X2015 — INF411
Contrôle classant / 2 novembre 2016 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les EV2 et EV3. Les ordinateurs, Ipad, dictionnaires électroniques, les tablettes et téléphones portables sont interdits.

L'énoncé est composé de 2 problèmes indépendants, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez 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. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Chemins dans un graphe acyclique

Dans tout ce problème, on considère des graphes orientés finis et *acycliques*, c'est-à-dire qui ne contiennent pas de cycle.

On suppose que les graphes sont réalisés par une classe `Graph<V>`, paramétrée par le type `V` des sommets. On ne précise pas comment cette classe `Graph` est réalisée mais on suppose qu'elle fournit aux moins deux méthodes `vertices` et `successors` :

```
Set<V> vertices()           // renvoie tous les sommets
Set<V> successors(V u)      // renvoie tous les voisins de u
```

En particulier, si `g` est un graphe, on pourra parcourir tous ses sommets avec une boucle de la forme

```
for (V v: g.vertices())
  ...
```

et tous les voisins d'un sommet `u` avec une boucle de la forme

```
for (V v: g.successors(u))
  ...
```

On suppose par ailleurs que le type `V` des sommets peut être utilisé comme type de clés dans des tables de hachage de la classe `HashMap` de Java.

Degré entrant. Le *degré entrant* d'un sommet u dans un graphe est le nombre d'arcs de la forme $v \to u$.

Question 1 Écrire une méthode `HashMap<V, Integer> indegree(Graph<V> g)` qui prend un graphe `g` en argument et renvoie une table donnant pour chaque sommet de `g` son degré entrant.

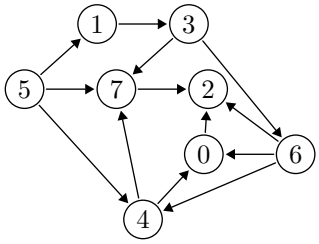
Correction : Pas de difficulté particulière. On commence par remplir la table avec 0 pour chaque sommet et on examine ensuite chaque arc du graphe.

```
HashMap<V, Integer> indegree() {
        HashMap<V, Integer> ind = new HashMap<V, Integer>();
        for (V v: g.vertices())
                ind.put(v, 0);
        for (V v: g.vertices())
                for (V w: g.successors(v))
                        ind.put(w, 1 + ind.get(w));
        return ind;
}
```

Question 2 Montrer que tout graphe orienté acyclique non vide contient au moins un sommet de degré entrant 0.

Correction : Notons $\delta^-(u)$ le degré entrant de u.
Procédons par récurrence sur le nombre $n \geq 1$ de sommets du graphe. Si $n = 1$, c'est immédiat. Sinon, choisissons un sommet u arbitrairement. Si $\delta^-(u) = 0$, c'est terminé. Sinon, considérons le sous-graphe P des sommets v tels que $v \to^\star u$. P ne contient pas u car sinon on aurait un cycle passant par u. Par ailleurs, P est non vide car $\delta^-(u) > 0$. Enfin, P ne contient pas de cycle, car sinon le graphe initial en contiendrait un également. On peut donc appliquer l'hypothèse de récurrence à P ce qui nous donne un sommet v de P de degré entrant 0. C'est également un sommet de degré entrant 0 dans le graphe initial, car tout prédécesseur de v serait aussi dans P.

Tri topologique. Le tri topologique d'un graphe orienté acyclique G est une liste ordonnée L des sommets de G compatible avec la relation d'adjacence dans le sens suivant : si $u \to v$ est un arc de G, alors u apparaît avant v dans la liste L. Si on considère par exemple le graphe suivant



alors $5, 1, 3, 6, 4, 7, 0, 2$ est un tri topologique de ses sommets.

Question 3 Donner un *autre* tri topologique pour le graphe précédent.

Correction : On peut échanger 7 et 0 :

5, 1, 3, 6, 4, 0, 7, 2

Il n'y a pas d'autre solution.

Un algorithme. Pour réaliser le tri topologique d'un graphe acyclique, on peut utiliser l'algorithme suivant :

1. on commence par calculer les degrés entrants de tous les sommets ;
2. on met tous les sommets de degré entrant 0 dans une file ;
3. tant que la file n'est pas vide,
   (a) on retire le premier sommet v de la file et on l'ajoute à la liste résultat,
   (b) pour chaque voisin w de v, on diminue son degré entrant de 1 et on ajoute w à la file s'il atteint 0.

Question 4 Compléter la méthode suivante qui réalise cet algorithme.

```
static LinkedList<V> topologicalSort(Graph<V> g) {
  HashMap<V, Integer> ind = indegree(g);
  LinkedList<V> list = new LinkedList<V>();
  Queue<V> queue = new LinkedList<V>();
  ...
  return list;
}
```

Correction :

```
for (V v: g.vertices())
  if (ind.get(v) == 0)
   queue.add(v);
while (!queue.isEmpty()) {
  V v = queue.poll();
  list.add(v);
  for (V w: g.successors(v)) {
    int d = ind.get(w) - 1;
    ind.put(w, d);
    if (d == 0)
      queue.add(w);
  }
}
```

Question 5 Quelle est la complexité en temps et en espace de cet algorithme ? Justifier.

Correction : La complexité en temps est $O(V + E)$. En effet, chaque sommet est ajouté au plus une fois à la file, lorsque son degré atteint 0. Chaque sommet est examiné au plus une fois, lorsqu'il sort de la file. Et chaque arc est traité au plus une fois, lorsque son sommet origine est traité. Par ailleurs, les opérations sur la file, la liste et la table de hachage sont toutes de temps constant.

La complexité en espace est $O(V)$ car chaque sommet est ajouté au plus une fois à la liste et à la file.

(En fait, on peut montrer que chaque sommet et chaque arc est traité *exactement* une fois, mais ce n'est pas demandé ici.)

```
class Dist {
  final Double d; // null signifie une distance infinie

  Dist() { this.d = null; }

  Dist(double d) { this.d = d; }

  Dist add(double x) { ... }

  boolean lt(Dist that) { ... }
}
```

FIGURE 1 – La classe `Dist`.

Plus courts chemins. On s'intéresse maintenant au calcul des plus courts chemins dans un graphe orienté acyclique, à partir d'un sommet source donné. Comme nous allons le voir, il est possible d'exploiter le caractère acyclique du graphe pour proposer quelque chose de plus efficace que l'algorithme de Dijkstra.

On suppose qu'à chaque arc du graphe est associée une *distance*, la longueur d'un chemin étant la somme des distances des arcs qui constituent ce chemin. Pour simplifier, on suppose ici que la distance est un nombre flottant, du type `double` de Java. On suppose donnée une méthode

```
double distance(V u, V v)
```

dans la classe `Graph`, qui renvoie la distance associée à l'arc `u` $\to$ `v` lorsqu'il existe.

Pour programmer l'algorithme de calcul des plus courts chemins, il sera pratique de représenter une distance non encore connue comme une distance *infinie*, notée ∞ par la suite. Pour cela, on introduit la classe `Dist` figure 1. Lorsque son champ `d` vaut `null`, un objet de la classe `Dist` représente la distance ∞. Sinon, il représente la distance donnée par la valeur du champ `d`. La méthode `add` renvoie l'addition de la distance `this` et de la distance finie `x` passée en argument, avec la convention que $\infty + \texttt{x} = \infty$. La méthode `lt` détermine si la distance `this` est strictement plus petite que la distance `that` passée en argument.

Question 6 Compléter les méthodes `add` et `lt` de la classe `Dist`. Pour la méthode `add`, on notera le caractère `final` du champ `d`. Pour la méthode `lt`, on pourra supposer que l'argument `that` n'est pas `null`.

Correction : Pas de difficulté. On peut renvoyer `this` directement dans le cas $\infty + x$ car les objets de la classe `Dist` sont immuables.

```
Dist add(double x) {
  if (this.d == null)
    return this;
  return new Dist(this.d + x);
}

boolean lt(Dist that) {
  if (this.d == null)
    return false;
  if (that.d == null)
```

```
      return true;
    return this.d < that.d;
  }
```

Calcul des plus courts chemins. Pour calculer tous les plus courts chemins à partir d'un sommet source donné, on peut procéder de la manière suivante. Initialement, on fixe la distance de tous les sommets à ∞, sauf pour la source dont la distance est fixée à 0. Puis on considère tous les sommets *dans l'ordre donné par un tri topologique*. Pour chaque sommet u, on considère tout arc sortant $u \to v$. Si emprunter cet arc améliore la distance actuellement connue pour v, alors on la met à jour.

Question 7 Compléter la méthode ci-dessous pour mettre en œuvre cet algorithme et renvoyer une table donnant la distance du sommet `source` à tout sommet du graphe `g`. Lorsqu'un sommet n'est pas atteignable depuis la source, cette table doit lui associer la distance ∞.

```
static HashMap<V, Dist> shortestPaths(Graph<V> g, V source) {
  HashMap<V, Dist> dist = new HashMap<V, Dist>();
  ...
  return dist;
}
```

Correction :

```
static HashMap<V, Dist> shortestPaths(Graph<V> g, V source) {
  HashMap<V, Dist> dist = new HashMap<V, Dist>();
  for (V v : g.vertices())
    dist.put(v, new Dist());      // distance infinie pour tous les sommets
  dist.put(source, new Dist(0));  // sauf pour la source
  for (V v : topologicalSort(g)) {
    for (V w : g.successors(v)) {
      Dist d = dist.get(v).add(distance(v, w));
      if (d.lt(dist.get(w)))
        dist.put(w, d);
    }
  }
  return dist;
}
```

Question 8 Quelle est la complexité en temps et en espace de cet algorithme ? Justifier.

Correction : Le coût du tri topologique est $O(V + E)$. Dans `shortestPaths`, chaque sommet est examiné une fois dans la première boucle et une fois dans la seconde. Chaque arc est considéré une fois dans la seconde boucle. Par ailleurs, les opérations sur la table de hachage sont toutes de temps constant. La complexité totale est donc $O(V + E)$.

La complexité en espace est celle de la table `dist` et de la liste résultat du tri topologique, donc $O(V)$.

Question 9 L'algorithme ci-dessus nécessite-t-il que les distances soient positives ou nulles ? Justifier.

Correction : Non. Il faut faire la preuve de correction de l'algorithme pour le justifier.

L'argument principal est le suivant : lorsqu'un sommet u est considéré par l'algorithme, alors sa distance à la source est connue (et donc ne changera plus par la suite). On le montre par récurrence sur le nombre de sommets déjà traités. Soit u le prochain sommet traité par l'algorithme. Tout chemin depuis la source jusqu'à u, le cas échéant, se termine par un arc $v \to u$. Le tri topologique implique que v a déjà été traité par l'algorithme. Par HR la distance de la source à v était fixée quand v a été examiné et la distance à u a été mise à jour si emprunter cet arc constituait un chemin plus court. Donc tout chemin menant à u a été considéré, avec à chaque fois la bonne distance. La distance à u est donc fixée au moment où u est considéré.

On constate que cette preuve n'exige nulle part le caractère positif ou nul des distances. On a seulement utilisé l'addition des distances et leurs comparaisons.

Question 10 Déduire de cet algorithme un algorithme pour calculer les *plus longs* chemins à partir d'un sommet source donné.

Correction : Puisque les distances négatives sont autorisées, il suffit de remplacer la distance de chaque arc par son opposé puis de calculer les plus courts chemins.

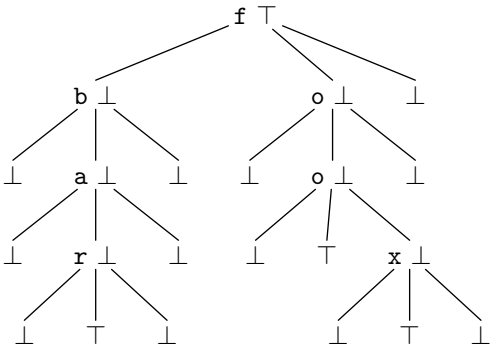
---

2 Arbres de préfixes ternaires

Dans ce problème, on considère une représentation alternative des arbres de préfixes vus en cours, appelés *arbres de préfixes ternaires*. L'objectif reste le même, à savoir représenter un ensemble de mots, en l'occurrence ici du type `String` de Java.

Un arbre de préfixes ternaire est, comme son nom l'indique, un arbre ternaire : tout nœud interne possède exactement trois sous-arbres, que nous appellerons par la suite sous-arbre *gauche*, sous-arbre *central* et sous-arbre *droit*. Chaque nœud, qu'il s'agisse d'une feuille ou d'un nœud interne, contient un booléen marquant la présence d'un mot dans l'ensemble s'il vaut `true`. Chaque nœud interne contient également un caractère c, utilisé comme ceci : les mots commençant par un caractère plus petit que c sont rangés dans le sous-arbre gauche ; les mots commençant par c sont rangés dans le sous-arbre central ; et les mots commençant par un caractère plus grand que c sont rangés dans le sous-arbre droit. Cette structure combine donc l'idée des arbres binaires de recherche avec celle des arbres de préfixes.

Voici un exemple d'arbre de préfixes ternaire contenant les quatre mots `""`, `"bar"`, `"foo"` et `"fox"`, où chaque booléen est représenté par $\top$ (pour `true`) ou $\bot$ (pour `false`) et où les caractères de branchements sont indiqués sur les nœuds internes.



On navigue dans un tel arbre de la façon suivante. Étant donné un mot, son premier caractère est comparé à la racine de l'arbre. En cas d'égalité, on descend dans le sous-arbre central et on passe au deuxième caractère du mot. Sinon, on descend soit dans le sous-arbre gauche, soit dans le sous-arbre droit, selon le résultat de la comparaison, en continuant de considérer le même caractère du mot. Lorsqu'on a fini d'examiner tous les caractères du mot, le booléen présent dans l'arbre à l'endroit où on est parvenu indique la présence du mot dans l'ensemble.

Si on recherche par exemple le mot `"fox"` dans l'arbre ci-dessus, on commence par comparer le premier caractère, `'f'`, avec celui de la racine. Comme il y a égalité, on descend dans le sous-arbre central. Puis on compare le deuxième caractère, `'o'`, avec celui du sous-arbre. Là encore, il y a égalité et on descend dans le sous-arbre central. On compare alors le troisième caractère, `'x'`, avec celui du sous-arbre, à savoir `'o'`. Cette fois, il n'y a pas égalité et on descend donc dans le sous-arbre droit, car `'x'` $>$ `'o'`. On compare de nouveau le troisième caractère, `'x'`, avec celui du sous-arbre, à savoir `'x'`. Comme il y a égalité, on descend dans le sous-arbre central. Comme on a fini d'examiner tous les caractères, on examine le booléen du sous-arbre. Il vaut ici `true`, ce qui indique que le mot `"fox"` est dans l'ensemble.

Si en revanche on avait recherché le mot `"for"`, la comparaison du troisième caractère, `'r'`, avec le caractère `'x'` nous aurait fait descendre dans le sous-arbre gauche du nœud contenant `'x'`, car `'r'` $<$ `'x'`. Comme on est parvenu à une feuille avant d'avoir examiné avec succès tous les caractères du mot, on en déduit que le mot `"for"` n'est pas dans l'ensemble.

Pour représenter les arbres préfixes ternaires en Java, on introduit une classe `TST` donnée figure 2. Dans le code qu'on écrira par la suite, on prendra soin de bien maintenir cet invariant :

soit `left`, `middle` et `right` sont tous `null`,
soit `left`, `middle` et `right` sont tous non `null`.

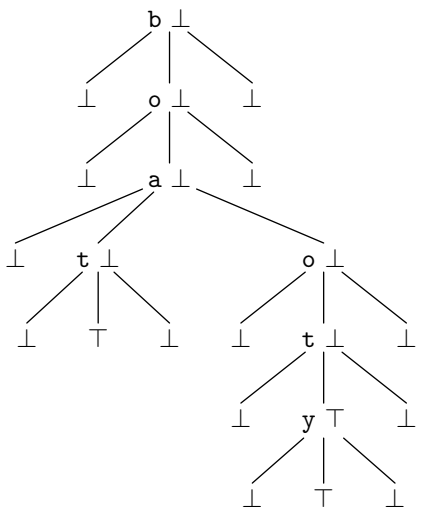
```
import java.util.*;
class TST {
  boolean word;                 // le mot est présent
  char    c;                    // caractère de branchement
  TST     left, middle, right;  // les trois branches

  TST() {
    this.word = false;
    this.c    = ' ';
    this.left = this.middle = this.right = null;
  }

  boolean contains(String w) { ... }
  void add(String w) { ... }

  LinkedList<String> toList() {
    LinkedList<String> res = new LinkedList<String>();
    toList(res, "");
    return res;
  }
  void toList(LinkedList<String> res, String prefix) { ... }
}
```

FIGURE 2 – La classe `TST`.

Dans le premier cas, il s'agit d'une feuille de l'arbre et le champ `c` est inutilisé ; dans le second, il s'agit d'un nœud interne et le champ `c` est le caractère de branchement.

Question 11 Donner un arbre de préfixes ternaire contenant les mots `"boot"`, `"boat"` et `"booty"`.

Correction :



Question 12 Deux arbres de préfixes ternaires contenant le même ensemble de mots sont-ils nécessairement égaux ? Justifier.

Correction : Non. Comme pour les arbres de préfixes usuels, on peut ajouter autant de sous-arbres ne contenant que des booléens `false` que l'on veut sans changer l'ensemble des mots.

Même si on se restreint à des arbres de taille minimale (où tout nœud de branchement contient au moins un booléen `true` dans l'un de ses sous-arbres), il reste possible de construire plusieurs arbres pour le même ensemble de mots. Ainsi, dans l'exemple donné au début du sujet, on pourrait brancher en premier lieu sur le caractère `'b'`, avec le mot `"ar"` dans le sous-arbre central et les mots `"foo"` et `"fox"` dans le sous-arbre droit.

Question 13 Ajouter à la classe `TST` une méthode `boolean contains(String w)` qui détermine si le mot `w` appartient à l'arbre `this`. (On rappelle que `char` est un type numérique dont les valeurs peuvent être comparées avec `<`, `==` et `>`.)

Correction : On procède comme pour les arbres de préfixes vus en cours, en avançant dans la chaîne `w` avec une boucle. Il faut cependant utiliser une boucle `while` plutôt qu'une boucle `for`, car certains déplacements dans l'arbre ne correspondent pas à un déplacement dans la chaîne.

```
boolean contains(String w) {
  TST t = this;
  int i = 0;
  while (i < w.length()) { // invariant t != null
    if (t.left == null) return false;
    char ci = w.charAt(i);
    if      (ci < t.c) t = t.left;
    else if (ci > t.c) t = t.right;
    else               { t = t.middle; i++; }
  }
  return t.word;
}
```

Question 14 Avec la méthode `contains` de la question précédente, on recherche un mot de longueur M dans un arbre contenant N mots au total. Si l'alphabet utilisé contient au plus A caractères différents, donner la complexité de cette recherche dans le pire des cas, en fonction du nombre de comparaisons de caractères effectuées. Justifier.

Correction : Dans le pire des cas, chaque caractère du mot amène à faire A comparaisons de caractères. La complexité dans le pire des cas est donc $O(A \times M)$. (Le nombre de mots dans l'arbre n'intervient pas.)

Question 15 Ajouter à la classe `TST` une méthode `void add(String w)` qui ajoute le mot `w` à l'arbre `this`.

Correction : On procède comme pour `contains`, avec une boucle `while`, mais il faut remplacer les feuilles par des nœuds au fur et à mesure de la descente.

```
void add(String w) {
  TST t = this;
  int i = 0;
  while (i < w.length()) { // invariant t != null
    char ci = w.charAt(i);
    if (t.left == null) {
      t.c = ci;
      t.left   = new TST();
      t.middle = new TST();
      t.right  = new TST();
    }
    if      (ci < t.c) t = t.left;
    else if (ci > t.c) t = t.right;
    else             { t = t.middle; i++; }
  }
  t.word = true;
}
```

Question 16 On souhaite maintenant ajouter à la classe `TST` une méthode `toList` qui renvoie la liste des mots contenus dans l'arbre, *triée par ordre alphabétique*. Le code de cette méthode est donné figure 2. Il utilise une seconde méthode `toList`, qui prend en arguments la liste résultat, à remplir, et une chaîne `prefix` représentant les caractères accumulés jusqu'à présent lors de descentes dans un sous-arbre central. Écrire le code de cette méthode.

Correction :

```
void toList(LinkedList<String> res, String prefix) {
  if (this.word) res.add(prefix);
  if (this.left == null) return;
  this.left.toList(res, prefix);
  this.middle.toList(res, prefix + this.c);
  this.right.toList(res, prefix);
}
```

Question 17 En déduire une méthode `LinkedList<String> sort(LinkedList<String> words)` qui trie une liste de mots par ordre alphabétique. La liste passée en argument ne doit pas être modifiée. La liste renvoyée doit contenir exactement les mots de la liste `words`, triés et sans répétition.

Correction :

```
 static LinkedList<String> sort(LinkedList<String> words) {
  TST t = new TST();
  for (String w: words)
    t.add(w);
  return t.toList();
}
```

Question 18 Montrer que la méthode `sort` a une complexité $O(N)$ en temps et en espace pour trier une liste de N mots, si on fait l'hypothèse que les mots ont une longueur bornée (par exemple au plus 10 caractères) et que l'alphabet est également borné (par exemple au plus 26 caractères différents). Cela est-il en contradiction avec le résultat vu en cours concernant la complexité optimale du tri ? Justifier.

Correction : Par le même raisonnement qu'à la question 14, l'ajout d'un mot dans le dictionnaire se fait en temps constant : chaque caractère du mot nous amène à au plus 26 comparaisons et il y a un nombre borné de caractères dans chaque mot. On en déduit un temps total de construction de l'arbre en $O(N)$. Il s'en suit un espace total occupé également en $O(N)$, car la complexité en espace ne peut pas dépasser la complexité en temps.

Note : un autre argument peut être utilisé pour justifier une complexité $O(N)$ en espace. En effet, chacun des nœuds contient un caractère qui provient des mots de la liste et chaque caractère d'un mot est utilisé au plus une fois comme nœud interne. Éventuellement il est partagé avec le même caractère apparaissant dans d'autres mots à la même position, ce qui peut amener à un arbre contenant strictement moins de nœuds internes que le nombre total de caractères.

Par ailleurs, le parcours de l'arbre se fait en temps linéaire en sa taille, donc en temps $O(N)$. Chaque insertion dans la liste résultat se fait en temps constant et le coût de chaque concaténation de chaînes (`prefix + this.c`) est borné car on a supposé les chaînes de longueur bornée. Le temps total de calcul est donc $O(N)$.

Cela n'est pas en contradiction avec le résultat vu en cours concernant la complexité optimale du tri, car nous sommes sortis de l'hypothèse qu'on ne fait que des comparaisons entre les valeurs à trier (ici les mots). En comparant les caractères individuellement, on exploite une information supplémentaire sur la structure des valeurs à trier, à savoir ici leur nature séquentielle. Un autre argument, tout aussi valable, consiste à dire que le nombre de mots à trier est ici borné, à savoir $N \leq A^M$, ce qui rend la notion de complexité en $O(N \log N)$ non significative.

Question 19 Expliquer pourquoi la structure d'arbres de préfixes vue en cours, où le branchement est réalisé par une table de hachage, ne permet pas d'en déduire aussi facilement une méthode `sort`.

Correction : Le branchement étant réalisé par une table de hachage, il n'y a pas de façon simple de parcourir les sous-arbres par ordre croissant des caractères (c'est-à-dire des clés dans cette table). On en est donc réduit à des choses comme

```
for (char c = 'a': c <= 'z'; c++)
  if (this.branches.containsKey(c))
    ...
```

qui est moins efficace. En théorie, cela reste de même complexité, car l'intervalle de caractères est borné, mais il peut être très grand (le type `char` contient 2^{16} valeurs).

Question 20 Le choix de représentation des arbres préfixes ternaires fait ici n'est pas optimal, en particulier parce que toutes les feuilles possèdent inutilement des champs `left`, `middle` et `right`. Proposer une autre représentation, plus économe en espace. (On ne demande pas de réécrire les opérations.)

Correction : On peut utiliser l'héritage, avec une classe abstraite `TST` et deux sous-classes pour les feuilles et les nœuds. Par exemple

```
abstract class TST     { protected boolean word; ... }
class Leaf extends TST { ... }
class Node extends TST { private char c;
                         private TST left, middle, right; ... }
```

Ce n'est pas encore parfait, car certains champs `word` restent inutilisés (quand on descend à gauche ou à droite).

A Bibliothèque standard Java

`class LinkedList<E>`	(implémente en particulier `Queue<E>`)
`boolean isEmpty()`	indique si la liste / file est vide
`void add(E e)`	ajoute l'élément `e` à la fin de la liste / file
`E poll()`	retire et renvoie l'élément situé au début de la liste / file
`class HashMap<K, V>`	
`void put(K k, V v)`	associe la valeur `v` à la clé `k` (en écrasant toute valeur précédemment associée à `k`, le cas échéant)
`boolean containsKey(K k)`	indique s'il existe une valeur associée à `k`
`V get(K k)`	renvoie la valeur associée à `k`, si elle existe, et `null` sinon
`class String`	
`int length()`	renvoie la longueur d'une chaîne
`char charAt(int i)`	renvoie le caractère à la position `i` (le premier caractère a la position 0)

* *
*