COMPOSITION d 'INF411

en date du 2/11/15

1632 INF411

1)

Q1)
```
boolean equals(Cell c) {
    return (this.i == c.i && this.j == c.j);
}
int hashCode() {
    return (this.i * this.j);
}
```

Q2)
```
Grid (int h, int w) {
    this.adj = new HashMap<Cell, LinkedList<Cell>>;
    for (int k=0; k<h; k++) {
        this.addVertex(new Cell(k,0));
        for(int l=0; l<w; l++) {
            this.addEdge(new Cell(k,0), new Cell(k,l))
        }
    }
}
```

NOTE

N° 1/3

Q3)

```
static LinkedList<Cell> shuffle (LinkedList<Cell> l) {

    LinkedList<Cell> liste = new LinkedList<Cell>();
    LinkedList<Cell> recopie = l;

    while (!recopie.isEmpty()) {

        int r = Math.random()* recopie.length;

        liste.add(recopie.get(r));

        recopie.remove(r);

    }
```

Q4) `void dfs() {`

```
    Cell c = new Cell(0,0);
```

Q5) Ce parcours est au pire de complexité $h \times w$

Q6) On peut avoir, pour $w = 4$ et $h = 3$

Avec visited :

sommet	antécédent
0,0	null
1,0	0,0
1,1	1,0
2,1	1,1
2,2	2,1
2,3	2,2
1,3	2,3
1,2	1,3
0,2	1,2
0,3	0,2
0,1	0,2
3,0	3,1

Q7) Le labyrinthe est parfait car on aura fait attention dans dfs() de ne jamais pouvoir rejoindre une pièce déjà visitée

2)

Q8)

```
class Const extends Expr {
    int val;

    Const(int x) {
        this.val = x;
    }
    String toString() { return "" + this.val; }
}

class Add extends Expr {
    String operation = "+";
    Expr d, g;

    Add ( Expr a, Expr b) {
        this.g = a;
        this.d = b;
    }

    String toString() {
        return this.g + "+" + this.d;
    }
}

class Mul extends Expr {
    String operation = "*";
    Expr d, g;

    Mult (Expr a, Expr b) {
        this.g = a;
        this.d = b;
    }
    String toString() {
        return this.g + "x" + this.d;
    }
}
```

Q9) • Dans la classe Expr :

```
abstract int eval();
```

• Dans la classe Const :

```
int eval() {
    return this.val;
}
```

• Dans la classe Add :

```
int eval() {
    return this.g.eval() + this.d.eval();
}
```

• Dans la classe Mul :

```
int eval() {
    return this.g.eval() * this.d.eval();
}
```

NOTE

NE PAS ÉCRIRE DANS CE CADRE

Q10)

```
LinkedList<String> toRPN() {
    LinkedList<String> liste = new LinkedList<String>();
    if (this.g.g != null) {          // c'est à dire que la branche gauche n'est pas une const mais une opération.
        this.g.toRPN();
        liste.add(this.operation);   // à la fin, on ajoute l'opération
    }
    liste.add(this.g.toString());    // on ajoute la const
    if (this.d.d != null) {
        this.d.toRPN();
        liste.add(this.operation);
    }
    liste.add(this.d.toString());
}
```

Q11)

```
static Expr fromRPN(LinkedList<String> l) {
    Expr e;
    Stack<Expr> pile = new Stack<Expr>();
    while (!l.getFirst().equals("+") || !l.getFirst().equals("*")) {
        pile.push(l.poll());
    }
    if (l.getFirst().equals("+")) {
        e = new Add(pile.pop(), pile.pop());
    }
    if (l.getFirst().equals("*")) {
        e = new Mul(pile.pop(), pile.pop());
    }
```

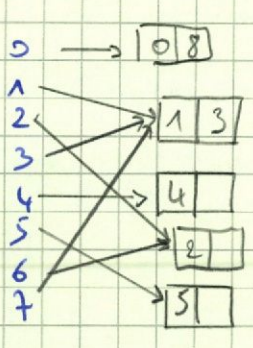
```
    return fromRPNrec(l, pile, e);      // Appel à une méthode récursive
}

Static Expr fromRPNrec(LinkedList<String> l, Stack<Expr> pile, Expr e) {
    while (!l.getFirst().equals("+") || !l.getFirst().equals("*")) {
        pile.push(l.pollFirst());
    }
    if(l.getFirst().equals("+")) {
        e = new Add(e, pile.pop());
    }
    if(l.getFirst().equals("*")) {
        e = new Mul(e, pile.pop());
    }
    if(l.isEmpty()) return e;
    return (fromRPN(l, pile, e));
}
```

NE PAS ÉCRIRE DANS CE CADRE

COMPOSITION d 'INF411

en date du 2/11/15

Q13)



Q14)

```
HashTable (int level) {
    this.level = level;
    this.buckets = new Vector<Bucket<E>>();
    for (int i = 0; i < power(level); i++) {
        buckets.add(new Bucket(level));
    }
}
```

Q15)

```
boolean contains (E x) {
    for (Bucket<E> b : this.buckets) {
        if (b.contains(x)) return true;
    }
    return false;
}
```

NOTE

N° 3/3

Q16/
```
void doubleSize() {
    HashTable h = new HashTable(this.level+1);
    for(int i = power2(level); i < power2(level+1); i++) {
        this.buckets.get(i).add(this.buckets.get(i - power2(level)));
    }
    level++;
}
```
recopie de la première moitié

Q17/
```
void reallyAdd(E x) {
    if(this.buckets.get(hashCode(x)).elements.size() == this.buckets.get(hashCode(x)).capacity)
        this.doubleSize();
    this.buckets.get(hashCode(x)).add(x);
}

void add(E x) {
    int h = hashCode(x);
    if(this.buckets.get(h).elements.size() < capacity) {
        this.buckets.get(h).add(x);
        return;
    }
```

NE PAS L

```
if(!(this.buckets.get(h).level < this.level)){
    this.doubleSize();
}
```

NE PAS ÉCRIRE DANS CE CADRE