UNIVERSITÉ DE MONTRÉAL
Département IRO

Été 2017
Neil Stewart

**IFT 2015 E17**

Devoir 3.

10/10, soit 10% de la note finale.

Les 10 points "Partie pratique" pour le cours E17 seront distribués de la façon suivante : Devoir #1 (1 point), Devoir #2 (3 points), Devoir #3 (6 points).

# 1 Partie Pratique (6 points)

Dans ce devoir, vous devrez implanter une *table de hachage*, mais avec des concepts plus avancés que ceux vus en cours. En particulier, vous devrez implanter la table d'une façon qui évite le réhachage, et pour laquelle le coût de l'agrandissement du tableau est réelement $\Theta(N)$.

Selon Weiss section 5.5, le réhachage coûte $O(N)$. Hors, ceci n'est pas strictement vrai : n'importe quelle fonction de hachage admissible doit obligatoirement utiliser toute l'information des clefs (par exemple chaque lettre dans un String). Si les clefs sont de taille fixe, le réhachage est effectivement $O(N)$, mais si les clefs sont de taille variable, ce sera plutôt $O(\sum_{n=1}^{N} |k_n|)$ où $k_n$ est la suite de symboles de la clef $n$, ce qui sera $O(N)$ si et seulement si la longueur moyenne des clefs est indépendante de $n$. Que faire alors ?

## 1.1 Quelques réalisations

Un premier élément de réponse est que, étant donné une bonne fonction de hachage $h_1$ ayant comme image $\{n|n \in \mathbb{N} \wedge 0 \leq n \leq N_{max} - 1\}$ où $N_{max}$ est la taille maximale d'un tableau, toute fonction $h_2 = h_1 \bmod M$ où $N_{max} \bmod M = 0$ est aussi une bonne fonction de hachage. Donc, si nous utilisons des tableaux de taille $2^k$ et une fonction de hachage donnant un résultat ayant autant de bits que le plus grand entier pouvant servir d'index dans notre langage de programmation (par exemple `Integer.MAX_VALUE` en Java, la valeur maximale d'un `unsigned long long` en C, etc), nous pourrons calculer $h_2 = h_1 \bmod 2^k$, qui sera une bonne fonction de hachage si $h_1$ l'est, et ce pour tous les tableaux de taille $2^k$ possibles. Comme nos nombres sont représentés en base 2, nous n'aurons qu'à masquer les $\lg(N_{max}) - k$ premiers bits de $h_1$ pour obtenir $h_2$. Ceci se fait en temps constant en utilisant un masque $m$ tel que $h_2 = h_1 \bmod 2^k = h_1 \& m$ et $m = 2^k - 1$. L'intérêt de ce changement étant ceci : d'une part, l'opération de «réhachage» ne fait jamais intervenir ni un hachage ni les clef, et au lieu d'avoir à calculer des restes de divisions entières (ce qui est l'opération la plus lente sur des entiers) nous n'avons qu'à effectuer une opération bit-à-bit (ce qui sont les opérations les plus rapides). Ceci implique qu'il ne sera nécessaire de calculer qu'un seul hachage par clef, nous donnant la possibilité d'utiliser des fonctions de hachage de haute qualité.

Un deuxième élément de réponse est que, dans le cadre d'un addressage ouvert avec *linear probing* ou *double hashing*, il n'est pas nécessaire que la taille du tableau soit premier.

En vertu du théorème des restes chinois, à partir d'un nombre quelconque d'équations $X = a_n \bmod m_n$ avec des entiers quelconques $a_n$, si les modules $m_n$ sont copremiers deux-à-deux (c'est-à-dire que le plus grand commun diviseur de $m_i$ et $m_j$ est 1) il existera un $X$ unique dans $0 \leq X < \prod m_n$ qui satisfera toutes les équations.

Dans notre cas, nous voulons que $h(x)+i\cdot p(x) = index \bmod M$ pour tout index, avec $0 \leq i < M$. Hors, nous pouvons également écrire que $h(x)+i\cdot p(x) = h(x) \bmod p(x)$. Nous pouvons alors isoler $i \cdot p(x)$, ce qui donne $i \cdot p(x) = index - h(x) \bmod M$ et $i \cdot p(x) = 0 \bmod p(x)$. Donc, il existe un $i$ tel que $0 \leq ip(x) < Mp(x)$, si $M$ et $p(x)$ sont copremiers.

En pratique, ceci implique que si nous utilisons des tableaux de taille $M = 2^k$, que $p(x)$ est impair, et que $p(x) < 2^k$, nous avons une garantie que nous visiterons toutes les cases du tableau en exactement $2^k$ essais.

### 1.2 Le travail à faire

Ce qui nous ammène au travail que vous devez réaliser. Vous devez implanter une classe `Hash` héritant de la classe abstraite `AbstractHash` qui

1. ne calculera la fonction de hachage qu'une fois par clef,
2. utilise un addressage ouvert,
3. peut visiter toutes les cases du tableau lors de collisions,

et ce en respectant les attentes de complexité envers une table de hachage.

Spécifiquement, vous devez

1. créer une fonction de hachage appropriée à partir d'une clef `String` de longueur quelconque, générant des valeurs entre 0 et `Integer.MAX_VALUE` (qui n'a aucunement besoin d'être parfaite, mais elle doit couvrir l'interval de résultats voulu),
2. implanter un constructeur public sans argument,
3. implanter la méthode `insert`,
4. implanter la méthode `delete` avec effacement parresseux,
5. implanter la méthode `find`,

avec la liberté totale sur la manière de le faire, sauf que vous ne pouvez pas utiliser de classes préexistantes (surtout pas une fonction de hachage, mais vous pouvez utiliser la méthode String.toCharArray sur les clefs).

Ceci dit, il vous faudra probablement définir deux classes (une pour les élément présents et une pour les pierres tombales), et plausiblement soit une classe abstraite ou une interface dont ces deux classes héritent ou que ces deux classes implantent. Il sera certainement nécessaire de conserver la clef, la valeur du *hash*, et la valeur associée à chaque entrée. Vous n'avez pas nécessairement à gérer l'insertion possible de valeurs nulles, et pouvez donc utiliser `null` comme symbole d'échec. Vous devez supporter à la fois l'agrandissement et le rétrécissement du tableau.

Veuillez remettre vos fichiers Java complétés sur StudiuM, parmi lesquels `Hash.java` hérite de la classe abstraite, avec les noms et matricules des auteurs en commentaire d'entête.

## 2 Partie Théorique (4 points)

1. ($1\frac{1}{2}$ points)
   (a) Supposons que nous ayons appliqué l'algorithme de Prim à un graphe de $N$ noeuds, en se servant d'une table avec le format utilisé dans le cours (et dans l'exemple disponible sur StudiUM). Expliquez comment on peut imprimer une liste des $N-1$ arêtes dans l'arbre sous-tendant minimal, avec le coût de chaque arête affiché à côté.

(b) Supposons que nous ayons appliqué l'algorithme de Dijkstra, en se servant encore d'une table $T$ avec le format utilisé dans le cours pour l'algorithme de Prim. Écrivez du pseudo-code (en utilisant la notation du cours, style $T[v].d$, ..., **pour**, **faire, si**, **sinon**, ...) pour une procédure récursive qui imprime le chemin pour se rendre du noeud d'origine au noeud donné. Par exemple, si le noeud d'origine était $v_4$, et le graphe était celui de l'exemple utilisé dans le cours pour illustrer l'algorithme de Prim, alors *Imprimer*$(v_4, T)$ va imprimer $v_4$ à $v_5$ à $v_7$ à $v_6$. Supposez disponible une fonction *Écrire*$(\ldots)$ capable d'écrire des chaînes de caractères et des sommets particuliers.

2. ($\frac{1}{2}$ point)
   Dans l'algorithme *Top-down* pour retirer un noeud d'une skip-liste déterministe, on "descendait" dans des "gaps" pour retrouver la clef à supprimer. Quand on était au niveau $h$ et on allait descendre dans une gap avec seulement un noeud de hauteur $h-1$, on a toujours "descendu" la hauteur $h$ du noeud qui définit la limite droite de la gap. Ici "descendre la hauteur $h$" veut dire remplacer la hauteur à $h-1$, et faire les modifications appropriées des pointeurs reliant les noeuds.

   La raison pour cette stratégie est très claire dans le cas $h = 2$. S'il n'y a qu'un seul noeud dans la gap, c'est le noeud de hauteur $h - 1 = 1$ avec la clef à retirer, et il serait impossible de retirer la clef en gardant bien-formée la structure skip-list.

   Par contre, aux hauteurs supérieures à $h = 2$, nous faisons quand même quelque chose de très semblable au retrait du noeud au niveau $h = 1$. Expliquez ce que c'est, et donnez un exemple simple qui montre que cela donnerait une erreur si on n'appliquait pas la règle pour les hauteurs $h > 2$.

3. (1 point)

   (a) Weiss, Exercice 9.1, page 417.

   (b) Weiss, Exercice 9.15, p. 419, algorithme de Prim seulement.

4. (1 point)

   (a) Est-ce qu'il est plus difficile de trouver un arbre sous-tendant *maximal*, avec l'algorithme de Prim, que de trouver un arbre sous-tendant *minimal* ?

   (b) Weiss, Exercice 9.16, page 419.
   Démontrez que l'algorithme fonctionne pour les coûts négatifs (vous pouvez prendre pour acquis que l'algorithme fonctionne pour les coûts non-négatifs), ou donnez un contre-exemple qui montre que l'algorithme ne fonctionne pas pour les coûts négatifs.

À réaliser en équipes de 1 ou 2. À remettre le 12 juillet, 2017, avant `9:00`. Les solutions seront affichées le 12 juillet. Les devoirs en retard ne seront pas acceptés.