L'organisation aléatoire (suite)

La résolution des collisions

- Par extension des bacs.

- Par adressage ouvert :

  – la méthode linéaire

$$
\begin{array}{l}
h(K) \\
h(K)+1 \\
h(K)+2 \\
\vdots \\
N-1 \\
0 \\
1 \\
\vdots \\
h(K)-1
\end{array}
$$

- le double hashing

$$
\begin{aligned}
& h_1(K) \\
& (h_1(K) + h_2(K)) \bmod N \\
& (h_1(K) + 2 \times h_2(K)) \bmod N \\
& \vdots \\
& (h_1(K) + (N-1) \times h_2(K)) \bmod N
\end{aligned}
$$

  * $h_2(K) \in [1, N-1]$

  * $h_2(K)$ relativement premier par rapport à N, ce qui garantit que la séquence de recherche est complète.

  * Exemples :

$$h_2(K) = 1 + (K \bmod (N-1))$$

$$h_2(K) = 1 + (K \bmod (N-2))$$

Exemple

- $N = 7$
- $h_1(K) = K \bmod 7$
- $h_2(K) = 1 + (K \bmod 5)$
- séquence de recherche pour $K = 23$:
  2, 6, 3, 0, 4, 1, 5

Problème de la suppression

- $h_1(K) = K \bmod 7$
- $h_2(K) = 1 + (K \bmod 5)$
- Après l'insertion des enregistrements de clés 23, 58, 93 et 128 :

bac	enregistrement	
	clé	donnée
0	128	*données_128*
1		
2	23	*données_23*
3	93	*données_93*
4		
5		
6	58	*données_58*

Après suppression de l'enregistrement de clé 58 :

bac	enregistrement	
	clé	donnée
0	128	*données_128*
1		
2	23	*données_23*
3	93	*données_93*
4		
5		
6		

Que se passe-t-il lors d'une recherche d'un enregistrement de clé 128?

Solutions

- Distinguer entre les bacs non pleins et les bacs qui n'ont jamais été pleins.

  – Inconvénient : rend les recherches très longues

- Avec un incrément fixe dans la séquence de recherche, on peut appliquer l'algorithme suivant : ...

1. Effacer l'enregistrement

2. Parcourir la séquence de recherche jusqu'à

   - soit trouver un bac non plein,
   - soit trouver un enregistrement E pouvant remplacer l'enregistrement effacé.

3. Si on a trouvé un bac non plein, on arrête.

4. Sinon

   - on copie E à la place de l'enregistrement effacé
   - on applique l'algorithme d'effacement à E.

Un enregistrement E peut remplacer un enregistrement E_0 qui doit être effacé si la position occupée par E_0 précède celle occupée par E dans la séquence de recherche de E.

Exemple

- $h(k) = k \bmod 7$

- résolution des collisions par adressage ouvert linéaire

bac	enregistrement	
	clé	donnée
0	74	*données_74*
1		
2		
3	17	*données_17*
4	11	*données_11*
5	31	*données_31*
6	13	*données_13*

- Effacement de l'enregistrement de clé 11 :
  - $E_0 = 4$
  - $E = 5$:
    - $h(31) = 3$
    - séquence de recherche : $3, 4, 5, 6, 0, 1, 2$.
    - E_0 précède E dans cette séquence.

  $\Rightarrow$ Déplacement de l'enregistrement de clé 31 vers le bac 4 :

bac	enregistrement	
	clé	donnée
0	74	*données_74*
1		
2		
3	17	*données_17*
4	31	*données_31*
5		
6	13	*données_13*

- Application de l'algorithme avec $E_0 = 5$:
  - $E = 6$?
    - $h(13) = 6$
    - séquence de recherche : $6, 0, 1, 2, 3, 4, 5$.
    - E_0 ne précède pas 6 dans cette séquence.
  - $E = 0$
    - $h(74) = 4$
    - séquence de recherche : $4, 5, 6, 0, 1, 2, 3$.
    - E_0 précède E dans cette séquence.

  $\Rightarrow$ Déplacement de l'enregistrement de clé 75 vers le bac 5 :

bac	enregistrement	
	clé	donnée
0		
1		
2		
3	17	*données_17*
4	31	*données_31*
5	74	*données_74*
6	13	*données_13*

Implémentation des opérations et performances

Les performances dépendent de :

- la fonction hash,
- la méthode de résolution des collisions,
- l'implémentation des bacs.

Estimation des performances

- Taille optimale des bacs : un bloc.
- Nombre d'accès en lecture ou écriture si il n'y a pas de collision : 1 (2 ou 3 si on utilise un index des bacs)
- En présence de collisions, ce nombre augmente. Cette augmentation est la moins rapide si l'on traite les collisions en utilisant des bacs extensibles.
- Pour obtenir de bonnes performances, il faut travailler avec un fichier qui est au plus 80 à 90% plein.
- Dans ce cas, on peut considérer que le nombre d'accès requis pour une opération sur un fichier aléatoire est constant ($O(1)$).

Considérations d'utilisation

- La taille du fichier est fixée *a priori*, mais il est très fréquent d'avoir une bonne estimation du nombre maximum d'enregistrements que le fichier contiendra.

- Si non, il est possible de travailler avec une organisation aléatoire extensible :

  – lorsque le fichier est plein, on double le nombre de bacs et on étend d'un bit la valeur calculée par la fonction hash (e.g., méthode par multiplication).

- Les implémentations des systèmes de fichiers prévoient souvent l'*organisation relative* qui permet simplement l'accès direct à des bacs numérotés. Il faut alors y ajouter une fonction hash et une méthode de traitement des collisions pour obtenir l'organisation aléatoire.

L'organisation indexée

L'organisation indexée : principe

- Contrairement à l'organisation aléatoire où l'on calcule le numéro de bac à partir de sa clé, l'organisation indexée utilise une table des numéros de bacs.

- Cette table peut être complète ; c'est le cas de l'*index dense* (*dense index*).

- On peut aussi, si les enregistrements du fichier sont maintenus triés, se contenter de conserver dans l'index la première clé de chaque bac ; c'est le cas de l'*index clairsemé* (*sparse index*).

Exemple : index dense

clé	bac
840558	5
841518	6
841939	0
842516	3
850056	4
850640	1
851286	2

Exemple : index clairsemé.

Le fichier est trié et l'on suppose qu'il est organisé en 4 bacs de 2 enregistrements :

bac	enregistrement		
0	840558 841518	Suarez Perez Mennicken	Yolanda Marc
1	841939 ***inutilisé***	Leonard	Anne
2	850056 842516	Petitjean Bizimana	Pierre Javan
3	850640 851286	Demoulin Irving	Pierre Carl

L'index clairsemé est alors :

clé	bac
840558	0
841939	1
842516	2
850640	3

L'implémentation de l'organisation indexée : l'index dense

- Les bacs sont de 1 bloc ou de quelques blocs liés par chaînage.

- Le problème de trouver le premier bloc de chaque bac n'existe plus. En effet, dans l'index on peut mémoriser l'adresse disque du bac contenant l'enregistrement plutôt que le numéro de celui-ci.

- L'index est lui même un fichier d'enregistrements que l'on peut organiser comme

  - un fichier aléatoire

  - un fichier géré à l'aide d'un index clairsemé

  - Un fichier séquentiel contigu sur lequel on utilise l'algorithme de recherche binaire pour trouver un enregistrement de clé donnée.

L'algorithme de recherche binaire

Cet algorithme permet de faire une recherche dans un tableau adressable directement d'éléments triés.

- De façon répétée, l'ensemble d'éléments est divisé en 2 ;
- et le recherche se poursuit avec la moitié de l'ensemble contenant l'élément.
- L'algorithme s'arrête lorsque
  - l'élément recherché est trouvé, ou
  - l'ensemble de recherche devient vide.

Recherche binaire dans un tableau

```
var a: array[1..n] of integer;
      x, j: integer;
      found: boolean;
      i, low, mid, high: integer;
begin
found := false;
low := 1; high := n;
while ¬found ∧ (low ≤ high) do
   begin
   mid := (low + high) ÷ 2;
   if a[mid] = x then begin j := mid ;
                            found := true
                     end
                else if a[mid] < x
                        then low := mid + 1
                        else high := mid − 1
   end;
end
```

Nombre max d'itérations pour un ensemble de n éléments ?

Soit I_n ce nombre

On a :

$$I_1 = 1 \tag{1}$$
$$I_n \leq 1 + I_{n \div 2} \tag{2}$$

Démontrons que

$$I_n \leq \lceil \log_2(n) \rceil + 1. \tag{3}$$

Procédons par induction complète sur la valeur de n.

Nous devons démontrer que

1. (3) est vraie pour $n = 1$ et

2. si l'équation

$$I_k \leq \lceil \log_2(k) \rceil + 1 \tag{4}$$

est vraie pour tout $k < n$, elle l'est aussi pour $k = n$.

1. La validité de (3) pour $n = 1$ est une conséquence directe de (1).

2. Vu (2), pour démontrer (3), il suffit de démontrer que :

$$1 + I_{n \div 2} \leq \lceil \log_2(n) \rceil + 1 \tag{5}$$

En effet, par transitivité (3) se déduit directement de (2) et (5).

L'équation (5) peut encore s'écrire :

$$I_{n \div 2} \leq \lceil \log_2(n) \rceil. \tag{6}$$

Or, pour $k = (n \div 2) < n$, (4) s'écrit :

$$I_{n \div 2} \leq \lceil \log_2(n \div 2) \rceil + 1. \tag{7}$$

Et (6) se déduit par transitivité de (7) et de

$$\lceil \log_2(n \div 2) \rceil + 1 \leq \lceil \log_2(n) \rceil \tag{8}$$

qui est aisément vérifiable.

Conclusion : l'algorithme de recherche binaire a une complexité asymptotique $O(\log(n))$

Index dense : performances et considérations d'utilisation

- Pour effectuer une opération sur un fichier géré à l'aide d'un index dense, il faut effectuer l'opération sur l'index et ensuite accéder au (et éventuellement modifier) le bac correspondant.

- Du point de vue des performances, les accès à des blocs sont ceux nécessaires pour exécuter l'opération sur l'index plus l'accès au bac (1 ou 2 opérations suivant que l'on fait une lecture ou une modification).

- Un index dense offre un certains nombre d'avantages.
  - Il permet de ne jamais réorganiser le fichier et déplacer ses enregistrements, ce qui est important s'il existe des pointeurs vers des enregistrements du fichier (enregistrements *cloués (pinned)*,
  - Il permet parfois un gain de place
  - Les enregistrements de l'index étant petits les performances des opérations sur l'index peuvent être meilleures que les performances sur un fichier comportant le même nombre d'enregistrements de taille plus importante.

Exemple : Supposons que

- nous avons un fichier aléatoire utilisé à un taux de 90%,
- la taille d'un bloc = 2048 octets,
- la taille d'un enregistrement = 1024 octets,
- le fichier comporte 10^6 enregistrements.

Sans index dense, le fichier occupera 555×10^3 blocs

Avec index dense de 10^6 enregistrements de 8 octets, nous aurons :

$500 \times 10^3 + (10^6 \times 8)/(2048 \times 0,9)$

$\simeq 504 \times 10^3$ blocs