ULB UNIVERSITÉ LIBRE DE BRUXELLES

Rapport Projet d'algorithmiques 3

Secundar Ismael

504107

16 Mai 2021

Table des matières

1 Introduction

L'objectif du projet 3 du cours INFO-F103 est d'implémenter en Python 3 différentes façons de résoudre les collisions des tables de hachage. Ce rapport a pour but de comparer les différentes méthodes implémentées entre elles ainsi que leurs complexitsé. De plus, les résultats obtenus seront comparés avec celles vue au cours.

La première implémention demandée était celle de la classe DictOpenAddressing. Une table de hachage peut régler les collisions soit par chaînage, soit par double hachage. Dans ce cas-ci, la méthode utilisée est celle du double hachage. Cette classe va analyser la table jusqu'à trouver une place correspondant à ce qui est recherché en utilisant la fonction de hachage h(k, j) = h1(k) + j *h2(k).[1]

La deuxième implémentation demandée était celle de la classe DictChainingLinkedList. Celle-ci gérait les collisions par chaînage. Pour cela, la classe UnorderedList et la classe Node ont été utilisé. Pour être un peu plus clair, nous allons illustrer cela avec une figure.

Sur la figure I, 19 a été ajouté à la clé 1 malgré que la clé 1 soit déjà utilisé par 28. Pour résoudre cette collision la classe DictChainingLinkedList va créer une UnorderedList à la case du tableau où il y a déjà un élément présent.

La troisième implémentation demandée était celle de la classe DictChainingSkipList. Cette classe est une amélioration de la classe DictChainingLinkedList, elle est organisée en plusieurs niveaux. Au niveau le plus bas, nous aurons une liste chaînée qui contient tous les éléments. Chaque couche supérieure contiendra un nombre inférieur d'éléments comparé à celle en dessous d'elle. Afin de visualiser cela nous allons présenter une figure explicative.

Nous pouvons ainsi voir sur la figure II que la première couche englobe tous les éléments alors que la couche juste au dessus ne contient que 4 éléments à savoir : 1,3,4,6,9.

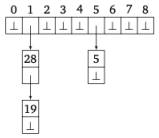


Figure I: Chaining Linked List

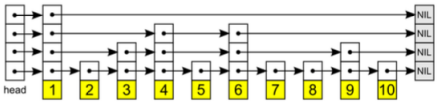


Figure II: Skip List

2 Méthodes

Nous avons effectué 3 tests différents. Le premier consistait à récupérer le temps d'exécution de chaque méthode pour pour la classe DictOpenAddressing , à savoir : l'insertion, la suppression et la recherche d'un élément dans la table en fonction du Loadfactor. Le LoadFactor est le rapport entre le nombre d'élément présent dans la table par la capacité totale. Le deuxième consistait à comparer le nombre de sondage qui sont effectué avant de trouver un élément dans la table à partir des résultats théoriques et ceux expérimenté. Le dernier test compare le temps d'exécution des méthodes pour la classe DictChainingSkipList et DictChainingLinkedList.

[1] h1 et h2 sont deux fonctions de hachage. h1 est défini selon l'algorithme de Kernighan & Ritchie alors que h2 selon l'algorithme djb2 de Daniel J. Bernstein

2.1 Temps d'exécution

2.1.1 En théorie

Nous avons vu lors du cours d'algorithmiques que le temps moyen pour une recherche prend en moyenne un temps 1 / (1-α) pour une recherche sans succès, alors qu'avec succès cela prenait ((1/α)*ln*1/(1-α).

2.1.2 En pratique

Pour tester le temps d'exécution, nous avons initialisé toutes les tables avec une capacité de 100. La table est initialement vide. Le LoadFactor est donc de 0/100 au début. On augmentera le LoadFactor de 1 jusqu'à ce qu'on remplisse complètement la table.

Le temps d'exécution a été multiplié par 10^6 pour faciliter la lisibilité des graphiques.

3 Résultats

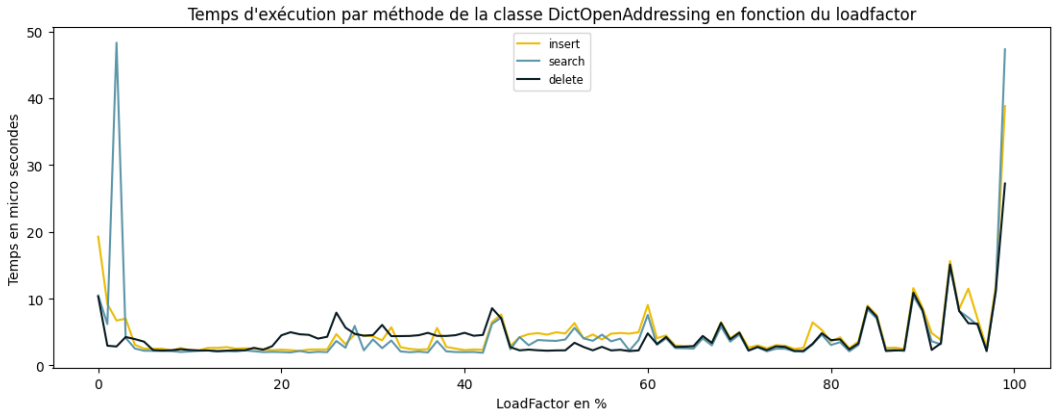
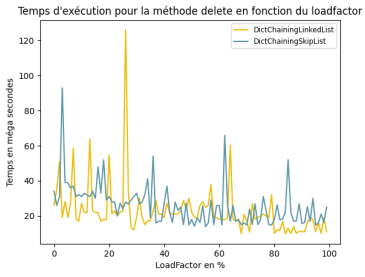
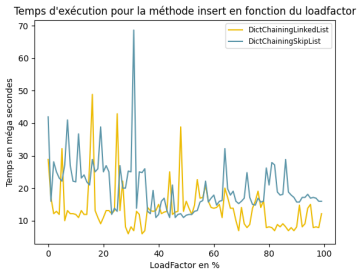


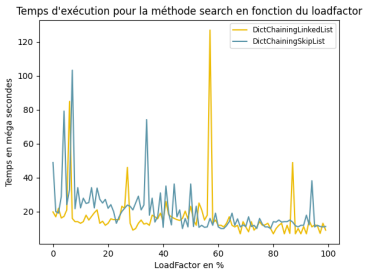
Figure III: Temps d'exécution par méthode pour la classe DictOpenAddressing



(1) Méthode delete



(2) Méthode insert



(3) Méthode search

Figure IV: Comparaison entre DictChainingLinkedList et DictChainingSkipList

Sondage moyen		
Méthode	Sondage théorique	Sondage expérimenté
Search avec succès 50%	<1,387	1,18
Search avec succès 90%	<2,559	1,54
Search sans succès 50%	2	2,08
Search sans succès 90%	10	4,34

Table 1: Sondage moyen

4 Discussion

4.0.1 Sondage

Nous observons sur la figure III que le temps d'exécution pour supprimer, rechercher et insérer un élément avec la classe DictOpenAddressing est globalement constant pour un LoadFactor <80% mais au delà de cela il se déstabilise. Il était difficile de réaliser les tests, nous avons donc enlever l'erreur qui était levée, à savoir, OverflowError("La table est pleine").

D'après les résultats que nous avons récupéré, l'on peut voir que le sondage moyen théorique est conforme aux résultats obtenus car il était mis que le nombre moyen de sondages est inférieur à 1.387 si la table est à moitié remplie. Nos résultats sont bien inférieur à 1.387. De plus, selon la théorie, si la table est remplie à 90%, le nombre moyen de sondages est inférieur à 2.559. Ce qui prouve encore une fois que la théorie était juste vue que l'on ne dépasse pas 2.559. Cependant, il nous est arrivé de dépasser 2 sondages pour la recherche sans succès. Sachant que ces résultats sont une moyenne de 10 tests que nous avons lancé. Mais rien de tout cela n'est alarmant car la différence n'est que de 0,08.

Pour ce qui en est de DictChainingSkipList et DictChainingLinkedList. Le sondage moyen sera de 1 vue que les collisions sont gérés. Pour la DictChainingSkipList on a plusieurs niveaux et les collisions sont gérés de telle sorte à ce qu'il y ait plusieurs listes. Pour DictChainingLinkedList, sur chaque cas où il y a déjà un élément un liste chainée va être créé.

4.0.2 Temps d'exécution

D'après les résultats obtenus sur la Figure IV, nous pouvons voir que DictChainingLinkedList est plus optimal que DictChainingSkipList. DictChainingLinkedList a globalement un temps d'exécution plus petit que DictChainingSkipList pour chaque méthode.

4.0.3 Avis externe

D'apres Aditya Chatterjee et Ethan Z. Booker[2], un des avantages des SkipList est que l'on peut accéder à l'élément en un temps constant. Mais elle n'est pas très amicale avec la cache. Elle prenne plus de place qu'un ABR.SkipList serait selon eux, une alternative aux listes chainées.

D'après Steve Zaretti[3], La Skiplist peut-être vite implémentée mais elle peut parfois être moins optimale qu'il le faut à cause des probabilités.

[2]Probabilistic Data Structures
[3]https://steve.zaretti.be/files/skiplist.pdf

5 Conclusion

Pour terminer, nous pouvons dire que les résultats théoriques sont globalement similaires à ceux obtenus lors des résultats. DictOpenAddressing possède une performance limitée. DictChainingSkipList peut parfois s'avérer être moins performant que DictChainingLinkedList. Tout dépend des probabilités. Pour finir, DictChainingSkipList est moins optimal que DictChainingLinkedList au niveau du temps d'exécution.