Tables de hachage

I. Introduction

Une structure de donnée impérative très utile pour implémenter un dictionnaire est la **table de hachage**.

L'idée est d'utiliser un tableau de taille m et d'avoir une fonction (de hashage) `h : 'a -> int` qui, à chaque élément de type `'a`, associe un entier indiquant la case où stocker la donnée. Ainsi, une donnée `d` sera stockée dans la case numéro `h d` (ou, plus précisément, `(h d) mod m`) du tableau.

Cependant, en pratique, on ne peut pas utiliser directement cette idée, à cause d'éventuelles *collisions*. On dit que deux données provoquent une collision si elles ont la même clé. On est obligé de gérer la possibilité de collisions dans l'implémentation d'une table de hashage.

> **Remarque** La possibilité d'une collision et les probabilités associées sont proches du paradoxe des anniversaires qui dit, que pour une assemblé d'au moins 23 personnes, il y a au moins 50% de chances que deux personnes soient nées le même jour de l'année. D'ailleurs, savez-vous retrouver ce résultat ?
> En termes de table de hashage, pour un tableau avec 365 cases, il suffit de vouloir stocker 23 éléments pour avoir au moins une chance sur deux d'obtenir une collision. Cela montre à quel point la gestion des collisions est importante.

Dans la suite de ce TD, nous allons étudier une méthode de gestion des collisions.

Pour illustrer la méthode utilisée, nous allons supposer que nous utilisons une table de dix cases (allant de 0 à 9) et que nous y insérons les chaînes de caractères suivantes, dans l'ordre et les cases indiqués :

	Élément	Clé de hashage
1.	`Il`	7
2.	`en`	0
3.	`faut`	5
4.	`peu`	5
5.	`pour`	7
6.	`être`	6
7.	`heureux`	7

Les clés successives à insérer forment la **séquence de hachage** $(7, 0, 5, 5, 7, 6, 7)$.

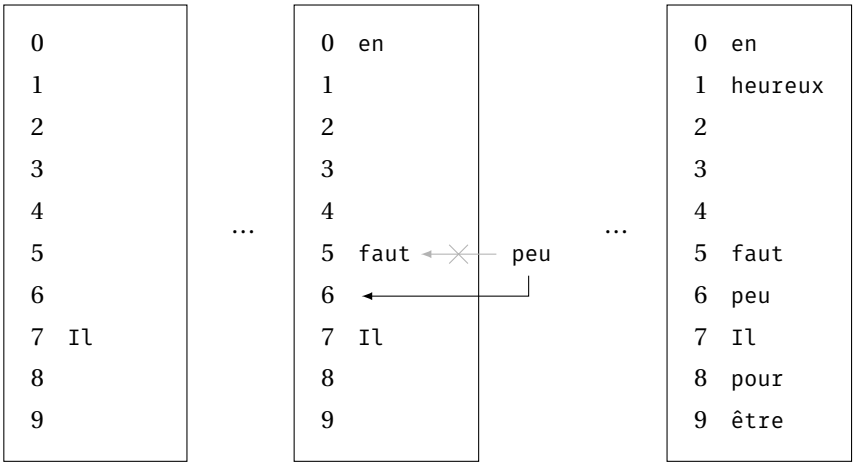
Les insertions successives dans le tableau se déroulent ainsi :

☞ Si la case correspondant à la clé de hachage est vide, on insère la valeur dans la case.

☞ Sinon, il y a **collision** et on doit chercher une autre case disponible (on suppose qu'il y en a une). Pour cela, nous allons procéder par **sondage linéaire** (linear probing) : on examine les cases suivantes dans l'ordre en sélectionnant la première case vide. Si l'on arrive au bout du tableau, on revient au début.

Ainsi, lorsque l'on veut insérer « `peu` » de clé 5, comme la case correspondante est déjà occupée, on mets le mot dans la case disponible suivante, ici la 6.

La séquence d'insertion des mots précédents peut donc se représenter ainsi :



Cette méthode s'appelle **sondage linéaire** (*linear probing* en anglais), puisqu'à partir de la case correspondant à la clé, on sonde les cases dans l'ordre (en cyclant éventuellement) jusqu'à trouver une case vide.

On suppose qu'il y a toujours une case vide dans la table lorsque l'on fait un insertion, une suppression ou une recherche.

II. Implémentation

1. Mise en place

Nous allons représenter une table de hachage par un table d'élements optionels, donc de type `'a option array`. Ainsi, la représentation de la table précédente est :

```
[| Some "en"; Some "heureux"; None; None; None; Some "faut";
   Some "peu"; Some "Il"; Some "pour"; Some "être" |]
```

On rappelle que le type `'a option` est défini par :

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

La fonction de hachage pour des données de type `'a`, de type `'a -> int`, est passée en argument à chaque appel de fonction la nécessitant. Les entiers renvoyés pouvant être quelconques, on utilisera le reste de la division par la taille de la table pour obtenir un indice convenable.

2. Recherche et insertion

Question 1 Écrire la fonction

```
val est_présent: 'a option array -> ('a -> int) -> 'a -> bool
```

telle que `est_présent t h e` retourne `true` ou `false` suivant que l'élément `e` est présent dans la table `t` en utilisant la fonction de hachage `h`.

Question 2 Écrire la fonction

```
val insérer: 'a option array -> ('a -> int) -> 'a -> unit
```

3. Suppression

Question 3 Reprenons l'exemple de table précédent, et supposons que l'on retire la valeur « `pour` » donnant la table suivante :

```
[| Some "en"; Some "heureux"; None; None; None; Some "faut";
   Some "peu"; Some "Il"; None; Some "être" |]
```

La recherche de la valeur « `être` » va donner une réponse négative, pourquoi? Comment y remédier? Et que dire de la recherche de « `heureux` »?

Question 4 Écrire la fonction

```
val supprimer: 'a option array -> ('a -> int) -> 'a -> unit
```

III. Analyse mathématique

Dans la suite, nous noterons n le nombre d'éléments présents dans la table de hachage et m la taille de la table de hachage. On suppose que l'on a toujours $n < m$, autrement dit que la table comporte toujours au moins une case vide.

Notons tout d'abord que la complexité de ces opérations peut être, dans le pire

des cas, en $O(n)$.

Question 5 Donner un exemple d'insertion nécessitant $n+1$ tests.

Nous allons montrer qu'en moyenne cependant, on peut avoir une complexité constante. Par *en moyenne*, nous allons supposer que toutes les m^n séquences de hachage sont équiprobables et, lors d'une recherche d'un élément présent, que tous les éléments recherchés sont équiprobables.

1. Recherche infructueuse

Question 6 Avec les notations précédentes, on a donc supposé que l'on a inséré n valeurs dans la table de taille m (initialement vide).

1. Combien y-a-t'il de cellules vides dans la table ?
2. En déduire que le nombre de séquences de hachage laissant vide la dernière case est
$$n^k\left(1-\frac{k}{n}\right)$$

Question 7 Montrer que le nombre de séquences de hachage laissant la case 0 vide, les cases 1 à k remplies et la case $k+1$ vide est

$$g(k,n,m)=\binom{n}{k}(k+1)^k\left(1-\frac{k}{k+1}\right)(m-k-1)^{n-k}\left(1-\frac{n-k}{m-k-1}\right)$$

Question 8 Montrer que la probabilité qu'une recherche infructueuse nécessite de parcourir k cellules pleines est :

$$p(k,n,m)=\frac{1}{m^n}\sum_{i=k}^{n} g(i,n,m)$$

Question 9 Montrer que le nombre moyen de tests nécessaire pour une recherche infructueuse est :

$$C'(n,m)=\sum_{k=0}^{n}(k+1)p(k,n,m)$$

On admettra que

$$C'(n,m)=\frac{1}{2}\left(1+\sum_{k=0}^{n}(k+1)\frac{n!}{m^k(n-k)!}\right)$$

On définit le **facteur de charge** $\alpha=n/m$.

Question 10 Montrer que

$$C'(n,m) \leq \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

2. Recherche fructueuse

On note $C(n,m)$ le nombre moyen de tests nécessaires lors d'une recherche fructueuse dans une table de hachage de taille m contenant n éléments.

Question 11

1. Justifier que $C'(n-1,m)$ le nombre moyen de tests nécessaire pour insérer une n-ème clé dans une table.
2. En déduire que
$$C(n,m) = \frac{1}{n}\sum_{k=0}^{n-1} C'(k,m)$$

Question 12 Montrer que

$$C(n,m) \leq \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

3. Conclusion

On a montré que la complexité des différentes opérations était majoré par le facteur de charge, indépendemment de n ou de m. En particulier, si l'on s'assure que la facteur de charge sera toujours inférieur à une constante définie à l'avance. On prendra en général comme facteur de charge maximal une valeur comme 0.5 ou 0.6 en s'assurant que la table de hachage a une taille convenable.

⁂

Historiquement, cette étude est très importante. En effet, c'est l'un des premiers résultats mathématiquement non triviaux d'analyse d'un algorithme. Plus précisément, c'est le premier résultat trouvé dans les années soixante par un jeune mathématicien de 24 ans, un certain Donald Knuth. Ce résultat influencera énormément sa carrière, puisqu'il décidera de consacrer sa vie à l'informatique et plus particulièrement à l'analyse des algorithmes. L'un des conséquences directes de ce choix sera l'écriture de la somme "*The Art of Computer Programming*" qui lui vaudra le prix Turing (équivalent informaticien du prix Nobel ou de la médaille Fields) ainsi que, lors de la réédition du deuxième volume de *The Art of* ..., la création du langage TEX qui permet d'écrire facilement et élégamment des documents scientifiques (même avec plein de formules mathématiques comme dans le présent sujet de TD).

Notons de plus que l'écriture de *The Art of ...* a débuté dans les années soixante. Prévu en sept volumes, le quatrième tome est paru en 2010. C'est que, entre le plan initial prévu et la rédaction, l'étude de l'informatique a véritablement explosé.

Enfin, une phrase circule sur Internet, attribuée tantôt à Bill Gates et tantôt à Steve Jobs, disant que si quelqu'un a lu, compris et assimilé *The Art of Computer Programming* dans son intégralité, il n'a qu'à envoyer un CV, il sera embauché de suite.

IV. Compléments mathématiques

Théorème - ... binomial d'Abel

Pour tous réels (ou complexes) x, y et z,

$$\sum_{k=0}^{n} \binom{n}{k} x(x-kz)^{k-1}(y+kz)^{n-k} = (x+y)^n$$

Question 13 Prouver le théorème binomial d'Abel.

Question 14 En posant

$$s(n,x,y) = \sum_{k=0}^{n} \binom{n}{k} (x+k)^{k+1}(y-k)^{n-k-1}(y-n),$$

montrer que :

$$\forall\, n \geq 1,\ s(n,x,y) = x(x+y)^n + ns(n-1,x+1,y-1).$$

Question 15 Prouver l'étape admise.