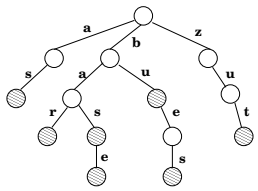
Projet 5 : Packages, `java.util`, polymorphisme. Réalisation d'une collection et de son itérateur.

1 Description du problème et première ébauche

On cherche à écrire une classe représentant un *ensemble* de mots de la langue française (un lexique). Un ensemble est défini de la façon suivante :

- c'est un regroupement d'objets (`Object`) en nombre quelconque, sans ordre particulier
- un objet donné ne peut figurer qu'une fois au plus dans un ensemble (au sens de l'égalité logique)
- on souhaite que cet ensemble soit manipulable comme une collection Java standard : la classe que vous écrirez devra donc implémenter l'interface `java.util.Set`, qui spécialise `java.util.Collection`.

Pour des raisons d'efficacité, on décide que cette classe sera réalisée au moyen d'un **arbre lexical**. Un arbre lexical sert à stocker des mots de façon compacte ; pour cela chaque nœud de l'arbre peut avoir jusqu'à 26 sous-arbres, correspondant chacun à une lettre de l'alphabet, et possède un attribut booléen qui indique si le nœud correspond à la fin d'un mot (cf. figure ci-dessous). Ce sont donc non pas des arbres binaires mais des arbres « 26-aires ».



Représentation d'un arbre lexical : chaque branche correspond à une lettre possible ; les nœuds contiennent un booléen qui indique si la succession des branches parcourues jusqu'à ce nœud correspond à un mot (ici : hachuré = vrai). L'arbre présenté ici contient donc les mots *as*, *bar*, *bas*, *base*, *bu*, *bues* et *zut*, mais pas le mot *bue* puisque le nœud auquel on arrive par les branches `b`–`u`–`e` est positionné à faux.

Q 1. Lisez attentivement le fichier `Squelette.java` fourni sur Moodle dans l'archive `pr5.tgz`, qui définit la structure de la classe `ArbreLexical` : attributs, constructeurs, et des « briques de base » à partir desquelles vous pourrez écrire les méthodes de haut niveau. **Cette classe devra faire partie d'un package** `mesCollections`.

Q 2. Écrivez les méthodes `isEmpty` et `size` (facile...).

Q 3. Écrivez les méthodes `containsAll`, `addAll`, `removeAll` et `retainAll`, en vous appuyant sur les méthodes `contains`, `add` et `remove` (non encore écrites).

Q 4. En vous servant des méthodes privées de bas niveau, écrivez ensuite (et de préférence dans cet ordre) les méthodes `contains`, `add` et `remove`, puis **testez-les** ainsi que celles de la question précédente.

Q 5. Déterminez à quelle condition un `ArbreLexical` peut être égal à un autre objet (cf. documentation de `Set`), puis écrivez en conséquence la méthode `equals` ; testez-la sur diverses collections.

Q 6. Écrivez la méthode `hashcode()` (après avoir lu la documentation de `Set`), testez-la.

⟶ *Reste à écrire la méthode* `iterator()`*...!*

2 Deuxième étape : un itérateur pour `ArbreLexical`

On cherche maintenant à écrire dans le package `mesCollections` une classe `ArbreIterator` qui implémente l'interface `java.util.Iterator` pour parcourir séquentiellement les mots stockés dans un `ArbreLexical`.

Q 7. Quels attributs faut-il donner à `ArbreIterator` pour résoudre le problème ? (l'itérateur doit « mémoriser » à la fois quel ensemble il doit parcourir et à quel stade du parcours il se trouve).

Q 8. Écrivez un constructeur pour cette classe, de façon à ce que la méthode `iterator` de `ArbreLexical` s'écrive ainsi : `public Iterator iterator() { return new ArbreIterator(this); }`

Q 9. Écrivez la méthode `hasNext`.

Q 10. Écrivez la méthode `next`. Il est peut-être nécessaire pour cela de modifier la signature de certaines méthodes de la classe `ArbreLexical`, voire d'en ajouter.

Q 11. Écrivez la méthode `remove`, qui enlève le dernier élément lu au moyen de `next` (même remarque).

3 Finalisation du package

Q 12. Générez la javadoc du package `mesCollections` : `javadoc -sourcepath sources -d doc mesCollections`
Vérifiez que *seule la classe* `ArbreLexical` apparaît **(l'itérateur doit être caché)**.

Q 13. Vérifiez ensuite que seuls les *constructeurs* de `ArbreLexical` ainsi que les méthodes présentes dans `Set` apparaissent dans la javadoc. En cas d'erreur modifiez les signatures des méthodes.

Q 14. Testez la classe en vérifiant en particulier (mais pas exclusivement) sa compatiblité avec d'autres collections (`ArrayList`, `HashSet`...). Les classes de test (extérieures au package `mesCollections`) devront être fournies et commentées.

4 Dernière étape : tests de performance

Q 15. Récupérez le fichier `dico.txt`. Écrivez un programme `LireDico` qui lit des mots (jusqu'à `STOP`) sur l'entrée standard (pour pouvoir utiliser la redirection : `java -classpath classes LireDico <dico.txt`) et effectue les opérations suivantes :

- placer les mots dans un arbre lexical et une autre collection (par exemple `ArrayList`) ;
- compter le nombre de mots dans chaque collection (il doit être le même évidemment !) ;
- afficher tous les mots en calculant le temps nécessaire (cf. classe `System`) ;
- rechercher 1000 fois un mot (par exemple « zygomatique ») en calculant le temps nécessaire pour chaque collection ;

Notez (dans les commentaires de `LireDico`) les résultats obtenus et expliquez-les.

Q 16. *QUESTION SUBSIDIAIRE :* Écrivez la méthode `motLePlusLong` qui retourne le mot le plus long du lexique.

Q 17. *QUESTION SUBSIDIAIRE :* Intégrez la méthode `motLePlusLong` dans les tests de `LireDico` et expliquez les mesures obtenues.