Files_Prio

January 13, 2019

1 Files de priorité

Marc Lorenzi
11 janvier 2019

```
In [1]: import random
```

1.1 1. Introduction

1.1.1 1.1 De quoi allons-nous parler ?

De nombreux problèmes algorithmiques stockent un ensemble de données et effectuent des opérations sur ces données. Quel genre d'opérations ? Cela dépend du problème.

Nous allons dans ce notebook nous intéresser aux problèmes pour lesquels, à chaque donnée, est associée une **priorité**. Qu'entendons-nous par priorité ? Une priorité est un élément d'un ensemble totalement ordonné. Pour prendre un exemple concret, les données de notre problème sont des tâches que doit exécuter un système d'exploitation. Certaines tâches sont urgentes (le curseur de la souris doit bouger), d'autres un peu moins (une fenêtre doit s'ouvrir), d'autres beaucoup moins (le système doit se mettre à jour). On affecte aux tâches extrêmement urgentes la priorité 1, à celles qui le sont un peu moins la priorité 2, ..., à la tâche qui consiste à préparer un café la priorité 50 (ou 0, cela dépend).

Un autre exemple ? Les données sont les sommets d'un graphe, les priorités sont leurs distances à un sommet fixé. Bref, Disposer de structures de données permettant le stockage d'objets possédant une priorité est peut-être intéressant.

Discuter en l'air d'une structure de type "ensembliste", c'est bien. Se demander quelles opérations nous aurons à faire sur cette structure est l'étape suivante.

Dans ce qui va suivre, les opérations que l'on désire effectuer sont les suivantes :

- Créer un nouvel ensemble.
- Tester si un ensemble est vide.
- Récupérer dans un ensemble un objet de priorité minimale.
- Insérer dans un ensemble un objet x ayant la priorité p.
- Supprimer d'un ensemble un objet de priorité minimale ET le renvoyer.

Et également

- Diminuer dans un ensemble la priorité d'un objet.

Une structure de données permettant d'effectuer ces opérations est appelée une **file de priorité**, ou **tas**. Pour être franc, les puristes considèrent le concept de file de priorité comme **abstrait** et la structure de tas comme une certaine implémentation **concrète** de ce type abstrait de données, celle que nous verrons à la section 3. Je vais tenter dans ce qui suit d'être puriste :-).

Remarque :

- Dans une file de priorité, plusieurs objets peuvent avoir la même priorité. Vous verrez parfois dans ce qui suit des phrases comme "UN objet de priorité minimale".
- En revanche, si un objet possède plusieurs priorités différentes, c'est le chaos. Nous voulons à tout prix pouvoir parler de LA priorité de l'objet x.

1.1.2 1.2 Listes aléatoires

Pour tester nos fonctions, il sera bien utile de pouvoir créer des listes aléatoires. Voici une fonction renvoyant une permutation aléatoire des entiers entre 0 et $n-1$. Cette fonction utilise l'algorithme de Fisher-Yates.

```
In [2]: def random_list(n):
            s = list(range(n))
            for i in range(n):
                j = random.randint(i, n - 1)
                s[i], s[j] = s[j], s[i]
            return s

In [3]: random_list(10)

Out[3]: [1, 5, 6, 3, 7, 0, 8, 4, 9, 2]
```

1.1.3 1.3 Files "aléatoires"

Pour créer une file "aléatoire", on part d'une file initialement vide et on insère dans la file des objets aléatoires. Pourquoi ces guillemets ? Parce que je ne sais pas mettre une probabilité sur l'ensemble des tas, puis prouver que la fonction `file_aleatoire` renvoie une file donnée avec une certaine probabilité.

Pourquoi n'avais-je pas mis de guillemets à "liste aléatoire" ? Parce je tiens à la disposition de qui le désire la **preuve** que l'algorithme de Fisher-Yates décrit plus haut renvoie effectivement une permutation aléatoire de la liste $[0, 1, \ldots, n-1]$.

```
In [4]: def file_aleatoire(n):
            prios = random_list(n)
            objets = random_list(n)
            f = nouvelle_file()
            for i in range(n): inserer(f, objets[i], prios[i])
            return f
```

Évidemment nous ne pouvons pas exécuter cette fonction puisque nous n'avons encore écrit aucune fonction sur les files de priorité.

1.2 2. Une implémentation naïve

Représentons une file de priorité par une simple liste tout en vrac, un tas d'objets quoi ! Les éléments de notre liste seront des couples (x, p) où x est un objet et p est un entier, la **priorité** de l'objet x.

Créer une nouvelle file et tester si une file est vide est immédiat. Ces deux opérations se font en complexité $O(1)$.

```
In [5]: def nouvelle_file(): return []

        def est_vide(f): return len(f) == 0
```

Insérer un objet x avec la priorité p est aussi immédiat et se fait en complexité $O(1)$. On utilise la méthode `append` du type `list` en priant Guido Van Rossum, créateur de Python, pour que cette opération soit en $O(1)$.

Remarque : Ce n'est en fait pas tout à fait vrai. Bien que l'implémentation des listes en Python soit un sujet fascinant, nous admettrons dans la suite que `append` est en $O(1)$.

```
In [6]: def inserer(f, x, p): f.append((x, p))
```

Maintenant nous pouvons créer des files "aléatoires".

```
In [7]: f = file_aleatoire(100)
        print(f)
```

```
[(31, 31), (22, 35), (97, 56), (54, 8), (53, 78), (2, 80), (20, 25), (9, 22), (30, 84), (5, 77),
```

Trouver un objet de priorité minimale ne pose pas non plus de problème. On nous demande de trouver le "plus petit" élément d'une liste. Mais remarquons que cette fois-ci la complexité de l'opération est $O(n)$, où n est la taille de la liste.

Petit aparté : Euh, au fait, dans une file, on stocke les couples (x, p) ou les couples (p, x) ? Vous avez déjà oublié ? Moi aussi. Imaginez un peu ce qui va se passer lorsque vous relirez votre code Python dans 2 ans. Ou pire, lorsque quelqu'un d'autre le lira parce que vous, vous aurez changé de boulot et que votre successeur doit écrire un patch pour votre programme ?

Maizalor, que faire ? Facile ! On définit deux petites fonctions.

Règle absolue : les nombres magiques tu n'utiliseras point. Tout le monde sera d'accord pour dire que `prio(t[k])` est plus lisible que `t[k][1]` (ou `[0]`, j'ai déjà oublié).

```
In [8]: def data(y): return y[0]
        def prio(y): return y[1]

In [9]: def top(f):
            n = len(f)
            if n == 0: raise Exception('File vide')
            else:
                m = 0
                for k in range(n):
                    if prio(f[k]) < prio(t[m]): m = k
                return t[m]
```

Pour supprimer de la file un objet de priorité minimale :

- On trouve d'abord l'indice de l'objet en question.
- On décale tous les éléments de la liste à droite de cet d'objet d'un cran vers la gauche.
- On supprime le dernier élément de la liste (eh oui, on a un élément de moins).

La complexité en pire cas de cette fonction que nous appellerons `pop` est clairement $O(n)$.

```
In [10]: def pop(f):
             n = len(f)
             if n == 0: raise Exception('File vide')
             else:
                 m = 0
                 for k in range(n):
                     if prio(f[k]) < prio(f[m]): m = k
                 y = f[m]
                 for k in range(m, n - 1):
                     f[k] = f[k + 1]
                 f.pop() # Le `pop` du type liste !
                 return y
```

```
In [11]: f = file_aleatoire(10)
         print(f)
         x, p = pop(f)
         print(x, p)
         print(f)
```

```
[(8, 6), (6, 3), (3, 2), (1, 0), (0, 7), (9, 4), (7, 9), (4, 5), (2, 1), (5, 8)]
1 0
[(8, 6), (6, 3), (3, 2), (0, 7), (9, 4), (7, 9), (4, 5), (2, 1), (5, 8)]
```

Pour diminuer la priorité d'un objet x :

- On trouve x dans la liste. Ici, c'est la fonction `data` qui entre en jeu.
- On ajuste sa priorité.

Ici encore, complexité en pire cas en $O(n)$. Remarquons que notre fonction marche aussi très bien pour **augmenter** la priorité de l'objet !

```
In [12]: def diminuer_prio(f, x, p):
             n = len(f)
             k = 0
             while k < n and data(f[k]) != x: k = k + 1
             if k == n:
                 raise Exception('%s non trouvé' % x)
             else:
                 f[k] = (x, p)
```

```
In [13]: f = file_aleatoire(10)
         print(f)

[(8, 9), (6, 4), (4, 8), (0, 7), (7, 2), (5, 6), (1, 3), (2, 1), (9, 0), (3, 5)]


In [14]: diminuer_prio(f, 8, -1)
         print(f)

[(8, -1), (6, 4), (4, 8), (0, 7), (7, 2), (5, 6), (1, 3), (2, 1), (9, 0), (3, 5)]
```

Exercice : On décide de représenter un tas par une liste **triée** par priorités croissantes. Écrivez les fonctions `inserer`, `top`, `pop` et `diminuer_prio`. Que deviennent les complexités de ces fonctions ? Qu'a-t-on gagné ? Qu'a-t-on perdu ?

Réponses et comparatif :

Fonction ..... Listes en vrac ..... Listes triées
`inserer` ..... $O(1)$ ..... $O(n)$
`top` ..... $O(n)$ ..... $O(1)$
`pop` ..... $O(n)$ ..... $O(n)$
`diminuer_prio` ..... $O(n)$ ..... $O(n)$

Bref, trier les listes n'est pas forcément souhaitable, et certainement pas souhaitable si on a beaucoup d'insertions à faire.

1.3 3. Une implémentation beaucoup plus efficace

1.3.1 3.1 Arbres binaires presque complets

Nous allons maintenant représenter un tas par ... un arbre, que nous allons implémenter comme ... une liste :-).

Définition : Un arbre binaire est **presque complet** lorsque tous les niveaux de l'arbre sont remplis, sauf le dernier, pour lequel seuls les noeuds les plus à gauche du niveau contiennent une information. Les noeuds les plus à droite du dernier niveau peuvent être vides.

Pour représenter un tel arbre on peut utiliser une liste t.

- $t[0]$ est la racine de l'arbre.
- $t[1]$ est le fils gauche de la racine.
- $t[2]$ est le fils droit de la racine.
- $t[3]$ est le fils gauche du fils gauche de la racine.
- $t[4]$ est le fils droit du fils gauche de la racine.
- $t[5]$ est le fils gauche du fils droit de la racine.
- $t[6]$ est le fils droit du fils droit de la racine.
- euh, normalement vous avez compris ...

Exercice : Dessinez un arbre binaire presque complet ayant 10 noeuds. Numérotez chaque noeud par son indice dans la liste qui représente l'arbre. Maintenant vous avez forcément compris.

Réponse : Au cas où vous n'avez toujours pas compris, il faut écrire les nombres de 0 à 9 de la gauche vers la droite et du haut vers le bas.

Exemple : la liste $t = [0, 1, 2, 3]$ peut être vue comme un arbre presque complet.

- À la racine on trouve 0.
- Le fils gauche est 1, le fils droit est 2.
- Le dernier niveau contient un unique noeud, le fils gauche du fils gauche de la racine, qui vaut 3.

Plus généralement, le fils gauche (s'il existe) du noeud stocké en position k dans la liste t est le noeud stocké en position $2k+1$, le fils droit est stocké en position $2k+2$.

```
In [15]: def fils_gauche(n): return 2 * n + 1
         def fils_droit(n): return 2 * n + 2
         def pere(n): return (n - 1) // 2
```

Exercice : Expliquez la fonction `pere`.

Proposition : Soit t un arbre binaire presque complet possédant n noeuds. La hauteur de t est $h(t) = \lfloor \lg(n+1) \rfloor$ où lg est le logarithme en base 2.

On convient ici que la hauteur de l'arbre vide est 0, et que la hauteur d'un arbre ayant un unique noeud est 1.

Démonstration : Une "vraie" preuve se ferait par récurrence sur la hauteur. Mais je préfère une explication plus "physique". Soit $k = \lfloor \lg(n+1) \rfloor$. On a $2^k \leq n+1 < 2^{k+1}$.

Pour $j = 0, \ldots, k-1$, le niveau j de l'arbre t contient 2^j noeuds. Cela donne au total

$$\sum_{j=0}^{k-1} 2^j = 2^k - 1$$

noeuds. Il reste donc encore des noeuds à placer donc nous avons au moins encore un niveau dans l'arbre. Combien précisément nous reste-t-il de noeuds à placer ? Eh bien

$$n - 2^k + 1 < 2^{k+1} - 2^k = 2^k$$

Ils tiennent donc tous sur le $k$ième niveau de l'arbre. Notre arbre est bien de hauteur k.

Définition : Soit t un arbre binaire presque complet. On dit que t est un tas lorsque

- t est vide, ou bien
- Le fils gauche et le fils droit de t sont eux-mêmes des tas, et les "valeurs" de leurs noeuds sont supérieures à la "valeur" de la racine de t.

La dernière condition équivaut à

- Pour tout noeud de t, le fils gauche et le fils droit de ce noeud ont une "valeur" supérieure à celle du noeud.

Écrivons une fonction qui prend une liste de couples (x, p) (objet, priorité) en paramètre et renvoie `True` si cette liste représente un arbre binaire presque complet, les "valeurs" des noeuds étant bien entendu les valeurs de la priorité p.

```
In [16]: def verifier_tas(t):
             n = len(t)
             if n == 0: return True
             else:
                 for k in range(n):
```

```
            fg = fils_gauche(k)
            fd = fils_droit(k)
            if fg < n and prio(t[k]) > prio(t[fg]): return False
            if fd < n and prio(t[k]) > prio(t[fd]): return False
        return True
```

Remarque : `prio((a, b))`, c'est a ou c'est b :-) ? Sans importance, un grand merci à la fonction `prio` !!!

Euh, je me suis avancé un peu vite. Pour tester il faut savoir ...

```
In [17]: t = [('a', 0), ('b', 1) , ('c', 2), ('d', 3)]
         verifier_tas(t)

Out[17]: True

In [18]: verifier_tas([('a', 0), ('b', 3) , ('c', 1), ('d', 2)])

Out[18]: False
```

Trouver un objet de plus petite priorité dans un tas est alors immédiat : il s'agit de sa racine.

```
In [19]: def top(t):
             if len(t) == 0: raise Exception('Tas vide')
             else: return t[0]

In [20]: top(t)

Out[20]: ('a', 0)
```

1.3.2 3.2 Insérer un objet

Pour insérer un objet x dans le tas t, représenté par une liste, on l'ajoute avec sa priorité en fin de liste. Évidemment, un tel ajout a de fortes chances d'être incompatible avec la structure de tas ! On fait donc remonter x dans l'arbre jusqu'à la bonne position : on compare la priorité de x avec celle de son père et on procède à un éventuel échange si le père de x a une priorité plus grande que celle de x. Puis on recommence avec le père et le grand-père de x, etc. C'est la fonction `bubble_up` qui se charge de faire remonter x à la bonne position.

```
In [21]: def inserer(t, x, p):
             t.append((x, p))
             bubble_up(t, len(t) - 1)

In [22]: def bubble_up(t, k):
             while k > 0 and prio(t[k]) < prio(t[pere(k)]):
                 t[k], t[pere(k)] = t[pere(k)], t[k]
                 k = pere(k)
```

Proposition : la complexité en pire cas de `inserer` est $O(\log n)$ où n est le nombre de noeuds de t.

Démonstration : la fonction `bubble_up` effectue une série de comparaisons sur les ancêtres d'un noeud. Le nombre d'ancêtres étant majoré par la hauteur de t, `inserer` effectue au plus $h(t)$ comparaisons, d'où le résultat..

La fonction `file_aleatoire` que nous avons écrite il y a un certain temps utilise maintenant la nouvelle fonction `inserer`.

```
In [23]: t = file_aleatoire(10)
         print(t)
```

```
[(7, 0), (0, 3), (3, 1), (8, 7), (9, 4), (4, 6), (1, 2), (2, 9), (6, 8), (5, 5)]
```

Notre fonction d'insertion fabrique-t-elle bien des tas ?

```
In [24]: verifier_tas(t)
```

```
Out[24]: True
```

1.3.3 3.4 Supprimer un objet de priorité minimale

Pour supprimer un objet x de plus petite priorité (la racine) dans le tas t :

- On remplace la racine de t sa feuille la plus à droite.
- On supprime la feuille en question.
- La racine viole alors peut-être la structure de tas : on l'échange si nécessaire avec celui de ses deux fils qui a la plus petite priorité, puis on recommence avec le fils concerné, etc.

La fonction `bubble_down` réalise le troisième point.

```
In [25]: def pop(t):
             n = len(t)
             if n == 0: raise Exception('Tas vide')
             else:
                 x = t[0]
                 t[0] = t[n - 1]
                 t.pop()
                 bubble_down(t, 0)
                 return x
```

```
In [26]: def bubble_down(t, k):
             while True:
                 i = k
                 fg = fils_gauche(k)
                 if fg < len(t) and prio(t[fg]) < prio(t[i]): i = fg
                 fd = fils_droit(k)
                 if fd < len(t) and prio(t[fd]) < prio(t[i]): i = fd
                 if i == k: break
                 t[i], t[k] = t[k], t[i]
                 k = i
```

```
In [27]: t = file_aleatoire(10)
         print(t)
         x, p = pop(t)
         print(x, p)
         print(t)
         print(verifier_tas(t))
```

```
[(5, 0), (9, 2), (7, 1), (3, 4), (8, 3), (1, 8), (4, 6), (2, 9), (0, 7), (6, 5)]
5 0
[(7, 1), (9, 2), (6, 5), (3, 4), (8, 3), (1, 8), (4, 6), (2, 9), (0, 7)]
True
```

Proposition : la complexité en pire cas de `pop` est $O(\log n)$ où n est le nombre de noeuds de t.

Démonstration : la fonction `bubble_down` effectue une série de comparaisons sur un chemin dans l'arbre qui part de la racine. Le nombre de noeuds sur un tel chemin étant majoré par la hauteur de t, la fonction effectue au plus $2h$ comparaisons où h est la hauteur de l'arbre, d'où le résultat.

1.3.4 3.5 Diminuer la priorité d'un objet

C'est là que le bât blesse : si nous voulons diminuer la priorité de l'objet x dans le tas t il va d'abord falloir trouver x. Mais où est-il ? Le mieux que l'on puisse faire est d'écrire une fonction de complexité $O(n)$, ce qui est **insoutenable**.

Règle : Un algorithme c'est comme un troupeau de moutons. La vitesse du troupeau c'est celle du mouton à 3 pattes.

Toutes les opérations sur les tas se font en complexité logarithmique ... sauf le mouton à 3 pattes `diminuer_priorite`. Il va donc falloir revoir notre structure de données :-(. Rassurez-vous, nous n'allons pas tout revoir ; nous allons juste compléter la structure que nous avons déjà.

Avant de nous atteler à cette tâche, petite digression : une application inattendue des tas est un algorithme de tri, appelé ... le **tri par tas**.

1.4 4. Le tri par tas

1.4.1 4.1 Comment ça marche ?

Soit s une liste que l'on désire trier par ordre croissant.

- On insère les éléments de s comme priorités d'objets quelconques dans un tas t initialement vide (les objets en eux-mêmes n'ont pas d'intérêt).
- On extrait un à un du tas les éléments de t et on insère leur priorité dans une liste initialement vide.

L'opération d'extraction récupère à chaque itération le plus petit élément du tas. On obtient donc une liste triée.

```
In [28]: def tri_tas(s):
             t, s1 = [], []
             for p in s: inserer(t, None, p)
             while not(est_vide(t)): s1.append(prio(pop(t)))
             return s1
```

```
In [29]: s = random_list(100)
         print(s)
         s1 = tri_tas(s)
         print(s1)
```

```
[25, 15, 69, 80, 82, 31, 84, 87, 70, 54, 20, 75, 36, 34, 42, 43, 83, 51, 10, 1, 12, 96, 27, 48,
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 2
```

Proposition : La fonction `tri_tas` a une complexité en $O(n \log n)$ où n est le nombre d'objets à trier.

Démonstration : À la $k$ième itération de la boucle des insertions, le tas possède k noeuds. La $k$ième insertion effectue donc au plus $\lfloor \lg(k+1) \rfloor \leq \lg(k+1)$ comparaisons. Le nombre total de comparaisons est donc majoré par

$$\sum_{k=0}^{n-1} \lg(k+1) = \sum_{k=1}^{n} \lg(k) \sim n \lg n$$

La boucle des extractions s'étudie de la même façon, avec un nombre de comparaisons majoré par

$$\sum_{k=0}^{n-1} 2\lg(k+1) \sim 2n \lg n$$

Ainsi, le nombre total de comparaisons effectué par `tri_tas` est majoré par

$$C_n \sim 3n \lg n$$

Le tri par tas est donc un algorithme de tri efficace. L'implémentation que nous avons donnée ci-dessus fait des recopies de données (création d'un tas, objets inutiles), mais ceci peut être évité pour améliorer la "constante cachée" dans la complexité de l'algorithme. Je n'en parlerai pas ici.

1.5 5. Représentation finale des tas

Nous y voilà. Nous allons écrire une **classe** `Tas`. L'idée pour pouvoir efficacement diminuer la priorité d'un objet est de savoir OÙ se trouve l'objet dans le tas. Un objet t de la classe `Tas` possède donc un **dictionnaire**, son champ `dico`. Si x est un objet du tas, alors `t.dico[x]` est l'indice de l'objet x dans la liste qui stocke les objets du tas.

Plutôt que de stocker une liste de couples, un objet de la classe `Tas` possède - un champ `data` qui contient les objets du tas - un champ `prio` qui contient les priorités des objets.

L'essentiel du code ci-dessous est une recopie de ce qui a déjà été fait dans le notebook avec quelques adaptations.

1.5.1 5.1 Le code

La classe `Tas` possède un certain nombre de méthodes.

- `__init__` est le constructeur. Il permet de créer un nouveau tas, vide.
- `echanger` échange dans le tas les objets situés aux indices i et j. La fonction se charge d'échanger les objets et les priorités, et de mettre à jour le dictionnaire.

- `top` renvoie un objet de plus petite priorité, ainsi que sa priorité.
- `pop` supprime un objet de priorité minimale et le renvoie.
- `inserer` insère un objet x avec la priorité p.
- `diminuer_prio` diminue la priorité de l'objet x à la valeur p.
- Il y a aussi, bien entendu, les méthodes `bubble_up` et `bubble_down`, analogues aux fonctions du même nom déjà étudiées plus haut.
- Enfin, les méthodes `verifier_tas` et `verifier_dico` permettent à un tas de s'auto-vérifier. Est-il bien un tas ? Son dictionnaire est-il cohérent ? Ces deux méthodes ont une complexité en $O(n)$, où n est le nombre d'objets du tas. Elles sont censées renvoyer tout le temps `True`. Ou alors c'est que nous nous sommes trompés quelque part !

La différence essentielle entre ces méthodes de la classe `Tas` et les fonctions que nous avons déjà écrites plus haut réside dans le fait que le dictionnaire doit garder la trace de la positions des objets. Chaque fois qu'un objet x est placé à la position i, on affecte à `dico[x]` la valeur i. Si un échange d'objets est effectué, deux valeurs du dictionnaire doivent être modifiées.

Je ne commenterai pas le code ci-dessous, sauf pour ce qui concerne la méthode `diminuer_prio`. Comment diminuer la priorité de l'objet x à la valeur p ?

- On trouve la position i de x grâce au dictionnaire.
- Si la priorité de x est déjà inférieure à p, on lève une exception.
- Sinon, on met la priorité de x à p. Évidemment la structure de tas est provisoirement défaillante, mais un appel à `bubble_up` la rétablit en complexité $O(\log n)$.

```
In [30]: class Tas:

             def __init__(self):
                 self.prio = []
                 self.data = []
                 self.dico = {}

             def __len__(self): return len(self.data)

             def echanger(self, i, j):
                 self.prio[i], self.prio[j] = self.prio[j], self.prio[i]
                 self.data[i], self.data[j] = self.data[j], self.data[i]
                 self.dico[self.data[i]] = i
                 self.dico[self.data[j]] = j

             def top(self): return (self.data[0], self.prio[0])

             def pop(self):
                 n = len(self)
                 if n == 0: raise Exception('Tas vide')
                 else:
                     p = self.prio[0]
```

```
            x = self.data[0]
            del self.dico[x]
            self.prio[0] = self.prio[n - 1]
            self.data[0] = self.data[n - 1]
            if len(self) > 1: self.dico[self.data[0]] = 0
            self.data.pop()
            self.prio.pop()
            self.bubble_down(0)
            return (p, x)

    def bubble_down(self, k):
        while True:
            i = k
            fg = fils_gauche(k)
            if fg < len(self) and self.prio[fg] < self.prio[i]: i = fg
            fd = fils_droit(k)
            if fd < len(self) and self.prio[fd] < self.prio[i]: i = fd
            if i == k: break
            self.echanger(i, k)
            k = i

    def inserer(self, x, p):
        self.prio.append(p)
        self.data.append(x)
        self.dico[x] = len(self) - 1
        self.bubble_up(len(self) - 1)

    def bubble_up(self, k):
        while k > 0 and self.prio[k] < self.prio[pere(k)]:
            self.echanger(k, pere(k))
            k = pere(k)

    def diminuer_prio(self, x, p):
        ix = self.dico[x]
        if self.prio[ix] < p:
            raise Exception('Priorité trop grande')
        else:
            self.prio[ix] = p
            self.bubble_up(ix)

    def verifier_tas(self):
        n = len(self)
        if n == 0: return True
        else:
            for k in range(n):
                fg = fils_gauche(k)
                fd = fils_droit(k)
                if fg < n and self.prio[k] > self.prio[fg]: return False
```

```
                    if fd < n and self.prio[k] > self.prio[fd]: return False
                return True

            def verifier_dico(self):
                for x in t.dico:
                    if t.data[t.dico[x]] != x: return False
                for i in range(len(self)):
                    if t.dico[t.data[i]] != i: return False
                return True
```

Exercice : Rajoutez à la classe `Tas` une méthode `augmenter_prio`. Puis écrivez une méthode `modifier_prio` qui appelle, selon les cas, la méthode `diminuer_prio` ou la méthode `augmenter prio`.

1.5.2 5.2 Tests

Vérifions que tout fonctionne. Créons des tas aléatoires et demandons leur de s'auto-vérifier.

```
In [31]: def tas_aleatoire(n):
             s = random_list(n)
             s1 = random_list(n)
             t = Tas()
             for k in range(n):
                 t.inserer(s1[k], s[k])
             return t
```

```
In [32]: def afficher_tas(t):
             print('indice objet prio')
             for k in range(len(t)):
                 print('%-7d%-6s%-4s' % (k, t.data[k], t.prio[k]))
             print(t.dico)
```

```
In [33]: N = 10
         t = tas_aleatoire(N)
         print(t.verifier_tas(), t.verifier_dico())
         afficher_tas(t)
```

```
True True
indice objet prio
0      5     0
1      6     1
2      3     2
3      0     3
4      7     4
5      1     9
6      9     8
7      4     7
8      2     6
9      8     5
```

```
{0: 3, 4: 7, 1: 5, 6: 1, 7: 4, 9: 6, 3: 2, 5: 0, 2: 8, 8: 9}
```

Exercice : Quelle est ci-dessus la plus grande valeur de N pour laquelle la création d'un tas aléatoire demande moins de 10 secondes ?

```
In [34]: t = tas_aleatoire(10)
         t.diminuer_prio(2, -1)
         print(t.verifier_tas(), t.verifier_dico())
         afficher_tas(t)
```

```
True True
indice objet prio
0      2     -1
1      4     1
2      5     0
3      3     3
4      1     2
5      6     6
6      8     5
7      0     9
8      7     8
9      9     4
{0: 7, 9: 9, 8: 6, 5: 2, 3: 3, 6: 5, 2: 0, 7: 8, 4: 1, 1: 4}
```

```
In [35]: t.pop()
         print(t.verifier_tas(), t.verifier_dico())
         afficher_tas(t)
```

```
True True
indice objet prio
0      5     0
1      4     1
2      9     4
3      3     3
4      1     2
5      6     6
6      8     5
7      0     9
8      7     8
{0: 7, 9: 2, 8: 6, 5: 0, 3: 3, 6: 5, 7: 8, 4: 1, 1: 4}
```

Exercice : Évaluez la cellule ci-dessus jusqu'à ce que le tas soit vide. Puis évaluez encore une fois pour lever l'exception "Tas vide".

1.5.3 5.3 Complexité des opérations de tas

Admettons que les opérations sur les dictionnaires Python s'effectuent en complexité $O(1)$. C'est effectivement le cas dans des circonstances "normales" que je ne détaillerai pas. Dans ce cas, la méthode `echanger` a une complexité en $O(1)$. Il s'ensuit les complexités en pire cas ci-dessous (où n est le nobre d'objets dans le tas) :

- `__init__` : $O(1)$.
- `echanger` : $O(1)$.
- `top` : $O(1)$
- `pop` : $O(\log n)$
- `inserer` : $O(\log n)$
- `diminuer_prio` : $O(\log n)$

1.5.4 5.4 Peut-on faire mieux ?

La réponse est **oui**, en utilisant des types d'arbres plus sophistiqués que les arbres binaires presque complets. Le lecteur intéressé pourra entre autres se documenter sur - les tas binomiaux - les tas de Fibonacci