UNIVERSITÉ DE MONTRÉAL
Département IRO

Été 2021
Neil Stewart

IFT 2015 E21

Devoir 3.

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

1 Partie Pratique (6 points)

Dans ce cours, vous avez vu les tables de hachage. Pour ce troisième (et dernier!) devoir, nous vous demandons d'en implanter une en *Java*, mais avec la particularité suivante : la taille de table devra être une puissance de deux, et l'opération de ré-hachage ne devra pas recalculer le *hash* de chaque clé.

On pourrait penser à priori que le ré-hachage se fait en $\Theta(N)$. Ceci est vrai en supposant que le temps moyen pour recalculer le *hash* d'une clé est en temps constant. Or, une bonne fonction de hachage devrait utiliser toute l'information contenue dans une clé. Ainsi, si la taille moyenne des clés n'est pas constante (ce qui risque d'arriver), le ré-hachage prendra un temps plus grand que $\Theta(N)$. Pour éviter cela, considérons l'approche suivante : prenons une table dont la taille est une puissance de 2, disons 2^k, et une fonction de hachage h_1 telle que $\forall x, 0 \leq h_1(x) \leq 2^{Max} - 1$, où $2^{Max} - 1$ est le plus grand entier positif qu'on peu exprimer dans notre langage de programmation (ce qui correspondrait à Integer.MAX_VALUE en java, soit $2^{31} - 1$, ce qui vaut 2147483647). Si h_1 est une bonne fonction de hachage, alors il en découle que $h_2(x) = h_1(x) \bmod 2^k$ sera aussi une bonne fonction de hachage, et dont l'image sera entre 0 et $2^k - 1$, soit les indexes possibles du tableau. Ainsi, nous n'avons qu'à calculer une seule fois $h_1(x)$, et à le garder en mémoire (vraisemblablement avec la clé x et l'élément inséré dans la table), et lorsque nous devons faire le ré-hachage vers une table de taille $2^{k'}$, nous n'avons qu'à prendre les k' derniers bits de $h_1(x)$ (sans la recalculer) en calculant $h_2(x)$ (en faisant $h_1(x) \bmod 2^{k'}$)). Ceci nous permettra donc d'avoir un ré-hachage réellement en temps $\Theta(N)$, et ce peu importe la taille des clés!

(Remarque : tant qu'à être efficace, notons aussi que calculer un modulo est une des opérations les plus lentes sur les entiers. Cependant, lorsque le terme de droite est une puissance de deux, le modulo peu se calculer avec un ET logique bit-à-bit (notons le par le symbole &, qui est le symbole utilisé en *Java*), de la façon suivante : $X \bmod 2^k = X\&(2^k - 1)$. Contrairement au modulo, le ET logique bit-à-bit est une des opérations les plus efficaces sur les entiers! Aussi, notons que 2^k peu se calculer efficacement à l'aide d'un *bit shift* (noté << en *Java*) de la façon suivante : $2^k = 1 << k$.)

On peut donc implémenter un ré-hachage dont le temps est indépendant de la

taille moyenne des clés, mais la technique présentée plus haut demande à ce que la taille de table soit une puissance de 2. Ceci pourrait peut-être poser problème, parce que les techniques présentées pour le *probing* exigeaient souvent que la taille de la table soit un nombre premier. Or, les puissance de deux (à l'exception de 2^1) ne sont pas des nombres premier. Ce n'est cependant pas un problème! Pour avoir un *probing* qui parcours toutes les cases, nous voudrions que pour chaque index $0 \leq j < 2^k$ il existe un $0 \leq i < 2^k$ tel que $j = (h(x) + i \cdot p(x)) \bmod 2^k$ (dans le cas du *linear probing*, on prend $p(x) = 1$). On peu démontrer (on ne le fera pas ici) que nous aurons effectivement un *probing* parcourant toutes les cases une seul fois si $p(x)$ est impaire.

(Remarque : si vous utilisez le *double hashing*, pour que le ré-hachage reste en $\Theta(N)$, il faudra que la valeur de $p(x)$ soit elle aussi gardée en mémoire, comme c'est le cas pour $h_1(x)$.)

Voici donc grosso modo les consignes que vous devez respecter :

- Vous devez obligatoirement utiliser l'adressage ouvert. Vous êtes cependant libre de choisir vous-même la façon dont vous voulez gérer les collisions (*linear probing*, *quadratic probing*, *double hashing*, etc)
- Vous êtes libre de choisir vous-même les seuils à partir desquels vous voulez faire le ré-hachage.
- Tel que décrit précédemment, la taille de la table devrait toujours être une puissance de 2 et le ré-hachage ne devrait pas recalculer les *hash* de chaque élément (de telle sorte à ce que le ré-hachage soit effectivement en $\Theta(N)$). Vous aurez donc vraisemblablement besoin de garder en mémoire les *hash* des clés dans la table (peut-être qu'il serait bon de vous faire une classe pour représenter le contenu d'une case?)
- Vous êtes libre de choisir vous-même l'algorithme utilisé par la (ou les) fonction(s) de hachage. Il faut que vous fassiez une "bonne" fonction de hachage qui évite le plus possible les collisions et dont la distribution de l'output est le plus uniforme possible. Vous devez implémenter vous-même cette fonction : vous ne pouvez pas en utiliser une déjà faite dans une librairie. Vous pouvez (et êtes encouragés) à vous renseigner sur les fonctions connues et en implémenter une qui existe déjà, en autant que vous citez vos sources!
- L'effacement d'un élément devrait utiliser une technique d'effacement paresseuse (avec des pierres tombales, par exemple. Il serait peut-être judicieux de faire une classe pour représenter une pierre tombale, et peut-être aussi une classe abstraite de laquelle cette classe et celle mentionnée dans le 3e point hériteraient).

Un fichier squelette nommé `HashTable.java` vous est déjà fourni. Des instructions plus détaillées à propos des méthodes à implanter sont décrites dans les

commentaires du fichier. Veuillez remettre votre version complétée du fichier sur StudiUM, avec les noms et matricules des auteurs en commentaire au début du fichier.

Ne changez pas le nom du fichier, ni les signatures des méthodes déjà présentes. Vous pouvez cependant (et vous êtes même encouragés à le faire!) vous ajouter des méthodes privées qui seraient utiles pour découper votre code. Si vous décidez d'ajouter une ligne "package" au début du fichier, retirez-la dans la version que vous soumettrez. Assurez-vous que votre fichier compile : il est difficile de donner des points pour du code qui ne compile pas!

Un fichier `Main.java` contenant quelques tests vous est aussi fourni. Vous pouvez l'utiliser comme bon vous semble pour tester votre implémentation. Vous n'avez pas à le remettre, et nous ne tiendrons pas compte des modifications que vous y ferez : nous ne regarderons que votre fichier `HashTable.java`, ainsi que n'importe quel autre fichier que vous aurez créer vous-même et qui serait utiliser par `HashTable.java`.

2 Partie Théorique (4 points)

1. (2 points)

   Le chiffre $\frac{1}{3}$ est souvent donné pour le facteur de gaspillage de mémoire du *buddy system.* Cet estimé est basé sur des observations empiriques. L'exercice suivant est donc intéressant.

   Supposons que nous devions allouer des blocs de taille $1, 2, 3, \ldots, n$, de sort que la quantité totale de mémoire à allouer est $1+2+3+\ldots+n$ cellules. Dans le *buddy system,* ça prendrait au moins $1+2+4+4+8+8+8+8+16+\ldots$ (il y a n termes dans la somme) cellules. Soient $b_n = 1+2+3+4+\ldots+n$, et $a_n = 1+2+4+4+8+8+8+8+16+\ldots$ (n termes).

   Soit α une constante telle que $n = 2^k\alpha$, $1 \leq \alpha < 2$. Montrez que

   - $b_n = 2^{2k-1}\alpha^2 + 2^{k-1}\alpha$
   - $a_n = 2^{2k+1}(\alpha - \frac{2}{3}) + \frac{1}{3}$.

   (Si vous n'arrivez pas à démontrer l'expression pour a_n, continuez quand méme. La preuve n'est pas si difficile que cela, mais un peu plus difficile que les autres étapes.)

   Nous avons donc $\frac{a_n}{b_n} \cong \rho(\alpha) \equiv \frac{4(\alpha-\frac{2}{3})}{\alpha^2}$ pour des grandes valeurs de n. Montrez aussi que

   - $\frac{4}{3} \leq \rho(\alpha) \leq \frac{3}{2}$, soit un facteur de gaspillage entre $\frac{1}{3}$ et $\frac{1}{2}$.

2. (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 (algorithme de Prim seulement).
   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.

3. (1 point) *Hachage pour chaînes*

   Il a été dit dans le cours qu'il est possible de faire le modulo au fur et à mesure dans le calcul de la fonction de hashing (règle de Horner).

   Si nous effectuons l'opération $a \leftarrow b \bmod M$ nous obtenons a tel que $a \equiv b \pmod{M}$ et $a \in [0, M-1]$. Nous nous intéressons au cas où a, b, x et M sont des entiers non-négatifs. Montrez que

$$(((ax) \bmod M) + b) \bmod M = (ax + b) \bmod M.$$

À réaliser en équipes de un ou deuz. À remettre le 14 juillet, 2021, avant `9:00`. Les devoirs en retard ne seront pas acceptés.