X2021 — INF411
Contrôle classant / 15 novembre 2022 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les FUI et FUI-FF. Les ordinateurs, Ipad, dictionnaires électroniques, les tablettes et téléphones portables sont interdits.

L'énoncé est composé de deux problèmes indépendants, que vous pouvez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pouvez, quand vous traitez une question, considérer comme déjà traitées les questions précédentes du même problème.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Mots croisés parfaits

Dans ce problème, on cherche à construire des grilles de mots croisés parfaites avec des mots de 5 lettres, c'est-à-dire des grilles 5×5 formant 5 mots du dictionnaire horizontalement et 5 mots du dictionnaire verticalement. Voici un exemple avec des mots anglais :

r	o	u	g	h
a	p	p	l	e
t	e	p	e	e
t	r	e	a	d
y	a	r	n	s

On suppose donné un dictionnaire contenant N mots de 5 lettres, sous la forme d'une table de hachage `words` de type `HashSet<String>`. Chaque mot est formé exclusivement de lettres alphabétiques minuscules, c'est-à-dire de caractères compris entre `'a'` et `'z'`.

Question 1 Justifier qu'il n'est pas réaliste de procéder par la force brute, c'est-à-dire en remplissant la grille avec toutes les combinaisons possibles de lettres puis en vérifiant à chaque fois si la grille obtenue est ou non une solution.

Correction : Vérifier si une grille est une solution se fait facilement, indépendamment de N, car il s'agit de vérifier la présence de 10 éléments dans une table de hachage. Mais il y a $26^{25} > 10^{35}$ façons de remplir la grille, ce qui est bien trop grand pour être exploré dans un temps raisonnable.

Rebroussement. On tente une première approche du problème par rebroussement. La figure 1 contient un code Java avec une méthode `solve` qui renvoie `true` si et seulement si une solution existe. En cas de succès, la solution se trouve alors contenue dans la matrice `grid`.

```
class PerfectCrosswords {
  HashSet<String> words; // tous les mots de 5 lettres

  char[][] grid = new char[5][5]; // la grille en cours de construction

  boolean checkColumns() {
    for (int j = 0; j < 5; j++) {
      String s = ""+grid[0][j]+grid[1][j]+grid[2][j]+grid[3][j]+grid[4][j];
      if (!words.contains(s)) return false;
    }
    return true;
  }

  boolean bt(int i, int j) {
    if (i == 5)
      return checkColumns();
    if (j == 5) {
      String s = ""+grid[i][0]+grid[i][1]+grid[i][2]+grid[i][3]+grid[i][4];
      if (!words.contains(s))
        return false;
      return bt(i + 1, 0);
    }
    for (char c = 'a'; c <= 'z'; c++) {
      grid[i][j] = c;
      if (bt(i, j+1))
        return true;
    }
    return false;
  }

  boolean solve() {
    return bt(0, 0);
  }
}
```

Figure 1 – Approche par rebroussement.

Question 2 Justifier que la méthode `solve` termine. Justifier que la méthode `solve` renvoie `true` si et seulement si une solution existe.

Correction :

terminaison : On a clairement l'invariant $0 \leq i, j \leq 5$. À chaque appel récursif de `bt`, le couple (i, j) augmente strictement pour l'ordre lexicographique, ce qui assure la terminaison.

correction : En entrée de `bt`, on a une grille partiellement remplie (pour les lignes $< i$ et la ligne i pour les colonnes $< j$) dont les lignes $k < i$ contiennent toutes des mots du dictionnaire `words`. En sortie de `bt`, on a `true` si et seulement si cette grille a pu être complétée en une solution.

Considérons les différents appels récursifs :

ligne 21 On a bien vérifié que la ligne i forme un mot (lignes 18–20) avant de passer à la ligne suivante.

ligne 25 On a ajouté un caractère en (i, j) et on passe à la colonne suivante.

Considérons alors les différents cas où un booléen est renvoyé :

ligne 16 On renvoie `true` si et seulement si les colonnes forment des mots (ce que vérifie `checkColumns`) et donc `true` si et seulement si `grid` est une solution.

ligne 20 La ligne i n'est pas un mot et on renvoie `false`.

ligne 26 L'appel récursif à `bt` a renvoyé `true`, et `grid` contient donc une solution, et on renvoie `true`.

ligne 28 On a essayé sans succès toutes les façons de remplir la case (i, j) et on renvoie `false`. Il n'y a pas de solution complétant la grille, car sinon il y aurait un caractère `c` pour lequel `bt(i,j+1)` aurait renvoyé `true`.

Dès lors, la méthode `solve` renvoie `true` si et seulement si la grille vide a pu être complétée en une solution.

Question 3 Modifier le code de la figure 1 pour trouver *toutes* les solutions et renvoyer leur nombre.

Correction : On ne renvoie plus de booléen (`bt` a maintenant le type `void`) et on ne s'arrête plus à la première solution trouvée.

```
int count = 0;

void bt(int i, int j) {
  if (i == 5) {
    if (checkColumns()) count++;
    return;
  }
  if (j == 5) {
    String s = ""+grid[i][0]+grid[i][1]+grid[i][2]+grid[i][3]+grid[i][4];
    if (!words.contains(s))
      return;
    bt(i + 1, 0);
    return;
  }
```

```
1  class TrieAlpha {
2    boolean word;
3    TrieAlpha[] branches; // de taille 26
4
5    TrieAlpha() {
6      this.word = false;
7      this.branches = new TrieAlpha[26];
8    }
9
10   boolean contains(String s) {
11     TrieAlpha t = this;
12     for (int i = 0; i < s.length(); i++) {
13       t = t.branches[s.charAt(i) - 'a'];
14       if (t == null) return false;
15     }
16     return t.word;
17   }
18
19   void add(String s) { ... }
20 }
```

Figure 2 – Arbre de préfixes.

```
    for (char c = 'a'; c <= 'z'; c++) {
      grid[i][j] = c;
      bt(i, j+1);
    }
  }

  int count() {
    bt(0, 0);
    return count;
  }
```

Amélioration. Malheureusement, l'approche ci-dessus n'est pas assez efficace et elle ne permet pas de trouver une solution dans un temps raisonnable, sans même parler de les compter toutes. Pour parvenir à une meilleure solution, on va commencer par stocker le dictionnaire dans un *arbre de préfixes* plutôt qu'une table de hachage. La figure 2 contient le code Java d'une classe `TrieAlpha` pour des arbres de préfixes contenant des mots formés exclusivement avec les caractères `'a'` à `'z'`. Le branchement est réalisé par un tableau `branches` de taille 26 : la première case pour `'a'`, la deuxième pour `'b'`, etc. Le booléen `word` indique la présence du mot correspondant à ce nœud. La méthode `contains` teste la présence du mot `s` dans l'arbre `this`. La méthode `add` ajoute le mot `s` dans l'arbre `this`.

Question 4 Représenter l'arbre de préfixes contenant les trois mots `apart`, `apple` et `apply`.

Correction :

```
     false
     a |
     false
     p |
     false
 a /  \ p
false false
r |   l |
false false
t |   e |\ y
true true true
```

Question 5 Donner le code de la méthode `add`.

Correction : C'est tout à fait analogue à ce qui a été vu en cours, la seule différence étant la nature du branchement.

```
void add(String s) {
  TrieAlpha t = this;
  for (int i = 0; i < s.length(); i++) { // invariant t != null
    int c = s.charAt(i) - 'a';
    if (t.branches[c] == null)
      t.branches[c] = new TrieAlpha();
    t = t.branches[c];
  }
  t.word = true;
}
```

Dans la suite, on suppose avoir construit un arbre de préfixes `twords` contenant les N mots de 5 lettres, comme ceci :

```
TrieAlpha twords = new TrieAlpha();
for (String w: words) twords.add(w);
```

Question 6 Quelle est la complexité, en fonction de N, de la recherche d'un mot dans l'arbre `twords` ?

Correction : On fait au plus cinq itérations de la boucle `for` de la méthode `contains`. Chaque itération effectue un nombre constant d'opérations. La complexité est donc $O(1)$. Elle ne dépend pas de N.

Question 7 Étant donné que tous les mots ont exactement 5 lettres, expliquer comment la classe `TrieAlpha` pourrait être simplifiée. On ne demande pas de réécrire le code Java, mais seulement de détailler les modifications à apporter.

```java
static boolean solve(TrieAlpha[] col, TrieAlpha row, int i, int j) {
  if (i == 5)
    return true;
  if (j == 5)
    return solve(col, twords, i+1, 0);
  for (char c = 'a'; c <= 'z'; c++) {
    int k = c - 'a';
    ...
  }
  return false;
}

static boolean solve() {
  return solve(new TrieAlpha[] { twords, twords, twords, twords, twords },
               twords, 0, 0);
}
```

FIGURE 3 – Une approche efficace.

Correction : On peut tout simplement supprimer le booléen `word`. L'idée est que toutes les branches ont la longueur 5 et que chaque feuille représente un mot.

- Dans `contains`, on commence par s'assurer que `s` est bien de longueur 5, sans quoi on renvoie `false`. Puis on remplace `return t.word` par `return true` (ligne 16).
- Dans `add`, on supprime la ligne `t.word = true;`.

Une approche efficace pour les mots croisés parfaits. Pour construire une grille 5×5 de mots croisés parfaits, on va conserver le principe du rebroussement, comme dans la figure 1, mais en se servant maintenant de l'arbre de préfixes. On continue de remplir la grille de la gauche vers la droite, et du haut vers le bas, avec deux indices i et j. À chaque instant, les lignes complètes forment des mots du dictionnaire, les colonnes forment des préfixes de mots du dictionnaire et la ligne courante i forme un préfixe de mots du dictionnaire. Ces 6 préfixes (les cinq colonnes et la ligne i) sont matérialisés par six nœuds de l'arbre `twords`. Voici une situation avec $i = 2$ et $j = 3$, où les six préfixes sont illustrés par des flèches :

				$j = 3$	
	r	o	u	g	h
	a	p	p	l	e
$i = 2$	t	e	p→	↓	↓
	↓	↓	↓		

L'algorithme consiste alors à considérer, pour la case (i, j), tous les caractères qui forment des branches *à la fois* pour la ligne i (ici le préfixe `tep`) et la colonne j (ici le préfixe `gl`). La figure 3 contient le code (partiel) d'une méthode `solve` qui réalise cet algorithme. Le tableau `col` contient les cinq nœuds de `twords` correspondant aux cinq colonnes et `row` contient le nœud de `twords` correspondant à la ligne courante.

Question 8 Illustrer les six nœuds de l'arbre `twords` correspondant à l'exemple donné ci-dessus.

Correction :

```
                  .
    g/     h/    o|     r\    t\   u\
    .      .     .      .     .     .
   l|     e|     p|     a|    e|    p|
 col[3] col[4]   .      .     .     .
  ...    ...     e|     t|    p|    p|
               col[1] col[0] row  col[2]
                ...    ...   ...   ...
```

Question 9 Compléter le code de la méthode `solve` (environ 7 lignes de code).

Correction : On progresse uniquement si la branche existe dans les deux arbres `col[j]` et `row` :

```
if (col[j].branches[k] == null || row.branches[k] == null) continue;
grid[i][j] = c;
TrieAlpha save = col[j];
col[j] = col[j].branches[k];
if (solve(col, row.branches[k], i, j+1))
  return true;
col[j] = save;
```

On prend soin de sauvegarder, puis restaurer, l'arbre `col[j]`.

Question 10 On cherche maintenant à construire des grilles de mots croisés de taille 8×8 contenant possiblement des cases noires, à partir d'un dictionnaire de mots de 1 à 8 lettres. Expliquer comment adapter l'approche ci-dessus. On ne demande pas d'écrire le code Java.

Correction : L'idée consiste à considérer une case noire comme une lettre comme les autres.

1. On ajoute un 27-ième caractère, représentant une case noire. (Le plus simple est de prendre le caractère `'{'` qui vient juste après `'z'` dans le jeu de caractères.) On adapte la classe `TrieAlpha` pour 27 caractères.
2. On construit tous les "mots" de 8 lettres, contenant ou non une case noire : `amandier`, `lard{une`, etc.
3. On applique l'algorithme ci-dessus pour trouver une grille 8×8 parfaite.

```
interface Enum {
  boolean hasNext();
  int next();
}

class Einterval implements Enum {
  private int current, end;
  Einterval(int start, int end) { this.current = start; this.end = end; }
  boolean hasNext() { return this.current < end; }
  int next() { return this.current++; }
}

class Test {
  static void print(Enum e) {
    while (e.hasNext()) { System.out.print(e.next() + "␣"); }
    System.out.println();
  }
}
```

FIGURE 4 – Énumérations.

2 Énumérations

Dans la bibliothèque standard Java, le parcours des éléments d'une collection se présente le plus souvent sous la forme d'une boucle ressemblant à ceci :

```
while (o.hasNext()) {
  T x = o.next();
  ...
}
```

L'objet `o` fournit deux méthodes : une méthode `hasNext` qui indique s'il reste des éléments à parcourir ; et une méthode `next` qui renvoie le prochain élément du parcours. (D'ailleurs, quand on utilise la boucle `for` de Java sous sa forme `for (T x: c)...`, on se retrouve à faire exactement ceci pour un certain objet `o` construit à partir de `c`.)

Dans ce problème, on étudie de tels objets, appelés *énumérateurs*. La figure 4 contient l'interface `Enum` que nous allons leur donner, avec les deux méthodes `hasNext` et `next`. On se limite ici à des énumérations d'entiers. Les règles du jeu sont les suivantes :

- On peut appeler la méthode `hasNext` quand on veut, autant de fois qu'on le souhaite. Si on l'appelle deux fois de suite (*i.e.*, sans appeler `next` entre les deux), le résultat doit être identique.
- Pour pouvoir appeler la méthode `next`, il faut avoir appelé juste avant la méthode `hasNext` et que celle-ci ait renvoyé `true`. En particulier, on ne doit pas appeler deux fois de suite la méthode `next` (*i.e.*, sans appeler `hasNext` entre les deux).

La figure 4 contient également un exemple `Einterval` de classe qui implémente l'interface `Enum` et une méthode statique `Test.print` pour imprimer tous les éléments d'une énumération `e`. La classe `Einterval` correspond à l'énumération d'un intervalle d'entiers. Ainsi, `Test.print(new Einterval(0, 10))` va afficher `0 1 2 3 4 5 6 7 8 9` sur une ligne.

Question 11 Proposer une classe `Ethen` de la forme

```
class Ethen implements Enum {
  ...
  Ethen(Enum x, Enum y) { ... }
  ...
}
```

telle que `new Ethen(x, y)` énumère les éléments de `x` puis les éléments de `y`. Ainsi, la commande `Test.print(new Ethen(new Einterval(0, 4), new Einterval(10, 14)))` doit afficher la ligne `0 1 2 3 10 11 12 13`.

Correction :

```
class Ethen implements Enum {
  private Enum x, y;
  Ethen(Enum x, Enum y) { this.x = x; this.y = y; }
  boolean hasNext() { return this.x.hasNext() || this.y.hasNext(); }
  int next() { if (this.x.hasNext()) return this.x.next(); return this.y.next(); }
}
```

Note : Les règles du jeu sont bien respectées !

Question 12 Proposer une classe `Ereverse` de la forme

```
class Ereverse implements Enum {
  ...
  Ereverse(Enum e) { ... }
  ...
}
```

telle que `new Ereverse(e)` énumère les éléments de `e` dans l'ordre inverse. On suppose que l'énumération `e` est finie. Ainsi, la commande `Test.print(new Ereverse(new Einterval(0, 10)))` doit afficher la ligne `9 8 7 6 5 4 3 2 1 0`.

Correction : On met tous les éléments de `e` sur une pile (d'où la nécessité que `e` soit finie).

```
class Ereverse implements Enum {
  private Stack<Integer> st = new Stack<>();
  Ereverse(Enum e) { while (e.hasNext()) st.push(e.next()); }
  boolean hasNext() { return !st.isEmpty(); }
  int next() { return st.pop(); }
}
```

Question 13 Écrire une méthode `static int compare(Enum x, Enum y)` qui compare lexicographiquement les deux énumérations `x` et `y`, c'est-à-dire qui renvoie

- 0 si `x` et `y` énumèrent exactement les mêmes entiers ;
- 1 si `x` et `y` coïncident sur un préfixe puis, soit le prochain élément de `x` est plus grand que le prochain élément de `y`, soit `x` a un prochain élément mais pas `y` ;

- -1 sinon.

On demande de ne *pas* utiliser une méthode récursive, au risque de faire déborder la pile d'appels, mais d'utiliser une boucle `while`.

Correction :

```
static int compare(Enum x, Enum y) {
  while (x.hasNext() || y.hasNext()) {
    if (!x.hasNext()) return -1;
    if (!y.hasNext()) return 1;
    int c = Integer.compare(x.next(), y.next());
    if (c != 0) return c;
  }
  return 0;
}
```

Question 14 Donner un exemple *réaliste* d'énumération infinie, c'est-à-dire où la méthode `hasNext` renvoie systématiquement `true`.

Correction : Un générateur pseudo-aléatoire.

Question 15 On propose la classe suivante pour énumérer les nombres de Fibonacci, c'est-à-dire la suite définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$.

```
class Efib implements Enum {
  private int a = 0, b = 1;
  boolean hasNext() { return true; }
  int next() { b = b+a; a = b-a; return b-a; }
}
```

Malheureusement, la capacité du type `int` finit par être dépassée, au-delà de quoi les valeurs renvoyées par `next` ne sont plus correctes. Proposer une modification de ce code pour que l'énumération s'arrête sur le dernier nombre de Fibonacci que le type `int` permet de représenter.

Correction : Il suffit de remplacer `return true`, dans `hasNext`, par `return a >= 0;`. En effet, la méthode `next` renvoie systématiquement la valeur de `a` — même s'il faut s'en persuader ! — et remplace `a` par `b` et `b` par `a+b`. Au premier débordement arithmétique, la valeur de `b` va devenir négative, car on ne fait qu'une addition. On peut alors encore renvoyer `a`, qui est le dernier nombre de Fibonacci représentable, puis `a` va prendre la valeur de `b`, qui est négative, et `hasNext` va alors renvoyer `false`.

```
class Sieve implements Enum {
  private Enum numbers = new Einterval(2, Integer.MAX_VALUE);
  boolean hasNext() {
    return this.numbers.hasNext();
  }
  int next() {
    int p = this.numbers.next();
    this.numbers = new Efilter(p, this.numbers);
    return p;
  }
}

class Efilter implements Enum {
  private int p;
  private Enum e;
  ...
  Efilter(int p, Enum e) { this.p = p; this.e = e; ... }
  boolean hasNext() { ... }
  int next() { ... }
}
```

FIGURE 5 – Crible d'Ératosthène.

Crible d'Ératosthène. On souhaite maintenant construire une énumération des nombres premiers, sur le principe du crible d'Ératosthène et sans se soucier d'efficacité. La figure 5 contient le code d'une classe `Sieve` qui réalise cette idée. Un appel à `Test.print(new Sieve())` va afficher tous les nombres premiers, dans l'ordre croissant. La classe `Sieve` utilise une classe `Efilter` qui filtre une énumération `e` (second argument du constructeur) en supprimant les multiples de `p` (premier argument du constructeur).

Question 16 Représenter la valeur de l'objet `e` en mémoire après les trois lignes de code suivantes :

```
Enum e = new Sieve();
if (e.hasNext()) System.out.println(e.next()); // affiche 2
if (e.hasNext()) System.out.println(e.next()); // affiche 3
```

Correction :

```
 +-+      +---------+
e|o+----> | Efilter |
 +-+      |   p=3   |   +---------+
          |   e=o---+-->| Efilter |
          +---------+   |   p=2   |    +-----------+
                        |   e=o---+--> | Einterval |
                        +---------+    | current=5 |
                                       | end=MAX_V |
                                       +-----------+
```

Note : possiblement, l'objet `Einterval` peut avoir un champ `current` qui ne vaut que 4 pour l'instant ; cela dépend de la façon dont la classe `Efilter` est programmée (question suivante).

Question 17 Compléter le code de la classe `Efilter`.

Indications : (1) bien relire les règles du jeu ; (2) on pourra ajouter un champ à la classe.

Correction : On ajoute un champ `next` contenant le prochain entier à renvoyer, ou `null` si l'énumération est terminée. C'est `hasNext` qui se charge de fixer la valeur de ce champ.

```
class Efilter implements Enum {
  private final int  p;
  private final Enum e;
  private Integer next; // le prochain élément à renvoyer, ou null

  Efilter(int p, Enum e) { this.p = p; this.e = e; this.next = null; }

  boolean hasNext() {
    if (this.next != null) return true;
    while (this.e.hasNext()) {
      int n = e.next();
      if (n % p != 0) { this.next = n; return true; }
    }
    return false;
  }

  int next() {
    assert(this.next != null); // on doit être passé par hasNext()
    int n = this.next;
    this.next = null;
    return n;
  }
}
```

En particulier, si `hasNext` a renvoyé `true`, alors le champ `next` n'est pas `null`.

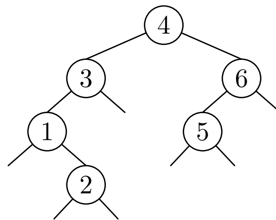
```
class Bintree {
  int value;
  Bintree left, right;
}

class Einorder implements Enum {
  private Stack<Bintree> st = new Stack<>();
  Einorder(Bintree b) { push(b); }
  private void push(Bintree b) {
    while (b != null) {
      this.st.push(b);
      b = b.left;
    }
  }
  boolean hasNext() {
    return !this.st.isEmpty();
  }
  int next() {
    Bintree b = this.st.pop();
    push(b.right);
    return b.value;
  }
}
```

FIGURE 6 – Parcours infixe d'un arbre binaire.

Parcours infixe d'un arbre binaire. Dans cette dernière partie, on considère l'énumération des éléments d'un arbre binaire selon un parcours infixe (d'abord le sous-arbre gauche, puis la racine, puis le sous-arbre droit). La figure 6 contient une classe `Bintree` pour des arbres binaires et une classe `Einorder` qui fournit une énumération correspondant au parcours infixe.

Question 18 Soit `b` l'arbre binaire ci-dessous et `e` l'énumération construite avec `new Einorder(b)`.



On exécute alors `Test.print(e)`, ce qui affiche `1 2 3 4 5 6`. Illustrer le contenu de la pile `e.st` à chaque étape de cette énumération, c'est-à-dire après la construction de `e` et après chaque appel à `e.next()` (7 lignes au total).

action	contenu de la pile `e.st` écrit de gauche à droite (fond à gauche, sommet à droite)
`e = new Einorder(b)`	...
`e.next()`	...
⋮	⋮

Correction :

action	contenu de la pile `e.st` (fond à gauche, sommet à droite)
`e = new Einorder(b)`	
`e.next()`	
`e.next()`	
`e.next()`	
`e.next()`	
`e.next()`	
`e.next()`	

Question 19 Justifier que le code de la classe `Einorder` ne provoque ni `NullPointerException` ni `EmptyStackException`. En quoi le caractère `private` du champ `st` et de la méthode `push` est-il important ?

Correction : Le champ `st` contient un objet `Stack` créé ligne 7 et donc les appels aux méthodes de `this.st` ne provoquent pas `NullPointerException`.

Pour ce qui est des accès aux champs de `b` aux lignes 12, 20 et 21, on invoque l'invariant suivant : les arbres contenus dans `st` ne sont jamais `null`. Cela est clairement garanti par la méthode `push` (ligne 11), la seule à ajouter des arbres dans `st`.

Pour ce qui est de `EmptyStackException`, on sait que `next()` n'est appelée qu'après `hasNext` et avec un résultat `true` renvoyé par `hasNext`. Dès lors, la pile `st` n'est jamais vide ligne 19.

Le caractère `private` de `st` est indispensable pour maintenir l'invariant que les arbres de `st` ne sont pas `null`. Le caractère `private` de `push` n'est pas essentiel à cet invariant, mais il est important pour garantir que l'énumération est bien celle de l'arbre `b`.

Question 20 Soit `b` un arbre binaire contenant N nœuds et `e` l'énumération `new Einorder(b)`. Montrer que, pendant `Test.print(e)`, chaque nœud de `b` est ajouté une fois et une seule à la pile `e.st`. En déduire que la complexité de `Test.print(e)` est $O(N)$. Que dire de la complexité de `e.hasNext()` et de `e.next()` ?

Correction : On commence par remarquer que tout nœud est ajouté à `st` par la ligne 11 du code, soit comme le fils gauche d'un nœud qui vient d'être ajouté par `push` (ligne 12), soit comme le fils droit d'un nœud qui vient d'être retiré (ligne 20).

On raisonne alors par récurrence sur la profondeur du nœud dans l'arbre. La racine de l'arbre est ajoutée dans `st` en tout premier lieu, lorsque la constructeur appelle `push`, et ne sera plus jamais ajoutée à `st` car la racine n'est pas le fils d'un autre nœud. Soit `t` un nœud de l'arbre de profondeur > 0 et `p` son parent. Par hypothèse de récurrence, `p` est ajouté une fois et une seule à `st` et donc également retiré une fois et une seule pendant l'énumération. Si `t` est le fils gauche de `p`, alors `t` sera ajouté lorsque `p` sera ajouté ; et si `t` est le fils droit de `p`, alors `t` sera ajouté lorsque `p` sera retiré.

La complexité de `hasNext` est $O(1)$. La complexité de `next` est le temps passé dans `push`, plus des opérations de temps constant. La complexité de l'énumération est donc $O(N)$ (N appels à `hasNext` et `next`) plus le temps total passé dans la méthode `push`. Mais ce dernier est directement proportionnel au nombre de nœuds ajoutés dans `st` (ligne 11), ce qui est $O(N)$ par le résultat précédent. La complexité totale de l'énumération est donc $O(N)$.

La complexité de `next` est donc $O(1)$ amortie.

A Bibliothèque standard Java

`class Stack<E>`	une pile dont les éléments sont de type `E`
`Stack()`	renvoie une nouvelle pile, vide
`void push(E x)`	ajoute l'élément `x` au sommet de la pile
`boolean isEmpty()`	renvoie `true` si et seulement si la pile est vide
`E pop()`	supprime et renvoie l'élément au sommet de la pile lève `EmptyStackException` si la pile est vide

Les trois opérations `push`, `isEmpty` et `pop` s'exécutent en temps constant.

`class HashSet<E>`	un ensemble dont les éléments sont de type `E`
`HashSet()`	renvoie un nouvel ensemble, vide
`void add(E x)`	ajoute l'élément `x`
`void remove(E x)`	supprime l'élément `x`
`boolean contains(E x)`	indique la présence de l'élément `x`
`int size()`	renvoie le cardinal de l'ensemble

On peut parcourir tous les éléments d'un ensemble `s` avec la construction

```
for (E x: s) ...
```

`class String`	le type des chaînes de caractères
`int length()`	la longueur de la chaîne
`char charAt(int i)`	le caractère d'indice `i`, pour $0 \leq$ `i` $<$ `length()`

*   *
*