# 4 Les tables de hachage

Une table de hachage (ou tableau associatif, ou encore dictionnaire suivant les langages) est une structure de données composée de couples clé/valeur où chaque clé fait référence au plus à une valeur. Il s'agit d'un tableau non ordonné où l'accès à un élément se fait grâce à la clé qui lui est associée et qui est transformée en indice grâce à une fonction de hachage comme le montre la figure 12.

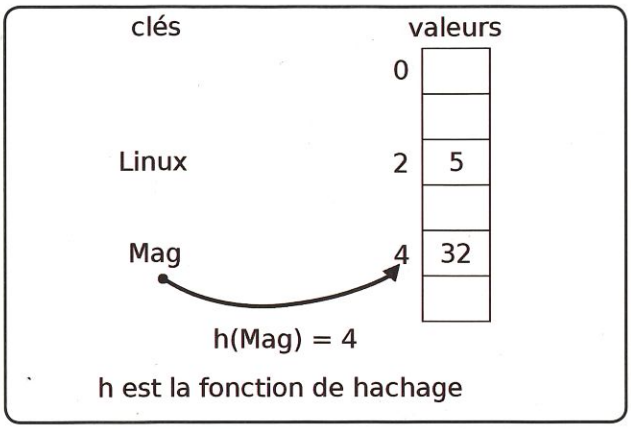


*Fig. 12 : Fonctionnement d'une table de hachage*

Le choix de la fonction de hachage est essentiel, car c'est elle qui permet de retrouver une valeur en fonction de sa clé et qu'il faut minimiser le plus possible les risques de collisions (deux clés distinctes pointent vers le même indice). En d'autres termes, si nous considérons que `h` est la fonction de hachage et que `(cle_1, valeur_1)` et `(cle_2, valeur_2)` sont deux couples de clé/valeur, une collision apparaît lorsque `h(cle_1) = h(cle_2)`. En général, il est malheureusement impossible d'éviter les collisions... Il faut donc être capable dans un premier temps de les minimiser puis, dans un second temps, de les traiter.

### Le paradoxe de la date anniversaire

Ce paradoxe dit que si au moins 23 personnes sont présentes dans une pièce, il y a de très fortes chances pour que deux d'entre elles aient le même jour et le même mois d'anniversaire. Si nous rapportons cela à une fonction de hachage, si nous avons 23 clés et une table de 365 cases, la probabilité pour que deux clés ne soient pas associées à une même case est de ... 0,4927. Les collisions ne peuvent pas être évitées, voilà pourquoi il faut les gérer !

## 4.1 Résolution des collisions

### 4.1.1 Résolution par chaînage

La solution la plus simple consiste à stocker les clés et les valeurs et, en cas de collision, de créer une liste chaînée (voir figure 13). Si nous notons `T` le tableau contenant

les données, **h** la fonction de hachage, l'algorithme permettant de rechercher un élément associé à une clé **cle_1** est le suivant :

```
i <- h(cle_1)
Si T[i] == nil Alors
    Erreur
Sinon
    tete <- T[i]
    Tant que tete <> nil Faire
        Si tete.cle == cle_1 Alors
            Valeur trouvée !
            Sortir
        Sinon
            tete = tete.suivant
    Erreur
```

Normalement, la majorité des éléments devraient être accessibles en temps constant... Mais c'est le pire des cas qu'il faut considérer et, dans ce cas, tous les éléments du dictionnaire occupent la même position et nous avons donc une complexité en O(n) pour accéder à un élément.

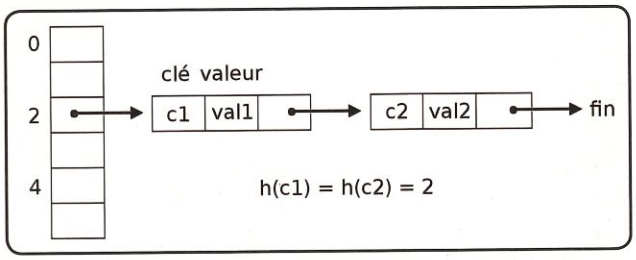


*Fig. 13 : Résolution des collusions dans une table de hachage par chaînage*

### 4.1.2 Résolution par adressage ouvert

Une autre solution permettant de résoudre les collisions est l'adressage ouvert : les informations associées à une même clé sont stockées de manière contiguë. L'idée est en fait d'avoir une fonction de hachage qui ne va non plus déterminer la position d'une valeur, mais plutôt d'une zone dans laquelle la référence doit pouvoir être trouvée comme le montre la figure 14. Cette fois, l'algorithme de recherche est le suivant :

```
i <- h(cle_1)
Tant que T[i] <> nil Alors
    Si T[i].cle == cle_1 Alors
            Valeur trouvée !
            Sortir
    Sinon
        i <- i + 1
Erreur
```

La recherche se fait encore une fois en temps linéaire (O(n)).

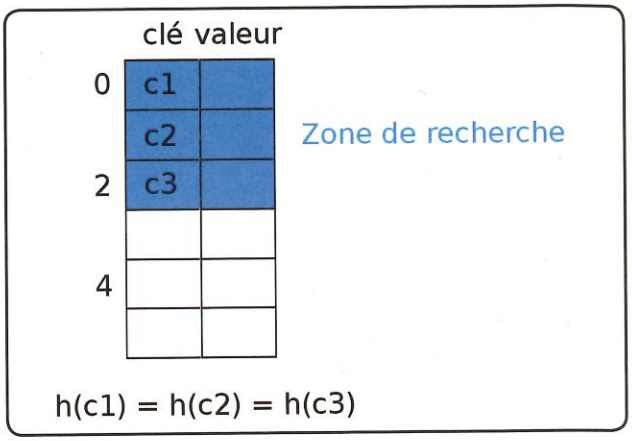


*Fig. 14 : Résolution des collusions dans une table de hachage par adressage ouvert*

## 4.2 La fonction de hachage

La fonction de hachage est au cœur du mécanisme des tables de hachage, elle est fondamentale et il faut donc la choisir avec rigueur. Sa mission est, en fonction d'une clé numérique ou alpha-numérique, de retourner un indice numérique avec le minimum de collisions possible. Voici les deux exemples les plus simples de fonctions de hachage s'appliquant à une résolution des collisions par chaînage.

### 4.2.1 Méthode par division

Soit **c** une clé et **m** la taille du tableau **T** permettant de stocker les données. **h** est une fonction de hachage telle que **0 <= h(c) < m** (de manière à ce que l'indice puisse être associé à **T**).

La méthode par division est particulièrement simple ; elle consiste à définir **h** telle que **h(c) = c mod m**.

Par exemple, pour une clé numérique valant 12 et un tableau de 5 éléments, **h(12) = 12 mod 5 = 2** : la valeur associée à la clé 12 sera stockée dans T[2].

### 4.2.2 Méthode par multiplication

La méthode par multiplication est plus difficile à appréhender, car il faut travailler avec des fractions plutôt qu'avec des entiers (mais nous ramenons bien sûr le résultat à une valeur entière). En posant **A** un nombre réel tel que **0 < A < 1**, la fonction de hachage est **h(c) = [ m(Ac - [Ac]) ]** où **[ ]** désigne la partie entière... Le problème de ne pas pouvoir utiliser LaTeX pour écrire

les articles fait que certains symboles mathématiques ne sont pas accessibles, comme par exemple la notation anglo-saxonne de la partie entière où la fermeture des crochets n'est présente qu'en haut ou en bas et qui permet de préciser l'arrondi par défaut ou par excès (ici c'est un arrondi par défaut). Cette écriture peut être simplifiée en `h(c) = [ m x { Ac } ]` où `{ }` représente la partie fractionnaire.

Donald Knuth **[2]**, professeur émérite en informatique de l'université de Standford, pionnier de l'algorithmique, créateur de TeX et auteur de la série d'ouvrages « The Art of Computer Programming », recommande de prendre pour valeur de A ce qu'il appelle le ratio d'or : `A = (√5 - 1) / 2 ≈ 0.618`.

En reprenant l'exemple précédent, pour une clé numérique valant 12 et un tableau de 5 éléments, `h(12) = [ 5 x { 0.6 x 12 } ] = [ 5 x { 7.2 } ] = [ 5 x 0.2 ] = 1` : la valeur associée à la clé 12 sera stockée dans T[1].

## Conclusion

Nous sommes revenus dans cet article sur des structures que l'on utilise tous les jours, mais dont on ignore parfois le fonctionnement lorsque l'on utilise des bibliothèques toutes prêtes. Si à un moment donné vous souhaitez optimiser votre code, il est essentiel de comprendre leur fonctionnement interne... Parfois, une structure non adaptée peut faire perdre beaucoup de temps de calcul...

Si cet article vous a ouvert l'appétit et que vous souhaitez obtenir plus d'informations sur le sujet, je vous recommande la lecture (même si elle est ardue) des ouvrages de Donald Knuth **[2][3]** dans lesquels vous trouverez de nombreuses explications. ■

## Références


[1] Bachmann P., « Die analytische Zahlentheorie », Leipzig B. G. Teubner, 1894. (peut être consulté sur https://archive.org/details/dieanalytischeza00bachuoft)

[2] Knuth D., « The Art of Computer Programming - Volume 1 Fundamental algorithms », Addison Wesley, 1998.

[3] Knuth D., « The Art of Computer Programming - Volume 3 Sorting and searching », Addison Wesley, 1998.