```c
/* Type structuré définissant un maillon de liste simplement chaînée */
typedef struct s_list {
    int value;
    struct s_list* next;
} list_elem_t;

/* Prototypes */
int insert_head(list_elem_t** l, int value);
int insert_tail(list_elem_t** l, int value);
list_elem_t* find_element(list_elem_t* l, int index);
int remove_element(list_elem_t** l, int value);
void reverse_list(list_elem_t** l);
```

```c
/*****************************************************************************
 * L3 Informatique                                              Module SYR1 *
 *                          TP de programmation en C                         *
 *                      Mise en oeuvre des listes chaînées                   *
 *                                                                           *
 * Groupe : 2.1                                                              *
 * Nom Prénom 1 : Noël-Baron Léo                                             *
 * Nom Prénom 2 : Sampaio Thierry                                            *
 *                                                                           *
 *****************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "list.h"

int nb_malloc = 0; // compteur global du nombre d'allocations

/*
 * SYNOPSIS :
 *   list_elem_t* create_element(int value)
 * DESCRIPTION :
 *   crée un nouveau maillon de liste, dont le champ next a été initialisé à
 *   NULL, et dont le champ value contient l'entier passé en paramètre.
 * PARAMETRES :
 *   int value : valeur de l'élément
 * RESULTAT :
 *   NULL en cas d'échec, sinon un pointeur sur une structure list_elem_t
 */
static list_elem_t* create_element(int value) {
    list_elem_t* new_elt = malloc(sizeof(list_elem_t));
    if (new_elt != NULL) {
        ++nb_malloc;
        new_elt->value = value;
        new_elt->next = NULL;
    }
    return new_elt;
}

/*
 * SYNOPSIS :
 *   void free_element(list_elem_t* l)
 * DESCRIPTION :
 *   libère un maillon de liste.
 * PARAMETRES :
 *   list_elem_t* l : pointeur sur le maillon à libérer
 * RESULTAT :
 *   rien
 */
static void free_element(list_elem_t* l) {
    --nb_malloc;
    free(l);
}

/*
 * SYNOPSIS :
 *   int insert_head(list_elem_t** l, int value)
 * DESCRIPTION :
 *   ajoute un élément en tête de liste ; à l'issue de l'exécution de la
 *   fonction, *l désigne la nouvelle tête de liste.
 * PARAMETRES :
 *   list_elem_t** l : pointeur sur le pointeur de tête de liste
```



```c
 *   int value : valeur de l'élément à ajouter
 * RESULTAT :
 *    0 en cas de succès, -1 si l'ajout est impossible
 */
int insert_head(list_elem_t** l, int value) {
    if (l == NULL)
        return -1;
    list_elem_t* new_elt = create_element(value);
    if (new_elt == NULL)
        return -1;

    new_elt->next = *l;
    *l = new_elt;
    return 0;
}

/*
 * SYNOPSIS :
 *   int insert_tail(list_elem_t** l, int value)
 * DESCRIPTION :
 *   ajoute un élément en queue de la liste (*l désigne la tête de liste).
 * PARAMETRES :
 *   list_elem_t** l : pointeur sur le pointeur de tête de liste
 *   int value : valeur de l'élément à ajouter
 * RESULTAT :
 *    0 en cas de succès, -1 si l'ajout est impossible
 */
int insert_tail(list_elem_t** l, int value) {
    if (l == NULL)
        return -1;
    list_elem_t* new_elt = create_element(value);
    if (new_elt == NULL)
        return -1;

    if (*l == NULL) { // Si la liste est vide on redirige le pointeur de tête
        *l = new_elt;
        return 0;
    }
    list_elem_t* tail = *l;
    // Sinon on parcourt jusqu'à la queue, puis on branche le nouvel élément
    // au dernier
    while (tail->next != NULL)
        tail = tail->next;
    tail->next = new_elt;
    return 0;
}

/*
 * SYNOPSYS :
 *   list_elem_t* find_element(list_elem_t* l, int index)
 * DESCRIPTION :
 *   retourne un pointeur sur le maillon à la position n°i de la liste (le 1er
 *   élément est situé à la position 0)
 * PARAMETRES :
 *   int index : position de l'élément à retrouver
 *   list_elem_t* l : pointeur sur la tête de liste
 * RESULTAT :
 *   NULL en cas d'erreur, sinon un pointeur sur le maillon de la liste
 */
list_elem_t* find_element(list_elem_t* l, int index) {
    if (l == NULL)
        return NULL;
```

```
    list_elem_t* found = l;
    int i = 0;
    // On parcourt la liste en décrémentant un compteur jusqu'à arriver
    // à la case voulue
    for (i=index-1; i>0 && found->next!=NULL; i--)
        found = found->next;
    if (i != 0)
        return NULL;
    else
        return found;
}

/*
 * SYNOPSIS :
 *   int remove_element(list_elem_t** l, int value)
 * DESCRIPTION :
 *   supprime de la liste (dont la tête a été passée en paramètre) le premier
 *   élément de valeur value, et libère l'espace mémoire utilisé par le maillon
 *   ainsi supprimé. Attention, à l'issue de la fonction la tête de liste
 *   peut avoir été modifiée.
 * PARAMETRES :
 *   list_elem_t** l : pointeur sur le  pointeur de tête de liste
 *   int value : valeur à supprimer de la liste
 * RESULTAT :
 *    0 en cas de succès, -1 en cas d'erreur
 */
int remove_element(list_elem_t** l, int value) {
    if (l == NULL || *l == NULL)
        return -1;

    // On parcourt la liste jusqu'à l'élément à détruire, tout en maintenant
    // un pointeur vers l'élément précédent
    list_elem_t* pre = NULL;
    list_elem_t* del = *l;
    while (del->value != value && del->next != NULL) {
        pre = del;
        del = del->next;
    }
    if (del->value != value) // Si la valeur n'a pas été trouvée, erreur
        return -1;

    if (del == *l)
        *l = del->next;
    else
        pre->next = del->next;
    free_element(del);
    return 0;
}

/*
 * SYNOPSIS :
 *   void reverse_list(list_elem_t** l)
 * DESCRIPTION :
 *   modifie la liste en renversant l'ordre de ses élements (le 1er élément est
 *   placé en dernière position, le 2nd en avant-dernière, etc).
 * PARAMETRES :
 *   list_elem_t** l : pointeur sur le pointeur de tête de liste
 * RESULTAT :
 *   rien
 */
void reverse_list(list_elem_t** l) {
```



```
    if (l == NULL || *l == NULL)
        return;

    // On maintient trois pointeurs vers trois éléments successifs de la liste
    // pour pouvoir renverser les pointeurs en un seul parcours sans perdre
    // d'information
    list_elem_t* pre = NULL;
    list_elem_t* cur = *l;
    list_elem_t* next = NULL;
    while (cur != NULL) {
        next = cur->next;
        cur->next = pre;
        pre = cur;
        cur = next;
    }

    *l = pre;
}
```

```
/*****************************************************************************
 * L3 Informatique                                              Module SYR1 *
 *                          TP de programmation en C                         *
 *                          Test des listes chaînées                         *
 *                                                                           *
 * Groupe : 2.1                                                              *
 * Nom Prénom 1 : Noël-Baron Léo                                             *
 * Nom Prénom 2 : Sampaio Thierry                                            *
 *                                                                           *
 *****************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>
#include "list.h"

/* Compteur du nombre d'allocations */
extern int nb_malloc;

/* Compte le nombre d'éléments de la liste */
static int list_size(list_elem_t* p_list) {
    int nb = 0;
    while (p_list != NULL) {
        nb += 1;
        p_list = p_list->next;
    }
    return nb;
}

/* Affiche le contenu de la liste */
void print_list(list_elem_t* p_list) {
    list_elem_t* pl = p_list;
    printf("%d élément(s) : ", list_size(p_list));
    while(pl != NULL) {
        printf("[%d]", pl->value);
        pl = pl->next;
        if (pl != NULL)
            printf("->");
    }
}

int main(int argc, char* argv[]) {
    list_elem_t* la_liste = NULL; // Pointeur de tête de liste
    char menu[] =
        "Programme de test de liste\n"\
        " 't/q' : ajout d'un élément en tête/queue de liste\n"\
        " 'f'  : recherche du ième élément de la liste\n"\
        " 's'  : suppression d'un élément de la liste\n"\
        " 'r'  : renverser l'ordre des éléments de la liste\n"\
        " 'x'  : quitter le programme\n"\
        "> ";
    int choice = 0; // Choix dans le menu
    int value = 0; // Valeur saisie

    printf("%s", menu);
    fflush(stdout);

    while (1) {
        fflush(stdin);
        choice = getchar();
```



```
        switch (choice) {
        case 'T' :
        case 't' :
            printf("Valeur du nouvel element : ");
            scanf("%d",&value);
            if (insert_head(&la_liste,value)!=0)
                printf("Impossible d'ajouter la valeur %d\n", value);
            break;
        case 'Q' :
        case 'q' :
            printf("Valeur du nouvel element : ");
            scanf("%d",&value);
            if (insert_tail(&la_liste,value)!=0)
                printf("Impossible d'ajouter la valeur %d\n", value);
            break;
        case 'F' :
        case 'f' :
            printf("Index à rechercher : ");
            scanf("%d",&value);
            list_elem_t* found = find_element(la_liste, value);
            if (found != NULL)
                printf("Elément trouvé : [%d]\n", found->value);
            else
                printf("Elément %d introuvable\n", value);
            break;
        case 's' :
        case 'S' :
            printf("Elément à supprimer : ");
            scanf("%d",&value);
            if (remove_element(&la_liste,value)!=0)
                printf("Impossible de supprimer la valeur %d\n", value);
            break;
        case 'r' :
        case 'R' :
            reverse_list(&la_liste);
            break;
        case 'x' :
        case 'X' :
            return 0;
        default:
            break;
        }

        print_list(la_liste);
        if (nb_malloc!=list_size(la_liste)) {
            printf("\nAttention : il y a une fuite mémoire dans votre " \
                "programme !\nLa liste contient %d élement, or il y a %d " \
                "élément alloués en mémoire !", list_size(la_liste),
                nb_malloc);
        }
        getchar(); // Consomme un RC et évite un double affichage du menu
        printf("\n\n%s",menu);
    }

    return 0;
}
```