

# 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	Exemple de structure de données linéaires dynamiques . . . . .	8
<b>A</b>	<b>Cours sur les pointeurs en C</b>	<b>10</b>
A.1	Syntaxe . . . . .	10
A.2	Opérateur autorisés sur les pointeurs . . . . .	10
<b>B</b>	<b>Liste des codes sources</b>	<b>13</b>
<b>C</b>	<b>Table des figures</b>	<b>14</b>
<b>D</b>	<b>Exercices</b>	<b>15</b>
D.1	TAD . . . . .	15
D.2	Pointeurs . . . . .	15

# 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>

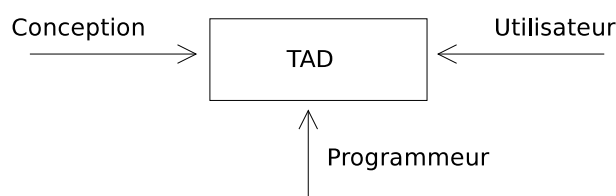


FIGURE 1.1 – Principe de base d'un TAD

## 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)
```

## 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.

**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]);
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 }

```

## 1.3 Exemple de structure de données linéaires dynamiques

### 1.3.1 Pile

Liste simplement chaînée dynamique à un point d'entrée

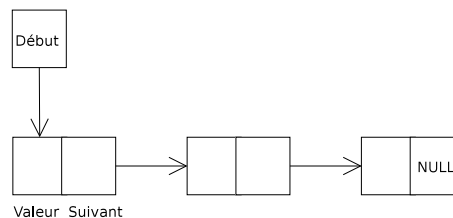


FIGURE 1.2 – Pile avec une liste simplement chaînée

```

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

```

### 1.3.2 File

Liste simplement chaînée à deux points d'entrée

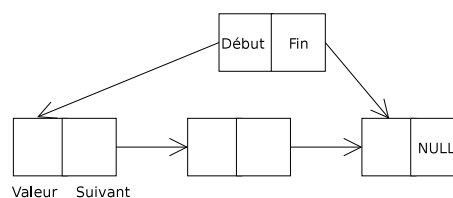


FIGURE 1.3 – File avec une liste simplement chaînée

```

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

```



## 1.3.3 Pile avec liste doublement chaînée

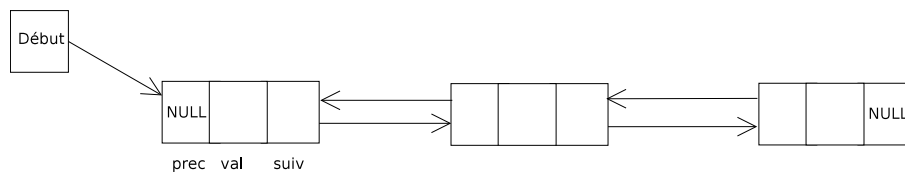


FIGURE 1.4 – Pile avec une liste doublement chaînée

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

## 1.3.4 File avec liste doublement chaîné

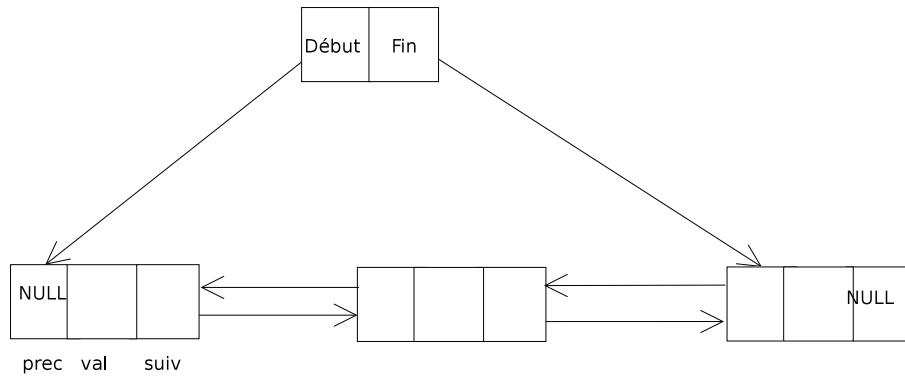


FIGURE 1.5 – File avec une liste doublement chaînée

---

# Cours sur les pointeurs en C

---

Déjà vu par le passages de paramètres.

## A.1 Syntaxe

### A.1.1 Déclaration

```
typePointé* nomPointeur
```

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

Listing A.1 – Exemple de déclaration

### A.1.2 Utilisation

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

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

### A.1.3 Constante

NULL représente une adresse inexistante.

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

## 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));

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

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);
```

1. Le programme contacte 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.

---

## Liste des codes sources

---

content/1.c . . . . .	7
A.1 Exemple de déclaration . . . . .	10
annexes/exercices/1.c . . . . .	15
annexes/exercices/pointeurs.c . . . . .	16
annexes/exercices/pointeurs2.c . . . . .	16
annexes/exercices/pointeurs3.c . . . . .	16
annexes/exercices/pointeurs4.c . . . . .	17

---

## Table des figures

---

1.1	Principe de base d'un TAD . . . . .	5
1.2	Pile avec une liste simplement chaînée . . . . .	8
1.3	File avec une liste simplement chaînée . . . . .	8
1.4	Pile avec une liste doublement chaînée . . . . .	9
1.5	File avec une liste doublement chaînée . . . . .	9

# Exercices

## D.1 TAD

### D.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**  $\text{remplacerOccurrence Pile} \times \text{Element} \times \text{Element} \rightarrow \text{Pile}$

**Préconditions** rien

**Axiones**

```

1 | remplacerOccurrence(creer(), x, y) = creer();
2 | remplacerOccurrence(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 | }

```

## 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;

```

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 | }

```

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 | }

```

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 | }

```