```c
/* Type structurÃ© dÃ©finissant un maillon de liste doublement chaÃ®nÃ© */
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);
int remove_element(list_elem_t** ppl, int value);
list_elem_t* get_tail(list_elem_t* l);
void reverse_list(list_elem_t** l);
list_elem_t* find_element(list_elem_t* l, int index);
int list_size(list_elem_t* l);
```

```
/*****************************************************************************
 * L3Informatique                                                  Module SYR1
 *                        TP de programmation en C :
 *                Mise en oeuvre de listes chaînées
 *
 * Groupe       : 2.1
 * Nom Prénom 1 : Labanca Hector
 * Nom Prénom 2 : Bardou Augustin
 *
 *****************************************************************************/

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


int nb_malloc=0;

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

/*
 * SYNOPSYS :
 *       list_elem_t* create_element(int value)
 * DESCRIPTION :
 *   crée un nouveau maillon de liste, dont le champ next a été initialisé à NUL
L, et
 *   dont le champ value contient l'entier passé en paramètre.
 * PARAMETRES :
 *   value : valeur de l'élément
 * RESULTAT :
 *   NULL en cas d'échec, sinon un pointeur sur une structure de type list_elem_
t
 */
list_elem_t* create_element(int value) {
  nb_malloc++;
  list_elem_t* newelt= malloc(sizeof(list_elem_t));
  if (newelt!=NULL){
    newelt->value=value;
    newelt->next=NULL;
  }
  return newelt;
}


/*
 * SYNOPSYS :
 *       int insert_head(list_elem_t** l, int value)
 * DESCRIPTION :
```



```
 *   Ajoute un élément en tête de liste, à l'issu de l'exécution de la fonc
 *l désigne la nouvelle 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_head(list_elem_t** l, int value) {
  list_elem_t* new_elt =  create_element(value);
  if ((l!=NULL) && (new_elt!=NULL)) {
    new_elt->next=*l;
    *l=new_elt;
    return 0;
  } else {
    return -1;
  }
}


/*
 * SYNOPSYS :
 *       int list_size(list_elem_t* l)
 * DESCRIPTION :
 *   retourne le nombre d'éléments dans la liste
 * PARAMETRES :
 *       list_elem_t* l : pointeur sur la tête de liste
 * RESULTAT :
 *        nombre de maillon dans la liste
 */
int list_size(list_elem_t* l) {
  int size =0;
  while (l!=NULL) {
    size++;
    l=l->next;
  }
  return size;
}

/*
 * SYNOPSYS :
 *       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) {
        list_elem_t* p = NULL;
        if(l != NULL)
        {
                p = *l;
                while(p->next != NULL)
                {
                        p = p->next;
                }
                list_elem_t* nvElt = create_element(value);
```

```
                if(nvElt != NULL)
                {
                        p->next = nvElt;
                }
                else    return -1;
        }
        else    return -1;

        return 0;
}

/*
 * SYNOPSYS :
 *       int remove_element(list_elem_t** ppl, int value)
 * DESCRIPTION :
 *   Supprime de la liste (dont la tête a été passée en paramètre) l'element don
t
 *   la valeur a été passée en paramètre, et libère l'espace mémoire utilisé par
 le maillon
 *   ainsi supprimé. Attention, on suppose que value n'apparaît qu'un seule fois
 dans
 *   la liste, et à l'issu de la fonction la tête de liste peut avoir été modifi
ée.
 * PARAMETRES :
 *      list_elem_t** ppl : pointeur sur le  pointeur de tête de liste
 *             int value  : valeur à supprimer de la liste
 * RESULTAT :
 *     0 en cas de succès.
 *    -1 si erreur
 */
int remove_element(list_elem_t** ppl, int value) {
        list_elem_t* p = NULL;
        list_elem_t* prec = NULL;
        if(ppl != NULL)
        {
                p = *ppl;
                do
                {
                        if(p->value == value)
                        {
                                if(p == *ppl)
                                        *ppl = (*ppl)->next;
                                else
                                        prec->next = p->next;
                                free_element(p);
                                return 0;
                        }
                        prec = p;
                        p = p->next;
                } while(p != NULL);
        }
        return -1;
}


/*
 * 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 :
 *    - un pointeur sur le maillon de la liste en cas de succès
 *    - NULL si erreur
 */
list_elem_t* find_element(list_elem_t* l, int index) {
  list_elem_t* p = NULL;
        if(l != NULL)
        {
                p = l;

                while((p->next != NULL)&&(index != 1))
                {
                        p = p->next;
                        index--;
                }
                if(index == 1)
                {
                        return p;
                }
        }
        return NULL;
}



/*
 * SYNOPSYS :
 *       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 :
 *   aucun
 */
void reverse_list(list_elem_t** l) {
        list_elem_t* p = NULL;
        list_elem_t* newList = NULL;
        if(l != NULL)
        {
                p = *l;
                while(p != NULL)
                {
                        insert_head(&newList,p->value);
                        list_elem_t* temp = p;
                        p = p->next;
                        free_element(temp);
                }
                *l = newList;
        }
}
```

```
/*****************************************************************************
 * L3Informatique                                                 Module SYR1
 *                      TP de programmation en C :
 *                       Test de listes chaînées
 *
 * Groupe       : 2.1
 * Nom Prenom 1 : Labanca Hector
 * Nom Prenom 2 : Bardou Augustin
 *
 *****************************************************************************/


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

list_elem_t* la_liste = NULL;

void print_list(list_elem_t* p_list) {
  list_elem_t *pl = p_list;
  printf("La liste contient %d element(s)\n",list_size(p_list));
  if (pl!=NULL) {
    while(pl!=NULL) {
      printf("[%d]",pl->value);
      pl=pl->next;
      if (pl!=NULL) {
        printf("->");
      }
    }
  }
}

extern int nb_malloc;

char menu[] =
  "\n Programme de test de listes\n"\
  " 't/q' : ajout d'un element en tete/queue de liste\n"\
  " 'f'  : recherche du ieme element de la liste\n"\
  " 's'  : suppression d'un element de la liste\n"\
  " 'r'  : renverser l'ordre des elements de la liste\n"\
  " 'x'  : quitter le programme\n"\
  " Quel est votre choix ?";

int main(int argc, char **argv) {
  int choix=0;
  int value=0;

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

  while(1) {
    fflush(stdin);
    choix = getchar();
    printf("\n");

    switch(choix) {
    case 'T' :
    case 't' :
      printf("Valeur du nouvel element ? ");
```



```
      scanf("%d",&value);
      if (insert_head(&la_liste,value)!=0) {
        printf("Erreur : impossible d'ajouter l'element %d\n",value);
      };
      break;

    case 'Q' :
    case 'q' :
      printf("Valeur du nouvel element ? ");
      scanf("%d",&value);
      if (insert_tail(&la_liste,value)!=0) {
              printf("Erreur : impossible d'ajouter l'element %d\n",value);
      };
      break;


    case 'F' :
    case 'f' :
              printf("Index de l'element a rechercher ? ");
              scanf("%d", &value);
              list_elem_t* result = find_element(la_liste, value);
              if(result != NULL)
                      printf("La valeur de l'element numero %d est %d\n", value, resu
value);
              else
                      printf("Erreur, l'element numero %d n'existe pas\n", value);
      break;



    case 's' :
    case 'S' :
      printf("Valeur de l'element a supprimer ? ");
          scanf("%d", &value);
          if(remove_element(&la_liste, value) != 0)
              printf("Erreur, aucun element ne possede la valeur %d\n", value);
          else
              printf("L'element de valeur %d a ete supprime avec succes\n", value);

      break;


    case 'r' :
    case 'R' :
      printf("Renversement des elements de la liste.\n");
          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 memoire dans votre programme !\n");
        printf("la liste contient %d element, or il y a %d element vivant en memoire !\n",list_size
liste),nb_malloc);
```

```
    }
    getchar(); // pour consommer un RC et eviter un double affichage du menu !
    printf("%s\n",menu);
  }
  return 0;
}
```