Université Joseph Fourier Année 2014-2015
RICM3

C pour la programmation système
Projet

Ce projet consiste à analyser le contenu d'un fichier de texte, afin de produire la liste, par ordre alphabétique, des **mots composant ce texte**, avec pour chacun la liste des **numéros de ligne + caractère dans la ligne** où le mot apparaît dans le texte.
Vous récupérerez le fichier Projet_Dict.tar.gz ici :
http://www-ufrima.imag.fr/INTRANET/placard/INFO/ricm3/LANGAGE_C/
et vous devrez rendre une archive qui respecte son organisation.

Nous considérerons que les mots sont en *lettres minuscules* non accentuées exclusivement, et que les séparateurs de mots peuvent être divers (point, virgule, point d'interrogation, etc…).
Par exemple, sur le texte suivant :

```
je ne laisserai pas se faner les pervenches,
sans aller ecouter ce qu'on dit sous les branches
```

Nous devrons obtenir :

```
aller (2,6)
branches (2,42)
ce (2,20)
dit (2,29)
ecouter (2,12)
faner (1,24)
je (1,1)
laisserai (1,7)
les (1,30) (2,38)
ne (1,4)
on (2,26)
pas (1,17)
pervenches (1,34)
qu (2,23)
sans (2,1)
se (1,21)
sous (2,33)
```

Nous vous fournissons le fichier header `read_word.h` et la bibliothèque `libtokenize.so` (versions Linux 32 bits et 64 bits) qui contient la fonction

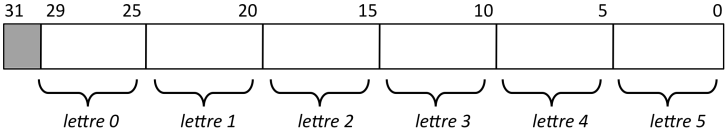
```
char *next_word(FILE *f, unsigned int *nblin, unsigned int *nbcol);
```

Vous devrez utiliser cette fonction pour l'analyse lexicale du fichier : chaque appel à cette fonction renvoie le prochain mot du texte contenu dans le fichier répéré par le descripteur `f` (on laissera bien sûr le fichier ouvert pendant tout le traitement). Les paramètres `nblin` et `nbcol` permettent de récupérer en outre le numéro de ligne et de colonne (i.e. de caractère dans la ligne) où se situe le mot.
Cette fonction `next_word` renvoie tout mot entre séparateurs de mots. L'ensemble des *séparateurs considérés* sera défini grâce à la macro `SEP` dans le fichier header, qui est une chaîne de caractères ayant une valeur par défaut, modifiable selon vos souhaits (noter que, bien que n'apparaissant pas dans cette macro, le caractère de fin de ligne est d'ores et déjà considéré comme un séparateur par la fonction `next_word`). Rien d'autre ne devra être modifié dans ce fichier header.

L'objectif du projet *est de construire une structure chaînée de tous les mots trouvés dans le texte, c'est à dire un dictionnaire (on fera en sorte d'insérer tout nouveau mot à la bonne place dans cette structure pour qu'elle soit en permanence triée alphabétiquement), avec pour chaque mot la liste des emplacements où il se trouve, puis de faire afficher cette liste à l'utilisateur.*
Votre programme devra recevoir en paramètre *le nom du fichier à traiter*, et on pourra prévoir qu'il travaille sur *l'entrée standard* si aucun nom de fichier ne lui est transmis.

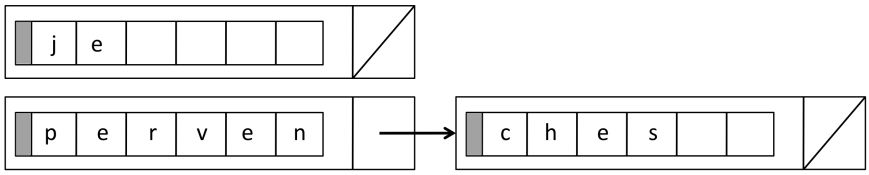
(1) Avant de commencer la réflexion sur les structures de données et fonctions associées, nous vous suggérons de préparer le corps de votre programme principal, avec la gestion du paramètre, l'ouverture du fichier, et l'itération sur la fonction `next_word` pour récupérer les mots du texte, afin de vous assurer que l'utilisation de cette fonction est bien comprise.

(2) Une fois que ceci est opérationnel, il va être possible de réaliser le codage des mots et du dictionnaire.
Afin d'optimiser le stockage des mots, nous associerons un code à chaque lettre minuscule, de la façon suivante : 'a' → 1, 'b' → 2, 'c' → 3, etc…, 'y' → 25, 'z' → 26. On remarque donc qu'il suffit de 5 bits pour coder une lettre. Nous allons utiliser des entiers sur 32 bits (uint32_t) pour coder 6 lettres successives, comme suit :

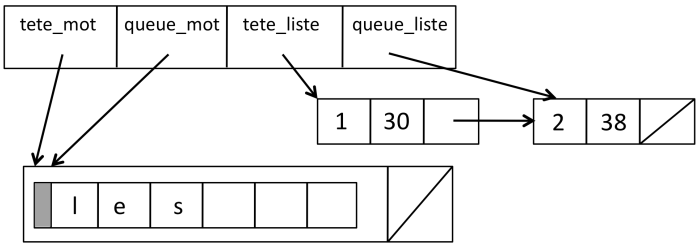


Note. Lorsque tout le projet sera réalisé, une amélioration possible sera de rendre cette structure de données paramétrable : l'utilisation de l'option -D à la compilation permettra de choisir une taille d'entiers (uint8_t, uint16_t, uint32_t ou uint64_t).

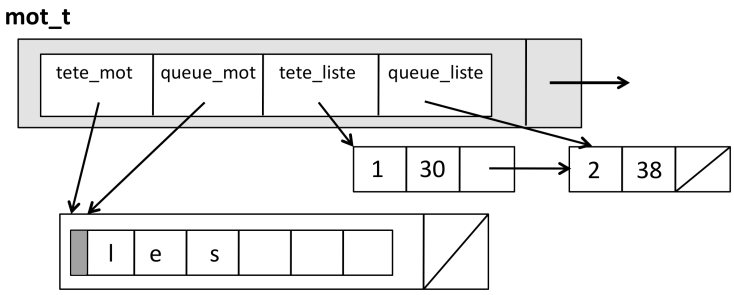
Définir un **type maillon_t** : pour tout mot de plus de 6 lettres, un premier maillon contiendra les 6 premières lettres et pointera sur le maillon contenant les 6 lettres suivantes, et ainsi de suite. Voici des exemples :



Définir un **type emplacement_t** et un **type mot_t** : chaque mot est bien sûr associé à une telle liste chaînée de maillons (on repérera le maillon de tête et le maillon de fin), mais aussi à une liste chaînée des emplacements (ligne, colonne) où ce mot apparaît, par exemple :



Les champs tete_mot et queue_mot indiquent le premier et le dernier maillon du mot, et les champs tete_liste et queue_liste indiquent le premier et le dernier maillon de la liste des emplacements de ce mot.
Le dictionnaire sera lui-même une liste chaînée de tels mots, il nous faudra donc un type mot_t comme suit :



Le programme préliminaire de la question (1) devra être repris pour construire un tel dictionnaire ordonné des mots du texte, puis le relire pour afficher les mots et leurs

emplacements. Vous testerez notamment sur les fichiers fournis dans le répertoire Examples (attention, nous testerons évidemment sur d'autres exemples !).
Nous vous suggérons de commencer par une version dans laquelle les mots sont insérés dans le dictionnaire dans leur ordre d'apparition, puis d'améliorer la fonction d'insertion de telle façon que tout nouveau mot soit inséré à la place adéquate (en utilisant une fonction compare_mots, voir ci-dessous).

Définir les fonctions nécessaires pour le traitement demandé, en particulier :

- les fonctions **char_to_num** et **num_to_char** pour la correspondance entre un caractère et son code ('a' <-> 1, 'b' <-> 2, … , 'z' <-> 26)
- les fonctions **set_charnum** et **get_charnum** pour modifier et consulter la $k^{ième}$ lettre ($0 \leq k \leq 5$) dans un maillon
- les fonctions nécessaires pour *convertir* une chaîne de caractères en liste de maillons, et inversement
- les fonctions de *création* d'un mot_t (avec un premier emplacement, pour la première rencontre de ce mot) et *d'affichage* d'un mot_t
- une fonction **compare_mots** qui compare les mots représentés par deux mot_t et renvoie : 0 s'ils sont identiques, un entier négatif si le premier mot est alphabétiquement plus petit que le deuxième, et un entier positif si le premier mot est alphabétiquement plus grand que le deuxième (aide : penser à utiliser la fonction get_charnum pour connaître chaque lettre)
- une fonction **insertion_dictionnaire** qui insère un mot à la bonne place dans le dictionnaire. Attention, si le mot est déjà présent, il faudra juste lui ajouter le nouvel emplacement où il a été trouvé
- une fonction d'affichage du dictionnaire.

Organisation logicielle : on attend une organisation logicielle avec des *fichiers headers et sources C* correctement conçus (séparer les différentes fonctionalités), placés dans les répertoires prévus à cet effet dans l'archive fournie, et un *Makefile* associé (aussi riche que vous pouvez le faire). Vous placerez, dans le répertoire doc, le rapport en pdf (voir ci-dessous) et toute autre documentation que vous jugerez utile (doc Doxygen, mans,…)

Remise : le projet est à faire en binôme (ou seul). Nous devrons voir votre état d'avancement (autant que possible il sera quasiment fini) à la séance du 7/4.
Vous enverrez (par mail à votre enseignant) avant le 12 avril au soir, un **fichier tar** ou **tar.gz** (tout autre format sera rejeté) contenant :

- vos **sources** *correctement commentés*, et Makefile. Si vous avez utilisé des fichiers de test, vous les joindrez également. Nettoyez vos répertoires : ne pas joindre les fichiers objets ou exécutables, des vieux fichiers d'auto-sauvegarde de l'éditeur,…
- un **rapport en pdf** présentant clairement le sujet, vos choix de conception (structures de données et fonctions) et de programmation, votre organisation logicielle, un ou des exemples d'exécution, la façon dont votre application se compile et s'utilise (pour que nous puissions facilement la tester), ses limitations ou extensions si elle en a,…

N'oubliez pas vos noms dans tous les fichiers !