

Informatique 4

SMA-S4
FP Khouribga

Chapitre 1

Les Pointeurs en C

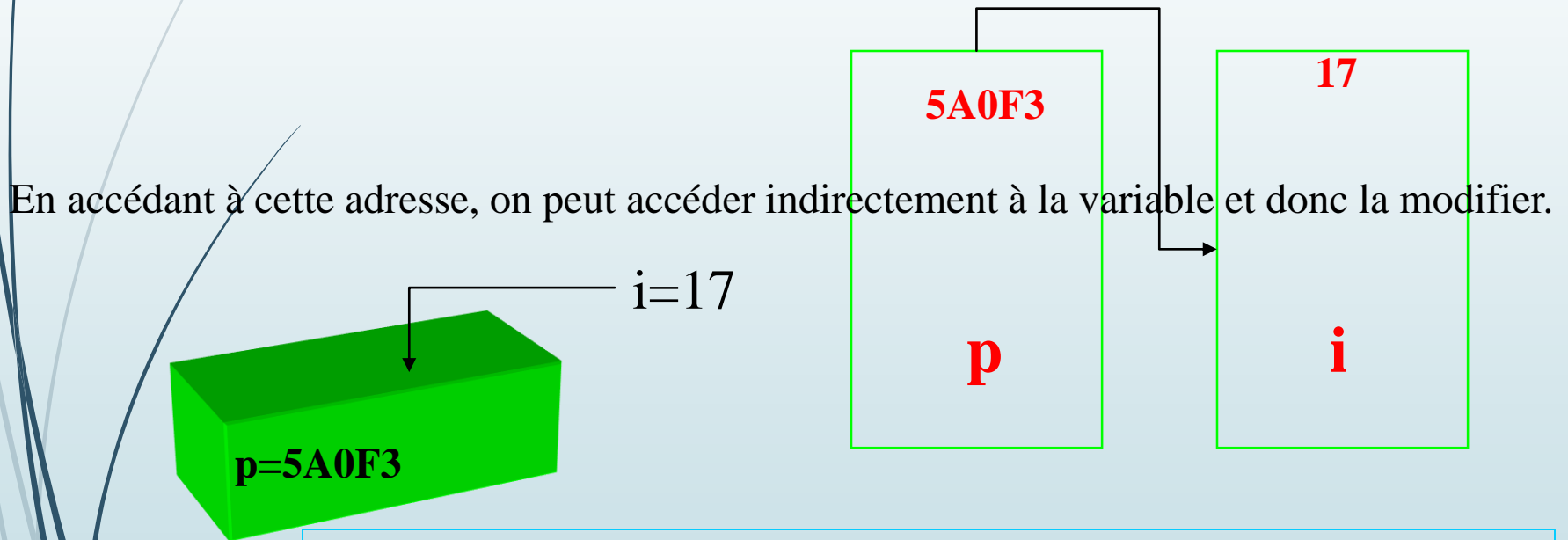
- La déclaration de pointeurs
- Valeurs pointées et adresses
- Passage de paramètres de fonction par référence
- Pointeurs et tableaux

Les pointeurs, c'est quoi?

3

Un pointeur est une variable particulière, dont la valeur est l'adresse d'une autre variable.

- Pointeur p: valeur 5A0F3 (adresse hexadécimale)
- Adresse 5A0F3: valeur 17 (correspondant à la valeur d'un entier i)



Un pointeur est une adresse mémoire. On dit que le pointeur p pointe vers i, puisque p pointe vers l'emplacement mémoire où est enregistrée i.

Les pointeurs: pourquoi ?

4

- Les pointeurs sont nécessaires pour:
 - ◆ effectuer les appels par référence (i.e. écrire des fonctions qui modifient certains de leurs paramètres)
 - ◆ manipuler des structures de données dynamiques (liste, pile, arbre,...)
 - ◆ allouer dynamiquement de la place mémoire

Déclaration de Pointeurs

5

Le symbole * est utilisé entre le type et le nom du pointeur

■ Déclaration d'un entier:

```
int i;
```

► Déclaration d'un pointeur vers un entier:

```
int *p;
```

Exemples de déclarations de pointeurs

```
int *pi;          /* pi est un pointeur vers un int
                  *pi désigne le contenu de l'adresse */
float *pf;        /* pf est un pointeur vers un float */

char c, d, *pc;    /* c et d sont des char */
                  /* pc est un pointeur vers un char */
double *pd, e, f; /* pd est un pointeur vers un double */
                  /* e et f sont des doubles */
double **tab;     /* tab est un pointeur pointant sur un pointeur qui
                  pointe sur un flottant double */
```

Opérateurs unaires pour manipuler les pointeurs, & (adresse de) et * (contenu)

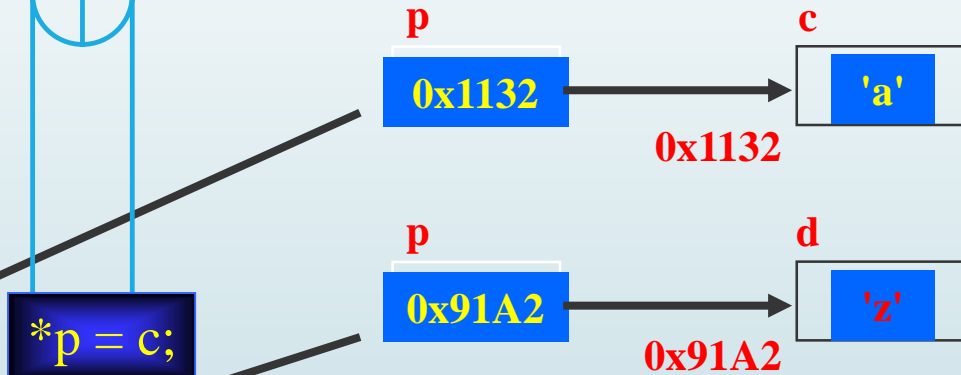
6

Exemple: `int i = 8;`
`printf("VOICI i: %d\n",i);`
`printf("VOICI SON ADRESSE EN HEXADECIMAL: %p\n",&i);`

```
void main(void)
{
    char c = 'a', d = 'z';
    char*p;

    p = &c;
    printf("%c\n", *p);
    p = &d;
    printf("%c\n", *p);
}
```

nom_de_Pointeur = &nom_de_variable



L'opérateur * (“valeur pointée par”)
reçoit l'adresse de c; donc pointe sur c.

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

```
int main() {
```

```
    int *p, x, y;
```

```
    p = &x;                /* p pointe sur x */
```

```
    x = 10;                /* x vaut 10 */
```

```
    y = *p - 1;    printf(" y= *p - 1 =?  = %d\n" , y);
```

y vaut ?

```
    *p += 1;    printf(" *p += 1 =? *p = x= ?  = %d %d\n" , *p, x);
```

x vaut ?

```
    (*p)++;    printf(" (*p)++ =? *p = x= ?  = %d %d alors y=%d \n" , *p, x, y);
```

incrémente aussi de 1 la variable pointée par p, donc x vaut ??.
y vaut 9

```
    *p=0;    printf(" *p=0 x=? = %d\n" , x);
```

comme p pointe sur x, maintenant x vaut ?

```
    p++; *p=20; printf(" *p++ x=? = %d\n" , x);
```

comme p ne pointe plus sur x, x vaut tjr ?

```
}
```

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

```
int main() {
```

```
    int *p, x, y;
```

```
    p = &x;
```

```
    x = 10;
```

```
    y = *p - 1;
```

```
    printf(" y= *p - 1 =?  = %d\n" , y);
```

```
    *p += 1;    printf(" *p += 1 =? *p = x= ?  = %d %d\n"
, *p, x);
```

```
    (*p)++;
```

```
    printf(" (*p)++ =? *p = x= ?  = %d %d  alors y=%d \n"
, *p, x, y);
```

```
    *p=0;
```

```
    printf(" *p=0  x=? =  %d\n" , x);
```

```
    (*p)++; *p=20;
```

```
    printf(" *p++  x=? =  %d\n" , x);
```

```
    *p=0;
```

```
    *(p++); *p=20;
```

```
    printf(" *(p++) x=? =  %d\n" , x);
```

```
}
```

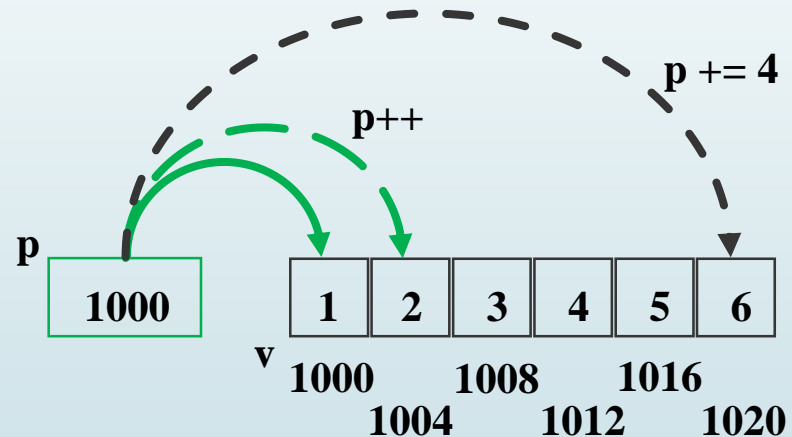
Explications

Utiliser des pointeurs

- On peut donc accéder aux éléments par pointeurs en faisant des calculs d'adresses (addition ou soustraction)

```
long v[6] = { 1,2,  
             3,4,5,6 };  
long *p;  
  
p = v;  
printf("%ld\n", *p);  
p++;  
printf("%ld\n", *p);  
p += 4;  
printf("%ld\n", *p);
```

1
2
6



* et ++

*p++ signifie:

*p++ trouver la valeur pointée

*p++ passer à l'adresse suivante

(*p)++ signifie:

(*p)++ trouver la valeur pointée

(*p)++ incrémenter cette valeur (sans changer le pointeur)

*(p++) signifie:

*(p++) passer à l'adresse suivante

*(p++) trouver valeur de nouvellement pointée

*++p signifie:

*++p incrémenter d'abord le pointeur

*++p trouver la valeur pointée



Passage des paramètres par valeur et par adresse

11

Syntaxe qui conduit à une erreur:

```
#include <stdio.h>

void echange(int x,int y)
{
    int tampon;
    tampon = x;
    x = y;
    y = tampon;
}

void main()
{
    int a = 5 , b = 8;
    echange(a,b);
    printf(" a=%d\n ", a );
    printf(" b=%d\n ", b );
}
```

PASSAGE DES PARAMETRES
PAR VALEUR

a et b:
variables
locales à
main(). La
fonction
echange ne
peut donc pas
modifier leur
valeur. On le
fait donc en
passant par
l'adresse de
ces variables.

a = ?
b = ?

Syntaxe correcte:

```
#include <stdio.h>

void echange(int *x,int *y)
{
    int tampon;
    tampon = *x;
    *x = *y;
    *y = tampon;
}

void main()
{
    int a = 5 , b = 8 ;
    echange(&a,&b);
    printf(" a=%d\n ", a );
    printf(" b=%d\n ", b );
}
```

PASSAGE DES PARAMETRES
PAR ADRESSE

a = ?
b = ?

Passage de paramètres de fonction par référence ou adresse

- Quand on veut modifier la valeur d'un paramètre dans une fonction, il faut passer ce paramètre par référence ou adresse
- En C, cela se fait par pointeur

```
void change ( int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Appel des fonctions par référence

13

```
#include <stdio.h>
void change(int* p);
void main(void)
{
  int var = 5;
  change(&var);
  printf("main: var = %d\n",var);
}

void change(int* p)
{
  *p *= 100;
  printf("change: *p = %d\n",*p);
}
```



change: *p = 500
main: var = 500

```
#include <stdio.h>

void somme(int , int , int *);
int modif(int , int *, int *);
void main(){
    int a, b, c;
    a = 2;  b = 8;
    somme(a, b, &c);
    printf("Somme de a=%d et b=%d : %d\n",a, b, c);
    a = modif(a, &b, &c);
    printf(" Modif : a=%d , b=%d et c= %d\n",a, b, c);
}

void somme(int x, int y, int *z){
    *z = x + y;
}

int modif(int x, int *y, int *z){
    x *= 2;    *y= x+ *y; *z= 5;
    return x;
}
```

Somme de a=? et b=? : ?

Modif : a=?, b=? et c= ?

Identification des tableaux et pointeurs

15

- En C, le nom d'un tableau représente l'adresse de sa composante 0, donc `a == &a[0]`
- *C'est pour cela que les tableaux passés comme paramètres dans une fonction sont modifiables*

Passer des tableaux aux fonctions

- Pour le compilateur, un tableau comme argument de fonction, c'est un pointeur vers sa composante 0 (à la réservation mémoire près).
- La fonction peut donc modifier n'importe quel élément (passage par référence)
- Le paramètre peut soit être déclaré comme tableau, soit comme pointeur

```
int add_elements(int a[], int size)
{
```

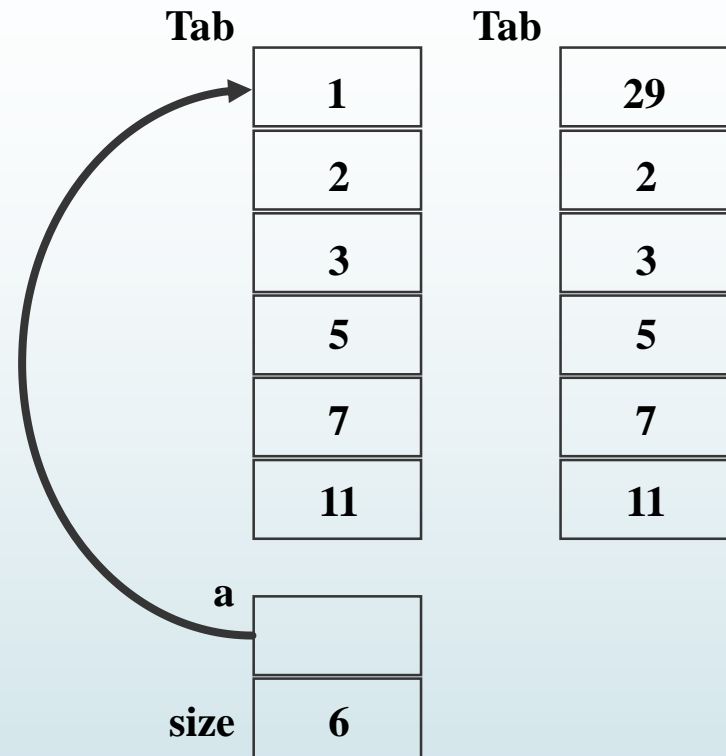
```
int add_elements(int *p, int size)
{
```

Example

16

```
#include <stdio.h>
void sum(long [], int);
int main(void)
{
    long Tab[6] = { 1, 2,
                    3, 5, 7, 11 };
    sum(Tab, 6);
    printf("%ld\n", Tab[0]);
    return 0;
}

void sum(long a[], int sz)
{
    int i;
    long total = 0;
    for(i = 0; i < sz; i++)
        total += a[i];
    a[0] = total;
}
```



Utiliser des pointeurs - exemple

17

```
#include <stdio.h>

long sum(long*, int);

int main(void)
{
    long Tab[6] = { 1, 2,
                    3, 5, 7, 11 };

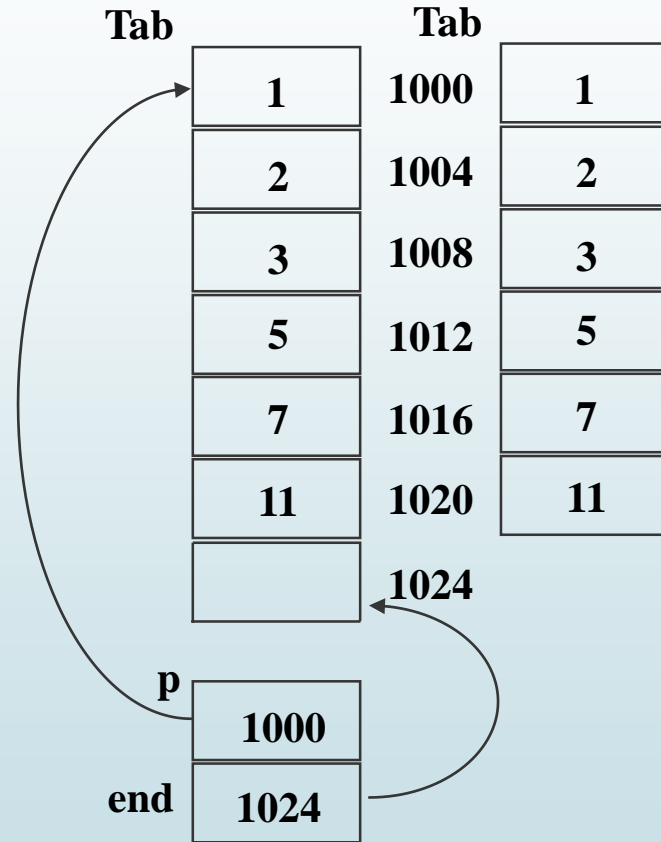
    printf("%ld\n", sum(Tab, 6));

    return 0;
}

long sum(long *p, int sz)
{
    long *end = p + sz;
    long total = 0;

    while(p < end)
        total += *p++;

    return total;
}
```



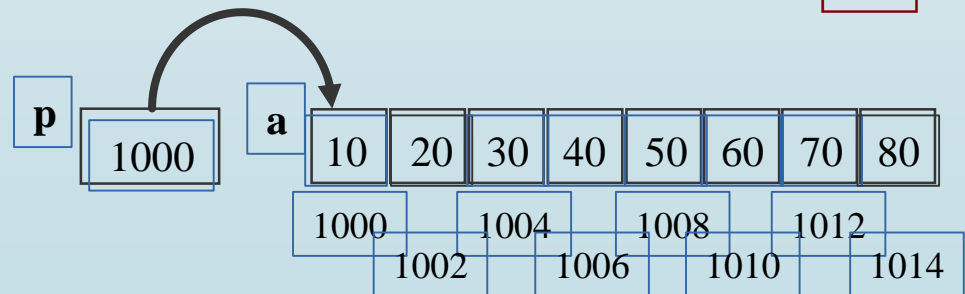
Quelle notation?

- $A[0]$ est équivalent à $*A$
- $A[i]$ est équivalent à $*(A + i)$
- $\&A[0]$ est équivalent à A

```
short  a[8] = { 10, 20, 30, 40, 50, 60, 70, 80 };  
short  *p = a;
```

```
printf("%d\n", a[3]);  
printf("%d\n", *(a + 3));  
printf("%d\n", *(p + 3));  
printf("%d\n", p[3]);
```

40
40
40
40



SCANF

19

Pour saisir un caractère ou un nombre
char c;

```
float Tab[10], X[5][7]; // *Tab,(*X)[7]
```

```
printf("TAPER UNE LETTRE: ");
```

```
scanf("%c",&c);
```

```
scanf("%f",&Tab[5]);
```

On saisit ici le contenu de l'adresse &c c'est-à-dire le caractère c lui-même.

pointeur

```
char *adr;
```

```
printf("TAPER UNE LETTRE: ");
```

```
scanf("%c",adr);
```

On saisit ici le contenu de l'adresse adr.

`float x[dim];` réserve un espace de stockage pour `dim` éléments
`x[0]` stocke l'adresse du premier élément.

`float *x;` ne réserve que le seul espace destiné au pointeur `x`, AUCUN espace n'est réservé pour une ou plusieurs variables.

La réservation d'espace est à la charge du programmeur qui doit le faire de façon *explicite*, en utilisant les fonctions standard `malloc()`, ... qui sont prototypées dans `<stdlib.h>` et `<alloc.h>`

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

```
void main()
{
```

```
    char *texte;
    float *adr1,*adr2;
```

```
    adr1 = (float*)malloc(4*sizeof(float));
    adr2 = (float*)malloc(10*sizeof(float));
    texte = (char*)malloc(10);
```

```
    *adr1 = -37.28;
```

```
    *adr2 = 123.67;
```

```
    printf("adr1 = %p adr2 = %p r1 = %f\n",adr1,adr2,*adr1,*adr2);
```

```
    free(adr1); // Libération de la mémoire
```

```
    free(adr2);
```

```
    free(texte);
```

```
}
```

sizeof

21

```
void main()
{
    int i;
    char c;
    float f;
    double d;
    printf ("caractère : %d \n ", sizeof (c));
    printf ("entier : %d \n ", sizeof (i));
    printf ("réel : %d \n ", sizeof (f));
    printf ("double : %d \n ", sizeof (d));
}
```

caractère : 1
entier : 2 ou 4
réel : 4
double : 8

EXP : Création d'un tableau de taille quelconque

22

```
#include <stdio.h>      // pour la fonction printf()
#include <stdlib.h>      // pour la fonction malloc()

int main(){
    int i , dim;         // compteur et taille du tableau
    long* tableau;       // pointeur pour stocker l'adresse du tableau

    scanf ("%d", &dim);  // saisie par l'utilisateur de la taille du tableau
    tableau = (long*) malloc(dim * sizeof(long)); //allocation (dynamique) du tableau

    // remplissage du tableau, on utilise les indices 0 à (dim -1)
    for (i = 0; i < dim; i++)
        tableau[i] = i;

    // affichage du contenu du tableau
    for (i = 0; i < dim; i++)
        printf("%ld\n", tableau[i]);

    // destruction du tableau : libération de la mémoire réservée
    free(tableau);
}
```

La mémoire dynamique varie en cours d'exécution, et peut n'être pas suffisante pour allouer les variables, provoquant le plantage du programme. Il faut donc tester le retour des fonctions d'allocation et traiter les erreurs.

Exemple:

```
#define BUFSIZE 100
long *var;

if ( !(var = (long *) malloc(BUFSIZE * sizeof(long))) )
    // Si échec de réservation de BUFSIZE emplacement de type long ( )  $\neq 0$ 
    {
        fprintf(stderr, "ERREUR espace mémoire insuffisant !\n");
        exit (1); // fin anticipée du programme ; code de retour 1
    }
else { // le programme continue
}
```

L'espace alloué doit être libéré par `free()`, dont l'argument est un pointeur réservé dynamiquement. `free(var);`

char * malloc(unsigned taille);

24

réserve taille octets, sans initialisation de l'espace. Retourne un pointeur sur une zone de taille octets.

char * calloc(unsigned nombre, unsigned taille);

réserve nombre éléments de taille octets chacun ; l'espace est initialisé à 0.

```
#define alloue(nb,type)  (type *)calloc(nb,sizeof(type))
```

```
char *s;
```

```
s = alloue(250,char);    // réserve 250 octets initialisés à '\0'
```

void * realloc(void *block, unsigned taille);

modifie la taille affectée au bloc de mémoire fourni par un précédent appel à malloc() ou calloc().

void free(void *block);

libère le bloc mémoire pointé par un précédent appel à malloc(), calloc() ou realloc().

Valable pour réserver de l'espace pour les autres types de données int, float, ... Elles retournent le pointeur NULL (0) si l'espace disponible est insuffisant.

Chapitre 2

Les Structures en C

- Concepts
- Créer un type de structure
- Créer une instance de structure
- Initialiser une instance
- Accéder aux membres d'une instance
- Passer les structures comme paramètres
- Listes chaînées

Concepts

- Une structure est une collection de plusieurs variables (champs) groupées ensemble pour un traitement commode
- Les variables d'une structure sont appelées *membres* et peuvent être de n'importe quel type, par exemple des tableaux, des pointeurs ou d'autres structures

Les étapes sont:

- ◆ déclarer le type de la structure
- ◆ utiliser ce type pour créer autant d'instances que désirées
- ◆ Accéder aux membres des instances

Déclarer les structures

- Les structures sont définies en utilisant le mot-clé struct

```
struct Date
{
    int jour;
    int mois;
    int an;
};
```

```
struct Livre
{
    char titre[80];
    char auteur[80];
    float prix;
};
```

```
struct Membre
{
    char nom[80];
    char adresse[200];
    int numero;
    float amende[10];
    struct Date emprunt;
    struct Date creation;
};
```

```
struct Pret
{
    struct Livre b;
    struct Date due;
    struct Membre *who;
};
```

Déclarer des instances

- Une fois la structure définie, les instances peuvent être déclarées
- Par abus de langage, on appellera structure une instance de structure

```
struct Date  
{  
    int jour;  
    int mois;  
    int an;  
} hier, demain;
```

```
struct Date paques;  
struct Date semaine[7];
```

```
struct Date nouvel_an = { 1, 1, 2001 };
```

← Déclaration
avant ‘;’ .

← Initialisation .

Des structures dans des structures

29

```
struct Date
{
    int  jour;
    int  mois;
    int  an;
};
```



```
struct Membre
{
    char    nom[80];
    char    adresse[200];
    int     numero;
    float   amende[10];
    struct  Date emprunt;
    struct  Date creation;
};
```

```
struct Membre    m = {
    "Arthur Dupont",
    "rue de Houdain, 9, 7000 Mons",
    42,
    { 0.0 },
    { 0, 0, 0 },
    { 5, 2, 2001 }
};
```

Accéder aux membres d'une structure

- Les membres sont accédés par le nom de l'instance, suivi de . , suivi du nom du membre

```
struct Membre m;
```

```
printf("nom = %s\n", m.nom);  
printf("numéro de membre = %d\n", m.numero);  
  
printf("amendes: ");  
for(i = 0; (i < 10) && (m.amende[i] > 0.0); i++)  
printf("%.2f Euros", m.amende[i]);  
  
printf("\nDate d'emprunt %d/%d/%d\n", m.emprunt.jour,  
      m.emprunt.mois, m.emprunt.an);
```

```
printf("nom = %s\n", m.nom);
printf("numéro de membre = %d\n", m.numero)
nom → char[20]
numero → int
struct Etudiant {
    char nom[20];
    int numero;
} et1, et2;
// remplissage de l'étudiant 1 Et1
printf("donnez le nom de l'étudiant 1\n");
scanf("%c", &et1.nom);
Printf("donnez le numero de l'étudiant 1");
Scanf("%d", &et1.numero);
// remplissage de la variable Et2
Printf("donner le nom de l'étudiant 2");
Scanf("%c", &et2.nom);
Printf("donner le numero de l'étudiant 2");
Scanf("%d", &et2.numero);
//Affichage
```

Assignation des structures

- L'opération d'affectation = peut se faire avec des structures
- Tous les membres de la structure sont copiés (aussi les tableaux et les sous-structures)

```
struct Membre  m = {  
    "Arthur Dupont",  
    ....  
};  
  
struct Membre temp;  
  
temp = m;
```


Passer des structures comme paramètres de fonction

- Une structure peut être passée, comme une autre variable, par valeur ou par adresse
- Passer par valeur n'est pas toujours efficace (recopiage à l'entrée)
- Passer par adresse ne nécessite pas de recopiage

```
void Par_valeur(struct Membre m);  
void Par_reference(struct Membre *m);  
  
Par_valeur(m);  
Par_reference(&m);
```

Quand la structure est un pointeur !

Utiliser p->name

- L'écriture p->name est synonyme de (*p).name, où p est un pointeur vers une structure

```
void  affiche_membre (struct Membre *p)
{
    printf("nom = %s\n", p->nom);
    printf("adresse = %s\n", p->adresse);
    printf("numéro de membre = %d\n", p->numero);

    printf("\nDate d'affiliation %d/%d/%d\n",
        p->creation.jour, p->creation.mois, p->creation.an);
}
```

Retour de structures dans une fonction

- Par valeur (recopiage) ou par référence

```
struct Complex add(struct Complex a, struct Complex b)
{
    struct Complex result = a;
    result.real_part += b.real_part;
    result.imag_part += b.imag_part;
    return result;
}
```

```
struct Complex  c1 = { 1.0, 1.1 };
struct Complex  c2 = { 2.0, 2.1 };
struct Complex  c3;

c3 = add(c1, c2); /* c3 = c1 + c2 */
```

Tableaux et structures

- On peut mixer les tableaux et les structures, par exemple:
- `struct point {int x;int y;};`
- `struct triangle {struct point sommets[3];};`

```
void f() {  
    struct triangle t;  
    for(i=0; i<3; i++) {  
        t.sommets[i].x = i;  
        t.sommets[i].y = i * 2;  
    }  
}
```

Initialisation mixte de tableaux et structures

On peut composer des initialisations de tableaux et de struct point {

```
    int x;
```

```
    int y;};
```

```
struct triangle {
```

```
    struct point sommets[3];};
```

```
struct triangle t = {
```

```
    { 1, 1 },
```

```
    { 2, 3 },
```

```
    { 4, 9 }
```

```
};
```



Exemple

- Écrire les déclarations d'un tableau de personnes. Chaque élément du tableau est de type structure nommé "personne" qui comporte les champs suivants :
 - âge : integer
 - taille et poids : 2 réels
- On a 100 personnes ou moins à remplir et à afficher.



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

```
#define N 10
```

```
typedef struct point {
```

```
    double abs;
```

```
    double ord; }point;
```

```
main() {
```

```
    point p[N];
```

```
    int i;
```

```
    p[0].ord = 0;
```

```
    p[0].abs = 1;
```

```
    for(i = 1 ; i < N ; i++) {
```

```
        p[i].ord = p[i - 1].ord + 1.;
```

```
        p[i].abs = p[i - 1].abs + 2.;
```

```
    }
```

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

```
        printf("p[%d] = (%f, %f)\n", i, p[i].abs, p[i].ord); }
```

```
    }
```



Structures et fonctions

- Lorsqu'on les passe en paramètre, les structures se comportent comme des variables scalaires, cela signifie qu'on ne peut les passer en paramètre que par valeur. Par contre, un tableau de structures est nécessairement passé en paramètre par référence. Réécrivons le programme précédent avec des sous-programmes :



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

```
#define N 10
```

```
typedef struct point {
```

```
    double abs; double ord; }point;
```

```
void initTableauPoints(point p[], int n) {
```

```
    int i; p[0].ord = 0;
```

```
    p[0].abs = 1;
```

```
    for(i = 1 ; i < n ; i++) {
```

```
        p[i].ord = p[i - 1].ord + 1.;
```

```
        p[i].abs = p[i - 1].abs + 2.; } }
```

```
void affichePoint(point p) {
```

```
    printf("(%f, %f)", p.abs, p.ord); }
```

```
void afficheTableauPoints(point p[], int n) {
```

```
    int i;
```

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

```
        printf("p[%d] = ", i);
```

```
        affichePoint(p[i]);
```

```
        printf("\n"); } }
```

```
main() {
```

```
    point p[N]; initTableauPoints(p, N);
```

```
    afficheTableauPoints(p, N); }
```

```

#include<stdio.h>
#define N 10
typedef struct point {
    double abs; double ord; }point;
point nextPoint(point previous) {
    point result;
    result.ord = previous.ord + 1.;
    result.abs = previous.abs + 2.;
    return result; }
void initTableauPoints(point p[], int n) {
    int i; p[0].ord = 0; p[0].abs = 1;
    for(i = 1 ; i < n ; i++)
        p[i] = nextPoint(p[i - 1]); }
void affichePoint(point p) {
    printf("(%f, %f)", p.abs, p.ord); }
void afficheTableauPoints(point p[], int n) {
    int i; for(i = 0 ; i < n ; i++) {
        printf("p[%d] = ", i);
        affichePoint(p[i]); printf("\n"); } }
main() {
    point p[N];
    initTableauPoints(p, N);
    afficheTableauPoints(p, N); }

```

- Comme une structure se comporte comme une variable scalaire, il est possible de retourner une structure dans une fonction, il est donc possible de modifier le programme ci-avant de la sorte :

Enoncé

Ecrire un programme permettant de :

Constituer un tableau de 20 Enseignants max(**NE_max**). La structure est la suivante :

```
struct {  
    char nom_pren[40];           // nom+pren  
    char nom[20];  
    char pren[20];  
    int NH[NM_max] ;           // NM_max=3 : nbre d'heures pr NM_max matières  
}
```

Le programme doit gérer en boucle le menu de choix suivant:

- 1- **Saisie et affichage**
- 2- **Construction et affichage**
- 3- **Modifier et affichage**
- 4- **Tri et affichage**
- 5- **Quitter**

Tri à bulles

44

```
while(???) {
    for j = 0 to .... {
        if tab[j] > tab[j+1] {
            <on échange tab[j] et tab[j+1]>
        }
    }
}
```

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	j
2	6	4	8	12	13	0	tab[j]

6 2
 6 4 2
 6 4 8 2
 6 4 8 12 2

tab[j] < tab[j+1]

1 3	1 2	8	6	4	2	0
--------	--------	---	---	---	---	---

6	4	8	12	13	2	0
---	---	---	----	----	---	---

Chapitre 3

Gestion des fichiers en C

Elle est assurée par la librairie standard `<stdio.h>` via un ensemble de fonctions commençant par “f”

- Avant de manipuler un fichier, il faut lui associer un descripteur (pointeur vers la structure fichier)
- Il y a 3 descripteurs automatiquement ouvert par tout programme C, `stdin`, `stdout` et `stderr`
- `stdin` (standard input) est connecté au clavier. On peut y lire
- `stdout` (standard output) et `stderr` (standard error) sont reliés au moniteur. On peut y écrire.

`FILE *f;`

`/*déclare un descripteur f de fichier*/`

Ouvrir et fermer des fichiers

46

Les fichiers sont ouverts avec la fonction fopen
ils sont fermés avec la fonction fclose

```
FILE* fopen(const char* name, const char* mode);  
int fclose (FILE *f);
```

```
#include <stdio.h>  
int main(void)  
{  
    FILE* in;  
    FILE*   out;  
    FILE*   append;  
    in = fopen("autoexec.bat", "r");  
    out = fopen("autoexec.bak", "w");  
    append = fopen("config.sys", "a");  
    /* ... */  
    fclose(in);  
    fclose(out);  
    fclose(append);  
}
```

Lecture et écriture sur fichier

Fonctions de lecture

```
int    fscanf(FILE* stream, const char* format, ...);  
int    fgetc(FILE* stream);  
char*  fgets(char* buffer, int size, FILE* stream);
```

Fonctions d'écriture

```
int    fprintf(FILE* stream, const char* format, ...);  
int    fputc(int ch, FILE* stream);  
int    fputs(const char* buffer, FILE* stream);
```

```
int feof(FILE *f); /*renvoie une valeur non nulle si fin de fichier*/
```

Exemple d'écriture (lecture) de fichier

48

```
#include <stdio.h>
void sauvegarde( char titre[], int n,
                 float x[], int ind [] )
{
    int i=0;
    FILE *f;
    f = fopen("monfichier.dat","w");
    if (f !=NULL){
        fprintf(f,"%s\n",titre);
        for (i=0; i < n; i++ ) {
            fprintf(f,"%f %d\n", x[i],ind[i]);
            Mon titre
        }
        3.0 1
        4.5 2
        7.3 3
    }
    fclose(f);
}
```

```
#include <stdio.h>
void main(void)
{
    char titre[81];
    float x[10];
    int ind[10], i=0;
    FILE *f;
    f = fopen("monfichier.dat","r");
    if (f!= NULL) {
        fgets(titre,80,f);
        while(!feof(f)) {
            fscanf(f,"%f %d",&x[i],&ind[i]);
            i++;
        }
    }
    fclose(f);
}
```

La constante NULL (définie comme 0 dans stdio.h) réfère à une adresse non définie

Compilation Séparée et édition de Lien

49

Un *programme C* est un ensemble de *fonctions* dans un ou plusieurs fichiers.

Fichier PP.c

```
main(){  
...  
appelle f()  
...  
}  
f(){  
appelle g()...  
}  
g(){  
...  
}
```

Cas I

Fichier PP.c

```
main(){  
...  
appelle f()  
...  
}  
f(){  
appelle g()...  
}
```

Cas II

Fichier SP.c

```
g(){  
...  
}
```

Structure d'un programme C

50

```
#include <stdio.h>
#define DEBUT -10
#define FIN 10
#define MSG "Programme de démonstration\n"

int fonc1(int x);
int fonc2(int x);

void main()
{
    /* début du bloc de la fonction main*/
    /* définition des variables locales */
    int i;

    i = 0 ;
    fonc1(i) ;
    fonc2(i) ;
}

/* fin du bloc de la fonction main */

int fonc1(int x) {
    return x;
}

int fonc2(int x) {
    return (x * x);
}
```

Directives du préprocesseur :
accès avant la compilation

Déclaration des fonctions

Programme
principal

Définitions des
fonctions

Structure d'un programme C

51

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

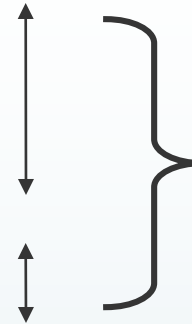
```
#define DEBUT -10
```

```
#define FIN 10
```

```
#define MSG "Programme de démonstration\n"
```

```
int fonc1(int x);
```

```
int fonc2(int x);
```



Fichier.h

```
#include "Fichier.h"
```

```
void main()
```

```
{
```

```
    int i;
```

```
    i = 0 ;
```

```
    fonc1(i) ;
```

```
    fonc2(i) ;
```

```
}
```

```
int fonc1(int x) {
```

```
    return x;
```

```
}
```

```
int fonc2(int x) {
```

```
    return (x * x);
```

```
}
```

/* début du bloc de la fonction main*/

/* définition des variables locales */

/* fin du bloc de la fonction main */



Programme
principal

Définitions des
fonctions

Règles de visibilité des variables

Les variables ne peuvent être utilisées que là où elles sont déclarées

```
double func(int x);
int glob=0;           // variable globale

int main(void)
{
    int i = 5, j, k = 2; //locales à main
    glob++;
    func(i);
    func(k);
}

double func(int v)
{
    double d, f=v;      //locales à func
    glob++;
    f += glob;
    return f;
}
```

```
#include <stdio.h>

int next(){
    static value = 0;
    return value++;
}

void main(void) {
    printf("Appel 1 : %d\n",next());
    printf("Appel 2 : %d\n",next());
}
```

Appel 1 : 0

Appel 2 : 1

Variables et fonctions externes

53

Le fichier impose lui aussi un domaine de visibilité. Une variable définie globale au fichier (en dehors de toute fonction) ne sera alors visible que par les fonctions de ce fichier. Le problème est de savoir comment exporter cette variable pour d'autres fonctions du programme (externes au module) si le besoin s'en fait ressentir ?

Fichier "Module.h"	Fichier "Module.c"
<pre>extern int a; void fct();</pre>	<pre>#include "module.h" int a = 0; void fct() { a++; }</pre>
Fichier "main.c"	
<pre>#include <stdio.h> #include "module.h" int main(void) { fct(); a++; printf("%d\n",a); }</pre>	

// Résultat affiché : 2

Directives de compilation

Les directives de compilation permettent d'inclure ou d'exclure du programme des portions de texte selon l'évaluation de la condition de compilation.

<code>#if defined (symbole)</code>	<code>/* inclusion si symbole est défini */</code>
<code>#ifdef (symbole)</code>	<code>/* idem que #if defined */</code>
<code>#ifndef (symbole)</code>	<code>/* inclusion si symbole non défini */</code>
<code>#if (condition)</code>	<code>/* inclusion si condition vérifiée */</code>
<code>#else</code>	<code>/* sinon */</code>
<code>#elif</code>	<code>/* else if */</code>
<code>#endif</code>	<code>/* fin de si */</code>
<code>#undef symbole</code>	<code>/* supprime une définition */</code>

Exemple :

55

```
#ifndef (BOOL)
#define BOOL char    /* type boolean */
#endif
```

```
#ifdef (BOOL)
BOOL FALSE = 0;    /* type boolean */
BOOL TRUE = 1;     /* définis comme des variables */
#else
#define FALSE 0     /* définis comme des macros */
#define TRUE 1
#endif
```

Utile pour les fichiers include.

```
#ifndef _STDIO_H_
#define _STDIO_H_
texte a compiler une fois
...
#endif
```

Pour stocker des données en mémoire, nous avons utilisé des variables simples (type int, double...), des tableaux et des structures personnalisées. Si vous souhaitez stocker une série de données, le plus simple est en général d'utiliser des tableaux.

Toutefois, les tableaux se révèlent parfois assez limités. Par exemple, si vous créez un tableau de 10 cases et que vous vous rendez compte plus tard dans votre programme que vous avez besoin de plus d'espace, il sera impossible d'agrandir ce tableau. De même, il n'est pas possible d'insérer une case au milieu du tableau.

Les listes chaînées représentent une façon d'organiser les données en mémoire de manière beaucoup plus flexible. Comme à la base le langage C ne propose pas ce système de stockage, nous allons devoir le créer nous-mêmes de toutes pièces.

Chapitre 4

Listes chaînées

Une liste est une structure de données constituée de cellules chaînées les unes aux autres par pointeurs.

- les listes simplement chaînée.


une cellule est un enregistrement qui peut être déclarée comme suit:

```
struct Node {  
    int      data;      /* les informations */  
    struct Node *next;   /* le lien */  
};
```

- Une liste doublement chaînée.

```
struct Node {  
    int      data;      /* les informations */  
    struct Node *next;   /* lien vers le suivant */  
    struct Node *prev;   /* lien vers le précédent */  
};
```

- les listes circulaires.

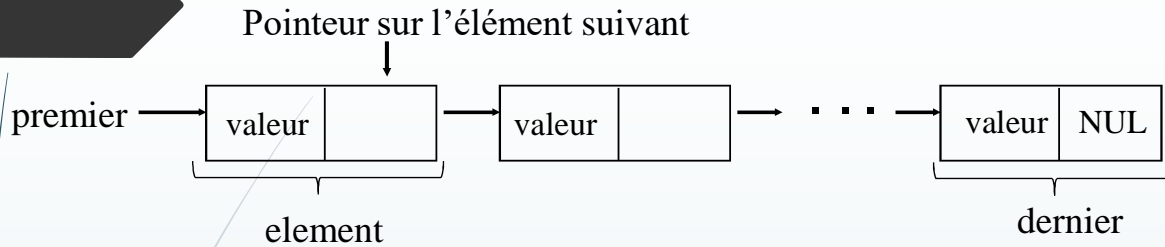


Dans une liste chaînée, un élément est la donnée d'un **couple valeur place (V , P)**.

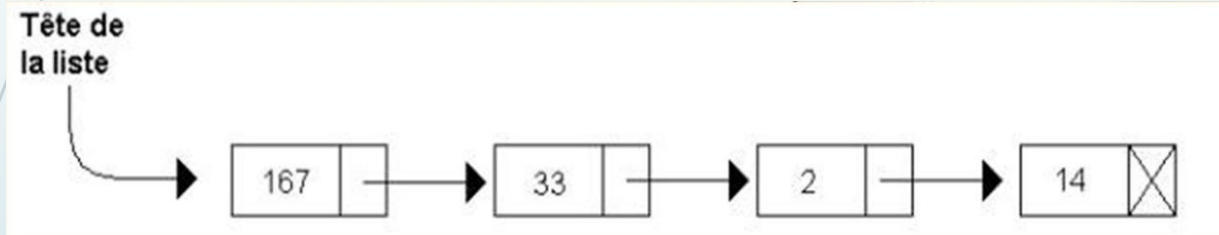
Sur une liste chaînée on peut définir trois fonctions :

- Une fonction qui permet de retourner le **1^{er} élément** de la liste
- Une fonction qui permet de retourner le **successeur** d'un élément dans la liste
- Une fonction qui permet de retourner la **valeur de l'élément** si bien sur la place de ce dernier est connue

Représentation graphique d'une liste chaînée



Exemple:



Déclaration d'une liste chaînée

```
struct elem {  
    int valeur ;  
    struct elem * suivant ; } ;
```

```
typedef struct elem  *liste;
```

On a définit un nouveau type « liste »

Explication

struct elem *suivant ; déclare un pointeur sur la données suivante.

« suivant » est le nom du pointeur et c'est là qu'on découvre à quoi sert l'étiquette. En effet à ce stade le nom de la structure n'est pas encore donné. Toutefois le pointeur va pointer sur un nouvel élément, donc sur une instance de la structure que l'on est justement en train de définir. On utilise donc l'étiquette précédée du mot clef « struct » qui est « elem ».

elem : il s'agit tout simplement du nom final donnée à la structure.

typedef struct elem *liste; : permet de déclarer un nouveau type qui est un pointeur vers un élément et que l'on appelle « liste ». Cette opération est facultative mais elle accroît sensiblement la lisibilité du programme.

Accès aux valeurs de la liste ...

L'accès à la valeur d'une variable d'un élément donné se fait en reprenant les notations de structure composée.

Exemple :

```
liste ls1 ; int a;  
a=(*ls1).valeur ;
```

Équivalent à:

```
liste ls1 ; int a;  
a=ls1->valeur ;
```

L'accès au(x) pointeur(s) se fera de la même façon :
(*ls1).suivant **Ou** ls1->suivant.

Structures de données dynamiques

62

```
typedef struct Node {  
    int    data;  
    struct Node  *next;  
}cellule;  
  
cellule * new_node(int value)  
{  
    cellule * p;  
  
    p = (cellule *)malloc(sizeof(cellule));  
    p->data = value;  
    p->next = NULL;  
  
    return p;  
}
```

Initialiser la liste chaînée

- Pour cela on utilise l'instruction « malloc » associé à « sizeof » :
 - `liste ls1;`
 - `ls1=(liste)malloc(sizeof(struct elem))`
-
- Notre liste ls1 est maintenant initialisée.
 - « ls1 » sera un **pointeur** sur le **premier élément**. On peut désormais affecter une valeur à cet élément. Il est important de se souvenir qu'il faudra réserver de la mémoire, donc faire un **malloc**, à chaque fois que l'on voudra **insérer un nouvel élément**, à l'emplacement de l'élément.

Saisie de plusieurs éléments

Admettons que l'utilisateur a entrée au clavier la valeur d'une variable **entière** «**n**» indiquant le **nombre d'élément à créer**. Le programme de saisie pourrait être celui-ci :

```
liste ls1,temp; //on déclare également une variable auxiliaire qui va nous servir pour
parcourir la liste
ls1=(liste)malloc(sizeof(struct elem )) ;//on initialise c'est à dire on fait pointer ls1
sur l'espace // mémoire réservé pour le premier élément
temp=ls1 ; //on fait pointer laux au même endroit que ls1
for(i=1 ;i<n+1 ;i++)
{ //on doit saisir n valeurs
scanf("%d",&temp->valeur) ; //saisie de la valeur on fait pointer le pointeur suivant sur
un nouvel //espace réservé
if(i==n) temp->suivant=NULL ; //on fait pointer le dernier élément sur un pointeur
NULL afin
// d'avoir un programme plus « propre »
else {temp->suivant=(liste)malloc(sizeof( struct elem )) ;
temp=temp->suivant ; //on fait pointer laux sur le nouvel espace créé afin de saisir la
valeur
```

```
//du prochain élément au prochain tour de boucle :
```


➤ Attention

- Il ne faut surtout pas utiliser le pointeur principal « ls1 » pour parcourir les éléments de la liste. Sinon on perdrait l'adresse du premier élément et donc de toute la liste chaînée.
- on utilise toujours un pointeur de parcours (liste auxiliaire)
- Il est important de savoir qu'il faut toujours faire l'allocation de mémoire **AVANT** d'ordonner au pointeur de pointer sur un élément.
 - Sion avait écrit : `laux=laux->suivant;`
 - `laux=(liste)malloc(sizeof(struct elem));` de même que:
 - `laux=ls1 ;`
 - `ls1=(liste)malloc(sizeof(struct elem));`
- la compilation aurait fonctionné mais l'exécution non. En effet l'adressage du pointeur change au moment de l'allocation de la mémoire. **Ceci représente une erreur type. Il faut y faire attention !**
- N'oubliez pas non plus que **toute saisie d'un nouvel élément doit être précédée d'une allocation de mémoire.**

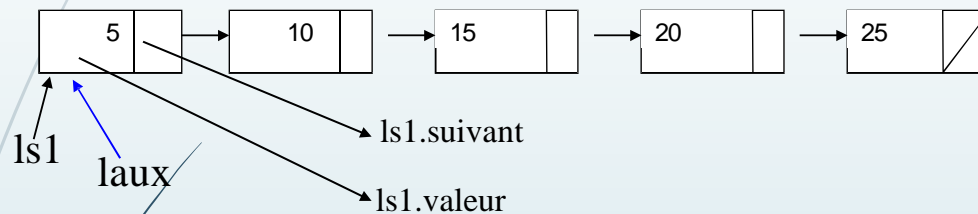
➤ Remarque

- En pratique, la saisie se fait rarement par une boucle mais par un appel successif d'une **fonction** définie par l'utilisateur qui saisie un élément en début ou en fin de liste suivant les cas.
- Il n'est pas obligatoire mais il est fortement recommandé de terminer la liste par le pointeur **NULL** (sauf en cas de liste cyclique, où le dernier pointeur pointe sur le premier élément).
- Anoter l'existence de la fonction **free()** ; qui libère la place assignée par un malloc.
- **Exemple** : `free(ls1);`

Parcours d'une liste chaînée

On suppose qu'il existe une liste chaînée d'entiers, préalablement déclarée

But : Afficher dans l'ordre de la liste tous les éléments de la liste : 5-10-15- 20-25



- **Remarque** : Penser à dupliquer la valeur du premier pointeur (laux).

Parcours d'une liste chaînée

Algorithme

Var *ls1, *laux : liste

Début

laux := ls1

Tantque (laux <> NULL) FAIRE

afficher(laux->valeur) laux:= laux->suivant

FIN TANTQUE

Programme en C

```
struct elem {int valeur ;  
struct elem * suivant ; } ; typedef struct elem  
*liste;  
void display (liste p_liste)  
{for (liste temp = p_liste ;temp!= NULL; temp=  
temp ->suivant)  
printf("%d",temp->valeur)  
}  
int main (void)  
{Liste ls1  
  
//remplir la liste display(ls1);  
return 0; }
```

Remarque :

- Quand on cherche le dernier élément d'une liste, le test à effectué est (temp->suivant) =NULL.
- Le test temp <> NULL permet de parcourir indifféremment tous les éléments de la liste.

Les opérations sur les listes

- Les opérations sur les listes sont très nombreuses, parmi elles :
 - ☒ Créer une liste vide
 - ☐ Tester si une liste est vide
 - ☐ Ajouter un élément à la liste
 - ajouter en début de liste (tête de liste)
 - ajouter à la fin de la liste (queue)
 - ajouter un élément à une position donnée
 - ajouter un élément avant ou après une position donnée
 - ☐ Supprimer un élément d'une liste
 - supprimer en début de liste
 - supprimer en fin de liste
 - supprimer un élément à une position donnée
 - supprimer un élément avant ou après une position donnée
 - ☐ Afficher ou imprimer les éléments d'une liste
 - ☐ Trier une liste par ordre croissant ou décroissant
 - ☐ Ajouter un élément dans une liste triée (par ordre croissant ou décroissant)

Ajout en tête de liste quand la liste n'est vide

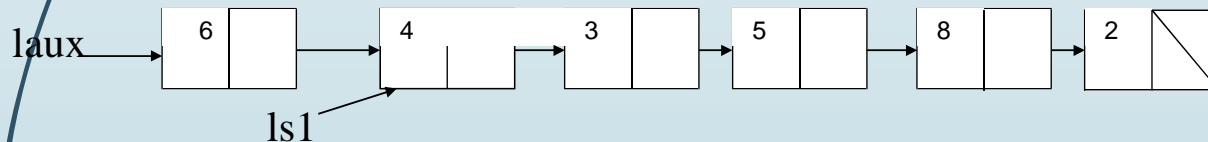
- **Exemple** : On suppose que l'on veut rajouter l'élément 6 au début de la liste suivante :



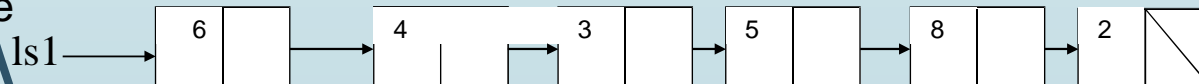
- a)- on commence par créer le nouveau **nœud** et lui donner son contenu



- b)- le **suivant** du nouveau nœud est l'ancienne tête de liste



- c)- on fait pointer le pointeur **ls1** vers la tête de liste



Algorithme Allouer

(nouveau)

lire (nouveau -> Valeur) nouveau -
->suivant \leftarrow ls1 ls1 \leftarrow nouveau

Programme C

```
struct elem{ int valeur;  
             struct elem *suivant;};  
typedef struct elem *liste ;  
liste add_head (liste ls, int value)  
{  
    liste temp=(liste)malloc(sizeof(l struct  
elem)) ; //allocation du nœud  
    if (temp!= NULL) // si tout s'est bien passé  
    { // mise a jour des champs données  
        temp ->valeur = value;  
        //on attache le nouveau élément à la  
        //liste  
        temp ->suivant = ls;  
        //On mis à jour la liste  
        ls=temp}  
    //On retourne la liste, mise à jour  
    return ls; }
```

Ajout en fin de liste (liste non vide)

- a) on cherche **le dernier élément** de la liste, pour cela on utilise un **pointeur de parcours**.
- b) création du nouveau **nœud** à insérer dans la liste.(nouveau)
- c) lier le nouveau **nœud** à la liste.

Algorithme

```
q ← ls1
TANTQUE (q -> Suivant <> nil) FAIRE
    q := q -> Suivant
FIN TANTQUE
Allouer (nouveau)
lire (nouveau -> Valeur)
nouveau -> suivant ← nil  q -> suivant
    ← nouveau
```

Programme C

```
liste add_end (liste p_head, int value)
{liste temp, p_new =(liste)malloc(sizeof(struct
elem));
                                //allocation du nœud
if (p_new != NULL)    // si tout s'est bien passé
{ p_new->valeur = value;
  // mise a jour des champs données
  p_new->suivant = NULL; // chaînage par défaut
  temp = p_head;
  // on cherche le dernier nœud
  while (temp ->suivant != NULL)
  {temp = temp ->suivant; }// pointer sur le suivant
  temp ->suivant = p_new; //modification du chaînage
}
return p_head; }
```

Ajout en fin de liste (liste non vide)

...

Version 2

Algorithme

$q \leftarrow ls1$

TANTQUE ($q \rightarrow \text{Suivant} \neq \text{nil}$)

FAIRE

$q := q \rightarrow \text{Suivant}$

FINTANTQUE

Allouer (nouveau)

lire (nouveau \rightarrow Valeur)

$\text{nouveau} \rightarrow \text{suivant} \leftarrow \text{nil}$

$q \rightarrow \text{suivant} \leftarrow \text{nouveau}$

Programme C

```
Void add_end (liste *p_head, int value)
```

```
{liste temp, p_new
```

```
=(liste)malloc(sizeof(struct elem)) ;
```

```
//allocation du nœud
```

```
if (p_new != NULL) // si tout s'est bien passé
```

```
{ p_new->valeur = value;
```

```
// mise a jour des champs données
```

```
p_new->suivant = NULL; // chaînage par défaut
```

```
temp = * p_head; // on cherche le dernier nœud
```

```
while (temp->suivant!= NULL)
```

```
{temp = temp->suivant; } // pointer sur le suivant
```

```
temp->suivant = p_new; //modification du chaînage
```

```
}
```

```
}
```


Remarque

Dans cette situation, si la liste n'est pas vide le pointeur nouveau n'est pas indispensable car le pointeur de parcours q pointe sur le dernier bloc.

Comment?

Programme C

```
liste add_end (liste p_head, int value)
{liste temp = p_head;
  // on cherche le dernier nœud while
  (temp ->suivant!= NULL)
  {temp = temp ->suivant; }// pointer sur le suivant
  temp ->suivant =(liste)malloc(sizeof(struct elem)) ;
  //allocation du nœud
  if (temp ->suivant != NULL) // si tout s'est bien passé
  {temp = temp ->suivant;
   temp ->suivant = NULL; // chaînage par défaut
   temp ->valeur = value; // mise a jour des champs
   données
  }
  return p_head; }
```

➤ Algorithme

- $q \leftarrow ls1$
- **TANTQUE** ($q \rightarrow \text{Suivant} \neq \text{nil}$)
 - **FAIRE**
- $q := q \rightarrow \text{Suivant}$
- **FINTANTQUE**
- Allouer ($q \rightarrow \text{Suivant}$)
- $q := q \rightarrow \text{Suivant}$
- $q \rightarrow \text{Suivant} := \text{nil}$
- entrer ($q \rightarrow \text{valeur}$)

Ajout en fin de liste (traitement le cas ou la liste est vide) ...

Algorithme

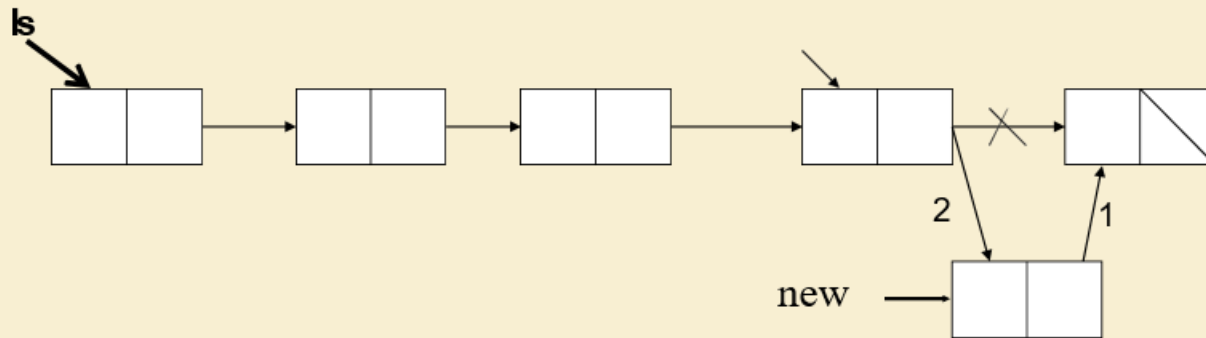
- $q \leftarrow ls1$
- **TANTQUE** ($q \rightarrow \text{Suivant} \neq \text{nil}$)
- **FAIRE**
- $q := q \rightarrow \text{Suivant}$
- **FINTANTQUE**
- Allouer ($q \rightarrow \text{Suivant}$) $q := q \rightarrow \text{Suivant}$
- $q \rightarrow \text{Suivant} := \text{nil}$ entrer ($q \rightarrow \text{valeur}$)

Programme C

```
liste add_end (liste p_head, int value)
{
    liste temp , p_new =(liste)malloc(sizeof(struct
    elem)) ; //allocation du nœud
    if (p_new != NULL) // si tout s'est bien passé
    {
        p_new->valeur = value;
        // mise a jour des champs données
        p_new->suivant = NULL; // chaînage par défaut
        temp = p_head;
        // on cherche le dernier nœud
        if (p_head!=NULL) //si la liste est non vide
        {
            while (temp ->suivant!= NULL)
            {
                temp = temp ->suivant; // pointer sur le suivant
            }
            temp ->suivant = p_new; // modification chaînage
        }
        else // la liste est vide
        {
            p_head = p_new;
        }
    }
    return p_head;
}
```

Ajout dans une liste chaînée ...

➤ *Ajout d'un élément après un nœud pointé par p*



La recherche a été effectuée et est fournie le pointeur sur le bloc qui précède celui à insérer.

Algorithme

a)- création du nouveau bloc

b)- mise à jour des liens.

Allouer (nouveau)

Entrer (nouveau -> Valeur)

nouveau -> Suivant := p->

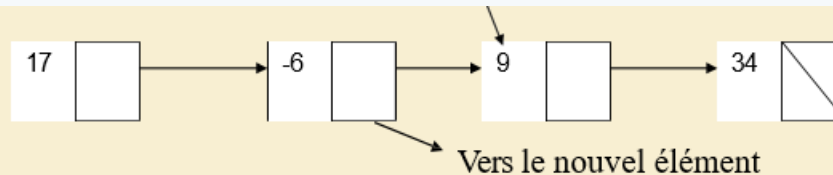
suivant p->suivant := nouveau

Programme C

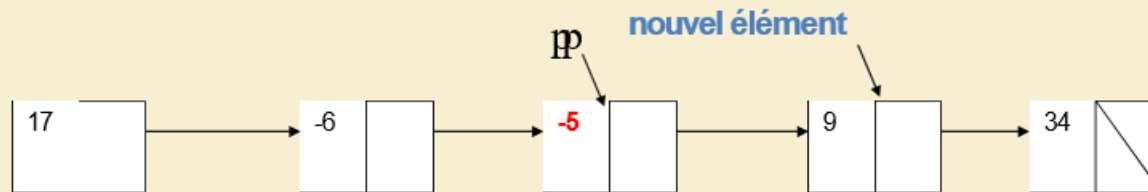
```
void add_after(liste p,int value)
{liste new =(liste)malloc(sizeof(struct
elem)); //allocation du nœud
if (new != NULL) // si tout s'est bien passé
{ new->valeur = value;
// mise a jour des champs données
new->suivant = p->suivant;
p->suivant = new; }
```

Ajout dans une liste chaînée

- *Ajout d'un élément avant un nœud pointé par p*



Sila valeur du nouvel élément est -5, le successeur du prédécesseur de p (-6) devient le nouvel élément (-5), or nous ne pouvons pas remonter un pointeur à reculons, il faudrait repartir au début de la liste. Une solution serait d'insérer le nœud après p, transférer la valeur de nœud p dans le nouvel élément et modifier la valeur de p avec la nouvelle valeur



Noter qu'avec cette solution le pointeur p qui pointait vers le nœud contenant 9 avant l'insertion du nouvel élément, pointe vers le nœud contenant -5 après l'insertion du nouvel élément.

Algorithme

- a)- création du nouveau bloc
- b)- mise à jour des liens.

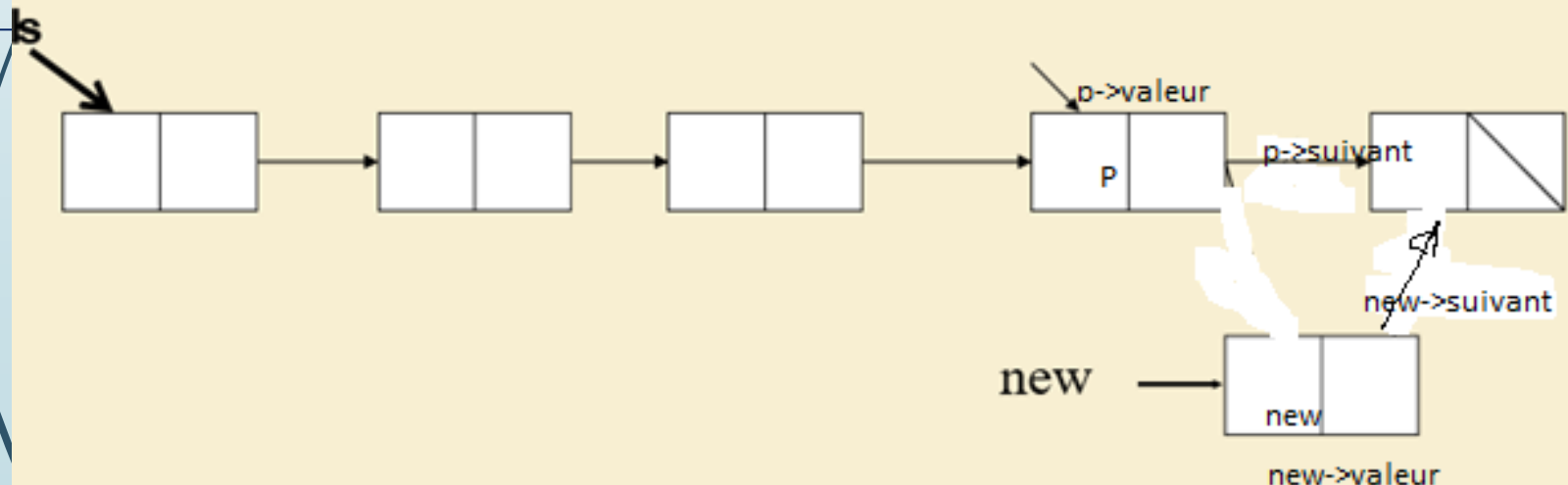
Allouer (q)

q -> Valeur := p -> Valeur
q -> Suivant := p -> suivant
p -> suivant := q

Entrer (p -> Valeur)

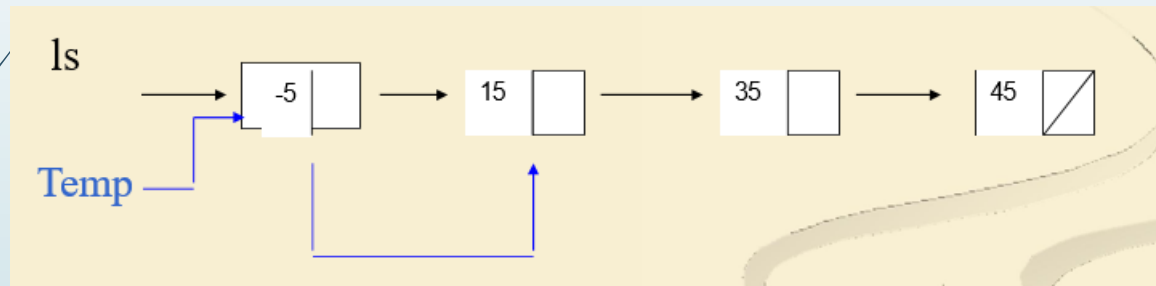
Programme C

```
void add_before(liste p,int value)
{liste
new=(liste)malloc(sizeof(struct
elem)); //allocation du nœud
if (new != NULL) // si tout s'est bien passé
{
new->valeur = p -> valeur;
new->suivant = p -> suivant;
p->suivant = new;
p ->valeur= value;
}
```



Suppression dans une liste chaînée

- On ne peut pas détruire un élément dans une liste vide.
- Suppression en tête de liste



- **ls := ls-> Suivant**
- Si on se content de cette instruction, le programme a logiquement éliminé de la liste le premier bloc, mais celui-ci est toujours en mémoire. De ce fait, il **occupe inutilement de la place mémoire**. Il faut donc **libérer cette place mémoire**.

Suppression dans une liste chaînée

- *Suppression en tête de liste*
- La fonction `free`, en langage C, permet de libérer cette place mémoire. Cette fonction prend un paramètre en l'occurrence, le pointeur de bloc qui doit être supprimé.

Algorithme

```
Temp := ls
ls := ls->Suivant Libérer
(Temp)
(Temp ne pointe sur rien du tout)
```

Programme C (Version 2)

```
liste delete_first(liste ls)
{liste Temp=ls; if(ls!=NULL)
  { ls=ls->suivant;
    free(Temp);}
return ls;}
```

Program C: V1

```
Void delete_first(liste *ls){
liste Temp = (*ls);
If(ls != NULL)
*ls = (*ls)->suivant;
Free(Temp);}
```


Suppression dans une liste chaînée

► Suppression en fin de liste

- Parcours de la liste à la recherche du dernier élément (sous réserve d'avoir en mémoire l'adresse du dernier élément).
- L'avant dernier bloc de la liste devient le dernier.
- On libère le bloc à supprimer.

```
dernier := ls
Avdernier:=ls
TQ (dernier-> Suivant) <> nil faire
    avdernier := dernier
    dernier := dernier-> Suivant
FTQ
avdernier -> Suivant := nil Libérer
(dernier)
```

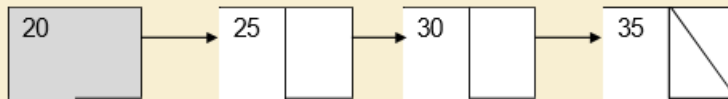
```
liste delete_last(liste ls)
{ liste dernier=ls, avdernier=ls; while(dernier-
>suivant!=NULL)
    {avdernier=dernier; dernier=dernier-
>suivant;}
free(dernier);
avdernier->suivant=NULL; return ls}
```

```
void delete_last (liste *ls)
{liste dernier=*ls,*avdernier=*ls;
while(dernier->suivant!=NULL)
    {avdernier=dernier;
    dernier=dernier->suivant;}
free(dernier);
avdernier->suivant=NULL; }
```

Suppression dans une liste chaînée

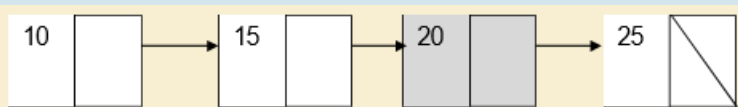
- *Suppression d'un élément de la liste correspondant à un critère*
- Le critère ici est que le champ valeur de l'élément soit égal à une valeur donnée
- Prenons par exemple 20 comme valeur à rechercher. Si un élément a 20 pour valeur entière il est supprimé et la recherche est terminée. Le principe est le suivant :

1) si c'est le premier élément :

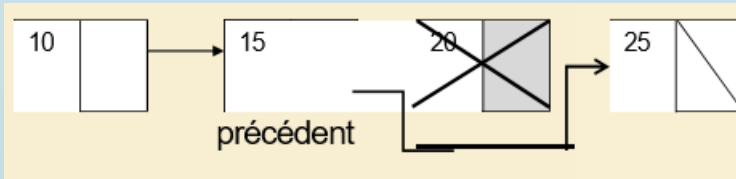


supprimer le premier
comme indiqué plus haut

2) sinon trouver dans la liste un élément où il y a 20 :



3) le supprimer, ce qui suppose d'avoir l'adresse du précédent afin de rétablir la chaîne avec le suivant l'élément 20



Algorithme

si ls != NULL

 si ls->valeur==val la:=ls

 ls:=ls->suivant free(la)

sinon

 prec:=ls

 la:= ls->suivant

 tantque (la !=NULL) faire si (la->val ==
 val)

 prec->suivant:=la->suivant free(la)

 provoquer la fin de laboucle avec un break;

 finsi prec=la la=la->suiv

fin tant que

finsi

finsi

Programme C

```
liste critere_supp_un1(liste ls,int val)
```

```
{liste la, prec; if(ls!=NULL)
```

```
  { if(ls->valeur==val) // si premier
```

```
    { la=ls;
```

```
      ls=ls->suivant; free(la);}
```

```
  else// sinon voir les autres
```

```
  { prec=ls;
```

```
    la=ls->suivant; while(la != NULL)
```

```
      { if (la->valeur==val)
```

```
        { prec->suivant=la->suivant; free(la);
```

```
          break;
```

```
        }
```

```
      prec=la;
```

```
      la=la->suivant;
```

```
    }
```

```
  }
```

```
}
```

```
return ls;}
```



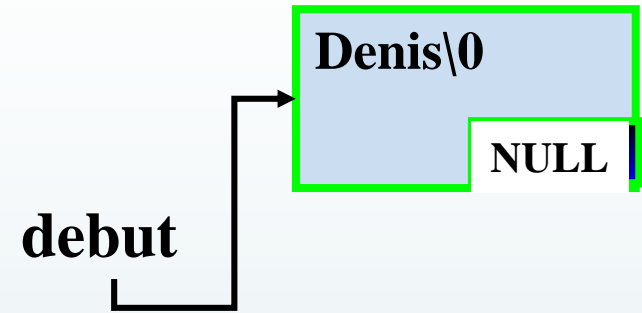


Quelques exemples

Nouvelle cellule dans une liste chaînée vide

86

```
cellule *debut;  
debut = (cellule *) malloc (sizeof(cellule));  
strcpy ( debut->name, "Denis");  
debut->next = NULL;
```

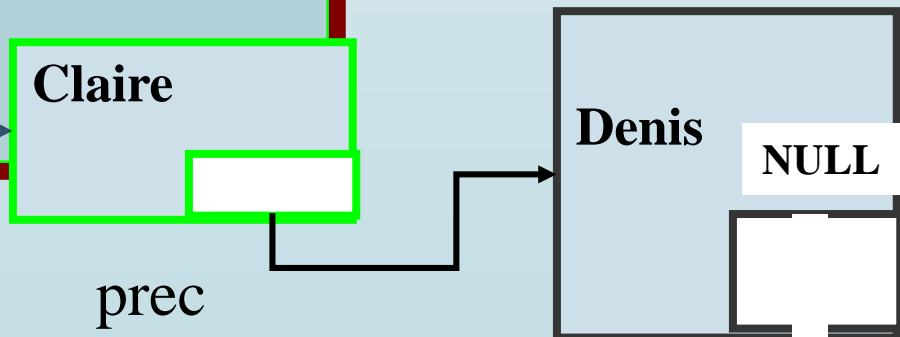


Le début de la liste est indiqué par un pointeur indépendant (debut) et la fin par NULL

Nouvelle cellule en début de liste chaînée

```
cellule *prec;  
prec = (cellule *) malloc (sizeof(cellule));  
strcpy ( prec->name, "Claire");  
prec->next = debut;  
debut = prec;
```

debut

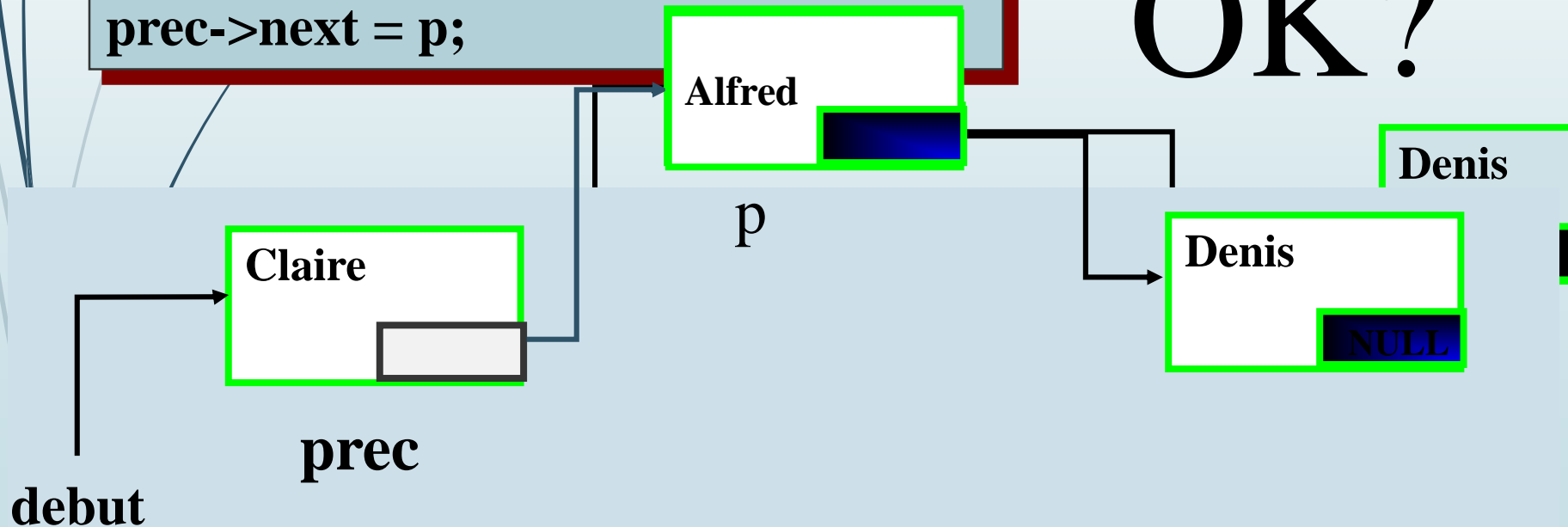


Nouvelle cellule après la cellule prec

```
cellule *p;  
p = (cellule *) malloc (sizeof(cellule));  
strcpy ( p->name, "Alfred");  
p->next = prec->next;  
prec->next = p;
```



OK?



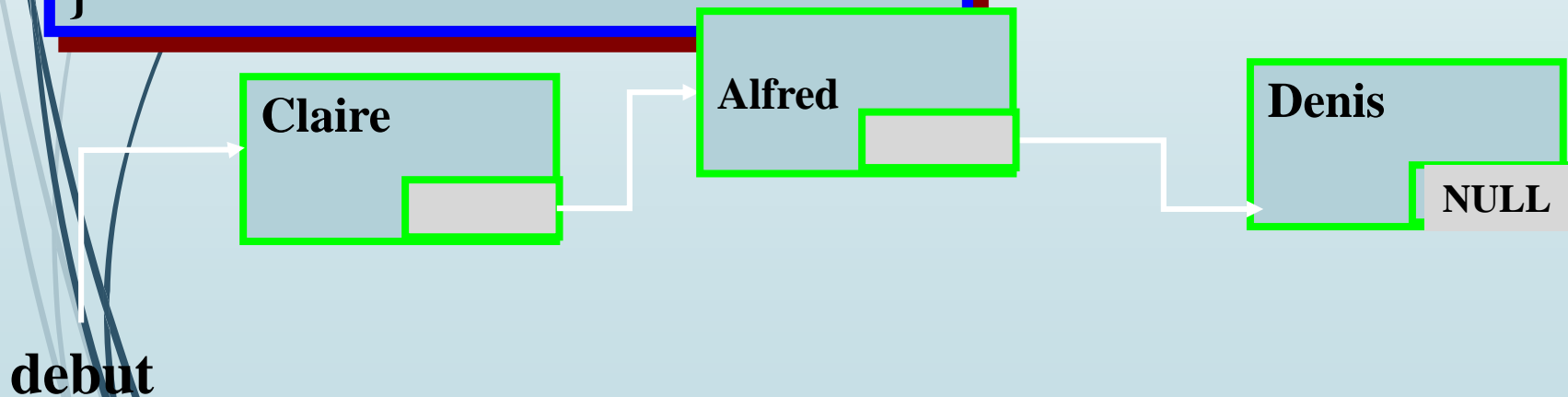
Parcourir une liste

88

```
void parcours (struct Node *debut)
{
    struct Node *p;
    p = debut;
    while ( p != NULL) {
        printf ("%s\n", p->name);
        p = p->next;
    }
}
```

debut est un pointeur sur la cellule qui contient le premier élément de la liste

Liste identifier par l'adresse de sa première cellule




```
void liberation(liste L){  
    if (L) {  
        liste temp = L->suivant;  
        free(L);  
        liberation(temp);  
    }  
}
```

```
void liberation (liste *lis)  
{  
    liste *p;  
    while ( lis != NULL) {  
        p = lis;  
        lis=lis->next;  
        free(p);  
    }  
}
```

Example

```
int main(){
struct node {
    int data;
    struct node *next;
};
struct node *head = NULL;
struct node *second = NULL;
struct node *third = NULL;

head = (struct node*)malloc(sizeof(struct node));
second = (struct node*)malloc(sizeof(struct node));
third = (struct node*)malloc(sizeof(struct node));

head->data = 1;
head->next = second;

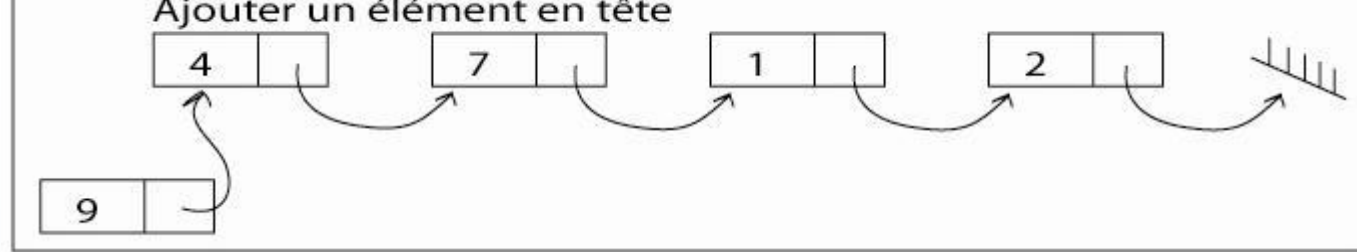
second->data = 2;
second->next = third;

third->data = 3;
third->next = NULL;

void printList(struct node *n){
    while(n != NULL){
        printf("%d", n->data);
        n = n->next;
    }
    printList(head);
return 0;
}
```

Déclaration en C d'une liste chaînée

```
include <stdlib.h>
typedef struct element element;
struct element {
    int val;
    struct element *nxt;
};
typedef element* llist;
int main(int argc, char **argv) {
    /* Déclarons 3 listes chaînées de façons différentes mais équivalentes */
    llist ma_liste1 = NULL;
    element *ma_liste2 = NULL;
    struct element *ma_liste3 = NULL;
    return 0;
}
```



```
lister AjouterEnTete(lister liste, int valeur) {
```

```
/* On crée un nouvel élément */
```

```
element* nouvelElement =(element*) malloc(sizeof(element));
```

```
/* On assigne la valeur au nouvel élément */
```

```
nouvelElement->val = valeur;
```

```
/* On assigne l'adresse de l'élément suivant au nouvel élément */
```

```
nouvelElement->nxt = liste;
```

```
/* On retourne la nouvelle liste, i.e. le pointeur sur le premier  
élément */
```

```
return nouvelElement;
```

```
}
```

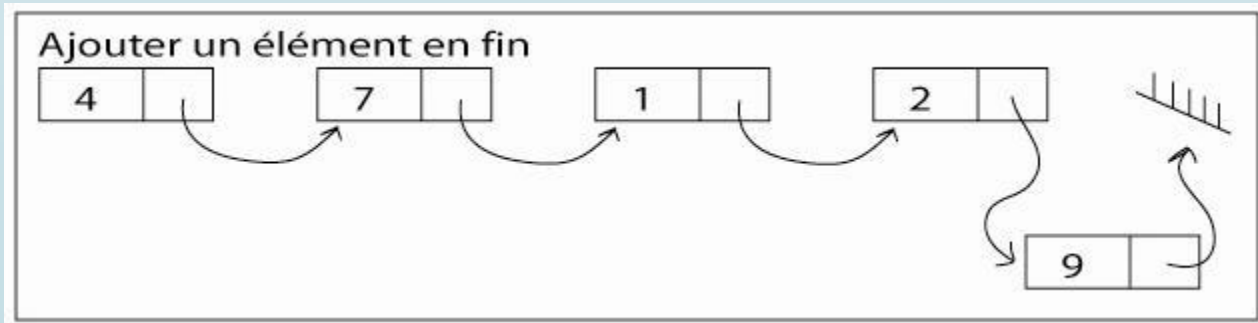
Ajout en tête

```

lister ajouterEnFin(lister liste, int valeur)
{
    /* On crée un nouvel élément */
    element* nouvelElement = malloc(sizeof(element));
    /* On assigne la valeur au nouvel élément */
    nouvelElement->val = valeur;
    /* On ajoute en fin, donc aucun élément ne va suivre */
    nouvelElement->nxt = NULL;
    if(liste == NULL) {
        /* Si la liste est vide il suffit de renvoyer l'élément créé */
        return nouvelElement; }
    else {
        /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on
        indique que le dernier élément de la liste est relié au nouvel élément */
        element* temp=liste;
        while(temp->nxt != NULL) {
            temp = temp->nxt; }
        temp->nxt = nouvelElement;
        return liste;
    }
}

```

Ajouter en fin de liste



LIFO

Exemple

Last-In-First-Out

- *Le dernier élément ajouté dans la liste, est le premier à en sortir.*
- *Opérations:
Créer la pile,
Ajouter un élément (Push), Effacer
un élément (Pop), Eliminer la pile .*

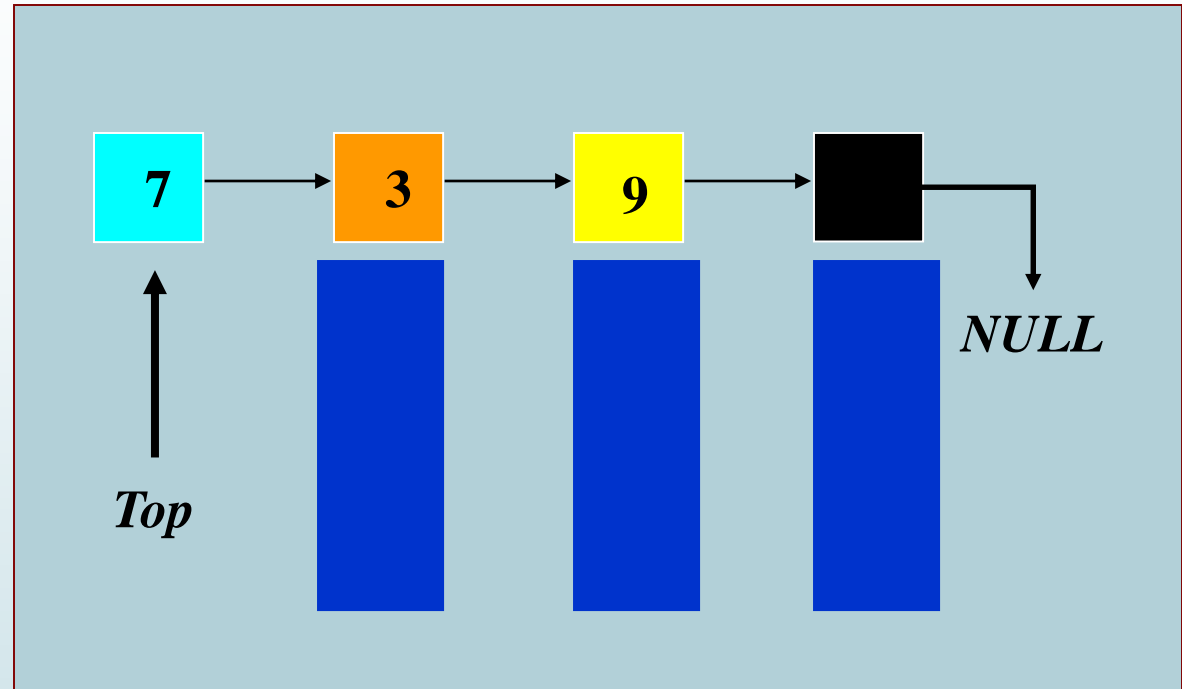
PUSH procédure

Push(5)

Push(9)

Push(3)

Push(7)



Overflow risk si la pile est pleine!!!

POP procédure

Pop(7)

Pop(3)

Pop(9)

Pop(5)

*Underflow risk
si la pile est vide !!!*

Simulation de la factorielle

97

$$5! = ??$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

0

1

2

3

4

5

Simulation de la factorielle

$$5! = ??$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

$$0! = 1$$

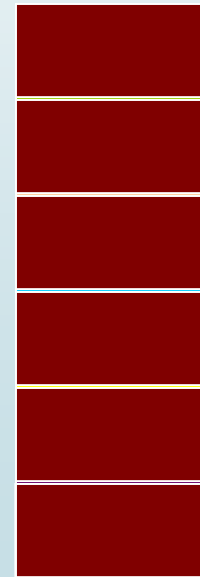
$$1! = 1 * 1 = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 = 6$$

$$4! = 4 * 6 = 24$$

$$5! = 5 * 24 = 120$$



TP3

99

Créer une liste simplement chaînée dont chaque cellule contient les champs suivants :

```
{  
    int      data;    /* les informations */  
    struct Node *next; /* le lien : pointeur sur la cellule suivante */  
}
```

Le programme doit gérer en boucle le menu de choix suivant :

- 1- Créer une liste avec insertion en tête de liste (avec affichage)
- 2- Sauver la liste
- 3- Insérer dans la liste (avec affichage)
- 4- Nettoyer la mémoire et sortir

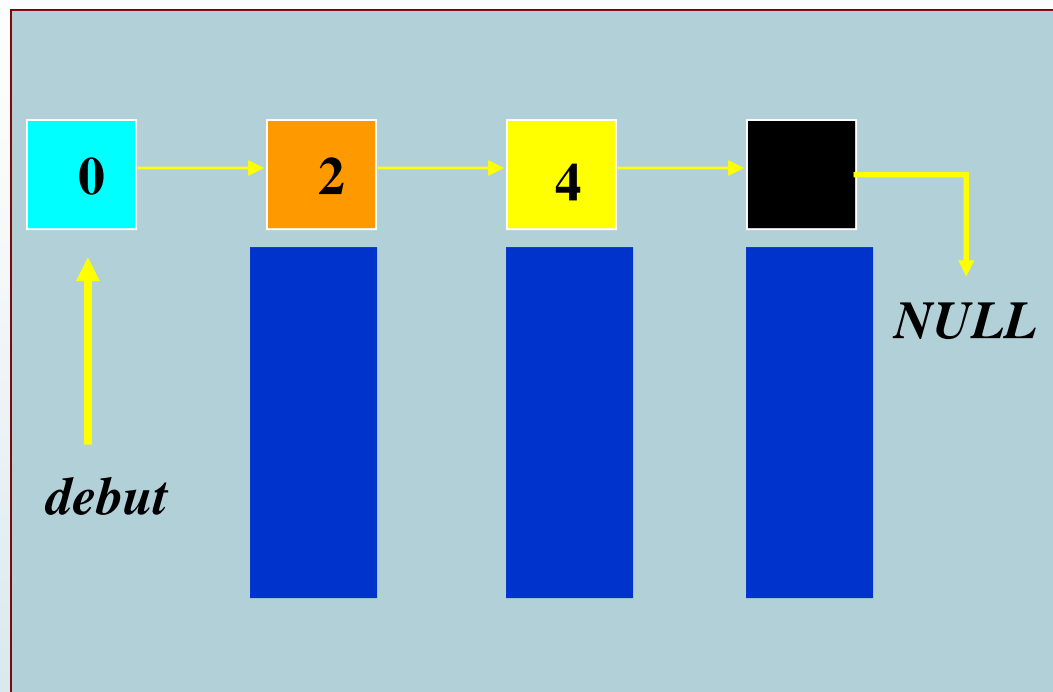
1 : Crée une liste en insérant $2n$, puis $2n-2$, puis $2n-4$, ... et enfin 0 qui sera en tête de la liste. (Liste : 0,2,4,6,8,10, ... $2n$.)

Push(6)

Push(4)

Push(2)

Push(0)



2 : Sauvegarde la liste dans le fichier Liste1

3 : Insère dans la liste -1 , puis 1 , puis 3 , ... et en fin $2n-1$.

(Liste finale : $-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \dots, 2n$)

2' : Sauvegarde la nouvelle liste dans le fichier Liste2

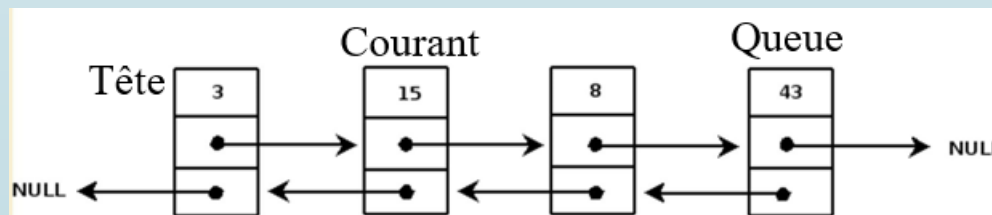
4 : Libère la mémoire avant de quitter le programme.

Listes doublement chaînées

■ Définition

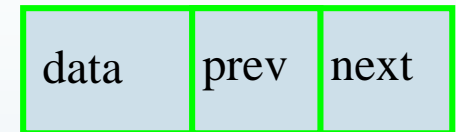
- Dans les listes simplement chaînées, à partir d'un nœud donné, on ne peut accéder qu'au successeur de ce nœud. Dans une liste doublement chaînée (ou bilatère), à partir d'un nœud donné, on peut accéder au nœud successeur et au nœud prédécesseur.

- Les éléments d'une liste bilatère contiennent 2 pointeurs :
 - *un pointeur sur le bloc suivant
 - *un pointeur sur le bloc précédent



Listes doublement chaînées

```
struct Node {  
    int      data;    /* les informations */  
    struct Node *prev; /* lien vers le précédent  
*/    struct Node *next; /* lien vers le suivant */  
};
```



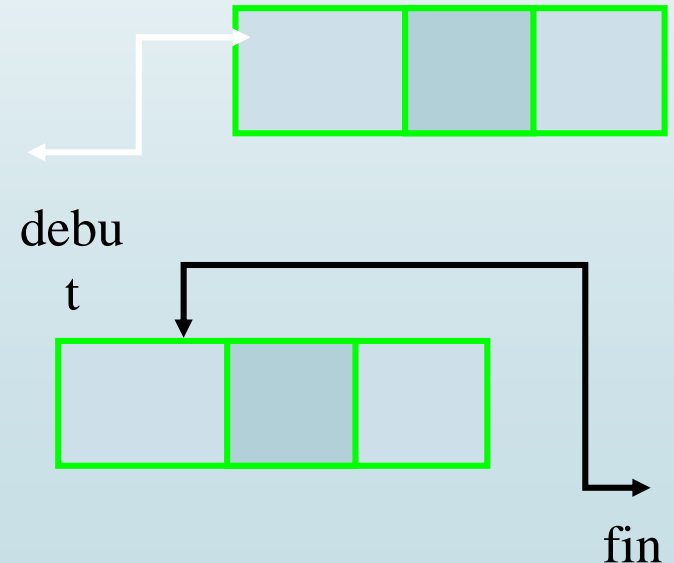
```
typedef struct Node Cell;
```

```
Cell *debut, *fin, *act;
```

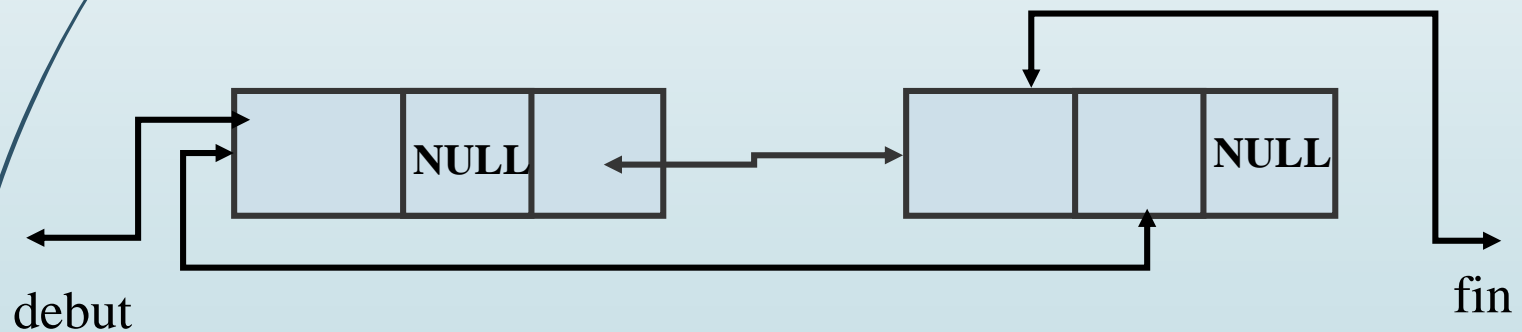
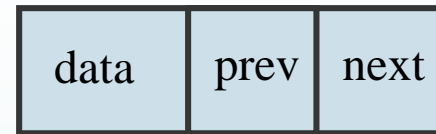
```
int dim = sizeof(Cell);
```

```
debut = (Cell *) malloc(dim);
```

```
fin   = (Cell *) malloc(dim);
```



```
debut->prev = NULL;  
debut->next = fin;  
fin->prev = debut;  
fin->next = NULL;
```



Insérer un élément en fin de liste

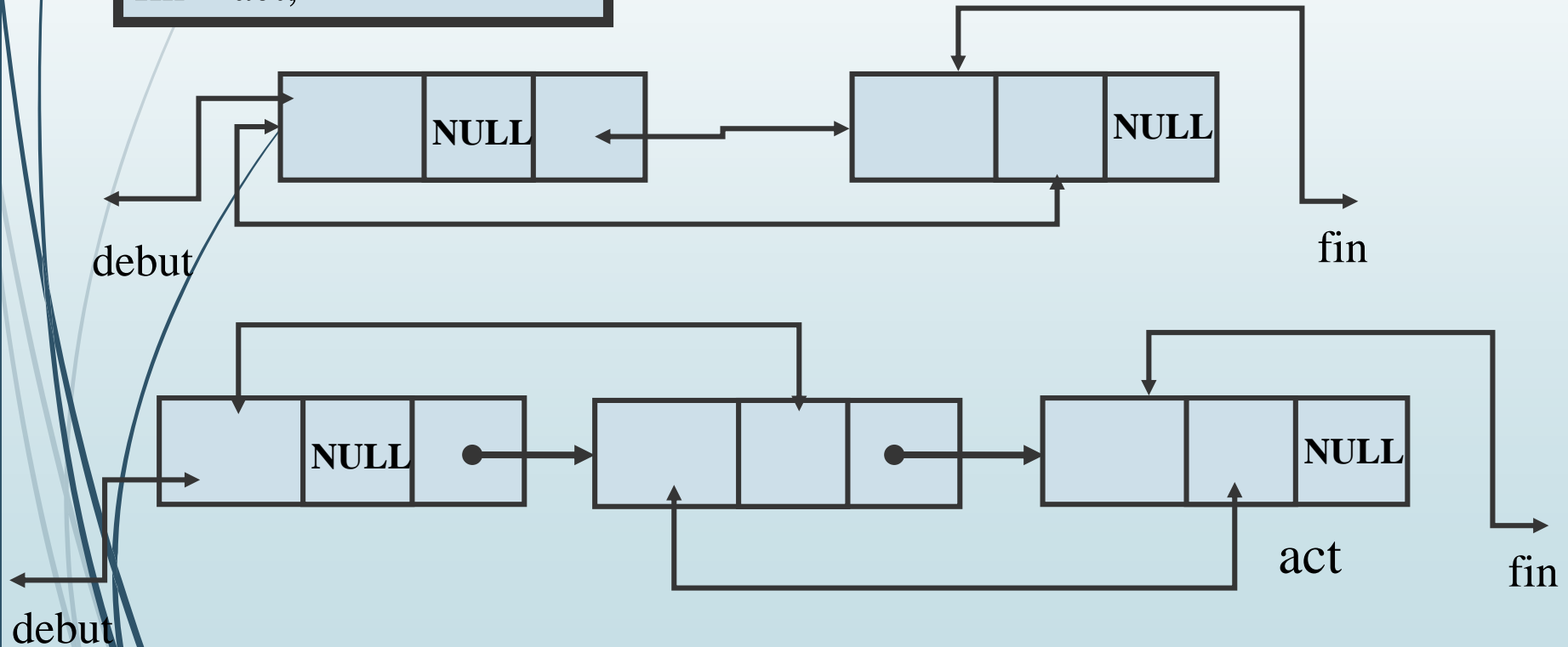
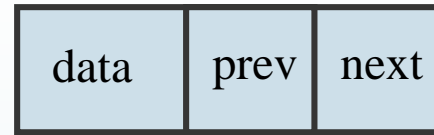
104

```
fin->next = act;
```

```
act->prev = fin;
```

```
act->next = NULL;
```

```
fin = act;
```



Insérer un élément dans la liste pos=qlq

10

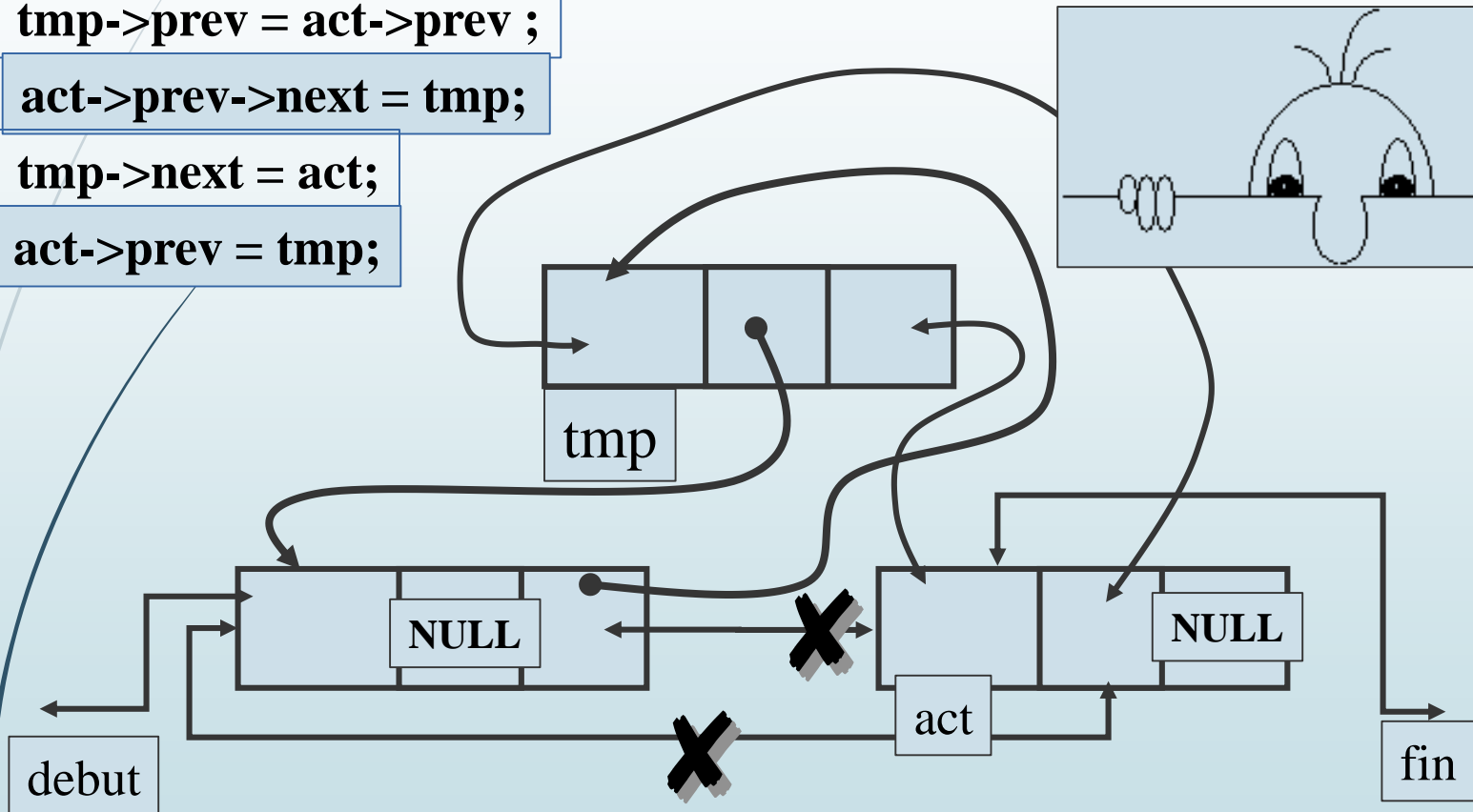
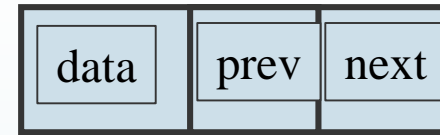
```
Cell *tmp;  
tmp = (Cell *) malloc(dim);
```

```
tmp->prev = act->prev ;
```

```
act->prev->next = tmp;
```

```
tmp->next = act;
```

```
act->prev = tmp;
```



Déclaration des types de données nécessaires

```
struct elem {  
    int valeur ;  
    struct elem * suivant;  
    struct elem *precedent; } ;  
typedef struct elem * liste;
```

Construction De La Liste

A)- liste vide

```
liste create_vide()  
{liste laux=NULL; return  
    laux;}
```

B)- Ajout du premier élément

```
liste add_first(int val)  
{liste l=(liste)malloc(sizeof(struct elem ));  
    if(l!=NULL){  
        l->suivant=NULL; l->prec=NULL;  
        l->valeur=val; } return l;}
```

C)- Ajout entête de liste si la liste n'est pas vide. void add_head(liste *ls,int val)

```
{liste la=(liste)malloc(sizeof(struct elem ));  
    if(la!=NULL){ la->valeur=val ;  
        la->precedent=NULL; la->  
            >suivant=*ls  
        *ls->precedent=la;  
        *ls=la}
```

```
}
```

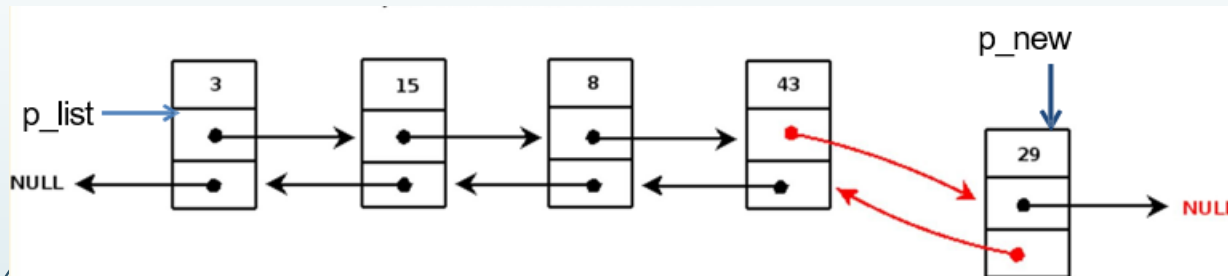
D)- Ajout en fin de liste dans les listes bilatères.

But : insérer une valeur dans une liste triée en ordre croissant a)-

Phase de parcours (à la recherche du dernier bloc).

b)- Création du nouveau bloc (à insérer). d)-

Mise à jour des liens.



ALGORITHMLE

laux : =l ;

TQ laux->suivant <> NIL

faire

 laux: =laux->suivant

FTQ

 Allouer (nouveau)

 Entrer (nouveau-> Valeur)

 nouveau->suivant : = NIL

 nouveau->précédent : =

 laux Laux->suivant : =

 nouveau

PROGRAMME C

```
liste add_last(liste p_list, int data)
```

```
{liste laux=p_list;
```

```
  while(laux->suivant !=NULL )
```

```
    {laux=laux->suivant;}
```

```
  /* Création d'un nouveau nœud */
```

```
  liste p_new = (liste)malloc(sizeof (struc elem));
```

```
  /* On vérifie si le malloc n'a pas échoué */
```

```
  if (p_new != NULL)
```

```
  { /* On 'enregistre' notre donnée */
```

```
    p_new->valeur= data;
```

```
    /* On fait pointer suivant vers NULL */
```

```
    p_new->suivant = NULL;
```

```
    // On fait pointer precedent vers la fin de la
```

```
    liste   p_new->precedent = laux;
```

```
  // On fait pointer la fin liste vers le nouveau
```

```
  nœud   laux->suivant =p_new;
```

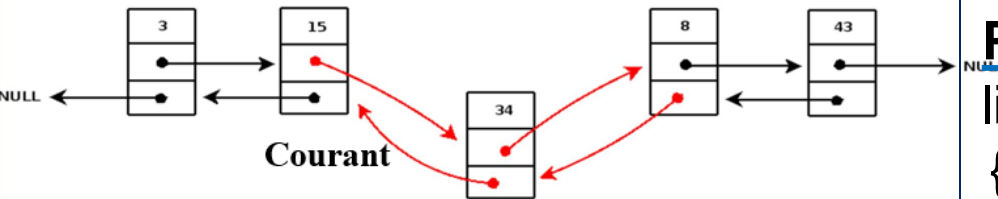
```
  }
```

```
  //on retourne notre nouvelle liste
```

```
  return p_list;
```

```
}
```

E)- Ajout à un endroit quelconque de la liste.



ALGORITHME

laux: =l

TQ (laux->Valeur<>15) ET (laux->suivant <>NIL)

FAIRE

laux := laux->suivant

FTQ

Allouer (nouveau) Entrer (nouveau->valeur)

nouveau->précédent := laux nouveau -

>suivant := laux->suivant

Laux->suivant->précédent := nouveau Laux-

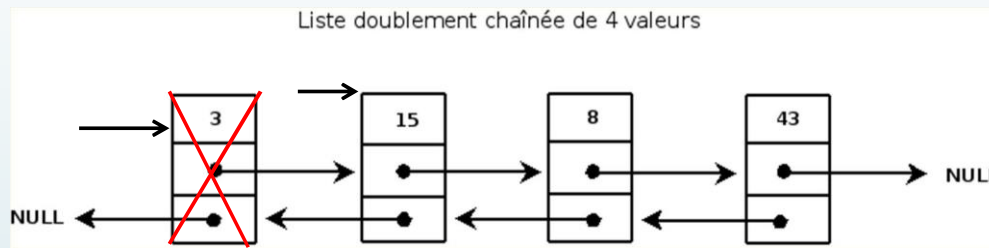
>suivant := nouveau

Programme C

```
liste add_pefore(liste pls, int pval, value)
{liste laux=pls;
While((laux->valeur!=pval)&&(laux-
>suivant!=NULL))
{laux=laux->suivant}
//allocation du nouveau nœud
liste p_new =(liste)malloc(sizeof(struct
elem)); )
// si tout s'est bien passé
if (p_new != NULL)
{ p_new->valeur = value;
p_new-->precedent= laux;
p_new->suivant = laux -
>suivant; laux ->suivant-
>precedent= p_new; laux -
>suivant= p_new;}
return pls;}
```

Destruction dans une liste bilatère

► *Suppression en tête de liste*



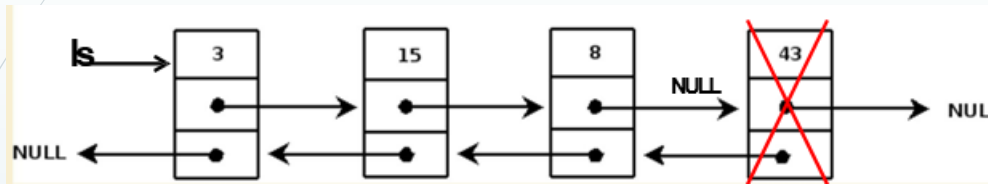
ALGORITHME

```
laux := ls  
ls := ls->suivant libérer (laux)  
ls->Précédent := NIL
```

Programme C

```
liste delete_first(liste pls)  
{liste laux=pls; if (pls !=  
NULL){  
    pls= pls ->suivant;  
    pls->precedent=NULL;  
    free(laux);}  
return ls;}
```

Suppression en fin de liste



ALGORITHME

laux := l

TQ laux->suivant <> NIL faire laux
:= laux->suivant

FTQ

Laux->precedent->suivant : = NIL

Libérer(laux)

PROGRAMME C

Liste delete_last(liste ls)

{ liste laux=ls;

while(laux->suivant!=NULL)

{ laux = laux->suivant; }

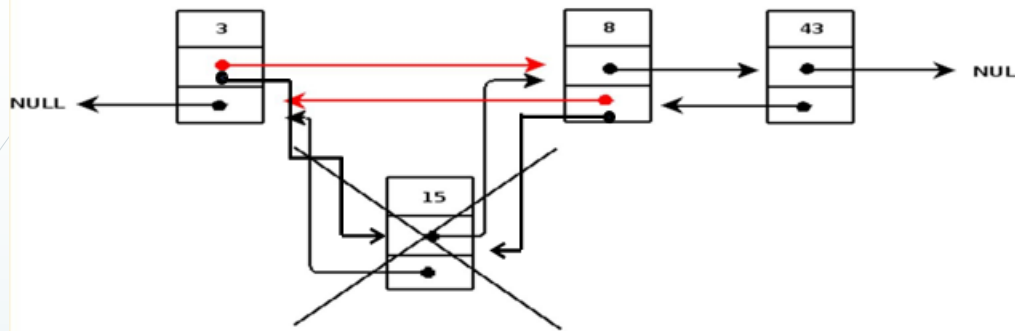
laux -> precedent->suivant=NULL;

free(laux);

return ls }

Attention cet algorithme n'est valable que si la liste contient plus d'un élément.

Supprimer un élément selon sa valeur



ALGORITHME

laux := ls ;
TQ (laux->Valeur <> 15) ET (laux->suivant <> NIL) faire laux:
= laux->suivant

FTQ
laux->precedent->suivant :=
laux->suivant laux->suivant -
>precedent := laux->
>precedent libérer (laux)

Programme C

```
liste delete_pefore(liste pls, int pval)
{liste laux=pls;
While((laux->valeur!=pval)&&(laux-
>suivant!=NULL))
{laux=laux->suivant}
laux->precedent->suivant =
laux->suivant laux->suivant -
>precedent = laux->precedent free
(laux)
return pls;}
```

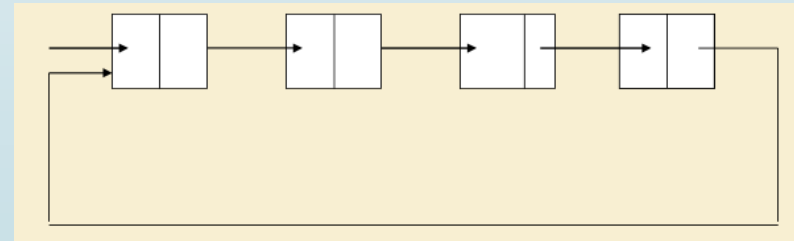

LISTES CIRCULAIRES ...

➤ Définition

- Une liste où le pointeur NUL du dernier élément est remplacé par l'adresse du premier élément est appelée **liste circulaire**.
- Dans une liste circulaire tous les nœuds sont accessibles à partir de n'importe quel autre nœud. Une liste circulaire n'a pas de premier et de dernier nœud.
- Une liste **circulaire** peut être simplement chaînée ou doublement chaînée.
- Noter que la concaténation de deux listes circulaires peut se faire sans avoir à parcourir les deux listes.

Déclaration des types de données nécessaires

```
typedef struct elem {  
    int valeur ;  
    struct elem * suivant;} element ;  
typedef element * anneau;
```



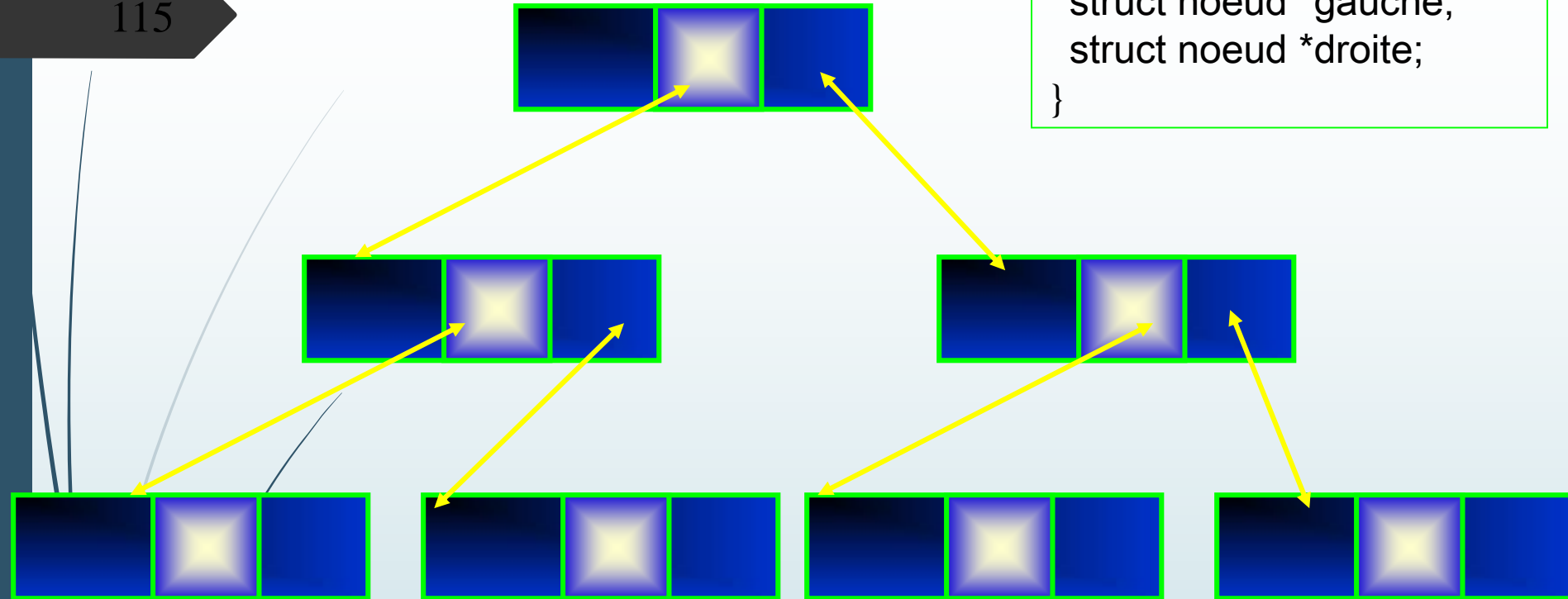


Chapitre 5

Arbres binaires de recherche

Arbre binaire

```
typedef struct noeud {  
    int donnee;  
    struct noeud *gauche;  
    struct noeud *droite;  
}
```

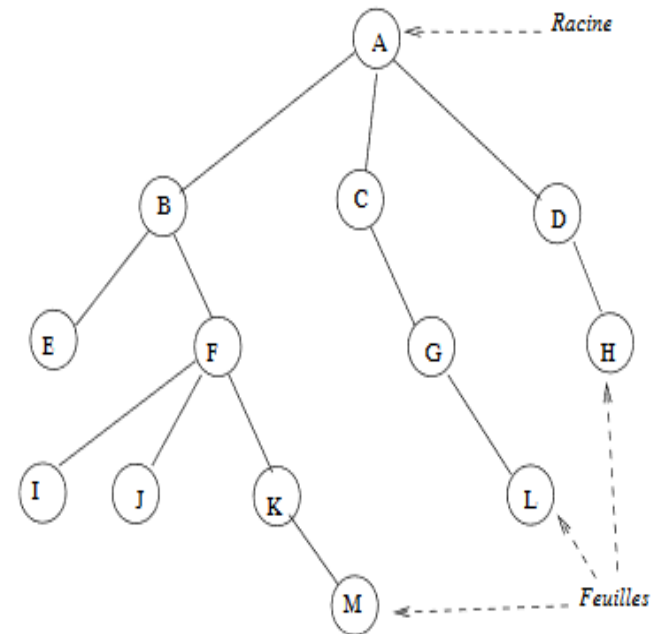


Tri par arbre binaire de recherche ABR



Arbre binaire de recherche

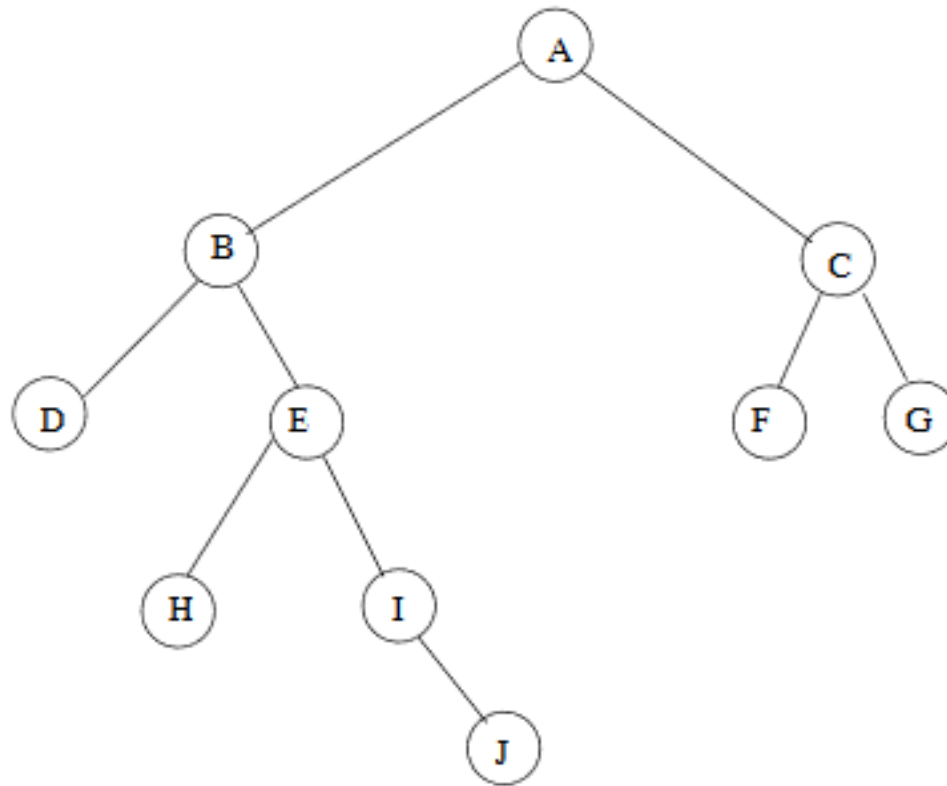
Un arbre est une structure composée de nœuds et de feuilles (nœuds terminaux) reliées par des branches. On le représente généralement en mettant la racine en haut et les feuilles en bas (contrairement à un arbre réel).



- Le nœud A est la racine de l'arbre.
- Les nœuds E, I, J, M, L et H sont des feuilles.
- Les nœuds B, C, D, F, G et K sont des nœuds intermédiaires.
- Si une branche relie un nœud n_i à un nœud n_j située plus bas, on dit que n_i est un ancêtre de n_j .
- Dans un arbre, un nœud n'a qu'un seul père (ancêtre direct).
- Un nœud peut contenir une ou plusieurs valeurs.
- La hauteur (ou profondeur) d'un nœud est la longueur du chemin qui le lie à la racine

Arbres binaires

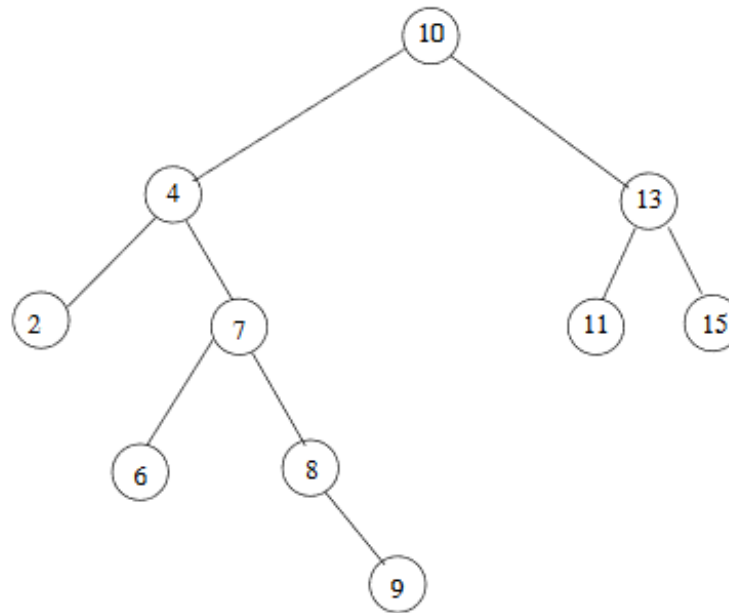
Un arbre binaire est un arbre tel que les nœuds ont au plus deux fils(gauche et droite).



Arbres binaires de recherche

Un arbre binaire de recherche est un arbre qui possède les propriétés suivantes:

- Tous les nœuds du sous-arbre de gauche d'un nœud de l'arbre ont une valeur inférieure ou égale à la sienne
- Tous les nœuds du sous-arbre de droite d'un nœud de l'arbre ont une valeur supérieure ou égale à la sienne



Recherche dans l'arbre

Un arbre binaire de recherche est fait pour faciliter la recherche d'informations. La recherche d'un nœud particulier de l'arbre peut être définie simplement de manière réursive:

Soit un sous-arbre de racine n_i ,

- si la valeur recherchée est celle de la racine n_i , alors la recherche est terminée.

On a trouvé le nœud recherché.

- sinon, si n_i est une feuille (pas de fils) alors la recherche est infructueuse et l'algorithme se termine.

- si la valeur recherchée est plus grande que celle de la racine alors on explore le sous-arbre de droite c'est à dire que l'on remplace n_i par son nœud fils de droite et que l'on relance la procédure de recherche à partir de cette nouvelle racine

- de la même manière, si la valeur recherchée est plus petite que la valeur de n_i , on remplace n_i par son nœud fils de gauche avant de relancer la procédure.

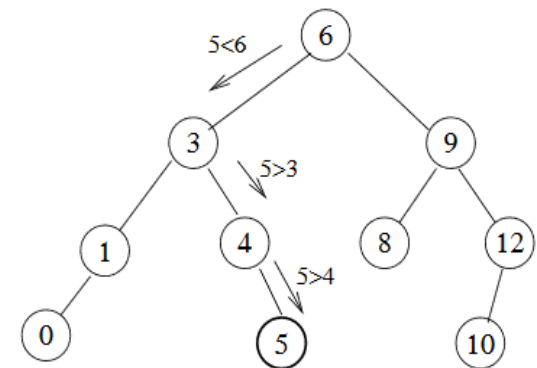
Si l'arbre est équilibré chaque itération divise par 2 le nombre de nœuds candidats. La complexité est donc en $O(\log_2 n)$ si n est le nombre de nœuds de l'arbre

Ajout d'un élément

Pour conserver les propriétés d'un arbre binaire de recherche nécessite de, l'ajout d'un nouvel élément ne peut pas se faire n'importe comment.

L'algorithme récursif d'ajout d'un élément peut s'exprimer ainsi:

- soit x la valeur de l'élément à insérer.
- soit v la valeur du nœud racine n_i d'un sous-arbre.
 - si n_i n'existe pas, le créer avec la valeur x . fin.
 - sinon
 - si x est plus grand que v ,
 - remplacer n_i par son fils droit.
 - recommencer l'algorithme à partir de la nouvelle racine.
 - sinon
 - remplacer n_i par son fils gauche.
 - recommencer l'algorithme à partir de la nouvelle racine



Implémentation

En Langage C, un nœud d'un arbre binaire peut être représenté par une structure contenant champ donnée et deux pointeurs vers les deux nœuds fils

```
struct s_arbre
{
    int valeur;
    struct s_arbre * gauche;
    struct s_arbre * droit;
};
typedef struct s_arbre t_arbre;
```

Implémentation

123

La fonction d'insertion qui permet d'ajouter un élément dans l'arbre et donc de le créer de manière à ce qu'il respecte les propriétés d'un arbre binaire de recherche peut s'écrire ainsi:

```
void insertion(t_arbre ** noeud, int v)
{
    if (*noeud==NULL) /* si le noeud n'existe pas, on le crée */
    {
        *noeud=(t_arbre*) malloc(sizeof(t_arbre));
        (*noeud)->valeur=v;
        (*noeud)->gauche=NULL;
        (*noeud)->droit=NULL;
    }
    else
    {
        if (v>(*noeud)->valeur)
            insertion(&(*noeud)->droit,v ); /* aller a droite */
        else
            insertion(&(*noeud)->gauche,v); /* aller a gauche */
    }
}
```