Programmation en C
Cours 6

Licence Maths-Info
Aix-Marseille Université
2011-2012

Valentin Emiya
valentin.emiya@lif.univ-mrs.fr

20 février 2012

Modalités du partiel

- Programme : tout ce que l'on a vu*.
- Date : pendant les journées banalisées 5-7 mars.
- Durée : 2h
- Document autorisé : une feuille **manuscrite** A4 recto-verso.
- Annales : en ligne+TD/TP.

* Langage de programmation, mais aussi comportement en mémoire, compilation, make, etc.

Lundi dernier

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

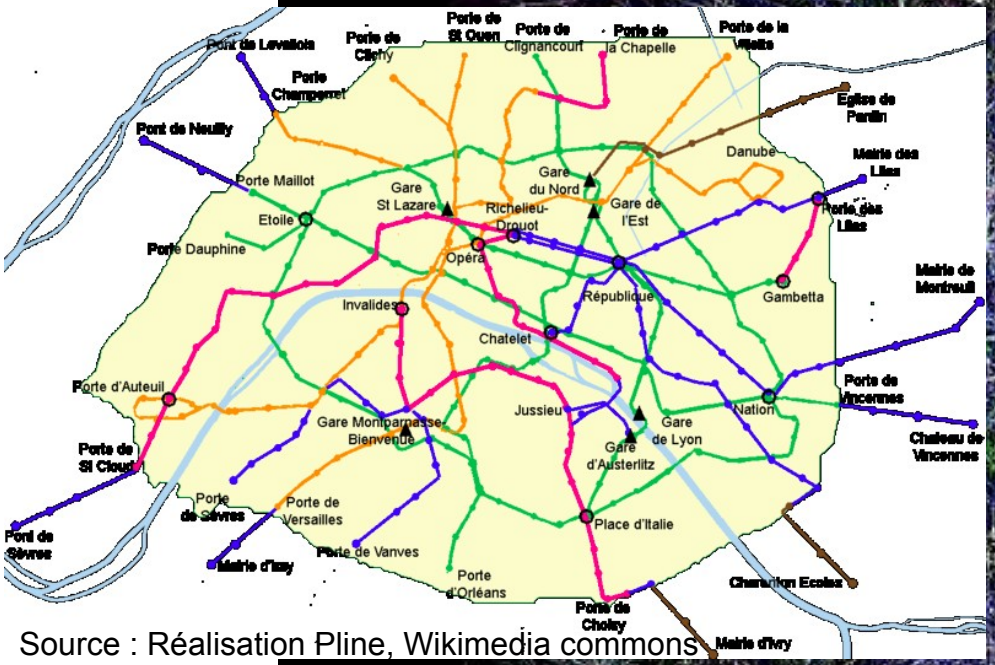
Aujourd'hui

Structures de données

* Structures linéaires
  * Tableaux
  * Listes chaînées
  * Piles et queues
* Structures non-linéaires
  * Les graphes
  * Les arbres

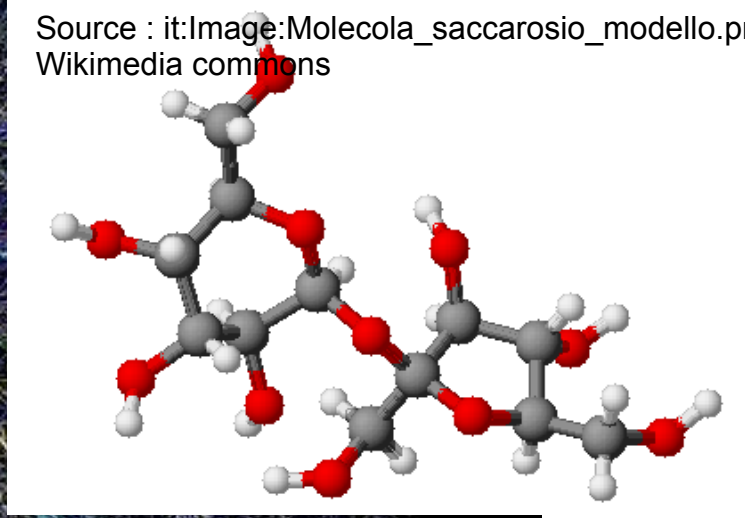
cf. poly No 2 distribué le 20/02

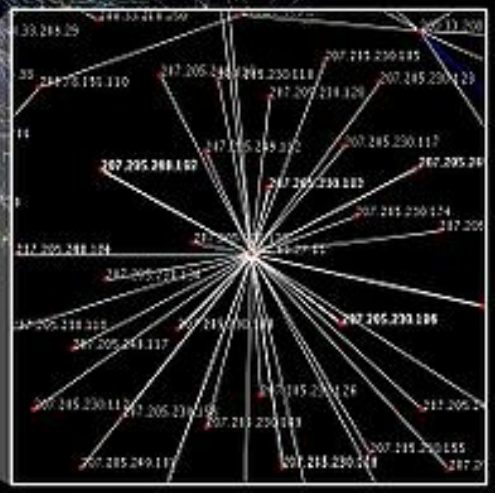
Données structurées



Source : Réalisation Pline, Wikimedia commons

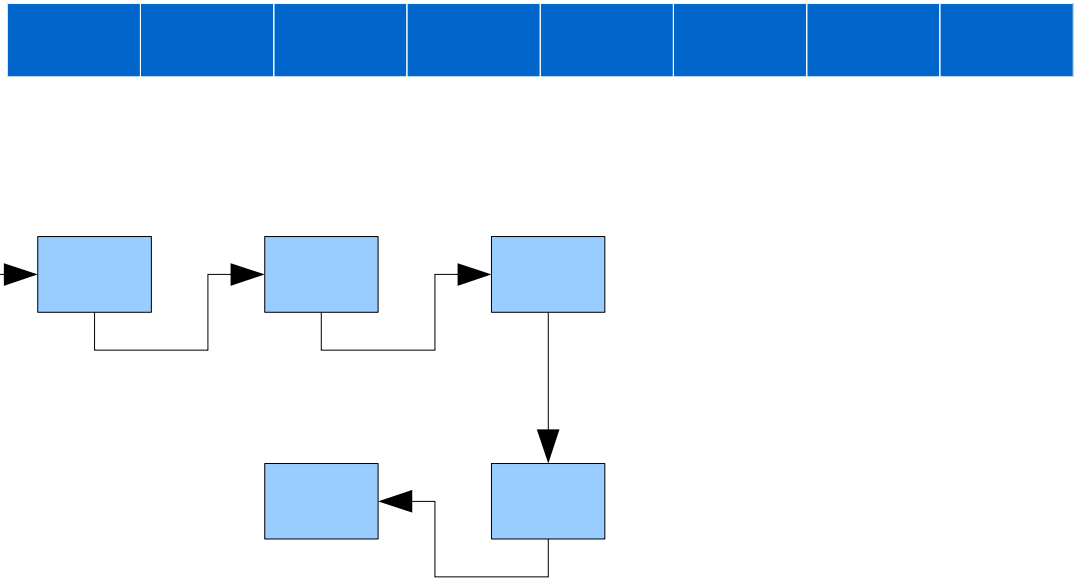
Source : it:Image:Molecola_saccarosio_modello.png
Wikimedia commons

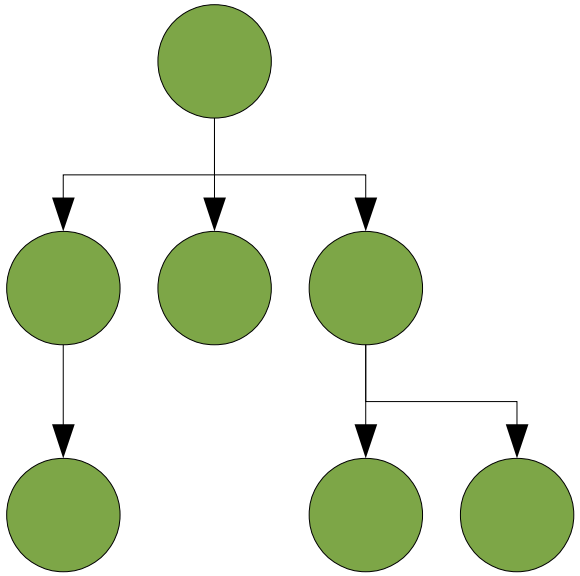


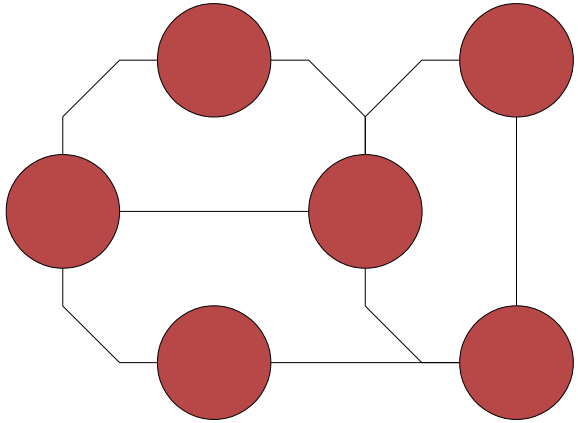


Source : The Opte Project Wikimedia commons

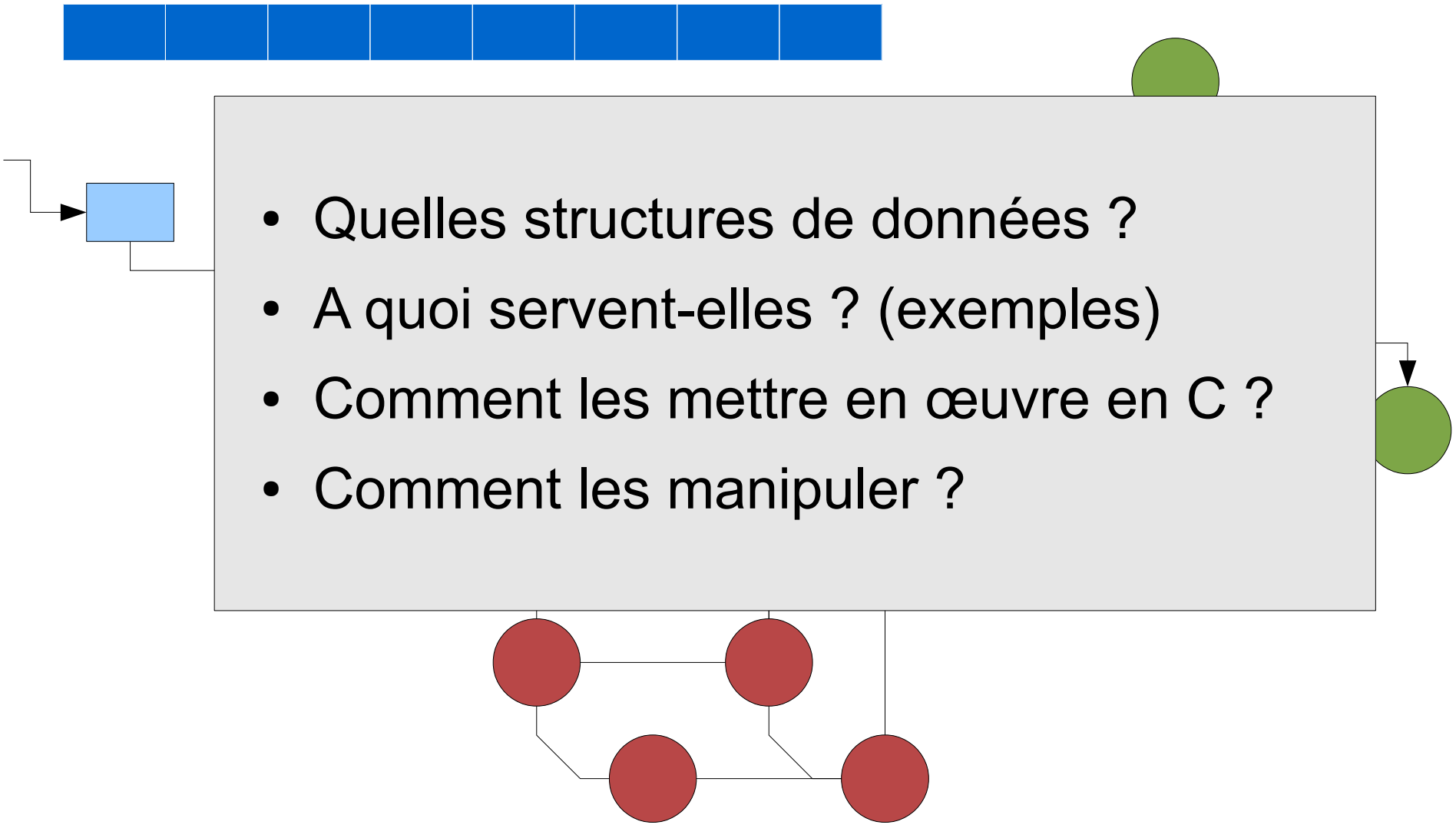
Structures de données







Structures de données



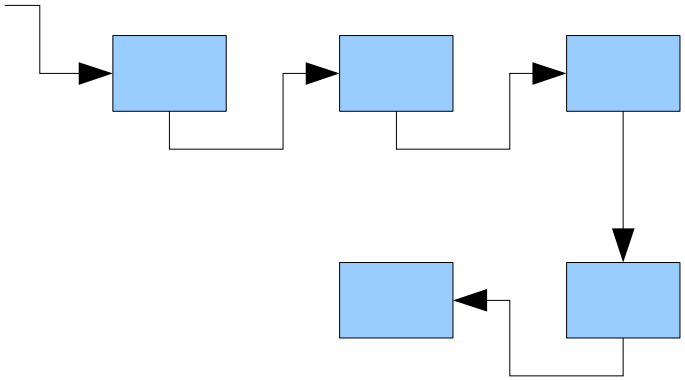
- Quelles structures de données ?
- A quoi servent-elles ? (exemples)
- Comment les mettre en œuvre en C ?
- Comment les manipuler ?

Aujourd'hui

Structures de données

* Structures linéaires
  * Tableaux
  * Listes chaînées
  * Piles et queues
* Structures non-linéaires
  * Les graphes
  * Les arbres

Structures linéaires



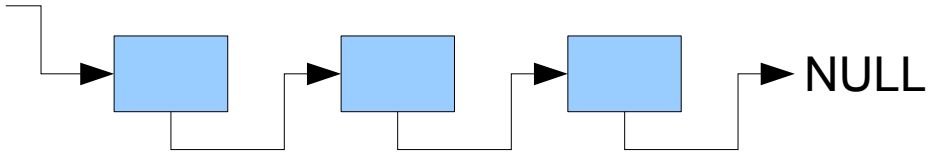
« Linéaire » = éléments en ligne

Exemples : tableaux, listes chainées, piles, queues

Les tableaux

* Avantages
  * Accès à l'élément i en temps constant : *(t+i)
* Inconvénients
  * Taille fixée à la création (allocation statique ou dynamique)
  * Tableau ordonné : insertion/suppression coûteuses
* Exemple d'utilisation : chaînes de caractères

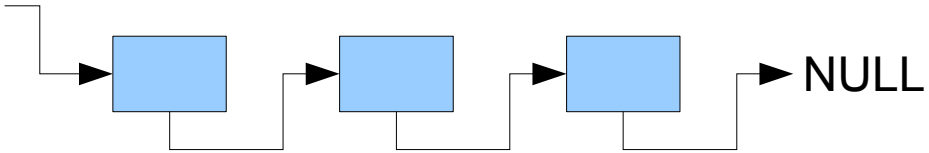
Les listes



* Avantages
  * Insertion/suppression faciles
  * Taille variable
* Inconvénients
  * Accès à l'élément i en temps linéaire (parcours des i premiers maillons)

Mise en œuvre en C ?

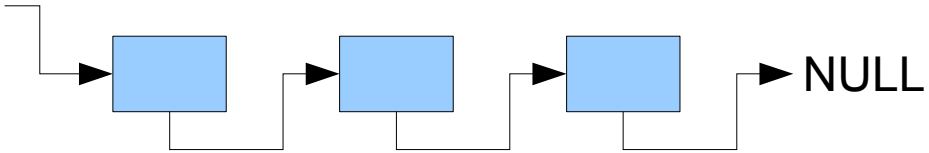
Définition



Définition d'une liste sur un ensemble X d'objets

* L'ensemble vide est une liste
* Si x est un élément de X et L est une liste sur X, alors (i,L) est une liste

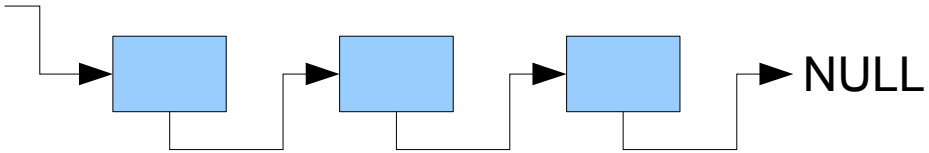
Structures auto-référentes



Une mauvaise idée :

```
typedef struct maillon
{
    OBJET info;
    struct maillon suiv;
} MAILLON;
```

Structures auto-référentes

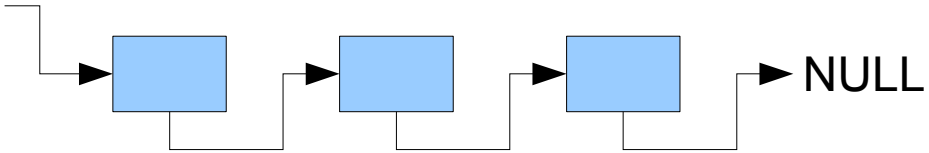


Une mauvaise idée :

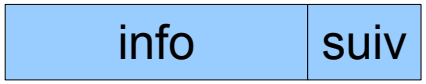
```
typedef struct maillon
{
    OBJET info;
    struct maillon suiv;
} MAILLON;
```

Impossible de déterminer la taille d'un maillon

Structures auto-référentes




```
typedef struct maillon
{
    OBJET info;
    struct maillon *suiv;
} MAILLON;
```



Taille d'un maillon :

```
sizeof(OBJET)+sizeof(struct maillon*)
```

Remarque : surcoût du stockage d'un pointeur dans chaque maillon

Liste chaînées : mise en œuvre fonction de base

```
typedef struct maillon{
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;

POINTEUR maillon(OBJET v, POINTEUR s){
    POINTEUR p;
    p = malloc(sizeof(MAILLON));
    if (!p)
        exit(EXIT_FAILURE);
    p->info = v;
    p->suiv = s;
    return p;
}
```

Liste chaînées : liste vide

```c
typedef struct maillon{
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;
```

```c
POINTEUR maillon(OBJET v, POINTEUR s){
    POINTEUR p;
    p = malloc(sizeof(MAILLON));
    if (!p)
        exit(EXIT_FAILURE);
    p->info = v;
    p->suiv = s;
    return p;
}
```

```c
POINTEUR liste_vide(void){
    OBJET v;
    return maillon(v, NULL);
}

int est_vide(POINTEUR p){
    return !p->suiv;
}
```

Technique du « maillon bidon » : le maillon de tête ne porte pas d'information utile.

Remarque : une autre définition consiste à définir la liste vide par NULL. Elle est plus simple en apparence, mais pose des problèmes dans la manipulation des listes (cf. « le célèbre maillon bidon », poly 2 p. 6)

Liste chaînées : ajout d'élément

```c
typedef struct maillon{
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;
```

```c
POINTEUR maillon(OBJET v, POINTEUR s){
    POINTEUR p;
    p = malloc(sizeof(MAILLON));
    if (!p)
        exit(EXIT_FAILURE);
    p->info = v;
    p->suiv = s;
    return p;
}
```

```c
void ajoute_position(OBJET v, int position, POINTEUR p)
{
    if (position==0 || est_vide(p))
        p->suiv = maillon(v,p->suiv);
    else
        ajoute_position(v, position-1, p->suiv);
}
```

Liste chaînées : suppression d'élément

```c
typedef struct maillon{
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;
```

```c
POINTEUR maillon(OBJET v, POINTEUR s){
    POINTEUR p;
    p = malloc(sizeof(MAILLON));
    if (!p)
        exit(EXIT_FAILURE);
    p->info = v;
    p->suiv = s;
    return p;
}
```

```c
void libere_maillon(POINTEUR p){ free(p); }
OBJET supprime_position(int position, POINTEUR p){
    POINTEUR m; OBJET v;
    if (position==0){
        v = p->info;
        m = p->suiv;
        p->suiv = m->suiv;
        libere_maillon(m);
        return v; }
    else
        return supprime_position(position-1, p->suiv);
}
```

Liste chaînées : parcours et effet

Effet sur chaque élément de la liste : fonction générique et exemple

```
void parcours_liste(POINTEUR p, void (*f)(OBJET)){
    if (!est_vide(p)) {
        f(p->suiv->info);
        parcours_liste(p->suiv,f);
    }
}
void affiche_info(OBJET v){...}

void affiche_liste(POINTEUR p){
    parcours_liste(p,affiche_info);
}
```

Insertion dans une liste triée

Liste triée :

si `p->suiv` existe, `p->info < p->suiv->info`

Insertion dans une liste triée

```
void insert_dans_liste(
    OBJET v,
    POINTEUR p,
    int (*cmp)(OBJET,OBJET))
{
     while (!est_vide(p) && cmp(p->suiv->info,v)>0) {
          p = p->suiv;
     }
     p->suiv = maillon(v,p->suiv);
}
```

Aujourd'hui

Structures de données

* Structures linéaires
  * Tableaux
  * Listes chaînées
  * Piles et queues
* Structures non-linéaires
  * Les graphes
  * Les arbres

Les piles et les queues

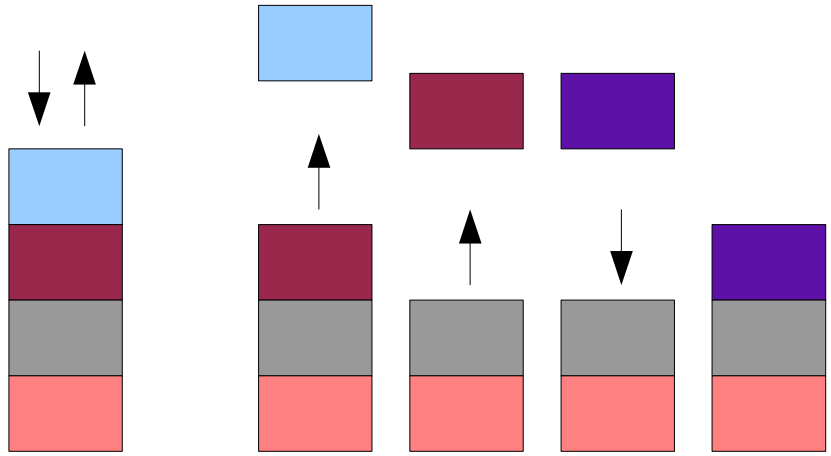
Des structures de données

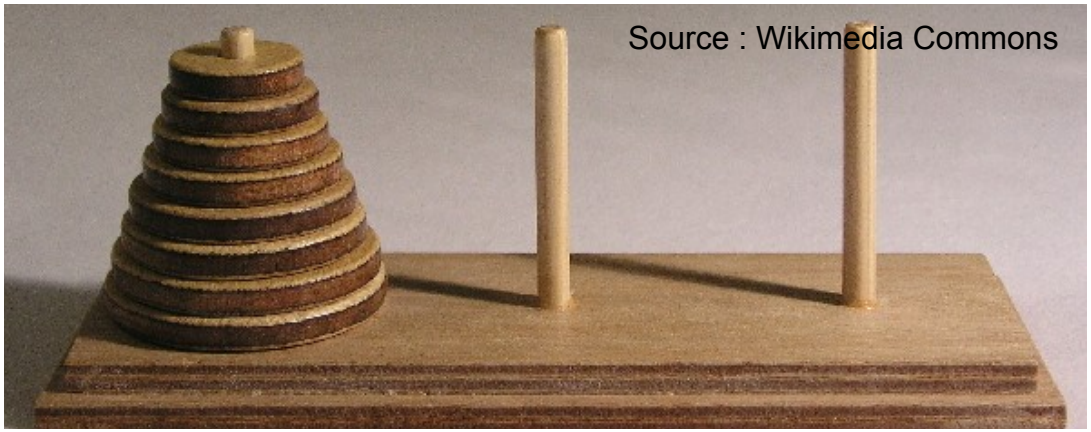
- **dynamiques** : insertion, suppression
- **abstraites** : décrites selon leur comportement, pas selon leur mise en œuvre (plusieurs mises en œuvre possibles via des tableaux ou des listes)

Les piles

Structure « LIFO » (Last In First Out)

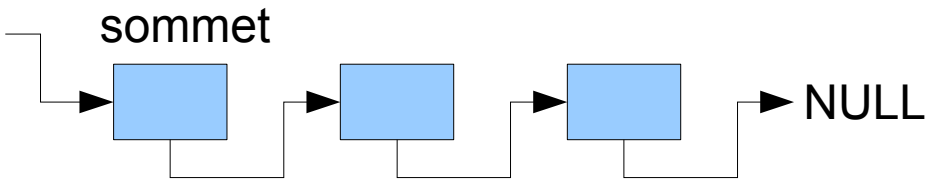
- Comportement : empiler, dépiler
- Exemple d'utilisation :
  - Tour de Hanoï (jeu)
  - Chemin dans un labyrinthe
  - Pile d'exécution (prochain cours)





Source : Wikimedia Commons

Mise en œuvre avec les listes

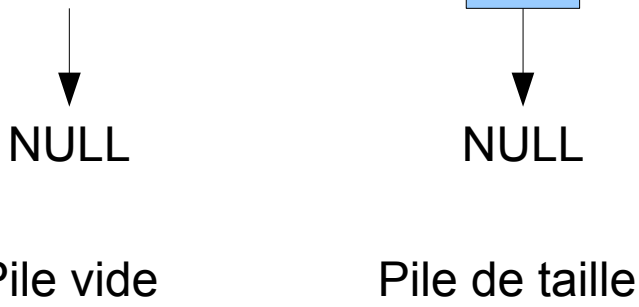



```
typedef struct maillon{
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;

POINTEUR pile;

void initialiser(void){
    pile = NULL; }

void empiler(OBJET x){
    pile = maillon(x, pile);
}
```

```
OBJET depiler(void){
    POINTEUR p;
    OBJET r;
    if (!pile)
       exit(EXIT_FAILURE);
    p = pile;
    pile = pile->suiv;
    r = p->info;
    free(p);
    return r;
}
```

Sans/Avec maillon bidon

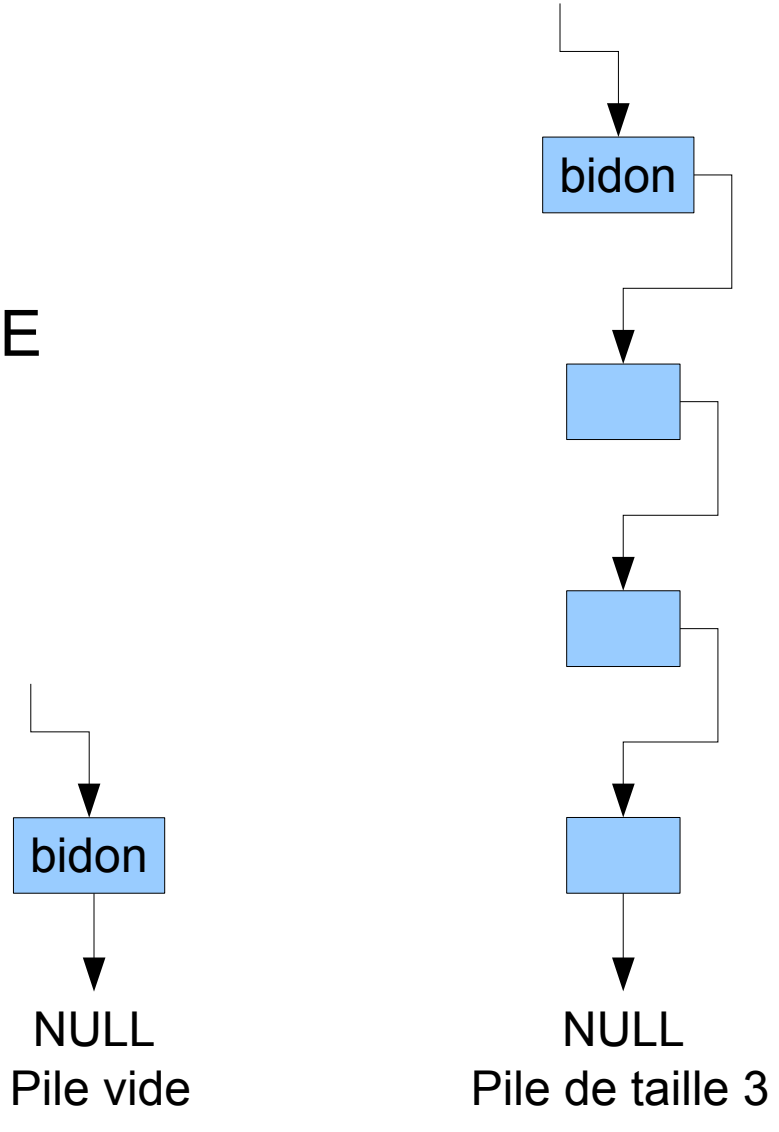
SAN
S



Pile vide

Pile de taille 3

AVE
C



NULL
Pile vide

NULL
Pile de taille 3

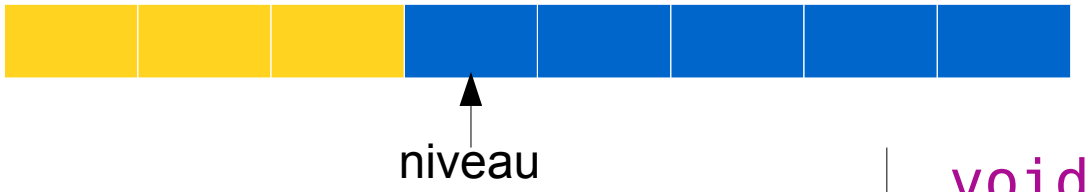
Intérêt ici : la mise en œuvre avec maillon bidon permet de passer la pile en argument, donc de manipuler plusieurs piles.

Sans/Avec maillon bidon

```
POINTEUR pile;

void initialiser(void){
 pile = NULL;
}
void empiler(OBJET x){
 pile = maillon(x,pile);
}
OBJET depiler(void){
 POINTEUR p;
 OBJET r;
 if (!pile)
  exit(EXIT_FAILURE);
 p = pile;
 pile = pile->suiv;
 r = p->info;
 free(p);
 return r;
}
```

```
void initialiser(POINTEUR pile){
 OBJET o;
 pile = maillon(o,NULL);
}
void empiler(OBJET x,
             POINTEUR pile){
 pile->suiv =
        maillon(x,pile->suiv);
}
OBJET depiler(POINTEUR pile){
    POINTEUR p;
    OBJET r;
    if (!pile->suiv)
       exit(EXIT_FAILURE);
    p = pile->suiv;
    pile->suiv = p->suiv;
    r = p->info;
    free(p);
    return r;
}
```

Mise en œuvre avec les tableaux



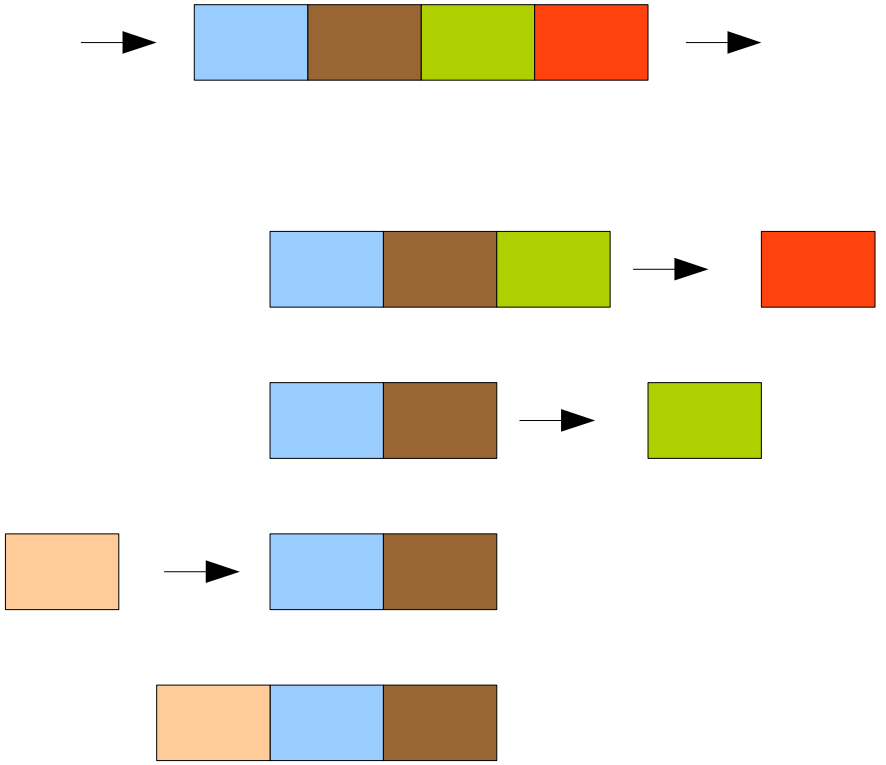

```
#define MAXPILE ...
typedef ... OBJET;

OBJET pile[MAXPILE];
int niveau;

void initialiser(void)
{
    niveau = 0;
}
```

```
void empiler(OBJET x)
{
    if (niveau >= MAXPILE)
        exit(EXIT_FAILURE);
    pile[ niveau++ ] = x;
}

OBJET depiler(void)
{
    if (niveau<1)
        exit(EXIT_FAILURE);
    return pile[ --niveau ];
}
```

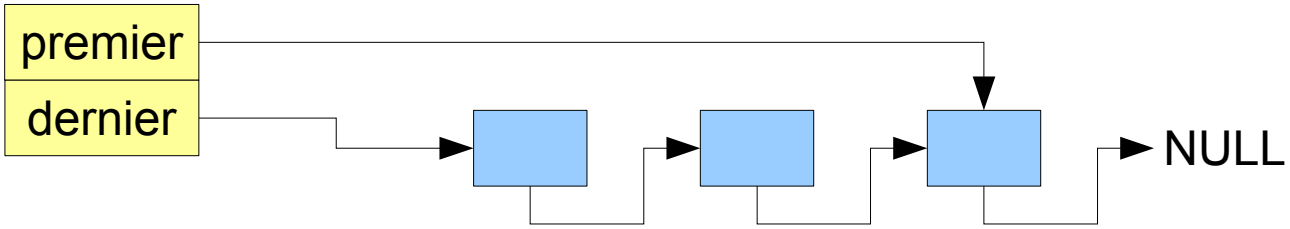
Les queues

Structure « FIFO » (First In First Out)

- Comportement : entrer, sortir
- Exemple d'utilisation :
  - File d'attente (central téléphonique, gestionnaire d'imprimante, files de priorité)
  - Tampons (*buffers*) : streaming, acquisition audio/vidéo



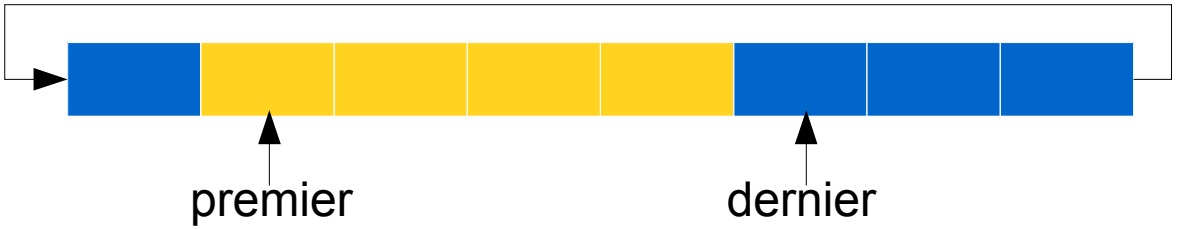
Mise en œuvre / listes

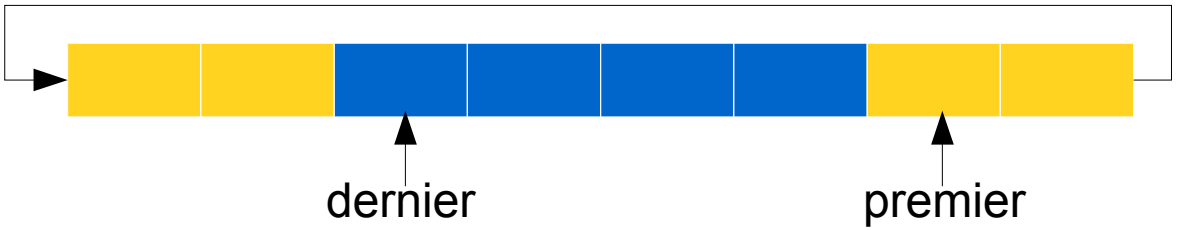



```
typedef struct maillon{
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;

POINTEUR premier, dernier;
```

cf. mise en œuvre des fonctions p.8-9 du poly « Structures de données »

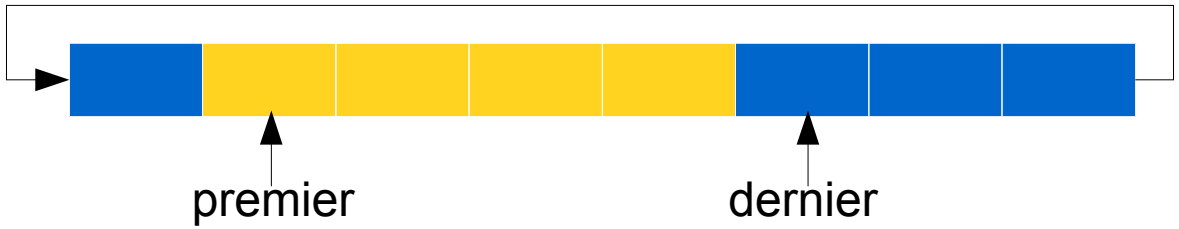
Mise en œuvre / tableaux circulaires





Mise en œuvre / tableaux circulaires

```
typedef ... OBJET;
#define MAXQUEUE 10

OBJET queue[MAXQUEUE];
int premier, dernier, nombre;
```




```
void initialiser(void){
  nombre = 0 ;
  premier = dernier = 0;
}

void entrer(OBJET x){
  if(nombre >= MAXQUEUE)
    exit(EXIT_FAILURE);
  nombre++;
  queue[dernier] = x;
  dernier = (dernier+1)%MAXQUEUE;
}
```

```
OBJET sortir(void){
  OBJET r;
  if (nombre < 1)
  exit(EXIT_FAILURE);
  nombre--;
  r = queue[premier];
  premier = (premier+1)%MAXQUEUE;
  return r;
}
```

Aujourd'hui

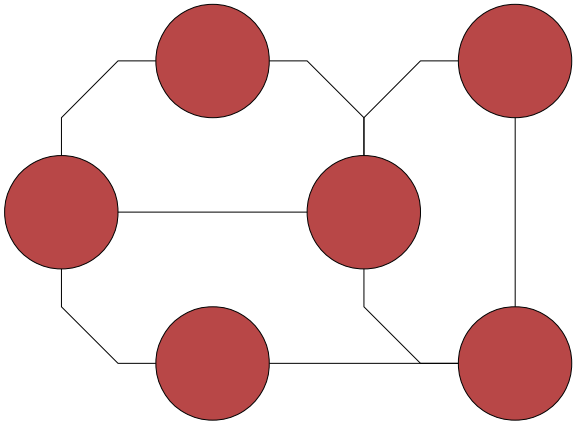
Structures de données

* Structures linéaires
  * Tableaux
  * Listes chaînées
  * Piles et queues
* Structures non-linéaires
  * Les graphes
  * Les arbres

Les graphes

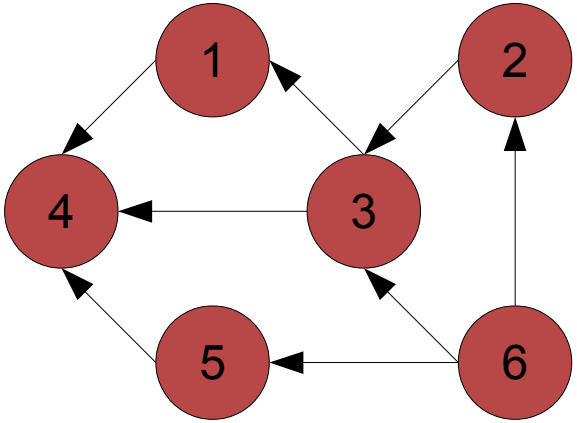
Définition :

Un graphe orienté G=(V,E) est caractérisé par un ensemble de sommets V et un ensemble d'arêtes $E \subset V \times V$.



Un graphe est non-orienté si pour tout $(x,y) \in E$, $(y,x) \in E$

Exemple :

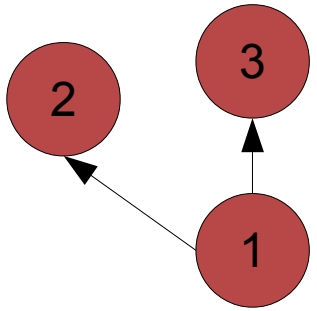


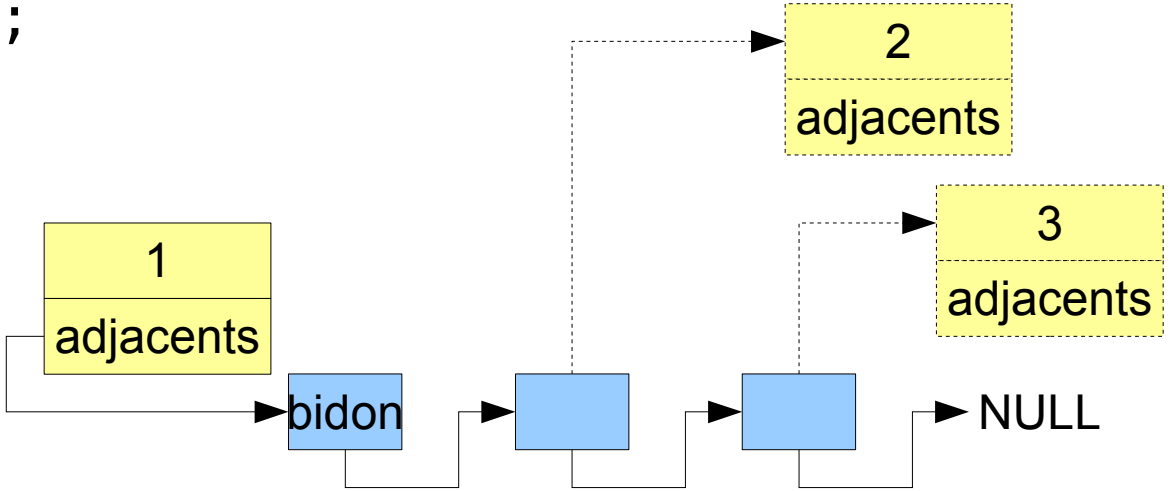
V={1,2,3,4,5,6} ; E={(1,4), (3,1), (2,3), (3,4), (5,4), (6,3), (6,2), (6,5)}

Utilisation : automates, plans, etc.

Mise en œuvre / noeuds

```
typedef struct noeud{
    OBJET info;
    struct liste_noeud *adjacents;
} *NOEUD;

typedef struct liste_noeud{
    NOEUD noeud;
    struct liste_noeud *suiv;
} MAILLON, *LISTE_NOEUD;
```





NB : via des pointeurs sur des structures, on peut définir des types se faisant mutuellement référence.

Mise en œuvre / noeuds

```
typedef struct noeud{OBJET info; struct liste_noeud *adjacents;} *NOEUD;
typedef struct liste_noeud{NOEUD noeud;struct liste_noeud *suiv;} MAILLON,
*LISTE_NOEUD;
LISTE_NOEUD maillon(NOEUD v, LISTE_NOEUD s){
    LISTE_NOEUD p;
    p = malloc(sizeof(MAILLON));
    if (!p) exit(EXIT_FAILURE);
    p->noeud = v;
    p->suiv = s;
    return p;
}
LISTE_NOEUD liste_vide(void){
    NOEUD n=NULL;
    return maillon(n, NULL);
}

void ajoute_noeud(NOEUD n, LISTE_NOEUD liste){
    liste->suiv = maillon(n,liste->suiv);
}
```

NON VU EN COURS

Mise en œuvre / Matrice

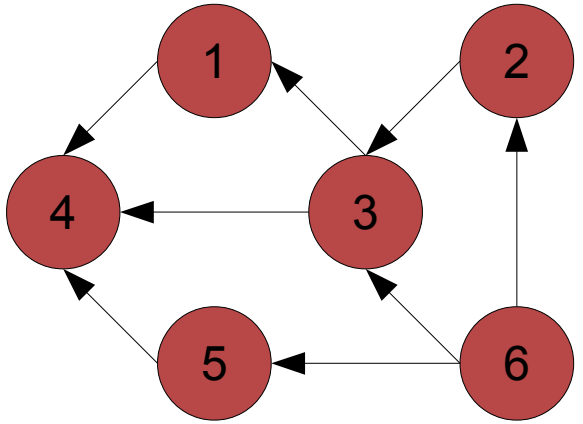
Plusieurs mises en œuvre matricielles possible.

Exemple : matrice d'adjacence

M_{ij} = 1 si (i,j) est une arête,

M_{ij} = 0 sinon

$$M = \begin{matrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{matrix}$$



Propriétés

- Le graphe est non orienté ssi M est symétrique.
- Puissance de M : $[M^k]_{ij}$ = nb de chemins de longueur k allant de i à j.

Aujourd'hui

Structures de données

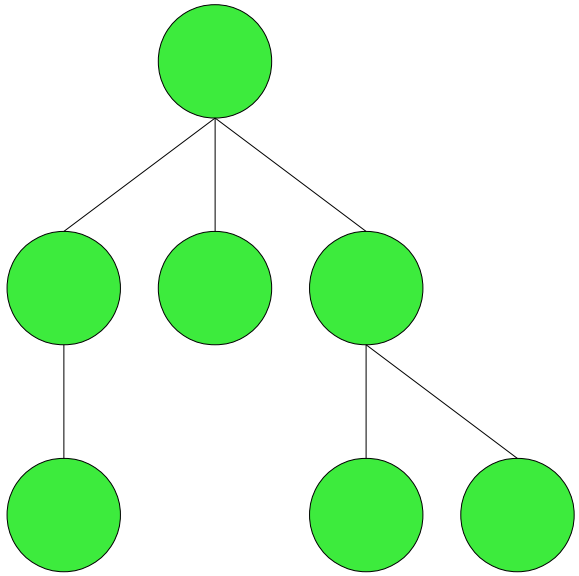
* Structures linéaires
  * Tableaux
  * Listes chaînées
  * Piles et queues
* Structures non-linéaires
  * Les graphes
  * Les arbres

NON VU EN COURS

Les arbres : définition générale

Un *arbre* est un graphe non orienté $G = (E,V)$ qui vérifie les propriétés équivalentes suivantes :

- G est connexe minimal (si on lui enlève une arête il n'est plus connexe) ;
- G est acyclique maximal (si on lui ajoute une arête on crée un cycle) ;
- $|V|=|E|-1$ et G est connexe ou acyclique;
- deux sommets quelconques de G sont reliés par un unique chemin élémentaire.



NB : connexe = tous les sommets (ou nœuds) sont connectés par un chemin

acyclique = il n'existe pas de chemin élementaire dont les extrémités sont identiques

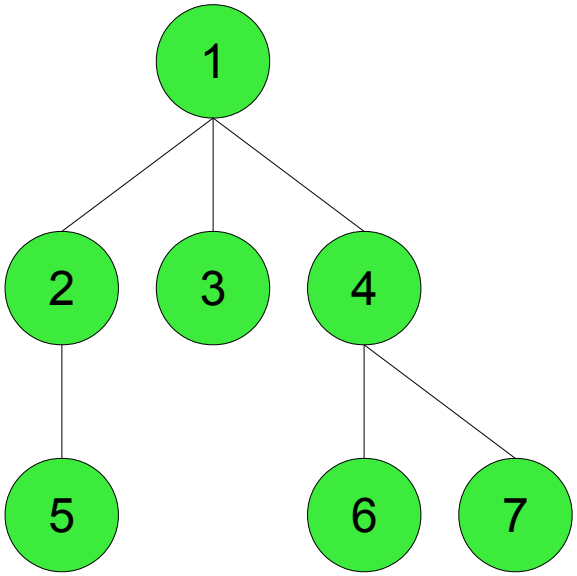
NON VU EN COURS

Arborescence

Une *arborescence* est un ensemble fini non vide *A* muni de la structure suivante :

• il existe un élément distingué, appelé la *racine* de *A* ;

• l'ensemble des autres éléments de *A*, s'il n'est pas vide, est *partitionné* en une famille de sous-ensembles qui sont des arborescences.



On ne fera pas la distinction entre un arbre et une arborescence.

Vocabulaire :

- *noeud*=sommet
- 5, 6, 7 sont les *feuilles*
- 1 est *père* de 2, 3 et 4 ; 2 est le *père* de 5
- 2, 3, 4 sont les *fils* de 1 ; 5 est le *fils* de 2

NON VU EN COURS

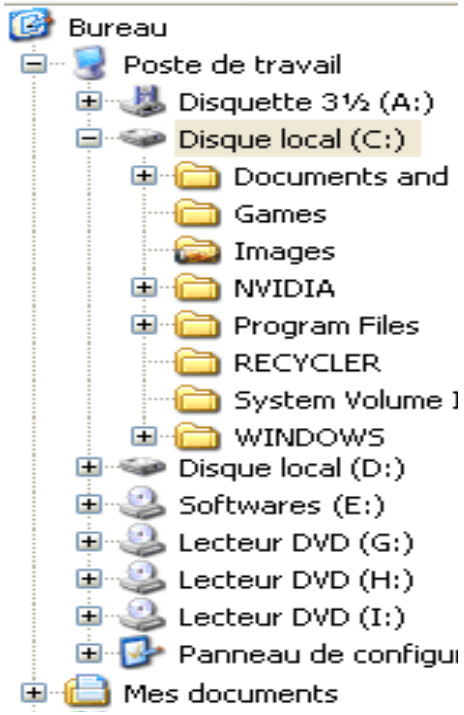
Exemple d'utilisation

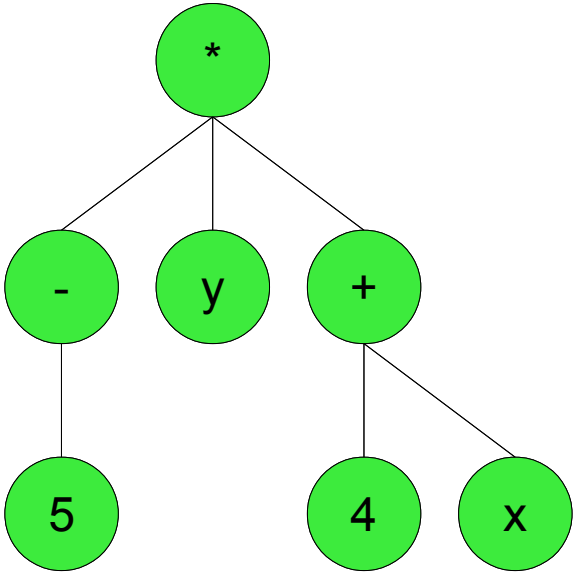
Recherche rapide : dictionnaire, indexation, complétion automatique

Codage, compression (zip, jpg, mp3)

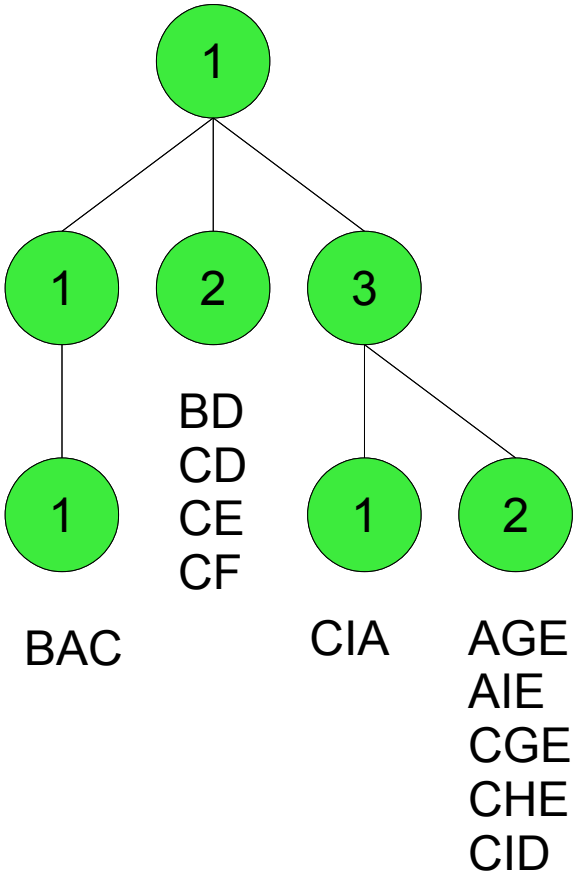
Analyse syntaxique (ex : -5*y*(4+x))

Système de fichiers





Ecriture SMS :
1=ABC
2=DEF
3=GHI

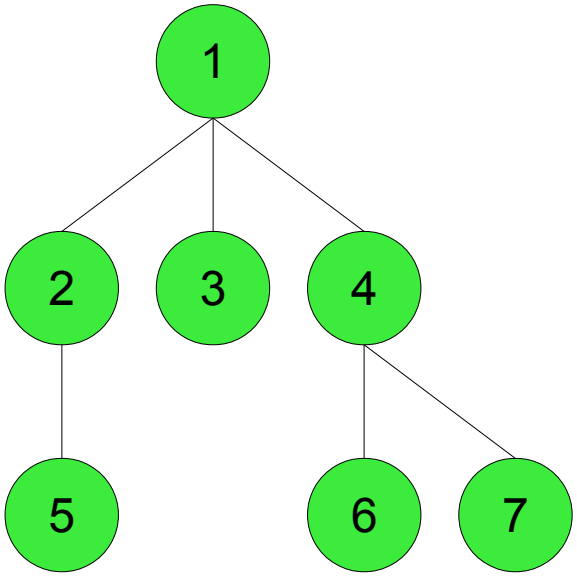


NON VU EN COURS

Mise en oeuvre

Version rigoureuse :

```
typedef struct arbre{
  OBJET info;
  LISTE_D_ARBRES fils;
} ARBRE;

typedef struct maillon{
  ARBRE sousArbre;
  struct maillon *frere;
} *LISTE_D_ARBRES;
```

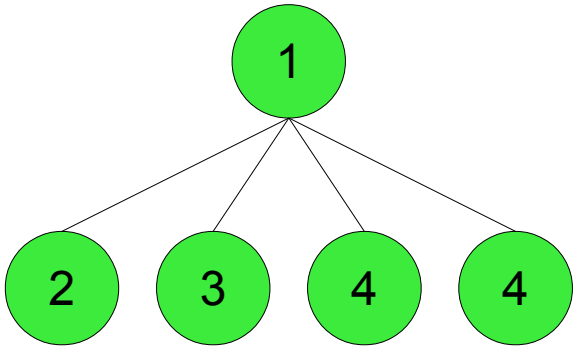


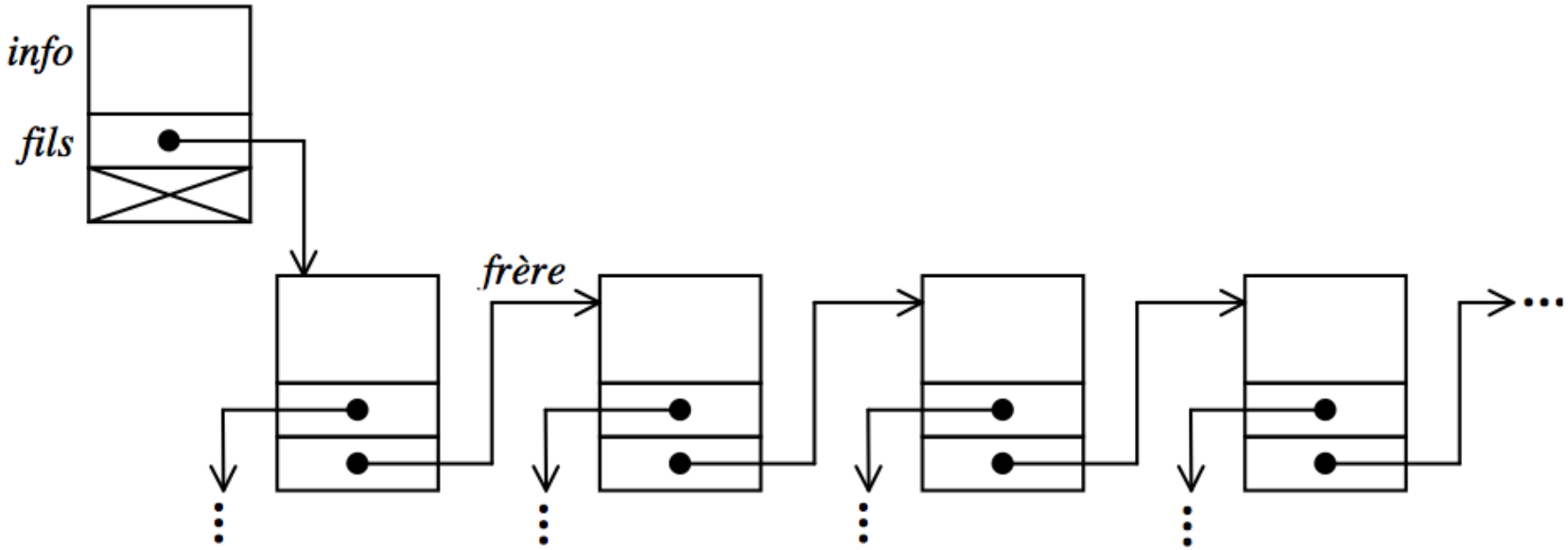
NON VU EN COURS

Mise en oeuvre

Version courte :

```
typedef struct noeud{
  OBJET info;
  struct noeud *fils;
  struct noeud *frere;
} NOEUD, *POINTEUR;
```



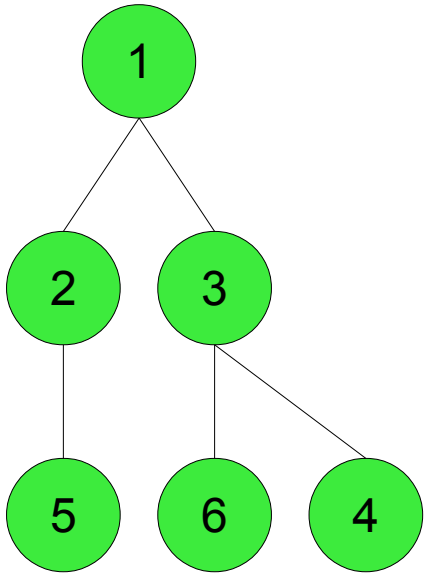


NON VU EN COURS

Arbres binaires

= chaque nœud a au plus 2 fils.

```
typedef struct noeud{
  OBJET info;
  struct noeud *gauche, *droite;
} NOEUD, *POINTEUR;
```



NON VU EN COURS

Parcours d'arbres binaires

Parcours « pré-ordre » :

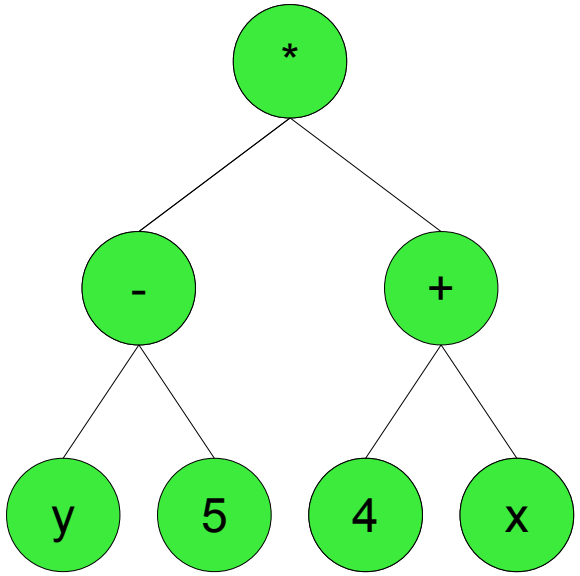
```
void parcours_pre(POINTEUR arbre){
  if (arbre != NULL){
    EXPLOITER(arbre->info);
    parcours(arbre->gauche);
    parcours(arbre->droite);
  }
}
```

Parcours « in-ordre » :

```
...
    parcours(arbre->gauche);
    EXPLOITER(arbre->info);
    parcours(arbre->droite);
...
```

Parcours « post-ordre » :

```
...
    parcours(arbre->gauche);
    parcours(arbre->droite);
    EXPLOITER(arbre->info);
...
```



Exemple : parcours avec une fonction d'affichage d'un nœud

Pré-ordre : notation polonaise

*-y5+4x

In-ordre : notation infixe

((y)-(5))*((4)+(x))

(y-5)*(4+x)

Post-ordre : notation polonaise inversée

y5-4x+*

Aujourd'hui

Structures de données

* Structures linéaires
  * Tableaux
  * Listes chaînées
  * Piles et queues
* Structures non-linéaires
  * Les graphes
  * Les arbres