# Arbres

## Algorithmique 1 - 2019-2020

Stéphane Grandcolas

Aix-Marseille Université

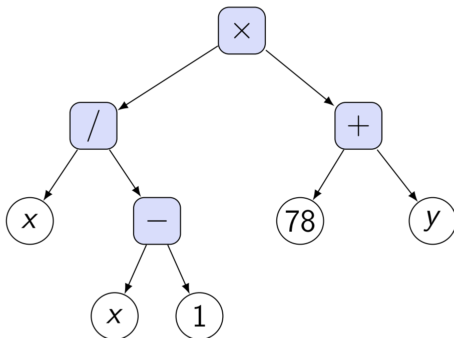
2019-2020

# Arbres

- définitions, vocabulaire,
- arbres binaires,
- abres binaires de recherche,

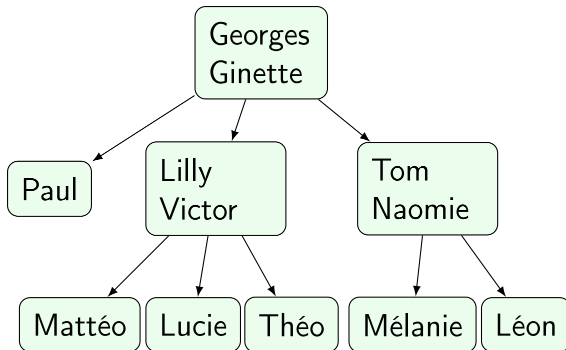
# Arbres

Arbre syntaxique représentant l'exp. arith. $\frac{x}{(x-1)} \times (78 + y)$



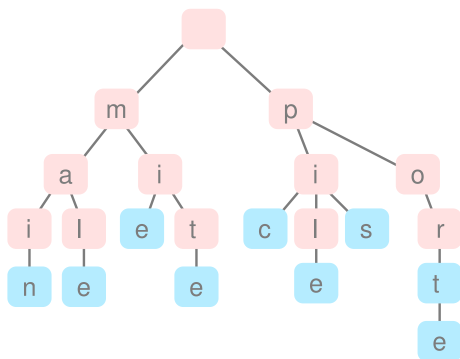
# Arbres

Un arbre généalogique descendant



# Arbres

Un arbre lexicographique, ou arbre en parties communes




main
male
mie
mite
pic
pile
pis
port
porte

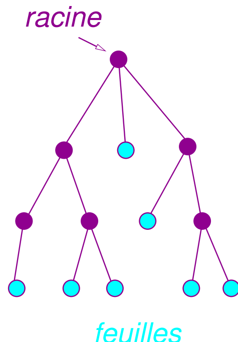
# Arbres : définition 1

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

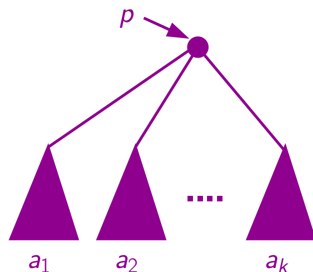


# Arbres : définition 2 (récursive)

Un arbre est constitué

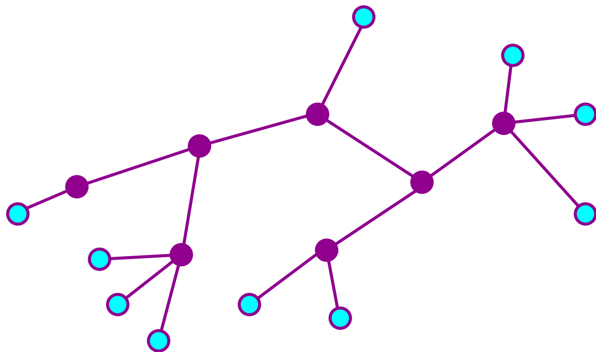
- d'un noeud $p$, sa racine,
- d'une suite de sous-arbres $(a_1, a_2, \ldots, a_k)$.

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



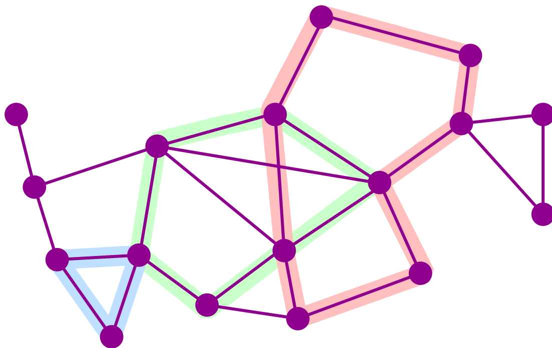
# Arbres : définition 3 (graphes)

Un arbre est un graphe connexe et sans cycle.



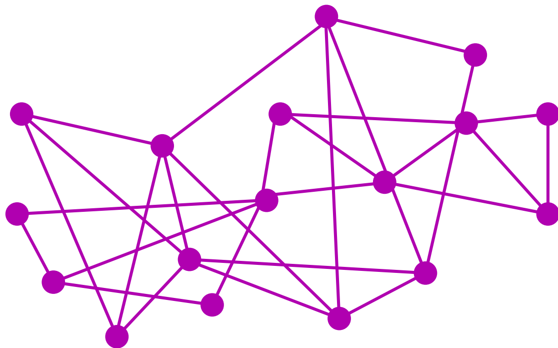
# Arbres : définition 3 (graphes)

Un graphe avec des cycles



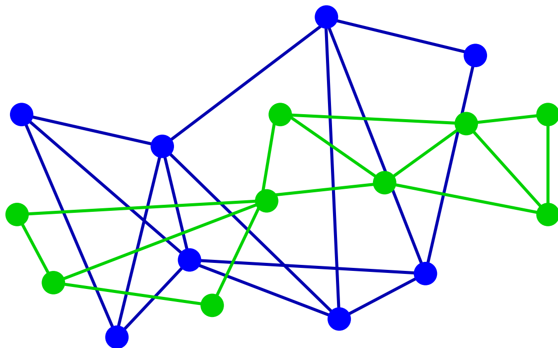
# Arbres : définition 3 (graphes)

Un graphe non connexe



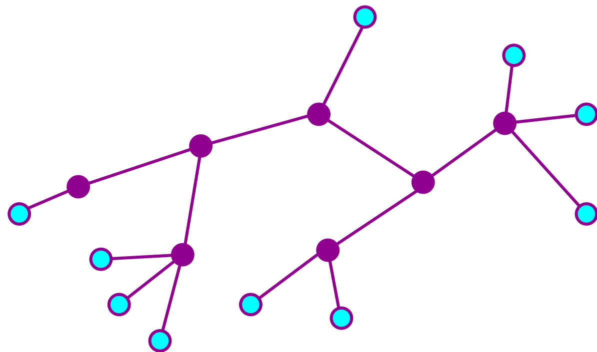
# Arbres : définition 3 (graphes)

Un graphe non connexe

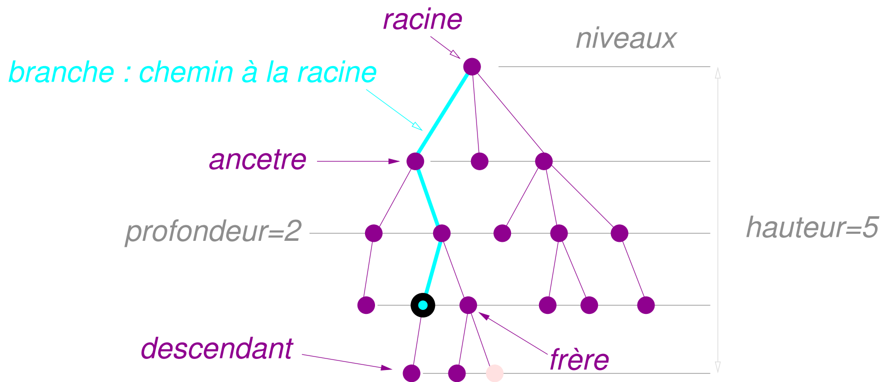


# Arbres : définition 3 (graphes)

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

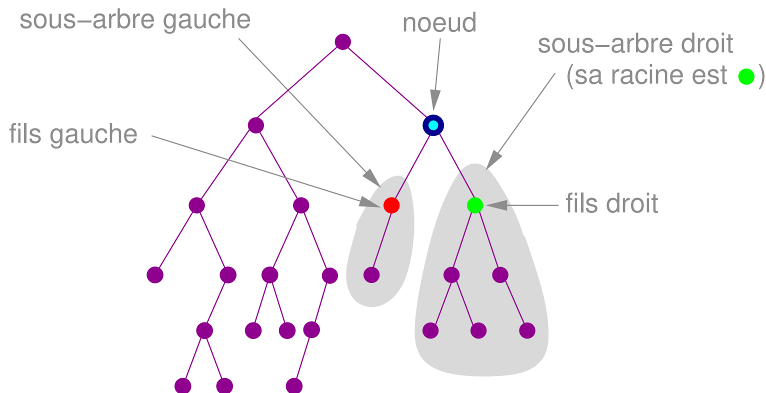


# Arbres : vocabulaire

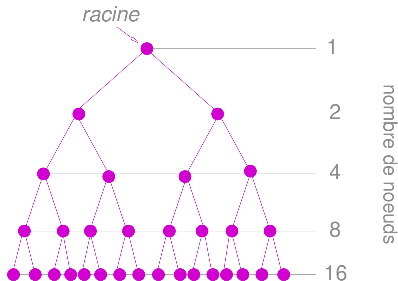


# Arbres binaires

Chaque noeud a au plus 2 fils : le fils gauche et le fils droit



# Arbres binaires




**Arbre binaire complet :**

- 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**

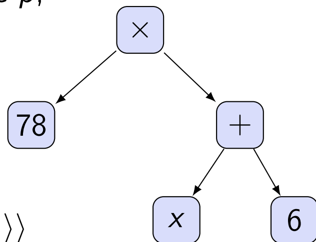
(hauteur = nombre de niveaux)

# Arbres binaires : représentation

**Arbre vide :** $\square$

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

- $x$ : information, étiquette,
- $G = filsG(p)$ : 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

```
public class Node <T> {
   T val;
   Node filsG;
   Node filsD;

   // constructeurs, getters, setters
   ...
}
```

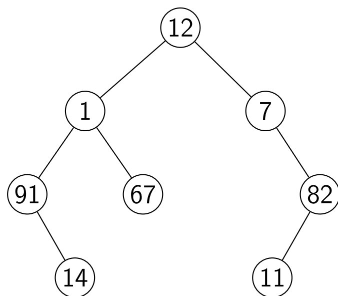
On utilisera `null` pour indiquer qu'il n'y a aucun noeud

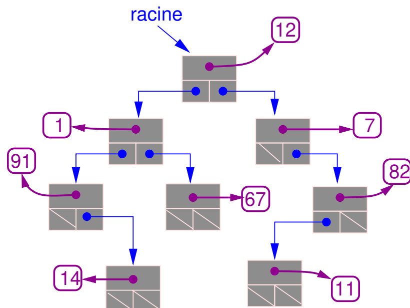
# Arbres binaires : implémentation

```
public class Arbre <T> {
    private Node racine;

    public Arbre() {
        this.racine = null;
    }

    boolean estVide() {
        return (this.racine == null);
    }
}
```

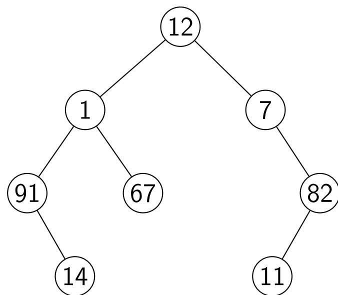
**Encapsulation** : `Arbre` connaît sa racine

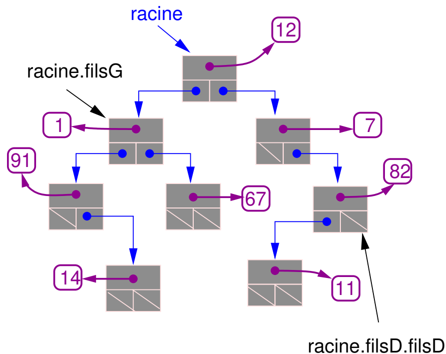
# Arbres binaires : implémentation



# Arbres binaires : implémentation



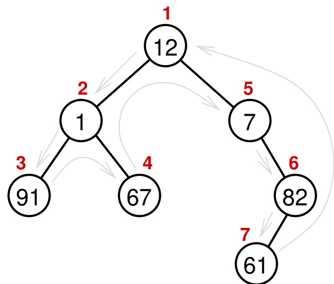
# Arbres binaires : parcours en profondeur

**Principe :** parcourir récursivement le fils gauche puis le fils droit

```
static public void parcours(Node node) {
  if (node != null) {
    // traitement avant
    parcours(node.filsG);
    // traitement entre les deux
       parcours
    parcours(node.filsD);
    // traitement apres
  }
}
```

# Arbres binaires : parcours préfixe

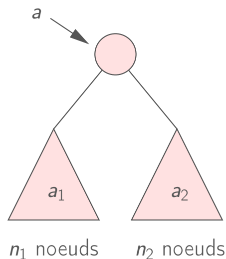
```
static public void parcours(Node
    node) {
  if (node != null) {
    node.printNode();
    parcours(node.filsG);
    parcours(node.filsD);
  }
}
```




12 1 91 67 7 82 61

# Arbres binaires : hauteur minimale

Soit $a$ un arbre de $n$ noeuds : $hauteur(a) \geq \log_2 n$

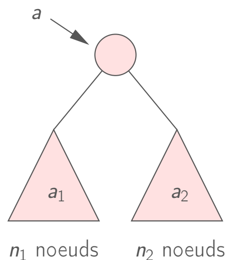


# Arbres binaires : hauteur minimale

Soit $a$ un arbre de $n$ noeuds : $\quad hauteur(a) \geq \log_2 n$

**Preuve.** Par récurrence en utilisant la propriété

$$h(a) = 1 + \max(h(a_1), h(a_2))$$



# Arbres binaires : hauteur minimale

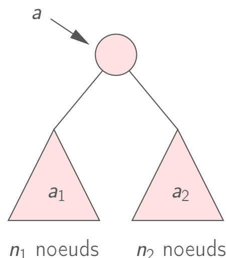
Soit $a$ un arbre de $n$ noeuds : $\quad hauteur(a) \geq \log_2 n$

**Preuve.** Par récurrence en utilisant la propriété

$$h(a) = 1 + \max(h(a_1), h(a_2))$$

Supposons $n_1 \geq n_2$, et donc $n_1 \geq n/2$.

**Hypothèse :** $h(a_1) \geq \log_2 n_1$



# Arbres binaires : hauteur minimale

Soit $a$ un arbre de $n$ noeuds : $\quad hauteur(a) \geq \log_2 n$

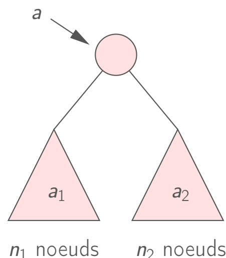
**Preuve.** Par récurrence en utilisant la propriété

$$h(a) = 1 + \max(h(a_1), h(a_2))$$

Supposons $n_1 \geq n_2$, et donc $n_1 \geq n/2$.

**Hypothèse :** $h(a_1) \geq \log_2 n_1$

donc $h(a_1) \geq \log_2 (n/2) = \log_2 n - 1$



# Arbres binaires : hauteur minimale

Soit $a$ un arbre de $n$ noeuds : $hauteur(a) \geq \log_2\ n$

**Preuve.** Par récurrence en utilisant la propriété

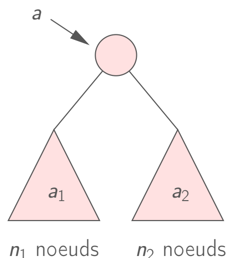
$$h(a) = 1 + \max(h(a_1), h(a_2))$$

Supposons $n_1 \geq n_2$, et donc $n_1 \geq n/2$.

**Hypothèse :** $h(a_1) \geq \log_2\ n_1$

donc $h(a_1) \geq \log_2\ (n/2) = \log_2\ n - 1$

donc $h(a) \geq 1 + h(a_1) \geq 1 + \log_2\ n - 1$,



# Arbres binaires : hauteur minimale

Soit $a$ un arbre de $n$ noeuds : $\quad hauteur(a) \geq \log_2 n$

**Preuve.** Par récurrence en utilisant la propriété

$$h(a) = 1 + \max(h(a_1), h(a_2))$$

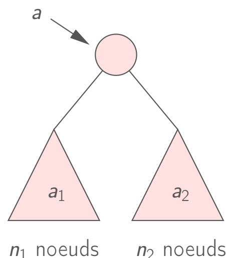
Supposons $n_1 \geq n_2$, et donc $n_1 \geq n/2$.

**Hypothèse :** $h(a_1) \geq \log_2 n_1$

donc $h(a_1) \geq \log_2 (n/2) = \log_2 n - 1$

donc $h(a) \geq 1 + h(a_1) \geq 1 + \log_2 n - 1$,

**Conclusion :** $h(a) \geq \log_2 n$



# Arbres binaires : hauteur minimale

Soit $a$ un arbre de $n$ noeuds : $\quad hauteur(a) \geq \log_2 n$

**Preuve.** Par récurrence en utilisant la propriété

$$h(a) = 1 + \max(h(a_1), h(a_2))$$

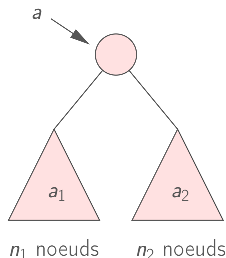
Supposons $n_1 \geq n_2$, et donc $n_1 \geq n/2$.

**Hypothèse :** $h(a_1) \geq \log_2 n_1$

donc $h(a_1) \geq \log_2 (n/2) = \log_2 n - 1$

donc $h(a) \geq 1 + h(a_1) \geq 1 + \log_2 n - 1$,

**Conclusion :** $h(a) \geq \log_2 n$




C'est aussi vrai si on considère $n$ le nombre de feuilles.

# Arbres binaires de recherche

Un **dictionnaire** est un ensemble dynamique d'objets comparables qui supporte les opérations suivantes :

**insérer :** ajout d'une nouvelle valeur

**rechercher :** recherche d'une valeur

**supprimer :** suppression d'une valeur donnée

**Exemples :** liste, tableau, arbre binaire de recherche,...

**Objectifs :** être efficace, utiliser peu d'espace

# Arbres binaires de recherche

Soit $E$ un ensemble muni d'une relation d'ordre total

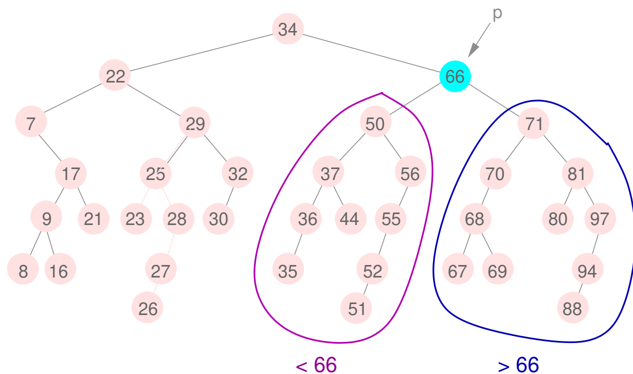
Un arbre binaire étiqueté avec des éléments de $E$ est un arbre binaire de recherche s'il satisfait l'ordre infixe,

i.e. pour tout noeud $p = \langle x, G, D \rangle$

- pour tout noeud $q \in G$, $val(q) < x$,
- pour tout noeud $q \in D$, $val(q) > x$.

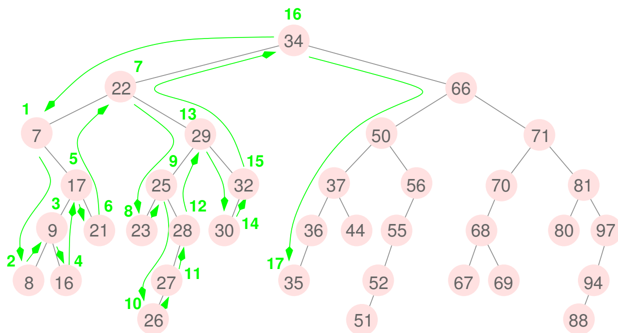
*Les éléments figurant dans l'arbre sont donc tous différents*

# Arbres binaires de recherche



# Arbres binaires de recherche : parcours infixé

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

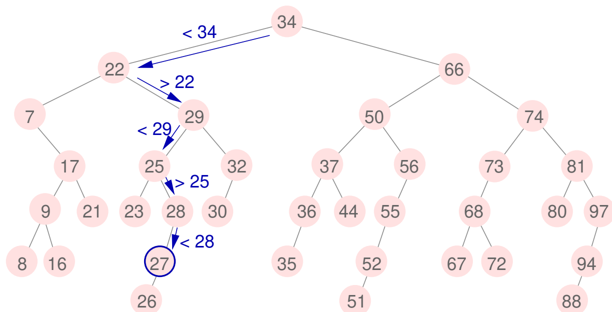



7 8 9 16 17 21 22 23 25 26 27 28 29 30 32 34 35 36 37 ...

# ABR : Recherche d'un élément

Supposons $v \in a$ :

- **si** $e < val(a)$ **alors** $e \in filsG(a)$
- **si** $e > val(a)$ **alors** $e \in filsD(a)$



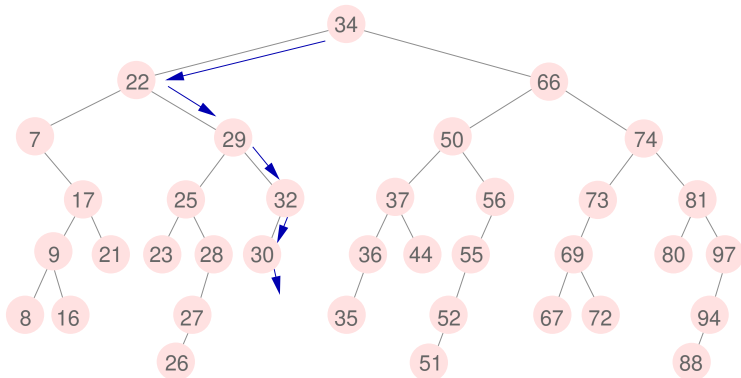

recherche de la valeur 27

# ABR : Recherche d'un élément

```
public static <T extends Comparable<T>> boolean
    contains(Node<T> node, T element) {

  if (node == null)
    return false;

  if (element.equals(node.getVal()))
    return true;

  if (element.compareTo(node.getVal()) < 0)
    return contains(node.filsG, element);

  return contains(node.filsD, element);
}
```

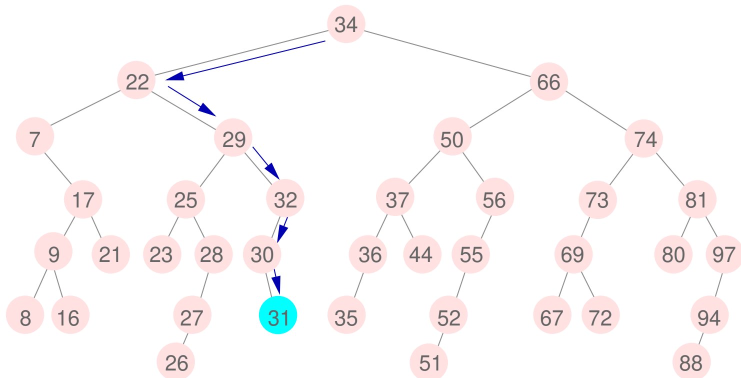
# ABR : ajout d'un élément (insertion)

**Insertion de la valeur 31**



# ABR : ajout d'un élément (insertion)

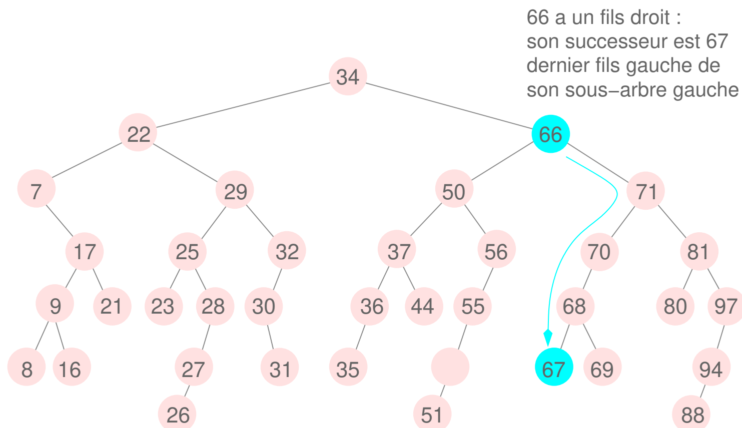
**Insertion de la valeur 31**



# ABR : ajout d'un élément (insertion)

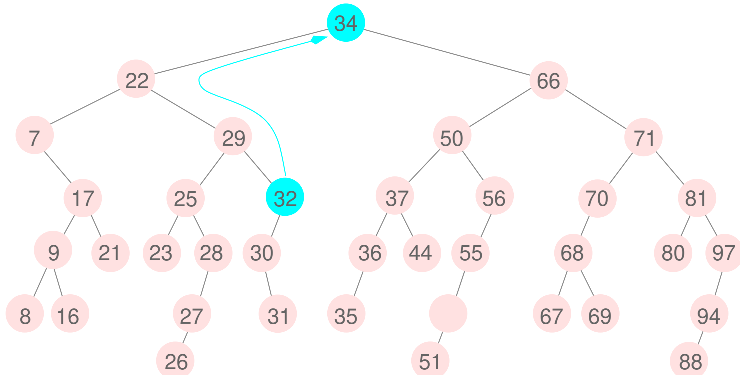
```
public static <T extends Comparable<T>> Node add(T e,
    Node<T> node) {

        if (node == null)
            return new Node<T>(e, null, null);

        if (node.val.compareTo(e) > 0) {
            node.filsG = add(e, node.filsG);
        }
        else {
            node.filsD = add(e, node.filsD);
        }
        return node;
    }
```

# Arbres binaires de recherche : successeur



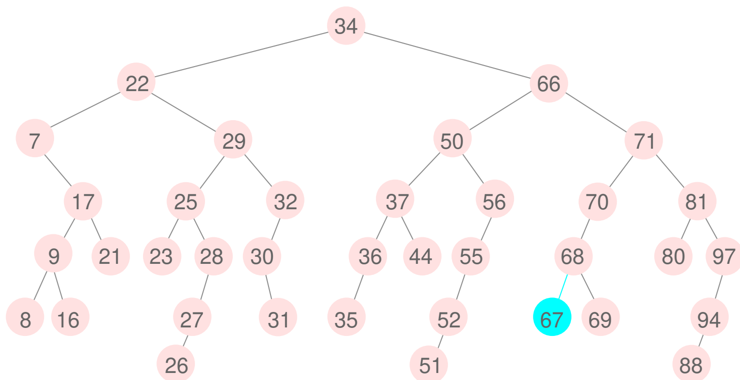
# Arbres binaires de recherche : successeur

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



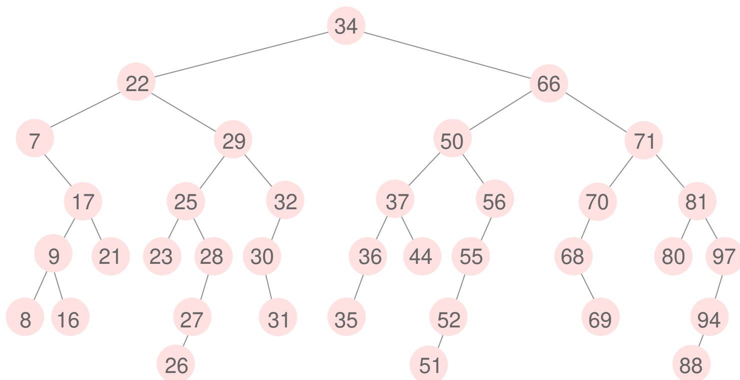
# Arbres binaires de recherche : suppression

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



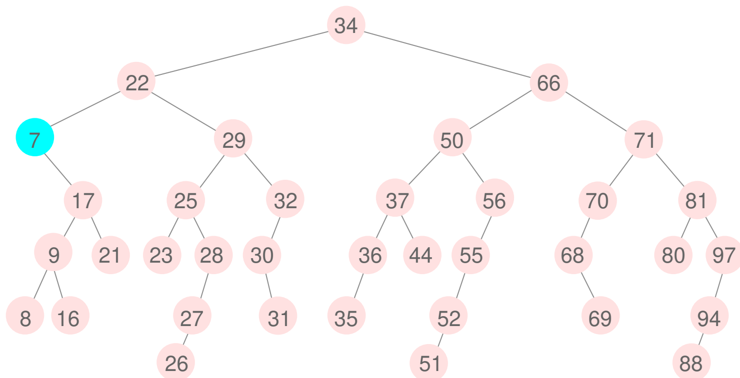
# Arbres binaires de recherche : suppression

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



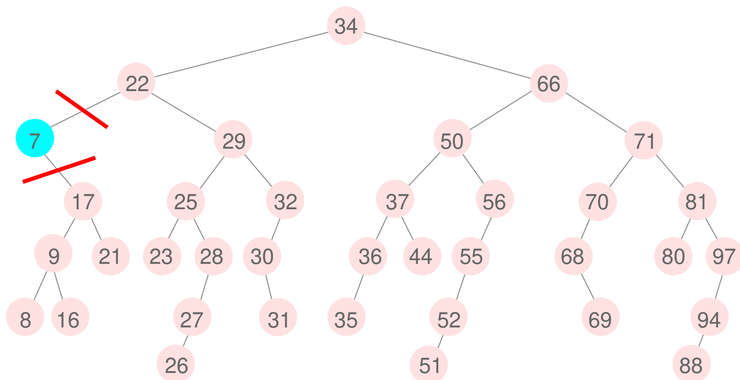
# Arbres binaires de recherche : suppression

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



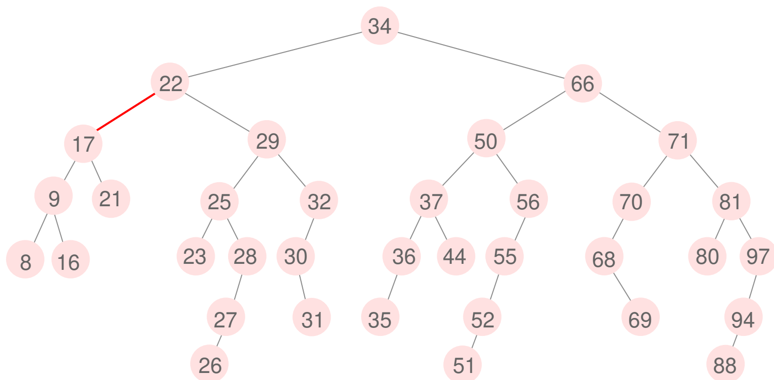
# Arbres binaires de recherche : suppression

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



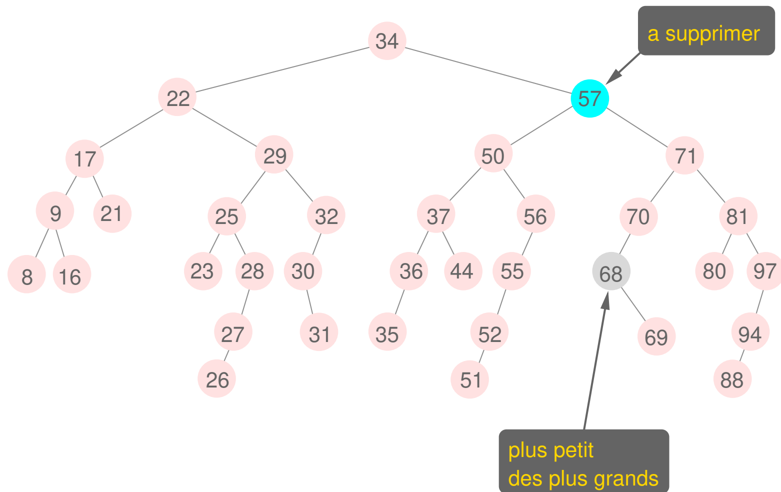
# Arbres binaires de recherche : suppression

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



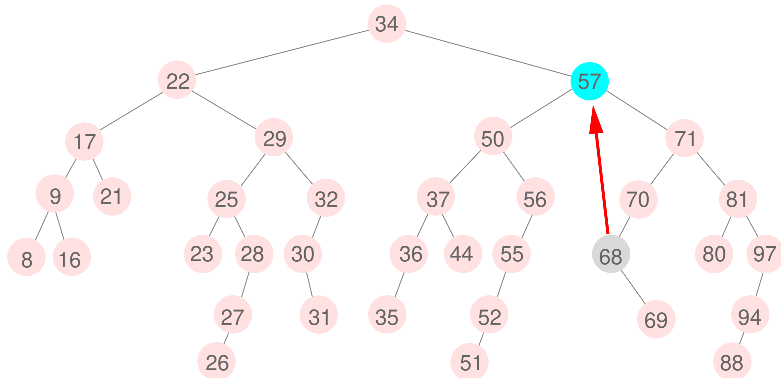
# Arbres binaires de recherche : suppression

**Cas 3 :** le noeud a deux fils



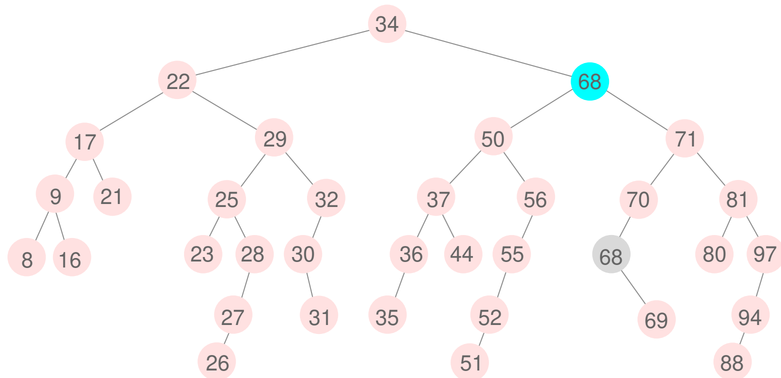
# Arbres binaires de recherche : suppression

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



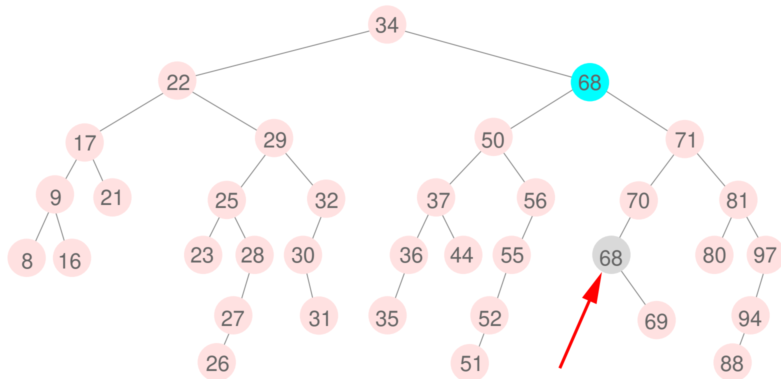
# Arbres binaires de recherche : suppression

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



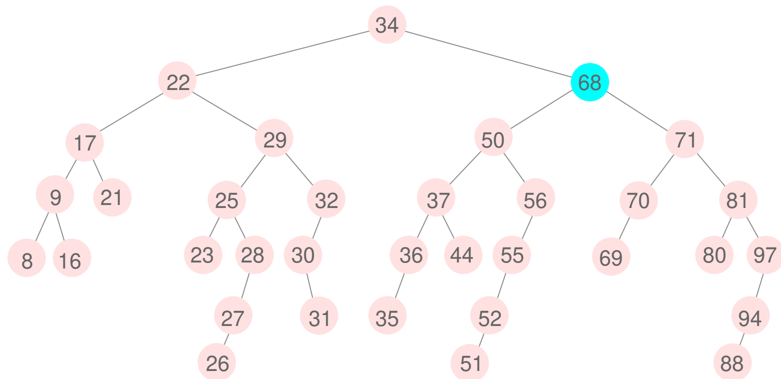
# Arbres binaires de recherche : suppression

**Cas 3** (deux fils) : *décrochage*



# Arbres binaires de recherche : suppression

**Cas 3** (deux fils) : *décrochage*



# Arbres binaires de recherche : suppression

```
public static <T extends Comparable<T>>
                    Node remove(T e, Node<T> node) {
    if (node.val.compareTo(e) > 0)          // e est dans le filsG
        node.filsG = remove(e, node.filsG);
    else if (node.val.compareTo(e) < 0)
        node.filsD = remove(e, node.filsD);
    else if (node.filsG == null)
        return (node.filsD;
    else if (node.filsD == null)
        return (node.filsG;
    else {
        node.val = node.filsD.getMin();
        node.filsD = removeLeftmostNode(node.filsD);
    }
    return node;
}
```

# Arbres binaires de recherche : suppression

```
public static <T extends Comparable<T> >
                              Node remove(T e, Node<T> node) {
    if (node.val.compareTo(e) > 0)
        node.filsG = remove(e, node.filsG);
    else if (node.val.compareTo(e) < 0)     // e est dans le filsD
        node.filsD = remove(e, node.filsD);
    else if (node.filsG == null)
        return (node.filsD;
    else if (node.filsD == null)
        return (node.filsG;
    else {
        node.val = node.filsD.getMin();
        node.filsD = removeLeftmostNode(node.filsD);
    }
    return node;
}
```

# Arbres binaires de recherche : suppression

```
public static <T extends Comparable<T> >
                        Node remove(T e, Node<T> node) {
    if (node.val.compareTo(e) > 0)
        node.filsG = remove(e, node.filsG);
    else if (node.val.compareTo(e) < 0)
        node.filsD = remove(e, node.filsD);
    else if (node.filsG == null)        // node.val = e, pas de filsG
        return node.filsD;
    else if (node.filsD == null)
        return (node.filsG;
    else {
        node.val = node.filsD.getMin();
        node.filsD = removeLeftmostNode(node.filsD);
    }
    return node;
}
```

# Arbres binaires de recherche : suppression

```
public static <T extends Comparable<T>>
                        Node remove(T e, Node<T> node) {
    if (node.val.compareTo(e) > 0)
        node.filsG = remove(e, node.filsG);
    else if (node.val.compareTo(e) < 0)
        node.filsD = remove(e, node.filsD);
    else if (node.filsG == null)
        return node.filsD;
    else if (node.filsD == null)        // node.val = e, pas de filsD
        return node.filsG;
    else {
        node.val = node.filsD.getMin();
        node.filsD = removeLeftmostNode(node.filsD);
    }
    return node;
}
```

# Arbres binaires de recherche : suppression

```
public static <T extends Comparable<T>>
                        Node remove(T e, Node<T> node) {
    if (node.val.compareTo(e) > 0)
        node.filsG = remove(e, node.filsG);
    else if (node.val.compareTo(e) < 0)
        node.filsD = remove(e, node.filsD);
    else if (node.filsG == null)
        return node.filsD;
    else if (node.filsD == null)
        return node.filsG;
    else {              // node.val = e, deux fils
        node.val = node.filsD.getMin();
        node.filsD = removeLeftmostNode(node.filsD);
    }
    return node;
}
```

# Arbres binaires de recherche : suppression

```
public static Node removeLeftmostNode(Node root) {

    Node pere = null;
    Node p = root;

    while (p.filsG != null) {
        pere = p;
        p = p.filsG;
    }
    if (pere == null)
        return root.filsD;

    pere.filsG = p.filsD;
    return root;
}
```

# 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 :

$\log n$ dans le meilleur des cas, $n$ dans le pire des cas