# Arbres binaires

Programmation Fonctionnelle
Master 2 I2L apprentissage

Sébastien Verel
verel@lisic.univ-littoral.fr
http://www-lisic.univ-littoral.fr/~verel

Université du Littoral Côte d'Opale
Laboratoire LISIC
Equipe OSMOSE

18 décembre 2014

# Avantages / inconvénients des structure de données

- Tableau :
  + Accès rapide à une donnée (temps constant $O(1)$)
  − Taille bornée
- Liste :
  + Taille non bornée
  − Accès 'lent' à une donnée (temps linéaire $O(n)$)

# Avantages / inconvénients des structure de données

- Tableau :
  + Accès rapide à une donnée (temps constant $O(1)$)
  − Taille bornée
- Liste :
  + Taille non bornée
  − Accès 'lent' à une donnée (temps linéaire $O(n)$)

Certaines solutions :

- Fausse liste : succession de tableaux de taille bornée
- Faux tableau : liste avec 'get(i)'

# Avantages / inconvénients des structure de données

- Tableau :
  + Accès rapide à une donnée (temps constant $O(1)$)
  − Taille bornée
- Liste :
  + Taille non bornée
  − Accès 'lent' à une donnée (temps linéaire $O(n)$)

Certaines solutions :

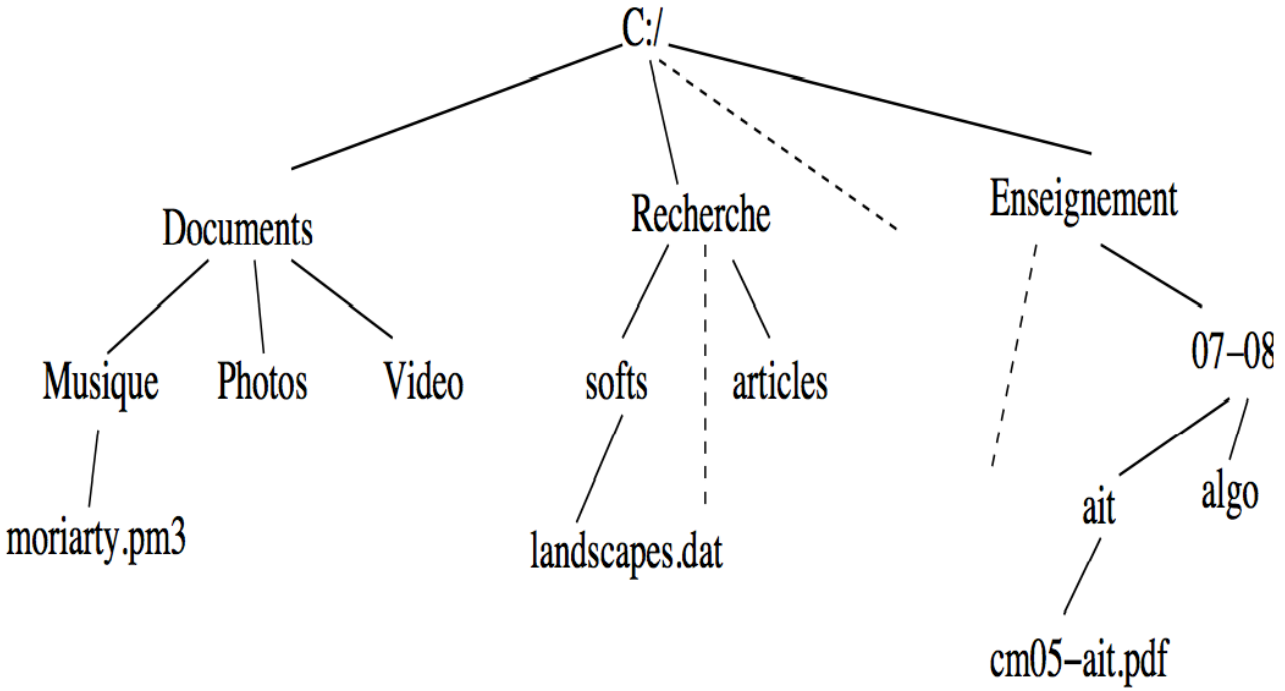
- Fausse liste : succession de tableaux de taille bornée
- Faux tableau : liste avec 'get(i)'

**Une autre solution**

Structure d'arbre

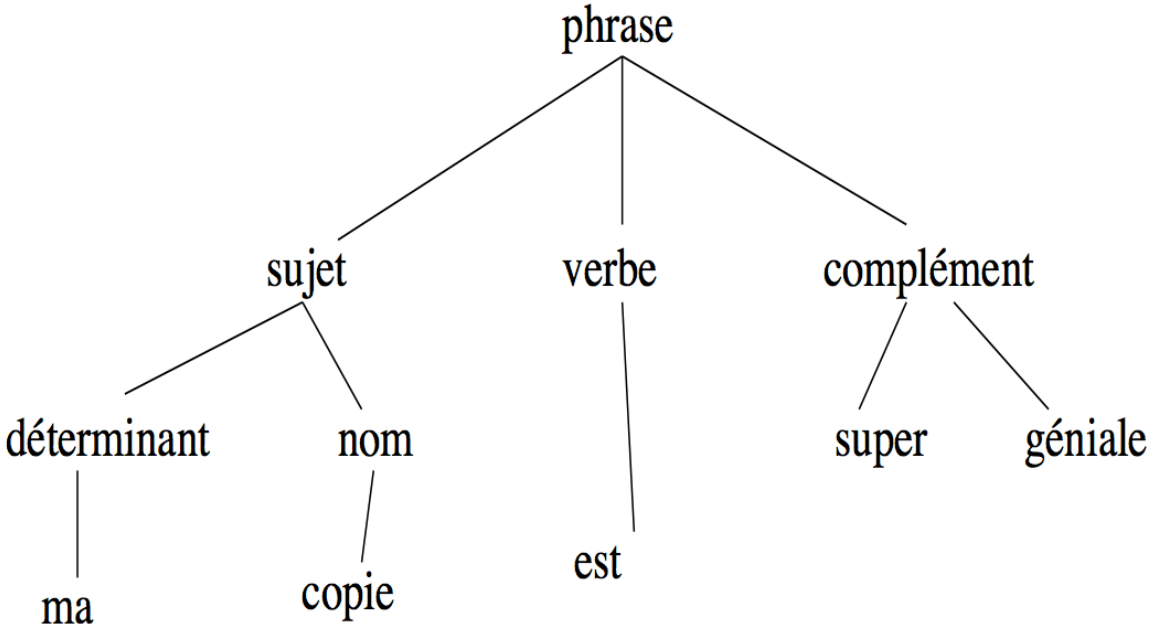
# Exemple de données en arbre

## Dossiers informatiques



# Exemple de données en arbre

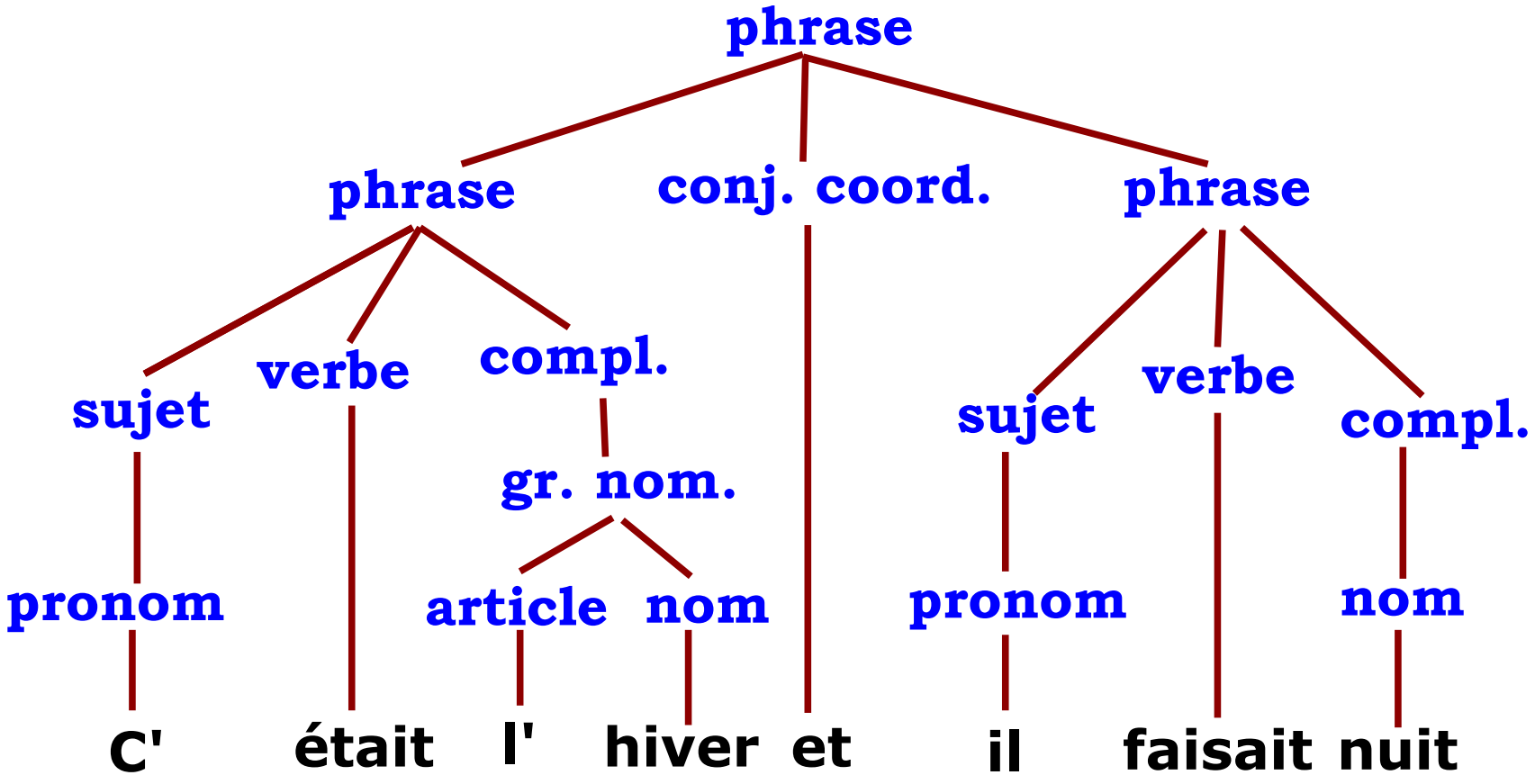
## Analyse grammaticale



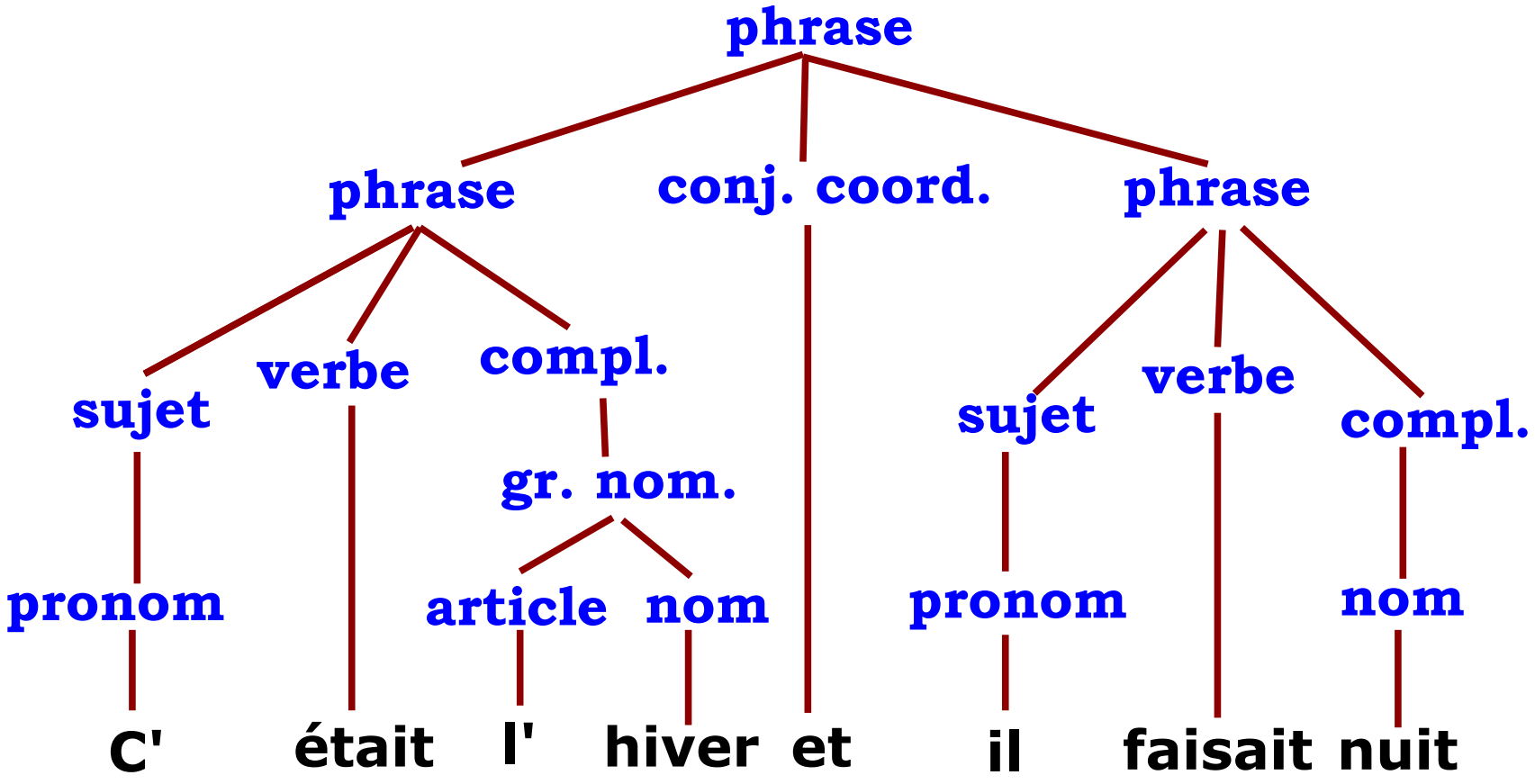

2 types de données :

- Eléments grammaticaux
- Eléments terminaux : les mots

# Exemple littéraire



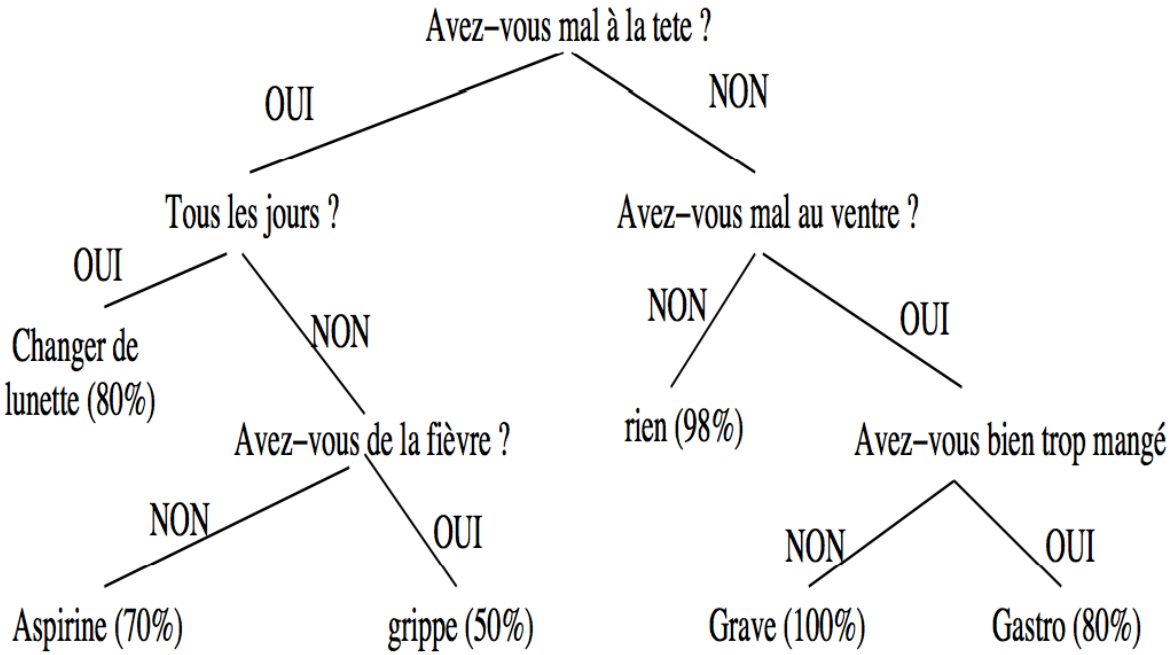
# Exemple littéraire




*Première phrase de "La position du tireur couché", J.-P. Manchette.*

# Exemple de données en arbre

## Arbre de décision

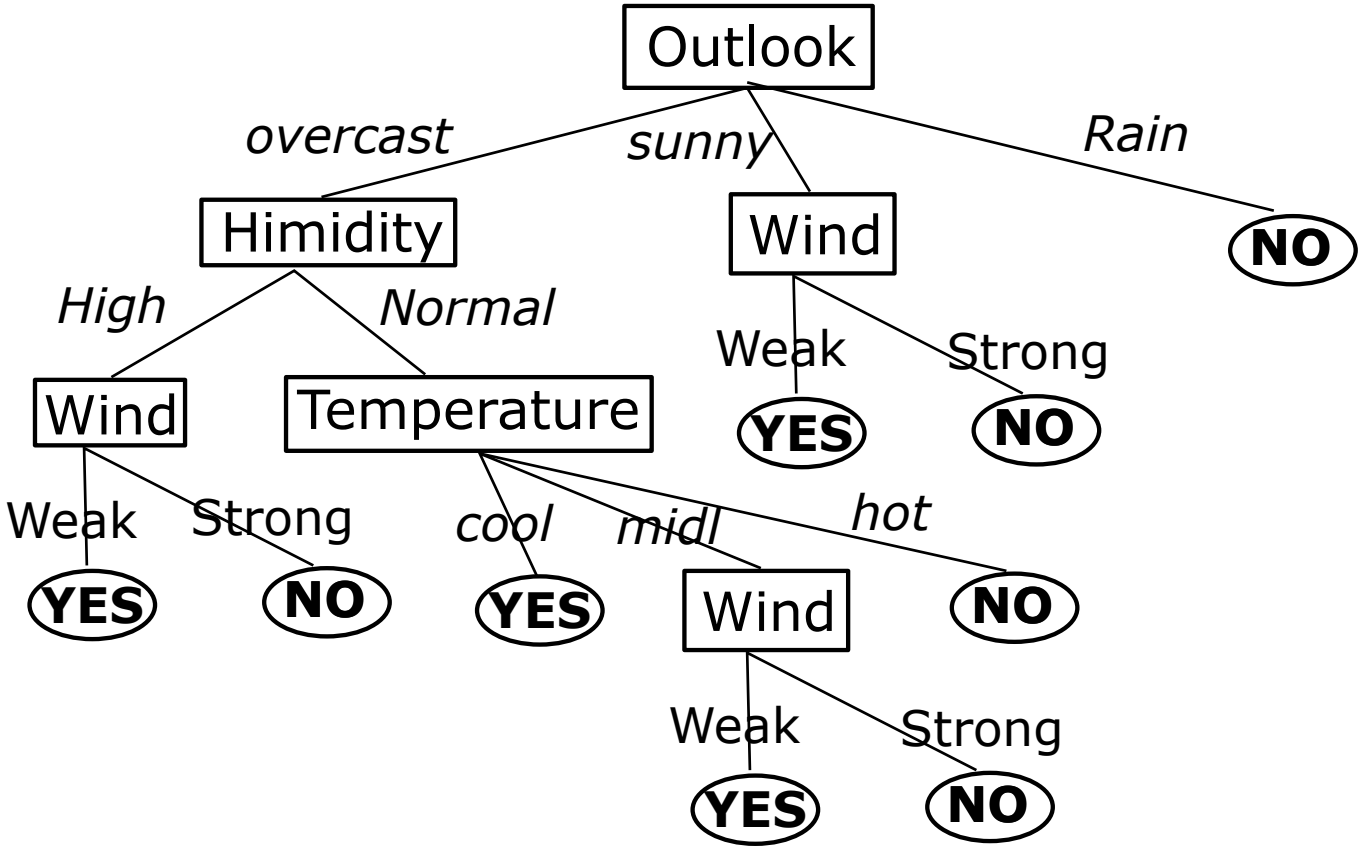



2 types de données :

- les questions
- Eléments terminaux : les décisions

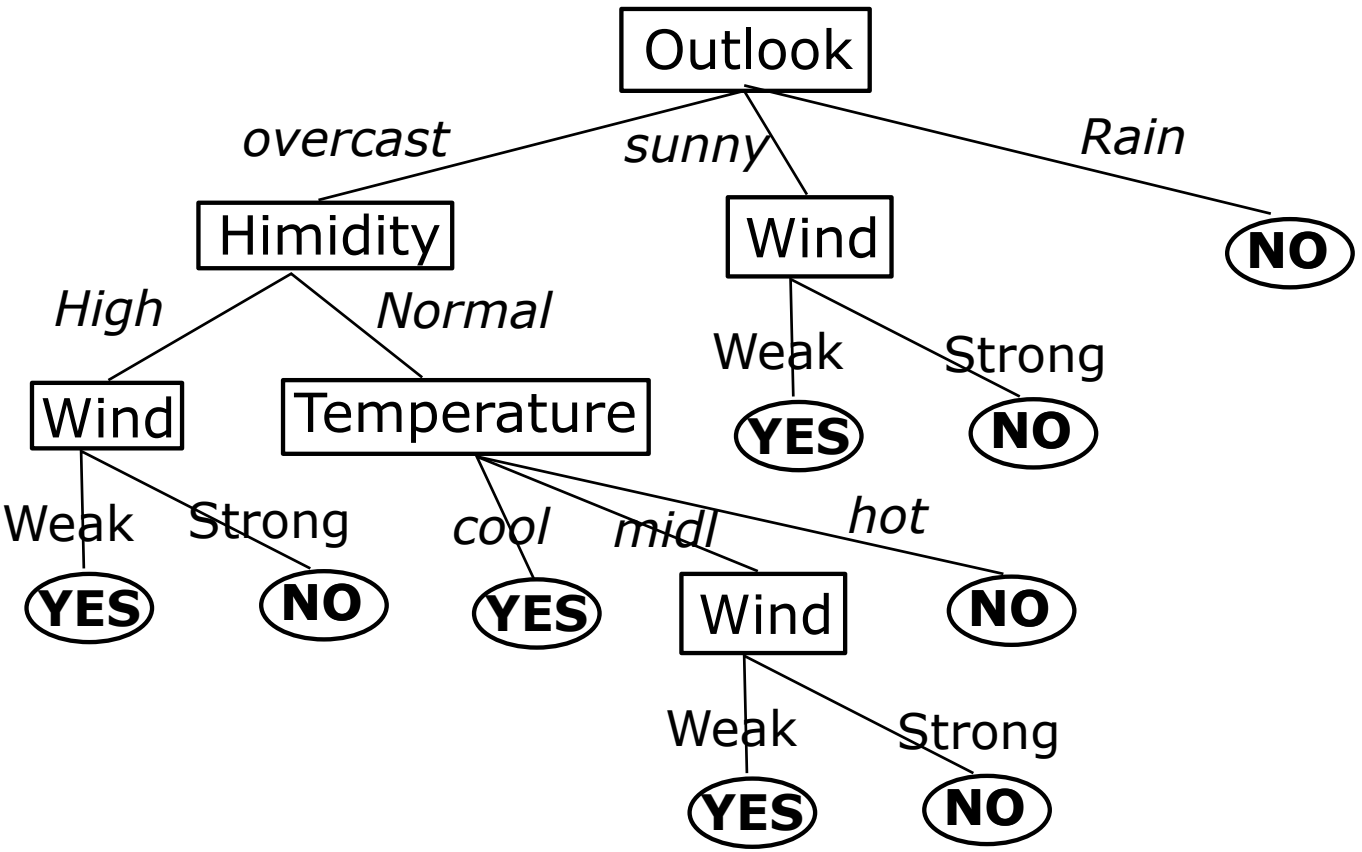
# Un arbre pour prendre une décision

## Classification à l'aide d'un arbre



# Un arbre pour prendre une décision

## Classification à l'aide d'un arbre

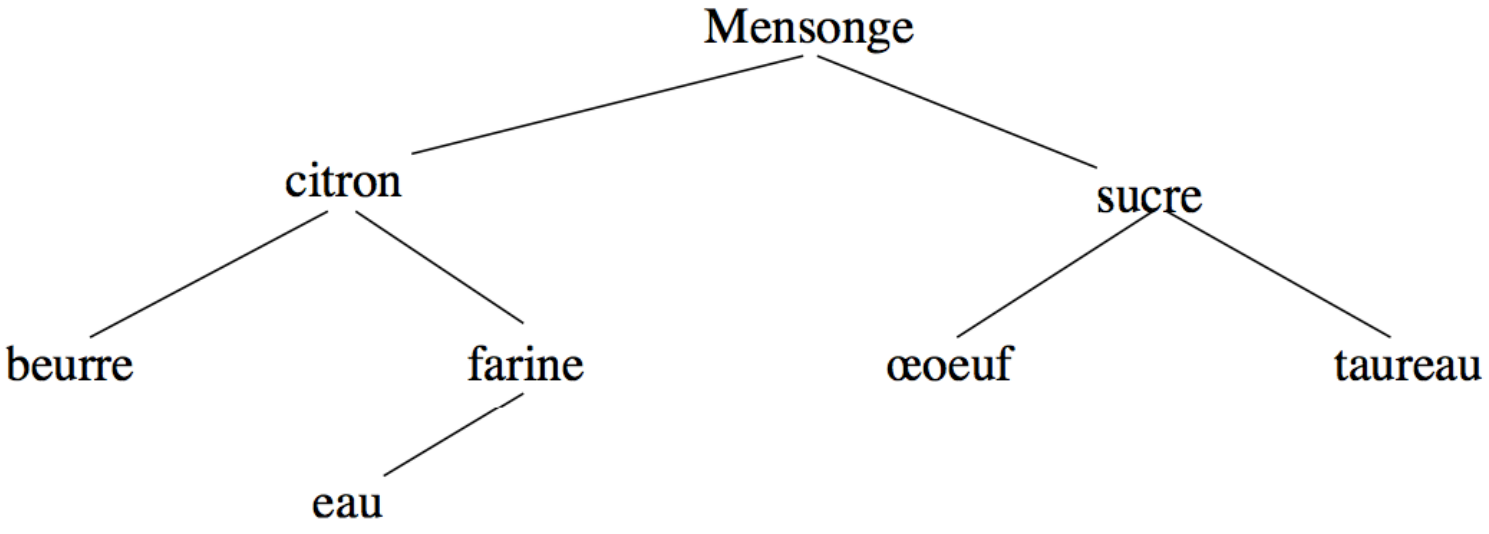



*Remarque* : un arbre code en fait un ensemble de règles (conjonctions, disjonctions)

Si Outlook = "overcast" et Humidity =... alors playball = Yes

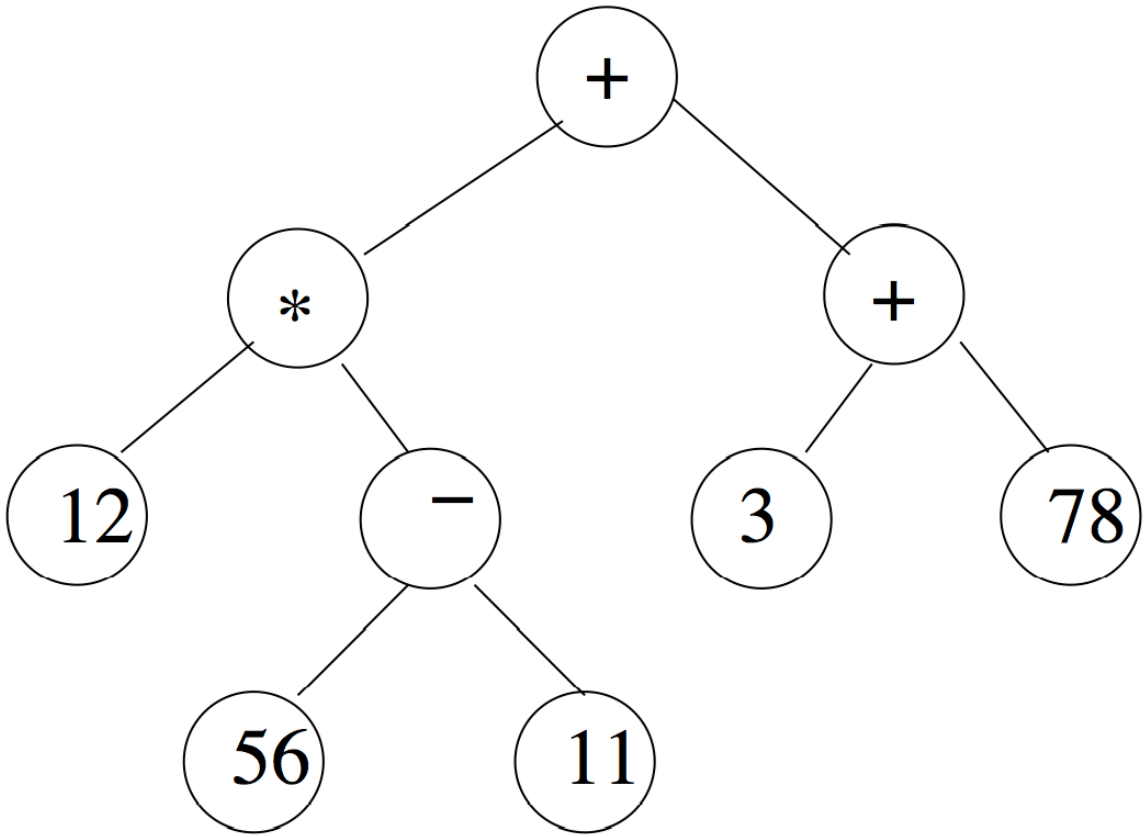
# Exemple de données en arbre

## Information organisée



# Exemple de données en arbre

## Expressions arithmétiques




2 types de données :

- les opérateurs (binaires)
- Eléments terminaux : les nombres

# Définitions

## Définition informelle

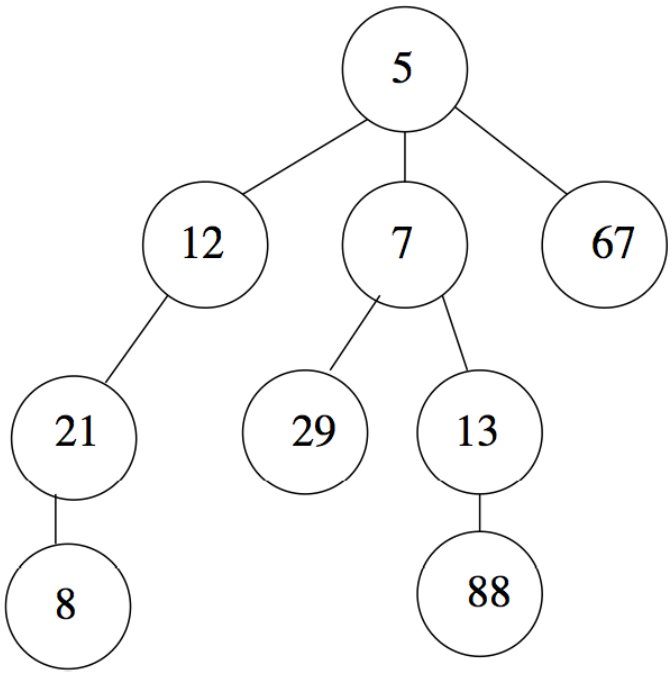
Ensemble de données organisées de manière hiérarchique

## Une définition (presque) formelle

Graphe orienté, connexe et acyclique.

# Représentation

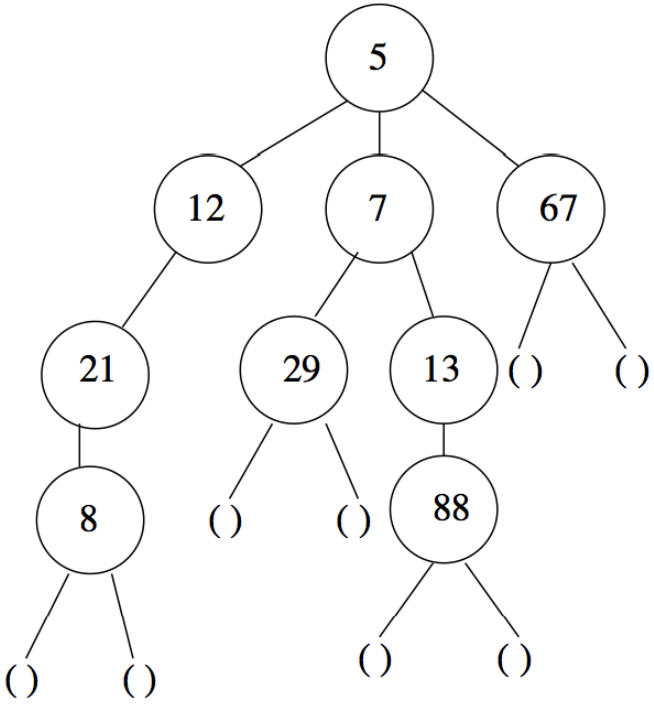
Les arbres (en informatique) se représentent la racine vers le "ciel" !




- 5 : racine de l'arbre
- 12, 7, 21, 13, 5 : nœuds internes
- 67, 8, 29, 88 : feuilles
- 12, 7, 67 : fils du nœud 5
- 7 : père de 29 et 13
- le nœud 5 a pour arité 3 et le nœud 21 a pour arité 1
- La hauteur de l'arbre est 4

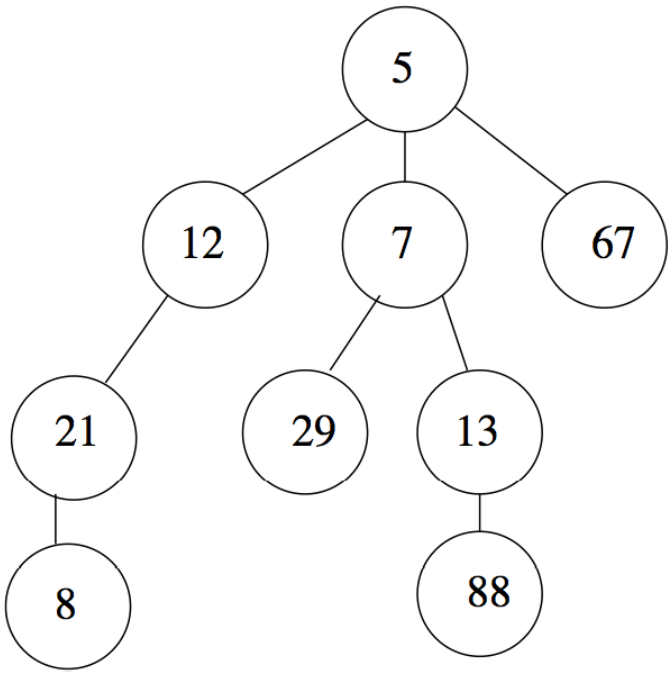
# Représentation

Comme pour les listes (et beaucoup de structures), il est utile de définir un arbre vide



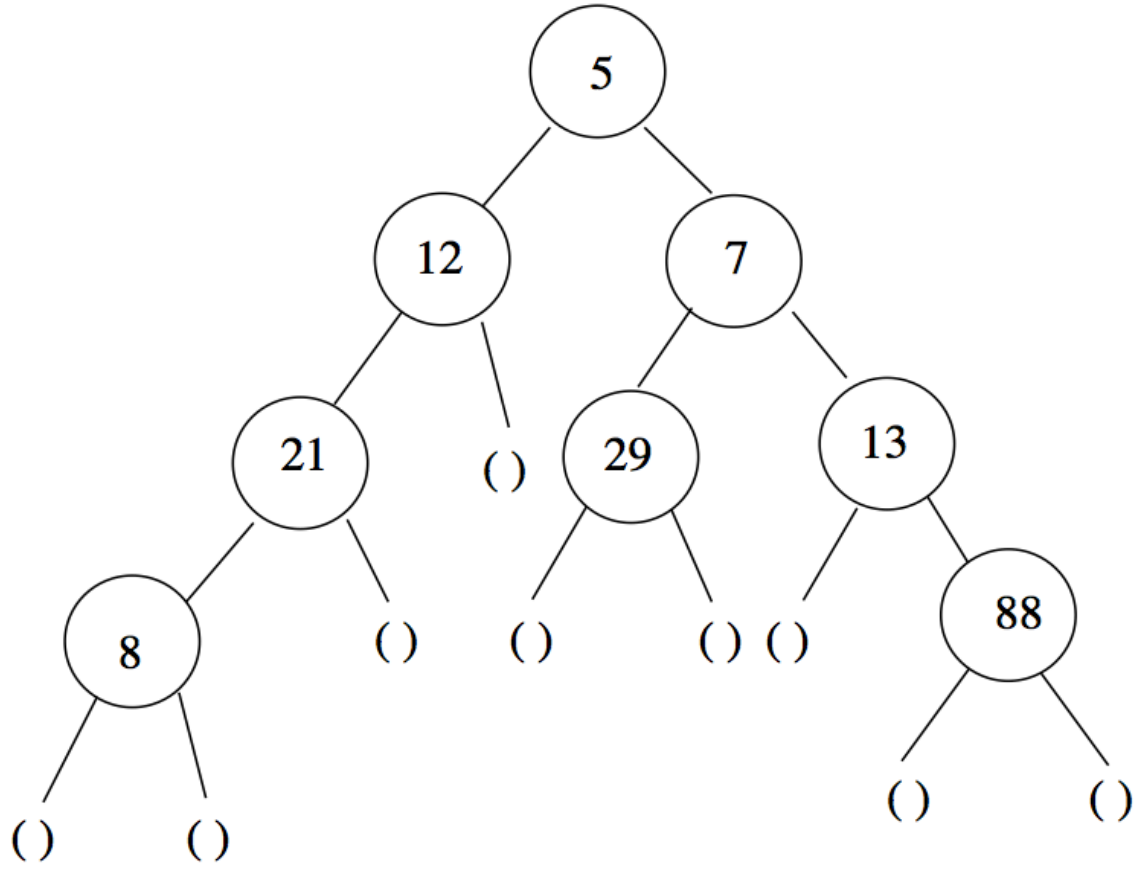

- 5 : racine de l'arbre
- 12, 7, 21, 13, 5 : nœuds internes
- 67, 8, 29, 88 : feuilles
- 12, 7, 67 : fils du nœud 5
- 7 : père de 29 et 13
- le nœud 5 a pour arité 3 et le nœud 21 a pour arité 1
- La hauteur de l'arbre est 4

# Définitions




- Nœud : élément d'un arbre
- Père d'un nœud : nœud directement antécédent (unique)
- Fils d'un nœud : nœuds directement précédents (multiple)
- Racine : nœud qui ne possède pas de père
- Feuille : nœud qui ne possède pas de fils non-vide
- Arité d'un nœud : nombre de fils non-vide

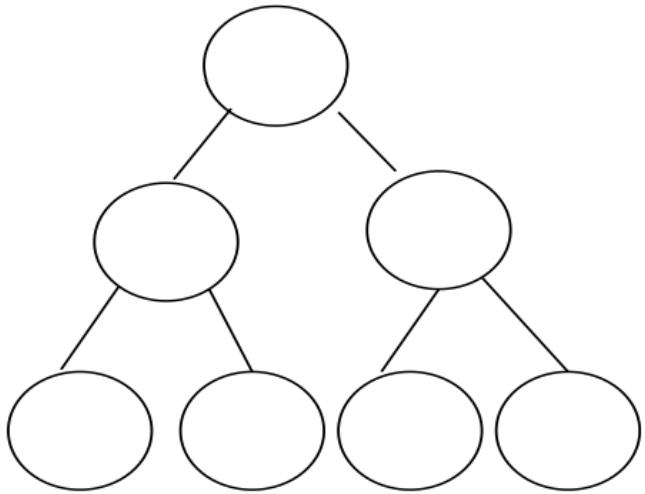
# Arbre binaire




**Définition**

Arbre dont tous les nœud possèdent au plus 2 fils

# Arbre binaire



**Arbre équilibré**

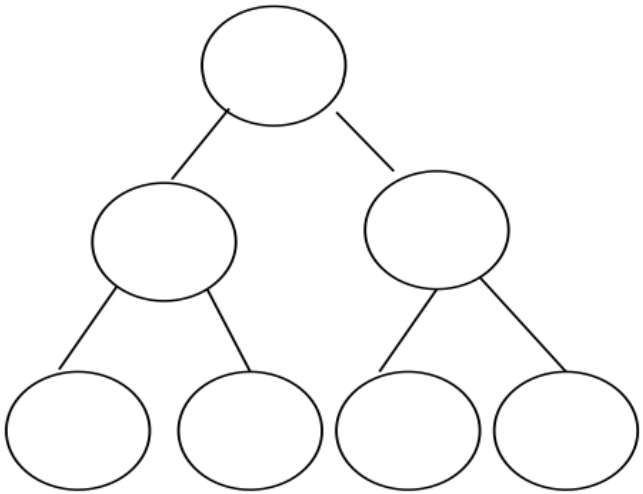
Arbre où chaque nœud a des arbres fils de même hauteur

**Arbre binaire complet**

Arbre binaire où chaque nœud est d'arité 0 ou 2.

Nombre de nœuds d'un arbre binaire équilibré et complet de hauteur $h$ :

# Arbre binaire



**Arbre équilibré**

Arbre où chaque nœud a des arbres fils de même hauteur

**Arbre binaire complet**

Arbre binaire où chaque nœud est d'arité 0 ou 2.

Nombre de nœuds d'un arbre binaire équilibré et complet de hauteur $h$ : $2^h - 1$

# Type de données abstrait

6 fonctions primitives :

- `arbreVide` : `rien` $\rightarrow$ `arbre`
  construit un arbre vide
- `arbreCons` : `element` $\times$ `arbre` $\times$ `arbre` $\rightarrow$ `arbre`
  construit un arbre binaire dont la racine est l'élément, l'arbre gauche le deuxième argument, l'arbre le troisième
- `racine` : `arbre` $\rightarrow$ `element`
  donne l'élément de la racine (si possible)
- `arbreGauche` : `arbre` $\rightarrow$ `arbre`
  donne l'arbre à gauche (si possible)
- `arbreDroit` : `arbre` $\rightarrow$ `arbre`
  donne l'arbre droit (si possible)
- `arbreEstVide?` : `arbre` $\rightarrow$ `boolean`
  teste si l'arbre est un arbre vide

# Type de données abstrait

## Remarque

- `racine` : `arbre` $\rightarrow$ `element`
  donne l'élément de la racine (si possible)
- `arbreGauche` : `arbre` $\rightarrow$ `arbre`
  donne l'arbre à gauche (si possible)
- `arbreDroit` : `arbre` $\rightarrow$ `arbre`
  donne l'arbre droit (si possible)

Ces fonctions ne s'appliquent qu'à un arbre non vide !

# Exemples

- `arbreVide()`

# Exemples

- `arbreVide()`

  ()

- `arbreEstVide?(arbreVide())`

# Exemples

- `arbreVide()`

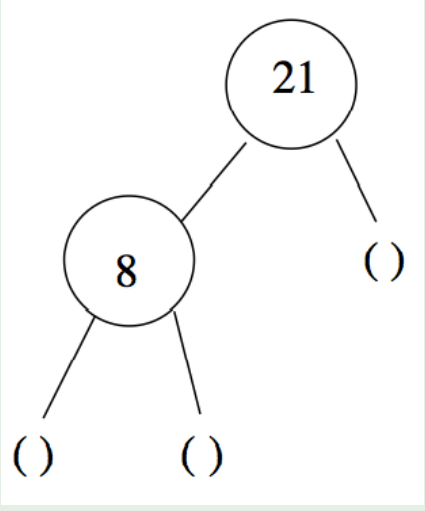
  `()`

- `arbreEstVide?(arbreVide())`

  `true`

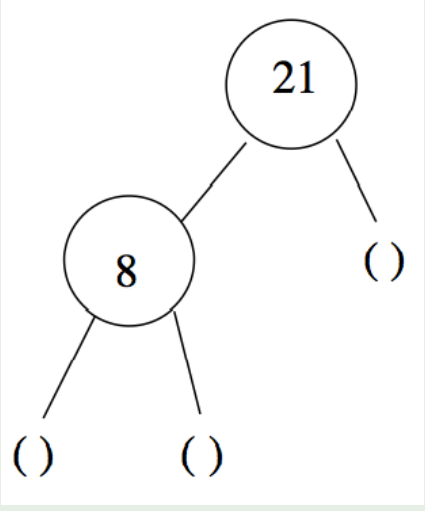
# Exemples

Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?



# Exemples

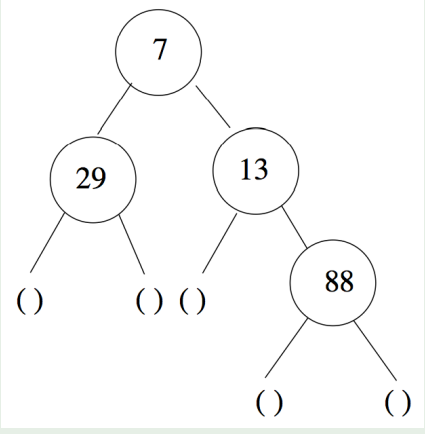
Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?




```
a ← arbreCons(21, arbreCons(8, arbreVide(),
arbreVide()), arbreVide())
```

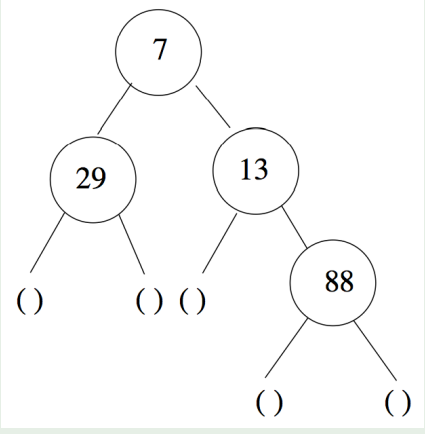
# Exemples

Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?



# Exemples

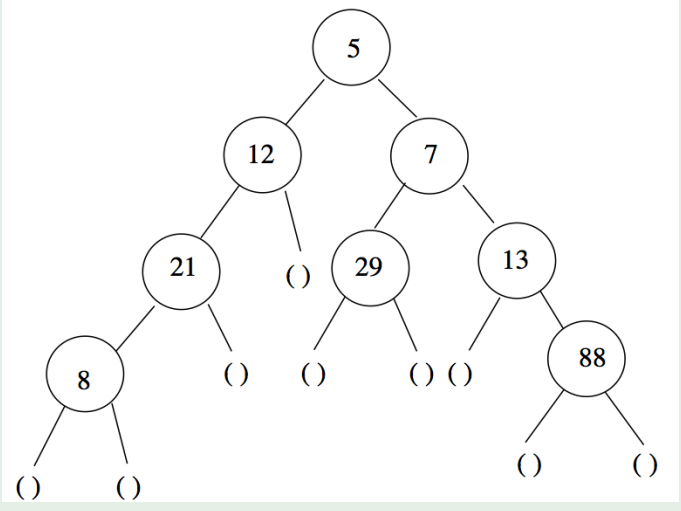
Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?




```
b ← arbreCons(7, arbreCons(29, arbreVide(),
arbreVide()), arbreCons(13, arbreVide(),
arbreCons(88, arbreVide(), arbreVide())))
```

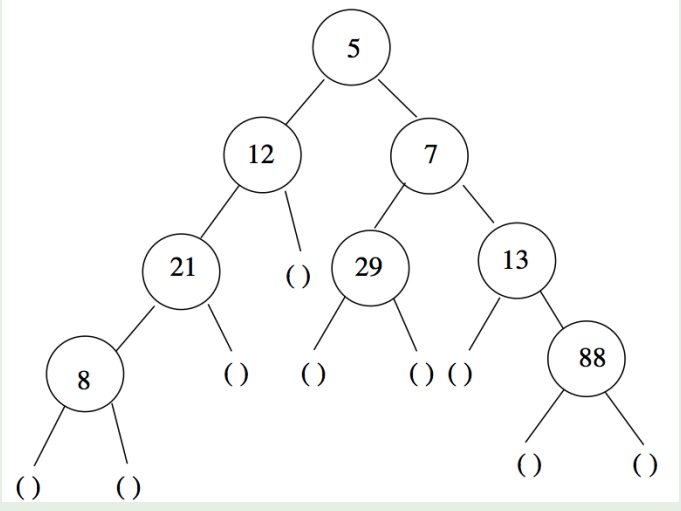
# Exemples

Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?



# Exemples

Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?




`c` $\leftarrow$ `arbreCons(5, arbreCons(12, a, arbreVide()), b)`

# Exemples

- `racine(c)`

# Exemples

- `racine(c)`

  5

- `arbreEstVide?(a)`

# Exemples

- `racine(c)`

  `5`

- `arbreEstVide?(a)`

  `false`

- `racine(b)`

  `7`

- `arbreGauche(c)`

# Exemples

- `racine(c)`

  `5`

- `arbreEstVide?(a)`

  `false`

- `racine(b)`

  `7`

- `arbreGauche(c)`

  `arbreCons(12, b, arbreVide())`

- `arbreDroit(a)`

# Exemples

- `racine(c)`

  `5`

- `arbreEstVide?(a)`

  `false`

- `racine(b)`

  `7`

- `arbreGauche(c)`

  `arbreCons(12, b, arbreVide())`

- `arbreDroit(a)`

  `arbreVide()`

# Les arbres en Erlang

## Synthaxe concrète

Un arbre vide est codé dans le langage Erlang par tuple pointé "vide" :

```
{ arbre }
```

Un arbre non-vide est codé dans le langage Erlang par un "triplet" pointé :

```
{ arbre, Elem, Ag, Ad }
```

où `Elem` est l'étiquette du nœud, `Ag` est l'arbre à gauche et `Ad` est l'arbre à droite

# Les arbres en Erlang

## Synthaxe concrète

Un arbre vide est codé dans le langage Erlang par tuple pointé "vide" :

```
{ arbre }
```

Un arbre non-vide est codé dans le langage Erlang par un "triplet" pointé :

```
{ arbre, Elem, Ag, Ad }
```

où `Elem` est l'étiquette du nœud, `Ag` est l'arbre à gauche et `Ad` est l'arbre à droite

## Exercice

Définir les 6 fonctions primitives du TDA arbre.

# TDA arbre en Erlang

# TDA arbre en Erlang

## Les constructeurs

```
arbreVide() ->
    {arbre}.

arbreCons(Elem, Ag, Ad) ->
    {arbre, Elem, Ag, Ad}.
```

# TDA arbre en Erlang

## Les accesseurs

```
racine({arbre, Elem, _, _}) ->
    Elem ;
racine({arbre}) ->
    errorVide.

arbreDroit({arbre, _, _, Ad}) ->
    Ad ;
arbreDroit({arbre}) ->
    errorVide.

arbreGauche({arbre, _, Ag, _}) ->
    Ag ;
arbreGauche({arbre}) ->
    errorVide.
```

# TDA arbre en Erlang

## Le test

```
arbreEstVide({arbre}) ->
    true ;
arbreEstVide({arbre, _, _, _}) ->
    false.
```

# Les arbres en Erlang

**Exercice**

Ajouter une primitive qui teste si le nœud est une feuille.

# Les arbres en Erlang

**Exercice**

Ajouter une primitive qui teste si le nœud est une feuille.

```
arbreEstFeuille(A) ->
   not(arbreEstVide(A)) andalso
   arbreEstVide(arbreGauche(A)) andalso
   arbreEstVide(arbreDroit(A)).
```

# Recherche arbre non ordonné

**Exercice**

Recherche d'un élément dans un arbre non ordonné

# Recherche arbre non ordonné

**Exercice**

Recherche d'un élément dans un arbre non ordonné

```
recherche( _, {arbre} )->
    false;
recherche( Elem, {arbre, Elem, _, _} ) ->
    true;
recherche( Elem, {arbre, _, Ag, Ad} ) ->
    recherche(Elem, Ag) orelse recherche(Elem, Ad).
```

# Recherche arbre non ordonné

**Exercice**

Recherche d'un élément dans un arbre non ordonné

```
recherche( _, {arbre} )->
    false;
recherche( Elem, {arbre, Elem, _, _} ) ->
    true;
recherche( Elem, {arbre, _, Ag, Ad} ) ->
    recherche(Elem, Ag) orelse recherche(Elem, Ad).
```

Attention ! Nous n'avons pas respecté le TDA ! ! !

# Recherche arbre non ordonné, version TDA

# Recherche arbre non ordonné, version TDA

```
rechercheTDA(Elem, Arbre) ->
    case arbreEstVide(Arbre) of
        true ->
            false;
        false ->
            case racine(Arbre) of
                Elem ->
                    true ;
                _ ->
           rechercheTDA(Elem, arbreGauche(Arbre))
           orelse rechercheTDA(Elem, arbreDroit(Arbre))
            end
    end.
```

# Recherche arbre non ordonné, version TDA, version 2

# Recherche arbre non ordonné, version TDA, version 2

```
rechercheTDA(Elem, Arbre) ->
    (not(arbreEstVide(Arbre)) andalso
    ((racine(Arbre) == Elem) orelse
    rechercheTDA(Elem, arbreGauche(Arbre)) orelse
    rechercheTDA(Elem, arbreDroit(Arbre)))).
```

# Recherche arbre ordonné

**Exercice**

Recherche d'un élément dans un arbre où l'on suppose que tous les éléments plus petit que l'étiquette sont dans l'arbre gauche et tous les éléments plus grand que l'étiquette sont dans l'arbre droit.

# Recherche arbre ordonné

## Exercice

Recherche d'un élément dans un arbre où l'on suppose que tous les éléments plus petit que l'étiquette sont dans l'arbre gauche et tous les éléments plus grand que l'étiquette sont dans l'arbre droit.

```
rechercheOrd( _, {arbre} )->
    false;
rechercheOrd( Elem, {arbre, Elem, _, _} ) ->
    true;
rechercheOrd( Elem, {arbre, E, Ag, Ad} ) ->
    case Elem < E of
        true ->
            rechercheOrd(Elem, Ag) ;
        false ->
            rechercheOrd(Elem, Ad)
    end.
```

# Recherche arbre ordonné

## Exercice

Recherche d'un élément dans un arbre où l'on suppose que tous les éléments plus petit que l'étiquette sont dans l'arbre gauche et tous les éléments plus grand que l'étiquette sont dans l'arbre droit.

```
rechercheOrd( _, {arbre} )->
    false;
rechercheOrd( Elem, {arbre, Elem, _, _} ) ->
    true;
rechercheOrd( Elem, {arbre, E, Ag, Ad} ) ->
    case Elem < E of
        true ->
            rechercheOrd(Elem, Ag) ;
        false ->
            rechercheOrd(Elem, Ad)
    end.
```

Principe du diviser pour règner,
en plus la stucture de données "calcule pour vous"

# Complexité des algorithmes de recherche

**Exercice**

Evaluer la complexité de chaque algorithme pour un arbre binaire équilibré complet dont le nombre de nœuds est $n$

# Complexité des algorithmes de recherche

**Exercice**

Evaluer la complexité de chaque algorithme pour un arbre binaire équilibré complet dont le nombre de nœuds est $n$

Recherche dans un arbre non ordonné :

$$O(n)$$

# Complexité des algorithmes de recherche

**Exercice**

Evaluer la complexité de chaque algorithme pour un arbre binaire équilibré complet dont le nombre de nœuds est $n$

Recherche dans un arbre non ordonné :

$$O(n)$$

Recherche dans un arbre ordonné :

$$O(log(n))$$

# Nombre de nœuds

## Calcul du nombre de nœuds

# Nombre de nœuds

## Calcul du nombre de nœuds

```
nbNoeuds({arbre}) ->
    0 ;
nbNoeuds({arbre, _, Ag, Ad}) ->
    1 + nbNoeuds(Ag) + nbNoeuds(Ad).
```

# Nombre de nœuds

## Calcul du nombre de nœuds

```
nbNoeuds({arbre}) ->
    0 ;
nbNoeuds({arbre, _, Ag, Ad}) ->
    1 + nbNoeuds(Ag) + nbNoeuds(Ad).
```

## Prototype de la fonction récursive sur les arbres

```
myFunction({arbre}) ->
    traitement arbre vide ;
myFunction({arbre, E, Ag, Ad}) ->
    traitement noeud ,
    ... myFunction(Ag) ... myFunction(Ad) ...
    .
```

# Nombre de nœuds, version TDA

```
nbNoeuds(A) ->
    case arbreEstVide(A) of
        true ->
            0 ;
        false ->
            1 + nbNoeuds(arbreGauche(A))
              + nbNoeuds(arbreDroit(A))
    end.
```

# Hauteur d'un arbre

La hauteur est le nombre maximal de nœuds entre la racine et une feuille

# Hauteur d'un arbre

La hauteur est le nombre maximal de nœuds entre la racine et une feuille

```
hauteur({arbre}) ->
    0;
hauteur({arbre, _, Ag, Ad}) ->
    1 + max(hauteur(Ag), hauteur(Ad)).
```

# Version TDA

```
hauteur(A) ->
    case arbreEstVide(A) of
        true ->
            0 ;
        false ->
            1 + max(hauteur(arbreGauche(A)),
                    hauteur(arbreDroit(A))
    end.
```

# Algorithme avec accumulation

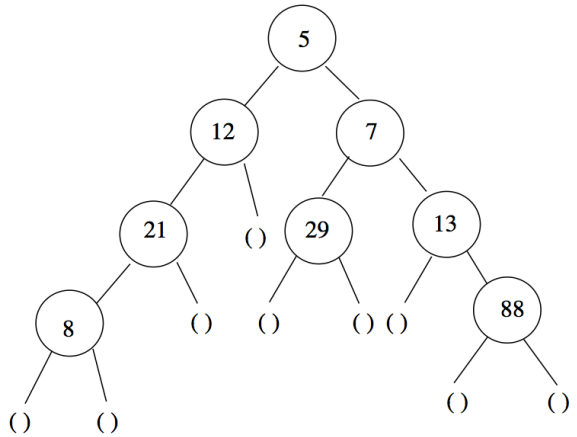
## Somme des éléments d'un arbre contenant des nombres

# Algorithme avec accumulation

Somme des éléments d'un arbre contenant des nombres

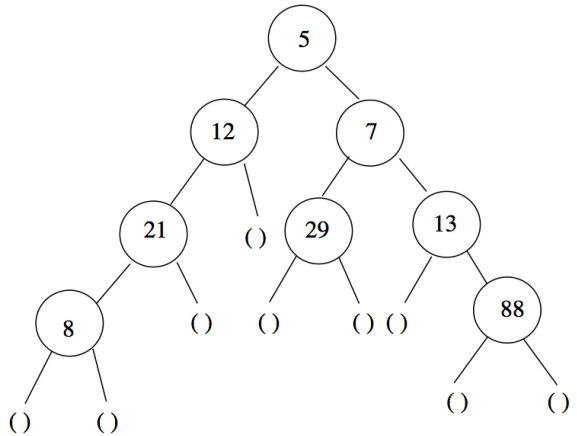
```
somme({arbre}) ->
    0;
somme({arbre, Elem, Ag, Ad}) ->
    Elem + somme(Ag) + somme(Ad).
```

# Parcours d'un arbre



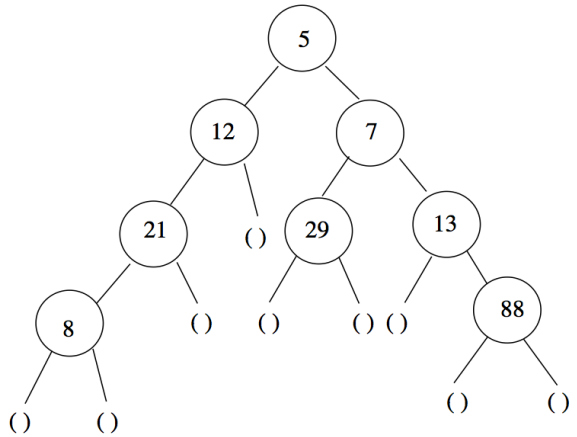

- Parcours préfixe : parcours "vendée globe"
  La racine est traitée avant le fils gauche puis l'arbre droit

# Parcours d'un arbre



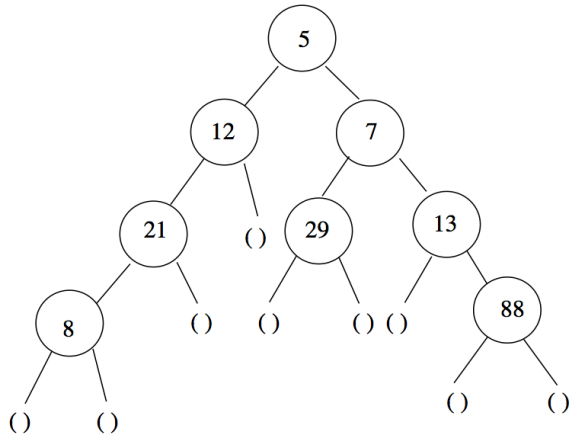

- Parcours préfixe : parcours "vendée globe"
  La racine est traitée avant le fils gauche puis l'arbre droit
  5, 12, 21, 8, 7, 29, 7, 13 , 88
- Parcours infixe : parcours "tout tombe"
  Le fils gauche est avant la racine, puis l'arbre droit

# Parcours d'un arbre




- Parcours préfixe : parcours "vendée globe"
  La racine est traitée avant le fils gauche puis l'arbre droit
  5, 12, 21, 8, 7, 29, 7, 13 , 88
- Parcours infixe : parcours "tout tombe"
  Le fils gauche est avant la racine, puis l'arbre droit
  8, 21, 12, 5, 29, 13, 88
- Parcours postfixe : parcours "à reculons"
  La racine est traitée en dernier après le fils gauche et droit

# Parcours d'un arbre




- Parcours préfixe : parcours "vendée globe"
  La racine est traitée avant le fils gauche puis l'arbre droit
  5, 12, 21, 8, 7, 29, 7, 13 , 88
- Parcours infixe : parcours "tout tombe"
  Le fils gauche est avant la racine, puis l'arbre droit
  8, 21, 12, 5, 29, 13, 88
- Parcours postfixe : parcours "à reculons"
  La racine est traitée en dernier après le fils gauche et droit
  8, 21, 12, 29, 88, 13, 7, 5

# Parcours préfixe

# Parcours préfixe

```
prefixe({arbre}) ->
    io:format("") ;
prefixe({arbre, Elem, Ag, Ad}) ->
    io:format("~B ", [ Elem ]),
    prefixe(Ag),
    prefixe(Ad).
```

# Parcours infixe

# Parcours infixe

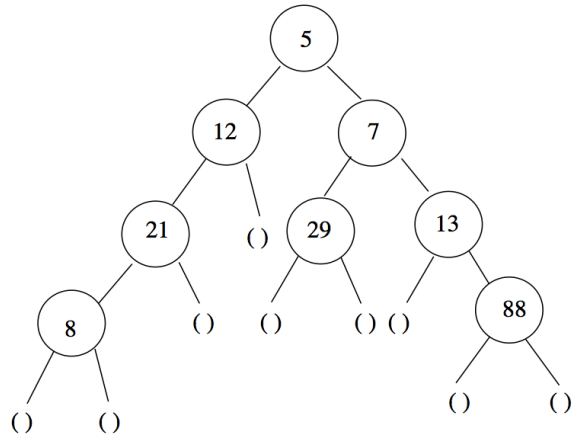
```
infixe({arbre}) ->
    io:format("") ;
infixe({arbre, Elem, Ag, Ad}) ->
    infixe(Ag),
    io:format("~B ", [ Elem ]),
    infixe(Ad).
```

# Parcours postfixe

# Parcours postfixe

```
postfixe({arbre}) ->
    io:format("") ;
postfixe({arbre, Elem, Ag, Ad}) ->
    postfixe(Ag),
    postfixe(Ad),
    io:format("~B ", [ Elem ]).
```

# Parcours en largeur
## Breadth First Search (BSF)




- Parcourir en largeur de l'arbre depuis la racine :
  Traiter la racine puis tous les fils avant les petits fils.
  5, 12, 7, 21, 29, 13, 8, 88

indice : utiliser le principe d'une file ...

# Parcours en largeur
## Breadth First Search (BSF)

```
bfs(Arbre) ->
    bfsAlgo([ Arbre ]).
```

# Parcours en largeur
## Breadth First Search (BSF)

```
bfs(Arbre) ->
    bfsAlgo([ Arbre ]).
```

```
bfsAlgo([ ]) ->
    io:format("") ;
bfsAlgo([ {arbre} | Reste ]) ->
    io:format(""),
    bfsAlgo( Reste ) ;
bfsAlgo([ {arbre, Elem, Ag, Ad} | Reste ]) ->
    io:format("~B ", [ Elem ]),
    bfsAlgo( Reste ++ [ Ag , Ad ] ).
```