

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

1	Types de données Abstraits (TAD)	6
1.1	Syntaxe des TAD	6
1.2	Implémentation d'un TAD	7
1.3	Protection du TAD	8
2	Structures de données classiques	10
2.1	Pile	10
2.2	File	14
2.3	File avec priorité	19
2.4	Liste avec priorité	20
3	Parcourir une collection	24
3.1	Itérateur sur la liste doublement chaînée	24
4	Les structures arborescentes	27
4.1	L'arbre GRD : «Gauche Racine Droite»	27
4.2	Les arbres rouges noirs	31
A	Cours sur les pointeurs en C	32
A.1	Syntaxe	32
A.2	Opérateur autorisés sur les pointeurs	33
A.3	Pointeur sur fonction	34
B	Liste des codes sources	36
C	Table des figures	37

D Exercices	38
D.1 Pointeurs	38

Types de données Abstraits (TAD)

Sommaire

1.1	Syntaxe des TAD	6
1.2	Implémentation d'un TAD	7
1.3	Protection du TAD	8

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 programme¹
- Les données²

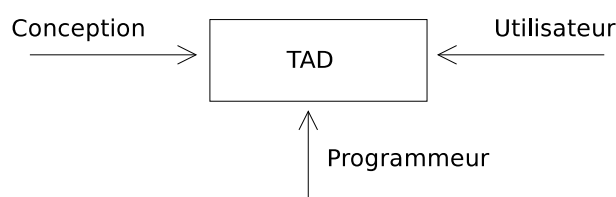


FIGURE 1.1 – Principe de base d'un TAD

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 TAD³ Donner les interfaces de la données

La sémantique abstraite du TAD⁴ 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 logiques 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.⁵

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 dynamique⁶.

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

1.3 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.

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


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

Listing 1.2 – 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 1.3 – Type de la pile statique – Présent dans le .h

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

Listing 1.4 – 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 1.5 – 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

Sommaire

2.1	Pile	10
2.2	File	14
2.3	File avec priorité	19
2.4	Liste avec priorité	20

2.1 Pile

```

1 typedef struct etPile* Pile ;
2
3 Pile creer();
4 Pile empiler(Pile p, Element x);
5 int estVide(Pile p);
6 Pile depiler(Pile p);
7 Element sommet(Pile p);
8 int appartient(Pile p, Element x);

```

Listing 2.1 – Pile sans protection du type – Header

2.1.1 Statique

2.1.1.1 Sans protection du type

- 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]);
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 2.2 – Pile statique sans protection du type – Implémentation

Amélioration de la Pile.

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.

Rajouter dans le champ des opérations $\text{remplacerOccurence Pile} \times \text{Element} \times \text{Element} \rightarrow \text{Pile}$

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 2.3 – Pile statique – Ajout de `remplacerOccurence`

2.1.1.2 Avec protection du type

```
1 #define N 1000
2
3 struct etPile {
4     Element Tab[N];
5     int nb;
6 } PileInterne;
7
8 Pile creer() {
9     Pile p ;
10    p = (Pile) malloc(sizeof(PileInterne));
11    p->nb = 0;
12
13    return p ;
14 }
15
16 Pile empiler(Pile p, Element x) {
17     assert(p->nb < N); // Si la condition est false alors arrête programme
18     p->tab[p->nb] = x;
19     p->nb++;
20
21     return (p);
22 }
23
24 int estVide(Pile p) {
25     return (p->nb == 0);
26 }
27
28 Pile depiler(Pile p) {
29     if(!estVide(p)) {
30         p->nb--;
31     }
32
33     return p;
34 }
35
36 Element sommet(Pile p) {
37     assert(!estVide(p)); // Pas indispensable masi plus robuste
38     return (p->tab[p->nb-1]);
39 }
40
41 int appartient(Pile p, Element x) {
42     if(estVide(p))
43         return 0;
44
45     if(x == sommet(p)) {
46         return 1;
47     }
48 }
```

```

49 | return (appartient(depiler(p), x));
50 | }

```

Listing 2.4 – Pile statique avec protection du type – Implémentation

2.1.2 Dynamique

Une pile dynamique peut être implémentée de différentes façons, la liste simplement chaînée et la liste doublement chaînée.

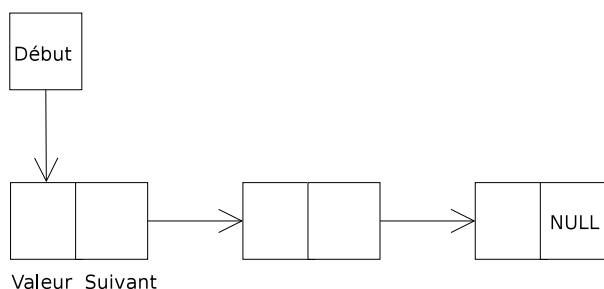


FIGURE 2.1 – Pile avec une liste simplement chaînée

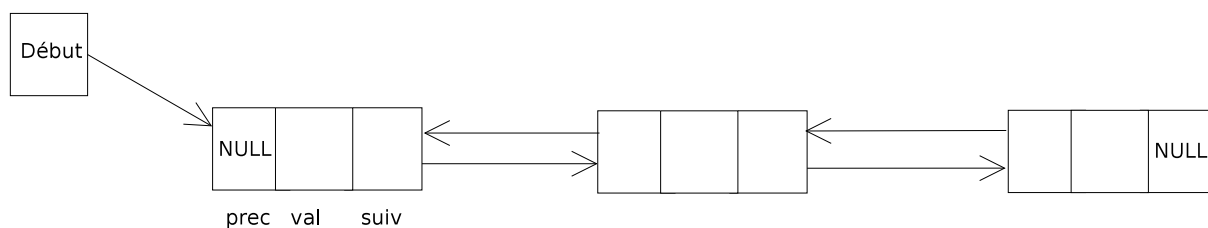


FIGURE 2.2 – Pile avec une liste doublement chaînée

Nous avons choisi de l'implémenter avec une liste simple chaînée, le double chaînage n'étant pas utile pour ce que nous souhaitons faire.

```

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 2.5 – Pile dynamique – Implémentation

2.2 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 2.6 – File – Axiones

```

1 #include "element.h"
2
3 typedef struct etFile* File ;
4 File creer();
5 File enfiler(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 2.7 – File – Headers

2.2.1 Statique

```

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 enfiler(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 2.8 – File statique – Implémentation

2.2.2 Dynamique

De la même manière que la pile, en dynamique la **File** peut être implémentée avec une liste simplement chaînée ou une liste doublement chaînée.

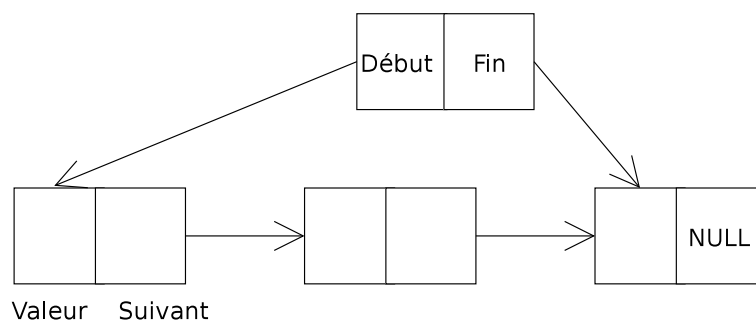


FIGURE 2.3 – File avec une liste simplement chaînée

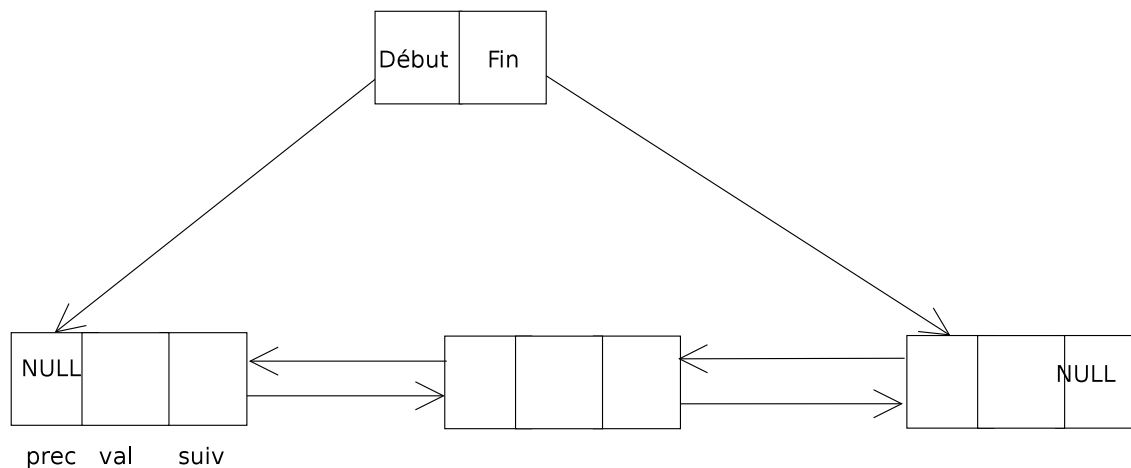


FIGURE 2.4 – File avec une liste doublement chaînée

Nous avons choisis la liste simplement chaînée.

```

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 2.9 – File dynamique – Implémentation

2.2.2.1 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 : $\text{File} \times \text{File} \rightarrow \text{File}$
- mixe : $\text{File} \times \text{File} \rightarrow \text{File}$

```
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 2.10 – File dynamique – Ajout 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;
23 }

```

Listing 2.11 – File – Application de fusion de voies routières

2.3 File avec priorité

Ce sont des files dans lesquelles on place chaque élément au «bon endroit» (en remplaçant les priorités), en l'occurrence, les éléments doivent être triés par **Element**.

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 2.12 – Element – Prototype `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.

2.4 Liste avec priorité

Nous allons utiliser une liste doublement chaînée.

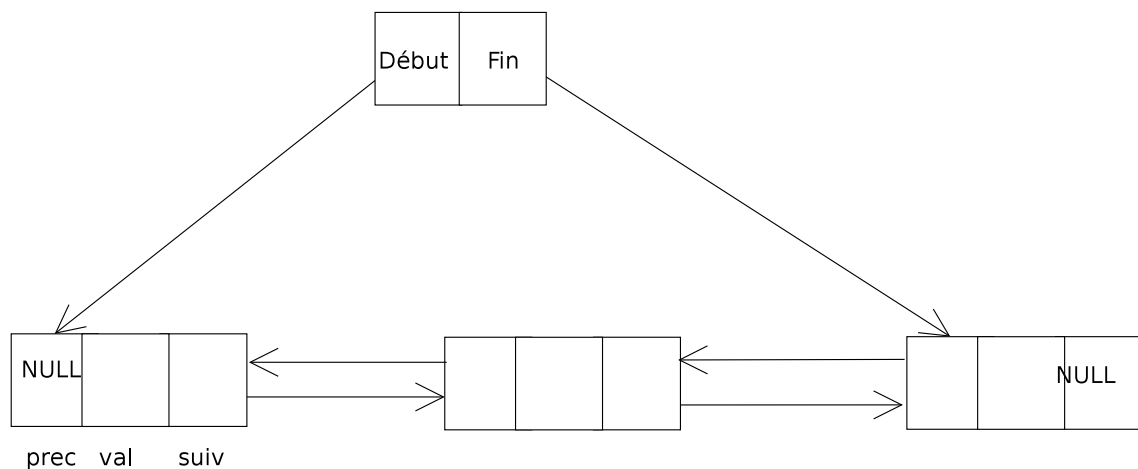


FIGURE 2.5 – Liste doublement chaînée

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 2.13 – Liste doublement chaînée – Header

```

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 affichageCroisant(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 2.14 – Liste doublement chaînée – Implémentation

Parcourir une collection

Un itérateur est une structure de données qui permet de parcourir une collection d'objets.

Ex

```
for(i=N-1; i >= 0; ++i);
```

Ici *i* joue le rôle d'un itérateur, il permet de parcourir une collection de *N* éléments du début à la fin ou de la fin au début.

Un itérateur est lié à une collection, on peut

- se placer en début/fin de la collection
- Passer à l'élément suivant/précédent de la collection
- savoir quand on est arrivé à la fin/début de la collection

En utilisant un langage pseudo objet, cela nous donnerai l'algorithme suivant :

```
i = creerIterateur(c); //c étant la collection
for(i = debut(i); !videSuivant(i) ; i = suivant(i)) {
    //...
}
```

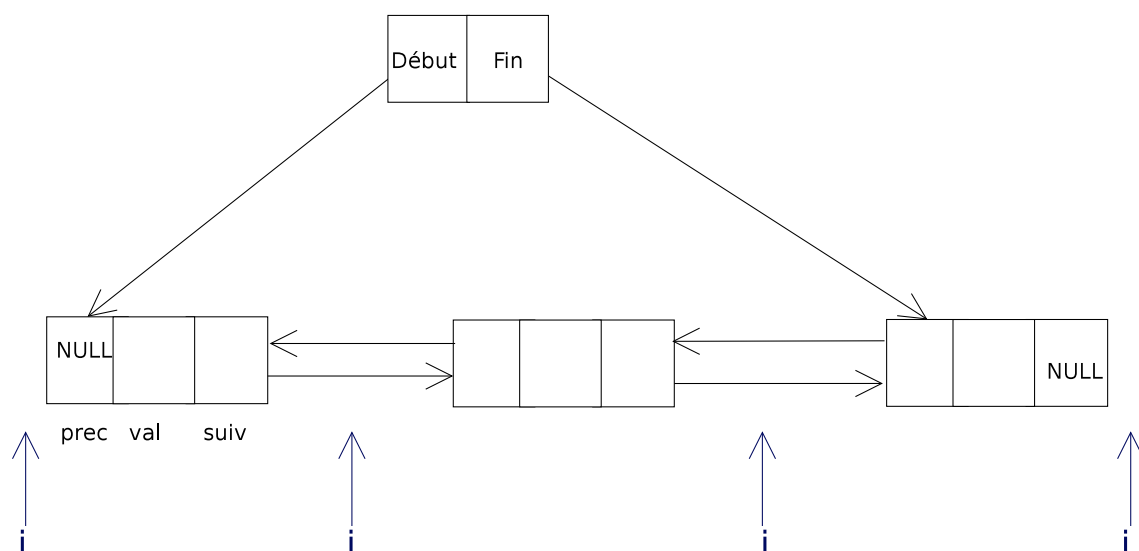


FIGURE 3.1 – Liste doublement chaînée avec itérateur

3.1 Itérateur sur la liste doublement chaînée

Création d'un itérateur sur liste doublement chaînée. Écrire les fonctions suivantes :

creerIte Création d'un itérateur sur une LDC et place l'itérateur en début de liste
next Déplace l'itérateur sur le suivant renvoie la valeur courant d'avant le déplacement
previous Déplace l'itérateur sur le précédent et renvoie la valeur d'avant le courant
hasNext, hasPrevious Renvoie 1 si l'élément suivant/précédent existe
begin, end Place l'itérateur sur le début/fin de la liste.

```

1 typedef struct etIte* Iterateur;
2
3 Iterateur creerIterateur(LDC);
4 Element next(Iterateur iterateur);
5 Element previous(Iterateur iterateur);
6 void begin(Iterateur iterateur);
7 void end(Iterateur iterateur);
8 int hasNext(Iterateur iterateur);
9 int hasPrevious(Iterateur iterateur);

```

Listing 3.1 – Itérateur sur Liste – Header

```

1 #include "iterateur.h"
2
3 typedef struct etIte {
4     LDC l;
5     Cell* cour;
6 } etIterateur;
7
8
9 Iterateur creerIterateur(LDC liste) {
10     Iterateur ite;
11     ite = (Iterateur) malloc(sizeof(etIterateur));
12     assert(ite != NULL);
13     ite->l = liste;
14     ite->cour = liste->debut;
15
16     return ite;
17 }
18
19 Element next(Iterateur iterateur) {
20     Element ret = iterateur->cour->val;
21     assert(hasNext);
22     iterateur = iterateur->cour->suivant;
23
24     return ret;
25 }
26
27 Element previous(Iterateur iterateur) {
28     Element ret;
29     assert(hasPrevious);
30     if(iterateur->cour != NULL)
31         iterateur->cour = iterateur->cour->precedent;
32     else
33         iterateur->cour = iterateur->l->fin;
34
35     ret = iterateur->cour->val;
36
37     return ret;
38 }
39 void begin(Iterateur iterateur) {
40     iterateur->cour = iterateur->l->debut;
41 }
42 void end(Iterateur iterateur) {
43     iterateur->cour = NULL;
44 }

```

```
45 | int hasNext(Iterateur itereur) {  
46 |     return (itereur->cour != NULL);  
47 | }  
48 | int hasPrevious(Iterateur itereur) {  
49 |     return (itereur->cour != itereur->l->debut);  
50 | }
```

Listing 3.2 – Iterateur sur Liste – Implémentation

Les structures arborescentes


Sommaire

4.1 L'arbre GRD : «Gauche Racine Droite»	27
4.2 Les arbres rouges noirs	31

Nous allons voir deux types d'arbres :

1. L'arbre GRD : « Gauche Racine Droite »
2. Les arbres rouges noirs

Ce sont des arbres binaires : chaque noeud de l'arbre à au lu deux fils.

 Les informations sont rangés dans l'arbre en respectant un certain critère

4.1 L'arbre GRD : «Gauche Racine Droite»

4.1.1 Critère de rangement

Quelque soit le noeud de l'arbre :

- les informations rangées à gauche de la racine de ce noeud sont inférieur ou égal à cette racine.
- les informations rangées à droite de la racine de ce noeud sont supérieur à cette racine.

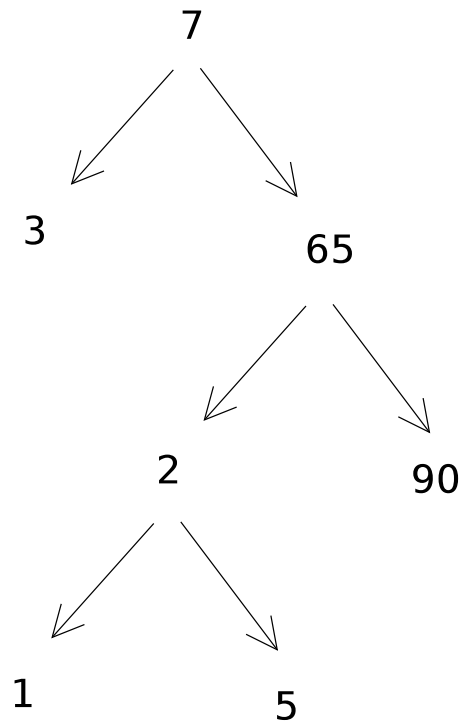


FIGURE 4.1 – Arbre GRB

Cet arbre est prévu pour effectuer un parcours en profondeur.

4.1.2 Implémentation du TAD

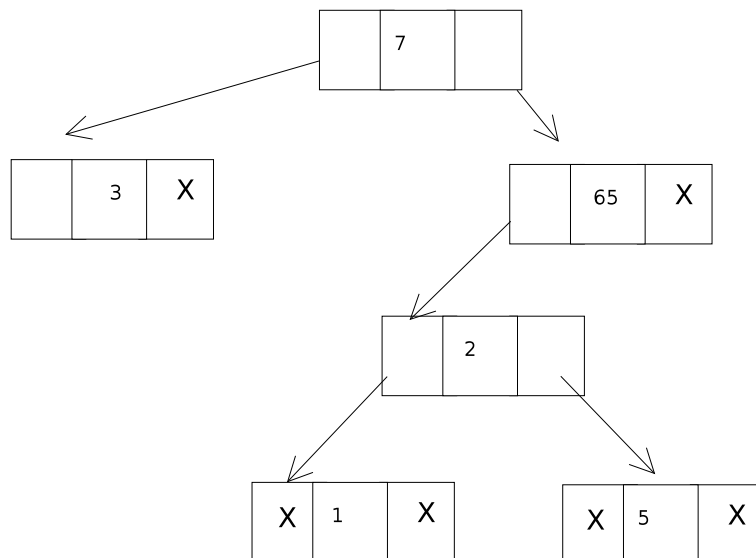


FIGURE 4.2 – Implémentation de l'arbre GRB

```

1 typedef struct etCell* Arbre;
2
3 Arbre creerGrd(void);
4 Arbre ajout(const Arbre arbre, int e);
5 void afficher(const Arbre arbre);
6 Arbre afficherIteratif(Arbre pArbre);
  
```

```

7 | int estVide(const Arbre arbre);
8 | int appartient(const Arbre arbre, int racine);

```

Listing 4.1 – Arbre GRD – Header

```

1 | typedef struct etCell {
2 |     struct etCell* gauche;
3 |     struct etCell* droite;
4 |     int racine;
5 | } Cell;
6 |
7 | Arbre creerGrd(void) {
8 |     return (NULL);
9 | }
10 |
11 | Arbre ajout(const Arbre arbre, int e) {
12 |     Cell* newCell;
13 |     newCell = (Cell*)malloc(sizeof(Cell));
14 |     newCell->racine = e;
15 |     newCell->gauche = NULL;
16 |     newCell->gauche = NULL;
17 |     if(estVide(arbre)) {
18 |         arbre = newCell;
19 |     } else {
20 |         if(e <= arbre->racine) {
21 |             arbre->gauche = ajout(arbre->gauche, e);
22 |         } else {
23 |             arbre->droite = ajout(arbre->droite, e);
24 |         }
25 |     }
26 |
27 |     return arbre;
28 | }
29 |
30 | // Parcours infixe
31 | void afficher(const Arbre arbre) {
32 |     Cell* cous = a;
33 |     if(!estVide(a)) {
34 |         affiche(a->gauche);
35 |         printf("%d", a->racine);
36 |         afficher(a->droite);
37 |     }
38 | }
39 |
40 | int estVide(const Arbre arbre) {
41 |     return (arbre != NULL);
42 | }
43 |
44 | int appartient(const Arbre arbre, int val) {
45 |     if(estVide(arbre))
46 |         return 0;
47 |     if(val <= arbre->racine)
48 |         return appartient(arbre->gauche, val);
49 |     if(val > arbre->racine)
50 |         return appartient(arbre->droite, val);
51 |
52 |     return 1;
53 | }
54 | // Private
55 | Arbre restructure(Arbre pArbre) {
56 |     // on est sur le noeud à créer
57 |     Arbre droit, gauche, aux;
58 |     droit = pArbre->droit;

```

```
59 gauche = pArbre->gauche;
60
61 if(droit == NULL) {
62     // on a rien à insérer
63     return gauche;
64 }
65 aux = droit;
66 while(aux->gauche != NULL) {
67     aux = aux->gauche;
68 }
69 aux->gauche = gauche;
70 free(pArbre);
71
72 return droit;
73 }
74 // Private
75 Arbre supprimerUnElement(Arbre pArbre, int val) {
76     assert(!estVide(pArbre));
77     if(pArbre->racine == val) {
78         return(restructure(pArbre, val));
79     }
80     if((pArbre->racine < val) && (pArbre->gauche != NULL)) {
81         pArbre->droit = supprimerUnElement(a->droit, val);
82     } else if((a->racine > val) && (a->gauche != NULL)) {
83         pArbre->gauche = supprimerUnElement(a->gauche, val);
84     }
85
86     return pArbre;
87 }
88
89 Arbre supprime(Arbre pArbre, int v) {
90     while(appartient(pArbre, v)) {
91         pArbre = supprimerUnElement(pArbre, v);
92     }
93     return pArbre;
94 }
```

Listing 4.2 – Arbre GRD – Implémentation

4.1.3 Différents types de parcours

Parcours infixe On parcourt à gauche, on appelle la valeur, on parcourt à droite.

Parcours préfixe On parcourt on appelle la valeur puis on parcourt à gauche et à droite.

Parcours postfixe On parcourt à droite puis à gauche et ensuite on appelle la valeur.

4.1.3.1 Exercices de parcours

Écrire une fonction qui permette l’affichage en profondeur d’un arbre GRD mais sans utiliser la récursivité.

Nous avons utilisé une Pile afin de simuler des appels récursifs (Pile système). La pile contient le nœud courant.

On suppose que l’on dispose du TAD Pile d’Element avec le type élément qui est une Cell.

```
1 Arbre afficherIteratif(Arbre pArbre) {
2     Cell n;
3     Arbre aux;
```

```

4  if(!estVide(pArbre)) {
5      Pile p = creer();
6      p = empiler(p, *pArbre);
7      while(!pileEstVide(p)) {
8          n = sommetPile(p);
9          p = depiler(p);
10         if((n.gauche == NULL) && (n.droite == NULL)) {
11             printf("%d ", n.racine);
12         } else {
13             if(n.droit != NULL) {
14                 p = empiler(p, *(n.droit));
15             }
16             aux = n.gauche;
17             n.droit = NULL;
18             n.gauche = NULL;
19             p = empiler(p,n);
20             if(aux != NULL) {
21                 p = empiler(p, *aux);
22             }
23         }
24     }
25 }
26 }

```

Listing 4.3 – Arbre GRD – Implémentation fonction affichage en profondeur itératif

Pour parcourir l'arbre en largeur, le principe est le même, à la place d'utiliser une `Pile` nous allons utiliser une `File`.

```

1  Arbre afficherLargeurIteratif(Arbre pArbre) {
2      Cell n;
3      if(!estVide(pArbre)) {
4          File p = creer();
5          p = enfiler(p, *pArbre);
6          while(!fileEstVide(p)) {
7              n = sommetFile(p);
8              p = defiler(p);
9              printf("%d ", n.racine);
10             if(n.gauche != NULL) {
11                 p = enfiler(p, *(n.gauche));
12             }
13             p = enfiler(p,n);
14             if(n.droit != NULL) {
15                 p = enfiler(p, n.droit);
16             }
17         }
18     }
19 }

```

Listing 4.4 – Arbre GRD – Implémentation fonction affichage en longueur

4.2 Les arbres rouges noirs

Cours sur les pointeurs en C

Déjà vu par le passages de paramètres, nous allons donc ici commencer par un rappel, et nous verrons l'allocation dynamique de mémoire.

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	7
1.2	Type de la pile statique originel – Présent dans le .h	9
1.3	Type de la pile statique – Présent dans le .h	9
1.4	Type de la pile statique – Présent dans le .c	9
1.5	Modification de la fonction <code>creer</code> s’adaptant à la protection de données	9
2.1	Pile sans protection du type – Header	10
2.2	Pile statique sans protection du type – Implémentation	10
2.3	Pile statique – Ajout de <code>remplacerOccurence</code>	11
2.4	Pile statique avec protection du type – Implémentation	12
2.5	Pile dynamique – Implémentation	13
2.6	File – Axiones	14
2.7	File – Headers	15
2.8	File statique – Implémentation	15
2.9	File dynamique – Implémentation	17
2.10	File dynamique – Ajout <code>concat</code> et <code>mixe</code>	18
2.11	File – Application de fusion de voies routières	19
2.12	Element – Prototype <code>comparer</code>	20
2.13	Liste doublement chaînée – Header	21
2.14	Liste doublement chaînée – Implémentation	21
3.1	Iterateur sur Liste – Header	25
3.2	Iterateur sur Liste – Implémentation	25
4.1	Arbre GRD – Header	28
4.2	Arbre GRD – Implémentation	29
4.3	Arbre GRD – Implémentation fonction affichage en profondeur itératif	30
4.4	Arbre GRD – Implémentation fonction affichage en longueur	31
A.1	Syntaxe de déclaration d’un pointeur	32
A.2	Exemple de déclaration	32
A.3	Syntaxe utilisation d’un pointeur	32
A.4	Exemple d’utilisation d’un pointeur	32
A.5	Exemple d’utilisation de la constante <code>NULL</code>	32
A.6	Syntaxe d’allocation dynamique	33
A.7	Exemple d’allocation dynamique	33
A.8	Syntaxe de libération de mémoire	34
A.9	Déclaration d’un pointeur de fonction	34
A.10	Utilisation d’un pointeur de fonction	34
A.11	Exemple d’utilisation d’un pointeur de fonction	34
D.1	Pointeurs – Exercice 1	38
D.2	Pointeurs – Exercice 2	38
D.3	Pointeurs – Exercice 3	39
D.4	pointeurs – Exercice 4	39

Table des figures

1.1	Principe de base d'un TAD	6
2.1	Pile avec une liste simplement chaînée	13
2.2	Pile avec une liste doublement chaînée	13
2.3	File avec une liste simplement chaînée	16
2.4	File avec une liste doublement chaînée	17
2.5	Liste doublement chaînée	20
3.1	Liste doublement chaînée avec iterateur	24
4.1	Arbre GRB	28
4.2	Implémentation de l'arbre GRB	28

Exercices

D.1 Pointeurs

D.1.1 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.1 – 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.1.2 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.2 – 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.1.3 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.3 – Pointeurs – Exercice 3

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

D.1.4 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.4 – pointeurs – Exercice 4