Informatique

test pour DST

Programmation avec C et OCaml

Ipesup

On souhaite réaliser une implémentation en langage C des opérations des programmes push-pop sur une structure de données `liste` codant une liste doublement chaînée. Le type de la structure de données `liste` est défini comme suit :

```
typedef struct chainon_s chainon;
struct chainon_s
{
    int val;
    chainon *prec;
    chainon *suiv;
};

typedef struct liste_s liste;
struct liste_s
{
    chainon *premier;
    chainon *dernier;
};
```

En particulier, les lettres sont donc représentées par des entiers ; cela permettra de réutiliser cette structure `liste` dans d'autres contextes.

Dans l'implémentation de cette structure de données, on veillera à ce qu'un pointeur dont la valeur n'est pas `NULL` pointe toujours vers un espace mémoire valide et pas, par exemple, vers une donnée libérée.

Question 1. Définir et initialiser une variable globale `lg` représentant la liste chaînée globale manipulée par les programmes push-pop ; on souhaite que cette liste soit initialement vide.

Question 2. Programmer une fonction `est_vide` de prototype `bool est_vide(void)` renvoyant `true` si `lg` représente une liste vide, `false` sinon.

Question 3. Programmer des fonctions `pushL` et `pushR` de prototypes `void pushL(int)` et `void pushR(int)` implémentant les opérations `pushL` et `pushR` sur la liste représentée par `lg`. On utilisera une assertion pour vérifier que l'allocation dynamique de mémoire est bien réalisée.

Question 4. Programmer des fonctions `popL` et `popR` de prototypes `int popL(void)` et `int popR(void)` implémentant les opérations `popL` et `popR` sur la liste représentée par `lg` (chacune renvoyant également la valeur extraite de la liste). On utilisera une assertion pour s'assurer que la liste n'est pas vide et on veillera à libérer la mémoire devenue inutile.

Question 5. Programmer des fonctions `output` et `vide_liste` de prototypes respectifs `void output(void)` et `void vide_liste(void)` ; `output` affiche le mot codé par la liste de caractères sur la sortie standard (suivi d'un retour à la ligne), tandis que `vide_liste` vide la liste (retire tous ses éléments) et libère la mémoire dynamique associée.

Question 5.

On souhaite écrire une fonction qui compte, dans une liste d'entiers de type `liste`, l'ensemble des triplets d'éléments *distincts* dont la somme vaut zéro, c'est-à-dire le nombre de triplets (x, y, z) tels que $x + y + z = 0$.

Un étudiant propose l'implémentation suivante :

```
int three_sum_liste(liste *l) {
    int c;                      // compteur de triplets
    chainon *i, *j, *k;

    i = l->premier->prec;       // on commence "avant" le premier élément
    while (i != NULL) {
        int ai = i->val;

        j = i;                  // on repart du même maillon
        while (j != NULL) {
            int aj = j->val;

            k = j;              // idem ici
            while (k->suiv != NULL) {   // on s'arrête "après" le dernier
                if (ai + aj + k->val = 0) { // teste si la somme vaut 0
                    c = c + 1;
                    return c;   // on peut déjà renvoyer le résultat
                }
                /* TODO: avancer k ? */
            }

            /* on recommence à partir de i pour être sûr de ne rien rater */
            j = i;
        }

        /* cas particulier : si la liste est vide on renvoie 0 */
        if (l->premier == NULL) {
            return 0;
        }
        /* on avance i de deux maillons pour aller plus vite */
        i = i->suiv->suiv;
    }

    /* si on arrive ici c'est qu'on n'a trouvé aucun triplet */
    return c;
}
```

Réécrire la fonction en repérant et en corrigeant **toutes** les erreurs du programme.

En supposant que l'on définisse `type 'a ilist = Empty | Cons of 'a * 'a ilist`, écrire la même fonction `three_sum_liste` en OCaml.