

# Algorithmique

## Partiel n° 2 (P2)

INFO-SUP (S2)  
EPITA

*29 mai 2017 - 13h45*

---

### Consignes (à lire) :

- ☐ Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
    - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
    - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
    - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
    - Aucune réponse au crayon de papier ne sera corrigée.
  - ☐ La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
  - ☐ **Le code :**
    - Tout code doit être écrit dans le langage Python (pas de C, CAML, ALGO ou autre).
    - **Tout code Python non indenté ne sera pas corrigé.**
    - Tout ce dont vous avez besoin (fonctions, méthodes) est indiqué en **annexe** !
  - ☐ Durée : 2h00
-

**Exercice 1 (Arbres 234 ... – 6 points)**

1. En appliquant les éclatements à la remontée, construire l'arbre 2.3.4. correspondant aux insertions successives des valeurs  $\{Q, U, E, S, T, I, O, N, B, A, Z, Y, K\}$  (Vous dessinerez uniquement l'arbre final).
2. Construire l'arbre rouge-noir associé à l'arbre 2.3.4. final de la question précédente. Vous considérerez que les 3-noeuds sont tous penchés à gauche.
3. Donner trois propriétés d'un arbre 2.3.4.
4. Donner trois propriétés d'un arbre rouge-noir.
5. Quelle méthode *simple* permet de déterminer la taille (le nombre de noeuds) d'un arbre 2.3.4. en utilisant l'arbre bicolore qui le représente?

**Exercice 2 (Arbres et mystère – 3 points)**

```
1  def __makeTree(n, i, cur):
2      if i > n:
3          return (None, cur)
4      else:
5          (left, cur) = __makeTree(n, 2*i, cur)
6          key = cur+1
7          (right, cur) = __makeTree(n, 2*i+1, key)
8          return (binTree.BinTree(key, left, right), cur)
9
10 def makeTree(n):
11     (B, val) = __makeTree(n, 1, 0)
12     return B
```

1. La fonction `makeTree(n)` construit (et retourne) un arbre binaire. Dessiner l'arbre résultat lorsque  $n = 13$ .
2. On appelle `makeTree(n)` avec  $n$  un entier strictement positif.  
Donner deux propriétés de l'arbre retourné.

**Exercice 3 (ABR → AVL – 5 points)**

Écrire une fonction qui construit à partir d'un arbre binaire classique (`BinTree`) un arbre équivalent au premier (contenant les mêmes valeurs aux mêmes places) mais avec le déséquilibre (le "champ" *bal*) renseigné en chaque nœud (AVL).

**Indice :** La fonction récursive retournera de plus la hauteur de l'arbre.

On ne s'occupera pas de vérifier si l'arbre est équilibré ou non.

### Exercice 4 (Arbres AA – 6 points)

Un *arbre AA* (Arne Andersson tree) est un dérivé des arbres bicolores, avec une propriété supplémentaire : les nœuds rouges ne peuvent être ajoutés qu'en fils droit.

Les arbres AA sont donc une simulation des arbres 2-3 plutôt que des arbres 2-3-4 : il n'y a que des 2-nœuds et des 3-nœuds, les 3-nœuds étant représentés penchés à droite.

Côté implémentation, on ajoute en chaque nœud une information supplémentaire (voir définition de la classe `AAtree` en annexe) : `level` qui représente le niveau, de bas en haut, dans l'arbre 2-3 correspondant (les feuilles au niveau 1). Deux clés appartenant au même 3-nœud de l'arbre 2-3 ont donc le même niveau (la plus grande clé, considérée comme rouge, étant dans le fils droit de la plus petite) dans l'arbre AA.

Par exemple, l'arbre 2-3 de la figure 1 sera représenté par l'arbre AA de la figure 2 : les clés "rouges" sont celles qui ont le même niveau que leur père (18, 15 et 28 dans l'exemple) et sont toujours à droite.

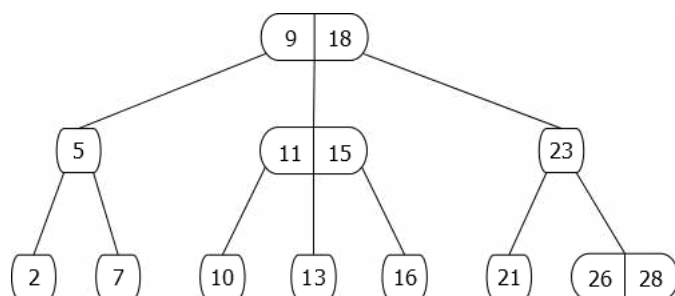


FIGURE 1 – Arbre 2-3

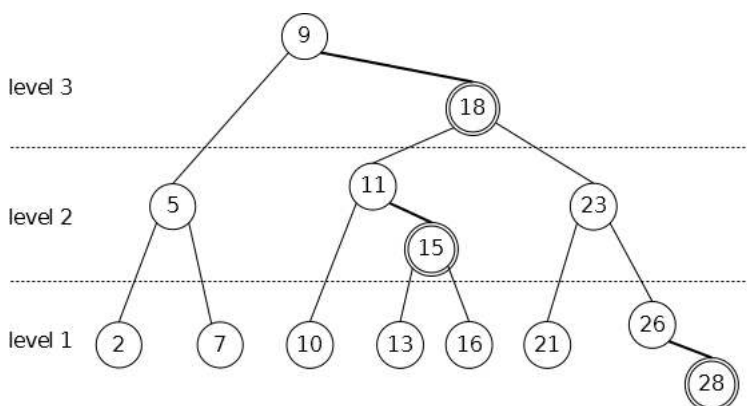


FIGURE 2 – Arbre AA A

Pour l'arbre AA, le niveau représente le nombre de liens gauches à suivre avant d'atteindre un fils vide :

- Le "niveau" de chaque feuille est donc 1.
- Tout nœud de niveau supérieur doit avoir au moins un fils droit de même niveau (considéré comme rouge) ou de niveau inférieur.
- Il ne peut pas avoir 3 nœuds de même niveau sur un même chemin (un nœud ne peut avoir le même niveau que son grand père).
- Un fils gauche est obligatoirement de niveau inférieur à son père (pas de fils rouge à gauche).

Il n'y a que deux transformations pour maintenir les propriétés de l'arbre après ajout ou suppression :

**Skew :** Une rotation droite utilisée lorsque une insertion ou une suppression crée un fils gauche rouge (le fils gauche a le même niveau que son père). Les deux nœuds conservent le même niveau.

**Split :** Une rotation gauche utilisée lorsque l'on se retrouve avec deux nœuds rouges consécutifs (3 nœuds de même niveau consécutifs à droite). La racine voit son niveau incrémenté (correspond à un éclatement).

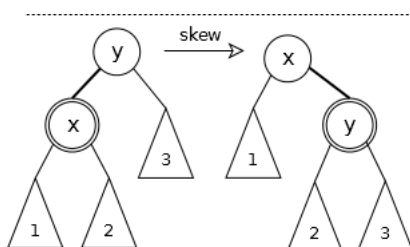


FIGURE 3 – Skew

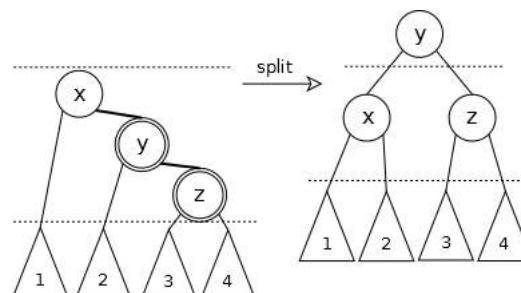


FIGURE 4 – Split

On supposera les fonctions **skew** et **split**, prenant toutes les deux un arbre AA en paramètre, implémentées. Elles effectuent la rotation, ainsi que la mise à jour éventuelle des niveaux et retournent l'arbre modifié.

Le principe de l'insertion de la valeur  $x$  dans l'arbre AA  $A$  est le suivant :

- La descente se fait comme pour l'insertion d'une clé en feuille dans un arbre binaire de recherche. La nouvelle feuille est insérée au niveau 1. Si la valeur  $x$  est déjà présente, elle n'est pas insérée.
- En remontant :
  - de la droite : s'il a 3 nœuds de même niveau (toujours à droite), on effectue un "split". Par exemple, voir le résultat de l'insertion de 31 dans l'arbre de la figure 2 (voir figure 5).
  - de la gauche : il faut s'assurer que le fils gauche n'a pas le même niveau que le nœud actuel (par exemple si on insère 1 dans l'arbre 2), si c'est le cas on effectue un "skew". Dans ce cas uniquement, on vérifie que le petit-fils droit (après transformation) n'a pas le même niveau lui aussi, auquel cas, un "split" est nécessaire. Ce dernier cas peut-être illustré par l'ajout de 25 dans l'arbre de la figure 2 (voir figure 6).

Les transformations pouvant se propager jusqu'à la racine.

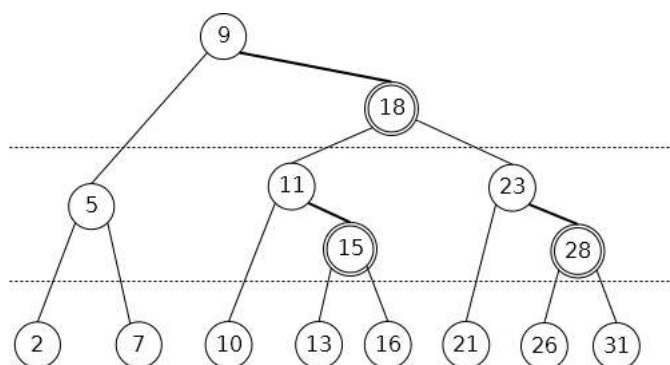


FIGURE 5 – Arbre  $A$  après insertion de 31

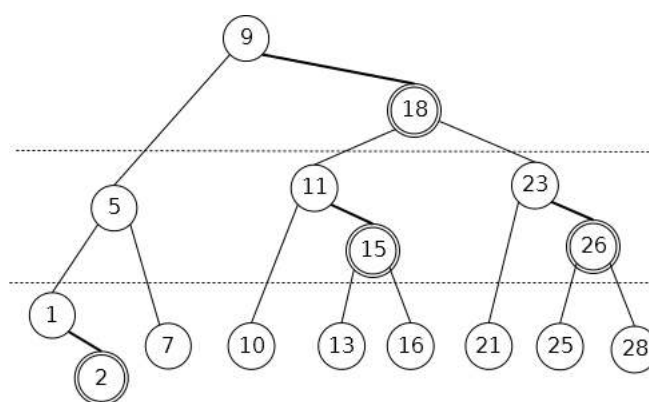


FIGURE 6 – Arbre  $A$  après insertion de 1 puis 25

1. Dessiner l'arbre AA obtenu après insertion de la valeur 4 dans l'arbre de la figure 6.
2. Écrire la fonction **insertAA**( $x$ ,  $A$ ) qui insère la clé  $x$  dans l'arbre AA  $A$ , sauf si celle-ci est déjà présente. La fonction retourne l'arbre résultat de l'insertion.

## Annexes

### Les arbres binaires

Les arbres binaires "classiques" :

```
1 class binTree:
2     def __init__(self, key, left, right):
3         self.key = key
4         self.left = left
5         self.right = right
```

Les AVL, avec les déséquilibres :

```
1 class AVL:
2     def __init__(self, key, left, right, bal):
3         self.key = key
4         self.left = left
5         self.right = right
6         self.bal = bal
```

Les arbres AA, avec le "level" :

```
1 class AAtree:
2     def __init__(self, key, left, right, level):
3         self.key = key
4         self.left = left
5         self.right = right
6         self.level = level
```

Dans tous les cas, l'arbre vide est None.

### Fonctions et méthodes autorisées

- abs
- min et max, mais uniquement avec deux valeurs entières !

### Vos fonctions

Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.