ASD2 Laboratoire 4

## Partie 1

Pour cette partie, nous avons récolté les données qui sont affichés par le programme et nous les avons mis sous forme de graphe. Cette implémentation de hash a été faite en liste chainé avec un MAX\_LOAD\_FACTOR de 4, ce qui représente le MAX\_LOAD\_FACTOR moyen des listes chaîné. Voici les 2 graphes représentant respectivement le pourcentage de collisions et de buckets vides ainsi que les performances du temps d’insertion et de recherche associé pour chaque algorithme. Voici les correspondances des numéros d’algorithmes :

|  |  |
| --- | --- |
| Numéros d’algorithme | Algorithme |
| 1 | Int |
| 2 | Long |
| 3 | PolILj2EE |
| 4 | PolILj31EE |
| 5 | PolILj37EE |
| 6 | Stl |
| 7 | Sha256 |
| 8 | City |
| 9 | WithoutAVS |
| 10 | WithoutAVS (Modifié) |

Notons cependant que l’algorithme 10 ne sera pas pris en compte dans la partie 1 car il correspond à l’algorithme demandé dans la partie 2.

### Max Load Factor = 4

On voit ici que le nombre de collisions et de buckets est approximativement semblable partout sauf pour le 9ième algorithme qui est clairement derrière en terme de collisions (Un facteur d’environ 3 fois plus de collisions que les autres algorithmes). Cela s’explique par le fait que cet algorithme est un algorithme très simple (Uniquement le hash du nom) ce qui peut provoquer beaucoup de collision due au fait que le nom est similaire dans plusieurs entrées (Une famille aura tous le même nom de famille). En réalité, le nom n’est pas une clé primaire à un individu, autrement dit cela ne caractérise pas obligatoirement un individu en particulier.

La raison pour laquelle le nombre de buckets vides croit avec le nombre de collisions est que si on ajoute un élément qui existe déjà, cela créés une collision mais compte également comme une nouvelle paire clé-valeur alors l’algorithme le compte dans le nombre d’entrée alors qu’en réalité cela n’a pas augmenté le nombre de buckets utilisé.

Ici, on voit clairement que l’algorithme 7 est le moins performant car il met le plus de temps en moyen pour insérer et chercher les clés. Cet algorithme correspond au SHA-256, une fonction de hachage couramment utiliser aujourd’hui. Le fait que cette fonction soit autant lourde est dû à ses nombreuses opérations couteuses (Un prétraitement puis un condensé du hash). Malgès ceci, on voit qu’elle n’a pas un score de collisions moins élevé que les autres tout en ayant des performances bien moins bonnes. Son utilisation est dû au fait qu’elle est très sécurisé et impossible à reverse engineerer. Il est fortement déconseillé d’utiliser cette fonction ci pour le traitement des hash dans notre structure de données.

Les autres fonctions ont des performances un peu près similaires mais la fonction 3 se démarque pour son temps de recherche et la 5 pour son temps d’insertion.

### Max Load Factor = 2

Nous avons cette fois changé le MAX\_LOAD\_FACTOR à 2 et non plus à 4 afin de voir ce que cela modifié dans nos résultats. Voici donc le graphe des collisions et des buckets vides

On voit donc une augmentation claire du nombre de buckets vides avec un pourcentage de collisions moindre. Cela est dû au fait que les buckets deviennent moins grand mais qu’il y en a plus et donc ils sont moins souvent remplies car il y a bien plus de valeurs de clé possible pour chaque hash. Si nous représentons la liste chainée comme un tableau, cela voudrait dire que nous avons plus de ligne mais moins de colonnes possibles pour chaque ligne. L’algorithme 9 reste le moins bon en termes de collisions et de buckets vides pour les mêmes raisons que précédemment.

Nous pouvons voir maintenant

### Max Load Factor = 6

Nous avons cette fois augmenté le MAX\_LOAD\_FACTOR à 6, ce qui fait des buckets plus grands mais en quantité moindre. Voici les résultats obtenues.

Nous constatons ici une très clair hausse du pourcentage de collisions avec une extrême diminution des buckets vides. Cette fois, si nous représentons notre liste comme un tableau, nous avons alors moins de lignes mais beaucoup plus de colonnes par ligne. Cela implique donc qu’il y aura beaucoup moins de possibilité de clé pour chaque hash et donc la quasi-totalité des buckets seront plein. Mais cela veut aussi dire que beaucoup auront la même clé, et donc il y aura un risque de collisions élevé.

Les temps sont cette fois très similaire à ceux trouver pour le MAX\_LOAD\_FACTOR = 4, il n’y a donc pas grand-chose à en dire.

### Meilleur algorithme

En conclusion à cette partie 1, nous pouvons dire que l’algorithme 3 (PolILj2EE) est probablement le plus adapté pour notre structure de données tant que le MAX\_LOAD\_FACTOR est assez grand (>= 4). En effet, il a des pourcentages de collisions et de buckets vides pareil que tous touts en ayant des performances meilleurs que les autres, notamment sur la recherche.

## Partie 2

Pour cette partie, il est demandé de modifier l’algorithme 9 (DirectoryWithoutAVS) qui crée un hash à partir du nom d’une personne. Pour cette partie, nous n’avons pas le droit d’utiliser le numéro AVS afin de crée le hash.

Nous avons opté pour une solution simple et performante qui consiste tous simplement à faire un XOR logique entre le hash de la stdLib du nom, du prénom, de la date de naissance et du genre. En effet, si nous prenons toutes ces données, cela corresponds à 99.99% de chance à une personne unique et cela peut donc être une manière efficaces de crée un hash au tant en terme de collisions que de performances. Nous pouvons voir sur les graphes de la partie 1 que dans la quasi-totalité des tests effectués, notre algorithme arrive en tête en ayant en moyenne le même nombre de collisions que les autres mais ayant des performances accrues. Cela s’explique par le fait qu’un simple XOR est très rapide à faire pour un ordinateur et cela ne requiert pas d’opérations compliqué comme des compressions ou des modifications bit par bit. Les seules opérations compliquées sont celle qui sont nécessaire durant le hash définis par la std lib.

Il est cependant important à noter que cela n’est pas un hash sûr. La raison est que si nous possédons 3 hashs sur 4, le quatrième hash peut être trouvé en faisant un simple XOR entre les 3 hash XORé dont on dispose et le hash final. En d’autre terme, si on possède hashNom, hashPrenom et hashDate ainsi que hashFinal, alors hashGenre peut être trouvé en faisant :

hashGenre = (hashNom ^ hashPrenom ^ hashDate) ^ hashFinal

Et cela est valide pour toutes les informations. De plus, cette algorithme utilise la std lib, fonction de hashage n’étant absolument pas secret et qui peut être reverse par n’importe qui ayant des connaissances de base en sécurité cryptographique. Cependant, ceci n’étant pas un cours de sécurité cryptographique, nous avons jugé qu’il n’était pas nécessaire de s’assurer de la bonne sécurité des hash.