APA: Advanced Programming, Algorithms and Data Structures

Emre Güney, PhD

Master in Bioinformatics for Health Sciences
Universitat Pompeu Fabra

Lectures 5-6

October 8-11, 2019

Previously on APA

- Computational complexity
  - Correctness of an algorithm
  - Efficiency of an algorithm
  - (Worst case) Time complexity in terms of input size
  - Big-Oh, Omega and Theta notations (O, Ω, Θ)
  - Space complexity
- Sorting
  - Buble sort, selection sort, insertion sort, merge sort
  - Exhaustive search (brute force) vs divide & conquer
  - Quick sort
- Data structures
  - Linked lists, stacks, queues
  - Priority queues and heaps
- More sorting
  - Heap sort

Binary Heap: implementation

```
# Elements in a Priority Queue must be comparable
# No easy way to check that in Python

class PriorityQueue:

    def __init__(self):
        # Table for the heap (location 0 not used)
        self.__v = ['*']  # one fake element at location 0

    # Number of elements in the queue
    def __len__(self):
        # The 0 location does not count
        return len(self.__v) - 1

    # Checks whether the queue is empty
    def empty(self):
        return len(self) == 0
```


© Dept. CS, UPC

Binary Heap: implementation

```python
# Returns the min element of the queue
def minimum(self):
    assert not self.empty()
    return self.__v[1]

# Inserts an element x into the queue
def insert(self, x):
    self.__v.append(x)          # Put element at the bottom
    self.__bubble_up(len(self)) # ... and bubble up

# Extracts and returns the min element from the queue
def remove_min(self):
    assert not self.empty()
    x = self.__v[1]             # Store the element at the root
    self.__v[1] = self.__v[-1]  # Move the last element to the root
    self.__v.pop()
    self.__bubble_down(1)       # ... and bubble down
    return x
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Binary Heap: implementation

```python
# ------ Private methods ------

# Bubbles up the element at location i
def __bubble_up(self, i):
    if i != 1 and self.__v[i // 2] > self.__v[i]:
        tmp = self.__v[i]
        self.__v[i] = self.__v[i // 2]
        self.__v[i // 2] = tmp
        self.__bubble_up(i // 2)

# Bubbles down the element at location i
def __bubble_down(self, i):
    n = len(self)
    c = 2*i
    if c <= n:
        if c+1 <= n and self.__v[c+1] < self.__v[c]:
            c += 1
        if self.__v[i] > self.__v[c]:
            tmp = self.__v[i]; self.__v[i] = self.__v[c]; self.__v[c] = tmp
            self.__bubble_down(c)
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Building a heap: implementation

```python
    # Constructor from a collection of items
    def __init__(self, items = []):
        self.__v = ['*']  # one fake element at location 0
        for e in items:
            self.__v.append(e)
        for i in range(len(self) // 2, 0, -1):
            self.__bubble_down(i)
```

Sum of the heights of all nodes:

- 1 node with height h
- 2 nodes with height $h-1$
- 4 nodes with height $h-2$
- 2^i nodes with height $h-i$

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \cdots + 2^{h-1}(1)$$

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \cdots + 2^h(1)$$

Subtract the two equations:

$$S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = \left(2^{h+1} - 1\right) - (h+1) = \mathrm{O}(N)$$

A heap can be built from a collection of N items in linear time.

© Dept. CS, UPC

Heap sort

```python
def heapSort(v):            # v is a collection
    p = PriorityQueue(v)
    for i in range(len(v)):
        v[i] = p.remove_min()
```

- Complexity: $O(n \log n)$
  - Building the heap: $O(n)$
  - Each removal is $O(\log n)$, executed n times.

Heap sort

```
def heapSort(v):             # v is a collection
    p = PriorityQueue(v)
    for i in range(len(v)):
        v[i] = p.remove_min()
```

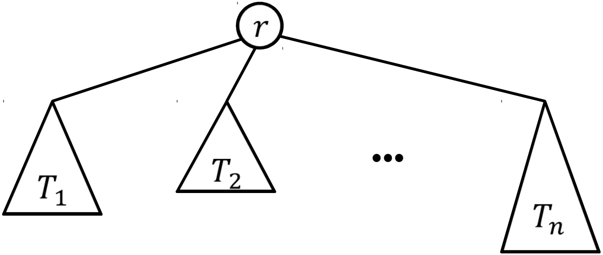
- Complexity: $O(n \log n)$
  - Building the heap: $O(n)$
  - Each removal is $O(\log n)$, executed n times.

Is this in-place sort?

© Dept. CS, UPC

Tree: definition

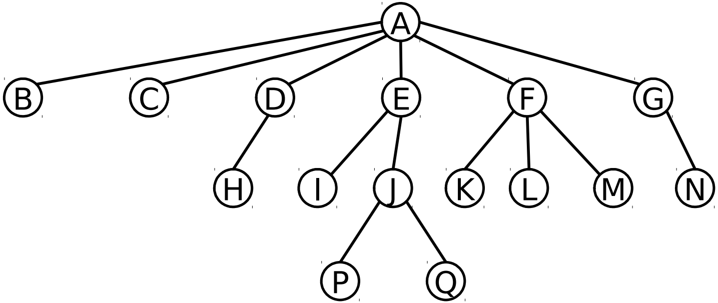
- Graph theory: a tree is an undirected graph in which any two vertices are connected by exactly one path.
  (and no circular paths between any two vertices)
- Recursive definition (CS). A non-empty tree T consists of:
  - a root node r
  - a list of trees T_1, T_2, …, T_n that hierarchically depend on r.



© Dept. CS, UPC

Source: UPC - J. Cortadella & J. Petit

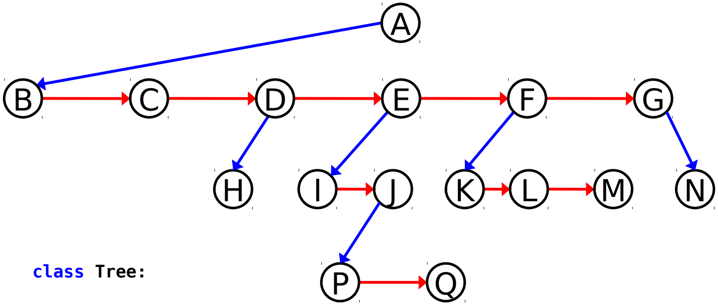
Tree: nomenclature



- A is the **root** node.
- Nodes with no children are **leaves** (e.g., B and P).
- Nodes with the same parent are **siblings** (e.g., K, L and M).
- The **depth** of a node is the length of the path from the root to the node.
  Examples: depth(A)=0, depth(L)=2, depth(Q)=3.

© Dept. CS, UPC

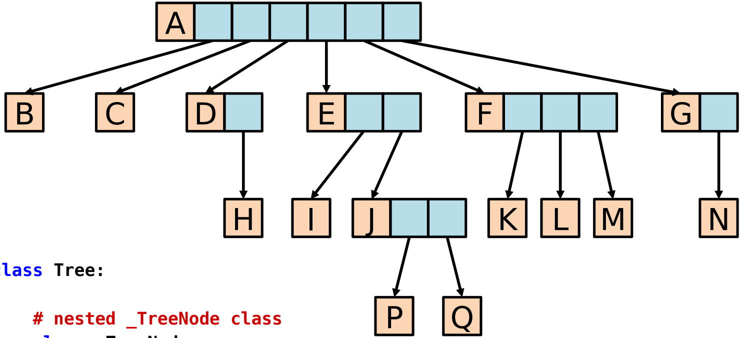
Tree: representation with linked lists




```
class Tree:

    # nested _TreeNode class
    class _TreeNode:
        __slots__ = '_element', '_children'

        def __init__(self, elem):
            self._element  = elem
            self._children = List() # linked list of children

    # tree methods
    (. . .)
```

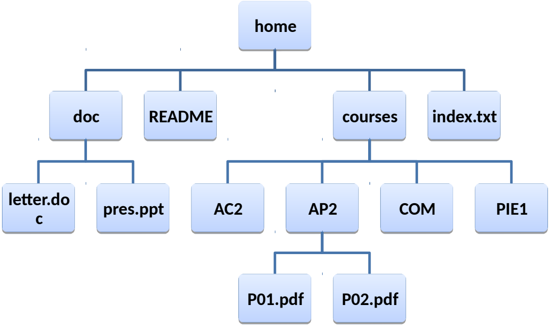

© Dept. CS, UPC

Tree: representation with vectors




```python
class Tree:

    # nested _TreeNode class
    class _TreeNode:
        __slots__ = '_element', '_children'

        def __init__(self, elem):
            self._element  = elem
            self._children = []   # vector (Python list) of children

    # tree methods
    (. . .)
```

Print a tree




```
home
  doc
    letter.doc
    pres.ppt
  README
  courses
    AC2
    AP2
      P01.pdf
      P02.pdf
    COM
    PIE1
  index.txt
```

```python
class Tree:
    __slots__ = '__name', '__children'

    def __init__(self, str):
        self.__name     = str
        self.__children = []

    (. . .)
```

```
printTree(t, depth=0)
```

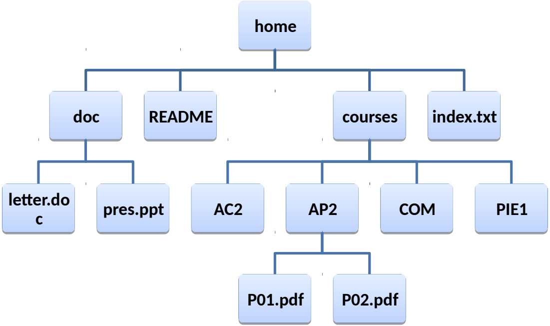
Print a tree

```python
# Prints a tree indented according to depth.
# * Pre: The tree is not empty. *
def printTree(t, depth=0):
    assert not t.empty()

    # Print the root indented by 2*depth
    print(' '*2*depth, end='', flush=True)
    print(t.name())

    # Print the children with depth + 1
    for child in t.children():
        printTree(child, depth + 1)
```

This function executes a ***preorder*** traversal of the tree: each node is processed *before* the children.

© Dept. CS, UPC

Print a tree (postorder traversal)




```
    letter.doc
    pres.ppt
  doc
  README
    AC2
      P01.pdf
      P02.pdf
    AP2
    COM
    PIE1
  courses
  index.txt
home
```

Postorder traversal: each node is processed after the children.

Print a tree (postorder traversal)

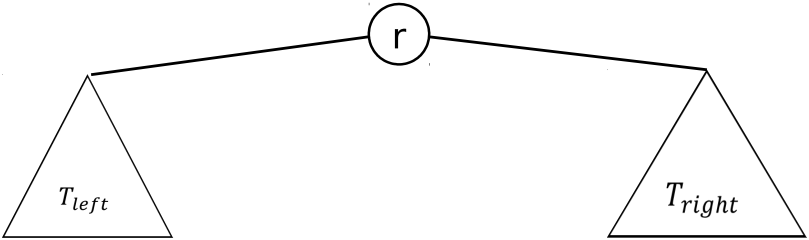
```
# Prints a tree (in postorder) indented according to depth.
# * Pre: The tree is not empty. *
def printPostOrder(t, depth=0):
    assert not t.empty()

    # Print the children with depth + 1
    for child in t.children():
        printPostOrder(child, depth + 1)

    # Print the root indented by 2*depth
    print(' '*2*depth, end='', flush=True)
    print(t.name())
```

This function executes a ***postorder*** traversal of the tree: each node is processed *after* the children.

© Dept. CS, UPC

Binary trees

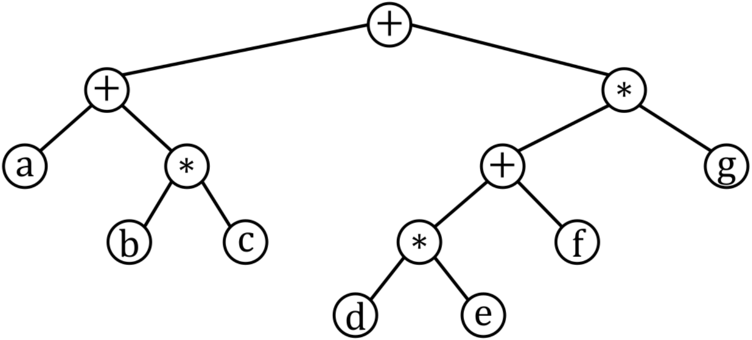
Nodes with at most two children.




```
class BinaryTree:
    __slots__ = '__elem', '__left', '__right'

    def __init__(self, e, l=None, r=None):
        self.__elem  = e
        self.__left  = l
        self.__right = r

    (. . .)
```


© Dept. CS, UPC

Example: expression trees



Expression tree for: **a + b∗c + (d∗e + f) ∗ g**

Postfix representation: **a b c ∗ + d e ∗ f + g ∗ +**

How can the postfix representation be obtained?

© Dept. CS, UPC

Example: expression trees

```
class ExprTree:
    __slots__ = '__op',\
                '__left',\
                '__right'
```

Expressions are represented by strings in **postfix** notation in which the characters 'a'…'z' represent operands and the characters '+' and '∗' represent operators.

```
# ===> Methods of ExprTree class <===

# Builds an expression tree from a correct
# expression represented in postfix notation.
@classmethod
def buildExpr(cls, strexpr):

# Generates a string with the expression in infix notation
def infixExpr(self):

# Evaluates an expression taking mapvals as the value of the
# variables (e.g., mapvals['a'] contains the value of a).
def evalExpr(self, mapvals):
```


© Dept. CS, UPC

Example: expression trees

```
class ExprTree:
    __slots__ = '__op', '__left', '__right'

    def __init__(self, op, l=None, r=None):
        self.__op    = op
        self.__left  = l
        self.__right = r

    def op(self):
        return self.__op

    def left(self):
        return self.__left

    def right(self):
        return self.__right

    def leaf(self):
        return self.__left == None and self.__right == None
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Example: expression trees

```
@classmethod
def buildExpr(cls, strexpr):
    s = Stack()

    for c in strexpr:
        if c >= 'a' and c <= 'z':
            # We have an operand in {'a'...'z'}. Create a leaf node.
            s.push(cls(c))
        else:
            # c is an operator ('+' or '*')
            right = s.top()
            s.pop()
            left = s.top()
            s.pop()
            s.push(cls(c, left, right))

    # The stack has only one element
    return s.top()
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Example: expression trees

```python
# Generates a string with the expression in infix notation
def infixExpr(self):

    # Let us first check the base case (an operand)
    if self.leaf():
        return self.op()

    # We have an operator. Return "( left op right )"
    return "(" + self.left().infixExpr() +\
        " " + self.op() + " " +\
        self.right().infixExpr() + ")
```

Inorder traversal: node is visited *between* the left and right children.

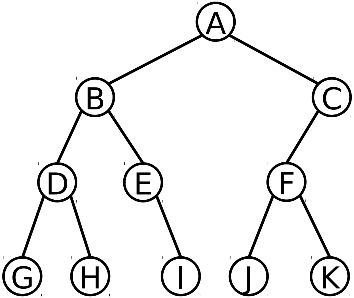
© Dept. CS, UPC

Example: expression trees

```
# Evaluates an expression taking mapvals as the value of the
# variables (e.g., mapvals['a'] contains the value of a).
def evalExpr(self, mapvals):
    if self.leaf():
        return mapvals[self.op()]
    l = self.left().evalExpr(mapvals)
    r = self.right().evalExpr(mapvals)
    return l + r if self.op() == '+' else l * r

# ------------------------------
# Example of usage of ExprTree.
# ------------------------------
def main():
    t = ExprTree.buildExpr("abc*+de*f+g*+")
    print(t.infixExpr())
          # ===> ((a + (b * c)) + (((d * e) + f) * g))
    print("Eval = ", t.evalExpr({'a': 3, 'b': 1, 'c': 0, 'd': 5,
                                 'e': 2, 'f': 1, 'g': 6}))
          # ===> Eval =  69
```


Source: UPC - J. Cortadella & J. Petit

Tree traversals



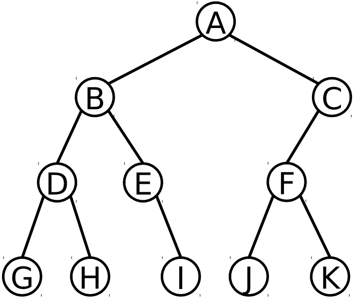
Traversal: algorithm to visit the nodes of a tree in some specific order.

The actions performed when visiting each node can be a parameter of the traversal algorithm.

```
# traversal is a higher order function since
# visitor must be a function

def traversal(tree, visitor):
```


Source: UPC - J. Cortadella & J. Petit

Tree traversals

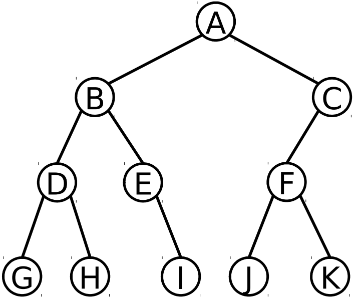


Preorder: A B D G H E I C F J K

```
def preorder(tree, visitor):
    if tree is not None:
        visitor(tree.elem())
        preorder(tree.left(), visitor)
        preorder(tree.right(), visitor)
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Tree traversals

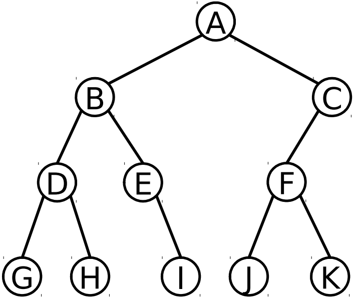


Preorder: A B D G H E I C F J K

Postorder: G H D I E B J K F C A

```
def postorder(tree, visitor):
    if tree is not None:
        postorder(tree.left(), visitor)
        postorder(tree.right(), visitor)
        visitor(tree.elem())
```

Tree traversals



Preorder: A B D G H E I C F J K

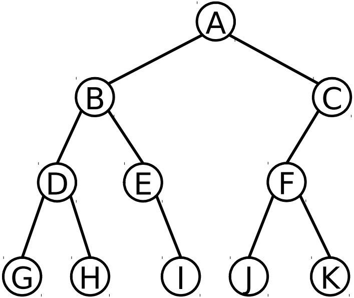
Postorder: G H D I E B J K F C A

Inorder: G D H B E I A J F K C

```
def inorder(tree, visitor):
    if tree is not None:
        inorder(tree.left(), visitor)
        visitor(tree.elem())
        inorder(tree.right(), visitor)
```


© Dept. CS, UPC

Tree traversals



Preorder: A B D G H E I C F J K

Postorder: G H D I E B J K F C A

Inorder: G D H B E I A J F K C

By levels: A B C D E F G H I J K

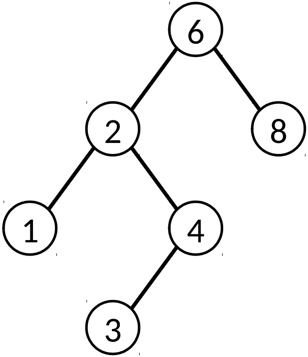
```
def byLevels(tree, visitor):
    q = Queue();  q.enqueue(tree)
    while not q.empty():
        t = q.first(); q.dequeue()
        if t is not None:
            visitor(tree.elem())
            q.enqueue(tree.left())
            q.enqueue(tree.right())
```


© Dept. CS, UPC

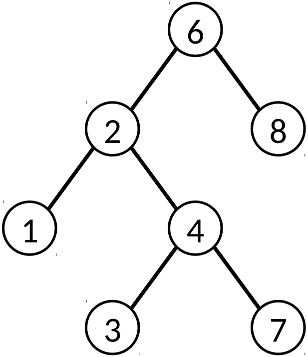
Binary Search Trees

BST property: for every node in the tree with value V:

- All values in the left subtree are smaller than V.
- All values in the right subtree are larger than V.



This is a binary search tree



This is *not* a binary search tree

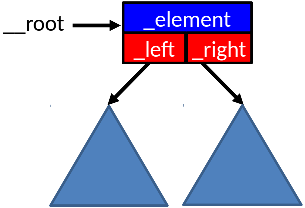
BST: public methods

```
class Set:
    # Constructor
    def __init__(self):

    # Finding elements
    def findMin(self):
    def findMax(self):
    def contains(self, x):
    def __len__(self):
    def isEmpty(self):

    # Insert/remove methods
    def insert(self, x):
    def remove(self, x):
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

BST: internal implementation

```
class _Node:   # nested class _Node
    def __init__(self, elem, left = None, right = None):
        self._element = elem # The element stored in the node
        self._left = left    # Pointer to the left subtree
        self._right =right   # Pointer to the right subtree

def __init__ (self):          # Class Set constructor
    self.__root = None        # Pointer to the root of the tree
    self.__n    = 0           # Number of elements
```



© Dept. CS, UPC

Source: UPC - J. Cortadella & J. Petit

BST: private methods

```
# Public methods require a private pointer-based
# version to traverse the tree.

    # Finding elements
    def __findMin(self, t):
    def __findMax(self, t):
    def __contains(self, x, t):

    # Insert/remove methods
    def __insert(self, x, t):
    def __remove(self, x, t):
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

findMin (recursive) and findMax (iterative)

```
# Find the smallest item in the (non-empty) subtree t.
# Returns (a ptr to) the node with the smallest item.
def __findMin(self, t):
    if t._left == None:
        return t
    return self.__findMin(t._left)
# Find the largest item in the (non-empty) subtree t.
# Returns (a ptr to) the node with the largest item.
def __findMax(self, t):
    tt = t
    while tt._right != None:
        tt = tt._right
    return tt
```

private

```
# Find the smallest item in the (non-empty) Set.
def findMin(self):
    assert not self.isEmpty()
    return self.__findMin(self.__root)._element
# findMax has a similar implementation
```

public

© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

contains and isEmpty

```
# Find an item in the subtree represented by t.
# Returns true if found, and false otherwise.
def __contains(self, x, t):
    if t == None:
        return False
    if x < t._element:
        return self.__contains(x, t._left)
    if x > t._element:
        return self.__contains(x, t._right)
    return True
```

private

```
# Find an item in the set.
# Returns true if found, and false otherwise.
def contains(self, x):
    return self.__contains(x, self.__root)
# Checks whether the tree is empty.
def isEmpty(self):
    return self.__root == None
```

public

© Dept. CS, UPC

insert

```python
# Inserts item x into the subtree t.
def __insert(self, x, t):
    if t == None:
        self.__n += 1
        return self._Node(x)
    elif x < t._element:
        t._left = self.__insert(x, t._left)
    elif x > t._element:
        t._right = self.__insert(x, t._right)
    return t
```

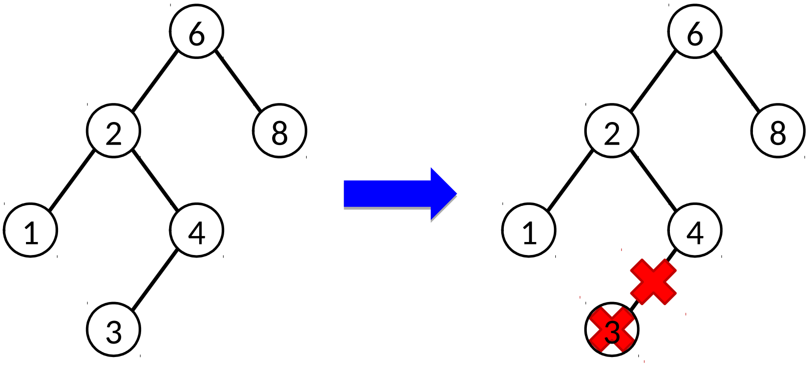
private

```python
# Inserts item x into the set.
def insert(self, x):
    self.__root = self.__insert(x, self.__root)
```

public

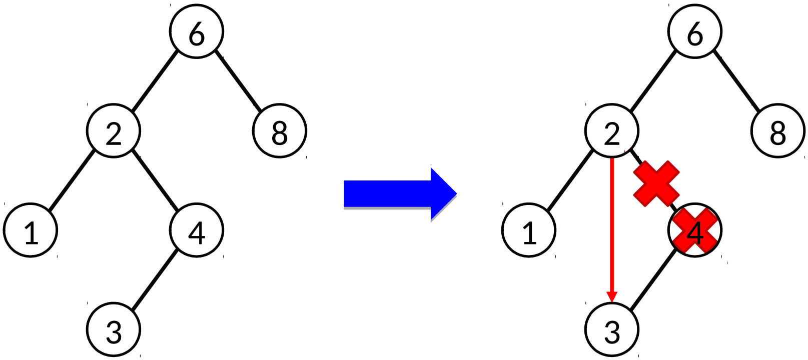
© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

remove: simple case (no children)



`remove(3)`

remove: simple case (one child)



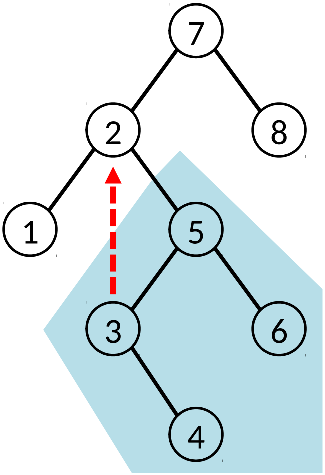
`remove(4)`

remove: simple cases

```
def __remove(self, x, t):
    if t != None:       # if t == None, x not in the Set. Do nothing
        if x < t._element:
            t._left = self.__remove(x, t._left)
        elif x > t._element:
            t._right = self.__remove(x, t._right)
        else:  # x == t._element
            # t has 0 or 1 child
            if t._left == None:
                self.__n -= 1
                return t._right
            elif t._right == None:
                self.__n -= 1
                return t._left
            else: # t has 2 childs
                  # (. . .)
        return t
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

remove: complex case (two children)

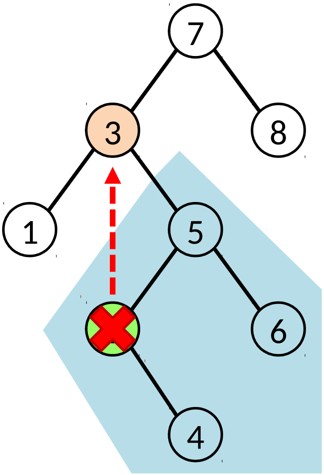


`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.

© Dept. CS, UPC

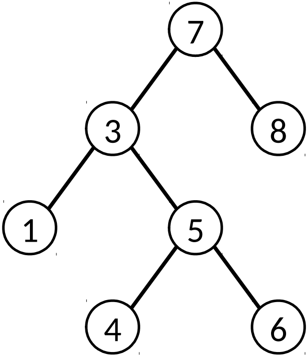
remove: complex case (two children)



`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

remove: complex case (two children)



1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

```
remove(2)
```

remove: complex case (two children)

```
# Removes item x from the subtree t.
def __remove(self, x, t):
    if t != None:  # if t == None, x not in the Set. Do nothing
        if         # (. . .)
        elif       # (. . .)
        else:      # x == t._element
            if     # (. . .)
            else:  # t has 2 childs
                # Find the minimum element in the right child
                m = self.__findMin(t._right)._element
                # copy it as element of _Node t
                t._element = m
                # then remove it from the right child
                t._right = self.__remove(m, t._right)
        return t
```


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

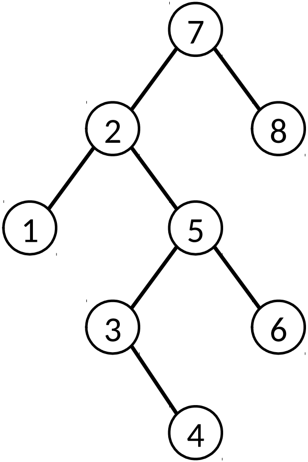
remove: all cases

```python
# Removes item x from the subtree t.
def __remove(self, x, t):
    if t != None:       # if t == None, x not in the Set. Do nothing
        if x < t._element:
            t._left = self.__remove(x, t._left)
        elif x > t._element:
            t._right = self.__remove(x, t._right)
        else:  # x == t._element
            if t._left == None:
                self.__n -= 1; return t._right
            elif t._right == None:
                self.__n -= 1; return t._left
            else:  # t has 2 childs
                m = self.__findMin(t._right)._element
                t._element = m
                t._right = self.__remove(m, t._right)
        return t

# Public method for remove.
def remove(self, x):
    self.__root = self.__remove(x, self.__root)
```


© Dept. CS, UPC

Source: UPC - J. Cortadella & J. Petit

Visiting the items in ascending order



Question:

How can we visit the items of a BST in ascending order?

Answer:

Using an in-order traversal

© Dept. CS, UPC

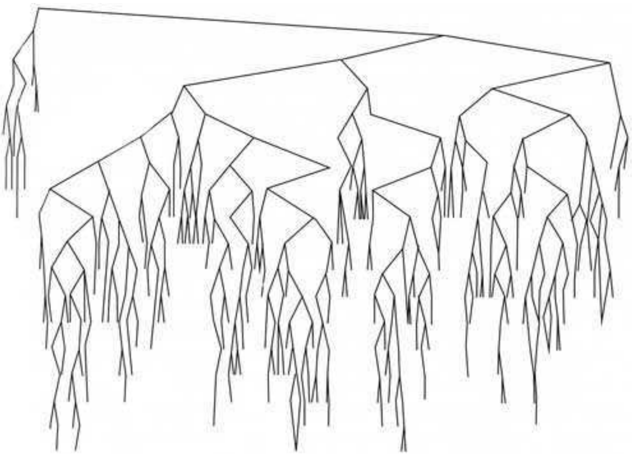
BST: runtime analysis

- We are mostly interested in the runtime of the `insert/remove/contains` methods.
  - The complexity is $\mathcal{O}(d)$, where *d* is the depth of the node containing the required element.

- But, how large is *d*?

© Dept. CS, UPC

Random BST



Source: Fig 4.29 of Weiss textbook

BST: runtime analysis

Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.

- $D(n)$ is the IPL of a tree with n nodes. $D(1) = 0$. The left subtree has i nodes and the right subtree has $n - i - 1$ nodes. Thus,

$$D(n) = D(i) + D(n - i + 1) + (n - 1)$$

- If all subtree sizes are equally likely, then the average value for $D(i)$ and $D(n - i - 1)$ is

$$\frac{1}{n}\sum_{j=0}^{n-1} D(j)$$

© Dept. CS, UPC

BST: runtime analysis

- Therefore,

$$D(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} D(j)\right] + n - 1$$

- The previous recurrence gives:

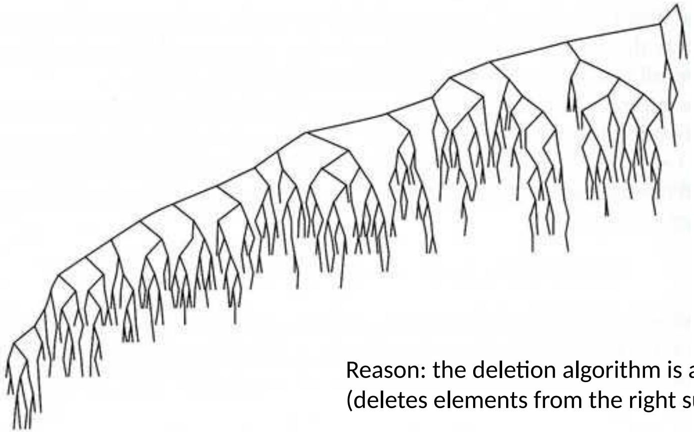
$$D(n) = \mathrm{O}(n \log n)$$

- The average height of nodes after n random insertions is $\mathrm{O}(\log n)$.

- However, the $\mathrm{O}(\log n)$ average height is not preserved when doing deletions.

© Dept. CS, UPC

Random BST after n^2 insert/removes

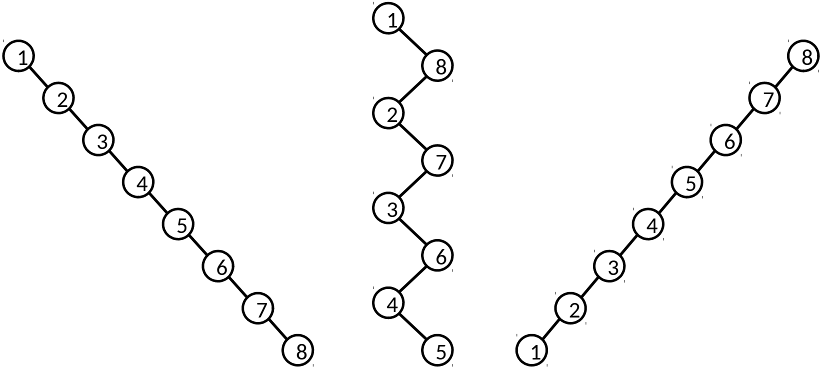


Reason: the deletion algorithm is asymmetric (deletes elements from the right subtree)

Source: Fig 4.30 of Weiss textbook

© Dept. CS, UPC

Worst-case runtime: $\mathcal{O}(n)$



© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Balanced trees

- The worst-case complexity for insert, remove and search operations in a BST is $O(n)$, where n is the number of elements.

- Various representations have been proposed to keep the height of the tree as $O(\log n)$:
  - AVL trees
  - Red-Black trees
  - Splay trees
  - B-trees

Source: UPC - J. Cortadella & J. Petit

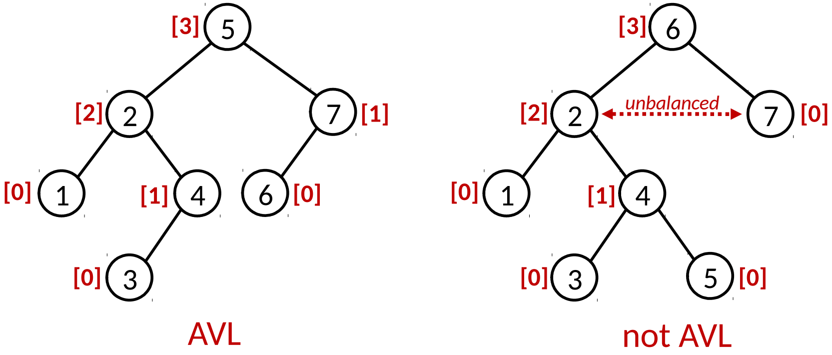
AVL trees

- Named after Adelson-Velsky and Landis (1962).
- Main idea: invest some additional time to balance the tree each time a new element is inserted or deleted.
- Properties:
  - The height of the tree is always $\Theta(\log n)$.
  - The time devoted to balancing is $O(\log n)$.

© Dept. CS, UPC

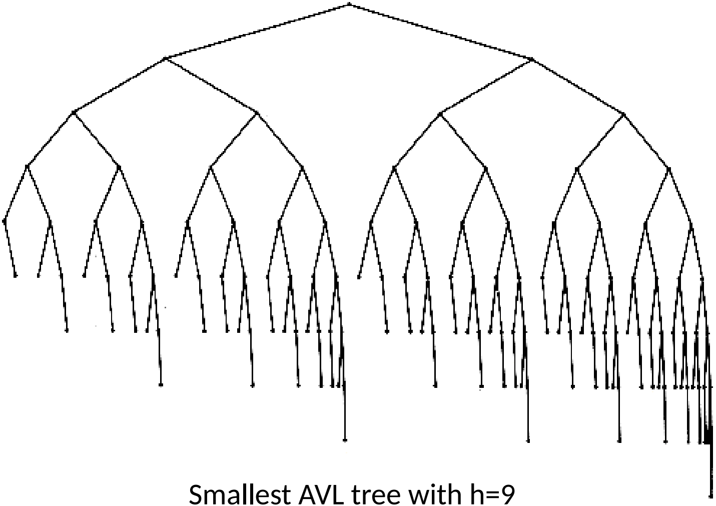
AVL tree: definition

- An AVL tree is a BST such that, for every node, the difference between the heights of the left and right subtrees is at most 1.



© Dept. CS, UPC

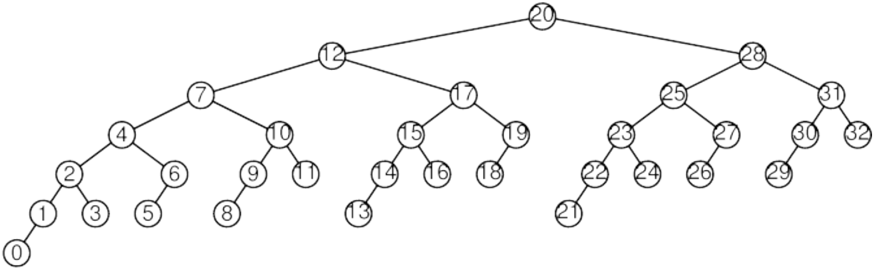
AVL trees



Smallest AVL tree with h=9

Source: UPC - J. Cortadella & J. Petit

AVL trees



Smallest AVL tree with h=6

The important question: what is the size of an AVL tree with height h?

© Dept. CS, UPC

Height of an AVL tree

- Theorem: The height of an AVL tree with n nodes is $\Theta(\log n)$.

- Proof in two steps:
  - The height is $\Omega(\log n)$.
  - The height is $O(\log n)$.

© Dept. CS, UPC

The height is $\Omega(\log n)$

The size n of a tree with height h is:

$$n \leq 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1.$$

(all levels full of nodes)

- Therefore,

$$\log_2(n+1) - 1 \leq h$$

and $h = \Omega(\log n)$.

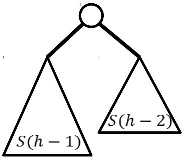
© Dept. CS, UPC

The height is $\mathcal{O}(\log n)$

Let $S(h)$ be the min number of nodes of an AVL tree with height h.

- One of the children (e.g., left) must have height $h-1$. The other child must have height $h-2$ (because the AVL has min size).

- Therefore,
$$S(h) = S(h-1) + S(h-2) + 1.$$



- Thus,
$$S(h) \geq 2 \cdot S(h-2).$$

- Given that $S(0) = 1$ and $S(1) = 2$, it can be easily proven, by induction, that:
$$S(h) \geq 2^{h/2}$$

- Since $n \geq S(h)$ and $\log_2 S(h) \geq h/2$, then $h \leq 2\log_2 n$:
$$h = O(\log n).$$

© Dept. CS, UPC

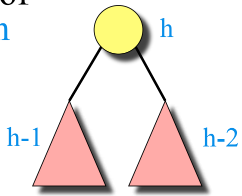
AVL trees have height $\Theta(\log n)$

Invariant: for every node x, the heights of its left child and right child differ by at most 1

- Let n_h be the minimum number of nodes of an AVL tree of height h
- We have $n_h \geq 1 + n_{h-1} + n_{h-2}$

$$\Rightarrow n_h > 2n_{h-2}$$
$$\Rightarrow n_h > 2^{h/2}$$
$$\Rightarrow h < 2 \lg n_h$$

- The constant "2" can be improved



How can we maintain the invariant ?

Source: MIT OpenCourseWare

Height of an AVL tree

- The recurrence

$$S(h) = S(h-1) + S(h-2) + 1$$

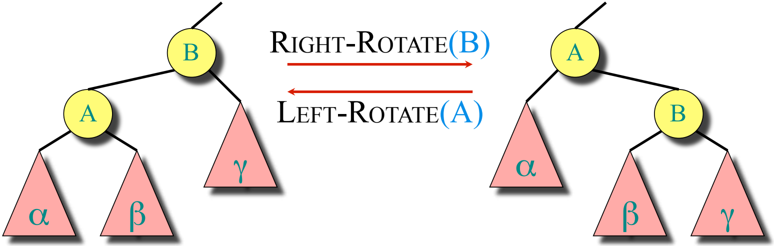
  resembles the one of the Fibonacci numbers. A tighter bound can be obtained.

- Theorem: the height of an AVL tree with n internal nodes satisfies:

$$h < 1.44 \log_2(n+2) - 1.328$$

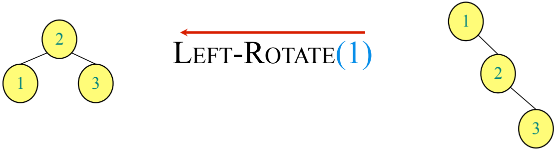
© Dept. CS, UPC

Rotations



Rotations maintain the inorder ordering of keys:

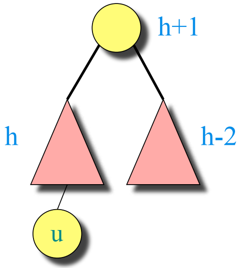
- $a \in \alpha, b \in \beta, c \in \gamma \;\Rightarrow\; a \le A \le b \le B \le c$.



Source: MIT OpenCourseWare

Insertions

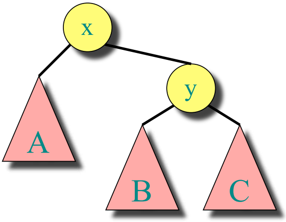
- Insert new node u as in the simple BST
  - Can create imbalance
- Work your way up the tree, restoring the balance
- Similar issue/solution when deleting a node



Source: MIT OpenCourseWare

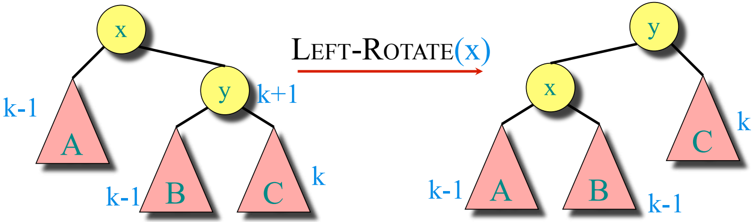
Balancing

- Let x be the lowest "violating" node
  - We will fix the subtree of x and move up
- Assume the right child of x is deeper than the left child of x (x is "right-heavy")
- Scenarios:
  - Case 1: Right child y of x is right-heavy
  - Case 2: Right child y of x is balanced
  - Case 3: Right child y of x is left-heavy



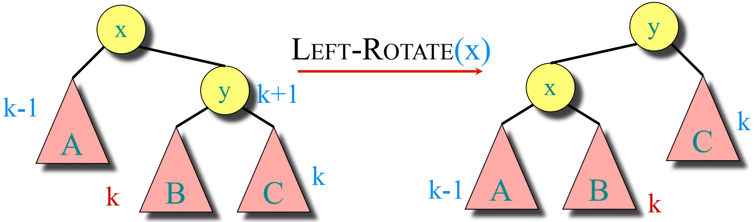
Source: MIT OpenCourseWare

Case 1: y is right-heavy



Source: MIT OpenCourseWare

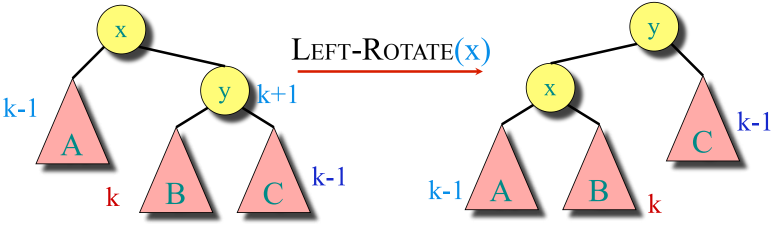
Case 2: y is balanced



Same as Case 1

Source: MIT OpenCourseWare

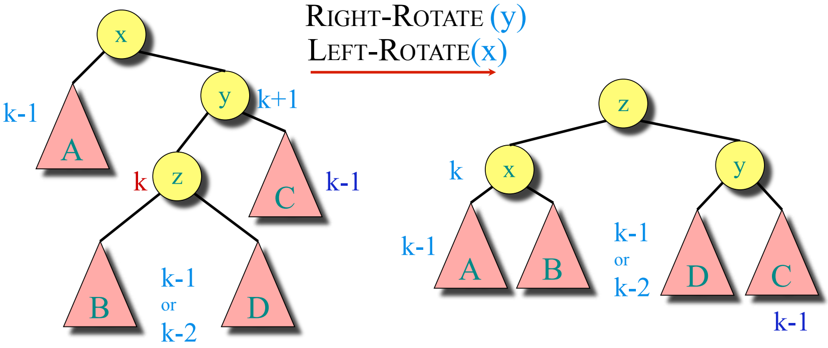
Case 3: y is left-heavy



Need to do more …

Source: MIT OpenCourseWare

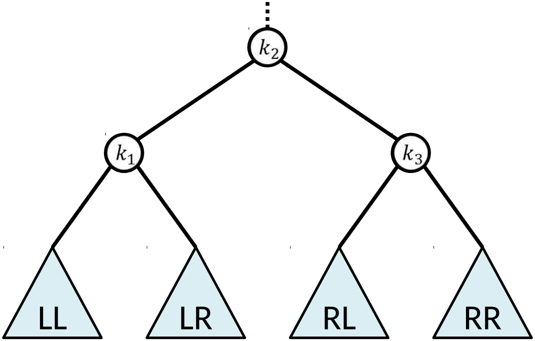
Case 3: y is left-heavy



And we are done!

Source: MIT OpenCourseWare

Unbalanced insertion: 4 cases



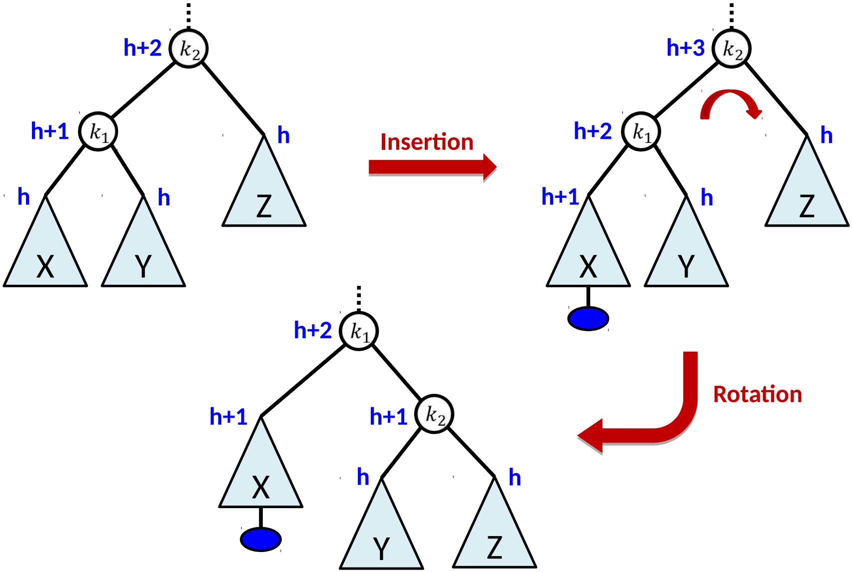
Any newly inserted item may fall into any of the four subtrees (LL, LR, RL or RR).

A new insertion may violate the balancing property. Re-balancing might be required.

© Dept. CS, UPC

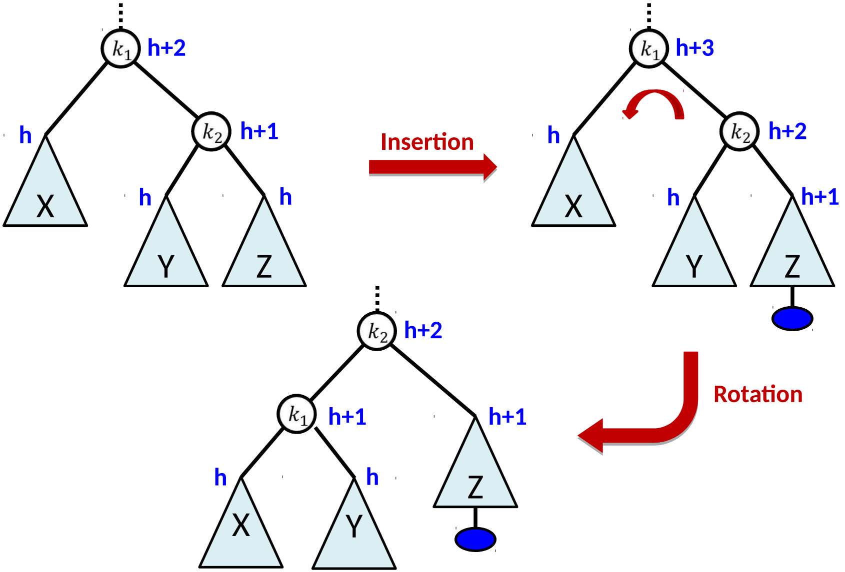
Source: UPC - J. Cortadella & J. Petit

Single rotation: the left-left case



© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

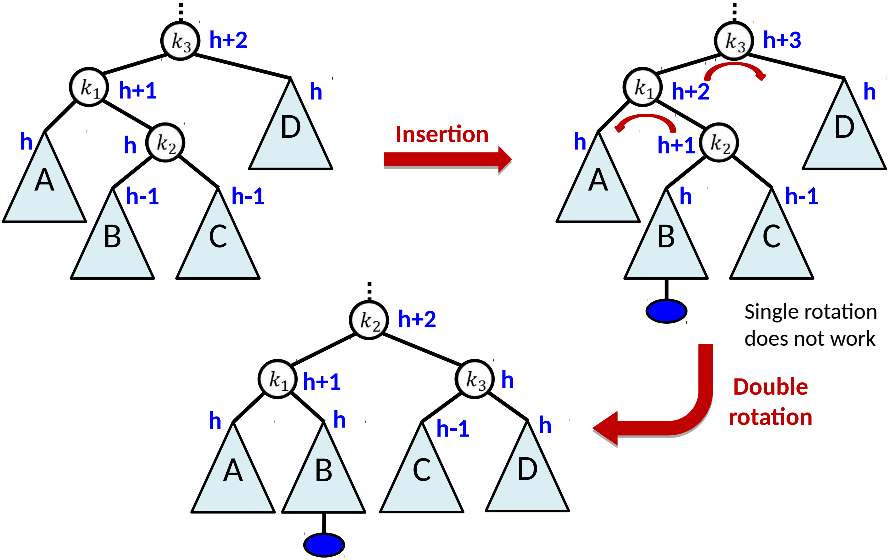
Single rotation: the right-right case



© Dept. CS, UPC

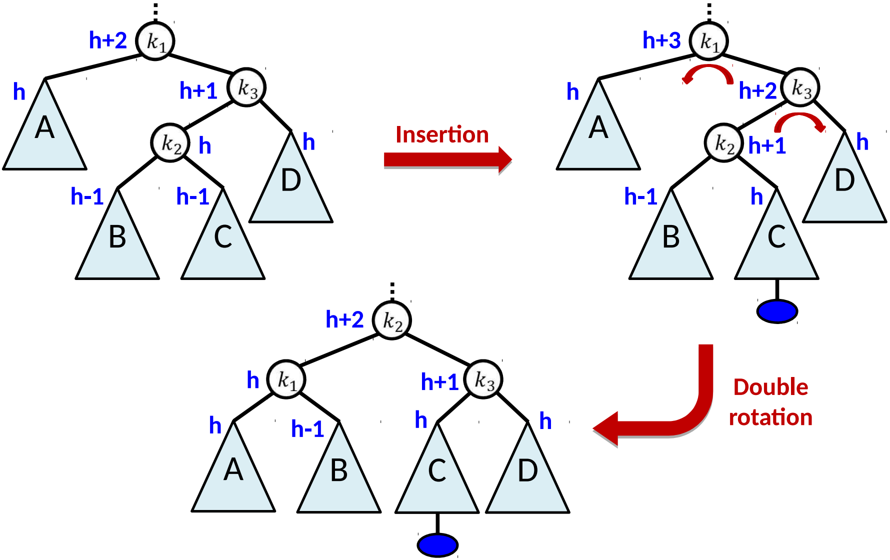
Source: UPC - J. Cortadella & J. Petit

Double rotation: the left-right case



Source: UPC - J. Cortadella & J. Petit

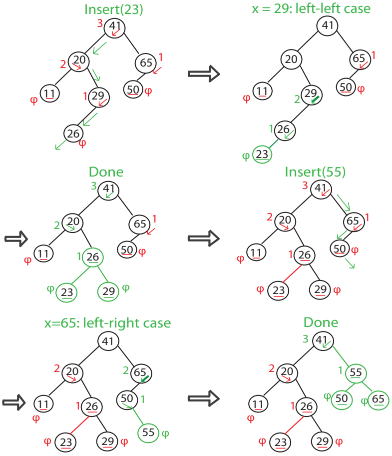
Double rotation: the right-left case



 © Dept. CS, UPC

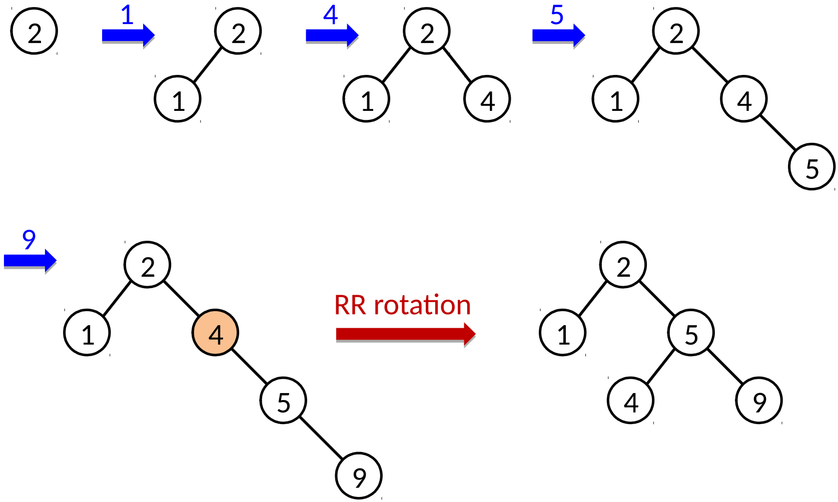
Source: UPC - J. Cortadella & J. Petit

Examples of insert/balancing



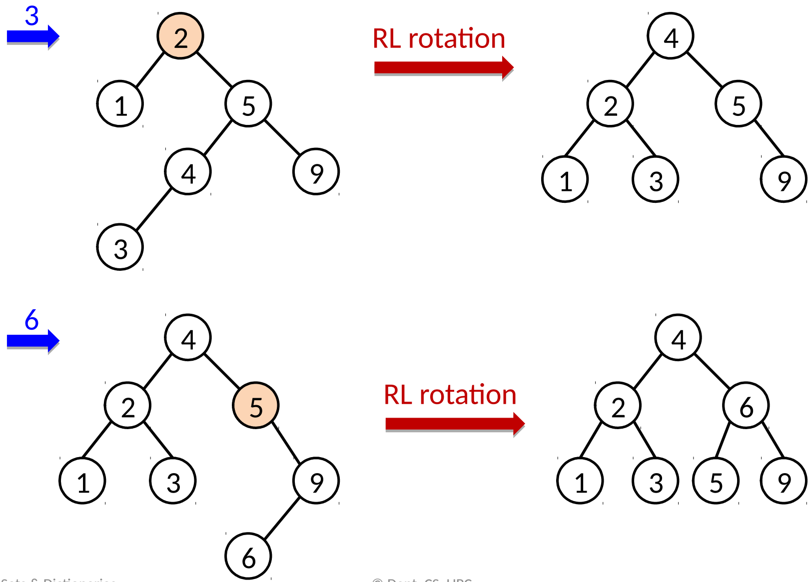
Source: MIT OpenCourseWare

Example: insertions

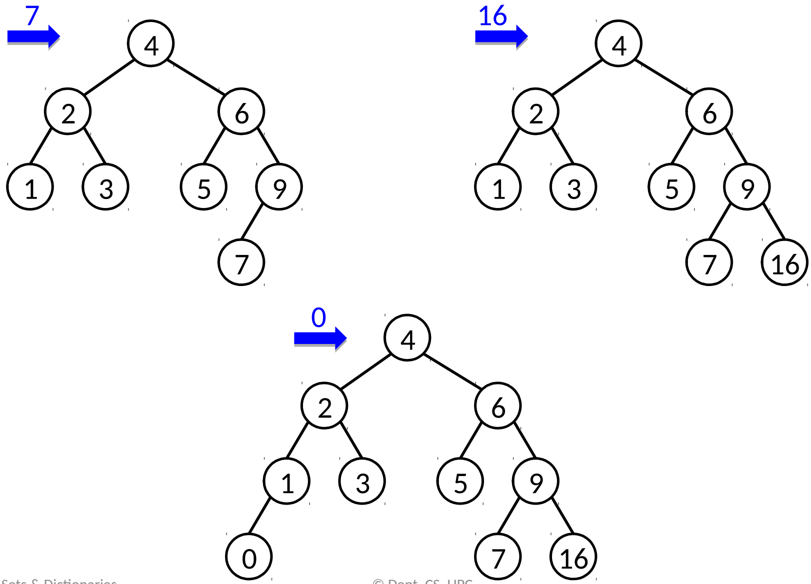


© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

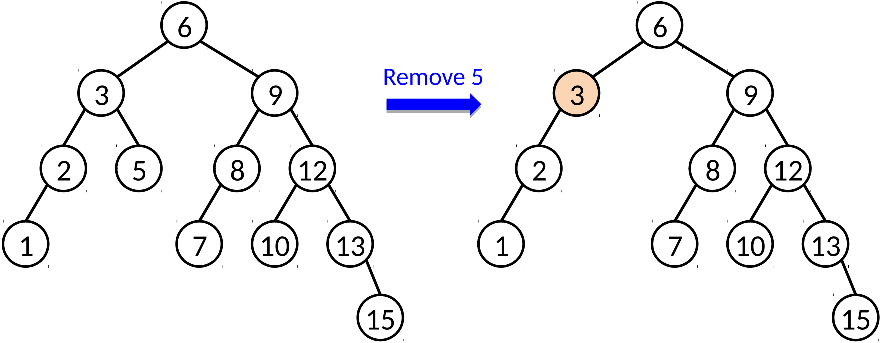
Example: insertions



Example: insertions



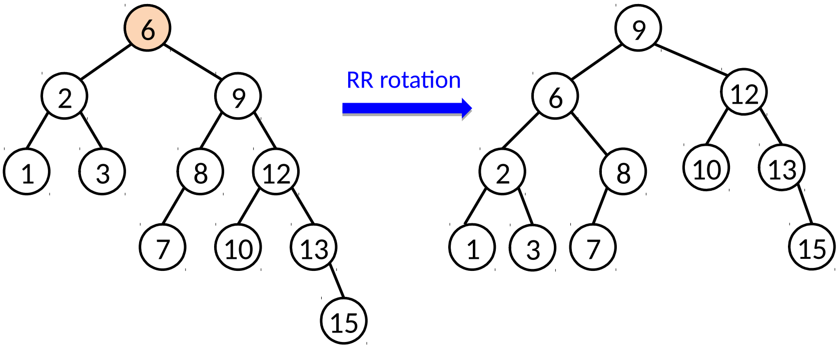
Example: deletion



Apply LL rotation on 3

Source: UPC - J. Cortadella & J. Petit

Example: deletion



Implementation details

- The height must be stored at each node. Only the unbalancing factor ({-1,0,1}) is strictly required.
- The insertion/deletion operations are implemented similarly as in BSTs (recursively).
- The re-balancing of the tree is done when the recursive calls return to the ancestors (check heights and rotate if necessary).

© Dept. CS, UPC
Source: UPC - J. Cortadella & J. Petit

Complexity

- Single and double rotations only need the manipulation of few pointers and the height of the nodes ($O(1)$).

- Insertion: the height of the subtree after a rotation is the same as the height before the insertion. Therefore, at most only one rotation must be applied for each insertion.

- Deletion: more complicated. More than one rotation might be required.

- Worst case for deletion: $O(\log n)$ rotations (a chain effect from leaves to root).

© Dept. CS, UPC

Balanced Search Trees …

- AVL trees (Adelson-Velsii and Landis 1962)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Scapegoat trees (Galperin and Rivest 1993)
- Treaps (Seidel and Aragon 1996)
- ….

Source: MIT OpenCourseWare