Structures de données : listes chaînées et files

1 Files

1.1 La structure `contenu`

Exercice 1. Définition et fonctions d'un type `contenu`

Soit la structure suivante :

```
#define TAILLEMAX 10
typedef struct {
    char tab[TAILLEMAX];
} contenu;
```

Le tableau de caractères `tab` peut contenir soit une chaîne de `TAILLEMAX-1` caractères ou moins terminée par un `'\0'`, soit `TAILLEMAX` caractères

Définir une fonction d'affichage `void afficher_contenu(contenu *pc)` qui affiche les caractères du champ `tab` du contenu pointé par `pc` jusqu'à, soit rencontrer un `'\0'`, soit en avoir affiché `TAILLEMAX`.

Définir une fonction de saisie `void saisir_contenu(contenu *pc)` qui stocke au maximum 10 caractères dans le champ `tab` du contenu pointé par `pc`, marque un `'\0'` en place d'un éventuel `'\n'`, et vide le buffer clavier.

Définir une fonction de comparaison `int comparer_contenu(contenu *pc1, contenu *pc2)` retournant un entier négatif si le champ `tab` du contenu pointé par `pc1` est avant celui de `pc2` par ordre alphabétique, un entier positif s'il est après et `0` en cas d'égalité.

Définir une fonction de comparaison `int comparer_chaine (contenu *pc1, char *s)` retournant un entier négatif si le champ `tab` du contenu pointé par `pc1` est avant `s` par ordre alphabétique, un entier positif s'il est après et `0` en cas d'égalité. Définir une fonction d'affectation `void affecter_contenu( contenu *pcdest, contenu *pcsource)` qui recopie dans le champ `tab` du contenu pointé par `pdest` les caractères du champ `tab` du contenu pointé par `psource`

1.2 Les files

Exercice 2. Définition et fonctions d'une file

Avec le type `contenu` ci-dessus, définir tous les types et toutes les fonctions nécessaires pour que le programme principal suivant fonctionne :

```
int main() {
    contenu txt;

    file *pf = creer_file();

    for(; ; ) {
        printf("Ajouter un texte a la file (! pour retirer, * pour finir) : ");
        saisir_contenu(&txt);
        if (comparer_chaine(&txt, "*") == 0)
            break;
        if (comparer_chaine(&txt, "!") == 0) {
            if (est_vide_file(pf)) {
               printf("Retrait imposible. File vide");
            }
           else {
              defiler_file(pf, &txt);
            }
            afficher_file(pf);
            printf("\n");
       }
        else {
            enfiler_file(pf, &txt);
            afficher_file(pf);
            printf("\n");
        }
    }
    liberer_file(pf);
    getchar();
    return 0;
}
```

Il devra produire un résultat similaire à l'exemple d'exécution ci-dessous :

```
Ajouter un texte a la file (! pour retirer, * pour finir) : riri
<riri>
Ajouter un texte a la file (! pour retirer, * pour finir) : fifi
<riri,fifi>
Ajouter un texte a la file (! pour retirer, * pour finir) : loulou
<riri,fifi,loulou>
Ajouter un texte a la file (! pour retirer, * pour finir) : !
<fifi,loulou>
Ajouter un texte a la file (! pour retirer, * pour finir) : !
<loulou>
Ajouter un texte a la file (! pour retirer, * pour finir) : !
<>
Ajouter un texte a la file (! pour retirer, * pour finir) : !
Retrait imposible. File vide<>
Ajouter un texte a la file (! pour retirer, * pour finir) : nafnaf
<nafnaf>
Ajouter un texte a la file (! pour retirer, * pour finir) : nifnif
<nafnaf,nifnif>
Ajouter un texte a la file (! pour retirer, * pour finir) : noufnouf
<nafnaf,nifnif,noufnouf>
Ajouter un texte a la file (! pour retirer, * pour finir) : !
<nifnif,noufnouf>
Ajouter un texte a la file (! pour retirer, * pour finir) : *
```

Réfléchir à un moyen de vérifier que tous les `malloc` sont bien appariés par des `free` lors de l'exécution du programme.

Si vous avez écrit tous les programmes dans le même fichier `.c`, réfléchissez à le découper en plusieurs fichiers sources `.c` et en-têtes `.h`, de manière à regrouper les programmes par fonctionnalités (file, contenu, tests).

2 Le tri par éclatement/fusion

Le but de cette section est d'écrire un tri par éclatement-fusion (*mergesort*, **le tri** sur les listes chaînées) dont le principe est le suivant :

- si la liste a au moins deux éléments, scinder la liste en deux listes : une liste contenant un maillon sur deux à partir de la tête de liste initiale et une liste contenant un maillon sur deux à partir du suivant de la tête de liste initiale ;
- trier récursivement chacune des deux listes ;
- fusionner les deux listes triées en une seule.

Ce tri est en $n \log(n)$ et a l'énorme avantage de laisser les éléments en place (seuls les chaînages sont modifiés).

Voici par exemple les opérations nécessaires à trier par ordre alphabétique les éléments d'une liste :

U	→	Y	→	T	→	R	→	E	→	Z	→	A
U	→	T	→	E	→	A		Y	→	R	→	Z
U	→	E		T	→	A		Y	→	Z		R
U		E		T		A		Y		Z		R
E	→	U		A	→	T		Y	→	Z		R
A	→	E	→	T	→	U		R	→	Y	→	Z
A	→	E	→	R	→	T	→	U	→	Y	→	Z

Exercice 3. Créer une liste chaînée

Avec la structure

```
typedef struct _m {
    int n;
    struct _m *svt;
} maillon, *liste;
```

créez une liste de 10 000 éléments entiers aléatoires.

Exercice 4. Scinder une liste

Écrire la fonction qui scinde une liste en deux : la liste des éléments de rang pair dans la liste initiale, et celle des éléments de rang impair.

```
void scission(liste lst, liste *p_lst1, liste *p_lst2;
```

Exercice 5. Fusionner deux listes triées

Écrire la fonction qui fusionne deux listes triées en une seule liste triée.

```
void fusion(liste lst1, liste lst2, liste *p_lst);
```

Exercice 6. Tri fusion

Écrire la fonction qui trie une liste sur elle-même.

```
void tri(liste * p_lst);
```

3 Le codage de Huffman (1)

Exercice 7. Liste des fréquences

Écrivez la fonction `void calcule_freq(FILE *f, unsigned long int t[])` qui remplit un tableau `t` des 256 fréquences d'octets du fichier pointé par `f`.

Créez la liste des fréquences avec les contenus suivants :

```
typedef struct {
    unsigned char c;
    unsigned long int freq;
} contenu;
```

et triez la par ordre de fréquence croissante. Testez-le avec le fichier dico.txt.