TYPES de DONNEES

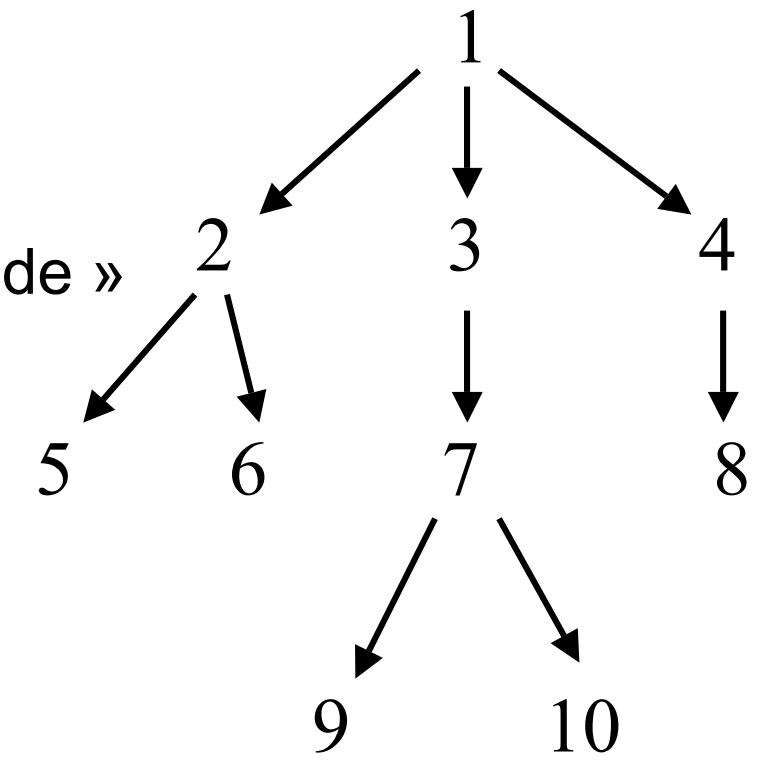
Esial 1ère année

Année 2006-2007

Type Arbre

Présentation

- Arbre ordinaire : $A = (N,P)$
  - N ensemble des nœuds
  - P relation binaire « parent de »
  - $r \in N$ la racine



Propriétés

$\forall\ x \in N\ \ \exists$ un seul chemin de r vers x

$r = y_o\ P\ y_1\ P\ y_2\ \ldots\ P\ y_n = x$

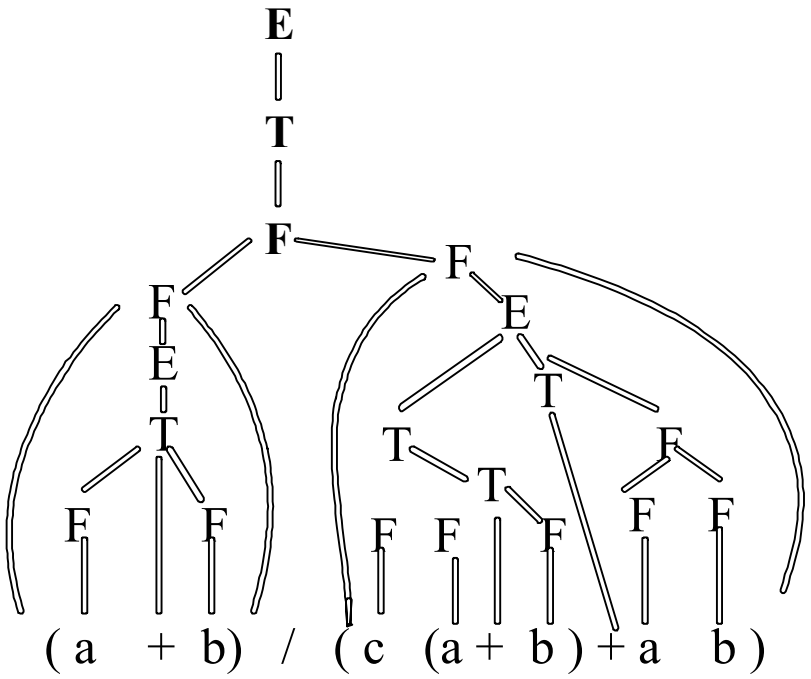
$\Rightarrow r$ n'a pas de parent

$\Rightarrow \forall\ x \in N - \{ r \}$ x a exactement un parent

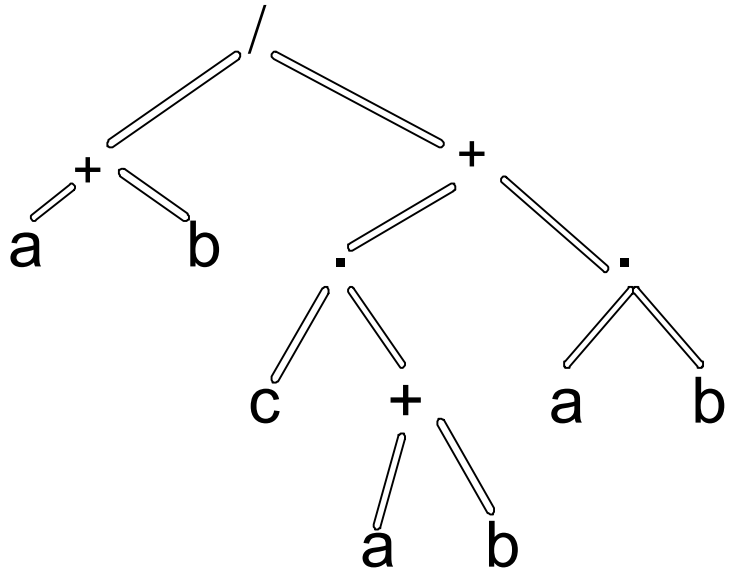
Etiquettes

Étiquette : $N \rightarrow E$

Arbre d'analyse,
pour une grammaire



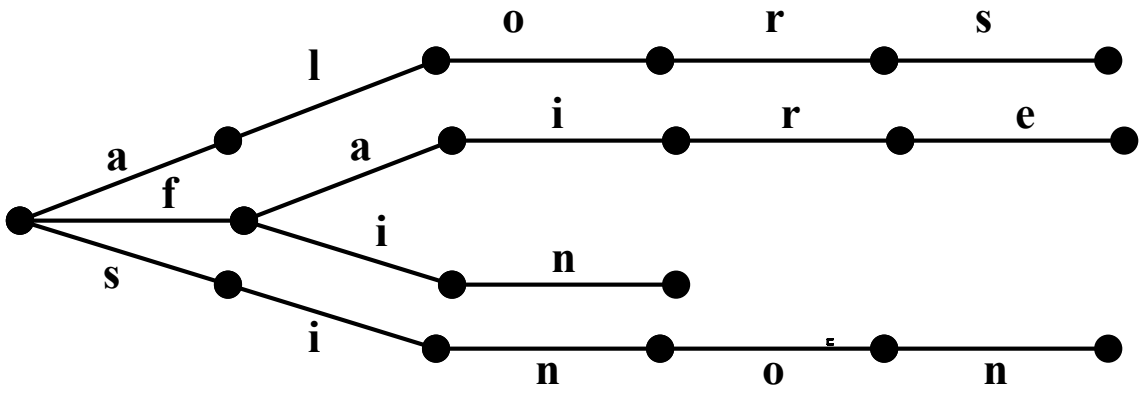
Arbre syntaxique,
arbre d'exécution



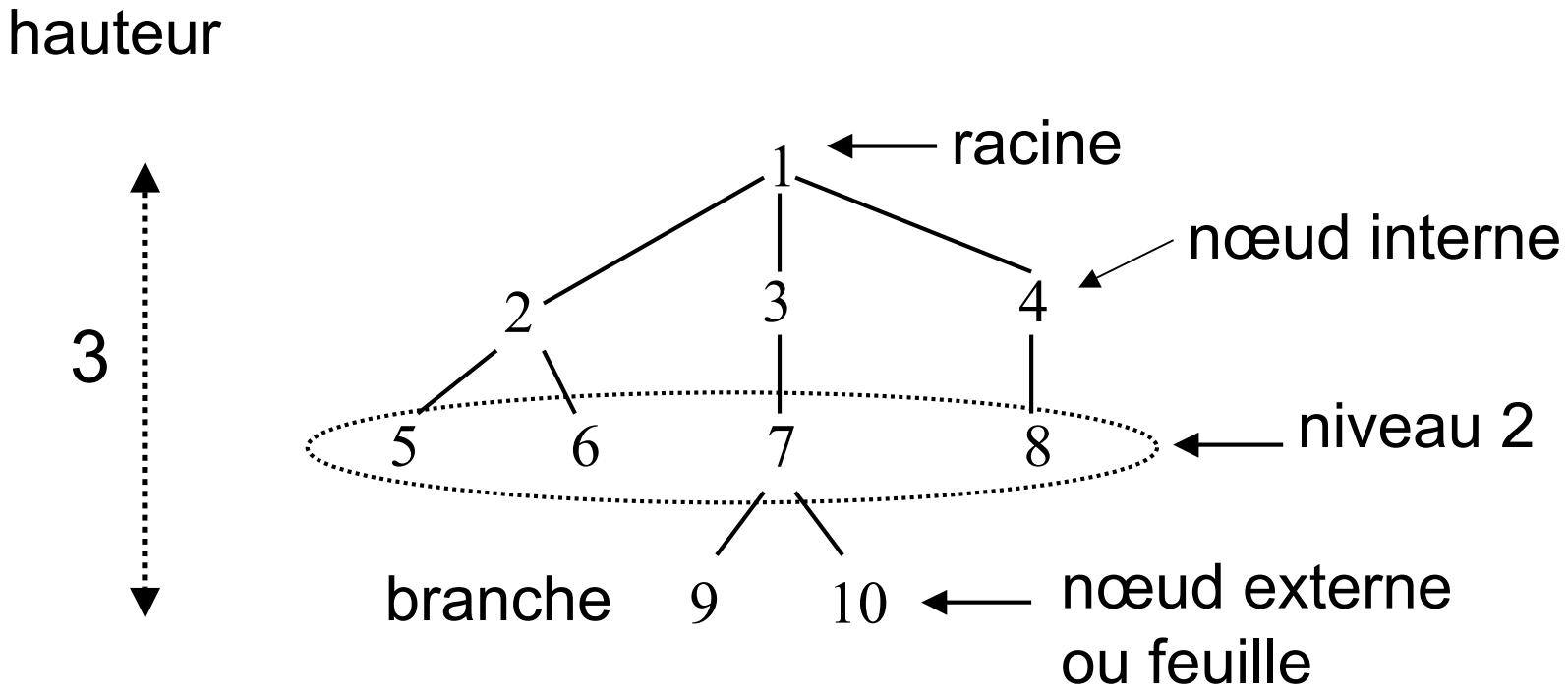
Arbres lexicogrqphiques

Arcs (*A*) = { (*x*,*y*) / *x* parent de *y* }

Étiquette : Arcs (*A*) → *E*



Terminologie



2, 3, 4	enfants de 1
3, 4	frères de 2
1, 3, 7	ancêtres de 7
7, 9, 10	descendants de 7

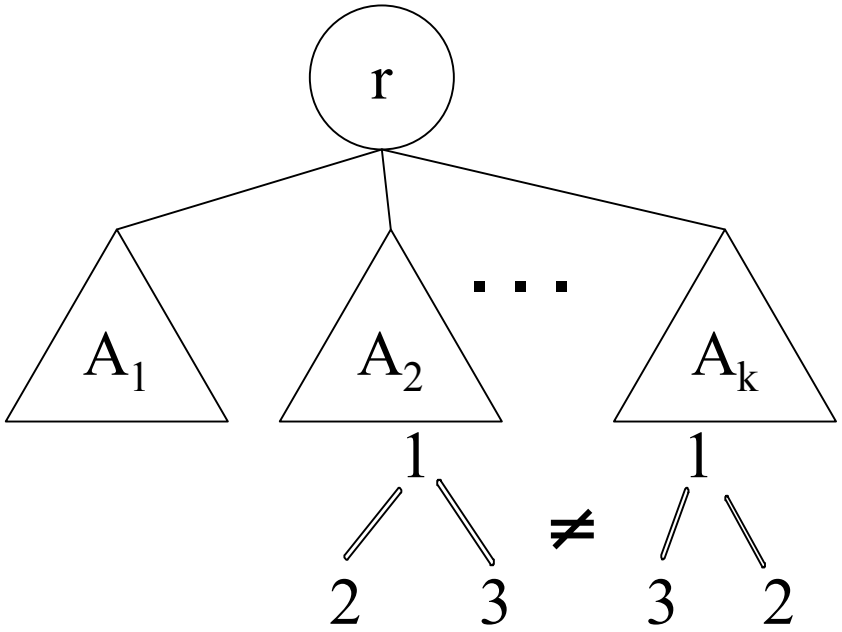
Définitions récursives

Arbre A = - Λ arbre vide ou
- (r, A_1, ..., A_k)

r élément, A_1, ..., A_k arbres

Nœuds (A) = {r} $\cup$ ($\cup$ Nœuds (A_i))

$A = \Lambda$ ou

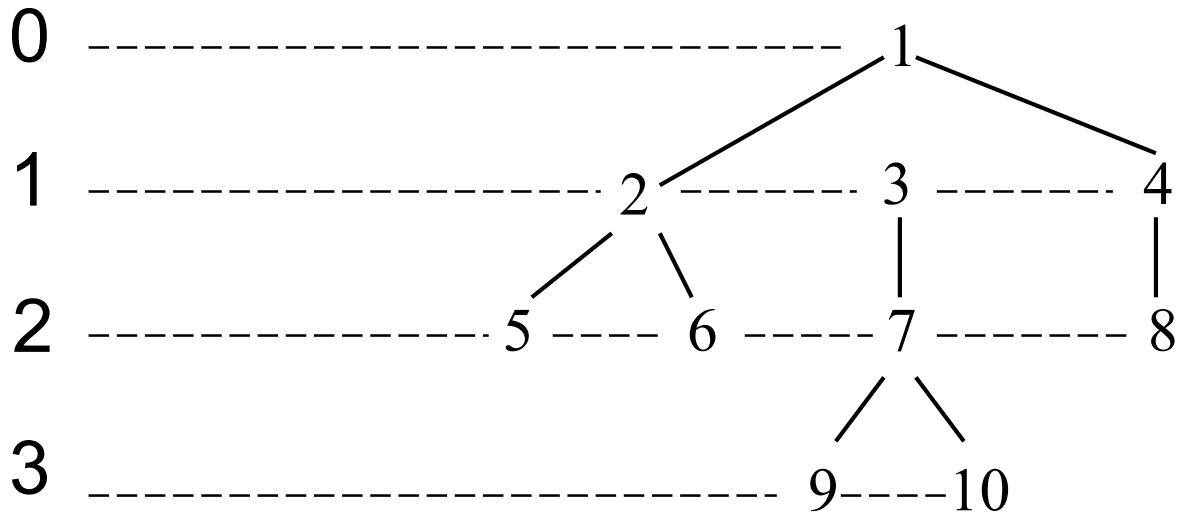


Niveaux

A arbre *x* nœud de *A*

niveau$_A$(*x*) = distance de *x* à la racine

niveau$_A$(*x*) = - 0 si *x* = racine(*A*)
- 1 + niveau$_A$(parent (*x*)) sinon



Hauteurs

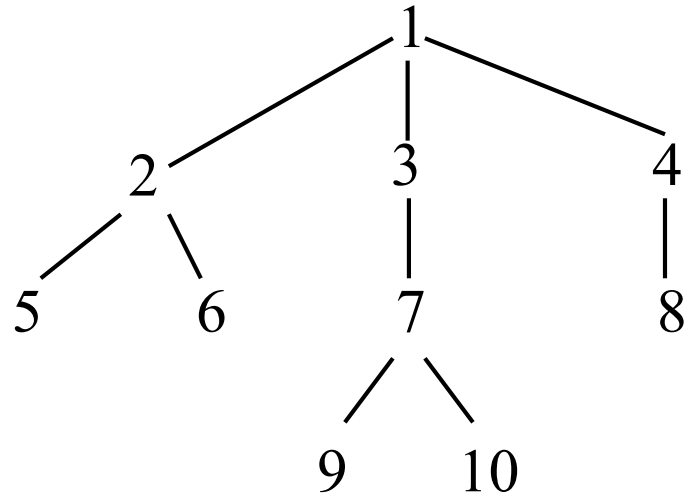
A arbre *x* nœud de *A*

$h_A(x)$ = distance de *x* à son plus lointain descendant qui est un nœud externe

$h_A(x)$ =
- 0 si *x* nœud externe
- 1 + max { $h_A(e)$ | *e* enfant de *x* } sinon

$h(A) = h_A(\text{racine}(A))$

$h_A(8) = 0$
$h_A(7) = 1$
$h_A(3) = 2$
$h(A) = h_A(1) = 3$



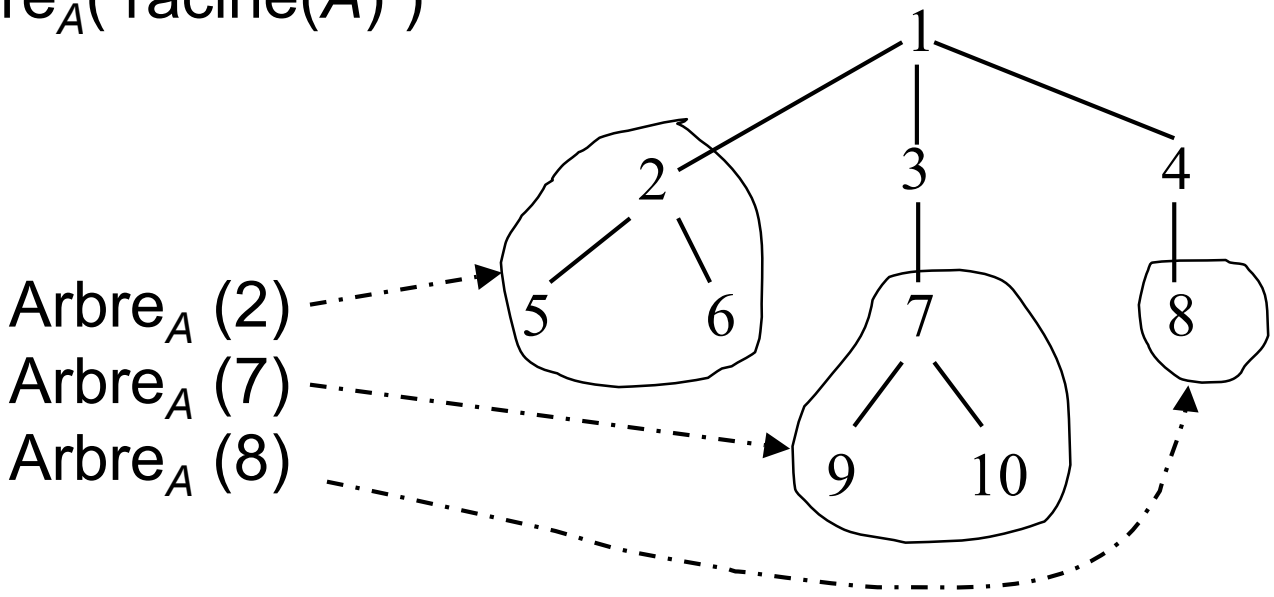
Sous-arbres

A arbre *x* nœud de *A*

$Arbre_A(x)$ = sous-arbre de *A* qui a racine *x*

$h_A(x) = h(\ Arbre_A(x)\ )$

$A = Arbre_A(\ racine(A)\ )$



Parcours

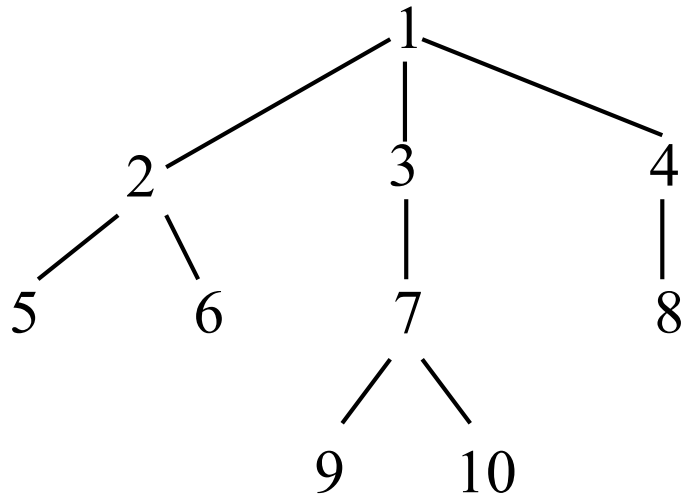
Fonction : arbre → liste de ses nœuds
arbre vide → liste vide

Utile pour l ’exploration des arbres

Deux types :

- parcours en profondeur
  - préfixe, suffixe, symétrique
  - parcours branche après branche
- parcours en largeur ou hiérarchique
  - parcours niveau après niveau

Parcours en profondeur



Arbre non vide $A = (r, A_1, A_2, \ldots, A_k)$

Parcours préfixe

$P(A) = r.P(A_1). \ldots .P(A_k)$ (1, 2, 5, 6, 3, 7, 9, 10, 4, 8)

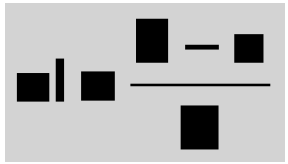
Parcours suffixe

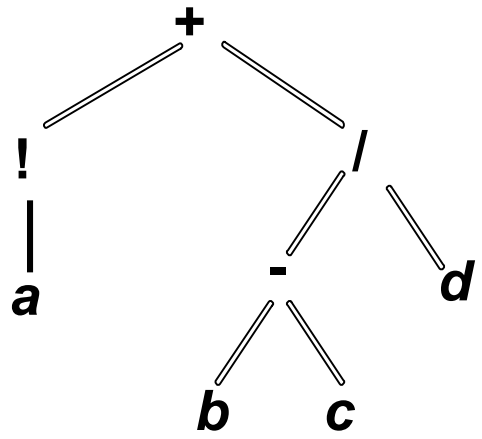
$S(A) = S(A_1). \ldots .S(A_k).r$ (5, 6, 2, 9, 10, 7, 3, 8, 4, 1)

Parcours symétrique (ou interne)

$I(A) = I(A_1).r.I(A_2). r \ldots r.I(A_k)$ (5, 2, 6, 1, 9, 7, 10, 3, 8, 4)

Expressions arithmétiques

Arbre syntaxique de 

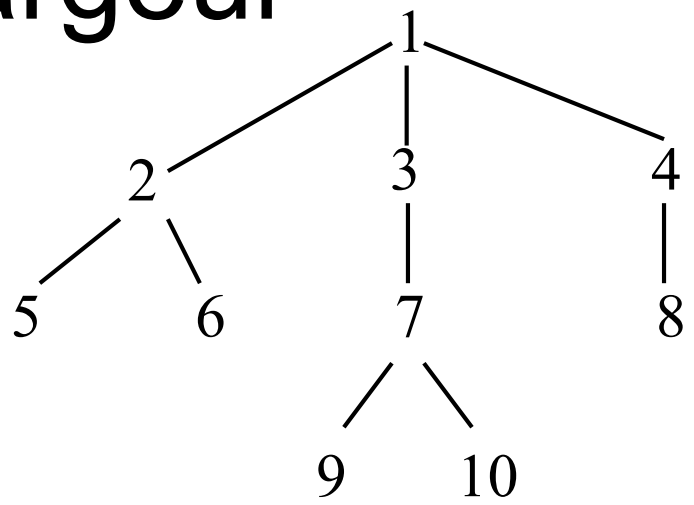


Parcours préfixe **+ ! *a* / - *b* *c* *d***

Parcours suffixe ***a* ! *b* *c* - *d* / +**

Parcours symétrique (parenthèses) **(*a* !) + ((*b* - *c*) / *d*)**

Parcours en largeur



Arbre non vide $A = (r, A_1, A_2, \ldots, A_k)$

Parcours hiérarchique

$$H(A) = (r, \underbrace{x_1, \ldots, x_i}_{1}, \underbrace{x_{i+1}, \ldots x_j}_{2}, x_{j+1}, \ldots, x_n)$$

nœuds de niveau 0, 1, 2, ...

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Opérations sur les arbres

Sorte Arbre

Opérations

arbre_vide	: → Arbre
racine	: Arbre → Noeud
enfants	: Arbre → Liste[Arbre]
cons	: Nœud x Liste[Arbre] → Arbre
est_vide	: Arbre → Booléen
hauteur	: Arbre → Entier

Axiomatisation

racine(cons(N,L)) = N

enfants(cons(N,L)) = L

hauteur(arbre_vide) = 0

hauteur(cons(N,L)) = 1 + max(HL(L))

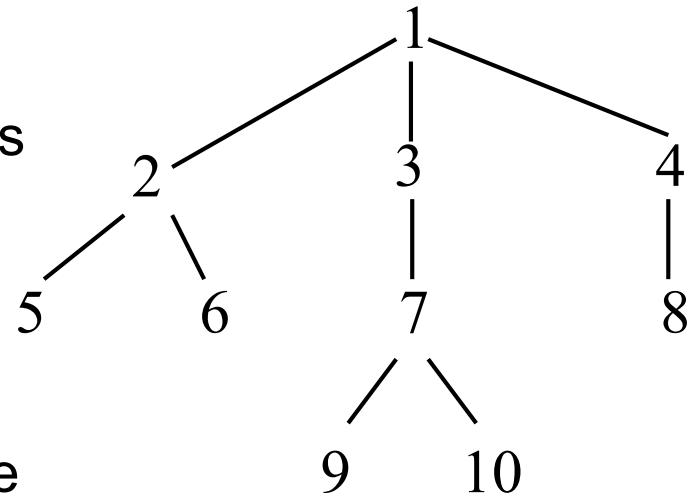
Comment définir HL ?

HL : Liste[Arbre] → Liste[Entier]

HL(liste_vide) = 0

HL(cons_liste(A,L')) = cons_liste(hauteur(A),HL(L'q))

Implantation



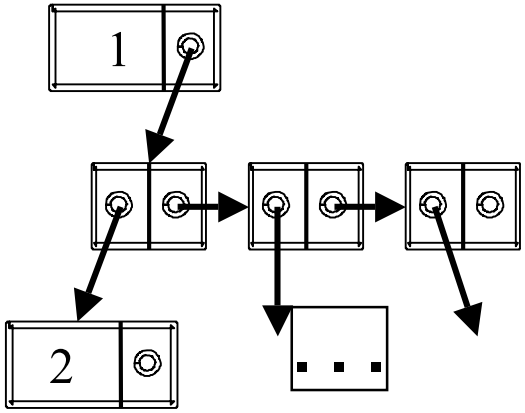
Représentation des listes de sous-arbres par chaînage

Avantages

- accès faciles depuis la racine
- correspond à la définition récursive

Inconvénients

- parcours difficiles vers la racine
- relativement gourmand en mémoire



Parcours préfixe

```
Préfixe() = L : Liste[Nœud]
L ← créer_liste()
si est_vide() alors retour
sinon
        L.ajout(Racine())
        pour B ← premier au dernier élément de Enfants() faire
                        L.concaténer(B.Préfixe())
        finpour
finsi
```

Temps d ’exécution : $O(n)$
sur un arbre à n nœuds représenté par pointeurs

Parcours suffixe

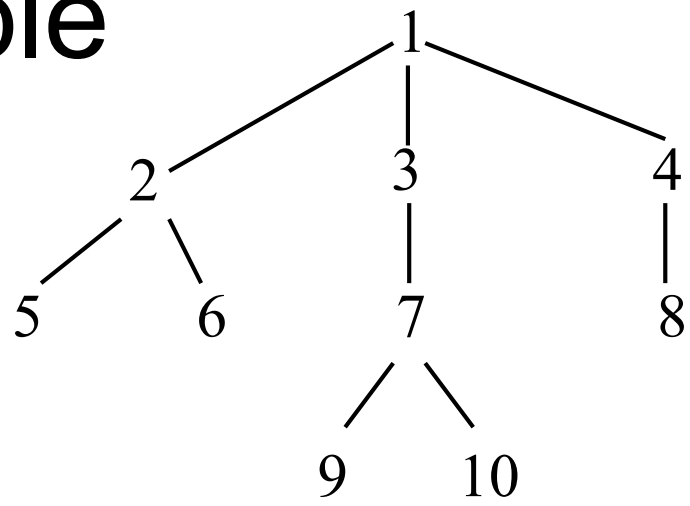
```
Suffixe() = L : Liste[Nœud]
L ← créer_liste()
si est_vide() alors retour
sinon
        pour B ← premier au dernier élément de Enfants() faire
                        L.concaténer(B.Suffixe())
        finpour
        L.ajout(Racine())
finsi
```

Temps d ’exécution : $O(n)$
sur un arbre à n nœuds représenté par pointeurs

Parcours préfixe itératif

```
Préfixe_iter() = L : Liste[Nœud]
L ← créer_liste()
Pile ← créer_pile().empiler(current)
tantque ¬Pile.est_vide() faire
        A' ← Pile.sommet()
        Pile.dépiler()
        si ¬A'.est_vide() alors
                L.ajout(A'.Racine())
                pour B ← dernier au premier élément de A'.Enfants() faire
                        Pile.empiler(B)
                finpour
        finsi
fintantque
```

Exemple



Liste *L* = 1 2 5 6

Pile = 1

A'=1 Pile = 4 3 2

A'=2 Pile = 4 3 6 5

A'=5 Pile = 4 3 6

A'=6 Pile = 4 3

etc.

```
Préfixe_iter() = L : Liste[Nœud]
L ← créer_liste()
Pile ← créer_pile().empiler(current)
tantque ¬Pile.est_vide() faire
        A' ← Pile.sommet()
        Pile.dépiler()
        si ¬A'.est_vide() alors
                L.ajout(A'.Racine())
                pour B ← dernier au premier élément de A'.Enfants() faire
                        Pile.empiler(B)
                finpour
        finsi
fintantque
```

Arbres k-aires

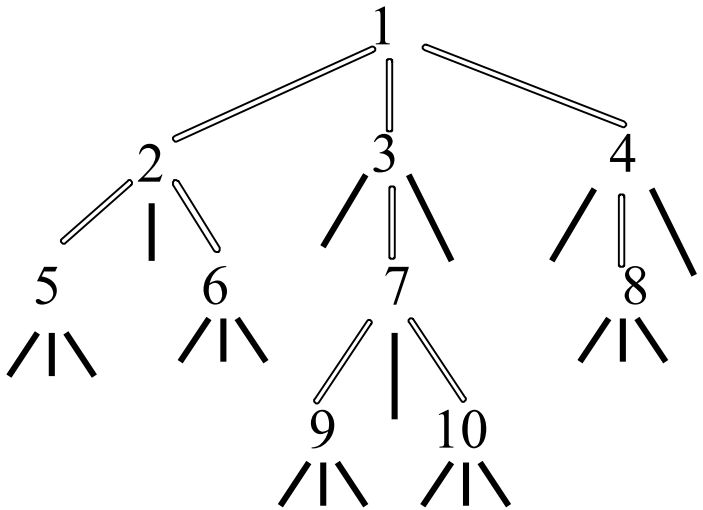
Arbre k-aire : tout nœud possède *k* sous-arbres (vides ou non), *k* fixé

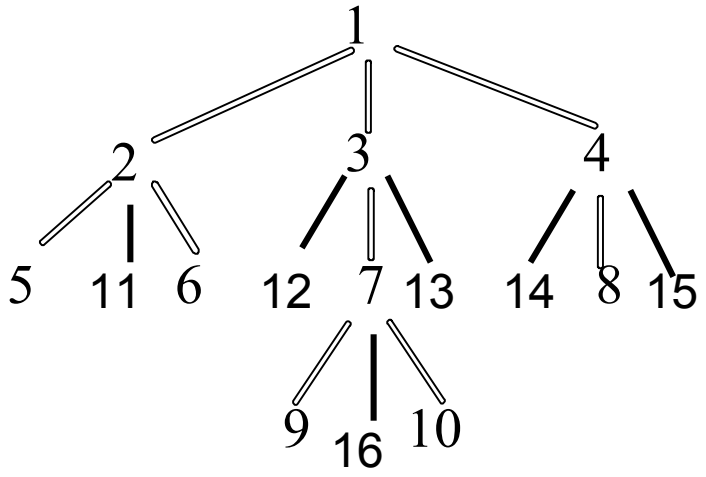
A = - Λ arbre vide ou
- $(r, A_1, ..., A_k)$

r élément, $A_1, ..., A_k$ arbres k-aires

Nœuds (A) = $\{r\} \cup$ ($\cup$ Nœuds (A_i))

Arbre k-aire complet : tout nœud interne possède *k* enfants





Arbres binaires

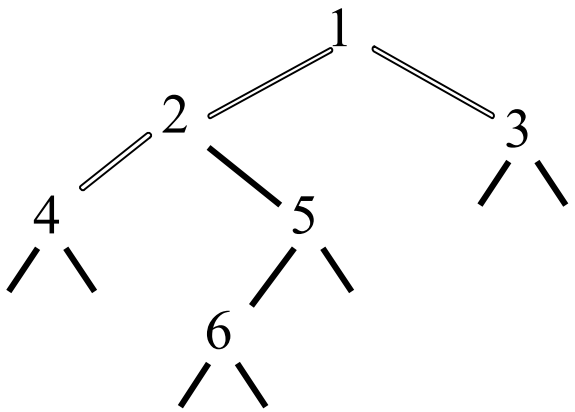
Arbre binaire : tout nœud possède deux sous-arbres (vides ou non)

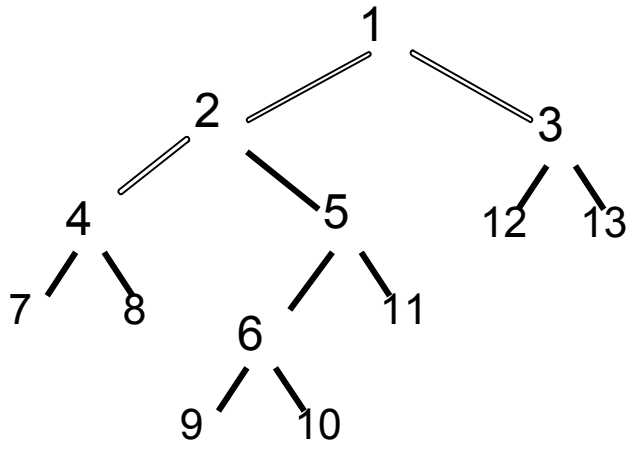
A = - Λ arbre vide ou
- (r, G, D)

r élément, G, D arbres binaires

Nœuds (A) = {r} ∪ Nœuds (G) ∪ Nœuds (D)

Arbre binaire complet : tout nœud interne possède deux enfants



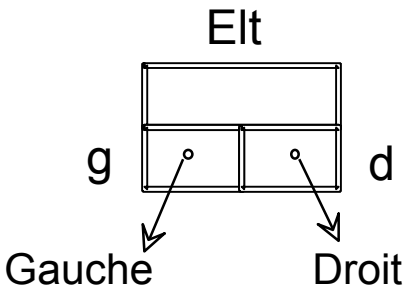


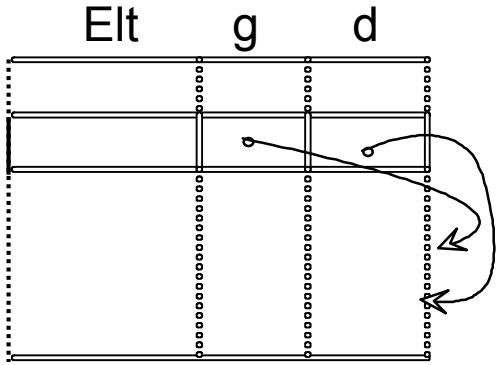
Définition des opérations

Opérations

arbre_vide	: → Arbre
racine	: Arbre → Nœud
gauche	: Arbre → Arbre
droit	: Arbre → Arbre
cons	: Nœud x Arbre x Arbre → Arbre
élément	: Nœud → Elément
est_vide	: Arbre → Booléen

Implantations par pointeurs ou curseurs





Quelques propriétés

Arbre binaire complet : nombre de feuilles = nombre de nœuds internes + 1

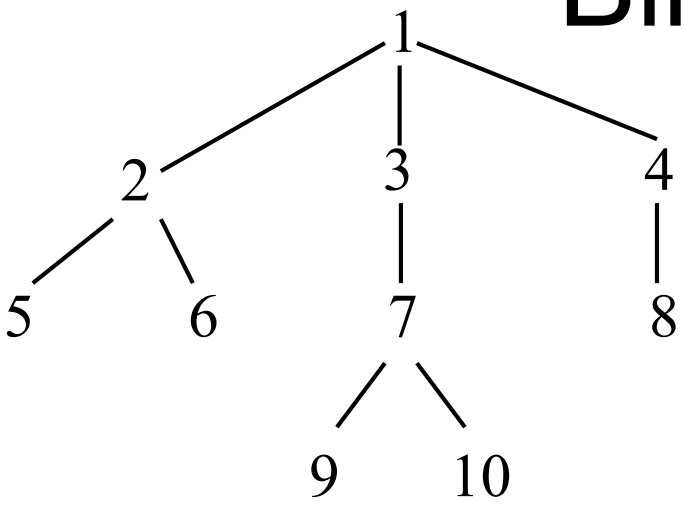
Récurrence sur le nombre de nœuds de l ’arbre *A* :

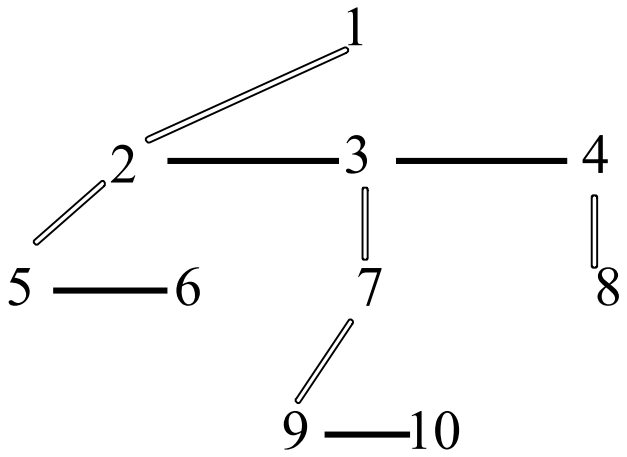
- si zéro nœud interne, c’est une feuille : propriété satisfaite.

- on suppose la propriété vraie pour les arbres ayant moins de n nœuds
  soit B = (r, B1, B2) ayant n nœuds internes
  on a n = 1 + n1 + n2
  par récurrence, nf1 = n1 + 1 et nf2 = n2 + 1
  les feuilles de B sont les feuilles de B1 et B2, donc
  nf = (n1 + 1) + (n2 + 1) = n + 1

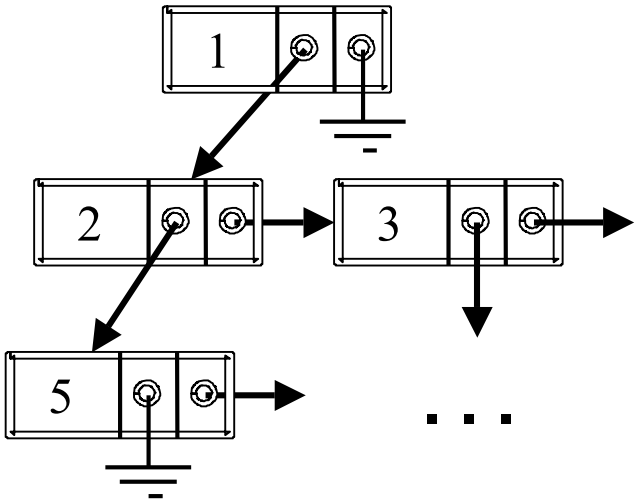
Un arbre binaire complet a 2.*n* + 1 nœuds

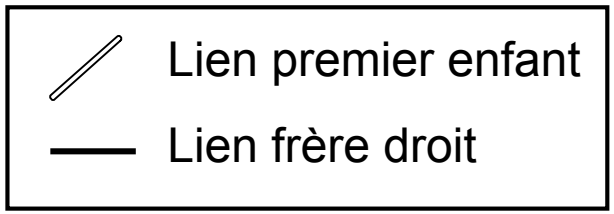
Binarisation





Implémentation des arbres planaires





A quoi ça sert ?

Application aux ensembles

Opérations sur les ensembles

Sorte Ensemble

Opérations

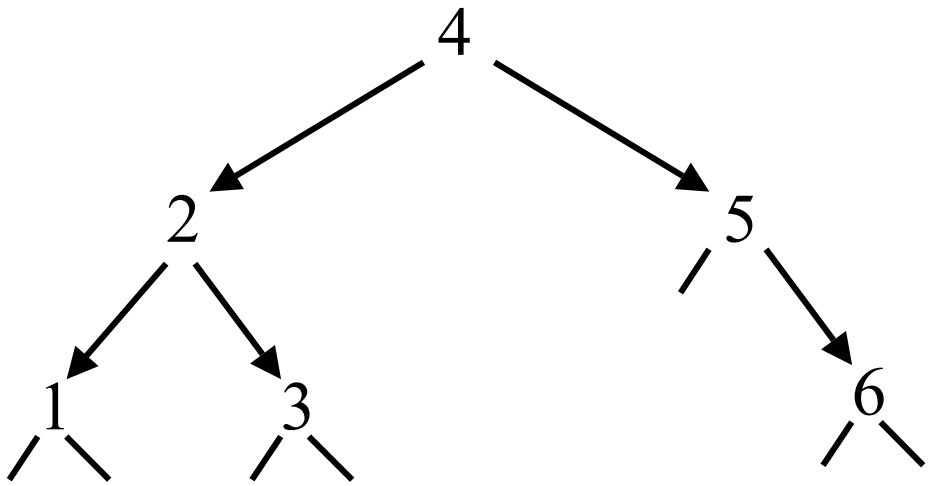
ens_vide	: → Ensemble
appartient	: Ensemble x Elément → Booléen
supprimer	: Ensemble x Elément → Ensemble
ajouter	: Ensemble x Elément → Ensemble
taille	: Ensemble → Entier
est_vide	: Ensemble → Booléen

Arbre binaire de recherche

A est un arbre binaire de recherche (ABR)

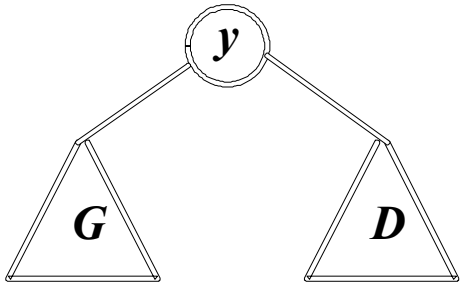
si le sous arbre gauche (resp. droit) d ’un nœud p
ne contient que des éléments inférieurs (resp. supérieurs)

conséquence : le parcours symétrique donne une liste croissante d’éléments



Recherche dans un ABR

appartient(A, x) = vrai ssi x est étiquette d'un noeud de A

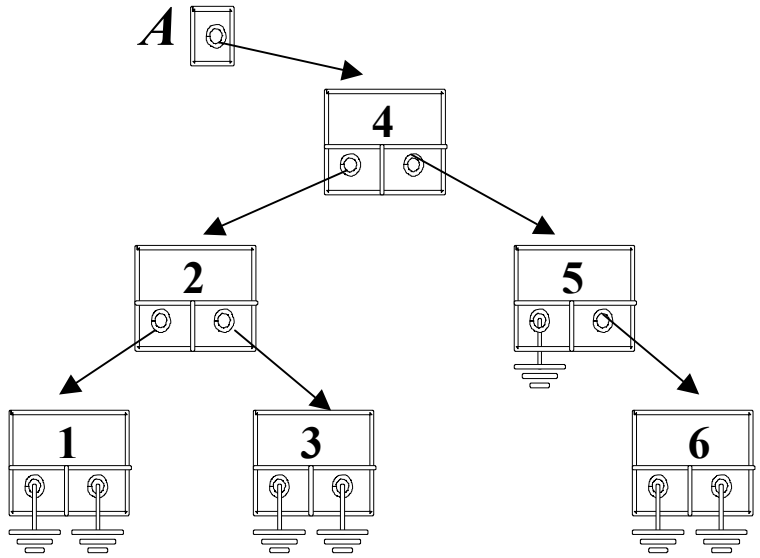


appartient(A, x) =

faux	si A vide
vrai	si $x = y$
appartient(G(A), x)	si $x < y$
appartient(D(A), x)	si $x > y$

Calcul en O(Hauteur(A))

Recherche itérative



Place (a : ABR, x : Elément) = n : Nœud

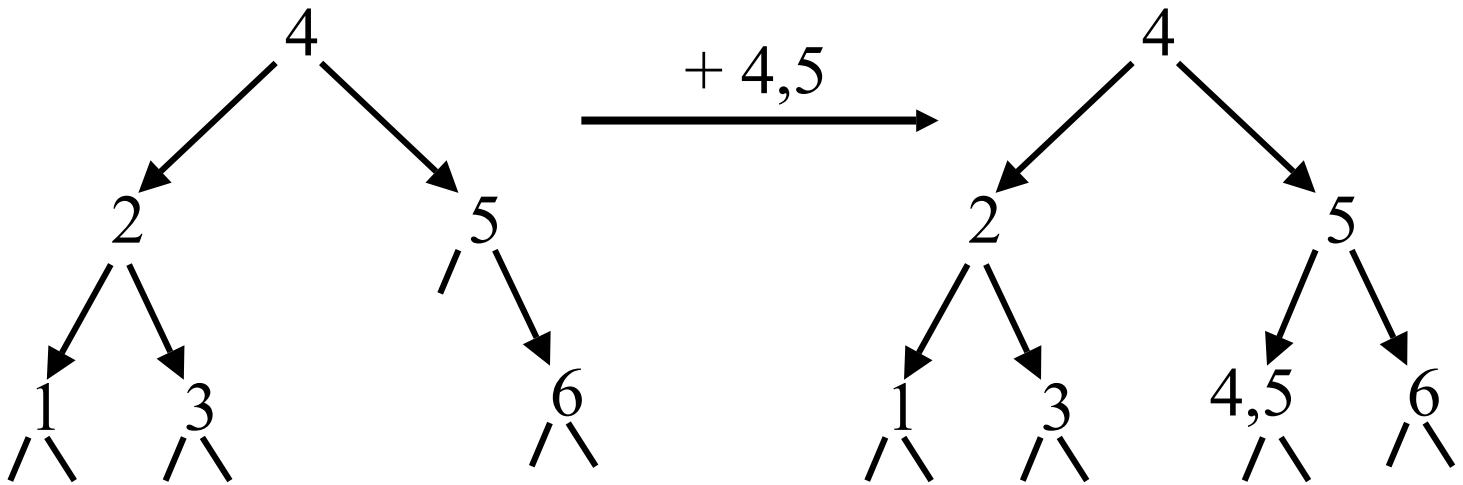
n ← a.racine

tantque n ≠ void et x ≠ n.élément faire
si x < n.élément

alors n ← n.gauche
sinon n ← n.droit

finsi
fintantque

Ajout dans un ABR

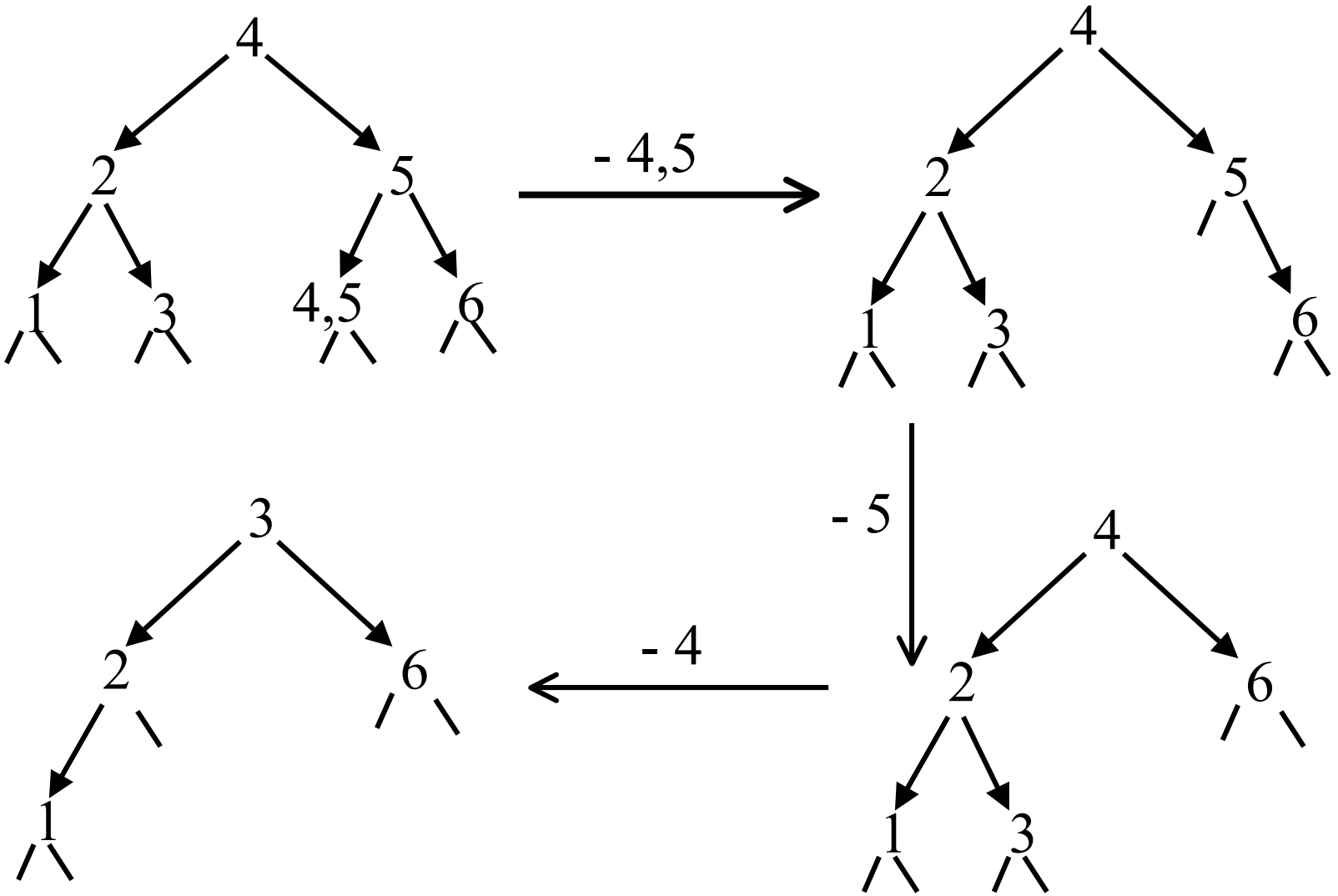


$$A + \{x\} = \begin{cases} (x, \wedge, \wedge) & \text{si } A = \wedge \text{ , arbre vide} \\ (r, G + \{x\}, D) & \text{si } x < \text{Elt}(r) \\ (r, G, D + \{x\}) & \text{si } x > \text{Elt}(r) \\ A & \text{sinon} \end{cases}$$

Ajouter

```
ajouter (a : ABR, x : Elément) = r : ABR
    si a.est_vide() alors
      r ← créer_abr(x, void, void)
    sinonsi  x < a.racine.élément alors
      r ← créer_abr(a.racine, ajouter(a.gauche,x), a.droit)
    sinonsi  x > a.racine.élément alors
      r ← créer_abr(a.racine, a.gauche, ajouter(a.droit,x))
    sinon
       r ← a
    finsi
```

Suppression dans un ABR



Suppression

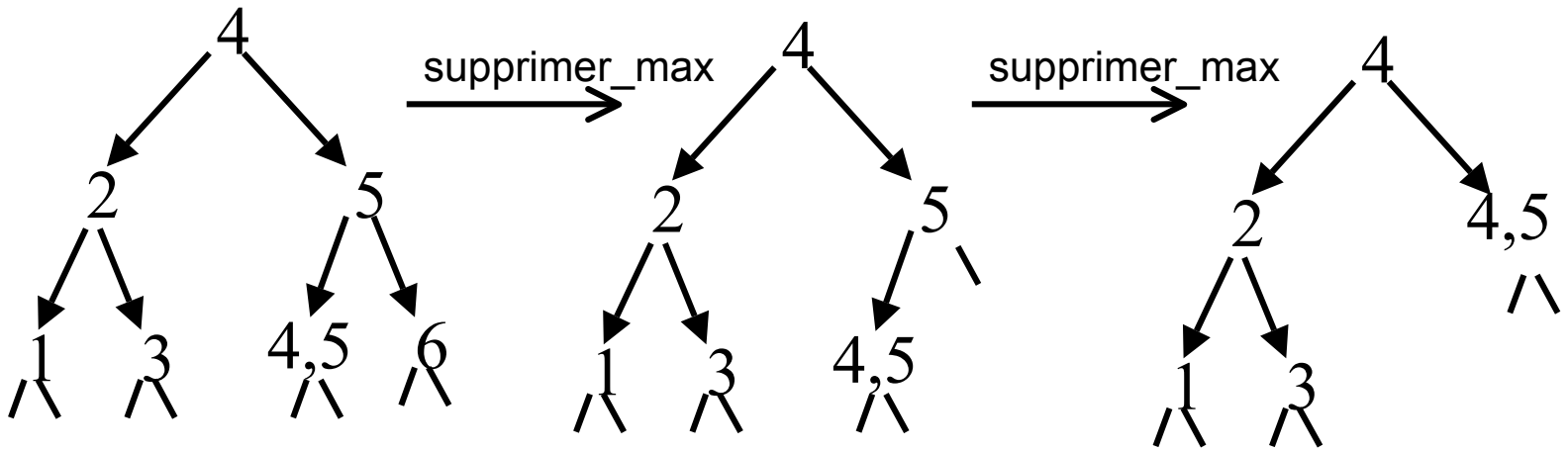
$A = (r, G, D)$

$$A - \{x\} = \begin{cases} (r, G - \{ x \}, D) & \text{si } x < \text{Elt}(r) \\ (r, G, D - \{ x \}) & \text{si } x > \text{Elt}(r) \\ D & \text{si } x = \text{Elt}(r) \text{ et } G \text{ vide} \\ G & \text{si } x = \text{Elt}(r) \text{ et } D \text{ vide} \\ (r', G - \{\text{MAX}(G)\}, D) & \text{sinon} \\ \text{avec Elt}(r') = \text{MAX}(G) & \end{cases}$$

Supprimer

```
supprimer (a : ABR, x : Elément) = r : ABR
    si a.est_vide() alors   r ← a
    sinonsi  x < a.racine.élément alors
      r ← créer_abr(a.racine, supprimer(a.gauche,x), a.droit)
    sinonsi  x > a.racine.élément alors
      r ← créer_abr(a.racine, a.gauche, supprimer(a.droit,x))
    sinonsi a.gauche.est_vide() alors r ← a.droit
    sinonsi a.droit.est_vide() alors r ← a.gauche
    sinon
      a.racine.setElément( max(a.gauche) )
      r ← créer_abr(a.racine, supprimer_max(a.gauche), a.droit)
    finsi
```

Supprimer l'élément maximum



supprimer_max (a : ABR) = r : ABR

si a.droit.est_vide() alors

r ← a.gauche

sinon

r ← créer_abr(a.racine, a.gauche, supprimer_max(a.droit))

finsi

Temps d'exécution

opérations sur ensemble

implémentation	Ens_vide	Ajouter Enlever	Élément	Min
table	cst	$O(1)^*$	$O(n)$	$O(n)$
table triée	cst	$O(n)$	$O(\log n)$	$O(1)$
liste chaînée	cst	$O(1)^*$	$O(n)$	$O(n)$
arbre équilibré	cst	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre	cst	$O(\log n)$	$O(\log n)$	$O(\log n)$
table hachée	$O(B)$	cst	cst	$O(B)$

en moyenne

n nombre d 'éléments, $B > n$

**sans le test d 'appartenance*

Type Graphe

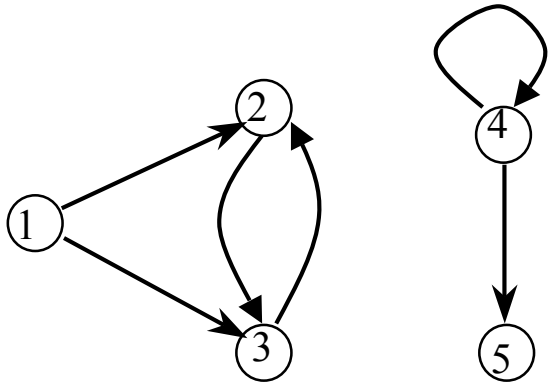
Présentation

Graphe (orienté) $G = (S, A)$

S ensemble fini des sommets

$A \subseteq S \times S$ ensemble des arcs,

i.e., relation sur S



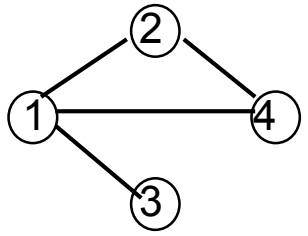
$S = \{ 1, 2, 3, 4, 5 \}$

$A = \{ (1, 2), (1, 3), (2, 3), (3, 2), (4, 4), (4, 5) \}$

Graphe non orienté $G = (S, A)$

A ensemble des arêtes,

relation symétrique

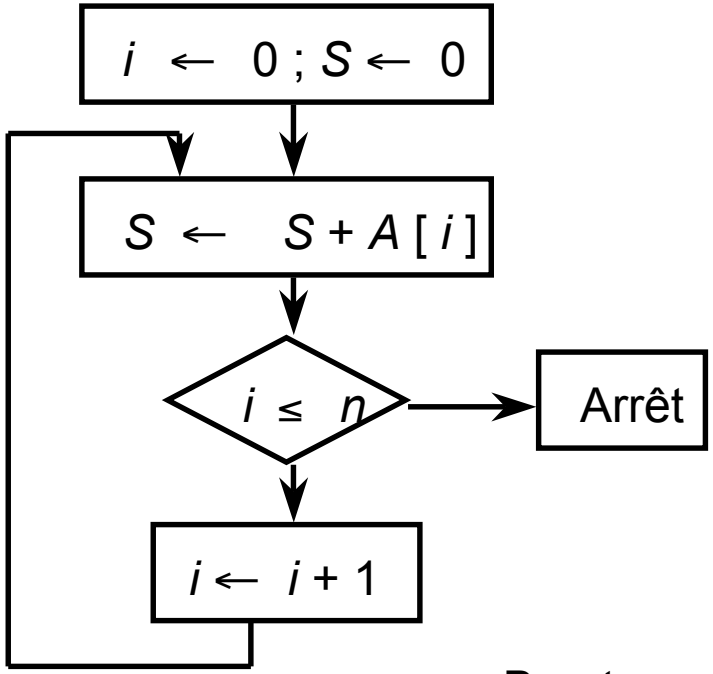


$S = \{ 1, 2, 3, 4 \}$

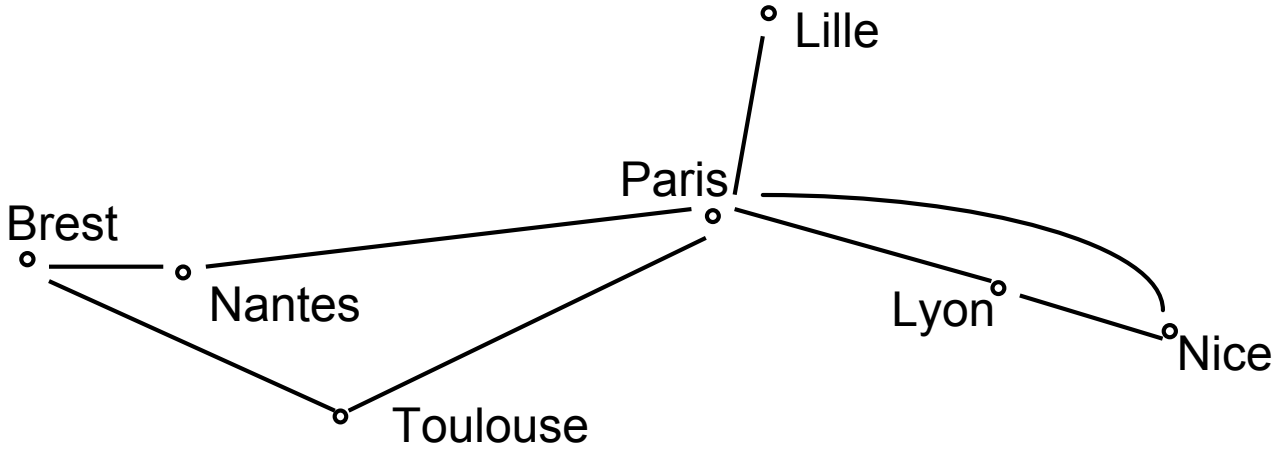
$A = \{ \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\} \}$

Applications

Flot de contrôle d'un programme

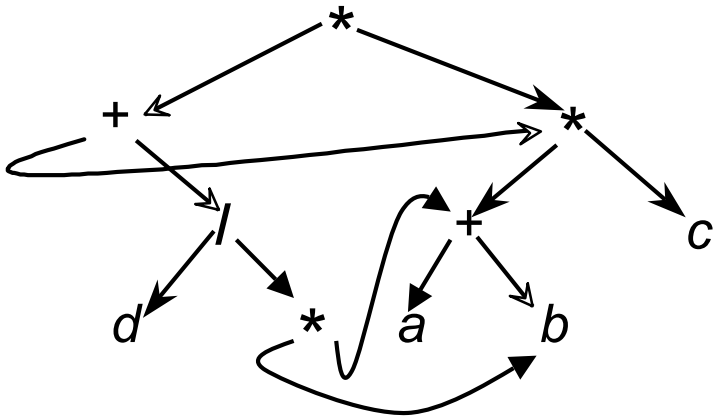


Réseau



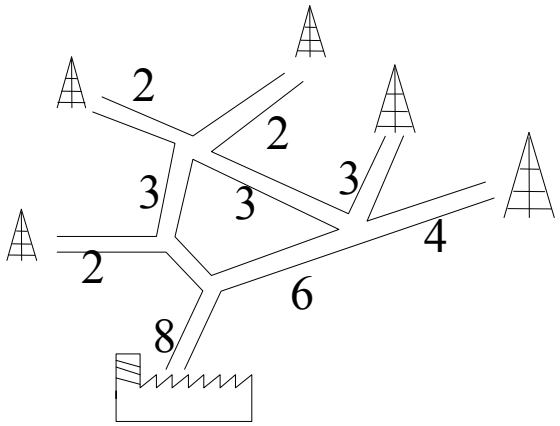
Applications

Graphe acyclique d'une expression (DAG)



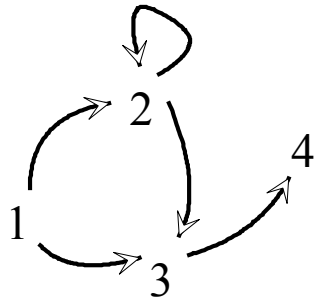
((a+b)*c+d/(b*(a+b))) * (a+b)*c

Pipelines



Terminologie

Graphe : $G = (S, A)$
Arc : $(s, t) \in A$ t adjacent à s, t successeur de s
Successeurs de s : $A(s) = \{ t \mid (s, t) \in A \}$
Boucle : $(t, t) \in A$



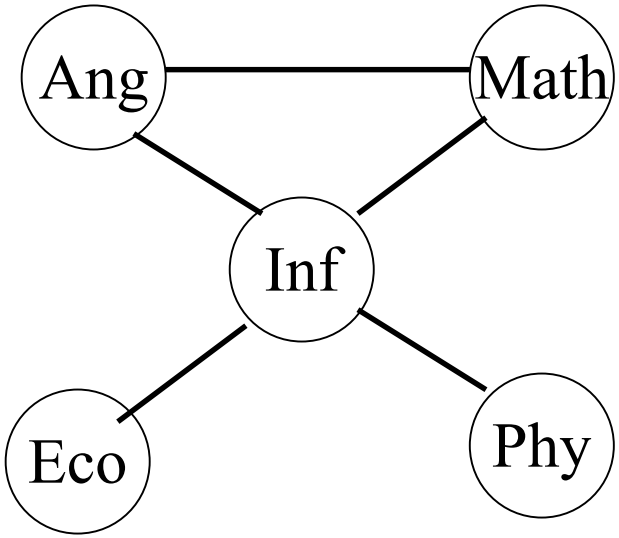
Chemins
Chemin : $c = ((s_0,s_1), (s_1,s_2), \ldots, (s_{k-1},s_k))$ où les $(s_{i-1},s_i) \in A$
origine = s_0
extrémité = s_k ((1,2), (2,2), (2,3), (3,4))
longueur = k
Circuit : chemin dont origine et extrémité coïncident

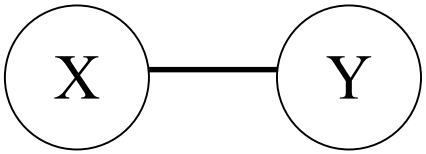
sommet, nœud = *vertex (vertices)*, arc = *arc*,
orienté = *directed*, arête = *edge*,
chemin = *path,* circuit = *circuit*

Problème d'emploi du temps

- 5 cours différents
- Comment planifier les cours pour permettre aux élèves de suivre tous les cours ?

Graphe pour la modélisation
d 'un problème





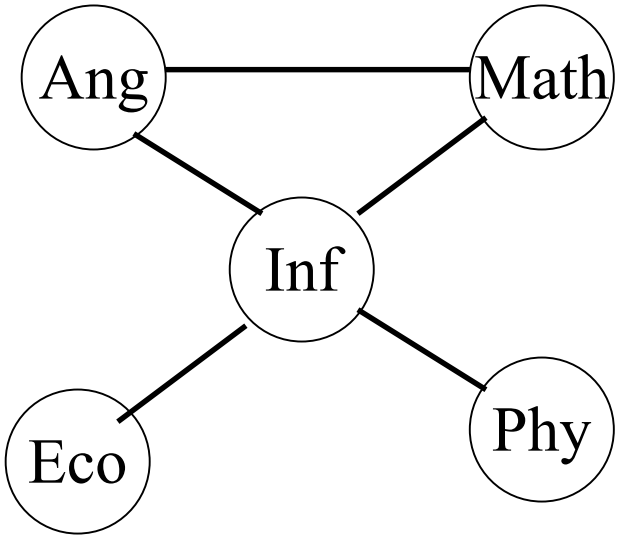
au moins 1 élève inscrit à X et Y
X et Y sont incompatibles

Problème de coloriage

$G = (S,A)$
coloration $f : S \rightarrow C$ telle que $(s,t) \in A \Rightarrow f(s) \neq f(t)$

nombre chromatique de *G :*
nombre minimum de couleurs nécessaires

Chr(*G*) = 3



1 : Ang, Eco, Phy
2 : Inf
3 : Math

Algorithme de coloriage

$G = (S, A)$ $S = \{ s_1, s_2, \ldots, s_n \}$
G sans boucle !

coloration-séquentielle (g : graphe) = nb : Entier
pour i de 1 à n faire
c ← 1
tantque il existe t adjacent à s_i avec coul(t) = c faire
c ← c + 1
fintantque
s_i.setCoul(c)
finpour
nb = max (coul(s_i), i = 1, …, n)

Temps d 'exécution : $O(n^2)$ Calcul de Chr (G) : $O(n^2\, n!)$
(appliquer la fonction à toutes les permutations de S)
Aucun algorithme polynomial connu !

Opérations sur les sommets

Sorte Sommet

Opérations

créer_sommet : Entier → Sommet

arc? : Sommet x Sommet → Booléen

numéro : Sommet → Entier

degré : Sommet → Entier

ième_succ : Sommet x Entier → Sommet

Opérations sur les graphes

Sorte Graphe

Opérations

nb_sommet : Graphe → Entier

créer_graphe: Graphe

ajout_sommet: Graphe x Sommet → Graphe

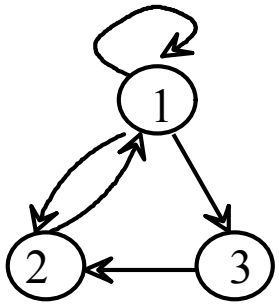
ajout_arc : Graphe x Sommet x Sommet → Graphe

degré : Graphe x Sommet → Entier

arc? : Graphe x Sommet x Sommet → Booléen

ième_succ : Graphe x Sommet x Entier → Sommet

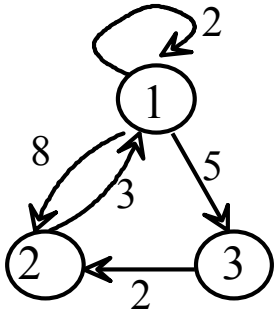
Matrices d'adjacence



S = { 1, 2, 3 }
A = { (1,1), (1, 2), (1, 3), (2, 1), (3, 2)}

$$M = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix}$$

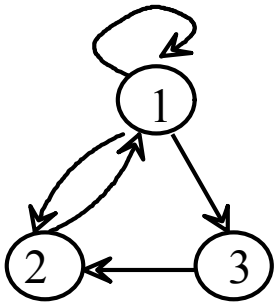
M [i, j] = 1 ssi j adjacent à i



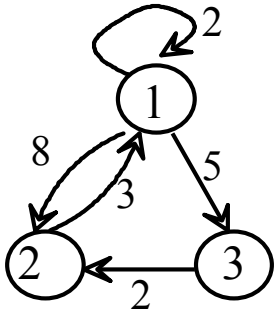
$$V = \begin{vmatrix} 2 & 8 & 5 \\ 3 & 0 & 0 \\ 0 & 2 & 0 \end{vmatrix}$$

Valuation : $v : A \longrightarrow X$

Listes de successeurs



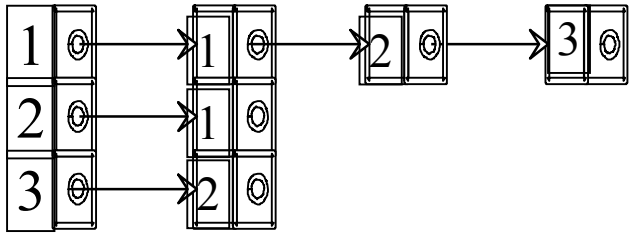
Listes des $A(s)$

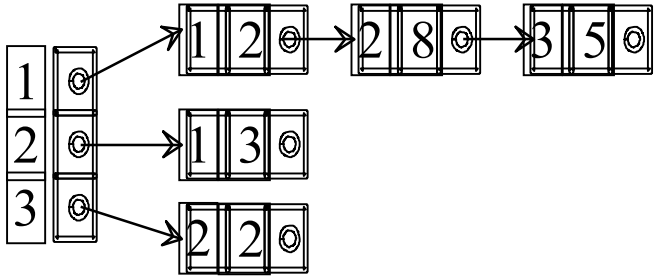


Valuation : $v : A \longrightarrow X$

$S = \{ 1, 2, 3 \}$

$A = \{ (1,1), (1, 2), (1, 3), (2, 1), (3, 2)\}$





Comparaison

- arc (u,v) ?
  - matrice : regarder arcs[u,v]
  - liste : trouver la liste de u et la parcourir
- examiner tous les successeurs de u ?
  - matrice : regarder toutes les cases arcs[u,i]
  - liste : trouver la liste de u
- examiner tous les prédécesseurs de v ?
  - matrice : regarder la colonne de v
  - liste : regarder les successeurs de tous les u

Exploration

$G = (S, A)$
Explorer G = visite de tous les sommets
et de tous les arcs

Algorithme de base pour

- recherche de cycles
- recherche des composantes connexes
- actions :
  - sur les sommets (coloriage, ...)
  - sur les arcs (valuation, ...)

Parcours en profondeur ou en largeur

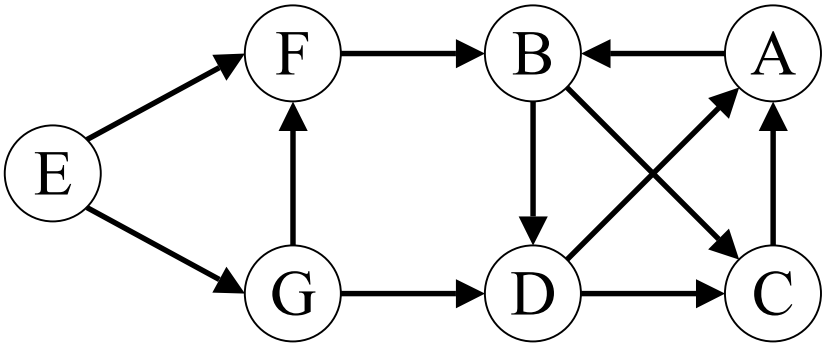
- extensions des parcours d'arbres

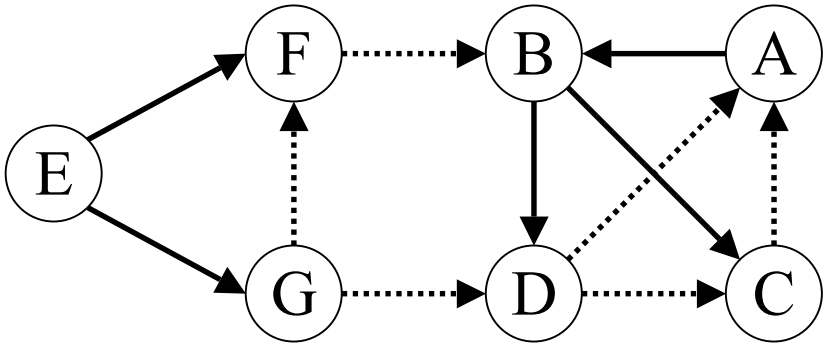
Parcours en profondeur

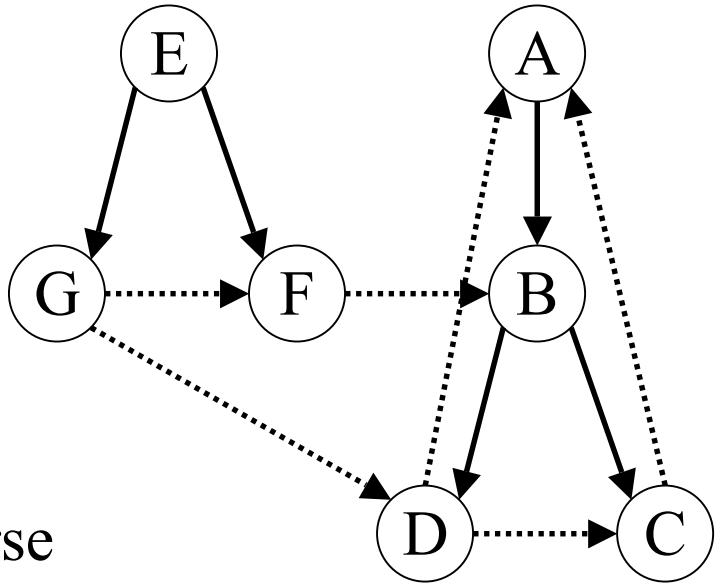
Marquage nécessaire
    pourchaque sommet de *s* de *G* faire
        visité[*s*] ← faux
    pourchaque sommet s de G faire
        si ¬ visité[*s*] alors Prof (*s*)

Prof (*s : S*ommet de *G*) =
    action préfixe sur *s*
    visité[*s*] ← vrai
    pourchaque *t* successeur de *s* faire
        action sur l'arc (*s,t*)
        si ¬ visité[*t*] alors
            Prof (*t*)
        finsi
    finpourchaque
    action suffixe sur *s*

Exemple



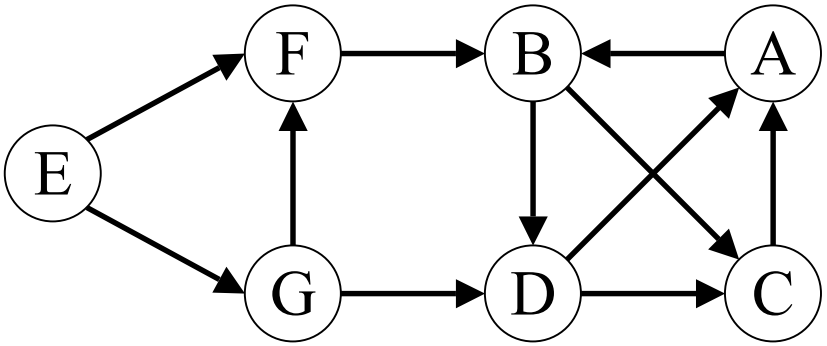




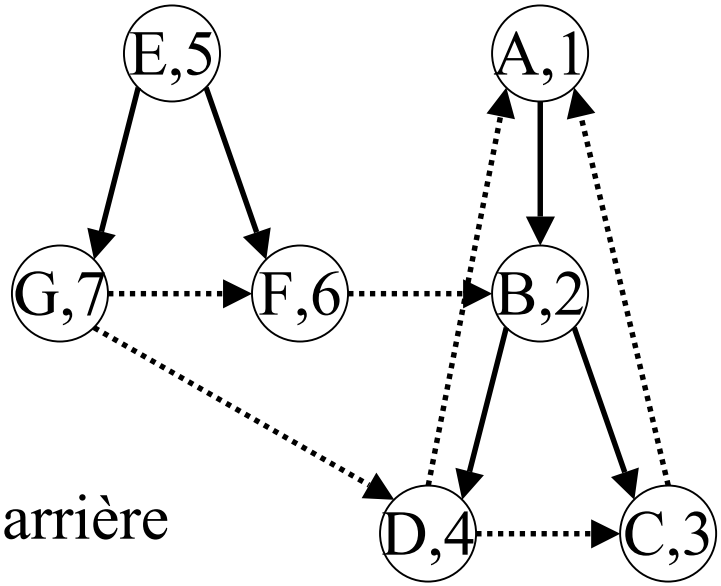
arc d'arbre

arc de traverse

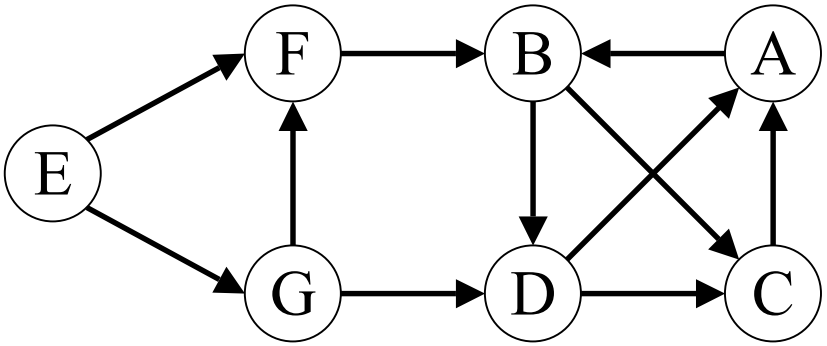
Numérotation



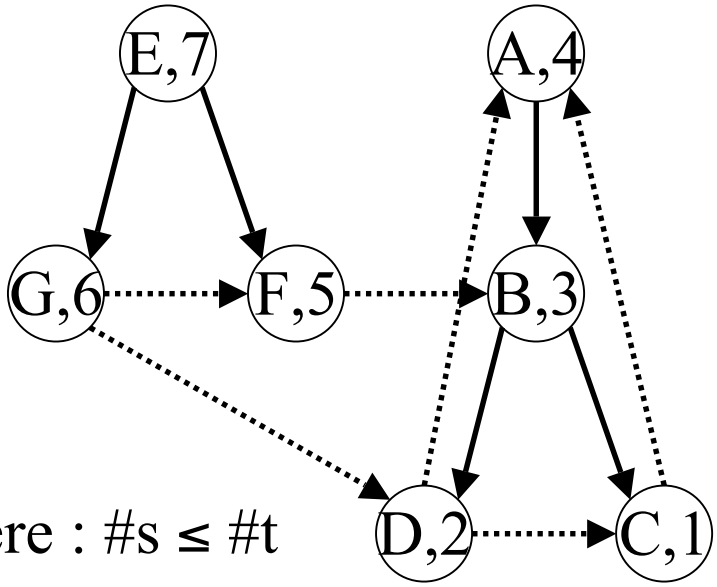
Prof (*s : S*ommet de *G*) =
ordre[s] ← compteur
compteur ← compteur + 1
visité[*s*] ← vrai
pourchaque *t* successeur de *s* faire
si ¬ visité[*t*] alors
Prof (*t*)
finsi
finpourchaque



Numérotation post-ordre




```
Prof (s : Sommet de G) =
        visité[ s ] ← vrai
        pourchaque t successeur de s faire
                si ¬ visité[ t ]  alors
                        Prof ( t )
                finsi
        finpourchaque
        ordre[ s ] ← compteur
        compteur ← compteur + 1
```



Application à la détection de circuits

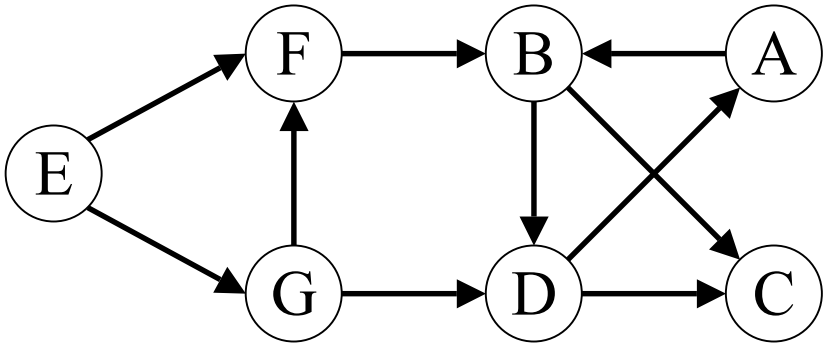
Proposition

G possède un circuit ssi il existe un arc arrière dans un parcours en profondeur de G

```
Cycle (g : Graphe) = r : Booléen
        r ← faux
        ProfForet(G)
        pourchaque sommet de s de G  faire
                pourchaque t successeur de s faire
                        si post-ordre(s) ≤ post-ordre(t)  alors
                                r ← vrai
                        finsi
                finpourchaque
        finpourchaque
```

Utilisation de 3 états

Au cours d'une exploration :
état [*s*] = blanc *s* non visité
état [*s*] = rouge *s* en cours de visite
état [*s*] = rose *s* déjà visité

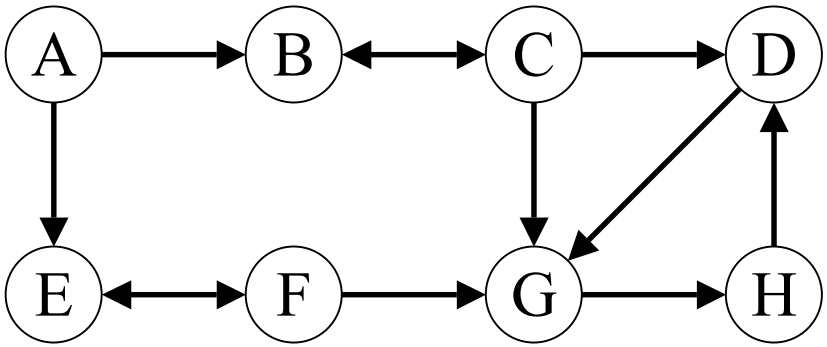


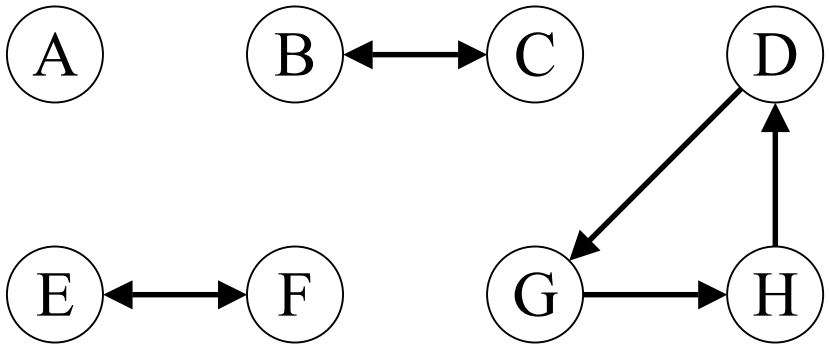
Pendant la visite du sommet D, on détecte un cycle passant par l ’arc (D, A) car A est aussi en cours de visite

Application à la détection composantes fortement connexes

G = (S, A) graphe
G' = (S',A') sous-graphe de G ssi $S' \subseteq S$ et $A' \subseteq S' \times S'$

F **composante fortement connexe** de G :
F sous-graphe maximal de G tel que
deux sommets quelconques de F sont reliés par un chemin.



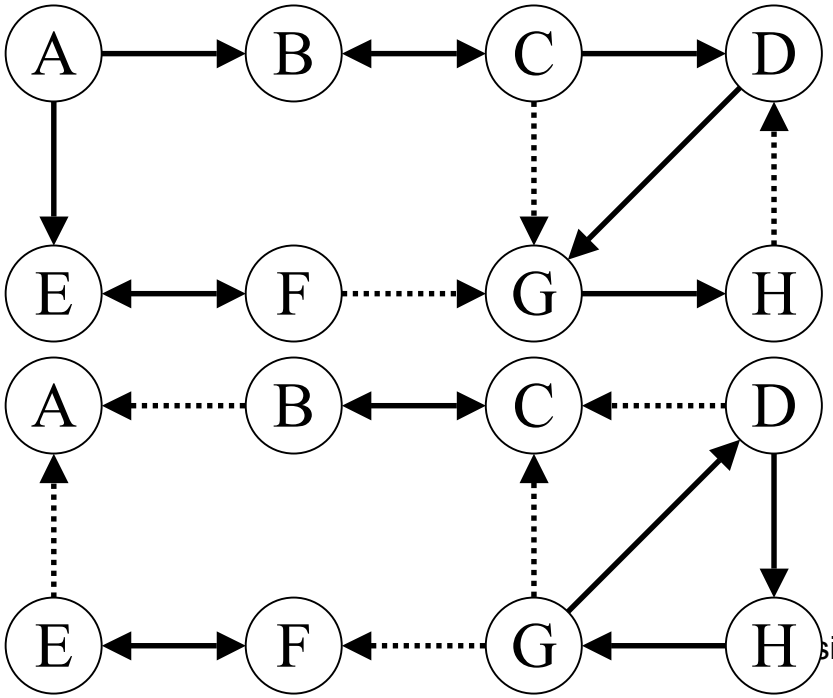


Algorithme

1. $L \leftarrow$ liste des sommets de G obtenus par parcours en profondeur suffixe

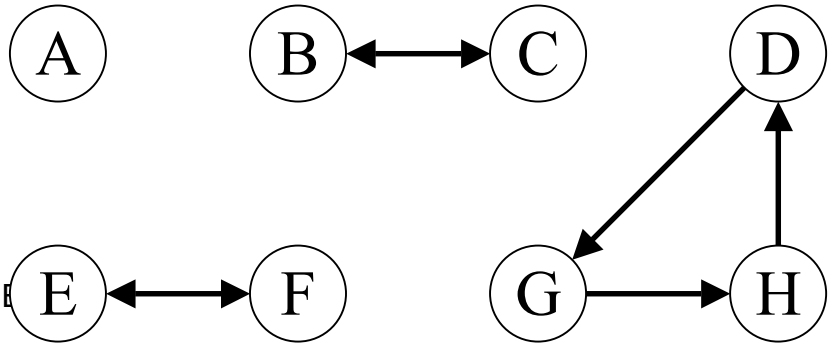
2. à partir de L^R, appliquer un parcours en profondeur à G^T
   G^T = graphe transposé de G

3. les arbres de cette exploration sont les composantes fortement connexes



L = H G D C B F E A

L^R = A E F B C D G H



The End