DATA STRUCTURES AND ALGORITHMS

LECTURE 13

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 12...

- Binary Tree Traversals
- Huffman coding
- Binary Search Tree

Today

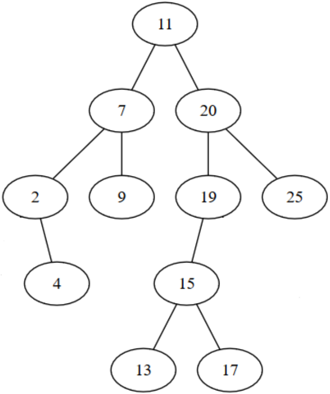
1. Binary Search Trees

2. AVL Trees

Binary search trees

- A *Binary Search Tree* is a binary tree that satisfies the following property:
  - if x is a node of the binary search tree then:
    - For every node y from the left subtree of x, the information from y is less than or equal to the information from x
    - For every node y from the right subtree of x, the information from y is greater than or equal to the information from x
- In order to have a binary search tree, we need to store information in the tree that is of type *TComp*.
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having "$\leq$" as in the definition).

Binary Search Tree Example



- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used).

Binary Search Tree

- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.
- In order to implement these containers on a binary search tree, we need to define the following basic operations:
  - search for an element
  - insert an element
  - remove an element
- Other operations that can be implemented for binary search trees (and can be used by the containers): get minimum/maximum element, find the successor/predecessor of an element.

Binary Search Tree - Representation

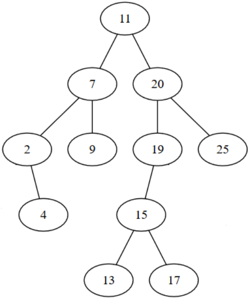
- We will use a linked representation with dynamic allocation (similar to what we used for binary trees)
- We will assume that the info is of type TComp and use the relation "$\leq$"

<u>BSTNode:</u>
  info: TComp
  left: $\uparrow$ BSTNode
  right: $\uparrow$ BSTNode

<u>BinarySearchTree:</u>
  root: $\uparrow$ BSTNode

Binary Search Tree - search operation

- How can we search for an element in a binary search tree?



- How can we search for element 15? And for element 14?

BST - search operation - recursive implementation

- How can we implement the *search algorit*hm recursively?

BST - search operation - recursive implementation

- How can we implement the *search algorithm* recursively?

function search_rec (node, elem) **is:**
//pre: node is a BSTNode and elem is the TComp we are searching for
    if node = NIL **then**
        search_rec ← false
    else
        if [node].info = elem **then**
            search_rec ← true
        else if [node].info < elem **then**
            search_rec ← search_rec([node].right, elem)
        else
            search_rec ← search_rec([node].left, elem)
    end-if
end-function

BST - search operation - recursive implementation

- Complexity of the search algorithm:

BST - search operation - recursive implementation

- Complexity of the search algorithm: $O(h)$ (which is $O(n)$)
- Since the *search* algorithm takes as parameter a node, we need a wrapper function to call it with the root of the tree.

function search (tree, e) **is:**
//pre: tree is a BinarySearchTree, e is the elem we are looking for
  search ← search_rec(tree.root, e)
end-function

BST - search operation - non-recursive implementation

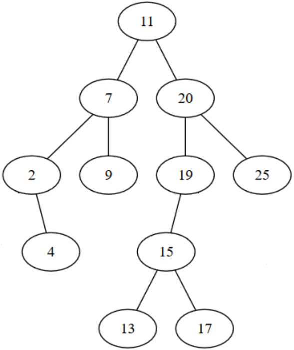
- How can we define the search operation non-recursively?

BST - search operation - non-recursive implementation

- How can we define the search operation non-recursively?

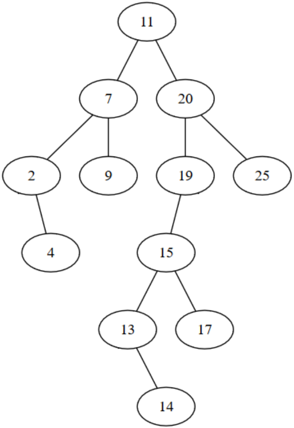
function search (tree, elem) **is:**
//pre: tree is a BinarySearchTree and elem is the TComp we are searching for
    currentNode ← tree.root
    found ← false
    while currentNode ≠ NIL and not found **execute**
        if [currentNode].info = elem **then**
            found ← true
        else if [currentNode].info < elem **then**
            currentNode ← [currentNode].right
        else
            currentNode ← [currentNode].left
        end-if
    end-while
    search ← found
end-function

BST - insert operation



- How/Where can we insert element 14?

BST - insert operation



BST - insert operation - recursive implementation

- How can we implement the *insert* operation?

BST - insert operation - recursive implementation

- How can we implement the *insert* operation?
- We will start with a function that creates a new node given the information to be stored in it.

function initNode(e) **is:**
//pre: e is a TComp
//post: initNode ← a node with e as information
    allocate(node)
    [node].info ← e
    [node].left ← NIL
    [node].right ← NIL
    initNode ← node
end-function

BST - insert operation - recursive implementation

function insert_rec(node, e) **is:**
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
  if node = NIL **then**
    node ← initNode(e)
  else if [node].info ≥ e **then**
    [node].left ← insert_rec([node].left, e)
  else
    [node].right ← insert_rec([node].right, e)
  end-if
  insert_rec ← node
end-function

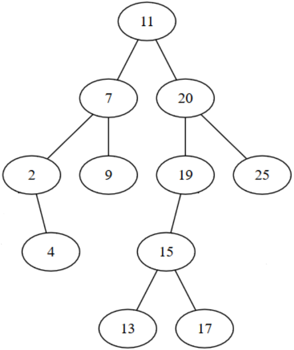
- Complexity:

BST - insert operation - recursive implementation

function insert_rec(node, e) **is:**
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
    if node = NIL **then**
        node ← initNode(e)
    else if [node].info ≥ e **then**
        [node].left ← insert_rec([node].left, e)
    else
        [node].right ← insert_rec([node].right, e)
    end-if
    insert_rec ← node
end-function

- Complexity: O(n)
- Like in case of the *search* operation, we need a wrapper function to call *insert_rec* with the root of the tree.

BST - Finding the minimum element



- How can we find the minimum element of the binary search tree?

BST - Finding the minimum element

```
function minimum(tree) is:
//pre: tree is a BinarySearchTree
//post: minimum ← the minimum value from the tree
   currentNode ← tree.root
   if currentNode = NIL then
      @empty tree, no minimum
   else
      while [currentNode].left ≠ NIL execute
         currentNode ← [currentNode].left
      end-while
      minimum ← [currentNode].info
   end-if
end-function
```

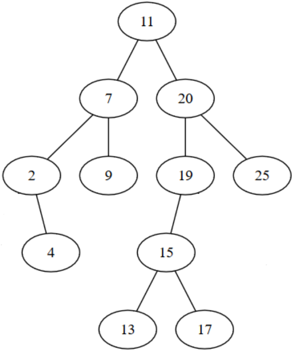
BST - Finding the minimum element

- Complexity of the minimum operation:

BST - Finding the minimum element

- Complexity of the minimum operation: $O(n)$
- We can have an implementation for the minimum, when we want to find the minimum element of a subtree, in this case the parameter to the function would be a node, not a tree.
- We can have an implementation where we return the node containing the minimum element, instead of just the value (depends on what we want to do with the operation)
- Maximum element of the tree can be found similarly.

Finding the parent of a node



- Given a node, how can we find the parent of the node? (assume a representation where the node has no parent field).

Finding the parent of a node

```
function parent(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node ≠ NIL
//post: returns the parent of node, or NIL if node is the root
   c ← tree.root
   if c = node then //node is the root
      parent ← NIL
   else
      while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute
         if [c].info ≥ [node].info then
            c ← [c].left
         else
            c ← [c].right
         end-if
      end-while
      parent ← c
   end-if
end-function
```

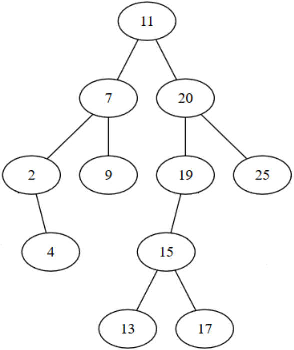
- Complexity:

Finding the parent of a node

function parent(tree, node) **is:**
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the parent of node, or NIL if node is the root
  c ← tree.root
  if c = node **then** *//node is the root*
    parent ← NIL
  else
    while c $\neq$ NIL **and** [c].left $\neq$ node **and** [c].right $\neq$ node **execute**
      if [c].info $\geq$ [node].info **then**
        c ← [c].left
      else
        c ← [c].right
      end-if
    end-while
    parent ← c
  end-if
end-function

- Complexity: $O(n)$

BST - Finding the successor of a node



- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11? After 17? After 13?

BST - Finding the successor of a node

function successor(tree, node) **is:**
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the node with the next value after the value from node
//or NIL if node is the maximum
    if [node].right $\neq$ NIL **then**
        c $\leftarrow$ [node].right
        while [c].left $\neq$ NIL **execute**
            c $\leftarrow$ [c].left
        end-while
        successor $\leftarrow$ c
    else
        p $\leftarrow$ parent(tree, c)
        while p $\neq$ NIL **and** [p].left $\neq$ c **execute**
            c $\leftarrow$ p
            p $\leftarrow$ parent(tree, p)
        end-while
        successor $\leftarrow$ p
    end-if
end-function

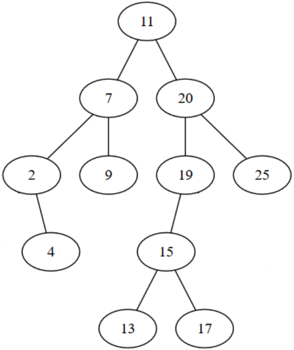
BST - Finding the successor of a node

- Complexity of successor:

BST - Finding the successor of a node

- Complexity of successor: depends on parent function:
  - If *parent* is $\Theta(1)$, complexity of successor is $O(n)$
  - If *parent* is $O(n)$, complexity of successor is $O(n^2)$
- What if, instead of receiving a node, successor algorithm receives as parameter a value from a node (assume unique values in the nodes)? How can we find the successor then?
- Similar to successor, we can define a predecessor function as well.

BST - Remove a node



- How can we remove the value 25? And value 2? And value 11?

BST - Remove a node

- When we want to remove a value (a node containing the value) from a binary search tree we have three cases:
  - The node to be removed has no descendant
    - Set the corresponding child of the parent to NIL
  - The node to be removed has one descendant
    - Set the corresponding child of the parent to the descendant
  - The node to be removed has two descendants
    - Find the maximum of the left subtree, move it to the node to be deleted, and delete the maximum
    OR
    - Find the minimum of the right subtree, move it to the node to be deleted, and delete the minimum

BST - Remove a node

function removeRec(node, elem) **is**
//pre: node is a pointer to a BSTreeNode and elem is the value we remove
//post: the node with value elem was removed from the (sub)tree that starts
//with node
  if node = NIL **then**
    removeRec ← NIL
  else if [node].info > elem **then**
    [node].left ← removeRec([node].left, elem)
    removeRec ← node
  else if [node].info < elem **then**
    [node].right ← removeRec([node].right, elem)
    removeRec ← node
  else *//[node].info = elem, we want to remove node*
//continued on the next slide...

```
        if [node].left = NIL and [node].right = NIL then
            removeRec ← NIL
        else if [node].left = NIL then
            removeRec ← [node].right
        else if [node].right = NIL then
            removeRec ← [node].left
        else
            min ← minimum([node].right)
            [node].info ← [min].info
            [node].right ← removeRec([node].right, [min].info)
            removeRec ← node
        end-if
    end-if
end-function
```

- We assume that *minimum* returns a node with the minimum, not just the value.
- Complexity:

```
    if [node].left = NIL and [node].right = NIL then
        removeRec ← NIL
    else if [node].left = NIL then
        removeRec ← [node].right
    else if [node].right = NIL then
        removeRec ← [node].left
    else
        min ← minimum([node].right)
        [node].info ← [min].info
        [node].right ← removeRec([node].right, [min].info)
        removeRec ← node
    end-if
  end-if
end-function
```

- We assume that *minimum* returns a node with the minimum, not just the value.
- Complexity: $O(n)$

Binary Search Tree

- Think about it:
  - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?

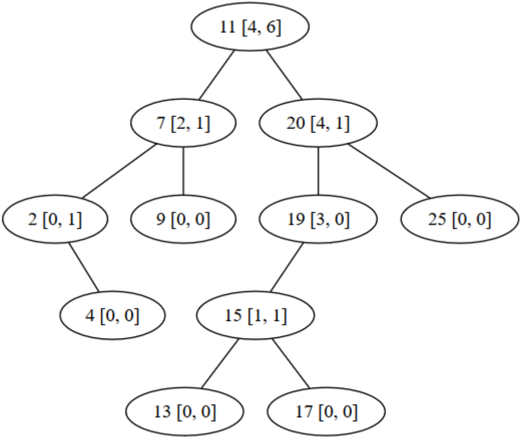
Binary Search Tree

- Think about it:
  - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
  - Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?

Binary Search Tree

- Think about it:
  - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
  - Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?
  - Hint: Keep in each node the number of nodes in the left subtree and the number of nodes in the right subtree.

Binary Search Tree



- Obviously, these values have to be modified when we add/remove an element.

Balanced Binary Search Trees

- Specific operations for binary trees run in $O(h)$ time, which can be $O(n)$ in worst case
- Best case is a balanced tree, where height of the tree is $O(log_2 n)$

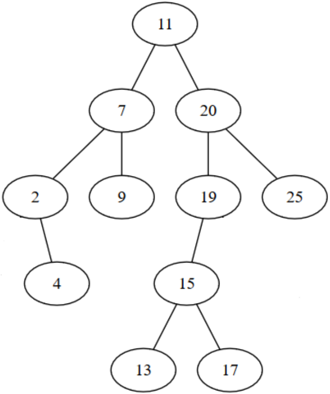
Balanced Binary Search Trees

- Specific operations for binary trees run in $O(h)$ time, which can be $O(n)$ in worst case
- Best case is a balanced tree, where height of the tree is $O(log_2 n)$
- To reduce the complexity of algorithms, we want to keep the tree balanced. In order to do this, we want every node to be balanced.
- When a node loses its balance, we will perform some operations (called rotations) to make it balanced again.

AVL Trees

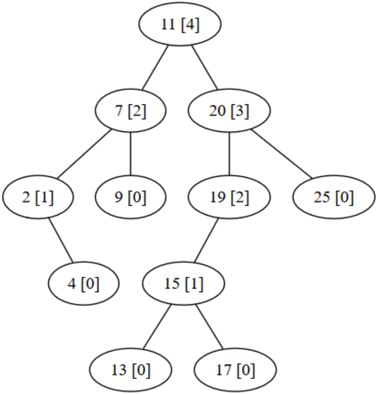
- Definition: An AVL (Adelson-Velskii Landis) tree is a binary tree which satisfies the following property (AVL tree property):
    - If x is a node of the AVL tree:
        - the difference between the height of the left and right subtree of x is 0, 1 or -1 (balancing information)
- Observations:
- Height of an empty tree is -1
- Height of a single node is 0

AVL Trees



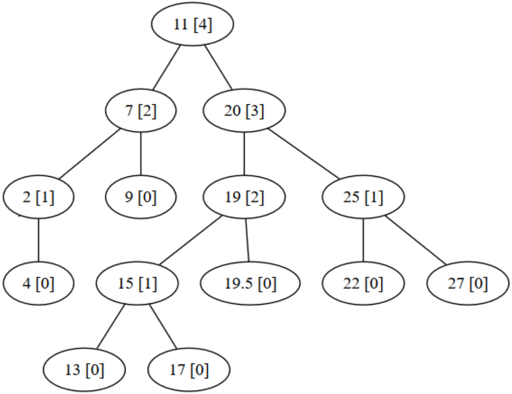
- Is this an AVL tree?

AVL Trees



- Values in square brackets show the height of a node. The tree is not an AVL tree, because the difference between the heights of the left and right subtree for nodes 19 and 20 is 2.

AVL Trees



- This is an AVL tree.

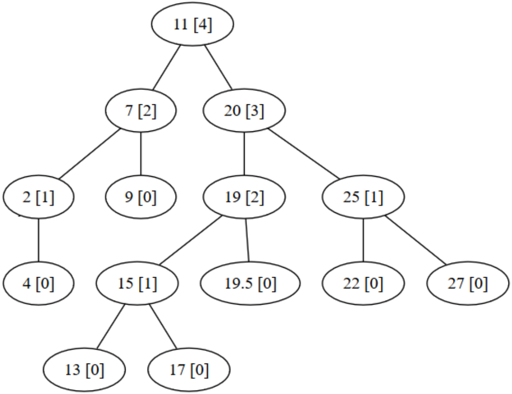
AVL Trees - rotations

- Adding or removing a node might result in a binary tree that violates the AVL tree property.
- In such cases, the property has to be restored and only after the property holds again is the operation (add or remove) considered finished.
- The AVL tree property can be restored with operations called **rotations**.

AVL Trees - rotations

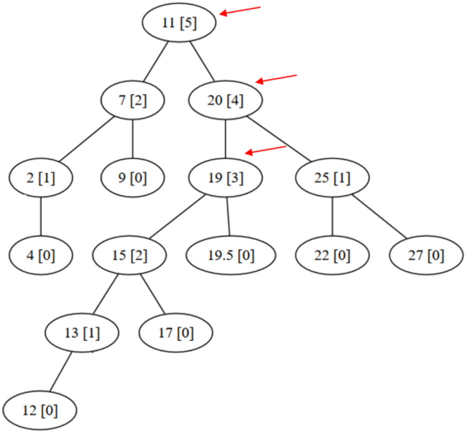
- After an insertion, only the nodes one the path to the modified node can change their height.
- We check the balancing information on the path from the modified node to the root. When we find a node that does not respect the AVL tree property, we perform a suitable *rotation* to rebalance the (sub)tree.

AVL Tress - rotations



- What if we insert element 12?

AVL Trees - rotations

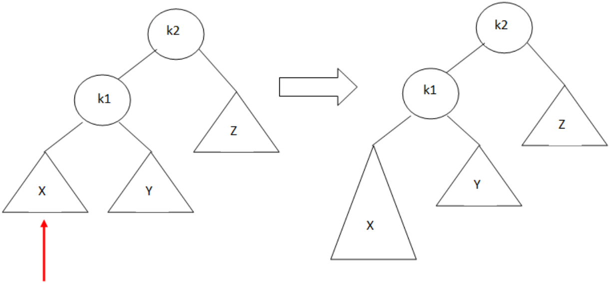


- Red arrows show the unbalanced nodes. We will rebalance node 19.

AVL Trees - rotations

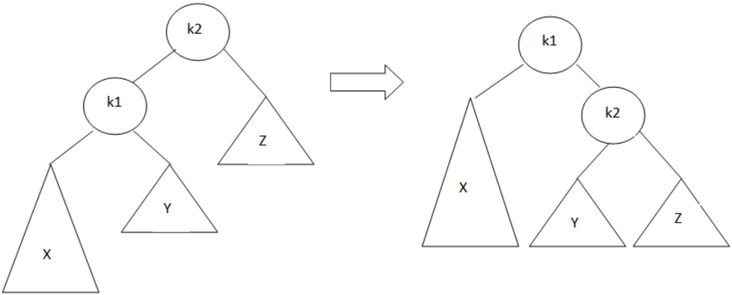
- Assume that at a given point α is the node that needs to be rebalanced.
- Since α was balanced before the insertion, and is not after the insertion, we can identify four cases in which a violation might occur:
  - Insertion into the left subtree of the left child of α
  - Insertion into the right subtree of the left child of α
  - Insertion into the left subtree of the right child of α
  - Insertion into the right subtree of the right child of α

AVL Trees - rotations - case 1

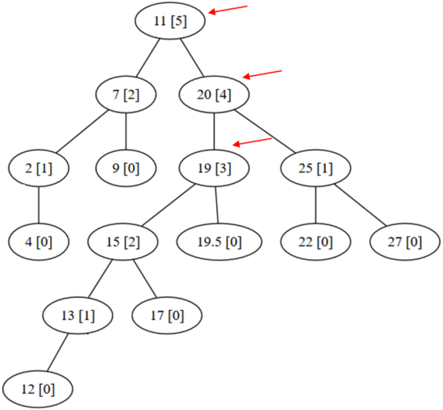


- Solution: single rotation to right

AVL Trees - rotation - Single Rotation to Right

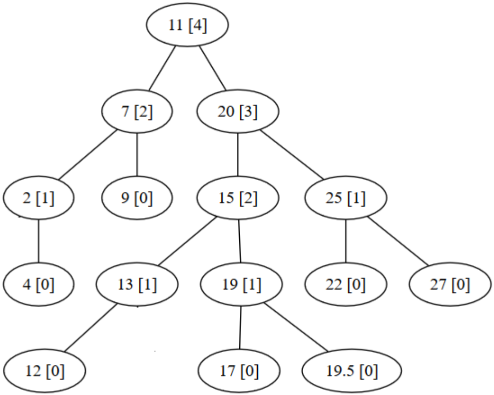


AVL Trees - rotations - case 1 example

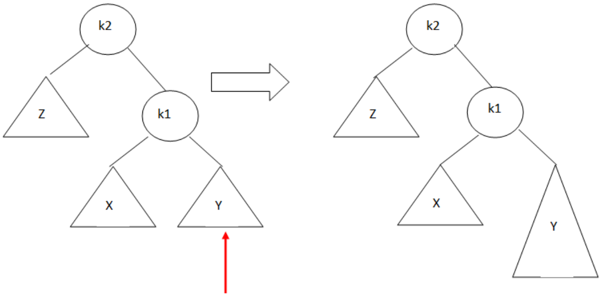


- Node 19 is imbalanced, because we inserted a new node (12) in the left subtree of the left child.

AVL Trees - rotation - case 1 example

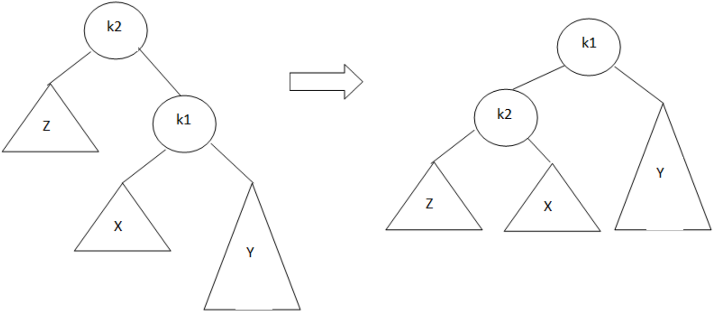


AVL Trees - rotations - case 4

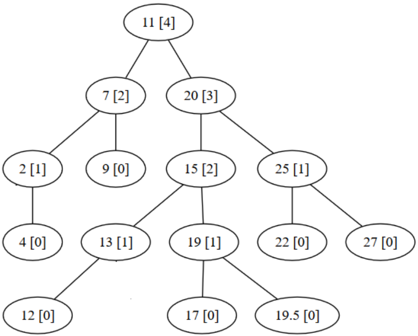


- Solution: **single rotation to left**

AVL Trees - rotation - Single Rotation to Left

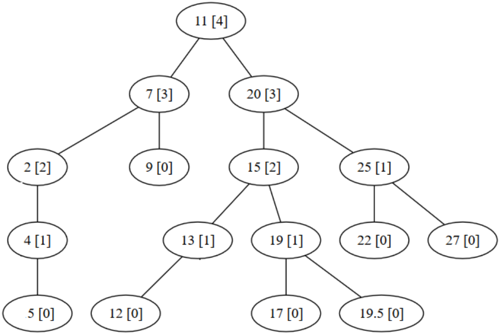


AVL Trees - rotations - case 4 example



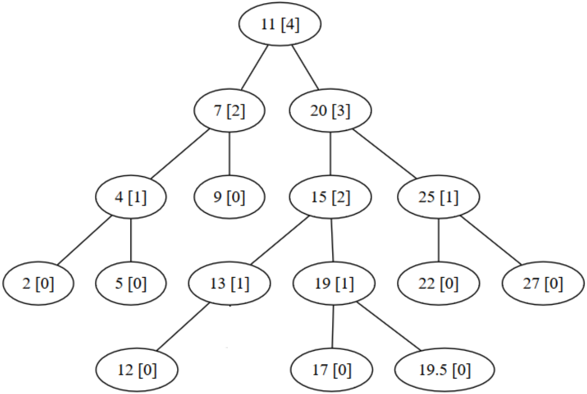
- Insert value 5

AVL Trees - rotations - case 4 example



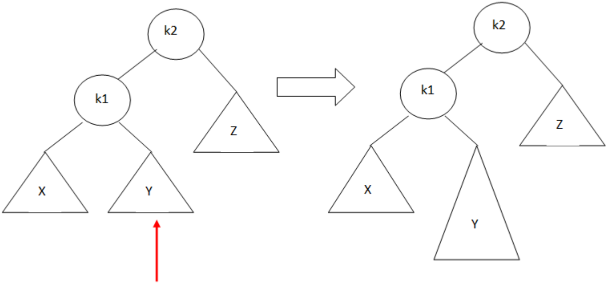
- Node 2 is imbalanced, because we inserted a new node (5) to the right subtree of the right child
- Solution: **single rotation to left**

AVL Trees - rotation - case 4 example



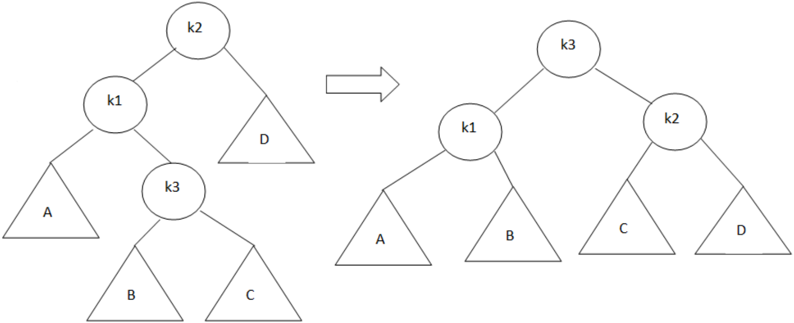
- After the rotation

AVL Trees - rotations - case 2

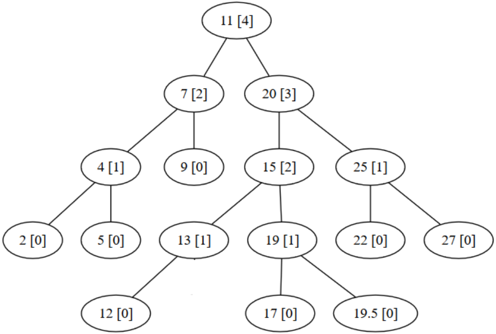


- Solution: **Double rotation to right**

AVL Trees - rotation - Double Rotation to Right

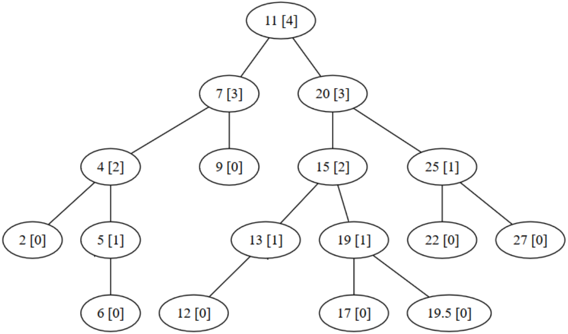


AVL Trees - rotations - case 2 example



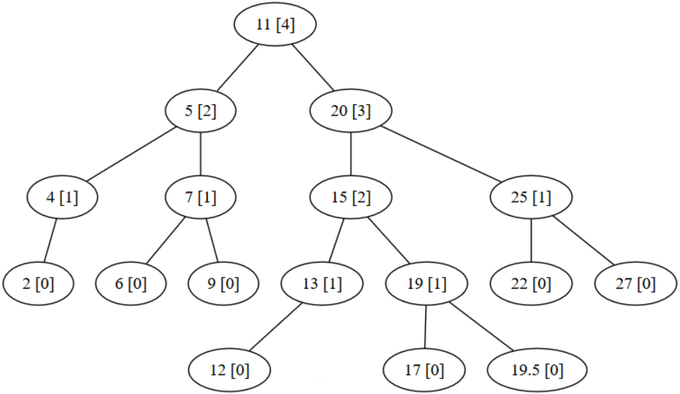
- Insert value 6

AVL Trees - rotations - case 2 example



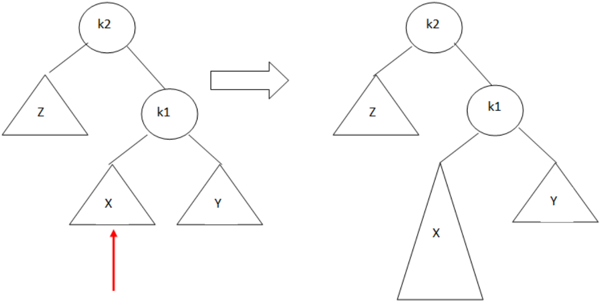
- Node 7 is imbalanced, because we inserted a new node (6) to the right subtree of the left child
- Solution: **double rotation to right**

AVL Trees - rotation - case 2 example



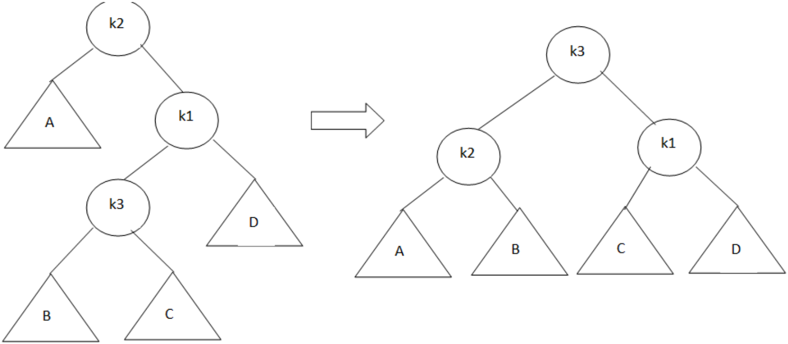
- After the rotation

AVL Trees - rotations - case 3

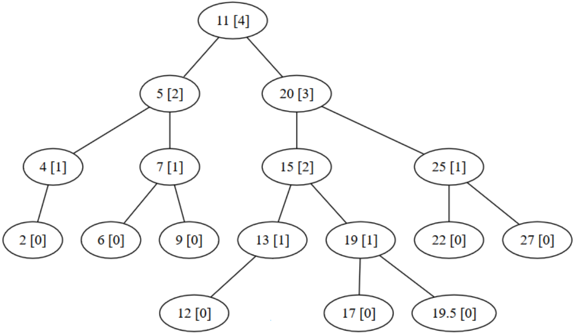


- Solution: **Double rotation to left**

AVL Trees - rotation - Double Rotation to Left

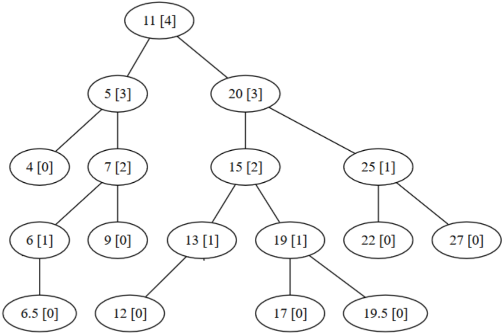


AVL Trees - rotations - case 3 example



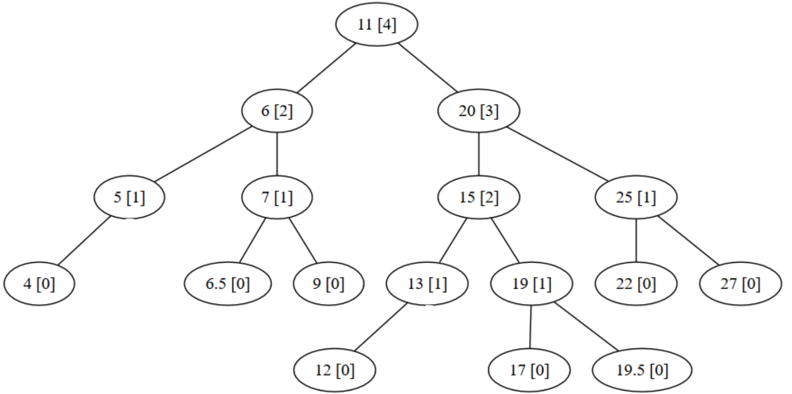
- Remove node with value 2 and insert value 6.5

AVL Trees - rotations - case 3 example



- Node 5 is imbalanced, because we inserted a new node (6.5) to the left subtree of the right child
- Solution: **double rotation to left**

AVL Trees - rotation - case 3 example



- After the rotation