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. ◇

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. ◇

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`? ◇

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. ◇

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. ◇

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 ? ◇

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()` ? ◇

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` ? ◇

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`**.

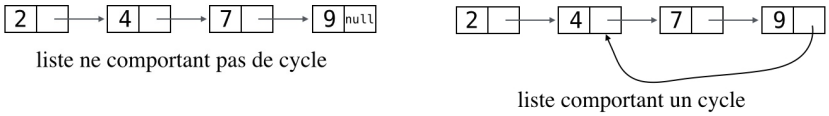


FIGURE 1 – Schémas mémoire de deux listes, avec et sans cycle.

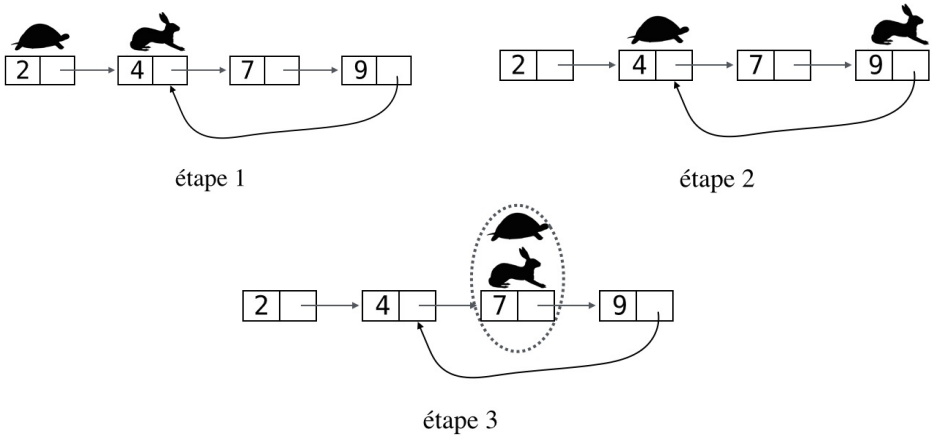


FIGURE 2 – Les étapes de l'algorithme du lièvre et de la tortue sur un exemple simple.

```
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. ◇

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.

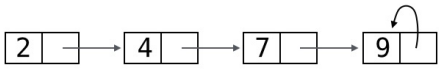
Question 11 Prouvez que cet algorithme termine. Quelle est sa complexité en temps ?

Prouvez sa correction. ◇

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

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. ◇

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). ◇

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); }

  ...
}
```

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

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.) ◇

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) ◇

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. ◇

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

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. ◇

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.*