Composition d'Informatique
Mécanismes de la Programmation Orientée-Objet (INF371)

Promotion 2018

1^{er} juillet 2019

Les parties sont indépendantes entre elles.

On accordera beaucoup d'importance à la clarté des réponses, ainsi qu'à la clarté et la concision du code.

1 Exercices

Les questions sont indépendantes entre elles (sauf les 4 et 5).

Question 1 Soit la séquence d'instructions suivante :

(1) PUSH(9)
(2) ☐
(3) GTO(1)
(4) STOP

a) Proposez une instruction à mettre dans l'emplacement vide pour que le programme boucle indéfiniment.

b) Proposez une instruction à mettre dans l'emplacement vide pour que le programme échoue sur un débordement de pile.

c) Proposez une instruction à mettre dans l'emplacement vide pour que le programme termine. ◇

Solution. a) POP

b) PUSH(2)

c) GTO(4) □

Question 2 On peut considérer qu'une opération arithmétique comme ADD, MUL... prend plus de temps qu'on opération élémentaire comme PUSH, POP.... Proposez une séquence d'opérations équivalente à PUSH(0) ; MUL mais plus efficace. ◇

Solution. Cette séquence remplace la valeur en haut de la pile par 0. On peut donc prendre POP ; PUSH(0). □

Question 3 Soit une classe `E` munie de la méthode suivante :

```
public void f() {
    while (true) f();
}
```

Que se passe-t-il lorsqu'on invoque la méthode par `e.f()` quand `e` est une instance de `E` ? ◇

Solution. La pile déborde. En effet, l'appel à `f()` est récursif et donc il y a une suite infinie d'appels récursifs qui empêche d'exécuter le corps de la boucle. □

Question 4 Proposez le code source d'une fonction statique dont le code compilé est :

```
RFR(-1)
PUSH(1)
CREAD
PUSH(2)
ADD
PXR
RET
```

◇

Question 5 Proposez un autre code source, cette fois d'une méthode (au lieu d'une fonction statique), qui donnerait le même code compilé. ◇

Solution. Le code récupère la valeur du premier champ de l'argument, ajoute 2 puis rend le résultat. On peut considérer que c'est une méthode avec l'argument `this` ou une fonction statique. Les deux donnent le code demandé :

```
static int f(E e) { return(e.x + 2); }

int f() { return (this.x + 2); }
```

Question 6 Soit la classe suivante définissant des points dans le plan :

```
class Point {
  float x;    float y;
  Point(float x, float y) { this.x = x;    this.y = y; }
}
```

Un collègue propose de définir la fonction suivante :

```
static Point copyAndTranslate(Point p) {
  Point q = p;
  q.x = q.x + 10;
  return q;
}
```

Ce code vous parait-il criticable ? Pourquoi ? (réponse attendue, quelques lignes)

Proposez une variante qui correspondrait mieux au nom de la fonction. ◇

Solution. Cette fonction translate l'argument mais ne le copie pas. Soit il faut créer un nouveau point `q` avec le constructeur, ou sinon il ne sert à rien d'utiliser une nouvelle variable locale `q`.

Une correction serait donc :

```
static Point copyAndTranslate(Point p) {
  Point q = new Point(p.x, p.y);
  q.x = q.x + 10;
  return q;
}
```

Question 7 On rappelle que lorsque `e` est un objet de classe `E` et `f()` une fonction statique de `E`, alors `e.f()` est complètement équivalent à `E.f()` (c'est-à-dire que le code compilé est le même).

Un programmeur a l'habitude d'utiliser systématiquement cette syntaxe ; c'est-à-dire qu'il écrit toujours `e.f()` et pas `E.f()`. Une fois, il oublie le mot clé `static` devant la définition d'une fonction ; il écrit donc :

```
public int f() { ... }
```

au lieu de :

```
public static int f() { ... }
```

Est-ce-que le temps d'exécution de `e.f()` peut en être affecté ? Pourquoi ? ◇

Solution. La fonction devient une méthode, donc elle prend un argument `this` en plus. A chaque appel de la fonction, il faut empiler un argument, puis le dépiler à la fin. Le temps d'exécution est donc, même très légèrement, plus long. □

Question 8 Soit la classe suivante :

```
class NoField {
   public void g() { System.out.println("G"); }
   public void h() { System.out.println("H"); }
}
```

Combien de mots mémoire sont réservés dans le tas chaque fois qu'on appelle le constructeur `new NoField()` ? ◇

Solution. Il n'y a pas de champ, donc il y a juste un mot mémoire (qui pointe vers la table d'adressage des méthodes). □

Question 9 On considère la classe suivante :

```
class Sing {
    private Sing() {} // le constructeur peut etre private
    private static Sing sing = new Sing();

    public static Sing export() { return sing; }
}
```

On remarque qu'on utilise la possibilité de rendre privé le constructeur de la classe.

Que peut-on dire des objets de classe `Sing` ? ◇

Solution. On ne peut pas appeler le constructeur depuis l'extérieur de la classe, donc la variable statique `sing` est le seul objet construit et on peut y accéder par la fonction statique `export()`.

Il y a donc une seule valeur que peut prendre un objet de classe `Sing` (en plus de `null`). □

2 Cycles et sloops

On considère des listes chaînées, définies par le code suivant (qui est tout à fait classique). La liste vide est représentée par la valeur `null`.

```
class Clist<E> {
    E cont;
    Clist<E> tail;

    Clist(E e, Clist<E> l){ cont = e; tail = l; }
}
```

Les objets de cette classe peuvent comporter un cycle ou pas ; voir la figure 1.

Question 10 Donnez une suite d'instructions qui construit la liste avec cycle de la figure 1. ◇

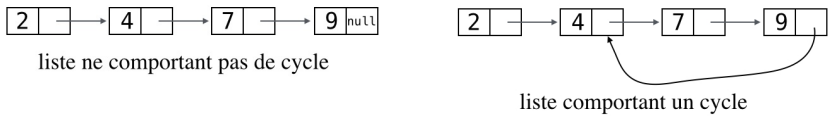


FIGURE 1 – Schémas mémoire de deux listes, avec et sans cycle.

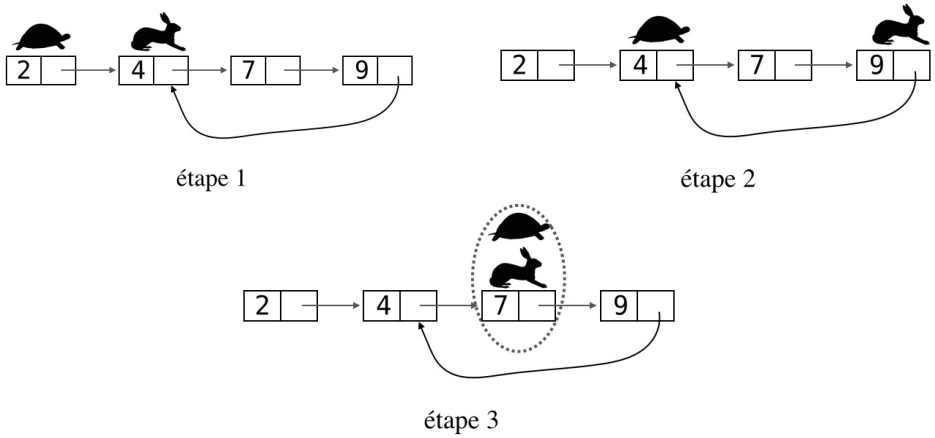


FIGURE 2 – Les étapes de l'algorithme du lièvre et de la tortue sur un exemple simple.

Solution. Par exemple :

```
Clist<Integer> a = new Clist(7, new Clist(9,null));
a = new Clist(2, new Clist(4, a));
a.tail.tail.tail = a.tail;
```

Etant donné un objet de la classe `Clist`, on veut maintenant détecter si cette liste comporte un cycle ou pas. Votre professeur propose l'algorithme suivant :

- On place deux pointeurs qui vont parcourir la liste. Le pointeur `hare` (lièvre en anglais) qui va avancer vite, et le pointeur `tortoise` (tortue) qui va avancer lentement. La tortue part du premier élément de la liste, le lièvre du deuxième élément.
- A chaque tour, `hare` avance de deux éléments dans la liste et `tortoise` avance d'un élément.
- On continue à faire avancer ces deux pointeurs jusqu'à ce que l'un des deux évènements suivants se produise :
  - on tombe sur un élément `null` ; dans ce cas la liste ne comporte pas de cycle.
  - `hare` et `tortoise` se trouvent sur le même élément ; dans ce cas la liste comporte un cycle.

La figure 2 illustre le déroulement de l'algorithme dans le cas de la liste avec cycle de l'exemple.

Question 11 Prouvez que cet algorithme termine. Quelle est sa complexité en temps ?

Prouvez sa correction. ◇

Solution. Si la liste ne contient pas de cycle, le champ `tail` de son dernier élément est `null` et le programme termine lorsque le lièvre atteint cet élément.

Si la liste contient un cycle, il y a deux phases :

- Tant que la tortue n'a pas atteint le cycle, elle se rapproche du cycle d'une case à chaque tour.
- Une fois que la tortue a atteint le cycle, le lièvre est également dans le cycle. A chaque tour il se rapproche de la tortue d'une case. Il finit donc par la rattrapper.

L'algorithme termine donc toujours en un temps proportionnel au nombre d'éléments de la liste.

Par ailleurs, le programme rend bien `false` si et seulement si la liste contient `null`, c'est-à-dire qu'il n'y a pas de cycle. Comme il termine, il rend `true` lorsqu'il y a un cycle. □

Question 12 A-t-on besoin de savoir tester l'égalité structurelle sur les instances de `E` pour implémenter cet algorithme ? ◇

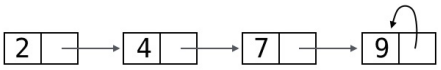
Solution. Non, la question porte sur les pointeurs et pa sur le contenu des listes (cf. solution question suivante). □

Question 13 Ecrivez une fonction `boolean cycleP(Clist<E> l)` qui implémente cet algorithme en rendant `true` si `l` comporte un cycle et `false` sinon. ◇

Solution. On implémente l'algorithme précédent.

```
boolean cycleP(Clist<E> l) {
    if (l == null) return(false);
    Clist turtle = l;
    Clist hare = l.tail;
    while (turtle != hare) {
        if (hare.tail == null || hare.tail.tail == null) return(false);
        turtle = turtle.tail;
        hare = hare.tail.tail;
    }
    return(true);
}
```

On veut maintenant, tout en gardant la classe `Clist`, utiliser un codage un peu différent des listes : on repère le dernier élément en le faisant pointer sur lui-même (voir ci-dessous). On appelle une telle liste un *sloop*.



un *sloop*

Question 14 Ecrivez une fonction une fonction `boolean sloopP(Clist<E> l)` qui rend `true` si et seulement si son argument est un *sloop* bien-formé (c'est-à-dire ne contient pas `null` et pas d'autre cycle). ◇

Solution. Il suffit d'adapter un peu la fonction précédente. S'il n'y a pas de cycle, ce n'est pas un sloop. S'il y a un cycle, le lièvre et la tortue se rencontrent sur un sommet du cycle. Il suffit alors de vérifier si ce sommet point sur lui-même.

```
static boolean sloopP(Sloop<E> s) {
    Sloop<E> tortue = s;
    if (s.tail == null) return(false);
    Sloop<E> lievre = s.tail;
    while(lievre != tortue) {
        if (lievre.tail == null || lievre.tail.tail == null) return(false);
        lievre = lievre.tail.tail;
        tortue = tortue.tail;
    }
    return(tortue.tail == tortue);
}
```

3 Files et coupes-files

On va, dans cette partie, implémenter des files d'attente avec des variantes, en utilisant, en interne, des piles.

Une classe `File<E>` doit représenter une file d'attente (first-in-first-out) dont les éléments sont de classe `E`.

Les méthodes publiques de la classe doivent être :

— `void push(E e)` qui ajoute l'élément `e` en queue de file.
— `E pop()` qui rend l'élément en tête de la file (et le fait sortir de la file). Cette méthode doit déclencher une `Error` si la file est vide.
— `boolean empty()` qui rend `true` si la file est vide et `false` sinon.
— Un constructeur `File<E>()` qui créé une file vide.

Question 15 Complétez le code suivant pour que la classe `File<E>` corresponde à la spécification ci-dessus.

On vous demande, dans cette question, de ne pas utiliser de champ supplémentaire, donc uniquement les deux piles `entree` *et* `sortie`.

On rappelle, en appendice, les méthodes principales de la classe `Stack<E>` de la bibliothèque Java.

```
class File<E> {
  private Stack<E> entree;
  private Stack<E> sortie;

  File() {
    entree = new Stack<E>();
    sortie = new stack<E>();
  }

  public void push(E e) { entree.push(e); }

  ...
}
```

Solution.

```
    File() {
        private entree = new Stack<E>();
        private sortie = new Stack<E>();
    }

    public void push(E e) {
        entree.push(e);
    }

    public E pop() {
        if (sortie.isEmpty()) {
            while (!entree.isEmpty()) sortie.push(entree.pop());
            if (sortie.isEmpty()) throw new Error("pop");
        }
        return(sortie.pop());
    }

    public boolean empty() {
        return(entree.isEmpty() && sortie.isEmpty());
    }
}
```

Question 16 Que pouvez-vous dire de la complexité en temps des méthodes `push` et `pop` ? ◇

Solution. Dans le pire cas, la méthode `push` est en temps constant, et la méthode `pop` est en temps proportionnel à la taille courante de la file. En moyenne, les deux méthodes sont en temps constant. □

Question 17 Ajoutez à votre classe une méthode `int length()` qui rend la longueur courante de la file. Quelle est sa complexité ?

(On suppose de la méthode `length()` de `Stack` est en temps linéaire par rapport au nombre d'éléments de la pile.) ◇

Solution.
```
public int length() { return(entree.length() + sortie.length(); }
```

Le temps de calcul est proportionnel à la taille courante de la file. □

Question 18 Comment procéderiez-vous pour avoir une méthode `length()` qui rende le résultat en temps constant ? Y a-t-il un rapport avec l'encapsulation ? (soyez précis et bref) ◇

Solution. On ajoute un champ privé `int length` qui contient la taille de la file et qui est incrémenté (resp. décrémenté) de 1 à chaque appel de `push` (resp. `pop`). On peut alors écrire :

```
public int length() { return length; }
```

qui est en temps constant. □

On va maintenant traiter un problème un peu plus complexe. On considère les classes équipées d'une méthode indiquant si un élément est prioritaire :

```
abstract class Tagged {
  abstract boolean priority();
}
```

Un élément `e` d'une sous-classe de `Tagged` est prioritaire si `e.priority()` vaut `true` (on dira alors qu'il dispose d'un *coupe-file*).

On va chercher à définir une classe de file d'attente de telles classes :

```
class CoupeFile<E extends Tagged> {
  CoupeFile() { ... }
  public void push(E e) { ... }
  public E pop() { ... }
  public E cpop() { ... }
  public boolean empty() { ... }
}
```

Le comportement attendu est :
- Le constructeur créé une file vide.
- La méthode `push(E e)` ajoute l'élément `e` en queue de file.
- La méthode `pop()` fait sortir le premier élément de la file, indépendamment de s'il possède un coupe-file ou pas (donc se comporte comme la méthode `pop()` des questions précédentes).
- La méthode `cpop()` va faire sortir le premier élément de la file possédant un coupe-file. Les positions des autres éléments de la file restent inchangées. Si jamais il n'y a pas d'élément possédant un coup-file, alors `cpop()` se comporte comme `pop()`.
- La méthode `empty()` indique si la file est vide, comme dans les questions précédentes.

Par exemple, si les `String` commençant par une majuscule possèdent un coupe-file, et que l'on effectue les opérations suivantes :

```
f.push("a");
f.push("b");
f.push("C");
f.push("D");
f.push("e");
```

les opérations suivantes, dans cet ordre, donneront ces chaînes de caractère :

```
f.pop()        "a"
f.cpop()       "C"
f.pop()        "b"
f.pop()        "D"
f.cpop()       "e"
```

Question 19 Proposez une implémentation de la classe `CoupeFile` en complémentant les implémentations des méthodes et des constructeurs, et ajoutant les champs nécessaires.

On vous demande de proposer une implémentation aussi claire, lisible et concise que possible.

Indication : vous pouvez utiliser la classe `File<E>` définie précédemment. ◇

Solution.

```
public class CoupeFile<E extends Tagged> {
    private File<E> main;
    private File<E> passed;

    CoupeFile() {
        main = new File<E>();
        passed = new File<E>();

    }

    public void push(E e) {
        main.push(e);
    }

    public E pop() {
        if (!passed.empty()) return passed.pop();
        return main.pop();
    }

    public E cpop() {
        while (!main.empty()) {
            E e = main.pop();
            if (e.priority()) return e;
            passed.push(e);
        }
        return pop();
    }
```

Question 20 Quelles sont les complexités en temps des méthodes de votre implémentation ? ◇

Solution. Les méthodes sont en temps constant en moyenne. □

Question 21 Dans cette question, on ne s'intéresse plus aux coupes-files. En revanche, on se donne l'interface suivante, qui encapsule une fonction d'une classe `E` vers une classe `F` :

```
    interface MapEF<E, F> {
    F f(E e);
}
```

On donne le début de l'implémentation d'une classe :

```
class MapFile<E, F> {
  MapEF<E, F> map;
  MapFile(MapEF<E, F> m) {
    map = m;
    ... }
  ...
}
```

Complétez cette définition pour qu'une instance de `MapFile<E, F>` soit une file d'attente, avec les méthodes suivantes :

- `void push(E e)` qui fait entrer un objet de classe `E` dans la file,
- `F pop()` qui fait sortir le premier élément de la file, mais après qu'il ait été transformé en objet de classe `F` par la fonction du champ `map`.
- la méthode `boolean empty()` usuelle.

Pensez à compléter (aussi) la définition du constructeur. ◇

Solution. On garde une implémentation des files suivant la même architecture, mais on applique la fonction soit à l'entrée, soit la sortie, soit entre les deux piles.

Par exemple :

```
    interface MapEF<E, F> {
    F f(E e);
}

class MapFile<E, F> {
    MapEF<E, F> map;
    Stack<E> entree;
    Stack<F> sortie;

    MapFile(MapEF<E, F> map) { this.map = map;
        entree = new Stack<E>();
        sortie = new Stack<F>();
    }

    public void push(E e) { entree.push(e); }

    public F pop() {
        if (sortie.empty()) flush();
        if (sortie.empty()) throw new Error();
        return (sortie.pop());
    }

    private void flush() {
        while (!entree.empty())
            sortie.push(map.f(entree.pop()));
    }

    public boolean empty() {
        return(entree.empty() && sortie.empty());
    }
```

Annexe : `Stack`

La classe générique `Stack<E>` fournie par la bibliothèque Java est munie d'un constructeur `Stack<E>()` qui créé une pile vide et des méthodes publiques suivantes :

— `void push(E e)` qui ajoute `e` en haut de la pile.
— `E pop()` qui fait sortir l'élément en haut de la pile et le rend comme résultat.
— `boolean empty()` qui indique si la pile est vide.
— `int length()` qui calcule la taille courante de la pile.

On considèrera que les méthodes sont toutes exécutés en temps constant, sauf `length` *qui prend un temps proportionnel à la hauteur de la pile.*