Cours d'Algorithmique et Complexité

Structures de données (suite)

Catalin Dima

Stockage et manipulation des données à nombre limité de fonctionnalités

- Nombreuses applications n'ont besoin, pour la manipulation de leurs données, que de trois primitives :
  - `insérer(clé)` : insérer une nouvelle donnée identifiable par sa clé (identifiant unique).
  - `rechercher(clé)`.
  - `supprimer(clé)`.
  - Penser aux implémentations des bases de données ! clé = attribut prenant des valeurs uniques dans toute la table !
- Toutefois, la taille des données et la fréquence des opérations fait qu'on doit chercher des structures avec une complexité faible de ces opérations.
- Critère de comparaison : nombre de clés qui doivent être stockées à un instant quelconque.
  - Quand l'espace des clés est petit, ou on arrive assez souvent à stocker un nombre assez important de clés dans l'espace des cles : tables à adressage direct.
  - Lorsqu'on a besoin de stocker beaucoup moins de clés que le nombre total des clés possibles : tables de hachage.

Tables d'adressage direct

- Tableau de dimension proportionnelle au nombre de clés.
- Utiliser les clés en tant qu'indices dans la table d'adressage.
  - Part de l'hypothèse que la clé est numérique, et a comme espace de valeurs un intervalle $[0..Max]$.
- Insertion, suppression, recherche : évidentes !

Tables de hachage

- Le principe est celui du codage de chaque clé k par une valeur numérique : le haché de la clé, $h(k)$.
- Nécessite une fonction de hachage $h : U \to \{0, 1, \ldots, M-1\}$, où U est l'univers des clés et M est la taille du tableau.
- Le but est de coder des clés d'un univers très grand dans des (relativement) petites valeurs d'indices, donc $card(U) \gg M$.
- Inconvenient (de principe) : h ne peut pas être une bijection !
  - Donc on peut avoir des collisions : $h(k) = h(k')$ pour $k \neq k'$.
- L'efficacité de la méthode réside dans les techniques de résolution des collisions.
  - On va en étudier une (chaînage), avant de découvrir des bonnes fonctions de hachage.

Résolution des collisions par *chaînage*

- Pour chaque indice i dans la table de hachage on utilise une liste chaînée pour les données stockées à l'indice i.
- Pour insérer une donnée de clé k :
  - On calcule $h(k)$ et on regarde si l'indice $h(k)$ est occupé dans la table de hachage.
  - Si case "occupée", alors la donnée sera placée à la fin de la liste chaînée d'indice $h(k)$.
  - Si case "libre", alors la donnée sera stockée en tant que tête de la liste chaînée d'indice $h(k)$.
- Pour retrouver une donnée de clé k :
  - On calcule $h(k)$.
  - Dans la liste chaînée retrouvée à l'indice $h(k)$, on cherche séquentiellement la clé k et on la renvoie (si trouvée), ou erreur (sinon).
- Pareil pour supprimer une donné de clé k !

Fonctions de hachage

- Fonctions satisfaisant l'hypothèse de *hachage uniforme*
  - Les chances d'une clé d'être hachée vers un indice sont indépendantes des endroits où sont placées les autres clés.
- Méthode de la division : choisir $\alpha \in [0, 1]$, puis calculer

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

  - Souvent on choisit M = nombre premier.
- Méthode de la multiplication :

$$h(k) = \lfloor M \cdot (k\alpha \mod 1) \rfloor$$

Ici modulo 1 c'est prendre que la partie décimale, c.à.d. $k\alpha - \lfloor k\alpha \rfloor$!

Classe `Hashtable`

- Classe template avec deux paramètres : `public class Hashtable<K,V>`, où :
  - `K` est le type de clés.
  - `V` est le type des valeurs (objets) stockés à l'aide des clés.
- Constructeur `Hashtable()`, mais aussi `Hashtable(int capinit)` et `Hashtable(int capinit, int factremplissage)` (voir théorème !).
- Méthodes importantes :
  - `boolean containsKey(Object key)` et `boolean containsValue(Object value)`.
  - `V get(Object key)` : attention au typage !
  - `V put(K key, V value)` : si la clé existe, la valeur stockée sous cette clé sera remplacée et l'ancienne valeur sera retournée ; sinon renvoie `null`.
  - `int size()`.
  - `boolean isEmpty()`.
  - `Enumeration<K> keys()` : renvoie un objet implémentant `Enumeration` des clés de référencement des objets stockés.
  - `Enumeration<V> elements()` : renvoie un objet implémentant `Enumeration` des valeurs stockés.

Tables de hachage à adressage ouvert

- Chaque “alvéole” est soit inoccupée, soit occupée par un seul élément.
- Fonctionne lorsqu’on est sûr de ne jamais avoir plus d’éléments que la taille du tableau.
- Lorsqu’on veut insérer une clé k :
  - On calcule un premier haché de la clé et on regarde (on sonde) si l’indice calculé est occupé dans la table de hachage.
  - Si l’alvéole d’indice calculé est libre, on met la clé et l’objet déterminé par k dans cette alvéole.
  - Sinon on calcule *un 2e haché*, qui doit être différent.
  - ... et on repète l’essai d’insertion dans l’alvéole d’indice calculé.

Tables de hachage à adressage ouvert (2)

- Il nous faut une fonction de hachage binaire :
  $h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$.
  - Rappel : U est le domaine des clés, m est la taille du tableau.
  - L'algorithme revient à sonder $h(k, 0)$, puis $h(k, 1)$, puis $h(k, 2)$ etc.
- Exemples :
  1. Pour toute $h : U \to \{0, \ldots, m-1\}$ fonction de hachage, on peut utiliser :

$$\overline{h}_{lin}(k, i) = \big(h(k) + i\big) \mod m \tag{1}$$

$$\overline{h}_{sq}(k, i) = \big(h(k) + c_1 i + c_2 i^2\big) \mod m \tag{2}$$

  2. Pour toute paire de fonctions de hachage $h_1, h_2 : U \to \{0, \ldots, m-1\}$, on peut utiliser :

$$\overline{h}_{prod}(k, i) = \big(h_1(k) + i * h_2(k)\big) \mod m$$