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

Robin Petit

Année académique 2020 - 2021

1 Introduction

L'objectif de ce troisième projet est de vous familiariser avec les différentes méthodes de résolution des collisions des tables de hachage vues au cours théorique et aux séances d'exercices ainsi qu'à l'analyse scientifique et rigoureuse de vos résultats. Vous allez pour cela devoir implémenter plusieurs algorithmes vus au cours théorique et les comparer dans un rapport.

2 Énoncé

Ce projet est séparé en deux parties :

1. une première partie de code où vous étendrez la classe codée pour le quatrième exercice coté et où vous coderez également une table de hachage résolvant les collisions par chaînage ;
2. et une seconde partie où vous rédigerez un court rapport dans lequel vous comparerez vos différentes implémentations et où vous discuterez la pertinence de vos résultats par rapport aux complexités théoriques vues en cours.

2.1 Code à écrire

En terme de code, il vous est demandé ceci :

1. Dans un fichier `hashtable.py`, écrivez une classe `DictOpenAddressing` définissant les méthodes :
   — `__init__(self, m)` qui initialise l'objet ;
   — `__len__(self)` qui retourne le nombre d'éléments stockés (et non la taille de la table) ;
   — `insert(self, key, value)` qui insère l'élément `value` associé à la clef `key` (et lance une `OverflowError` si l'insertion est impossible) ;
   — `search(self, key)` qui renvoie la valeur associée à la clef `key` si cette dernière existe dans le conteneur (et lance une `KeyError` si la clef n'existe pas dans le conteneur) ;
   — `__setitem__(self, key, value)` qui appelle `insert` ;
   — `__getitem__(self, key)` qui appelle `search`,
   et la propriété `load_factor` qui renvoie le facteur de charge (donc n/m pour n le nombre d'éléments stockés dans le conteneur et m la taille de la table).
2. À cette classe, ajoutez une méthode `delete(self, key)` qui retire l'élément de clef `key` du conteneur s'il existe et lance une `KeyError` sinon (voir ci-dessous pour une approche de suppression) ainsi qu'une méthode `__delitem__(self, key)` qui appelle `delete`.
3. Dans le même fichier, écrivez une classe `DictChainingLinkedList` et une classe `DictChainingSkipList` qui implémentent toutes deux les mêmes méthodes que la classe `DictOpenAddressing`, si ce n'est que l'insertion ne peut pas lancer d'exception car l'insertion y est toujours possible.

La classe `DictChainingLinkedList` devra gérer les collisions à l'aide de listes chaînées comme vu au cours théorique et aux séances d'exercices alors que la classe `DictChainingSkipList` devra gérer les collisions à l'aide de *skip-lists*. Notez bien que l'insertion dans un liste chaînée se fait en début de chaîne, mais pas l'insertion dans une skip-list.

Vous pouvez supposer que toutes les clefs utilisées dans ce projet sont des chaînes de caractères (donc des objets de type `str`), que la classe `DictOpenAddressing` utilise les fonctions de hachage K&R et DJB2 pour h_1 et h_2 respectivement telles que présentées dans l'exercice coté et que les classes `DictChainingLinkedList` et `DictChainingSkipList` utilisent la fonction DJB2 comme fonction de hachage.

Remarque : les clefs sont ici uniques (comme dans un `dict` en `Python`), ce qui implique que lors d'une insertion à une clef `k` donnée dans le conteneur, si cette clef existe déjà, il faut *remplacer* la valeur stockée actuellement et non insérer une nouvelle entrée.

2.1.1 La suppression dans une table de hachage à adressage ouvert

La suppression d'un élément dans une table de hachage à adressage ouvert n'est pas triviale. En effet puisque l'adressage ouvert consiste en un sens à *simuler* le chaînage des éléments dont la clef est hachée vers la même valeur en utilisant des cellules libres de la table, mettre un élément quelconque à `None` revient à casser la liste chaînée en retirant un élément *et tous les suivants*. Il est donc important de maintenir la structure afin de pouvoir tout de même faire une recherche sur les éléments qui suivent dans la table.

Une solution naïve à ce problème est de déplacer *toutes* les entrées aux positions $h(k, \ell)$ pour $\ell > j$ où $h(k, j)$ est l'indice de la valeur à retirer à chaque suppression. Il est assez évident qu'une telle approche peut très vite s'avérer très lourde s'il y a beaucoup de suppressions dans la table et est dès lors à proscrire.

Une solution plus intéressante est de *marquer* les cellules de la table qui ont contenu une entrée dans le passé mais qui ont été supprimées depuis. Ainsi, lors de l'exécution de la méthode `search`, ces cellules ne forceront pas l'arrêt de la boucle, mais la recherche continuera bien jusqu'à trouver soit `None` soit une entrée de la clef recherchée.

2.1.2 Spécificités d'implémentation

Vous ne pouvez bien entendu pas utiliser la classe `dict` de Python qui implémente déjà une table de hachage mais vous avez bien sûr le droit d'utiliser la classe `list` afin de représenter un tableau.

Cependant faites bien attention : une `list` en Python n'est **pas** une liste chaînée mais est bien un vecteur de taille dynamique, en particulier l'insertion d'un élément quelconque ne se fait pas en temps constant !

Vous avez le droit d'utiliser tous les codes vus au cours théorique, aux séances d'exercices ou dans les exercices cotés précédents. Faites bien attention à les utiliser de manière pertinente et correcte.

2.2 Rapport à écrire

Avec ce code, écrivez un rapport en LaTeX de maximum 5 pages (page de garde comprise) en suivant le *template* fourni avec cet énoncé 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 de hachage implémentées ;
2. des statistiques sur le temps nécessaire pour l'exécution des méthodes `insert`, `search` et `delete` de la classe `DictOpenAddressing` 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 ;

4. une comparaison du nombre de sondages effectués par les méthodes `insert`, `search` et `delete` des classes `DictChainingLinkedList` et `DictChainingSkipList` ;
5. une discussion sur les avantages et inconvénients de ces deux classes ;
6. les difficultés que vous avez rencontrées lors de la réalisation de ce projet.

Remarque

Il est possible d'améliorer légèrement la méthode `DictOpenAddressing.delete` afin de garder le nombre de sondages infructueux relativement bas : lors d'une recherche d'un élément existant dans la table, avant de renvoyer l'élément demandé, l'entrée associée dans la table est *déplacée* à l'indice de la première case marquée qui a été visitée pendant les sondages.

Si le cœur vous en dit, vous pouvez, pour un maximum de 2 points bonus sur 20, ajouter des éléments à ce projet tels que :
- l'implémentation de l'amélioration ci-dessus ;
- la paramétrisation des fonctions de hachage utilisées pour les différentes classes utilisées ;
- une discussion de l'influence des différentes fonctions de hachage sur l'efficacité de vos implémentations ;
- une classe `ResizableDictOpenAddressing` qui, lors d'une insertion dans une table pleine, ajuste la taille du conteneur afin d'éviter un overflow (attention en ce cas à vous assurer que les éléments placés précédemment soient toujours accessibles).

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 33 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 16 mai à 23 : 59 : 59 au plus tard ;
4. vous devez remettre une unique archive appelée `projet3.zip` respectant l'exemple fourni sur sur l'UV avec cet énoncé ;
5. un projet remis après la deadline **stricte** sera considéré comme non-remis et ne sera donc pas corrigé ;
6. respectez **scrupuleusement** les consignes (en particulier les noms de fichiers lors de la remise) car tout manquement à ces consignes sera sévèrement sanctionné ;
7. **attention** : ceci est un cours d'algorithmique, donc veillez bien à rendre votre code le plus efficace possible ;
8. venez poser toute question sur ce projet aux séances de Q&A et aux TPs, mais notez que je **ne répondrai pas** aux questions par mail.