Data Structures and Algorithms – W08

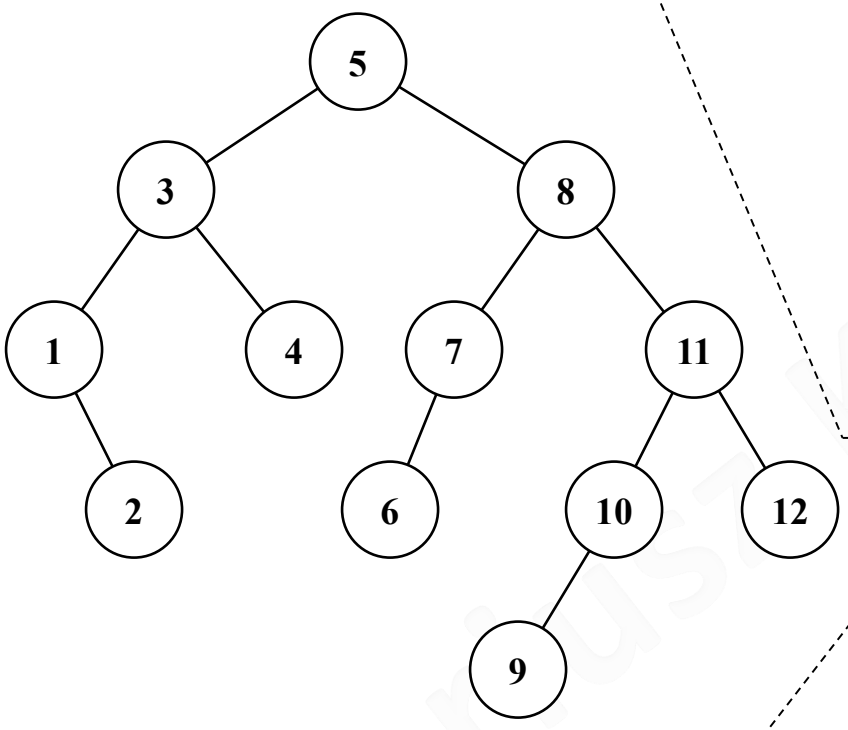
Balanced trees:

Red-black tree, B-tree

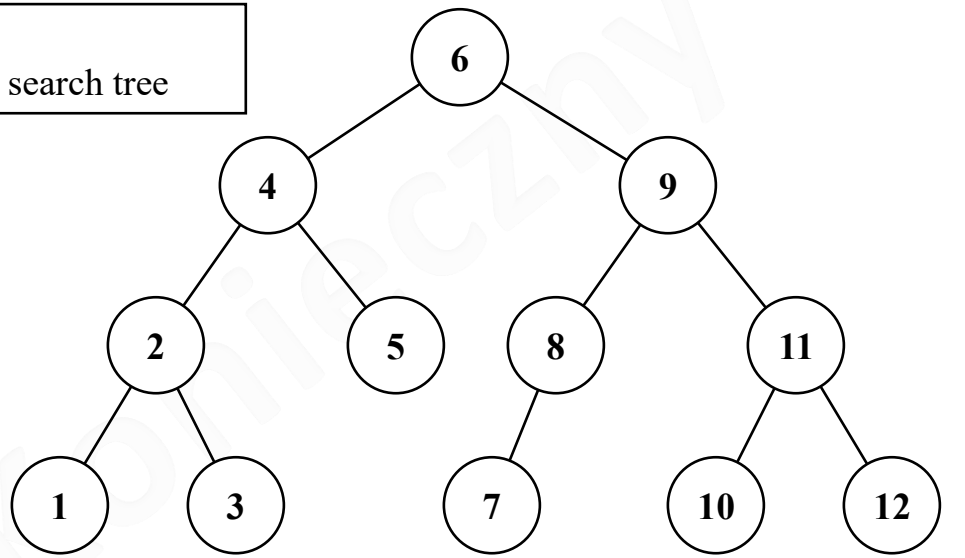
Contents

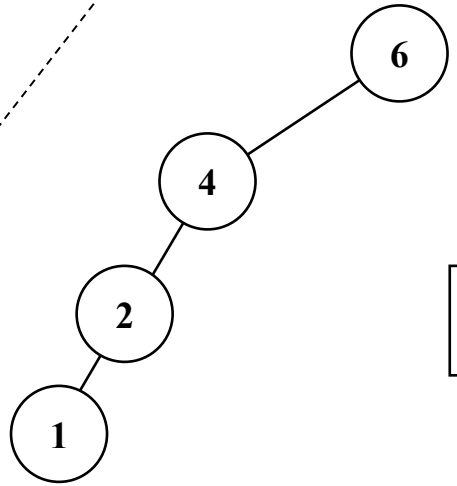
Binary trees

balanced
binary search tree



full
binary search tree





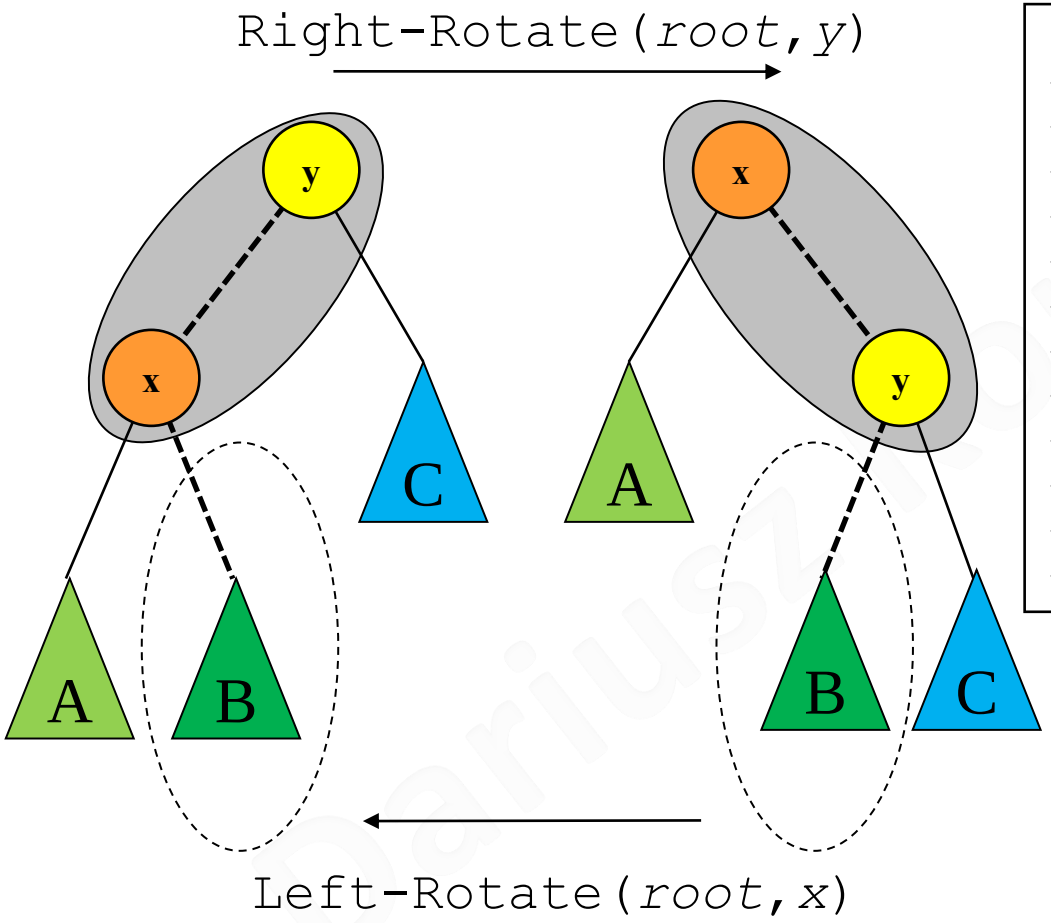
unbalanced
binary search tree

Self-balancing binary search tree

- Most operations on a binary search tree take time **directly proportional to the tree's height**, so it is desirable to keep the height small.
- Ordinary BST have the primary disadvantage that they can attain very large heights in rather ordinary situations, such as when the keys are inserted in order. The result is a data structure similar to a linked list, making all operations on the tree expensive.
- **Self-balancing binary trees** solve this problem by performing transformations on the tree (such as *tree rotation*) at key times, in order to reduce the height. Although a certain overhead is involved, it is justified in the long run by drastically decreasing the time of later operations.
- The height must always be at least the ceiling of log n, since there are at most 2^k nodes on the *k*th level; a complete or full binary tree has exactly this many levels. Balanced BSTs are not always so precisely balanced, since it can be expensive to keep a tree at minimum height at all times; instead, they keep the height within a constant factor of this lower bound. That means, the basic operations (Lookup, Insertion, Removal) are done in O(log *n*), so in time $c*\log n$, where $1<c<=2$.

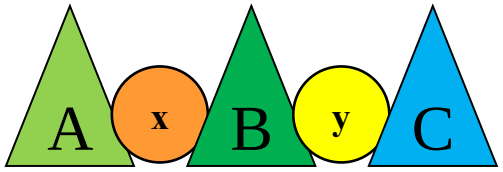
Tree rotation

- A **tree rotation** is an operation on a BST that changes the structure **without interfering** with the **order of** the elements.

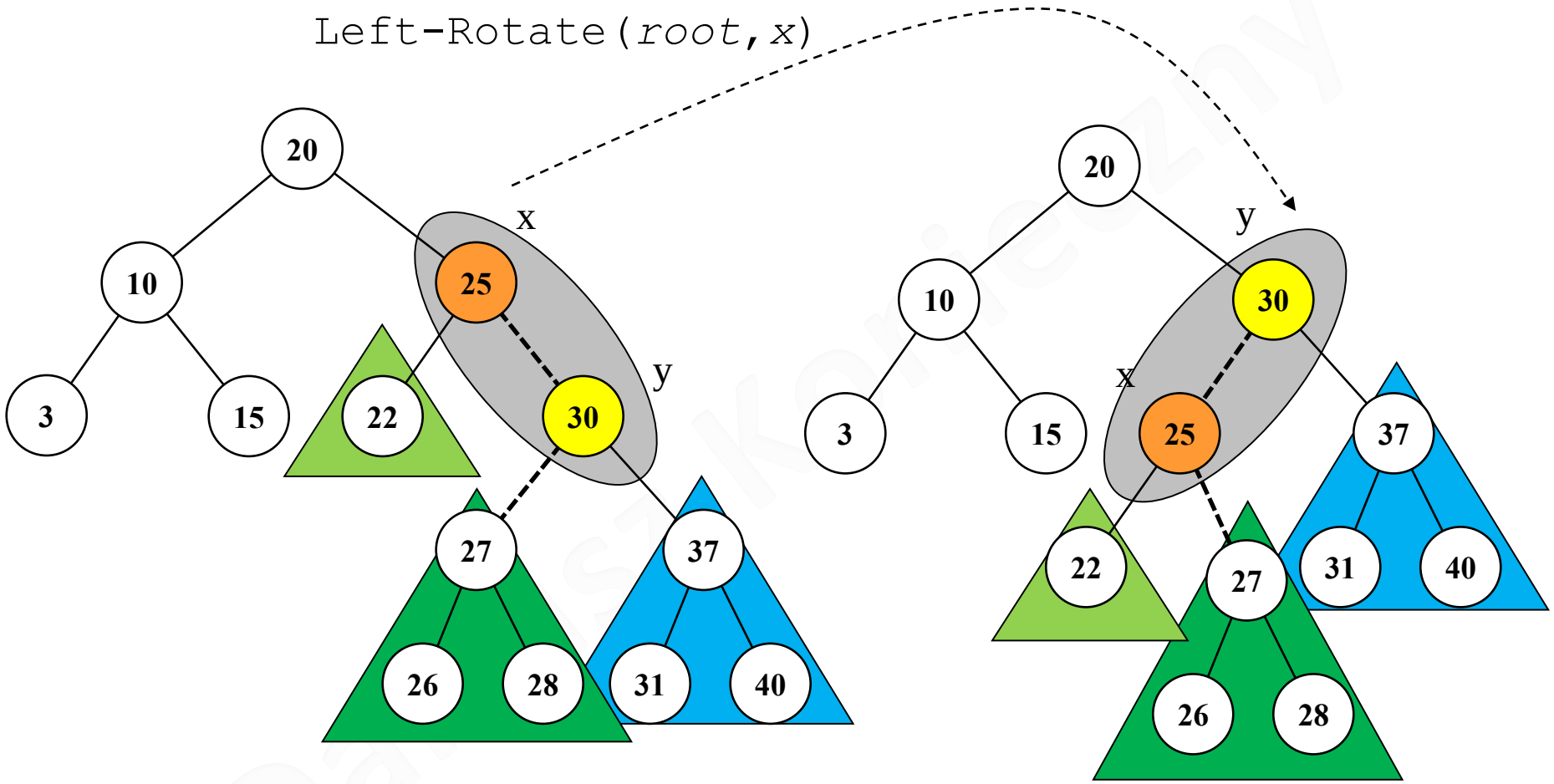



```
      Left-Rotate(root, x)
{ 1}  y := right[x]
{ 2}  right[x] := left[y]
{ 3}  if left[y]<>null
{ 4}        then p[left[y]] := x
{ 5}  p[y] := p[x]
{ 6}  if p[x] = null
{ 7}        then root := y
{ 8}  else if x = left[p[x]]
{ 9}           then left[p[x]] := y
{10}           else right[p[x]] := y
{11}  left[y] := x
{12}  p[x] := y
```

in-order left in-order right

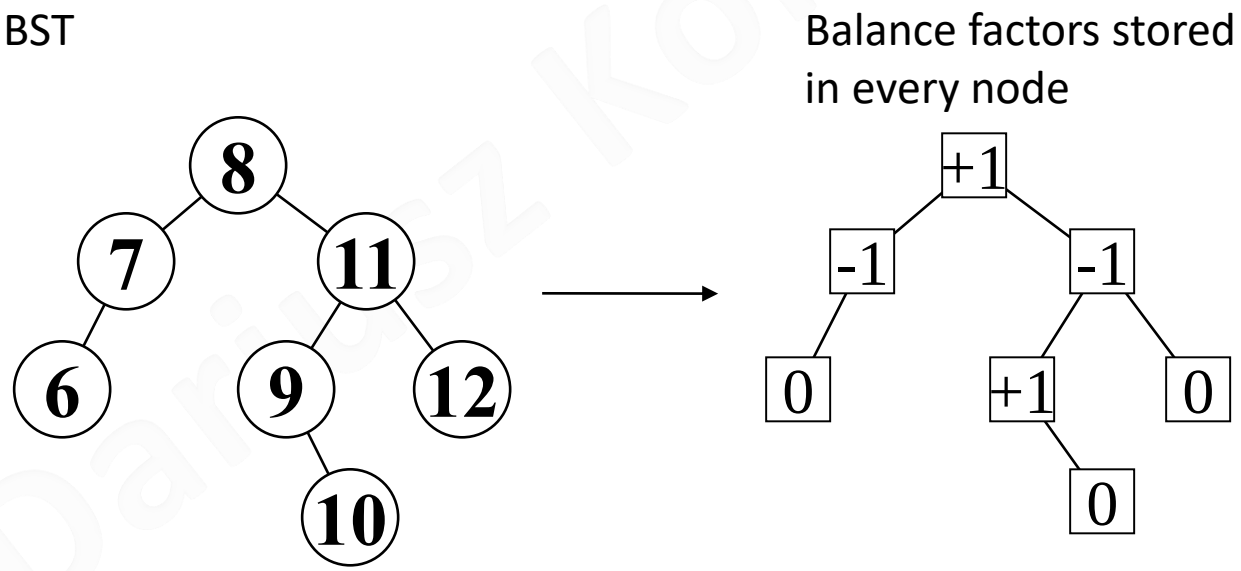


Tree rotation - example



AVL tree

- The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis
- **The balance factor** of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor **is stored directly at each node**.



AVL tree - analysis

Minimum number of nodes in AVL tree is represented by recursively formula (the notation AVL_h means a number of nodes in the AVL tree of the height h):

$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$

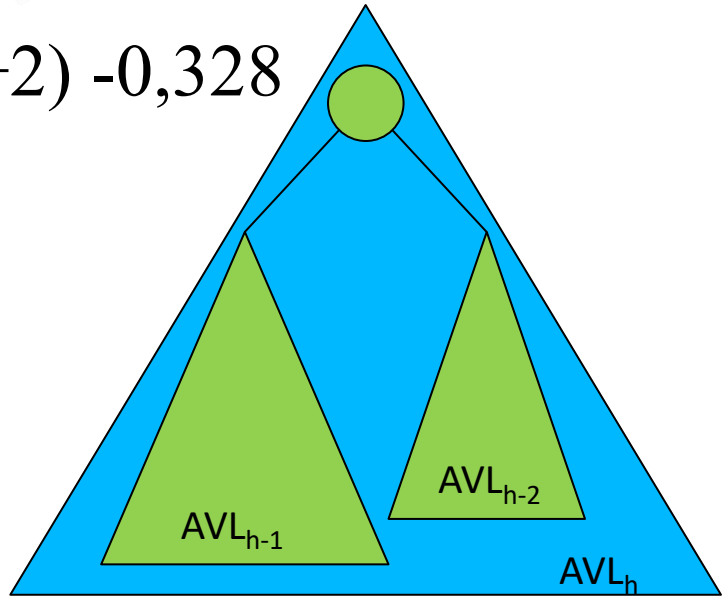
where: $AVL_0 = 0$, $AVL_1 = 1$,

therefore a limitation of AVL tree's height h is represented by formula (proved by the authors):

$$\lg(n+1) \leq h < 1{,}44 \lg(n+2) - 0{,}328$$

It means:

$$h = O(\lg n)$$



AVL tree - summary

- The AVL tree needs to store an additional five different values in a node: -2, -1, 0, 1, 2.
- Tree height is not more than 1.5 * log *n*.
- Thanks to the rotation performed during the addition or removal, it maintains correct balancing factors.
- Repairing occurs at levels: from the level of modification of the tree towards the root. At each level, a maximum of two rotations are performed, i.e. the complexity of these operations is O(log *n*).
- In standard libraries of many programming languages AVL trees have been replaced by a **red-black tree**, which despite the theoretically higher pessimistic height, in practice works faster.

Red-Black tree

- A red-black tree is a binary search tree with **one extra bit** of storage per node: its **color**, which can be either **RED** or **BLACK**. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that **no such path is more than twice** as long as any other, so that the tree is approximately **balanced**.
- Each node of the tree now contains the fields *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL. We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.
- A binary search tree is a red-black tree if it satisfies the following **red-black properties**:

  1) Every node is either red or black

  2) The root is black

  3) Every leaf (NIL) is black

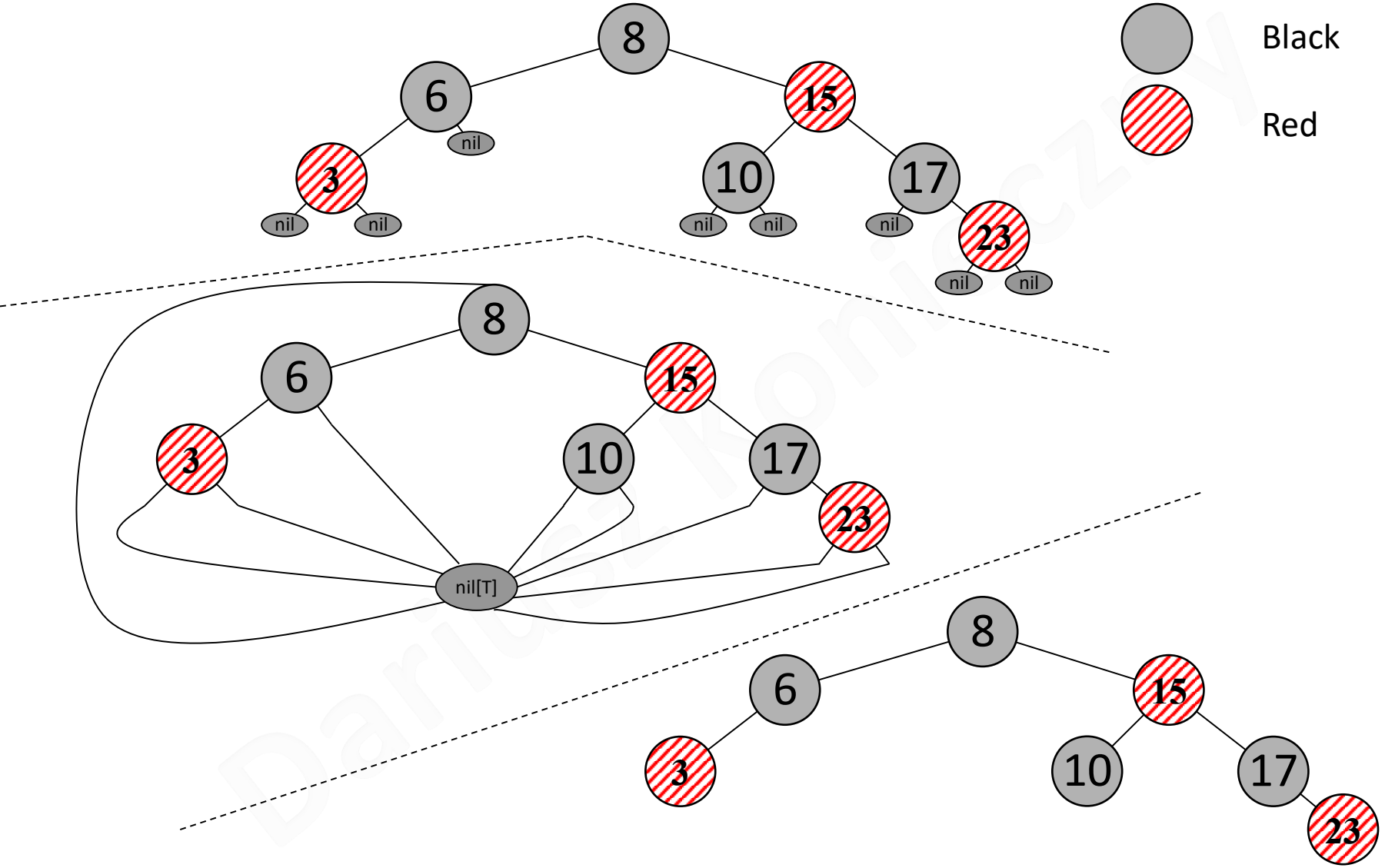
  4) If a node is red, then both children are black

  5) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

RB-Tree – black height (bh)

- The number of black nodes on any path from node x (excluding node x) to the leaf is called the **black node height**, which is denoted bh(x).
- By the black height of the RB-tree we will understand the black height of its root.
- Fact: A red-black tree with *n* internal nodes has a height of at most 2lg(*n* + 1).
- On subsequent slides, bh will mean the black height to a given place in the tree.

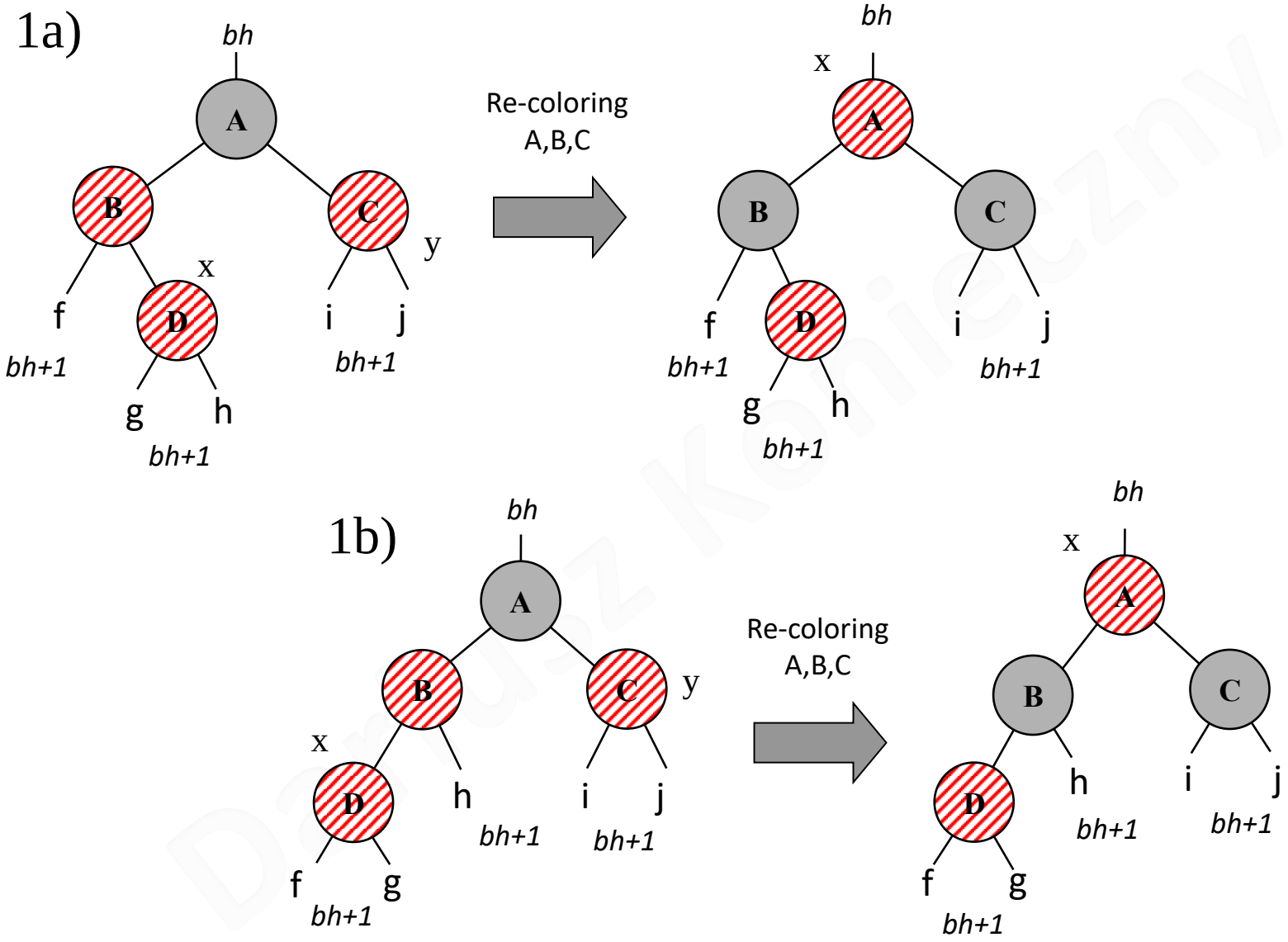
RB-Tree - representations



Insertion in RB-tree 1/3

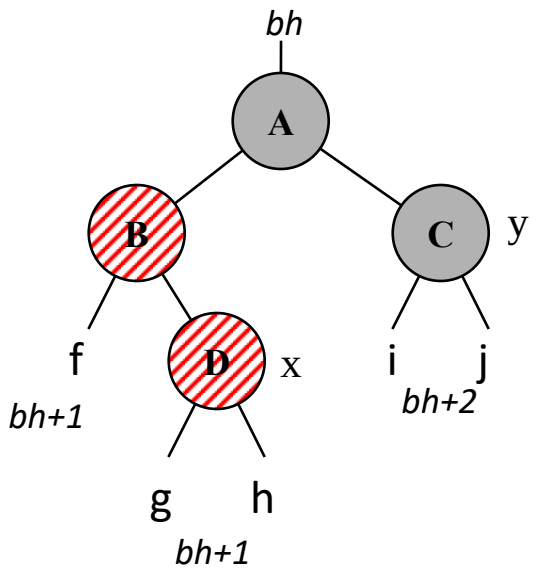
- Insertion involves **inserting a new node as a leaf tree into a regular binary tree** and **coloring** the node in **red** (then property 5 is not broken, but property 4 may be broken).
- In case of breaking the property 4, the tree is repaired from the place of insertion by re-coloring and possibly rotations in the tree.
- Next slides - the repair rules, when node x is currently analyzed in the **left subtree** of **his grandfather**. Similarly, it will be in symmetrical cases.

Insertion in RB-tree 2/3



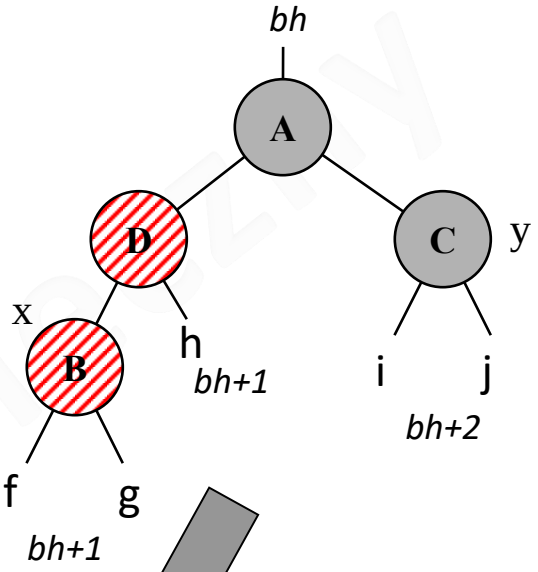
Insertion in RB-tree 3/3

2)

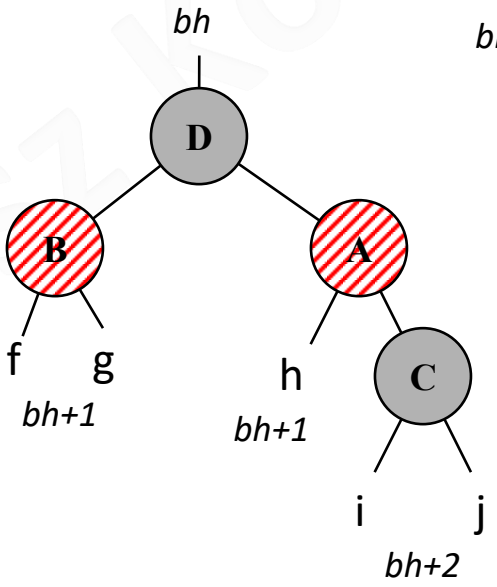


3)

`Left-Rotate(B)`



`Right-Rotate(A)`,
Re-coloring A,D

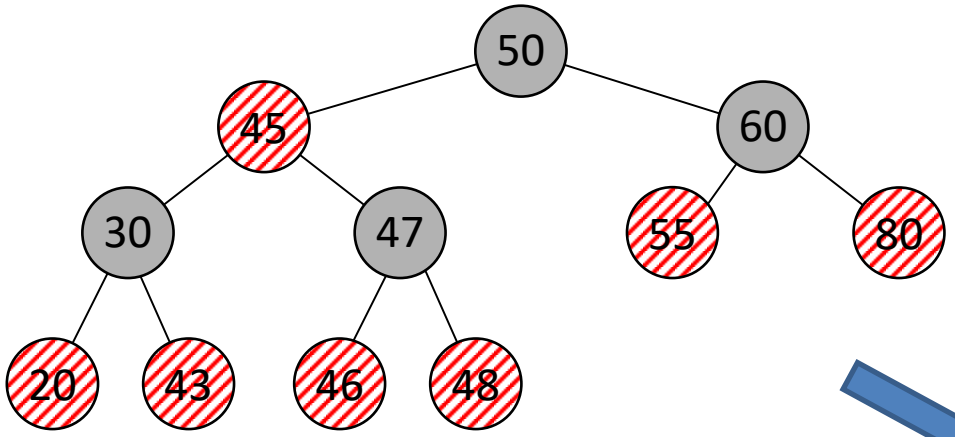


Stop

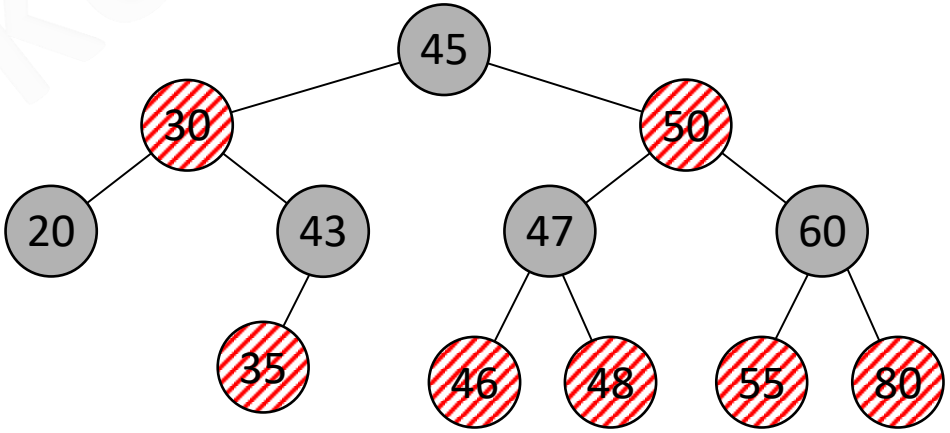
Insertion into RB-tree - code

```
     RB-Insert(root, x)
{ 1}   Tree-Insert(root,x)
{ 2}   color[x] := RED
{ 3}   while x <> root and color[p[x]] = RED do
{ 4}       if p[x] = left[p[p[x]]] then
{ 5}          y := right[p[p[x]]]
{ 6}          if color[y] = RED then          { case 1}
{ 7}              color[p[x]] := BLACK        { case 1}
{ 8}              color[y] := BLACK           { case 1}
{ 9}              color[p[p[x]]] := RED       { case 1}
{10}              x := p[p[x]]
{11}          else
{12}               if x = right[p[x]] then
{13}                   x := p[x]              { case 2}
{14}                   Left-Rotate(root,x)    { case 2}
{15}              color[p[x]] := BLACK        { case 3}
{16}              color[p[p[x]]] := RED       { case 3}
{17}              Right-Rotate(root, p[p[x]])   { case 3}
{18}       else ....{ same as then clause with „right” and „left”
{19}                  exchanged}
{20}   color[root] := BLACK
```

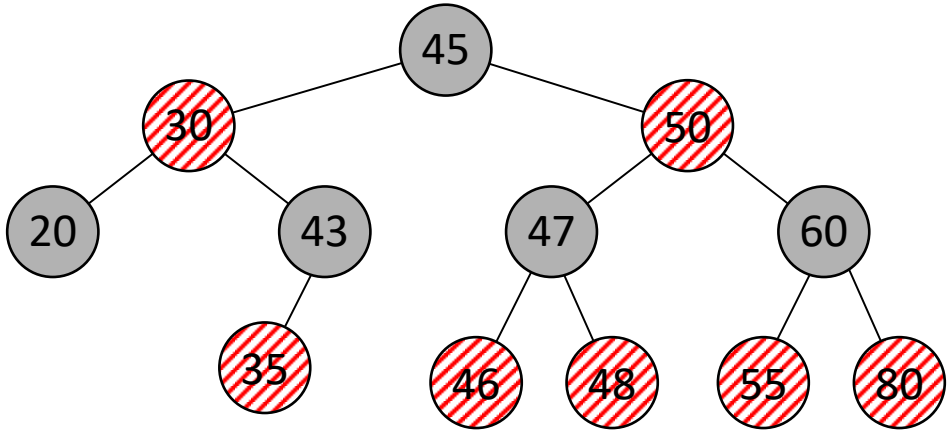
Insertion into RB-tree – example 1/3




```
RB-Insert(root,35)
```

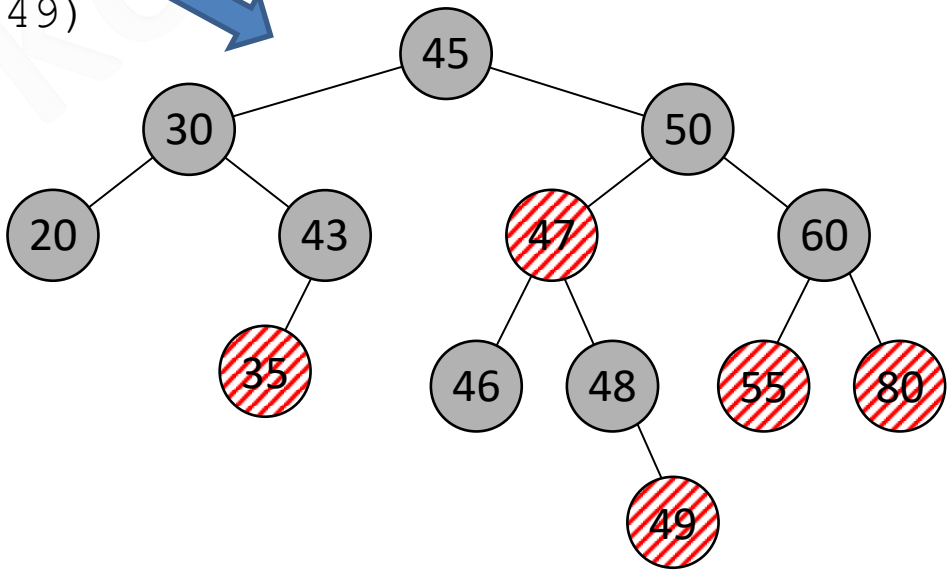


Insertion into RB-tree – example 2/3

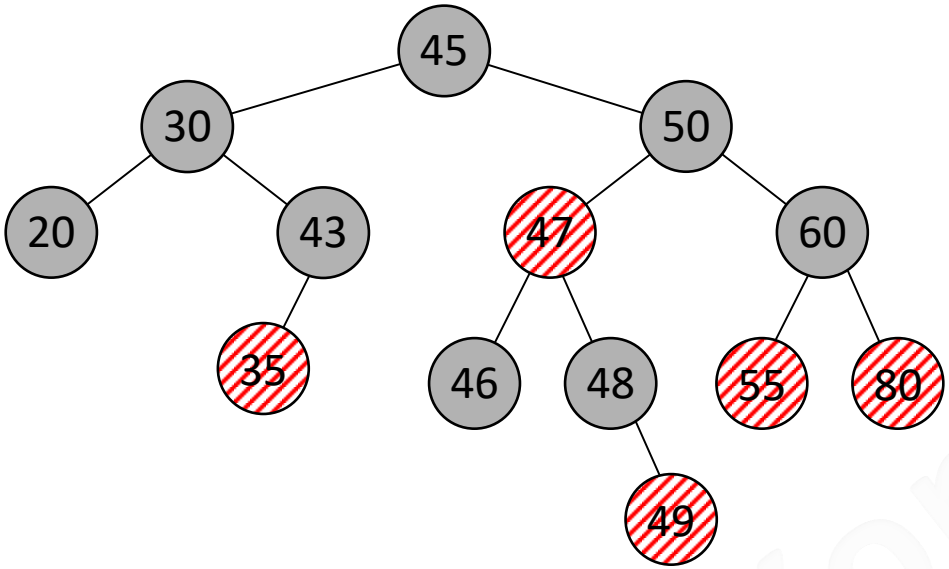



```
RB-Insert(root,49)
```

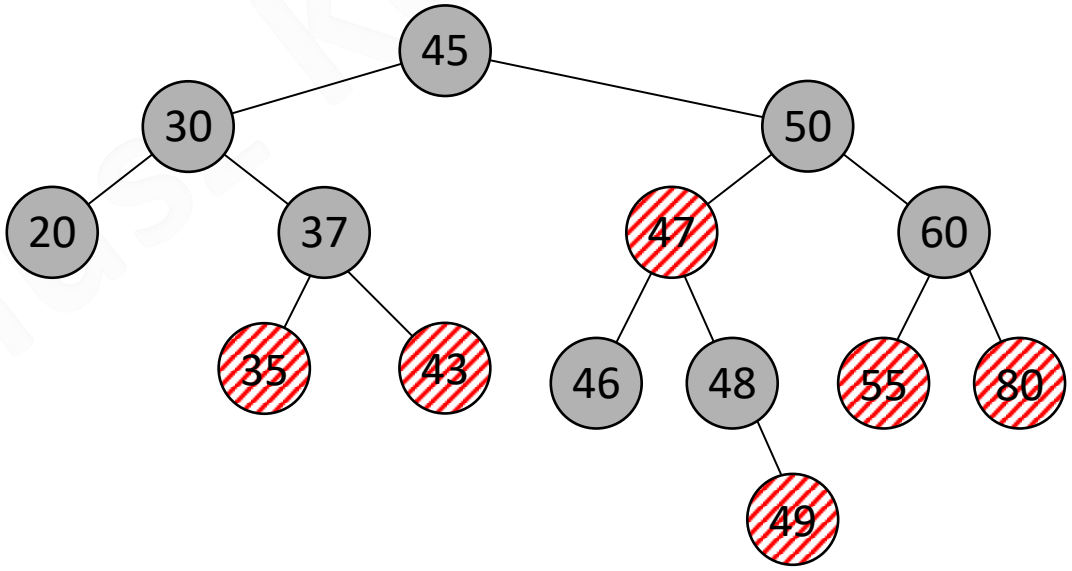
bh++



Insertion into RB-tree – example 3/3



RB-Insert(root,37)



Deletion from RB-Tree

- `RB-Delete` is similar to `Tree-Delete`, but:
  - all reference to `NIL` are replaced by references to sentinel `nil[T]`
  - test for whether `x` is `NIL` is removed and an assignment `p[x]:=p[y]` is performed unconditionally. Thus, if `x` is the sentinel `nil[T]`, its parent pointer points to the parent of the spliced-out node `y`.
  - After deleting, if `y` is **`black`**, call to `RB-Delete-Fixup` is made, because some red-black properties can be violated.

Deletion from RB-Tree - code

```
      RB-Delete(T, z)
{ 1}  if (left[z] = nil[T]) or (right[z] = nil[T])
{ 2}      then y := z
{ 3}      else y := Tree-Successor(z)
{ 4}  if left[y] <> nil[T]
{ 5}      then x := left[y]
{ 6}      else x := right[y]
{ 7}  p[x] = p[y]
{ 8}  if p[y] = nil[T]
{ 9}      then root[T] := x
{10}      else if y = left[p[y]]
{11}               then left[p[y]] := x
{12}               else right[p[y]] := x
{13}  if y <> z
{14}      then swap(key[z], key[y])
{15}      { if node y has another fields, copy them here}
{16}  if color[y] = BLACK
{17}      then RB-Delete-Fixup(T,x)
      return y
```

Deletion fixup in RB-Tree

- If we spliced-out node y in `RB-Delete`, three problems may arise:

1) `y` had been the `root` and a red child of `y` becomes the new `root` (property 2 is violated)

2) both `x` and `p[y]` (now also `p[x]`) were **red** (property 4 is violated)

3) `y`'s removal causes any path that previously contained `y` to have one fewer black node (property 5 is violated)

We can correct the problem 3) by saying that node x has an „**extra**" black: `RED` x is **red-and-black**, `BLACK` x is **doubly-black**.

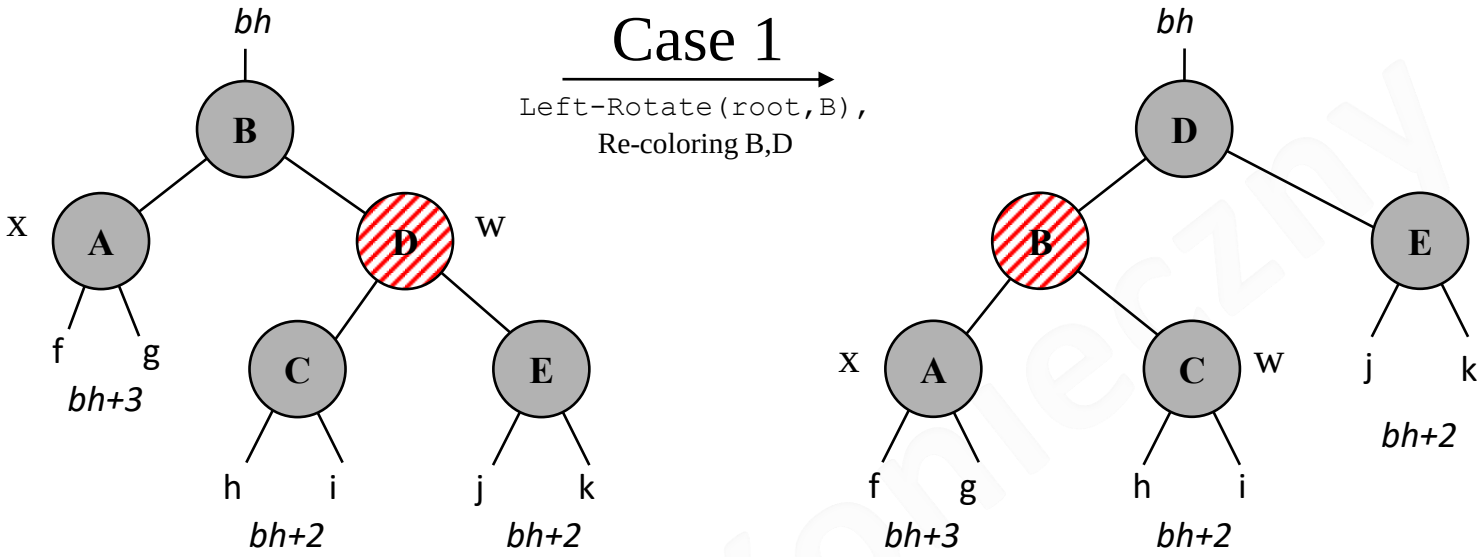
We traverse (in a specific way) the tree to the root looking for red node which we can recolor to black.

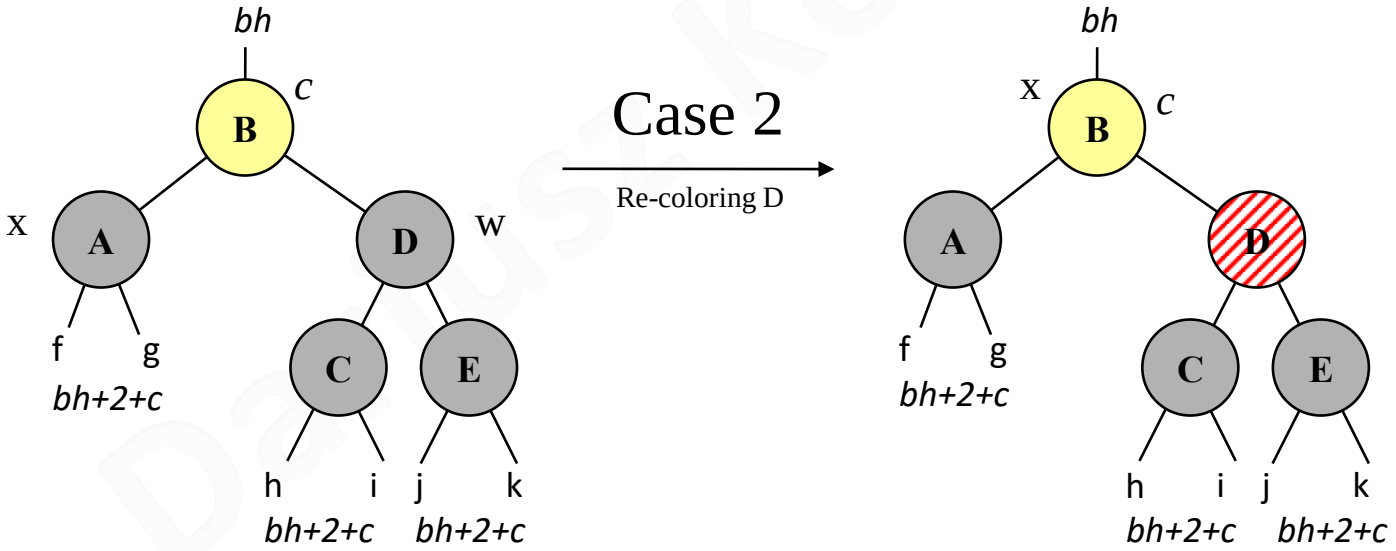
The method of operation will depend on the color of the brother of the analyzed node and on the colors of his children.

`x` - node analyzed, with an "additional" black token
`w` - brother of the node `x`

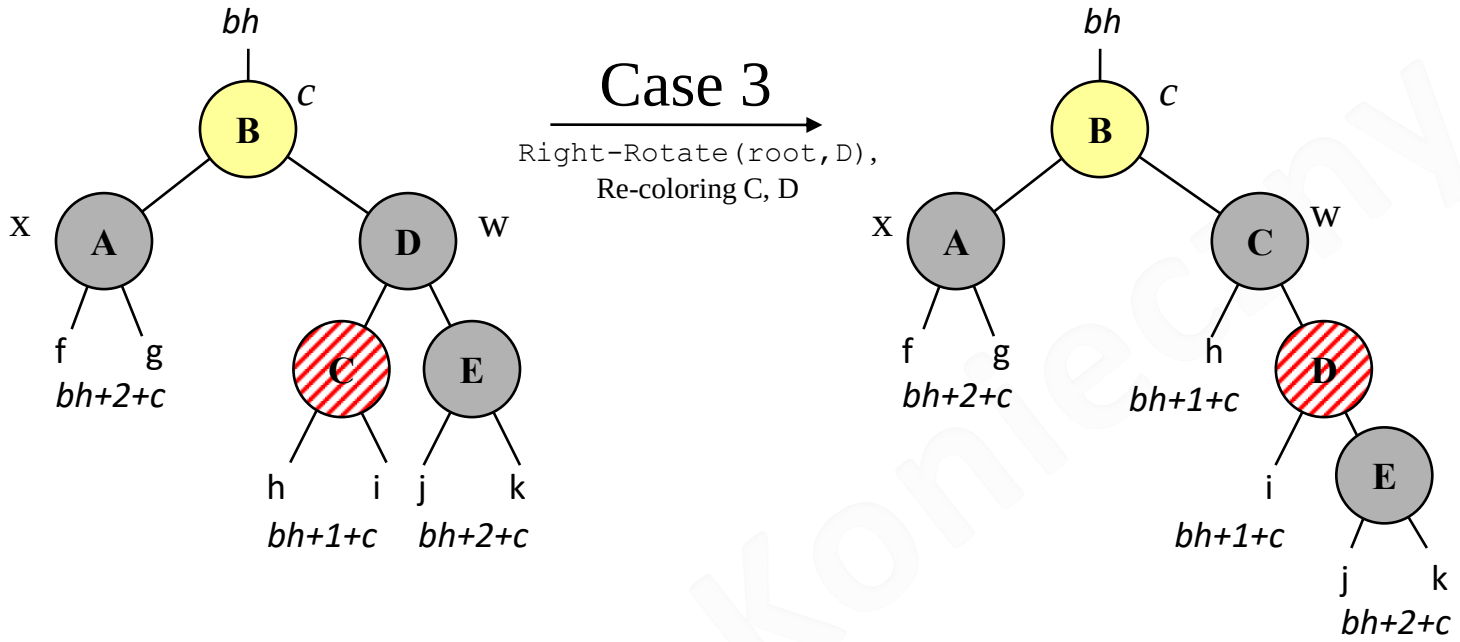
Presented only cases when x is in the left subtree of his father.

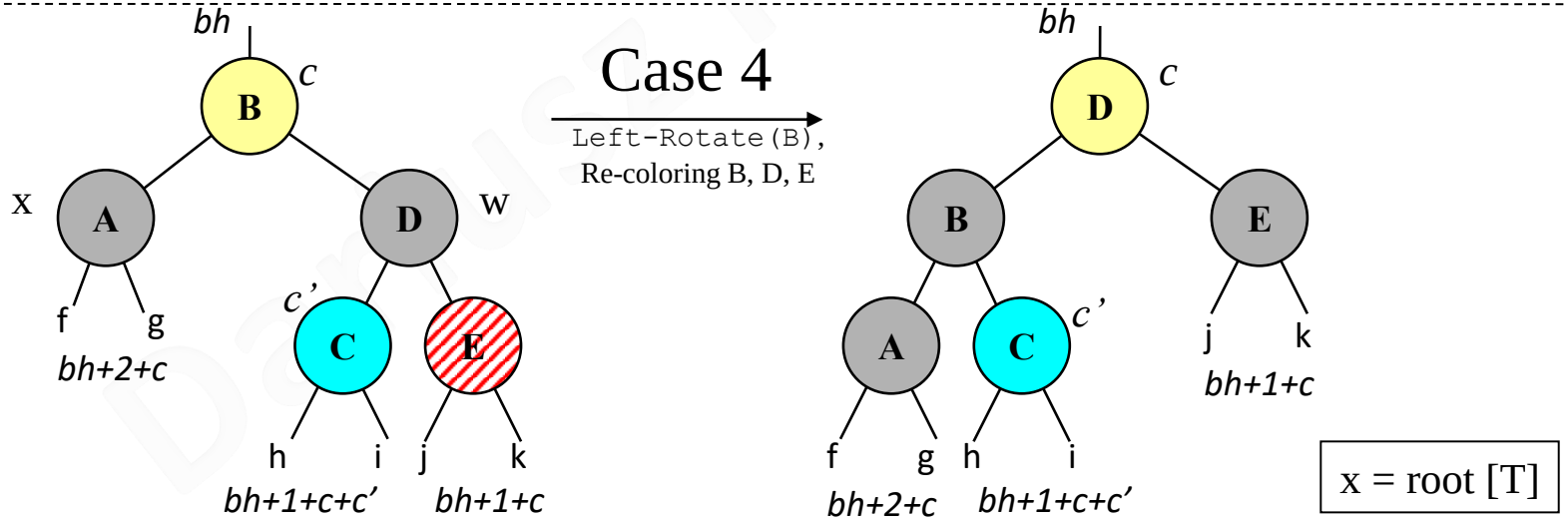
Fixing RB-tree 1/2





Fixing RB-tree 2/2



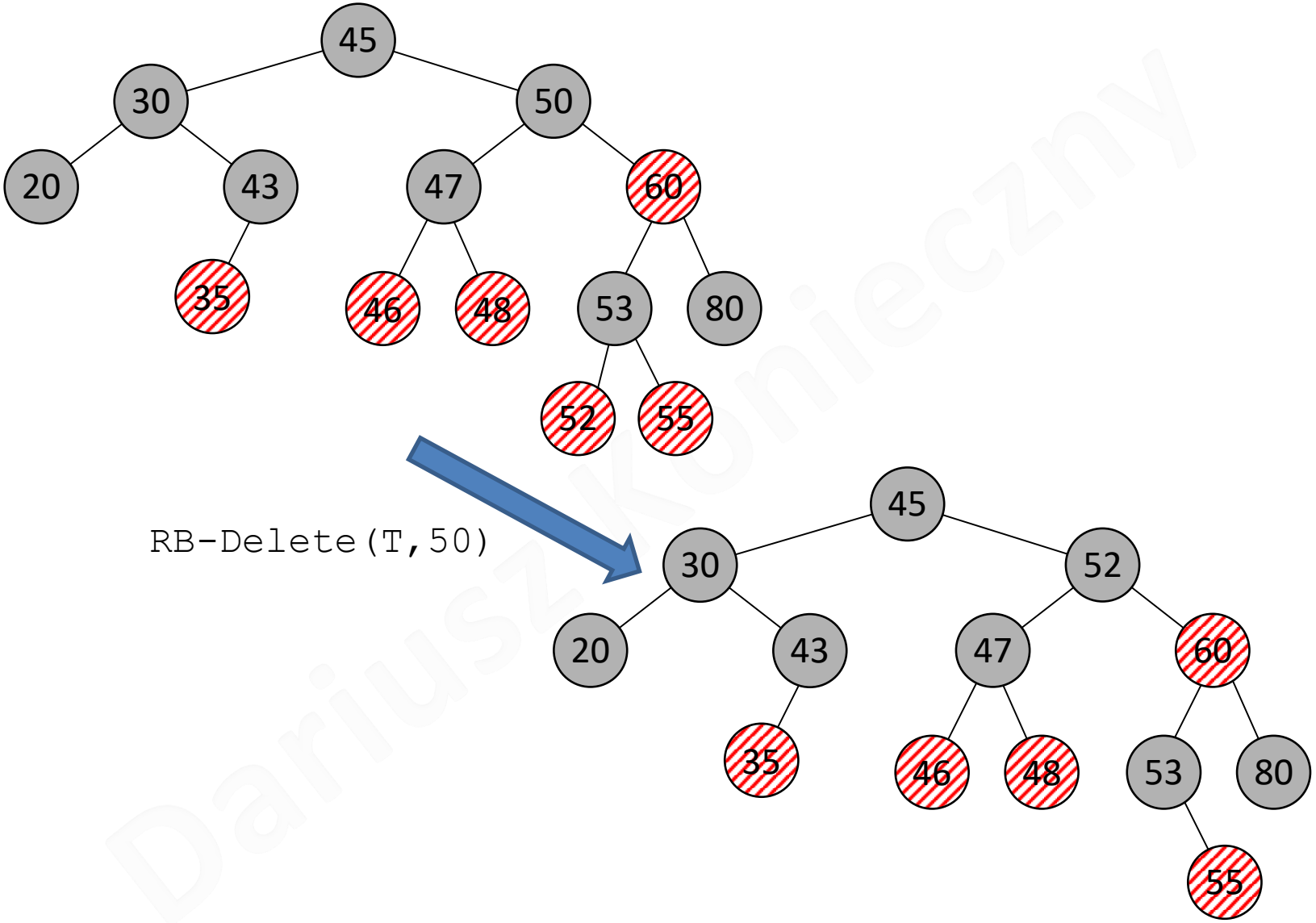


x = root [T]

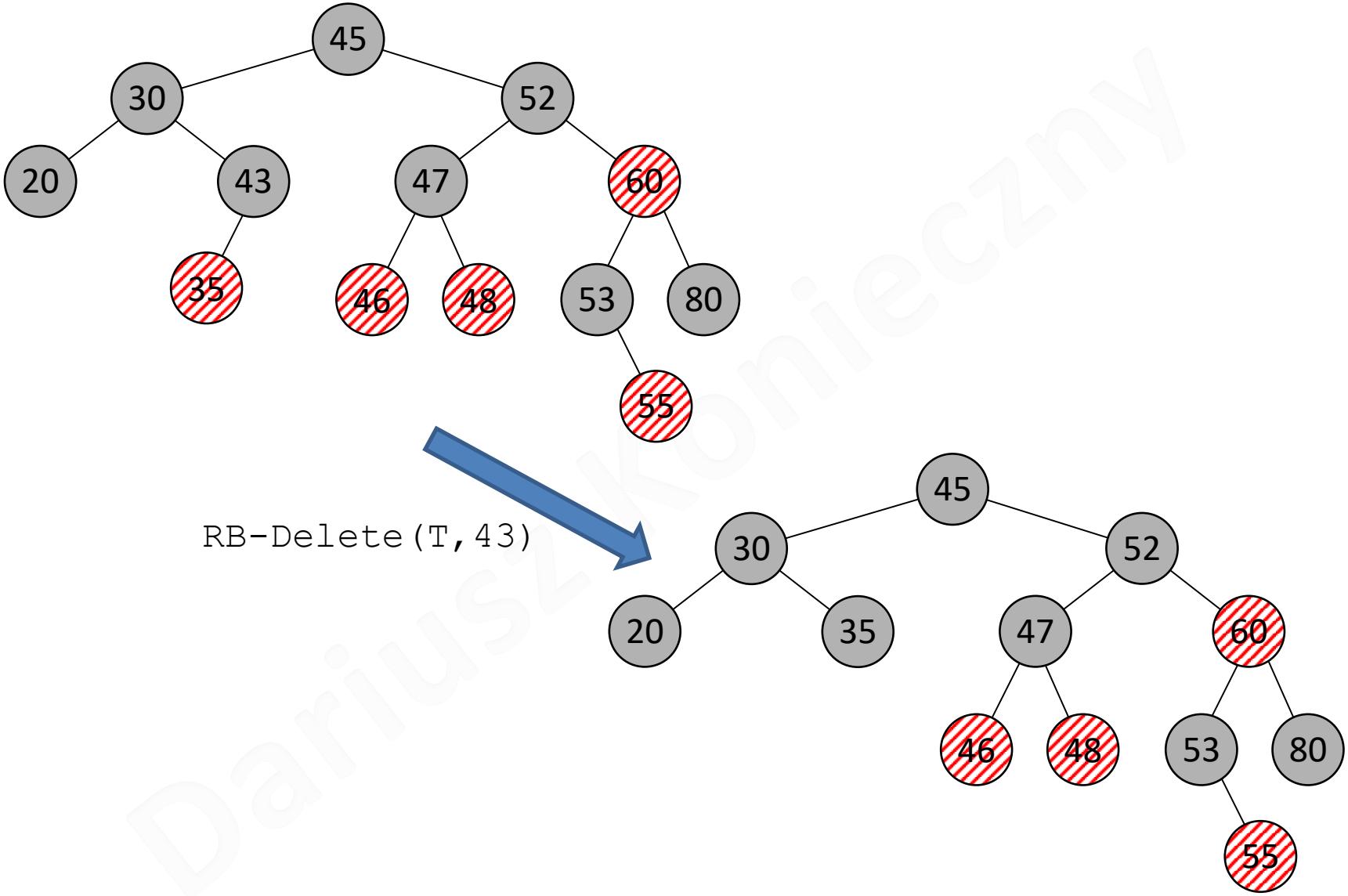
Fixing RB-Tree - code

```
     RB-Delete-Fixup(T, x)
{ 1} while x!= root[T] and color[x]=BLACK
{ 2}     do if x=left[p[x]] then
{ 3}           w:= right[p[x]]
{ 4}           if color[w]=RED then        // case 1
{ 5}              color[w]:=BLACK          // case 1
{ 6}              color[p[x]]:=RED         // case 1
{ 7}              Left-Rotate(T,p[x])        // case 1
{ 8}              w:=right[p[x]]     // case 1
{ 9}           if color[left[w]]=BLACK and color[right[w]]=BLACK then
{10}              color[w]:=RED            // case 2
{11}              x:=p[x]                  // case 2
{12}           else if color[right[w]]=BLACK then
{13}                   color[left[w]]:=BLACK  // case 3
{14}                   color[w]:=RED      // case 3
{15}                   Right-Rotate(T,w)  // case 3
{16}                   w:=right[p[x]]     // case 3
{17}              color[w]:=color[p[x]]   // case 4
{18}              color[p[x]]:=BLACK      // case 4
{19}              color[right[w]]:=BLACK // case 4
{20}              Left-Rotate(T,p[x])        // case 4
{21}              x:=root[T]           // case 4
{22}        else /* */
{23} color[x]=BLACK
```

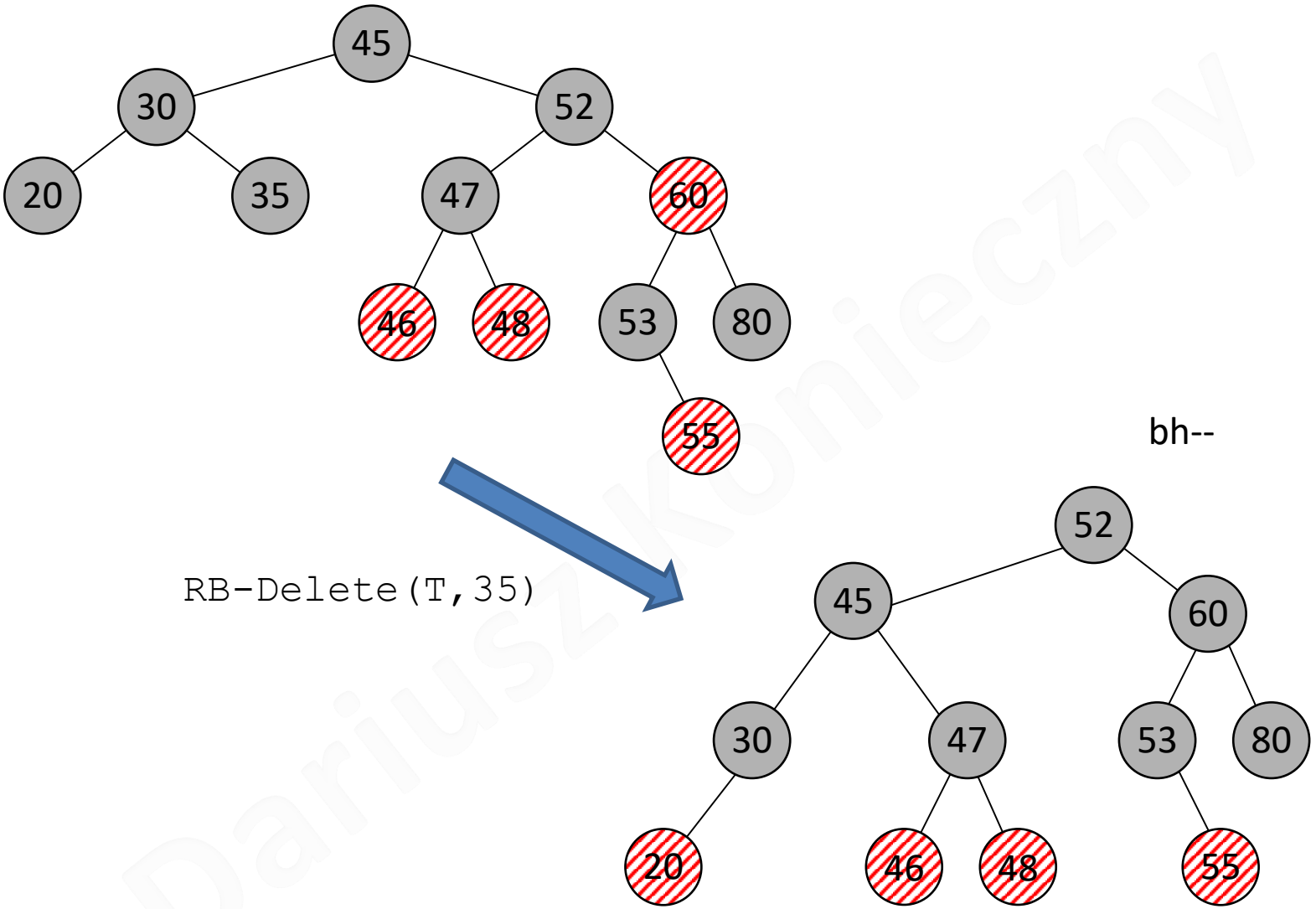
Deletion from RB-tree – example 1/3



Deletion from RB-tree – example 2/3



Deletion from RB-tree – example 3/3

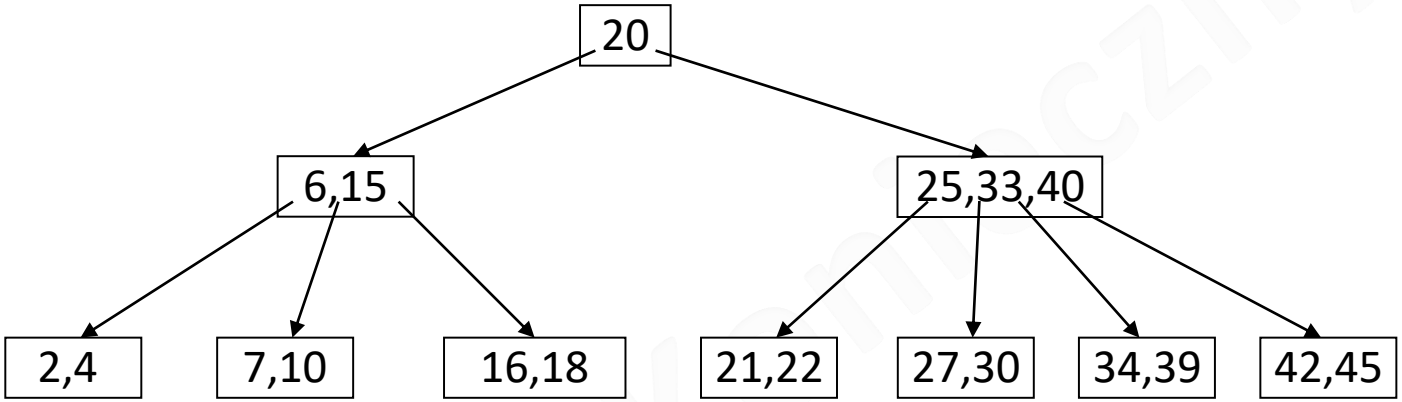


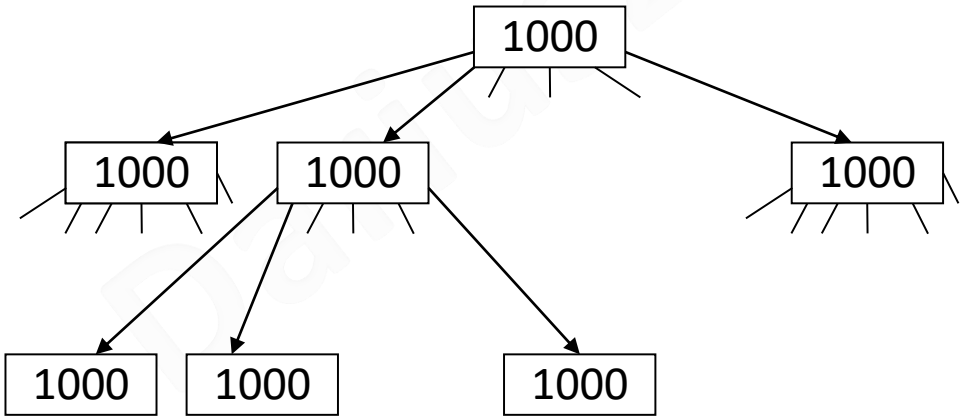
Complexity of RB-Tree

- Insertion:
  - insertion as into BST: O(*h*)
  - fixup - O(*h*) recolor and 2 rotation: O(*h*)
  - worse-case complexity: O(*h*)=O(log *n*)
- Deletion:
  - deletion as from BST: O(*h*)
  - fixup – O(*h*) recolor and 2 rotation: O(*h*)
  - worse-case complexity: O(*h*)=O(log *n*)
- All other operations depending on the height of the tree - complexity O(log n)
- The `TreeMap<K,V>` class is implemented using red-black trees according to the Cormen at al. algorithm presented in this lecture.

B-tree – idea, example

- Let us expand the idea of a tree of binary search tree, which will have more than two descendants, but still have to be regular, if possible.





1 node
1000 keys

1001 node
1 001 000 keys

1 002 001 node
1 002 001 000 keys

B-Tree – why/when

- big number of keys
- external memory: disks divided onto sectors
  - long time operation:
    - DISK-READ
    - DISK-WRITE
  - short time operation:
    - CPU operation like assigments

B-Tree - definition

A ***B-tree*** T is a rooted tree (whose root is $root[T]$) having the following properties:

1) Every node x has the following fields:
   - *a)* $n[x]$ - the number of keys currently a. stored in node x,
   - b) the $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$,
   - *c)* $leaf[x]$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2) Each internal node x also contains $n[x]+1$ pointers $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3) The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then
   $$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \cdots \le key_{n[x]}[x] \le k_{n[x]+1}.$$
4) All leaves have the same depth, which is the tree's height h.
5) There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:

   a) Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

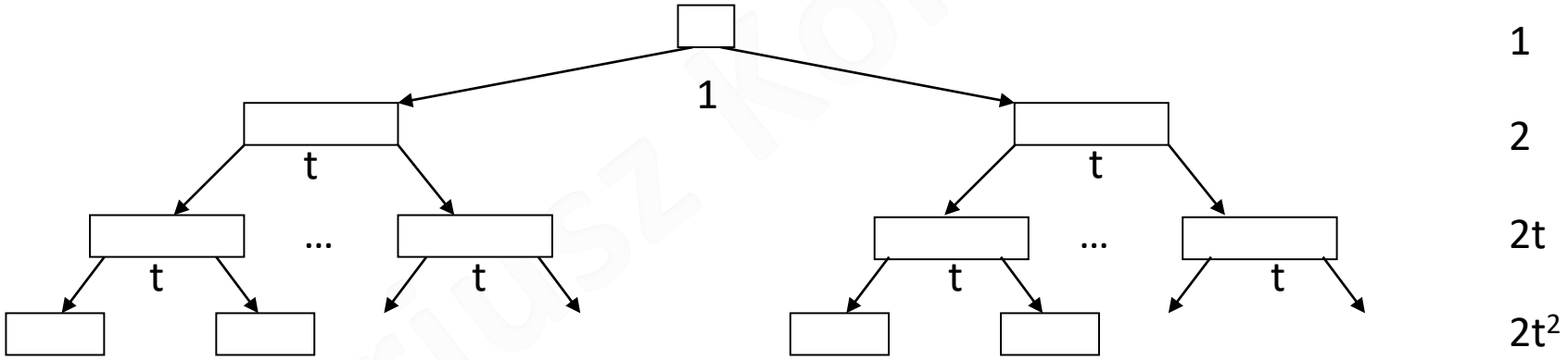
   b) Every node can contain at most $2t$ - 1 keys. Therefore, an internal node can have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t$ - 1 keys.

If the t=2 (minimal degree), we call such tree a **2-3-4 tree**.

B-Tree - height property

- The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree.
- If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \lg_t \frac{n+1}{2} \quad \Rightarrow \quad h = O(\lg_t n)$$



B-tree search

- Idea: similar to BST, only on each node there are more keys to compare
- The procedure returns a pair: **node** and **index** within this node where the key is found. In the case of a failure, it returns **`null`**.
- disk pages accesses: $\Theta(h) = \Theta(\lg_t n)$
- CPU time: $O(th) = \Theta(t \lg_t n)$

```
     B-Tree-Search(x, k)
{ 1} i:=1
{ 2} while i ≤ n[x] and k > key_i[x] do
{ 3}     i:=i + 1
{ 4} if i != n[x] and k = key_i[x] then
{ 5}     return (x, i)
{ 6} if leaf [x] then
{ 7}     return NIL
{ 8} else
{ 9}     DISK-READ(c_i[x])
{10}     return B-Tree-Search(c_i[x], k)
```

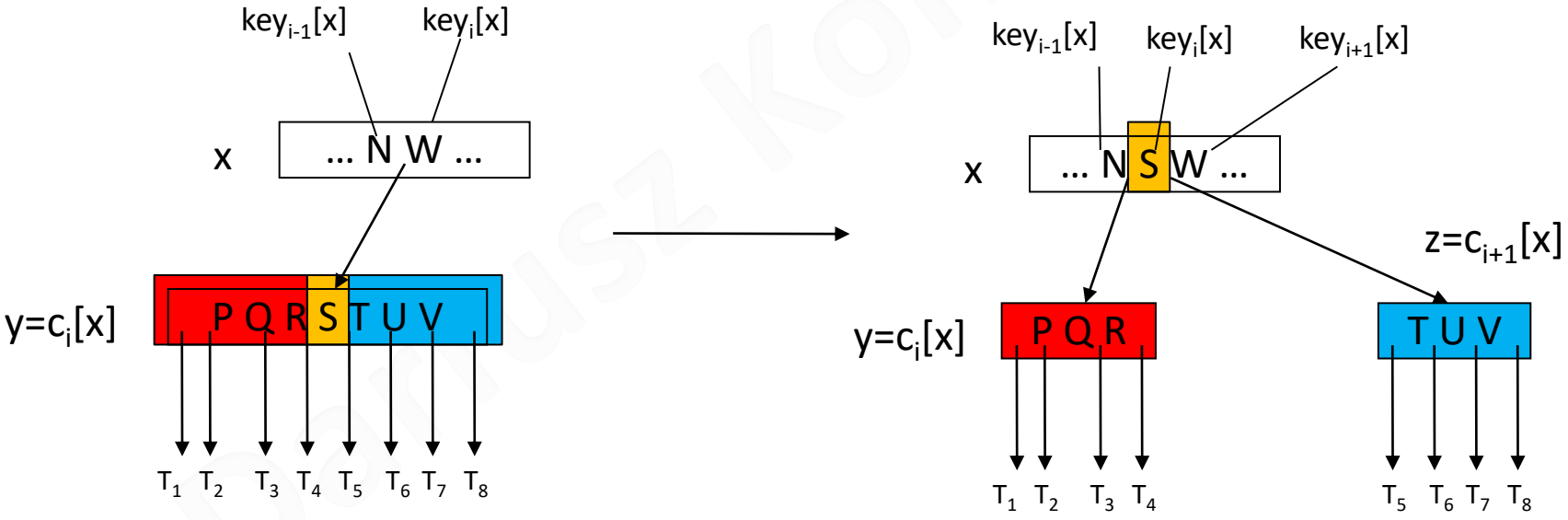
B-tree create

- To build a B-tree *T*, we first use `B-Tree-Create` to create an empty root node and then call `B-Tree-Insert` to add new keys. Both of these procedures use an auxiliary procedure `Allocate-Node`, which allocates one disk page to be used as a new node in *O*(1) time. We can assume that a node created by `Allocate-Node` requires no DISK_READ, since there is as yet no useful information stored on the disk for that node.
- disk pages accesses: O(1)
- CPU time: O(1)

```
     B-Tree-Create(T)
{ 1} x := Allocate-Node()
{ 2} leaf[x]=true
{ 3} n[x]=0
{ 4} DISK_WRITE(x)
{ 5} root[T] := x
```

B-tree - Split child

- The procedure `B-Tree-Split-Child` takes as input a ***nonfull*** internal node x (assumed to be in main memory), an index i, and a node y (also assumed to be in main memory) such that $y = c_i[x]$ is a *full* child of x. The procedure then splits this child in two and adjusts x so that it has an additional child.



B-tree - Split child - code

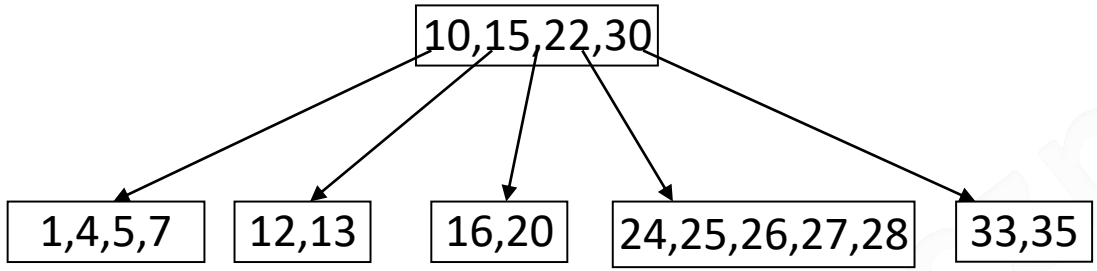
```
     B_Tree_Split_Child(x,i,y)
{ 1} z := Allocate_Node()
{ 2} leaf[z]:=leaf[y]
{ 3} n[z]:=t - 1
{ 4} for j:=1 to t - 1 do
{ 5}     key_j[z] := key_{j+t}[y]
{ 6} if not leaf [y] then
{ 7}     for j:=1 to t do
{ 8}         c_j[z]:=c_{j+t}[y]
{ 9} n[y]:=t - 1
{10} for j:=n[x] + 1 downto i + 1 do
{11}     c_{j+1}[x]:=c_j[x]
{12} c_{i+1}[x]:=z
{13} for j:=n[x] downto i do
{14}     key_{j+1}[x]:=key_j[x]
{15} key_i[x]:=key_t[y]
{16} n[x]:=n[x] + 1
{17} DISK-WRITE(y)
{18} DISK-WRITE(z)
     DISK-WRITE(x)
```

- disk pages accesses: O(1)
- CPU time: Θ(t)

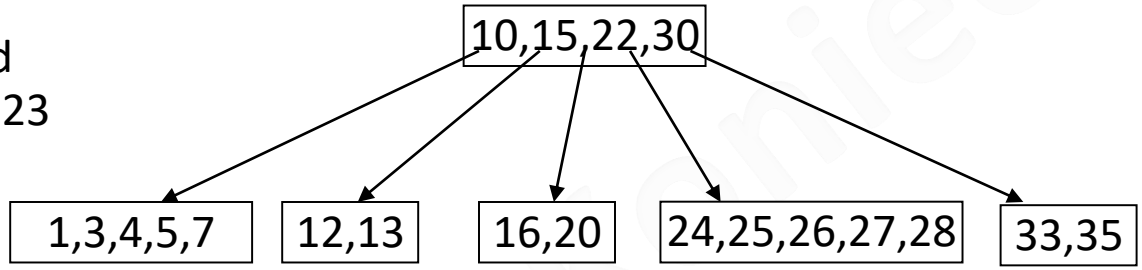
B-tree insertion – example 1/2

initial tree

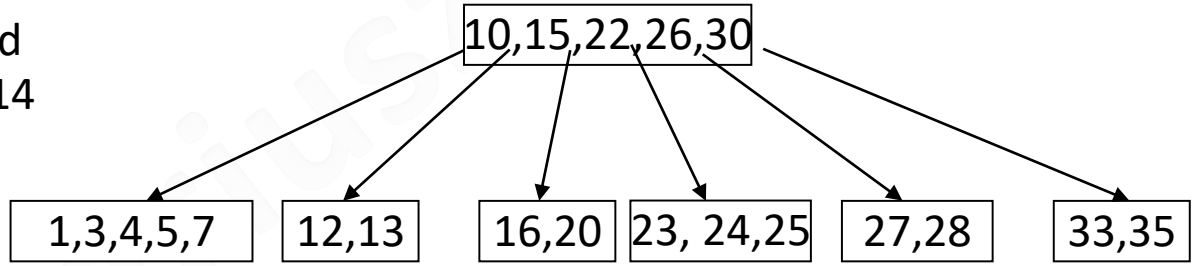
t=3
let insert 3



3 inserted
let insert 23

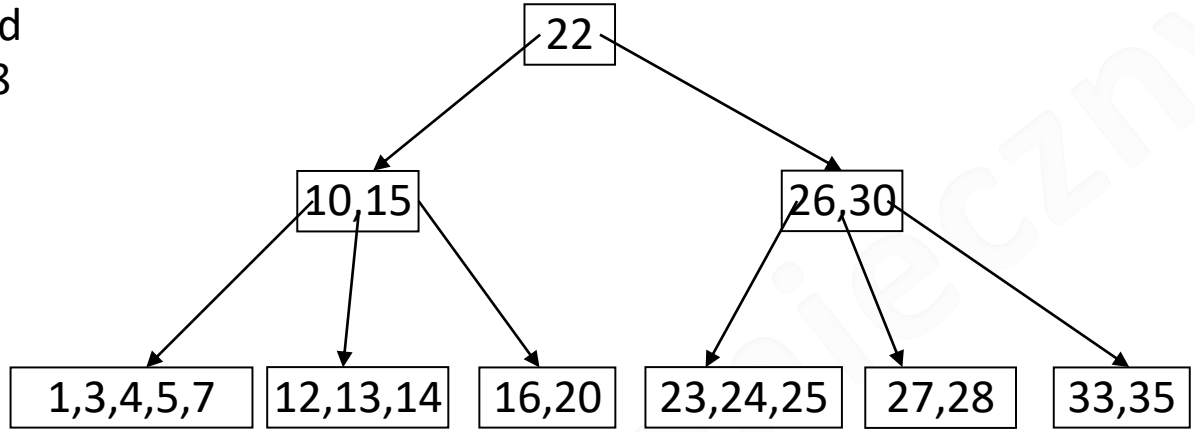


23 inserted
let insert 14

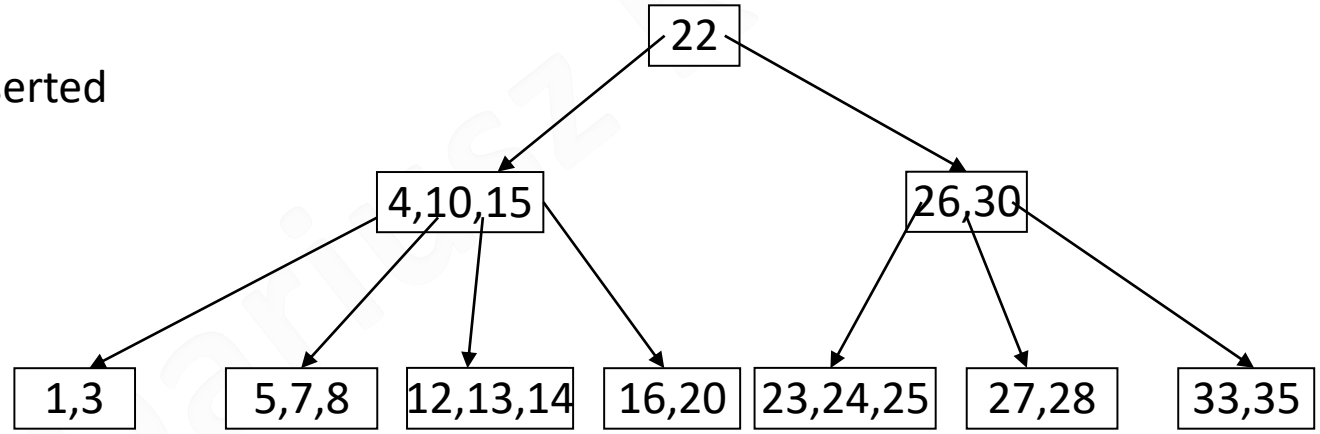


B-tree insertion – example 2/2

14 inserted
let insert 8



8 inserted



B-tree insert

- To split a full root, we will first make the root a child of a new empty root node, so that we can use `B-Tree-Split-Child`. The tree thus grows in height by one; splitting is the **only** means by which the tree grows.

```
     B-Tree-Insert(T,k)
{ 1} r := root[T]
{ 2} if n[r]=2*t-1 then
{ 3}   s := Allocate-Node()
{ 4}   root[T] := s
{ 5}   leaf[s] := false
{ 6}   n[s] := 0
{ 7}   c1[s] := r
{ 8}   B-Tree-Split-Child(s,1,r)
{ 9}   B-Tree-Insert-Nonfull(s,k)
{10} else
{11}   B-Tree-Insert-Nonfull(r,k)
```

B-tree insert nonfull node

```
     B-Tree-Insert-Nonfull(x,k)
{ 1} i := n[x]
{ 2} if leaf[x] then
{ 3}   while i>= 1 and k<key_i[x] do
{ 4}     key_{i+1}[x] := key_i[x]
{ 5}     i := i-1
{ 6}   key_{i+1}[x]:=k
{ 7}   n[x] := n[x]+1
{ 8}   DISK-WRITE(x)
{ 9} else
{10}   while i>=1 and k<key_i[x] do
{11}     i := i-1
{12}   i:=i+1
{13}   DISK-READ(c_i[x])
{14}   if n[c_i[x]]=2*t-1 then
{15}     B-Tree-Split-Child(x,i,c_i[x])
{16}     if k > key_i[x] then
{17}       i := i+1
{18}   B-Tree-Insert-Nonfull(c_i[x],k)
```

B-tree deletion

- Deletion from a B-tree is analogous to insertion but a little more complicated, because a key may be deleted from any node-not just a leaf-and deletion from an internal node requires that the node's children be rearranged.
- As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (sometimes we need to merge nodes).
- Detailed rules and possible cases in the book Cormen et al. they use almost 2 pages of text. For possible independent reading and analysis.

B-tree summary

- Searching:
  - disk pages accesses: $O(h) = O(\lg_t n)$
  - CPU time: $O(th) = O(t \lg_t n)$
- Insertion:
  - disk pages accesses: $O(h) = O(\lg_t n)$
  - CPU time: $O(th) = O(t \lg_t n)$
- Deletion:
  - disk pages accesses: $O(h) = O(\lg_t n)$
  - CPU time: $O(th) = O(t \lg_t n)$
- More popular in database applications and file systems are B+trees, which are a special case of B-trees, storing data only in leaves.
  - Btrfs (B-tree File System) - a file system for Linux.