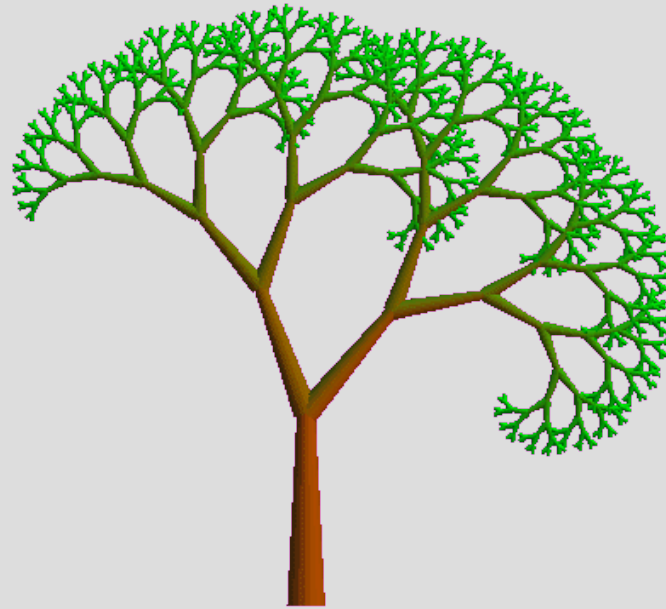


# Qu'est-ce qu'un arbre ?



# Vocabulaire

Les arbres sont composés de deux éléments de base :

les nœuds (ou vertex),

un nœud peut être étiqueté ou non : c'est-à-dire porter une ou plusieurs valeurs,

les arcs (ou arêtes) qui relient les nœuds (un arc relie deux nœuds différents),

un arc peut être pondéré ou non : c'est-à-dire porter une ou plusieurs valeurs.

# Vocabulaire

De manière informelle, nous définirons un chemin entre deux nœuds comme :

un ensemble d'arcs permettant de relier ces deux nœuds.

Par définition, un arc ne peut intervenir qu'une seule fois dans un chemin donné.

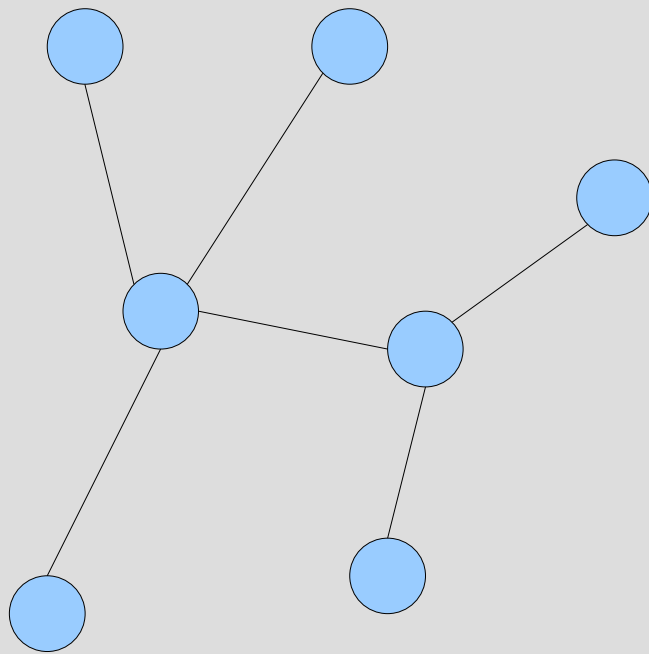
# Définitions

Dans la théorie des graphes, deux notions différentes sont considérées :

la notion d'arbre dont les arcs ne sont pas orientés (ou bidirectionnels),

et celle d'arborescence dont les arcs sont orientés.

# Un arbre



Un arbre est un graphe non orienté acyclique, c'est-à-dire qu'il n'existe qu'un chemin unique entre deux nœuds.

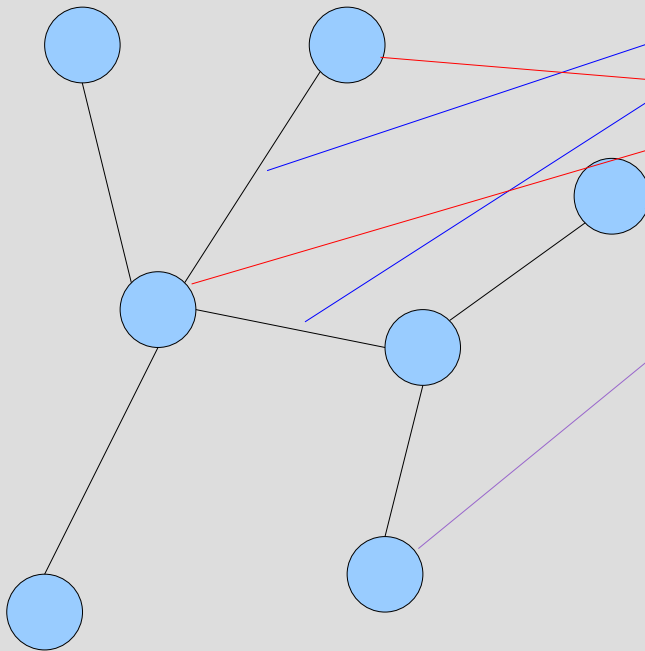
# Un arbre

Il se compose :

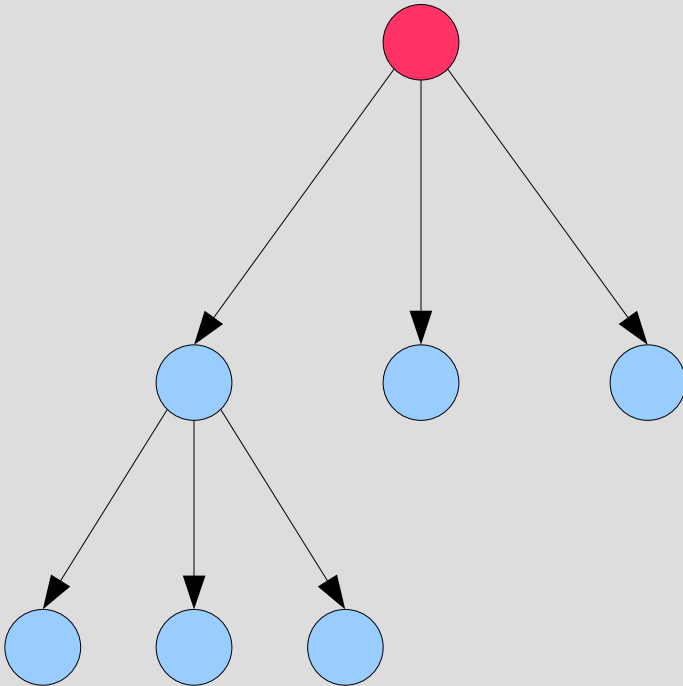
d'arcs

de nœuds,

les nœuds n'ayant qu'un seul voisin sont appelés feuille.



# Une arborescence



Une arborescence est un arbre contenant :

une racine,

des arcs orientés de la racine vers les feuilles.

**une feuille est un nœud qui n'a pas de fils**

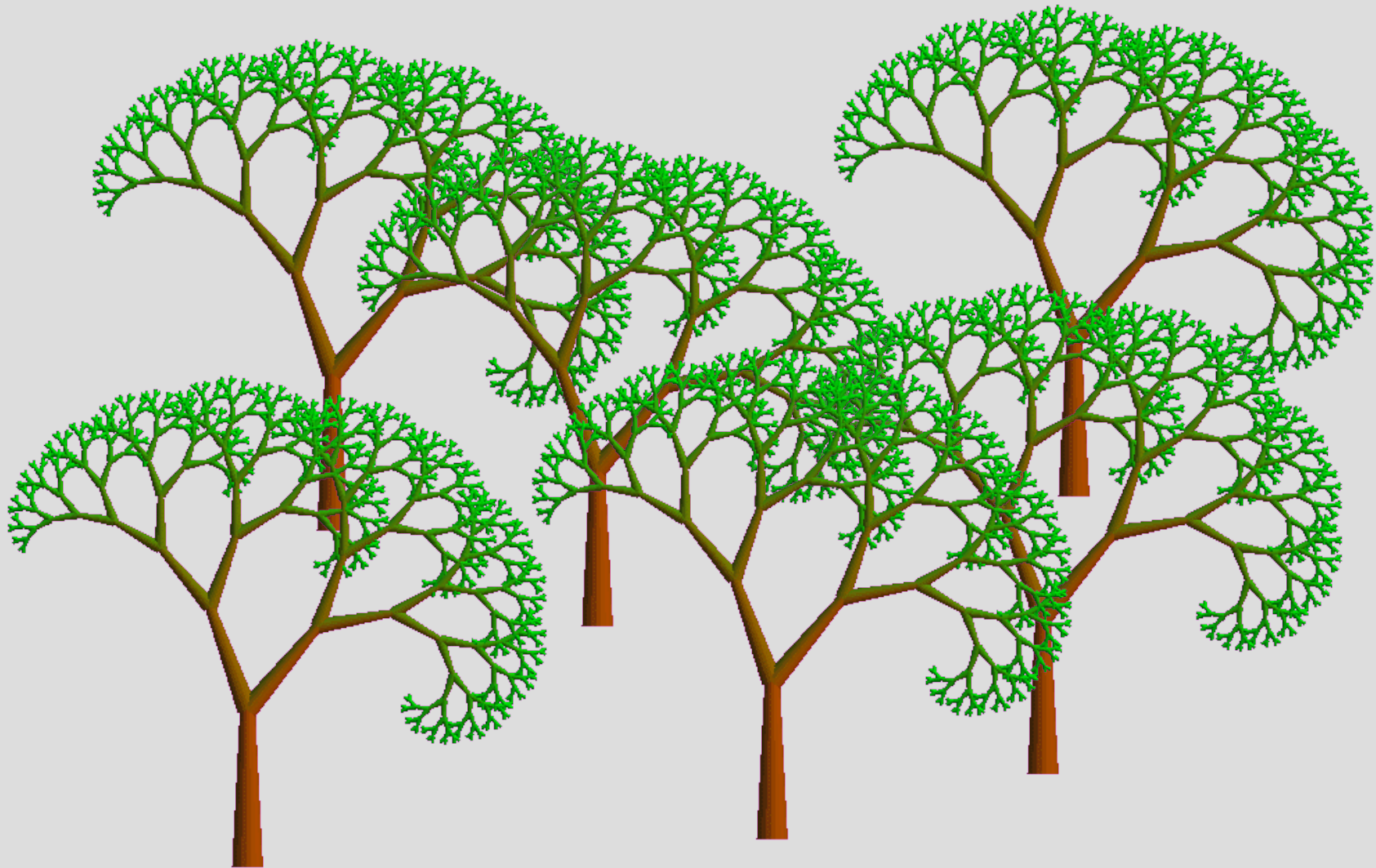
# Arbre et arborescence

La transformation de l'un à l'autre est triviale au niveau des concepts.

Le passage d'arbre à arborescence consiste à choisir un nœud racine (l'orientation des arcs en découlera).

Le passage inverse consiste à oublier la racine et les orientations des arcs.

# À quoi servent les arbres ?



# À quoi servent les arbres ?

Organisation de systèmes de fichiers.

Organisations hiérarchiques.

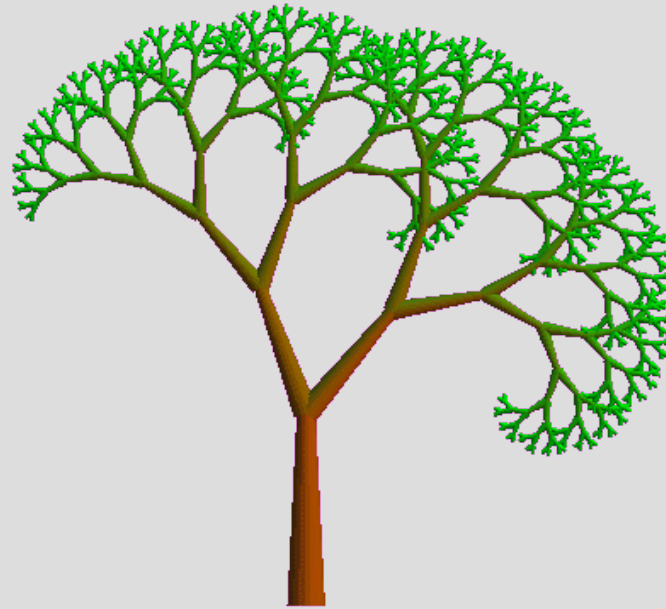
Ordonnancement de données.

Recherche de données.

Réalisation de programmes.

...

# Les arbres en Scheme ?



# Aparté : define-struct

Il est possible en Scheme de définir des structures complexes via la fonction :

define-struct

Syntaxe :

```
(define-struct nom (nom-elem1 nom-elem2...))
```

Exemple :

```
(define-struct complexe (reel image))
```

# Aparté : define-struct

Avec la création d'une structure, différentes fonctions sont automatiquement créées :

un constructeur :

make-*nom*

des accesseurs :

*nom-nom-elem1, nom-nom-elem2, nom-...*

un test

*nom?*

et quelques autres...

# Aparté : define-struct

; Un complexe est constitué d'une partie réelle

; et d'une partie imaginaire

(define-struct complexe (reel image))

; a contient le complexe  $1+2i$

(define a (make-complexe 1 2))

; test si a est un complexe

(complexe? a)

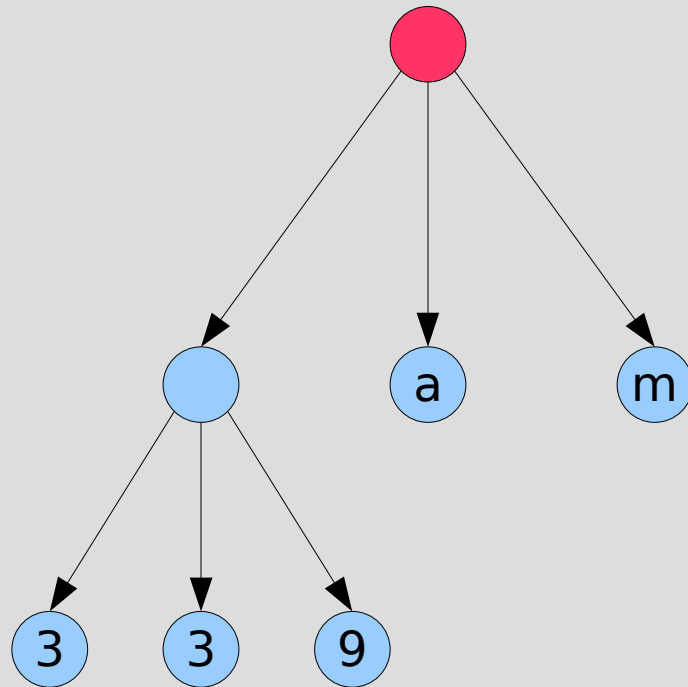
; rend la partie réelle de a

(complexe-reel a)

; rend la partie imaginaire de a

(complexe-image a)

# Première arborescence en Scheme



Soit l'arborescence ci-  
contre portant  
uniquement des  
étiquettes sur les  
feuilles.

Comment le  
représenter en  
Scheme ?

# Première arborescence en Scheme

Comme dans tout programme, la première étape est l'analyse et la définition des structures de données.

D'où : considèrerons-nous qu'une arborescence est :

un ensemble de nœuds et un ensemble d'arcs ?

un ensemble de nœuds, un autre de feuilles et un dernier d'arcs ?

# Première arborescence en Scheme

Considérons-nous qu'une arborescence est :

un ensemble de nœuds où un nœud contient  
l'ensemble des arcs dont il est l'origine ?

un nœud contenant l'ensemble de ses sous-  
arborescences ?

et comment allons-nous gérer les étiquettes ?

# Première arborescence en Scheme

Premier cas :

un ensemble de nœuds numérotés (ex : (1, 2, 3, 4, 5, 6, 7) ),

un ensemble d'arcs (paire de nœuds) (ex : ((1.2), (1.3), (1.4), (2.5), (2.6), (2.7)) ),

un ensemble d'étiquettes (paire de nœud et valeur) pour représenter les feuilles étiquetées (ex : ((3.a), (4.m), (5. 3), (6.3), (7.9)) ).

# Première arborescence en Scheme

Premier cas en Scheme :

L'arborescence :

```
; ANALYSE ET DEFINITION DES DONNEES  
(define-struct arborescence (node edge label))  
; une arborescence est une structure :  
;(make-arborescence node edge label)  
; où node est une liste de noeuds, edge une liste  
; d'arcs et label une liste d'étiquettes.  
(define arbo1 (make-arborescence (list 1 2 3 4 5 6 7)  
(list '(1 . 2) '(1 . 3) '(1 . 4) '(2 . 5) '(2 . 6) '(2  
. 7)) (list '(3 . 'a) '(4 . 'm) '(5 . '3) '(6 . '3)  
'(7 . '9))))  
; TEMPLATE  
; (define (foo-arborescence arbo)  
; (cond [(..(arborescence-node arbo)  
; ..(arborescence-edge arbo)..
```

# Une autre arborescence en Scheme

Second cas étudié :

une arborescence est un nœud ou une feuille,

un nœud est une liste de sous-arborescences ou de feuilles,

une feuille est un symbole, le symbole représente l'étiquette.

Rappel : l'arborescence étudiée ne possède des étiquettes que sur ses feuilles.

# Une autre arborescence en Scheme

Second cas étudié en Scheme :

La feuille :

```
; ANALYSE ET DEFINITION DES DONNEES  
(define-struct leaf (content))  
; une feuille est une structure : (make-leaf content)  
; où content est l'étiquette de la feuille.  
(define leaf1 (make-leaf 'a))  
(define leaf2 (make-leaf 'm))  
(define leaf3 (make-leaf '3))  
(define leaf4 (make-leaf '3))  
(define leaf5 (make-leaf '9))  
; TEMPLATE  
; (define (foo-leaf l)  
; (cond [(..(leaf-content l) ...
```

# Une autre arborescence en Scheme

Second cas étudié en Scheme :

Le nœud :

```
; ANALYSE ET DEFINITION DES DONNEES  
(define-struct node (sub-tree))  
; un noeud est une structure : (make-node sub-tree)  
; où sub-tree est la liste des sous-arborescences du  
; noeud.  
(define node1 (make-node (list st1 leaf1 leaf2)))  
; TEMPLATE  
; (define (foo-node n)  
; (cond [(..(leaf-sub-tree n) ...
```

# Une autre arborescence en Scheme

Second cas étudié en Scheme :

L'arborescence (ou sous-arborescence) :

```
; ANALYSE ET DEFINITION DES DONNEES  
; une arborescence (ou sous-arborescence) est  
; soit un noeud soit une feuille  
(define st1 (make-node (list leaf3 leaf4 leaf5)))  
; TEMPLATE  
; (define (foo-arborescence arbo)  
; (cond ((leaf? Arbo) ... )  
;       ((node? Arbo) ... ) ...
```

# Une dernière arborescence en Scheme

Dernier cas étudié (en cours), une version très Scheme :

une arborescence est une liste de sous-arborescences ou de feuilles,

une feuille est un symbole, le symbole représente l'étiquette.

Rappel : l'arborescence étudiée ne possède des étiquettes que sur ses feuilles.

# Une dernière arborescence en Scheme

Dernier cas étudié en Scheme :

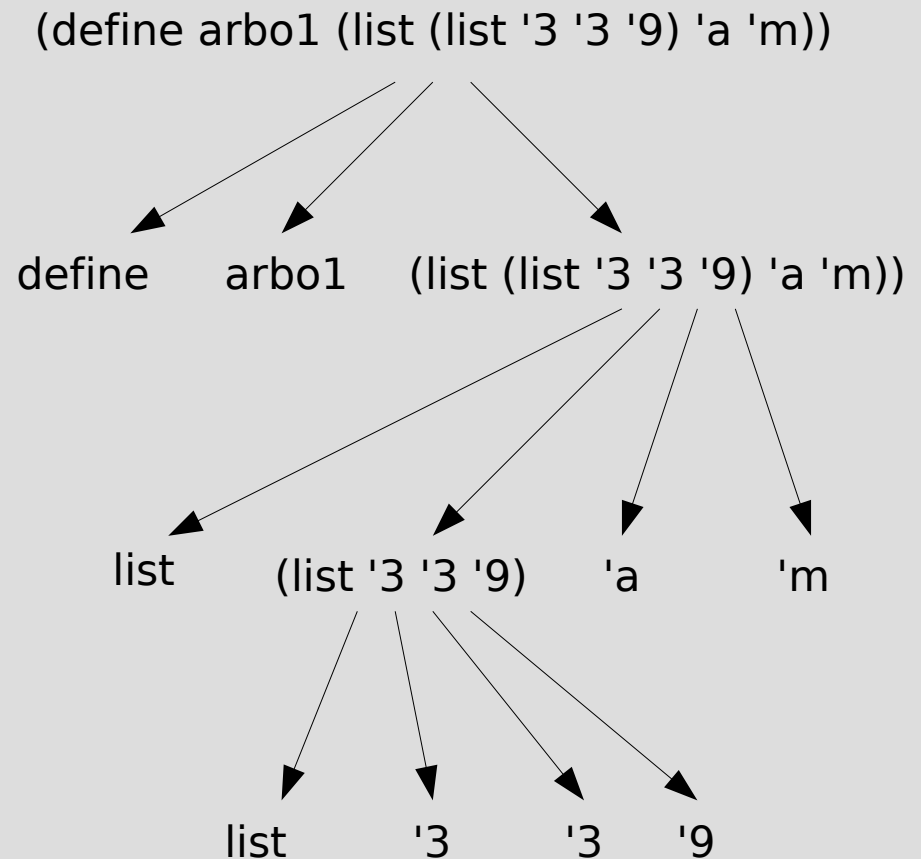
L'arborescence (ou sous-arborescence) :

```
; ANALYSE ET DEFINITION DES DONNEES  
; une arborescence (ou sous-arborescence) est  
; une liste de sous-arborescences ou de feuilles  
(define arbo1 (list (list '3 '3 '9) 'a 'm))  
; TEMPLATE  
; (define (foo-arborescence arbo)  
; (cond ((symbol? arbo) ... )  
;       ((list? arbo) ... ) ...
```

# Exemple pratique d'arborescence.

Un programme se représente sous forme d'arborescence.

Son interprétation équivaut à un parcours de cette arborescence.



# Arbres et abstraction

Le choix d'une représentation de données peut avoir un impact très important sur la programmation.

Il est donc nécessaire de proposer des primitives permettant de faire abstraction des données.

Ci-après quelques primitives utiles à la manipulation d'arbres et arborescences.

# Arbres et primitives

valeur : noeud -> objet

vide : -> arbre

vide? : arbre -> booléen

fil : noeud -> list

car-fil : noeud -> noeud

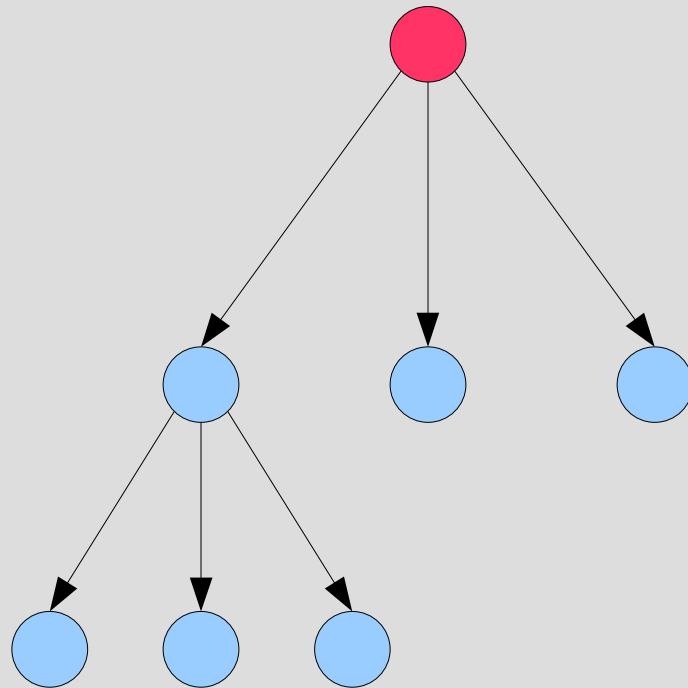
cdr-fil : noeud -> list

cons-noeud : objet -> noeud

ajout-fil : noeud x noeud -> noeud

...

# Parcours en profondeur

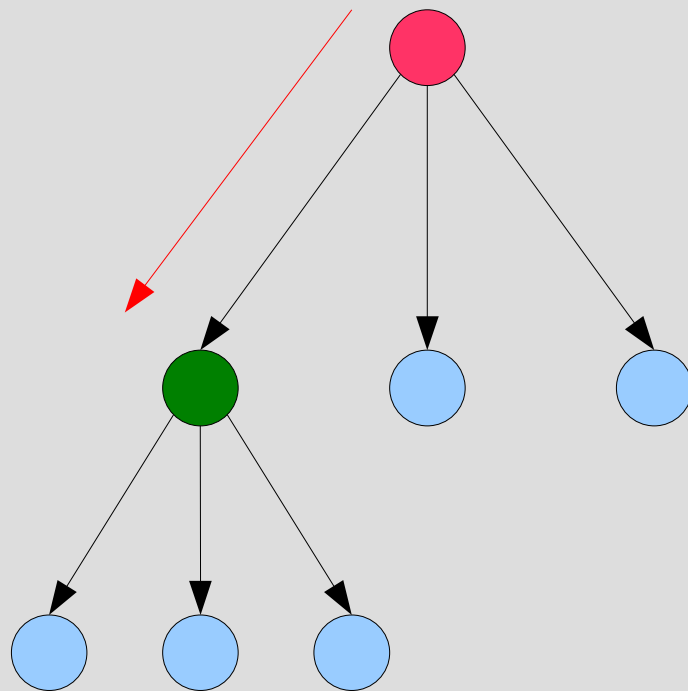


Ce parcours consiste à descendre le plus profond possible

Par la gauche ou la droite

Avant d'explorer la branche suivante.

# Parcours en profondeur

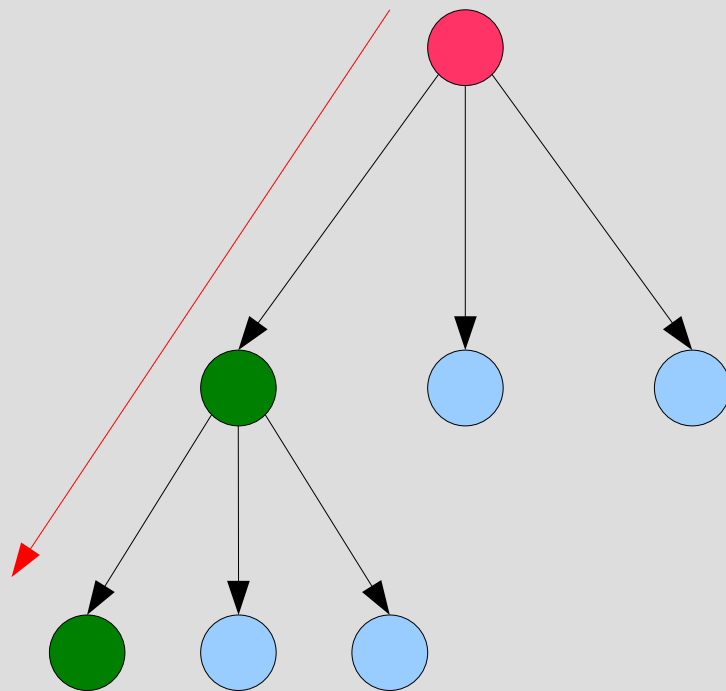


Ce parcours consiste à descendre le plus profond possible

Par la gauche ou la droite

Avant d'explorer la branche suivante.

# Parcours en profondeur

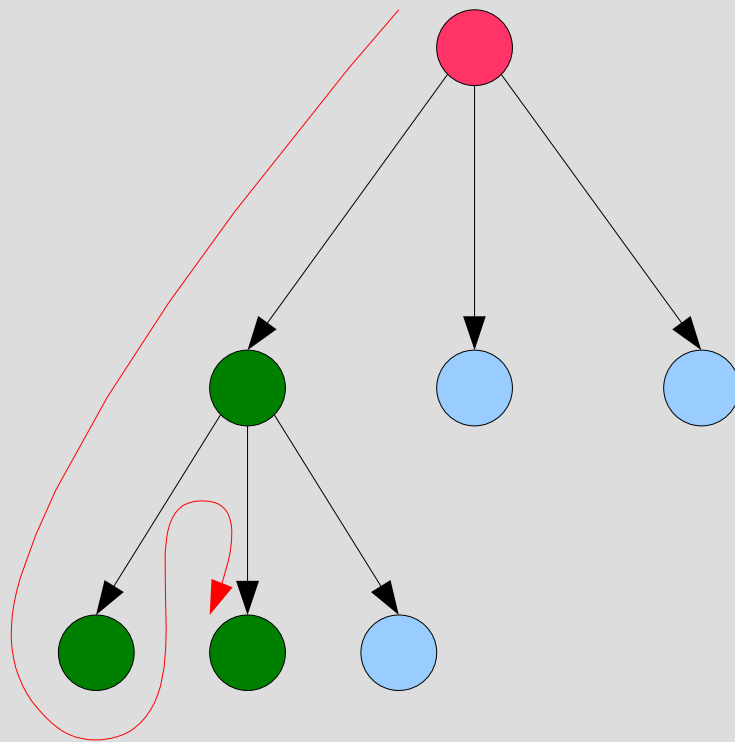


Ce parcours consiste à descendre le plus profond possible

Par la gauche ou la droite

Avant d'explorer la branche suivante.

# Parcours en profondeur

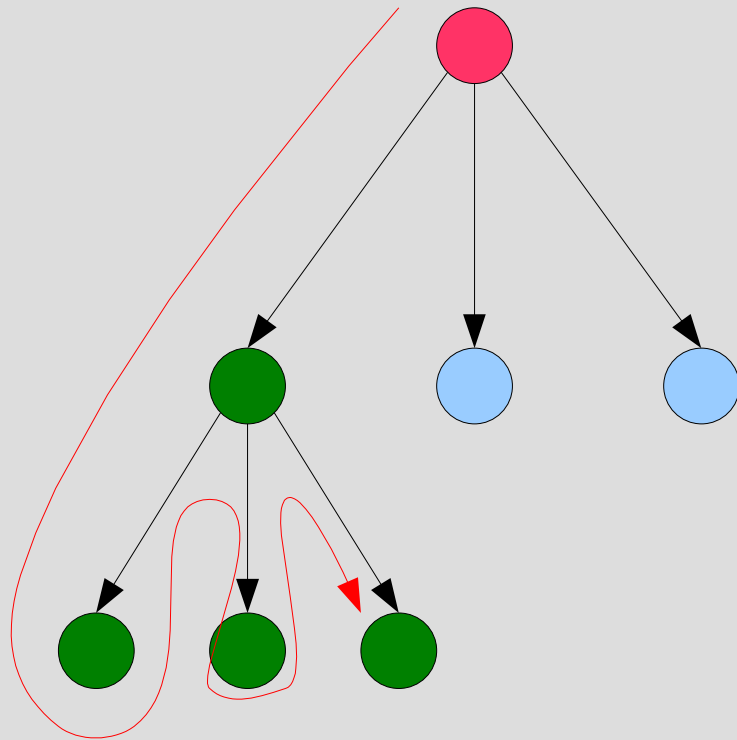


Ce parcours consiste à descendre le plus profond possible

Par la gauche ou la droite

Avant d'explorer la branche suivante.

# Parcours en profondeur

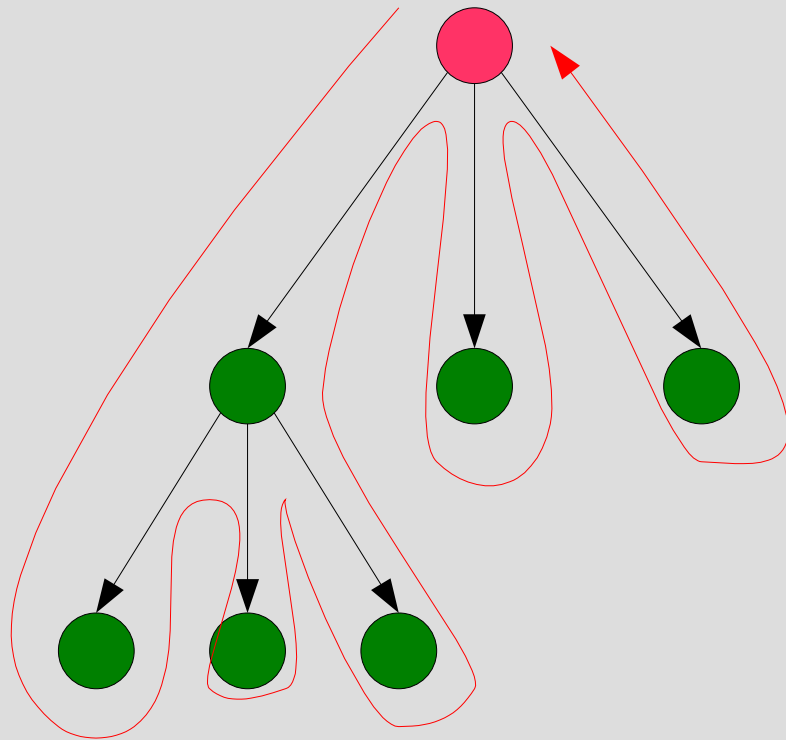


Ce parcours consiste à descendre le plus profond possible

Par la gauche ou la droite

Avant d'explorer la branche suivante.

# Parcours en profondeur



Ce parcours consiste à descendre le plus profond possible

Par la gauche ou la droite

Avant d'explorer la branche suivante.

# Parcours en profondeur en Scheme

Choix du parcours :

en profondeur par la gauche

pour des raisons de facilité (utilisation des fonctions `car` et `cdr` ou équivalente)

utilisation d'un parcours récursif

implantation plus facile en Scheme (la récursivité est intégrée à la philosophie de programmation en Scheme)

# Parcours en profondeur en Scheme

Principe récursif :

pour un nœud donné

si c'est une feuille

traitement de la feuille.

si ce n'est pas une feuille

parcours du fils le plus à gauche non traité

parcours des  $n-1$  fils le plus à droite non traités

# Parcours en profondeur en Scheme : code (1/2)

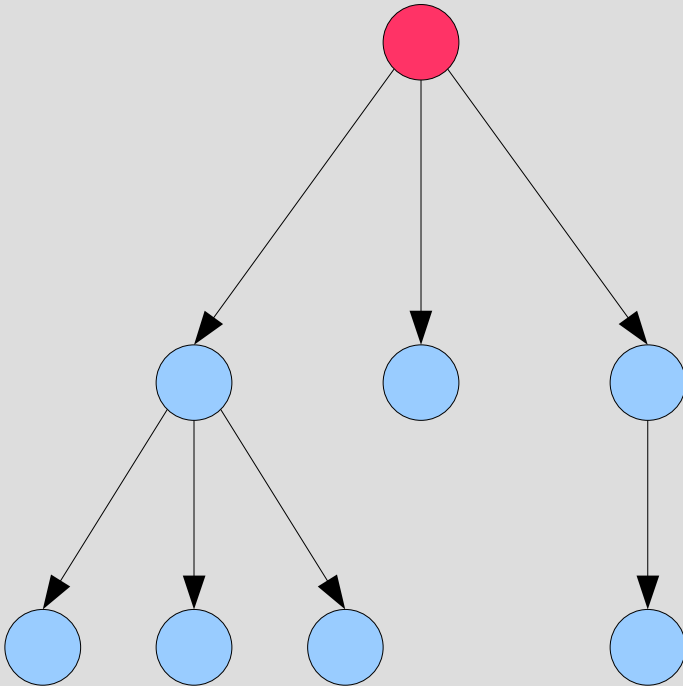
```
; Structure générale d'un parcours  
; en profondeur d'un arbre via  
; une fonction de parcours en  
; profondeur de foret  
; arbre_profondeur : arbre -> ...  
(define (arbre_profondeur A)  
  (cond ((empty? A) ...)  
        (else (foret_profondeur (list (racine A)))))  
  )  
)
```

# Parcours en profondeur en Scheme : code (2/2)

```
;; fonction de parcourt en profondeur  
;; d'une foret (liste de noeuds racines)  
;; foret_profondeur : liste -> ...  
(define (foret_profondeur LA)  
  (cond ((empty? LA) ...)  
        ( ;; traitement du noeud courant (car LA)  
          ;; attention à l'appelle recursif  
          ;; (foret_profondeur (cdr LA))  
          (else (foret_profondeur (append  
                                   (fils (car LA)) (cdr LA))))))  
  )  
)
```

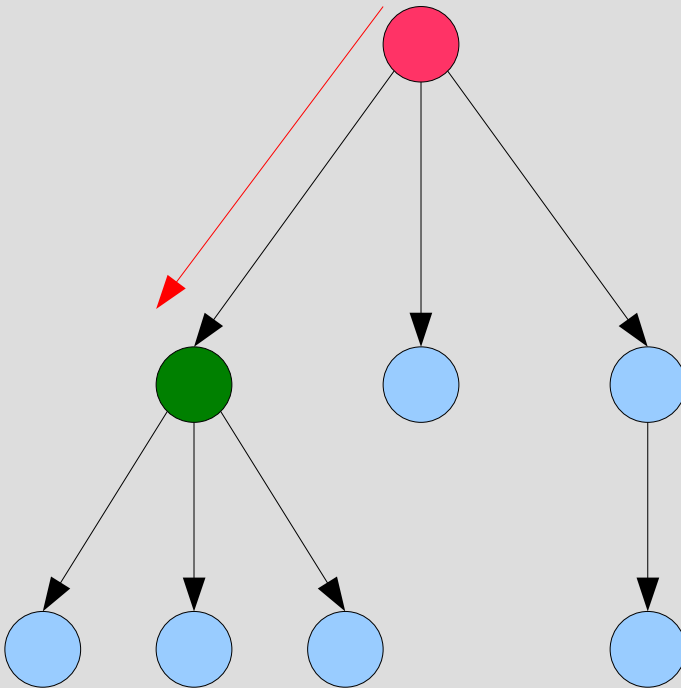
# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



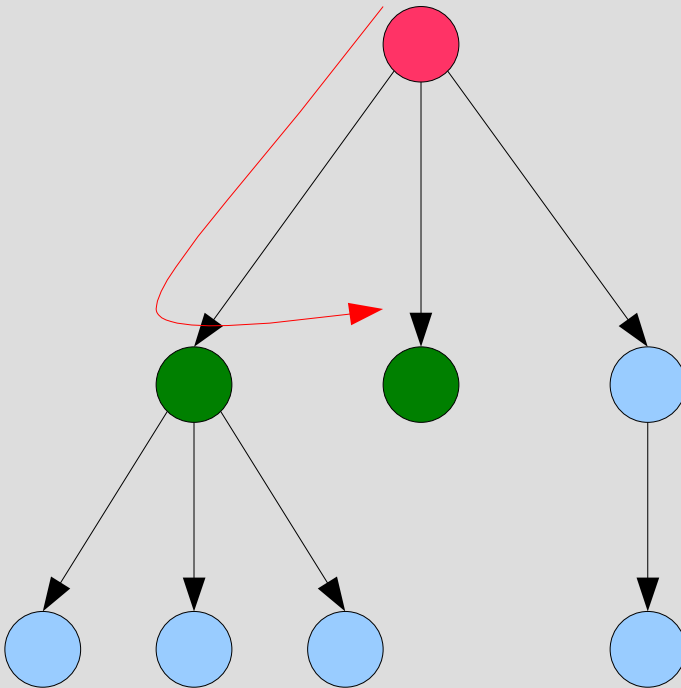
# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



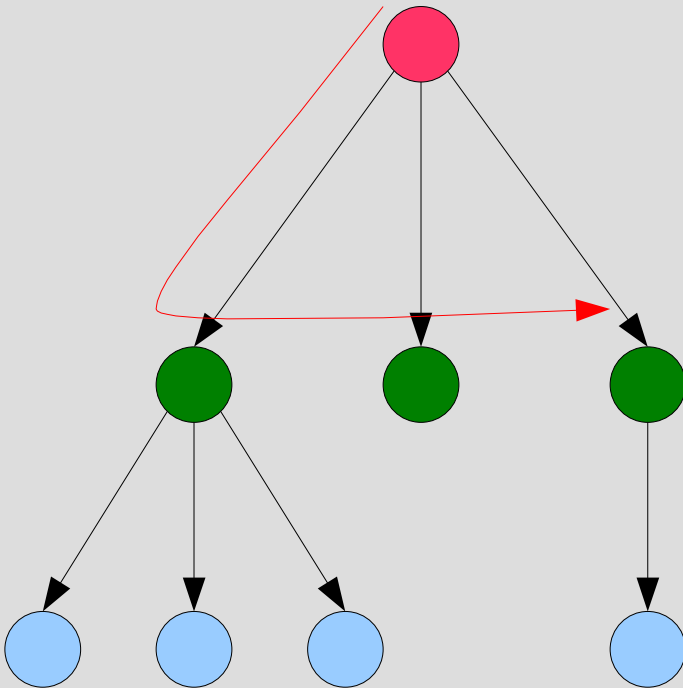
# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



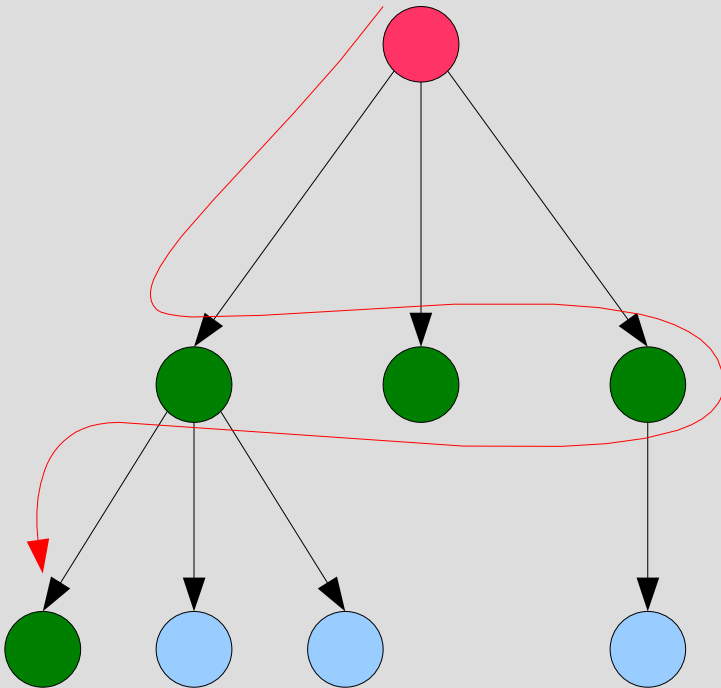
# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



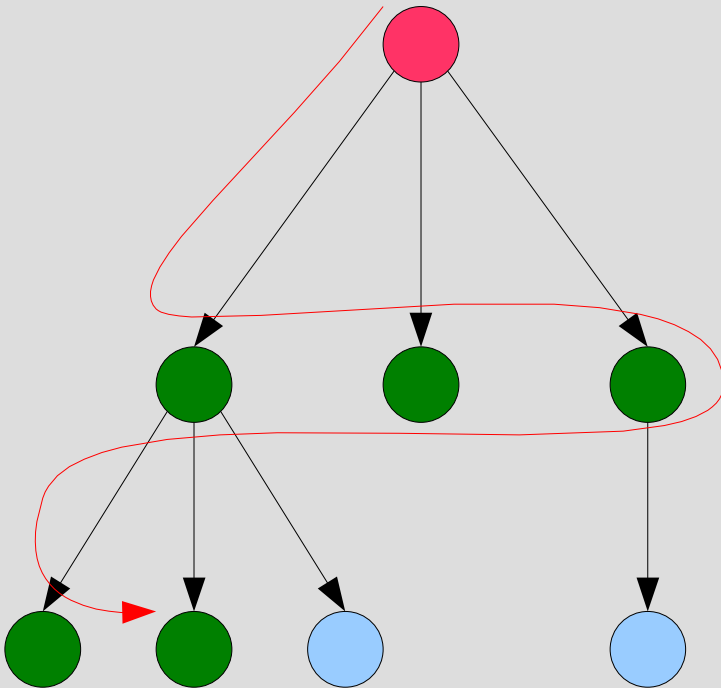
# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



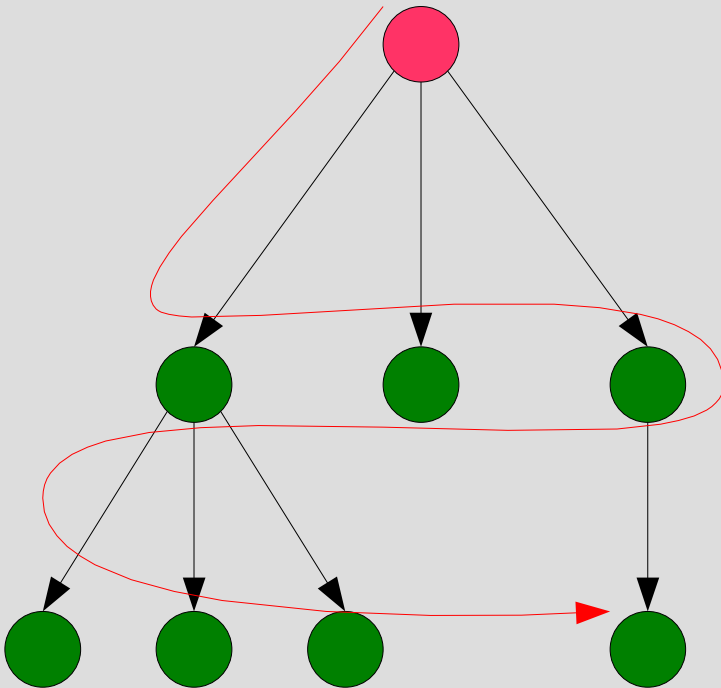
# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



# Parcours en largeur

Ce parcours consiste à parcourir tous les fils d'un nœud avant de parcourir ses petits fils.



# Parcours en largeur

Ce parcours est moins intuitif à mettre en place que celui en profondeur.

Il nécessite une liste de type FIFO pour stocker les futurs nœuds à traiter.

Quand un nœud est traité, il ajoute ses fils à la FIFO.

Passer au nœud suivant consiste à traiter le premier nœud de la FIFO.

# Parcours en largeur en Scheme : code (1/2)

```
; Structure générale d'un parcours  
; en largeur d'un arbre via  
; une fonction de parcours en  
; largeur de forêt  
; arbre_profondeur : arbre -> ...  
(define (arbre_largeur A)  
  (cond ((empty? A) ...)  
        (else (foret_largeur (list (racine A)))))  
  )  
)
```

# Parcours en largeur en Scheme : code (2/2)

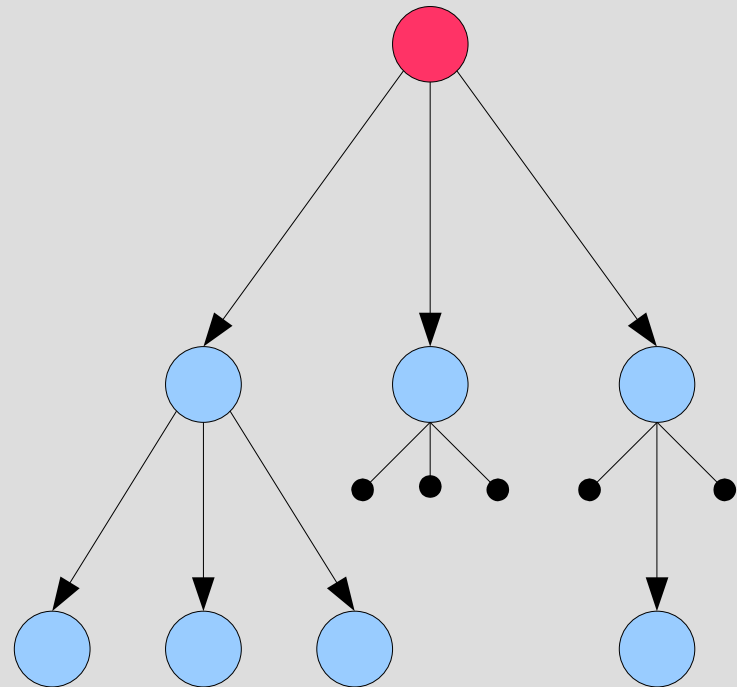
```
;; fonction de parcourt en largeur  
;; d'une foret (liste de noeuds racines)  
;; foret_profondeur : liste -> ...  
(define (foret_largeur LA)  
  (cond ((empty? LA) ...)  
        ( ;; traitement du noeud courant (car LA)  
          ;; attention à l'appelle recursif  
          ;; (foret_largeur (cdr LA))  
          (else (foret_largeur (append  
                                (cdr LA) (fils (car LA))))))  
        )  
  )  
)
```

# Arbres n-aires

Un arbre n-aire est un arbre, tel que chaque nœud ait  $n$  fils.

Une feuille d'un tel arbre est un nœud dont les  $n$  fils sont nulles (ou vides).

Les arbres binaires sont les plus utilisés.



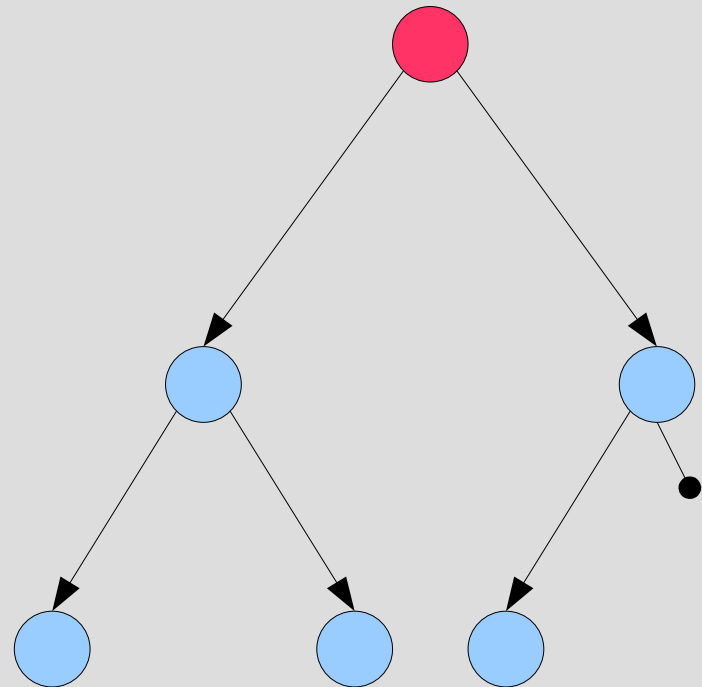
# Arbres binaires

Un arbre binaire est un arbre, tel que chaque nœud ait deux fils :

fils gauche

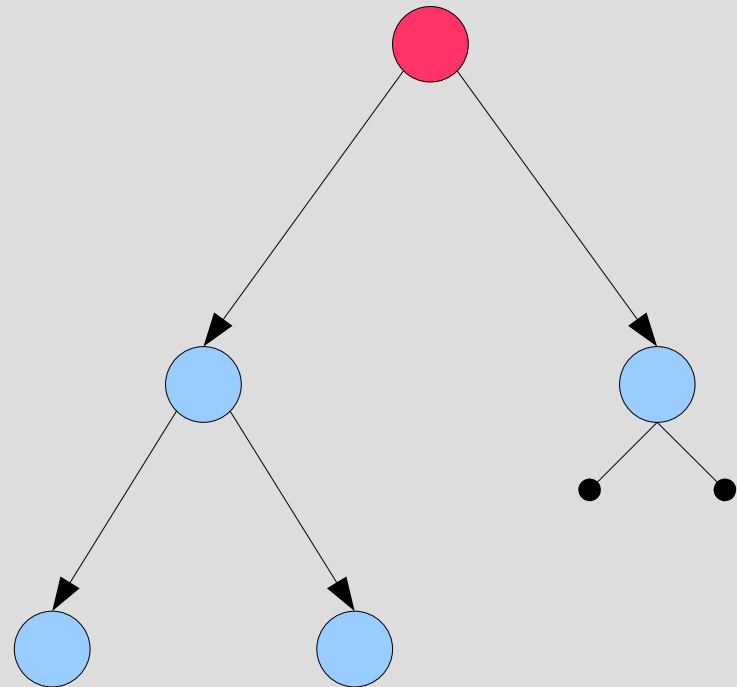
fils droit

Attention, un abus de langage est fait entre arbre et arborescence binaire.



# Arbres binaires entiers

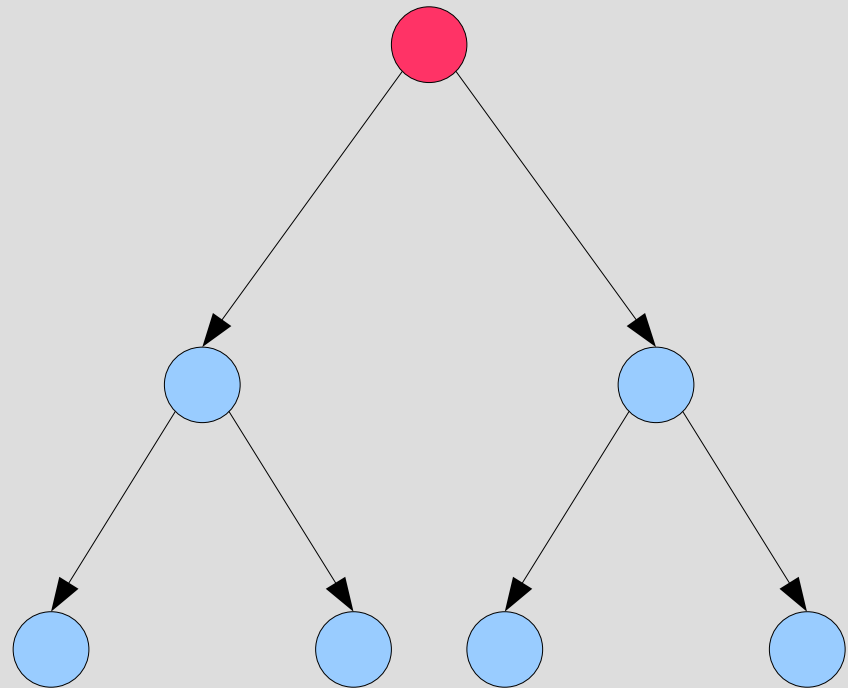
arbre binaire dont tous les nœuds possèdent zéro ou deux fils



# Arbres binaires parfaits

Arbre binaire entier dans lequel toutes les feuilles sont à la même distance de la racine.

Il sera dit quasi parfait s'il manque des nœuds à la droite du dernier étage.



# TD arbres binaires

Concevez les structures nécessaires à la modélisation d'un arbre binaire.

Proposez des primitives permettant de manipuler votre arbre binaire indépendamment de la structure choisie.

Écrivez les fonctions suivantes :

existe? : arbre x object -> boolean

count : arbre x object -> integer

profondeur : arbre -> integer



# TD arbres binaires : primitive

racine: arbre → noeud

leaf?: noeud → boolean

sous-arbre-gauche: noeud → arbre

sous-arbre-droit: noeud → arbre

etiquette: noeud → object

empty?: arbre → boolean

fil: noeud → list

# TD arbres binaires : existe?

## version arbres binaires

```
;; existe? retourne vraie si et seulement si l'objet  
;; passé en paramètre est une étiquette de l'arbre  
;; binaire passé en paramètre.  
;; existe? : objet x arbre -> boolean  
(define (existe? obj a)  
  (cond ((empty? a) #f)  
        ((equal? obj (etiquette (racine a))) #t)  
        (else (or (existe? obj (sous-arbre-gauche a))  
                   (existe? obj (sous-arbre-droit a)))))  
  )  
)
```

# TD arbres binaires : existe? version générique (1/2)

;; existe? retourne vraie si et seulement si l'objet  
;; passé en paramètre est une étiquette de l'arbre  
;; passé en paramètre.

;; existe? : objet x arbre -> boolean

```
(define (existe? obj a)
  (cond ((empty? a) #f)
        (else (foret_existe? obj (list (racine a))))
  )
)
```

# TD arbres binaires : existe? version générique (2/2)

```
;; sous-fonction de existe? qui recherche l'existence  
;; d'un objet dans une foret (liste de noeuds  
;; racines)  
;; foret_existe? : objet x liste -> boolean  
(define (foret_existe? obj LA)  
  (cond ((empty? LA) #f)  
        ((equals? obj (etiquette (car LA))) #t)  
        (else (foret_existe? obj (append  
          (fils (car LA)) (cdr LA))))))  
  )  
)
```

# TD arbres binaires : count

## version arbres binaires

;; count retourne le nombre d'occurrences de l'objet  
;; passé en paramètre apparaissant comme  
;; étiquette de l'arbre binaire passé en paramètre.  
;; count : objet x arbre -> entier

```
(define (count obj a)
  (cond ((empty? a) 0)
        ((equal? obj (etiquette (racine a)) (+ 1
                                         (count obj (sous-arbre-gauche a))
                                         (count obj (sous-arbre-droit a))))
        (else (+ (count obj (sous-arbre-gauche a))
                  (count obj (sous-arbre-droit a))))
  )
)
```

# TD arbres binaires : count version générique (1/2)

;; count retourne le nombre d'occurrences de l'objet  
;; passé en paramètre apparaissant comme  
;; étiquette de l'arbre passé en paramètre.  
;; count : objet x arbre -> entier

```
(define (count obj a)
  (cond ((empty? a) 0)
        (else (foret_count obj (list (racine a))))
  )
)
```

# TD arbres Binaires : count version générique (2/2)

```
;; sous-fonction de count qui compte le nombre  
;; d'occurrence d'un objet dans une foret (liste  
;; de noeuds racines)  
;; foret_count : objet x liste -> entier
```

```
(define (foret_count obj LA)  
  (cond ((empty? LA) 0)  
        ((equals? obj (etiquette (car LA))) (+ 1  
                                              (foret_count obj (append (fils (car LA))  
                                                                    (cdr LA)))))  
        (else (foret_count obj (append  
                                  (fils (car LA)) (cdr LA)))))  
  )  
)
```

# TD arbres binaires : profondeur

## version arbres binaires

```
;; profondeur retourne la profondeur de l'arbre  
;; binaire passé en paramètre.  
;; profondeur : arbre -> entier  
(define (profondeur a)  
  (cond ((empty? a) 0)  
        ((leaf? (racine a)) 1)  
        (else (max  
                (+ 1 (profondeur (sous-arbre-gauche a)))  
                (+ 1 (profondeur (sous-arbre-droit a))))))  
  )  
)
```

# TD arbres binaires : profondeur version générique (1/2)

```
;; profondeur retourne la profondeur de l'arbre  
;; passé en paramètre.  
;; profondeur : arbre -> entier  
(define (profondeur a)  
  (cond ((empty? a) 0)  
        (else (foret_profondeur (list (racine a)))))  
  )  
)
```

# TD arbres binaires : profondeur version générique (2/2)

```
;; sous-fonction de profondeur qui compte le  
;; nombre d'occurrence d'un objet dans une foret  
;; (liste de noeuds racines)  
;; foret_profondeur : liste -> entier  
(define (foret_profondeur LA)  
  (cond ((empty? LA) 0)  
        ((leaf? (car LA)) (max 1  
                               (foret_profondeur (cdr LA))))  
        (else (max  
              (+ 1 (foret_profondeur (fils (car LA))))  
              (foret_profondeur (cdr LA))  
              )  
        )  
)
```

# Construction d'arbre

Ajouter un fils à un nœud n'est pas un problème.

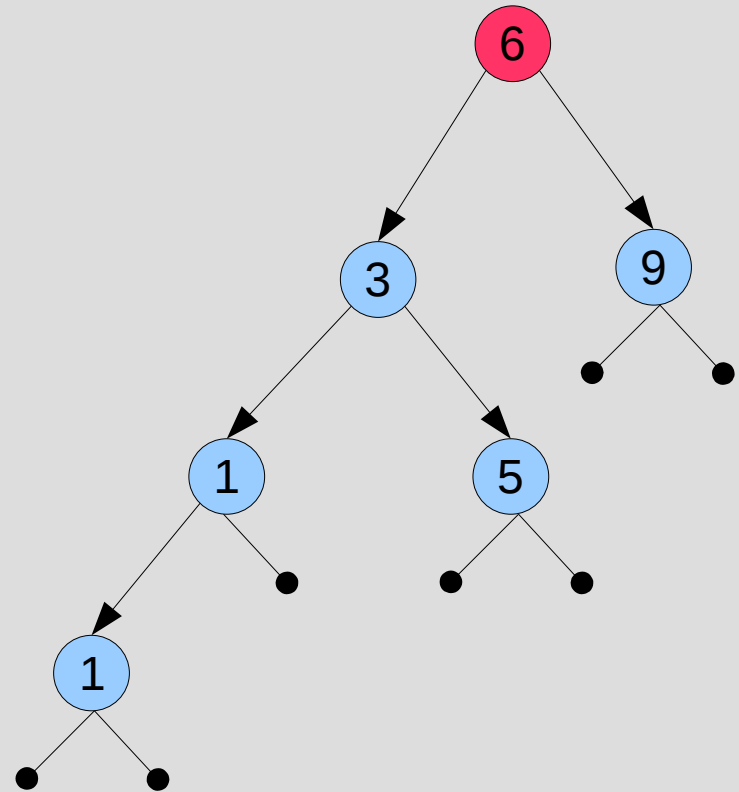
Ajouter un nœud (ou une feuille) à un arbre est nettement plus délicat.

Où – à quel nœud – doit-il être attaché ?

La réponse dépend de l'utilité de l'arbre et n'est jamais faite au hasard.

# Arbres binaires de recherche

Un arbre de recherche est un arbre binaire construit de telle sorte qu'il y ait une relation d'ordre entre un nœud et son sous-arbre gauche et entre ce nœud et son sous-arbre droit.



# Arbres binaires de recherche

Utilisation d'un arbre binaire de recherche :

Je recherche l'élément 'a' dans l'arbre,

S'il est sur la racine... bingo !

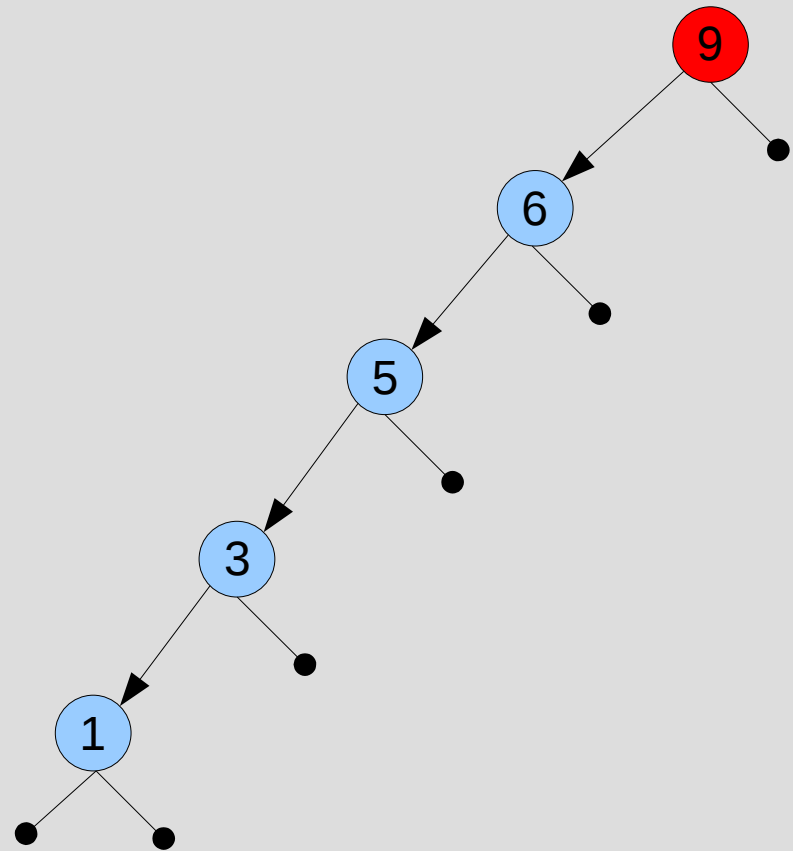
S'il est plus petit que la racine – relativement la relation d'ordre utilisée –, la recherche est lancée dans le sous-arbre gauche,

S'il est plus grand que la racine – relativement la relation d'ordre utilisé –, la recherche est lancée dans le sous-arbre droit.

# Arbres de recherche

Un problème peut apparaître si l'arbre est dégénéré.

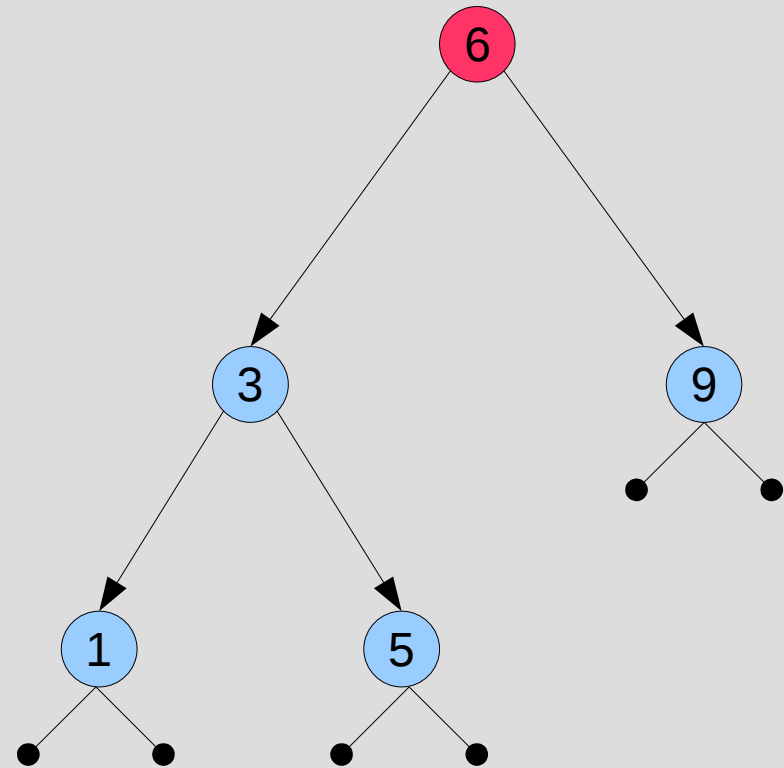
Il n'y a plus d'avantage par rapport à une simple liste dans le pire des cas



# Arbres équilibrés

Un arbre équilibré en hauteur est un arbre pour lequel chaque nœud est constitué de deux sous-arbres dont la hauteur ne diffère pas plus de 1.

Il permet d'optimiser les recherches.



# TD : arbres binaires de recherche

Comment construire un arbre binaire de recherche à partir d'une liste d'éléments ?

(1 5 9 7 6 4 8 3 2)

(vache mouton chien chat poule oie canard)

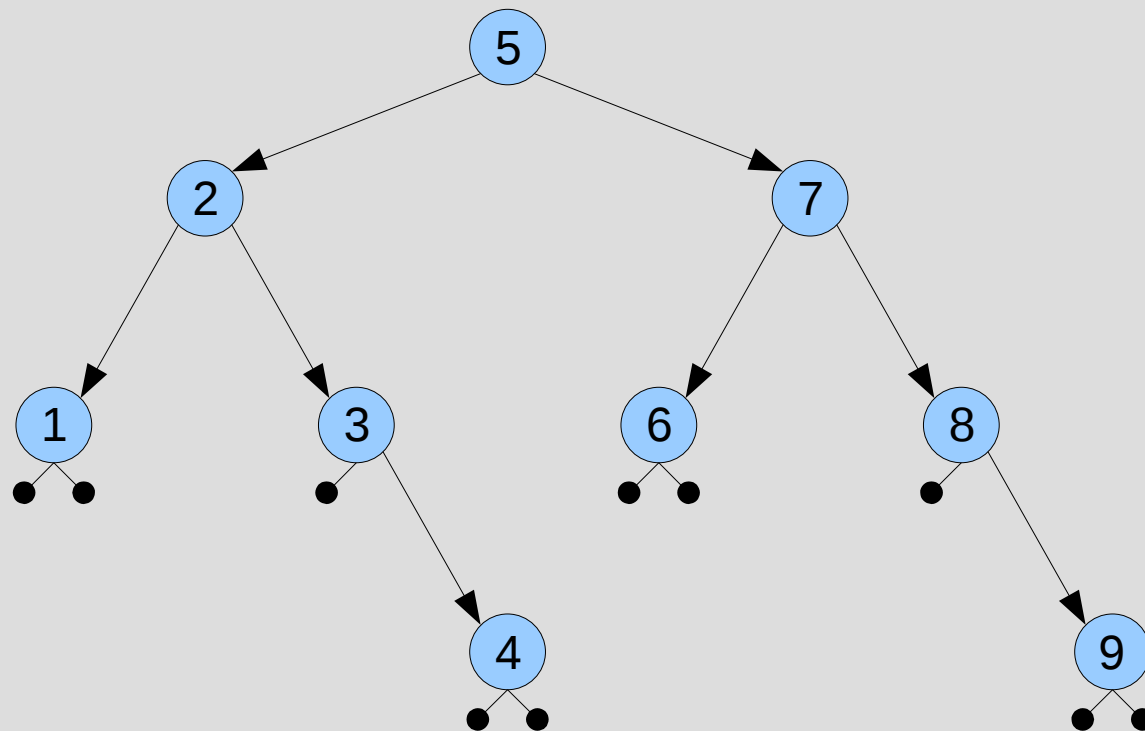
Chercher l'élément médian par rapport à la relation d'ordre, il deviendra racine.

Créer deux ensembles de telle sorte que :

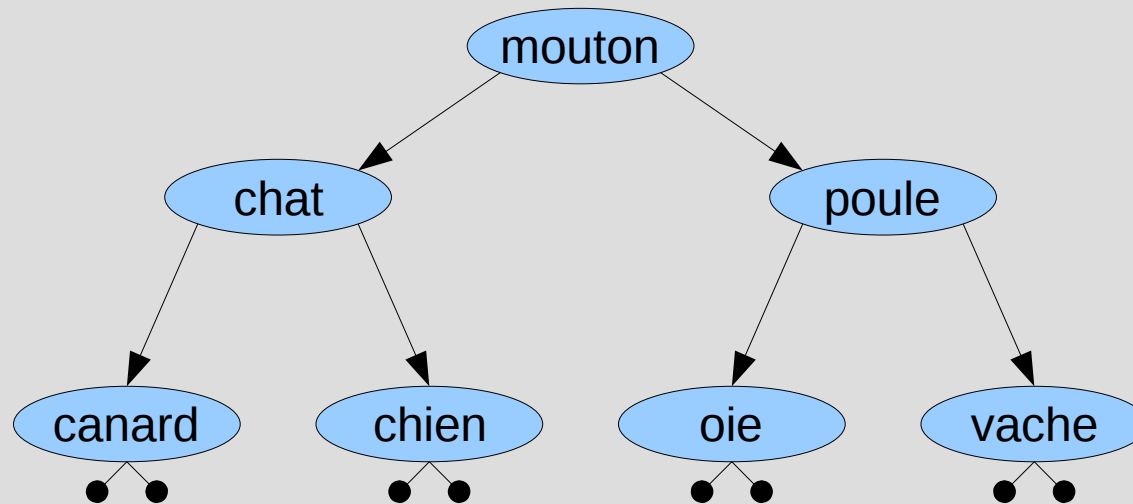
Les éléments du premier soient inférieurs au médian

Les éléments du premier soient supérieurs au médian par rapport à la relation d'ordre.

# TD : arbres binaires de recherche (1 5 9 7 6 4 8 3 2)



# TD : a. b. r. (vache mouton chien chat poule oie canard)



# Bibliographie

Élément de programmation en Scheme, *P. Gribomont*, Dunod.

Recueil de petits problèmes en Scheme, *L. Moreau, C. Queinnec, D. Ribbens M. Serrano*, Springer.

La Programmation une approche fonctionnelle et récursive avec Scheme, *L. Ardit, S. Ducasse*, Eyrolles