Introduction

Dans ce laboratoire, nous avons étudié les tables de hachage et plus particulièrement différentes fonctions de hachages utilisées pour remplir une structure du type std::unordered_set.

Partie 1

Nous allons prendre en considération plusieurs paramètres afin d'analyser l'efficacité des fonctions de hachage. Ces derniers seront représentés sous forme graphique selon les caractéristiques ci-dessous :

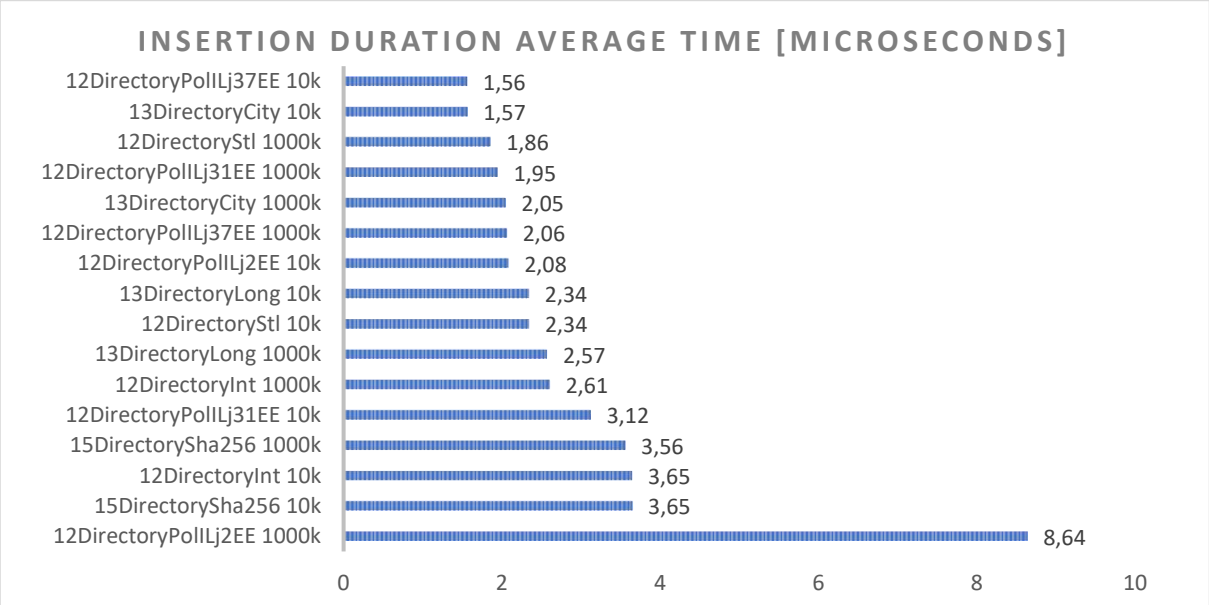
- Durée d'insertion moyenne d'une donnée
- Le nombre de bucket (nombre d'emplacement dans la liste de hachage)
- Le nombre de collision (plusieurs éléments dans un même bucket)
- Le nombre de bucket vide (place utilisée pour rien)
- La taille du plus gros bucket (taille de la collision la plus grosse)
- Le temps moyen pour trouver un élément

Toujours en fonction des différentes fonctions de hachage.

Les comparaisons se feront sur deux listes de donnée. L'une de 10'000 entrées et l'autre de 1'000'000. Nous avons réalisé 5 tests pour chacun des sets de données et algorithme et avons représenté les moyennes sur les graphiques ci-dessous.

Nous détaillerons finalement la variation du load factor (par défaut pour la première partie à 0.5).

Temps d'insertion



Deux algorithmes sortent du lot lors de l'insertion des données :

- 13DirectoryCity
- 12DirectoryPolILj37EE

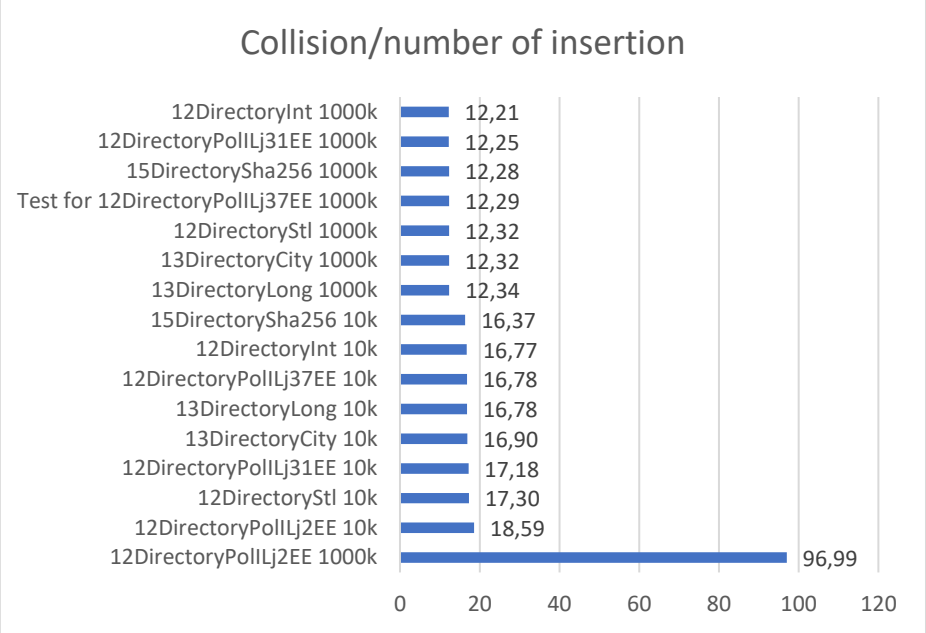
Ces derniers ont l'avantage d'être particulièrement lors de l'insertion de peu de données mais également avec de grosses listes.

L'algorithme 12DirectoryStl est quant à lui particulièrement efficace lors de l'insertion de gros sets de données, à l'inverse du 12DirectoryPolILj2EE qui, lui, devient particulièrement lent lors de l'insertion d'un gros set et en devient donc déconseillé malgré un très bon score lors de l'insertion d'une petite liste.

Les algorithmes en SHA256 sont globalement les plus lent sur toutes les listes. Ce dernier utilise très probablement un procédé plus complexe MAIS, comme nous allons le constater plus tard, ce dernier possèdera une bonne répartition des données.

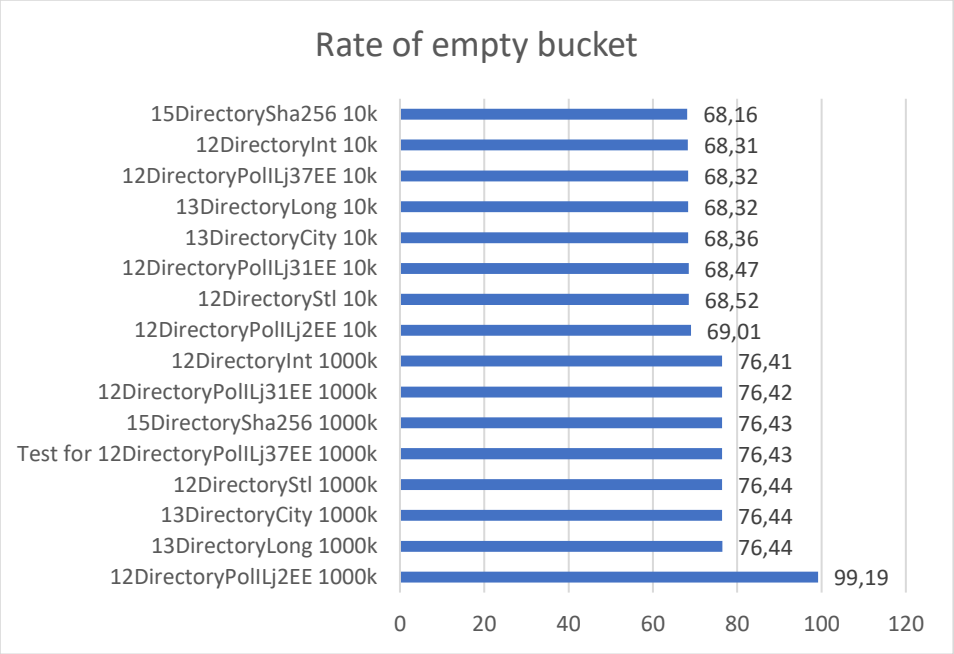
Collisions et buckets vides

Nous avons ici vérifié le nombre de collisions en fonction du nombre de donnée à insérer.

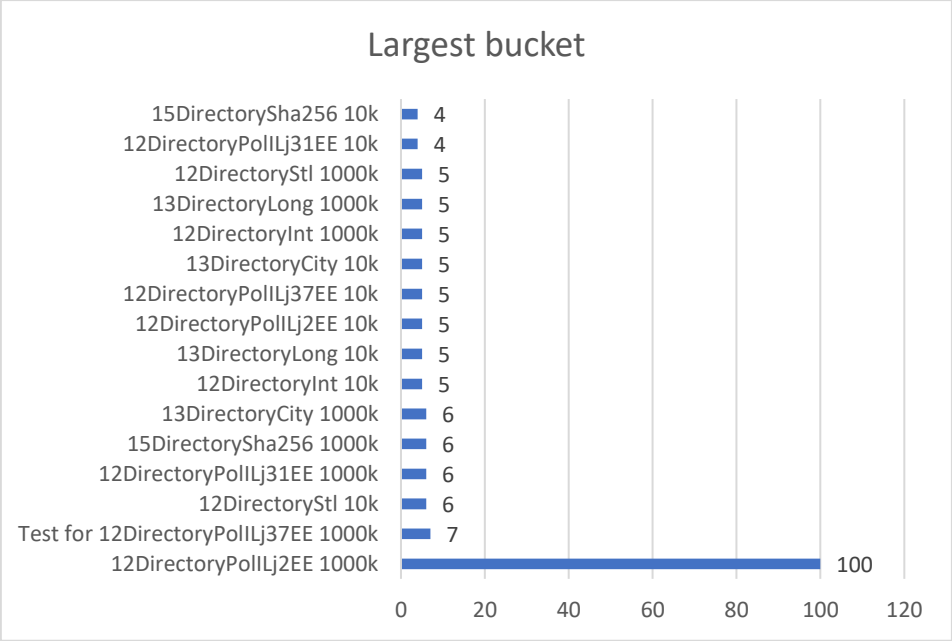


Le premier constat que l'on peut en tirer est la régularité de ce résultat. En effet, si l'on enlève le taux particulièrement élevé de l'algorithme 12DirectoryPolILj2EE (avec une grosse liste mais également plus élevé que les autres avec la petite), les autres se tiennent dans un intervalle très serré.

Soulignons également la bonne répartition des données insérée avec les algorithme SHA256 et directoryInt.

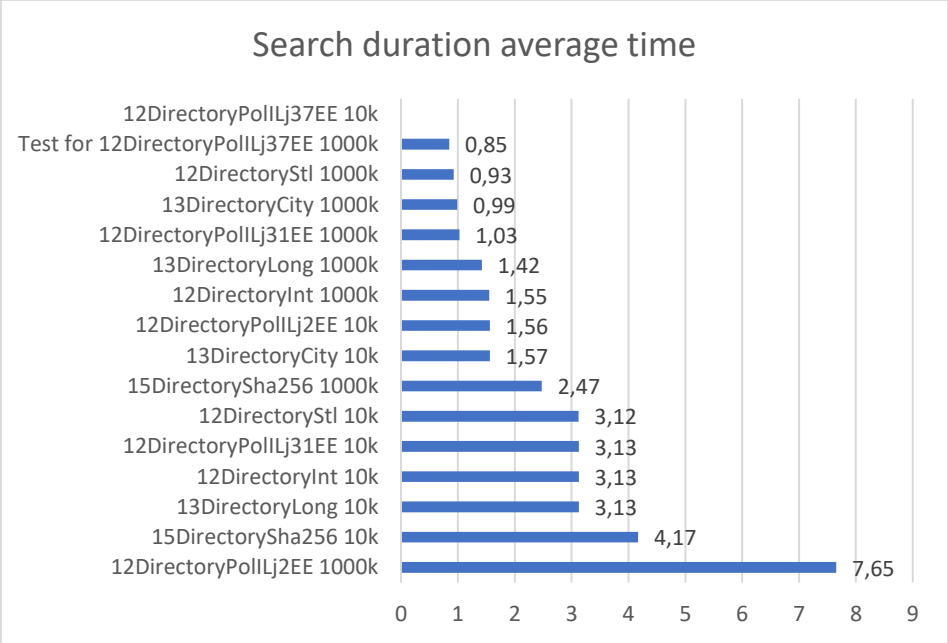


C'est ainsi le même algorithme qui termine 12DirectoryPolILj2EE qui termine en pied de liste, à la fois pour les petits que les grands sets de données. Sans surprise, les mêmes résultats apparaissent pour la taille des buckets.



Temps de recherche

Nous avons obtenu des résultats très différents d'une prise l'autre concernant les petits sets de données. Les valeurs avec les sets plus grands paraissaient plus régulières et ont ainsi moins de résultats surprenant.

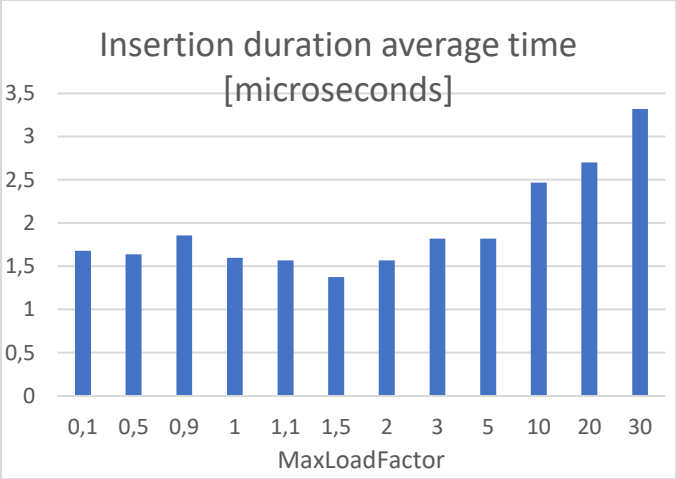


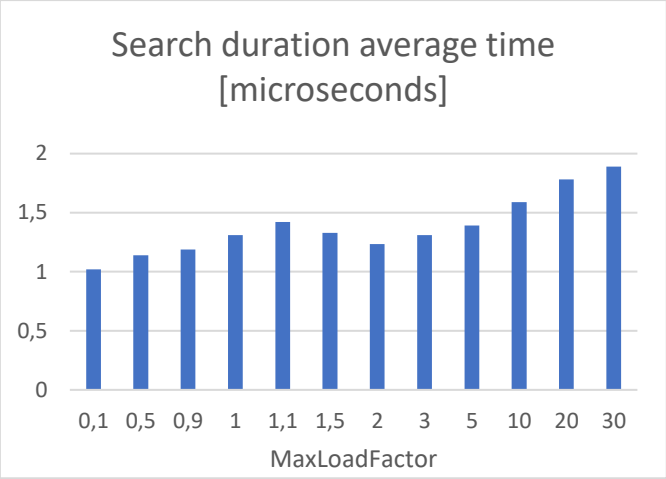
L'algorithme qui semble effectivement le plus rapide à chaque fois est l'algorithme 12DirectoryPolILj37EE. En effet ce dernier est celui qui termine systématiquement en tête tant avec de petites que de grandes listes.

Variation du maxLoadFactor

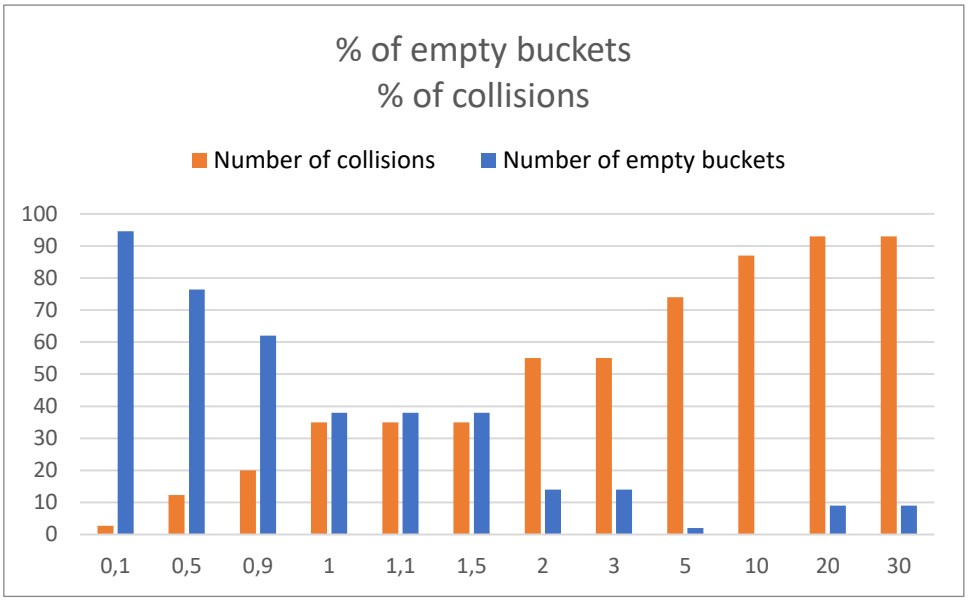
Nous avons effectué des testes de variation du MaxLoadFactor sur l'Algorithme sha256. Ce dernier nous fournissait lors de la première partie les résultats les plus stables c'est pour cela que nous l'avons choisi pour cette partie.

Les tests ont été effectué en faisant varier le LoadFactor et chacun d'eux a été effectué 5 fois afin d'en calculer la moyenne sur la liste de 1000k données et en release mode.

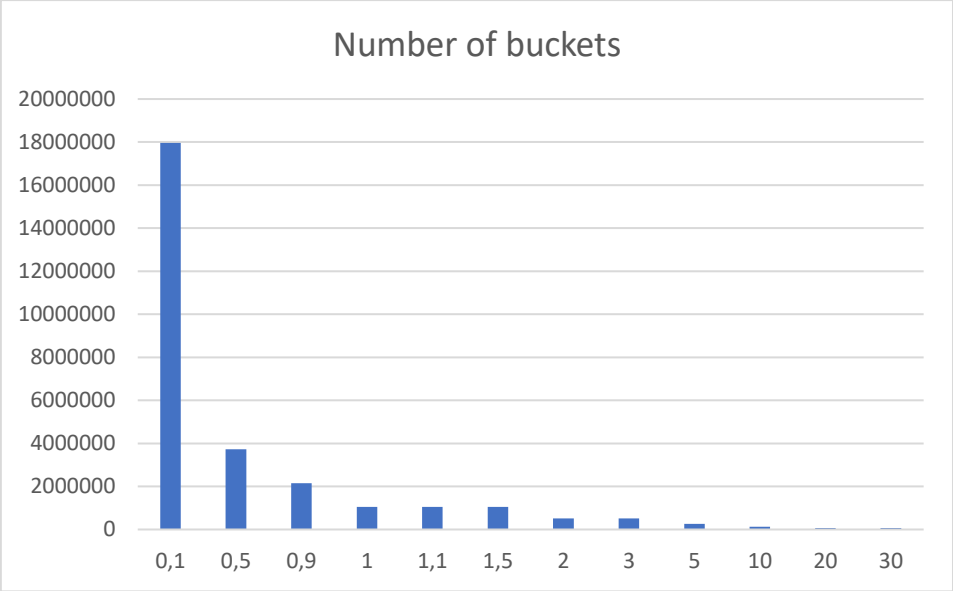




Il est relativement difficile d'affirmer haut et fort une tendance particulière en ce qui concerne les temps d'insertions et de recherche.



Cependant, il est évident que la variation de ce paramètre va en contrepartie influencer le nombre de collision, de nombre de bucket et donc de bucket vide. Le graphique ci-dessus représente le pourcentage de bucket vide qui diminue fortement (en bleu) et le pourcentage de collision qui va donc augmenter. Cela provient du nombre de bucket qui diminue fortement comme on peut le constater dans le graphique ci-dessous.



La valeur par défaut du max_load_factor pour l'unorderd_map est de 1.0. Pour rappelle il représente le ratio entre le nombre d'élément et le nombre de bucket. Les résultats ci-dessus nous paraissent donc pertinent.

Partie 2

La fonction de hachage donnée pour DirectoryWithoutAvs, qui hash uniquement le nom des personnes, n'est pas suffisante. Il peut y avoir trop facilement des personnes avec le même nom de famille et donc un nombre beaucoup trop élevé de collisions.

Comme vu dans le cours, il est fortement recommandé de combiner chaque attribut de la classe dans la fonction de hachage.

La première idée a donc été hacher chaque attribut de classe, sauf le genre car il était estimé inutile :

```
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);
```

Pour obtenir les résultats suivants :

Test for 19DirectoryWithoutAVS

Insertion:

Insertion total: 10000

Insertion duration total time: 66195 microseconds

Insertion duration average time: 6.6195 microseconds

Distribution:

Number of buckets: 26267

Number of collisions: 1712 (17.12%)

Number of empty buckets: 17979 (68.4471%)

Largest bucket: 5

Smallest bucket: 0

Search:

Search total: 10000

Found total: 10000 (100%)

Search duration total time: 63198 microseconds

Search duration average time: 6.3198 microseconds

Ce qui n'est pas très concluant comparé aux résultats des fonctions de hachage de la partie 1.

Deuxième idée : utiliser la « recette standard » décrite dans le cours, et utilisée par la bibliothèque standard.

On a aussi remarqué que hasher un int était plus rapide que hasher un string. C'est pourquoi nous convertissons l'attribut gender en int (M = 1, F = 0) tout comme l'attribut date d'anniversaire (suppression des points).

Nous avons aussi testé la différence entre utiliser l'attribut gender dans la fonction de hachage ou ne pas l'utiliser. Résultat : sans l'attribut gender, la fonction est plus rapide mais il y a plus de collisions (comme on peut s'y attendre, vu qu'il y a moins « d'unicité ») et avec l'attribut gender, la fonction est légèrement plus lente mais il y a moins de collisions. Nous avons donc opté pour la solution avec l'attribut gender comme ci-dessous :

```
int genderInt = 0;
if (d.getGender() == "M") genderInt = 1;
unsigned int birthdayInt;
//convert string to long
try {
   //we should use std::stoi(std::string s) but unfortunately it's not available on MinGW gcc 4.8.1
   birthdayInt = std::atoi(d.getBirthday().c_str());
} catch(std::out_of_range& e) {
   std::cerr << "Not a valid int: " << d.getBirthday() << std::endl;
   birthdayInt = 0; //default value
}
size_t hashval = 17;
hashval = 31 * hashval + hash<string>()(d.getName());
hashval = 31 * hashval + hash<string>()(d.getFirstname());
hashval = 31 * hashval + hash<int>()(birthdayInt);
```

```
    hashval = 31 * hashval + hash<int>()(genderInt);
    return hashval;
```

Avec les résultats suivants :

Test for 19DirectoryWithoutAVS

Insertion:

Insertion total: 10000

Insertion duration total time: 16061 microseconds

Insertion duration average time: 1.6061 microseconds

Distribution:

Number of buckets: 26267

Number of collisions: 1711 (17.11%)

Number of empty buckets: 17978 (68.4433%)

Largest bucket: 5

Smallest bucket: 0

Search:

Search total: 10000

Found total: 10000 (100%)

Search duration total time: 11046 microseconds

Search duration average time: 1.1046 microseconds

Notre fonction de hachage nous donne des résultats plutôt proches de ceux des autres fonctions de hachage. Nous sommes donc satisfaits. Ils ne sont pas aussi performants car il y a plus d'attribut que juste le N°AVS, donc plus de calculs à faire. Et plus il y a d'attributs à gérer, plus il y a de possibilités d'optimisations.

Conclusion

Nous avons appris, par ce laboratoire, comment la fonction hash de la librairie std s'utilisait. Nous avons pu comprendre comment paramétrer cette fonction, et voir ainsi comment cela pouvait modifier son comportement.

On a pu aussi voir à quel point une fonction de hashage peut-être différente, et réaliser qu'il y a une infinité de fonctions de hachage différentes. Il existe donc pour chaque situation, contexte, et besoin, une fonction de hashage idéale, mais qui peut être loin d'être triviale.