

UE ALGO5 — TD2 — Séance 9 : Dictionnaire arborescent

Objectifs

À la fin de cette séance, vous devriez être capable de :

- utiliser des arbres pour résoudre des problèmes ;
- parcourir des arbres de manière récursive ou itérative ;
- spécifier et prouver des algorithmes de parcours d'arbre.

Exercice 1.

On considère un type `Mot` qui définit une séquence de caractères. On souhaite construire un type `Dico`, permettant de gérer un ensemble de `Mot` (dictionnaire).

Les primitives que devra fournir ce type `Dico` sont :

- la recherche d'un `Mot` dans le dictionnaire
- l'insertion d'un nouveau mot dans le dictionnaire
- la suppression d'un mot présent dans le dictionnaire
- l'impression des mots du dictionnaire selon l'ordre lexicographique

Pour minimiser la mémoire nécessaire, on souhaite que l'implémentation de ce type `Dico` repose sur une structure arborescente permettant de «partager» les *préfixes* communs à plusieurs `Mot`.

On suppose que tous les mots s'écrivent en ASCII, chaque lettre pouvant être stockée dans un octet. On ne cherchera pas à gérer les lettres accentués ou autres de l'UTF-8.

- Q1.* Dessinez un tel dictionnaire sous la forme d'un arbre (n-aire) contenant les mots suivants :
{arbitre, arbre, art, article, articuler, astre, bois, bon, botte, chat, zero}

Corrigé —

Le dessin de l'arbre n-aire fait apparaître rapidement la nécessité de représenter un «mot» ou «suffixe» vide.

Deux possibilités :

- on étend le type des arbre (ou des noeuds)
- ou, solution plus pratique, on utilise un caractère spécial pour représenter le suffixe vide (dans la suite, le caractère '\$').

On peut réfléchir aux propriétés que doit avoir ce caractère spécial : les caractères sont un type ordonné ; on va utiliser cet ordre pour l'implémentation du dictionnaire. Afin de respecter l'ordre lexicographique standard («art» < «article»), ce caractère doit être un minorant de l'ensemble des caractères utilisés (ou alors il faut modifier l'opération de comparaison).

-
- Q2.* Pour faciliter l'implémentation, une approche possible consiste à transformer cet arbre n-aire en un arbre binaire. Proposez une telle transformation, et dessinez le dictionnaire obtenu sur l'exemple précédent.

Corrigé —

Les noeuds de l'arbre binaire sont étiquetés par un caractère. Un des fils (appelé "successeur") représente la suite du mot, l'autre fils ("alternant") représente les alternatives au caractère courant.

L'exemple de la Q1 peut donc être représenté ainsi (le successeur est ici le fils droit/horizontal, l'alternant le fils gauche/vertical) :

```

a--r--b--i--t--r--e--$
|  |  |  |
|  |  |  r--e--$
|  |  |
|  |  t--$
|  |  |
|  |  i--c--l--e--$
|  |  |
|  |  u--l--e--r--$
|  |
|  s--t--r--e--$
|
b--o--i--s--$
|  |
|  n--$
|  |
|  t--t--e--$
|
c--h--a--t--$
|
z--e--r--o--$

```

Q3. Proposez une structure de données basée sur du chaînage par pointeurs pour implémenter ce dictionnaire (sous la forme d'un arbre binaire).

Corrigé —

```

1 typedef struct cellule {
    char l ; /* lettre ou "$" */
3     struct cellule *succ ; /* successeur */
    struct cellule *alt ; /* alternant */
5 } Cellule;

7 typedef Cellule *Dico ; /* un dico est un pointeur sur une cellule */

```

Q4. Spécifier et écrire une fonction **récursive** de recherche d'un mot dans le dictionnaire.

Corrigé —

On suppose que le mot à chercher est représenté par une liste chaînée de caractère, toujours terminée par '\$' (donc jamais vide).

```

1 typedef struct cellule_mot {
    char l ; /* lettre ou "$" */
3     struct cellule_mot *suiv ;
} Cellule_Mot;
5
typedef Cellule_Mot *Mot ;

```

Il suffit alors de parcourir simultanément le mot et le dico :

- on avance sur les successeurs si les caractères sont identiques ;
- on avance sur les alternants si le caractère du mot est inférieur à celui du dico.

Condition d'arrêt ?

- le dico est vide (plus d'alternant), il ne contient pas le mot
- les 2 caractères valent '\$' (le mot a été trouvé)

—le caractère courant du dico est supérieur au caractère courant du mot (le dico ne contient pas le mot).

Une solution récursive possible :

```
int Recherche (Mot m, Dico d) {
2 /* m pointe vers une séquence non vide terminée par "$" */
  /* vaut vrai si m appartient a D */
4
  if (d != NULL)
6     if (m->l == d->l)
            return (m->l == '$') || Recherche (m->suiv, d->succ) ;
8     else if (m->l > d->l)
            return Recherche (m, d->alt) ;
10    else
            return 0 ;
12    else
            return 0 ;
14 }
```

Q 5. Même question pour une fonction **itérative**

Corrigé —

```
int Recherche (Mot m, Dico d) {
2 /* m pointe vers une séquence non vide terminée par "$" */
  /* vaut vrai si m appartient a D */
4
  while ((d != NULL) && (m->l >= d->l) && (m->l != '$')) {
6     if (m->l == d->l) {
            m = m->suiv ; d = d->succ ;
8     } else
            d = d->alt ;
10  }
  if ((d != NULL) && (m->l == d->l) && (m->l == '$'))
12     return 1 ;
  else
14     return 0 ;
}
```

Q 6. Spécifier et écrire une fonction **récursive** d'insertion d'un mot dans le dictionnaire.

Corrigé —

C'est assez proche de la recherche :

—on parcourt simultanément m et d pour trouver leur plus grand préfixe commun

—si ce préfixe ne se termine pas par '\$', alors m n'appartient pas à d, et on insère donc le suffixe propre de m restant.

Deux situations possibles :

—on insère ce suffixe en fin d'une liste d'alternant ex : ajout de «avance» sur l'exemple

—on insère ce suffixe en tête ou milieu d'une liste d'alternant, il faut donc «raccrocher» la fin de cette liste en tant qu'alternant du nouveau suffixe ex : ajout de «bière» ou «bol» sur l'exemple.

```

void Insérer (Mot m, Dico *d) {
2  /* m pointe vers une séquence non vide terminée par "$" */
   /* m est inséré dans d, d désigne donc le nouveau dico
4   a la fin de l'exécution */
   /* si m appartient a d, alors d est inchangé */
6
   Dico tmp ;
8   if (*d != NULL)
       if (m->l == *d->l) {
10      if (m->l != '$')
           /* la fin de m n'est pas atteinte */
12      Insérer (m->suiv, &(*d->succ)) ;
       } else {
14      if (m->l > *d->l) {
           /* on continue a chercher un préfixe commun */
16      Insérer (m, &(*d->alt))
       } else {
18      /* m->l < *d->l */
           /* on a atteint le plus grand préfixe commun */
20      /* on crée et insère le suffixe propre */
           /* en tête ou milieu d'une liste d'alternants */
22      tmp = CreerSuffixe(m) ;
           tmp->alt = *d
24      *d = tmp ;
       }
26   }
   else
28      /* on a atteint le plus grand préfixe commun */
       /* on crée et insère le suffixe propre */
30      /* en fin d'une liste d'alternants */
       *d = CreerSuffixe(m)
32 }

34 Dico CreerSuffixe (Mot m) {
   /* m est terminée par '$', CreerSuffixe(m) renvoie un Dico
36   ne contenant que le mot m */

38   Dico d;
   d = (Dico) malloc (sizeof(Cellule)) ;
40   d->l = m->l ;
   d->alt = NULL ;
42   if (m->l != '$')
       d->succ = CreerSuffixe(m->succ) ;
44   else
       d->succ = NULL ;
46   return d;
}

```

Q 7. Spécifier et écrire une fonction **récurive** de suppression d'un mot dans le dictionnaire.

Corrigé —

Le problème est similaire, il faut trouver le plus grand préfixe commun à m et d et supprimer le suffixe propre à m restant (d devant être inchangé si ce suffixe propre est vide).

Attention, lorsque l'on supprime ce suffixe, il faut le remplacer par sa liste d'alternant. (ex : suppression de «arbitre», on remplace «itre\$» par «re\$»).

```

void Supprimer (Mot m, Dico *d) {
2  /* m pointe vers une séquence non vide terminée par "$" */

```

```

4  /* m est supprimé dans d, d désigne donc le nouveau dico
   a la fin de l'exécution */
6  /* si m n'appartient pas a d, alors d est inchangé */
6
   Dico tmp ;
8  if (*d != NULL) {
   if (m->l == *d->l) {
10     if (d->l == '$') {
       /* m est présent, on supprime '$' */
12     tmp = *d ;
       *d = *d->alt ;
14     free(tmp) ;
   } else {
16     Supprimer(m->succ, &(*d->succ)) ;
       if (*d->succ == NULL) {
18         /* m était présent dans D, et n'avait */
           /* pas d'alternants */
20         /* on peut supprimer la cellule d et la
           remplacer par son alternant */
22         tmp = *d ;
           *d = *d->alt ;
24         free(tmp) ;
       }
26     }
   } else {
28     if (m->l > *d->l) {
       /* on continue a chercher un préfixe commun */
30     Supprimer (m, &(*d->alt))
   } else {
32     /* m n'appartient pas à d, on ne fait rien ... */
   }
34 }
}
36 /* sinon, m n'appartient pas à d, on ne fait rien ... */
}

```

Dans cette solution les lettres du suffixe propre de m sont supprimées depuis la fin vers le début (on fait l'appel récursif sur m->succ avant de supprimer la cellule courante).

Q 8. Spécifier et écrire une fonction d'impression du dictionnaire dans l'ordre lexicographique (récursive ou itérative, au choix).

Corrigé —

On souhaite afficher l'ensemble des mots dans l'ordre lexicographique. Il suffit donc d'effectuer un parcours en profondeur d'abord du dico dans l'ordre «successeur», puis «alternant», en mémorisant la séquence courante depuis la racine (c'est à dire un Mot). Quand on atteint une feuille marquée '\$', on imprime cette séquence courante.

```

void Imprimer (Dico d, Mot m) {
2
    if (d!=NULL) {
4        if (d->l == '$')
            Afficher (m) ;
6        else
            Imprimer (d->succ, AjoutFin(m, d->l)) ;
8            Imprimer (d->alt, m) ;
    }
10 }

```

La fonction AjoutFin(m,c) construit un nouveau mot en ajoutant c en fin du mot m. Pour être plus efficace on peut aussi construire m «à l'envers» (par ajout en tête), et le «retourner» dans la fonction Imprimer(m).
