ÉCOLE POLYTECHNIQUE **FILIÈRE MP**
OPTION INFORMATIQUE

CONCOURS D'ADMISSION 2006

COMPOSITION D'INFORMATIQUE

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Dictionnaires

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction.

Dans tout ce problème, on s'intéressera à des structures de données pouvant représenter un ensemble de mots, dont l'archétype est un dictionnaire, tel qu'il est utilisé dans un correcteur orthographique.

La première partie de ce problème introduit la structure de données permettant de représenter des mots. Les trois parties suivantes étudient une structure de données permettant de représenter de façon efficace un dictionnaire par partage de préfixes. Cette structure de données s'appelle *trie* et deux façons de l'implanter sont proposées. La cinquième partie résout la recherche du mot le plus long. La sixième et dernière partie s'intéresse à la recherche d'anagrammes.

Partie I. Mots

Étant donné un alphabet $\mathcal{A}$, contenant un nombre fini de lettres (typiquement 26), on rappelle qu'un mot est une suite finie d'éléments de $\mathcal{A}$, et que l'ensemble des mots est noté $\mathcal{A}^*$.

Pour les réponses aux questions ci-dessous, afin qu'elles ne dépendent pas de l'alphabet choisi, on supposera définies une constante entière `N` qui est le cardinal de l'alphabet, et une fonction `lettre` qui prend en entrée un entier i compris entre 1 et `N` et qui écrit la $i^{ième}$ lettre de $\mathcal{A}$. De plus, `lettre(0)` écrit un espace, et `lettre(-1)` fait passer l'impression à la ligne suivante.

On définit un type `mot` permettant de représenter un mot sous la forme d'une liste d'entiers strictement positifs (les indices des lettres du mot). Selon le contexte, le parcours de la liste donnera les lettres du mot de gauche à droite (surnommé `motGD`) ou bien de droite à gauche (surnommé `motDG`). Cette seconde option sera choisie lorsque le rajout d'une lettre en fin de mot devra se faire en temps constant.

(Caml *)*

```
type mot == int list ;;
```

{ Pascal }

```
type mot = ^Cellule; Cellule = record
  lettre:integer; suite:mot; end;
```

En Pascal la liste vide est `nil` et l'on pourra utiliser la fonction suivante pour construire des mots :

```
function nouveauMot(c:integer; u:mot) : mot;
  var v:mot; begin new(v); v^.lettre := c; v^.suite := u; nouveauMot := v end;
```

Question 1 Définir une fonction `imprimer` qui imprime un mot de type `motDG` et passe à la ligne. Par exemple appeler cette fonction sur la liste $\langle 1, 2, 3\rangle$ avec l'alphabet usuel affiche `cba` suivi d'un passage à la ligne.

(* *Caml* *)	{ *Pascal* }
`imprimer : mot -> unit`	`procedure imprimer (`u`:mot);`

Question 2 Définir une fonction `inverseDe` qui transforme un mot de type `motDG` en un type `motGD`, et réciproquement. Par exemple $\langle 1, 2, 3\rangle$ devient $\langle 3, 2, 1\rangle$.

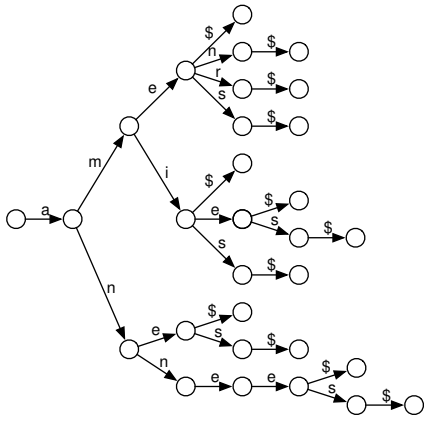
(* *Caml* *)	{ *Pascal* }
`inverseDe : mot -> mot`	`function inverseDe (`u`:mot) : mot;`

Partie II. Dictionnaires

Pour simplifier les structures de données, on ajoute à l'alphabet $\mathcal{A}$ une lettre de terminaison, notée \$. À tout mot dans $\mathcal{A}^*$ on associe le mot de $\mathcal{A}^*\{\$\}$ obtenu en rajoutant un \$ à la fin du mot ; ainsi, aucun mot du dictionnaire n'est préfixe d'un autre.

On représentera un dictionnaire au moyen de la structure de données appelée *trie*, qui est un arbre dont chaque nœud a un nombre arbitraire de fils, et dont les branches sont étiquetées.

Chaque branche de l'arbre est étiquetée par un élément de $\mathcal{A}$, sauf les branches qui mènent à une feuille et qui sont étiquetées par \$. De plus, les étiquettes des branches issues d'un même nœud sont toutes distinctes. L'ensemble des mots présents dans le dictionnaire est exactement l'ensemble des mots qu'on obtient en listant les étiquettes vues pendant un parcours de la racine jusqu'à une feuille. Voici un exemple de trie, pour le dictionnaire { ame, ames, amen, amer, ami, amis, amie, amies, ane, anes, annee, annees }. (Pour simplifier, les accents sont ignorés).



Plusieurs implantations des tries sont possibles.

Dans cette partie, on choisira d'utiliser une structure *tabulée* : Puisqu'on peut identifier par leurs étiquettes les branches issues d'un même nœud, chaque nœud contiendra un tableau indicé de 0 à `N`, dont la case d'indice i indique le sous-arbre issu de la branche d'étiquette i, ou bien l'arbre vide si cette branche n'est pas présente.

On définit le type `dictTab` qui permet de représenter un dictionnaire tabulé.

(* *Caml* *)	{ *Pascal* }

`type dictTab =`
`  VideT |`
`  NoeudT of dictTab vect ;;`
	`type tabN = array[0..N] of dictTab;`
	`  dictTab = ^NoeudT;`
	`  NoeudT = record fils:tabN; end;`

En Pascal l'arbre vide est représenté par `nil` et l'on pourra utiliser le constructeur suivant :

```
function nouveauDictTab(var a: tabN) : dictTab;
  var r:dictTab; begin new(r); r^.fils := a; nouveauDictTab := r end;
```

Question 3

a) Décrire la représentation tabulée du dictionnaire vide et celle du dictionnaire contenant comme unique élément le mot vide.

b) Prouver qu'une valeur de type `dictTab` représente un dictionnaire tabulé si, et seulement si, les feuilles sont exactement les nœuds rattachés à leur père par une branche d'étiquette \$. Cette propriété pourra être nommée *cohérence* d'une valeur de type `dictTab`.

Question 4 Écrire la fonction `imprimerDictTab` qui prend en argument un dictionnaire tabulé et écrit successivement tous les mots du dictionnaire (il s'agit bien d'afficher à l'écran les mots du dictionnaire, et non pas d'en retourner la liste).

(* *Caml* *)	{ *Pascal* }
`imprimerDictTab : dictTab -> unit`	`procedure imprimerDictTab (`d`:dictTab);`

La principale opération utile pour la correction orthographique est le test d'appartenance au dictionnaire.

Question 5 Écrire la fonction `estDansDictTab` qui prend en argument un mot de type `motGD` et un dictionnaire tabulé, et qui détermine si le mot est dans le dictionnaire. La réponse doit être donnée par la valeur de retour de la fonction.

(* *Caml* *)	{ *Pascal* }
`estDansDictTab :` `  mot -> dictTab -> bool`	`function estDansDictTab (`u`:mot, `d`:dictTab)` `  : boolean;`

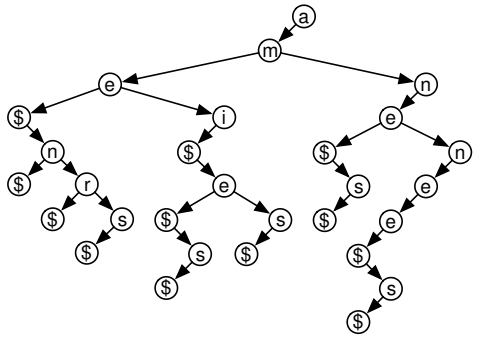
Une autre opération importante pour la correction orthographique est l'insertion d'un nouveau mot dans le dictionnaire.

Question 6 Écrire la fonction `ajoutADictTab` qui prend en argument un mot u de type `motGD` et un dictionnaire tabulé et qui retourne le dictionnaire modifié après insertion du mot u.

(* *Caml* *)	{ *Pascal* }
`ajoutADictTab :` `  mot -> dictTab -> dictTab`	`function ajoutADictTab (`u`:mot, `d`:dictTab)` `  : dictTab;`

Partie III. Dictionnaire binaire

Une autre implantation est la structure *binaire*. Il s'agit de faire descendre les étiquettes des branches vers les fils puis d'utiliser l'isomorphisme entre les arbres dont les nœuds ont un nombre arbitraire de fils (ordonnés), et les arbres binaires. L'arbre binaire se construit ainsi : dans la liste des fils d'un nœud, le premier fils devient le fils gauche de son père, et chacun des fils suivants devient le fils droit de son frère immédiatement précédent ; enfin, on fait disparaître la racine.



On définit le type `dictBin` qui permet de représenter un dictionnaire binaire.

(* *Caml* *)

```
type dictBin =
    VideB | NoeudB of
      dictBin * int * dictBin ;;
```

{ *Pascal* }

```
type dictBin = ^NoeudB; NoeudB = record
  etiq:integer; filsG:dictBin;
  filsD:dictBin; end;
```

En Pascal l'arbre vide est représenté par `nil` et l'on pourra utiliser le constructeur suivant :

```
function nouveauDictBin(c:integer; a,b:dictBin) : dictBin;
  var r:dictBin; begin new(r); r^.etiq = c;
  r^.filsG := a; r^.filsD := b; nouveauDictTab := r end;
```

Question 7 Expliquer pourquoi on fait disparaître la racine. Décrire la représentation binaire du dictionnaire vide et celle du dictionnaire contenant comme unique élément le mot vide.

Question 8 Écrire la fonction `imprimerDictBin` qui prend en argument un dictionnaire binaire et écrit successivement tous les mots du dictionnaire.

(* *Caml* *)

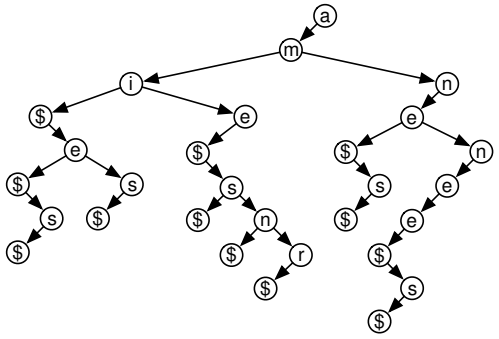
```
imprimerDictBin : dictBin -> unit
```

{ *Pascal* }

```
procedure imprimerDictBin (d:dictBin);
```

Pour les deux prochaines questions, deux réponses sont possibles, selon qu'on impose ou non que les étiquettes parcourues en descendant vers les fils droits soient par ordre croissant des indices des lettres dans l'alphabet. Il faudra choisir la solution supposant que les fils droits *ne* sont *pas* triés en ordre croissant.

Ainsi le dictionnaire binaire ci-contre peut aussi représenter le dictionnaire { ame, ames, amen, amer, ami, amis, amie, amies, ane, anes, annee, annees }.



Question 9 Écrire la fonction `estDansDictBin` qui prend en argument un mot de type `motGD` et un dictionnaire binaire et qui détermine si le mot est dans le dictionnaire.

(* *Caml* *)

```
estDansDictBin :
    mot -> dictBin -> bool
```

{ *Pascal* }

```
function estDansDictBin (u:mot, d:dictBin)
    : boolean;
```

Question 10 Écrire la fonction `ajoutADictBin` qui prend en argument un mot de type `motGD` et un dictionnaire binaire et qui modifie le dictionnaire pour y ajouter le mot.

(* *Caml* *)

```
ajoutADictBin :
    mot -> dictBin -> dictBin
```

{ *Pascal* }

```
function ajoutADictBin (u:mot; d:dictBin)
    : dictBin;
```

Partie IV. Comparaison des coûts ; conversion de représentations

On se place dans le cas où le dictionnaire manipulé contient n^5 mots de longueur moyenne n.

Question 11

a) Donner un encadrement du nombre de sommets S en fonction de n et `N`. Calculer en fonction de S le coût mémoire de chacune des deux représentations, et comparer.

b) Évaluer et comparer la complexité en temps du test d'appartenance et de l'ajout d'un mot de longueur ℓ, pour chacune des deux représentations.

Question 12 Écrire la fonction `tabVersBin` qui prend en argument un dictionnaire tabulé et retourne le dictionnaire binaire équivalent.

(* *Caml* *)	{ *Pascal* }
`tabVersBin : dictTab -> dictBin`	`function tabVersBin (`d`:dictTab) : dictBin;`

Question 13 Écrire la fonction `binVersTab` qui prend en argument un dictionnaire binaire et retourne le dictionnaire tabulé équivalent.

(* *Caml* *)	{ *Pascal* }
`binVersTab : dictBin -> dictTab`	`function binVersTab (`d`:dictBin) : dictTab;`

Partie V. Le mot le plus long

Il s'agit, étant donné un chevalet rempli de lettres, de trouver tous les mots du dictionnaire qu'on peut composer à l'aide de ces lettres, et en particulier le(s) plus long(s) d'entre eux. Le dictionnaire est représenté par un trie. Le choix de l'une des deux implantations ci-dessus est libre.

Question 14 Écrire la fonction `imprimerMotsDans` qui prend en argument un mot et un dictionnaire et qui retourne la liste des mots de ce dictionnaire composés de lettres du mot fourni en entrée. Une même lettre pourra être utilisée au plus le nombre de fois où elle apparaît dans le mot initial.

Estimer la complexité de cette fonction.

(* *Caml* *)	{ *Pascal* }
`imprimerMotsDans :` `  dictXXX -> mot -> unit`	`procedure imprimerMotsDans (`d`:dictXXX, `u`:mot);`

Partie VI. Anagrammes

Un mot est un *anagramme* d'un mot u donné s'il est composé de mots du dictionnaire et si chacune de ses lettres apparaît le même nombre de fois que dans u. Par exemple si $u =$ `cceeeehillnoopqtuy`, le mot u a pour anagrammes, entre autres, `ecole␣polytechnique`, `hellenique␣type␣coco` ou encore `pole␣cyclone␣ethique`. Les mots du dictionnaire sont séparés dans un même anagramme par le caractère ␣ imprimé en appelant la fonction `lettre` avec le paramètre 0.

Question 15 Écrire la fonction `imprimerAnagrammes` qui prend en argument un mot et un dictionnaire et qui imprime la liste des anagrammes de ce mot composés de mots du dictionnaire. Estimer la complexité de cette fonction.

(* *Caml* *)

```
imprimerAnagrammes :
  dictXXX -> mot -> unit
```

{ *Pascal* }

```
procedure imprimerAnagrammes (d:dictXXX;
   u:mot);
```

* *
*