

# *Algorithmique*

# *Programmation Objet*

## *Python*

---



**Andrea G. B. Tettamanzi**

Université de Nice Sophia Antipolis

Département Informatique

[andrea.tettamanzi@unice.fr](mailto:andrea.tettamanzi@unice.fr)

## *CM - Séance 11*



# **Arbres et Graphes**



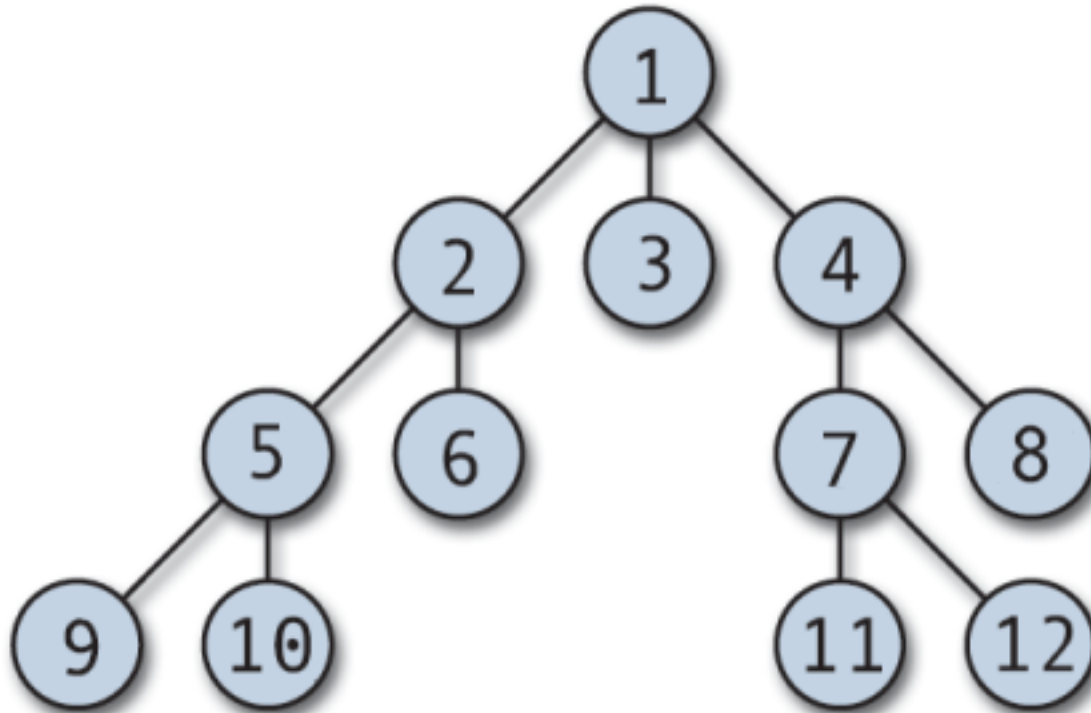
# *Plan*

- Arbres
  - Définitions
  - Parcours
  - Réalisation
- Graphes

# Arbre

- Structure de données récursive
- Un arbre est formé par
  - Un nœud (dit « racine »), contenant
    - Des données ou une référence à des données
    - Des références (ou pointeurs) à des (sous-)arbres
  - Zéro ou plus sous-arbres
- Un nœud n'ayant pas des sous-arbres est dit « feuille »
- Les autres nœuds sont dits « nœuds internes »
- Nous avons déjà rencontré ce type de structure quand on a étudié les tas

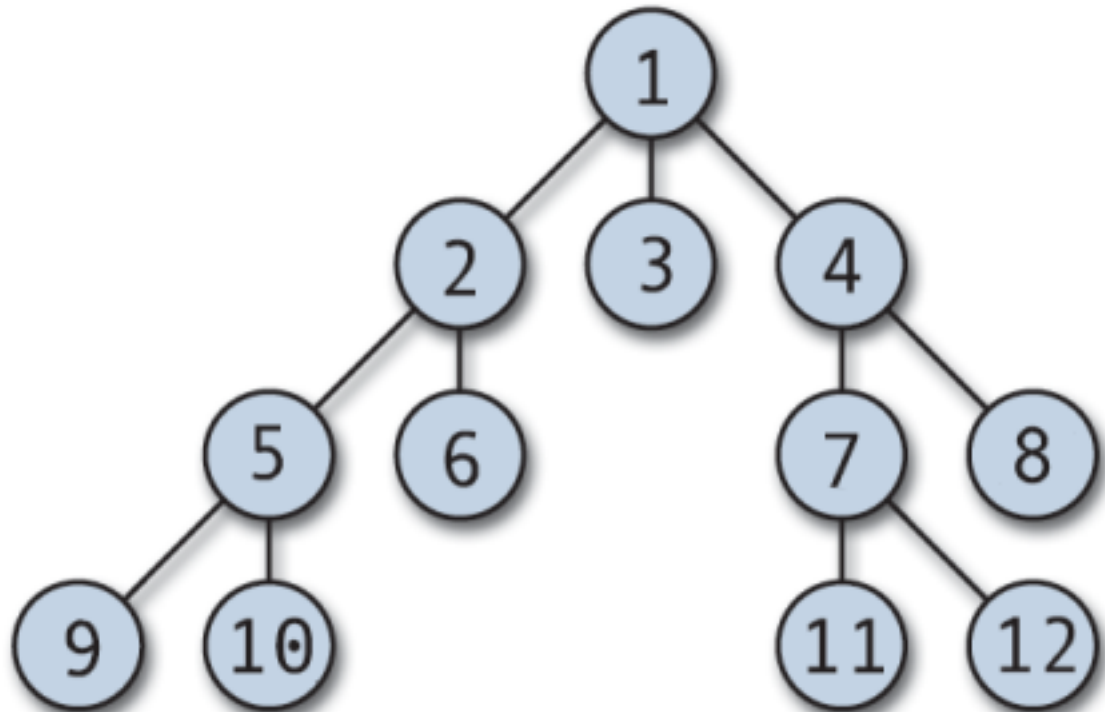
# Arbre



# Arbre

- La racine  $r$  de l'arbre est l'unique nœud ne possédant pas de parent
- Tout nœud  $x$  qui n'est pas la racine a
  - un unique parent, noté  $x.\text{parent}$  ou  $\text{parent}(x)$  (appelé « père » parfois)
  - 0 ou plusieurs fils ;  $x.\text{fils}$  ou  $\text{fils}(x)$  désigne l'ensemble des fils de  $x$
- Si  $x$  et  $y$  sont des nœuds tels que  $x$  soit sur le chemin de  $r$  à  $y$ ,
  - $x$  est un ancêtre de  $y$
  - $y$  est un descendant de  $x$
- Les feuilles n'ont pas de fils

# Arbre



1 est la racine

9,10,6,3,11,12,8 sont les feuilles

11 est un descendant de 4, mais pas de 2

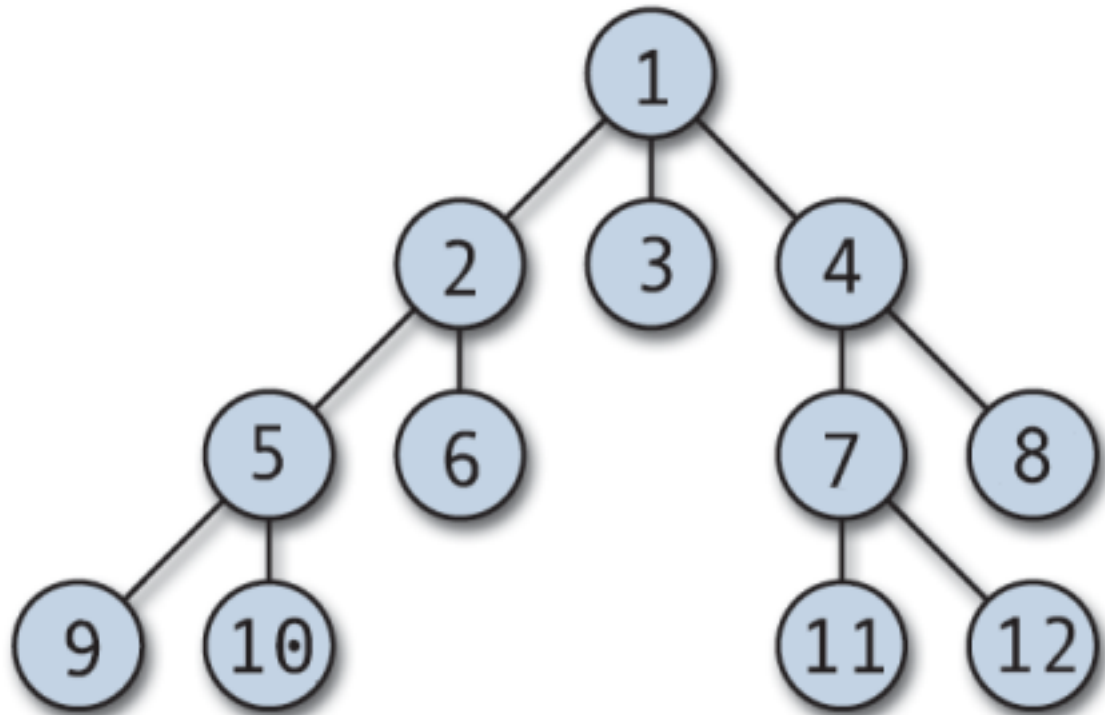
2 est un ancêtre de 10

# Arbre

- Quand il n'y a pas d'ambiguïté, on regarde les arêtes d'un arbre comme étant orienté de la racine vers les feuilles
- La profondeur d'un nœud (*depth*) est définie récursivement par
  - $\text{prof}(v) = 0$ , si  $v$  est la racine
  - $\text{prof}(v) = \text{prof}(\text{parent}(v)) + 1$
- La hauteur d'un nœud (*height*) est la plus grande profondeur d'une feuille du sous-arbre dont il est la racine



# Arbre



1 est la racine

2,3,4 sont à la profondeur 1

5,6,7,8 à la profondeur 2

La hauteur de 2 est 2, celle de 9 est 0, celle de 3 est 0, celle de 1 est 3

# Arbre : Utilisation

- Les arbres sont un peu partout en Informatique
- Représenter la structure syntaxique d'expressions (*parsing*)
  - Mathématiques ou logiques
  - De langages de programmations
  - Du langage naturel
- Gérer des bases de données
- Indexation de fichiers.
- Tri par tas
  
- Ils permettent des recherches rapides et efficaces.

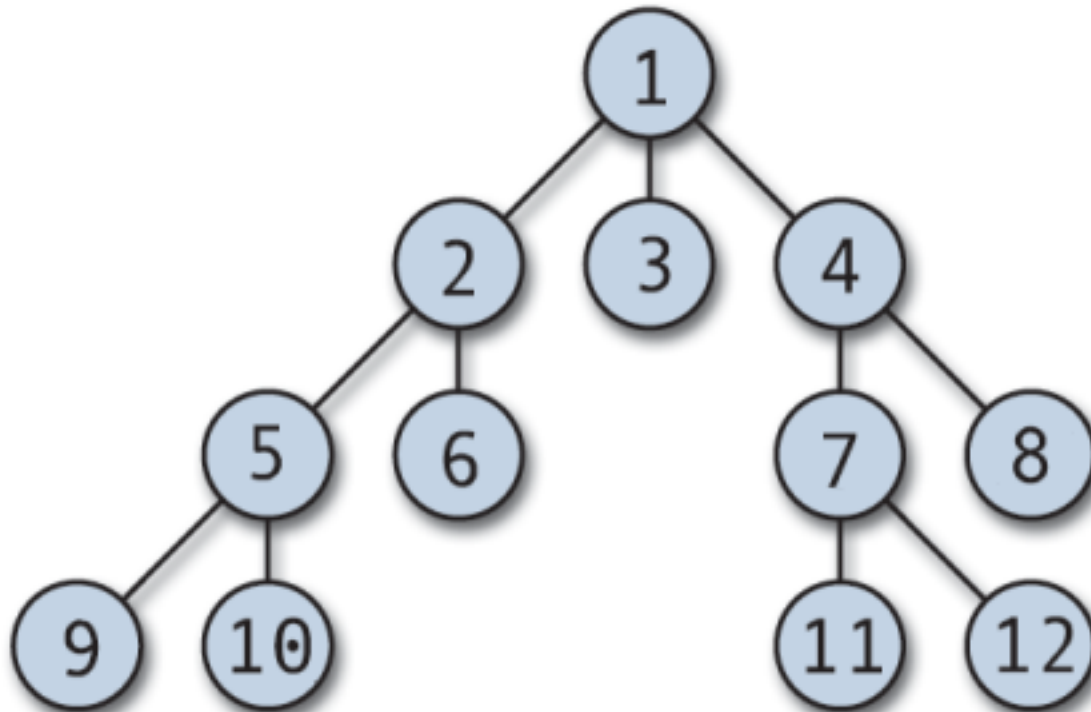
# Arbre : Parcours

- Parcours (*tree traversal*) :
  - on traverse l'ensemble des nœuds de l'arbre
- Parcours en largeur d'abord (*breadth first*)
- Parcours en profondeur d'abord (*depth first*)
  - Préfixé
  - Infixé
  - Postfixé

# *Parcours en largeur d'abord*

- On visite la racine
- Puis on répète le processus suivant jusqu'à avoir visité tous les sommets :
  - visiter un fils non visité du sommet le moins récemment visité qui a au moins un fils non visité
- On visite tous les sommets à la profondeur 1,
- puis tous ceux à la profondeur 2,
- puis tous ceux à la profondeur 3
- etc...

## *Parcours en largeur d'abord*



Largeur d'abord: ordre de visite  
1, 2 ,3 ,4 ,5, 6 , 7, 8, 9 , 10 ,11 ,12

# *Parcours en largeur d'abord*

Liste ParcoursEnLargeur()

F ← File()

F.enfiler(racine())

L ← Liste()

tant que non F.estVide()

    x ← F.premier() ; F.défiler()

    L.ajouter(x)

    pour chaque fils y de x

        F.enfiler(y)

renvoyer L

# Parcours en largeur d'abord avec passes

Liste ParcoursEnLargeurAvecPasses()

F1 ← File() ; F2 ← File()

F1.enfiler(racine())

L ← Liste()

répéter :

  tant que non F1.estVide()

    x ← F1.premier() ; F1.défiler()

    L.ajouter(x)

    pour chaque fils y de x

      F2.enfiler(y)

    échanger(F1, F2)      # Fin d'une passe, début de la suivante

  tant que non F1.estVide()

renvoyer L

# Parcours en largeur d'abord avec passes

Largeur d'abord avec passes

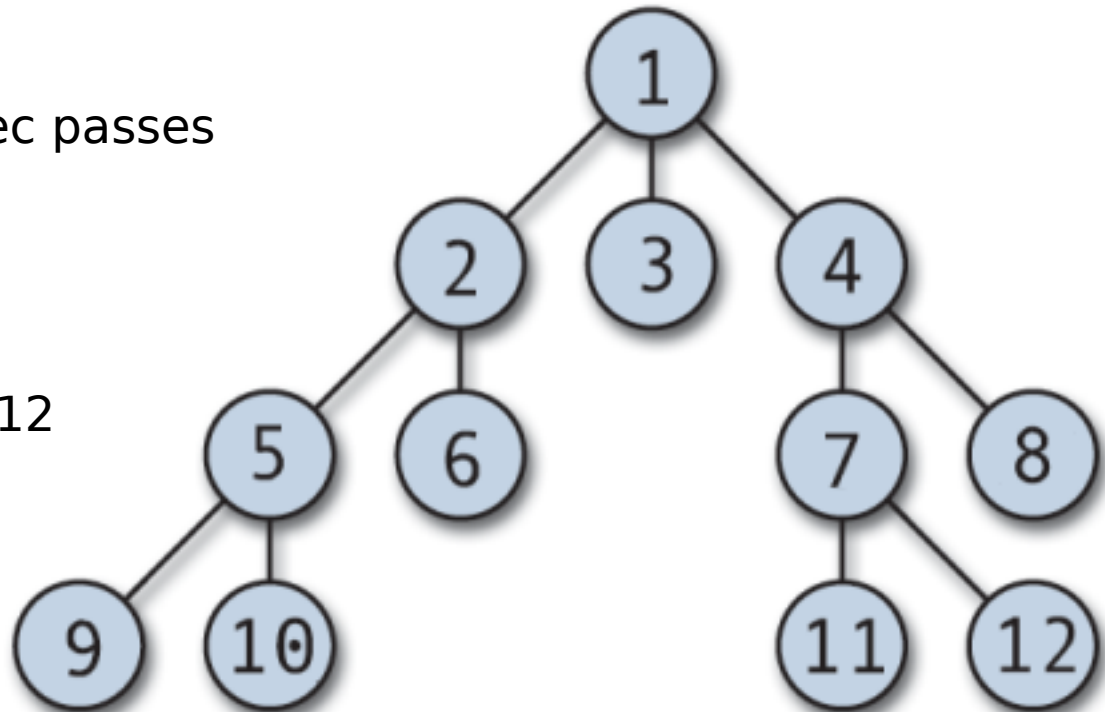
Ordre de visite :

Passé 1 : 1

Passé 2 : 2 ,3 ,4

Passé 3 : 5, 6 , 7, 8

Passé 4 : 9, 10 ,11 ,12





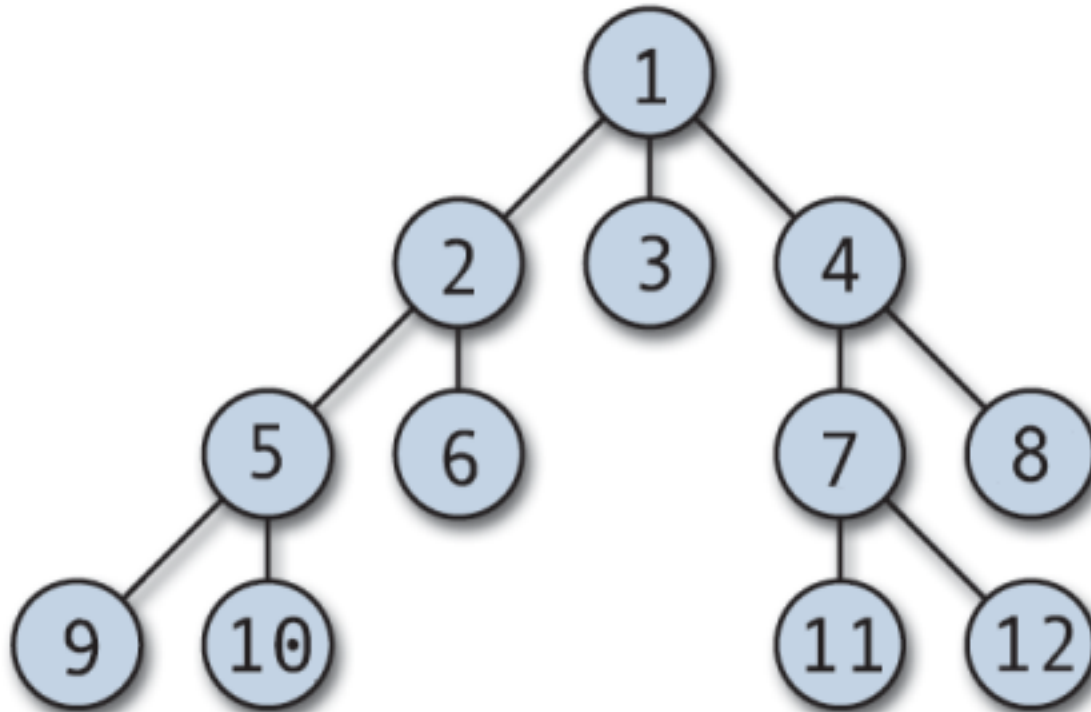
# *Parcours en profondeur d'abord*

- Défini de façon récursive
- visit(noeud x)  
  previsit(x)  
  pour chaque fils y de x  
    visit(y)  
  postvisit(x)
- Premier appel : visit(T.racine())

# *Parcours en profondeur d'abord*

- Ordre préfixé ou postfixé dépend des fonctions previsit et postvisit
- Si previsit(x) : met x dans liste
  - alors liste contient l'ordre préfixé
- Si c'est postvisit qui le fait
  - alors liste contiendra l'ordre postfixé

# Parcours en profondeur d'abord

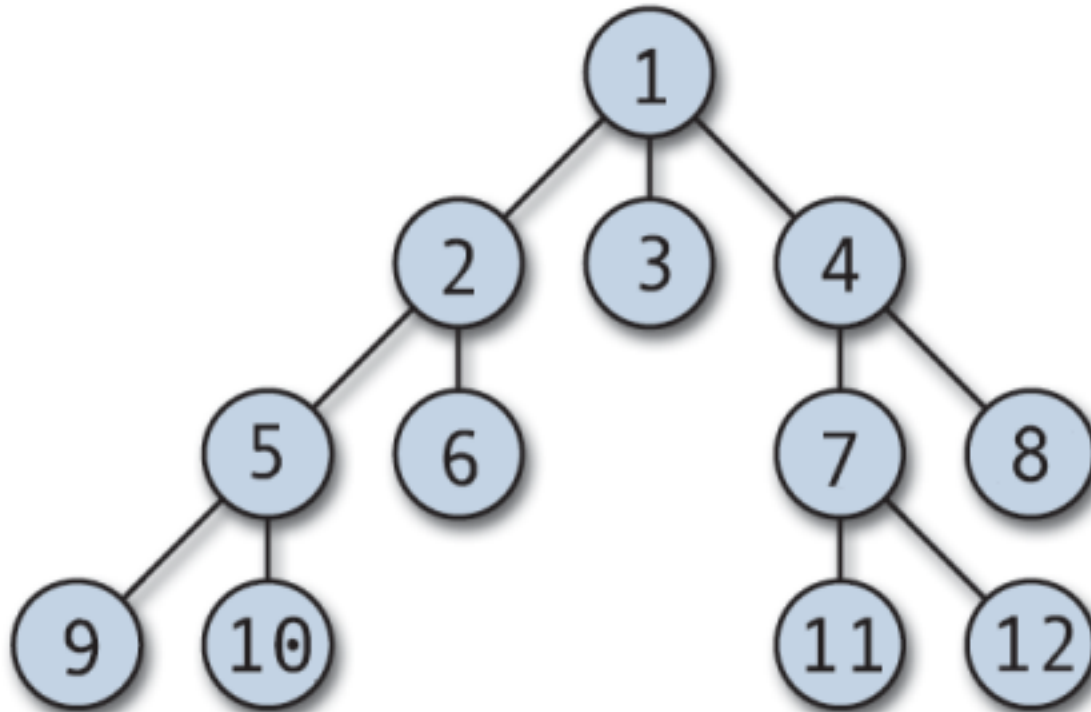


Ordre de visite préfixé

On marque quand on atteint le noeud

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

# *Parcours en profondeur d'abord*



Ordre de visite postfixé

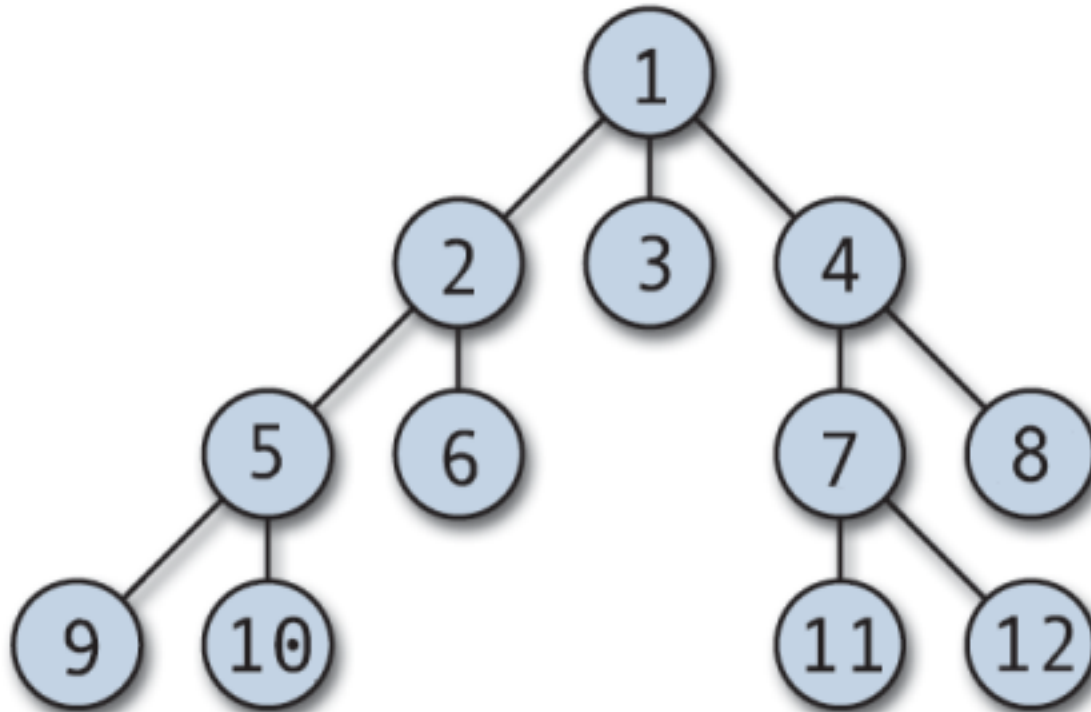
On marque quand on quitte le noeud

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

# *Parcours infixé*

- Valable uniquement pour les arbre binaires (2 fils au plus)
- visit(sommet x)  
previsit(x)  
si fils gauche y existe alors visit(y)  
invisit(x)  
si fils droit y existe alors visit(y)  
postvisit(x)
- Premier appel : visit(T.racine())
- invisit(x) place x dans la liste

## *Parcours en profondeur d'abord*



Ordre de visite infixé (on ignore 3)  
On marque entre les deux fils  
9, 5, 10, 2, 6, 1, 11, 7, 12, 4, 8

## *Profondeur d'abord itérative*

- On peut éviter d'utiliser un algorithme récursif pour représenter un parcours en profondeur d'abord
- Il faut utiliser une pile
  - Visiter un fils revient à empiler le père
  - Et visiter le fils (qui va être empilé par ces fils et ainsi de suite)
  - Remonter (terminer l'appel récursif) revient à dépiler
  - Quand il n'y a plus de fils, on reprend le sommet de la pile, on dépile et on passe au fils suivant

# Arbre : réalisation

- Représentation des fils
  - Par une liste :
    - Le parent possède un premier fils
    - Chaque sommet possède un pointeur vers son frère suivant (liste chaînée des fils)
- Par un tableau si le nombre de fils est connu à l'avance (arbre k-aire)
- Dans le cas binaire, le parent possède le fils gauche et le fils droit

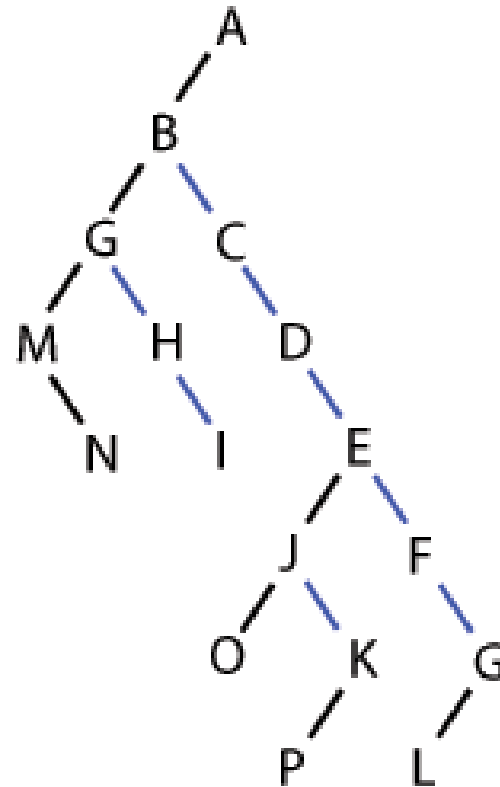
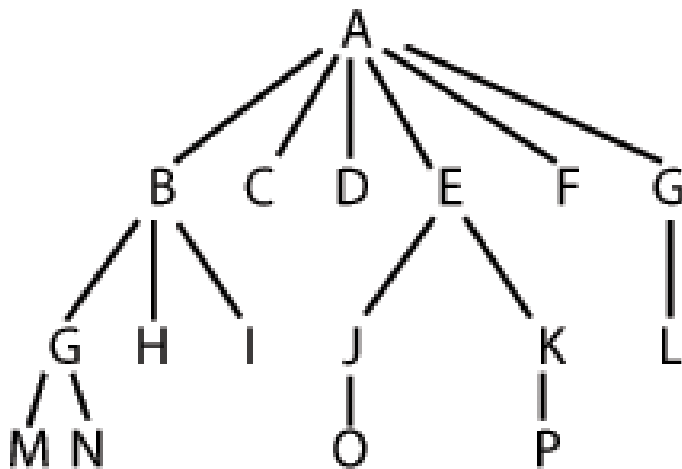


# *Arbre : réalisation*

- Arbre binaire complet :
  - on peut utiliser un tableau pour représenter tout l'arbre
  - voir exemple du tas

## Arbre non binaire

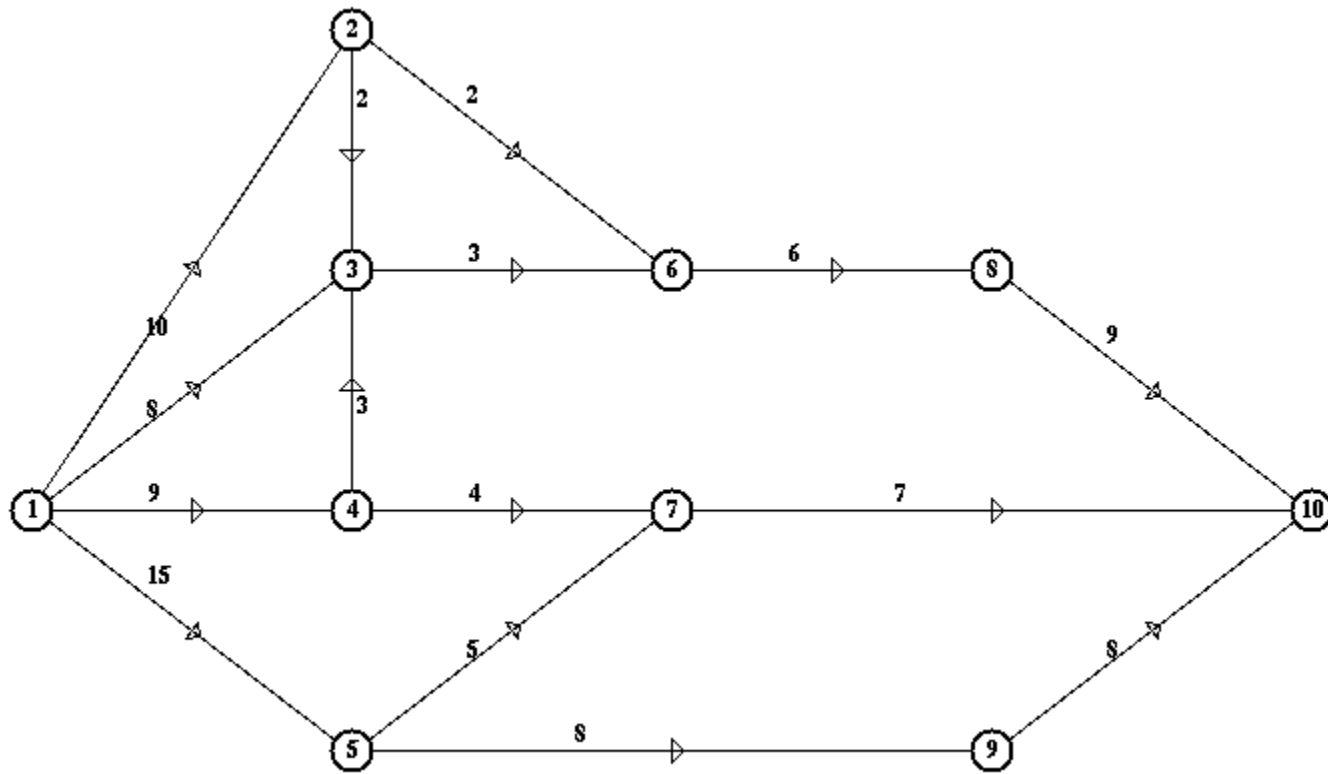
- Tout arbre non binaire peut être représenté par un arbre binaire



# Graphe orienté

- Un Graphe Orienté  $G=(X,U)$  est déterminé par la donnée :
  - d'un ensemble de sommets ou nœuds  $X$
  - d'un ensemble ordonné  $U$  de couples de sommets appelés arcs.
- Si  $u = (i, j)$  est un arc de  $G$ , alors
  - $i$  est l'extrémité initiale de  $u$
  - $j$  est l'extrémité terminale de  $u$ .
- Les arcs ont un sens (« orientés »).
  - L'arc  $u = (i, j)$  va de  $i$  vers  $j$ .
- Ils peuvent être munis d'un coût, d'une capacité etc. (arcs étiquetés)

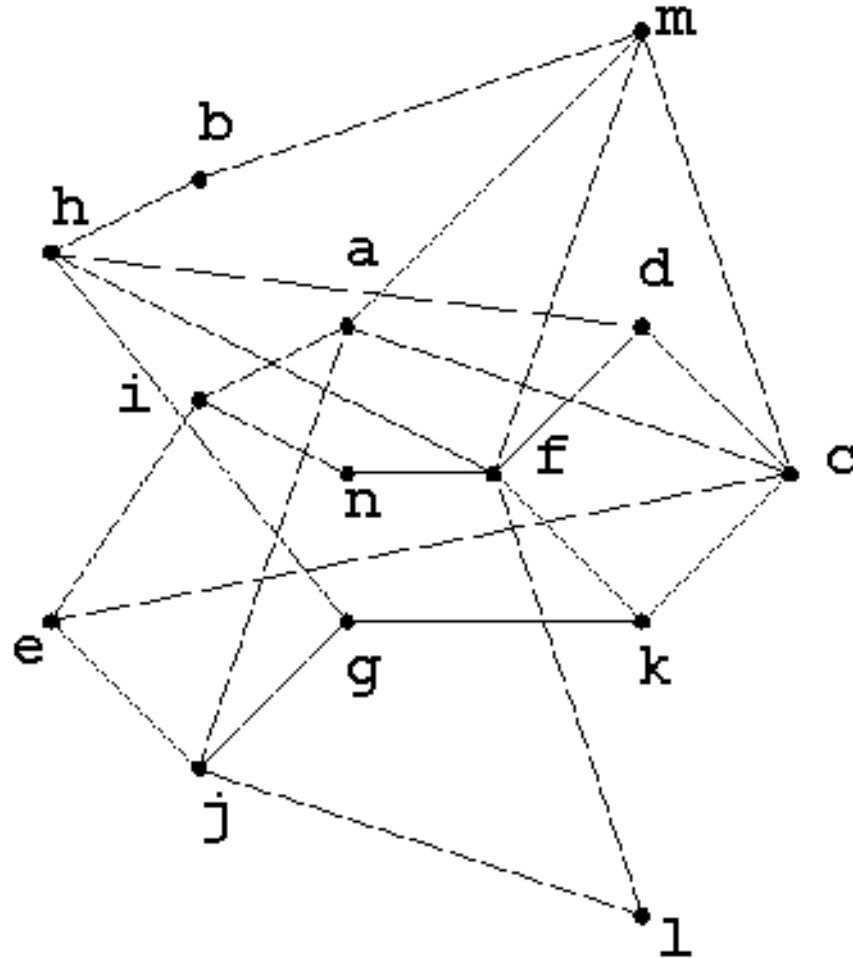
# Graphe



# *Graphe non orienté*

- Un Graphe Orienté  $G=(X,U)$  est déterminé par la donnée :
  - d'un ensemble de sommets ou nœuds  $X$
  - d'un ensemble de paires de sommets appelées « arêtes ».
- Les arêtes ne sont pas orientées

# Graphe non orienté



# Chemins et circuits

- Chemin de longueur  $q$  : séquence de  $q$  arcs  $\{u_1, u_2, \dots, u_q\}$  telle que
  - $u_1 = (i_0, i_1)$
  - $u_2 = (i_1, i_2)$
  - $u_q = (i_{q-1}, i_q)$
- Chemin : tous les arcs orientés dans le même sens
- Circuit : chemin dont les extrémités coïncident

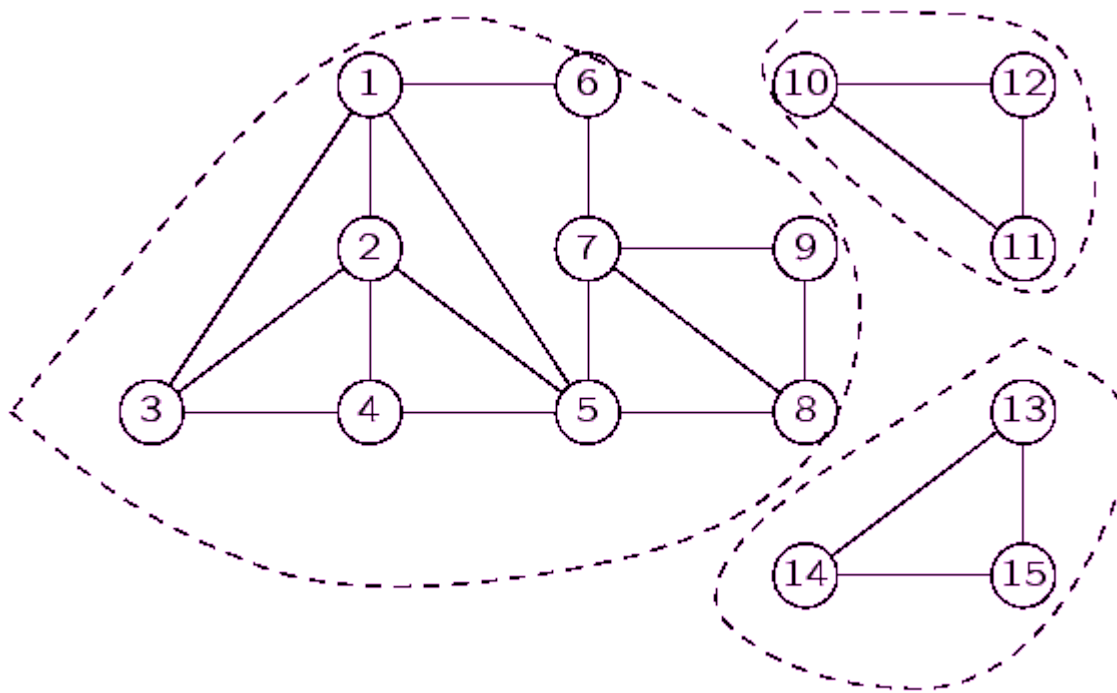
# Chaînes et cycles

- Chaîne de longueur  $q$  : séquence de  $q$  arêtes  $\{u_1, u_2, \dots, u_q\}$  telle que
  - $u_1 = (i_0, i_1)$
  - $u_2 = (i_1, i_2)$
  - $u_q = (i_{q-1}, i_q)$
- Cycle : chaîne dont les extrémités coïncident



# Connexité

- Connexité : un graphe non orienté est connexe s'il existe une chaîne entre tout paire de sommets



# Arbres comme graphes

- Un arbre est un graphe non orienté connexe et sans cycle
- Un graphe non orienté  $G$  ayant  $n$  sommets est un arbre si et seulement si il vérifie l'une des deux propriétés
  - $G$  est connexe et possède  $n - 1$  arêtes
  - $G$  n'a pas de cycle et a  $n - 1$  arêtes

*Merci de votre attention*

