# TD/TME Semaine 8-11
# Jeux de mots

Version du 29 octobre 2017

## Objectif(s)

- ✯ Révision et mise en œuvre des listes chaînées
- ✯ Arbres binaires de recherche,
- ✯ Arbres lexicographiques,

## Exercice(s)

Nous allons définir des structures de données et écrire une bibliothèque de fonctions permettant de vérifier, rapidement, si un mot existe ou non dans un ensemble (potentiellement très grand) de mots. Plusieurs implantations possibles seront comparées et la bibliothèque la plus efficace sera utilisée pour programmer des jeux basés sur une recherche de mots.

# 1 Dictionnaire

### Exercice 1 *(obligatoire)* – Liste chaînée

Nous allons dans un premier temps utiliser une liste chaînée pour stocker et manipuler notre dictionnaire dont voici la définition :

```
typedef struct Lm_mot_ {
  char *mot;
  struct Lm_mot_ *suiv;
} Lm_mot;
```

Le fichier `french_za` vous est fourni. Il contient plus de 130.000 mots de la langue française. Les caractères spécifiques à la langue française (accents et 'ç') ont été enlevés pour simplifier le dictionnaire. Chaque ligne contient un seul mot. L'objectif étant d'accélérer au maximum la recherche d'un mot dans le dictionnaire, la liste de mots va être ordonnée (ordre alphabétique croissant).

Les fonctions suivantes vous sont fournies :

```
/* creation d'un element de liste */
Lm_mot *creer_Lm_mot(char *mot);

/* destruction de la liste donnee en argument */
void detruire_Lmot(Lm_mot *lm);
```

1. Écrivez la fonction d'ajout d'un mot dans la liste triée de mots :

   ```
   /* ajout en place d'un mot dans la liste donnee en premier argument */
   Lm_mot *ajouter_Lmot(Lm_mot *lm, char *mot);
   ```

2. Écrivez la fonction permettant de construire la liste à partir du fichier dont le nom est fournit en argument. Son prototype est le suivant :

```
Lm_mot *lire_dico_Lmot(const char *nom_fichier);
```

3. Écrivez une fonction permettant de rechercher un mot dans la liste. Son prototype est le suivant :

   ```
   Lm_mot *chercher_Lm_mot(Lm_mot *lm, const char *mot);
   ```

4. Quelle est, approximativement, la taille en mémoire de la liste chaînée contenant tous ces mots exprimées en fonction du nombre de mots et de leur longueur moyenne ? Combien de comparaisons faut il faire lors d'une recherche d'un mot ? Vous rappellerez comment deux mots sont comparés.
5. Écrivez dans un autre fichier `main_liste.c` un programme permettant de tester votre dictionnaire *liste*. Dans ce programme, vous testerez la validité de votre bibliothèque en recherchant un mot présent dans le dictionnaire et un mot qui n'en fait pas partie. Pour tester la performance (en terme de temps de recherche) de ce dictionnaire vous allez répéter ces opérations un nombre important de fois (plusieurs milliers ou millions de fois) et mesurer le temps à l'aide de la commande `time`. Cette fonction main recevra en argument (ligne de commande) le nombre répétitions de la recherche à effectuer. Son prototype est le suivant :

   ```
   int main(int argc, char **argv)
   ```

## Exercice 2 *(obligatoire)* – Arbre Binaire de Recherche

Une solution très commune pour représenter un dictionnaire consiste à utiliser un arbre binaire de recherche (ABR). À chaque noeud correspond un mot. L'arbre est structuré suivant l'ordre alphabétique des mots qu'il contient :

— Tous les mots qui précèdent le mot porté par un noeud sont dans le sous arbre gauche ;
— Tous les mots qui suivent le mot porté par un noeud sont dans le sous arbre droit ;
— Ces règles sont valables pour tous les noeuds de l'arbre.

Pour représenter les noeuds de cet ABR, nous avons défini la structure de données suivante :

```
typedef struct Nd_mot_ {
  char *mot;
  struct Nd_mot_ *g;
  struct Nd_mot_ *d;
} Nd_mot;
```

Comme vous le savez, dans le pire des cas la complexité d'un ABR peut équivaloir celle d'une liste. Pour éviter cet écueil nous allons nous assurer que l'ABR est équilibré par construction. Nous allons pour cela partir de la liste de mots ordonnée pour construire cet arbre.

L'algorithme est simple, nous allons choisir comme racine de l'ABR le noeud se trouvant au milieu de la liste, utiliser la première moitié de liste pour construire le sous arbre droit et la seconde pour le sous arbre gauche. Cet algorithme est naturellement récursif car il suffira de l'appliquer à nouveau pour construire chacun des sous arbres.

Les similitudes avec l'algorithme du tri rapide sont nombreuses, et comme pour celui-ci la fonction clef sera celle réalisant la partition de la liste en 2 listes. Dans ce cas la liste étant déjà ordonnée il donc aisé de la partitionner équitablement.

1. Écrivez une fonction permettant de couper une liste en 2, son prototype est le suivant :

   ```
   Lm_mot *part_Lmot(Lm_mot **pl)
   ```

   Cette fonction doit retourner un pointeur sur l'élément pivot qui servira de racine à l'ABR et modifiera la liste transmise en argument pour la scinder en son milieu. Il est nécessaire pour cette fonction d'utiliser un double pointeur pour supporter les cas ou la liste comporte moins de 3 éléments.

   Pour décomposer le travail à réaliser vous pourrez écrire une fonction retournant le nombre d'éléments contenus dans une liste de mots.

   Le pointeur retourné correspondra à l'élément pivot, à partir de son champ `suiv` vous pourrez accéder aux éléments de valeurs supérieures. La liste transmise en argument et modifiée par la fonction ne contiendra plus que les valeurs précédent le pivot.

2. Écrivez maintenant la fonction qui, en partant de la liste ordonnée de mots la transforme en ABR. Au fur et à mesure de son exécution les éléments de la liste sont détruits, les noeuds crées et les chaînes de caractères correspondant aux mots réutilisés. Cette fonction est intrinsèquement récursive, elle procède par dichotomie de la liste, créant à chaque exécution un noeud correspondant à l'élément pivot dans la liste qui lui est détruit. Lors du premier appel le noeud créé correspondra à la racine de l'ABR. Après exécution de cette fonction l'ABR sera construit et la liste détruite, il ne sera donc nécessaire de libérer la mémoire correspondant à cette liste ultérieurement. Son prototype est le suivant :

   ```
   Nd_mot *Lm2abr(Lm_mot *l);
   ```

3. Écrivez une fonction permettant de rechercher un mot dans l'ABR. Son prototype est le suivant :

   ```
   Nd_mot *chercher_Nd_mot(Nd_mot *abr, const char *mot);
   ```

4. Pensez à écrire une fonction qui libère la mémoire correspondant à l'arbre binaire de recherche. Son prototype est le suivant :

   ```
   void detruire_abr_mot(Nd_mot *abr);
   ```

5. Quelle est, approximativement, la taille en mémoire de l'ABR contenant tous ces mots exprimées en fonction du nombre de mots et de leur longueur moyenne ? Combien de comparaisons faut il faire lors d'une recherche d'un mot ?
6. Écrivez dans un autre fichier `main_abr.c` un programme permettant de tester votre dictionnaire *abr*. Dans ce programme, vous testerez la validité de votre bibliothèque en recherchant un mot présent dans le dictionnaire et un mot qui n'en fait pas partie. Pour tester la performance (en terme de temps de recherche) de ce dictionnaire vous allez répéter ces opérations un nombre important de fois (plusieurs milliers ou millions de fois) et mesurer le temps à l'aide de la commande `time`. Cette fonction main recevra en argument (ligne de commande) le nombre de répétitions de la recherche à effectuer. Son prototype est le suivant :

   ```
   int main(int argc, char **argv)
   ```

## Exercice 3 *(obligatoire)* – Arbre lexicographique

On peut utiliser une structure différente pour stocker un dictionnaire. Plutôt que de stocker chaque mot séparément, on peut construire un arbre dans lequel chaque noeud contient une lettre. Les noeuds fils contiennent la lettre suivante dans le mot et les noeuds frères les autres lettres possibles à cet emplacement. La figure 1 contient un exemple d'arbre lexicographique. Le '.' initial permet de relier toutes les initiales des mots dans un même arbre plutôt que de construire un arbre séparé pour chacune d'elles.

Chaque lettre a jusqu'à 26 successeurs. Nous pouvons le représenter sous la forme d'un arbre binaire. Dans ce cas, le fils gauche contient une des lettres suivantes et le fils droit contient le frère suivant (c'est la représentation d'un arbre général avec un arbre binaire). Avec cette représentation, le noeud '.' n'est plus nécessaire. La figure 2 montre une représentation schématique d'un tel arbre binaire pour stocker les mêmes mots que dans l'arbre de la figure 1.

 

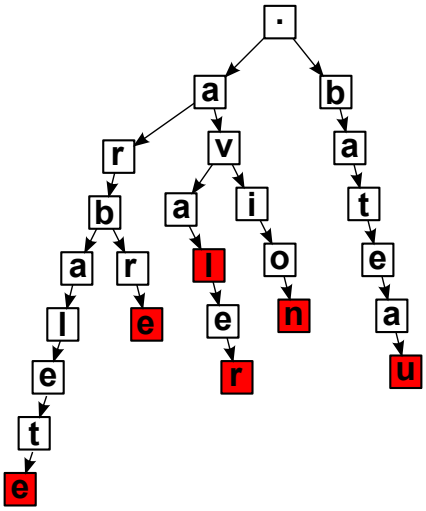


FIGURE 1 – Exemple d'arbre lexicographique contenant les mots "arbalete", "arbre", "aval", "avaler", "avion", "bateau". Les noeuds contenant la lettre finale d'un mot sont colorés.

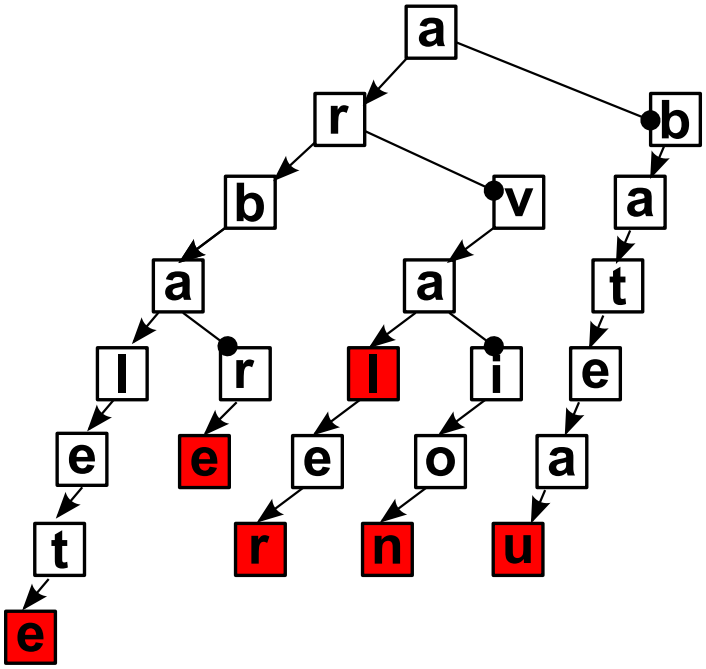


FIGURE 2 – Arbre lexicographique représenté avec un arbre binaire. Cet arbre contient les mêmes mots que sur la figure 1. Les arêtes se terminant par une flèche indiquent une lettre suivante (noeud fils) et les liens se terminant par un rond noir indiquent une lettre alternative (noeud frère). Les noeuds colorés indiquent que le noeud correspondant est la lettre finale d'un mot.

Pour stocker un tel arbre lexicographique, nous avons défini la structure de données suivante :

```
typedef struct noeud *PNoeud;
typedef struct noeud {
  char lettre;
  FDM fin_de_mot;
  PNoeud fils;
  PNoeud frere_suivant;
} Noeud;
```

Avec `FDM` une énumération définie ainsi :

```
// fin = il existe un mot finissant par cette lettre
// non_fin = il n'existe pas de mot finnissant par cette lettre
typedef enum _FDM {fin,non_fin} FDM;
```

1. Écrivez une fonction permettant d'allouer la mémoire associée à un noeud de l'arbre et d'initialiser ses différents champs. Le prototype est le suivant :

   ```
   PNoeud creer_noeud(char lettre);
   ```

2. Écrivez une fonction permettant d'afficher tous les mots stockés dans un arbre lexicographique. Le prototype est le suivant :

   ```
   void afficher_dico(PNoeud racine);
   ```

   L'affichage d'un mot se fera de la façon suivante : pendant le parcours de l'arbre, une chaîne de caractères est remplie pour garder en mémoire les lettres précédentes. Le mot est affiché lorsque le champ `fin_de_mot` est positionné sur `fin`. Vous pourrez donc utiliser une fonction intermédiaire de prototype :

   ```
   void afficher_mots(PNoeud n, char mot_en_cours[], int index);
   ```

   `mot_en_cours` est la chaîne dans laquelle les lettres sont stockées, `index` indique la position à laquelle écrire la prochaine lettre dans `mot_en_cours`.

3. Écrivez une fonction permettant de libérer la mémoire associée à un dictionnaire. Le prototype est le suivant :

   ```
   void detruire_dico(PNoeud dico);
   ```

4. Écrivez la fonction de recherche d'un mot dans le dictionnaire. Le prototype est le suivant :

   ```
   int rechercher_mot(PNoeud dico, char *mot);
   ```

   Vous pourrez écrire une fonction dédiée à la recherche d'une lettre dans la liste des frères. Cette fonction pourra renvoyer le pointeur vers le noeud trouvé (ou `NULL` sinon).

5. Nous allons maintenant écrire la fonction permettant d'ajouter un mot. Comme les fonctions précédentes, cet ajout se fera par récursion, lettre par lettre. Les lettres absentes de l'arbre seront ajoutées lorsque nécessaire (dans l'ordre alphabétique). Il ne faudra pas oublier de positionner le champ de fin de mot à la fin de l'ajout. Le prototype est le suivant :

   ```
   PNoeud ajouter_mot(PNoeud racine, char *mot);
   ```

   Plutôt que d'utiliser la fonction de recherche dans la liste des frères, vous pourrez écrire une autre fonction qui fera à la fois la recherche d'une lettre dans la liste des frères et l'ajout de la lettre en place si nécessaire.

6. Écrivez la fonction de lecture d'un dictionnaire depuis un fichier. Le prototype de la fonction est le suivant :

   ```
   PNoeud lire_dico(const char *nom_fichier);
   ```

7. Quelle est, très approximativement, la taille en mémoire de l'arbre contenant tous les mots d'un dictionnaire ? Combien de comparaisons faut il faire lors d'une recherche d'un mot ?

8. Écrivez dans un autre fichier `main_arbre.c` un programme permettant de tester votre dictionnaire utilisant des arbres lexicographiques. Dans ce programme, vous testerez la validité de votre bibliothèque en recherchant un mot présent dans le dictionnaire et un mot qui n'en fait pas partie. Pour tester la performance (en terme de temps de recherche) de ce dictionnaire vous allez répéter ces opérations un nombre important de fois (plusieurs milliers ou millions de fois) et mesurer le temps à l'aide de la commande `time`. Cette fonction main recevra en argument (ligne de commande) le nombre de répétitions de la recherche à effectuer. Son prototype est le suivant :

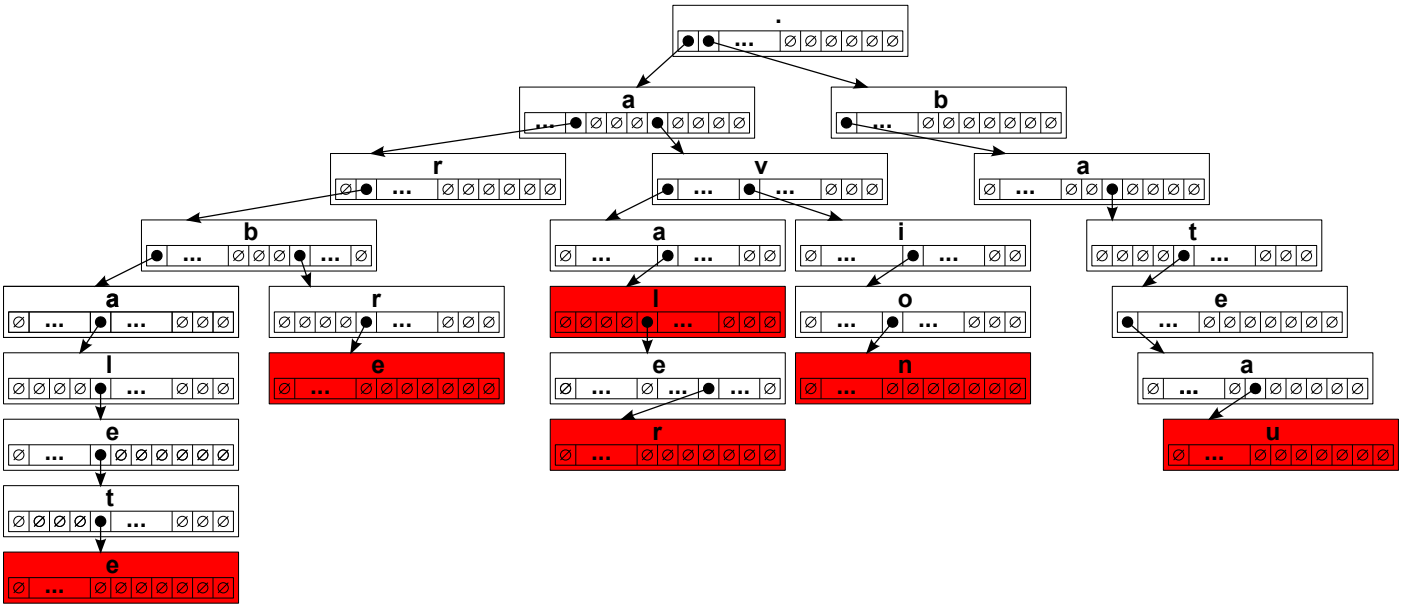

FIGURE 3 – Arbre lexicographique représenté avec des noeuds contenant un tableau de pointeurs sur les noeuds fils. Le symbole ∅ indique que la case correspondante du tableau contient la valeur `NULL`.

## Exercice 4 *(entraînement)* – Arbre lexicographique bis

Dans la représentation utilisée précédemment, la recherche peut être ralentie par le fait qu'il faut parcourir la liste des noeuds frères avant de trouver le bon. L'utilisation d'une liste chaînée pour stocker les frères a l'avantage de réduire la taille de la structure (seuls les noeuds utiles sont créés), mais cela a un coût en terme d'efficacité : il faut allouer les noeuds un par un et parcourir la liste pour trouver le bon. Nous allons définir une autre structure d'arbre lexicographique. Cette fois, tous les noeuds auront un tableau de 26 fils, un par lettre de l'alphabet (voir figure 3).

La structure de données sera la suivante :

```
typedef struct noeudTab *PNoeudTab;
typedef struct noeudTab {
  char lettre;
  FDM fin_de_mot;
  PNoeudTab fils[26];
} NoeudTab;
```

1. Écrivez les fonctions adaptées à cette nouvelle structure de données. Les prototypes sont les suivants :

   ```
   PNoeudTab creer_noeud(char lettre);
   PNoeudTab ajouter_mot(PNoeudTab racine, char *mot);
   void afficher_mots(PNoeudTab n, char mot_en_cours[LONGUEUR_MAX_MOT], int index);
   void afficher_dico(PNoeudTab racine);
   void detruire_dico(PNoeudTab dico);
   int rechercher_mot(PNoeudTab dico, char *mot);
   ```

   Ces fonctions doivent se comporter de façon similaire aux fonctions que vous avez écrites avec l'autre implantation de l'arbre lexicographique.
2. Écrivez la fonction de lecture d'un dictionnaire depuis un fichier. Le prototype de la fonction est le suivant :

   ```
   PNoeudTab lire_dico(const char *nom_fichier);
   ```

3. Quelle est, toujours approximativement, la taille mémoire occupée par l'arbre contenant tous les mots d'un dictionnaire ? Combien de comparaisons faut il faire lors d'une recherche d'un mot ?

## Exercice 5 *(obligatoire)* – Comparaison

1. Les différents programmes de test que vous réalisés vous permettent de comparer en pratique l'efficacité des différents dictionnaires que vous avez implémentés. Quelles sont vos conclusions ?

À partir de maintenant, vous utiliserez la structure de données (et les fonctions associées) qui permettent la recherche la plus rapide dans le dictionnaire. À titre d'exercice, vous pourrez essayer d'utiliser d'autres implantations d'arbres pour comparer les vitesses d'exécution en pratique.

## Exercice 6 *(entraînement)* – Le mot le plus long

Vous allez maintenant écrire un programme permettant de rechercher le mot le plus long du dictionnaire à partir d'un ensemble de lettres données.

1. Pour commencer, écrivez une fonction permettant de trouver tous les mots du dictionnaire contenant une ou plusieurs des lettres indiquées, et seulement celles-là. Il faudra considérer toutes les séquences de lettres possibles et interroger à chaque fois le dictionnaire pour savoir si la séquence est un mot valide. Vous écrirez cette fonction dans le fichier `mot_le_plus_long.c`

   Le prototype de la fonction sera le suivant :

   ```
   Lm_mot *trouver_les_mots(char *lettres, ??? dico);
   ```

   L'argument `lettres` contient les lettres à considérer (attention à toujours bien finir par un '\0'). Chaque lettre ne pourra être utilisée au maximum qu'une seule fois. Vous utiliserez le type approprié pour le dictionnaire.

   Vous pourrez utiliser une fonction intermédiaire récursive.

2. Écrivez maintenant la fonction permettant de trouver le mot le plus long. Vous prendrez soin de libérer la mémoire qui aura été allouée. Prototype de la fonction :

   ```
   void le_mot_le_plus_long(char *lettres, ??? dico, char *res);
   ```

   Le mot le plus long sera renvoyé dans l'argument `res` auquel l'appelant prendra soin de donner une taille suffisante. La chaîne de caractères `lettres` correspond aux lettres à considérer.

3. Par extension, écrivez une fonction permettant de renvoyer tous les mots ayant une longueur donnée et contenant uniquement les lettres indiquées. Prototype de la fonction :

   ```
   Lm_mot *mots_de_longueur_donnee(char *lettres, ??? dico, int l);
   ```

4. Écrivez une fonction main pour tester votre programme à partir de caractères tapés au clavier. Le programme, écrit dans le fichier `main_mot_le_plus_long.c`, demandera à l'utilisateur de saisir des caractères et de préciser la longueur souhaitée (-1 pour le mot le plus long).

## Exercice 7 *(entraînement)* – Carrés magiques

Vous allez maintenant écrire des fonctions permettant de créer des carrés magiques. Un carré magique est composé de mots pouvant se lire soit horizontalement ou verticalement avec la particularité que ce sont les mêmes mots qui sont lus dans les deux sens. Exemple de carré magique :

C U L T E
U N I E S
L I O N S
T E N T A
E S S A I

 

1. Écrivez une fonction permettant de renvoyer tous les mots d'une longueur donnée dans le dictionnaire, nous nous en servirons pour limiter les recherches de mots à ceux qui ont la bonne taille. Vous prendrez soin de choisir la structure la plus adaptée pour le dictionnaire. Vous écrirez votre fonction dans un fichier `carre_magique.c`.
2. La recherche d'un carré magique consiste à créer un carré de la taille donnée, puis à tester tous les mots possibles sur la première ligne/première colonne. Lorsque ce premier mot est positionné, le carré incomplet résultant est transmis à une fonction chargée de le compléter par récurrence. La récursion s'arrête lorsque le carré est complet. Une étape de la récursion consiste à extraire la ligne/colonne à compléter (la première qui est incomplète), à rechercher la liste des mots permettant de compléter cette ligne/colonne et à les ajouter les uns après les autres avant de refaire appel à la même fonction.

   Écrivez la fonction permettant de déterminer la liste des mots commençant par un préfixe donné. Quel type de dictionnaire sera le plus approprié pour cela ?
3. Écrivez à présent la fonction de recherche d'un carré magique avec la fonction récurrente de remplissage du carré.
4. Écrivez une fonction main dans un fichier `main_carre_magique.c` pour tester vos fonctions. Le programme demandera à l'utilisateur de saisir la taille souhaitée pour les carrés magique et il affichera tous les carrés trouvés (il peut y en avoir beaucoup...).

 