

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)	5
1.1	Syntaxe des TAD	5
1.2	Implémentation d'un TAD	6
1.3	Protection du TAD	9
A	Cours sur les pointeurs en C	11
A.1	Syntaxe	11
A.2	Opérateur autorisés sur les pointeurs	11
B	Exemple de structure de données linéaires dynamiques	14
B.1	Pile	14
B.2	File	14
B.3	Pile avec liste doublement chaînée	15
B.4	File avec liste doublement chaîné	15
C	Liste des codes sources	16
D	Table des figures	17
E	Exercices	18
E.1	TAD	18
E.2	Pointeurs	19

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 ¹
- Les données ²

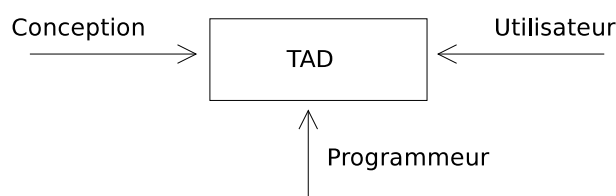


FIGURE 1.1 – Principe de base d'un TAD

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 ³ Donner les interfaces de la données

La sémantique abstraite du TAD ⁴ 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.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

1.2.1.1 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.

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.

Exemple de structure de données linéaires dynamiques

B.1 Pile

Liste simplement chaînée dynamique à un point d'entrée

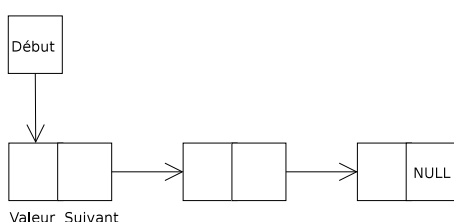


FIGURE B.1 – Pile avec une liste simplement chaînée

```
typedef struct etCel {
    Element val;
    struct etCel *suiv;
} CelSc;
```

Listing B.1 – Structure `CelSc` – Pile avec liste simplement chaînée

B.2 File

Liste simplement chaînée à deux points d'entrée

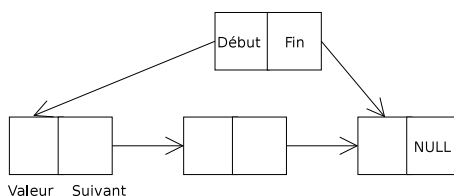


FIGURE B.2 – File avec une liste simplement chaînée

```
typedef struct etCel2 {
    LSC fin;
    LSC debut;
} LSC2;
```

Listing B.2 – Structure `LSC2` – File avec liste simplement chaînée

B.3 Pile avec liste doublement chaînée

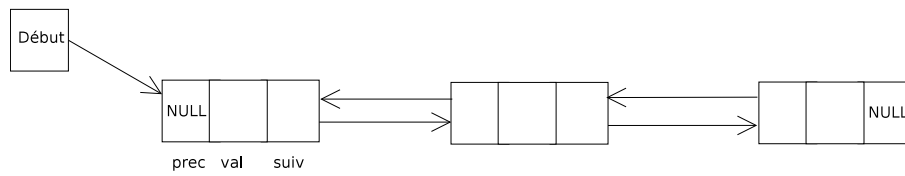


FIGURE B.3 – Pile avec une liste doublement chaînée

```
typedef struct etCelDC {
    Element val;
    struct etCelDC* suiv;
    struct etCelDC* precedent;
} CelDC;
typedef celDC* LDC;
```

Listing B.3 – Structure CelDC – Pile avec liste doublement chaînée

B.4 File avec liste doublement chaîné

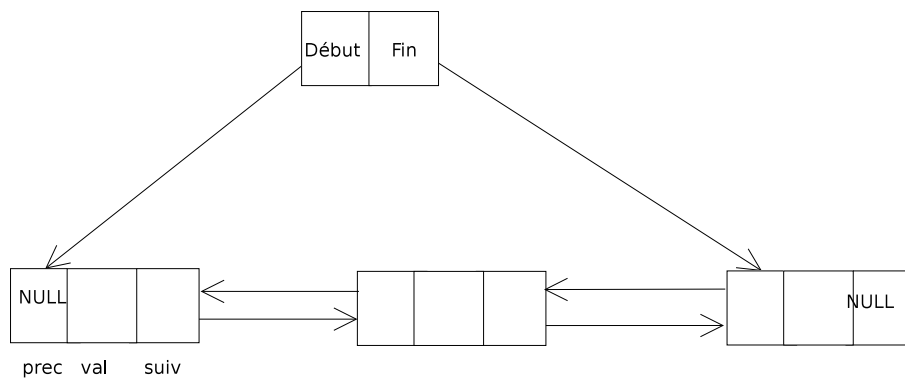


FIGURE B.4 – File avec une liste doublement chaînée

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
A.1	Syntaxe de déclaration d'un pointeur	11
A.2	Exemple de déclaration	11
A.3	Syntaxe utilisation d'un pointeur	11
A.4	Exemple d'utilisation d'un pointeur	11
A.5	Exemple d'utilisation de la constante <code>NULL</code>	11
A.6	Syntaxe d'allocation dynamique	12
A.7	Exemple d'allocation dynamique	12
A.8	Syntaxe de libération de mémoire	13
B.1	Structure <code>Ce1Sc</code> – Pile avec liste simplement chaînée	14
B.2	Structure <code>LSC2</code> – File avec liste simplement chaînée	14
B.3	Structure <code>Ce1DC</code> – Pile avec liste doublement chaînée	15
E.1	TAD Pile	18
E.2	Pointeurs – Exercice 1	19
E.3	Pointeurs – Exercice 2	19
E.4	Pointeurs – Exercice 3	20
E.5	pointeurs – Exercice 4	20

Table des figures

1.1	Principe de base d'un TAD	5
B.1	Pile avec une liste simplement chaînée	14
B.2	File avec une liste simplement chaînée	14
B.3	Pile avec une liste doublement chaînée	15
B.4	File avec une liste doublement chaînée	15

Exercices

E.1 TAD

E.1.1 Suite du TAD 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 `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 E.1 – TAD Pile

E.1.2 TAD File

Sorte File

Utilise Element, boolean

Constructeurs

`creer` \rightarrow File`enfiler` File \times Element \rightarrow FileProjecteurs `estVide` file \rightarrow Booleen`appartient` file \times Element \rightarrow Booleen`defiler` file \rightarrow file`premier` file \rightarrow Element`dernier` file \rightarrow Element**Précondition** `premier(f) <=> non est Vide(f) dernier(f) <=> non est Vide(f)`

E.2 Pointeurs

E.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 E.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	

E.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 E.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

E.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 E.4 – 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

E.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 E.5 – pointeurs – Exercice 4