

## Travaux Pratiques

### Séance nr. 2 (semaine du 07/02/2022)

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 unique (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  $\Omega$  : 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  $\Omega$ .

Pour le **rangement des chaînes et des identifiants**, on se propose d'utiliser une structure qui associe le mot à son identifiant :

```
#define LONG_MOT_MAX = 64
typedef struct s_mot {
    char          chaine[LONG_MOT_MAX]; // le mot
    unsigned int  ident; // l'identifiant associé
} mot;
```

La table consiste en un tableau contigu de taille bornée par N. Nommons ce tableau ***mots[N]***. Le mot et son identifiant sont implicitement liés par adressage associatif (mêmes indices pour les mots et les identifiants). Le type du tableau est un pointeur sur une structure *mot*.

Le hachage est basé sur un générateur de nombres « aléatoires » : le générateur de Borland C++, défini par :

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

où  $X_n$  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 chaîne dans le tableau ***mots[]*** 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 :

$$X_0(\text{« lucas »}) = 12*1 + 21*2 + 3*3 + 1*4 + 19*5$$

L'index de rangement **idx** est obtenu en calculant  $idx = X_i \bmod N$ ,  $N$  étant le nombre maximal d'éléments (on prendra  $N=100$  ici). La **gestion des collisions** se fera par table mineure, modélisées sous la forme d'un chainage (on prendra simplement une liste chaînée générique). 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 {
    mot          *mots[N]; // les chaines de car
    listeg       mots_collid[N]; // gestion des collisions
} *Dico;
```

**listeg** est un type « liste chaînée générique » (le contenu est **void \***). Cette liste doit offrir les opérations usuelles : création, destruction, insertion en tête, suppression du  $i$ ème, accès au  $i$ ème, 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, cette liste utilisera un **chaînage simple** (càd un seul pointeur vers le suivant).

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

### 1. Implémenter des opérations permettant de manipuler des mots (couples chaîne + entier):

```
mot *motNouv(char *s, unsigned int id) ;
void detruireMot(mot *m) ;
mot *cloneMot(mot *m);
void afficheMot(mot *m);
char *chaineDuMot(mot *m);
unsigned int identDuMot(mot *m);
int compareMot(mot *m1, char *nom) ; // compare les chaines
```

### 2. Implanter le type **listeg** avec ses opérations de base.

### 3. On plantera 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 *m)`

si le mot n'existe pas dans la table, on renvoie la valeur  $\Omega=0$

- adjonction : on donne en paramètre le couple mot / identifiant:  
`void adj(Dico *d, char *m, unsigned int id)`

- Suppression :  
`void supStr(Dico *d, char *m)`

Donner, le cas échéant, les pré-conditions de ces fonctions.

**4. 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**

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 ?

Sans l'implémenter, imaginer une modification de la structure *Dico* pour rendre la fonction *getStr* plus efficace.