Arbres binaires

Programmation Fonctionnelle
Master 2 I2L apprentissage

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

Université du Littoral Côte d'Opale
Laboratoire LISIC
Equipe OSMOSE

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 :

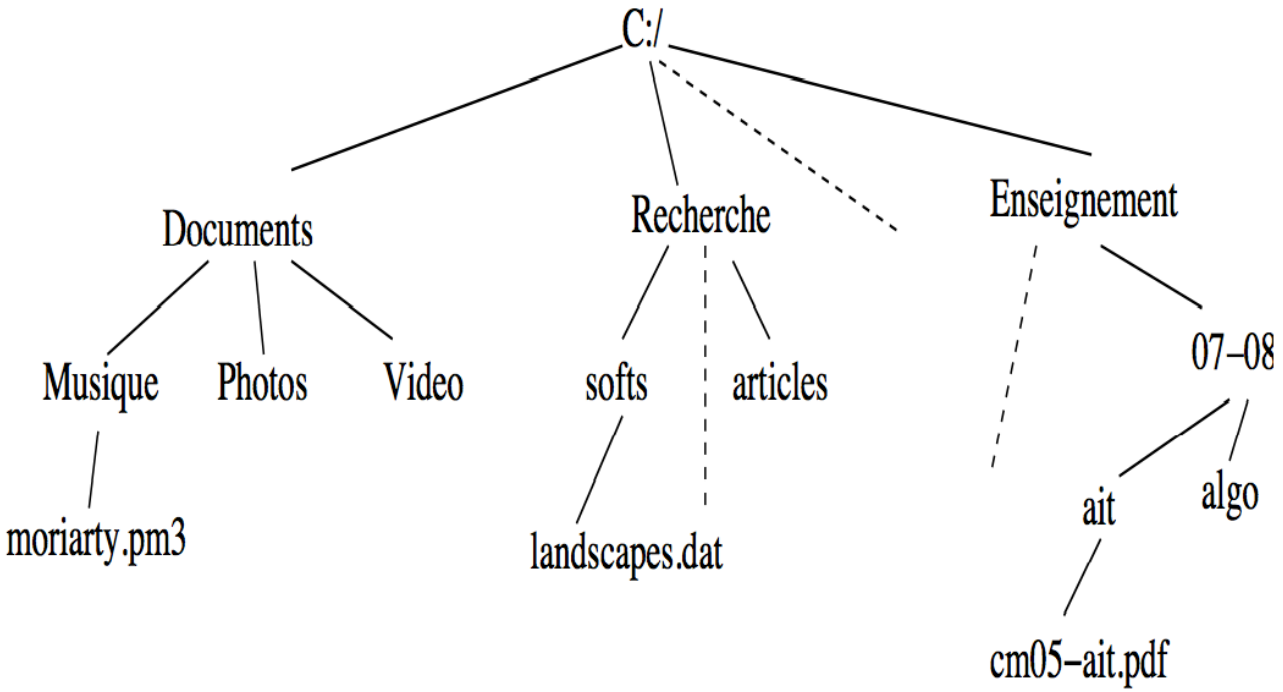
- Fausse liste : succession de tableaux de taille bornée
- Faux tableau : liste avec 'get(i)'

Une autre solution

Structure d'arbre

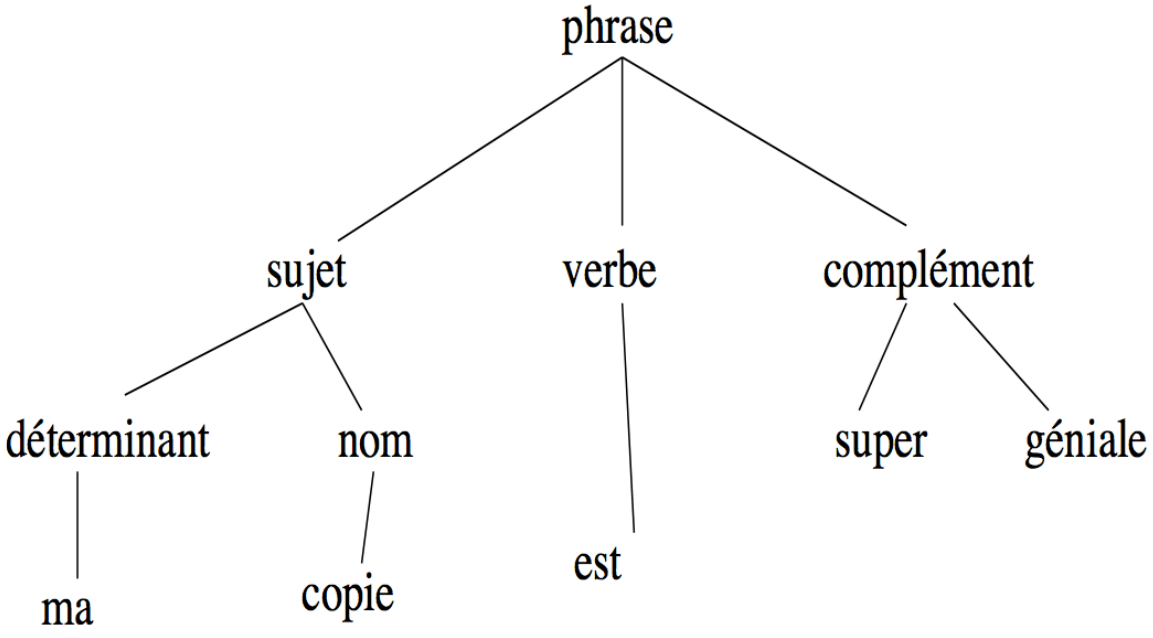
Exemple de données en arbre

Dossiers informatiques



Exemple de données en arbre

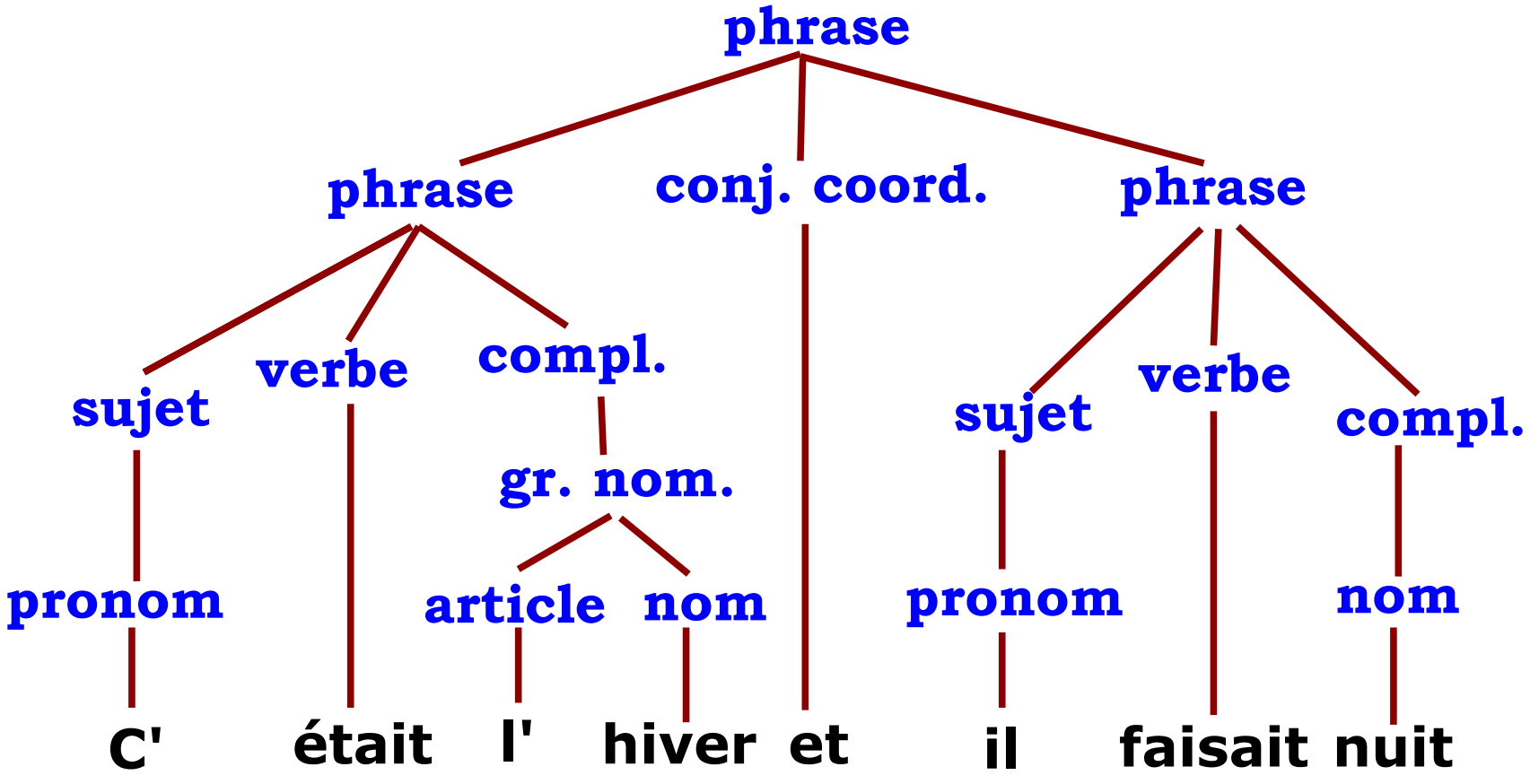
Analyse grammaticale



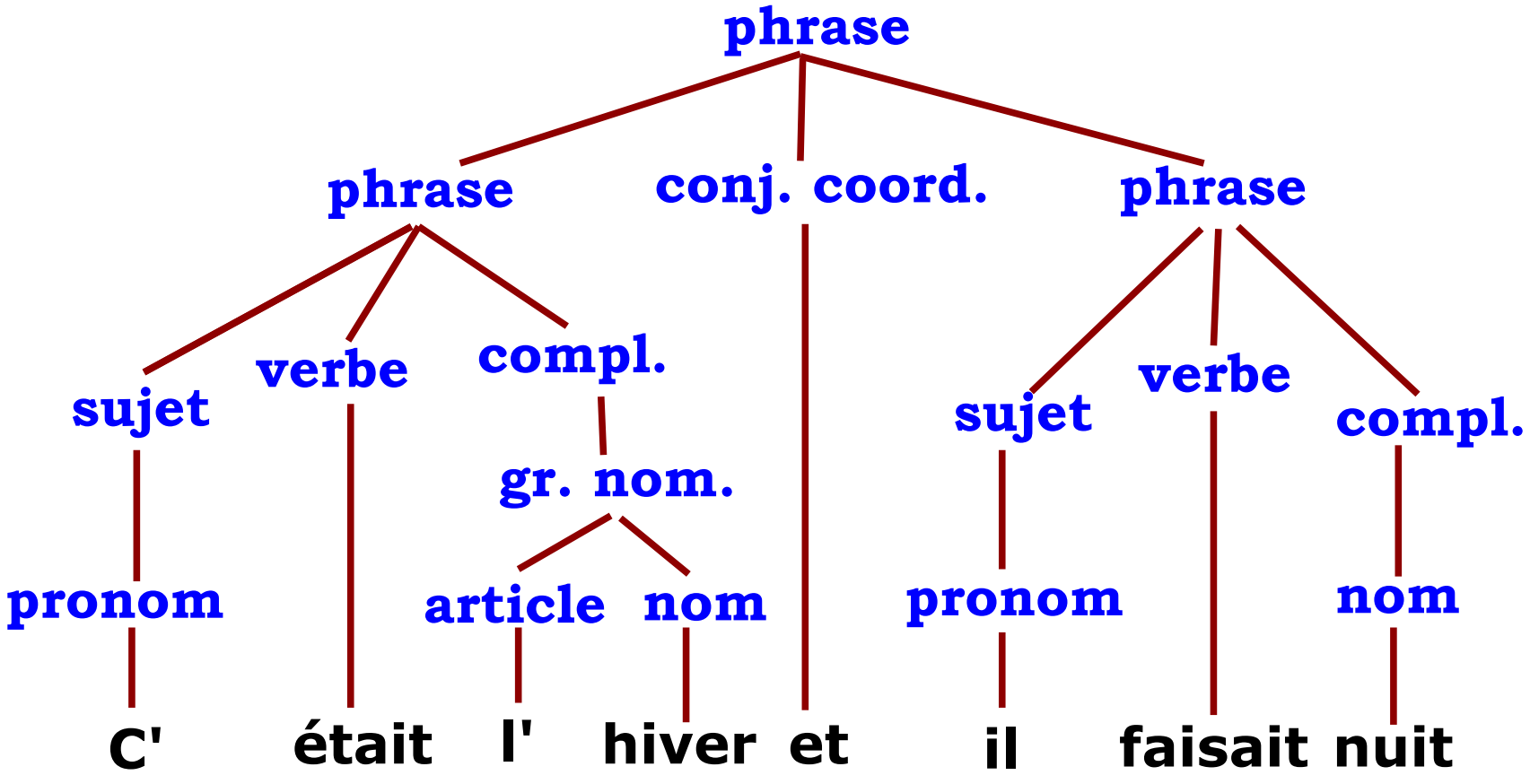
2 types de données :

- Eléments grammaticaux
- Eléments terminaux : les mots

Exemple littéraire



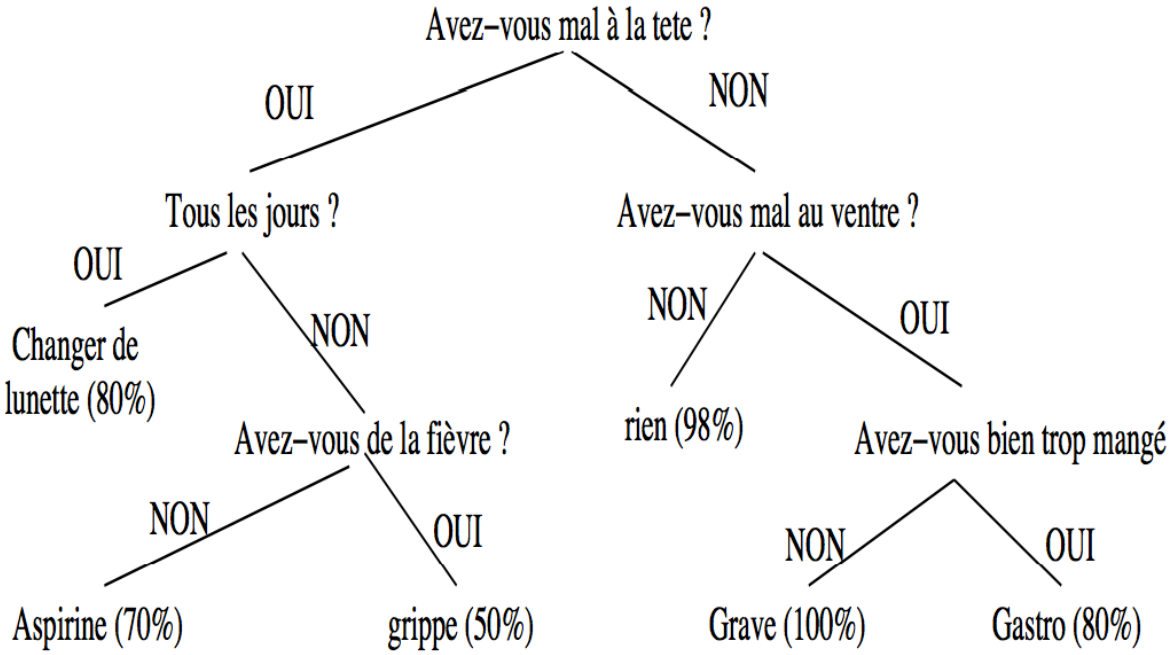
Exemple littéraire



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

Exemple de données en arbre

Arbre de décision

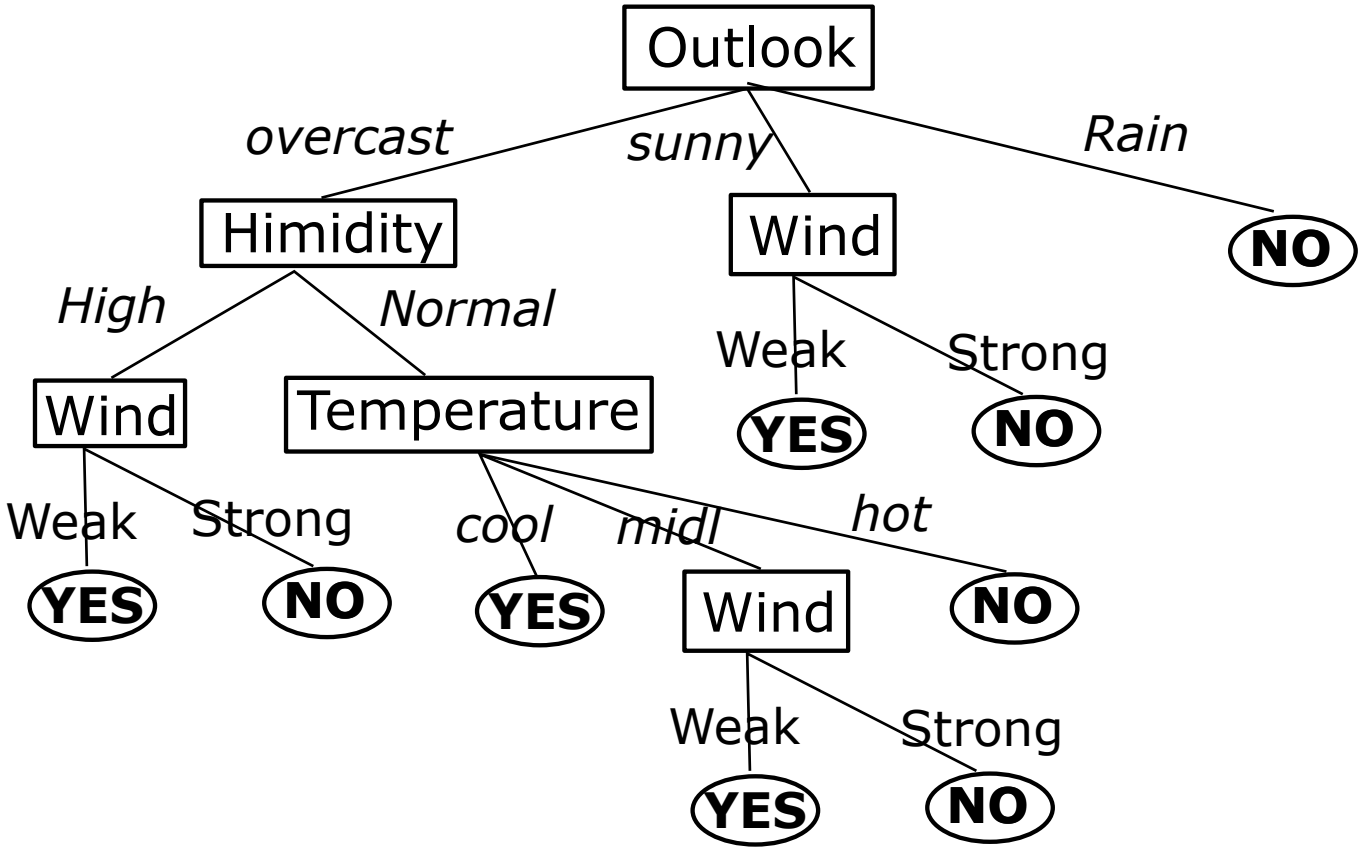


2 types de données :

- les questions
- Eléments terminaux : les décisions

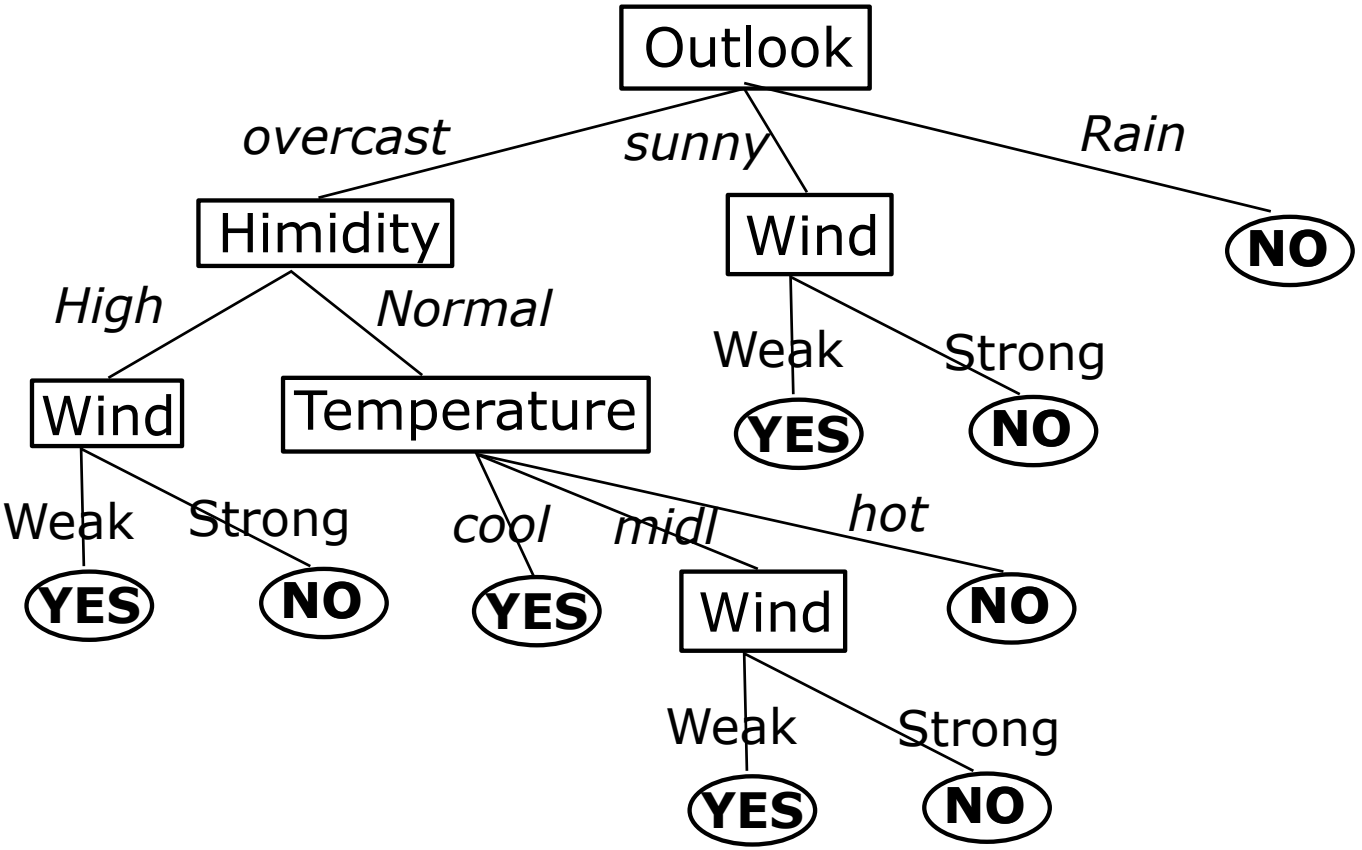
Un arbre pour prendre une décision

Classification à l'aide d'un arbre



Un arbre pour prendre une décision

Classification à l'aide d'un arbre

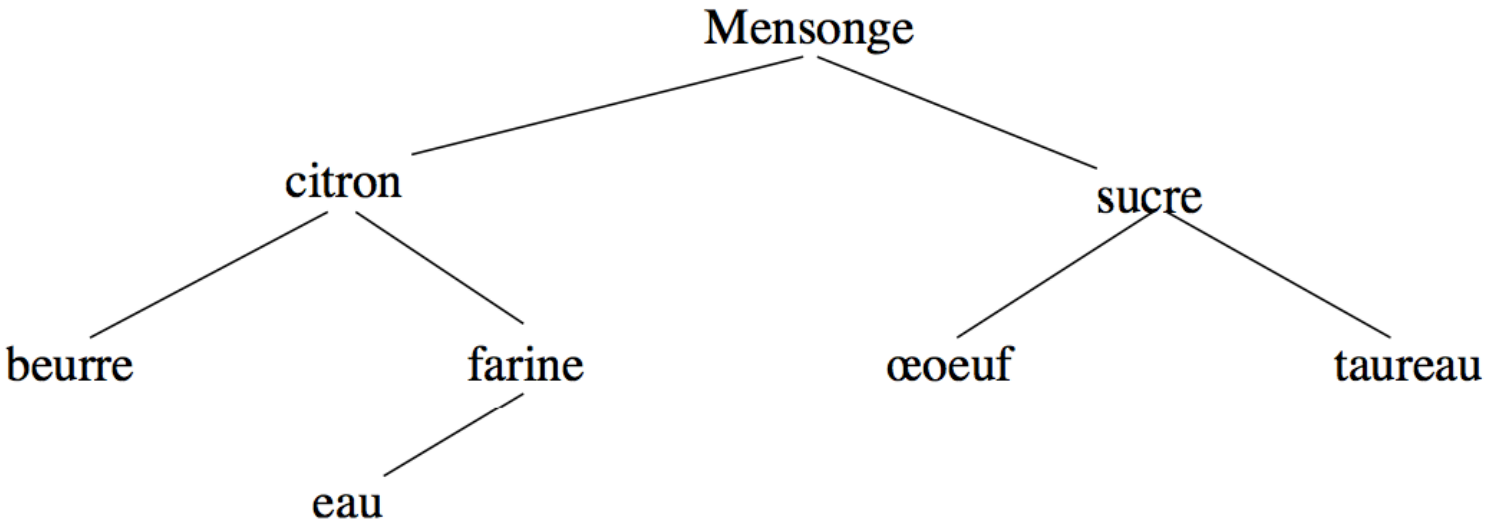


Remarque : un arbre code en fait un ensemble de règles (conjonctions, disjonctions)

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

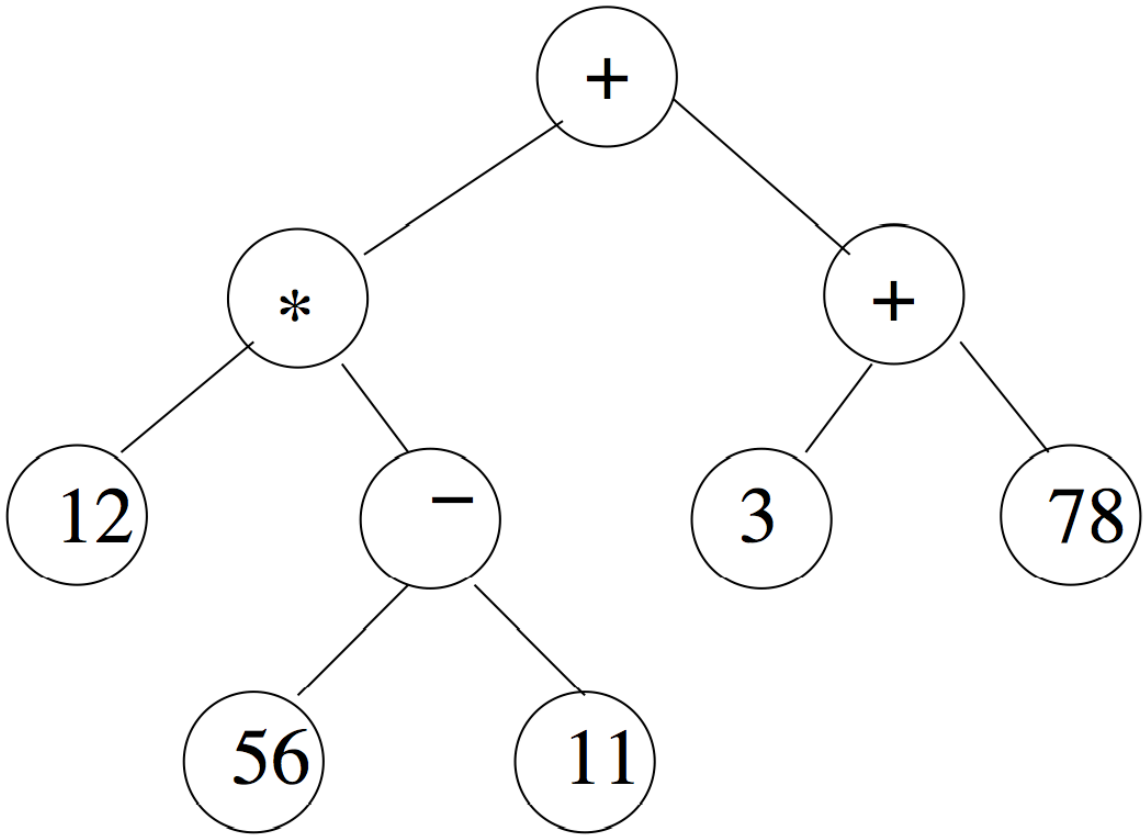
Exemple de données en arbre

Information organisée



Exemple de données en arbre

Expressions arithmétiques



2 types de données :

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

Définitions

Définition informelle

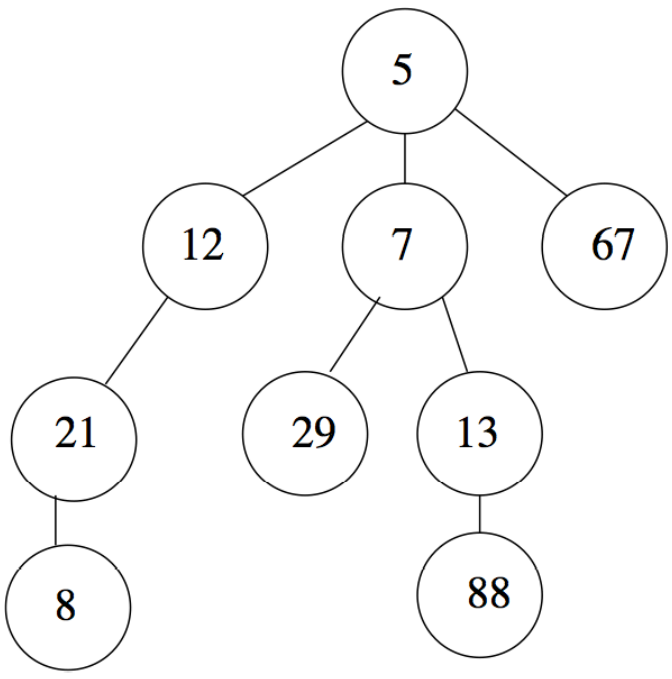
Ensemble de données organisées de manière hiérarchique

Une définition (presque) formelle

Graphe orienté, connexe et acyclique.

Représentation

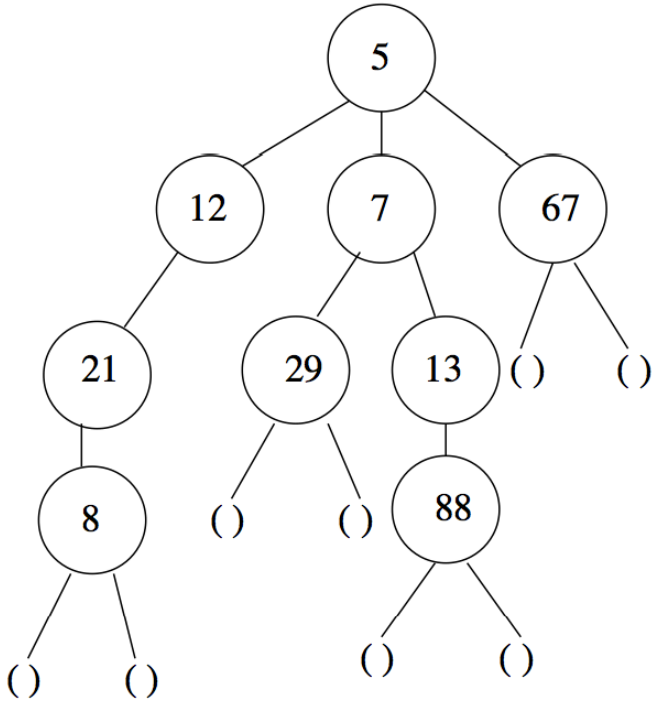
Les arbres (en informatique) se représentent la racine vers le "ciel" !



- 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

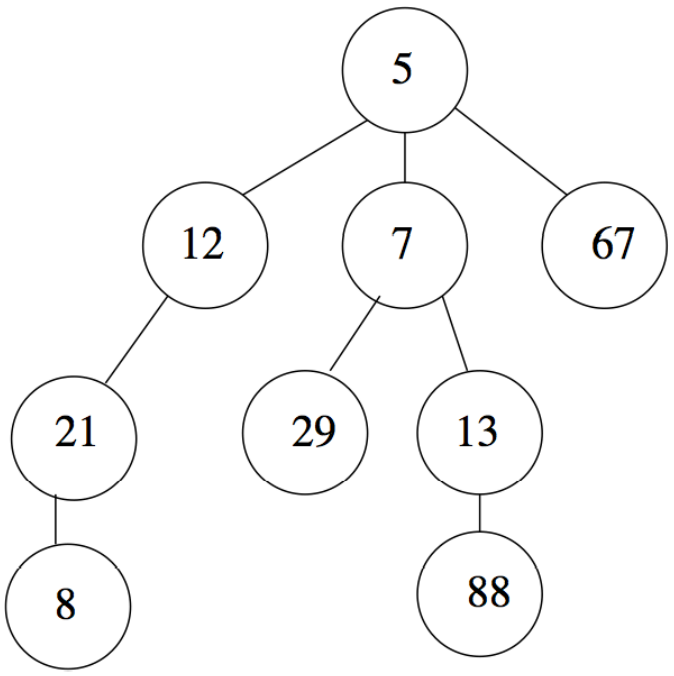
Représentation

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



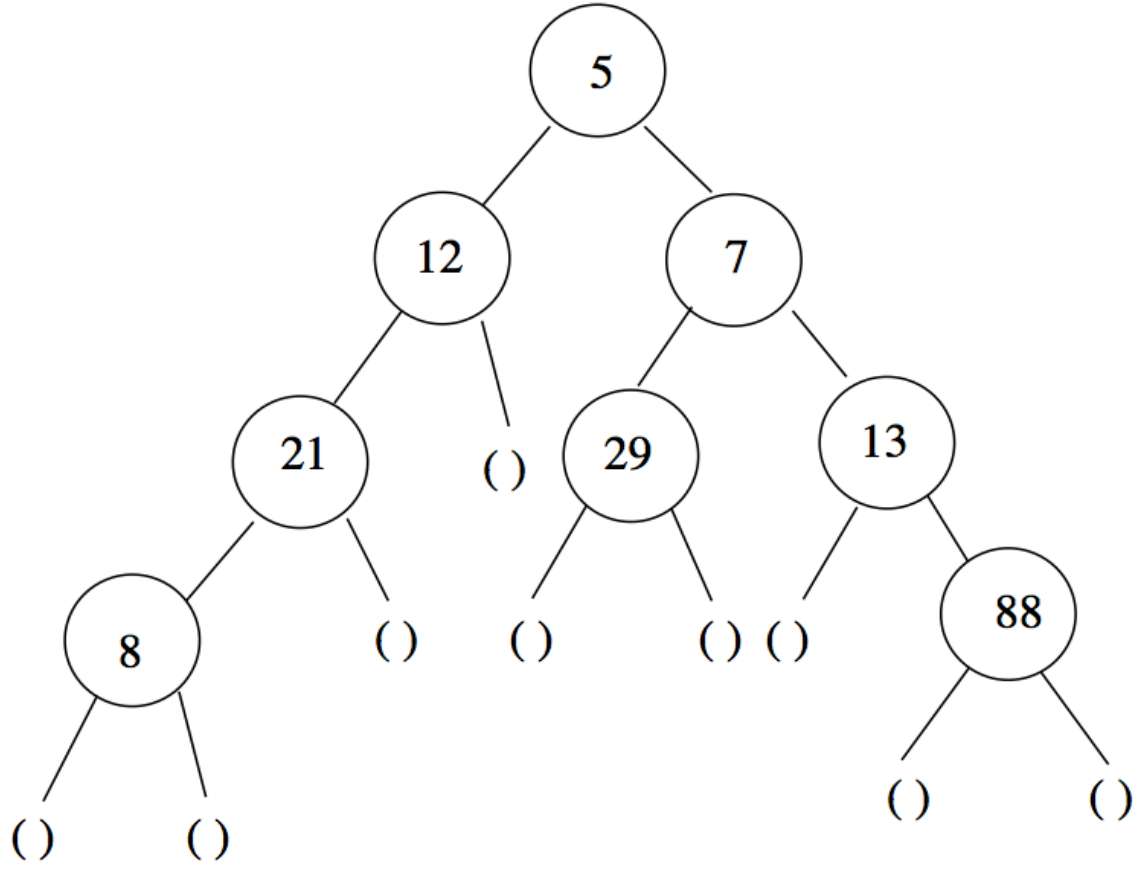
- 5 : racine de l'arbre
- 12, 7, 21, 13, 5 : nœuds internes
- 67, 8, 29, 88 : feuilles
- 12, 7, 67 : fils du nœud 5
- 7 : père de 29 et 13
- le nœud 5 a pour arité 3 et le nœud 21 a pour arité 1
- La hauteur de l'arbre est 4

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

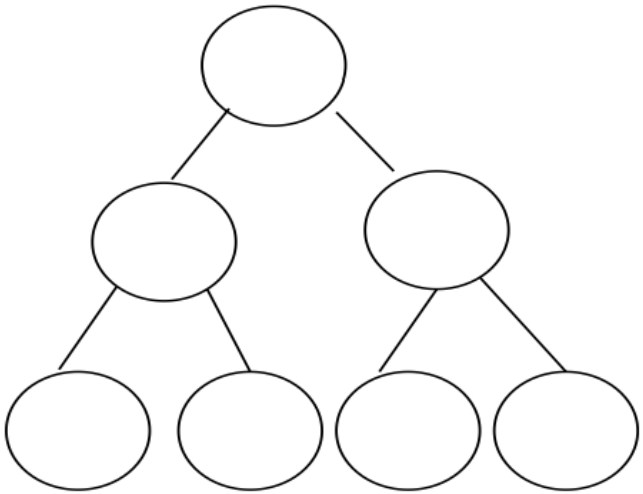
Arbre binaire



Définition

Arbre dont tous les nœud possèdent au plus 2 fils

Arbre binaire



Arbre équilibré

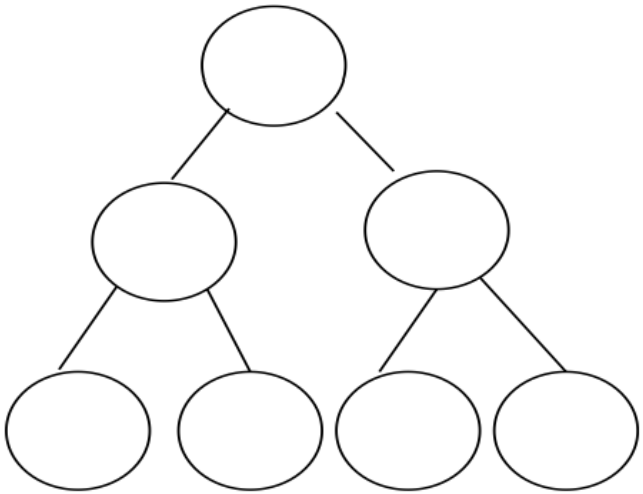
Arbre où chaque nœud a des arbres fils de même hauteur

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()`

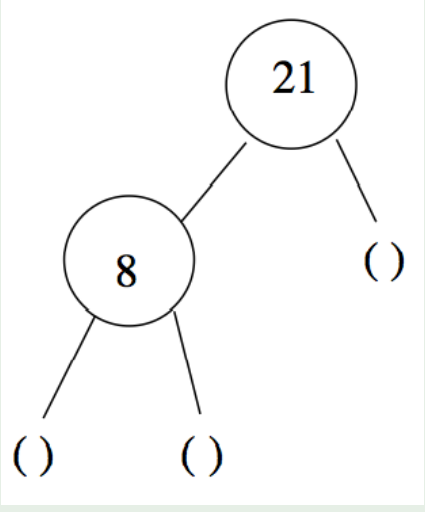
  ()

- `arbreEstVide?(arbreVide())`

  `true`

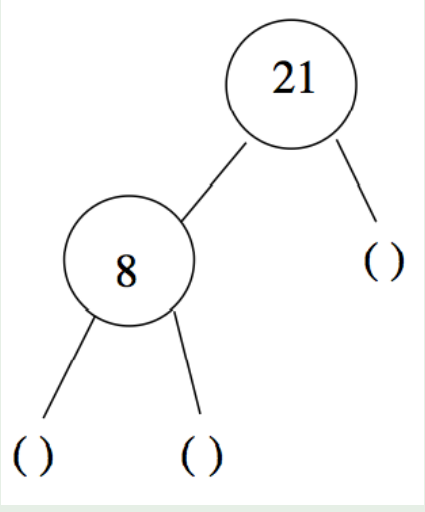
Exemples

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



Exemples

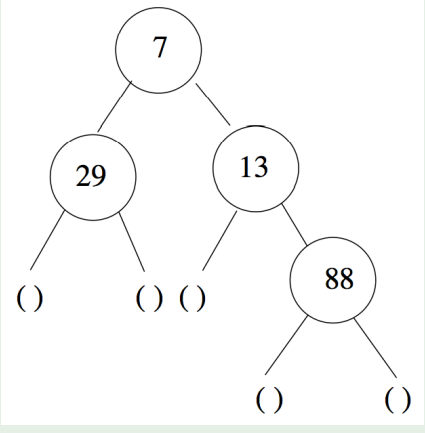
Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?



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

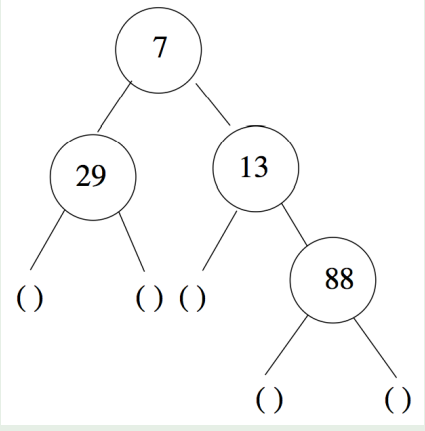
Exemples

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



Exemples

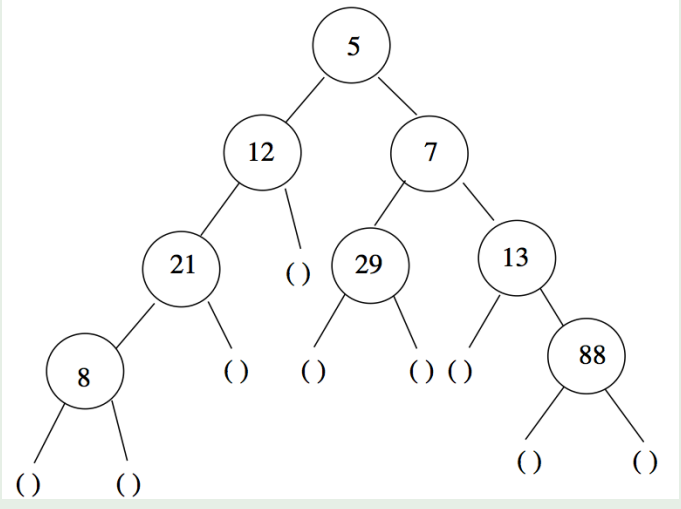
Comment s'écrivent à l'aide des fonctions primitives les arbres suivants ?




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

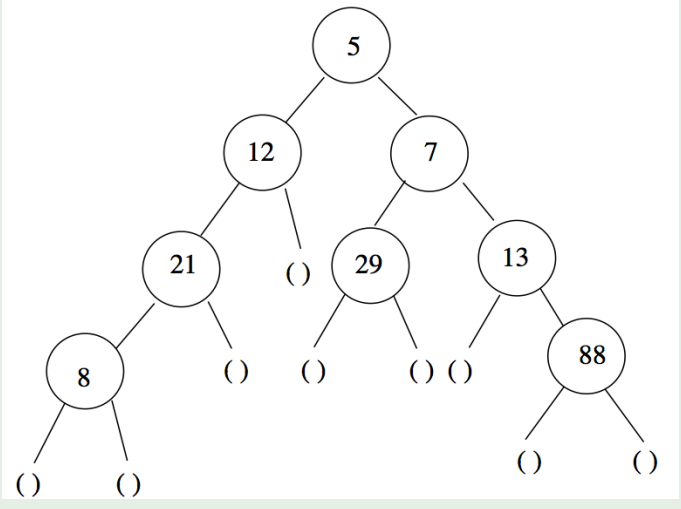
Exemples

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 Haskell

Synthaxe concrète

Un arbre est défini à l'aide d'un type récursif :

```
data Tree a = Empty | Node a (Tree a) (Tree a)
        deriving (Show)
```

Les arbres en Haskell

Synthaxe concrète

Un arbre est défini à l'aide d'un type récursif :

```
data Tree a = Empty | Node a (Tree a) (Tree a)
        deriving (Show)
```

Exercice

Définir les fonctions primitives du TDA arbre.

TDA arbre en Haskell

Les accesseurs

```
root :: Tree a -> a
root (Node x _ _) = x

right :: Tree a -> Tree a
right (Node _ _ d) = d

left :: Tree a -> Tree a
left (Node _ g _) = g
```

TDA arbre en Haskell

Le test

```
isEmpty :: Tree a -> Bool
isEmpty Empty = True
isEmpty _ = False
```

Les arbres en Haskell

Exercice

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

Les arbres en Haskell

Exercice

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

```
isLeaf :: Tree a -> Bool
isLeaf a =  not (isEmpty a) && (isEmpty (left a))
          && (isEmpty (right a))

isLeaf' :: Tree a -> Bool
isLeaf' (Node _ Empty Empty) = True
isLeaf' _ = False
```

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

```
search :: (Eq a) => a -> Tree a -> Bool
search _ Empty = False
search x (Node elem ag ad) = (x == elem) || (search x ag)
```

Recherche arbre non ordonné

Exercice

Recherche d'un élément dans un arbre non ordonné

```
search :: (Eq a) => a -> Tree a -> Bool
search _ Empty = False
search x (Node elem ag ad) = (x == elem) || (search x ag)
```

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

Recherche arbre non ordonné, version TDA

Recherche arbre non ordonné, version TDA

```
search' :: (Eq a) => a -> Tree a -> Bool
search' x t = not (isEmpty t) &&
    ((x == (root t)) ||
      (search x (left t)) || (search x (right t)))
```

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.

```
searchOrd :: (Ord a) => a -> Tree a -> Bool
searchOrd _ Empty = False
searchOrd x (Node elem ag ad) = (x == elem) ||
   if (x < elem) then (searchOrd x ag) else (searchOrd x ad)
```

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.

```
searchOrd :: (Ord a) => a -> Tree a -> Bool
searchOrd _ Empty = False
searchOrd x (Node elem ag ad) = (x == elem) ||
   if (x < elem) then (searchOrd x ag) else (searchOrd x ad)
```

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

```
numberOfNodes :: Tree a -> Int
numberOfNodes Empty = 0
numberOfNodes (Node _ ag ad) = 1 +
          (numberOfNodes ag) + (numberOfNodes ad)
```

Nombre de nœuds

Calcul du nombre de nœuds

```
numberOfNodes :: Tree a -> Int
numberOfNodes Empty = 0
numberOfNodes (Node _ ag ad) = 1 +
          (numberOfNodes ag) + (numberOfNodes ad)
```

Prototype de la fonction récursive sur les arbres

```
myFunction Empty =
    traitement arbre vide
myFunction (Node elem ag ad) ->
    traitement noeud
    ... myFunction ag ... myFunction ad ...
    .
```

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

```
height :: Tree a -> Int
height Empty = 0
height (Node elem ag ad) = 1 +
     max (height ag) (height ad)
```

Algorithme avec accumulation

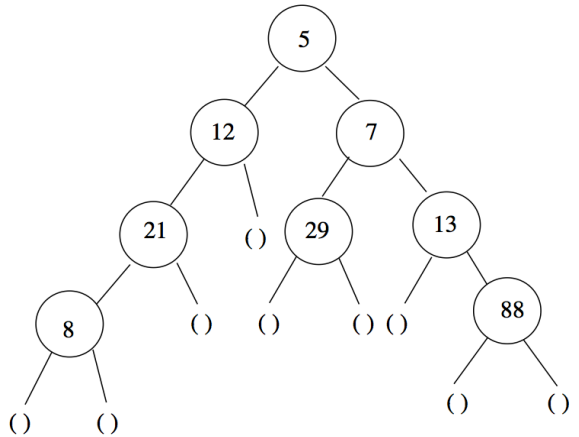
Somme des éléments d'un arbre contenant des nombres

Algorithme avec accumulation

Somme des éléments d'un arbre contenant des nombres

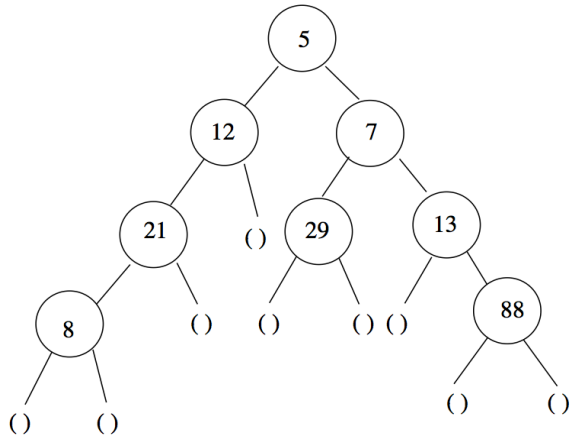
```
sumTree :: Integral a => Tree a -> a
sumTree Empty = 0
sumTree (Node elem ag ad) = elem +
     (sumTree ag) + (sumTree ad)
```

Parcours d'un arbre



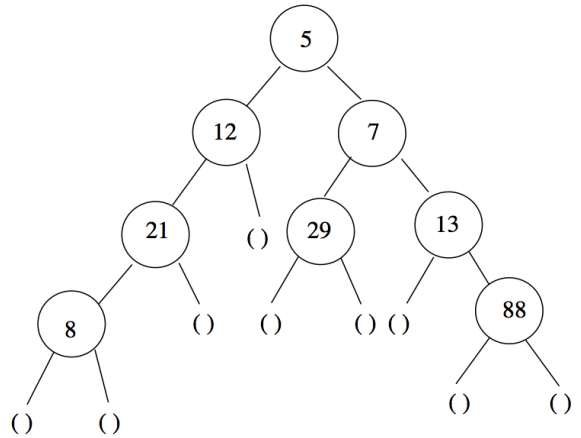
- Parcours préfixe : parcours "vendée globe"
  La racine est traitée avant le fils gauche puis l'arbre droit

Parcours d'un arbre



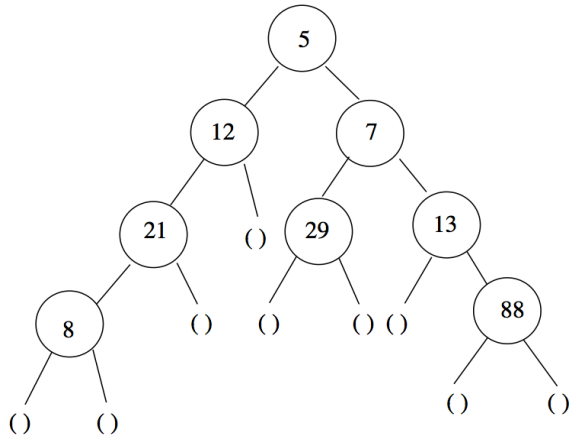
- Parcours préfixe : parcours "vendée globe"
  La racine est traitée avant le fils gauche puis l'arbre droit
  5, 12, 21, 8, 7, 29, 13, 88
- Parcours infixe : parcours "tout tombe"
  Le fils gauche est avant la racine, puis l'arbre droit

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, 13, 88
- Parcours infixe : parcours "tout tombe"
  Le fils gauche est avant la racine, puis l'arbre droit
  8, 21, 12, 5, 29, 7, 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, 13, 88
- Parcours infixe : parcours "tout tombe"
  Le fils gauche est avant la racine, puis l'arbre droit
  8, 21, 12, 5, 29, 7, 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' :: Tree a -> [a]
prefixe' Empty = []
prefixe' (Node elem ag ad) = elem:(prefixe' ag) ++
    (prefixe' ad)
```

Parcours infixe

Parcours infixe

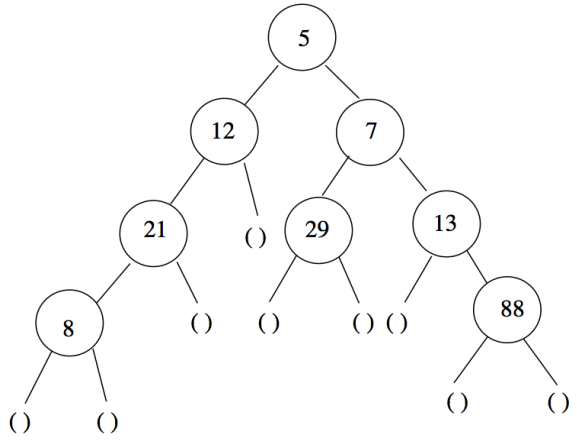
```
infixe' :: Tree a -> [a]
infixe' Empty = []
infixe' (Node elem ag ad) = (infixe' ag) ++
    (elem:(infixe' ad))
```

Parcours postfixe

Parcours postfixe

```
postfixe' :: Tree a -> [a]
postfixe' Empty = []
postfixe' (Node elem ag ad) = (postfixe' ag) ++
    (postfixe' ad) ++ [ 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 [] = []
bfsAlgo (Empty:tail) = bfsAlgo tail
bfsAlgo ((Node elem lt rt):tail) =
   elem : bfsAlgo (tail ++ [lt, rt])
```