Dictionnaires : Implémentation

1 Introduction

Dans ce chapitre, on voit comment implémenter la structure de dictionnaire vue en première année. Rappelons que cette structure permet de stocker des couples (k, e) où k est une *clé* et e un *élément* (ou *valeur*). Une contrainte est qu'une clé donnée ne peut être associée qu'à un seul couple. Les opérations qui doivent être permises sont les suivantes :

- Créer un dictionnaire vide : instruction $\boxed{d = \{\}}$
- Insérer une clé k et l'élément associé e : instruction $\boxed{d[k] = e}$
- Déterminer si une donnée k est une clé : expression booléenne $\boxed{k \text{ in } d}$
- Accès à l'élément associé e si la clé k est présente : instruction $\boxed{d[k]}$
- Modifier l'élément e_1 associé à une clé k : instruction $\boxed{d[k] = e_2}$. (si la clé k est présente)
  Si la clé k est absente du dictionnaire, cette instruction est une opération d'insertion.
- suppression du couple (k, e) à partir de la clé k : instruction $\boxed{\text{del } d[k]}$

Ce chapitre explique comment réaliser ces opérations de manière efficace, à l'aide de listes et de *fonctions de hachage*. En pratique, c'est de cette manière que sont implémentés les dictionnaires en Python.

2 Adressage direct et limites

2.1 Principe de l'adressage direct

Supposons que l'on sache que les clés à stocker soient des entiers de $K = [\![0, m-1]\!]$ avec m un entier pas trop grand. Il es alors possible de réaliser la structure de dictionnaire dans une liste L de taille m. En supposant qu'un élément associé à une clé ne puisse jamais être None, on pourra réaliser le dictionnaire avec la convention : " pour $k \in [\![0, m-1]\!]$, $L[k]$ =None si la clé k n'est pas présente, et $L[k]$ = l'élément e associé sinon ".

Voici un exemple avec un dictionnaire dont on sait que les clés ne peuvent être que des entiers de $[\![0, 9]\!]$.
Le dictionnaire contient 3 couples : $(1, 'abc'), (5, 54), (7, True)$.
Alors $L = [None, 'abc', None, None, None, 54, None, True, None, None]$
Il est facile d'implémenter toutes les opérations ci-dessus avec cette structure, et elles s'effectuent toutes en temps constant, sauf la création.

2.2 Limites de l'adressage direct

L'adressage direct, outre le fait qu'il impose une contrainte forte sur les clés (nécessairement des entiers positifs pas trop grands) demande également une complexité en mémoire $O(m)$ quel que soit le nombre de clés effectivement présentes dans le dictionnaire. Il est préférable d'avoir une complexité mémoire linéaire en ce nombre, pour ne pas gaspiller de la mémoire.

3 Tables de hachage de largeur fixe

Si l'ensemble des clés possibles est gros, la stratégie précédente ne fonctionne pas. L'idée des *fonctions de hachage* est de ramener l'univers des clés possibles vers une ensemble fini $[\![0, \omega-1]\!]$, avec ω pas trop grand. On utilisera alors une liste de taille ω pour stocker les couples (clé, élément). Plus précisément, avec D une telle

liste, et i un entier de $[\![0, \omega - 1]\!]$, $D[i]$ sera la liste des couples (clé, élément) tels que l'image de la clé par la fonction de hachage est i. En effet, une fonction de hachage n'est pas injective, et il se peut que deux clés k et k' aient la même image i via la fonction de hachage h. On verra comment gérer de telles *collisions*.

Une fonction de hachage est une fonction qui, à partir d'une donnée fournie en entrée, renvoie un nombre de taille fixe. Dans une utilisation en cryptographie, une fonction de hachage doit vérifier les propriétés suivantes :

- le résultat *paraît* aléatoire : modifier un tout petit peu l'entrée modifie énormément la sortie (le processus est toutefois déterministe : le résultat d'un hachage est toujours le même pour la même entrée)
- il est très difficile de construire des collisions : c'est-à-dire de trouver deux entrées différentes dont l'image par la fonction de hachage est la même.

Une fonction de hachage, aujourd'hui obsolète , est MD5, qui donne pour résultat un nombre hexadécimal à 32 chiffres, soit 128 bits.

Deux applications des fonctions de hachage :

- lorsqu'on télécharge un fichier d'installation, on peut vouloir s'assurer que la transmission s'est faite sans erreur. Le site officiel d'Ubuntu donne la valeur de hachage du fichier d'installation du système d'exploitation par la fonction de hachage SHA-256 : il suffit de vérifier que le hachage du fichier téléchargé est identique.
- un site web permettant l'authentification des utilisateurs ne doit pas stocker " en clair" les mots de passe de ses utilisateurs pour des raisons de sécurité. Ainsi, celui-ci ne stocke que les images de ces mots de passe par une fonction de hachage donnée. Lorsqu'un utilisateur rentre son mot de passe, l'image du mot de passe par la fonction de hachage est calculée, et c'est cette valeur qui est comparée à ce qui est enregistré sur le site web.

3.1 le hachage en Python

Python propose une fonction de hachage : **hash**, ou une méthode __hash__().

```
>>>hash(45)
45
>>>hash(45.0)
45
>>>45.__hash__()
45
>>>hash('mot')
6485514797961243303
>>>'mot'.__hash__()
6485514797961243303
>>>hash('mot!')
2127309913818875041
```

Si vous essayez chez vous, vous obtiendrez un résultat différent pour les chaînes de caractères car Python dispose d'une famille de fonctions de hachage , et en choisit une différente à chaque lancement de Python.
La fonction hash est efficace :

- Si i est un entier différent de -1 et si $-2^{61} + 1 < i < 2^{61} - 1$, alors $hash(i)$ vaut i.
  $hash(2 * *61 - 1)$ et $hash(-2 * *61 + 1)$ valent 0, $hash(-1)$ vaut -2.
- si x est un flottant égal à p/q, alors $hash(x)$ vaut $(int(p * M/q))\%M$, avec $M = 2^{61} - 1$
- Si la clé est une chaîne de caractères , la valeur de hachage est calculée avec une part de hasard à chaque nouvelle session, et a pour valeur un entier qui s'écrit sur 64 bits. C'est aussi le cas pour un type composé.
  On se sert du code ASCII de chaque caractère obtenu avec la fonction ord, et on construit une suite de blocs d'octets $b_m \cdots b_0$ et on utilise un polynôme p défini par $p(x) = b_0 + b_1 x + \cdots + b_m x^m$ avec une valeur fixée de x.

3.2 Utilisation pour la structure de dictionnaire

Fixons un entier ω (la largeur de hachage) et proposons une implémentation de la structure de dictionnaire dans une liste L de taille ω fixée. On peut forcer la fonction de hachage à prendre des valeurs entre 0 et $\omega - 1$ en utilisant simplement le modulo.
en supposant *omega* définie comme variable globale, on aura :

```
def hachage(x):
    return hash(x)% omega
```

ou

```
def hachage(x):
    return x.__hash__() % omega
```

Dans la suite, on note h cette fonction de hachage à valeurs dans $[\![0, \omega - 1]\!]$. La liste L contiendra ω listes, qu'on appelle des *alvéoles*. Lorsqu'on veut stocker un couple (k, e) dans le dictionnaire, on calcule la valeur $h(k)$. En notant $i = h(k)$, on rajoute à l'alvéole $L[i]$ le couple (k, e).
Cette méthode de gestion des collisions est appelée résolution par chaînage ou adressage ouvert.
De même, lorsqu'on veut tester si un couple de clé k est présent , on calcule $i = h(k)$ et on cherche dans l'alvéole $L[i]$ si un couple de clé k est présent, il faut parcourir toute l'alvéole dans le pire des cas.
On procède de manière similaire pour supprimer un couple à partir de sa clé, ou modifier l'élément associé à une clé.
Illustration d'une telle table de hachage pour une largeur $\omega = 5$.

.

3.3 Implémentation en Python

On va utilise la variable globale *omega* et la fonction *hachage* décrite plus haut.

- Création d'un dictionnaire vide : il s'agit de créer une liste de ω alvéoles, toutes vides :

```
def creer_dict():
    return[[] for i in range(omega)]
```

- test de présence d'une clé k. Il s'agit de voir si un couple de clé k existe dans ce dictionnaire . Il suffit de regarder dans l'alvéole d'indice $i = h(k)$; il est inutile de regarder toutes les alvéoles.

```
def est_presente(D,k):
    """teste si la clé k est présente dans le dictionnaire D"""
    i=hachage(k)
    for c in D[i]:
       if c[0]==k:
           return True
    return False
```

- élément associé à une clé supposée présente

```
def element(D,k):
    """renvoie l'élément associé à une  clé k , supposée  présente dans le dictionnaire D"""
    assert est_presente(D,k)
    i=hachage(k)
    for c in D[i]:
       if c[0]==k:
           return c[1]
```

- ajout du couple (k, e), k supposée non présente

```
def ajout(D,k,e):
    """ajoute le couple (k,e)  , k supposée non  présente"""
    assert  not est_presente(D,k)
    i=hachage(k)
    D[i].append((k,e))
```

- suppression du couple de clé k, k supposée présente

```
def suppression(D,k):
    """supprime le couple de clé k  ,k supposée   présente"""
    assert  est_presente(D,k)
    i=hachage(k)
    for j in range(len(D[i])):
        if D[i][j][0]==k:
            D[i].pop(j)
            return
```

- modification de l'élément associé à une clé k , k supposée présente

```
def modification(D,k,e):
    """modifie l'élément associé à une clé  , supposée   présente"""
    assert   est_presente(D,k)
    i=hachage(k)
    for j in range(len(D[i])):
        if D[i][j][0]==k:
            D[i].pop(j)
            D[i].append((k,e))
            return
```

Une autre méthode de gestion des collisions est de prendre la première place libre dans la liste L après l'indice $i = h(k)$, on étudie donc les éléments de L d'indice $(h(k) + j)$ modulo ω pour j allant de 1 à $\omega - 1$. Il s'agit d'un sondage linéaire.
Si les clés sont des entiers en majorité consécutifs, cette méthode n'est pas très efficace.
On peut aussi effectuer un sondage quadratique avec la première place libre d'indice $(h(k) + j^2)$ modulo ω.
En pratique, voir section suivante, on est sûr de trouver de la place, car la capacité de la liste est supérieure au nombre de couples à stocker.

3.4 Complexité

En supposant que l'on travaille sur des objets de taille bornée (de sorte que la complexité de la fonction de hachage est constante, tout comme le test d'égalité de deux clés) , la complexité des opérations précédentes (sauf la création) est linéaire en la taille de l'alvéole numéro $i = h(k)$, où k est la clé impliquée. Dans le pire des cas, on peut imaginer que toutes les clés du dictionnaire sont présentes dans la même alvéole, et dans ce cas, la la complexité n'est pas meilleure que si on avait utilisé naïvement une liste.
A contrario, si on suppose que la fonction de hachage a tendance à bien répartir les clés entre alvéoles, on peut imaginer que chacune d'entre elles a une taille environ n/ω, où n est le nombre de couples stockés, et ω la largeur de hachage.
Sans rentrer dans les détails, on admet que la complexité est de $O(1 + n/\omega)$ pour les opérations précédentes.

4 Tables de hachage de largeur variable

On a dit en première année que la complexité des opérations de dictionnaire pouvait être considérée comme en $O(1)$, ce qui ne peut clairement pas être atteint avec la stratégie précédente si ω est fixe.
Il suffit de faire varier la largeur de la table de manière à avoir toujours le rapport n/ω borné, avec n le nombre d'entrées stockées dans le dictionnaire.
On discute brièvement d'une stratégie efficace, sans coder.

Famille de fonctions de hachage. On a vu que le fonction de hachage de Python était à valeur dans un ensemble de taille 2^{64}, cette taille étant très élevée, elle dominera toujours très nettement les largeurs de hachage dont on pourrait avoir besoin en pratique.
En considérant des largeurs ω variables, on obtient modulo ω une famille de fonctions de hachage (h_ω), chacune à valeurs dans $[\![0, \omega - 1]\!]$.

Ajout d'un élément à une table de largeur variable
C'est essentiellement la seule opération à modifier, puisque les autres n'augmentent pas le nombre n d'entrées stockées. Prenons une stratégie où l'on veut garantir que l'on a toujours $n/\omega \leqslant C$ (la constante C importe peu, mais on veut une constante, en pratique en Python, cette constante vaut 2/3).
On veut rajouter une clé non présente dans la table, on suppose que n est le nombre d'entrées stockées .

Si $(1+n)/\omega \leqslant C$, on rajoute la clé dans la bonne alvéole. Sous réserve de bonne répartition des clés, la complexité est en $O(1)$.
Si $(1+n)/\omega > C$, on réarrange la table, par exemple en doublant sa largeur ω, c'est-à-dire que l'on travaille avec $\omega' = 2\omega$. Il faut alors, pour les n entrées (k, e) de la table, recalculer les valeurs $h_{\omega'}(k)$ pour placer les clés dans les bonnes alvéoles de la nouvelle table. On peut alors ajouter la nouvelle clé comme au point précédent. l'ajout de cet élément particulier (au moment de la bascule par rapport à la constante C) se fait alors en temps $O(n)$, mais l'avantage est que le ratio n/ω' a été divisé par 2, on pourra donc réaliser de l'ordre de n ajouts supplémentaires sans avoir à changer la largeur de la table, donc chacun en $O(1)$.
Les $1+n$ ajouts évoqués dans ce point ont donc au total une complexité en $O(n)+O(n) = O(n)$, soit une complexité moyenne pour chacun de $O(1)$.
On dit que les opérations ont une complexité en $O(1)$ *amortie*.

Preuve du mécanisme en Python . la fonction sys.getsizeof permet de connaître la taille en bits d'un objet. Le code suivant crée un dictionnaire et ajoute successivement des entrées, on demande la taille du dictionnaire en bits à chaque étape (sans rentrer dans les détails, la taille des clés et des valeurs n'est pas prise en compte ici, c'est seulement la taille de la "structure" du dictionnaire) .
Les sauts dans la taille montrent un réarrangement de la table. Les prochains sauts se produisent au 22ème, 43 ème, 86 ème $\cdots$ éléments.

```
import sys
D={}
for i in range(1,13):
    D[i]=0
    print(i," éléments, taille :" , sys.getsizeof(D))
```

```
1  éléments, taille : 232
2  éléments, taille : 232
3  éléments, taille : 232
4  éléments, taille : 232
5  éléments, taille : 232
6  éléments, taille : 360
7  éléments, taille : 360
8  éléments, taille : 360
9  éléments, taille : 360
10  éléments, taille : 360
11  éléments, taille : 640
12  éléments, taille : 640
```