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

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

Correction :

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

---

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.

Correction : Immédiat.

```
  Table() {
    this.elts = new E[M];  // note : pas une classe générique
    this.size = 0;
  }
```

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.

Correction : Immédiat.

```
  public int size() { return this.size; }
  public boolean isEmpty() { return this.size == 0; }
```

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

Correction : Immédiat. Bien sûr, on peut utiliser une boucle `for` si on ne connaît pas `Arrays.fill`.

```
  public void clear() {
    Arrays.fill(this.elts, null);
    this.size = 0;
  }
```

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.

Correction :

```
public LinkedList<E> elements() {
  LinkedList<E> r = new LinkedList<E>();
  for (E x: this.elts) if (x != null) r.add(x);
  return r;
}
```

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 arr tée. Ainsi, sur la table (1), `searchFor(`h`)` renvoie 6 et `searchFor(`j`)` renvoie 12.

Correction :

```
private int searchFor(E x) {
  int i = x.hashCode() % M;
  while (this.elts[i] != null && !this.elts[i].equals(x)) {
    i++;
    if (i == this.M) i = 0;
  }
  return i;
}
```

Note : on pourrait aussi écrire

```
i = (i + 1) % this.M;
```

m me si cela est un peu plus coûteux.

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

Correction : Soit on trouve l'élément `x` soit on atteint forcément une case vide car $\texttt{M} > N$ et donc il y a toujours au moins une case vide.

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

Correction :

```
public boolean contains(E x) {
  int i = searchFor(x);
  return this.elts[i] != null;
}
```

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.

Correction :

```
public void add(E x) {
  int i = searchFor(x);
  if (this.elts[i] != null) return; // déjà présent
  this.elts[i] = x;
  this.size++;                      // à ne pas oublier
}
```

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

Correction : Si on cherche ensuite h, on ne le trouvera plus, car le trou qui vient d' tre ajouté interrompt la recherche.

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`.

Correction :

```
public void remove(E x) {
  int i = searchFor(x);
  if (this.elts[i] == null) return; // absent
  this.elts[i] = null;
  this.size--;
  i++; if (i == this.M) i = 0;
  while (this.elts[i] != null) {
    E e = this.elts[i];
    this.elts[i] = null;
    this.size--;                    // car add va faire size++
    add(e);
    i++; if (i == this.M) i = 0;
  }
}
```

Noter la mise à jour de `size`, qui est subtile.

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.

Correction : Il suffit d'utiliser un tableau redimensionnable plutôt qu'un tableau pour le champ `this.elts`. Dans la méthode `add`, on double la taille de ce tableau dès que la charge atteint 1/2 (et on y réinsère tous les éléments). Dans la méthode `remove`, on divise par deux la taille de ce tableau dès que la charge atteint 1/8 (et de m me on y réinsère tous les éléments).

---

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 m me 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 m me 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 m me 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 m me 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

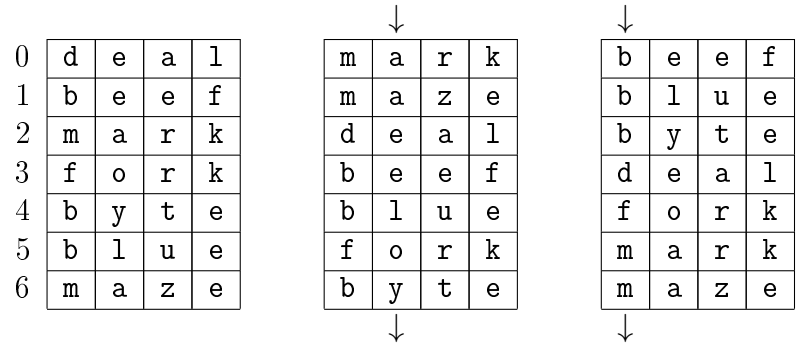
↓

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 m me 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 :



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 $W - i$.

Correction : On montre par récurrence sur $W - i$ que, à l'issue de l'étape i, le tableau a est trié pour la relation $s \leq_i t$ définie comme $s[i..[\leq t[i..[$. Initialement, le tableau est trié pour $\leq_W$. Supposons-le trié pour $\leq_{i+1}$. On effectue un tri stable selon le caractère i. Soient $0 \leq j \leq k < N$. Si $a[j]_i \neq a[k]_i$ alors $a[j]_i < a[k]_i$ car on vient de trier selon ce caractère. Sinon, le caractère stable du tri assure que $a[j] \leq_{i+1} a[k]$ et donc $a[j] \leq_i a[k]$.

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.

Correction :

```
private void fillQueues(String[] a, int i) {
  for (String s: a)
    this.queues[s.charAt(i) - 'a'].add(s);
}
```

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.)

Correction : Quatre files contiennent des éléments, à savoir :

```
'e':  <- maze, blue, byte <-
'f':  <- beef             <-
'k':  <- mark, fork       <-
'l':  <- deal             <-
```

Les 22 autres files sont vides.

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

Correction :

```
private void emptyQueues(String[] a) {
  int j = 0;
  for (LinkedList<String> q: this.queues)
    while (!q.isEmpty())
      a[j++] = q.poll();
  assert j == a.length;
}
```

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.

Correction :

```
void sort(String[] a) {
  for (int i = this.W - 1; i >= 0; i--) {
    fillQueues(a, i);
    emptyQueues(a);
  }
}
```

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 $N \log N$?

Correction : Chaque passe a clairement un coût proportionnel à N car chaque élément est inséré une fois dans une file et retiré une fois d'une file. On répète ceci exactement W fois.

Non, il n'y a pas contradiction, car on n'est plus ici dans le modèle où l'on fait uniquement des comparaisons et sans posséder d'information particulière sur les éléments. En particulier, on exploite ici le fait que les éléments sont des chaînes, qu'elles ont toutes la m me longueur et que leurs caractères prennent des valeurs entre `'a'` et `'z'`. Ce dernier point implique notamment que $\log N \leq W \log(26)$.

Question 20 Expliquer comment cet algorithme de tri peut tre 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.

Correction : Si les entiers sont positifs ou nuls, il suffit de considérer que le i-ième « caractère » de chaque entier x est son i-ième chiffre en base 10 en partant de la droite, c'est-à-dire $\lfloor x/10^i \rfloor$ modulo 10. On a donc $\texttt{R} = 10$ files. Mieux encore, on peut considérer les entiers octets par octets, c'est-à-dire prendre $\texttt{R} = 256$, avec $(x\texttt{>>}8i)\texttt{\&}255$.

En présence d'entiers négatifs, il faut procéder plus subtilement. Si on a opté pour le découpage en octets, par exemple, la quatrième et dernière passe doit interpréter le signe, en faisant $(x\texttt{>>}24) + 128$ au lieu de $(x\texttt{>>}24)\texttt{\&}255$.

---

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.

Correction : On commence par écrire une méthode qui construit un arbre à partir des `n` premiers éléments d'une liste, en les consommant. Le cas de base est `n` $= 0$. Sinon, on coupe en deux, en prenant un élément dans la liste entre les deux appels récursifs.

```
static Tree ofList(LinkedList<E> l, int n) {
  if (n == 0) return null;
  int n1 = (n - 1) / 2;
  Tree left = ofList(l, n1);
  E v = l.poll();
  Tree right = ofList(l, n - n1 - 1);
  return new Tree(left, v, right);
}
```

On en déduit facilement la réponse à la question :

```
static Tree ofList(LinkedList<E> l) {
  return ofList(l, l.size());
}
```

La complexité est clairement linéaire (par récurrence sur `n`) car on ne fait qu'une quantité bornée de calculs entres les appels récursifs.

Pour ce qui est de la hauteur, montrons par récurrence sur h que si `n` $< 2^h$ alors l'arbre construit par `ofList(l,n)` a une hauteur au plus h. C'est vrai pour $h = 0$. Sinon, prenons `n` > 0. On a `n1` $<$ `n`$/2 < 2^{h-1}$ d'une part et `n-n1-1` $\leq$ `n`$/2 < 2^{h-1}$ d'autre part. Donc par hypothèse de récurrence les deux sous-arbres construits récursivement ont une hauteur au plus $h - 1$.

Note : C'est sans doute la solution la plus simple mais ce n'est pas la seule. On pouvait aussi stocker tous les éléments dans un tableau `a` puis écrire un méthode `buildTree(a, l, r)` qui construit un arbre à partir des éléments du segment `a[l..r[`, récursivement, en partageant à chaque fois l'intervalle en deux moitiés égales.

---

* *
*