X2014 — INF411
Contrôle classant / 2 novembre 2015 / 8h30–11h30

Tous documents autorisés (poly, notes de cours, notes de TD). Dictionnaires électroniques autorisés.

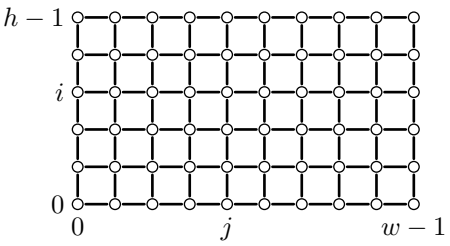
L'énoncé est composé de 3 problèmes indépendants, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez une question, considérer comme déjà traitées les questions précédentes du même problème.

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.

Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Construction d'un labyrinthe parfait

Dans ce problème, on se propose de construire un labyrinthe à l'aide de l'algorithme de parcours en profondeur. Pour deux entiers w et h donnés, on considère le graphe non orienté suivant contenant $w \times h$ sommets :



Chaque sommet est un couple (i, j), avec $0 \leq i < h$ et $0 \leq j < w$, que l'on représente par la classe `Cell` donnée figure 1.

Question 1 Redéfinir la méthode `equals` de la classe `Cell`, pour qu'elle corresponde à l'égalité structurelle, et sa méthode `hashCode` pour qu'elle soit cohérente avec `equals`.

Correction :

```
public boolean equals(Object o) {
  Cell that = (Cell)o;
  return this.i == that.i && this.j == that.j;
}
public int hashCode() { return 5003 * i + j; }
```

Le graphe lui-même est représenté par la classe `Grid` également donnée figure 1. Cette classe contient une table de hachage qui associe à chaque sommet la liste de ses voisins dans le graphe.

Question 2 Écrire un constructeur `Grid(int h, int w)` qui prend les valeurs de h et w en arguments et construit le graphe illustré plus haut.

Correction : La seule subtilité consiste à ne pas oublier d'appeler `addVertex` avant de chercher à ajouter des arcs.

```
Grid(int h, int w) {
  this.adj = new HashMap<>();
  for (int i = 0; i < h; i++)
    for (int j = 0; j < w; j++) {
      Cell c = new Cell(i, j);
      addVertex(c);
      if (i > 0)   addEdge(c, new Cell(i-1, j));
      if (i < h-1) addEdge(c, new Cell(i+1, j));
      if (j > 0)   addEdge(c, new Cell(i, j-1));
      if (j < w-1) addEdge(c, new Cell(i, j+1));
    }
}
```

Question 3 Écrire une méthode `static LinkedList<Cell> shuffle(LinkedList<Cell> l)` qui prend en argument une liste `l` et renvoie une *nouvelle* liste contenant les mêmes éléments que `l` mais dans un ordre aléatoire. La liste `l` ne doit pas être modifiée.

Correction : On a vu en cours comment mélanger les éléments d'un tableau (mélange de Knuth) de façon équiprobable. Il suffit de se ramener à cette situation :

```
static LinkedList<Cell> shuffle(LinkedList<Cell> l) {
  int n = l.size();
  Cell[] a = new Cell[n];
  for (Cell c: l) a[--n] = c;
  shuffle(a); // mélange de Knuth d'un tableau
  LinkedList<Cell> r = new LinkedList<>();
  for (Cell c: a) r.add(c);
  return r;
}
```

Pour réaliser le parcours en profondeur, on ajoute à la classe `Grid` un nouveau champ `visited` de type `HashMap<Cell, Cell>` qui associe à chaque sommet déjà visité le sommet qui a permis de l'atteindre. Par convention, le sommet initial est associé à `null`.

Question 4 En utilisant la méthode `shuffle`, écrire une méthode `void dfs()` qui réalise un parcours en profondeur *aléatoire* à partir du sommet $(0, 0)$, en remplissant la table `visited`.

Correction : La seule différence avec le cours est le mélange des voisins avant de les parcourir :

```
void dfs(Cell from, Cell v) {
  if (this.visited.containsKey(v)) return;
  this.visited.put(v, from);
  for (Cell w: shuffle(this.adj.get(v)))
    dfs(v, w);
}
void dfs() {
  dfs(null, new Cell(0, 0));
}
```

Question 5 Quelle est la complexité de ce parcours ?

Correction : On a vu en cours que la complexité est $O(E)$ donc ici $O(w \times h)$.

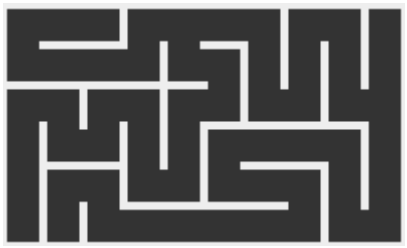
Question 6 Donner un résultat possible de ce parcours en profondeur pour $w = 4$ et $h = 3$, en indiquant l'ordre de découverte des sommets.

Correction : Par exemple

6	5	4	7
1	2	3	8
0	11	10	9

mais il en existe beaucoup d'autres.

Question 7 Pour construire un labyrinthe à partir du résultat de ce parcours en profondeur, on procède ainsi : chaque sommet du graphe représente une pièce et chaque arc emprunté pendant le parcours en profondeur constitue un couloir entre deux pièces. Voici un exemple pour $w = 10$ et $h = 6$.



Justifier que le labyrinthe construit par cette méthode est toujours un labyrinthe *parfait*, c'est-à-dire un labyrinthe où deux pièces sont reliées par un chemin et un seul.

Correction : Il y a deux points à justifier :

existence du chemin

C'est là une propriété du parcours en profondeur vue en cours (amphi 7). Tout sommet (i, j) est accessible depuis $(0, 0)$, par exemple par le chemin qui va d'abord toujours à droite puis ensuite toujours vers le haut. Comme $(0, 0)$ a forcément été visité, si (i, j) n'était pas visité par le parcours en profondeur, alors il existerait un

```
import java.util.*;


class Cell {
  final int i, j;

  Cell(int i, int j) {
    this.i = i;
    this.j = j;
  }
}
```

```
class Grid {
  HashMap<Cell, LinkedList<Cell>> adj;

  void addVertex(Cell x) {
    this.adj.put(x, new LinkedList<>());
  }

  void addEdge(Cell x, Cell y) {
    this.adj.get(x).add(y);
  }
}
```

FIGURE 1 – Classes `Cell` et `Grid`.

premier arc $x \to y$ sur ce chemin où x a été visité mais pas y, ce qui contredirait le code du parcours en profondeur. Il existe donc un chemin entre $(0,0)$ et (i,j) et donc un chemin entre deux pièces quelconques (en passant par $(0,0)$).

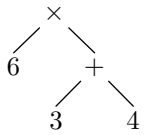
unicité du chemin

Supposons qu'il existe deux chemins distincts entre x et y. Alors il existe un sommet z avec $a \to z$ sur le premier chemin et $b \to z$ sur le second chemin pour deux sommets distincts a et b. Mais cela signifie alors que z a été visité deux fois, ce qui contredit là encore le code du parcours en profondeur.

---

2 Expressions arithmétiques

Dans ce problème, on considère des expressions arithmétiques construites à partir de constantes entières, de l'addition et de la multiplication. Une telle expression peut être représentée par un *arbre*, où un nœud interne représente soit une addition, soit une multiplication, et où une feuille contient une constante. Ainsi l'expression $6 \times (3 + 4)$ est représentée par l'arbre



On se donne une classe abstraite `Expr` pour représenter de telles expressions.

```
abstract class Expr { }
```

Question 8 Introduire trois sous-classes `Const`, `Add` et `Mul` de `Expr` pour représenter respectivement une constante, une addition et une multiplication, avec leurs constructeurs. On doit pouvoir construire l'expression ci-dessus avec `new Mul(new Const(6), new Add(new Const(3), new Const(4)))`.

Correction :

```
class Const extends Expr {
  private int n;
  Const(int n) { this.n = n; } }
class Add extends Expr {
  private Expr e1, e2;
  Add(Expr e1, Expr e2) { this.e1 = e1; this.e2 = e2; } }
```

La classe `Mul` est identique à la classe `Add`.

Question 9 Déclarer une méthode `int eval()` dans la classe `Expr` et la définir dans chacune des trois sous-classes, de telle sorte que `e.eval()` renvoie la valeur de l'expression représentée par `e`.

Correction :

```
abstract class Expr { abstract int eval(); }
class Const extends Expr { ... int eval() { return this.n; } }
class Add extends Expr { ... int eval() { return e1.eval()+e2.eval(); }}
class Mul extends Expr { ... int eval() { return e1.eval()*e2.eval(); }}
```

Notation post-fixée. La notation post-fixée (en anglais RPN, pour *Reverse Polish Notation*) est une façon d'écrire une expression arithmétique sans utiliser de parenthèses, en indiquant les opérations *après* leurs opérandes. Ainsi, la notation post-fixée pour l'expression $6 \times (3 + 4)$ est 6 3 4 + $\times$. Dans les deux questions suivantes, on représente la notation post-fixée par une liste de chaînes de caractères, de type `LinkedList<String>`, où chaque élément est soit la chaîne `"+"`, soit la chaîne `"*"`, soit une chaîne représentant un entier (telle que `"6"` ou encore `"-7"`).

Question 10 Ajouter une méthode `LinkedList<String> toRPN()` à la classe `Expr`. Elle doit renvoyer une représentation post-fixée de l'expression. Pour l'expression `e` utilisée en exemple plus haut, `e.toRPN()` doit renvoyer la liste `["6", "3", "4", "+", "*"]`. La complexité doit être proportionnelle à la taille de l'expression.

Indication : on pourra introduire une méthode plus générale, prenant en argument une liste et ajoutant à la fin de cette liste la représentation post-fixée de l'expression.

Correction : On suit l'indication en introduisant une méthode `toRPN` prenant `l` en argument. La méthode demandée en découle :

```
abstract class Expr {
  abstract void toRPN(LinkedList<String> l);
  LinkedList<String> toRPN() {
    LinkedList<String> l = new LinkedList<>(); toRPN(l); return l;
  }
```

Reste à écrire `toRPN` dans les trois sous-classes :

```
class Const ...
  void toRPN(LinkedList<String> l) { l.add("" + this.n); }
class Add ...
  void toRPN(LinkedList<String> l) {
    this.e1.toRPN(l); this.e2.toRPN(l); l.add("+");
  }
```

et de même pour `Mul`.

Question 11 Écrire une méthode `static Expr fromRPN(LinkedList<String> l)` qui convertit une représentation post-fixée, donnée sous la forme d'une liste `l` de chaînes de caractères, en expression. On suppose que `l` contient la représentation post-fixée d'une unique expression. En particulier, `fromRPN(e.toRPN())` doit renvoyer une expression identique à `e`. La liste `l` ne doit pas être modifiée.

Indication : on pourra se servir d'une *pile* de type `Stack<Expr>` contenant des expressions déjà reconnues.

Correction :

```
static Expr fromRPN(LinkedList<String> l) {
  Stack<Expr> st = new Stack<>();
  for (String s: l) {
    switch(s) {
    case "+": st.push(new Add(st.pop(), st.pop())); break;
    case "*": st.push(new Mul(st.pop(), st.pop())); break;
    default: st.push(new Const(Integer.parseInt(s)));
    }
  }
  return st.pop();
}
```

Notation usuelle. On souhaite maintenant reconnaître une expression écrite de façon usuelle, c'est-à-dire avec des opérations infixes et des parenthèses. Pour simplifier, on suppose que *toute* opération d'addition ou de multiplication est parenthésée et qu'il n'y a pas d'autres parenthèses. Ainsi, l'expression utilisée en exemple plus haut s'écrit $(6 \times (3 + 4))$.

Question 12 Écrire une méthode `static Expr parse(LinkedList<String> l)` qui reconnaît une expression décrite par la liste `l` dans la notation usuelle. On suppose que la liste `l` contient la représentation d'une unique expression, telle que la liste `["(", "6", "*", "(", "3", "+", "4", ")", ")"]` par exemple. La liste `l` ne doit pas être modifiée.

Indication : on pourra se servir de *deux piles*, l'une de type `Stack<Expr>` contenant des expressions déjà reconnues et l'autre de type `Stack<String>` contenant des chaînes `"+"` ou `"*"` pour lesquelles on n'a pas encore reconnu la seconde opérande.

Correction :

```
static Expr parse(LinkedList<String> l) {
  Stack<Expr> exprs = new Stack<>();
  Stack<String> ops = new Stack<>();
  for (String s: l)
    switch(s) {
    case "(": break;
    case ")":
      Expr e1 = exprs.pop(), e2 = exprs.pop();
      exprs.push(ops.pop().equals("+") ? new Add(e1,e2) : new Mul(e1,e2));
      break;
    case "+":
    case "*": ops.push(s); break;
    default: exprs.push(new Const(Integer.parseInt(s)));
  }
  return exprs.pop();
}
```

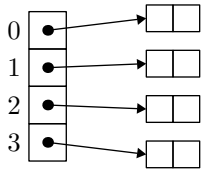
Note : il n'est pas nécessaire d'utiliser `switch`. On peut écrire `if (s.equals(")") ... else if (s.equals("+")) ... else ...`.

---

3 Tables de hachage extensibles

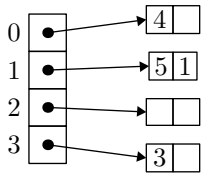
Lorsque la charge d'une table de hachage devient trop importante, une solution consiste à la remplacer par une autre table deux fois plus grande dans laquelle on ré-insère tous les éléments. Dans ce problème, on étudie une stratégie de redimensionnement différente, qui évite notamment d'avoir à ré-insérer tous les éléments.

L'idée est la suivante. La table de hachage est un tableau de taille 2^n, dont chaque élément contient initialement un seau de capacité finie. Si $n = 2$ et si chaque seau a une capacité initiale de deux éléments, on a donc la situation initiale suivante :



Le seau d'indice 0 reçoit les éléments pour lesquels la valeur de hachage vaut 0 modulo 4, le seau d'indice 1 ceux pour lesquels elle vaut 1 modulo 4, etc.

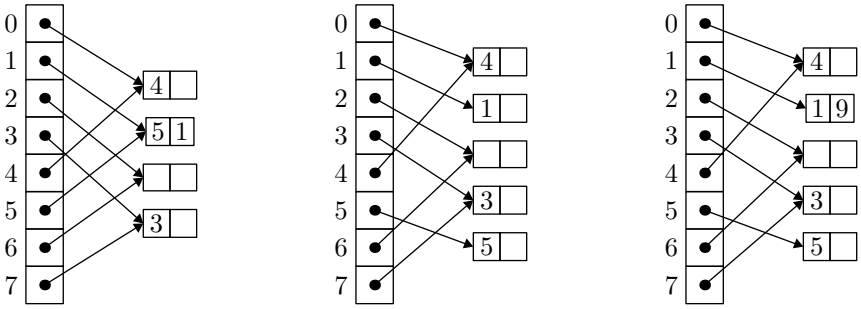
Pour simplifier les choses, supposons que les éléments à stocker dans la table de hachage sont des entiers et que la fonction de hachage est l'identité. Si on insère successivement les éléments 5, 4, 3 et 1, on arrive à la situation suivante :



Si on souhaite maintenant insérer l'élément 9, alors on a un problème car le seau correspondant (d'indice 1) est plein. On procède alors en trois étapes.

1. On double la taille du tableau et on recopie la première moitié dans la seconde, en partageant les seaux entre la première et la seconde moitié ;
2. On remplace le seau qui était plein par deux nouveaux seaux, aux indices 1 et 5 respectivement, dans lesquels on range les éléments du seau qui était plein (l'élément 1 se retrouve dans le seau 1 et l'élément 5 dans le seau 5, car la valeur de hachage est maintenant considérée modulo 8) ;
3. puis enfin on insère le nouvel élément 9 (dans le seau 1).

Ces trois étapes sont illustrées ci-dessous :

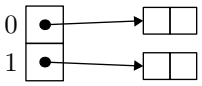


Comme on le constate, certains seaux sont communs à plusieurs indices (par exemple le seau des indices 0 et 4) alors que d'autres ne le sont pas (par exemple le seau d'indice 1). Pour faire cette distinction, une table de taille 2^n est dite de *niveau* n et un seau est dit de niveau $k \leq n$ s'il est partagé entre 2^{n-k} indices. Dans la dernière situation ci-dessus, on a donc une table de niveau 3, deux seaux de niveau 3 (aux indices 1 et 5) et trois seaux de niveau 2 (aux paires d'indices $\{0, 4\}$, $\{2, 6\}$ et $\{3, 7\}$ respectivement).

D'une manière générale, l'insertion d'un nouvel élément x dans la table est réalisée par l'algorithme suivant :

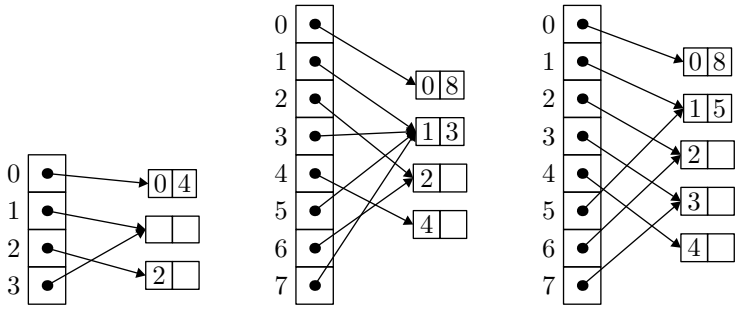
1. On calcule l'indice i du seau dans lequel x doit être rangé.
2. Si le seau i n'est pas plein, on y ajoute x et on a terminé.
3. Si le niveau du seau i est strictement inférieur à celui de la table, on passe directement à l'étape 4. Sinon, on double la taille du tableau, on recopie sa première moitié dans sa seconde moitié et on incrémente le niveau de la table.
4. Le niveau k du seau i est strictement inférieur au niveau n de la table. Le seau i est commun à 2^{n-k} indices de la table. On remplace alors ce seau par deux nouveaux seaux de niveau $k+1$, qui vont chacun être commun à 2^{n-k-1} indices de la table. Plus précisément, le premier seau est placé aux indices $i + j \times 2^{k+1}$ et le second seau aux indices $i + 2^k + j \times 2^{k+1}$, tous les indices étant compris modulo 2^n, pour $0 \leq j < 2^{n-k-1}$.
5. On insère les éléments de l'ancien seau i ainsi que x dans ces deux nouveaux seaux. Cette fois, si malgré tout un seau vient à être plein, on se contente d'en doubler la capacité.

Question 13 Donner la situation finale après l'insertion successive des éléments $0, 2, 4, 8, 1, 3, 5$, dans cet ordre, dans une table initialement vide de niveau 1, c'est-à-dire



(On suppose toujours que la fonction de hachage est l'identité.)

Correction : On a successivement les situations suivantes :



Réalisation. La figure 2 contient le code d'une classe `Bucket` servant à représenter un seau. Il ne sera pas demandé d'écrire du code dans cette classe. En revanche, on prendra le temps de bien lire et comprendre le code de cette classe. La table de hachage proprement dite est réalisée par une classe `HashTable` de la forme suivante

```
class HashTable<E> {
  private int level; // le niveau de la table
  private final Vector<Bucket<E>> buckets; // ses seaux
  ...
}
```

où le champ `buckets` est un tableau redimensionnable de taille $2^{\texttt{level}}$. On se donne par ailleurs trois méthodes statiques pour simplifier l'écriture du code à venir.

```
class Bucket<E> {
  final int       level;    // le niveau du seau
  final Vector<E> elements; // ses éléments
  private int     capacity; // la capacité maximale du seau

  final static int initialCapacity = 2;

  Bucket(int level) {
    this.level    = level;
    this.elements = new Vector<>(initialCapacity);
    this.capacity = initialCapacity;
  }

  boolean contains(E x) {
    return this.elements.contains(x);
  }

  boolean add(E x) {
    if (this.elements.size() == this.capacity) return false;
    this.elements.add(x);
    return true;
  }

  void reallyAdd(E x) {
    if (this.elements.size() == this.capacity)
      this.capacity *= 2;
    this.elements.add(x);
  }
}
```

FIGURE 2 – La classe `Bucket`.

`static int power2(int n)`	renvoie $2^{\texttt{n}}$
`static int modulo(int x, int n)`	renvoie `x` modulo $2^{\texttt{n}}$
`static int concat(int y, int x, int n)`	renvoie $\texttt{y} \times 2^{\texttt{n}} + (\texttt{x}$ modulo $2^{\texttt{n}})$

On ne demande pas d'écrire le code de ces trois méthodes.

Question 14 Écrire un constructeur `HashTable(int level)` qui construit une table vide de niveau `level`, dont chaque seau est un nouveau seau de niveau `level`.

Correction : Pas de difficulté, si ce n'est qu'il ne faut pas oublier d'appeler `setSize` sur le tableau redimensionnable `buckets`.

```
HashTable(int level) {
  this.level = level;
  int n = power2(level);
  this.buckets = new Vector<Bucket<E>>(n);
  this.buckets.setSize(n); // à ne pas oublier
  for (int i = 0; i < n; i++)
    this.buckets.set(i, new Bucket<>(level));
}
```

Question 15 Écrire une méthode `boolean contains(E x)` dans la classe `HashTable` qui détermine si l'élément `x` apparaît dans la table.

Correction : Ce n'est pas différent de la recherche dans une table de hachage usuelle :

```
boolean contains(E x) {
  int i = modulo(x.hashCode(), this.level);
  return this.buckets.get(i).contains(x);
}
```

Question 16 Écrire une méthode `void doubleSize()` dans la classe `HashTable` qui réalise l'étape 3 de l'algorithme ci-dessus.

Correction : Il ne faut pas oublier d'incrémenter le niveau de la table.

```
private void doubleSize() {
  int n = this.buckets.size();
  this.buckets.setSize(2 * n);
  this.level++;
  for (int i = 0; i < n; i++)
    this.buckets.set(n + i, this.buckets.get(i));
}
```

Question 17 Écrire une méthode `void add(E x)` dans la classe `HashTable` qui réalise l'insertion d'un nouvel élément `x` selon l'algorithme décrit plus haut. On suppose que `x` n'est pas déjà présent dans la table.

Indication : Commencer par écrire une méthode `void reallyAdd(E x)` qui ajoute `x` dans le seau qui lui correspond avec la méthode `reallyAdd` de la classe `Bucket`. Cette méthode pourra alors être utilisée dans l'étape 5 de l'algorithme.

Correction :

```
void add(E x) {
  int i = modulo(x.hashCode(), this.level);
  Bucket<E> b = this.buckets.get(i);
  if (b.contains(x)) return;
  if (b.add(x)) return;
  if (b.level == this.level) doubleSize();
  Bucket<E> b0 = new Bucket<>(b.level + 1);
  Bucket<E> b1 = new Bucket<>(b.level + 1);
  for (int j = 0; j < power2(this.level - b.level - 1); j++) {
    this.buckets.set(concat(2*j  , b.level, i), b0);
    this.buckets.set(concat(2*j+1, b.level, i), b1);
  }
  for (E v: b.elements) reallyAdd(v);
  reallyAdd(x);
}

private void reallyAdd(E x) {
  int i = modulo(x.hashCode(), this.level);
  this.buckets.get(i).reallyAdd(x);
}
```

Question 18 Donner un scénario pouvant conduire à une table contenant N éléments et occupant un espace $O(2^N)$. Proposer une solution pour y remédier.

Correction : Si on insère successivement les éléments $2, 4, \ldots, 2^i$, alors ils tombent systématiquement dans le seau numéro 0 et la table double donc de taille toutes les deux insertions.

Une solution possible consiste à n'effectuer l'étape 3 de l'algorithme que lorsque la charge de la table dépasse une valeur fixée (par exemple 1/2).

A Bibliothèque standard Java

`class LinkedList<E>`	
`void add(E e)`	ajoute l'élément `e` à la fin de la liste
`class Stack<E>`	
`void push(E e)`	ajoute l'élément `e` au sommet de la pile
`E pop()`	supprime et renvoie le sommet de la pile
`class Integer`	
`int parseInt(String s)`	convertit la chaîne `s` en entier
`class HashMap<K, V>`	
`void put(K k, V v)`	associe la valeur `v` à la clé `k`
`boolean containsKey(K k)`	indique s'il existe une valeur associée à `k`
`V get(K k)`	renvoie la valeur associée à `k`, si elle existe, et `null` sinon

* *
*