Berney Alec, Forestier Quentin, Herzig Melvyn

ASD2 – 26.12.2020

Tables de hachage et correcteur orthographique

Laboratoire 4



Table des matières

[Introduction 2](#_Toc59887498)

[Première partie : Tables de hachage 3](#_Toc59887499)

[Fichiers principaux 3](#_Toc59887500)

[Question 1.1 3](#_Toc59887501)

[Génération de la structure 3](#_Toc59887502)

[Récupération de la structure 3](#_Toc59887503)

[Complexité totale 3](#_Toc59887504)

[Question 1.2 3](#_Toc59887505)

[Question 1.3 4](#_Toc59887506)

[Question 1.4 4](#_Toc59887507)

[Deuxième partie : Correcteur orthographique 5](#_Toc59887508)

[Fichiers principaux 5](#_Toc59887509)

[Justification choix des structures STL. 5](#_Toc59887510)

[Utilisation 5](#_Toc59887511)

[Comparaison des structures 6](#_Toc59887512)

[Question 2.1 7](#_Toc59887513)

[Question 2.2 7](#_Toc59887514)

[Question 2.3 7](#_Toc59887515)

# Introduction

Ce laboratoire a été réalisé dans le cours « Algorithmes et structures de données 2 ». Dans un premier temps nous implémenterons deux tables de hachages : une par chaînage et l’autre par sondage linéaire. En seconde partie, nous réaliserons un correcteur orthographique anglophone simplifié qui emploiera, nos deux structures précédentes, deux structures de la STL et un arbre de recherche ternaire implémenté par nos soins. Au terme de ce travail, nous comparerons les performances des différentes approches.

# Première partie : Tables de hachage

## Fichiers principaux

|  |  |
| --- | --- |
| **HashMapCommon.h** | Classe contenant les méthodes virtuels que doivent implémenter HashMapChain et HashMapLinearSample ainsi que leurs implémentations communes. |
| **HashMapLinearSample.h** | Classe implémentant une table de hachage par sondage linéaire. |
| **HashMapChain.h** | Classe implémentant une table de hachage par chaînage. |
| **Person.h** | Classe modélisant une personne et redéfinissant la fonction de hachage de la STL. |

## Question 1.1

Bien que les unordered\_set de la STL proposent d’insérer, chercher et supprimer des clés en O(1), ils sont non ordonnés.

Admettons que nous voulions stocker un grand nombre de données et régulièrement nous cherchons à trier le contenu de la structure en constantes évolutions

Posons N = #données distinctes stockés dans le set et l’unordered\_set à un moment M.

Posons T = #de fois que nous désirons trier le contenu.

### Génération de la structure

|  |  |
| --- | --- |
| Set O(N\*log(N)) | Unordered\_set O(N) |

### Récupération de la structure

|  |  |
| --- | --- |
| Set O(N) | Unordered\_set O(N) |

A cette étape, le set a l’avantage d’être déjà trié.

Quant à l’unordered\_set, son contenu doit encore être trié en O(N\*log(N))

### Complexité totale

|  |  |
| --- | --- |
| Set O(T\*N+N\*Log(N)) | Unordered\_set O(N+T\*N\*Log(N)) |

Ainsi dans ce genre de situation, la structure set est préférable à l’unordered\_set pour un T élevé.

## Question 1.2

Les collisions sont gérées par chainage. Les valeurs qui génèrent la même clé sont stockées dans le même « bucket » (équivalent des listes dans la théorie du cours).

Le max\_load\_factor est le ratio entre le nombre d’élément réellement stocké et le nombre de buckets. Quand le ratio, par défaut à 1, est dépassé, la nombre de buckets et augmenté.

## Question 1.3

Cela permet de ne pas doubler inutilement la taille de la table de hachage. La table garde ainsi une taille relative au nombre d’élément que nous y insérons/supprimons.

## Question 1.4

Le problème de la fonction de hachage ainsi proposé est qu’elle est incomplète. Seule une partie de l’état d’un objet personne est hachée et forme ainsi la clé. En l’occurrence, deux personnes de même nom auront le même hash. Comme il n’est pas rare que deux personnes aient le même nom (comme dans une même famille), les collisions seront fréquentes.

Dans le fichier Person.h nous proposons notre amélioration de la fonction de hash pour un objet Person. Conformément à la recette c++, la clé est une constante non nulle à laquelle nous additionnons le hash de chaque attribut de l’objet multiplié par 31. De cette façon, la hachage classique C++ est implémenté et minimise les collisions entre personnes.

# Deuxième partie : Correcteur orthographique

## Fichiers principaux

|  |  |
| --- | --- |
| **Corrector.h** | Classe permettant de corriger une fichier texte selon un fichier dictionnaire. |
| **TSTreeWords.h/.cpp** | Classe implémentant un arbre ternaire de recherche équilibré. |
| **AdaptaterCorrector.h** | Classe permettant d’interfacer une structure STL avec Corrector.h. |

## Justification choix des structures STL.

Nous avons décidé de choisir deux structures qui n’étaient pas inadaptée pour cette tâche.

**unordered\_set** : Cette structure est une table de hachage par chaînage. Par nature, elle est adaptée à la situation, elle permet une insertion et récupération de ses éléments en O(1), idéal pour un dictionnaire. Elle permettra de définir l’idéal à atteindre.

**set :** Cette structure permet de tester une approche dichotomique. Ses performances en O(logN) en récupération permettront de poser le seuil à ne pas dépasser tout en étant raisonnable.

## Utilisation

Dans le fichier main.cpp :

**Note :** Le fichier de correction sera toujours généré dans le répertoire du fichier à corriger.

## Comparaison des structures

Pour cette analyse, nous exclurons les fichiers simple et lates. Ces deux fichiers sont corrigés entre 0 et 4 ms avec une différence de 1 ms entre les structures. Pour pouvoir les analyser correctement, il aurait fallu travailler avec des microsecondes. Les tendances seraient alors semblables à celles que nous allons observer avec wikipedia et sh.

Soit les complexités de la fonction « contains » suivantes :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table de hachage chaînage | Table de hachage sondage | set | unordered\_set | Ternary search trie |
| O(1) | O(1) | O(log₂N) | O(1) | O(L\*log₂26) |

**L** = longueur du mot, **N** = nombre de mots

Comme attendu, la structure la moins rapide est le std ::set avec un accès aux données en O(logN), elle est deux fois plus lente que l’avant dernière structure.

Nous pouvons ensuite remarquer que les quatre autres structures se valent. On comprend facilement que nos tables de hachages soient en compétition avec le unordered\_set de la STL car toutes ont la même complexité en O(1).

**Mais pourquoi le TST n’est-il pas avec std ::set ?**Le ternary search trie est un arbre qui travaille au niveau des lettres et non au niveau des mots comme le standard set. Cela implique que rechercher un mot, parmi les 637’850 mots du dictionnaire, demandera plus de recherches en comparant les mots entre eux plutôt que les lettres par rapport à leur position dans le mot. En plus le TST tire avantage du fait que les mots d’une langue ne sont pas aléatoires et que certains sont des préfixes d’autres, que certaines lettres sont peu/pas utilisées ce qui réduit encore plus la vitesse de recherche. Ainsi les opérations demandées pour rechercher un mot dans un TST sont inférieures à celle dans un set. De plus, elles sont partiellement compensées par les quelques opérations de décalage et de hachage dans les tables de hachage. C’est pourquoi, nous obtenons de performances très similaires.

**Quid des tables de hachage avec chaînage et sondage linéaire ?**Toutes deux sont semblables à l’unordered\_set et au TST. Etonnamment, notre implémentation de la table de hachage par sondage semble sensiblement plus rapide tandis que celle par chaînage plus lente que son égal de la STL**.** Nous n’avons aucune explication stricte concernant ces deux phénomènes. Toutefois, nous pensons que la table par sondage est plus rapide car elle demande de parcourir de moins gros ensemble d’éléments pour trouver une appartenance. Concernant les performances entre table de hachage par chaînage et unordered\_set, nous spéculons sur le fait que la STL effectue des optimisations que nous ne faisons pas car nous n’en avons pas connaissance. Malgré tout, les performances restent respectables et compétitives.

## Question 2.1

Avantages :

* Demande moins d’espace que les tables de hachage.
* N’a pas besoin de l’entièreté d’un string pour se rendre compte qu’il n’est pas stocké.
* Fonctionnalité supplémentaire[[1]](#footnote-1), par exemple « near-neighbor searching ».

Inconvénients :

* Efficace pour des types décomposable comme des nombres ou des chaînes de caractères, mais plus difficile à mettre en place avec des objets complexes.

## Question 2.2

Les fonctions de hachages permettent de vite savoir où se situe un élément dans un tableau, offrant ainsi une insertion/suppression/vérification constante. Toutefois, elles ne sont jamais exempts de collisions.

Comme nous l’avons vu, moyennant le traitement par chaînage ou sondage, elles peuvent être utilisées pour la mise en place de notre dictionnaire. Au contraire, si nous l’utilisons avec un simple tableau, la méthode n’est pas adaptée. En cas de collision, la seconde valeur écraserait la première, ce qui entrainerait un dictionnaire incomplet et donc incorrect.

## Question 2.3

L’utilisation d’un tableau n’est pas adaptée car la vérification d’appartenance d’un mot se fait en O(N) avec N le nombre de mots distincts du dictionnaire. Ainsi pour un texte de M mots, on a un chargement du dictionnaire en O(N) et une correction en O(MN).

En cas d’obligation, la meilleure méthode est d’employer un tableau trié pour entreprendre des recherche dichotomiques.

Le chargement du dictionnaire serait vraiment mauvais en O(N²). Mais les futures corrections en O(M\*log₂N).

# Conclusion

Au terme de ce projet, nous avons réussi à mettre en place des structures de différents types qui n’ont pas à rougir devant les performances de la STL. Nous avons approfondi nos connaissances relatives aux hachages, aux tables de hachages ainsi que le arbres ternaires de recherche.

Comme nous l’avons discuté, le choix de la bonne structure est important pour avoir un programme performant.

1. https://en.wikipedia.org/wiki/Ternary\_search\_tree [↑](#footnote-ref-1)