INFO-F103 — Algorithmique 1
Projet 3 – Implémentation et analyse des tables de hachage

Robin Petit

Année académique 2019 - 2020

1 Introduction

Comme vous avez vu au cours théorique, une table de hachage est une structure de données très intéressante pour associer des clefs qui ne sont pas entières (ou des valeurs entières non successives) à des valeurs quelconques. Cependant, l'efficacité de cette structure dépend grandement de la fonction de hachage implémentée et de la méthode de gestion des collisions.

L'objectif de ce troisième projet est de vous faire travailler sur cette structure de données, et de vous amener à vous rendre compte de l'impact des différents paramètres sur l'efficacité de la structure.

2 Énoncé

Ce projet est séparé en deux parties : la première est un ensemble de fichiers comprenant le code d'une table de hachage et des différentes composantes nécessaires (ce qui est détaillé juste après) ; et la seconde est un rapport de 5 à 7 pages (page de garde non comprise) détaillant vos choix d'implémentations, et montrant une comparaison entre les différentes fonctions demandées, ainsi qu'une comparaison entre les résultats théoriques vus au cours (chapitre 12) et les valeurs obtenues empiriquement avec votre programme.

En terme de code, il vous est demandé ceci :

1. Dans un fichier `hashers.py`, écrivez les classes `HasherDjb2`, `HasherCrc32` et `HasherKR` qui implémentent toutes trois une méthode `hash_key(self, string)` qui prend une clef (de type `str`) et qui renvoie un entier (donc de type `int`) et qui implémentent respectivement l'algorithme `djb2` (de Daniel J. Bernstein) et l'algorithme `CRC32` (Cyclic Redundancy Check-32) et l'algorithme de Kernighan & Ritchie (voire annexe pour le pseudo-code des algorithmes). Dans ce même fichier, implémentez une classe `DoubleHasher` dont le constructeur prend en paramètre deux instances de hasher définis juste avant, et implémentant une méthode `hash_key(self, k, i)` effectuant un double hachage.
2. dans un fichier `hashtable.py`, écrivez une classe `HashTableChaining` représentant une table de hachage gérant les collisions par chaînage et une classe `HashTableDouble` représentant une table de hachage gérant les collisions par double hachage. Le constructeur de `HashTableChaining` doit prendre en paramètre `m` (la taille du conteneur), et une instance d'un des hashers définis au point 1 ; et le constructeur de `HashTableDouble` doit prendre en paramètre `m` et un instance de `DoubleHasher` défini au point 1. Ces classes doivent contenir les méthodes suivantes :
   — `insert(self, key, value)` : ajoute la valeur `value` liée à la clef `key`. Si aucun emplacement libre n'a été trouvé, la méthode doit lancer une exception de type `OverflowError`. [1]
   — `delete(self, key)` : supprime l'entrée associée à la clef `key` de la table de hachage si l'entrée existe et lance une exception de type `KeyError` si l'entrée n'existe pas.

1. En Python, cette exception est destinée à être lancée quand un overflow est rencontré lors d'une opération arithmétique, mais nous considérerons ici qu'elle est suffisamment explicite pour que son objectif initial soit légèrement dépassé.

— `get(self, key)` : renvoie la valeur associée à la clef `key` si l'entrée existe et lance une exception de type `KeyError` si l'entrée n'existe pas.
— `load_factor(self)` : renvoie le *load factor* $\alpha \in [0, 1]$.
— `size(self)` : renvoie le nombre d'entrées présentes dans le conteneur.

3. Dans un fichier `plots.py`, mettez tout le code permettant de faire les graphiques que vous mettrez dans votre rapport. Tous les graphiques doivent être générés avec le package `matplotlib` (que vous pouvez installer simplement avec `pip install matplotlib` si ce n'est pas déjà fait).
4. Dans un fichier `timer.py`, mettez tout le code permettant de chronométrer les différentes étapes de votre programme. Pour cela, vous devez utiliser le package `timeit`. Ce dernier est standard, i.e. il est déjà installé avec votre distribution de Python3.

Avec ce code, écrivez un rapport en LaTeXen suivant le template mis sur l'UV dans lequel vous devez expliquer vos choix d'implémentation et analyser les performances de vos classes.

Plus précisément, votre rapport doit contenir (entre autres) les points suivants :

1. une discussion sur l'hypothèse de hachage uniforme pour les différentes fonctions implémentées ;
2. des statistiques sur le temps nécessaire pour l'exécution des méthodes `insert` et `delete` pour différentes valeurs de α ainsi que des représentations graphiques ;
3. une comparaison de ces valeurs ainsi que du nombre moyen de sondages avant de trouver une cellule libre avec les résultats théoriques vus au cours théorique ;
4. une comparaison du nombre de collisions en fonction de la fonction de hachage utilisée ;
5. un pseudo-code des fonctions de hachage supplémentaires que vous auriez implémentées ;
6. les difficultés que vous avez rencontrées lors de la réalisation de ce projet.

Remarque

Si le cœur vous en dit, nous ne pouvons que vous inciter à creuser votre compréhension et à étoffer vos comparaisons en implémentant - par exemple - plus de fonctions de hachage que celles demandées explicitement. Notez cependant que toute fonctionnalité non-demandée doit également être correcte au risque d'en être pénalisé.

Si après tout cela vous en voulez encore, il vous est proposé d'implémenter une taille dynamique à vos tables de hachage pour un bonus maximum de 2 points sur 20 : dans ce cas, un overflow ne sera pas géré en lançant une exception mais bien en adaptant la taille du conteneur. La taille peut également être augmentée quand α devient trop grand. Si vous décidez d'implémenter cette fonctionnalité, faites une (ou plusieurs) nouvelle classe(s) dans le fichier `hashtable.py`, et bien entendu, discutez les résultats ainsi obtenus dans votre rapport.

3 Consignes de remise

1. Une FAQ pour ce projet se trouve sur l'UV, allez la vérifier régulièrement (et lisez donc bien vos mails) car tout ajout dans cette FAQ correspond à un ajout aux consignes ;
2. un fichier `test.py` vous est fourni avec cet énoncé, il contient 26 tests unitaires que votre code **doit** passer (vous pouvez lancer ces tests avec la commande suivante : `pytest test.py`, où `pytest` est un programme spécialisé pour les tests unitaires en Python, que vous pouvez installer à l'aide de `pip install pytest`) ;
3. le projet doit être remis sur l'UV pour le dimanche 10 mai à 23 : 59 : 59 au plus tard ;
4. vous devez remettre une unique archive appelée `projet3.zip` contenant un dossier `src/` contenant les fichiers sources demandés ci-dessus et un dossier `rapport/` contenant le(s) fichier(s) `.tex` de votre rapport ainsi que votre rapport au format PDF appelé `rapport.pdf` ;

5. le projet doit être codé en Python 3, et seuls les packages `matplotlib` et `timeit` mentionnées ci-dessus vous sont autorisés ;
6. un projet remis après la deadline **stricte** sera considéré comme non-remis et ne sera donc pas corrigé ;
7. respectez **scrupuleusement** les consignes (en particulier les noms de fichiers lors de la remise) car tout manquement à ces consignes sera soldé en une note nulle ;
8. **attention** : ceci est un cours d'algorithmique, donc veillez bien à rendre votre code le plus efficace possible ;
9. pour toute question concernant ce projet, envoyez-moi un mail à l'adresse `robpetit@ulb.ac.be` (veuillez systématiquement commencer le sujet de votre mail par la mnémonique du cours, i.e. `INFO-F103`)

Annexe

Un pseudo-code décrivant le fonctionnement des trois fonctions de hachage mentionnées ci-dessus vous est fourni dans cette annexe. **Attention** : bien que cela ne soit pas explicité dans les algorithmes, ces fonctions renvoient toutes un entier encodé sur 32 bits. Tout overflow dans la valeur du hash pendant le calcul est tout simplement non pris en compte.

3.1 Algorithme K & R

Pseudo-code 1 Algorithme de Kernighan & Ritchie

```
1: procedure KR(string s)
2:     hash ← 0
3:     for all char c in s do
4:         hash ← hash + c
5:     end for
6:     return hash
7: end procedure
```

3.2 Algorithme djb2

Pseudo-code 2 Algorithme djb2 de Daniel J. Bernstein

```
1: procedure DJB2(string s)
2:     hash ← 0x1505
3:     for all char c in s do
4:         hash ← 33× hash + c
5:     end for
6:     return hash
7: end procedure
```

3.3 Algorithme CRC32

L'algorithme CRC-32 est habituellement implémenté à l'aide d'une table de 256 valeurs précalculées qui vont servir à effectuer itérativement un **xor** entre la valeur du hash calculée jusqu'à présent et une entrée de la table. Cette table vous est donnée dans le fichier `crc32.txt` : chaque ligne du fichier correspond à une entrée de la table écrite en hexadécimal (la 1e ligne correspond à `table[0]`, la seconde ligne correspond à `table[1]`, etc. jusqu'à la 256e ligne qui correspond à `table[255]`).

Pseudo-code 3 Algorithme CRC-32

```
procedure CRC32(string s)
    hash ← 0xFFFFFFFF
    for all char c in s do
        idx ← (hash xor c) and 0xFF
        hash ← (hash shr 8) xor table[idx]
    end for
    return hash xor 0xFFFFFFFF
end procedure
```