Algorithmics
Correction Final Exam #2 (P2)

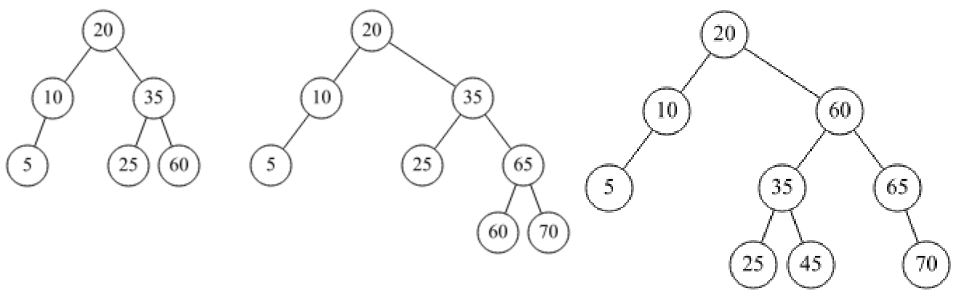
Undergraduate 1^{st} year S2 – Epita

30 May 2018 - 14 : 00

Solution 1 **(AVL – *3 points*)**

Final AVL from th list $[25, 60, 35, 10, 20, 5, 70, 65, 45]$.

Final AVL:



Rotations:

```
rlr(25) rdg(25)
lrr(25) rgd(25)
rr(35)  rd(35)
--------------
rlr(60) rdg(60)
--------------
rlr(35) rdg(35)
```

Solution 2 **(Leonardo trees – *3 points*)**

1. The Fibonacci tree A_5 is the one in figure 1 with each node containing its balance factor value.

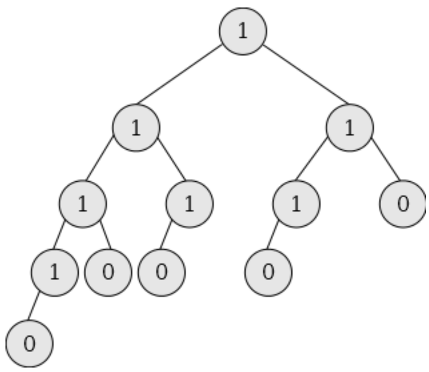


Figure 1: The Fibonacci tree A_5

2. (a) $h_n = n - 1$

   (b) A_0 is a leaf, A_1 has a single node at its left, nothing at its right : these trees are height-balanced.
   With $n \geq 2$, A_n height is $n - 1$. Its subtrees are A_{n-1} of height $n - 2$ and A_{n-2} of height $n - 3$. Thus, the balance factor of the root of A_n is 1 $(n - 2 - (n - 3))$.
   All internal nodes of a Fibonacci tree have a balance factor of 1 : it is an height-balanced tree.

Solution 3 (List → AVL – ***5 points***)

Specifications:

The function `list2avl(`L`)` returns an A.-V.L. (class `AVL`) built from the list L sorted in stricly increasing order.

First version :

- Works on $[left, right]$ (as in lecture)
- Recursive function returns the height: to compute balance factors in each node

```
def __sortedList2AVL(L, left, right):
    """
    L[left, right] -> AVL
    """
    if left > right:
        return (None, -1)
    else:
        mid = left + (right-left) // 2 # or (left + right) // 2
        B = avl.AVL(L[mid], None, None, 0)
        (B.left, hl) = __sortedList2AVL(L, left, mid - 1)
        (B.right, hr) = __sortedList2AVL(L, mid + 1, right)
        B.bal = hl - hr
        return (B, 1 + max(hl, hr))

def sortedList2AVL(L):
    (A, _) = __sortedList2AVL(L, 0, len(L)-1)
    return A
```

***Solution 4* (AVL - Minimum deletion – *6 points*)**

1. Rotations and height changes after minimum deletion:

bal(root)	*bal(right child)*	rotation	Δh
-2	-1	lr	1
	0		0
	1	rlr	1

2. **Specifications:** The function `del_min_avl (A)` deletes the node containing the minimum value of the non-empty AVL A. It returns a pair: the new tree and a boolean that indicates whether the tree height has changed.

```python
def del_min_avl(A):
    if A.left == None:
        return (A.right, True)
    else:
        (A.left, dh) = del_min_avl(A.left)
        if dh:
            A.bal -= 1
            if A.bal == -2:
                if A.right.bal == +1:
                    A = rlr(A)  # rdg(A)
                else:
                    A = lr(A)   # rg(A)
            return (A, A.bal == 0)
        else:
            return (A, False)

# long version
def del_min_avl2(A):
    if A.left == None:
        return (A.right, True)
    else:
        (A.left, dh) = del_min_avl2(A.left)
        if not dh:
            return (A, False)
        else:
            A.bal -= 1
            if A.bal == 0:
                return (A, True)
            elif A.bal == -1:
                return (A, False)
            else:   # A.bal == -2
                if A.right.bal == -1:
                    A = lr(A)   # rg(A)
                    return (A, True)
                elif A.right.bal == 0:
                    A = lr(A)   # rg(A)
                    return (A, False)
                else:
                    A = rlr(A)  # rdg(A)
                    return (A, True)
```

***Solution 5* (BST and mystery – *4 points*)**

1. *Returned results?*
   (a) `call(25,` B`)` : None
   (b) `call(21,` B`)` : 26
   (c) `call(20,` B`)` : 21
   (d) `call(9,` B`)` : 15
   (e) `call(53,` B`)` : None

2. `bst_mystery(x, B)` (`B` any BST, with distinct elements).
   At the end of part 1:
   (a) `B` is the tree that contains x in its root, None if x is not in the tree.
   (b) On the search path, `P` is the tree which root is the last encounter node before descending on the left (it stays None if we never go to the left).

3. `call(x, B)`: if x is found in B and is not the greatest value, the function returns the value just after x in B. Otherwise it returns `None`.