Rapport ASD2 - Labo4 Table de hashage

James Smith, Jérémy Chatillon et Adrien Allemand

# Introduction

Dans ce laboratoire nous avons étudier les tables de hachage et plus particulièrement différentes fonctions de hachages utilisées pour remplir une structure du type std::unordered\_set.

# Partie 1

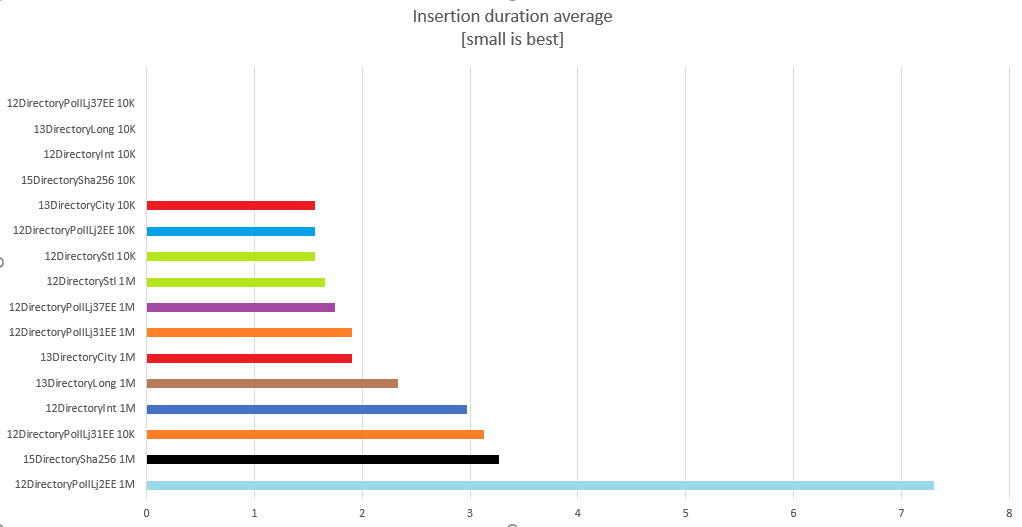
Pour analyser l’efficacité des fonctions de hashage, il faut prendre en considération plusieurs paramètres :

* Durée d’insertion moyenne d’une donnée
* Le nombre de « bucket » (emplacement dans la liste de hashage)
* Le nombre de collision (plusieurs éléments dans un même bucket)
* Le nombre de bucket vide (Place utilisé pour rien)
* La taille du plus gros bucket (Taille de la collision la plus grosse)
* Le temps moyen pour trouver un éléments

Pour avec une idée plus élargie, nous allons tester sur deux tailles de données, 10k et 1000k. Toutes l’analyse suivante s’est faite avec un load factor de 0.5.

## Durée d’insertion moyenne d’une donnée

La durée d’insertion moyenne permet de savoir si l’algorithme utilisé permet une insertion rapide des données dans la table de hashage.

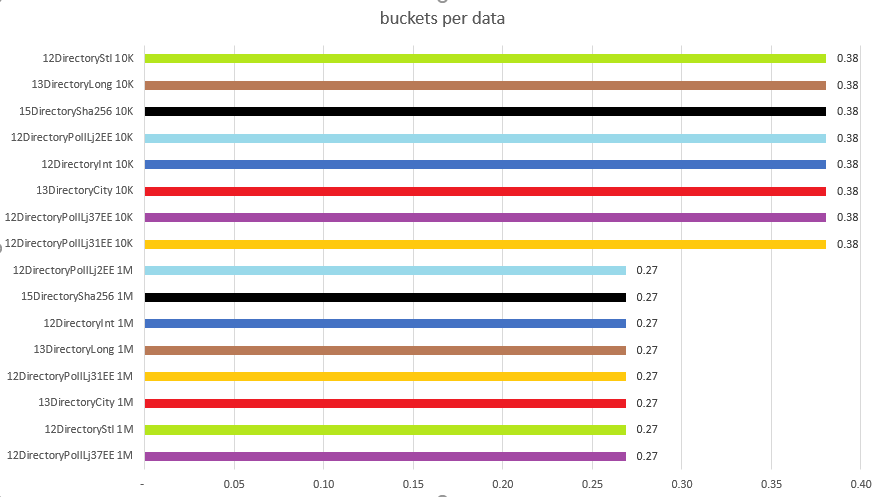


Les algorithmes :

* **12DirectoryPoIl37EE** : Très rapide avec peu de données et encore efficace avec beaucoup (top 2).
* **12DirectoryStl** : Le temps avec peu ou beaucoup de données est presque équivalent. Donc très rentable plus on a de donnée mais pas efficace quand on a peu.
* **12DirectoryPoIlj2EE** : est pas efficace avec peu et beaucoup de donné.
* **15DirectorySha256** : Très rapide avec peu mais très lent avec beaucoup.

## Nombre de bucket

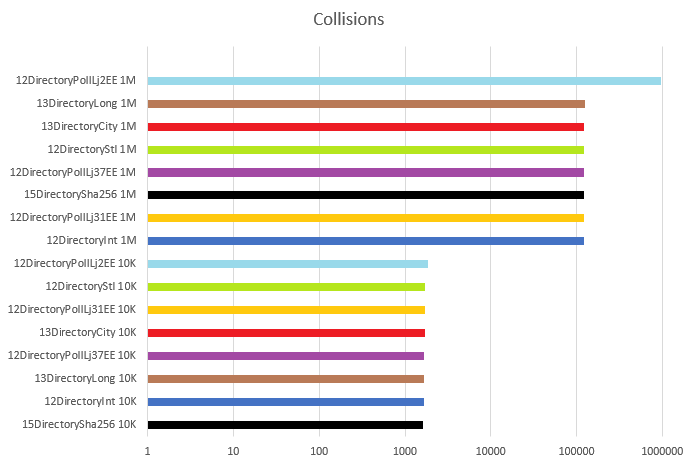
Le nombre de bucket permet de savoir combien d’indice sont disponible pour accueillir des valeurs.



Le nombre de bucket est totalement similaire pour chaque algorithme

## Le nombre de collision

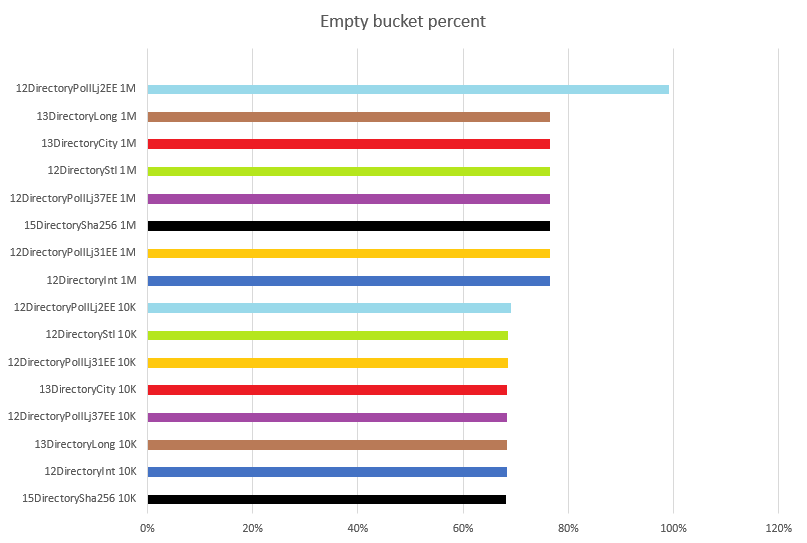
Le nombre de collision montre combien d’élément ont eu le même hash.



Pour des petites quantités de donnée, il y a autant de collision avec tous les algorithmes et avec plus de données, il y a que l’algorithme 12DirectoryPoILj2EE qui est en dessus des autres.

## Le nombre de bucket vide

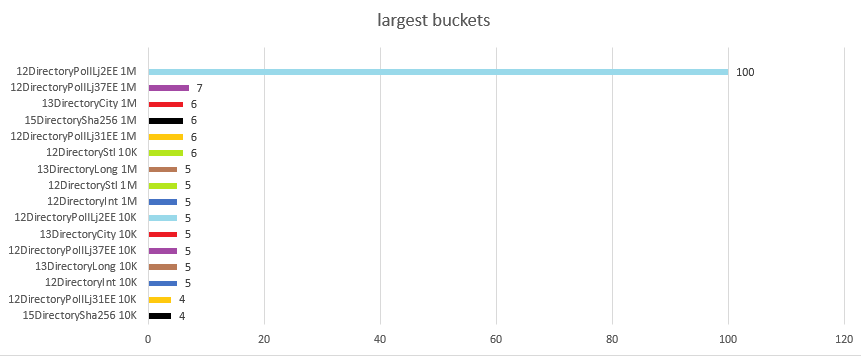
Le nombre de « bucket vide » montre le nombre d’indices qui n’ont pas été utilisé.



Même analyse que pour le nombre de collision. Tous très similaire, à part pour le 12DirectoryPoILj2EE.

## La taille du plus gros bucket

La taille du plus gros bucket montre le nombre de collision maximum qu’il y a eu sur un même indice (hash).

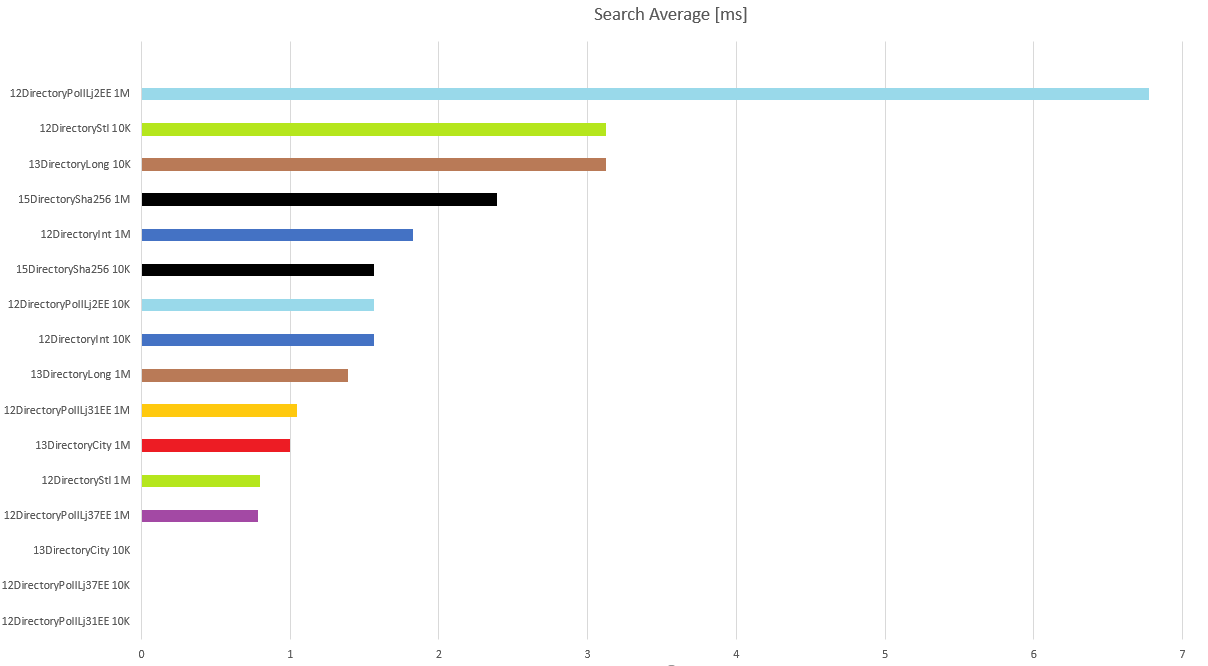


Les algorithme 12DirectoryPoILj31EE et 15DirectorySha256 possède le minimum avec 4 éléments. Les autres algorithmes sont entre 4 et 7 éléments. Il y a que 12DirectoryPoILj2EE qui est de nouveau supérieur à la moyenne avec 100 élément dans un buckets lorsqu’il y a 1M de données.

Chose surprenante, l’algorithme 12DirectoryStl à moins un plus petit bucket lorsque les quantités de données sont élevées.

## Le temps moyen pour trouver un élément

C’est le temps que met l’algorithme pour trouvé un élément dans notre table de hashage.



Les algorithmes 12Directory37EE, est très rapide (top 1) avec des petites données et est extrêmement rapide comparé aux autres avec un grand jeu de données (top 1).

L’algorithmes 13DirectoryCity est lui aussi très rapide dans les deux cas.

L’algorithmes 12Directory est très rapide avec beaucoup de données (top2) mais très lent avec peu de données.

## Analyse

Les caractéristiques qui nous intéressent le plus en général en utilisant ces structures de données c’est l’écriture et la lecture des données. D’après ce critère, l’algorithme **12DirectoryPoIl37EE**  est le plus efficace avec un load factor de 0.5.

On peut en déduire que le nombre de collision et la taille du plus gros bucket ne change pas la rapidité d’un algorithme.

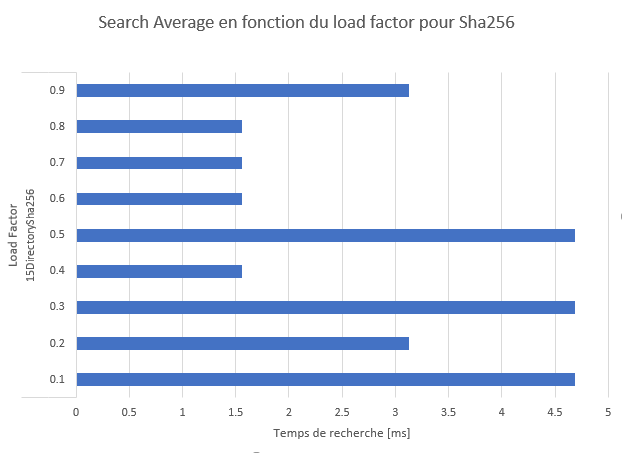
L’algorithme 12DirectedPoIL2EE est lui très peu efficace dans tous les cas…

## Le load factor

Nous avons ensuite testé le temps d’accès aux données en changeant le load factor.

Nous avons constater que cela changeait assez drastiquement mais de façon assez aléatoire.

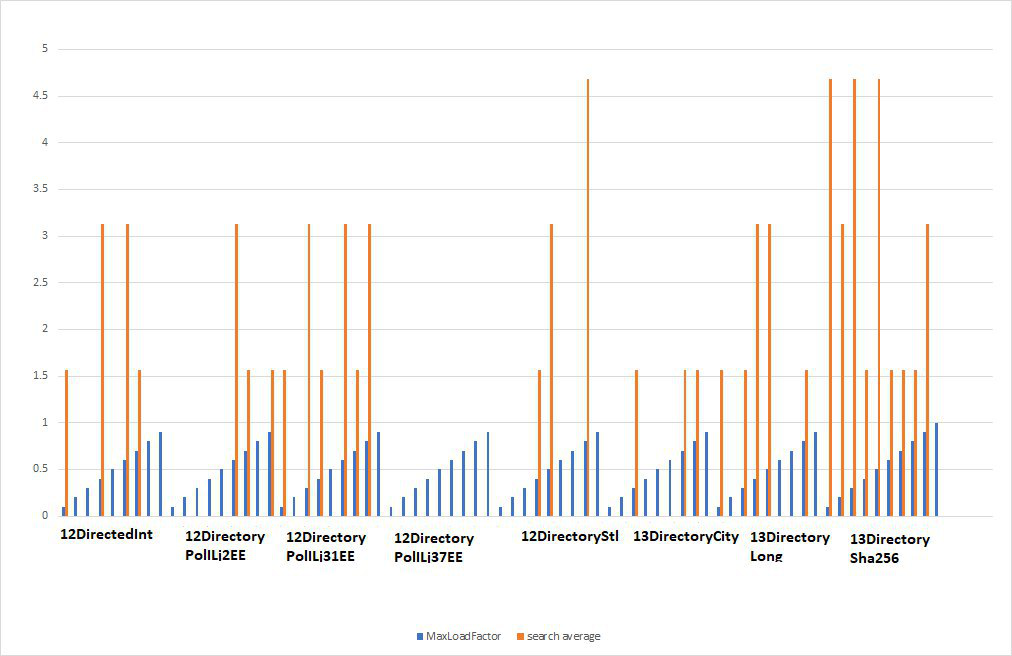
Par exemple regardez l'impacte du load factor sur le temps de recherche après l'utilisation de la fonction SHA256 :



Nous avons choisi SHA256 car elle ne retourne aucune valeur de 0 pour les temps de recherche. On constate bien qu'il y a une zone entre 0.8 et 0.6 ou la fonction semble trouver plus rapidement mais comment généraliser ce cas et pourquoi restent des mystères.

Devant le temps colossal que prenait le calcul de 10x chaque test avec le ficher à 1M d'entrées (plus de 20 minutes sans résultat) nous nous sommes contentés du fichier à 10K entrées.

Voici un graphique résumant l'impacte du load factor sur chaque algorithme de hashage (les barres bleu sont le maxloadfactor trié de 0.1 à 0.9 par pallier de 0.1, les barres orange sont le temps de recherche. On constate bien l'aléà de la chose.



# Partie 2

Dans cette partie, nous avons cherché à minimiser les collisions. Pour ceci, la valeur à passer dans la fonction de hash doit provoquer le moins de collisions que possible et pas trop long. C’est pourquoi le numéro AVS correspondait bien. Donc nous avons pensé à utiliser le nom, prénom et dates d’anniversaire. Nous avons éliminé le genre car cet attribut est trop restrictif.

Nous avons premièrement choisi de faire une chaine composée de certains caractères de ces 3 attributs. Nous nous sommes dit que les 2 premiers caractères ainsi que les deux derniers du nom et du prénom sont suffisamment uniques. Nous avons aussi ajouté les 2 premiers caractères de la date qui correspondent au jour (ce qui provoques moins de collisions que l’année).

|  |
| --- |
| string toAsh = name.substr(0,2) + name.substr(name.size() -2, name.size()) + firstName.substr(0,2) + firstName.substr(firstName.size() -2, firstName.size()) + d.getBirthday().substr(0, 2);  return std::hash<string>()(toAsh);  Insertion:  Distribution:  Number of buckets: 126271  Number of collisions: 366 (3.66%)  Number of empty buckets: 116637 (92.3704%)  Largest bucket: 3  Smallest bucket: 0  Search:  Search total: 10000  Found total: 10000 (100%)  Search duration total time: 24082 microseconds  Search duration average time: 2.4082 microseconds |

Cependant, cette méthode est un peu trop faite à tâtons. Elle peut aussi être dangereuse si un attribut est formé d’une chaine de caractère plus petite que celle recherché.

Nous avons donc opté pour une deuxième option qui consiste à hasher le nom, prénom, date et ensuite concaténer les résultats en string et rehasher celui-ci.

|  |
| --- |
| size\_t ashName = hash<string>()(d.getName()) ;  size\_t ashFirstName = std::hash<string>()(d.getFirstname());  size\_t ashDate = std::hash<string>()(d.getBirthday());  string toAsh = to\_string(ashName) + to\_string(ashFirstName) + to\_string(ashDate);  return std::hash<string>()(toAsh);  Insertion:  Distribution:  Number of buckets: 126271  Number of collisions: 377 (3.77%)  Number of empty buckets: 116648 (92.3791%)  Largest bucket: 3  Smallest bucket: 0  Search:  Search total: 10000  Found total: 10000 (100%)  Search duration total time: 77599 microseconds  Search duration average time: 7.7599 microseconds |

À notre plus grand étonnement, la nombre de collision était plus grand. Nous nous sommes dit que cela était dû au fait que le retour du hash donnait un eniter de 9 ou 10 caractères. Ce qui donne une fois concaténer ~30 caractères. Ce qui est un peu grand pour une clé de hashage.

Donc, par tâtons, nous avons cherché un nombre de caractères (en gardant les caractères finaux) car ils sont unique du début (ceux des millier, en référence au support de cour).

Nous avons finalement trouvé une clé optimale qui est :

|  |
| --- |
| size\_t ashName = hash<string>()(d.getName()) ;  size\_t ashFirstName = std::hash<string>()(d.getFirstname());  size\_t ashDate = std::hash<string>()(d.getBirthday());  string toAsh = to\_string(ashName%1000000) + to\_string(ashFirstName%100000) + to\_string(ashDate%1000000);  return std::hash<string>()(toAsh);  Insertion:  Distribution:  Number of buckets: 126271  Number of collisions: 336 (3.36%)  Number of empty buckets: 116607 (92.3466%)  Largest bucket: 3  Smallest bucket: 0  Search:  Search total: 10000  Found total: 10000 (100%)  Search duration total time: 70325 microseconds  Search duration average time: 7.0325 microseconds |

Cependant, il y a un désavantage temporel en appliquant la deuxième méthode. Cette méthode à besoin d’appliquer 4 fois la fonction de hashage. Si on est limité en temps, il et que les collisions n’ont pas d’importance, il faudrait utiliser un dérivé de la première méthode.

Une partie compliquée de cette partie 2 était que nous n’avons pas trouvé de fonction qui calcule la taille (nombre de caractère) optimal à appliquer en paramètre de la fonction de hash.

# Conclusion

Pour résumer simplement nos expérimentations sur les fonctions de hashage :

L'algorithme de hashage "12Directory37EE" est le plus performant en matière de temps de recherche de données et de temps d'insertion.

Le Max Load Factor semble avoir un impact différnet sur les fonctions de hashage. Il produit des graphiques incohérents où un load facteur incrémenté par pallier de 0.1 produit un graphique rempli de valeurs atypiques et ne forme ni courbe ni droites analysables.

La meilleure clé pour minimiser les collisions (voir -> encadré 3) est étonnamment moins performante au niveau de la durée moyenne de recherche que celle présenté dans l'encadré 1.