COURS 421-b, COMPOSITION D'INFORMATIQUE

Philippe Jacquet, Luc Maranget, Philippe Baptiste

27 janvier 2006

⋆ ⋆ ⋆

Partie I, Programmation en Java

Question 1. Les entiers et les tableaux. Chacune des sous-questions donne une définition de méthode, que l'on suppose survenir dans une classe `Test`. Pour chacune des définitions de méthode, il faut :

1. Indiquer si la méthode est acceptée par le compilateur ou non. Si non, dire pourquoi et corriger.
2. Commenter rapidement le sens de la méthode définie, en oubliant pas terminaison et échec.
3. Lorsque que la méthode ne calcule pas ce qu'elle annonce, corriger les erreurs de programmation.

a)

```java
static int f1(int y){
  if (y <= 1)
    return 0 ;
  else
    return f1(y-1) + f1(y-2) ;
 }
```

Réponse : Correct, termine et renvoie toujours 0. □

b)

```java
static int f2(int y){
  if (y <= 0)
    return 1 ;
  else
    return f2(y-1) + f2(y-1) ;
 }
```

Réponse : Correct, renvoie 2^y si $y \geq 0$, et 1 si $y < 0$. Très lente. □

c) Calcule la factorielle.

```java
static int fact(int y){
  if (y == 0) return 1 ;
  y*fact(y-1) ;
}
```

Réponse : Incorrect, la fonction déclare renvoyer un **int**, mais n'exécute pas toujours **return**. Il faut remplacer la dernière ligne par **return** `y*fact(y-1)`. Notons que `fact` ne termine pas pour $n < 0$. □

d) Renvoie un tableau de taille n dont les cases sont initialisées aux valeurs $0, 1, \ldots, n-1$.

```
static int[] init(int n){
  int[] t = new int (n);
  for (int i = 0 ; i < n ; i++) t[i] = i;
  return t;
}
```

Réponse : Incorrect, la syntaxe correcte de la création de tableau est **new int** `[n]`. La fonction échoue pour $n < 0$. □

e) Appelée avec un tableau d'entiers t et un entier n, renvoie un indice k tel que `t[k] == n` ou -1 si n n'est pas dans le tableau.

```
static int find(int [] t, int n) {
  for (int k = 1 ; k <= t.length ; k++) {
    if (t[k] == n) return k ;
  }
  return -1 ;
}
```

Réponse : Acceptée par le compilateur. L'itération est incorrecte et conduit à l'échec systématique si n n'est pas dans t. Il faut écrire :

```
static int find(int [] t, int n) {
  for (int k = 0 ; k < t.length ; k++) {
    if (t[k] == n) return k ;
  }
  return -1 ;
}
```

□

f) La fonction `find2` réalise la même recherche que celle de sous-question précédente, sous une condition portant sur le tableau t qu'il faut identifier. Il est inutile de discuter la correction, la fonction `find2` est correctement programmée.

```
static int find2(int [] t, int n) {
  int low = 0, high = t.length-1 ;
  while (low <= high) {
    int m = (low+high) / 2 ;
    if (n < t[m]) {
      high = m-1 ;
    } else if (n > t[m]) {
      low = m+1 ;
    } else {
      return m ;
    }
  }
  return -1 ;
}
```

Réponse : La fonction compile et réalise une recherche dite dichotomique de coût $\log(T)$ où T est la taille de t, à condition que t soit trié en ordre croissant. □

Question 2. Même exercice que la question précédente mais sur les arbres. Les méthodes sont cette fois définies dans la classe **Tree**.

```
class Tree {
  int val ;
```

```
  Tree left, right ;

  Tree (Tree left, int val, Tree right) {
    this.left = left ; this.val = val ; this.right = right ;
  }
}
```

a) Renvoie la valeur (champ `val`) contenue dans la cellule d'arbre.

```
static int getVal() { return val ; }
```

Réponse : Ne compile pas, car la méthode est statique et donc n'a pas accès à **this** `.val`. Il faut écrire une méthode dynamique.

```
int getVal() { return val ; }
```

□

b) Renvoie la hauteur de l'arbre passé en argument.

```
static int hauteur(Tree t) {
  if (t == null)
    return 0 ;
  return 1+Math.max(t.left.hauteur(), t.right.hauteur()) ;
}
```

Réponse : Ne compile pas, car la méthode statique `hauteur` est appelée récursivement comme une méthode dynamique. Il faut remplacer la dernière ligne par.

```
  return 1+Math.max(hauteur(t.left), hauteur(t.right)) ;
```

□

c) Teste si un arbre est une feuille.

```
static isLeaf(Tree t) {
  return t.left == null && t.right == null ;
}
```

Réponse : Ne compile pas, il manque le type des valeurs renvoyées dans la déclaration. Une fois cette correction effectuée, la fonction échoue sur l'arbre vide. Il faut écrire par exemple.

```
static boolean isLeaf(Tree t) {
  if (t == null) {
    return false ;
  } else {
    return t.left == null && t.right == null ;
  }
}
```

□

d) Affiche le contenu de toutes les feuilles de l'arbre t.

```
static void printLeaves(Tree t) {
  if (isLeaf(t)) {
    System.out.println(t.getVal()) ;
  } else {
    printLeaves(t.left) ; printLeaves(t.right) ;
  }
}
```

Réponse : Compile, échoue sur l'arbre vide, on peut écrire.

```
static void printLeaves(Tree t) {
  if (t == null) return ;
  ...
}
```

□

e)

```
static void f1() {
  Tree t = new Tree(null, 1, null) ;
  t = new Tree(t, 2, t) ;
  t = new Tree(t, 3, t) ;
  t = new Tree(t, 4, t) ;
  printLeaves(t) ;
}
```

Réponse : Affiche huit fois 1. □

f)

```
static void f2() {
  Tree t = new Tree(null, 1, null) ;
  t = new Tree(t, 2, t) ;
  t = new Tree(t, 3, t) ;
  t = new Tree(t, 4, t) ;
  t.right = t ;
  printLeaves(t) ;
}
```

Réponse : Affiche une infinité de 1. □

Partie II, Compression de texte

Les algorithmes de compression sont indispensables lorsqu'il s'agit de stocker ou de déplacer des quantités importantes d'information.

Partie II-A, Texte

Dans tout le problème nous considérons des textes qui sont des suites de bits valant donc 0 ou 1.Nous définissons la classe **Text** :

```
class Text {
  int bit ; // 0 ou 1
  Text next ;

  Text(int a, Text c) { bit = a ; next = c ; }
}
```

On notera donc que les textes sont tout simplement des listes de zéros et de uns, et que le texte vide est représenté par **null**.

Question 3. Écrire une méthode *statique* **static** `String toString(`**Text** `c)` qui renvoie la chaîne de zéros et de uns qui exprime le texte c passé en argument. On suppose que les textes sont petits et on ne demande pas de porter une attention exagérée à l'efficacité.

Réponse : Portons une attention exagérée et écrivons un `toString` linéaire en la taille du texte.

```
static String toString(Text c) {
  StringBuffer b = new StringBuffer () ;
  for ( ; c != null ; c = c.next)
    b.append(c.bit) ;
  return b.toString() ;
}
```

Sans attention exagérée, `toString` quadratique acceptable dans les conditions de l'énoncé.

```
static String toString(Text c) {
  String r = "" ;
  for ( ; c != null ; c = c.next)
    r = r + c.bit ;
  return r ;
}
```

□

Question 4. Écrire une méthode **static boolean** `equals(`**Text** `c1,` **Text** `c2)` qui renvoie **true** si les deux textes sont identiques en terme de contenu, ou renvoie **false** autrement.

Réponse :

```
  static boolean equals(Text c1, Text c2) {
    if (c1 == null) {
      return c2 == null ;
    } else if (c2 == null) {
      return false ;
    } else {
      return c1.bit == c2.bit && equals(c1.next, c2.next) ;
    }
  }
```

□

Question 5. Écrire une méthode **static Text** `addBit(`**Text** `c,` **int** `b)` qui renvoie le texte obtenu en ajoutant le bit b à la fin du texte c.

Réponse :

```
  static Text addBit(Text c, int b) {
    if(c==null){
      return new Text(b,null) ;
    } else {
      return new Text(c.bit,addBit(c.next,b));
    }
  }
```

□

Question 6. On dit qu'un texte c_1 est préfixe d'un texte c_2, si il existe un texte s tel que la concaténation de c_1 et de s est égale à c_2, c'est-à-dire $c_2 = c_1 \cdot s$. Le texte s est alors un *suffixe* du texte c_2. Écrire une méthode **static boolean** `isPrefix(`**Text** `c1,` **Text** `c2)` qui teste si le texte c_1 est préfixe du texte c_2.

Réponse :

```
static boolean isPrefix(Text c1, Text c2) {
  if (c1 == null) {
    return true ;
  } else if (c2 == null) {
    return false ;
  } else {
    return c1.bit == c2.bit && isPrefix(c1.next, c2.next) ;
  }
}
```

□

Question 7. En supposant que le texte c_1 est un préfixe du texte c_2, écrire une méthode **static Text** `getSuffix(`**Text** `c1,` **Text** `c2)` qui renvoie le suffixe s du texte c_2 qui suit c_1 dans c_2, c'est-à-dire tel que $c_2 = c_1 \cdot s$.

Réponse :

```
static Text getSuffix(Text c1, Text c2) {
  if (c1 == null) {
    return c2 ;
  } else {
    return getSuffix(c1.next, c2.next) ;
  }
}
```

□

Partie II-B, Base de préfixes

Dans cette partie on traite d'un ensemble de textes appelé « base de préfixes » La base de préfixes satisfait la propriété de préfixe croissant : pour chaque texte non vide contenu dans la base, il existe dans la base, un texte préfixe de longueur exactement inférieure d'une unité. Ce préfixe est appelé le parent du texte en question. Par ailleurs, un texte donné est toujours présent au plus une fois dans la base de préfixes.

Chaque texte c de la base de préfixes est associé à un numéro d'ordre, au numéro d'ordre de son texte parent, et au bit complémentaire qu'il faut rajouter à la fin du parent pour obtenir le texte c. Par exemple, si le texte est `010011` son texte parent est `01001` et le bit complémentaire est `1`. L'ensemble de ces informations est donné par un objet de la classe **Info** suivante :

```
class Info {
  Text text ;
  int order ;   // Numéro d'ordre de text
  int parent ;  // Numéro du parent de text
  int nextBit ; // Bit complémentaire

  Info (Text t, int n, int p, int b) {
    text = t ; order = n ;
    parent = p ; nextBit = b ;
  }
}
```

Par la suite, on supposera que le texte **null** est présent dans la base de préfixes, associé au numéro d'ordre zéro, que le numéro d'ordre de son parent est par convention égal à -1, et que son bit complémentaire vaut aussi -1.

Dans un premier temps on implémente la base de préfixes par une liste :

```
class List{
  Info info ;
  List next ;

  /* Constructeur de la base initiale */
  List() { info = new Info (null, 0, -1, -1) ; next = null ; }

  /* Constructeur usuel */
  List(Info i, List d) { info = i ; next = d; }
}
```

Les deux méthodes suivantes sont des méthodes statiques de cette classe **List**.

Question 8. Écrire une méthode **static int** `find(`**Text** `c, `**List** `db)` qui renvoie le numéro d'ordre du texte c dans la base de préfixes `db`, ou -1 si le texte c n'est pas présent dans la base.

Réponse :

```
static int find(Text c, List db){
  for ( ; db != null ; db = db.next) {
    if (Text.equals(c,db.info.text)) {
      return db.info.order ;
    }
  }
  return -1 ;
}
```

□

Question 9. Écrire une méthode **static Info** `findLargestPrefix(`**Text** `c, `**List** `db)` qui renvoie les informations associées au plus long préfixe du texte c présent dans la base de préfixes `db`. On observera que `findLargestPrefix` est correctement définie, puisque la base contient au moins le préfixe vide **null** (associé à 0, -1 et -1).

Réponse :

```
static Info findLargestPrefix(Text c, List db) {
  Info r = new Info (null, 0, -1, -1) ;
  for ( ; db != null ; db = db.next) {
    if (Text.isPrefix(db.info.text, c) &&
        Text.isPrefix(r.text, db.info.text)) {
      r = db.info ;
    }
  }
  return r ;
}
```

En fait, les préfixes étant introduits par ordre croissant dans la base (cf. la méthode `add` de la question suivante), on peut se contenter de renvoyer le premier préfixe trouvé.

```
static Info findLargestPrefix(Text c, List db) {
  for ( ; db != null ; db = db.next) {
    if (Text.isPrefix(db.info.text, c)) {
      return db.info ;
    }
  }
  // Not reached
```

```
  throw new Error ("La base de préfixes est incorrecte") ;
}
```

□

Afin de permettre plusieurs implémentations de la base de préfixes, celle-ci sera représentée par un objet qui implémente l'interface suivante :

```
interface Database {
  public void add(Info i) ;
  public Info findLargestPrefix(Text c) ;
}
```

Où `add(Info i)` ajoute l'information `i` à la base de préfixes (on supposera que `i.text` n'est pas dans la base), et `findLargestPrefix (Text c)` renvoie les informations associées par la base au plus long préfixe du texte `c`.

Question 10. Compléter la classe suivante **DataList** qui implémente l'interface **Database** à l'aide des listes.

```
class DataList implements Database {
 private List p ;

 DataList() { p = new List () ; }
 ⋮
}
```

Réponse :

```
class DataList implements Database {
  private List p ;

  DataList() { p = new List () ; }

  public void add(Info i) { p = new List (i, p) ; }

  public Info findLargestPrefix(Text c) {
    return List.findLargestPrefix(c, p) ;
  }
}
```

□

Partie II-C, Compression de Ziv-Lempel

L'algorithme Ziv-Lempel (le fameux utilitaire zip) compresse un texte en le coupant en tranches successives. Le découpage se fait à partir du texte original et de la base de préfixes initialement constituée du texte vide **null**. À chaque étape de compression, on dispose d'un numéro d'étape n, d'une base de préfixes courante et d'un suffixe courant s du texte original. Le découpage d'une tranche de s consiste, tant que s n'est pas **null**, à chercher le plus court préfixe s_1 du texte s qui n'est pas dans la base de préfixes. Il y a alors deux cas,

- Si s_1 existe, on insère s_1 dans la base de préfixes avec le numéro d'ordre n (et les informations additionnelles numéro du parent et bit complémentaire), on retire s_1 du suffixe courant et on met n à $n + 1$.

- Si s_1 n'existe pas, alors, par construction de la base de préfixes, s est présent dans la base de préfixes. On se contente alors de retirer le préfixe complet s du suffixe courant s — ce qui fait que la compression est terminée, s devenant vide.

En outre, dans les deux cas ci-dessus, le texte compressé est émis au passage : chaque fois qu'un préfixe q est retiré de s (q est s_1 dans le premier cas et s dans le second cas), on affichera la paire composée du numéro d'ordre du *parent* p de q et du bit complémentaire b tel que $q = p \cdot b$.

Figure 1: Compression du texte `0100110111100`

étape	tranche	paire émise	ajouté à la base
1	`0`	0 0	`0` $\rightarrow (1, 0, 0)$
2	`1`	0 1	`1` $\rightarrow (2, 0, 1)$
3	`00`	1 0	`00` $\rightarrow (3, 1, 0)$
4	`11`	2 1	`11` $\rightarrow (4, 2, 1)$
5	`01`	1 1	`01` $\rightarrow (5, 1, 1)$
6	`111`	4 1	`111` $\rightarrow (6, 4, 1)$
7	`00`	1 0	

Considérons par exemple la compression du texte `0100110111100`, ce texte est découpé en `0`, `1`, `00`, `11`, `01`, `111`, `00`, et compressé comme indiqué à la figure 1.

Question 11. Donner l'ordre de grandeur de la taille du code compressé —c'est-à-dire du nombre de paires émises— dans le cas particulier où le texte d'origine est une suite de n zéros.

Réponse : Tant qu'on n'a pas atteint l'état final la longueur du k préfixe inséré dans la base de préfixes est k et la longueur du suffixe restant est $n - \sum_{i=1}^{k} i = n - \frac{(k+1)k}{2}$. Le nombre d'étape est donc environ $\sqrt{2n}$ ce qui est l'ordre de grandeur de la taille du code compressé. □

Pour décrire un état de la compression on introduit la classe **Zip**. Un objet de la classe **Zip** contient le numéro d'étape et le suffixe courant. La base de préfixes est inchangée durant la compression (mais pas son contenu, évidemment), elle est encodée comme une variable statique de la classe **Zip**.

```
class Zip {
  int stepNumber ; // Numéro d'étape
  Text suffix;     // Suffixe courant
  static Database db; // Base de préfixes

  /* Constructeur de l'état initial */
  Zip(Text c) {
    stepNumber = 1 ; suffix = c ;
  }

  /* Constructeur d'un état quelconque */
  Zip(int n, Text c) {
    stepNumber=n ; suffix=c;
  }
```

Question 12. Écrire une méthode **Zip** `step()` qui donne le nouvel état de la compression après l'application de l'étape de compression sur le suffixe courant `suffix`. On supposera que `suffix` est distinct de **null**. La méthode `step` devra par ailleurs modifier la base de préfixes et afficher la paire d'entiers représentant le préfixe compressé. Cette question et la suivante sont liées, consultez la question suivante maintenant.

Réponse :

```
Zip step() {
  Info prefix = db.findLargestPrefix(suffix) ;
  Text rem = Text.getSuffix(prefix.text, suffix) ;
  if (rem == null) {
    System.out.println(prefix.parent + " " + prefix.nextBit) ;
    return new Zip(stepNumber, null) ;
  } else {
    Text t = prefix.text ;
    int bit = rem.bit ;
    t = Text.addBit(t, bit) ;
    System.out.println(prefix.order + " " + bit) ;
    db.add(new Info (t, stepNumber, prefix.order, bit)) ;
    return new Zip (stepNumber+1, rem.next) ;
  }
}
```

□

Question 13. Ecrire une méthode **static void** compress(**Text** c) qui affiche toutes les paires résultant de la compression du texte c. La méthode compress est responsable de l'initialisation de la base de préfixes.

Réponse :

```
static void compress(Text c) {
  Zip.db = new DataList () ;
  Zip state = new Zip(c) ;
  while (state.suffix != null) {
    state = state.step() ;
  }
}
```

□

Partie II-D, Compression efficace

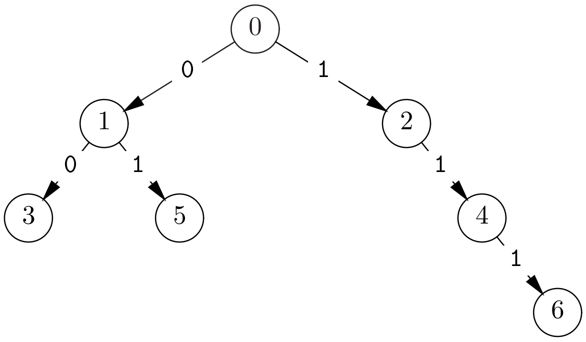
L'usage d'une liste pour réaliser la base de préfixes introduit une inefficacité importante. Nous allons corriger ce point en implémentant la base par un arbre. Nos arbres sont des arbres binaires utilisés pour associer les numéros d'ordre aux préfixes.

```
class Tree {
  int val ; // Numéro d'ordre
  Tree left, right ; // Sous-arbres gauche et droit

  Tree (Tree left, int val, Tree right) {
    this.left = left ; this.val = val ; this.right = right ;
  }
}
```

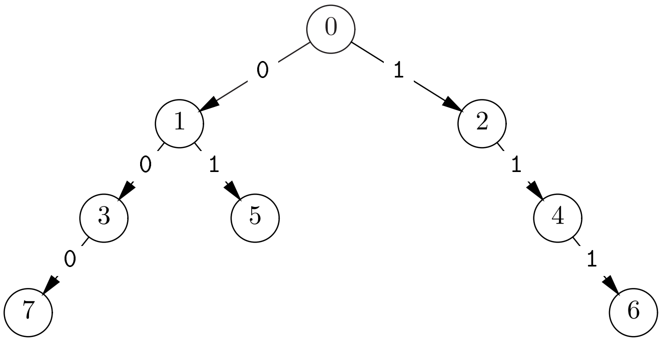
On observe que les sommets des arbres ne comportent plus qu'une unique information : le numéro d'ordre. Les préfixes sont encodés implicitement par un chemin dans l'arbre, et les autres informations (numéro d'ordre du parent et bit complémentaire) peuvent se retrouver grâce à la structure d'arbre. La figure 2 représente l'arbre qui résulte de la compression du texte 0100110111100 (cf. figure 1). On interprète un chemin dans l'arbre comme un texte, 0 signifiant « à gauche » et 1 signifiant « à droite ». Ainsi, l'arbre de la figure 2 associe le numéro d'ordre 5 au text 01, et le numéro d'ordre 6 aux texte 111. On remarque que la propriété de préfixe croissant entraîne que tous les numéros des sommets de l'arbre sont pertinents.

Figure 2: Une base de préfixes sous forme d'arbre



Question 14. Dessiner l'arbre qui résulte de la compression du texte `01001101111000`.

Réponse :



□

Question 15. Écrire une méthode **static int** `find(`**Text** `c,` **Tree** `t)` qui renvoie le numéro d'ordre associé au texte c, ou -1 si aucun chemin de t ne représente le texte c.

Réponse :

```
static int find(Text c, Tree t) {
  if (t == null) {
    return -1 ;
  } else if (c == null) {
    return t.val ;
  } else if (c.bit == 0) {
    return find(c.next, t.left) ;
  } else { // c.bit == 1
    return find(c.next, t.right) ;
  }
}
```

□

Question 16. Écrire une méthode **static Tree** `insert(`**Text** `c,` **int** `n,` **Tree** `t)` qui renvoie l'arbre obtenu par insertion du texte c dans l'arbre t. Dans le nouvel arbre, le numéro du sommet correspondant au chemin c est n. On se place dans les conditions de la compression, c'est-à-dire que tous les préfixes de c sont présents dans l'arbre, sauf c lui même.

Réponse :

```
static Tree insert(Text c, int n, Tree t) {
  if (c == null) {
    return new Tree (null, n, null) ;
  } else if (c.bit == 0) {
    return new Tree (insert(c.next, n, t.left), t.val, t.right) ;
  } else { // c.bit == 1
    return new Tree (t.left, t.val, insert(c.next, n, t.right)) ;
  }
}
```

□

Question 17. Ecrire une méthode **static Info** `findLargestPrefix` (**Text** `c`, **Tree** `t`) qui renvoie les informations associées au plus long préfixe du texte c présent dans l'arbre. On se place ici encore dans les conditions de la compression c'est-à-dire que l'arbre t contient au moins la racine portant le numéro d'ordre 0.

Réponse : Voici un code qui procède en une recherche dans l'arbre.

```
// p is parent number, b is parent next bit, Tree t != null
private static Info findLargestPrefix(Text c, int p, int b, Tree t) {
  if (c == null) { // text present in data base
    return new Info (null, t.val, p, b) ;
  } else if (c.bit == 0) {
    if (t.left == null) {
      return new Info (null, t.val, p, b) ;
    } else {
      Info i = findLargestPrefix(c.next, t.val, 0, t.left) ;
      return
        new Info (new Text(0, i.text), i.order, i.parent, i.nextBit) ;
    }
  } else { // c.bit == 1
    if (t.right == null) {
      return new Info (null, t.val, p, b) ;
    } else {
      Info i = findLargestPrefix(c.next, t.val, 1, t.right) ;
      return
        new Info (new Text(1, i.text), i.order, i.parent, i.nextBit) ;
    }
  }
}

static Info findLargestPrefix(Text c, Tree t) {
  return findLargestPrefix(c, -1, -1, t) ;
}
```

Pour une fois un code itératif est peut-être plus simple à produire.

```
  // Dans les conditions de la compression, on a t != null
  static Info findLargestPrefix(Text c, Tree t) {
    Text prefix = null ; // chemin dans l'arbre : de la racine -> t
    int prevVal = -1 ; // Numéro du parent
    int prevBit = -1 ; // Bit 'parent'
    for ( ; c != null ; c = c.next) {
      // avancer dans t d'abord, on a tjs ici t != null
      Tree next = c.bit == 0 ? t.left : t.right ;
```

```
      if (next == null) { // sorti de l'arbre, chemin -> t  plus long prefixe
        return new Info (prefix, t.val, prevVal, prevBit) ;
      }
      // Ajuster toutes les variables
      prevVal = t.val ; t = next ;
      prefix = Text.addBit(prefix, c.bit) ;
      prevBit = c.bit ;
    }
    // On ne peut parvenir ici que pour c (intial) chemin valide dans l'arbre
    return new Info (prefix, t.val, prevVal, prevBit) ;
  }
```

Le code ci-dessus est quadratique dans la longueur du préfixe renvoyé (à cause de la construction de `prefix` par **Text**.`addBit`. En fait on aurait pu construire le préfixe à l'envers et l'inverser au tout dernier moment, quand on construit l'objet **Info**.

Enfin, un code nettement plus simple procède en plusieurs étapes.

```
private static Text findPrefix(Text c, Tree t) {
  if (c == null) {
    return null ;
  } else if (c.bit == 0) {
    return new Text (0, findPrefix(c.next, t.left)) ;
  } else {
    return new Text (1, findPrefix(c.next, t.right)) ;
  }
}

private static int last(Text c) {
  for ( ; c.next != null ; c = c.next) ;
  return c.bit ;
}

private static Text exceptLast(Text c) {
  if (c.next == null) {
    return null ;
  } else {
    return new Text (c.bit, exceptLast(c.next)) ;
  }
}

static Info findLargestPrefix(Text c, Tree t) {
  Text p = findPrefix(c, t) ;
  int order = find(p, t) ;
  if (p == null) {
    return new Info (p, order, -1, -1) ;
  } else {
    return new Info (p, order, find(exceptLast(p)), last(p)) ;
  }
}
```

On note que les trois codes proposés sont linéaires en la taille du préfixe renvoyé (compte tenu de la modification décrite du code itératif). □

Question 18. Compléter la classe suivante **DataTree**, qui réalise la base de préfixes à l'aide d'un arbre.

```
class DataTree implements Database {
  private Tree t ;
  DataTree() { ... }
  ⋮
}
```

En outre, expliquer comment modifier la méthode `compress` de la classe **Zip**, afin d'employer les arbres à la place des listes.

Réponse :

```
class DataTree implements Database {
  private Tree t ;

  DataTree() {
    t = new Tree(null, 0, null) ;
  }

  public void add(Info i) {
    t = Tree.insert(i.text, i.order, t) ;
  }

  public Info findLargestPrefix(Text c) {
    return Tree.findLargestPrefix(c, t) ;
  }
}
```

Il faut en outre, dans la méthode `compress`, remplacer l'initialisation
`Zip.db = new DataList()` par `Zip.db = new DataTree()`. □

Question 19. Quel est l'ordre de grandeur du coût en temps de la compression d'un texte composé de n bits.

Réponse : La compression divise le texte en sous-mots dont la somme des tailles vaut n. Pour un sous-mot donné de taille k on procède à un nombre d'opérations élémentaires proportionnel à k, la compression est donc en $O(n)$. □

Partie II-E, décompression

Le texte compressé est représenté en machine par deux tableaux de n entiers chacun. Le premier tableau regroupe les numéros d'ordre, et le second les bits (de la paire de codage de rang k). Par exemple, le texte compressé à la figure 1 est représenté par deux tableaux de 7 entiers.

```
{0, 0, 1, 2, 1, 4, 1}
{0, 1, 0, 1, 1, 1, 0}
```

Question 20. Écrire une méthode `static Text decompress(int [] order, int [] bit)` qui renvoie le texte d'origine à partir de son code compressé `order` et `bit`. Indication : commencer par la fin.

Réponse : Toute l'astuce consiste à remarquer que la paire (p, b) de numéro d'ordre k est au rang $k-1$ dans les tableaux `order` et `bit`. On profite de ce que le texte décodé est naturellement construit à l'envers (c'est une liste), ce qui permet d'interpréter les numéros de séquence en remontant vers le début du texte codé.

```
static Text decompress(int [] order, int [] bit) {
```

```
  int n=c.length;
  Text t=null;
  for (int i = n ; i > 0 ; i--) {
    int k = i ;
    do {
      t = new Text (bit[k-1], t);
      k = order[k-1] ;
    } while (k > 0) ;
  }
  return t;
}
```

Le code est assez astucieux et la question teste la bonne compréhension de la technique de compression. L'astuce est malheureusement sans grande portée pratique, car lire le texte compressé à l'envers n'est pas naturel. □