

Informatique pour tous - MP* - PC*

Partie 2 : Dictionnaires

Florent Pompigne
pompigne@crans.org

Lycée Buffon

année 2026/2027

Plan

- 1 Structure de dictionnaire
- 2 Utilisation en Python
- 3 Implémentation par tables de hachage

Structure de dictionnaire

Principe

Un dictionnaire est une structure dans laquelle des **valeurs** sont associées à des **clés**, chaque clé étant associée à une unique valeur. Un dictionnaire doit permettre de rechercher efficacement cette valeur associée à une clé donnée.

Exemples :

- un dictionnaire classique : les mots jouent le rôle de clés, et les définitions jouent le rôle de valeurs ;
- la table des symboles utilisée par un compilateur, qui associe à chaque nom de variable les informations utiles (type, adresse en mémoire...) ;
- on peut voir un tableau de taille n comme un cas particulier de dictionnaire, dans lequel les clés (les indices) sont limitées à l'intervalle $\llbracket 0, n - 1 \rrbracket$.

Opérations de la structure de dictionnaire

- **créer_dictionnaire()** : renvoie un dictionnaire vide.
- **ajouter_couple(D,c,v)** : modifie le dictionnaire D en lui ajoutant la valeur v associée à la clé c , si cette clé n'est pas déjà utilisée.
- **modifier_valeur(D,c,v)** : modifie le dictionnaire D en remplaçant la valeur précédemment associée à la clé c (si elle est présente) par la valeur v .
- **est_une_clé(D,c)** : teste si la clé c est présente dans le dictionnaire D .
- **valeur_associée(D,c)** : renvoie, si elle existe, la valeur associée à la clé c dans le dictionnaire D .
- **supprimer(D,c)** : modifie le dictionnaire D en lui retirant la clé c (et sa valeur associée), si elle est présente.

Plan

- 1 Structure de dictionnaire
- 2 Utilisation en Python
- 3 Implémentation par tables de hachage

Syntaxe Python des dictionnaires

- `{}` : renvoie un dictionnaire vide.
- `D[c] = v` : modifie le dictionnaire `D` en lui ajoutant la valeur `v` associée à la clé `c` (si cette clé n'est pas déjà présente), ou en remplaçant la valeur précédemment associée à `c` par `v`.
- `c in D` : teste si la clé `c` est présente dans le dictionnaire `D`.
- `D[c]` : renvoie la valeur associée à la clé `c` dans le dictionnaire `D` si elle existe, lève une erreur sinon.
- `del D[c]` : modifie le dictionnaire `D` en lui retirant la clé `c` (et sa valeur associée) si elle est présente, lève une erreur sinon.

On considère que toutes ces opérations ont une complexité en $O(1)$ (il s'agit en réalité d'une complexité moyenne, et non dans le pire cas).

Il est également possible de parcourir les clés du dictionnaire `D` en écrivant `for c in D:`

Exemple : dictionnaire des occurrences

Un exemple classique est le calcul d'un dictionnaire donnant pour chaque élément d'une liste son nombre d'occurrences :

```
def dico_occurrences(L):  
    D = {}  
    for x in L:  
        if x in D:  
            D[x] += 1  
        else:  
            D[x] = 1  
    return D
```

Exercice : En déduire une fonction renvoyant un élément ayant le plus grand nombre d'occurrences dans une liste non vide. Quelle est sa complexité ?

Exercice : inversion

Notons qu'on peut facilement obtenir la valeur associée à une clé du dictionnaire, mais pas l'inverse (il peut d'ailleurs y avoir plusieurs clés associées à la même valeur).

Écrire une fonction `inverser` prenant en entrée un dictionnaire `D` et renvoyant un dictionnaire inversant les clés et les valeurs, sur le modèle suivant :

```
>>> D = {'Alice': 1, 'Bob': 2, 'Charlie': 1}
>>> inverser(D)
{1: ['Alice', 'Charlie'], 2: ['Bob']}
```

Préciser la complexité de cette fonction.

Exemple : tri par dénombrement

Le tri par dénombrement permet de trier une liste contenant des entiers dans un intervalle $\llbracket a; b \rrbracket$:

```
def tri_denumerement(L, a, b):  
    D = dico_occurrences(L)  
    R = []  
    for k in range(a, b+1):  
        if k in D:  
            R += [k]*D[k]  
    return R
```

En notant $n = \text{len}(L)$, la complexité de ce tri est en $O(b - a + n)$, ce qui est intéressant lorsque l'amplitude $b - a$ des valeurs est en $O(n)$: le tri est alors de complexité linéaire.

Plan

- 1 Structure de dictionnaire
- 2 Utilisation en Python
- 3 Implémentation par tables de hachage

Listes d'associations

Si on souhaitait implémenter une structure de dictionnaire en Python, une première idée serait d'utiliser des **listes d'associations**, c'est-à-dire des listes de couples clé-valeur.

On aurait ainsi pour la fonction `valeur_associee` :

```
def valeur_associee(D, c):  
    for (cle, valeur) in D:  
        if c == cle:  
            return valeur
```

Le problème de cette implémentation est son manque d'efficacité : dans le pire cas, il faut parcourir l'ensemble du dictionnaire pour trouver la clé demandée, d'où une complexité linéaire.

Exercice : Implémenter les autres opérations des dictionnaires par listes d'associations.

Tables de hachage

Définition

Une **table de hachage** est une (grande) liste de listes d'associations. Lorsqu'on y cherche une clé, il faut commencer par déterminer quelle case (**alvéole**) de la liste considérer : c'est le rôle de la **fonction de hachage**, qui associe à chaque clé un entier (son **hash**), qui correspond à l'indice de son alvéole.

Lorsque deux clés distinctes tombent dans la même alvéole, on dit qu'il y a une **collision**. Une bonne fonction de hachage doit être rapide à calculer, et provoquer peu de collisions. Les clés sont alors bien réparties entre les alvéoles, et le coût d'une recherche dans une liste d'associations reste faible.

Lorsque le nombre moyen de clés par alvéole devient trop grand, on crée une table plus grande et on y réinsère tous les couples.

Exemple

Table à $m = 5$ alvéoles ; les clés sont des entiers, de hash $h(c) = c \% m$.

Après ajout des clés 12, 7, 20 et 33, de valeurs v_1, v_2, v_3, v_4 :

alvéole	liste d'associations
0	$[(20, v_3)]$
1	$[\]$
2	$[(12, v_1), (7, v_2)]$
3	$[(33, v_4)]$
4	$[\]$

Les clés 12 et 7 sont en collision. Pour chercher la clé 7, on ne parcourt que la liste de l'alvéole $7 \% 5 = 2$.

Bilan

- **Complexité.** Avec n clés bien réparties dans m alvéoles, une liste contient en moyenne n/m couples : si m reste en $O(n)$, les opérations sont en $O(1)$ en moyenne. Dans le pire cas (toutes les clés dans la même alvéole), elles sont en $O(n)$.
- **Clés autorisées en Python.** Le hash d'une clé ne doit jamais changer : une clé ne peut donc pas être un objet mutable comme une liste. On utilise des entiers, des chaînes de caractères, des tuples.