**Main.c**

/\*

----------------------------------------------------------------------------------

Nom du fichier : main.h

Auteur(s) : Romain Fleury, Nicolet Victor

Date creation : 26.04.2023

Description : Programme de test pour la librairie "listes\_dynamique"

Remarque(s) : -

Compilateur : Mingw-w64 gcc 12.2.0

----------------------------------------------------------------------------------

\*/

#include <stdio.h>

#include "listes\_dynamiques.h"

**bool** critere(**size\_t** pos, **const** Info \*n) {

//return true;

**if** (pos % 2 == 0 || \*n > 1 && \*n < 6) {

**return** **true**;

}

**return** **false**;

}

**int** main() {

**size\_t** position = 0;

**int** tailleListe;

Info x = 0;

printf("-----====### INITIALISATION ###====-----\n");

Liste \*lptr1 = initialiser();

printf("\nINITIALISATION : tableau de variable et de pointeur.\n");

printf("%-20s %-20s %-20s \n", "Variable", "Adresse", "Adresse pointeur");

printf("%-20s %-20p %-20p \n", "lptr", (**void** \*) &lptr1, (**void** \*) lptr1);

printf("%-20s %-20p %-20p \n", "lptr->tete", (**void** \*) &(lptr1->tete),

lptr1->tete);

printf("%-20s %-20p %-20p \n", "lptr->queue", (**void** \*) &(lptr1->queue),

lptr1->queue);

printf("\nINITIALISATION : test des fonctions estVide() et longueur().\n");

printf("%-20s : %d \n", "Liste est vide", estVide(lptr1));

printf("%-20s : %zu \n", "Taille de la liste", longueur(lptr1));

printf("\nINITIALISATION : Affichage d'une liste vide.\n");

printf("FORWARD -> ");

afficher(lptr1, FORWARD);

printf(" et BACKWARD -> ");

afficher(lptr1, BACKWARD);

printf("\n");

printf("\n-----====### SUPPRIMER ET INSERER EN TETE ###====-----\n");

printf("\nSUPPRIMER EN TETE : utiliser supprimerEnTete() sur une liste vide.\n");

printf("%-18s : %d \n%-18s : ", "-Status",

supprimerEnTete(lptr1, &x), "-Liste");

afficher(lptr1, FORWARD);

printf("\n");

printf("\nINSERER EN TETE : utiliser insererEnTete() pour "

"inserer les valeurs de 0 a 5.\n");

x = 0;

printf("%-18s : %d \n%-18s : ", "-Status",

insererEnTete(lptr1, &x), "-Liste");

**for** (Info i = 1; i <= 5; ++i)

insererEnTete(lptr1, &i);

afficher(lptr1, FORWARD);

printf("\n");

printf("\nSUPPRIMER EN TETE : utiliser supprimerEnTete() sur une liste.\n");

printf("%-18s : %d \n", "-Status", supprimerEnTete(lptr1, &x));

printf("%-18s : %d \n%-18s : ", "-Valeur supprimee", x, "-Liste");

afficher(lptr1, FORWARD);

printf("\n");

printf("\n-----====### SUPPRIMER ET INSERER EN QUEUE ###====-----\n");

Liste \*lptr2 = initialiser();

printf(

"\nSUPPRIMER EN QUEUE : utiliser supprimerEnQueue() sur une liste vide.\n");

printf("%-18s : %d \n%-18s : ", "-Status",

supprimerEnQueue(lptr2, &x), "-Liste");

afficher(lptr2, FORWARD);

printf("\n");

printf("\nINSERER EN QUEUE : utiliser insererEnQueue() pour "

"inserer les valeurs de 0 a 5.\n");

x = 0;

printf("%-18s : %d \n%-18s : ", "-Status",

insererEnQueue(lptr2, &x), "-Liste");

**for** (Info i = 1; i <= 5; ++i)

insererEnQueue(lptr2, &i);

afficher(lptr2, FORWARD);

printf("\n");

printf("\nSUPPRIMER EN QUEUE : utiliser supprimerEnQueue() sur une liste.\n");

printf("%-18s : %d \n", "-Status", supprimerEnQueue(lptr2, &x));

printf("%-18s : %d \n%-18s : ", "-Valeur supprimee", x, "-Liste");

afficher(lptr2, FORWARD);

printf("\n");

printf("\n-----====### VIDER ###====-----");

printf("\n\nVIDER : utiliser vider() sur une liste vide.\n");

Liste \*listeTestVider = initialiser();

printf("%-45s : ", "-Initilisation d'une liste vide");

afficher(listeTestVider, FORWARD);

vider(listeTestVider, position);

printf("\n%s %-3zu : ", "-Apres utilisation de vider() en position", position);

afficher(listeTestVider, FORWARD);

printf("\n\nVIDER : utiliser vider() sur une liste de un element.\n");

printf("%-45s : ", "-Ajout d'un element dans la liste");

x = 0;

insererEnQueue(listeTestVider, &x);

afficher(listeTestVider, FORWARD);

printf("\n%s %-3zu : ", "-Apres utilisation de vider() en position", position);

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

printf("\n\nVIDER : utiliser vider() sur une liste de plusieurs elements.\n");

printf("%-45s : ", "-Ajout de plusieurs elements dans la liste");

tailleListe = 10;

**for** (Info i = 0; i < tailleListe; ++i)

insererEnQueue(listeTestVider, &i);

afficher(listeTestVider, FORWARD);

position = 5;

printf("\n%s %-3zu : ", "-Apres utilisation de vider() en position", position);

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

position = 2;

printf("\n%s %-3zu : ", "-Apres utilisation de vider() en position", position);

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

position = 0;

printf("\n%s %-3zu : ", "-Apres utilisation de vider() en position", position);

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

printf(

"\n\nVIDER : utiliser une position plus grande que la taille de la liste\n");

tailleListe = 5;

position = 6;

**for** (Info i = 0; i < tailleListe; ++i)

insererEnQueue(listeTestVider, &i);

printf("%-59s : ", "-Liste de base");

afficher(listeTestVider, FORWARD);

printf("\n%s %zu %s %d : ", "-Utiliser vider() en position",

position, "sur une liste de longueur", tailleListe);

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

printf("\n%-59s : ", "-Vider completement la liste");

position = 0;

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

position = 3;

printf("\n%s %zu %-27s : ", "-Utiliser vider() en position",

position, "sur une liste vide");

vider(listeTestVider, position);

afficher(listeTestVider, FORWARD);

printf("\n");

printf("\n-----====### ELEMENTS ###====-----\n");

tailleListe = 5;

printf("\nELEMENT : initialisation et ajout de %d valeur dans une liste.\n",

tailleListe);

Liste \*lptr3 = initialiser();

**for** (Info i = 0; i < tailleListe; ++i)

insererEnTete(lptr3, &i);

printf("%s : ", "-Liste apres initialisation et ajouts d'elements");

afficher(lptr3, FORWARD);

printf("\n");

printf("%-48s : %d \n",

"-Element de tete", lptr3->tete->info);

printf("%-48s : %d \n",

"-Element suivant de tete", lptr3->tete->suivant->info);

printf("%-48s : %d \n",

"-Element de queue", lptr3->queue->info);

printf("%-48s : %d \n",

"-Element precedent de queue", lptr3->queue->precedent->info);

printf("\n");

printf("-----====### EGALITE ###====-----\n");

printf("\nEGALITE : 1 = egales / 0 = non-egales\n");

vider(lptr1, 0);

vider(lptr2, 0);

printf("%-60s : ", "-Egalite de deux listes vides");

afficher(lptr1, FORWARD);

printf(" et ");

afficher(lptr2, FORWARD);

printf(" %16s %d", "=", sontEgales(lptr2, lptr1));

printf("\n");

**for** (Info i = 0; i < 4; ++i)

insererEnTete(lptr2, &i);

**for** (Info i = 0; i < 4; ++i)

insererEnTete(lptr1, &i);

printf("%-60s : ", "-Egalite de deux listes de memes infos et meme ordre");

afficher(lptr1, FORWARD);

printf(" et ");

afficher(lptr2, FORWARD);

printf(" %2s %d", "=", sontEgales(lptr2, lptr1));

printf("\n");

vider(lptr1, 0);

**for** (Info i = 4; i > 0; --i)

insererEnTete(lptr1, &i);

printf("%-60s : ", "-Egalite de deux listes de memes infos et d'ordre different");

afficher(lptr1, FORWARD);

printf(" et ");

afficher(lptr2, FORWARD);

printf(" %2s %d", "=", sontEgales(lptr2, lptr1));

printf("\n");

vider(lptr2, 2);

printf("%-60s : ", "-Egalite de deux listes de differente taille");

afficher(lptr1, FORWARD);

printf(" et ");

afficher(lptr2, FORWARD);

printf(" %6s %d", "=", sontEgales(lptr2, lptr1));

printf("\n");

printf("\n-----====### SUPPRIMER SELON CRITERE ###====-----\n");

printf("\nLes criteres qui permette la suppression utilises pour la fonction "

"critere sont les suivant :\n"

"-La position doit etre impaire ou la valeur doit se trouver dans "

"l'intervalle ]1,6[.\n");

tailleListe = 10;

Liste \*listeTestSupprimerSelonCritere = initialiser();

**for** (Info i = 0; i < tailleListe; ++i)

insererEnQueue(listeTestSupprimerSelonCritere, &i);

printf("\n%s %d %-12s : ", "-Creation d'une liste de", tailleListe, "elements");

afficher(listeTestSupprimerSelonCritere, FORWARD);

printf("\n");

supprimerSelonCritere(listeTestSupprimerSelonCritere, critere);

printf("%-40s : ", "-La liste apres supprimerSelonCritere()");

afficher(listeTestSupprimerSelonCritere, FORWARD);

printf("\n");

**return** 0;

}

**Liste\_dynamique.h**

/\*

----------------------------------------------------------------------------------

Nom du fichier : listes\_dynamiques.h

Auteur(s) : René Rentsch

Date creation : 18.04.2023

Description : Librairie permettant la gestion de listes doublement chaînées

non circulaires

Remarque(s) : -

Compilateur : Mingw-w64 gcc 12.2.0

----------------------------------------------------------------------------------

\*/

#ifndef LISTES\_DYNAMIQUES\_H

#define LISTES\_DYNAMIQUES\_H

#include <stdbool.h>

#include <stddef.h>

// Pour la gestion des "exceptions"

**typedef** enum **{**

OK**,** MEMOIRE\_INSUFFISANTE**,** LISTE\_VIDE**,** POSITION\_NON\_VALIDE

**}** Status**;**

// Modes d'affichage possibles de la liste

// FORWARD : dans le sens tete -> queue

// BACKWARD : dans le sens queue -> tete

**typedef** enum **{**

FORWARD**,** BACKWARD

**}** Mode**;**

**typedef** int Info**;**

**typedef** struct element **{**

Info info**;**

struct element**\*** suivant**;**

struct element**\*** precedent**;**

**}** Element**;**

**typedef** struct **{**

Element**\*** tete**;**

Element**\*** queue**;**

**}** Liste**;**

// ------------------------------------------------------------------------------

// Initialisation de la liste.

// N.B. Cette fonction doit obligatoirement être utilisée pour se créer une liste

// car elle garantit la mise à NULL des champs tete et queue de la liste

// Renvoie NULL en cas de mémoire insuffisante

Liste**\*** initialiser**(**void**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Renvoie true si liste est vide, false sinon.

bool estVide**(**const Liste**\*** liste**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Renvoie combien il y a d'éléments dans liste.

size\_t longueur**(**const Liste**\*** liste**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Affiche le contenu intégral de liste sous la forme : [info\_1,info\_2,...]

// Dans le cas d'une liste vide, affiche : []

// En mode FORWARD, resp. BACKWARD, l'affichage se fait en parcourant liste

// dans le sens tete -> queue, resp. queue -> tete.

void afficher**(**const Liste**\*** liste**,** Mode mode**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Insère un nouvel élément (contenant info) en tête de liste.

// Renvoie OK si l'insertion s'est déroulée avec succès et MEMOIRE\_INSUFFISANTE

// s'il n'y a pas assez de mémoire pour créer le nouvel élément.

Status insererEnTete**(**Liste**\*** liste**,** const Info**\*** info**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Insère un nouvel élément (contenant info) en queue de liste.

// Renvoie OK si l'insertion s'est déroulée avec succès et MEMOIRE\_INSUFFISANTE

// s'il n'y a pas assez de mémoire pour créer le nouvel élément.

Status insererEnQueue**(**Liste**\*** liste**,** const Info**\*** info**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Renvoie, via le paramètre info, l'info stockée dans l'élément en tête de liste,

// puis supprime, en restituant la mémoire allouée, ledit élément.

// Renvoie LISTE\_VIDE si la liste passée en paramètre est vide, OK sinon.

Status supprimerEnTete**(**Liste**\*** liste**,** Info**\*** info**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Renvoie, via le paramètre info, l'info stockée dans l'élément en queue de liste,

// puis supprime, en restituant la mémoire allouée, ledit élément.

// Renvoie LISTE\_VIDE si la liste passée en paramètre est vide, OK sinon.

Status supprimerEnQueue**(**Liste**\*** liste**,** Info**\*** info**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Supprime, en restituant la mémoire allouée, tous les éléments de la liste qui

// vérifient le critère passé en second paramètre.

// Exemple: on souhaite supprimer de la liste tous les éléments dont la position est

// impaire et pour lesquels info est compris dans un certain intervalle de valeurs

void supprimerSelonCritere**(**Liste**\*** liste**,**

bool **(\***critere**)(**size\_t position**,** const Info**\*** info**));**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Supprime, en restituant la mémoire allouée, tous les éléments de la liste

// à partir de la position position

// N.B. Vider à partir de la position 0 signifie vider toute la liste.

void vider**(**Liste**\*** liste**,** size\_t position**);**

// ------------------------------------------------------------------------------

// ------------------------------------------------------------------------------

// Renvoie true si liste1 et liste2 sont égales (au sens mêmes infos et infos

// apparaissant dans le même ordre), false sinon.

// N.B. 2 listes vides sont considérées comme égales.

bool sontEgales**(**const Liste**\*** liste1**,** const Liste**\*** liste2**);**

// ------------------------------------------------------------------------------

#endif

**Liste\_dynamique.c**

/\*

----------------------------------------------------------------------------------

Nom du fichier : listes\_dynamiques.c

Auteur(s) : Romain Fleury, Victor Nicolet

Date creation : 26.04.2023

Description : Librairie permettant la gestion de listes doublement chaînées

non circulaires

Remarque(s) : le status "POSITION\_NON\_VALIDE" n'est jamais utilisé car la seule

fonction qui utilise "position" ne retourne rien (void)

Compilateur : Mingw-w64 gcc 12.2.0

----------------------------------------------------------------------------------

\*/

#include <stdlib.h>

#include <stdio.h>

#include "listes\_dynamiques.h"

Liste \*initialiser(**void**) {

Liste \*lptr = (Liste \*) calloc(1, **sizeof**(Liste));

**return** lptr;

}

**bool** estVide(**const** Liste \*liste) {

**if** (liste->tete == **NULL** && liste->queue == **NULL**) {

**return** **true**;

}

**return** **false**;

}

**size\_t** longueur(**const** Liste \*liste) {

**if** (estVide(liste)) {

**return** 0;

} **else** {

Element \*e = liste->tete;

**if** (e->suivant == **NULL**) {

**return** 1;

} **else** {

**size\_t** compteur = 1;

**while** (e->suivant != **NULL**) {

++compteur;

e = e->suivant;

}

**return** compteur;

}

}

}

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

**if** (estVide(liste)) {

printf("[]");

} **else** {

**if** (mode) { // forward

Element \*eptr = liste->queue;

printf("[");

**for** (**size\_t** i = 0; i < longueur(liste); i++) {

printf("%d", eptr->info);

**if** (i != longueur(liste) - 1) { printf(","); }

eptr = eptr->precedent;

}

printf("]");

} **else** { // backward

Element \*eptr = liste->tete;

printf("[");

**for** (**size\_t** i = 0; i < longueur(liste); i++) {

printf("%d", eptr->info);

**if** (i != longueur(liste) - 1) { printf(","); }

eptr = eptr->suivant;

}

printf("]");

}

}

}

Status insererEnTete(Liste \*liste, **const** Info \*info) {

Element \*eptr = (Element \*) calloc(1, **sizeof**(Element));

**if** (eptr != **NULL**) {

**if** (estVide(liste)) {

liste->tete = eptr;

liste->queue = eptr;

liste->tete->info = \*info;

**return** OK;

} **else** {

Element \*tmp = liste->tete;

liste->tete = eptr;

liste->tete->suivant = tmp;

liste->tete->suivant->precedent = liste->tete;

liste->tete->info = \*info;

**return** OK;

}

} **else** {

**return** MEMOIRE\_INSUFFISANTE;

}

}

Status insererEnQueue(Liste \*liste, **const** Info \*info) {

Element \*eptr = (Element \*) calloc(1, **sizeof**(Element));

**if** (eptr != **NULL**) {

**if** (estVide(liste)) {

liste->tete = eptr;

liste->queue = eptr;

liste->queue->info = \*info;

**return** OK;

} **else** {

Element \*tmp = liste->queue;

liste->queue = eptr;

liste->queue->precedent = tmp;

liste->queue->precedent->suivant = liste->queue;

liste->queue->info = \*info;

**return** OK;

}

} **else** {

**return** MEMOIRE\_INSUFFISANTE;

}

}

Status supprimerEnTete(Liste \*liste, Info \*info) {

**if** (estVide(liste)) {

**return** LISTE\_VIDE;

} **else** {

**if** (longueur(liste) == 1) {

free(liste->tete);

liste->tete = **NULL**;

liste->queue = **NULL**;

**return** OK;

} **else** {

Element \*eptr = liste->tete;

\*info = eptr->info;

liste->tete = liste->tete->suivant;

free(eptr);

liste->tete->precedent = **NULL**;

**return** OK;

}

}

}

Status supprimerEnQueue(Liste \*liste, Info \*info) {

**if** (estVide(liste)) {

**return** LISTE\_VIDE;

} **else** {

**if** (longueur(liste) == 1) {

free(liste->queue);

liste->tete = **NULL**;

liste->queue = **NULL**;

**return** OK;

} **else** {

Element \*eptr = liste->queue;

\*info = liste->queue->info;

liste->queue = liste->queue->precedent;

free(eptr);

liste->queue->suivant = **NULL**;

**return** OK;

}

}

}

**void** supprimerSelonCritere(Liste \*liste, **bool** (\*critere)(**size\_t** position,

**const** Info \*info)) {

**if** (estVide(liste)) {

**return**;

} **else** {

Element \*eptr = liste->tete;

**size\_t** i = 0; // position physique dans la liste

**size\_t** p = 0; // position relative dans la liste (paramètres de la fonction critère)

Info x;

**while** (eptr != **NULL**) {

**if** (critere(p, &eptr->info)) {

**if** (i == 0) { // effacer le premier

supprimerEnTete(liste, &x);

} **else** **if** (i == longueur(liste)) { // effacer le dernier

supprimerEnQueue(liste, &x);

**break**; // superflu ?

} **else** { // Effacer au milieu

Element \*tmp = eptr; // avancer dans la liste jusqu'a l'élément à supprimer

eptr->suivant->precedent = eptr->precedent;

eptr->precedent->suivant = eptr->suivant;

free(tmp);

tmp = **NULL**;

}

eptr = liste->tete;

**for** (**size\_t** j = 1; j < i; j++) { eptr = eptr->suivant; }

} **else** {

i++;

eptr = eptr->suivant;

}

++p;

}

}

}

**void** vider(Liste \*liste, **size\_t** position) {

**if** (position >= longueur(liste)) {

**return**;

} **else** {

Element \*eptr = liste->tete;

Info x;

**for** (**size\_t** i = 0; i < position; i++)

eptr = eptr->suivant;

**while** (liste->queue != eptr->precedent)

supprimerEnQueue(liste, &x);

}

}

**bool** sontEgales(**const** Liste \*liste1, **const** Liste \*liste2) {

**if** (longueur(liste1) == longueur(liste2)) {

Element \*ptr1 = liste1->tete, \*ptr2 = liste2->tete;

**for** (**size\_t** i = 0; i < longueur(liste1); i++) {

**if** (ptr1->info != ptr2->info) {

**return** **false**;

}

}

**return** **true**;

} **else** {

**return** **false**;

}

}