# Searching.

DS 2018/2019

# Content

# The search problem

- The static aspect:
  - $U$ the universe set, $S \subseteq U$
  - the search operation:
    - Instance: $a \in U$
    - Question: $a \in S$?
- The dynamic aspect:
  - the insert operation
    - Input: $S, \quad x \in U$
    - Output: $S \cup \{x\}$
  - the delete operation
    - Input: $S, \quad x \in U$
    - Output: $S - \{x\}$

# Searching in linear lists - complexity

| Data type | Implementation | Search | Insertion | Deletion |
|---|---|---|---|---|
| Linear list | Arrays | $O(n)$ | $O(1)$ | $O(n)$ |
| | Linked lists | $O(n)$ | $O(1)$ | $O(1)$ |
| Ordered list | Arrays | $O(\log n)$ | $O(n)$ | $O(n)$ |
| | Linked lists | $O(n)$ | $O(n)$ | $O(1)$ |

# Content

# Binary search: the static aspect

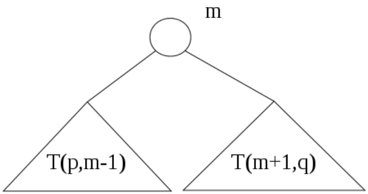
- The universe set is totally ordered: $(U, \leq)$

- The used data structure:
  - the array $s[0..n-1]$
  - $s[0] < ... < s[n-1]$

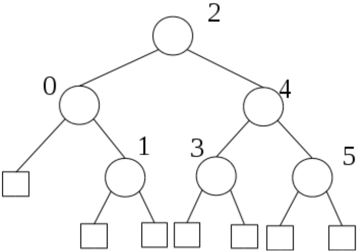
**Function** *pos*($s[0..n-1], n, a$)
**begin**
    $p \leftarrow 0; q \leftarrow n-1$
    $m \leftarrow (p+q)/2$
    **while** $(s[m]! = a \text{ and } p < q)$ **do**
        **if** $(a < s[m])$ **then**
            $q \leftarrow m-1$
        **else**
            $p \leftarrow m+1$
        $m \leftarrow (p+q)/2$
    **if** $(s[m] = a)$ **then**
        return $m$
    **else**
        return $-1$
**end**

# The binary tree associated to the binary search

$T(p, q)$




$T = T(0, n-1)$
$n = 6$



# Content

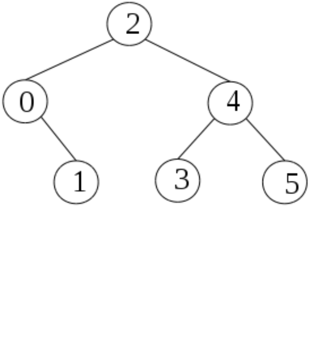
The set $S$ suffers update operations (insertion / deletion).

**The binary search tree**:

- In any node **v** it is stored a value from a totally ordered set.
- The values stored in the left subtree of **v** are lower than the value of **v**.
- The value of **v** is less than the values stored in the right subtree of **v**.

# Binary search trees

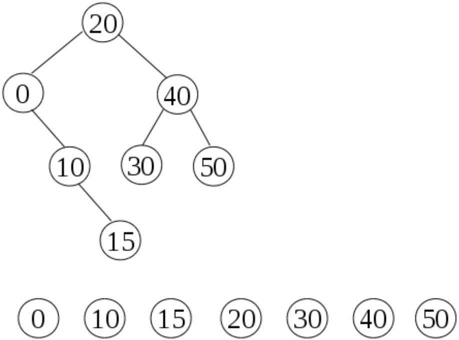
- The binary search tree associated with a key set is not unique.



# Binary search trees: sorting

- Inorder traversal

**Function** *inorder(v, visit)*
**begin**
    **if** $(x == NULL)$ **then**
        return
    **else**
        inorder($v \to stg$, visit)
        visit(v)
        inorder($v \to drp$, visit)
**end**




- Time complexity: $O(n)$

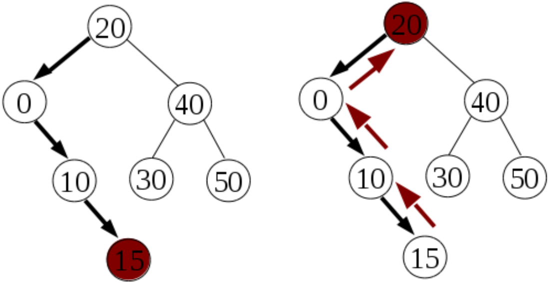
# Binary search trees: searching

**Function** *poz(t, x)*
**begin**
    $p \leftarrow t$
    **while** $(p! = NULL \ and \ p \rightarrow val! = x)$ **do**
        **if** $(x < p \rightarrow val)$ **then**
            $p \leftarrow p \rightarrow stg$
        **else**
            $p \leftarrow p \rightarrow drp$
    return $p$
**end**

- Time complexity: $O(h)$, $h$ the height

# Predecessor/Successor

- Modify the search operation: if the searched value $x$ is not in the tree, then return:
  - either the highest value $< x$ (predecessor),
  - either the smallest value $> x$ (successor).




the predecessor of 18

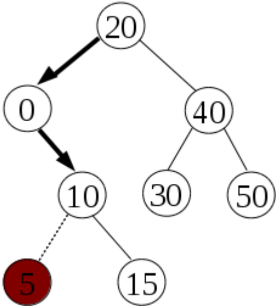
the successor of 18

# Successor

```
Function successor(t)
begin
    if (t → drp! = NULL) then
        /*min(t → drp)*/
        p ← t → drp
        while (p → stg! = NULL) do
            p ← p → stg
        return p
    else
        p ← t → pred
        while (p! = NULL and t == p → drp) do
            t ← p
            p ← p → pred
        return p
end
```

# Binary search trees: insertion

- Search in the tree the place to insert the new element (similarly with the search operation).
- Add the node with the new information, and the left subtree, respectively the right one is NULL.




Time complexity: $O(h)$, $h$ the height of the tree.

# Binary search trees: insertion

```
Procedure insBinarySearchTree(t, x)
begin
    if (t == NULL) then
        new(t); t → val ← x; t → stg ← NULL; t → drp ← NULL
    else
        p ← t
        while (p! = NULL) do
            predp ← p
            if (x < p → val) then p ← p → stg;
            else
                if (x > p → val) then p ← p → drp;
                else p ← NULL;
        if (predp → val! = x) then
            if (x < predp → val) then
                /* add x as left child of predp */
            else /* add x as right child of predp */ ;
end
```

# Binary search trees: elimination

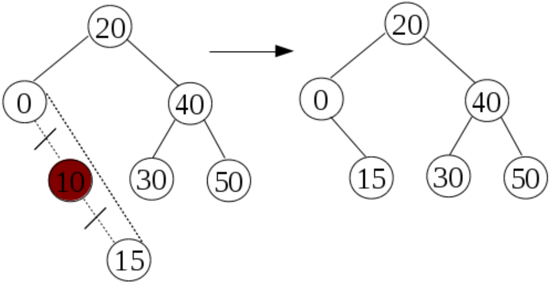
Search $x$ in the tree $t$; if it is found, then distinguish the following cases:

- Case 1: the node $p$ which stores $x$ has no children;
- Case 2: the node $p$ which stores $x$ has a single child;
- Case 3: the node $p$ which stores $x$ has both children.
  - Find the node $q$ which stores the highest value $y$ smaller than $x$ (get down from $p$ to the left and then to the right as much as possible).
  - Interchange the values from $p$ and $q$.
  - Delete $q$ as in case 1 or 2.

Time complexity: $O(h)$, $h$ the height.

# Binary search trees: elimination

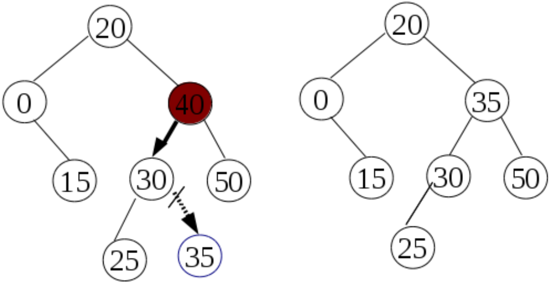
- Case 2. Example.



- Case 1 or 2

**Procedure** *elimCase1or2(t, predp, p)*
**begin**
    **if** $(p == t)$ **then**
        /* $t$ becomes void or */
        /* the only child of $t$ becomes the root */
    **else**
        **if** $(p \rightarrow stg == NULL)$ **then**
            /* replace in *predp*, $p$ with $p \rightarrow drp$ */
        **else**
            /* replace in *predp*, $p$ with $p \rightarrow stg$ */
**end**

- Case 3. Example.



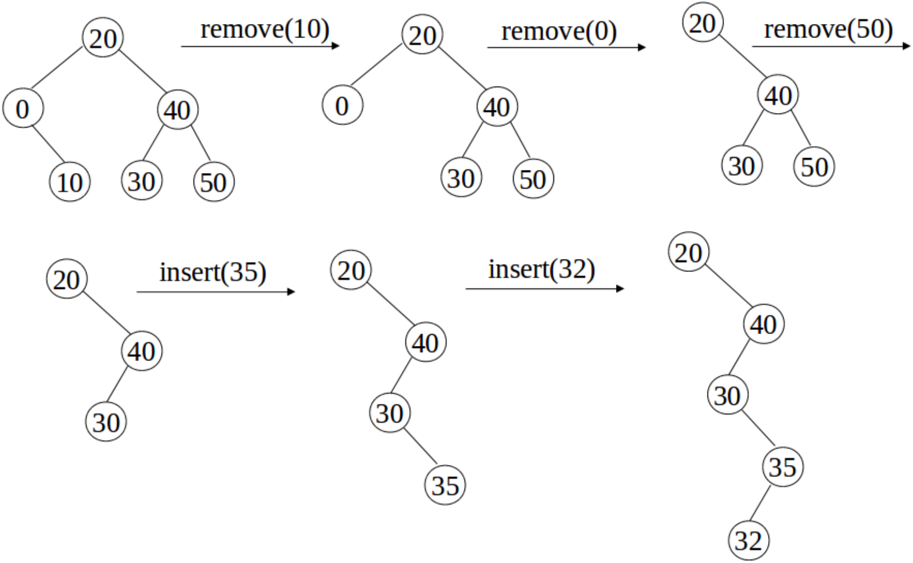
```
Procedure elimBinarySearchTree(t, x)
begin
    if (t! = NULL) then
        p ← t; predp ← NULL
        while (p! = NULL and p → val! = x) do
            predp ← p
            if (x < p → val) then p ← p → stg;
            else p ← p → drp;
        if (p! = NULL) then
            if (p → stg == NULL or p → drp == NULL) then
                elimCase1or2(t, predp, p)
            else
                q ← p → stg; predq ← p
                while (q → drp! = NULL) do
                    predq ← q; q ← q → drp
                p → val ← q → val
                elimCase1or2(t, predq, q)
end
```

# Binary search trees: analysis

Time complexity

- The worst case: $O(n)$, $n$ elements
- The average case: $O(logn)$

# The degeneration of binary search in linear search



# Content

# Balanced search trees

- AVL trees (Adelson-Velsii and Landis, 1962)
- B trees/2-3-4 trees (Bayer and McCreight, 1972)
- Red-black trees (Bayer, 1972)
- Splay Trees (Sleator and Tarjan, 1985)
- Treaps (Seidel and Aragon, 1996)

# Balanced search trees

- $\mathcal{C}$ is a class of balanced trees if
  for any tree $t$ with $n$ vertices from $\mathcal{C}$: $h(t) \leq c \log n$, $c$ constant.

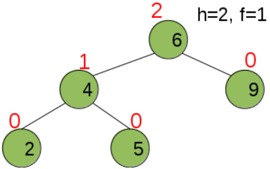
- $\mathcal{C}$ is a class of balanced trees $O(\log n)$-stable if
  there are algorithms for the operations of search, insertion, deletion in $O(\log n)$, and the resulted trees belong to class $\mathcal{C}$.

# AVL trees
# (G. **A**delson-**V**elskii, E.M. **L**andis 1962)

- A binary search tree $t$ is a balanced **AVL tree** if for each vertex $v$,

$$|h(v \to stg) - h(v \to drp)| \leq 1$$

- $h(v \to stg) - h(v \to drp)$ is called the **balancing factor**.
- Example:




- **Lemma**

  If $t$ is an AVL tree with $n$ internal nodes then $h(t) = \Theta(\log n)$.

  Proof. At class.

# AVL trees

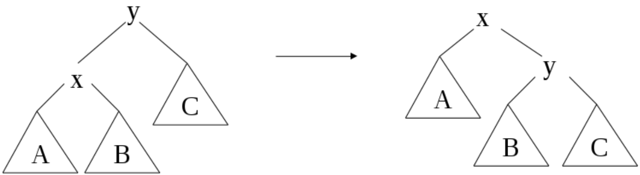
- **Theorem**

  The class of AVL trees is $O(\log n)$ stable.

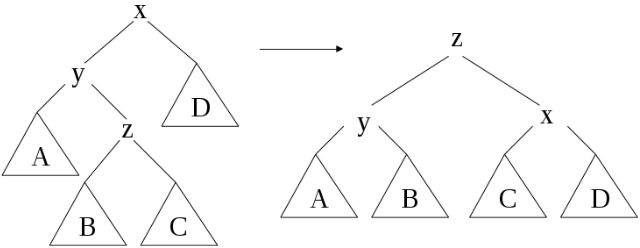
- The search/deletion algorithm
  - The nodes have also saved the balancing factors $(-1, 0, 1)$.
  - Store the path from the root to the added/deleted node in a stack ($O(\log n)$).
  - Traverse the path stored in the stack in reverse order and rebalance the unbalanced nodes with one of the operations: left/right rotation simple/double ($O(\log n)$).

# Rotations

Right rotation (simple)

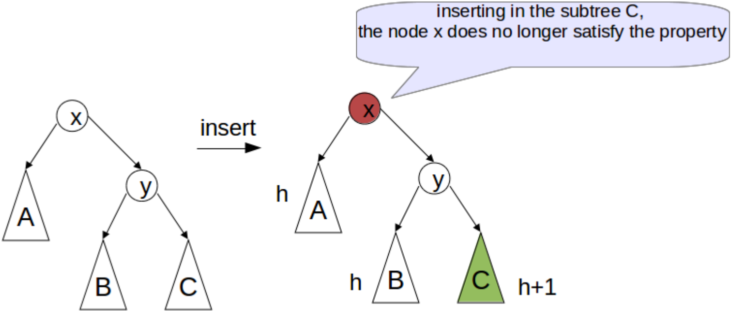



Double right rotation

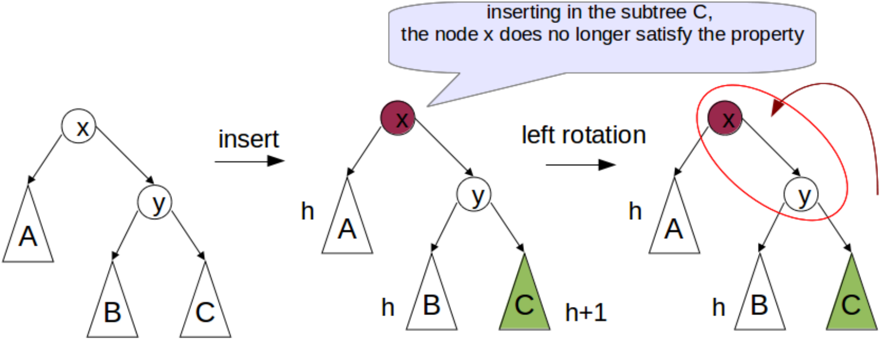



Similarly for simple left rotation, respectively for double left rotation.

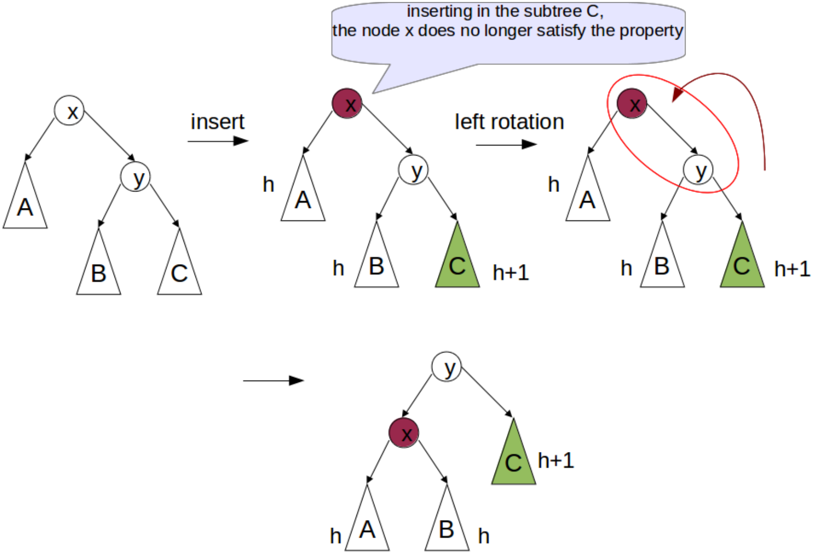
# Simple left rotation



# Simple left rotation (cont.)



# Simple left rotation (cont.)



# Simple left rotation

**Procedure** *leftRotation(x)*
**begin**

$y \leftarrow x \rightarrow drp$

$x \rightarrow drp \leftarrow y \rightarrow stg$

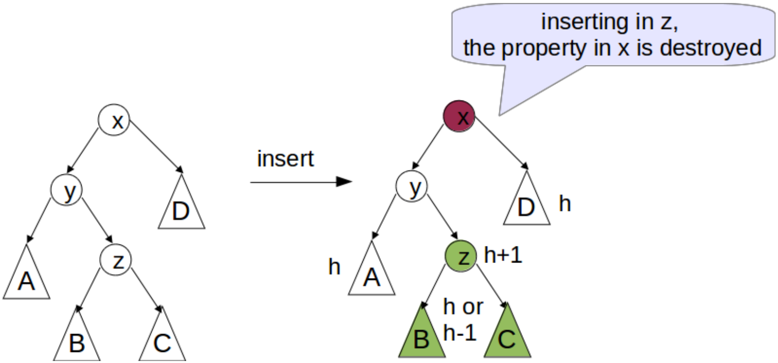
$y \rightarrow stg \leftarrow x$

return $y$

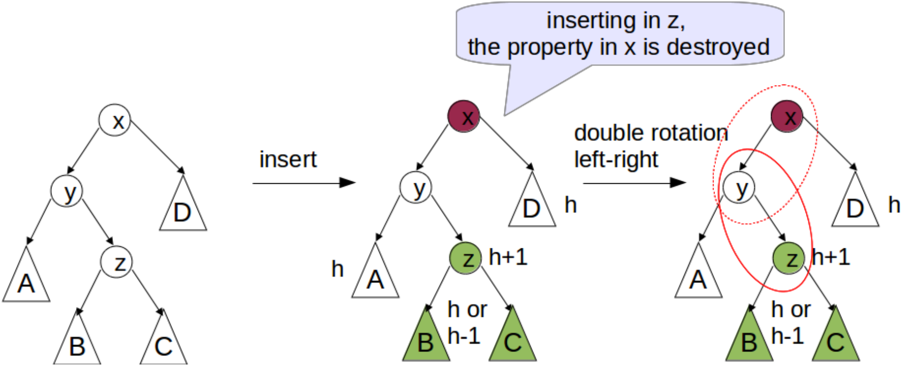
**end**

- Time complexity: $O(1)$

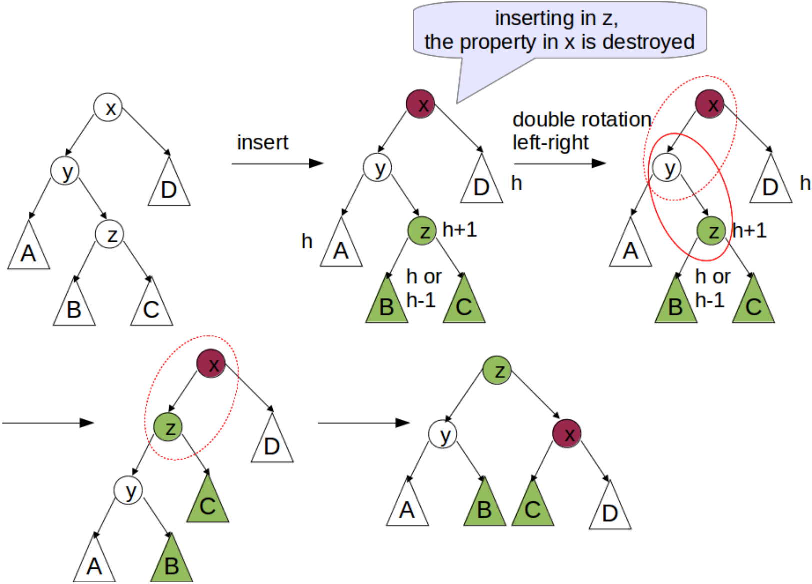
# Double rotation



# Double rotation (cont.)



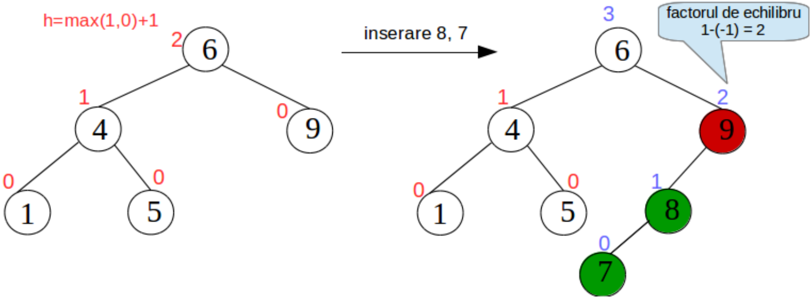
# Double rotation (cont.)



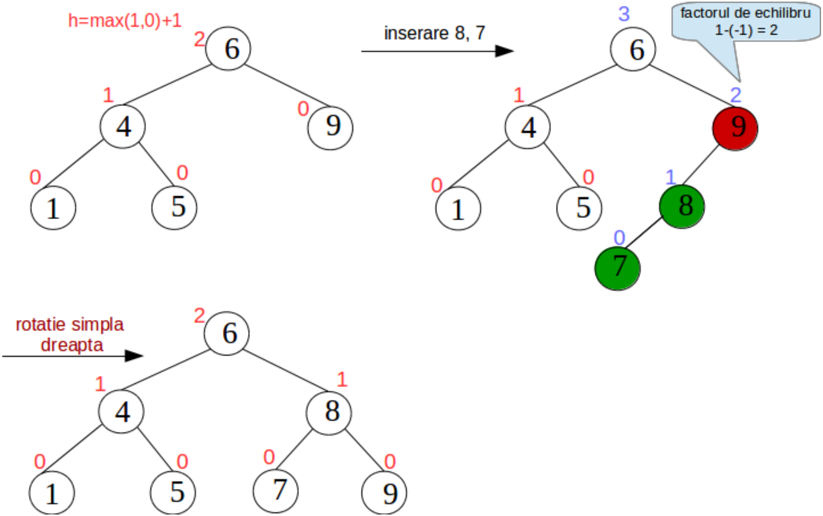
# Insertion: algorithm

```
Procedure balancing(t, x)
begin
    while (x! = NULL) do
        /* update the height h(x) */
        if (h(x → stg)) ≥ 2 + h(x → drp)) then
            if (h(x → stg → stg)) ≥ h(x → stg → drp)) then
                rightRotation(t, x)
            else
                leftRotation(t, x → stg); rightRotation(t, x)
        else
            if (h(x → drp)) ≥ 2 + h(x → stg)) then
                if (h(x → drp → drp)) ≥ h(x → drp → stg)) then
                    leftRotation(t, x)
                else
                    rightRotation(t, x → drp); leftRotation(t, x)
        x ← pred[x]
end
```

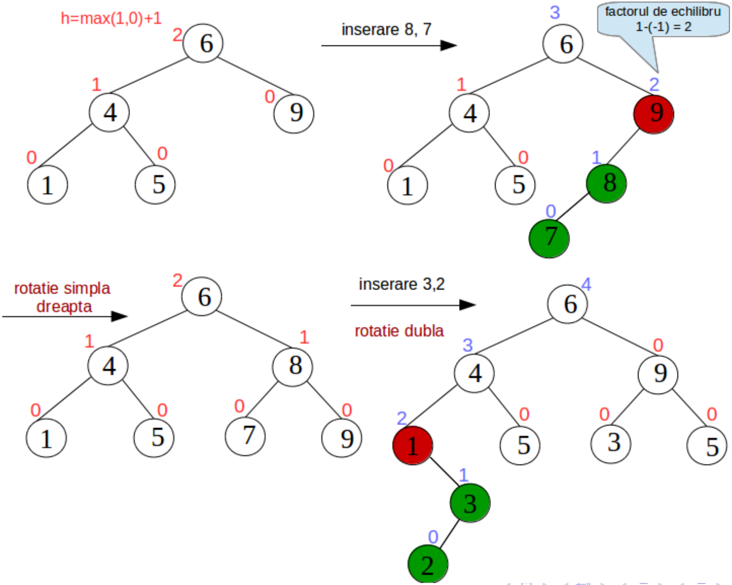
# Example: insertion



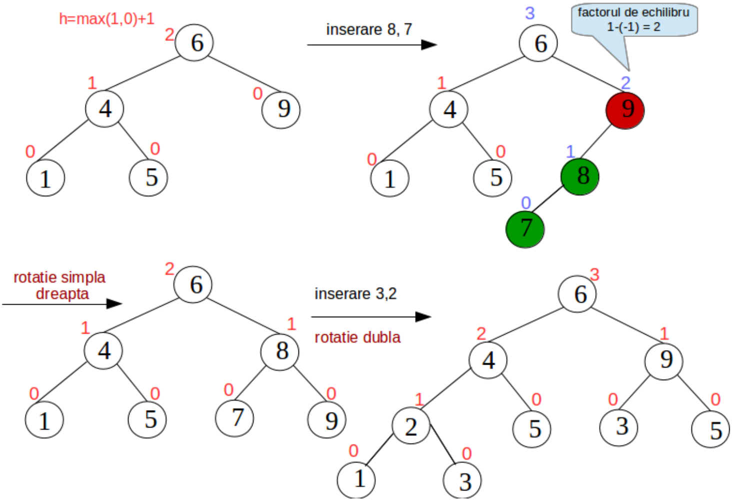
# Examplu: insertion (cont.)



# Example: insertion (cont.)



# Example: insertion (cont.)



# Advantages/drawbacks of AVL trees

- Advantages:
  - Searching, insertion and deletion takes $O(\log n)$ complexity.
- Drawbacks:
  - Additional space for storing the height / the balancing factor.
  - The re-balancing operations are expensive.
- Are favorite when we are making more searches and fewer insertions and deletions
- Applications in Data Analysis, Data Mining