

# Rapport de TP : Développement d'un module de table de hachage en Python

Wolodia ZDETOVETZKY

03/2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Présentation du sujet</b>	<b>3</b>
<b>3</b>	<b>Les tables de hachage</b>	<b>4</b>
<b>4</b>	<b>Implémentation Python</b>	<b>5</b>
<b>5</b>	<b>Tests de performances</b>	<b>6</b>
5.1	Prévisions . . . . .	6
5.2	Résultats . . . . .	6
5.3	Analyse . . . . .	6
<b>6</b>	<b>Complexité algorithmique</b>	<b>6</b>
6.1	Définition . . . . .	6
6.2	Complexité de l'implémentation . . . . .	6
<b>7</b>	<b>Conclusion</b>	<b>6</b>
7.1	Synthèse . . . . .	6
7.2	Perspectives . . . . .	6
7.3	limites . . . . .	6
<b>8</b>	<b>Annexes</b>	<b>6</b>
<b>9</b>	<b>Code source</b>	<b>6</b>

# 1 Introduction

L'utilisation de structures de données efficaces est un élément clé dans le développement d'applications performantes en informatique. Parmi les nombreuses structures de données existantes, les tables de hachage sont largement utilisées en raison de leur efficacité en termes de temps d'accès aux données. Dans cette étude, nous nous concentrerons sur le développement d'un module de table de hachage en Python. Les tables de hachage sont une structure de données qui permet de stocker et d'accéder rapidement à des valeurs à partir d'une clé. L'objectif principal des tables de hachage est de fournir une fonction de hachage efficace qui permet de calculer un index de tableau à partir d'une clé. Le processus de hachage consiste à appliquer une fonction de hachage à une clé pour calculer un index, qui est ensuite utilisé pour accéder à la valeur correspondante.

Les tables de hachage sont particulièrement adaptées pour stocker des données de grande taille et sont souvent utilisées dans les bases de données, les systèmes de gestion de fichiers et les applications de recherche. Les tables de hachage ont également l'avantage de pouvoir être utilisées pour résoudre des problèmes de recherche et de filtrage en temps constant.

Cependant, les tables de hachage présentent également certaines limites. Tout d'abord, les collisions peuvent se produire lorsqu'il y a deux clés différentes qui se retrouvent avec le même index. Cela peut ralentir les performances de la table de hachage. De plus, les tables de hachage ont une taille fixe, ce qui signifie qu'elles ne peuvent pas être agrandies une fois qu'elles sont créées.

Dans ce rapport, nous présentons le développement d'un module de table de hachage en Python qui offre un type abstrait `Table`. Le module se base sur une implémentation de table de hachage avec des collisions résolues par adressage fermé puis ouvert. La taille de la table est fixe et l'utilisateur peut choisir entre plusieurs fonctions de hachage et méthodes de rehachage. Le module offre également des méthodes d'insertion, de suppression, de recherche, d'union et d'intersection de tables ainsi qu'une méthode d'affichage.

Nous présentons aussi une analyse des performances de notre module de table de hachage en fonction de la taille de la table, du nombre d'insertions et de recherche d'une clé. Nous comparons les performances de notre module avec d'autres structures de données et implémentations.

Dans les sections suivantes, nous détaillons l'implémentation de notre module de table de hachage en Python, les tests de performances effectués ainsi que l'analyse de la complexité algorithmique de notre implémentation.

## 2 Présentation du sujet

Le sujet de ce rapport est le développement d'un module de table de hachage en Python. L'objectif est de créer un type abstrait `Table` en utilisant une implémentation de table de hachage, où la clé sera un entier et la taille de la table sera fixe.

Le module devra offrir plusieurs méthodes pour la manipulation des tables de hachage :

- `init(taille_table, fonction_hachage, type_rehachage)` - cette méthode initialisera la table de hachage en spécifiant la taille de la table, la fonction de hachage à utiliser et le type de rehachage (linéaire, quadratique ou double hachage).
- `insert(key, value)` - cette méthode insérera une paire clé-valeur dans la table.
- Si la clé existe déjà dans la table, la valeur associée sera mise à jour.
- `delete(key)` - cette méthode supprimera la paire clé-valeur associée à la clé spécifiée de la table.
- `exist(key)` - cette méthode renverra `True` si la clé spécifiée existe dans la table, `False` sinon.
- `value(key)` - cette méthode renverra la valeur associée à la clé spécifiée.
- `union(autre_table)` - cette méthode renverra une table résultant de l'union de la table actuelle avec une autre table spécifiée en argument.
- `intersection(autre_table)` - cette méthode renverra une table résultant de l'intersection de la table actuelle avec une autre table spécifiée en argument.
- `affichage()` - cette méthode affichera les éléments insérés dans la table ainsi que le nombre de rehachage nécessaire pour chaque insertion.

Dans un premier temps, la taille de la table sera fixe et les collisions seront gérées par adressage fermé, puis par adressage ouvert si nécessaire. La fonction de hachage sera donnée par l'utilisateur et le type de rehachage sera linéaire, quadratique ou double hachage.

Le module de table de hachage sera testé pour mesurer ses performances en termes de temps de recherche d'une clé en fonction du nombre d'éléments dans la table. Les tests seront effectués pour le rehachage linéaire et les résultats obtenus seront comparés à d'autres structures de données et implémentations existantes.

### 3 Les tables de hachage

Les tables de hachage sont des structures de données qui permettent de stocker des informations en associant une clé unique à chaque élément. Elles sont particulièrement utiles pour les opérations de recherche, d'insertion et de suppression dans une grande collection de données.

Le principe de base des tables de hachage consiste à stocker les éléments dans un tableau en utilisant une fonction de hachage qui calcule l'indice d'un élément dans le tableau en fonction de sa clé. La fonction de hachage doit être déterministe, c'est-à-dire qu'elle doit toujours renvoyer le même indice pour une même clé.

Les tables de hachage peuvent être implémentées de différentes manières, mais toutes reposent sur le même principe de base. L'efficacité de la table dépend de la qualité de la fonction de hachage choisie, ainsi que de la gestion des collisions (c'est-à-dire la gestion des cas où deux clés différentes aboutissent au même seau). Les techniques courantes incluent l'adressage ouvert, où les éléments sont stockés dans des emplacements vides du tableau de hachage, et l'adressage fermé, où les éléments sont stockés dans des emplacements différents du tableau de hachage.

En Python, les tables de hachage sont implémentées dans le dictionnaire, qui est une structure de données native, permettant de stocker des clés et des valeurs. Les dictionnaires Python utilisent une table de hachage interne pour stocker les éléments. Cela signifie que les opérations de recherche et d'insertion sont très rapides, quelle que soit la taille du dictionnaire.

Cependant, il peut être pertinent de développer notre propre module de table de hachage en Python pour plusieurs raisons. Tout d'abord, nous pouvons personnaliser notre implémentation pour répondre à des besoins spécifiques, dans notre cas en proposant des fonctionnalités spécifiques telles que l'union, l'intersection et l'affichage de tables de hachage. De plus, nous pouvons mieux comprendre le fonctionnement interne des tables de hachage en développant notre propre module. Enfin, en développant notre propre module de table de hachage, nous avons un plus grand contrôle sur les performances de notre application.

## 4 Implémentation Python

La classe `Table` a plusieurs méthodes pour effectuer des opérations sur la table, notamment `insert`, `delete`, `exist`, `value`, `union`, `intersection` et `affichage`.

La méthode `__init__` initialise la table en prenant en entrée la taille de la table, la fonction de hachage à utiliser et le type de rehachage à utiliser. La table elle-même est stockée dans une liste Python et initialisée avec la valeur `None`. Le nombre de rehash est également initialisé à 0.

La méthode `_rehash` est utilisée pour résoudre les collisions. Elle prend en entrée une clé et un nombre d'itérations et renvoie la clé rehashée correspondante en utilisant la méthode de rehashing spécifiée dans le constructeur.

La méthode `insert` insère une paire clé-valeur dans la table. Si la case correspondante est vide, la paire est insérée directement. Si la case correspondante contient déjà la même clé, la valeur est mise à jour. Si la case correspondante contient une autre clé, la méthode de rehashing est utilisée pour trouver une nouvelle case vide. La méthode renvoie `True` si l'insertion a réussi et `False` sinon.

La méthode `delete` supprime une paire clé-valeur de la table. Si la case correspondante est vide, rien n'est fait. Si la case correspondante contient la même clé, la paire est supprimée (la case est remplacée par `None`). Si la case correspondante contient une autre clé, la méthode de rehashing est utilisée pour trouver la case contenant la clé à supprimer.

La méthode `exist` vérifie si une clé existe dans la table. Si la case correspondante est vide, la clé n'existe pas. Si la case correspondante contient la même clé, la clé existe. Si la case correspondante contient une autre clé, la méthode de rehashing est utilisée pour trouver la case contenant la clé cherchée. La méthode renvoie `True` si la clé existe et `False` sinon.

La méthode `value` renvoie la valeur associée à une clé. Si la case correspondante est vide, `None` est renvoyé. Si la case correspondante contient la même clé, la valeur est renvoyée. Si la case correspondante contient une autre clé, la méthode de rehashing est utilisée pour trouver la case contenant la clé cherchée.

Les méthodes `union` et `intersection` renvoient de nouvelles tables qui sont l'union ou l'intersection de la table actuelle et d'une autre table spécifiée en entrée. Les nouvelles tables ont la même taille et la même fonction de hachage que la table actuelle. Les éléments des deux tables sont ajoutés à la nouvelle table selon les règles de l'union ou de l'intersection.

La méthode `affichage` affiche la table de hachage sous forme de tableau.

Le code source de la classe `Table` est disponible dans l'annexe ??.

## 5 Tests de performances

### 5.1 Prévisions

### 5.2 Résultats

### 5.3 Analyse

## 6 Complexité algorithmique

### 6.1 Définition

### 6.2 Complexité de l'implémentation

## 7 Conclusion

### 7.1 Synthèse

### 7.2 Perspectives

### 7.3 limites

## 8 Annexes

### 8.1 Code source

Ci-dessous est présenté le code source de la classe `Table`.

```
import time
import random
from tqdm import tqdm
import matplotlib.pyplot as plt

class Table:
    def __init__(self, table_size, hash_func, rehash_type):
        """
        Constructor for Table class

        :param table_size: size of the table
        :param hash_func: hash function to use
        :param rehash_type: rehash type to use
        """
        self.table_size = table_size
        self.table = [None] * self.table_size
        self.hash_func = hash_func
        self.rehash_type = rehash_type
```

```

        self.rehash_count = 0

    def _rehash(self, key, i):
        """
        Rehash function to use for collision resolution

        :param key: key to rehash
        :param i: iteration number
        :return: rehashed key
        """
        if self.rehash_type == 'linear':
            return (key + i) % self.table_size
        elif self.rehash_type == 'quadratic':
            return (key + i * i) % self.table_size
        elif self.rehash_type == 'double':
            hash1 = key % self.table_size
            hash2 = 1 + (key % (self.table_size - 2))
            return (hash1 + i * hash2) % self.table_size

    def insert(self, key, value):
        """
        Inserts a key-value pair into the table

        :param key: key to insert
        :param value: value to insert
        :return: True if successful, False otherwise
        """
        i = 0
        while i < self.table_size:
            hashed_key = self.hash_func(key)
            rehashed_key = self._rehash(hashed_key, i)
            if self.table[rehashed_key] is None:
                self.table[rehashed_key] = (key, value)
                return True
            elif self.table[rehashed_key][0] == key:
                self.table[rehashed_key] = (key, value)
                return True
            else:
                i += 1
                self.rehash_count += 1
        return False

    def delete(self, key):
        """
        Deletes a key-value pair from the table

```

```

        :param key: key to delete
        """
        i = 0
        while i < self.table_size:
            hashed_key = self.hash_func(key)
            rehashed_key = self._rehash(hashed_key, i)
            if self.table[rehashed_key] is None:
                return
            elif self.table[rehashed_key][0] == key:
                self.table[rehashed_key] = None
                return
            else:
                i += 1
        return

def exist(self, key):
    """
    Checks if a key exists in the table

    :param key: key to check
    :return: True if key exists, False otherwise
    """
    i = 0
    while i < self.table_size:
        hashed_key = self.hash_func(key)
        rehashed_key = self._rehash(hashed_key, i)
        if self.table[rehashed_key] is None:
            return False
        elif self.table[rehashed_key][0] == key:
            return True
        else:
            i += 1
    return False

def value(self, key):
    """
    Returns the value associated with a key

    :param key: key to check
    :return: value associated with key
    """
    i = 0
    while i < self.table_size:
        hashed_key = self.hash_func(key)
        rehashed_key = self._rehash(hashed_key, i)
        if self.table[rehashed_key] is None:

```



```

        return None
    elif self.table[rehashed_key][0] == key:
        return self.table[rehashed_key][1]
    else:
        i += 1
    return None

def union(self, other_table):
    """
    Returns a new table that is the union of the current table
    and the other table

    :param other_table: other table to union with
    :return: new table that is the union of the current table
    and the other table
    """
    # Create a new table with the same size and hash function
    # as the current table
    union_table = Table(self.table_size, self.hash_func, self.rehash_type)

    # Add all the elements from the current table to the new table
    for i in range(self.table_size):
        if self.table[i] is not None:
            union_table.insert(self.table[i][0], self.table[i][1])

    # Add all the elements from the other table to the new table
    for i in range(other_table.table_size):
        if other_table.table[i] is not None:
            union_table.insert(
                other_table.table[i][0],
                other_table.table[i][1]
            )

    return union_table

def intersection(self, other_table):
    """
    Returns a new table that is the intersection
    of the current table and the other table

    :param other_table: other table to intersect with
    :return: new table that is the intersection
    of the current table and the other table
    """
    # Create a new table with the same size
    # and hash function as the current table

```

```

        intersection_table = Table(
            self.table_size,
            self.hash_func,
            self.rehash_type
        )

        # Add elements to the new table only if they exist in both tables
        for i in range(self.table_size):
            if self.table[i] is not None \
               and other_table.exist(self.table[i][0]):
                intersection_table.insert(
                    self.table[i][0],
                    self.table[i][1]
                )

        return intersection_table

def display(self):
    """
    Displays the table
    """
    for i in range(self.table_size):
        print(f"{i}: {self.table[i]}")

def test_table(size=5, hash_func=lambda x: x % 5, rehash_type='linear'):
    """
    Tests the Table class

    :param size: size of table
    :param hash_func: hash function to use
    :param rehash_type: rehash function to use
    """

    # create a new table object
    table = Table(size, hash_func, rehash_type)

    # test insert method
    assert table.insert(10, "value1") is True
    assert table.insert(5, "value2") is True
    assert table.insert(20, "value3") is True

    # test exist method
    assert table.exist(10) is True
    assert table.exist(5) is True
    assert table.exist(20) is True
    assert table.exist(15) is False

```

```

# test value method
assert table.value(10) is "value1"
assert table.value(5) is "value2"
assert table.value(20) is "value3"
assert table.value(15) is None

# test delete method
table.delete(10)
assert table.exist(10) is False
assert table.value(10) is None

# create another table object
other_table = Table(size, hash_func, rehash_type)
other_table.insert(5, "other_value1")
other_table.insert(15, "other_value2")
other_table.insert(25, "other_value3")

# test union method
union_table = table.union(other_table)
assert union_table.exist(5) is True
assert union_table.exist(15) is True
assert union_table.exist(20) is True
assert union_table.exist(25) is True

# test intersection method
intersection_table = table.intersection(other_table)
assert intersection_table.exist(5) is True
assert intersection_table.exist(15) is False
assert intersection_table.exist(20) is False
assert intersection_table.exist(25) is False

# test display method
table.display()

test_table(5, lambda x: x % 5, 'linear')

result = {
    'linear': {},
    'quadratic': {},
    'double': {}
}

def test_rehashing(n, rehash_type='linear', table_size=100):
    """
    test the Table class with rehashing

```



```

linear_data = []
quadratic_data = []
double_data = []

for method in ['linear', 'quadratic', 'double']:
    data = []
    for key in range(1, 100000, 1000):
        data.append([
            key, result[method][key]['number_of_elements'],
            result[method][key]['search_time'],
            result[method][key]['insert_time'],
            result[method][key]['rehash_count']
        ])
    if method == 'linear':
        linear_data = data
    elif method == 'quadratic':
        quadratic_data = data
    elif method == 'double':
        double_data = data

plt.plot(
    [x[1] for x in linear_data],
    [x[2] for x in linear_data],
    label='linear')
plt.plot(
    [x[1] for x in quadratic_data],
    [x[2] for x in quadratic_data],
    label='quadratic')
plt.plot(
    [x[1] for x in double_data],
    [x[2] for x in double_data],
    label='double')

plt.xlabel('Number_of_insertions')
plt.ylabel('Total_time_(s)')
plt.title(f'Search_time_performance_by_rehashing_method_\
~~~~~for_table_size_{table_size}')
plt.legend()
plt.show()

plt.plot(
    [x[1] for x in linear_data],
    [x[3] for x in linear_data],
    label='linear')
plt.plot(
    [x[1] for x in quadratic_data],

```

```

        [x[3] for x in quadratic_data],
        label='quadratic')
plt.plot(
    [x[1] for x in double_data],
    [x[3] for x in double_data],
    label='double')

plt.xlabel('Number_of_insertions')
plt.ylabel('Total_time_(s)')
plt.title(f'Insert_time_performance_by_rehashing_method_\
~~~~~for_table_size_{table_size}')
plt.legend()
plt.show()

plt.plot(
    [x[1] for x in linear_data],
    [x[4] for x in linear_data],
    label='linear')
plt.plot(
    [x[1] for x in quadratic_data],
    [x[4] for x in quadratic_data],
    label='quadratic')
plt.plot(
    [x[1] for x in double_data],
    [x[4] for x in double_data],
    label='double')

plt.xlabel('Number_of_insertions')
plt.ylabel('Total_rehash_count')
plt.title(f'Rehash_count_performance_by_rehashing_method_\
~~~~~for_table_size_{table_size}')
plt.legend()
plt.show()

```