CSCI203

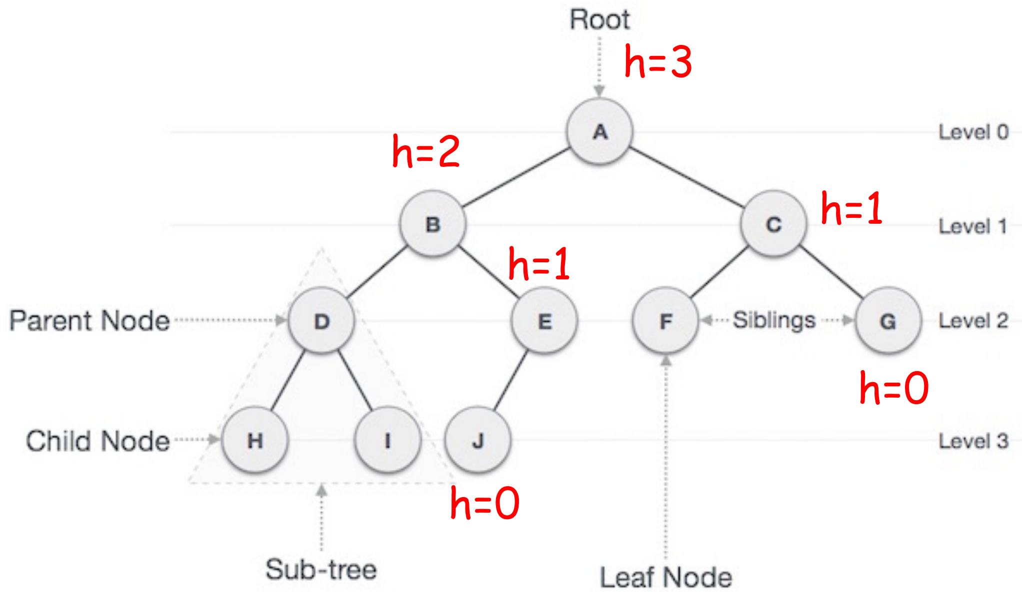
Algorithms and Data Structures

Trees (Part I)

Lecturer: Dr. Fenghui Ren
Room 3.203
Email: fren@uow.edu.au

Binary Trees Revisited

- A Binary tree is a tree in which each node has a maximum of two child nodes, the left child and the right child.
- Important terms
  - Path, Root, Parent, Child, Leaf, Subtree, Visiting , Traversing, Levels (depth), Height, keys



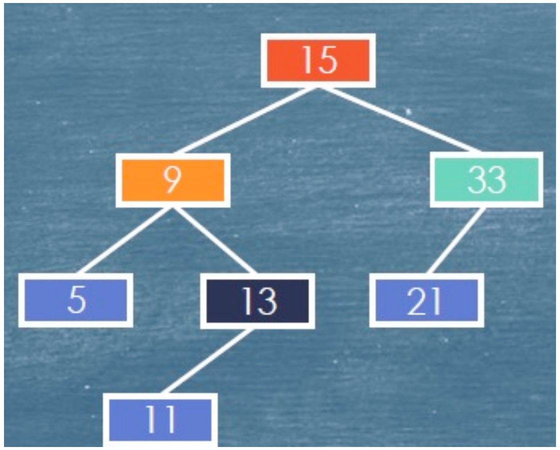
Binary Trees

- A binary tree can be implemented in several ways.
- Some useful approaches are:
  - In an array:
    - `tree: array of stuff`
    - `Root is tree[1]`
    - `The children of tree[i] are tree[2*i] and tree[2*i+1]`
  - As a collection of dynamic records:
    - ```
      type tree_node = record
          contents: stuff
          left: ^tree_node
          right: ^tree_node
          root: ^tree_node
      ```
  - As an array of records:
    - ```
      type tree_array_node = record
          contents: stuff
          left: int
          right: int
      ```
    - `tree: array of tree_array_node`

Binary Search Trees (BSTs)

- This is a binary tree with one extra condition:
  - For each non-leaf node:
    - The contents of the left child ≤ the contents of the node;
    - The contents of the node ≤ the contents of the right node.

$5 \leq 9 \leq 13$



$9 \leq 15 \leq 33$

$21 \leq 33$

$11 \leq 13$

BST - Basic Operations

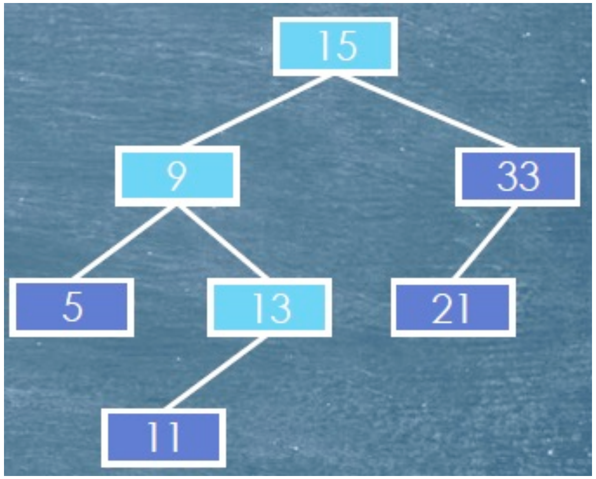
- Insert
  - Inserts an element in a tree/create a tree.
- Find
  - Searches an element in a tree.
- Delete
  - Deletes an element in a tree
- Traversals
  - Preorder - Traverses a tree in a pre-order manner, i.e., root, left, right
  - Inorder - Traverses a tree in an in-order manner, i.e., left, root, right
  - Postorder - Traverses a tree in a post-order manner, i.e., left, right, root.

Searching a BST

```
//use the list
procedure find(value: stuff, node: ^tree_node): ^tree_node
    if value == nil then
       return not_found
    fi
    if value == node.contents then
       return node
    else if value < node.contents then
       find(value, node.left)
    else
       find(value, node.right)
    fi
end
```

Searching: An example

- Consider the BST shown at the right:
  - `find(13, root)`
  - `13 < 15; find(13, root.left)`
  - `find(13, node)`
  - `13 > 9; find(13, node.right)`
  - `find(13, node)`
  - `13 = 13; return node`



Building a BST

- To build a BST - we add nodes, one at a time by:
  - Searching the existing BST for the value (e.g. key) to be inserted.
  - If, the value is not found:
    - Create a new node;
    - Add the new node to the tree as the appropriate child of the last node examined.
- A comparison between the value to be inserted and the value stored in the last valid node will determine which child is to be selected.
- The first node is a special case:
  - Here we must create the first node of the tree and point root at it.

Building a BST

```
procedure insert_first(value): ^tree_node
    node: ^tree_node
    start = new_tree_node
    start.contents = value
    return start
end
```

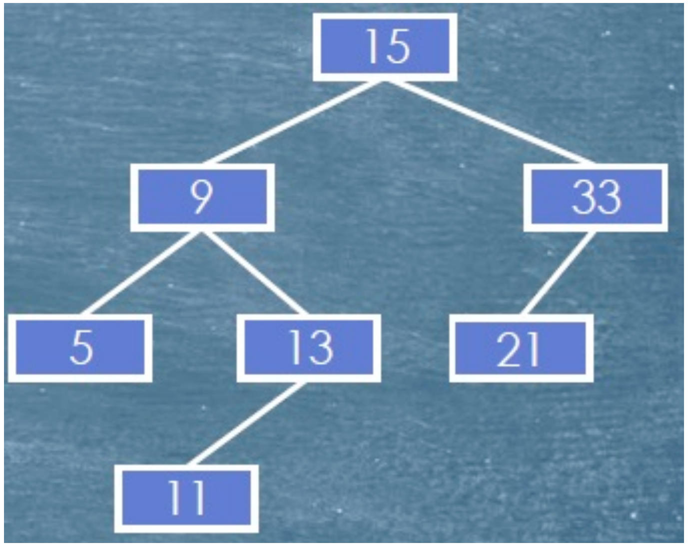
- We create the BST by:
  - `root = insert_first(value)`

Building a BST…

```
procedure insert(value: stuff, node): ^tree_node
    next: ^tree_node, left: boolean
    if value == node.contents then
        return                         // already in the tree
    else if value < node.contents then
        next = node.left; left= true  // we need to go left
    else
        next = node.right; left = false // we need to go right
    fi
    if next != nil then
        insert (value, next)              // keep trying
    else
        next = new_tree_node              // make a new node
        next.contents = value             // store the value
        if left then                      // update the parent
            node.left = next
        else
            node.right = next
        fi
    fi
end
```

Building a BST: An Example

- Let us build a BST from the following values:
  - 15, 33, 9, 13, 5, 21, 11
- `root = insert_first(15)`
- `insert(33)`
- `insert(9)`
- `insert(13)`
- `insert(5)`
- `insert(21)`
- `insert(11)`

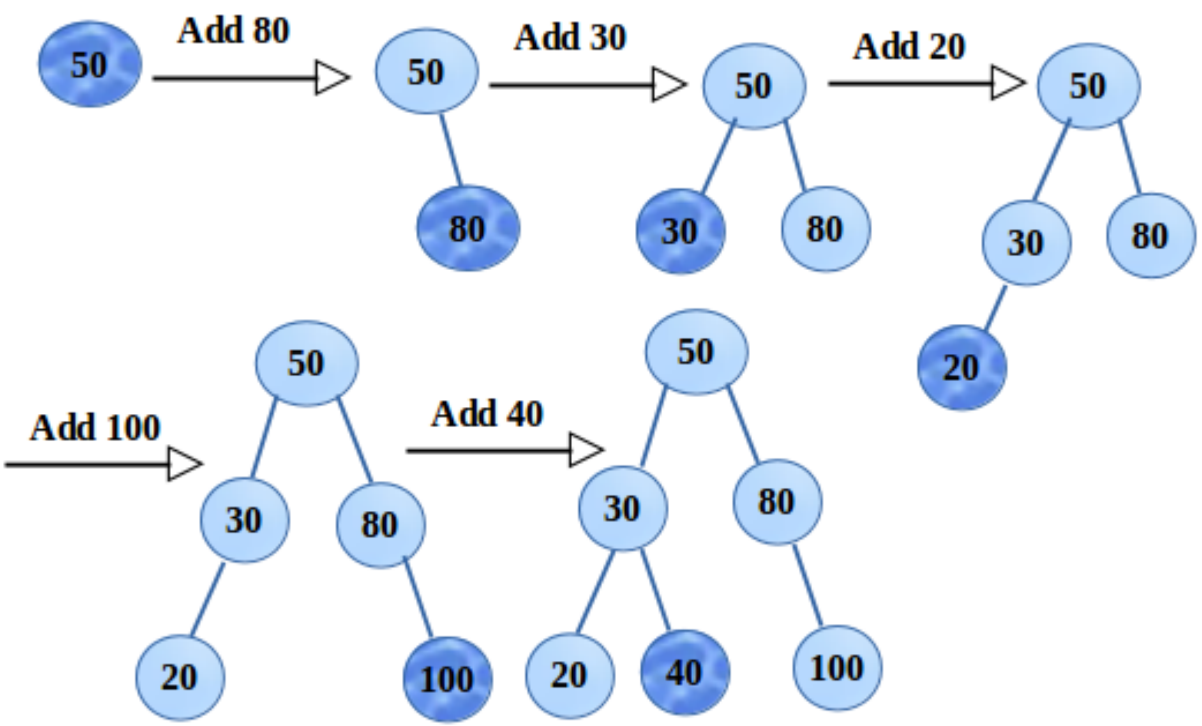


Delete a node

```
Node deleteNode(Node root, int valueToDelete) {
  if root = null
    return node
  if root.value < valueToDelete
    deleteNode(root.right, valueToDelete) //search in the right subtree
  if root.value > valueToDelete
    deleteNode(root.left, valueToDelete) //search in the life subtree
  else if root.value == valueToDelete
    if (root.left==null && root.right==null)
      delete the root //delete the node if it is a leaf
      return null
    if (root.right == null)  //replace the root with its left subtree
      delete the root
      return root.left
    if (root.left == null)  //replace the root with its right subtree
      delete the root
      return root.right
    else  //replace the root with the minimum node in its right subtree
      minRNode = the minimum node in the right subtree
      root.value = minRNode.value
      deleteNote(root.right, minRNode.value)
      return root
```

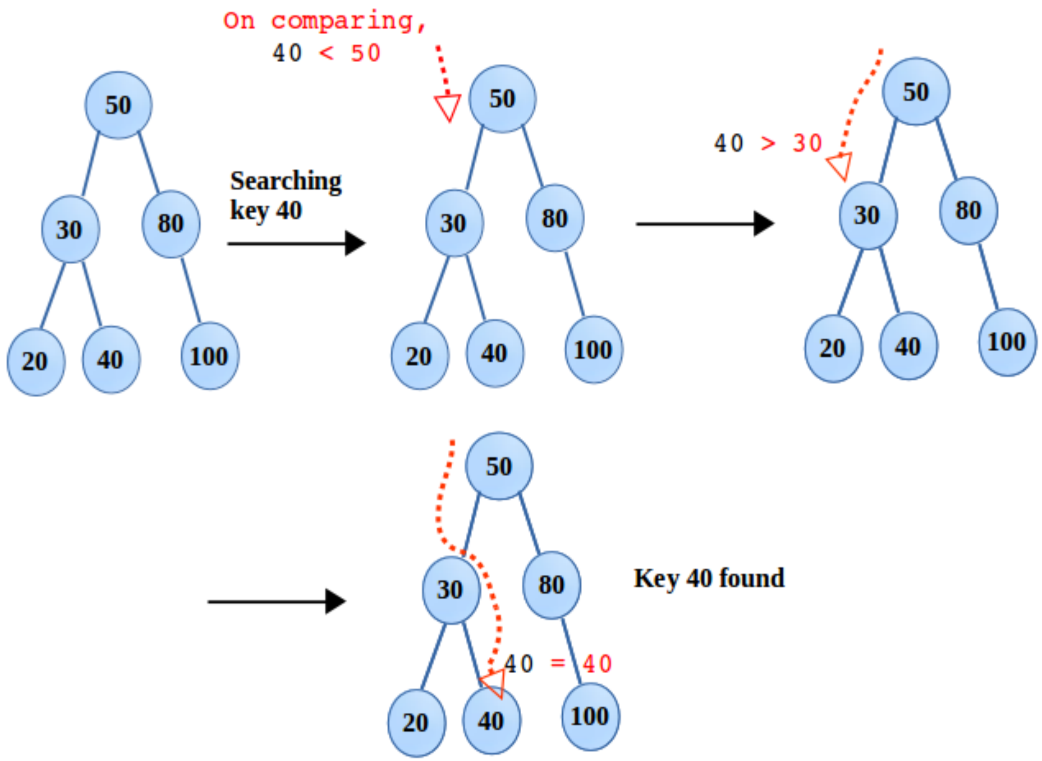
BST - Example

- Insertion



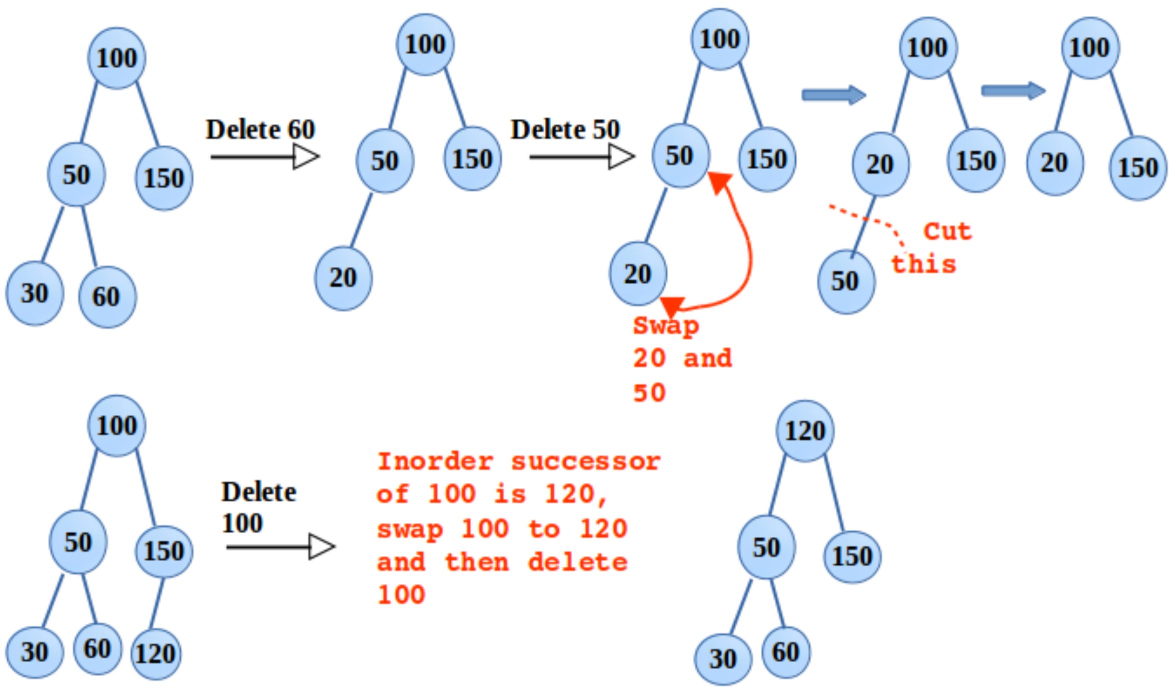
BST - Example

- Searching



BST - Example

- Deletion



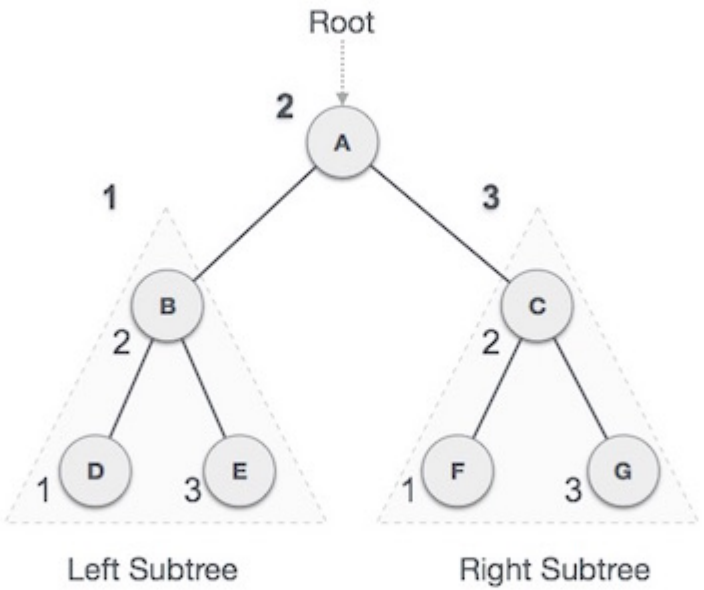
BSTs - Traversals

- Traversal is a process to visit all the nodes of a tree and may print their values too.
  - Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –
  - In-order Traversal: root is in the middle of left and right
  - Pre-order Traversal: root is before left and right
  - Post-order Traversal: root is after left and right

- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

BST - In-order Traversal

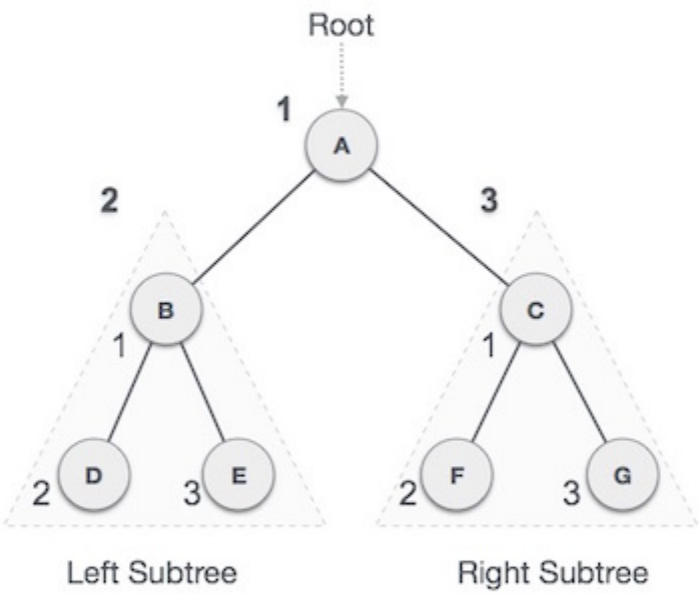
- Until all nodes are traversed –
  1. Recursively traverse left subtree.
  2. Visit root node.
  3. Recursively traverse right subtree.



D → B → E → A → F → C → G

BST - Pre-order Traversal

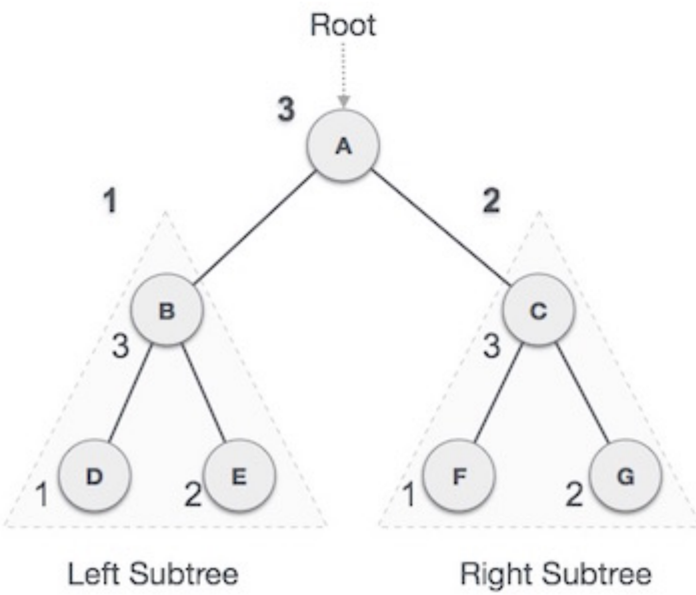
- Until all nodes are traversed –
  1. Visit root node.
  2. Recursively traverse left subtree.
  3. Recursively traverse right subtree.



A → B → D → E → C → F → G

BST - Post-order Traversal

- Until all nodes are traversed –
  1. Recursively traverse left subtree.
  2. Recursively traverse right subtree.
  3. Visit root node.

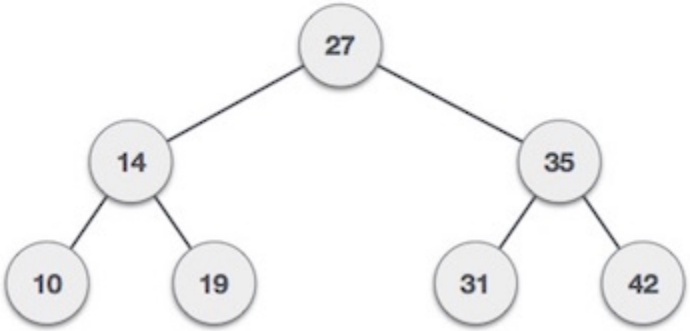


D → E → B → F → G → C → A

Sorting with BSTs

- If we perform an in-order traversal of a binary search tree, the nodes of the tree will be listed in sorted order. Why?
- Recall - In order traversal:

```
procedure visit(node: ^tree_node)
   if node.left != nil then
     visit(node.left)
   fi
   print(node.contents)
   if node.right != nil then
     visit(node.right)
   fi
   return
end
```

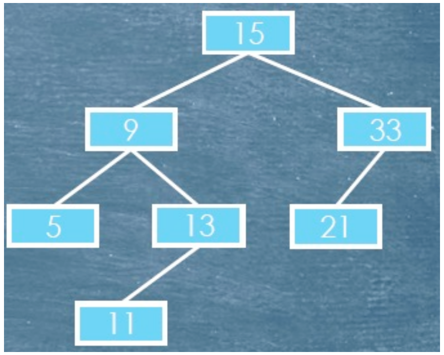


- This is BST Sort - simply call `visit(root)`.

Sorting: An Example

Considering the BST shown to the right

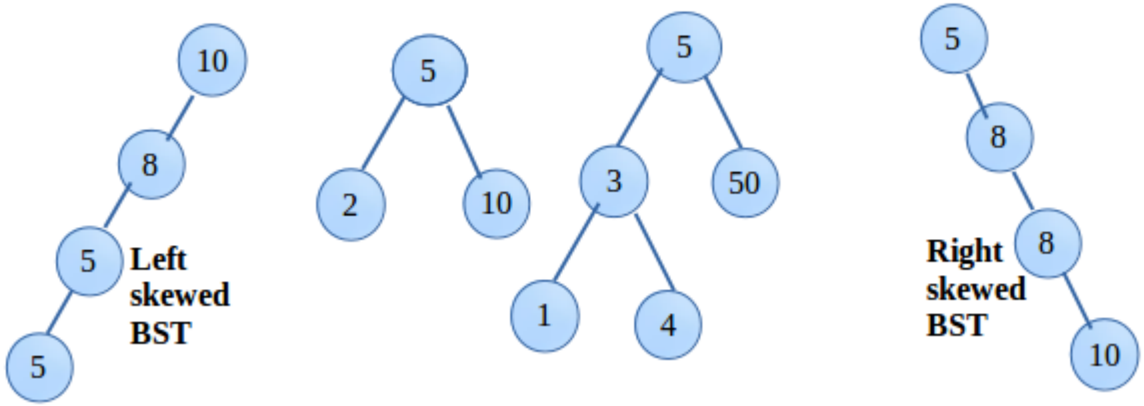
- visit(root)
- visit(root.left)
- visit(node.left)
- print(node.contents) **5**
- return
- print(node.contents) **9**
- visit(node.right)
- visit(node.left)
- print(node.contents) **11**
- return
- print(node.contents) **13**
- return
- return
- print(node.contents) **15**
- visit(node.right)
- visit(node.left)
- print(node.contents) **21**
- return
- print(node.contents) **33**
- return
- return



5	9	11	13	15	21	33

The Problem with BST

- If we create a BST from the following sequence:
  - 10, 8, 5, 5 or 5, 8, 8, 10
- We get the following BST:



- This tree is severely unbalanced.

Balancing a BST

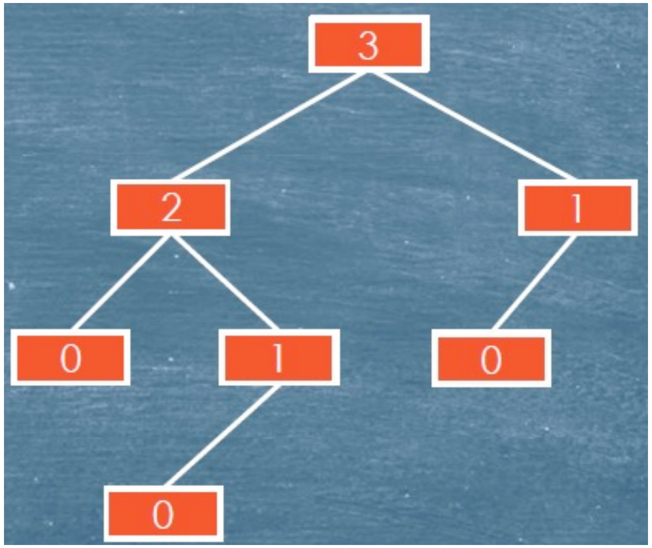
- Operations on BSTs are only $\Theta(\log(n))$ for balanced trees.
- Can we adjust a BST, as we operate on it, to keep it more or less balanced?
- How efficient is this?
- Is it worth the effort?
- What do we mean by balanced, anyway?

Balanced?

- Try "left and right subtrees must be of the same height"
  - This is easy to achieve but not very useful.
  - Too soft.
- Try "every node must have left and right subtrees of the same height"
  - This is impossible unless the tree is complete.
  - Too hard.
- Try "every node must have left and right subtrees which differ in height by at most 1"
  - This is the AVL (Adelson-Velski and Landis) balance condition.
  - Just right. (As we shall now see.)

AVL Trees

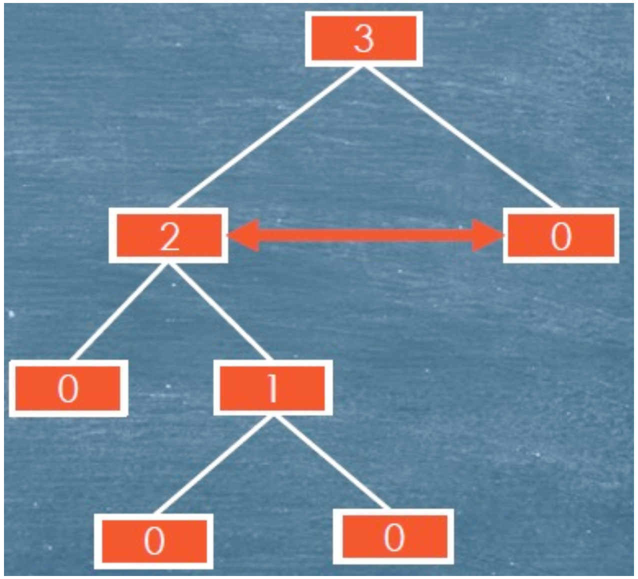
- This is an AVL tree:
- These are the heights:



- Note that, at each node, the heights of its children differ by at most one.

AVL Trees

- This is not an AVL tree:
- These are the heights:
- Note that the root node has children which differ in height by two.



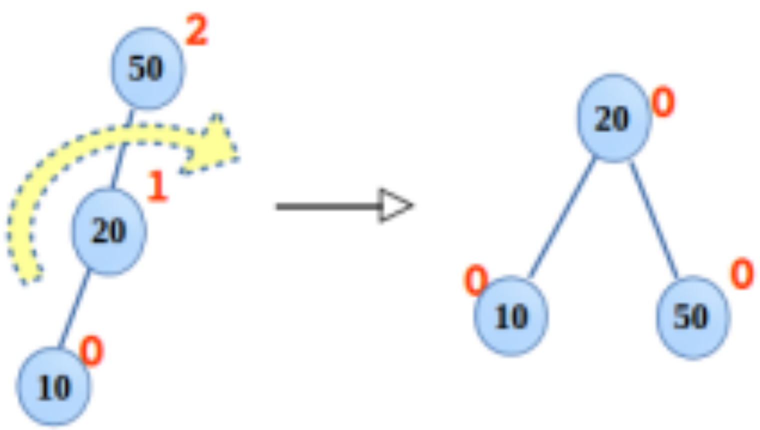
AVL Trees- Balancing Operations

- The balancing condition of AVL tree:
  - *Balance factor = height(Left subtree) – height(Right subtree),*

- It should be -1, 0 or 1. Other than this will cause restructuring (or balancing) the tree. Balancing performed is carried in the following operations
  - Right rotation (RR)
  - Left rotation (LL)
  - Right left double rotation(RL)
  - Left right double rotation(LR)

AVL Trees - Right rotation (RR)

- 20 will be the new root.
- 50 takes ownership of 20's right child,
- 10 is still the left child of 20.

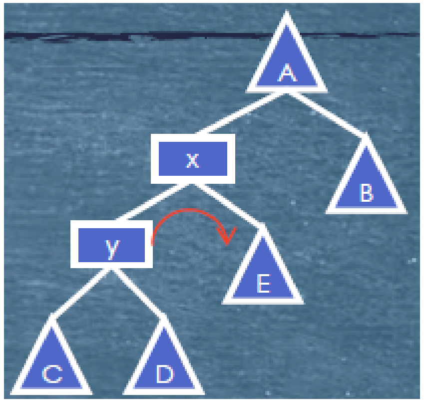


AVL Trees - Implementation

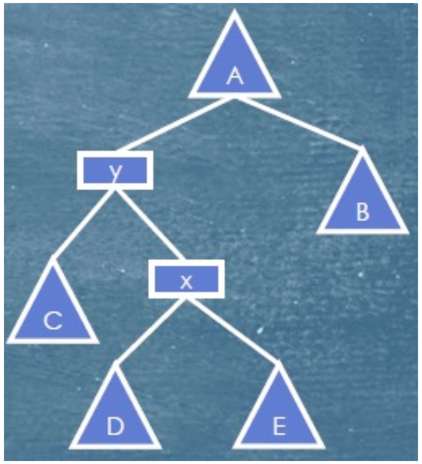
```
type avl_node = record
    value: stuff
    left: ^avl_node
    right: ^avl_node
    height: int
```

AVL Trees - Implementation

```
procedure rotate_right(k2)
    k1 = k2.left
    k2.left = k1.right
    k1.right = k2
    k2.height = max((k2.left).height), (k2.right).height) + 1
    k1.height = max((k1.left).height), k2.height) + 1
    k2 = k1
end
```

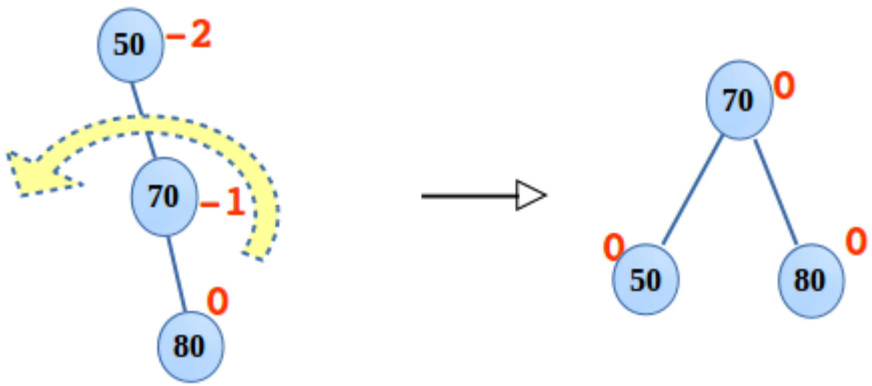


k2=x, k1=y



AVL Trees - Left rotation (LL)

- 70 will be the new root.
- 50 takes ownership of 70's left child.
- 80 is still the right child of 70.



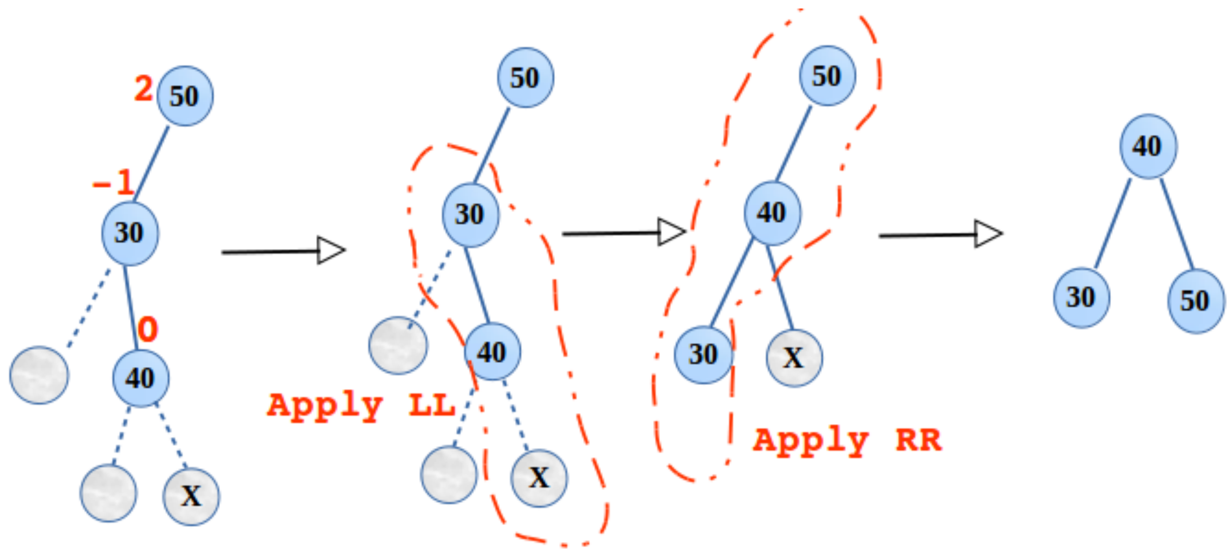
AVL Trees - Implementation

```
procedure rotate_left(k2)
    k1 = k2.right
    k2.right = k1.left
    k1.left = k2
    k2.height = max((k2.left).height), (k2.right).height) + 1
    k1.height = max(k2.height, (k1.right).height), ) + 1
    k2 = k1
End
```

- This is a mirror of the rotate-right operation

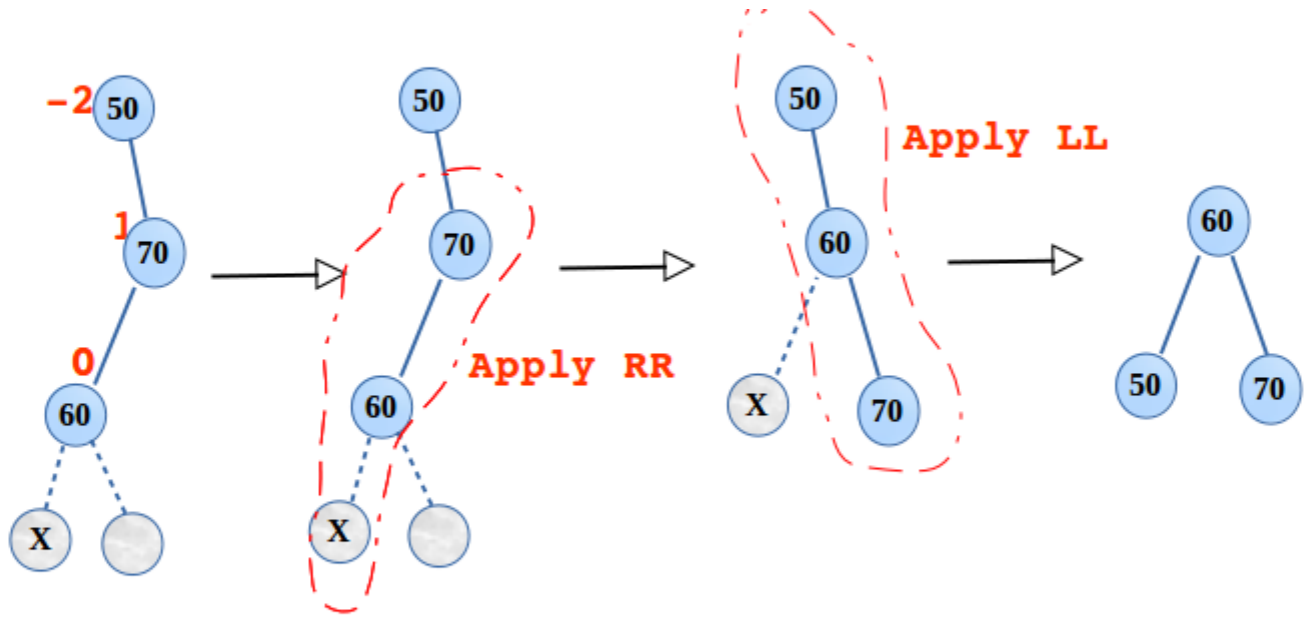
AVL Trees - Right left double rotation (double-right)

- Left rotation is applied at 30, after restructuring 40 takes the place of 30 and 30 as the left child of 40.
- Now right rotation is required at the root 50, 40 becomes root. 30 and 50 becomes the left and right child respectively.



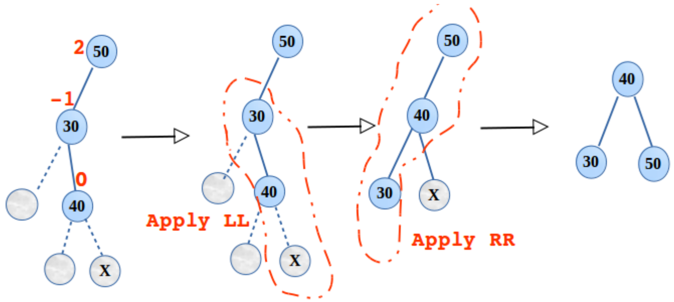
AVL Trees - Left right double rotation (double-left)

- Right rotation is applied at 70, after restructuring, 60 takes the place of 70 and 70 as the right child of 60.
- Now left rotation is required at the root 50, 60 becomes the root. 50 and 70 become the left and right child respectively

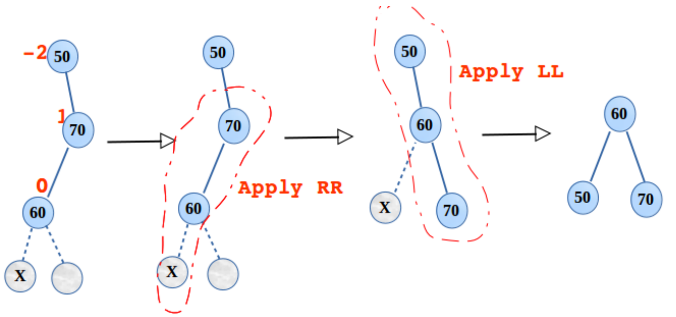


AVL Trees - Implementation

```
procedure double_right( k3)
    rotate_left(k3.left)
    rotate_right(k3)
end
```




```
procedure double_left( k3)
    rotate_right(k3.right)
    rotate_left(k3)
end
```



Note: The pseudo-code presented here does not take into account the fact that the parent of the top node must change as part of the rotation.

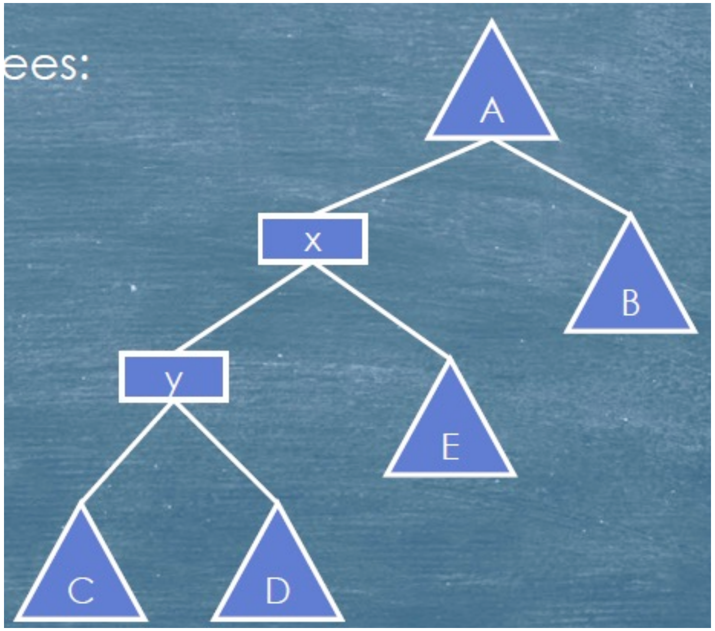
Losing My Balance

- Insertion can unbalance an AVL tree node β
  1. An insertion into the left subtree of the left child of β
  2. An insertion into the right subtree of the left child of β
  3. An insertion into the left subtree of the right child of β
  4. An insertion into the right subtree of the right child of β

- Cases 1 and 4 are equivalent, as are cases 2 and 3 (although there are still 4 cases from a coding viewpoint).

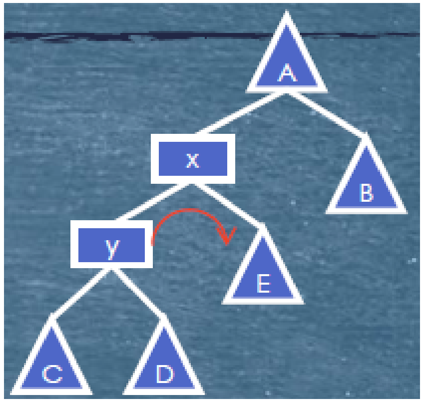
An Abstract Tree

- Consider the following abstract tree:
  - Triangles represent sub-trees:
  - Boxes represent nodes.

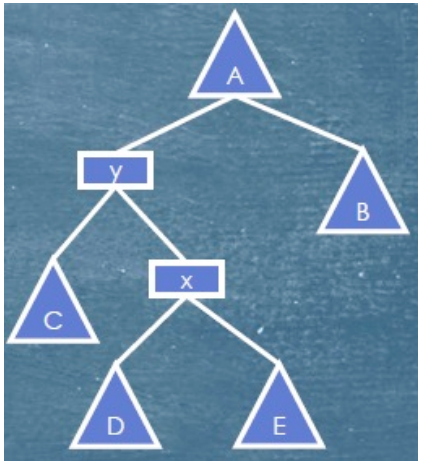


Tree Rotation

- Right rotation on pivot X
  - Node y replaces node x as the "local root".
  - Node x becomes the right child of node y.
  - Subtree D moves from being the right child of node y to the left child of node x.



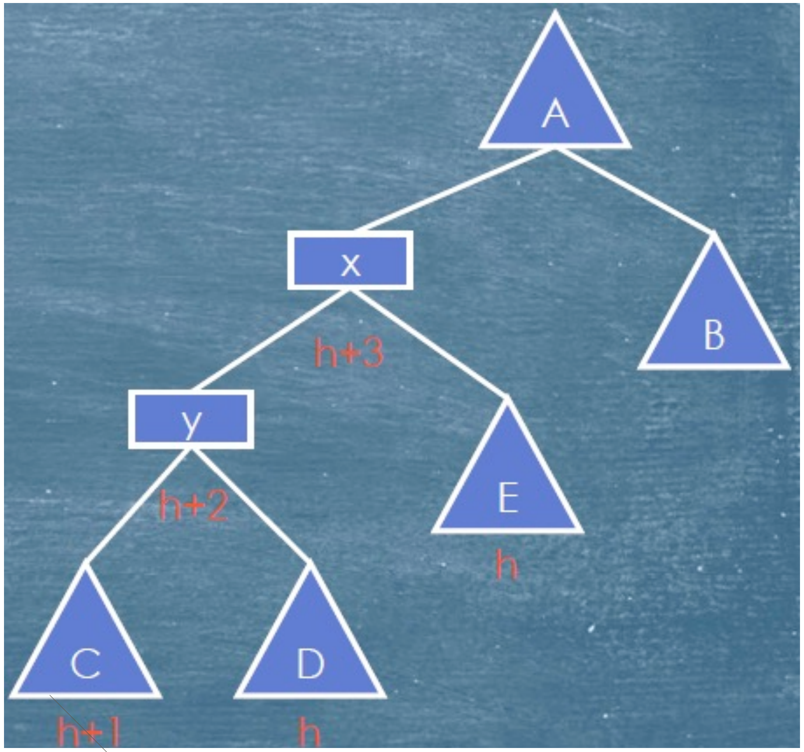
- This is a right rotation.
  - The pivot node becomes the right child.



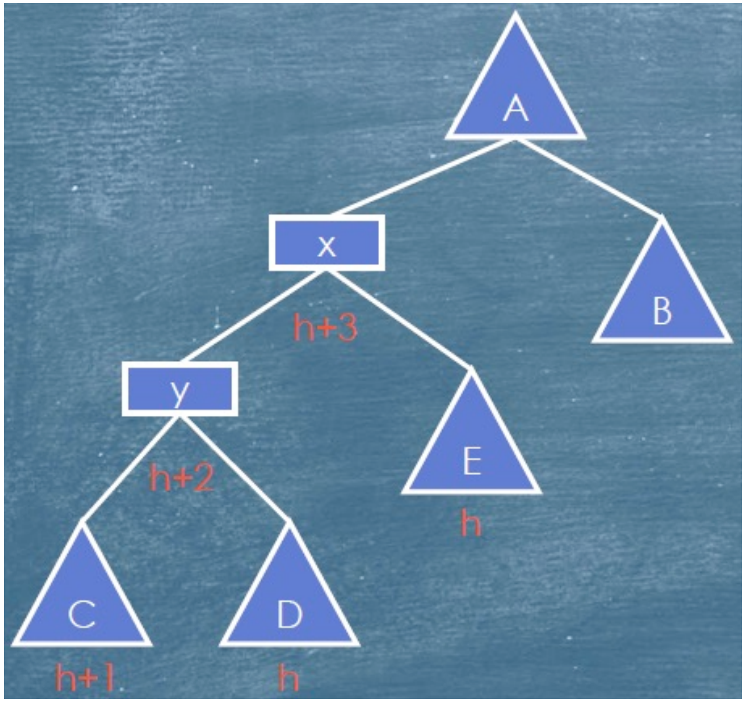
- Note: If we started with a BST we still have a BST! Why?
- Before rotation: c<=y<=d<=x<=e<=a<=b
- After rotation: c<=y<=d<=x<=e<=a<=b

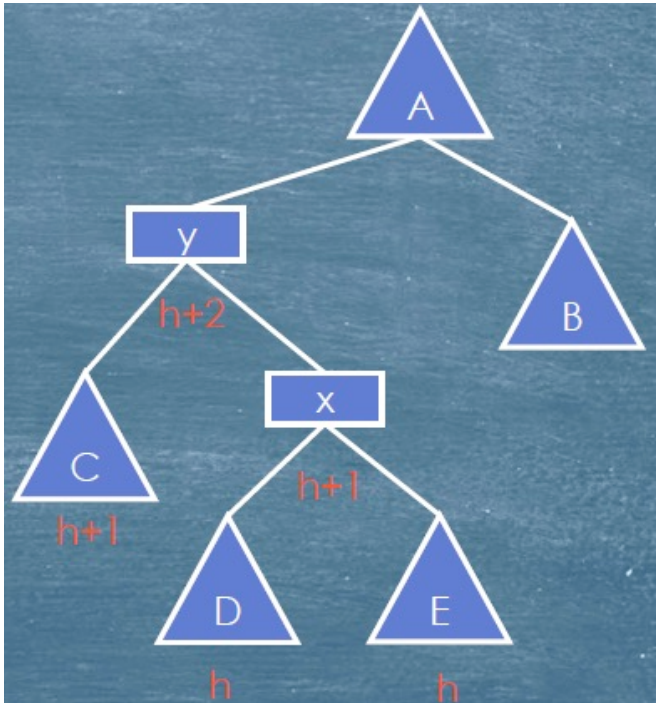
Case 1

- Consider what happens when insertion into the left subtree of node y causes the tree to lose its AVL balance at node x:
- Tree heights are shown below the components:



Before | After



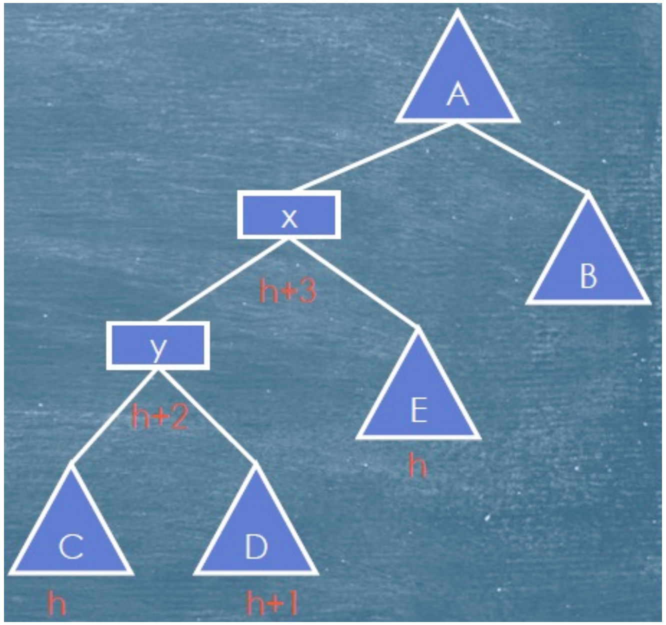


We have restored the balance!

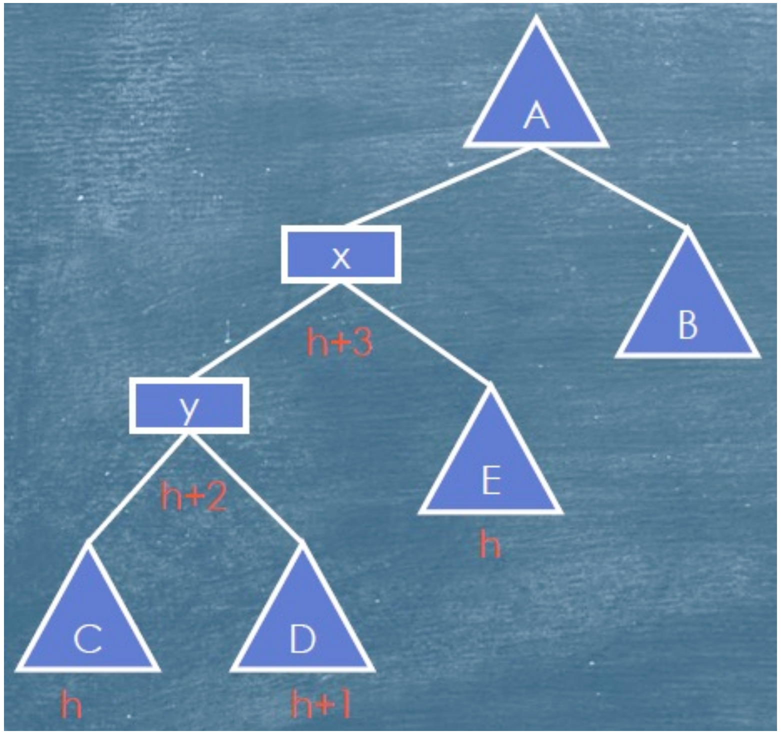
Case 2

- Consider what happens when the imbalance occurs as a result of insertion into the right subtree of the left child (D):

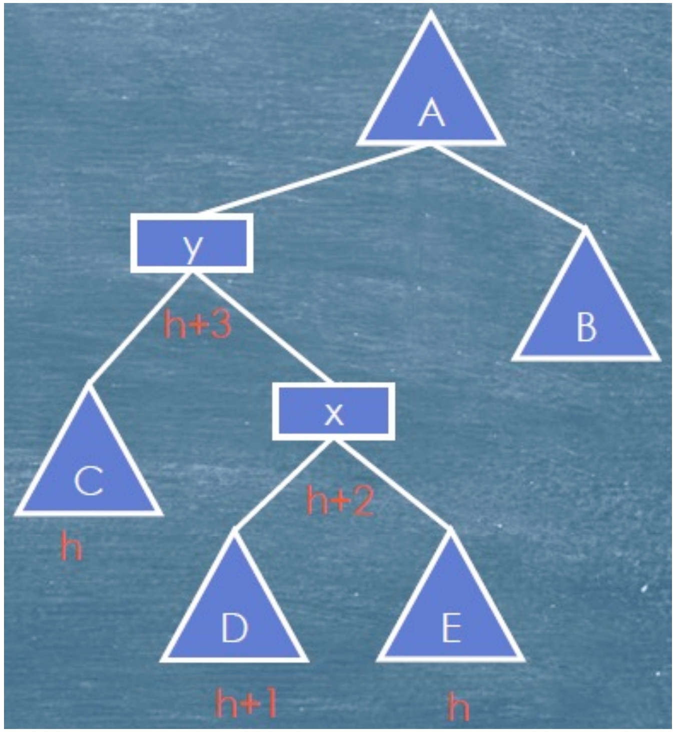
- Will a single rotation work this time?



Before



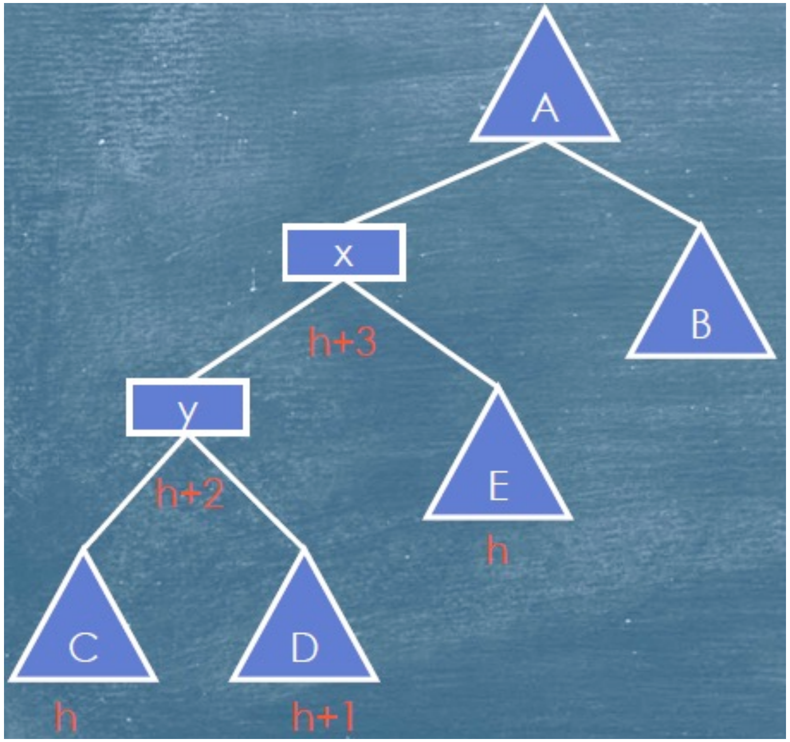
After

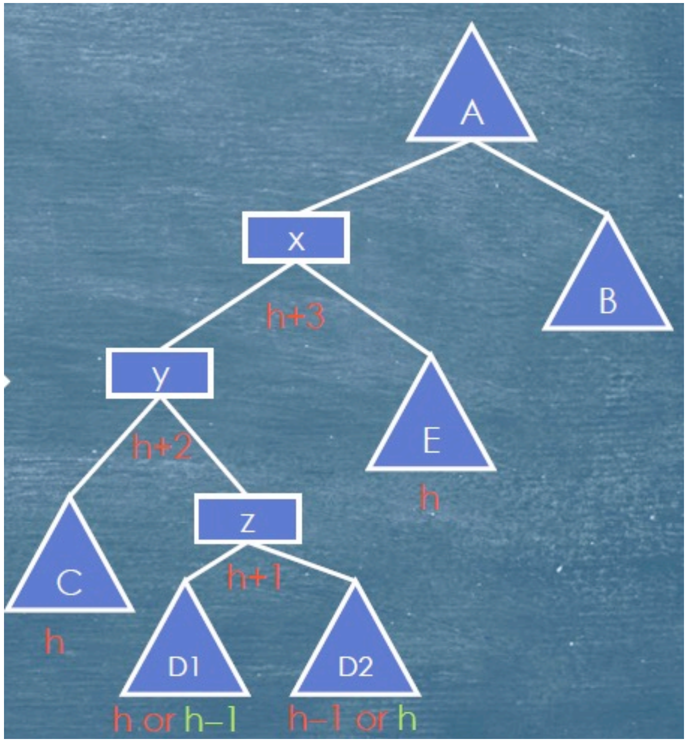


We still have an imbalance !

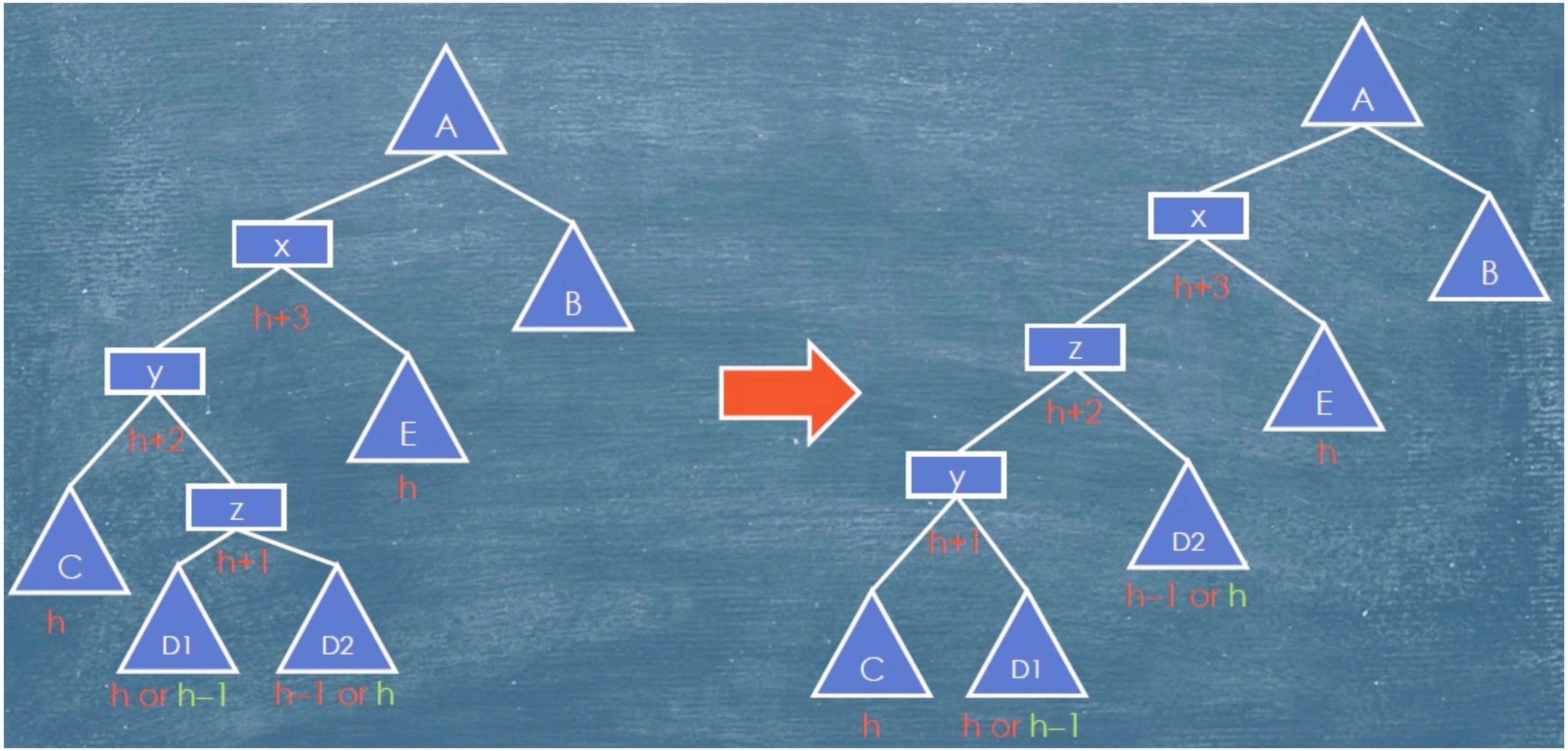
Case 2

- Let us expand subtree D.

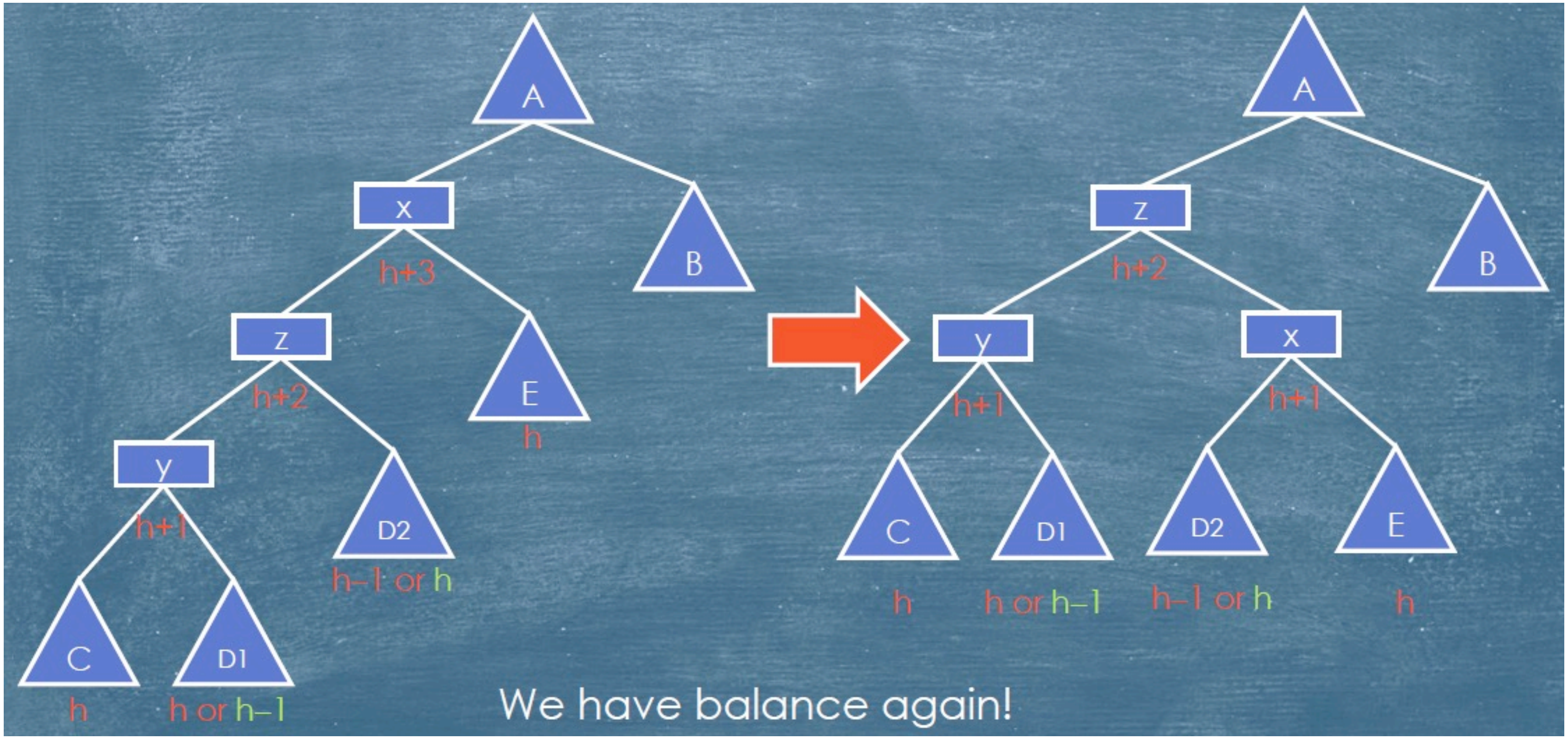




- First perform a left rotation on node y



- Then perform a right rotation on node x



RL operation

Building an AVL tree

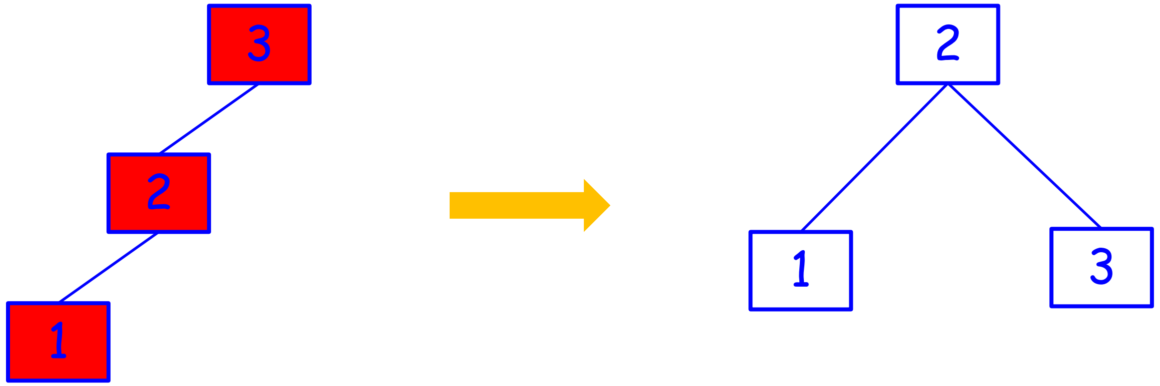
```
Initialize an AVL-tree
Repeat until all stuff are done
    Create a node from a key
    Insert the node into the tree
    Balance T
End
```

AVL Trees - Implementation

```
procedure avl_insert(key, tree)
    if (tree = nil) then
        tree = new avl_node(key, nil, nil, 0)
    else if key < tree.value then
        avl_insert(key, tree.left)
        if (tree.left).height - (tree.right).height) = 2 then
            if key < (tree.left).value then
                rotate_right(tree) // case 1
            else
                double_right(tree) // case 2
            fi
    else if key > tree.value then
            avl_insert(key, tree.right)
            if (tree.right).height - (tree.left).height) = 2 then
                if key < (tree.right).value then
                    double_left(tree) // case 3
                else
                    rotate_left(tree) // case 4
                fi
    fi
    tree.height = max((tree.left).height, (tree.right).height) + 1
end
```

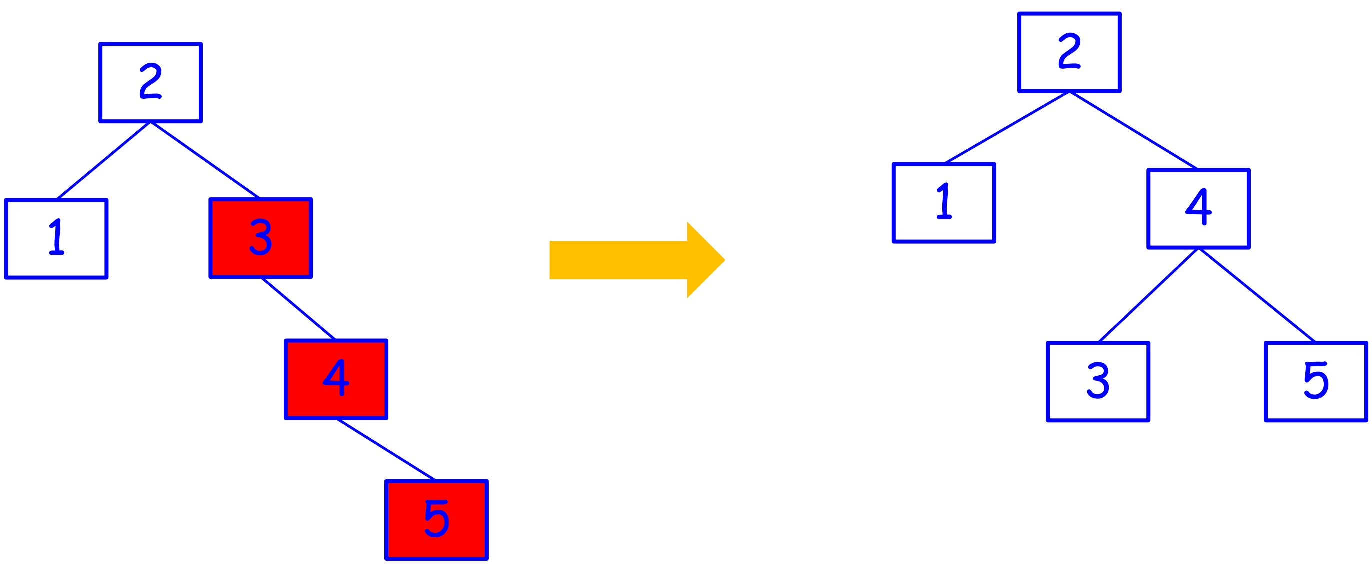
An Example

- Let us build an AVL tree one node at a time:
  - Insert 3
  - Insert 2
  - Insert 1
    - We have an imbalance at node 3
    - Right rotate node 3



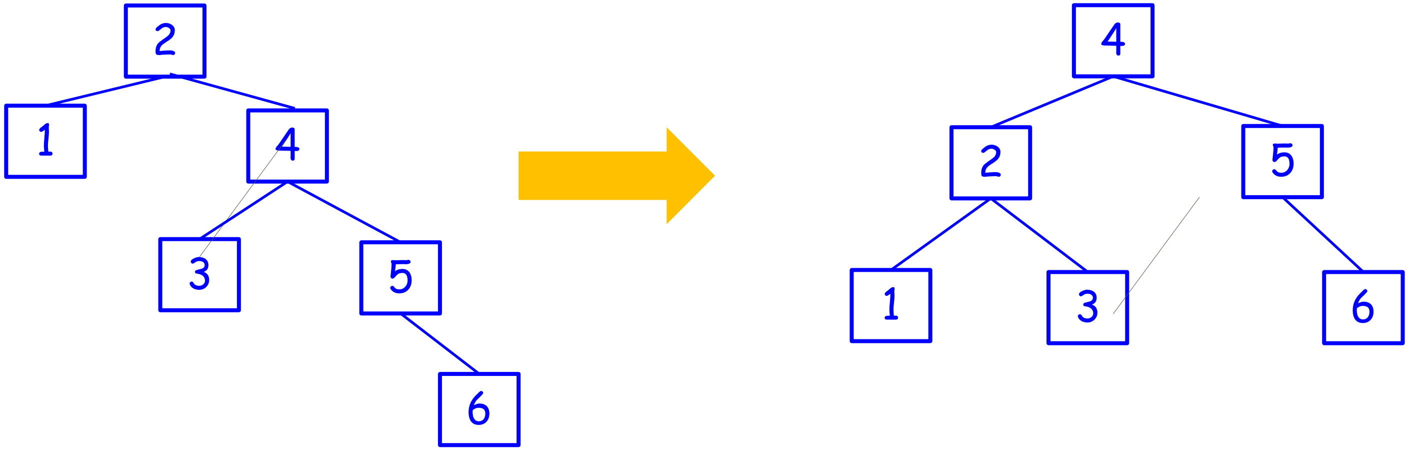
An Example

- Insert 4
- Insert 5
  - We have an imbalance at node 3
  - Left rotate node 3



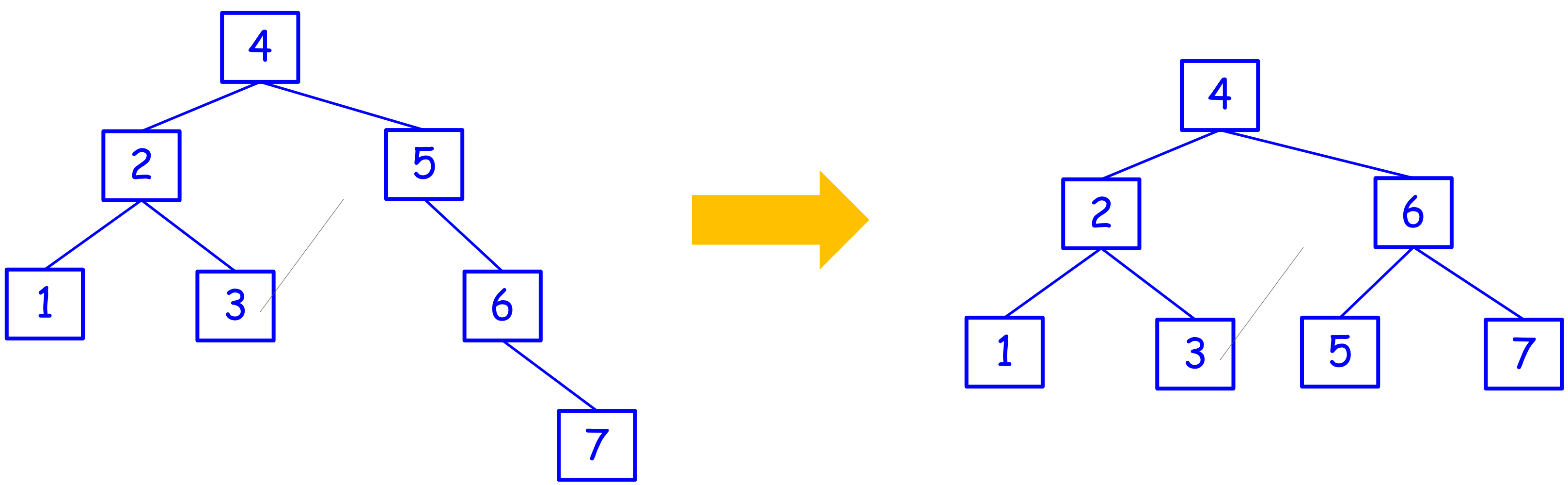
An Example…

- The tree so far
  - Insert 6
    - We have an imbalance at node 2
    - Left rotate node 2



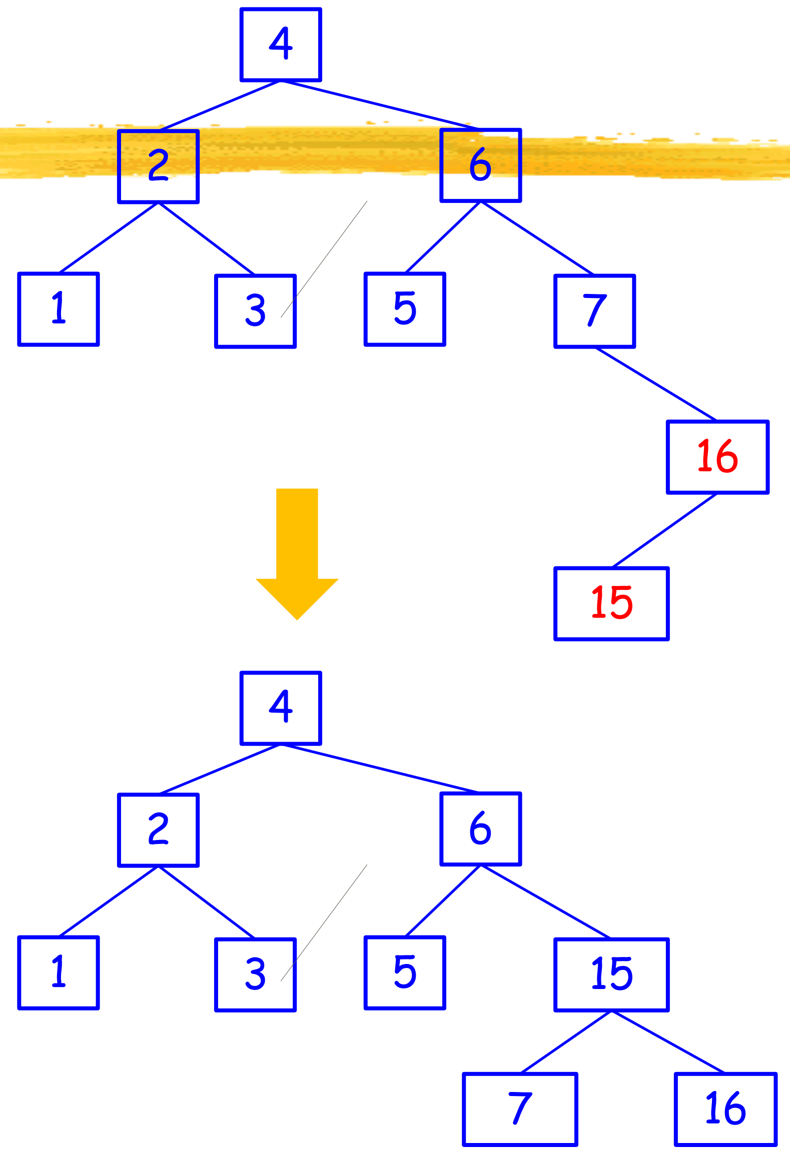
An Example…

- The tree so far
  - Insert 7
    - We have an imbalance at node 5
    - Left rotate node 5



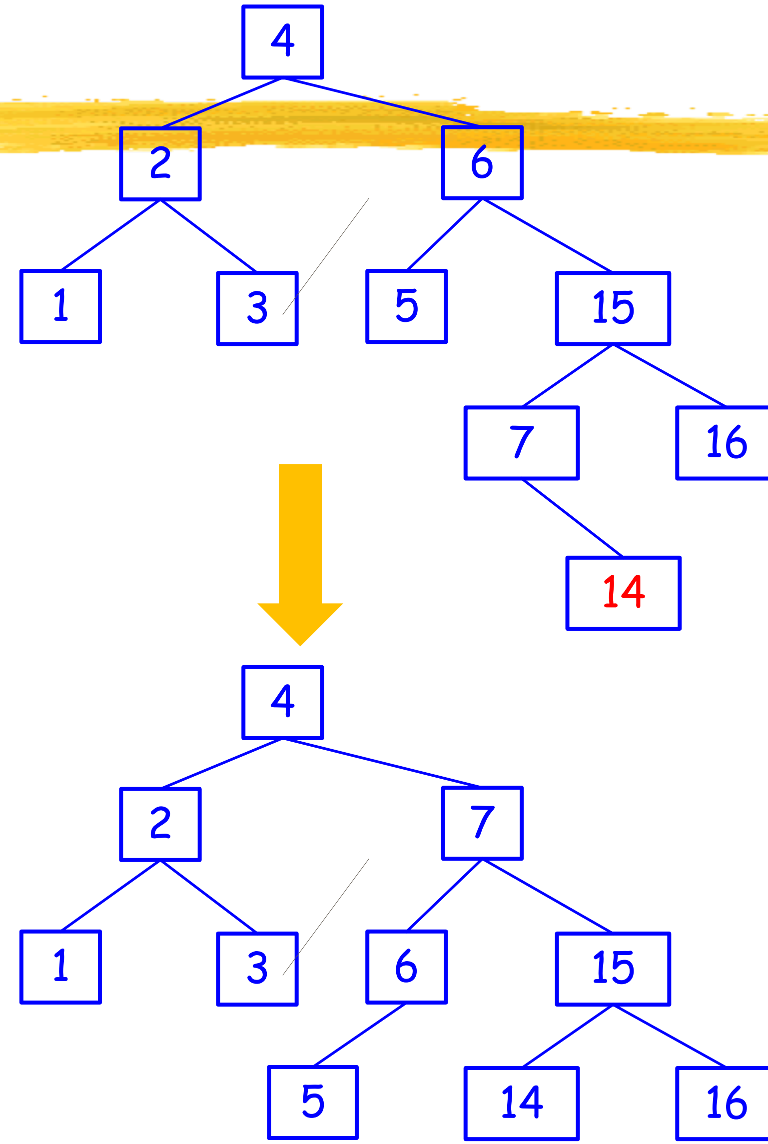
An Example...

- The tree so far
  - Insert 16
  - Insert 15
    - We have an imbalance at node 7
    - A double rotation is needed
      - First, right rotate node 16
      - Then, left rotate node 7



An Example…

- The tree so far
  - Insert 14
    - We have an imbalance at node 6
    - A double rotation is needed
      - First, right rotate node 15
      - Then left rotate node 6
- And so on.



Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 6.3
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 13.3