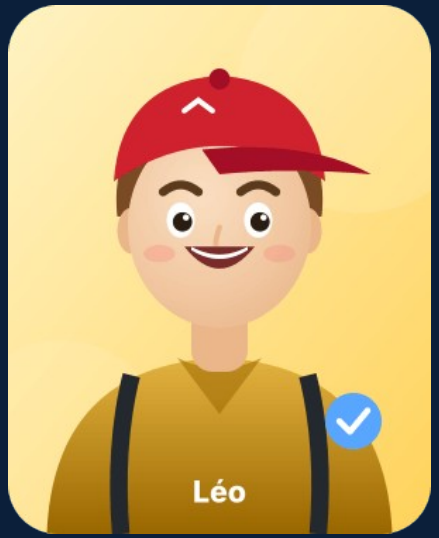
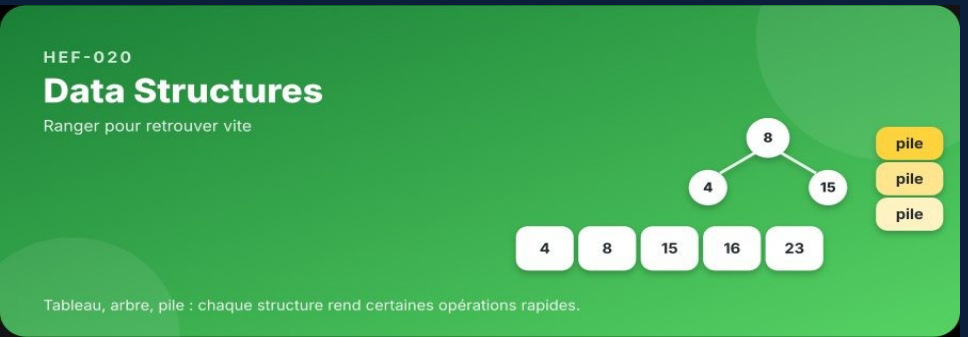
HEF-020 · MISSION 04 / 10

Tables de hachage : retrouver sans chercher

Comment retrouver une valeur presque instantanément, sans parcourir toute la collection ?





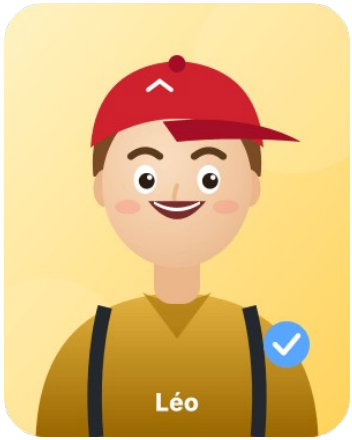
Badge : Magicien du hachage

75 min · débutant

L'histoire

HashBox a maintenant 2 millions de comptes. Quand un membre se connecte, le serveur doit retrouver son compte à partir de son identifiant. Avec une liste, c'est trop lent. Pourtant, le dictionnaire Python y arrive en un clin d'œil.

> *« Ce n'est pas de la magie, dit Sam. C'est une idée très maligne. Elle s'appelle le hachage. »*



Tu découvres la structure la plus utilisée de l'informatique moderne.

Ta mission

À la fin, je serai capable de…

1. expliquer le principe d'une table de hachage
2. décrire ce qu'est une fonction de hachage
3. expliquer une collision et une façon de la gérer
4. implémenter une petite table de hachage
5. dire ce qui rend une clé utilisable (hachable)

Défi d'échauffement

Défi d'échauffement

Une bibliothèque a 10 000 livres. Au lieu de les ranger par ordre alphabétique, on décide : **le livre va dans l'étagère numéro (somme des lettres du titre) modulo 100**. Comment retrouves-tu un livre ? Que se passe-t-il quand deux livres tombent sur la même étagère ?

Le concept

L'idée

Une **table de hachage** (*hash table*) associe des **clés** à des **valeurs** et permet de retrouver une valeur à partir de sa clé en un temps quasi constant.

Principe :

1. On a un tableau de **cases** (appelées *seaux*, *buckets*).
2. Une **fonction de hachage** transforme la clé en un nombre.
3. Ce nombre, modulo le nombre de cases, donne **l'indice** où ranger la valeur.

```
clé "ada"  ──hachage──▶ 9234 ──mod 8──▶ case 2
clé "bob"  ──hachage──▶ 1007 ──mod 8──▶ case 7
```

Le concept

Les collisions

Deux clés différentes peuvent tomber dans la même case : c'est une **collision**. C'est inévitable (il y a plus de clés possibles que de cases).

Solution courante : le **chaînage**. Chaque case contient une petite liste de couples (clé, valeur).

```
class TableHachage:
    def __init__(self, nb_cases=8):
        self._cases = [[] for _ in range(nb_cases)]

    def _indice(self, cle):
        return hash(cle) % len(self._cases)

    def mettre(self, cle, valeur):
        case = self._cases[self._indice(cle)]
        for i, (k, _) in enumerate(case):
```

Les collisions (suite)

```
            if k == cle:
                case[i] = (cle, valeur)
                return
        case.append((cle, valeur))

    def lire(self, cle, defaut=None):
        for k, v in self._cases[self._indice(cle)]:
            if k == cle:
                return v
        return defaut
```

Pourquoi c'est rapide

Si la fonction de hachage répartit bien les clés, chaque case contient très peu d'éléments : on regarde une case, pas toute la collection.

Le concept

Pourquoi c'est rapide (suite)

Si trop d'éléments s'entassent, on **agrandit** la table (on la reconstruit avec plus de cases). Ce rééquilibrage s'appelle le *rehachage*.

Clés hachables

Une clé doit avoir un hachage qui **ne change pas** : nombres, textes, tuples de valeurs immuables. Une **liste** Python ne peut pas servir de clé (elle est modifiable).

🧠 ***À retenir** : un dictionnaire Python (dict) est une table de hachage. Quand tu écris d["ada"], tout ce mécanisme travaille pour toi.*

Exemple pas à pas

Problème : compter combien de fois chaque mot apparaît dans un texte.

```
def compter_mots(texte):
    compteur = {}
    for mot in texte.lower().split():
        compteur[mot] = compteur.get(mot, 0) + 1
    return compteur

print(compter_mots("le chat et le chien et le loup"))
# {'le': 3, 'chat': 1, 'et': 2, 'chien': 1, 'loup': 1}
```

Sans table de hachage, il faudrait pour chaque mot parcourir la liste des mots déjà vus. Avec le dictionnaire, chaque mise à jour est quasi instantanée.

Une collision en pratique avec 4 cases :

Exemple pas à pas

Exemple pas à pas (suite)

Clé	Indice calculé	Contenu de la case
"a"	1	[("a", 5)]
"e"	1 (collision)	[("a", 5), ("e", 9)]
"b"	2	[("b", 7)]

Pour lire `"e"` : on calcule l'indice 1, puis on parcourt les 2 éléments de la case.

À toi, avec aide

Ajoute à la classe `TableHachage` : `supprimer(cle)` et `taille()` (nombre de couples). Vérifie avec 100 clés pour qu'il y ait des collisions avec 8 cases.

À toi, avec aide · réponse

Indices progressifs

Indice 1

`supprimer` : trouve la case avec `_indice`, cherche la clé, retire le couple.

Indice 2

`taille` : somme des longueurs de toutes les cases.

Indice 3

Test : mets 100 clés, supprime 40, vérifie `taille() == 60` et que les clés supprimées renvoient la valeur par défaut.

À toi, avec aide · réponse

Réponse

```
class TableHachage:
    def __init__(self, nb_cases=8):
        self._cases = [[] for _ in range(nb_cases)]

    def _indice(self, cle):
        return hash(cle) % len(self._cases)

    def mettre(self, cle, valeur):
        case = self._cases[self._indice(cle)]
        for i, (k, _) in enumerate(case):
            if k == cle:
                case[i] = (cle, valeur)
                return
        case.append((cle, valeur))

    def lire(self, cle, defaut=None):
        for k, v in self._cases[self._indice(cle)]:
            if k == cle:
                return v
```

Réponse (suite)

```
        return defaut

    def supprimer(self, cle):
        case = self._cases[self._indice(cle)]
        for i, (k, _) in enumerate(case):
            if k == cle:
                case.pop(i)
                return True
        return False

    def taille(self):
        return sum(len(c) for c in self._cases)

t = TableHachage()
for i in range(100):
    t.mettre(i, i * i)
for i in range(40):
    t.supprimer(i)
assert t.taille() == 60
```

Réponse (suite)

```
assert t.lire(10) is None
assert t.lire(50) == 2500
print("ok")
```

À toi, en solo

Ajoute à `TableHachage` l'**agrandissement automatique** : quand `taille() > 2 × nombre_de_cases`, double le nombre de cases et replace tous les couples.
Vérifie que, après 1 000 insertions, la case la plus remplie contient peu d'éléments (affiche le maximum). Compare avec 8 cases fixes.

À toi, en solo · réponse

Pistes de correction

```
class TableHachage:
    def __init__(self, nb_cases=8):
        self._cases = [[] for _ in range(nb_cases)]
        self._n = 0

    def _indice(self, cle):
        return hash(cle) % len(self._cases)

    def _agrandir(self):
        anciennes = self._cases
        self._cases = [[] for _ in range(len(anciennes) * 2)]
        for case in anciennes:
            for k, v in case:
                self._cases[self._indice(k)].append((k, v))

    def mettre(self, cle, valeur):
        case = self._cases[self._indice(cle)]
        for i, (k, _) in enumerate(case):
            if k == cle:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
                case[i] = (cle, valeur)
                return
        case.append((cle, valeur))
        self._n += 1
        if self._n > 2 * len(self._cases):
            self._agrandir()

    def plus_remplie(self):
        return max(len(c) for c in self._cases)

t = TableHachage()
for i in range(1000):
    t.mettre(i, i)
print(len(t._cases), t.plus_remplie())
assert t.plus_remplie() < 15
```

Sans agrandissement, avec 8 cases, la case la plus remplie contiendrait environ 125 éléments : on retomberait sur une recherche lente.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Utiliser une liste comme clé	`TypeError: unhashable`	Utiliser un tuple ou un texte
Croire que l'ordre est garanti par le hachage	Le hachage ne trie pas	Utiliser d'autres structures si l'ordre compte
Mauvaise fonction de hachage	Tout tombe dans la même case, retour à une recherche lente	Une bonne répartition est essentielle
Oublier de comparer les clés dans la case	Mauvaise valeur renvoyée en cas de collision	Toujours comparer `k == cle`

Mini-boss

Mini-boss

Construis un **mini-moteur de recherche** : à partir d'une liste de documents (textes courts), crée un **index inversé** : dictionnaire `mot → ensemble des numéros de documents contenant ce mot`.

1. Écris `construire_index(documents)`.
2. Écris `chercher(index, mot)` et `chercher_tous(index, mots)` (documents contenant **tous** les mots).
3. Teste avec 5 documents de ton choix, dont un cas sans résultat.
4. Explique pourquoi c'est bien plus rapide que de relire tous les documents à chaque recherche.

Mini-boss · réponse

Pistes de correction

```
def construire_index(documents):
    index = {}
    for numero, texte in enumerate(documents):
        for mot in set(texte.lower().split()):
            index.setdefault(mot, set()).add(numero)
    return index

def chercher(index, mot):
    return index.get(mot.lower(), set())

def chercher_tous(index, mots):
    ensembles = [chercher(index, m) for m in mots]
    return set.intersection(*ensembles) if ensembles else set()

docs = ["le chat dort", "le chien court", "le chat court vite", "un oiseau vole",
   "chat et chien"]
idx = construire_index(docs)
print(chercher(idx, "chat"))                    # {0, 2, 4}
print(chercher_tous(idx, ["chat", "court"]))    # {2}
```

Mini-boss · réponse

Pistes de correction (suite)

```
print(chercher(idx, "poisson"))                # set()
```

La construction se fait une fois ; chaque recherche est ensuite une lecture dans le dictionnaire au lieu d'une relecture de tous les textes.

Quiz éclair

Quiz éclair

1. Une collision, c'est :

A. deux clés dans la même case
B. une panne
C. un doublon de valeurs

2. Une table de hachage permet de retrouver par clé en temps :

A. proportionnel à la taille
B. quasi constant en moyenne
C. impossible

3. Une clé de dictionnaire doit être :

A. modifiable

Quiz éclair

Quiz éclair (suite)

B. une liste
C. hachable (immuable)

4. 🔁 *Révision mission 03* : Une pile suit la règle :

A. dernier arrivé, premier sorti
B. premier arrivé, premier sorti
C. aléatoire

5. 🔁 *Révision mission 02* : Dans une liste chaînée, atteindre le 5e élément :

A. est immédiat
B. demande de suivre les maillons
C. est impossible

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

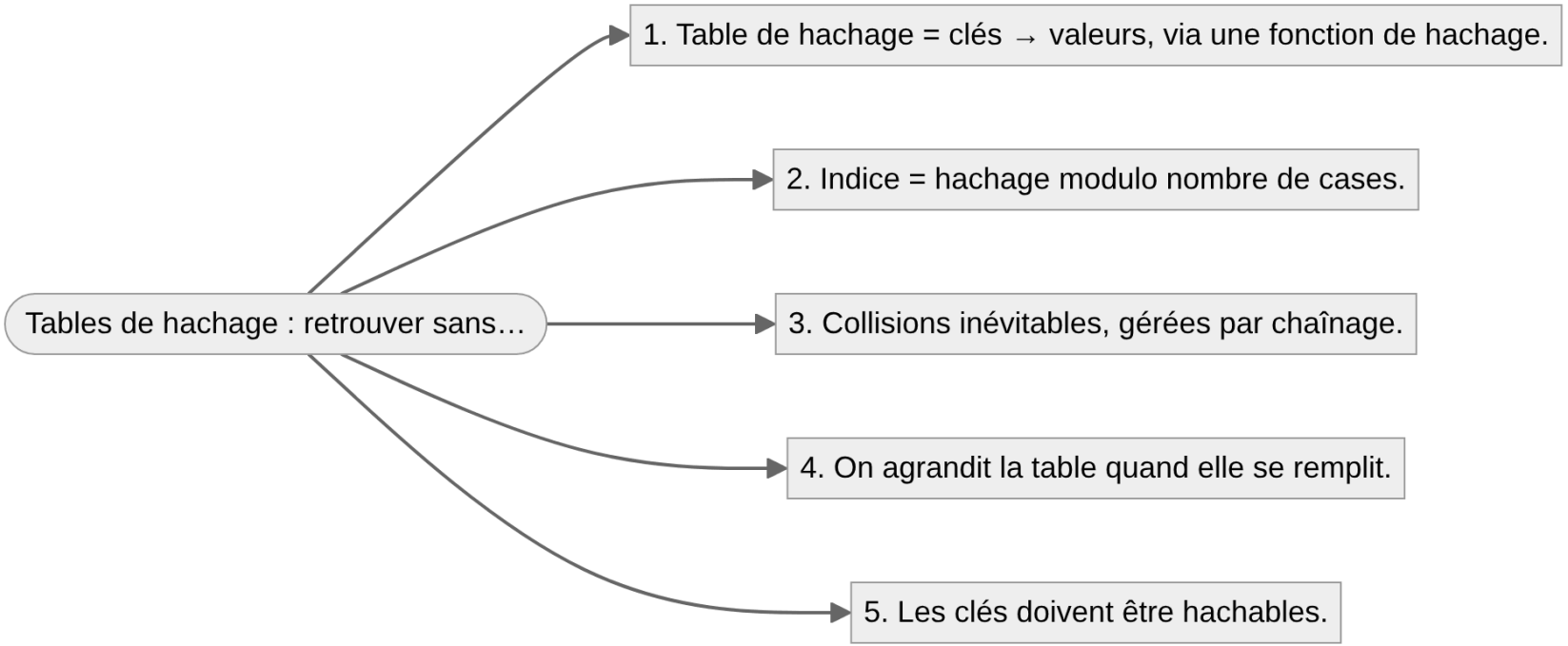
1. Quelle situation de ta vie ressemble à une table de hachage (un « numéro » qui te mène directement au bon endroit) ?
2. Pourquoi la qualité de la fonction de hachage compte-t-elle autant ?

À retenir

1. Table de hachage = clés → valeurs, via une fonction de hachage.
2. Indice = hachage modulo nombre de cases.
3. Collisions inévitables, gérées par chaînage.
4. On agrandit la table quand elle se remplit.
5. Les clés doivent être hachables.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Magicien du hachage

Prochaine mission : Ensembles et dictionnaires

