Travaux Dirigés – TD 9

Exercice 55. Opérations sur les listes chaînées

Considérons une liste chaînée contenant des étudiants de l'UTT dont chaque étudiant est représenté par :

```
typedef struct student
{
      char nom[20];
      char prénom[100];
      int identifiant;
      char specialite[3];
      struct student * suivant;
} etudiant;
```

Les spécialités considérées sont "GSI", "GSM" et "ISI".

1. Écrire la fonction `etudiant* creerListe(int nbEtudiant)` qui permet de créer une liste chaînée à partir des données de l'utilisateur et de retourner un pointeur vers la tête de la liste.
2. Écrire la procédure `void trier(etudiant* L)` qui permet de trier les étudiants de liste `L` dans un ordre alphabétique croissant selon le nom.
3. Écrire une fonction `int insererEtudiant(etudiant* L, etudiant e)` qui permet d'insérer un étudiant `e` dans la liste `L`. La fonction retourne 1 si l'insertion est bien faite, 0 si l'étudiant existe dans la liste ou un échec d'insertion est survenu. Trier la liste avant l'insertion si elle ne l'est pas.
4. Écrire une fonction `int supprimerEtudiant(etudiant* L, etudiant e)` qui permet de supprimer un élément de la liste `L`. La fonction retourne 1 si la suppression est faite, 0 sinon.
5. Écrire une procédure `void recupererSelonSpecialite(etudiant* L, etudiant** GSI, etudiant** GSM, etudiant** ISI)` qui permet de récupérer à partir de la liste `L` trois listes d'étudiants selon leur branche.

   Nous souhaitons maintenant rajouter une liste d'UVs suivies par chaque étudiant. Chaque UV est nommée selon la convention UTT. L'UV NF05 est enregistrée comme "NF05".
6. Modifier la structure `etudiant` en conséquence.
7. Suite à cette modification, quelles sont les fonctions précédemment écrites à adapter ? Adapter ces fonctions.

Exercice 56. Listes linéaires chaînées VS tableaux

Considérons un tableau `tab` de chaînes de caractères. Un utilisateur souhaite copier les éléments de ce tableau dans une liste linéaire chaînée `L`.

1. Proposer la structure d'un élément de la liste. On définit la structure avec le nom `chaine`
2. Écrire une fonction `chaine* construireListeDeTableau(char** tab)` qui prend un tableau de chaînes de caractères et renvoie une liste d'éléments `chaine`, où chaque élément de la liste représente une case du tableau de chaînes de caractères
3. Écrire une procédure `void occurrencesAlphabetique(chaine* L)` qui prend une liste de chaînes de caractères `L` et affiche le nombre d'apparition de chaque alphabet dans l'ensemble des chaînes de caractères de la liste `L`
4. Écrire une fonction `int chercherSousChaine(chaine* L, char* s)` qui recherche la chaîne de caractères `s` dans toutes les chaînes de la liste `L`. La fonction retourne 1 si la chaîne est trouvée dans la liste, 0 sinon
5. Écrire une procédure `void ecrireListeDansFichier(chaine* L, char* chemin)` qui permet d'écrire toutes les chaînes de caractères de la liste `L` dans le fichier dont le chemin est `chemin`. Les chaînes doivent être séparées par un saut de ligne.

Exercice 57. Matrices dans des fichiers

Un utilisateur souhaite lire une matrice carrée symétrique à partir d'un fichier. Pour réduire la taille occupée par le fichier sur le disque, la représentation de la matrice dans le fichier est la suivante :

— Le fichier est une suite d'entiers séparés par des blancs
— Le premier entier représente le nombre de lignes/colonnes de la matrice, noté `n`.
— Seuls les éléments en dessus de la diagonale de la matrice sont sauvegardés dans le fichier.
— Les `n` entiers suivants le premier entier représentent la première ligne de la matrice. Ensuite, les `n-1` entiers suivants représentent la deuxième ligne de la matrice et ainsi de suite. Pour une matrice :

$$\begin{pmatrix} 10 & 7 & 7 & 3 \\ 7 & 0 & 8 & 4 \\ 7 & 8 & 1 & 6 \\ 3 & 4 & 6 & 2 \end{pmatrix}$$

Le fichier contient : 4 10 7 7 3 0 8 4 1 6 2

Pour réduire la taille de la mémoire utilisée à la lecture du fichier, on utilise une liste linéaire chaînée qui permet de sauvegarder la matrice dans la mémoire. Chaque élément de liste contient un entier. Le type défini pour un élément est `elem`.

1. Définir la structure `elem`
2. Écrire une fonction `elem* ListeMatrice(char* chemin)` qui permet de lire une matrice à partir d'un fichier dont le chemin est `chemin`
3. Écrire une fonction `elem* elementMatrice(elem* L, int ligne, int colonne)` qui permet de récupérer l'élément de la matrice sauvegardée dans la liste `L` dont la ligne et la colonne sont `ligne` et `colonne` respectivement
4. Écrire la fonction `int NombreLignesMatrice(elem* L)` qui permet de donner le nombre de lignes/colonnes de la matrice sauvegardée dans la liste `L`
5. Écrire la procédure `void ecrireMatriceFichier(elem* L, char* chemin)` qui permet d'écrire la matrice sauvegardée dans la liste `L` dans le fichier dont le chemin est `chemin`. La matrice doit être écrite dans sa forme complète, comme présenté ci-dessus. La première ligne du fichier doit contenir un seul entier représentant le nombre de lignes/colonnes de la matrice. Pour la même matrice de l'exemple ci-dessus, le fichier créé par la fonction `ecrireMatriceFichier` doit être :

```
4
10 7 7 3
7 0 8 4
7 8 1 6
3 4 6 2
```