INF2010 - Structures de données et algorithmes
Automne 2017
Travail Pratique 3
Fonctions de hachage et tables de dispersement

PROBLÈME 1: Hachage parfait (4 pts)

Les tables de dispersement tentent de réaliser l'idée simple que des données peuvent être accédées en temps $O(1)$ si une fonction de hachage est utilisée pour retrouver la position en mémoire des éléments. Le problème principal associé à cette approche est que deux objets différents peuvent produire par hachage la même position en mémoire, ce que l'on appelle une collision. Différentes techniques existent pour gérer les collisions (listes-chaînées, sondage linéaire ou quadratique etc.). Le hachage parfait est une technique permettant l'implémentation de tables de dispersement sans collision qui repose sur le fait que dans certains cas, les données sont connues à avance et elles ne changent pas (par exemple dans le cas d'un éditeur de texte qui utilise la coloration syntaxique d'un nombre fini de mot clés (for, if, while, etc.).

Objectifs :

- Implanter une table de dispersement parfaite (sans collision).
- Minimiser l'espace occupée par une telle table.

Exercice 1 : Hachage parfait à occupation d'espace quadratique (1 points)

Supposons que nous disposions d'un ensemble de n objets statiques (qui ne changent pas après leur création) et qui ont une valeur de `hashCode()` inférieure à un nombre premier p donné. 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 object x est donnée par : $((a \cdot x.\texttt{hashCode()} + b) \bmod p) \bmod m$, où $m < p$, $0 < a < p$ et $0 \leq b <$ p, et où $x.\texttt{hashCode()}$ est supposé renvoyer une valeur inférieur à p !

Sachant qu'il y a moins de 50% de chances qu'une collision survienne si a et b sont choisis au hasard, proposez une implémentation correcte de cette méthode en modifiant la classe `QuadratiqueSpacePerfectHashing` qui vous est fournie, pour insérer des éléments (en modifiant la méthode `AllocateMemory`). `AllocateMemory` prend en paramètre un tableau d'élément à insérer dans la table de hachage, il faut générer aléatoirement des valeurs pour `a` et `b` tant que tout les éléments de ce tableau sont insérés sans qu'aucune collision n'est trouvé, si une collision est détecté, il faut générer de nouvelles valeurs aléatoires de `a` et `b`.

Pour vérifier si un élément existe (en modifiant la méthode `containsValue`), vérifier si une clé existe (en modifiant la méthode `containsKey`), supprimer un élément (en modifiant la méthode `remove`), récupérer la clé d'un élément (en modifiant la méthode `getKey`). Complétez la méthode « `makeEmpty` » qui supprime tout les éléments de la table de hachage. Complétez la méthode « `toString` » qui permet de retourner une chaine de caractères afin d'afficher les éléments de l'ensemble sous la forme suivante :

`(clé_1, val_1), (clé_2, val_2),…, (clé_n, val_n).`

Vous n'avez pas le droit de changer la signature des méthodes. Vous avez le droit de rajouter de nouvelles méthodes au besoin, bien que cela ne soit pas nécessaire. On vous impose $p = 46\,337$.

Exercice 2 : Minimiser l'espace requis (1.5 points)

Dans cette première approche, le hachage parfait était possible au prix d'une occupation de mémoire proportionnelle au carré du nombre de données n. Il est possible d'atteindre une occupation d'espace qui soit linéairement proportionnelle à la quantité de données en utilisant une approche qui réunit les concepts de listes chaînées et de hachage double.

Dans un premier temps, on utilise un premier tableau de taille $m=n$. Chaque objet x vise l'alvéole (case) $j = ((a \cdot x.\texttt{hashCode()} + b) \bmod p) \bmod m$, où $m = n < p$ $0 < a < p$, $0 \leq b <$ p, et où $x.\texttt{hashCode()}$ est supposé renvoyer une valeur inférieur à p !. On peut choisir a et b aléatoirement pour ce premier niveau. Pour chaque alvéole j, on a n_j objets en collusion. On utilise alors un hachage parfait à occupation quadratique pour stocker les n_j objets dans un espace $m_j = n_j^2$ sans collision Proposez une implémentation de cette approche en modifiant la classe `LinearSpacePerfectHashing` qui est fournie, en modifiant les méthodes `AllocateMemory, containsValue, containsKey, getKey, remove et makeEmpty.`

Complétez la méthode « `toString` » qui permet de retourner une chaine de caractères afin d'afficher les éléments de l'ensemble sous la forme suivante :

[clé_1] -> `(clé_i, val_i), (clé_j, val_j),…, (clé_n, val_n).`
[clé_2] ->
…
[clé_X] -> `(clé_x, val_x), (clé_y, val_y).`

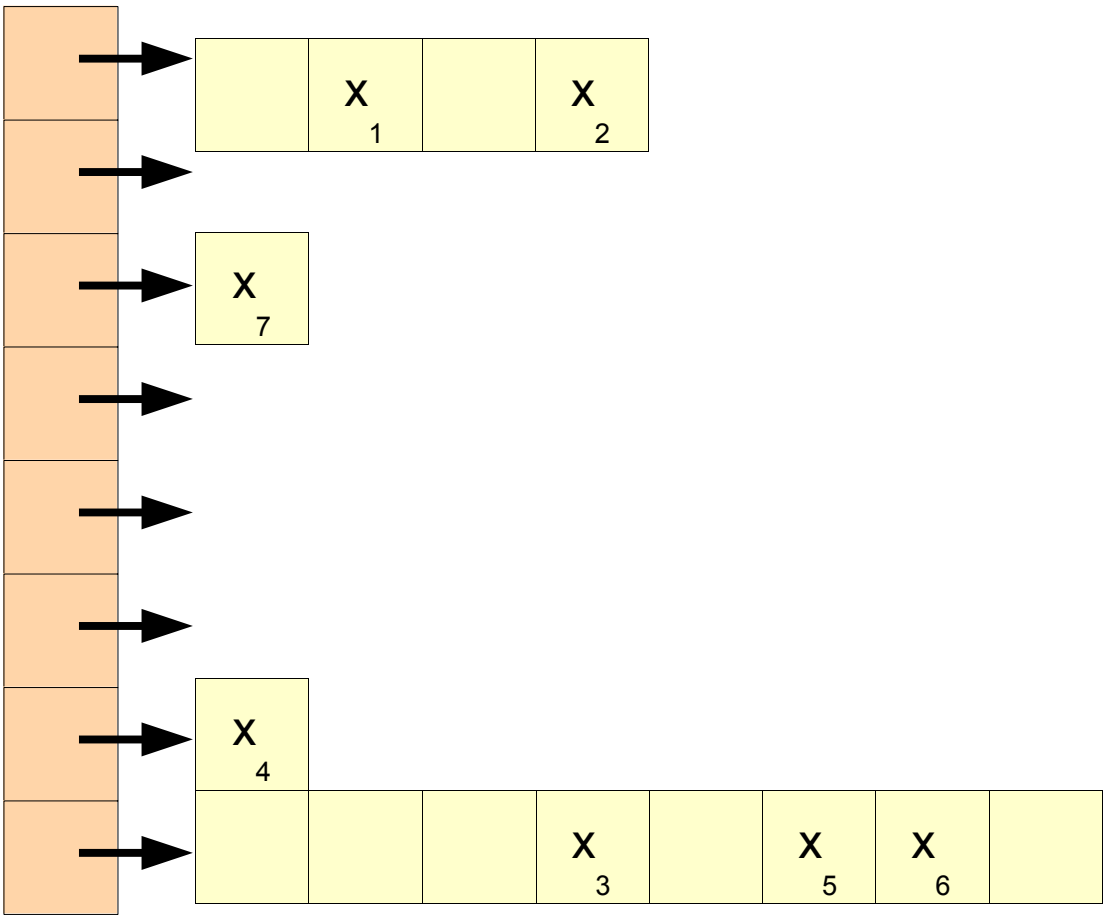

Figure 1 Illustration d'un LinearPerfecthashing contenant 7 éléments.

Question 1 : Linéaire ou pas ? (1.5 points)

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. Implémentez la fonction `randomIntegers(int length)`, permettant de créer un `ArrayList` de taille `length` comportant des `Integer` dont la valeur est majorée par p. 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`. Quelle limite sur la taille des données cela impose-t-il ? Votre code reflète-t-il cette limite ?

Problème 2 : Hachage double (1.5 pts)

Une mappe de dispersement est identique à une table de dispersement à l'exception près qu'on y stocke des paires <clé, valeur> au lieu d'une clé simplement. La fonction de dispersement est calculée sur la clé et non sur la paire. De cette façon, il est possible de stocker des doublons, à condition que les clés qui leurs sont associés diffèrent. En vous basant sur le fichier *QuadraticProbingHashTable.java* tiré de votre manuel de cours, créez une table à double fonction de dispersement *DoubleHashingTable.java*.

Les fonctions à utiliser sont :

$$\text{position} = (H_1(x) + i\, H_2(x)) \,\%\, N$$
$$H_1(x) = x \,\%\, N$$
$$H_2(x) = R - (X \,\%\, R)$$

où R<N est un nombre premier et N représente la taille de la table. On vous demande d'implémenter la fonction H_2 de telle façon que R soit choisi dynamiquement en fonction de N.

On vous demande de créer et implémenter la méthode « `nbreOccurence` » qui calcule le nombre d'occurrence d'une valeur X dans la mappe de dispersement (la méthode prend une valeur X en paramètre et retourne un entier qui représente le nombre d'occurrence de X).

Instructions pour la remise :

Le travail doit être fait par équipe de 2 personnes idéalement et doit être remis via Moodle :

- 23 Octobre avant 23h59 pour le groupe Mardi (B2).
- 29 Octobre avant 23h59 pour le groupe Lundi (B2).
- 30 Octobre avant 23h59 pour le groupe Mardi (B1).
- 05 Novembre avant 23h59 pour le groupe Lundi (B1).

Veuillez envoyer vos fichiers .java seulement dans une archive de type *.zip qui portera le nom inf2010_lab3_MatriculeX_MatriculeY (de sorte que MatriculeX < MatriculeY). Les travaux en retard seront pénalisés de 20 % par jour de retard. Aucun travail ne sera accepté après 4 jours de retard.