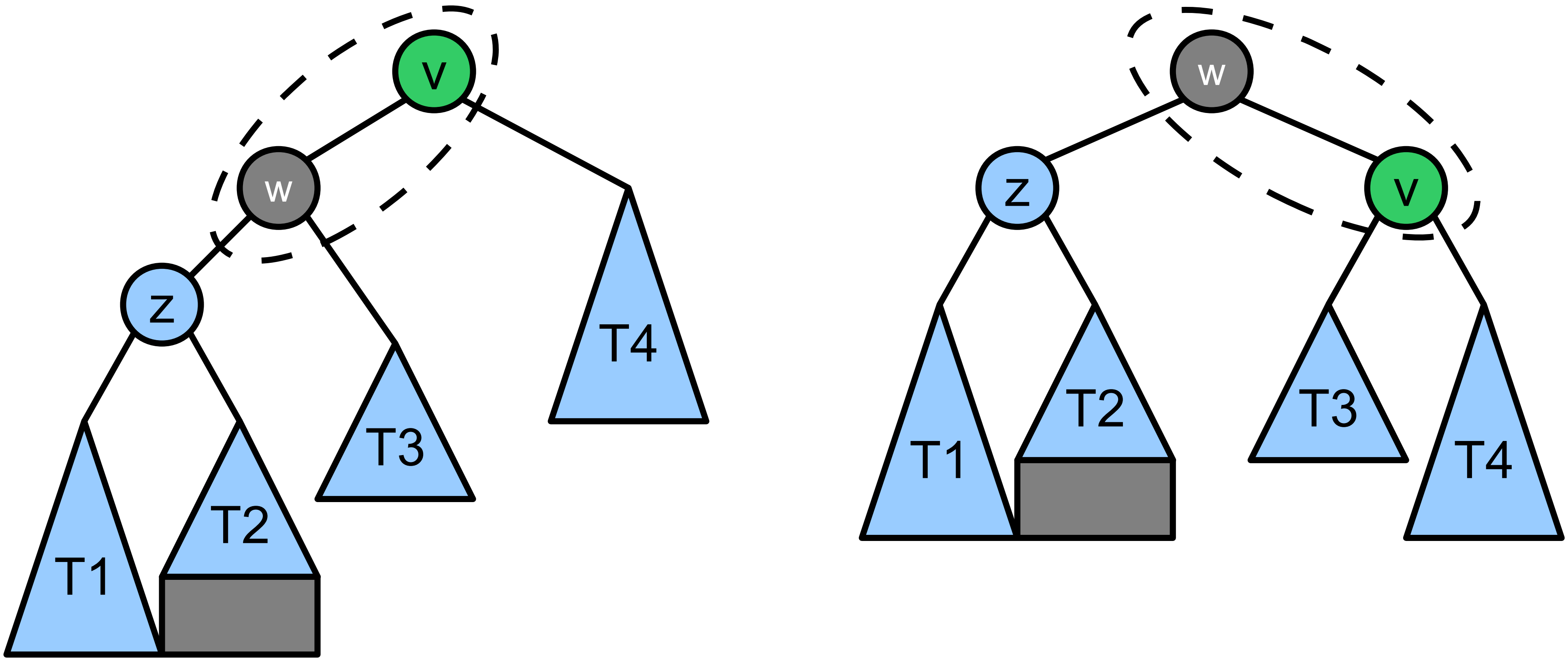
Rebalancing: rotation SD (does not work!)



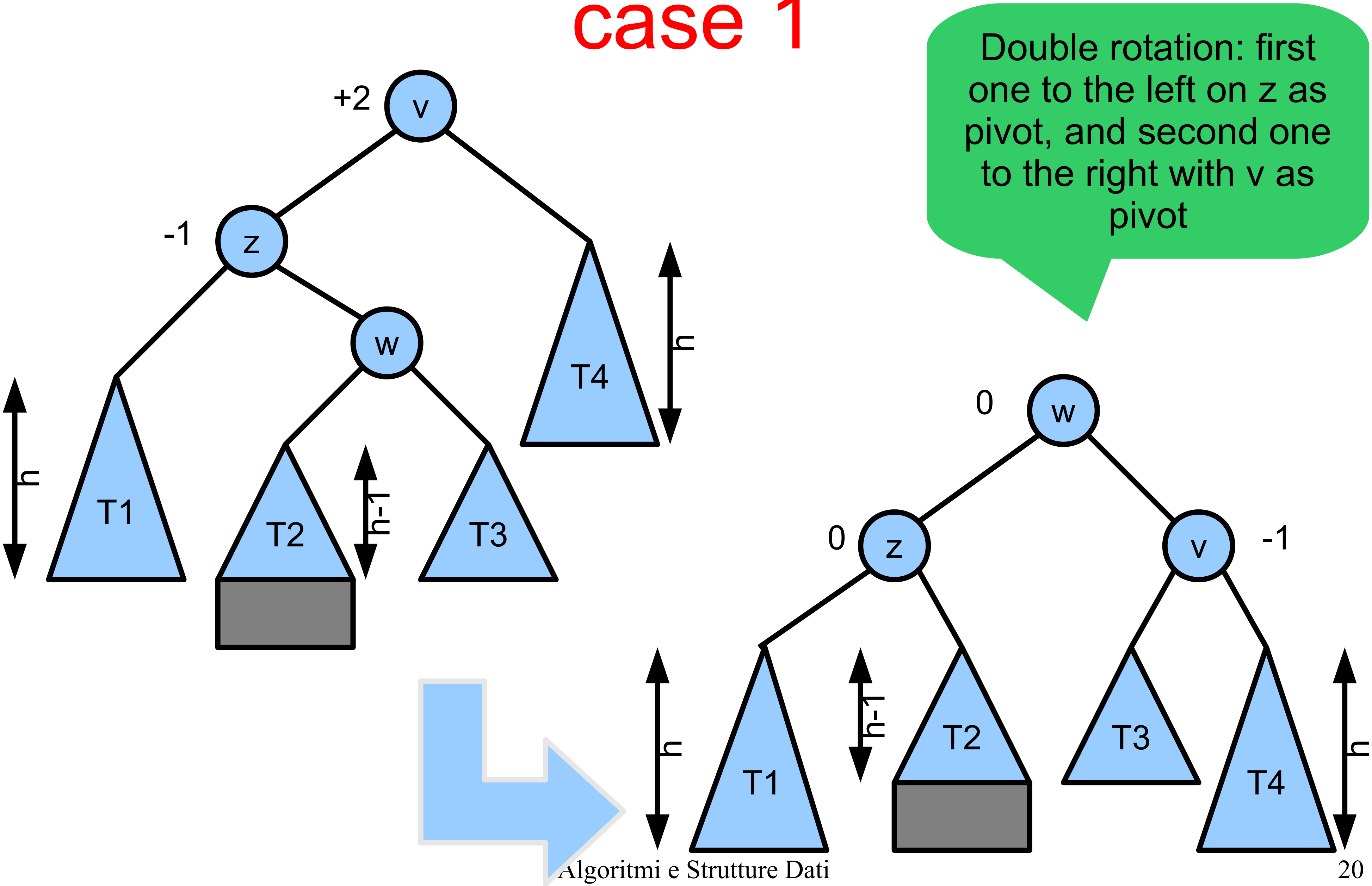
Rebalancing: rotation SD
first step



Rebalancing: rotation SD second step

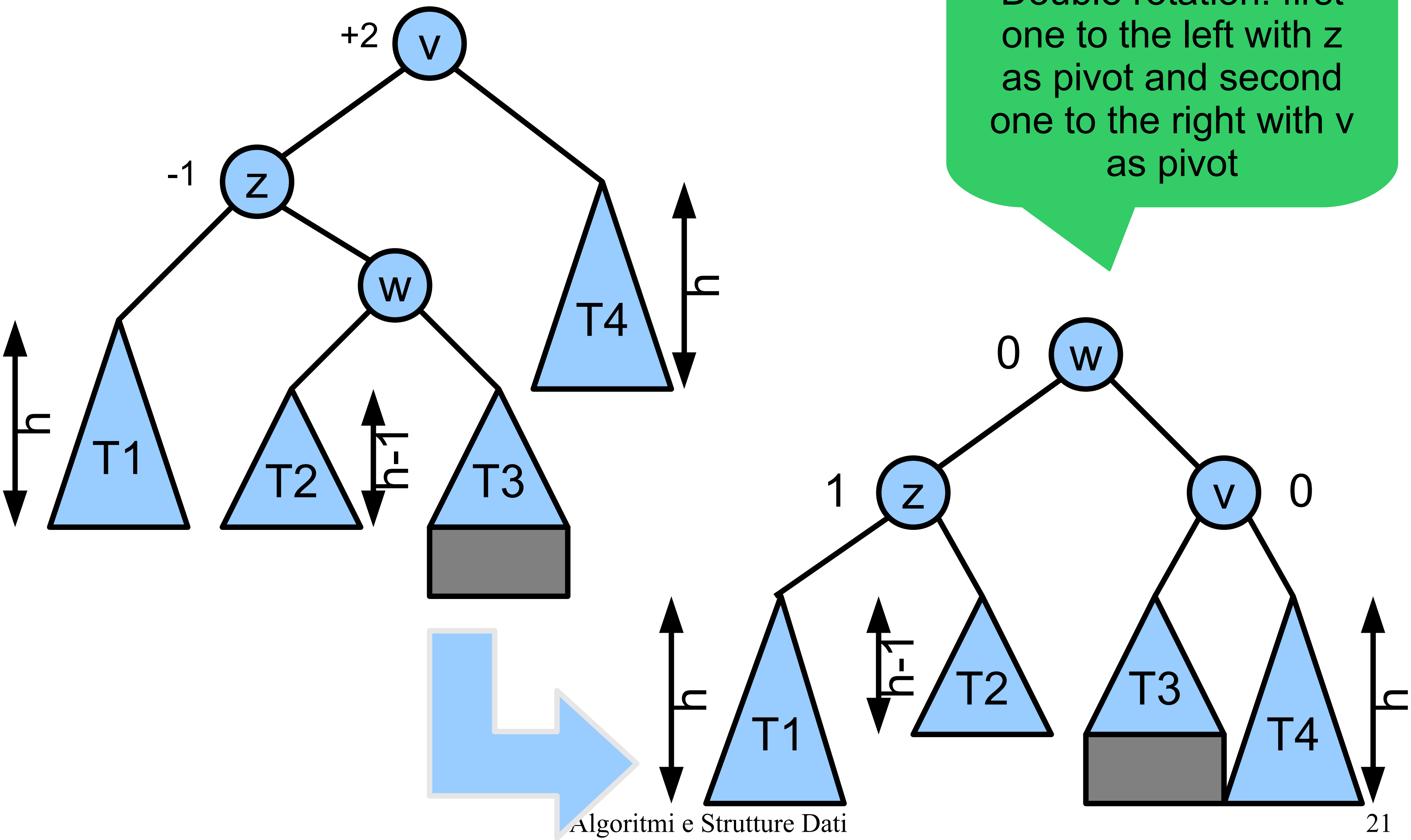


Rebalancing: rotation SD case 1



Rebalancing: rotation SD case 2

Double rotation: first one to the left with z as pivot and second one to the right with v as pivot



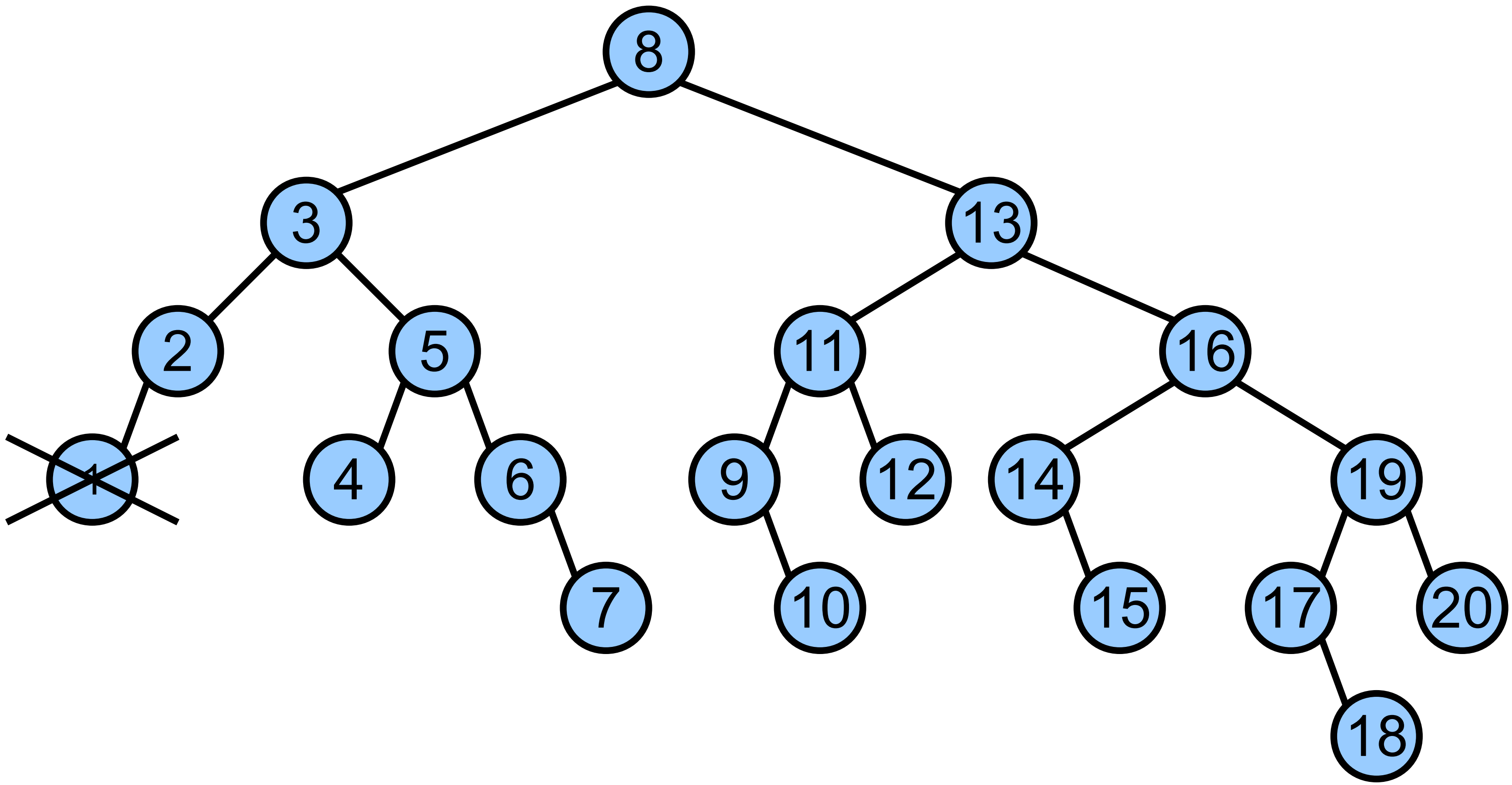
AVL tree: Insertion

- Insert a new value like in traditional BSTs
- Recalculate all the balancing factors changed:
  - At most, the recalculation is done for nodes on the path from the leaf inserted up to the root, hence cost is O(log n)
- If at least a node has balancing factor ±2 (critical node), we need to rebalance the tree by using the rotations
  - Note: in caso of insertion, there is only one critical node.
- Overall cost: O(log n)

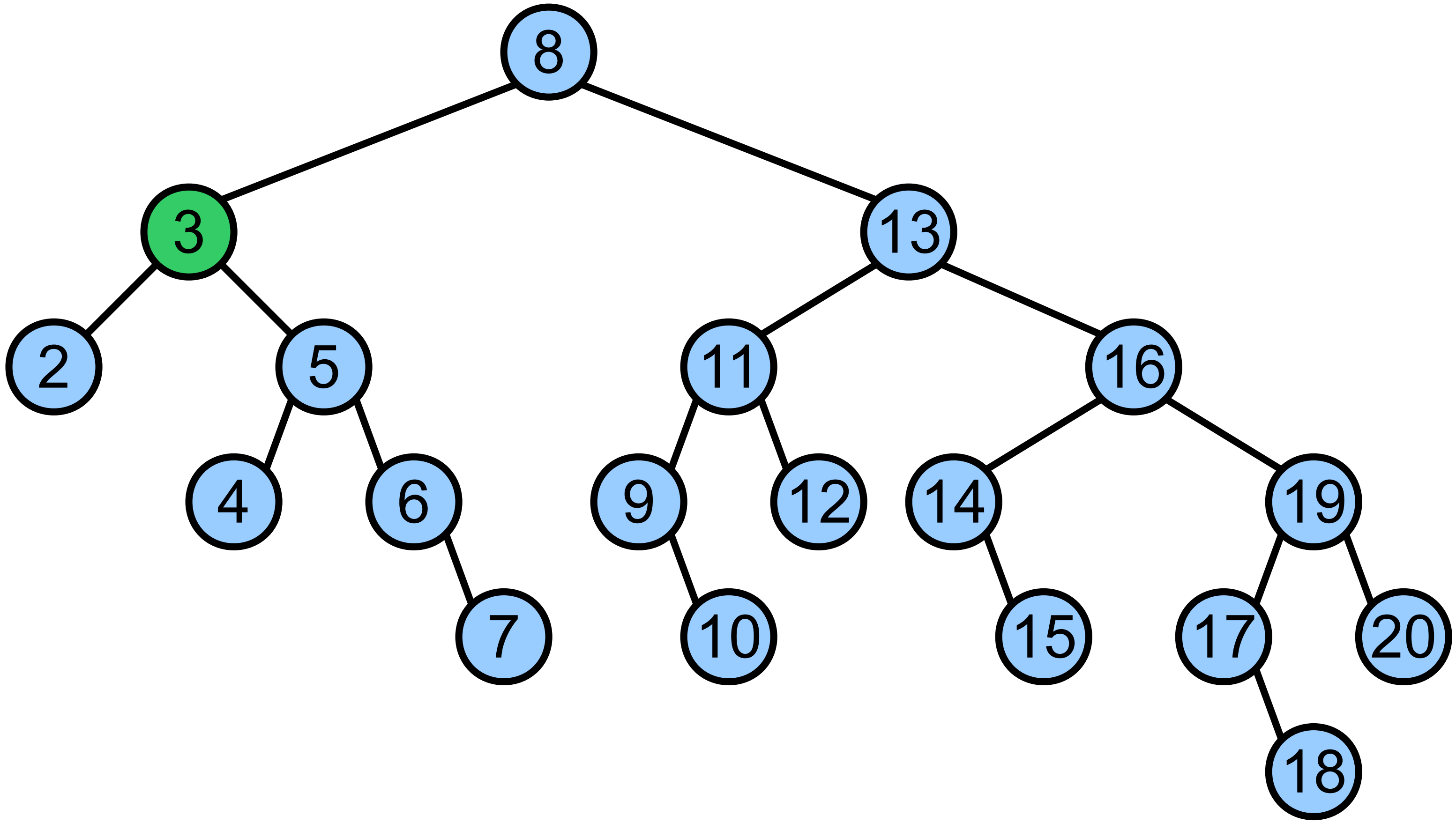
AVL tree: deletion

- Remove a node like in traditional BSTs
- Recalculate all the balancing factors changed:
  - At most, the recalculation is done for nodes on the path from the leaf deleted up to the root, hence cost is O(log n)
- For each node with balancing factor ±2 (critical node), we need to rebalance the tree by using the rotations
  - Note: in case of deletion, more than one nodes could result with a balancing index ±2
- Overall cose: O(log n)

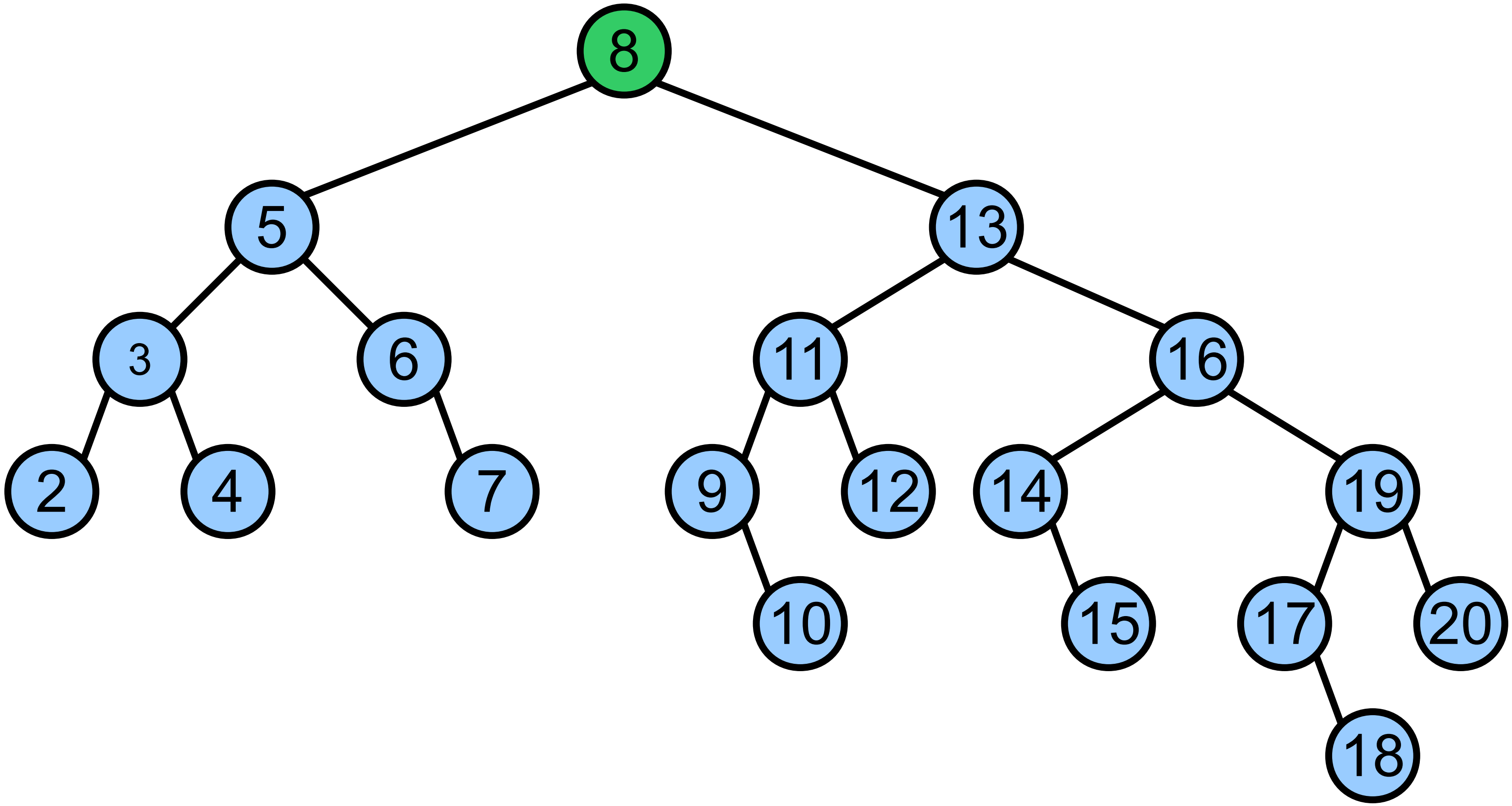
Example: deletion with cascade rotations



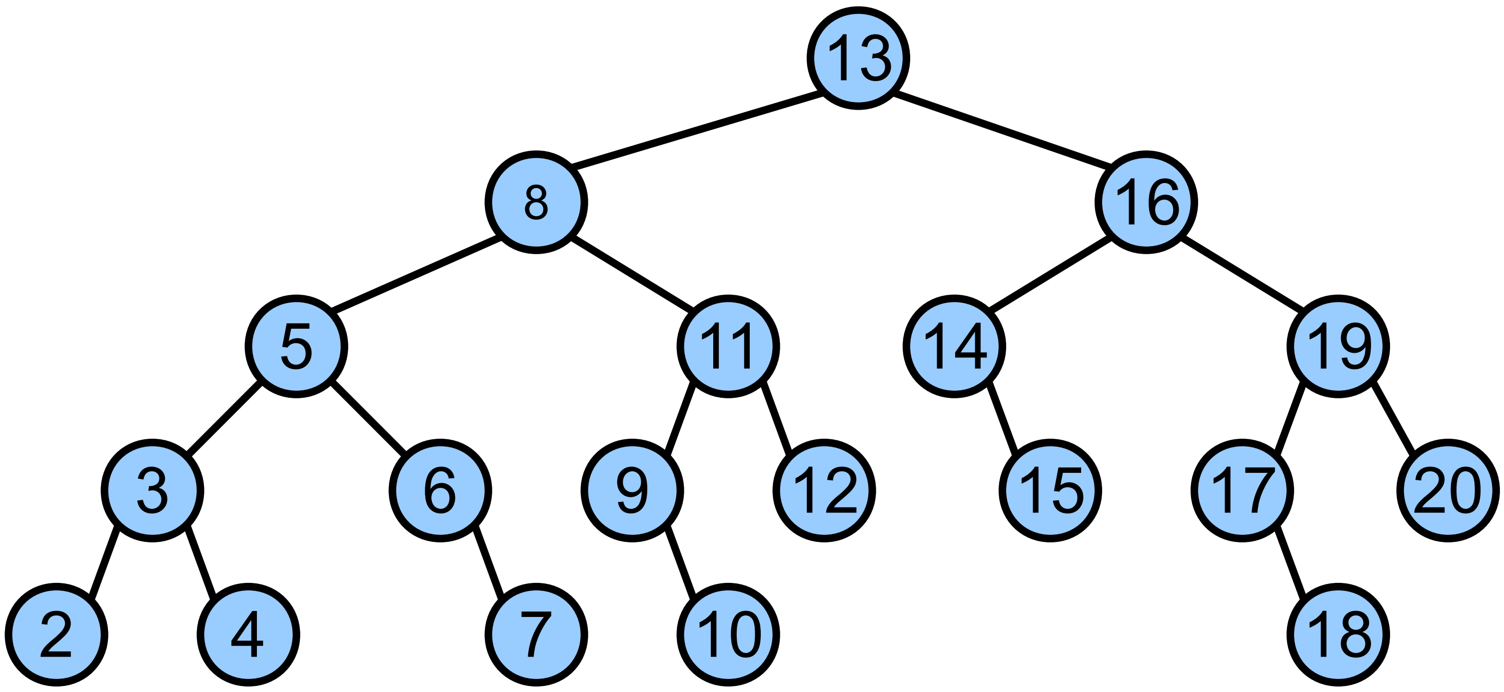
Apply left rotation on 3



Apply left rotation on 8



New balanced AVL



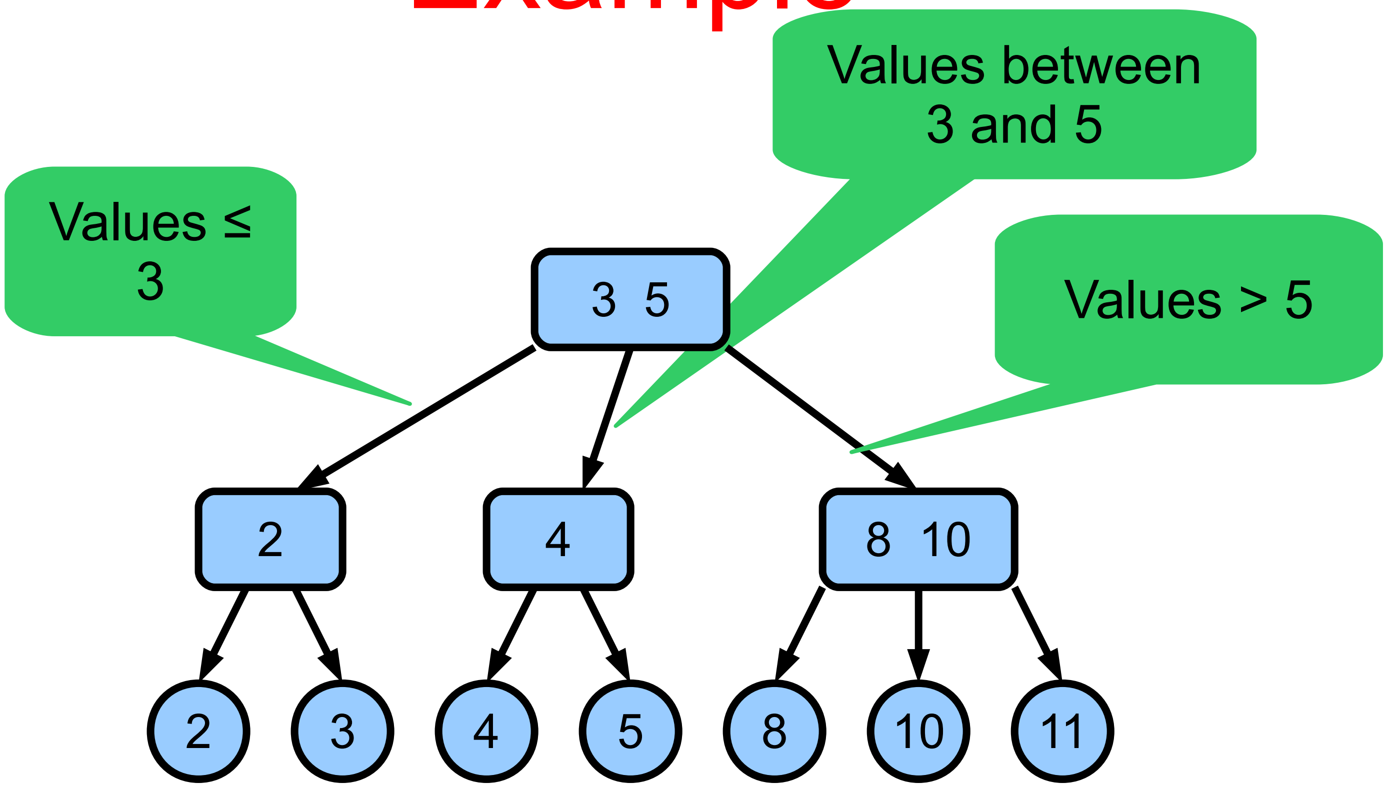
AVL trees: summary

- search(Key k)
  - O(log n) in the worst case
- insert(Key k, Item t)
  - O(log n) in the worst case
- delete(Key k)
  - O(log n) in the worst case

2-3 trees

- Definition: a 2-3 tree is a tree where:
  - Every internal node has 2 or 3 children and all the paths root/leaf have the same length
  - The leaves contain the keys and associated values, and they are sorted from left to rigth in ascending order of key
  - Every internal node v mantains two information:
    - S[v] is the max key in the subtree whose root is the left child
    - M[v] is the max key in the subtree whose root is the central child (if v has only 2 children, it will contain S[v] only)

Example



Heigth of 2-3 trees

- let T be a 2-3 tree with n nodes, f leaves and heigth h. Then the following inequalities hold:

$$2^{h+1}-1 \leq n \leq (3^{h+1}-1)/2$$

$$2^h \leq f \leq 3^h$$

- In particular, we can conclude that the heigth of a 2-3 tree is Θ(log n)

Heigth of 2-3 trees proof

- By induction on h: if h=0, the tree has only one node (leaf) and the relations are satisfied.
- if h>0, let's consider the 2-3 tree T' without the lower level (leaves). Let n' and f' be the number of nodes and leaves in T'
  - Inductive assumption $2^{h-1} \leq f' \leq 3^{h-1}$
  - Every leaf in T' can have 2 or 3 children, so we obtain

$$2 \times 2^{h-1} \leq f \leq 3 \times 3^{h-1}$$
$$2^h \leq f \leq 3^h$$

Heigth of 2-3 trees proof

- for the number of nodes, the inductive assumption is

$$2^h - 1 \leq n' \leq (3^h - 1)/2$$

- We observe that n = n' + f, hence

$$2^h - 1 \leq n' \leq (3^h - 1)/2$$

$$2^h \leq f \leq 3^h$$

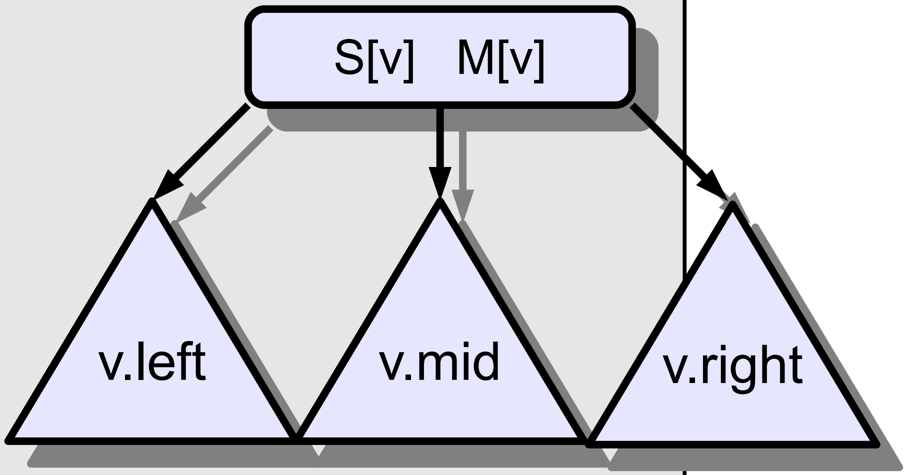
and we obtain

$$2^h + 2^h - 1 \leq n \leq (3^h - 1)/2 + 3^h$$

$$2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$$

search

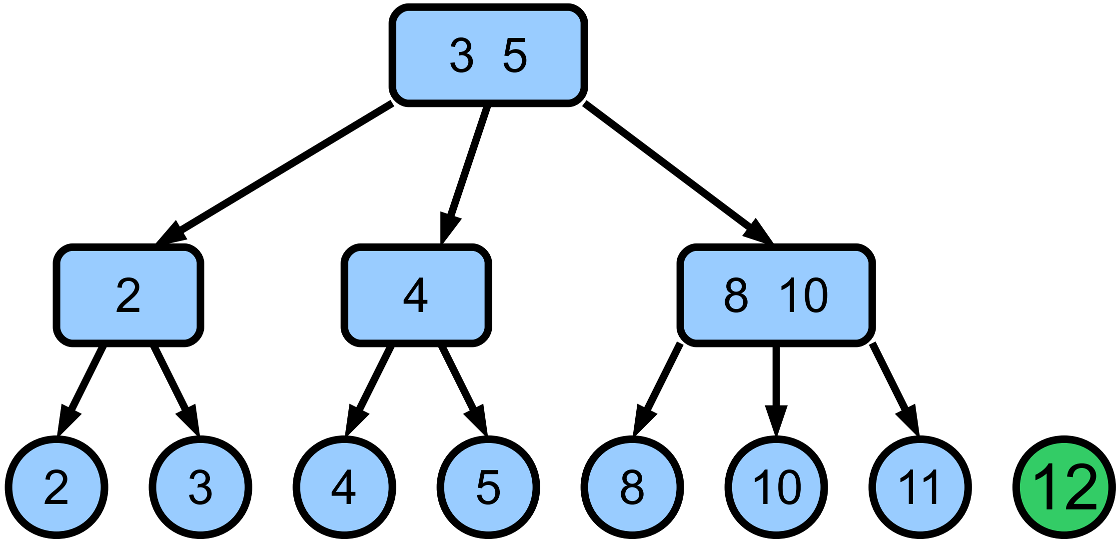
```
Algorithm 23search( T, k )
    if ( T == null ) then
        return null;
    endif
    node v := T.root;
    if ( v is a leaf ) then
        if ( key of v == k ) then
            return v;
        else
            return null;
        endif
    else // v is not a leaf
        if ( k ≤ S[v] ) then
            return 23search( v.left, k );
        elseif ( v.right != null && k > M[v] ) then
            return 23search( v.right, k );
        else
            return 23search( v.mid, k );
        endif
    endif
```



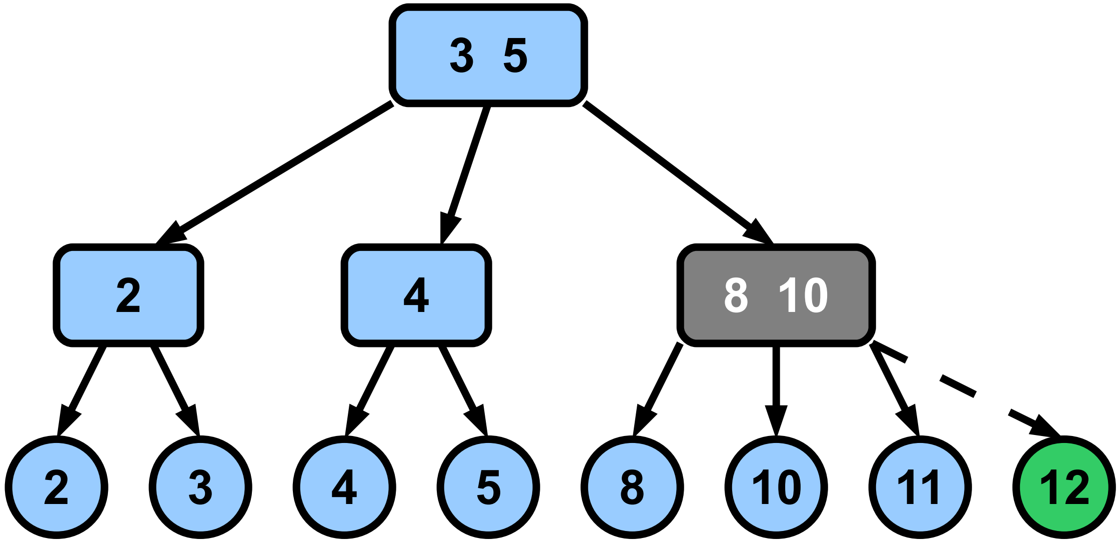
Insertion

- Create a leaf v with key k
- By using the search operation, we find a node u in the penultimate level, who will become the father of v
- We add v as a child of u, if possible
  - if u already has 3 children, we need to make an operation of splitting (split), which could also propagate back up to the root.

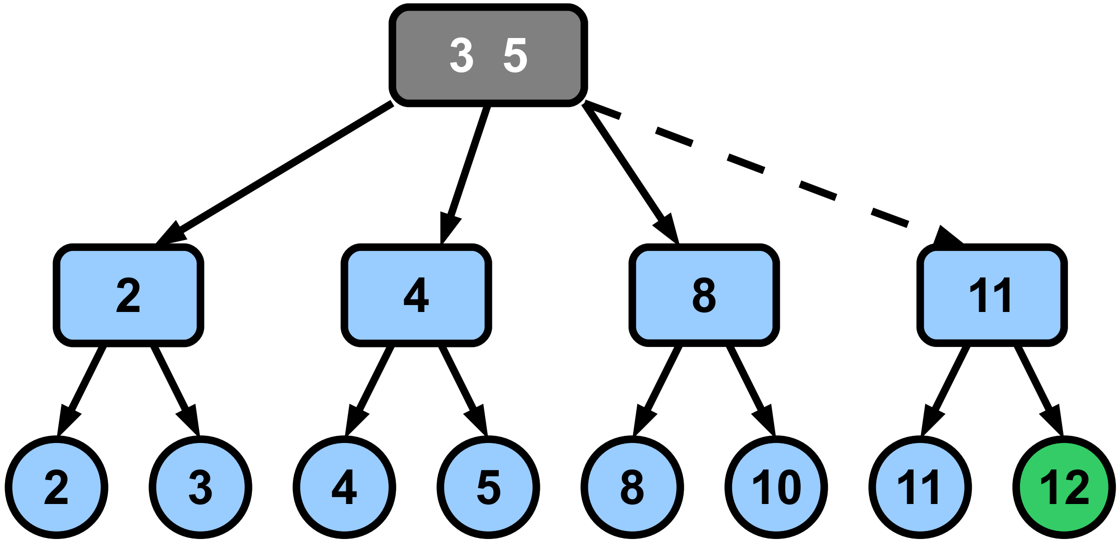
Example



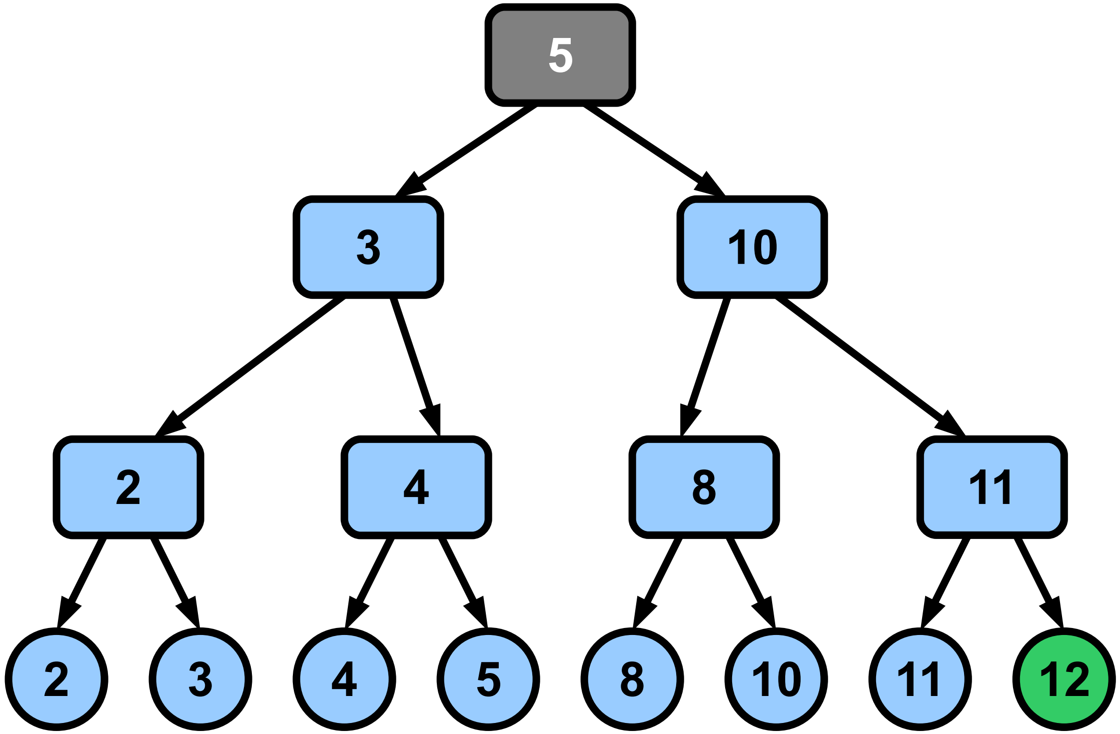
Example



Example



Example

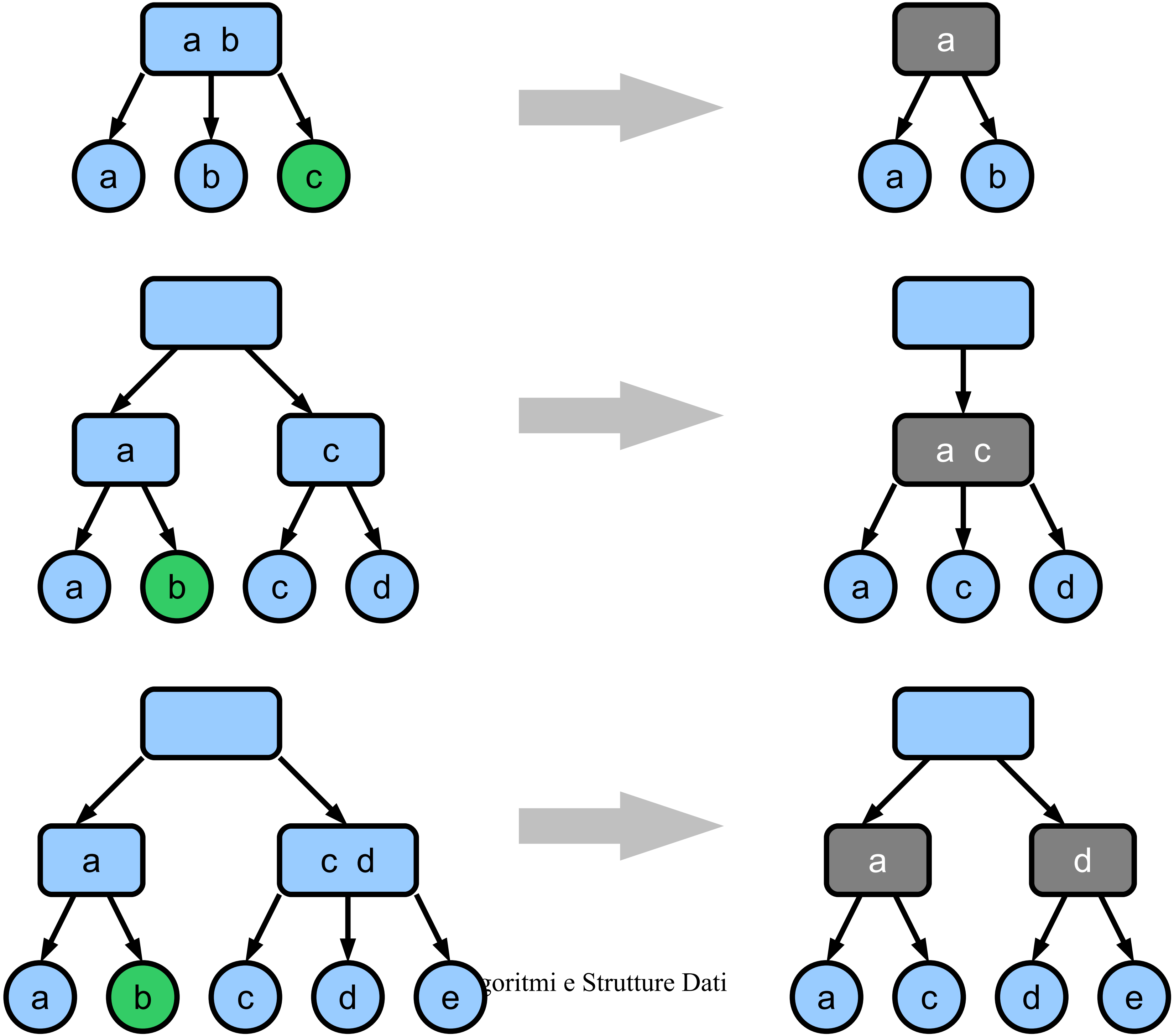


Insertion: cost

- O(log n) to identify the father of the new node
- O(log n) split in the worst case, each one with cost O(1)
- Overall, the cost of the insertion is O(log n)

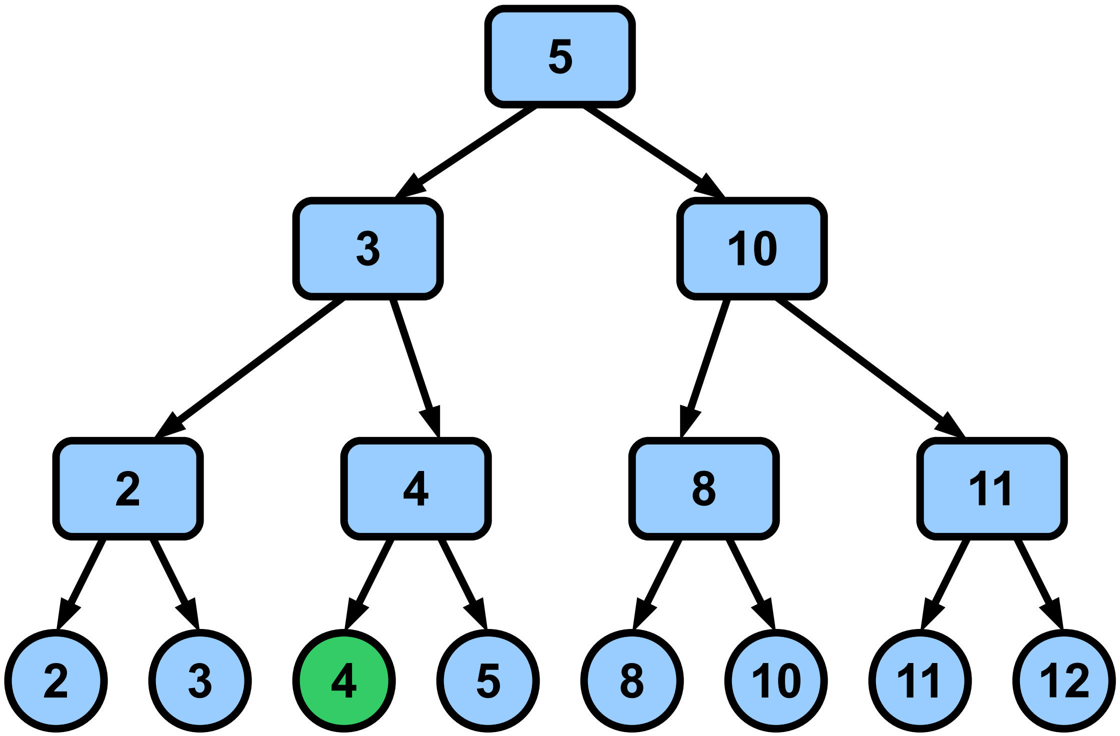
Deletion

- We find a leaf v with the key to delete
- We remove v, detaching the node from the father u
  - If u had 2 children, it remains with only 1 child (violating the property of 2-3 trees). So we need to merge the node U with a neighbor.
  - The merging operation could propagate up to the root.

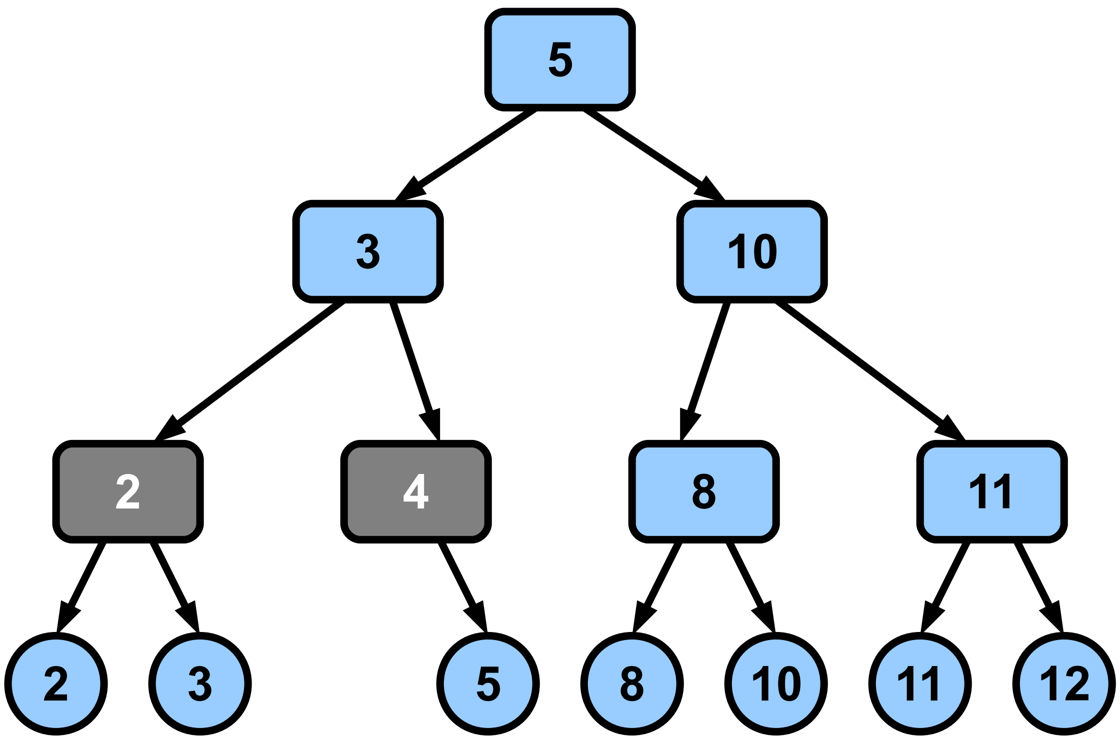


a b
a
b
c
a
a
b
a
c
a
b
c
d
a c
a
c
d
a
c d
a
b
c
d
e
a
d
a
c
d
e

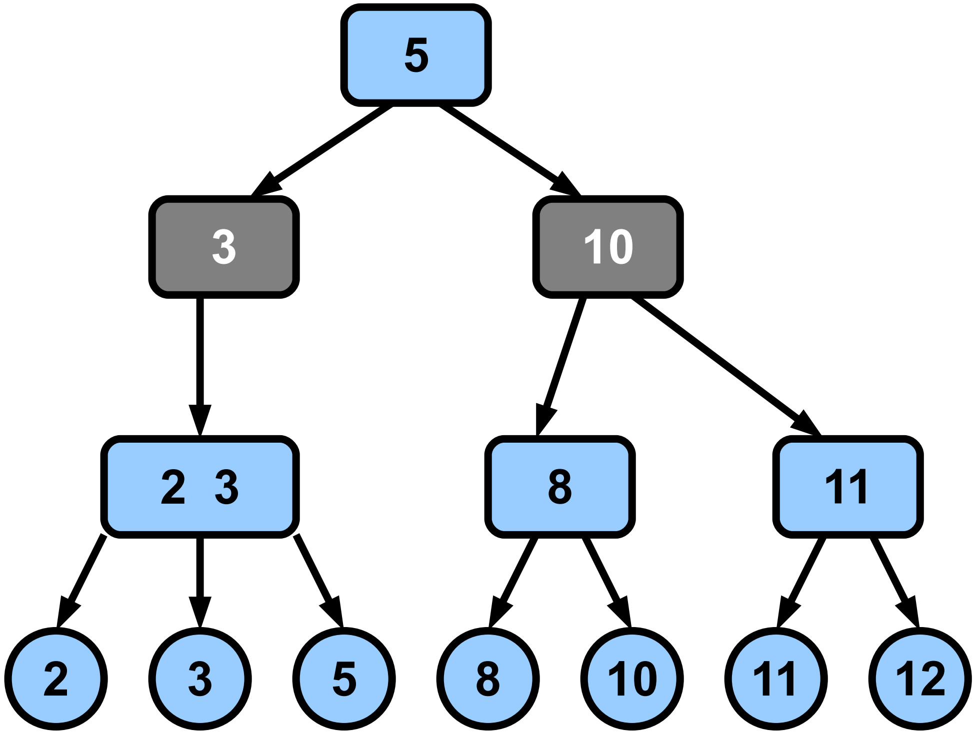
Example



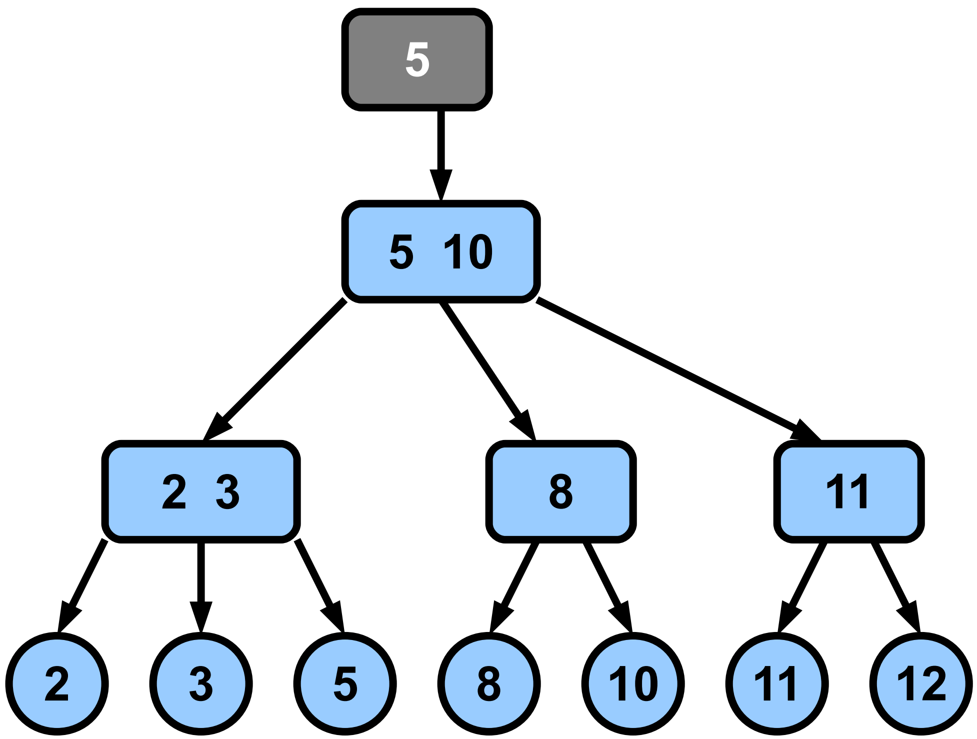
Example



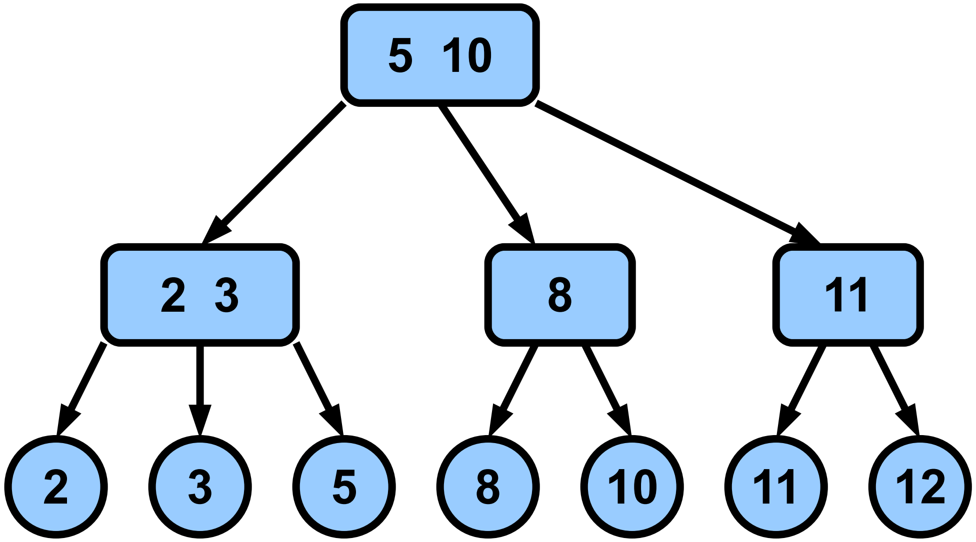
Example



Example



Example

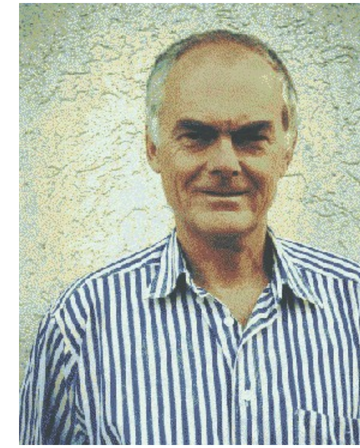


2-3 trees: summary

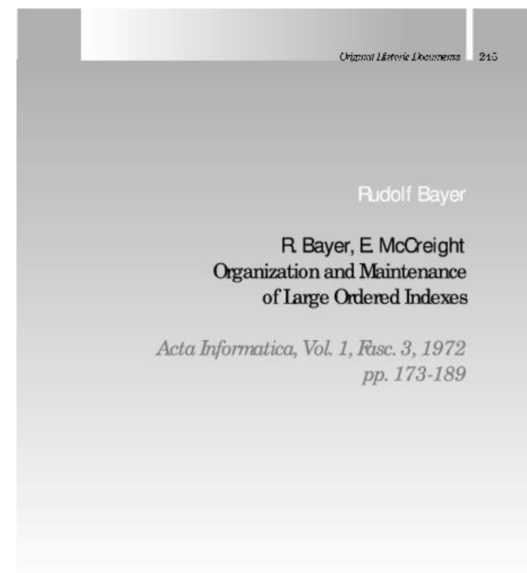
- search(Key k)
  - O(log n) in the worst case
- insert(Key k, Item t)
  - O(log n) in the worst case
- delete(Key k)
  - O(log n) in the worst case

B-Tree

Prof. Rudolf Bayer



- Data structure used in applications needing to manage sets of ordered keys
- a variation (B+-Tree) is used in:
  - **Filesystem**: btrfs, NTFS, ReiserFS, NSS, XFS, JFS to index metadata
  - **Relational Database**: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASI, PostgreSQL, Firebird, MySQL to index tables


Original Historic Documents 215

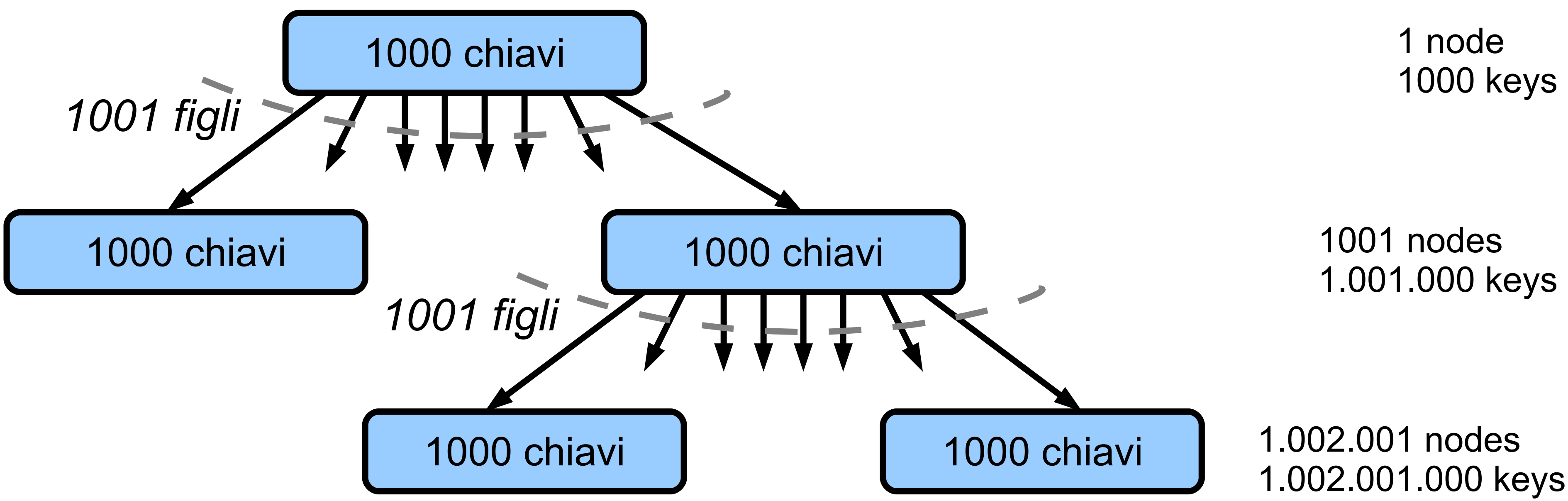
Rudolf Bayer

R. Bayer, E. McCreight
Organization and Maintenance
of Large Ordered Indexes

Acta Informatica, Vol. 1, Fasc. 3, 1972
pp. 173-189

B-Tree

- Since every node can have a high number of children, B-trees can efficiently index big amounts of data on external memory (discs), reducing I/O operations.



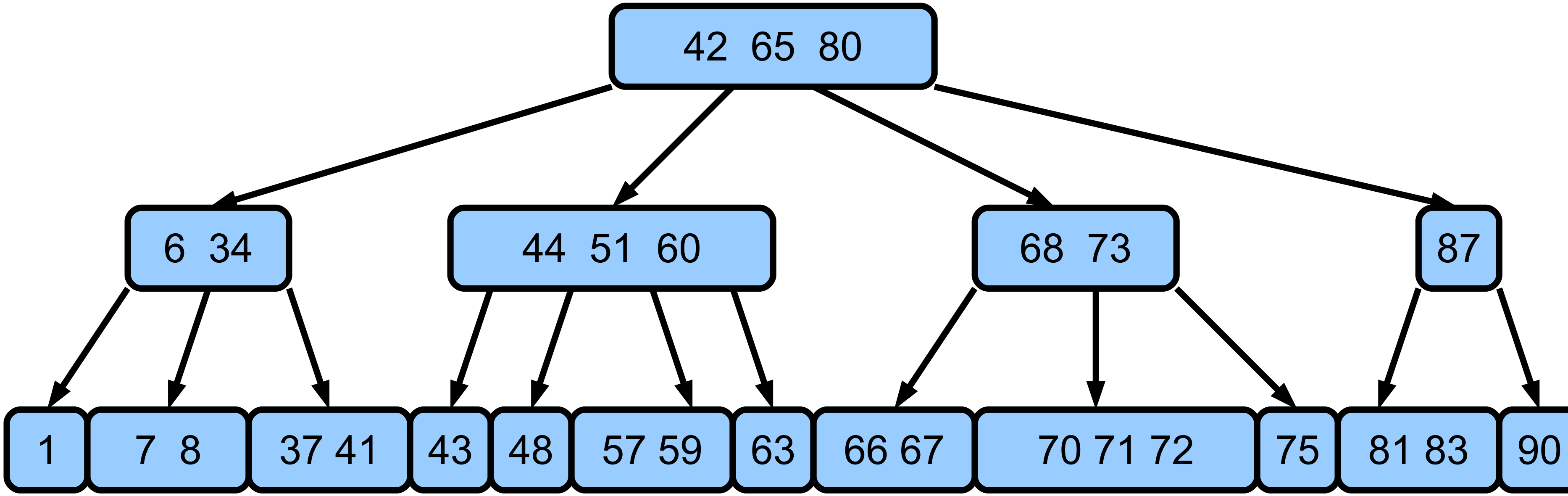
B-Tree

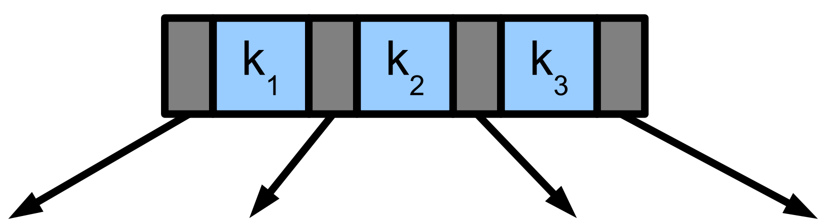
- a B-Tree with grade t (≥2) has the following properties:
  - All the leaves have the same depth
  - Every node v different than the root maintains k(v) ordered keys:

    $$key_1(v) \leq key_2(v) \leq \ldots \leq key_{k(v)}(v)$$

    such that $t-1 \leq k(v) \leq 2t-1$
  - The root has at least 1 and at most 2t-1 ordered keys
  - Every internal node v has k(v)+1 children
  - The keys key(v) split the intervals of keys stored in every subtree. If c_i is a key of the i-th subtree of a node v, then $c_1 \leq key_1(v) \leq c_2 \leq key_2(v) \leq \ldots \leq c_{k(v)} \leq key_{k(v)}(v) \leq c_{k(v)+1}$

Example: B-Tree with t=2





$k \leq k_1$ $k_1 \leq k \leq k_2$ $k_2 \leq k \leq k_3$ $k \geq k_3$

Heigth of a B-Tree

- a B-Tree with n keys has heigth

$$h \leq \log_t \frac{n+1}{2}$$

- proof
  - Given all B-trees of grade t, the higest one is the one with the lower number of children per node (that is, with t children)
  - 1 node has depth zero (the root)
  - 2 nodes have depth 1
  - 2t nodes have depth 2
  - $2t^2$ nodes have depth 3
  - …
  - $2t^{i-1}$ nodes have depth i

Heigth of a B-Tree

- Total number of nodes in a B-Tree with heigth h

$$1+\sum_{i=1}^{h} 2t^{i-1}$$

- Since every node but the root contains exactly t-1 keys, the number of keys n satisfies:

$$n \geq 1+(t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$=1+2(t-1)\frac{t^h-1}{t-1}=2t^h-1$$

$$\sum_{i=1}^{h} t^{i-1}=\sum_{i=0}^{h-1} t^i=\frac{t^h-1}{t-1}$$

Heigth of a B-tree

- given $n \geq 2t^h - 1$

  we get $t^h \leq \frac{n+1}{2}$

  and applying the log base t we get:

$$h \leq \log_t \frac{n+1}{2}$$

Search operation on B-tree

- Is a generalization of the search on BST
  - In each step we search the key in the current node
  - If the key is found we stop
  - If the key is not found we search it in the subtree who may contain it

```
algorithm search(root v of a B-Tree, key x) → elem
    i ← 1
    while (i≤k(v) && x>key_i(v)) do
        i ← i+1;
    endwhile
    if (i≤k(v) && x==key_i(v)) then
        return elem_i(v);
    else
        if (v is a leaf) then
            return null
        else
            return search(i-th child of v, x);
        endif
    endif
```

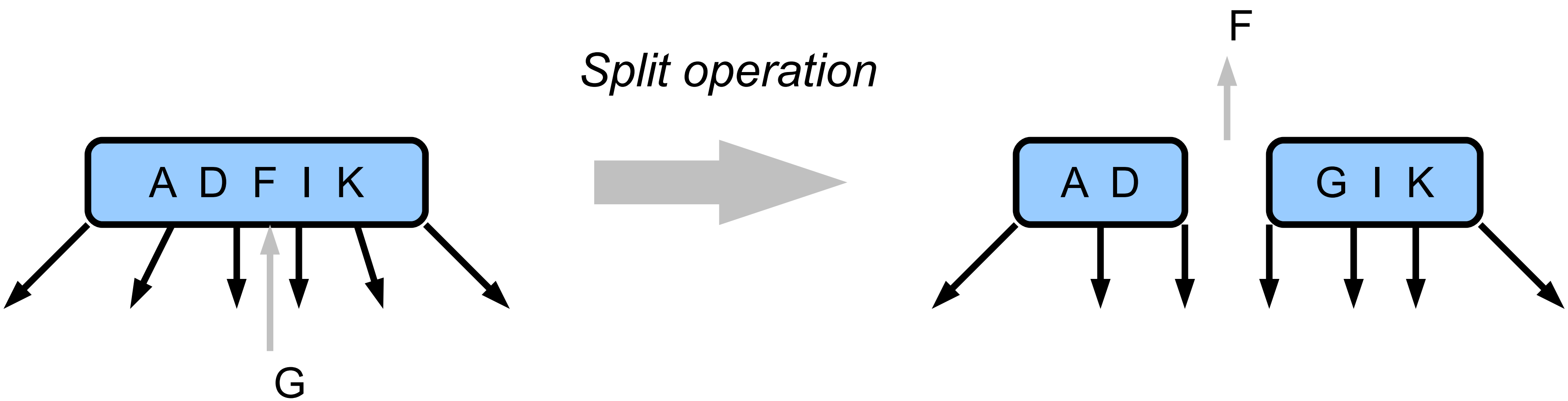
Search opepration on B-tree

- Computational cost
  - Number of visited nodes is $O(\log_t n)$
  - Every visit costs $O(t)$ doing a linear scan of the keys.
  - Total $O(t \log_t n)$
    - However, since the keys are sorted in each node, we can exploit a binary search in time O(log t) instead of O(t). In this case, the total cost becomes $O(\log t \log_t n) = O(\log n)$ (using the rule for changing the base of log)

Insert a key in a B-tree

- We search() the leaf f in which to insert key k
- If the leaf is not full (it has less than 2t-1 keys) we insert k in the correct position and we stop.
- If the leaf is full (has 2t-1 keys) then
  - Node f is split into two (split operation) and the t-th key is moved in the father of f
  - If the father of f already had 2t-1 keys (full) we need to split it in the same way, (this may continue up to the root).
  - In the worst case (when all the path from the leaf f to the root is made of full nodes) the consecutive splits will create a new root.

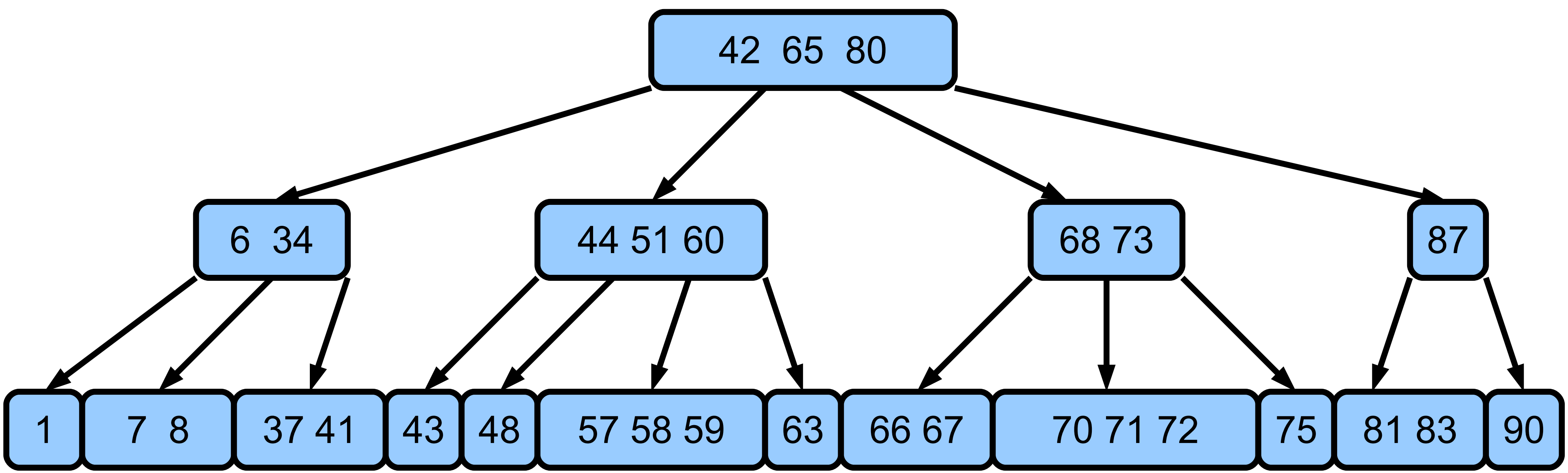
Insert a key in a B-tree



- Computational cost
  - Visited nodes are $O(\log_t n)$
  - Each visit costs $O(t)$ in the worst case (due to split operations)
  - Total $O(t \log_t n)$

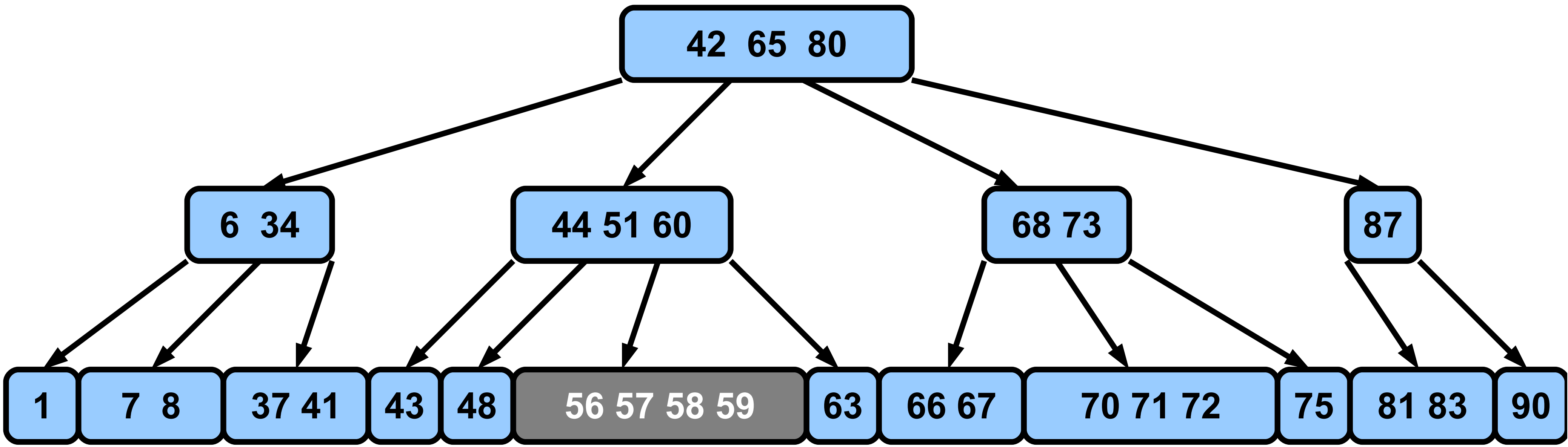
Insert a key in a B-tree

- Example (t=2)

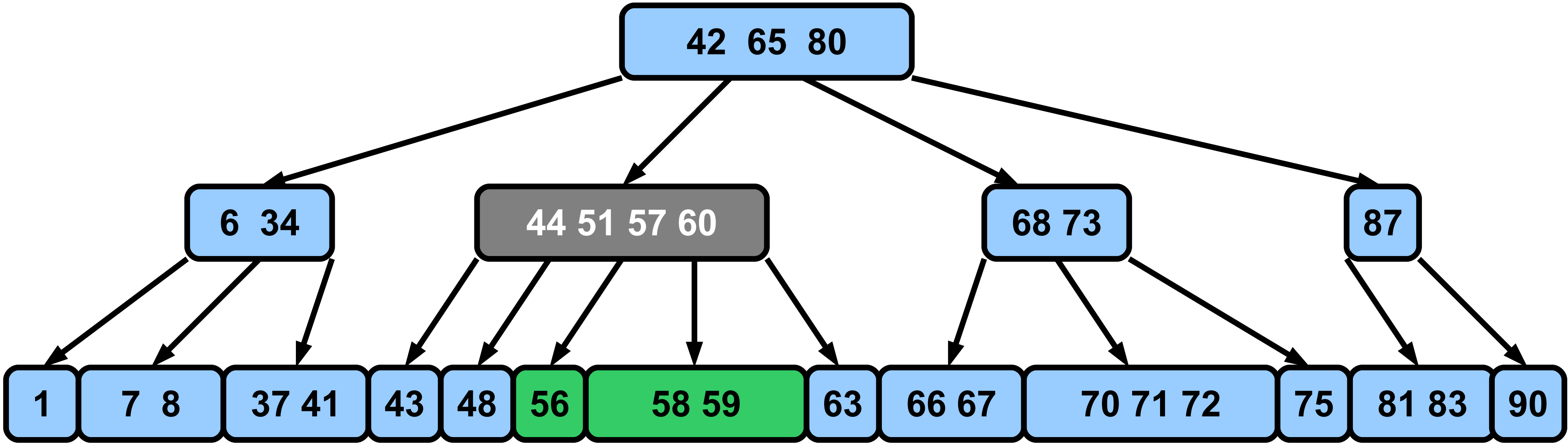


Insert a key in a B-tree

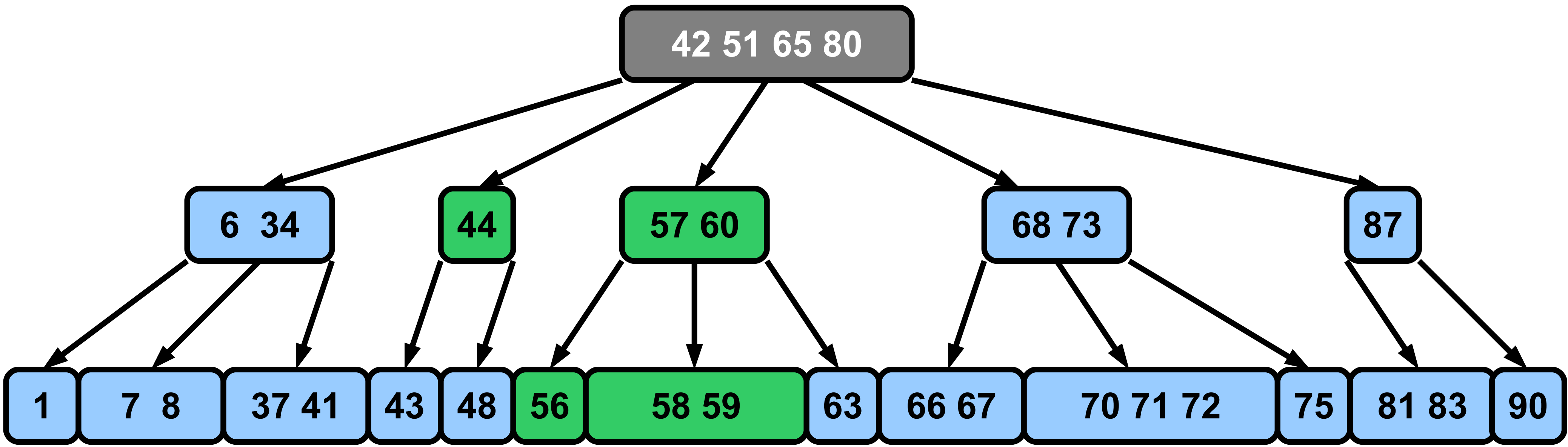
- Insert 56



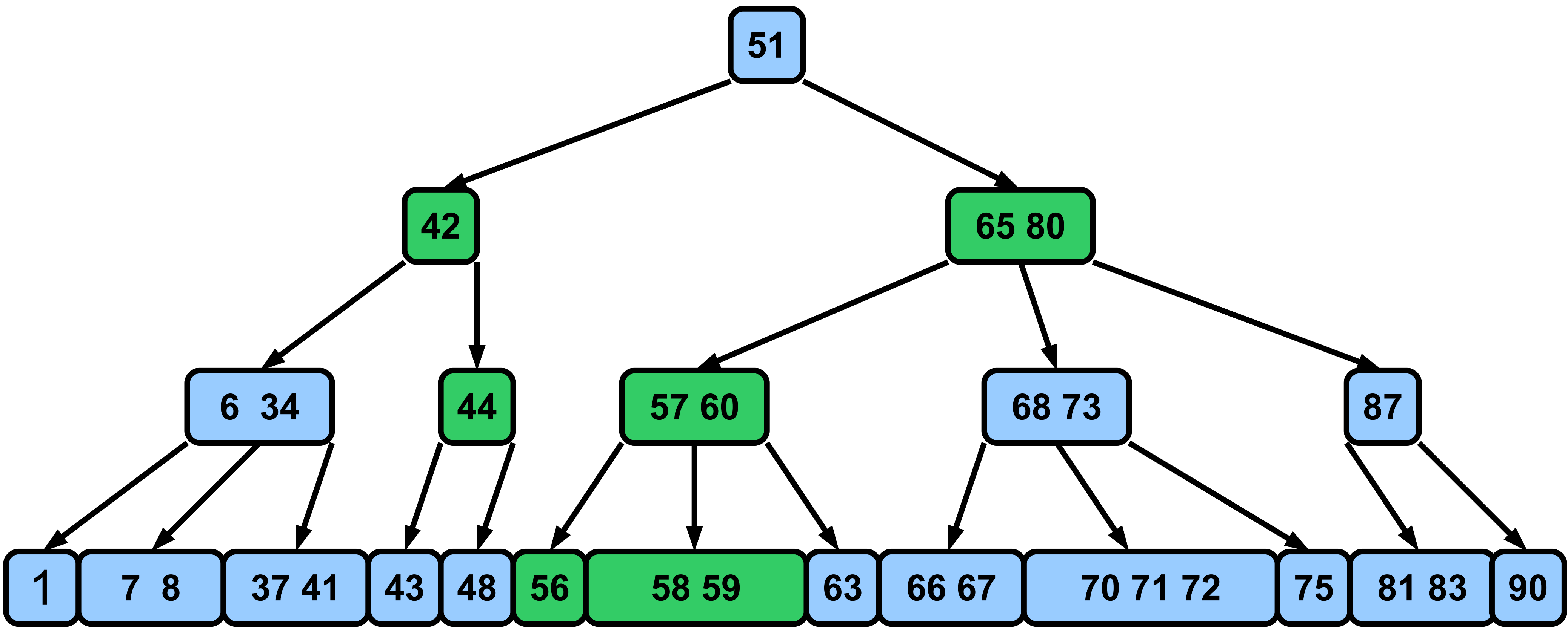
Insert a key in a B-tree



Insert a key in a B-tree



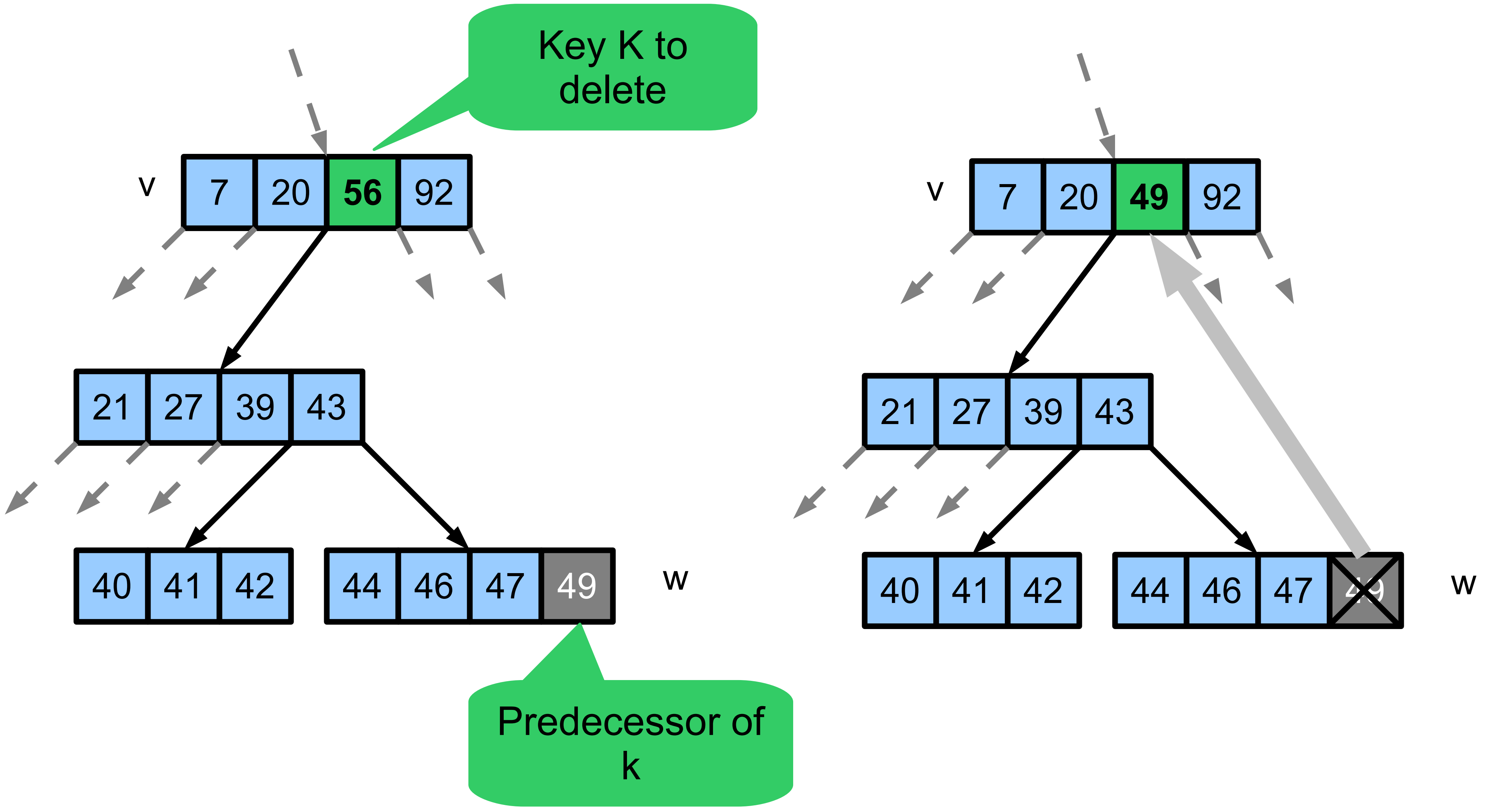
Insert a key in a B-tree



Delete a key from a B-tree

- If the key k to delete is in a node v which is not a leaf
  - We find the node containing the predecessor value of k
  - We move the max key in w in the place of the deleted key k
  - We exploit the next case by removing the max key in w
- If the key k to delete is in a leaf v
  - If the leaf has more than t-1 keys, just remove k and stop
  - If the leaf contains k-1 keys, by removing k we go below the minimum threshold. So we have to cases based on adjacent brothers:
    - If at least uno of the brothers has >t-1 keys we redistribute the keys
    - If none of the adjacent brothers has >t-1 keys we make a *fusion* operation.

B-Tree operations: deletion from internal node

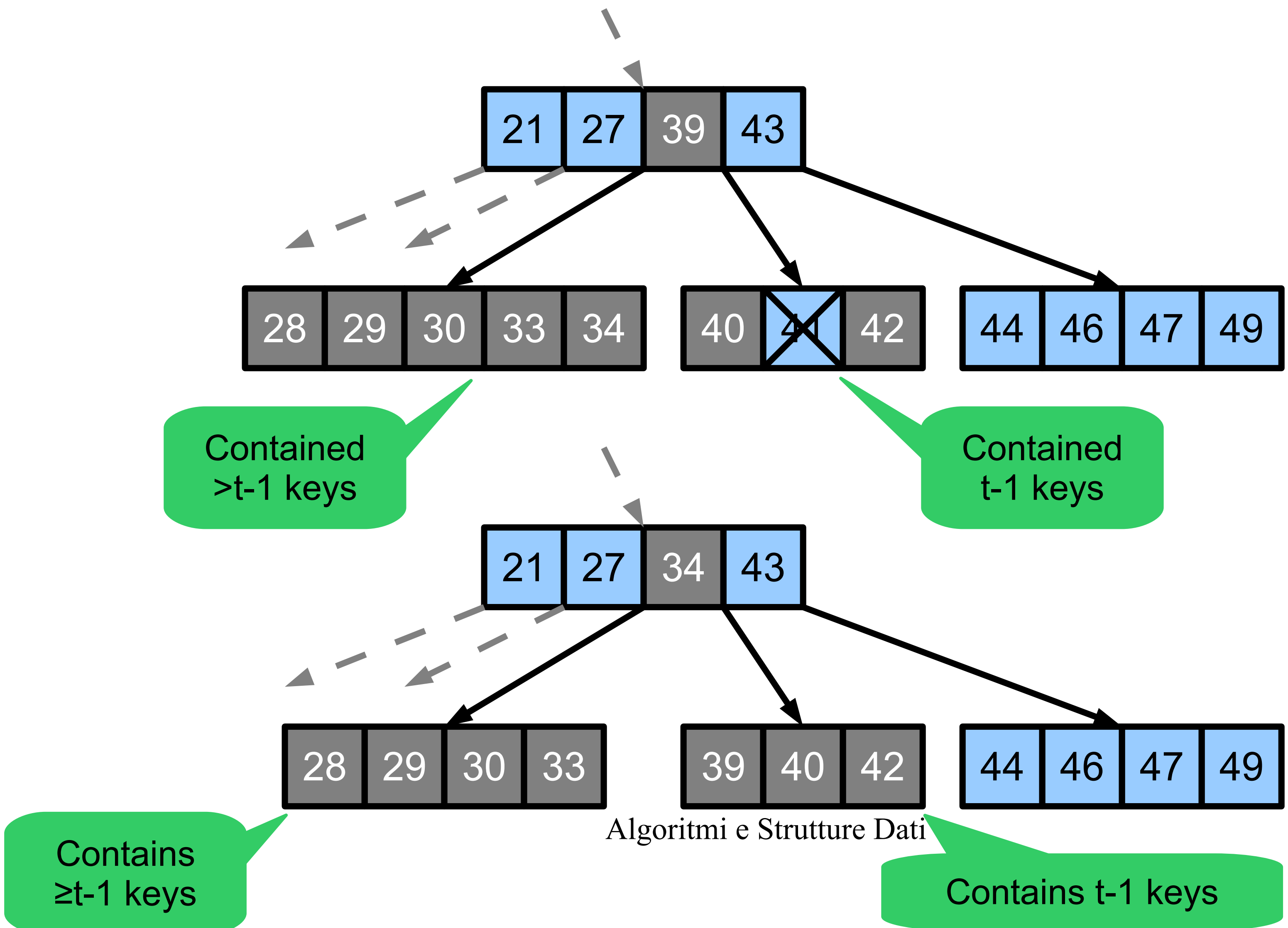


B-tree operations deletion from a leaf

- First case: leaf contains > t-1 keys
  - We remove the key from the leaf (now leaf contains ≥t-1 keys)
- Second case: the leaf contains <u>exactly</u> t-1 keys. We have two possibilities:
  - Redistribute keys with one adjacent brother
  - Merge the leaf with an adjacent brother

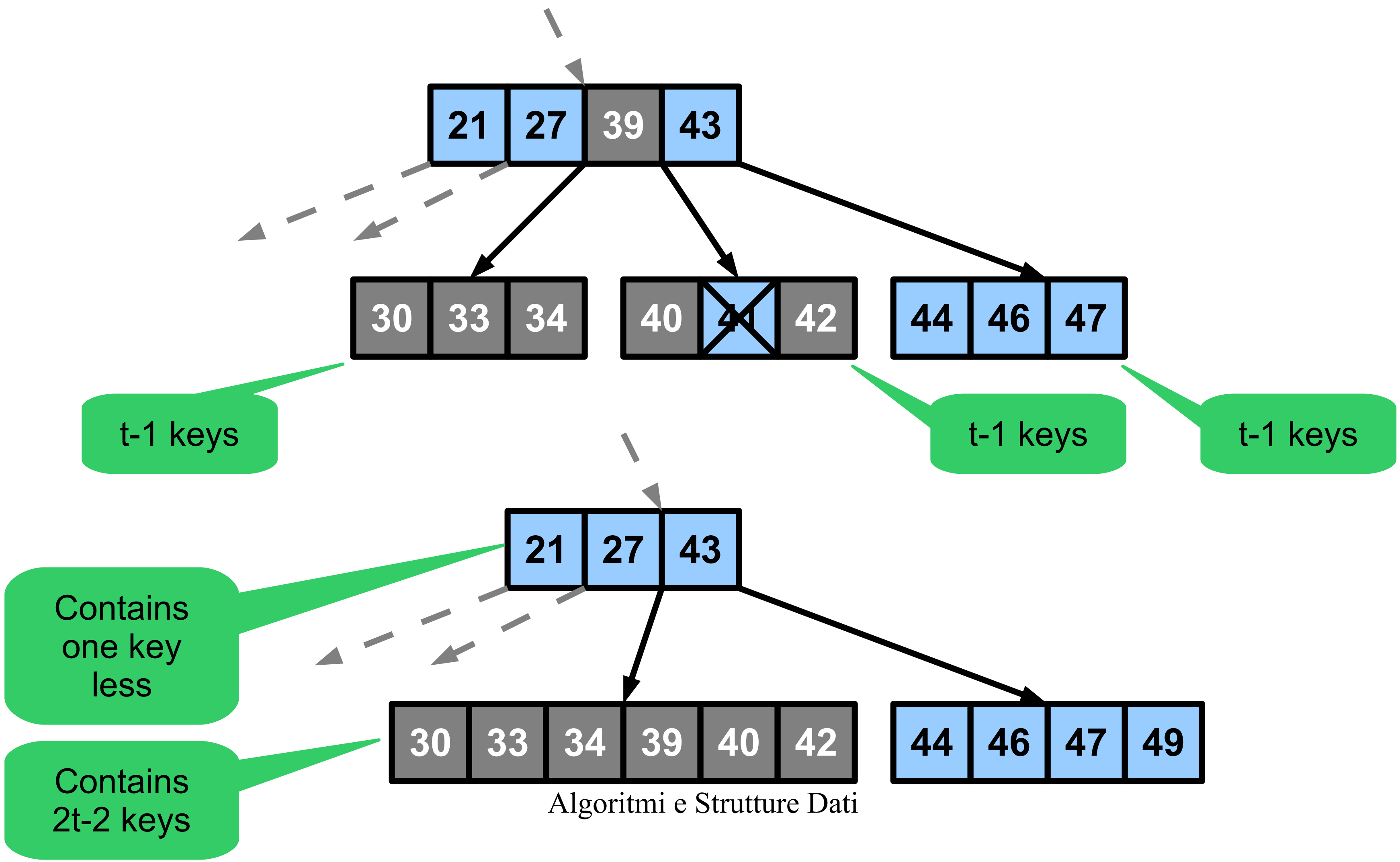
B-tree operations
delete from almost empyt leaf—case 1

- Given a B-tree fragment with t=4



B-tree operations delete from almost empyt leaf—case 2

- Given a B-tree fragment with t=4 (fusion)



summary

	search	insert	delete
Sorted array	O(log n)	O(n)	O(n)
Unsorted list	O(n)	O(1)	O(n)
BST	O(h)	O(h)	O(h)
AVL tree	O(log n)	O(log n)	O(log n)
2-3 tree	O(log n)	O(log n)	O(log n)
B-Tree	$O(\log t \log_t n) = O(\log n)$	$O(t \log_t n)$	$O(t \log_t n)$

Note all the costs refer to worst cases.