Informatique du Tronc Commun

Chapitre n°2 Les dictionnaires

Introduction

Les dictionnaires sont utilisés en boîte noire dès la première année ; les principes de leur fonctionnement sont présentés en deuxième année. Ils peuvent être utilisés afin de mettre en mémoire des résultats intermédiaires quand on implémente une stratégie d'optimisation par programmation dynamique.

Objectifs

— Créer et utiliser un dictionnaire en Python (rappels de 1re année).

— Expliquer l'implémentation d'un dictionnaire par une table de hachage.

Pré-requis

TP de 1re année concernant les dictionnaires.

Programme officiel

Notions	Commentaires
Dictionnaires, clés et valeurs.	On présente les principes du hachage, et les limitations qui en découlent sur le domaine des clés utilisables.
Usage des dictionnaires en programmation Python.	Syntaxe pour l'écriture des dictionnaires. Parcours d'un dictionnaire.

Plan du cours

I Rappels de 1re année

I.1 Dictionnaires

Définition

Un **dictionnaire** est une structure de données dans laquelle les éléments sont indicés par des clés appartement à un ensemble $\mathcal{K}$ (au lieu d'être indicés par un entier dans une liste).
Soit $c \in \mathcal{K}$, une **clé** d'un dictionnaire `d`, alors `d[c]` est la **valeur** v de $\mathcal{V}$ qui correspond à la clé c.

Les opérations que l'on peut réaliser sur un dictionnaire sont :

— la recherche de la présence d'une clé,

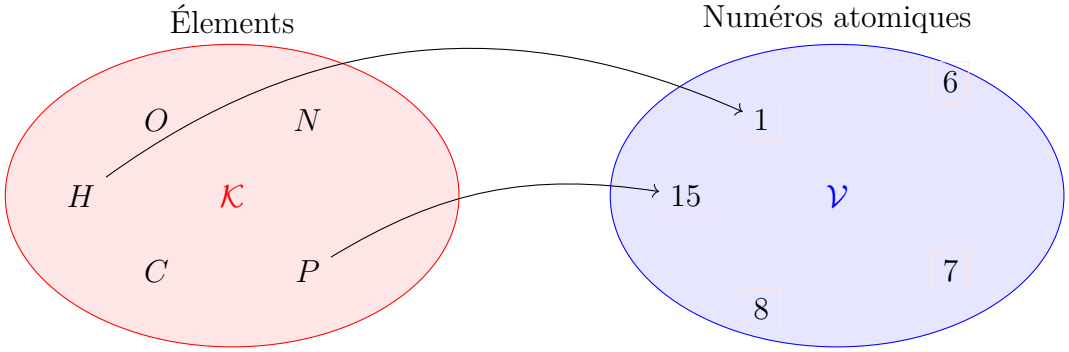
— l'accès à la valeur correspondant à une clé,

— l'insertion d'une valeur associée à une clé,

— la suppression d'une valeur associée à une clé.

Un dictionnaire relie donc directement une clé, qui n'est pas nécessairement un entier, à une valeur : pas besoin d'index intermédiaire pour rechercher une valeur comme dans une liste. Par contre, cette **clé est nécessairement d'un type immuable**, c'est-à-dire qui ne peut pas être modifiée.

Exemple 1. Élément chimique
On suppose que les éléments chimiques sont enregistrés via une chaîne de caractères : `"C", "O","H", "Cl", "Ar", "N"`. Soit `d`, un dictionnaire correspondant à la figure ci-dessous : les clés sont les symboles des éléments chimiques, les valeurs leurs numéros atomiques. Accéder au numéro atomique de l'élément `C` s'écrit : `d["C"]`.



Attention
Un dictionnaire n'est pas une structure ordonnée, à la différence des listes ou des tableaux.

Exemple 2. Usage des dictionnaires
Les dictionnaires sont utiles notamment dans le cadre de la programmation dynamique (**cf chapitre suivant**) pour la mémoïsation, c'est à dire l'enregistrement des valeurs d'une fonction selon ses paramètres d'entrée. Par exemple :

— a-t-on déjà rencontré un sommet lorsqu'on parcourt un graphe ?

— a-t-on déjà calculé la suite de Fibonacci pour $n = 4$?

Répondre à ces questions exige de savoir si pour une clé donnée il existe une valeur.
Si on utilise une liste pour stocker ces informations, par exemple `(n, fibo(n))`, les performances de l'exécution d'un algorithme peuvent être mauvaises : en effet, rechercher un élément dans une liste peut présenter dans le pire cas une complexité linéaire.
Si on implémente bien un dictionnaire, tester l'appartenance à un dictionnaire peut être de complexité constante, ce qui peut accélérer grandement l'exécution d'un algorithme.

I.2 Création d'un dictionnaire

■ Création d'un dictionnaire

À retenir
Les dictionnaires sont définis par des accolades et on met dans l'ordre la clé et la valeur correspondante séparés par un double point : `dico={clé1:valeur1,clé2:valeur2,}`

```
>>> dico = {} # création d'un dictionnaire vide
>>> type(dico)
<class 'dict'>
>>> dico = {'H':1,'C':6,'N':7,'O':8}
>>> print(dico)
{'H':1,'C':6,'N':7,'O':8}
```

■ Les **clés** d'un dictionnaire ne sont pas forcément toutes du même type.
Nous pouvons utiliser tout objet non mutable (immuable) pour les clés : des **entiers** (type `int`), des **flottants** (type `float`), des **chaînes de caractères** (type `str`), ou bien des **n-uplets** (type `tuple`).
À l'inverse, une liste ne peut pas être une clé.

■ Les **valeurs** peuvent être de **n'importe quel type de données**.

```
>>> dico2={3 : [] , 'cle' : 42 , (15,15) : [1,2,3,4]}
```

■ On peut également **créer un dictionnaire en compréhension** comme pour les listes :

```
>>> dico3={x:x**2 for x in range(1,6)}
>>> dico3
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Cette méthode de création est pratique mais difficile à manipuler (sauf dans un cas très simple comme ci-dessus).

I.3 Opérations sur un dictionnaire

I.3.a) Nombre d'éléments

La fonction `len` renvoie le nombre d'éléments associés d'un dictionnaire.

```
>>> len(dico)
4
```

I.3.b) Accès à un élément

On accède aussi à un élément avec la même notation entre crochet que pour les listes, mais en donnant cette fois la clé d'accès en argument : `dictionnaire[clé]`

```
>>> dico['C'] # accès à la valeur de la clé 'C'
6
>>> dico['P'] # erreur quand on demande l'accès à une clé non présente dans le dictionnaire
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: 'P'
```

I.3.c) Ajout et suppression d'un élément

L'opérateur `[]` est nécessaire pour ajouter un élément.

```
>>> dico['P']=14 # ajoute l'élément de clé 'P' de valeur 14
>>> print(dico)
{'H':1,'C':6,'N':7,'O':8,'P':14}
>>> dico['P']=15 # la clé 'P' existe déjà, on la modifie avec la valeur 15
>>> print(dico)
{'H':1,'C':6,'N':7,'O':8,'P':15}
>>> del dico['C'] # supprime l'élément de clé 'C'
>>> print(dico)
{'H':1,'N':7,'O':8,'P':15}
```

I.3.d) Test d'appartenance

Pour tester si une clé est présente dans le dictionnaire, on utilise `clé in dictionnaire` qui renvoie un booléen : `True` si la clé est présente, `False` sinon.

```
>>> 'O' in dico
True
>>> 'Ar' in dico
False
```

I.3.e) Parcours

Pour **parcourir les clés** d'un dictionnaire avec une boucle `for`, on utilise la syntaxe :

```
for cle in dictionnaire :
```

Par exemple, avec le dictionnaire précédent, si on veut construire une liste `L` contenant les valeurs du dictionnaire `dico` :

```
L=[]
for cle in dico :
    L.append(dico[cle])
```

On peut aussi le faire en définissant la liste par compréhension.

```
L= [ dico[c] for c in dico]
```

I.4 Copies

Si une valeur est d'un type mutable, on peut alors la modifier sans réaffectation. Tous les exemples précédents montrent bien que les dictionnaires sont des objets mutables en Python. Se poseront donc les **mêmes problèmes de copie que pour les listes**.

On pourra utiliser la méthode `copy()` pour réaliser une copie superficielle d'un dictionnaire.

```
>>> d={0:['a','b']}
>>> e=d
>>> d[1]=['c','d']
>>> d
{0: ['a', 'b'], 1: ['c', 'd']}
>>> e
{0: ['a', 'b'], 1: ['c', 'd']}
```

```
>>> d={0:['a','b']}
>>> e=d.copy()
>>> d[1]=['c','d']
>>> d[0].append(['e','f'])
>>> d
{0: ['a', 'b', ['e', 'f']], 1: ['c', 'd']}
>>> e
{0: ['a', 'b', ['e', 'f']]}
```

La fonction `deepcopy` de la bibliothèque `copy` est à utiliser pour réaliser une **copie profonde** de ce dictionnaire, lorsque les valeurs sont des listes de listes par exemple.

```
>>> from copy import deepcopy
>>> d={0:['a','b']}
>>> e=deepcopy(d)
>>> d[1]=['c','d']
>>> d[0].append(['e','f'])
>>> d
{0: ['a', 'b', ['e', 'f']], 1: ['c', 'd']}
>>> e
{0: ['a', 'b']}
```

II Hachage

L'objectif de cette partie est de comprendre comment les dictionnaires sont implémentés en utilisant les tables de hachage afin de garantir, notamment, un accès, une insertion ou modification en temps constant indépendant de la taille du dictionnaire.

II.1 Principe et définition

Définition : Fonction de hachage

Une **fonction de hachage** h est une fonction qui à une clé associe un entier appelée **valeur de hachage de la clé**.
Une fonction de hachage n'est pas toujours injective : deux clés peuvent avoir la même valeur de hachage.
Deux clés ayant même valeur de hachage créent une **collision**.

Définition : Table de hachage

Une **table de hachage** est constituée d'un tableau t et d'une fonction de hachage h. Elle implémente un dictionnaire sur un ensemble de clés $\mathcal{K}$ et un ensemble de valeurs $\mathcal{V}$.
Pour tout élément c de $\mathcal{K}$, $h(c)$ est l'indice de la case du tableau t auquel on stocke la valeur v.

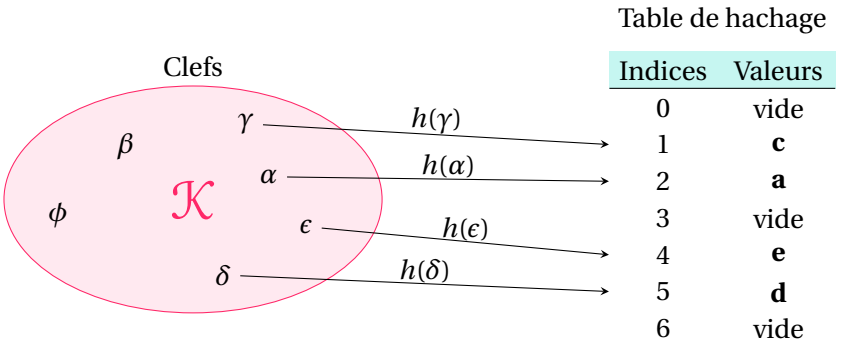


FIGURE 1 – Illustration de l'implémentation d'un dictionnaire par table de hachage.

La fonction de hachage h permet de calculer les indices de la case du tableau où stocker les valeurs associées aux clés. Soit la clé α associée à la valeur a : on calcule $h(\alpha)$ qui donne la case du tableau où trouver la valeur a. Ici $h(\alpha) = 2$.
Toutes les clés n'ont pas forcément de valeur associée à un moment donné de l'algorithme. Dans ce cas, à l'indice associé à cette clé, le tableau est vide.

Soit k le cardinal de $\mathcal{K}$, c'est à dire le nombre de clés possibles. On pourrait représenter un dictionnaire à l'aide d'un tableau de dimension k et une fonction bijective $h : \mathcal{K} \longrightarrow [\![0, k-1]\!]$. L'accès à un élément et l'ajout d'un élément seraient en $O(1)$, le temps de calculer la valeur de la fonction bijective pour une clé donnée.
Néanmoins, d'un point de vue complexité mémoire, cette solution n'est pas réalisable : le nombre de clés possibles est souvent immense alors que les clés effectivement utilisées sont moins nombreuses. Ce qui nous amènerait à réserver un espace mémoire bien supérieur aux besoins réels.

On utilisera un tableau de taille m petite devant le nombre de clés possibles : $m \ll k$.
En procédant ainsi, on renonce à l'injectivité de la fonction de hachage et donc à sa bijectivité : il pourra exister des clés différentes pour lesquelles la valeur de hachage calculée par la fonction de hachage sera la même. On engendre des **collisions**, c'est un problème qui devra être géré.

II.2 Fonctions de hachage

Le choix d'une fonction de hachage n'est pas évident. **Cette fonction doit permettre de générer un index dont la taille est inférieure à celle du tableau, tout en distinguant au mieux les clés, c'est-à-dire en évitant le plus possible les collisions.**

On recherche donc une fonction de hachage qui possède les caractéristiques suivantes :

- son **calcul doit être rapide**,
- **pour une même clé, on obtient un même code** (cohérence),
- **pour des clés différentes, on obtient des codes différents** (injectivité).
  Dans le cas contraire, on obtient un **collision** qu'on **cherchera à minimiser**.
- les codes doivent présenter une distribution uniformément répartie sur l'espace des indices, ce qui permet de minimiser les collisions.

Une fonction de hachage h peut procéder en deux étapes :

1. Une fonction h_e qui encode la clé d'entrée,
2. et une fonction h_c qui compresse le code précédent obtenu dans l'ensemble des indexes possibles (c'est-à-dire entre 0 et $m-1$, si on note m la taille du tableau).

Cela donne donc : $h = h_e \circ h_c$

Exemple 3. Encodage des clés
Pour encoder les clés qui sont des chaînes de caractères $c_0c_1...c_{p-1}$, on peut utiliser le code ASCII associé à chaque caractère, $\text{ascii}(c_i)$ (s'obtient avec la fonction `ord` de `Python`), puis calculer $\sum_{i=0}^{p-1} \text{ascii}(c_i) \times 2^{8+i}$.
Des chaînes de caractères similaires auront des codes différents. Et deux clés identiques auront le même code. C'est ce que l'on recherche.

Exemple 4. Compression des codes
Une fois les clés encodées, on cherche à **compresser la valeur encodée dans l'intervalle des index possibles** $[\![0, m-1]\!]$, si m est la taille de la table de hachage.
Pour cela on peut utiliser une division en calculant la valeur encodée modulo m.

Exemple 5. En Python
La fonction `hash` code les clés. Elle prend en argument un objet non mutable (entiers de type `int`, flottants de type `float`, chaînes de caractères de type `str` et des n-uplets constitués des éléments précédents de type `tuple`).
Puis il faut réduire chaque entier pour obtenir un entier appartenant aux index possibles, c'est-à-dire dans $[\![0, m-1]\!]$, si m est la taille de la table de hachage. Pour cela, on pourra utiliser l'opérateur modulo : `hash(cle)%m`.

II.3 Gestion des collisions

Deux méthodes permettent de gérer les collisions :

résolution par chainage : chaque case de la table contient une liste des paires (clé, valeur) qui ont la même valeur de hachage. Une fois la case trouvée, la recherche est alors linéaire en la taille de la liste.

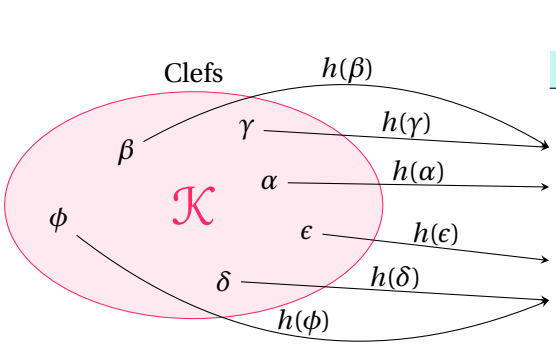


Table de hachage

Indices	Valeurs
0	vide
1	$[(\gamma, c), (\beta, b)]$
2	a
3	vide
4	e
5	$[(\delta, d), (\phi, f)]$
6	vide

FIGURE 2 – Illustration de l'implémentation d'un dictionnaire par table de hachage avec chainage. Les clés β et φ, engendrent des collisions. On stocke donc les valeurs possibles pour un même code (indice) dans une liste.

Cette solution induit une augmentation de la complexité car en cas de collision, on n'accède plus à l'élément directement : il faut le chercher dans une liste.

résolution par adressage ouvert : dans le cas d'une collision, on cherche une autre place vide dans le tableau pour stocker la valeur.

Cela induit que le nombre de clés utilisées est inférieur à la taille du tableau et ralentit l'accès à un élément. Lorsque le tableau est trop petit, on peut en changer pour un plus grand : cela a un coût également.

Pour implémenter des dictionnaires (de type `dict`), `Python` utilise des tables de hachage en adressage ouvert dont la taille évolue dynamiquement (comme le type `list`).

II.4 En `Python`

Voici comment fonctionne un dictionnaire en `Python` :

- Lorsqu'un dictionnaire (vide) `d` est créé, un tableau `T` est créé, d'une taille `m` donnée.
- Lorsque l'on insère une valeur `v` associée à une clé `c` (par l'affectation `d[c]=v`), la valeur de hachage de la clé `val_hach=hash(c)%m` est calculée, et la valeur v est affectée à la case d'indice `val_hach` du tableau `T`.
- S'il y a une collision (c'est-à-dire si la case `val_hach` contient déjà une valeur), `Python` parcourt les cases suivantes et remplit la première case non vide du tableau `T`.

  `Python` utilise l'adressage ouvert.
- Des mécanismes (cachés pour vous) assurent le fonctionnement de tout ceci : retrouver les valeurs déjà définies lorsque l'on utilise `d[c]`, augmenter la taille du tableau `T` lorsqu'il devient trop rempli, gérer les collisions, les ajouts/suppressions de clés, etc.

Les clés utilisables doivent pouvoir être passées en argument de la fonction `hash`, d'où les limitations annoncées précédemment : entiers de type `int`, flottants de type `float`, chaînes de caractères de type `str` et des n-uplets constitués des éléments précédents de type `tuple`. **Les listes ne peuvent pas être utilisées pour les clés.**

II.5 Complexités des opérations

À retenir

En pratique, on pourra supposer les complexités suivantes pour les opérations au programme, pour un dictionnaire `d` contenant n clés.

Opération	Commande	Complexité
Création vide	`d={}`	$\mathcal{O}(1)$
Création avec `n` valeurs	`d={c1:v1,...,cn:vn}`	$\mathcal{O}(n)$
Accès	`d[c]`	$\mathcal{O}(1)$
Insertion ou modification	`d[c]=v`	$\mathcal{O}(1)$
Test de présence	`c in d`	$\mathcal{O}(1)$
Parcours	`for c in d :`	$\mathcal{O}(n)$
Nombre d'éléments	`len(d)`	$\mathcal{O}(1)$