DM - Filtrage d'un texte

Dans ce DM, on se propose d'implémenter des arbres binaires de recherche de chaines de caractères, dans le but de trouver tous les mots d'un texte qui ne sont pas déjà présent dans un autre texte.

1 Conditions de rendus

Ce devoir maison est à réaliser en binôme au sein d'un même groupe de TP.

Le devoir est à rendre sur le e-learning pour le dimanche 12 mars 2023, 23h59. Un fichier `.zip` y sera déposé, contenant a minima les fichiers `ABR.c`, `ABR.h`, `filtrage.c` ainsi qu'un `makefile` et organisé suivant les préceptes du cours de `Perfectionnement C`. Le nom de ce fichier `.zip` sera formaté par `nom1_nom2.zip` où `nom1` et `nom2` désigne les noms de famille des membres du binôme.

La correction s'effectuera en partie de manière automatique. Il est donc particulièrement important que vous respectiez scrupuleusement les noms de fonctions demandées.

Une grande attention devra être portée à éviter toute fuite mémoire. Pour vérifier cela, vous pourrez utiliser l'outil `valgrind` sous `Linux`. L'absence de fuite de mémoire sera un élément important de la notation.

2 Partie 1 : Manipulation d'un ABR de chaînes de caractères

On rappelle que l'ordre alphabétique $<$ est l'ordre des lettres de l'alphabet. Celui-ci s'étend en un ordre $<$ sur l'ensemble des `char` en `C` en utilisant la coercition implicite des `char` en `int`.

Un mot (qu'il soit prononçable ou non) est une suite de caractères. En partant de l'ordre alphabétique, on construit alors l'ordre lexicographique sur l'ensemble des mots comme suit :

- le mot vide, *i.e.* le mot ne comportant aucune lettre, est plus petit que n'importe quel mot comportant au moins une lettre.
- un mot `A = a A'` est plus petit qu'un mot `B = b B'`, `A'` ou `B'` pouvant ne comporter aucune lettre, si l'on est dans l'un des cas suivant :
  - ⇝ `a < b` pour l'ordre alphabétique ;
  - ⇝ `a == b` et `A' < B'` pour l'ordre lexicographique.

Écrire un module `ABR.c`, accompagné de son `.h` correspondant, permettant de créer et manipuler des arbres binaires de recherche dont les clés sont des chaines de caractères (l'ordre sur celles-ci étant naturellement l'ordre lexicographique).

Pour cela, on s'appuyera sur la structure suivante :

```
typedef struct noeud {
    char * mot;
    struct noeud *fg, *fd;
} Noeud, *Arbre;
```

Les fonctions suivantes devront être disponibles :

— `Noeud * alloue_noeud(char * mot);` :
allocation d'un nœud en recopiant strictement la chaîne de caractères passée en argument.

— `void parcours_infixe(Arbre A)` ; :
affichage des mots contenus dans l'arbre `A`, un par ligne, sur la sortie standard dans l'ordre d'un parcours en profondeur infixe de `A`

— `Noeud * ajout(Arbre *A, char * mot);` :
ajout dans l'arbre `*A` du `mot` passé en paramètre en maintenant la structure d'arbre binaire de recherche ; la fonction renverra l'adresse du nœud créé si cela a été possible ou `NULL` sinon.

— `Noeud * extrait_max(Arbre *A);` :
suppression de l'arbre `*A` du plus grand mot qu'il contient ; la fonction renverra l'adresse du nœud supprimé.

— `Noeud * suppression(Arbre * A, char * mot);` : suppression de l'arbre `*A` du `mot` passé en paramètre s'il est présent ; la fonction renverra l'adresse du nœud supprimé. On privilégiera la remontée du plus grand mot présent dans le sous arbre de gauche lorsqu'on aura le choix.

— `void libere(Arbre * A);` :
libération de tous les nœuds contenu dans l'arbre `*A`.

— `void dessine(char * nom, Arbre A);` :
crée un fichier `.pdf` dont le `nom` est passé en paramètre qui contiendra une représentation graphique de l'arbre `A` réalisée à l'aide de l'utilitaire `dot`.

Le module `ABR.c` ne contiendra pas de fonction `main` ; vos tests seront effectuées dans un autre fichier nommé `test_ABR.c`.

3 Partie 2 : Création d'un ABR à partir d'un fichier texte

Ajouter au module précédent la fonction `int cree_arbre(char * nom, Arbre * A);` qui lit le contenu du fichier `nom` mot à mot et les insère successivement dans l'arbre `*A`. Initialement, l'arbre `*A` pourra être considéré comme étant vide.

Pour lire le fichier mot à mot, on pourra :

1. lire le contenu du fichier par bloc de `512` caractères en utilisant `fgets`
2. découper chaque bloc en "tokens" grâce à `strtok` en utilisant comme séparateurs la chaîne

`"␣\n,;:.?!\"()-'"`

Attention : Aucune modification des chaînes de caractères ne sera réalisée : une majuscule restera une majuscule de sorte que les mots `"En"` et `"en"` seront considérés comme étant différents.

4 Partie 3 : Filtrage des mots d'un texte

1. Dans un fichier `filtrage.c` incluant `ABR.h`, écrire une fonction

   ```
   int filtre(Arbre * A, Arbre filtre, Arbre * utilises);
   ```

   qui supprime de l'arbre `*A` tous les nœuds contenant un mot présent dans l'arbre `*filtre` et insère ce mot dans l'ABR `*utilises`.

2. Dans ce même fichier, écrire un `main` qui :

   - reçoit en paramètre les chemins d'accès vers deux fichiers textes `texte` et `filtre` ;
   - construit les deux ABR contenant respectivement les mots présents dans les fichiers textes précédents ;
   - affiche sur le terminal les mots présents dans le fichier `texte` mais qui ne sont pas dans `filtre`, puis les mots présents simultanément dans les deux fichiers.

     Ces deux listes de mots seront triées par ordre lexicographique.

   L'option `-v` de ce script permettra en plus de créer quatre fichiers `.pdf` contenant une représentation visuelle des ABR suivant :

   — l'arbre contenant les mots présents dans le fichier `texte` ;
   — l'arbre contenant les mots présents dans le fichier `filtre` ;
   — l'arbre contenant les mots présents dans `texte`, mais pas dans `filtre` ;
   — l'arbre contenant les mots présents simultanément dans `texte` et `filtre`.

   Les noms des fichiers créés devront être respectivement le nom du fichier `texte` suivi de `.pdf`, le nom du fichier `filtre` suivi de `.pdf`, `filtrage.pdf` et `en_commun.pdf`.

5 Annexe : Un exemple de sortie du script

Considérons les fichiers `texte` et `filtre` donnés par la Figure 1.

Un arbre binaire est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé noeud.	Auprès de mon arbre, Je vivais heureux, J'aurais jamais dû m'éloigner de mon arbre...

FIGURE 1 – Contenu des fichiers `texte.txt` et `filtre.txt` respectivement.

La sortie du script sera alors donnée par la Figure 2.

Les arbres crées avec l'option `verbose` seront alors données par les Figures 3 à 6.

```
Mots présents uniquement dans le texte de référence :
-----------------------------------------------------

En
appelé
binaire
chaque
d
données
dont
est
forme
hiérarchie
informatique
la
noeud
peut
qui
représenter
se
sous
structure
un
une
élément

Mots présents dans les deux textes :
------------------------------------

arbre
de
```

FIGURE 2 – Résultat du script suite à la commande `./filtre -v texte.txt filtre.txt`

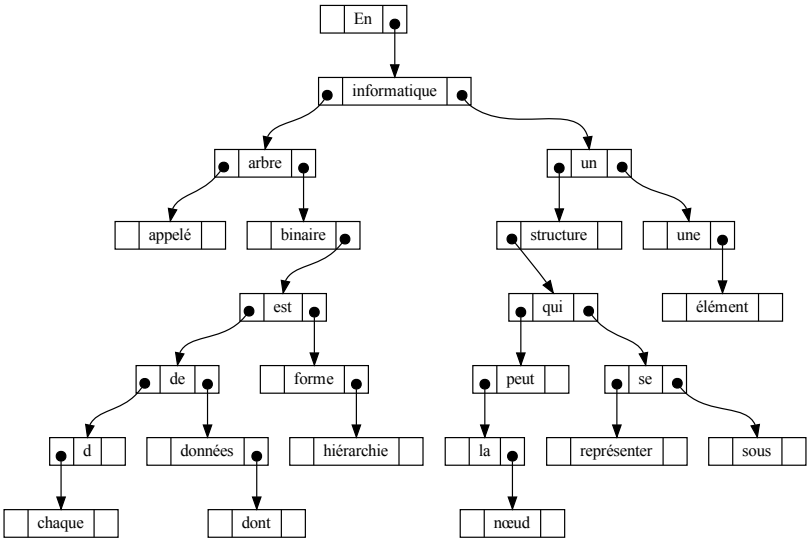

FIGURE 3 – Représentation de l'arbre contenant les mots du fichier `texte`

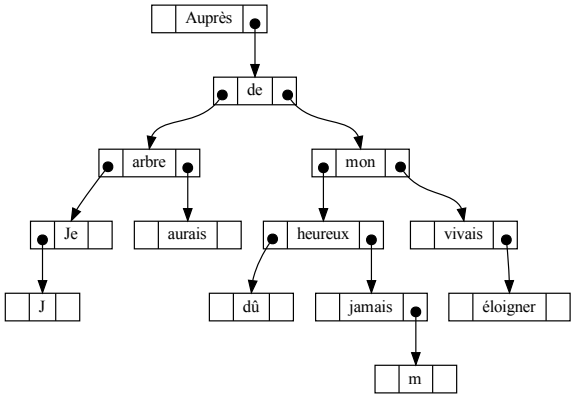

FIGURE 4 – Représentation de l'arbre contenant les mots du fichier `filtre`

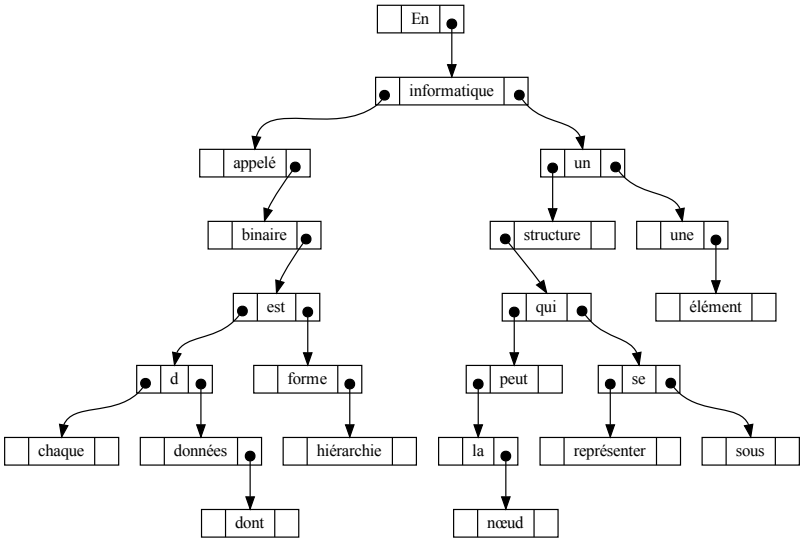

FIGURE 5 – Représentation de l'arbre contenant les mots présent dans le fichier `texte` et absent de `filtre`

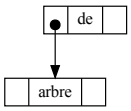

FIGURE 6 – Représentation de l'arbre contenant les mots communs aux fichiers `texte` et `filtre`