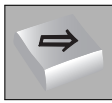
Chapitre 5

Les dictionnaires et leur implémentation

1 Rappels sur les dictionnaires

Ce sont des structures de données (`dict`) modifiables et non séquentielles ; un élément n'est pas repéré à l'aide d'un indice mais à l'aide d'un nom (sa **clé**). Un élément est constitué d'un **champ** : la donnée d'une clé et d'une valeur.

- La clé est un nombre, une chaîne ou un t-uplet. Ce sont des types non-mutables.
- La valeur peut être de type quelconque.



Le type `dict` permet la définition de conteneurs dont les valeurs sont repérées non plus par des indices mais par des clés.

Déclaration d'un dictionnaire

Un dictionnaire se définit par extension, par la suite des champs (la clé suivie de la valeur, séparées de `:`) entre accolades {.}.

• Déclaration

```
In [1]: Dic = { 'A' : 0, 'B' : 1, 'C' : 2, 'D' : 3 }
```

Ce qui est identique à :

```
In [1]: Dic = {}
In [2]: Dic['A'] = 0
In [3]: Dic['B'] = 1
In [4]: Dic['C'] = 2
In [5]: Dic['D'] = 3
```

• Opérations sur les dictionnaires

L'accès à un élément se fait à l'aide de sa clé :

```
In [6]: Dic['B']
Out[6]: 1
```

Pour ajouter un champ, affecter une valeur à une nouvelle clé. Pour supprimer un champ utiliser la fonction `del()`.

```
In [7]: Dic['E'] = 4     # Ajout d'un champ
In [8]: del(Dic['C'])    # Suppression d'un champ
In [9]: Dic
Out[9]: {'A' : 0, 'B' : 1, 'D' : 3, 'E' : 4}
```

La fonction `len` s'applique aux dictionnaires pour renvoyer son nombre de champs :

```
In [10]: len(D)          # Nombre de champs
Out[10]: 4
```

L'instruction `key in Dic` renvoie le booléen `True` si `key` est la clé d'un champ du dictionnaire `Dic`.

```
In [11]: 'A' in Dic
Out[11]: True
In [12]: 'Z' in Dic
Out[12]: False
```

Les dictionnaires sont itérables : `for k in Dic:` fait parcourir à la variable `k` toutes les clés du dictionnaires :

```
In [13]: for key in Dic:
...          print(key, Dic[key])
...
A 0
B 1
D 3
E 4
```

• Méthodes des dictionnaires

Les dictionnaires admettent les méthodes suivantes :

Méthodes des dictionnaires	
`D.pop(key)`	supprime le champ de clé `key` du dictionnaire, si ce champ existe.
`D.keys()`	retourne un itérateur des clés du dictionnaire `D`
`D.values()`	retourne un itérateur des valeurs du dictionnaire `D`
`D.items()`	retourne un itérateur des champs `(key,value)` de `D`
`D.copy()`	retourne une copie du dictionnaire `D`

Remarque : L'instruction `for k in D` n'est qu'une version concise de `for k in D.keys`.

2 Implémentation des dictionnaires

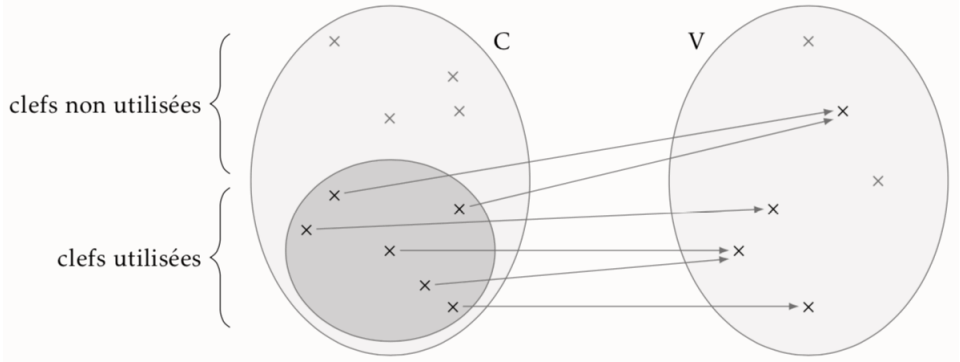
2.1 Introduction

Un dictionnaire (ou *Table d'associations*) est un type de donnée qui associe à des clefs leur valeur. Plus formellement :

Définition 2.1
Un dictionnaire est une partie D du produit cartésien $C \times V$ où C désigne l'ensemble des clefs et V l'ensemble des valeurs possibles, et tel que à toute clef $c \in C$ est associé au plus un élément $v \in V$ avec $(c, v) \in T$.

Les éléments de T sont appelés des associations

On peut voir un dictionnaire comme une application d'une partie de C dans V. La contrainte étant qu'à une clef ne peut être associé qu'au plus une valeur.



⇨ Dans un dictionnaire à une clef est associée au plus une valeur.

Un dictionnaire doit supporter les opérations (primitives) suivantes :
– *création* d'un dictionnaire vide.
– *ajout* d'une nouvelle association $(c, v) \in C \times V$ dans T.
– *suppression* d'une association (c, v) de T.
– *recherche* d'une association dans T de clef c donnée.
– *accès* pour *lecture* ou *modification* à la valeur v associée à une clef c dans T.

Toutes ces primitives doivent s'exécuter en temps borné (complexité $O(1)$) ou quasi borné (complexité $O(1)$ en moyenne).

C'est de leur implémentation que dépend la complexités de ces primitives. Par exemple implémenter un dictionnaire comme une liste de couples (c, v) produirait une complexité :
– bornée pour la *création*,
– quasi bornée pour l'*ajout*,
– mais linéaire pour les trois dernières *suppression, recherche, accès*.

2.2 Adressage direct et limitation

Plaçons nous dans la situation où les clés soient des entiers dans $[\![0, N]\!]$ avec N un entier pas trop grand. Le principe de l'adressage direct consiste à stocker les valeurs dans une liste `L` de la façon suivante :

$$\forall \texttt{k} \in [\![0, N]\!],\ \texttt{L[k]} = \begin{cases} \texttt{None} & \text{si k n'est pas une clé} \\ \texttt{v} & \text{si (k,v) est une association} \end{cases}$$

Bien sûr pour cela la valeur `None` ne pourra figurer parmi les valeurs possibles.

Par exemple pour stocker le dictionnaire {`(1,'abc'),(3,13),(6,True)`}, l'adressage direct implémenterait la liste :

```
[None, 'abc', None, 13, None, None , True]
```

Avec cette approche toutes les primitives s'exécuteraient en temps borné, hormis la création du dictionnaire qui s'exécute en temps $O(N)$ (où N majorant des clefs possibles, est fixé par avance).

Cette approche présente trop de limites :
– Les clés ne peuvent être que des entiers positifs (dans ce cas un dictionnaire présente rarement un avantage par rapport à une liste),
– La création est en $O(N)$, tout comme la complexité spatiale, quelque soit la taille du dictionnaire. Il serait préférable d'avoir une complexité spatiale linéaire en fonction de

la longueur du dictionnaire.

Pour ces raisons cette approche naïve n'est pas utilisée. Nous allons voir le principe utilisé pour une implémentation plus efficace plus loin. Regardons d'abord une autre approche intermédiaire

2.3 Table de hachage de largeur fixe

L'idée des fonctions de hachage consiste à ramener l'ensemble des clés possibles à une partie de $[\![0, N-1]\!]$ où N est un entier pas trop grand fixé, à l'aide d'une fonction de hachage $h : C \longrightarrow [\![0, N-1]\!]$. On utilisera alors une liste `L` de longueur N, et l'association (c, v) sera stockée à l'indice $h(c)$ de la liste `L`.

Mais l'entier N n'étant pas trop grand, la fonction h ne peut être injective : on peut avoir deux clés c, c' ayant même image par h : $h(c) = h(c')$. Les associations (c, v) et (c', v') devraient alors être stockées au même indice de la liste `L`. On parle de collision.

Afin de limiter les collisions, on demande à la fonction de hachage d'avoir une distribution uniforme.

Soit N un entier naturel, et C un ensemble de clefs. Une fonction de hachage est une application $h : C \longrightarrow [\![0, N-1]\!]$ qui à une clef associe un entier naturel.

On requière généralement qu'elle :

– soit facile à calculer,

– ait une distribution la plus uniforme possible (c'est à dire que tous les éléments de $[\![0, N-1]\!]$ aient environ le même nombre d'antécédents

L'ensemble C ayant en général un cardinal plus grand que N, on ne requière pas qu'elle soit injective.

Bien sûr créer une fonction de hachage est un problème complexe. Nous ne l'étudierons quasiment pas.

Python propose une fonction de hachage qui s'applique à tous les types de données non-mutables :

```
In [1]: hash("12345")
Out[1]: 6211590660532949358

In [2]: hash("12346")
Out[2]: -2486072076257595014

In [3]: hash(12345)
Out[3]: 12345

In [4]: hash(12345.)
Out[4]: 12345

In [5]: hash(12345.5)
Out[5]: 1152921504606859321

In [6]: hash((1,2,3,4,5))
Out[6]: 8315274433719620810

In [7]: hash([1,2,3,4,5])
```

```
TypeError: unhashable type: 'list'
```

Le résultat est un entier de 64 bits codé en complément à deux, c'est à dire un entier dans $[\![-2^{63}, 2^{63}-1]\!]$. Dans une liste de longueur 2^{64} les associations dont la fonction de hachage de la clé aurait une valeur entre 0 et $2^{63}-1$ seraient stockés en premier, suivis de ceux de valeurs allant de -2^{63} à -1.

• Utilisation pour l'implémentation des dictionnaires

Utilisons la fonction de hachage de Python pour implémenter les dictionnaires par une table de hachage à largeur fixe.

Fixons un entier Ω ; ce sera la largeur de la table de hachage. On peut forcer la fonction de hachage à prendre des valeurs dans $[\![0, \Omega]\!]$ en renvoyant son résultat modulo Ω :

```
# Largeur de Hachage : variable globale.
Omega = 100

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

On notera dans la suite h cette fonction de hachage. On va implémenter un dictionnaire à l'aide d'une liste `L` contenant Ω éléments, indicés de 0 à $\Omega - 1$ et appelés des alvéoles. Une association (c, v) sera stockée dans l'alvéole d'indice $h(c)$ de la liste `L`. Bien sûr à cause des collisions, plusieurs associations peuvent être stockés dans une même alvéole.

Pour tester si une clé c est présente, ou lire/modifier sa valeur, ou supprimer l'association :

– On calcule $h(c)$,

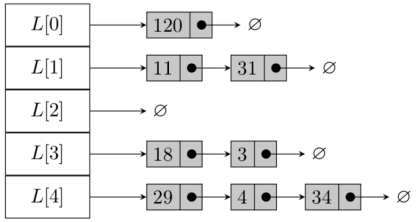
– On parcourt l'alvéole jusqu'à trouver, ou non, une association de clé c.

Le parcourt devra s'exécuter en temps linéaire sur le nombre d'association dans l'alvéole. Si la longueur des alvéoles est uniformément bornée par une valeur pas trop grande, on pourra considérer qu'elle s'exécute en temps quasi constant.

• Résolution des collisions

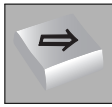
Pour résoudre les collisions, une alvéole est habituellement une liste chaînée contenant les différentes associations de l'alvéole. On parle d'une résolution par chainage.

Pour l'implémentation nous utiliserons plutôt une liste ; les temps d'accès seront semblables. Seule la gestion de la mémoire sera un peu plus couteuse.



Dans ce schéma nous avons omis les clés, et conservé seulement les valeurs ; il aurait fallu stocker les couples (clé,valeur). Avec cette même simplification on définirait la liste :

```
L = [ [120], [11, 31], [], [18, 3], [29, 4,34]]
```



Dans une résolution des collisions par chainage, chaque alvéole est une liste chainée : la première association pointe vers la suivante, etc. jusqu'à la dernière.

Nous pouvons maintenant écrire les fonctions qui implémentent nos dictionnaires. On utilisera la variable globale `Omega` et la fonction `hachage` donnés plus haut.

• La création d'un dictionnaire vide consiste à déclarer une liste de Ω listes vides.

```
def creer_dictionnaire():
    return [ [] for i in range(Omega) ]
```

• Déterminer la présence d'une clé : il s'agit de la rechercher en parcourant son alvéole.

```
def existe(Dic, c):
    """Teste si la clé c est présente dans Dic"""
    i = hachage(c)
    for k in Dic[i]:
        if k[0] == c:
            return True
    return False
```

• Accéder à la valeur d'une clé.

```
def valeur(Dic, c):
    """Renvoie la valeur associée à la clé c présente dans Dic"""
    i = hachage(c)
    for k in Dic[i]:
        if k[0] == c:
            return k[1]
```

• Ajout/modification d'un champ.

```
def ajout(Dic, c, v):
    """Si c n'est pas une clé présente, ajout de (c,v)
        sinon m.a.j. de la valeur de c en v"""
        i = hachage(c)
        for k in Dic[i]:
            if k[0] == c:
                k[1] = v
                return
        Dic[i].append((c,v))
```

• Suppression d'un champ supposé présent.

```
def suppression(Dic, c):
    """Suppression du champ de clé c"""
        i = hachage(c)
        for j in range(len(Dic[i])):
            if D[i][j][0] == c:
                Dic[i].pop(j)
                return
```

• Complexité

La création d'un dictionnaire est en $O(\Omega)$; *Omega* étant une constante on peut supposer que c'est en $O(1)$. Toutes les autres fonction ont une complexité en $O(K)$ dans le pire des cas où K est la longueur maximale d'une alvéole. Si la fonction de hachage répartit uniformément les associations dans les alvéoles, et si n est le nombre d'association, la complexité est alors en $O(n/\Omega)$ dans le pire des cas. Quoiqu'il en soit, avec cette implémentation, la complexité dans le pire des cas est linéaire en fonction de la taille n du dictionnaire (sauf la création d'un dictionnaire vide).

2.4 Table de hachage de largeur variable

Une table de hachage à largeur fixe ne permet pas d'obtenir une implémentation des dictionnaires avec des primitives s'exécutant en temps quasi-constant mais en temps $O(n/\Omega)$.

L'idée est alors de modifier la longueur de la liste lors de l'ajout d'élément, Ω, de façon à ce que le rapport n/Ω reste majoré par une constante, disons par exemple $n/\Omega \leqslant 2$. On débute avec une valeur Ω fixé, et dès que n dépasse 2Ω on allonge la liste.

Il est alors clair que toutes les primitives, autres que l'ajout d'un champ, se feront en temps borné.

Regardons ce qu'il se passe lors de l'ajout d'un élément.

La fonction de hachage h de Python renvoie 2^{64} valeurs possibles. Pour chaque valeur de $\Omega \leqslant 2^{63}$, la fonction $h_\Omega = h \bmod \Omega$ donne une fonction de hachage ; si les valeurs de h sont uniformément réparties, il en est de même de $h \bmod \Omega$.

• Ajout d'un élément.

Supposons qu'on ajoute un élément à une table de hachage de largeur Ω ayant n éléments avec $n/\Omega \leqslant 2$.

– Si $n+1 \leqslant 2\Omega$, on ajoute le champ dans son alvéole, sans change la largeur de la table.

– Si $n+1 > 2\Omega$; on double la largeur de la table à $\Omega' = 2\Omega$. On aura alors $(n+1)/\Omega' \leqslant 2$. Mais il faut alors replacer les n éléments de la table dans leur nouvelle alvéole, en utilisant la nouvelle fonction de hachage $h_{\Omega'}$. Pour chacun d'entre eux le déplacement se fait en $O(1)$, soit en tout un redimensionnement de la table en $O(n)$, plus l'ajout du nouvel élément en $O(1)$; bref un coup de $O(n+1) = O(n)$ pour l'ajout d'un élément provoquant le doublement de la table.

Mais la table contenant désormais $n+1$ éléments et $2n$ alvéoles, l'ajout de $n-1$ éléments supplémentaires ne modifiera plus la largeur de la table et l'ajout se fera pour chacun en $O(1)$.

Bref si l'ajout d'un élément est dans le pire des cas en $O(n)$, l'ajout de n éléments est dans le pire des cas aussi en $O(n)$, soit en moyenne $O(1)$ par élément. On parle de complexité en $O(1)$ en "temps amorti" : le coup de l'ajout de l'élément doublant la largeur de la table est amorti par le coup de l'ajout de $n-1$ éléments supplémentaires.

C'est cette stratégie qu'utilise Python, pour implémenter les dictionnaires, même si la constante qui majore n/Ω n'est pas égale à 2. Avec cette implémentation :

– *Création* en $O(1)$.

– *Présence* d'une clé en $O(1)$.

– *Lecture/Modification* d'un élément en $O(1)$.

– *Suppression* d'un élément en $O(1)$.

– *Ajout* d'un élément en $O(1)$ en temps amorti (mais linéaire dans le pire des cas).

Vérifions l'augmentation de taille lors de l'ajout d'un champ au dictionnaire à l'aide de la méthode `__sizeof__` qui renvoie l'emplacement mémoire occupé :

```
Dic = { }
for k in range(12):
    Dic[k] = k
    print(k+1,"elements ;","taille", Dic.__sizeoff__())
```

ce qui produit :

```
1 elements ; taille 264
2 elements ; taille 264
3 elements ; taille 264
4 elements ; taille 264
5 elements ; taille 264
6 elements ; taille 456
7 elements ; taille 456
8 elements ; taille 456
9 elements ; taille 456
10 elements ; taille 456
11 elements ; taille 456
12 elements ; taille 840
```

Python implémente les dictionnaires à l'aide de table de chainage de largeur variable ; lors de l'ajout d'éléments, il peut rallonger la largeur de la table de façon à ce que la rapport entre nombre d'éléments et largeur reste inférieur à une constante prédéfinie. Cette approche permet d'obtenir des primitives dont la complexité est quasi-constante.