Structures de données probabilistes

Bruno Grenet

Septembre 2017

Introduction

Ce chapitre est consacré à l'étude de structures de données probabilistes. L'objectif est de concevoir des structures de données pour représenter des ensembles ordonnés avec des garanties (probabilistes !) sur les performances. Pour fixer les idées, les opérations que l'on souhaite effectuer sont :

— RECHERCHE d'un élément dans l'ensemble ;
— INSERTION d'un nouvel élément, s'il n'y était pas encore ;
— SUPPRESSION d'un élément.

D'autres opérations peuvent bien sûr être utiles, comme l' UNION de deux ensembles dont l'un ne contient que des *petites* valeurs et l'autre des *grandes* valeur, ou la PARTITION d'un ensemble en deux sous-ensembles, de part et d'autre d'un pivot. Le fait que nos structures de données permettent d'implanter ces opérations de manière efficace est laissé en exercice.

Cadre de travail. Dans toute la suite, les éléments que l'on manipule appartiennent à un univers U que l'on supposera pour simplifier égal à l'ensemble $\{1,\dots,u\}$ où u est un entier strictement positif. L'objectif est de représenter un ensemble $X \subset U$ de taille n de manière efficace. On notera en général $X = \{x_1,\dots,x_n\}$ avec $x_1 < x_2 < \cdots < x_n$.

Dans la suite, nous présentons différentes structures de données probabilistes. La première (les *tarbres*) est une méthode probabiliste pour assurer l'équilibre d'un arbre binaire de recherche. La deuxième (les *listes à raccourcis*) est basée sur la structure de liste chaînée et la troisième (les *tables de hachage*) est basée sur la structure de tableau.

1 Les tarbres [1]

Avant de voir leur version probabiliste, nous rappelons le fonctionnement des *arbres binaires de recherche* (ABR). On rappelle qu'un ABR est un arbre binaire dont les nœuds contiennent les éléments de l'ensemble X et qui vérifie la propriété suivante : si un nœud contient un élément x, tous les nœuds dans son sous-arbre gauche sont inférieurs (strictement [2]) à x, et tous les nœuds dans son sous-arbre droit sont supérieurs (strictement) à x. La RECHERCHE et l' INSERTION s'effectuent par de simples parcours de l'ABR. La SUPPRESSION est un peu plus complexe : on remplace le nœud x à supprimer par son successeur direct y (dans l'ordre sur les éléments) ; ce successeur direct ne pouvant pas avoir de fils gauche [3], on fait remonter le fils droit de y à la place de y. Ces différentes opérations ont toutes une complexité $O(h)$

1. Contraction des mots « tas » et « arbre », *Treaps* en anglais, contraction de *tree* et *heap*.
2. On suppose les éléments distincts deux à deux.
3. Sinon, ce fils gauche serait compris entre x et y.

où h est la hauteur courante de l'ABR. Cette structure de données a donc une bonne complexité *tant que l'on sait garder l'ABR équilibré*, c'est-à-dire de hauteur $O(\log n)$. De nombreuses techniques, relativement complexes, existent pour cela, comme les arbres rouges-noirs ou AVL.

Il existe une idée relativement simple, probabiliste, pour garder la structure équilibrée. Elle se base sur le constat que l'espérance de la hauteur d'un ABR est $O(\log n)$ quand on ajoute n éléments dans un ordre aléatoire. Pour simuler l'ajout dans un ordre aléatoire, l'idée est de rajouter une structure de *tas* à l'ABR. Rappelons qu'un tas est un arbre (binaire) dont les nœuds contiennent, comme dans le cas d'un ABR, les éléments d'un ensemble et qui vérifie la propriété suivante : l'élément contenu dans un nœud est strictement plus grand que tous les éléments se trouvant dans ses sous-arbres [4].

Définition 1.1. Un *tarbre* pour un ensemble de couples $\{(x_1, p_1), \ldots, (x_n, p_n)\}$ est un arbre binaire dont chaque nœud contient un couple (x_i, p_i) et qui respecte à la fois la structure d'ABR vis-à-vis des valeurs x_i et la structure de tas vis-à-vis des priorités p_i.

Théorème 1.2. *Tout ensemble de couples valeur-priorité peut être représenté par un tarbre. Ce tarbre est unique si les priorités sont deux-à-deux distinctes.*

Démonstration. Supposons que les priorités sont deux-à-deux distinctes. Par définition de la structure de tas, la racine du tarbre contient nécessairement le couple (x_i, p_i) dont la priorité est maximale. Pour respecter la structure d'ABR, il faut insérer tous les coupes (x_j, p_j) avec $x_j < x_i$ dans le sous-arbre gauche, et tous les couples (x_j, p_j) avec $x_j > x_i$ dans le sous-arbre droit. Par induction, on obtient l'existence et l'unicité.

Si les priorités ne sont pas distinctes, on peut les rendre distinctes en les modifiant d'une valeur infinitésimale. Supposons par exemple qu'il y a k couples de priorité p, et que la plus petite priorité supérieure strictement à p soit q. On peut définir ϵ tel que $p + (k-1)\epsilon < q$ ($\epsilon = (q-p)/k$ convient) et modifier les k priorités p par les priorités toutes distinctes p, $p+\epsilon$, …, $p+(k-1)\epsilon$. Si on fait ça pour chaque priorité qui apparaît plusieurs fois, on obtient un tarbre unique par la preuve précédente. Il suffit de remettre les bonnes priorités dans le tarbre obtenu.

□

L'opération de RECHERCHE est la même que pour un ABR en ignorant simplement la structure de tas des priorités. Pour l' INSERTION, l'algorithme est légèrement plus complexe, afin de préserver la structure de tas. On commence par insérer le nouveau couple (x, p) comme dans un ABR. On fait ensuite *remonter* le couple par rotations successives pour qu'il atteigne sa bonne profondeur dans l'arbre : notons (x', p') le parent de (x, p) ; si $p' < p$, on remplace (x', p') par (x, p) et (x', p') devient le fils (gauche ou droit, selon le signe de $x - x'$) de (x, p) ; si (x', p') avait un autre fils, il reste son fils ; on applique récursivement cet algorithme tant que le père de (x, p) a une priorité inférieure à p. Cet algorithme a une complexité $O(h)$ où h est la hauteur de l'arbre. La SUPPRESSION est similaire. Ainsi donc, ajouter la structure de tas ne change pas la complexité des opérations.

Pour un ensemble $X = \{x_1, \ldots, x_n\}$, on peut construire un tarbre de manière probabiliste. Pour chaque élément x_i, on tire aléatoirement (et indépendamment) une priorité p_i, puis on construit le tarbre associé aux couples (x_i, p_i). En théorie, on peut tirer les p_i uniformément selon une distribution de probabilité continue, de telle sorte qu'avec probabilité 1, les priorités sont distinctes deux-à-deux. En pratique, on peut par exemple tirer des priorités entières dans un ensemble assez grand pour limiter le nombre de collisions. Si les priorités ne sont pas distinctes, on peut départager deux couples de même priorité en

4. Ainsi la racine de l'arbre est le maximum du tas. Bien entendu, une définition strictement équivalente dans laquelle le minimum se trouve à la racine est possible.

choisissant celui qui a la plus petite valeur comme étant de priorité plus élevée. Dans la suite, on ignorera cette difficulté en se plaçant dans le cadre théorique de priorités tirées selon une distribution continue.

Théorème 1.3. *Soit T un tarbre probabiliste pour un ensemble X de taille n. Alors l'espérance de la hauteur de chaque nœud de T est $O(\log n)$.*

Démonstration. On note $X = \{x_1, \dots, x_n\}$ avec $x_1 < x_2 < \cdots < x_n$. Pour tout k, on définit la variable H_k égale à la hauteur de x_k dans T. On écrit $x_i \twoheadrightarrow x_j$ si x_i est un ancêtre de x_j dans T. Alors $\mathbb{E}[H_k] = \sum_i \mathbb{P}[x_i \twoheadrightarrow x_k]$: la hauteur de x_k est le nombre d'éléments qui sont ancêtres de x_k.

Pour $1 \le i \le n$, soit

$$X_{ik} = \begin{cases} \{x_i, x_{i+1}, \dots, x_k\} & \text{si } x_i \le x_k, \\ \{x_k, x_{k+1}, \dots, x_i\} & \text{sinon.} \end{cases}$$

En particulier, $X_{ik} = X_{ki}$ pour tous k et i. On va montrer que $x_i \twoheadrightarrow x_k$ si et seulement si x_i a la priorité maximale de l'ensemble X_{ik}, par récurrence sur $|k-i|$. C'est clair si $k = i$ car $|X_{ik}| = 1$.

Soit $x_j \in X_{ik}$ l'élément de priorité maximale, et supposons qu'il est différent de x_i. Considérons le sous-arbre de T qui contient x_i, x_j et x_k. Par hypothèse de récurrence, $x_j \twoheadrightarrow x_k$ car x_j est l'élément de priorité maximale de X_{jk} et $|k-j| < |k-i|$. Or x_i ne peut ni être un ancêtre de x_j (car il est de priorité moindre), ni sur le chemin entre x_j et x_k (car $x_i < x_j < x_k$). Donc $x_i \not\twoheadrightarrow x_k$. Réciproquement, supposons que $x_i \not\twoheadrightarrow x_k$, et notons x_j leur plus proche ancêtre commun : x_i est dans le sous-arbre gauche de x_j et x_k dans son sous-arbre droit, ou inversement. Dans les deux cas, $x_j \in X_{ik}$ et sa priorité est supérieure à celle de i.

Comme chaque élément de X_{ik} a la même probabilité d'être de priorité maximale, $\mathbb{P}[x_i \twoheadrightarrow x_k] = \frac{1}{|k-i|+1}$. On en déduit que pour tout k, l'espérance de H_k est

$$\begin{aligned}
\sum_{\substack{i=1 \\ i \neq k}}^{n} \mathbb{P}[x_i \twoheadrightarrow x_k] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\
&= \sum_{\ell=2}^{k} \frac{1}{\ell} + \sum_{m=2}^{n-k+1} \frac{1}{m} && (\ell = k-i+1, m = i-k+1) \\
&= H_k - 1 + H_{n-k} - 1 && (H_k : k\text{-ème nombre harmonique}) \\
&< \ln(k+1) + \ln(n-k) - 2 && (\forall k, H_k < \ln(k+1))
\end{aligned}$$

Ainsi, l'espérance de la hauteur de chaque nœud dans T est $O(\log n)$.

□

On en déduit facilement que les opérations de RECHERCHE, d' INSERTION et de SUPPRESSION ont une espérance de temps de calcul $O(\log n)$ dans le pire cas.

Remarque. La preuve précédente est très similaire à celle effectuée pour l'analyse de QUICKSORT. Ce n'est pas une coïncidence ! On peut voir la construction du tarbre probabiliste comme la construction d'un ABR pour X dans lequel on insérerait les éléments dans un ordre aléatoire. Il est relativement facile de montrer que cet algorithme est en fait exactement le même que QUICKSORT, exprimé dans un autre langage.

Remarque. La preuve précédente *ne démontre pas* que l'espérance de la hauteur de l'arbre est $O(\log n)$, car $\mathbb{E}[\max_k H_k] \neq \max_k \mathbb{E}[H_k]$! Il faut en réalité effectuer une preuve plus complexe pour obtenir ce résultat. Sans surprise d'après la remarque précédente, cette preuve plus complexe est dans la même veine que la preuve montrant que QUICKSORT s'exécute en temps $O(n \log n)$ avec grande probabilité.

2 Listes à raccourcis [5]

La structure de liste à raccourcis est construite sur la structure très classique de liste chaînée. Si on représente les éléments d'un ensemble totalement ordonné X à n éléments par une liste chaînée (fig. 2), la complexité des opérations de RECHERCHE, d' INSERTION et de SUPPRESSION est $O(n)$ dans le pire cas.

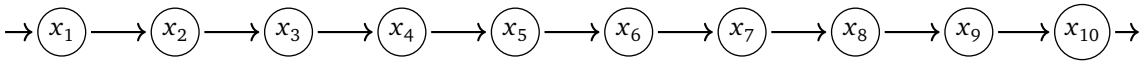

FIGURE 1 – Exemple de liste chaînée pour un ensemble de 10 éléments.

Pour améliorer cette complexité, on ajoute des *raccourcis*, c'est-à-dire des pointeurs supplémentaires qui permettent d'aller directement à un élément situé plus loin dans la liste, en sautant par au dessus ceux situés en chemin. Plus précisément, on crée une seconde liste qui ne contient que certains éléments de X. Par exemple avec $n = 10$, on peut créer une liste $x_1 \to x_4 \to x_7 \to x_{10}$. Pour chercher un élément, disons x_6, on parcourt en premier lieu la liste des raccourcis ; une fois sur x_4, on se rend compte que $x_7 > x_6$; on *descend* alors dans la liste principale, au niveau de x_4, puis on recherche x_6 de manière classique (fig. 2). Au lieu d'avoir suivi 5 pointeurs comme dans une liste classique, on aura suivi 1 pointeur dans la liste des raccourcis, puis 1 pointeur vers le bas, puis 2 pointeurs dans la liste principale. On économise donc un suivi de pointeur. Pour que cet algorithme de recherche fonctionne, on a sous-entendu le fait que chaque élément dans la liste des raccourcis possède un pointeur vers son équivalent dans la liste principale.

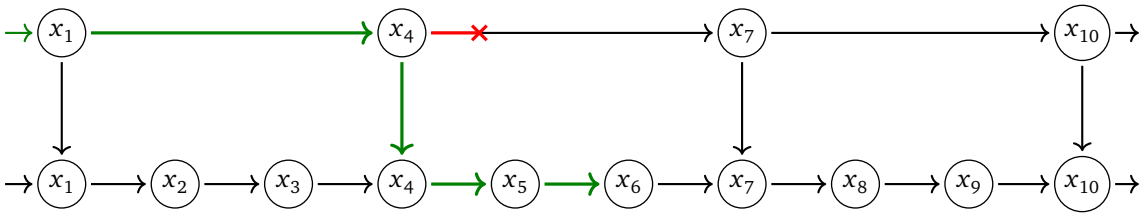

FIGURE 2 – Exemple de liste à raccourcis à deux niveaux pour un ensemble de 10 éléments. En vert, le chemin de recherche de l'élément x_6.

La vision en niveaux se prête parfaitement à une généralisation de l'idée. Au lieu d'avoir deux niveaux, on considère $h + 1$ niveaux, de 0 à h. On se convainc aisément qu'il faut, pour que l'algorithme de recherche continue à fonctionner de la même manière, que pour tout $i < h$, l'ensemble des éléments au niveau i contienne l'ensemble des éléments au niveau $i + 1$. De cette manière, on ne doit jamais revenir plus d'un pas en arrière. Concevoir des listes à raccourcis récursives de manière déterministe et analyser le temps des opérations de RECHERCHE, d' INSERTION et de SUPPRESSION est une tâche difficile qui fait l'objet de recherches récentes. Mais les listes à raccourcis récursives ont été imaginées à l'origine dans leur version probabiliste.

L'idée est de construire les niveaux supérieurs de manière probabiliste. Pour chaque élément de la liste de départ (niveau 0), on tire des bits aléatoires : tant qu'on tire un 1, on ajoute l'élément au niveau supérieur ; dès qu'on tire un 0, on s'arrête. Un élément de la liste va donc se retrouver aux niveaux 0 à i où i est le nombre de 1 consécutifs obtenus avant le premier 0 dans le tirage des bits aléatoires (fig. 2).

5. *Skip lists* en anglais. On pourrait traduire par liste à sauts si la structure de données appelée *jump list* n'existait pas…

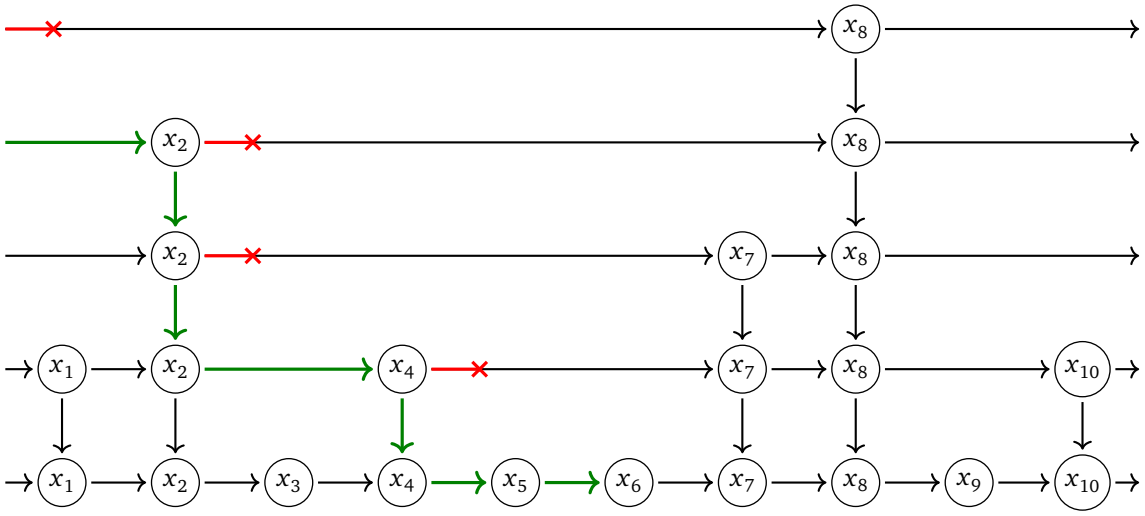

FIGURE 3 – Exemple de liste à raccourcis probabiliste pour un ensemble de 10 éléments. En vert, le chemin de recherche de l'élément x_6.

Pour l' INSERTION d'un nouvel élément, on commence par tirer des bits aléatoirement pour décider son niveau. Si son niveau dépasse celui des éléments déjà en place, il faut créer les nouveaux niveaux supplémentaires. Ensuite, il faut trouver son emplacement dans la liste de niveau 0 en faisant une RECHERCHE. À chaque niveau auquel le nouvel élément appartient, il faut mettre à jour les pointeurs pendant la recherche pour que le nouvel élément soit à sa place dans la liste du niveau. Il en est de même pour la SUPPRESSION d'un élément. Dans les deux cas, la complexité de l'opération est de l'ordre de celle de la recherche initiale.

Théorème 2.1. *Soit L une liste à raccourcis probabiliste pour un ensemble X de taille n, et H le niveau maximal de L. Alors*

$$\mathbb{E}[H] \leq \log n + 1.$$

Démonstration. Soit $x \in X$, et $H(x)$ le niveau maximal auquel appartient x. Alors $\mathbb{P}[H(x) \geq h] \leq 2^{-h}$ pour tout h. D'après l'inégalité de Boole,

$$\mathbb{P}[H \geq h] \leq \sum_{x \in X} \mathbb{P}[H(x) \geq h] = n/2^h$$

pour tout h. Puisque pour $h < \log n$, $n/2^h > 1$, on utilise dans ce cas la borne triviale $\mathbb{P}[H \geq h] \leq 1$. Alors

$$\begin{aligned}
\mathbb{E}[H] &= \sum_{1 \leq h < \log(n)} \mathbb{P}[H \geq h] + \sum_{h \geq \log(n)} \mathbb{P}[H \geq h] \\
&\leq \sum_{1 \leq h < \log(n)} 1 + \sum_{h \geq \log(n)} \frac{n}{2^h} \\
&< \log n + \sum_{i \geq 1} \frac{1}{2^i} = \log n + 1
\end{aligned}$$

avec le changement de variable $i = h - \lceil \log n \rceil$. □

Corollaire 2.2. *L'opération de* RECHERCHE *d'une liste à raccourcis probabiliste à n éléments a une complexité espérée* $O(\log n)$.

Démonstration. Remarquons en premier lieu que la borne sur le nombre de niveau ne suffit pas à conclure. En effet, il pourrait arriver que l'on descende très vite au niveau 0 puis qu'on suive un chemin de longueur $O(n)$. Il faut donc montrer que cette situation n'est pas la situation habituelle. D'autre part, l'opération de RECHERCHE n'est pas elle-même probabiliste : c'est l'INSERTION qui l'est, donc la construction de la liste à raccourcis. Autrement dit, une fois la liste construite, il existe pour chaque élément x_k un unique chemin de RECHERCHE qui y mène. On cherche donc à étudier l'espérance de la longueur de ce chemin.

On fixe un élément x_k. Pour chaque élément x_i de la liste, on note $x_i^0, \ldots, x_i^{h_i}$ les copies de x_i dans les niveaux 0 à h_i, où h_i est le niveau maximal auquel apparaît x_i. On ajoute un élément virtuel $x_0 < x_1$ et des nœuds virtuels $x_0^0, x_0^1, \ldots, x_0^H$ où $H = \max_{i>0} h_i$. Le chemin de RECHERCHE de x_k part (virtuellement) du nœud x_0^H pour terminer sur un nœud x_k^h. On peut supposer que lorsqu'on a rencontré x_k^h, on continue à suivre des pointeurs vers x_k^0.

Si on parcourt le chemin en sens inverse, le prédécesseur d'un nœud x_i^h est soit x_i^{h+1}, soit x_j^h avec $j < i$. Le premier cas est appelé un pointeur *vers le bas*, le second un pointeur *vers la droite*. De plus, si le nœud x_i^{h+1} existe, il s'agit nécessairement du prédécesseur de x_i^h puisque le chemin ne peut pas descendre vers un x_j^h, $j < i$, si x_i^{h+1} existe. On peut imaginer que la liste à raccourcis est construite pendant qu'on parcourt le chemin en sens inverse : à chaque nœud x_i^h, on décide soit que le prédécesseur est x_i^{h+1}, soit un x_j^h avec $j < i$. D'après la construction de la liste, si x_i^h existe, la probabilité pour que x_i^{h+1} existe est $1/2$ (puisque x_i^{h+1} existe si le bit aléatoire que l'on tire à ce moment-là est à 1). Autrement dit, en imaginant que le chemin est construit à la volée pendant qu'on le parcourt en sens inverse, on se rend compte qu'avec probabilité $1/2$ on ajoute un pointeur vers le bas, et avec probabilité $1/2$ un pointeur vers la droite.

Si le chemin est de longueur ℓ, l'espérance du nombre de pointeur vers la droite est

$$\sum_{t=1}^{\ell} \mathbb{P}[\text{pointeur } t \text{ vers la droite}] = \ell/2$$

(par linéarité). Autrement dit, il y a autant de pointeurs vers la droite que vers le bas. Or on connaît le nombre de pointeur vers le bas ! C'est la hauteur maximale de la liste. Donc $\mathbb{E}[\text{nb de pointeurs vers le bas}] = \mathbb{E}[H] = O(\log n)$. Et l'espérance du nombre total de pointeur est au plus $2\mathbb{E}[H] = O(\log n)$.

□

3 Tables de hachages

Les tables de hachages sont une autre structure de données pour représenter des ensembles, probabiliste par nature. L'idée est d'utiliser la technique de l'empreinte (*fingerprinting*) pour représenter un ensemble de manière compacte et accélérer ainsi les algorithmes de RECHERCHE, INSERTION et SUPPRESSION. Les meilleures constructions atteignent une complexité $O(1)$ pour ces opérations. Dans les parties précédentes, on se contentait de faire des comparaisons sur les entiers de l'ensemble X : on peut démontrer qu'avec cette restriction, la complexité $O(\log n)$ obtenue est l'optimale. Les tables de hachages dépassent cette limitation en s'autorisant des calculs plus complexes sur les données.

On rappelle le contexte dans lequel on se place. On dispose d'un ensemble X inclus dans un univers $U = \{1, \ldots, u\}$. On note n la taille de X.

Définition 3.1. Une *table de hachage* pour $X \subset U$ est une structure de données constituée d'un tableau T de taille m et d'une *fonction de hachage* $h : U \to \{0, \dots, m-1\}$.

On dit qu'il y a une *collision* entre deux éléments distincts x et y lorsque $h(x) = h(y)$.

On s'intéresse au cas où $u > m$. Sinon, on retrouve la structure classique de tableau en prenant l'identité $h : x \mapsto x$ comme fonction de hachage.

Une table de hachage s'utilise quasiment comme un tableau classique. Pour stocker un élément x, on calcule le *haché* $h(x)$ de x, et on met x dans la case $T[h(x)]$. S'il y a des collisions, c'est-à-dire deux éléments distincts x et y tels que $h(x) = h(y)$, la case correspondante de T doit contenir (au moins) deux valeurs. Pour cela, à la différence d'un tableau classique, le tableau de la table de hachage contient des listes chaînées d'éléments. La RECHERCHE s'effectue de la manière suivante : on calcule le haché $h(x)$ de l'élément x à rechercher, puis on parcourt la liste contenue dans la case $T[h(x)]$. De même, l'INSERTION et la SUPPRESSION s'effectuent en calculant $h(x)$ puis en insérant ou supprimant x de la liste contenue dans $T[h(x)]$ avec les algorithmes classiques pour les listes chaînées.

Hypothèses. Les résultats de cette partie sont basées sur deux hypothèses :

— l'accès à une case quelconque de T se fait en temps $O(1)$;

— le calcul de $h(x)$ s'effectue en temps $O(1)$.

La première hypothèse signifie qu'on est dans un modèle de calcul avec accès direct à la mémoire (modèle RAM). Ce modèle est tout à fait réaliste tant que la taille m du tableau est raisonnable. La deuxième hypothèse dépend bien entendu des fonctions de hachage considérées. En pratique, les fonctions de hachage utilisées vérifient cette hypothèse.

Sous ces hypothèses, les algorithmes de RECHERCHE, d'INSERTION et de SUPPRESSION s'effectuent en temps $O(1 + \ell(x))$ dans le pire cas, où $\ell(x)$ est la taille de la liste située en case $T[h(x)]$: le calcul de $h(x)$ s'effectue en temps $O(1)$, puis le parcours de la liste en temps $O(\ell(x))$.

Tout l'enjeu des tables de hachage repose donc sur la construction de fonctions de hachage performantes, c'est-à-dire garantissant un nombre de collisions petit. Comme on doit assurer cette propriété quelque soit l'ensemble X considéré, la seule méthode possible est de recourir à de l'aléatoire. On peut en réalité distinguer plusieurs *modèles* de fonctions de hachage. On en utilisera deux dans ces notes.

Définition 3.2. Soit $U = \{1, \dots, u\}$ et m un entier.

— **Modèle aléatoire** : une *fonction de hachage aléatoire* est une fonction choisie de manière aléatoire uniforme dans l'ensemble des fonctions de U dans $\{0, \dots, m-1\}$.

— **Modèle universel** : une famille de fonctions est dit *universelle* [6] si pour tout $x \neq y$, $\mathbb{P}[h(x) = h(y)] \leq 1/m$; une *fonction de hachage universelle* est une fonction choisie de manière aléatoire uniforme dans une famille de fonctions universelle.

Le modèle aléatoire permet de prouver des résultats très forts mais a l'inconvénient de ne pas être très réaliste : pour représenter une fonction de hachage totalement aléatoire, il faudrait $u \log m$ bits et l'évaluation de h ne serait absolument pas en $O(1)$. La motivation de la définition de famille universelle est que $\mathbb{P}[h(x) = h(y)] = 1/m$ dans le modèle aléatoire. Le modèle universel cherche donc à mimer le modèle aléatoire. Les résultats que l'on prouve dans ce modèle sont moins forts, mais on verra que l'on sait construire des familles universelles qui permettent l'évaluation en temps constant de $h(\cdot)$.

D'autres modèles sont envisageables et envisagés, même si on n'en parlera pas dans ces notes. D'une part, la notion d'universalité peut être renforcée (on obtient des résultats plus forts mais il est plus difficile de construire des familles rentrant dans le modèle) ou amoindrie (on obtient des résultats un peu moins forts, mais on dispose de nombreuses constructions de familles pour ces modèles). D'autre

6. Ou 2-universelle.

part, des modèles bien différents peuvent être utilisés. C'est le cas par exemple des fonctions de hachage *cryptographiques*, qui ont en général des propriétés très fortes mais qui reposent sur des conjectures de théorie de la complexité. D'autre part, certaines fonctions de hachage sont de l'ordre de la recette de cuisine sans qu'on ait de propriétés prouvées, comme par exemple la fonction utilisée pour hacher les chaînes de caractères dans le langage Python.

Dans la suite, on s'intéresse à borner le nombre de collisions pour une fonction de hachage prise aléatoirement, dans les modèles aléatoire et universel. On construit ensuite explicitement une famille universelle. Enfin, on remet en cause l'utilisation de listes chaînées dans les cases du tableau, pour obtenir du *hachage parfait* : en remplaçant les listes par des tables de hachage (on a donc une table de tables), on montre que l'espérance de temps de calcul dans le pire cas des opérations RECHERCHE, INSERTION et SUPPRESSION est $O(1)$.

3.1 Borner les collisions

Dans cette partie, nous montrons que les deux modèles, aléatoire et universel, permettent de montrer des bornes sur le nombre de collisions.

On se place dans le modèle aléatoire, et pour fixer les idées on suppose que $m = n$, c'est-à-dire qu'on crée une table de hachage ayant autant de cases que d'éléments à insérer. Puisque h est choisie uniformément parmi les fonctions de U dans $\{0,\dots,m-1\}$, borner le remplissage maximal d'une case de la table est équivalent au problème suivant : étant données n boules jetées uniformément et indépendamment dans n urnes, quelle est l'espérance du remplissage de l'urne la plus remplie ?

Théorème 3.3. *Si on jette n boules dans n urnes uniformément et indépendamment, l'urne la plus remplie contient $O(\log n/\log\log n)$ boule avec probabilité au moins $1-1/n^{O(1)}$.*

Démonstration. On numérote les urnes de 1 à n, et pour tout j on note X_j la variable aléatoire qui compte le nombre de boules dans l'urne j. Alors

$$\mathbb{P}\big[X_j \geq k\big] \leq \binom{n}{k}\left(\frac{1}{n}\right)^k$$

car pour chaque sous-ensemble de k boules, la probabilité que toutes les boules du sous-ensemble appartiennent à l'urne j est $(1/n)^k$, et qu'il y a $\binom{n}{k}$ sous-ensembles de tailles k. De plus, $\binom{n}{k} \leq n^k/k!$, donc la probabilité est bornée par $1/k!$. Or on peut borner inférieurement $k!$ par $k^{k/2}$. Pour cela, on remarque que $k! > (k/e)^k = k^k/(e^2)^{k/2} > k^{k/2}$ pour $k > e^2$. Et on peut vérifier aisément que la borne reste vraie pour k entre 1 et $e^2 < 8$.

On pose maintenant $k = c\log n/\log\log n$ pour une certaine constante $c > 0$. Pour n suffisamment grand, $c\log n/\log\log n \geq \sqrt{\log n}$: en effet, $c\log n/\sqrt{\log n}\log\log n = c\sqrt{\log n}/\log\log n$, qui tend vers $+\infty$ quand n tend vers $+\infty$. D'où, pour n suffisamment grand,

$$1/k! < k^{-k/2} = \left(\frac{\log\log n}{c\log n}\right)^{c\log n/2\log\log n} \leq \left(\sqrt{\log n}\right)^{c\log n/2\log\log n} = \frac{1}{n^{c/4}}.$$

Chaque urne contient donc plus de $c\log n/\log\log n$ boules avec probabilité $\leq 1/n^{c/4}$. D'où

$$\mathbb{P}\left[\max_j X_j \geq \frac{c\log n}{\log\log n}\right] \leq \frac{n}{n^{c/4}} = \frac{1}{n^{c/4-1}}.$$ □

Remarque. On peut montrer la borne inférieure correspondante, à savoir que le nombre de boules dans l'urne la plus remplie est $\Omega(\log n/\log\log n)$. La borne du théorème 3.3 est optimale.

La conséquence de ce théorème est que dans une table de hachage dans le modèle aléatoire, la case la plus remplie du tableau contient $O(\log n/\log\log n)$ élément avec forte probabilité.

On se place maintenant dans le modèle universel, c'est-à-dire qu'on suppose disposer d'une famille universelle H de fonctions de hachage, et qu'on choisit uniformément un élément $h \in H$. Un premier résultat facile à obtenir est une borne sur l'espérance du nombre d'élément dans une case du tableau [7]. En effet, si on considère un élément x qui n'appartient pas à l'ensemble X (c'est-à-dire qui n'est pas dans la table) et qu'on note $\ell(x)$ le nombre d'éléments dans la case $T[h(x)]$, on a

$$\mathbb{E}[\ell(x)] = \sum_{y\in X} \mathbb{P}[h(x) = h(y)] \leq \frac{n}{m}$$

car dans le modèle universel, $\mathbb{P}[h(x) = h(y)] \leq 1/m$ pour tous $x \neq y$. De même, si $x \in X$, on obtient $\mathbb{E}[\ell(x)] \leq 1 + (n-1)/m$. Ainsi, quand $m = O(n)$, le temps de RECHERCHE moyen est $O(1)$. Ceci ne suffit pas à borner le temps de RECHERCHE dans le pire cas. On va en effet montrer que ce temps dans le pire cas est borné par $O(\sqrt{n})$. On montre en réalité un résultat beaucoup plus général.

Théorème 3.4. *Soit H une famille universelle de fonctions de hachages de U dans $\{0,\dots,m-1\}$, et X un ensemble de n éléments. On note M le nombre d'éléments dans la case la plus remplie du tableau lorsqu'on insère tous les éléments de X. Alors*

$$\mathbb{P}\left[M \geq n\sqrt{\frac{2}{m}}\right] \leq 1/2.$$

En particulier,

- *si $m = n$, le remplissage maximal est $\sqrt{2n}$ avec probabilité $\geq 1/2$;*
- *si $m \geq n^2$, le remplissage maximal est 1 avec probabilité $\geq 1/2$.*

Démonstration. On note C le nombre total de collisions pour l'ensemble X, c'est-à-dire le nombre de couples $(x,y) \in X^2$, $x < y$, tels que $h(x) = h(y)$. Alors

$$\mathbb{E}[C] = \sum_{x\neq y} \mathbb{P}[h(x) = h(y)] \leq \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2}\frac{1}{m} \leq \frac{n^2}{2m}.$$

En utilisant l'inégalité de Markov, il vient que

$$\mathbb{P}\left[C \geq \frac{n^2}{m}\right] \leq \mathbb{P}[C \geq 2\mathbb{E}[C]] \leq \frac{1}{2}.$$

Si une case contient M éléments de X, ces M éléments forment $\binom{M}{2}$ collisions. Donc $C \geq \binom{M}{2}$. On en déduit que $\mathbb{P}\left[\binom{M}{2} \geq n^2/m\right] \leq 1/2$, d'où le résultat.

□

On note que le deuxième cas particulier peut être montré très simplement sans le résultat général. À partir de la borne sur l'espérance de C, on en déduit que si $m \geq cn^2$, l'espérance du nombre de collisions est bornée par $1/c$. Ce qui implique la probabilité qu'il y ait au moins une collision est aussi bornée par $1/c$.

7. Ce qui n'est pas la même chose que l'espérance du maximum de remplissage !

Ce que l'on vient de montrer est donc qu'on a deux solutions extrêmes possibles si on dispose d'une famille universelle : soit on crée une table de taille $O(n)$ et la RECHERCHE s'effectue en temps $O(\sqrt{n})$ dans le pire cas, soit on crée une table de taille $O(n^2)$ et la RECHERCHE s'effectue en temps $O(1)$ dans le pire cas. Dans la partie 3.3, on verra comment obtenir le meilleur des deux mondes.

3.2 Une famille universelle

On a vu dans la partie précédente l'intérêt de disposer de familles de fonctions de hachage universelles. On va maintenant définir une telle famille.

La famille qu'on construit est définie en fonction d'un nombre premier p supérieur au nombre total d'éléments de l'univers U. Pour tout entier $a \in \{1, \dots, p-1\}$, on définit $h_a : U \to \{0, \dots, m-1\}$ par $h_a(x) = (ax \mod p) \mod m$. On rappelle que la notation $x \mod y$ représente l'unique entier z compris entre 0 et $y-1$ tel qu'il existe $k \in \mathbb{Z}$ tel que $x = z + ky$.

Théorème 3.5. *La famille $H = \{h_a : 0 < a < p\}$ est universelle.*

Pour démontrer ce résultat, on utilise le résultat facile suivant de théorie des nombres.

Lemme 3.6. *Soit $0 < z, c < p-1$. Alors il existe un unique $a < p$ tel que $az \mod p = c$.*

Démonstration. Supposons qu'il existe $a \neq b$ tels que $az \mod p = bz \mod p$. Alors $(a-b)z = 0 \mod p$. Autrement dit $(a-b)z = kp$ pour un entier k. Comme p est premier et $z < p$, $a-b$ est lui-même divisible par p. Mais puisque $a, b < p$, $a-b$ est nécessairement nul. De même, $az \mod p \neq 0$ si $0 < a, z < p$.

Ainsi, $\{az \mod p : 0 < a < p\}$ contient $p-1$ éléments distincts non nuls inférieurs ou égaux à $p-1$: cet ensemble est donc égal à $\{1, \dots, p-1\}$ et il existe un unique a tel que $az \mod p = c$.

□

On en déduit une démonstration du théorème 3.5.

Démonstration. On veut montrer que pour tout $x \neq y$, $\mathbb{P}[h_a(x) = h_a(y)] \leq 1/m$. Or

$$\begin{aligned} h_a(x) - h_a(y) &= (ax \mod p) \mod m - (ay \mod p) \mod m \\ &= (ax \mod p - ay \mod p) \mod m \\ &= ((ax - ay) \mod p) \mod m \\ &= h_a(x - y) \end{aligned}$$

En posant $z = x - y$, on a $z \neq 0$ et $h_a(z) = 0$ si et seulement si $h_a(x) = h_a(y)$. Puisque x et y sont arbitraires, il faut et il suffit de montrer que pour tout $z \neq 0$, $\mathbb{P}[h_a(z) = 0] \leq 1/m$.

On fixe donc un z arbitraire non nul. Alors $h_a(z) = 0$ si $az \mod p$ est un multiple de m. Il y a $\lfloor (p-1)/m \rfloor$ multiples de m strictement inférieurs à p. Pour chaque multiple $c < p$ de m, il y a un unique a tel que $az \mod p = c$ d'après le lemme 3.6. Il y a donc $\lfloor (p-1)/m \rfloor$ valeurs de a telles que $h_a(z) = 0$. Alors $\mathbb{P}[h_a(z) = 0] = \lfloor (p-1)/m \rfloor / (p-1) \leq 1/m$.

□

Remarque. Un autre modèle très utile est celui de famille *fortement universelle* dans lequel on impose la condition plus restrictive que pour tout $x \neq y$ et tout $s, t \in \{0, \dots, m-1\}$, $\mathbb{P}[h(x) = s \text{ et } h(y) = t] \leq 1/m^2$. On peut montrer que la famille des fonctions $h_{a,b} : x \mapsto (ax + b \mod p) \mod m$ est fortement universelle.

3.3 Hachage parfait

Dans cette partie, on atteint enfin notre but : une fonction de hachage de taille $O(n)$ avec un temps de RECHERCHE $O(1)$ dans le pire cas. Bien sûr, les bornes de tailles et de temps de calcul sont des bornes sur les espérances.

L'idée de la construction est assez simple. On utilise une table de hachage principale de taille $m = n$. Il y aura donc nécessairement des collisions dans la table. Mais au lieu d'utiliser des listes chaînées pour régler ces collisions, on crée pour chaque case du tableau une table de hachage secondaire. Si la case contient n_j éléments, cette table secondaire aura comme taille n_j^2. D'après l'analyse effectuée précédemment, la table secondaire n'aura pas de collision avec probabilité au moins $1/2$. En cas de collision, on recrée la table secondaire avec une nouvelle fonction de hachage aléatoire.

Il faut montrer que cette technique ne fait pas trop augmenter la taille de la table globale.

Théorème 3.7. *Soit h une famille universelle de fonctions de hachage de U dans $\{0,\dots,n-1\}$, et $h \in H$ une fonction choisie uniformément. Si on insère n éléments dans la table à l'aide de h, et qu'on dénote par n_j le nombre d'éléments en case j (pour $0 \le j < n$), alors $\mathbb{E}\left[\sum_j n_j^2\right] < 2n$.*

Démonstration. Soit X l'ensemble des n éléments insérés dans la table. Notons $A_{x,j}$ la variable indicatrice de l'égalité $h(x) = j$: $A_{x,j} = 1$ si $h(x) = j$ et 0 sinon. Alors

$$\begin{aligned}
\sum_{j=0}^{n-1} n_j^2 &= \sum_{j=0}^{n-1}\left(\sum_{x\in X} A_{x,j}\right)^2 = \sum_{j=0}^{n-1}\left[\left(\sum_{x\in X} A_{x,j}\right)\left(\sum_{y\in X} A_{y,j}\right)\right] \\
&= \sum_{j=0}^{n-1}\left[\sum_{x\in X}\left(\sum_{y\in X} A_{x,j}A_{y,j}\right)\right] = \sum_{j=0}^{n-1}\sum_{x\in X} A_{x,j}^2 + 2\sum_{j=0}^{n-1}\sum_{x\neq y} A_{x,j}A_{y,j} \\
&= \sum_{x\in X}\sum_{j=0}^{n-1} A_{x,j}^2 + 2\sum_{x\neq y}\sum_{j=0}^{n-1} A_{x,j}A_{y,j}.
\end{aligned}$$

La somme $\sum_j A_{x,j}^2$ est égale à 1 pour tout x puisque x est haché à une seule position. Donc la première somme vaut n. La somme $\sum_j A_{x,j}A_{y,j}$ compte le nombre de collisions entre x et y, et vaut donc 0 ou 1. L'espérance de cette somme est donc égale à la probabilité que $h(x) = h(y)$, c'est-à-dire $\le 1/n$ car $n = m$. Par linéarité de l'espérance,

$$\mathbb{E}\left[\sum_j n_j^2\right] \le n + 2\sum_{x\neq y} 1/n = n + \frac{n(n-1)}{n} \le 2n-1. \qquad \square$$

On en conclut que la somme des tailles des tables secondaires est borné par $2n-1$. En ajoutant la table primaire, la taille totale de la structure de données est $O(n)$. L'espérance du temps de RECHERCHE dans le pire cas est $O(1)$ puisque pour chercher un élément x, il suffit de calculer la case $i = h(x)$ de la case primaire à laquelle il appartient, puis la case $j = h_i(x)$ de la table secondaire contenue en case i. Ces deux calculs s'effectuent en temps constant par hypothèse.

Pour l'INSERTION, si l'élément doit être inséré en case i de la table primaire, et que cette case contient déjà n_i éléments, on essaie d'abord de l'insérer de manière standard. S'il y a collision, il faut créer une

nouvelle table secondaire de taille $(n_i + 1)^2$ et insérer les $n_i + 1$ éléments dedans, en recommençant tant qu'il y a collision. Dans un contexte *statique* (la table est remplie au départ, et on ne fait que des RECHERCHEs ensuite), on peut commencer par calculer tous les $h(x)$ pour $x \in X$, et créer les tables secondaires de la bonne taille. On montre alors que l'espérance du temps de calcul dans le pire cas est $O(n)$. Dans un contexte *dynamique* (avec des INSERTIONs et des RECHERCHEs entremélées, avec éventuellement aussi des SUPPRESSIONs), le fait de devoir reconstruire la table s'il y a collision fait que l'INSERTION *n'est pas* en pire cas $O(1)$. Cependant, on peut montrer qu'elle est en temps *amorti* $O(1)$, c'est-à-dire que si on effectue un certains nombres d'INSERTIONs, le temps pris par celles-ci est *en moyenne* $O(1)$.