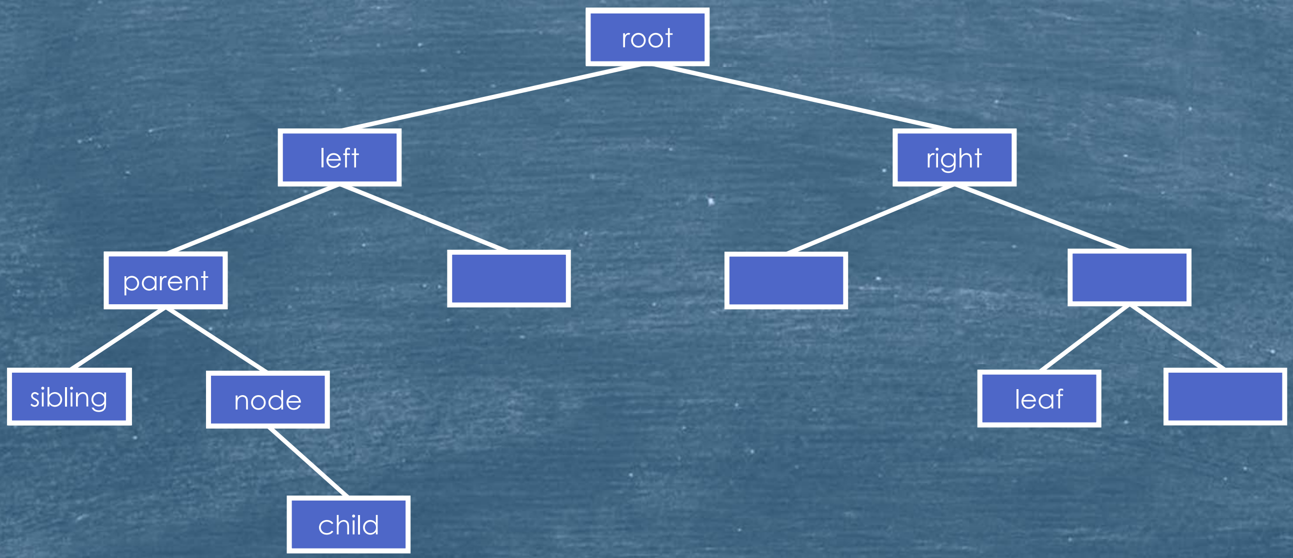
Ian Piper
CSCI203

Algorithms and Data Structures

Week 4 – Lecture A

Binary Trees Revisited

- Remember from 1st year.
- A Binary tree is a tree in which each node has a maximum of two child nodes, the left child and the right child.



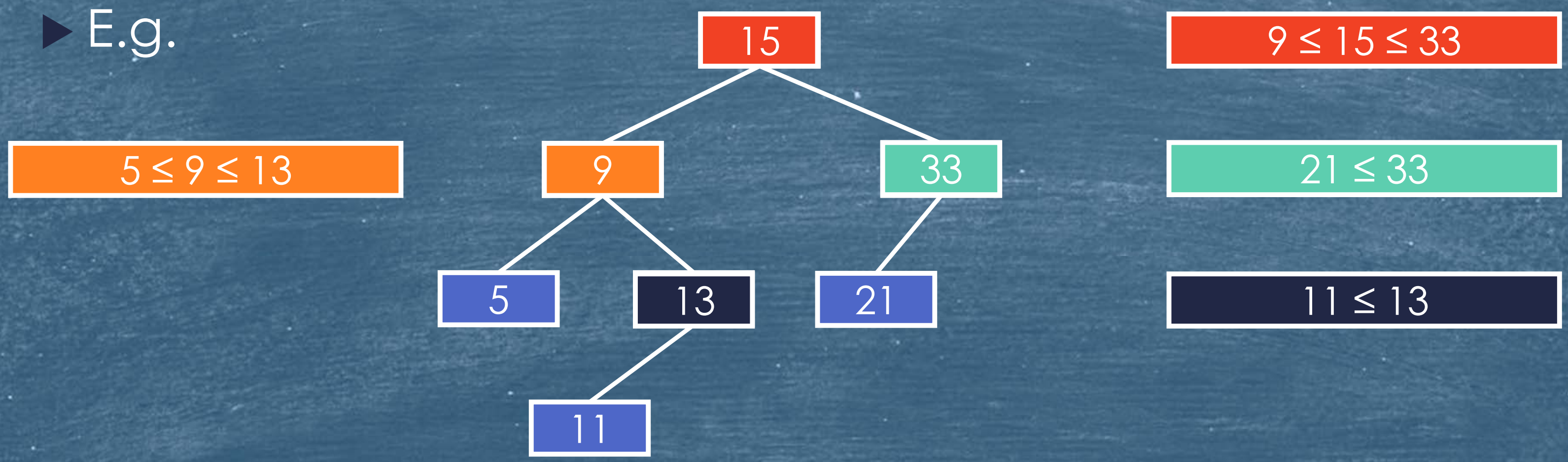
Binary Trees

- 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`**

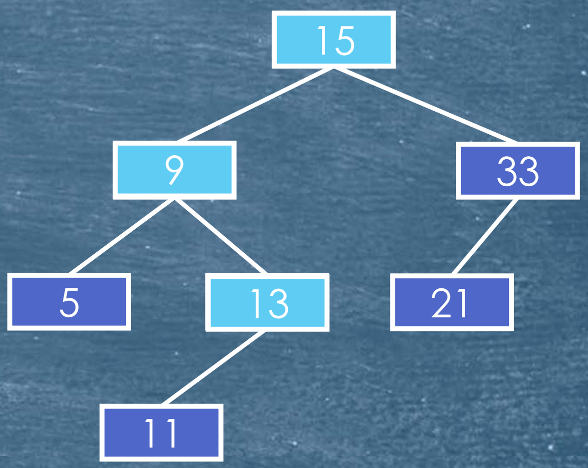
Binary Search Trees

Binary Search Tree

- 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.
- E.g.

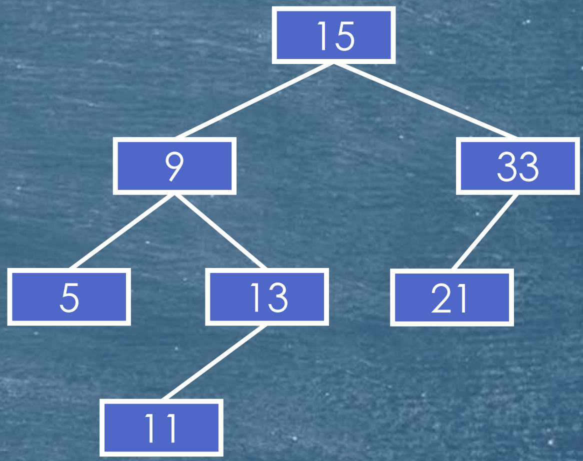


Searching a BST

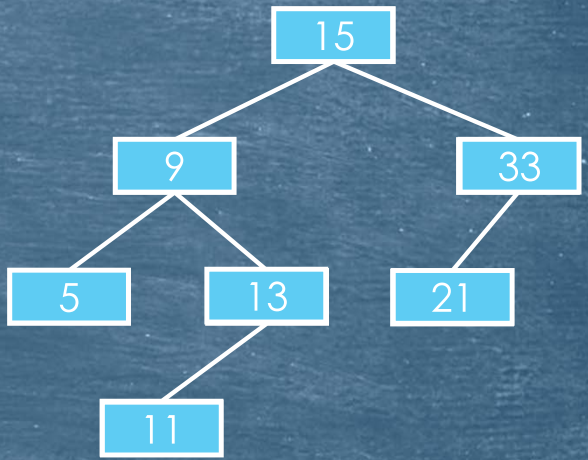
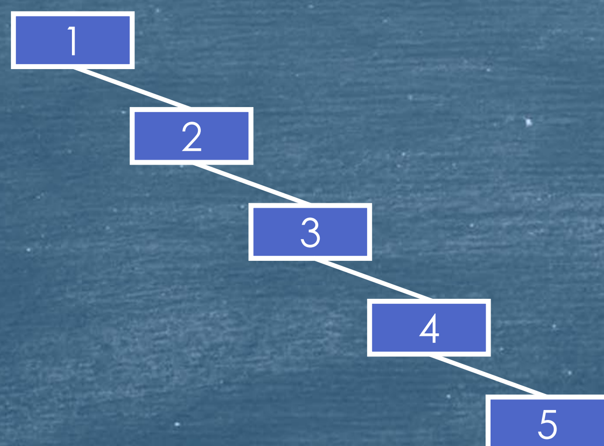
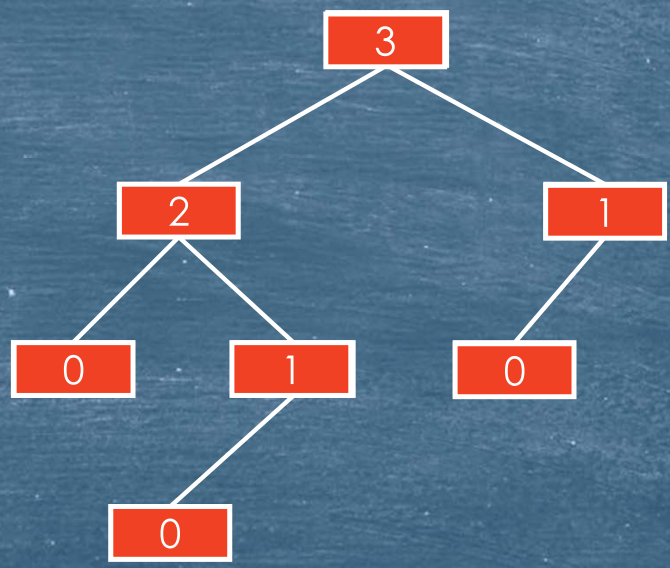
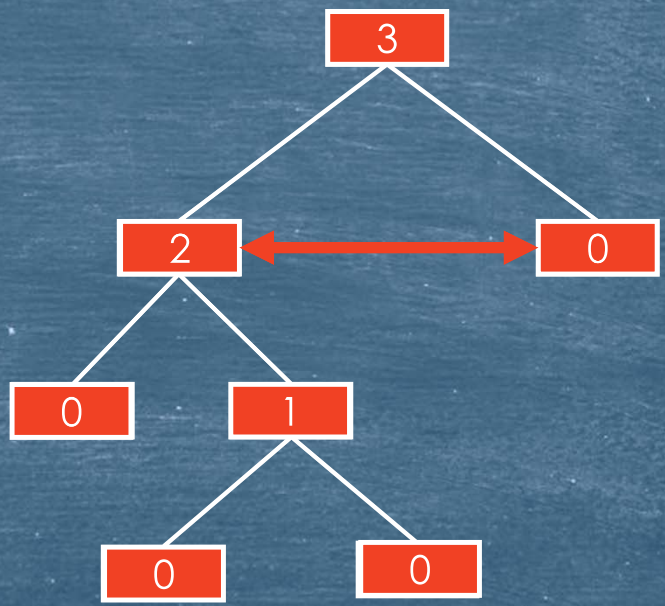
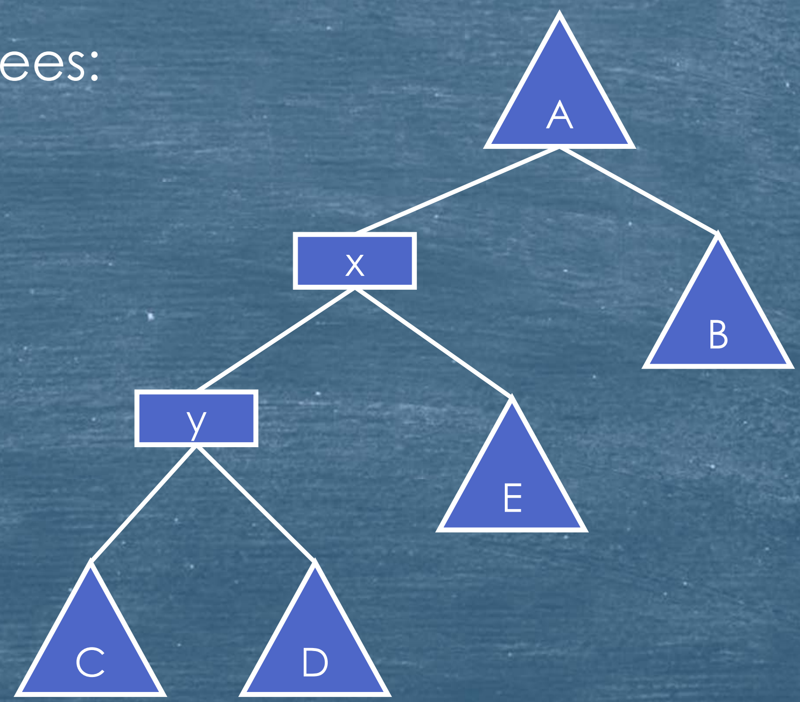
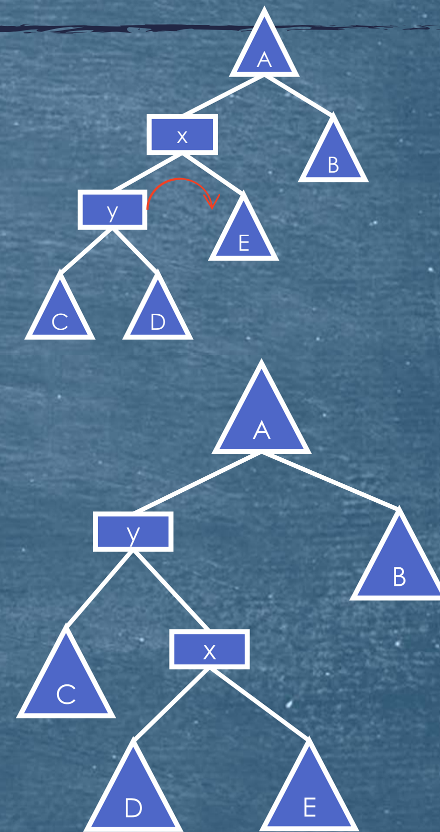
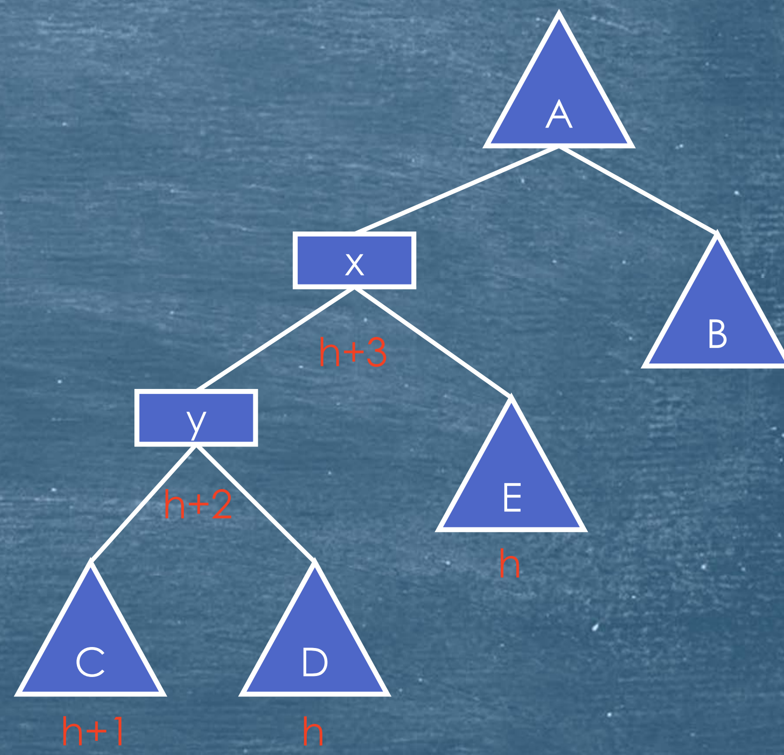
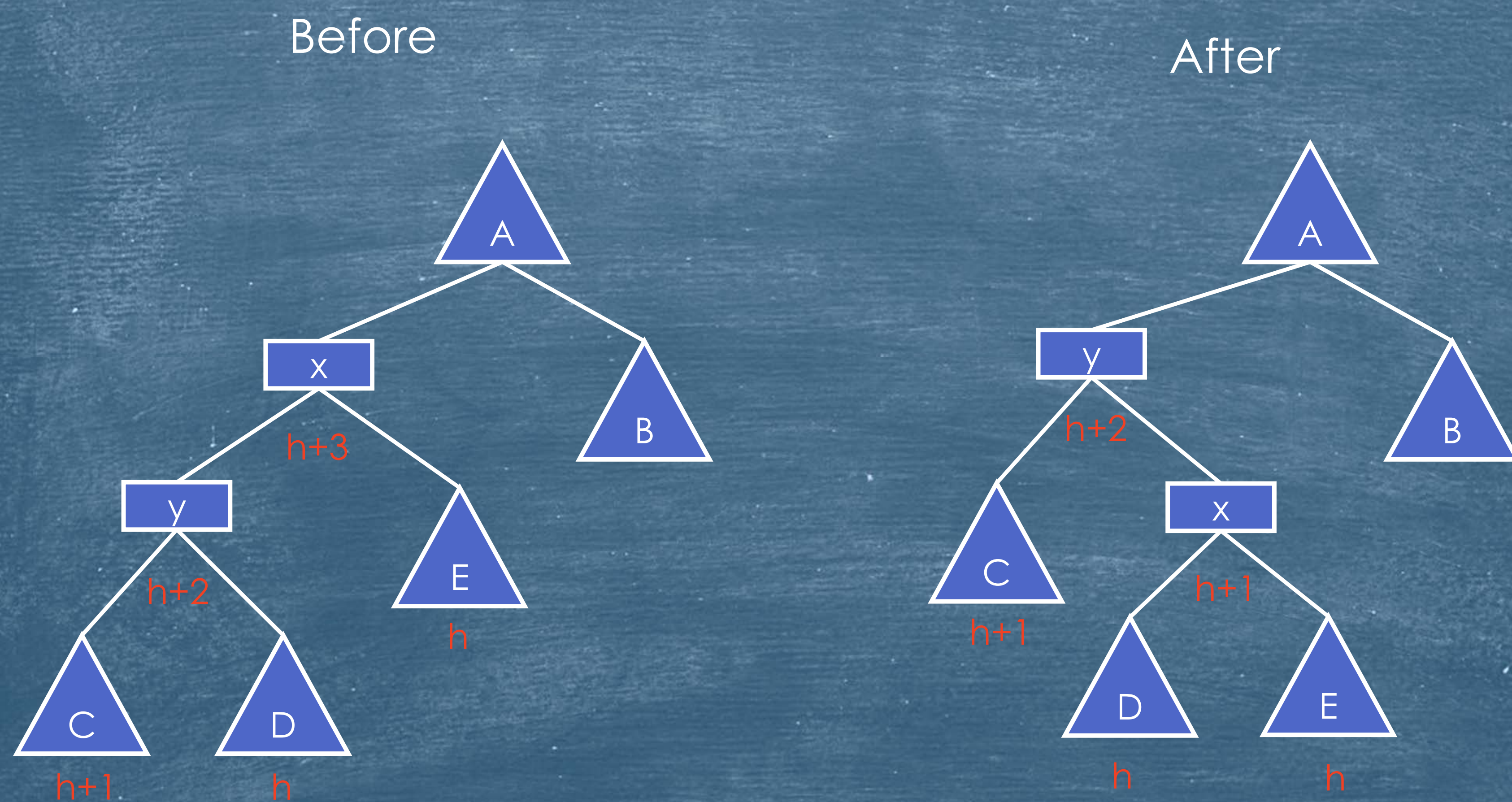
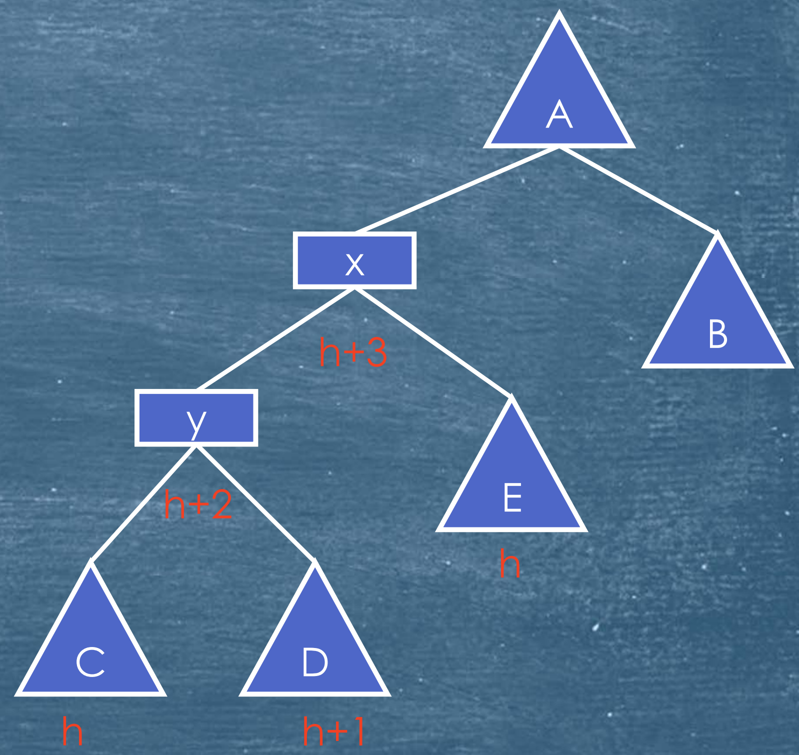
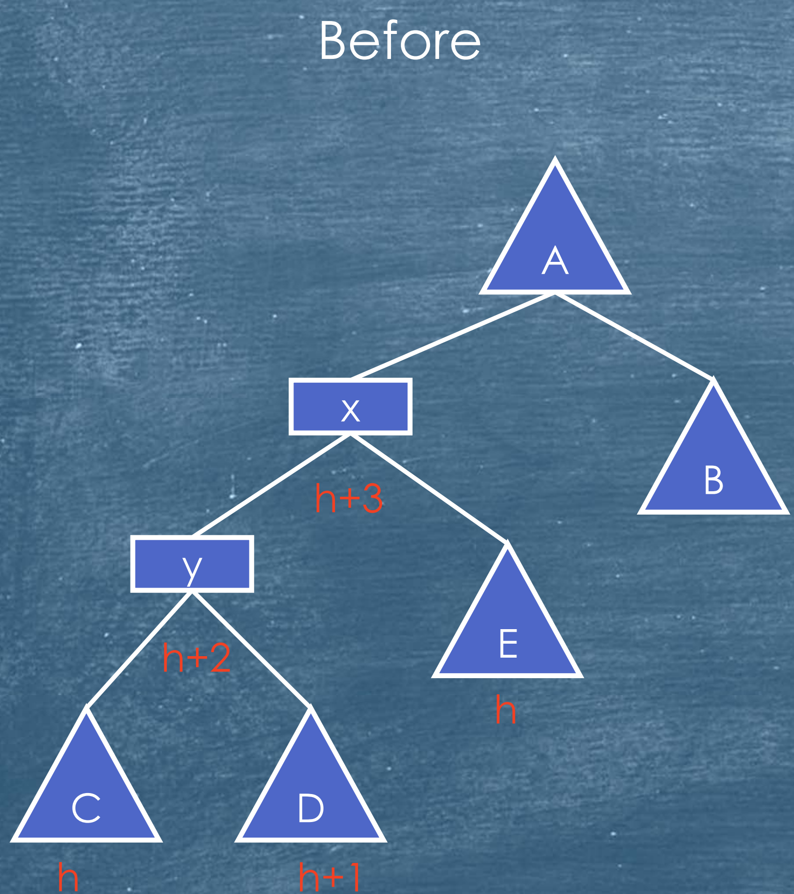
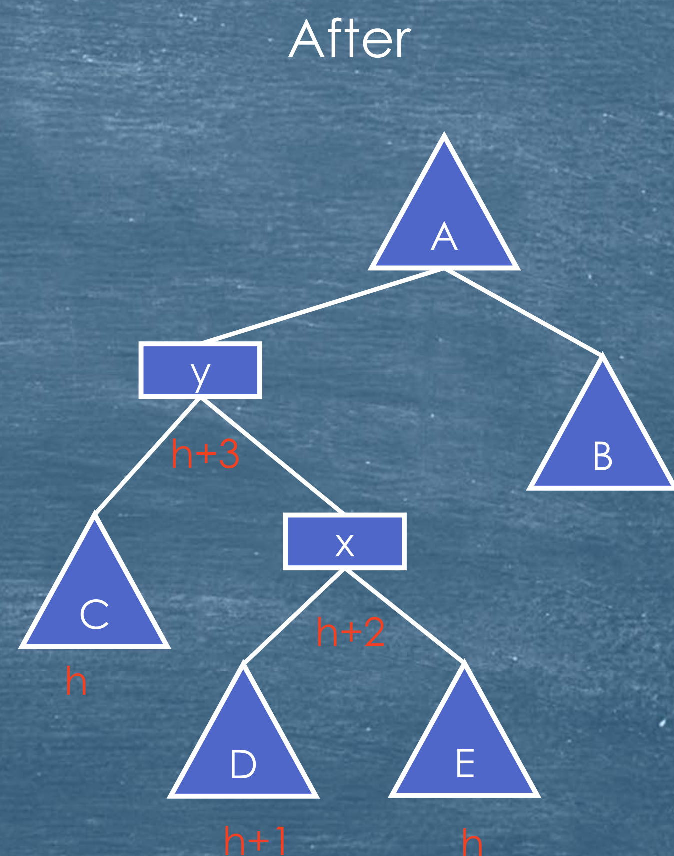
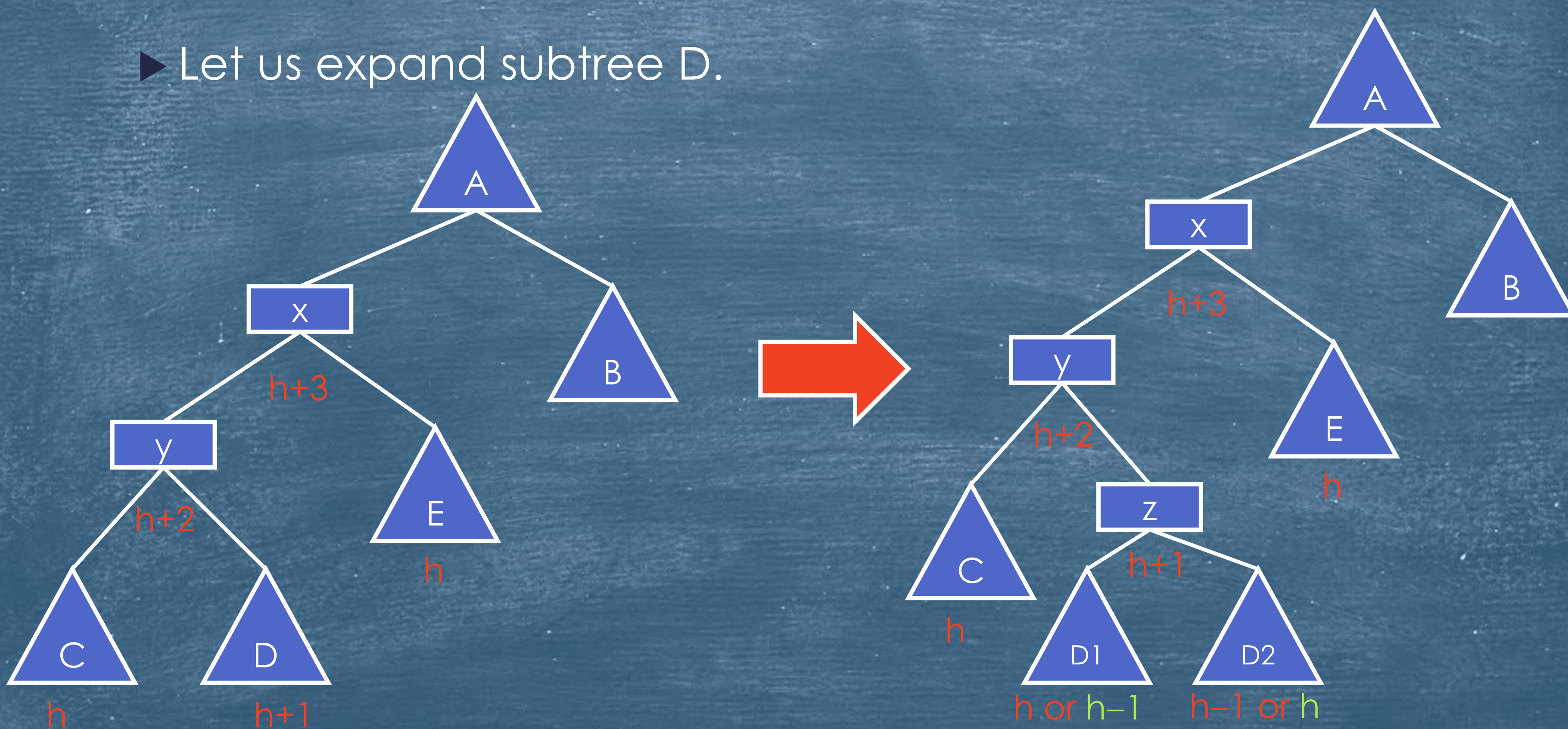
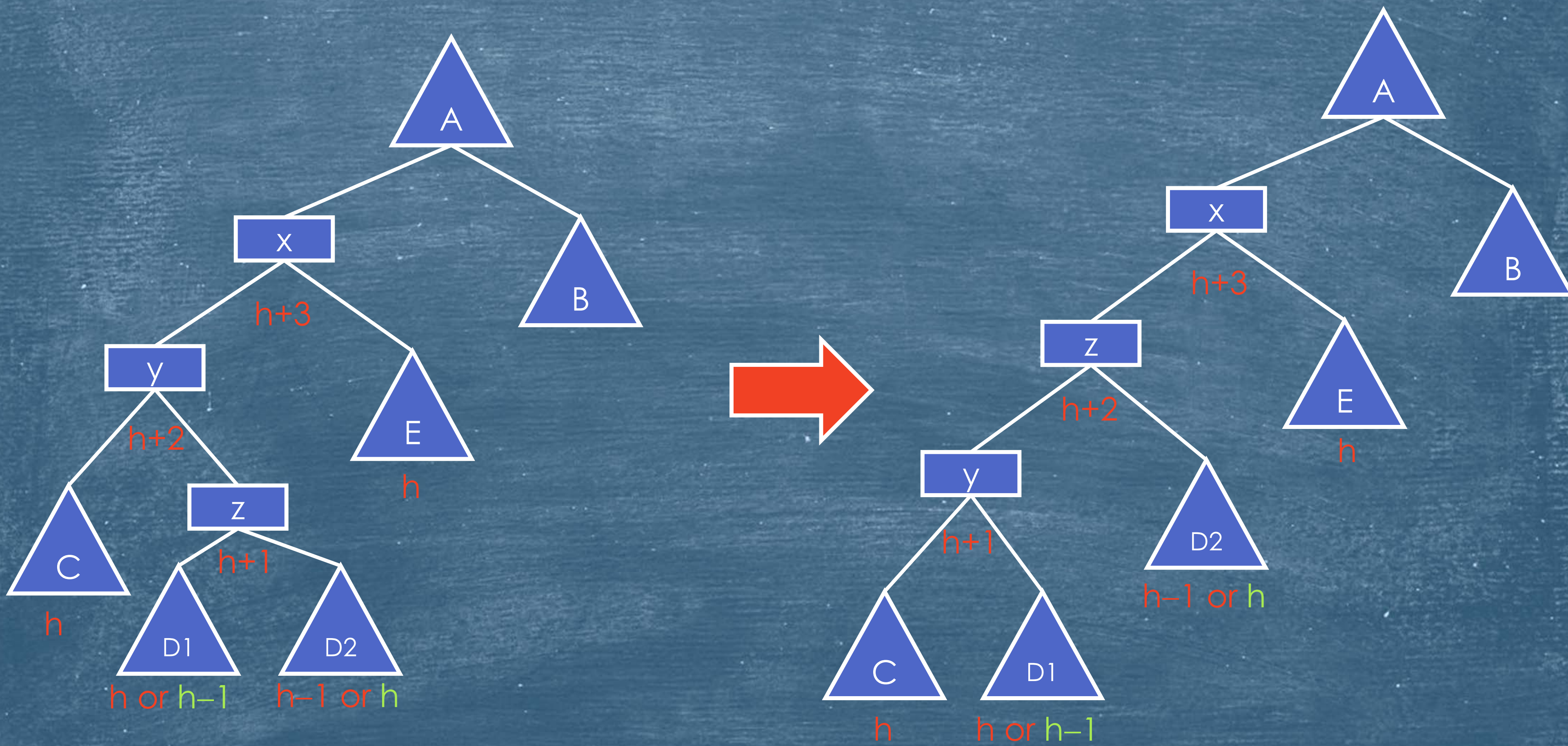
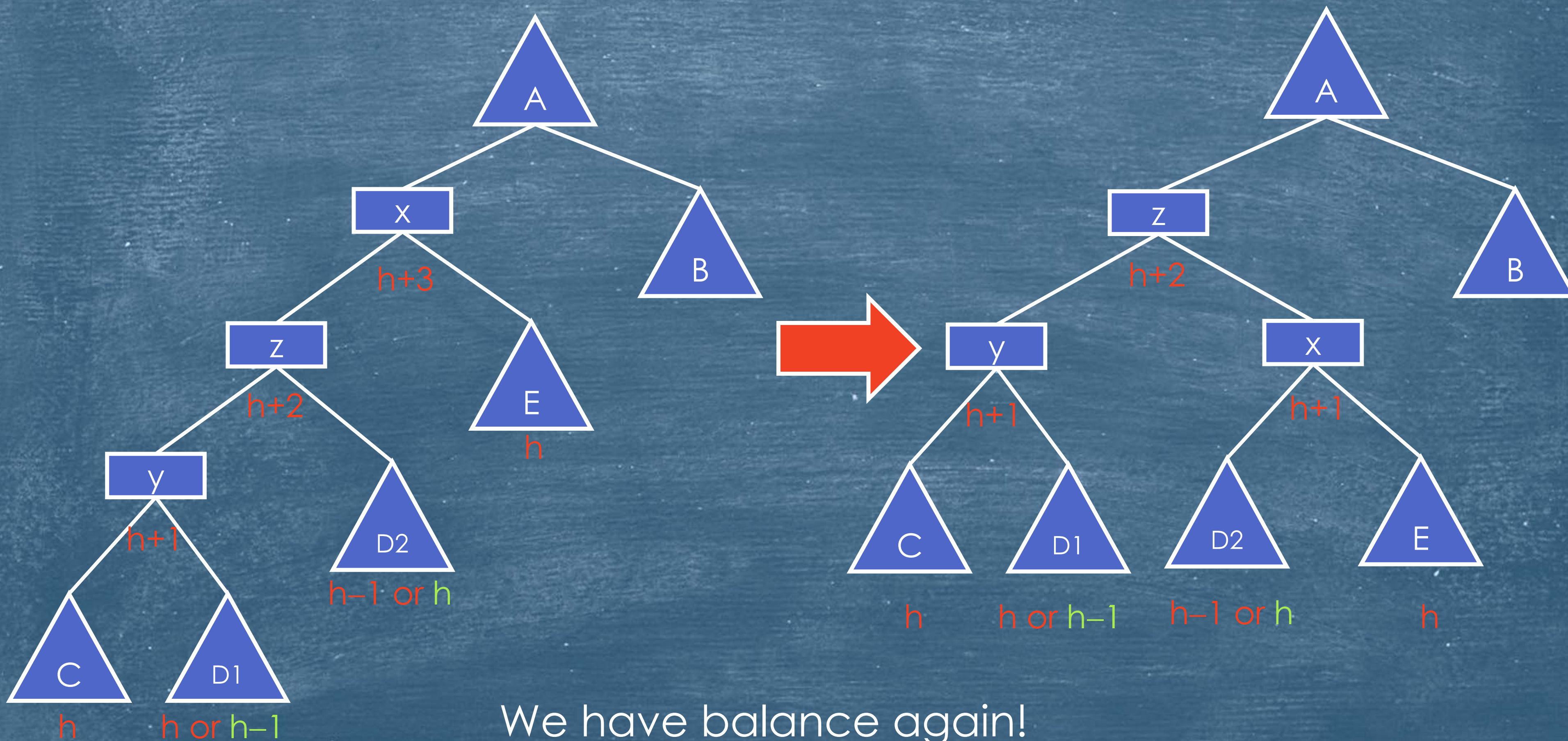
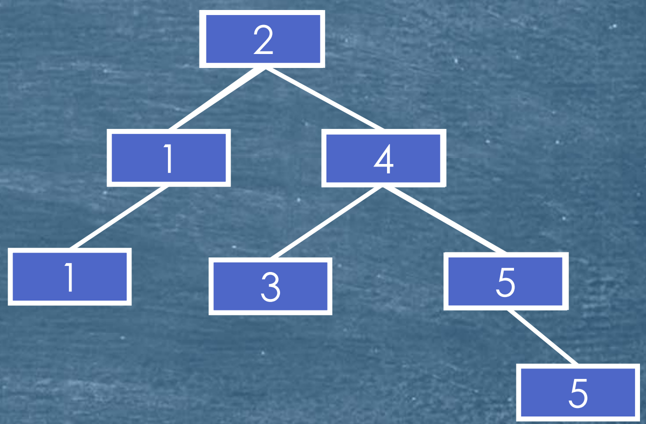
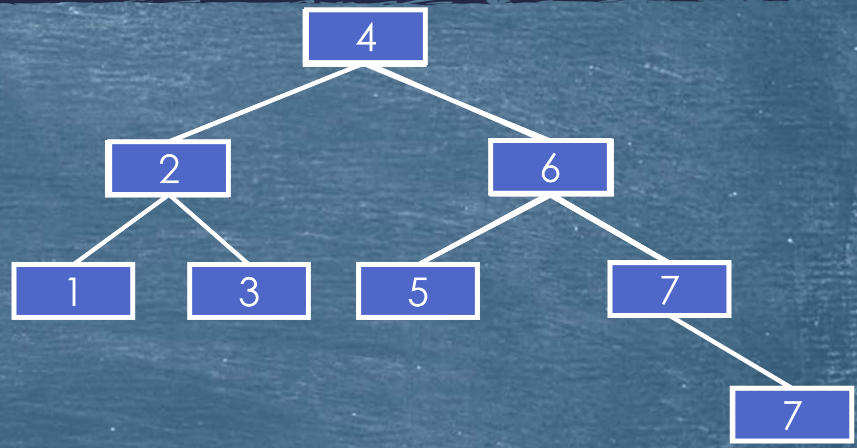
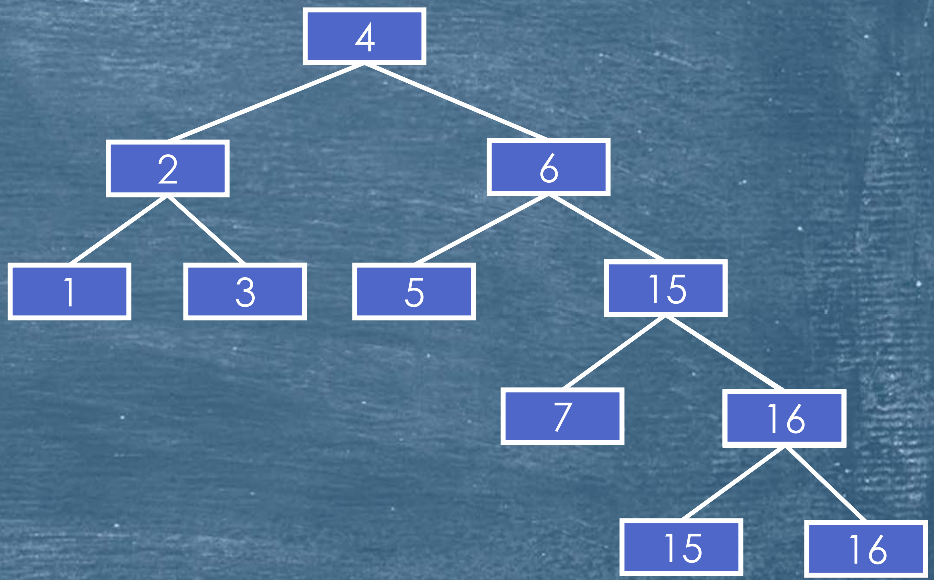
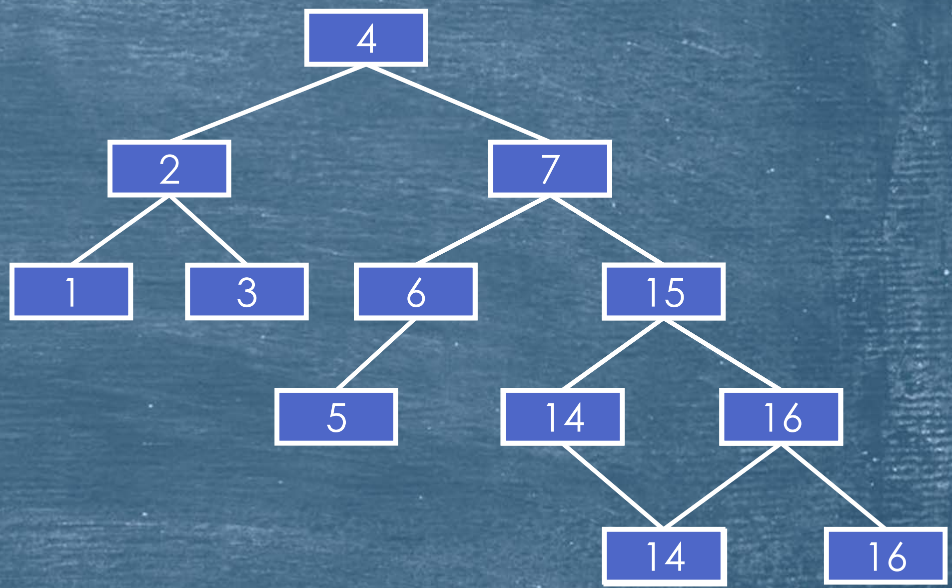
- ```
  procedure find(value: stuff, node: ^tree_node): ^tree_node
      if value == nil then
              return not_found
      fi
      if value == node.contents then
              return node
      else if value < node.contents then
              find(value, node.left)
      else
              find(value, 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 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(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

- ```
  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: 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)`



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.
- 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

- Consider the BST shown to the right:
  - `visit(root)`
  - `visit(root.left)`
  - `visit(node.left)`
  - `print(node.contents)`
  - `return`
  - `print(node.contents)`
  - `visit(node.right)`
  - `visit(node.left)`
  - `print(node.contents)`
  - `return`
  - `print(node.contents)`
  - `return`
  - `return`
  - `print(node.contents)`
  - `visit(node.right)`
  - `visit(node.left)`
  - `print(node.contents)`
  - `return`
  - `print(node.contents)`
  - `return`
  - `return`



5	9	11	13	15	21	33

The Problem with BSTs

- If we create a BST from the following sequence:

1	2	3	4	5

- We get the following BST:



- This tree is severely unbalanced.

Balancing a BST

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.



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

- We define a simple operation on this tree; rotation.
- As we can see:
  - 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!



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:





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:
- Will a single rotation work this time?







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



We have balance again!

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



Continued

- The tree so far
  - Insert 6
    - We have an imbalance at node 2
    - Left rotate node 2
  - Insert 7
    - We have an imbalance at node 5
    - Left rotate node 5



Continued

- 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



Continued

- 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.



AVL Trees – Implementation

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

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
```

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
  ```

- ```
  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
  ```

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.
  - Implementation of this detail is left as an exercise. ☺