Séquence 12 - Hachage

Table des matières

I. Fonction de hachage

Définition 1 (Fonction de hachage)

Une fonction de hachage h est une application déterministe qui, à toute clé k issue d'un ensemble $\mathcal{K}$, associe un entier dans un certain intervalle d'entiers de taille m $[\![0, m-1]\!]$ où $m \in \mathbb{N}$ est fixé.

$$\begin{array}{rccc} h : & \mathcal{K} & \longrightarrow & [\![0, m-1]\!] \\ & k & \mapsto & h(k) \end{array}$$

Remarque (Fonctions de hachage dites « aléatoires »). Une fonction de hachage doit être déterministe, ce qui signifie que pour une clé d'entrée donnée, elle doit toujours générer la même empreinte. Cette exigence exclut les fonctions de hachage qui dépendent de paramètres variables externes, tels que les générateurs de nombres pseudo-aléatoires ou l'heure de la journée. Il exclut également les fonctions qui dépendent de l'adresse mémoire de la clé hachée dans les cas où l'adresse peut changer pendant l'exécution (comme cela peut arriver sur les systèmes qui utilisent certaines méthodes de récupération de place).
Le déterminisme s'entend dans le cadre de la réutilisation de la fonction. Il existe des fonctions de hachage de type aléatoire, utilisant un système de germe (*seeded hash*), qui sont utilisées de le cadre de l'authentification Web par exemple. Certaines d'entre elles sont implémentées en Python ou en OCaml. Mais si les valeurs sont persistantes (par exemple, écrites sur le disque), elles ne peuvent plus être traitées comme des valeurs de hachage valides, car lors de la prochaine exécution, la valeur aléatoire peut différer. Ainsi, ces fonctions de hachage sont déterministes pour une session mais ne le sera pas sur plusieurs exécutions.

Définition 2 (Empreinte d'une clé par une fonction de hachage)

L'image $h(k)$ d'une clé k par h est appelée **empreinte** associée à la clé k. On parle aussi parfois de **haché** ou de **condensat** ou **digestat** (*message digest*) [a]

a. On retrouve la métaphore culinaire/digestive : la clé a été... hachée et digérée par la fonction de hachage !

Une fonction de hachage h est, a priori :

non injective : plusieurs valeurs peuvent avoir la même empreinte, on parle alors de **collision**

non surjective : certaines empreintes ne sont jamais atteintes, c'est-à-dire qu'aucune clé k ne sera associée à cette empreinte.

I. 1. Fonctions de hachage injectives (mappings) et tableaux associatifs

Pour certains types de clés, il existe des fonctions de hachage à la fois naturelles et **injectives**, qu'il est important de connaître. On parle alors plutôt de *mapping*.

Définition 3 (Mapping)

On appelle **mapping** une fonction de hachage **injective**, c'est à dire une fonction de hachage pour laquelle **aucune collision** ne peut avoir lieu : deux clés différentes ont toujours deux empreintes différentes.

Exemple 1 (Mapping des couples d'entiers)

Vous avez déjà vu le mapping très classique qui fait correspondre une valeur entière à un couple d'entiers :

$$\begin{array}{rccc} h_1 : & [\![0,\,(m_1-1)]\!] \times [\![0,\,(m_2-1)]\!] \subseteq \mathbb{N}\times\mathbb{N} & \longrightarrow & [\![0,\,m_1\times m_2-1]\!] \\ & (i,j) & \mapsto & h_1(i,j) = j\times m_1 + i \end{array}$$

Ce mapping n'est pas seulement injectif, il est même bijectif.
Ce mapping a été utilisé pour **linéariser des tableaux bidimensionnels**, par exemple pour le stockage des matrices. Plutôt que d'utiliser un tableau bidimensionnel, les coefficients $m_{i,j} \in \mathbb{R}$ d'une matrice $M \in \mathbb{R}^{m_1\times m_2}$ peuvent être stockés dans un tableau uni-dimensionnel t en utilisant le mapping h. [a]

a. Même lorsque vous utilisez des tableaux bidimensionnels pour faire cela, le compilateur tente d'optimiser votre code en effectuant ce type de linéarisation.

Exemple 2 (Mapping des p-uplets d'entiers)

On peut bien sûr généraliser ce mapping aux p-uplets d'entiers :

$$\begin{array}{rccc} h_p : & [\![0,\,(m_1-1)]\!] \times \ldots \times [\![0,\,(m_p-1)]\!] \subseteq \mathbb{N}^p & \longrightarrow & [\![0,\,m_1\times\ldots\times m_p-1]\!] \\ & (i_1,\ldots,i_p) & \mapsto & h_p(i_1,\ldots,i_p) = i_1 + m_1\times(i_2 + m_2\times(\ldots i_{p-1} + m_p\times \end{array}$$

Exemple 3 (Mapping des caractères ASCII)

Un autre mapping très naturel est celui basé sur le format ASCII. Celui-ci définit, de fait, une injection entre l'ensemble des caractères simples encodés sur 8 bits (table ASCII étendue, comprenant, en plus des caractères de base, quelques caractères accentués dont ceux du français) et celui des entiers naturels inférieurs ou égaux à 255.

$$\begin{array}{rccc} h_c : & \mathcal{C}_{\text{ASCII}} & \longrightarrow & [\![0,\,255]\!] \\ & c & \mapsto & h_c(c) = \text{code\_ascii}(c) \end{array}$$

Par exemple, le caractère `e` est associé à la série de 8 bits `0110 0101` qui s'écrit 65 en hexadécimal et est associée à l'entier :

$$6\times 16 + 5 = 96 + 5 = 101 \in [\![0,\,255]\!]$$

La table ASCII est disponible, par exemple, ici : `https://www.commentcamarche.net/informatique/technologies/1589-code-ascii/`

Ce mapping ASCII est une fonction discrète qui a été définie à la main, caractère par caractère, par l'intermédiaire d'une convention adoptée au niveau international.

Exemple 4 (Mapping des chaînes de caractères de taille fixée)

On peut bien sûr composer les deux mappings h_c et h_{p} pour associer, à toute chaîne de caractères ayant exactement p caractères, un entier :

$$\begin{array}{rccl} h_{p,\text{c}}: & \mathcal{C}^p & \longrightarrow & [\![0,\, 255^p - 1]\!] \\ & c_1 c_2 \dots c_p & \mapsto & h_{p,\text{c}}(c_1 c_2 \dots c_p) = h_p(h_c(c_1),\, h_c(c_2), \dots,\, h_c(c_p)) \end{array}$$

La composition de deux fonction injectives étant injective, la fonction de hachage obtenue est bien injective : il s'agit bien d'un mapping.
Par exemple, pour $p = 7$, la chaîne de caractères `jillian` est associée à l'entier :

$$h_{7,\text{c}}(''\texttt{jillian}'') = h_c(\texttt{'j'}) + 255 \times (h_c(\texttt{'i'}) + 255 \times (h_c(\texttt{'l'}) + 255 \times (h_c(\texttt{'l'}) + 255 \times (h_c(\texttt{'i'}) + 255 \times (h_c(\texttt{'a'}) +$$

$$h_{7,\text{c}}(''\texttt{jillian}'') = 106+255\times(105+255\times(108+255\times(108+255\times(97+255\times110)))) = 1203962737000$$

Remarque (Dépassement mémoire (overflow) lors du calcul d'une empreinte). Cela fonctionne... mais l'entier obtenu est hors du champ de représentation des entiers non signé 32 bits `uint32_t`, puisque l'entier non signé maximal représentable sur $N = 32$ bits est $2^N - 1 = 2^{32} - 1 \approx 2^2 \times (2^10)^3 \approx 4 \times (10^3)^3 \approx 4$ milliards. Par contre, il sera représentable sur un entier 64 bits, noté `uint64_t` car $2^{64} - 1 = 1.844674407 \times 10^{19}$.
Cependant, pour un nombre plus important de caractères, même les entiers 64 bits ne suffiront plus.
Nous retrouverons cette problématique de la représentation machine pour la plupart des fonctions des mapping et des fonctions de hachage générales que nous rencontrerons.

I. 2. Tableaux associatifs

Cette correspondance injective objet↔indice permet de créer des tableaux associatifs.

Définition 4 (Tableau associatif)

Soit un ensemble de clés $\mathcal{K}$ (de type quelconque) que l'on a réussi à étiqueter par des entiers $[\![0,\, m-1]\!]$ grâce à un fonction de hachage injective (mapping) h.
Un tableau associatif est un tableau dont la case d'indice $i = h(k)$ stocke l'information relative au couple clé-valeur (k, v).
Lorsqu'une telle fonction de hachage injective existe, les tableaux associatifs permettent une **implémentation concrète très efficace de la structure de données abstraite de dictionnaire**, pour laquelle chaque clé k est associée à une unique valeur v.

Exemple 5

Soit un texte donné. On pourrait utiliser le mapping h_{c} pour créer un dictionnaire qui, à chaque caractère ASCII, associe le nombre d'occurrences de ce caractère dans le texte.

Exemple 6

On pourrait utiliser le mapping $h_{7,c}$ pour mapper des plaques d'immatriculation modernes du type `CK995RE` à 7 caractères et créer un dictionnaire, qui, à chaque immatriculation associé l'entrée constituée de la clé immatriculation et d'un pointeur vers un structure de données contenant les informations relatives au véhicule.

Étudions la complexité temporelle et spatiale de la structure de données de dictionnaire implémentée avec un tableau associatif (quand l'implémentation d'une telle structure est possible !) :

Concernant la complexité temporelle, pour une clé k, une fois calculée l'empreinte $h(k)$ associée à cette clé, l'accès à la donnée (k, v) se fait en temps constant, grâce à l'adressage direct inhérent à la contiguïté mémoire des tableaux. C'est beaucoup plus rapide que l'implémentation par ABR par exemple

Concernant la complexité spatiale, si la fonction de hachage n'est pas surjective, certaines cases du tableau sont inutiles et on a un gaspillage de mémoire. Mais le gain en termes de complexité temporelle est généralement suffisamment convaincant pour accepter ce gaspillage.

Remarque. Autre avantage des tableaux associatifs, aucune information supplémentaire sur l'ensemble des clés. Par exemple, on n'a pas besoin de connaître ou de définir un ordre total sur $\mathcal{K}$ comme pour l'implémentation d'un dictionnaire avec un ABR.

I. 3. Fonctions de hachage non injectives

Nous avons vu jusqu'ici des exemples où tout se passe pour le mieux, où il semble exister des manières naturelles d'associer une empreinte à une clé (clés de type p-uplet, caractère, chaîne de caractères de taille fixe). Mais, dans la plupart des cas, un tel mapping n'existe tout simplement pas.

En effet, l'intervalle des empreintes est réduit à une plage de valeurs entières. Cette contrainte vient directement des contraintes machines : on ne peut pas représenter des nombres arbitrairement grands avec un ordinateur. Sur une machine moderne, où l'on peut envisager d'utiliser des entiers non signés 64 bits, par exemple des `uint64_t` en C, la plage des empreintes sera au maximum de taille $m = 2^{64} = 1.8446744 \times 10^{19}$ De plus, nous verrons que ces empreintes seront utilisées comme indice dans des tableaux et il est impossible de stocker un tableau de longueur infinie.

Ainsi, l'ensemble d'arriver de la fonction de hachage est nécessairement fini ! Pour qu'une fonction de hachage soit injective, il faudrait donc que l'ensemble des clés que l'on cherche à

mettre en bijection avec un intervalle d'entiers soit également fini, ce qui est très restrictif.

Exemple 7 (Cas où il n'existe pas de mapping)

Par exemple, il n'existe pas de fonction de hachage injective permettant de mapper l'ensemble des **chaînes de caractères de taille quelconque** $\mathcal{S}$ sur une plage d'entiers du type $[\![0,\, m-1]\!]$ car $\mathcal{S}$ est infini !
Il faut donc abandonner tout espoir d'implémenter un dictionnaire dont les clés sont des chaînes de caractères de taille quelconque avec un tableau associatif !
Mais il y a d'autres moyens de procéder : rappelez vous, nous en avons déjà vu une ! [a]

a. Réponse : dictionnaire avec des clés de type chaîne de caractères de taille quelconque, implémenté avec un ABR en utilisant l'ordre lexicographique sur les clés pour ordonner le couples clé-valeur dans l'ABR

Ainsi, **dès que l'on manipule des ensembles infinis de clés, il n'existe pas de fonction de hachage injective**, mais il est toutefois possible de **construire des fonctions de hachage en essayant le plus possible de se rapprocher d'une forme d'injectivité, c'est-à-dire en cherchant à limiter le nombre de collisions.**

Définition 5 (Collision)

On dit qu'il y a une collision lorsque deux clés k_1 et k_2 donnent la même empreinte par la fonction de hachage h :

$$h(k_1) = h(k_2)$$

Une fonction de hachage performante sera conçue pour limiter au maximum les collisions.

I. 4. Quelques exemples de fonctions de hachage

Il existe de très nombreuses fonctions de hachage. Le choix de la fonction de hachage dépend du degré de sécurité, de performance et de rapidité de calcul souhaité, mais aussi d'éventuelles connaissances a priori sur les clés qui vont êtres hachées par cette fonction. Bref, ce choix doit être adapté à chaque application.
On présente ci-dessous des fonctions de hachage possible sur des clé de type chaîne de caractères de taille quelconque $\mathcal{S}$.

I. 4. a. Une première fonction de hachage naïve

Une première fonction de hachage naïve consisterait à utiliser uniquement le code ASCII du premier caractère de la chaîne comme empreinte :

$$\begin{array}{rccc} h : & \mathcal{S} & \longrightarrow & [\![0,\, 255]\!] \\ & c_1 \dots c_r & \mapsto & h_c(c_1) \end{array}$$

Bien sûr, cette fonction de hachage crée beaucoup de collisions !

I. 4. b. Fonction de hachage sommative

Une autre possibilité est de sommer les codes ASCII et d'effectuer une opération de modulo pour ramener le résultat dans la plage d'entiers souhaitée $[\![0,\, p-1]\!]$ avec p premier grand.

Attention (Opérateur de congruence modulo). Attention toutefois, l'opérateur modulo, s'il est effectué de nombreuses fois sur de gros entiers, peut devenir coûteux car il est généralement microprogrammé. Il faut donc bien réfléchir à son utilisation (comme celle des multiplications ou des divisions, qui sont en lien avec le nombre de bits des entiers manipulés)

$$\begin{array}{rccl} h: & \mathcal{S} & \longrightarrow & [\![0,\, p-1]\!] \\ & c_0 \dots c_{r-1} & \mapsto & \displaystyle\sum_{j=0}^{r-1} \text{code\_ascii}(c_j) \bmod p \end{array}$$

Cette fonction de hachage est assez peu coûteuse à calculer, et elle possède une propriété de **déroulement** intéressante. Elle n'est cependant pas très performante pour éviter les collisions car elle ne tient pas compte de l'ordre de lettre : deux mots ayant les mêmes lettres mais dans des ordres différents auront la même empreinte. Il est donc très facile de trouver des collisions.

I. 4. c. Fonction de hachage dite de conversion de base

Une fonction de hachage plus utile et performante en pratique est la fonction de hachage dite de conversion de base suivante :

$$\begin{array}{rccl} h: & \mathcal{S} & \longrightarrow & [\![0,\, p-1]\!] \\ & c_0 \dots c_{r-1} & \mapsto & \displaystyle\sum_{j=0}^{r-1} \text{code\_ascii}(c_j) \times B^j \bmod p \end{array}$$

où B est un entier. En toute rigueur, comme il y a 256 caractères, on a envie de choisir $B = 256$ pour faire le lien entre l'écriture $c_1 \dots c_r$ et l'écriture en base $B = 256$ d'un entier, avec 256 symboles. On a alors la propriété d'unicité de l'écriture en base 256 qui garantit l'injectivité...

Bien entendu, au final, l'injectivité est une cause perdue pour les fonctions de hachage sur $\mathcal{S}$ puisque l'on cherche à faire rentrer un ensemble infini dans un intervalle d'entier, ce que nous rappelle la présence de l'opérateur de congruence `mod`. Il n'est ainsi pas totalement obligatoire de choisir $B = 256$, puisque, de toute façon, l'injectivité sera perdue. Il faudra donc voir au cas par cas : si on souhaite vraiment s'approcher le plus possible de l'injectivité, on gardera $B = 256$, et on prendra p premier le plus grand possible. Mais attention, on va très vite manipuler de très grands entiers et créer des dépassements de capacité, y compris avec des entiers 64 bits. Il faudra donc calculer procéder astucieusement en utilisant les propriétés du calcul modulaire ! ! (non aborderons un peu cela en TP). Sinon, on prendra une valeur de base B inférieure.

Cette fonction de hachage a de très bonnes performances pour réduire les collisions. Si p est un nombre premier, la somme correspond à l'évaluation en B d'un polynômes dont les coefficients appartiennent à un corps (au corps fini $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$). La fonction n'est pas injective, mais si deux chaînes de caractères $c_1 \dots c_r$ et $c'_1 \dots c'_{r'}$ avec $r' > r$ par exemple ont la même empreinte, cela signifie que :

$$\sum_{j=0}^{r-1} \text{code\_ascii}(c_j) \times B^j = j = 0^{r'-1} \text{code\_ascii}(c'_j) \times B^j \bmod p$$

et donc que

$$\sum_{j=0}^{r'-1} (\text{code\_ascii}(c_j) - \text{code\_ascii}(c'_j)) \times B^j = 0 \bmod p$$

en posant $\text{code\_ascii}(c_j) = 0$ pour $j \in [\![r+1,\, r']\!]$.

Le polynôme $P(X) = \sum_{j=0}^{r'-1} \big((\text{code\_ascii}(c_j) - \text{code\_ascii}(c'_j)) \bmod p\big) \times X^j$ est donc un polynôme à coefficients dans le corps $\mathbb{F}_p$ et B est donc telle que $P(B) = 0$, il s'agit donc d'une racine de ce polynôme dans $\mathbb{F}_p$. Or, ce polynôme, dont les coefficients sont bien à valeurs dans un corps, a au plus $r'-1$ racines. Donc B a $\dfrac{r'-1}{p}$ chances d'être une racine, ce qui signifie qu'une collision a $\dfrac{r'-1}{p}$ chances de se produire, ce qui sera très très faible si p est très grand par rapport à la taille des chaînes de caractères hachées.

En plus de posséder de bonnes propriétés d'injectivité pour des choix judicieux de B et de m, cette fonction **se calcule facilement, avec une complexité temporelle linéaire par une méthode de Horner**.

Enfin, dans sa version renversée :

$$\begin{array}{rccl} h: & \mathcal{S} & \longrightarrow & [\![0,\, p-1]\!] \\ & c_0 \dots c_{r-1} & \mapsto & \displaystyle\sum_{j=0}^{r-1} \text{code\_ascii}(c_j) \times B^{r-1-j} \bmod p \end{array}$$

elle possède une propriété de **fonction déroulante** (démontrez-là, c'est facile) qui peut être utile dans certains algorithmes, comme l'algorithme de Rabin-Karp :

$$h(c_{i+1} \dots c_{i+p}) = \big(b \times (h(c_i \dots c_{i+p-1}) - \text{code\_ascii}(c_i) \times b^{p-1}) + \text{code\_ascii}(c_{i+p})\big) \bmod m$$

I. 4. d. Fonctions de hachage en Python

Il existe une fonction de hachage générique `hash` pour un certain nombre de types natifs en Python, notamment les types numériques et les chaînes de caractères. Une valeur de -1 signifie une erreur, et si la fonction de hachage doit théoriquement renvoyer -1, elle renvoie -2 à la place.

Pour hacher des clés entières. Pour une clé $k \in \mathbb{Z}$:

- si $k \geq 0$, $\texttt{hash}(k)$ renvoie $k \bmod m$, où $m = 2^{61} - 1$ (qui est premier).
- sinon, $\texttt{hash}(k)$ renvoie $\texttt{hash}(-k)$

Pour hacher des clés flottantes. Pour une clé $k \in \mathbb{Q}$ avec $k = \dfrac{p}{q}$: (donc pour un flottant puisque les flottants sont des nombres réels avec un nombre fini de chiffres après la virgule), la valeur renvoyée est $\texttt{hash}(p) \times \texttt{hash}(q)^{-1}$, où x^{-1} désigne l'inverse de x dans $\mathbb{Z}/m\mathbb{Z}$ (inverse modulaire). Ceci permet que la fonction de hachage renvoie la même empreinte pour un entier qu'il soit vu comme entier ou flottant, et le calcul est de plus faisable efficacement. Le code détaillé est en *open source* et disponible ici :

`https://github.com/python/cpython/blob/main/Python/pyhash.c`

Pour hacher des clés de type chaînes de caractères. Pour les chaînes de caractères :

— une variante de fonction `FNV` est implémentée, qui utilise notamment une graine aléatoire qui rendra les empreintes différentes d'une exécution à l'autre,

— ainsi qu'une autre fonction appelée `SipHash`, offrant de meilleure garantie de sécurité (afin d'éviter des attaques liées à l'utilisation de collisions).

Le code source est disponible ici :

`https://github.com/python/cpython/blob/main/Python/pyhash.c`

Voici également une ressource qui discute de la sécurité de ces fonctions de hachage :

`https://peps.python.org/pep-0456/#current-implementation-with-modified-fnv`

I. 4. e. Fonctions de hachage utilisées pour la sécurité

Les algorithmes de hachage les plus couramment utilisés agissent directement sur des séries de bits de taille quelconque, ce qui les rend très polyvalents. Les plus connus sont :

MD5 : (*MD* signifiant *Message Digest*, un digesteur de message !). Développé par Rivest en 1991, MD5 crée une empreinte digitale de 128 bits (16 octets) à partir d'un texte de taille arbitraire en le traitant par blocs de 512 bits. `http://fr.wikipedia.org/wiki/MD5`

SHA 0, 1 ou 2 : urlhttp ://fr.wikipedia.org/wiki/SHA-2 (pour *Secure Hash Algorithm*) crée des empreintes d'une longueur de 160 bits. SHA-1 est une version améliorée de SHA datant de 1994 et produisant une empreinte de 160 bits à partir d'un message d'une longueur maximale de 264 bits en le traitant par blocs de 512 bits.

Whirlpool : `http://www.seas.gwu.edu/~poorvi/Classes/CS381_2007/Whirlpool.pdf` (64 octets / 512 bits)

De manière générale, les empreintes d'une série de bits - qui sont considérés comme les clés dans ce contexte - par ce type de fonctions de hachage peuvent être utilisées comme une sorte d'empreinte digitale (en anglais *finger print*), comme une signature, pour identifier un utilisateur ou bien pour valider l'intégrité ou l'authenticité de données reçues.

Voici quelques exemples classiques d'utilisation de fonctions de hachage.

Exemple 8 (Vérification de l'intégrité de données téléchargées)

Par exemple, lorsque vous téléchargez certaines données volumineuses, comme une image ISO, une clé MD5 est généralement fournie. Il s'agit de l'empreinte obtenue par la fonction de hachage MD5 sur les données mise à disposition en téléchargement (données numériques vues comme une série de bits). Après téléchargement, pour vérifier l'intégrité des données récupérées, et repérer d'éventuelles corruptions des données pendant le transfert, vous êtes invité à calculer l'empreinte MD5 de la série de bits récupérées et enregistrées sur votre machine, par exemple en utilisant la commande Linux `md5sum` [a]. Normalement, la valeur de l'empreinte calculée doit être la même que celle fournie par le site de téléchargement. Si les valeurs diffèrent, vous êtes certains qu'il y a eu un souci lors du téléchargement. Par contre, si vous êtes extrêmement malchanceux, il peut arriver que l'empreinte soit bien celle attendue mais qu'une corruption soit quand même présente, car cette fonction de hachage n'est pas totalement injective et il peut exister des collisions.
En 2004, des chercheurs chinois ont montré qu'il était possible, pour une empreinte donnée, de trouver une série de bits différentes donnant la même empreinte... ce qui pourrait être utilisé par des hackers pour introduire des éléments malveillants dans un fichier exécutable téléchargé sur Internet, sans être repérable par la vérification MD5.

a. https ://en.wikipedia.org/wiki/Md5sum

Exemple 9 (Utilisation en cryptographie (obsolète))

Sur le même principe, les tous premiers emplois du hachage ont été une pseudo cryptographie à sens unique destinée à vérifier la justesse d'une information connue.
Un message confidentiel, considéré comme une série de bits, sont hachées par une fonction de hachage. L'empreinte obtenue est donnée au destinataire du message. Lorsqu'un message lui remet, par des voies sécurisées le message, le destinataire hache le message reçu avec la même fonction de hachage. Si l'empreinte obtenue est la même que celle espérée, alors il pourra considérer que les données reçues sont les bonnes, ce qui, du fait des collisions potentielles, n'est pas vrai à 100% mais est presque vrai au sens statistique. Ces méthodes sont délaissées depuis les années 2000 car l'assouplissement des législations autour de l'utilisation de la cryptographie, ainsi que plusieurs travaux ont montré que ce type de système de cryptage n'était plus fiable. Elles sont toujours utilisées, mais conjuguées à d'autres méthodes de cryptographie.

II. Tables de hachage

II. 1. Structure de données abstraite de table de hachage

Lorsque h n'est pas injective, on ne peut pas, à proprement parler, réaliser de tableau associatif, car il existe des collisions : une même empreinte i peut correspondre à deux clés k_1 et k_2 différentes : $i = h(k_1) = h(k_2)$.
Il nous est donc impossible d'utiliser cette empreinte i comme indice pour stocker les entrées (k_1, v_1) et (k_2, v_2) dans un tableau associatif, car cela reviendrait à stocker ces deux informations dans une même case, ce qui est impossible.
Par contre, il est possible d'associer à la case i d'un tableau la **liste** des entrées de type clé-valeur (k_j, v_j) telles que $h(k_j) = i$. On appelle une telle structure de données abstraite une table de hachage.

Définition 6 (Table de hachage (*hash table*))

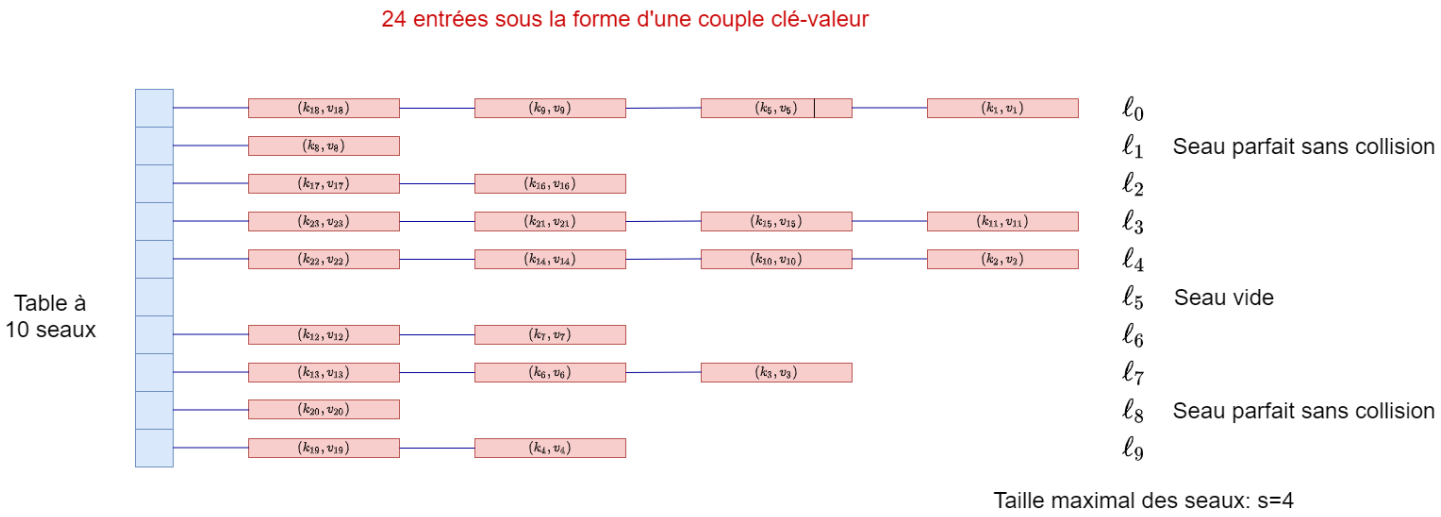
On se donne un ensemble $\mathcal{K}$ de clés et une fonction de hachage $h : \mathcal{K} \to [\![0, m-1]\!]$ permettant d'associer, à chaque clé $k \in \mathcal{K}$ un entier compris entre 0 et $m-1$, pas forcément de manière injective.
Une table de hachage est une structure de données abstraite permettant de ranger des entrées de type clé-valeur en regroupant ensemble, par exemple dans une liste, les entrées dont les clés ont la même empreinte par h.

Définition 7 (Entrée)

Pour désigner une **donnée de type clé-valeur** à ranger dans une structure de données, on parle indifféremment d'**entrée** (*entry*), de **couple clé-valeur** (*key-value pair*), d'**enregistrement** (*record*), de **liaison** (*binding*).

Définition 8 (Seau (*bucket*))

La structure de données ℓ_i regroupant toutes les entrées ayant la même empreinte i est appelée **seau** (*bucket* en anglais).
Si la fonction de hachage a pour ensemble d'arrivée l'intervalle $[\![0, m-1]\!]$, de taille m, la table de hachage comportera m seaux : $(\ell_i)_{i \in [\![0, m-1]\!]}$.



Remarque. On peut voir les tables de hachage comme une généralisation du principe du tableau associatif lorsque l'association objet→indice par la fonction de hachage n'est pas injective.

Les tables de hachage sont utilisées principalement pour la recherche rapide d'une donnée v à partir de sa clé k.
Les primitives de manipulation de la structure de données abstraite de table de hachage, que nous appellerons `hshtbl`, sont les suivantes :

`hashtbl_create :` constructeur, alloue une structure de données vide

`hashtbl_free :` destructeur, libère l'espace mémoire alloué à la structure de données

`hashtbl_find :` accesseur, permet de savoir si une entrée (k, v) est présente dans la structure de données à partir de sa clé k. Elle utilise l'empreinte $h(k)$ de la clé pour localiser la clé dans la table. Cette fonction renvoie 2 informations : un booléen indiquant si l'entrée a été trouvée et la valeur v associée à la clé k, si cette dernière a été trouvée.

`hashtbl_add :` ajout d'une entrée (k, v) dans la table, c'est-à-dire calcul de de l'empreinte $h(k)$ de sa clé k puis ajout de l'entrée (k, v) dans le seau d'indice $h(k)$.

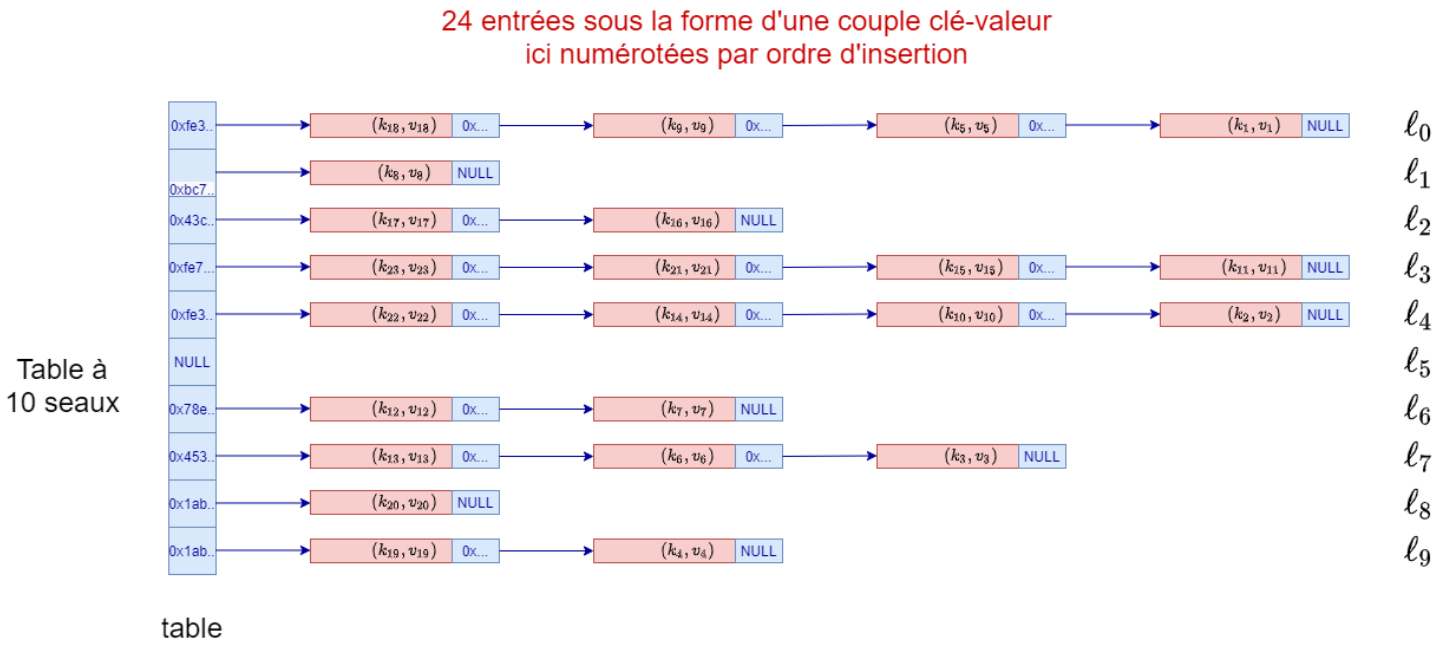
`hashtbl_remove :` suppression d'une entrée (k, v) dans la table, c'est-à-dire calcul de l'empreinte $h(k)$ de sa clé puis suppression de l'entrée dans le seau d'indice $h(k)$ correspondant

`hashtbl_replace :` remplacement de la valeur v d'une entrée (k, v) dans la table, c'est-à-dire calcul de l'empreinte $h(k)$ de sa clé puis, si l'entrée est présente dans le seau $h(k)$, modification de sa valeur v par une nouvelle valeur. Si l'entrée n'est pas présente, une nouvelle entrée (k, v) est ajoutée dans le seau d'indice $h(k)$.

La structure de données abstraite de table de hachage est généralement implémentée comme une structure de données **mutable**, c'est-à-dire modifiable en place.

II. 2. Implémentation concrète en C avec des tableaux et des maillons chaînés

Voici un exemple d'implémentation concrète où les seaux sont des piles implémentées par maillons chaînés :



On aurait aussi pu utiliser des piles implémentées par tableaux ou tableaux redimensionnables : sur le dessin, on ajouterait les entrées en fin de seau au lieu de les ajouter en début de seau comme on l'a fait pour des piles implémentées par maillons chaînés [1]
Voici la définition du type C correspondant :

```c
struct hashtbl
{
  unsigned int n_buckets; // nombre de seaux. Ici, on stocke le nombre de seaux sur un unsigned int
  // donc on aura au max 2^{32}-1 = environ 4 milliards de seaux
  // en dessous de 10 milliards d'entrées à ranger dans la table, on est à l'aise
  list **tab; // tableau de pointeurs vers des listes de maillons chaînés utilisées comme des piles pour représenter
              // les seaux
};

typedef struct hashtbl hashtbl;
```

L'implémentation des primitives de manipulation sera réalisée en TP.

II. 3. Implémentation concrète en OCaml : module `Hashtbl`

OCaml propose un module `Hashtbl` implémentant une structure de données de table de hachage à travers le type polymorphe `('a, 'b) Hashtbl.t`. Le polymorphisme est double, sur les clés et sur les valeurs : `'a` est le type des clés, `'b` celui des valeurs associées.

`https://v2.ocaml.org/api/Hashtbl.html`

Ce module est cité dans le programme de MPII comme *devant être reconnu et utilisable après rappel*. Il utilise une fonction de hachage générique nommée `Hashtbl.hash` pour construire les tables de hachage. Aucune documentation sur cette fonction de hachage n'est facilement disponible : cette fonction reste cachée à l'utilisateur, qui doit faire confiance aux concepteurs du module et aux bonnes propriétés d'équilibrage de cette fonction de hachage générique.
Les primitives de la structure de données de table de hachage sont nommées de la manière suivante dans ce module :

1. Rappelez vous : coût d'ajout constant au début pour les implémentations maillons chaînés, coût constant en fin de tableau pour les implémentations tableau

`Hashtbl.create: int -> ('a, 'b) Hashtbl.t :` constructeur. L'utilisateur doit préciser le nombre de seaux m souhaités.

`Hashtbl.add: ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit :` transformateur, ajout d'un couple clé-valeur dans la table. Cette fonction fait appel à la fonction de hachage générique `Hashtbl.hash`. Le type de sorti `unit` permet immédiatement de comprendre que la structure de données est *mutable*.

`Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool :` indique si une entrée ayant un clé donnée est présente parmi toutes les données stockées dans la table de hachage.

`Hashtbl.find: ('a, 'b) Hashtbl.t -> 'a -> 'b :` accesseurs, cherche si une clé est présente dans la table de hachage et retourne la valeur associée si elle existe. Si la clé n'est pas trouvée, l'exception `Not_found` est renvoyée. Il faut donc mettre en place un mécanisme de récupération des exceptions lorsque l'on utilise cette fonction

`Hashtbl.remove: ('a, 'b) Hashtbl.t -> 'a -> unit :` transformateur, retire le couple clé-valeur associé à une clé k de la table de hachage. Si la clé n'est pas présente, cette fonction en fait rien.

`Hashtbl.replace: ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit :` transformateur, remplace la valeur associée à une clé par une nouvelle valeur si elle existe, sinon, elle l'ajoute.

`Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit :` applique une fonction `f: 'a -> 'b -> unit` sur toutes les entrées de la table de hachage.

`Hashtbl.stats:('a, 'b) Hashtbl.t -> unit :` renvoie des informations statistiques sur la table de hachage construite : nombre d'entrées de la table, nombre de seaux, taille maximale s des seaux, diagramme en bâtons des seaux i.e pour chaque longueur, nombre de sauts ayant cette longueur.

Ce module est l'occasion d'introduire une nouvelle syntaxe OCaml : le type `'a option`. Ce module permet de gérer de manière explicite et propre la présence ou l'absence d'une valeur, sans manipuler des exceptions. Le type `'a option` est un type somme à deux constructeurs :

```
type 'a option = None | Some of 'a;;
```

Il permet de gérer le cas de valeurs indéfinies ou d'absence de valeurs sans avoir recours aux exceptions, un peu à la manière de `None` en Python.
Voici un exemple d'utilisation en OCaml, qui généralise une fonction d'addition de deux valeurs dans le cas où l'une des valeurs est non définie :

```
# let a = Some(3);;
val a : int option = Some 3
# let b = None;;
val b : 'a option = None
# let c = Some(4);;
val c : int option = Some 4
# let add_opt a b =
  match (a,b) with
  |(None, _) -> None
  |(_, None) -> None
  |(Some(va), Some(vb)) -> Some(va+vb);;
val add_opt : int option -> int option -> int option = <fun>
# add_opt a b;;
- : int option = None
# add_opt a c;;
- : int option = Some 7
```

Dans le module OCaml `Hashtbl`, il existe une fonction `Hashtbl.find_opt` qui utilise ce type optionnel et renvoie un objet de type `'a option` : si la clé cherchée n'est pas présente dans la table, la fonction renvoie un objet de type `'a option` construit avec le constructeur `None`. Sinon, elle renvoie un objet construit avec le constructeur `Some` avec, comme argument, la valeur associée à la clé trouvée dans la table.

```
(* creation table de hachage à 7 seaux *)
let ht = Hashtbl.create 7;;
val ht : ('_a, '_b) Hashtbl.t = <abstr>
#
(* ajout du couple clé-valeur ('c', 3) *)
Hashtbl.add ht 'd' 3;;
- : unit = ()
#
(* ajout du couple clé-valeur ('a', 5) *)
Hashtbl.add ht 'a' 5;;
- : unit = ()
# ht;;
- : (char, int) Hashtbl.t = <abstr>
#
(* recherche de la clé 'a' dans la table *)
Hashtbl.find_opt ht 'a';;
- : int option = Some 5
#
(* recherche de la clé 'f' dans la table, non présente*)
Hashtbl.find_opt ht 'f';;
- : int option = None
```

Remarque (Convention de nommage). Dans les modules OCaml, les fonctions qui utilisent le type `'a option` possède généralement un nom avec le suffixe `_opt`.

II. 4. Complexité des primitives de manipulation

II. 4. a. Métaphore des tiroirs

Pour illustrer à la fois la simplicité et l'efficacité des tables de hachage, on utilise souvent la **métaphore** des tiroirs.
Lorsque l'on range minutieusement ses habits dans une commode, l'un des tiroirs sera réservé aux chaussettes, l'autre aux culottes, l'autre aux gants, l'autre aux T-shirts... On peut voir ces catégories comme des clés : lorsque vous cherchez votre paire de chaussette rouge favorite, vous irez directement fouiller dans le tiroir des chaussettes, sans perdre votre temps à parcourir les objets présents dans les autres tiroirs. Cela sera beaucoup plus efficace -vous en avez sûrement déjà fait l'expérience - que d'avoir à rechercher vos habits parmi un tas informe de vêtements entassés dans un coin de votre chambre.

II. 4. b. Étude de complexité temporelle et spatiale

On note $s = \max_{i \in [\![0, m-1]\!]} (|\ell_i|)$ la taille maximale des seaux une fois l'ensemble des données couples clé-valeur stockées dans les m seaux de la table de hachage.
Soit (k, v) un couple clé-valeur. Outre le coût de calcul de l'empreinte $i = h(k)$, les primitives de manipulation de la table de hachage ont les complexités temporelles suivantes :

`hshtbl_add :` accès en temps constant au seau ℓ_i par adressage direct dans le tableau, puis ajout dans la liste en temps constant, puisqu'on utilise en fait le seau comme une pile ;

`hshtbl_find :` accès en temps constant au seau ℓ_i par adressage direct dans le tableau, puis parcours des couples clé-valeur stockés dans le seau ℓ_i pour essayer de trouver un couple ayant la clé k ;

`hshtbl_remove :` accès en temps constant au seau ℓ_i par adressage direct dans le tableau, puis parcours des couples clé-valeur stockés dans le seau ℓ_i pour localiser et supprimer le ou les entrées ayant la clé k ;

Ainsi l'ajout d'une entrée se fait en temps constant, tandis que la recherche ou la suppression d'une entrée nécessite, dans le pire des cas, le parcours des éléments contenu dans un seul seau. Si l'on tombe sur le saut le plus rempli, le coût de ces deux dernières primitives est, au pire, en $O(s)$.

II. 4. c. Choix du nombre de seaux m

Il est facile de comprendre que, plus il y a de seaux, plus on va se rapprocher de l'injectivité car cela élargit l'ensemble d'arrivée de la fonction de hachage. Ce nombre de seaux m doit a priori être choisi au moment de la création de la table. Le choix de cet entier m est plutôt délicat et dépend des applications.
Il impacte fortement la taille maximale s des seaux et donc la complexité temporelle. En effet, si la fonction de hachage optimise la répartition des entrées dans les différents sauts, la probabilité qu'un seau reçoive beaucoup plus que n/m entrées devrait être extrêmement faible. En particulier, si n est inférieur à m , très peu de compartiments doivent avoir plus d'un ou deux enregistrements ce qui est favorable en terme de complexité temporelle.

Définition 9 (Facteur d'équilibrage)

On appelle facteur d'équilibrage le facteur :

$$\alpha = \frac{n}{m} \in \mathbb{R}$$

où n est le nombre de couples clé-valeur ajoutés dans la table de hachage et m le nombre seaux de la table de hachage.
Le facteur d'équilibrage représente la taille optimale des seaux obtenue dans le cas très favorable où les entrées ont été réparties de manière équilibrée entre tous les seaux par la fonction de hachage. Il représente le nombre d'entrées par seau théorique que l'on peut espérer, pour un nombre n d'entrées fixé et pour un nombre de seaux m choisi.

On pourrait penser que, plus le nombre de seaux m est grand, en tout cas plus grand que n, mieux c'est, sans se poser plus de questions. Au-delà d'un éventuel gâchis de mémoire, c'est oublier que le nombre de seaux correspond aussi à l'entier m utilisé pour l'opération modulo dans le calcul de certaines fonctions de hachage. Mal choisi, même grand, il peut augmenter les collisions du fait des propriété de l'ensemble $\mathbb{Z}/m\mathbb{Z}$.
En pratique, le nombre de seaux m est souvent choisi de sorte que :

— assez proche du nombre d'entrées n à ranger dans la table, ce qui permet d'espérer un nombre restreint de collisions, puisque le facteur d'équilibrage est alors proche de 1.

— m est un nombre premier, ce qui, en lien avec les résultats de l'arithmétique modulaire, limite les sous-cycles modulaires et donc le nombre de collisions ;

Remarque. Il est possible d'implémenter des tables de hachage redimensionnables, dont le nombre de seaux peut être réajusté au cours de l'exécution si l'on s'aperçoit que certains seaux sont trop remplis, où si l'on ne connait pa à l'avance le nombre d'entrées n qui seront rangées dans la table. On parle alors de tables de hachage dynamiques.

II. 4. d. Qualité de la fonction de hachage

Comme nous l'avons vu, dans la vraie vie, les fonctions de hachage peuvent avoir des défauts et ne pas équilibrer parfaitement les seaux, certains étant trop remplis (au-dessus du facteur d'équilibrage) tandis que d'autres sont en sous-effectif. Un petit nombre de collisions est donc pratiquement inévitable, même si le nombre de seaux m est beaucoup plus grand que le nombre de données n.
Bien sûr, la fonction de hachage ultime, qui répondrait à toutes les situations, n'existe pas, mais deux propriétés sont très souvent recherchées :

Uniformité de la distribution des empreintes dans les seaux : dans le cas où toutes les clés de mon jeu de données sont équiprobables, plus les empreintes des clés sont distribuées de manière uniforme dans tous les seaux, plus les seaux de la table de hachage seront équilibrés et plus la complexité temporelle moyenne des primitives sera optimale. Sinon, il faut adapter la fonction de hachage à la probabilité d'apparition des clés, en trouvant une clé de hachage qui minimise la taille des seaux fréquemment parcourus, c'est-à-dire ceux associés aux clés les plus fréquentes. Mais cela suppose de posséder en amont des informations sur les données (et en particulier les clés) qui vont être stockées.

Rapidité de l'évaluation : plus la fonction de hachage est rapide à calculer, mieux c'est !

La construction et l'étude des fonctions de hachage ayant de bonnes propriétés d'équilibrage constitue un domaine de recherche à part entière. Ces études peuvent être :

— très théoriques, en faisant notamment intervenir des méthodes probabilistes pour étudier la loi de distribution des empreintes sur la plage d'entiers ciblée en fonction de la probabilité d'apparition des clés, et vérifier l'optimisation de la taille des seaux ;

— mais elles sont également, très souvent, validées expérimentalement sur des données réelles, en étudiant le nombre de collisions, l'équilibrage réel des seaux comparé avec le facteur d'équilibrage α optimal.

Très souvent, des méthodes heuristiques permettent, au cas par cas et en fonction des applications, d'améliorer notablement les performances des tables de hachage dans tel ou tel contexte.

III. Bilan

III. 1. Bilan sur les tables de hachage

Les performances en terme de coût temporel et spatial d'une table de hachage dépendent donc très fortement :

Du nombre de seaux : si le nombre de seaux est trop petit par rapport aux nombre de données à ranger dans la table, la table de hachage sera peu efficace. Mais, s'il y a trop de seaux et que de nombreux seaux sont vides, on aura gaspillé de l'espace mémoire. En général, le nombre de seaux m est choisie de sorte que $n = \beta m$ où β est un petit entier typiquement $\beta = 1, 2 ou 3$. Le nombre de seaux peut avoir un impact sur le nombre de collisions en lien avec l'arithmétique modulaire. Pour éviter ce type de problème, il peut être intéressant de choisir un nombre de seaux m premier.

Du choix de la fonction de hachage : c'est en effet la fonction de hachage qui va répartir les différentes entrées dans les seaux à partir des empreintes de leurs clés, et va donc conditionner la présence ou non de seaux surchargés. Si la probabilité d'apparition des clés de mon ensemble de clés est équiprobable, les performances seront d'autant meilleures que mes seaux sont de même taille, proche du facteur d'équilibrage α.

Lorsque le nombre de seaux est adaptée et lorsque la fonction de hachage est bien choisie, c'est-à-dire lorsque qu'elle **minimise la taille des seaux fréquemment consultés en donnant l'illusion d'une quasi-injectivité**, les tables de hachage constituent des structures de données extrêmement performantes pour trouver une entrée à partir de sa clé en temps constant très faible, dans des structures de données abstraites de dictionnaire, d'ensemble... ou autre. Il faut les privilégier dès que cela est possible.
Elles sont abondamment utilisées dans tous les domaines de l'informatique :

— gestion des tables en programmation réseau,

— dans les codes de calcul scientifique,

— dans les algorithmes de manipulation de très grandes bases de données.

Nous étudierions bientôt les bases de données SQL, qui utilisent, en sous-main et de manière très abondante, des mécanismes de hachage implémentés en langage C/C++.

III. 2. Bilan sur toutes les méthodes de recherche vues et leur complexité

On note n le nombre de couples clés valeurs du jeu de données, m le nombre de seaux pour les tableaux associatifs (dans ce cas $m \geq n$) et les tables de hachage.

Méthode utilisée	C_{spatiale}	C_{temp} recherche	C_{temp} insertion	C_{temp} suppression
Recherche séq.	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Recherche dicho [2]	$\Theta(1)$	$\Theta(\log_2(n))$	$\Theta(n)$	$\Theta(n)$
Par tableau associatif [3]	$\Theta(m)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Par ABR bien équilibré	$\Theta(n)$	$\Theta(\log_2(n))$	$\Theta(1)$	$\Theta(\log_2(n))$
Par table de hach. bien équilibrée [4]	$\Theta(n)$	$\Theta(\frac{n}{m})$	$\Theta(1)$	$\Theta(\frac{n}{m})$

Toutes ces méthodes de recherche nécessitent la définition d'une relation d'égalité sur l'ensemble des clés : il faut être capable de définir ce que signifie l'égalité entre deux clés. La recherche dichotomique et les ABR imposent que les clés appartiennent à un ensemble totalement ordonné et la définition de cette relation d'ordre. La recherche séquentielle et les tables de hachage ne l'imposent pas.