

# Structures de données

---

Semestre 4



---

# Avant-propos

---

Suite du module d’algorithmique et programmation, accent sur les structures de données

- Pile
- File
- Arbre
- ...

## Heures

- 24h de CTDI
- 26 de TDM

## Notation

**Contrôle intermédiaire** 30%

**Contrôle terminal** 50%

**TP** 20%

**TP Noté** 50%

**Devoir écrit** 25%

**Devoir TP** 25 %

---

# Table des matières

---

<b>1</b>	<b>Types de données Abstraits (TAD)</b>	<b>5</b>
1.1	Syntaxe des TAD . . . . .	5
1.2	Implémentation d'un TAD . . . . .	6
1.3	Protection du TAD . . . . .	9
<b>2</b>	<b>Structures de données classiques</b>	<b>11</b>
2.1	Pile . . . . .	11
2.2	Dynamique . . . . .	11
2.3	File . . . . .	13
2.4	File avec priorité . . . . .	18
2.5	Liste avec priorité . . . . .	18
<b>A</b>	<b>Cours sur les pointeurs en C</b>	<b>22</b>
A.1	Syntaxe . . . . .	22
A.2	Opérateur autorisés sur les pointeurs . . . . .	22
A.3	Pointeur sur fonction . . . . .	24
<b>B</b>	<b>Liste des codes sources</b>	<b>25</b>
<b>C</b>	<b>Table des figures</b>	<b>26</b>
<b>D</b>	<b>Exercices</b>	<b>27</b>
D.1	TAD . . . . .	27
D.2	Pointeurs . . . . .	27

# Types de données Abstraits (TAD)

C'est une méthode de spécification de structures de données (SD).

C'est utile pour la programmation « En large », c'est-à-dire à plusieurs, pour cela nous sommes obligés de travailler sur la communication et l'échange sur le code produit, on utilise pour cela les **spécifications** :

- Les Entrées Sorties du programme <sup>1</sup>
- Les données <sup>2</sup>

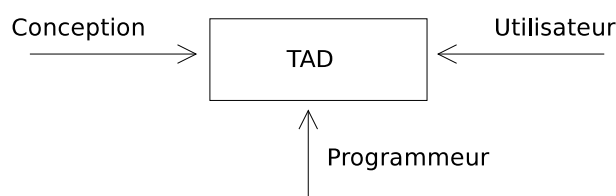


FIGURE 1.1 – Principe de base d'un TAD

## Ex Les entiers

**Utilisateur : Représentation Interne** 1, 2, 3, +, -, /, +, %

**Programmeur : Représentation Externe** Entiers « machine » 0000 0011  
pour le 3

## 1.1 Syntaxe des TAD

La syntaxe d'un TAD est répartie en deux étapes :

**La signature du TAD** <sup>3</sup> Donner les interfaces de la données

**La sémantique abstraite du TAD** <sup>4</sup> Décrire logiquement le fonctionnement de la données.

Une donnée c'est une ou un ensemble de valeurs mais aussi les opérations qui permettent de la manipuler. Cette étape nous donne :

- Les limitations de la donnée (préconditions)
- Les descriptions longueurs du fonctionnement de chaque opération

1. Vu au S3

2. Nous nous occuperons de cette partie

### 1.1.1 Signature du TAD Pile

Une pile est une structure de données qui permet de rassembler des éléments de telle sorte que le dernier élément entré dans la pile soit le premier à en sortir.<sup>5</sup>.

#### Signature de base

Sorte Pile

Utilise Élément, Booleen

#### Opérations

**creer**  $\rightarrow$  Pile

**empiler** Pile  $\times$  Element  $\rightarrow$  Pile

**estVide** Pile  $\rightarrow$  Booleen

**sommet** Pile  $\rightarrow$  Pile

**appartient** Pile  $\times$  Element  $\rightarrow$  Booleen

#### Signature étendue

##### Préconditions

–  $\text{sommet}(p) \Leftrightarrow \neg \text{estVide}(p)$

##### Axiones

Avant toute chose, on partitionne l'ensemble des opérations en deux sous ensembles :

- Les constructeurs
- Les opérateurs

L'ensemble des constructeurs est nécessaire et suffisant pour pouvoir gagner n'importe quelle valeur de la donnée

```
// On applique chaque constructeur à chaque opérateur et on décrit logiquement
// ce qu'il se passe
estVide(creer()) = true;
estVide(empiler(p, x)) = false;
depiler(creer()) = creer();
depiler(empiler(p, x)) = p;
sommet(empiler(p, x)) = x;
appartient(creer(), x) = false;
appartient(empiler(p, x), y) = (x = y)  $\vee$  appartient(p, y)
```

Listing 1.1 – Opérations du TAD Pile

## 1.2 Implémentation d'un TAD

1. Implémenter la structure de données
2. Implémenter les opérateurs
3. Séparer l'interface du corps des opérations

**But 1** Permet de modifier les opérations sans remettre en cause la manière d'utiliser le TAD

**But 2** Protéger les données

---

5. Last In First Out

## 1.2.1 Implémentation de la structure de données et des opérateurs

Trouver une représentation interne de la structure de données, celle-ci est contrainte par le langage choisi.

Celle-ci peut être statique ou dynamique<sup>6</sup>.

**Statique** La donnée ne peut plus changer de place ni de taille mémoire ou dynamique.

- Problème de gaspillage de place
- Avantage de l'efficacité

**Dynamique** La donnée peut changer de taille ou de place pendant l'exécution du programme.

- Pas de gaspillage de place
- Inconvénient de l'efficacité

### 1.2.1.1 Implémentation statique du TAD Pile

- Utilisation d'un tableau
- Utilisation d'un entier donnant le nombre d'éléments rangés dans la pile

```

1  #define N 1000
2
3  struct eltPile {
4      Element Tab[N];
5      int nb;
6  } Pile;
7
8  Pile creer() {
9      Pile p;
10     p.nb = 0;
11
12     return p;
13 }
14
15 Pile empiler(Pile p, Element x) {
16     assert(p.nb < N); // Si la condition est false alors arrête programme
17     p.tab[p.nb] = x;
18     p.nb++;
19
20     return (p);
21 }
22
23 int estVide(Pile p) {
24     return (p.nb == 0);
25 }
26
27 Pile depiler(Pile p) {
28     if(!estVide(p)) {
29         p.nb--;
30     }
31
32     return p;
33 }
34
35 Element sommet(Pile p) {
36     assert(!estVide(p)); // Pas indispensable masi plus robuste
37     return (p.tab[p.nb-1]);

```

6. Des exemples de structures de données dynamiques du TAD sont disponibles annexes ??

```
38 }
39
40 int appartient(Pile p, Element x) {
41     if(estVide(p))
42         return 0;
43
44     if(x == sommet(p)) {
45         return 1;
46     }
47
48     return (appartient(depiler(p), x));
49 }
```

Listing 1.2 – Implémentation des fonctions du type Pile en statique

### 1.2.1.2 Implémentation dynamique du TAD Pile

```
1  typedef struct etCel {
2      Element val;
3      struct etCel* suiv;
4  } Cel;
5
6  typedef cel* Pile;
7
8  Pile creer() {
9      return NULL;
10 }
11
12 Pile empiler(Pile p, Element x) {
13     Pile pAux;
14     pAux = (pile)malloc(sizeof(Cel));
15     assert(pAux != NULL);
16     pAux->val = x;
17     pAux->suiv = p;
18
19     return (pAux);
20 }
21
22 int estVide(Pile p) {
23     return (p == NULL);
24 }
25
26 Pile depiler(Pile p) {
27     Pile pAux = NULL;
28     if(p != NULL) {
29         pAux = p->suivant;
30         free(p);
31     }
32
33     return pAux;
34 }
35
36 Element sommet(Pile p) {
37     return (p->val);
38 }
39
40 int appartient(Pile p, Element x) {
41     if(estVide(p))
42         return 0;
43 }
```



```

44 while(!estVide(p)) {
45     if(p->suiv == x)
46         return 1;
47
48     p = p->suiv;
49 }
50
51 return 0;
52 }

```

Listing 1.3 – Implémentation des fonctions du type Pile en dynamique

## 1.3 Protection du TAD

La protection d'un TAD se fait en deux phases :

**séparer corps - interface** Bibliothèque

Protéger le type

### 1.3.1 Séparation du corps et de l'interface

Cela correspond à une bibliothèque, ainsi nous allons séparer le fichier source en trois fichiers :

#### 1.3.1.1 Fichiers

**fichier.h** Contient les prototypes de fonctions et les **typedef**.

**fichier.c** Contient **#include "fichier.h"** et les implémentations de fonctions sauf le main.

**testFichier.c** Contient **#include "fichier.h"** et le **main**.

#### 1.3.1.2 Compilation

- gcc -c fichier.c
- gcc -c testFichier.c
- gcc fichier.o testfichier.o -o nomExe

### 1.3.2 Protection du type

Nous allons étudier le cas de la pile statique.

```

#define N 1000
typedef struct etPile {
    element tab[N];
    int nb;
} Pile;

```

Listing 1.4 – Type de la pile statique originel – Présent dans le .h

Nous allons devoir cacher ce type afin que l'utilisateur ne le modifie pour cela, il sera caché dans le .c et un pointeur présent dans le .h.

```
typedef struct etPile* pile;
```

Listing 1.5 – Type de la pile statique – Présent dans le .h

```
#define N 1000
typedef struct etPile {
    element tab[N];
    int nb;
}Pile;
```

Listing 1.6 – Type de la pile statique – Présent dans le .c

Nous devons ainsi modifier le fichier source afin d'utiliser le pointeur sur pile.

```
Pile p;
p = (Pile)malloc(sizeof(PileInterne));
p->nb = 0

return p;
```

Listing 1.7 – Modification de la fonction `creer` s'adaptant à la protection de données

**R** Désormais nous ferons systématiquement la séparation corps - interface et la protection du type.

# Structures de données classiques

## 2.1 Pile

### 2.1.1 Statique

Cf exemple de cours section 1.1.1 page 6, une amélioration est également présente ci-dessous.

1. Implémenter la fonction permettant de remplacer toute les occurrences de l'élément  $x$  par l'élément  $y$  dans la pile.
2. Implémenter la fonction d'affichage de la Pile.
3. Réécrire les méthode de la pile statique pour prendre en compte la protection du type

**Rajouter dans le champ des opérations** `remplacerOccurence Pile  $\times$  Element  $\times$  Element  $\rightarrow$  Pile`

**Préconditions** rien

**Axiones**

```

1 | remplacerOccurence(creer(), x, y) = creer();
2 | remplacerOccurence(empiler(p, x), x1, x2) =
3 |   p1  $\wedge \forall z$  (appartient(p1, z)  $\rightarrow$  (z  $\neq$  x1) (empiler(p, x), z')  $\wedge$  z' = x1))

1 | Pile remplacer(Pile pPile, Element pX, Element pY) {
2 |   int i;
3 |   for(i=0 ; i < p.nb ; ++i) {
4 |     if(p.tab[i] == x) {
5 |       p.tab[i] = y;
6 |     }
7 |   }
8 |
9 |   return p;
10 | }

11 |
12 | void afficherPile(Pile pPile) {
13 |   int i;
14 |   for(i=0 ; i < p.nb; ++i) {
15 |     afficheElement(p.tab[i]);
16 |   }
17 | }
```

Listing 2.1 – TAD Pile

## 2.2 Dynamique

```
1 typedef struct etCel {
2     Element val;
3     struct etCel* suiv;
4 } Cel;
5
6 typedef cel* Pile;
7
8 Pile creer() {
9     return NULL;
10 }
11
12 Pile empiler(Pile p, Element x) {
13     Pile pAux;
14     pAux = (pile)malloc(sizeof(Cel));
15     assert(pAux != NULL);
16     pAux->val = x;
17     pAux->suiv = p;
18
19     return (pAux);
20 }
21
22 int estVide(Pile p) {
23     return (p == NULL);
24 }
25
26 Pile depiler(Pile p) {
27     Pile pAux = NULL;
28     if(p != NULL) {
29         pAux = p->suivant;
30         free(p);
31     }
32
33     return pAux;
34 }
35
36 Element sommet(Pile p) {
37     return (p->val);
38 }
39
40 int appartient(Pile p, Element x) {
41     if(estVide(p))
42         return 0;
43
44     while(!estVide(p)) {
45         if(p->suiv == x)
46             return 1;
47
48         p = p->suiv;
49     }
50
51     return 0;
52 }
```

Listing 2.2 – TAD Pile

```
1 #ifndef __PILEDYNAMIQUE
2 #define __PILEDYNAMIQUE
3
4 typedef int Element;
5 typedef struct etCel {
6     Element value;
7     struct etCel* next;
8 } Cel;
```

```

9
10 typedef Cel* Pile;
11
12 Pile pile_init();
13 void pile_afficher(Pile pStack);
14 bool pile_estVide(const Pile pStack);
15 bool pile_pleine(const Pile pStack);
16 void pile_empiler(Pile* pStack, Element pElem);
17 Element pile_depiler(Pile* pStack);
18 Pile pile_saisir();
19 Element pile_sommet(Pile pStack);
20
21 #endif

```

Listing 2.3 – TAD Pile

## 2.3 File

Sorte File

Utilise Element, booleen

Constructeurs

**creer**  $\rightarrow$  File

**enfiler** File  $\times$  Element  $\rightarrow$  File

Projecteurs **estVide** file  $\rightarrow$  Booleen

**appartient** file  $\times$  Element  $\rightarrow$  Booleen

**defiler** file  $\rightarrow$  file

**premier** file  $\rightarrow$  Element

**dernier** file  $\rightarrow$  Element

Précondition

**premier** premier(f)  $\Leftrightarrow \neg$ estVide(f)

**dernier** dernier(f)  $\Leftrightarrow \neg$ estVide(f)

Axiones

```

estVide(creer()) = true;
estVide(enfiler(f,x)) = false;
appartient(creer(), x) = false;
appartient(enfiler(f,x),y) = (x = y)  $\vee$  appartient(f,y)
defiler(creer()) = creer()
defiler(enfiler(f,x) = creer() si estVide(f)
           = enfiler(defiler(f), x) sinon
premier(enfiler(f,x)) = premier(f) si !estVide
                       = x sinon
dernier(enfiler(f,x)) = x

```

Listing 2.4 – Axiones du TAD File

### 2.3.1 Statique

```

1 #include "element.h"
2
3 typedef struct etFile* File ;
4 File creer();

```

```
5 File enfile(File pFile, Element pElement);
6 File defiler(File pFile);
7 int appartient(File pFile, Elment pElement);
8 Element premier(File pFile);
9 Element dernier(File pFile);
10 int estPleine(File pFile);
```

Listing 2.5 – TAD File dynamique Headers

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <assert.h>
4 #include "file.h"
5
6 #define N 1000
7
8 struct eltFile {
9     Element Tab[N];
10    int nb;
11    int tete; // Buffer en rond
12 } FileInterne;
13
14 File creer() {
15     File f;
16     f = (File) malloc(sizeof(FileInterne));
17     assert(f != NULL);
18     f->nb = 0;
19     f->tete = 0;
20
21     return f;
22 }
23
24 File enfile(File pFile, Element pElement) {
25     assert(!estPleine(pFile));
26     pFile->tab[(f->n+f->tete)%N] = pElement;
27     ++f->nb;
28     return pFile;
29 }
30
31 int estPleine(File pFile) {
32     return (pFile->nb == N);
33 }
34
35 int estVide(File pFile) {
36     return (pFile->nb == 0);
37 }
38
39 File defiler(File pFile) {
40     if(!estVide(f)) {
41         pFile->nb--;
42         f->tete = (f->tete+1)%N;
43     }
44
45     return pFile;
46 }
47
48 int appartient(File pFile, Elment pElement) {
49     int i;
50     for( i=0 ; i < f->nb ; ++i ) {
51         if(x == f->tab[(i+f->tete)%N])
52             return 1;
53     }
54 }
```

```

55     return 0;
56
57 }
58
59 Element premier(File pFile) {
60     return (pFile->tab[pFile->tete]);
61 }
62
63 Element dernier(File pFile) {
64     return (pFile->tab[(f->tete+f->nb-1)%N]);
65
66 }

```

Listing 2.6 – TAD File dynamique Implémentation

## 2.3.2 Dynamique

```

1  #include "element.h"
2
3  typedef struct etFile* File;
4  File creer();
5  File enfile(File pFile, Element pElement);
6  File defiler(File pFile);
7  int appartient(File pFile, Element pElement);
8  Element premier(File pFile);
9  Element dernier(File pFile);
10 int estPleine(File pFile);

```

Listing 2.7 – TAD File dynamique Headers

```

1  typedef struct etCell {
2      struct etCell* suivant;
3      Element elem;
4  } Cell;
5
6  typedef struct etFile {
7      Cell* premier;
8      Cell* dernier;
9  } FileInterne;
10
11
12 File creer() {
13     File f;
14     f = (File) malloc(sizeof(FileInterne));
15     assert(f != NULL);
16     f->premier = NULL;
17     f->dernier = NULL;
18
19     return f;
20 }
21
22 File enfile(File pFile, Element pElement) {
23     Cell* c;
24     c = (Cell*) malloc(sizeof(Cell));
25     assert(c != NULL);
26     c->elem = pElement;
27     c->suivant = NULL;
28     if(!estVide(pFile)) {
29         pFile->dernier->suivant = c;
30     } else {
31         pFile->premier = c;

```

```
32     }
33     pFile->dernier = c;
34
35     return pFile;
36 }
37
38 int estPleine(File pFile) {
39     return (false);
40 }
41
42 int estVide(File pFile) {
43     return (pFile->premier == NULL);
44 }
45
46 File defiler(File pFile) {
47     if(!estVide(pFile)) {
48         File buff = pFile;
49         pFile->premier = pFile->suivant;
50         if(pFile->suivant == NULL) {
51             pFile->dernier = NULL
52         }
53
54         free(buff);
55     }
56
57     return pFile;
58 }
59
60 int appartient(File pFile, Elment pElement) {
61     Cell* courant;
62     courant = pFile->premier;
63     while(courant != NULL) {
64         if(pFile->premier->element == pElement) {
65             return 1;
66         }
67         courant = courant->suivant;
68     }
69
70     return 0;
71 }
72
73 Element premier(File pFile) {
74     return (pFile->premier->elem);
75 }
76
77 Element dernier(File pFile) {
78     return (pFile->dernier->elem);
79 }
80 }
```

Listing 2.8 – TAD File dynamique Implémentation

### 2.3.2.1 Application de la File à la fusion de voies routières

**Amélioration de la File en dynamique** Ecrire dans le module `File` (en dynamique) les deux fonctions suivantes :

- `concat : File × File → File`
- `mixe : File × File → File`



```

1 File concat(File f1, File f2) {
2     File retour = creer();
3     Cell* courant = f1->premier;
4     int i;
5     for(i=0 ; i < 2 ; ++i) {
6         while(courant != NULL) {
7             enfiler(retour, courant->elem);
8             courant = courant->suivant;
9         }
10        courant = f2->premier;
11    }
12
13    return retour;
14 }
15
16 File mixe(File f1, File f2) {
17     File fileRetour = creer();
18     Cell* courant1 = f1->premier;
19     Cell* courant2 = f2->premier;
20
21     while(courant1 != NULL || courant2 != NULL) {
22         if(courant1 != NULL) {
23             enfiler(fileRetour, courant1);
24             courant1 = courant1->suivant;
25         }
26         if(courant2 != NULL) {
27             enfiler(fileRetour, courant2);
28             courant2 = courant2->suivant;
29         }
30     }
31
32     return retour;
33 }

```

Listing 2.9 – TAD File dynamique Implémentation concat et mixe

## Écrire l'application

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "file.h"
4
5 int main(int argc, char** argv) {
6     File deParis;
7     File deGordeaux;
8     File versToulouse;
9
10    deParis = creer();
11    // Saisie file
12    deBordeaux = creer();
13    // Saisie file
14
15    // saisie de l'info manifestant
16
17    if(manifestation) {
18        versToulouse = concat(deParis, deBordeaux);
19    } else {
20        versToulouse = mixe(deParis, deBordeaux);
21    }
22    return 0;

```

## 2.4 File avec priorité

Ce sont des files dans lesquelles on place chaque élément au «bon endroit» (en remplaçant les priorités).

On va considérer qu'il existe dans le module **Element**, une fonction qui permet de comparer deux éléments entre eux.

```
/*
 * Renvoie 0 si e1 et e2 sont de même priorité
 *        -1 si e1 est moins prioritaire que e2
 *        1 si e1 est plus prioritaire que e2
 */
int compare(elem e1, elem e2);
```

Listing 2.11 – Fonction comparer

⚠ Le main de la fonction pourrait changer suivant les éléments

Réécrire enfiler en utilisant un pointeur de fonction pour accéder à la fonction de comparaison.

## 2.5 Liste avec priorité

Nous allons utiliser une liste doublement chaînée.

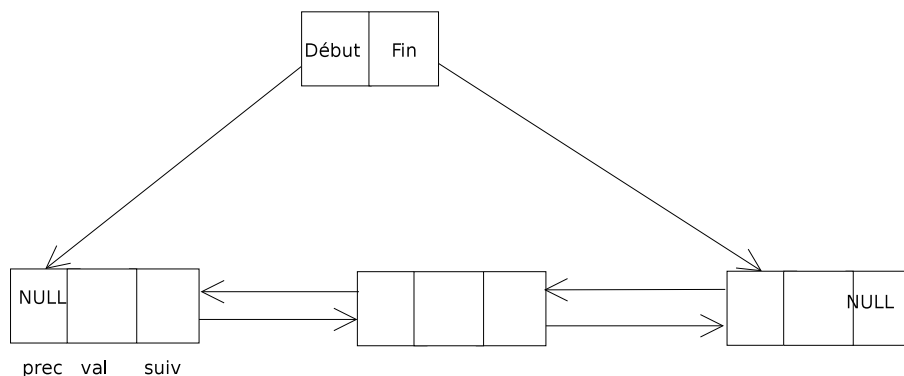


FIGURE 2.1 – Liste doublement chaînée

1. Proposer un type C pour la liste doublement chaînée
2. Écrire les méthodes suivantes :

```

LDC creer();
LDC ajouter(LDC, Elem);
void affichageCroissant(LDC);
void afficheDecroissant(LDC);
LDC supprimer(LDC, Elem);
/* Application de la fonction à chacun des éléments de la LDC et renvoie
 * la LDC des résultats
 */
LDC map(fonction, LDC)

```

```

1 typedef struct LDCInterne* LDC;
2
3 LDC creer(void);
4 LDC ajouter(LDC, Elem);
5 void affichageCroissant(LDC);
6 void afficheDecroissant(LDC);
7 LDC supprimer(LDC, Elem);
8 LDC map((Element* fc)(Element), LDC liste);

```

Listing 2.12 – Header liste doublement chaînée

```

1 typedef struct etCell {
2     struct etCell* prec;
3     struct etCell* suiv;
4     Elem val;
5 } Cell;
6
7 typedef struct LDCInterne {
8     Cell* premier;
9     Cell* dernier;
10 } LDCInterne;
11
12 /*
13  * Fonction interne
14  */
15 void trouverPlace(LDC l, Element e, Cell** prec, Cell** suiv) {
16     Cell* courant = l->premier;
17     courant = l->premier;
18     *prec = NULL;
19     *suiv = courant;
20
21     while(courant != NULL) {
22         if(compare(e, courant->val) == 1) {
23             *prec = courant;
24             *suiv = courant->suiv;
25             courant = courant->suiv;
26         } else {
27             courant = NULL;
28         }
29     }
30 }
31
32 LDC creer(void) {
33     LDC newLdc;
34     newLdc = (LDC) malloc(sizeof(LDCInterne));
35     newLdc->premier = NULL;
36     newLdc->dernier = NULL;
37
38     return newLdc;
39 }
40
41 LDC ajouter(LDC liste, Elem element) {

```

```
42 Cel* current;
43 Cel* avant;
44 Cel* apres;
45 courant = (Cell*) malloc(sizeof(Cell));
46 assert(courant != NULL);
47 courat-val = e;
48 trouverPlace(liste, element, &avant, &apres);
49 courant->prec = avant;
50 courant->suiv = apres;
51
52 if(avant == NULL) { //début de liste
53     liste->premier = courant;
54 } else {
55     liste->premier->suiv = courant;
56 }
57
58 if(apres == NULL) { //fin de liste
59     liste->dernier = courant;
60 } else {
61     liste->dernier->prec = courant;
62 }
63
64 return liste;
65 }
66
67
68 void affichageCroissant(LDC liste) {
69     Cell* courant = liste->premier;
70     while(courant != NULL) {
71         afficherElement(courant->val);
72         courant = courant->suiv;
73     }
74 }
75
76 void afficheDecroissant(LDC liste) {
77     Cell* courant = liste->dernier;
78     while(courant != NULL) {
79         afficherElement(courant->val);
80         courant = courant->prec;
81     }
82 }
83
84 LDC supprimer(LDC liste, Elem element) {
85     Cell* courant;
86     Cell* ajeter;
87
88     courant = liste->premier;
89     while(courant != NULL) {
90         if(compare(e, courant->val) == 0) {
91             // On a trouvé
92             if(courant->prec != NULL) {
93                 courant->prec->suiv = courant->suiv;
94             } else {
95                 l->premier = courat->suiv;
96             }
97
98             if(courant->suiv != NULL) {
99                 courant->suiv->prec = courant->prec;
100             } else {
101                 liste->dernier = courant->prec;
102             }
103         }
```

```
104     ajeter = courant;
105     courant = courant->suiv;
106     free(ajeter);
107 } else {
108     courant = courant->suiv;
109 }
110 }
111 return liste;
112 }
113
114 LDC map((Element* fc)(Element), LDC liste) {
115     Cell* courant;
116     courant = liste->premier;
117     LDC listeRes = creer();
118
119     while(courant != NULL) {
120         listeRes = ajouter(listeResultat, fc(courant->a1));
121         courant = courant->suiv;
122     }
123
124     return liste;
125 }
```

Listing 2.13 – Source lite doublement chaînée

---

# Cours sur les pointeurs en C

---

Déjà vu par le passages de paramètres.

## A.1 Syntaxe

### A.1.1 Déclaration

```
typePointé* nomPointeur
```

Listing A.1 – Syntaxe de déclaration d'un pointeur

```
| int n; // n correspond à un entier  
| int *ptr; // ptr correspond à l'adresse d'un entier
```

Listing A.2 – Exemple de déclaration

### A.1.2 Utilisation

```
nomPointeur // manipule l'adresse  
*nomPointeur //manipule la zone pointée
```

Listing A.3 – Syntaxe utilisation d'un pointeur

```
pe=&n; //opérateur d'adressage
```

Listing A.4 – Exemple d'utilisation d'un pointeur

### A.1.3 Constante

NULL représente une adresse inexistante.

```
pe = NULL;  
*pe; // Erreur à l'exécution
```

Listing A.5 – Exemple d'utilisation de la constante NULL

## A.2 Opérateur autorisés sur les pointeurs

### A.2.1 L'affectation

```
nomPointeur = expression correspondant à une adresse ou à NULL
```

## A.2.2 Addition et la soustraction entre un pointeur et un entier

```
nomPointeur = nomPointeur + 10;
nomPointeur = nomPointeur - 15;
```

On obtient une expression correspondant à une adresse

```
pe = pe+10; //pe contient l'adresse du 10e entier après la valeur initiale de pe.
```

 À utiliser que si pe pointe sur un tableau

## A.2.3 Soustraction de deux pointeurs

Renvoie un entier donnant le nombre d'éléments pointés entre les deux pointeurs

 Uniquement si les deux pointeurs sont sur le même tableau

## A.2.4 Comparaison sur des pointeurs

Ce sont les opérateurs de comparaison classique : `=` et `!=`

## A.2.5 Allocation dynamique de mémoire

```
nomPointeur = (typePointeur) malloc(sizeof(typePointé));
nomPointeur = (typePointé*) malloc(n*sizeof(typePointé));
```

Listing A.6 – Syntaxe d'allocation dynamique

```
int *e;
pe = (int*) malloc(sizeof(int));
```

Listing A.7 – Exemple d'allocation dynamique

1. Le programme demande au gestionnaire mémoire d'avoir une place de la taille `sizeof(int)`
2. Si la place est disponible retourne l'adresse demandée ou la première case du «tableau» dynamique
3. Sinon retourne NULL


## A.2.6 Libération dynamique de mémoire

```
free(nomPointeur);
```

Listing A.8 – Syntaxe de libération de mémoire

1. Le programme contact le gestionnaire mémoire
2. Le gestionnaire mémoire «libère» la place

Cela veut dire que la place n'est plus réservé au programme, elle pourra être alloué à un autre programme.

 Le gestionnaire de mémoire ne met pas à jour la case mémoire, celle-ci contient toujours la valeur, si personne ne récupère la case, il sera toujours possible d'accéder à la donnée. C'est donc aléatoire, c'est une source d'erreurs.

## A.3 Pointeur sur fonction

Un pointeur de fonction est un pointeur qui contient l'adresse d'une fonction.

### A.3.1 Syntaxe

```
| typedef retour (*nomPtrFonction)(listeDesParametres type1 p1, type2 p2, type3 p3);
```

Listing A.9 – Déclaration d'un pointeur de fonction

### A.3.2 Utilisation

```
| nomPtrFonction (listeDesArguments);
```

Listing A.10 – Utilisation d'un pointeur de fonction

### A.3.3 Exemple

```
/* Module 1 */
int fctTest(int(*f)(int), int p) {
    return f(p);
}

/* Module 2 */
#include "module1.h"
int toto(int a) {
    return a*a;
}
int main(void) {
    int res = fctTest(&toto, 10);
}
```

Listing A.11 – Exemple d'utilisation d'un pointeur de fonction



## Liste des codes sources

1.1	Opérations du TAD Pile . . . . .	6
1.2	Implémentation des fonctions du type Pile en statique . . . . .	7
1.3	Implémentation des fonctions du type Pile en dynamique . . . . .	8
1.4	Type de la pile statique originel – Présent dans le .h . . . . .	9
1.5	Type de la pile statique – Présent dans le .h . . . . .	10
1.6	Type de la pile statique – Présent dans le .c . . . . .	10
1.7	Modification de la fonction <code>creer</code> s’adaptant à la protection de données . . . . .	10
2.1	TAD Pile . . . . .	11
2.2	TAD Pile . . . . .	12
2.3	TAD Pile . . . . .	12
2.4	Axiones du TAD File . . . . .	13
2.5	TAD File dynamique Headers . . . . .	13
2.6	TAD File dynamique Implémentation . . . . .	14
2.7	TAD File dynamique Headers . . . . .	15
2.8	TAD File dynamique Implémentation . . . . .	15
2.9	TAD File dynamique Implémentation <code>concat</code> et <code>mixe</code> . . . . .	16
2.10	Application de fusion de voies routières . . . . .	17
2.11	Fonction comparer . . . . .	18
2.12	Header liste doublement chaînée . . . . .	19
2.13	Source lite doublement chaînée . . . . .	19
A.1	Syntaxe de déclaration d’un pointeur . . . . .	22
A.2	Exemple de déclaration . . . . .	22
A.3	Syntaxe utilisation d’un pointeur . . . . .	22
A.4	Exemple d’utilisation d’un pointeur . . . . .	22
A.5	Exemple d’utilisation de la constante <code>NULL</code> . . . . .	22
A.6	Syntaxe d’allocation dynamique . . . . .	23
A.7	Exemple d’allocation dynamique . . . . .	23
A.8	Syntaxe de libération de mémoire . . . . .	24
A.9	Déclaration d’un pointeur de fonction . . . . .	24
A.10	Utilisation d’un pointeur de fonction . . . . .	24
A.11	Exemple d’utilisation d’un pointeur de fonction . . . . .	24
D.1	TAD Pile . . . . .	27
D.2	Pointeurs – Exercice 1 . . . . .	27
D.3	Pointeurs – Exercice 2 . . . . .	28
D.4	Pointeurs – Exercice 3 . . . . .	28
D.5	pointeurs – Exercice 4 . . . . .	29

---

## Table des figures

---

1.1	Principe de base d'un TAD . . . . .	5
2.1	Liste doublement chaînée . . . . .	18

---

# Exercices

---

## D.1 TAD

### D.1.1 Suite du TAD Pile

```
1 Pile remplacer(Pile pPile, Element pX, Element pY) {
2     int i;
3     for(i=0 ; i < p.nb ; ++i) {
4         if(p.tab[i] == x) {
5             p.tab[i] = y;
6         }
7     }
8
9     return p;
10 }
11
12 void afficherPile(Pile pPile) {
13     int i;
14     for(i=0 ; i < p.nb; ++i) {
15         afficheElement(p.tab[i]);
16     }
17 }
```

Listing D.1 – TAD Pile

## D.2 Pointeurs

### D.2.1 Exercice 1

```
1 int *p, *q; // 1
2 p = NULL; // 2
3 q = p; //3
4 p = (int*)(malloc(sizeof(int))); // 4
5 q = p; // 5
6 q = (int*)malloc(sizeof(int)) // 6
7 free(p);
8 *q = 10;
```

Listing D.2 – Pointeurs – Exercice 1

1		2		3		4		5		6		7		8	
										@2		@2		@2	10
p		p	NULL	p	NULL	p	@1	p	@1	p	@1	p	@1	p	@1
q		q		q	NULL	q	NULL	q	@2	q	@2	q	@2	q	@2
						@1						@1		@1	

## D.2.2 Exercice 2

```

1 typedef int Zone;
2 typedef Zone *Ptr;
3
4 void miseAJour(Ptr p, Zone v) {
5     *p = v;
6 }
7
8 int main(void) {
9     Ptr p; // 1
10    p = (Ptr) malloc(sizeof(Zone)); //2
11
12    if(p != NULL)
13        miseAJour(p, 10); // 3
14 }

```

Listing D.3 – Pointeurs – Exercice 2

1		2 malloc OK		2 malloc non OK		3 malloc OK		3 malloc non OK	
p		p	@1	p	NULL	p	@1	p	NULL
		@1				@1	10		

**R** Dans le du malloc qui ne marche pas, ce que contient la mémoire est inconnu, si on accède à \*p nous aurons une segmentation fault. Ainsi on rajoute un test

## D.2.3 Exercice 3

```

1 typedef struct etCell {
2     int val;
3     int* suiv;
4 } Cel
5
6 typedef Cel* Ptr;
7
8 int main(void) {
9     Cel c; //1
10    c.val = 10; //2
11    c.suiv = (int*) malloc(sizeof(int)); //3
12    *(c.suiv) = 11; //4
13 }

```

Listing D.4 – Pointeurs – Exercice 3

1	2	3	4
c.val	c.val 10	c.val 10	c.val 10
c.suiv	c.suiv	c.suiv @1	c.suiv @1
		@1	@1 11

## D.2.4 Exercice 4 – Même exercice avec une autre valeur

```

1 typedef struct etCell {
2     int val;
3     struct etCell* suiv;
4 } Cel
5
6 typedef Cel* Ptr;
7
8 int main(void) {
9     Cel c; //1
10    c.val = 10; //2
11    c.suiv = (Ptr) malloc(sizeof(Cel)); //3
12    (*(c.suiv)).val = 11;
13    (*(c.suiv)).suiv = (Ptr) malloc(sizeof(Cel));
14    c.suiv->suiv->val = 12; // Ou ((*(*c.suiv)).suiv).val = 12;
15 }
```

Listing D.5 – pointeurs – Exercice 4