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. ◇

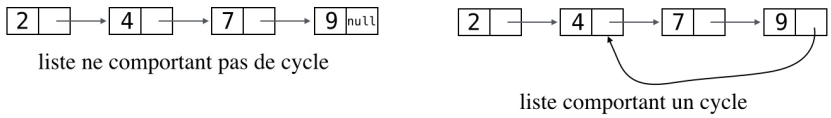


FIGURE 1 – Schémas mémoire de deux listes, avec et sans cycle.

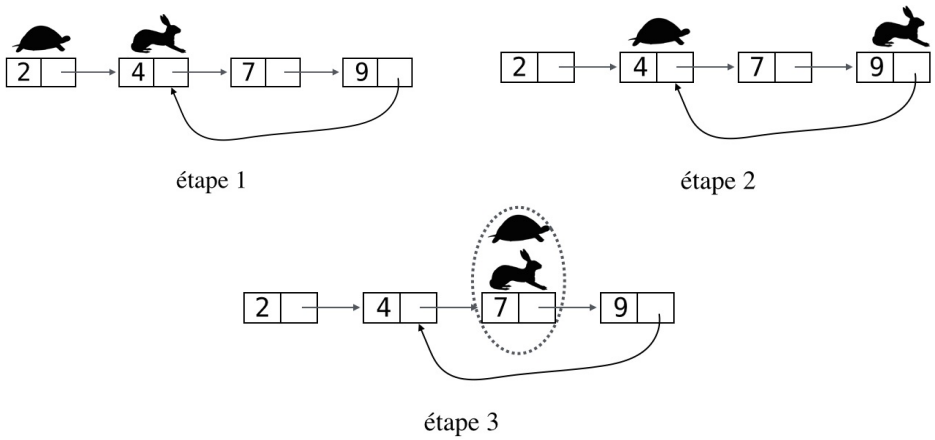


FIGURE 2 – Les étapes de l'algorithme du lièvre et de la tortue sur un exemple simple.

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 ? ◇

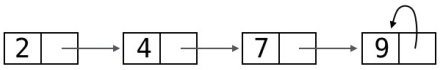
Solution. Non, la question porte sur les pointeurs et pa sur le contenu des listes (cf. solution question suivante). □

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.*

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

Promotion 2020

28 juin 2021

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

1 Exercices

Question 1 Soit la séquence d'instructions suivante :

(1) PUSH(9)
(2) PUSH(1)
(3) ADD
(4) PUSH(0)
(5) ☐
(6) GTO(2)
(7) 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.

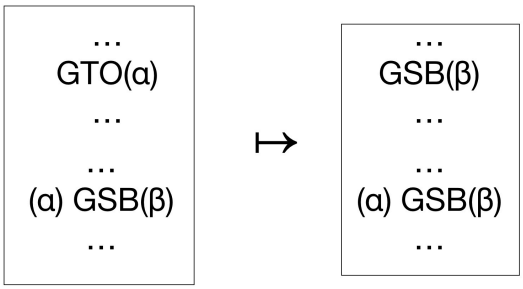
c) Proposez une instruction à mettre dans l'emplacement vide pour que le programme termine. ◇

Solution. a) POP ou ADD. . .

b) PUSH(0). . .

C) GTO(7) □

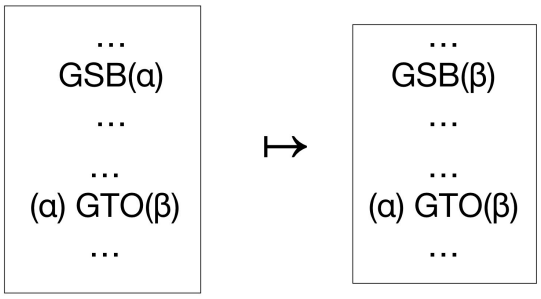
Question 2 Un collègue vous propose l'optimisation suivante sur du code machine (remplacer un GTO par un GSB quand on est dans la situation de gauche).



Cette optimisation vous semble-t-elle correcte ? Sinon pourquoi ? ◇

Solution. Elle est fausse (on sera au mauvais endroit après le RET). □

Question 3 Même question pour cette optimisation (remplacer un GSB(α) par un GSB(β) quand on est dans la situation de gauche).



Solution. Elle est correcte. □

Question 4 Soit la classe suivante :

```
class A {
  static int x = 0;
  int a, b;
  String s;
  A(int a, int b, String s) {
    this.a = a; this.b = b; this.s = s;
  }
  int f() { return (a + b); }
}
```

Quand on exécute l'instruction suivante, quelle est le nombre de mots mémoire de la séquence vers laquelle pointe la variable v : `A v = new A(2, 3, "a");` ?

Et après `A v = new A(2, 3, "aaaa");` ? ◇

Solution. Quatre mots dans les deux cas (trois champs et l'adresse de la table d'adressage). □

2 Expressions booléennes

Dans cette partie on sera amené à utiliser quelques classes de la bibliothèque de Java dont les caractéristiques principales sont rappelées en appendice à la fin du sujet.

On va représenter et manipuler des expressions booléennes. Ces expressions peuvent être :
— Soit les constantes `true` et `false`,
— soit des variables, qu'on notera `x`, `y`, `z`...
— soit construites avec les connecteurs $\wedge$ (et), $\vee$ (ou) et $\neg$ (non).
Des exemples d'expressions sont donc : ($\neg$`x` $\vee$ `y`), (`x` $\wedge$ ($\neg$`y` $\vee$ `z`)), $\neg$(*x* $\vee$ `false`) *etc...*

On va représenter ces expressions de manière semblable à ce qu'on a vu en cours. Une expression sera représentée par un objet de classe `Expr`.
— `Expr` sera une classe abstraite,
— on aura des sous-classes concrètes `TrueExpr`, `FalseExpr` (pour les constantes), `VarExpr` (pour les variables), `AndExpr` (pour la conjonction), `OrExpr` (pour la disjonction), `NotExpr` (pour la négation),
— pour simplifier, on considèrera que les variables sont identifiées par un `int`.
Voici un début d'implémentation :

```
abstract class Expr {  }
class TrueExpr extends Expr {  }
class FalseExpr extends Expr {  }
class VarExpr extends Expr {
    int n;  // l'identifiant de la variable
    VarExpr(int n) { this.n = n; }
}
class AndExpr extends Expr { ... }   // a completer
```

Question 5 Complétez ce qui précède en donnant les définitions des classes `AndExpr` et `NotExpr` (pas la peine d'écrire la définition de `OrExpr` qui est très similaire à `AndExpr`). ◇

Solution.
```
class AndExpr extends Expr {
    Expr left;
    Expr right;
    AndExpr(Expr a, Expr b) { left = a; right = b; }
}

class NotExpr extends Expr {
    Expr e;
    NotExpr(Expr e) { this.e = e; }
}
```

Question 6 Equipez les sous-classes concrètes de `Expr` d'une méthode `public String toString()` qui affiche l'expression sous forme infixe. Les variables pourront être affichées simplement par leur numéro. Vous pouvez écrire `V` et `F` pour les constantes, et, respectivement, `ET`, `OU` et `NON` pour les connecteurs et vous n'êtes pas obligés d'optimiser le nombre de parenthèses.

Par exemple $x \vee \neg y$ peut être affiché `(1 OU (NON 2))` (si les variables `x` et `y` portent les numéros 1 et 2). ◇

Solution. Pas la peine de la mentionner dans `Expr` puisque c'est hérité de `Object`.

Par exemple, respectivement pour `VarExpr`, `OrExpr` `AndExpr`et `NotExpr` :

```
public String toString() { return(""+ n); }
public String toString() {
  return( "(" + left.toString() + " OU " + right.toString() + ")" ); }
public String toString() {
  return( "(" + left.toString() + " ET " + right.toString() + ")" ); }
public String toString() {
  return("(NON" + e.toString() + ")"); }
```

Valuations et tautologies

Une valuation est une fonction qui donne une valeur de vérité (de type `bool`) à chaque variable. On représente cela en Java comme un objet implémentant l'interface suivante :

```
interface Valuation { boolean value(int n); }
```

Etant donnée une valuation, toute expression booléenne a elle même une valeur de vérité. Par exemple considérons une valuation I, telle que $I(x) = I(y) =$ `true` et $I(z) =$ `false`. Alors, pour I, l'expression $x \wedge (y \vee z)$ vaudra `true` et $(x \wedge z) \vee (y \wedge z)$ vaudra `false`.

On ajoute à `Expr` une méthode abstraite pour calculer la valeur d'une expression :

```
abstract boolean eval(Valuation v);
```

Question 7 Implémentez cette méthode dans les sous-classes concrètes de `Expr`. ◇

Solution. Pour `TrueExpr` :

```
boolean eval(Valuation v) { return true; }
```

Pour `FalseExpr` :

```
boolean eval(Valuation v) { return false; }
```

Pour `VarExpr` :

```
boolean eval(Valuation v) { return (v.eval(n)); }
```

Pour `AndExpr` :

```
boolean eval(Valuation v) { return (left.eval(v) && right.eval(v)); }
```

Pour `OrExpr` :

```
boolean eval(Valuation v) { return (left.eval(v) || right.eval(v)); }
```

Pour `NotExpr` :

```
boolean eval(Valuation v) { return (!e.eval(v)); }
```

Une *tautologie* est une expression dont la valeur vaut `true` pour n'importe quelle valuation. Par exemple $x \vee \neg x$ ou $y \vee$ `true` sont des tautologies. En revanche `true` $\wedge\, y$ n'en est pas une.

Question 8 Donnez deux autres exemples de tautologies. ◇

Solution. Par exemple `true`, `true` $\vee$ `false`, $\neg$`false`... □

Question 9 On équipe la classe `Expr` d'une méthode pour ajouter à un `HashSet` de variables toutes les variables d'une expression (en fait les nombres correspondants aux variables).

```
void vars(HashSet<Integer> ev) { }
```

Cette définition par défaut doit être réécrite pour les sous-classes de `Expr` *susceptibles de contenir des variables.*

Par exemple si on invoque `e.vars(ev)` pour `x` $\vee$ `y`, alors `x` et `y` doivent être ajoutées à `ev`.

Réécrivez cette méthode dans les classes concernées pour que ce soit le cas. ◇

Solution. Il faut réécrire la méthode pour les trois classes contenant potentiellement des variables.

Pour `VarExpr` :

```
void vars(HashSet<Integer> ev) { ev.add(n) }
```

Pour `AndExpr` et `OrExpr` :

```
void vars(HashSet<Integer> ev) {
  left.vars(ev);
  right.vars(ev);
}
```

Pour `NotExpr` :

```
void vars(HashSet<Integer> ev) { e.vars(ev); }
```

Question 10 Utilisez la méthode précédente pour ajouter à `Expr` une méthode qui rend sous forme de `LinkedList<Integer>` l'ensemble des variables d'une expression :

```
LinkedList<Integer> lvars() { ... }
```

On pourra utiliser les caractéristiques des classes `HashSet<E>` et `LinkedList<E>` qui sont rappelées à la fin de l'énoncé. ◇

Solution. On peut l'ajouter directement à `Expr` sans toucher aux sous-classes :

```
LinkedList<Integer> lvars() {
  LinkedList<Integer> r = new LinkedList<Integer>();
  HashSet<Integer> h = new HashSet<Integer>();
  vars(h);
  r.addAll(h);
  return(r);
}
```

Question 11 Ecrivez une fonction qui étant donnée une valuation `v`, une variable `x` et un booléen `b` construit la valuation qui vaut `b` en `x` et est identique à `v` ailleurs.

```
static Valuation subst(Valuation v , int x, boolean b) { ... }
```

Solution.

```
static Valuation subst(Valuation v , int x, boolean b) {
  Valuation nv =
       m -> {if (m == x) return(b); return(v.value(m)); };
  return nv;
}
```

Question 12 Sans chercher à être efficace, écrivez une fonction statique qui teste si une expression est une tautologie.

```
static boolean tauto(Expr e) { ... }
```

Que pouvez-vous dire de sa complexité ? ◇

Solution. On se donne une fonction auxiliaire récursive pour tester toutes la valeurs sur chaque variable. La complexité est évidemment exponentielle par rapport au nombre de variables.

```
static boolean tauto(Expr e) {
  LinkedList<Integer> lv = e.lvars();
  Valuation v = (n -> false);
  return(aux(e, lv, v));
}

static boolean aux(Expr e, LinkedList<Integer> lv, Valuation v) {
  if (lv.isEmpty()) return (e.eval(v));
  int n = lv.remove();
        return(aux(e, lv, Gen.setTrue(n, v))&&aux(e, lv, Gen.setFalse(n, v)));
      }
```

BDDs

On se donne maintenant une autre manière de représenter les expressions booléennes : les diagrammes de décisions booléens ou BDDs (pour *binary decision diagrams*). Un BDD est soit :

— une constante booléenne (`true` ou `false`),

— soit un nœud `IfBDD`(x, l, r) où x est une variable propositionnelle et l et r sont eux-mêmes chacun un BDD.

La valeur d'un BDD de la forme `IfBDD`(x, l, r) est définie ainsi : si x vaut `true`, alors `IfBDD`(x, l, r) vaut la même valeur que l, si x vaut `false`, alors `IfBDD`(x, l, r) vaut la même valeur que r.

Question 13 On se donne une classe abstraite `BDD` :

```
abstract class BDD {
  abstract boolean value(Valuation v);
}
```

Complétez en proposant trois sous-classes concrètes de `BDD` : `TrueBDD`, `FalseBDD` et `IfBDD`. ◇

Solution.

```
class FalseBDD extends BDD {
  boolean value(Valuation v) { return false; }
}
class TrueBDD extends BDD {
  boolean value(Valuation v) { return true; }
}

class IfBDD extends BDD {
  int var;
  BDD trueCase;
  BDD falseCase;
  IfBDD(int n, BDD t, BDD f) {
    var = n;
    trueCase = t;
    falseCase = f;
  }
  boolean value(Valuation v) {
    if (v.value(n)) return trueCase.value(v);
    return falseCase.value(v);
  }
}
```

On peut traduire une `Expr` en `BDD` en utilisant le principe de l'expansion de Shannon : étant donnés une expression `e` et une variable `x` apparaissant dans `e` on peut remarquer que pour toute valuation `v` on a :

— `e.eval(v)` égal à `e.eval(subst(v, x, true))` lorsque `v.value(x)` vaut **`true`**.

— `e.eval(v)` égal à `e.eval(subst(v, x, false))` lorsque `v.value(x)` vaut **`false`**.

Question 14 En partant de la remarque précédente, écrivez une fonction `translate` pour traduire une expression en BDD (ce peut être une fonction statique ou une méthode, à votre guise).

Indication : on pourra également utiliser la méthode `lvars()`. ◇

Solution. On teste sur chaque variable pour faire un grand peigne (de 2^n feuilles s'il y a n variables).

```
static BDD translate(Expr e) {
  LinkedList<Integer> lv = e.lvars(lv);
  Valuation v = (n -> false); // en fait on pourrait prendre
```

```
  // n'importe quelle valuation
  return(traux(e, lv, v));
}

static BDD traux(Expr e, LinkedList<Integer> lv, Valuation v) {
  if (lv.isEmpty())
            if (e.eval(v)) return new TrueBDD();
            else return new FalseBDD();
  int n = lv.remove();
  return(new IfBDD(n, traux(e, lv, subst(v, n, true)),
                                   traux(e, lv, subst(v, n, false))));
          }
```

On va maintenant essayer d'optimiser la représentation sous forme de BDD.

Question 15 Equipez les sous-classes de `BDD` d'une méthode de hachage `int hashCode()` correcte. On pourra prendre la fonction définie par :

$$\begin{aligned} h(\texttt{true}) &= 3 \qquad h(\texttt{false}) = 1 \\ h(\texttt{IfBDD}(x,l,r)) &= x + 31(h(l) + 31h(r)) \end{aligned}$$ ◇

Solution. Pour `FalseBDD`, `TrueBDD` et `IfBDD` respectivement :

```
public int hashCode() { return 1; }
public int hashCode() { return 3; }
   public int hashCode() {
        return(var+31*(trueCase.hashCode() + 31 * falseCase.hashCode()));
    }
```

Question 16 Implémentez la méthode `public boolean equals(Object o)` des sous-classes de `BDD` pour qu'elle teste correctement l'égalité structurelle entre `BDD`. ◇

Solution. Respectivement :

```
public boolean equals(Object o) {
  return(o instanceof FalseBDD);
}

public boolean equals(Object o) {
  return(o instanceof TrueBDD);
}

public boolean equals(Object o) {
  return((o instanceof IfBDD) &&
              ((IfBDD)o).trueCase.equals(trueCase) &&
              ((IfBDD)o).falseCase.equals(falseCase));
            }
```

On veut éviter d'avoir en mémoire plusieurs copies d'un même BDD. Pour cela on va maintenir un cache.

On se donne une table d'association entre `BDD` :

```
static HashMap<BDD, BDD> cache = new HashMap<BDD, BDD>();
```

L'idée est la suivante :
— `cache` ne contiendra que des associations entre un `BDD` et lui-même.

— Lorsqu'on construira un nouveau `BDD`, on regardera dans `cache` si on a déjà une copie structurellement égale à ce `BDD`. Dans ce cas, on utilisera la version du cache. Sinon on ajoutera ce nouveau `BDD` au cache.

Question 17 Ecrivez une fonction `static BDD retrieve(BDD e)` qui se comporte ainsi :
— Si un BDD structurellement égal à `e` est élément du cache, alors on renvoie cet élément du cache.
— Sinon on renvoie `e` après avoir ajouté `e` au cache. ◇

Solution.

```
static BDD retrieve (BDD b) {
  BDD r = cache.get(b);
  if (r != null) return r;
  cache.put(b, b);
  return b;
  }
```

On peut remarquer que chaque élément du cache n'apparait qu'une fois (la clé et la valeur sont physiquement égales). □

Un BDD est dit normalisé s'il vérifie les deux propriétés suivantes :

1. Si deux sous-parties du BDD sont identiques, elles sont partagées en mémoire.
2. Il n'y a pas de `IfBDD`(x, l, r) dont les deux branches l et r sont identiques.

On peut comprendre la deuxième clause comme : une (sous-)expression `IfBDD`(x, e, e) peut être remplacée par l'expression e.

Question 18 Ecrivez une fonction qui normalise, ou essaye de normaliser au maximum, un BDD :

```
static BDD normalize(BDD e) { .. }
```

On pourra utiliser la variable globale `cache` et la fonction de la question précédente. ◇

Solution. Une possibilité est de commencer à ajouter une méthode `void norm()` à `BDD`.

```
    // Pour TrueBDD et FalseBDD :
BDD norm() { return(Test.cache(this)); }

// pour IfBDD :
BDD norm() {
  BDD nl = Test.cache(trueCase.norm());
  BDD nr = Test.cache(falseCase.norm());
  if (nr.equals(nl)) return nr;
  return(Test.cache(new IfBDD(var, nl, nr)));
}

// et la fonction de normalisation :
static BDD normalize(BDD b) {
  BDD d = b.norm();
  if (b.equals(d)) return b;
  return(normalize(d));
}
```

Ici on itère `norm` jusqu'à ça ne bouge plus. On peut réfléchir à si c'est vraiment nécessaire ; je suis preneur d'arguments précis si vous en avez. On sera libéral dans la correction sur ce point de toute façon. □

Question 19 Un collègue regarde le code et fait la remarque qu'il est utile, dans ce cas, de commencer le test d'égalité structurelle sur `IfBDD` (cad. la méthode `equals`) par un test d'égalité physique.

Pouvez-vous deviner l'argument du collègue ? (brièvement)

Ecrivez-la méthode `equals(Object o)` ainsi modifiée. ◇

Solution. A cause de l'utilisation du cache, il y a beaucoup de chances que deux `BDD`s structurellement égaux soient aussi physiquement égaux (car pointant vers le même objet du cache). Cela vaut donc la peine de commencer par vérifier s'ils sont physiquement égaux, auquel cas ils seront aussi structurellement égaux.

Sinon on fait le test classique. On peut se poser la question de si le test d'égalité physique suffit, mais c'est au mieux compliqué.

```
public boolean equals(Object o) {
  if (o == this) return true;
  // puis le code deja donne
```

Question 20 On prend une `Expr e` qui est une tautologie. On traduit cette expression en BDD avec la fonction `translate` de la question 14, puis on normalise le résultat avec `normalize` (question 18). Que peut-on dire du résultat ? Justifiez brièvement. ◇

Solution. Ce sera forcément un élément de `TrueBDD` (en fait l'unique copie de `TrueBDD` dans `cache`).

En effet, dans la traduction vers les BDDs, chaque variable apparaissant dans l'expression n'est considérée qu'une fois. Il ne peut donc y avoir de chemin de la racine du BDD vers une feuille `FalseBDD`. Les parties du BDD de la forme `IfBDD(x, TrueBbb, TrueBDD)` seront simplifiées vers `TrueBDD` et on recommencera jusqu'à arriver au BDD `trueBDD`. □

Rappels sur quelques classes de la bibliothèque

La classe `HashSet<E>`

Elle permet de gérer facilement un ensemble d'objets de classe `E`. Le constructeur `HashSet()` créé un nouvel ensemble vide. Elle dispose en particulier des méthodes suivantes :

`add(E e)`	qui ajoute l'élément `e` à l'ensemble (s'il n'y apparait pas déjà).
`boolean contains(E e)`	qui indique si l'objet `e` appartient à l'ensemble.

Comme `LinkedList`, `HashSet` est une implémentation de `Collection`, et donc aussi de `Iterable` (voir ci-dessous).

La classe `LinkedList<E>`

C'est une implémentation de listes chaînées. Elle est en particulier munie des méthodes suivantes :

`add(E e)`	qui ajoute l'élément `e` à la liste
`E remove()`	qui enlève et renvoie le premier élément de la liste
`addAll(Collection<E> l)`	qui ajoute à la liste tous les éléments de `l`

On rappelle que `LinkedList` et `HashSet` sont des implémentations de `Collection`, et donc aussi de `Iterable`, ce qui a pour conséquences que :

— L'argument de `addAll` peut être une `LinkedList` ou un `HashSet`.
— On peut itérer sur les éléments d'une `LinkedList` ou d'un `HashSet` avec la notation `for (E e : l) ...`.

La classe `HashMap<K,V>`

Elle est munie des méthodes suivantes :

`put(K k, V v)`	qui associe la valeur `v` à la clé `k`.
`V get(K k)`	qui renvoie la valeur associée à la clé `k` et renvoie `null` si aucune valeur n'est associée à `k`).

Le constructeur `HashMap()` crée une nouvelle table d'association vide.

Composition d'Informatique
Mécanismes de la programmation orientée-objet (INF371)
Promotion 2021

27 juin 2022

Les parties sont indépendantes entres elles et les exercices indépendants entre eux.

1 Exercices

Question 1 Donnez trois instructions XVM qui peuvent causer un débordement de pile (*overflow*). ◇

Solution. PUSH, FETCH(0), GSB, PRX, ALLOC, RFR...

Question 2 L'exécution de l'instruction POP peut-elle causer une erreur ? Et alors quand ? ◇

Solution. Oui,si la pile ne contient d'élément (stack underflow). □

Question 3 Que donne l'exécution du code suivant :

```
static String f() { return ("a" + f()); }
public static void main(String[] args) {
  System.out.println(f()); }
```

Solution. Un débordement de pile (et c'est tout). □

Question 4 Que se passe-t-il quand on compile et exécute le code suivant :

```
try {    System.out.print("Hello " + 1 / 0);   }
catch (ArithmeticException e)
   {  System.out.println("World ! ");   }
```

Solution. Ca compile bien. L'évaluation de l'argument du premier print déclenche une exception avant qu'on ait affiché quelque chose. Donc on voit juste `World !` à l'écran. □

2 Compilation des `switch`

Pour simplifier, on considèrera un seul exemple de type énuméré, à savoir les jours de la semaine :

```
enum Jour = { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
```

On se rappelle qu'on utilise souvent, avec les types énumérés, la construction `switch` :

```
switch (e) {
  case LUNDI : p0;
     break;
  case MARDI : p1;
     break;
     ...
  case DIMANCHE : p6;
     break;
   }
```

On s'intéresse à comment compiler le code source ci-dessus, sachant que :
— `e` est une expression de type `Jour`,
— `p0`, `p1`, `p2`,. . .,`p6` sont des instructions ou séquences d'instructions.

Un premier collègue, A, propose d'utiliser simplement le schéma de compilation habituel, en compilant le code ci-dessus comme on compilerait :

```
if (e == LUNDI) { p0; }
else if (e == MARDI) { p1; }
else if (e == MERCREDI) { p2; }
...
else { p6; }
```

Un autre collègue, B, propose de se ramener plutôt à ce code source qui utilise une variable locale :

```
{ Jour j = e;
    if (j == LUNDI) { p0; }
    else if (j == MARDI) { p1; }
    else if (j == MERCREDI) { p1; }
    ...
    else if (j == SAMEDI) { p5; }
    else { p6; } }
```

où `j` est un nom de variable qui n'est pas utilisé ailleurs.

Question 5 Que pouvez-vous dire de la différence entre ces deux approches ? Laquelle est meilleure ? Sont-elles toutes les deux correctes ? ◇

Solution. La solution de B est correcte. Celle de A est (1) moins efficace car la valeur de `e` est calculée sept fois et (2) incorrecte lorsque ce calcul a des effets de bord (modifie la mémoire) □

Un collègue C trouve la solution précédente pas assez efficace et peu élégante. Il regrette qu'il faille jusqu'à 6 tests pour "trouver" le code exécuté, et que certaines branches (comme `LUNDI`) sont trouvées avant d'autres (`DIMANCHE`).

On veut donc exploiter la représentation des valeurs de type `Jour` pour proposer des schémas de compilation plus efficaces. On sait donc que `LUNDI` est représenté par 0 dans le code machine, `MARDI` par 1, jusqu'à `DIMANCHE` qui est représenté par 6.

Dans un premier temps, C propose d'exploiter le fait que les représentations machine des `Jour` sont ordonnées, pour que le code du switch fasse au plus 3 comparaisons (en fait fasse au plus $\log_2(n)$ comparaisons pour un type énuméré à n valeurs).

Question 6 Donnez le début de cette manière de compiler le switch (disons les six à huit premières instructions machines - il s'agit de montrer que vous avez compris, sans se perdre dans des détails bureaucratiques). ◇

Solution. On fait des tests d'inégalité dichotomiques sur la valeur de `e`. Ca commence, par exemple, par :

$[\![e]\!]$
FETCH(0)
PUSH(3)
LEQ
GTZ(α)
FETCH(0)
PUSH(5)
LEQ
GTZ(β)

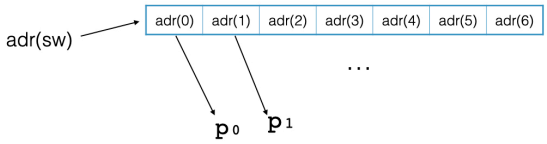


FIGURE 1 – table pour la compilation du `switch` (question 7)

Un autre collègue propose une autre approche qui, après discussion, convainc tout le monde :
— Les branches `p0`, `p1`... sont compilées et les codes machines correspondants sont placés respectivement aux adresses adr(0), adr(1), ... adr(6).
— Ces adresses sont elles-mêmes rangées dans un bloc mémoire placé en adr(*sw*). C'est-à-dire que adr(*sw*) contient la valeur adr(0), adr(*sw* + 1) contient adr(1), *etc.* ... Voir figure 1.
— Le switch lui-même sera alors compilé en utilisant un adressage indirect, similaire à ce qui est fait pour l'appel des méthodes.

Question 7 Proposez un code machine pour le code source ci-dessus à partir de là. ◇

Solution.

PUSH(adr(*sw*))
⟦e⟧
CREAD
GTO

On remarque qu'il faut ajouter un GTO à la fin de chaque bloc `p0`, `p1`... pour continuer l'exécution au bon endroit. □

3 Trouver les vélos les plus proches

Dans cette partie, une grande importance sera accordée à la clarté du code. Vous essayerez également de le rendre aussi concis que possible.

On rappelle en appendice un certain nombre de fonctions ou de classes de bibliothèques Java qui peuvent être utilisées dans ce problème.

Une municipalité ouvre un service de vélos en libre service. Les vélos peuvent être déposés et empruntés n'importe où (pas de bornes).

Les points de la carte sont désignés par deux coordonnées de type `double`.

On voudra afficher tous les vélos disponibles sur un rectangle (correspondant à un écran). Pour cela on va utiliser la distance entre deux positions (x, y) et (x', y') définie par $\max(|x-x'|, |y-y'|)$ (voir figure 2).

Question 8 Définissez une classe `Position` composée des champs `x` et `y` qui corresponde à un point sur la carte. Munissez-la d'une méthode `double distance(Position a)` qui calcule la distance avec la position `a`. ◇

Solution.

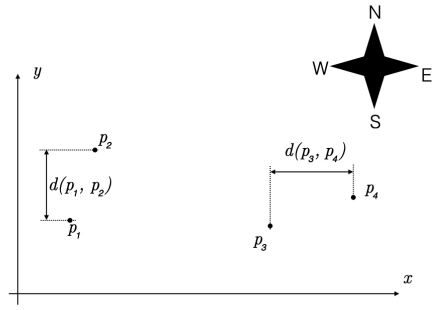


Figure 2 - Distances entre deux positions

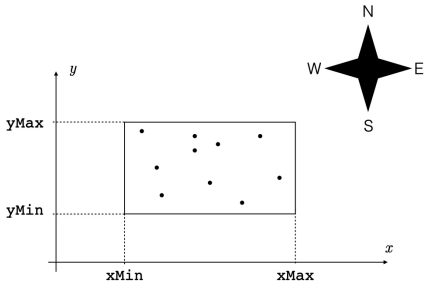


Figure 3 - Visualisation des champs de la carte

```
class Position {
    double x;
    double y;

    Position(double vx, double vy) { x = vx; y = vy; }

    double distance(Position p) {
        return(Math.max(Math.abs(x-p.x), Math.abs(y-p.y)));
    }
}
```

Question 9 Définissez une classe `Velo` qui étend la classe `Position` et comporte en plus un champ `public int iD` qui correspond au numéro d'identification du vélo. Munissez la classe d'un constructeur. ◇

Solution.
```
class Velo extends Position {
    final int iD;
    Velo (int i, double x, double y) { super(x,y); iD = i; }
}
```

Question 10 On veut être sûr qu'un numéro d'identification n'est attribué qu'une seule fois. Comment procédez-vous pour cela ?

Modifiez le constructeur pour qu'il refuse d'utiliser un `iD` qui a déjà été attribué (il pourra lancer une `Error` dans ce cas). ◇

Solution. On va garder un `HashSet<Integer>` des immatriculations déjà attribuées :

```
static HashSet<Integer> immat = new HashSet<Integer>();
```

et on fait la vérification et la mise à jour dans le constructeur :

```
Velo (int i, double x, double y) {
        super(x,y);
        if (immat.contains(i)) throw new Error("immat");
        immat.add(i);
         iD = i; }
}
```

Une `Carte` va être une structure contenant tous les vélos d'un rectangle. On donne pour cela la classe abstraite suivante :

```
abstract class Carte {
    double xMax, xMin, yMax, yMin;
    abstract LinkedList<Velo> closeBy(Position a, double distance);
    abstract void add(Velo r);

    protected int card;
    public int card() { return card; }
}
```

Une carte va donc avoir comme champs les coordonnées délimitant ce rectangle (voir figure 3). De plus, on veut avoir les opérations suivantes sur une carte :

— Pouvoir ajouter un vélo sur la carte (méthode `add`).
— Donner la liste de tous les vélos de la carte situés à une distance inférieure ou égale à `d` d'une position `a` avec la méthode `closeBy`.
— Enfin le champ `card` contiendra le nombre de vélos présents dans le rectangle.

Question 11 Ajoutez à la classe abstraite `Carte` une méthode concrète `double distance(Position a)` qui donne la distance entre la position `a` et le rectangle de la carte[1]. ◇

Solution. On utilise le fait qu'on peut ajouter une méthode concrète dans une classe abstraite.

```
 double distance(Position a) {
        return(Math.max(Math.max(Math.max(xMin - a.x, 0), Math.max(a.x - xMax, 0)),
                        Math.max( Math.max(yMin - a.y, 0) , Math.max(a.y - yMax, 0))));
    }
```

Question 12 Donnez une implémentation simple de la classe abstraite `Carte`. On appellera cette classe `Base`. Cette classe pourra, par exemple, contenir la liste des vélos présents dans le rectangle.

Le constructeur `Base(double x1, double x2, double y1, double y2)` devra construire une carte vide, sans vélos. ◇

Solution.

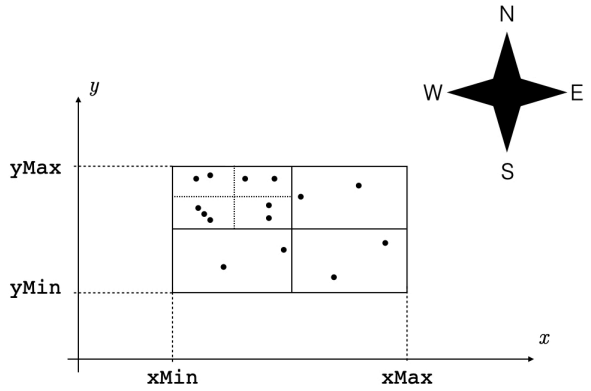
```
class Base extends Carte {
    private LinkedList<Velo> contenu;

    Base(double x1, double x2, double y1, double y2)
        {this.xMax = x2; xMin = x1; yMin = y1; yMax = y2;
            card = 0;
            contenu = new LinkedList<Velo>();
          }

    LinkedList<Velo> closeBy(Position a, Double distance) {
        LinkedList<Restaurant> l = new LinkedList<Velo>();
        for (Restaurant r : contenu) if (r.distance(a) <= distance) l.add(r);
        return(l);
      }

    void add(Velo r) {
      card++;
      contenu.add(r);
    }
}
```

1. C'est-à-dire entre `a` et les points du rectangle qui sont les plus proches de `a`.



Les quatre secteurs de la partition de la carte. Ici le secteur nord-ouest comporte beaucoup de vélos et peut être lui-même partitionné en quatre sous-parties.

Figure 4

Question 13 Quelles sont les complexités des méthodes `add` et `closeBy` de votre implémentation ? ◇

Solution. Ici, respectivement $O(1)$ et $O(n)$, où n est le nombre de vélos présents dans le rectangle.□

On veut rendre plus efficace l'opération de recherche de vélos (méthode `closeBy`). Pour cela, on va :

1. se donner une borne supérieure sur le nombre de vélos qui peuvent être présents dans une instance de `Base`. Appelons `nMax` cette borne supérieure.
2. Lorsqu'une carte comporte plus de `nMax` vélos, on découpe en quatre rectangles de surfaces égales le rectangle de la carte. Ces quatre rectangles correspondront donc aux secteurs nord-est, nord-ouest, sud-est et sud-ouest. Pour chacun de ces rectangles, on aura une carte contenant exactement les vélos appartenant à ces rectangles.
3. Pour chacune de ces cartes, si le rectangle comporte moins de `nMax` vélos, elle sera représentée par une instance de `Base`. Sinon la carte sera elle-même composée de 4 cartes, et ainsi de suite.

Ces cartes correspondent donc à des arbres dont les nœuds ont quatre fils et les feuilles sont des `Base`. On les appelle des *quadtrees*.

La figure 4 illustre la partition en quatre parties de la carte.

Voici un début de définition de la classe `QuadTree` :

```
class QuadTree extends Carte {

    static final int nMax = 10;
    private Carte[] cartes;
```

On comprend donc que le tableau `cartes` contiendra toujours 4 éléments.

Question 14 Commencez par compléter cette définition avec le constructeur `QuadTree(double xMax, double xMin, double yMax, double yMin)` qui construira une carte dont les quatre secteurs seront des `Base` vides. ◇

Solution.

```
public QuadTree( double xMax,double xMin,double yMax,double yMin) {
      card = 0;
      this.xMin = xMin;
      this.xMax = xMax;
      this.yMin = yMin;
      this.yMax = yMax;
      cartes = new Carte[4];
      cartes[0] = new Base(xMax, (xMin+xMax / 2), yMax, (yMax+yMin) / 2);
      cartes[1] = new Base(xMax, (xMin+xMax / 2), (yMax+yMin) / 2, yMin);
      cartes[2] = new Base((xMin+xMax / 2), xMin, (yMax+yMin) / 2, yMin);
      cartes[3] = new Base((xMin+xMax / 2), xMin, yMax, (yMax+yMin) / 2);
    }
```

Question 15 Ecrivez maintenant la méthode `closeBy` de `QuadTree`. ◇

Solution.

```
public LinkedList<Velo> closeBy(Position a, double d) {
    LinkedList<Velo> r = new LinkedList<Velo>();
    for (int i = 0; i < cartes.length; i++)
        if (cartes[i].distance(a) <= d)
            r.addAll(cartes[i].closeBy(a, d));
    return r;
}
```

Question 16 Sans justifier précisément, et sans rentrer dans le détail, quelle complexité pourra-t-on espérer pour `closeBy` si la distance de recherche n'est pas trop grande et que les vélos sont uniformément distribués sur la ville ? ◇

Solution. Dans certains cas, on va avoir quatre appels récursifs (si la position est proche du milieu de la carte). Mais en général on peut espérer éliminer trois ou deux parmi les quatre sous-cartes. La complexité pratique devrait donc être logarithmique par rapport au nombre de vélos.

Elle devrait également être proportionnelle à `nMax`. □

Question 17 Ecrivez une fonction `static Carte smartAdd(Carte c, Velo r)` qui rende la Carte équivalente à `c` à laquelle on aura ajouté le vélo `r`. Le point important est que si `c` est de la classe `Base` et que le nombre de vélo dépasse `nMax`, alors on devra créer un QuadTree. ◇

Solution.

```
static Carte smartAdd(Carte c, Velo v) {
     c.add(v);
     if (!(c instanceof Base) ||  (c.card() < nMax)) return(c);
     Carte n = new QuadTree(c.xMax, c.xMin, c.yMax, c.yMin);
     for (Velo w : ((Base)c).contenu)
         n.add(w);
     return n;
}
```

Question 18 Avec ce qui précède, terminez l'implémentation de la classe `QuadTree` en écrivant sa méthode `add`. ◇

Solution.

```
public void add(Velo v) {
      for  (int i = 0; i < cartes.length; i++)
          if (cartes[i].distance(v) <= 0) {
              cartes[i] = smartAdd(cartes[i], v);
              card++;
              return;
          }
      throw new Error("no map - should not happen");
  }
```

Question 19 Pouvez-vous dire quelque chose de la complexité de cette dernière méthode ? (en supposant que les vélo sont distribués uniformément à travers la ville) ◇

Solution. La taille des `Base` étant bornée par la constante `nMax`, le temps pour les `split` est aussi borné par une constante. La complexité est donc logarithmique par rapport au nombre de vélos. □

Question 20 Ajoutez aux classes `Base` et `QuadTree` une méthode `boolean remove(Velo v)` qui :
- lorsque le vélo est présent dans la carte le retire et renvoie `true`,
- lorsqu'il n'est pas présent laisse la carte inchangée et renvoie `false`.

On pourra supposer que le test d'égalité structurelle pour `Velo` a été bien implémenté et qu'un vélo est présent au plus une fois dans la carte. ◇

Solution. Dans `Base` :

```
      public boolean remove(Velo v) {
       if (contenu.remove(v)) {
           card --;
           return true;
       }
       return false;
  }
```

dans `QuadTree` :

```
public boolean remove(Velo v) {
      for (Carte c : cartes)
          if (c.distance(v) == 0 && c.remove(v)) {
              card --;
              return true;
          }
      return false;
  }
```

Question 21 Supposons que l'on veuille changer le type de la méthode `closeBy` pour qu'elle prenne aussi en argument un niveau de charge de la batterie du vélo. Comment pourrait-on organiser les données pour rendre la recherche encore plus efficace ? ◇

Solution. Ajouter la charge dans la recherche revient à ajouter une dimension à l'espace dans lequel on cherche les vélos. On pourrait donc imaginer des cartes qui sont bornées non seulement par des x et des y minimaux et maximaux, mais également des charges minimales et maximales. Dans ce cas, une carte non basique serait découpé en huit sous-cartes, et non pas en quatre.

Remarquons que c'est une question théorique dont l'intérêt pratique est discutable !

Les complexités asymptotiques resteraient inchangées. □

Appendice

Fonctions mathematiques Java

On dispose en particulier des fonctions statiques suivantes :

`double Math.abs(double x)`	valeur absolue
`double Math.max(double x, double y)`	maximum de deux nombres
`double Math.min(double x, double y)`	minimum de deux nombres

La classe `LinkedList<E>`

Elle est munie des méthodes suivantes :

`add(E e)`	qui ajoute l'élément `e`
`E remove()`	qui enlève et renvoie le premier élément de la liste
`addAll(Collection<E> l)`	qui ajoute à la liste tous les éléments de `l`
`boolean remove(Object o)`	qui enlève l'objet `o` de la liste et renvoie `true`, si `o` fait partie de la liste, et renvoie `false` et laisse la liste inchangée sinon.

Pour cette dernière méthode, on pourra supposer qu'elle vérifie l'égalité structurelle sur les objets.

On rappelle que `LinkedList` est une implémentation de `Collection` donc l'argument de `addAll` peut être une `LinkedList`.

On rappelle aussi qu'on peut itérer sur les éléments d'une `LinkedList` avec la notation `for (E e : l) ...`.

La classe `HashSet<E>`

Elle est munie des methodes suivantes :

`add(E e)` qui ajoute l'élément `e` à l'ensemble
`boolean contains(E e)` qui vérifie si `e` appartient à l'ensemble.