

Travaux Dirigés n°1 : chaînes de caractères

Exercice 1

Ecrire une fonction `int nombre_caract(char *chaîne)` qui retourne la taille d'une chaîne de caractères.

Exercice 2

Ecrire la fonction `void concatene_chaines(char *debut_chaine, char *ajout)` qui concatène les deux chaînes de caractères (la seconde à la suite de la première).

Exercice 3

Ecrire une fonction `char* min2maj(char *chaine)` qui alloue une chaîne de caractères de la même taille que celle en entrée, recopie la chaîne mais en changeant toutes les lettres minuscules en majuscules.

Exercice 4

Ecrire une fonction `int compte_mots(char *phrase)` qui compte le nombre de mots dans une phrase (on considérera que les mots ne sont séparés que par un seul espace, et ne contient pas de ponctuation).

Exercice 5

Ecrire une fonction `char** decompose_en_mots(char *phrase)` qui décompose une chaîne en un tableau de chaînes de caractères (dont on allouera et renverra un double pointeur).

Travaux Dirigés n°2 : algorithmes de tri

Exercice 1

Ecrire la fonction de recherche dichotomique d'un élément dans un tableau. Discuter de l'intérêt de la méthode en comparant avec une méthode de recherche exhaustive..

Exercice 2

Ecrire le programme de tri par sélection et permutation. Application par exemple au tri par ordre alphabétique d'une liste de noms rangés dans un tableau de chaînes (utiliser la fonction `int strcmp(char *, char *)`).

Exercice 3

Ecrire le programme du tri par bulle.

Exercice 4

Ecrire le programme du tri par décalage et insertion ...

Travaux Dirigés n°2 : algorithmes de tri

Exercice 1

Ecrire la fonction de recherche dichotomique d'un élément dans un tableau. Discuter de l'intérêt de la méthode en comparant avec une méthode de recherche exhaustive.

Exercice 2

Ecrire le programme de tri par sélection et permutation. Application par exemple au tri par ordre alphabétique d'une liste de noms rangés dans un tableau de chaînes (utiliser la fonction `int strcmp(char *, char *)`).

Exercice 3

Ecrire le programme du tri par bulle.

Exercice 4

Ecrire le programme du tri par décalage et insertion ...

Travaux Dirigés n°3 : structures

Exercice 1

Définir la structure permettant de stocker un point. Ecrire les fonctions permettant de saisir un point et d'afficher un point. Définir un tableau permettant de stocker un ensemble de points (version statique et version dynamique).

Exercice 2

Définir la structure permettant de représenter un nombre complexe. Définir les fonctions complexes suivantes (passages par valeur) :

- double imag(Complex z),
- double real(Complex z),
- Complex mul(Complex z1, Complex z2),
- double abs(Complex z),

Réécrire ces fonctions en utilisant le passage par adresses.

Exercice 3

Ecrire un programme de saisie de données pour un répertoire (nom, prénom, téléphone). Ces données doivent être placées dans un tableau de structures, chacune d'elles contenant un enregistrement. Le programme devra contenir une fonction d'affichage de toutes les données.

Travaux Dirigés n°4 : listes chaînées

Exercice 1

Rappeler l'intérêt du passage par adresse des structures.

Exercice 2

Manipulation des listes chaînées : reprendre l'exercice 3 du TD 3 (gestion d'un répertoire).

- Redéfinir la structure représentant une adresse et permettant un chaînage avant des éléments de la liste ;
- Définir la fonction de création et de saisie d'un nouvel élément ;
- Ecrire la fonction d'ajout d'un élément en tête de liste ;
- Ecrire la fonction d'ajout d'un élément en fin de liste ;
- Ecrire la fonction permettant l'affichage de la liste ;
- Ecrire la fonction permettant de retirer un élément de la liste selon le critère suivant :
 nom=... ET prenom=... ;
- Ecrire la fonction permettant de saisir une liste
- Ecrire la fonction permettant de libérer la liste.

TP n°1 C

L'objectif du tp est la manipulation de chaîne de caractères, ainsi que les algorithmes de tri.

Important : Il est impératif de tester chaque fonction écrite dans la fonction `main()` pour s'assurer de son bon fonctionnement. Il n'est pas raisonnable d'écrire tout l'exercice et de se lancer dans l'exécution à la fin. On valide chaque brique séparément et on avance pas à pas.

1 Rappels

1.1 Saisie de chaînes de caractères

La fonction standard `scanf(char *format, ...)` permet de saisir des chaînes de caractères. Mais attention, pour cette fonction le caractère blanc (espace) est également un caractère *séparateur*. `scanf()` ne peut donc saisir une chaîne comportant plusieurs mots séparés en une seule fois. Une autre fonction de la librairie `stdio.h` permet de saisir une chaîne composée de plusieurs mots : `gets(char *)`. Cette fonction cependant est d'un emploi dangereux car aucun contrôle de longueur de la chaîne saisie ne peut être fait. On lui préférera donc une troisième fonction de la même librairie, `fgets(char*, int, FILE*)`, mais qui sera expliquée plus tard avec les fonctions de manipulation de fichiers. Voici un exemple d'utilisation de cette fonction :

```
#include <stdio.h>

void main(void)
{
    char chaine[21];
    printf("saisir une chaine d'au plus 20 caractères : ");
    fgets(chaine,20,stdin);
    printf("chaine saisie = %s\n",chaine);
}
```

Si la longueur de la chaîne tapée par l'utilisateur devait dépasser 20, seuls les 20 premiers caractères seraient pris en compte. Aucune erreur de dépassement de tableau n'est donc possible.

1.2 Tri à bulle

1.2.1 Principe

Le principe consiste à parcourir les éléments de l'ensemble de $i=0$ à $n-1$ en permutant les éléments consécutifs non ordonnés. L'élément le plus grand se trouve alors en bonne position. On recommence la procédure pour l'ensemble de $i=0$ à $n-2$ sauf si aucune permutation n'a été nécessaire à l'étape précédente. Les éléments les plus grands se déplacent ainsi comme des bulles vers la droite du tableau.

1.2.2 Illustration

Le tableau suivant illustre le fonctionnement du tri à bulle sur un tableau d'entiers :

18	10	3	25	9	2
10	18	3	25	9	2
10	3	18	25	9	2
10	3	18	25	9	2
10	3	18	9	25	2
10	3	18	9	2	25
3	10	18	9	2	25
3	10	18	9	2	25
3	10	9	18	2	25
3	10	9	2	18	25
3	10	9	2	18	25
3	9	10	2	18	25
3	9	2	10	18	25
3	9	2	10	18	25
3	2	9	10	18	25
2	3	9	10	18	25

1.2.3 Algorithme

En pseudo-code, l'algorithme est le suivant :

```
k ← N - 1
FAIRE
  POUR i=0 à J-1 FAIRE
    SI t[i] > t[i+1] ALORS
      permuter t[i] et t[i+1]
      permutation=VRAI
    FIN SI
  FIN POUR
TANT QUE permutation=VRAI
```

2 Travail à faire

2.1 Manipulation de chaînes de caractères.

Réalisez et testez, à fur et mesure, chacune des fonctions ci-dessous.

- Ecrire une fonction
`int compte_caract(char *chaine)`
qui compte le nombre de caractères dans une chaîne de caractères.
- Ecrire une fonction
`char * saisie_chaine()`
qui saisie une chaîne de caractères. Puis alloue et renvoie un pointeur sur cette chaîne de caractères.
- Ecrire une procédure
`void affiche_chaine(char *chaine)`
qui affiche à l'écran, la chaîne de caractère passée en argument.
- Ecrire une fonction
`int compte_espace(char *chaine)`
qui compte le nombre d'espaces dans une chaîne de caractères.
- Ecrire une fonction
`int compte_mot(char *chaine)`
qui compte le nombre de mots dans une phrase simple (on supposera que les mots sont séparés par un caractère espace unique).
- Ecrire une fonction
`char *maj2min(char *mot)`
qui convertit toutes les majuscules d'un mot en minuscules.

2.2 Tri de chaînes de caractères

Utilisez les fonctions et les procédures précédentes pour réaliser les fonctions suivantes.

- Ecrire une fonction
`char **decompose_mot(char *chaine)`
qui décompose une phrase en un tableau de chaînes de caractères; chaque case du tableau comptera un mot de la chaîne passée en argument d'entrée.
- Ecrire une fonction
`int compare_ch(char *ch1, char*ch2)`
qui compare deux chaînes de caractères ch1 et ch2; cette fonction renvoie 0 si les deux chaînes de caractères sont égales. Si les deux chaînes de caractères ne sont pas égales, la fonction renvoie la différence en code ascii des deux premiers caractères différents dans la chaîne.
- Créer une fonction
`char **compare_ch(char **chaîne, int Nb_mots)`
qui trie un tableau de chaînes de caractères par ordre alphabétique, à l'aide de l'algorithme de tri à bulle. Il est important de créer des fonctions et procédures pour réaliser les sous tâches comme la permutation,...
- Créer une fonction principale
`void main()`
utilisant toutes les procédures et fonctions précédentes.

TP n°2 C

Le but du tp est de vous aider, à vous familiariser avec les concepts d'allocations dynamiques et de structures de données.

Important : Il est impératif de tester chaque fonction écrite dans la fonction `main()` pour s'assurer de son bon fonctionnement. Il n'est pas raisonnable d'écrire tout l'exercice et de se lancer dans l'exécution à la fin. On valide chaque brique séparément et on avance pas à pas.

3 Préparation du TP

Il est important de procéder au travail de préparation sans lequel il serait illusoire de penser finir le TP. Prenez le temps de comprendre les bases et de faire les exercices préparatoires, et rappelez vous que ce qui importe c'est le chemin qui mène à la solution et non la solution elle-même. Sans les connaissances de base ce tp n'est en aucun envisageable. Préparez aussi l'ensemble des exercices avant de venir, la séance de tp doit être un échange entre vous et l'encadrant et non une séance de découverte. La préparation du TP est donc primordiale. La préparation aborde simplement les connaissances de bases nécessaires.

3.1 Allocation dynamique et tableaux

Dans la fonction `main()`, chronologiquement :

- Déclarer une variable **a** et un pointeur **p** de type **int**.
- Initialiser **p** à partir de l'adresse de **a**. Initialiser **a** à la valeur **10** par l'intermédiaire de **p**. Afficher **a** et le contenu de l'adresse pointée par **p**.
- Allouer de la mémoire de la taille d'un **int** et stocker l'adresse de retour dans **p**.
Question : est il nécessaire d'appeler la fonction **free** avec cette allocation ?
- Remplir la case allouée avec la valeur **4**, afficher le contenu de la case.
- Déclarer la variable **b** et la remplir en additionnant le contenu de **a** avec le contenu de la case allouée. Afficher **b**.
- Cette fois **p** demande la mémoire en fonction d'une entrée utilisateur. Initialiser toutes les cases du tableau ainsi alloué à la valeur **0**.
Question : que convient il de faire avant d'allouer le tableau ? comment s'appelle cette erreur ?

3.2 Variables qui contiennent l'adresse d'un pointeur

- Ecrire dans le `main()` les code suivant :

```
int a, *b,**c;
```

Quel est le type de chaque variable ?

- Initialiser **b** à partir de **a**.
- Initialiser **c** à partir de **b**.
- Initialiser **a** à partir de **c** à la valeur **10**.
- Afficher deux fois le contenu de **a** à l'écran en passant par **b** et **c**.

3.3 Mise au point sur les structures de données

Soit la structure de données suivante :

```
typedef struct mas
{
    int elem1;
    float elem2;
}MAS;
```

- Déclarer au sein de la fonction `main()` deux variables **a**, **b** de type `MAS`. Initialisez les variables avec des valeurs, et procédez au transfert du contenu de la variable **b** dans **a**. Comment feriez vous ?
- Quel est le type de chacune des variables suivantes : **a.elem1**, **a.eleme2**. Ecrire en utilisant un `scanf` la ligne qui permet de remplir ces champs.

4 Travail à faire

On commencera la séance par préalablement tester sur machine ce que vous avez préparé. Dans un second temps on passera aux exercices qui suivent.

4.1 Exercice

Les trois premières questions sont assez classiques (c'est du cours), celles qui suivent sont une application du concept d'un pointeur qui contient l'adresse d'un autre pointeur. Ces fonctions sont très courtes car elles vont chacune appeler les trois premières fonctions écrites.

- Ecrire une fonction `int* allouertab(int)` ; qui reçoit un nombre d'éléments et alloue un tableau qui a le nombre de cases désiré.
- Ecrire la fonction `void remplirtab(int*, int)` ; qui reçoit un tableau et son nombre de cases et procède à la saisie par l'utilisateur du contenu de ses cases.
- Ecrire la fonction `void affichertab(int*, int)` ; qui reçoit un tableau et son nombre de cases et procède à l'affichage de son contenu.
- Ecrire une fonction `int** allouersupertab(int)` ; qui crée un tableau où chaque case de ce tableau (dont le nombre de cases et donné en argument) est une adresse sur un `int`. Concretement chaque case du tableau contiendra un `int*`.
- Ecrire la fonction `void remplirsupertab(int**,int)` ; qui reçoit le tableau d'adresses sur `int` et son nombre de cases, et procède au remplissage du tableau d'adresses. Le remplissage doit s'effectuer en stockant dans chaque case du tableau d'adresses, l'adresse d'un tableau d'entiers qui proviendra de l'appel de la fonction `allouertab` précédemment écrite.
- Ecrire la fonction `void remplirsupertabtab(int**,int)` ; qui reçoit le tableau d'adresses et son nombre d'éléments et procède au remplissage des tableaux d'entiers (dont l'adresse de début est stockée dans les cases du tableau) par l'utilisateur. On utilisera pour cela la fonction `remplirtab` précédemment écrite.
- Ecrire la fonction `void affichesupertabtab(int**,int)` ; qui reçoit le tableau d'adresses et son nombre d'éléments et procède à l'affichage des tableaux d'entiers pointés par les cases du tableau. On utilisera pour cela la fonction `affichertab`.

4.2 Exercice

Soit la structure de données suivante :

```
typedef struct lss
{
    int nb,*tab;
}LSS;
```

- Ecrire la fonction `LSS* creerstructure(int)` ; La fonction commence par allouer de la mémoire pour un `LSS`. La fonction reçoit un entier qui servira aussi à initialiser le champ **nb**. On utilise alors le champ `nb` duement rempli pour allouer un tableau d'entiers qui aura `nb` cases. On stockera l'adresse de ce

tableau dans le champ `tab`. Le tableau alloué devra être initialisé à zéro. On retournera l'adresse de la structure `LSS` que l'on aura alloué au sein de la fonction.

- Ecrire la fonction **`void affichestructure(LSS*)`**; qui reçoit l'adresse d'une variable de type `LSS` et procède à l'affichage de tous ses champs de données, le contenu du tableau alloué y compris, bien sûr s'il existe (il se peut qu'il n'a pas été alloué).
- Ecrire la fonction **`void transfertcontenu(LSS*src, LSS*dest)`**; qui transfère le contenu d'une variable de type `LSS*` dans une autre variable de type `LSS*`. Les deux variables étant passées par adresse. On se rappellera que le transfert devra transférer le contenu intégral du tableau de la source vers celui de la destination. En effet si le tableau de la destination ne convient pas avec la taille de la source (ou bien trop petit ou trop grand), on désallouera le tableau de la destination pour le réallouer pour qu'il soit de la même taille que celui de la source, et on mettra à jour le champ `nb` de la destination

Question : Que peut-on dire de l'affectation dans le cas de cet exercice comparé à celui demandé dans la préparation du tp ? qu'est-ce qui change ?

- Ecrire une fonction **`void remplirtablss(LSS* var,int ind, int val)`**; qui reçoit l'adresse d'une variable de type `LSS` ainsi que deux variables de type `int`. Le but de la fonction est de ranger le contenu de la variable `val` à la case dont l'indice est contenu dans la variable `ind`. Tout cela au sein du tableau alloué par la variable `var` de type `LSS` passée par adresse.
- Dans la fonction `main()`, créer dynamiquement un tableau de type `LSS` à partir d'une valeur entrée par l'utilisateur. Ecrire une fonction **`void saisirtout(LSS*)`**; qui sert à remplir le tableau et une fonction **`void affichetout(LSS*)`**; laquelle comme son nom l'indique affiche le contenu de chaque case du tableau.

TP n°3 C

5 Préparation du TP

Il est impératif d'avoir terminé le TP précédent car les listes chaînées ne sont qu'une variation autour du thème de la structure. Ce TP est très long et demande une bonne préparation.

5.1 Variation autour du thème de la structure

On considère les deux structures suivante :

```
typedef struct mastruc      typedef struct liste
{
  int nb;                  {
  int *t;                  int nb;
  } MASTRUC;              struct liste *suivant;
                          } LISTE;
```

- La structure de droite est celle vue dans le TP précédent. Celle de gauche diffère sur un point très léger, lequel ?
- Dans la fonction `main()` déclarez deux variables de type **LISTE**, **a** et **b**. Remplir les champs de chaque variable et initialiser les pointeurs à **NULL**. Pour chaque variable, afficher le contenu du champ **nb**.
- Initialiser le champ **suivant** de **a** avec l'adresse de **b**. Maintenant en utilisant juste la variable **a** afficher à l'écran le contenu des champs **nb** pour **a** et **b**.
- On déclare maintenant deux pointeurs de type **LISTE**, ***p** et ***q**, reprendre les questions précédentes (sans utiliser ni **a** ni **b**) en utilisant cette fois seulement les deux pointeurs.

6 Travail à faire

On commencera la séance par préalablement tester sur machine ce que vous avez préparé. Dans un second temps on passera aux exercices qui suivent.

Pour chaque question de chaque exercice il conviendra de tester chaque fonction séparément, on procède pas à pas en testant chaque fonction dans le `main()` avant de passer à la question suivante.

Il y aura un fichier par exercice.

6.1 Exercice

Soit la structure suivante :

```
typedef struct liste
{
  int nb;
  struct liste *suivant;
}LISTE;
```

- Ecrire la fonction **LISTE *creer_maillon()** qui alloue de la mémoire pour un élément de type **LISTE** et retourne son adresse. On initialisera impérativement **suivant** à **NULL**. Question : Pourquoi retourne-t-on l'adresse de l'élément alloué ? aurais-t-on pu déclarer tout simplement une variable dans la fonction et retourner son adresse ? pourquoi ? . Pourquoi initialiser le champ **suivant** à **NULL** ?

- Ecrire la fonction `void affiche_maillon(LISTE*)` qui affiche le contenu d'un élément de type `LISTE` passé par adresse et ne retourne rien. Question : Pourquoi ne retourne-t-on rien ?
- Ecrire la fonction `LISTE *chainage_avant(LISTE*,LISTE*)` qui reçoit la tête de la liste chaînée et l'adresse du maillon à insérer et procède à son insertion à la première position. On retourne l'adresse du premier maillon de la liste chaînée. Question : pourquoi retourne-t-on l'adresse de la liste chaînée ?
- Ecrire la fonction `LISTE *chainage_arriere(LISTE*,LISTE*)` qui reçoit la tête de la liste chaînée et l'adresse du maillon à insérer et procède à son insertion à la dernière position. On retourne l'adresse du premier maillon de la liste chaînée.
- Ecrire la fonction `void affiche_liste(LISTE*)` qui reçoit la tête de la liste chaînée et affiche le contenu du champ `nb` de tous les maillons.
- Ecrire la fonction `void supprime_liste(LISTE*)` qui reçoit la tête de la liste chaînée et procède à la libération mémoire de tous ses maillons.
- Ecrire la fonction `LISTE *chaine_listes(LISTE*, LISTE*)` qui reçoit deux listes chaînées et les chaîne pour n'en former plus qu'une, on retournera la tête de la nouvelle chaîne.
- Dans le `main()` créer deux listes chaînées, les remplir à partir d'entrées utilisateurs (on utilisera un type de chaînage par liste). Les listes chaînées devront avoir au minimum 10 éléments. Les afficher puis chaîner les deux listes, pour n'avoir plus qu'une seule. Afficher le contenu de la nouvelle liste puis on supprimera tous les maillons alloués.

6.2 Exercice

Soit la structure suivante :

```
typedef struct liste
{
    int nb;
    char nom[30], tel[10];
    struct liste *suivant,*precedent;
}LISTE;
```

La seule différence avec l'exercice précédent tient au fait que cette liste est doublement chaînée, c'est à dire que chaque maillon possède l'adresse de l'élément qui le suit mais aussi de celui qui le précède.

- Reprendre les six premières questions de l'exercice précédent avec la nouvelle structure.
- Ecrire la fonction `LISTE *trouver_element(LISTE*, int)` qui reçoit la liste chaînée et un entier. La fonction doit parcourir la liste à la recherche d'un élément dont le champ `nb` serait égal à l'entier passé en argument. On retournera l'adresse de ce maillon, si jamais l'élément n'existe pas on retournera `NULL`.
- Ecrire la fonction `LISTE *supprime_maillon(LISTE*, LISTE*)` qui reçoit la tête de la liste chaînée et l'adresse du chaînon à supprimer et retourne la tête de la liste. Question : quel est l'intérêt de retourner la tête de la liste dans ce cas ci ?
- Ecrire la fonction `LISTE *trouve_et_supprime(LISTE*, int)` qui reçoit la liste chaînée et un entier et procède à l'élimination de tous les maillons dont la valeur est multiple de l'entier passé en argument. Rappel : n'oubliez pas les vertues de l'opérateur `%` sur les entiers.
- Ecrire la fonction `LISTE *trier_liste(LISTE*)` qui reçoit une liste chaînée et procède à son tri par la méthode du tri à bulle. On retournera la tête de la liste chaînée ainsi triée. Attention : on n'utilisera pas une nouvelle liste chaînée.
- Ecrire la fonction `LISTE *insérer_element(LISTE*,LISTE*)` qui reçoit une liste chaînée et l'adresse d'un maillon et procède à son insertion au sein de la liste à la place qui conviendra pour qu'elle reste triée. On retournera la tête de la liste chaînée.
- Dans le `main()` écrire l'application qui permet de gérer un répertoire téléphonique en utilisant l'ensemble des fonctions écrites. Note : on utilisera la fonction `switch` pour les menus.

TP n°4 C, Révision Générale

7 Problème

On considère la structure doublement chaînée suivante :

```
typedef struct unmot
{
    char *mot;
    struct unmot *suiv,*prec;
} UNMOT;
```

- Le but de la liste chaînée à écrire est de contenir des mots (chaînes de caractères), chaque maille de la liste devra contenir un mot que l'utilisateur aura entré. Ecrire les fonctions qui gèrent une liste chaînée de type **UNMOT**, on aura besoin de chaînage arrière, d'affichage de la liste, de suppression de la liste, affichage et allocation d'une maille ... et de tout ce que vous jugerez utile par la suite. Toutes ces fonctions doivent maintenant être écrites de manière rapide et ne doivent plus présenter une difficulté particulière.
- Ecrire la fonction void decoupephrase(char*) qui reçoit une phrase dans une chaîne de caractères et qui procède au découpage de la phrase en mots qu'il faudra afficher un à un à l'écran. On supposera que les mots sont séparés par un seul espace. On utilisera fgets pour saisir la chaîne de caractères.
- Modifier la fonction précédente mais cette fois au lieu d'afficher les mots, on les stockera dans une liste chaînée de type UNMOT. Cette liste chaînée contiendra à la fin du découpage de la phrase, une maille par mot. La fonction aura pour prototype UNMOT* decoupephraseenliste(char*); elle recevra la chaîne de caractère et retournera la tête de la liste chaînée ainsi remplie.
- Allouer dans le main un tableau que portera le nom de Texte qui aura DIM cases (DIM défini en #define), le tableau contiendra des adresses sur le type UNMOT* (type UNMOT **).
- Ecrire la fonction void remplirligne(UNMOT **, UNMOT*, int); qui reçoit une liste chaînée contenant des mots et le tableau Texte (déclaré dans le main) et procède au stockage de la tête de la liste chaînée dans le tableau Texte à la case donnée en argument int.
- Ecrire la fonction int remplirtexte(UNMOT**), qui reçoit le tableau texte et procède à son remplissage à partir des entrées utilisateurs. Le procédé de remplissage est le suivant : L'utilisateur tape une phrase qui est stockée dans une chaîne de caractères, on suppose que la chaîne de caractères n'excède pas 80 caractères. On découpe la phrase en mots et on stocke les mots dans une liste chaînée. On stocke la tête liste chaînée contenant les mots dans le tableau Texte à la case qui correspond au numéro de la ligne en cours. Le numéro de ligne est initialisé à zéro et s'incrémente selon le nombre de phrases entrées. On retournera le nombre de lignes saisies.
- Ecrire la fonction void Affichetexte(UNMOT**,int) qui reçoit le tableau Texte et le nombre de ligne et affiche tous les mots stockés.
- Ecrire une fonction qui transforme tous les mots de Texte de minuscules à majuscules. void Minamaj(UNMOT**,int) on reçoit le tableau et le nombre de lignes.
- Ecrire la fonction void supprimeligne(UNMOT**,int,int), qui reçoit le tableau et le nombre de lignes et le numéro de la ligne à supprimer et qui procède à l'élimination de la ligne. On procédera à une destruction de la liste chaînée puis à un décalage pour combler la disparition de la ligne. Ecrire la fonction void supprimetout(UNMOT**,int) qui supprime toutes lignes.
- Ecrire la fonction void remplacemot(UNMOT**,char*,char*,int) qui reçoit le tableau Texte et son nombre de lignes et deux mots stockés dans deux chaînes de caractères et procède au remplacement du premier mot par le deuxième dans tout le texte saisi.