Travaux Pratiques
Séance nr. 2 (semaine du 08/02/2021)

Dans cet exercice on se propose de définir **une table de chaînes de caractères** pour réaliser **un dictionnaire de mots**. Un dictionnaire de langue française permet par exemple d'associer une définition à un mot. Un dictionnaire de langue étrangère permet d'associer un mot d'une autre langue, etc. Ici, l'objectif du dictionnaire est de pouvoir vérifier si un certain mot existe ou pas. Nous utilisons un adressage associatif avec un indexage calculé (hachage) pour définir un rangement dispersé.

1. Table de chaînes de caractères

La table associe **un identifiant (entier non signé strictement positif)** à une chaîne de caractères « quelconque » (le mot). Par exemple le mot peut être un nom et l'identifiant un numéro de téléphone. Nous allons nous limiter à des mots composés de lettres minuscules et sans caractères spéciaux.

T : char * ----> unsigned int

La valeur d'identifiant 0 correspond à Ω : identifiant invalide => le mot n'est pas dans la table. Les identifiants ont donc des valeurs supérieures ou égales à 1. Pour vérifier qu'un mot existe dans le dictionnaire il suffit de vérifier que son identifiant est différent de Ω.

Pour le **rangement des chaînes et des identifiants,** on se propose d'utiliser deux tableaux contigus de taille bornée par N. Nommons ces deux tableaux ***mots[N]*** et ***ident[N]***. Le mot et son identifiant sont liés par adressage associatif (mêmes indices pour les mots et les identifiants). Le hachage est basé sur un générateur de nombres « aléatoires » : le générateur de Borland C++, défini par :

$X_{n+1} = (((22695477\ X_n + 1) \bmod 2^{32}) >> 16)$

où Xn forme une séquence de nombres dits *pseudo-aléatoires*.

Nous allons faire **un hachage selon le mot**. Comparé à un rangement ordonné par ordre alphabétique, le hachage permet d'accélérer la recherche et l'insertion. Concrètement, le rangement de la chaine dans le tableau ***mot[]*** utilise un hachage selon la formule suivante : X_0 est la somme des indices des caractères du mot à ranger ('a' -> 1, 'b' -> 2 etc.) multiplié par leur rang au sein du mot. Par exemple :

X_0(« lucas ») = 12*1 + 21*2 + 3*3 + 1*4 + 19*5

L'index de rangement ***idx*** est obtenu en calculant $idx = X_1 \bmod N$, ***N*** étant le nombre maximal d'éléments (on prendra ***N=100*** ici). L'identifiant est rangé dans ***ident[]*** au même emplacement ***idx***.

La **gestion des collisions** se fera par tables mineures, modélisées sous la forme d'un chainage (on prendra simplement une liste chaînée). Un élément en collision est placé en tête de liste.

La liste n'est pas ordonnée. Voici la structure correspondante :

```
#define N 100

typedef struct s_dico {
       char                    *mots[N]; // les chaines de car
       unsigned int            ident[N]; // les identifiants associés
       listeg                  mots_collid[N];// gestion des collisions
       listeui                 ident_collid[N];
} *Dico;
```

listeg et ***listeui*** sont resp. de type « liste chaînée générique » (le contenu est ***void ****) et « liste chainée d'entiers non signés » (le contenu est ***unsigned int***). Ces listes offrent les opérations usuelles : création, destruction, insertion en tête, suppression du kième, accès au kième, longueur, test de vacuité, recherche d'un élément (pour ce dernier, on renvoie le pointeur vers le maillon ou NULL). Dans le cas de la liste générique, on passera en paramètre un pointeur de fonction si nécessaire, comme vu en TD : en particulier, les éléments doivent être clonés lorsqu'ils sont ajoutés à la liste. Contrairement au TD, ces deux listes utilisent un **chaînage simple** (càd un seul pointeur vers le suivant).

0. Avant de commencer : faire le schéma d'un exemple de table!

1. Implanter les deux types de listes : listeg et listui

2. On implantera ensuite la table « Dico », et en particulier les opérations suivantes :

- Recherche d'un identifiant : il s'agit de récupérer l'identifiant à partir d'une chaîne
  ```
  unsigned int getIdent(Dico *d, char *mot)
  ```
  si le mot n'existe pas dans la table, on renvoie la valeur Ω=0

- adjonction : on donne en paramètre le couple mot / identifiant:
  ```
  void adj(Dico *d, char *mot, unsigned int id)
  ```
  Le mot est cloné en mémoire avec ***strcpy***.

- Suppression :
  ```
  void supStr(Dico *d, char *mot)
  ```
  Donner, le cas échéant, les pré-conditions de ces fonctions.

3. Définir les opérations d'initialisation et de destruction, ainsi que le cardinal et le test de vacuité pour la table *Dico*.

2. Exercice complémentaire optionnel : table « à double sens »

On souhaite ajouter une opération permettant de retrouver un mot à partir de son identifiant :

```
char * getStr(Dico *d, unsigned int id)
```

si l'identifiant n'existe pas, on renvoie NULL Cela permet également de vérifier qu'un identifiant existe.

Implanter cette opération sans changer la structure ***Dico***. Quelle est sa complexité ?

Imaginer une modification de la structure ***Dico*** pour rendre la fonction ***getStr*** plus efficace.

On peut imaginer utiliser par exemple un hachage pour l'identifiant. Ce hachage peut consister par exemple à prendre X_0=id. Une position de rangement est alors obtenue en calculant $\boldsymbol{X_1 \bmod N}$.

Noter que l'on ne souhaite pas modifier la recherche des mots (la fonction ***getIdent***) : remplacer trivialement le hachage par mot précédent par un hachage selon l'identifiant rendrait ***getIdent*** peu efficace. Il faut donc s'arranger pour que les deux fonctions, ***getIdent*** et ***getStr*** soient toutes les deux efficaces. En pratique, cela revient à créer la table symétrique :

T' : unsigned int ----> char *

en supposant que les identifiants sont uniques.

En clair : la structure ***Dico*** doit contenir **deux tables,** qui se partagent les mêmes données.