

tp_collisions_2025

October 7, 2025

Dans ce TP, les clés sont toujours des entiers (pour simplifier)

1 Q1

Dans cet exercice, les dictionnaires sont implantés par une liste appelée *tableau sous-jacent* :

```
TAILLE = 5
table = [None] * TAILLE
def hachage(cle, TAILLE):
    #renvoie le modulo de la clé par la taille

def inserer(dico, cle, valeur):
    #inserer (cle,valeur) dans le dictionnaire selon le h de la clé
```

Ecrire les fonctions `hachage` et `inserer`. On ne cherche pas à gérer les collisions.

```
# Exemple d'utilisation
inserer(table,12, "Alice")
inserer(table,7, "Bob")
```

```
print(table)
```

Que constate-t-on ?

```
[3]: *****
# On simule un dictionnaire avec un tableau de taille fixe
TAILLE = 5
table = [None] * TAILLE

def hachage(cle, TAILLE):
    return cle % TAILLE

def inserer(table,cle, valeur):
    i = hachage(cle,len(table))
    table[i] = (cle, valeur)

# Exemple d'utilisation
inserer(table,12, "Alice")
inserer(table,7, "Bob")
```

```
print(table)  # observer le rangement
```

2 Q2

Modifier la fonction d'insertion pour qu'elle affiche un message signalant la présence de collisions.

```
[5]: # Test
inserer(table, 12, "Alice")
inserer(table, 17, "Bob")  # Collision !

Collision entre 12 et 7
Collision entre 17 et 12

[6]: inserer(table, 22, "Charlie")  # collision avec 12
print(table)

Collision entre 22 et 17
[None, None, (22, 'Charlie'), None, None]
```

3 Q3

On implante la gestion des collisions par chaînage. Les alvéoles sont maintenant des listes.

```
TAILLE = 5
table = [[] for _ in range(TAILLE)]
```

Modifier l'insertion.

```
[8]: # Test
TAILLE = 5
table = [[] for _ in range(TAILLE)]

inserer(table, 12, "Alice")
inserer(table, 17, "Bob")
inserer(table, 22, "Charlie")
inserer(table, 27, "Alice2")
inserer(table, 32, "Bob2")
inserer(table, 37, "Charlie2")
print(table)

[[], [], [(12, 'Alice'), (17, 'Bob'), (22, 'Charlie'), (27, 'Alice2'), (32, 'Bob2'), (37, 'Charlie2')], [], []]
```

```
[9]: l = [1, 2]
dir(l)
```

```
[9]: ['__add__',
      '__class__',
      '__class_getitem__',
```

```
'__contains__',
'__delattr__',
'__delitem__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__getitem__',
'__getstate__',
'__gt__',
'__hash__',
'__iadd__',
'__imul__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__reversed__',
'__rmul__',
'__setattr__',
'__setitem__',
'__sizeof__',
'__str__',
'__subclasshook__',
'append',
'clear',
'copy',
'count',
'extend',
'index',
'insert',
'pop',
'remove',
'reverse',
'sort']
```

```
[13]: l1= [1,2]
      l2 = [1,2]
      l1 == l2, id(l1) == id(l2)
```

```
[11]: hash(5)
```

```
[11]: 5
```

4 Q4

Clés de même haché et comportement des dictionnaires Python.

Dans cette question seulement, on abandonne la représentation des dictionnaires par la liste `table` et on étudie le comportement de Python pour les clefs de même haché.

On définit une classe `MaCle` qui force le hachage de ses instances et redéfinit l'égalité:

```
class MaCle:
    def __init__(self, valeur):
        self.valeur = valeur
    def __hash__(self):#force le hache des instances
        return 42 # hash constant -> collisions forcées
    def __eq__(self, autre):
        return self.valeur == autre.valeur
```

De la sorte `MaCle("A")` et `MaCle("B")` ont le même hache. L'égalité est redéfinie.

```
[18]: MaCle("A")
```

```
[18]: A
```

```
[8]: MaCle("A")==MaCle("A"), MaCle("A") is MaCle("A")
```

```
[8]: (True, False)
```

```
[9]: hash(MaCle("A")),hash(MaCle("B"))
```

```
[9]: (42, 42)
```

Pourtant, vérifier que Python ne perd pas d'informations avec

```
d = {}
d[MaCle("A")] = "Alice"
d[MaCle("B")] = "Bob"
d[MaCle("C")] = "Charlie"
```

```
print(d)
```

On attend que toutes les clefs soient présentes dans le dictionnaire Python.

5 Q5

Gestion des collisions par adressage ouvert (et sondage linéaire).

Dans cet exercice, les dictionnaires sont encore implantés par une table de taille fixe et les alvéoles contiennent les tuples (clés,valeurs). On ne redimensionne pas le tableau.

```
TAILLE = 11
table = [None] * TAILLE # chaque case: None ou (cle, valeur)
```

1. Ecrire la fonction `_trouver_slot(cle,table)` qui renvoie l'indice de la clé si elle est présente et sinon `None` dans un contexte de sondage linéaire : si le haché de la clé n'est pas la position d'un tuple de même clé on essaye toutes les positions suivantes jusqu'à trouver la clé ou `None`.
2. Ecrire la fonction `_premier_trou(table,cle)` qui renvoie `None` si la clé est présente. Sinon la fonction renvoie la position du premier trou après le haché de la clé dans un contexte de sondage linéaire : si le haché de la clé n'est pas la position d'un tuple de même clé on essaye toutes les positions suivantes jusqu'à trouver la clé (on renvoie alors `None`) ou bien `None` (et on renvoie la position de cet objet).
3. Ecrire la fonction `insérer(table,cle,valeur)` qui insère le tuple (clé,valeur) dans le dictionnaire. Si la clé est présente, on remplace l'ancienne valeur associée par la nouvelle, sinon on insère le tuple à la première place vide après le haché. Si la table est pleine, une exception est soulevée.
4. Ecrire la fonction `chercher(table,cle)` qui renvoie la valeur associée à la clef si la clef est présente. Dans le cas contraire une exception `KeyError(cle)` (définie par défaut en Python) est soulevée

On donne:

```
def afficher():
    for i, slot in enumerate(table):
        print(f"{i:02d}: {'.' if slot is None else f'{slot[0]} → {slot[1]}'}")
```

On attend le rendu suivant :

```
[12]: TAILLE = 11
table = [None] * TAILLE # chaque case: None ou (cle, valeur)

# Choisissons des clés qui collisionnent modulo 11 (ex: 0, 11, 22, 33...)

for k, v in [(0, "A"), (11, "B"), (22, "C"), (33, "D")]:
    insérer(table,k, v)

afficher(table)
print("chercher(22) =", chercher(table,22))

# remplacement de valeur sur une clé existante
insérer(table,11, "B'")

print(chercher(table,33))
```

```
afficher(table)
```

```
00: 0 → A
01: 11 → B
02: 22 → C
03: 33 → D
04: .
05: .
06: .
07: .
08: .
09: .
10: .
chercher(22) = C
D
00: 0 → A
01: 11 → B'
02: 22 → C
03: 33 → D
04: .
05: .
06: .
07: .
08: .
09: .
10: .
```

6 Q6 Redimensionnement

Un dictionnaire `d` est maintenant une liste de longueur 2.

Le premier membre de la liste désigne le nombre d'associations (clé,valeur) du dictionnaire.

Le second, que nous appelons *tableau sous jacent*, est une liste qui a le même comportement qu'à la question précédente.

1. Ecrire une fonction `init()` qui renvoie un dictionnaire vide dont le tableau sous-jacent est de longueur 8.
2. Écrire la fonction `insérer2(dico,cle,valeur)` qui insere une nouvelle association et redimensionne le tableau sous-jacent sinon.
Règle : si le facteur de facteur de compression est supérieur à 2/3, il faut redimensionner, c.a.d. prendre un tableau deux fois plus grand et y insérer les associations déjà présentes.

```
[14]: init()
```

```
[14]: [0, [None, None, None, None, None, None, None, None]]
```

```
[16]: d = init()
      print(d)
```

```
# pas de redimensionnement puisque 5/8 < 2/3
for k, v in [(0, "A"), (11, "B"), (22, "C"), (33, "D"), (5, "E")]:
    inserer2(d, k, v)
print(f"nombre d'associations :{d[0]}")
afficher(d[1])
```

```
[0, [None, None, None, None, None, None, None, None]]
nombre d'associations :5
00: 0 → A
01: 33 → D
02: .
03: 11 → B
04: .
05: 5 → E
06: 22 → C
07: .
```

```
[17]: # redimensionnement car 6/8 > 2/3
for k, v in [(7, "F"), (23, "G")]:
    inserer2(d, k, v)
print(f"nombre d'associations :{d[0]}")
afficher(d[1])
```

```
nombre d'associations :7
00: 0 → A
01: 33 → D
02: .
03: .
04: .
05: 5 → E
06: 22 → C
07: 7 → F
08: 23 → G
09: .
10: .
11: 11 → B
12: .
13: .
14: .
15: .
```

```
[18]: 6/8 > 2/3, 5/8 > 2/3
```

```
[18]: (True, False)
```

```
[ ]:
```