



Arbres

Structuration en arbre

Terminologie, définitions, notations

Représentation des arbres

Parcours d'arbres

Manipulation d'arbres

- Arbres n-aires
- Arbres binaires

Arbres n-aires

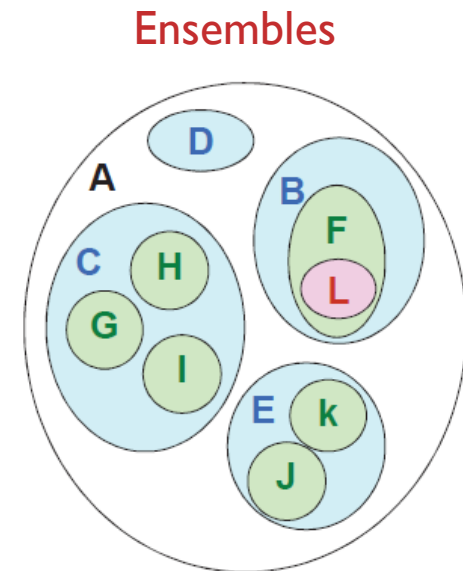
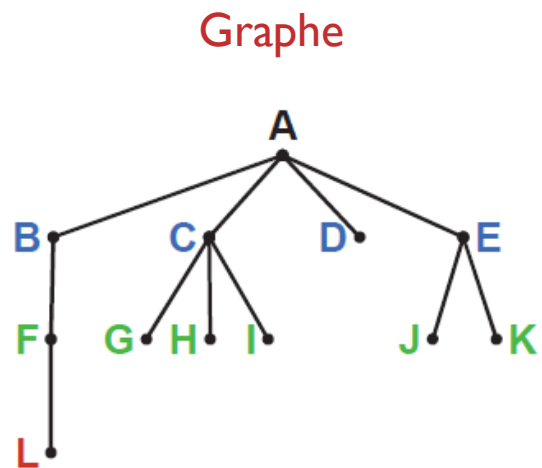
- Exemples d'arbres n-aires de la vie courante
 - Sommaire d'un livre :
 - Ensemble de titres (chapitres, sous-chapitres, paragraphes, etc.)
 - Arborescence de fichiers
 - Ensemble de noms (dossiers, fichiers)
 - Classification
 - Noms de classes, sous-classes
 - Expression arithmétique
 - Ensemble d'opérateurs et d'opérandes
 - Arbre de tâches (cf cours d'IHM)
 - Ensemble des tâches possibles par un logiciel interactif
 - Arbre de décision
 - Ensemble de questions

Définition d'un Arbre

- Un **Arbre** est un ensemble non vide structuré comme suit :
 - un des éléments est désigné comme étant la « **racine** » de l'arbre
 - il existe une partition sur les éléments restants, et chaque classe de cette partition est elle-même un arbre : on parle des **sous-arbres** de la racine.
- Si le nombre de sous-arbres est variable, l'arbre est dit **n-aire**.
- L'ensemble représenté par un arbre est la réunion d'un élément (la racine) et des sous-arbres qui lui sont directement associés.
- Chaque élément de l'ensemble structuré en arbre est appelé un **nœud**. À tout nœud est associée une information élémentaire.
- Pour décrire les relations entre les nœuds on utilise la terminologie de la généalogie, un nœud est donc le **père** de ses **fils**.
- Le **degré** d'un nœud est le nombre de ses fils. On distingue :
 - les **nœuds non terminaux** de degré non nul
 - les **nœuds terminaux** ou **feuilles**, de degré nul.

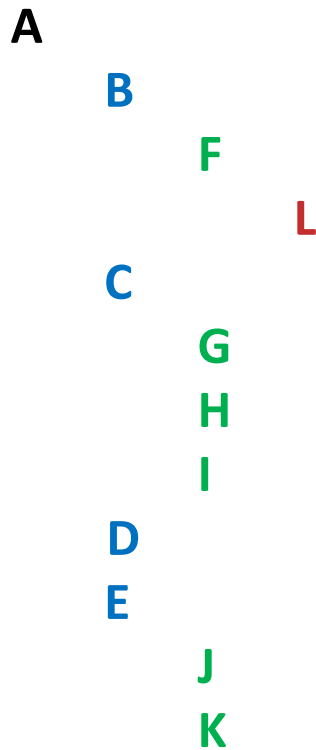
Arbres n-aires

- Pour structurer un ensemble **non vide** en arbre
 - Choisir un élément
 - Répartir les éléments restants en sous-ensembles **disjoints** et **les structurer en arbres**
- Exemples de représentations



Autres exemples de représentation

Indentation



Ecriture préfixée

(racine puis les sous-arbres)

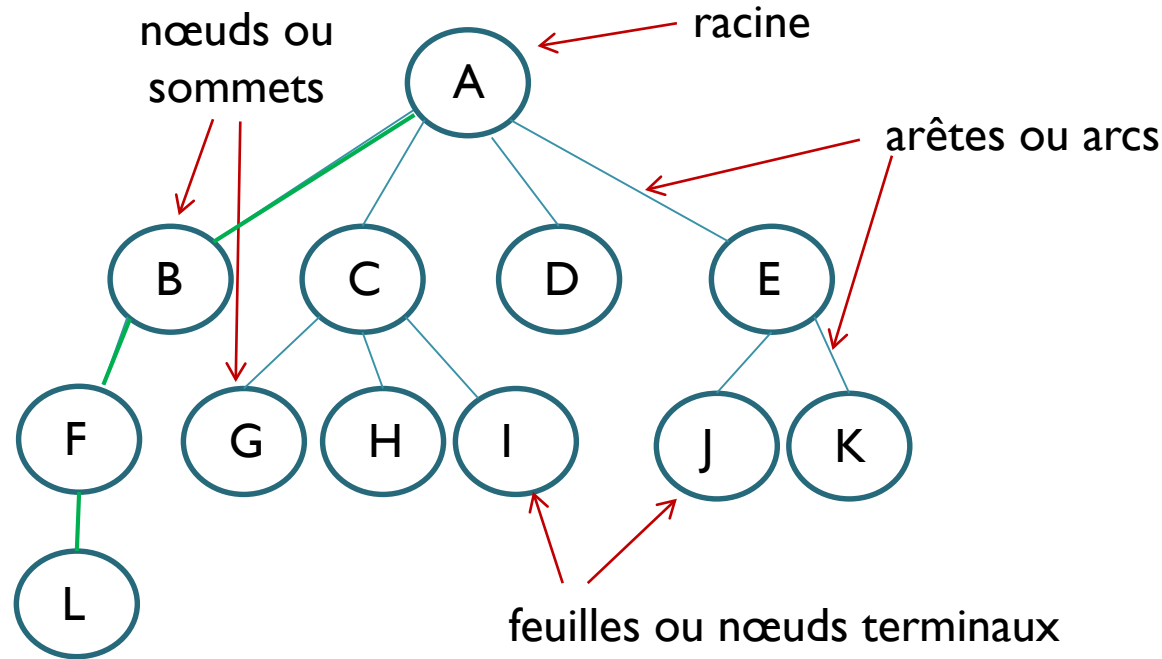
(A (B (F (L))) (C (G) (H) (I))) (D) (E (J) (K)))

Ecriture postfixée

(les sous-arbres puis la racine)

(((L) F) B) ((G) (H) (I) C) (D) ((J) (K) E) A)

Terminologie (graphes)



Définitions

Feuille : nœud **terminal**, n'a pas de fils.

Longueur d'un chemin : nombre de **nœuds** sur le chemin.

Niveau d'un nœud : longueur du chemin de la racine au nœud

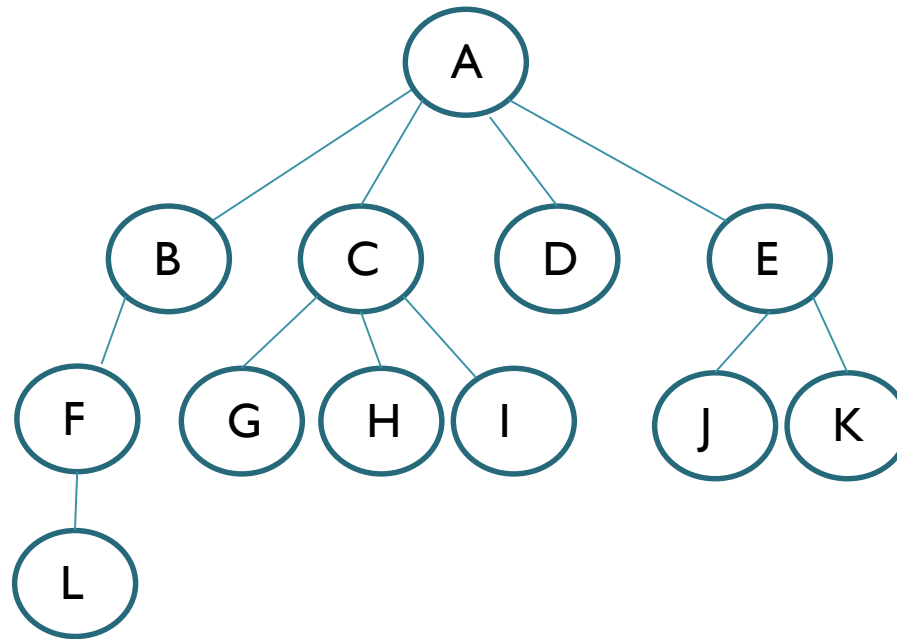
- niveau de la racine : 1
- niveau d'un nœud : 1 + niveau du nœud père

Profondeur d'un arbre : niveau maximum de l'arbre

Largeur d'un arbre : nombre maximum de nœuds d'un même niveau

Degré d'un nœud : nombre des ses fils

Terminologie (relations père-fils)



- A est la **racine** de l'arbre
- E est le **père** de J, J est le **fils** de E
- G, H et I sont **frères**
- F et L forment la **descendance** de B
- F, B et A forment l'**ascendance** de L
- F, H et E sont des **descendants** de A

Arbres n-aires

- La structure d'arbre représente un **ordre partiel** sur les nœuds.
- Deux nœuds sont en relation s'ils font partie de la même arborescence.
- Si de plus il existe un ordre entre les fils, on dit que **l'arbre est ordonné**.
- Un arbre n-aire est composé d'une **racine** et d'une **forêt** de sous-arbres.
- Une **forêt** est une séquence d'arbres (éventuellement vide)

Représentation informatique d'un arbre n-aire

Un arbre n-aire est représenté de manière chaînée. Chaque nœud de l'arbre est représenté par un triplet :

- un champ correspondant à la **valeur** du nœud
- un champ pointe sur la **liste des fils**, plus précisément sur le **fils aîné**
- un champ pointe sur la **liste des frères**, plus précisément sur le **frère cadet**

Valeur du nœud	Liste des fils (fils aîné)	Liste des frères (cadets)
el	fils	frère

Représentation informatique d'un arbre n-aire

Nœud : type agrégat

el : Elément // valeur

fil : Arbre // tête de la liste des fils

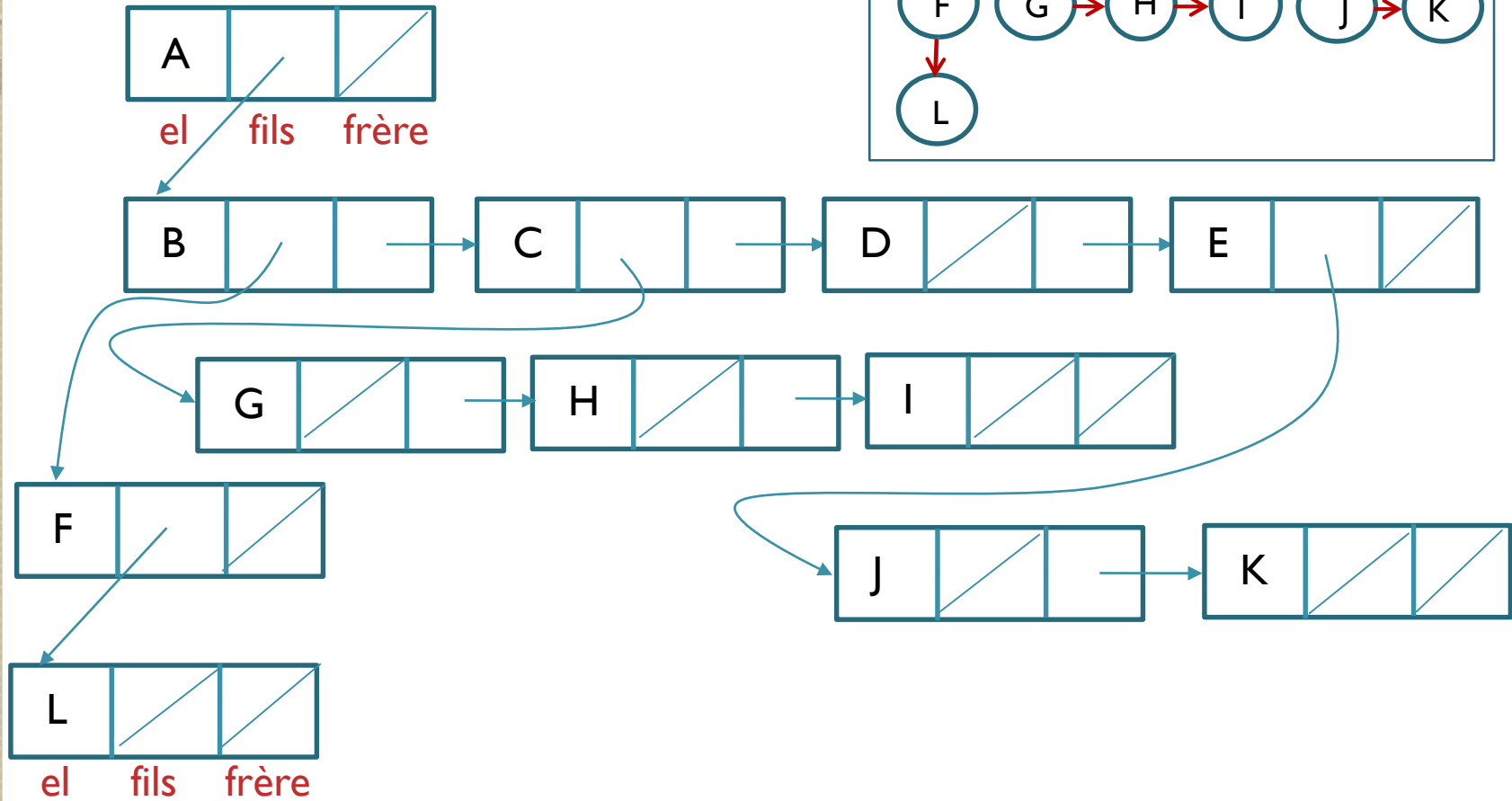
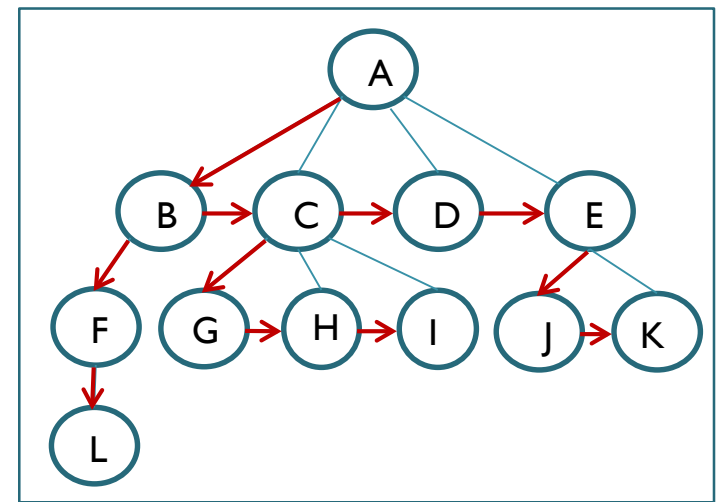
frère : Arbre // frère cadet

agrégat

Arbre : type pointeur de Nœud

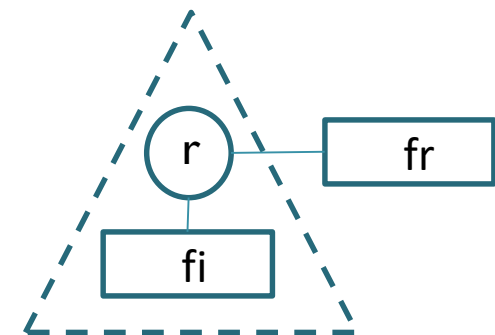
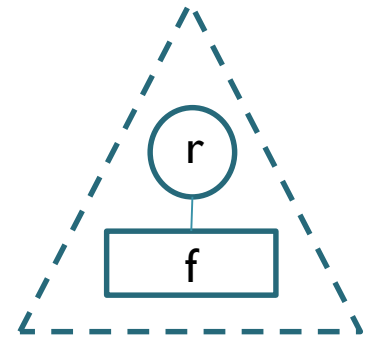
r : Arbre // arbre de racine r

Exemple



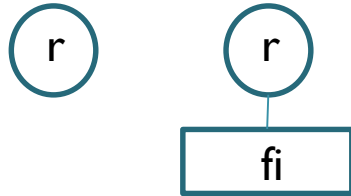
Principe d'analyse

- Un arbre n-aire est formé
 - de sa racine r
 - de la forêt de sous-arbres f (éventuellement vide)
- Une forêt est une séquence d'arbres formée
 - d'un premier arbre a
 - d'une forêt f
- Une forêt non vide est formée
 - d'un premier arbre de racine r
 - d'une forêt de sous-arbres fils fi
 - d'une forêt d'arbres frères fr
- Traiter un arbre n-aire :
 - Traiter la racine r
 - Traiter les forêts de sous-arbres fi et fr
- Traiter une forêt
 - Algorithme de parcours séquentiel : on énumère une séquence d'arbres éventuellement vide (récursif ou itératif)



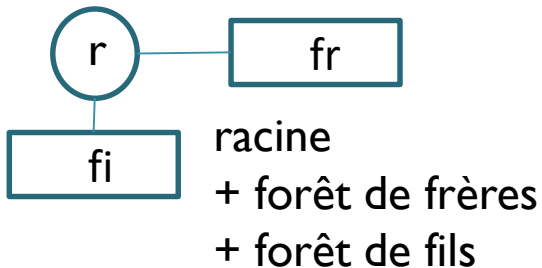
Modèles d'analyse

Arbre n-aire



Forêt vide incluse : 2 cas à examiner

<> forêt vide



Forêt non vide : 4 cas à examiner



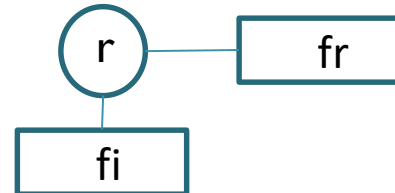
racine seule



forêt de fils
pas de frère



forêt de frères
pas de fils



forêt de frères
et forêt de fils

Modèles d'analyse

Forêt non vide ($r \neq \text{nil}$): 4 cas à examiner



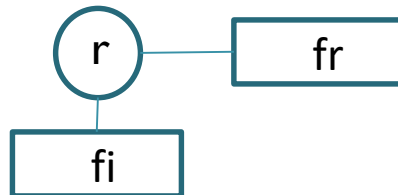
$r \uparrow . \text{fils} = \text{nil}$ et $r \uparrow . \text{frère} = \text{nil}$



$r \uparrow . \text{fils} \neq \text{nil}$ et $r \uparrow . \text{frère} = \text{nil}$



$r \uparrow . \text{fils} = \text{nil}$ et $r \uparrow . \text{frère} \neq \text{nil}$



$r \uparrow . \text{fils} \neq \text{nil}$ et $r \uparrow . \text{frère} \neq \text{nil}$

Modèles d'analyse

Arbre n-aire non vide



$r \uparrow . \text{fils} = \text{nil}$ et $r \uparrow . \text{frère} = \text{nil}$



$r \uparrow . \text{fils} \neq \text{nil}$ et $r \uparrow . \text{frère} = \text{nil}$



Forêt éventuellement vide : 2 cas à examiner

$\langle \rangle$

$r = \text{nil}$



$r \neq \text{nil}$



Exemple

- Calcul du nombre de nœuds d'un arbre n -aire

Nombre de nœuds d'un arbre n-aire

Solution 1 : arbre non vide, parcours de la forêt des fils

nbn_A : fonction (r : Arbre) \rightarrow entier > 0

// nbn_A(r) renvoie nombre de nœuds de l'arbre de racine r, r \neq nil

lexique

// paramètre r : Arbre racine de l'arbre

n : entier > 0 // résultat à calculer

ac : Arbre // sous-arbre courant

algorithme

n \leftarrow 1 // traitement de la racine

// traitement de la forêt des sous-arbres : itération de parcours

ac \leftarrow r \uparrow .fils

tantque ac \neq nil faire

n \leftarrow n + nbn_A(ac)

ac \leftarrow ac \uparrow .frère

ftq

renvoyer(n)

Nombre de nœuds d'un arbre n-aire

Solution 2: Parcours d'une forêt

nbn_f : fonction (f : Arbre) → entier ≥ 0

// nbn_f(f) renvoie nombre de nœuds de la forêt f (éventuellement vide)

lexique

// paramètre: f : Arbre forêt

algorithme

si f = nil alors renvoyer(0)

sinon renvoyer(1+nbn_f(f↑.fils) + nbn_f(f↑.frère))

fsi

nbn_A : fonction (r : Arbre) → entier > 0

// nbn_A(r) renvoie nombre de nœuds de l'arbre de racine r, r ≠ nil

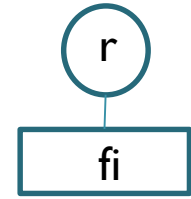
Algorithme

renvoyer(1+nbn_f(r↑.fils))

Fonctions facilitant l'écriture des algorithmes et faisant abstraction de la représentation de l'arbre n-aire

$\text{fils}(r) \rightarrow \text{Arbre}$	$r \uparrow . \text{fils}$
$\text{frère}(r) \rightarrow \text{Arbre}$	$r \uparrow . \text{frère}$
$\text{existeFils}(r) \rightarrow \text{booléen}$	$r \uparrow . \text{fils} \neq \text{nil}$
$\text{existeFrère}(r) \rightarrow \text{booléen}$	$r \uparrow . \text{frère} \neq \text{nil}$

Traitement séquentiel d'un arbre n-aire



- Parcourir un arbre n-aire consiste à traiter sa **racine** et la liste de **ses sous-arbres**.
- On peut décider de traiter d'abord la racine, puis les sous-arbres :
parcours préfixé (pré-ordre)
- Ou de traiter d'abord les sous-arbres puis la racine :
parcours postfixé (post-ordre)

Parcours préfixé d'un arbre n-aire non vide

action parcoursPréfixé (a : Arbre)

// applique « traiter » à tous les nœuds de a

lexique

// paramètre a : Arbre arbre à parcourir

ac : Arbre // sous-arbre courant

algorithme

traiter(a↑.el)

ac ← a↑.fils

tantque ac ≠ nil faire

parcoursPréfixé(ac)

ac ← ac↑.frère

ftq

≡ parcoursPréfixéForêt(a↑.fils)

Parcours préfixé d'une forêt

```
action parcoursPréfixéForêt (f : Arbre)
// applique « traiter » à tous les nœuds de la forêt f
lexique
// paramètre f : Arbre forêt à parcourir
algorithme
  si f ≠ nil
  alors
    traiter(f↑.el)
    parcoursPréfixéForêt(f↑.fils)
    parcoursPréfixéForêt(f↑.frère)
  fsi
```

Parcours postfixé

Il suffit de changer la place de traiter

action parcoursPostfixé (a : Arbre)

// applique « traiter » à tous les nœuds de a

lexique

// paramètre a : Arbre arbre à parcourir

ac : Arbre // sous-arbre courant

algorithme

ac ← a↑.fils

tantque ac ≠ nil faire

parcoursPostfixé(ac)

ac ← ac↑.frère

ftq

traiter(a↑.el)

action parcoursPostfixéForêt (f : Arbre)

// applique « traiter » à tous les nœuds de f

lexique

// paramètre : f : Arbre forêt à parcourir

algorithme

si f ≠ nil

alors

parcoursPostfixéForêt(f↑.fils)

traiter(f↑.el)

parcoursPostfixéForêt(f↑.frère)

fsi

action parcoursPostfixé (a : Arbre)

// applique « traiter » à tous les nœuds de a (version récursive

lexique

// paramètre : a : Arbre arbre à parcourir

action utilisée : parcoursPostfixéForêt

algorithme

parcoursPostfixéForêt(a↑.fils)

traiter(a↑.el)

Recherche dans un arbre n-aire non vide (préfixé)

fonction recherchePréfixé (a : Arbre) → Arbre

// renvoie l'adresse du premier élément vérifiant P dans l'ordre préfixé

lexique

// paramètre : a : Arbre arbre à parcourir

ac : Arbre // sous-arbre courant

x : Arbre // premier nœud vérifiant P

algorithme

si $P(a \uparrow .el)$ alors $x \leftarrow a$

sinon

$ac \leftarrow a \uparrow .fils ; x \leftarrow nil$

tantque $ac \neq nil$ et $x = nil$ faire

$x \leftarrow \text{recherchePréfixé}(ac)$

$ac \leftarrow ac \uparrow .frère$

ftq

fsi

renvoyer(x)

Recherche dans une forêt (préfixé)

fonction recherchePréfixéF(f : Arbre) → Arbre

// renvoie l'adresse du premier élément de f vérifiant P

// dans l'ordre préfixé

lexique

// paramètre : f : Arbre forêt à parcourir

x : Arbre // premier nœud vérifiant P

algorithme

si f = nil alors P(f↑.el)

alors x ← f

sinon x ← recherchePréfixéF(f↑.fils)

si x = nil alors x ← recherchePréfixéF(f↑.frère)

fsi

fsi

renvoyer(x)

Recherche dans une forêt (postfixé)

fonction recherchePostfixéF(f : Arbre) → Arbre

// renvoie l'adresse du premier élément de f vérifiant P

// dans l'ordre postfixé

lexique

// paramètre : f : Arbre forêt à parcourir

x : Arbre // premier nœud vérifiant P

algorithme

si f = nil

alors x ← nil

sinon x ← recherchePostfixéF(f↑.fils)

si x = nil

alors si P(f↑.el)

alors x ← f

sinon x ← recherchePostfixéF(f↑.frère)

fsi

fsi

fsi

renvoyer(x)

Exercice

- Ecrire une fonction qui calcule le nombre de nœuds de niveau n

solution 1 :

`nbnNivA : fonction(a : Arbre, n: entier > 0) → entier ≥ 0`

`// nombre de nœuds de niveau n dans l'arbre non vide a`

solution 2 :

`nbnNivF : fonction(f : Arbre , n: entier > 0) → entier ≥ 0`

`// nombre de nœuds de niveau n dans la forêt f`

Nombre de nœuds de niveau n (arbre n-aire non vide)

fonction nbnNivA : fonction(a : Arbre, n: entier > 0) → entier ≥ 0
// nombre de nœuds de niveau n dans l'arbre non vide a

lexique

// paramètre : a : Arbre arbre à parcourir

// paramètre : n : entier > 0 niveau

nb : entier ≥ 0 // valeur calculée : nombre de nœuds de niv. N

ac : Arbre // sous-arbre courant

algorithme

si n = 1 alors nb ← 1

sinon ac ← f↑.fils ; nb ← 0

tantque ac ≠ nil faire

nb ← nb + nbnNivA (ac, n-1)

ac ← ac↑.frère

ftq

fsi

renvoyer(nb)

Nombre de nœuds de niveau n (forêt)

fonction nbnNivF : fonction(f : Arbre, n: entier > 0) → entier ≥ 0
// nombre de nœuds de niveau n dans la forêt f

lexique

// paramètre : f : Arbre forêt à parcourir

// paramètre n : entier > 0 : niveau

Algorithme

si f = nil alors renvoyer(0)

sinon

si n = 1 alors renvoyer(1 + nbnNivF(f↑.frère, n))

sinon renvoyer (nbnNivF(f↑.fils, n-1) + nbnNivF(f↑.frère, n)

fsi

fsi

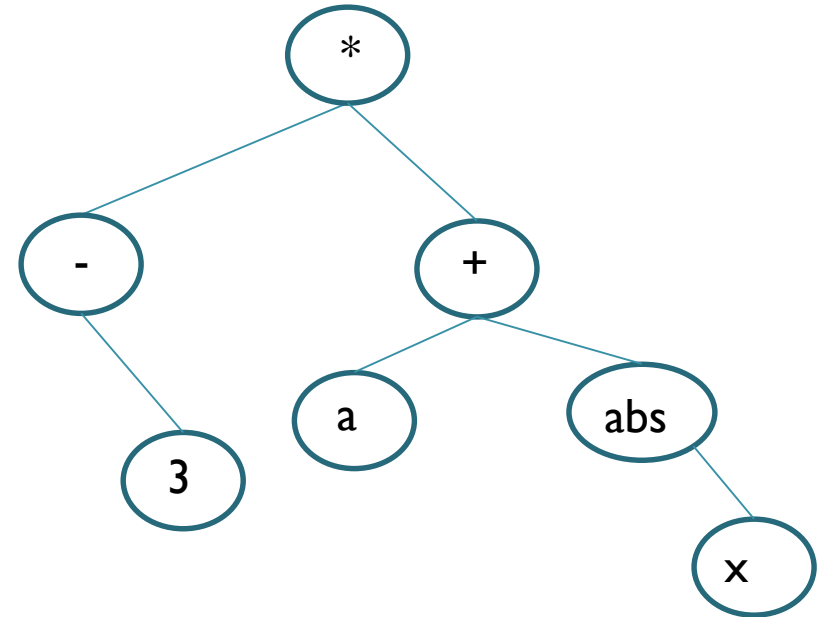
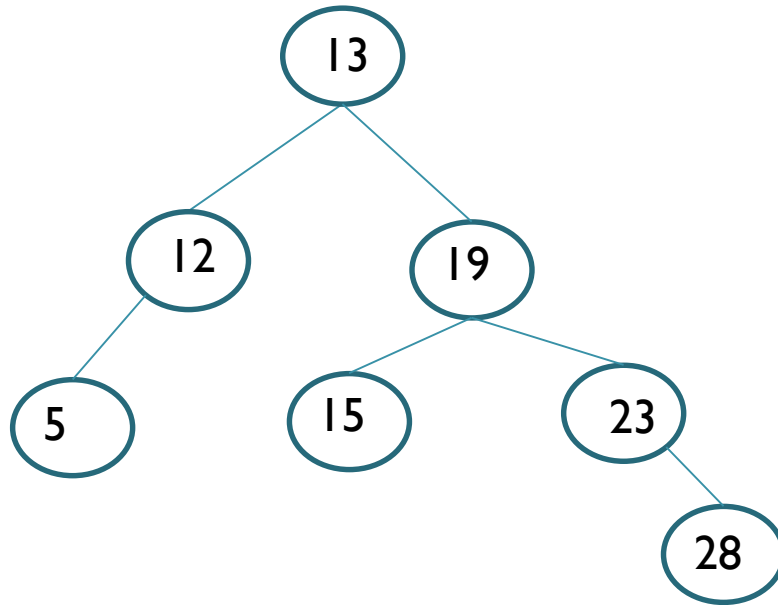


Arbres binaires

Arbres binaires

- Un **arbre binaire** est un arbre ordonné pour lequel tout nœud a au plus deux fils.
- Un arbre binaire est un ensemble fini qui est soit **vide**, soit composé d'une racine et de deux sous-arbres binaires appelés **sous-arbre gauche** et **sous-arbre droit**.
- On peut donc dire qu'un arbre binaire est :
 - soit l'arbre vide
 - soit un nœud qui a exactement deux sous-arbres éventuellement vides

Exemples d'arbres binaires



Représentation informatique d'un arbre binaire

Un arbre binaire est représenté de manière chaînée. Chaque nœud de l'arbre est représenté par un triplet :

- un champ correspondant à la **valeur** du nœud
- un champ pointe sur le **sous-arbre gauche**, plus précisément sur le **fils gauche**
- un champ pointe sur le **sous-arbre droit**, plus précisément sur le **fils droit**

Valeur du nœud	Sous-arbre gauche	Sous-arbre droit
----------------	-------------------	------------------

e_l

g

d

Représentation informatique d'un arbre binaire

Nœud : type agrégat

el : Elément // valeur

g : Arbre // sous-arbre gauche

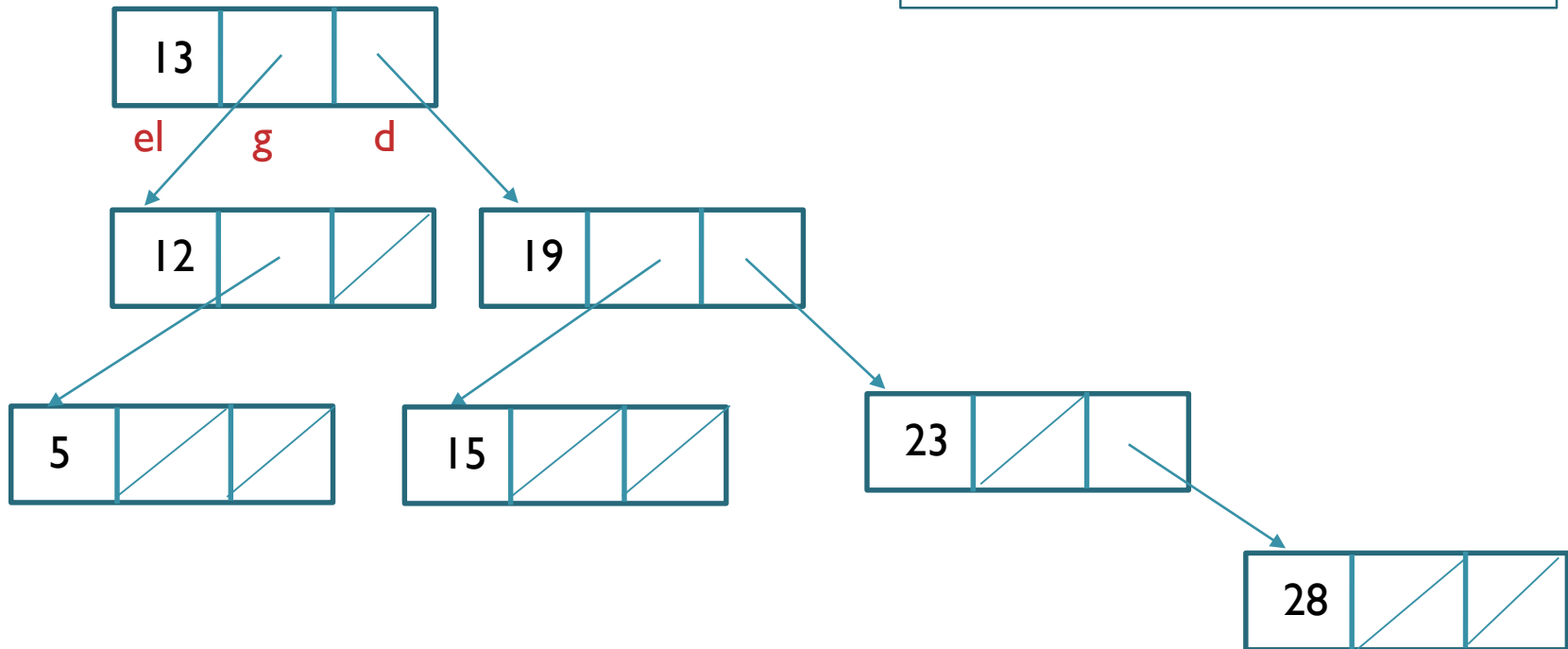
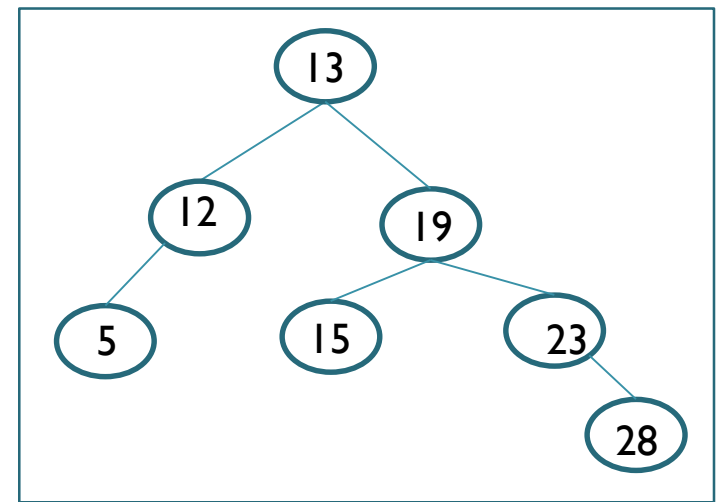
d : Arbre // sous-arbre droit

fagrégat

Arbre : type pointeur de Nœud

r : Arbre // arbre binaire de racine r

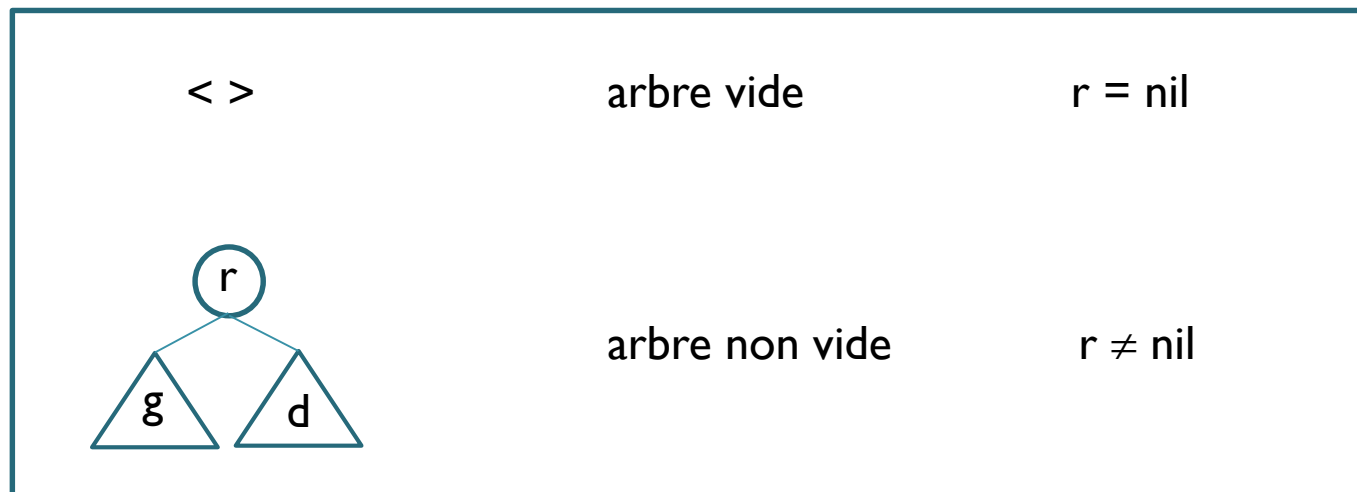
Exemple



Principes d'analyse

- Comme pour les arbres n-aires il existe deux formes d'analyse
 - Soit on considère l'arbre vide comme cas de base, et le cas général est un nœud racine avec 2 sous-arbres éventuellement vides

Modèle 1



Principes d'analyse

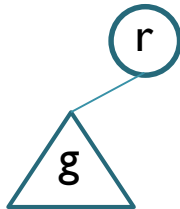
- Soit on considère un arbre non vide, dans ce cas nous avons en général 4 cas à examiner

Modèle 2



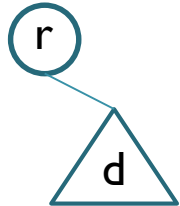
une racine (feuille)

$$r \uparrow .g = \text{nil} \text{ et } r \uparrow .d = \text{nil}$$



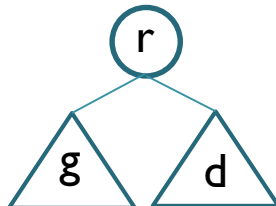
une racine et sous-arbre gauche non vide
(nœud unaire gauche)

$$r \uparrow .g \neq \text{nil} \text{ et } r \uparrow .d = \text{nil}$$



une racine et sous-arbre droit non vide
(nœud unaire droit)

$$r \uparrow .g = \text{nil} \text{ et } r \uparrow .d \neq \text{nil}$$



une racine et deux sous-arbre s non vides
(nœud binaire)

$$r \uparrow .g \neq \text{nil} \text{ et } r \uparrow .d \neq \text{nil}$$

Exemple : nombre de valeurs positives dans un arbre binaire

- Solution 1 : arbre éventuellement vide

nbPos : fonction (r : Arbre) \rightarrow entier ≥ 0

// nbp(r) renvoie nombre de nœuds de l'arbre de racine r ayant une valeur > 0

lexique

// paramètre r : Arbre racine de l'arbre

Algorithme

si r = nil

alors renvoyer(0)

sinon

si r.el > 0 alors renvoyer(1+ nbPos(r↑.g) + nbPos(r↑.d))

sinon renvoyer(nbPos(r↑.g) + nbPos(r↑.d))

fsi

fsi

Exemple : nombre de valeurs positives dans un arbre binaire

- Solution 2 : arbre non vide

nbPos : fonction (r : Arbre) → entier > 0

// nbp(r) renvoie nombre de nœuds de l'arbre de racine r ayant une valeur > 0

// r ≠ nil

lexique

// paramètre : r : Arbre racine de l'arbre examiné

nb : entier > 0 // nombre de valeurs positives trouvées

Algorithme

si r↑.el > 0 alors nb ← 1 sinon nb ← 0 fsi

selon r↑

r↑.g = nil et r↑.d = nil : // action vide

r↑.g ≠ nil et r↑.d = nil : nb ← nb + nbPos(r↑.g)

r↑.g = nil et r↑.d ≠ nil : nb ← nb + nbPos(r↑.d)

r↑.g ≠ nil et r↑.d ≠ nil : nb ← nb + nbPos(r↑.g) + nbPos(r↑.d)

fselon

renvoyer(nb)

Exemple : nombre de valeurs positives dans un arbre binaire

- Solution 2bis : arbre non vide

nbPos : fonction (r : Arbre) → entier > 0

// nbp(r) renvoie nombre de nœuds de l'arbre de racine r ayant une valeur > 0

// r ≠ nil

lexique

// paramètre : r : Arbre racine de l'arbre examiné

nb : entier > 0 // nombre de valeurs positives trouvées

Algorithme

si r↑.el > 0 alors nb ← 1 sinon nb ← 0 fsi

si r↑.g ≠ nil alors nb ← nb + nbPos(r↑.g) fsi

si r↑.d ≠ nil alors nb ← nb + nbPos(r↑.d) fsi

renvoyer(nb)

Nombre d'appels récursifs engendrés ?

- Pour un arbre binaire de N nœuds
- Modèle 1
 - $2N$ appels
- Modèle 2
 - $N-1$ appels
- Nombre de sous-arbres : $2N$
 - non vides : $N-1$
 - vides : $N+1$

Propriétés sur les arbres binaires

- Se démontrent par récurrence :
 - **Base** : on vérifie que la propriété est vraie pour l'arbre feuille
 - **H.R.** : on suppose la propriété vraie pour les sous-arbres
 - On démontre que la propriété est vraie pour l'arbre

- **Exemple**

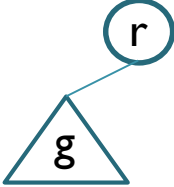
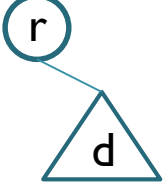
Un arbre binaire est composé de **N** nœuds :

- **B** nœuds binaires
- **U** nœuds unaires
- **F** feuilles

Démontrer que : $B = F - 1, \forall N > 0$

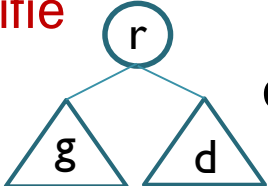
Propriétés sur les arbres binaires

- Démontrons que $B = F - 1, \forall N > 0$
- Base : si r est une feuille, $F = 1$ et $B = 0$ **vérifié** r
- H.R. : la propriété est vraie pour les sous-arbres **g** et **d**

- Pour les cas  et  propriété vérifiée car

$$B = B_g; F = F_g \quad B = B_d; F = F_d$$

l'ajout de r à l'arbre n'ajoute pas de nœud binaire ni de feuille \Rightarrow **vérifié**

- Pour le cas  on a : $B = B_g + B_d + 1$ et $F = F_g + F_d$

par H.R. on a : $B_g = F_g - 1$ et $B_d = F_d - 1$

$B = F_g - 1 + F_d - 1 + 1 = F_g + F_d - 1 = F - 1 \Rightarrow$ **vérifié**

Démonstration directe

- Démonstration directe de $B = F - 1$

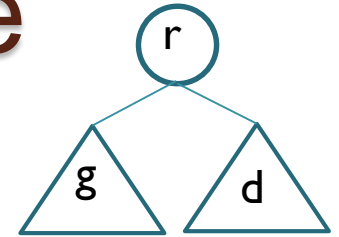
$$N = F + U + B \quad (1)$$

- Nombre de nœuds fils :
 - $N - 1$ (car on ne compte pas la racine)
 - Mais ce sont aussi tous les fils des nœuds unaires et tous les fils des nœuds binaires, soit au total : $U + 2B$
 - Donc $N - 1 = U + 2B$ (2)
- Soustraction des égalités (1) – (2) :
$$1 = F - B \Rightarrow B = F - 1$$

Fonctions facilitant l'écriture des algorithmes et faisant abstraction de la représentation de l'arbre binaire

$\text{gauche}(r) \rightarrow \text{Arbre}$	$r \uparrow . \text{gauche}$
$\text{droit}(r) \rightarrow \text{Arbre}$	$r \uparrow . \text{droit}$
$\text{existeG}(r) \rightarrow \text{booléen}$	$r \uparrow . g \neq \text{nil}$
$\text{existeD}(r) \rightarrow \text{booléen}$	$r \uparrow . d \neq \text{nil}$
$\text{feuille}(r) \rightarrow \text{booléen}$	$r \uparrow . g = \text{nil} \text{ et } r \uparrow . d = \text{nil}$
$\text{estVide}(r) \rightarrow \text{booléen}$	$r = \text{nil}$

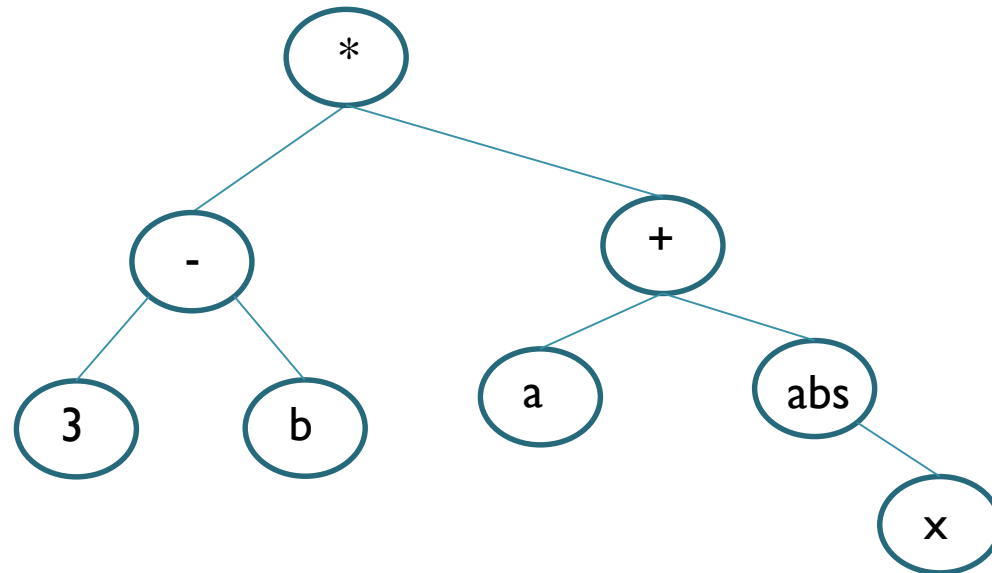
Parcours d'arbre binaire



3 parcours possibles :

- **Préfixé** (ou préordre):
 - racine, sous-arbre gauche, sous-arbre droit
- **Infixé** (ou ordre symétrique) :
 - sous-arbre gauche, racine, sous-arbre droit
- **Postfixé** (ou ordre terminal) :
 - sous-arbre gauche, sous-arbre droit, racine

Parcours d'arbre binaire



- Préfixé : * - 3 b + a abs x
- Infixé : 3 - b * a + abs x
- Postfixé : 3 b - a x abs + *

Parcours préfixé d'un arbre binaire

```
action parcoursPréfixé (a : Arbre)
// applique « traiter » à tous les nœuds de a (non vide)
lexique
  // paramètre a : Arbre arbre à parcourir
algorithme
traiter(a↑.el)
si a↑.g ≠ nil alors parcoursPréfixé(a↑.g) fsi
si a↑.d ≠ nil alors parcoursPréfixé(a↑.d) fsi
```

```
action parcoursPréfixé (a : Arbre)
// applique « traiter » à tous les nœuds de a (peut être vide)
lexique
// paramètre a : Arbre : arbre à parcourir
algorithme
si a ≠ nil alors
  traiter(a↑.el)
  parcoursPréfixé(a↑.g)
  parcoursPréfixé(a↑.d)
fsi
```


Parcours infixé d'un arbre binaire

action parcoursInfixé (a : Arbre)

// applique « traiter » à tous les nœuds de a (non vide)

lexique

// paramètre a : Arbre arbre à parcourir

algorithme

si a↑.g ≠ nil alors parcoursInfixé(a↑.g) fsi

traiter(a↑.el)

si a↑.d ≠ nil alors parcoursInfixé(a↑.d) fsi

action parcoursInffixé (a : Arbre)

// applique « traiter » à tous les nœuds de a (peut être vide)

lexique

// paramètre a : Arbre : arbre à parcourir

algorithme

si a ≠ nil alors

parcoursInfixé(a↑.g)

traiter(a↑.el)

parcoursInfixé(a↑.d)

fsi

Parcours postfixé d'un arbre binaire

action parcoursPostfixé (a : Arbre)

// applique « traiter » à tous les nœuds de a (non vide)

lexique

// paramètre a : Arbre arbre à parcourir

algorithme

si a↑.g ≠ nil alors parcoursPostfixé(a↑.g) fsi

si a↑.d ≠ nil alors parcoursPostfixé(a↑.d) fsi

traiter(a↑.el)

action parcoursPostfixé (a : Arbre)

// applique « traiter » à tous les nœuds de a (peut être vide)

lexique

// paramètre a : Arbre : arbre à parcourir

algorithme

si a ≠ nil alors

parcoursPostfixé(a↑.g)

parcoursPostfixé(a↑.d)

traiter(a↑.el)

fsi

Exercice

- On considère un arbre binaire dont les nœuds portent des valeurs entières
- Ecrire une **fonction** qui calcule la somme des valeurs des nœuds d'un tel arbre

Somme des éléments d'un arbre

fonction sommeAB : fonction(a : Arbre) → entier ≥ 0

// sommeAB(a) renvoie la somme des éléments de a

lexique de sommeAB

// paramètre a : Arbre : arbre dont on somme les éléments

algorithme de sommeAB

selon a

a = nil : renvoyer(0)

a \neq nil : renvoyer(a↑.el + sommeAB(a↑.g) + sommeAB(a↑.d))

fselon

Egalité de deux arbres binaires

fonction egauxAB : fonction(a : Arbre, b : Arbre) → booléen
// egauxAB(a,b) renvoie vrai si a et b ont même structure
// et mêmes valeurs

lexique de egauxAB

// paramètre a,b : Arbre : arbres binaires à comparer

algorithme de egauxAB

selon a,b

a = nil et b = nil : renvoyer(vrai)

a = nil et b ≠ nil : renvoyer(faux)

a ≠ nil et b = nil : renvoyer(faux)

a ≠ nil et b ≠ nil : renvoyer(a↑.el = b↑.el

etpuis egauxAB(a↑.g ,b↑.g)

etpuis egauxAB(a↑.d ,b↑.d)

fselon

Recopie d'un arbre binaire

action recopie(consulté a : Arbre, élaboré b : Arbre)
// effet: construit l'arbre b, copie de a

Lexique de recopie

// paramètre a : Arbre : arbre binaire à copier

// paramètre b : Arbre : copie de a

Algorithme de recopie

si a = nil alors b ← nil

sinon

créer(b) ;

b↑.el ← a↑.el

recopie(a↑.g, b↑.g)

recopie(a↑.d, b↑.d)

fsi

Construction d'une liste chaînée correspondant au parcours en ordre symétrique (infixé) d'un arbre binaire

Lexique partagé

Cellule : type agrégat el : Elément ; suc : AdrCel fagrégat

AdrCel : type pointeur de Cellule

Nœud : type agrégat el : Elément ; g : Arbre ; d : Arbre fagrégat

Arbre : type pointeur de Nœud

action creerListeSym(consulté a : Arbre ; élaboré t : AdrCel)

// Effet: Construit la liste chaînée t des éléments de l'arbre

// binaire a, parcouru en ordre symétrique

Idée : construction de la liste par des ajouts systématiques en tête de liste => Parcours en ordre symétrique inversé

Construction d'une liste chaînée correspondant au parcours en ordre symétrique d'un arbre binaire

action creerListeSym(consulté a : Arbre ; modifié t : AdrCel)

// Effet: Construit la liste chaînée t des éléments de l'arbre

// binaire a, parcouru en ordre symétrique

Lexique

k : AdrCel // adresse de la cellule créée

Algorithme

si a ≠ nil alors

 creerListeSym(a↑.d, t)

 créer(k) ; // on place la valeur de la racine en tête de la liste t

 k↑.el ← a↑.el ; k↑.suc ← t ; t ← k

 creerListeSym(a↑.g, t)

fsi

Pour construire la liste de tête t1 à partir de l'arbre a1,

Appel : t1 ← nil ; creerListeSym(a1,t1)

Recherche dans un arbre binaire (préfixé)

fonction rechercheP (a : Arbre) → booléen

// renvoie vrai si a comporte un élément vérifiant P

lexique

// paramètre a : Arbre : arbre à parcourir

algorithme

selon a

a = nil : renvoyer(faux)

a ≠ nil : renvoyer(P(a↑.el) oualors rechercheP(a↑.g)

oualors rechercheP(a↑.d))

fselon

Recherche dans un arbre binaire (préfixé)

fonction recherchePréfixé (a : Arbre) → Arbre

// renvoie l'adresse du premier élément vérifiant P dans l'ordre préfixé

lexique

// paramètre a : Arbre : arbre à parcourir

x : Arbre // adresse premier nœud vérifiant P

algorithme

si a = nil alors x ← nil

sinon

si P(a↑.el) alors x ← a

sinon

x ← recherchePréfixé(a↑.g)

si x = nil alors x ← recherchePréfixé(a↑.d) fsi

fsi

fsi

renvoyer(x)

Exercice

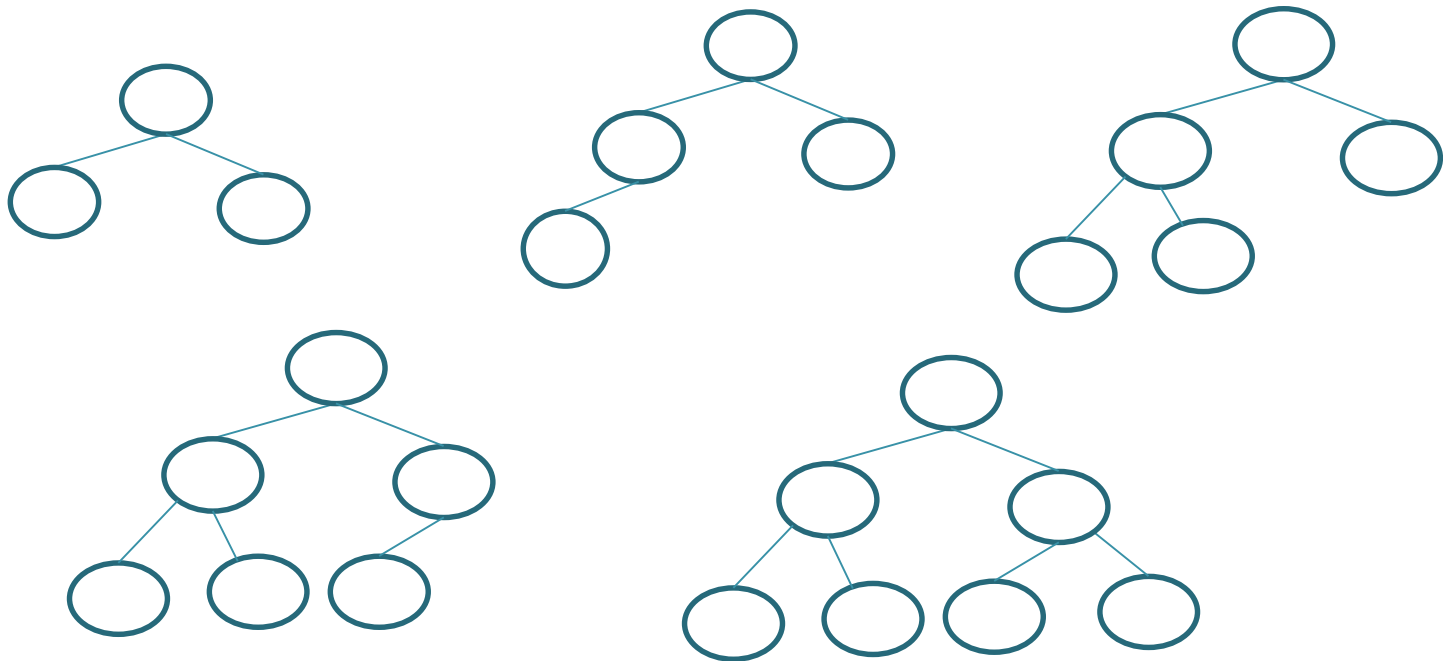
- Ecrire les algorithmes de recherche d'un nœud dans un arbre binaire vérifiant P
 - dans l'ordre infixé
 - dans l'ordre postfixé



Arbres binaires complets

Arbre binaire complet

- Cas particulier important des arbres binaires
- Arbre binaire dont toutes les feuilles sont au niveau p ou $p-1$ (p = profondeur)
- Arbre « plein » et « tassé à gauche »



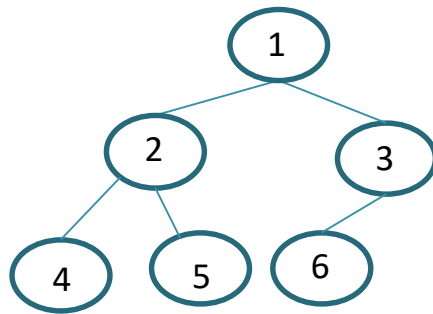
Arbre binaire complet

- Pour définir un arbre binaire complet, on considère les deux numérotations de nœuds suivantes:
 1. Numérotation par niveau de gauche à droite, la racine a le numéro 1 : $\text{num_niveau}(\text{racine}) = 1$
 2. La numérotation définie par récurrence comme suit:
 - Base : $\text{numéro}(\text{racine}) = 1$
 - Récurrence :
 $\text{numéro}(\text{gauche}(a)) = \text{numéro}(a)*2$
 $\text{numéro}(\text{droite}(a)) = \text{numéro}(a)*2+1$
- Un arbre binaire est **complet** si et seulement si les deux numérotations associent les mêmes numéros aux nœuds de l'arbre :

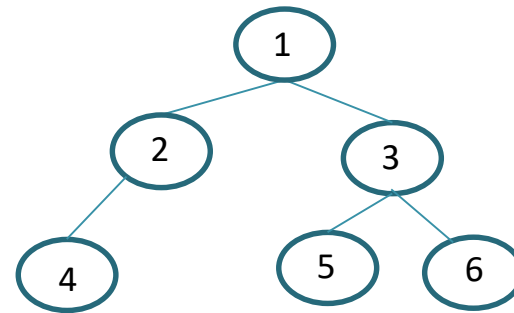
Pour tout nœud n de l'arbre on a :

$$\text{num_niveau}(n) = \text{numéro}(n)$$

Arbre binaire complet



Arbre binaire complet



Arbre binaire pas complet

Représentation contiguë d'un arbre binaire complet

- Séquence des n nœuds : mémorisée par niveau dans un tableau de taille n

T : tableau sur $[1..n]$ d'Élément

Racine en position 1

X en position i

Gauche(X) en position $i*2$

Droit(X) en position $i*2+1$

Père(X) en position $i \text{ div } 2$

Correspondance Arbre /Tableau

a : Arbre **a : entier sur 1..n**

$a \uparrow .el$ **T[a]**

a=nil **a > n**

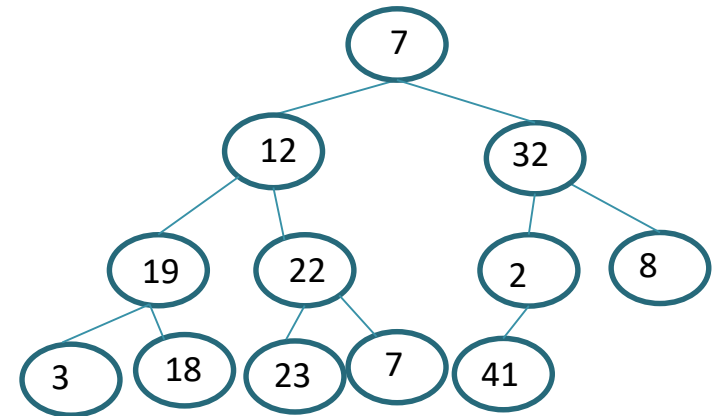
$a \uparrow .g$ **a*2**

$a \uparrow .d$ **a*2+1**

$a \uparrow .g \neq \text{nil}$ **$2*a \leq n$**

$a \uparrow .d \neq \text{nil}$ **$2*a+1 \leq n$**

feuille(a) **a*2 > n**



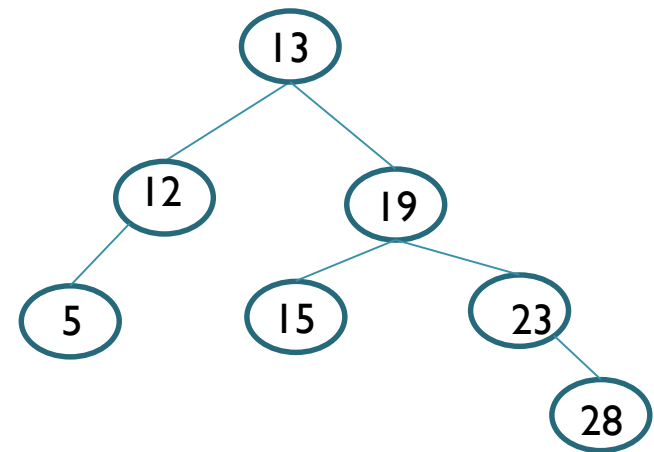
T	7	12	32	19	22	2	8	3	18	23	7	41	
	1	2	3	4	5	6	7	8	9	10	11	12	



Algorithmes **itératifs** de parcours d'arbres binaires

Algorithmes itératifs de parcours d'arbres binaires

- Parcours en profondeur
 - préfixé, infixé, postfixé
- Parcours en largeur
 - Parcours par niveau



- Préfixé: **13,12,5,19,15,23,28**
- Infixé: **5,12,13,15,19,23,28**
- Postfixé: **5,12,15,28,23,19,13**
- Largeur: **13,12,19,5,15,23,28**

Algorithmes itératifs de parcours d'arbres binaires

Principes

- Utilisation d'une structure de données **d** auxiliaire contenant des nœuds
- Deux primitives associées à **d**
 - **ajouter** (n, d)
 - **extraire** (n, d)
- Choix de **d**
 - Parcours en profondeur : **d** est une **pile**
 - Parcours par niveau : **d** est une **file**

La classe Pile

Classe Pile

t : tableau sur [1..n] d' Élément // pile de taille n

s : entier sur 0..n // indice de l'élément en sommet de pile

// **Méthodes publiques:**

public action créerPile (consulté n :entier > 0)

// creation d'une Pile vide de taille n (constructeur)

public sommet : fonction → Élément

// renvoie l'élément au sommet de la pile

public empiler : action (consulté e : Élément)

// place e au sommet de la pile

public dépiler : action (élaboré e : Élément)

// supprime l'élément en sommet de la pile et le place dans e

public décapiter : action

// supprime l'élément au sommet de la pile

public pileVide : fonction → booléen

// renvoie vrai si la pile est vide

public viderPile : action

// vide la pile

La classe File

Classe File

t : tableau sur [0..n-1] d' Élément // file de taille n (table circulaire)

p : entier sur 0..n-1 // indice du premier élément et de la file

d : entier sur 0..n-1 // indice du dernier élément et de la file

// Méthodes publiques:

public action creerFile (consulté n :entier > 0)

// creation d'une file vide de taille n (constructeur)

public premier : fonction → Élément

// renvoie la valeur du premier élément de la file

public dernier : fonction → Élément

// renvoie la valeur du dernier élément de la file

public enfiler : action (consulté e : Élément)

// place e en queue de la file

public défiler: action (elaboré e : Élément)

// supprime le premier élément de la file et le place dans e

public fileVide : fonction → booléen

// renvoie vrai si la file est vide

public viderFile : action

// vide la file

Parcours itératif d'arbre binaire

Schéma de parcours en ordre Préfixé

lexique

p : Pile // pile d'Arbres (pointeurs de nœuds)
n : entier > 0 // taille de la pile
a : Arbre // racine de l'arbre à parcourir en ordre préfixé
nc : Arbre // adresse du nœud courant

algorithmme

p.créerPile(n)

p.empiler(a)

initialisationTraitement

répéter

p.dépiler(nc) ; traiter(nc)

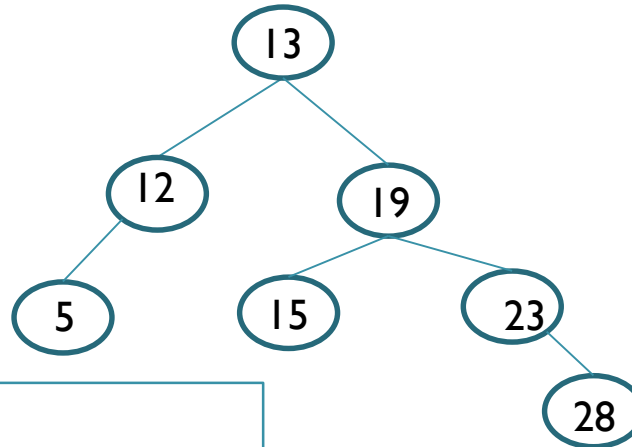
si existeD(nc) alors p.empiler(droit(nc)) fsi

si existeG(nc) alors p.empiler(gauche(nc)) fsi

jusqu'à p.pileVide

terminaisonTraitement

Parcours itératif en ordre **Préfixé**



algorithme

p.créerPile(n)

p.empiler(a)

initialisationTraitement

répéter

p.dépiler(nc) ; Traiter(nc)

si existeD(nc) alors

p.empiler(droit(nc))

fsi

si existeG(nc) alors

p.empiler(gauche(nc))

fsi

jusqu'à p.pileVide

Terminaison Traitement

Traitement	Pile
	@13
13	@12 @19
12	@5 @19
5	@19
19	@15 @23
15	@23
23	@28
28	

Parcours itératif d'arbre binaire

Schéma de parcours **par niveau**

lexique

f : File // file d'Arbres (pointeurs de nœuds)
n : entier > 0 // taille de la file
a : Arbre // racine de l'arbre à parcourir en ordre préfixé
nc : Arbre // adresse du nœud courant

algorithme

f.créerFile(n) // file vide de taille n créée

f.enfiler(a)

initialisationTraitement

répéter

d.défiler(nc) ; Traiter(nc)

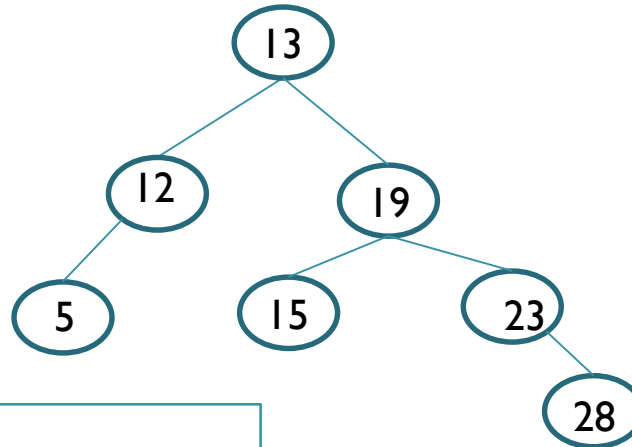
si existeG(nc) alors f.enfiler(gauche(nc)) fsi

si existeD(nc) alors f.enfiler(droit(nc)) fsi

jusqu'à f.fileVide

terminaisonTraitement

Parcours itératif par niveau



algorithme

f.créerFile(n) // file vide de taille n

f.enfiler(a)

initialisationTraitement

répéter

d.extraire(nc) ; Traiter(nc)

si existeG(nc) alors

f.enfiler(gauche(nc))

fsi

si existeD(nc) alors

f.enfiler(droit(nc))

fsi

jusqu'à f.fileVide

Terminaison Traitement

Traitement	File
	@13
13	@12 @19
12	@19 @5
19	@5 @15 @23
5	@15 @23
15	@23
23	@28
28	

Parcours itératif d'arbre binaire

- Parcours en ordre infixé et postfixé
- On a besoin de **marquer les nœuds qui ont été traités**
- Chaque nœud doit donc avoir un champ supplémentaire (booléen) indiquant s'il est marqué ou non.
- Pour gérer ce marquage, nous disposons des primitives suivantes :
 - fonction estMarqué(a : Arbre) → booléen
// renvoie vrai si la racine de a est marquée
 - action marquer(consulté a : Arbre)
// marque le nœud racine de a

Parcours itératif d'arbre binaire

Schéma de parcours en ordre infixé

lexique

p : Pile // pile d'Arbres (pointeurs de nœuds)

n : entier > 0 // taille de la pile

a : Arbre // racine de l'arbre à parcourir en ordre préfixé

nc : Arbre // adresse du nœud courant

algorithme

p.créerPile(n)

p.empiler(a)

initialisationTraitement

répéter

nc ← p.sommet // on ne dépile pas

si existeG(nc) et puis non estMarqué(gauche(nc)) alors p.empiler(gauche(nc))

sinon

traiter(nc) ; marquer(nc);

p.décapiter

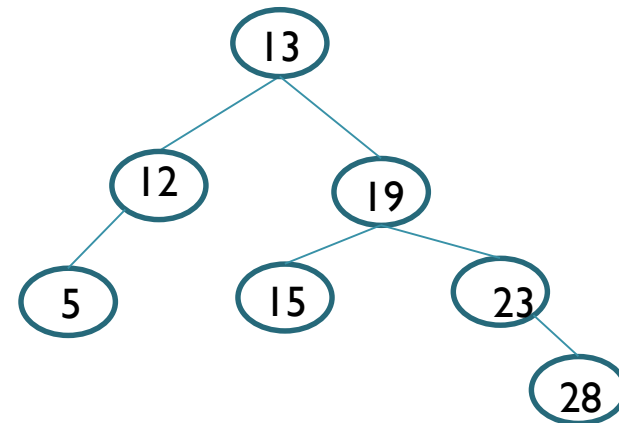
si existeD(nc) alors p.empiler(droit(nc)) fsi

fsi

jusqu'à p.pileVide

terminaisonTraitement

Parcours itératif en ordre **Infixé**



algorithme

p.créerPile(n)

p.empiler(a)

initialisationTraitement

répéter

nc ← p.sommet

si existeG(nc) et puis non estMarqué(gauche(nc))

alors p.empiler(gauche(nc))

sinon

traiter(nc) ; marquer(nc);

p.décapiter

si existeD(nc) alors p.empiler(droit(nc)) fsi

fsi

jusqu'à p.pileVide

terminaisonTraitement

Traitement	Pile
	@13
	@12 @13
	@5 @12 @13
5	@12 @13
12	@13
13	@19
	@15 @19
15	@19
19	@23
23	@28
28	

Parcours itératif d'arbre binaire

Schéma de parcours en **ordre postfixé**

lexique

p : Pile // pile d'Arbres (pointeurs de nœuds)

n : entier > 0 // taille de la pile

a : Arbre // racine de l'arbre à parcourir en ordre préfixé

nc : Arbre // adresse du nœud courant

algorithme

p.créerPile(n)

p.empiler(a)

initialisationTraitement

répéter

nc ← p.sommet // on ne dépile pas

si existeG(nc) et puis non estMarqué(gauche(nc)) alors p.empiler(gauche(nc))

sinon

si existeD(nc) et puis non estMarqué(droit(nc)) alors p.empiler(droit(nc))

sinon

traiter(nc) ; marquer(nc);

p.décapiter

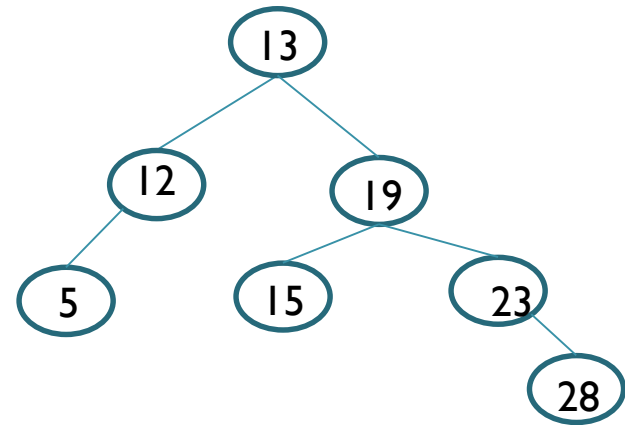
fsi

fsi

jusqu'à p.pileVide

terminaisonTraitement

Parcours itératif en ordre Postfixé



algorithme

p.créerPile(n)

p.empiler(a)

initialisationTraitement

répéter

nc ← p.sommet

si existeG(nc) et puis non estMarqué(gauche(nc))

alors p.empiler(gauche(nc))

sinon

si existeD(nc) et puis non estMarqué(droit(nc))

alors p.empiler(droit(nc))

sinon

traiter(nc) ; marquer(nc);

p.décapiter

fsi

fsi

jusqu'à p.pileVide

terminaisonTraitement

Traitement

Pile

	@13
	@12 @13
	@5 @12 @13
5	@12 @13
12	@13
	@19 @13
	@15 @19 @13
15	@19 @13
	@23 @19 @13
	@28 @23 @19 @13
28	@23 @19 @13
23	@19 @13
19	@13
13	

Parcours itératif d'arbre n-aire

Préfixé

Initialiser le traitement

p.viderPile

p.empiler(racine)

répéter

p.dépiler(nc) ; traiter(nc)

empiler tous les fils de nc

jusqu'à p.pileVide

Par niveau Initialiser le
traitement

f.viderfile

f.enfiler(racine)

répéter

f.défiler(nc) ; traiter(nc)

fc ← fils(nc)

tantque fc ≠ nil faire

f.enfiler(fc) ; fc ← frère(fc)

ftq

jusqu'à f.fileVide

Postfixé

Initialiser le traitement

p.viderPile

p.empiler(racine)

répéter

nc ← p.sommet

si nc a un fils non marqué

alors

p.empiler(fils non marqué)

sinon

traiter(nc) ; **marquer(nc)**

p.decapiter

fsi

jusqu'à p.pileVide

Parcours préfixé d'arbre n-aire

p.viderPile

p.empiler(racine)

initialiser le traitement

répéter

p.dépiler(nc) ; traiter(nc)

// empiler tous les fils de nc dans leur ordre inverse

// => utilisation d'une seconde pile p2

fc ← fils(nc) ; p2.viderPile

tantque fc ≠ nil faire

p2.empiler(fc) ; fc ← frère(fc)

ftq

// dernier frère en sommet de la pile p2

tantque non p2.pileVide faire

p2.dépiler(fc); p.empiler(fc)

ftq

jusqu'à p.pileVide

Parcours postfixé d'arbre n-aire

Initialiser le traitement

p.viderPile

p.empiler(racine)

répéter

nc ← p.sommet

fc ← fils(nc)

tantque fc ≠ nil etpuis estMarqué(fc) faire

fc ← frère(fc)

ftq // fc = nil oualors non estMarqué(fc)

si fc ≠ nil

alors

p.empiler(fc)

sinon

traiter(nc) ; marquer(nc)

p.decapiter

fsi

jusqu'à p.pileVide

