

## Introduction

*Dans ce TD nous allons implémenter et manipuler trois types de structures de données : les dictionnaires, les piles/files et les tas. Ce TD est non-noté mais servira de travail préliminaire pour le prochain TD 3 noté.*

## 1 Rappel sur les Dictionnaires (15min)

Le service Communication de l'Ecole nous demande de lui fournir quelques statistiques sur la dernière promotion fraîchement diplômée. On vous fournit en entrée une liste d'étudiants dans un fichier texte au format CSV (Comma-Separated Values), avec les descriptions de données sur la première ligne, et ensuite une ligne par étudiant. Voici un exemple de fichier `etudiants.txt` :

```
nom;prenom;filier;e;moyenne;absences
Dupond;Pierre;MP;12;0
Clavier;Christian;PSI;14;1
Gilles;Eric;PC;16;3
Arthaud;Nathalie;MP;15;0
```

Le code Python pour charger ce fichier ligne par ligne vous est donné ci-dessous :

```
data = []

with open("etudiants.txt") as f:
    keys = None
    for line in f:
        l = [w.strip() for w in line.split(';')]
        if keys is None:
            keys = l
        else:
            data.append({k:v for k, v in zip(keys, l)})

print(data)
```

**Exercice 1.1 (facile !)** – Écrivez un programme qui lit un fichier d'étudiants en entrée, et imprime la moyenne générale en sortie. Votre programme doit effectuer un unique `print` du résultat. Selon vous que fait la fonction `zip` ?. Voici la moyenne attendue :

```
16.0
```

## 2 Piles et Files (30min)

On souhaite maintenant appeler les étudiants selon leur ordre d'arrivée dans la classe (qui est le même ordre que dans le fichier `etudiants.txt`). Pour cela vous utiliserez des piles et files qui sont des structures de données permettant de manipuler les étudiants de manière séquentielle.

**Exercice 2.1** – Créez une structure de données `Pile` (sous forme de classe) qui possède des méthodes d'ajout (`ajoute`) et accès (`supprime`) aux éléments de la liste `etudiants.txt`, où le dernier arrivé est le premier appelé. Pour tester que votre `Pile` fonctionne bien, ajoutez les étudiants dans leur ordre d'apparition dans le fichier fourni dans l'exercice 1, et si vous supprimez un élève vous obtenez l'élève suivant (vous pouvez aussi utiliser le fichier de test `code/test_pile.py`):

```
>>> p = Pile()
>>> for d in data:
>>>     p.ajoute(d)
>>>
>>> e = p.supprime()
>>> print(e['nom'] + " " + e['prenom'])
```

Arthaud Nathalie

**Exercice 2.2** – Créez désormais une `File` (sur le même format que `Pile`) qui renvoie cette fois le premier étudiant arrivé.

Dupond Pierre

**Exercice 2.3** – Ajoutez à votre `Pile` et `File` une nouvelle fonction `renvoie` qui prend en argument un critère (par exemple une `lambda` fonction) permettant par exemple de retrouver facilement les étudiants ayant par exemple réalisé une prépa PC (et pas une autre). A noter que cette fonction ne supprime pas la valeur de la `Pile/File`. Le résultat suivant est attendu :

```
>>> # on a au préalable initialisé la file..
>>> e = file.renvie(lambda x : x['filiere'] == "PC")
>>> print(e['nom'] + " " + e['prenom'])
```

Gilles Eric

## 3 Tas (75min)

Le professeur souhaite désormais chercher un étudiant selon un critère particulier : celui avec la note minimale. Il pourrait certes utiliser les `Pile` et `File` créées précédemment dans la partie 2, et utiliser une fonction de trie (par exemple la fonction `sorted`). Cependant, re-trier la liste à chaque

ajout/suppression nécessiterait un tri global ce qui serait coûteux et non-optimal (on souhaite la plus petite value et non la liste trie).

Nous allons ainsi voir la structure de donnée de **Tas** qui propose l'accès à la valeur minimum de toutes les valeurs contenues. Un tas peut être représenté en utilisant un arbre binaire issu d'un tableau (car il s'agit d'un arbre complet) comme illustré figure 1. L'arbre binaire possède des noeuds ayant un index  $i$ , avec un fils gauche et un fils droit. Le tableau et l'arbre sont reliés de la façon suivante :

- La racine a la position  $i = 0$  (`get_racine`)
- Le parent a la position  $\lfloor (i - 1)/2 \rfloor$  (`get_parent`)
- Le fils gauche a la position  $2 \times i + 1$  (`get_fils_gauche`)
- Le fils droit a la position  $2 \times i + 2$  (`get_fils_droit`)

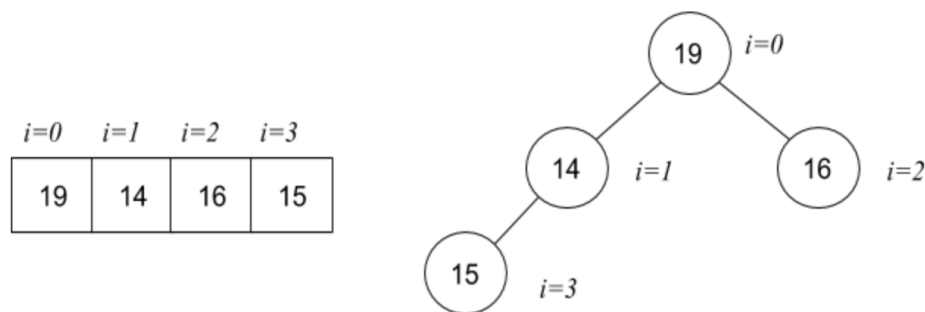


Figure 1: Exemple d'arbre binaire (droite) et son lien avec les index d'un tableau (gauche).

**Exercice 3.1** – Créez une structure de données de **Tas** similaire aux **Pile** et **File** (avec les méthodes `ajoute` et `supprime`) et qui implémente les méthodes `get` indiquées ci-dessus correspondant aux accès du tableau qui stocke les valeurs. Pour le moment ce tableau ne les trie pas. Ci-dessous un exemple de résultat d'accès à la racine et à ses deux enfants. Attention en Python les index de tableaux commencent à 0 (comme sur la figure 1).

```
>>> tas = Tas()

>>> for d in data:
>>>     tas.ajoute(int(d['moyenne']))

>>> r = tas.get_racine()
>>> fg, fg_i = tas.get_fils_gauche(0)
>>> ffg, ffg_i = tas.get_fils_droit(0)

>>> print(r, fg, ffg) # affiche racine, fis gauche et petit-fils gauche

19 14 15
```

**Exercice 3.2** – Maintenant nous souhaitons que l'ajout d'une valeur dans le tas conserve une relation d'ordre qui nous permettra ensuite d'accéder à la valeur minimale en temps constant  $O(1)$  (qui sera la racine de l'arbre, autrement dit le premier élément de la liste). Il faut pour cela ordonner l'arbre de façon à faire remonter à la racine la plus petite valeur. Cela nécessite d'implémenter une fonction **insérer** (que l'on utilisera à la place d'**ajoute**) dont le principe fonctionne comme suit :

1. Chaque nouveau noeud est rajouté comme dernier élément du tableau (à la fin donc)
2. Comparez ce noeud à son parent et si il est plus grand que ce parent inversez le
3. Répétez cela jusqu'à la racine

Implémentez la méthode **insérer** et vérifiez que votre tas affiche les valeurs comme suit :

```
>>> tas = Tas()
>>> for d in data:
>>>     tas.insérer(int(d['moyenne']))
>>> print(tas.afficher())

14 15 16 19
```

**Exercice 3.3** – L'élève avec la plus mauvaise moyenne a été convoqué et doit désormais être supprimé du Tas (et donc de la liste interne). Voici comment procéder afin de conserver la relation d'ordre et donc la propriété de tas :

1. Enlever l'élément racine du tableau
2. Déplacer le dernier élément du tableau à la place de la racine
3. Vérifier que la racine conserve la propriété de tas (qu'elle est inférieure à ses enfants); si ce n'est pas le cas alors prendre le plus grand des enfants, échanger sa place avec lui, et répéter.

Implémentez une méthode **enlever** qui supprime la racine et renvoie les valeurs du nouveau tas :

```
>>> tas = Tas()

>>> for d in data:
>>>     tas.inserer(int(d['moyenne']))

>>> tas.enlever(0)
>>> print(tas.afficher())

15 16 19
```

## 4 Bonus

**Exercice 4.1** – Pour continuer dans les statistiques, on souhaite déterminer les filières d'origine les plus courantes des étudiants (champ `filiere`). Écrivez un programme qui calcule le nombre d'étudiants par filière, les classe par nombre décroissant et les imprime ligne par ligne au format "`filiere nombre`". Pour le tri, pensez à convertir le dictionnaire en liste (à l'aide du code `list(dict.items())`) et utilisez ainsi la fonction `sort()`, qui prend en paramètre `reverse` pour définir si ordre croissant ou décroissant `reverse=True` (ou `False`), et enfin un paramètre `key` qui applique une fonction de choix d'attribut pour trier (inspirez vous de la fonction identique `key=lambda t: t`). Lorsque deux filières ont autant d'étudiants, elles sont classées par ordre alphabétique croissant. Pour le fichier d'exemple ci-dessus, le résultat attendu serait :

```
MP 2
PC 1
PSI 1
```

**Exercice 4.2** – Comparez votre implémentation des piles et files avec l'implémentation Python dans le module `queue`. Voici un exemple d'utilisation de ce module <https://docs.python.org/3/library/queue.html> :

```
import queue
pile = queue.LifoQueue()

for i in range(5): pile.put(i)

while not pile.empty():
    print(pile.get(), end=" ")
```

```
# 4 3 2 1 0
```

**Exercice 4.3** – Comparez votre implémentation des tas avec l'implémentation Python dans le module `heapq`. Voici un exemple d'utilisation de ce module <https://docs.python.org/fr/3.7/library/heapq.html> :

---

```
import heapq
tas = []

for i in range(5): heapq.heappush(tas, i)

while not len(tas) == 0:
    print(heapq.heappop(tas), end=" ")
```

# 0 1 2 3 4

---

**Exercice 4.4** – Réalisez une analyse empirique de la performance de vos classes et en particulier comparez les avec les implémentations équivalentes en Python. Illustrez comme sur la figure 2 avec le code ci-dessous :

---

```
%%
import time
import random
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

nvalues = [100, 500, 1000, 1500, 2000, 2500, 3000]
timesEtudiants1 = []

for i in nvalues:

    random.seed()
    p = 12**2 # Ordre de grandeur des valeurs
    liste = []

    for x in range(i): liste.append(random.randint(0, p))

    a=time.perf_counter()
    e1 = []
    for n in liste:
        e1.append(n)
    b=time.perf_counter()
    timesEtudiants1.append(b-a)

plt.plot(nvalues,timesEtudiants1, "r-", label="Etudiants 1")
plt.title("Comparaison des performances")
# %%
```

---

**Exercice 4.5** – Ecrire une fonction de tri par tas qui prend en entrée une liste de valeurs numériques quelconque, qui la transforme en tas, puis la tri en appliquant retirant à chaque étape la plus grande

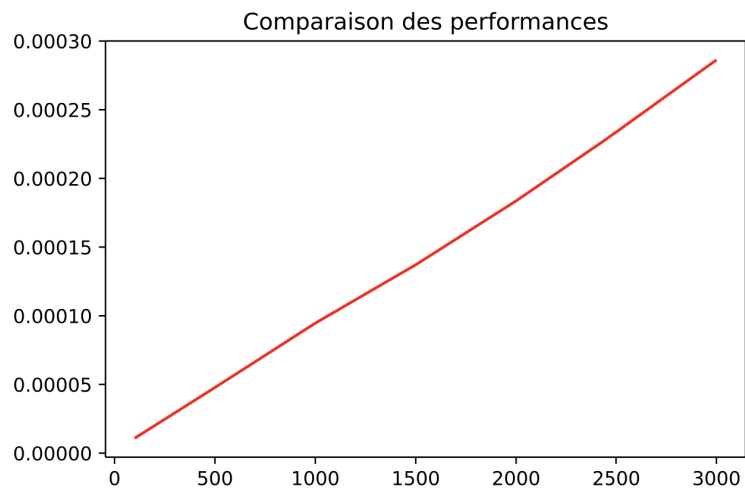


Figure 2: Exemple de résultat d'analyse.

valeur.