CPPL1 : TD 3 : C : Liste bi-chaînée

Nicolas Vansteenkiste Romain Absil Jonas Beleho *
(Esi – He2b)

Année académique 2017 – 2018

Ce TD [1] aborde l'implémentation en langage C du type abstrait de données appelé liste bi-chaînée [2].

Durée : 2 séances.

Ex. 3.1 Soit :

```
struct DLNode
{
    struct DLNode * previous;
    struct DLNode * next;
    value_t         value;
};
```

Il s'agit d'un type structuré qui sert à représenter les éléments d'une liste bi-chaînée [3], aussi appelée liste bidirectionnelle ou liste doublement chaînée. La signification précise de ses champs est décrite dans le fichier `dlnode.h` reproduit en annexe A.2.

*Et aussi, lors des années passées : Monica Bastreghi, Stéphan Monbaliu, Anne Rousseau et Moussa Wahid.

1. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td03_c/td03_c.pdf
2. https://en.wikipedia.org/wiki/Doubly_linked_list
3. https://fr.wikipedia.org/wiki/Liste_cha%C3%AEn%C3%A9e#Liste_doublement_cha.C3.AEn.C3.A9e

Le type `value_t` est un alias de type [4] quelconque. Il est défini dans le fichier `value_t.h`. On en trouve un exemple pour le type `int` en annexe A.1.

Implémentez les fonctions de manipulation de `struct DLNode` suivantes, dont une documentation précise est fournie comme pour doxygen [5] dans `dlnode.h` (voir l'annexe A.2) :

```
struct DLNode * newDLN(value_t value);

void deleteDLN(struct DLNode * * adpDLN);

struct DLNode * getPreviousDLN(const struct DLNode * pDLN);

struct DLNode * getNextDLN(const struct DLNode * pDLN);

value_t getValueDLN(const struct DLNode * pDLN);

void setPreviousDLN(struct DLNode * pDLN,
                    struct DLNode * pNewPrevious);

void setNextDLN(struct DLNode * pDLN, struct DLNode * pNewNext);

void setValueDLN(struct DLNode * pDLN, value_t newValue);
```

Ex. 3.2 Implémentez les fonctions d'utilisation de `struct DLNode` (voir Ex. 3.1) suivantes, dont une documentation précise est fournie dans `dlnode_utility.h` reproduit en annexe A.3 :

```
struct DLNode * forwardDLN(struct DLNode * pDLN, int value);

struct DLNode * backDLN(struct DLNode * pDLN, int value);
```

Ex. 3.3 Soit :

```
struct DLList
{
    struct DLNode * head;
    struct DLNode * tail;
};
```

Le type structuré `struct DLList` sert à représenter une liste bi-chaînée. Le type de ses champs `head` et `tail` est `struct DLNode *` tel que défini à l'Ex. 3.1. La signification précise de ces champs est décrite dans le fichier `dllist.h` reproduit en annexe A.4.

Implémentez les fonctions de manipulation de `struct DLList` suivantes, dont une documentation précise est fournie dans `dllist.h` (voir l'annexe A.4) :

4. https://en.wikipedia.org/wiki/Typedef
5. https://www.stack.nl/~dimitri/doxygen/

```
struct DLList * newDLL();

void deleteDLL(struct DLList * * adpDLL);

void clearDLL(struct DLList * pDLL);

struct DLNode * getHeadDLL(const struct DLList * pDLL);

value_t getHeadValueDLL(const struct DLList * pDLL);

struct DLNode * getTailDLL(const struct DLList * pDLL);

value_t getTailValueDLL(const struct DLList * pDLL);

bool isEmptyDLL(const struct DLList * pDLL);

size_t getSizeDLL(const struct DLList * pDLL);

struct DLNode * insertHeadDLL(struct DLList * pDLL, value_t value);

struct DLNode * removeHeadDLL(struct DLList * pDLL);

struct DLNode * insertTailDLL(struct DLList * pDLL, value_t value);

struct DLNode * removeTailDLL(struct DLList * pDLL);

struct DLNode * insertAfterDLL(struct DLList * pDLL,
                               struct DLNode * pDLN,
                               value_t newValue);

struct DLNode * insertBeforeDLL(struct DLList * pDLL,
                                struct DLNode * pDLN,
                                value_t newValue);

struct DLNode * removeDLL(struct DLList * pDLL,
                          struct DLNode * pDLN);
```

Ex. 3.4 Implémentez les fonctions d'utilisation de `struct DLList` (voir Ex. 3.3) suivantes, dont une documentation précise est fournie dans `dllist_utility.h` reproduit en annexe A.5 :

```
value_t * to_arrayDLL(size_t * size, const struct DLList * pDLL);
```

```
value_t * to_array_reverseDLL(size_t * size,
                               const struct DLList * pDLL);

struct DLList * reverseDLL(const struct DLList * pDLL);

void sortDLL(struct DLList * pDLL, int (* comp)(value_t, value_t),
             enum SortingMethod sm);

struct DLList * mergeDLL(const struct DLList * pDLL_l,
                         const struct DLList * pDLL_r,
                         int (* comp)(value_t, value_t));
```

L'énumération `enum SortingMethod` utilisée par la fonction `sortDLL()` est définie et documentée dans le fichier `dllist_utility.h` (annexe A.5).

A. Fichiers d'en-têtes

A.1. `value_t.h`

```
/*!
 * \file value_t.h
 *
 * \brief Définition de l'alias du type contenu par un élément
 *        de liste bi-chaînée.
 */
#ifndef VALUE_T_H
#define VALUE_T_H

/*!
 * \brief Type de la valeur contenue par un élément de liste
 *        bidirectionnelle struct DLNode.
 */
typedef int value_t;

#endif // VALUE_T_H
```

A.2. `dlnode.h`

```
/*!
 * \file dlnode.h
 *
 * \brief Définition d'un type représentant un élément de liste
 *        bi-chaînée.
```

```
 */
#ifndef DLNODE_H
#define DLNODE_H

#include "value_t.h"

/*!
 * \brief Valeurs d'erreurs associées à un élément de liste.
 */
enum DLNError
{
    /*!
     * \brief Erreur lors d'une allocation mémoire d'un élément de
     *        liste.
     */
    EDLNMEMORYFAIL = 50
};

/*!
 * \brief Structure représentant le type d'un élément d'une
 *        liste doublement chaînée
 *        ([doubly linked list]
 *         (https://en.wikipedia.org/wiki/Doubly_linked_list)).
 */
struct DLNode
{
    /*!
     * \brief Adresse de l'élément _précédant_ dans la liste.
     *
     * S'il n'y a pas d'élément précédant, `previous` vaut `NULL`.
     */
    struct DLNode * previous;

    /*!
     * \brief Adresse de l'élément _suivant_ dans la liste.
     *
     * S'il n'y a pas d'élément suivant, `next` vaut `NULL`.
     */
    struct DLNode * next;

    /*!
     * \brief Valeur conservée par l'élément de la liste.
     */
    value_t         value;
```

```
50 };
51
52 /*!
53  * \brief Création d'une instance de struct DLNode.
54  *
55  * L'élément de liste créé est dans un état valide : il ne possède ni
56  * précédent, ni suivant.
57  *
58  * Il est alloué dynamiquement et doit donc être détruit quand
59  * son usage n'est plus requis.
60  *
61  * Si l'allocation mémoire échoue :
62  *   + `errno` est mis à ::EDLNMEMORYFAIL ;
63  *   + `NULL` est retourné.
64  *
65  * \param value la valeur contenue dans le struct DLNode.
66  *
67  * \return l'adresse du struct DLNode créé.
68  */
69 struct DLNode * newDLN(value_t value);
70
71 /*!
72  * \brief Destruction d'une instance de struct DLNode.
73  *
74  * Le struct DLNode dont l'adresse est fournie est dédruit
75  * puis mis à `NULL`.
76  *
77  * Aucun maillage n'est modifié par la fonction !
78  *
79  * Si `adpDLN` est `NULL`, le comportement de la fonction est
80  * indéterminé.
81  *
82  * \param adpDLN adresse d'un pointeur de struct DLNode vers le
83  *               struct DLNode à détruire.
84  */
85 void deleteDLN(struct DLNode * * adpDLN);
86
87 /*!
88  * \brief Accès en lecture à l'élément précédant de la liste.
89  *
90  * Si `pDLN` est `NULL`, le comportement de la fonction est
91  * indéterminé.
92  *
93  * \param pDLN adresse du struct DLNode dont on désire connaître le
```

```
94  *             précédant.
95  *
96  * \return Adresse du struct DLNode précédant celui pointé par ‘pDLN‘.
97  */
98 struct DLNode * getPreviousDLN(const struct DLNode * pDLN);
99
100 /*!
101  * \brief Accès en lecture à l’élément suivant de la liste.
102  *
103  * Si ‘pDLN‘ est ‘NULL‘, le comportement de la fonction est
104  * indéterminé.
105  *
106  * \param pDLN adresse du struct DLNode dont on désire connaître le
107  *             suivant.
108  *
109  * \return Adresse du struct DLNode suivant celui pointé par ‘pDLN‘.
110  */
111 struct DLNode * getNextDLN(const struct DLNode * pDLN);
112
113 /*!
114  * \brief Accès en lecture à la valeur stockée dans l’élément de liste.
115  *
116  * Si ‘pDLN‘ est ‘NULL‘, le comportement de la fonction est
117  * indéterminé.
118  *
119  * \param pDLN adresse du struct DLNode dont on désire connaître la
120  *             valeur qu’il contient.
121  *
122  * \return valeur contenue dans le struct DLNode pointé par ‘pDLN‘.
123  */
124 value_t getValueDLN(const struct DLNode * pDLN);
125
126 /*!
127  * \brief Accès en écriture à l’élément précédant de la liste.
128  *
129  * Seul le maillage du struct DLNode pointé par ‘pDLN‘ est modifié
130  * par cette fonction.
131  * Celui de l’élément pointé par ‘pNewPrevious‘ n’est pas modifié. La
132  * mémoire n’est pas gérée ici.
133  *
134  * Si ‘pDLN‘ est ‘NULL‘, le comportement de la fonction est
135  * indéterminé.
136  *
137  * \param pDLN adresse du struct DLNode dont on désire modifier le
```

```
138  *              précédant.
139  * \param pNewPrevious adresse du nouveau struct DLNode précédant
140  *                     celui pointé par ‘pDLN‘.
141  */
142 void setPreviousDLN(struct DLNode * pDLN,
143                     struct DLNode * pNewPrevious);
144 
145 /*!
146  * \brief Accès en écriture à l’élément suivant de la liste.
147  *
148  * Seul le maillage du struct DLNode pointé par ‘pDLN‘ est modifié
149  * par cette fonction.
150  * Celui de l’élément pointé par ‘pNewNext‘ n’est pas modifié. La
151  * mémoire n’est pas gérée ici.
152  *
153  * Si ‘pDLN‘ est ‘NULL‘, le comportement de la fonction est
154  * indéterminé.
155  *
156  * \param pDLN adresse du struct DLNode dont on désire modifier le
157  *             suivant.
158  * \param pNewNext adresse du nouveau struct DLNode suivant celui
159  *                 pointé par ‘pDLN‘.
160  */
161 void setNextDLN(struct DLNode * pDLN, struct DLNode * pNewNext);
162 
163 /*!
164  * \brief Accès en écriture à la valeur contenue dans l’élément de
165  *        liste.
166  *
167  * Si ‘pDLN‘ est ‘NULL‘, le comportement de la fonction est
168  * indéterminé.
169  *
170  * \param pDLN adresse du struct DLNode dont on désire modifier la
171  *             valeur.
172  * \param newValue nouvelle valeur à conserver dans le struct DLNode
173  *                 pointé par ‘pDLN‘.
174  */
175 void setValueDLN(struct DLNode * pDLN, value_t newValue);
176 
177 #endif // DLNODE_H
```

A.3. dlnode_utility.h

```c
/*!
 * \file dlnode_utility.h
 *
 * \brief Fonctions diverses de traitement de éléments de listes
 *        bi-chaînées.
 */
#ifndef DLNODE_UTILITY_H
#define DLNODE_UTILITY_H

#include "dlnode.h"

/*!
 * \brief Accès en lecture d'un élément suivant en position donnée.
 *
 * \param pDLN adresse du struct DLNode dont on désire accèder à
 *             un suivant.
 * \param value position relative de l'élément désiré :
 *                + une valeur positive indique un déplacement via
 *                  le champ 'next' ;
 *                + une valeur négative indique un déplavement via
 *                  le champ 'previous'.
 *
 * \return adresse de l'élément 'value' positions après celui
 *         d'adresse 'pDLN' ou 'NULL' s'il n'y en a pas.
 */
struct DLNode * forwardDLN(struct DLNode * pDLN, int value);

/*!
 * \brief Accès en lecture d'un élément précédant en position donnée.
 *
 * \param pDLN adresse du struct DLNode dont on désire accèder à
 *             un précédant.
 * \param value position relative de l'élément désiré :
 *                + une valeur positive indique un déplacement via
 *                  le champ 'previous' ;
 *                + une valeur négative indique un déplavement via
 *                  le champ 'next'.
 *
 * \return adresse de l'élément 'value' positions avant celui
 *         d'adresse 'pDLN' ou 'NULL' s'il n'y en a pas.
 */
struct DLNode * backDLN(struct DLNode * pDLN, int value);
```

```
43
44 #endif // DLNODE_UTILITY_H
```

A.4. dllist.h

```
1  /*!
2   * \file dllist.h
3   *
4   * \brief Définition d'un type représentant une liste bi-chaînée.
5   */
6  #ifndef DLLIST_H
7  #define DLLIST_H
8
9  #include <stdbool.h>
10 #include <stddef.h>
11
12 #include "dlnode.h"
13
14 /*!
15  * \brief Valeurs d'erreurs associées à une liste.
16  */
17 enum DLLError
18 {
19     /*!
20      * \brief Erreur lors d'une allocation mémoire d'une liste ou
21      *        d'un de ses éléments.
22      */
23     EDLLMEMORYFAIL = 60,
24
25     /*!
26      * \brief Opération interdite car la liste est vide.
27      */
28     EDLLEMPTY
29 };
30
31 /*!
32  * \brief Structure représentant une liste doublement chaînée
33  *        ([doubly linked list]
34  *         (https://en.wikipedia.org/wiki/Doubly_linked_list)).
35  */
36 struct DLList
37 {
38     /*!
```

```
39      * \brief Tête de la liste bi-chaînée.
40      *
41      * L'élément de tête de liste est celui qui ne possède pas de
42      * précédant.
43      */
44     struct DLNode * head;
45
46     /*!
47      * \brief Queue de la liste bi-chaînée.
48      *
49      * L'élément en queue de liste est celui qui ne possède pas de
50      * suivant.
51      */
52     struct DLNode * tail;
53 };
54
55 /*!
56  * \brief Création d'une liste bi-chaînée.
57  *
58  * La liste est créée vide, c'est-à-dire que ses champs `head`
59  * et `tail` sont mis à `NULL`.
60  *
61  * Si l'allocation dynamique échoue :
62  *   + `errno` est mis à ::EDLLMEMORYFAIL ;
63  *   + `NULL` est retourné.
64  *
65  * \return adresse de la struct DLList créée.
66  */
67 struct DLList * newDLL();
68
69 /*!
70  * \brief Destruction d'une liste bi-chaînée.
71  *
72  * La struct DLList dont l'adresse est fournie est dédruite
73  * puis mise à `NULL`. La destruction de la liste implique
74  * la destruction de tous ses éléments.
75  *
76  * Si `adpDLL` est `NULL`, le comportement de la fonction est
77  * indéterminé.
78  *
79  * \param adpDLL adresse d'un pointeur de struct DLList vers la
80  *               struct DLList à détruire.
81  */
82 void deleteDLL(struct DLList * * adpDLL);
```

```
83
84  /*!
85   * \brief Destruction du contenu de la liste.
86   *
87   * Tous les struct DLNode qui constituent la liste sont détruits,
88   * mais pas la liste elle-même. En fin de fonction, la liste est
89   * vide, ses champs `head` et `tail` sont mis à `NULL`.
90   *
91   * Si `pDLL` est `NULL`, le comportement de la fonction est
92   * indéterminé.
93   *
94   * \param pDLL adresse de la struct DLList dont on désire
95   *             détruite les éléments.
96   */
97  void clearDLL(struct DLList * pDLL);
98
99  /*!
100  * \brief Accès en lecture de l'élément en tête de liste.
101  *
102  * Si la liste pointée par `pDLL` est vide, `NULL` est retourné.
103  *
104  * Si `pDLL` est `NULL`, le comportement de la fonction est
105  * indéterminé.
106  *
107  * \param pDLL adresse de la struct DLList dont on désire connaître
108  *             le struct DLNode de tête.
109  *
110  * \return adresse du struct DLNode en tête de la liste pointée
111  *         par `pDLL`.
112  */
113 struct DLNode * getHeadDLL(const struct DLList * pDLL);
114
115 /*!
116  * \brief Accès en lecture de la valeur de l'élément en tête de liste.
117  *
118  * Si la liste est vide :
119  *   + la valeur retournée est indéterminée ;
120  *   + `errno` est mis à ::EDLLEMPTY.
121  *
122  * Si `pDLL` est `NULL`, le comportement de la fonction est
123  * indéterminé.
124  *
125  * \param pDLL adresse de la struct DLList dont on désire connaître
126  *             la valeur de tête.
```

```
127  *
128  * \return valeur contenue dans le struct DLNode en tête de la liste
129  *         pointée par `pDLL`.
130  */
131 value_t getHeadValueDLL(const struct DLList * pDLL);
132 
133 /*!
134  * \brief Accès en lecture de l'élément en queue de liste.
135  *
136  * Si la liste pointée par `pDLL` est vide, `NULL` est retourné.
137  *
138  * Si `pDLL` est `NULL`, le comportement de la fonction est
139  * indéterminé.
140  *
141  * \param pDLL adresse de la struct DLList dont on désire connaître
142  *             le struct DLNode de queue.
143  *
144  * \return adresse du struct DLNode en queue de la liste pointée
145  *         par `pDLL`.
146  */
147 struct DLNode * getTailDLL(const struct DLList * pDLL);
148 
149 /*!
150  * \brief Accès en lecture de la valeur de l'élément en queue de liste.
151  *
152  * Si la liste est vide :
153  *   + la valeur retournée est indéterminée ;
154  *   + `errno` est mis à ::EDLLEMPTY.
155  *
156  * Si `pDLL` est `NULL`, le comportement de la fonction est
157  * indéterminé.
158  *
159  * \param pDLL adresse de la struct DLList dont on désire connaître
160  *             la valeur de queue.
161  *
162  * \return valeur contenue dans le struct DLNode en queue de la
163  *         liste pointée par `pDLL`.
164  */
165 value_t getTailValueDLL(const struct DLList * pDLL);
166 
167 /*!
168  * \brief Accès en lecture de la nature vide ou non de la liste.
169  *
170  * Si `pDLL` est `NULL`, le comportement de la fonction est
```

```
171  * indéterminé.
172  *
173  * \param pDLL adresse de la struct DLList dont on désire savoir
174  *             si elle est vide ou non.
175  *
176  * \return `true` si la liste pointée par `pDLL` ne contient aucun
177  *         struct DLNode, `false` sinon.
178  */
179 bool isEmptyDLL(const struct DLList * pDLL);
180 
181 /*!
182  * \brief Accès en lecture de la taille de la liste.
183  *
184  * La taille de la liste est le nombre de struct DLNode qui la
185  * constituent. Une liste vide est donc de taille nulle.
186  *
187  * Si `pDLL` est `NULL`, le comportement de la fonction est
188  * indéterminé.
189  *
190  * \param pDLL adresse de la struct DLList dont on désire
191  *             connaître la taille.
192  *
193  * \return nombre d'éléments de la liste pointée par `pDLL`.
194  */
195 size_t getSizeDLL(const struct DLList * pDLL);
196 
197 /*!
198  * \brief Insertion d'un élément en tête de liste.
199  *
200  * Si l'instanciation du struct DLNode destiné à être la
201  * nouvelle tête de liste échoue :
202  *   + la liste est laissée telle quelle ;
203  *   + `errno` est mis à ::EDLLMEMORYFAIL.
204  *
205  * Si `pDLL` est `NULL`, le comportement de la fonction est
206  * indéterminé.
207  *
208  * \param pDLL adresse de la liste dont on veut modifier
209  *             l'élément de tête.
210  * \param value valeur que doit renfermer l'élément en tête de liste.
211  *
212  * \return adresse de la nouvelle tête de liste... ou
213  *         l'ancienne en cas d'échec.
214  */
```

```
215 struct DLNode * insertHeadDLL(struct DLList * pDLL, value_t value);
216
217 /*!
218  * \brief Suppression de l'élément en tête de liste.
219  *
220  * Si la liste pointée par `pDLL` est initialement vide :
221  *   + la liste est laissée telle quelle ;
222  *   + `errno` est mis à ::EDLLEMPTY ;
223  *   + `NULL` est retourné.
224  *
225  * Si `pDLL` est `NULL`, le comportement de la fonction est
226  * indéterminé.
227  *
228  * \param pDLL adresse de la liste dont on veut ôter
229  *             l'élément de tête.
230  *
231  * \return adresse de la nouvelle tête de liste... ou
232  *         `NULL` si elle est désormais vide.
233  */
234 struct DLNode * removeHeadDLL(struct DLList * pDLL);
235
236 /*!
237  * \brief Insertion d'un élément en queue de liste.
238  *
239  * Si l'instanciation du struct DLNode destiné à être la
240  * nouvelle queue de liste échoue :
241  *   + la liste est laissée telle quelle ;
242  *   + `errno` est mis à ::EDLLMEMORYFAIL.
243  *
244  * Si `pDLL` est `NULL`, le comportement de la fonction est
245  * indéterminé.
246  *
247  * \param pDLL adresse de la liste dont on veut modifier
248  *             l'élément de queue.
249  * \param value valeur que doit renfermer l'élément en queue de liste.
250  *
251  * \return uadresse de la nouvelle queue de liste... ou
252  *         l'ancienne en cas d'échec.
253  */
254 struct DLNode * insertTailDLL(struct DLList * pDLL, value_t value);
255
256 /*!
257  * \brief Suppression de l'élément en queue de liste.
258  *
```

```
259  * Si la liste pointée par ‘pDLL‘ est initialement vide :
260  *   + la liste est laissée telle quelle ;
261  *   + ‘errno‘ est mis à ::EDLLEMPTY ;
262  *   + ‘NULL‘ est retourné.
263  *
264  * Si ‘pDLL‘ est ‘NULL‘, le comportement de la fonction est
265  * indéterminé.
266  *
267  * \param pDLL adresse de la liste dont on veut ôter
268  *             l'élément de queue.
269  *
270  * \return adresse de la nouvelle queue de liste... ou
271  *         ‘NULL‘ si elle est désormais vide.
272  */
273 struct DLNode * removeTailDLL(struct DLList * pDLL);
274
275 /*!
276  * \brief Insertion d'une nouvelle valeur dans la liste _après_ un
277  *        élément spécifique.
278  *
279  * Si l'instanciation du struct DLNode destiné à être inséré dans
280  * la liste échoue :
281  *   + la liste est laissée telle quelle ;
282  *   + ‘errno‘ est mis à ::EDLLMEMORYFAIL.
283  *
284  * La fonction ne vérifie pas que l'élément pointé par ‘pDLN‘ se
285  * trouve bien dans la liste pointée par ‘pDLL‘. Si ce n'est pas
286  * le cas, l'intégrité de la liste n'est pas garantie.
287  *
288  * Si ‘pDLL‘ ou ‘pDLN‘ sont ‘NULL‘, le comportement de la fonction
289  * est indéterminé.
290  *
291  * \param pDLL adresse de la liste dans laquelle on désire insérer
292  *             un nouvel élément.
293  * \param pDLN adresse de l'élément de la liste après lequel
294  *             l'insertion doit avoir lieu.
295  * \param newValue valeur conservée dans le nouvel élément à
296  *                 insérer.
297  *
298  * \return adresse du nouvel élément inséré... ou ‘pDLN‘
299  *         en cas d'échec.
300  */
301 struct DLNode * insertAfterDLL(struct DLList * pDLL,
302                                struct DLNode * pDLN,
```

```
303                                 value_t newValue);
304
305 /*!
306  * \brief Insertion d'une nouvelle valeur dans la liste _avant_ un
307  *        élément spécifique.
308  *
309  * Si l'instanciation du struct DLNode destiné à être inséré dans
310  * la liste échoue :
311  *   + la liste est laissée telle quelle ;
312  *   + `errno` est mis à ::EDLLMEMORYFAIL.
313  *
314  * La fonction ne vérifie pas que l'élément pointé par `pDLN` se
315  * trouve bien dans la liste pointée par `pDLL`. Si ce n'est pas
316  * le cas, l'intégrité de la liste n'est pas garantie.
317  *
318  * Si `pDLL` ou `pDLN` sont `NULL`, le comportement de la fonction
319  * est indéterminé.
320  *
321  * \param pDLL adresse de la liste dans laquelle on désire insérer
322  *             un nouvel élément.
323  * \param pDLN aderesse de l'élément de la liste avant lequel
324  *             l'insertion doit avoir lieu.
325  * \param newValue valeur conservée dans le nouvel élément à
326  *                 insérer.
327  *
328  * \return adresse du nouvel élément inséré... ou `pDLN`
329  *         en cas d'échec.
330  */
331 struct DLNode * insertBeforeDLL(struct DLList * pDLL,
332                                 struct DLNode * pDLN,
333                                 value_t newValue);
334
335 /*!
336  * \brief Suppression d'un élément de la liste.
337  *
338  * La fonction ne vérifie pas que l'élément pointé par `pDLN` se
339  * trouve bien dans la liste pointée par `pDLL`. Si ce n'est pas
340  * le cas, l'intégrité de la liste n'est pas garantie.
341  *
342  * Si la liste pointée par `pDLL` est initialement vide et que
343  * `pDLN` est bel et bien `NULL`, rien ne se passe.
344  *
345  * Si `pDLL` est `NULL` ou si `pDLN` est `NULL` alors que la liste,
346  * n'est pas vide, le comportement de la fonction est indéterminé.
```

```
347  *
348  * \param pDLL adresse de la liste dont on désire supprimer
349  *             un élément.
350  * \param pDLN adresse de l'élément à supprimer.
351  *
352  * \return adresse de l'élément de la liste qui se trouve, après
353  *         suppression, en même position dans la liste que l'élément
354  *         supprimé, avant sa suppression, c'est-à-dire l'adresse
355  *         de l'élément qui suivait l'élément supprimé avant la
356  *         suppression, ou l'élément de queue de liste si c'est
357  *         l'élément en queue de liste qui a été supprimé ou `NULL`
358  *         si la liste est finalement vide.
359  */
360 struct DLNode * removeDLL(struct DLList * pDLL, struct DLNode * pDLN);
361
362 #endif // DLLIST_H
```

A.5. dllist_utility.h

```
1  /*!
2   * \file dllist_utility.h
3   *
4   * \brief Fonctions diverses de traitement de listes bi-chaînées.
5   */
6  #ifndef DLLIST_UTILITY_H
7  #define DLLIST_UTILITY_H
8
9  #include "dllist.h"
10
11 /*!
12  * \brief Description de la méthode de tri.
13  *
14  * Cette énumération est utilisée par la fonction ::sortDLL().
15  */
16 enum SortingMethod
17 {
18     /*!
19      * \brief Tri par remaillage des éléments de la liste.
20      */
21     SORT_BY_CHANGING_LINK,
22     /*!
23      * \brief Tri par modification des valeurs des éléments de
24      *        la liste.
```

```
25       */
26      SORT_BY_CHANGING_VALUE
27  };
28
29  /*!
30   * \brief Conversion d'une liste en tableau dynamique.
31   *
32   * Le premier élément du tableau dynamique retourné est l'élément
33   * en tête de liste, le deuxième celui qui suit (`pDLL->head->next`),
34   * etc. jusqu'au dernier qui est la queue de liste.
35   *
36   * Si la liste est vide, la variable pointée par `size` est mise à
37   * 0 et `NULL` est retourné.
38   *
39   * Si l'allocation de mémoire échoue :
40   *   + le contenu de la variable pointée par `size` n'est pas
41   *     modifié ;
42   *   + `errno` est mis à ::EDLLMEMORYFAIL ;
43   *   + `NULL` est retourné.
44   *
45   * \param size adresse d'une variable où le nombre d'éléments du
46   *             tableau dynamique retourné.
47   * \param pDLL adresse de la liste à convertir en tableau.
48   *
49   * \return adresse du premier élément du tableau dynamique produit.
50   */
51  value_t * to_arrayDLL(size_t * size, const struct DLList * pDLL);
52
53  /*!
54   * \brief Conversion d'une liste en tableau dynamique.
55   *
56   * Le premier élément du tableau dynamique retourné est l'élément
57   * en queue de liste, le deuxième celui qui précède
58   * (`pDLL->tail->previous`), etc. jusqu'au dernier qui est la tête
59   * de liste.
60   *
61   * Si la liste est vide, la variable pointée par `size` est mise à
62   * 0 et `NULL` est retourné.
63   *
64   * Si l'allocation de mémoire échoue :
65   *   + le contenu de la variable pointée par `size` n'est pas
66   *     modifié ;
67   *   + `errno` est mis à ::EDLLMEMORYFAIL ;
68   *   + `NULL` est retourné.
```

```
69  *
70  * \param size adresse d'une variable où le nombre d'éléments du
71  *             tableau dynamique retourné.
72  * \param pDLL adresse de la liste à convertir en tableau.
73  *
74  * \return adresse du premier élément du tableau dynamique produit.
75  */
76 value_t * to_array_reverseDLL(size_t * size,
77                               const struct DLList * pDLL);
78
79 /*!
80  * \brief Production d'une liste inverse d'une originale
81  *
82  * Si `pDLL` est `NULL`, le comportement de la fonction
83  * est indéterminé.
84  *
85  * Si l'allocation de mémoire pour produire la liste fusionnée
86  * échoue :
87  *   + `errno` est mis à ::EDLLMEMORYFAIL ;
88  *   + `NULL` est retourné.
89  *
90  * \param pDLL l'adresse de la liste dont on désire produire
91  *             une liste inversée.
92  *
93  * \return adresse de la liste inverse de celle pointée par
94  *         `pDLL`.
95  */
96 struct DLList * reverseDLL(const struct DLList * pDLL);
97
98 /*!
99  * \brief Tri d'une liste.
100 *
101 * Si `pDLL` ou `comp` sont `NULL`, le comportement de la fonction
102 * est indéterminé.
103 *
104 * La fonction `comp` définit l'ordre des éléménent de la liste
105 * triée comme la fonction d'ordre utilisée par la fonction standard
106 * [qsort](http://en.cppreference.com/w/c/algorithm/qsort).
107 *
108 * Il est possible de choisir le type de tri :
109 *   + avec `sm` valant ::SORT_BY_CHANGING_LINK, le maillage des
110 *     éléments de la liste est modifié, mais les valeurs contenues
111 *     par les éléments ne sont pas modifiées ;
112 *   + avec `sm` valant ::SORT_BY_CHANGING_VALUE, les valeurs
```

```
113  *     contenues dans les éléments sont modifiées, mais le maillage
114  *     des éléments de la liste n'est pas modifié.
115  *
116  * \param pDLL adresse de la liste à trier.
117  * \param comp fonction d'ordre utilisée pour ordonner les éléments
118  *             de la liste.
119  * \param sm choix du type de tri.
120  */
121 void sortDLL(struct DLList * pDLL, int (* comp)(value_t, value_t),
122              enum SortingMethod sm);
123
124 /*!
125  * \brief Fusion de listes triées.
126  *
127  * La liste produite est ordonnée selon la fonction d'ordre `comp`.
128  *
129  * Si les listes à fusionner ne sont pas ordonnées comme la fonction
130  * `comp` le définit, le comportement de la fonction est indéterminé.
131  *
132  * Si l'allocation de mémoire pour produire la liste fusionnée
133  * échoue :
134  *   + `errno` est mis à ::EDLLMEMORYFAIL ;
135  *   + `NULL` est retourné.
136  *
137  * Si `pDLL_l` ou `pDLL_r` sont `NULL`, le comportement de la
138  * fonction est indéterminé.
139  *
140  * \param pDLL_l adresse d'une liste triée selon l'ordre défini par
141  *               `comp`.
142  * \param pDLL_r adresse d'une liste triée selon l'ordre défini par
143  *               `comp`.
144  * \param comp fonction d'ordre utilisée pour ordonner `pDLL_l`,
145  *             `pDLL_r` et la fusion de ces listes.
146  *
147  * \return adresse d'une liste dont le contenu est la fusion des
148  *         listes `pDLL_l` et `pDLL_r`.
149  */
150 struct DLList * mergeDLL(const struct DLList * pDLL_l,
151                          const struct DLList * pDLL_r,
152                          int (* comp)(value_t, value_t));
153
154 #endif // DLLIST_UTILITY_H
```