Listes d'entiers

Licence Informatique : Programmation Nécessaire

Stéfane

Table des Matières

1 Première partie

Commencez par récupérer les fichiers sur *Arche* et complétez les

Le TA `list_elm_t` et ses fonctions.

```c
/**
 * @brief Le type abstrait d'un _élément_ de liste qui contient:
 *  @param x : un entier,
 *  @param suc : un pointeur sur son successeur (ou NULL s'il n'y en a pas)
 */
typedef struct list_elm {
  int x;
  struct list_elm * suc;
} list_elm_t;

list_elm_t * new_list_elm ( int value ); /// @brief Créer un élément de liste et y ranger value
void del_list_elm ( list_elm_t ** ptrE ); /// @brief Supprimer un élément de liste

int getX ( list_elm_t * E ); /// @brief Renvoyer la valeur entière de E
void setX ( list_elm_t * E, int v ); /// @brief Modifier la valeur entière de E

list_elm_t * getSuc ( list_elm_t * E ); /// @brief Renvoyer le successeur de E
void setSuc ( list_elm_t * E, list_elm_t * S ); /// @brief Modifier le successeur de E
```

Le TA `list_t` et ses fonctions.

```c
/** @brief Le type abstrait d'une _liste_ :
 * @param head : le premier élément de la liste
 * @param tail : le dernier élément de la liste
 * @param numelm : le nombre d'élément dans la liste
 */
typedef struct list {
  struct list_elm * head;
  struct list_elm * tail;
  int numelm;
} list_t;

bool empty_list ( const list_t * L ); // @brief Vérifier si la liste L est vide ou pas

list_t * new_list (); // @brief Construire une liste vide
void del_list ( list_t ** ptrL ); /// @brief Libèrer la mémoire occupée par la liste

void cons ( list_t * L, int v ); // @brief Ajouter en tête de L la valeur v
void queue ( list_t * L, int v ); // @brief Ajouter en queue de L la valeur v

void view_list ( list_t * L ); // @brief Visualiser les éléments de la liste L
```

Les définitions des fonctions de `list_elm_t`.

```c
list_elm_t * new_list_elm ( int value ) { /** @todo */ }
void del_list_elm ( list_elm_t ** ptrE ) { /** @todo */ }

int getX ( list_elm_t * E ) { /** @todo */ }
void setX ( list_elm_t * E, int v ) { /** @todo */ }

list_elm_t * getSuc ( list_elm_t * E ) { /** @todo */ }
void setSuc ( list_elm_t * E, list_elm_t * S ) { /** @todo */ }
```

Les définitions des fonctions de `list_t`.

```c
list_t * new_list() {
  /** @note : calloc fonctionne de manière identique à malloc
    et de surcroît met à NULL(0) tous les octets alloués */
  list_t * L = calloc ( 1, sizeof(list_t) );
  assert(L);
  return L;
}
void del_list ( list_t ** ptrL ) { /** @todo */ }

bool empty_list ( const list_t * L ) {
  assert(L);
  return L->numelm == 0;
}
void view_list ( list_t * L ) {
  printf( "[ " );
  for( list_elm_t * E = L->head; E; E = E->suc ) {
    printf( "%d ", E->x );
  }
  printf( "]\n\n" );
}

void cons ( list_t * L, int v ) { /** @todo */ }
void queue ( list_t * L, int v ) { /** @todo */ }
```

Vous allez compléter les définitions des fonctions et vérifier leur bon fonctionnement grâce à la fonction principale `main` :

La fonction principale.

```c
/** ALGORITHME (FONCTION PRINCIPALE) **/
int main () {
  int v;
  list_t * L = new_list ();
  scanf ( "%d", &v );
  while ( v > 0 ) {
    cons ( L, v );
    // queue ( L, v )
    // insert_ortdered ( L, v )
    scanf ( "%d", &v );
  }
  view_list ( L );
  del_list ( &L );
  return 0;
}
```

Compilez grâce à la commande *`make`* et au fichier *`makefile`* :

Le makefile.

```
BDIR = bin
IDIR = include
ODIR = obj
SDIR = src

CC = gcc
CFLAGS = -Wall -Wextra -std=gnu17 -I$(IDIR)
LFLAGS = -lm

PROG = $(BDIR)/form

_DEPS = elmlist.h list.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJS = elmlist.o list.o main.o
OBJS = $(patsubst %,$(ODIR)/%,$(_OBJS))

.PHONY : run all dirs clean delete

run : all
  ./$(PROG)

all : dirs $(OBJS)
  $(CC) -o $(PROG) $(OBJS) $(LFLAGS)

dirs :
  @mkdir -p $(ODIR)
  @mkdir -p $(BDIR)

$(ODIR)/%.o : $(SDIR)/%.c $(DEPS)
  $(CC) $(CFLAGS) -c -o $@ $<

clean :
  rm -rf $(ODIR) core

delete : clean
  rm -rf $(BDIR)
```

2 Seconde partie

Complétez ces 2 TA avec les fonctions `insert_after` et `insert_ordered`.

2.1 La fonction `insert_after`

La fonction prend trois arguments:

```
insert_after(struct list_t * L, int value, struct list_elm_t * place)
```

La liste `L` est modifiée par l'insertion de la valeur entière `value` **après** l'emplacement désigné par `place`.

Hypothèse)
Par convention:

- Soit `place` est `NULL` et l'insertion se fait alors en tête de la liste `L`: `cons(L,v)`

- Soit `place` désigne le dernier élément, et l'on fait appel à `queue(L,v)`
- Soit `place` désigne **obligatoirement** un élément de la liste `L`.
  - Il faut alors créer un nouvel élément de liste
  - Y ranger la valeur entière
  - Indiquer que le successeur est celui de `place`
  - Puis déplacer le successeur de `place` sur ce nouvel élément.

Définition ***(Fonction privée)***

Cette fonction est privée c-à-d. qu'elle n'est accessible que par les fonctions publiques du `TA list_t`..

Pour cette raison sa déclaration est faite en tête du fichier `list.c` (et non dans le fichier `list.h`).

Sa définition est écrite après les définitions de toutes les fonctions publiques.

2.2 La fonction `insert_ordered`

La fonction `insert_ordered(struct list_t * L, const int value)` insère dans l'ordre *croissant* la valeur entière `value` dans la liste `L`. Vous rencontrerez plusieurs situations:

Soit la liste `L` est vide.

L'insertion revient à faire une insertion en tête: `cons(L,v)`

Soit `v` est plus petite que la première valeur entière de `L`.

L'insertion se fait également en tête: `cons(L,v)`

Soit `value` est plus grande que la dernière valeur de `L`.

L'insertion se fait alors en queue: `queue(L,v)`

Dans les autres cas.

Sachant que la liste contient au moins 2 éléments, la fonction `insert_ordered` recherche la **place après** laquelle `value` doit être insérée.

Une fois la place trouvée, la fonction appelle `insert_after` avec les bons paramètres.