COURS : TABLES ET FONCTIONS DE HACHAGE

I) INTRODUCTION

I.1. Présentation des tables de hachage

Dans ce chapitre, nous allons étudier les **tables de hachage**, des structures pensées pour des recherches ultra-rapides sur un ensemble qui évolue, avec trois opérations phares : la recherche, l'insertion et la suppression. Typiquement, ces opérations se font en temps constant quand l'implémentation (fonction de hachage, taille de table) est bien choisie et que les données ne sont pas pathologiques.

Nous verrons pourquoi les collisions sont inévitables et comment les résoudre via la technique de chaînage ou d'adressage ouvert. Nous parlerons également du rôle du facteur de charge sur les performances.

Les tables de hachage, comme les tas (heaps) et les arbres de recherche, maintiennent un ensemble évolutif d'objets associés à des clés (et éventuellement à beaucoup d'autres données). À la différence des tas et des arbres de recherche, elles ne conservent aucune information d'ordre, mais une table de hachage peut dire ce qui est présent et ce qui ne l'est pas, et elle peut le faire très, très vite (bien plus rapidement qu'un tas ou un arbre de recherche ou encore qu'une liste en Python).

Nous commencerons par étudier les opérations prises en charge avant de passer aux applications puis à quelques détails d'implémentation.

I.2. Qu'est-ce qu'une clé ?

Une **clé** est un identifiant unique qui permet d'accéder à une valeur sans parcourir toute la structure. Par exemple, les valeurs peuvent correspondre à des fiches d'employés, avec pour clés leurs numéros d'identification. Elles peuvent aussi représenter les arêtes d'un graphe, avec des clés correspondant aux longueurs des arêtes. Ou bien il peut s'agir d'événements programmés dans le futur, chaque clé indiquant l'instant auquel l'événement aura lieu.

Les dictionnaires en Python utilisent des clés. Par exemple, dans le dictionnaire ci-dessous :

```
annuaire = {
    "Alice": "+33 6 12 34 56 78",
    "Bob":   "+33 7 11 22 33 44",
}
```

… les clés sont `Alice` et `Bob`, et les valeurs associées sont les numéros de téléphone. Les clés d'un dictionnaire doivent être « hashables » (les `str` comme `Alice` le sont) et une clé est unique dans un dictionnaire : réassigner la même clé remplace la valeur.

II) OPÉRATIONS PRISES EN CHARGE PAR LES TABLES DE HACHAGE

Le but d'une table de hachage est de suivre un ensemble évolutif d'objets associés à des clés tout en prenant en charge des recherches rapides (par clé), afin qu'il soit facile de vérifier ce qui est présent et ce qui ne l'est pas.

Par exemple, si une entreprise gère un site d'e-commerce, elle peut utiliser une table de hachage pour suivre les employés (avec, par exemple, les noms comme clés), une autre pour stocker les transactions passées (avec les identifiants de transaction comme clés), et une troisième pour mémoriser les visiteurs du site (avec les adresses IP comme clés).

Conceptuellement, on peut voir une table de hachage comme un tableau. L'un des points forts des tableaux, c'est l'accès aléatoire immédiat. En effet, si on connaît la position d'un objet dans un tableau (par exemple la position numéro 17), on peut accéder à cette valeur immédiatement, en temps constant.

Supposons qu'on souhaite avoir une structure de données pour mémoriser les numéros de téléphone de nos amis. Imaginons qu'ils soient numérotés avec des entiers positifs, disons entre 1 et 10 000. Dans ce cas, on peut stocker les numéros de téléphone dans un tableau de taille 10 000. Si votre meilleur ami s'appelle 173, rangez son numéro à la position 173 du tableau. Pour « oublier » votre ex-ami 548, écrasez la position 548 avec une valeur par défaut. Cette solution à base de tableau fonctionne bien, même si vos amis changent au fil du temps : les besoins en espace restent modestes et les insertions, suppressions et recherches s'exécutent en temps constant.

Bien sûr, nos amis ont des prénoms plus intéressants, mais moins pratiques que des numéros, comme Alice, Bob, Carole, etc. Peut-on malgré tout utiliser une solution basée sur un tableau ? En principe, on pourrait maintenir un tableau dont les cases sont indexées par tous les noms possibles qu'on pourrait rencontrer (comportant au plus, disons, 25 lettres). Pour retrouver le numéro d'Alice, il suffirait de regarder alors la case « Alice » du tableau :

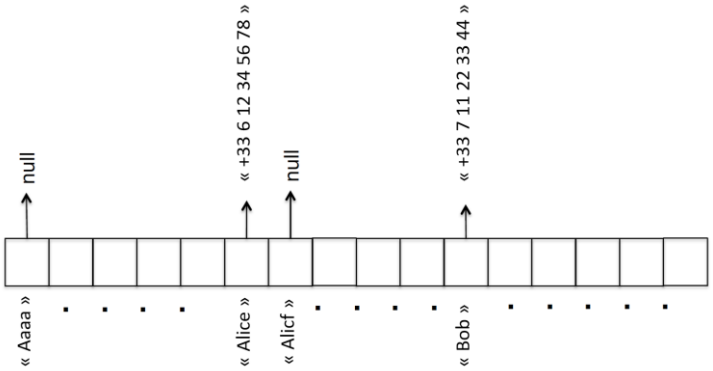


Figure 1 : Tableau indexé par des chaînes de caractères d'au plus 25 caractères

Mais un tableau de ce genre nécessiterait un nombre d'enregistrements de $N = 26^{25}$ chaînes de caractères, ce qui est bien trop grand ! Il nous faut donc une autre structure de données qui reproduise toutes les fonctionnalités d'un tableau (avec des insertions, suppressions et recherches en temps constant), tout en n'utilisant qu'un espace proportionnel au nombre d'objets stockés. Une table de hachage est précisément une telle structure de données.

Les **opérations prises en charge** par une table de hachage sont les suivantes :

Opérations prises en charge par une table de hachage

Recherche d'un élément (lookup) : pour une clé k, renvoyer un pointeur vers un objet de la table de hachage ayant la clé k (ou indiquer qu'aucun objet n'existe).

Insertion : étant donné un nouvel objet x, ajouter x à la table de hachage.

Suppression : pour une clé k, supprimer de la table de hachage un objet ayant la clé k, s'il en existe un.

Dans une table de hachage, toutes **ces opérations s'exécutent en général en temps constant**, (comme pour la solution naïve basée sur un tableau), sous certaines conditions que nous étudierons et qui tiennent généralement en pratique.

Une table de hachage utilise un espace linéaire par rapport au nombre d'objets stockés. C'est radicalement inférieur à l'espace requis par la solution naïve à base de tableau, qui est proportionnel au nombre de tous les objets imaginables susceptibles d'être stockés un jour.

En résumé, les tables de hachage n'offrent pas beaucoup d'opérations, mais ce qu'elles font, elles le font vraiment, vraiment bien. Dès que les recherches représentent une part significative du travail d'un algorithme, une table de hachage est sûrement la bonne solution à utiliser.

III) EXEMPLES D'APPLICATIONS

De nombreuses applications différentes se ramènent à des recherches répétées et appellent donc une table de hachage. Dans les années 1950, les chercheurs qui construisaient les premiers compilateurs avaient besoin d'une table des symboles, c'est-à-dire d'une bonne structure de données pour suivre les noms de variables et de fonctions d'un programme. Les tables de hachage ont été inventées précisément pour ce type d'application.

Pour un exemple plus moderne, imaginez qu'un routeur réseau doive bloquer les paquets provenant de certaines adresses IP, appartenant par exemple à des expéditeurs de pourriel. À chaque arrivée d'un nouveau paquet, le routeur doit vérifier si l'adresse IP source figure dans la liste de blocage. Si oui, il jette le paquet ; sinon, il le transmet vers sa destination. Là encore, ces recherches répétées sont exactement le terrain de prédilection des tables de hachage.

III.1. Déduplication

La déduplication est un processus qui élimine les copies excessives de données et réduit considérablement les besoins en capacité de stockage. C'est une application canonique des tables de hachage.

Supposons qu'on doive traiter une quantité massive de données qui arrivent morceau par morceau, sous forme de flux. Par exemple :

- Parcourir une seule fois un énorme fichier stocké sur disque, comme l'ensemble des transactions d'une grande enseigne de vente au détail sur l'année écoulée.
- Explorer le Web (web crawling) et traiter des milliards de pages.
- Suivre les paquets de données qui traversent un routeur réseau à un rythme torrentiel.

Dans le problème de déduplication, le but est d'ignorer les doublons et de ne conserver que les clés distinctes déjà observées. Par exemple, s'intéresser au nombre d'adresses IP distinctes qui ont accédé à un site Web, en plus du nombre total de visites.

L'utilisation des tables de hachage dans le problème de déduplication repose sur le principe suivant : lorsqu'un nouvel objet x de clé k arrive :

- Utiliser l'opération de recherche pour vérifier si la table de hachage contient déjà un objet de clé k.
- Si ce n'est pas le cas, insérer x dans la table de hachage.

Après le traitement des données, la table de hachage contient exactement un objet par clé présente dans le flux de données.

III.2. Recherches dans un vaste espace d'états

Les tables de hachage servent avant tout à accélérer la recherche. Un domaine d'application où la recherche est omniprésente est le jeu, et plus généralement les problèmes de planification.

Pensez, par exemple, à un programme de jeu d'échecs qui explore les conséquences de différents coups. Les séquences de coups peuvent être vues comme des chemins dans un immense graphe orienté : les sommets correspondent aux états du jeu (position de toutes les pièces et joueur dont c'est le tour) et les arêtes correspondent aux coups (d'un état à un autre). La taille de ce graphe est astronomique (plus de 10^{100} sommets), il est donc hors de question de l'écrire explicitement pour y appliquer les algorithmes de recherche de graphes.

Une alternative plus praticable consiste à exécuter un algorithme de recherche comme la recherche en largeur (BFS), à partir de l'état courant, et à explorer les conséquences à court terme des différents coups jusqu'à atteindre une limite de temps.

Pour apprendre le plus possible, il est crucial d'éviter d'explorer un même sommet plusieurs fois ; l'algorithme doit donc mémoriser quels sommets ont déjà été visités. Comme dans notre application de déduplication, cette tâche est taillée pour une table de hachage : lorsque l'algorithme atteint un sommet, il le recherche dans la table de hachage. S'il s'y trouve déjà, l'algorithme l'ignore et revient en arrière ; sinon, il insère le sommet dans la table et poursuit l'exploration.

IV) IMPLÉMENTATION : IDÉES GÉNÉRALES

Cette partie présente les idées les plus importantes pour l'implémentation d'une table de hachage : les fonctions de hachage (qui associent les clés à des positions dans un tableau), les collisions (plusieurs clés mappées sur la même position) et les stratégies de résolution de collisions les plus courantes.

IV.1. Pourrait-on utiliser une liste en Python ?

Supposons que l'on veuille gérer l'enregistrement du nombre de visites d'un site web par adresse IP. L'univers U dans cet exemple serait l'ensemble des 2^{32} adresses IPv4 possibles, les clés seraient les adresses IP et les objets à gérer seraient le nombre de visites. Une façon conceptuellement simple d'implémenter les opérations d'accès, d'insertion et de suppression rapides consisterait à stocker le nombre de visites dans une grande liste, avec une case pour chaque adresse IP possible de l'ensemble U.

Bien entendu, la taille de l'univers U est énorme et cela demanderait beaucoup trop d'espace mémoire. La solution serait donc que la liste gère un sous-ensemble $S \subseteq U$ de taille raisonnable plutôt que l'univers U dans son ensemble. Ainsi, dans notre exemple, l'ensemble S représenterait uniquement les adresses IP ayant visité une page web au cours des dernières 24 heures. Dans ce cas, l'espace mémoire utilisé serait uniquement proportionnel à $|S|$, c'est-à-dire au nombre réel d'éléments stockés. Dans la plupart des applications utilisant des tables de hachage, la taille de l'univers U est immense, tandis que celle du sous-ensemble S reste gérable.

Cependant, les temps d'exécution des opérations de recherche avec les listes augmentent linéairement avec $|S|$ car l'indice (ou l'adresse) de l'élément recherché n'est pas connu à l'avance. Python doit donc parcourir la liste élément par élément jusqu'à trouver la bonne valeur, ou arriver à la fin. La liste ne possède aucune information interne sur la position de chaque valeur.

Les tables de hachage apportent une solution à ce problème : au lieu de parcourir toutes les valeurs, elles utilisent une clé pour calculer directement la position où se trouve la donnée. Ce calcul passe par une fonction de hachage, qui transforme la clé en un indice numérique (ou adresse).

IV.2. Fonction de hachage et table de hachage

Une **fonction de hachage** effectue la traduction entre ce qui nous intéresse réellement, comme les noms de nos amis, les adresses IP ayant visité un site web au cours des dernières 24 heures, etc., et les positions dans la table de hachage.

De manière formelle, une fonction de hachage est une fonction qui associe chaque clé possible (appartenant à l'ensemble U) à une position dans le tableau. Dans une table de hachage, les positions sont généralement numérotées à partir de 0, ce qui fait que l'ensemble des positions du tableau est : $\{0, 1, 2, \ldots, n-1\}$.

La figure 2 illustre le principe de fonctionnement d'une fonction de hachage et comment celle-ci est associée à une table de hachage (certaines positions i = 2, 4, 5 sont vides) :

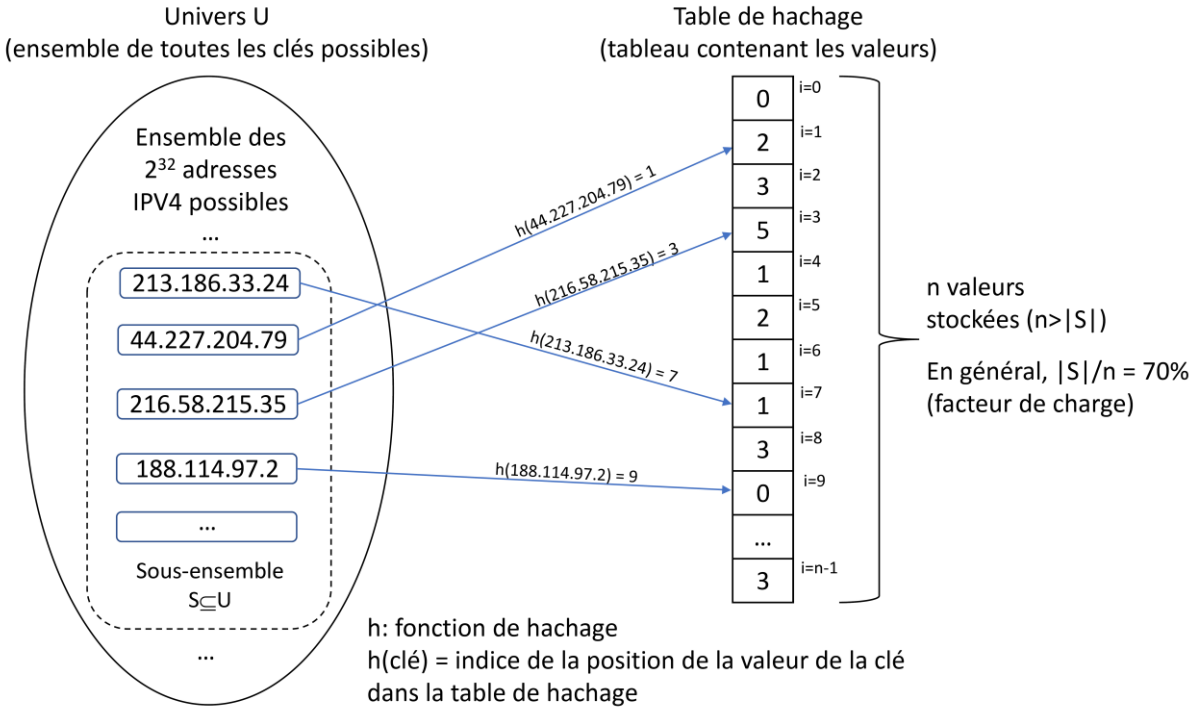


Figure 2 : Principe de fonctionnement d'une table de hachage

Une fonction de hachage indique où commencer la recherche d'un objet. Dans notre exemple, si on choisit une fonction de hachage h telle que h(213.186.33.24) = 7, alors la position 7 du tableau est l'endroit où il faut commencer à chercher le nombre de visites effectuées. De même, la position 7 sera le premier emplacement à essayer pour insérer le nouveau nombre de visites de cette IP dans la table de hachage.

IV.3. Le problème des collisions

Que se passe-t-il si deux clés différentes (par exemple 213.186.33.24 et 66.16.10.23) sont hachées vers la même position (par exemple 7) ? Si on cherche le nombre de visites de l'IP 66.16.10.23 mais qu'on trouve le nombre de visites de l'IP 213.186.33.24 dans la table de hachage à l'indice i = 7, comment savoir si le nombre de visites de l'IP 66.16.10.23 se trouve également dans la table de hachage ? Et on essaye d'insérer le nombre de visites de l'IP 66.16.10.23 à la position i = 7 mais que celle-ci est déjà occupée, où devons-nous le placer ?

Lorsqu'une fonction de hachage h associe deux clés différentes k_1 et k_2 à la même position, autrement dit, quand $h(k_1) = h(k_2)$, on appelle cela une **collision**. Les collisions créent de la confusion quant à l'emplacement réel d'un objet dans la table de hachage, et il est donc souhaitable de les réduire autant que possible.

Malheureusement, elles sont inévitables. La raison tient au principe des tiroirs : pour tout entier positif n, quelle que soit la façon dont on place n + 1 chaussettes dans n cases, il y aura au moins une case contenant au moins deux chaussettes. Ainsi, toute fonction de hachage, aussi ingénieuse soit-elle, provoquera au moins une collision chaque fois que le nombre de positions disponibles dans le tableau est inférieur à la taille de l'univers U.

En fait, les collisions sont encore plus inévitables que ne le suggère l'argument fondé sur le principe des tiroirs. La raison en est le paradoxe des anniversaires :

> *Considérons n personnes dont les anniversaires sont aléatoires, chacun des 365 jours de l'année étant équiprobable. À partir de quelle valeur de n y a-t-il au moins 50 % de chances que deux personnes aient le même anniversaire ?*

Avec 366 personnes, il y aurait 100 % de chances que deux personnes partagent le même anniversaire (par le principe des tiroirs). Mais avec déjà 57 personnes, la probabilité est d'environ 99 %. En réalité, 23 personnes suffisent pour ≈ 50 %, car ce qui compte est la multiplicité des paires possibles : avec 23 personnes, il y a C(23, 2) = 253 paires, ce qui fait déjà beaucoup d'occasions de coïncider.

Imaginons donc une fonction de hachage qui assigne chaque clé, de façon indépendante et uniforme au hasard, à une position dans {0, 1, 2, …, n−1}. Nous verrons que ce n'est pas une fonction de hachage utilisable en pratique, mais de telles fonctions aléatoires constituent une base théorique de fonctions « idéales » qui servent de comparaison pour les fonctions de hachage utilisées en pratique.

Le paradoxe des anniversaires implique que, même pour cette fonction idéale, nous commencerons probablement à voir des collisions dans une table de taille n dès qu'une petite constante fois n objets auront été insérés. Par exemple, lorsque n = 10 000, l'insertion de 200 objets a de fortes chances de provoquer au moins une collision, alors même qu'au moins 98 % des cases du tableau restent totalement inoccupées.

IV.4. Gestion des collisions par chaînage

Les collisions étant inévitables, une table de hachage a besoin d'une méthode pour les résoudre. Cette section et la suivante décrivent les deux approches dominantes : le **chaînage** séparé (ou simplement chaînage) et **l'adressage ouvert**.

Les deux mènent à des implémentations où les insertions et les recherches s'exécutent généralement en temps constant, à condition que la taille de la table et la fonction de hachage soient bien choisies et que les données ne soient pas pathologiques.

IV.4.1. Utilisation des listes

L'idée clé de la méthode du chaînage est de revenir par défaut à la solution basée sur une liste (on parle plutôt de liste chaînée) pour gérer les multiples objets mappés sur la même position du tableau. Avec le chaînage, les positions du tableau sont souvent appelées des **seaux** ou des **compartiments** car chacune **peut contenir plusieurs objets**. Les opérations de recherche, d'insertion et de suppression se réduisent alors à une évaluation de la fonction de hachage (pour déterminer le bon seau) suivie de l'opération correspondante sur la liste.

La figure 3 ci-dessous illustre une table de hachage dont les collisions sont résolues par chaînage, avec n seaux et n objets. Les clés « 188.144.97.2 » et « 83.2.1.10 » entrent en collision dans le seau n°9. La valeur de la clé « 83.2.1.10 » (5) est donc insérée dans le seau n°9, qui contient maintenant la liste [0, 5] :

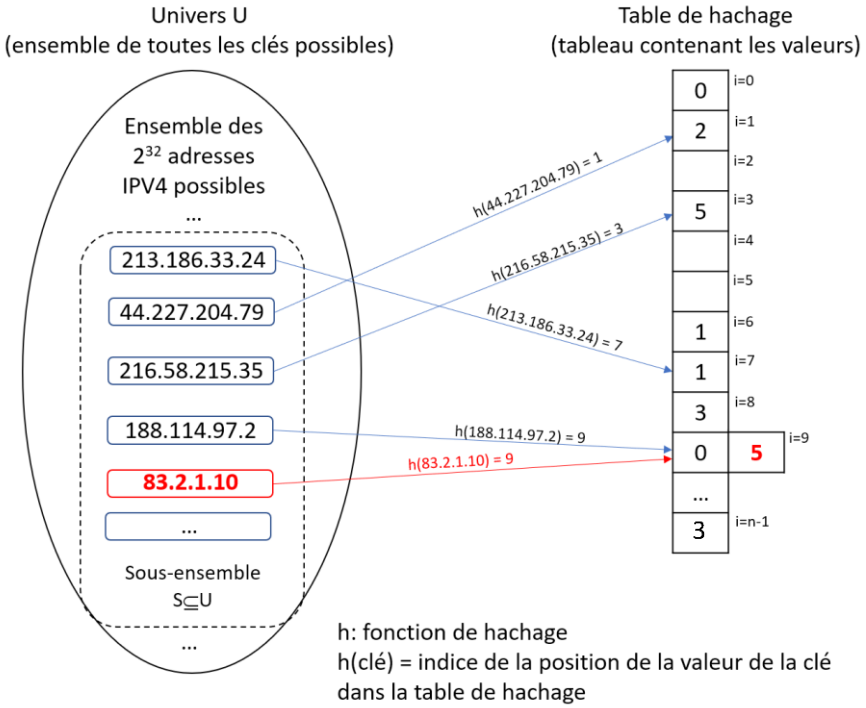


Figure 3 : Gestion des collisions par chaînage

Pour les opérations de recherche, d'insertion ou de suppression d'un objet de clé k, on effectue l'opération souhaitée sur la liste du seau A[h(k)], où h est la fonction de hachage et A est le tableau de la table de hachage.

IV.4.2. Performances de la gestion par chaînage

À condition que h puisse être évaluée en temps constant, l'opération d'insertion prend elle aussi un temps constant car le nouvel objet peut être inséré immédiatement dans la liste.

Les opérations de recherche et de suppression doivent, elles, parcourir la liste stockée dans A[h(k)], ce qui prend un temps proportionnel à la longueur de cette liste. Pour obtenir des recherches en temps constant dans une table de hachage avec chaînage, il faut que les listes des seaux restent courtes, idéalement de longueur au plus une petite constante.

La longueur des listes (et donc les temps de recherche) se dégrade si la table de hachage devient très remplie. Par exemple, si 100·n objets sont stockés dans une table dont le tableau a une longueur n, un seau typique contient 100 objets à parcourir.

Les temps de recherche peuvent aussi se dégrader avec une mauvaise fonction de hachage qui provoque beaucoup de collisions. Par exemple, dans le cas extrême où tous les objets entrent en collision et se retrouvent dans le même seau, les recherches peuvent prendre un temps linéaire en la taille du jeu de données. Nous verrons comment gérer la taille d'une table de hachage et choisir une fonction de hachage appropriée pour atteindre de bonnes performances.

IV.5. Gestion des collisions par adressage ouvert

La deuxième méthode populaire pour résoudre les collisions est **l'adressage ouvert**. L'adressage ouvert est beaucoup plus facile à implémenter et à comprendre lorsque la table de hachage ne doit prendre en charge que les opérations d'insertion et de recherche (et pas de suppression) ; nous nous concentrerons sur ce cas.

Avec l'adressage ouvert, chaque position du tableau **stocke 0 ou 1 objet**, plutôt qu'une liste. Pour que cela ait un sens, la taille |S| du jeu de données ne peut pas dépasser la taille n de la table. Les collisions posent un dilemme immédiat pour l'opération d'insertion : où placer un objet de clé k si un objet différent est déjà stocké à la position A[h(k)] ?

IV.5.1. Séquences de sondage

L'idée est d'associer à chaque clé k une séquence de « sondage » de positions, et non pas une seule position. Le premier nombre de la séquence indique la position à considérer en premier ; le second, la position suivante à considérer si la première est déjà occupée ; et ainsi de suite. L'objet est alors stocké dans la première position inoccupée donnée par la séquence de sondage.

La figure 4 illustre le cas où la clé « 83.2.1.10 » entre en collision avec la clé « 188.114.97.2 » et où la séquence de sondage est par exemple {9, 8, 6, 4, 1, 2, …}. Lors de la collision en i = 9, on tente une insertion sur i = 8, mais comme cet espace est déjà occupé, on teste i=6, puis i = 4 et comme i = 4 est libre, la valeur de la clé « 83.2.1.10 » (5) est placée à cet endroit.

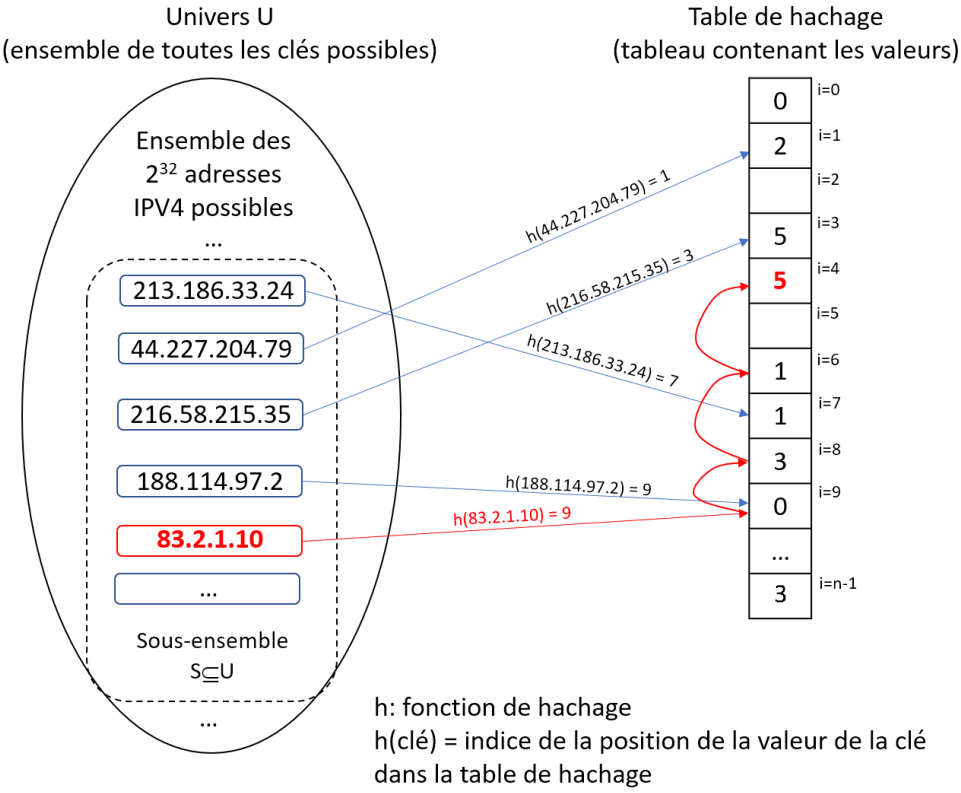


Figure 4 : Principe de la gestion des collisions par adressage ouvert

Voyons maintenant deux techniques utilisées pour sonder les positions.

IV.5.2. Sondage linéaire

Cette technique utilise la fonction de hachage de manière « **linéaire** » pour définir une séquence de sondage. Elle utilise une unique fonction de hachage h et définit la séquence de sondage pour une clé k comme h(k), puis h(k) + 1, puis h(k) + 2, et ainsi de suite (en repartant au début lorsqu'on atteint la dernière position).

Autrement dit, la fonction de hachage indique la position de départ pour une insertion ou une recherche, et l'opération balaye vers la droite jusqu'à trouver l'objet désiré ou une position vide.

IV.5.3. Sondage par double hachage

Une méthode plus sophistiquée est le **double hachage**, qui utilise deux fonctions de hachage. La première donne la première position de la séquence de sondage, et la seconde indique le pas (offset) pour les positions suivantes.

Par exemple, si $h_1(k) = 17$ et $h_2(k) = 23$, le premier endroit où chercher un objet de clé k est la position 17 (position par défaut), puis la position 40, puis 63, puis 86, et ainsi de suite.

Pour une autre clé k', la séquence peut être très différente : si $h_1(k') = 42$ et $h_2(k') = 27$, la séquence de sondage sera 42, puis 69, puis 96, puis 123, etc.

IV.5.4. Performances de la gestion par adressage ouvert

Avec le chaînage, le temps d'exécution d'une recherche est gouverné par la longueur des listes des seaux. Avec l'adressage ouvert, c'est le nombre typique de sondages nécessaires pour trouver soit une case vide, soit l'objet recherché.

Il est plus difficile de comprendre les performances d'une table de hachage avec adressage ouvert qu'avec chaînage, mais il est intuitivement clair que les performances se dégradent à mesure que la table se remplit : moins il y a de cases libres, plus une séquence de sondage mettra longtemps à en trouver une. Le choix de la fonction de hachage est également important, car un mauvais choix peut provoquer beaucoup de collisions.

Avec une taille de table et une fonction de hachage appropriées, l'adressage ouvert atteint de bonnes performances pour les opérations d'insertion et de recherche.

IV.6. Comment choisir une bonne fonction de hachage

Quelle que soit la stratégie de résolution des collisions employée, les performances d'une table de hachage se dégradent avec le nombre de collisions. Comment choisir une fonction de hachage de façon à ne pas en avoir trop ?

IV.6.1. Exemple d'une mauvaise fonction de hachage

Il existe une myriade de façons de définir une fonction de hachage, et le choix est important. Voyons par exemple l'impact sur les performances d'une table de hachage si on choisit la fonction de hachage la plus « stupide » possible :

> *Considérons une table de hachage de longueur n ($n \geq 1$), et soit h la fonction de hachage telle que $h(k) = 0$ pour toute clé $k \in U$. Supposons qu'un ensemble de données S soit inséré dans la table, avec $|S| \leq n$. Quel est le temps d'exécution typique des opérations de recherche ultérieures ?*

Dans le cas d'une gestion des collisions par chaînage, la fonction de hachage renvoie l'objet S dans le même seau (le seau 0). La table de hachage dégénère alors en une simple solution par liste, avec un temps de recherche en $O(|S|)$.

Dans le cas de l'adressage ouvert, supposons que la table utilise le sondage linéaire (l'histoire est similaire pour des stratégies plus sophistiquées comme le double hachage). Le premier objet de S sera placé à la position 0 du tableau, le suivant à la position 1, et ainsi de suite. L'opération de recherche dégénère en une recherche linéaire à travers les $|S|$ premières positions d'un tableau non trié, ce qui nécessite également un temps en $O(|S|)$.

IV.6.2. Données pathologiques

Au lieu d'implémenter la fonction de hachage « stupide » précédente, il faut essayer de concevoir une fonction de hachage garantissant peu de collisions. Mais même la plus maligne des fonctions de hachage offrira de piètres performances si le jeu de données est mal choisi, comme un immense jeu de données pour lequel tous les objets entrent en collision.

Malheureusement, on peut toujours construire un jeu de données « catastrophique » qui provoque un tas de collisions au même endroit.

En effet, si on répartit $|U|$ clés dans n cases, la moyenne par case est $|U|/n$. Donc au moins une case contient au moins $|U|/n$ clés. En prenant S comme l'ensemble des clés qui tombent dans cette case, on obtient un ensemble de taille $|S|/n$.

Si la malchance choisit précisément un tel S, alors notre table dégénère : dans le cas de la gestion par chaînage, la liste étant très longue dans un seau, le temps de recherche sera en $O(|S|)$. Dans le cas de l'adressage ouvert, il y aura de longues séquences de sondage, ce qui engendrera également un temps linéaire.

En conclusion, on ne peut pas promettre un temps $O(1)$ pour tous les jeux de données possibles, quelle que soit la fonction h. Contrairement à la plupart des algorithmes et structures de données, la garantie du temps d'exécution dépend des hypothèses sur l'entrée. Le mieux que l'on puisse espérer, c'est une garantie qui s'applique à tous les jeux de données non pathologiques, c'est-à-dire définis indépendamment de la fonction de hachage choisie.

Avec une fonction de hachage bien conçue, il n'y a généralement pas lieu de s'inquiéter des jeux de données pathologiques en pratique. Les applications de sécurité constituent toutefois une exception importante à cette règle car aucune fonction de hachage n'est à l'abri d'un jeu de données spécialement conçu contre elle.

IV.6.3. Fonctions de hachage purement aléatoires

Les jeux de données pathologiques montrent qu'aucune fonction de hachage ne garantit d'avoir un petit nombre de collisions pour tous les jeux de données. Le mieux que l'on puisse espérer est une fonction de hachage qui engendre peu de collisions pour tous les jeux de données non pathologiques.

Une approche radicale pour décorréler le choix de la fonction de hachage et le jeu de données consiste à choisir une fonction aléatoire, c'est-à-dire une fonction h telle que, pour chaque clé $k \in U$, la valeur h(k) soit choisie indépendamment et uniformément au hasard parmi les positions du tableau {0, 1, 2, …, n−1}. La fonction h est fixée une fois pour toutes lors de la création initiale de la table de hachage.

Intuitivement, on s'attend à ce qu'une telle fonction aléatoire répartisse en moyenne les objets d'un jeu de données S à peu près uniformément sur les n positions, pourvu que S soit défini indépendamment de h. Tant que n est de l'ordre de |S|, cela conduit à un nombre de collisions maîtrisé.

Cependant, une « vraie » fonction aléatoire n'est pas utilisable en pratique, car pour chaque clé $k \in U$, la valeur h(k) est indépendante des autres valeurs. Donc pour décrire complètement h, il faut mémoriser la sortie pour chaque entrée, autrement dit créer une table de correspondance de |U| entrées. Or, quand l'univers est grand (comme dans la plupart des applications), cela est impossible en pratique (mémoire et temps astronomiques).

On pourrait essayer de définir la fonction de hachage au fur et à mesure des besoins, en attribuant une valeur aléatoire à h(k) la première fois que la clé k apparaît. Mais alors, évaluer h(k) exige d'abord de vérifier si elle a déjà été définie. Cela revient à effectuer une recherche pour k, c'est-à-dire exactement le problème que nous sommes censés résoudre !

L'idée d'introduire de l'aléa n'est cependant pas à rejeter complètement, comme nous le verrons dans le chapitre consacré aux familles de fonctions de hachage universelles.

IV.6.4. Les bonnes fonctions de hachage

Une « bonne » fonction de hachage est celle qui bénéficie des avantages d'une fonction aléatoire sans ses inconvénients et qui remplit ces exigences :

- Peu coûteuse à évaluer, idéalement en O(1) ;
- Facile à stocker, idéalement avec une mémoire O(1) ;
- Imite une fonction aléatoire en répartissant les jeux de données non pathologiques à peu près uniformément sur les positions de la table de hachage.

Considérons des clés qui sont des entiers compris entre 0 et un grand nombre M.

Une première tentative naturelle pour une fonction de hachage consiste à prendre la valeur de la clé modulo le nombre n de seaux :

$$h(k) = k \bmod n$$

… où k mod n est le résultat obtenu en soustrayant n de k de façon répétée jusqu'à obtenir un entier compris entre 0 et n−1.

La figure 5 illustre cette idée avec n = 10 dans le cas où U est l'ensemble des entiers :

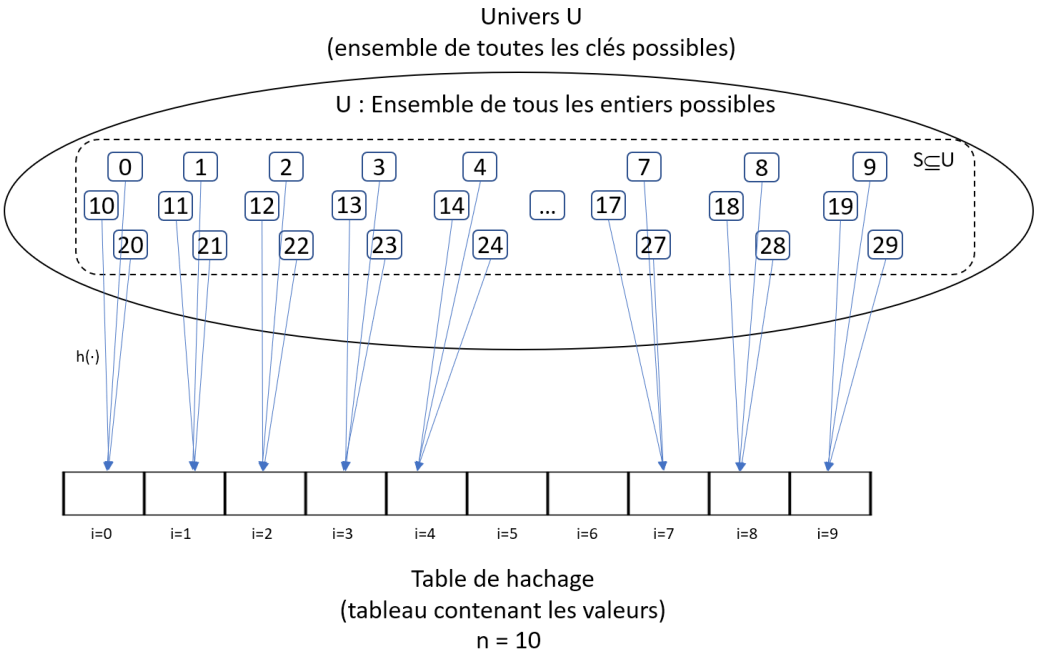


Figure 5 : Fonction de hachage k mod n avec n=10

La bonne nouvelle, c'est que cette fonction est peu coûteuse à évaluer et ne nécessite aucun stockage (au-delà de la mémorisation de n).

La mauvaise nouvelle, c'est que même si cette fonction semble correctement distribuer les h(k), elle « regarde » surtout certains chiffres/bits de la clé. Si ces chiffres/bits ne sont pas variés dans les données, on aura des collisions massives.

Par exemple, si n = 1000 et que toutes les clés ont les trois mêmes derniers chiffres (en base 10) (par exemple des salaires d'entreprise tous multiples de 1000, ou des prix de voitures se terminant tous par « 999 »), alors toutes les clés sont hachées vers la même position. N'utiliser que les bits de poids fort peut causer des problèmes similaires (pensez, par exemple, aux indicatifs de pays et de région des numéros de téléphone).

Une idée serait de mélanger une clé avant d'appliquer l'opération de modulo :

$$h(k) = (a \cdot k + b) \bmod n$$

… où n, a et b sont des entiers dans {1, 2, …, n-1}.

On voit sur la figure 6 que lorsque les données de S ⊆ U sont uniformes, les deux fonctions se comportent bien et se ressemblent. La répartition des clés dans les 1000 seaux est assez homogène (les courbes se superposent quasi complètement) :

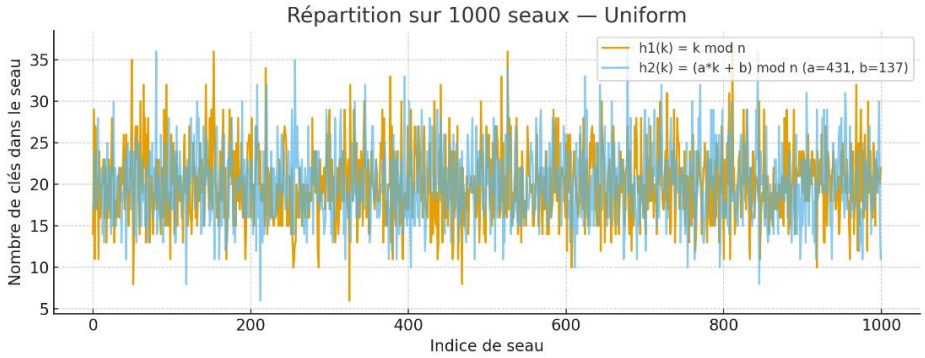


Figure 6 : Répartition des données uniformes dans les sceaux

Cependant, pour un jeu de données « mixte » (70% multiples de 1000 + 30% bruit uniforme), on observe un gros pic sur un seau (les multiples) et un bruit réparti ailleurs. L'effet est identique pour h1 et h2. La transformation linéaire h(k) = (a·k+b) mod n est peu coûteuse et facile à stocker, mais n'améliore pas une distribution déjà dégénérée modulo n : elle re-étiquette simplement les seaux :

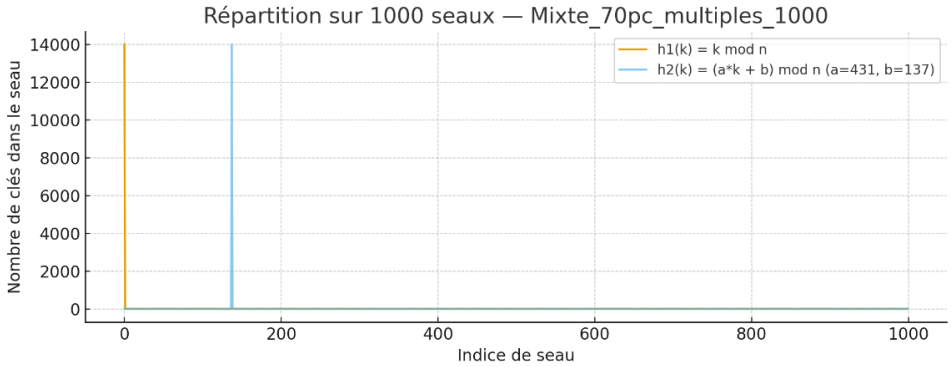


Figure 7 : Répartition des données « mixtes » dans les sceaux

Cela ne veut pas dire que les fonctions de hachage de ce type sont inutilisables en pratique, mais il faut bien faire attention avec quelles données elles sont associées.

IV.6.5. Code de hachage et fonction de compression

On peut voir la conception d'une fonction de hachage comme composée de deux parties distinctes. Par définition, une fonction de hachage prend en entrée un élément de l'univers (des objets) comme une adresse IP, un nom, … et on obtient un numéro de compartiment.

On prend donc d'abord un objet qui n'est pas intrinsèquement numérique (par exemple une chaîne de caractères ou quelque chose de plus abstrait) et on le transforme d'une façon ou d'une autre en un nombre. Ensuite, ce nombre est mappé vers l'indice d'un compartiment.

Dans certains cas, on donne à ces deux étapes des noms distincts : la première étape consiste à formuler le **code de hachage** d'un objet, et la seconde à appliquer une **fonction de compression**. La première étape n'est pas toujours obligatoire. Par exemple, si les clés sont des numéros de sécurité sociale ou des numéros de téléphone, ce sont déjà des entiers.

Mais il existe des applications où les objets ne sont pas numériques : par exemple, des chaînes de caractères, des noms à mémoriser, etc. Dans ce cas, la production du code de hachage revient essentiellement à écrire une sous-routine qui prend en entrée une chaîne et produit en sortie un nombre.

Il existe des méthodes pour faire cela : chaque caractère d'une chaîne peut être vu comme un nombre correspondant à son code ASCII (ou Unicode) et il suffit d'agréger tous ces nombres (un par caractère) en un nombre global. Une façon de faire (méthode de hachage polynomial) est d'itérer sur les caractères un par un et de maintenir une somme courante. À chaque caractère, on multiplie la somme par une constante, on ajoute le nouveau caractère, puis, si nécessaire, on prend un modulo pour éviter les dépassements :

1. Encoder chaque caractère en un entier (ASCII/Unicode)
2. Itérer sur la chaîne : on maintient une valeur k
3. À chaque caractère de code x, on met à jour :

$$k_i = (k_{i-1} \cdot B + x_i) \bmod M$$

B : base (une constante > taille de l'alphabet, ex. 131 ou 257, en évitant les puissances de 2).
M = modulo (grand entier pour éviter les débordements et réduire les collisions) (on peut prendre un grand nombre premier M = 1 000 000 007, ou bien s'appuyer sur le débordement 64 bits non signé en prenant M = 2^{64})

C'est une méthode rapide en O(n) qui tient en quelques lignes et si B et M sont choisis correctement, elle répartit en général bien les chaînes. De plus, la forme $h = h \cdot B + x$ est exactement l'évaluation d'un polynôme en base B : $h = x_0 \cdot B^{n-1} + x_1 \cdot B^{n-2} + \ldots + x_{n-1}$, calculé efficacement avec la règle de Horner.

Reste donc la question de la conception de la fonction de compression. Peut-être que les clés sont déjà numériques (numéros de sécurité sociale, adresses IP), ou qu'elles sont le résultat de l'application d'une sous-routine qui convertit une chaîne de caractères en un grand nombre. Dans tous les cas, il y a un grand nombre de données (plusieurs millions ou milliards) qu'il faut transformer en l'un des compartiments disponibles (peut-être qu'il y en a mille environ). La manière la plus simple de faire cela est quelque chose que nous avons déjà vu : prendre le reste modulo le nombre de compartiments :

4. Pour obtenir un numéro de compartiment entre 0 et n-1, on peut appliquer ensuite la fonction de compression à base de modulo vue au chapitre IV.6.4 :

$$h(k) = k \bmod n$$

IV.6.6. Choix du nombre de compartiments

La fonction de compression précédente est aussi simple à coder qu'à évaluer. C'était notre deuxième objectif pour une fonction de hachage : elle ne doit rien stocker (il n'y a effectivement rien à stocker) et elle doit être rapide à évaluer.

Le problème, c'est que la première propriété souhaitée d'une fonction de hachage est de bien répartir les données. Or, si le nombre de compartiments n est mal choisi, on peut échouer sur ce premier critère et, par conséquent obtenir une mauvaise fonction de hachage.

Reprenons l'exemple du hachage polynomial, et imaginons que le nombre de compartiments n soit pair. Le résultat de chaque étape de hachage k est dans l'intervalle [0, M-1] et pour affecter chaque valeur de k à un compartiment parmi n, on calcule h = k mod n. La valeur du pas de hachage k peut donc s'écrire $k = q \cdot n + r$, avec $0 \leq r < n$, où r est exactement le numéro du compartiment. Or, puisque n est un entier pair, $q \cdot n$ est pair et la parité de r est donc la même que celle de k. Ainsi, si par malchance les régularités dans les données manipulées par le code de hachage conduisent à produire des clés uniquement paires, tous les emplacements impairs resteront vides.

Il existe des règles empiriques pour choisir le nombre de compartiments (le modulo) afin de limiter les problèmes susceptibles de survenir lorsqu'on utilise une fonction de compression de type modulo.

D'abord, on veut éviter d'avoir à coup sûr des compartiments vides, et cela quel que soit le jeu de données. Dans l'exemple précédent, le problème était que toutes les clés étaient divisibles par 2, et que le nombre de compartiments l'était aussi. Comme ils partageaient un facteur commun (2), cela garantissait que tous les compartiments impairs restaient vides.

Plus généralement, si les clés partagent un facteur commun avec le nombre de compartiments n, c'est problématique. Il faut donc que n ne partage aucun facteur avec les clés, et donc avec les données hachées. Pour réduire les facteurs communs, on peut choisir n avec très peu de facteurs, autrement dit, choisir n premier.

Le nombre de compartiments devrait aussi être du même ordre de grandeur que la taille de l'ensemble à stocker (un facteur constant proche suffit). Et on peut toujours trouver un nombre premier dans un intervalle proche de la cible. Si n n'est pas trop grand (milliers ou dizaines de milliers), on peut consulter une liste de nombres premiers et en choisir un de l'ordre de grandeur souhaité. Si l'on a besoin d'un n très grand (millions et plus), il existe des tests de primalité qui permettent d'en trouver un dans la plage voulue.

Il existe aussi des optimisations de second ordre. Il ne faut pas que le nombre premier choisi soit trop proche de motifs présents dans les données. Dans l'exemple des numéros de téléphone, des motifs apparaissaient en base 10 (forte concentration sur les premiers chiffres à cause de l'indicatif). Pour les adresses mémoire exprimées en base 2, on observe de fortes corrélations dans les bits de poids faible. Ce sont les deux cas les plus courants : des motifs dans certaines positions en base 10 ou en base 2. Cela suggère que n ne devrait pas être trop proche d'une puissance de 2 ni d'une puissance de 10.

V) FACTEUR DE CHARGE ET PERFORMANCES

Il n'existe pas de solution miracle en matière de conception de tables de hachage mais il y a certaines recommandations qu'il faut respecter. Les plus importantes sont :

- gérer le facteur de charge de la table de hachage ;
- utiliser une fonction de hachage moderne et bien testée ;
- tester plusieurs implémentations concurrentes afin de déterminer la meilleure pour une application particulière.

V.1. Charge et performances d'une table de hachage

Les performances d'une table de hachage se dégradent à mesure qu'elle se remplit : avec le chaînage, les listes des seaux s'allongent ; avec l'adressage ouvert, il devient plus difficile de trouver une case vide.

V.1.1. Charge d'une table de hachage

Le « **taux de remplissage** » d'une table de hachage est mesuré via son facteur de charge :

$$\text{charge de la table} = \frac{\text{nombre d'objets enregistrés dans la table}}{\text{longueur du tableau n}} = \frac{|S|}{n}$$

Par exemple, dans une table de hachage avec chaînage, le facteur de charge correspond à la population moyenne d'un seau de la table.

V.1.2. Performances avec la gestion par chaînage

Dans une table de hachage avec chaînage, le temps d'exécution d'une opération de recherche ou de suppression évolue avec la longueur des listes des seaux. Dans le meilleur des cas, la fonction de hachage répartit parfaitement les objets entre les seaux. Avec un facteur de charge α, ce scénario idéalisé aboutit à au plus α objets par compartiment. Les opérations de recherche et de suppression ne prennent alors qu'un temps en $O(\alpha)$, et sont donc en temps constant dès lors que $\alpha = O(1)$.

Comme les bonnes fonctions de hachage répartissent la plupart des jeux de données de manière à peu près uniforme entre les seaux, cette performance « dans le meilleur des cas » est approximativement atteinte par des implémentations pratiques de tables de hachage à chaînage (avec une bonne fonction de hachage et des données non pathologiques).

V.1.3. Performances avec la gestion par adressage ouvert

Dans une table de hachage avec adressage ouvert, le temps d'exécution d'une opération de recherche ou d'insertion évolue avec le nombre de sondages nécessaires pour trouver une case vide (ou l'objet recherché). Lorsque le facteur de charge est α, il y a $\alpha \cdot n$ cases pleines et $n \cdot (1-\alpha)$ cases vides.

Dans le meilleur des cas, chaque sondage est décorrélé du contenu de la table et a une probabilité $(1-\alpha)$ de trouver une case vide. Dans ce scénario idéalisé, le nombre moyen de

sondages requis est $1/(1-\alpha)$. Si α reste suffisamment loin de 1, par exemple autour de 70 %, le temps d'exécution idéal de toutes les opérations est en O(1). Cette performance « dans le meilleur des cas » est approximativement atteinte par des tables implémentées en pratique avec double hachage ou d'autres séquences de sondage sophistiquées.

Avec le sondage linéaire, les objets ont tendance à s'agglutiner dans des cases consécutives, ce qui ralentit les opérations : il a été montré que le nombre de sondages requis est autour de $1/(1-\alpha)^2$, même dans le cas idéalisé. Cela reste en O(1) tant que α est nettement inférieur à 100 %.

V.2. Gestion du facteur de charge des tables de hachage

Pour garder un facteur de charge à une valeur cible, une implémentation de table de hachage doit mettre à jour le nombre de places disponibles dans la table. Une bonne règle empirique consiste à redimensionner périodiquement le tableau de la table afin que le facteur de charge reste en dessous de 70 % (voire moins, selon l'application et la stratégie de résolution des collisions). Ainsi, avec une fonction de hachage bien choisie et des données non pathologiques, la plupart des stratégies de résolution des collisions conduisent généralement à des opérations en temps constant sur la table de hachage.

La façon la plus simple d'implémenter le redimensionnement du tableau consiste à suivre le facteur de charge et, dès qu'il atteint 70 %, à doubler le nombre n de seaux. Tous les objets sont alors re-hachés dans la nouvelle table, plus grande (qui a désormais un facteur de charge de 35 %). En option, si une série de suppressions fait suffisamment baisser le facteur de charge, on peut réduire la taille du tableau pour économiser de l'espace (en re-hachant tous les objets restants dans la table plus petite). De tels redimensionnements peuvent être coûteux en temps, mais dans la plupart des applications ils restent assez rares.

V.3. Choisir une bonne fonction de hachage

Concevoir de bonnes fonctions de hachage est difficile. Heureusement, plusieurs programmeurs astucieux ont mis au point un éventail de fonctions de hachage bien testées et publiquement disponibles.

Comme différentes fonctions de hachage se comportent mieux selon les distributions de données, l'idéal est de comparer les performances de plusieurs fonctions dans une application et un environnement d'exécution spécifique. De bons points de départ à explorer incluent FarmHash, MurmurHash3, SpookyHash et MD5. Ce sont toutes des fonctions de hachage non cryptographiques, qui ne sont pas conçues pour se protéger contre des attaques (par exemple des attaques par déni de service consistant à envoyer des valeurs de paquets TCP/IP différentes mais conçues pour se hacher dans le même seau d'une table de hachage intégrée à un système réseau).

Les fonctions de hachage cryptographiques sont plus complexes et plus lentes à évaluer que leurs homologues non cryptographiques, mais elles protègent contre de telles attaques. Un bon point de départ ici est la fonction SHA-1 et ses parentes plus récentes comme SHA-256.

Une autre possibilité est de construire une famille bien pensée de fonctions de hachage et de les choisir au hasard. L'objectif est que, pour tout jeu de données, en moyenne (par rapport au choix aléatoire à l'exécution de la fonction de hachage), la fonction se comporte bien, au sens où elle répartit uniformément les données. Ainsi, là où l'on disait que, si l'on fixe une seule fonction de hachage, il existe un jeu de données qui la met en défaut, on peut dire que pour chaque jeu de données fixé, un choix aléatoire d'une fonction dans la famille fonctionnera bien en moyenne sur ce jeu de données.

Cela n'empêche pas de rendre le programme open-source : on peut publier du code qui expose la famille de fonctions de hachage et qui, à l'exécution, en choisit une aléatoirement. Le point essentiel est que, même en inspectant le code, on ne saura pas quels ont été les choix aléatoires effectifs effectués au moment de l'exécution ; on ne connaîtra donc pas la fonction de hachage réellement utilisée et on ne pourra pas reconstituer un jeu de données pathologique adapté à ce choix en temps réel.

Nous détaillerons cette seconde solution dans le dernier chapitre de ce cours.

V.4. Choisir la stratégie de gestion des collisions

Le chaînage consomme plus d'espace que l'adressage ouvert, donc ce dernier peut être préférable lorsque l'espace est une contrainte de premier ordre. Les suppressions sont plus compliquées avec l'adressage ouvert qu'avec le chaînage, de sorte que le chaînage peut être préférable dans les applications avec beaucoup de suppressions.

Comparer le sondage linéaire à des implémentations d'adressage ouvert plus sophistiquées comme le double hachage est également délicat. Le sondage linéaire entraîne de plus gros amas d'objets consécutifs dans la table de hachage et donc davantage de sondages que des approches plus élaborées, mais ce coût peut être compensé par une bonne gestion de la mémoire de l'environnement d'exécution.

Comme pour le choix d'une fonction de hachage, pour du code critique, rien ne remplace le fait d'implémenter plusieurs solutions concurrentes et de voir laquelle fonctionne le mieux pour une application donnée.

VI) LES FONCTIONS DE HACHAGE UNIVERSELLES

Nous avons vu qu'on ne peut pas avoir une seule fonction de hachage qui se comporte toujours bien pour tous les jeux de données, car toute fonction de hachage fixe admet un jeu de données pathologique. Nous allons étudier ici la solution randomisée : utiliser une **famille de fonctions de hachage** et, au moment de l'exécution, en choisir une **aléatoirement**. Avec cette approche, on a la garantie d'avoir de bonnes performances en moyenne, quel que soit le jeu de données.

Nous commencerons par proposer une définition mathématique d'une « bonne » fonction de hachage aléatoire et par définir formellement ce qu'est une famille universelle de fonctions de hachage. Ensuite, nous verrons qu'il existe des exemples simples et faciles à calculer qui satisfont cette définition. Enfin, nous étudierons les performances du hachage

(avec chaînage) lorsque l'on utilise du hachage universel. Nous montrerons que si l'on choisit au hasard une fonction dans une famille universelle, alors l'espérance du coût de toutes les opérations est constante (en supposant, bien sûr, que le nombre de compartiments est du même ordre de grandeur que le nombre d'objets dans la table, condition que nous avons déjà vue comme nécessaire pour obtenir de bonnes performances).

VI.1. Définition mathématique d'une « bonne » fonction de hachage

Pour cette définition, on suppose l'univers des clés fixé (adresses IP, prénoms d'amis, configurations d'un échiquier, etc.) et on suppose que le nombre de compartiments n est fixé. On appelle **l'ensemble H universel** s'il vérifie la condition suivante : pour chaque paire d'éléments distincts, la probabilité qu'ils entrent en collision (même compartiment) ne doit pas être plus grande que sous le « graal » du hachage parfaitement uniforme au hasard.

Autrement dit, pour toutes clés distinctes (x, y), si l'on choisit au hasard une fonction $h \in H$, la probabilité que x et y hachent dans le même compartiment est au plus 1/n (où n est le nombre de compartiments).

Définition : Soit $\mathcal{H}$ l'ensemble des fonctions de hachage de $U \rightarrow \{0, 1, 2, 3, \ldots, n-1\}$ (n est le nombre d'emplacements).
On dit que $\mathcal{H}$ est universel si et seulement si, pour tout $x, y \in U$ $(x \neq y)$, $h \sim \text{Unif}(\mathcal{H})$, on a :

$$P\left[h(x) = h(y)\right] \leq \frac{1}{n}$$

Ainsi :

- chaque clé a au plus 1/n chance d'être envoyée vers n'importe quel compartiment ;
- la probabilité d'avoir une collision est au plus de 1/n.

On choisit 1/n parce que, dans l'idéal théorique (mais inutilisable en pratique), on pourrait affecter indépendamment et uniformément à chaque clé un compartiment aléatoire : si Alice tombe dans le compartiment 17, alors Bob, choisi indépendamment et uniformément parmi n compartiments, ne la percute qu'avec probabilité 1/n.

> *Exemple : Soit une famille de fonctions de hachage $\mathcal{H}$ de $U \rightarrow \{0, 1, 2, \ldots, n-1\}$. Supposons que $\mathcal{H}$ possède la propriété suivante : pour chaque emplacement i et clé k, il y a une proportion 1/n des fonctions de $\mathcal{H}$ dont l'image de k est i. Est-ce que $\mathcal{H}$ est universel ?*

Si $\mathcal{H}$ est l'ensemble de toutes les fonctions de l'univers, alors chaque clé a exactement 1/n chances d'être envoyée vers n'importe quel compartiment. Choisir h dans $\mathcal{H}$ au hasard revient à choisir une fonction complètement aléatoire et donc la probabilité de collision de deux clés distinctes est exactement 1/n. $\mathcal{H}$ est donc universel.

Si $\mathcal{H}$ est une famille minuscule de n fonctions constantes, où h_0 envoie toutes les clés dans le compartiment 0, h_1 dans le compartiment 1, … jusqu'à n-1. Pour une clé et un compartiment fixé (disons 31), la probabilité qu'une fonction tirée au hasard envoie la clé dans 31 est bien 1/n (il suffit de tomber sur la fonction constante h_{31}). Mais cette famille n'est pas universelle, car les collisions sont systématiques pour chaque fonction : $\Pr[h_1(x) = h_1(y)] = 1$.

VI.2. Exemple : hachage d'adresses IP

VI.2.1. Construction de l'ensemble universel

Considérons que l'univers U soit l'ensemble des adresses IP. Une adresse IP est un entier sur 32 bits, soit quatre octets (quatre entiers de 0 à 255) :

$$IP = (x_1, x_2, x_3, x_4)\text{, xi} \in \{0, 1, \ldots, 255\}$$

La fonction de hachage que nous allons construire est proche des fonctions « rapides et simples » à base de modulo vues précédemment, mais cette fois-ci nous utiliserons une famille universelle. Nous utiliserons la même compression : modulo un nombre premier de compartiments. La différence est que nous multiplierons les x_i par des coefficients aléatoires et prendrons une combinaison linéaire aléatoire de x_1, x_2, x_3, x_4.

Choisissons un nombre de compartiments n > 255, n premier, et de l'ordre de deux fois le nombre d'objets stockés. Si l'on veut mémoriser 500 adresses IP, on peut prendre le nombre premier n = 997. Nous verrons pourquoi ce choix sur n est important par la suite.

Pour construire notre ensemble $\mathcal{H}$, on définit une fonction de hachage $h_a \in \mathcal{H}$ par un 4-uplet. On fait en sorte qu'il y ait directement une correspondance avec les indices des compartiments en choisissant les $a_i \in \{0, \ldots, n-1\}$:

$$a = (a_1, a_2, a_3, a_4)\text{, } a_i \in \{0, \ldots, n-1\}$$

… et on obtient ainsi n^4 fonctions.

Pour calculer le hash d'une adresse IP x = (x_1, x_2, x_3, x_4), on réalise le produit scalaire entre l'adresse IP et la fonction de hachage h_a, et on réduit le résultat obtenu module n afin d'obtenir des indices dans le bon intervalle pour le compartiment :

$$h_a(x) = h_a \cdot x = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \bmod n$$

Cette opération se fait en temps constant (4 multiplications, 3 additions et 1 modulo) et espace constant (on mémorise seulement a_1, a_2, a_3 et a_4).

VI.2.2. Preuve que la famille est universelle

Par définition, une famille universelle exige que, pour chaque paire de clés distinctes (ici deux adresses IP), la probabilité (sur le choix aléatoire de $h \in \mathcal{H}$) qu'elles entrent en collision soit au plus 1/n, comme avec un hachage parfaitement aléatoire.

Fixons deux paires de clés distinctes (x, y) et supposons qu'elles diffèrent sur la quatrième composante : $x_4 \neq y_4$ (le choix de la composante n'a pas d'importance ; il suffirait de traiter 4 cas symétriques) :

$$x = (x_1, x_2, x_3, x_4)\text{ ; }y = (y_1, y_2, y_3, y_4)\text{ ; }x_1 = y_1, x_2 = y_2, x_3 = y_3 \text{ et } x_4 \neq y_4$$

On va chercher la fraction des choix de (a_1, a_2, a_3, a_4) qui provoquent une collision :

$$h_a(x_1, x_2, x_3, x_4) = h_a(y_1, y_2, y_3, y_4)$$

Écrivons cette condition de collision en une forme pratique :

$$h_a(x_1, x_2, x_3, x_4) = h_a(y_1, y_2, y_3, y_4)$$
$$\Leftrightarrow (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \bmod n = (a_1y_1 + a_2y_2 + a_3y_3 + a_4y_4) \bmod n$$
$$\Leftrightarrow a_4(x_4 - y_4) \bmod n = \sum_{i=1}^{3} a_i(y_i - x_i) \bmod n$$

Dans la définition d'une fonction de hachage universelle, on analyse les probabilités de collisions lorsque le choix des fonctions de hachage est aléatoire. Dans notre exemple, choisir une fonction de hachage aléatoirement revient à choisir a_1 et a_2 et a_3 et a_4 de manière aléatoire, et donc de réaliser quatre choix aléatoires.

On va utiliser le **principe des décisions différées**, qui consiste à ne « révéler » qu'une partie de l'aléa à la fois :

> *Soient des tirages aléatoires R_1, …, R_k (indépendants ou non) et un évènement E qui dépend de R_1, …, R_k . Si pour tout choix fixé (r_1, …, r_{k-1}) on a :*
>
> $$\Pr[E \mid R_1 = r_1, \ldots, R_{k-1} = r_{k-1}] \leq \alpha$$
>
> *alors :*
>
> $$\Pr[E] \leq \alpha$$
>
> *On « diffère » la révélation de R_k. On conditionne d'abord sur R_1, …, R_{k-1} (les « décisions » déjà révélées), on borne la probabilité résiduelle en faisant varier seulement R_k, puis on moyenne.*

Fixons arbitrairement les résultats obtenus sur les trois premiers (a_1, a_2, a_3) et laissons a_4 aléatoire. On veut montrer qu'au plus une fraction égale à 1/n des choix de (a_1, a_2, a_3) valide l'équation précédente :

$$a_4(x_4 - y_4) \bmod n = \sum_{i=1}^{3} a_i(y_i - x_i) \bmod n$$

Si (a_1, a_2, a_3) est fixé, alors on a :

$$a_4(x_4 - y_4) \bmod n = b, \; b \in \{0, 1, \ldots, n-1\}$$

Comme $x_4 \neq y_4$ et n > (max(x_i) = 255), alors $(x_4 - y_4) \not\equiv 0 \pmod n$. De plus comme n est premier, $(x_4 - y_4)$ est **inversible modulo n** et l'équation $a_4 \cdot (x_4 - y_4) \bmod n$ a exactement une solution en a_4 dans {0, …, n – 1}. Puisque a4 ~ Unif ({0, 1, …, n – 1}), on obtient :

$$\Pr[h_a(x) = h_a(y) \mid a_1, a_2, a_3] = \frac{1}{n}$$

Cette borne étant valable pour tout choix fixé de (a_1, a_2, a_3), d'après le principe des décisions différées, on obtient :

$$\Pr[h_a(x) = h_a(y)] \leq \frac{1}{n}$$

VI.3. Complexité temporelle avec gestion par chaînage

VI.3.1. Coût en moyenne par opération

Nous allons montrer que si l'on choisit une fonction de hachage uniformément au hasard dans une famille universelle (comme celle vue précédemment), alors on a la **garantie d'obtenir un temps d'exécution constant (en moyenne)** pour toutes les opérations prises en charge.

L'univers des clés est fixé (par exemple des adresses IP) et le nombre de compartiments est également fixé (par exemple 10 000). Nous nous concentrerons uniquement sur les tables de hachage implémentées avec chaînage. Nous allons analyser les performances (en moyenne) sur le choix aléatoire de la fonction h tirée uniformément d'une famille universelle $\mathcal{H}$. Par exemple, pour la famille que nous avons construite précédemment, cela revient simplement à choisir au hasard et uniformément les coefficients d'une fonction linéaire.

Ce théorème, ainsi que la définition des fonctions de hachage universelles, remontent à un article de recherche de 1979, de Carter et Wegman. L'idée du hachage existait déjà depuis les années 1950, mais cela montre que parfois les idées ont besoin de mûrir avant qu'on ne trouve la bonne façon de les formaliser. Carter et Wegman ont proposé une approche élégante et modulaire pour penser la performance du hachage à travers cette notion de hachage universel.

Quelques remarques importantes sur la garantie en temps que nous allons obtenir :

- Cette garantie est une espérance, c'est-à-dire une moyenne sur le choix de la fonction de hachage h. Cependant, cette espérance est valable pour n'importe quel ensemble de données : on ne fait aucune hypothèse sur les données, et le temps moyen reste garanti.
- Pour obtenir des performances constantes, il faut aussi contrôler le taux de charge α de la table et faire en sorte qu'il soit constant : $\alpha = O(1)$
- Enfin, il faut que l'évaluation de la fonction de hachage elle-même prenne un temps constant, d'où l'importance des fonctions simples vues précédemment.

Même si les tables de hachage prennent en charge diverses opérations, nous pouvons simplement chercher à borner le temps d'exécution d'une recherche infructueuse, ce qui sera suffisant pour borner le temps d'exécution de l'ensemble des opérations.

Dans une table de hachage avec chaînage, on commence par hacher la clé pour trouver le compartiment approprié, puis on effectue l'insertion, la suppression ou la recherche correspondante dans la liste de ce compartiment. Le pire cas est donc celui où on cherche un élément qui n'est pas là, car il faut examiner chaque élément de la liste, un par un, jusqu'à la fin, avant de pouvoir conclure que la recherche a échoué. Bien sûr, comme nous l'avons vu, l'insertion est toujours en temps constant ; la suppression et les recherches réussies, elles, peuvent s'arrêter plus tôt, avant la fin de la liste, si on trouve l'élément.

Appelons $\mathcal{S}$ l'ensemble des données, c'est-à-dire les objets que nous stockons dans notre table de hachage. Nous supposons que la taille de $\mathcal{S}$, notée $|\mathcal{S}|$, est comparable au nombre de compartiments n ($|\mathcal{S}| = O(n)$) afin de garantir un facteur de charge constant.

Supposons maintenant que nous cherchions un objet x qui n'est pas dans $\mathcal{S}$. On peut décrire le temps d'exécution de cette recherche infructueuse en deux parties :

- D'abord, on évalue la fonction de hachage pour déterminer le bon compartiment (en supposant encore une fois que la fonction de hachage est simple et que cette évaluation prend un temps constant). Cela prend un temps en O(1).
- Ensuite, on accède directement au bon compartiment, puis on parcourt la liste, pour finalement échouer à trouver l'objet recherché. Parcourir une liste prend un temps proportionnel à la longueur de la liste.

Nous retrouvons donc quelque chose que nous avons déjà mentionné : le temps d'exécution des opérations d'une table de hachage avec chaînage est régi par la longueur des listes. Appelons cette longueur $\mathcal{L}$.

La longueur de la liste $\mathcal{L}$ dépend du choix de la fonction de hachage h que nous faisons au hasard. C'est donc une variable aléatoire qui peut être aussi petite que 0. Notre problème (le temps attendu de la recherche infructueuse) peut donc se réduire à déterminer l'espérance de la variable aléatoire $\mathcal{L}$, c'est-à-dire la longueur moyenne de la liste dans le compartiment de x.

Pour un élément $y \in \mathcal{S}$ ($x \neq y$ car $x \notin \mathcal{S}$), on définit la variable indicatrice Z_y :

$$Z_y = \begin{cases} 1 & \text{si } h(y) = h(x) \\ 0 & \text{sinon} \end{cases}$$

Autrement dit, pour un y donné dans le jeu de données $\mathcal{S}$, on fixe x et y (par exemple deux adresses IP distinctes), et selon la fonction de hachage choisie h, ces deux éléments distincts peuvent ou non être mappés dans le même compartiment.

On peut maintenant écrire la longueur de la liste $\mathcal{L}$ (nombre d'éléments qui entrent en collision avec x) que nous voulons analyser à l'aide de cette variable indicatrice :

$$\mathcal{L} = \sum_{y \in \mathcal{S}} Z_y$$

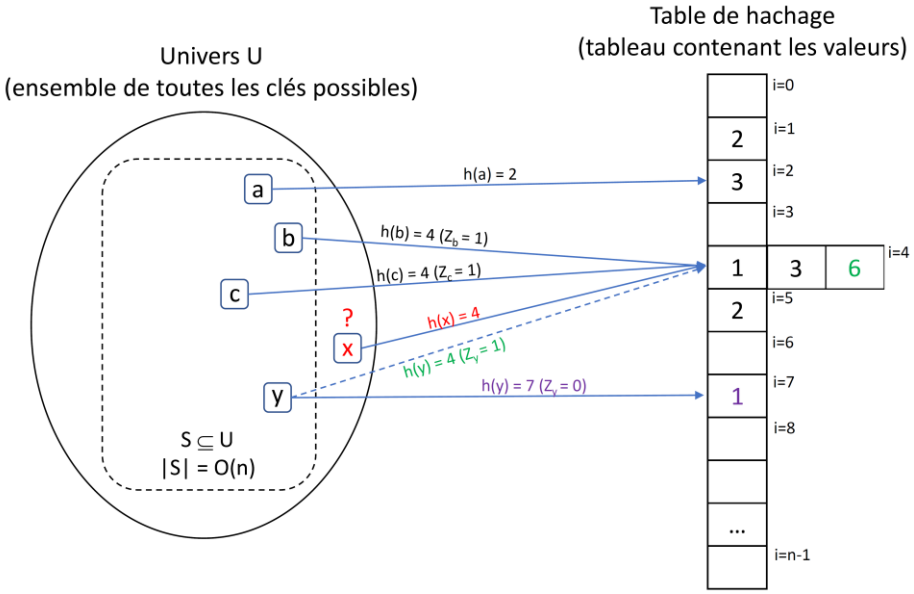


Figure 8 : Principe d'estimation de la longueur de L avec la fonction indicatrice Z_y

Maintenant que nous avons décomposé notre variable aléatoire $\mathcal{L}$ en somme de variables indicatrices, nous pouvons appliquer la linéarité de l'espérance :

$$E\left[\mathcal{L}\right] = E\left[\sum_{y\in\mathcal{S}} Z_y\right] = \sum_{y\in\mathcal{S}} E\left[Z_y\right] = \sum_{y\in\mathcal{S}} \left(0\cdot \Pr\left[Z_y = 0\right] + 1\cdot \Pr\left[Z_y = 1\right]\right)$$
$$= \sum_{y\in\mathcal{S}} \left(\Pr\left[Z_y = 1\right]\right) = \sum_{y\in\mathcal{S}} \left(\Pr\left[h(y) = h(x)\right]\right)$$

Or, puisque h $\in \mathcal{H}$, par définition on a : $\Pr\left[h(y) = h(x)\right] \leq \frac{1}{n}$ et donc :

$E\left[\mathcal{L}\right] \leq \sum_{y\in\mathcal{S}} \left(\frac{1}{n}\right) = \frac{|\mathcal{S}|}{n} = \alpha$, où α est la charge de la table.

Ainsi, si on fait en sorte d'avoir une charge constante α = O(1), on obtient :

$$E\left[\mathcal{L}\right] \leq \alpha$$

En résumé :

- Si la fonction de hachage est tirée uniformément au hasard d'une famille universelle,
- Si le facteur de charge $\alpha = |\mathcal{S}|/n$ est borné par une constante,
- Et si l'évaluation de h est en temps constant,

… alors toutes les opérations de la table de hachage s'exécutent en temps constant (en moyenne en O(1 + α)), quelle que soit la distribution des données.

VI.3.2. Coût amorti sur une séquence de redimensionnement

Maintenir le facteur de charge borné permet de garantir de bonnes performances. Quand on maintient α borné en redimensionnant une table contenant n éléments, on crée une nouvelle table dans la quelle on réinsère tous les anciens éléments et le ou les nouveaux :

- Le coût du hachage des éléments est O(n) = O($|\mathcal{S}|$) avec α constant ;
- Le coût de l'ensemble des opérations d'insertion est O(n·(1 + α)) = O(n) = O($|\mathcal{S}|$)

Donc le coût total sur la séquence de redimensionnement est O($|\mathcal{S}|$) et le **coût amorti** par opération (moyenne sur la séquence) est O($|\mathcal{S}|$) / $|\mathcal{S}|$ = O(1).

VI.4. Complexité temporelle avec gestion par adressage ouvert

Le chaînage n'est pas la seule manière de gérer les collisions. Il existe également la méthode d'adressage ouvert.

Cette méthode consiste à ne stocker qu'un seul objet par case, et à chercher une case vide par sondage pour insérer un nouvel objet dans la table de hachage. C'est donc différent du cas du chaînage, où nous pouvons avoir une liste arbitrairement longue dans un compartiment donné de la table de hachage.

Avec au plus un objet par case, il est évident que l'adressage ouvert n'a de sens que lorsque le facteur de charge α est inférieur à 1, c'est-à-dire lorsque le nombre d'objets stockés est inférieur au nombre de cases disponibles.

Il existe deux stratégies pour produire une séquence de sondages : le double hachage et le sondage linéaire. Le double hachage consiste à utiliser deux fonctions de hachage différentes. La première indique dans quelle case chercher en premier, et chaque fois qu'une case est pleine, on ajoute un incrément spécifié par la seconde fonction.

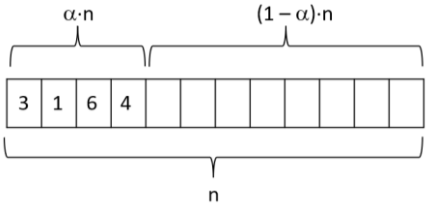
Le sondage linéaire est plus simple : il n'y a qu'une seule fonction de hachage qui indique où chercher en premier, puis on ajoute seulement 1 à l'indice du compartiment jusqu'à trouver une case vide.

Nous allons ici donner un calcul approximatif qui suggère, au moins dans un environnement idéalisé, le type de performance auquel on peut s'attendre d'une table de hachage à adressage ouvert bien implémentée, en fonction du facteur de charge α.

Plus précisément, nous allons faire l'hypothèse que nous utilisons une fonction de hachage telle que chacune des n! séquences de sondages possibles soient équiprobables. Aucune fonction de hachage ne satisfera réellement cette hypothèse, en particulier le double hachage ou le sondage linéaire, mais cela nous permettra de trouver une borne supérieure idéale, avec laquelle il sera possible de comparer les performances d'autres implémentations de tables de hachage.

Nous allons montrer que sous cette hypothèse, le temps moyen attendu pour insérer un nouvel objet dans la table de hachage est essentiellement égal à 1/(1 – α). Cela signifie que, dans ce scénario idéalisé, si on garde la charge sous contrôle (disons à 50 %), alors le temps d'insertion sera excellent : si α = 0,5, alors 1/(1 – α) = 2, donc on peut s'attendre à deux sondages en moyenne avant d'insérer avec succès un nouvel objet. Pour la recherche, ce sera au moins aussi bon que pour l'insertion car dans le pire cas (recherche infructueuse), on tombera sur une case vide, ce qui a le même coût qu'une insertion. En pratique, avec l'adressage ouvert, il faut donc garder la charge bien en dessous de 1 et éviter de dépasser 0,7.

Lors du premier sondage, on trouve une case vide avec une probabilité de $(1 - \alpha)$ et l'insertion se termine :



Mais avec une probabilité α, la case est déjà occupée, et il faut réessayer. On choisit alors une nouvelle case aléatoire, et on recommence. Encore une fois, on trouve une case vide avec une probabilité de $(1 - \alpha)$.

En réalité, cette expérience surestime légèrement le temps d'insertion attendu, car, dans une vraie table, on ne réessaie jamais la même case deux fois. On parcourt toutes les cases dans un ordre aléatoire. Mais une borne supérieure valide consiste à supposer une probabilité fixe de succès de (1 – α) à chaque sondage.

L'espérance du nombre de sondages N avant un succès est donc $E[N] = 1 + \alpha \cdot E[N]$, soit :

$$E[N] = \frac{1}{1-\alpha}$$

Ainsi, sous notre hypothèse idéalisée, le temps moyen d'insertion est borné par 1/(1 - α). Donc, tant que la charge α reste bien inférieure à 1, la table de hachage à adressage ouvert fonctionnera très rapidement.

En pratique, le temps moyen d'insertion va dépendre du type d'adressage ouvert utilisé, de la qualité de la fonction de hachage, et du caractère pathologique ou non des données. Avec le double hachage et des données non pathologiques, on peut s'attendre à observer en pratique 1/(1 – α).

Avec le sondage linéaire, en revanche, on n'aura pas cette performance, même dans un scénario totalement idéal. Le sondage linéaire est sujet à un phénomène appelé « clustering » (regroupement) : des groupes contigus de cases pleines apparaissent, ce qui dégrade les performances.

Avec le sondage linéaire, notre hypothèse idéalisée (toutes les séquences sont également probables) est profondément fausse, encore plus fausse que pour le double hachage. Mais on peut la remplacer par une hypothèse plus raisonnable (mais encore fausse). Elle consiste à supposer que les sondages initiaux sont uniformément aléatoires et indépendants entre les clés : on suppose que pour chaque clé k, l'indice du premier compartiment essayé h(k) est choisi uniformément parmi les n cases (donc avec toutes la même probabilité de 1/n) et de manière indépendante des autres clés (le choix pour k_1 n'influence pas celui pour k_2).

Une fois le premier sondage connu, tout le reste de la séquence est déterminé : en sondage linéaire, si la table a n cases, dès qu'on connaît le premier indice $i_0 = h(k)$, la séquence de sondage est déterminée par i_0, $i_0 + 1$, $i_0 + 2$, … (mod n). Donc seul le premier sondage est aléatoire ; les suivants sont purement déterministes

Sous cette hypothèse, un résultat classique datant de 1962 de Donald Knuth montre que le temps moyen d'insertion dans une table de hachage avec sondage linéaire et un facteur de charge α est en moyenne de $1/(1 – \alpha)^2$, mais reste une fonction uniquement de α, et non du nombre total d'objets dans la table. Par exemple, si la table est remplie à 50 %, le temps d'insertion moyen sera d'environ 4 sondages. Mais si elle est remplie à 90 %, cela grimpe déjà à environ 100 sondages en moyenne.

Autrement dit, le sondage linéaire n'offre pas d'aussi bonnes performances que le double hachage, mais tant que α reste borné loin de 1, il conserve des opérations en temps constant en moyenne.

Malgré le fait que les performances du sondage linéaire sont moins bonnes que celles du double hachage, il est très utilisé en pratique car son accès séquentiel exploite à fond la hiérarchie mémoire, ce qui le rend très performant en pratique (si on contrôle la charge) et très simple à implémenter et à optimiser.

VII) LES DICTIONNAIRES PYTHON

VII.1. Structure générale

La structure des **dictionnaires** est très intéressante, car elle reprend l'ensemble des notions étudiées jusqu'à présent.

Pour rappel, le rôle d'un dictionnaire Python (`dict`) est d'associer une clé à une valeur, avec des opérations de recherche, d'insertion et de suppression en O(1) en moyenne. Depuis Python 3.7, l'ordre d'insertion est garanti lors de l'itération.

Techniquement, un `dict` en Python est une table de hachage à adressage ouvert. Il utilise une structure scindée :

- une **table d'indices** (taille de puissance de 2) contenant, pour chaque case, soit `EMPTY`, soit `DUMMY` (après suppression), soit l'indice d'une entrée dans une table compacte ;
- une **table compacte d'entrées** (contiguë) qui stocke les informations permettant de retrouver les valeurs associées au clés, sous un 3-uplet (hash, ptr_clé, ptr_valeur).

Les deux tables n'ont pas le même rôle, ni la même « densité ». La table d'indices correspond à la table de hachage (au sens des compartiments). Elle est clairsemée (peut contenir des cases vides `EMPTY`) et sa taille est dynamique pour contrôler le facteur de charge et guider le sondage lors de l'adressage ouvert.

La table compacte d'entrées est un tableau dense (sans « trous »). Par défaut, chaque entrée stocke le hash, un pointeur vers la clé et un pointeur vers la valeur associée. On l'appelle « compacte » parce que ses éléments sont contigus en mémoire, compacts (un par clé), et réutilisable après mise à jour lors des redimensionnements. Cela réduit l'empreinte mémoire (pas de cases vides intercalées), améliore la localité cache (itération rapide), et permet de préserver l'ordre d'insertion (l'ordre est l'ordre des entrées).

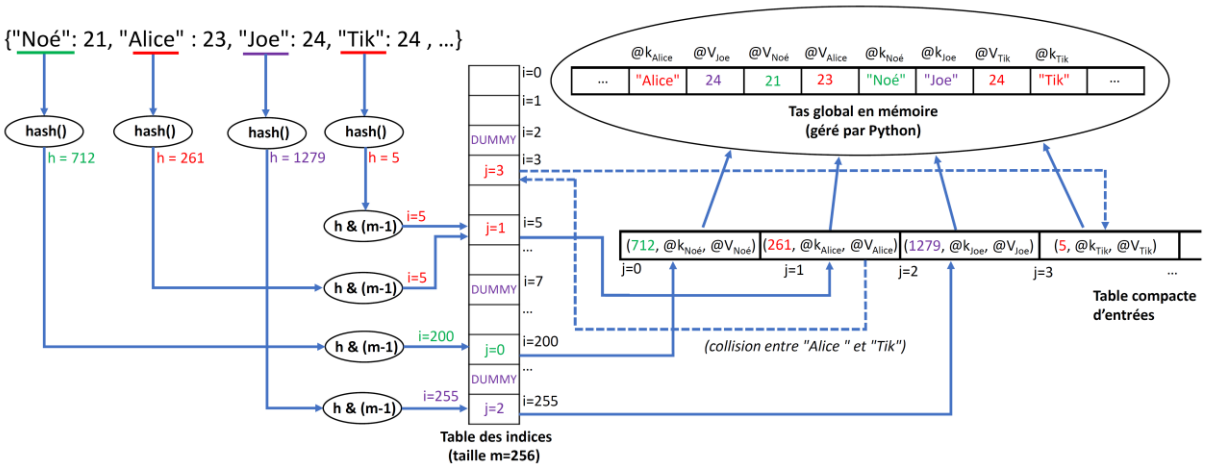


Figure 9 : Structure d'un dictionnaire Python (mode par défaut)

Le hash code est calculé avec la fonction Python `hash()` en un temps O(1) : `h = hash(clé)` qui renvoie un entier signé de la taille machine (64 bits ou 32 bits). Pour les types `str` ou `bytes`, Python utilise un hachage randomisé, qui garantit quasiment les mêmes espérances qu'un hachage de type universel mais qui est plus robuste contre les attaques de collisions.

La table de hachage a une taille $m = 2^k$ et Python compresse le hash code à l'aide d'une opération de masquage : i = h & (m – 1), plus rapide qu'une opération modulo m.

Pour gérer les collisions, Python utilise un adressage ouvert avec une suite de sondages « perturbés » dérivés du hash. Ce n'est pas un sondage linéaire simple (on n'essaie pas i + 1, i + 2, …) ni un double hachage classique (pas de seconde fonction h_2 indépendante). La suite de cases est pseudo-aléatoire mais déterministe pour un hash donné. On obtient des performances proches d'un double hachage bien choisi, avec un coût moindre et une excellente localité.

Python redimensionne tôt pour maintenir un facteur de charge bas. Celui-ci est maintenu inférieur à environ 2/3. Quand la table est trop pleine ou qu'il y a trop de `DUMMY`, il la redimensionne en doublant sa taille et en réinsérant toutes les valeurs. En moyenne, la complexité en temps des opérations est en $O(1 + \alpha)$ avec α borné et donc le redimensionnement se fait en O(1) amorti.

Les dictionnaires ont deux modes de stockage. Dans le mode « combiné » (par défaut), chaque entrée de la table compacte stocke le hash, un pointeur vers la clé et un pointeur vers la valeur. Le contenu d'une entrée est donc (hash, ptr_clé, ptr_valeur).

L'autre mode (« split dict » – aussi appelé key-sharing), permet d'optimiser le fonctionnement lorsque plusieurs dictionnaires partagent le même ensemble de clés (et donc la même table d'indices), mais qu'ils ont chacun des valeurs qui leurs sont propres. Dans ce cas, la table des indices et la table compacte sont partagées par l'ensemble des dictionnaires, mais la table compacte ne contient que les 3-uplets (hash, ptr_clé) et les valeurs sont stockées dans un autre tableau de valeurs propre à chaque dictionnaire. Il y a donc en tout une table des indices, une table compacte d'entrées et autant de tableaux de valeurs que de dictionnaires. Cela permet d'économiser de la mémoire quand beaucoup d'instances de dictionnaires partagent les mêmes clés, tout en gardant un accès très rapide aux valeurs.

Dans ce mode, lorsqu'une nouvelle clé est ajoutée à un dictionnaire qui n'est pas partagée avec les autres dictionnaires, Python peut soit cesser de partager ce dictionnaire et basculer vers le mode combiné, ou modifier la table compacte où sont stockées les (hash, ptr_clé).

VII.2. Insertion d'une valeur

Lors de l'insertion d'une valeur (`dico[clé]=valeur`), le hash de la clé est calculé avec la fonction `hash()` et l'indice de la table de hachage est déduit du hash avec l'opération de masquage. L'indice calculé ne sert qu'à choisir la case où commencer le sondage, le hash complet sera sauvegardé dans la table compacte.

Avec l'indice, la valeur stockée dans la table de hachage est retrouvée :

- Si la case est vide, l'insertion se fait sur cet emplacement : on enregistre l'index où sera stocké le 3-uplet (hash, ptr_clé, ptr_valeur) dans la table compacte.
- Si la case est `DUMMY`, on mémorise cet emplacement puis on continue à sonder.
- Si la case contient un index vers la table compacte, on vérifie si l'enregistrement existe déjà (ce qui est le cas si le hash calculé correspond à celui enregistré et que les

clés sont identiques), auquel cas on met à jour la valeur (pas de doublon). Sinon, comme il y a une collision, on sonde vers la prochaine case libre dans la table de hachage, pour être certain que l'enregistrement n'existe pas et on stocke l'index de la table compacte dans le `DUMMY` sauvegardé, ou dans la première case libre trouvée suite au sondage

La complexité de cette opération est en moyenne de O(1 + α), soit O(1) amorti si α est borné. Dans le pire des cas, le coût est de O(n).

On peut trouver cela étonnant de stocker le hash qui est sur 64 bits… mais il existe forcément des clés différentes qui partagent le même hash. Pour 300 000 clés, la probabilité qu'au moins une collision survienne est infime mais non nulle (environ 5.10^{-9}).

VII.3. Suppression d'une valeur

Pour supprimer une valeur, on calcule l'indice dans la table de hachage à l'aide des opérations {code hash + compression} puis, si un index existe dans la table de hachage, on lit les valeurs associées dans la table compacte, puis on sonde éventuellement jusqu'à trouver le bon enregistrement. Si aucun enregistrement n'est trouvé, une erreur de type `keyError` est renvoyée.

Une fois l'enregistrement trouvé, la case dans la table de hachage est marquée `DUMMY` afin de préserver la continuité de la sonde pour d'autres clés insérées après (un `EMPTY` couperait le sondage et ferait échouer des recherches légitimes).

Les valeurs du nombre de pointeurs actifs globaux (`refcount`) qui pointent vers la clé et la valeur associée dans la mémoire du tas global géré par Python sont décrémentés. Si ces `refcount` tombent à zéro, c'est que plus aucun objet python actif ne pointe vers ces valeurs en mémoire, et dans ce cas la mémoire dans le tas est libérée. L'emplacement dans la table compacte reste actif jusqu'au prochain redimensionnement.

La complexité d'une suppression vaut le coût d'une recherche, soit en moyenne O(1 + α) et donc O(1) amorti. Dans le pire des cas, en cas de collisions pathologiques, le coût est de O(n).

VII.4. Recherche d'une valeur

La procédure pour la recherche est identique à celle utilisée pour insertion d'une valeur : calcul du hash, sondage jusqu'à trouver le 3-uplet (hash, ptr_clé, ptr_val) qui correspond au même hash et à la même clé que l'élément recherché.

La complexité est encore une fois de O(1 + α) et donc O(1) amorti. Dans le pire des cas, en cas de collisions pathologiques, le coût est de O(n).