Cours - Structure de données avec le dictionnaire

I Définition/présentation

La structure de dictionnaire permet d'associer une clé d'un ensemble K à une valeur d'un ensemble V. On l'appelle aussi tableau associatif. Elle a déjà été présentée en première année sous forme de boite noire. Le principe et de pouvoir utiliser une clé d'un type possiblement différent d'un entier et d'y associer une valeur d'un type quelconque, ce qui donne une association (clé,valeur). Un dictionnaire se présente donc comme un ensemble non ordonné de couple (clé,valeur) comme ceci : [(clé1,valeur1),(clé2,valeur2),(clé3,valeur3),...,(clé n,valeur n)].
A cette structure sont associées des fonctions primitives (qu'on appellera primitives par la suite), permettant de la manipuler efficacement (complexité la meilleure possible). La liste suivante est non-exhaustive :

— créer un dictionnaire

— supprimer un élément sur la base de sa clé

— insérer une paire clé-valeur

— lire une valeur sur la base de sa clé

Pour rappel, en Python :

```
d={} #déclaration d'un dictionnaire
d["une chaine"]=3 #insertion de la paire clé "une chaine"
d["autre chaine"]=7 #une autre
d[2]=[1,2] #association de la clé 2 à une liste

#contenu du dictionnaire [("une chaine",3),("autre chaine",7),(2,[1,2])]


d.keys() # retourne l'ensemble des clés du dictionnaire sous forme
                    # d'une structure particulière hors-cadre du cours.
```

Remarque 1 : dans l'exemple ci-dessus, même si en python il est possible d'associer des types différents de clés à des valeurs de types différents, cela peut mener à une certaine confusion. En général, le type des clés et le type des valeurs sont chacun homogènes.

Remarque 2 : il n'est pas possible d'associer une valeur à une clé de type non mutable. Ainsi, les entiers, chaînes de caractères et les tuples sont utilisables. Par contre, les listes ne le sont pas.

Un des avantages est de pouvoir utiliser des clés de type entier dont les valeurs peuvent être arbitrairement grandes sans devoir instancier une structure d'une taille dont l'ordre de grandeur est celle des clés. Par exemple, avec des clés de type entier on peut avoir ce genre de configuration : [(1,"mot1"),(10,"mot2"),(10000,"mot3"),...,(1000000000,"mot4")]. Dans ce cas, notre dictionnaire comporte 4 paires clé-valeur avec un indice allant jusqu'à 1000000000, ce qui n'est pas possible avec une simple liste ou un tableau. Dans ce cas, il faudrait disposer d'une liste d'environ 1000000000 valeurs dont seulement 4 seraient utiles. Quelle perte de place !

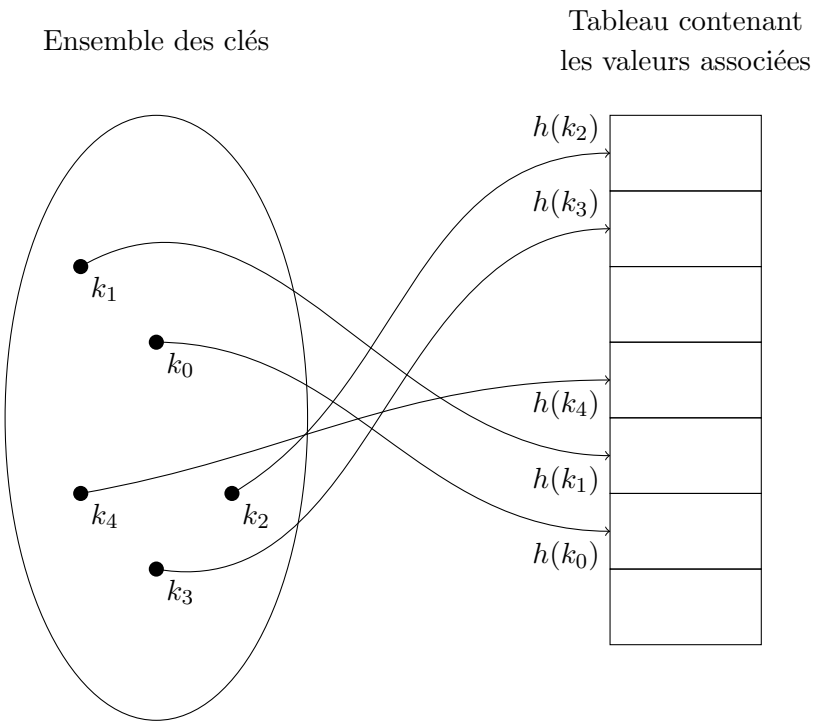

FIGURE 1 – Schéma représentant les liens entre l'ensemble des clés et les indices d'un tableau grâce à la fonction de hachage.

II Fonctionnement interne

II.1 Généralités

Bien évidemment, on souhaite avoir la rapidité en même temps que la souplesse d'utilisation. Pour cela, un dictionnaire est organisé en interne de la manière suivante :

— on garde une structure indicée comme une liste ou un tableau, ce qui garantit un temps d'accès à une valeur quelconque en temps constant (en $\mathcal{O}(1)$) ;

— on a besoin d'une fonction dont le rôle est de calculer un indice sur la base de la clé. Cette fonction est appelée *fonction de hachage*. On la notera h dans la suite du cours telle que $h : K \to \mathbb{N}$. Elle calcule l'indice correspondant à la clé en un temps très court pour ne pas dégrader les performances des primitives (un exemple très simple est exposé plus loin).

Une représentation du fonctionnement vous est fournie à la figure 1. Les clés sont notées k_n avec $n \in \mathbb{N}$. La fonction de hachage calcule un indice $h(k_n)$ de type entier ce qui permet de faire le lien (matérialisé par une flèche) entre une clé et un indice du tableau, Dans la suite, et pour simplifier notre exemple, nous partons avec des clés de type chaîne de caractères et des valeurs de type entier (à ne pas confondre avec les indices du tableau qui sont aussi des entiers). La structure pour stocker les paires clé-valeur sera un tableau. En suivant la syntaxe python, notre dictionnaire aura l'allure suivante : `{"key20":45 , "key31":2678 , "key2":954 , "key17":4269 , "key58":329}`.

```python
def h(chaine,n):
        s=0
        for c in chaine :
                s+=ord(c) # ord() est la fonction ascii() définie auparavant.
                                # elle est présente nativement dans python.
        return s%n
```

FIGURE 2 – Code python d'une fonction de hachage

II.2 Fonction de hachage

Comme mentionné précédemment, la fonction de hachage permet de calculer un entier à partir d'une clé quelconque. Plusieurs solutions pour assurer cette tâche sont envisageables mais cela sort du cadre de ce court. Dans l'idéal cette fonction doit :

— calculer en un temps très court ;

— être injective, c'est à dire fournir une valeur différente pour chaque clé.

Dans notre exemple, on peut simplement utiliser une fonction opérant la somme du code ASCII de chaque caractère constituant la clé et renvoyant le reste de la division entière entre cette somme et la taille du tableau. On est ainsi assuré de toujours obtenir une valeur entre 0 et l'indice maximal disponible. Formellement cela donne :

$$\begin{array}{rccl} h: & K & \to & V \\ & c & \mapsto & \left(\sum_{i=0}^{n-1} ascii(c_i)\right) mod\ n \end{array}$$

avec c une chaîne de caractères, c_i le caractère d'indice i de c et $ascii()$ une fonction prenant un caractère et retournant son code ASCII. Le code python d'une telle fonction est donnée à la figure 2. Pour l'exemple donné plus haut avec un tableau de sept éléments, cela donne la représentation de la figure 3.

II.3 Collision

Dans notre exemple avec des clés de type chaîne de caractères, l'ensemble est de cardinal infini. De l'autre côté, le tableau possède un nombre fini d'indices. La fonction ne peut donc être injective. Quelle que soit la taille de ce tableau, il y a forcément au moins un cas où pour deux clés différentes, la fonction de hachage produira le même entier. Dans ce cas, on dit qu'il y a **collision**. Un exemple est représenté à la figure 4 en ajoutant deux nouvelles paires clé-valeur à notre dictionnaire exemple (les deux dernières). Nous avons maintenant `{"key20":45 , "key31":2678 , "key2":954 , "key17":4269 , "key58":329 , "key873":7245 , "key33":97 }`. Les clés `"key2"` et `"key873"` comme paramètre de la fonction h donnent le même entier, ainsi qu'avec "key58" et "key33".

II.4 Résolution des collisions par chaînage

Une des solutions en cas de collision est le chaînage. Une illustration est fournie à la figure 5. L'idée consiste à placer les paires clé-valeur à l'indice désigné par la fonction de hachage. Dans notre exemple, pour les clés `"key2"` et `"key873"`, nous avons à chaque fois l'indice 1. Les deux paires clé-valeur sont placées à cet indice. Pour les clés `"key58"` et `"key33"`, on arrive sur l'indice 4.

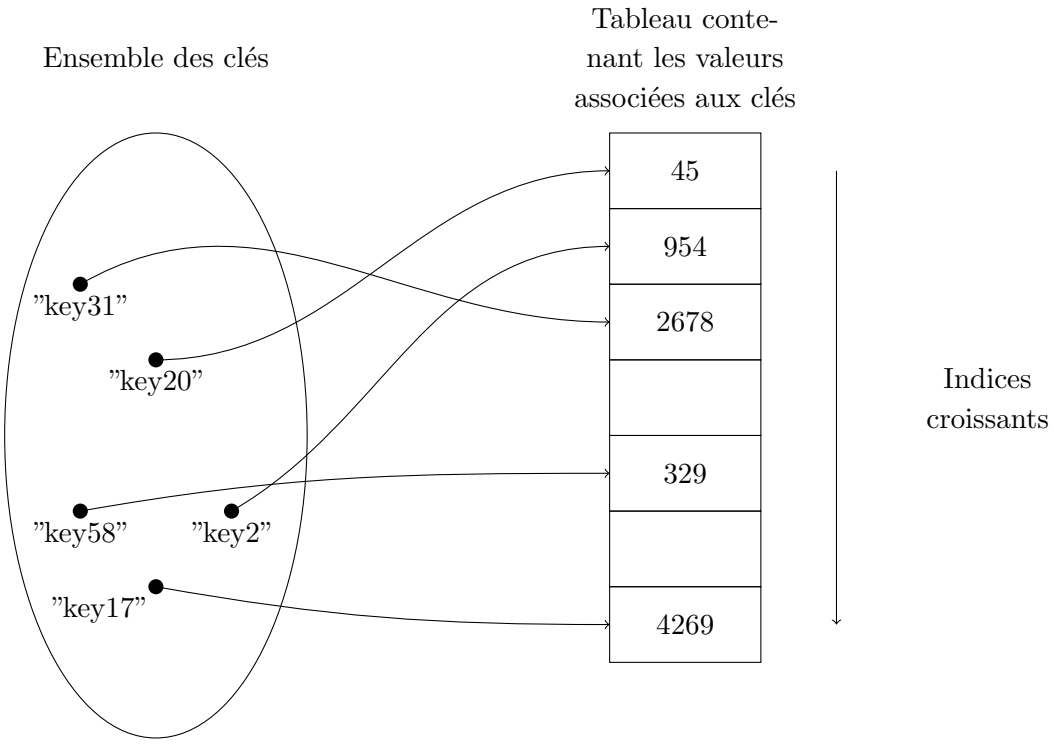

FIGURE 3 – Schéma illustrant le dictionnaire de l'exemple.

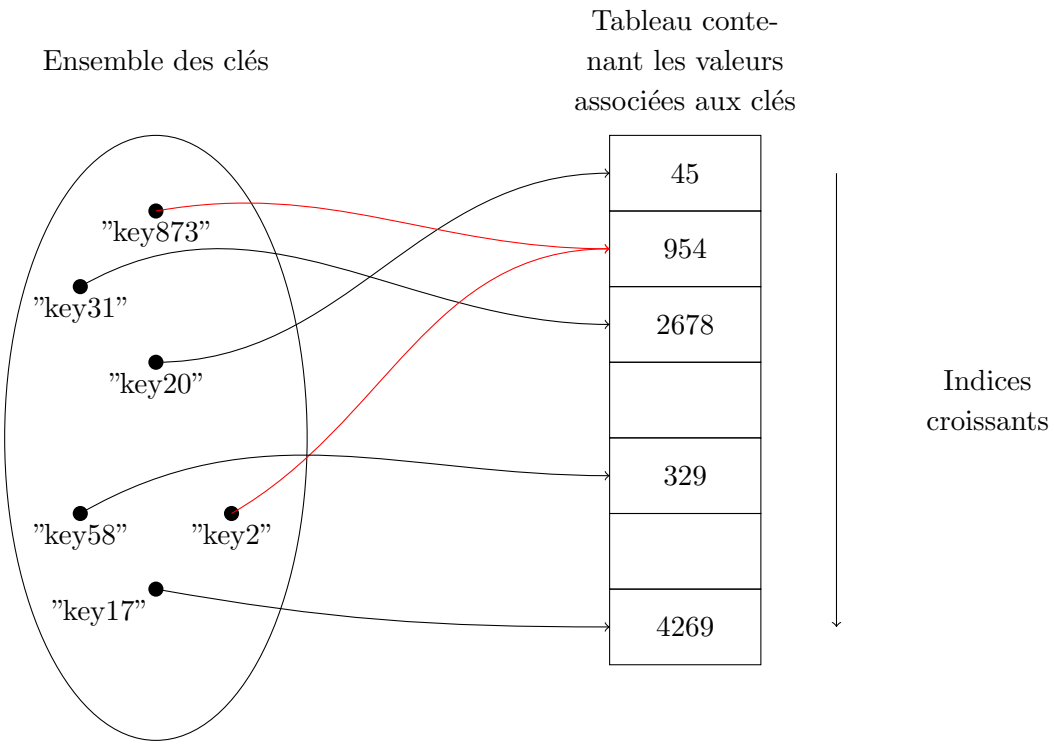

FIGURE 4 – Schéma illustrant une collision.

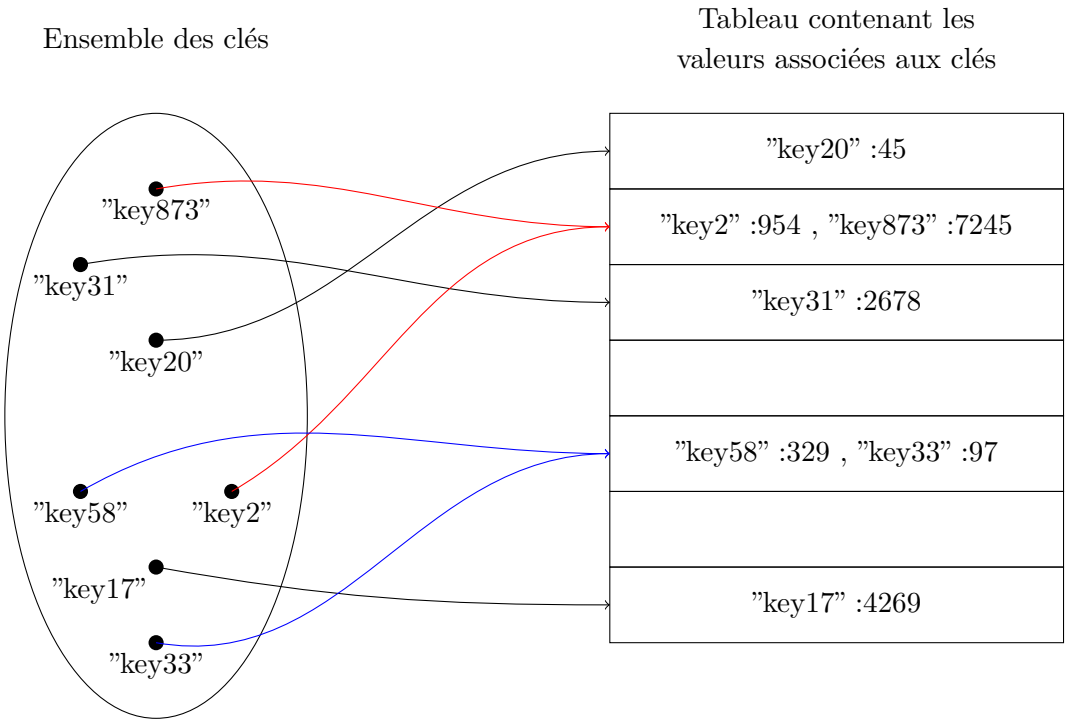

FIGURE 5 – Schéma illustrant deux collisions.

Plusieurs problèmes peuvent apparaître :

— la fonction de hachage est mal choisie et le résultat mène très souvent au même indice. On se retrouve alors avec un déséquilibre au niveau d'un indice avec un chaînage très important ;

— on suppose que la fonction de hachage répartit bien les données, mais le tableau contient trop de données à chaque indice et on perd l'avantage de départ, c'est à dire la recherche ou l'insertion d'une donnée en temps constant.

La solution consiste à doubler la taille du tableau au delà d'un certain "remplissage". Dans notre exemple, la taille du tableau est de 7. Si le nombre de données est supérieure au double de la taille du tableau alors on crée un nouveau tableau dont on double aussi la taille. La fonction de hachage donnera alors des résultats différents puisque la taille du tableau est un des paramètres. Toutes les données sont relogées dans le nouveau tableau avec une répartition bien meilleure, ce qui permet de garder des complexités intéressantes pour les primitives. Seules la création de ce nouveau tableau et la nouvelle répartition des données "coûtera" du temps, mais ce temps consommé ne sera que ponctuel, le reste du temps les opérations seront à coût constant. Difficile cependant de ne pas le compter dans la complexité. On parle de complexité amortie lorsqu'on considère que globalement les opérations sont à coût constant sauf de temps en temps lorsque le besoin est nécessaire de modifier la structure. Ci-dessous, un code python permettant de voir l'évolution de la taille en octets d'un dictionnaire. Celle-ci est grosso-modo doublée lorsque c'est nécessaire.

```
1  #permet de voir l'évolution dynamique d'un dictionnaire
2  d={}
3  for i in range(1001):
4      d[i]=i
5      print(sys.getsizeof(d))
```