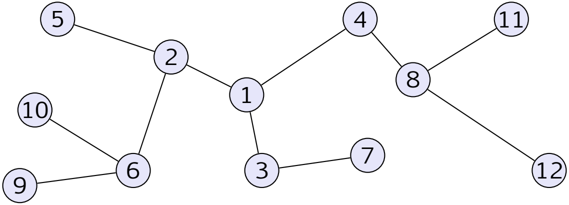
Arbres

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

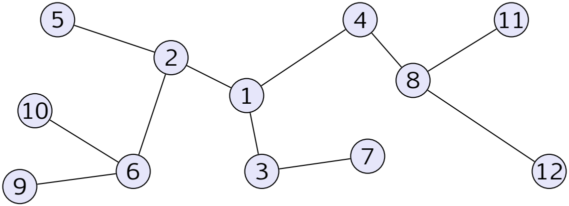
Arbres et théorie des graphes

un arbre est un graphe simple non orienté, acyclique et connexe.



Arbres et théorie des graphes

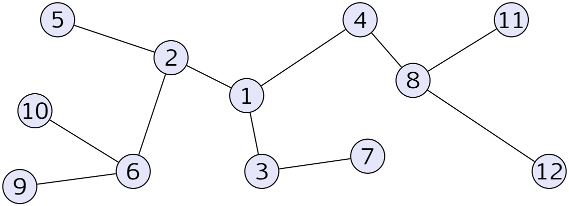
un arbre est un graphe simple non orienté, acyclique et connexe.



- simple : il n'existe pas d'arête reliant un sommet à lui-même et deux sommets distincts sont reliés par au plus une arête ;

Arbres et théorie des graphes

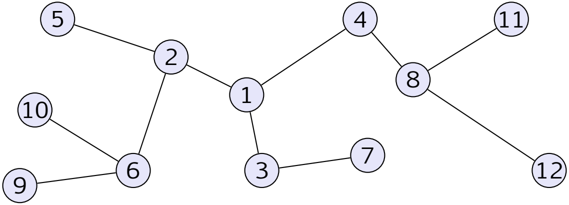
un arbre est un graphe simple non orienté, acyclique et connexe.



- simple : il n′existe pas d′arête reliant un sommet à lui-même et deux sommets distincts sont reliés par au plus une arête ;
- acyclique : il n′existe pas de chemin fermé ;

Arbres et théorie des graphes

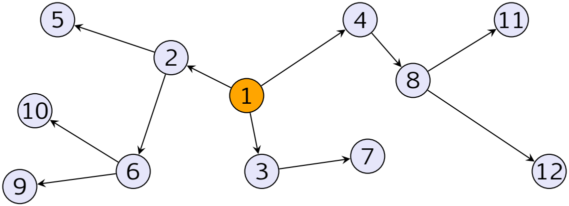
un arbre est un graphe simple non orienté, acyclique et connexe.



- simple : il n'existe pas d'arête reliant un sommet à lui-même et deux sommets distincts sont reliés par au plus une arête ;
- acyclique : il n'existe pas de chemin fermé ;
- connexe : il existe toujours un chemin reliant deux sommets distincts.

Arbres et théorie des graphes

un arbre est un graphe simple non orienté, acyclique et connexe.

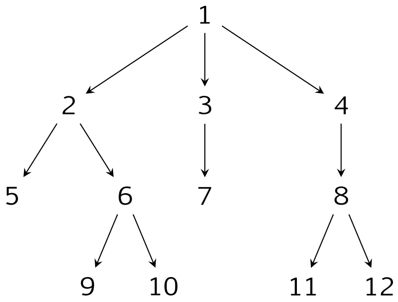


- simple : il n'existe pas d'arête reliant un sommet à lui-même et deux sommets distincts sont reliés par au plus une arête ;
- acyclique : il n'existe pas de chemin fermé ;
- connexe : il existe toujours un chemin reliant deux sommets distincts.

Le choix d'une racine induit une orientation implicite du graphe.

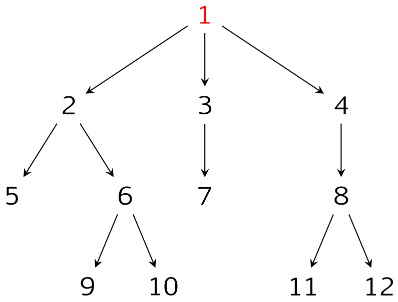
Terminologie

On convient de représenter sur une même ligne les sommets qui sont à la même distance de la racine (ce qui rend implicite l'orientation) :



Terminologie

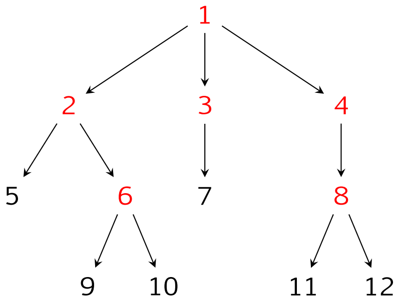
On convient de représenter sur une même ligne les sommets qui sont à la même distance de la racine (ce qui rend implicite l'orientation) :



- 1 est la *racine* ;

Terminologie

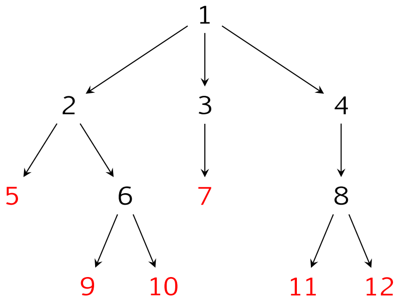
On convient de représenter sur une même ligne les sommets qui sont à la même distance de la racine (ce qui rend implicite l'orientation) :



- 1 est la *racine* ;
- 1, 2, 3, 4, 6, 8 sont des *nœuds internes* ;

Terminologie

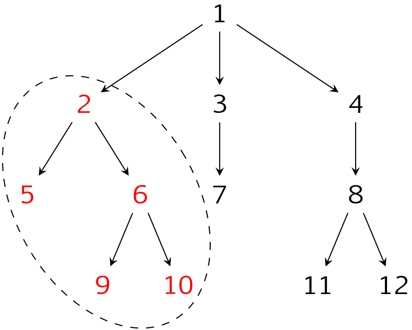
On convient de représenter sur une même ligne les sommets qui sont à la même distance de la racine (ce qui rend implicite l'orientation) :



- 1 est la *racine* ;
- 1, 2, 3, 4, 6, 8 sont des *nœuds internes* ;
- 5, 7, 9, 10, 11, 12 sont des *nœuds externes* (ou *feuilles*).

Terminologie

On convient de représenter sur une même ligne les sommets qui sont à la même distance de la racine (ce qui rend implicite l'orientation) :



Chaque nœud est identifié avec le sous-arbre dont il est la racine.

Définition d'une structure de données

Arbre binaire strict

Dans un arbre *binaire strict* tout nœud interne a une arité égale à 2.

$$\text{Arbre} = \text{feuille} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Définition d'une structure de données

Arbre binaire strict

Dans un arbre *binaire strict* tout nœud interne a une arité égale à 2.

$$\text{Arbre} = \text{feuille} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Dans le cas où nœuds et feuilles sont étiquetés différemment :

```
type ('a, 'b) arbre =
  | Feuille of 'a
  | Noeud of 'b * ('a, 'b) arbre * ('a, 'b) arbre ;;
```

Définition d'une structure de données

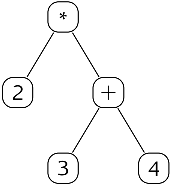
Arbre binaire strict

Dans un arbre *binaire strict* tout nœud interne a une arité égale à 2.

$$\text{Arbre} = \text{feuille} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Dans le cas où nœuds et feuilles sont étiquetés différemment :

```
type ('a, 'b) arbre =
  | Feuille of 'a
  | Noeud of 'b * ('a, 'b) arbre * ('a, 'b) arbre ;;
```




```
# let arb = Noeud ("*", Feuille 2, Noeud ("+", Feuille 3, Feuille 4)) ;;
arb : (int, string) arbre = ...
```

Définition d'une structure de données

Arbre binaire

Dans un arbre *binaire*, tout nœud interne a une arité égale à 1 ou 2.

$$\text{Arbre} = \text{nil} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Les feuilles sont les nœuds dont les deux fils sont vides.

Définition d'une structure de données

Arbre binaire

Dans un arbre *binaire*, tout nœud interne a une arité égale à 1 ou 2.

$$\text{Arbre} = \text{nil} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Les feuilles sont les nœuds dont les deux fils sont vides.

```
type 'a arbre =
  | Nil
  | Noeud of 'a * 'a arbre * 'a arbre ;;
```

Définition d'une structure de données

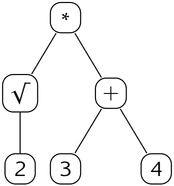
Arbre binaire

Dans un arbre *binaire*, tout nœud interne a une arité égale à 1 ou 2.

$$\text{Arbre} = \text{nil} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Les feuilles sont les nœuds dont les deux fils sont vides.

```
type 'a arbre =
  | Nil
  | Noeud of 'a * 'a arbre * 'a arbre ;;
```




```
# let arb = Noeud ("*", Noeud ("sqrt", Noeud ("2", Nil, Nil), Nil),
                        Noeud ("+", Noeud ("3", Nil, Nil),
                                    Noeud ("4", Nil, Nil))) ;;
arb : string arbre = ...
```

Définition d'une structure de données

Arbre *n*-binaire

Dans le cas général, l'ensemble des descendants d'un nœud est donné sous la forme d'une liste d'arbres.
On peut utiliser le type :

```
type 'a arbre = Noeud of 'a * ('a arbre list) ;;
```

ou

```
type 'a arbre = Nil | Noeud of 'a * ('a arbre list) ;;
```

Définition d'une structure de données

Arbre *n*-binaire

Dans le cas général, l'ensemble des descendants d'un nœud est donné sous la forme d'une liste d'arbres.
On peut utiliser le type :

```
type 'a arbre = Noeud of 'a * ('a arbre list) ;;
```

ou

```
type 'a arbre = Nil | Noeud of 'a * ('a arbre list) ;;
```

Dans la suite du cours on ne donnera des exemples en Caml que pour des arbres binaires représentés par le type :

```
type 'a btree = Nil | Node of 'a * 'a btree * 'a btree
```

Nombre de feuilles et de nœuds

```
let rec nb_feuilles = function
  | Nil                        -> 0
  | Node (_, Nil, Nil)         -> 1
  | Node (_, fils_g, fils_d) -> (nb_feuilles fils_g) +
                                   (nb_feuilles fils_d) ;;
```

```
let rec nb_noeuds = function
  | Nil                        -> 0
  | Node (_, fils_g, fils_d) -> 1 + (nb_noeuds fils_g)
                                   + (nb_noeuds fils_d) ;;
```

Nombre de feuilles et de nœuds

```
let rec nb_feuilles = function
  | Nil                      -> 0
  | Node (_, Nil, Nil)       -> 1
  | Node (_, fils_g, fils_d) -> (nb_feuilles fils_g) +
                                  (nb_feuilles fils_d) ;;
```

```
let rec nb_noeuds = function
  | Nil                      -> 0
  | Node (_, fils_g, fils_d) -> 1 + (nb_noeuds fils_g)
                                  + (nb_noeuds fils_d) ;;
```

Si un arbre binaire strict possède n nœuds internes et f feuilles alors $f = n + 1$.

Une preuve par induction est possible.

Nombre de feuilles et de nœuds

```
let rec nb_feuilles = function
  | Nil                    -> 0
  | Node (_, Nil, Nil)     -> 1
  | Node (_, fils_g, fils_d) -> (nb_feuilles fils_g) +
                                (nb_feuilles fils_d) ;;
```

```
let rec nb_noeuds = function
  | Nil                    -> 0
  | Node (_, fils_g, fils_d) -> 1 + (nb_noeuds fils_g)
                                + (nb_noeuds fils_d) ;;
```

Si un arbre binaire strict possède n nœuds internes et f feuilles alors $f = n + 1$.

Une preuve par induction est possible.

Sinon, on peut remarquer que le nombre de sommets ayant un père est égal à $n + f - 1$ (la racine n'a pas de père), et chaque père a deux fils donc :

$$2n = n + f - 1 \iff f = n + 1.$$

Hauteur d'un arbre

La profondeur d'un nœud ou d'une feuille est la distance qui sépare ce nœud ou cette feuille de la racine.
La hauteur de l'arbre est la profondeur maximale d'une feuille.

Hauteur d'un arbre

La profondeur d'un nœud ou d'une feuille est la distance qui sépare ce nœud ou cette feuille de la racine.
La hauteur de l'arbre est la profondeur maximale d'une feuille.

```
let rec hauteur = function
  | Nil                        -> -1
  | Node (_, fils_g, fils_d) -> 1 + max (hauteur fils_g)
                                          (hauteur fils_d) ;;
```

Hauteur d'un arbre

La profondeur d'un nœud ou d'une feuille est la distance qui sépare ce nœud ou cette feuille de la racine.
La hauteur de l'arbre est la profondeur maximale d'une feuille.

Si h désigne la hauteur d'un arbre binaire, le nombre de feuilles est majoré par 2^h.

Preuve par récurrence sur la hauteur de l'arbre.

Hauteur d'un arbre

La profondeur d'un nœud ou d'une feuille est la distance qui sépare ce nœud ou cette feuille de la racine.
La hauteur de l'arbre est la profondeur maximale d'une feuille.

Si h désigne la hauteur d'un arbre binaire, le nombre de feuilles est majoré par 2^h.

Preuve par récurrence sur la hauteur de l'arbre.

Un arbre binaire possédant n feuilles a pour hauteur minimale $\lceil \log_2 n \rceil$.

Hauteur d'un arbre

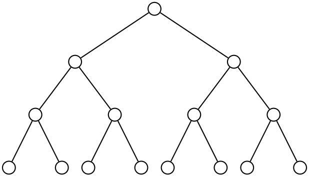
La profondeur d'un nœud ou d'une feuille est la distance qui sépare ce nœud ou cette feuille de la racine.
La hauteur de l'arbre est la profondeur maximale d'une feuille.

Si h désigne la hauteur d'un arbre binaire, le nombre de feuilles est majoré par 2^h.

Preuve par récurrence sur la hauteur de l'arbre.

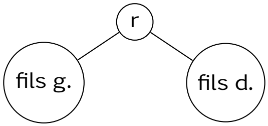
Un arbre binaire possédant n feuilles a pour hauteur minimale $\lceil \log_2 n \rceil$.

Un arbre binaire de hauteur h à 2^h feuilles est dit complet :



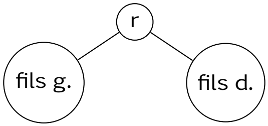
Parcours en profondeur

Dans un parcours en profondeur, chaque sous-arbre est exploré dans son entier avant d'explorer le sous-arbre suivant.



Parcours en profondeur

Dans un parcours en profondeur, chaque sous-arbre est exploré dans son entier avant d'explorer le sous-arbre suivant.

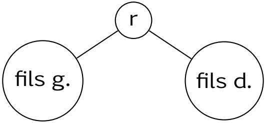


Parcours préfixe : r ⟶ fils g. ⟶ fils d.

```
let rec parcours_prefixe = function
  | Nil                       -> ()
  | Noeud (r, fils_g, fils_d) -> print_int r ;
                                 parcours_prefixe fils_g ;
                                 parcours_prefixe fils_d ;;
```

Parcours en profondeur

Dans un parcours en profondeur, chaque sous-arbre est exploré dans son entier avant d'explorer le sous-arbre suivant.

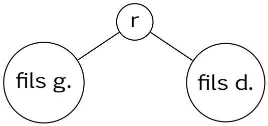


Parcours suffixe : fils g. ⟶ fils d. ⟶ r

```
let rec parcours_suffixe = function
  | Nil                      -> ()
  | Noeud (r, fils_g, fils_d) -> parcours_suffixe fils_g ;
                                 parcours_suffixe fils_d ;
                                 print_int r ;;
```

Parcours en profondeur

Dans un parcours en profondeur, chaque sous-arbre est exploré dans son entier avant d'explorer le sous-arbre suivant.

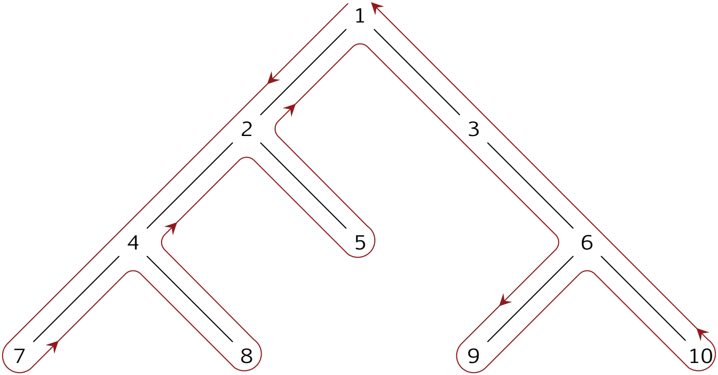


Parcours infixe : fils g. ⟶ r ⟶ fils d.

```
let rec parcours_infixe = function
  | Nil                    -> ()
  | Noeud (r, fils_g, fils_d) -> parcours_infixe fils_g ;
                                 print_int r ;
                                 parcours_infixe fils_d ;;
```

Parcours en profondeur

Illustration graphique

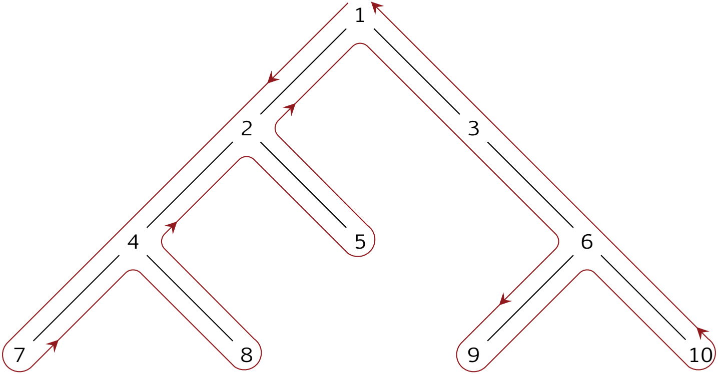


Parcours préfixe : dans l'ordre du premier passage à gauche d'un nœud.

$$1 \to 2 \to 4 \to 7 \to 8 \to 5 \to 3 \to 6 \to 9 \to 10$$

Parcours en profondeur

Illustration graphique

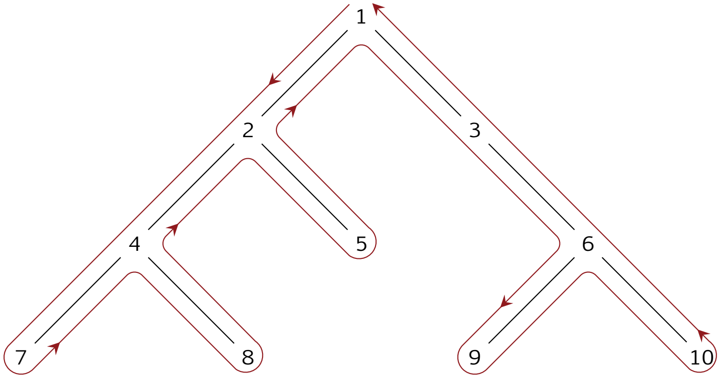


Parcours suffixe : dans l'ordre du premier passage à droite d'un nœud.

$$7 \to 8 \to 4 \to 5 \to 2 \to 9 \to 10 \to 6 \to 3 \to 1$$

Parcours en profondeur

Illustration graphique

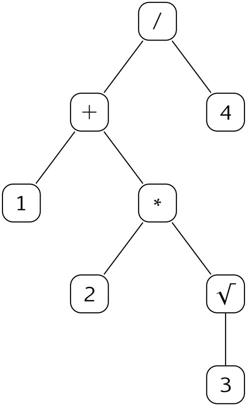


Parcours infixe : dans l'ordre du premier passage sous un nœud.

$$7 \to 4 \to 8 \to 2 \to 5 \to 1 \to 3 \to 9 \to 6 \to 10$$

Représentations d'une expression arithmétique

La représentation préfixe (resp. postfixe) d'une expression arithmétique correspond au parcours préfixe (resp. postfixe) de l'arbre qui la représente.

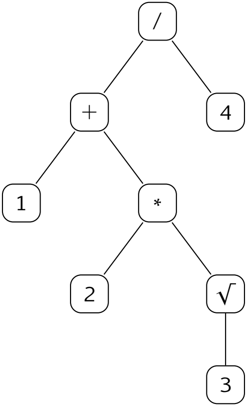


préfixe : `/ + 1 * 2 sqrt 3 4`

postfixe : `1 2 3 sqrt * + 4 /`

Représentations d'une expression arithmétique

La représentation préfixe (resp. postfixe) d'une expression arithmétique correspond au parcours préfixe (resp. postfixe) de l'arbre qui la représente.



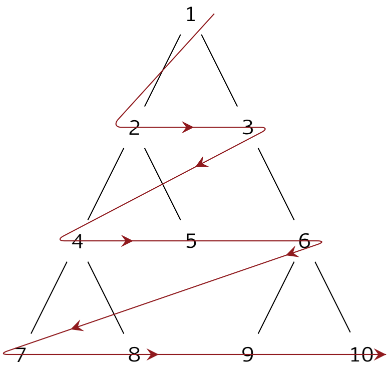
préfixe : `/ + 1 * 2 sqrt 3 4`

postfixe : `1 2 3 sqrt * + 4 /`

Nous démontrerons que ces expressions sont non ambiguës : l'usage des parenthèses est superflu.

Parcours hiérarchique

Ou parcours en largeur, il consiste à parcourir les nœuds et feuilles en les classant par profondeur croissante (et en général de la gauche vers la droite pour une profondeur donnée) :



Nous traiterons du parcours hiérarchique au moment de l'étude des files d'attente.