TME 6 : Arbres lexicaux

Introduction

Ce TME reprend le partiel de l'année 2014-2015 (épreuve individuelle en 2,5 heures) et son système de correction automatique. Vous pourrez donc soumettre vos solutions au fur et à mesure que vous les écrivez pendant le TME et après si besoin est.

Le texte complet du sujet est en ligne et accessible à partir de la page de l'UE.

Saisie textuelle sur un pas smart phone

On cherche à construire les outils pour pouvoir traiter la saisie de mots à partir d'un clavier numérique d'un téléphone portable. Les touches numériques d'un clavier téléphonique regroupent chacune plusieurs lettres. Nous prendrons la configuration décrite à la figure 1.

touche	association
1	' ' (espace)
2	ABC
3	DEF
4	GHI
5	JKL
6	MNO
7	PQRS
8	TUV
9	WXYZ

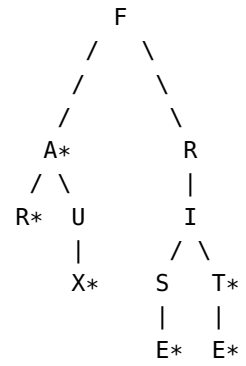
Figure 1: Association touches téléphoniques et groupes de lettres

La saisie classique d'un mot revient à appuyer p fois sur le chiffre correspondant au groupe de la lettre, le p permettant d'avancer dans le groupe. Par exemple le mot «FRITE» nécessite de presser 12 touches : 3 fois la 3, puis 3 fois la 7, puis 3 fois la 4, puis 1 fois la 8 et enfin 2 fois la 3.

La saisie intuitive permet d'éviter ces multiples frappes en sélectionnant uniquement le groupe d'appartenance, ici seulement 5 appuis : 1 fois la touche 3, puis 1 fois la 7, puis 1 fois la 4, puis 1 fois la 8 et 1 fois la 3.

Parmi les différentes combinaisons possibles de lettres des groupes, seulement une petite partie correspond à un vrai mot. Il devient nécessaire d'utiliser un dictionnaire pour vérifier l'existence d'un ou de plusieurs mots, dans ce dernier cas la liste des mots pourra être proposée à l'utilisateur. Enfin dès qu'il n'y a plus qu'une seule possibilité de choix de mot, celui-ci peut-être directement complété et proposé à l'utilisateur.

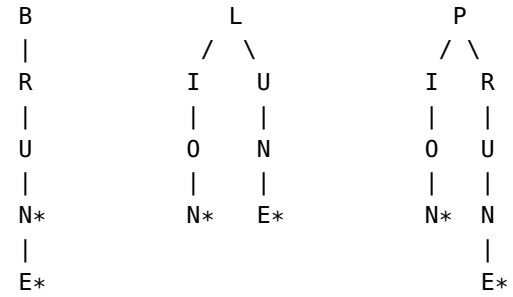
Dans cet exercice, on aura besoin d'un dictionnaire, représenté ici par un arbre lexical non cyclique où chaque nœud contient une lettre et un booléen indiquant si ce nœud peut être la dernière lettre d'un mot, et d'une liste de sous-nœuds. Par exemple à la figure 2 le dictionnaire contenant les mots «FA», «FAR», «FAUX», «FRISE», «FRIT» et «FRITE» est représenté par l'arbre lexical suivant (l'étoile dans un nœud indique une terminaison de mot):




```
11 noeuds et 6 mots : FA, FAR, FAUX, FRISE, FRIT, FRITE
```

Figure 2: Extrait du dictionnaire sous forme d'un arbre lexicographique

La figure 3 montre un autre dictionnaire pour les mots commençant par B, L et P.




```
20 noeuds et 6 mots : BRUN, BRUNE, LION, LUNE, PION, PRUNE
```

Figure 3: Autre extrait du dictionnaire sous forme d'un arbre lexicographique

Ce type d'arbre permet d'optimiser l'occupation mémoire en factorisant les préfixes communs des mots quand il y en a.

Questions

On va chercher à écrire les différentes briques pour ce comportement de saisie intuitive pour l'écriture rapide de textes. Pour cela on aura besoin de l'environnement de saisie, c'est-à-dire de l'association touche de téléphone et groupe de lettres, et d'un dictionnaire permettant de naviguer rapidement dans les mots possibles suite à l'appui d'une ou de plusieurs touches.

Attention certains fragments de programmes sont donnés pour vous indiquer la manière dont certains tests pourront se dérouler et s'enchaîner. Il faut néanmoins savoir que les affichages sont là à titre indicatif pour faciliter la compréhension des questions posées et ne se retrouveront pas dans les tests du correcteur.

Pour les environnements on se donne les interfaces et classes suivantes pour permettre de construire ensuite des environnements associant une «touche d'un clavier de téléphone» à une suite de lettres:

IEnv.java

```java
package pobj.partiel2014nov;

/** Associe à des entiers (les touches) l'ensemble des lettres sur la touche */
```

```java
public interface IEnv {
      /**
       * Rend les lettres associées à une touche donnée.
       *
       * @param touche un entier compris entre 1 et 9 inclus
       * @return une chaine de caractères contenant les lettres associées à cette
       * touche.
       * @throws MauvaiseTouche si l'entier n'est pas compris entre 1 et 9
       */
      String get(int touche) throws MauvaiseTouche;

      /**
       * Associe un ensemble de lettres à une touche; si la touche est déjà
       * associée a des lettres, écrase l'ancienne association.
       *
       * @param touche
       * un entier compris entre 1 et 9 inclus
       * @param lettres
       * une chaine de caractères contenant les lettres à associer à
       * cette touche.
       * @throws MauvaiseTouche
       * si l'entier n'est pas compris entre 1 et 9
       */
      void set(int touche, String lettres) throws MauvaiseTouche;
}
```

MauvaiseTouche.java

```java
package pobj.partiel2014nov;

public class MauvaiseTouche extends Exception {

}
```

6.1 Question 1

On cherche maintenant à écrire une classe `Env` pour les environnements qui respecte l'interface `IEnv`.

Les méthodes des classes qui implantent l'interface `IEnv` doivent vérifier la propriété suivante : $\forall i \in [1..9]$ l'appel de `set(i,s)` suivi de `get(i)` retourne `s2` une chaîne équivalente à `s` (au sens de `equals`). C'est-à-dire que `s.equals(s2)` retourne `true`.

On ne s'intéresse qu'aux touches de 1 à 9. Le set d'une autre touche déclenche l'exception `MauvaiseTouche`, idem pour le get.

Le set crée une nouvelle association touche - chaîne dans l'environnement sans se soucier s'il en existait une précédemment.

Ecrivez une classe `pobj.partiel2014nov.Env` qui implante l'interface `IEnv` selon cette spécification.

6.2 Question 2

On cherche à construire l'environnement spécifique donné à la figure 1. On ne s'intéressera qu'aux touches allant de 1 à 9, les autres ne sont pas à être définies. La chaîne associée à la touche 1 est équivalente à la chaîne contenant un unique espace (" "), celle à la touche 2 est "ABC", ..., celle à la touche 9 est "WXYZ".

On définira la méthode statique `creerEnv` sans paramètre qui construira une instance de la classe Env, qui implante l'interface IEnv, correspondant à l'environnement de la figure 1.

Ecrivez une classe `EnvFactory` du paquetage `pobj.partiel2014nov` dont le squelette suit :

EnvFactory.java

```java
package pobj.partiel2014nov;

public class EnvFactory {
	public static IEnv creerEnv() {
	// TODO
	}
}
```

6.3 Question 3

On cherche maintenant à écrire les classes pour construire un dictionnaire sous forme d'un arbre lexicographique acyclique. Un dictionnaire sera représenté par une liste de nœuds respectant l'interface INoeud suivante:

INoeud.java

```java
package pobj.partiel2014nov;

import java.util.List;

public interface INoeud {
	/**
	 * @return la lettre associée à ce noeud de l'arbre.
	 */
	char getLettre();

	/**
	 * @return la liste des fils de ce noeud
	 */
	List<INoeud> getFils();

	/**
	 * @return true si le noeud est marqué (c'est la fin d'un mot) ou false sinon.
	 */
	boolean isMarque();

	/**
	 * Met à jour la propriété marqué ou non du noeud.
	 * @param isMarque la nouvelle valeur du marquage.
	 */
	void setMarque (boolean isMarque);
}
```

⇒ Construisez la classe Noeud une implémentation de INoeud.

⇒ Complétez la classe NoeudFactory ci-dessous.

NoeudFactory.java

```java
package pobj.partiel2014nov;

import java.util.List;

public class NoeudFactory {

	/**
	 * Fabrique un nouveau noeud de l'arbre avec les caractéristiques souhaitées.
	 * @param lettre la lettre qui étiquette ce noeud
	 * @param fils la liste des fils de ce noeud
```

```
	 * @param isMarked vrai si le noeud est marqué (c'est la fin d'un mot)
	 * @return un noeud nouvellement crée
	 */
	public static INoeud createNoeud (char lettre, List<INoeud> fils, boolean isMarked) {
		// TODO
	}
}
```

6.4 Question 4

Un dictionnaire va correspondre à un arbre lexicographique muni d'opérations d'ajout de mots et de recherche d'appartenance d'un mot au dictionnaire.

On fournit la classe abstraite `DicoAbs` réalisant une partie des traitements utiles.

DicoAbs.java

```
package pobj.partiel2014nov;
import java.util.List;
import java.util.ArrayList;

public abstract class DicoAbs {

	private List<INoeud> dico;

	public DicoAbs() {
		this(new ArrayList<INoeud>());
	}

	public DicoAbs(List<INoeud> dico) {
		this.dico = dico;
	}

	public List<INoeud> getDico() {
		return dico;
	}

	/**
	 * Recherche un noeud portant une lettre particulière au sein d'une liste de noeuds.
	 * @param liste la liste de noeuds
	 * @param lettre la lettre recehrchée
	 * @return le noeud de la liste qui porte la bonne lettre ou null si on n'es trouve
	    pas.
	 */
	public static INoeud chercheNoeud(List<INoeud> liste, char lettre) {
		for (INoeud n : liste)
			if (n.getLettre() == lettre)
				return n;
		return null;
	}

	/**
	 * Crée un arbre (une chaine en pratique, un seul fils par noeud) qui correspond
	 * au mot argument à partir de la lettre "index". Sa feuille est marquée.
	 * @param s un mot à stocker sous forme d'arbre
	 * @param index l'index de la première lettre à représenter, compris entre 0 (on veut
	    le mot entier) et s.length-1 (que la dernière lettre)
	 * @return un arbre contenant (s.length-pos) noeuds représentant la fin de s.
	 */
	public INoeud creerSuffixe(String s, int index) {
```

```
            INoeud suffixe;
            int size = s.length();
            // l'extrémité est un noeud marqué
            suffixe = NoeudFactory.createNoeud(s.charAt(size-1), new ArrayList<INoeud>(),
                true);
            // on itère depuis l'avant dernière lettre vers le début du mot, jusque index.
            for (int i = size - 2; i >= index; i--) {
                  ArrayList<INoeud> al = new ArrayList<INoeud>();
                  al.add(suffixe);
                  // un nouveau noeud pour la lettre courante, de fils le suffixe déjà  crée
                  suffixe = NoeudFactory.createNoeud(s.charAt(i), al, false);
            }
            return suffixe;
      }

      /**
       * Ajoute un mot au dictionnaire ; crée ou modifie les noeuds existants du
           dictionnaire.
       * @param mot
       */
      public void ajoute(String mot) {
            List<INoeud> l = dico;
            for (int i = 0; i < mot.length(); i++) {
                  INoeud n = chercheNoeud(l, mot.charAt(i));
                  if (n==null) {
                        // pas trouvé : créer
                        l.add(creerSuffixe(mot, i));
                        return;
                  }
                  if (i == mot.length()-1) {
                        // on assure que le dernier noeud du mot est marqué
                        n.setMarque(true);
                  } else {
                        // la lettre a été trouvée, on se décale sur les fils
                        l = n.getFils();
                  }
            }
      }

      /**
       * Cherche un mot dans le dictionnaire.
       * @param s un mot à chercher
       * @return true si le mot est présent dans le dictionnaire
       */
      public abstract boolean appartient(String s);

}
```

Un dictionnaire est représenté par une liste de nœuds, chaque nœud contient une lettre et les fils correspondant aux suites des préfixes de mots déjà rencontrés. La classe possède deux constructeurs : un constructeur sans argument créant un dictionnaire vide, et un constructeur prenant en argument une liste de nœuds créant un dictionnaire de ces entrées.

Ce type de représentation essaie de partager au maximum les préfixes communs des mots du dictionnaires. Lorsqu'un nouveau mot est ajouté, il faut alors parcourir le dictionnaire et le mot tant que l'on rencontre un chemin correspondant au mot à ajouter tant qu'on le peut, puis créer la nouvelle branche de la fin du mot (suffixe) que l'on ne peut pas partager.

La méthode ajoute prend un mot en paramètre et l'ajoute au dictionnaire, en enchaînant les étapes décrites précédemment. Dans la version donnée dans l'énoncé, si un mot que l'on ajoute existe déjà, il ne se passe rien et le dictionnaire est inchangé.

La méthode creerSuffixe permet de créer l'arbre lexical du suffixe d'un mot à partir d'un certain index dans la chaîne de caractères. Celui-ci est alors raccroché aux fils du nœud correspondant au dernier caractère du préfixe commun.

La méthode statique chercheNoeud prend une liste d'INoeud et une lettre, et retourne le nœud correspond à cette lettre s'il existe ou null sinon.

⇒ Ecrire une classe concrète Dico qui hérite de la classe abstraite DicoAbs, et qui implante la méthode abstraite `appartient` de celle-ci. Cette méthode retourne true si le mot passé en argument fait partie du dictionnaire et false sinon. Enfin il est demandé de définir deux constructeurs `public Dico()` et `public Dico(List<INoeud> dico)` permettant de construire respectivement un dictionnaire vide et un dictionnaire prenant une liste de nœuds comme entrées.

6.5 Question 5

On cherche à construire le dictionnaire de la figure 2 à partir de la classe Dico. Pour cela on construit tout d'abord un dictionnaire vide sur lequel on ajoute tous les mots de la figure 2. On définira la méthode statique creerDico sans paramètre qui construira une instance de la classe Dico correspondant au dictionnaire de la figure 2.

Question5.java

```
package pobj.partiel2014nov;
import java.util.ArrayList;

public class Question5 {
      /**
       * Construit le dictionnaire défini figure 2 de l'énoncé.
       * @return le dictionnaire construit
       */
  public static Dico creerDico() {
      //TODO
  }
}
```

6.6 Question 6

On cherche à compter le nombre de nœud et le nombre de mots potentiels d'un INoeud. On se donne la classe Compte suivante qu'il faudra compléter.

Compte.java

```
package pobj.partiel2014nov;

import java.util.List;

public class Compte {
      /**
       * Calcule le nombre de noeuds dans l'arbre dont "in" est la tête.
       * @param in un noeud, tête de l'arbre à dénombrer
       * @return le nombre de noeuds dans l'arbre dont in est la tête.
       */
      public static int nombreNoeud(INoeud in){
      // TODO
      }

      /**
       * Calcule le nombre de mots dans l'arbre dont "in" est la tête.
       * @param in un noeud, tête de l'arbre à dénombrer
       * @return le nombre de mots dans l'arbre dont in est la tête.
```

```
     */
    public static int nombreMot(INoeud in){
    //TODO
    }
}
```

La méthode statique nombreNoeud compte l'ensemble de nœuds du paramètre passé, c'est-à-dire lui et ses descendants. La méthode statique nombreMot compte l'ensemble des nœuds du paramètre passé correspondant à des mots.

Par exemple en construisant le dictionnaire de la figure 2 et en récupérant le nœud de la lettre 'F', le fragment de programme suivant:

```
Dico d6 = Question5.creerDico();
INoeud x = DicoAbs.noeudDe(d6.getDico(), 'F');
System.out.println(Compte.nombreNoeud(x) + " -- " + Compte.nombreMot(x));
```

devrait afficher :

```
11 -- 6
```

⇒ Complétez le squelette donné de la classe Compte en définissant les corps des méthodes nombreNoeud et nombreMot.

6.7 Question 7

On va étendre ces mesures au niveau des dictionnaires, pour pouvoir compter le nombre de nœuds et le nombre de mots d'un dictionnaire. Pour cela on va étendre les fonctionnalités de la classe Dico. On se donne l'interface Comptable suivante:

Comptable.java

```
package pobj.partiel2014nov;

public interface Comptable {
    /**
     * @return le nombre de noeuds total dans la représentation
     */
   int nombreNoeud();
    /**
     * @return le nombre total de mots stockés
     */
   int nombreMot();
}
```

Les méthodes `nombreNoeud` et `nombreMot` comptent respectivement le nombre de nœuds et le nombre de mots (comme par exemple dans un dictionnaire).

Ecrivez une classe `DicoCompte` qui hérite de `Dico` et implante l'interface `Comptable`. Les implantations des méthodes nombreNoeud et nombreMot comptent, comme leur nom l'indique, le nombre de nœuds et le nombre de mots du dictionnaire complet. Enfin il est demandé de définir un constructeur `DicoCompte(List<INoeud> dico)` permettant de construire un dictionnaire prenant une liste de nœuds comme entrées.

Si la variable `dicofig3` correspond bien au dictionnaire de la figure 3, alors l'appel suivant :

```
System.out.println(dicofig3.nombreNoeud()+ " ... " +dicofig3.nombreMot());
```

devrait afficher :

```
20 ... 6
```

6.8 Question 8

On cherche à redéfinir une méthode toString sur les dictionnaires qui construit une chaîne de caractères contenant tous les mots séparés par des retours à la ligne (caractère "
n"). Le nombre de caractère saut de ligne doit être égal au nombre de mots du dictionnaire (au sens de DicoCompte).

Question8.java

```java
package pobj.partiel2014nov;

public class Question8 {

	/**
	 * Rend l'ensemble des mots dans l'arbre dont le noeud n est la tête, un mot par ligne.
	 * @param n la tête d'un arbre
	 * @return une String contenant tous les mots de l'arbre séparés par des \n
	 */
	public static String toString(INoeud n) {
		//TODO
	}

	/**
	 * Rend l'ensemble des mots dans un dictionnaire donné, un mot par ligne.
	 * @param d le dictionnaire qu'on souhaite afficher
	 * @return une String contenant tous les mots du dictionnaire séparés par des \n
	 */
	public static String toString(Dico d) {
		// TODO
	}

}
```

On pourra la tester de la manière suivante avec ce fragment de programme (où `dico` représente le dictionnaire de la figure 3):

```
String res = Question8.toString(dico);
System.out.print(res);
System.out.println("---");
```

qui produira l'affichage suivant :

```
BRUN
BRUNE
LION
LUNE
PION
PRUNE
---
```

⇒ Complétez le squelette de la classe Question8 en définissant les corps des deux méthodes toString . Pour la méthode toString traduisant un INoeud en chaîne de caractères il est conseillé d'écrire une méthode auxiliaire prenant en paramètre supplémentaire le préfixe correspondant aux lettres déjà rencontrées avant d'arriver sur cet INoeud.

6.9 Question 9

On s'intéresse maintenant à simuler la saisie intuitive, c'est-à-dire à pouvoir limiter le nombre de mots possibles en fonction des touches appuyées. On définit pour cela l'interface Saisissable suivante:

Saisissable.java

```
package pobj.partiel2014nov;
import java.util.List;

interface Saisissable {
      /**
       * Filtre parmi les noeuds de alc ceux qui pourraient correspondre) à la touche pressé
           e.
       * @param alc une liste de noeuds : les complétions possibles
       * @param touche la touche préssée, comprise entre 1 et 9
       * @return le sous ensemble de alc dont la lettre correspond à la touche pressée selon
           l'environnement fourni.
       * @throws MauvaiseTouche si la touche n'est pas entre 1 et 9
       */
   List<INoeud> uneTouche(List<INoeud> alc, int touche) throws MauvaiseTouche;

   /**
    * Rend les noeuds qui permettent de compléter la séquence de touches donnée.
    * @param alc les têtes de mots du dictionnaire
    * @param touches une séquence de touches ayant été pressées
    * @return une liste de noeuds compatibles avec la séquence de touches pressée.
       * @throws MauvaiseTouche si la touche n'est pas entre 1 et 9
    */
   List<INoeud> seqTouche(List<INoeud> alc, int[] touches) throws MauvaiseTouche;
}
```

et le squelette de la classe Clavier dont le constructeur prend un environnement en argument:

Clavier.java

```
package pobj.partiel2014nov;

import java.util.List;
import java.util.ArrayList;

public class Clavier implements Saisissable {
  IEnv env;

  public Clavier(IEnv env){ this.env = env; }

  public List<INoeud> uneTouche(List<INoeud> alc, int touche) throws MauvaiseTouche {
      //TODO
  }

  public List<INoeud> seqTouche(List<INoeud> alc, int[] touches) throws MauvaiseTouche {
      // TODO
  }

}
```

La méthode uneTouche prend une liste de nœuds et une touche et retourne la liste des nœuds commençant par une lettre associée à cette touche. Cela correspond à un filtre sur une liste de nœuds.

La méthode seqTouche prend une liste de nœuds et un tableau de touches et itère le processus de filtre de uneTouche sur l'ensemble des fils résultat. C'est une simplification du problème car on ne

tient pas compte des préfixes possibles. Néanmoins il faut alors traiter des listes de nœuds dont plusieurs peuvent avoir la même lettre.

Le fragment de programme suivant où d7 représente le dictionnaire de la figure 3 que l'on enrichit des mots «LIONNE», «LIONS» et «LIONCEAU»:

```
d7.ajoute("LIONNE");
d7.ajoute("LIONS");
d7.ajoute("LIONCEAU");
int[] touches = {5, 4, 6};
List<INoeud>il9 = cv.seqTouche(d7.getDico(), touches);
DicoCompte  d9 = new DicoCompte(il9);
System.out.println(d9.nombreNoeud()+ "*<||>* " + d9.nombreMot());
System.out.println(Question8.toString(d9));
```

devrait afficher :

```
9*<||>* 4
ON
ONNE
ONS
ONCEAU
```

qui correspond après les appuis de 5 (JKL), 4 (GHI) puis 6 (MNO) pour le début «LIO» aux quatre mots encore possibles dans ce dictionnaire.

⇒ Ecrivez une classe Clavier qui implante l'interface Saisissable en complétant les méthodes uneTouche et seqTouche dans le squelette donné.