Arbres

Algorithmique mpci

Stéphane Grandcolas

Aix-Marseille Université

2023-2024

Arbres

Arbre : structure très utilisée en informatique

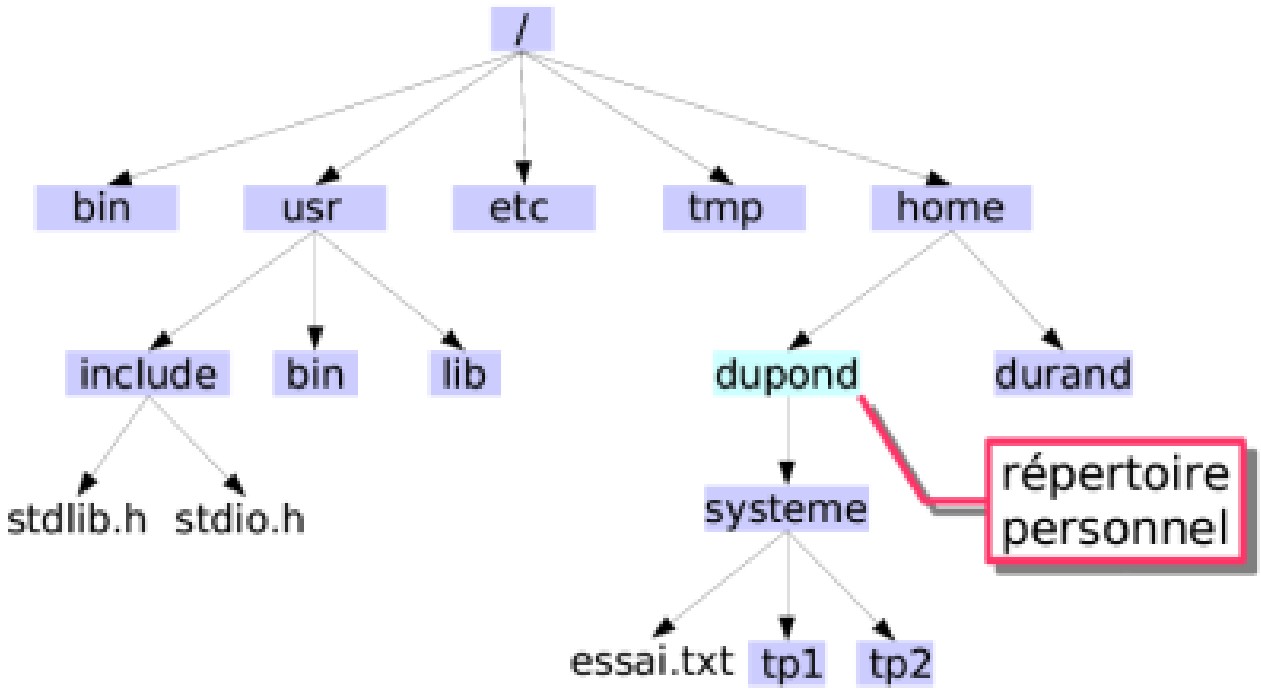
- hiérarchique
- récursive
- dynamique

Programme

- définitions, vocabulaire,
- arbres binaires,
- arbres binaires de recherche (dictionnaire),
- arbres d'arité quelconque,

Arbres

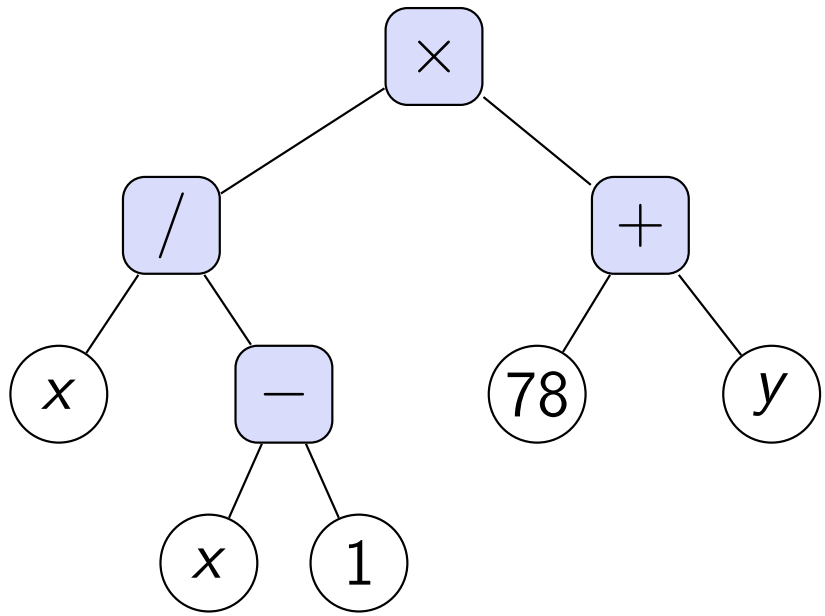
Arborescence des répertoires et fichiers (système Unix)



Arbres

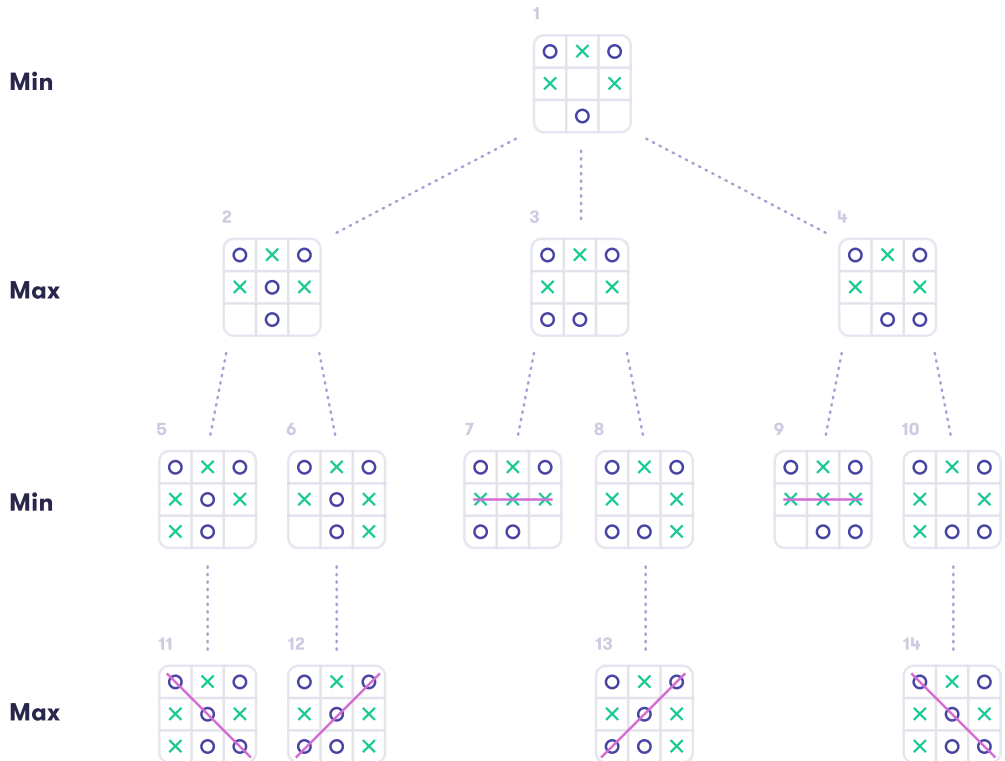
Arbre syntaxique représentant une expression arithmétique

$$\frac{x}{(x-1)} \times (78 + y)$$



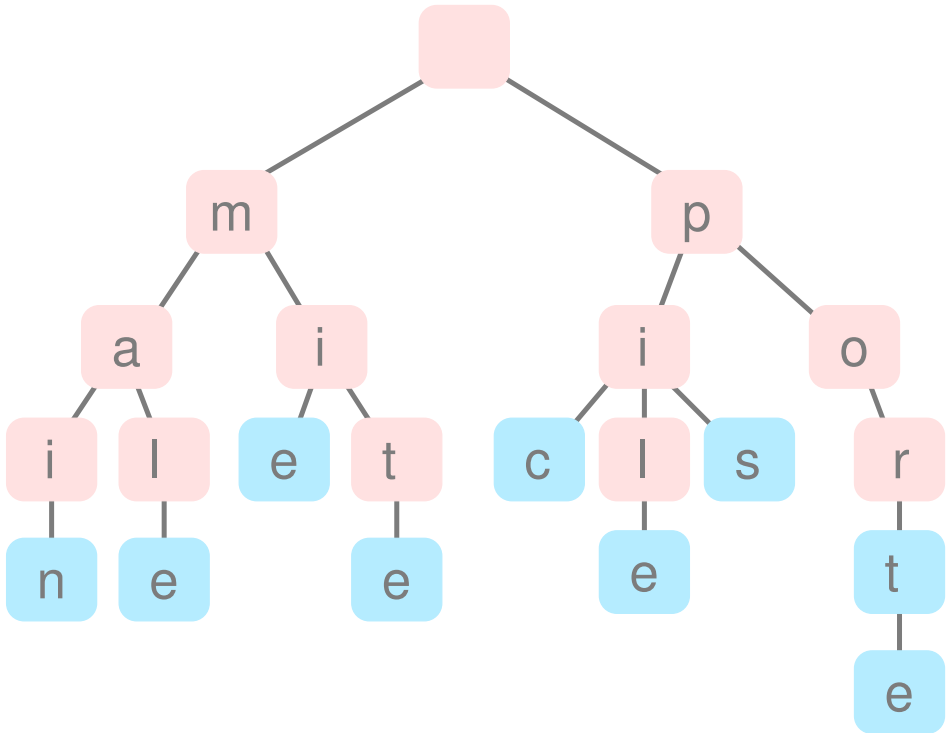
Arbres

Arbre min-max pour le calcul du meilleur coup



Arbres

Un arbre lexicographique, ou arbre de préfixes ou *trie*



main
male
mie
mite
pic
pile
pis
port
porte

Arbres : définition 1

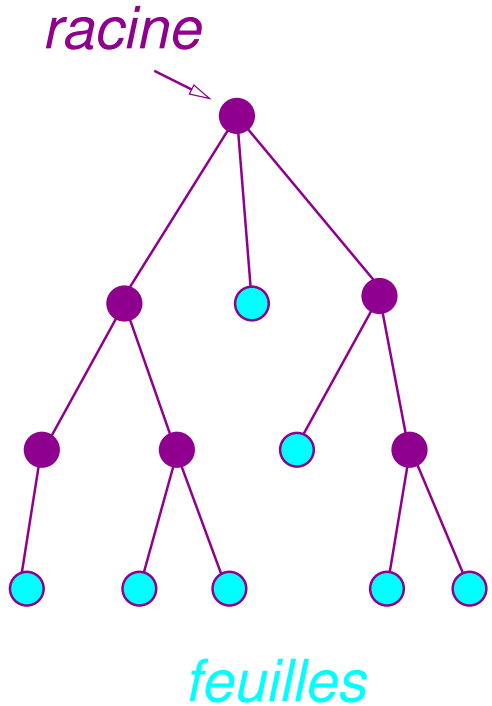
Un arbre est un ensemble organisé de noeuds :

- chaque noeud a un père et un seul,
- excepté un noeud, la racine, qui n'a pas de père.

Les fils d'un noeud p sont les noeuds dont le père est p

Les feuilles d'un arbre sont les noeuds qui n'ont pas de fils

traditionellement on représente le père au-dessus de ses fils, et donc la racine tout en haut

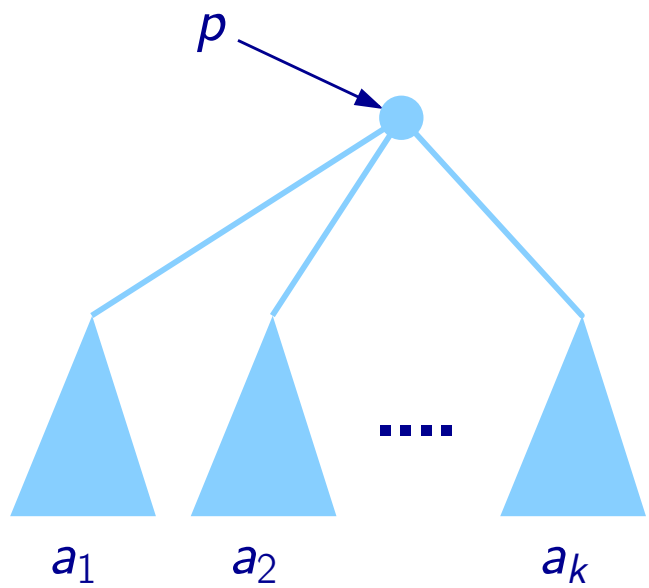


Arbres : définition 2 (récursive)

Un arbre (non vide) est constitué

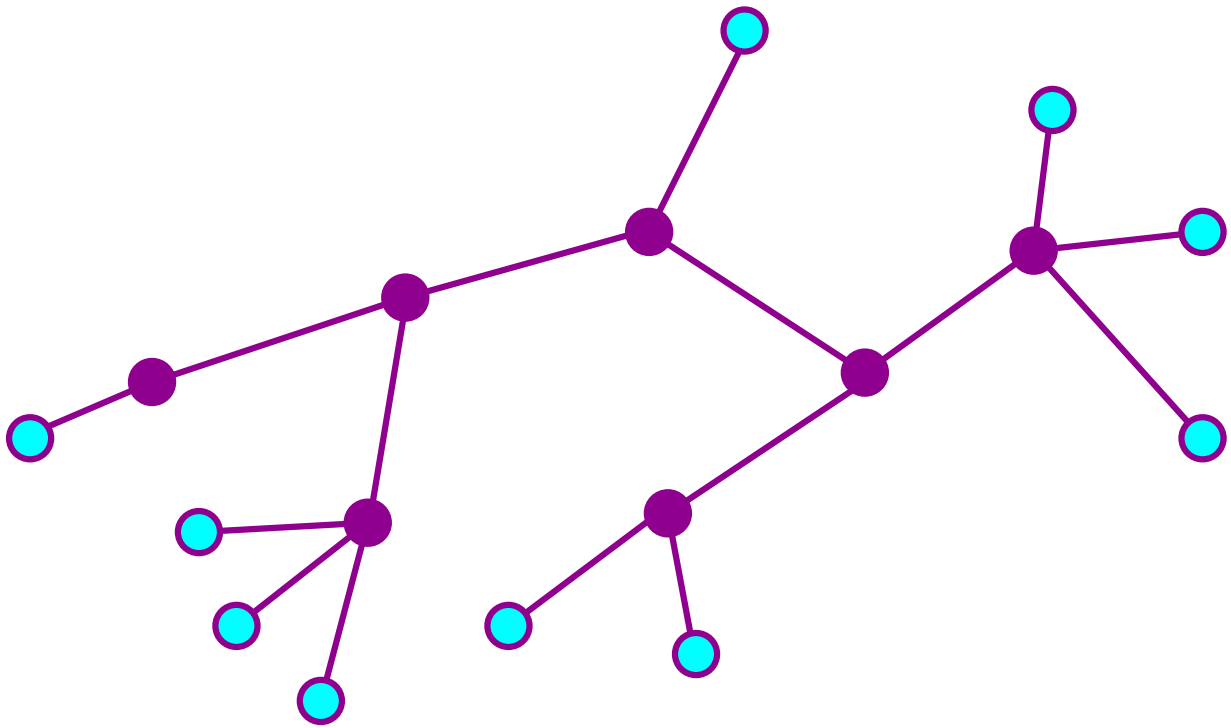
- d'un noeud p, sa racine,
- d'une suite de sous-arbres $(a_1, a_2, \ldots, a_k)$.

Les racines des arbres
$a_1, a_2, \ldots, a_k$ sont les fils de p



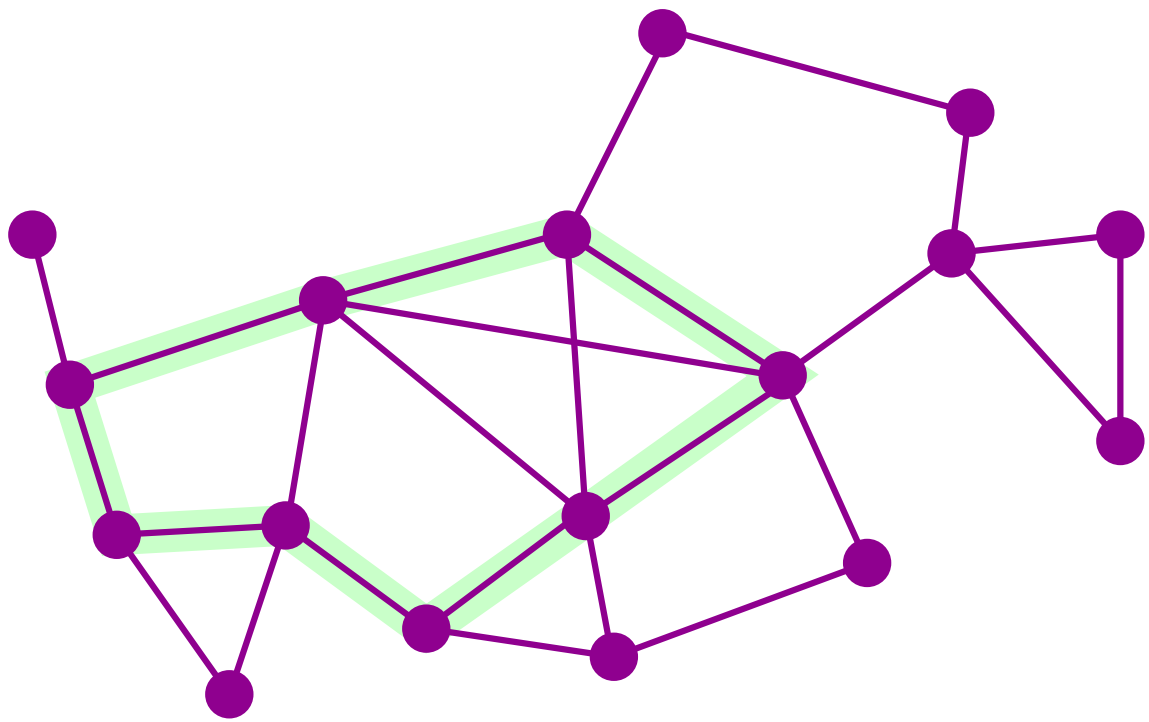
Arbres : définition 3 (graphes)

Un arbre est un graphe connexe et sans cycle.



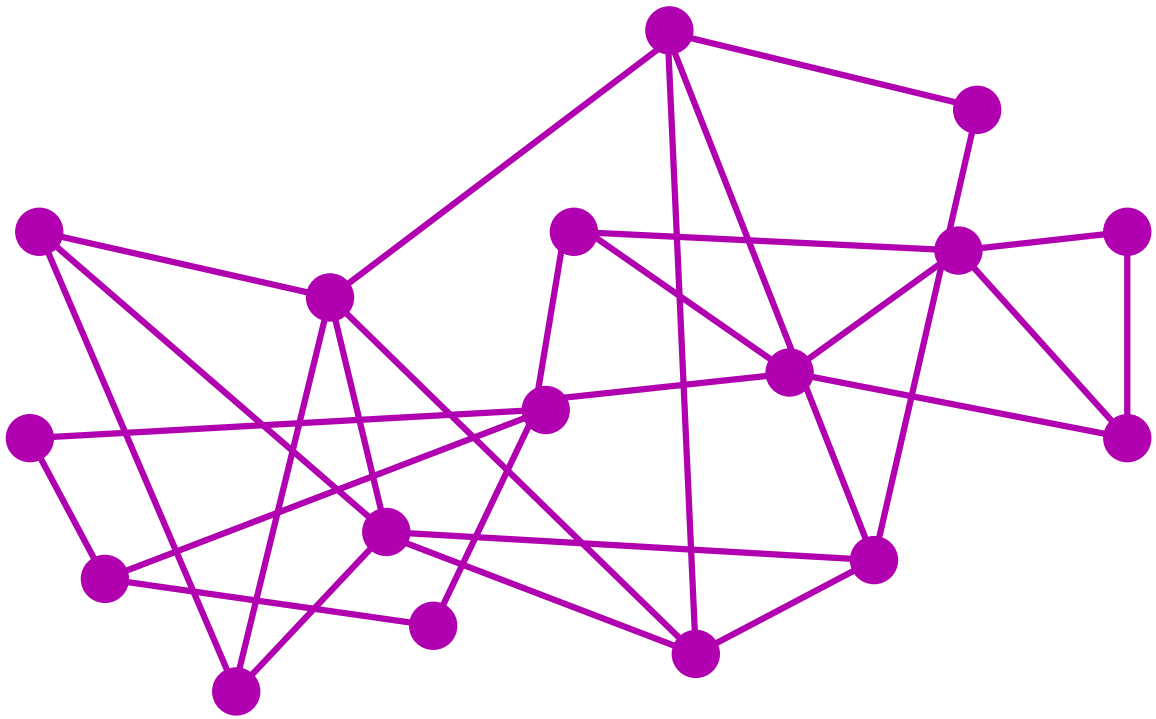
Arbres : définition 3 (graphes)

Un graphe avec des cycles



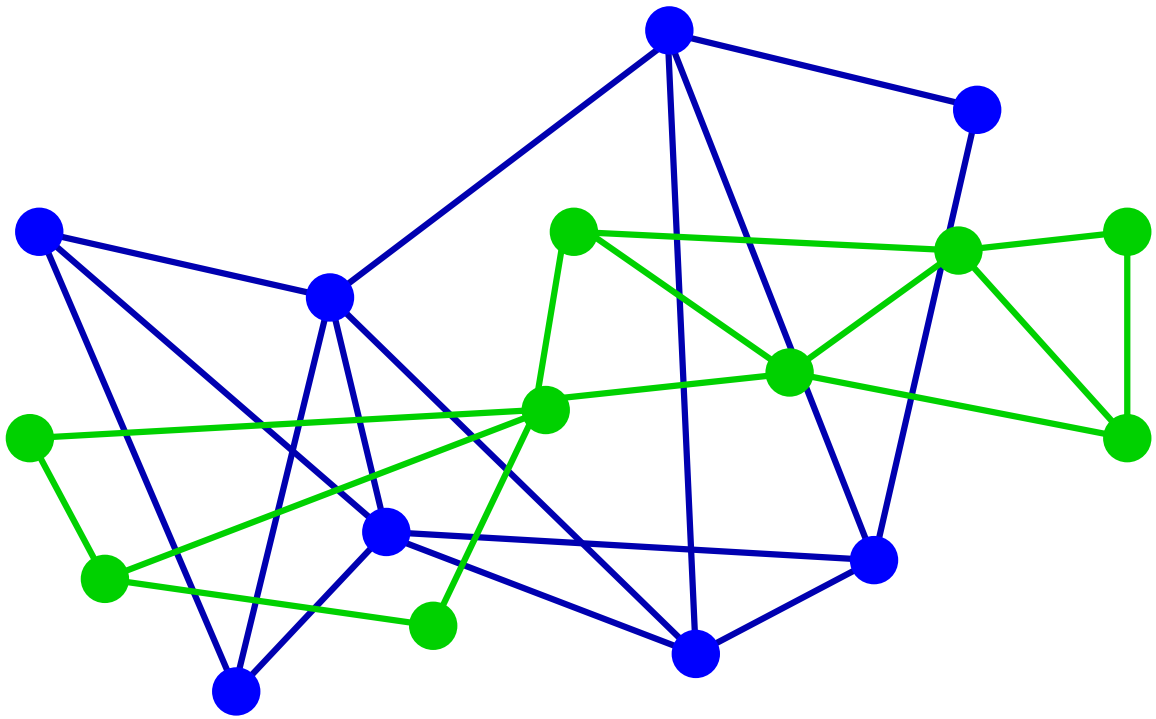
Arbres : définition 3 (graphes)

Un graphe non connexe



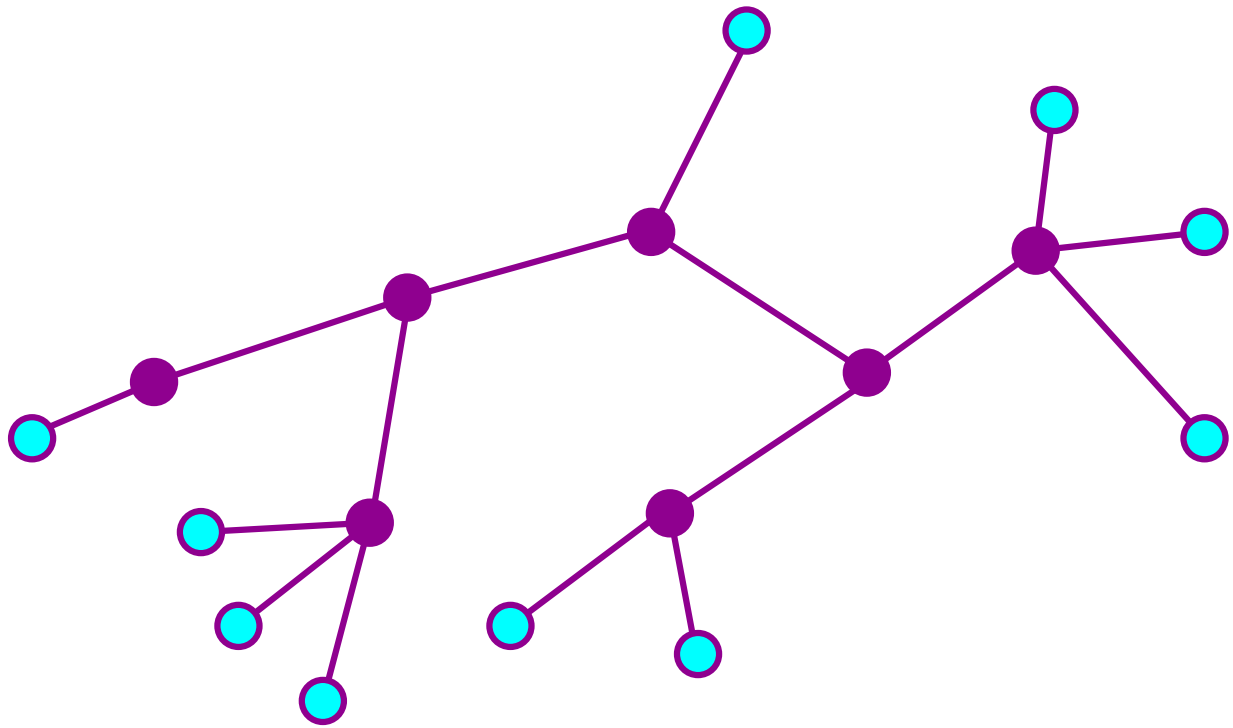
Arbres : définition 3 (graphes)

Un graphe non connexe

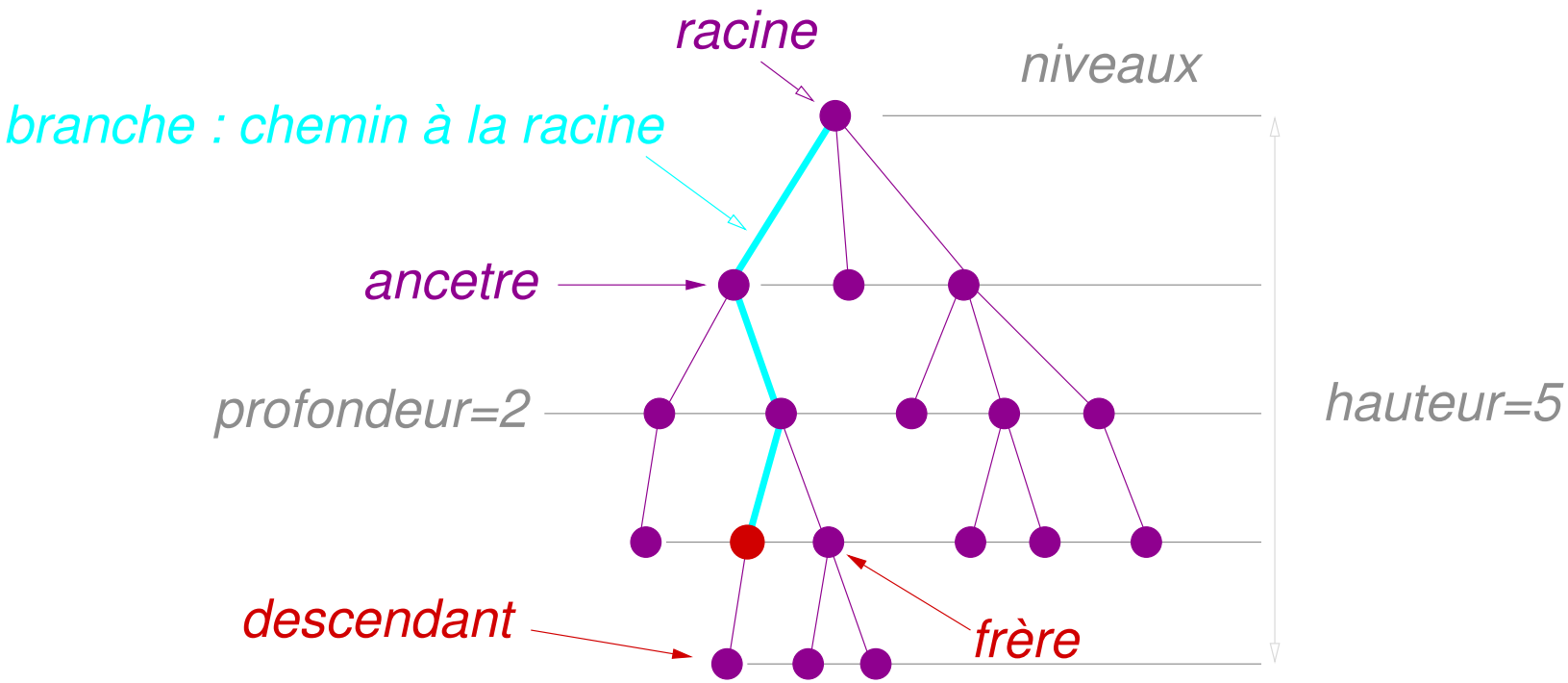


Arbres : définition 3 (graphes)

Graphe connexe et sans cycle : *nb arêtes* $=$ *nb sommets*-1

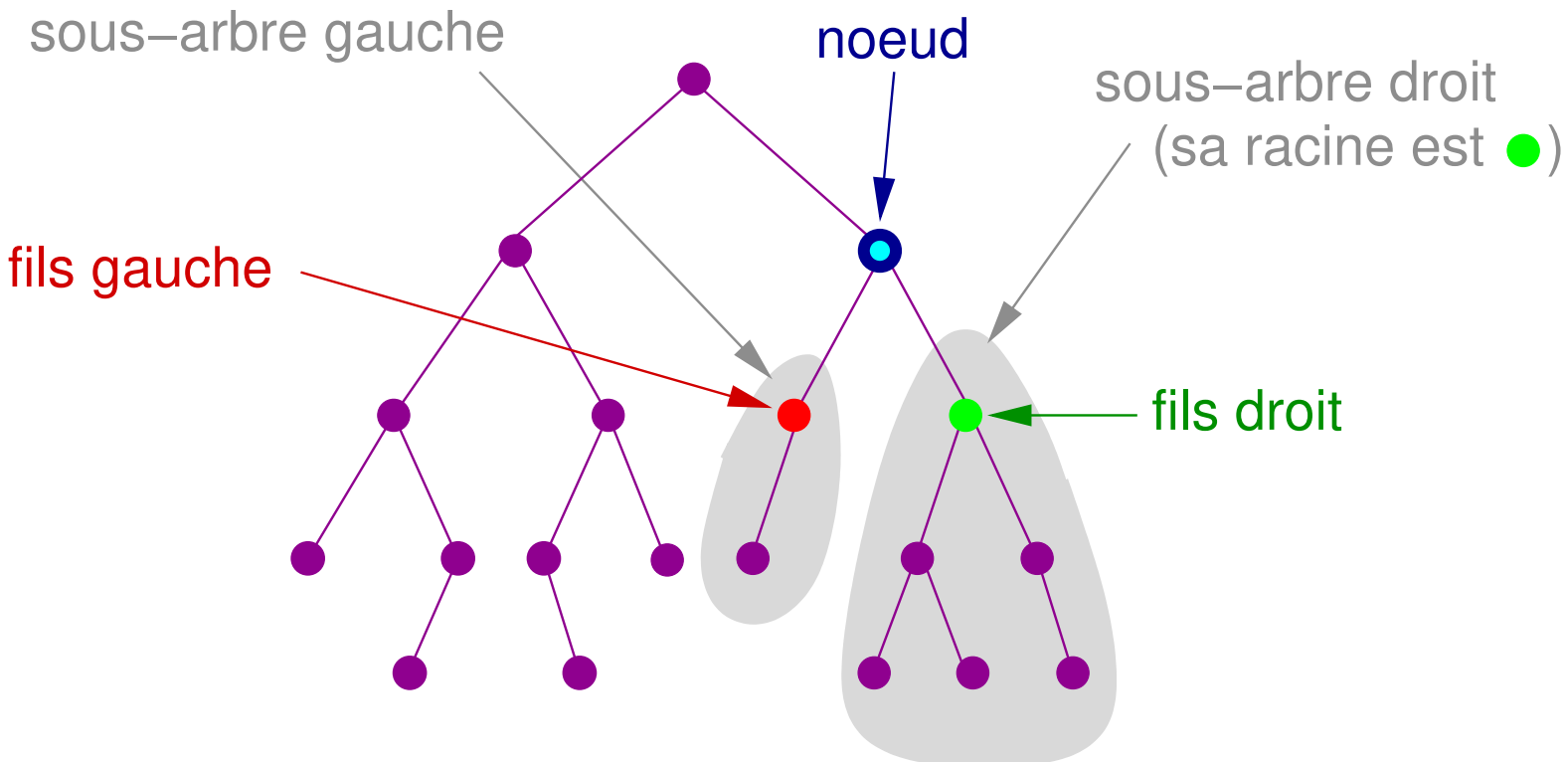


Arbres : vocabulaire

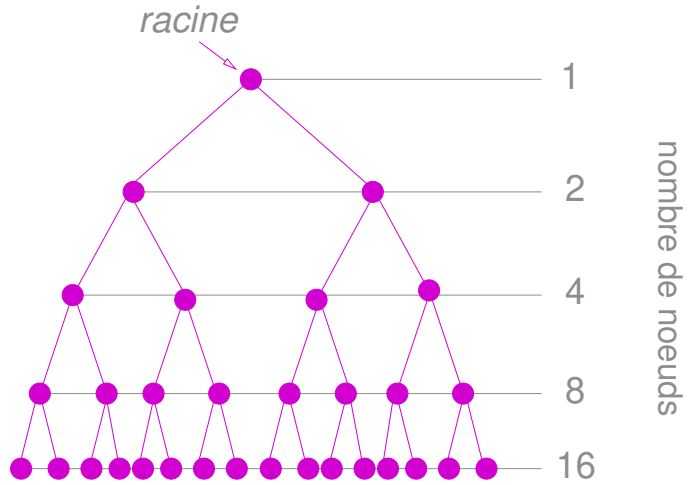


Arbres binaires

Chaque noeud a *potentiellement* un fils gauche et un fils droit



Arbres binaires



Arbre binaire parfait :

- à la profondeur p : 2^p noeuds
- nombre total de noeuds : $\sum_{i=0}^{h-1} 2^i = 2^h - 1$

Un arbre binaire de hauteur h contient au plus $2^h - 1$ noeuds

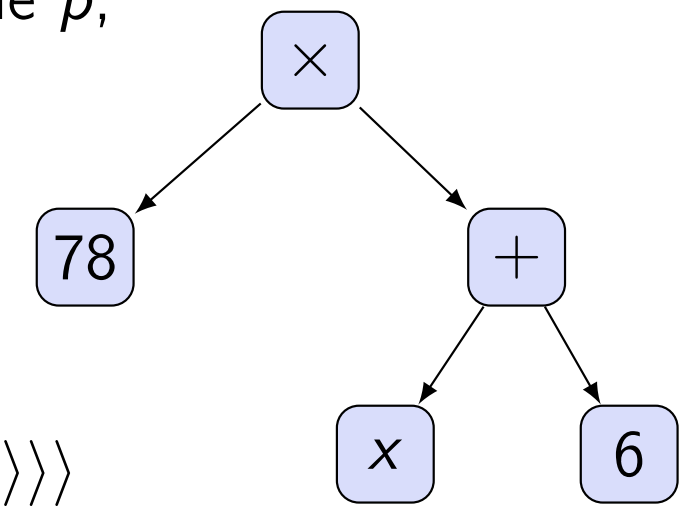
(si la hauteur est le nombre de niveaux)

Arbres binaires : représentation

Arbre vide : $\square$

Arbre non vide : $p = \langle\ x, G, D\ \rangle$

- $x = val(p)$ **:** élément, information, label, valeur,
- $G = filsG(p)$ **:** sous-arbre gauche de p,
- $D = filsD(p)$ **:** sous-arbre droit de p,



$\langle \times, \langle 78, \square, \square \rangle, \langle +, \langle x, \square, \square \rangle, \langle 6, \square, \square \rangle \rangle \rangle$

Arbres binaires : implémentation

class Noeud

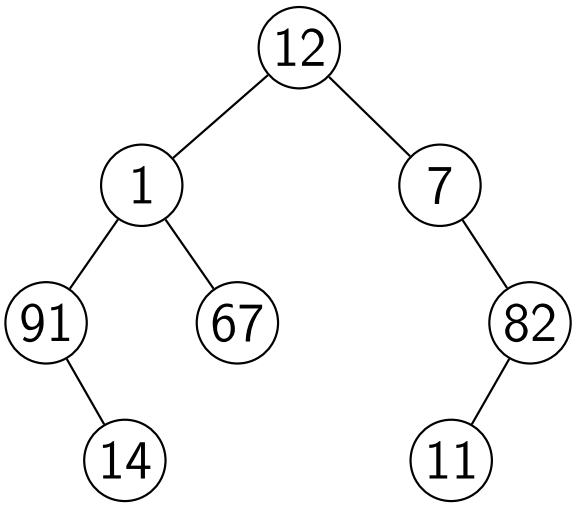
element : TYPE_ELEMENT,
filsgauche : Noeud,
filsdroit : Noeud,

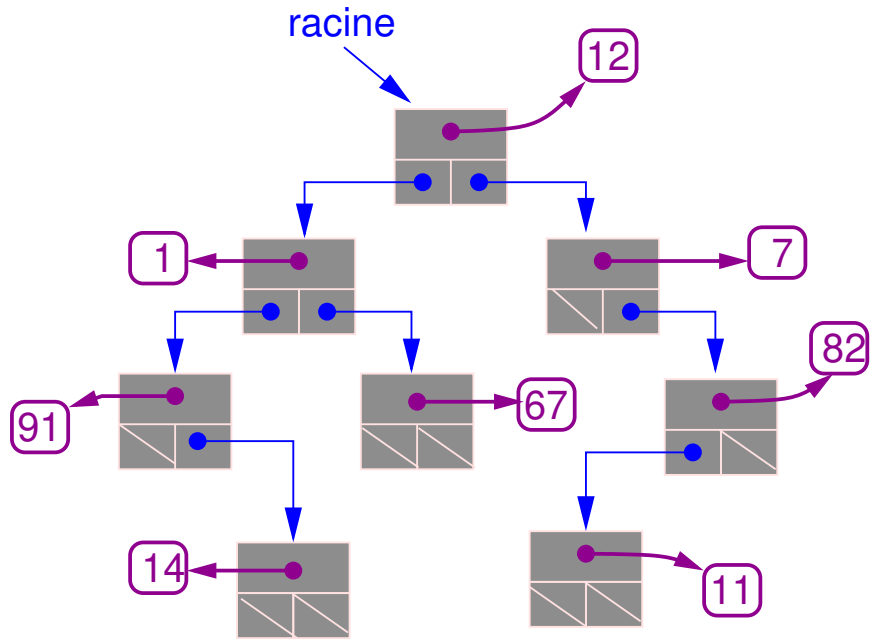
On utilisera `None` pour indiquer qu'un noeud n'existe pas

class Arbre

racine : Noeud,

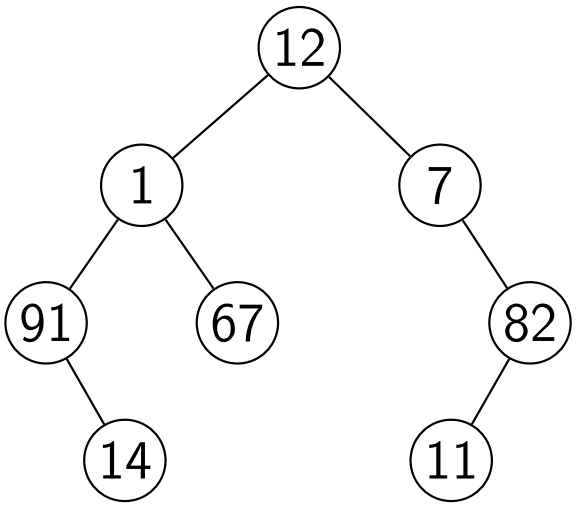
Arbres binaires : implémentation

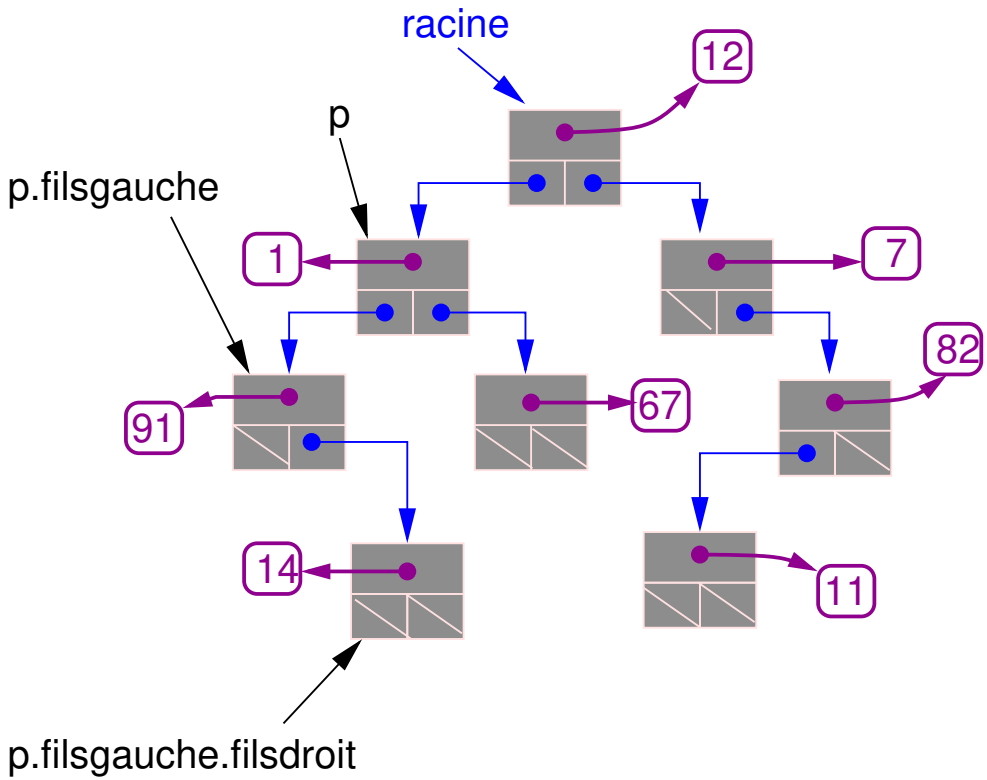




$\langle 12, \langle 1, \langle 91, \square, \langle 14, \square, \square \rangle \rangle, \langle 67, \square, \square \rangle \rangle, \langle 7, \square, \langle 82, \langle 11, \square, \square \rangle, \square \rangle \rangle \rangle$

Arbres binaires : implémentation





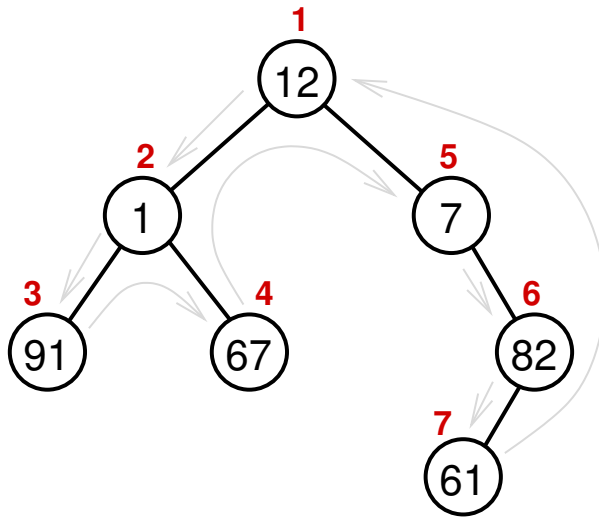
Arbres binaires : parcours en profondeur

Principe : explorer le sous-arbre gauche, puis explorer le sous-arbre droit

```
fonction EXPLORE(p),
    si p ≠ □ alors
        → ici traitement avant
        EXPLORE(filsG(p)),
        → traitement après l'exploration du fils gauche
        EXPLORE(filsD(p)),
        → ici traitement après l'exploration du fil droit
    finsi
fin fonction
```

Arbres binaires : parcours préfixe

fonction PARCOURS-PREFIXE(p),
  si $p \neq \square$ **alors**
    Afficher($val(p)$),
    PARCOURS-PREFIXE($filsG(p)$),
    PARCOURS-PREFIXE($filsD(p)$),
  finsi
fin fonction



12 1 91 67 7 82 61

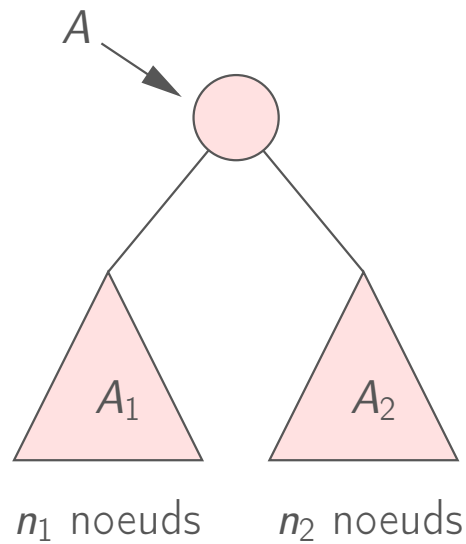
Parcours *infixe* : 91, 1, 67, 12, 7, 61, 82.
Parcours *postfixe* : 91, 67, 1, 61, 82, 7, 12.

Arbres binaires : hauteur minimale

Propriété. Pour tout arbre A de taille n, $n > 0$,

$$h(A) \geq \lfloor \log_2 n \rfloor + 1$$

taille : nombre de noeuds, $h(A)$: hauteur de l'arbre A



Arbres binaires : hauteur minimale

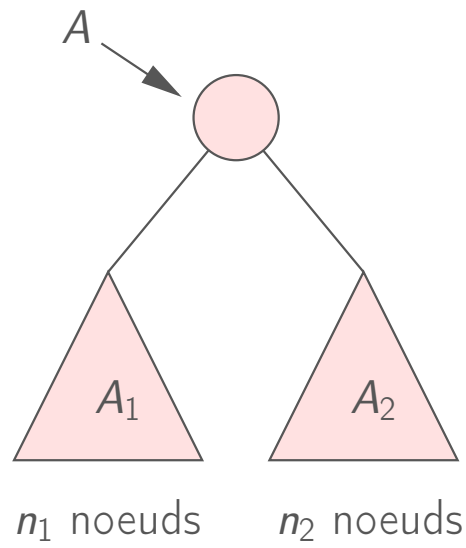
Propriété. Pour tout arbre A de taille n, $n > 0$,

$$h(A) \geq \lfloor \log_2 n \rfloor + 1$$

taille : nombre de noeuds, $h(A)$: hauteur de l'arbre A

Preuve. Si $n = 1$ alors $h(A) = 1$, sinon

$$h(A) = 1 + \max(h(A_1), h(A_2))$$



Arbres binaires : hauteur minimale

Propriété. Pour tout arbre A de taille n, $n > 0$,

$$h(A) \geq \lfloor \log_2 n \rfloor + 1$$

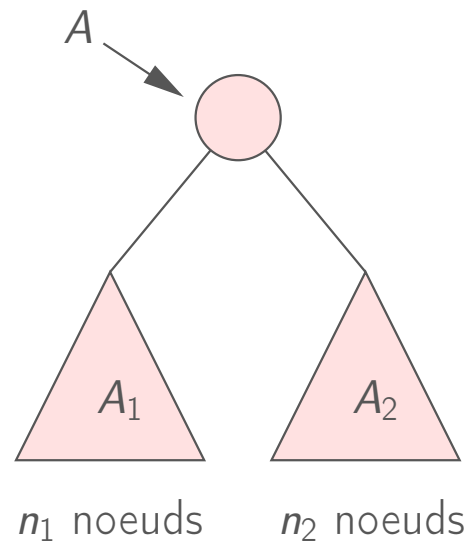
taille : nombre de noeuds, $h(A)$: hauteur de l'arbre A

Preuve. Si $n = 1$ alors $h(A) = 1$, sinon

$$h(A) = 1 + \max(h(A_1), h(A_2))$$

On suppose $n_1 \geq n_2$, et donc $n_1 \geq n/2$

Induction : $h(A_1) \geq 1 + \lfloor \log_2 n_1 \rfloor$
donc $h(A_1) \geq 1 + \lfloor \log_2 n/2 \rfloor = \lfloor \log_2 n \rfloor$,
et puisque $h(A) \geq 1 + h(A_1)$,
on en déduit $h(A) \geq 1 + \lfloor \log_2 n \rfloor$.



Arbres binaires : hauteur minimale

Propriété. Pour tout arbre A de taille n, $n > 0$,

$$h(A) \geq \lfloor \log_2 n \rfloor + 1$$

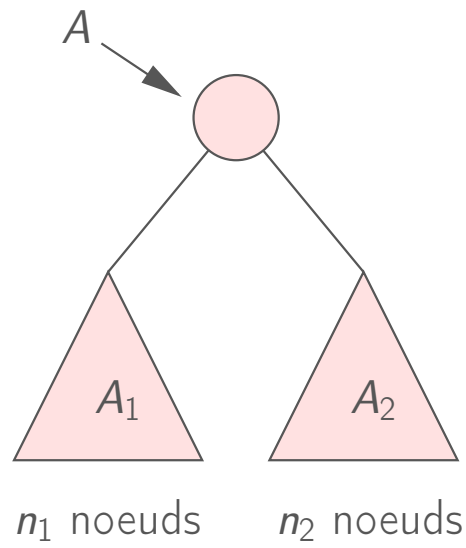
taille : nombre de noeuds, $h(A)$: hauteur de l'arbre A

Preuve. Si $n = 1$ alors $h(A) = 1$, sinon

$$h(A) = 1 + \max(h(A_1), h(A_2))$$

On suppose $n_1 \geq n_2$, et donc $n_1 \geq n/2$

Induction : $h(A_1) \geq 1 + \lfloor \log_2 n_1 \rfloor$
donc $h(A_1) \geq 1 + \lfloor \log_2 n/2 \rfloor = \lfloor \log_2 n \rfloor$,
et puisque $h(A) \geq 1 + h(A_1)$,
on en déduit $h(A) \geq 1 + \lfloor \log_2 n \rfloor$.

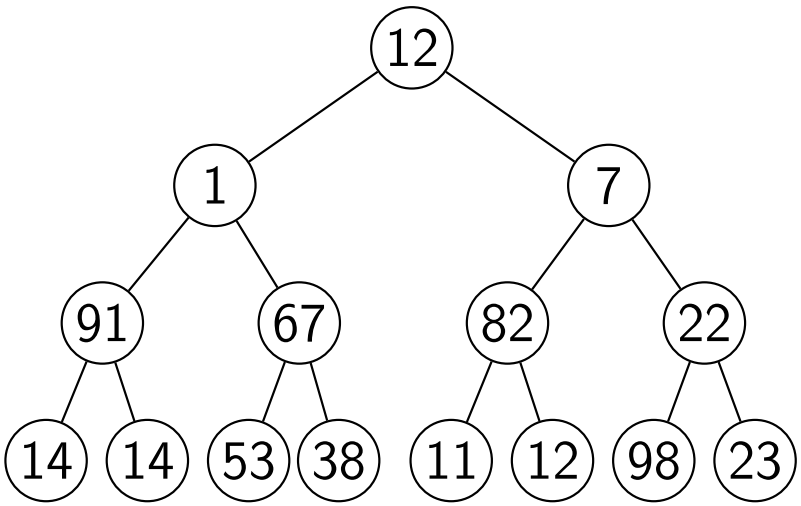


$\lceil \log_2 n_f \rceil + 1$ si n_f est le nombre de feuilles.

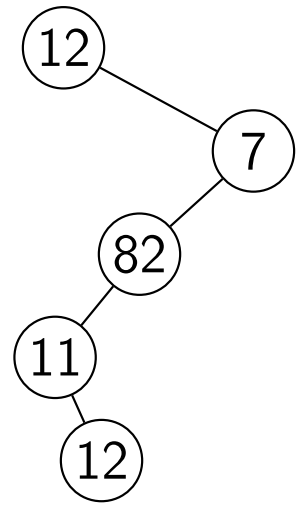
Arbres binaires : hauteur

Propriété. Pour tout arbre A constitué de n noeuds

$$\lfloor \log_2 n \rfloor + 1 \leq h(A) \leq n$$



15 noeuds, $h = 4$



5 noeuds, $h = 5$

Structure de données *dictionnaire*

Ensemble dynamique d'objets comparables qui supporte les opérations suivantes :

estvide() : détermine si le dictionnaire est vide

ajouter(e) : ajout d'un élément

supprimer(e) : suppression d'un élément

rechercher(e) : recherche d'un élément

Implémentation : liste, tableau, arbre binaire de recherche, table de hashage,...

Objectifs : être efficace, utiliser peu d'espace.

Arbres binaires de recherche

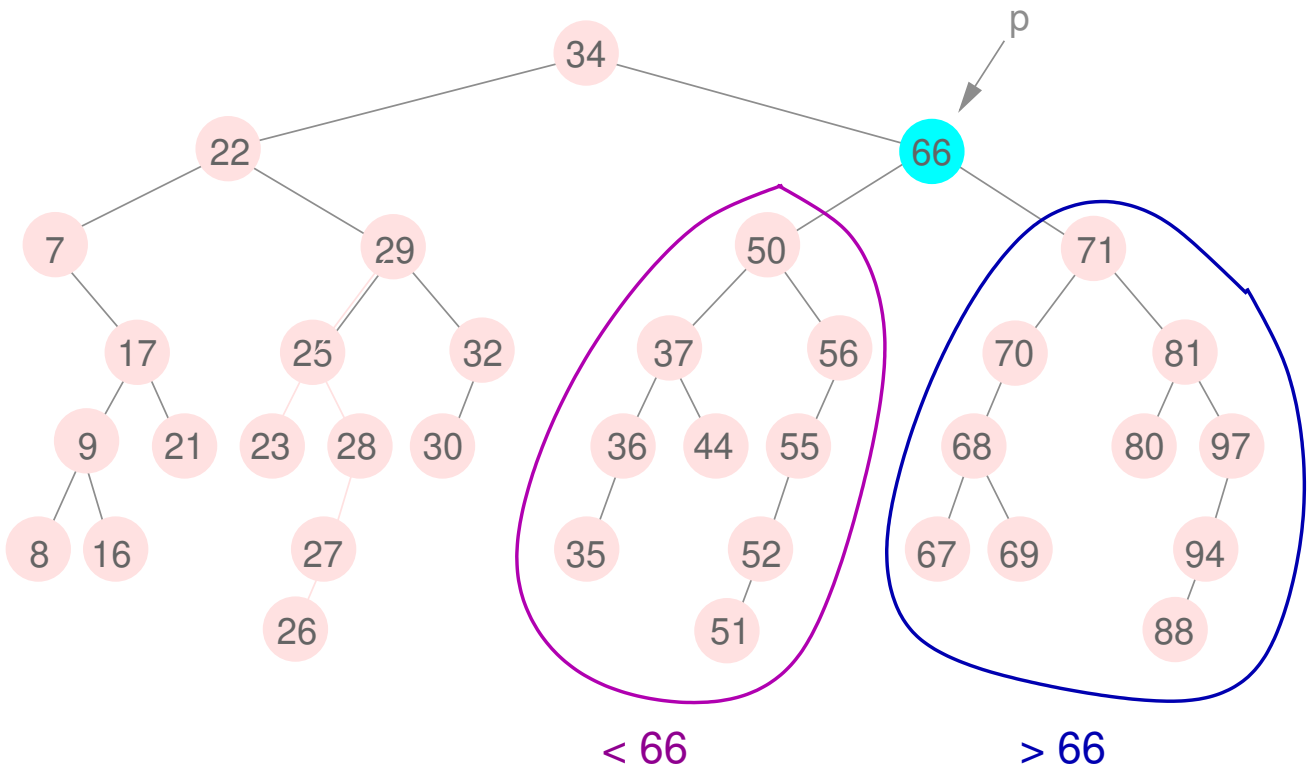
Soit E un ensemble muni d'une relation d'ordre $\prec$ (ordre total)

Un arbre binaire A étiqueté avec des éléments de E est un arbre binaire de recherche s'il satisfait l'ordre infixe, c'est à dire, pour tout noeud $p \in A$

- $\forall q \in \mathit{filsG}(p)$, $\mathit{val}(q) \prec \mathit{val}(p)$,
- $\forall q \in \mathit{filsD}(p)$, $\mathit{val}(q) \succ \mathit{val}(p)$.

Les éléments figurant dans l'arbre sont tous différents
(i.e. si $x \preceq y$ et $y \preceq x$ alors x et y sont le même élément)

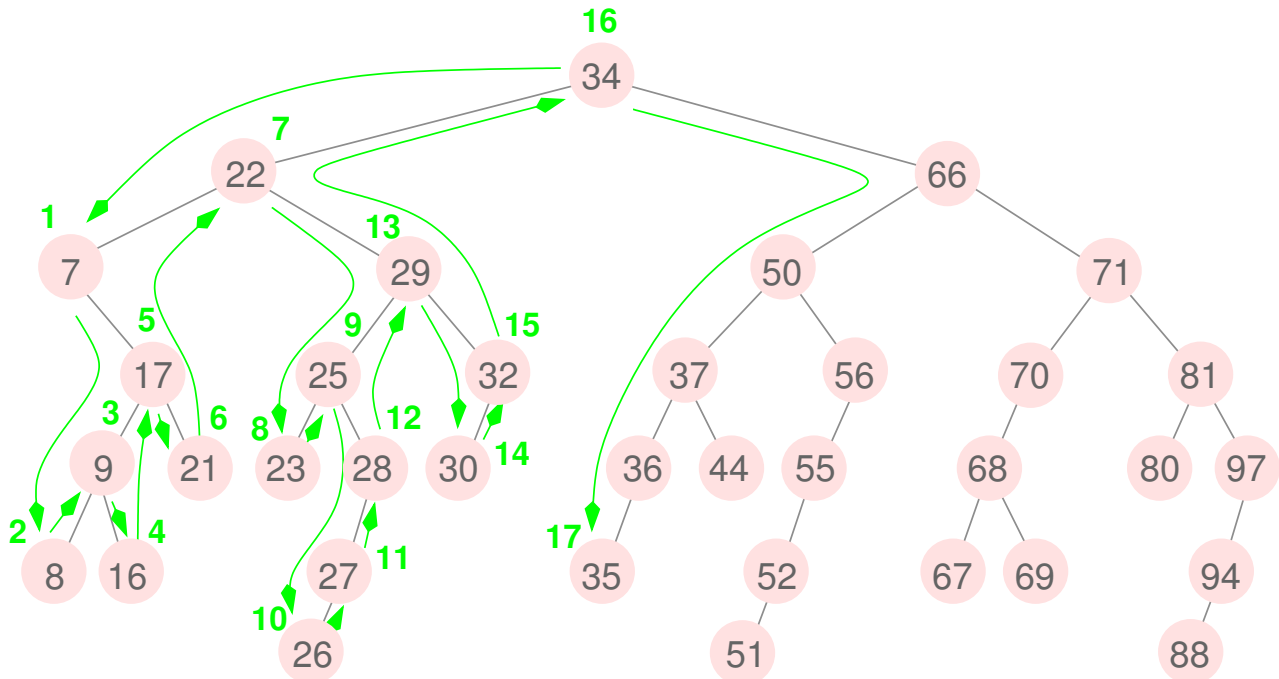
Arbres binaires de recherche



éléments pris dans $(\mathbb{N}, <)$

Arbres binaires de recherche : parcours infixe

Le parcours infixe de l'arbre produit la suite ordonnée des éléments

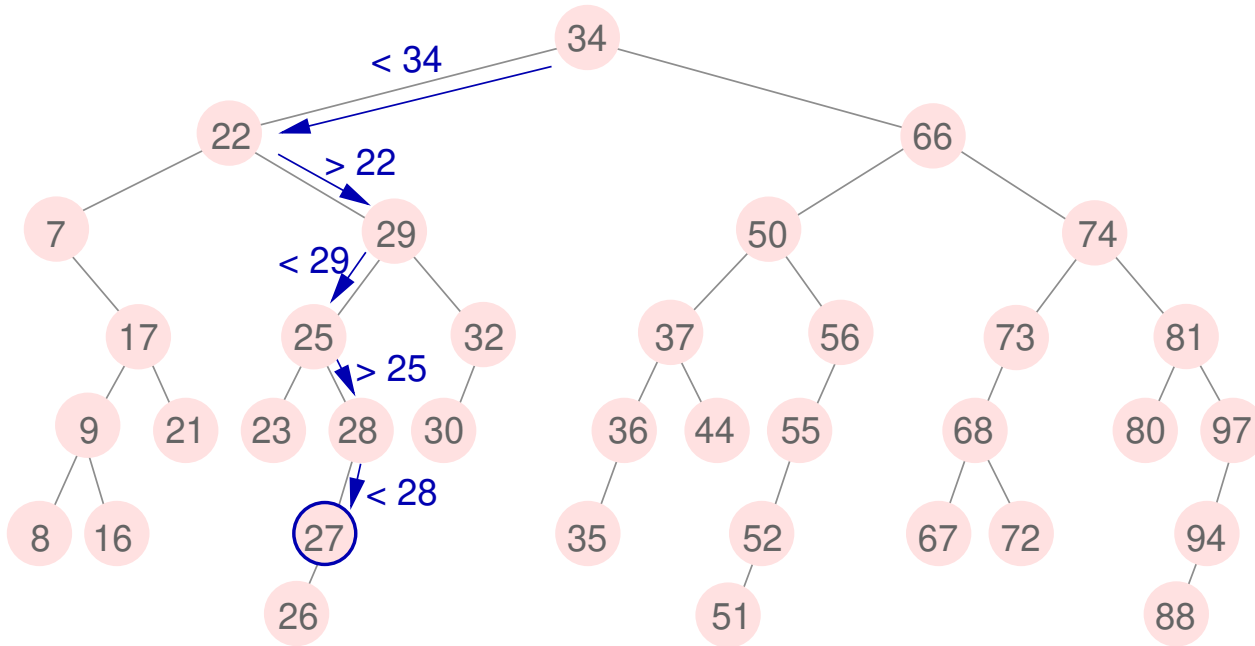


7 8 9 16 17 21 22 23 25 26 27 28 29 30 32 34 35 36 37 ...

ABR : Recherche d'un élément

Soit e un élément qui apparaît dans l'arbre de racine p.
Si $e \neq val(p)$ alors

- soit $e \prec val(p)$ et $e \in filsG(p)$
- soit $e \succ val(p)$ et $e \in filsD(p)$



recherche de la valeur 27

ABR : Recherche d'un élément

fonction RECHERCHE(x, p)

out : $\square$ si x n'apparaît pas dans le sous-arbre enraciné en p, le noeud de valeur x sinon,

si $p = \square$ **alors renvoyer** $\square$,

si $val(p) = x$ **alors renvoyer** p,

si $val(p) > x$ **alors renvoyer** RECHERCHE$(x, filsG(p))$,

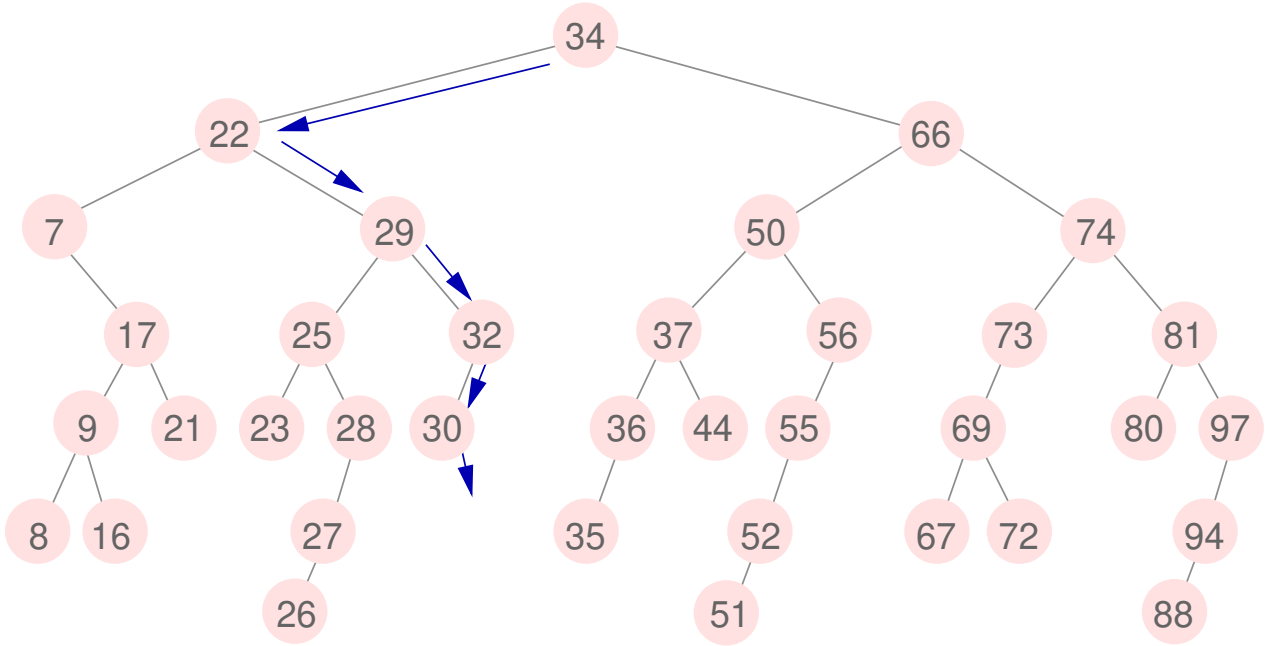
renvoyer RECHERCHE$(x, filsD(p))$,

fin fonction

Parcours d'une branche : $\mathcal{O}(h(A))$

ABR : ajout d'un élément (insertion)

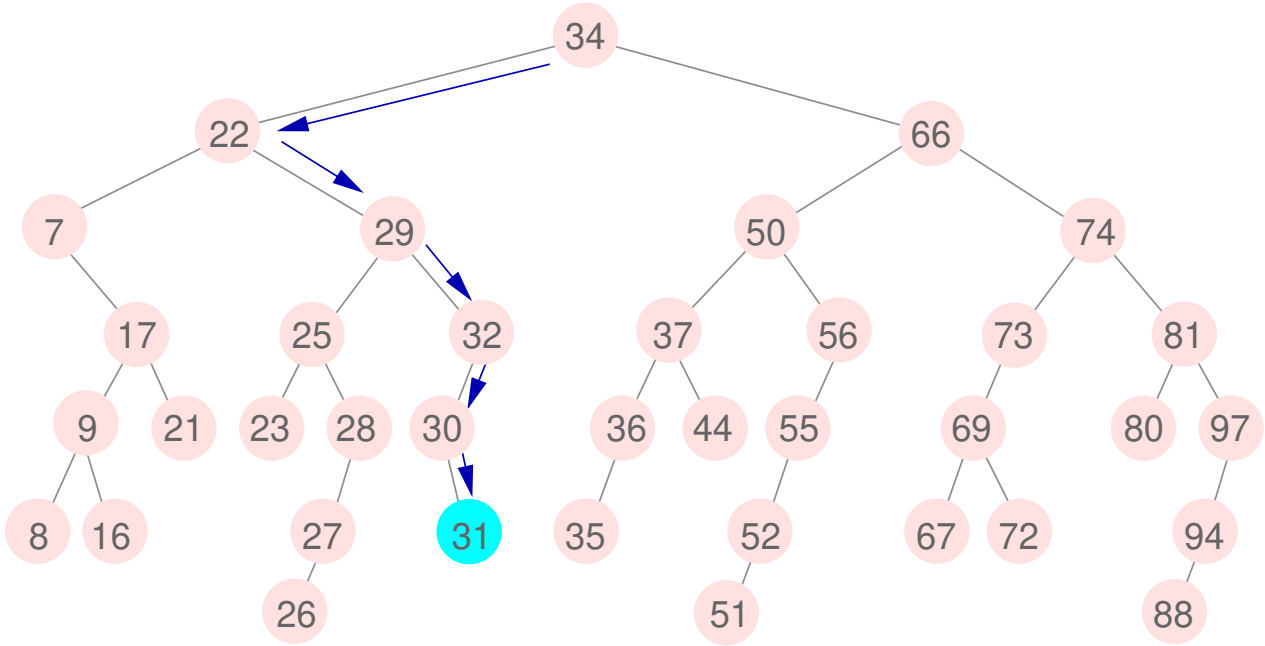
Ajout de la valeur 31



Même principe que la recherche : parcours d'une branche

ABR : ajout d'un élément (insertion)

Ajout de la valeur 31



Ajout au bout de la branche

ABR : ajout d'un élément (insertion)

fonction INSERER(x, p)

insère la valeur x dans l'arbre p, renvoie l'arbre résultant,

si $p = \square$ **alors renvoyer** $\langle x, \square, \square \rangle$,

si $val(p) > x$ **alors** filsG(p) :=INSERER(x,filsG(p)),

si $val(p) < x$ **alors** filsD(p) :=INSERER(x,filsD(p)),

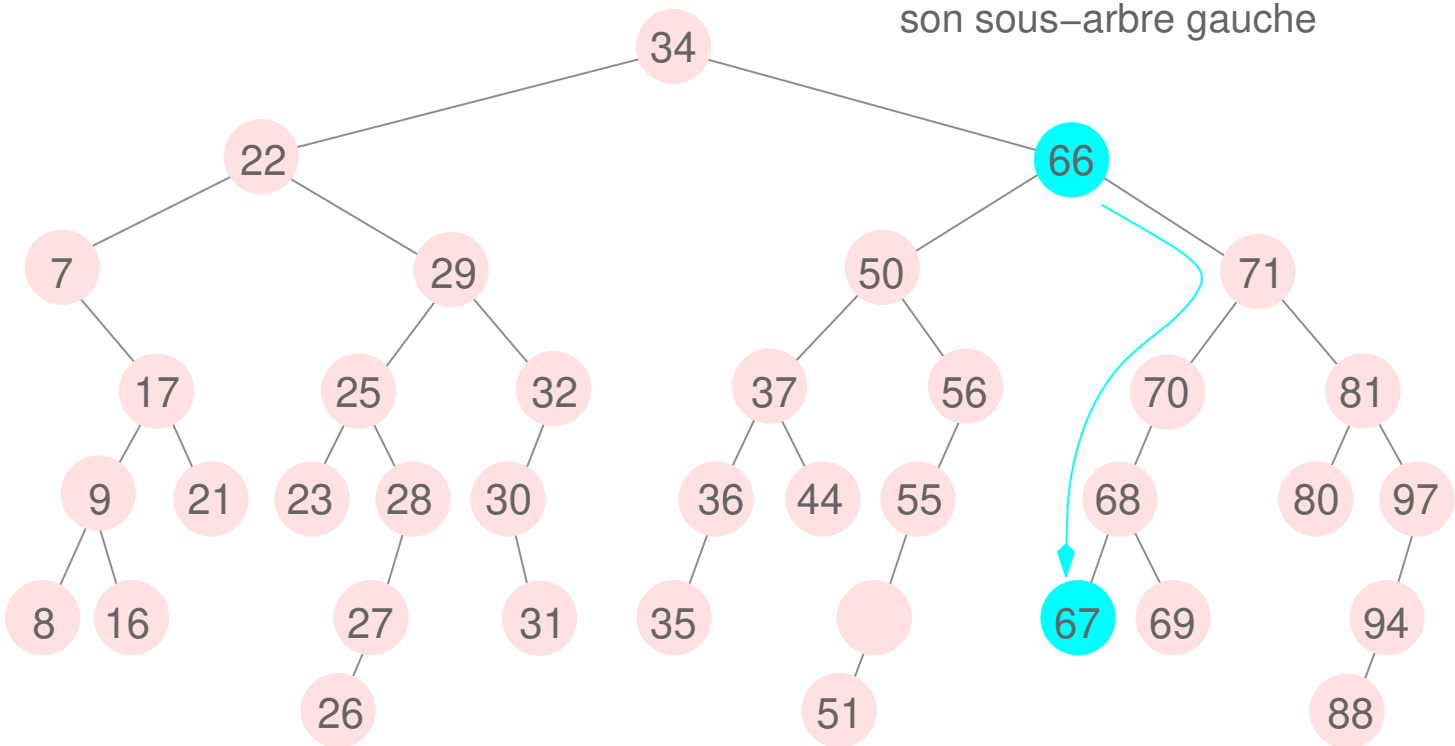
renvoyer p,

fin fonction

Parcours d'une branche : $\mathcal{O}(h(A))$

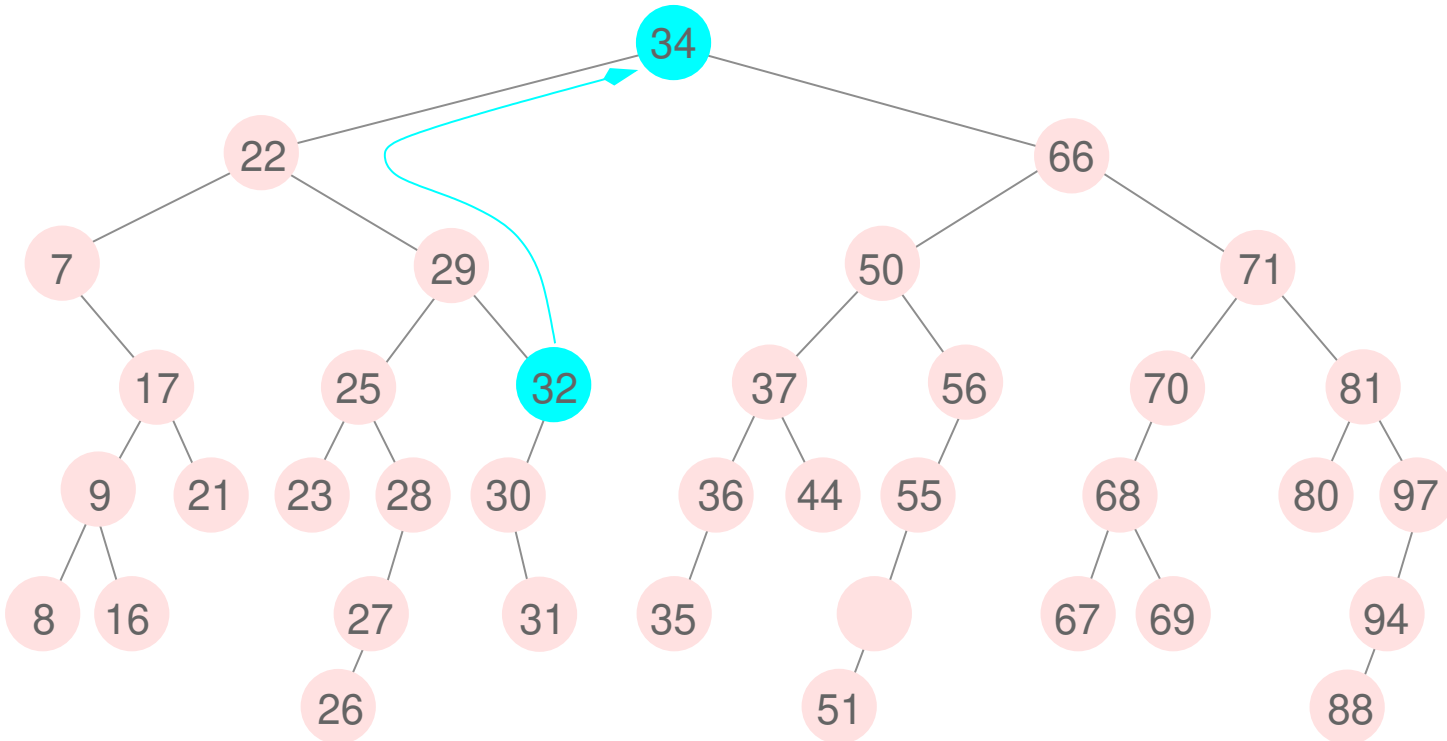
ABR : successeur

le noeud de valeur 66 a un fils droit,
son successeur est 67,
dernier fils gauche de
son sous–arbre gauche



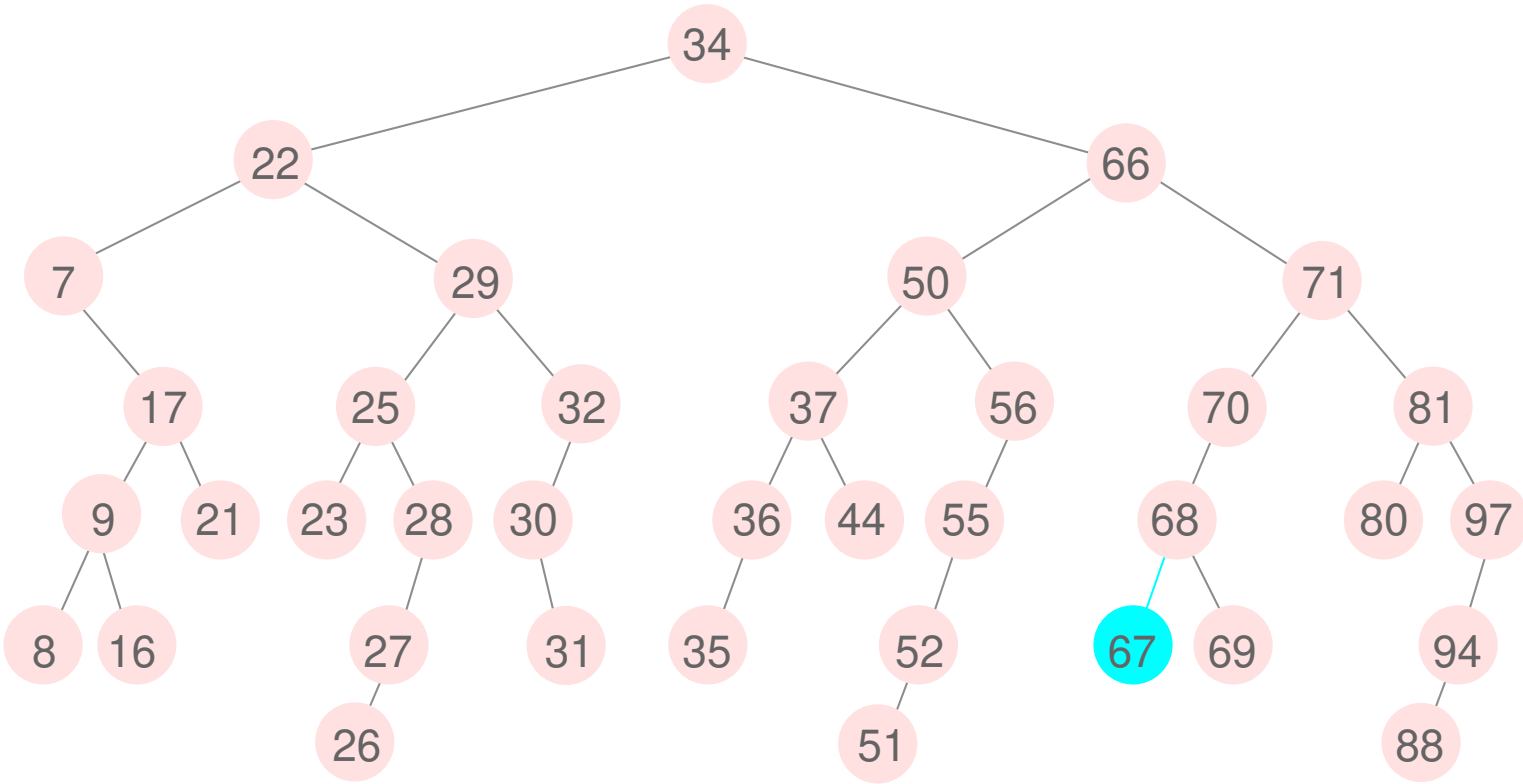
ABR : successeur

le noeud de valeur 32 n'a pas de fils droit :
son successeur est 34, premier ascendant de 32
tel que 32 figure dans son sous–arbre gauche



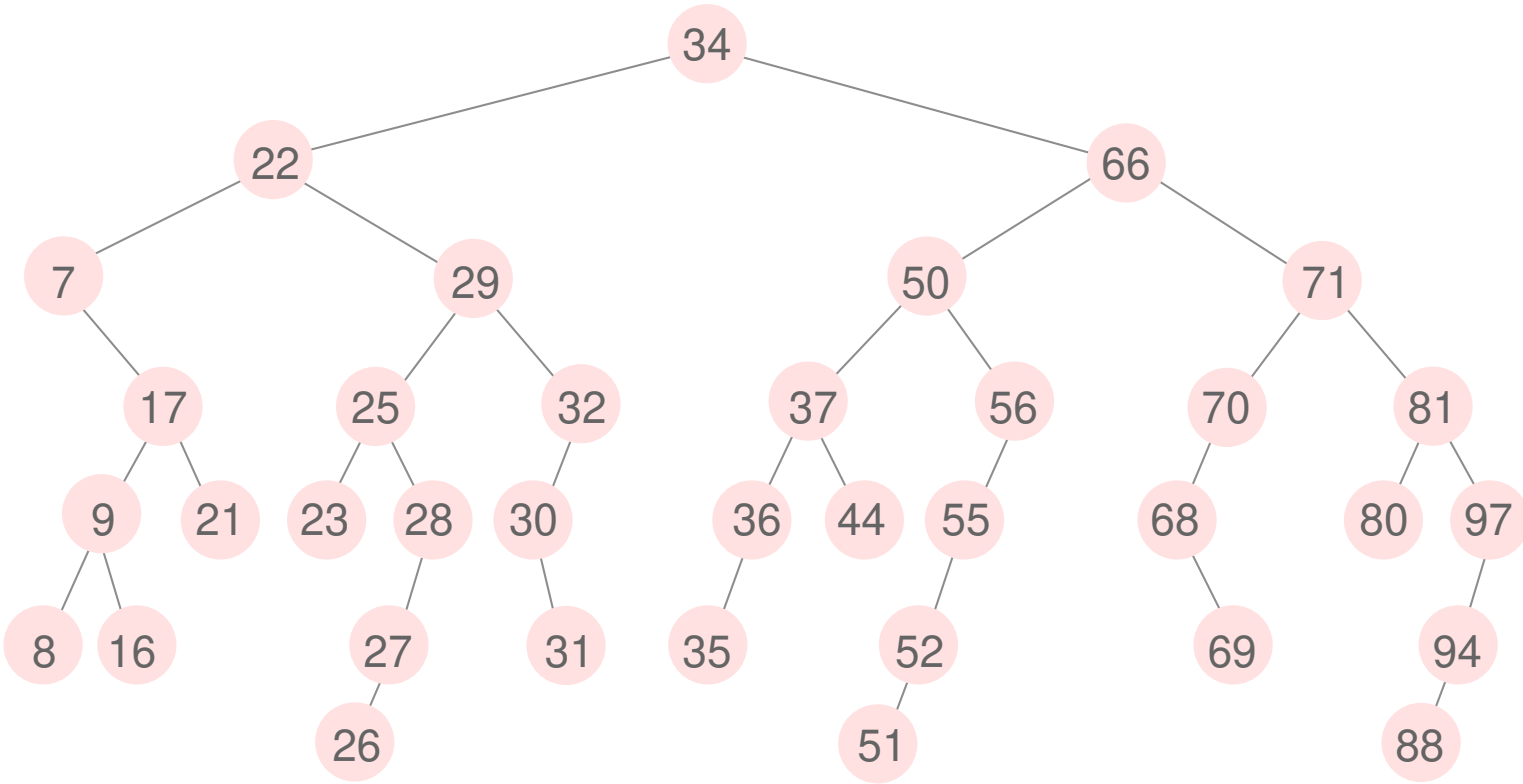
ABR : suppression d'un élément

Cas 1 : le noeud n'a pas de fils : *décrochage*



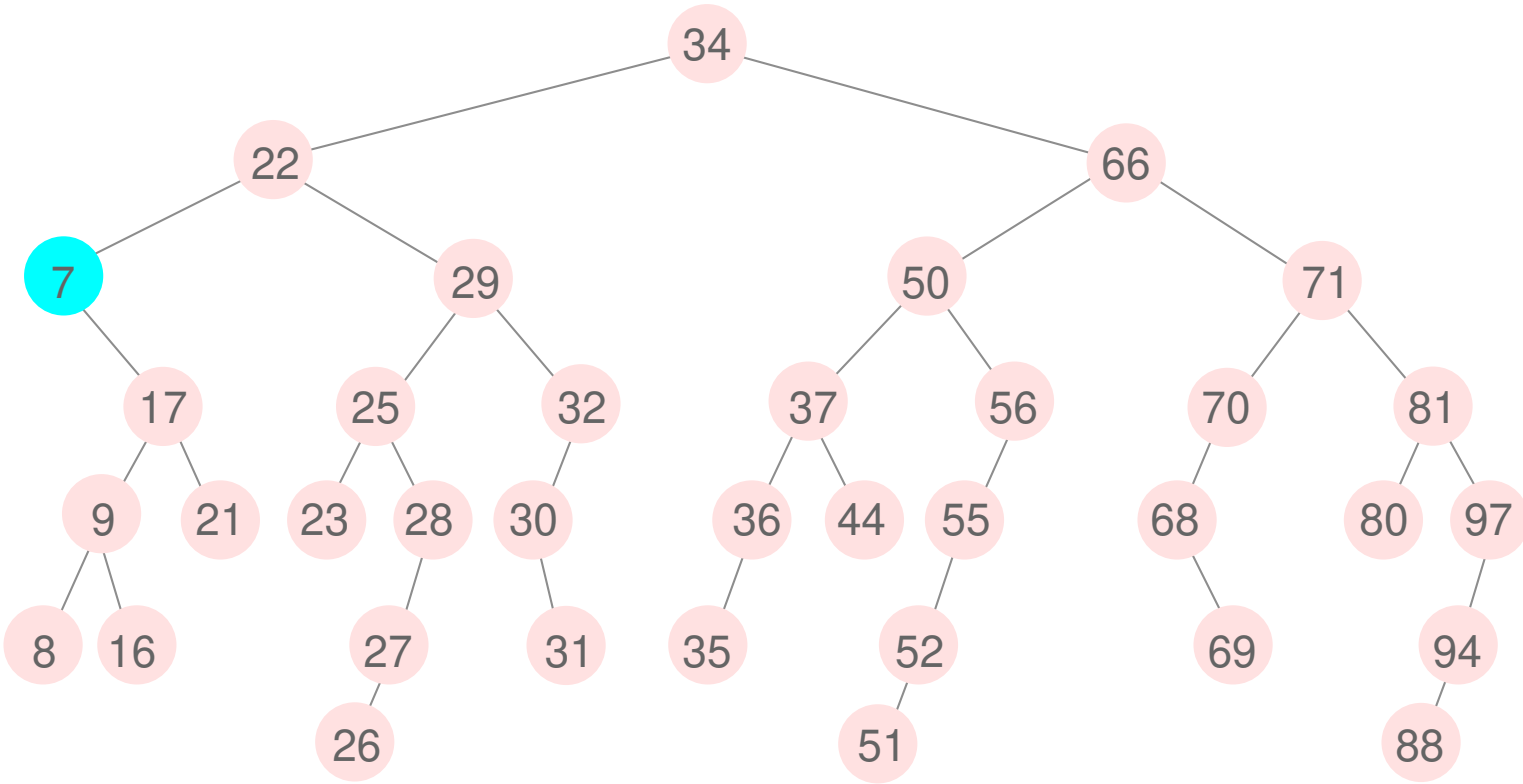
ABR : suppression d'un élément

Cas 1 : le noeud n'a pas de fils : *décrochage*



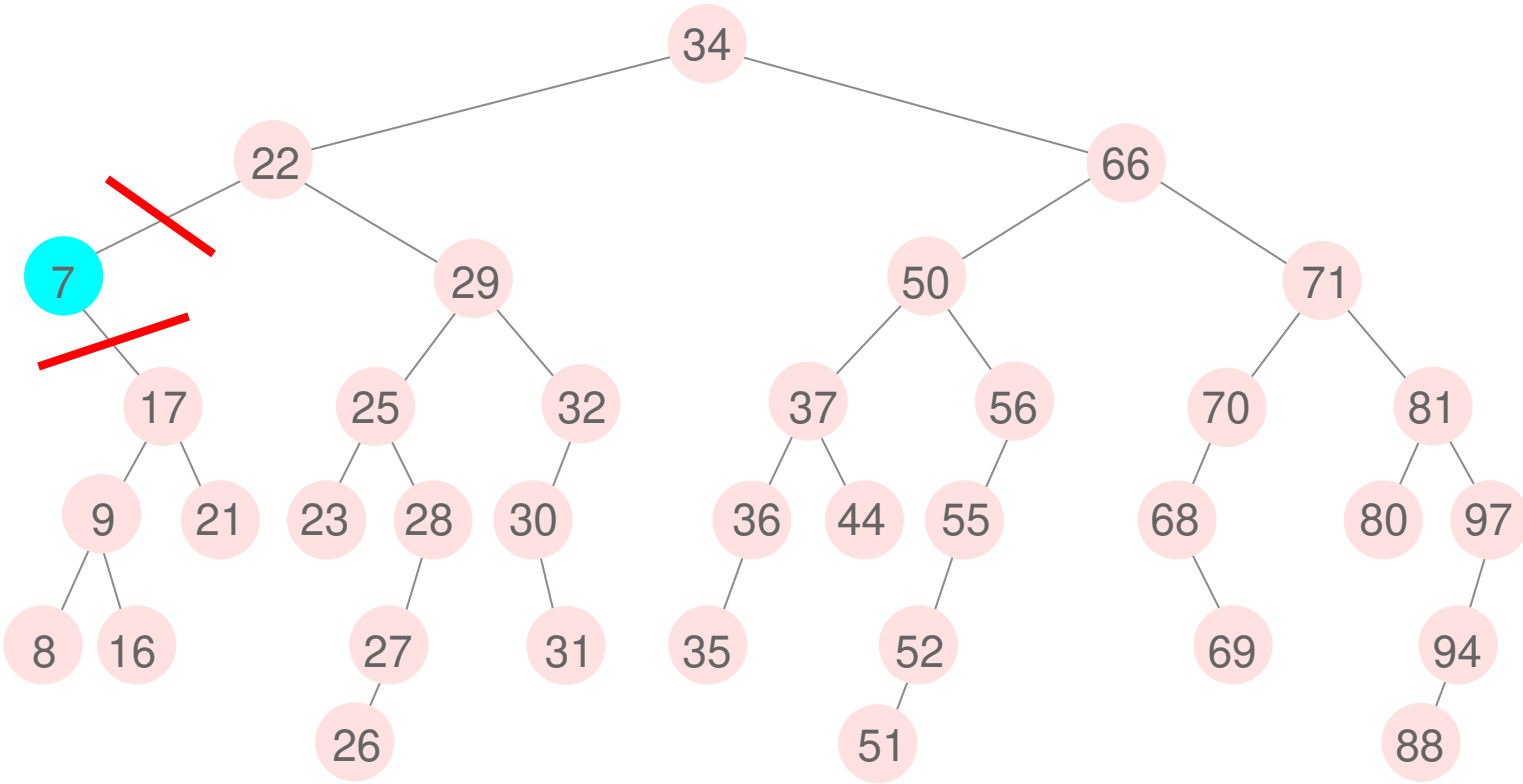
ABR : suppression d'un élément

Cas 2 : le noeud a un seul fils : *décrochage*



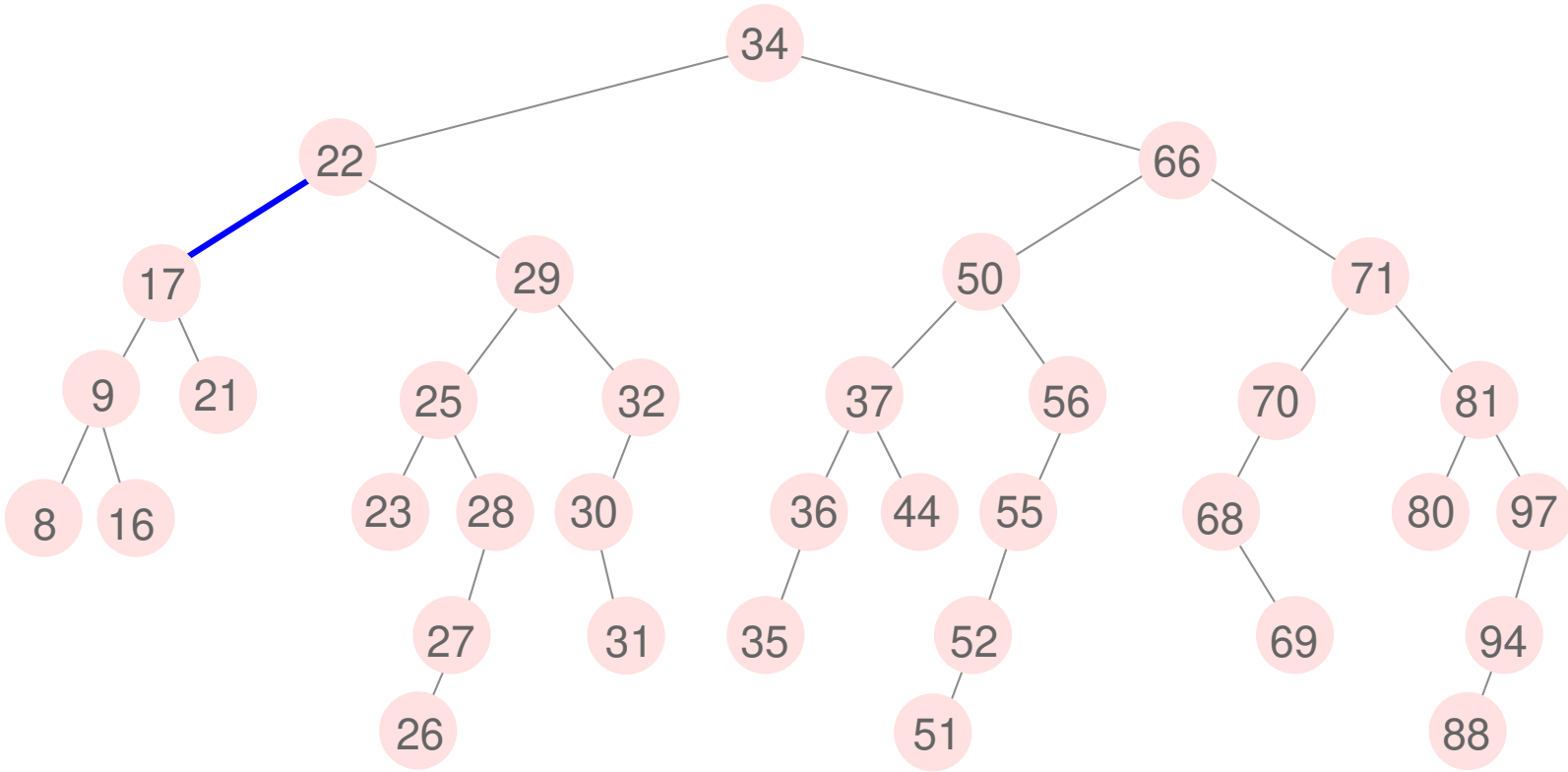
ABR : suppression d'un élément

Cas 2 : le noeud a un seul fils : *décrochage*



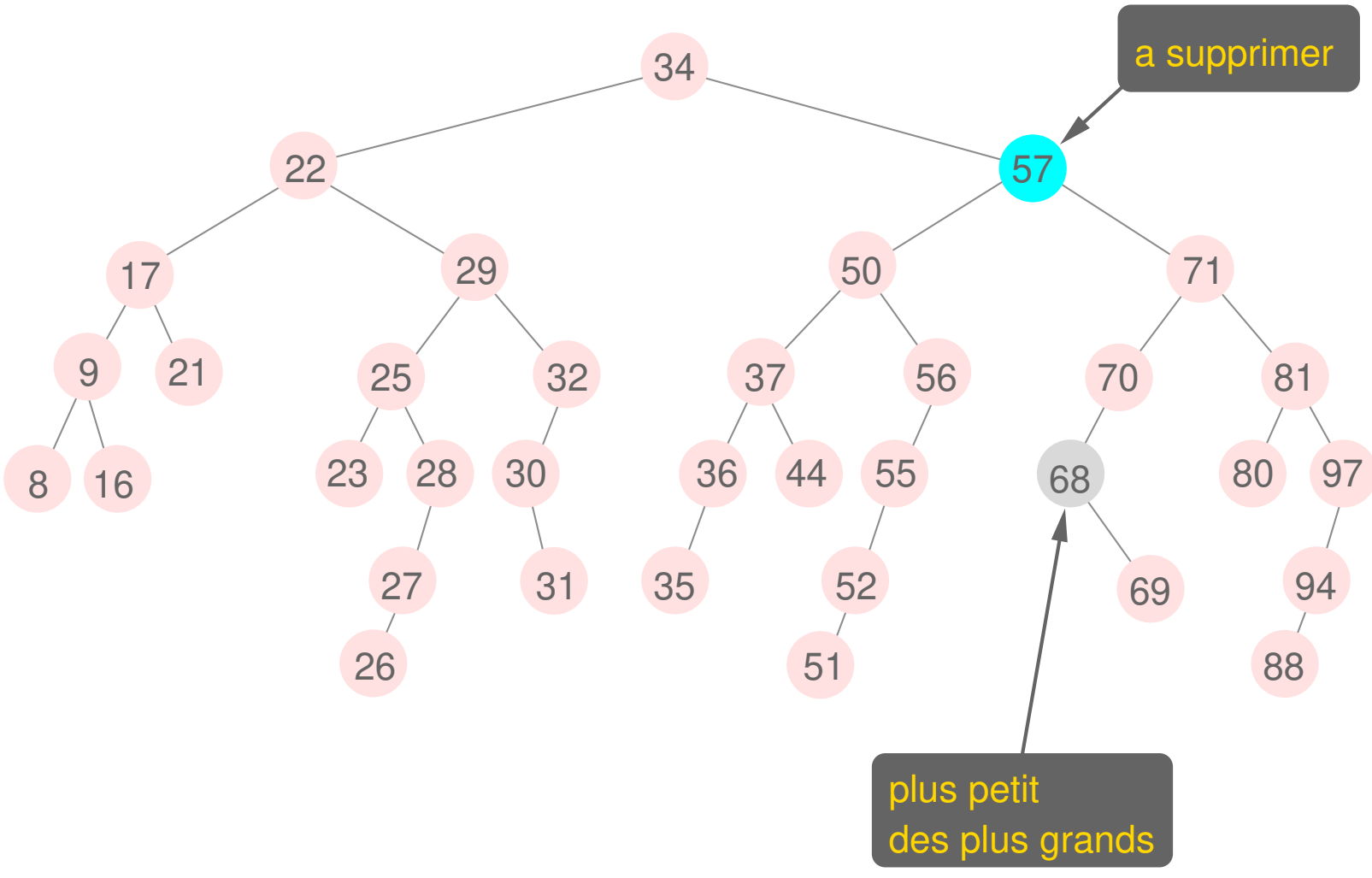
ABR : suppression d'un élément

Cas 2 : le noeud a un seul fils : *décrochage*



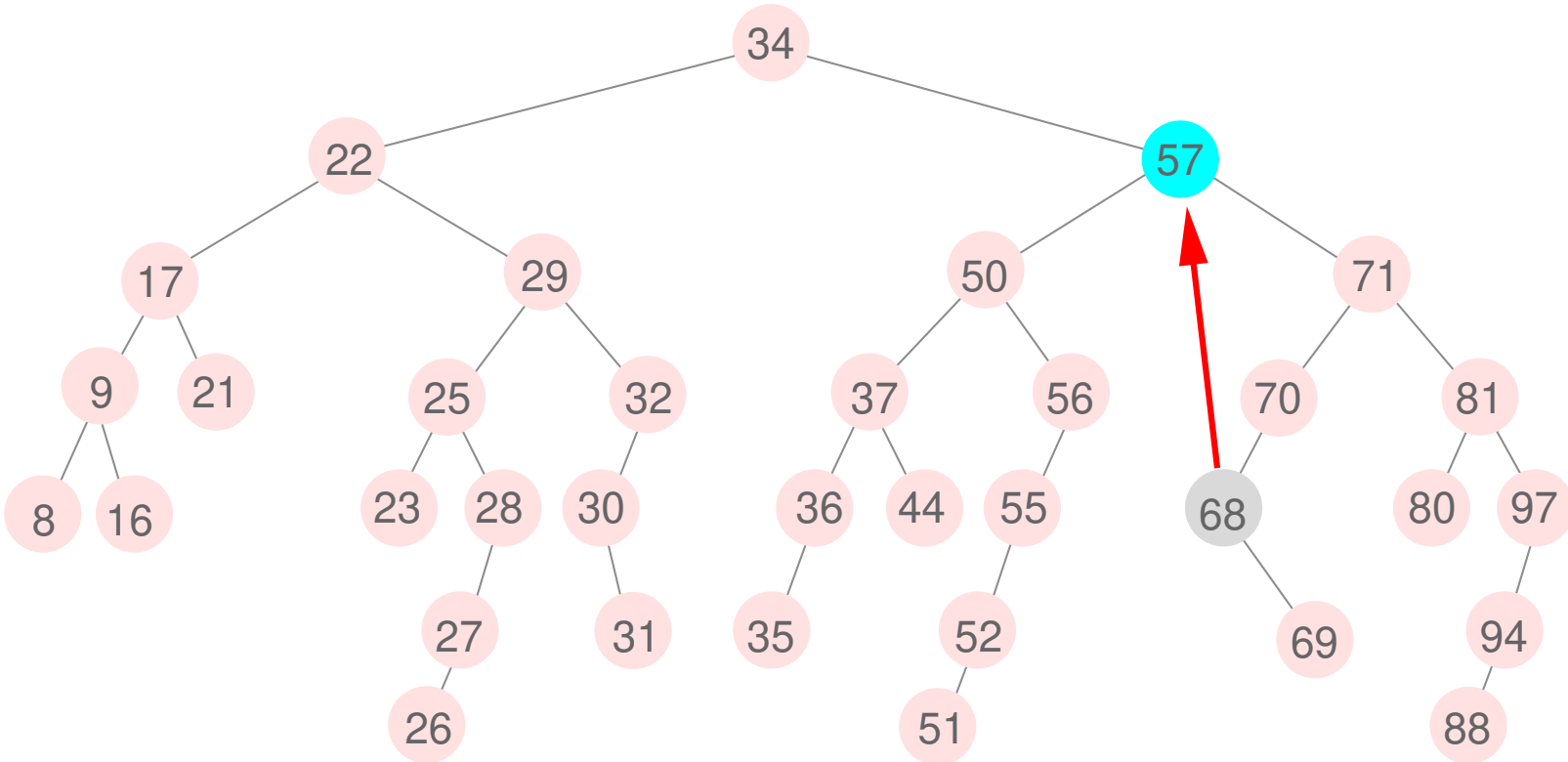
ABR : suppression d'un élément

Cas 3 : le noeud a deux fils



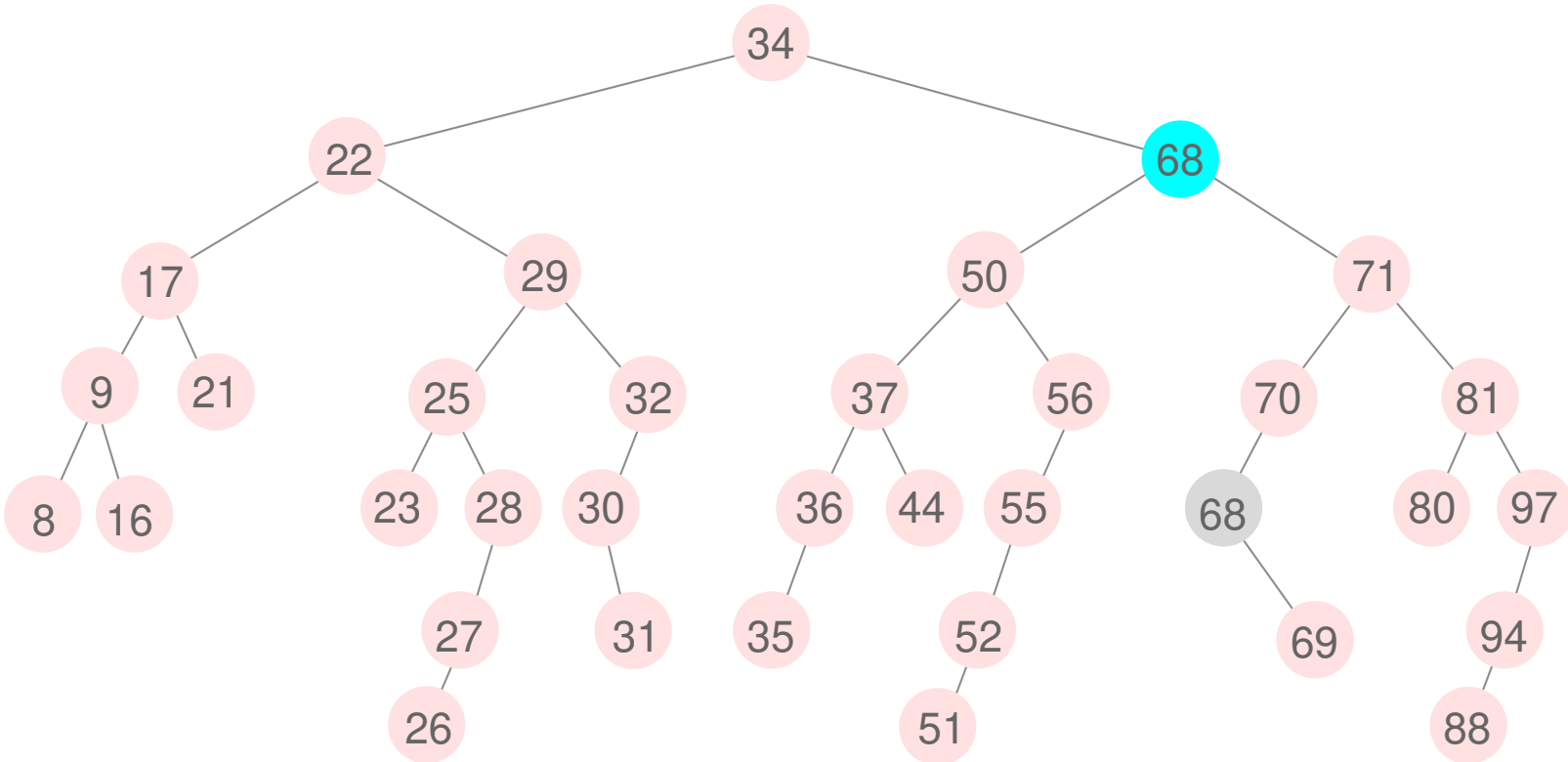
ABR : suppression d'un élément

Cas 3 (deux fils) : *copie du plus petit des plus grands*



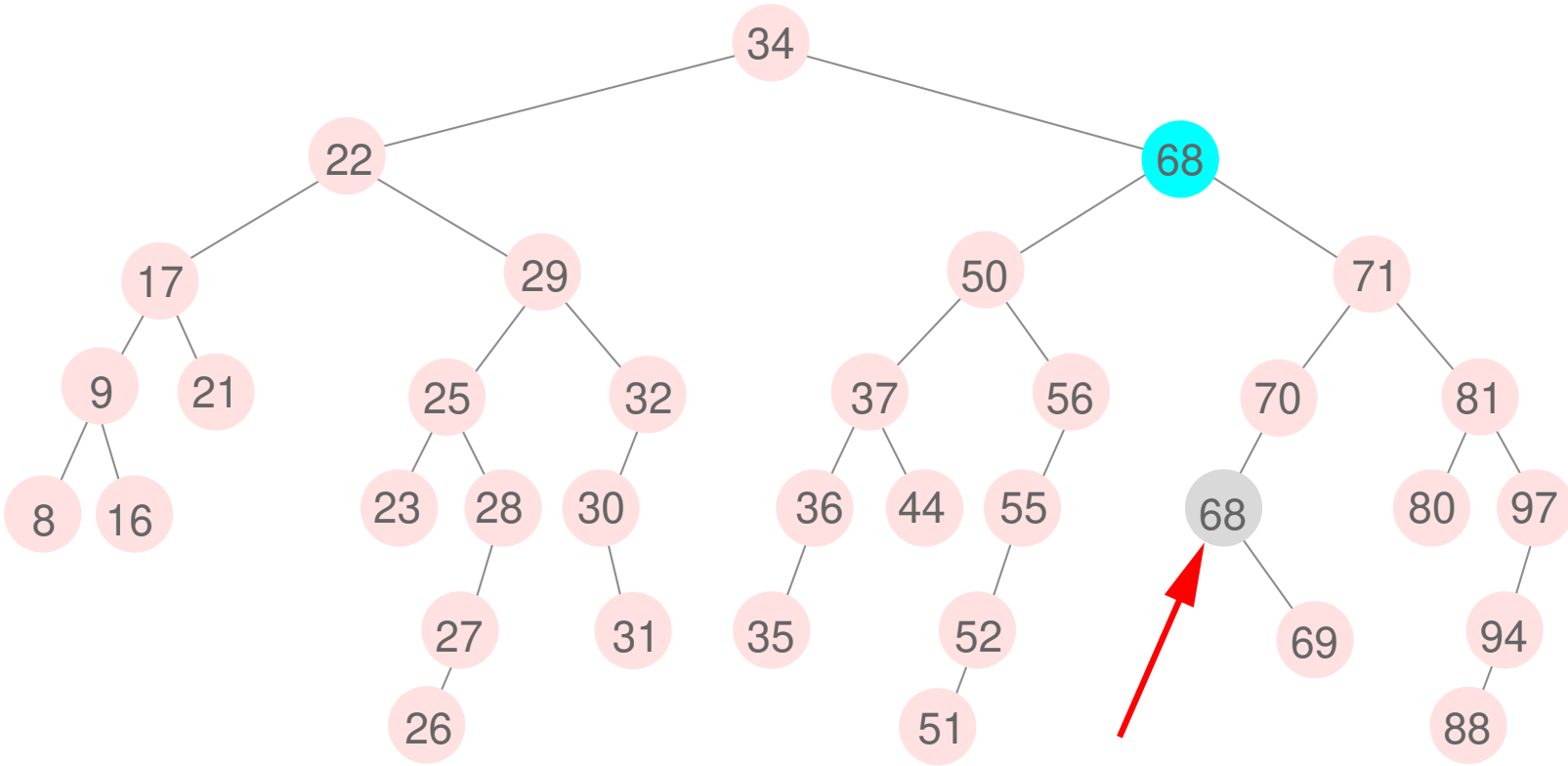
ABR : suppression d'un élément

Cas 3 (deux fils) : *copie du plus petit des plus grands*



ABR : suppression d'un élément

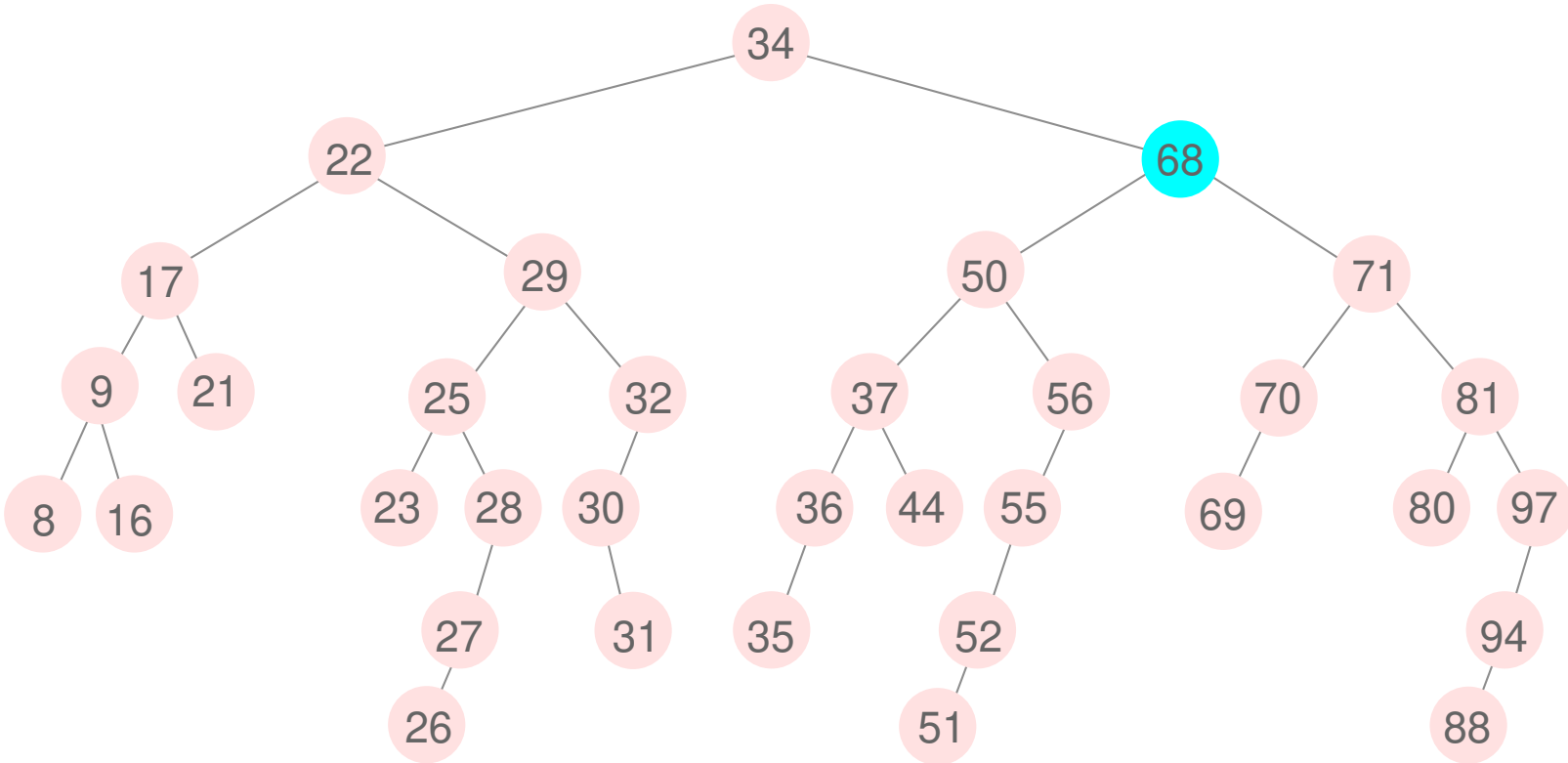
Cas 3 (deux fils) : *décrochage*



le plus petit des plus grand n'a pas de fils gauche

ABR : suppression d'un élément

Cas 3 (deux fils) : *décrochage*



ABR : suppression d'un élément

```
fonction SUPPRIMER(x, p)
    si val(p) > x alors
        filsG(p) :=SUPPRIMER(x,filsG(p)),
    sinon si val(p) < x alors
        filsD(p) :=SUPPRIMER(x,filsD(p)),
    sinon si filsG(p) = □ alors
        renvoyer filsD(p),
    sinon si filsD(p) = □ alors
        renvoyer filsG(p),
    sinon
        val(p) := GETMIN(filsD(p)),
        filsD(p)) :=SUPPRIMER(val(p),filsD(p)),
    renvoyer p,
fin fonction
```

Supprime la valeur x de l'arbre p, renvoie l'arbre résultant.
Suppose que l'élément est présent dans l'arbre.

ABR : suppression d'un élément

```
fonction SUPPRIMER(x, p)
    si val(p) > x alors                    x est dans le sous-arbre gauche
        filsG(p) :=SUPPRIMER(x,filsG(p)),
    sinon si val(p) < x alors
        filsD(p) :=SUPPRIMER(x,filsD(p)),
    sinon si filsG(p) = □ alors
        renvoyer filsD(p),
    sinon si filsD(p) = □ alors
        renvoyer filsG(p),
    sinon
        val(p) := GETMIN(filsD(p)),
        filsD(p)) :=SUPPRIMER(val(p),filsD(p)),
    renvoyer p,
fin fonction
```

Supprime la valeur x de l'arbre p, renvoie l'arbre résultant.
Suppose que l'élément est dans le dictionnaire.

ABR : suppression d'un élément

```
fonction SUPPRIMER(x, p)
    si val(p) > x alors
        filsG(p) :=SUPPRIMER(x,filsG(p)),
    sinon si val(p) < x alors          x est dans le sous-arbre droit
        filsD(p) :=SUPPRIMER(x,filsD(p)),
    sinon si filsG(p) = □ alors
        renvoyer filsD(p),
    sinon si filsD(p) = □ alors
        renvoyer filsG(p),
    sinon
        val(p) := GETMIN(filsD(p)),
        filsD(p)) :=SUPPRIMER(val(p),filsD(p)),
    renvoyer p,
fin fonction
```

Supprime la valeur x de l'arbre p, renvoie l'arbre résultant.
Suppose que l'élément est dans le dictionnaire.

ABR : suppression d'un élément

```
fonction SUPPRIMER(x, p)
    si val(p) > x alors
        filsG(p) :=SUPPRIMER(x,filsG(p)),
    sinon si val(p) < x alors
        filsD(p) :=SUPPRIMER(x,filsD(p)),
    sinon si filsG(p) = □ alors        val(p) = x, pas de fils gauche
        renvoyer filsD(p),
    sinon si filsD(p) = □ alors
        renvoyer filsG(p),
    sinon
        val(p) := GETMIN(filsD(p)),
        filsD(p)) :=SUPPRIMER(val(p),filsD(p)),
    renvoyer p,
fin fonction
```

Supprime la valeur x de l'arbre p, renvoie l'arbre résultant.
Suppose que l'élément est dans le dictionnaire.

ABR : suppression d'un élément

```
fonction SUPPRIMER(x, p)
    si val(p) > x alors
        filsG(p) :=SUPPRIMER(x,filsG(p)),
    sinon si val(p) < x alors
        filsD(p) :=SUPPRIMER(x,filsD(p)),
    sinon si filsG(p) = □ alors
        renvoyer filsD(p),
    sinon si filsD(p) = □ alors          val(p) = x, pas de fils droit
        renvoyer filsG(p),
    sinon
        val(p) := GETMIN(filsD(p)),
        filsD(p)) :=SUPPRIMER(val(p),filsD(p)),
    renvoyer p,
fin fonction
```

Supprime la valeur x de l'arbre p, renvoie l'arbre résultant.
Suppose que l'élément est dans le dictionnaire.

ABR : suppression d'un élément

```
fonction SUPPRIMER(x, p)
    si val(p) > x alors
        filsG(p) :=SUPPRIMER(x,filsG(p)),
    sinon si val(p) < x alors
        filsD(p) :=SUPPRIMER(x,filsD(p)),
    sinon si filsG(p) = □ alors
        renvoyer filsD(p),
    sinon si filsD(p) = □ alors
        renvoyer filsG(p),
    sinon                                   val(p) = x, deux fils,
        val(p) := GETMIN(filsD(p)),
        filsD(p)) :=SUPPRIMER(val(p),filsD(p)),
    renvoyer p,
fin fonction
```

Supprime la valeur x de l'arbre p, renvoie l'arbre résultant.
Suppose que l'élément est dans le dictionnaire.

Arbres binaires de recherche : suppression

```
fonction GetMin(p)
    si p = □ alors
        renvoyer □,
    si filsG(p) = □ alors
        renvoyer p,
    renvoyer GetMin(filsG(p)),
fin fonction
```

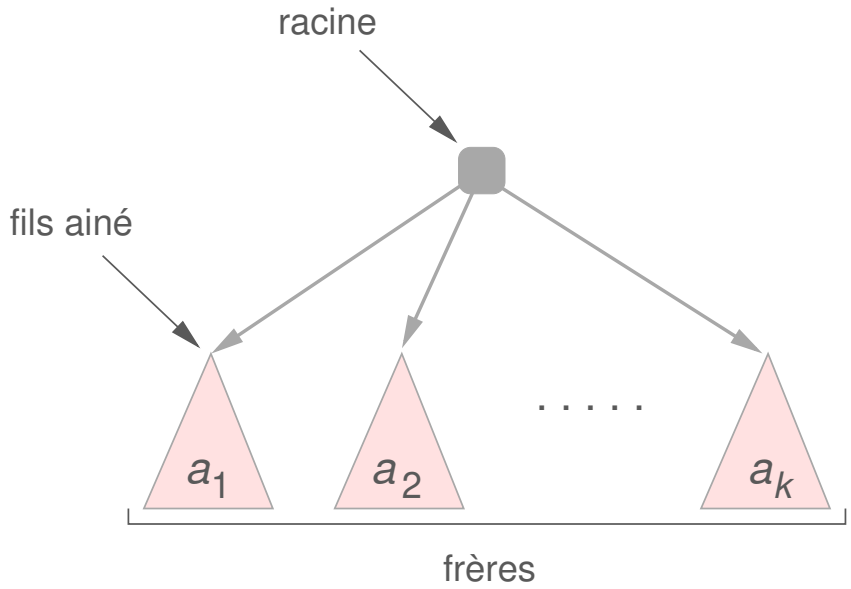
Arbres binaires de recherche : coûts

Structure	insertion	recherche	suppression
tableau	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
tableau trié	$\mathcal{O}(n)$	$\mathcal{O}(\log\ n)$	$\mathcal{O}(n)$
liste	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
ABR	$\mathcal{O}(h)$	$\mathcal{O}(h)$	$\mathcal{O}(h)$

où h est la hauteur de l'arbre :

$$1 + \lfloor \log_2\ n \rfloor \leq h \leq n$$

Arbres d'arité indéfinie (n-aires)



Arbres d'arité indéfinie : implémentation

class NoeudN

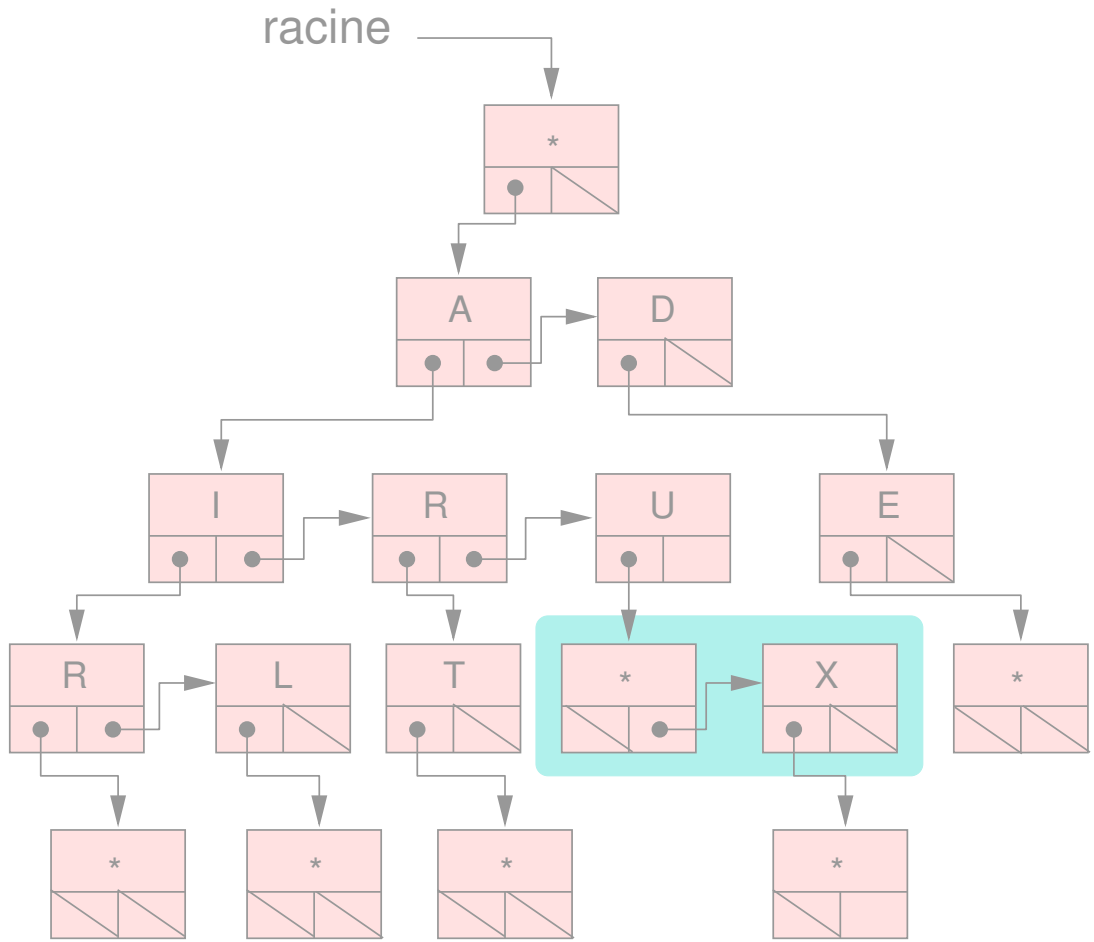
element : TYPE_ELEMENT,
filsainé : NoeudN,
frère : NoeudN,

Lien vers le frère : liste chainée des fils

class ArbreN

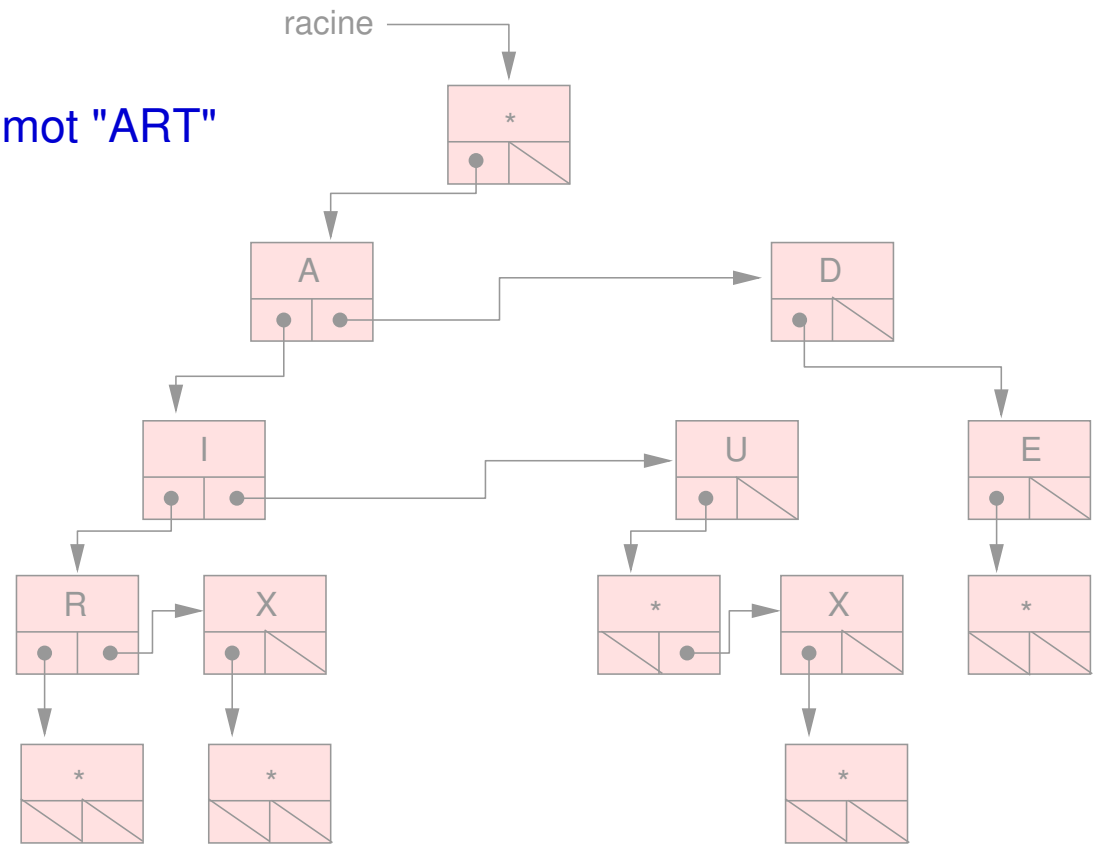
racine : NoeudN,

Arbres n-aires : arbres lexicographiques



Arbres n-aires : arbres lexicographiques

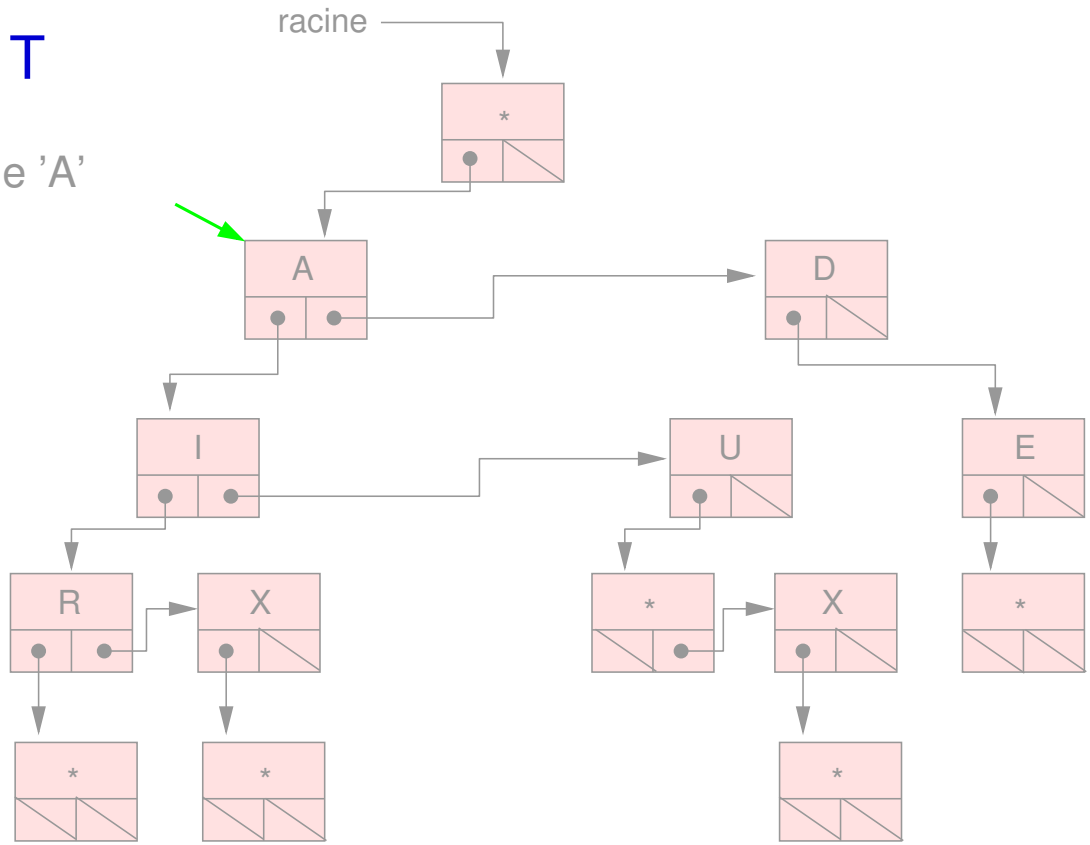
insertion du mot "ART"



Arbres *n*-aires : arbres lexicographiques

A R T

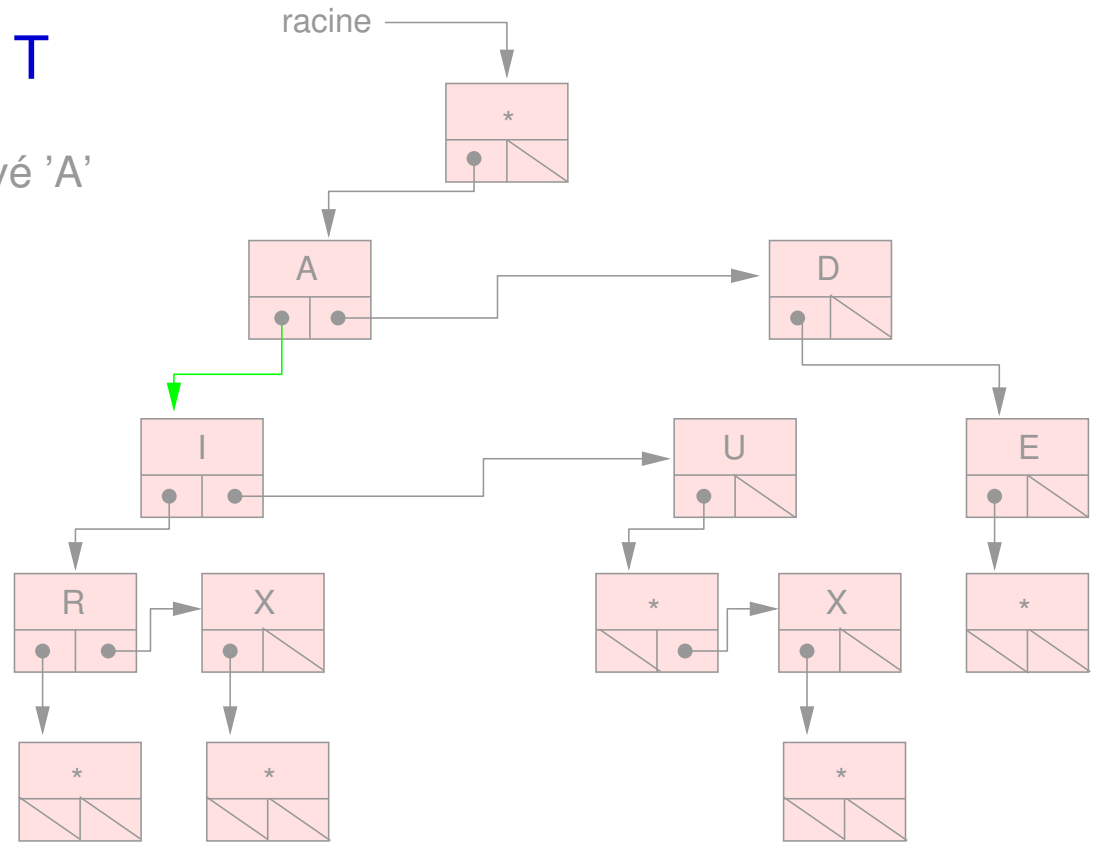
on cherche 'A'



Arbres *n*-aires : arbres lexicographiques

A R T

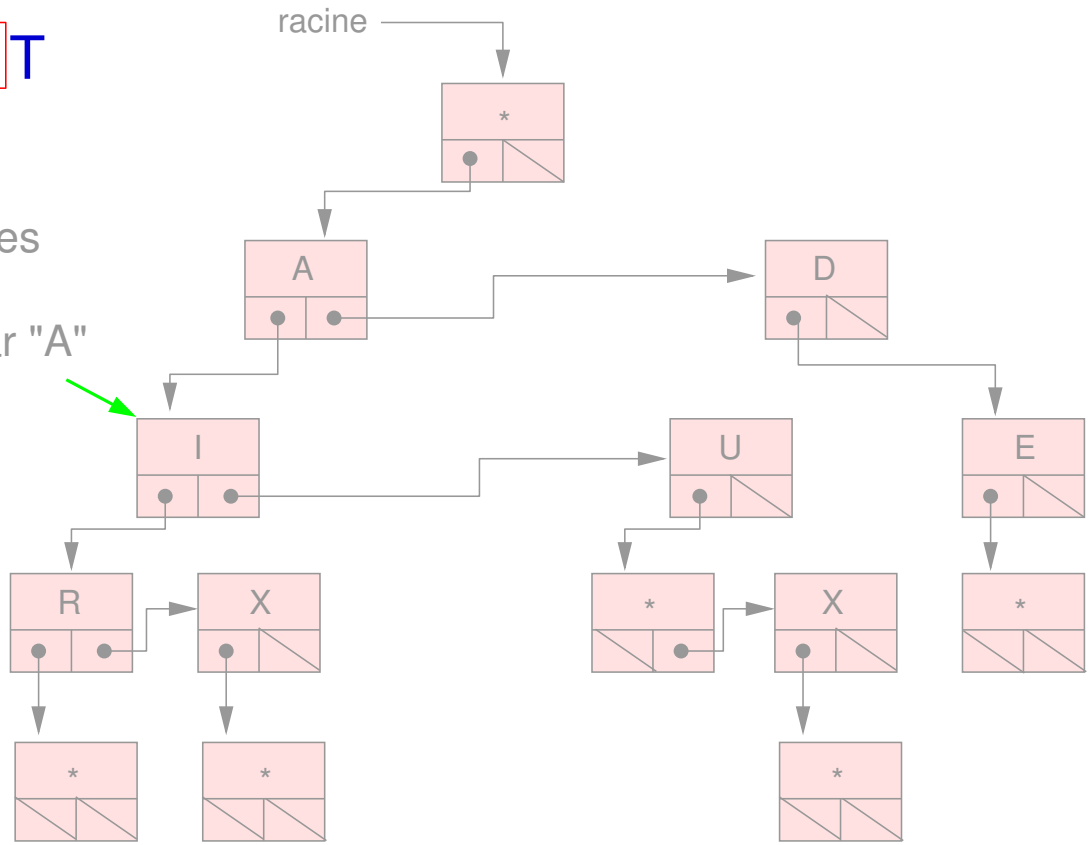
on a trouvé 'A'



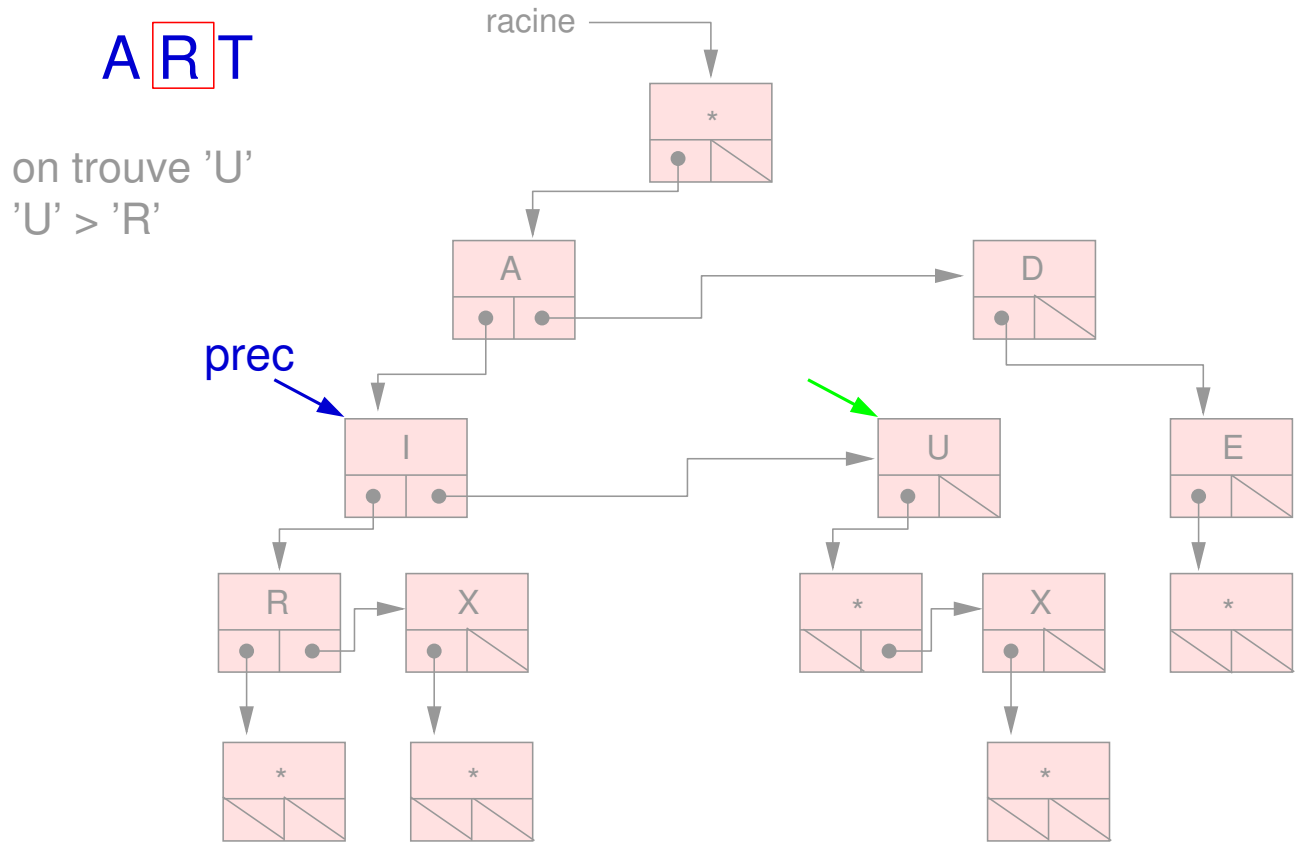
Arbres *n*-aires : arbres lexicographiques

A R T

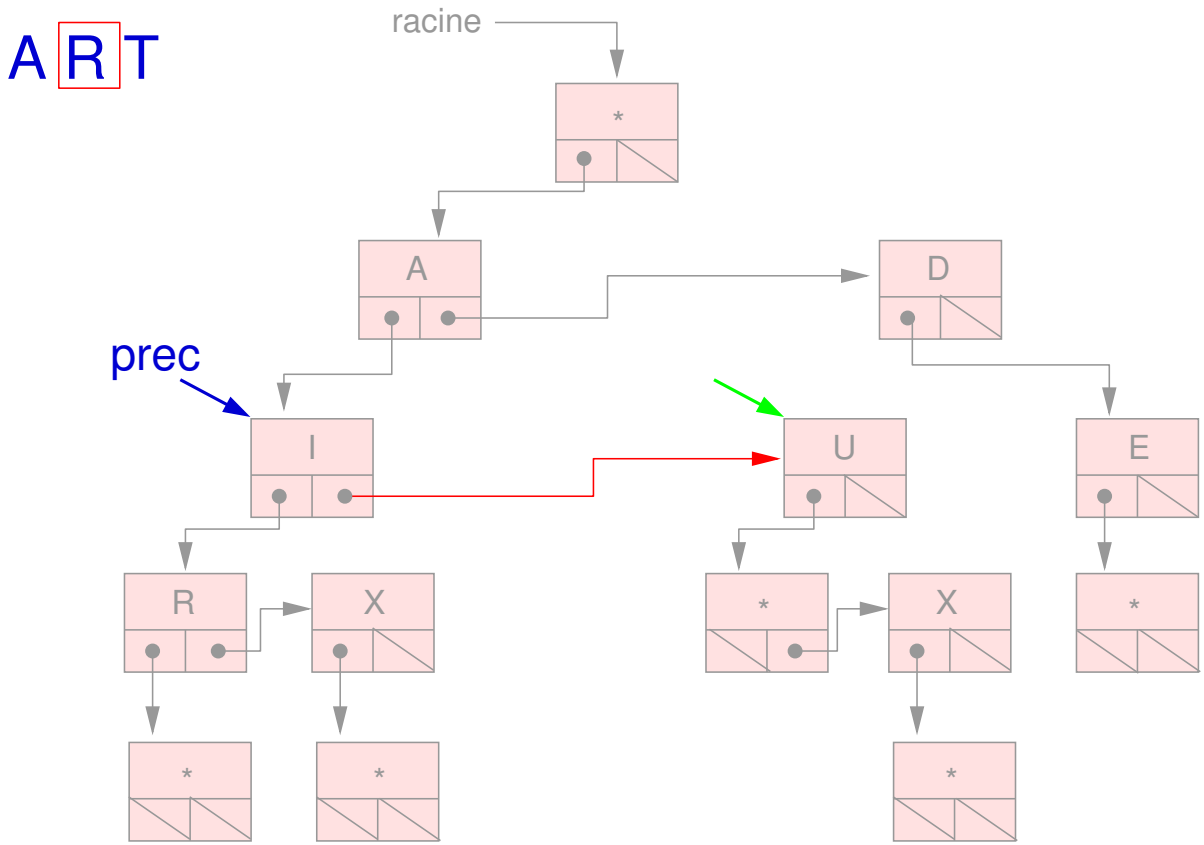
on cherche 'R' :
on parcourt
la liste des 2èmes
lettres des mots
commencant par "A"



Arbres *n*-aires : arbres lexicographiques



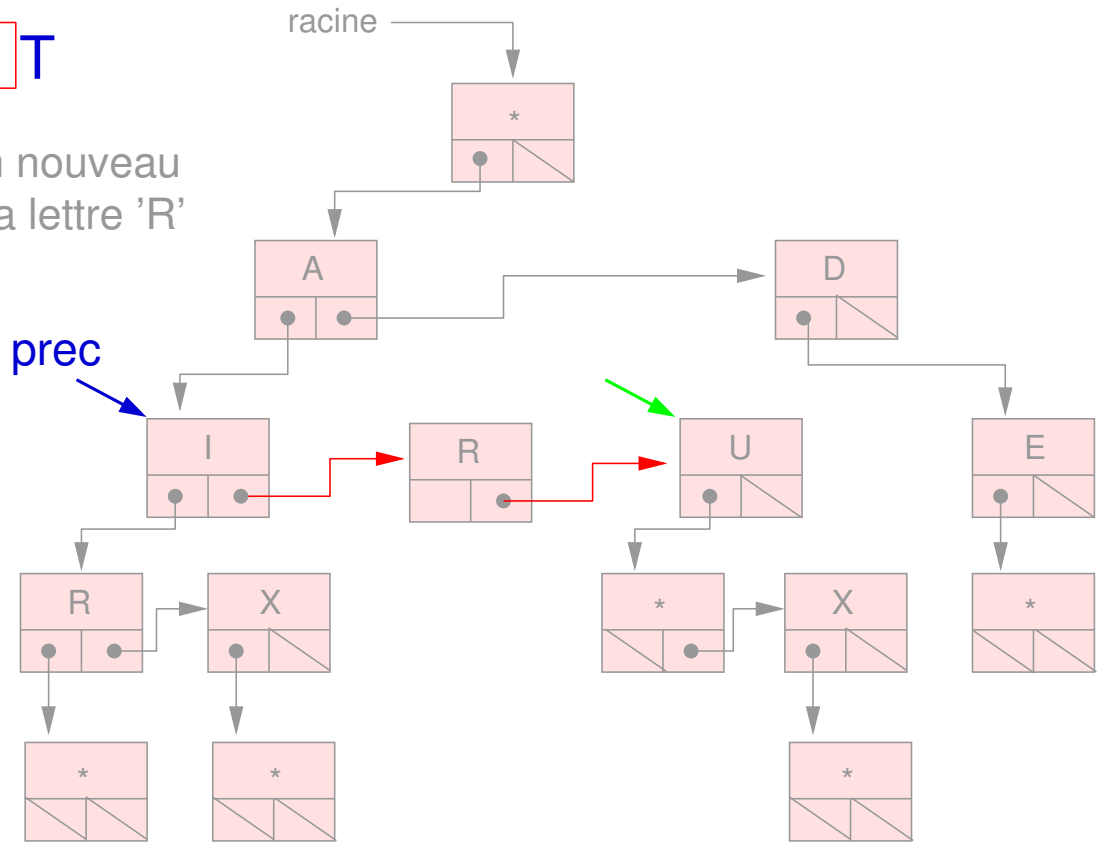
Arbres n-aires : arbres lexicographiques



Arbres *n*-aires : arbres lexicographiques

A R T

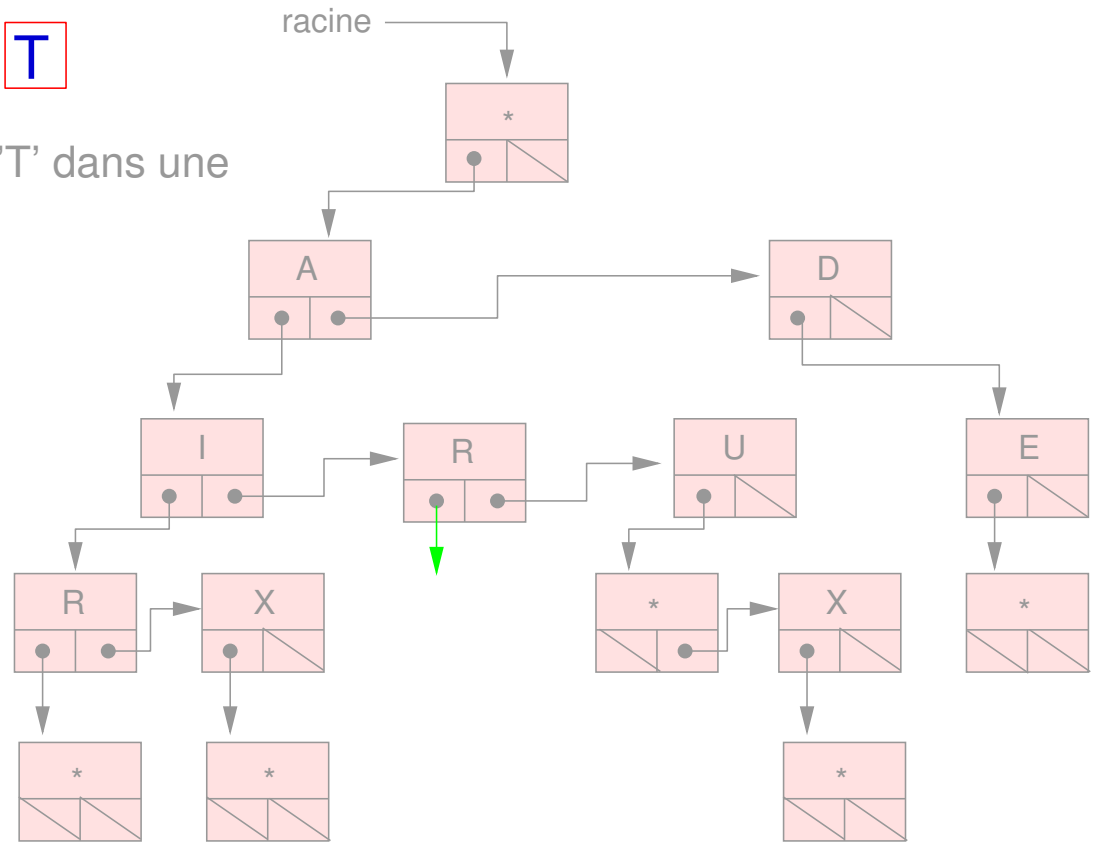
insertion d'un nouveau
noeud avec la lettre 'R'



Arbres *n*-aires : arbres lexicographiques

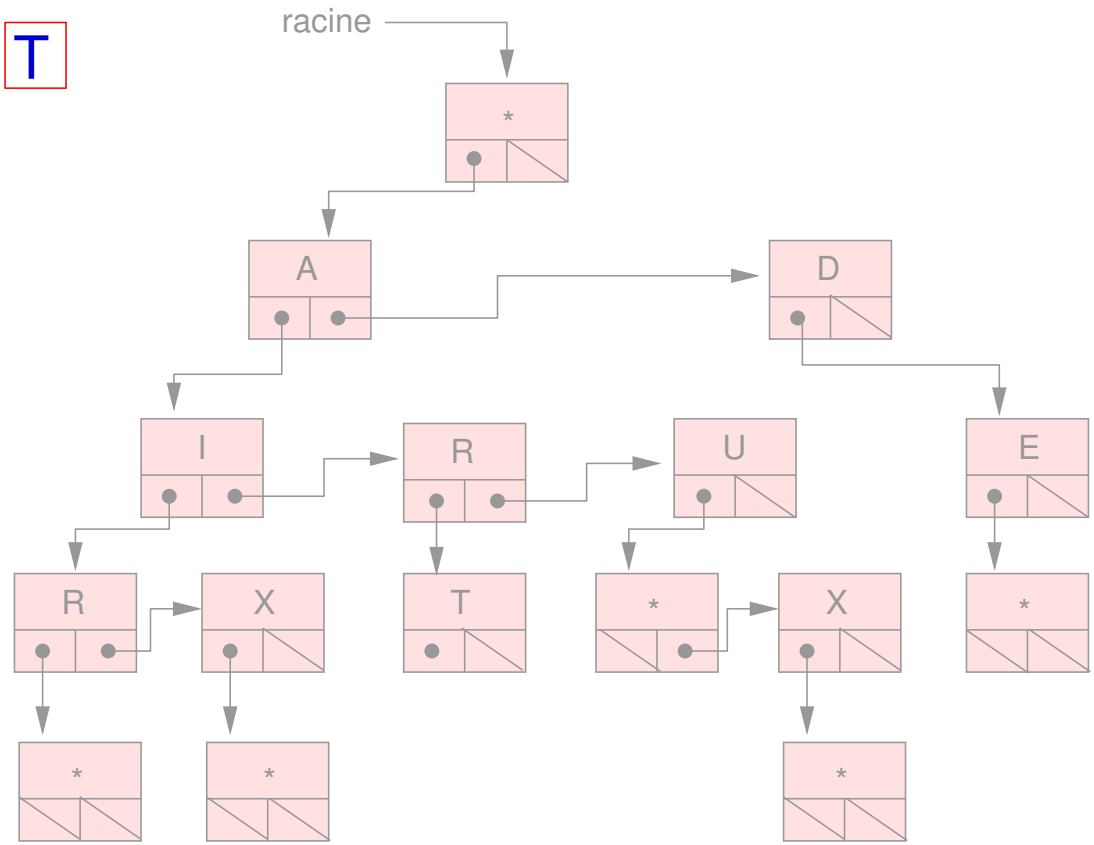
A R T

on cherche 'T' dans une liste vide



Arbres *n*-aires : arbres lexicographiques

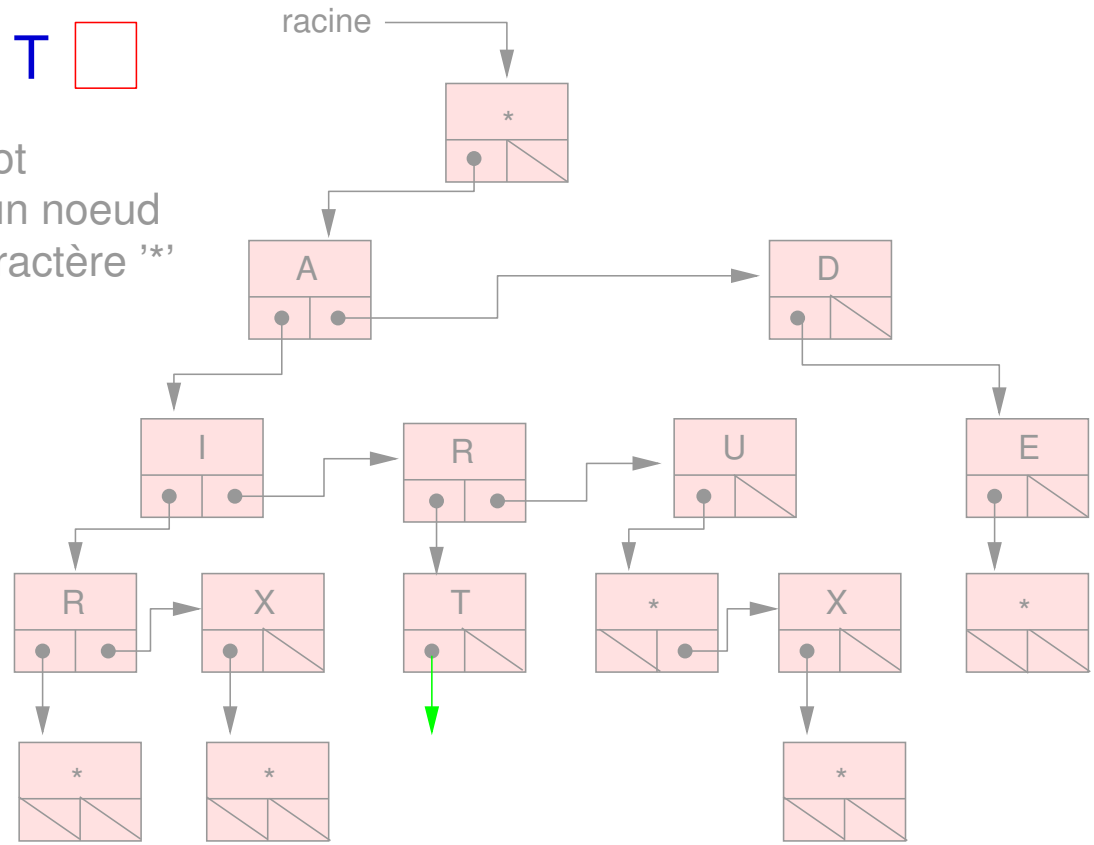
A R T



Arbres *n*-aires : arbres lexicographiques

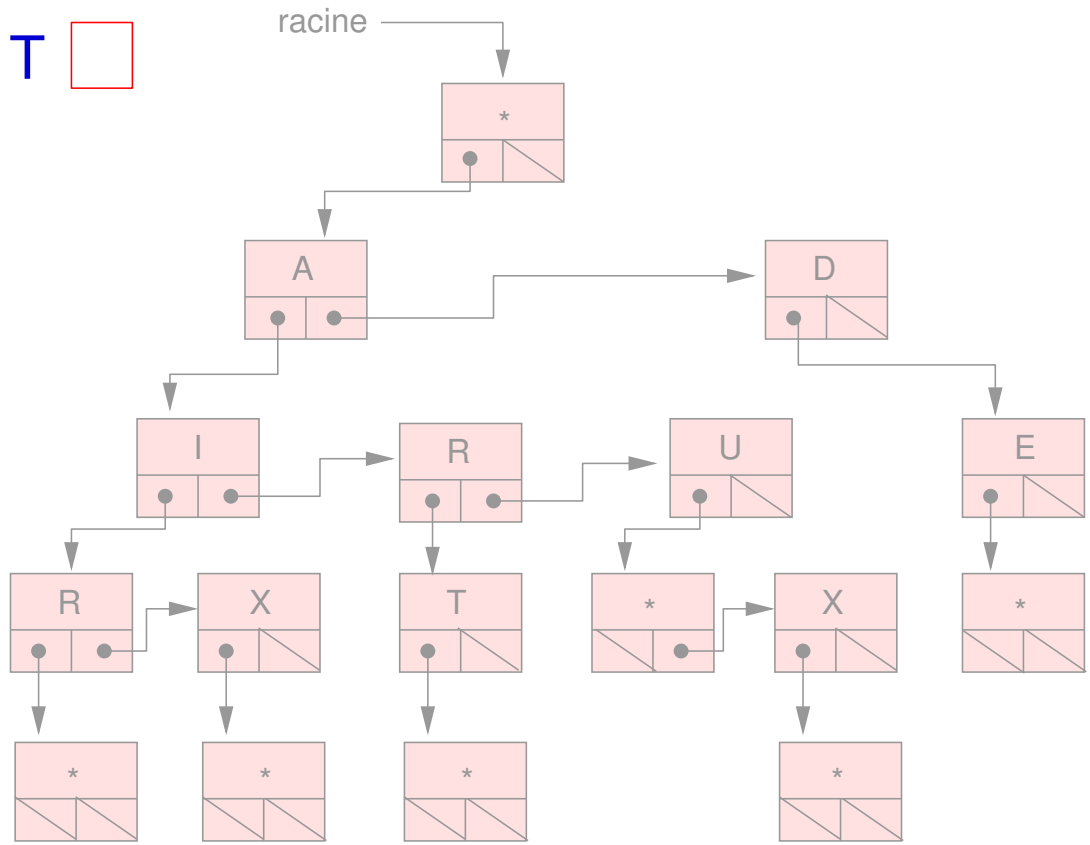
A R T □

à la fin du mot
on cherche un noeud
portant le caractère '*'



Arbres *n*-aires : arbres lexicographiques

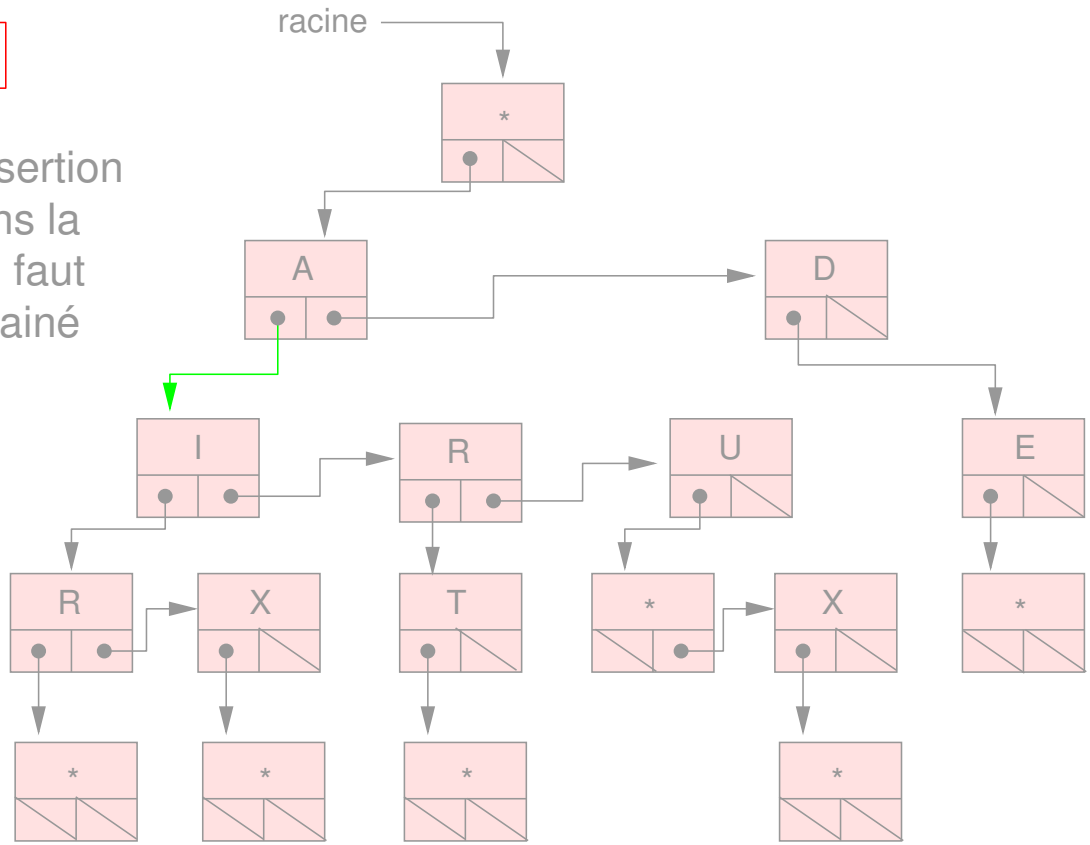
A R T □



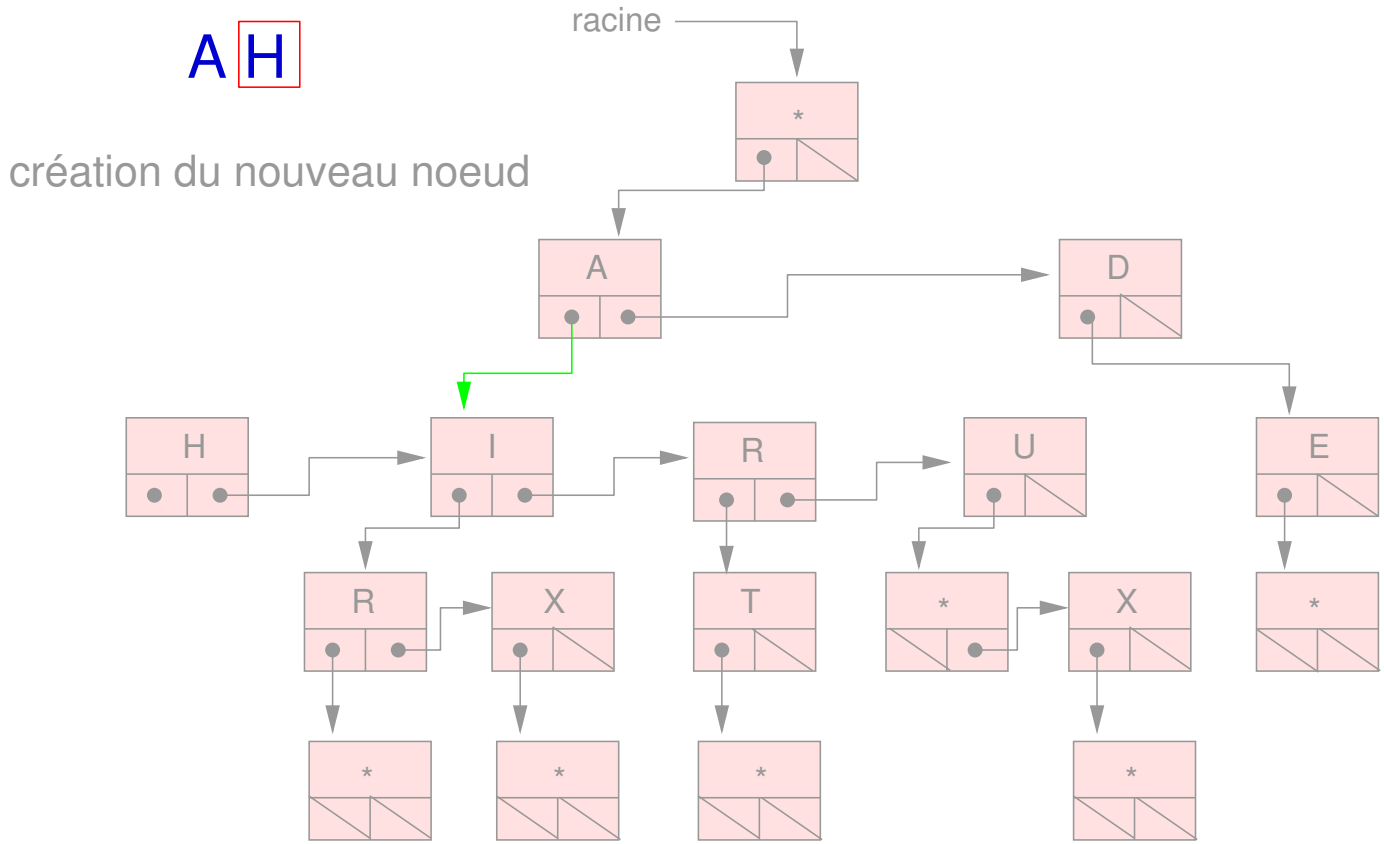
Arbres n-aires : arbres lexicographiques

A H

Attention à l'insertion en premier dans la liste des fils : il faut modifier le fils ainé



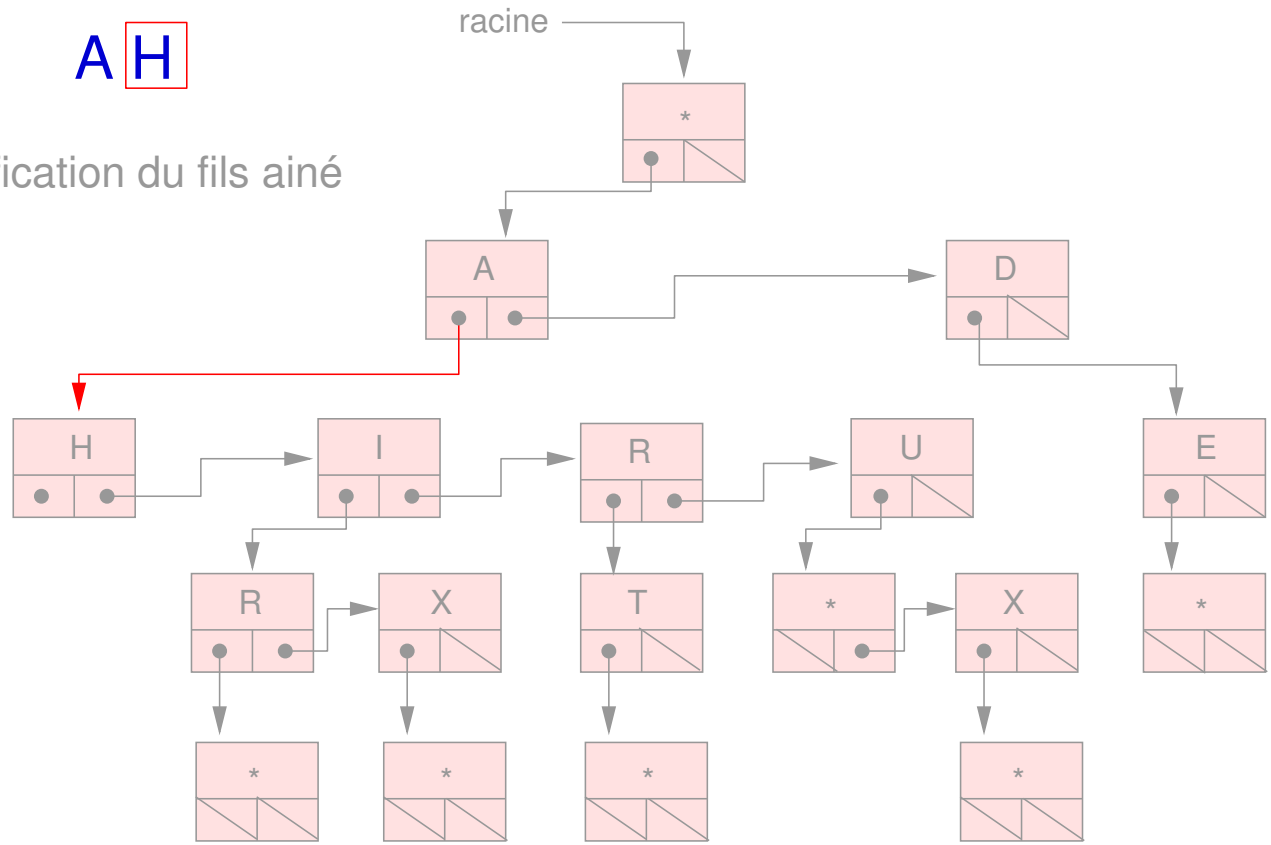
Arbres n-aires : arbres lexicographiques



Arbres *n*-aires : arbres lexicographiques

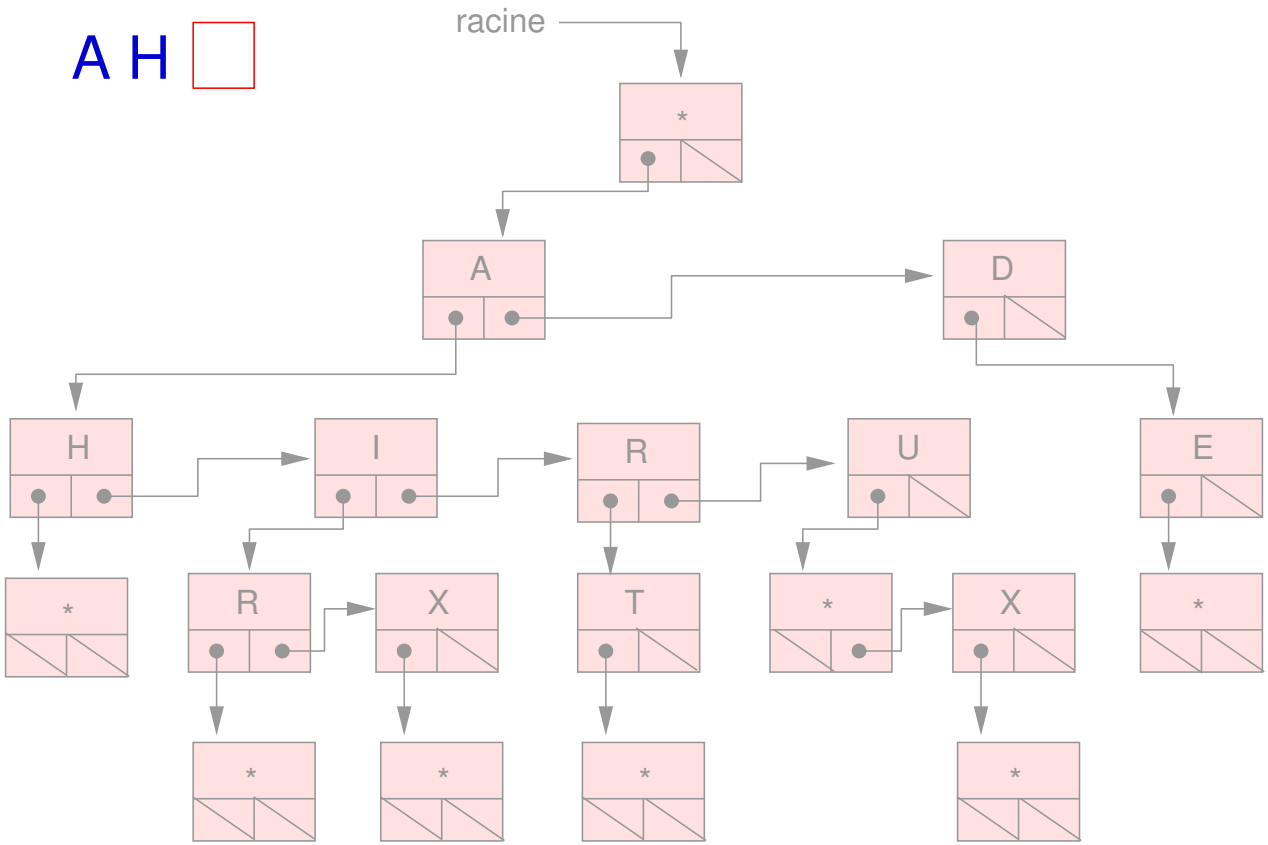
A H

modification du fils ainé



Arbres n-aires : arbres lexicographiques

A H



Arbres *n*-aires : arbres lexicographiques

```
fonction inserer(s, i, p)
   // Principe : chercher s[i] dans la liste des fils de p, s'il ny est pas le créer
   f = filsaine(p),
   prec = None,
   tant que f != None et val(f) < s[i] faire // Ajout dans l'ordre alphabétique
      prec = f,
      f = frere(f),
   fin faire
   si f = None || val(f) > s[i] alors // on n'a pas trouvé s[i] il faut créer un nouveau noeud
      the = new NodeN(s[i], None, f),
      si prec = None alors
         filsaine(p) = the,      // insertion en tête de liste
      sinon
         frere(prec) = the,    // insertion après prec
      finsi
   sinon         // il existe un fils qui porte la valeur s[i]
      the = f,
   finsi
   si s[i] != '*' alors        // si tous les caractères sont insérés (dont '*') on stoppe
      inserer(s, i+1, the),  // sinon on doit insérer la fin du mot en dessous de the
   fin fonction
```