Chapître III. Tables de hachage

1. Structures de données

La notion d'ensemble est fondamentale pour les mathématiques et aussi pour l'informatique. Les ensembles mathématiques sont stables.

Les ensembles de l'algorithmique manipulés par les algorithmes peuvent croître, diminuer, ou subir d'autres modifications au cours du temps. On les appèle ensembles dynamiques.

Les types d'opération à effectuer sur les ensembles dynamiques varient d'un algorithme à l'autre.

Alors on utilise les différentes structures de données (abrév. SD) pour la représentation et le maintien des ensembles dynamiques.

Un ensemble dynamique qui supporte les opérations RECHERCHER, INSERER et SUPPRIMER est appelé un <u>dictionnaire</u>.

Opérations sur les ensembles dynamiques

$RECHERCHER(S, k)$ Une requête qui, etant donnés un ensemble S et une valeur de clé k, retourne un pointeur x sur un élément y de S tel que k est la valeur de clé de y, ou NIL si il n'y a pas d'élément $y \in S$ pour lequel la valeur de clé est k.

$INSERER(S, x)$ Une opération de modification qui ajoute à l'ensemble S l'élément y pointé par pointeur x.

$SUPPRIMER(S, x)$ Une opération de modification qui, étant donné un pointeur x vers un élément y de l'ensemble S, élimine l'élément y.

Remarque. On remplacera souvent cette version générale d'opérations par une version plus simple dépendant de la structure de données considerée.

Quelques structures de données déjà connues

-) Pile (LIFO)
[*stack*]

-) File (FIFO)
[*queue*]

-) Liste chaînée
[*linked list*]

-) Liste doublement chaînée
[*doubly linked list*]

-) Arbre binaire de recherche
(Arbre binaire ordonné)
[*binary search tree*]

-) Arbre binaire de recherche équilibré
[*balanced binary search tree*]

2. Une application

De nombreuses applications font appel à des ensembles dynamiques qui ne supportent que les opérations RECHERCHER, INSERER et SUPPRIMER.

Le compilateur d'un langage de programmation doit tenir à jour l'ensemble de tout les identificateurs (noms de variable, etc) du programme à compiler. Les opérations nécessaires sont RECHERCHER et INSERER, peut-être aussi SUPPRIMER. C'est-à-dire qu'il faut une implémentation d'un dictionnaire. Pour le problème de compilateur on utilise toujours une table de hachage.

Il n'est pas facile d'analyser le temps des opérations d'une table de hachage. Quoi qu'il en soit, en pratique la table de hachage est une structure de données très efficace.

Problème:

- donnée: l'univers $U = \{0, 1, 2, \ldots, |U| - 1\}$
- chaque clé d'ensemble dynamique appartient à U
- trouvez une implementation d'un dictionnaire sur U

Remarque. On supposera que deux éléments ne peuvent pas partager la même clé.

3. Tables à adressage direct

Question: Pourquoi ne pas prendre un tableau?

Si la taille de l'univers U n'est pas trop grande, on peut utiliser un tableau $T[0..|U|-1]$, appelé table à adressage direct dans lequel chaque position (alvéole) correspond à une clé dans l'univers.

Bien sûr on peut implémenter les opérations RECHERCHER, INSERER et SUPPRIMER facilement et le temps d'exécution d'une opération est $O(1)$.

$RECHERCHER{-}ADRESSAGE{-}DIRECT(T, k)$

retourner T[k]

$INSERER - ADRESSAGE - DIRECT(T, k)$

$T[cle[x]] \leftarrow x$

$SUPPRIMER{-}ADRESSAGE{-}DIRECT(T, k)$

$T[cle[x]] \leftarrow NIL$

Désavantage:

Dans une application typique (comme notre problème de compilateur) la taille de l'ensemble dynamique K est très petite comparée à U: $|K| << |U|$.

Exemple:

On suppose la syntaxe de langage de programmation est très stricte:
un identificateur doit être un mot d'au plus de 8 lettres.

$\Rightarrow \quad |U| = \sum_{i=1}^{8} 26^i = \frac{26^7 - 1}{25} - 1 = 321272406$

Mais un programme ou un logiciel a rarement plus de 10000 identificateurs.

$\Rightarrow \quad |K| \leq 10000$

4. Tables de hachage

On suppose $|K| << |U|$.

Une table de hachage requiert moins de place de stockage qu'une table à addressage direct. En particulier, les besoins en espace de stockage peuvent être réduits à $\Theta(|K|)$.

Remarque. On verra que la recherche d'un élément dans la table de hachage s'effectue toujours en $O(1)$ en temps moyen.

Même si une analyse du temps des opérations RECHERCHER, INSERER et SUPPRIMER est difficile et réquiert des hypothèses un peu artificielles mais raissonables:

La conclusion est que le hachage ($hashing$) est une technique extrèmement efficace et pratique.

Idée fondamentale

<u>Fonction de hachage</u> $h : U \to \{0, 1, \ldots, m-1\}$

<u>Table de hachage</u> $T[0..m-1]$
($m << |U|$, $m = \Theta(|K|)$)

Un élement de clé k est <u>haché</u> dans l'alvéole $h(k)$. $h(k)$ est la <u>valeur de hachage</u> de la clé k.

Inconvénient du hachage:

On l'appele collision, qui arrive quand deux clés sont hachées dans la même alvéole.

Comme $|U| > m$, il existe obligatoirement au moins deux clés ayant la même valeur de hachage. Donc on ne peut pas éviter completement les collisions.

Nous considérons deux techniques de résolution des collisions.

Remarque. En choisant h (et m) on peut essayer de minimiser le nombre de collisions.

5. Hachage avec chaînage

Tous les éléments hachés dans la même alvéole sont placés dans une liste chaînée.

L'alvéole j contient un pointeur sur la tête de liste de tous les éléments hachés en j; si aucun élément n'est présent, l'alvéole j contient NIL.

Implémentation des opérations

$INSERER-HACHAGE-CHAINEE(T,x)$
insère l'élément x en tête de la liste $T[h(cle[x])]$

$RECHERCHER-HACHAGE-CHAINEE(T,k)$
recherche un élément de clé k dans la liste $T[h(k)]$

$SUPPRIMER-HACHAGE-CHAINEE(T,x)$
supprime l'élément x de la liste $T[h(cle[x])]$

Remarque. Pour les opérations $INSERER$ et $SUPPRIMER$ l'élément x et donné par un pointeur.

Analyse du temps dans le pire des cas

On ne compte pas le temps de calcul de $h(k)$.

1) $INSERER$: $O(1)$
   parce que l'élément x est inséré en tête de la liste

2) $RECHERCHER$: $O(|T[h(k)]|)$
   parce que dans le pire des cas il faut rechercher dans toute la liste

3) $SUPPRIMER$: $O(1)$
   si les listes sont doublement chaînées et x est donné par un pointeur

Si tous les éléments sont hachés à la même alvéole alors le temps au pire des cas d'exécution d'une opération $RECHERCHER$ est $\Theta(n)$ (plus le temps de calcul de la fonction de hachage).

Certainement ça n'explique pas l'efficacite de hachage.

Analyse du temps dans le cas moyen

Pour expliquer l'efficacité de hachage avec chaînage il faut analyser le temps que prend en moyenne l'opération $RECHERCHER$, c'est-à-dire il faut analyser le longeur moyenne des listes chainées de table de hachage $T[0..m-1]$.

Soit n le nombre d'éléments qui sont conservés dans T. On définit pour T le

<u>Facteur de remplissage</u> $\alpha = \frac{n}{m}$,

$\Rightarrow \quad \alpha$ est égal àu nombre moyen stocké dans une liste chaînée de T.

La performance en moyenne du hachage dépend de la manière dont la fonction h répartit en moyenne l'ensemble des clés à stocker parmi les m alvéoles.

Hypothèse: Hachage uniforme simple

Chaque élément a la même chance d'être haché dans l'une quelconque des alvéoles, indépendamment de l'endroit où les autres éléments sont tombés.

On suppose que la valeur de hachage $h(k)$ peut être calculée en $O(1)$.

Théorème. *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une opération* $RECHERCHE$ *prend en moyenne un temps en* $\Theta(1+\alpha)$*, sous l'hypothèse d'un hachage uniforme simple.*

Corollaire. *Si* $n = O(m)$ *et les listes sont doublement chaînées, alors toutes les opérations de dictionnaire peuvent être supportées par la table de hachage avec chaînage dans un temps en* $O(1)$ *en moyenne, sous l'hypothèse d'un hachage uniforme simple.*

6. Fonctions de hachage

Une bonne fonction de hachage vérifie (approximativement) l'hypothèse de hachage uniforme simple: chaque clé a autant de chances d'être hachée dans l'une quelconque des m alvéoles.

En pratique, on peut faire appel à des techniques heuristiques pour créer une fonction de hachage qui a de bonnes chances d'être efficace.

Exemple: Considérons la table des identificateurs d'un compilateur. Il arrive souvent que des symboles proches, comme

`pt1`, `pt2`, `pt3`, ... , `pt8`,

soient employés dans le même programme.
Une bonne fonction de hachage minimisera les chances pour ces variantes de se retrouver dans la même alvéole.

La plupart des fonctions de hachage supposent que $U \subseteq \mathbb{N}$. Donc, si les clés ne sont pas des entiers naturels, on doit trouver un moyen de les interpréter comme des entiers naturels.

Exemple:

L'identificateur `pt` peut être interprété comme la paire d'entiers décimaux
$(112, 116)$, puisque $\mathtt{p} = 112$ et $\mathtt{t} = 116$ dans l'ensemble des caractères ASCII.

En l'exprimant en base 128, `pt` devient $(112 \cdot 128) + 116 = 14452$.

6.1. La méthode de la division

$$h(k) = k \mod m$$

C'est-à-dire, $h(k)$ est le reste de la division de k par m. Comme il ne demande qu'une seul opération de division, le hachage par division est très rapide.

On évite en général certaines valeurs de m:

-) Puissance de 2
parce que si $m = 2^t$, $h(k)$ sera constituée des t bits d'ordre inférieur à k

-) Puissance de 10
si les clés utilisées par l'application sont des nombres décimaux

Recommandation:

Les nombres premiers assez éloignés des puissances exactes de 2 sont de bonnes valeurs pour m.

Exemple:

On veut stocker environ $n = 2000$ chaînes de caractères, avec des caractères sur 8 bits.

Longueur de liste en moyenne 3 n'est pas catastrophique.

$\Rightarrow$ Prenons $m = 701$ parce qu'il est premier et proche de 2000/3, sans être proche d'une puissance de 2.

$\Rightarrow$ fonction de hachage: $h(k) = k \mod 701$

6.2. La méthode de la multiplication

$$h(k) = \lfloor m \cdot (k \cdot A \mod 1) \rfloor, \quad 0 < A < 1$$

où "$kA \mod 1$" répresente la partie fractionnaire de kA, c'est-à-dire $kA - \lfloor kA \rfloor$.

On choisit en général $m = 2^p$ ce qui permet d'implémenter facilement la fonction h sur la plupart des ordinateurs de la manière suivante: supposons que la taille d'un mot-machine soit w bits et que k tienne dans un seul mot.

KNUTH a étudié le choix de A et suggère que

$$A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots$$

a de bonnes chances de bien marcher.

7. Hachage en adressage ouvert

Tous les éléments sont conservés dans la table de hachage elle-même. Il n'existe ni listes ni éléments conservés hors la table.

$\Rightarrow$ facteur de remplissage $\alpha \leq 1$

Avantage: On évite complètement le recours aux pointeurs. La mémoire supplémentaire libérée par la non conservation des pointeurs permet d'augmenter le nombre d'alvéoles de la table pour la même quantité de mémoire.

S'il y a une collision, alors on cherche - sonde - une autre alvéole pour cette clé. La fonction de hachage crée une séquence de sondage.

Fonction de hachage

$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

Séquence de sondage pour la clé k:

$h(k, 0), h(k, 1), \ldots, h(k, m-1)$

Il est raisonable d'exiger que
$h(k, 0), h(k, 1), \ldots, h(k, m-1)$
est une permutation de
$0, 1, \ldots, m-1$
de façon que chaque position de la table de hachage peut être considérée comme alvéole pour une nouvelle clé lors du remplissage de la table.

Implémentation des opérations

$INSERER-HACHAGE-OUVERT(T,k)$

$i \leftarrow 0$
répéter $j \leftarrow h(k,i)$
    si $T[j] = NIL$
        alors $T[j] \leftarrow k$
            retourner j
        sinon $i \leftarrow i+1$
jusqu'à $i = m$
erreur "débordement de la table"

$RECHERCHER-HACHAGE-OUVERT(T,k)$

$i \leftarrow 0$
répéter $j \leftarrow h(k,i)$
    si $T[j] = k$
        alors retourner j
    $i \leftarrow i+1$
jusqu'à $T[j] = NIL$ ou $i = m$
retourner NIL

Remarque. L'opération $SUPPRIMER$ dans une table à adressage ouvert est difficile.

Hypothèse: Hachage uniforme

Chacune des $m!$ permutations possibles de $\{0, 1, 2, \ldots, m-1\}$ a autant de chances de constituer la séquence de sondage de chaque clé.

Remarque. Le véritable hachage uniforme est difficile à implémenter, et en pratique on utilisera des approximations pertinentes comme le **double hachage**:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

ou h_1 et h_2 sont des fonctions de hachage auxiliaires.

Remarque. Autres méthodes de hachage en adressage ouvert: <u>sondage linéaire</u> et <u>sondage quadratique</u>.

Analyse du temps

Théorème. *Etant donnée une table de hachage en adressage ouvert avec un facteur de remplissage* $\alpha = n/m < 1$, *le nombre attendu de sondages lors d'une recherche infructueuse vaut au plus*

$$1/(1-\alpha),$$

si l'on suppose que le hachage est uniforme.

Théorème. *Etant donnée une table de hachage en adressage ouvert avec un facteur de remplissage* $\alpha < 1$, *le nombre attendu de sondages lors d'une recherche fructueuse vaut au plus*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

si l'on suppose que le hachage est uniforme, et que chaque clé de la table a autant de chance d'être recherchée que les autres.