10. *Tableau de hachage*

LA TABLE DE SYMBOLES OU DICTIONNAIRE[1] (*symbol table/dictionary/map*) est un type abstrait de données représentant un ensemble d'objets avec clés, supportant une opération de recherche par clé. Typiquement (mais non pas toujours !) les clés sont comparables (abstraction : nombres naturels).

[1] W(en):*symbol table*

Opération principale :

- ⋆ search(k) : recherche d'un élément à clé k ⟵ peut être fructueuse ou infructueuse.

Si dictionnaire modifiable («dynamique») :

- ⋆ insert(x) : insertion de l'élément x (clé+info)
- ⋆ delete(k) : supprimer élément avec clé k

Avec clés uniques : search(k) retourne l'élément x avec clé k si un tel existe, ou null sinon ; insert(x) *remplace* l'élément avec la même clé, si un tel est déjà présent dans l'ensemble.

Implémentations. On a vu dés implémentations naïves : par liste chaînée (non-triée) ou tableau (trié ou non-trié). Notre structure pour implanter le TA table de symboles d'une façon tres efficace : **tableau de hachage**. (Plus tard on examinera les arbres de recherche, aussi utilisées souvent pour implémenter le TA.)

Applications.

- ⋆ recherche sur Web (mot ⟶ list de pages avec mot), dans une banque de données ou texte (mot ⟶ pages d'occurrence) ;
- ⋆ génomique (BLAST : recherche des occurrences d'une séquence moléculaire) ;
- ⋆ compilateurs/interpreteurs (variables, méthodes, procédures, . . .) ;
- ⋆ réseautage (nom ⟶ addresse IP)

10.1 *Inverted index*

Supposons que les clés sont des entières $\mathcal{U} = 0, 1, 2, \ldots, M-1$. On peut alors utiliser un tableau $\mathsf{T}[0..M-1]$, et mettre l'élément avec clé k dans la k-ème cellule. Opérations en $O(1)$ mais mémoire de taille $\Theta(M)$. N'est utile que si M n'est pas trop grand.

Clés multiples ? On peut facilement accommoder des éléments avec clés multiples (p.e., personnes avec date d'anniversaire — mois, jour — comme clé) : mettre des listes dans les cellules.

10.2 Hachage et les collisions

On peut accommoder des clés non-entières ou d'un espace de clés $\mathcal{U}$ trop grand. On utilise une **fonction de hachage** $h\colon \mathcal{U} \mapsto \{0, 1, \ldots, M-1\}$ pour définir l'emplacement d'une clé $k \in \mathcal{U}$ dans le tableau.

Définition 10.1. *Deux clés k, k' sont **en collision** si $h(k) = h(k')$.*

Hachage uniforme. On veut assurer que $h(k)$ est uniforme : il faut une fonction h telle que

- ⋆ [uniformité] $h(k) = i$ avec probabilité $1/M$ pour tout $i = 0, 1, 2, \ldots, M-1$.
- ⋆ [indépendance] pour plusieurs clés $(k_1, k_2, \ldots, k_m)$ $\bigl(h(k_1), \ldots, h(k_m)\bigr)$ a une distribution uniforme.

Mauvaise nouvelle : mais même avec une distribution uniforme, on aura des collisions quand $M = o(n^2)$.

W(fr):paradoxe des anniversaires

Théorème 10.1 (Birthday paradox). *Avec des clés uniformement distribués, on a au moins une collision avec probabilité $> 1/2$ quand $n > 1.18\sqrt{M}$.*

Démonstration. Probabilité d'aucune collision : $p = 1\left(1 - \frac{1}{M}\right)\left(1 - \frac{2}{M}\right)\cdots\left(1 - \frac{n-1}{M}\right) < \prod_{i=0}^{n-1} e^{-i/M} = \exp\left(-\frac{(n-1)n}{2M}\right)$. Avec $n/\sqrt{M} > \sqrt{2\ln 2} = 1.177\cdots$, on a $p < 1/2$. ■

Comment choisir une fonction de hachage ? On ne connaît pas la distribution des clés ! Heureusement, il existe des méthodes qui mènent à une distribution proche à uniforme dans la grande majorité d'applications.

Méthode de la division. On utilise $h(k) = k \bmod M$. Il faut bien choisir M pour éviter la réduction de l'espace de valeurs de hachage à cause des clés non-aléatoires :

- ⋆ éviter $M = 2^j$: les derniers j bits déterminent h
- ⋆ éviter $M = 10^j$: les derniers j chiffres d'une clé décimale déterminent h

$\Rightarrow$ choisir un nombre premier loin de 2^j et 10^j.

Méthode de la multiplication. On utilise $h(k) = \lfloor M\{\gamma k\}\rfloor$ avec une valeur flottante γ (partie fractionnaire : $\{x\} = x - \lfloor x \rfloor$). La dispersion des valeurs de hachage dépend principalement de γ. Un bon choix est $\gamma = \frac{\sqrt{5}-1}{2}$. Ici, on choisit une taille $M = 2^p$ pour un calcul rapide avec opérations entières (multiplication, décalage de bits). Calcul rapide pour un entier k representé sur w bits [p.e., $w = 32$ pour `int` de Java] : écrire $\gamma = A/2^w$, alors `h = (A*k) >>> (w-p)` (» dénote décalage de bits vers la droite).

Clés composées — hachage universel. On a une clé de r caractères : $k = \langle k_1, k_2, \ldots, k_r\rangle$ (p.e., `String`). Fonction de hachage $h^{(r)}(k) = \left(\sum_{i=1}^{r} h_i(k_i)\right) \bmod$

M avec des fonctions de hachage $h_i(x)$ choisies «au hasard». En pratique, on utilise une règle approximative simple comme $h_i(x) = a^{r-i} \cdot x$ (où a est un nombre premier) : initialiser $h \leftarrow 0$; **for** $i \leftarrow 1, 2, \ldots, r$ **do** $h \leftarrow (a \cdot h + k_i) \bmod M$.

10.3 *Tableau de hachage*

Qu'est-ce qu'on fait lors des collisions ?

W(fr):table de hachage

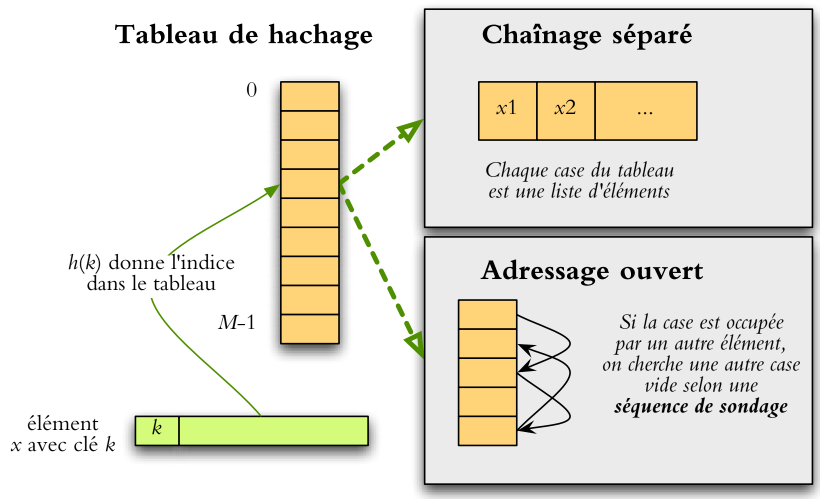

FIG. 1: Deux stratégies fondamentales

☞ En **chaînage séparé**, chaque case du tableau contient une liste d´éléments avec clés qui donnent la même valeur de hachage.

☞ En **adressage ouvert**, tous les éléments sont placés directement dans le tableau. La suite de cases à essayer pour le placement est définie par une séquence de sondage qui commence avec l'indice $h(k)$.

Un tableau de hachage performe bien dans **le cas moyen** — dans le pire cas, la performance est comme pour une liste chaînée ($\Theta(n)$ pour insérer ou rechercher). Idéalement, on veut utiliser une fonction de hachage qui mène à une **distribution uniforme** de $h(k)$. La performance moyenne est déterminée par la **facteur de charge** $\alpha = n/M$ quand on a n éléments dans le tableau.

Chaînage séparé

On utilise une liste chaînée (chaque case donne la tête de sa liste), ou un tableau pour stocker les éléments à chaque indice. Les éléments sont non-triés en général (surtout si les clés ne sont pas comparables), parce que α (longueur moyenne d'une liste) reste assez petite dans toutes les implantations efficaces.

Adressage ouvert

En adressage ouvert, on fait la **sondage** (*probing*) d'une séquence de positions : dépend de la clé à insérer. L'adressage ouvert ne permet pas

$\alpha > 1$. On examine les cases $h_0(k)$, $h_1(k)$, ... avec une fonction f : $h_i(k) = h(k) + f(i,k) \bmod M$. La fonction f représente la **stratégie de résolution de collision**.

Méthodes de sondage :

- ⋆ sondage linéaire $h_i(k) = h(k) + ic$. Cela ne dépend pas de la clé k, $c = 1$ est typique.
- ⋆ double hachage $h_i(k) = h(k) + ih'(k)$ avec fonction de hachage auxiliaire h'

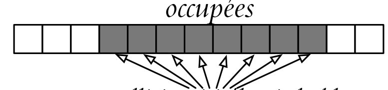

FIG. 2: Grappe forte avec sondage linéaire

Sondage linéaire. (*Linear probing*) : $h(k), h(k)+1, h(k)+2, \ldots$. Mène à la **grappe forte** (*primary clustering*) —blocs de cases occupées.

Double hachage. Généralisation de sondage linéaire : $h(k), h(k)+c, h(k)+2c, h(k)+3c, \ldots$. Ici, c dépend de la clé k : $c = h'(k)$. Cette méthode est très proche d'une résolution idéale. Exemples : $h'(x) = 1 + x \bmod M'$ avec $M' < M$ ou $h'(x) = M' - (x \bmod M')$.

Suppression. On utilise souvent une **approche paresseuse** (*lazy deletion*) : suppression = remplacement par une sentinelle. search doit passer les sentinelles, insert peut les recycler.

Performances : temps moyen en fonction de la facteur de charge α

Dans le pire des cas, quand on a toutes clés en collision, les opérations s'exécutent en $\Theta(n)$. Par contre, en moyenne et typiquement (=avec une probabilité proche de 1), le temps de calcul est fabuleux : $\Theta(1)$.

	chaînage séparé (nœuds examinés)	**sondage linéaire** (cases examinées dans la séquence de sondage)	**double hachage**
recherche fructueuse	$\frac{1+\alpha}{2}$	$\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$	$\approx \frac{1}{n}\sum_{i=0}^{n-1} \frac{1}{1-i/M} \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
recherche infructueuse	$1+\alpha$	$\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\approx 1 + \alpha + \alpha^2 + \cdots \approx \frac{1}{1-\alpha}$
insertion	1	*même que recherche infructueuse*	*même que recherche infructueuse*