```
/*
 ------------------------------------------------------------------------------
 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
```