X2013 — INF411
Contrôle classant / 21 octobre 2014 / 8h30–11h30

Tous documents autorisés (poly, notes de cours, notes de TD). Dictionnaires électroniques autorisés pour les élèves étrangers.

L'énoncé est composé de 2 problèmes et d'un exercice, indépendants, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez une question, considérer comme déjà traitées les questions précédentes de la même partie.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

1 Adressage ouvert

Dans ce problème, on étudie une autre façon de réaliser une table de hachage. On souhaite représenter un *ensemble* d'au plus N éléments d'un type `E` donné. Ce type est muni d'une fonction de hachage, sous la forme d'une méthode `hashCode`, et d'une égalité, sous la forme d'une méthode `equals`. Pour les besoins des exemples ci-dessous, on se donne $N = 11$ valeurs $a, b, c, \ldots, k$ de type `E`, toutes distinctes pour `equals`, dont voici les valeurs par la méthode `hashCode` :

valeur `x`	a	b	c	d	e	f	g	h	i	j	k
`x.hashCode()`	5	42	15	68	10	0	31	4	2	27	5

Le principe de cette table de hachage est le suivant. On choisit un entier `M` strictement plus grand que N, par exemple `M` $= 16$, et on alloue un tableau de taille `M` dont toutes les cases sont initialisées avec `null`. Une case contenant `null` est dite *vide* et notée $\perp$ ci-dessous. Pour insérer un élément `x` dans la table, on commence par calculer l'entier `x.hashCode() % M`. Si la case du tableau correspondante est vide, on y insère l'élément `x`. Sinon, on considère la case suivante. Si elle est vide, on y insère `x`. Sinon, on considère de nouveau la case suivante. Et ainsi de suite. Si on atteint l'extrémité du tableau, alors on passe à la première case ; autrement dit, on utilise le tableau circulairement. Voici les valeurs de `x.hashCode() % M` pour les éléments $a, b, \ldots, k$ ci-dessus.

élément `x`	a	b	c	d	e	f	g	h	i	j	k
`x.hashCode() % M`	5	10	15	4	10	0	15	4	2	11	5

Si on insère alors successivement les valeurs a, b, c, d, e, f, g, h, dans cet ordre, dans une table initialement vide, on obtient la situation suivante :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f	g	$\perp$	$\perp$	d	a	h	$\perp$	$\perp$	$\perp$	b	e	$\perp$	$\perp$	$\perp$	c

(1)

Il n'est pas inutile de prendre quelques instants pour bien comprendre le processus d'insertion.

Question 1 Montrer que la recherche de l'élément h dans cette table nécessite trois comparaisons avec `equals`, que celle de j en nécessite une seule et que celle de i n'en nécessite aucune.

Question 2 Donner le contenu de la table (1) après l'insertion des trois éléments supplémentaires i, j, k, dans cet ordre.

Réalisation en Java. Pour réaliser une telle table de hachage en Java, on se donne la classe suivante :

```
class Table {
  final static int M = 16;
  private E[] elts;
  private int size;
```

Le champ `elts` contient un tableau de taille `M`. Le champ `size` contient le nombre d'éléments contenus dans la table, c'est-à-dire le nombre de valeurs dans le tableau `elts` qui ne sont pas `null`.

Question 3 Écrire un constructeur pour la classe `Table` qui renvoie une table initialement vide.

Question 4 Écrire une méthode `int size()` qui renvoie le nombre d'éléments contenus dans la table et une méthode `boolean isEmpty()` qui détermine si la table est vide.

Question 5 Écrire une méthode `void clear()` qui vide la table de tous ses éléments.

Question 6 Écrire une méthode `LinkedList<E> elements()` qui renvoie une liste contenant tous les éléments de la table. L'ordre des éléments dans cette liste n'est pas important.

Question 7 Écrire une méthode `int searchFor(E x)` qui cherche la valeur `x` dans la table et renvoie la position à laquelle `x` a été trouvé. Si `x` n'est pas dans la table, elle renvoie la position de la case vide sur laquelle la recherche s'est arrêtée. Ainsi, sur la table (1), `searchFor(`h`)` renvoie 6 et `searchFor(`j`)` renvoie 12.

Question 8 Justifier que la méthode `searchFor` termine toujours.

Question 9 Déduire de la méthode `searchFor` une méthode `boolean contains(E x)` qui détermine si l'élément `x` est dans la table.

Question 10 Déduire de la méthode `searchFor` une méthode `void add(E x)` qui ajoute l'élément `x` dans la table. Si `x` est déjà présent dans la table, `add(x)` ne fait rien.

Question 11 Expliquer pourquoi il ne suffit pas de mettre `null` dans la case 5 du tableau (1) pour supprimer l'élément a.

Question 12 Écrire une méthode `void remove(E x)` qui supprime l'élément `x` de la table, s'il est présent, et ne fait rien sinon. *Indication :* commencer par appeler `searchFor(x)` et par considérer le cas où `x` n'est pas dans la table ; sinon, effacer `x` en le remplaçant par `null` puis, pour toutes les cases non vides suivantes, remplacer l'élément qui s'y trouve par `null` et le réinsérer avec `add`.

Question 13 Que faire pour ne pas limiter la capacité totale de la table de hachage ? On ne demande pas d'écrire le code Java, mais seulement d'expliquer la démarche.

---

2 Tri par base

Dans ce problème, on étudie un algorithme de tri appelé *tri par base*. On suppose que les N éléments à trier sont des chaînes de caractères qui ont toutes la même longueur W et qu'on souhaite les trier par ordre alphabétique. Comme exemple, prenons un tableau contenant $N = 7$ chaînes qui ont toutes $W = 4$ caractères :

```
{ "beef", "maze", "blue", "mark", "deal", "fork", "byte" }
```
(2)

L'idée est la suivante : on commence par trier les chaînes selon le *dernier* caractère et on le fait de manière *stable*, c'est-à-dire en conservant l'ordre d'apparition des chaînes qui ont le même dernier caractère. Voici le résultat, le tableau étant représenté verticalement, avant et après :

0	b	e	e	f
1	m	a	z	e
2	b	l	u	e
3	m	a	r	k
4	d	e	a	l
5	f	o	r	k
6	b	y	t	e

↓

m	a	z	e
b	l	u	e
b	y	t	e
b	e	e	f
m	a	r	k
f	o	r	k
d	e	a	l

↓

On note en particulier que trois mots ont la même dernière lettre `'e'`, à savoir `"maze"`, `"blue"` et `"byte"`, et que leurs positions relatives ont été conservées.

On procède ensuite à un deuxième tri du tableau, cette fois selon *l'avant-dernier* caractère de chaque mot. Là encore, on effectue un tri stable qui conserve l'ordre des mots ayant le même caractère en avant-dernière position. Voici le résultat de ce second tri :

0	m	a	z	e
1	b	l	u	e
2	b	y	t	e
3	b	e	e	f
4	m	a	r	k
5	f	o	r	k
6	d	e	a	l

↓

d	e	a	l
b	e	e	f
m	a	r	k
f	o	r	k
b	y	t	e
b	l	u	e
m	a	z	e

↓

Là encore, on note que les deux mots ayant le même avant-dernier caractère `'r'`, à savoir `"mark"` et `"fork"`, ont conservé leurs positions relatives.

On procède ensuite à un troisième tri avec le deuxième caractère de chaque mot, puis enfin à un quatrième tri avec le premier caractère de chaque mot. À chaque fois, on fait un tri stable. Voici le résultat de ces deux derniers tris :

0	d	e	a	l
1	b	e	e	f
2	m	a	r	k
3	f	o	r	k
4	b	y	t	e
5	b	l	u	e
6	m	a	z	e

↓

m	a	r	k
m	a	z	e
d	e	a	l
b	e	e	f
b	l	u	e
f	o	r	k
b	y	t	e

↓

↓

b	e	e	f
b	l	u	e
b	y	t	e
d	e	a	l
f	o	r	k
m	a	r	k
m	a	z	e

↓

Comme on le constate, on obtient au final un tableau qui est trié pour l'ordre alphabétique. D'une manière générale, le tri par base procède ainsi :

pour $i = W-1, \ldots, 1, 0$
    trier le tableau selon le caractère i de chaque chaîne, avec un tri stable

Il n'est pas inutile de prendre quelques instants pour bien comprendre cet algorithme.

Correction de l'algorithme. On note $\leq$ l'ordre alphabétique sur les chaînes. Si s est une chaîne et i un entier, on note s_i le caractère d'indice i de s (qu'on écrirait `s.charAt(`i`)` en Java). Pour $0 \leq i \leq W$, on note $s[i..[$ le suffixe de s qui démarre au caractère i (qu'on écrirait `s.substring(`i, W`)` en Java). En particulier, $s[W..[$ est la chaîne vide.

Question 14 Montrer que le tableau est trié à l'issue de l'algorithme du tri par base. *Indication :* On pourra procéder par récurrence sur $W - i$.

Réalisation en Java. Pour réaliser le tri par base en Java, on suppose que les caractères des chaînes prennent des valeurs entre `'a'` et `'z'`. On va exploiter cette information pour réaliser les différents passes de tri en utilisant 26 files. On se donne donc la classe suivante pour réaliser le tri par base :

```
class RadixSort {
  final static             int R = 26;  // caractères entre 'a' et 'z'
  private                  int W;       // longueur des chaînes
  private LinkedList<String>[] queues;  // tableau de R files
```

Le constructeur de cette classe initialise `W` et `queues`. On ne demande pas de l'écrire. On rappelle qu'une `LinkedList` s'utilise comme une file grâce aux méthodes `add`, qui ajoute un élément à la fin de la file, et `poll`, qui supprime et renvoie l'élément au début de la file.

Question 15 Écrire une méthode `void fillQueues(String[] a, int i)` qui parcourt le tableau `a`, dans l'ordre, et ajoute chacun de ses éléments dans la file correspondant à son `i`-ième caractère.

Question 16 Donner le contenu des 26 files après l'appel à `fillQueues(a, 3)` pour le tableau `a` de l'exemple (2). (On rappelle que les caractères sont indexés à partir de 0 et que 3 désigne donc ici le dernier caractère de chaque chaîne.)

Question 17 Écrire une méthode `void emptyQueues(String[] a)` qui remplit le tableau `a` avec les éléments contenus dans les files `queues`$[0], \ldots,$ `queues`$[$`R`$-1]$, dans cet ordre. Les files doivent être vides à la fin de l'exécution.

Question 18 Écrire une méthode `void sort(String[] a)` qui réalise le tri par base. On se servira des méthodes `fillQueues` et `emptyQueues` pour réaliser le tri stable de chaque étape.

Question 19 Montrer que le coût total de la méthode `sort` est proportionnel à NW. Cela est-il en contradiction avec le résultat vu en cours qui dit que la complexité optimale du tri est $N \log N$?

Question 20 Expliquer comment cet algorithme de tri peut être adapté pour trier un tableau d'entiers de type `int`. On ne demande pas d'écrire le code Java, mais seulement d'expliquer la démarche.

---

3 Exercice

On se donne la classe suivante pour représenter des arbres binaires persistants dont les nœuds sont étiquetés par des valeurs d'un certain type `E`.

```
class Tree {
  final    E value;
  final Tree left, right;

  Tree(Tree left, E value, Tree right) {
    this.value = value;
    this.left  = left;
    this.right = right;
  }
}
```

Écrire une méthode statique `Tree ofList(LinkedList<E> l)` qui prend en argument une liste `l` contenant N éléments $x_1, x_2, \ldots, x_N$ et renvoie un arbre binaire dont les nœuds sont étiquetés par $x_1, x_2, \ldots, x_N$ quand ils sont parcourus dans l'ordre infixe.

On demande que la complexité soit $O(N)$ et que l'arbre renvoyé ait une hauteur en $O(\log N)$. On pourra écrire une méthode auxiliaire si nécessaire.

* *
*