TP9 – Tables de hachages

Projet de programmation M1

01 Décembre 2015

Les tables de hachage sont une structure de données permettant de représenter un ensemble de données d'un certain type, appelons-le `t` (voir exercice 5 pour une autre application très importante des tables de hachages). On veut pouvoir effectuer efficacement les opérations suivantes :

— ajouter un élément
— enlever un élément
— rechercher un élément

On pourrait représenter un ensemble d'objet de type `t` par un tableau d'éléments de type `t`. Cependant les opérations ajouter/enlever des éléments deviendraient difficiles à implémenter à cause de leur caractère dynamique.

On pourrait aussi représenter aussi un ensemble avec un ABR. On a implémenté dans le TP7 les trois opérations précédentes sur les arbres binaires de recherche. Donc on a gagné, fin du TP, non ? Non. Le problème avec les ABR c'est que cela ne fonctionne que si le type `t` qu'on considère est ordonné. On veut que les tables de hachages puissent s'adapter à des types quelconques.

L'idée principale des tables de hachage est de *partitionner* l'ensemble des éléments de type `t`) en n ensembles disjoints $S_0, \ldots, S_{n-1}$. Pour cela, on introduit une *fonction de hachage* h qui associe à élément de type `t` le numéro i de la partition S_i dans laquelle il se trouve. On veut :

— h doit être facile à calculer
— h doit répartir assez uniformément les éléments de type `t`.

Comment utiliser cela pour accélérer les opérations des tables de hachages ? Il y a plusieurs méthodes, nous présentons ici la principale, les *tables de hachage chaînées*. On suppose qu'on a notre fonction de hachage h : {éléments de type `t`} $\to [n]$. On définit n listes chaînées et on les stocke dans un tableau t de taille n. Maintenant, on va maintenir l'invariant suivant : un élément e sera toujours stocké dans la liste $t[h(e)]$. Donc pour ajouter/rechercher/enlever un élément e, il faut simplement calculer $h(e)$ et ajouter/rechercher/enlever e dans la liste $t[h(e)]$, qui, on l'espère, sera bien plus petite que la taille totale de la table.

Pour fixer les idées, prenons `t = int`. On suppose n fixé, disons $n = 100$. On choisit $h : \mathbb{Z} \to [n]$ par $h(i) = i \mod n$. h est facile à calculer et h répartit les entiers de façon uniforme. C'est une bonne fonction de hachage. On a donc un tableau t de taille n contenant au départ n listes vides. On insère 121 dans la table : $121 \mod 100 = 21$ donc on insère la valeur 121 à dans la liste `t[21]`. Si on veut chercher 534, on peut se contenter de chercher dans la liste `t[34]` etc. Si la table est bien équilibrée (c'est-à-dire que les listes du tableau ont des tailles à peu près égales), alors on gagne un temps non-négligeable pour chacune des opérations voulues.

Exercice 1. Implémentez les fonctions `ladd(list *h, int v)`/`lsearch(list *h, int v)`/`lrem(list *h, int v)` pour ajouter/rechercher/supprimer un élément d'une liste chaînée dont le premier élément se trouve à l'adresse `h`. On utilisera le type `list` suivant :

```
struct list {
  int val;
  struct list *next;
};
typedef struct list list;
```

Exercice 2. En utilisant les fonctions de l'exercice 1, implémentez les tables de hachages. On utilisera le type :

```
struct hash {
  int size;
```

```
  list **t;
};
typedef struct hash hash;
```

On implémentera les fonctions : `hash hcreate(int size)` qui initialise une table de hachage (réservation de la mémoire pour t, initialisation de la taille), `hadd(hash h, int k)`, `hrem(hash h, int k)` et `hsearch(hash h, int k)` qui ajoute/enlève/recherche une valeur dans `h`.

Exercice 3. Vous êtes un hacker militant. Le site internet d'un méchant monsieur utilise des tables de hachage sur les entiers avec la fonction de hachage $h(i) = i \mod 666$ et vous savez comment insérer des valeurs de votre choix dans sa table de hachage. Comment feriez pour ralentir sensiblement son site et le rendre inutilisable ?

Voir les slides de Roberto ou le Cormen sur comment on peut choisir aléatoirement une bonne fonction de hachage sur les entiers pour éviter ce genre de vulnérabilité (fun fact : cette faille existait dans le langage PHP, elle a été corrigée le 21 Décembre 2012).

Exercice 4. Proposer une fonction de hachage pour la structure :

```
struct point {
  int x;
  int y;
}
```

Et pour les chaînes de caractères ? Est-ce une bonne fonction de hachage ?

Exercice 5. Dans la vraie vie, on utilise les tables de hachage pour associer une valeur de type `u` à une donnée de type `t`. Pour cela, on définit une structure plus riche :

```
struct hash_element {
  t key;
  u value;
};
```

On utilise toujours une fonction de hachage sur les éléments de types `t`. Seulement désormais on insère des éléments de types `hash_element` dans les listes. Réfléchissez (et implémentez en exercice, mais ce sera un peu long) à comment vous pourriez associer une valeur de type `float` à un point du plan (c'est-à-dire `t = point` et `u = float`).