Cours d'algorithmique et de programmation

Frédéric Lardeux

Bureau H208

lardeux@info.univ-angers.fr

Objectif principal

- Apprentissage de méthodes de programmation
- Langage : C

Plan

1) Rappels
2) Conditionnelle
3) Tant que / Pour / Répéter
4) Tableaux
5) Procédures / Fonctions
6) Récursivité
7) Algorithmes de tri

Plan

8) Structures

9) Pointeurs

10) Listes / Piles / Files

11) Arbres

12) Graphes

13) Complexité

1) Rappels

Qu'est-ce qu'un algorithme ?

Algorithme

- Définition : suite finie d'opérations élémentaires permettant de résoudre un problème donné
- Al'Khawarizmi : mathématicien arabe du IX siècle
- Exemple déjà connu : algorithme d'Euclide de recherche du PGCD de deux entiers

Conception d'un algorithme

- Réfléchir aux informations à manipuler
- Analyser le problème en le décomposant en sous problèmes
- Écrire un algorithme compréhensible
- Penser à l'utilisateur

Exemple : PGCD de de A et B

	A	B	
Et. 1	24	9	On cherche le plus grand
Et. 2	15	9	A est remplacé par A-B
Et. 3	6	9	
Et. 4	6	3	B est remplacé par B-A
Et. 5	3	3	A=B le pgcd est 3

Décomposition du problème

- Prendre deux nombres A et B
- Mettre en œuvre l'algorithme d'Euclide
- Fournir les résultats

1.1) Rappels

Que trouve-t-on dans un algorithme ?

Variables

- Notion de variable
  - Exemple Poids, X, mon_age ...
- La valeur d'une variable est stockée dans la mémoire à une certaine adresse
  - &Poids, &X, &mon_age ...

Instructions simples

- Nos 3 premières instructions indispensables : lire une valeur, afficher une valeur, affecter une valeur à une variable
- Lire et afficher nécessite des informations sur la bibliothèque standard

#include <stdio.h>

Instructions simples : lecture

- **scanf("format", &var);**
- Le programme va s'interrompre et attendre que l'utilisateur tape une valeur au clavier et l'affectera à la variable désignée selon le format spécifié.

✓ **&var** plutôt que **var** car l'instruction scanf a besoin de savoir où ranger la valeur

Instructions simples : écriture

- **printf(Expression);**
- **Expression** peut être une valeur, le contenu d'une variable, le résultat d'un calcul, un message, une expression mixte
- Exemple :

```
printf("Je sais que 1 + 1 =%d",1+1);
```

Instructions simples : affectation

- **var = Expression;**
- **Expression** est évaluée et sa valeur rangée dans la variable **var**
- **Expression** peut être une valeur, le contenu d'une autre variable, le résultat d'un calcul

1.2) Rappels

Comment écrire un programme en C ?

Structure d'un programme

Bibliothèque #include <… .h>

Déclarations :

constante #define PI 3.14;

variable int X;

Corps :

```
void main()
{
  <Instruction 1>;
  /*Commentaires*/
  <Instruction n>;
}
```

Exercice

Ecrire l'algorithme qui détermine l'âge d'une personne après lui avoir demandé l'année courante et son année de naissance.

Programme 1

```
#include <stdio.h>
int anneec,anneen,age;
void main(int argc, char * argv[])
{
        printf("Donner l'année courante : \n");
        scanf("%d",&anneec);
        printf("Donner votre année de naissance : \n");
        scanf("%d",&anneen);
        age= anneec-anneen;
        printf("Vous avez %d ans\n",age);
}
```

1.3) Rappels

Types et opérations élémentaires

Types de base

- Entiers : int %d
- Réels : float %f
  double
- Caractères : char %c

Valeurs possibles

- Entiers : Sous ensemble de Z
- Réels : Sous ensemble de R
- Caractères : "a", "A", "§" ...

Ne pas choisir un réel si un entier suffit !

Opérations

- Entiers : +, -, *, / *(division entière)*, %
- Réels : +, -, *, /
- Caractères : vues ultérieurement

Opérateurs de comparaison

- Entiers :
- Réels : ==, <, >, <=, >=, !=
- Caractères :

Exercice

Ecrire l'algorithme qui demande à un étudiant la note obtenue dans chacune des matières suivantes : informatique, mathématiques, chimie. Affichez la moyenne de cet étudiant sachant que les matières ont respectivement des coefficients de 3, 3, 2.

Programme 2

```
#include <stdio.h>

float info,maths,chimie,moy;

void main(int argc, char * argv[])
    {
        printf("Donner la note d'info: \n");
        scanf("%f",&info);
        printf("Donner la note de maths: \n");
        scanf("%f",&maths);
        printf("Donner la note de chimie: \n");
        scanf("%f",&chimie);
        moy= (3*info+3*maths+2*chimie)/8;
        printf("Vous avez une moyenne de %f \n",moy);
    }
```

2) Conditionnelle

- Lecture, écriture, affectation : instructions séquentielles
- Branchement conditionnel (choix binaire)

If (...) ... else ...

- Complète :

```
if (<condition>)
   <instr1>;
else
   <instr2>;
```

- Incomplète :

```
if (<condition>)
   <instr1>;
```

- Remarque : les { } sont obligatoires lorsqu'il y a plusieurs instructions

```
if (<condition>)
{
<instr1>;
<instr2>;
}
```

Conditionnelles imbriquées

```
if (<cond1>) {
        if (<cond2>)
            <inst1>;
        else
            <inst2>;
}
else <inst3>;
```

Cas possibles :

cond1 et cond2 → inst1

cond1 et non cond2 → inst2

non cond1 → inst3

Condition

- Expression booléenne
- Exemples :
  - `(Prix > 2)`
  - `(nb == 5)`
  - `((Prix > 2) && (nb == 5))`
- `((Prix > 2) && (nb == 5))`
- `Remarque :`

  A == faux équivaut à ! A

  A == vrai équivaut à A
  - `((Prix > 2) && (trouve))`

Tables de vérité

		NON	ET	OU
A	**B**	**! A**	**A && B**	**A || B**
vrai	vrai	faux	vrai	vrai
faux	vrai	vrai	faux	vrai
vrai	faux	faux	faux	vrai
faux	faux	vrai	faux	faux

Exercice

Problème :

Demander à l ’utilisateur de donner un nombre entier et lui répondre si ce nombre est pair ou impair.

Algo :

Saisie d ’un entier

Test de parité

Affichage du résultat

Programme 3

```
#include <stdio.h>
int X;
void main(int argc, char * argv[])
{
   printf("Donner votre nombre: ");
   scanf("%d",&X);
   if X%2==0 /* X est pair */
      printf ("Nombre pair");
   if X%2!=0 /* X est impair */
      printf ("Nombre impair");
}
```

Programme 3 bis

```
#include <stdio.h>
int X;
void main(int argc, char * argv[])
{
   printf("Donner votre nombre: ");
   scanf("%d",&X);
   if X%2==0 /* X est pair */
     printf ("Nombre pair");
   else /* X est impair */
     printf ("Nombre impair");
}
```

Exercice : OU exclusif

A	B	A Xor B
vrai	vrai	faux
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

Écrire un algorithme qui demande deux valeurs booléennes (1→vrai, 0→faux) à l'utilisateur et qui retourne le résultat de l'opérateur xor sur ces valeurs.

Programme 4

```
#include <stdio.h>

int A,B,Xor;

void main(int argc, char * argv[])
{
   printf("Donner A (0/1) :");
   scanf("%d",&A);
   printf("Donner B (0/1) :");
   scanf("%d",&B);
```

Programme 4 (suite)

```
if (A==1)
          {
               if (B==1) Xor=0;
               else Xor=1;
          }
     else
          if (B==1) Xor=1;
          else Xor=0;
     /* Décodage du résultat */
     if (Xor==1)
          printf(" A Xor B est vrai ");
     else
          printf(" A Xor B est faux ");
}
```

Programme 4 (optimisation)

```
#include <stdio.h>

int A,B;

void main(int argc, char * argv[])
{
   printf("Donner A (0/1) :");
   scanf("%d",&A);
   printf("Donner B (0/1) :");
   scanf("%d",&B);
   if (((A==1) || (B==1)) && (! ((A==1) && (B==1))))
      printf(" A Xor B est vrai ");
   else
      printf(" A Xor B est faux ");

}
```

Switch

- But : éviter les imbrications de conditionnelles
- Permet de traiter directement des choix qui ne sont plus binaires
- Instruction à choix multiples

Syntaxe du switch

```
Switch expression {

case constante1:
    <inst1>;
    break;
  case constante2:
    <instn>;
    break;
  default :
    <inst0>;
    break;
}
```

break permet une sortie immédiate du swith

Exercice

- Ecrire un programme qui retourne la monnaie utilisée en fonction du pays choisie.

Algo :

Choisir un pays

Si 1 (France) afficher euro

Si 2 (Etats-Unis) afficher dollar

Si 3 (Mexique) afficher Peso

Si 4 (Chine) afficher Yen

Programme 5

```
#include <stdio.h>
int pays;
void main(int argc, char * argv[])
{
 printf("Choisisez un pays (1->France, 2->Etats-Unis, 3->Mexique, 4->Chine) : ");
 scanf("%d",&pays);
 switch (pays)
            {
            case 1:  printf("euro\n");
                        break;
            case 2:  printf("dollar\n");
                        break;
            case 3:  printf("peso\n");
                        break;
            case 4:  printf("yen\n");
                        break;
            default :printf("mauvaise sélection\n");
                        break;
            }
}
```

Exercice

- Ecrire un programme qui donne le nombre de jours dans un mois.

Algo :

Saisir le numéro du mois

Si le numéro est 1,3,5,7,8,10,12 afficher 31

Si le numéro est 4,6,9,11 afficher 30

Si le numéro est 2 afficher 28 ou 29

Programme 6

```
#include <stdio.h>
int mois;
void main(int argc, char * argv[])
{
printf("Donner un numéro de mois :");
scanf("%d",&mois);
switch (mois)
  {
  case 1:case 3:case 5:case 7: case 8:case 10: case 12:
          printf("31 jours\n");
          break;
  case 4:case 6:case 9:case 11:
          printf("30 jours\n");
          break;
  case 2:
          printf("28 ou 29 jours");
          break;
  }
}
```

3) Tant que / Pour / Répéter

- On peut souhaiter qu'une action soit effectuée à plusieurs reprises : notion de boucle
- En C trois manières différentes :
  - ✓ boucle tant que
  - ✓ boucle pour
  - ✓ boucle répéter

Boucle tant que : syntaxe

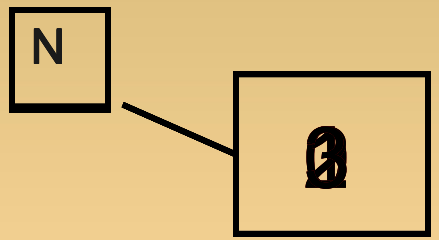
```
While <Condition>
    <inst>;


While <Condition>
    {
        <inst1>;
        …
        <instn>;
    }
```

Boucle tant que : sémantique

- On teste la condition
- Si la condition est vraie on effectue l ’instruction/le bloc d ’instructions et on revient au début de la boucle
- On teste la condition
- ….
- Quand la condition n ’est plus vraie on passe à l’instruction suivante

Exemple

```
#include <stdio.h>
int N;
main()
{
    N=0;
    while N<3
        {
        printf("un tour \n");
        N=N+1;
        }
    printf("Fini");
}
```




```
un tour
un tour
un tour
Fini
```

Retour à Euclide

- Rappel du principe général
  - Prendre deux nombres A et B
  - Mettre en œuvre l'algorithme d'Euclide
  - Fournir les résultats
- Raffinement
  - Tant que A<>B faire
  - Si A>B remplacer A par A-B dans le cas contraire remplacer B par B-A

Exercice

Ecrire un programme qui calcule le pgcd en utilisant l'algorithme d'Euclide.

Programme 7

```
#include <stdio.h>

int a,b;

void main(int argc, char * argv[])
    {
    printf("Donner deux nombres\n");
    scanf("%d %d",&a,&b);
    while (a!=b)
        {
        if (a<b)
            {
            b=b-a;
            }
        else
            {
            a=a-b;
            }
        }
    printf("Le pgcd est %d\n",a);
    }
```

Boucle pour : syntaxe

```
int i;
for (i=val_init;
      i<val_max;
       val que doit prendre i après un passage)
   {
   <instruction>;
   }
```

Exemple :

```
int i;
for (i=0;i<10;i++)
   {
   printf("%d\n",i);
   }
```

```
0
1
2
...
9
```

Boucle pour : sémantique

- On initialise le compteur
- Si la condition est vraie
  - on effectue l ’instruction/le bloc d ’instructions
  - on incrémente (décrémente) le compteur
  - on revient au début de la boucle
- On teste la condition
- ….
- Quand la condition n ’est plus vraie on passe à l’instruction suivante

Exercice

Ecrire un programme qui affiche les tables de multiplication.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	36	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Programme 8

```
#include <stdio.h>

int a,b;

void main(int argc, char * argv[])
    {
    for (a=1;a<11;a++)
        {
        for (b=1;b<=10;b++)
            printf("%d ",a*b);
        printf("\n");  //retour à la ligne
        }
    }
```

Boucle répéter : syntaxe

```
do
   {
   <instruction>;
   }

while (<condition>);
```

Exemple :

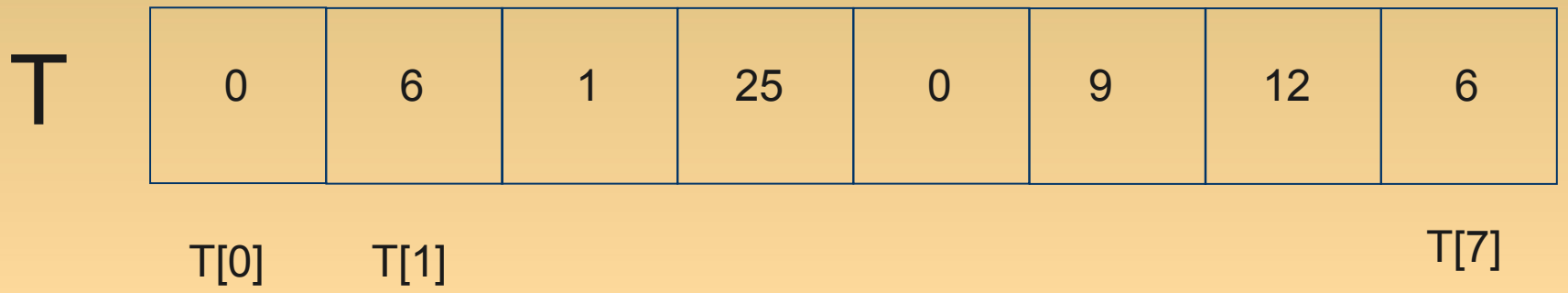
```
int i;
i=0;
do
   {
   printf("%d\n",i);
   i++;
   }
while (i<10);
```

```
0
1
2
...
9
```

4) Tableaux

- Regrouper en une même entité un grand nombre de donnée de même type
- Un **tableau** à une dimension de taille n = une structure composée de n éléments **tous de même type**
- Notion de donnée structurée

Exemple



- Tableau contenant 8 entiers
- Chaque T[i] se comporte comme une variable entière

Syntaxe

- 2 manières de définir un tableau :
  - manière directe :

    int tabent[8];

    définition d'un nouveau type
  - manière indirecte :

    typedef int tableau[8];

    tableau tabent;

    Très utile lorsqu'il y a beaucoup de tableaux ayant les mêmes caractéristiques

Remarques

- Les instructions de lecture, écriture peuvent s'appliquer à chacun des éléments mais pas au tableau tout entier.
  - ~~printf("%d \n",T);~~ printf("%d \n",T[0]);
- Des expressions peuvent être utilisées pour indicer les éléments du tableau.
  - Ex : T[3*X+1].

Algorithmes classiques

- Manipulations de base autour d'un tableau d'entiers : saisie et affichage des éléments.
- Saisie :

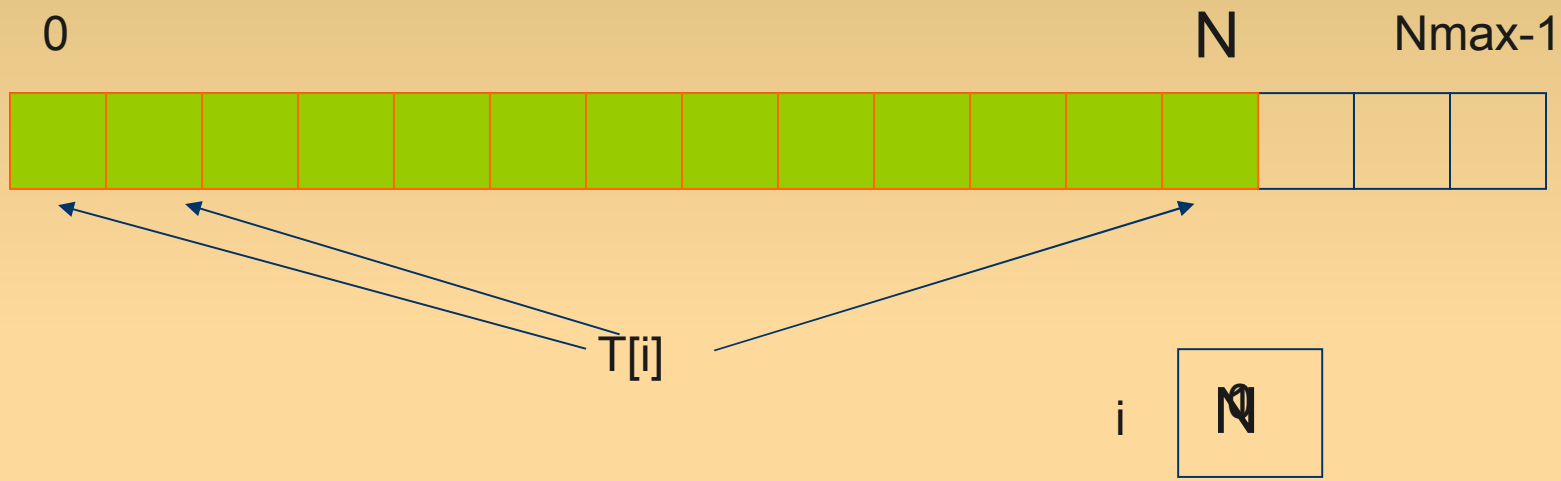
  Tant que tout les éléments n'ont pas été saisis

  {saisir un élément puis passer au suivant}

- Affichage : idem

Principes de bases

Déclarer un tableau de Nmax cases



Utilisation d'une partie 0..N (partie utile) en fonction des besoins

Le parcours se fait grâce à une variable indice qui variera de 0 à N pour accéder aux différents éléments.

Programme : déclarations

```
#include <stdio.h>

#define Nmax 30

int tableau[Nmax]; /* La taille du tableau
                      est limitée à 30 éléments*/

int n; // Nombre d'éléments réellement saisis

int i; // Pour parcourir le tableau
```

Programme : saisie

```
void main(int argc, char * argv[])
   {
   printf("Combien d'éléments voulez-vous saisir?\n");
   scanf("%d",&n);
   /* Saisie des éléments */
   for(i=0;i<n;i++)
   i=0;
   while(i<n)
       {
       printf("Donnez le %d ème élément\n",i+1);
       scanf("%d",&tableau[i]);
       }
       scanf("%d",&tableau[i]);
       i=i+1;
       }
```

Programme : affichage

```
/* Affichage des éléments */
    for(i=0;i<n;i++)
    while(i<n)       printf("%d ",tableau[i]);
    printf("\n");
}           printf("%d ",tableau[i]);
            i++;
        }
    printf("\n");
}
```

Exercice

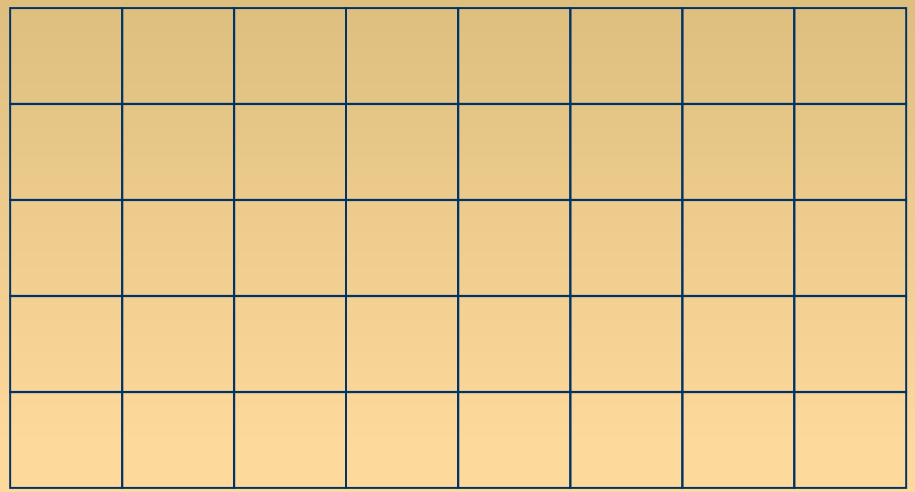
Une machine possède 65336 ports TCP. Les 1001 premiers (de 0 à 1000) sont réservés et l'utilisateur ne doit pas y affecter de connexions. Par contre, les ports de 1001 à 65336 sont libres. On peut donc créer des liens avec d'autres machines en les utilisant.

Ecrire un programme qui demande à l'utilisateur les 10 ports dont il souhaite se servir et qui affiche ceux qui sont réservés.

Programme 9

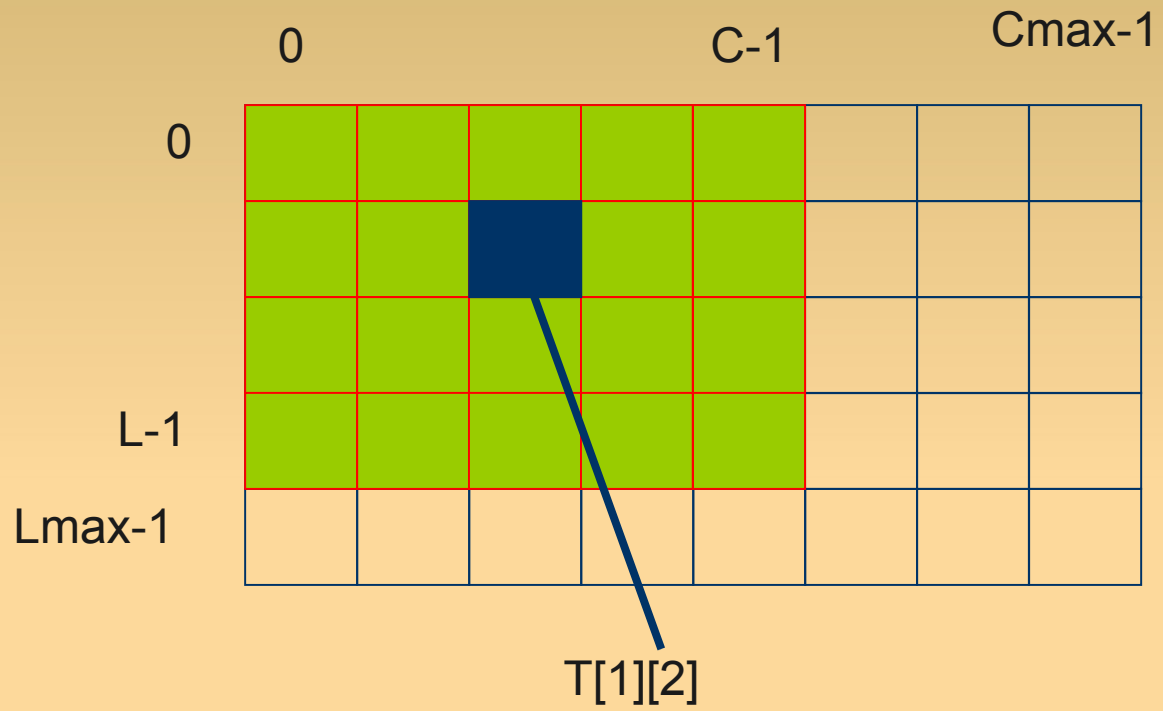
```
#include <stdio.h>
#define MAX 10// constante
int ports[MAX];
int i;
void main(int argc, char * argv[])
    {
    // SAISIE
    for (i=0;i<10;i++)
        {
        printf("Donnez le port n°%d : \n",i);
        scanf("%d",&ports[i]);
        }
    // AFFICHAGE
    for (i=0;i<10;i++)
        if (ports[i] <= 1000)
            printf(“Le port %d est réservé.\n”,ports[i]);
    }
```

Tableaux à deux dimensions (Matrice)



```
#define Lmax 5
#define Cmax 8 {Nb de lignes et de colonnes max.}

typedef int Matrice[Lmax][Cmax];
```

Utilisation



Accès aux éléments par deux indices

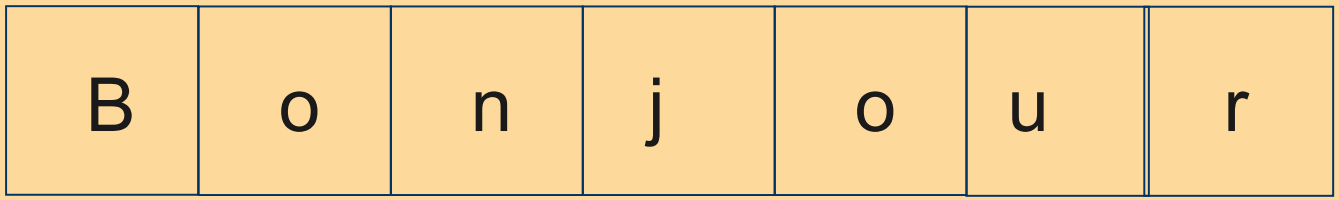
Parcours d'une matrice

Ecrire un algorithme qui permet de remplir une matrice 5x3.

```
#include <stdio.h>

#define sizeL 5
#define sizeC 3

int mat[sizeL][sizeC];
int i,j;

void main(int argc, char * argv[]))
{
//SAISIE
for(i=0;i<sizeL;i++)
     for(j=0;j<sizeC;j++)
        {
        printf("Valeur de la case (%d,%d) : ",i,j);
        scanf("%d",&mat[i][j]);
        }

}
```

Chaînes de caractères

- Tableau à une dimension contenant des caractères
  - Ex : ‘ Bonjour ’



Déclaration

```
#include <string.h>
char mot[N];
```

- N ne doit pas dépasser 255
- Lorsque l'on utilise la déclaration [N], N doit être une constante.

Opérations et fonctions

- Concaténation :

  strcat("Bon","jour") = "Bonjour"
- Comparaison (ordre lexicographique)

  "Paris" < "Pascal"
- Longueur : fonction strlen(ch) renvoie un entier qui est la longueur de la chaîne ch
  - printf(“%d”,strlen(“toto“)); => 4

Exemple : Déterminer si un mot est un palindrome (1/2)

- Palindrome : mot qui se lit dans les deux sens
  - exemple: LAVAL

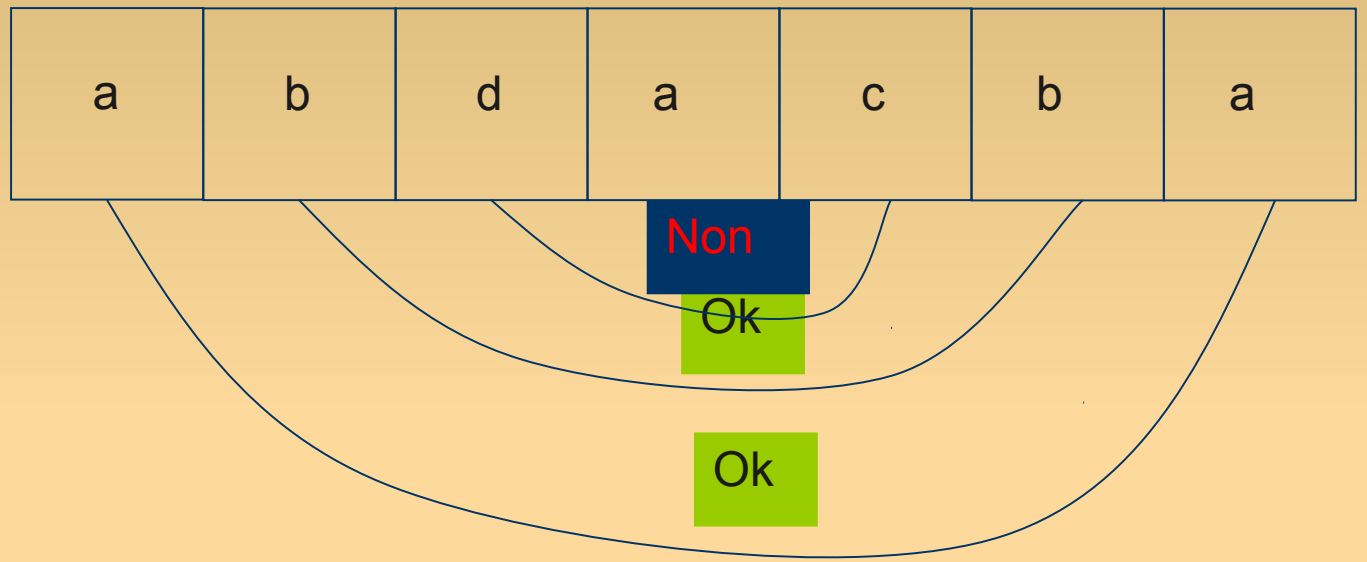
- **Algorithme**

Saisir un mot

Effectuer le test palindrome

Afficher le résultat

Exemple : Déterminer si un mot est un palindrome (2/2)



Tester l 'égalité entre deux lettres « diamétralement » opposées

Le milieu du mot est soit à long / 2 si long est paire soit à long / 2 +1. Donc E(long/2).

Exercice

Ecrire un programme qui teste si le mot saisi par l'utilisateur est un palindrome.

Programme 10

```
#include <stdio.h>
#include <string.h>
char mot[50];
int longueur,i,palin;
void main(int argc, char * argv[])
{
	printf("Donnez un mot :\n");
	scanf("%s",mot);
	longueur=strlen(mot);
	i=0;
	palin=1;
	while ((i<=longueur/2) && (palin==1))
		{
			if (mot[i]!=mot[longueur-i-1])
				palin=0;
			i++;
		}
	if (palin==1)
		printf("C'est un palindrome");
	else
		printf("Ce n'est pas un palindrome");
}
```

5) Procédures et fonctions

Comment décomposer un algorithme ?

Sous-programme

- Le langage possède un nombre d ’instructions limité

  => Créer de nouvelles instructions

- Analyse Descendante (Programmation modulaire)

Problème

Sous-Prob Sous-Prob Sous-Prob Sous-Prob

Sous-programme

- Sous bloc d ’instructions séparé du bloc principal
- Deux points
  - définition
  - appel du sous programme
- Deux types de sous-programmes :
  - les fonctions
  - les procédures

Définition du sous-programme

Bibliothèque #include <… .h>

Déclarations :

constante `#define PI 3.14;`

`variable` `int X;`

Prototype :

entête du sous-programme;

Corps :

entête sous-programme

{

définition du sous-programme

}

```
void main(int argc, char * argv[])
{
   ...
}
```

Les fonctions : syntaxe

Type ma_*fonction* (paramètres formels avec types)

{

Variables locales utilisées

Instructions de la fonction

return expression;

}

Les fonctions : paramètres

Type ma_*fonction* (paramètres formels avec types)

Paramètres formels et effectifs

Ex :

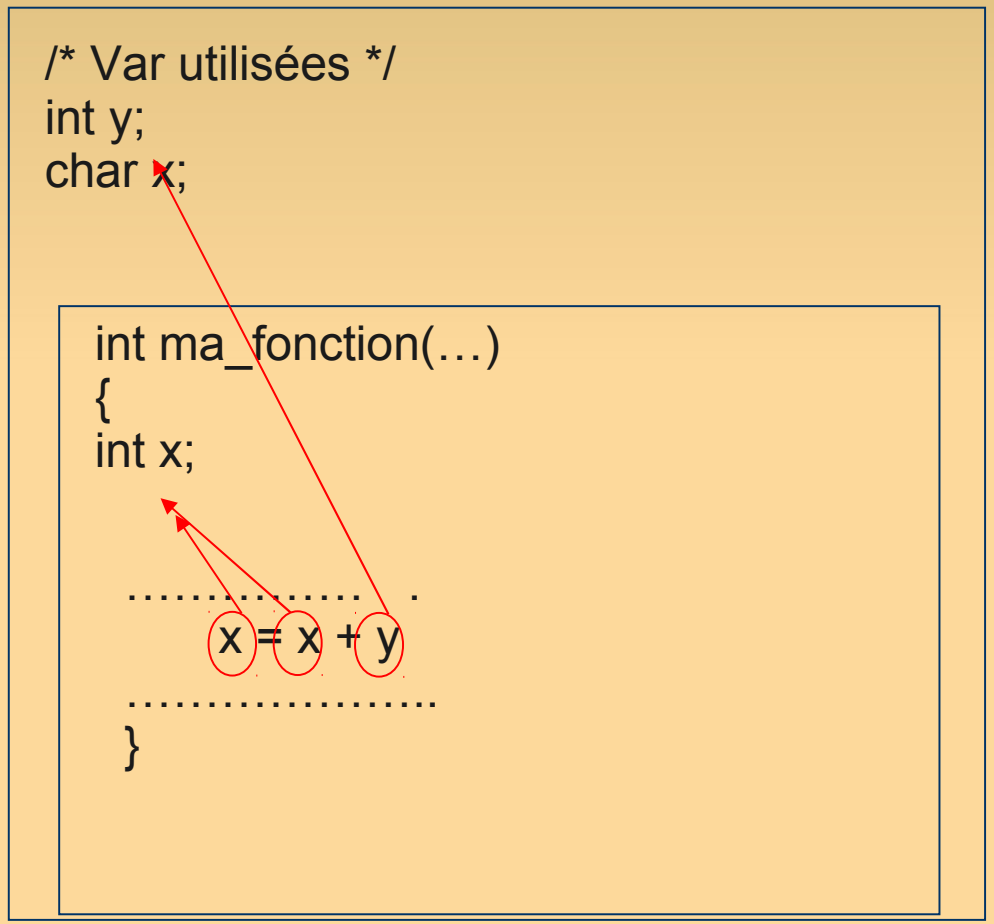
f : N N
f(x)=2*x+3

f(3)
f(a) avec a=3

Attention respecter l'ordre lors de l'appel

Variables locales

Règles de visibilité



Appel d'une fonction

- Correspond à **une expression dont on évalue la valeur**

Ex :

  - printf("Le minimun est : %d",min(x,y));
  - x = x-min(x,y);
- On ne fait **jamais d'affichage** du résultat **dans la fonction**
- **Jamais de lecture**

  les **données sont passées en paramètres**

Exercice

Ecrire une fonction qui retourne la valeur absolue d'un nombre.

Programme 11

```
int valeur_absolue(int x)
{
  if (x < 0)
        return –x;
  else
        return x;
}
```

Notion de Procédure

- Exemple :
  - *Écrire un programme qui calcule la somme de deux matrices saisies par l'utilisateur*
- Analyse et découpage :

But des sous programmes

- Saisie
  {stocke une matrice carrée, fournie par l'utilisateur, dans un tableau à deux dimensions}
- Somme
  {calcule la somme de 2 matrices carrées}
- Affichage
  {Affiche, à l'écran, une matrice carrée stockée dans un tableau à deux dimensions}

Paramètres formels

- **Saisie :** {stocke dans un tableau **M** de type Matrice les **n** lignes et n colonnes d'une matrice carrée, (n demandé à l'utilisateur dans le programme principal **ou** dans le sous-programme)}
  - paramètres formels : M, n
- **Somme :** {calcule la somme **Z** de 2 matrices carrées **X** et **Y** de dimension **n,** X,Y et Z sont stockées dans un tableau de type Matrice}
  - paramètres formels : X, Y, n, Z
- **Affichage :** {affiche une matrice carrée **M**, de dimension **n**, stockée dans un tableau de type Matrice }
  - paramètres formels : M, n

Statut des paramètres

- Saisie : paramètres formels M, n
  - Les données stockées dans M résultent de l'action du sous programme de saisie, **M est un résultat**
  - Si n est demandé dans le programme principal, la donnée de n est nécessaire à la réalisation du sous programme de saisie, **n est ici une donnée**
  - Par contre si n est demandé dans le sous programme, n comme M résulte du sous programme de saisie, **n est alors un résultat**
- Affichage : paramètres formels M, n
  - la donnée de M ainsi que celle de n sont nécessaires à la réalisation du sous programme d'affichage, **M et n sont ici des données**
- Somme : paramètres formels X, Y, n, Z ??

Statut des paramètres

- Somme : paramètres formels X, Y, n, Z
  - X, Y ainsi que n sont nécessaires à la réalisation du sous programme de somme, **X, Y et n sont ici des données**
  - Les données stockées dans Z résultent de l'action du sous programme de saisie, **Z est un résultat**

Statuts des paramètres

- Trois statuts possibles : donnée, résultat ou donnée modifiée
  - Donnée : si la valeur du paramètre doit être donnée au sous-programme pour qu'il puisse réaliser l'action dont il est chargé.
  - Résultat : si la valeur du paramètre résulte de l'action effectuée par le sous-programme.
  - Donnée modifiée : si la valeur du paramètre doit être donnée et qu'elle est modifiée par l'action effectuée par le sous-programme.

Lors de la déclaration du sous programme on indique ce statut des paramètres

Statuts des paramètres

Il n'existe que deux statuts en C (on ne distingue pas les résultats et les données modifiées)

Donnée modifiée : *

Résultat : *

Donnée :

Entêtes des sous-programmes

- **void Saisie (Matrice *M, int *n);**
- **void Somme (Matrice X, Matrice Y, int n, Matrice *Z);**
- **void Affichage (Matrice M, int n);**

Passage de paramètres

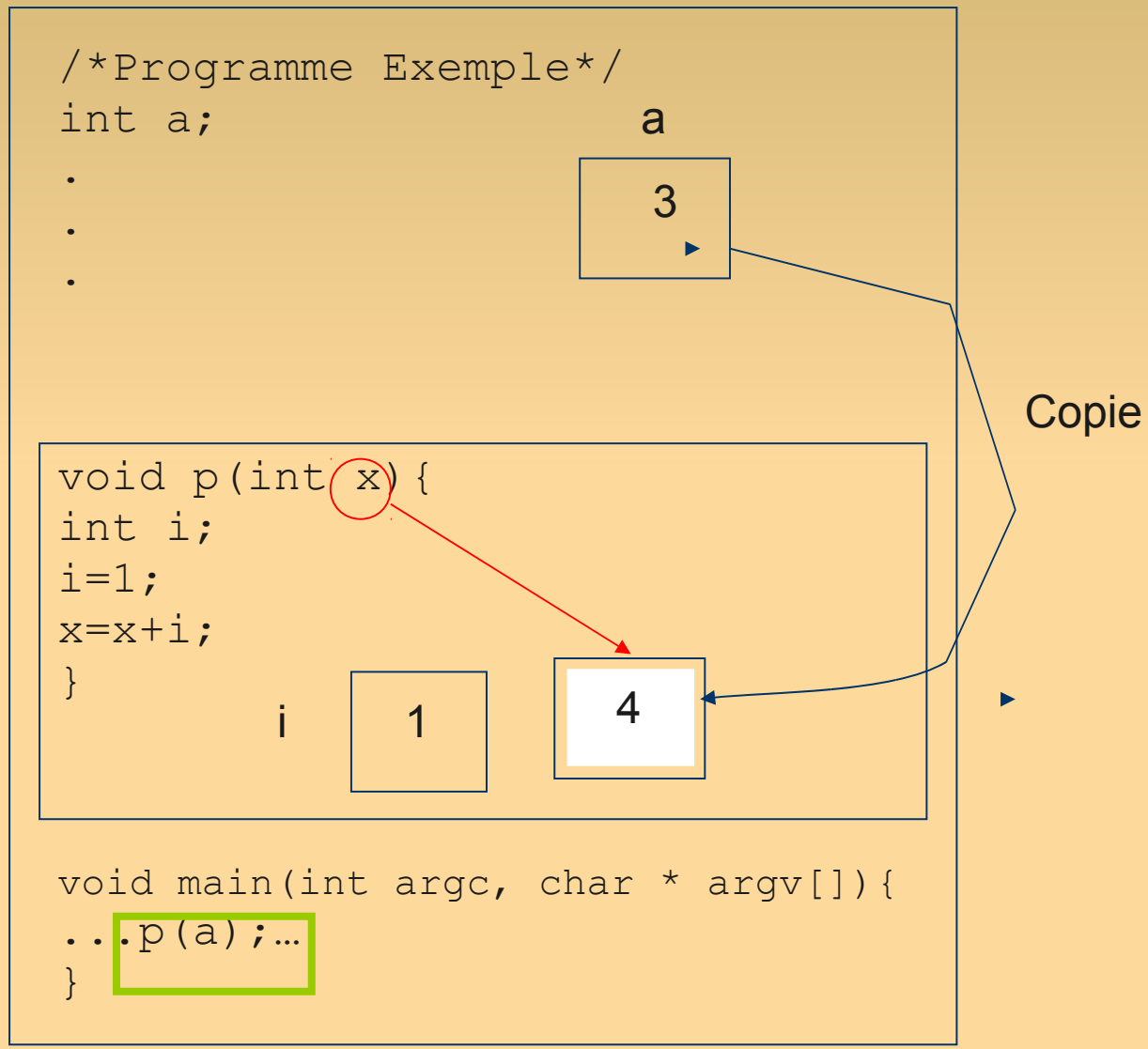
- **Passage par valeur**

  La valeur de l'expression passée en paramètre est copiée dans une variable locale.
  Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.

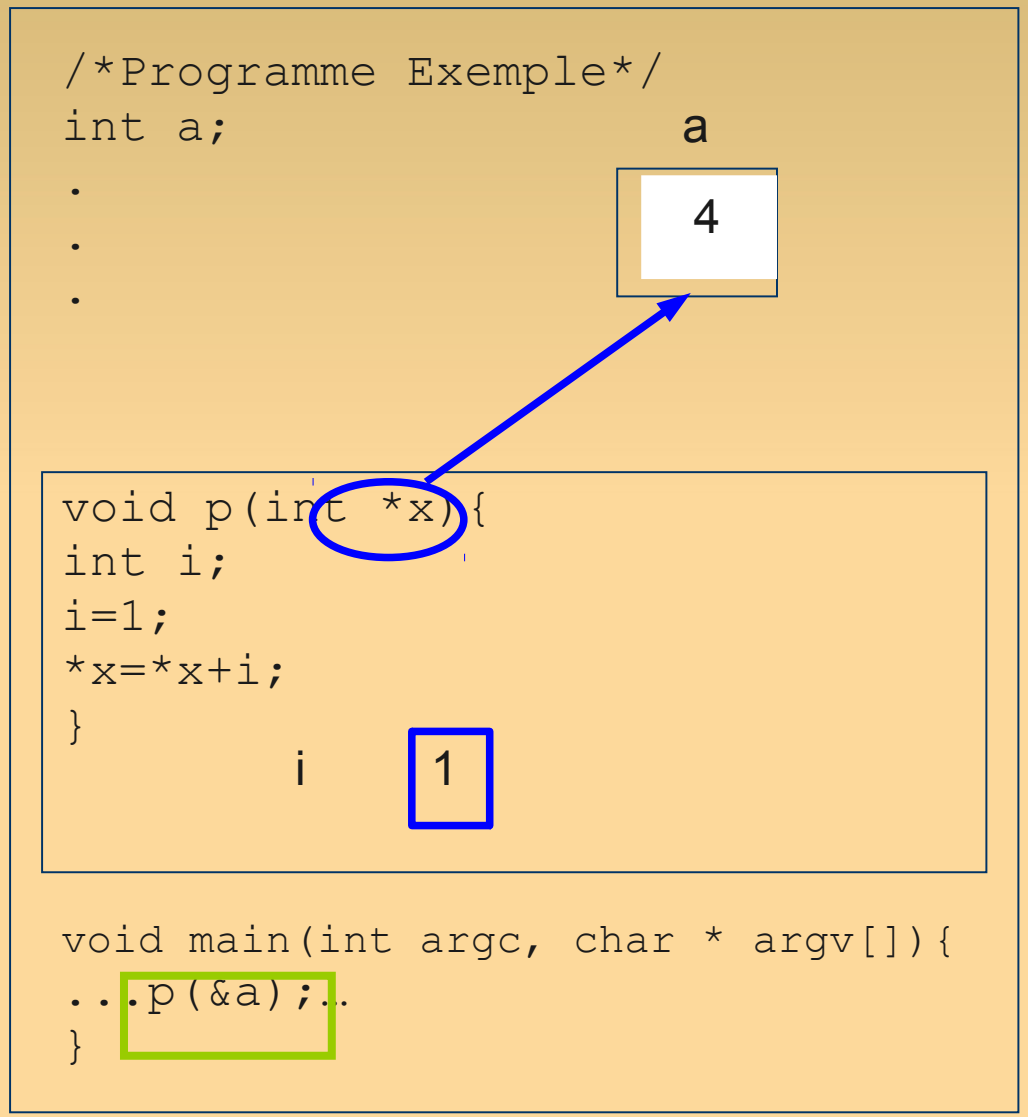
- **Passage par variable**

  Avec le passage par variable, il est possible de modifier la valeur d'un paramètre.
  Il faut passer l'adresse de la variable(&var). Cela constitue donc une application des pointeurs.(scanf)

Exemple d'exécution



Exemple d'exécution



Exercice

```
int x;
void t(int x)
{
      u(&x);
      x=2;
}
void u(int *y)
{
      x=1;
      *y=3;
}
```

```
main()
{
x=0;
t(x);
printf("%d",x);
}
```

1

Exercice

```
int x;
void t(int * z)
{
      u(*z);
      *z=2;
}
void u(int y)
{
      x=1;
      y=3;
}
```

```
main()
{
x=0;
t(&x);
printf("%d",x);
}
```

2

Exemple

```
int a;
void alias(int * x,int * y)
{
	*x=*x+*y;
	*y=*y+*x;
}
main()
{
	a=1;
	alias(&a,&a);
	printf("%d",a);
}
```

4

Récapitulatif

- Procédure : suite d'instructions décrivant une action simple qui devient une macro instruction
- Fonction : définition d 'un nouvel opérateur qui produit un résultat de type simple
- Paramètres formels - paramètres effectifs
- Statut des paramètres (données, données modifiées, résultats)
- Utilisation : P(a,3); x=x+f(y,z);

6) Récursivité

Une fonction f est définie récursivement lorsque sa définition utilise f elle-même

Définition

- La forme est toujours une définition par cas :
  - un cas général, dans lequel l'objet défini intervient
  - un cas particulier, dans lequel on a une valeur immédiate

- Il peut bien sûr y avoir plusieurs cas particuliers !

Exemple : Factorielle (1/2)

- 5! = 5 * 4 * 3 * 2 * 1 = 120
- 4! = 4 * 3 * 2 * 1 = 24
- 3! = 3 * 2 * 1 = 6
- 2! = 2 * 1 = 2
- 1! = 1

Pour n >= 0 :

fact(n) = n * fact(n-1);

fact(0) = 1;

Exemple : Factorielle (2/2)

```
int fact(int n)
{
      if (n==0)
            return 1;
      else
            return n*fact(n-1);
}

void main (int argc, char * argv[])
{
    printf("%d\n",fact(5));
}
```

Pour n >= 0 :

fact(n) = n * fact(n-1);

fact(0) = 1;

Exercice

Ecrire un programme récursif qui calcule la somme des n premiers entiers.

Programme 12

```
int som(int n)
{
      if (n==0)
            return 0;
      else
            return n+som(n-1);
}

void main (int argc, char * argv[])
{
    printf("%d\n",som(5));
}
```

Exercice

Ecrire un programme récursif qui calcule le PGCD de deux nombres entiers.

Programme 13

```
int pgcd(int a, int b)
{
    if (a==b)
        return a;
    else
        if (a>b)
            return pgcd(a-b,b);
        else
            return pgcd(a,b-a);
}
```

7) Algorithmes de tris

Il existe plusieurs manières de trier une suite d'éléments.

8 6 3 9 11 2 4 1 10 5 7

Tri par sélection

- On cherche l'élément le plus petit et on le place en tête.
- On réitère le processus sur les éléments restants jusqu'à ce tous les éléments soient bien placés.

8	6	3	9	11	2	4	1	10	5	7
1	6	3	9	11	2	4	8	10	5	7
1	2	3	9	11	6	4	8	10	5	7

...

Tri par insertion

- Pour chaque élément x de la liste, on cherche le premier élément y supérieur à x se trouvant avant x dans la liste.
- On insère x juste avant y ce qui décale le reste.

8	3	9	6	11	2	4	1	10	5	7
3	8	9	6	11	2	4	1	10	5	7
3	6	8	9	11	2	4	1	10	5	7

...

Tri à bulle

- On parcourt la liste en comparant deux à deux les éléments successifs et on les permute s'ils ne sont pas dans l'ordre
- On réitère l'opération jusqu'à ce la liste soit triée

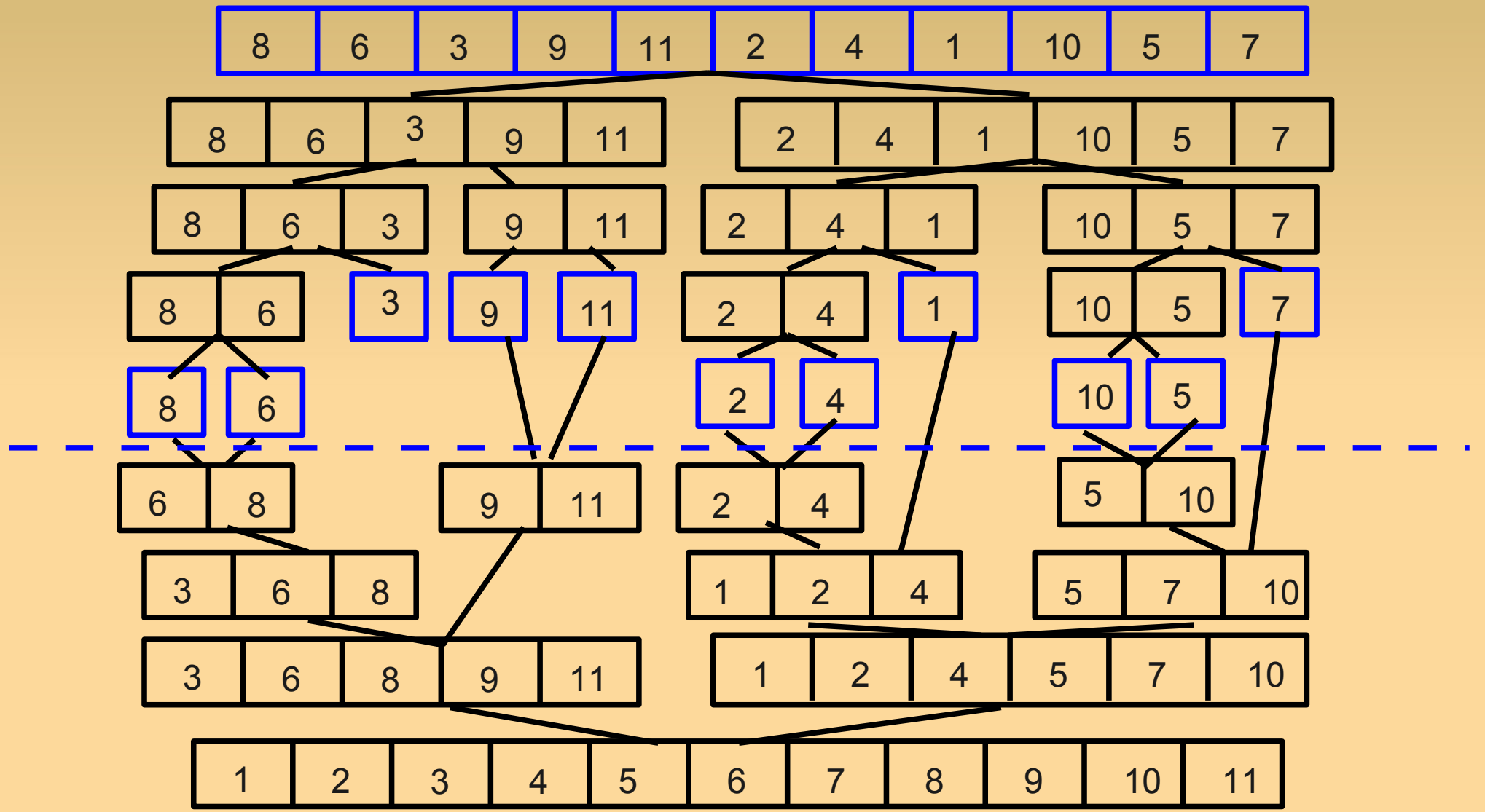
6 8 8 9 11 2 4 1 10 5 7

...

Tri fusion (1/2)

- On divise la liste en deux sous listes qui seront elles-mêmes triées avec le tri fusion
- Les deux listes triées sont ensuite fusionnées
- Remarque : une liste ne contenant qu'un seul élément est forcément triée.

Tri fusion (2/2)



Tri rapide (1/2)

- Aussi appelé Quicksort ou tri dichotomique
- Une valeur (pivot) est choisie dans la liste
- Toutes les valeurs inférieures au pivot sont placées avant et toutes les valeurs supérieures ou égales sont placées après
- L'opération est répétée sur chacune des sous listes placées avant et après le pivot

Tri rapide (2/2)

8	6	3	9	11	2	4	1	10	5	7
6	3	2	4	1	5	7	8	9	10	11
3	2	4	1	5	6	7	8	9	10	11
2	1	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11

Le choix tu pivot est très important pour la performance de ce tri !

8) Structures

Il est parfois nécessaire de définir des types de données particuliers

Définition

- Une structure permet de grouper un certain nombre de variables d'un type différent
- Cela correspond à la notion d'enregis-trement d'autres langages
- Il existe trois manières de déclarer et d'utiliser une structure

Sans étiquette de structure

```
struct
 {
    char nom[20] ;
    char prenom[20] ;
    int age ;
  } p1, p2 ;
```

Sans étiquette de structure

```
struct
  {
       char nom[20] ;
       char prenom[20] ;
       int age ;
   } p1, p2 ;
```

- Aucun nom n'est donné à la structure
- Impossible de déclarer une autre variable de même type
- Deux structures ayant les mêmes champs ne sont pas égales !

Avec étiquette et déclaration

```
struct personne
  {
     char nom[20] ;
     char prenom[20] ;
     int age ;
   }pers1, pers2;
  struct personne p1, p2 ;
```

Avec étiquette de structure

```
struct personne
 {
    char nom[20] ;
    char prenom[20] ;
    int age ;
 };
 struct personne p1, p2 ;
```

Avec étiquette de structure

```
struct personne
  {
      char nom[20] ;
      char prenom[20] ;
      int age ;
   };
  struct personne p1, p2 ;
```

- Permet de séparer déclaration du type et déclaration de la variable

Utilisation

- L'accès aux différents membre d'une structure se fait avec l'opérateur .
  - Exemple : p1.nom
- L'initialisation peut être directe :
  - struct personne p = {"Paul", "Dupont", 25} ;
- Il est possible d'affecter une structure directement avec l'opérateur d'affectation :
  - Exemple

```
struct personne p1 = {"Paul", "Dupont", 25}, p2 ;
...
p2 = p1 ;
```

9) Pointeurs

Vers une meilleure gestion de la mémoire...

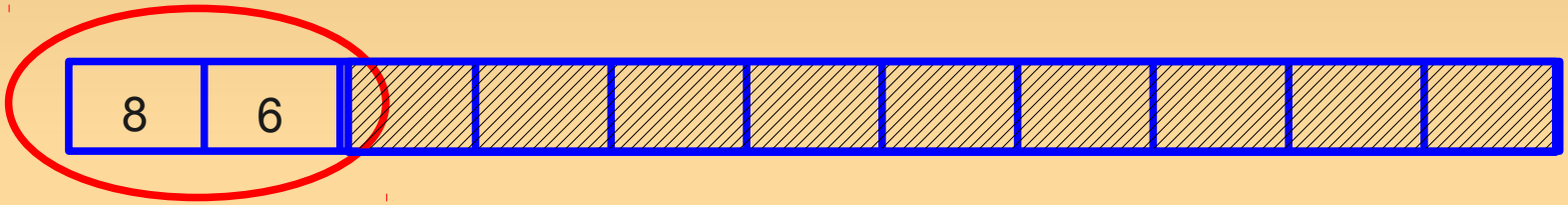
Variables statiques

- Toujours déclarées en tête du programme ou du bloc dans lequel elles sont utilisées
- Elles occupent une place mémoire qui leur est allouée pendant toute l'exécution du programme ou du bloc
- Leur adressage est direct : elles sont accessibles directement par leur identifiant

Variables statiques

int tableau[11];

11 cases de réservées

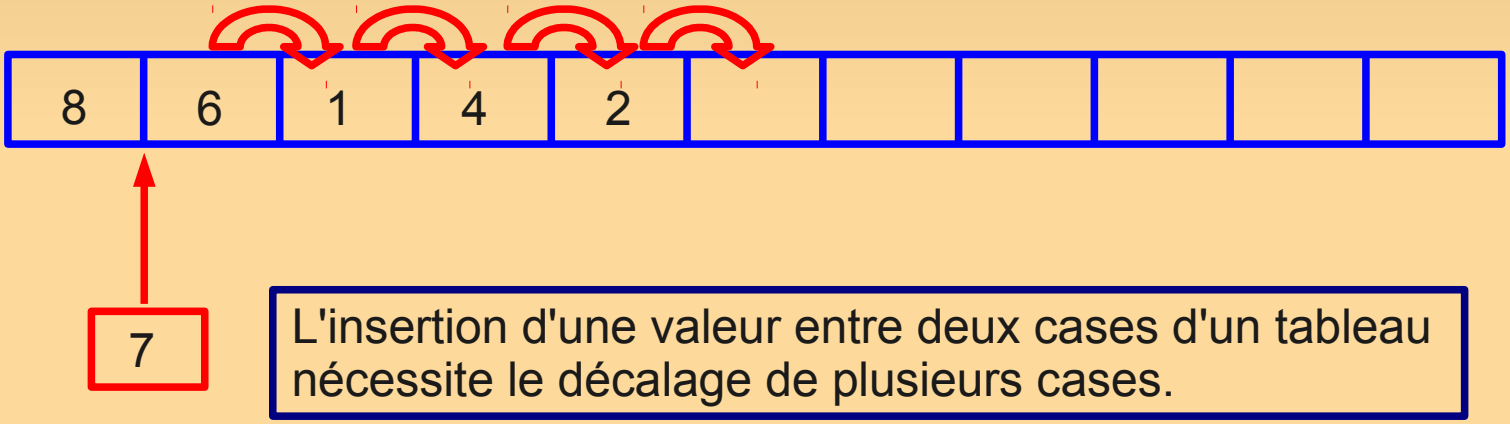


2 cases d'utilisées

9 cases réservées mais non utilisées

Variables statiques

int tableau[11];



Variables statiques

```
int tableau[11];
```

Améliorer l'utilisation de l'espace mémoire

- Il est souhaitable de pouvoir créer et utiliser des variables, uniquement au fur et à mesure des besoins
- On veut pouvoir récupérer l'espace mémoire devenu inutile
- Il serait bon de pouvoir supprimer et insérer des éléments sans toucher au reste des données

Variable dynamique

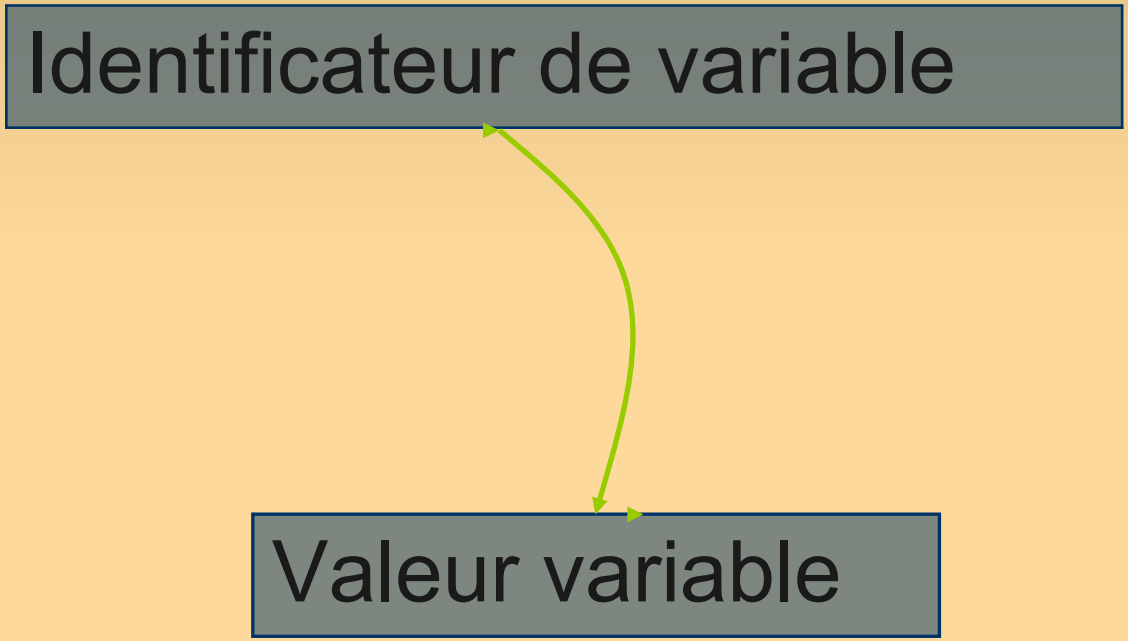
Propriétés :

- Elle peut être générée et détruite pendant l'exécution du bloc, ce qui permet de libérer la place mémoire
- Elle ne peut être adressée directement par son identificateur

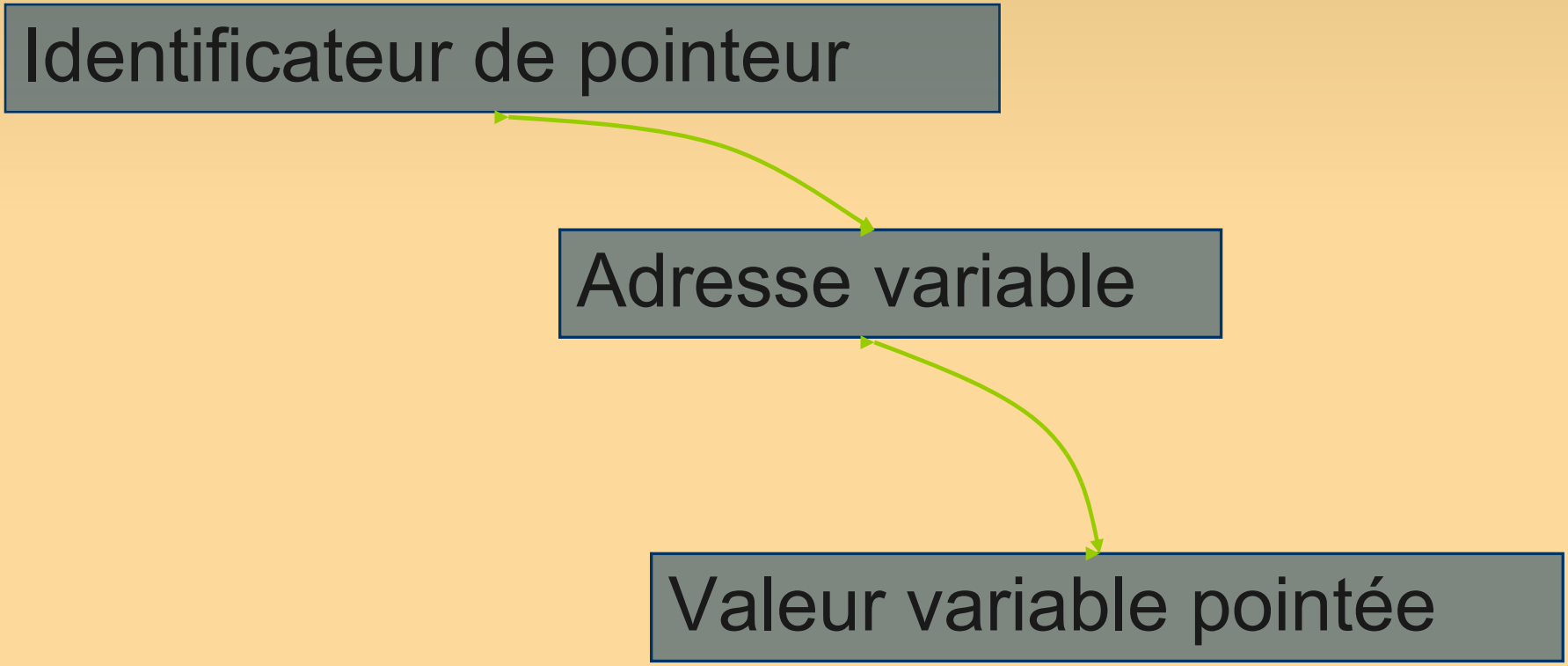
Variable dynamique

- Elle n'est accessible que par l'intermédiaire d'une variable spéciale appelée variable pointeur (ou pointeur), qui est une variable statique et qui contient l'adresse de cette variable dynamique
- On dit que le pointeur pointe vers cette variable ou que cette variable est pointée.

Adressage : variable statique



Adressage : variable dynamique



Modes d'adressage

- L'adressage indirect permet une gestion dynamique de la mémoire (supression, insertion, tri des données …)
- NB : Dans le cas d'un processus récursif, le programme utilise une variable dynamique

Définition d'un pointeur

- int * P;
- P est un pointeur vers un entier.

- typedef int * Point;

  Point P;
- Le type Point est défini ici comme un pointeur qui pointe vers un objet de type int.

Création de la variable pointée

- P est une variable pointeur qui n'a de sens que lorsqu'on aura crée la variable pointée.

```
type_de_P * P;
P=(type_de_P *)malloc(sizeof(type_de_P));
```

- **Remarque**: malloc est une fonction qui retourne un pointeur d'un type spécial. Il faut donc forcer le pointeur à prendre le type demandé (type_de_P *)

Création de la variable pointée

Effets de la fonction malloc :

- Elle réserve une case mémoire pour la variable pointée qui sera désignée par

```
*P
```

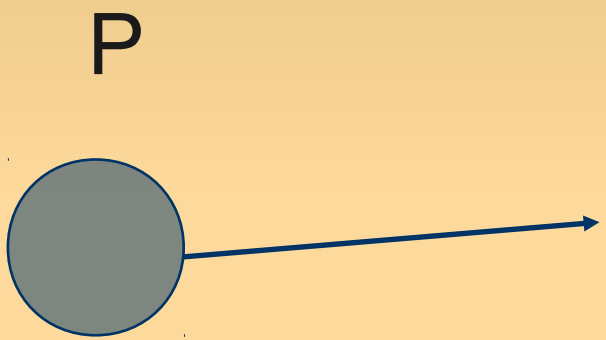
- Elle donne au pointeur P la valeur de l'adresse de la case réservée pour `P`
- Attention, cette adresse n'a pas à être connue de l'utilisateur, elle est gérée par le système

Schématisation de la création

- Le pointeur P contient l'adresse de *P mais *P contient une valeur indéterminé pour le moment

Affectations et effets

- On peut envisager, une fois P et *P misent en place, deux types d'affectation pour les pointeurs :
  - P=NULL



NULL est la constante *pointeur vide* qui signifie que ce pointeur n'est plus indéterminé, il contient l'équivalent d'une adresse nulle

Affectations et effets

- On peut envisager, une fois P et *P misent en place, deux types d'affectation pour les pointeurs :
  - P=NULL
  - P=Q

Affectations et effets

P=Q

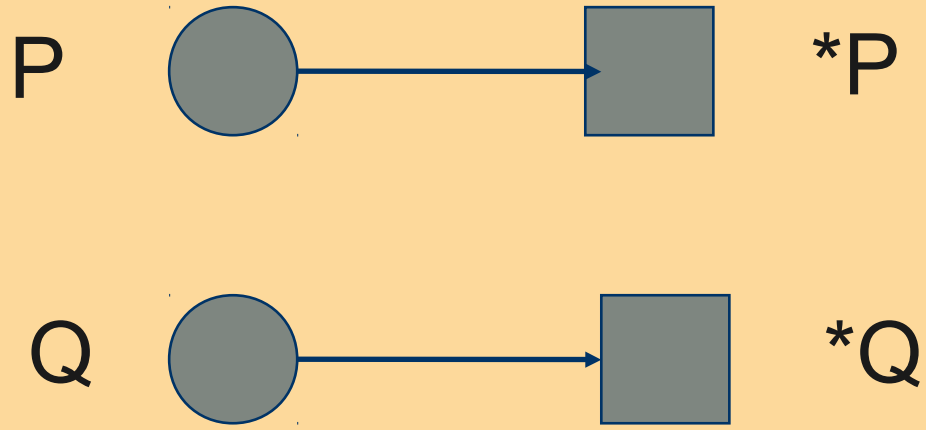
- Une telle affectation suppose que la variable Q a été déclarée auparavant comme une variable pointeur qui pointe vers un élément de même type que celui pointé par P.

```
un_type *P,*Q;
```

Affectations et effets

P=Q

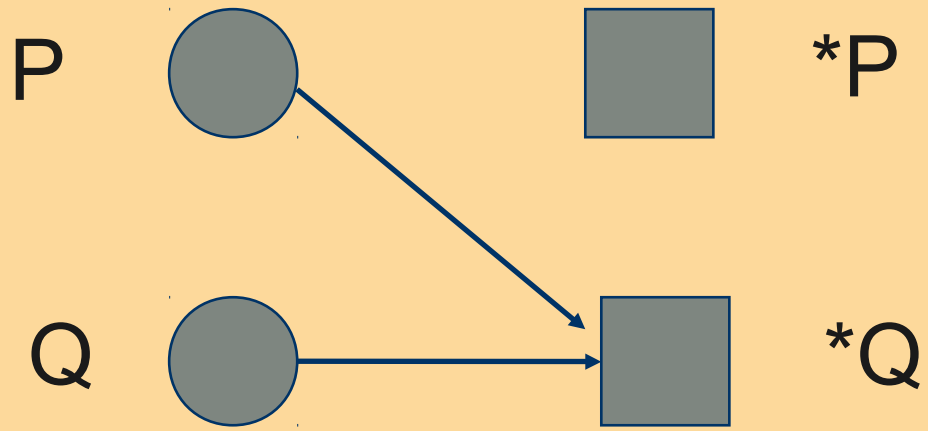
Avant *(après initialisation supposée des pointeurs)*



Affectations et effets

P=Q

Après



Affectations et effets

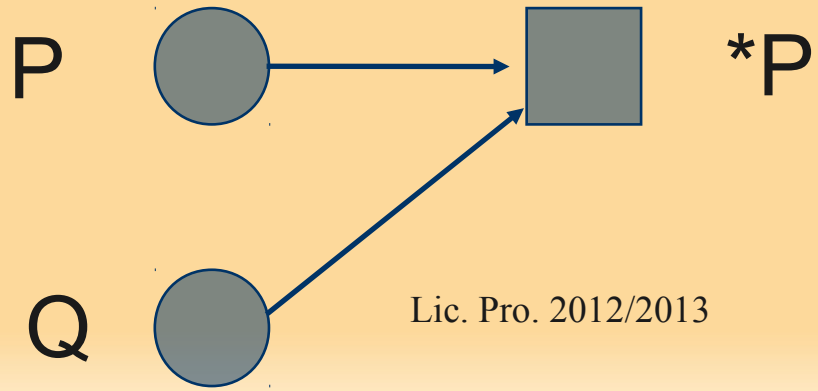
P=Q

- L'affectation P=Q fait que les deux pointeurs P et Q contiennent la même adresse, celle de Q
- Elle signifie que désormais P pointe vers la même variable que Q, c'est à dire *Q
- Après P=Q, l'ancienne valeur de *P n'est plus accessible par la variable pointeur P

Affectations et effets

P=Q

- Dans le cas présent, P=Q suppose que Q contient une adresse déterminée résultant d'un traitement antérieur, ou d'une création de la variable pointée *Q par malloc
- Par contre, l'affectation inverse Q=P n'implique pas de création de *Q puisque désormais Q pointe vers *P



Affectation sur les variables pointées

*P=15;

- La variable *P contient désormais la valeur de la chaîne 15

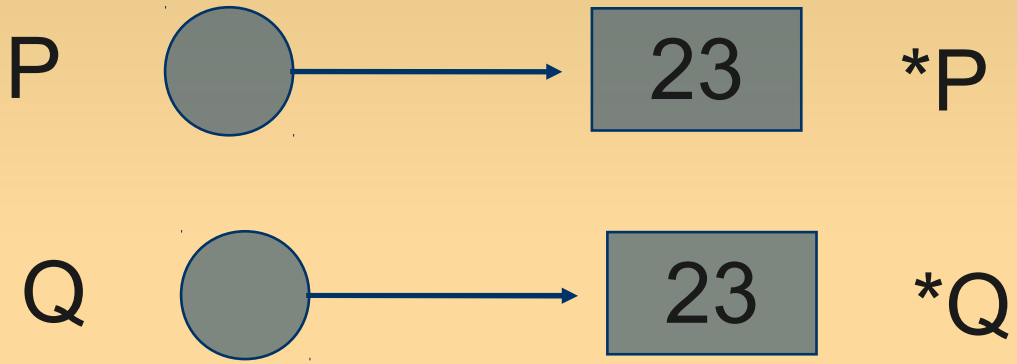
Affectation sur les variables pointées

```
scanf("%d",Q);
```

- Si l'on rentre 23 au clavier, cela aura pour effet d'affecter la valeur 23 à la variable pointée *Q

Affectation sur les variables pointées

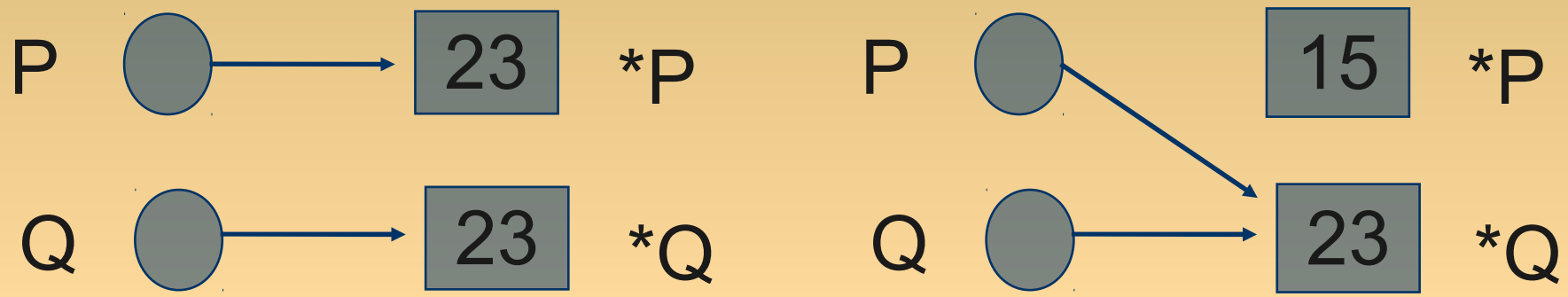
*P=*Q;



- Cette affectation a pour effet que *P contient la même valeur que la variable *Q. Les pointeurs associés contiennent eux des valeurs différentes

Affectation sur les variables pointées

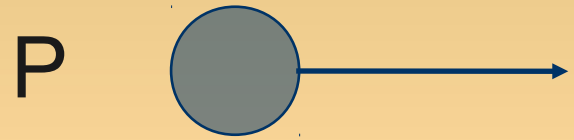
Attention



- On voit qu'il ne faut pas confondre *P=*Q et P=Q

Récupération de la place mémoire

```
free(P);
```



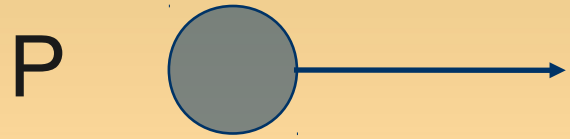
- A pour effet de libérer la place préalablement occupée par *P. La case P reste réservée mais l'adresse qui y est stockée n'a plus de signification

Exemple

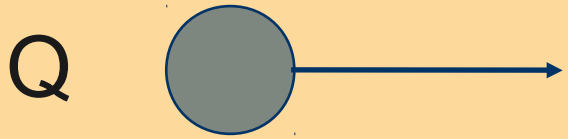
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct type_fiche{
   char nom[20];
   int note;
};

typedef struct type_fiche * fiche;

fiche P,Q,R;
```

Exemple

```
main(){
P=(fiche) malloc(sizeof(struct type_fiche));
```

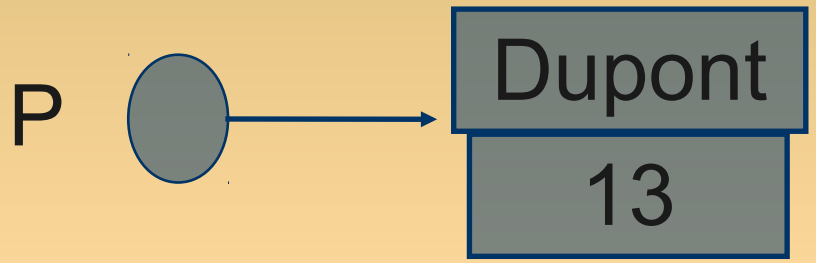



```
Q=(fiche) malloc(sizeof(struct type_fiche));
```

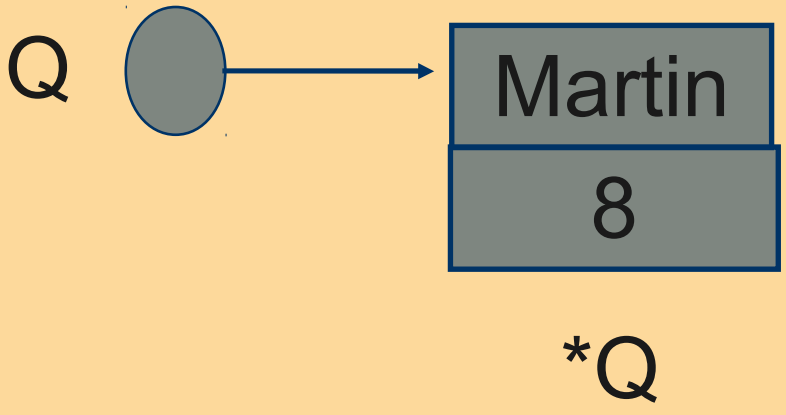


Exemple

```
strcpy(P->nom,"Dupont");
P->note=13;
```

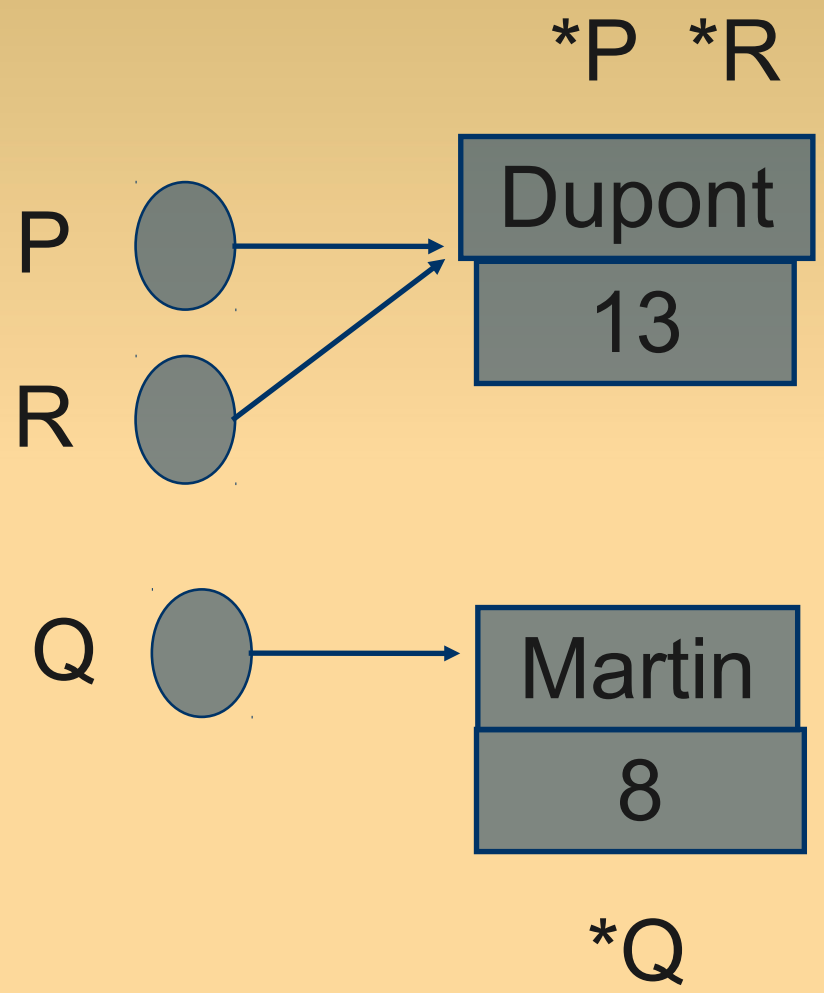



```
strcpy(Q->nom,"Martin");
Q->note=8;
```



Exemple

```
R=P;
```

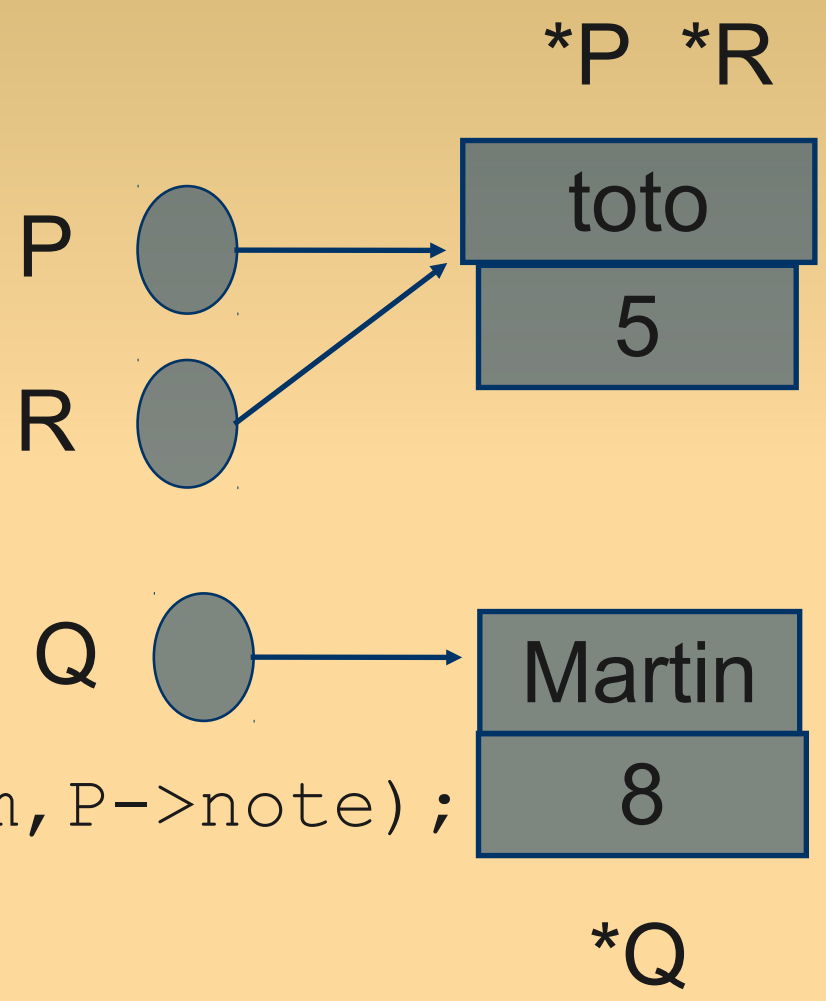


Exemple

```
R=P;
```

```
strcpy(R->nom,"toto");
R->note=5;
```

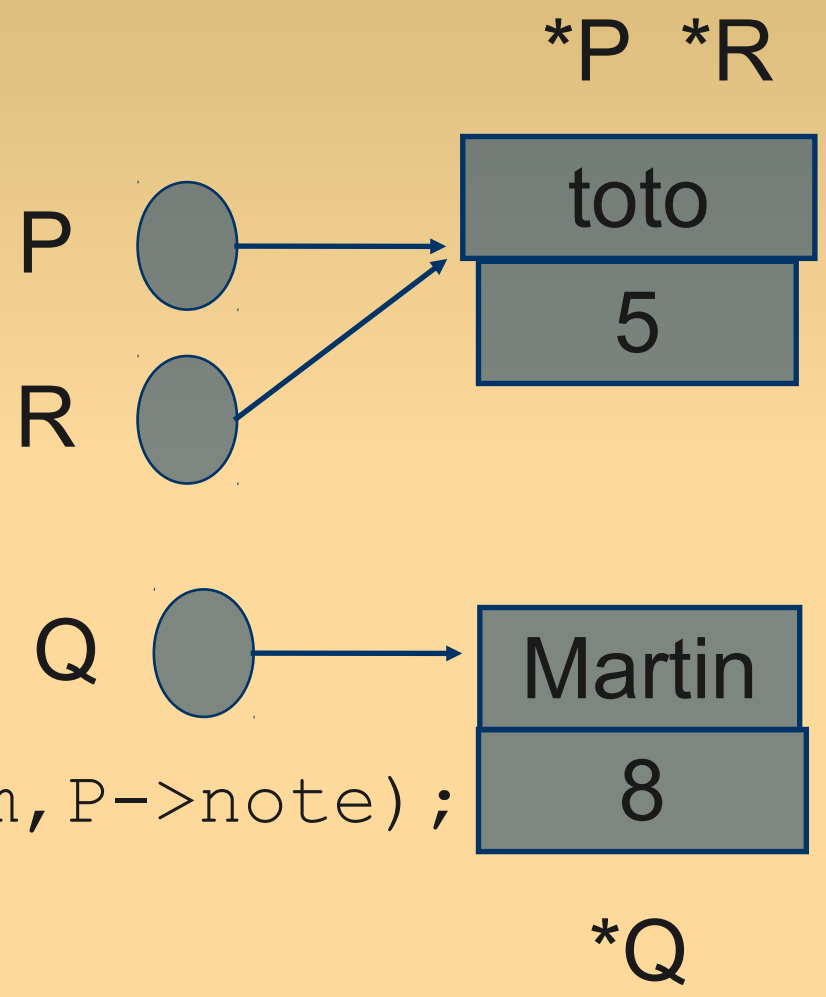
```
printf("%s %d\n",P->nom,P->note);
```



Exemple

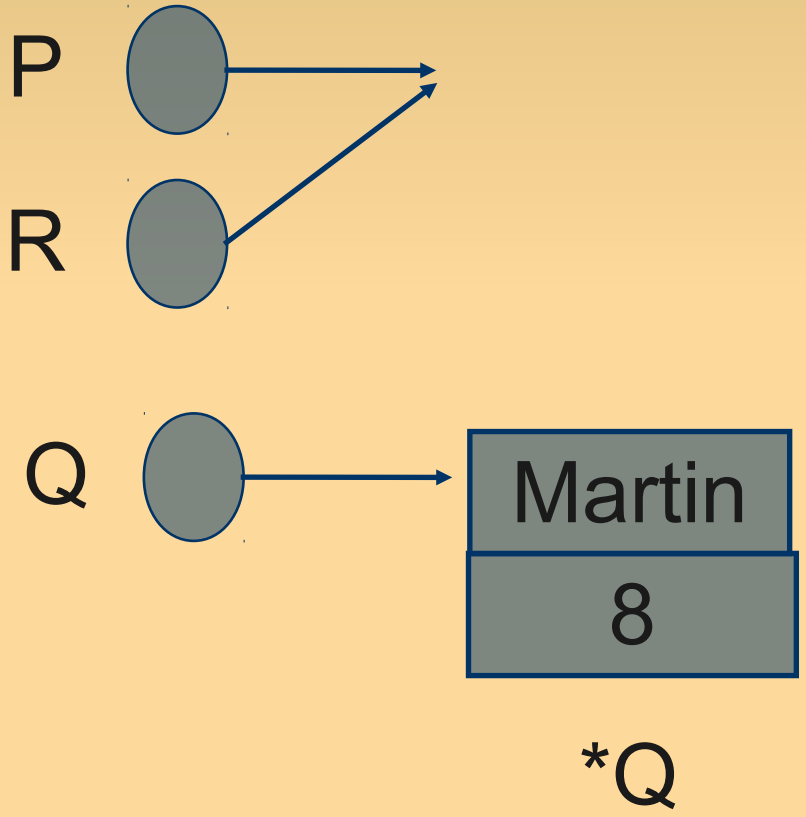
```
R=P;

strcpy(R->nom,"toto");
R->note=5;

printf("%s %d\n",P->nom,P->note);
```

toto 5



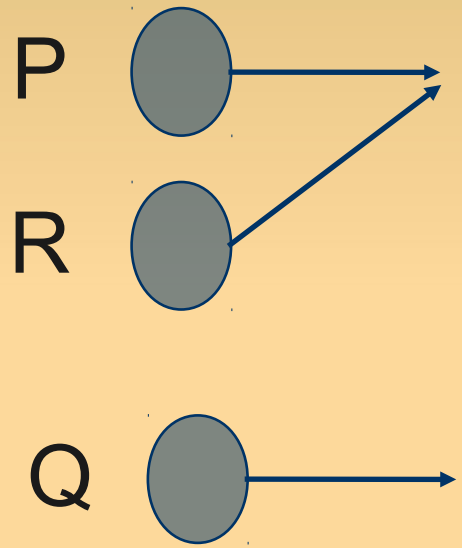
Exemple

```
free(R);
```



Exemple

```
free(R);
free(Q);
}
```



10) Listes / Piles /Files

Une bonne exploitation des pointeurs

Notion de liste chaînée

- Application importante des pointeurs
- Une liste chaînée est constituée par une suite d'éléments
- Chaque élément possède un successeur (sauf le dernier)
- Contrairement aux éléments d'un tableau, les éléments d'une liste chaînée ne sont pas indicés

Notion de liste chaînée

- Pour que l'ordre des éléments soit respecté, il faut qu'un lien existe entre un élément et son successeur dans la structure
- Ce lien est fourni par un pointeur
- Chaque élément est donc constitué de deux parties : sa valeur proprement dite et un pointeur contenant l'adresse de l'élément successeur

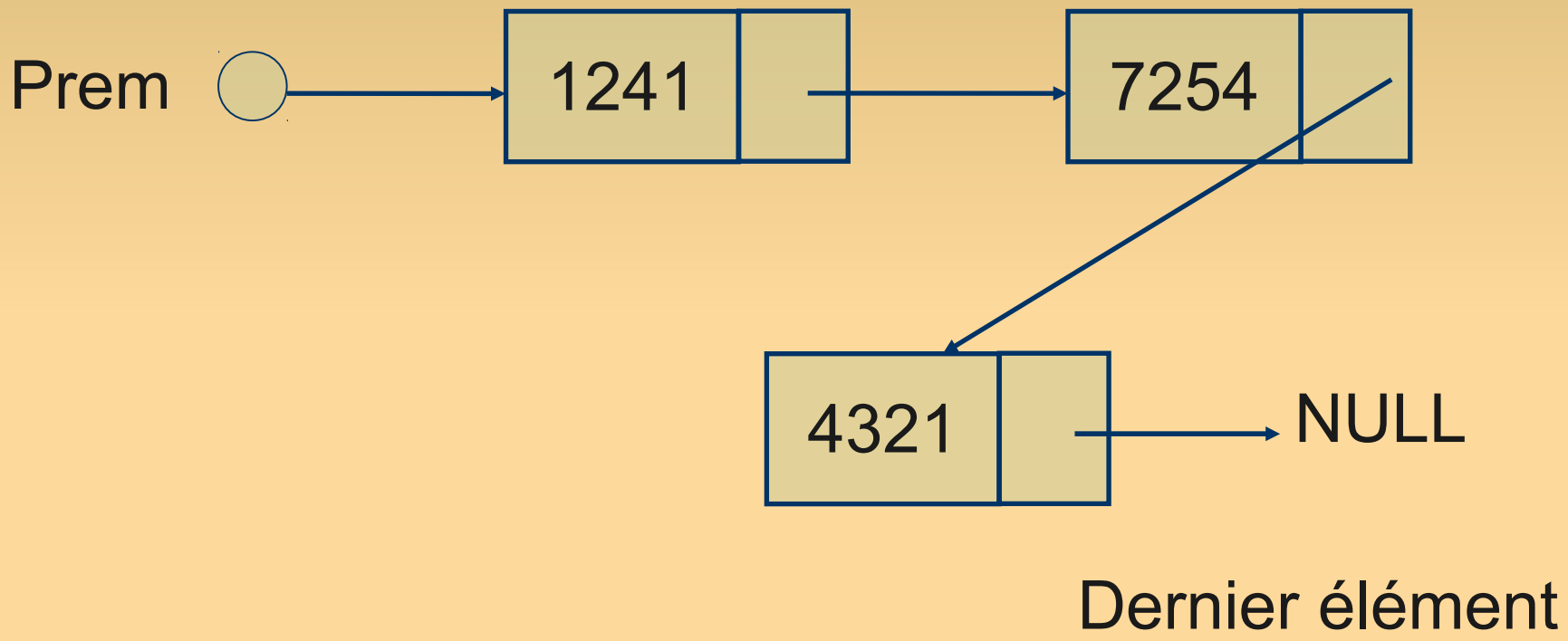
Exemple : déclaration d'une liste chaînée de numéro

```
#include <stdlib.h>
struct Element
{
    int Numero;
    struct Element * Suivant;
};
typedef struct Element * Liste;
int main()
{
 Liste l;
 l = (Liste)malloc(sizeof(struct Element));
 l->Numero = 24100;
 l->suivant = NULL;

...
 free(l);
 return 0;
}
```

Accès aux éléments

- Pour accéder à un élément, il faut parcourir la chaîne jusqu'à cet élément
- Pour accéder directement aux éléments il faudrait utiliser autant de pointeurs externes que d'éléments
- On privilégie ici l'espace mémoire utilisé par rapport au facteur temps
- Le dernier élément contient généralement la valeur NULL
- Pour se référer à la liste il suffit d'un seul pointeur externe Prem

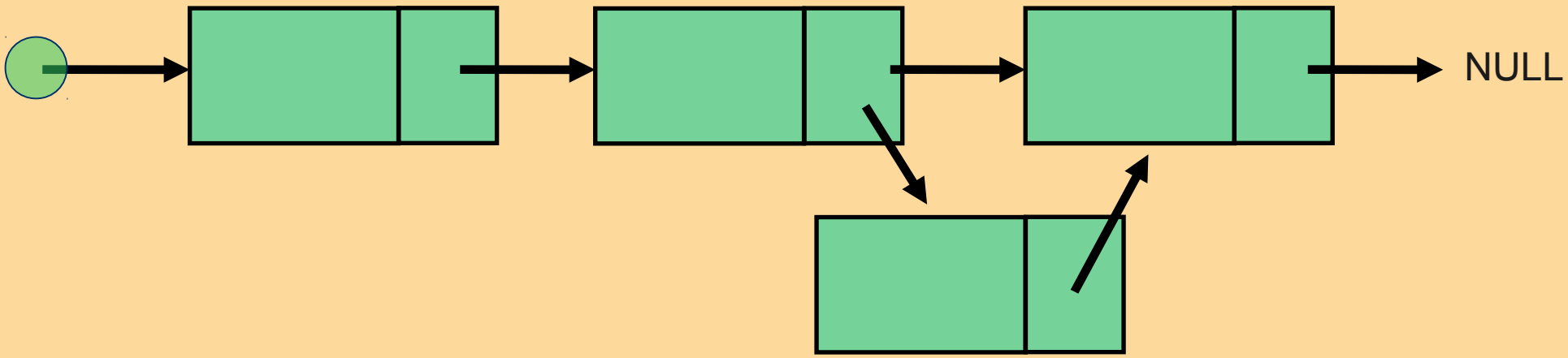
Exemple



Intérêts de la liste chaînée

- Souplesse de manipulation des éléments (insertion, suppression …)

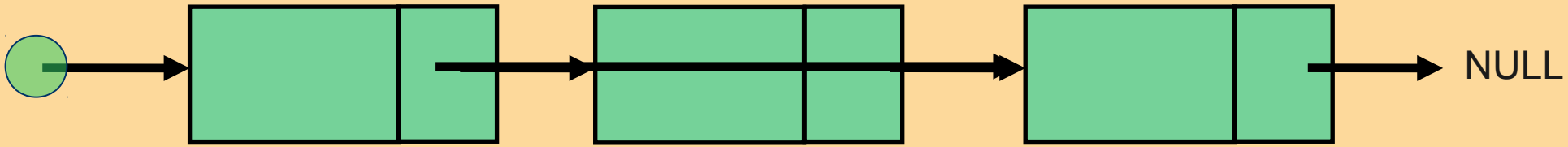
Insertion



Intérêts de la liste chaînée

- Souplesse de manipulation des éléments (insertion, suppression …)

Suppression

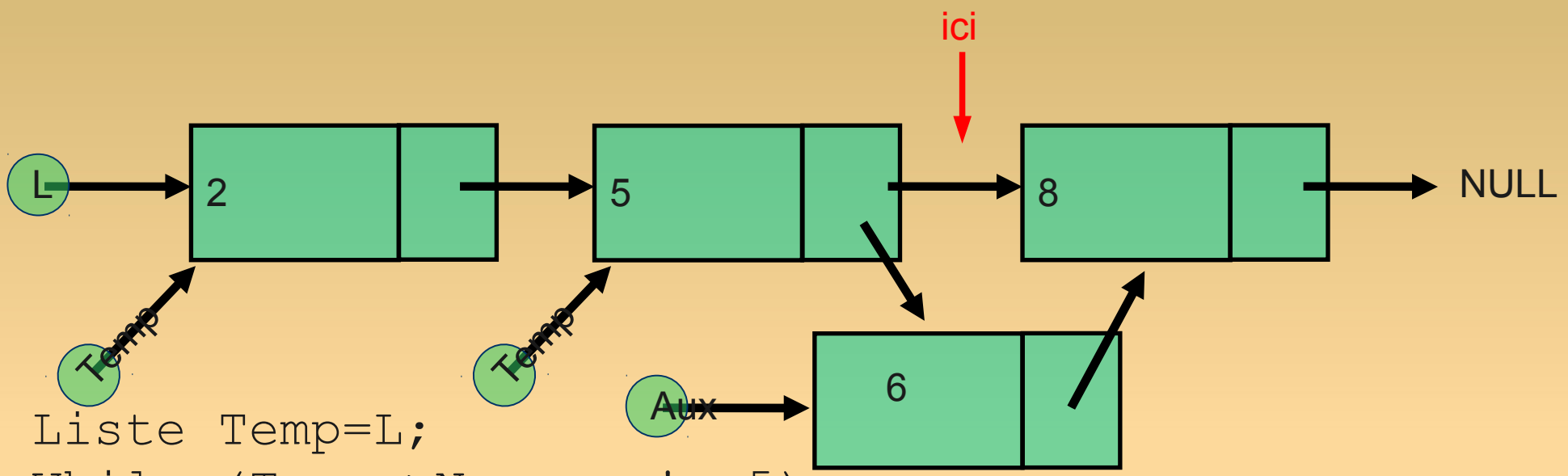


Manipulation

Attention !

- Dans les opérations de réorientation des pointeurs internes, il faut faire attention à l'ordre de ces opérations de manière à ne pas perdre une adresse avant d'avoir effectué le branchement voulu

Insertion




```
Liste Temp=L;
While (Temp->Numero != 5)
   Temp = Temp->suivant;

Liste Aux;
Aux=(Liste)malloc(sizeof(struct Element));
Aux->Numero=6;
Aux->suivant=Temp->suivant;
Temp->suivant=Aux;
```

Exercice

- Ecrire une fonction qui vérifie qu'un numéro appartient à une liste donnée (retourner 1 si l'élément appartient et 0 sinon).

Programme 14

```
int rechercher(Liste l, int val)
{
	Liste tmp = l;
	while ((tmp != NULL) && (tmp->Numero != val))
		tmp=tmp->suivant;
	if (tmp == NULL)
		return 0;
	else
		return 1;
}
```

Exercice

- Ecrire une fonction qui permet de supprimer un élément d'une liste en fonction de son numéro.

Programme 15

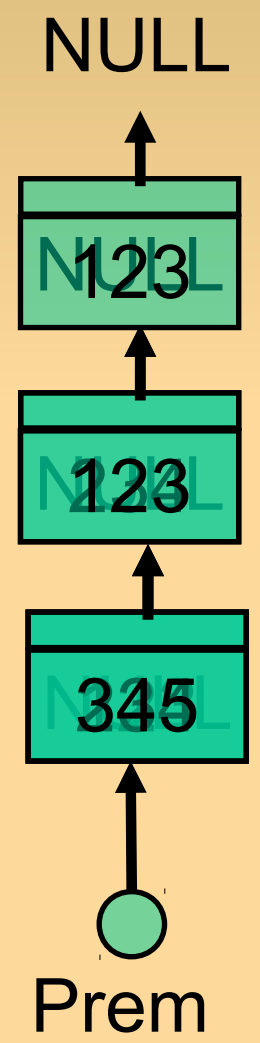
```
void supprimer(Liste *l, int val)
{
	Liste sup,tmp = *l;
	if ((*l)->Numero == val)
		{
			sup = *l;
			*l = (*l)->suivant;
		}
	else
		{
			while ((tmp->suivant != NULL) && (tmp->suivant->Numero != val))
				tmp=tmp->suivant;
			if (tmp->suivant->Numero == val)
				{
					sup = tmp->suivant;
					tmp->suivant=tmp->suivant->suivant;
				}
		}
	free(sup);
}
```

Listes chaînées particulières

- Dépend de l'ordre dans lequel on désire pouvoir la lire
- Deux structures de données possibles
  - La file
  - La pile

Utilisation d'une file

- Point de départ NULL
- On ajoute l'élément 123
- On ajoute l'élément 234
- On ajoute l'élément 345
- On défile (l'élément 123)
- ...



FIFO (First In First Out)

Utilisation d’une file

- Pour mémoriser temporairement des transactions qui doivent attendre pour être traitées.
- Les serveurs d'impression qui traitent les requêtes dans l'ordre dans lequel elles arrivent et qui les insèrent dans une file d'attente (ou une queue).

Manipulation d'une file

- La file est-elle vide?:

```
int estvide(Liste maliste)
{
    if (maliste == NULL)
        return 1;
    else
        return 0;
}
```

Manipulation d'une file

- Enfiler un élément :

```
void enfiler(Liste *maliste, Liste elt)
{
    elt->suivant = *maliste;
    *maliste=elt;
}
```

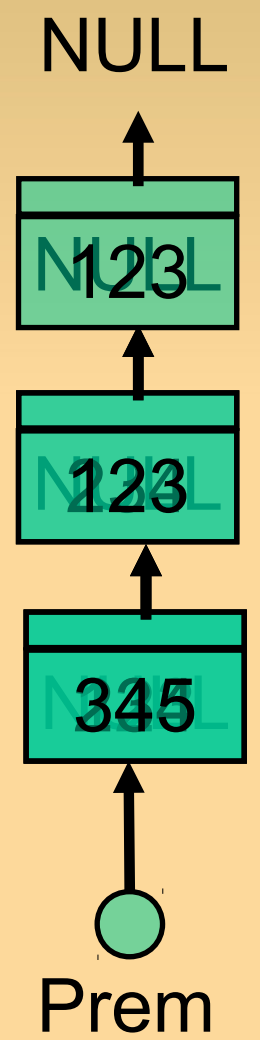
Manipulation d'une file

- Défiler un élément :

```
Liste defiler(Liste *maliste){
        Liste tmp,tmp2;
        if (*maliste == NULL)
                return NULL;
        else {
                        tmp = *maliste;
                        if ((*maliste)->suivant == NULL){
                                        (*maliste) = NULL;
                                        return tmp;}
                        else {
                                        while(tmp->suivant->suivant != NULL)
                                                tmp=tmp->suivant;
                                        tmp2 = tmp->suivant;
                                        tmp->suivant = NULL;
                                        return tmp2;}
                }
}
```

Utilisation d'une pile

- Point de départ NULL
- On ajoute l'élément 123
- On ajoute l'élément 234
- On ajoute l'élément 345
- On dépile (l'élément 345)
- ...



LIFO (Last In First Out)

Utilisation d’une pile

- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Précédent ».
- La fonction « Annuler la frappe » (en anglais Undo) d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

Manipulation d'une pile

- La pile est-elle vide?:

```
int estvide(Liste maliste)
{
    if (maliste == NULL)
        return 1;
    else
        return 0;
}
```

Manipulation d'une pile

- Empiler un élément :

```
void empiler(Liste *maliste, Liste *elt)
{
    (*elt)->suivant = *maliste;
    *maliste=*elt;
}
```

Manipulation d'une pile

- Dépiler un élément :

```
Liste depiler(Liste *maliste)
{
    Liste tmp;
    if (*maliste == NULL)
        return NULL;
    else
        {
            tmp = *maliste;
            *maliste=(*maliste)->suivant;
            tmp->suivant = NULL;
            return tmp;
        }
}
```

11) Arbres

Vers une meilleure organisation des données

Arbres et arbres binaires

- Un **arbre** est un ensemble de nœuds organisés de façon hiérarchique à partir d'un nœud distingué appelé racine.
- Un **arbre binaire** est un arbre particulier qui intervient dans de nombreuses représentations et de nombreuses applications informatiques.

Arbre binaire

- Un arbre binaire est soit vide (noté Λ) soit de la forme B=<o,B1,B2>, où B1 et B2 sont des arbres binaires disjoints et o est un nœud appelé racine.
- Un arbre binaire dont les nœuds contiennent des éléments est appelé **arbre binaire étiqueté**. Si <o,a1,a2> est un arbre binaire étiqueté la racine contient l'élément r, on notera parfois de manière abusive <r,a1,a2>.

Vocabulaire

- b est un **sous-arbre** de a=<o,a1,a2> ssi b=a, ou b=a1, ou b=a2, ou b est un sous-arbre de a1 ou de a2.
- **Fils gauche** (resp. droit) **d'un nœud**=racine du sous-arbre gauche (resp. droit).
- Père, frère, ascendant ou ancêtre, descendant.

Vocabulaire

- Tous les nœuds d'un arbre binaire ont au plus deux fils.
- nœud double = nœud avec deux fils
- nœud simple à gauche (resp. à droite) = nœud avec seulement un fils gauche (resp. droit).
- feuille = nœud sans fils

Mesures sur les arbres

- La taille d'un arbre est le nombre de ses nœuds.
- La hauteur d'un nœud (ou profondeur de niveau) est la longueur du chemin entre la racine et ce nœud.
  - h(x) = 0 si x est la raine
  - h(x) = 1 + h(y) si y est le père de x
- La hauteur ou profondeur d'un arbre est la hauteur maximum de ses feuilles
- La longueur de chemin d'un arbre est la somme des hauteurs de ses nœuds.

Arbre particulier

- Un arbre binaire est dit complet si chacun de ses niveaux est complètement rempli, c'est à dire 2 nœuds au niveau 1, 4 nœuds au niveau 2, ..., 2^h au niveau h.
  - taille(B) = $1 + 2 + \ldots + 2^h = 2^{h+1} - 1$
    si h est la hauteur de B

Codage d'un arbre

```
struct noeud
{
        int val;
        struct noeud * gauche;
        struct noeud * droit;
};
typedef struct noeud * arbre;
```

Exercice

- Ecrire une fonction qui calcule la taille d'un arbre donné.

Programme 16

```
int taille(arbre a)
{
    if (a == NULL)
        return 0;
    else
        return 1 + taille(a->gauche) + taille(a->droit);
}
```

Exercice

- Ecrire une fonction qui calcule la hauteur d'un élément donné dans un arbre.

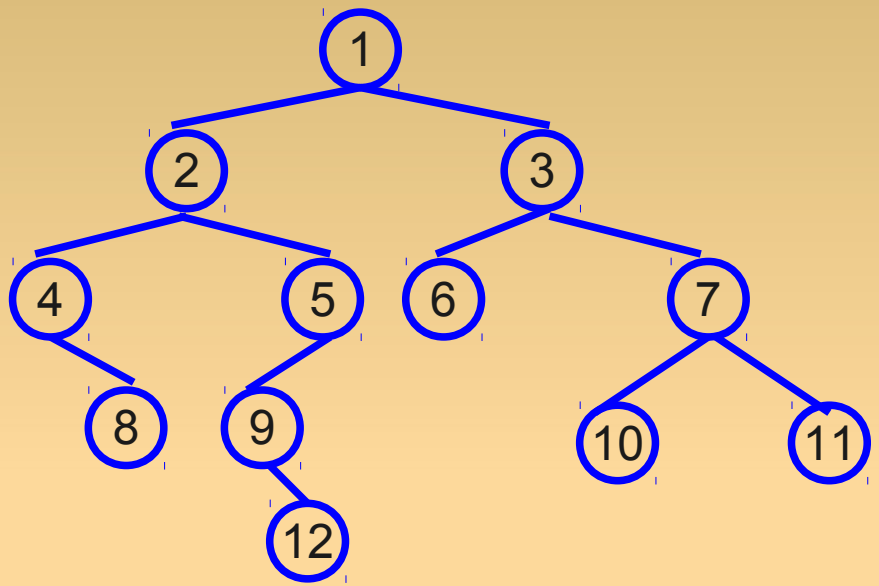
Programme 17

```
int hauteur(arbre a,int n)
    {
    int h;
    if (a == NULL)
         return -1;
    if (a->val == n)
         return 0;
    h = hauteur(a->gauche,n)+1;
    if (h == 0)
         {
         h = hauteur(a->droit,n)+1;
         if (h == 0)
              return -1;
         else
              return h;
         }
    else
         return h;
    }
```

Parcours d'un arbre

- Il existe plusieurs façons de parcourir un arbre :
  - largeur d'abord
  - profondeur d'abord
    - parcours préfixe
    - parcours infixe
    - parcours suffixe

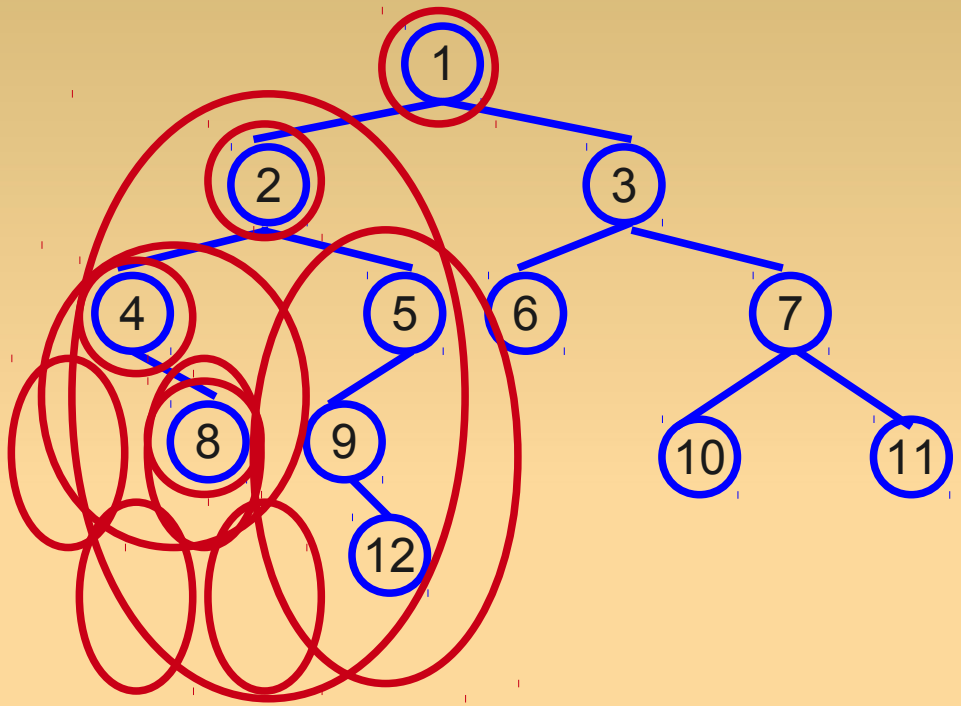
Parcours en largeur d'abord



- Le parcours en largeur consiste à parcourir l'arbre niveau par niveau.

1 2 3 4 5 6 7 8 9 10 11 12

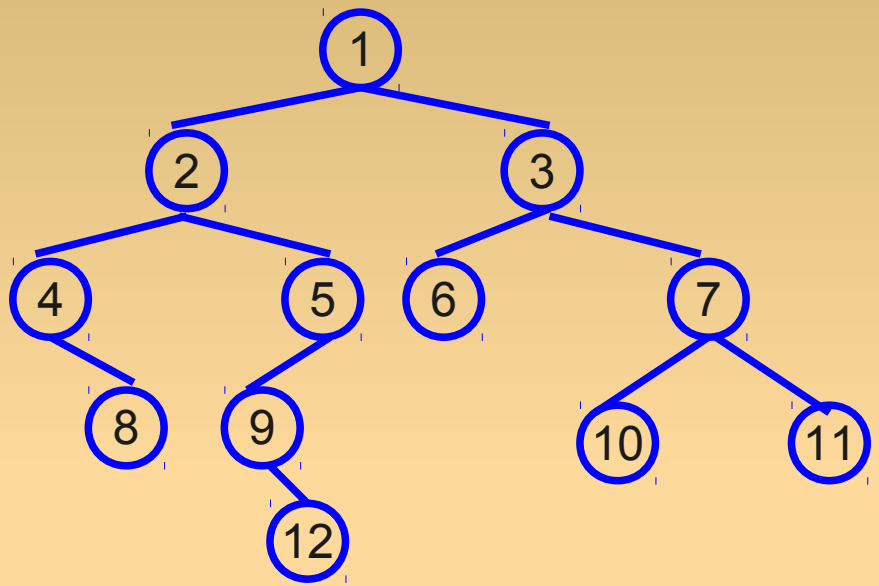
Parcours en profondeur d'abord : préfixe



- Le parcours préfixe consiste à afficher la racine puis à parcourir de manière préfixée le sous-arbre gauche suivi du sous-arbre droit.

1 2 4 8 5 9 12 3 6 7 10 11

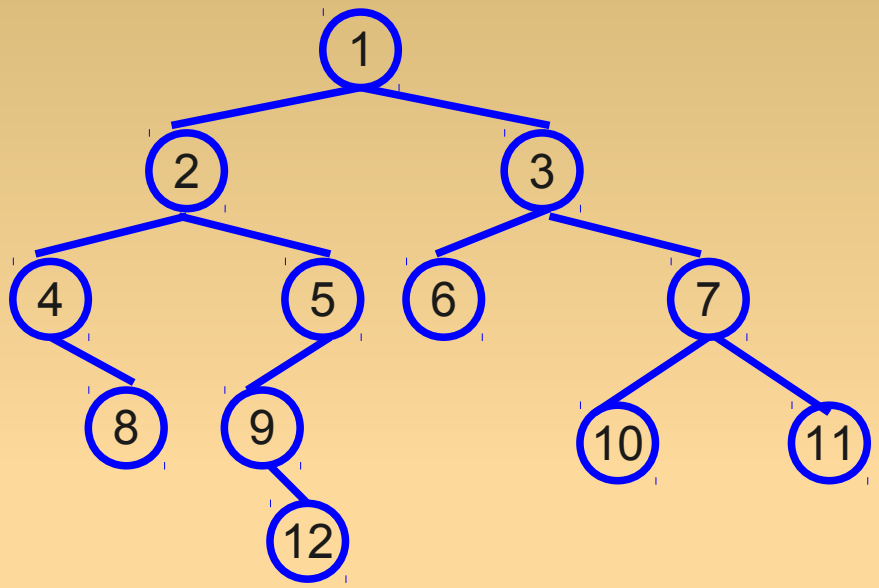
Parcours en profondeur d'abord : infixe



- Le parcours infixe consiste à parcourir de manière infixée le sous-arbre gauche puis à afficher la racine et enfin à parcourir de manière infixée le sous-arbre droit.

4 8 2 9 12 5 1 6 3 10 7 11

Parcours en profondeur d'abord : suffixe



- Le parcours suffixe consiste à parcourir de manière suffixée le sous-arbre gauche puis à parcourir de manière sufixée le sous-arbre droit et enfin à afficher la racine .

8 4 12 9 5 2 6 10 11 7 3 1

Exercice

- Ecrire un sous-programme permettant l'affichage pour chacun des parcours en profondeurs.

Programme 18

```
void prefixe(arbre a)
   {
       if (a != NULL)
           {
           printf("%d ",a->val);
           prefixe(a->gauche);
           prefixe(a->droit);
           }
   }
```

Programme 19

```
void infixe(arbre a)
  {
     if (a != NULL)
        {
        infixe(a->gauche);
        printf("%d ",a->val);
        infixe(a->droit);
        }
  }
```

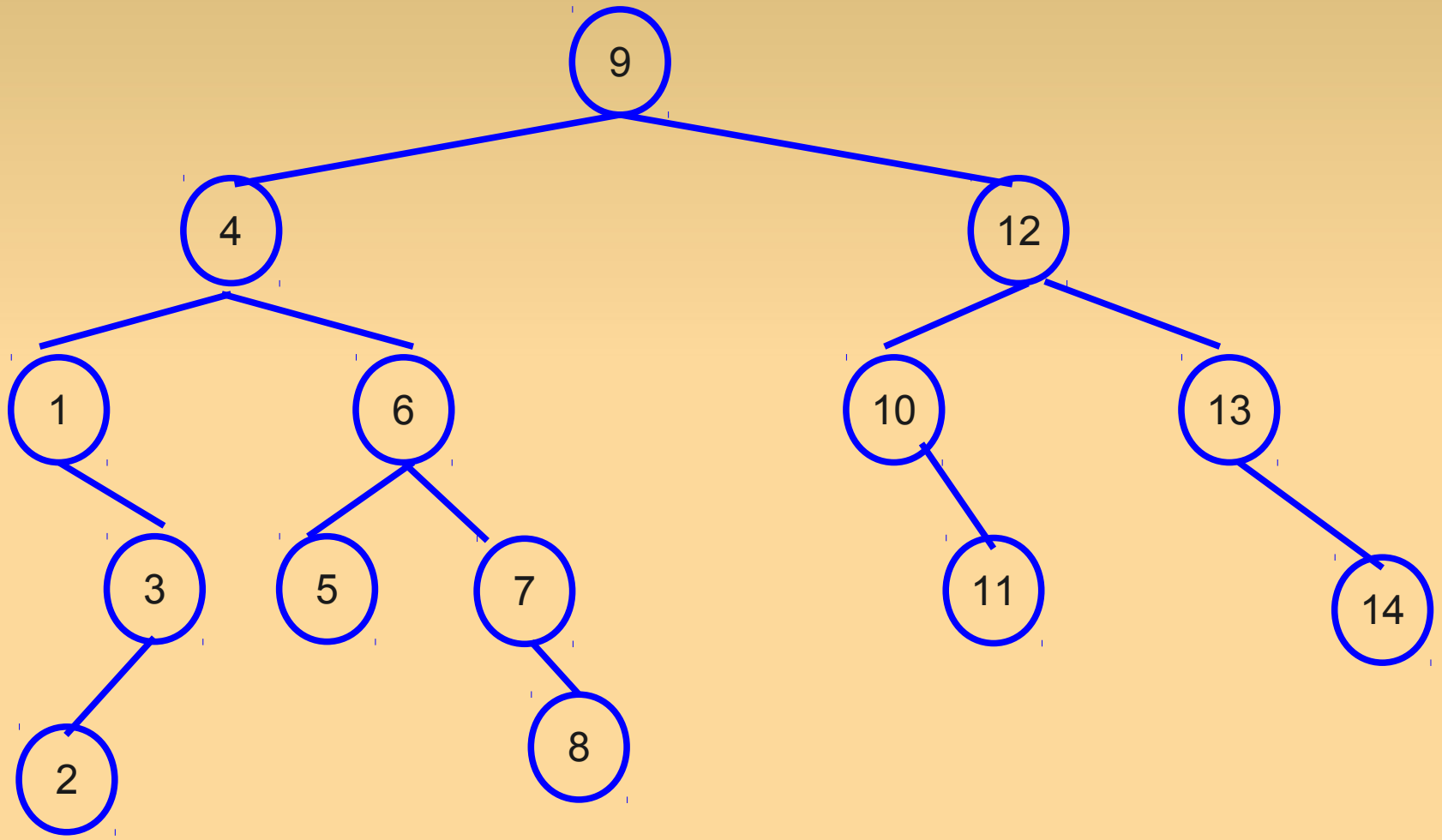
Programme 20

```
void suffixe(arbre a)
  {
     if (a != NULL)
        {
        suffixe(a->gauche);
        suffixe(a->droit);
        printf("%d ",a->val);
        }
  }
```

Arbre binaire de recherche (ABR)

- Les propriétés d'un ABR sont :
  - c'est un arbre binaire
  - tous les éléments du sous-arbre gauche sont inférieurs (ou égaux) à la racine
  - tous les éléments du sous-arbre droit sont supérieurs à la racine
  - les sous-arbres sont des ABR

Exemple



Exercice

- Ecrire un sous-programme permettant d'insérer une valeur dans un ABR.

Programme 21

```
void inserer(arbre *a,int x)
{
    if (*a == NULL)
        {
            *a = (arbre) malloc(sizeof(struct noeud));
            (*a)->gauche = NULL;
            (*a)->droit = NULL;
            (*a)->val = x;
        }
    else
        if ((*a)->val < x)
            inserer(&(*a)->droit,x);
        else
            inserer(&(*a)->gauche,x);
}
```

Arbre binaire de recherche (ABR)

- Avantages :
  - Ils permettent de faire la recherche d'une valeur plus efficacement
  - Les valeurs sont très facilement affichées dans l'ordre (infixe)
- Inconvénients :
  - Il n'y a pas unicité de l'arbre obtenu

Exercice

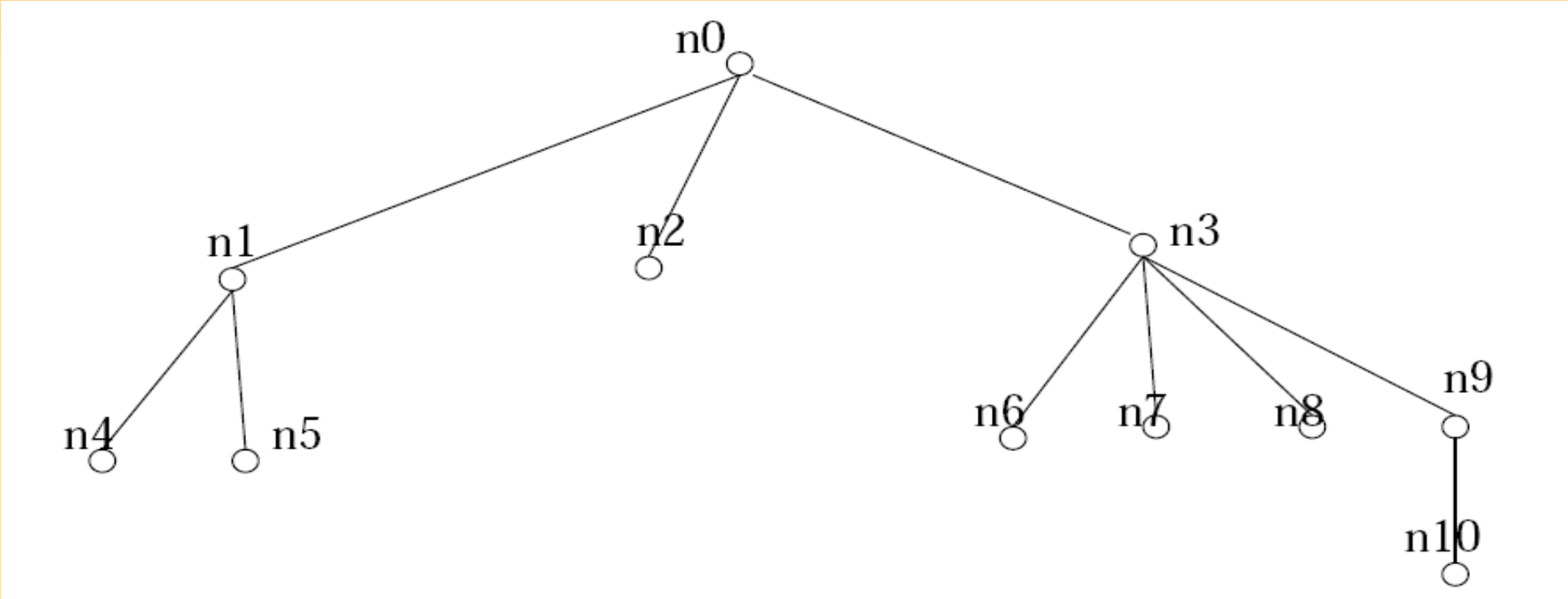
- Ecrire un sous-programme retournant 0 si une valeur donnée se trouve dans un ABR et 1 dans le cas contraire.

Programme 22

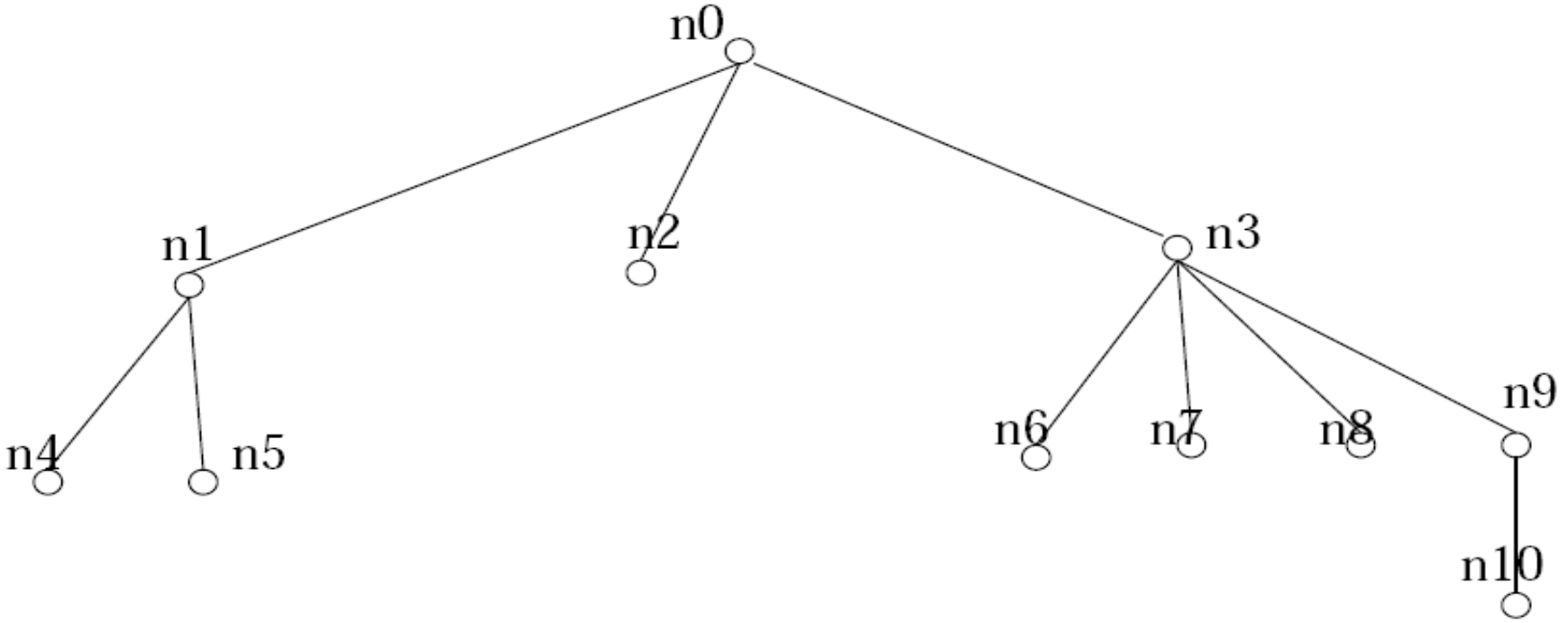
```
int rechercher(arbre a, int x)
{
    if (a == NULL) // ARBRE VIDE
        return 1;
    else
        if (a->val == x) // VALEUR TROUVEE
            return 0;
        else
            if (a->val > x) // VALEUR PLUS PETITE
                return rechercher(a->gauche,x);
            else // VALEUR PLUS GRANDE
                return rechercher(a->droit,x);
}
```

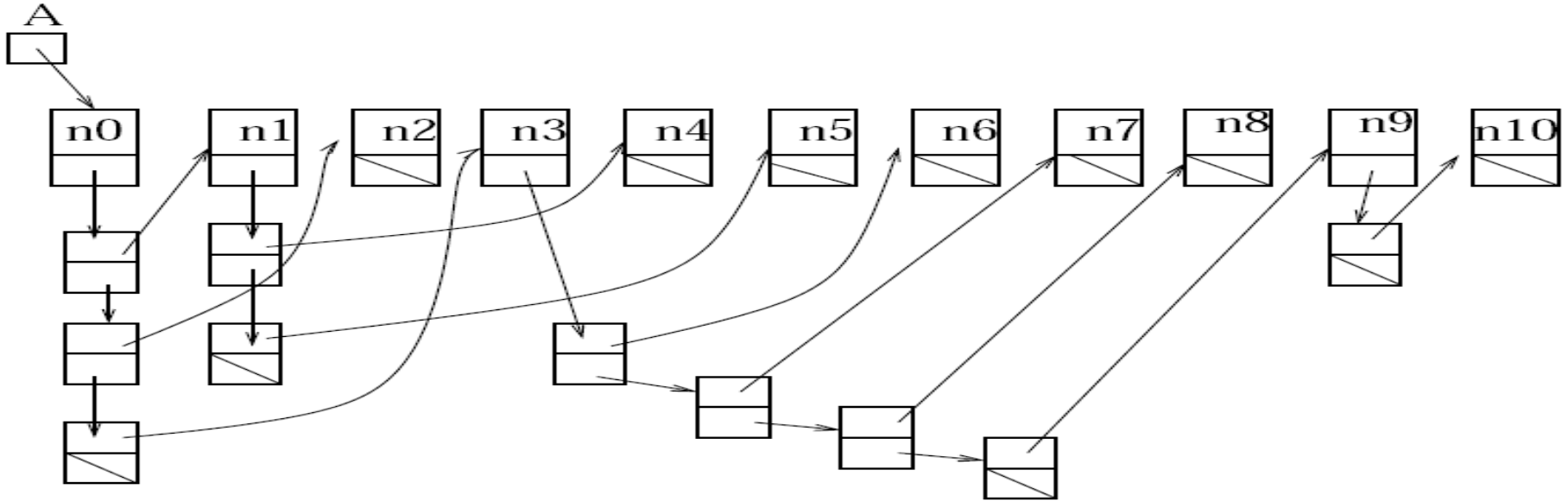
Arbre général

- Il est beaucoup plus dur de manipuler les arbres quelconques
- La structure de données est plus complexe

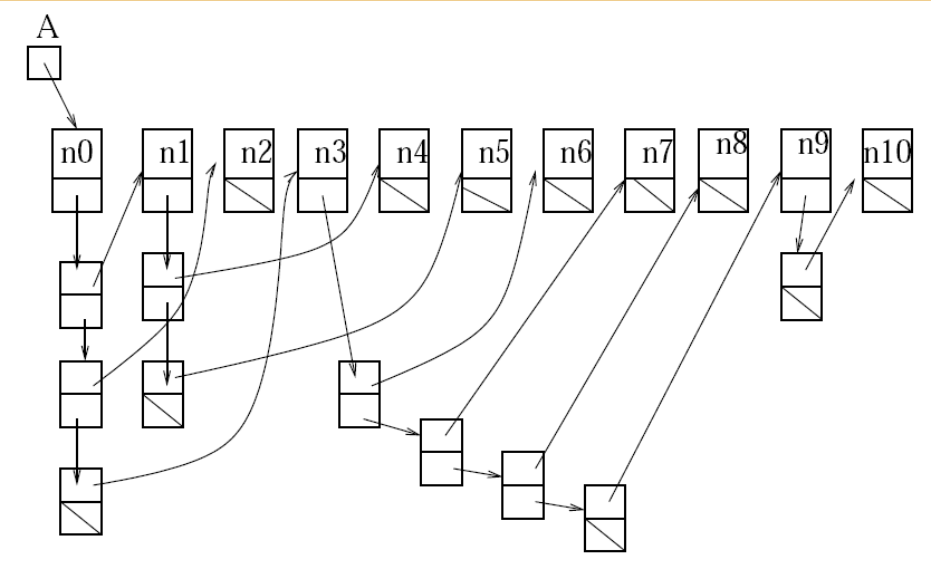


Structure de données





Structure de données




```
struct fils
   {
   struct noeud * n;
   struct fils * suivant;
   }
struct noeud
   {
   int val;
   struct fils * f;
   }
```

12) Les graphes

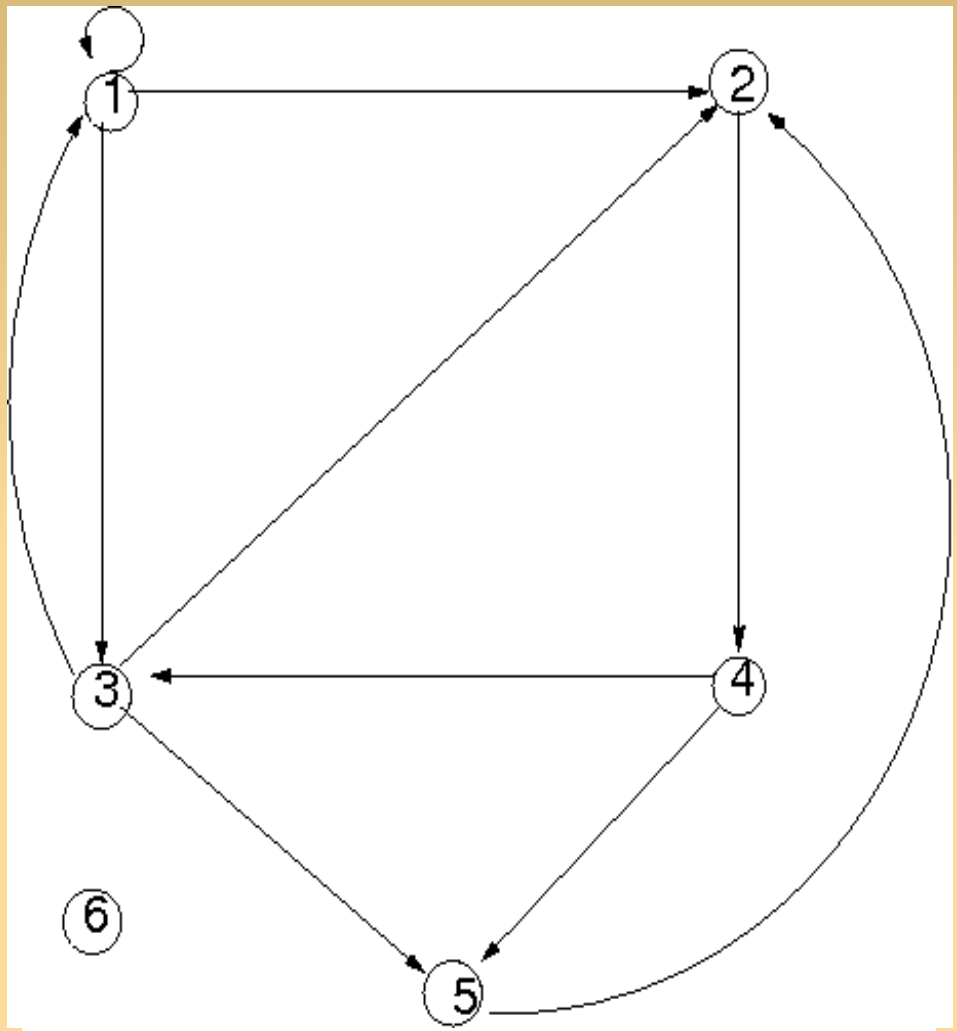
Une structure très puissante ...

Définition

- Graphe: ensemble d'objets appelés sommets et de relations (arcs) entre ces sommets.
- Exemples :
  - réseau routier, aérien, des télécommunica-tions
  - organisation des tâches dans une entreprise, dans un atelier, ...
  - carte géographique

Graphe orienté

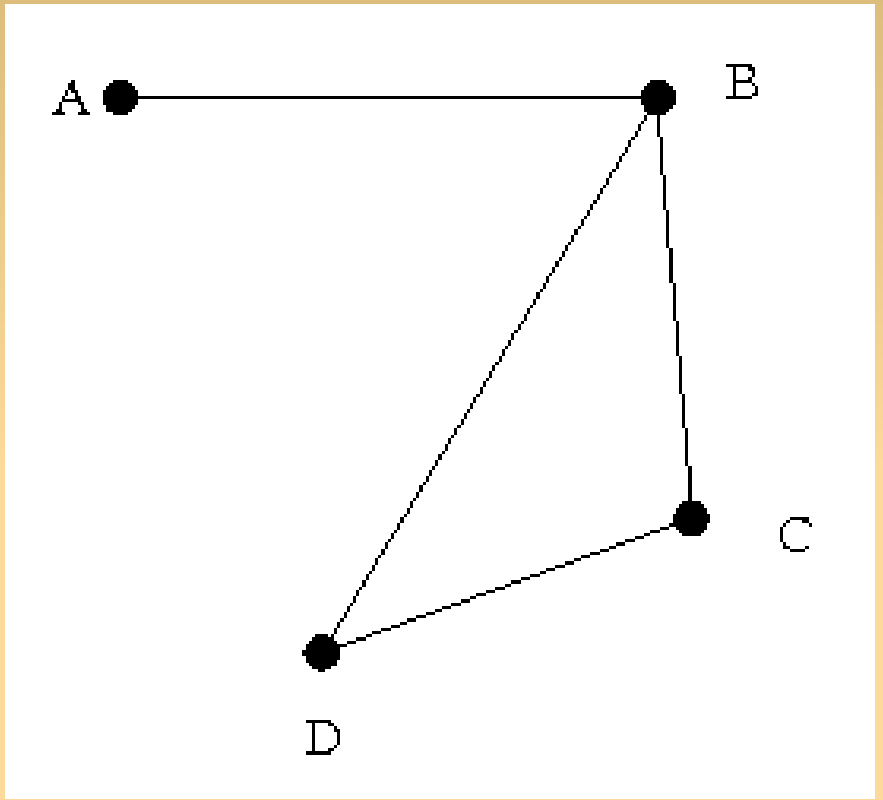
- Un **graphe orienté** est un couple **(S,A)** formé:
- d'un ensemble de sommets **S**
- d'un ensemble **A** de couples de sommets: ce sont les arcs du graphe.



$\mathcal{A} = \{(1,1), (1,2), (1,3), (2,4), (3,1), (3,2), (3,5), (4,3), (4,5),(5,2)\}$

Graphe non orienté

- Un **graphe non orienté** est un couple **(S,A)** formé:
- d'un ensemble de sommets **S**
- d'un ensemble **A** de paires de sommets: ce sont les arêtes du graphe.



$S = \{A, B, C, D\}$

$A = \{\{A, B\}, \{B, C\}, \{B, D\}, \{C, D\}\}$

Vocabulaire (1/2)

- Si (x,y) est un arc, x est appelé extrémité initiale de l'arc et y extrémité terminale. On dit que y est un successeur de x et que x est un prédécesseur de y.
- Le nombre d'arcs partant de x est appelé demi-degré extérieur de x et noté $\mathbf{d^+(x)}$.
- Le nombre d'arcs arrivant en x est appelé demi-degré intérieur de x et noté $\mathbf{d^-(x)}$.

Vocabulaire (2/2)

- Le degré de x, **d(x)** est le nombre d'arcs dont x est une extrémité :
  - $d(x)=d^{+}(x) + d^{-}(x)$
- Dans un graphe non orienté, le degré d'un sommet x est le nombre d'arêtes dont x est une extrémité.

Concepts

- Un chemin de longueur l est une suite (s_1, ..., s_{l+1}) de (l+1) sommets tels que (s_1,s_2), ... ,(s_l, s_{l+1}) sont des arcs.
- Une chaîne de longueur l est une suite (s_1, ..., s_{l+1}) de (l+1) sommets tels que (s_1,s_2), ... ,(s_l, s_{l+1}) sont des arêtes.
- Un chemin ou une chaîne (s_1, ..., s_k) est élémentaire si aucun sommet n'apparaît plus d'une fois dans s_1, ..., s_k

Circuits et cycles (1/2)

- Dans un graphe orienté, un chemin (s_1, ..., s_{l+1}) dont les l arcs sont tous distincts 2 à 2 et dont les sommmets aux extrémités coïncident est un circuit.
- Dans un graphe non orienté, une chaîne (s_1, ..., s_{l+1}) dont les l arêtes sont toutes distinctes 2 à 2 et dont les sommmets aux extrémités coïncident est un cycle.

Circuits et cycles (2/2)

- Un chemin est acyclique si aucun sommet n'apparaît plus d'une fois dans le chemin.
- Pour u != v, s'il existe un chemin de u à v, il existe un chemin acyclique de u à v.

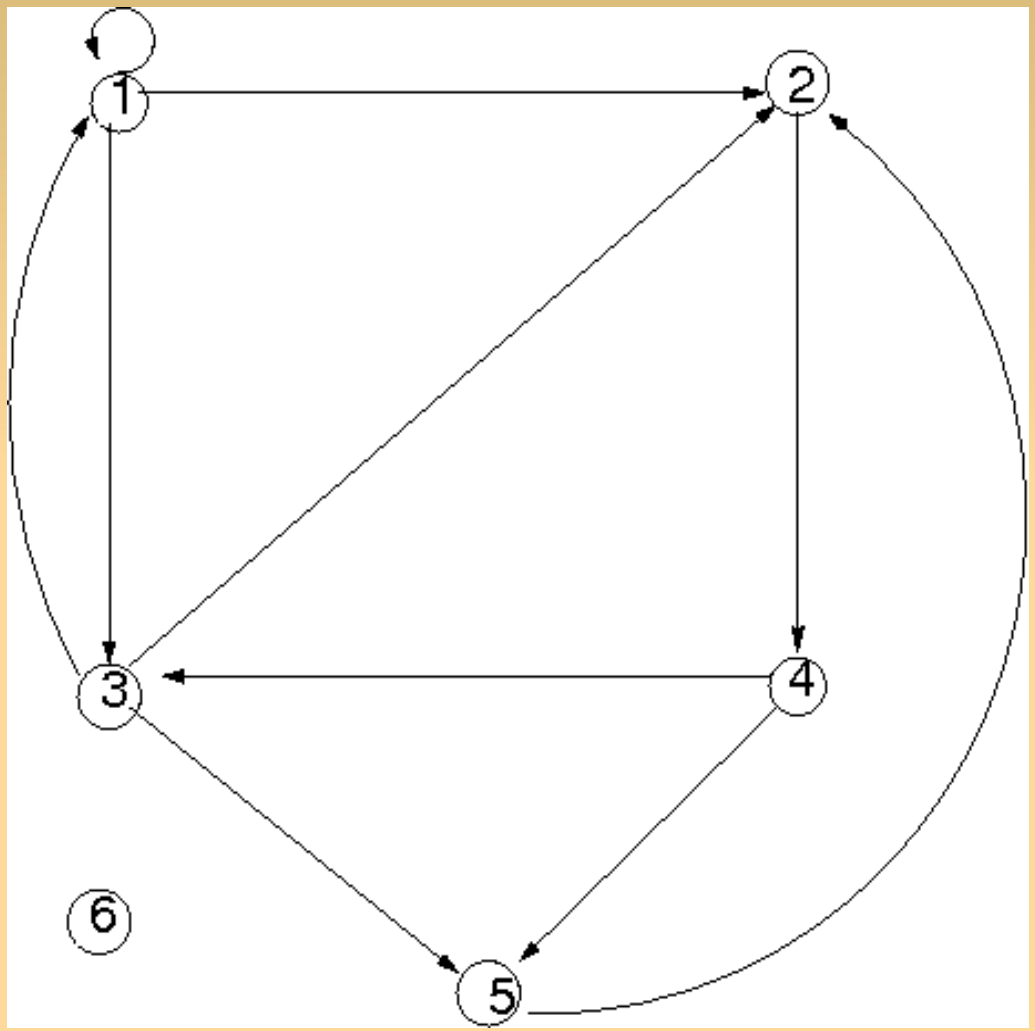
Implémentation des graphes

- Deux représentation principales:
  - Par matrice d'adjacence
  - Par listes d'adjacence
- On se place ici dans le cas où le nombre de sommets *n* n'évolue pas.

Représentation par Matrice d'adjacence

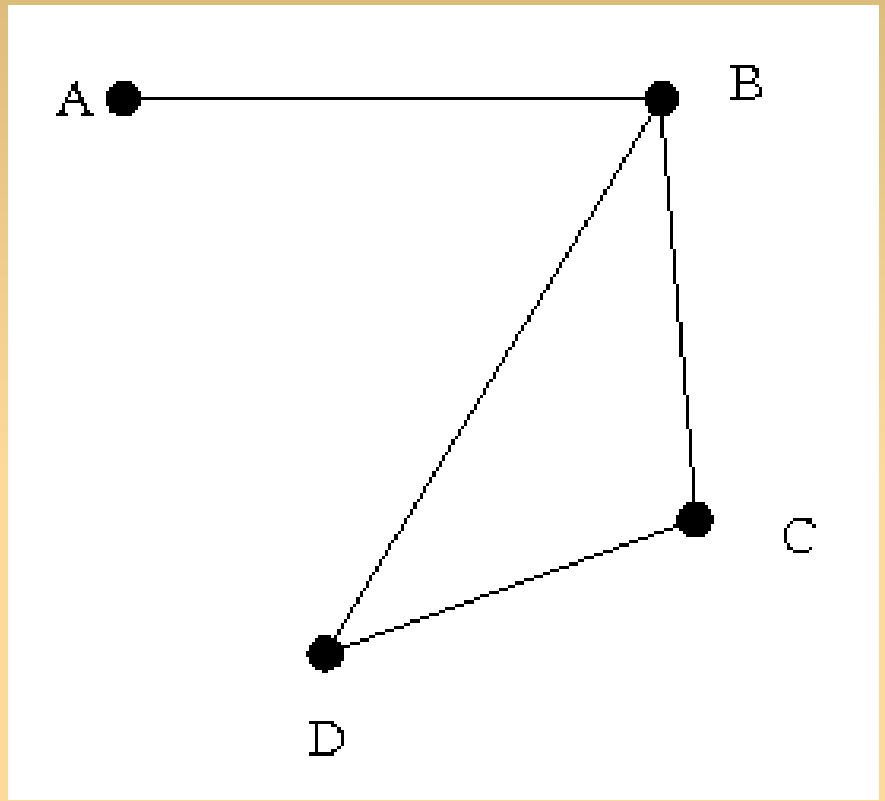
- L'ensemble des arcs du graphe est représenté dans une matrice d'adjacence.
- #define n ...
- typedef int GRAPHE[n][n];
- G[i][j] est à 1 si et seulement si il existe un arc entre les sommets i et j (0 dans le cas contraire).

Représentation par Matrice d'adjacence (1/3)



	1	2	3	4	5	6
1	1	1	1	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	1	0	1	0
5	0	1	0	0	0	0
6	0	0	0	0	0	0

Représentation par Matrice d'adjacence (2/3)



	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0

Dans le cas d'un graphe non orienté, la matrice est symétrique.

Représentation par Matrice d'adjacence (3/3)

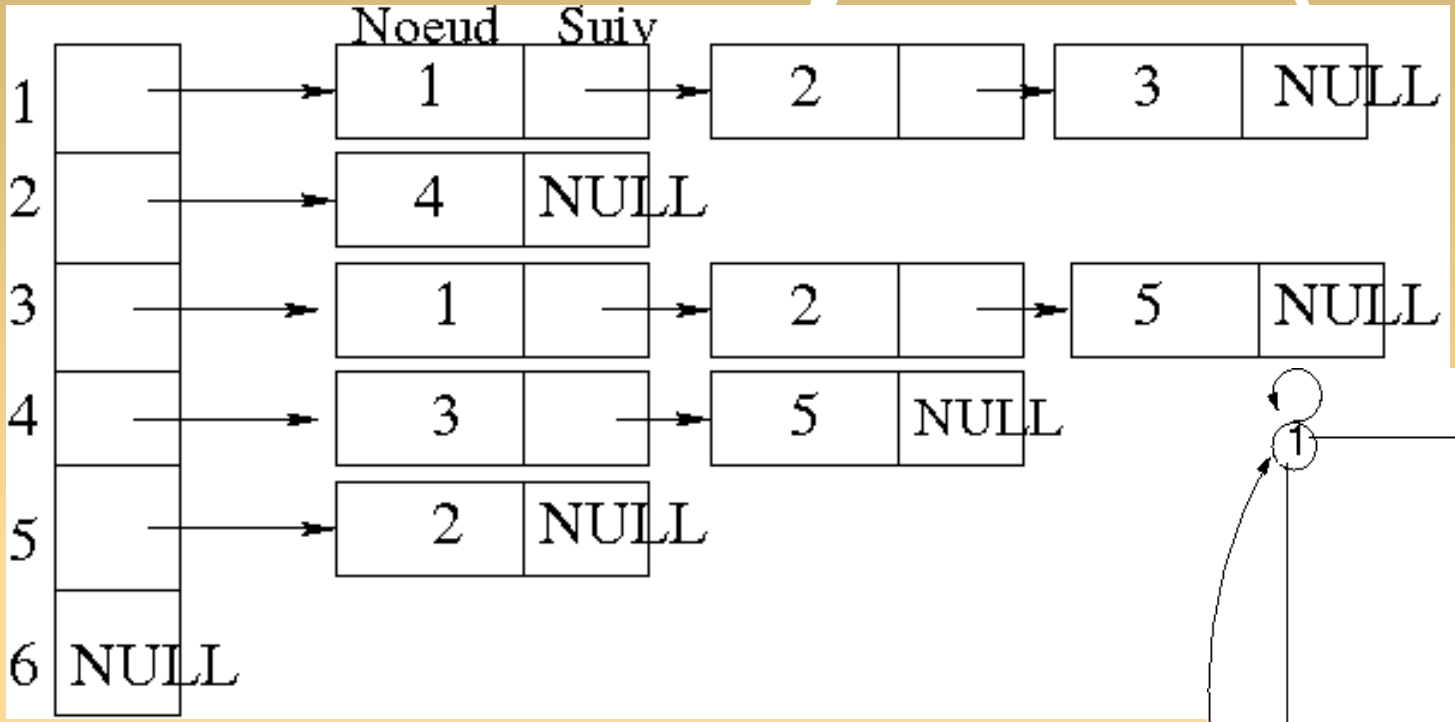
- Dans le cas d'un graphe valué, G[i,j] a pour valeur :
  - le poids de l'arc (de l'arête) entre i et j si l'arc (l'arête) entre i et j existe
  - une valeur arbitraire qui ne peut pas être un poids et qui représente l'absence d'arc (d'arête)

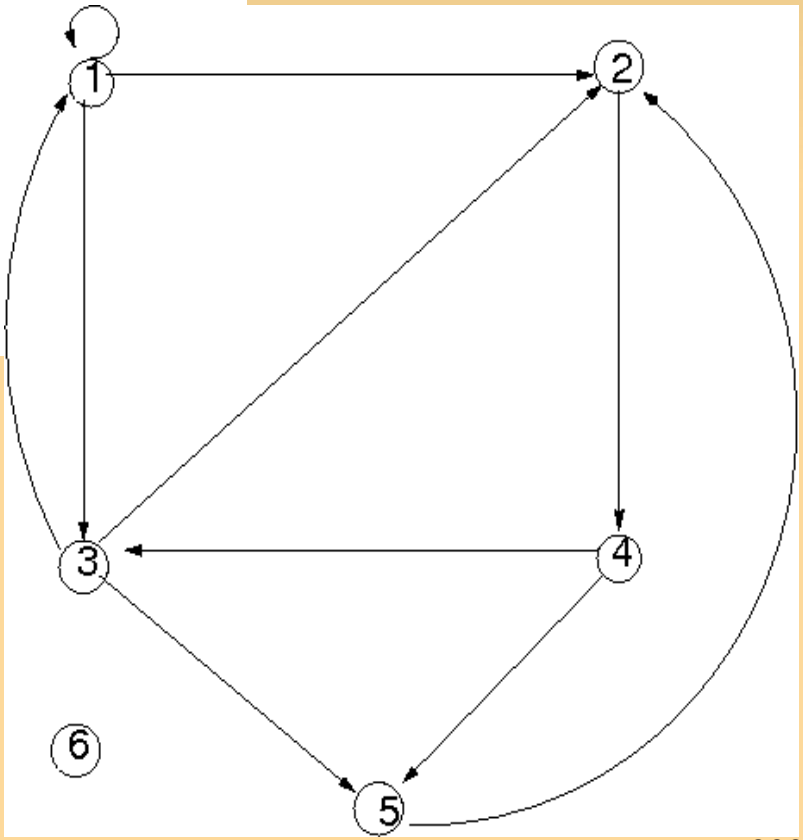
Représentation par Listes d'adjacence (1/2)

- On représente l'ensemble des sommets et on associe à chaque sommet la liste de ses successeurs rangés dans un certain ordre:

```
struct cellule
{
    int noeud;
    struct cellule * suivant;
};
struct cellule * GRAPHE[n];
```

Représentation par Listes d'adjacence (1/2)





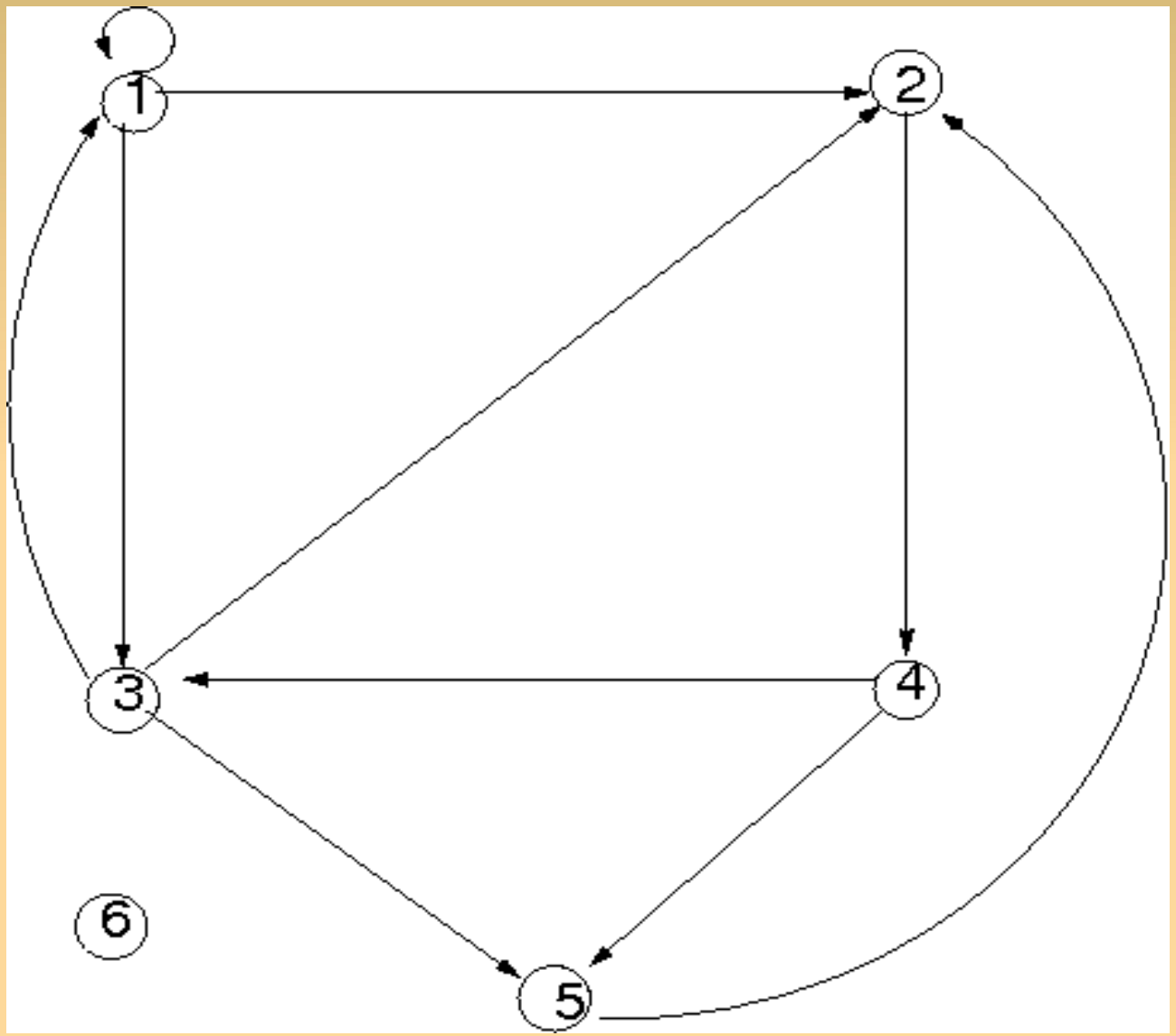
Parcours d'un graphe

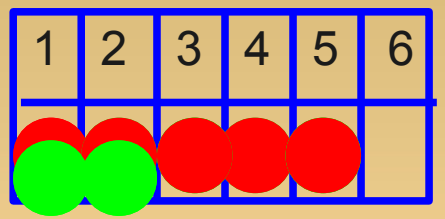
- Comme pour les arbres, il existe deux principaux parcours :
  - le parcours en profondeur
  - le parcours en largeur

Parcours en profondeur

- Parcours de tous les sommets atteignables à partir d'un sommet donné :
  - A partir d'un sommet donné, suivre un chemin le plus loin possible(profondeur).
  - Lorsqu'on arrive au bout d'un chemin, on explore l'embranchement le plus récent non exploré.
  - Un marquage permet de ne pas passer plusieurs fois par un même sommet.

Parcours en profondeur





1 : 1 2 4 3 5

2 : 2 4 3 1 5

3 : 3 1 2 4 5

4 : 4 3 1 2 5

5 : 5 2 4 3 1

6 : 6

Algorithme récursif de parcours en profondeur

Liste d'adjacence

```
/* s est un sommet. On parcourt tous les sommets que l'on peut atteindre depuis s. M est le tableau de marquage des sommets initialisé avec des 0 */
void prof(int s, GRAPHE g, int M[n])
{
    struct cellule * tmp = g[s-1];
    printf("%d ",s);
    M[s-1] = 1;
    while (tmp != NULL)
    {
        if (M[tmp->noeud-1] == 0)
         prof(tmp->noeud,g,M)
         tmp = tmp->suiv;
     }
}
```

Algorithme récursif de parcours en profondeur

Matrice d'adjacence

```
/* s est un sommet. On parcourt tous les sommets que l'on peut atteindre depuis s. M est le tableau de marquage des sommets initialisé avec des 0 */
void prof(int s, GRAPHE g, int * M[n])
{
     int i;
     printf("%d ",s);
     *M[s-1] = 1;
     for(i=0;i<n;i++)
         if ((g[s-1][i] == 1) && (*M[i] == 0))
           prof(i+1,g,&(*M));
 }
```

Parcours en largeur

- Pour un sommet de départ **s**, on visite d'abord tous les sommets à distance "1 arc" de **s**, puis tous les sommets à distance "2 arcs" de **s**, ...
- On utilise une file pour réaliser ce parcours de façon itérative: à partir d'un sommet **s**, on visite ses successeurs non marqués et on les range dans une file puisque la recherche au niveau suivant repartira de chacun de ces successeurs à partir du premier.

Algorithme de parcours en largeur (matrice d'adjacence)

```
void larg(int s, int g[n][n], int M[n])
{
      int v,i;
      File f,tmp;
      f = NULL;
      M[s-1] = 1;
      printf("%d ",s);
      enfiler(&f, s);
      while (estvide(f) == 1)
        {
          tmp = defiler(&f);
          v = tmp->val;
          free(tmp);
          for(i=0;i<n;i++)
          if ((M[i] == 0) && (g[v-1][i] == 1))
            {
                  printf("%d ",i+1);
                  M[i]  = 1;
                  enfiler(&f,i+1);
            }
        }
}
```

La théorie des graphes

- Coloriage de graphes
- Plus court chemin
- Problème du voyageur de commerce
- ...

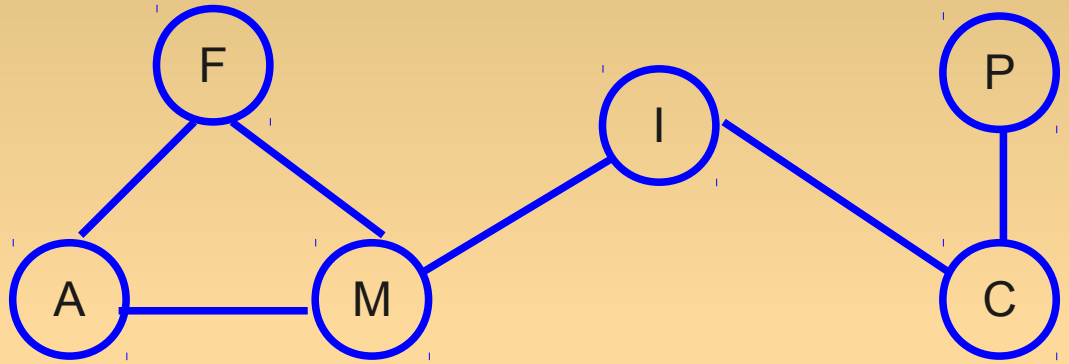
Coloriage de graphes

- Problème du coloriage : attribuer une couleur à chaque sommet de telle sorte que deux sommets adjacents n'aient pas la même couleur.
- Le nombre chromatique d'un graphe est le plus petit nombre de couleurs permettant de colorier tous les sommets sans que deux sommets adjacents soient de la même couleur.

Application (1/2)

- Problème: on veut organiser les examens dans une licence où il y a 6 matières: Français (F), Anglais(A), Maths (M), Physique (P), Informatique (I) et Chimie (C).
- Les étudiants se répartissent dans 4 filières où les choix des matières sont respectivement:
  - F et A et M
  - P et C
  - I et C
  - I et M
- Chaque épreuve dure une demi-journée. Quel est le temps minimal nécessaire pour organiser les examens?

Application (2/2)



F et A et M
P et C
I et C
I et M

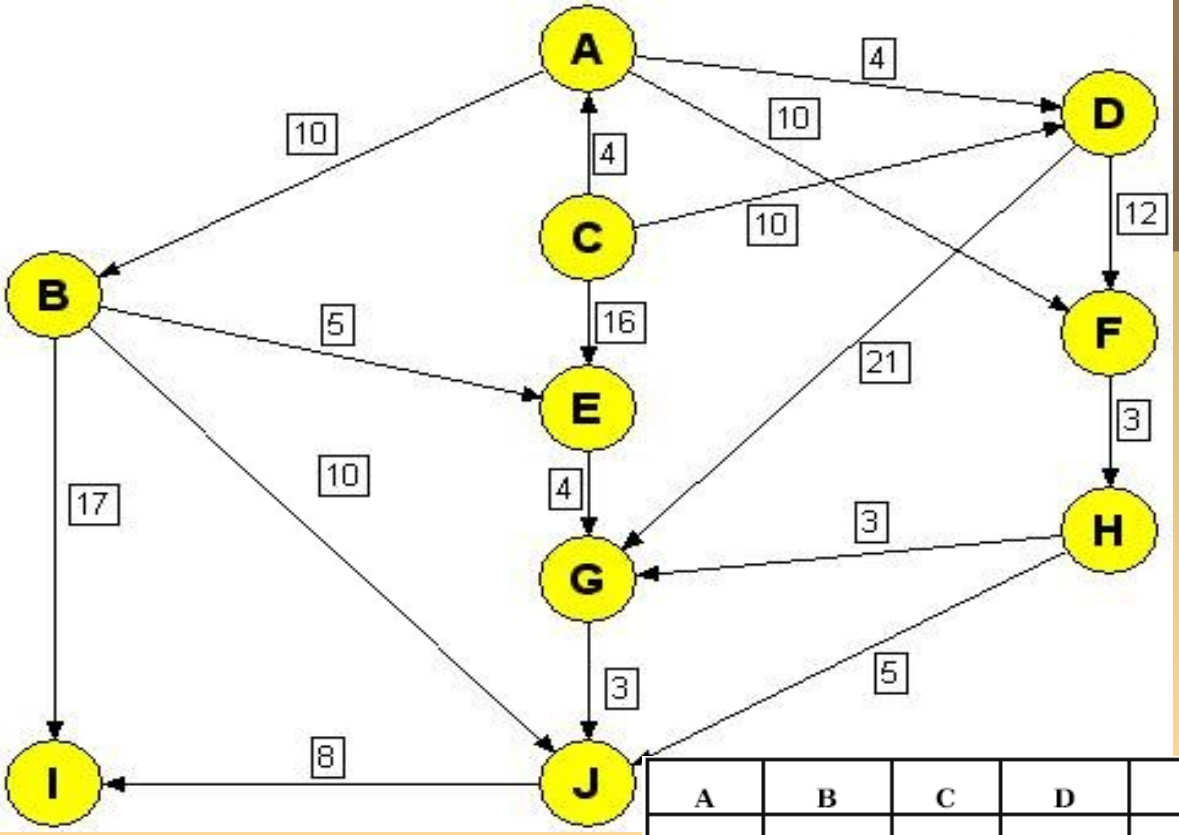
3

Le plus court chemin (routage internet)

- On considère G=(S,A) un graphe valué, c'est-à-dire qu'à tout arc a=(i,j) est associée une valeur appelée longueur ou coût de l'arc.
- Le problème du plus court chemin entre 2 sommets **i** et **j** consiste à trouver un chemin de longueur minimale entre **i** et **j**.
- Applications: la valeur de l'arc peut correspondre à:
  - - une distance
  - - un coût de parcours
  - - un temps de parcours

Algorithme de Dijkstra

- Graphe valué orienté ou non, où tous les coûts des arcs sont positifs ou nuls.
- Pour un sommet source **s** donné, calcule pour tous les autres sommets **u** du graphe le plus court chemin entre **s** et **u**.
- On considère un ensemble **F** de sommets *fixés* : sommets **u** pour lesquels le plus court chemin de **s** à **u** est connu. A chaque itération, on ajoute un sommet dans **F** et on met à jour la meilleure distance connue entre **s** et chaque sommet non fixé.



A	B	C	D	E	F	G	H	I	J	Sél	Coef
∞	∞	0	∞	∞	∞	∞	∞	∞	∞	C	0
4 C	∞		10 C	16 C	∞	∞	∞	∞	∞	A	4
	14 A		8 A	16 C	14 A	∞	∞	∞	∞	D	8
	14 A			16 C	14 A	29 D	∞	∞	∞	B	14
				16 C	14 A	29 D	∞	31 B	24 B	F	14
				16 C		29 D	17 F	31 B	24 B	E	16
						20 E	17 F	31 B	24 B	H	17
						20 E		31 B	22 H	G	20
								31 B	22 H	J	22
								30 J		I	30

F.Lardeux

Le problème du voyageur de commerce

- On appelle circuit ou cycle hamiltonien un circuit ou un cycle qui passe une et une seule fois par chaque sommet du graphe.
- Le problème du voyageur de commerce consiste à trouver un cycle hamiltonien de coût minimal dans un graphe valué.

Graphe de jeux

- Exemple: jeu de grundy
  - Deux joueurs ont en face d'eux un tas de pièces de monnaie. Chaque joueur à son tour doit diviser le tas initial ou un des tas sur la table en deux tas **inégaux**. Le jeu s'arrête lorque chaque tas contient une seule ou deux pièces. Le premier des 2 joueurs qui ne peut pas jouer est le perdant.
  - Supposons que les joueurs s'appellent A et B. A joue le premier avec un tas initial qui contient 7 pièces . Construire un graphe qui montre que B peut toujours gagner quelque soit ce que fait A.

13) Complexité

Programmer intelligemment

Qu'est-ce que la complexité

- Exécution d ’un programme = utilisation des ressources de l’ordinateur :
  - temps de calcul pour exécuter les opérations
  - occupation mémoire (programme + données)
- Mesurer ces 2 grandeurs pour comparer entre eux différents algorithmes
  - « Sur toute machine, quelque soit le langage de programmation, l’algorithme A1 est meilleur que A2 pour des données de grande taille »

Temps d ’exécution d ’un programme

- Mettre en évidence les opérations fondamentales: le temps est proportionnel au nombre de ces opérations
  - Recherche d ’un élément E dans une liste L : nombre de comparaisons entre E et les éléments de L
  - Recherche d ’un élément sur disque : nombre d’accès à la mémoire secondaire

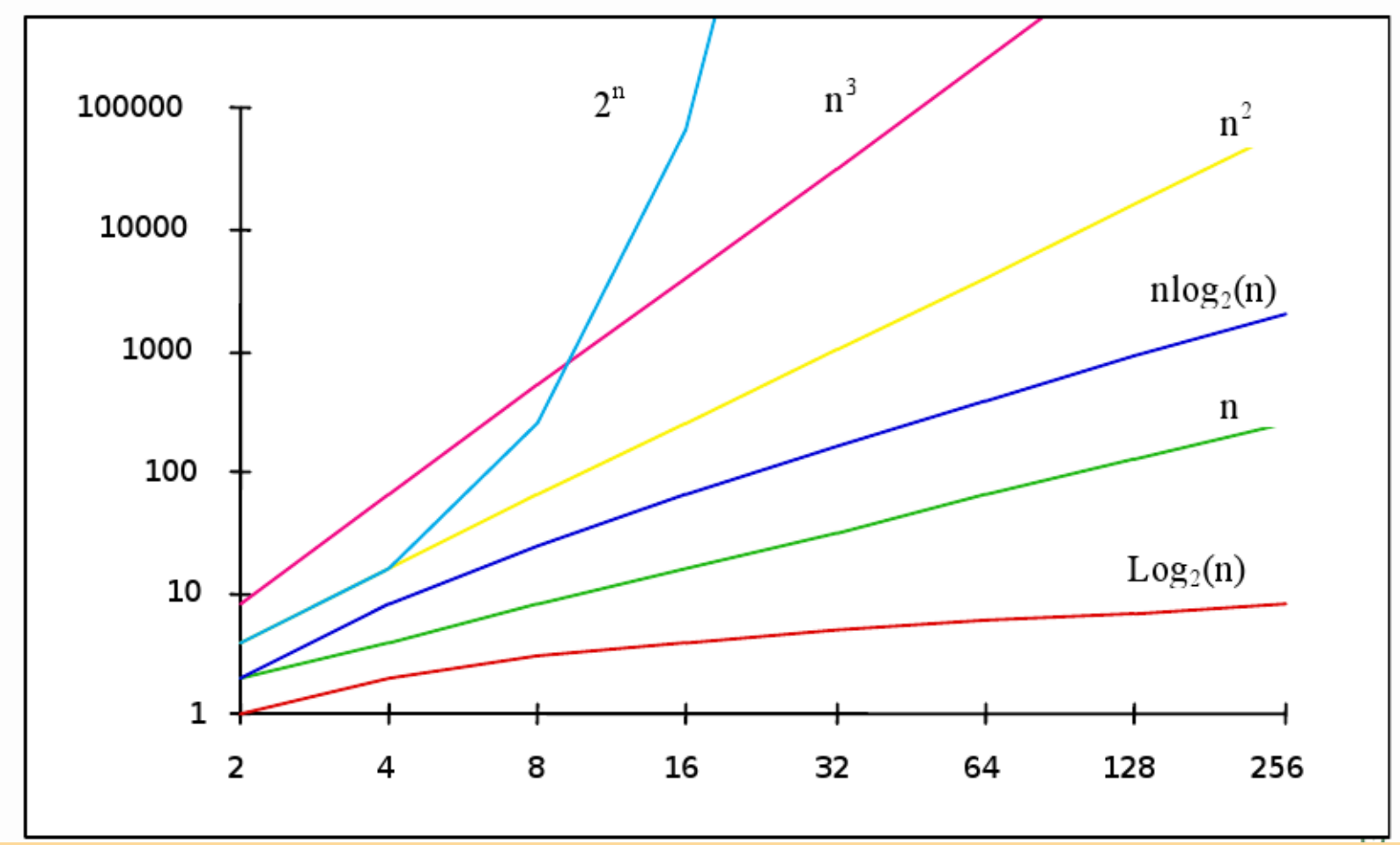
Temps d ’exécution d ’un programme

- Soit n la taille des données soumise au programme
  - taille de la liste dans laquelle on cherche E
  - taille de la liste à trier
  - dimensions d ’une matrice dont on calcule le carré
  - ...
- T(n) : temps d ’exécution du programme

Appellation des ordres de grandeur courant

- O(1): constante
- O(log(n)): logarithmique
- O(n): linéaire
- O(n log n): n log n
- O(n2) : quadratique
- O(n3) : cubique
- O(2n): exponentielle

Croissance comparée des fonctions fréquemment utilisées en complexité



Exemple

```
Liste defiler(Liste *maliste)
{
        Liste tmp,tmp2;
        if (*maliste == NULL)
                return NULL;
        else
                {
                        tmp = *maliste;
                        if ((*maliste)->suivant == NULL)
                                {
                                        (*maliste) = NULL;      }
                                        return tmp;
                                }
                                        else
                                                {
                                                        while(tmp->suivant->suivant != NULL)
                                                                tmp=tmp->suivant;
                                                        tmp2 = tmp->suivant;
                                                        tmp->suivant = NULL;
                                                        return tmp2;
                                                }
        }
```

Conclusion

Ce qu'il faut retenir ...

Conclusion

- Penser à bien choisir votre structure de données en fonction de l'utilisation (taille des données à manipuler) et de la machine où tournera votre programme.
- Penser à écrire du code ne gaspillant pas l'espace mémoire (libération des zones réservées) et ne faisant pas des opérations inutiles (complexité).

Conclusion

FIN ...