

PROGRAMMATION C (PARTIE 2)



Les chapitres traités en Partie 1

2

- ❑ Chapitre 1: Spécificités du Langage C
- ❑ Chapitre 2: Les variables scalaires et les opérateurs
- ❑ Chapitre 3: Les structures de contrôles
- ❑ Chapitre 4: Les fonctions d'entrées/sorties
- ❑ Chapitre 5: Les structures de données
- ❑ Chapitre 6: Types de données personnalisés
- ❑ Chapitre 7: Structurations des programmes C

Les pointeurs

A la fin de ce chapitre, l'étudiant sera capable de:

- ☐ Déclarer un pointeur
- ☐ Initialiser un pointeur
- ☐ Modifier le contenu d'un pointeur
- ☐ Utiliser les pointeurs dans le prototype d'une fonction (paramètres d'entrée , retour)
- ☐ Manipuler un tableau à travers un pointeur.



Chapitre 8: Les pointeurs

5

□ Introduction (1 / 2)

- l'accès au contenu (valeur) d'une variable se faisait par l'intermédiaire de son nom.

- Si par exemple on devait affecter à une variable "a" la valeur d'une autre variable "b", on y arrivait par l'instruction `a = b` ; dans laquelle un nom de variable figurait des deux côtés de l'opérateur d'affectation « = »

➔ **Adressage direct: Accès au contenu d'une variable par le nom de la variable.**



Chapitre 8: Les pointeurs

6

□ Introduction (2/2)

- Une variable n'est donc rien d'autre qu'une zone mémoire portant un certain nom défini par le programmeur.
- Mais on peut aussi choisir un chemin d'accès indirect par le biais de l'adresse de la variable. Pour ce la, on utilise ce qu'on appelle un **pointeur**.

➔ Adressage indirect: Accès au contenu d'une variable en passant par un Pointeur qui contient l'adresse de la variable.



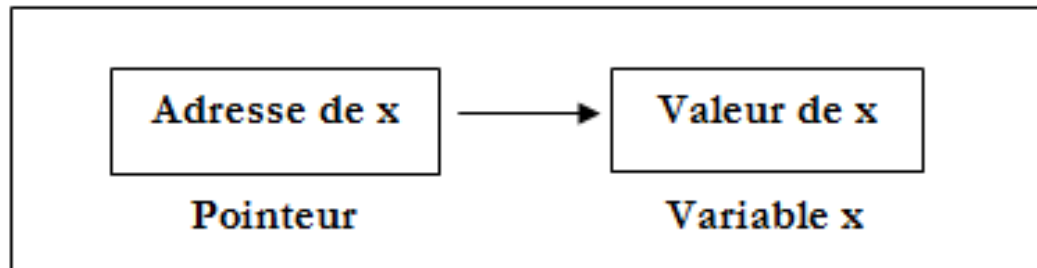
Chapitre 8: Les pointeurs

7

□ Définition (1 / 2)

- Un pointeur est une **variable** spéciale qui peut contenir l'**adresse** d'une autre variable.
- Chaque pointeur est **limité** à un type de données.
- Si un pointeur P contient l'adresse d'une variable A, on dit que P **pointe** sur A.
 - ▣ On dit aussi que le pointeur fait **référence à la variable**.

Mémoire



Variable pointeur
mémoire l'adresse d'une
donnée



Chapitre 8: Les pointeurs

8

□ Définition (2/2)

- Il existe aussi des pointeurs constants (adresse constante)
- **Exemple** : les noms des tableaux sont des pointeurs constants équivalents à l'adresse (de début) du tableau concerné

■ Variables et adresses :

`int x ;`

`&x` ← adresse de x dans la mémoire en Hexadécimal.

`int tab[10] ;`

`tab` ← adresse du premier élément du tableau (`&tab[0]`)



Chapitre 8: Les pointeurs

9

□ Variable ~ Pointeur

- Les pointeurs et les noms de variables ont le même rôle: ils donnent accès à un emplacement en mémoire.
- Par contre, un pointeur peut contenir différentes adresses mais le nom d'une variable reste toujours lié à la même adresse.



Chapitre 8: Les pointeurs

10

□ Définition de variables pointeurs

■ Syntaxe :

`< type-données> * < nom-pointeur> ;`

■ Exemple :

`int * pi ;` ← la variable "pi" est un pointeur vers une donnée de type int.

`char * pc ;` ← pc un pointeur vers un "char".

■ Peut contenir l'adresse d'une donnée de type "char".

■ pc à le type "char*"



Chapitre 8: Les pointeurs

11

□ Définition de variables pointeurs

▣ Initialisation vide :

- Le symbole NULL permet d'initialiser un pointeur qui ne pointe sur rien. Cette valeur NULL peut être affectée à tout pointeur quel que soit le type.
- Le symbole NULL est défini dans la librairie stdlib.h.

Exemple:

```
int *pNombre=NULL;
```

Dès qu'on déclare un pointeur, il est préférable de l'initialiser à NULL.



Chapitre 8: Les pointeurs

12

□ Définition de variables pointeurs

■ Initialisation :

`int x ;` // variable ordinaire

`int * pi ;` // variable pointeur

`float *pf ;`

`float y ;`

■ L'opérateur unaire & appliqué à une variable permet d'obtenir l'adresse de cette variable.

■ Soit : `pi = &x ;` pi est un pointeur vers int, initialisé avec l'adresse de la variable x ;

■ Affichage : `printf ("%x",&x) ;` // Hexadécimal

Ou `"%d"` //decimal



Chapitre 8: Les pointeurs

13

□ Accès indirect aux variables

- Pour pouvoir accéder, via un pointeur, à une donnée, on a besoin de l'opérateur "*"
- l'opérateur unaire "*" : appelé opérateur d'indirection (opérateur de référence).
- **Remarque** : L'utilisation de l'opérateur d'indirection "*" doit être ici strictement différenciée de son utilisation dans les définitions de variables pointeurs.

Exemple 1:

1. $x=3;$
2. $pi=&x ;$
3. $y =*pi ; \Leftrightarrow y=x;$



Chapitre 8: Les pointeurs

14

Exemple 2:

```
int *pNombre;
```

```
int n=20;
```

```
pNombre=&n; /* pNombre pointe sur l'entier n  
qui contient la valeur 20 */
```

```
printf ( ' %d ', *pNombre); /*affiche la valeur 20 */
```

```
*pNombre=40;
```

```
printf( ' %d ', *pNombre); /*affiche la valeur 40 */
```

```
printf( ' %d ', n); /*affiche la valeur 20 */
```



Chapitre 8: Les pointeurs

15

□ Les opérateurs définis sur les pointeurs

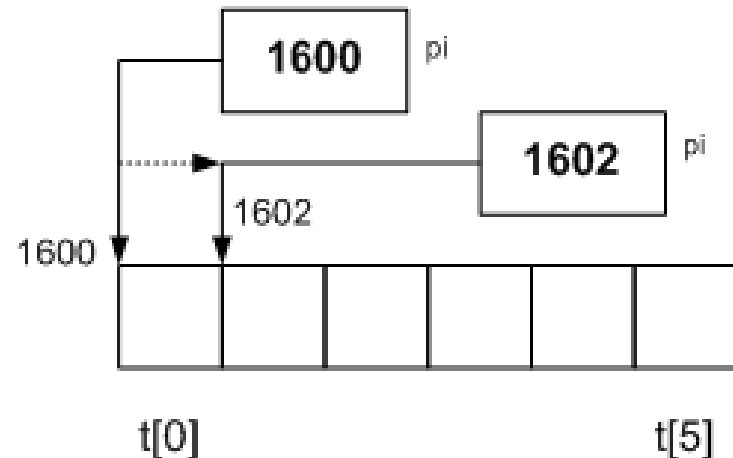
▣ Addition

■ Soit : `int t [6] ;`
`int * pi ;`

■ Alors : `pi =& t[0] ;` // pi pointe sur le premier élément du tableau 't' vers la variable "int" t[0]

L'instruction :

`pi=pi +1 ;`
// l'adresse dans
//pi est passée
//de 1600 à 1602



Chapitre 8: Les pointeurs



16

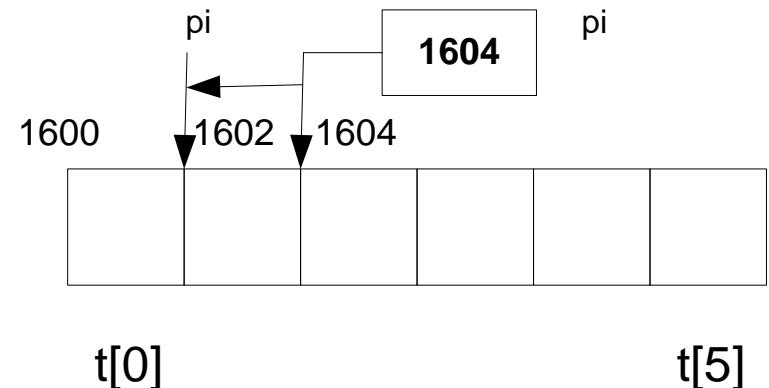
□ Incrémentation et décrémentation

▣ Incrémentation :

Soit : `int i[10] ;`
 `pi = & i[0] ;` avec `int * pi;`
 `pi = pi + 1 ;` \Leftrightarrow `pi ++ ;` ou `++ pi ;`
 `(*pi) ++ ;` // On augmente de 1 la valeur de la
 // donnée pointée actuellement par pi.

▣ Décrémentation :

Soustraction :
`pi = pi - 1 ;` \Leftrightarrow `pi - - ;` ou `--pi;`





Chapitre 8: Les pointeurs

17

□ Priorité des opérateurs

- ⑩ Les opérateurs `*` et `&` ont la même priorité que les autres opérateurs unaires (`!`, `++`, `--`).
- ⑩ Si les parenthèses ne sont pas utilisées, les expressions sont évaluées de droite à gauche.

Exemple:

Après l'instruction

```
P = &X;
```

les expressions suivantes, sont équivalentes:

```
Y = *P+1    ⇔ Y = X+1
```

```
*P = *P+10  ⇔ X = X+10
```

```
*P += 2     ⇔ X += 2
```

```
++*P        ⇔ ++X
```

```
(*P)++     ⇔ X++
```

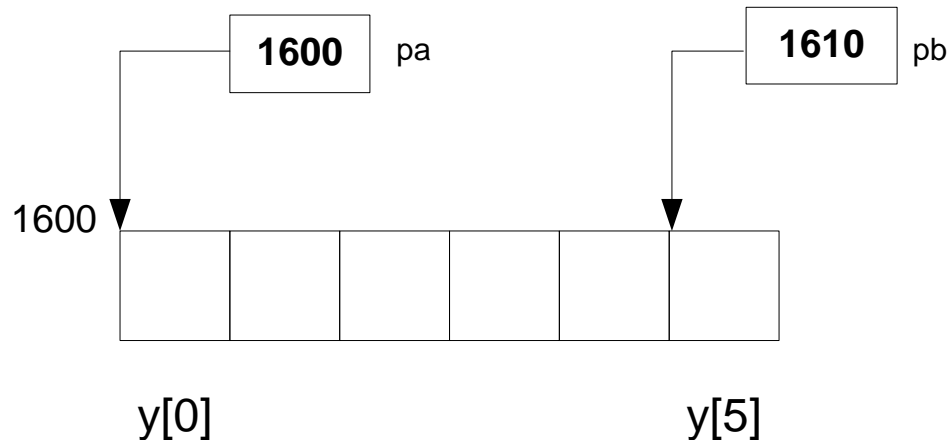


Chapitre 8: Les pointeurs

18

□ Comparaison

Soit : `short y[6];`
 `short * pa = &y[0], * pb = &y[5];`



`if (pa == pb) { // ("les deux pointeurs pointent vers la même donnée") ; }`
nous pouvons utilisé aussi : `<=`, `>=`, `!=`

Chapitre 8: Les pointeurs

19

Exercices

Serie 1



Chapitre 8: Les pointeurs

20

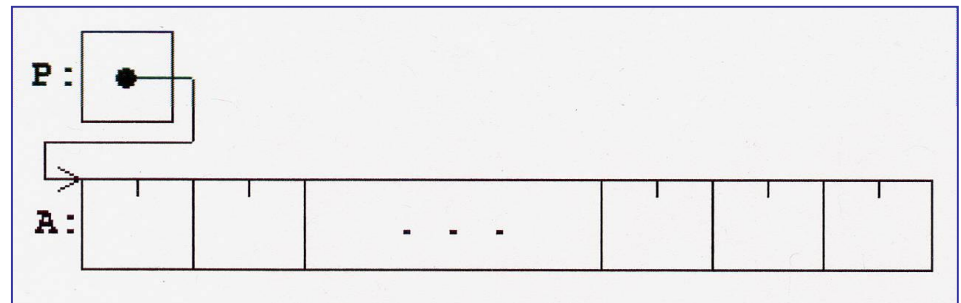
□ Les tableaux et les pointeurs

- Chaque opération avec des **indices** de tableaux peut être aussi exprimée à l'aide de pointeurs.
- Comme nous l'avons déjà vu dans le cours, le nom d'un tableau représente l'**adresse** du **premier élément** du tableau.

`&tableau[0]` et `tableau` représentent l'adresse du premier élément du tableau

- Le nom d'un tableau est un pointeur constant sur le premier élément du tableau.

```
int A[10];  
int * P;  
P = A;    /*est équivalente à P = &A[0]*/
```





Chapitre 8: Les pointeurs

21

□ Les tableaux et les pointeurs

Comme **A** représente l'adresse de **A[0]**,

*** (A+1)** désigne le contenu de **A[1]**

*** (A+2)** désigne le contenu de **A[2]**

...

*** (A+i)** désigne le contenu de **A[i]**

- Un *pointeur* est une variable,
donc des opérations comme **P = A** ou **P++** sont permises.

- Le *nom d'un tableau* est une constante,
donc des opérations comme **A = P** ou **A++** sont impossibles.



Chapitre 8: Les pointeurs

22

□ Les tableaux et les pointeurs

▣ Arithmétique des pointeurs

- ⑩ On peut **déplacer** un pointeur dans un plan mémoire à l'aide des opérateurs :
d'addition, de soustraction, d'incrémentement, de décrémentation.

Affectation par un pointeur sur le même type

- ⑩ P1 et P2 deux pointeurs sur le même type de données

P1 = P2; //affecte l'adresse contenue dans P2 à P1 → P1 pointe sur la même variable que P2.

Addition et soustraction d'un nombre entier

- ⑩ Si P pointe sur l'élément A[i] d'un tableau, alors

P+n pointe sur A[i+n]

P-n pointe sur A[i-n]

- ⑩ le déplacement d'un pointeur par l'opérateur + ou – se fait par un nombre d'octets multiple de la taille de la variable sur laquelle il pointe.



Chapitre 8: Les pointeurs

23

□ Les tableaux et les pointeurs

▣ Arithmétique des pointeurs

Incrémentation et décrémentation d'un pointeur

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

P++; P pointe sur A[i+1]

P+=n; P pointe sur A[i+n]

P--; P pointe sur A[i-1]

P-=n; P pointe sur A[i-n]



Chapitre 8: Les pointeurs

24

□ Les tableaux et les pointeurs

▣ Arithmétique des pointeurs

Soustraction de deux pointeurs

⑩ Soient P1 et P2 deux pointeurs qui pointent *sur deux cases du même tableau*:

P1-P2 fournit le nombre de cases comprises entre P1 et P2.

Le résultat de la soustraction **P1-P2** est

- négatif, si P1 précède P2
- nul, si P1 = P2
- positif, si P2 précède P1

Comparaison de deux pointeurs

On peut comparer deux pointeurs par <, >, <=, >=, ==, !=.

La comparaison de deux pointeurs qui pointent *sur deux cases du même tableau* est équivalente à la comparaison des indices correspondants.



Chapitre 8: Les pointeurs

25

□ Les tableaux et les pointeurs

▣ Récapitulatif

Soit un tableau A de type quelconque et i un indice d'une composante de A:

A désigne l'adresse de **A[0]**

A+i désigne l'adresse de **A[i]**

***(A+i)** désigne le contenu de **A[i]**

Si $P = A$, alors

P pointe sur l'élément **A[0]**

P+i pointe sur l'élément **A[i]**

***(P+i)** désigne le contenu de **A[i]**



Chapitre 8: Les pointeurs

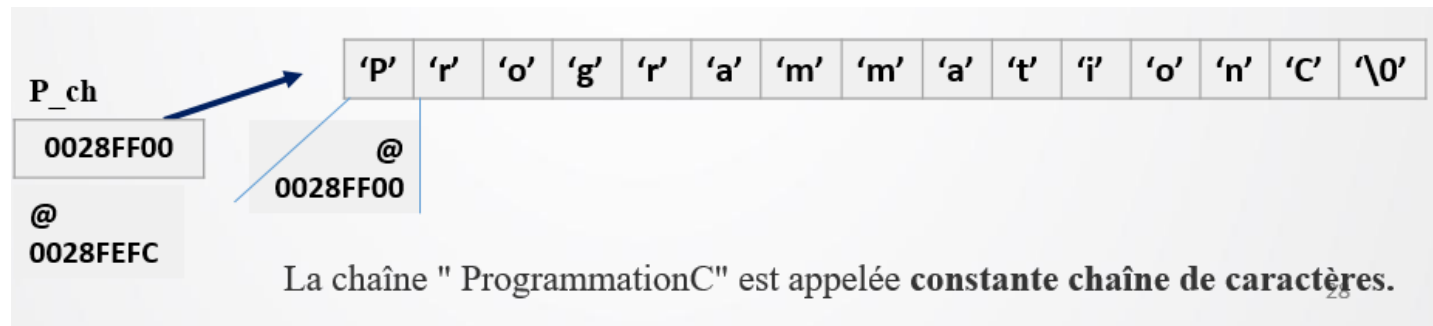
26

□ Les pointeurs et les chaînes de caractères

- Tout ce qui a été mentionné concernant les pointeurs et les tableaux reste vrai pour les pointeurs et les chaînes de caractères.
- Une chaîne de caractère est un tableau de caractères qui se termine par le caractère spécial ' \0 '.
- On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur char :

Exemple:

`char* p_ch = " ProgrammationC ";` //p_ch est un pointeur sur une variable de type char.





Chapitre 8: Les pointeurs

27

□ Les pointeurs et les chaînes de caractères

- Le **compilateur** place la chaîne « ProgrammationC" dans une zone mémoire de 15 octets, c'est-à-dire un tableau de 15 caractères qui contient les 14 lettres du mot ProgrammationC et le caractère ' \0 '.
- **A l'exécution**, l'affectation `p_ch = " ProgrammationC "` met l'adresse de cette zone mémoire dans le pointeur `p_ch` → `p_ch` pointe donc sur le premier caractère de la chaîne.

Chapitre 8: Les pointeurs



Chapitre 8: Les pointeurs

29

Exercices

Serie 2



Chapitre 8: Les pointeurs

30

□ Allocation dynamique de tableaux

- Un tableau statique a une dimension bien définie à l'avance et possède en outre un nom, à l'aide du quel on peut accéder aux divers éléments du tableau.
- Un tableau dynamique, en revanche, a une dimension variable et ne possède pas de nom par lequel aurait accès aux éléments du tableau. Désormais, on utilise un pointeur qui pointe vers des données du type des valeurs saisies.
- Fonctions prédéfinies pour la gestion dynamique de la mémoire. Nous ferons appel à "stdlib.h" dans nos programmes.



Chapitre 8: Les pointeurs

31

□ Allocation de mémoire avec malloc

■ Syntaxe :

`<pointeur> = malloc (< taille>) ; // taille en octet`

- Retourne l'adresse du bloc de mémoire réservé rangée dans un pointeur. Et on peut via ce pointeur accéder au bloc de mémoire alloué.

■ Exemple :

- `double * tab ;`
- `tab = malloc (400) ; // allocation d'un bloc mémoire de
//400 octets et rangement de son adresse
//dans un pointeur double.`

tab est un tableau de 50 éléments de type double



Chapitre 8: Les pointeurs

32

□ Allocation dynamique de tableaux

▣ Gestion des erreurs :

- Si malloc n'arrive pas à réserver le bloc mémoire souhaité (pas assez d'espace mémoire) elle retourne le pointeur NULL.

```
if(tab==NULL)
```

```
    printf("Allocation demander impossible");
```

```
else
```

```
{ for(i=0;i<50;i++)
```

```
    tabl[i]=(10*i); }
```




Chapitre 8: Les pointeurs

33

□ Allocation dynamique de tableaux

- Si on manipule autre types de données :

- **Exemple** : int occupent → 2 ou 4 octets selon la machine.

 - Soit : 200 elem (2 octets) ou 100 elem (4 octets)

- On résout le problème par l'opérateur "**sizeof**" pour 100 valeurs int :

On aura:

```
tab = malloc (100 * sizeof(int))
```



Chapitre 8: Les pointeurs

34

□ Allocation dynamique de tableaux

▣ Conversion des pointeurs « void »

- La valeur que malloc renvoie au programme est un pointeur de type « void ». Un tel pointeur permet de mémoriser les adresses de données de tous types. Ce pendant, il est conseillé de convertir un pointeur de types indéterminé.

▣ On aura :

```
tab = (double*) malloc (100 * sizeof (double))
```

▣ Ou

```
tab = ( int*) malloc (100 * sizeof(int)) ;
```



Chapitre 8: Les pointeurs

35

□ Allocation dynamique de tableaux

▣ Libération de la mémoire avec « free »

- Elle permet de libère l'espace mémoire qui a été alloué par les fonctions malloc, et realloc.

▣ Syntaxe :

`free (<pointeur>) ;`

▣ Exemple :

`free (tab) ;`



Chapitre 8: Les pointeurs

36

□ Pointeurs et structures

- Les structures pouvant être gérées par des pointeurs.
On définit les pointeurs vers des structures

Soit :

<pre>struct article { char name[20] ; int num ; };</pre>	<pre>struct article * px; /*définit un pointeur px vers des données de type "struct article"*/ struct article x; px=&x; // Ranger l'adresse de la variable // structurée « x » dans le pointeur « px »</pre>
--	---

- x et px permettant d'accéder aux champs de la variable structurée « x »
 - x.name ; et x.num ; \Leftrightarrow (*px).name; et (*px).num ;



Chapitre 8: Les pointeurs

37

□ Pointeurs et structures

▣ Accès aux champs d'une structure pointée :

■ Syntaxe :

$(* < \text{pointeur} >). < \text{champ} > ;$

▣ Peut être remplacé par :

$< \text{pointeur} > \rightarrow < \text{champ} > ;$

$(*px).name \Leftrightarrow px \rightarrow name ;$



Chapitre 8: Les pointeurs

38

□ Les pointeurs et les fonctions

□ Passage des paramètres par adresse (référence)

- En C, **return** ne permet de retourner qu'**une seule** valeur.
- Parfois une fonction doit retourner **plusieurs** résultats en sortie.
- Il n'y a qu'une solution pour retourner plusieurs résultats: passer en paramètre l'**adresse** des variables où seront stockés les résultats.
- Le passage par adresse est aussi utile pour modifier le contenu des variables déclarées dans d'autres fonctions .



Chapitre 8: Les pointeurs

39

□ Les pointeurs et les fonctions

□ Passage des paramètres par adresse (référence)

■ **Exemple:** Une fonction qui retourne le minimum et le maximum de deux entiers a et b.

```
void min_max(int a, int b, int *pmin, int *pmax)
```

```
{  
    if(a<b) {  
        *pmin=a;  
        *pmax=b;  
    }  
    else {  
        *pmin=b;  
        *pmax=a;  
    }  
}
```

```
int main()  
{  
    int a,b, min,max;  
    printf("Entrer deux entiers");  
    scanf("%d %d", &a,&b);  
    min_max(a,b,&min,&max);  
    printf("le min est %d le max est %d",min,max);  
}
```

Chapitre 8: Les pointeurs

40

Exercices

Serie 3

Les listes chaînées

Chapitre 9: Les listes chaînées

42

❖ Les tableaux forment une suite de variables de même type associées à des emplacements **consécutifs** dans la mémoire.

→ **Problème:** On peut avoir suffisamment d'espace mais qui n'est pas forcément **contiguë**!

❖ Les opérations d'insertion ou de suppression d'une case dans un tableau ne sont pas coûteuses dans le cas où elles sont effectuées **à la fin**. Par contre, l'insertion ou la suppression d'un élément au **début** ou au **milieu** nécessitent un décalage du contenu de plusieurs cases

→ **Problème:** un traitement coûteux en terme de temps d'exécution d'un programme.

❖ Lorsqu'il n'y a pas assez de place dans le tableau, il est nécessaire d'effectuer une réallocation afin de l'agrandir

→ **Problème:** il est possible qu'une zone entièrement différente soit réservée, ce qui implique de recopier l'intégralité du tableau dans une nouvelle zone mémoire et de libérer l'ancienne zone.

Chapitre 9: Les listes chaînées

43

Afin de contourner les difficultés liées aux tableaux, il faut adopter une structure de données:

- qui n'exige pas une contiguïté des éléments à stocker en mémoire
- dont les opérations d'ajout et de suppression sont moins coûteuses en terme de temps par rapport aux tableaux.

C'est le but des *listes chaînées*.

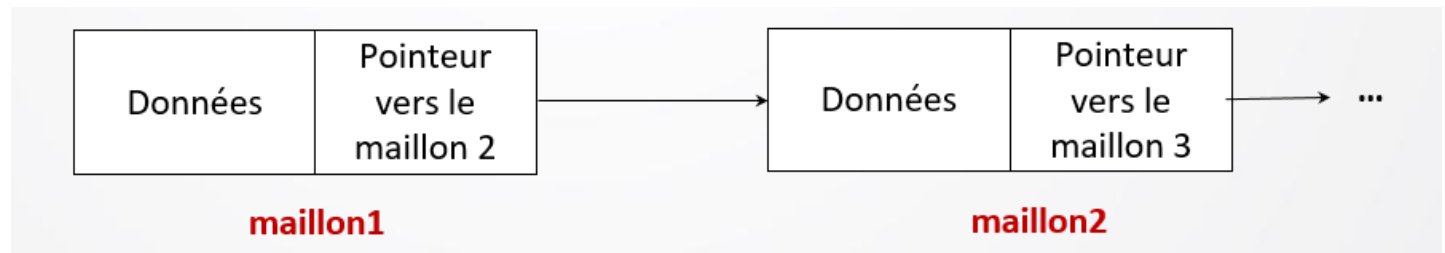


Chapitre 9: Les listes chaînées

44

Définition:

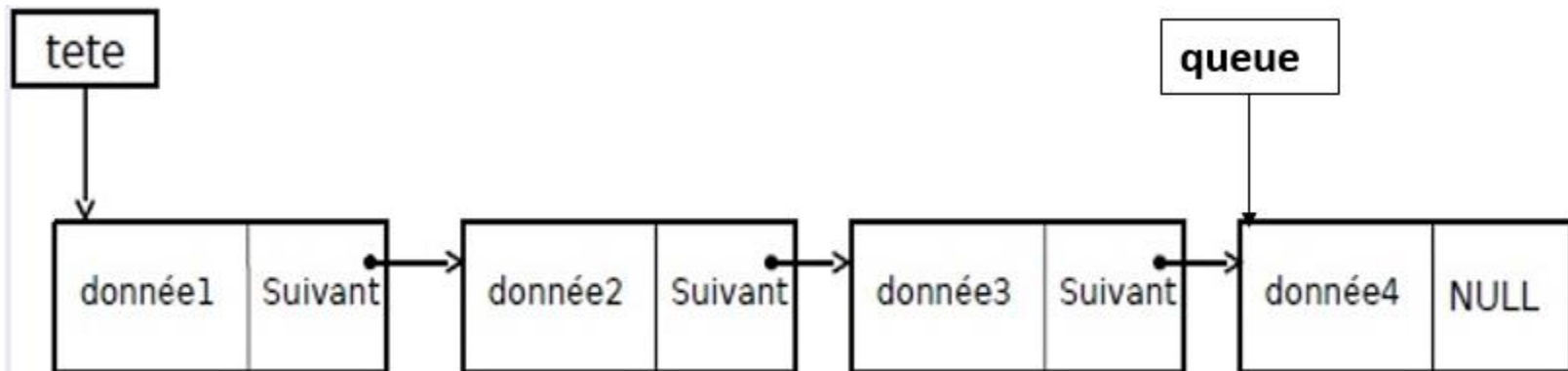
- Une liste chaînée est une structure de données linéaire qui permet de stocker une suite d'éléments de même type qui sont chaînés entre eux.
- L'élément de base d'une liste chaînée s'appelle **maillon**, **cellule** ou **nœud**
- Chaque cellule de la liste est constitué de **deux parties**:
 - ✓ Un champ de données.
 - ✓ un pointeur vers une structure de même type (l'élément suivant de la liste).



Chapitre 9: Les listes chaînées

45

- Le champ **pointeur vers un maillon** pointe vers le maillon suivant de la liste.
- S'il n'y a pas de maillon suivant, **le pointeur vaut NULL**.
- Une **liste vide** est une liste qui ne contient pas de maillon. Elle a donc **la valeur NULL**.
- La terminologie suivante est généralement employée :
 - L'adresse du **premier** maillon de la liste est appelé **tête** ;
 - L'adresse du **dernier** maillon de la liste est appelé **queue**.
- Une liste est représentée par sa **tête**. Accédant à la tête, on peut accéder à tous les autres éléments



Chapitre 9: Les listes chaînées

46

□ Liste chaînée: Caractéristiques

- Une liste chaînée est une structure linéaire qui **n'a pas de dimension fixée** à sa création.
- Ses éléments de **même type** sont **éparpillés** dans la mémoire et **reliés** entre eux par des **pointeurs**.

Par exemple: Le premier élément de la liste peut se trouver à l'adresse 1024, le second à l'adresse 256, le troisième en 532, le quatrième en 2084, etc.

- La liste est accessible uniquement par **sa tête** indiquant l'adresse de son premier élément.
- **La fin** de la liste est caractérisée par **un pointeur NULL**.
- Le dernier élément existant, son champ **suivant** vaut **NULL**.

Chapitre 9: Les listes chaînées

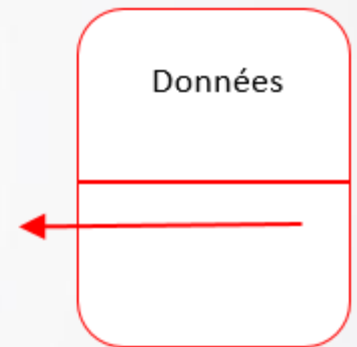
47

Syntaxe :

```
struct Nom_cellule
```

```
{  
  Data D; } Les données à stocker
```

```
struct Nom_cellule * suivant ; } Pointeur sur l'élément suivant  
};
```



Chapitre 9: Les listes chaînées

48

□ Exemple: une liste d'entiers

```
struct Cellule
{
    int valeur ; // Donnée
    struct Cellule * suivant ; // pointeur sur
    l'élément suivant
};
typedef struct Cellule * liste;
```

Ou bien

```
typedef struct Cellule Cellule;
struct Cellule
{
    int valeur ; // Donnée
    Cellule * suivant ; // pointeur sur l'élément
    suivant
};
typedef Cellule * liste;
```

Ces instructions déclarent une structure Cellule composée de :

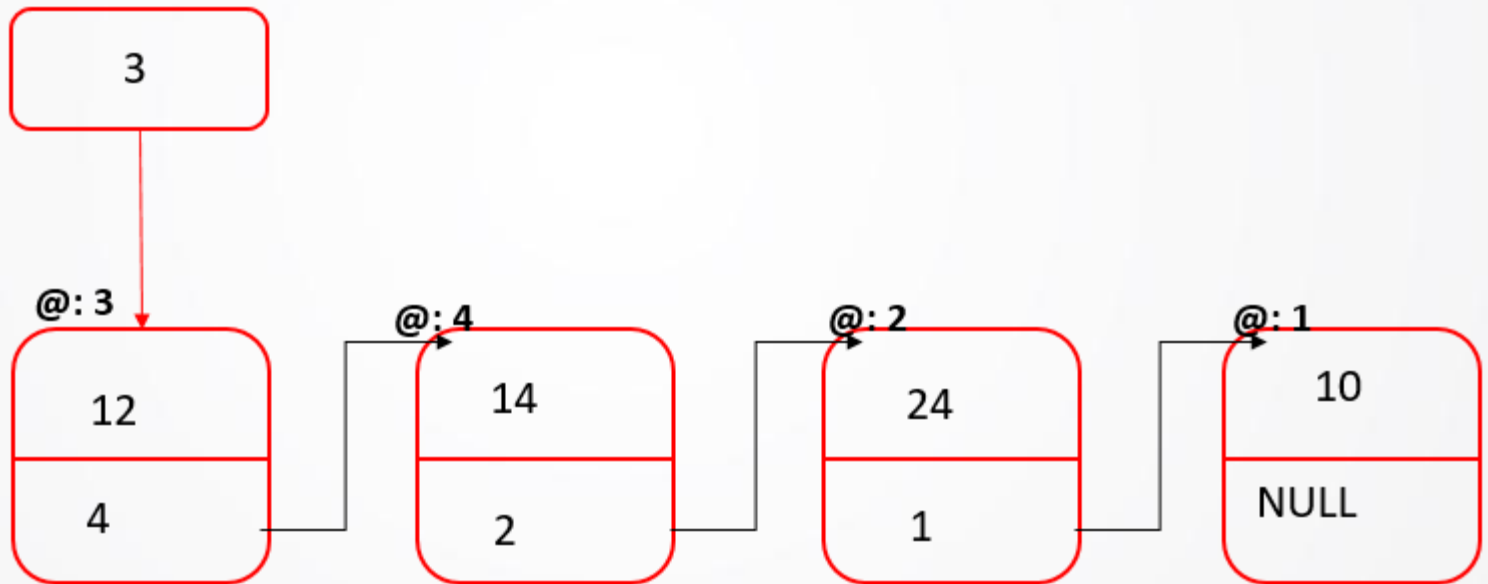
1. Un premier champ contenant la donnée (un entier dans ce cas).
2. Un second champ indiquant le pointeur sur la cellule suivante.
3. Pour des raisons pratiques de facilité de manipulation, on définit le nouveau type **liste** comme étant un pointeur sur une cellule.

Chapitre 9: Les listes chaînées

49

- Le 1er élément de la liste vaut 12 et se trouve à l'adresse 3, début de la liste.
- Le 2e élément de la liste vaut 14 et se trouve à l'adresse 4 (car le suivant de la cellule d'adresse 3 est égal à 4)

Tête: liste



Chapitre 9: Les listes chaînées

50

□ Opérations élémentaires sur les listes

1. **Liste vide:** savoir si une liste est vide ou pas.
2. **Parcourir :** passer chaque élément de la liste dans l'ordre du début vers la fin.
3. **Ajouter** une cellule à la liste, soit au début, soit à la fin, soit à une position donnée.
4. **Supprimer :** enlever une cellule de la liste.
5. **Détruire une liste :** libérer tous les maillons de la liste

Chapitre 9: Les listes chaînées

51



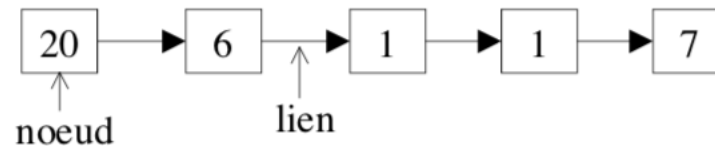
Les tableaux :

- ont une taille fixe ;
- occupent un espace contiguë.

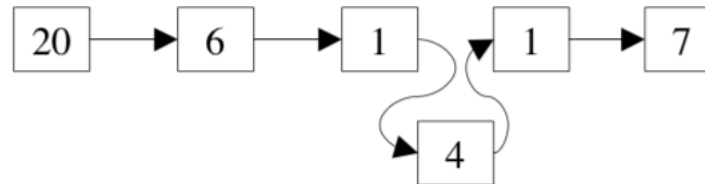
	20	6	1	...	1	7
Indices :	0	1	2	...	n-1	n



Une liste chaînée est un ensemble d'éléments organisés séquentiellement



Ajouter un élément :



Supprimer un élément :



Chapitre 9: Les listes chaînées

52



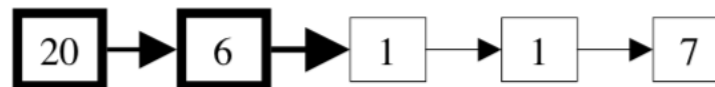
Avec un tableau : accès direct à un élément en connaissant son indice.

20	6	1	1	7
----	---	---	---	---

indice : 2



Avec une liste chaînée : parcourir la liste pour accéder à un élément.



Chapitre 9: Les listes chaînées

53

□ Liste vide

- ▣ Si la tête pointe vers **NULL** c'est que la liste est vide.

Déclaration de la liste

```
struct Cellule
{
    int valeur ;
    struct Cellule * suivant ;
};
typedef struct Cellule * liste;
int main{
    liste l=NULL; // déclaration et initialisation de la liste
}
```

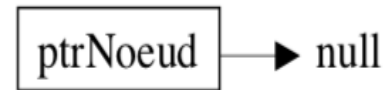
Liste vide

```
int liste_vide(liste l)
{
    return (l==NULL);
}
```

Chapitre 9: Les listes chaînées

54

□ Initialiser liste



```
NŒUD* initialiser(){  
    return NULL ;  
}
```

```
void main(){  
    NŒUD* ptrNoeud ;  
    ptrNoeud = initialiser() ;  
}
```

Chapitre 9: Les listes chaînées

55

□ Liste vide

- ▣ Si la tête pointe vers **NULL** c'est que la liste est vide.

Déclaration de la liste

```
struct Cellule
{
    int valeur ;
    struct Cellule * suivant ;
};
typedef struct Cellule * liste;
int main{
    liste l=NULL; // déclaration et initialisation de la liste
}
```

Liste vide

```
int liste_vide(liste l)
{
    return (l==NULL);
}
```

Chapitre 9: Les listes chaînées

56

□ Parcours d'une liste

➤ Parcourir une liste c'est-à-dire aller d'un bout à l'autre de la liste en traitant chaque élément consécutivement. Pour cela, l'approche standard consiste:

- ⑩ à utiliser un pointeur temporaire, que nous noterons **tmp**, et qui ne sera utilisé que pour cette tâche de parcours.
- ⑩ Le pointeur tmp est initialisé au début de la liste à la tête, et est modifié dans une boucle en lui affectant à chaque fois l'adresse de la cellule suivante.

void parcourir(liste l)

Parcours

```
void parcourir(liste l)
```

```
{
```

```
    struct cellule* tmp=l;
```

Cas tête
Null

```
    if(l==NULL)
```

```
        printf("la liste est vide");
```

```
    else
```

```
    {
```

```
        while (tmp!=NULL)
```

```
        {
```

```
            printf("%d",tmp->valeur);
```

```
            tmp=tmp->suivant;
```

```
        }
```

```
    }
```

Cas tête
Non Null

Chapitre 9: Les listes chaînées

57

□ Ajout d'un élément :

Ajouter un élément suppose:

- 1.L'initialisation du champ de données.
- 2.L'initialisation du champ indiquant l'adresse de son suivant.
- 3.Décider où l'élément sera ajouté :
 - **Au début**
 - **A la fin**
 - **Au milieu, avant un critère donné.**

Chapitre 9: Les listes chaînées

58

Ajout d'un élément :

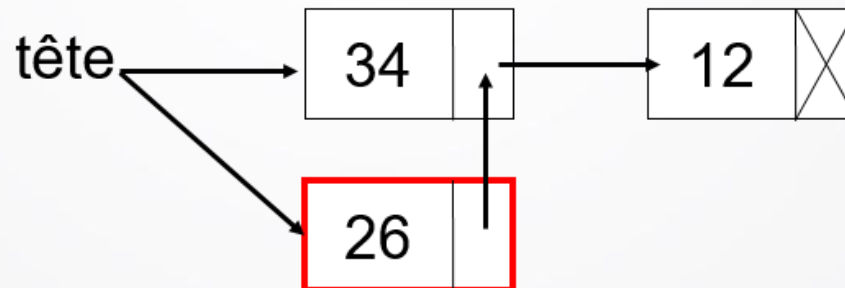
au début de la liste - à la fin de la liste – au milieu avant critère

1. Créer une nouvelle cellule (Allocation et initialisation du champs de donnée)

SI la création fut un succès alors

2. On fait pointer la nouvelle cellule vers le premier élément de la liste.

3. On fait pointer la tête de liste sur le nouveau nœud.



Chapitre 9: Les listes chaînées

59

Ajout d'un élément :

au début de la liste - à la fin de la liste – au milieu avant critère

Ajout au début de la liste	
liste ajouter_ deb (liste l, int val)	
{	
Déclaration	struct Cellule* nouv;
Allocation dynamique Et remplissage d'une nouvelle cellule	nouv = (struct Cellule*) malloc(sizeof(struct Cellule)); nouv->valeur=val; nouv->suivant=NULL;
Le suivant de la nouvelle cellule est l'ancienne tête	nouv->suivant=l;
La nouvelle tête est l'adresse de la nouvelle cellule	l=nouv;
L'adresse de la première cellule a changé -->elle doit être retournée	return(l); }

Chapitre 9: Les listes chaînées

60

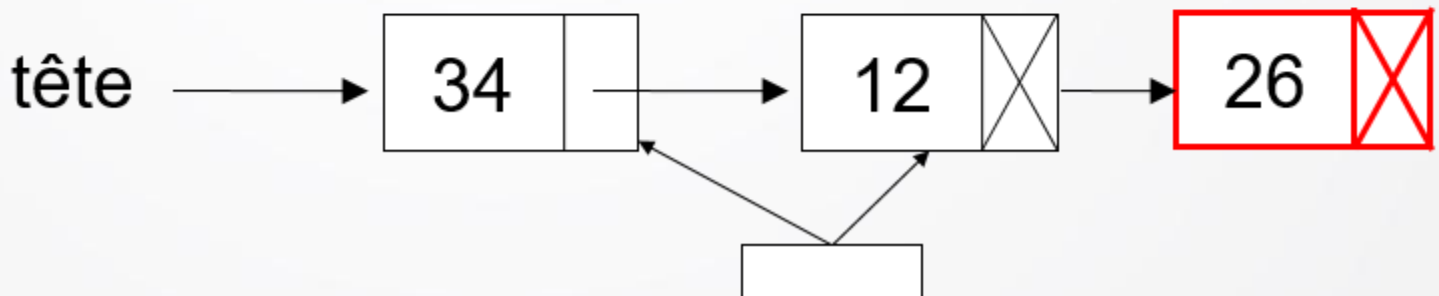
Ajout d'un élément :

au début de la liste - **à la fin de la liste** – au milieu avant critère

1. Allouer dynamiquement une nouvelle cellule .
2. Initialiser le champs de donnée et le pointeur suivant à NULL

SI la création fut un succès ALORS

3. La tête est l'adresse de la nouvelle cellule
4. On parcourt la liste jusqu'à atteindre l'adresse de la dernière cellule.
5. On fait pointer la dernière cellule sur la nouvelle cellule



Chapitre 9: Les listes chaînées

61

Ajout d'un élément :

au début de la liste - **à la fin de la liste** – au milieu avant critère

	<pre>liste ajouter_Fin(liste l, int val) {</pre>
Déclaration	<pre> struct Cellule * nouv, *parc;</pre>
Création nouvelle cellule	<pre> nouv = (struct Cellule*) malloc(sizeof(struct Cellule)); nouv->valeur=val; nouv->suivant=NULL;</pre>
Cas tête ==Null	<pre> if(l==NULL) { l=nouv; }</pre>
Cas tête !=Null	<pre> else { parc=l; while(parc->suivant!=NULL) parc=parc->suivant; parc->suivant = nouv; }</pre>
Fin	<pre> return(l); }</pre>

Chapitre 9: Les listes chaînées

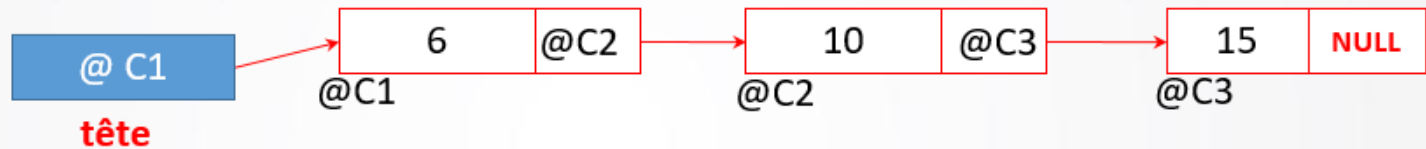
62

Ajout d'un élément :

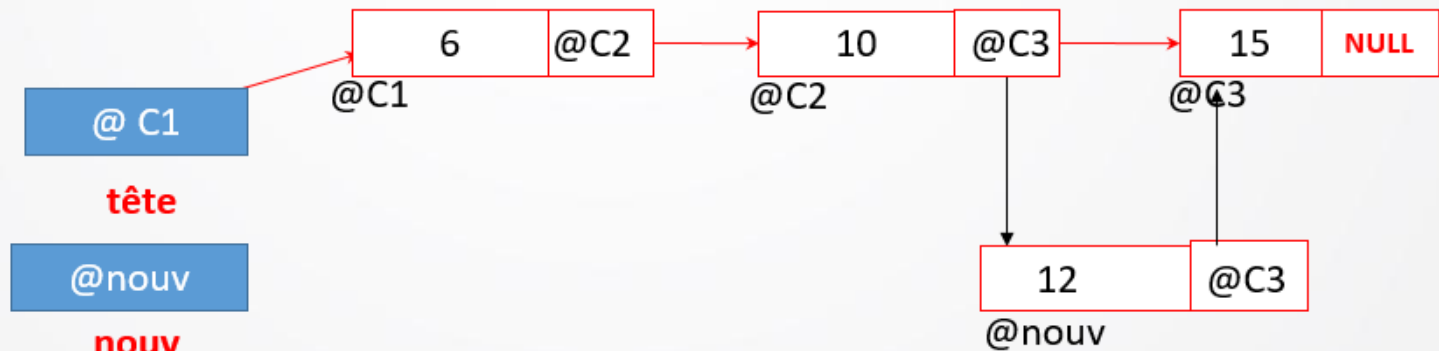
au début de la liste - à la fin de la liste – **au milieu avant critère**

Par exemple, insérer 12 avant la cellule qui contient la valeur 15

Liste initiale



Liste résultat



Chapitre 9: Les listes chaînées

63

Ajout d'un élément :

au début de la liste - à la fin de la liste – **au milieu avant critère**

	<pre>liste ajoutmilieu (liste l , int val, int critere) {</pre>
Déclaration	<pre>struct Cellule* nouv=NULL, *precedent=NULL,*courant=l;</pre>
Recherche de la cellule vérifiant le critère et de la cellule qui la précède	<pre>while (courant!=NULL && courant->valeur !=critere) { precedent=courant; courant=courant->suivant; }</pre>
Pas d'insertion	<pre>if(courant==NULL)//liste vide ou critère non vérifié dans toutes les cellules. printf(«critère non vérifié »); else //critère vérifiée</pre>
Allocation et remplissage d'une nouvelle cellule	<pre>{ nouv=(struct Cellule*)malloc(sizeof(struct Cellule)); nouv->valeur=val;</pre>
Changement de la tête	<pre>nouv->suivant=courant; If(courant==l) l=nouv;</pre>
Changement du chainage	<pre>else precedent->suivant=nouv; }</pre>
Ne pas perdre la nouvelle tête	<pre>return l; }</pre>

Chapitre 9: Les listes chaînées

64

□ Supprimer un élément

Pour supprimer une cellule, il faut tester les 4 scenarios suivants:

- La liste est vide: **suppression impossible**
- La liste contient un seul élément qui vérifie le critère de suppression: **la liste devient vide**
- La liste contient plusieurs éléments dont le premier vérifie le critère de suppression: **changement de la tête**
- La cellule qui vérifie le critère de suppression se trouve au milieu ou à la fin: **changement du chaînage**

Chapitre 9: Les listes chaînées

65

Suppression d'un élément :

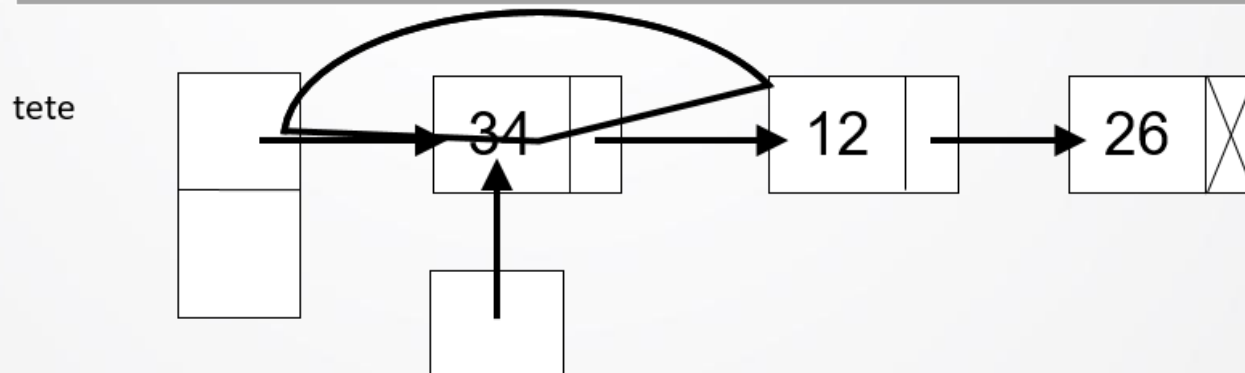
au début de la liste - à la fin de la liste – au milieu de la liste

SI la liste n'est pas vide:

1. On fait pointer un pointeur sur le premier élément de la liste.

2. On pointe la tête de liste sur le deuxième nœud (NULL s'il n'y avait qu'un seul nœud).

3. On détruit le nœud pointé par le pointeur.



Chapitre 9: Les listes chaînées

66

Suppression d'un élément :

au début de la liste - à la fin de la liste – au milieu de la liste

Déclaration

```
liste supprimerDebut(liste l)
{
    struct Cellule* tmp;
```

**Cas liste
Non vide**

```
    if(l!=NULL)
    {
        tmp=l;// pointer sur le premier élément de la liste
        l=l->suivant; //pointer la tête sur le deuxième élément de la liste

        free(tmp); // détruire la cellule
    }
```

Fin

```
    return(l);
}
```

Chapitre 9: Les listes chaînées

67

Suppression d'un élément :

au début de la liste - **à la fin de la liste** – au milieu de la liste

SI la liste n'est pas vide **ALORS**

SI la liste ne contient qu'un seul élément
ALORS

Supprimer au début de la liste.

SINON

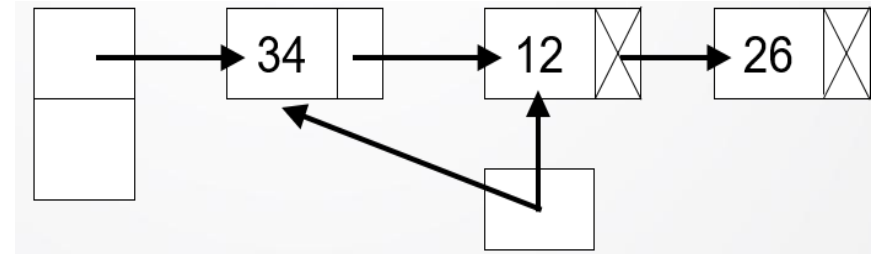
On fait pointer un pointeur sur
l'avant dernier nœud.

On détruit le dernier nœud.

On fait pointer le suivant de
l'avant dernier nœud vers NULL.

FINSI

FINSI



Chapitre 9: Les listes chaînées

68

Suppression d'un élément :

au début de la liste - **à la fin de la liste** – au milieu de la liste

	<pre>liste supprimer_Fin(liste l) {</pre>
Déclaration	<pre> struct cellule* tmp, courant;</pre>
Cas d'une liste qui contient une seule cellule	<pre> if (l!=NULL) // si la liste n'est pas vide { if(l->suivant==NULL) // s'il n'y a qu'une seule cellule { free(l); l=NULL; } </pre>
Cas liste qui Contient au moins deux cellules	<pre> else // s'il y a au moins deux cellules { tmp=l; courant=l; // à partir du premier, while(tmp->suivant!=NULL) // regarder toujours le suivant du suivant pour accéder à l'avant dernier { courant =tmp; tmp=tmp->suivant; } courant->suivant=NULL; free(tmp); } </pre>
Fin	<pre> } return l; }</pre>

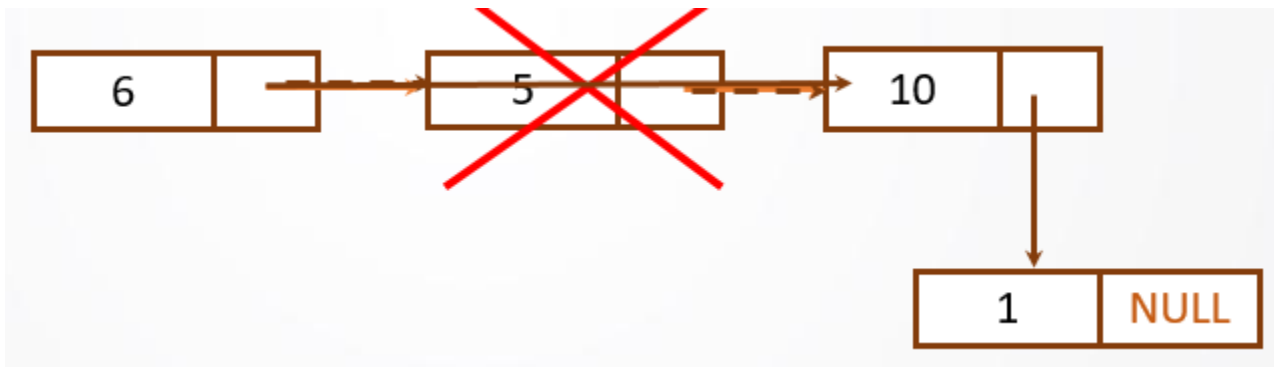
Chapitre 9: Les listes chaînées

69

Suppression d'un élément :

au début de la liste - à la fin de la liste – **au milieu de la liste**

Par exemple, Supprimer le nœud contenant la valeur « 5 »



Chapitre 9: Les listes chaînées

70

Suppression d'un élément :

au début de la liste - à la fin de la liste – **au milieu de la liste**

1. Définir deux pointeurs

- Pointeur courant vers l'élément à supprimer
- Pointeur précédent vers l'élément qui le précède
 - struct Cellule * courant = tete ;
 - struct Cellule * precedent = tete ;

2. Tester si l'élément à supprimer se trouve au début de la liste

3. Parcours de la liste jusqu'à trouver l'élément à supprimer

- while(courant != NULL && Trouve==0)
 - ✓ If(courant ->valeur != val){
 - ✓ precedent = courant;
 - ✓ courant = courant->suivant; }
 - ✓ else
 - ✓ Trouve=1;

4. Se débarrasser du maillon pointé par le pointeur courant et ensuite le libérer grâce à la fonction free

- precedent -> suivant = courant->suivant ;
- free(courant);

Chapitre 9: Les listes chaînées

71

Suppression d'un élément :

au début de la liste - à la fin de la liste – **au milieu de la liste**

	<pre>liste supprimer_milieu(liste l, int v) {</pre>
Déclaration	<pre> struct cellule * courant=l,* precedent=NULL; Int trouve=0;</pre>
Cas suppression début	<pre> if(l!=NULL) { if(l->val==v) l=supprimer_deb(l); else{</pre>
Cas suppression milieu	<pre> while(courant!=NULL&&!trouve) { if(courant->val==v) trouve=1; else { precedent=courant; courant=courant->suivant; } } If(trouve) { precedent->suivant=courant->suivant; free(courant); }</pre>
Fin	<pre> return l; }</pre>

Chapitre 9: Les listes chaînées

72

Libération de la mémoire de la liste

- Vue que les cellules d'une liste chaînée sont allouées de manière dynamique, la libération de la mémoire allouée est obligatoire dès qu'on n'a plus besoin.
- Pour ce faire, il faut parcourir la liste et libérer chaque cellule en utilisant **free** et remettre la tête à NULL.

Chapitre 9: Les listes chaînées

73

Synthèse

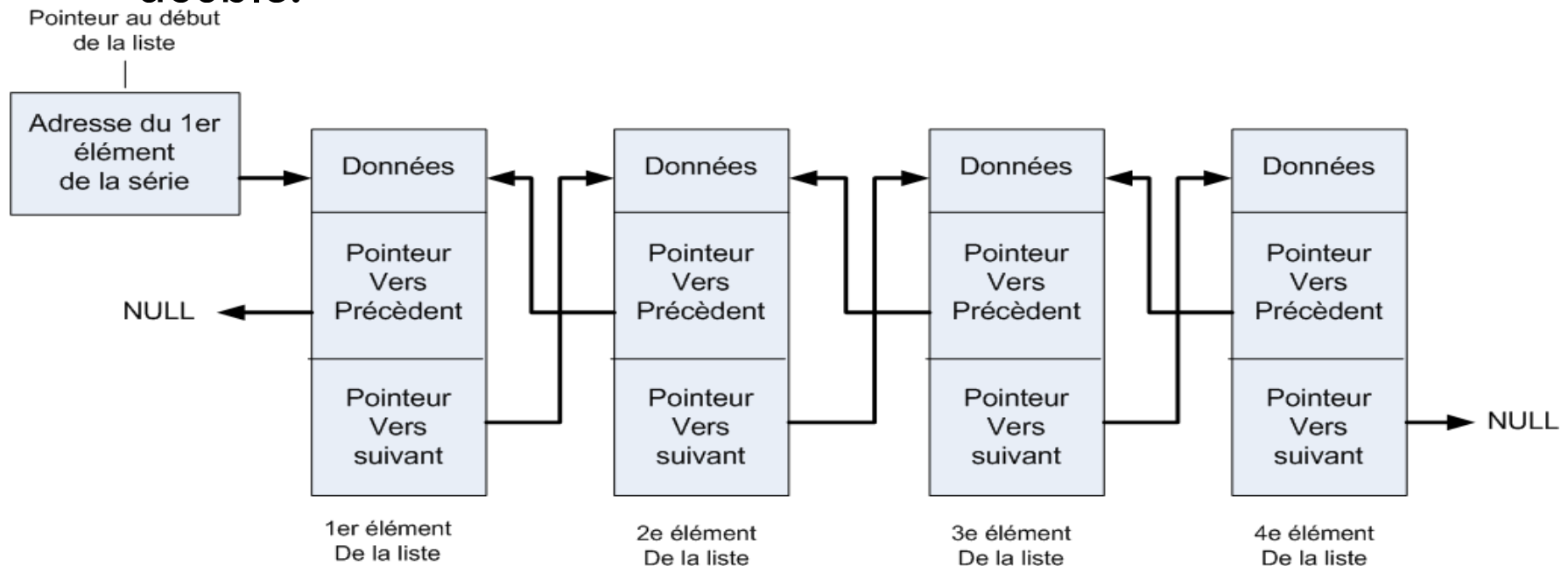
Structure	Ajout/suppression début/ au milieu	Ajout/suppression A la fin	Accès à une information connaissant son indice
Tableau dynamique	—	+	Directement
Liste chaînée	+	—	Séquentiellement par le chainage avec les pointeurs de chaque élément

Chapitre 9: Les listes chaînées

Listes chaînées double

74

- Lorsque les éléments de la liste ne contiennent pas seulement un pointeur vers l'élément immédiatement suivant (dans l'ordre logique), mais également un pointeur vers l'élément immédiatement précédent, on parle alors de liste chaînée double.



Chapitre 9: Les listes chaînées

Listes chaînées double

75

- A la différence d'une liste chaînée simple, on peut se déplacer dans une liste chaînée double, non seulement vers l'avant mais vers l'arrière, cela grâce au pointeur vers le précédent.
- On simplifie ainsi des opérations spécifiques sur la liste telles que l'insertion de nouveaux éléments ou la suppression d'éléments existants.

Les arbres binaires

Chapitre 10: les arbres binaires

77

Motivation

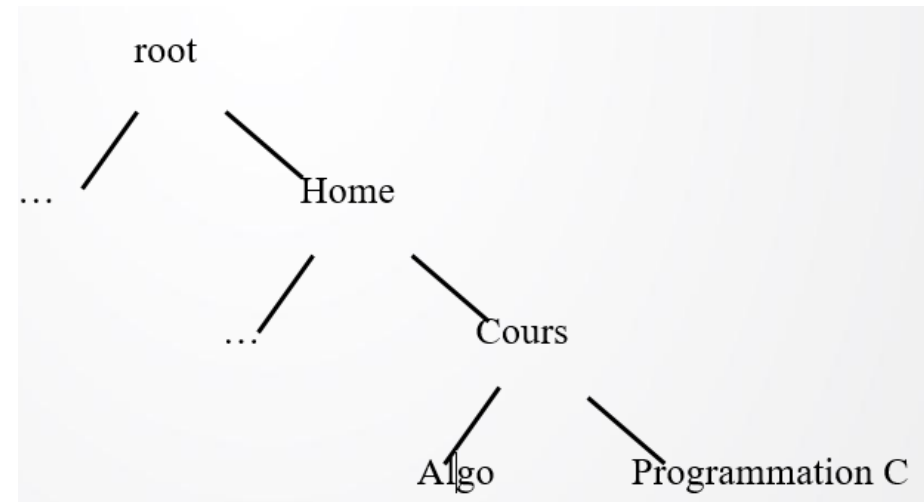
- Les structures (Tableaux, listes) sont des structures linéaires:
 - ▣ Les données sont organisées de manière ordonnée les uns à la suite des autres
 - ▣ Pour chercher un élément, nous sommes obligés de parcourir toute la structure de donnée jusqu'à le trouver.
- ➔ La recherche d'un élément dans un arbre binaire de recherche est beaucoup plus rapide que la recherche dans une structure de donnée linéaire.

Chapitre 10: les arbres binaires

78

Définition

- Un arbre est une structure de données composée d'un ensemble de nœuds.
- Chaque nœud contient les données spécifiques de l'application et des pointeurs vers d'autres nœuds (d'autres sous-arbres).
- Plusieurs traitements en informatique sont de nature arborescente tel que:
 - ▣ La représentation des expressions arithmétiques,.. Etc.
 - ▣ La hiérarchie des répertoires et des fichiers

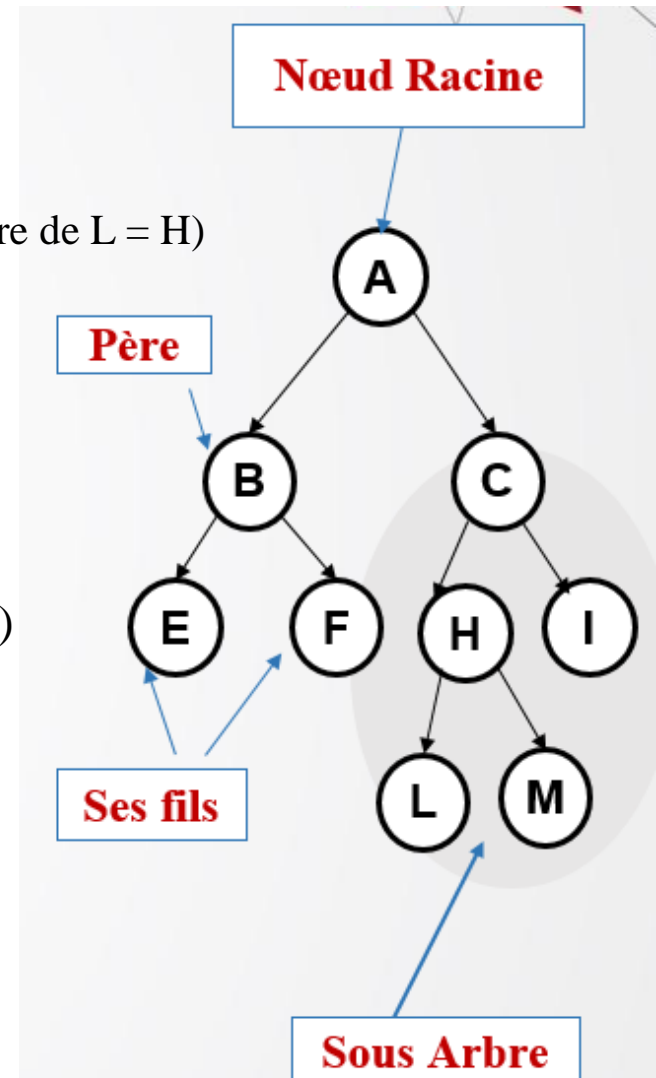


Chapitre 10: les arbres binaires

79

Terminologie

- Le prédécesseur s'il existe s'appelle **père** (père de C = A, père de L = H)
- Le successeur s'il existe s'appelle **fil**
(fils de A = { B,C }, fils de H = { L,M })
- Le nœud qui n'a pas de prédécesseur s'appelle **racine** (A)



Chapitre 10: les arbres binaires

80

Terminologie

- Le nœud qui n'a pas de successeur s'appelle **feuille** (Exemples: E,F,L,M,I)
- Un nœud **descendant** n d'un autre nœud X est tout nœud se trouvant dans le chemin partant du nœud X jusqu'à une feuille (y compris le nœud feuille).

Exemple: Les descendants de $C=\{H,I,L,M\}$, de $B=\{E,F\}$

- Un nœud **ascendant** n d'un autre nœud X est tout nœud se trouvant dans le chemin partant du nœud X jusqu'à la racine(y compris la racine).

Exemple: Les ascendants de $L=\{H,C,A \}$, $E=\{B,A\}$

Chapitre 10: les arbres binaires

81

Terminologie

- Le nœud qui n'a pas de successeur s'appelle **feuille** (Exemples: E,F,L,M,I)
- Un nœud **descendant** n d'un autre nœud X est tout nœud se trouvant dans le chemin partant du nœud X jusqu'à une feuille (y compris le nœud feuille).

Exemple: Les descendants de $C=\{H,I,L,M\}$, de $B=\{E,F\}$

- Un nœud **ascendant** n d'un autre nœud X est tout nœud se trouvant dans le chemin partant du nœud X jusqu'à la racine(y compris la racine).

Exemple: Les ascendants de $L=\{H,C,A \}$, $E=\{B,A\}$

Chapitre 10: les arbres binaires

82

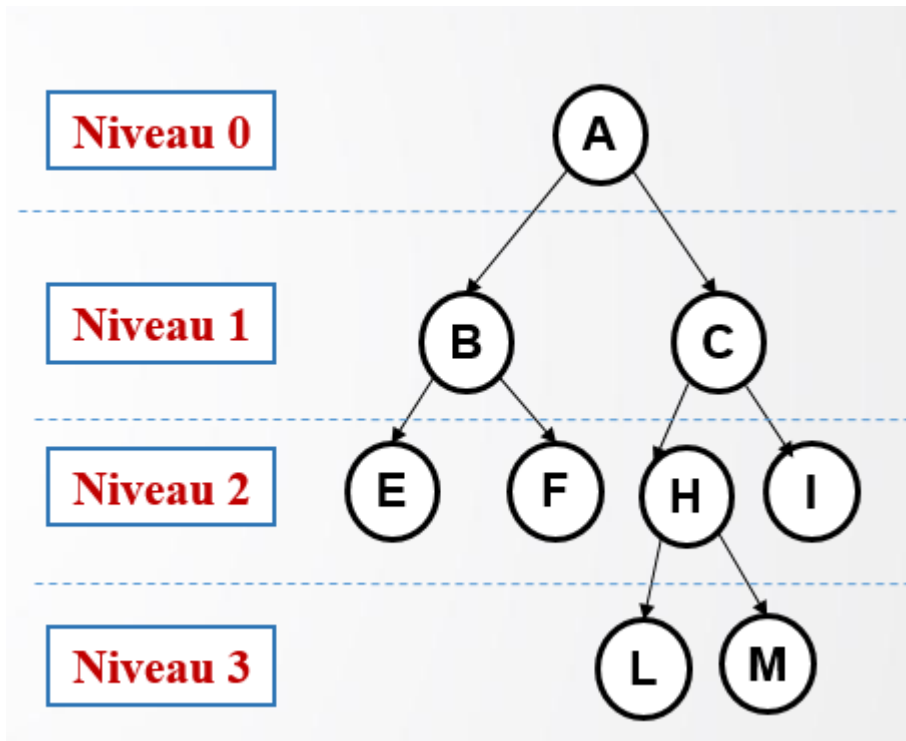
Mesures sur les arbres

➤ Taille d'un arbre

- On appelle taille d'un arbre le nombre total de nœuds de cet arbre.
- Taille de l'arbre suivant = 9
- Un arbre vide est de taille 0.

➤ Niveau d'un nœud

- Le niveau de la racine = 0
- Le niveau de chaque nœud = niveau de son père + 1
- Niveau de $\{E, F, H, I\} = 2$



Chapitre 10: les arbres binaires

83

Mesures sur les arbres

➤ Profondeur (Hauteur) d'un arbre

- C'est le niveau maximum dans cet arbre.

Profondeur de l'arbre suivant = 3

➤ Degré d'un nœud

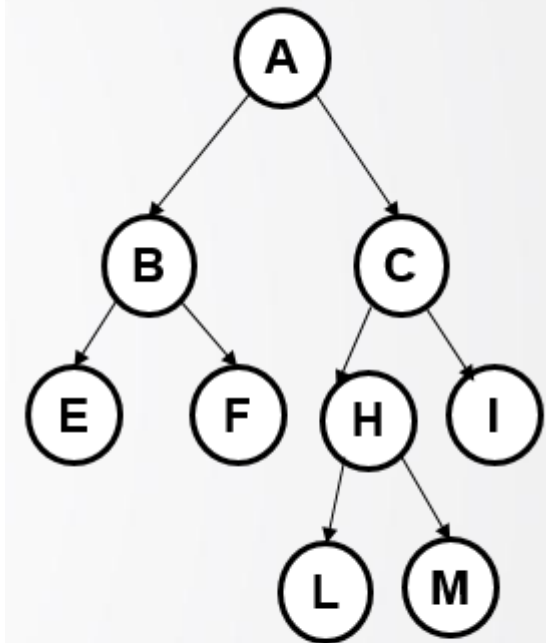
- Le degré d'un nœud est égal au nombre de ses fils.
- Degré de (A = 2, B = 2, C = 2, E = 0, H = 2)

➤ Degré d'un arbre

- C'est le degré maximum de ses nœuds.

Le degré d'un arbre binaire est égal à 2.

Si le degré d'un arbre est égal à N, l'arbre est dit **N-aire**.

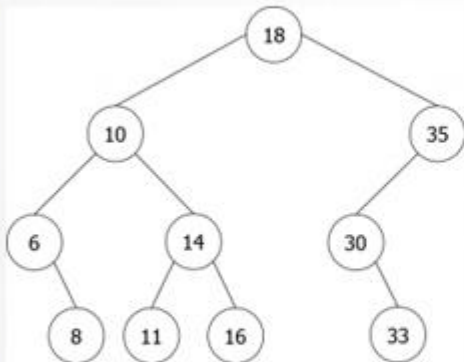


Degré de l'arbre = 2

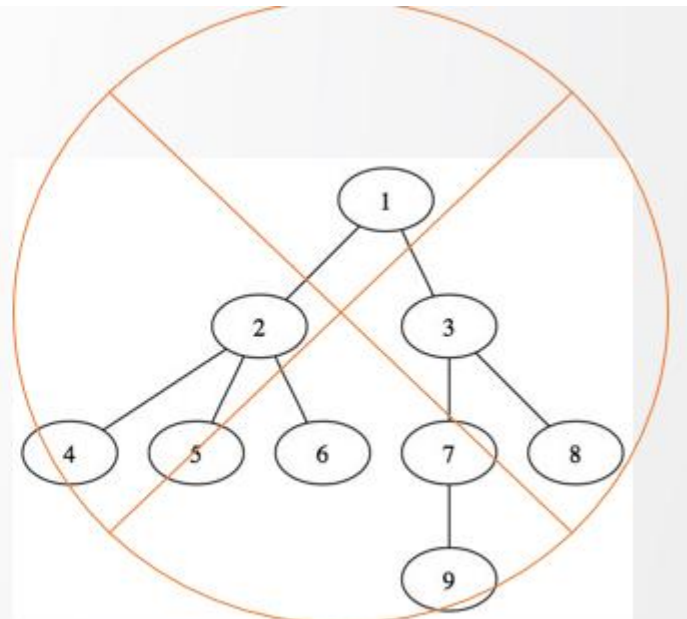
Chapitre 10: les arbres binaires

84

Un arbre **binaire** est un arbre où chaque nœud a un fils gauche, un fils droit ou les deux à la fois.
➔ c'est un arbre où le degré maximum d'un nœud est égal à 2.



Arbre binaire



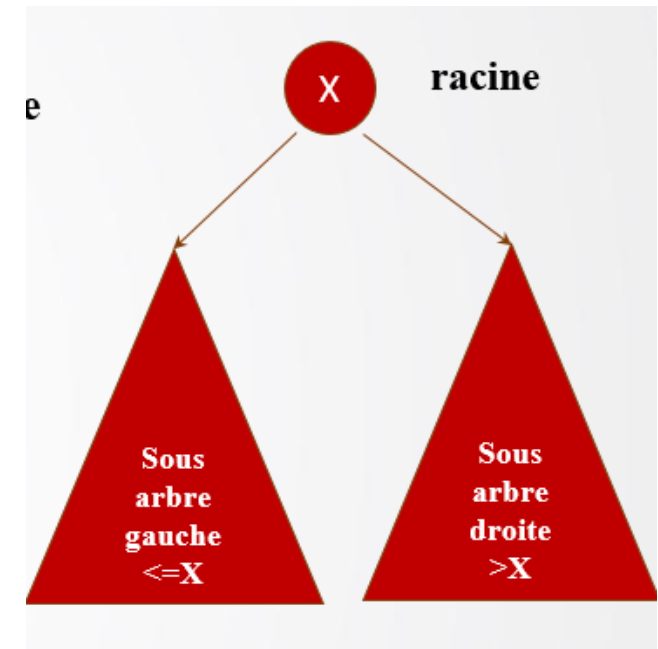
Arbre non binaire

Chapitre 10: les arbres binaires

85

Un arbre binaire A de racine X est dit **arbre binaire de recherche** (ABR) si et seulement si :

- ✓ Toute valeur associée à un nœud de son sous-arbre principal gauche est $\leq X$
- ✓ Toute valeur associée à un nœud de son sous-arbre principal droit est $> X$
- ✓ Tout sous-arbre de A est lui-même un ABR.

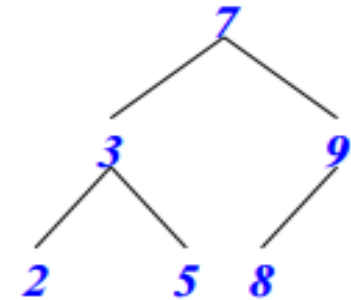
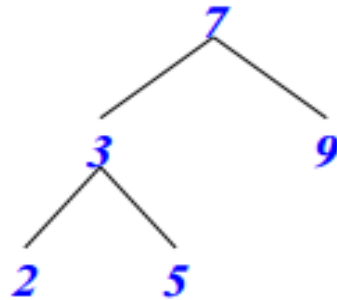


Chapitre 10: les arbres binaires

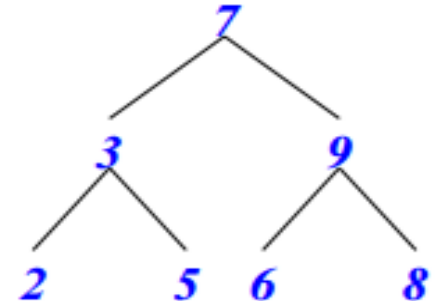
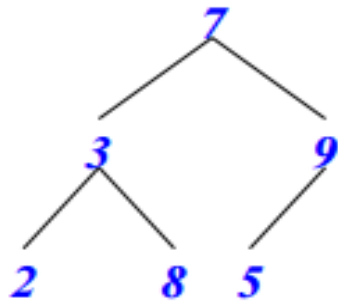
86

□ exemples

Des ABR



Des arbres non ABR



Chapitre 10: les arbres binaires

87

Structure

- 3 types de données sont stockées dans un nœud. :
 - La donnée data
 - Un pointeur de type Nœud vers le sous arbre gauche
 - Un pointeur de type Nœud vers le sous arbre droit
 - Relations entre types: Structure récursive
 - Un arbre binaire est caractérisé par une racine qui est un nœud
 - Les descendants d'un nœud sont des arbres binaires
 - définition récursive de l'arbre en fonction d'elle-même.
- Solution** : les descendants d'un nœud sont des pointeurs vers d'autres nœuds.

Chapitre 10: les arbres binaires

88

Struct Nœud

```
{  
    TYPE data;                // data peut avoir n'importe  
                               // quel type  
    Struct Nœud * FG;         // FG et FD sont deux  
                               // pointeur vers d'autres  
                               // noeuds */  
    Struct Nœud * FD;  
};
```

```
Typedef Struct Nœud * Arbre;
```


Chapitre 10: les arbres binaires

89

Le parcours

- Le parcours d'un arbre consiste à passer par tous ses nœuds pour en effectuer un traitement.
- On distingue deux types de parcours :
 - ✓ Parcours en profondeur
 - ✓ Parcours en largeur

Chapitre 10: les arbres binaires

90

parcours en profondeur

Dans un parcours en profondeur, Commençant par la racine:

1. On descend le plus profondément possible dans l'arbre puis
2. Une fois qu'une feuille est atteinte, on remonte pour explorer les autres branches en commençant par la branche "la plus basse" parmi celles non encore parcourues.

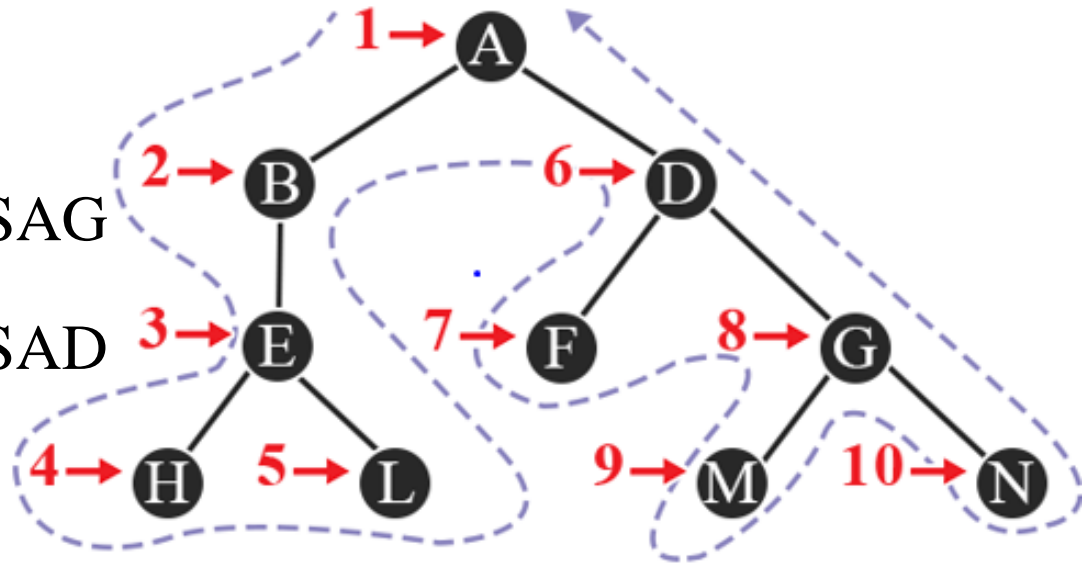
Chapitre 10: les arbres binaires

91

Parcours Préfixé

La racine est traitée en premier

- 1 Traiter la racine
- 2 Parcours préfixé du SAG
- 3 Parcours préfixé du SAD



□ Parcours préfixé : A, B, E, H, L, D, F, G, M, N.

Chapitre 10: les arbres binaires

92

Parcours Préfixé

```
void ParcoursPrefixe ( Arbre R)
{
    if (R !=NULL)                // Si l'arbre n'est pas vide
    {
        printf("\n Racine = %d", R->X);    // Afficher la valeur de racine
        ParcoursPrefixe(R->FG);            // Appel récursif
        ParcoursPrefixe(R->FD);
    }
}
```

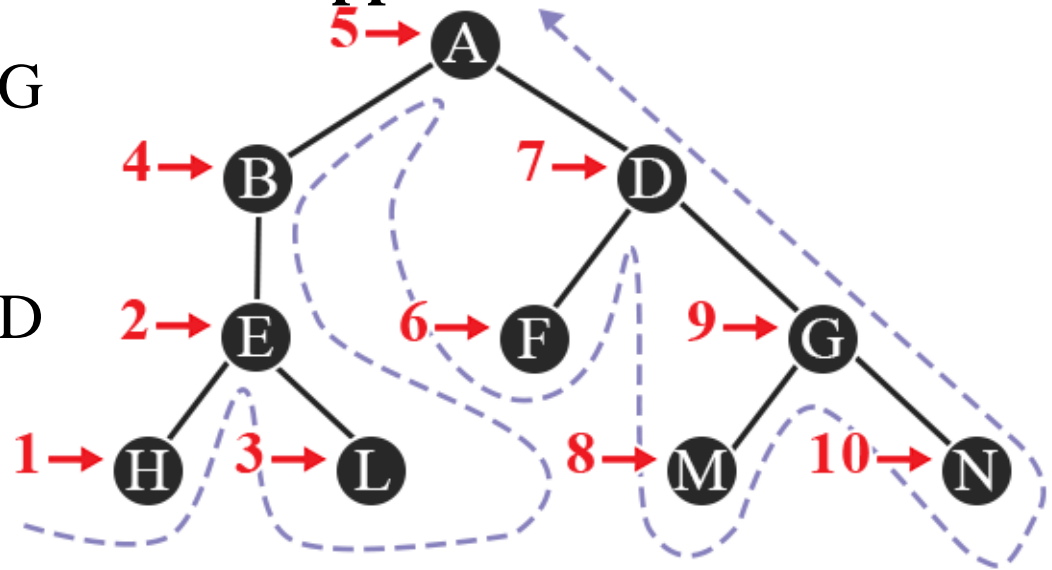
Chapitre 10: les arbres binaires

93

Parcours infixé

La racine est traitée entre les deux appels récurifs

- 1 Parcours Infixé du SAG
- 2 Traiter la racine
- 3 Parcours Infixé du SAD



□ Parcours infixé : H, E, L, B, A, F, D, M, G, N.

Chapitre 10: les arbres binaires

94

Parcours infixé

```
void ParcoursInfixe ( Arbre R)
{
    if (R !=NULL)
    {
        ParcoursInfixe(R->FG);
        printf("\n Racine = %d", R->X);
        parcoursInfixe(R->FD);
    }
}
```

Chapitre 10: les arbres binaires

95

Parcours infixé

Exemple d'application:

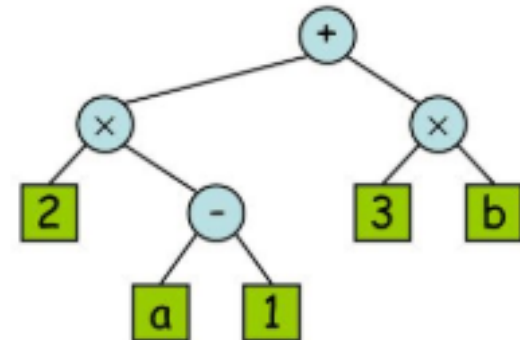
Une expression arithmétique peut être représentée par un arbre.

Pour évaluer l'expression, il faut partir du bas et effectuer les calculs en remontant.

Arbre de l'expression arithmétique:

$((2*(a-1))+(3*b))$

- Nœuds intérieurs: **opérateurs**
- Nœuds extérieurs (feuilles) : **opérandes**



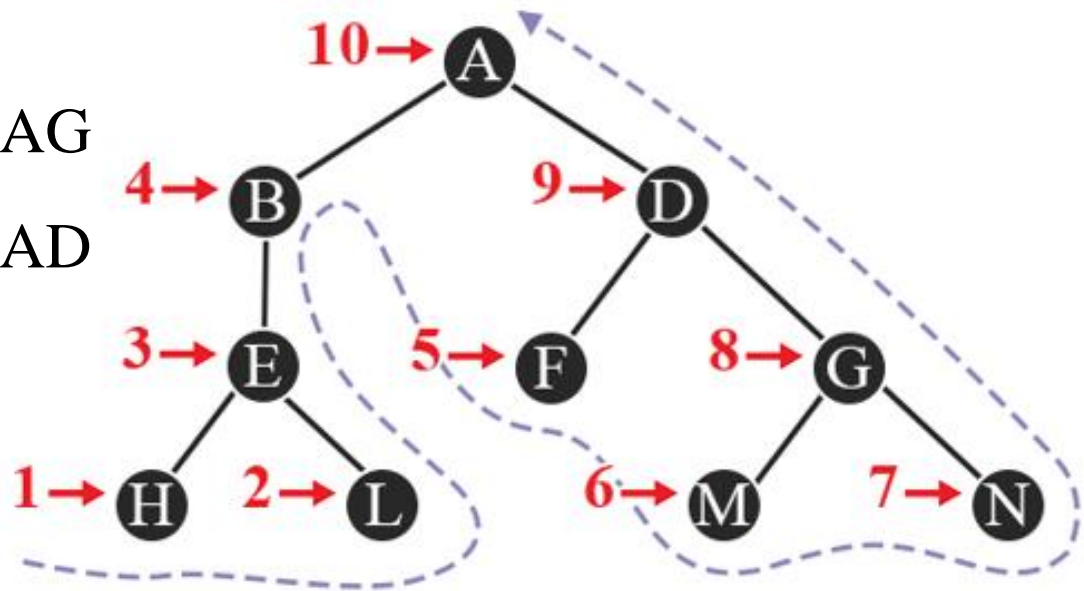
Chapitre 10: les arbres binaires

96

Parcours Postfixé

La racine est traitée après les deux appels récuratifs

- 1 Parcours Postfixé du SAG
- 2 Parcours Postfixé du SAD
- 3 Traiter la racine



□ Parcours postfixé : H, L, E, B, F, M, N, G, D, A.

Chapitre 10: les arbres binaires

97

Parcours Postfixé

```
void ParcoursPostfixe ( Arbre R)
{
    if (R !=NULL)
    {
        ParcoursPostfixe(R->FG);
        parcoursPostfixe(R->FD);
        printf("\n Racine = %d", R->X);
    }
}
```

Chapitre 10: les arbres binaires

Ajout d'un nœud

98

- Algo INSERT(v) // insère la clé v dans l'arbre **itératif**
 - ▣ $x \leftarrow \text{racine}$
 - ▣ si $x = \text{null}$ alors initialiser avec une racine de clé v et retourner
 - ▣ tant que vrai faire
 - si $v = \text{cle}(x)$ alors retourner
 - si $v < \text{cle}(x)$
 - alors si $\text{gauche}(x) = \text{null}$
 - alors attacher nouvel enfant gauche de x avec clé v et retourner
 - sinon $x \leftarrow \text{gauche}(x)$
 - sinon si $\text{droit}(x) = \text{null}$
 - alors attacher nouvel enfant droit de x avec clé v et retourner
 - sinon $x \leftarrow \text{droit}(x)$

Chapitre 10: les arbres binaires

Ajout d'un nœud

99

➤ Fonction récursive d'ajout d'un élément :

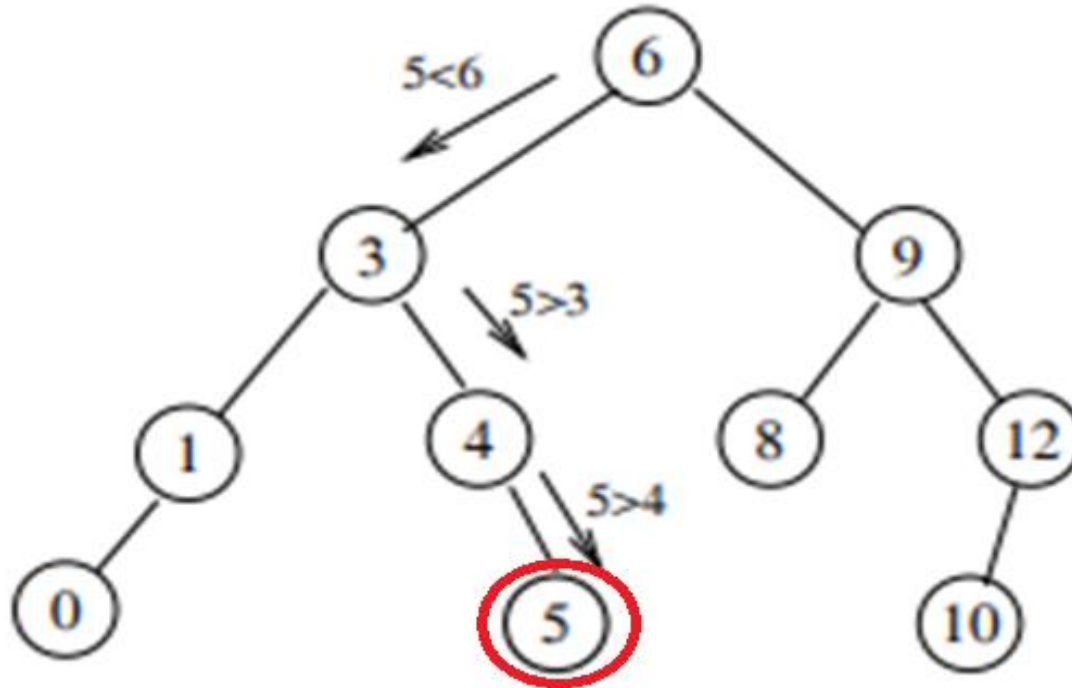
- ✓ soit `nouv` un pointeur sur le nouveau nœud à insérer.
- ✓ soit `R` un pointeur sur le nœud racine.
- 1. **Si** `R == NULL` **alors** la racine devient l'adresse du nouveau nœud (`nouv`)
- 2. **Si** valeur de `nouv` \leq valeur de la Racine (`R`) **alors**
Ajouter l'élément dans le sous arbre gauche ayant pour racine le fils gauche de l'ancienne racine
- 3. **Si** valeur de `nouv` $>$ valeur de la Racine (`R`) **alors**
Ajouter l'élément dans le sous arbre droit ayant pour racine le fils droit de l'ancienne racine

Chapitre 10: les arbres binaires

Ajout d'un nœud

100

- Exemple: Ajout d'un nœud ayant la valeur 5 à l'arbre



Chapitre 10: les arbres binaires

Ajout d'un nœud

101

```
// AjoutNoeudABR est une fonction qui ajoute
// un noeud à un arbre binaire de recherche
Arbre AjoutNoeudABR ( Arbre R, int val )
{
    struct Noeud * nouv;
    if (R == NULL)
    {
        nouv = (struct Noeud *)malloc (sizeof(struct Noeud)); // allouer de l'espace mémoire pour le nouveau élément à insérer
        nouv -> X = val;
        nouv -> FG = NULL;
        nouv -> FD = NULL;

        R = nouv;
    }
    else
    {
        if ( val <= R->X)           // si la valeur du nouveau noeud est <= à la valeur de R
                                   // Insérer nouv dans le sous arbre gauche de R
            R->FG = AjoutNoeudABR ( R->FG , val);

        else                       // la valeur de nouv est > à la valeur de R
                                   // Insérer nouv dans le sous arbre droite de R
            R->FD = AjoutNoeudABR( R->FD, val);
    }

    return R;
}
```

Chapitre 10: les arbres binaires

Ajout d'un nœud

102

```
// AjoutNoeudABR est une fonction qui ajoute
// un nœud à un arbre binaire de recherche
void AjoutNoeudABR ( Arbre *R, int val )
{
    struct Noeud * nouv;
    if ((*R) == NULL)
    {
        nouv = (struct Noeud *)malloc (sizeof(struct Noeud)); // allouer de l'espace mémoire pour le nouveau élément à insérer
        nouv -> X = val;
        nouv -> FG = NULL;
        nouv -> FD = NULL;

        (*R) = nouv;
    }
    else
    {
        if ( val <= (*R)->X) // si la valeur du nouveau nœud est <= à la valeur de R
                           // Insérer nouv dans le sous arbre gauche de R
            AjoutNoeudABR ( &(*R)->FG , val);

        else // la valeur de nouv est > à la valeur de R
             // Insérer nouv dans le sous arbre droite de R
            AjoutNoeudABR( &(*R)->FD, val);
    }
}
```

Manipulation des Fichiers

Chapitre 11: Les fichiers

104

Motivation

Un programme a en général besoin de:

- ☐ Sauvegarder des variables qui sont supprimées de la mémoire vive une fois le programme arrêté (impossible d'accéder à leurs valeurs)
- ☐ Lire des données (texte, nombre, image, son)
- ☐ Sauvegarder des résultats. Cela se fait en lisant et en écrivant dans des **fichiers**

Chapitre 11: Les fichiers

105

Qu'est ce qu'un fichier?

- ☐ Données organisées, support de sauvegarde (disquette, disque dur, CD/DVD)
- ☐ Peut contenir du texte, une vidéo, des données pour des applications
- ☐ Mémoire de masse persistante (non effacée quand hors-tension)

Chapitre 11: Les fichiers

106

Opérations Standards

- ☐ Ouvrir un fichier: lui associer une variable que l'on appelle descripteur de fichier
- ☐ Lire ou écrire des informations à partir de ce descripteur avec des fonctions spécialement prévues pour les fichiers
- ☐ Fermer le fichier: indiquer qu'on a terminé de travailler avec ce fichier
- ☐ Supprimer un fichier
- ☐ Renommer un fichier

Chapitre 11: Les fichiers

107

Pour manipuler un fichier, on a besoin d'un certain nombre d'informations comme:

- ☐ Le mode d'accès à ce fichier
- ☐ L'adresse de la mémoire tampon où se trouve le fichier
- ☐ La position de la tête de lecture/écriture
- ☐ État d'erreur

Toutes ces informations sont rassemblées dans le type **FILE**

défini dans la bibliothèque **<stdio.h>**

Un objet de type **FILE*** est nommé un **flot de données**

Chapitre 11: Les fichiers

types des fichiers

108

Fichiers Textes

Un fichier texte est un fichier dont le contenu représente une suite de caractères lisibles



Dans un fichier texte, on enregistre un **texte lisible**

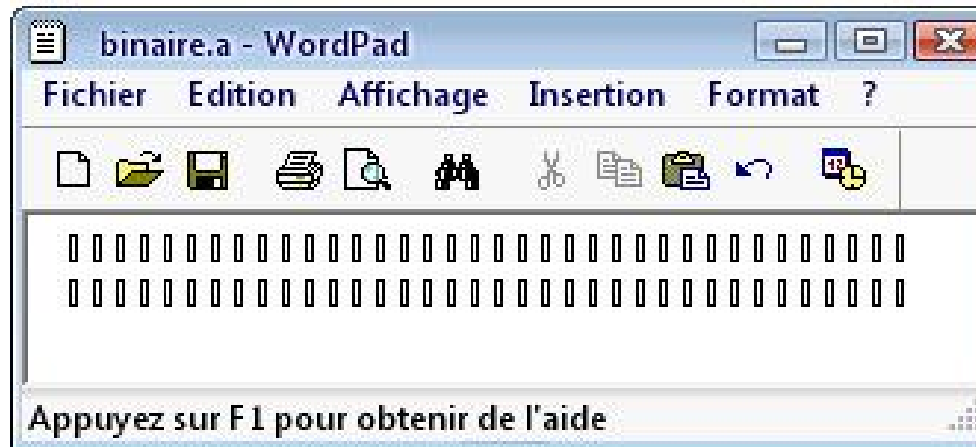
Chapitre 11: Les fichiers

types des fichiers

109

Fichiers Binaires

Tout fichier qui n'est pas de type texte est un fichier binaire



Dans un fichier binaire, on peut enregistrer n'importe quelle donnée (texte, image, son) mais qui n'est **pas interprétable**

Chapitre 11: Les fichiers

Ouvrir un fichier

110

- ❑ Pour pouvoir lire et écrire dans un fichier, il faut commencer par l'ouvrir
Syntaxe: `f=fopen("chemin", "mode");`
- ❑ **f** est une variable de type **FILE***
- ❑ La fonction **fopen** ouvre un fichier et lui associe un flot de données (elle renvoie un pointeur sur le fichier)
- ❑ Le premier paramètre est le **chemin** d'accès au fichier

Exemple:

"monFich.txt" s'il est situé dans le même dossier que l'exécutable

" C:\mondossier\monFich.txt" s'il est situé n'importe où ailleurs sur le disque dur

- ❑ Le deuxième paramètre est une chaîne de caractère spécifiant le **mode d'accès**

Chapitre 11: Les fichiers

Ouvrir un fichier

111

❑ Modes d'Accès à un Fichier Texte

"r"	ouverture d'un fichier texte en lecture
"w"	ouverture d'un fichier texte en écriture
"a"	ouverture d'un fichier texte en écriture à la fin
"r+"	ouverture d'un fichier texte en lecture/écriture
"w+"	ouverture d'un fichier texte en lecture/écriture
"a+"	ouverture d'un fichier texte en lecture/écriture à la fin

Chapitre 11: Les fichiers

Ouvrir un fichier

112

□ Modes d'Accès à un Fichier Binaire

"rb"	ouverture d'un fichier binaire en lecture
"wb"	ouverture d'un fichier binaire en écriture
"ab"	ouverture d'un fichier binaire en écriture à la fin
"r+b"	ouverture d'un fichier binaire en lecture/écriture
"w+b"	ouverture d'un fichier binaire en lecture/écriture
"a+b"	ouverture d'un fichier binaire en lecture/écriture à la fin

Chapitre 11: Les fichiers

Ouvrir un fichier

113

- ☐ Si le mode contient la lettre **r**, le fichier doit exister
- ☐ Si le mode contient la lettre **w**, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu
- ☐ Si le mode contient la lettre **a**, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent

Chapitre 11: Les fichiers

Ouvrir un fichier

114

Mode		Accès	Position	Comportement	
Fichier texte	Fichier binaire			Si le fichier existe	Si le fichier n'existe pas
r	rb	Lecture	Début	***	Retourne le pointeur NULL
w a	wb ab	Écriture Écriture	Début Fin	Mis à zéro ***	Création Création
r+ w+ a+	rb+ wb+ ab+	Lecture et Écriture Ajout	Début Début fin	Mis à zéro Mis à zéro ***	Retourne le pointeur NULL Création Création

Chapitre 11: Les fichiers

Ouvrir un fichier

115

Juste après l'ouverture du fichier, il faut impérativement vérifier si l'ouverture est réussie ou non:

- ☐ Si le pointeur vaut **NULL**, l'ouverture a échoué
- ☐ Sinon (s'il vaut autre chose que **NULL**), l'ouverture a réussi

On va donc suivre systématiquement le schéma suivant:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "r+");
    if (fichier != NULL)
    {
        // On peut lire et écrire dans le fichier
    }
    else
    {
        // On affiche un message d'erreur si on veut

        printf("Impossible d'ouvrir le fichier test.txt");
    }

    return 0;
}
```

Chapitre 11: Les fichiers

Fermer un fichier

116

- ❑ Après les manipulations, on doit annuler la liaison entre le fichier et le flot de données via la fonction **fclose**

Syntaxe: fclose(f);

- ❑ Cette fonction a pour rôle de libérer la mémoire, c'est-à-dire supprimer le fichier chargé dans la mémoire vive
- ❑ Si on oublie de libérer la mémoire, le programme risque à la fin de prendre énormément de mémoire qu'il n'utilise plus
- ❑ La fonction **fclose** retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur)

Chapitre 11: Les fichiers

Fermer un fichier

117

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "r+");

    if (fichier != NULL)
    {
        // On lit et on écrit dans le fichier

        // ...

        fclose(fichier); // On ferme le fichier qui a été ouvert
    }

    return 0;
}
```

Chapitre 11: Les fichiers

Ecrire dans un fichier texte

118

☐ Ecriture Formatée dans un Fichier Texte

La fonction **fprintf**, analogue à **printf**, permet d'écrire des données dans un fichier texte à part le fait qu'on doit indiquer le fichier en premier paramètre

Syntaxe:

int fprintf(f,"chaîne de contrôle",expression-1, ..., expression-n)

La valeur de retour indique le nombre de caractères écrits

Exemple: fprintf(f, " %s %s %d\n",nom, prenom, age);

Remarques:

- ☐ Un **\n** écrit dans un fichier provoquera un retour à la ligne
- ☐ Le caractère de fin de fichier est ajouté automatiquement, pas besoin de le mettre

Chapitre 11: Les fichiers

Ecrire dans un fichier texte

119

□ Syntaxe :

```
char * fputs(char * <pointeur_tampon>,  
             FILE * <pointeur_fichier>)
```

/*<pointeur_tampon> désigne ici l'adresse de l'emplacement mémoire qui contient la chaîne de caractère à écrire. */

□ Exemple :

```
char chaine[ ]=" exemple d'écriture ";
```

```
fputs (chaine,fp);
```

//Ou bien

```
fputs(" exemple d'écriture ",fp);
```

//Écrire " exemple d'écriture " dans le fichier référencié par fp

Chapitre 11: Les fichiers

Lire à partir d'un fichier texte

120

❑ Lecture Formatée dans un Fichier Texte

La fonction **fscanf**, analogue à **scanf**, permet de lire des données dans un fichier texte

Syntaxe:

int fscanf(f,"chaîne de contrôle",argument-1, ..., argument-n);

fscanf fournit le nombre de valeurs lues convenablement ou la valeur **EOF** si une erreur s'est produite ou si une fin de fichier a été rencontrée avant qu'une seule valeur ait pu être lue

Exemple: fscanf(f, " %s %s %d\n",nom, prenom, &age);

Chapitre 11: Les fichiers

Lire à partir d'un fichier texte

121

- Lecture caractère par caractère

- **Syntaxe :**

```
int putc(int<caractère>,FILE*<nom_pointeur>)
```

```
int fputc(int<caractère>,FILE*< nom_pointeur>)
```

Les deux
expressions
sont équivalentes

- **Exemple :**

```
fputc( 'A',fp); /* écriture de 'A' dans le fichier fp  
dans la position du traitement courante.*/
```

Chapitre 11 : Les fichiers

Lire à partir d'un fichier texte

122

- Lecture de chaîne de caractères

- **Syntaxe :**

`char* fgets(char* <pointeur_tampon>, int <nbre>, FILE * <pointeur_fichier>)`

- `fgets` lit sur `<pointeur_fichier>` un nombre de caractères jusqu'à ce que l'un de ces événements se produise :
 - `<nbre>-1` caractères ont été lus.
 - rencontre d'un retour-chariot `'\n'`, fin du fichier atteinte.

- **Exemple:**

```
FILE * fichier;
```

```
char texte[25];
```

```
fichier=fopen(" projet.dat"," r" );
```

```
fgets(texte,25,fichier);
```

```
/* Lire au plus 24 caractères et les ranger dans texte. Caractère '\0'  
sera placé dans la dernière case. */
```

Chapitre 11: Les fichiers

Lire/Ecrire dans un fichier binaire

123

❑ Entrées/Sorties Binaires

Elles sont plus efficaces que les entrées/sorties standard car les données sont transférées sans transcodage

Inconvénient: les fichiers binaires ne sont pas portables car le codage dépend de la machine

Chapitre 11: Les fichiers

Ecrire dans un fichier binaire

124

- ❑ La fonction qui permet d'écrire des données à partir d'un fichier binaire:

size_t fwrite(void *pointeur, size_t taille, size_t nombre, FILE *f);

- ❑ **pointeur**: zone de mémoire où se trouvent les éléments à écrire dans le fichier
- ❑ **taille**: taille en octets d'un élément
- ❑ **nombre**: nombre d'éléments à écrire
- ❑ **f**: fichier où aura lieu l'écriture

- ❑ La valeur de retour indique le nombre d'éléments effectivement écrits

Exemple: pour écrire une donnée de type **Etudiant** et la sauvegarder dans le fichier **f**

fwrite(&e, sizeof(Etudiant), 1, f);

Chapitre 11: Les fichiers

Lire à partir d'un fichier binaire

125

- ❑ La fonction qui permet de lire des données à partir d'un fichier binaire:

size_t fread (void *pointeur, size_t taille, size_t nombre, FILE *f);

- ❑ **pointeur**: zone de mémoire où sera stockée les données lus à partir du fichier
- ❑ **taille**: taille en octets d'un élément
- ❑ **nombre**: nombre d'éléments à lire
- ❑ **f**: fichier où aura lieu la lecture.

- ❑ La valeur de retour indique le nombre d'éléments effectivement lus

Exemple: pour lire une donnée de type **Etudiant** et la sauvegarder dans la variable **e**


fread(&e, sizeof(Etudiant), 1, f);

Chapitre 11: Les fichiers

Accès direct(Fonctions de positionnement)

126

- 
- Syntaxe :
 - `Int fseek (FILE*<pointeur_fichier>,long<offset>,int<base>);`

- 
- "fseek" déplace le pointeur de <offset> cases à partir de <base>. Si <offset> est positif, le déplacement sera vers la fin du fichier, sinon vers de début.

- 
- <base> s'exprime en fonction de trois pseudo-constantes définies dans <stdio.h>

- 
- SEEK_SET équivaut à 0 //début du fichier
 - SEEK_CUR équivaut à 1 //position courante
 - SEEK_END équivaut à 2 //fin du fichier

Chapitre 11: Les fichiers

La fonction « fseek »

127

□ Exemple :

```
FILE*fp;
```

```
fseek(fp,25,0) ;
```

*//déplacer le pointeur fp de 25 cases à partir du
//début du fichier*

//Ou bien

```
fseek(fp,25,SEEK_SET) ;
```

Chapitre 11: Les fichiers

La fonction « ftell »

128

- Elle retourne, sous forme de valeur "long" , la position actuelle du pointeur de position au début du fichier.

- **Syntaxe :**

long ftell (FILE * <pointeur_fichier>)

- **Exemple:**

```
FILE* fp;
```

```
long pos;
```

```
pos=ftell(fp) ; //pos contient la valeur de la  
// position courante de fp
```


Chapitre 11: Les fichiers

La fonction « rewind »

129

- Cette fonction ramène le pointeur de position en début de fichier.

- **Syntaxe :**

```
void rewind (FILE*<pointeur_fichier>)
```

```
// rewind positionne le <pointeur_fichier> au
```

```
//début du fichier
```

- **Exemple:**

```
FILE*fp;
```

```
rewind(fp) ; //fp est placé au début du fichier
```

Chapitre 11: Les fichiers

Renommer un fichier

130

- ❑ La fonction **rename** permet de renommer un fichier

Prototype:

int rename(const char* ancienNom, const char* nouveauNom);

- ❑ La fonction renvoie 0 si elle a réussi à renommer, sinon une valeur différente

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    rename("test.txt", "1A.txt");

    return 0;
}
```

Chapitre 11: Les fichiers

Supprimer un fichier

131

❑ La fonction **remove** permet de supprimer un fichier

Prototype:

int remove(const char* fichier);

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    remove("1A.txt");

    return 0;
}
```

FIN

