INF2010
Structures de données et algorithmes
TP02 : Hachage

Hiver 2019

Objectifs du TP:

- Implémanter des table de dispersement parfaites.

1 Hachage parfait

Les tables de dispersement réalisent un accès aux données en temps constant par le moyen d'une fonction de hachage pour localiser les éléments en mémoire. L'inconvénient de cette approche est que deux objets différents peuvent viser la même position en mémoire, soit une collision.

Le hachage parfait est une technique réalisant une table de dispersement sans collision dont le principe repose sur le fait que, dans certains cas, les données sont connues à avance et qu'elles ne changent plus par la suite — c'est le cas par exemple des mots du dictionnaire utilisé pour la coloration syntaxique d'un éditeur de texte (**for**, **if**, **while**, etc.).

1.1 Hachage parfait à occupation d'espace quadratique

Supposons que nous disposions d'un ensemble de n objets statiques[1]. Il existe alors un hachage permettant de stocker ces n objets dans un espace mémoire de taille $m = n^2$ sans causer une seule collision. La position de chaque objet x est donnée par:

$$\textbf{mod}(\textbf{mod}(ax + b, p), m), \tag{1}$$

où p est un nombre premier tel que $0 < m < p$, $0 < a < p$ et $0 \leq b < p$.

[1]Qui ne changent pas après leur création

Il est à noter que si a et b sont choisis au hasard, il y a seulement 50% de chances qu'une collision survienne. Ainsi, a et b peuvent être générés aléatoirement jusqu'à ce qu'aucune collision ne survienne.

1.1.1 Questions

- Proposez une implémentation correcte de la méthode *findPos* de la classe *QuadratiqueSpacePerfectHashing* qui vous est fournie. La méthode retourne la position de l'élément en mémoire selon la formule de l'équation (1). Assurez-vous que le résultat est bien dans l'intervalle $[0, m)$.
- Proposez une implémentation correcte de la méthode *unsuccessfulMemoryAllocation* de la classe *QuadratiqueSpacePerfectHashing*. La méthode génère aléatoirement a et b et alloue l'espace mémoire requis. Elle tente alors d'insérer les éléments dans le tableau *items* alloué. Si une collision existe, la méthode renvoie true. Si la méthode a réussi à insérer l'ensemble des éléments, alors elle retourne et renvoie vrai.

1.2 Minimiser l'espace requis

Dans cette première approche, le hachage parfait était possible au prix d'une occupation de mémoire m proportionnelle au carré du nombre de données $m = n^2$. Il est possible d'atteindre une occupation d'espace qui soit linéairement proportionnelle à la quantité de données. Dans un premier temps, on utilise un premier tableau de taille n. Chaque objet x vise la position:

$$j = \textbf{mod}(\textbf{mod}(ax + b, p), n), \tag{2}$$

où p est un nombre premier tel que $n < p$, $0 < a < p$ et $0 \leq b < p$. On peut choisir a et b aléatoirement. Pour chaque position j obtenue par l'équation (2), on dispose alors de n_j objets en collision. On utilise un hachage parfait à occupation quadratique pour stocker ces n_j objets dans un espace $m_j = n_j^2$ sans collision. Le concept est illustré à la figure 1.

1.2.1 Questions

- Complétez la méthode *findPos* de la classe *LinearSpacePerfectHashing* qui vous est fournie. La méthode retourne la position de l'élément en mémoire selon la formule de l'équation (2). Assurez-vous que le résultat est bien dans l'intervalle $[0, n)$.

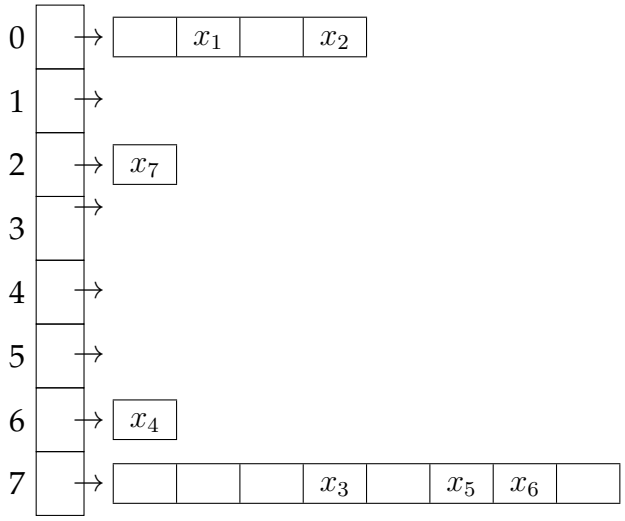

Figure 1: Illustration d'un LinearPerfecthashing contenant 7 éléments.

- Complétez la méthode *contains* de la classe *LinearSpacePerfectHashing* qui vous est fournie. La méthode retourne vraie si l'élément x qui lui est fourni en paramètre est présent dans la structure de données. Elle retourne faux autrement.
- Complétez la méthode *allocateMemory* de la classe *LinearSpacePerfectHashing* qui vous est fournie. En plus d'instancier le tableau de *QuadraticSpacePerfectHashing*, la méthode doit calculer le taille de la mémoire allouée, *memorySize*. Si on se réfère à l'exemple de la figure 1, *memorySize* $= 4 + 1 + 1 + 9 = 15$
- Complétez la méthode *toString* de la classe *LinearSpacePerfectHashing* qui permet d'afficher le contenu de la structure de données sous la forme retourner une chaine de caractères afin d'afficher les éléments de l'ensemble sous la forme suivante:

  - 0 -> x_0^1, x_0^2,...
  - 1 -> x_1^1, x_1^2,...
  - ...
  - $n-1$ -> x_{n-1}^1, x_{n-1}^2,...

1.3 Tests aléatoires

La preuve que l'approche implémentée dans *LinearSpacePerfectHashing* occupe un espace linéairement proportionnel à la quantité de données dépasse le cadre de notre cours. On vous demande de vous en convaincre en effectuant des tests aléatoires.

1.3.1 Questions :

- Implémentez la fonction *randomIntegers* qui retourne un *ArrayList* de taille *length* et comportant des objets de type *Integer*. La liste obtenue ne doit pas inclure de doublons!
- Rapportez sur un graphique les points obtenus en utilisant un tableur ou un logiciel mathématique tel que Matlab ou Octave.
- Expliquez pourquoi on a choisi $p = 46\,337$ pour les classes *LinearSpacePerfectHashing* et *QuadratiqueSpacePerfectHashing*.

2 Instructions pour la remise

Le travail doit être remis via Moodle :

- 19 Février avant 23h55 pour le groupe 01
- 12 Février avant 23h55 pour le groupe 02

Veuillez envoyer dans une archive de type $*$.zip qui portera le nom inf2010_lab2_MatriculeX_ MatriculeY (MatriculeX $<$ MatriculeY) :

- Vos fichiers .java
- Un document .pdf qui contient vos réponses.

Les travaux en retard seront pénalisés de 20 % par jour de retard. Aucun travail ne sera accepté après 4 jours de retard.

2.1 Barème de correction

6 pts: Questions 1.1.1

10 pts: Questions 1.2.1

4 pts: Questions 1.3.1