Chapitre 3

Les tables de hachage

Nous avons vu plusieurs méthodes de recherche dans un tableau, en particulier la recherche linéaire dans le cas d'un tableau quelconque puis la recherche dichotomique dans le cas d'un tableau trié. Une méthode qui semble plus efficace dans le cas des chaînes de caractères a été introduite à propos de la *table des symboles* lors de la conception des compilateurs et réutilisée souvent depuis. Il s'agit de la méthode des **tables de hachage** (*hash table* en anglais).

Le principe en est simple. Considérons un ensemble A de N éléments, sous-ensemble d'un ensemble B de cardinal important (ce qui est le cas de celui des chaînes de caractères de longueur maximale max). On consière un tableau d'éléments de B de dimension un peu plus grande que ce qui est vraiment nécessaire, par exemple un tableau de dimension $2N$ et une **fonction de hachage** f qui, à tout élément de B, associe un entier (l'index dans le tableau) compris entre 0 et $2N-1$. Le premier élément a_1 de A est placé à l'index $f(a_1)$. Le deuxième élément a_2 est placé à l'index $f(a_2)$ si celui-ci est différent de $f(a_1)$. Sinon on dit qu'il y a **collision** et on le place un peu plus loin. On appelle **numéro de hachage** l'index retenu.

Il existe plusieurs **méthodes de hachage**, chacune se différentiant par sa fonction de hachahe et surtout par son traitement des collisions.

3.1 Vue d'ensemble

3.1.1 Un problème simple

Nous expliquerons les différentes méthodes de hachage à l'aide d'un problème simple : nous voulons une table de hachage chargée de contenir quelques entiers, disons de l'ordre de cinq.

Notre table de hachage sera un tableau de dix entiers :
`int tab[10];`

La fonction de hachage sera très simple. On prend le reste modulo dix :
`f(n) = n%10;`
ce qui donne un index compris entre 0 et 9.

Essayons d'insérer les éléments 89, 18, 109, 16 et 30 dans cette table. Au départ la table est vide :

0	1	2	3	4	5	6	7	8	9

Après insertion de 89, on obtient :

0	1	2	3	4	5	6	7	8	9
									89

Après insertion de 18, on obtient :

0	1	2	3	4	5	6	7	8	9
								18	89

Lors de l'essai d'insertion de 109 on obtient une collision. Nous allons voir dans les sections suivantes comment traiter ces collisions.

3.1.2 Modèle abstrait

Un type de tables de hachage sera représenté par une classe en langage orienté objet, par exemple en langage C++. Une table de hachage sera un objet, instance d'une telle classe.

Les éléments à placer dans la table de hachage seront des éléments d'un type (des objets d'une classe en langage orienté objet), disons `Etype` (pour type des éléments).

Les attributs seront privés. Il s'agira souvent d'un tableau de `Etype` et d'une dimension maximum mais nous verrons qu'il peut s'agir de structures de données plus complexes.

Les méthodes publiques seront, outre les constructeurs :

- une méthode d'insertion d'un élément dans la table :
  `int insert(Etype);`
  l'entier renvoyé étant le numéro de hachage ;
- une méthode pour retirer un élément dans la table s'il s'y trouve :
  `int remove(Etype);`
- une méthode pour renvoyer l'item passé en argument :
  `Etype find(Etype);`

en fait on passe souvent en argument une **clé** et on renvoie l'élément en entier ;
- une méthode pour savoir si l'élément passé en argument se trouve dans la table de hachage :
  `int isIn(Etype);`
  renvoyant un booléen ;
- une méthode pour savoir si la table est vide :
  `int isEmpty();`
  renvoyant un booléen ;
- une méthode pour savoir si la table est pleine :
  `int isFull();`
  renvoyant un booléen.

3.2 Table de hachage à sondage linéaire

3.2.1 Principe

Dans le cas du **sondage linéaire** (*linear probing* en anglais), en cas de collision on cherche la première place libre après l'index. On revient à zéro lorsqu'on est arrivé en bout de table.

Reprenons notre exemple là où nous nous en étions arrêté, c'est-à-dire à l'insertion de 109. On ne peut pas l'insérer à l'index 9 déjà occupé. L'index suivant, 0, est libre. On peut donc l'insérer à cet emplacement :

0	1	2	3	4	5	6	7	8	9
109								18	89

L'insertion de 16 se fait sans difficulté :

0	1	2	3	4	5	6	7	8	9
109						16		18	89

L'insertion de 30 donne lieu à une collision. On peut placer l'élément tout de suite après :

0	1	2	3	4	5	6	7	8	9
109	30					16		18	89

3.2.2 Implémentation dans le cas d'entiers naturels

Écrivons un programme C++ implémentant une telle table de hachage, y compris un test, dans le cas d'une table de hachage d'entiers naturels.

```
/* linear.cpp */

/* Table de hachage d'entiers naturels avec sondage lineaire */

#include <iostream.h>

class hash
  {
    // attributs
```

```
    int max;
    int *tab;
    int n;         // nombre d'elements dans la table

    // methodes

    int suiv(int);
  public:
    hash();
    hash(int);
    int insert(int);
    int remove(int);
    int find(int);
    int isIn(int);
    int isEmpty();
    int isFull();
};

hash::hash()
{
  max = 0;
  tab = new int[0];
  n = 0;
}

hash::hash(int m)
{
  int i;
  max = 2*m;
  tab = new int[max];
  n = 0;
  for (i = 0; i < max; i++)
    tab[i] = -1;
}

int hash::isEmpty()
{
  if (n == 0) return 1;
  else return 0;
}

int hash::isFull()
{
  if (n >= max-1) return 1;
  else return 0;
}

int hash::suiv(int h)
{
  if (h < max-1) return h+1;
  else return 0;
}

int hash::insert(int a)
{
  int h, trouve;
  if (isFull()) return -1;

  h = a % max; // fonction de hachage
  trouve = 0;

  while(!trouve)
```

```
    {
      if(tab[h] == -1)
{
  tab[h] = a;
  n++;
  trouve = 1;
  return h;
}
      h = suiv(h);
    }
}

int hash::remove(int a)
{
  int h0, h;
  int trouve;

  h0 = a % max; // fonction de hachage
  trouve = 0;
  h = h0;

  do
    {
      if(tab[h] == a)
{
  tab[h] = -1;
  n--;
  trouve = 1;
  return h;
}
      h = suiv(h);
    }
  while(!trouve && h != h0); // il faut peut-etre parcourir tout le tableau
  return -1;   // non trouve
}

int hash::find(int a)
{
  int h0, h;
  int trouve;

  h0 = a % max; // fonction de hachage
  trouve = 0;
  h = h0;

  do
    {
      if(tab[h] == a)
{
  trouve = 1;
  return a;
}
      h = suiv(h);
    }
  while(!trouve && h!=h0);
  return -1;    // non trouve
}

int hash::isIn(int a)
{
  int h0, h;
  int trouve;
```

```
  h0 = a % max; // fonction de hachage
  trouve = 0;
  h = h0;

  do
    {
      if(tab[h] == a)
{
  trouve = 1;
  return 1;
}
      h = suiv(h);
    }
  while(!trouve && h != h0);
  return 0;
}

// test

void main(void)
{
  int m, a, c;

  cout << "dimension : ";
  cin >> m;
  hash H(m);

do
  {
  cout << "1 : inserer\n";
  cout << "2 : retirer\n";
  cout << "3 : verifier\n";
  cout << "4 : quitter\n";
  cout << "votre choix : ";
  cin >> c;

  if (c == 1)
    {
      cout << "element a inserer : ";
      cin >> a;
      H.insert(a);
    }

  if (c == 2)
    {
      cout << "element a retirer : ";
      cin >> a;
      H.remove(a);
    }

  if (c == 3)
    {
      cout << "element a verifier : ";
      cin >> a;
      if (H.isIn(a)) cout << "element present\n";
      else cout << "element absent\n";
    }
  }
 while(c != 4);
}
```

3.2.3 Fonction de hachage dans le cas des chaînes de caractères

Il est plus simple de voir le principe des tables de hachage dans le cas des entiers mais le principe s'applique à tous les types d'entités. Une façon universelle de passer de l'un à l'autre est de commencer par coder les entités par des entiers naturels.

En principe c'est simple. Considérons le cas des chaînes de caractères. Une chaîne de caractères est un mot :

$$w = a_1 a_2 \dots a_n$$

sur un alphabet A donné. Soit a le nombre d'éléments de cet alphabet et ϕ une bijection quelconque de A dans [0,a-1]. On peut coder le mot de la façon suivante :

$$cod(w) = \sum_{i=1}^{n} \phi(a_i).a^{i-1}.$$

Le problème est qu'on arrive rapidement à un grand entier qui dépasse la capacité des entiers de l'ordinateur. On pourrait utiliser des entiers illimités en créant une structure de données pour cela. Mais c'est en général inutile puisqu'on va lui appliquer une fonction de hachage.

Ceci nous conduit à une fonction de hachage du type suivant, en supposant que l'on utilise un code ASCII à 256 caractères et en utilisant la méthode de Hörner :

```
// Fonction de hachage pour les chaines de caracteres

unsigned int hash(char * mot, int size)
{
  unsigned int hashVal = 0;
  int i;

  for (i = 0; mot[i] != '\0'; i++)
    hashVal = (HashVal*256 + mot[i]) % size;

  return hashVal;
}
```

Un inconvénient avec cette fonction de hachage est que l'opération modulo est très onéreuse en temps machine. Il est préférable d'utiliser le modulo dû au dépassement de capacité dans la boucle et de ne réaliser l'opération modulo qu'à la fin. Il vaut mieux également utiliser un décalage au lieu d'une multiplication. De plus un décalage de 5 (c'est-à-dire une multiplication par 32) au lieu de 8 permet de mieux tenir compte des premiers caractères. On peut également utiliser la disjonction exclusive au lieu de l'addition. L'ajout de la taille à chaque étape de la boucle permet de tenir compte des premiers caractères. Ceci nous conduit à la fonction de hachage suivante dont l'expérience a montré qu'elle était satisfaisante :

```
// Bonne fonction de hachage pour les chaines de caracteres

unsigned int hash(char * mot, int size)
{
  unsigned int hashVal = 0;
```

```
  int i;

  for (i = 0; mot[i] != '\0'; i++)
    hashVal = (hashVal << 5) ^ mot[i] ^ size;

  return hashVal % size;
}
```

Exercice 1.- Adapter l'exemple sur les entiers naturels pour les chaînes de caractères.

Corrigé des exercices

Exercice 1.- On a, par exemple :

```
/* mot.cpp */

/* Table de hachage (de moins de 200 items)
   de chaines de caracteres d'au plus 99 caracteres
   avec sondage lineaire */

#include <iostream.h>
#include <string.h>
#include <stdio.h>

typedef char mot[100];

class hash
  {
    // attributs
    int max;
    mot tab[200];
    int n;         // nombre d'elements dans la table

    // methodes

    int suiv(int);
    int fhash(mot);
  public:
    hash();
    hash(int);
    int insert(mot);
    int remove(mot);
    int isIn(mot);
    int isEmpty();
    int isFull();
};

int hash::suiv(int h)
{
  if (h < max-1) return h+1;
  else return 0;
}

int hash::fhash(mot s)
{
  unsigned int hashVal = 0;
  int i;

  for (i = 0; s[i] != '\0'; i++)
    hashVal = (hashVal << 5) ^ s[i] ^ max;

  return (int)(hashVal % max);
}

hash::hash()
{
  max = 0;
  n = 0;
}
```

```
hash::hash(int m)
{
  int i;
  max = 2*m;
  n = 0;
  for (i = 0; i < max; i++)
    strcpy(tab[i], ""); // Initialisation au mot vide
}

int hash::isEmpty()
{
  if (n == 0) return 1;
  else return 0;
}

int hash::isFull()
{
  if (n >= max-1) return 1;
  else return 0;
}

int hash::insert(mot s)
{
  int h;
  int trouve;
  if (isFull()) return -1;

  h = fhash(s); // fonction de hachage
  trouve = 0;

  while(!trouve)
    {
      if(!strcmp(tab[h], "")) // Nous sommes surs d'avoir un tel emplacement
{
  strcpy(tab[h], s);
  n++;
  trouve = 1;
  return h;
}
      else h = suiv(h);
    }
}

int hash::remove(mot s)
{
  int h0, h;
  int trouve;

  h0 = fhash(s); // fonction de hachage
  trouve = 0;
  h = h0;

  do
    {
      if(!strcmp(tab[h], s))
{
  strcpy(tab[h], "");
  n--;
  trouve = 1;
  return h;
}
      else h = suiv(h);
```

```
    }
  while (!trouve && h!=h0);
  return -1; // non trouve
}

int hash::isIn(mot s)
{
  int h0, h;
  int trouve;

  h0 = fhash(s); // fonction de hachage
  trouve = 0;
  h = h0;

  do
    {
      if(!strcmp(tab[h], s))
{
  trouve = 1;
  return 1;
}
      else h = suiv(h);
    }
  while(!trouve && h!=h0);
  return 0;    // non trouve
}


// test

void main(void)
{
  int m, c;
  mot s;

  cout << "dimension : ";
  cin >> m;
  hash H(m);

do
  {
  cout << "1 : inserer\n";
  cout << "2 : retirer\n";
  cout << "3 : verifier\n";
  cout << "4 : quitter\n";
  cout << "votre choix : ";
  cin >> c;

  if (c == 1)
    {
      getchar();
      cout << "element a inserer : ";
      gets(s);
      H.insert(s);
    }

  if (c == 2)
    {
      getchar();
      cout << "element a retirer : ";
      gets(s);
      H.remove(s);
```

```
    }

  if (c == 3)
    {
      getchar();
      cout << "element a verifier : ";
      gets(s);
      if (H.isIn(s)) cout << "element present\n";
      else cout << "element absent\n";
    }
  }
 while(c != 4);
}
```