UNIVERSITE CHEIKH ANTA DIOP **DE DAKAR**

ECOLE SUPERIEURE POLYTECHNIQUE

DEPARTEMENT GENIE INFORMATIQUE

ANNÉE UNIVERSITAIRE
2023/ 2024

Algorithmique et programmation avancées - DIC2 – Suite d'exercices

Exercice 1 : Listes chaînées et chaînes de caractères

On désire stocker en mémoire une suite de mots entrés au clavier. Ne connaissant pas à l'avance le nombre de mots qui seront lus ni leur taille, on décide de les stocker dans une liste chaînée dont les maillons pointent sur des chaînes de caractères.

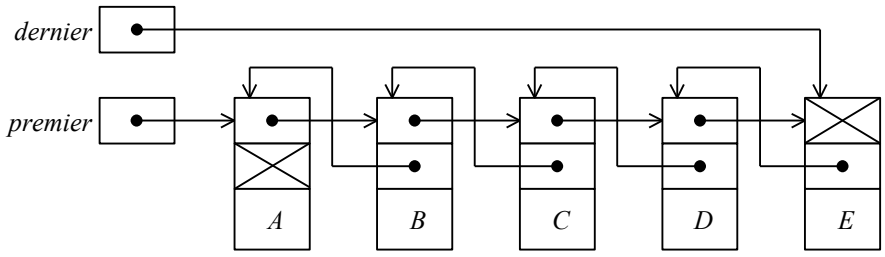
```
struct maillon{ char *mot ; struct maillon *suiv ;}

typedef struct maillon MAILLON, *PTR ;
```

Ecrire les fonctions suivantes :

1. ***PTR ajoute_debut(char *mot, PTR L)*** qui crée un maillon pour mot, le rajoute en tête de la liste chaînée L, et retourne la nouvelle liste.
2. ***PTR ajoute_fin(char *mot, PTR L)*** qui crée un maillon pour mot, le rajoute en fin de la liste chaînée L, et retourne la nouvelle liste.
3. ***PTR supprimer(char *mot, PTR l)*** qui supprime le maillon qui contient le mot considéré dans la liste L et retourne la nouvelle liste.
4. ***void premiers(PTR liste, int n)*** qui affiche les n premiers mots de la liste.
5. ***void purifie(MAILLON *liste)*** qui ne conserve dans liste qu'une seule occurrence d'un mot représenté plusieurs fois dans la liste chaînée triée liste.

Exercice 2 : Listes chaînées bidirectionnelles de chaînes de caractères

Une *Liste Chaînée Bidirectionnelle* (ou *LCB*) est constituée de maillons, chacun comportant une information dépendant du problème considéré et deux pointeurs : un vers le maillon suivant et un vers le maillon précédent :



1. En supposant que l'information dépendant du problème considéré soit de type *chaîne de caractères*, écrivez la déclaration de la structure *maillon* et des variables globales *premier* et *dernier* nécessaires à la réalisation d'une *LCB* comme indiqué sur la figure ci-dessus.
2. Ecrivez la fonction

```
void ajouter_devant(char *s);
```

qui crée un nouveau maillon associé à la chaîne *s* et l'ajoute en tête de la liste.

3. Réécrire la fonction *ajouter_devant* en supposant que *premier* et *dernier* sont des paramètres de la fonction, non des variables globales.
4. Ecrivez la fonction

void supprimer(char *s);

qui supprime de la *LCB* le premier maillon portant l'information représentée par *s*, s'il existe. Cette fonction accède aux variables globales *premier* et *dernier*. Notez que l'existence d'un double chaînage permet de parcourir la liste avec un seul pointeur.

Exercice 3: Traitement de listes chaînées

On considère des listes chaînées d'entiers.

Faire les déclarations nécessaires, et écrire en C, les procédures et fonctions permettant d'effectuer les opérations suivantes :

1. Créer une liste avec dix valeurs saisies au clavier.
2. Tester l'égalité de deux listes.
3. Concaténer deux listes :
    a) Dans une troisième liste.
    b) Sans créer une troisième liste.

Exercice 4. Dérivée d'un polynôme

On veut représenter un polynôme par une liste chaînée dans laquelle on ne représente que les coefficients non nuls.

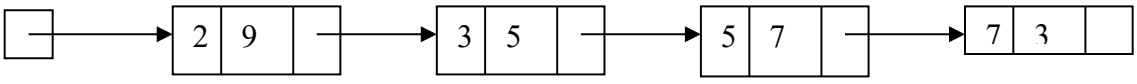
1. Donner la structure permettant de faire cette représentation.
2. Donner une fonction DERIVEE(P) qui reçoit en entrée un polynôme et donne en sortie la dérivée P' de ce polynôme.
3. Donner une fonction DERIVEEKIEME(P,k) qui reçoit en entrée un polynôme P et donne en sortie sa dérivée k-ième.

Exercice 5 : Vecteurs creux

Etant donné un entier $n > 0$, un vecteur à n composantes est une suite de n nombres réels $(x_0, x_1, .., x_{n-1})$.

On s'intéresse à des vecteurs dont beaucoup de composantes sont nulles, si bien que, pour économiser la mémoire, on décide de les représenter par des listes chaînées de maillons dynamiquement alloués dans lesquelles seules apparaissent les composantes x_i vérifiant $x_i \neq 0$.

Une telle liste sera appelée vecteur creux. Chaque maillon est un triplet (i, x_i, s) où i est l'indice d'une composante, x_i sa valeur et s un pointeur vers le maillon suivant de la liste. Par exemple, le vecteur creux dessiné ci-dessous correspond au vecteur (0, 0, 9, 5, 0, 7, 0, 3, 0, 0).



On décide en outre qu'un vecteur creux doit être ordonné par l'ordre croissant des indices des composantes : si (i, x_i, s) et (j, x_j, s') sont deux maillons consécutifs, alors $i < j$.

1. Déclarer en C des types **MAILLON** et **PTR** permettant de réaliser des vecteurs creux.
2. Ecrire une fonction **nouveau_maillon** permettant de créer un nouveau maillon avec des valeurs initiales fournies en arguments.
3. Ecrire une fonction **PTR vecteur_creux (float t[ ], int n)** qui prend un tableau **t** ayant **n** éléments et construit sa représentation sous forme de vecteur creux.
4. Ecrire une fonction **PTR somme (PTR a, PTR b)** qui reçoit deux vecteurs creux **a** et **b** et retourne le vecteur creux qui représente leur somme (c'est-à-dire l'addition des deux vecteurs, composante par composante).

Exercice 6 : Matrices symétriques

On considère des programmes manipulant des matrices carrées à *n* lignes et *n* colonnes à coefficients réels tels que $n \leq$ NMAX, NMAX une constante donnée.

1. Déclarer un type *MATCARREE* permettant de représenter de telles matrices au moyen d'un tableau.
2. Ecrire une fonction *symetrique* qui permet de vérifier qu'une matrice *m* de type *MATCARREE*, représentant une matrice carrée *nx n* est symétrique.

On s'intéresse maintenant à la représentation compacte des matrices symétriques, c-à-d celle qu'on obtient en ne mémorisant qu'un seul des coefficients $m_{i,j}$ et $m_{j,i}$.

3. Combien faut-il de nombres réels pour représenter sans redondance une matrice symétrique à *n* lignes et *n* colonnes ?
4. Ecrire la fonction *double *symCompacte(MATCARREE m, int n)* qui reçoit une matrice carrée à *n* lignes et *n* colonnes et qui construit et renvoie la représentation compacte de m par un tableau dynamiquement alloué. Cette fonction renvoie NULL si m n'est pas symétrique.
5. Ecrire la fonction *double acces(double *c, int i, int j)* qui reçoit *c*, la représentation compacte de *m*, une matrice symétrique *n* x *n,* et deux indices *i* et *j*, tels que $0 \leq i,j \leq n$, et qui renvoie le coefficient $m_{i,j}$.
6. En supposant qu'on dispose par ailleurs d'une fonction *traiterCoef(double x)*, écrire la fonction *void traiterLigne(double *c, int n, int i)* qui parcourt la ieme ligne de la matrice symétrique *n* x *n,* dont *c* est la représentation compacte, en effectuant sur chaque coefficient le traitement exprimé par traiterCoef.
7. Ecrire la fonction *void afficher(double *c, int n)* qui affiche normalement (c-à-d sous une forme matricielle) la matrice symétrique *n* x *n* dont *c* est la représentation compacte. On pourra se servir des fonctions *traiterLigne* et *traiterCoef* (écrire une version utile de cette fonction).

Exercice 7: Terminologie sur les arbres

1°) Ecrire la déclaration d'une structure **ARB_BIN** permettant de représenter un arbre binaire d'entier. Soit A une variable de type **ARB_BIN.**

2°) Ecrire une procédure **FEUILLE (A)** qui affiche la liste des étiquettes des feuilles de **A**, en commençant, à tout moment, par la feuille la plus à gauche.
3°) Le degré d'un nœud de A est son nombre de fils. Les feuilles sont de degré 0. Ecrie une procédure **DEGRE(A)** qui affiche tous les nœuds de A avec leurs degrés respectifs.

4°) Soit x un nœud de l' arbre A. La profondeur de x dans A est la longueur du chemin allant

de la racine de A à x. Ecrie une procédure qui cherche et affiche la profondeur de x.

5°) La hauteur d'un arbre A est le maximum des profondeurs de ses nœuds. Ecrie une procédure **HAUTEUR(A)** qui retourne la hauteur de A.

5. 6°) Ecrire une fonction, **SOM_NOEUDS(A)**, qui calcule et retourne la somme des nœuds de A.

Exercice 8: LCB et ABR

On souhaite construire l'index alphabétique des noms propres cités dans un livre.
Pour cela, on se propose d'utiliser la structure d'un Arbre Binaire de Recherche (ABR).
Le contenu de chaque nœud de l'arbre représente une ligne dans l'index et est composé d'un nom propre et des numéros des pages où il apparaît dans le livre. Ces numéros de page sont chaînés entre eux dans une Liste Chaînée Bidirectionnelle (LCB), *dans l'ordre croissant, sans répétition*. Les pointeurs de tête et de queue de la LCB sont stockés dans le nœud de l'ABR.
A titre d'exemple, considérons le petit index ci-dessous :

Nom	*Pages*
Fatou	110, 250, 300
Mamadou	3, 14, 101
Ousseynou	11, 50
Pierre	3, 7, 100, 287
Soda	6, 10, 34, 66, 98

On déclare les types **MAILLON**, **PTR** et **LCB** définis ci-dessous pour représenter les numéros de page :

```
typedef struct maillon {
        int numero ;
        struct maillon * suiv, *prec ;
} MAILLON, *PTR ;

typedef struct lcb {
        PTR tete, queue;
}  LCB ;
```

1°) En déduire la définition d'un type **ABR** permettant de représenter un index.

2°) Ecrire la fonction **LCB ajout_numero (int num, LCB numeros)** qui insère un numéro dans une LCB et retourne la LCB résultante. La fonction doit exploiter le double chaînage afin de simplifier le parcours de la liste.

3°) Ecrire une fonction **ABR ajout_nompropre (char* nom, int t[], int nombre, ABR a)** qui insère, dans un ABR, un nouveau nom et les numéros des pages dans lesquelles il apparaît. Les numéros de pages sont supposées déjà saisies dans un tableau dont le nombre d'éléments est fourni en argument. La fonction renvoie l'ABR modifié.

4°) Ecrire une fonction **ABR supprimer_numero (char *nom, int numero, ABR a)** qui supprime un numéro de page pour un nom donné et renvoie l'ABR modifié. La fonction doit exploiter le double chaînage afin de simplifier le parcours de la liste.

5°) Proposer une représentation graphique de l'ABR représentant le petit index fourni précédemmant en exemple. On suppose que les noms sont insérés dans l'ABR selon l'ordre suivant : *Fatou*, *Mamadou*, *Oussenou*, *Pierre*, *Soda*.

6°) Ecrire une fonction void afficher_index (ABR a) qui affiche l'index selon l'ordre

alphabétique des noms (comme montré dans l'exemple).

7°) Les performances de la recherche dans l'ABR de l'exemple ne sont pas meilleures que celles qu'on obtiendrait si on s'était servi d'une liste chaînée simple à la place de l'ABR. Pourquoi ? Proposer une solution à ce problème.