PSL - Université Paris Dauphine | **Algorithmes et Applications**
CPES2 | **2019-2020**

TD5
Programmation Orientée Objet & Arbres

Arbre Binaire

Un **arbre binaire** est un ensemble hierarchisé d'éléments, appelés noeuds, tel que :

- Chaque noeud a un père et un seul, à l'exception d'un seul noeud, appelé racine, qui n'a pas de père et que l'on représente conventionnellement en haut
- Chaque noeud a 0, 1 ou 2 fils (fils gauche et fils droit)
- Un noeud sans fils est appelé une feuille

La profondeur d'un noeud i est la longueur (en nombre d'arêtes) du chemin allant de la racine à i. La hauteur d'un arbre est la profondeur maximum.

Dans la suite, on représentera des arbres étiquetés, c'est-à-dire que chaque noeud possède une valeur qu'on appelle étiquette.

Exemple : Arbre binaire de racine d'étiquette 1, de hauteur 3 et contenant 6 noeuds

```
   1
 /   \
2     3
 \
  4
 / \
5   6
```

Tous les noeuds ayant une même profondeur forment un niveau de l'arbre binaire. Dans un **arbre binaire équilibré**, les seules feuilles manquantes doivent se trouver à droite des feuilles situées au plus bas niveau de l'arbre.

Exemple : Arbre binaire équilibré de hauteur 2 contenant 6 noeuds

```
     1
   /   \
  2     3
 / \   /
4   5 6
```

Un arbre binaire est **complet** si tous ses noeuds ont deux fils sauf les feuilles.

Exemple : Arbre binaire complet de hauteur 2

```
         1              2^0 sommets à la profondeur 0
       /   \
      2      3          2^1 sommets à la profondeur 1
    / \    / \
   4   5  6   7         2^2 sommets à la profondeur 2
```

Un arbre binaire complet de hauteur h comporte : $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ noeuds. A la profondeur p, on a 2^p sommets.

Programmation Orientée Objet (POO)

Soit l'arbre binaire suivant :

```
        12
       /  \
      2    20
    / \      \
   9   4      3
        \
         10
```

Pour représenter un arbre binaire en Python, on va introduire la classe Arbre suivante :

```
class Arbre:
    def __init__(self, entier):
        self.noeud = entier
        self.filsDroit = None
        self.filsGauche = None
        self.pere = None
```

La fonction `__init__(self)` est un constructeur. C'est une fonction reconnue par Python. Pour créer un objet de la classe `Arbre`, il suffit alors d'écrire :

```
arbre = Arbre(12)
```

Ici, on appelle le nom de la classe que l'on a définie, `Arbre`, en lui passant en argument un entier. En réalité, lorsqu'on appelle ainsi le nom de la classe, Python comprend qu'il doit utiliser le constructeur `__init__(self)`, où `self` correspond à l'arbre que l'on crée. C'est pour cela qu'on a passé en argument un entier qui correspond à l'argument `entier` de `__init__(self)`. L'arbre ainsi créé, suivant la définition du constructeur contient donc maintenant plusieurs champs :

- `arbre.noeud` : contient la valeur de l'argument `entier`, ici 12
- `arbre.filsDroit` : vaut `None`
- `arbre.filsGauche` : vaut `None`
- `arbre.pere` : vaut `None`

La variable `arbre` correspond alors à l'arbre :

```
                 arbre.noeud :  12
                               /  \
     arbre.filsGauche : None     None : arbre.filsDroit
```

Le noeud 12 est racine de l'arbre car son père est égal à None.
Si on veut ajouter un fils droit d'étiquette 20 à `arbre`, on écrira :

```
noeud = Arbre(20)
arbre.filsDroit = noeud
noeud.pere = arbre
```

ce qui correspond à :

```
                     arbre.noeud :  12
                                   /  \
          arbre.filsGauche : None     20 : arbre.filsDroit
                                     /  \
   arbre.filsDroit.filsGauche : None    None : arbre.filsDroit.filsDroit
```

Pour parcourir un arbre, on part de la racine et on descend en profondeur en allant toujours le plus à gauche possible. Pendant ce parcours, chaque noeud sera visité 3 fois (descente, montée gauche, montée droite). Si on marque les noeuds pendant le parcours, on distingue classiquement 3 ordres de marquage :

- Préfixe : marquage à la descente, première rencontre du noeud
- Infixe : marquage lors de la montée gauche, deuxième rencontre du noeud
- Postfixe : marquage lors de la montée droite, troisième rencontre du noeud

Dans le graphe ci-dessus, les ordres de marquage sont :

- Préfixe : 12 2 9 4 10 20 3
- Infixe : 9 2 4 10 12 20 3
- Postfixe : 9 10 4 2 3 20 12

Exercice 1 : Affichage d'arbres

1. Construire l'arbre :

```
        12
       /  \
      2    20
     / \     \
    9   4     3
         \
          10
```

   On l'utilisera pour tester les fonctions de cet exercice.

2. Ecrire une fonction qui renvoie le taille de l'arbre, c'est-à-dire son nombre de noeuds.
3. Ecrire une fonction qui renvoie la hauteur de l'arbre.
4. Ecrire une fonction récursive qui étant donné un arbre binaire, affiche chaque noeud de l'arbre selon un ordre préfixe.
5. Ecrire une fonction récursive qui étant donné un arbre binaire, affiche chaque noeud de l'arbre selon un ordre infixe.
6. Ecrire une fonction récursive qui étant donné un arbre binaire, affiche chaque noeud de l'arbre selon un ordre postfixe.
7. Reunir les 3 fonctions précédentes en une unique fonction `affichage()` prenant un argument supplémentaire `type` permettant de préciser l'affichage désiré : préfixe, infixe ou postfixe.

Correction

1. ```
def taille(arbre):
    if arbre is None:
        return 0
    return 1 + taille(arbre.filsDroit) + taille(arbre.filsGauche)
```

2. ```
def hauteur(arbre):
    if arbre is None:
        return 0
    return 1 + max(hauteur(arbre.filsDroit), hauteur(arbre.filsGauche))
```

3. ```
def prefixe(a, marque):
    if a is not None:
        marque.append(a.noeud)
        prefixe(a.filsGauche, marque)
        prefixe(a.filsDroit, marque)
```

4. ```
def infixe(a, marque):
    if a is not None:
        infixe(a.filsGauche, marque)
        marque.append(a.noeud)
        infixe(a.filsDroit, marque)
```

5. ```
def postfixe(a, marque):
    if a is not None:
        postfixe(a.filsGauche, marque)
        postfixe(a.filsDroit, marque)
        marque.append(a.noeud)
```

6. ```
def affichage(a, marque, type):
    if a is not None:
        if type == 'prefixe':
            marque.append(a.noeud)
        affichage(a.filsGauche, marque, type)

        if type == 'infixe':
            marque.append(a.noeud)
        affichage(a.filsDroit, marque, type)

        if type == 'postfixe':
            marque.append(a.noeud)
```

Exercice 2 : Arbres binaires de recherche

Remarque : les noeuds sont étiquetés par des valeurs numériques toutes différentes.

Un arbre binaire de recherche est un arbre binaire tel que pour tout noeud n de l'arbre :

- Tous les noeuds du sous-arbre gauche, s'il existe, ont une étiquette inférieure ou égale à celle de n
- Tous les noeuds du sous-arbre droit, s'il existe, ont une étiquette supérieure ou égale à celle de n

Exemple :

```
            25
          /    \
        8       37
      /  \     /  \
    4    12  33   42
   / \     \
  2   5    20
          /
        15
```

1. Ecrire une fonction `initialisationABR()` qui renvoie la racine de l'arbre de type Arbre. Les étiquettes des noeuds sont saisies au clavier par l'utilisateur. Appliquer votre fonction de parcours sur cet arbre pour afficher l'ordre infixe des noeuds. Que constatez vous ?

2. Implémenter une fonction `recherche()` qui prend en argument un arbre binaire de recherche A et une étiquette x, et renvoie `True` si x appartient à A et `False` sinon. Pour rechercher x, on procède de la façon suivante : on compare x à la racine, si x est égal à la racine, renvoyer `True`, sinon, si x est inférieur (resp. supérieur) à la racine, poursuivre la recherche dans le sous-arbre gauche (resp. droit), si le sous-arbre est vide, renvoyer `False`. Appeler votre fonction pour tester l'appartenance de 20 puis 3 à l'exemple.

3. Implémenter une fonction `maxA()` qui prend en argument un arbre binaire de recherche A et retourne sa plus grande étiquette. Pour cela, on parcourt A en partant de la racine, sa plus grande étiquette est nécessairement dans le sous-arbre droit. On descend jusqu'à arriver à un noeud sans fils-droit. Ce noeud a nécessairement l'étiquette maximale. Appeler votre fonction sur l'exemple.

4. Implémenter une fonction `insereFeuille()` qui prend en argument un arbre binaire de recherche A et une étiquette x à insérer dans A. On procède de la façon suivante : on part de la racine, si x est plus grand (resp. plus petit), on descend dans le sous-arbre droit (resp. gauche) jusqu'à ce que le sous-arbre soit vide et on insère x à cette position. Appeler votre fonction pour insérer le noeud d'étiquette 29 à l'exemple.

Correction

1. ```python
   def initialisationArbre(e):
       abr = Arbre(e)

       print('Fils Gauche de ', e)
       x = input()
       if x != '':
           noeud = initialisationArbre(int(x))
           abr.filsGauche = noeud
           noeud.pere = abr

       print('Fils Droit de ', e)
       y = input()
       if y != '':
           noeud = initialisationArbre(int(y))
           abr.filsDroit = noeud
           noeud.pere = abr
       return abr
   ```

   Sur l'arbre de l'énoncé, on obtient en ordre infixe `[2, 4, 5, 8, 12, 15, 20, 25, 33, 37, 42]` : c'est la liste triée des étiquettes de l'arbre.
   ```

2.
```python
def recherche(A, x):
    while A != None:
        if x == A.noeud:
            return True
        elif x < A.noeud:
            A = A.filsGauche
        else:
            A = A.filsDroit
    return False
```

3.
```python
def maxA(abr):
    while(abr.filsDroit != None):
        abr = abr.filsDroit
    return abr.noeud
```

4.
```python
def insereFeuille(A, x):
    if x < A.noeud:
        if A.filsGauche == None:
            noeud = Arbre(x)
            A.filsGauche = noeud
            noeud.pere = A
        else :
            insereFeuille(A.filsGauche, x)
    else :
        if A.filsDroit == None:
            noeud = Arbre(x)
            A.filsDroit = noeud
            noeud.pere = A
        else :
            insereFeuille(A.filsDroit, x)
```

Exercice 3 : Suppression

Soit l'arbre binaire de recherche ci-dessous.

```
      5
    /   \
   4     9
  /     / \
1      7   20
           /
        16
      /     \
    12       18
   /  \      /
11     13  17
```

Pour supprimer un noeud d'étiquette x dans un arbre binaire de recherche A, il faut tout d'abord déterminer sa place dans A, puis effectuer la suppression qui s'accompagne d'une réorganisation de l'arbre. Cette réorganisation dépend de la place du noeud d'étiquette x dans A :

- s'il s'agit d'une feuille, alors la suppression est immédiate et n'engendre aucune réorganisation,
- si le noeud d'étiquette x a un seul fils, alors il suffit de le remplacer par son fils,

- si le noeud d'étiquette x a deux fils, il faut remplacer x par le noeud qui lui est immédiatement inférieur y ($y < x$). Pour trouver y, il faut chercher la plus grande étiquette du sous-arbre ayant pour racine le fils-gauche de x : y est le noeud le plus à droite sans fils-droit. Il s'agit alors de supprimer y (ce qui est simple car y a 0 ou 1 fils) puis de remplacer x par y.

1. Donner les nouveaux arbres binaires de recherche obtenus en supprimant successivement les éléments 13, puis 20, puis 5.
2. Ecrire la fonction `supprimemax()` qui prend en entrée l'arbre binaire de recherche A et retourne le plus grand élément de A tout en le supprimant de A.
3. Ecrire la fonction `supprimer()` qui supprime l'élément x de l'arbre binaire de recherche A.

Correction

1. *Suppression de 13*

```
     5
    / \
   4   9
  /   / \
 1   7  20
       /
      16
    /   \
   12    18
  /      /
11      17
```

Suppression de 20

```
     5
    / \
   4   9
  /   / \
 1   7  16
       /  \
     12   18
    /    /
   11   17
```

Suppression de 5

```
     5
    / \
   1   9
      / \
     7  16
       /  \
     12   18
    /    /
   11   17
```

2.
```python
def supprimemax(A):
    if taille(A) == 1:
        return A.noeud, None

    elif A.filsDroit == None:
        return A.noeud, A.filsGauche

    else :
        courant = A.filsDroit
        while courant.filsDroit != None:
            courant = courant.filsDroit
        if courant.filsGauche == None:
            courant.pere.filsDroit = None
        else:
            courant.pere.filsDroit = courant.filsGauche
            courant.filsGauche.pere = courant.pere.filsDroit
        return courant.noeud, A
```

3.
```python
def supprimer(x, A):
    if x < A.noeud:
        A.filsGauche = supprimer(x, A.filsGauche)

    elif x > A.noeud:
        A.filsDroit = supprimer(x, A.filsDroit)

    else:
        if A.filsDroit == None and A.filsGauche == None:
            return None

        elif A.filsDroit == None :
            return A.filsGauche

        elif A.filsGauche == None :
            return A.filsDroit

        else:
            val_max, A.filsGauche = supprimemax(A.filsGauche)
            A.noeud = val_max
    return A
```

Exercice 4 : Le tri par tas

Un arbre binaire valué partiellement ordonné, ou **minimier**, est un arbre étiqueté dont l'étiquette de chaque noeud est inférieure ou égale à celle de chacun de ses fils.

Exemple : Minimier de hauteur 3 comportant 10 noeuds étiquetés

```
                3
              /   \
           7         5
          / \       / \
         8   9     8   6
        / \  /
      12  11 15
```

Le noeud de plus petite étiquette est le noeud racine.

Si on le supprime, on détruit la structure d'arbre. Pour la recomposer, il suffit de prendre la feuille la plus à droite du niveau le plus bas et on la place temporairement à la racine. Puis, il faut pousser cet élément aussi bas que possible en l'échangeant avec celui de ses fils ayant la plus petite étiquette inférieure. On s'arrête lorsque le noeud est soit devenu une feuille, soit a ses fils de plus grande étiquette. Le nombre d'opérations élémentaires dépend donc de la hauteur de l'arbre.

Exemple : Suppression de la racine puis recomposition du minimier

```
           15
         /    \
      7         5
     / \       / \
   8    9    8    6
  / \
12   11
```

Etape 2 pour recomposer le minimier

```
           5
         /    \
      7         6
     / \       / \
   8    9    8    15
  / \
12   11
```

Pour insérer un nouveau noeud dans l'arbre, on réalise la procédure "inverse". On commence par placer le noeud en question aussi loin que possible à gauche au plus bas niveau de l'arbre. Puis, on le pousse aussi haut que possible dans l'arbre de la façon suivante : si son père à une etiquette supérieure à la sienne, il faut les échanger, et on réitère les comparaisons jusqu'à ce que le noeud inséré se trouve à la racine ou bien ait une étiquette inférieure à celle de son père. Le nombre d'opérations élémentaires dépend donc également de la hauteur de l'arbre.

Exemple : Insertion du noeud d'étiquette 2 : étape 1

```
           5
         /    \
      7         6
     / \       / \
   8    9    8    15
  / \   /
12   11 2
```

Etape 2 pour placer le nouveau noeud

```
           2
         /    \
      5         6
     / \       / \
   8    7    8    15
  / \   /
12   11 9
```

Le **tas** est une structure de données pouvant être utilisée pour représenter en machine un minimier. Soit un arbre contenant n noeuds. Dans le tas, on utilise les n premières cellules d'un tableau unidimensionnel T de la façon suivante : les noeuds remplissent les cellules $T[1], T[2], \ldots, T[n]$ niveau par niveau à partir du haut, et à l'intérieur d'un même niveau de la gauche vers la droite. Ainsi, le fils gauche, s'il existe, du noeud $T[i]$ se trouve en $T[2i]$ et le fils droit, s'il existe, en $T[2i+1]$; le père de $T[i]$ se trouve en $T[\lfloor \frac{i}{2} \rfloor]$.

Exemple : Le tas T associé au dernier minimier ci-dessus

T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]
2	5	6	8	7	8	15	12	11	9

Les fils du noeud étiqueté par 8, stocké à l'indice $i = 4$ du tas, sont à l'indice $2i = 8$ du tas pour le fils gauche et à l'indice $2i + 1 = 9$ pour le fils droit.
Le père du noeud étiqueté par 8, stocké à l'indice $i = 4$ du tas, est à l'indice $\lfloor \frac{4}{2} \rfloor = 2$ du tas.

Pour trier par ordre croissant une liste L contenant n entiers et stocker cette liste triée dans une liste appelée $Ltri$, un algorithme efficace consiste à :

Etape 1. Stocker la liste L dans un minimier noté T. Créer une liste vide $Ltri$.

Etape 2. Extraire la racine r du minimier, recomposer le minimier et ajouter r à $Ltri$.

Etape 3. Répéter l'étape 2 jusqu'à avoir vidé le minimier.

1. Ecrire une classe `Heap` dont le constructeur initialise deux champs L et n (ce dernier contiendra la longueur de L dans la suite de l'exercice) sous forme respectivement de liste vide et d'un entier égal à 0.

2. Ajouter à `Heap` une méthode `push(self, x)` qui place l'élément `x` à la bonne place dans le tas et donc dans `self.L`. On oubliera pas d'incrémenter n de 1.

3. Ajouter à `Heap` une méthode `fromList(self, l)` qui passe une liste `l` dans `self.L` avec la structure de tas qui correspond à celle d'un minimier.

4. Ajouter à `Heap` une méthode `pop(self)` qui enlève et retourne le premier élément de `self.L` en recomposant le tas comme on recompose un minimier. On oubliera pas de décrémenter n de 1.

5. Ecrire en dehors de `Heap` une fonction `triTas()` qui implémente l'algorithme de tri par tas en utilisant `Heap`.

6. En déduire la complexité de l'algorithme de tri par tas.

7. Implémenter à nouveau l'algorithme en utilisant maintenant le module `heapq` qui propose une implémentation efficace des tas. Les fonctions suivantes sont notamment définies :

   - `heapq.heappush(T, x)` : ajoute la valeur x au tas T (en conservant la propriété de tas)
   - `heapq.heappop(T)` : enlève et retourne le premier élément du tas T
   - `heapq.heapify(L)` : transforme la liste L en un tas (en place et en temps linéaire)

8. Comparer les temps d'éxécution du tri par tas selon qu'on utilise notre classe `Heap` ou le module `heapq` sur des listes de nombres aléatoires tirés de -100 à 100 et de taille 100 à 100000.

Correction

1. ```
   class Heap:
       def __init__(self):
           self.L = []
           self.n = 0
   ```

2. Dans la classe `Heap` :

   ```
   def push(self, x):
       self.L.append(x)
       self.n += 1
       pos = self.n-1
       parent_pos = (pos+1)//2-1
       while pos > 0 and self.L[parent_pos] > self.L[pos]:
           self.L[parent_pos], self.L[pos] = self.L[pos], self.L[parent_pos]
           pos = parent_pos
           parent_pos = (pos+1)//2-1
   ```

3. Dans la classe `Heap` :

   ```
   def fromList(self, l):
       self.L = []
       for x in l:
           self.push(x)
   ```

4. Dans la classe `Heap` :

   ```
   def pop(self):
       to_return = self.L[0]
       if self.n == 1:
           return to_return

       self.L[0] = self.L.pop()
       self.n -= 1
       pos = 0
       child_pos = 2*pos+1
       while child_pos < self.n:
           right_pos = child_pos+1
           if right_pos < self.n and self.L[right_pos] < self.L[child_pos] and
       self.L[right_pos] < self.L[pos]:
               child_pos = right_pos

           if self.L[pos] < self.L[child_pos]:
               break

           self.L[pos], self.L[child_pos] = self.L[child_pos], self.L[pos]
           pos = child_pos
           child_pos = 2*pos+1
       return to_return
   ```

5. ```
   def triTas(L):
       tas = Heap()
       tas.fromList(L)
       Ltri = []
       for k in range(tas.n):
           Ltri.append(tas.pop())
       return Ltri
   ```

6.
```python
import heapq

def triTasQ(L):
    n = len(L)
    Ltri = []
    heapq.heapify(L)
    for i in range(n):
        Ltri.append(heapq.heappop(L))
    return(Ltri)
```

7. $O(n \log n)$

8.
```python
import time
import random
import matplotlib.pyplot as plt

def listeAlea(n):
    l=[]
    for i in range(n):
        l.append(random.uniform(-1000,1000))
    return l

num = 15
tailles = np.logspace(2, 5, 15, dtype='int')
tps, tpsq = np.zeros(num), np.zeros(num)
reps = 10
for i in range(num):
    for k in range(reps):
        L = listeAlea(tailles[i])

        t0 = time.time()
        triTas(L[:])
        tps[i] += time.time()-t0

        t0 = time.time()
        triTasQ(L[:])
        tpsq[i] += time.time()-t0

tps /= reps
tpsq /= reps

plt.plot(tailles, tps, label='Heap')
plt.plot(tailles, tpsq, label='heapq')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('n')
plt.ylabel('t')
plt.title("Comparaison des temps d'execution du tri par tas")
plt.legend()
plt.show()
```