INDEX D'UN TEXTE

Le but de ce devoir est de créer une structure permettant un accès très rapide aux mots d'un texte et aux phrases qui les contiennent. Cette structure est basée sur une *table de hachage*; elle utilise des listes chaînées de mots.

Dans toute la suite, on appelle `mot` une suite de caractères ne contenant pas les caractères ' ', '\n', '\t', '.', ';', '?', '!'.
On appelle `position` d'un mot le nombre d'octets qui séparent le début du fichier du début d'une phrase contenant le mot. Deux mots différents peuvent donc avoir la même position.
Une phrase est une suite de mots se terminant un point ('.','?','!').
On confondra minuscules et majuscules (les formes "`ToTo`" et "`toto`" correspondent au même mot).
Pour simplifier, les fichiers traités ne contiennent que des caractères de code ascii compris entre 0 et 128. Il n'y a donc aucune lettre accentuée ("`interne`" et "`Interné`" ont le même statut) ni caractères tel que 'ç'.
On appelle **dictionnaire d'un texte** l'ensemble des mots qui apparaissent dans le texte et **index d'un texte** un dictionnaire dans lequel chaque mot est associé à la liste de ses positions dans le texte.

Le but du devoir est de représenter un index.

On devra pouvoir effectuer les opérations suivantes:

- Test d'appartenance d'un mot au texte.
- Affichage de la liste des positions d'un mot dans le texte.
- Affichage des phrases contenant un mot donné.
- Affichage de la liste triée des mots du texte et de leur position.
- Affichage des mots commençant par un préfixe donné.
- Sauvegarde de l'index

Représentation du dictionnaire

1. **Hachage ouvert**
   On veut diminuer autant que possible le temps d'accès à un mot.
   On range les mots dans des listes chainées, et les listes dans un tableau de N listes.
   On détermine l'indice de la liste dans laquelle ajouter ou rechercher un mot à l'aide d'une fonction h : de l'ensemble des mots dans l 'ensemble des entiers naturels.
   L'indice choisi pour le mot m est $h(m)$ mod N où N est la taille de la table. Tous les mots ayant la même valeur de hachage sont placés dans la même liste.

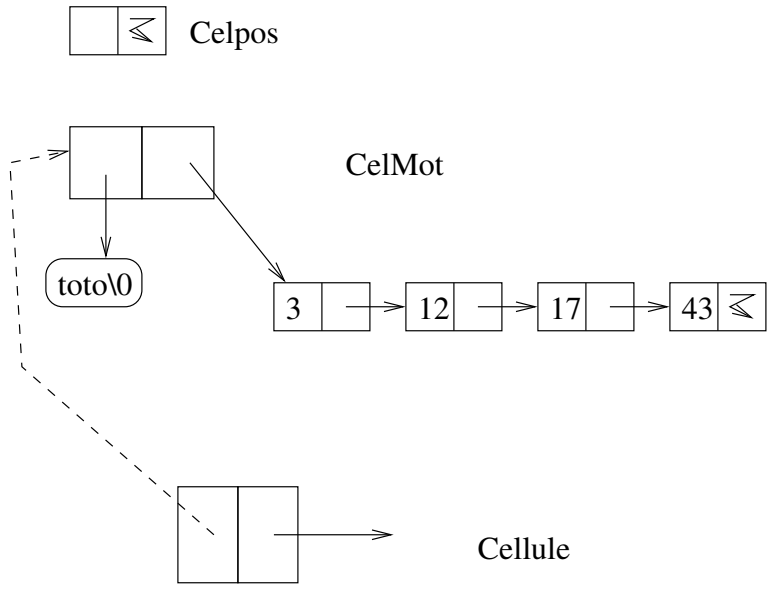
   Le temps moyen de recherche dans une liste dépendant de sa longueur, il faut que les listes soient le plus courte possibles. On doit également pouvoir trouver rapidement à quelle liste un mot donné est succeptible d'appartenir.
   Pour éviter de manipuler des listes trop longues, on choisit une grande taille de tableau, et une fonction de hachage éparpillant au maximum les données (toutes les listes ont donc à peu près le même -petit- enombre d'éléments). Une telle fonction pour des mots est fournie en annexe.
   Les mots seront ajoutés aux listes en ordre lexicographique, chaque mot étant relié à la liste de ses positions.
   Si un mot apparait plusieurs fois dans une phrase, on ne conserve qu'une fois la position.

   On utilisera les structures:

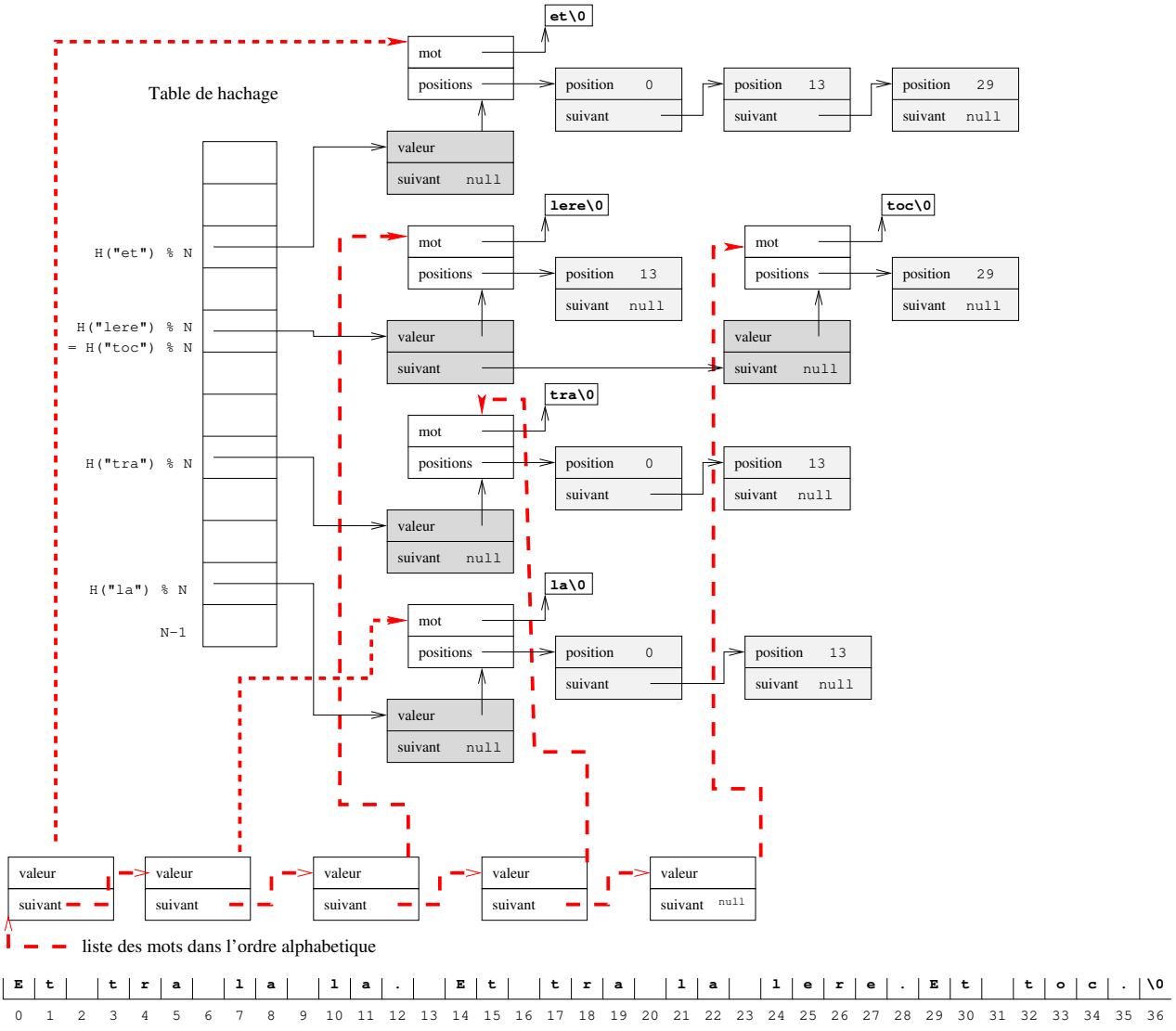
```
typedef struct celpos{
      int position;
      struct  celpos *suivant;
}Celpos, *Listepos;/* pour les positions des mots*/

typedef struct celmot{
      char *mot;
      Listepos positions;
}Celmot;/* pour les mots et l'acces aux positions

typedef struct cellule{
      Celmot *valeur;
      struct cellule *suivant;
}Cellule, *Liste; /* pour les listes chainées de mots*/
```



Les positions d'un mot forment une liste `Listepos` de `Celpos`.
On accède à un mot et sa liste de position par une cellule `Celmot`.
On accède à une `Celmot` via `Cellule`.
Les `Cellule` forment les `Liste` de la table de hachage.
La table de hachage est un tableau de `N Liste`.



Commencer par construire un simple dictionnaire avant de le transformer en index, puis de construire la liste triée.

2. **Liste triée**
   Une fois l'ensemble des mots du texte ajouté, on construira une `Liste` triée des mots du texte. Les champs `valeur` des cellules de cette liste pointent sur les `Celmot` présentes dans la table de hachage, prises dans l'ordre lexicographique. Pour construire cette liste on pourra s'inspirer du tri fusion.

Compilation et exécution

Le programme devra pouvoir être compilé sur les machines de l'université. La compilation produira un exécutable de nom `Index`.
Lancé sans paramètre, il donnera accès à un menu permettant de tester toutes les fonctionalités.
Les tests devront également pouvoir être effectués à l'aide de la ligne de commande sous forme:
`Index -question [mot] Fichier` où:

- l' option `question` précise l'interrogation.
  - `-a mot` teste l'appartenance de `mot` au texte.
  - `-p mot` affiche la suite des positions de `mot` dans le texte.
  - `-P mot` affiche les phrases du texte contenant `mot` .
  - `-l` affiche la liste triée des mots du texte.
  - `-d mot` affiche l'ensemble des mots du texte ayant `mot` pour préfixe.
  - `-D` sauve dans un fichier la liste triée des mots du texte et leur position dans un fichier. Le fichier aura pour nom le nom du fichier entré suivi du suffixe ".DICO". Chaque ligne de ce fichier sera constituée d'un mot suivit de la liste de ses positions.
- le texte traité est contenu dans `Fichier`

Remise et soutenance

Le projet est à effectuer en binôme (**2** étudiants). Si les membres du binôme ne sont pas dans le même TP, les deux chargés de TP devront avoir donné leur accord.
Les sources et un rapport de projet devront ête déposés sur `moodle` , sous forme d'une archive `.zip` de titre `Nom1Nom2.zip` Une soutenance pourra être organisée.
Le rapport de projet est un document décrivant:
• partie utilisateur : ce que fait le projet, comment le compiler, comment l'utiliser;
• partie développeur : comment est réalisé le projet: description et justifications des méthodes utilisés et structure du programme les réalisant. Cette partie n'est pas une liste exhaustive des fonctions; elle doit permettre à un programmeur de bon niveau de reprendre votre programme et de pouvoir le modifier sans difficultés excessives. On détaillera en particulier la construction de la liste triée des mots et on justifiera la méthode choisie en du point de vue de la complexité.

Annexes

- Des fichiers textes sont disponibles à l'URL `abu.cnam.fr`. On ne traitera que des textes écrits avec le code ASCII 128 bits. Un programme remplaçant les caractères de code ascii ≥ 128 dans ces fichiers vous est fourni (il ne fonctionne que pour des textes utilisant le codage ISO-Latin-1).

- Une fonction de hachage possible pour les chaines de caractères est:

```
unsigned int hache(char *p){
unsigned  int h=0,g;
for(;*p;p++){
    h=(h<<4)+*p;
    if(g=h&0xf0000000) /*il y a des valeurs dans les 4 bits de poids fort*/
      {
        h=h^(g>>24);/*elles vont influencer  l'octet de poids faible*/
        h=h^g;/*on les supprime du haut de h*/
      }
  }
 return h;
}
```

- La fonction `int fseek(FILE *stream, long position, int whence);` de <stdio.h> utilisée sous forme `fseek(stream,position,SEEK_SET)` place le pointeur de lecture-écriture à `position` octets du début du fichier manipulé par `stream` .`SEEK_SET` est une macro-constante définie dans <stdio.h>.

- Remarque: c'est grace au hachage que sont implémentés les dictionnaires de python. Ceci explique pourquoi seuls des objets non-mutables peuvent servir de clef.

Bonus

Tester d'autres fonctions de hachage ainsi que plusieurs tailles de table.
Déterminer:

- La proportion de liste vides.

- La longueur moyenne des listes non vides et l'écart-type.

- ...