PRG2 – Labo 2

Listes Dynamiques

Groupe M

HARUN Ouweis, MOUTI Amir, SERZEDELO COSTA André Miguel

main.c

```c
/*
 ------------------------------------------------------------------------------
 Nom du fichier : main.c
 Auteur(s)      : Andre Costa, Amir Mouti, Ouweis Harun
 Date creation  : 27.04.2023

 Description    : Ce fichier teste l'implementation de la librairie pour
                  des listes doublement chaînées non circulaires.

 Remarque(s)    : L'utilisation d'assertions est extensive dans ces tests donc ce
                  programme doit être compilé en mode DEBUG.

 Compilateur    : Mingw-w64 gcc 12.2.0
 ------------------------------------------------------------------------------
*/
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include "listes_dynamiques.h"

/**
 * Fonction qui affiche le test qui a été passé
 * @param test nom du test
 */
void afficherTestOk(const char *test);

/**
 * Fonction qui teste la fonction initialiser de notre liste dynamique
 */
void testInitialiser(void);

/**
 * Fonction qui teste les fonctions d'insertion et suppression en tête de notre
 * liste dynamique
 */
void testInsererEtSupprimerEnTete(void);

/**
 * Fonction qui teste la fonction estVide de notre liste dynamique
 */
void testEstVide(void);

/**
 * Fonction qui teste la fonction longueuer de notre liste dynamique
 */
void testLongueur(void);

/**
 * Fonction qui teste la fonction afficher de notre liste dynamique
 */
void testAfficher(void);

/**
 * Fonction qui teste les fonctions d'insertion et suppression en queue de notre
 * liste dynamique
 */
void testInsererEtSupprimerEnQueue(void);

/**
 * Fonction qui teste la fonction vider de notre liste dynamique
 */
void testVider(void);
```

```c
/**
 * Fonction qui teste la fonction supprimerSelonCritere de notre liste dynamique
 */
void testSupprimerSelonCritere(void);

/**
 * Fonction qui teste la fonction sontEgales de notre liste dynamique
 */
void testSontEgales(void);

/**
 * Fonction pour tester la suppression selon critere de notre liste dynamique
 * @param pos position dans la liste
 * @param val valeur de l'element dans la position pos de la liste
 * @return true si la valeur n'est pas comprise entre 3 et 7, false sinon
 */
bool estEntre3et7(size_t pos, const Info *val);

/**
 * Fonction pour tester la suppression selon critere de notre liste dynamique
 * @param pos position dans la liste
 * @param val valeur de l'element dans la position pos de la liste
 * @return true si la position est paire, false sinon
 */
bool positionEstPaire(size_t pos, const Info *val);

/**
 * Fonction pour tester la suppression selon critere de notre liste dynamique
 *
 * @param pos position dans la liste
 * @param val valeur de l'element dans la position pos de la liste
 * @return true pour supprimer tous les éléments.
 */
bool suppressionComplete(size_t pos, const Info *val);

int main(void) {
   testInitialiser();
   testInsererEtSupprimerEnTete();
   testEstVide();
   testLongueur();
   testAfficher();
   testInsererEtSupprimerEnQueue();
   testVider();
   testSupprimerSelonCritere();
   testSontEgales();
}

void afficherTestOk(const char *test) {
   printf("%s: %s\n", test, "OK");
}

void testInitialiser(void) {
   Liste *liste = initialiser();
   assert(liste != NULL);
   assert(liste->tete == NULL);
   assert(liste->queue == NULL);
   free(liste);
   afficherTestOk("Test initialiser()");
}
```

```
void testInsererEtSupprimerEnTete(void) {
   Liste *liste = initialiser();
   assert(liste != NULL);
   // Test 1 : liste vide, insertion en tête
   Info info1 = 42;
   assert(insererEnTete(liste, &info1) == OK);
   assert(liste->tete != NULL);
   assert(liste->tete->info == info1);
   assert(liste->queue == liste->tete);
   afficherTestOk("Test insererEnTete(), liste vide");

   Info infoObtenue1;
   assert(supprimerEnTete(liste, &infoObtenue1) == OK);
   assert(infoObtenue1 == info1);
   assert(liste->tete == NULL);
   assert(liste->queue == liste->tete);
   afficherTestOk("Test supprimerEnTete(), 1 element");

   // Test 2 : liste non vide, insertion en tête
   Info info2 = 84;
   assert(insererEnTete(liste, &info1) == OK);
   assert(insererEnTete(liste, &info2) == OK);
   // pas besoin de verifier liste->tete, ça a deja ete teste
   assert(liste->tete->suivant != NULL);
   assert(liste->tete->suivant->info == info1);
   assert(liste->queue == liste->tete->suivant);
   assert(liste->queue->precedent != NULL);
   assert(liste->queue->precedent == liste->tete);
   afficherTestOk("Test insererEnTete(), liste non vide");

   Info infoObtenue2;
   assert(supprimerEnTete(liste, &infoObtenue2) == OK);
   assert(infoObtenue2 == info2);
   assert(liste->tete != NULL);
   assert(liste->tete->info == info1);
   assert(liste->tete->suivant == NULL);
   assert(liste->queue == liste->tete);

   assert(supprimerEnTete(liste, &infoObtenue1) == OK);
   assert(infoObtenue1 == info1);
   assert(liste->tete == NULL);
   assert(liste->queue == liste->tete);
   afficherTestOk("Test supprimerEnTete(), 2 elements");

   //test 3. Insertion avec ptr null
   assert(insererEnTete(liste, NULL) == OK);
   assert(liste->tete->info == 0);
   afficherTestOk("Test insererEnTete(), pointeur NULL");
   assert(supprimerEnTete(liste, NULL) ==OK);

   //test 4 suppression liste vide
   assert(supprimerEnTete(liste, NULL) == LISTE_VIDE);
   afficherTestOk("Test supprimerEnTete(), liste vide");
   free(liste);
}
```

```
void testEstVide(void) {

   Liste *liste = initialiser();
   assert(liste != NULL);
   assert(estVide(liste));

   Info info = 42;
   assert(insererEnTete(liste, &info) ==OK);
   assert(!estVide(liste));

   supprimerEnTete(liste, NULL);
   assert(estVide(liste));

   afficherTestOk("Test estVide()");
   free(liste);
}

void testLongueur(void) {
   Liste *liste = initialiser();
   assert(liste != NULL);
   assert(longueur(liste) == 0);
   Info info1 = 1, info2 = 2, info3 = 3;
   assert(insererEnTete(liste, &info1) == OK);
   assert(insererEnTete(liste, &info2) == OK);
   assert(insererEnTete(liste, &info3) == OK);
   assert(longueur(liste) == 3);
   supprimerEnTete(liste, &info3);
   assert(longueur(liste) == 2);
   supprimerEnTete(liste, &info3);
   assert(longueur(liste) == 1);
   supprimerEnTete(liste, &info3);
   assert(longueur(liste) == 0);
   free(liste);
   afficherTestOk("Test longueur()");
}

void testAfficher(void) {
   Liste *liste = initialiser();
   assert(liste != NULL);
   printf("Liste vide : ");
   afficher(liste, FORWARD);
   printf("\n");
   printf("Liste vide : ");
   afficher(liste, BACKWARD);
   printf("\n");
   Info info1 = 1, info2 = 2, info3 = 3;
   assert(insererEnTete(liste, &info1) == OK);
   assert(insererEnTete(liste, &info2) == OK);
   assert(insererEnTete(liste, &info3) == OK);
   printf("Liste Forward: ");
   afficher(liste, FORWARD);
   printf("\n");
   printf("Liste Backward: ");
   afficher(liste, BACKWARD);
   printf("\n");
   afficherTestOk("Test afficher()");

   while (!estVide(liste)) {
      supprimerEnTete(liste, NULL);
   }
   free(liste);

}
```

```
void testInsererEtSupprimerEnQueue(void) {
   Liste *liste = initialiser();
   assert(liste != NULL);
   // Test 1 : liste vide, insertion en queue
   Info info1 = 42;
   assert(insererEnQueue(liste, &info1) == OK);
   assert(liste->tete != NULL);
   assert(liste->tete->info == info1);
   assert(liste->queue == liste->tete);
   afficherTestOk("Test insererEnQueue(), liste vide");

   Info infoObtenue1;
   assert(supprimerEnQueue(liste, &infoObtenue1) == OK);
   assert(infoObtenue1 == info1);
   assert(liste->tete == NULL);
   assert(liste->queue == liste->tete);
   afficherTestOk("Test supprimerEnQueue(), 1 element");

   // Test 2 : liste non vide, insertion en queue
   Info info2 = 84;
   assert(insererEnQueue(liste, &info1) == OK);
   assert(insererEnQueue(liste, &info2) == OK);
   assert(liste->tete->suivant != NULL);
   assert(liste->tete->info == info1);
   assert(liste->tete->suivant->info == info2);
   assert(liste->queue == liste->tete->suivant);
   assert(liste->queue->precedent != NULL);
   assert(liste->queue->precedent == liste->tete);
   afficherTestOk("Test insererEnQueue(), liste non vide");

   Info infoObtenue2;
   assert(supprimerEnQueue(liste, &infoObtenue2) == OK);
   assert(infoObtenue2 == info2);
   assert(liste->tete != NULL);
   assert(liste->tete->info == info1);
   assert(liste->tete->suivant == NULL);
   assert(liste->queue == liste->tete);

   assert(supprimerEnQueue(liste, &infoObtenue1) ==OK);
   assert(infoObtenue1 == info1);
   assert(liste->tete == NULL);
   assert(liste->queue == liste->tete);
   afficherTestOk("Test supprimerEnQueue(), 2 elements");

   //test 3. Insertion avec ptr null
   assert(insererEnQueue(liste, NULL) ==OK);
   assert(liste->tete->info == 0);
   afficherTestOk("Test insererEnQueue(), pointeur NULL");
   assert(supprimerEnQueue(liste, NULL) ==OK);
   assert(liste->tete == NULL);

   //test 4. suppression liste vide
   assert(supprimerEnQueue(liste, NULL) == LISTE_VIDE);
   afficherTestOk("Test supprimerEnQueue(), liste vide");
   free(liste);
}
```

```
void testVider(void) {
   const size_t LONGUEUR = 20;
   Liste *liste = initialiser();
   assert(liste != NULL);
   for (size_t i = 0; i < LONGUEUR; i++) {
      Info aux = (Info) i;
      insererEnQueue(liste, &aux);
   }
   // Test 1 vider moitié
   vider(liste, LONGUEUR / 2lu);
   assert(longueur(liste) == LONGUEUR / 2lu);
   assert(liste->queue->info == 9);
   assert(liste->tete->info == 0);
   afficherTestOk("Test vider(), moitie");

   // Test 2 vider a partir de position non valable
   vider(liste, LONGUEUR / 2);
   assert(longueur(liste) == 10);
   assert(liste->queue->info == 9);
   assert(liste->tete->info == 0);
   afficherTestOk("Test vider(), position non valable");

   // Test 3 vider tout
   vider(liste, 0);
   assert(estVide(liste));
   assert(longueur(liste) == 0);
   assert(liste->tete == NULL);
   assert(liste->queue == NULL);
   afficherTestOk("Test vider(), tout");

   free(liste);
}

void testSupprimerSelonCritere(void) {
   Liste *liste = initialiser();
   assert(liste != NULL);
   for (int i = 0; i < 10; i++) {
      insererEnQueue(liste, &i);
   }
   supprimerSelonCritere(liste, estEntre3et7);
   assert(liste->tete->info == 3);
   assert(liste->queue->info == 7);
   assert(longueur(liste) == 5);
   afficherTestOk("Test supprimer selon critere (info)");
   vider(liste, 0);

   for (int i = 0; i < 10; i++) {
      insererEnQueue(liste, &i);
   }
   supprimerSelonCritere(liste, positionEstPaire);
   assert(liste->tete->info == 1);
   assert(liste->queue->info == 9);
   assert(longueur(liste) == 5);
   afficherTestOk("Test supprimer selon critere (position)");

   vider(liste, 0);
   for (int i = 0; i < 10; i++) {
      insererEnQueue(liste, &i);
   }
   supprimerSelonCritere(liste, suppressionComplete);
   assert(liste->tete == NULL);
   assert(liste->queue == NULL);
   assert(estVide(liste));
   assert(longueur(liste) == 0);
   afficherTestOk("Test supprimer selon critere (tout)");

   free(liste);
}
```

```
void testSontEgales(void) {
   Liste *l1 = initialiser();
   Liste *l2 = initialiser();
   assert(l1 != NULL);
   assert(l2 != NULL);
   assert(sontEgales(l1, l2));
   afficherTestOk("Test listes vides egales");

   Info info1 = 1, info2 = 2, info3 = 3;
   assert(insererEnQueue(l1, &info1) == OK);
   assert(insererEnQueue(l1, &info2) == OK);
   assert(insererEnQueue(l1, &info3) == OK);
   assert(insererEnQueue(l2, &info1) == OK);
   assert(insererEnQueue(l2, &info2) == OK);
   assert(insererEnQueue(l2, &info3) == OK);

   assert(sontEgales(l1, l2));
   afficherTestOk("Test listes egales, 3 elements chaque");

   assert(insererEnQueue(l1, &info1) == OK);
   assert(insererEnQueue(l1, &info2) == OK);
   assert(insererEnQueue(l1, &info3) == OK);
   assert(!sontEgales(l1, l2));
   afficherTestOk("Test listes differentes, 3 premiers elements "
                  "correspondent");

   vider(l1, 0);
   vider(l2, 0);
   assert(insererEnQueue(l1, &info1) == OK);
   assert(insererEnQueue(l1, &info2) == OK);
   assert(insererEnQueue(l1, &info3) == OK);
   assert(insererEnQueue(l2, &info3) == OK);
   assert(insererEnQueue(l2, &info2) == OK);
   assert(insererEnQueue(l2, &info1) == OK);
   assert(!sontEgales(l1, l2));
   afficherTestOk("Test listes differentes, meme nombre d'elements");

   vider(l1, 0);
   vider(l2, 0);
   free(l1);
   free(l2);
}

bool estEntre3et7(size_t pos, const Info *val) {
   (void) pos; //eviter le warning de parametre non utilise
   return *val < 3 || *val > 7;
}

bool positionEstPaire(size_t pos, const Info *val) {
   (void) val; //eviter le warning de parametre non utilise
   return pos % 2 == 0;
}

bool suppressionComplete(size_t pos, const Info *val) {
   (void) pos; //eviter le warning de parametre non utilise
   (void) val;
   return true;
}
```

Listes_dynamiques.c

```c
/*
 ------------------------------------------------------------------------------
 Nom du fichier : listes_dynamiques.c
 Auteur(s)      : Andre Costa, Amir Mouti, Ouweis Harun
 Date creation  : 27.04.2023

 Description    : Ce fichier implémente une librairie pour des listes doublement
                  chaînées non circulaires.

 Remarque(s)    : Aucune vérification sur le pointeur 'liste' passé en paramètre
                  aux fonctions n'est effectuée

 Compilateur    : Mingw-w64 gcc 12.2.0
 ------------------------------------------------------------------------------
*/

#include "listes_dynamiques.h"
#include <stdlib.h>
#include <stdio.h>

const char AFFICHAGE_LISTE_DEBUT = '[';
const char AFFICHAGE_LISTE_FIN = ']';
const char AFFICHAGE_LISTE_ENTRE_ELEMENTS = ',';

/**
 * Macro pour creer des fonctions d'affichage selon le premier element et le
 * prochain element
 * @param SUFFIX suffix du nom de la fonction (afficher#SUFFIX)
 * @param PREMIER_ELEMENT comment atteindre le premier element de la liste (tete
 * ou queue)
 * @param PROCHAIN comment atteindre le prochain element de la liste selon le mode
 * (prochain ou suivant)
 */
#define CREER_FONCTION_AFFICHER(SUFFIX, PREMIER_ELEMENT, PROCHAIN)\
void afficher##SUFFIX (const Liste* liste)                        \
{                                                                 \
   printf("%c", AFFICHAGE_LISTE_DEBUT);                           \
   if(!estVide(liste))                                            \
   {                                                              \
      Element* elementActuel = liste->PREMIER_ELEMENT;            \
      printf("%d", elementActuel->info);                          \
      while((elementActuel = elementActuel->PROCHAIN))            \
      {                                                           \
         printf("%c", AFFICHAGE_LISTE_ENTRE_ELEMENTS);            \
         printf("%d", elementActuel->info);                       \
      }                                                           \
   }                                                              \
   printf("%c", AFFICHAGE_LISTE_FIN);                             \
}

CREER_FONCTION_AFFICHER(EnAvant, tete, suivant)

CREER_FONCTION_AFFICHER(EnArriere, queue, precedent)

/**
 *
 * @param element pointeur vers l'element a supprimer
 * @param info pointeur pour retourner l'information qui se trouve dans l'element
 * a supprimer
 */
static void supprimerElement(Element *element, Info *info);
```

```c
/**
 *
 * @param liste liste sur laquelle on veut trouver l'element
 * @param position position de la liste de l'element souhaité
 * @param element paramètre d'entrée/sortie pour retourner l'element à position
 * position. Si NULL est passé en paramètre, ce pointeur n'est pas modifié.
 * @return OK si on a trouvé l'element. POSITION_NON_VALIDE si la position
 * position n'est pas valide.
 */
static Status getElement(const Liste *liste, size_t position, Element **element);

Liste *initialiser(void) {
   // on utilise calloc pour mettre directement les éléments à 0 = NULL
   return (Liste *) calloc(1, sizeof(Liste));
}

bool estVide(const Liste *liste) {
   return liste->tete == NULL;
}

size_t longueur(const Liste *liste) {
   Element *elementActuel = liste->tete;
   size_t taille = 0;

   // tant qu'on un element dans la liste, on incrémente la taille
   while (elementActuel) {
      elementActuel = elementActuel->suivant;
      taille++;
   }
   return taille;
}

void afficher(const Liste *liste, Mode mode) {

   switch (mode) {
      case FORWARD:
         afficherEnAvant(liste);
         break;
      case BACKWARD:
         afficherEnArriere(liste);
         break;
      default:
         //on ne devrait jamais arriver ici.
         break;
   }
}
```

```
Status insererEnTete(Liste *liste, const Info *info) {
   // On utilise calloc pour avoir tous les elements de Element à 0
   // Pas besoin d'explicitement indiquer la valeur de precedent = NULL
   Element *element = (Element *) calloc(1, sizeof(Element));
   if (element) {

      if (info) {
         element->info = *info;
      }

      // Notre nouveau element doit pointer (avec suivant) sur la tete actuelle de
      // la liste
      element->suivant = liste->tete;

      if (liste->tete) {
         // La tete actuelle a maintenant un precedent (ce nouveau element)
         liste->tete->precedent = element;
      }
      // Ce nouveau element devient la nouvelle tete de la liste
      liste->tete = element;

      //si on a pas encore une queue, il devient aussi la nouvelle queue de la liste
      if (!liste->queue) {
         liste->queue = element;
      }
      return OK;
   }
   return MEMOIRE_INSUFFISANTE;
}

Status insererEnQueue(Liste *liste, const Info *info) {
   // On utilise calloc pour avoir tous les elements de Element à 0
   // Pas besoin d'explicitement indiquer la valeur de suivant = NULL
   Element *element = (Element *) calloc(1, sizeof(Element));

   if (element) {
      if (info) {
         element->info = *info;
      }

      // Notre nouvel element doit pointer (avec precedent) sur la queue actuelle
      // de la liste
      element->precedent = liste->queue;

      if (liste->queue) {
         // La queue actuelle a maintenant un suivant (ce nouveau element)
         liste->queue->suivant = element;
      }
      // Ce nouveau element devient la nouvelle queue de la liste
      liste->queue = element;

      //si on a pas encore une tete, il devient aussi la nouvelle tete de la liste
      if (!liste->tete) {
         liste->tete = element;
      }
      return OK;
   }
   return MEMOIRE_INSUFFISANTE;
}
```

```
void supprimerElement(Element *element, Info *info) {

   if (!element) {
      return;
   }

   //stocker l'information de l'élément actuel dans notre paramètre d'entrée/sortie
   if (info) {
      *info = element->info;
   }

   //Mettre à jour les pointeurs des elements voisins. En vérifiant d'abord s'ils
   // existent
   if (element->precedent) {
      element->precedent->suivant = element->suivant;
   }
   if (element->suivant) {
      element->suivant->precedent = element->precedent;
   }

   // restituer mémoire
   free(element);
}

Status supprimerEnTete(Liste *liste, Info *info) {
   // verifier si la liste est vide avant de faire quoi que ce soit
   if (estVide(liste)) {
      return LISTE_VIDE;
   }

   // garder un pointeur sur la tete actuelle pour pouvoir restituer la mémoire
   Element *tete = liste->tete;

   // nouvelle tete est le deuxieme element (s'il existe sinon NULL)
   liste->tete = tete->suivant;

   //si la liste devient vide, il faut aussi mettre à jour la queue
   if (!liste->tete) {
      liste->queue = NULL;
   }

   // supprimer l'ancienne tete
   supprimerElement(tete, info);

   return OK;
}

Status supprimerEnQueue(Liste *liste, Info *info) {
   // verifier si la liste est vide avant de faire quoi que ce soit
   if (estVide(liste)) {
      return LISTE_VIDE;
   }
   // garder un pointeur sur la queue actuelle pour pouvoir restituer la mémoire
   Element *queue = liste->queue;

   // nouvelle queue est l'avant dernier element (s'il existe sinon NULL)
   liste->queue = queue->precedent;

   //si la liste devient vide, il faut aussi mettre à jour la tete
   if (!liste->queue) {
      liste->tete = NULL;
   }
   // supprimer l'ancienne queue
   supprimerElement(queue, info);

   return OK;
}
```

```
void supprimerSelonCritere(Liste *liste,
                           bool (*critere)(size_t position, const Info *info)) {
   // verifier les parametres d'entree
   if (!critere || estVide(liste)) {
      return;
   }
   size_t position = 0;
   Element *elementActuel = liste->tete;

   //iterer sur la liste et verifier le critere pour chacun des elements
   while (elementActuel) {
      if (critere(position, &elementActuel->info)) {
         //critere ok, on supprime l'element
         // verifier si c'est la tete ou la queue
         if (elementActuel == liste->tete) {
            supprimerEnTete(liste, NULL);
            elementActuel = liste->tete;
         } else if (elementActuel == liste->queue) {
            supprimerEnQueue(liste, NULL);
            elementActuel = NULL;
         } else {
            // pas besoin de verifier si suivant existe car on serait en train de
            // supprimer la queue si c'etait le cas
            elementActuel = elementActuel->suivant;
            supprimerElement(elementActuel->precedent, NULL);
         }
      } else {
         // on passe au prochain element
         elementActuel = elementActuel->suivant;
      }
      position++;
   }
}

void vider(Liste *liste, size_t position) {
   Element *elementActuel;
   Status status = getElement(liste, position, &elementActuel);
   if (status == POSITION_NON_VALIDE) {
      return;
   }

   //Pour vider la liste, on fait la queue pointer sur l'element precedent
   // a l'element a la position position et on libere ensuite la memoire elements.
   liste->queue = elementActuel->precedent;


   if (elementActuel == liste->tete) {
      liste->tete = NULL;
   }

   // verifier si on a toujours une queue
   if (liste->queue) {
      // notre queue n'a pas d'element suivant
      liste->queue->suivant = NULL;
   }

   // Liste a jour, il suffit de restituer la memoire
   while (elementActuel) {
      Element *tmp = elementActuel;
      elementActuel = elementActuel->suivant;
      free(tmp);
   }
}
```

```
static Status getElement(const Liste *liste, size_t position, Element **element) {
   if (estVide(liste)) {
      return POSITION_NON_VALIDE;
   }

   Element *elementActuel = liste->tete;

   // on itere sur notre liste tant que on a un element valable
   // en decrementant la position
   while (elementActuel && position--) {
      elementActuel = elementActuel->suivant;
   }

   if (!elementActuel) {
      return POSITION_NON_VALIDE;
   }

   if (element) {
      *element = elementActuel;
   }
   return OK;
}

bool sontEgales(const Liste *liste1, const Liste *liste2) {
   Element *elementActuelL1 = liste1->tete;
   Element *elementActuelL2 = liste2->tete;

   // itérer tant qu'on a des pointeurs valables
   while (elementActuelL1 && elementActuelL2) {
      if (elementActuelL1->info != elementActuelL2->info) {
         return false;
      }
      elementActuelL1 = elementActuelL1->suivant;
      elementActuelL2 = elementActuelL2->suivant;
   }
   // si on est arrivé à la fin des deux listes, p1 et p2 sont nulles et les listes
   // sont égales
   return elementActuelL1 == NULL && elementActuelL2 == NULL;
}
```