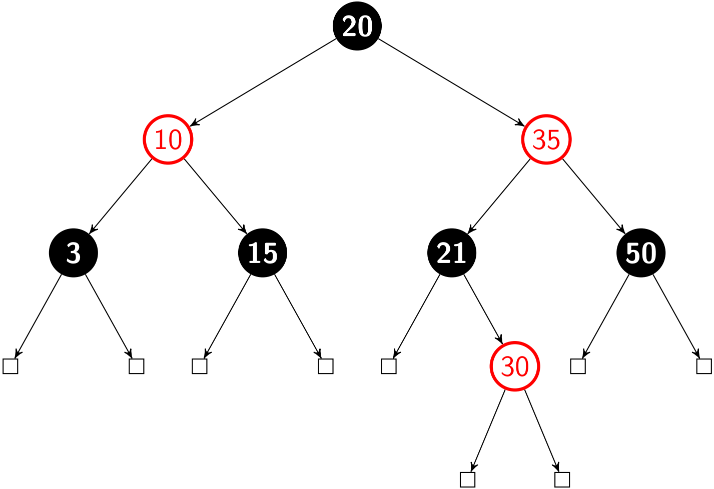
Balanced search trees.

DS 2018/2019

Red-black trees

- *Symmetric binary B-tree*, Rudolf Bayer, 1972.
- The balancing is maintained by using a coloring of the nodes.
- The red-black trees are binary search trees that satisfy the following properties:
  1. a node is colored with red or black;
  2. the root and the leaf nodes (*nil* – that belong to the structure) are colored with black;
  3. if a node is red, then his both children are black;
  4. the paths from a node to the boundary nodes have the same number of black nodes.

Red-black trees - example



Red-black trees

Lemma:
Any subtree of a red-black tree has at least $2^{bh(v)} - 1$ *internal nodes, where:*

- v *the root of the subtree,*
- $bh(v)$ *the number of black nodes found on a path from* v *down to a leaf, not including* v*;*

Proof.
At class. □

Theorem:

A red-black tree with n internal nodes has the height $h \leq 2\log_2(n+1)$.

Proof.

According to property 3,

$n \geq 2^{h/2} - 1 \quad \Rightarrow \quad h/2 \leq \log_2(n+1) \quad \Rightarrow \quad h \leq 2\log_2(n+1)$. $\square$

Red-black trees: operations

Corollary:

In a red-black tree with n nodes, the search operation has a time complexity of $O(\log n)$.

The insert operation

- Search for the insertion position and insert the new value as in the case of ordinary binary search trees.
- Color the new node with red.
- Restore the red-black properties by recoloring nodes and applying simple rotations.

The insert operation

- Property 1: satisfied.
- Property 2 – satisfied (both children of the inserted node are *nil*). If the inserted node is the root $\rightarrow$ recolor it in black.
- Property 4 – satisfied (the new red node replaces a leaf).
- The property 3 may not hold - if the parent of the node is red.
  - Move above this situation by recoloring the nodes until it can be repaired by rotation operations and recoloring.

The insert operation: the restauration of property 3

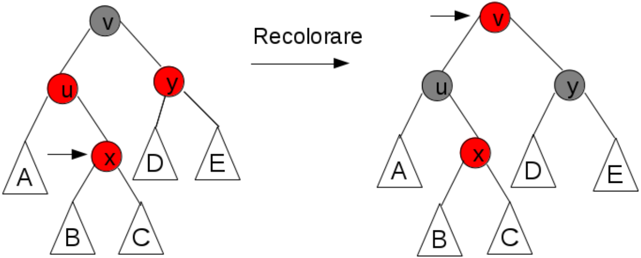
- **Case 1:** the "uncle" of the inserted node is red $\rightarrow$
  Recolor the "parent" and the "uncle" in black and the "grandfather" in red.

- **Case 2:** the "uncle" of the inserted node is black and the inserted node is the right child of a left child $\rightarrow$
  Apply a simple left rotation between the current node and the parent node.

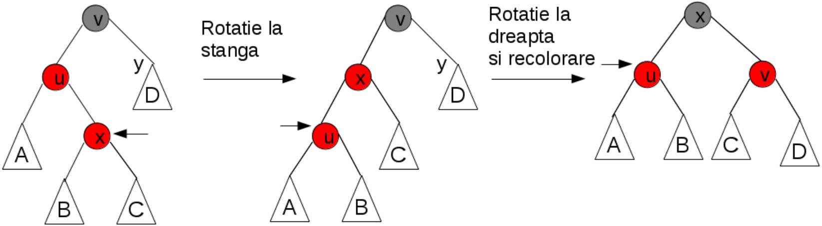
- **Case 3:** the "uncle" of the inserted node is black and the inserted node is the left child of a left child $\rightarrow$
  Apply a simple right rotation between the "parent" node and the "grandfather" node + recolor the "parent" node (in black) and the "grandfather" node (in red).

Obs.: Apply similar operations for the symmetric case.

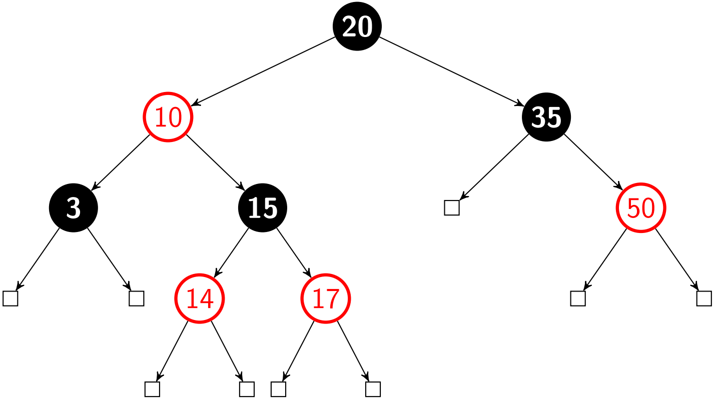
The insert operation - case 1



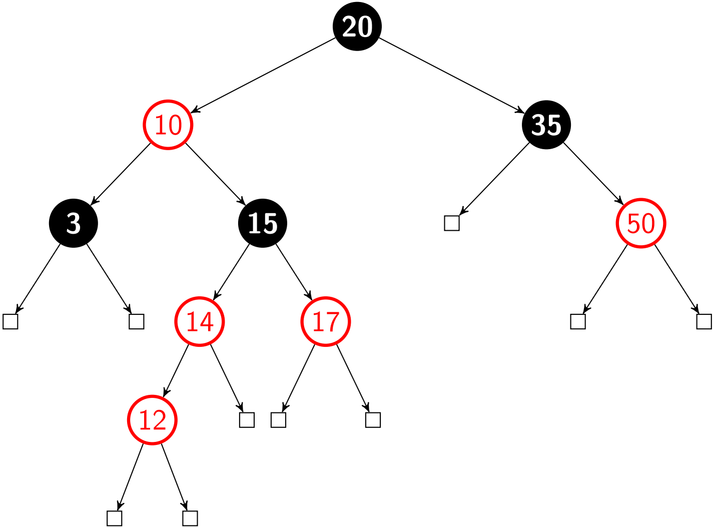
The insert operation - Case 2 and 3



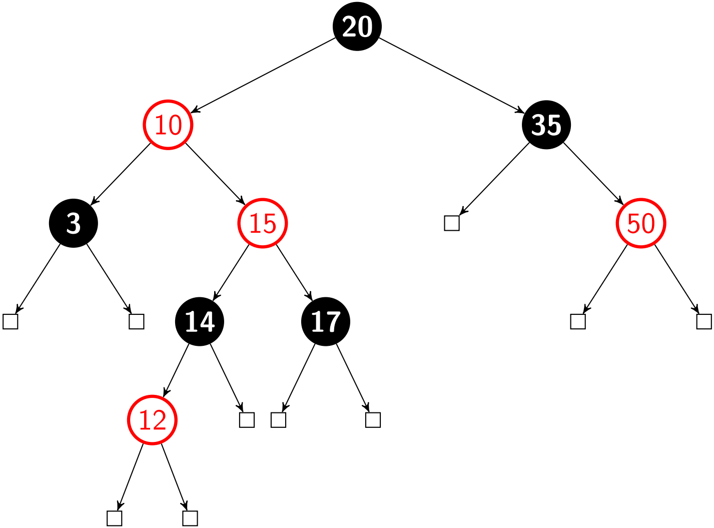
Insertion – example: node 12



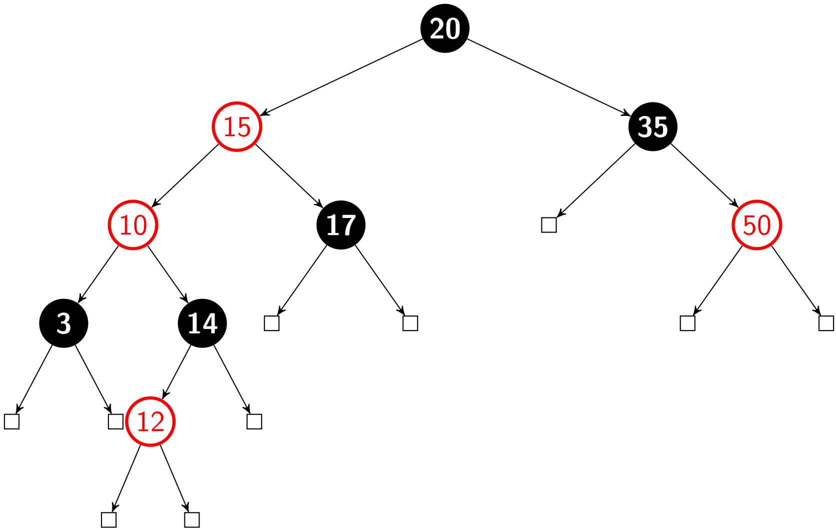
Insertion – CASE 1: recoloring



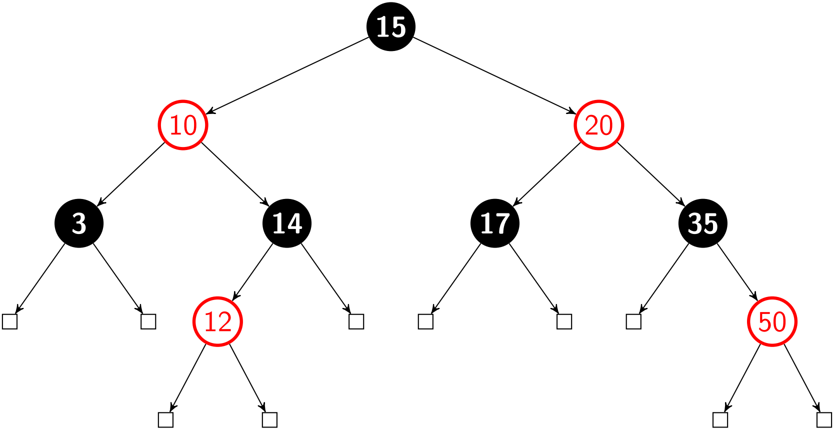
Insertion – CASE 2: left rotation



Insertion – CASE 3: right rotation + recoloring



Insertion – The valid red-black tree



The insert operation: algorithm

Consider that each node of the tree is a structure with the following fields:

- *key*: the useful information of the node;
- *color*: red / black;
- *pred*: the address of the parent node (null for the root);
- *left*: the address of the left child;
- *right*: the address of the right child.

The insert operation: algorithm

```
Procedure insert(t, x)
begin
    insBinarySearchTree(t, x)
    x → color ← red
    while (x! = t and x → pred → color == red) do
        if (x → pred == x → pred → pred → left) then
            y ← x → pred → pred → right
            if (y → color == red) then
                Case 1
            else
                if (x == x → pred → right) then
                    Case 2
                Case 3
        else
            similarly to the branch "then", but interchanging left with right
    t → color ← black
end
```

The insert operation: Case 1

$x \rightarrow pred \rightarrow color \leftarrow$ black
$y \rightarrow color \leftarrow$ black
$x \rightarrow pred \rightarrow pred \rightarrow color \leftarrow$ red
$x \leftarrow x \rightarrow pred \rightarrow pred$

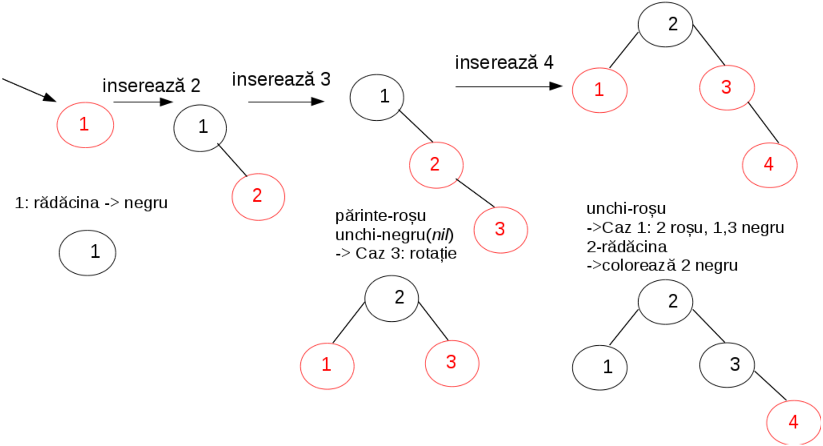
The insert operation: Case 2

$x \leftarrow x \rightarrow pred$
left-rotation(t, x)

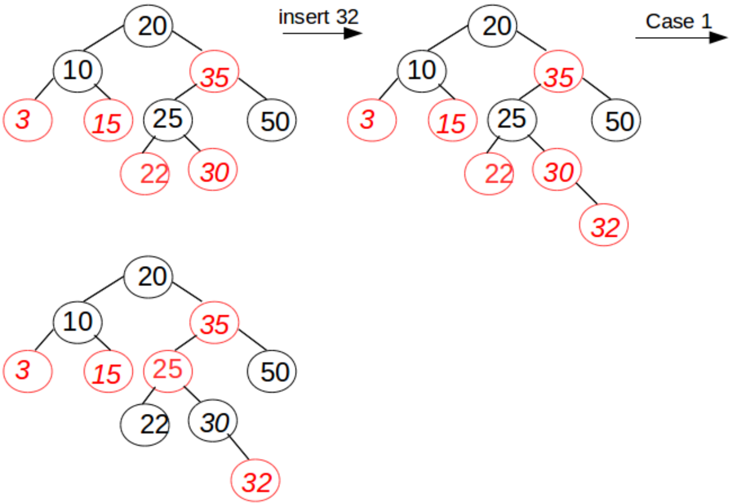
The insert operation: Case 3

$x \rightarrow pred \rightarrow color \leftarrow \text{black}$
$x \rightarrow pred \rightarrow pred \rightarrow color \leftarrow \text{red}$
$\text{right-rotation}(t, x \rightarrow pred \rightarrow pred)$

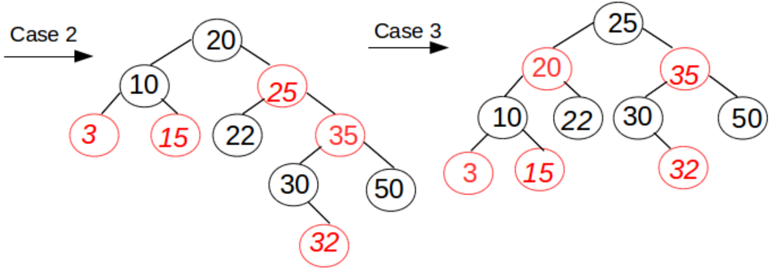
The insert operation - example 2



The insert operation - example 3



The insert operation - example 3



The delete operation

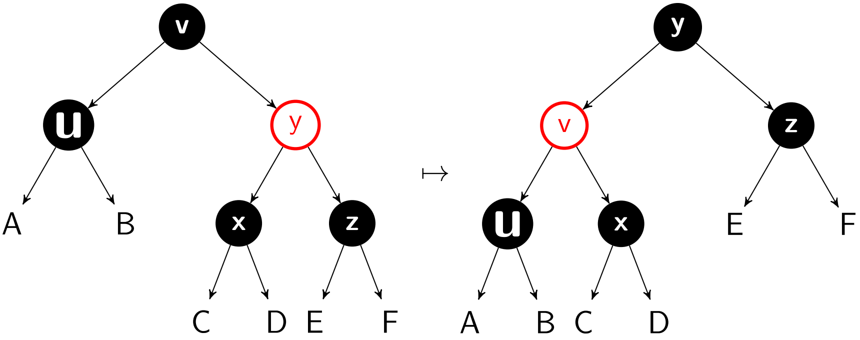
- Similarly to the delete operation from the ordinary binary search trees.
- Take into account that the "null" nodes belong to the structure.
- After deletion it is possible that the property 4 do not hold any more.
- Restore the properties of red-black trees by recoloring nodes and applying simple rotations.

The delete operation: the restoration of property 4

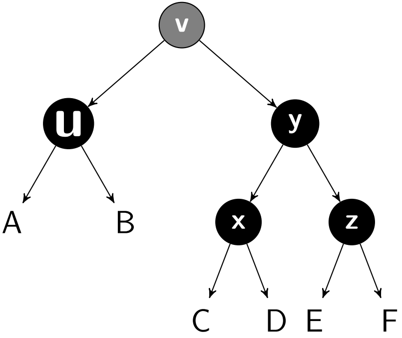
- **Case 1:** Transform it into one of the cases 2), 3), 4) by a rotation.
- **Case 2:** The node for which the property is not satisfied is moved to the root with one level by recoloring a node.
- **Case 3:** Transform it into case 4) by an interchanging of colors and a rotation.
- **Case 4:** In this case restore the red-black tree property for the whole tree.

Deletion – CASE 1

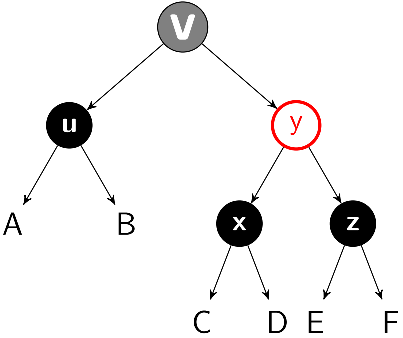
Caz 1: Transform it into one of the cases 2), 3), 4) by a rotation.



Case 2: The node for which the property is not satisfied is moved to the root with one level by recoloring a node.

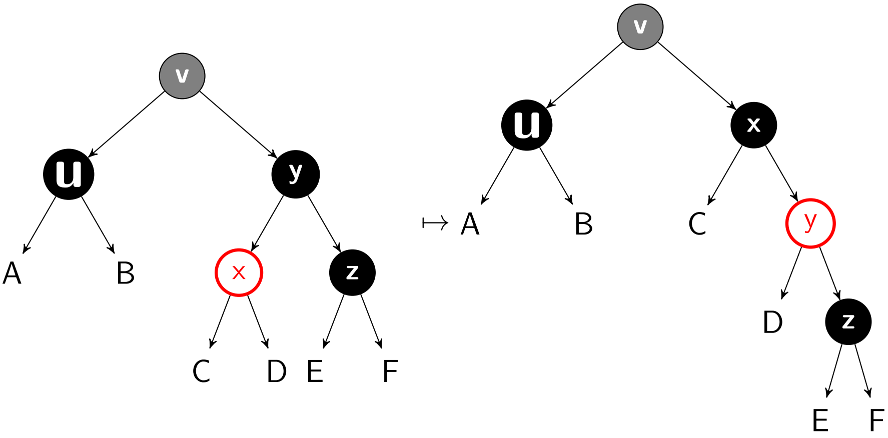


$\mapsto$



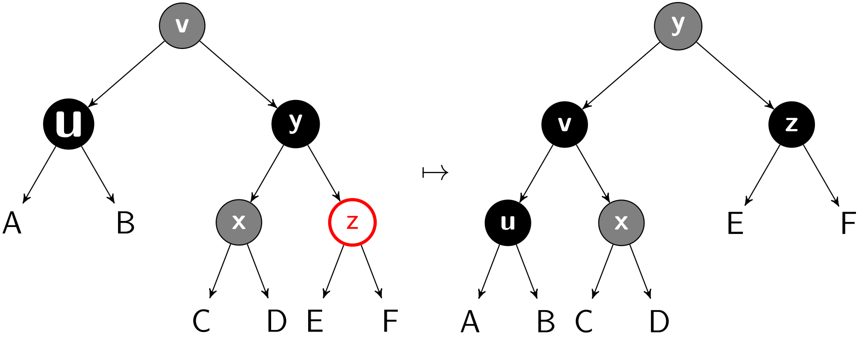
Deletion – CASE 3

Case 3: Transform it into case 4) by an interchanging of colors and a rotation.



Deletion – CASE 4

Caz 4: In this case restore the red-black tree property for the whole tree.



Red-black trees

- The complexity of insert / delete algorithms: $O(\log n)$.

Corollary:

The class of red-black trees is $O(\log n)$*–stable.*

Red-black trees

- The complexity of insert / delete algorithms: $O(\log n)$.

Corollary:

The class of red-black trees is $O(\log n)$–stable.

- Applications:
  - System symbol tables.
  - Kernel Linux (Completely Fair Scheduler).
  - Runway reservation system
  - Java: TreeMap, TreeSet; C++ STL: map, multimap, multiset