

Module:

Informatique 4

STRUCTURES DE DONNÉES

Pour SMA-S4
A.U: 2018-2019

Notion de Type Abstrait de Données (TAD)

Définition: Un type Abstrait de Données (TAD) est un *ensemble de données (valeurs) muni d'opérations sur ces données, sans faire référence à une implémentation particulière (en langage C, en java, en Python, ...)*.

Exemples:

- *Dans un algorithme qui manipule des entiers, on s'intéresse, non pas à la représentation des entiers, mais aux opérations définies sur les entiers : +, -, *, /.*

On ne se soucie pas de la représentation d'un entier en mémoire: complément à 2, signe+valeur absolue, ...

Notion de Structure de Données (SD)

Définition: Une structure de données Correspond à l'implémentation d'un type abstrait de données. Elle est composée d'un algorithme pour chaque opération et des données spécifiques à la structure pour sa gestion.

Dans ce module nous allons voir les structures de données suivantes:

- 1) Structures de données linéaires: listes, files et piles
- 2) Structures de données arborescentes: arbres binaires, arbres binaire de recherche, tas, hachage, arbre équilibrée.
- 3) Graphes: terminologie, représentation, algorithmes de parcours.

RAPPEL SUR LES FONCTIONS EN LANGAGE C

```
<Type de résultat> <Nom de la fonction> <liste des entrées (paramètres)>  
{ .....  
    return (.....) ; // l'existence de void implique l'absence de return  
}
```

Exemple 1:

```
float Somme( float a, float b)  
{ float c;  
    c=a+b;  
    return (c) ; }
```

Exemple 2:

```
void AfficherSomme( float a, float b)  
{ float c;  
    c=a+b;  
    printf(« La somme = %f » ,c) ; }
```

Utilisation des fonctions (appel):

- L'instruction `r=Somme(10,30)` ; met dans la variable `r` la valeur 40
- L'instruction `AfficherSomme (10,30)` affiche la valeur 40 sur l'écran

RAPPEL SUR LES FONCTIONS EN LANGAGE C

Exemple 3:

```
int fact(int n)
{ int f=1, i;
  for(i=1;i<=n;i++)
    f=f*i;
  return (f) ;
}
```

Exemple 4:

```
int NombreArrangements(int n, int p)
{ int r;
  r = fact(n)/fact(n-p);
  return(r);
}
```

Exemple 5:

```
int NbrCombinaisons(int n, int p)
{ int r;
  r= fact(n)/(fact(p)*fact(n-p));
  return(r);
}
```

Exemple 6:

```
int NbrCombinaisons2(int n, int p)
{ int r,q;
  r= NombreArrangements(n,p);
  q=r/fact(p);
  return(q); }
```

En mathématiques une fonction récursive est une fonction qui s'appelle elle-même. Il est possible en langage C de définir une fonction qui s'appelle elle-même, on parlera alors de fonction récursive.

Prenons le cas de la suite de Fibonacci, définie par :

- en mathématiques :

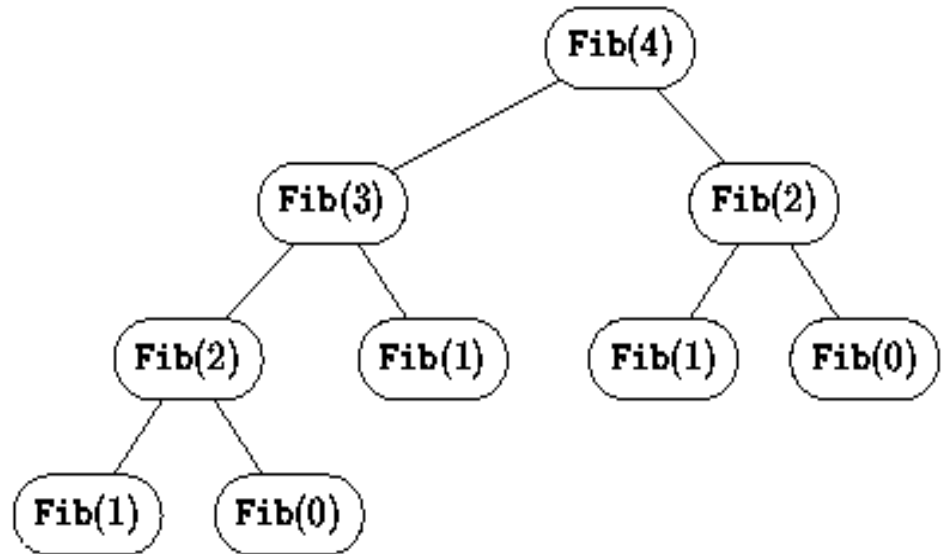
$$u_0 = u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2} \text{ pour } n > 1$$

- en informatique :

```

int fib (int n)
{ if((n==0) || (n==1) )return(1);
  return fib (n-1) + fib (n-2); }
  
```



Appels récursifs pour fib (4)

$$\begin{aligned}
 \text{fib}(4) &= \text{fib}(3) + \text{fib}(2) \\
 &= \text{fib}(2) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0) \\
 &= \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0)
 \end{aligned}$$

Un autre exemple de fonction récursive

Prenons l'exemple de la fonction factorielle :

- en mathématiques :

$$n! = n.(n-1)!$$

pour $n \geq 1$

avec $0!=1$

- en informatique :

```

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

Exemple: int factorielle (3)

```

{
return 3*factorielle(2);
}
  
```

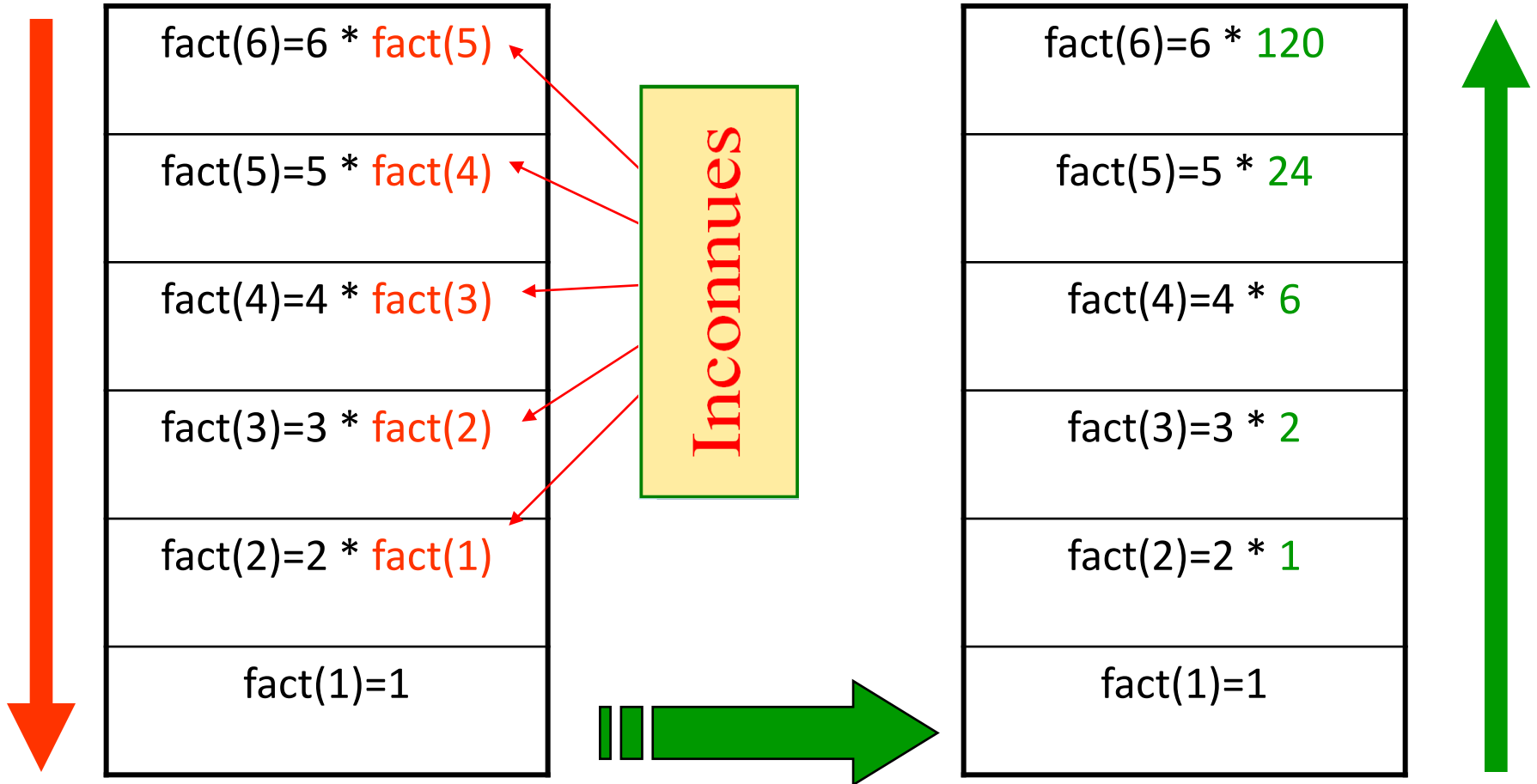
mémoire les résultats intermédiaires

```

int factorielle (2)
{
return 2*factorielle(1);
}
  
```

```

int factorielle (1)
{
return 1*factorielle (0);}
  
```



On remarque bien les appels récursifs.

Pour calculer factorielle de i il faut attendre le calcul des factoriels des j qui lui sont strictement inférieurs. Donc on descend et on remonte !

RAPPEL SUR LES POINTEURS EN LANGAGE C

- 1) Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés par un numéro qu'on appelle adresse mémoire.
- 2) On dit que p est un pointeur sur la variable n si $p = \&n$; // p contient l'adresse de n

Exemple 7:

```
int main()
{ int a=11;
  int* p=&a;
  printf(" le contenu de l'adresse %ld = %d ", p, *p);
  return(0);
}
```

Ce programme affiche sur l'écran:

le contenu de l'adresse 3542274588= 11

L'adresse **3542274588** peut changer si on exécute de nouveau ce programme sur la même machine ou sur une autre machine

RAPPEL SUR LES FONCTIONS (SUITE)

Question importante : Quant-est-ce qu'une fonction f reçoit un argument x (paramètre) par adresse pointeur obligatoirement ?

Réponse: Parmi ces cas, c'est lorsqu'on veut changer la valeur d'un paramètre x d'une fonction f à l'intérieur de cette fonction.

Exemple 8:

```
void permuter (int*adresse1, int*adresse2)
{ int c = *adresse1;
  *adresse1=*adresse2;
  *adresse2=c ;}

int main()
{ int a=11, b=99;
  permuter(&a, &b);
  printf(" a=%d et b= %d ", a, b);
  return(0); }
```

Ce programme affiche sur l'écran: a=99 et b=11

Structures de données séquentielles (SDS)

- *SDS tableau (vu en S2 et S3), muni des opérations: trier, recherche séquentielle, recherche dichotomique dans un tableau trié, insertion d'un élément, suppression d'un élément, inversion,....*
- *SDS chaîne de caractères, muni des opérations: longueur, comparaison de 2 chaînes, copie d'une chaîne dans une autre, concaténation de deux chaînes, extraction d'une sous chaîne, rendre en majuscule/minuscule*

Remarque: *la SDS chaîne de caractères est un cas particulier de la SDS tableau*

Tri à bulles

Le principe du tri à bulles consiste à comparer deux à deux les éléments consécutifs d'un tableau et d'effectuer une permutation s'ils ne sont pas ordonnés. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation à faire.

La fonction (procédure) permuter:

Entrées: - un tableau T de taille n
- deux positions i et j.

Sortie: le tableau T après permutation des cases T[i] et T[j]

Début

```
C ← T[i]
T[i] ← T[j]
T[j] ← C
```

Fin

TRI À BULLES

Algorithme Tri à bulles:

Entrées: un tableau T de taille n

Sorties: le tableau T trié

Début

Pour $i:=1$ à $n-1$ faire:

Pour $j:=1$ à $n-1$ faire:

Si $T[j] > T[j+1]$ alors:
 permuter ($T, j, j+1$)

Fin si

Fin pour

Fin pour

Fin

3	7	2	6	5	1	4
3	2	6	5	1	4	7
2	3	5	1	4	6	7
2	3	1	4	5	6	7
2	1	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

La fonction permuter en langage C:

```
Void permuter(float T[], int i, int j)
{
    float c=T[i];
    T[i]=T[j];
    T[j]=c;
}
```

La fonction Tri à Bulles en langage C:

```
Void TriAbulles(float T[], int N)
{
    int i,j;
    for (i=0;i<N-1;i++)
        for(j=0;j<N-1;j++)
            if(T[j]>T[j+1])
                {permuter(T,j,j+1);}
}
```

TRI PAR SÉLECTION

Principe de Tri par sélection:

Le tri par sélection consiste à chercher la plus petite valeur de la liste, puis de la mettre à la première place en l'échangeant avec la première valeur. On répète alors la procédure sur le sous tableau constitué des nombres restants et cela jusqu'à obtenir un tableau trié.

```
void permuter(float T[], int i, int j)
{float c=T[i];  T[i]=T[j];T[j]=c; }
```

```
void TriSelection(float T[], int N)
{int i,j,Posmin;
 for(i=0;i<N;i++)
  {Posmin=i;
   for(j=i+1;j<N;j++)
    if (T[j]<T[Posmin]) Posmin=j;
   permuter(T,i,Posmin);
  }
}
```

3	7	2	6	5	1	4
1	7	2	6	5	3	4
1	2	7	6	5	3	4
1	2	3	6	5	7	4
1	2	3	4	5	7	6
1	2	3	4	5	7	6
1	2	3	4	5	6	7

TRI PAR INSERTION

Principe de Tri par insertion:

Le tri par insertion consiste à insérer le premier élément du tableau à trier au premier emplacement d'une nouvelle liste, puis on insère chaque nouveau élément à sa bonne place dans la liste déjà triée. On répète jusqu'à placer tous les éléments à la bonne place).

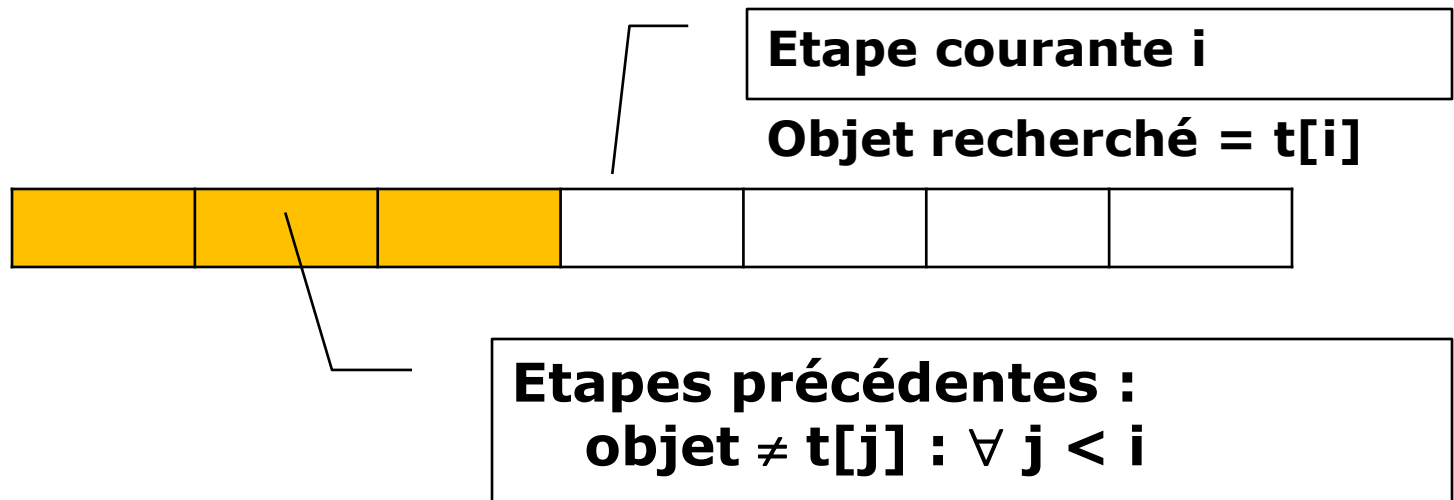
```
void permuter(float T[], int i, int j)
{float c=T[i];  T[i]=T[j];T[j]=c; }
```

```
void TriInsertion(float T[], int N)
{int i,j,Posmin;
for(i=0;i<N;i++)
{ j=i-1;
  while ((j>=0) && (T[j]>T[j+1]))
    {permuter(T,j,j+1); j=j-1; }
}
```

3	7	2	6	5	1	4
3	7	2	6	5	1	4
2	3	7	6	5	1	4
2	3	6	7	5	1	4
2	3	5	6	7	1	4
1	2	3	5	6	7	4
1	2	3	4	5	6	7

Recherche séquentielle (linéaire)

- Principe : comparer les uns après les autres tous les éléments du tableau avec l'objet recherché. Arrêt si :
- l'objet a été trouvé
 - tous les éléments ont été passés en revue et l'objet n'a pas été trouvé



Recherche séquentielle (linéaire)

Fonction RechSequentielle:

Entrées: -T un tableau de taille N

- ObjRecherche: l'élément à chercher

Sortie: La position de la première occurrence de ObjRecherche dans T
s'il existe dans T, sinon la sortie = -1

Début

Pour i=1 à N faire:

Si (t[i] == ObjRecherche) alors:

Retourner(i)

Fin Si

Retourner (-1);

Fin

Recherche séquentielle (linéaire)

```
int RechSequentielle (float T[], float ObjRecherche , int N)
{ int i;
  for (i=0 ; i<N ; i++)
    if(T[i]== ObjRecherche) {return(i);}
  return(-1);
}
```

Recherche dichotomique

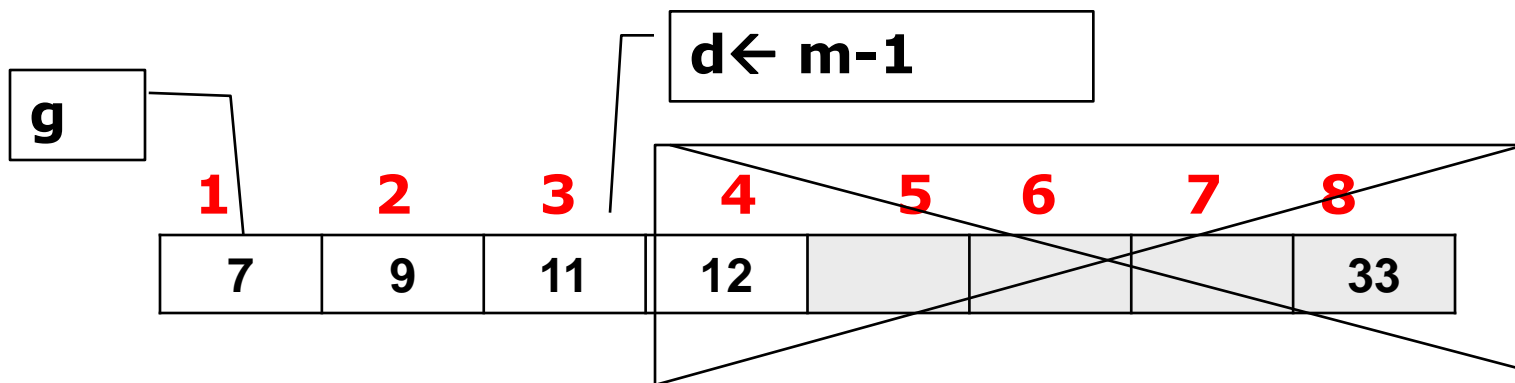
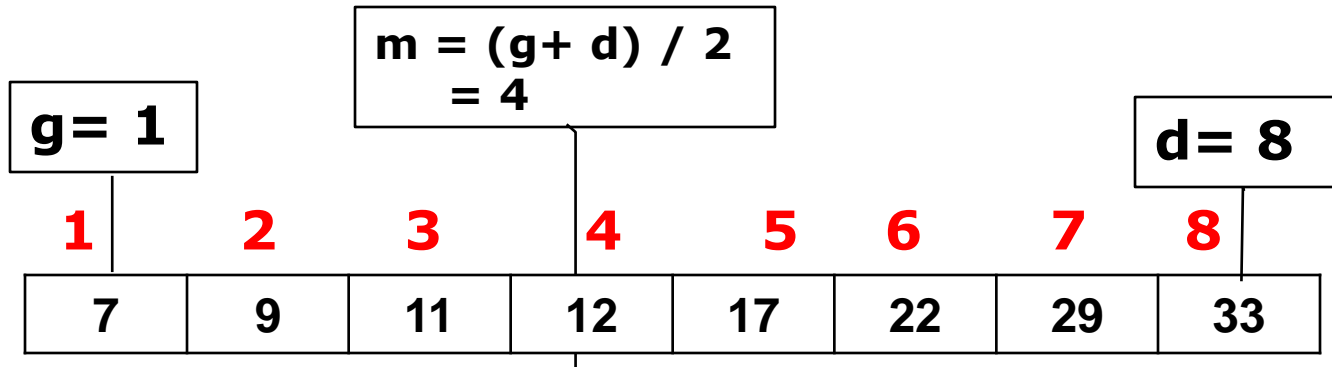
ATTENTION : il n'est applicable que sur un tableau trié.

A chaque étape :

- ➔ découpage du tableau en deux sous-tableau à l'aide d'un indice milieu (tableau à gauche) et (tableau à droite)
- ➔ comparaison de la valeur située à l'indice milieu et de l'objet recherché,
 1. si l'objet est égal à la valeur $T[\text{milieu}]$ alors il est trouvé !
 2. si l'objet est supérieur à la valeur $T[\text{milieu}]$ relancer la recherche avec le tableau à droite,
 3. sinon relancer la recherche avec le tableau à gauche.

Exemple:

ObjRecherche = 9



Recherche dichotomique

Fonction RechDichotomique:

Entrées: -T un tableau **Trié** de taille N
- ObjRecherche: l'élément à chercher

Sortie: Une position de ObjRecherche dans T s'il existe, sinon sortie = -1

Début

$g \leftarrow 1; d \leftarrow N$

Tant que $(g < d)$ faire:

$m \leftarrow (g+d)/2$

Si $T[m] = \text{ObjRecherche}$ alors Retourner(m)

Sinon Si $T[m] < \text{ObjRecherche}$ alors $g \leftarrow m+1$

Sinon $d \leftarrow m+1$

FinSi

FinSi

Fin tant Que

Retourner (-1);

Fin

Recherche dichotomique

```
int RechDico(float T[], float ObjRecherche, int N)
{int i,g=0,d=N-1,m;
  while(g<d)
  {m=(g+d)/2;
   if(T[m]== ObjRecherche)
    {return(m);}
   else if(T[m]<ObjRecherche)
    {g=m+1;}
   else {d=m-1;}
  }
  return(-1);
}
```

Question: Pour un tableau T trié de taille N, Quelle est la méthode de recherche la plus rapide ? Séquentielle ou dichotomique ?

Chaines de caractères

1) Une chaîne de caractères est traitée comme un tableau de caractères qui se termine par '\0'.

2) Initialisation d'une chaîne : CHAINE[] = "Hello";

**3) Lecture d'une chaîne de caractère du clavier : char ch[20] ;
scanf("%s", ch); il n'y a pas & avant ch.**

Ou aussi gets(ch) ;

4) Affichage sur l'écran : puts(ch) ; ou printf(" %s ",ch) ;

5) Quelques fonctions de la bibliothèque string.h :

strlen(chaine) :fournit la longueur de la chaîne sans compter le '\0'

strcat(chaine1, chaine2) : ajoute chaine2 à la fin de chaine1

strcmp(chaine1, chaine2) : compare chaine1 et chaine2 selon ASCII

strcpy (chaine1, chaine2) : recopie la chaîne chaine2 dans chaine1.

Allocation dynamique de la mémoire

Problème $\xrightarrow{\text{solution}}$ Algorithme = Données + Traitement

“Comment Organiser au Mieux l’Information dans un Programme ?”

scalaires

Char, int, float, ...

Tableaux statiques

```
int tab[10];
```

Matrices statiques

```
float Mat[10][10];
```

Nouveau!

Tableaux dynamiques

```
Int* tab=(int*)  
malloc(10*sizeof(int));
```

Allocation dynamique de la mémoire

L'objectif ici est de réserver un espace mémoire dont la taille est déterminée par l'utilisateur selon ses besoins.

Exemple: On veut réaliser un programme pour calculer la moyenne générale des notes d'une classe. Ces notes sont stockées dans un tableau T de taille N.

Problèmes:

- ***T est insuffisant pour des classes dont l'effectif dépasse N***
- ***Une partie de T n'est pas utilisée lorsque l'effectif $\ll N$***



Allocation dynamique de la mémoire

Le **programmeur** peut spécifier quand une variable doit être **allouée**:

```
void * malloc() ;
```

Le **programmeur** peut spécifier aussi quand une variable doit être **libérée**:

```
void free() ;
```

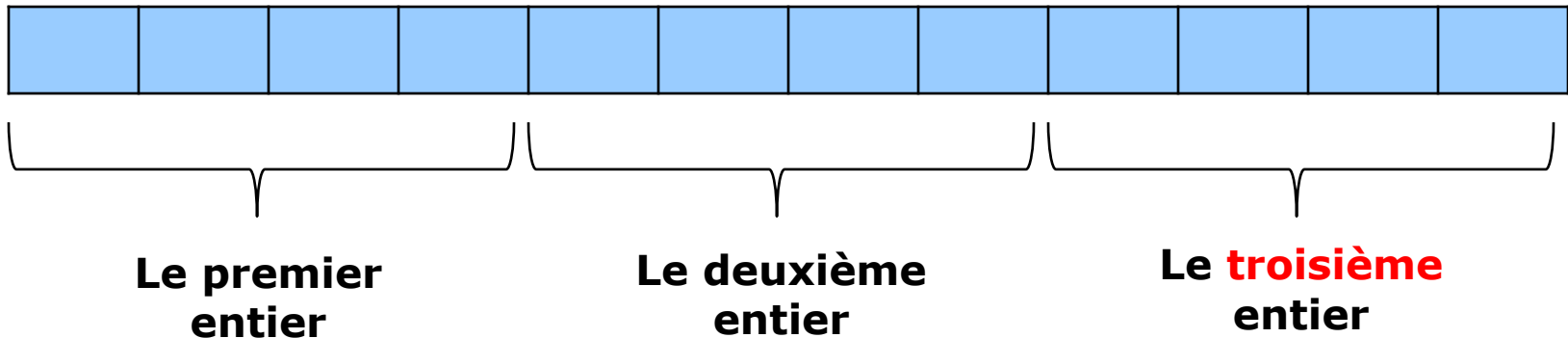


Allocation dynamique de la mémoire

Allocation mémoire pour un tableau contenant 3 nombre réels:

```

#include <stdlib.h>
int*p=(int*)malloc(3*sizeof(int));
      ⇔
int*p=(int*)malloc(12);
  
```



Libération mémoire du pointeur *p*:

```

free(p); // après l'exécution de cette instruction, la
zone mémoire contenant les douze octets (les 3
entiers) dont la première a l'adresse p n'est plus réservée
et elle peut être allouée par un autre programme.
  
```

Problème $\xrightarrow{\text{solution}}$ Algorithme = Données + Traitement

“Comment Organiser au Mieux l’Information dans un Programme ?”

Nouveau!

scalaires

Char, int, float, ...

Matrices statiques

float Mat[10][10];

Tableaux statiques

int tab[10];

Tableaux dynamiques

Structures

```
struct point  
{char NomPoint;  
float x;  
float y;};
```

Le Type Structure

Nous pouvons créer nos propres types puis déclarer des variables ou des tableaux de ce type, ainsi on peut avoir le type Livre, Etudiant, Matière.

2. Exemples de déclaration d'une structure:

```
struct Livre {  
    char Titre[20];  
    char Auteur[20];  
    int AnEdition;  
    float Prix; };
```

```
struct monome  
{  
    int degre;  
    float Coefficient;  
};
```

```
struct Etudiant{  
    int CNE;  
    char NomComple[40];  
    float MoyenneGenerale;  
};
```

Création de variables de type structure

Exemples: Struct Livre X, Y;

X

Titre:	"Mathématiques "
Auteur:	" A. Amrani "
AnEdition:	2013
Prix:	300,00

Y

Titre:	" Informatique "
Auteur:	" M. Saadi"
AnEdition:	2014
Prix:	400,00

Manipulation globale et élémentaire des structures

L'affectation s'applique aussi à des variables de type structures, par exemple l'instruction $X=Y$ (deux variables de types struct livre) permet de copier le contenu de tous les champs de Y dans X.

```

X=Y ⇔ strcpy(X.Titre, Y.Titre);
      strcpy(X.Auteur, Y.Auteur);
      X.AnEdition=Y.AnEdition;
      X.Prix=Y.Prix;
  
```

X

Titre:	" Informatique "
Auteur:	" M. Saadi"
AnEdition:	2014
Prix:	400,00

Y

Titre:	" Informatique "
Auteur:	" M. Saadi"
AnEdition:	2014
Prix:	400,00

Pointeur sur une structure

Soit P un pointeur sur la structure X précédente.
 P peut être déclarée comme suit:
 struct livre * P; P=&X;
 Le contenu de l'adresse P est toute la structure X.

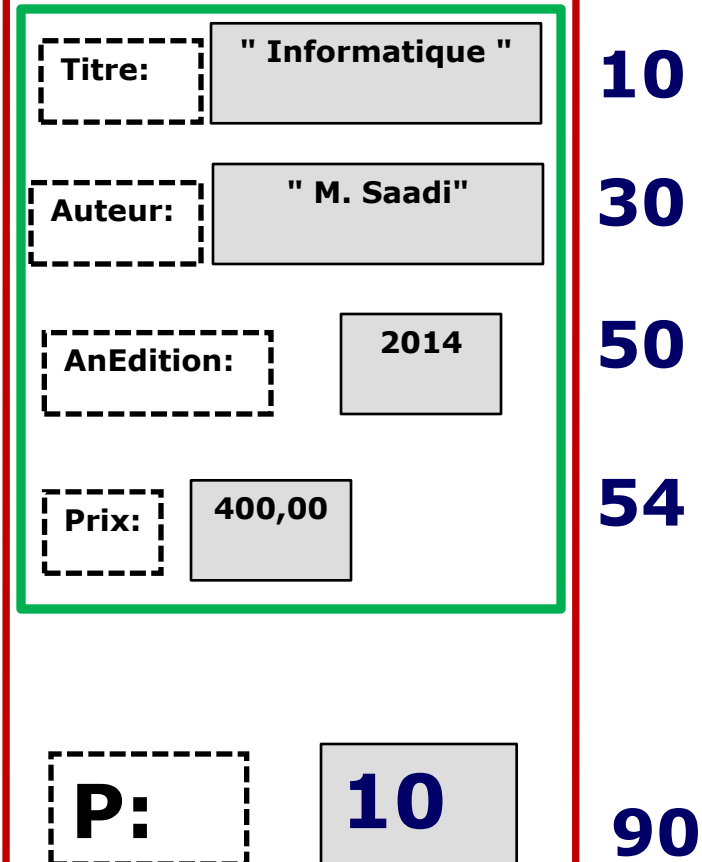
Pour afficher ce contenu on peut écrire:
 Printf(" %s", (*P).Titre);
 Printf(" %s", (*P).Auteur);
 Printf(" %d", (*P).AnEdition);



//Printf(" %f", (*P).Prix);
 // Printf(" %s", P->Titre);
 //Printf(" %s", P->Auteur);
 //Printf(" %d", P-> AnEdition);
 //Printf(" %f", P->Prix);

Mémoire RAM

Adresses
mémoire



Structure contenant une autre structure

A

Soit la structure date suivante:

```
struct date {int jour; int mois; int annee;}
```

On peut créer la structure étudiant suivante:

```
struct etudiant {char NomComple[40];
                 struct date D_Inscription_Fac;
                 }
```

```
struct etudiant A;
strcpy(A. NomComple, « Aalim Amine »);
A. D_Inscription_Fac.jour=12;
A. D_Inscription_Fac.mois=9;
A. D_Inscription_Fac. annee=2013;
```

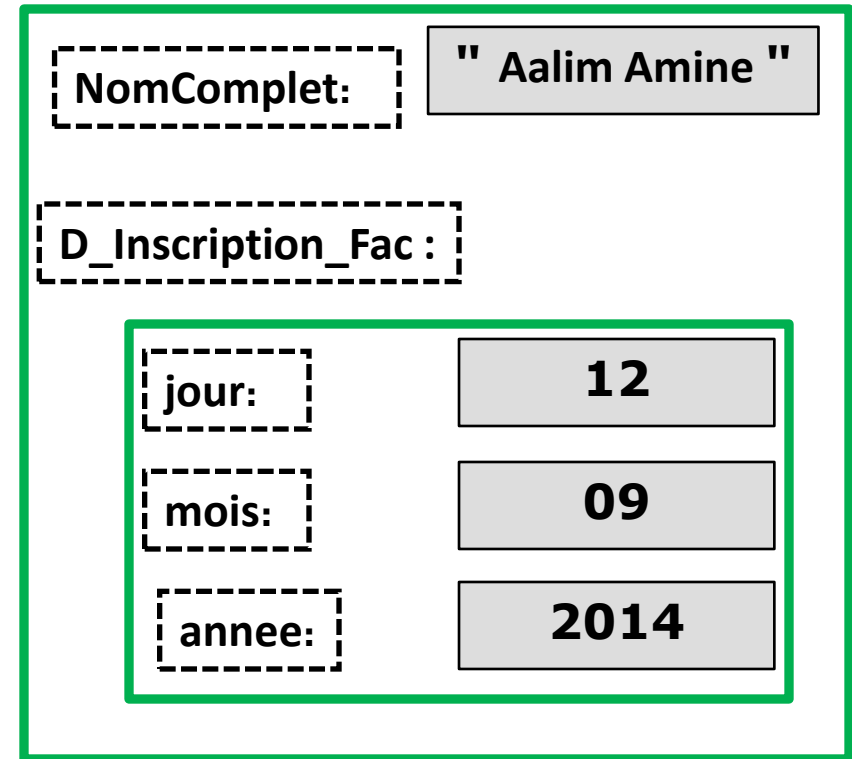


Tableau de structures

```
struct point { char nom ;    int x ;    int y ;    } ;  
struct point courbe [20] ;
```

La structure point sert à représenter un point d'un plan. un point est identifié par son nom (caractère) et ses deux coordonnées.

courbe représente un tableau de 50 éléments du type point.

-Si i est un entier <20, la notation : courbe[i].nom représente le nom du point de rang i du tableau courbe. Il s'agit donc d'une valeur de type char.

- la notation : courbe[i].x

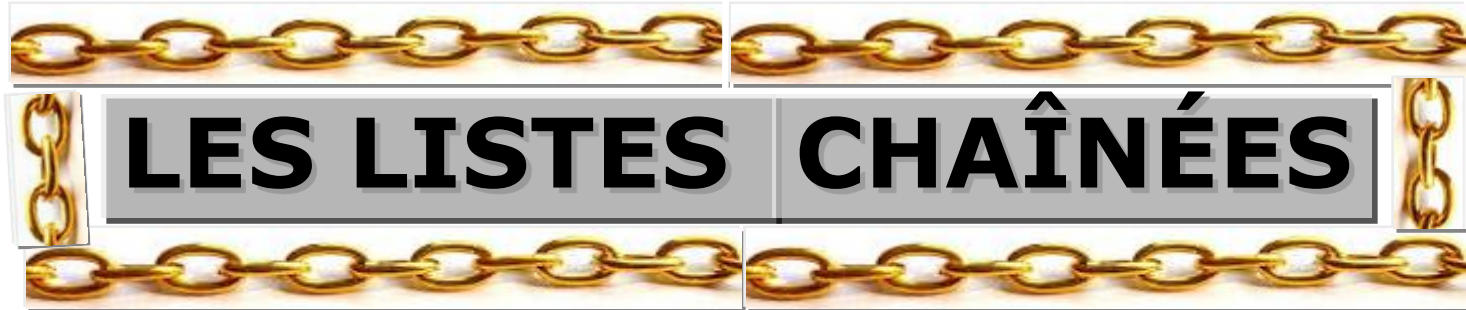
désigne la valeur du champ x de l'élément de rang i du tableau courbe.

- courbe[4] représente la structure de type point correspondant au cinquième élément du tableau courbe.

- courbe est un identificateur de tableau : désigne son adresse de début.

Voici un exemple d'initialisation de la variable courbe, lors de sa déclaration :

```
struct point courbe[3]= { {'A', 10, 25}, {'M', 12, 28}, {'P', 18,2} };
```



Problème $\xrightarrow{\text{solution}}$ **Algorithme = Données + Traitement**

“Comment Organiser au Mieux l’Information dans un Programme ?”

Tableaux
Statiques
Et dynamiques

Structures

Matrices

scalaires

Nouveau!

Listes
Chainées

Avantages et inconvénients des structures séquentielles

Exemple: Dans un algorithme, un polynôme à une seule variable X peut être représenté par plusieurs structures de données.

Le polynôme $P(X) = 10 + 14X^3 + 6X^9 + 4X^{11}$ peut être représenté par le tableau à une dimension comme ci-dessous:

0	1	2	3	4	5	6	7	8	9	10	11	n
10	0	0	14	0	0	0	0	0	6	0	4	0

Le polynôme $P(X)$ peut être représenté aussi par un tableau à deux dimensions :

		j →						
		0	1	2	3	n
i ↓	0	0	3	9	11	n
	1	10	14	6	4	0

Perte de mémoire :

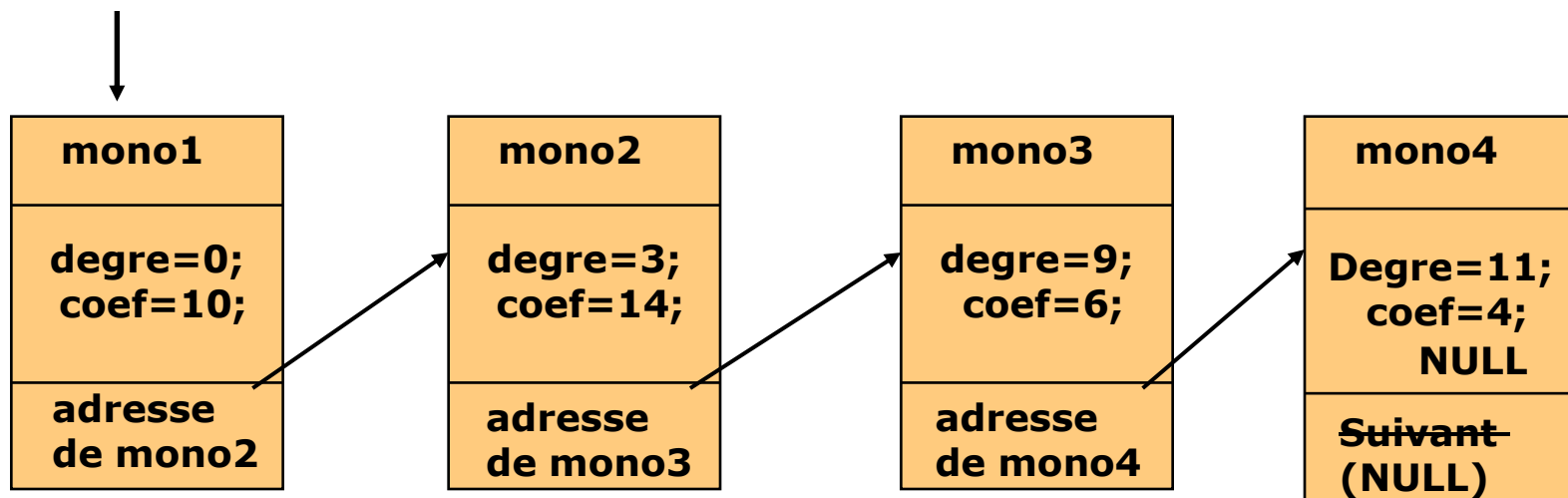
- plusieurs cases creuses,
- on général le nombre n est inconnu au départ, ce qui conduit au choix d'une limite $n = n_{\max}$, donc un espace mémoire alloué peut ne pas être utilisé ou insuffisant !

Avantages et inconvénients des structures séquentielles

- **Avantage:** simples, accès direct (adresse d'un élément peut être calculée à partir de son indice), ...
- **Inconvénients:**
 - **Les tableaux statiques** ont une taille fixe → gaspillage de mémoire !
 - Occupent un espace mémoire contiguë : Maintenance (ajout, suppression, déplacement) coûteuse en termes de temps!

Le polynôme $P(X) = 10 + 14X^3 + 6X^9 + 4X^{11}$ précédent peut être représenté par un ensemble d'enregistrements (le type struct) liés entre eux (chaînés) comme ci-dessous:

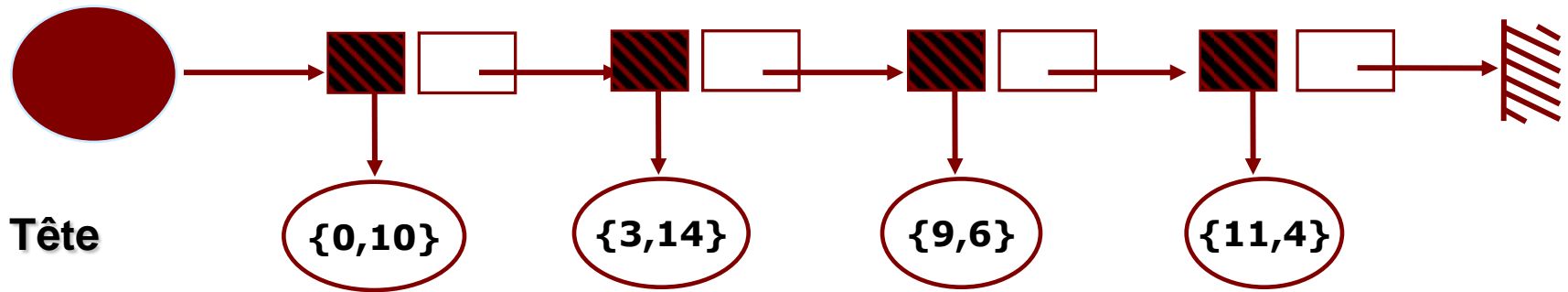
Premier= dresse
de mono1



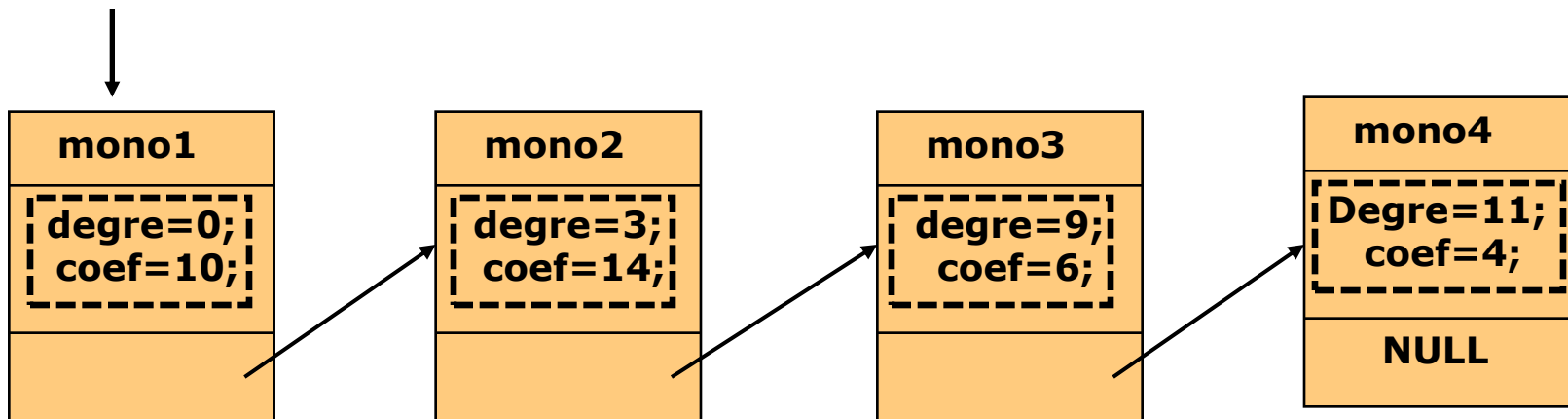
Définition d'une liste chaînée

- ➔ Une structure autoréférentielle correspond à une structure dont au moins un des champs contient un pointeur vers une structure de même type.
- ➔ Une liste chaînée est une structure de données autoréférentielle représentant une collection ordonnée et de taille arbitraire d'éléments (appelés noeuds) de même type.
- ➔ Dans une liste chaînée, les noeuds sont ordonnés de façon linéaire (c-à-d chaque élément nous donne une information sur le lieu (l'adresse mémoire) de son suivant.
- ➔ À la différence d'un tableau, où l'ordre **séquentiel** est déterminé par les indices du tableau, l'ordre dans une liste chaînée est déterminé par un pointeur dans chacun des noeuds.
- ➔ Lorsque la structure contient des données, un pointeur vers la structure suivante, et un pointeur vers la structure précédente on parle de liste chaînée double (ou liste doublement chaînée).

Le polynôme $P(X)$ précédent peut être représenté par la liste chaînée suivante:



premier



Caractéristiques d'une liste simplement chaînée

- **Avantages:** Taille variable; l'ajout ou le retrait d'un nœud dans la liste ne nécessite le déplacement d'aucune donnée en mémoire; Les données n'ont pas besoin d'être contigües en mémoire.
- **Inconvénient:** pour consulter un élément, il faut consulter **tous** ses prédécesseurs.
- **Quelques opérations sur les listes chaînées:**
 - Création
 - Accès à un élément (au début, à la fin et ailleurs)
 - Ajout d'un élément (au début, à la fin et ailleurs)
 - Suppression d'un élément (au début, à la fin et ailleurs)
 - Recherche d'un élément dans une liste chaînée
 - Calcul de la longueur d'une liste chaînée
 - Affichage du contenu d'une liste chaînée
 - Inverser une liste chaînée
 - Tri selon une composante ou plusieurs composantes **des nœuds**.
 -

Étapes de création d'une liste chaînée vide:

- 1) On déclare la structure des nœuds
- 2) La redéfinition du type struct nœud* par typedef si nécessaire
- 3) La création de la tête de la liste chaînée.
- 4) Initialisation de la tête de la liste chaînée par NULL.

En langage C:

```

struct donnees
{.....
  ..... }
  
```

```

struct noeud
{struct donnees D;
  struct noeud*suivant;
};
  
```

```

typedef struct noeud* ptr_noeud;
  
```

```

ptr_noeud TeteDeListe=NULL;
  
```

EN algorithmique:

Type donnees: enregistrement {.....}

Type noeud=

Enregistrement{

var D: donnees;

var suivant: pointeur sur noeud}

/* pointeur sur noeud $\Leftrightarrow \wedge$ noeud */

Type ptr_noeud= pointeur sur noeud

début

var T: ptr_noeud;

T←NULL

fin

Exemple: EN algorithmique:

```
Type donnees_monome=
    enregistrement {var degre: entier;
                    var coef: réel;}

Type noeud_monome=
    enregistrement{
        var D: donnees_monome;
        var suivant: pointeur sur noeud_monome
    }
    /* pointeur sur un noeud peut être noté ^noeud */

Type ptr_noeud_monome= ^noeud_monome

début
    var T: ptr_noeud_monome;
    T ← NULL
fin
```

Exemple1: (EN langage C)

```
struct donnees_monome
    {int degre;   float coef;};

struct noeud_monome
{struct donnees _monome D;
 struct noeud_monome *suivant;
};

typedef struct   noeud_monome * mon;

typedef struct donnees_monome DM;

main()
{ mon TeteDeListe=NULL;
  .....

}
```

Création d'une liste chaînée

Exemple2: (EN langage C)

```
struct donnees_etudiant{
    char nom[20];
    char Prenom[20];
    char CIN[20];
    int Annee_Naissance;
    float MG;};

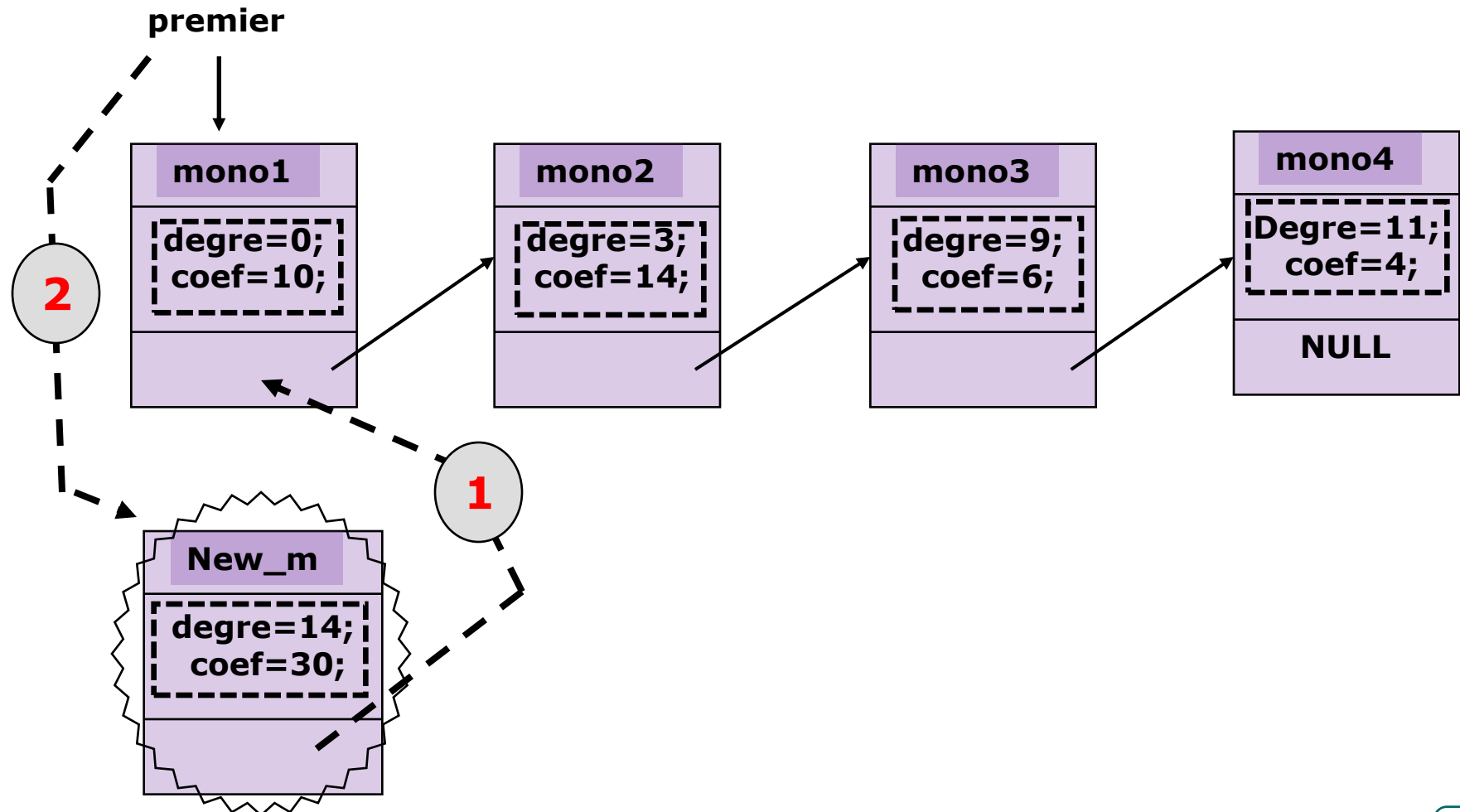
struct noeud_etudiant
{struct donnees_etudiant D;
 struct noeud_etudiant *suivant;
};
```

Exemple3: (EN langage C)

```
struct donnees_Livre {
    char Titre[20];
    char Auteur[20];
    int AnEdition;
    char MaisonEdition[30];
    float Prix; };

struct noeud_Livre
{struct donnees_Livre D;
 struct noeud_Livre *suivant;
};
```

Ajout d'un nœud en tête d'une liste chaînée:



Ajout d'un nœud en tête d'une liste chaînée:

EN langage C:

```
mon ajouterEnTete(mon Tete, DM d)
{ /* On crée un nouveau élément */
  int taille=sizeof(struct noeud_monome);
  mon nouvelleMonome =(mon) malloc(taille);

  /* On remplit les données du nouveau élément */
  nouvelleMonome->D=d;

  // On affecte l'adresse de l'ancienne tete au champs suivant de la nouvelle
  //tete qui est nouvelleMonome
  nouvelleMonome->suivant = Tete;

  //On retourne la nouvelle tete
  return nouvelleMonome;
}
```


Ajout d'un nœud en tête d'une liste chaînée:

EN algorithmique:

Fonction ajouterEnTete(TeteActuelle: ptr_noeud, contenu: donnees)

Début

var nouveau_noeud : ptr_noeud

nouveau_noeud \leftarrow réserver_mémoire(noeud)

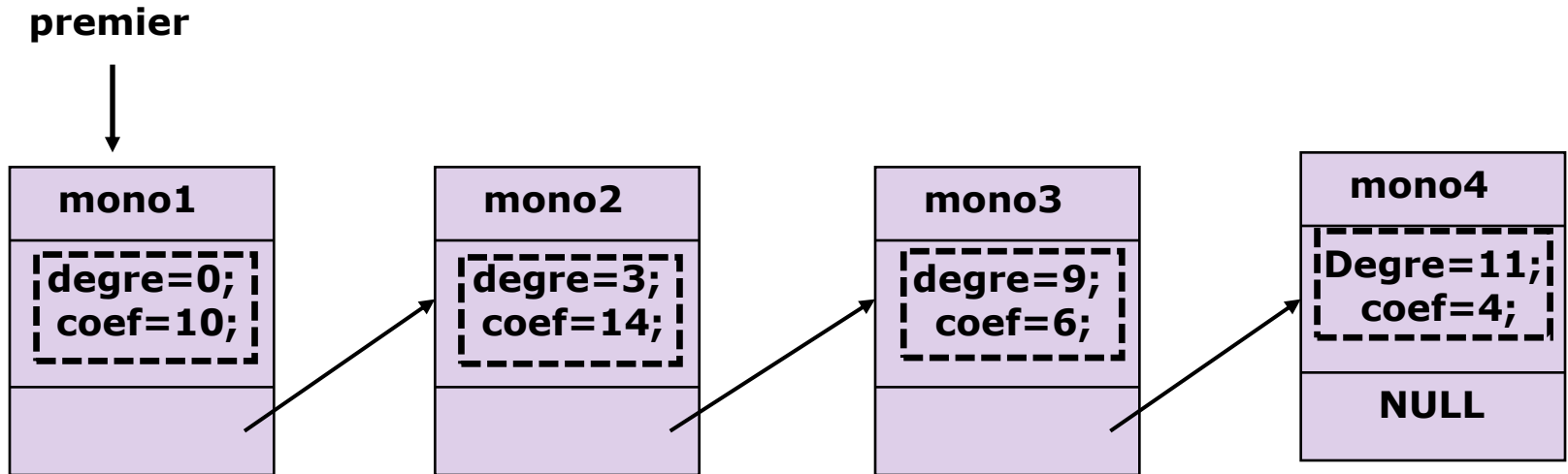
nouveau_noeud \rightarrow D \leftarrow contenu;

nouveau_noeud \rightarrow D \leftarrow TeteActuelle;

ajouterEnTete \leftarrow nouveau_noeud;

Fin

Affichage d'une liste chaînée:



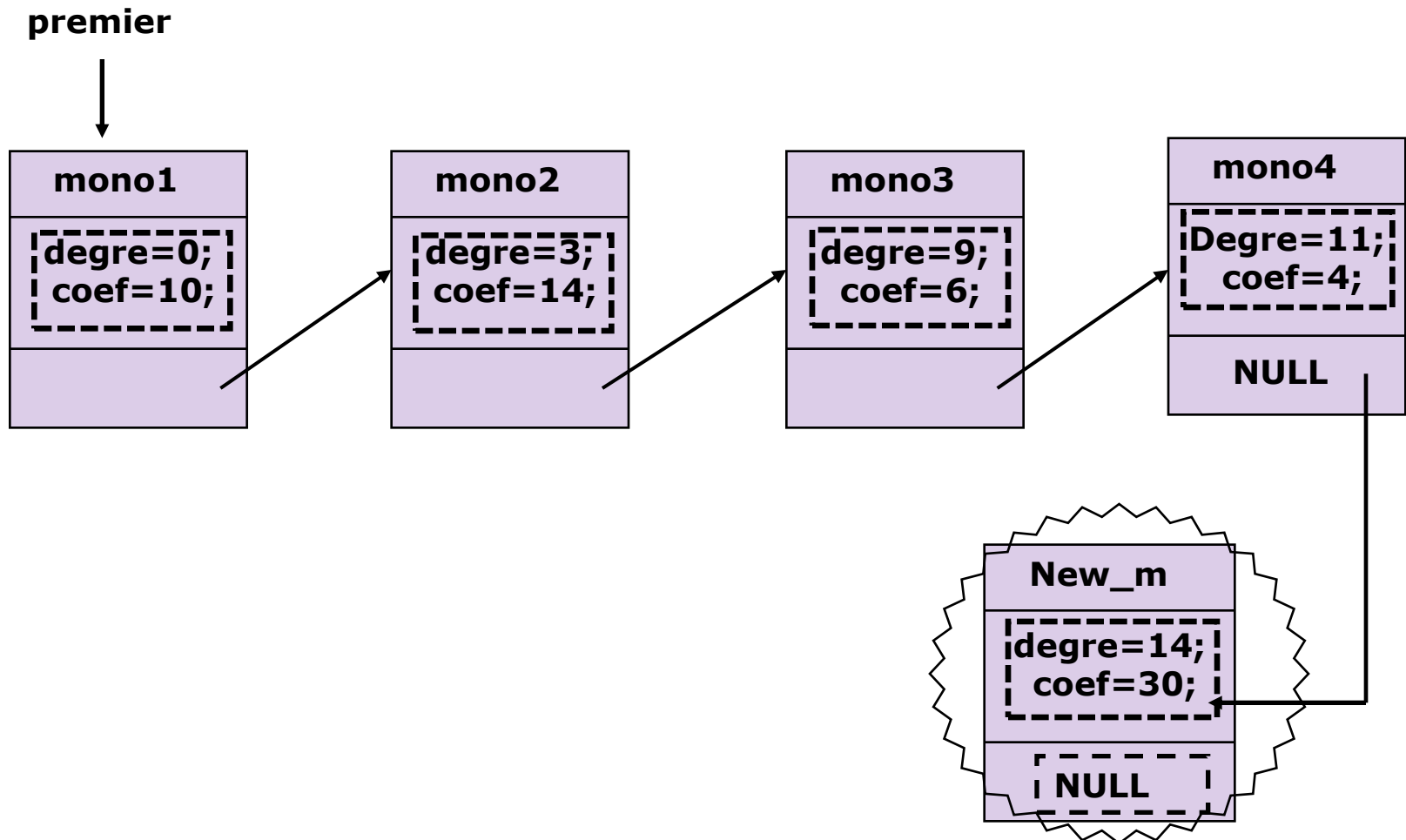
```

void afficherDonnees(struct donnees_monome D)
{ printf("%.2f*X^%d", D.coef, D.degre); }
  
```

```

void afficherListeMonomes(mon Tete)
{ printf("\n\n affichage de la liste : \n");
  mon P=Tete;
  while(P!=NULL)
  { afficherDonnees(P->D);
    P=P->suivant;
  }
}
  
```

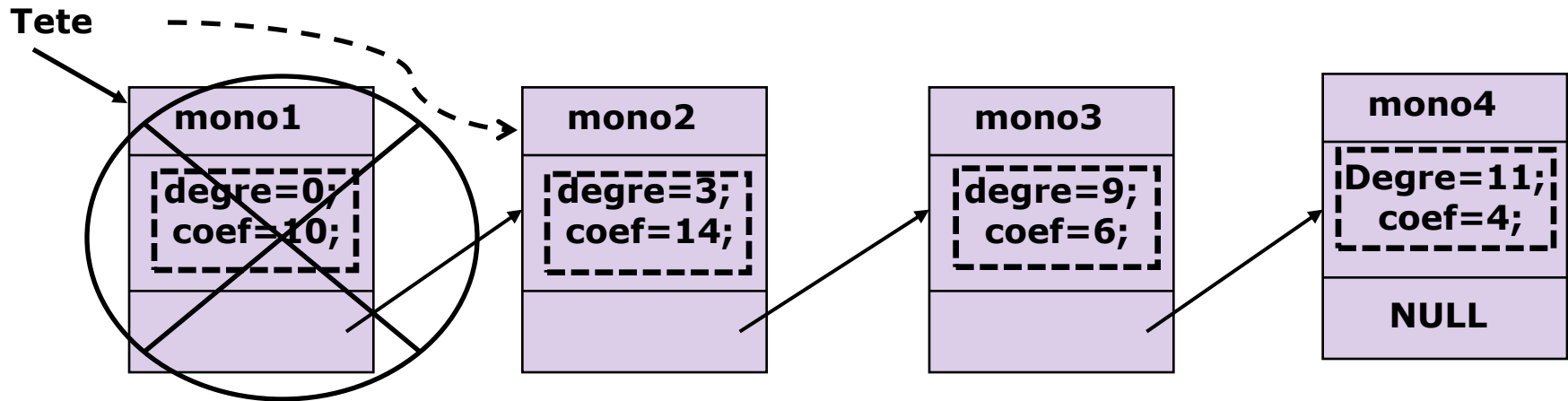
Ajout d'un nœud en fin d'une liste chaînée:



Ajout d'un nœud en fin d'une liste chaînée:

```
mon ajouterEnFin(mon TeteActuelle, DM d)
{ // On crée un nouveau élément
int taille=sizeof(struct noeud_monome);
mon nouvelleMonome=(mon)malloc(taille);
/* On remplit les données du nouveau élément */
nouvelleMonome->D=d;
//On ajoute en fin, donc aucun élément ne va suivre nouvelleMonome-
nouvelleMonome->suivant = NULL;
if(TeteActuelle == NULL) {return nouvelleMonome;}
else
{ //on parcourt la liste à l'aide d'un pointeur temporaire et on relie le
//dernier élément de la liste avec le nouveau élément
mon temp=TeteActuelle;
while(temp->suivant != NULL)
{temp = temp->suivant;}
temp->suivant = nouvelleMonome; return TeteActuelle; }
}
```

Suppression d'un nœud en tête d'une liste chaînée:

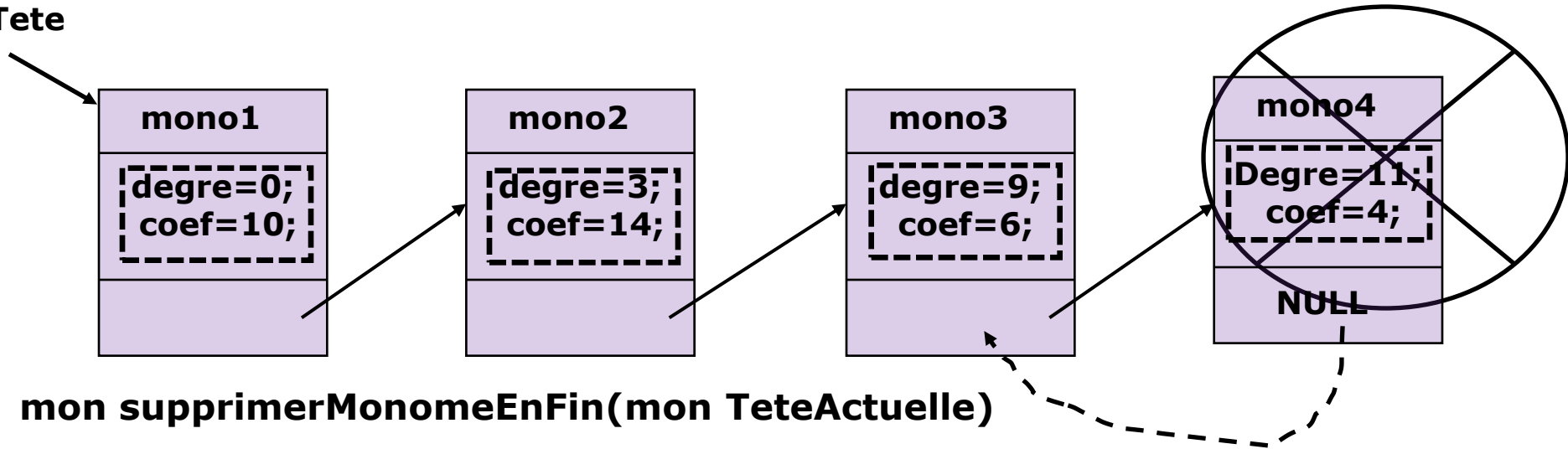


```

mon supprimerMonomeEnTete(mon Tete)
{ if(TeteActuelle == NULL) return NULL;
  else { //Mémoriser la nouvelle tete dans NouvTete
    mon NouvTete = Tete->suivant;
    //On libère le premier élément (le contenu de l'ancienne tete
    free(Tete);
    return NouvTete; /*On retourne la nouvelle tete */}
}
  
```

Suppression d'un nœud en fin d'une liste chaînée:

Tete



mon supprimerMonomeEnFin(mon TeteActuelle)

{if(TeteActuelle == NULL) return NULL;

if(TeteActuelle->suivant == NULL) {free(TeteActuelle); return NULL;}

/* Si la liste contient au moins deux éléments */

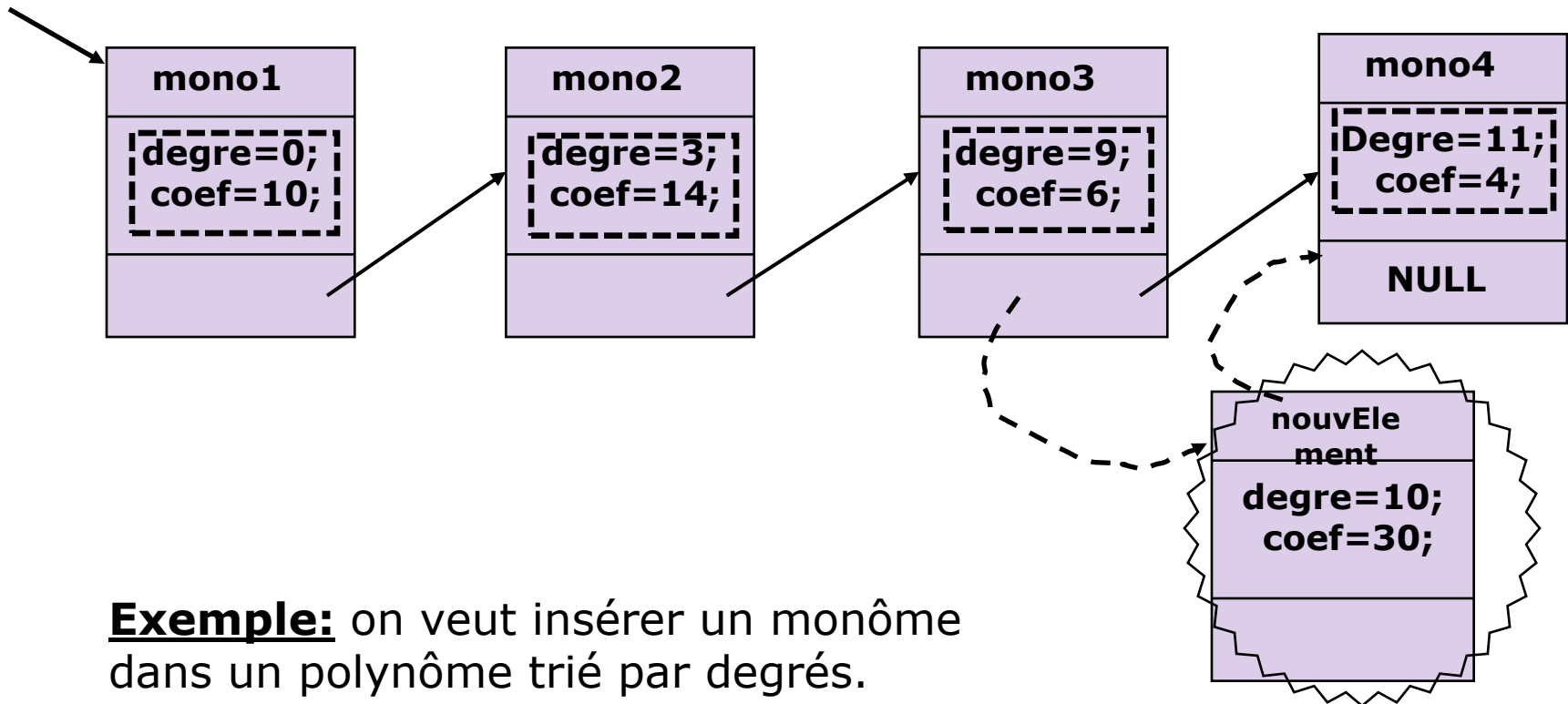
else{ mon tmp = TeteActuelle;

while(tmp->suivant->suivant != NULL)

{tmp = tmp->suivant;}

free(tmp->suivant); tmp->suivant=NULL; return TeteActuelle;}}

Insertion dans une position précise



Exemple: on veut insérer un monôme dans un polynôme trié par degrés.

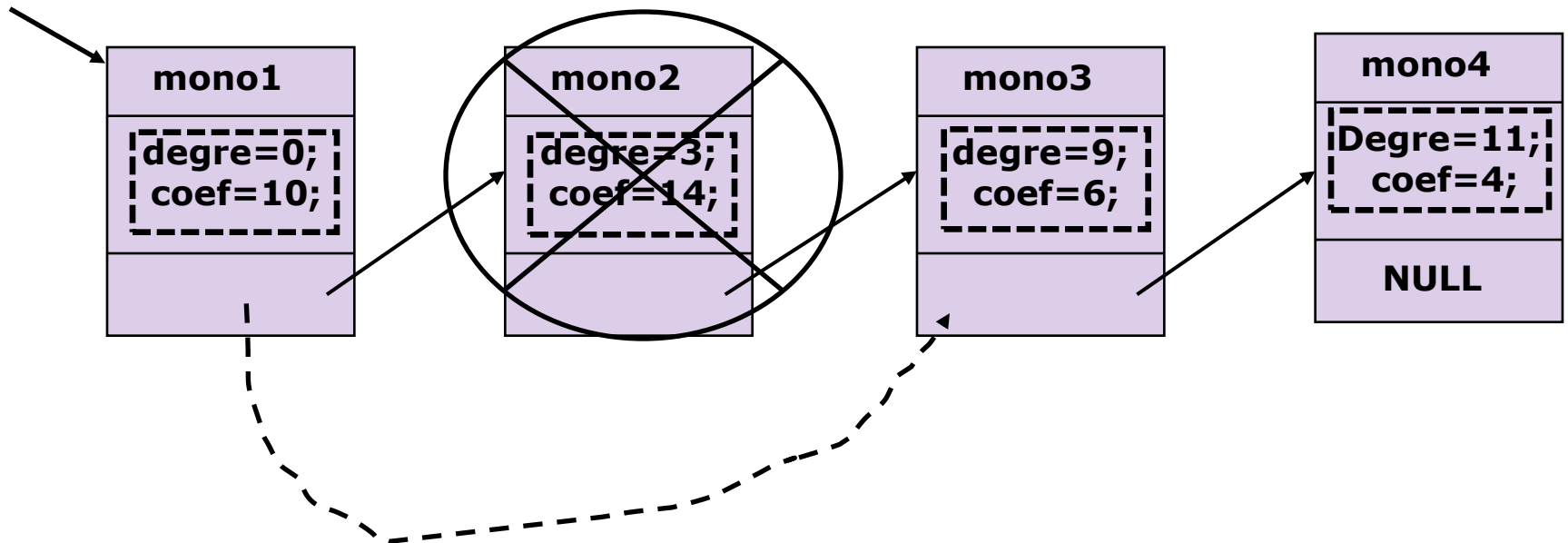
Insertion dans une position quelconque

```
mon inserer(DM d, mon Tete)
{
    mon courant,
    nouvElement=(mon)malloc(sizeof(struct noeud_monome));
    nouvElement->D=d;
    if(Tete==NULL )
    {
        nouvElement->suivant=NULL;
        return(nouvElement);
    }
    if(d.degre<Tete->D.degre)
    {
        nouvElement->suivant=Tete;
        return(nouvElement);
    }
    if(d.degre==Tete->D.degre)
    {
        (Tete->D.coef)+=d.coef; free(nouvElement); return(Tete);
    }
}
```


Insertion dans une position quelconque

```
courant=Tete;  
while(courant->suivant!=NULL && courant->suivant->D.degre<d.degre)  
{courant=courant->suivant;}  
  
if(courant->suivant==NULL)  
{courant->suivant=nouvElement;  
nouvElement->suivant=NULL;return(Tete);}  
else if(courant->suivant->D.degre!=d.degre)  
{nouvElement->suivant=courant->suivant;  
courant->suivant=nouvElement; return(Tete);}  
else {(courant->suivant->D.coef)+=d.coef;  
free(nouvElement);return(Tete);}  
}
```

Suppression d'un élément dans une position quelconque



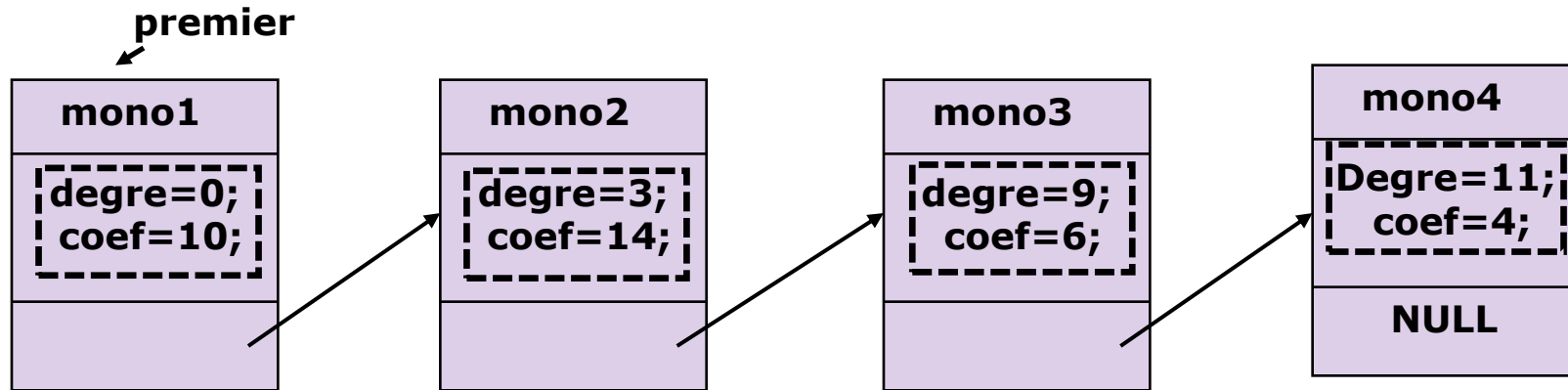
Exemple: on veut supprimer un monôme dans un polynôme.

Suppression d'un élément dans une position quelconque

```

mon supprimerMonome(mon TeteActuelle, int degre)
{
if(TeteActuelle == NULL) return NULL;
mon memorise,tmp = TeteActuelle;
if((TeteActuelle->suivant == NULL) &&(TeteActuelle->D.degre == degre))
    {free(TeteActuelle); return NULL;}
/* Si la liste contient au moins deux éléments */
else if((TeteActuelle->suivant!= NULL)&&(TeteActuelle->D.degre == degre) )
    {memorise=TeteActuelle;
        TeteActuelle=TeteActuelle->suivant;
        free(memorise);return TeteActuelle;}
else{
    while((tmp->suivant!=NULL)&&(tmp->suivant->D.degre!=degre))
    {tmp = tmp->suivant;}
    memorise=tmp->suivant;
    if(tmp->suivant!=NULL)
        {tmp->suivant=tmp->suivant->suivant;
        }
    free(memorise); }
    return TeteActuelle;}
}
    
```

Recherche d'un nœud par sa valeur :



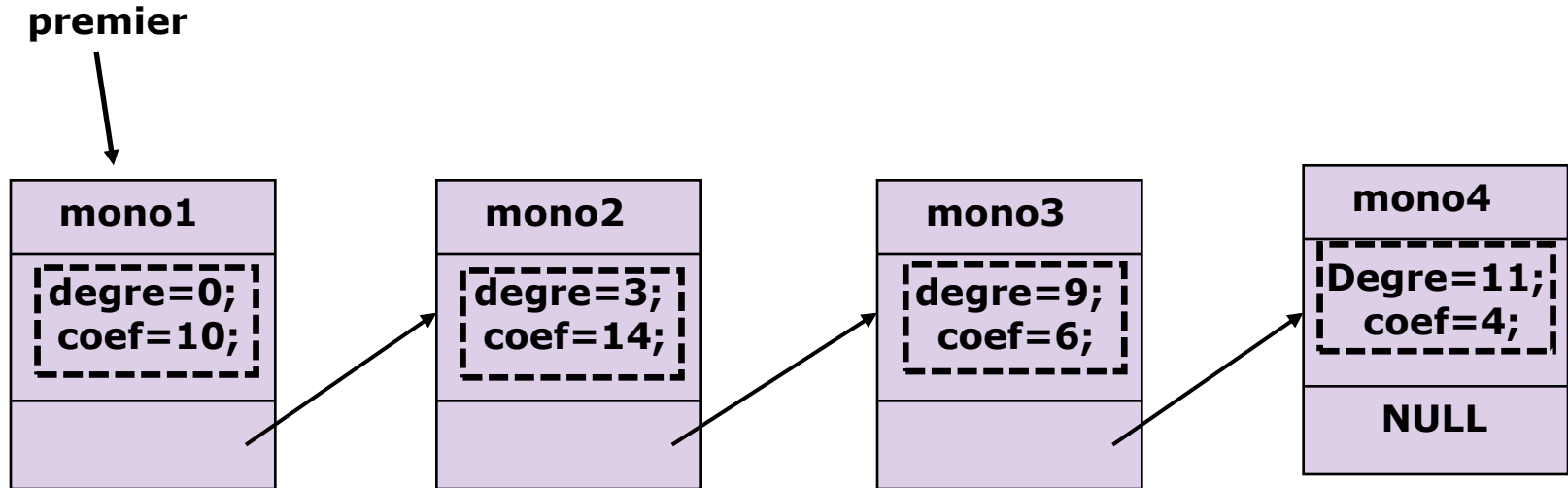
```

int comparaison(struct donnees_monome X,
                struct donnees_monome Y)
{if(X.degree!=Y.degree || X.coef!=Y.coef)  return(1);
 else return(0); }
  
```

```

mon rechercherElement(mon T,struct donnees_monome X)
{ mon tmp=T;
  /* Tant que l'on n'est pas au bout de la liste */
  while(tmp != NULL)
  { if(comparaison(tmp->D,X)==0)
    {//Si l'élément a la valeur recherchée, on renvoie tmp;
     return(tmp); }
    tmp = tmp->suivant; }
  return NULL;}
  
```

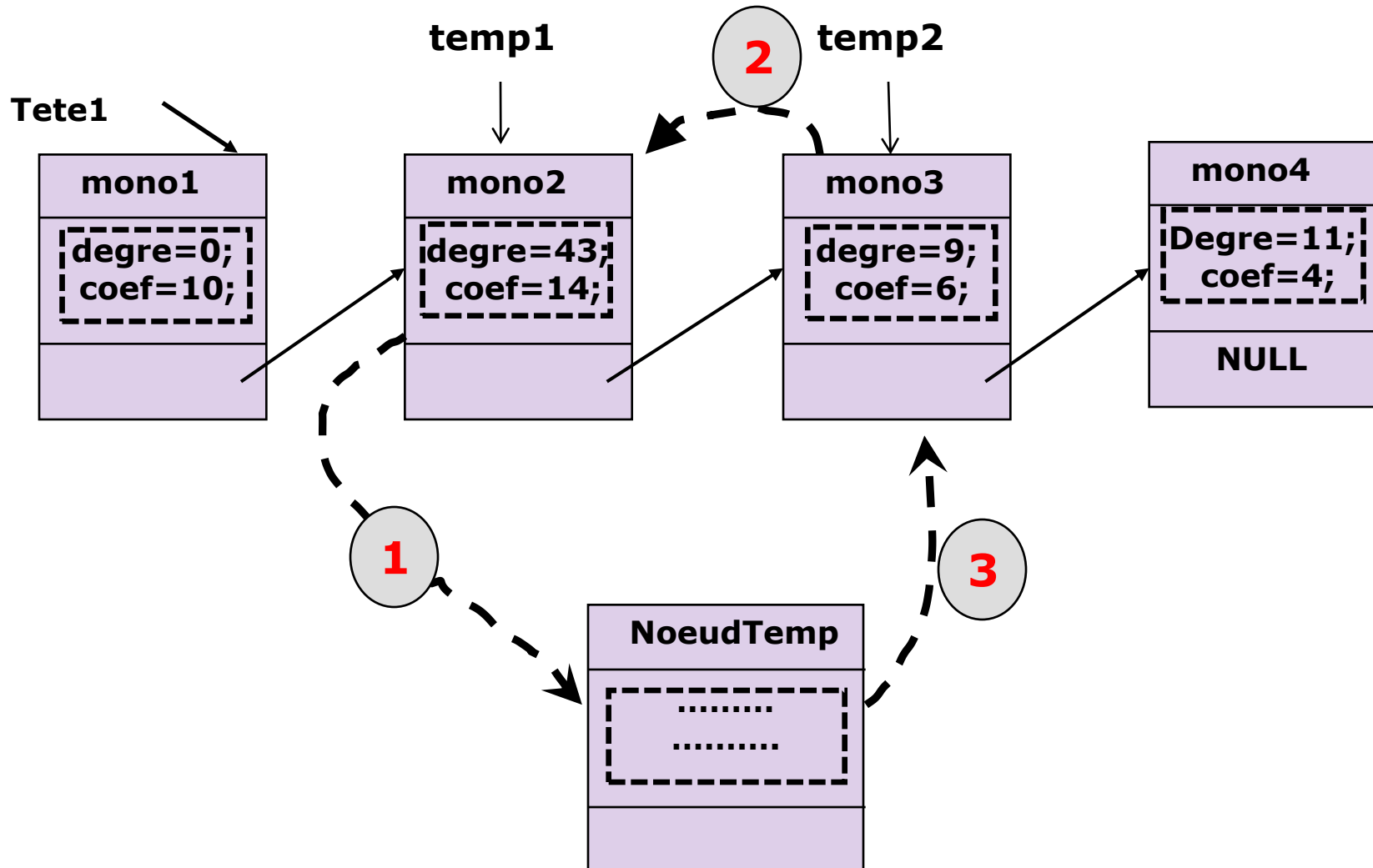
Calcul de la longueur d'une liste chaînée



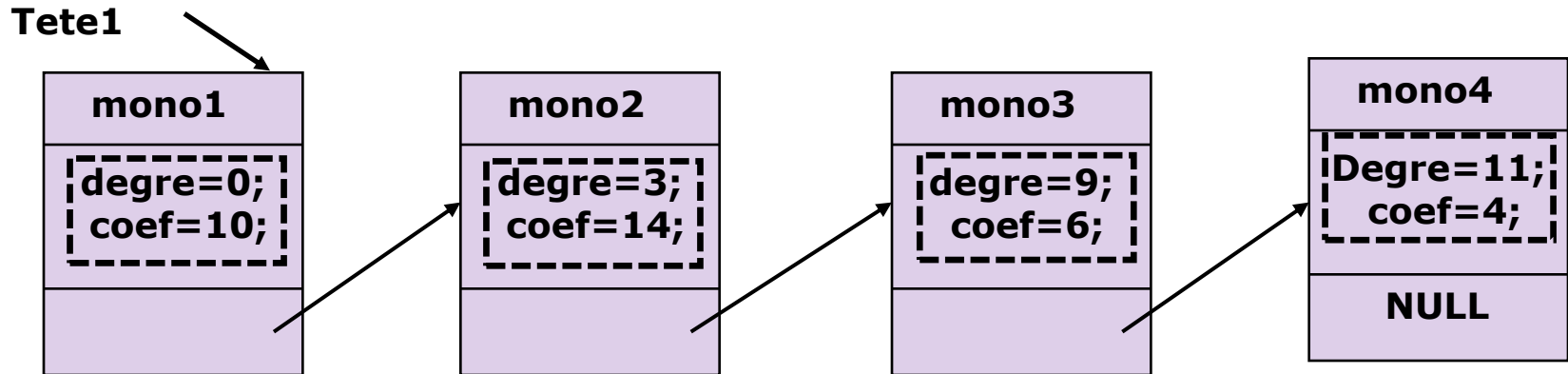
```

int lengthList(mon Tete)
{
  int n=0;
  mon P=Tete;
  while(P!=NULL)
  {
    n++;
    P=P->suivant;
  }
  return(n);
}
  
```

Application du Tri à bulle sur une liste chaînée



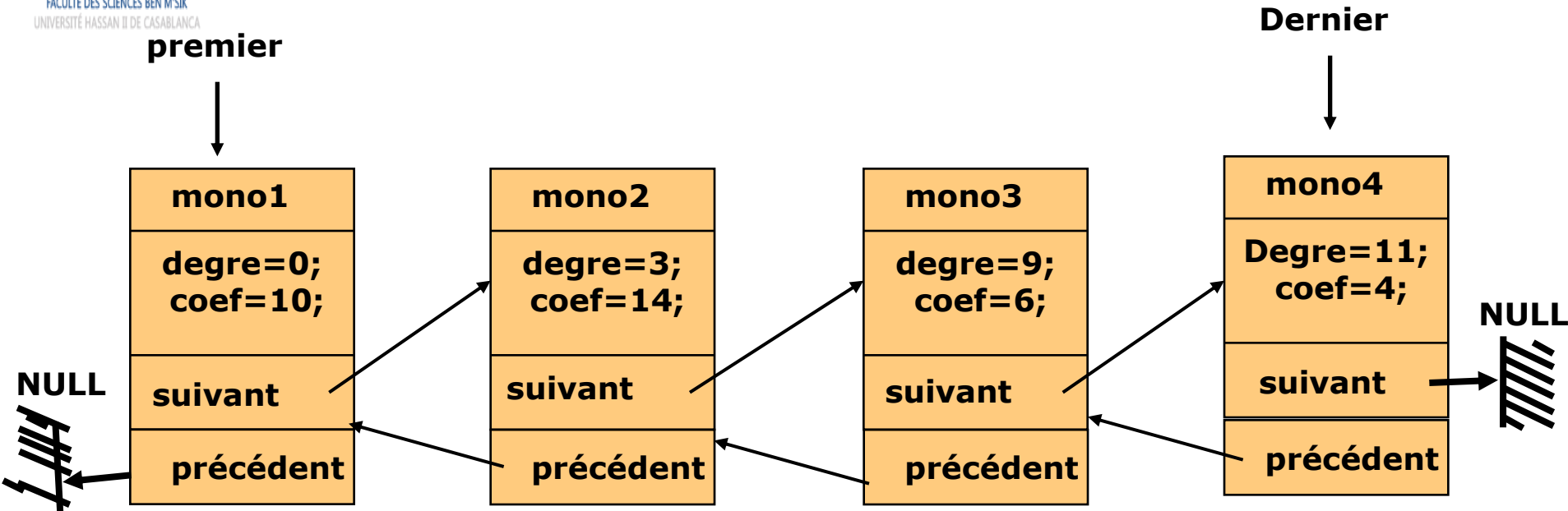
Application du Tri à bulle sur une liste chaînée



```

mon TriBulList(mon Tete1)
{int taille=sizeof(struct noeud_monome);
mon NoeudTemp=(mon)malloc(taille);
mon temp1=Tete1, temp2; int i,j;
int n=lengthList(Tete1);
for(i=1;i<=n-1;i++)
{temp1=Tete1; temp2=temp1->suivant;
while(temp2!=NULL)
{if(temp2->D.degre<temp1->D.degre)
{NoeudTemp->D=temp1->D; temp1->D=temp2->D;
temp2->D=NoeudTemp->D;}
temp1=temp1->suivant; temp2=temp2->suivant;
}}
return(Tete1);}
  
```

Listes Doublement Chainées(LDC)



Avantages d'une liste doublement chaînée par rapport à une liste simplement chaînée:

- 1) Dans une LDC on peut accéder directement au dernier élément, sans parcourir toute la liste comme on le fait dans une LSC (liste simplement chaînée). Donc l'ajout et la suppression du dernier élément sont moins complexe (sont plus rapides).
- 2) On peut parcourir une LDC dans les 2 sens (on peut donc revenir en arrière). Par exemple, On peut vérifier si une liste est symétrique .

Ajout en tête d'une liste doublement chaînée

```
struct donnees_monome{int degre; float coef;};  
struct noeud_monome  
{struct donnees_monome D;  
struct noeud_monome *suivant;  
struct noeud_monome *precedent; };  
  
typedef struct donnees_monome DM;  
typedef struct noeud_monome* mon;  
  
mon ajouterEnTete(mon Tete, DM d)  
{ int taille=sizeof(struct noeud_monome);  
mon nouvelleMonome =(mon) malloc(taille);  
nouvelleMonome->D=d;  
nouvelleMonome->suivant = Tete;  
nouvelleMonome->precedent=NULL;  
if(Tete!=NULL)Tete->precedent=nouvelleMonome;  
return nouvelleMonome;}
```

Ajout en fin d'une liste doublement chaînée

```
mon ajouterEnFin(mon QueActuelle, DM d)
{int taille=sizeof(struct noeud_monome);
mon nouvelleMonome=(mon)malloc(taille);
nouvelleMonome->D=d; nouvelleMonome->suivant = NULL;
if(QueActuelle == NULL)
    {nouvelleMonome->precedent = NULL;
      return nouvelleMonome;}
else
    {QueActuelle->suivant=nouvelleMonome;
      nouvelleMonome->precedent = QueActuelle;
      return nouvelleMonome;
    }
}
```

Ajout dans une position interne d'une liste doublement chaînée

```

void ajouterMonomeIemePosInterne(mon TeteActuelle, DM d,int pos)
{
  if((pos>=longueur(TeteActuelle))||(pos == 1))
    {printf("\n c'est pas un noeud interne");return;}
  int taille=sizeof(struct noeud_monome);
  mon nouvelleMonome=(mon)malloc(taille);
  nouvelleMonome->D=d;
    nouvelleMonome->suivant = NULL;
    int i; mon temp=TeteActuelle;
  for(i=1;i<pos;i++) temp=temp->suivant;
  nouvelleMonome->precedent=temp->precedent;
  nouvelleMonome->suivant=temp;
  (temp->precedent)->suivant=nouvelleMonome;
  temp->precedent=nouvelleMonome;
}
  
```

mon supprimerMonomeEnTete(mon TeteActuelle)

```

{  if(TeteActuelle == NULL)  return NULL;
  else
  {  mon NouvTete = TeteActuelle-&gtsuivant
    if(NouvTete!=NULL) NouvTete-&gtprecedent=NULL;
    free(TeteActuelle);
    return NouvTete;
  }}

```

mon supprimerMonomeEnFin(mon QueActuelle)

```

{  if(QueActuelle == NULL)  return NULL;
  if(QueActuelle-&gtprecedent == NULL)
  {free(QueActuelle); return NULL; }
    mon NouvQue=QueActuelle-&gtprecedent
    NouvQue-&gtsuivant=NULL; free(QueActuelle);
    return NouvQue;
}

```

```

mon supprimerMonomeIemePosInterne(mon TeteActuelle,mon QueActuelle, int pos)
{  if((pos>longueur(TeteActuelle))||(TeteActuelle == NULL))
    return(TeteActuelle);
if((pos==1)||(pos==longueur(TeteActuelle)))
    {printf("\n c'est pas un noeud interne"); return(TeteActuelle);}
int i; mon temp=TeteActuelle;
for(i=1;i<pos;i++) temp=temp->suivant;
(temp->precedent)->suivant=temp->suivant;
    (temp->suivant)->precedent=temp->precedent;
    free(temp);
    return(TeteActuelle);
}
  
```



LES PILES



- La pile est une structure de données, qui permet de stocker les données dans l'ordre **LIFO** (Last In First Out) – (*Dernier Entré Premier Sorti*).
- La récupération des données sera faite dans l'ordre inverse de leur insertion.
- Pour l'implémentation nous pouvons utiliser:
 - 1) Un tableau, l'insertion et la suppression se font à la fin.
 - 2) Une liste simplement ou doublement chaînée. L'insertion se fait toujours à la fin de la liste, la suppression se fait aussi en fin.

Exemple d'implémentation d'une pile de caractères par tableau

```
#define max 100
```

```
struct pile {char vec[max]; int sommet;};
```

```
struct pile p; // p est une pile déclarée globale
```

```
void raz()  
{p.sommet=-1;}
```

```
int estVide()//vide  
{if (p.sommet== -1) return 1; else return 0;}
```

```
void depiler()  
{ p.sommet--; }
```

```
char dernier()  
{ return(p.vec[p.sommet]); }
```

```
void empiler(char x)  
{ p.sommet++; p.vec[p.sommet]=x; }
```

A l'aide de la pile ci-dessus, on peut vérifier par exemple qu'une expression algébrique est valide ou non. (Ex: $4 * (T[8 + M[3] / F[0]]) / (x + 7 * Y)$ est valide)

Exemple d'implémentation d'une pile de caractères par liste chaînée

```

struct caractere
{char c; struct caractere *nxt;};

typedef struct caractere* pc;

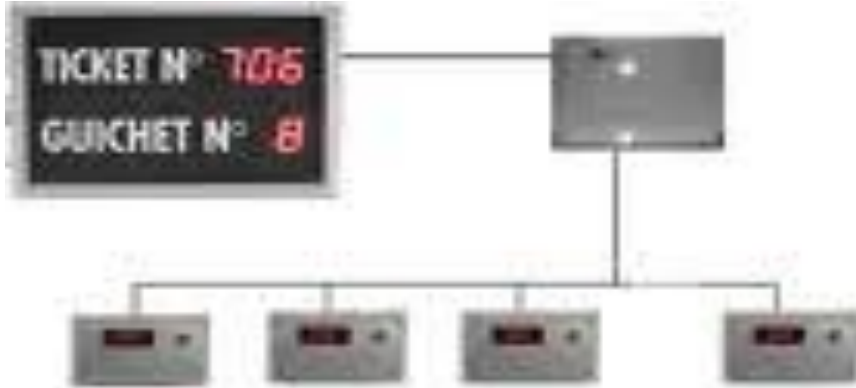
pc Empiler(pc T, char x)
{ //c'est la fonction ajouterEnFin
  .....}

pc Depiler(pc T)
{ //c'est la fonction SupprimerEnFin
  ..... }

char dernier(pc T)
{ if(T != NULL)
    {pc tmp = T;
      while(tmp->nxt!=NULL) tmp=tmp->nxt;
      return tmp->c;
    }
  else
    {return '#';}
}
  
```

Remarque: On peut voir la pile à l'envers et faire l'ajout et la suppression en Tête.

LES FILES



- La **file** (à corriger) est une structure de données, qui permet de stocker les données dans l'ordre **FIFO** (First In First Out) – (*Premier Entré Premier Sorti*).
- La récupération des données sera faite dans l'ordre de leur insertion.
- Pour l'implémentation nous pouvons utiliser:
 - 1) Un tableau, l'insertion en fin et la suppression au début.
 - 2) Une liste simplement ou doublement chaînée. L'insertion se fait toujours à la fin de la liste, la suppression se fait en tête de liste.

Exemple d'implémentation d'une file d'étudiants par tableau

```
#include<string.h>
#define max_lignes 100
#define max_colones 20
struct file
{char vec[max_lignes][max_colones];
  int sommet;};

struct file f;

void raz()
{f.sommet=-1;}

void defiler() // supprimer le premier élément
{int i;
  for(i=0;i<=f.sommet;i++)
    strcpy(f.vec[i],f.vec[i+1]);
  f.sommet=f.sommet-1;
}

void enfiler(char nom[])
{if(f.sommet>=max_lignes-1)
  printf("\n Depassement de capacite de la file");
else
  {f.sommet++; strcpy(f.vec[f.sommet],nom);} }
```

Exemple d'implémentation d'une file de caractères par liste chaînée

```
struct caractere
{char c; struct caractere *nxt;};

typedef struct caractere* pc;

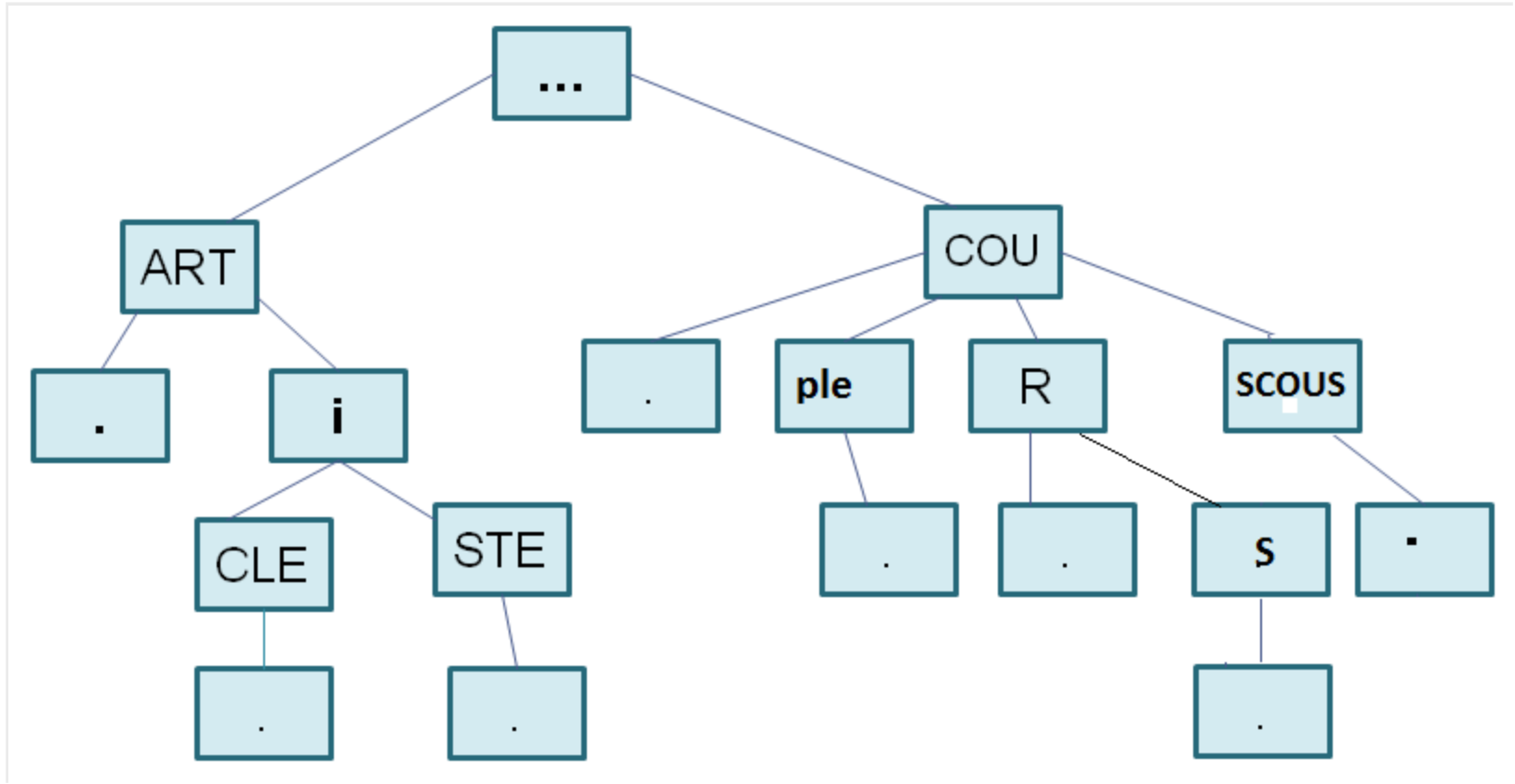
pc Enfiler(pc TeteActuelle, char x)
{ //c'est la fonction ajouterEnFin
  //.....
}

pc Defiler(pc TeteActuelle)
{ //c'est la fonction SupprimerEnTete
  //.....
}
```

Arbres

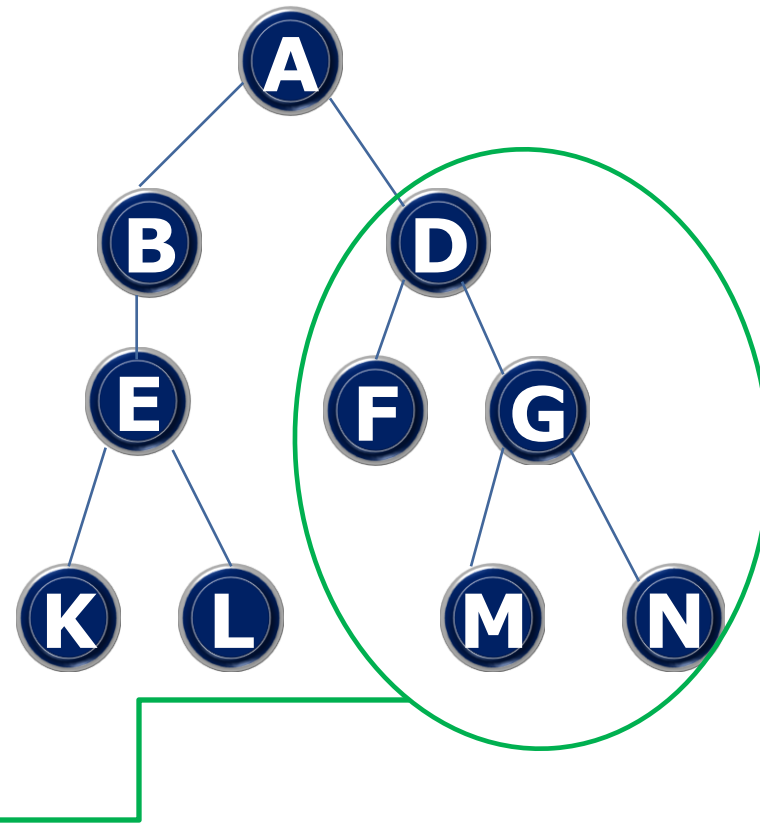


Définition: Une arborescence (ou arbre) est une structure de données en forme d'arbre qui permet d'organiser les données en mémoire ou sur disque, de manière logique et hiérarchisée.



Définitions

- **Père** : prédécesseur direct d'un nœud
 $\text{Père}(F)=D, \text{Père}(M)=G$
- **Fils**: successeurs direct d'un nœud;
 $\text{Fils}(E)=\{K,L\}; \text{Fils}(M)=\{ \}$
- **Feuille**: un nœud sans fils;
 K, L, M, N sont des feuilles
- **Génération**:
 les nœuds d'un même niveau;
 $\text{Niveau } 3 = E, F, G$. E est un frère de F .
- **Branche**: un chemin qui commence par la racine et se termine par une feuille; $[A,D,G,M], [A,B,E,L]$
- **Sous-arbre**: un nœud accompagné de toute sa descendance;



Définitions

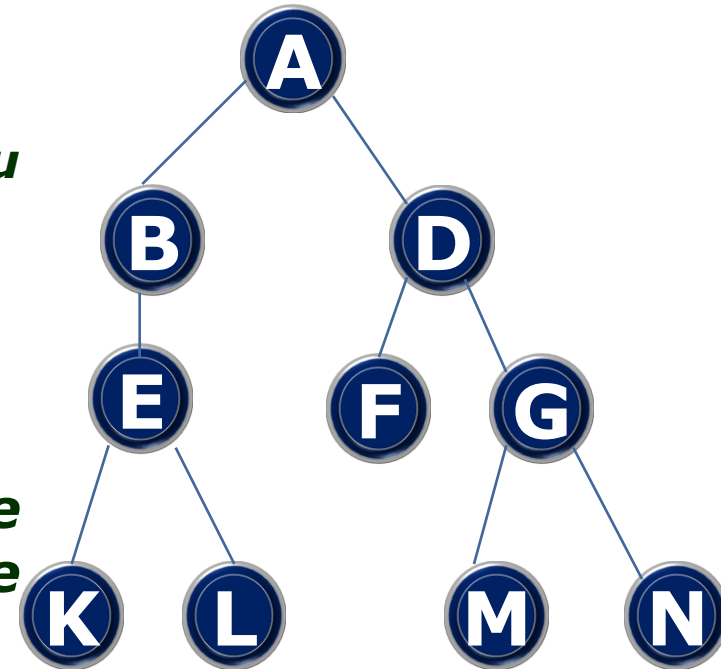
→ **Taille d'un arbre:** le nombre de nœud de l'arbre; ici taille= 10

→ **Profondeur d'un nœud :** *longueur du chemin entre la racine et ce nœud.*
Profondeur(D)=2; Profondeur(L)=4

→ **Hauteur d'un nœud N:**
longueur maximale du chemin entre ce nœud et une feuille du sous arbre de racine N.

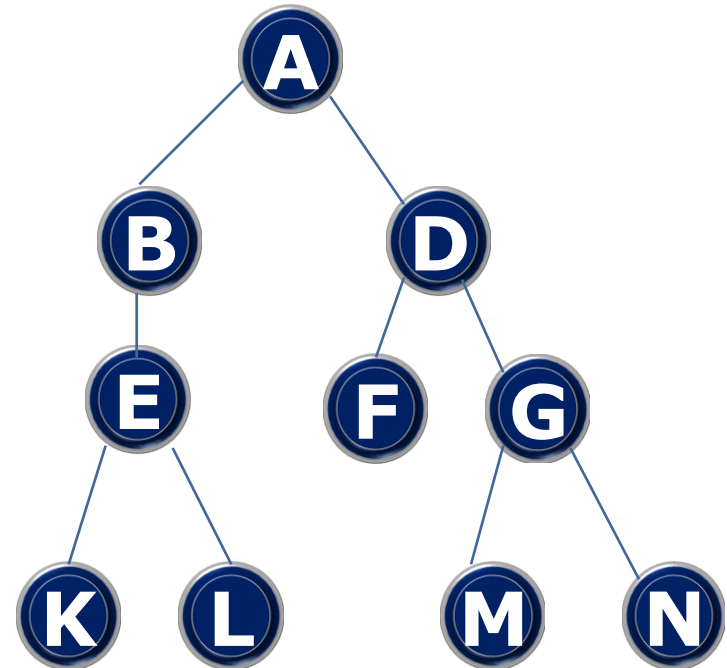
Hauteur(K)=1; Hauteur(G)=2

→ **Profondeur (ou hauteur) d'un arbre:**
 le nombre de nœuds de la branche la plus longue;

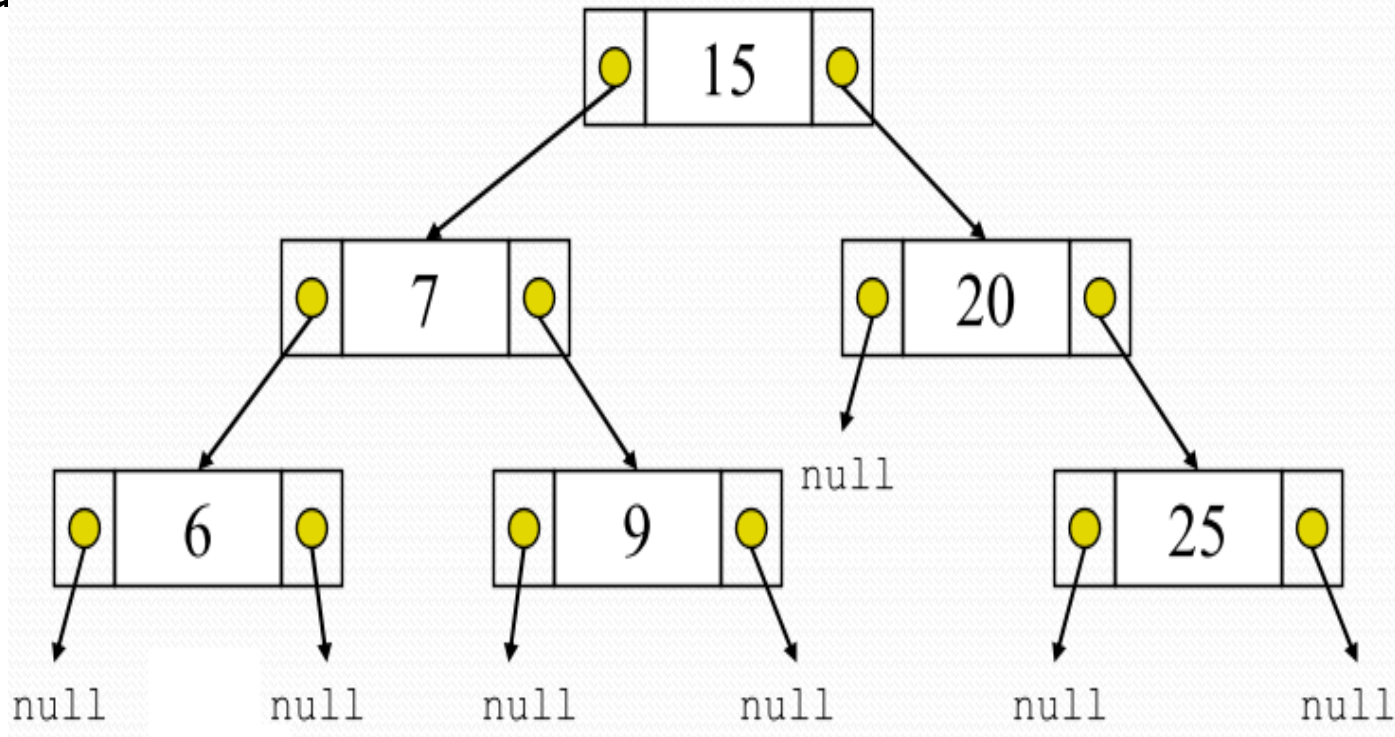


Définitions

- **Degré d'un nœud:** le nombre de ses fils;
- **Degré (ordre) d'un arbre:** le degré maximum de ses nœuds;
- **Arbre binaire:** un arbre d'ordre 2;
- **Arbre binaire complet:** chaque nœud autre qu'une feuille admet deux fils et toutes les feuilles sont au même niveau;



Arbre binaire de recherche: Soit A un arbre binaire. On dit que A est un arbre binaire de recherche (ABR) si pour chaque nœud N de ABR on a les valeurs des nœuds du sous arbre gauche de N sont inférieure a la valeur de N et les valeurs des nœuds du sous arbre droit de N sont supérieu



Implémentation d'un arbre binaire en langage C

```
typedef struct element  
{int contenu;  
  struct element* gauche;  
  struct element* droit;  
} noeud ;
```

```
noeud* CreerNoeud(int valeur)  
{ noeud*N=(noeud*)malloc(sizeof(noeud));  
  N->contenu=valeur;  
  N->gauche = NULL;  
  N->droit = NULL;  
  return N;  
}
```

Ajout d'un nœud dans un arbre binaire

```
noeud* ajouterNoeud(noeud* Arbre, char chemin[], int valeur)
{ int i;
```

```
    noeud* p=CreerNoeud(valeur);
    if(Arbre==NULL) return (p);
```

```
    noeud* temp=Arbre;
    for(i=0;((i<strlen(chemin)-1)&&
        (temp->gauche!=NULL || temp->droit!=NULL));i++)
    {if(chemin[i]=='-') temp=temp->gauche;
      else temp=temp->droit;
    }
```

```
    if(chemin[strlen(chemin)-1]=='-') temp->gauche=p;
    else temp->droit=p;
    return(Arbre);
}
```

```
void Parcour_NGD(noeud* Arbre)
{if(Arbre!=NULL)
  {printf("%d\n",Arbre->contenu);
   Parcour_NGD(Arbre->gauche);
   Parcour_NGD(Arbre->droit);
  }}
```

```
void Parcour_GND(noeud* Arbre)
{if(Arbre!=NULL)
  {Parcour_GND (Arbre->gauche);
   printf("%d\n",Arbre->contenu);
   Parcour_GND (Arbre->droit);
  }}
```

```
void Parcour_GDN(noeud* Arbre)
{if(Arbre!=NULL)
  {Parcour_GDN(Arbre->gauche);
   Parcour_GDN(Arbre->droit);
   printf("%d\n",Arbre->contenu);
  }}
```

```

void ParcoursEnlargeur(noeud* Arbre)
{
  if(Arbre!=NULL)
  {int n,i;
  noeud** T=(noeud**)malloc(100*sizeof(noeud*));
  T[0]=Arbre; n=1;
  while(n>0)
  {printf("%d\n",T[0]->contenu);
   if (T[0]->gauche!=NULL)
     {T[n]=T[0]->gauche; n++;}
   if (T[0]->droit!=NULL)
     {T[n]=T[0]->droit; n++;}
  }
  for(i=1;i<n;i++)
    T[i-1]=T[i];
  n--;
}
}

```

Libération mémoire d'un arbre binaire

```
void DesalouerArbre(noeud *R)
{if (R != NULL)
  { DesalouerArbre(R->gauche);
    DesalouerArbre(R->droit);
    free(R);
  }
}
```

Taille(nombre de nœuds) d'un arbre binaire

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

```
int max(int x,int y)  
    {if(x<y) return(y);  
    else return(x);}
```

```
int hauteur(noeud* arbre)  
    {if(arbre==NULL)  
        return(0);  
    noeud* X=arbre->gauche;  
    noeud* Y=arbre->droit;  
    return(1+max(hauteur(X),hauteur(Y)));  
    }
```

```
int feuille(noeud* nd)  
    {if(nd==NULL)  
        return(0);  
    else if((nd->gauche==NULL) && (nd->droit==NULL))  
        return(1);  
    else return(0);  
    }
```

Nombre de feuilles d'un arbre binaire

int N=0; // variable globale

```
void Nombre_de_Feuilles_NGD(noeud* Arbre)  
{  
  if(Arbre!=NULL)  
  {  
    if(feuille(Arbre)==1)N++;  
    Nombre_de_Feuilles_NGD(Arbre->gauche);  
    Nombre_de_Feuilles_NGD(Arbre->droit);  
  }  
}
```


Vérification d'un arbre binaire s'il est complet

```
int EstArbreBinaireComplet(noead* A)
{if((A==NULL) || (feuille(A)==1))
    return(1);
else if(hauteur(A->gauche)==hauteur(A->droit))
    {int cond1=EstArbreBinaireComplet(A->gauche);
    int cond2=EstArbreBinaireComplet(A->droit);
    return(cond1 && cond2);
    }
else return(0);
}
```

//On suppose que A est binaire, on peut écrire:

```
int EstArbreBinaireCompletV2(noead* A)
{ return(taille(A)==(pow(2,hauteur(A))-1)); }
```

Vérification d'un arbre binaire s'il est de recherche

```

int Min;
void Minimum(noeud* Arbre)
{if(Arbre!=NULL)
 {if(Arbre->contenu<Min)
  Min=Arbre->contenu;
  Minimum(Arbre->gauche);
  Minimum(Arbre->droit); }}
    
```

```

int MinArbre(noeud* A)
{Min=INT_MAX;
 Minimum(A);
 return(Min); }
    
```

```

int Max;
void Maximum(noeud* Arbre)
{if(Arbre!=NULL)
 {if(Arbre->contenu>Max)
  Max=Arbre->contenu;
  Maximum (Arbre->gauche);
  Maximum(Arbre->droit); }}
    
```

```

int MaxArbre(noeud* A)
{Max=-INT_MAX;
 Maximum(A);
 return(Max); }
    
```

```

int max3(int x,int y, int z)
{int m=x;    if(m<y) m=y;
  if(m<z) m=z;  return(z);}
    
```

```

int MaxArbreV2(noeud* A)
{
if(A==NULL) return(-INT_MAX);
else if(taille(A)==1) return(A->contenu);
  else return(max3(A->contenu, MaxArbreV2(A->gauche),MaxArbreV2(A->droit)));
}
    
```

```
int ABR(noead* A)
{ if (A==NULL || feuille(A)==1)return 1;
  else if ((A->gauche==NULL)&&(A->droit!=NULL))
    {if(A->contenu<MinArbre(A->droit)) return (ABR(A->droit));
     else return(0);}
  else if ((A->droit==NULL)&&(A->gauche!=NULL))
    {if(A->contenu>MaxArbre(A->gauche)) return(ABR(A->gauche));
     else return(0); }
  else {int cond1=(A->contenu>MaxArbre(A->gauche));
        int cond2=(A->contenu<MinArbre(A->droit));
        int cond3=ABR(A->gauche);
        int cond4=ABR(A->droit);
        return (cond1 && cond2 && cond3 && cond4);
    } }
```

```
int N=0; // variable globale
void Nombre_de_Feuilles_NGD(noead* Arbre)
{
  if(Arbre!=NULL)
  { if(feuille(Arbre)==1)N++;
    Nombre_de_Feuilles_NGD(Arbre->gauche);
    Nombre_de_Feuilles_NGD(Arbre->droit);
  }
}
```

tri Maximier: présentation

- ▶ L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un arbre binaire.
- ▶ Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément, etc.
- ▶ Ainsi le n^{e} élément a pour enfants les éléments $2n+1$ et $2n+2$. Si le tableau n'est pas de taille $\sum_{k=0}^{n-1} 2^k$ avec $n > 1$, les branches ne se finissent pas toutes à la même profondeur.
- ▶ Dans l'algorithme, on cherche à obtenir un **tas**, c'est-à-dire un **arbre binaire** vérifiant les propriétés suivantes (les deux premières propriétés découlent de la manière dont on considère les éléments du tableau) :

tri Maximier: présentation

- ▶ la différence maximale de profondeur entre deux feuilles est de 1 (i.e. toutes les feuilles se trouvent sur la dernière ou sur l'avant-dernière ligne) ;
- ▶ les feuilles de profondeur maximale sont « tassées » sur la gauche.
- ▶ chaque nœud est de valeur supérieure (resp. inférieure) à celles de ses deux fils, pour un tri ascendant (resp. descendant).

Il en découle que la racine du tas (le premier élément) contient la valeur maximale (resp. minimale) de l'arbre. Le tri est fondé sur cette propriété.

Tri Maximier: Implémentation

```
void permuter(float T[], int i, int j)
{float c=T[i];  T[i]=T[j];  T[j]=c; }
```

```
void faire_tas_pour_le_pere(float tab[],int pere,int N)
{int fils1, fils2, GF;
  while (2*pere + 1 < N)
    {fils1 = 2*pere + 1;
     fils2 = 2*pere + 2;
     GF=fils1;
     if ((fils2 < N) && (tab[fils1] < tab[fils2]))
       GF=fils2;
     if (tab[pere] < tab[GF])
       {permuter(tab,GF,pere); pere = GF;}
     else break;
    }
}
```

Tri Maximier: Implémentation

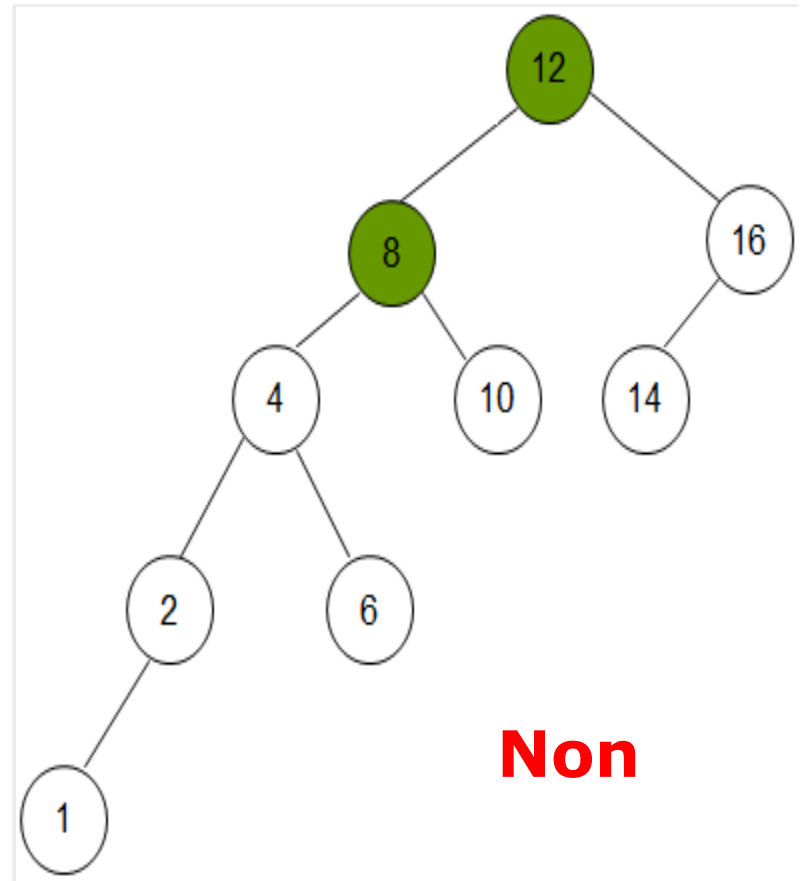
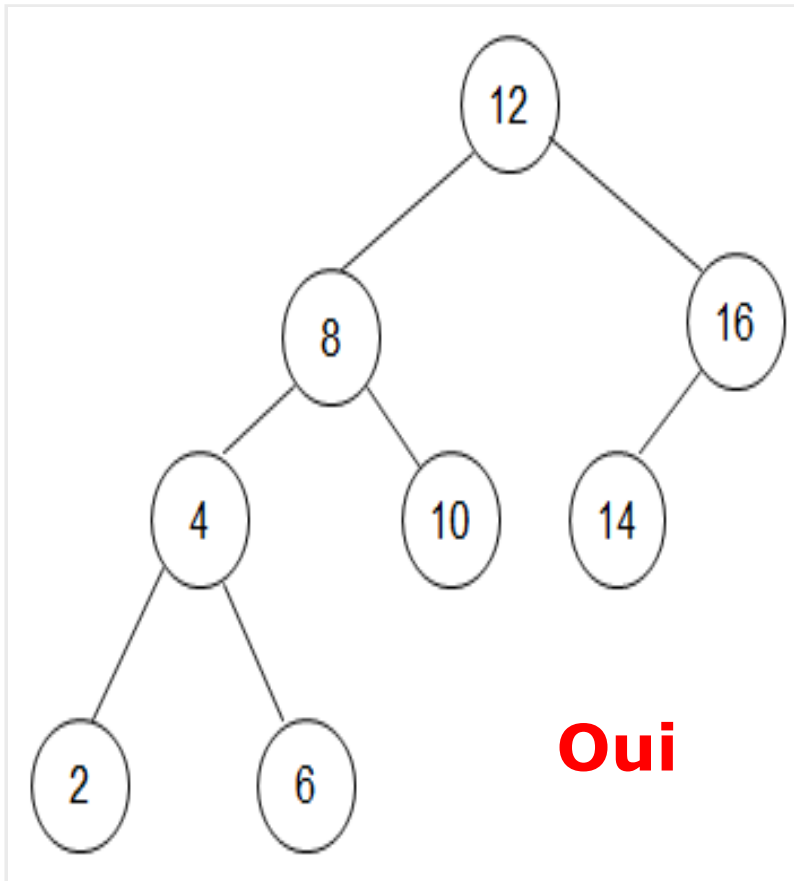
```
void faire_tas_la_premiere_foi(float tab[],int N)
{int pere;
  for (pere=N/2-1; pere>=0;pere--)
    faire_tas_pour_le_pere(tab,pere,N);
}
```

```
void faire_tas_uniquement_Tab0(float tab[],int N)
{faire_tas_pour_le_pere(tab,0, N);}
```

```
void Tri_Maximier_par_tas(float tab[],int N)
{int fin;
  faire_tas_la_premiere_foi(tab,N);
  for(fin=N-1;fin>=0;fin--)
    {permuter(tab,0,fin);
     faire_tas_uniquement_Tab0(tab,fin);
    }
}
```

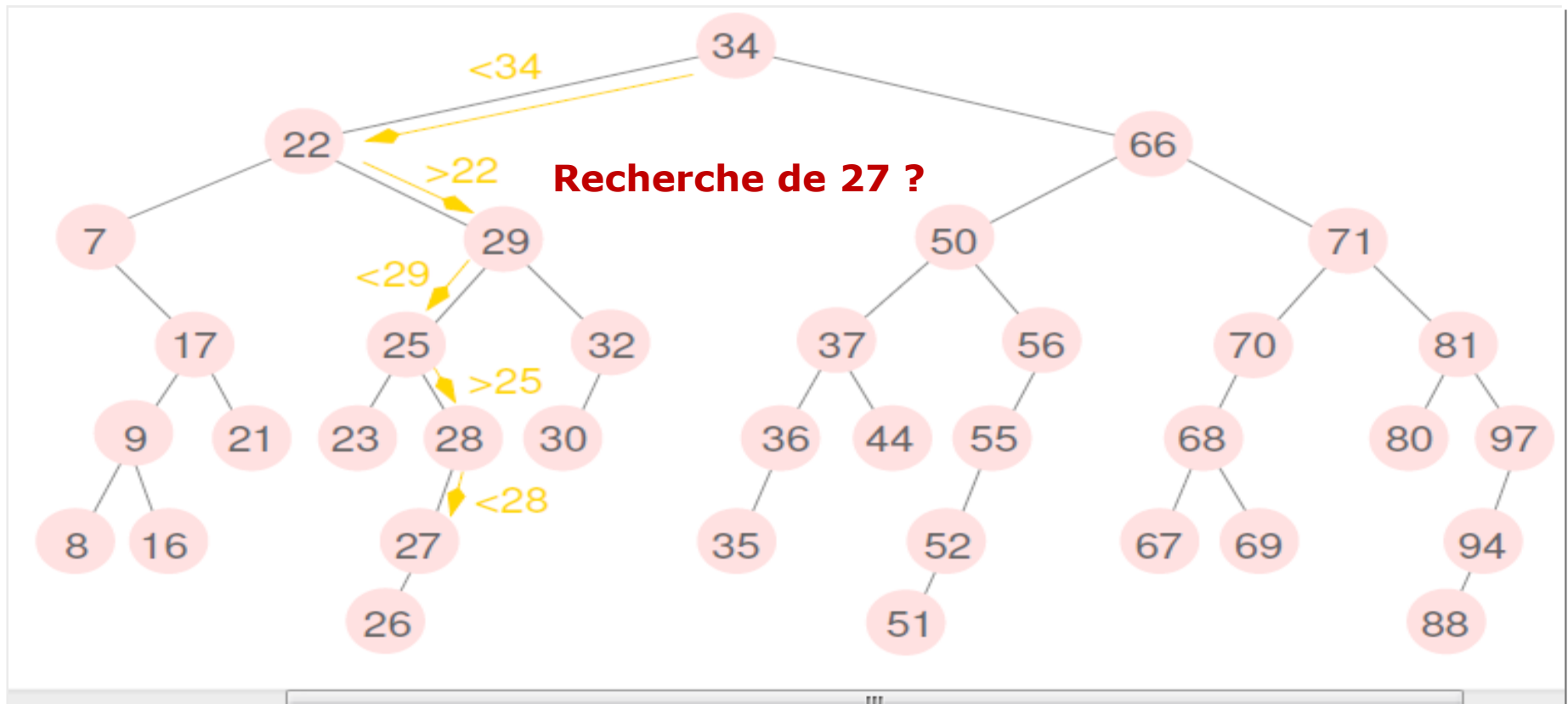
Arbre binaire équilibré

C'est un arbre binaire de recherche tel que pour chaque noeud, les hauteurs des sous-arbres gauches et droits sont différents d'au plus 1.



Algorithme de recherche dans un ABR

La recherche d'une valeur dans un ABR (arbre binaire de recherche) consiste à parcourir une branche en partant de la racine, en descendant chaque fois sur le fils gauche ou sur le fils droit suivant que la clé portée par le noeud est plus grande ou plus petite que la valeur cherchée. La recherche s'arrête dès que la valeur est trouvée ou que l'on a atteint l'extrémité d'une branche.



Algorithme d'insertion dans un ABR

Algorithme Insertion: Insertion d'une nouvelle clé dans un ABR

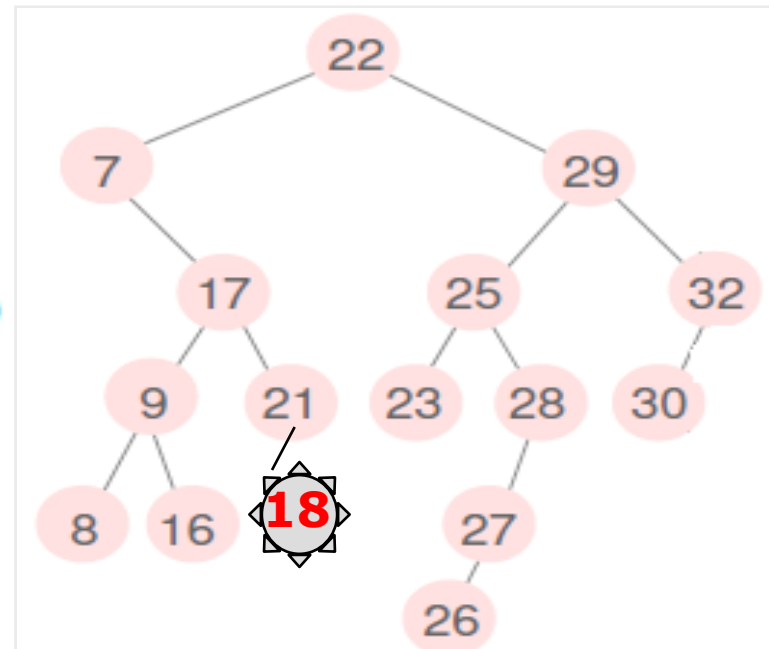
entrée : a est un ABR, v est une clé.

résultat : v est insérée dans a

début

```

    si est_vide( $a$ ) alors
        |  $a = \text{cree\_arbre}(v, \text{cree\_arbre\_vide}(), \text{cree\_arbre\_vide}())$ 
    sinon
        si  $v < \text{val}(a)$  alors
            |  $\text{Insertion}(v, \text{fils\_gauche}(a))$ 
        sinon
            si  $v > \text{val}(a)$  alors
                |  $\text{Insertion}(v, \text{fils\_droit}(a))$ 
            finsi
        finsi
    fin
    
```



Hachage

Les tableaux et les listes chaînées (structure de données linéaires SDL) ont un grand défaut lorsqu'on souhaite connaître ce qu'elles contiennent : il n'est pas possible **de trouver directement un élément précis dans SDL. Il faut parcourir la SDL en avançant d'élément en élément jusqu'à trouver celui qu'on recherche. Cela pose des problèmes de performance dès que la SDL devient volumineuse. Imaginez une SDL de 10000000 éléments où celui que l'on recherche est tout à la fin ! Combien de temps faut-il attendre ?**

Hachage: C'est une structuration des données qui permet d'accélérer la recherche de celles-ci ultérieurement. Elle repose sur l'utilisation d'un tableau de hachage TH tel que la position d'un élément E dans le tableau TH est fonction de l'élément E lui-même.

Hachage: Exemple 1

```
#define MAX 100
```

```
struct donnees_etudiant {char nom[20]; char prenom[20];  
                           char adresse[30]; float MG;};
```

```
struct donnees_etudiant TabeStructEtudiants[MAX];
```

```
// tableau à une seule dimension.
```

```
int PositionHachage(char prenom[])
```

```
{ int h = 0; int i; // Valeur calculée
```

```
    for (i = 0; prenom[i] != '\0'; i++) h = h+ prenom[i];
```

```
    h=h%MAX;    return h; }
```

```
void Ajouter(struct donnees_etudiant DE)
```

```
{ int h = PositionHachage(DE.prenom);
```

```
    TabeStructEtudiants[h]=DE; }
```

Hachage: Exemple 1

```
int egal(struct donnees_etudiant A, struct donnees_etudiant B)
{if(strcmp(A.nom,B.nom)||strcmp(A.prenom,B.prenom)||
    strcmp(A.adresse,B.adresse)||A.MG!=B.MG) return(0);
else return(1); }
```

```
void Chercher(struct donnees_etudiant DE)
{ int h = PositionHachage(DE.prenom);
  if(egal(TabeStructEtudiants[h],DE)==1)
    {printf("\n %s %s existe dans le tableau",DE.nom,DE.prenom);}
  else {printf("\n %s %s n'existe pas dans le tableau",DE.nom,DE.prenom);}
}
```

Hachage: Exemple 1

```

main()
{int i;
for(i=0;i<MAX;i++)
{struct donnees_etudiant DE={"", "", "", 0};
TabStructEtudiants[i]=DE;}

struct donnees_etudiant A={"Alami", "amine", "Casablanca", 11};
struct donnees_etudiant B={"Alami", "imane", "Casablanca", 11};
Ajouter(A); Ajouter(B); Chercher(A); Chercher(B); }
  
```

```

amine Alami n'exite pas dans le tableau
imane Alami existe dans le tableau
-----
  
```

Inconvénient: si des prénoms des étudiants sont sous forme d'anagrammes comme « amine » et « imane » alors ils seront stockés dans la même case du tableau de hachage TabStructEtudiants et par conséquent l'insertion de « imane » écrase « amine » qui était dans la même case ! Ce problème est appelé collision des clés de hachage.

Hachage: Exemple 2

Soit le type personne suivant:

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

Soit le tableau struct personne T[140][1000]; // tableau à deux dimensions

Le hachage peut être fait par exemple en mettant les personnes ayant le même âge dans la même ligne de T; On dit que la clé de hachage est l'âge.

Abouali Amine 17	Ahmadi Rachida 17	*	*	*
*	*	*	*	*
Morchid imane 46	Amrani Abdellah 46	Mouslim Salim 46	*	*
*	*	*	*	*

Inconvénients:

- 1) Gaspillage de mémoire dû aux cases inexploitées !
- 2) Parfois la largeur du tableau est insuffisante (par exemple plus que 1000 personnes ont 20 ans !)

Hachage: Exemple 2

De la même manière, le hachage peut être fait par exemple en mettant les personnes ayant la même somme des codes ASCII des caractères composant le prénom dans la même ligne du tableau....

*	*	*	*	*
*	*	*	*	*
Morchid imane 27	Amrani amine 39	*	*	*
*	*	*	*	*

Hachage: Exemple 2

```
#define MAXLIGNES 100
#define MAXCOLONES 100
struct donnees_etudiant {char nom[20]; char prenom[20]; char adresse[30]; float MG;};
struct donnees_etudiant TabeStructEtudiants[MAXLIGNES][MAXCOLONES];
int PosRemplissage[MAXLIGNES]; // initialisé par -1
```

Le hachage peut être fait par exemple en mettant les personnes ayant la même somme des codes ASCII des caractères composant le prénom dans la même ligne de TabeStructEtudiants; On dit que la clé de hachage est le prénom dans ce cas.

```
int LigneCorrespondante(char prenom[])
{ int h = 0; int i; // Valeur calculée
  for (i = 0; prenom[i] != '\0'; i++) h = h+ prenom[i];
  h=h%MAXLIGNES;
  return h; }
```

Hachage: Exemple 2

```
void Ajouter(struct donnees_etudiant DE)  
{ int h = LigneCorrespondante(DE.prenom);  
  PosRemplissage[h]++;  
  if(PosRemplissage[h]>=MAXCOLONES) {printf("\n ligne pleine"); return; }  
  else TabeStructEtudiants[h][PosRemplissage[h]]=DE;  
}
```

```
int egal(struct donnees_etudiant A, struct donnees_etudiant B)  
{if(strcmp(A.nom,B.nom)  
  ||strcmp(A.prenom,B.prenom)||strcmp(A.adresse,B.adresse)||A.MG!=B.MG)  
  return(0);  
  else return(1);  
}
```

Hachage: Exemple 2

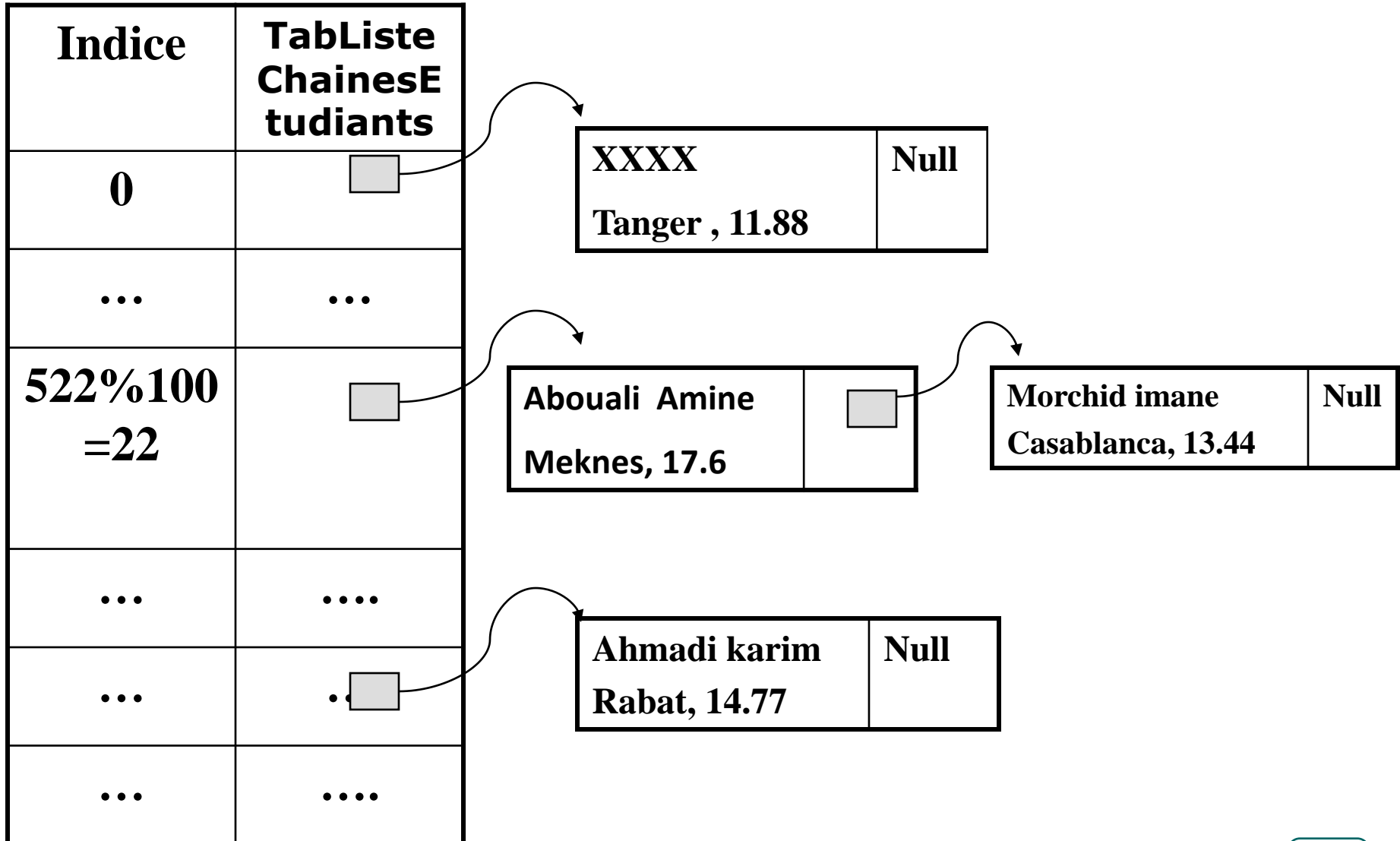
```

void Chercher(struct donnees_etudiant DE)
{ int i, trouve=0, h = LigneCorrespondante(DE.prenom);
  for(i=0;i<=PosRemplissage[h];i++)
    if(egal(TabeStructEtudiants[h][i],DE)==1)
      {printf("\n %s %s existe dans le tableau",DE.nom,DE.prenom);
        trouve=1; break;}
  if(trouve==0)
    {printf("\n %s %s n'existe pas dans le tableau",DE.nom,DE.prenom);}
}
  
```

```

main()
{int i; for(i=0;i<MAXLIGNES;i++) {PosRemplissage[i]=-1;}
struct donnees_etudiant A={"Alami","amine","Casablanca",11};
struct donnees_etudiant B={"Alami","imane","Casablanca",11};
Ajouter(A); Ajouter(B); Chercher(A); Chercher(B); }
  
```

Exemple 3: Traitement des collisions



Exemple 3: Traitement des collisions

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAXLIGNES 100
struct donnees_etudiant{char prenom[20];    char nom[20];
                        char adresse[30];    float MG;};
struct noeud_etudiant { struct donnees_etudiant D;

    struct noeud_etudiant* suivant; };

typedef struct noeud_etudiant* PtrLstEtuds;
typedef struct donnees_etudiant SDE;

PtrLstEtuds TabListeChainesEtudiants[MAXLIGNES];

PtrLstEtuds ajouterEnTete(PtrLstEtuds Tete, SDE X)
{  int taille=sizeof(struct noeud_etudiant);
   PtrLstEtuds nouvEtudiant =(PtrLstEtuds) malloc(taille);
   nouvEtudiant->D=X;
   nouvEtudiant->suivant = Tete;
   return nouvEtudiant;
}
  
```

Exemple 3: Traitement des collisions

```

int LigneCorrespondante(char prenom[])
{ int h = 0; int i;    // Valeur calculée
  for (i = 0; prenom[i] != '\0'; i++)    h = h+ prenom[i];
  h=h%MAXLIGNES;    return h; }

void Ajouter(SDE X)
{int indice=LigneCorrespondante(X.prenom);
  TabListeChainesEtudiants[indice]=ajouterEnTete(TabListeChainesEtudiants[indice],X);
}

void Chercher(struct donnees_etudiant DE)
{int indice=LigneCorrespondante(DE.prenom),trouve=0;
  PtrLstEtuds tmp=TabListeChainesEtudiants[indice];
  while(tmp != NULL)
  { if(egal(tmp->D,DE)==1)
    {printf("\n %s %s existe dans le tableau",DE.nom,DE.prenom);
      trouve=1; break; }
    tmp = tmp->suivant;
  }
  if(trouve==0)
  {printf("\n %s %s n'existe pas dans le tableau",DE.nom,DE.prenom);}
}
  
```

Exemple 3: Traitement des collisions

```
main()  
{int i; for(i=0;i<MAXLIGNES;i++)  
    {TabListeChainesEtudiants[i]=NULL;}  
  
struct donnees_etudiant A={"Alami","amine","Casablanca",11};  
struct donnees_etudiant B={"Alami","imane","Casablanca",11};  
  
Ajouter(A); Ajouter(B); Chercher(A); Chercher(B);  
}
```

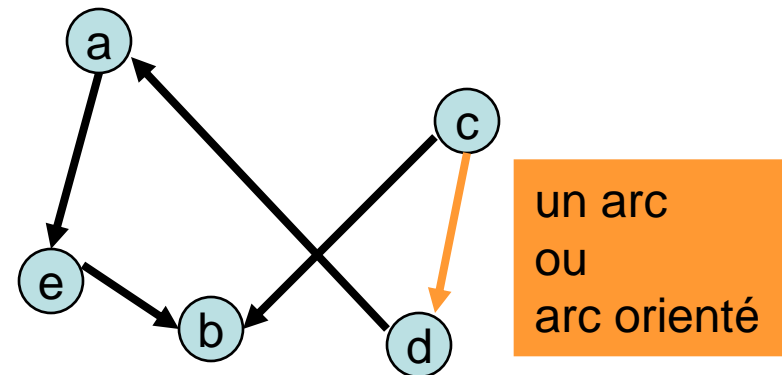
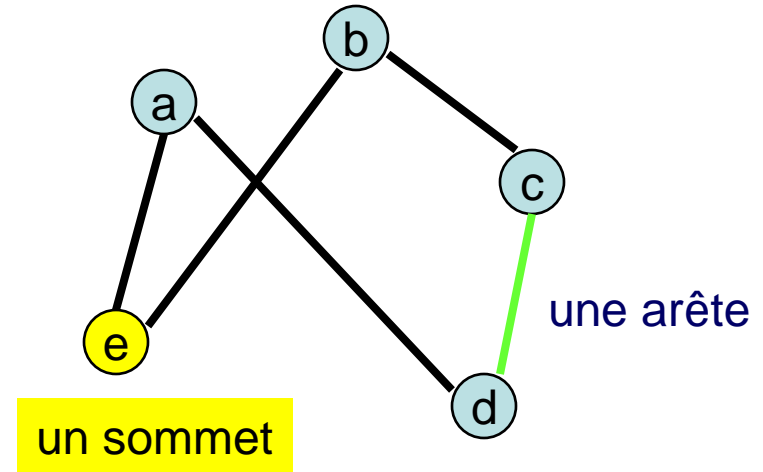
```
amine Alami existe dans le tableau  
imane Alami existe dans le tableau  
-----
```

Les graphes

Un graphe est défini par:

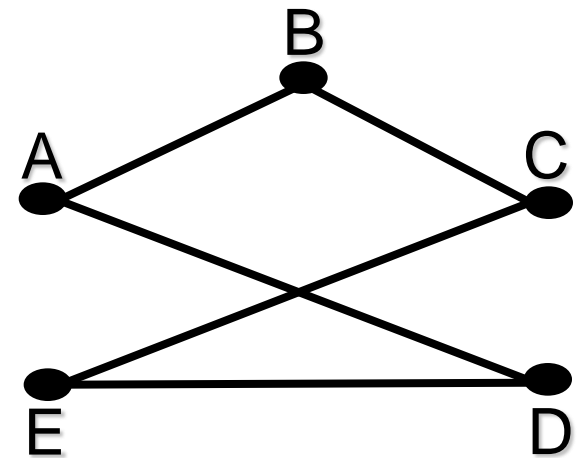
- par un ensemble S de points (appelés « **sommets** »), le plus souvent symbolisés par des numéros 1, 2, 3, etc...., ou par des lettres a, b, c...
- par des liens reliant certains sommets entre eux ; ces liens qui créent donc des couples de sommets, se nommeront (et se représenteront sur le dessin) par des « **arcs** » ou des « **arêtes** » selon que le graphe est « **orienté** » ou « **non orienté** ».

Un graphe **non orienté**



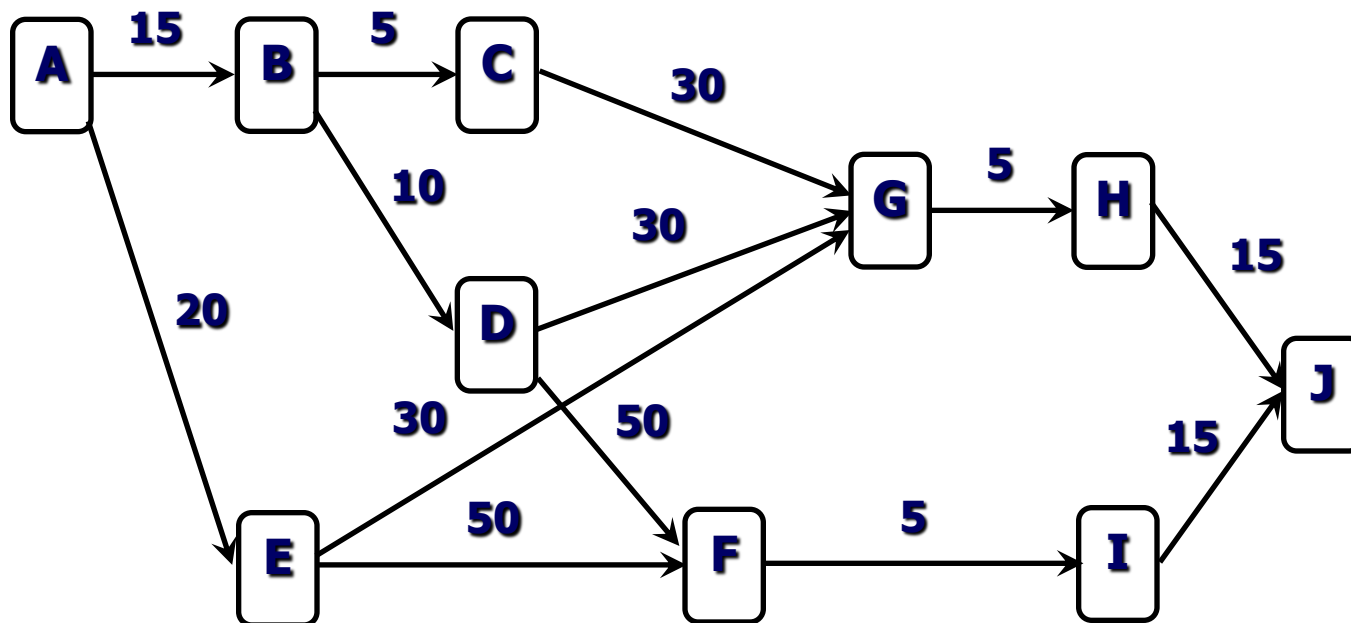
Un graphe **orienté**

Exemple 1: Voici la représentation d'un mini-réseau Facebook de 5 personnes.



On constate donc, entre autres,
que A est « ami » avec B.
Cependant, A n'est pas « ami » avec C.

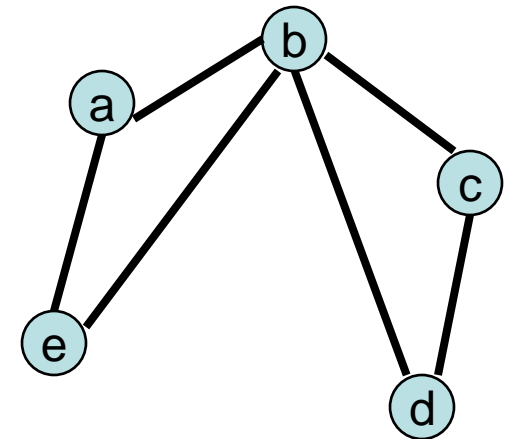
Exemple 2: A,B,C, D,E, F, G,H,I,J : des villes
distance entre A et E= 20 km



- **ordre d'un graphe** : nombre de sommets du graphe.
- Une **chaîne** dans un graphe G , est une suite d'arêtes qui se suivent et relient certains sommets du graphe.
Si le premier sommet est a et le dernier b , on dira que la chaîne **relie** a et b .
- On dira que la chaîne a pour **longueur k** lorsque le nombre d'arêtes de la chaîne est k .
Une chaîne doit comporter au moins une arête.
- Un **cycle** dans un graphe G , est une chaîne qui a le même point de départ et d'arrivée.
- Deux arcs sont dits **adjacents** s'ils ont une extrémité en commun.
- Deux sommets sont dits **adjacents** si un arc les relie.

• Par exemple, $a-b-c-d-b$ est une chaîne qui relie a à b ; elle a pour longueur 4.

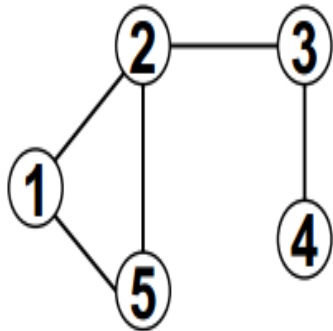
• Ce graphe a pour ordre 5.



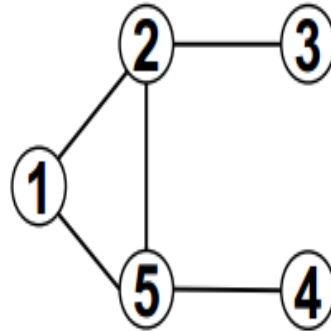
- ❑ Une arête joignant un sommet à lui-même, est appelée boucle.
- ❑ Un graphe est complet si chaque sommet est adjacent à tous les autres, c'est à dire si toutes les arêtes possibles existent (sauf les boucles).
- ❑ Un sommet est isolé s'il n'est adjacent à aucun autre sommet.
- ❑ Un graphe est nul s'il n'a aucune arête, c'est ensemble de sommets isolés.
- ❑ Un graphe est dit connexe si on peut relier deux quelconques de ses sommets par une chaîne.
- ❑ Graphe eulérien: on peut « parcourir » le graphe en partant d'un sommet quelconque et en empruntant exactement une fois chaque arête pour revenir au sommet de départ. On dit que ce graphe contient un cycle Eulerien.
- ❑ **Théorème 1:** Théorème d'Euler (1736):
Un graphe connexe est eulérien si et seulement si chacun de ses sommets est incident à un nombre pair d'arêtes.

- **Distance** entre deux sommets i et j est la longueur de la chaîne la plus courte qui les relie
- **Chaîne Eulérienne**: une chaîne est dite eulérienne si cette chaîne comporte exactement une fois toutes les arêtes du graphe.
- **Théorème 2**: Un graphe connexe G admet une chaîne eulérienne distincte d'un cycle Eulerien si et seulement si le nombre de sommets de G de degré impair est égal à 2.
- **Cycle Hamiltonien** : c'est un cycle passant une seule fois par tous les **sommets** d'un graphe et revenant au sommet de départ.

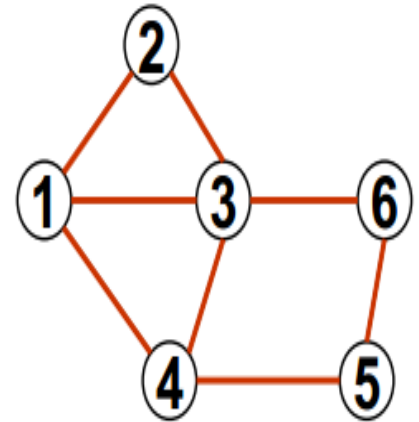
graphe A



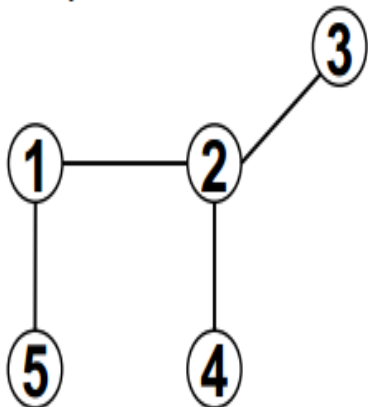
Graphe B



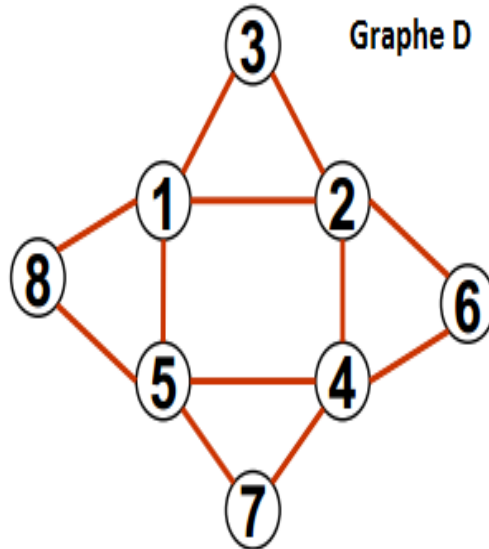
Graphe C



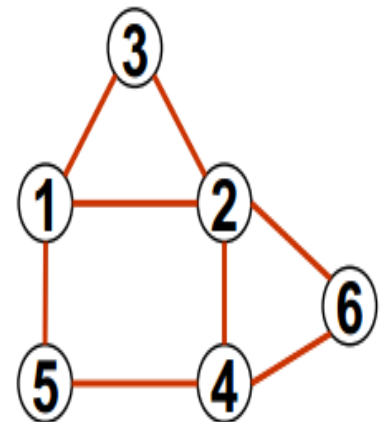
Graphe D



Graphe D

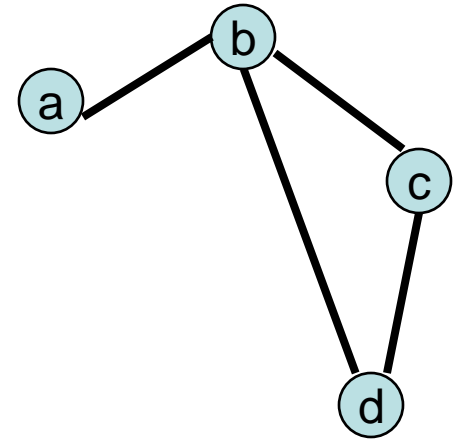


Graphe E



Définition : matrice d'adjacence associée à un graphe

Dans un graphe orienté, pour un arc $(x;y)$ donné, on dit que y est le successeur de x et que x est le prédécesseur de y .
Si le graphe est non orienté, x est à la fois successeur de y et prédécesseur de y .



Pour le traitement informatique, tout graphe possède une matrice booléenne (i.e avec des 0 et des 1 seulement) associée : chaque ligne indique les successeurs par un 1, et l'absence de successeur par un 0.

	a	b	c	d
a	0	1	0	0
b	1	0	1	1
c	0	1	0	1
d	0	1	1	0

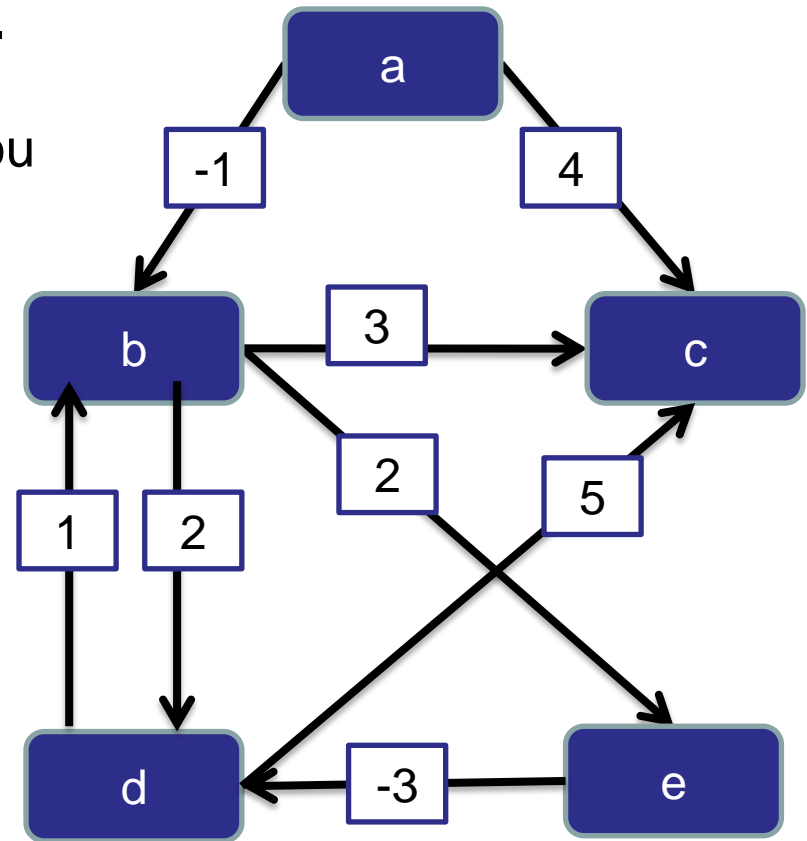
c est en **relation** avec **b** et **d**
mais **pas en relation** avec **a**

Dans un graphe **valué** ou **pondéré** les arrêtes (ou arcs) ont des **coûts (poids)**.

Dans un graphe non **valué** les arrêtes (ou arcs) n'ont pas des **coûts (poids)**.

Matrice de coûts:

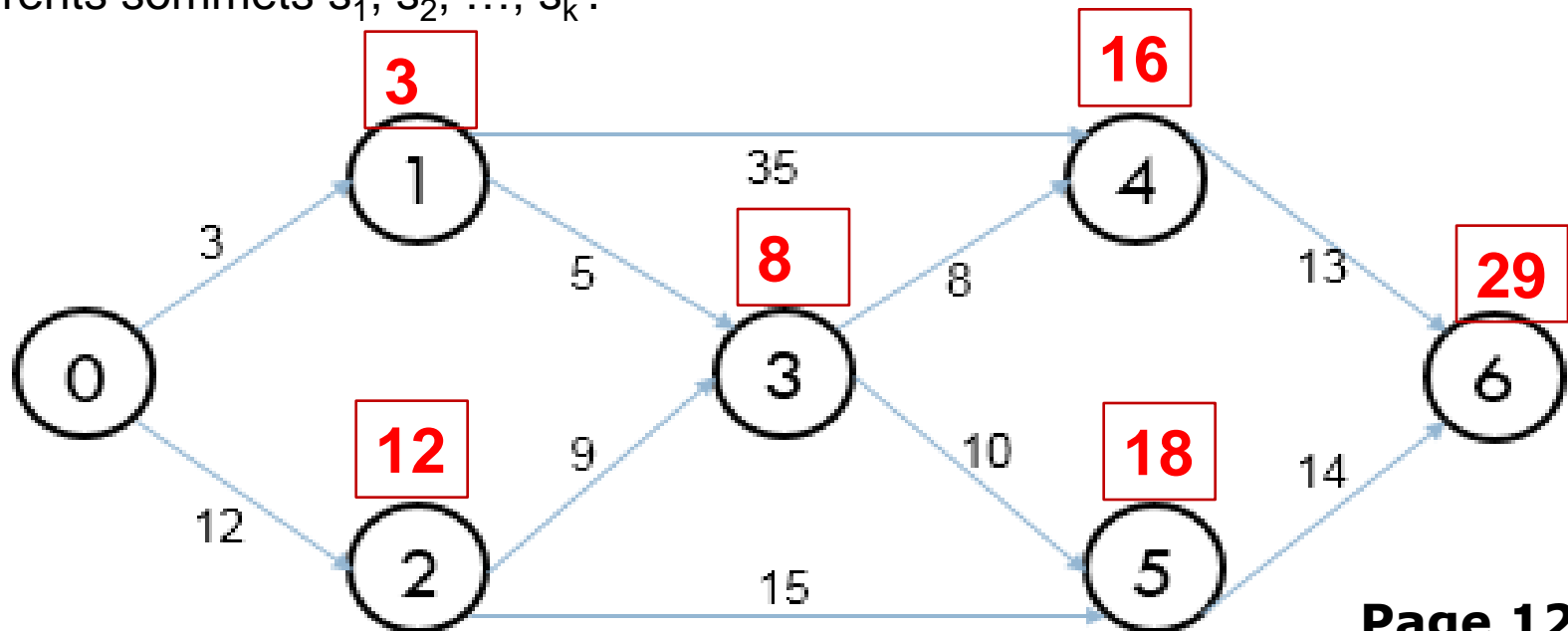
	a	b	c	d	e
a	0	-1	4	0	0
b	0	0	3	2	2
c	0	0	0	0	0
d	0	1	5	0	0
e	0	0	0	-3	0



Le Plus court chemin

Il existe de nombreux algorithmes déterminant un ou le plus court chemin dans un graphe connexe pondéré. Par exemple: Warshall, Floyd, Dijkstra, Branch and Bound, Bellman-Ford On se limitera à la recherche d'un plus court chemin entre deux sommets du graphe pondéré avec des poids positifs en utilisant la matrice d'adjacence pour représenter le graphe.

Algorithme de dijkstra : L'algorithme dû à Dijkstra est basé sur le principe suivant : Si le plus court chemin reliant E à S passe par les sommets s_1, s_2, \dots, s_k alors, les différentes étapes sont aussi les plus courts chemins reliant E aux différents sommets s_1, s_2, \dots, s_k .



Algorithme de parcours en profondeur :

Voisins(s) : renvoie la liste des sommets adjacents à s.
Marquer(s) : marque un sommet s comme exploré, de manière à ne pas le considérer plusieurs fois.

Le parcours en profondeur DFS (Depth First Search) :

1) Initialement, aucun sommet n'est marqué.

2) DFS (graphe G, sommet s)

{ Marquer(s);

POUR CHAQUE élément s_fils de Voisins(s) FAIRE:

SI NonMarqué(s_fils) ALORS:

DFS(G,s_fils);

FIN-SI

FIN-POUR }

3) Pour tout sommet x non marqué faire DFS(G,x)

Algorithme de parcours en largeur (BFS) :

- 1) Initialement, aucun sommet n'est marqué.
- 2) **ParcoursLargeur(Sommet s)**
 {f = CreerFile();
 f.enfiler(s);
 TANT-QUE f.vide()=Faux FAIRE:
 s = f.defiler();
 marquer(s);
 POUR TOUT voisin de s FAIRE:
 Si voisin est non marqué FAIRE:
 f.enfiler(voisin);
 FIN Si
 FIN POUR
 FIN TANT QUE
 }
3) Pour tout sommet x non marqué faire **ParcoursLargeur(x)**

```
#define MAX 8  
int G[MAX][MAX];
```

```
int NonVide(int Courant[MAX])  
{int i;  
    for (i=0;i<MAX;i++) if(Courant[i]!=-1) return(1);  
    return(0);  
}
```

```
void afficher(int X[MAX])  
{int i; printf("\n affichage: ");  
for(i=0;i<MAX;i++) printf("%2d  ",X[i]);  
}
```

```
void succNonVisite(int G[MAX][MAX],int test[MAX], int s, int sucNV[MAX])  
    {int i,pos=-1;  
    for(i=0;i<MAX;i++) sucNV[i]=-1;  
    for(i=0;i<MAX;i++)  
        if (G[s][i]!=0 && test[i]==0)  
            {pos++;sucNV[pos]=i;}  
    }
```

```

void DFS(int G[MAX][MAX],int test[MAX], int s)
    {int i,j;
    if(test[s]==0)
    {
    int sucNV[MAX];
    test[s]=1;
    printf("\n %d est marque ",s);
    for (i=0;i<MAX;i++)
    {succNonVisite(G,test,s, sucNV);
    for(j=0;j<MAX;j++)
        if(sucNV[j]!=-1) DFS(G,test,sucNV[j]);
    }
    }
    for (i=0;i<MAX;i++)
        if(test[i]==0) {DFS(G,test,i);}
    }

```

```

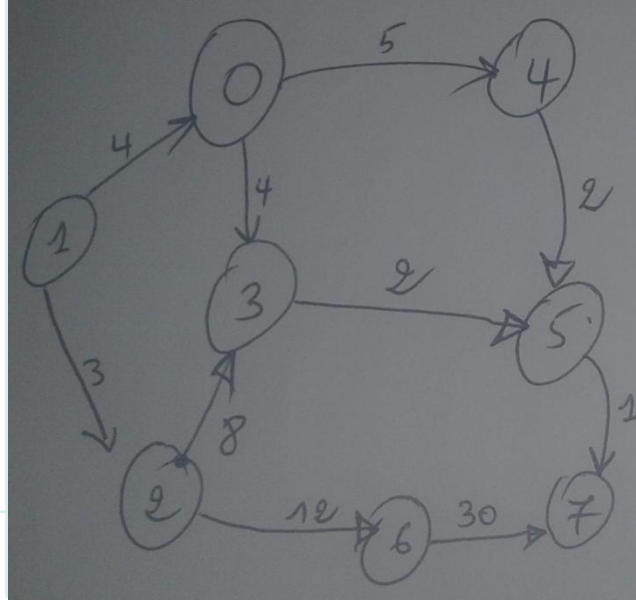
Void parcours_DFS(int G[MAX][MAX],int test[MAX], int s)
    {int i;
    for(i=0;i<MAX;i++) if(test[i]==0) DFS(G,test,i);
    }

```

```

void BFS(int G[MAX][MAX], int s)
{printf("\n -----BFS----- \n");
int test[MAX],fifo[MAX],sucNV[MAX];
int i,j,sommet,pos;
for (i=0;i<MAX;i++) test[i]=0;
recommencer:
pos=-1;
for (i=1;i<MAX;i++) fifo[i]=-1;
pos++; fifo[0]=s;
while(pos!=-1)
{sommet=fifo[0];
for (i=1;i<MAX;i++) fifo[i-1]=fifo[i]; pos--;
if(test[sommet]==0)
{test[sommet]=1;
printf("\n %d est marque ",sommet);//getch();
succNonVisite(G,test,sommet, sucNV);
for(j=0;j<MAX;j++)
    if(sucNV[j]!=-1) {pos++; fifo[pos]=sucNV[j];};
}
}
for (i=0;i<MAX;i++)
    if(test[i]==0) {s=i; goto recommencer;}
}

```



```

main()
{int i,j,test[MAX];
for(i=0;i<MAX;i++) test[i]=0;
for(i=0;i<MAX;i++)
for(j=0;j<MAX;j++)
G[i][j]=0;

```

```

G[1][2]=3; G[1][0]=4; G[0][3]=4; G[0][4]=5;
G[2][6]=12; G[2][3]=8;
G[6][7]=30; G[3][5]=2; G[5][7]=1; G[4][5]=2;
for(i=0;i<MAX;i++) afficher(G[i]);
parcours_DFS(G,test,i);
BFS(G,0);
}

```

```

le noeud 0 est marque
le noeud 3 est marque
le noeud 5 est marque
le noeud 7 est marque
le noeud 4 est marque
le noeud 1 est marque
le noeud 2 est marque
le noeud 6 est marque
-----BFS-----

```

```

0 est marque
3 est marque
4 est marque
5 est marque
7 est marque
1 est marque
2 est marque
6 est marque
-----

```