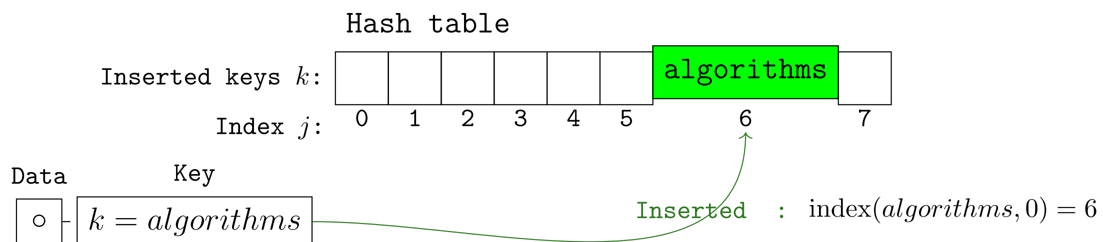
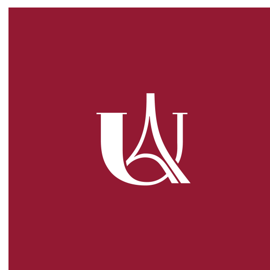
# Introduction aux Tables de Hachage

**Hugo Demaret**
hugo.demaret@polytechnique.edu

Version 1.0.0



**Université Paris Cité**
UFR de Mathématiques et d'Informatique

*Cette page est intentionnellement laissée vide.*

Je remercie sincèrement pour leur aide à la relecture de ce cours Coralie Desquiens, Mathilde Bonin, Djamel (Ouassim) Ait Moussa, Eyal Cohen, Ioana Ileana, Luc Salvon, Max Maîche, Onur Basci, Pierre Averty, Ghali Boucetta, Clément Gho, Maily Ciavaldini.

# SOMMAIRE

**1 Guide de lecture** **4**
1.1 Les informations . . . . . 4
1.2 Les exercices . . . . . 5

**2 Introduction** **6**
2.1 Problème du tableau associatif . . . . . 6
2.2 Bases du hachage . . . . . 7

**3 Chaînage séparé** **11**
3.1 Utilisation de listes chaînées . . . . . 11
3.2 Autres structures de données . . . . . 14
3.3 Exercices . . . . . 14

**4 Adressage ouvert** **17**
4.1 Linear probing . . . . . 17
4.2 Quadratic probing . . . . . 21
4.3 Pour aller plus loin . . . . . 24
4.4 Exercices . . . . . 24

# CHAPITRE 1 GUIDE DE LECTURE

Tout d'abord, ce cours n'est probablement pas vide d'erreurs : je vous encourage à me contacter si vous pensez en avoir trouvé une, et c'est avec plaisir que j'ajouterai votre nom à la liste des remerciements ! Vous pouvez trouver mes contacts sur mon site : https ://hugodemaret.fr.

## 1.1 Les informations

Dans ce cours, plusieurs informations sont entourées de boîtes colorées.
Voici leur signification !

**Les erreurs communes**

Dans les boîtes de couleur rouge sont données des informations importantes, et qui sont souvent sources d'erreurs ou de confusion. Par exemple, nous pourrions citer comme erreur commune le fait de debug avec des `printf` (rassurez-vous, on le fait tous – mais s'il vous plait utilisez un debugger).

**Définitions**

Les boites de couleur cyan donnent une définition. Elles contiennent la définition en langage naturel, ainsi que la définition mathématique du concept. Par exemple, nous pouvons définir la suite de Catalan, qui est formée d'entier naturels. Cette suite est très utilisée en combinatoire.

**Définition 1.1.1** (Nombre de Catalan). *Soit $n \in \mathbb{N}$. Le nombre de Catalan d'indice $n$ est défini par :*

$$C_n = \binom{2n}{n} \frac{1}{n+1}$$

**Anecdotes**

Les boîtes de couleur verte sont des sources d'informations complémentaires, des sortes d'anecdotes plus ou moins importantes, mais pouvant être ressorties à table, lors d'un dîner entre amis par exemple !
Ainsi, vos amis seront sûrement ravis de savoir que le nom des arbres équilibrés AVL provient des noms de leurs créateurs, à savoir Georgy **A**delson-**V**elsky et Evgenii **L**andis.

**Point implémentation**

Les boîtes de couleur orange signalent un point implémentation. Cela signifie par exemple un conseil, ou alors des informations sur des implémentations dans certains langages de programmation ou logiciels.

## 1.2 Les exercices

Les exercices et les questions peuvent avoir plusieurs niveaux de difficulté. Ces niveaux de difficulté sont donnés par des nombres de 1 à 5, 1 étant le plus facile et 5 le plus difficile.
Voici le temps approximatif que peut prendre une question selon sa difficulté (sans trop prendre en compte le temps de rédaction) :

1. La solution est immédiate
2. En moyenne, ce type de question doit prendre moins de 10 minutes
3. En moyenne, ce type de question doit prendre moins de 25 minutes
4. En moyenne, ce type de question doit prendre moins de 1 heure
5. Ce type de question peut être un mini projet

**Note**

Notez que pour les exercices d'implémentation, la notion de difficulté que nous donnons ne s'applique pas vraiment, nous l'omettrons donc.

Il existe plusieurs types d'exercices et de questions, organisé selon trois thèmes principaux :
- Des exercices plus orientés mathématiques : MATHS
- Des exercices applicatifs : APPLI
- Des exercices qui demandent d'implémenter différents concepts : IMPLE

Bien entendu, un exercice peut avoir plusieurs types.

# CHAPITRE 2

# INTRODUCTION

## 2.1 Problème du tableau associatif

Un tableau associatif[1] est un type de données abstrait qui contient une collection de paires **clef-valeur**, tel que chaque clef ne peut apparaître au plus qu'une fois. Mathématiquement, on peut représenter cela par une fonction ayant un domaine fini, et associant une clef à sa valeur.

Le problème du tableau associatif correspond au fait de trouver une structure de donnée efficace pour implémenter ce type abstrait. Il y a deux façons majeures de résoudre ce problème :

1. l'utilisation d'arbres de recherche : AVL, Red-Black trees, B-Trees, B+trees, Trie (pour les chaînes de caractères)...
2. l'utilisation de table de hachage.

Il existe beaucoup de façons différentes d'implémenter une table de hachage. L'idée générale reste toutefois la même : étant donné un tableau de taille $n$, il est simple d'accéder à une case du tableau en connaissant son index : cela peut être fait en temps constant si les tableaux sont une zone contigüe dans la mémoire. La table de hachage correspond à l'association d'un tableau et d'une fonction de hachage, qui à chaque clef associe un indice unique dans le tableau. La case mémoire où une clef peut être insérée est appelée "bucket". La fonction de hachage calcule, en fonction de la clef, sa localisation dans le tableau. Étant déterministe, une même fonction de hachage assocociera toujours une même clef à un même indice dans le tableau. Cela permet d'insérer la paire clef-valeur dans le tableau, et ensuite de la retrouver seulement à l'aide de la clef, en temps constant en moyenne.

$\Phi$

**Principe des tiroirs**

Le principe dit la chose chose suivante : si on a $n+1$ chaussettes et $n$ tiroirs, et que l'on cherche à ranger les chaussettes dans les tiroirs, on aura nécessairement au moins un tiroir qui contiendra au moins deux chaussettes. Mathématiquement, on peut le définir de la manière suivante :

**Lemme 2.1.1** (Principe des tiroirs)**.** *Soient $E$ et $F$ deux ensembles finis, tels que $|E| > |F|$. Alors il n'existe pas d'application injective de $E$ dans $F$.*

L'ensemble $E$ représente les chaussettes, et $F$ les tiroirs.

La conception des tables de hachage soulève toutefois un problème important. Le domaine étant fini, il existe nécessairement (par application du principe des tiroirs) des clefs qui ont la même

1. souvent appelé map ou dictionnaire

image par la fonction de hachage, c'est-à-dire le même indice. On appelle un tel évènement une *collision*. On peut montrer qu'un tel évènement est loin d'être rare à l'aide du problème des anniversaires. La stratégie de résolution des collisions, c'est-à-dire la façon dont les clefs qui entrent en collision sont rangées dans le tableau, est un vaste domaine de recherche, et beaucoup de manières différentes de procéder ont été proposées. Nous expliquerons différentes manières de faire dans les chapitres qui suivent, mais sachez que ces méthodes se classent souvent en deux grandes approches : le chaînage séparé, et l'adressage ouvert.

**Ensembles**

On peut implémenter les ensembles comme des dictionnaires dont les valeurs sont les clefs, la valeur `NULL`...
Dans le cas où l'on travaille avec des ensembles, il peut être important d'implémenter certaines opérations importantes, telles que l'union, l'intersections, la différence ensembliste... Nous ne rentrerons pas dans les détails d'une telle implémentation car cela dépasse le cadre de ce cours, mais n'hésitez pas à vous renseigner, c'est très intéressant (et en lien avec les bases de données !).

## 2.2 Bases du hachage

Une *table de hachage* (hash table, en anglais), est une structure de données stockant les informations sous la forme de paires clef-valeur, tel qu'une clef ne peut apparaitre au plus qu'une fois. En général, les tables de hachage implémentent trois opérations de base : *insertion*, *deletion*, *lookup*. L'insertion correspond au fait d'insérer, c'est-à-dire ajouter une valeur dans la table. La deletion correspond au fait de supprimer une valeur de la table. Le lookup correspond au fait de trouver une valeur dans la table.

**Facteur de charge**

On appelle *facteur de charge* (load factor), le ratio $\alpha = M/n$, avec $n$ la taille de la table (en nombre de buckets), et $M$ le nombre d'éléments dans la table.

### 2.2.1 Fonction de hachage

Les fonctions de hachage permettent de déterminer l'indice où insérer un élément. Il existe plusieurs familles de fonctions de hachage, et nous ne ferons pas de liste exhaustive. Nous présenterons deux méthodes, et en détaillerons une seule. Toutefois, nous vous encourageons à vous informer sur sujet, et à aller plus loin : la théorie derrière la conception des fonctions de hachage est passionante, et fait appel à de nombreuses diciplines mathématiques : probabilités, théorie des nombres, arithmétique, théorie des groupes...

Les fonctions de hachage étant déterministes, il est impossible de générer du hasard grâce à celles-ci. Toutefois, lorsque l'on en conçoit une, on essaie de conserver l'*effet avalanche*. Pour satisfaire cet effet, la modification d'un seul bit doit changer drastiquement le résultat. Notez aussi qu'en général, les clefs ne sont pas des entiers. Afin de contrecarrer cela, on hache la clef grâce à une fonction de hachage spéciale qui la transforme en entier, puis on utilise cet entier

pour le calcul de l'indice.

Souvent, dans l'analyse des tables de hachages, on suppose que la fonction de hachage distribue les valeurs uniformément, c'est-à-dire qu'il n'existe pas de bucket privilégié. S'il y a $n$ buckets, alors la chance de tomber dans un bucket quelconque en particulier [2] est $1/n$. De façon générale, nous pouvons définir une fonction de hachage d'indice de la façon suivante.

### Fonction d'indice

La fonction d'indice est une fonction de hachage, qui étant donnée une clef de type entier dans l'ensemble des clefs, renvoie l'indice d'un bucket de la table, dans lequel on peut insérer la paire clef-valeur.

**Définition 2.2.1** (Fonction de hachage (indice))**.** *Soit $\mathbb{K}$ l'ensemble des clefs. Soit $T = \{0, 1, \ldots, n-1\}$ l'ensemble des buckets d'une table de taille $n$. Une fonction de hachage* index *associe une clef de $\mathbb{K}$ à une valeur de $T$. Plus formellement, pour $k \in \mathbb{K}$ et $i \in T$, on a*

$$\text{index} : \mathbb{K} \to T$$
$$\text{index} : k \mapsto i$$

### Fonction de hachage de clefs

La fonction de hachage de clefs est une fonction de hachage qui transforme une clef d'un type $T$ en un entier naturel.

**Définition 2.2.2** (Hachage de clefs)**.** *Soit $\mathbb{K}$ l'ensemble des clefs. On définit* hash *la fonction qui associe une clef $k$ de $\mathbb{K}$ à un entier positif dans $n\mathbb{N} = \{0, 1, \ldots, n-1\}$. On a alors, pour $k \in \mathbb{K}$ et $m \in n\mathbb{N}$ :*

$$\text{hash} : \mathbb{K} \to n\mathbb{N}$$
$$\text{hash} : k \mapsto m$$

### Fonction d'indice ou de hachage ?

Souvent, les clefs ne sont pas des entiers. Les fonctions de hachage des tables de hachage ne prenant en entrée que des entiers, il est donc nécessaire de transformer les clefs en entier. Pour cela, on utilise en général une autre fonction de hachage, qui transforme la clef en entier. Nous ne rentrerons pas dans les détails, mais dès maintenant, lorsque nous parlerons de fonction d'indice, nous parlerons de la fonction de hachage qui renvoie l'indice dans la table, tandis que lorsque nous parlerons de fonction de hachage de clef, nous parlerons de la fonction qui renvoit un entier à partir d'une clef. La fonction d'indice et la fonction de hachage de clef sont toutes deux des fonctions de hachage : ce sont des fonctions dans un domaine fini !

2. Notez bien la structure de la phrase : si on avait dit "la chance de tomber dans un bucket quelconque", alors la probabilité aurait été de 1... On tombe en effet toujours dans un bucket !

CC BY NC ND

**Hachage de différents types**

Il existe plusieurs façons d'implémenter le hachage de clefs. Une méthode assez simple est de prendre les $k$ premiers bits dans la représentation binaire de l'objet, et de les considérer comme un entier. Cette méthode a l'avantage d'être extrèmement rapide, toutefois elle pose un certain nombre d'inconvénients. Par exemple, si l'on considère des chaînes de caractères, alors cela revient à prendre les tous premiers caractères. Voyez-vous le problème ?
Pour des mots similaires, commençant par les mêmes caractères, alors la fonction de hachage renverrait les mêmes bits ! On aurait donc beaucoup de collisions, ce qui n'est pas idéal...

### 2.2.2 Hachage (d'indice) par multiplication

La méthode par multiplication est assez peu utilisée, toutefois elle a le mérite d'être plutôt efficace. Nous ne rentrerons pas trop dans les détails (et vous êtes encouragés à approfondir le concept), mais voici l'idée. On prend un réel $0 < a$, et on pose $n$ la taille de la table. Si $k$ est une clef, alors son indice est déterminé par la formule suivante :

$$\text{index}(k) = \lfloor n(a\text{hash}(k) \mod 1) \rfloor$$

Il a été suggéré, notamment par Donald Knuth, d'utiliser le nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$ comme valeur pour $a$.

### 2.2.3 Hachage (d'indice) par division

Cette méthode est la plus répandue et utilisée. L'idée est ici de calculer le reste de la division euclidienne du hachage de la clef par la taille de la table, et d'insérer à l'indice obtenu. Ainsi, si $k$ est la clef, et que la table est de taille $n$, alors on aura

$$\text{index}(k) = \text{hash}(k) \mod n$$

On peut remarquer plusieurs choses sur cette méthode. Premièrement, elle est plutôt simple à implémenter dans la plupart des langages de programmation. Deuxièmement, on peut remarquer que certaines valeurs de $n$ (la taille de la table) sont meilleures que d'autres.
En effet, si $n$ est pair, alors index($k$) sera pair si hash($k$) est pair, impair sinon. Cela peut apporter certains biais, et favoriser les collisions dans certains cas. De plus, si $n$ est une puissance de la base de l'ordinateur (typiquement 2 – le binaire), cela est encore pire. En effet, le reste de la division par une puissance de 2 en binaire n'est rien d'autre que les $n$ bits les moins significatifs (les $n$ bits à droite dans la représentation binaire) [3].

$$\overbrace{1010 \quad 0101 \quad 1101 \quad 0111 \quad 0110 \quad 0111 \qquad \underbrace{1100 \quad 0111}_{\text{8 bits les moins significatifs}}}^{\text{32 bits}}$$

Pour d'éviter ce problème, il est souvent recommandé d'utiliser un $n$ premier. En pratique toutefois, vous remarquerez que la taille des tables de hachage est très souvent une puissance de 2. Pourquoi ? Car même si cela a des inconvénients, il y a aussi un énorme avantage. En effet, comme dit précédemment, le reste de la division d'un entier par une puissance de 2 en base 2 revient à prendre les bits les moins significatifs. Cela est extrêmement léger pour l'ordinateur, et est donc un trade-off[3] pratique acceptable vis-à-vis de l'augmentation du nombre de collisions théorique.

3. dans le sens "bon" compromis

### 2.2.4 Pour aller plus loin

Nous recommandons fortement le livre **Algorithms** [2] de Sanjoy Dasgupta, Christos Papadimitriou et Umesh Vazirani. Celui-ci peut être trouvé en ligne, notamment ici :
http ://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf.
C'est une lecture très intéressante, et beaucoup de choses sont abordées. D'autres livres tels que **Introduction to Algorithms** [1] de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein ainsi que **The Art of Computer Programming** [3] (ainsi que le volume 3) de Donald Knuth sont plus difficile à lire, mais sont néanmoins de très bonnes références. Notez qu'il existe des traductions en français sur internet.

?

**Hachage cryptographique**

Dans les cas d'applications où la sécurité est extrèmement importante, on pourra être amené à privilégier des fonctions cryptographiques pour obtenir le hachage de mots-clefs (avant de le passer à la fonction d'indice). Les fonctions de hachage cryptographiques ont quelques propriétés très intéressantes, notamment l'*effet avalanche*. Celui-ci peut être expliqué de la manière suivante : si l'on a $k_1$ et $k_2$ deux clefs assez proches (elles diffèrent d'un nombre faible de bits), alors leurs hachages respectifs par une fonction de hachage cryptographique doivent être éloignés. En pratique, cela se traduit par un caractère pseudo-aléatoire de cette fonction de hachage.
L'un des désavantages de ces fonctions est qu'elles sont en général plus lentes, car demandent des suites d'opérations plus longues et complexes.

CHAPITRE 3

# CHAÎNAGE SÉPARÉ

En chaînage séparé, l'idée est de résoudre les collisions entre valeurs en ajoutant une structure de données annexe pointée par le bucket. Les clefs sont donc stockées dans cet espace mémoire supplémentaire. De ce fait, le facteur de charge peut-être supérieur à 1 ($\alpha > 1$).

## 3.1 Utilisation de listes chaînées

Une des méthodes les plus simples de chaînage est celle des listes chaînées. S'il y a une collision entre des valeurs, le bucket d'insertion pointera vers le début de la liste chaînée, et les valeurs seront insérées dans la liste chaînée. Cette méthode est assez simple et facile à implémenter. Toutefois, elle a un désavantage : quand il y a beaucoup de collisions sur la même valeur, la liste grandit beaucoup, ce qui provoque des ralentissements. En effet, afin de chercher un élément dans la liste, on prend le risque de devoir la parcourir entièrement, ce qui est linéaire en sa taille. Cela peut poser un problème dans le cas où les données peuvent venir d'utilisateurs (site internet, par exemple). On pourrait en effet pré-calculer des clefs qui entraineront des collisions, et ainsi ralentir fortement le système. Cela s'appelle le HashDoS, pour Hashing Denial of Service (et c'est interdit par la loi !). Afin de limiter ça, des protections sont souvent mises en place, notamment l'utilisation d'une séquence de bits aléatoires (appelée "salt"), nouvelle pour chaque table de hachage, ajoutée lors du calcul d'indice.

### 3.1.1 Insertion

Dans les exemples qui suivent, on suppose que la stratégie d'insertion est celle d'une file : premier arrivé, premier servi (FIFO). Ainsi, une nouvelle valeur est insérée à la fin de la liste chaînée. Il existe d'autres stratégies, notamment dernier arrivé, premier servi, qui garantit une insertion constante en pire cas.

Soit $T$ la table de hachage suivante. Soit $\text{index}(k) = k \mod 11$ la fonction associée à cette table de hachage. Insérons, dans l'ordre donné, les valeurs $4, 155, 114, 9, 122, 2, 11, 244$ dans la table.

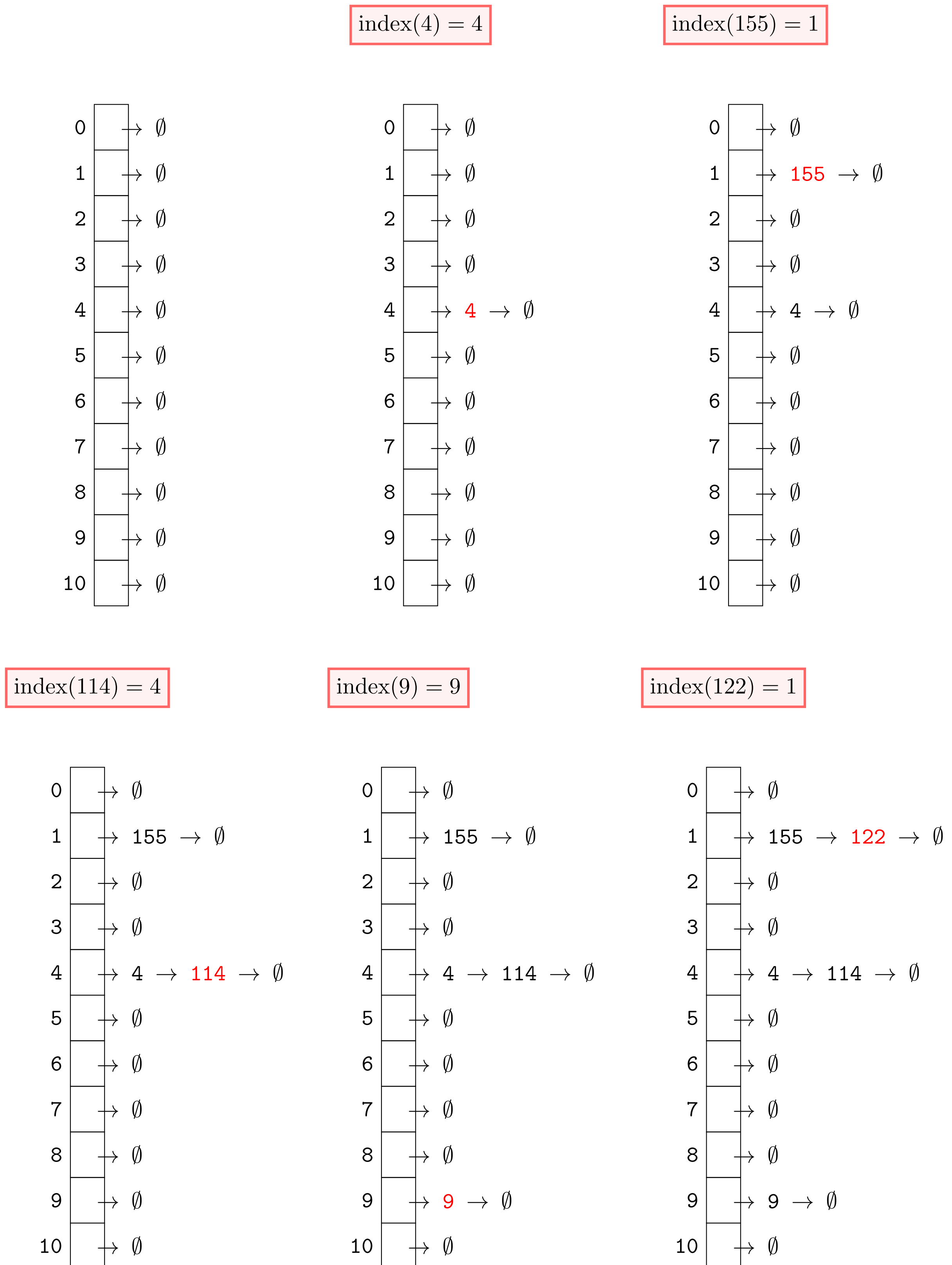



FIGURE 3.1 – Insertion dans une table de hachage

## 3.1.2 Recherche

Lorsqu'on recherche une clef dans la table, on procède de la même manière que pour l'insertion. La différence est que l'on s'arrête lorsqu'on a trouvé la clef, ou que le bucket pointe sur `NULL`, donc que l'on est en bout de chaîne, et qu'il n'y a pas la valeur.

### 3.1.3 Suppression

La suppression dans une table de hachage par liste chaînée est en général assez simple : on localise la clef, et on la supprime comme dans une liste chaînée classique. Voici un exemple :

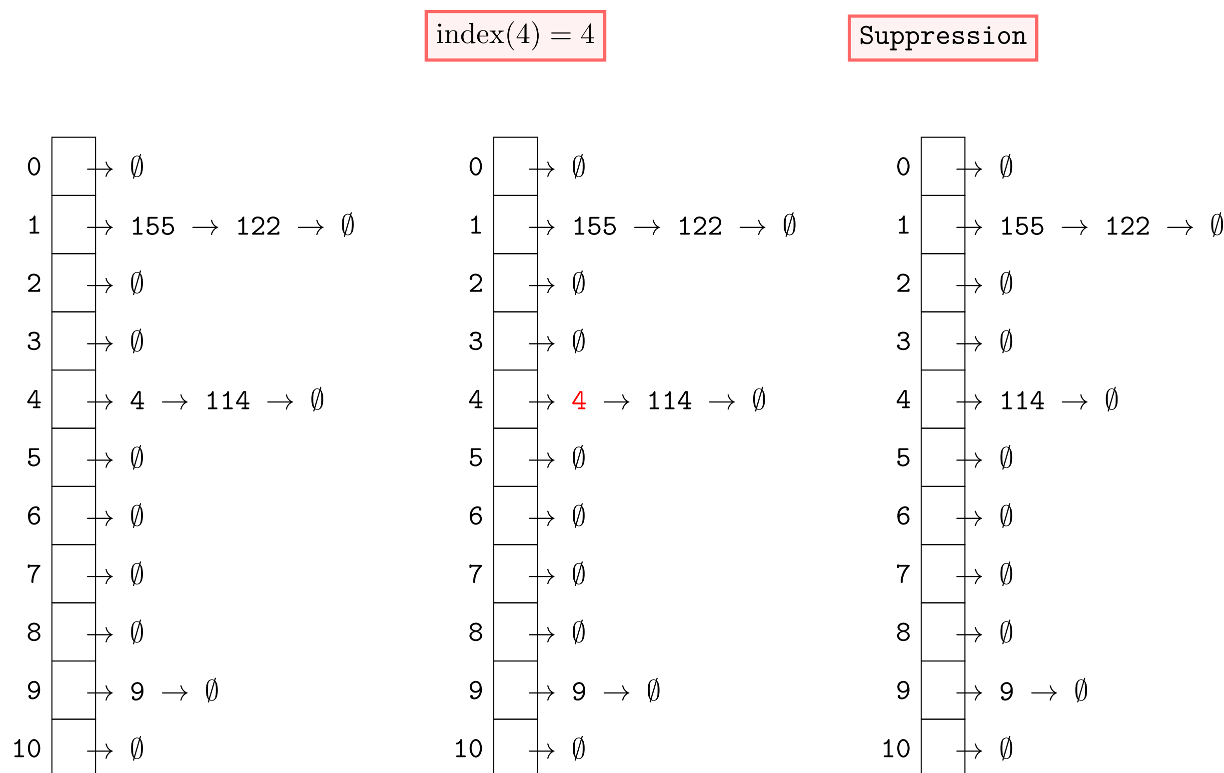



FIGURE 3.2 – Suppression de la clef 4

### 3.1.4 Redimensionnement

Dans certains cas (cela dépend des implémentations), on sera amené à redimensionner la table de hachage. En effet, quand le facteur de charge $\alpha$ est plus grand que 1, les performances se dégradent très vite car on doit parcourir des listes chaînées. Afin d'éviter cela, on redimensionne la table de hachage, selon une règle précise (là aussi définie par l'implémentation). L'idée est donc de redimensionner puis d'insérer toutes les paires clef-valeur qui étaient préalablement dans la table. Pour cela, on re-hache toutes les clefs dans la nouvelle table.

### 3.1.5 Complexité algorithmique

Nous allons ici analyser la complexité en pire cas des tables de hachage par liste chaînée. Soit $T$ une table de hachage quelconque par liste chaînée avec $n$ buckets et $m$ valeurs. On a donc un facteur de charge $\alpha = m/n$.

**Théorème 3.1.1** (Complexité en pire cas)**.** *Soit $T$ une table de hachage par chainage utilisant des listes chaînées, tel que la taille de $T$ est $n$. On suppose qu'il y a $M$ éléments dans $T$. On a donc un facteur de charge $\alpha = M/n$. Pour l'insertion, la suppression et la recherche, la complexité en pire cas est linéaire en le nombre d'éléments $M$.*

*Démonstration.* Supposons qu'il y ait $M$ valeurs qui entrent en collision entre elles. Alors, l'un des buckets contient une chaîne de $M$ éléments. Voici les trois cas que nous traiterons : l'insertion,

CC BY NC ND

puis le lookup et enfin la suppression.

1. Insertion : l'insertion est linéaire si l'on insère toujours à la fin (on doit dans ce cas parcourir toute la liste). On est donc linéaire en $M$. *Notez qu'il existe des stratégies où l'on insère à l'entrée de la liste...*
2. Recherche : supposons que l'élément que l'on cherche ne soit pas présent. Tant que l'on n'a pas trouvé l'élément, on continue à parcourir la liste, jusqu'à ce que l'on arrive à la fin. On parcourt les $M$ valeurs (sans rien trouver). On est donc en temps linéaire en $M$.
3. Suppression : pour la suppresion, on suppose en plus que l'élément à supprimer n'est pas présent dans la liste. On doit donc parcourir toute la liste chaînée pour se rendre compte que celui-ci n'y est pas, et nous sommes donc linéaire en $M$.

Dans les trois cas, nous avons vu que nous étions linéaire en $M$. Dans le pire cas, la complexité est donc linéaire en $M$ pour l'insertion, la suppression et la recherche. □

**Théorème 3.1.2** (Complexité moyenne (admis))**.** *Soit $T$ une table de hachage par chainage utilisant des listes chaînées, telle que la taille de $T$ est $n$. On suppose qu'il y a $M$ éléments dans $T$. On a donc un facteur de charge $\alpha = M/n$. Pour l'insertion, la suppression et la recherche, la complexité en moyenne est constante.*

*Démonstration.* La preuve peut être trouvée dans [1]. Bien qu'elle dépasse l'objectif de ce cours, il peut être intéressant de la regarder et d'essayer de la comprendre. L'idée derrière la démonstration est de regarder la longueur moyenne des chaînes en fonction de $\alpha$. □

## 3.2 Autres structures de données

Les listes chaînées ont pour désavantage un temps de recherche linéaire en leur taille, et cela devient assez embêtant quand elles sont grandes. D'autres solutions existent, et on peut remplacer les listes chaînées par d'autres structures de données, selon l'application souhaitée. Par exemple, en Java, à partir d'un certain nombre de collisions dans un même bucket, la liste chaînée du bucket est remplacée par un arbre binaire de recherche équilibré. Cela permet de garantir une recherche, une insertion et une suppression en temps logarithmique en le nombre d'éléments (et donc de collisions) dans le bucket. Le désavantage des arbres binaires est que l'on doit pouvoir comparer les éléments entre eux, car ceux-ci nécessitent une relation d'ordre.

**Java et les tables de hachage**

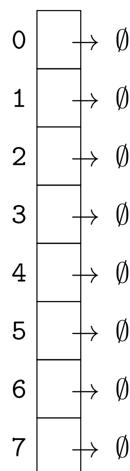
En Java, lorsque vous implémentez les méthodes `hashCode` et `equals` pour un objet afin de le rendre hachable, il est conseillé d'implémenter l'interface **Comparable**. En effet, Java pourra utiliser des arbres binaires équilibrés s'il y a un trop grand nombre de collisions : vous gagnerez en performances et vous protégerez du HashDoS.

## 3.3 Exercices

### Exercice 1 (Opérations dans une table de hachage par liste chainée)

**Type :** APPLI

Soit $T$ la table de hachage par liste chaînée suivante. On suppose que celle-ci implémente la stratégie premier arrivé, premier servi.




**Question 1.1** (Fonction d'indice)
**Difficulté :** 1
Justifiez que la fonction d'indice suivante est adaptée à $T$

$$\text{index}(k) = k \mod 8$$

Dans le reste de l'exercice, nous utiliserons cette fonction d'indice.

**Question 1.2** (Insertions de valeurs)
**Difficulté :** 1
Insérez dans l'ordre donné les valeurs suivantes dans la table de hachage $T$ :

$$[16, 22, 1, 0, 8, 265, 24, 33, 19]$$

Détaillez les étapes.

**Question 1.3** (Suppression de valeurs)
**Difficulté :** 1
Supprimez les valeurs suivantes de la table de hachage $T$ :

$$[19, 22, 8, 0, 34, 33]$$

Détaillez les étapes.

**Question 1.4**
**Difficulté :** 1
Quelle est la valeur du facteur de charge $\alpha$ ? Avant suppression et après suppression.

## Exercice 2 (Coder une table de hachage)

**Type :** IMPLE
Implémentez en C une table de hachage par liste chainée. Vous pourrez utiliser une implémentation de liste chaînée que vous avez préalablement faite. La fonction d'indice de la table doit être la suivante :

$$\text{index}(k) = (k) \mod n$$

avec $k$ une clef et $n$ la taille de la table. La taille de $n$ et la politique de redimensionnement (c'est-à-dire comment grandit $n$) ne sont pas imposés. Votre table de hachage doit permettre de :

- Insérer des valeurs (de type chaîne de caractères), avec une clef de type entier positif
- Supprimer des paires clef-valeur
- Vérifier la présence d'une clef dans la table. Si la clef demandée est présente, votre fonction doit retourner true, sinon false.

— Retourner l'indice d'une clef. Un peu comme la méthode de vérification, sauf que votre fonction doit retourner l'indice de la clef dans la table si elle est présente. Si elle n'est pas présente, vous pouvez choisir une autre valeur de retour (typiquement, NULL).
— Redimensionner automatiquement la taille de la table si le facteur de charge dépasse 1 (et donc rehash toutes les clefs).
— Retourner la paire clef-valeur si la clef est présente.

CHAPITRE 4

# ADRESSAGE OUVERT

En adressage ouvert (open addressing en anglais), l'idée est d'insérer les valeurs directement dans la table, sans autre structure de données. Pour cela, quand une collision arrive, la position dans laquelle on doit insérer ou chercher l'élément est trouvée grâce à un *probing*. Le probing, ou sondage, est la séquence de bucket qui sera parcourue. Ainsi, lors de la conception d'une table par adressage ouvert, la question primordiale que l'on doit se poser est celle de la méthode de probing à utiliser. Il existe en effet beaucoup de méthodes différentes, et nous allons en présenter quelques unes. Tout d'abord, sachez qu'en adressage ouvert, et contrairement au chaînage séparé, le facteur de charge $\alpha = M/n$ doit être plus petit (ou égal) à 1 ($\alpha \leq 1$). En effet, ici on ne dispose que des buckets pour ranger les clefs. Notez qu'en pratique, on aura souvent $\alpha < 1$, et cela est dû à une dégradation des performances quand le facteur de charge approche 1.

## 4.1 Linear probing

L'idée derrière le linear probing est d'utiliser une fonction linéaire pour déterminer la séquence de bucket à sonder.

Φ

**Fonction d'indice pour le linear probing**

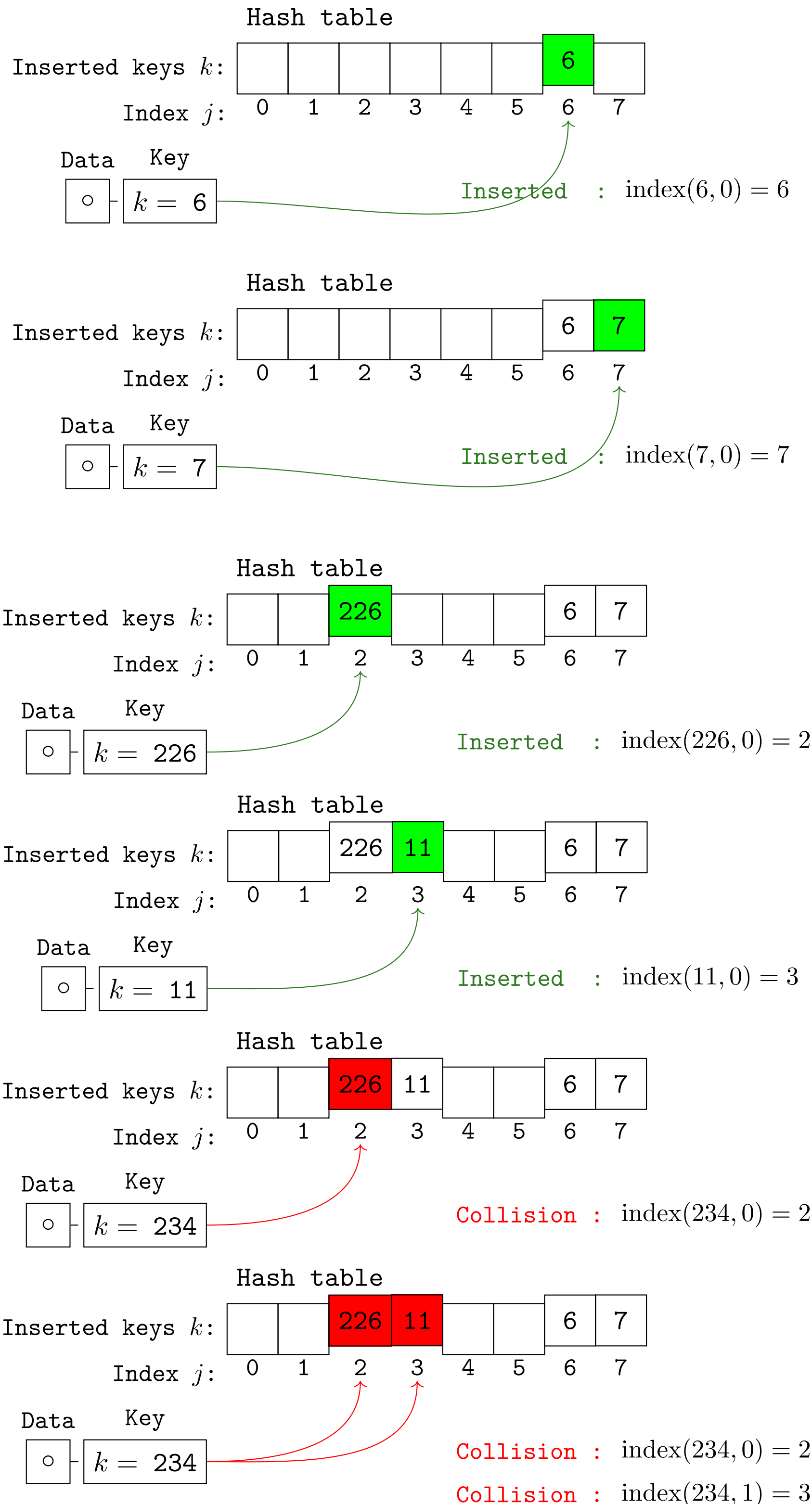
**Définition 4.1.1** (Fonction d'indice pour le linear probing)**.** *Soit $T$ une table de hachage de taille $n$ utilisant le linear probing. Soit $k$ une clef. Soient $a, b$ des entiers, tel que $b \neq 0$. On pose $i$ le nombre d'itérations à effectuer avant de trouver un bucket valide, avec $i = 0$ lors de l'initialisation.*

$$\text{index}(k, i) = (a\text{hash}(k) + bi) \mod n$$

Si pour $i = 0$ le bucket est indisponible, alors on cherche pour $i = 1$, puis $i = 2$, et ainsi de suite en incrémentant $i$ de 1 à tant que le bucket est indisponible.

### 4.1.1 Insertion

Ci-dessous, vous pouvez voir un exemple d'insertion dans une table de hachage par linear probing, avec la fonction suivante : $\text{index}(k, i) = \text{hash}(k) + i \mod n$. Ici, $n$ vaut 8 et $\text{hash}(k) = k$ (on insère des entiers).




CC BY NC ND

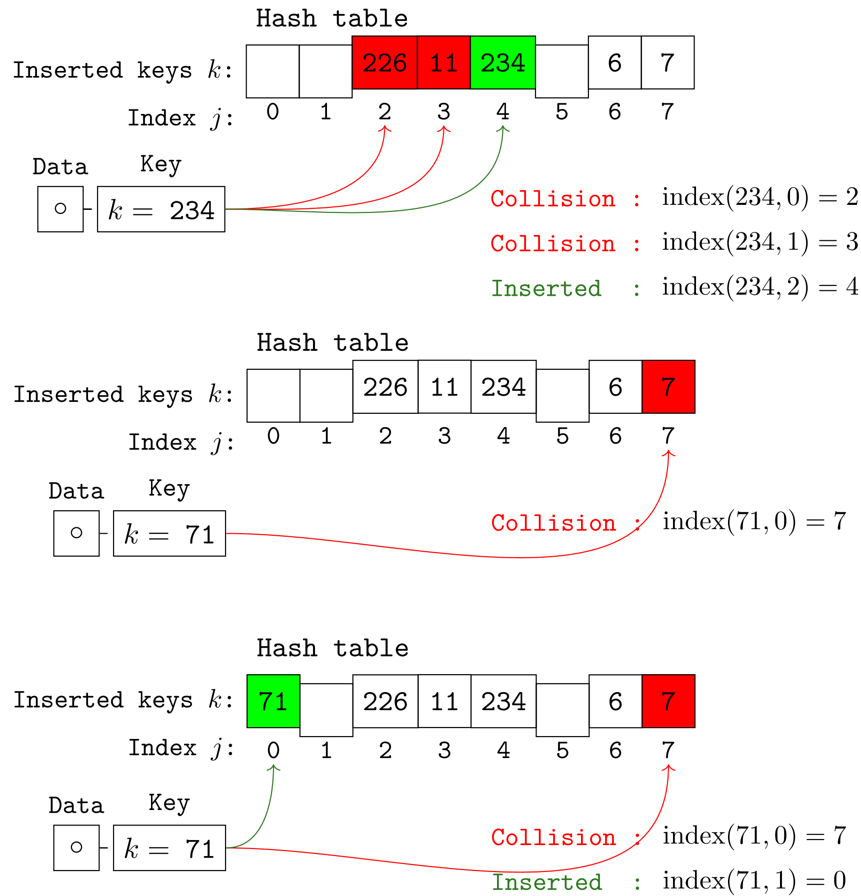



FIGURE 4.1 – Insertion dans une table de hachage de différentes valeurs

Vous avez peut être remarqué lors de l'insertion des clefs que le bucket suivant est toujours celui à droite sauf quand on arrive au bord. Cela entraîne la formation de longues séquences qui remplissent la table de manière contigüe. C'est un des problèmes majeurs du linear probing : le *primary clustering*. En effet, plus la séquence est grande, disons $k$ clefs, plus la probabilité qu'une prochaine clef demande à y être insérée est grande : $k/n$... Et c'est un cercle vicieux : si la prochaine clef y est insérée, alors la longueur de la séquence augmente, et donc la probabilité que la prochaine clef y soit insérée...

### 4.1.2 Recherche

Lorsqu'on recherche une clef dans la table, on procède de la même manière que pour l'insertion. La différence est qu'on s'arrête lorsque l'on a trouvé la clef, ou que l'on tombe sur un bucket vide et donc que la clef est absente.

### 4.1.3 Et la suppression ?

La suppression dans une table de hachage par adressage ouvert, et donc par linear probing, n'est pas aussi simple que dans le cas du chaînage. En effet, si dans le cas du chaînage on peut se contenter de supprimer la valeur et de raccorder les maillons entre eux (ce qui se fait en temps constant), on ne peut pas dans le cas de l'adressage ouvert

CC BY NC ND

utiliser cette technique.

La raison est que si nous procédions de la sorte, étant donné que l'algorithme s'arrête quand il trouve une case vide, alors il considérerait qu'il n'y a plus rien à chercher après ! Ce n'est pas souhaitable, et nous avons donc besoin de signaler à l'algorithme que la clef a été supprimée, mais que d'autres clefs se situent encore après. Pour cela, on va placer une *tombstone* (pierre tombale), souvent représentée par $\perp$ dans la littérature. Celle-ci signifie à l'algorithme qu'il peut continuer. Dans le cas où l'on voudrait insérer une nouvelle clef, celle-ci pourrait bien entendu remplacer la tombstone. Vous vous demandez pourquoi on ne déplace tout simplement pas toutes les clefs placées après la clef supprimée de sorte qu'elles prennent chacunes la place de la précédente. C'est une stratégie assez lente, qui est donc peu utilisée en pratique : on préfère l'utilisation de tombstones.

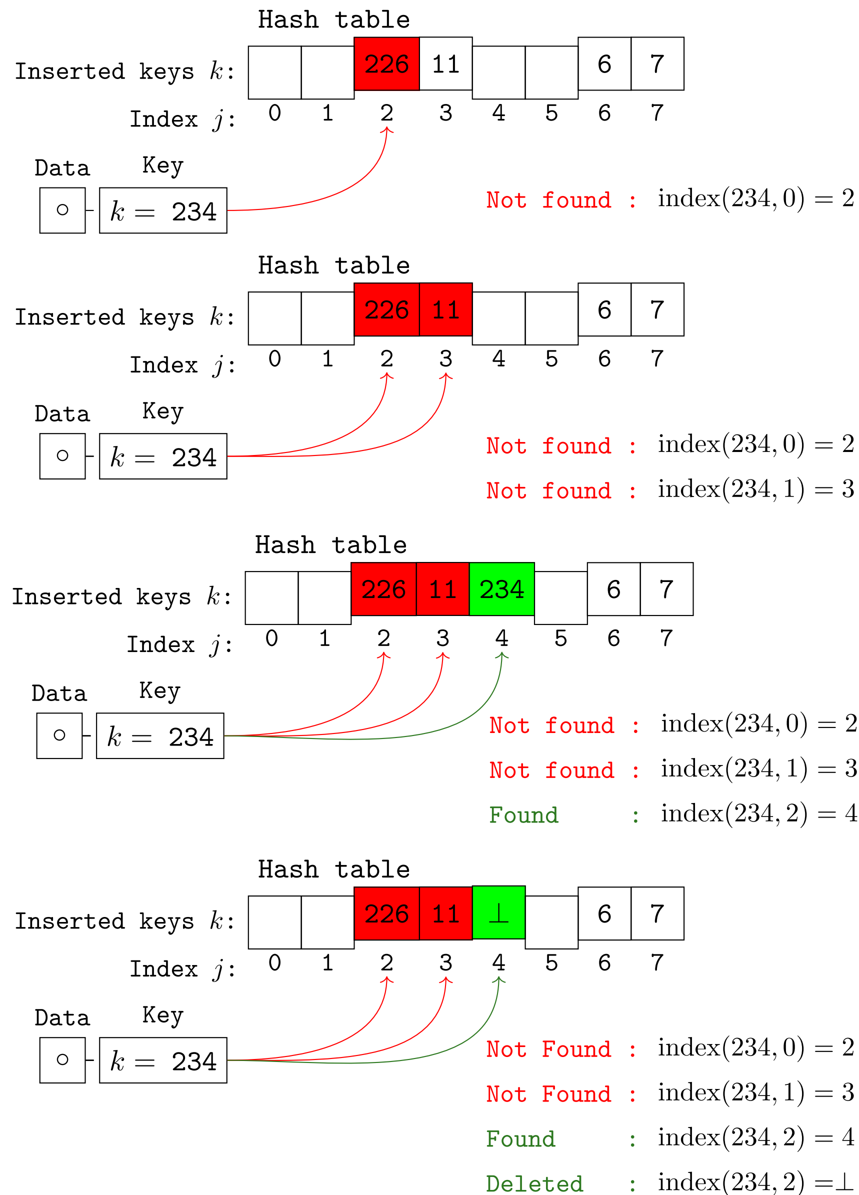



FIGURE 4.2 – Suppression dans une table de hachage de la clef $k = 234$

**Suppression de clefs**

Attention lorsque vous supprimez des clefs à bien mettre les tombstones $\perp$ ! Sinon, votre algorithme ne terminera pas...

**Implémentation des tombstones**

Il existe plusieurs manières d'implémenter les tombstones. L'une des plus simples est d'insérer une valeur prédéfinie qui ne peut être prise par aucune clef. Cette méthode a l'inconvénient d'empêcher certaines valeurs, et est donc à proscrire. Il peut donc être utile d'implémenter un bucket comme une structure contenant la paire clef-valeur, mais aussi un bit signifiant si une valeur était préalablement présente. Il peut aussi être intéressant d'utiliser un octet afin de mettre plus d'informations : est-ce que le prochain bucket est vide, etc.

### 4.1.4 Redimensionnement

Comme le facteur de charge $\alpha$ ne peut pas dépasser 1 en adressage ouvert, on est obligé d'agrandir la taille de la table. En pratique, on n'attend pas d'arriver à 1 pour redimensionner, car selon les stratégies de probing utilisées, les performances se dégradent bien avant... On fixe souvent la valeur maximale dans $[0.6, 0.75]$ pour le facteur de charge en linear probing.
Quand ce facteur de charge maximal est atteint, on redimensionne la table avec une règle qui dépend de l'implémentation, et on re-hache toutes les clefs qui étaient préalablement dans la table.

## 4.2 Quadratic probing

Le fonctionnement du quadratic probing diffère de celui du linear probing en sa fonction d'indice. Là où la fonction d'indice du linear probing est linéaire, celle du quadratic probing est un polynôme de degré 2.

**Fonction d'indice en Quadratic Probing**

**Définition 4.2.1** (Fonction d'indice en Quadratic probing)**.** *Soit $T$ une table de hachage de taille $n$ par adressage ouvert. Soient $k$ une clef et $i$ le nombre d'itération. Soient $a, b, c \in \mathbb{R}^+$ avec $b \neq 0$. En pratique, $a, b, c$ sont des entiers. Ainsi, pour $k$ une clef, $n$ la taille de la table et $i$ le nombre d'itérations, on a :*

$$\text{index}(k, i) = \left(a\text{hash}(k) + bi^2 + ci\right) \mod n$$

*Notez que le résultat devant être un entier et $a, b, c$ pouvant être des réels positifs, on peut être amené à prendre une partie entière avant d'appliquer le modulo.*

En y regardant de plus près, un des avantages du quadratic probing est qu'il évite le primary clustering. En effet, on "saute" plus loin à chaque probing, et on évite donc de former des séquences contigües en mémoire. Toutefois, on observe que deux clefs qui entrent en collision le feront à chaque itération. On appelle ce phénomène *secondary clustering*.

**Quelles valeurs et quelle taille pour la table ?**

Selon la taille de la table, les valeurs $a, b$ et $c$ peuvent faire varier les performances. On se place dans un cadre où $a, b, c$ sont des entiers naturels, avec $b$ non nul.

**Théorème 4.2.1** (Probing total). *Soit $T$ une table de hachage de taille $p^N$, avec $p$ premier et $N$ un entier naturel. Soit* index *la fonction d'indice associée à $T$, et tel que pour $k$ une clef, $a, b, c$ des entiers ($b$ non nul) et $i$ le nombre d'itérations, la valeur de* index *est :*

$$\text{index}(k, i) = \big(a\text{hash}(k) + bi^2 + ci\big) \mod p^N$$

*Alors, pour $i \in \{0, 1, ..., p^N - 1\}$, si $p$ divise $b$ et $p$ ne divise pas $c$, les valeurs prises formeront une permutation de $\{0, 1, ..., p^N - 1\}$. Ainsi, tous les buckets pourront être visités.*

*Démonstration.* La preuve est un peu avancée pour ce cours. Une des méthodes consiste à utiliser le lemme de Hansel. □

## 4.2.1 Insertion, suppression et recherche

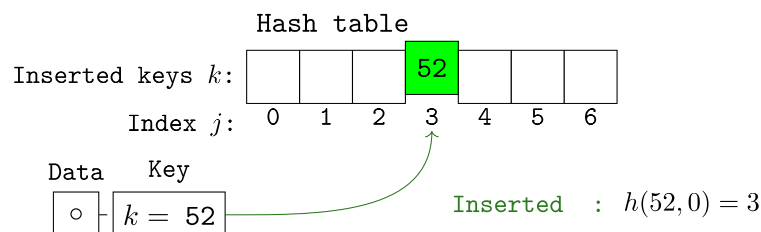
Le fonctionnement de l'insertion, de la suppression et de la recherche d'une clef est le même que pour le linear probing, mais utilise la fonction de quadratic probing à la place.

## 4.2.2 Exemple

On considère une table de hachage de taille 7 utilisant le quadratic probing, avec la fonction d'indice $\text{index}(k, i) = \big(\text{hash}(k) + 14i^2 + 3i\big) \mod 7$. On souhaite insérer (dans l'ordre) les clefs suivantes :

$$[52, 210, 213, 40, 227]$$

Voici à quoi cela ressemble :




CC BY NC ND

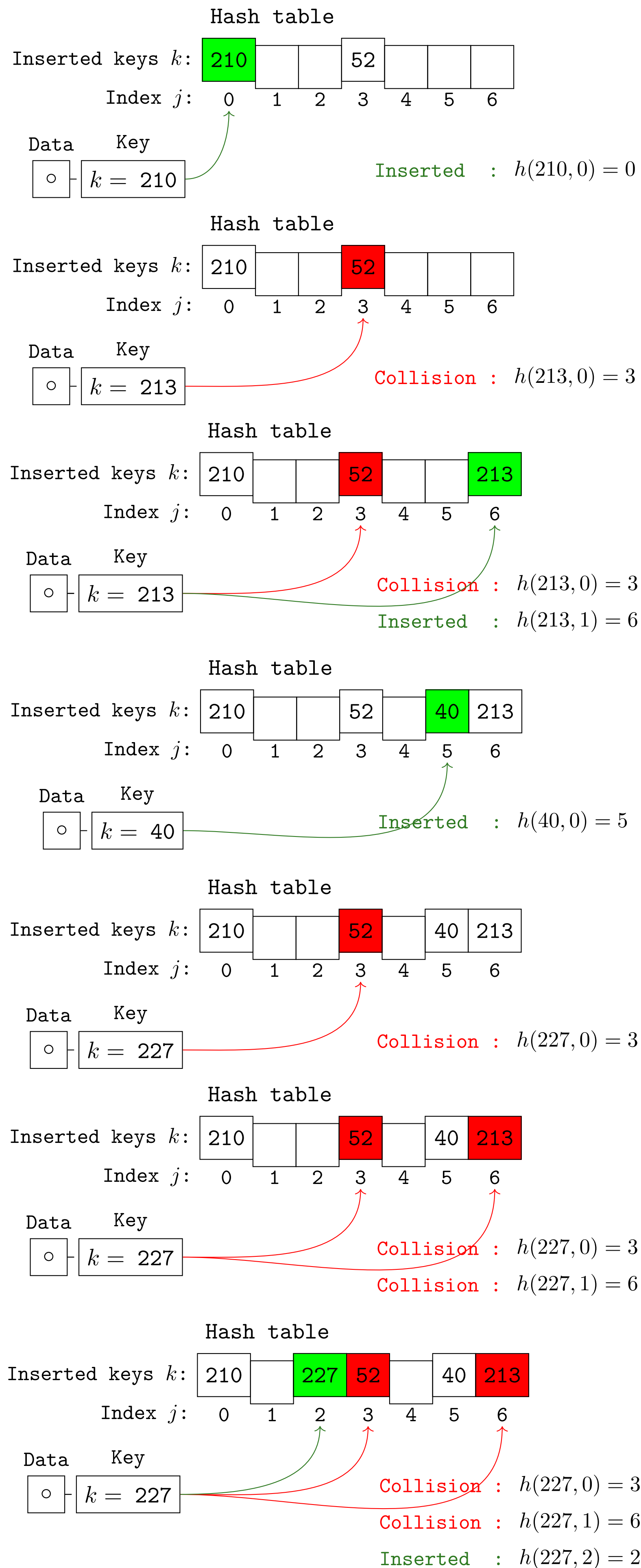



FIGURE 4.3 – Insertion par quadratic probing

## 4.3 Pour aller plus loin

Comme nous l'avons vu, le linear probing possède le problème du primary clustering. Ces méthodes ont chacunes des avantages et des inconvénients. D'autres méthodes existent, comme le double hashing qui est un peu plus lent (car il faut calculer deux fonctions de hachage), et est affecté par le *tertiary clustering* : des clefs entrant en collision pour les deux fonctions de hachage entreront toujours en collision. Il existe encore d'autres méthodes que nous n'expliquerons pas ici, comme le Cuckoo Hashing, le Robin Hood Hashing...

## 4.4 Exercices

**Exercice 3** (Opérations dans une table de hachage par adressage ouvert)
**Type :** APPLI
Soit $T$ la table de hachage par adressage ouvert suivante.

Hash table

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Inserted keys $k$: | | | | | | | | | | | | | | | | |
| Index $j$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

**Question 3.1** (Fonction d'indice)
**Difficulté :** 2
Donnez un exemple de fonction d'indice qui fonctionne pour $T$. Celle-ci peut utiliser du linear probing, du quadratic probing, ou une autre méthode que vous prendrez le soin de détailler. Elle doit impérativement, et pour n'importe quelle valeur de $k$, pouvoir visiter tous les buckets.

Dans le reste de l'exercice, nous supposerons que la fonction d'indice utilisée est de type linear probing. Pour $k$ une clef et $i$ le nombre d'itérations, la fonction d'indice index est la suivante :

$$\text{index}(k, i) = (7k + i) \mod 16$$

**Question 3.2** (Insertion de valeurs)
**Difficulté :** 1
Insérez dans l'ordre donné les valeurs suivantes dans la table de hachage $T$ :

$$[242, 47, 27, 118, 52, 137, 252, 155, 73, 233, 151]$$

Détaillez les étapes.

**Question 3.3** (Suppression de valeurs)
**Difficulté :** 1
Supprimez les valeurs suivantes de la table de hachage $T$ :

$$[52, 47, 73, 137, 151]$$

Détaillez les étapes.

**Question 3.4**
**Difficulté :** 1
Quelle est la valeur du facteur de charge $\alpha$ ? Avant suppression et après suppression.

CC BY NC ND

## Exercice 4 (Coder une table de hachage)

**Type :** IMPLE

Implémentez en C une table de hachage par adressage ouvert utilisant le linear probing avec la fonction d'indice suivante :

$$\text{index}(k, i) = (k + i) \mod n$$

avec $k$ la clef, $i$ le nombre d'itérations, et $n$ la taille de la table et une puissance de deux. Votre table de hachage doit permettre de :

- Insérer des valeurs de type chaîne de caractères, avec une clef de type entier positif
- Supprimer des paires clef-valeur
- Vérifier la présence d'une clef dans la table. Si la clef demandée est présente, votre fonction doit retourner true, sinon false.
- Retourner l'indice d'une clef. Un peu comme la méthode de vérification, sauf que votre fonction doit retourner l'indice de la clef dans la table si elle est présente. Si elle n'est pas présente, vous pouvez choisir une autre valeur de retour (typiquement, NULL).
- Redimensionner automatiquement la table de hachage en prenant comme nouvelle taille $m = 2n$ (doublement de taille) quand le facteur de charge dépasse 0.65.
- Retourner la paire clef-valeur si la clef est présente

## Exercice 5 (Permutations et tables de hachage)

**Type :** MATHS

Différentes fonctions d'indices vous sont données, et chacune d'entre elle est associée à une table d'une certaine taille. Pour chaque fonction d'indice, dites si elle permet de visiter tous les buckets pour toute valeur de clef, et démontrez-le. On utilise l'adressage ouvert.

*Indice : vérifiez que les valeurs prises par la fonction sont une permutation de* $\{0, ..., n-1\}$*, avec* $n$ *la taille de la table.*

**Question 5.1** (Linear probing classique)
**Difficulté :** 2

Soit $T$ une table de hachage par adressage ouvert de taille $n = 2^m$, $m$ entier positif, et

$$\text{index}_1(k, i) = (k + i) \mod n$$

la fonction associée à $T$.

**Question 5.2** (Triangular Probing)
**Difficulté :** 3

Soit $T$ une table de hachage par adressage ouvert de taille $n = 2^m$, $m$ entier positif, et

$$\text{index}_2(k, i) = \left(k + \frac{i(i+1)}{2}\right) \mod n$$

la fonction associée à $T$.

**Question 5.3** (Puissance de 3)
**Difficulté :** 1

Soit $T$ une table de hachage par adressage ouvert de taille $n = 3^m$, $m$ entier positif, et

$$\text{index}_3(k, i) = (3k + 3i) \mod n$$

la fonction associée à $T$.

CC BY NC ND

**Question 5.4** (Puissances de 11 et quadratic probing)
**Difficulté :** 3
Soit $T$ une table de hachage par adressage ouvert de taille $n = 11^m$, $m$ entier positif, et

$$\text{index}_4(k, i) = \left(k + 22i^2\right) \mod n$$

la fonction associée à $T$.

**Question 5.5** (Puissances de 11 et quadratic probing bis)
**Difficulté :** 3
Soit $T$ une table de hachage par adressage ouvert de taille $n = 11^m$, $m$ entier positif, et

$$\text{index}_4(k, i) = \left(k + 14i^2\right) \mod n$$

la fonction associée à $T$.

# BIBLIOGRAPHIE


[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. isbn : 0262032937. url : http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937.

[2] Sanjoy Dasgupta et Christos H Papadimitriou. *Algorithms*. 2006.

[3] Donald E. Knuth. *The Art of Computer Programming, Vol. 3 : Sorting and Searching*. Third. Reading, Mass. : Addison-Wesley, 1997. isbn : 0201896834 9780201896831.

*C'est la dernière page.*