TP : Tables de hachage et gestion des collisions

Dans ce TP, nous étudions plusieurs façons de gérer les collisions dans une table de hachage. Sauf dans la partie consacrée aux dictionnaires Python, les clés sont des entiers. Les fonctions d'insertion modifient la table reçue en argument et ne renvoient rien.

1 Insertion sans gestion des collisions

Le dictionnaire est représenté par une liste appelée *tableau sous-jacent*. Chaque case contient soit `None`, soit un couple `(cle, valeur)`.

```
TAILLE = 5
table = [None] * TAILLE
```

Question 1. Écrire la fonction `hachage(cle, taille)` qui renvoie le reste de la division de la clé par la taille, supposée strictement positive.

Remarque. Dans ce TP, les clés étant des entiers, nous calculons directement le reste de leur division par la taille du tableau pour obtenir un indice. Plus généralement, on utilise d'abord une fonction de hachage, comme `hash` en Python, qui associe un entier à chaque clé : cet entier est appelé son *haché*. On ramène ensuite ce haché à un indice du tableau en calculant le reste de sa division par la taille du tableau :

```
indice = hash(cle) % len(table)
```

Deux clés différentes peuvent ainsi conduire au même indice : c'est une *collision*.

Question 2. Écrire la fonction `inserer(table, cle, valeur)` qui place le couple dans la case indiquée par la fonction de hachage. La taille à utiliser est celle de la table reçue en argument. Pour le moment, une insertion écrase le contenu précédent de cette case. Exécuter le code suivant et expliquer le résultat.

```
inserer(table, 12, "Alice")
inserer(table, 7, "Bob")
print(table)
```

Résultat attendu :

```
[None, None, (7, 'Bob'), None, None]
```

Question 3. Modifier l'insertion pour afficher un message lorsque la case contient déjà une association dont la clé est différente. L'insertion continue d'écraser cette association. Une mise à jour de la valeur associée à une même clé n'est pas une collision.

```
table = [None] * 5
inserer(table, 12, "Alice")
inserer(table, 17, "Bob")
inserer(table, 22, "Charlie")
print(table)
```

Exemple d'affichage attendu :

```
Collision entre 17 et 12
Collision entre 22 et 17
[None, None, (22, 'Charlie'), None, None]
```

2 Gestion des collisions par chaînage

Chaque alvéole est maintenant une liste de couples. Plusieurs clés peuvent donc être rangées dans la même alvéole.

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

Question 4. Modifier `inserer(table, cle, valeur)` pour utiliser cette représentation. Si la clé est déjà présente, remplacer sa valeur ; sinon, ajouter le couple à la fin de l'alvéole correspondante.

```
1 inserer(table, 12, "Alice")
2 inserer(table, 17, "Bob")
3 inserer(table, 22, "Charlie")
4 print(table)
```

Résultat attendu :

```
[[], [], [(12, 'Alice'), (17, 'Bob'), (22, 'Charlie')], [], []]
```

3 Collisions dans les dictionnaires Python

Dans cette partie seulement, nous utilisons les dictionnaires de Python. On donne la classe suivante ; il n'est pas demandé de savoir écrire une classe. La méthode `__hash__` impose un même haché à toutes les instances. La méthode `__eq__` définit leur égalité en comparant leurs attributs `valeur`.

```
1 class MaCle:
2     def __init__(self, valeur):
3         self.valeur = valeur
4
5     def __hash__(self):
6         return 42
7
8     def __eq__(self, autre):
9         return self.valeur == autre.valeur
```

Ainsi, deux instances construites avec la même valeur sont égales, même s'il s'agit de deux objets distincts. Deux instances construites avec des valeurs différentes ont le même haché mais ne sont pas égales.

Question 5. Exécuter le code suivant. Vérifier que les trois associations sont conservées, puis que la dernière affectation modifie seulement la valeur associée à la clé construite avec `"A"`. Que montre cet exemple sur la façon dont Python distingue les clés ?

```
1 d = {}
2 d[MaCle("A")] = "Alice"
3 d[MaCle("B")] = "Bob"
4 d[MaCle("C")] = "Charlie"
5 print(len(d))
6 print(d[MaCle("A")], d[MaCle("B")], d[MaCle("C")])
7
8 d[MaCle("A")] = "Alice au pays des merveilles"
9 print(len(d))
10 print(d[MaCle("A")], d[MaCle("B")], d[MaCle("C")])
```

Résultat attendu :

```
3
Alice Bob Charlie
3
Alice au pays des merveilles Bob Charlie
```

4 Adressage ouvert et sondage linéaire

Nous revenons à une table de taille fixe, sans redimensionnement. Chaque case contient `None` ou un couple.

```
TAILLE = 11
table = [None] * TAILLE
```

En cas de collision, on examine les cases suivantes, en revenant à la case zéro après la dernière. On examine au plus toutes les cases de la table. Aucune suppression n'est effectuée dans ce TP.

Question 6. Écrire `_trouver_slot(table, cle)` qui renvoie l'indice de la clé si elle est présente, et `None` sinon. La recherche commence à l'indice donné par le hachage et s'arrête lorsqu'on trouve la clé, une case vide ou lorsqu'on a examiné toutes les cases.

Question 7. Écrire `_trouver_trou(table, cle)` qui renvoie `None` si la clé est déjà présente. Sinon, elle renvoie l'indice de la première case vide rencontrée par sondage linéaire. Si la clé est absente et la table pleine, elle renvoie également `None`.

Question 8. Écrire `inserer(table, cle, valeur)`. Si la clé est présente, remplacer sa valeur. Sinon, insérer le couple à la première place vide rencontrée par sondage linéaire. Si la clé est absente et la table pleine, soulever une exception `ValueError("Table pleine")`.

Question 9. Écrire `chercher(table, cle)` qui renvoie la valeur associée à la clé. Si celle-ci est absente, soulever l'exception `KeyError(cle)`.

On donne la fonction d'affichage suivante :

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

Tester les fonctions avec :

```
table = [None] * 11
for k, v in [(0, "A"), (11, "B"), (22, "C"), (33, "D")]:
    inserer(table, k, v)
afficher(table)
print("chercher(22) =", chercher(table, 22))
inserer(table, 11, "B'")
print(chercher(table, 33))
afficher(table)
```

Résultat attendu :

```
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: .
```

5 Redimensionnement

Un dictionnaire est maintenant une liste de longueur deux : `dico[0]` est le nombre d'associations et `dico[1]` est le tableau sous-jacent, organisé par adressage ouvert comme dans la partie précédente. Le *facteur de charge* est le rapport du nombre d'associations à la longueur du tableau sous-jacent.

Question 10. Écrire `init()` qui renvoie un dictionnaire vide dont le tableau sous-jacent est de longueur huit.

```
print(init())
```

Résultat attendu :

```
[0, [None, None, None, None, None, None, None, None]]
```

Question 11. Écrire `inserer2(dico, cle, valeur)`. Une mise à jour ne change pas le nombre d'associations. Avant d'ajouter une nouvelle clé, si le facteur de charge qui résulterait de l'ajout dépasse 2/3, doubler la taille du tableau sous-jacent et réinsérer toutes les associations dans la nouvelle table.

Question 12. Écrire `chercher2(dico, cle)` qui renvoie la valeur associée à la clé ou soulève `KeyError(cle)` si elle est absente.

Tester le redimensionnement avec :

```
d = init()
for k, v in [(0, "A"), (11, "B"), (22, "C"), (33, "D"), (5, "E")]:
    inserer2(d, k, v)
print(d[0], len(d[1]))
afficher(d[1])

inserer2(d, 7, "F")
print(d[0], len(d[1]))
inserer2(d, 23, "G")
print(d[0], len(d[1]))
afficher(d[1])

inserer2(d, 11, "B'")
print(d[0], len(d[1]))
print(chercher2(d, 11))
```

Résultat attendu :

```
5 8
00: 0 -> A
01: 33 -> D
02: .
03: 11 -> B
04: .
05: 5 -> E
06: 22 -> C
07: .
6 16
7 16
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: .
7 16
B'
```