



Université Paris XI
I.U.T. d'Orsay
Département Informatique
Année scolaire 2003-2004

Algorithmique : Volume 5

- Récursion
- Listes récursives
- Arbres binaires

Exemple 1: Calcul de la factorielle

- $\text{fact}(3) = 3 \times 2 \times 1$
- $\text{fact}(N) = N \times \text{fact}(N - 1)$ la fonction fait appel à elle-même
vrai pour $N > 0$
- $\text{fact}(0) = 1$ condition d'arrêt

```
1.  Fonction fact(N) retourne (entier)
    paramètre      (D) N : entier
    variable       résultat : entier
    début
2.          si (N = 0)
3.              alors  retourne (1)
4.              sinon  résultat ← ( N × fact(N-1) )
5.              retourne (résultat)
          fsi
    fin
```

Simulation

Principes d'écriture pour la simulation

1. Numéroter les instructions

- 1 : en-tête,
- 2 : instruction qui suit *début*,
- ...
- k : instruction qui précède *fin*

2. Indexer, par le niveau d'appel, les noms des paramètres et variables

3. Donner la trace de l'exécution (avec flèches indiquant les appels et les retours)

Exemple : fact(3)

1^{er} appel :

- 1. $N1 = 3$
- 2. $(N1 = 0) ?$ faux
- 4. $r1 \leftarrow 3 * \text{fact}(2)$

Exemple 2: le PGCD de deux entiers

- $\text{pgcd}(A,B) = \text{pgcd}(B, A \bmod B)$ appel récursif,
vrai pour $B > 0$
- $\text{pgcd}(A,0) = A$ condition d'arrêt

```
1.  Fonction pgcd(A,B) retourne (entier)
    paramètres      (D) A,B : entiers
    variable        résultat : entier
    début
2.      si B = 0
3.      alors        retourner (A)
4.      sinon        résultat ← pgcd(B, A mod B)
5.      retourner (résultat)
    fsi
  fin
```

test d'arrêt de la récursion → (pointing to "si B = 0")

appel récursif → (pointing to "**pgcd(B, A mod B)**")

Exemple 2: simulation

réponse \leftarrow pgcd(21, 12)

1^{er} appel :

1. $A_1 = 21, B_1 = 12$

Exemple 3: recherche d'un élément dans un tableau

Version itérative

Fonction recherche(val, tab, nbr) retourne (booléen)

paramètres (D) val, nbr : **entiers**

(D) tab : **tableau** [1, MAX] d'entiers

variables trouvé : **booléen** ; cpt : **entier**

début

cpt \leftarrow 0 ; trouvé \leftarrow FAUX

tant que (non trouvé et cpt < nbr) **faire**

 cpt \leftarrow cpt +1

 trouvé \leftarrow (tab[cpt] = val)

ftq

retourne(trouvé)

fin

Idée d'une solution récursive

- **recherche(val, tab, nbVal)**
 - est-ce que val est la dernière valeur du tableau?
 - si oui, fin (retourne Vrai), sinon, reprendre dans le tableau sans la dernière valeur :
recherche(val, tab, nbVal - 1)
→ appel récursif, possible pour $\text{nbVal} > 0$
- **nbVal = 0** condition d'arrêt

Fonction recherche(val, tab, nbVal) retourne (booléen)

paramètres (D) val : entier

(D) tab : tableau [1, MAX] d'entiers

(D) nbVal : entier

début

si (nbVal = 0)

alors retourne (FAUX)

sinon si (tab[nbVal] = val)

alors retourne (VRAI)

sinon retourne (recherche(val, tab, nbVal - 1))

fsi

fsi

fin

test d'arrêt
de la récursion

appel
récursif

Exemple 3: simulation

unTab

1	5	8	2
---	---	---	---

recherche(5, unTab, 4)

1^{er} appel :

1. $val_1=5$, $tab_1=unTab$, $nbr_1=4$

version réursive alternative

→ Comparer les enchaînements réursifs (très inefficace)

Fonction recherche(val, tab, nbr) retourne (booléen)

paramètres (D) val : **entier**
 (D) tab : **tableau** [1, MAX] **d'entiers**
 (D) nbr : **entier**

début

si (nbr = 0)
 alors retourne (FAUX)
 sinon **si** (**recherche(val, tab, nbr - 1)**)
 alors retourne (VRAI)
 sinon retourne (tab[nbr] = val)
 fsi

fsi

fin

écriture récursive alternative

→ intéressante en C++ ("évaluation paresseuse"),
mais très inefficace en algo

Fonction recherche(val, tab, nbr) retourne (booléen)

paramètres (D) val : **entier**

(D) tab : **tableau** [1, MAX] **d'entiers**

(D) nbr : **entier**

début

si (nbr = 0)

alors retourne (FAUX)

sinon retourne (tab[nbr] = val OU **recherche(val, tab, nbr - 1)**)

fsi

fin

Exemple 4: inversion d'une chaîne de caractères

Exemple : abcde

1. Enlever le premier caractère
a
2. Inverser le reste de la chaîne
edcb
→ **appel récursif**
3. Rajouter le premier en fin de chaîne
edcba

Condition d'arrêt ?

Inverser une chaîne de longueur 1 = ne rien faire

Exemple 4: simulation

1^{er} appel : **inverser "algo"**

1. enlever "a"
2. inverser "lgo"
3. rajouter "a"

Procédure d'inversion

Procédure inverser (mot, nbcar)

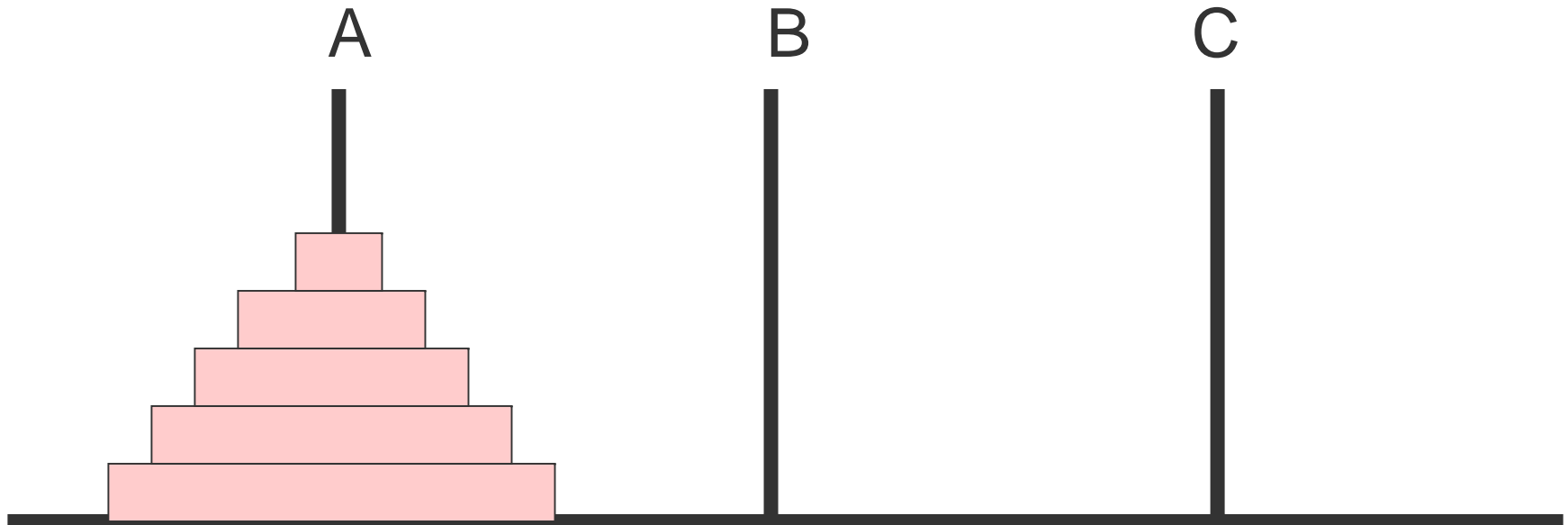
paramètres (D/R) mot : tableau [1, MAX] de caractères
 (D) nbcar : entier

Exemple 5: les tours de hanoi

Problème :

transférer les N disques de l'axe A à l'axe C, en utilisant B, de sorte que jamais un disque ne repose sur un disque de plus petit diamètre.

Ici : $N = 5$



Idée d'une solution récursive

- supposons que l'on sache résoudre le problème pour $(N-1)$ disques :
 1. on déplace $(N-1)$ disques de A vers B en utilisant C
 2. on fait passer le N-ième grand disque en C *{déplacer le plus grand}*
 3. on déplace les $(N-1)$ disques de B vers C en utilisant A
- c'est l'idée générale de la récursion :

par exemple, si je sais calculer $\text{fact}(N-1)$, je sais calculer $\text{fact}(N)$

{ « réduction » du problème, jusqu'à la condition d'arrêt}
- pour les tours de hanoi, le problème de taille N donne naissance à deux problèmes de taille $(N-1)$

Squelette d'une procédure récursive

1. **Procédure hanoi (source, but, aux, nb)**
paramètres
début
2. **si** nb = 1
 alors déplacer un disque de source vers but
3. **sinon** **hanoi (source, aux, but, nb – 1)**
4. déplacer un disque de source vers but
5. **hanoi (aux, but, source, nb – 1)**
- fsi**
- fin**

Simulation de hanoi(A,C,B,3)

[il manque les flèches d'appel et de retour!]

1^{er} appel : hanoi(A,C,B,3)

1. Faux
3. hanoi(A,B,C,2)
4. déplacer 1 disque de A à C
5. hanoi(B,C,A, 2)

5^{ème} appel : hanoi(B,C,A, 2)

1. Faux
3. hanoi(B,A,C,1)
4. déplacer 1 disque de B à C
5. hanoi(A,C,B,1)

2^{ème} appel : hanoi(A,B,C,2)

1. Faux
3. hanoi(A,C,B,1)
4. déplacer 1 disque de A à B
5. hanoi(C,B,A,1)

6^{ème} appel : hanoi(B,A,C,1)

1. vrai
2. déplacer 1 disque de B à A

7^{ème} appel : hanoi(A,C,B,1)

1. vrai
2. déplacer 1 disque de A à C

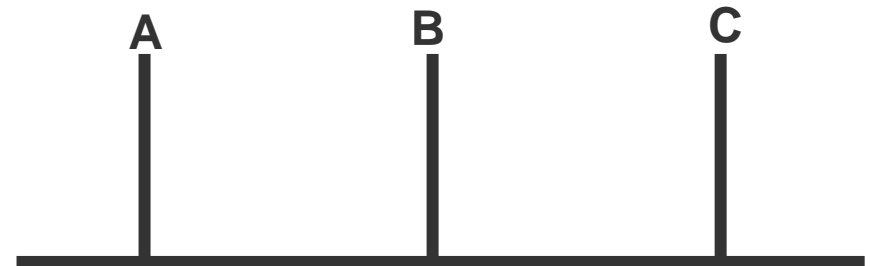
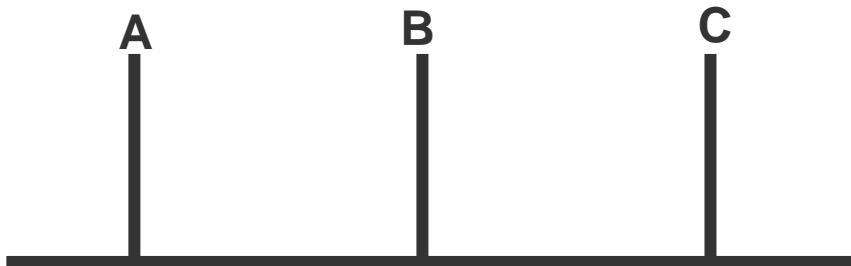
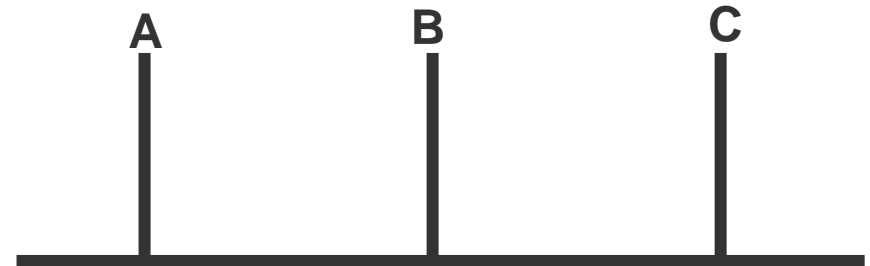
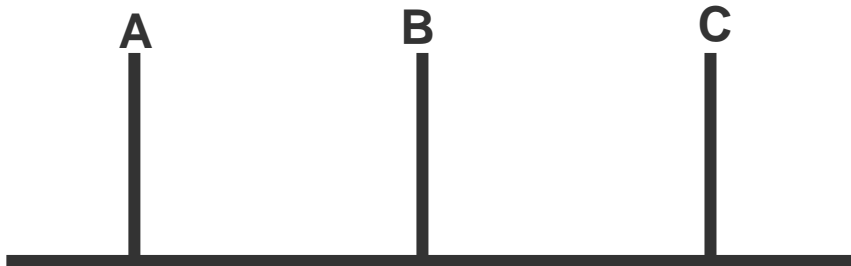
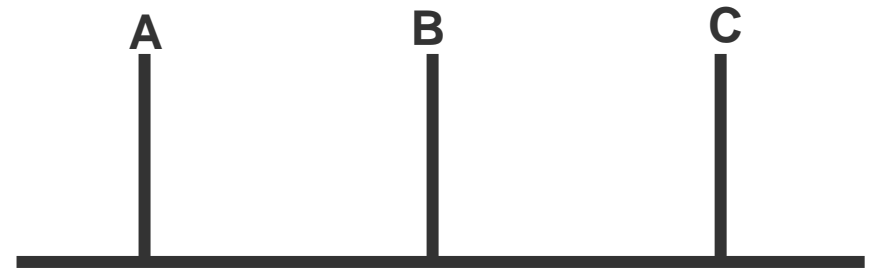
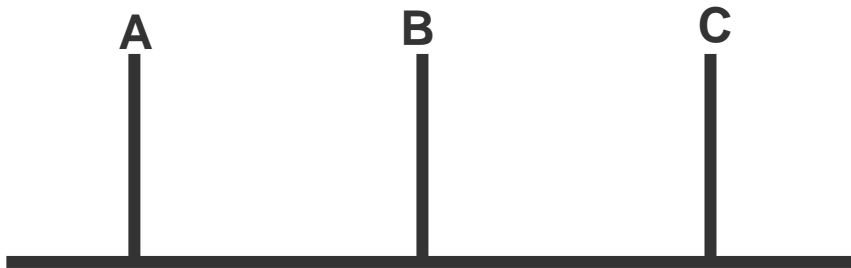
3^{ème} appel : hanoi(A,C,B,1)

1. vrai
2. déplacer 1 disque de A à C

4^{ème} appel : hanoi(C,B,A,1)

1. vrai
2. déplacer 1 disque de C à B

Simulation de hanoi(A,C,B,3)



Récapitulation

- Fonction récursive ?
 - Fonction qui s'appelle elle-même.
- Comment éviter les appels infinis ?
 - Trouver le test qui va déterminer l'arrêt.
- Comment les écrire?
 - Définir le cas général : il contiendra l'appel récursif.
 - Définir le cas particulier : c'est le test d'arrêt.

Un autre exemple de simulation

Procédure manipTableau(tab, val1, val2)

paramètres (D/R) tab : tableau[1:Max]

 (D) val1: entier

 (D/R) val2 : entier

début

si val1 \neq 0

alors manipTableau(tab, val1 – 1, val2)

si tab[val1] \neq BLANC

alors val2 \leftarrow val2 + 1

 tab[val2] \leftarrow tab[val1]

fsi

fsi

fin

Un dernier exemple

Soit un tableau de pixels à deux dimensions $N \times N$. On représente les pixels allumés à l'aide de la valeur 1 et les pixels éteints à l'aide de la valeur 0. Ecrire une fonction récursive **surfaceRégion** qui prend en paramètres (entre autres) les coordonnées **x** et **y** d'un pixel et qui renvoie la **surface** de la **région** à laquelle appartient ce pixel. La **surface** est évaluée ici en terme de nombre de pixels allumés. Une **région** est un ensemble de pixels allumés liés par continuité horizontale, verticale ou diagonale.

Exemple, ce tableau ci-contient 3 régions et surfaceRégion doit retourner

- 5 pour le point (3,4),
- 2 pour le point (1,2)
- 0 pour le point (5,5)
- 4 pour le point (5,1)

		x				
		1	2	3	4	5
y	1	1	0	1	1	1
	2	1	0	0	1	0
	3	0	0	0	0	0
	4	0	1	1	0	0
	5	1	0	1	1	0

Listes récursives



Nouvelle définition d'une liste

- Une liste, c'est
 - Soit une liste vide
 - Soit une cellule suivie d'une liste
- Définition récursive car une liste est définie par une liste.
- La règle "soit une liste vide" assure l'arrêt et donc la cohérence de la définition

Définition de la classe

Liste Récursive

Classe ListeRéc

Attributs :

non explicités

{vous êtes utilisateurs de la classe}

Méthodes :

MFonction **vide()**

{vrai si la liste est vide}

MFonction **infoTête()**

{valeur enregistrée dans la tête}

MFonction **reste()**

{liste obtenue en enlevant la tête}

MFonction **préfixer(val)**

{liste obtenue en insérant val en tête}

Méthodes

MFonction **vide()** retourne booléen

{Retourne vrai si la liste est vide, faux sinon}

paramètre (D) cible : ListeRéc

MFonction **infoTête()** retourne Info

{Retourne la valeur enregistrée dans la tête. Erreur si la liste est vide.}

paramètre (D) cible : ListeRéc

MFonction **reste()** retourne ListeRéc

{Retourne la liste obtenue en enlevant la tête. Erreur si la liste est vide.}

paramètre (D) cible : ListeRéc

MFonction **préfixer(val)** retourne ListeRéc

{Retourne la liste obtenue en insérant val en tête.}

paramètre (D) cible : ListeRéc

Affichage d'une liste récursive

Procédure **afficherListe(uneListe)**

{Affiche les valeurs contenues dans une liste.}

paramètre (D) uneListe : ListeRéc

début

si non uneListe.vide()

{test d'arrêt}

alors afficher(uneListe.infoTête())

{traitement}

afficherListe(uneListe.reste())

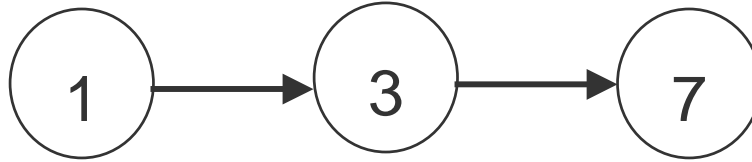
{appel récursif}

fsi

fin

Affichage: simulation

uneL



afficherListe(uneL)

1^{er} appel :

1. uneL₁

Deux types de récursion

Récursion "descendante"

- récursion terminale
("fausse" récursion)
- rien au retour
- traitement suivi de l'appel récursif
- données n'ont pas besoin
d'être stockées
- version itérative possible

Récursion "ascendante"

- récursion non terminal
- exécution au retour, en remontant
- appel récursif suivi du traitement
- données ont besoin d'être stockées
- version itérative beaucoup plus difficile

Affichage dans l'ordre inverse

Procédure **afficherInverse(uneListe)**

{Affiche les valeurs contenues dans une liste dans l'ordre inverse.}

paramètre (D) uneListe : ListeRéc

début

si non uneListe.vide()

{test d'arrêt}

alors **afficherInverse**(uneListe.reste())

{appel récursif}

afficher(uneListe.infoTête())

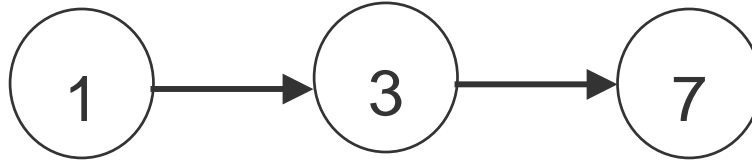
{traitement}

fsi

fin

Affichage inverse : simulation

uneL



afficherInv(uneL)

1^{er} appel :

1. uneL₁

Longueur d'une liste récursive

Fonction `longueur(uneListe)` retourne (entier)

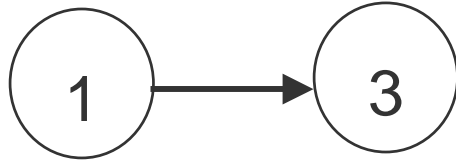
{retourne le nombre de cellules faisant partie dans une liste.}

paramètre (D) uneListe : ListeRéc

début

Longueur: simulation

uneL



longueur(uneL)

1^{er} appel :

1. uneL₁

Inversion d'une liste récursive

Retourner à l'exemple 4 , transparent 13

- **Cas particulier :**
 - Liste vide : ne rien faire (condition d'arrêt)
- **Cas général :**
 1. Enlever la tête
 2. Inverser le reste de la liste (appel récursif)
 3. Rajouter la tête en fin de liste

Fonction inverser(uneListe) retourne (ListeRéc)

{retourne la liste inverse de la liste passée en paramètre.}

paramètre (D) uneListe : ListeRéc

variables premier : Info
 nouvListe : LISTE-REC

début

si uneListe.vide()

alors retourner (uneListe)

sinon premier ← uneListe.infoTête()
 nouvListe ← **inverser(uneListe.reste())**
 ajoutQueue(premier, nouvListe)
 retourner(nouvListe)

fsi

fin

écriture alternative :

Fonction inverser(uneListe) retourne (ListeRéc)

{retourne la liste inverse de la liste passée en paramètre.}

paramètre (D) uneListe : ListeRéc

début

si uneListe.vide()

alors retourner (uneListe)

```
sinon    retourner (ajoutQueue (uneListe.infoTête(),  
                                inverser(uneListe.reste()) )
```

fsi

fin

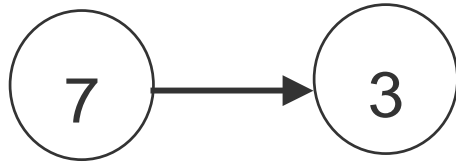
Fonction ajoutQueue(uneVal, uneListe) retourne (ListeRéc)

{retourne la liste résultant de l'ajout de uneVal en queue de uneListe.}

paramètre (D) uneVal : entier
 (D) uneListe : ListeRéc

ajoutQueue : simulation

uneL



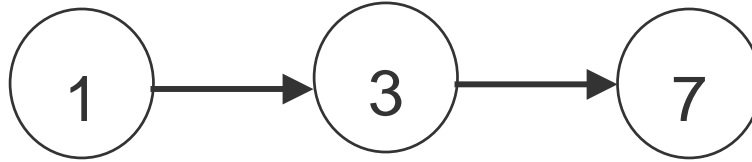
ajoutQueue(uneL,1)

1^{er} appel :

1. uneL₁

inverser : simulation

uneL



inverser(uneL)

1^{er} appel :

1. uneL₁

Arbres binaires

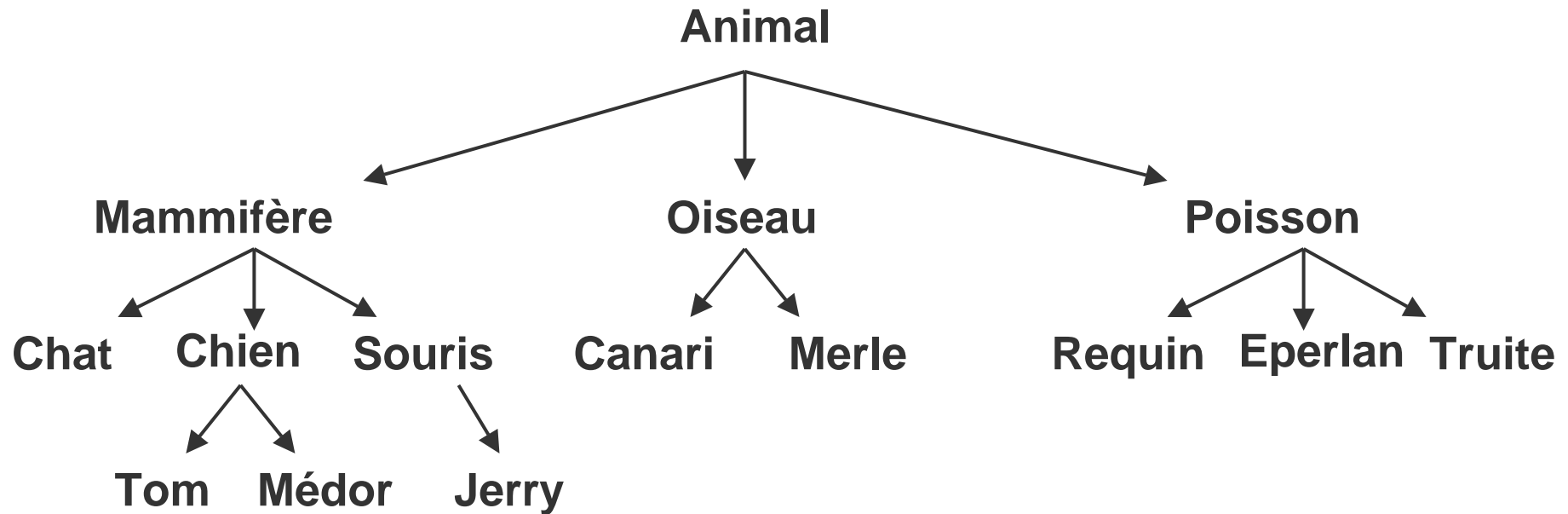


Représentations arborescentes

Les arborescences sont utilisées :

- Dans la vie de tous les jours : pour représenter des hiérarchies, des classifications, des partitions
- Dans le domaine de l'informatique : pour représenter les informations ci-dessus, et aussi :
 - L'organisation interne des fichiers en mémoire
 - Les mode de calcul d'une expression
 - L'organisation de données triées

Exemple :



Signification du lien :

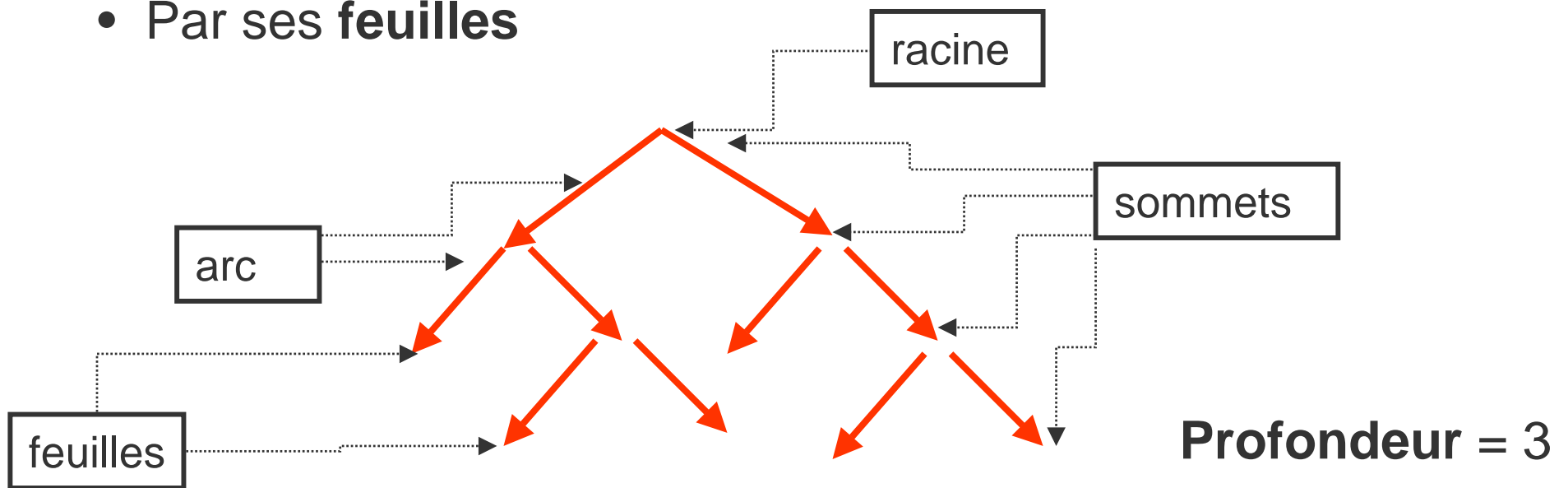
- du plus générique au plus spécifique

Autres significations possibles :

- du plus ancien au plus récent
- de la plus haute autorité à la moindre
- du plus complexe au plus simple

Comment caractériser un arbre?

- Par sa **racine**
- Par ses **sommets**
- Par les **arcs** reliant les sommets entre eux
- Par ses **feuilles**



Définition récursive d'un arbre binaire

- Un arbre binaire, c'est
 - Soit un arbre binaire vide
 - Soit une racine avec deux sous-arbres binaires (appelés fils gauche et fils droit)
- Définition récursive car un arbre binaire est défini par un arbre binaire.
- La règle "soit un arbre binaire vide" assure l'arrêt et donc la cohérence de la définition.

Définition de la classe ArbreBinaire

Classe ArbreBinaire

Attributs :

non explicités

{vous êtes utilisateurs de la classe}

Méthodes :

MFonction **vide()**

{vrai si l'arbre binaire est vide}

MFonction **info()**

{valeur enregistrée à la racine}

MFonction **filsG()**

{"fils gauche" : sous arbre gauche}

MFonction **filsD()**

{"fils droit" : sous arbre droit}

Méthodes

MFonction vide() retourne booléen

{Retourne vrai si l'arbre binaire est vide, faux sinon}

paramètre (D) cible : ArbreBinaire

MFonction info() retourne Info

{Retourne la valeur enregistrée dans la racine. Erreur si l'arbre est vide.}

paramètre (D) cible : ArbreBinaire

MFonction filsG() retourne ArbreBinaire

*{Retourne l'arbre binaire formé par le sous arbre gauche ("fils gauche").
Erreur si l'arbre binaire est vide.}*

paramètre (D) cible : ArbreBinaire

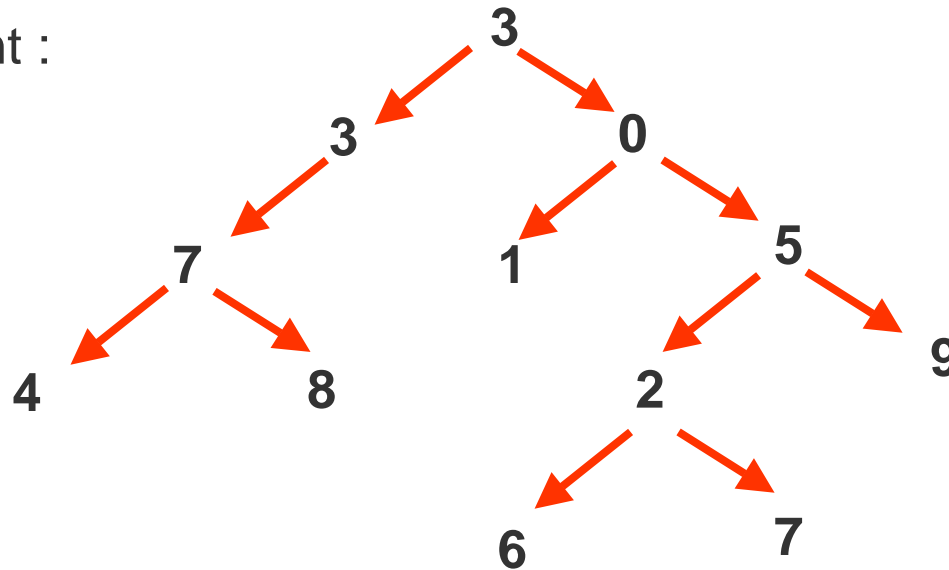
MFonction filsD() retourne ArbreBinaire

*{Retourne l'arbre binaire formé par le sous arbre droit ("fils droit").
Erreur si l'arbre binaire est vide.}*

paramètre (D) cible : ArbreBinaire

Affichages : ordres possibles

Soit l'arbre binaire suivant :



Affichage

- ordre préfixe : **3** 3 7 4 8 0 1 5 2 6 7 9 (racine d'abord)
- ordre suffixe : 4 8 7 3 1 6 7 2 9 5 0 **3** (racine en dernier)
- ordre infixe : 4 7 8 3 **3** 1 0 6 2 7 5 9 (racine au milieu)

Affichage : ordre préfixe

Procédure **affichePréfixe (unArbre)**

*{Affiche les valeurs portées par les sommets de l'arbre binaire, en affichant la valeur portée par la racine **avant** les valeurs portées par les sous-arbres gauche et droit}*

paramètre (D) unArbre : **ArbreBinaire** {supposons *TypInfo* = réel}

début

si non unArbre.vide()

alors afficher (unArbre.info())

affichePréfixe (unArbre.filsG())

affichePréfixe (unArbre.filsD())

fsi

fin

Affichage : simulation

Affichage : ordre suffixe

Procédure **afficheSuffixe (unArbre)**

*{Affiche les valeurs portées par les sommets de l'arbre binaire, en affichant la valeur portée par la racine **après** les valeurs portées par les sous-arbres gauche et droit}*

paramètre (D) unArbre : **ArbreBinaire** *{supposons TypInfo = réel}*

début

si non unArbre.vide()

b



afficheSuffixe (unArbre.filsD())

afficher (unArbre.info())

fsi

fin

Affichage : ordre Infixe

Procédure **afficheInfixe (unArbre)**

*{Affiche les valeurs portées par les sommets de l'arbre binaire, en affichant la valeur portée par la racine **entre** les valeurs portées par les sous-arbres gauche et droit}*

paramètre (D) unArbre : **ArbreBinaire** {supposons *TypInfo* = réel}

début

si non unArbre.vide()

alors **afficheInfixe (unArbre.filsG())**

 afficher (unArbre.info())

afficheInfixe (unArbre.filsD())

fsi

fin

Nombre de sommets d'un arbre binaire

Cas particulier : arbre vide : résultat = 0

Cas général : 1 (sommet de l'arbre courant)

+ nb sommets dans FG

+ nb sommets dans FD

Fonction **compteSommets(unArbre)** retourne entier

{retourne le nombre de sommets de l'arbre binaire}

paramètre (D) unArbre : **ArbreBinaire** *{supposons TypInfo = réel}*

début

 si unArbre.vide()

alors retourne (0)

sinon retourne (1 + **compteSommets (unArbre.filsG())**
 + **compteSommets (unArbre.filsD())**)

fsi

fin

Application : arbres binaires de recherche (ABR)

Un arbre binaire de recherche est un arbre binaire dans lequel la valeur de chaque sommet est

- supérieure [ou égale] à toutes les valeurs étiquetant les sommets du sous-arbre gauche de ce sommet,
- et inférieure à toutes les valeurs étiquetant les sommets du sous-arbre droit de ce sommet.

Algorithme de construction d'un ABR

Soit val la valeur à placer dans l'ABR

(l'ajout se fera toujours sur une «feuille» : arbre binaire dont le FG et le FD sont vides)

- Si l'arbre est vide, en créer un, réduit à sa racine, étiquetée avec val
- Sinon si $val \leq$ valeur portée par la racine,
 - alors l'ajouter au sous-arbre gauche : si cet arbre n'est pas vide, reprendre l'algorithme sur ce sous-arbre
 - Sinon l'ajouter au sous-arbre droit : si cet arbre n'est pas vide, reprendre l'algorithme sur ce sous-arbre

Construction d'un ABR : simulation

12 45 -34 0 23 18 19 56 -12 18 45

Ajout d'une valeur dans un ABR

MProcédure **ajout(uneVal)**

{ajoute une valeur à un arbre binaire de recherche}

paramètres (D) uneVal : entier
 (D/R) cible : ABR

début

si cible.vide()

alors cible.création(val)

sinon si uneVal \leq cible.info()

alors (cible.filsG()).ajout(uneVal)

sinon (cible.filsD()).ajout(uneVal)

fsi

fsi

fin

MProcédure **création(uneVal)**

{crée un arbre binaire réduit à sa racine et y affecte unVal}

paramètre (R) cible : ArbreBinaire
 (D) uneVal : entier

Ajout : simulation

Utilisation d'un ABR : trier une liste

construire
Arbre binaire

afficher arbre
Ordre infixe

Liste → arbre binaire de recherche → affichage liste triée

début

```
    uneListe.saisirListe()  
    construireABR(uneListe, unABR)  
    afficheInfixe(unABR)
```

fin

Construire un ABR à partir d'une liste

Procédure construireABR(uneListe, unABR)

{construit un arbre binaire de recherche à partir d'une liste}

paramètre (D) uneListe : LISTE-Rec

 (D/R) unABR : ArbreBinaire *{supposons TypInfo = entier}*

début

si non uneListe.vide()

alors

 val ← uneListe.infoTête()

 unABR.ajout(val)

construireABR(uneListe.reste(), unABR)

fsi

fin

Trier une liste : simulation

fin Volume 5