TP8 – Tables de hachages et révisions

Projet de programmation M1

09 Décembre 2014

1 Tables de hachage

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 4 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 une liste. On a implémenté dans le TP6 les trois opérations précédentes sur les listes chaînés. Donc on a gagné, fin du TP, non ? Non. Le problème avec les listes chaînées c'est que la recherche d'un élément est coûteuse. Si l'élément ne se trouve pas dans la liste, on doit quand même parcourir *toute* la liste. On voudrait une structure de données capable de fonctionner avec un ensemble de taille importante.

L'idée principale des tables de hachage est de *partitionner* l'ensemble des données possibles (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 taille.

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. En utilisant le fichier `list.c` (sur ma page ou celle de Roberto), implémentez les tables de hachages pour les entiers (la taille n de la table est un paramètre).

Exercice 2. 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 3. 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 4. 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.

2 Révisions

Exercice 5. Sans l'exécuter, donner la sortie du programme suivant :

```
void f(int x) {
  x = 42;
}
void g(int *x) {
  *x = 26;
}

int main() {
  int x =  3;
  f(x);
  printf("%d", x);
  g(&x);
  printf("%d", x);
  int t[5], i;
  for(i=0;i<5;i++)
    t[i] = 0;
  g(t);
  g(t+1); // wtf :) ?
  for(i=0;i<5;i++)
  printf("%d", t[i]);

  return 0;
}
```

Exercice 6. Écrire une fonction qui étant donné un tableau t d'entiers et un entier n, vérifie que la fonction $i \mapsto t[i]$ est une permutation de $[0, n-1]$.

Exercice 7. [Passer au crible] Écrire une fonction qui étant donné un entier n, affiche tous les nombres premiers plus petits que n.

Exercice 8. Écrire un petit jeu de pierre, feuille, ciseau avec un joueur et un ordinateur. L'ordinateur pourra jouer aléatoirement (ou vous pouvez faire une IA pour ce jeu. Ah bon ? Juré : `http://www.nytimes.com/interactive/science/rock-paper-scissors.html?_r=0`).

Exercice 9. Implémentez une fonction, qui étant donné deux chaînes deux caractères `motif, texte`, vérifie si `motif` apparaît dans `texte`. Quelle est la complexité de votre fonction ?

Exercice 10. Voir les exercices de révisions du TP4.