Searching.

DS 2020/2021

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: $(U, \leq)$

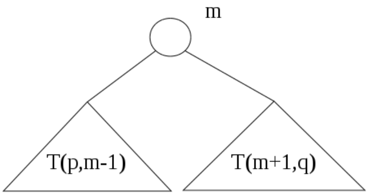
- The used data structure:
  - the array $s[0..n-1]$
  - $s[0] < ... < s[n-1]$

The binary search: the static aspect

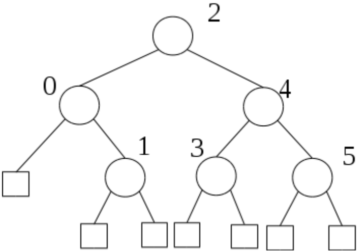
```
Function pos(s[0..n − 1], n, a)
begin
    p ← 0; q ← n − 1
    m ← (p + q)/2
    while (s[m]! = a and p < q) do
        if (a < s[m]) then
            q ← m − 1
        else
            p ← m + 1
        m ← (p + q)/2
    if (s[m] = a) then
        return m
    else
        return −1
end
```

The binary tree associated to the binary search

$T(p, q)$



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



Content

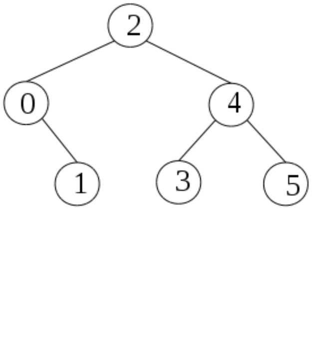
Binary search: the dynamic aspect

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

The binary search tree:

- In any node **v** a value from a totally ordered set is stored.
- 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**.

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

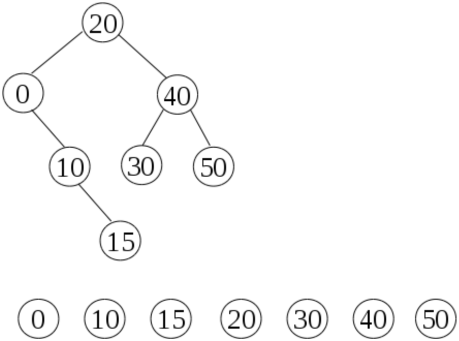


Binary search trees: sorting

- Inorder traversal

Function *inorder(v, visit)*
begin
  if $(v == NULL)$ **then**
    return
  else
    inorder($v \rightarrow left$, visit)
    visit(v)
    inorder($v \rightarrow right$, visit)
end

- Time complexity: $O(n)$



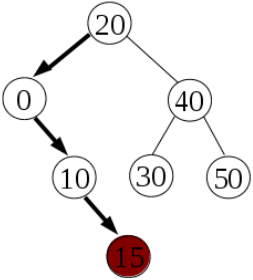
Binary search trees: searching

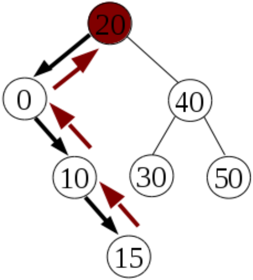
Function *pos*(t, x)
begin
    $p \leftarrow t$
    while $(p! = NULL \textit{ and } p \rightarrow val! = x)$ **do**
        if $(x < p \rightarrow val)$ **then**
            $p \leftarrow p \rightarrow left$
        else
            $p \leftarrow p \rightarrow right$
    return p
end

- 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).





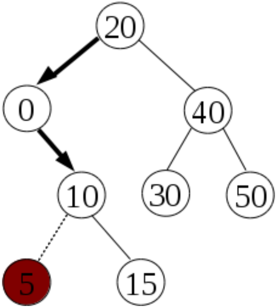
the predecessor of 18
the successor of 18

Successor

```
Function successor(t)
begin
    if (t → right != NULL) then
        /*min(t → right)*/
        p ← t → right
        while (p → left != NULL) do
            p ← p → left
        return p
    else
        p ← t → pred
        while (p != NULL and t == p → right) 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 insertBinarySearchTree(t, x)
begin
    if (t == NULL) then
        new(t); t → val ← x; t → left ← NULL; t → right ← NULL
    else
        p ← t
        while (p! = NULL) do
            predp ← p
            if (x < p → val) then p ← p → left;
            else
                if (x > p → val) then p ← p → right;
                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

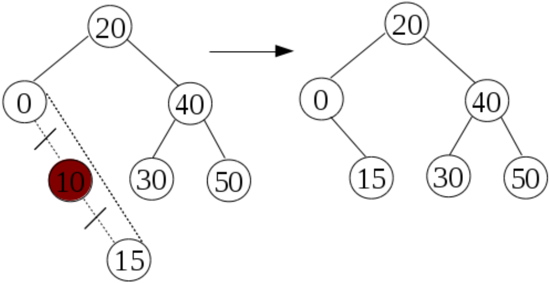
Search x in the tree t; if it is found, then distinguish the following cases:

- 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.

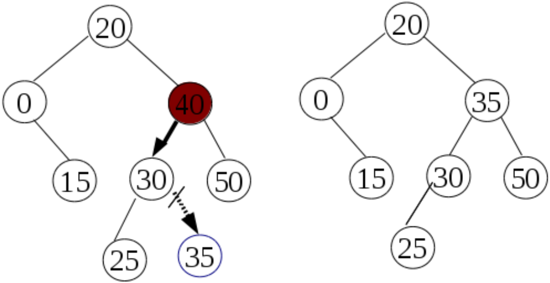


- Case 1 or 2

Procedure *eliminateCase1or2(t, predp, p)*
begin
    if $(p == t)$ **then**
        /* t becomes void or */
        /* the only child of t becomes the root */
    else
        if $(p \rightarrow left == NULL)$ **then**
            /* replace in *predp*, p with $p \rightarrow right$ */
        else
            /* replace in *predp*, p with $p \rightarrow left$ */
end

Binary search trees: elimination

- Case 3. Example.



Binary search trees: elimination

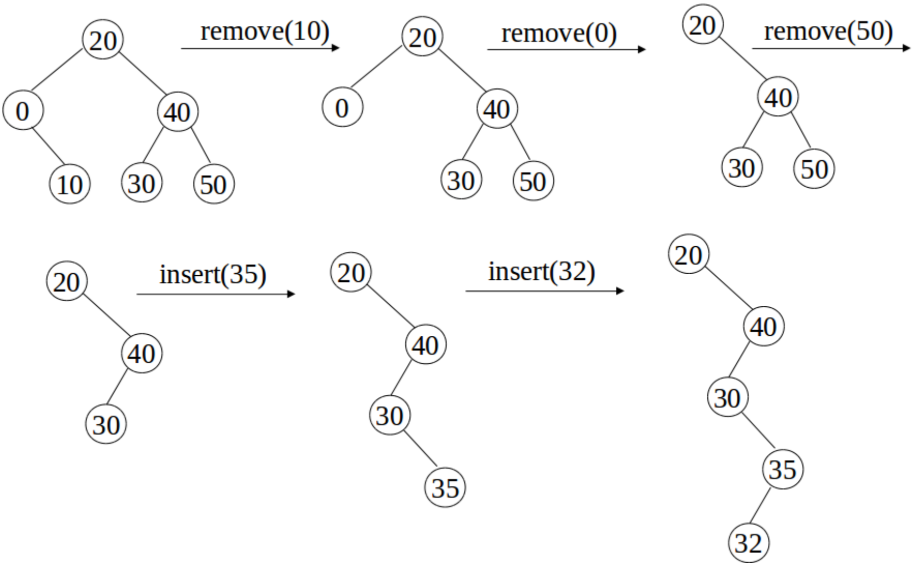
```
Procedure eliminateBinarySearchTree(t, x)
begin
    if (t != NULL) then
        p ← t; predp ← NULL
        while (p != NULL and p → val != x) do
            predp ← p
            if (x < p → val) then p ← p → left;
            else p ← p → right;
        if (p != NULL) then
            if (p → left == NULL or p → right == NULL) then
                eliminateCase1or2(t, predp, p)
            else
                q ← p → left; predq ← q
                while (q → right != NULL) do
                    predq ← q; q ← q → right
                p → val ← q → val
                eliminateCase1or2(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.

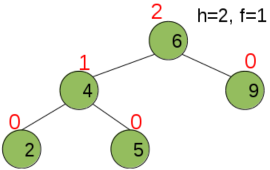
- $\mathcal{C}$ is a class of balanced trees $O(\log n)$-stable if
  there are algorithms for the operations of search, insertion, deletion in $O(\log n)$, and the resulted trees belong to class $\mathcal{C}$.

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,

$$|h(v \rightarrow left) - h(v \rightarrow right)| \leq 1$$

- $h(v \rightarrow left) - h(v \rightarrow right)$ is called the **balance factor**.
- Example:



- **Lemma**

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

Proof. At class.

AVL trees

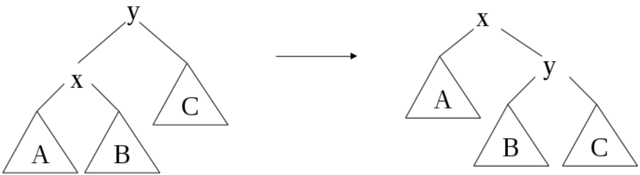
- **Theorem**

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

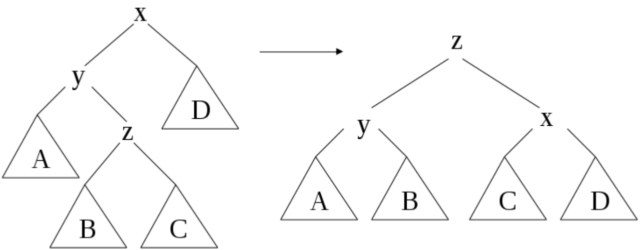
- The insertion/deletion algorithm
  - Save the balance factors for each node $(-1, 0, 1)$.
  - Store the path from the root to the added/deleted node in a stack $(O(\log n))$.
  - Traverse the path stored in the stack in reverse order and rebalance the unbalanced nodes with one of the operations: left/right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

Rotations

Right rotation (simple)

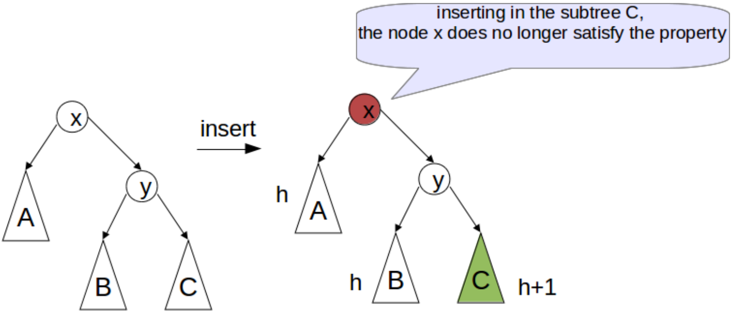


Double right rotation

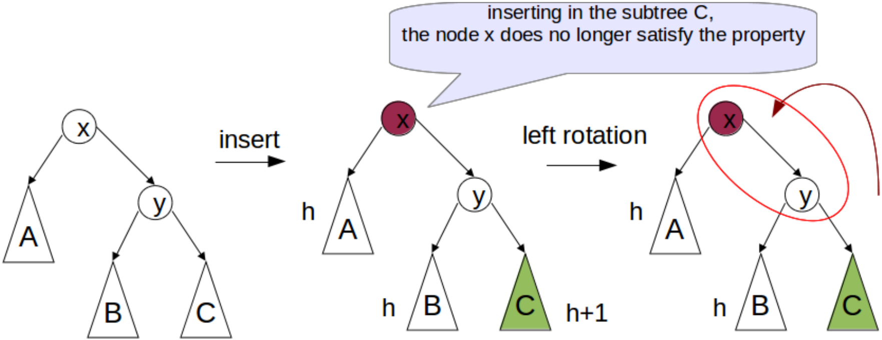


Similarly for simple left rotation, respectively for double left rotation.

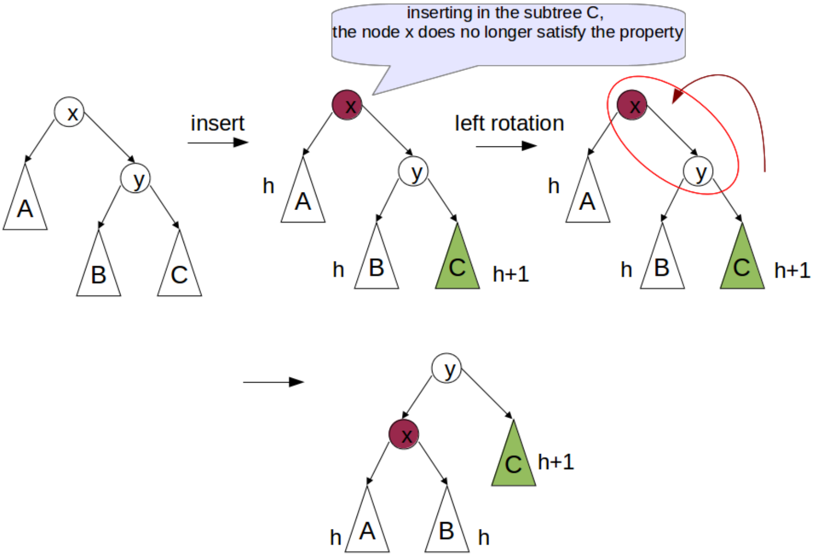
Simple left rotation



Simple left rotation (cont.)



Simple left rotation (cont.)



Simple left rotation

Procedure *leftRotation(x)*
begin

$y \leftarrow x \rightarrow right$

$x \rightarrow right \leftarrow y \rightarrow left$

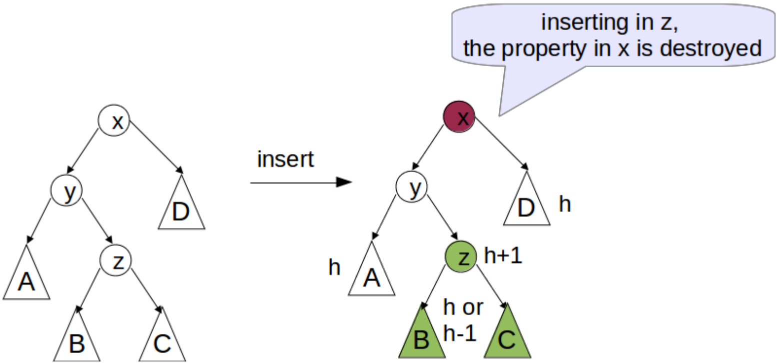
$y \rightarrow left \leftarrow x$

return y

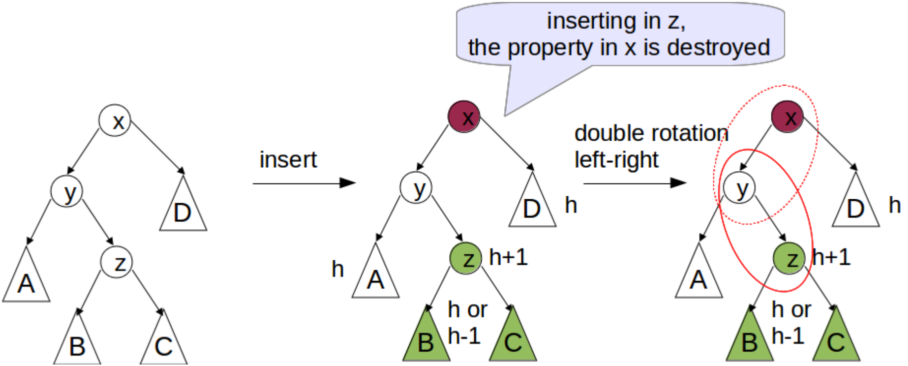
end

- Time complexity: $O(1)$

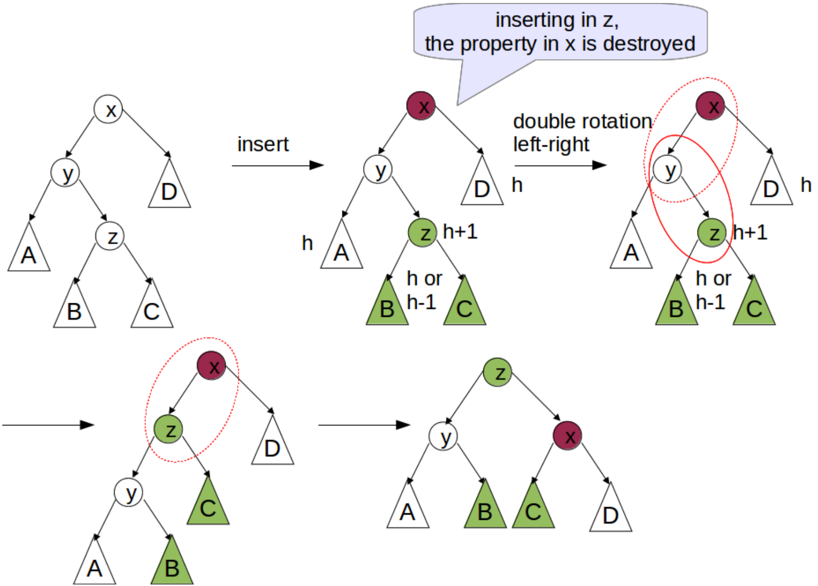
Double rotation



Double rotation (cont.)



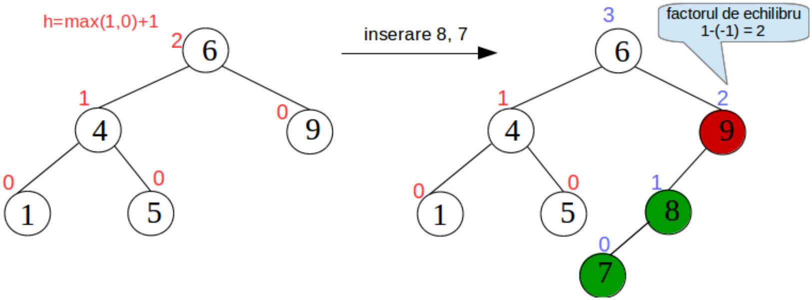
Double rotation (cont.)



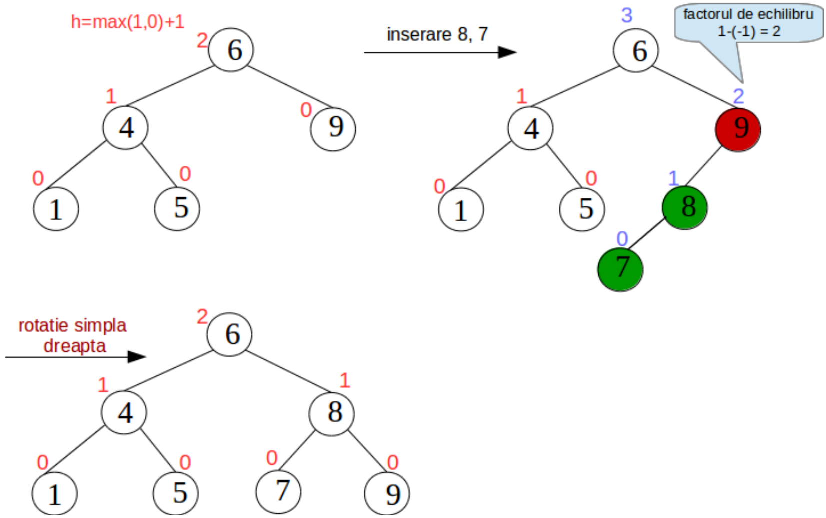
Insertion: algorithm

Procedure *balancing(t, x)*
begin
  while $(x! = NULL)$ **do**
    /* update the height $h(x)$ */
    if $(h(x \rightarrow left)) \geq 2 + h(x \rightarrow right))$ **then**
      if $(h(x \rightarrow left \rightarrow left)) \geq h(x \rightarrow left \rightarrow right))$ **then**
        $rightRotation(t, x)$
      else
        $leftRotation(t, x \rightarrow left)$; $rightRotation(t, x)$
    else
      if $(h(x \rightarrow right)) \geq 2 + h(x \rightarrow left))$ **then**
        if $(h(x \rightarrow right \rightarrow right)) \geq h(x \rightarrow right \rightarrow left))$ **then**
          $leftRotation(t, x)$
        else
          $rightRotation(t, x \rightarrow right)$; $leftRotation(t, x)$
    $x \leftarrow x \rightarrow pred$
end

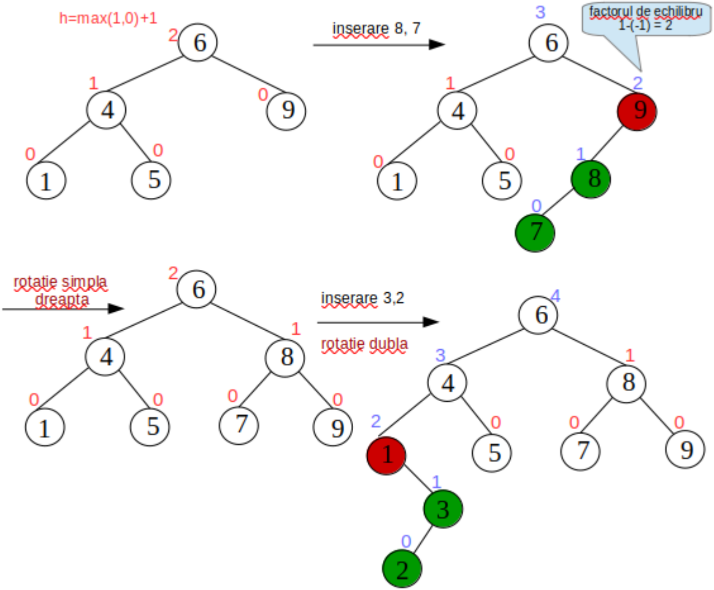
Example: insertion



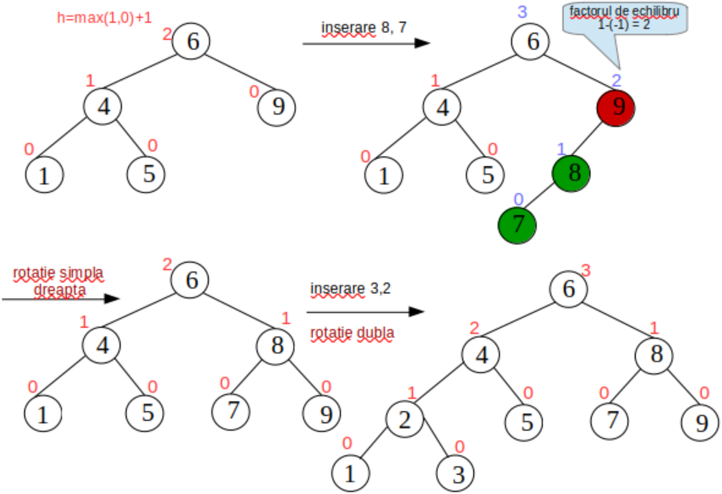
Example: insertion (cont.)



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 balance 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