ALGORITHMES ET STRUCTURES DE DONNÉES

IFT-2008/GLO-2100

Chapitre 8 : Arbres binaires de recherche

Thierry Eude, Kim Rioux-Paradis

Département d'informatique et de génie logiciel

UNIVERSITÉ LAVAL

Arbres binaires de recherche (de tri)

- Peuvent être utilisés pour implémenter les dictionnaires.
- Dictionnaire : conteneur d'éléments devant supporter efficacement les opérations suivantes:
  - Insertion d'un élément
  - Suppression d'un élément
  - Trouver un élément (déterminer si un élément est présent)
- s'effectuent en $O(\log n)$ lorsque l'arbre de n éléments est équilibré.

Arbres binaires de recherche (de tri)

- Ce chapitre portera entièrement sur les arbres binaire de recherche qui supportent efficacement ces opérations en plus de conserver les éléments dans un ordre.
  - L'affichage des éléments en ordre croissant des valeurs de clé se fait à l'aide du parcours en-ordre.
- Nous verrons en détail les arbres AVL
  - Arbres qui restent toujours équilibrés grâce à l'exécution de certaines opérations (rotations) lors des insertions et des suppressions d'éléments

Implantation d'un arbre binaire par chaînage

```
template <typename E>
class Arbre
{
public:
        //méthodes publiques
private:
        // classe Nœud
        class Noeud
        { public:
                E data; // la donnée est ici la clé
                Noeud *gauche;
                Noeud *droite;
                int hauteur;
                Noeud(const E & d): gauche(0),data(d),droite(0),hauteur(0) { }
        };
        // Les membres données
        Noeud * racine; //racine de l'arbre
        long cpt; // Nombre de noeuds dans l'arbre
        // méthodes privées
};
```

Modèle d'implantation par chaînage

Implantation par chaînage

- Puisque chaque nœud possède au maximum deux nœuds fils, on maintient un pointeur sur chacun d'eux.
- Avantages:
  - La taille de l'arbre est dynamique.
  - Facile d'opérer sur des pointeurs.
- Inconvénients:
  - On doit éviter les fuites de mémoire et les doubles références.
  - Le parcours explicite de l'arbre ne se fait que de la racine vers les feuilles.
    - Mais il est possible de remonter vers la racine grâce aux retours d'appels récursifs.

Identifiant des éléments d'un arbre de tri

- Attribut data (de type E) de la classe Nœud
  - utilisé pour identifier les éléments dans l'arbre et appliquer la propriété d'ordonnancement de l'arbre de tri.
  - définit la clé (ie. l'identifiant) de l'élément
- Donc, pour tout nœud de l'arbre de tri, sa clé (champ data) doit être:
  - > aux clés des noeuds de son sous arbre de gauche
  - < aux clés des nœuds de son sous arbre de droite
- L'opérateur < doit être surchargé pour le type E.
  - C'est donc operator< du type E qui définit l'ordre utilisé par l'arbre de tri.

Identifiant des éléments d'un arbre de tri

- Conteneur associatif: À la place du seul attribut data de la classe nœud, il est possible d'en avoir deux: un attribut clé (utilisé pour l'ordonnancement des éléments) et un attribut valeur (associée à la clé).
  - ➢ les éléments sont identifiés par l'attribut clé.

Éléments équivalents (duplicatas)

- Deux éléments a et b sont considérés comme équivalents lorsque nous avons ! $(a < b)$ et ! $(b < a)$.
- La présence d'éléments équivalents (duplicatas), est habituellement interdit dans les arbres binaires de recherche.
  - Cas des conteneurs set et map de la STL.
  - C'est ce que nous supposerons dans ce chapitre.
- quand même possible de permettre la présence de duplicatas

Éléments équivalents (duplicatas)

- Moyens possibles:
  - En ajoutant un autre attribut dans la classe nœud qui indique la fréquence de cet élément.
  - En ajoutant un autre conteneur (ex: liste) pour les éléments équivalents
  - En modifiant la définition d'un arbre de tri comme suit:
    - ✓ Toute clé d'un nœud doit être ≥ aux clés de son sous arbre gauche et ≤ aux clés de son sous arbre droit.
    - ✓ Mais contribue à augmenter la profondeur de l'arbre et le temps d'exécution pour les opérations sur le conteneur…

Implantation d'un arbre binaire par chaînage

```
template <typename E>
class Arbre
{
public:
        //Constructeurs
        Arbre();
        Arbre(const Arbre& source) ;

        //Destructeur
        ~Arbre();

        //Les fonctions membres
        bool estVide() const;
        const E& max() const;
        const E& min() const;
        //…
}
```

Implantation d'un arbre binaire par chaînage

```
template <typename E>
class Arbre
{
public:
        //Les fonctions membres
        int nbNoeuds() const;
        int hauteur() const;
        bool appartient(const E&) const ;
        void inserer(const E&);
        void insererAVL(const E&);
        void enleverAVL(const E&);
        std::vector<E> parcoursSymetrique() const;
        //..
}
```

Implantation d’un arbre binaire par chaînage

```
template <typename E>
class Arbre
{
private:
        //Attributs internes de Arbre
        Noeud* racine; //racine de l'arbre
        long nb_noeuds;

        //.. Les membres méthodes privés (les auxiliaires récursifs)
        void _auxInserer( Noeud* &, const E&);
        void _auxInsererAVL(Noeud* &, const E &);
        void _auxEnleverAVL( Noeud* &, const E&);
        Noeud* _auxAppartient(Noeud* arbre, const E &) const;
}
```

Implantation d'un arbre binaire par chaînage

```
template <typename E>
class Arbre
{
private:
      // Les membres privés propres aux arbres AVL
      int _hauteur(Noeud*) const;
      void _balancer(Noeud* &);
      void _zigZigGauche(Noeud* &);
      void _zigZigDroit(Noeud* &);
      Void _zigZagGauche(Noeud* &);
      void _zigZagDroit(Noeud* &);
      //..
}
```

Parcours en-ordre pour un arbre binaire implémenté par chaînage

```
std::vector<E> parcoursSymetrique() const
{
	std::vector<E> parcours; //pour stocker les résultats du parcours
	parcours.reserve(nb_noeuds); //pour éviter une ré-allocation du tableau
	_parcoursSymetrique(racine, parcours);
	return parcours;
}
```

Parcours en-ordre pour un arbre binaire implémenté par chaînage

```
template <typename E>
void Arbre<E>::_parcoursSymetrique(Noeud* arb, std::vector<E>& parcours)
{
	if (arb!=0)
	{
		_parcoursSymetrique(arb->gauche, parcours); // appel récursif
		parcours.push_back(arb->data); // traitement
		_parcoursSymetrique(arb->droite, parcours); // appel récursif
	}
}
```

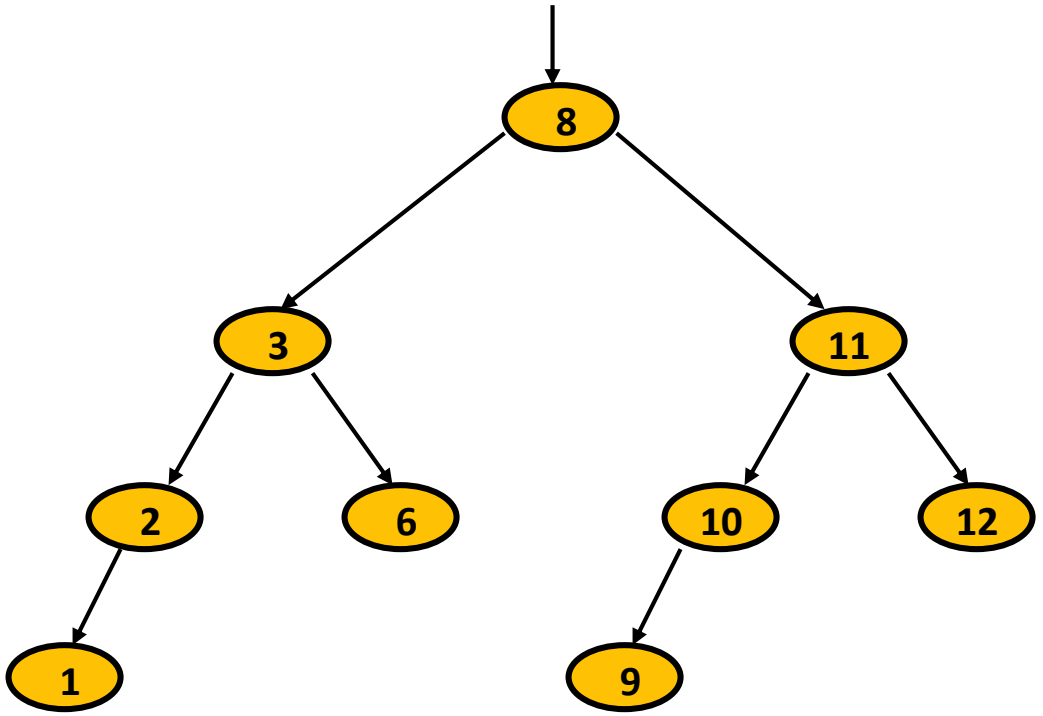
Insertion dans un arbre de tri (non AVL)

```
void inserer(const E&)
{
        _auxInserer(racine, data);
}
```

Insertion dans un arbre de tri (non AVL)

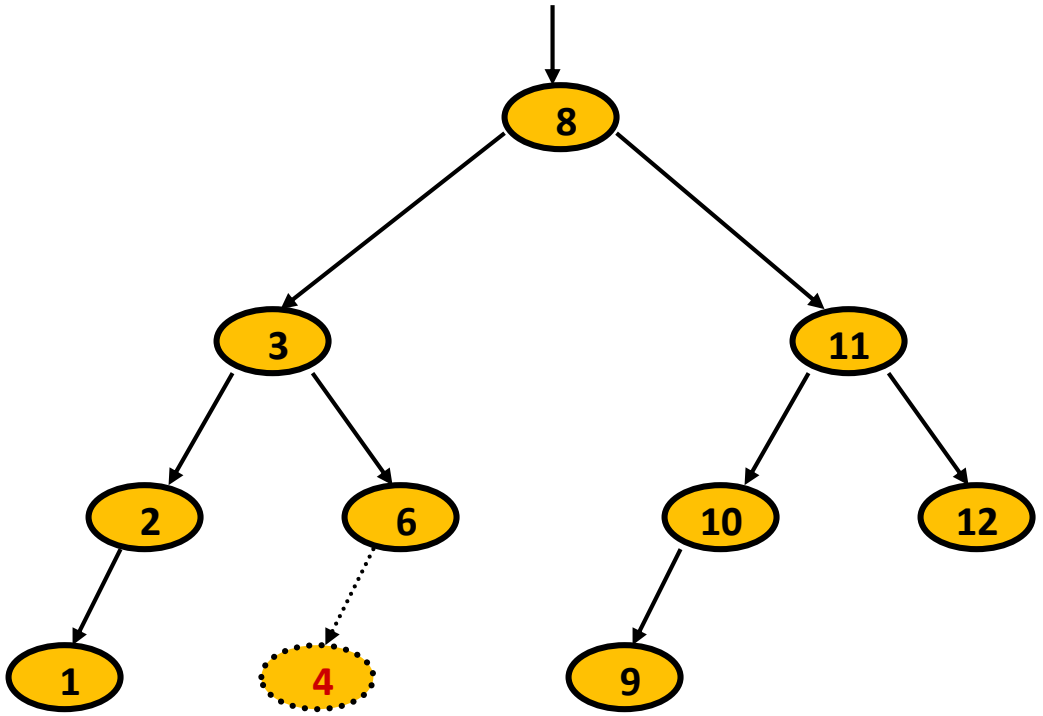
```
void _auxInsererAVL(Noeud* & arbre, const E &)
{
	if (arbre == 0)
	{
		arbre = new Noeud(data);
		nb_noeuds++;
		return;
	}
	else if( data < arbre->data )
		_auxInserer(arbre->gauche, data); //appel récursif
	else if( arbre->data < data )
		_auxInserer(arbre->droite, data); //appel récursif
	else
		throw logic_error("Les duplicatats sont interdits");
}
```

Ajout d'éléments dans un arbre de tri



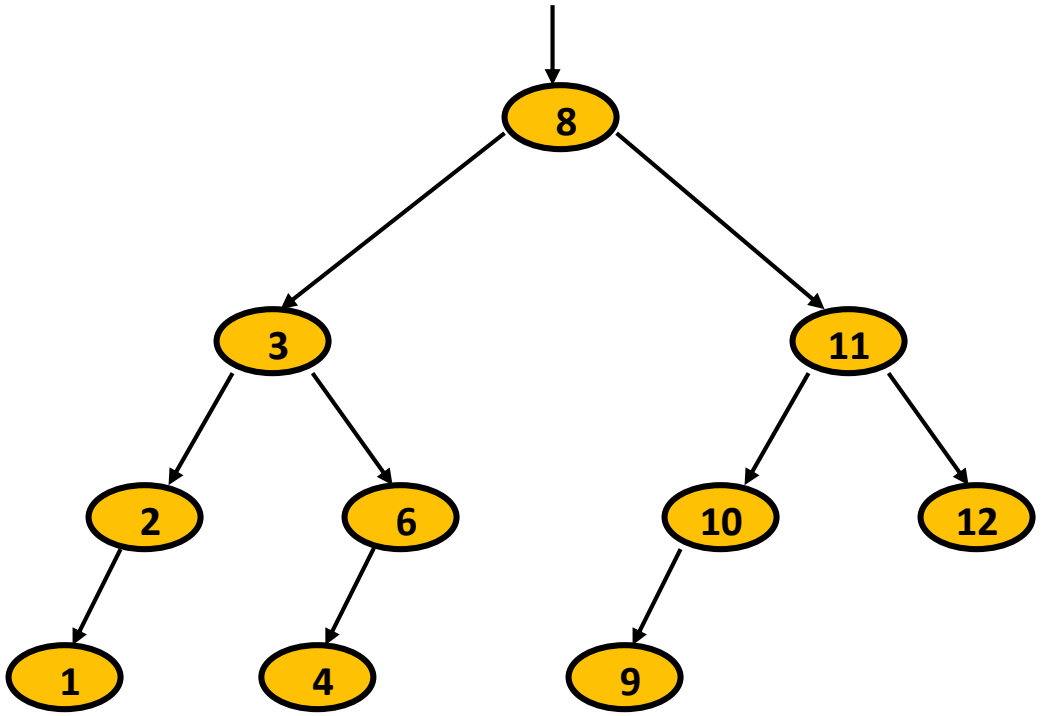
8,3,11,2,1,6,10,9,12,4,5,14

Ajout d'éléments dans un arbre de tri



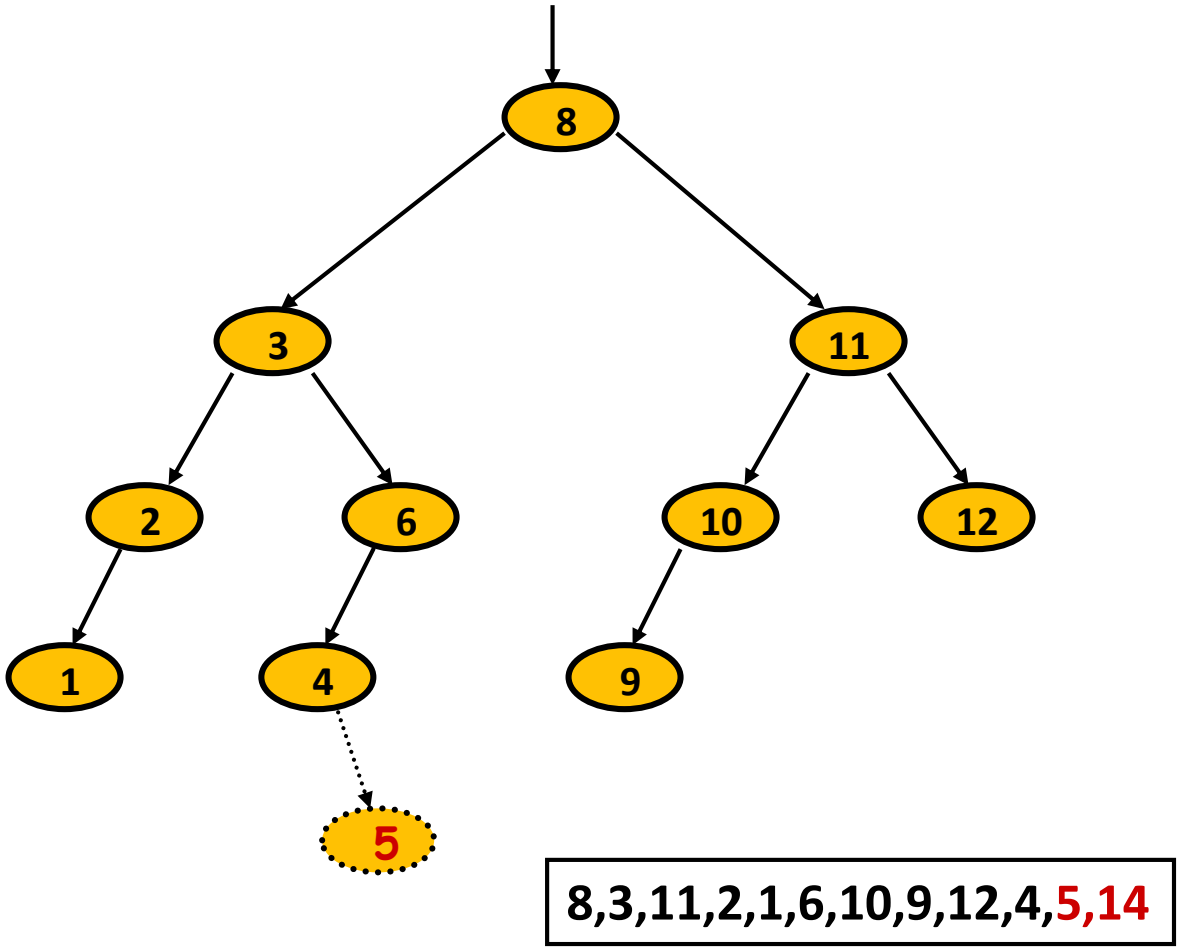
8,3,11,2,1,6,10,9,12,4,5,14

Ajout d'éléments dans un arbre de tri

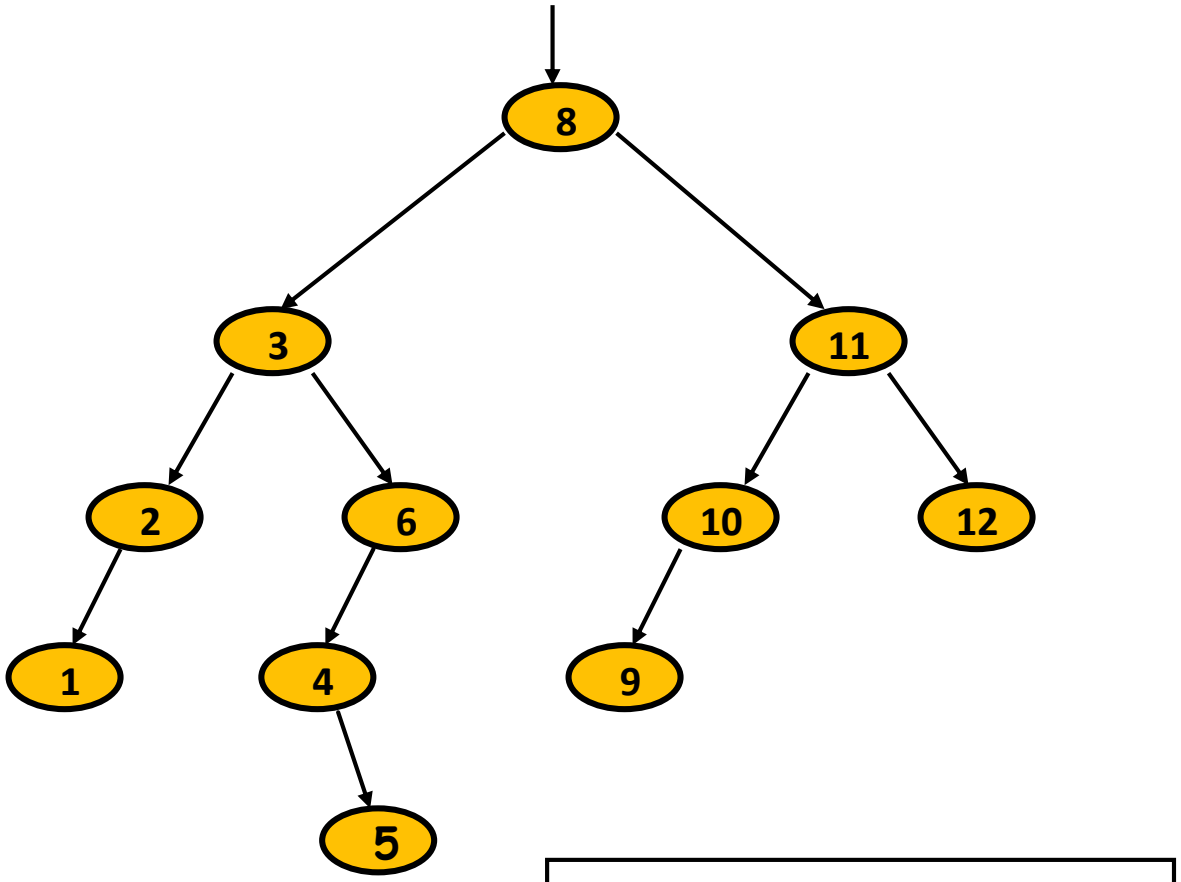


8,3,11,2,1,6,10,9,12,4,5,14

Ajout d'éléments dans un arbre de tri

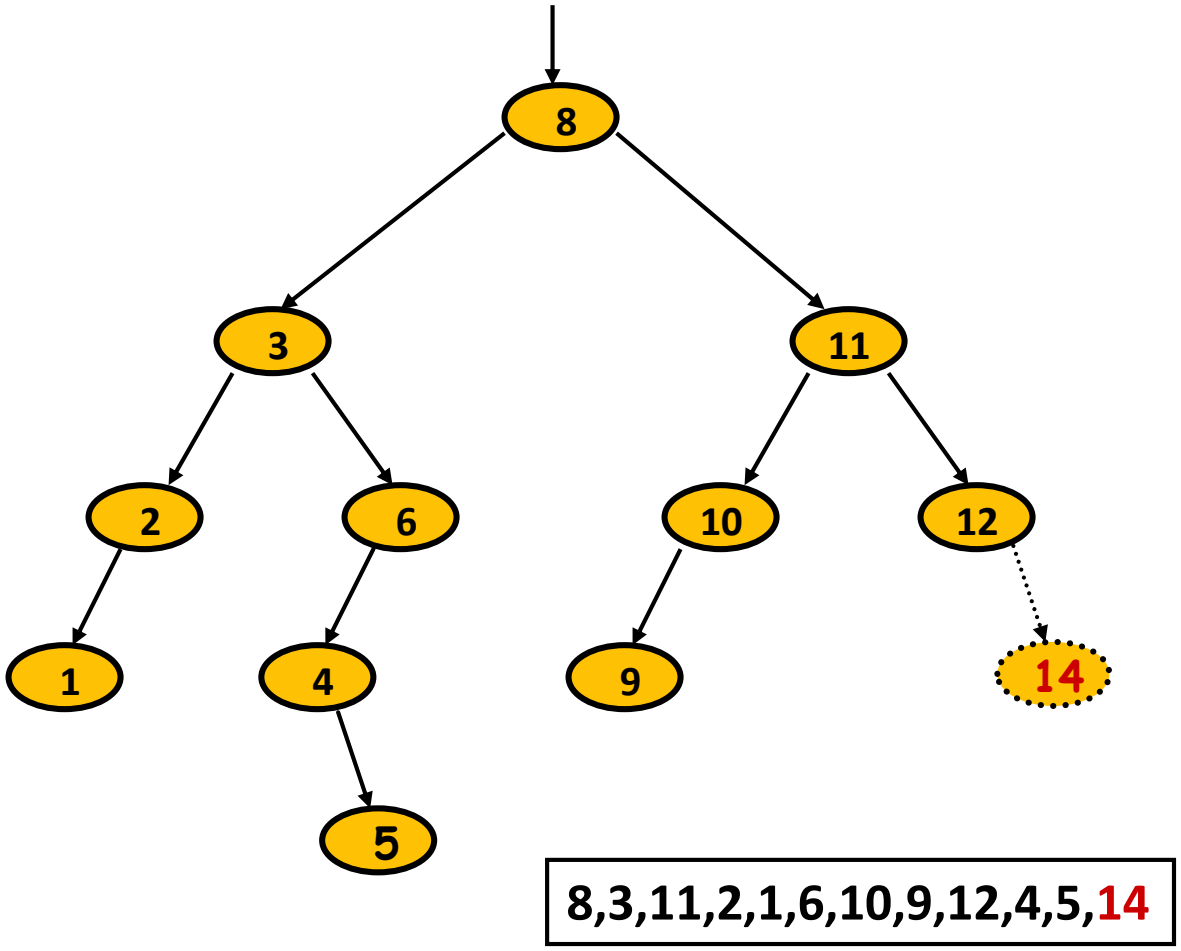


Ajout d'éléments dans un arbre de tri

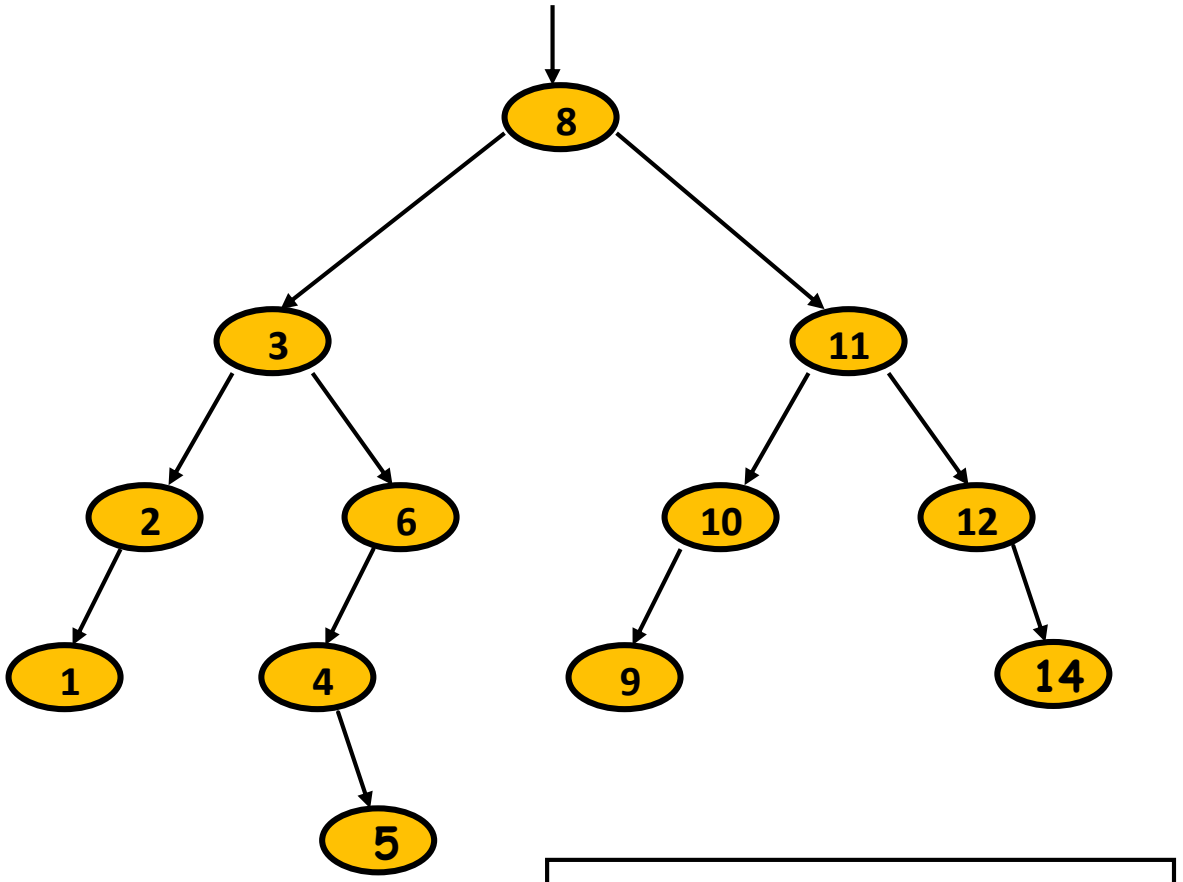


8,3,11,2,1,6,10,9,12,4,5,14

Ajout d'éléments dans un arbre de tri



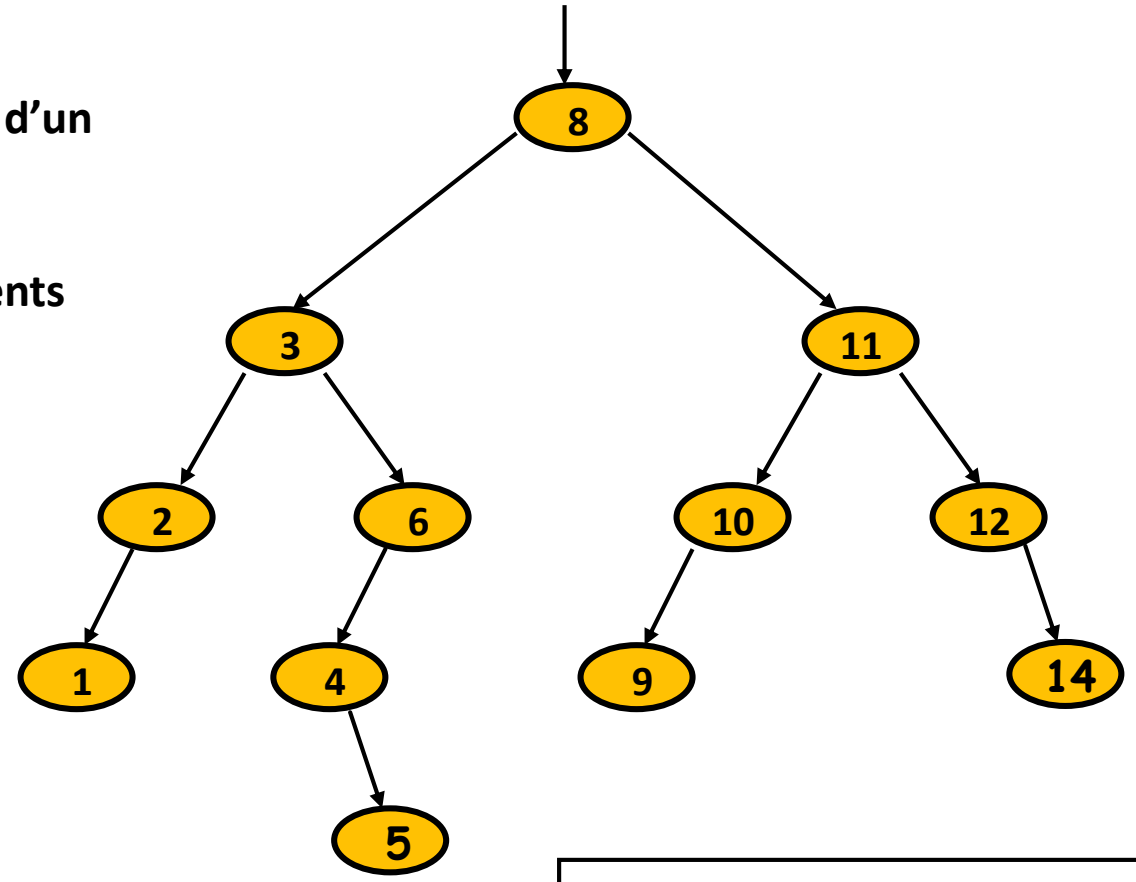
Ajout d'éléments dans un arbre de tri



8,3,11,2,1,6,10,9,12,4,5,14

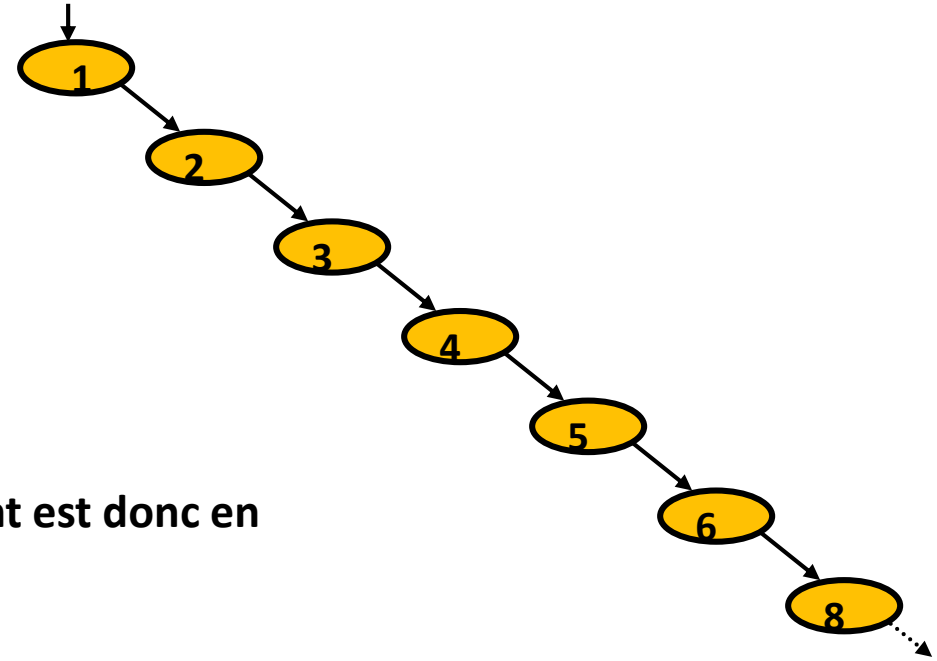
Première sequence d'insertion

La complexité d'insertion d'un élément est en $O(h(n))$, où $h(n) \equiv$ la hauteur de l'arbre contenant n éléments



8,3,11,2,1,6,10,9,12,4,5,14

Deuxième sequence d'insertion



L'insertion d'un élément est donc en $O(n)$ en pire cas

1,2,3,4,5,6,8,9,10,11,12,14

Complexité de l'insertion/ recherche/ suppression dans un arbre de tri non équilibré

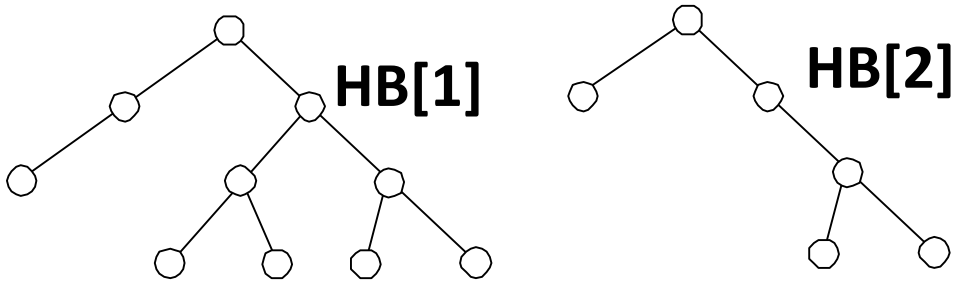
- Meilleur cas: $O(1)$
- Pire cas: $O(n)$

Synthèse

- Arbres binaires de recherche (de tri)
  - Dictionnaire : conteneur d'éléments devant supporter efficacement les opérations
    - Insertion, Suppression, Trouver un élément
  - Implantation par chaînage
    - classe Nœud interne : data (clé): type E
    - Si conteneur associatif : Clé + valeur
    - operator< du type E surchargé
    - Sous-arbres :gauche < droit >
    - Méthodes publiques qui font des appels récursifs de méthodes privées (_m) (ex : parcours en pré-ordre, insertion)
  - Insertion/recherche/suppression dans un arbre de tri non équilibré
    - Meilleur cas: $O(1)$
    - Pire cas: $O(n)$

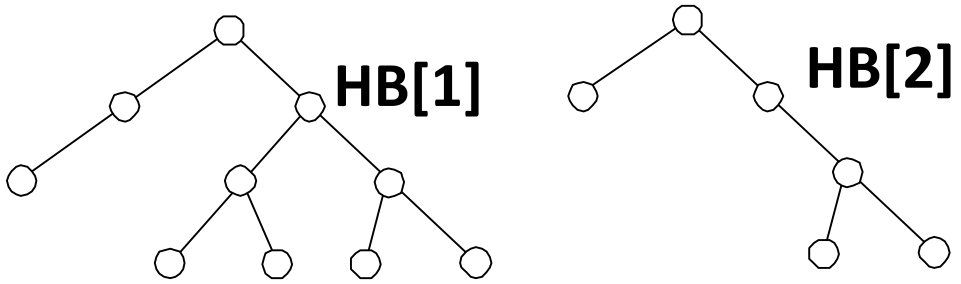
Les arbres AVL

- Un arbre binaire de recherche (de tri) doit être équilibré pour supporter efficacement les opérations de recherche, d'insertion et de suppression
- Le facteur d'équilibre d'un nœud est de k lorsque
  - |hauteur(sous-arbre droit) - hauteur(sous-arbre gauche)| = k.
  - Remarque: la hauteur d'un arbre de 0 nœuds = -1.
- Un arbre est $HB[k]$ lorsque le facteur d'équilibre de ses nœuds est $\leq k$



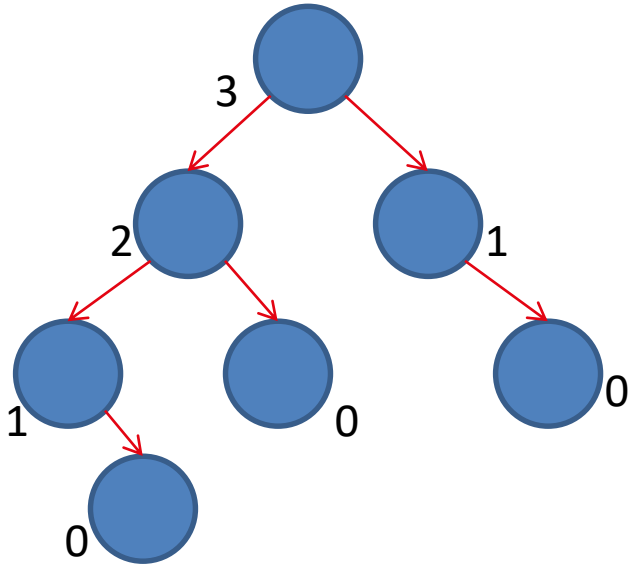
Les arbres AVL

- Les arbres AVL sont des arbres $HB[1]$
- Nous verrons que $h(n)$ est en $O(\log n)$ pour les arbres AVL.
- Le temps d'exécution des insertions/suppressions/recherches sera donc toujours en $O(\log n)$ pour un arbre AVL de n nœuds.

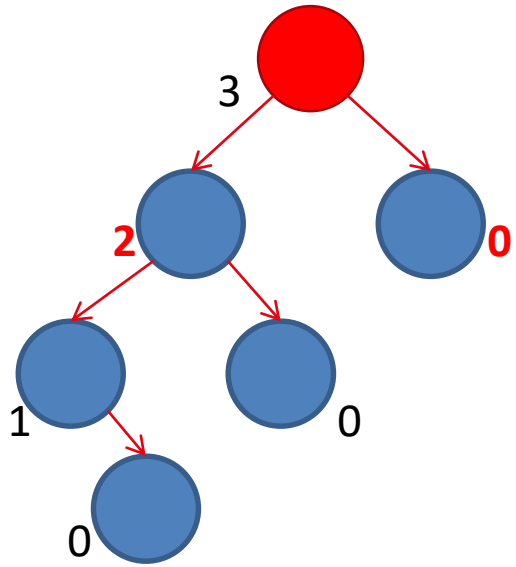


Arbres équilibrés (AVL)

Bien équilibré selon la règle AVL



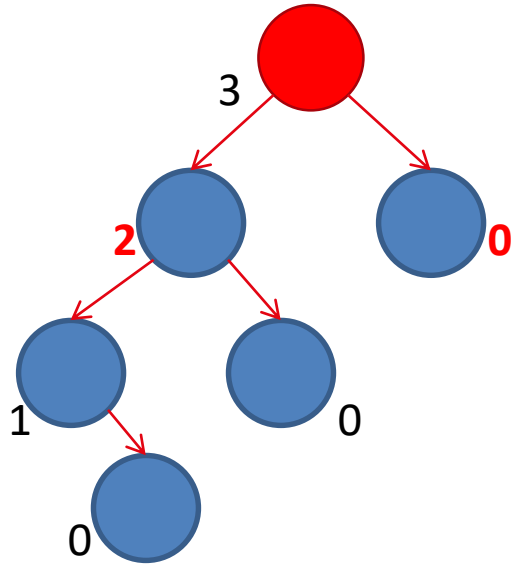
Mal équilibré selon la règle AVL



Nœud critique

- Quand il y a un déséquilibre, le nœud le plus profond à partir duquel il y a une différence de 2 ou plus est appelé le **nœud critique**.

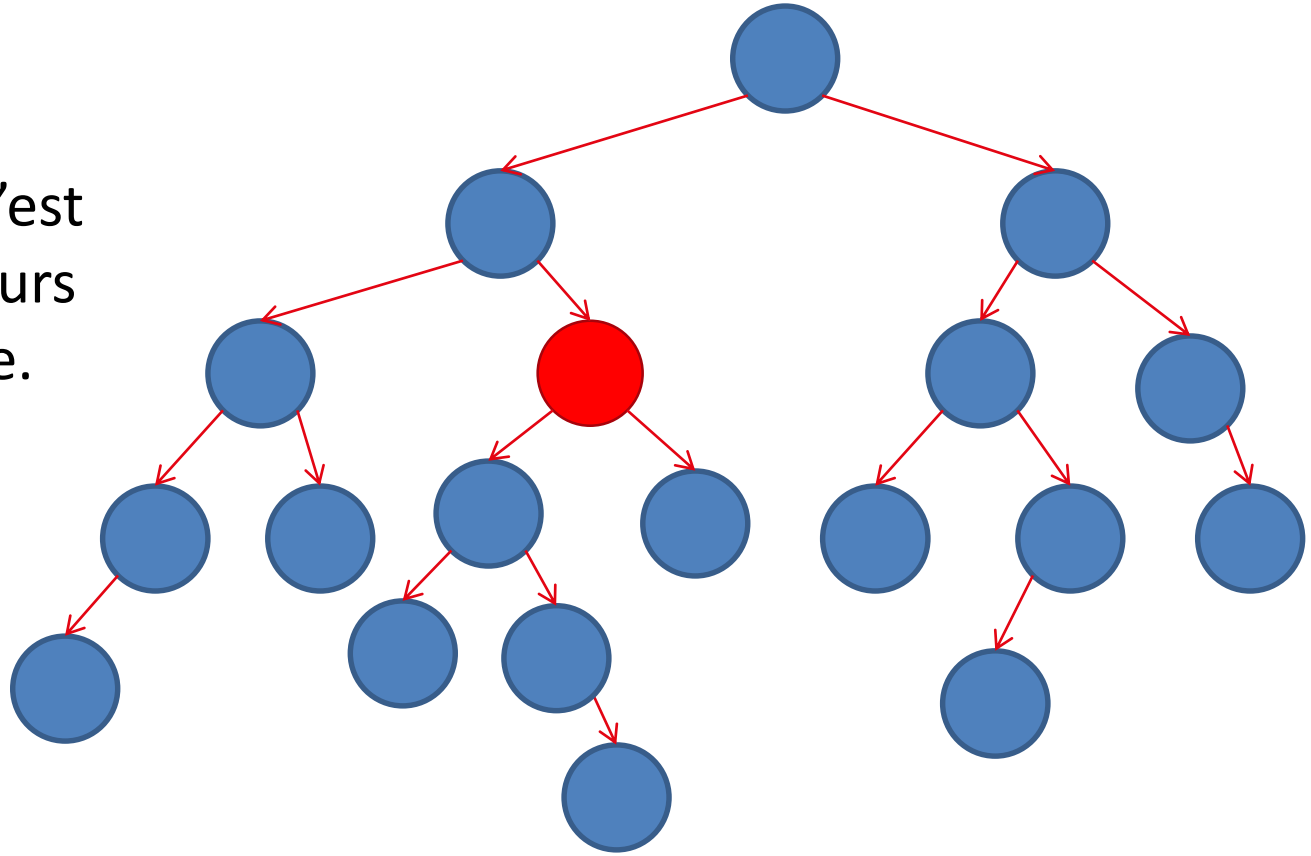
Ici, le nœud critique du déséquilibre est la racine



Nœud critique

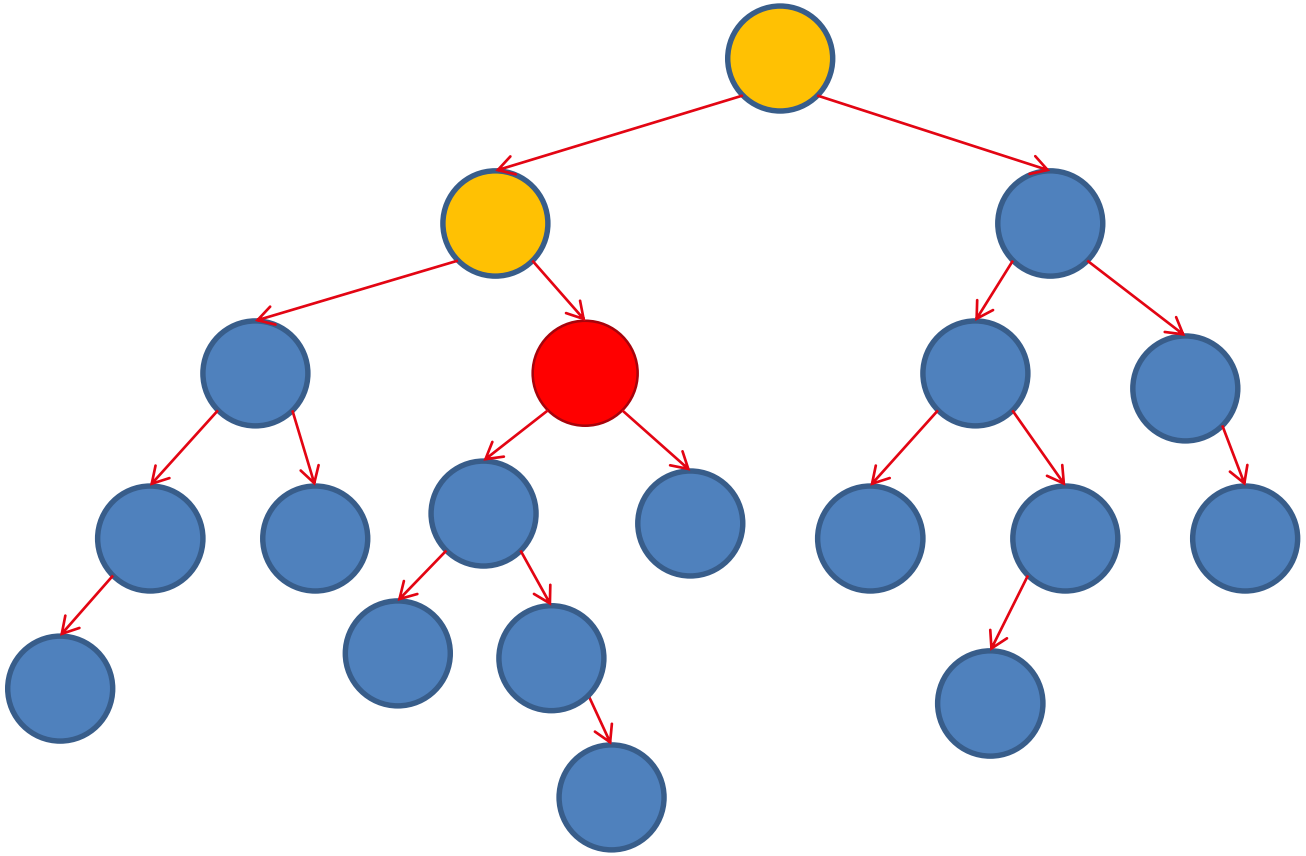
- Quand il y a un déséquilibre, le nœud le plus profond à partir duquel il y a une différence de 2 ou plus est appelé le **nœud critique**.

Mais ce n'est pas toujours la racine.



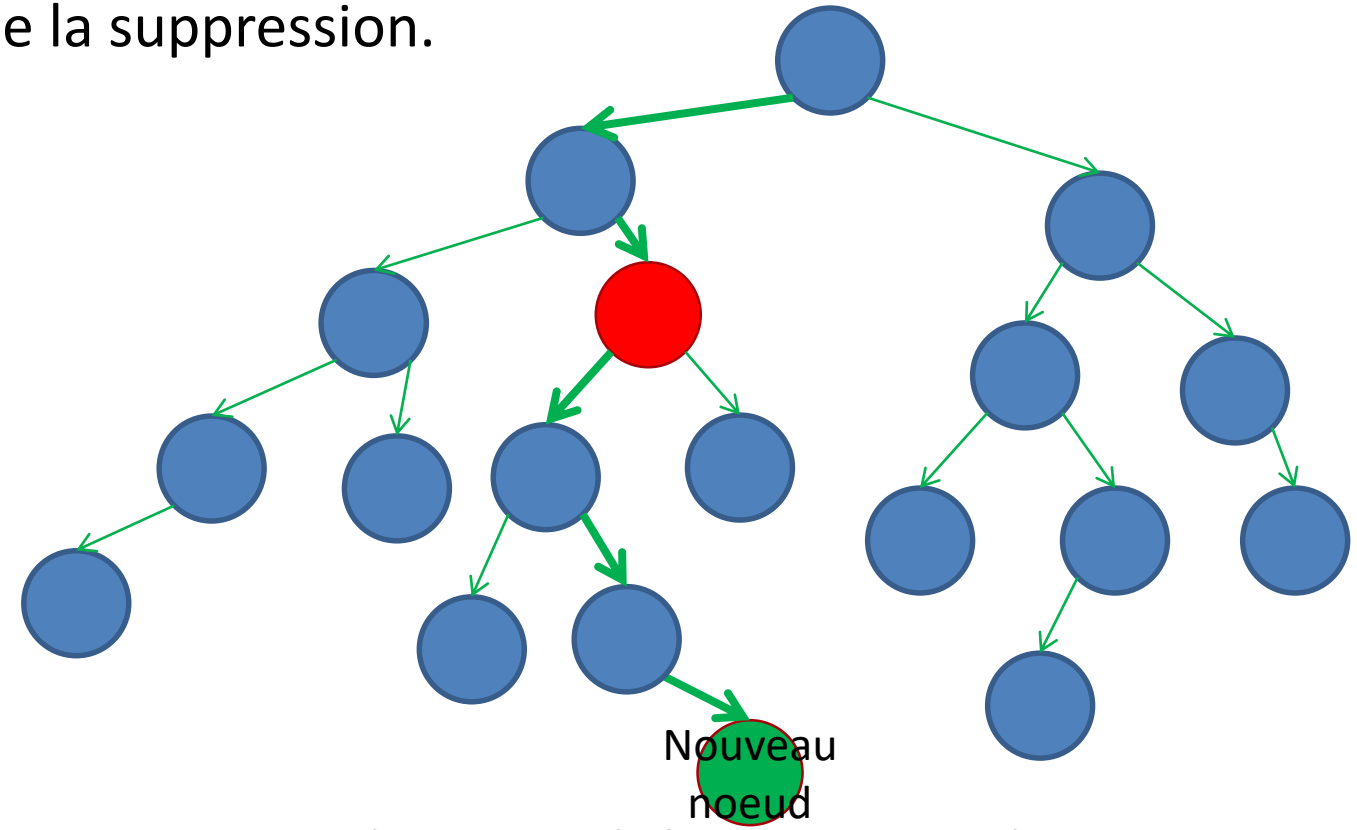
Nœud critique

- Il peut parfois y avoir plusieurs nœuds débalancés. Dans ce cas, on s'occupe d'abord du plus profond de tous, c'est le nœud critique.



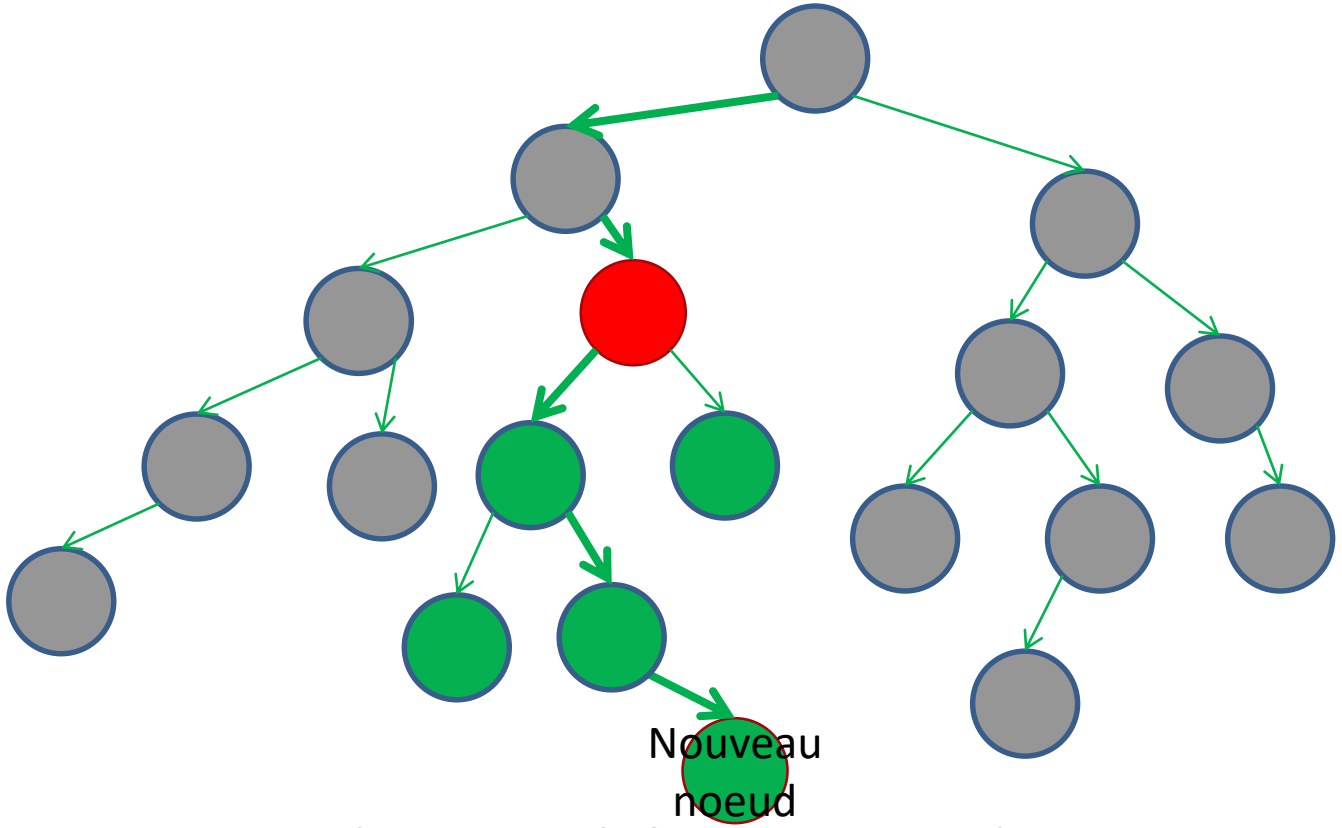
Déséquilibre

- Surviennent lors d'un ajout ou lors d'une suppression.
- S'il y a lieu, le ou les nœuds débalancés engendrés sont toujours sur le chemin de l'ajout ou de la suppression.



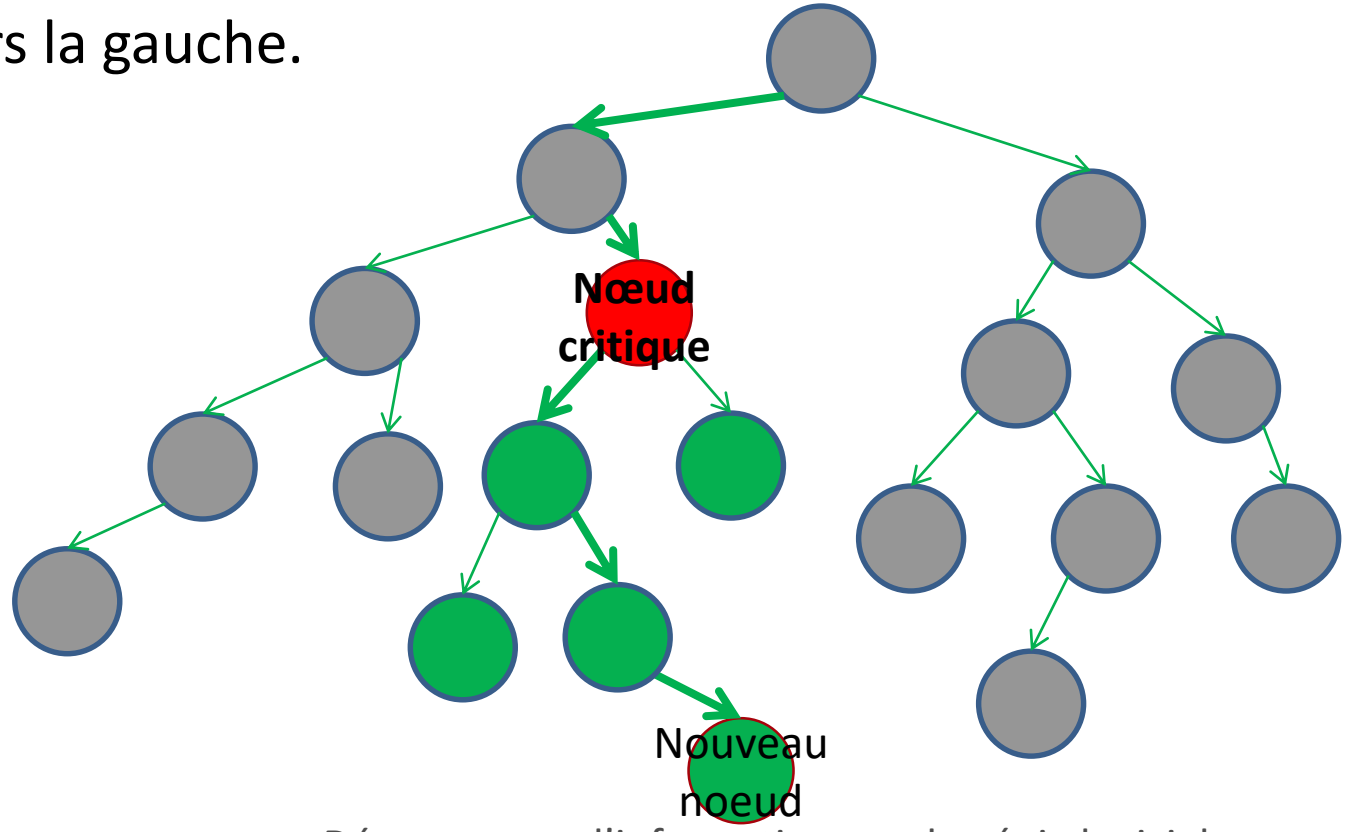
Rééquilibrer un arbre déséquilibré

- Quand un déséquilibre apparaît, il faut remodeler la partie de l'arbre dont la racine est le nœud critique.



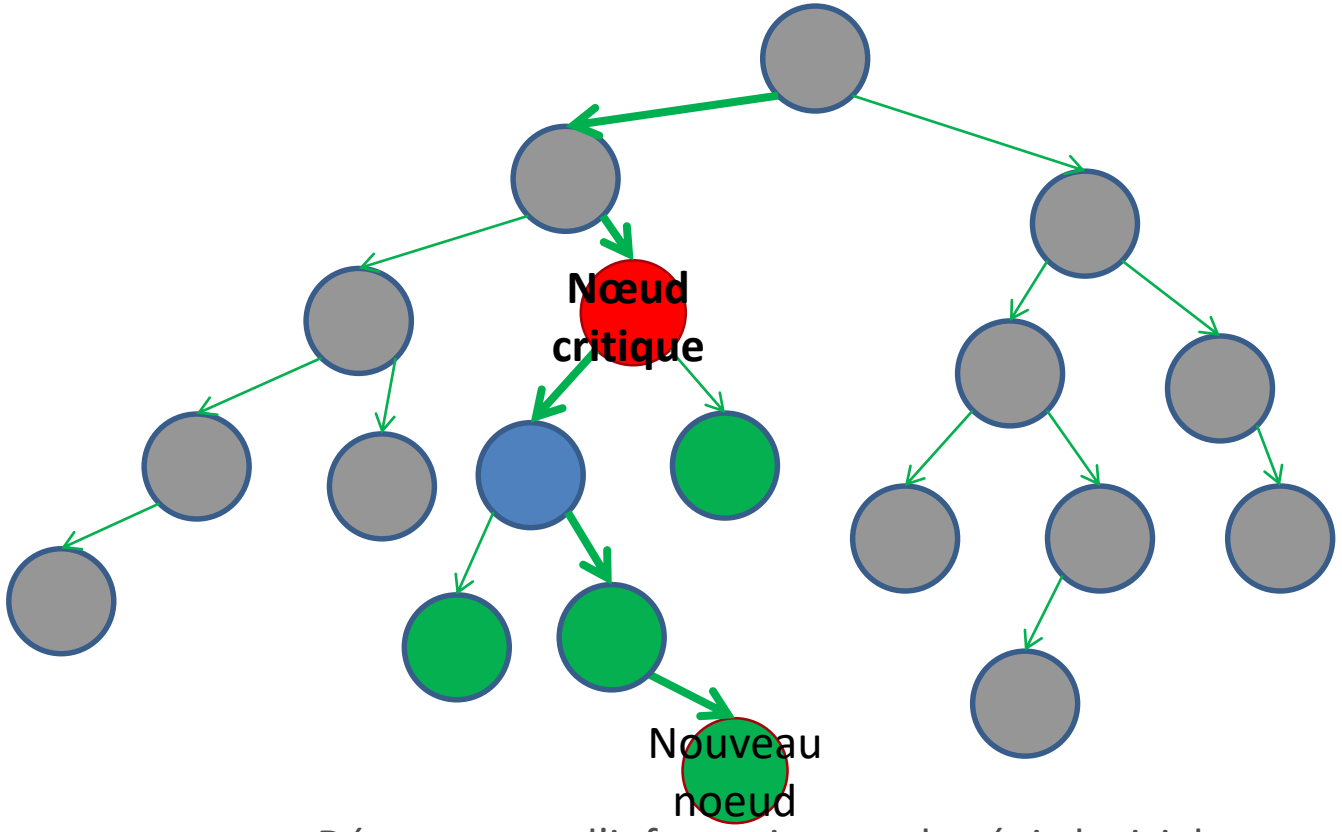
Trouver le cas de déséquilibre

- Identifier le genre de déséquilibre
  - Voir de quel côté l'arbre penche à partir du nœud critique.
- Ici, c'est vers la gauche.



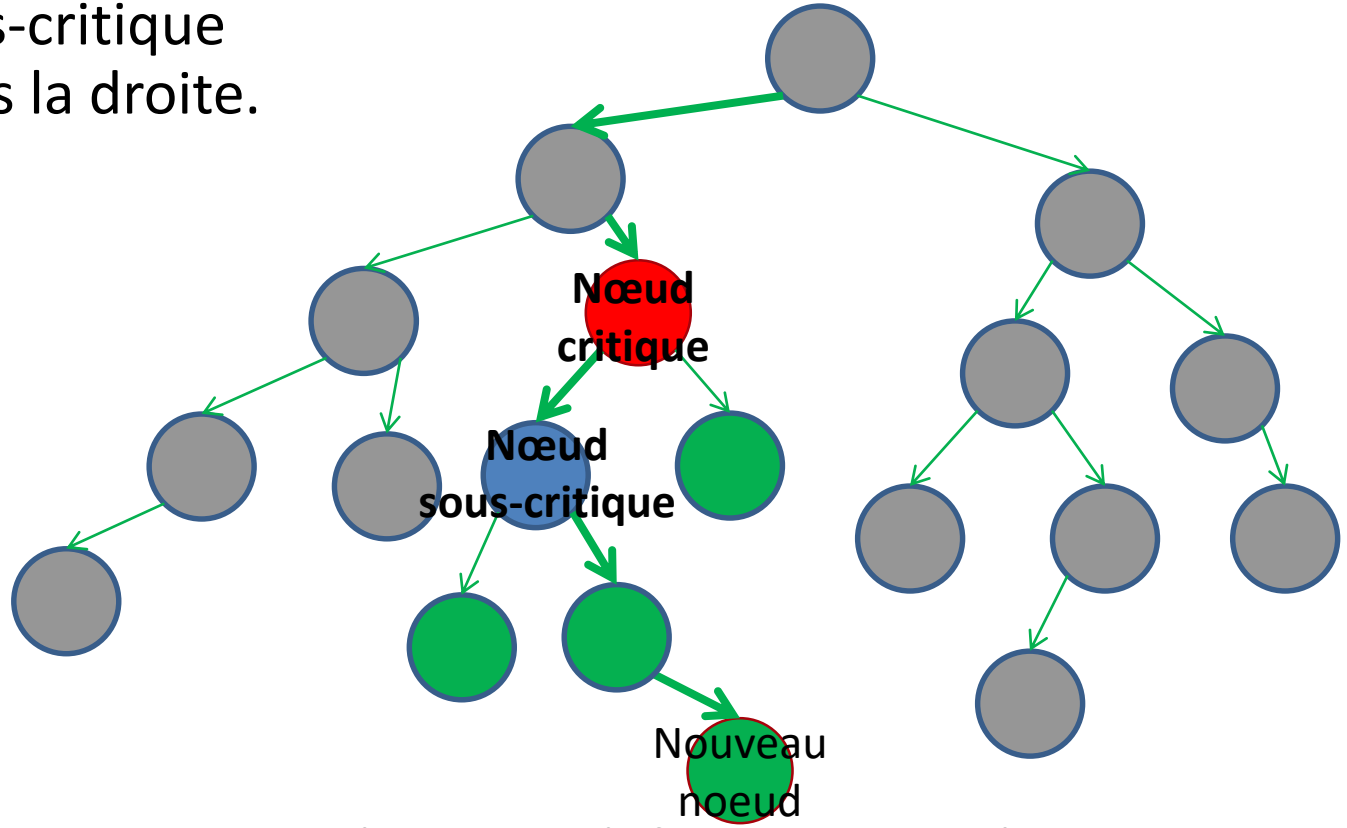
Trouver le cas de déséquilibre

- Puis, regarder de quel côté penche l'arbre à partir du nœud sous-critique (l'enfant du nœud critique sur le côté le plus profond)



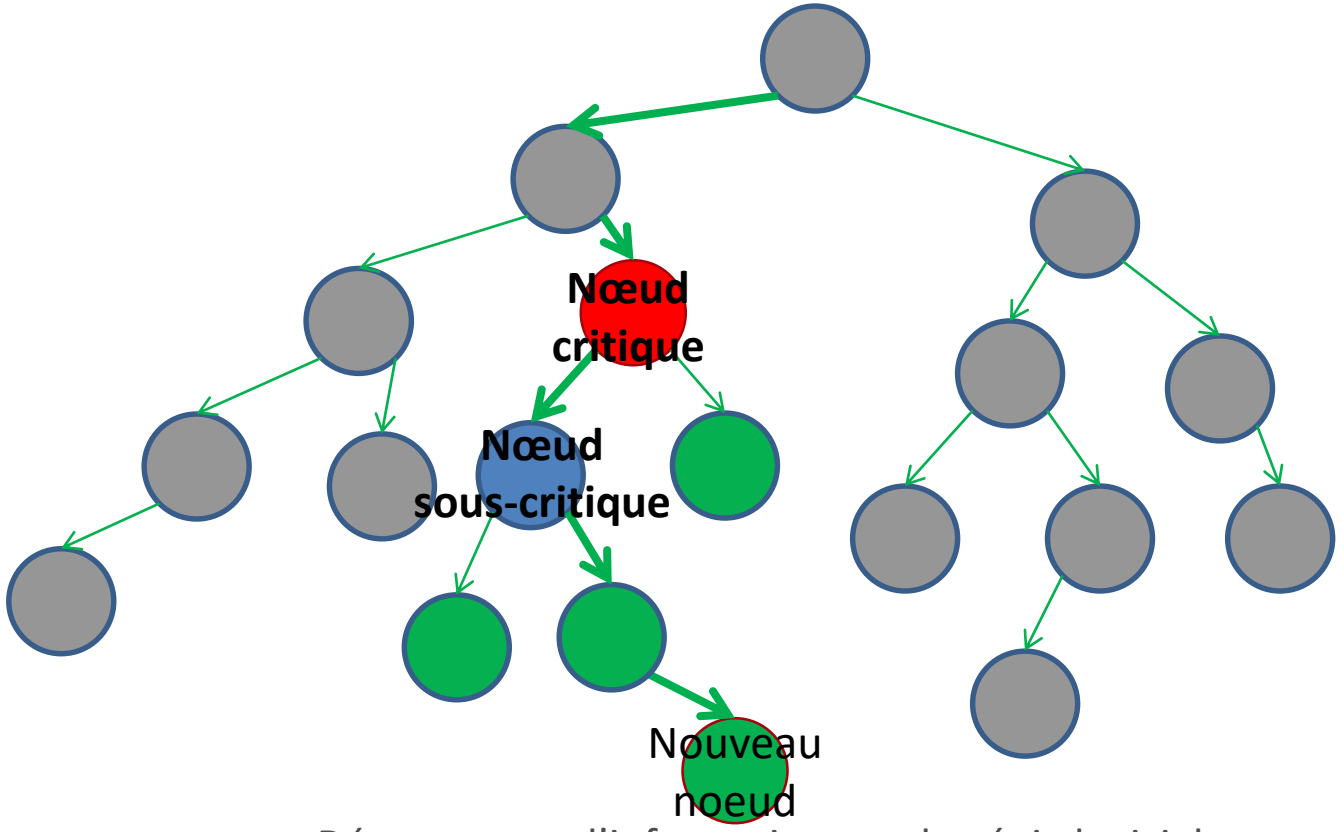
Trouver le cas de déséquilibre

- L'arbre sous-critique n'a pas besoin d'être déséquilibré pour qu'on considère qu'il penche. Une différence de 1 suffit pour identifier le cas.
- L'arbre sous-critique penche vers la droite.



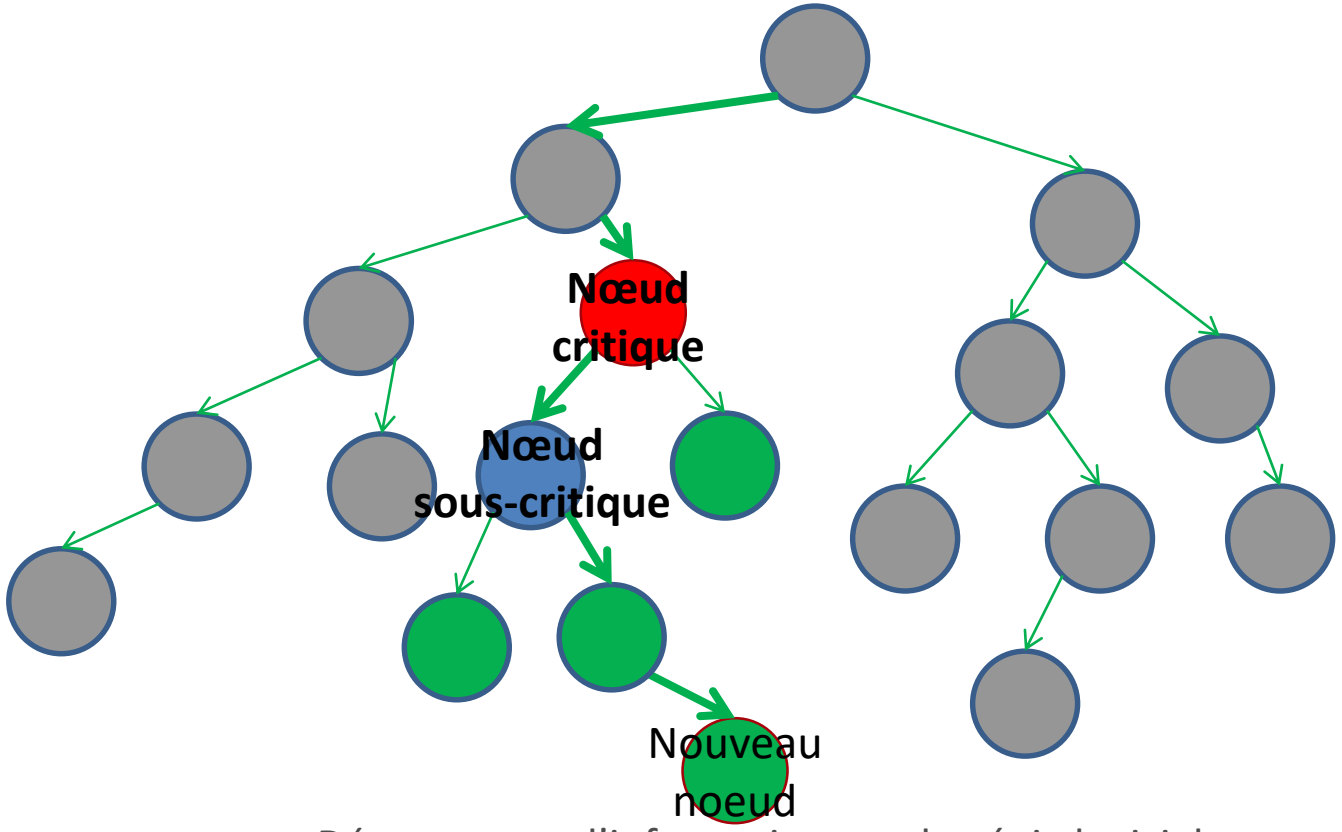
Trouver le cas de déséquilibre

- Dans cet exemple-ci, un déséquilibre vers la gauche, avec un arbre sous-critique qui penche vers la droite.



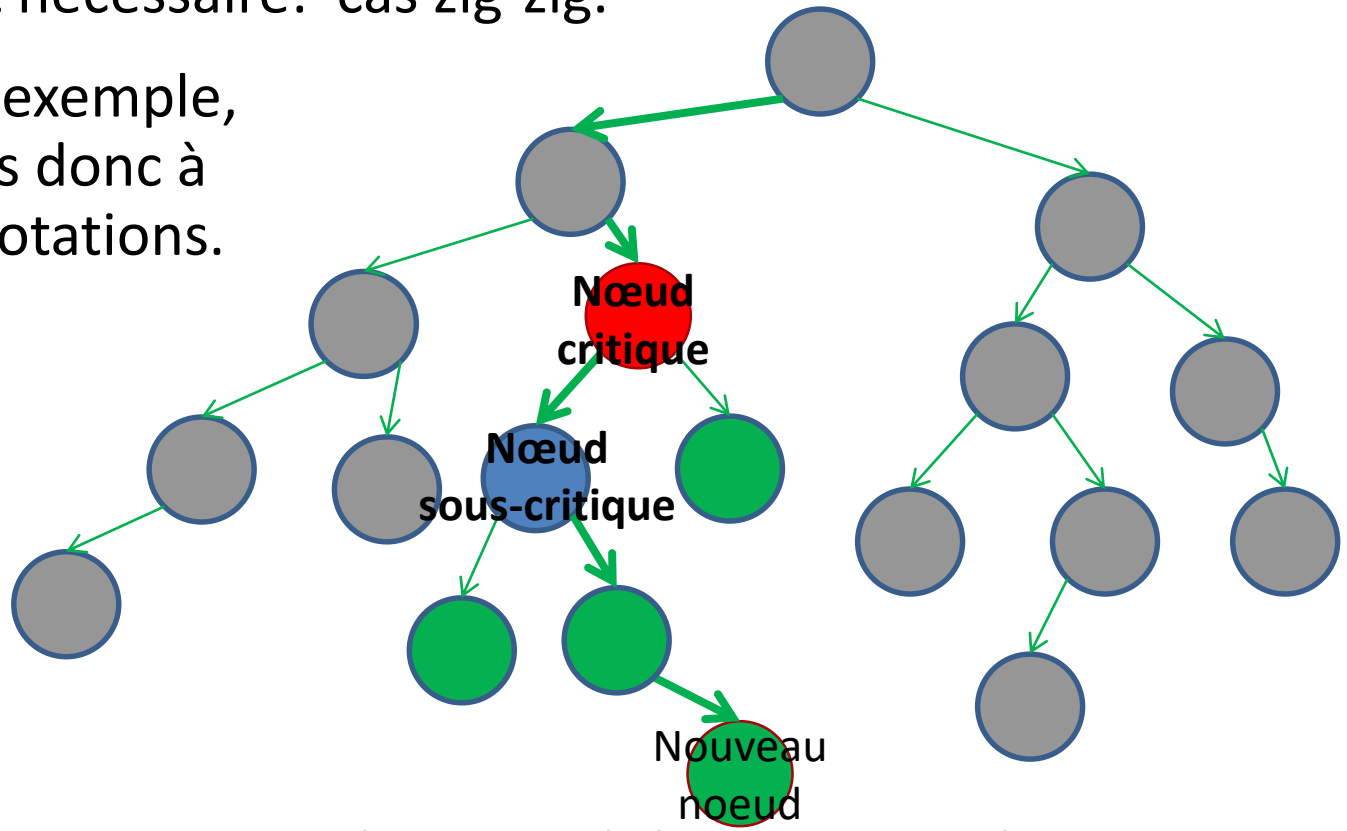
Trouver le cas de déséquilibre

- Quand le sens du déséquilibre principal est différent du sens dans lequel l'arbre sous-critique penche (déséquilibre secondaire), alors deux rotations sont nécessaires: cas zig-zag.



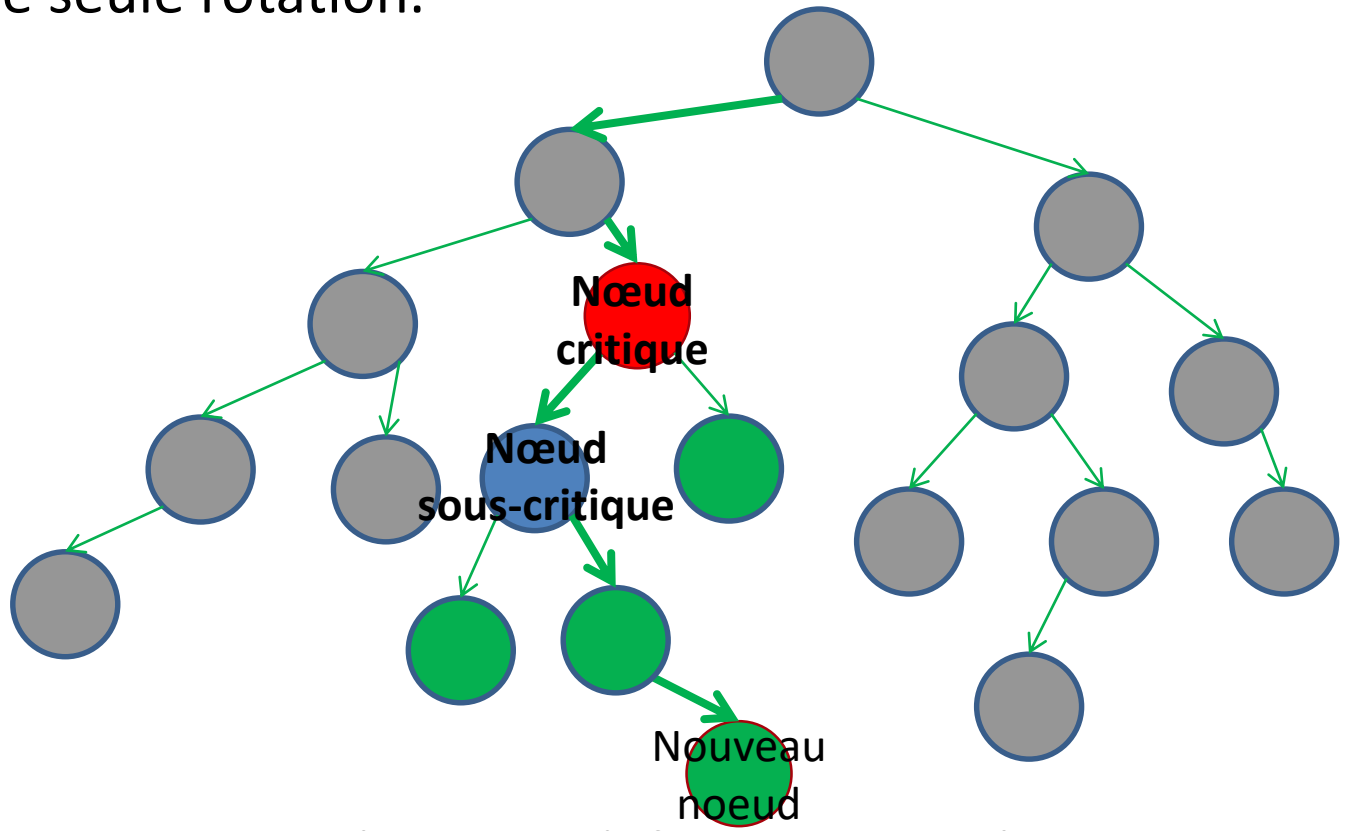
Trouver le cas de déséquilibre

- Quand le sens du déséquilibre principal est le même que le sens du déséquilibre secondaire, ou bien que l'arbre sous-critique ne penche pas du tout, alors une seule rotation est nécessaire: cas zig-zig.
- Dans notre exemple, nous aurons donc à faire deux rotations.



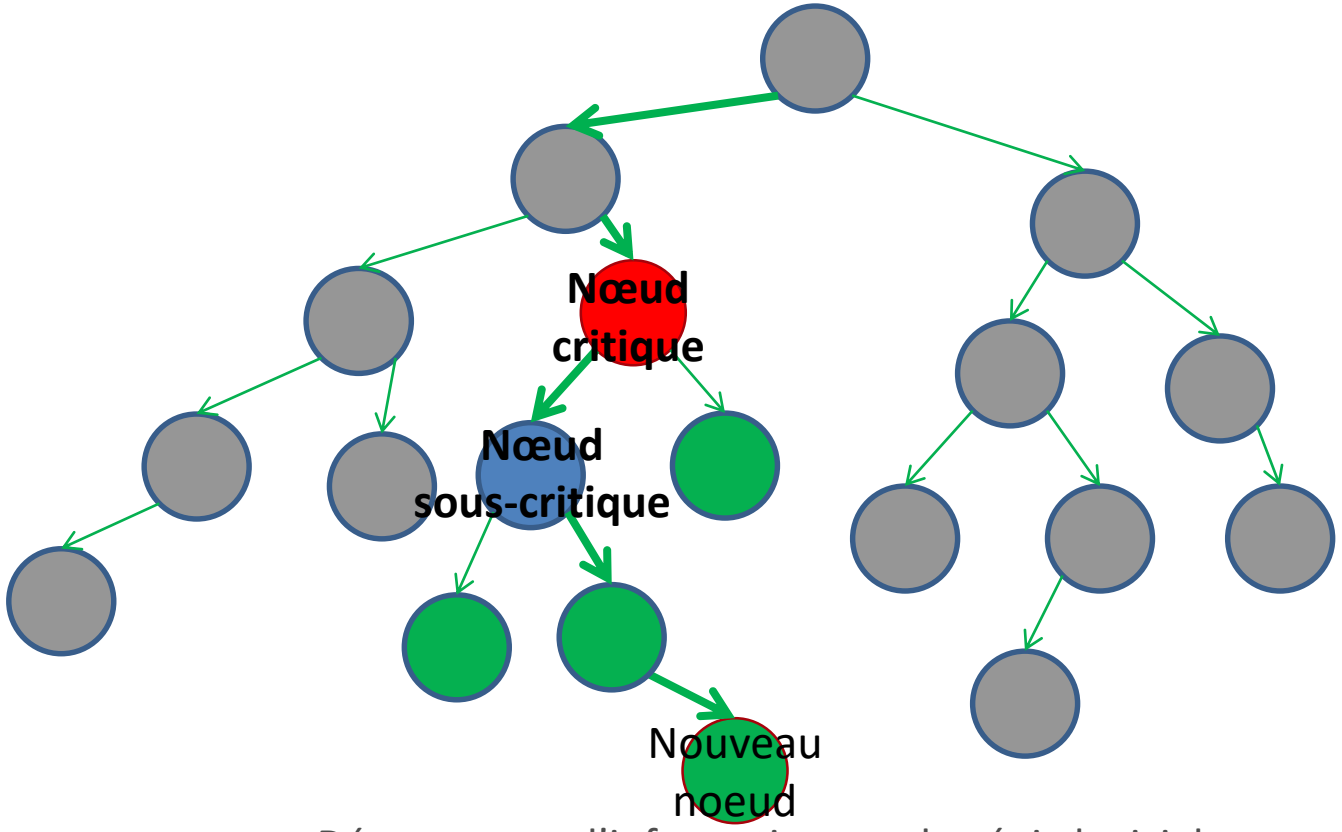
Les rotations

- Cas **zig-zag** : il faut faire deux rotations, la première sert à faire pencher l'arbre sous-critique dans le même sens que l'arbre critique, de façon à nous retrouver dans le cas simple d'une seule rotation.



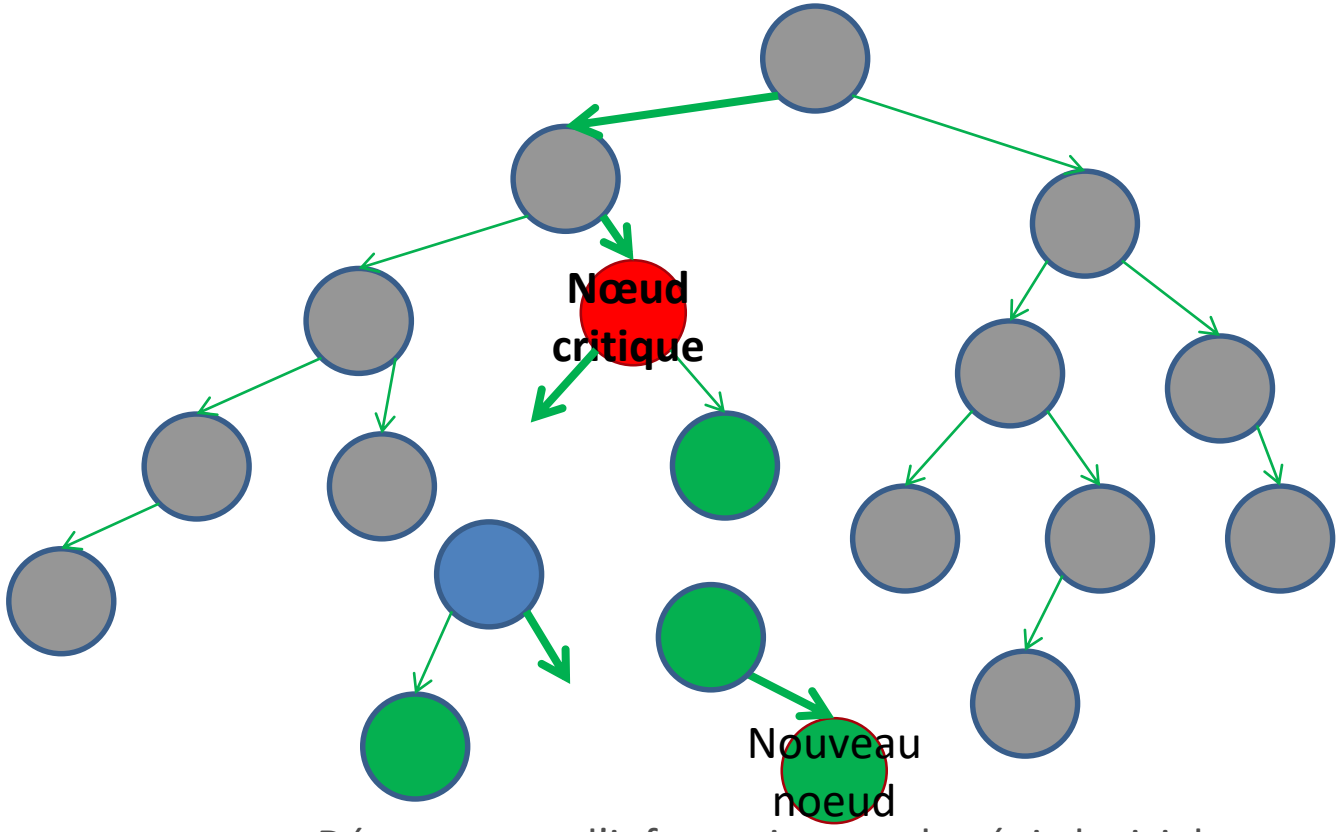
Les rotations

- Première rotation (préparatoire à la deuxième).



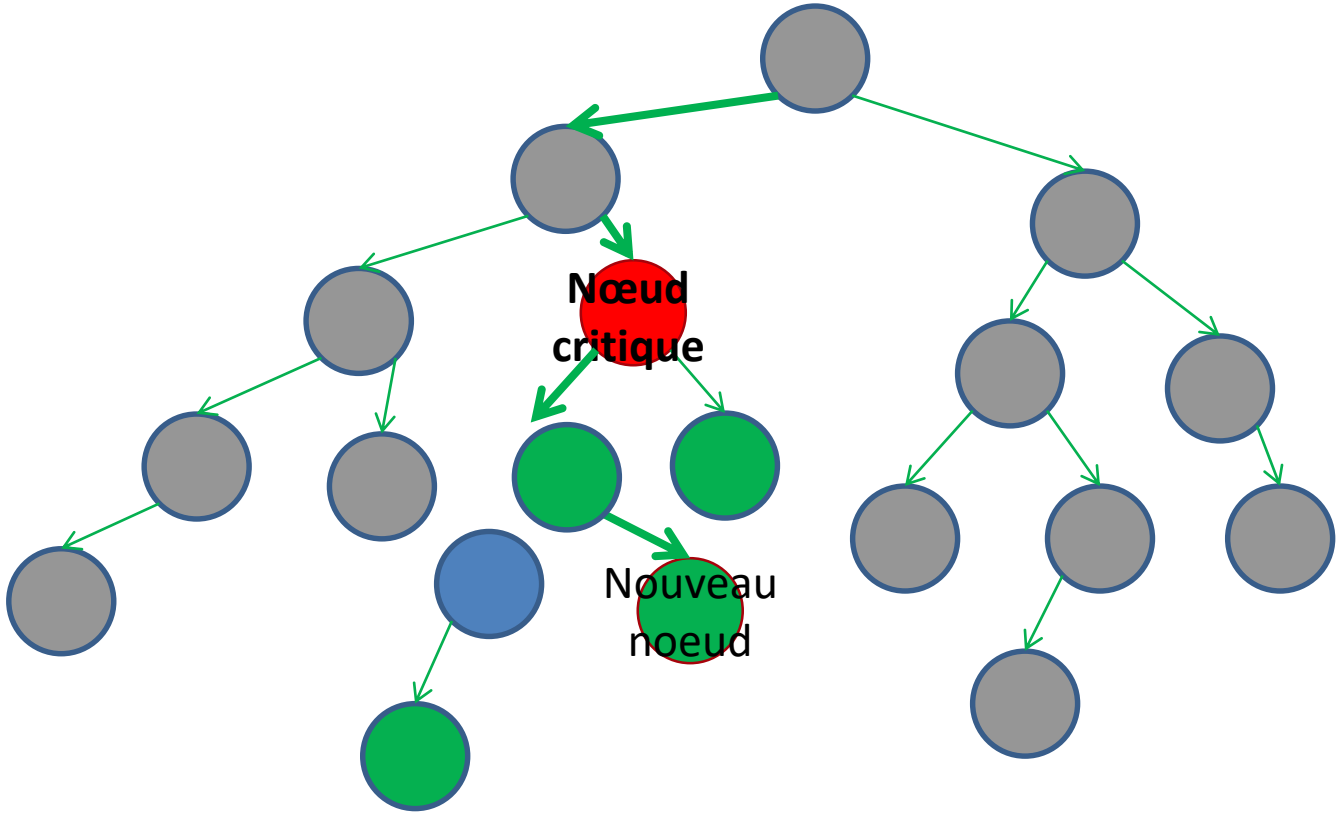
Les rotations

- Première rotation (préparatoire à la deuxième).



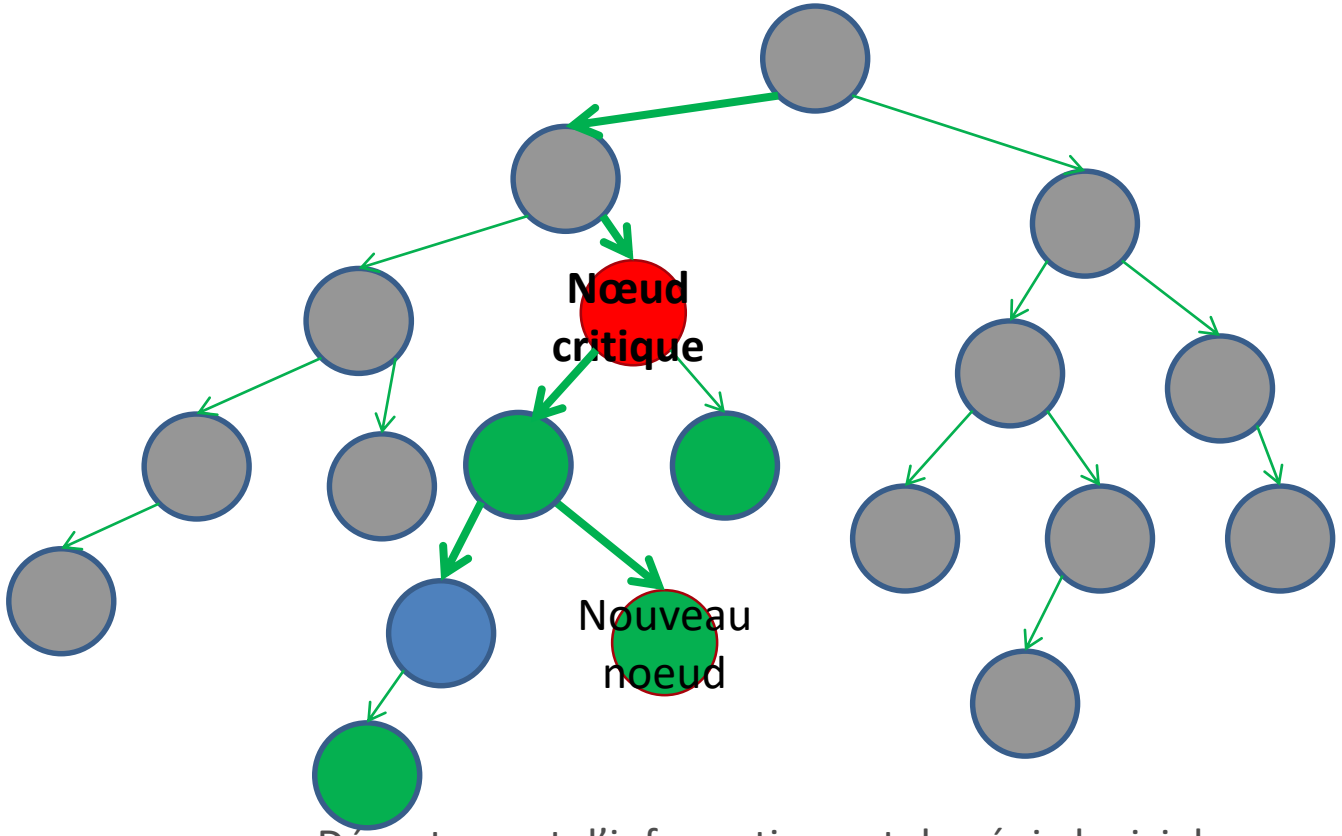
Les rotations

- Première rotation (préparatoire à la deuxième).



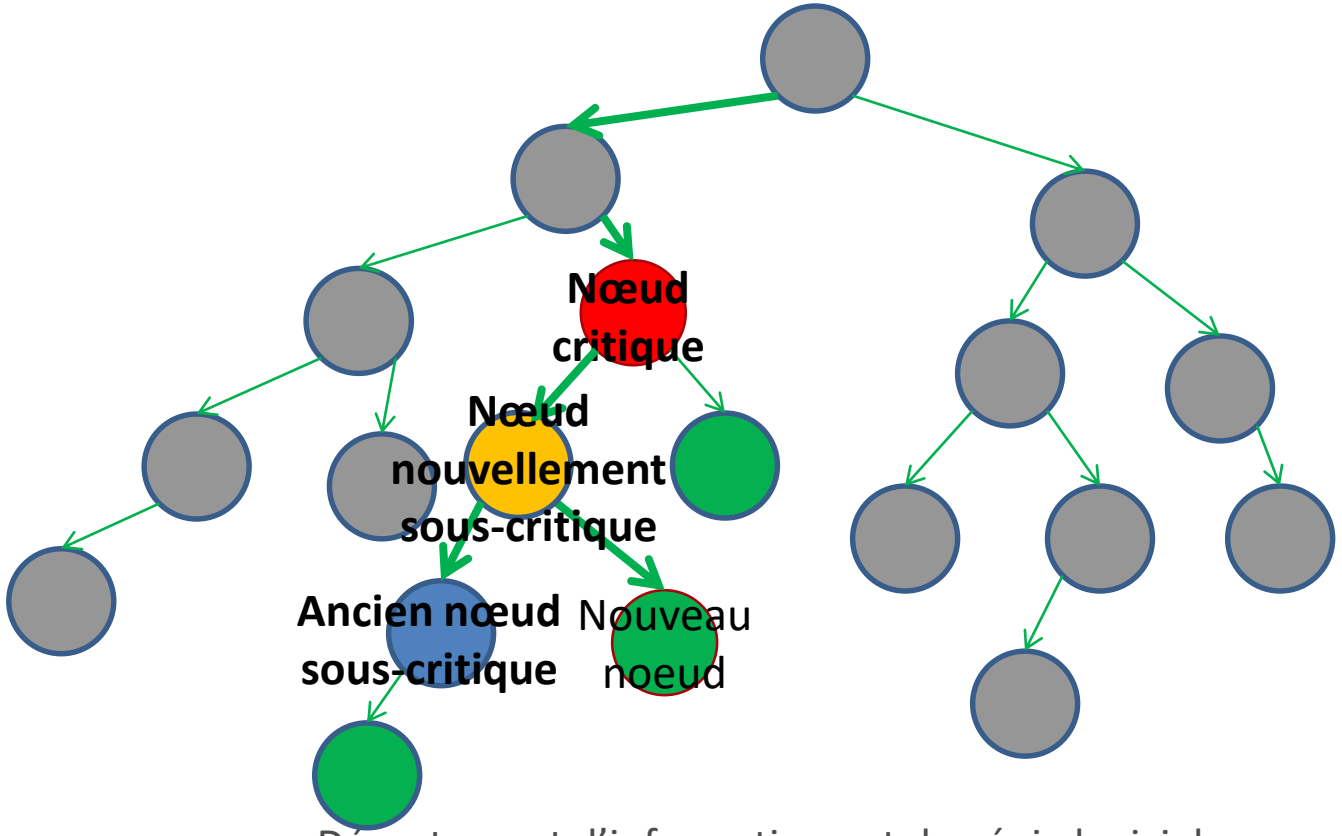
Les rotations

- Première rotation (préparatoire à la deuxième).



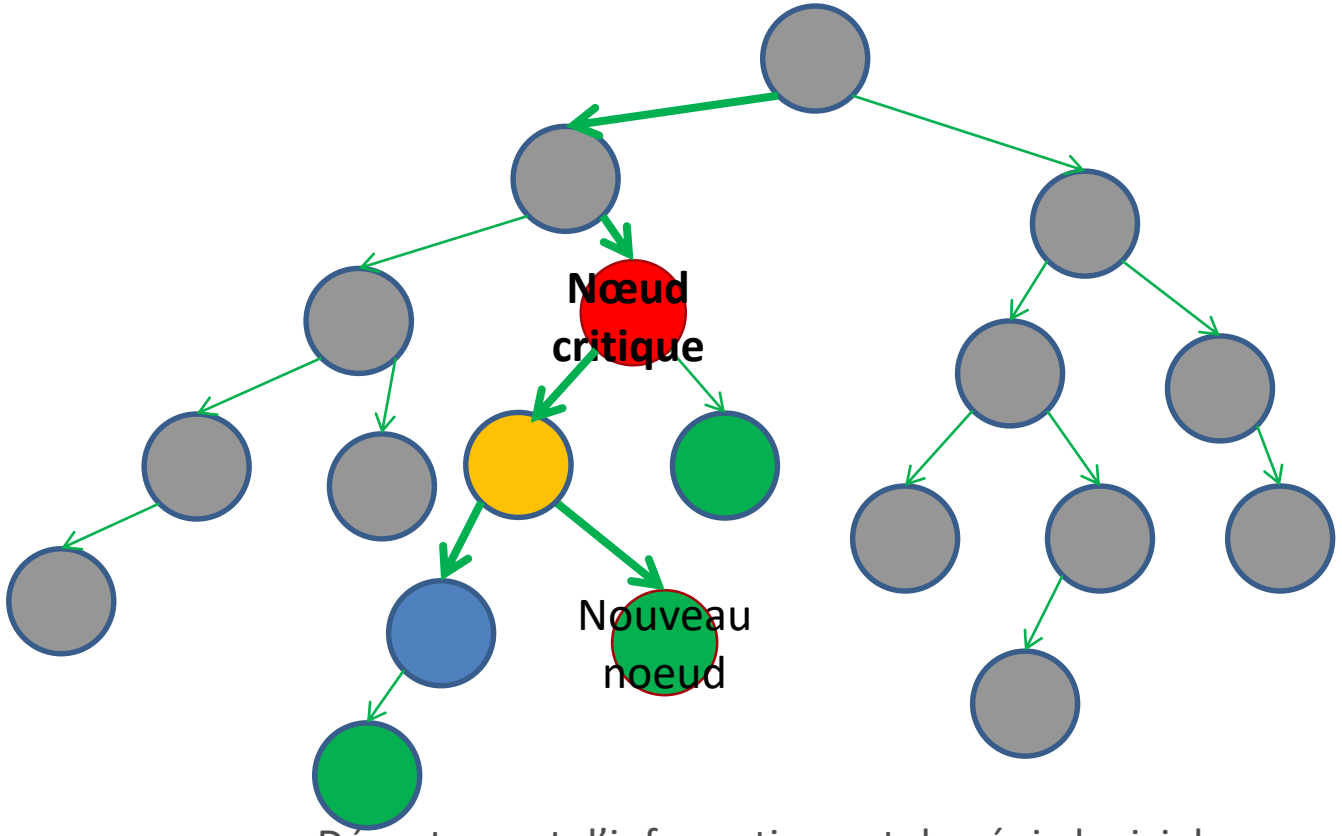
Les rotations

- Première rotation (préparatoire à la deuxième).



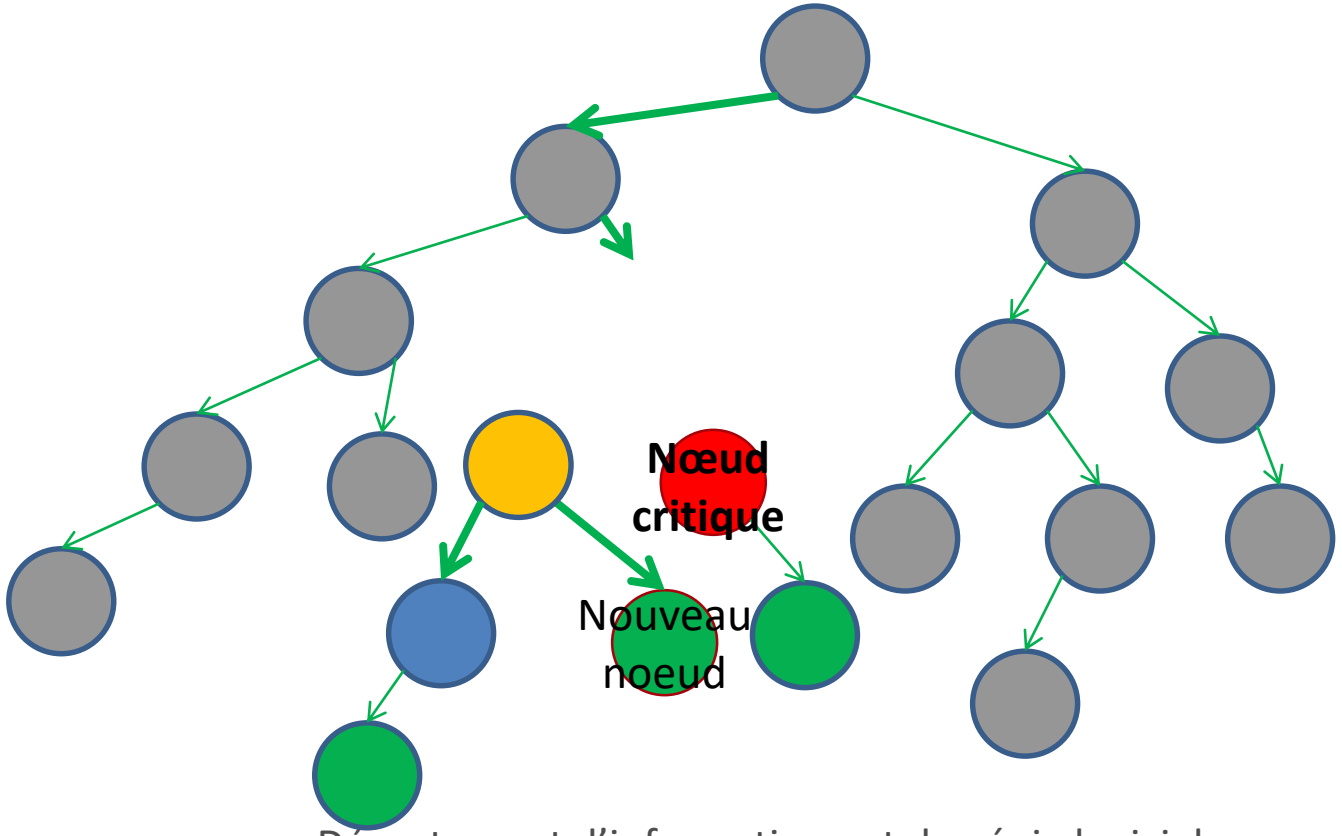
Les rotations

- Deuxième rotation



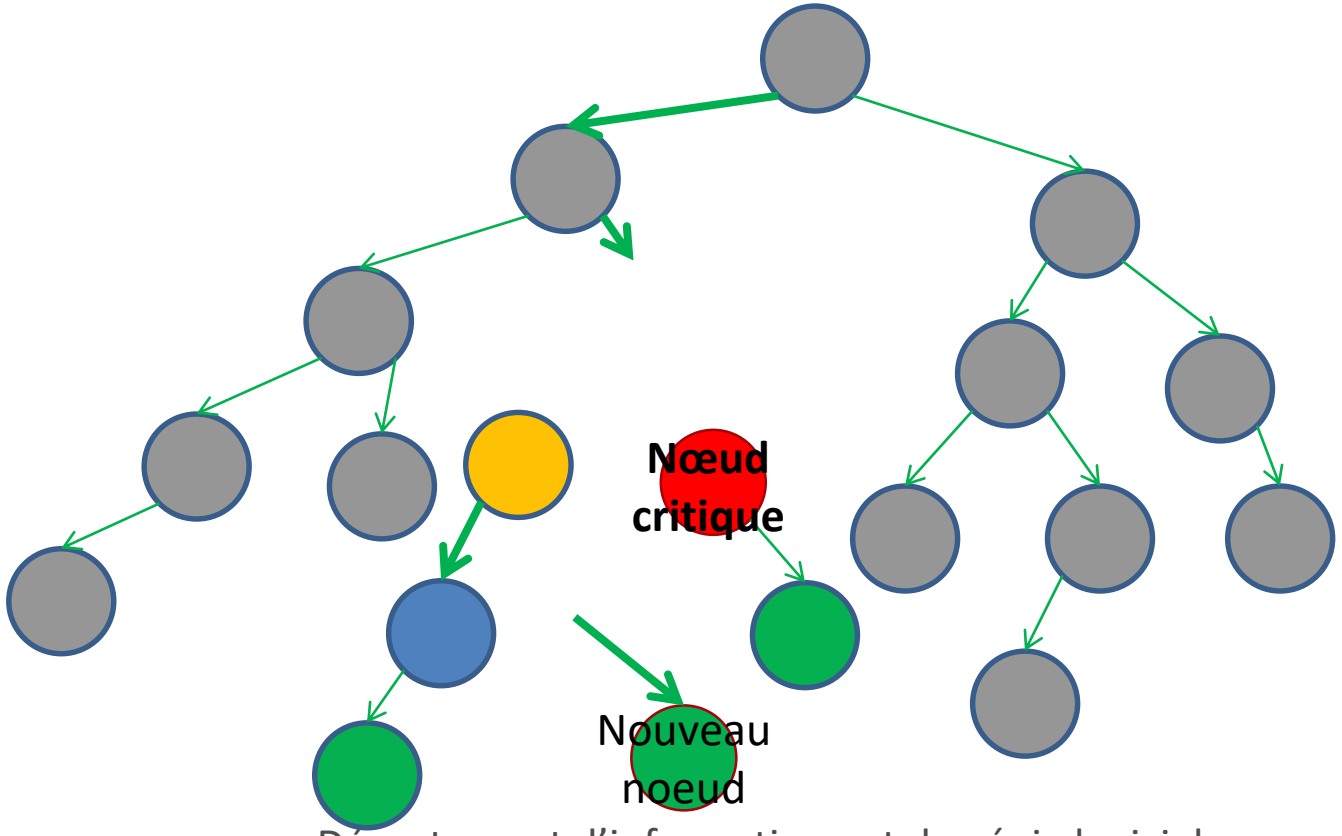
Les rotations

- Deuxième rotation



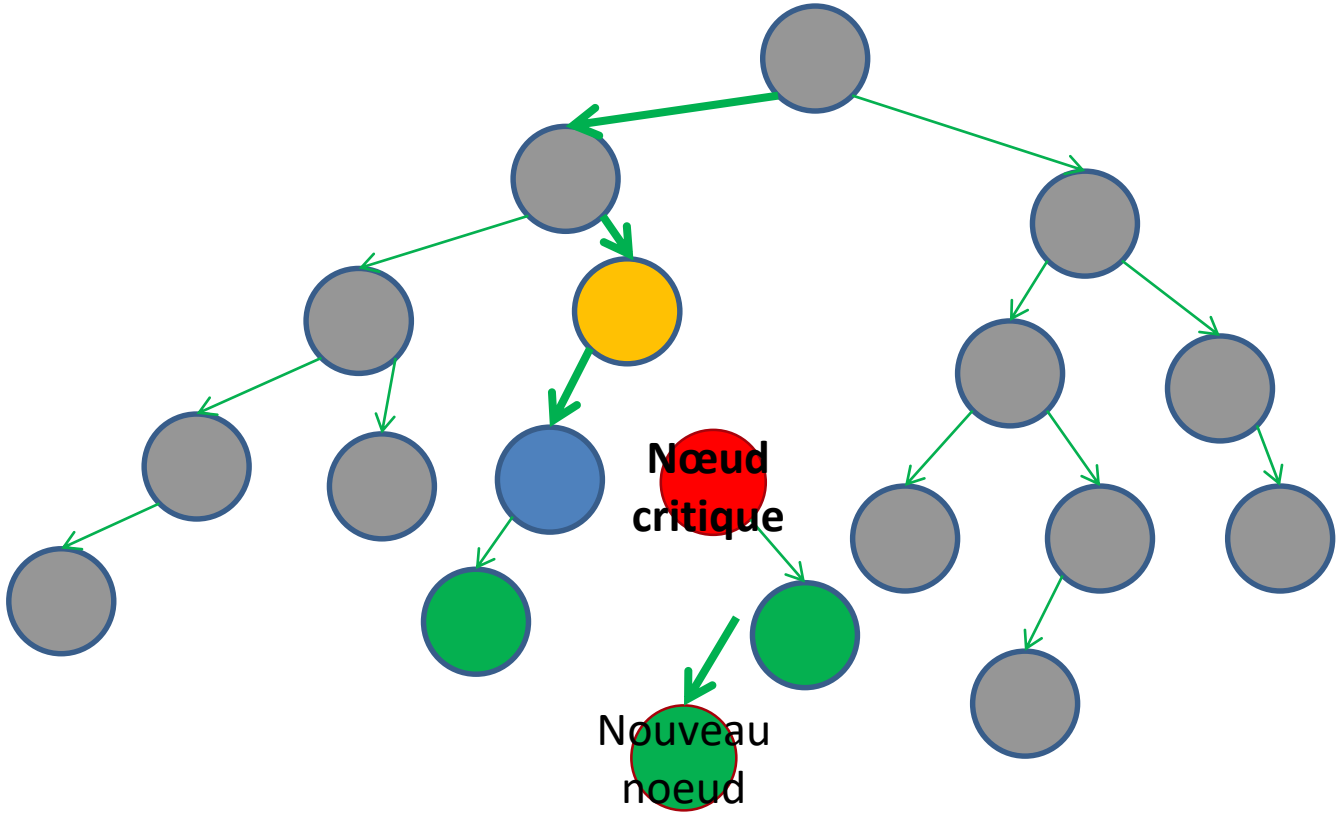
Les rotations

- Deuxième rotation



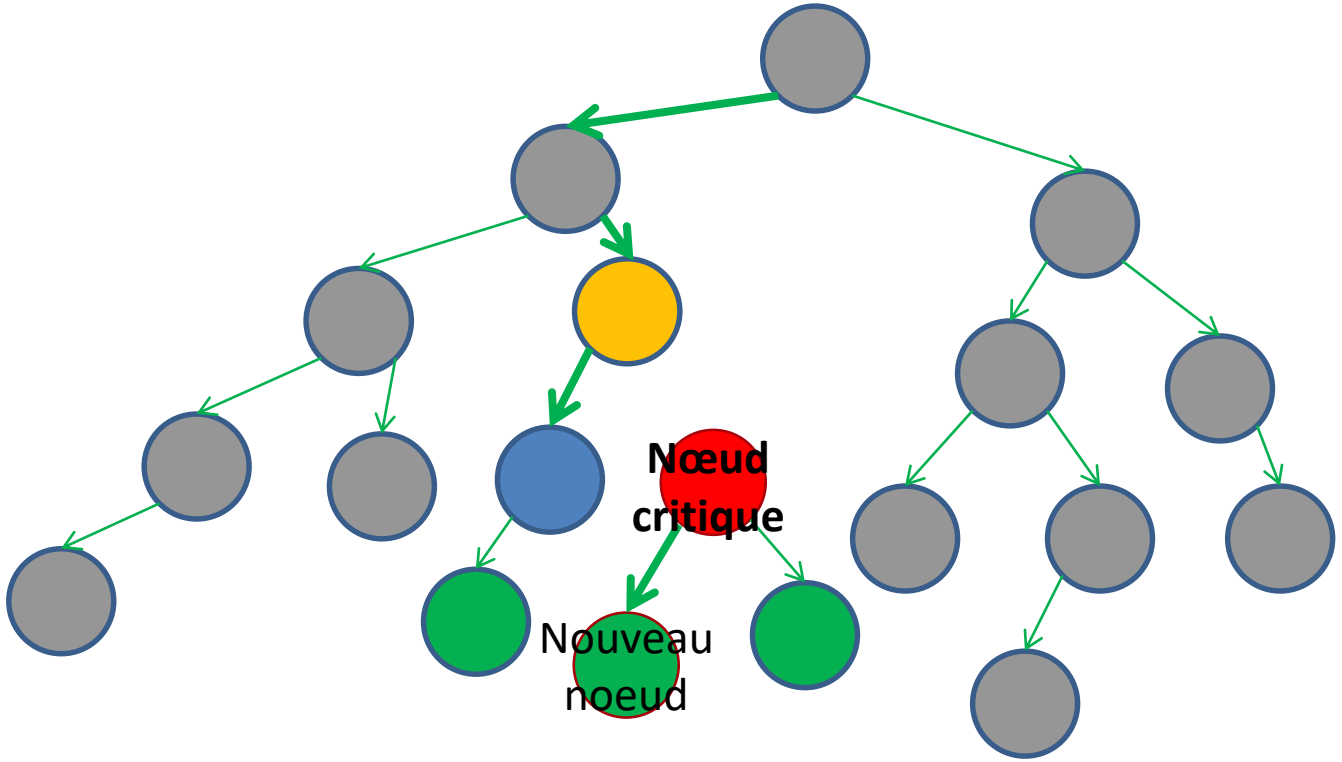
Les rotations

- Deuxième rotation



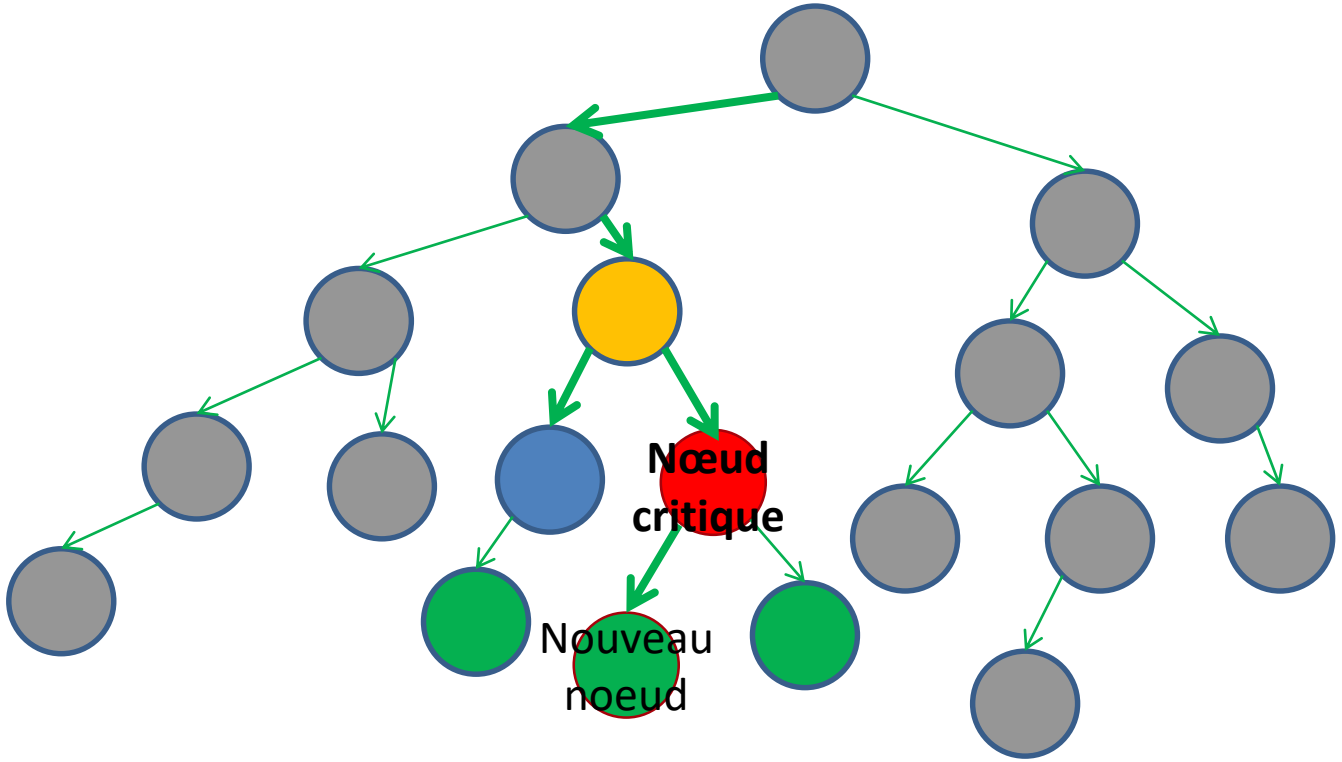
Les rotations

- Deuxième rotation



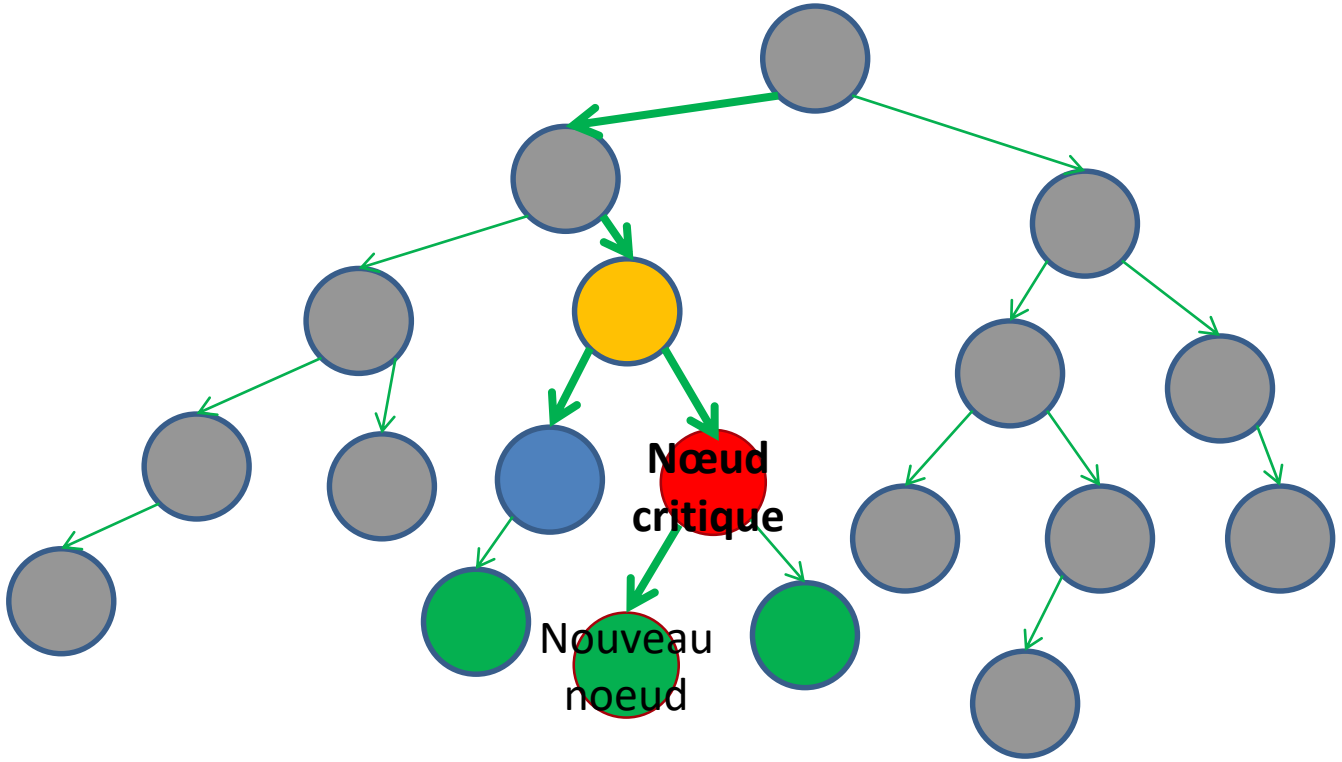
Les rotations

- Deuxième rotation



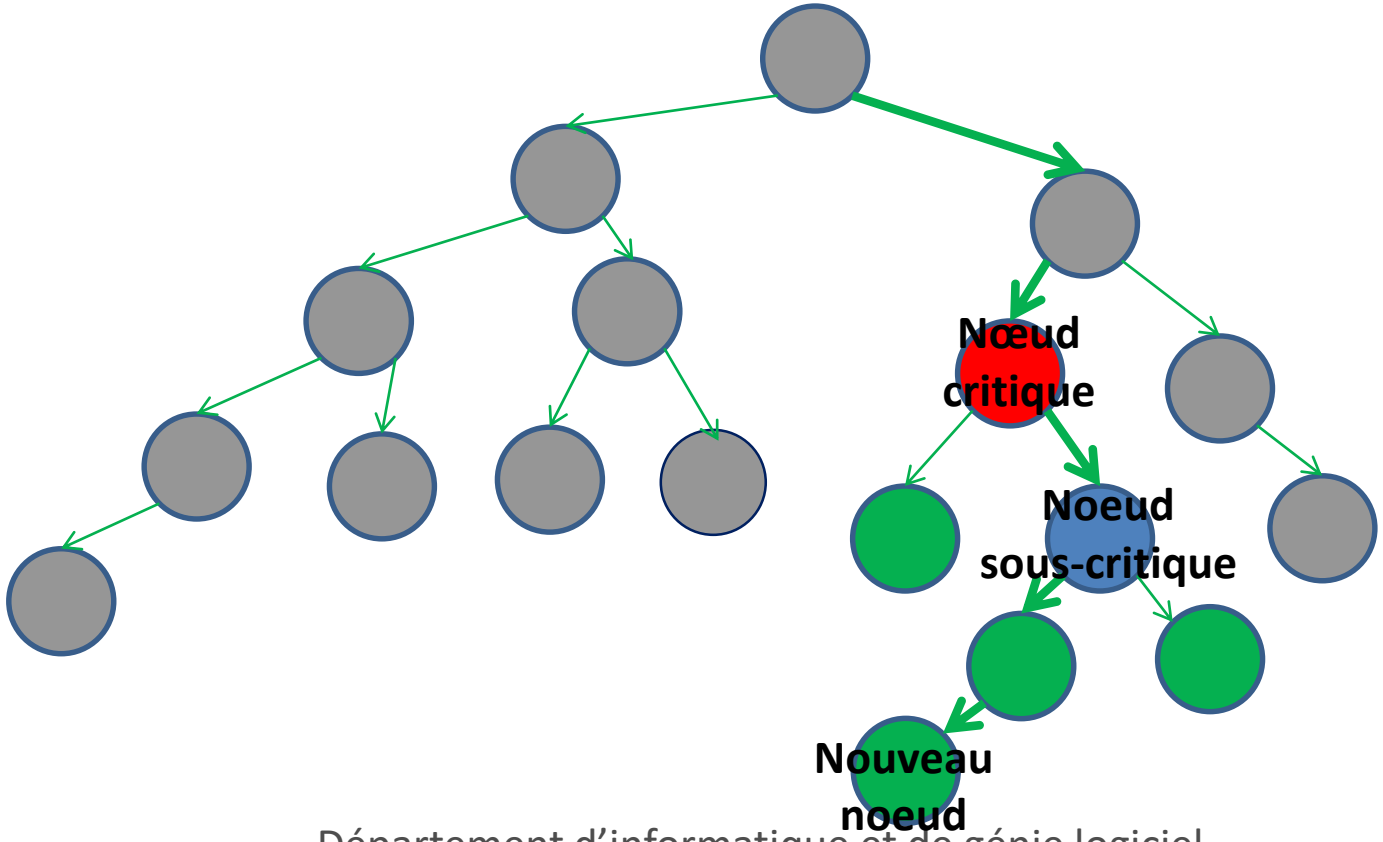
Les rotations

- Les deux rotations sont terminées, nous avons maintenant un arbre AVL équilibré.



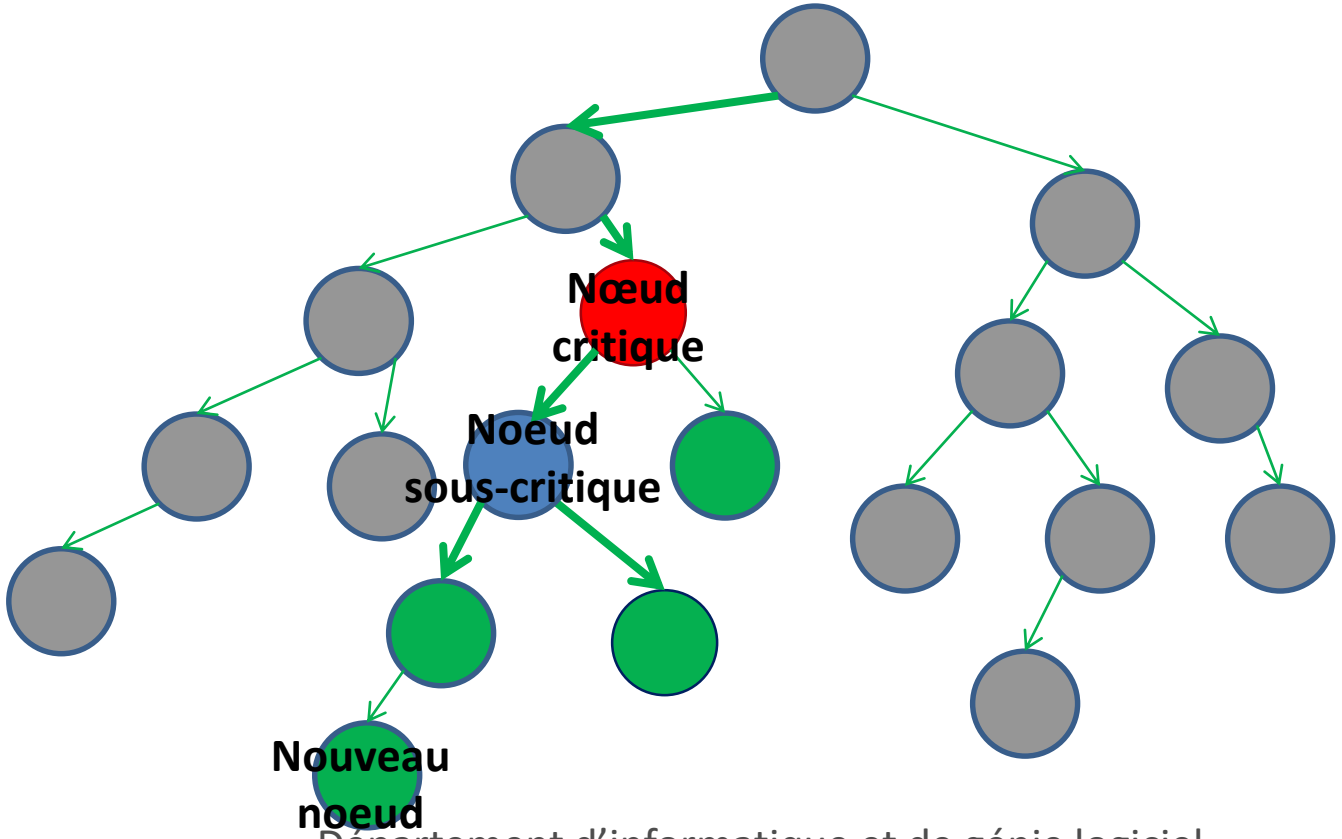
Les rotations

- Cas du zig-zagDroit.



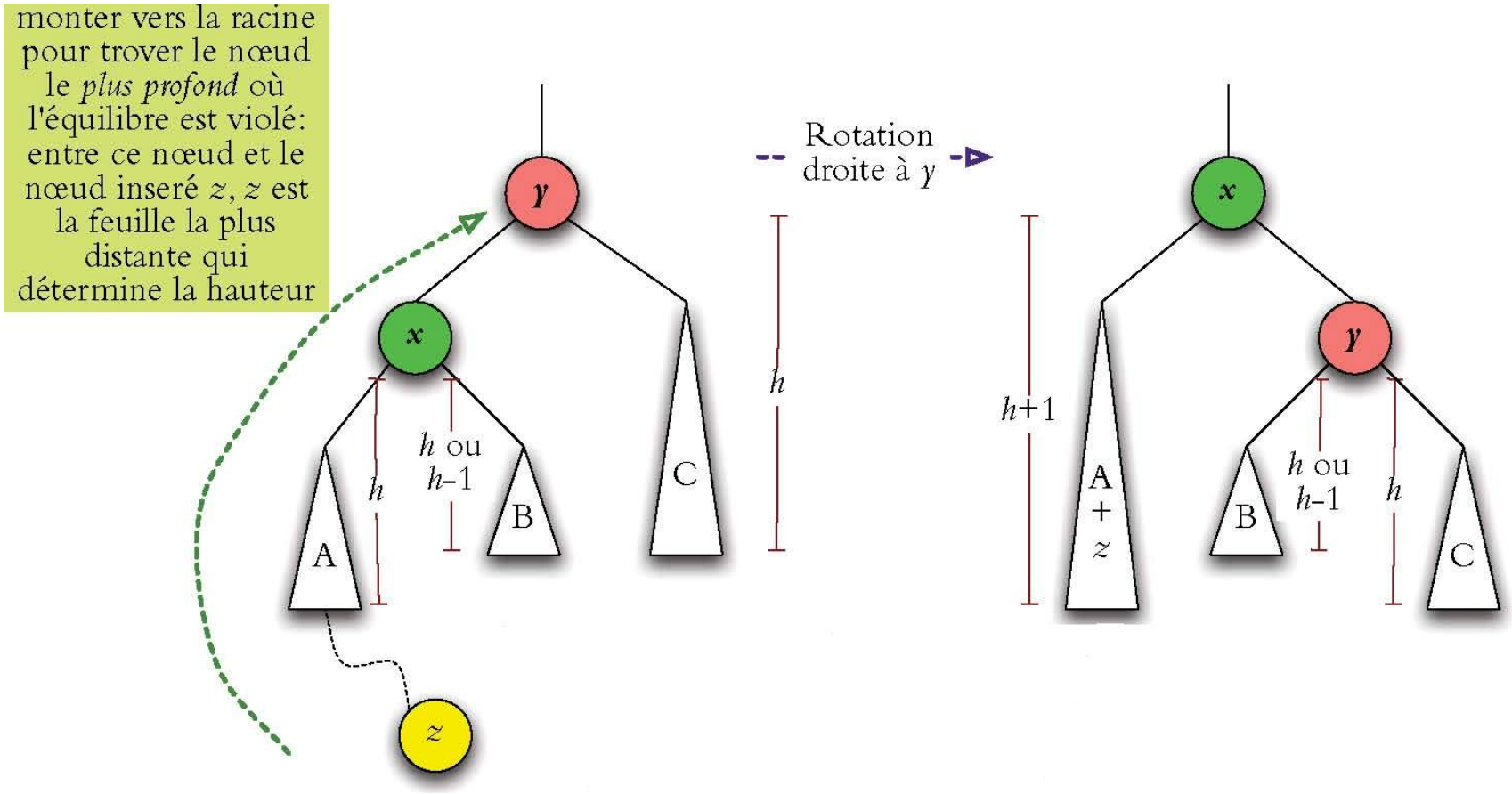
Les rotations

- Cas du zig-zigGauche.



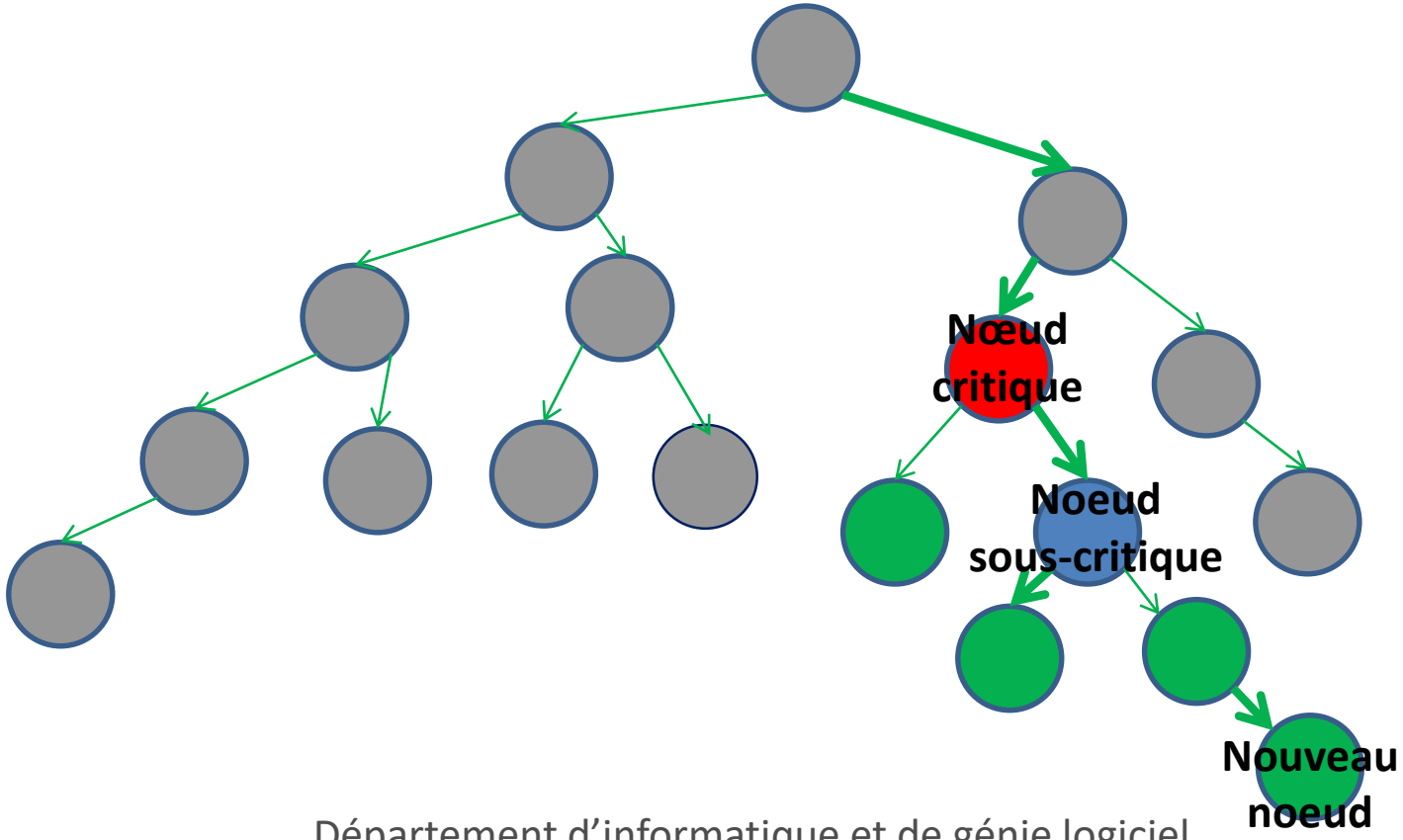
Rotation pour le cas zig-zigGauche

- Cas du **zig-zigGauche**. Les hauteurs avant et après le balancement.



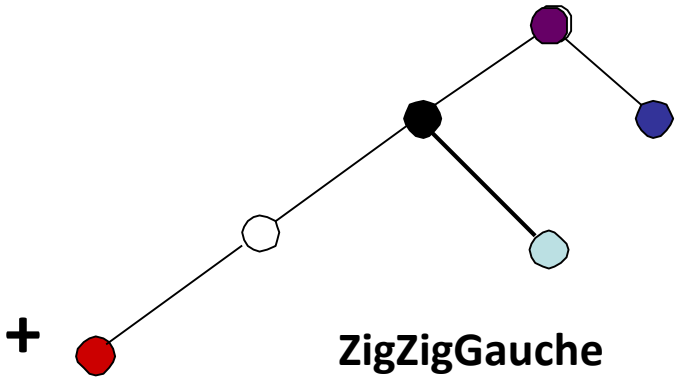
Les rotations

- Cas du zig-zigDroit.

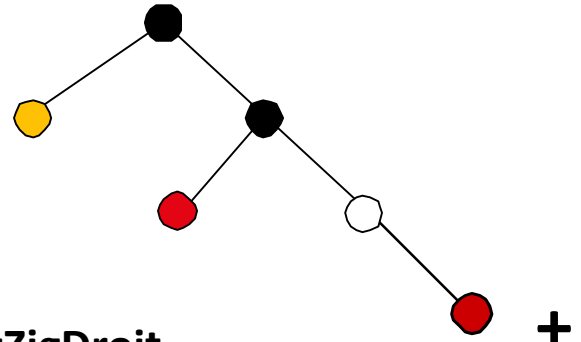


Quatre cas de déséquilibre

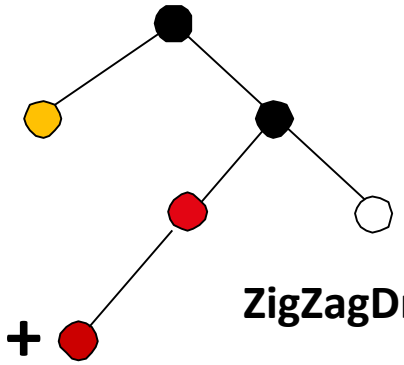
- Ici, noeud critique est la racine.



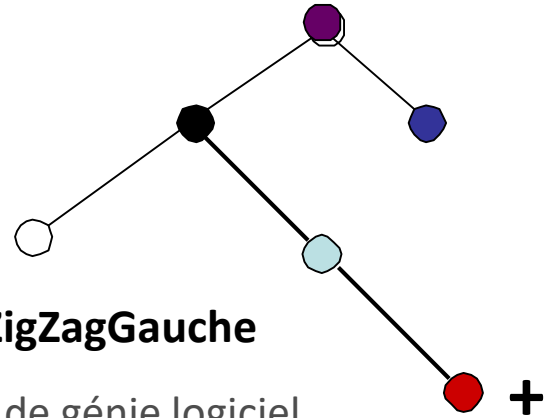
ZigZigGauche



ZigZigDroit



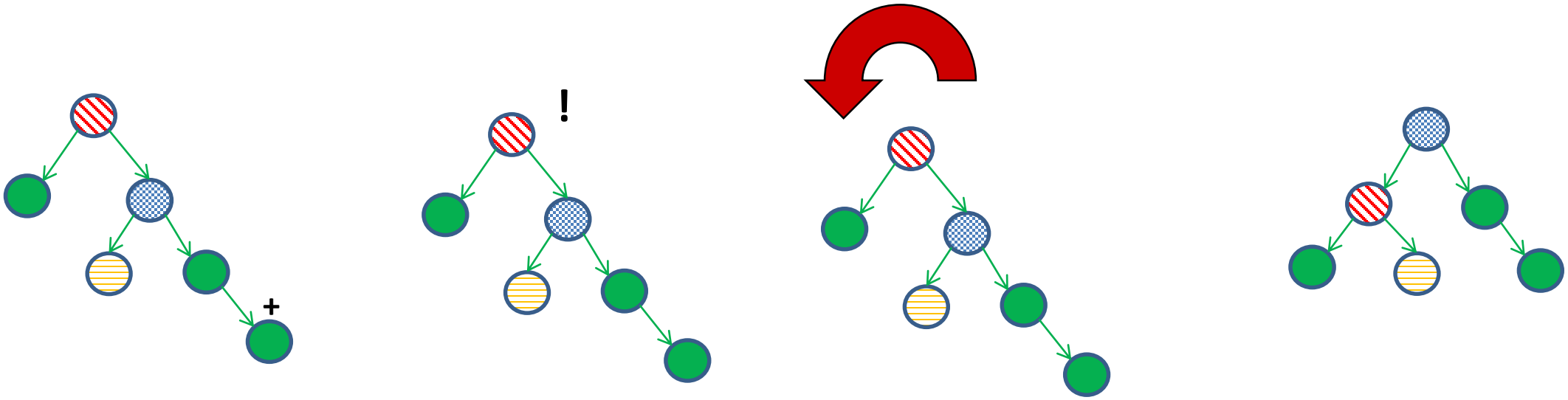
ZigZagDroit



ZigZagGauche

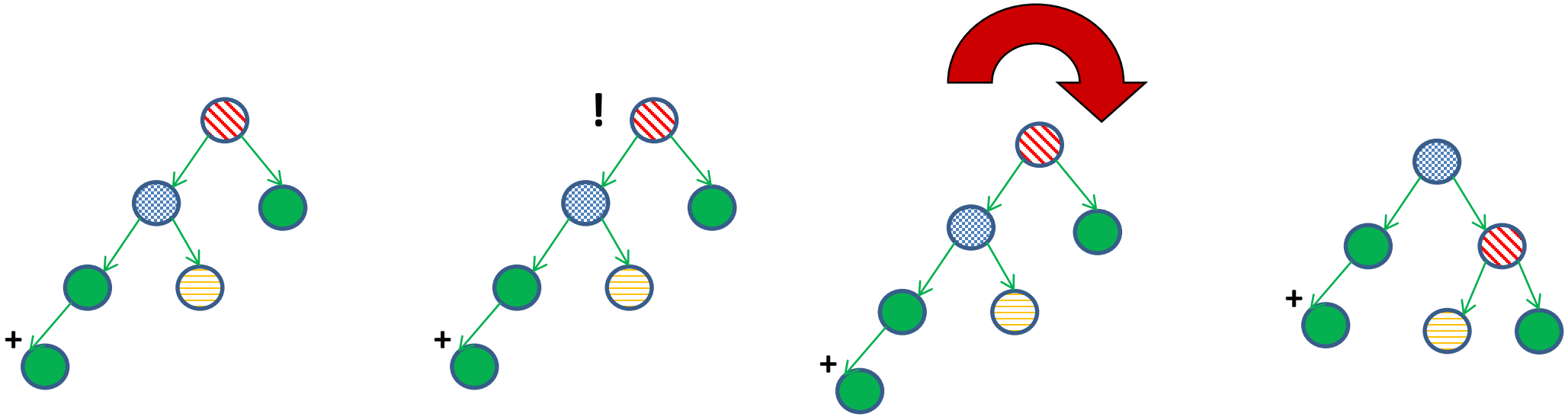
Balancement

- ZigZigDroit



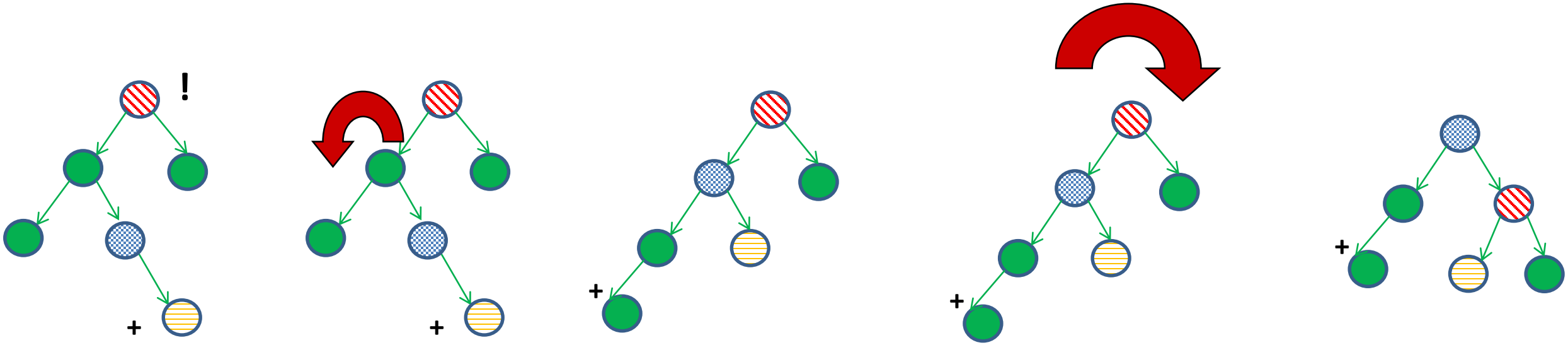
Balancement

- ZigZigGauche



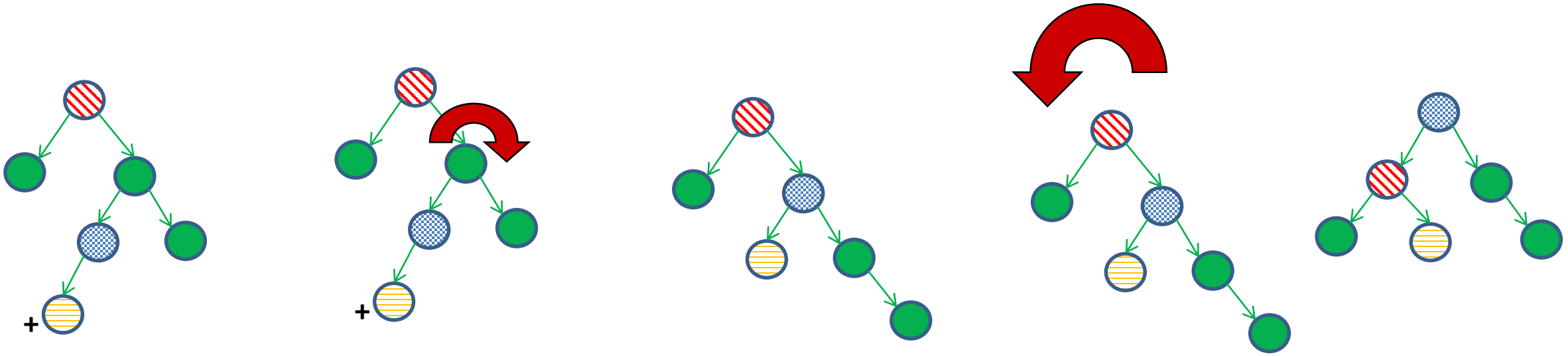
Balancement

- ZigZagGauche



Balancement

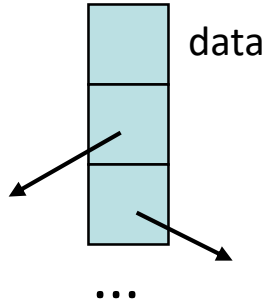
- ZigZagDroit



Modèle d'implémentation d'arbre AVL

```
template <typename E>
class Arbre
{
public:
        //..
private:
        // Les membres données
        Noeud* racine; //racine de l'arbre
        long nb_noeuds;
        //...
};
```

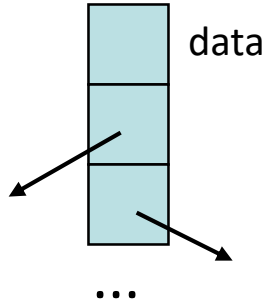
Modèle d'implantation par chaînage



Modèle d'implémentation d'arbre AVL

```
template <typename E>
class Arbre
{
private:
        // classe Noeud
        class Noeud
        {
        public:
                E data;
                Noeud* gauche;
                Noeud* droite;
                int hauteur;
                Noeud(const E & d) :
        gauche(0),data(d),droite(0),hauteur(0) { }
        };
};
```

Modèle d'implantation par chaînage



Insertion dans un arbre AVL

```
template<typename E>
void Arbre<E>::insererAVL(const E& data)
{
	_auxInsererAVL(racine, data);
}
```

Insertion dans un arbre AVL

```
template<typename E>
void Arbre<E>::_auxInsererAVL(Noeud*& arbre, const E& data)
{
	if(arbre == 0)
	{
		arbre = new Noeud(data);
		nb_noeuds++;
		return;
	}
	if(data < arbre->data)
		_auxInsererAVL(arbre->gauche, data);
	else if( arbre->data < data )
		_auxInsererAVL(arbre->droite, data);
	else
		throw logic_error("Les duplicatats sont interdits");
	_balancer(arbre); //balancer et mise à jour des hauteurs
}
```

```
template<typename E>
void Arbre<E>::_balancer(Noeud*& arbre)
{
	if (_debalancementAGauche(arbre)) {
		// Lorsque le débalancement est à gauche, on fait un
			zigZig lorsque le sousArbre penche à gauche OU est balancé.
		if (_sousArbrePencheADroite(arbre->gauche))
			{ _zigZagGauche(arbre); }
		else
			{ _zigZigGauche(arbre); }
	}
	else if (_debalancementADroite(arbre)) {
		// Lorsque le débalancement est à droite, on fait un
		// zigZig lorsque le sousArbre penche à droite OU est balancé
		if (_sousArbrePencheAGauche(arbre->droite))
		{ _zigZagDroit(arbre); }
		else
		{ _zigZigDroit(arbre); }
	else { //aucun débalancement; seulement m.à.j. des hauteurs
		if (arbre!=0)
		{arbre->hauteur = 1 + std::max(_hauteur(arbre->gauche),_hauteur(arbre->droite)); }
	}
}
```

Insertion dans un arbre AVL

```
template <typename E>
int Arbre<E>::_hauteur(Noeud* arbre) const
{
	if (arbre == 0)
	{
		return -1; // la hauteur du vide = -1
	}
	return arbre->hauteur;
}
```

Insertion dans un arbre AVL

```
template<typename E>
bool Arbre<E>::_debalancementAGauche(Noeud* arbre) const
{
	if (arbre == 0)
	{
		return false;
	}
	return 1 < _hauteur(arbre->gauche) - _hauteur(arbre->droite);
}
// similairement pour _debalancementADroite()
```

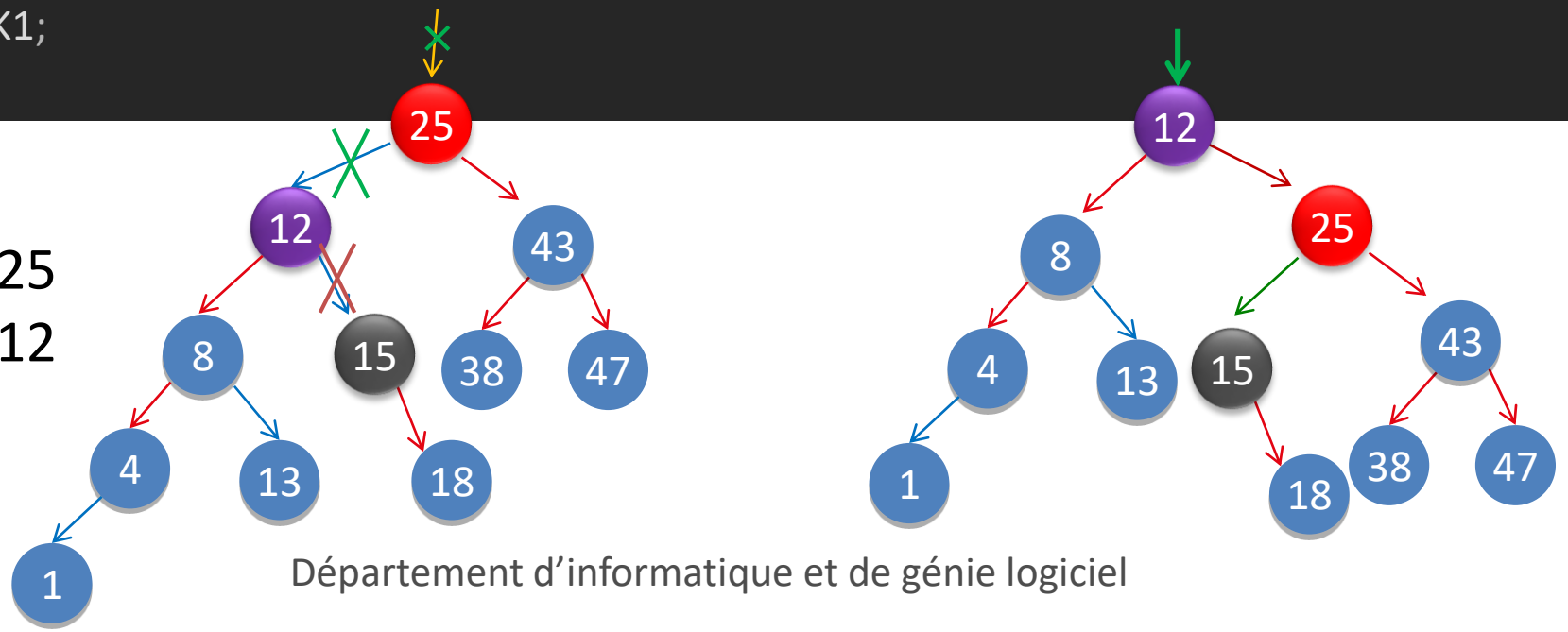
Insertion dans un arbre AVL

```
template<typename E>
bool Arbre <E>::_sousArbrePencheADroite(Noeud* arbre) const
{
	if (arbre == 0)
	{
		return false;
	}
	return _hauteur(arbre->gauche) < _hauteur(arbre->droite);
}
// similairement pour _sousArbrePencheAGauche()
```

Implémentation de zigZigGauche

```
template <typename E>
void Arbre<E>::_zigZigGauche(Noeud*& K2)
{
	Noeud* K1 = K2->gauche;
	K2->gauche = K1->droite;
	K1->droite = K2;
	K2->hauteur = 1+ max(_hauteur(K2->gauche),_hauteur(K2->droite));
	K1->hauteur = 1 + max(_hauteur(K1->gauche), K2->hauteur);
	K2 = K1;
}
```

K2 est le noeud 25
K1 est le noeud 12

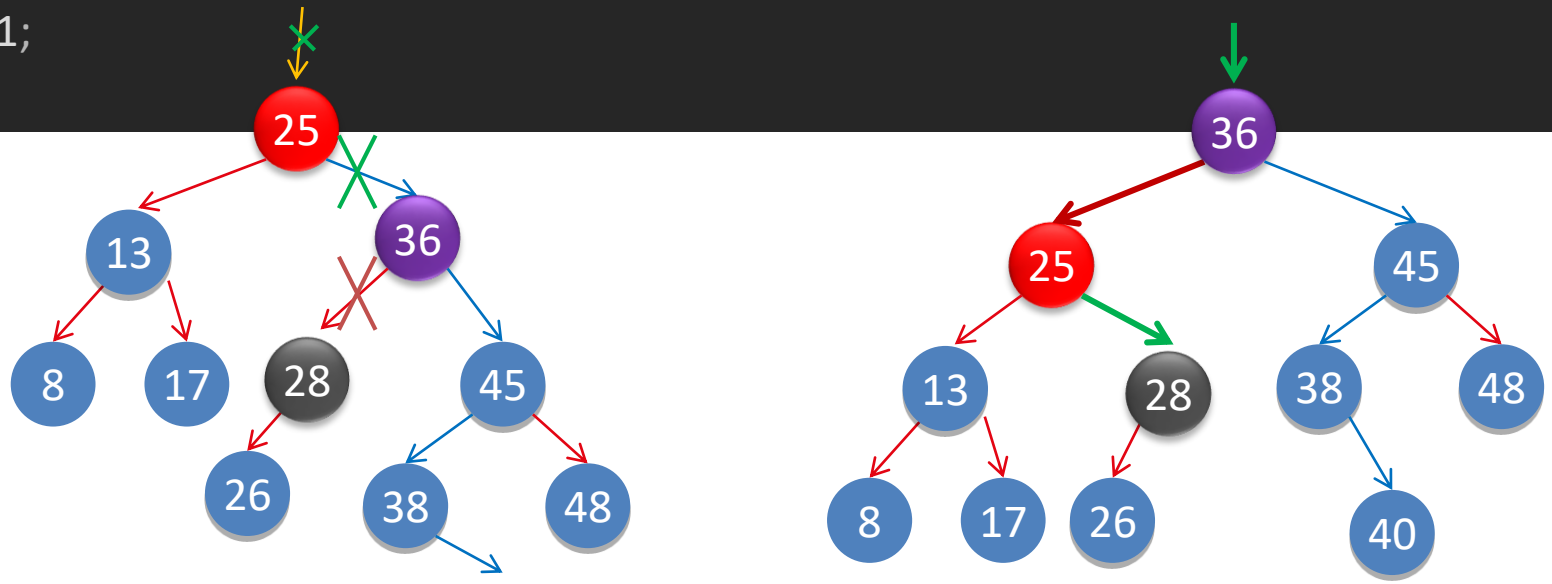


Implémentation de zigZigDroit

```
template <typename E>
void Arbre<E>::_zigZigGauche(Noeud*& K2)
{
	Noeud* K1 = K2->droite;
	K2->droite = K1->gauche;
	K1->gauche = K2;
	K2->hauteur = 1 + max(_hauteur(K2->droite), _hauteur(K2->gauche));
	K1->hauteur = 1 + max(_hauteur(K1->droite), K2->hauteur);
	K2 = K1;
}
```

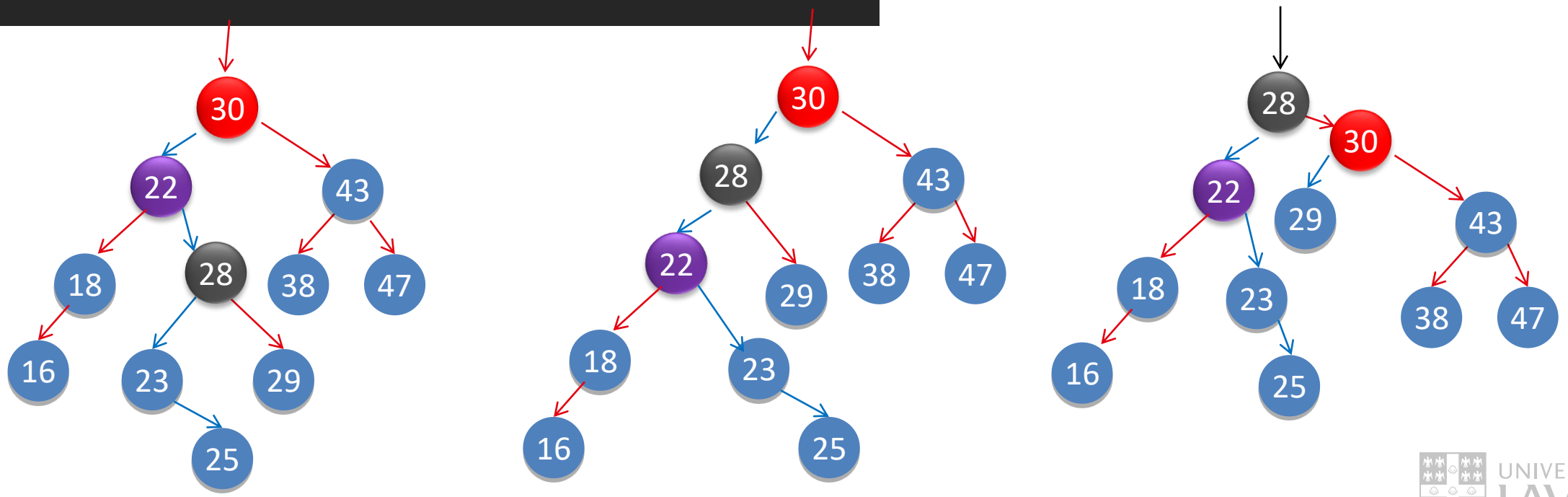
K2 est le noeud 25

K1 est le noeud 36



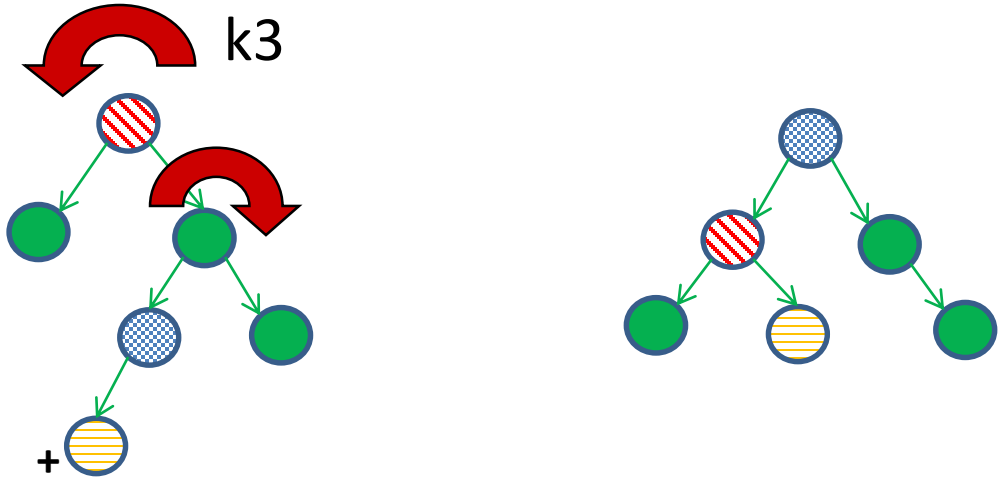
Implémentation de zigZagGauche

```
template <typename E>
void Arbre<E>::_zigZagGauche(Noeud*& K3)
{
	_zigZigDroit(K3->gauche);
	_zigZigGauche(K3);
}
```



Implémentation de zigZagDroit

```
template <typename E>
void Arbre<E>::_zigZagDroit(Noeud*& K3)
{
        _zigZigGauche(K3->droite);
        _zigZigDroit(K3);
}
```



Synthèse

- Arbres AVL : équilibrés
  - pour supporter efficacement les opérations de recherche, d'insertion et de suppression
  - facteur d'équilibre
    - |hauteur(sous-arbre droit) - hauteur(sous-arbre gauche)| = k
    - $HB[k]$ lorsque ≤ k
    - AVL → arbres $HB[1]$
  - Déséquilibre → balancement
    - Nœud critique
    - Rotations : zigzig droit/gauche, zigzag droit/gauche
    - Implémentation

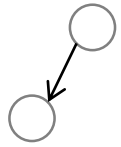
Analyse: complexité pour l'insertion dans AVL

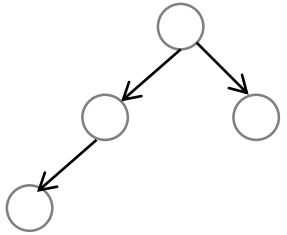
- Les opérations suivantes se font en temps $O(1)$:
  - Insertion d'une donnée à partir d'une feuille
  - Vérification du débalancement (lecture des hauteurs) d'un noeud
  - Mise à jour de la hauteur d'un noeud
  - Le rebalancement (zigZig et zigZag) à une position donnée
- Soit $h(n)$ = hauteur de l'arbre AVL de n nœuds.
- Or, la profondeur de la feuille la moins profonde dans un arbre AVL de hauteur h est $\geq \lceil \boldsymbol{h/2} \rceil$ (voir exercices).
- Les opérations suivantes se font donc en O($h(n)$) :
  - trouver le point d'insertion (de la racine à une feuille)
  - Remonter du point d'insertion jusqu'à la racine (en rebalançant lorsque nécessaire)
- L'insertion se fait donc en $O(h(n))$
- Or, nous allons voir que $h(n) \in O(\log n)$ pour un arbre AVL.
- L'insertion d'une donnée dans un arbre AVL se fait donc en $O(\log n)$

Analyse: hauteur d'un arbre AVL

- **Théorème**: La hauteur d'un arbre AVL de n nœuds est en $O(\log n)$
- Démonstration :
  - Soit $f(h) \equiv$ nombre minimum de nœuds que doit posséder un arbre AVL de hauteur h
  - Puisque $n \geq f(h)$, nous aurons que $h \leq f^{-1}(n)$ si f^{-1} est une fonction non décroissante. Cela nous donnera donc une borne supérieure sur la hauteur h de l'arbre de n nœuds.
  - En plus de $f(0) = 1$, nous avons les cas de base suivants:

f(1) = 2

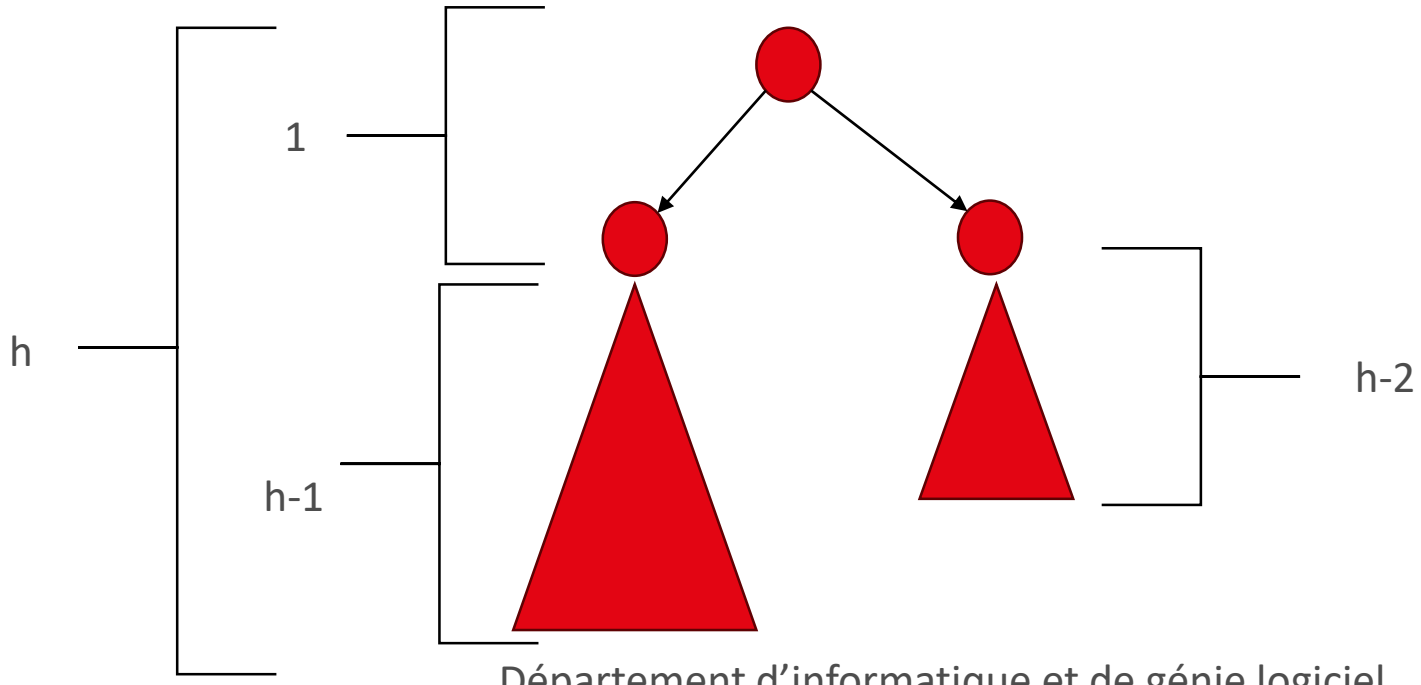




f(2) = 4

Analyse: hauteur d'un arbre AVL

- Pour tout arbre AVL de hauteur $h \geq 2$, le nombre minimum de nœuds est réalisé lorsque l'un des sous arbres de la racine a une hauteur plus petite que l'autre.
- Donc pour tout $h \geq 2$, on a: $f(h) = 1 + f(h-1) + f(h-2)$
  - (le 1 vient du fait que l'on compte la racine)



Analyse: hauteur d'un arbre AVL

- Cette dernière relation implique que $f(h)$ est strictement croissant avec $f(h) > f(h-1) > f(h-2) > \ldots$
- Donc $f(h) = 1 + f(h-1) + f(h-2)$ implique $f(h) > 2f(h-2)$ pour tout $h \geq 2$.
- Nous avons alors:
  - $f(h) > 2f(h-2)$ pour tout $h \geq 2$
  - $> 2\,(\,2f(h-4)\,)$ pour tout $h \geq 4$
  - $> 2\,(\,2\,(\,2f(h-6)\,)\,)$ pour tout $h \geq 6$
  - $\ldots$
  - $> 2^k\, f(h-2k)$ pour tout $h \geq 2k$
  - $= 2^k f(2)$ pour $h = 2k + 2$ (donc pour h **pair**)
  - $= 2^{(h/2)-1}\, 4 = 2^{(h/2)+1}$ (en utilisant $f(2) = 4$ et $k = (h/2) - 1$)

Analyse: hauteur d'un arbre AVL

- $f(h) = 2^{(h/2)+1}$

- Donc $log_2(f(h)) > (h/2) + 1$.
- Alors $h < 2log_2(f(h)) - 2 \leq 2log_2(n) - 2$.
- Donc $h \in O(\log n)$ lorsque h est pair.

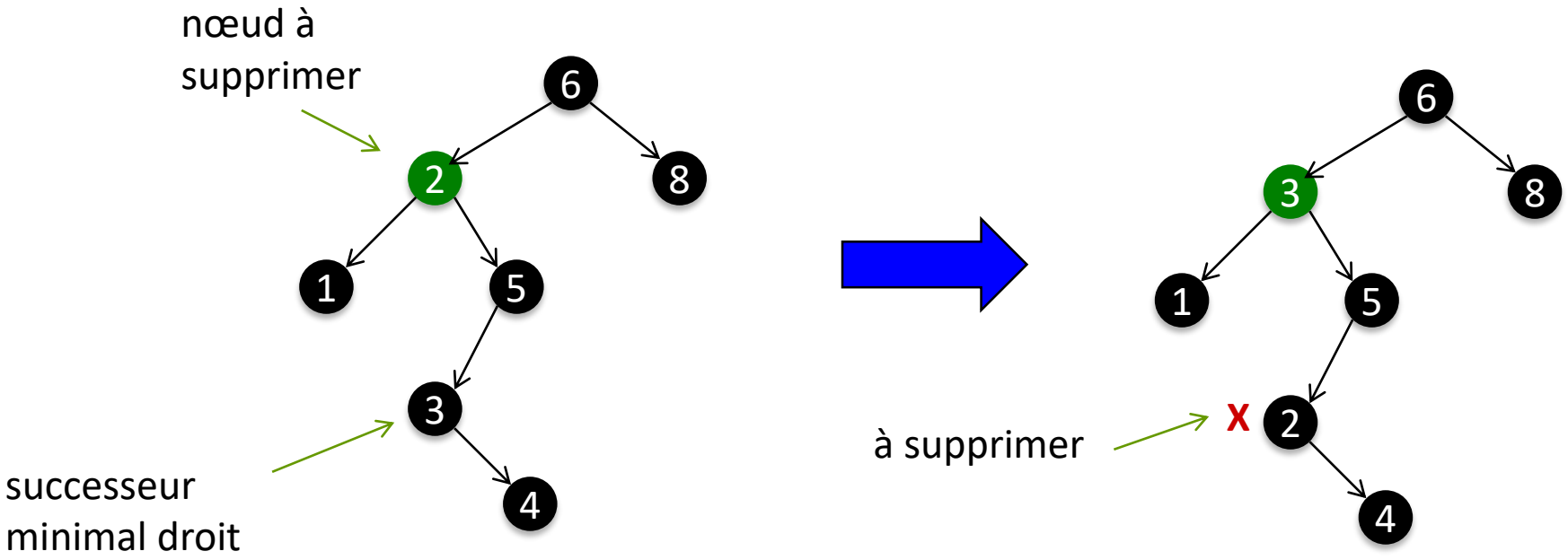
Analyse: hauteur d'un arbre AVL

- Lorsque h est impair, retournons à :
  - $f(h) > 2^k f(h - 2k)$ pour tout h ≥ 2k
- Examinons alors le cas $h = 2k + 1$ (donc h est impair)
- Alors:
  - $f(h) > 2^{\left((h-1)/2\right)} f(1) = 2 * 2^{\left((h-1)/2\right)} = 2^{\left((h+1)/2\right)}$
  - Donc $log_2(f(h)) > (h + 1)/2$
  - Donc $h < 2log_2(f(h)) - 1 \leq 2log_2(n) - 1$
  - Donc $h \in O(\log n)$ lorsque **h est impair**
  - Donc $h \in O(\log n)$ dans **tous les cas.**
- CQFD

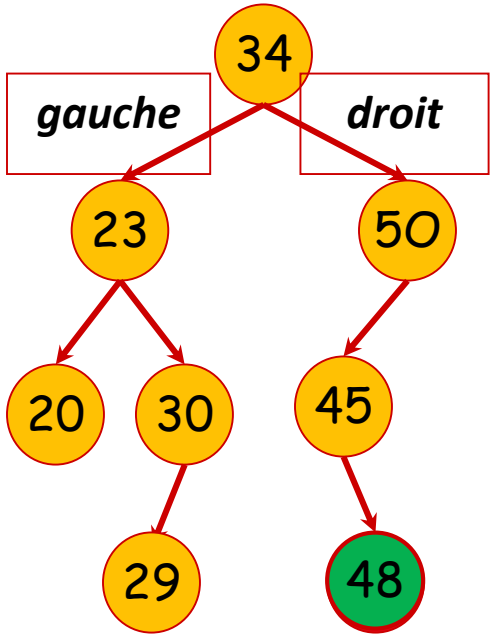
Enlèvement dans un arbre AVL

- Pour supprimer un nœud dans un arbre AVL, il y a deux cas simples et un cas compliqué:
  - Premier cas simple: le nœud à supprimer est une feuille.
    - Dans ce cas, il suffit de le supprimer directement.
  - Deuxième cas simple: le nœud à supprimer possède un seul enfant.
    - Dans ce cas, il suffit de le remplacer par son seul enfant.
  - Cas compliqué: le nœud à supprimer possède deux enfants.
    - Dans ce cas, il faut d'abord échanger ce nœud avec son successeur minimal à droite, ce qui nous mènera nécessairement à l'un des deux cas simples car ce successeur minimal à droite n'a pas d'enfant à gauche!

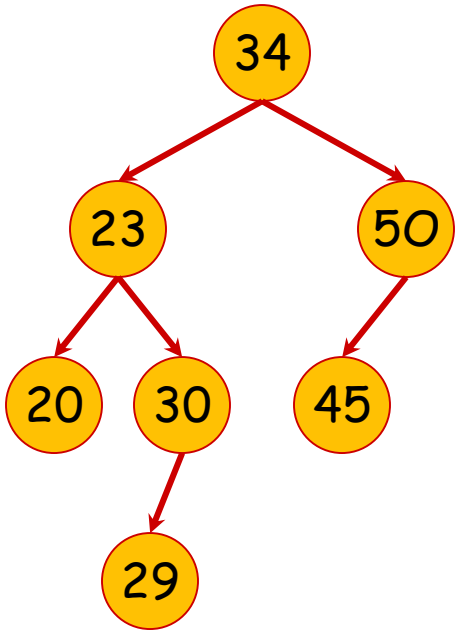
Enlèvement dans un arbre AVL



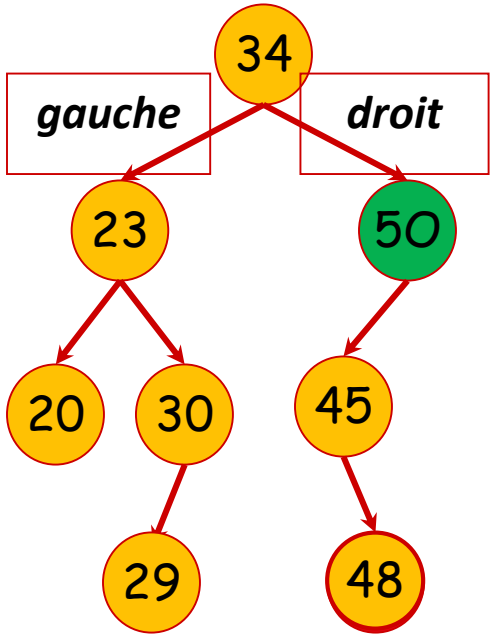
Enlèvement dans un arbre AVL



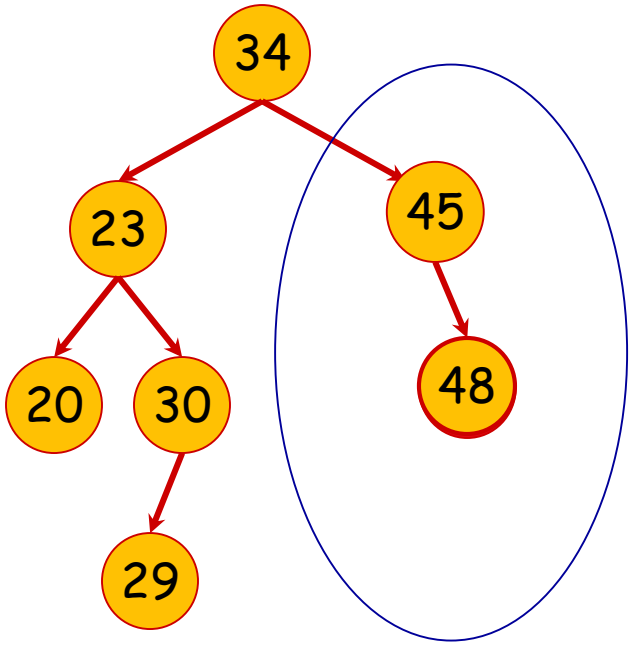
48 une **feuille** : trivial



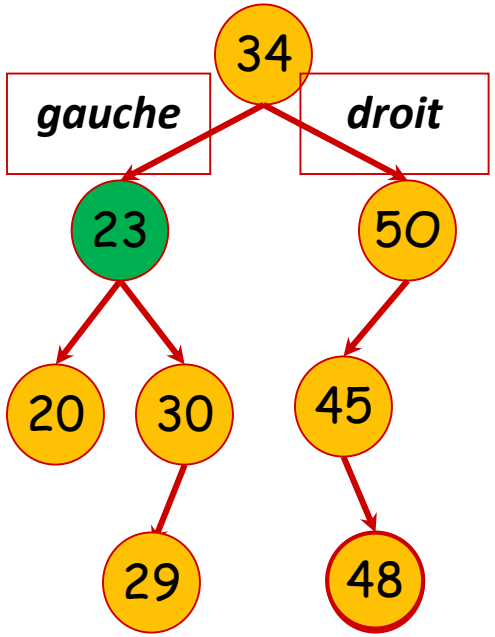
Enlèvement dans un arbre AVL



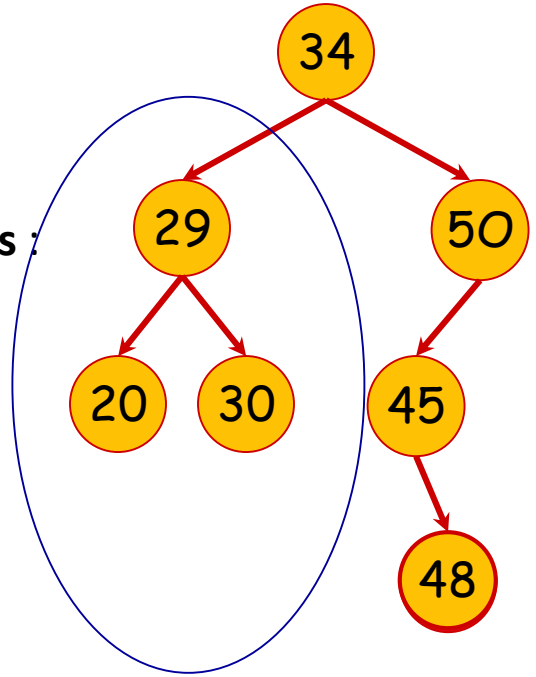
un **nœud ayant 1 seul enfant** :
on le remplace par son unique fils



Enlèvement dans un arbre AVL

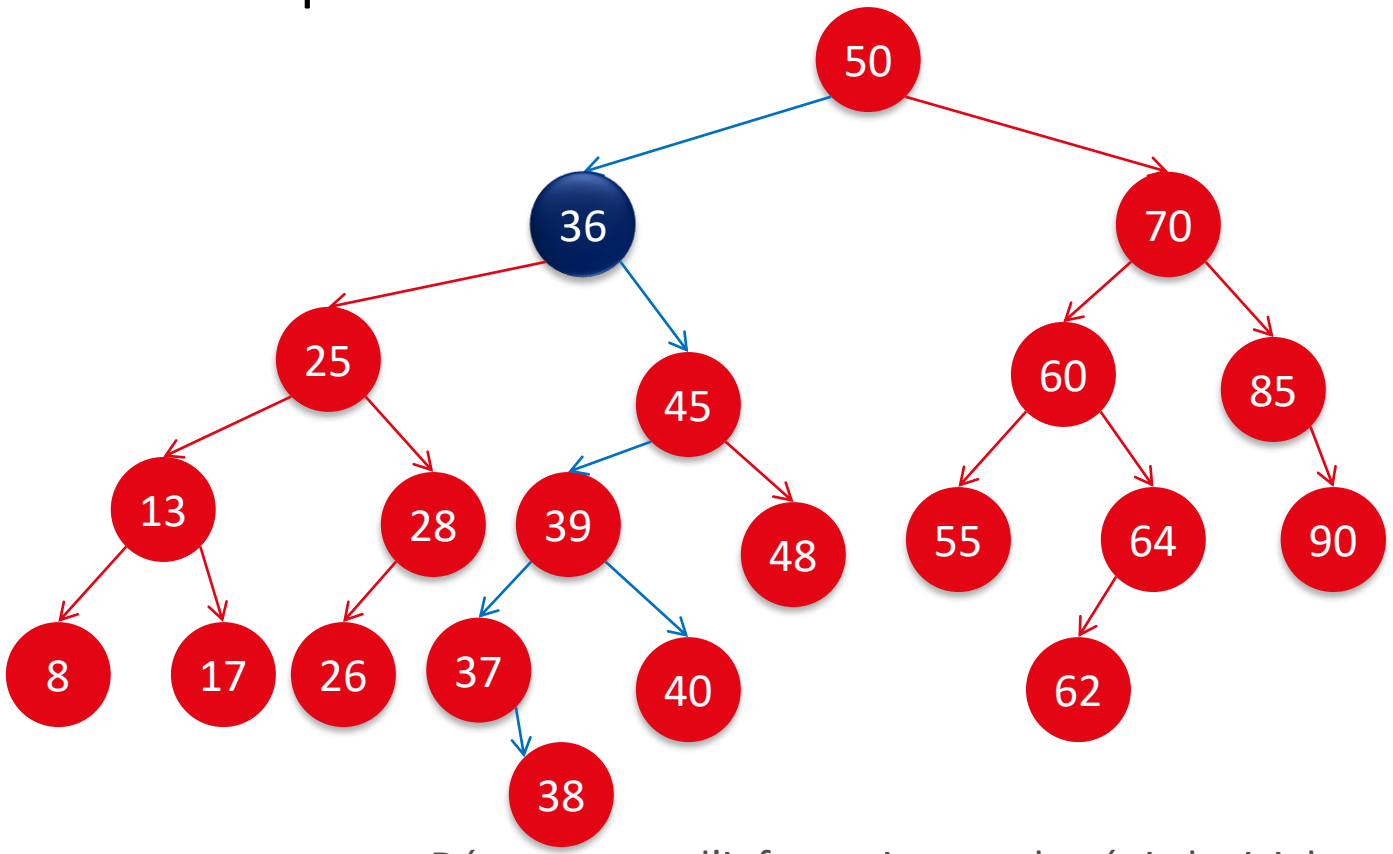


un **nœud ayant 2 enfants** : on lui donne la valeur minimale de son sous-arbre droit (ex 29) et on supprimer le nœud qui a cette valeur



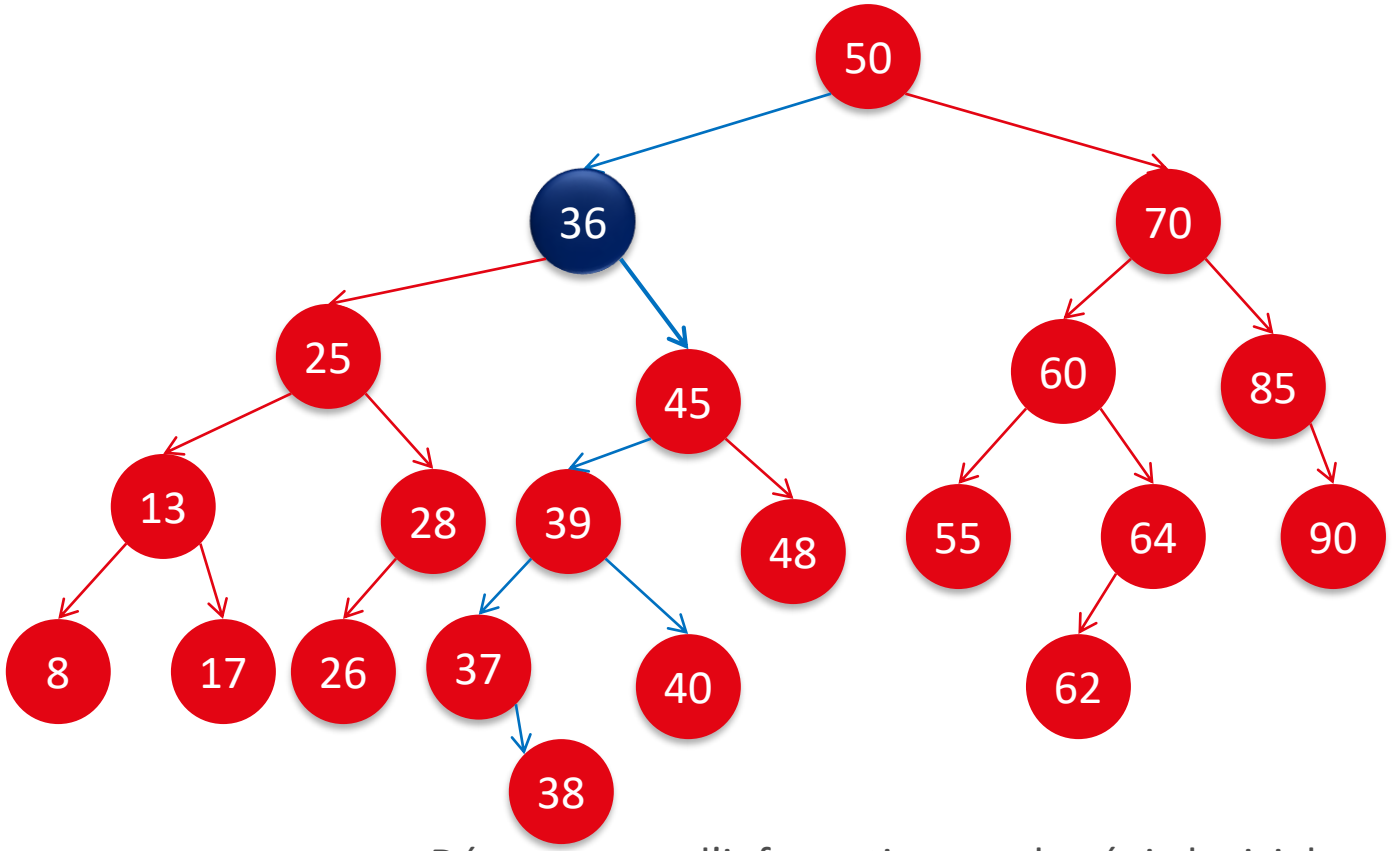
Enlèvement dans un arbre AVL

- Supprimons le nœud 36 de cet arbre-ci.
- Il faut d'abord le repérer avec une recherche conventionnelle à partir de la racine.



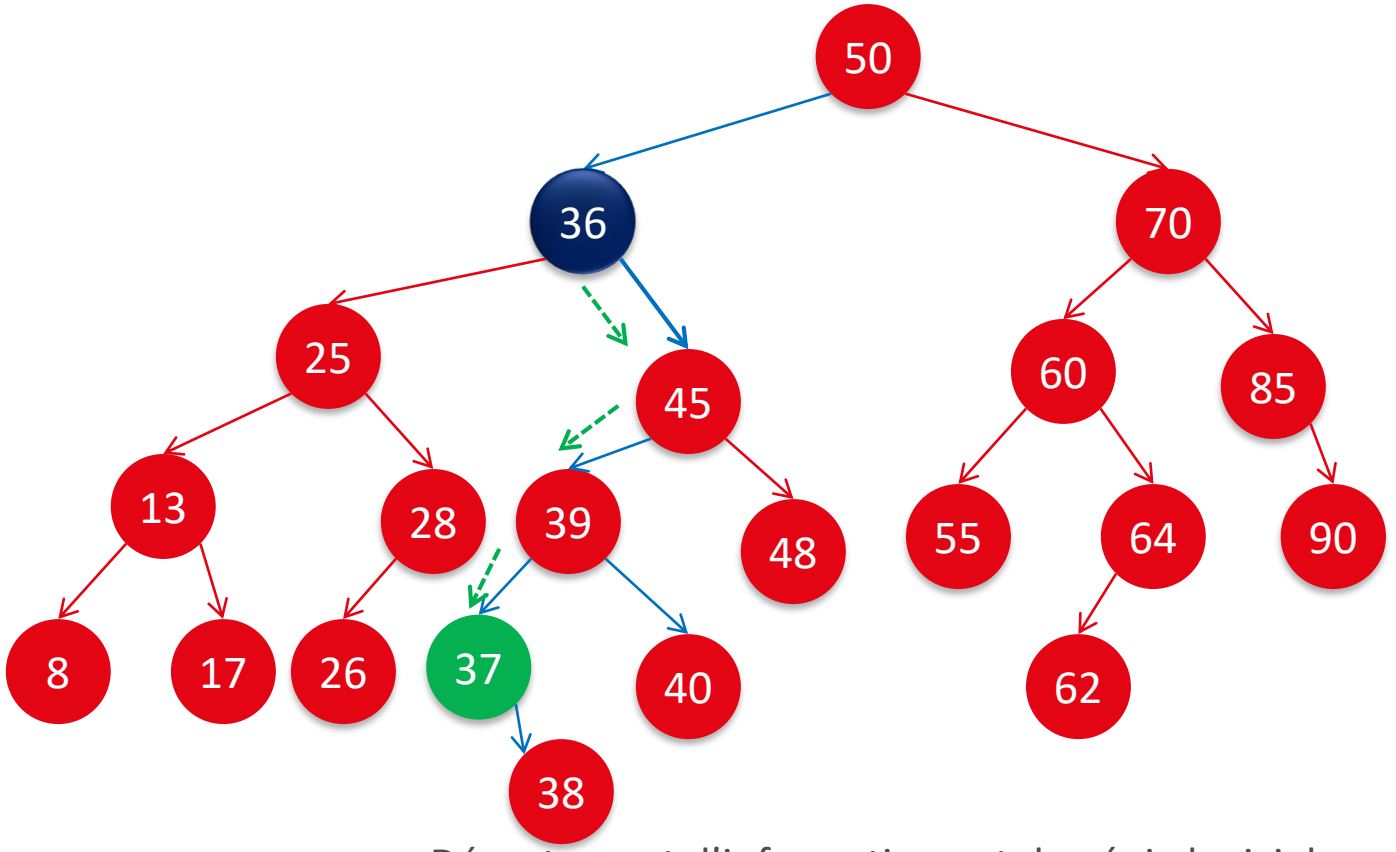
Enlèvement dans un arbre AVL

- Puis, nous établissons qu'il s'agit d'un cas compliqué car le nœud à supprimer a deux enfants.



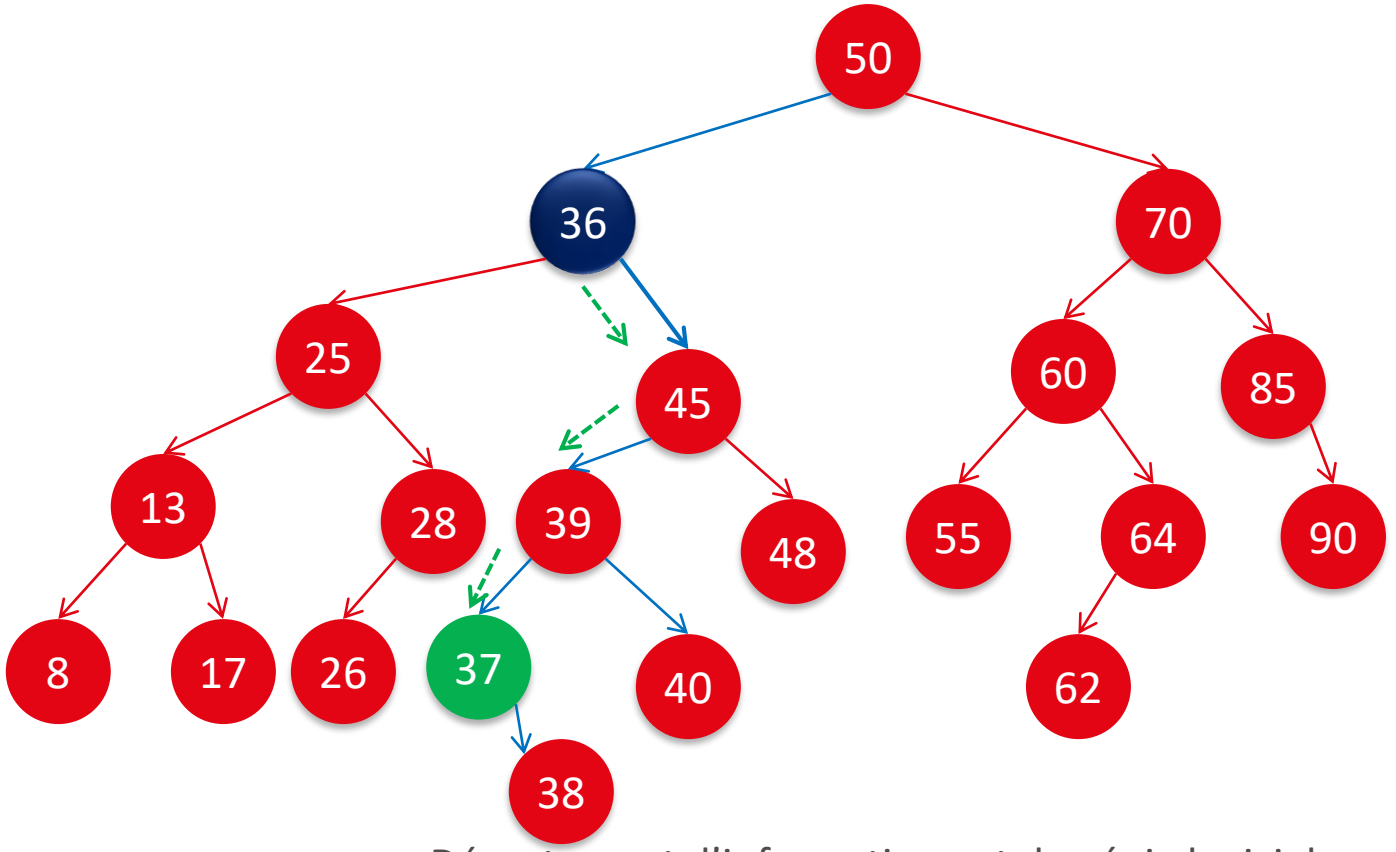
Enlèvement dans un arbre AVL

- Il faut donc d'abord retrouver son successeur minimal à droite à l'aide d'une boucle simple (une fois à droite, plein de fois à gauche).



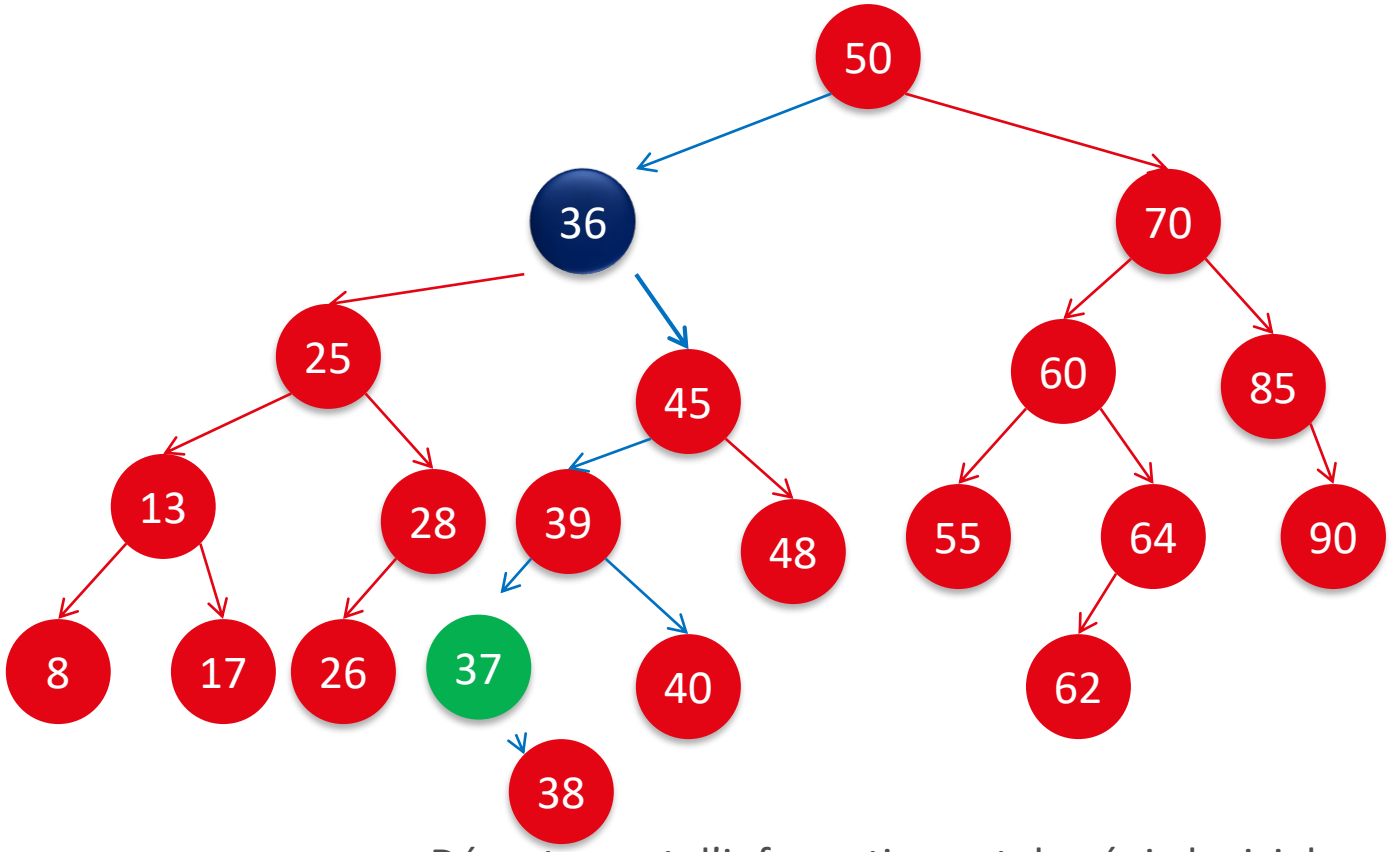
Enlèvement dans un arbre AVL

- Puis on les échange.



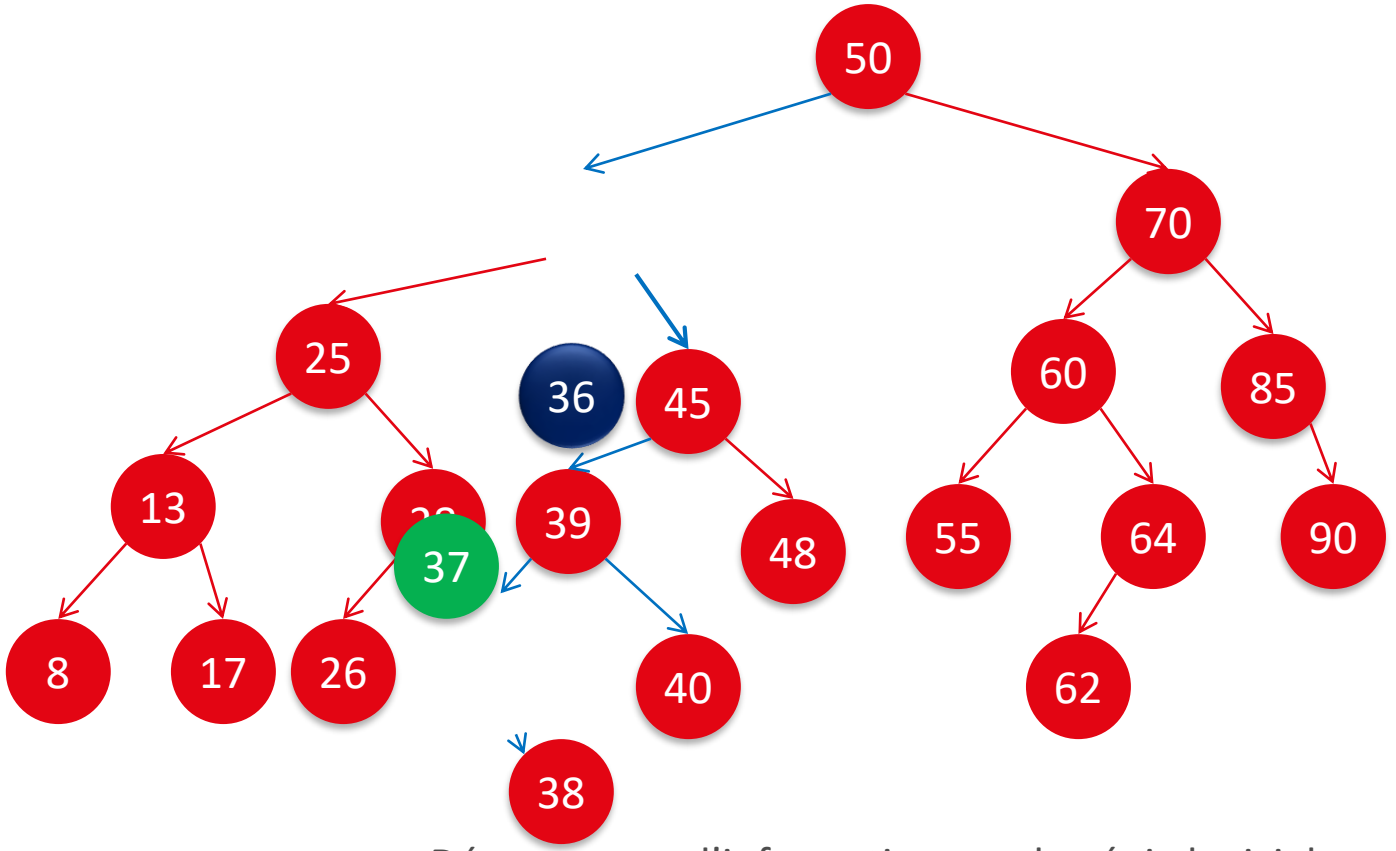
Enlèvement dans un arbre AVL

- Puis on les échange.



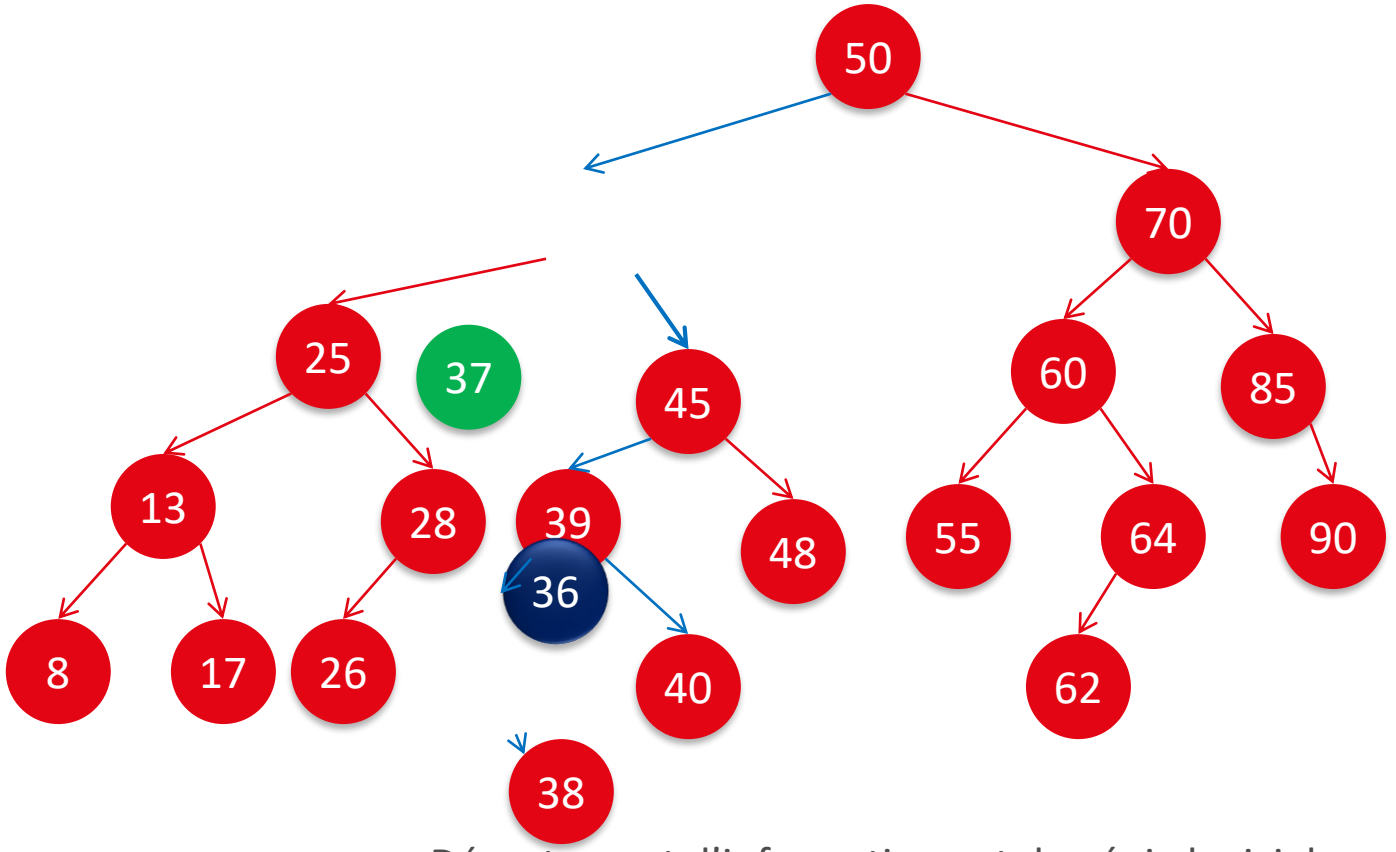
Enlèvement dans un arbre AVL

- Puis on les échange.



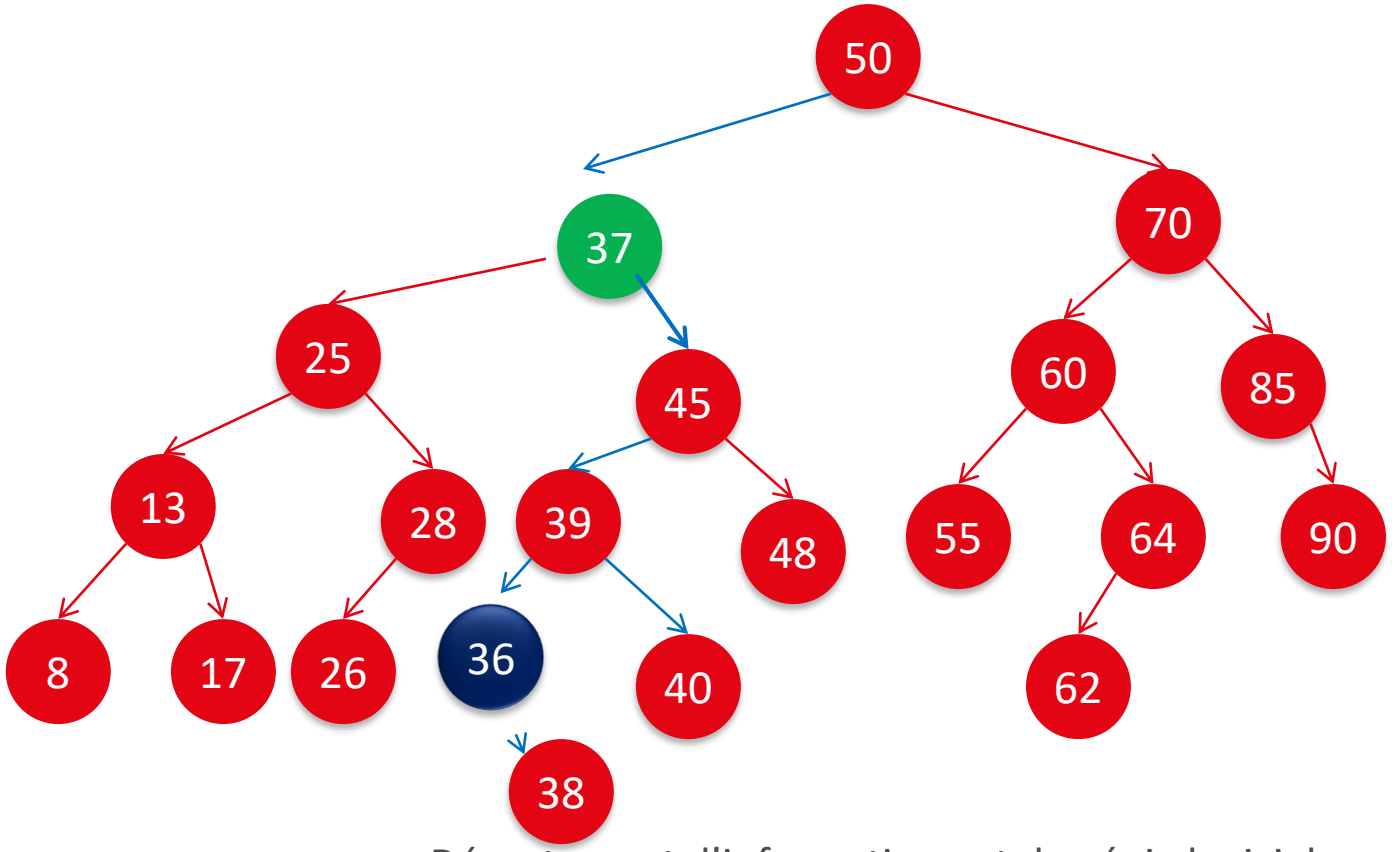
Enlèvement dans un arbre AVL

- Puis on les échange.



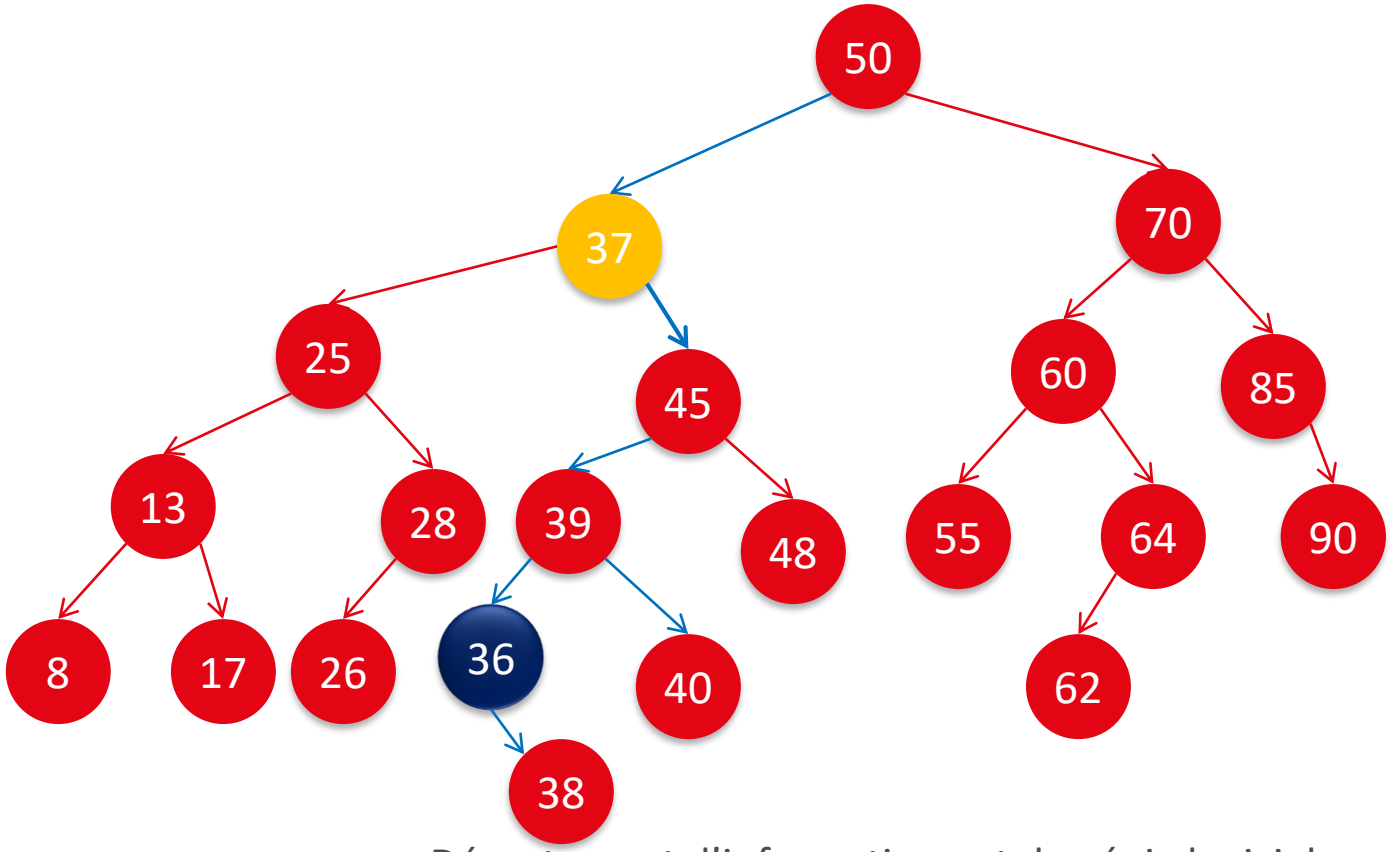
Enlèvement dans un arbre AVL

- Puis on les échange.



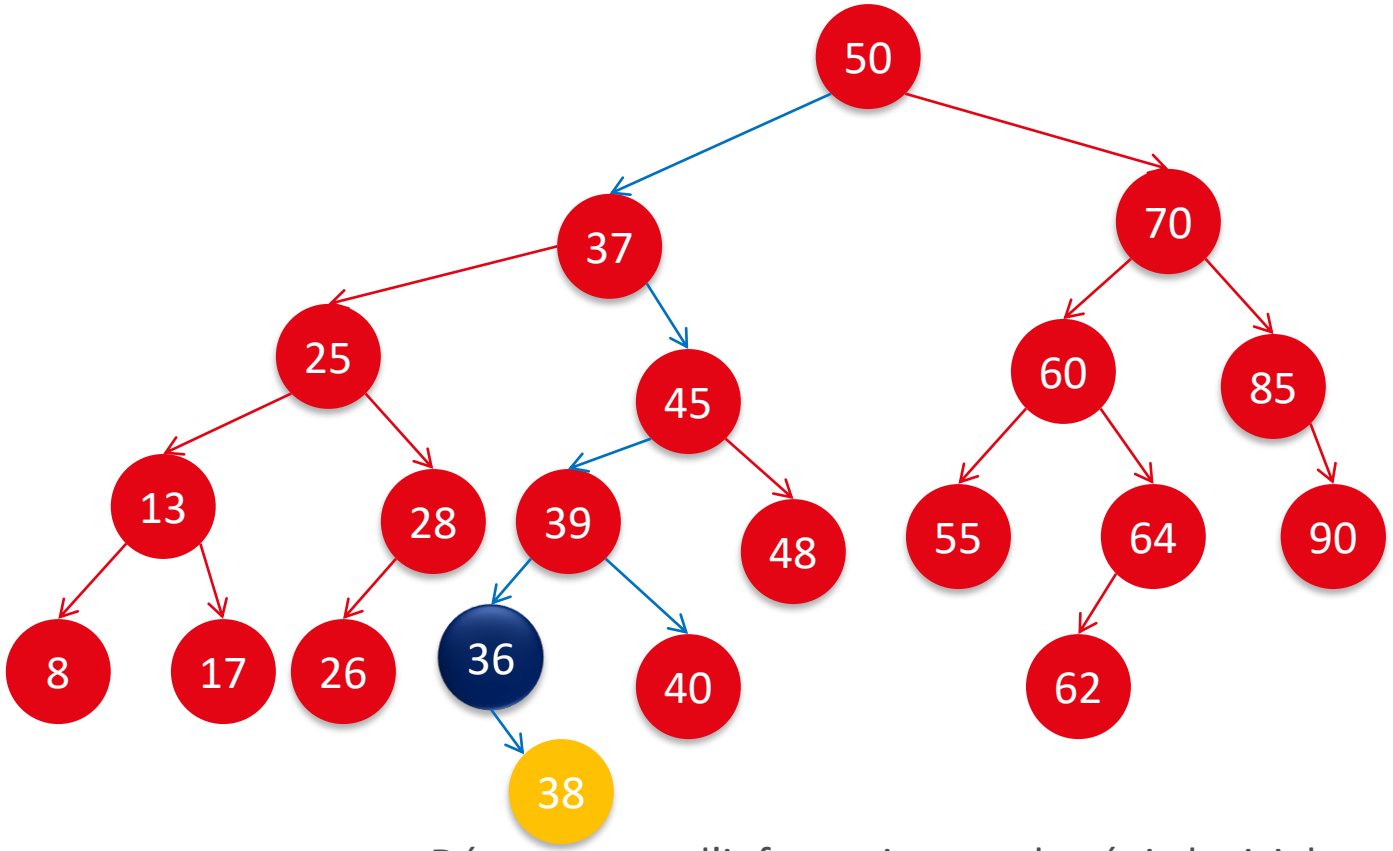
Enlèvement dans un arbre AVL

- Puis on les échange.



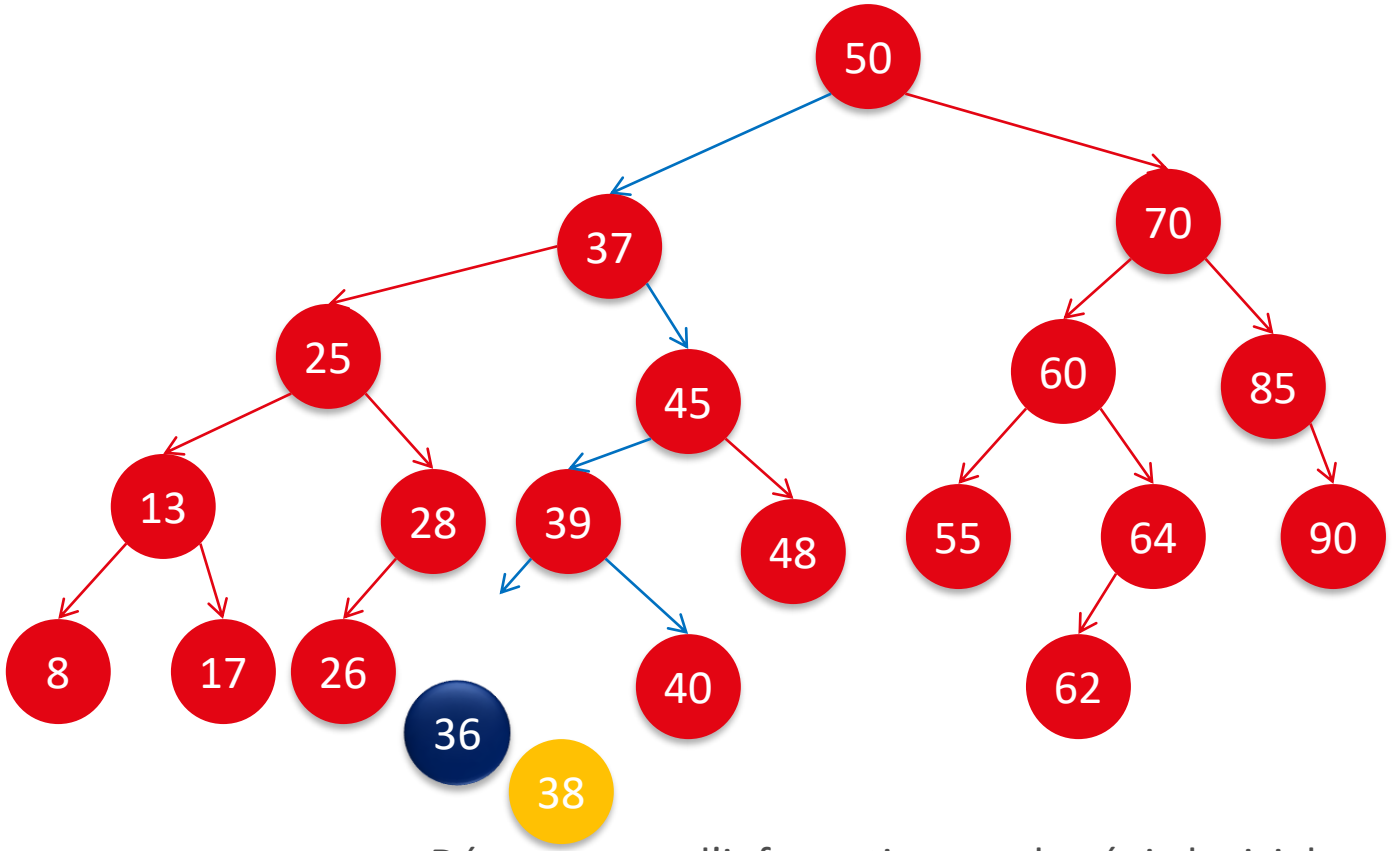
Enlèvement dans un arbre AVL

- Il s'agit maintenant d'enlever le noeud 36, on arrive nécessairement à l'un des deux cas simples.



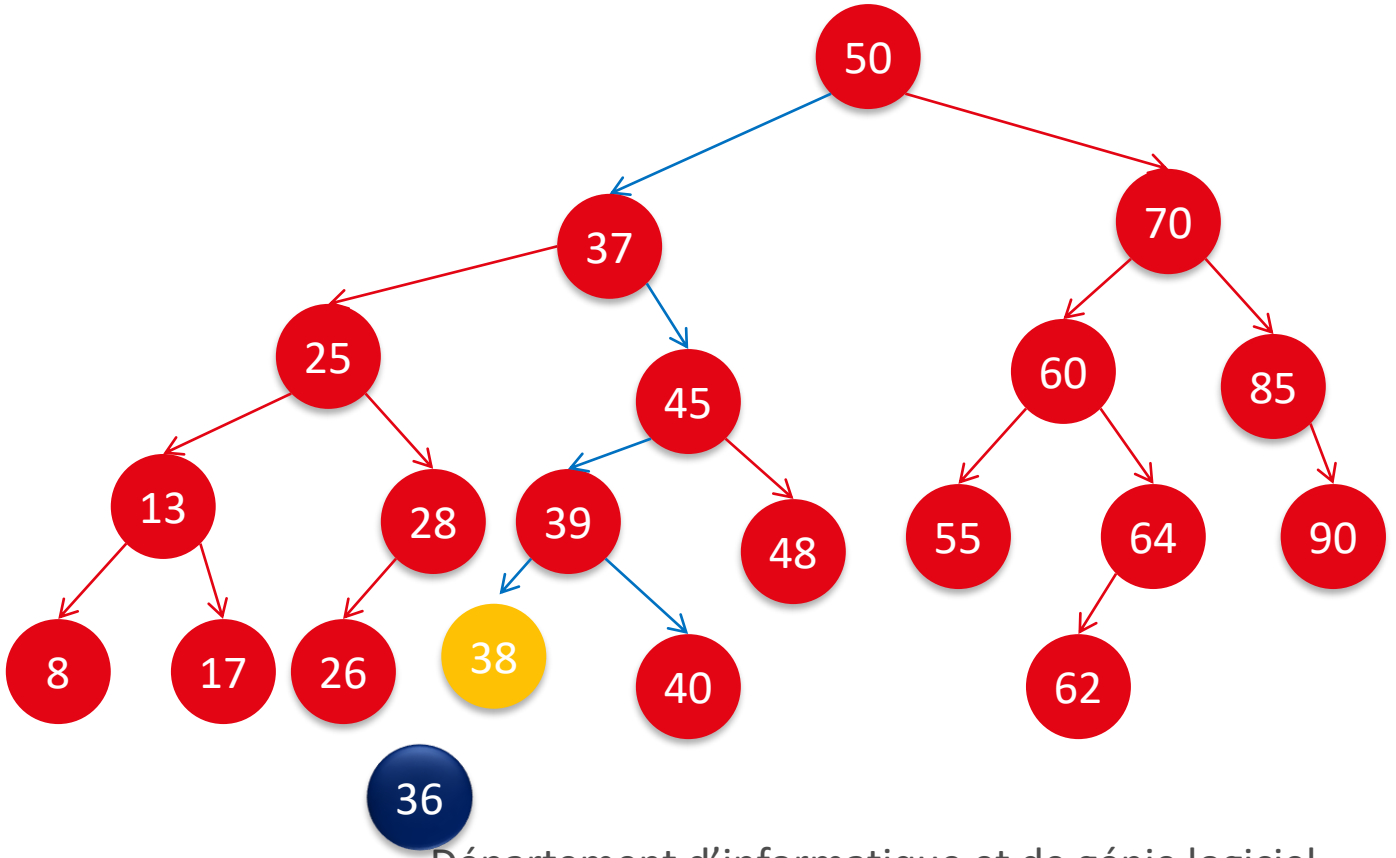
Enlèvement dans un arbre AVL

- Il s'agit maintenant d'enlever le noeud 36, on arrive nécessairement à l'un des deux cas simples.



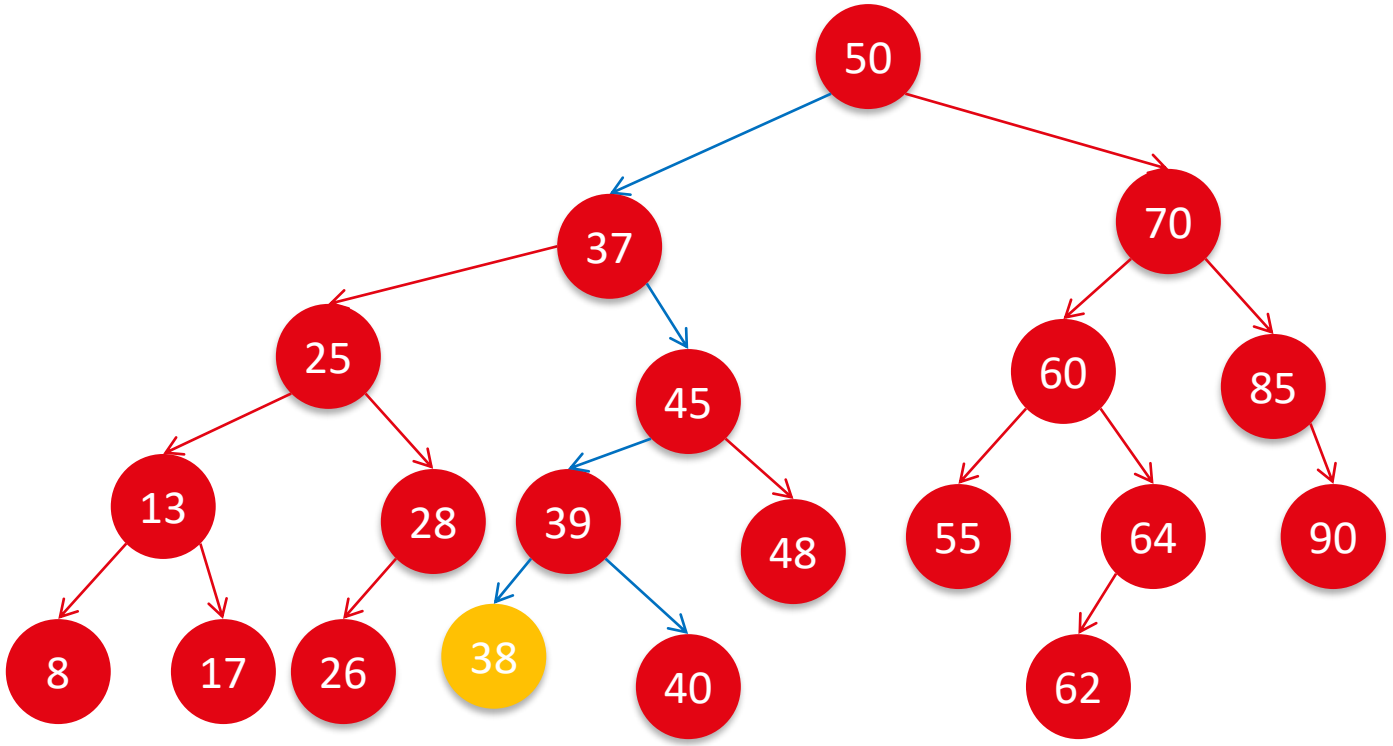
Enlèvement dans un arbre AVL

- Il s'agit maintenant d'enlever le noeud 36, on arrive nécessairement à l'un des deux cas simples.



Enlèvement dans un arbre AVL

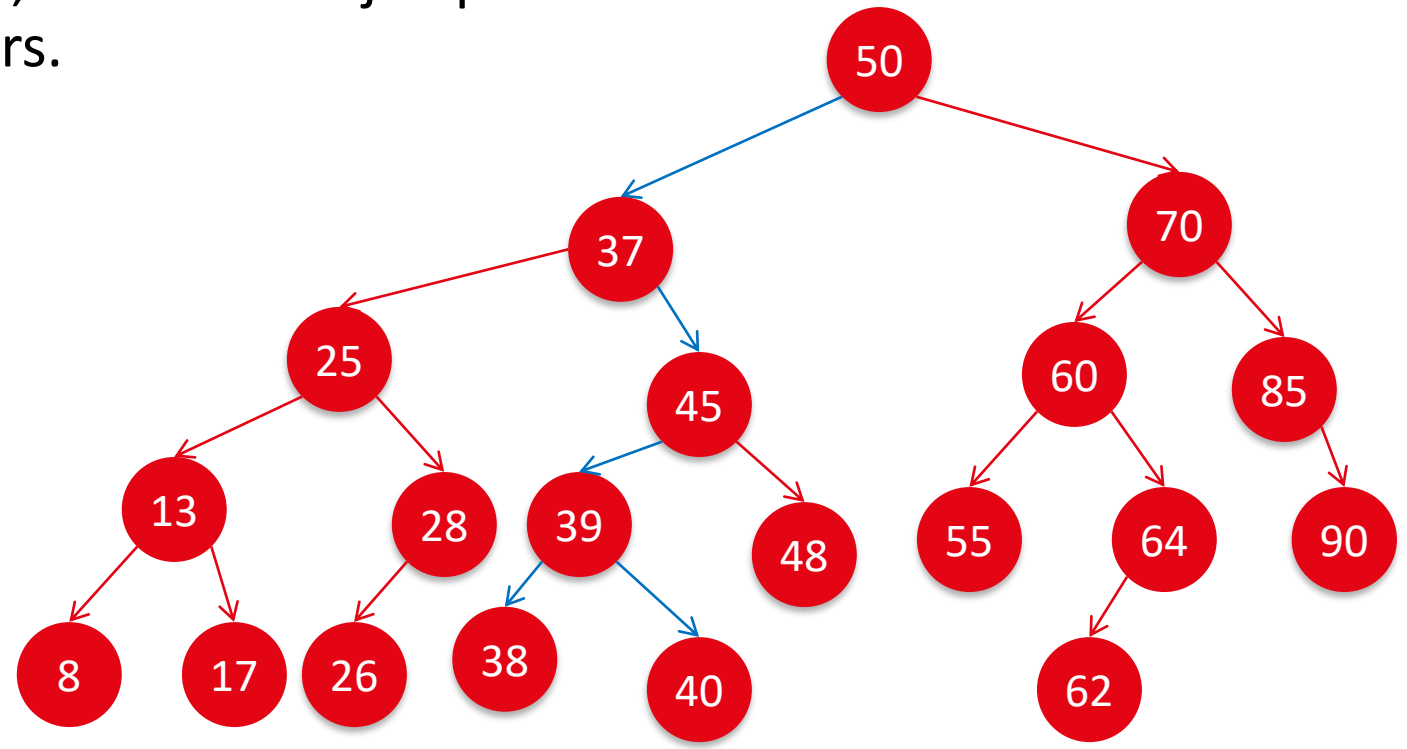
- Il s'agit maintenant d'enlever le noeud 36, on arrive nécessairement à l'un des deux cas simples.



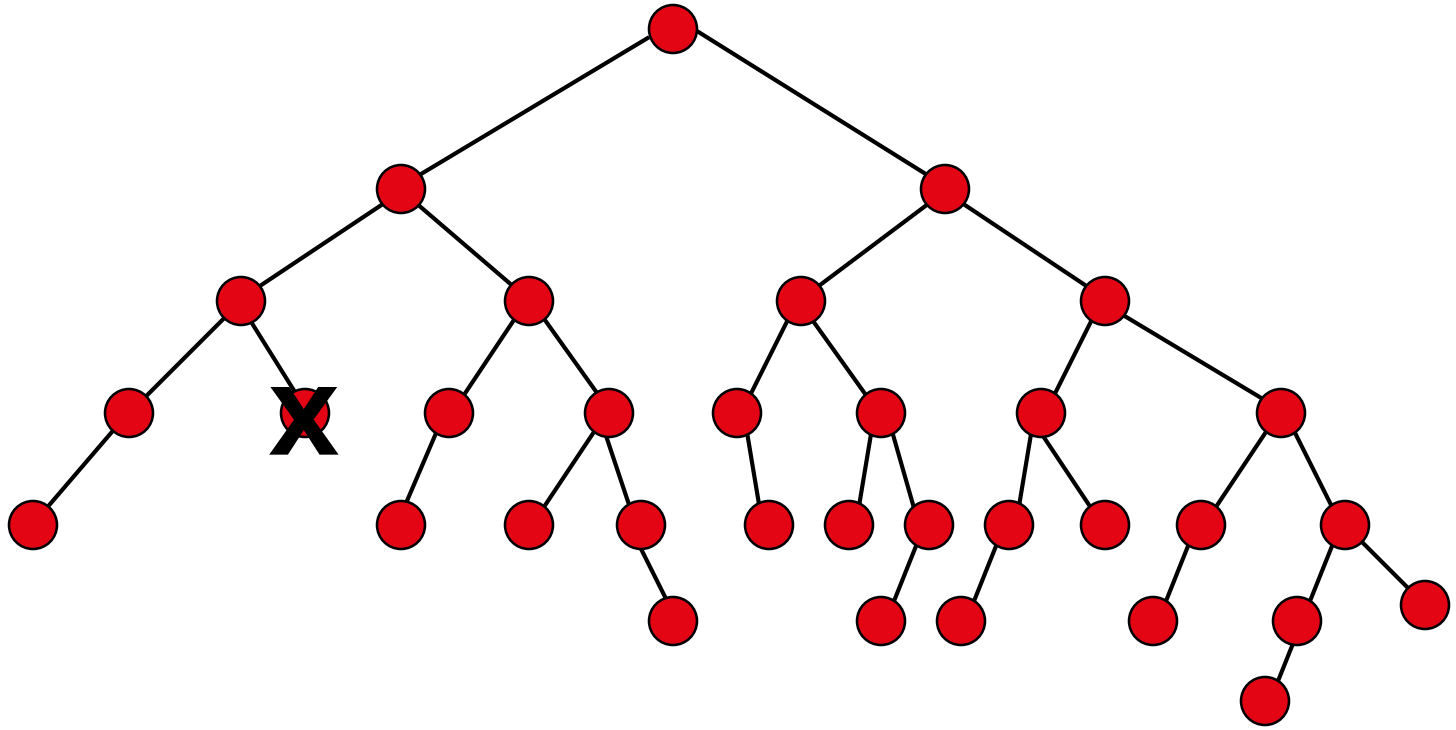
36

Enlèvement dans un arbre AVL

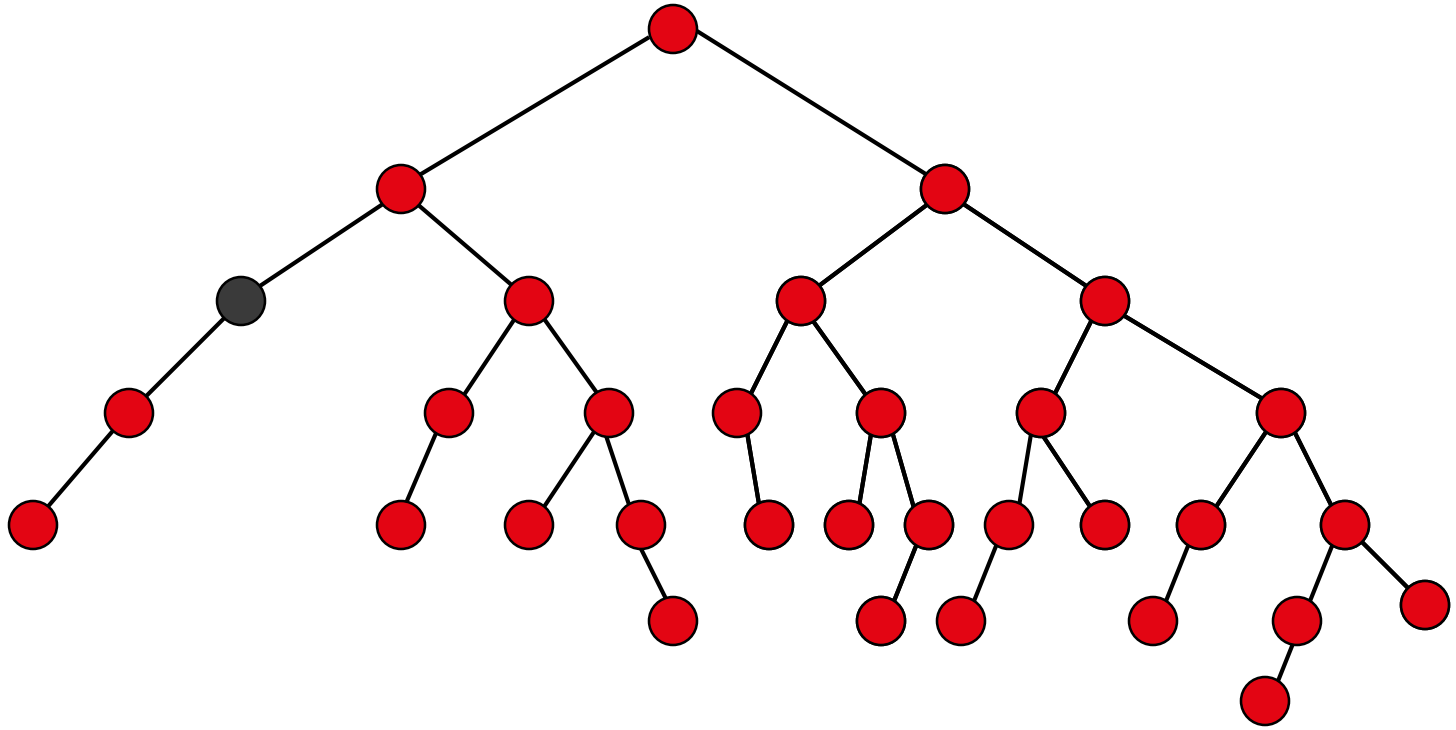
- Puis le nœud 36 est détruit avec « delete ».
- Ensuite, on remonte jusqu'à la racine et l'on rebalance l'arbre au besoin le long du parcours.



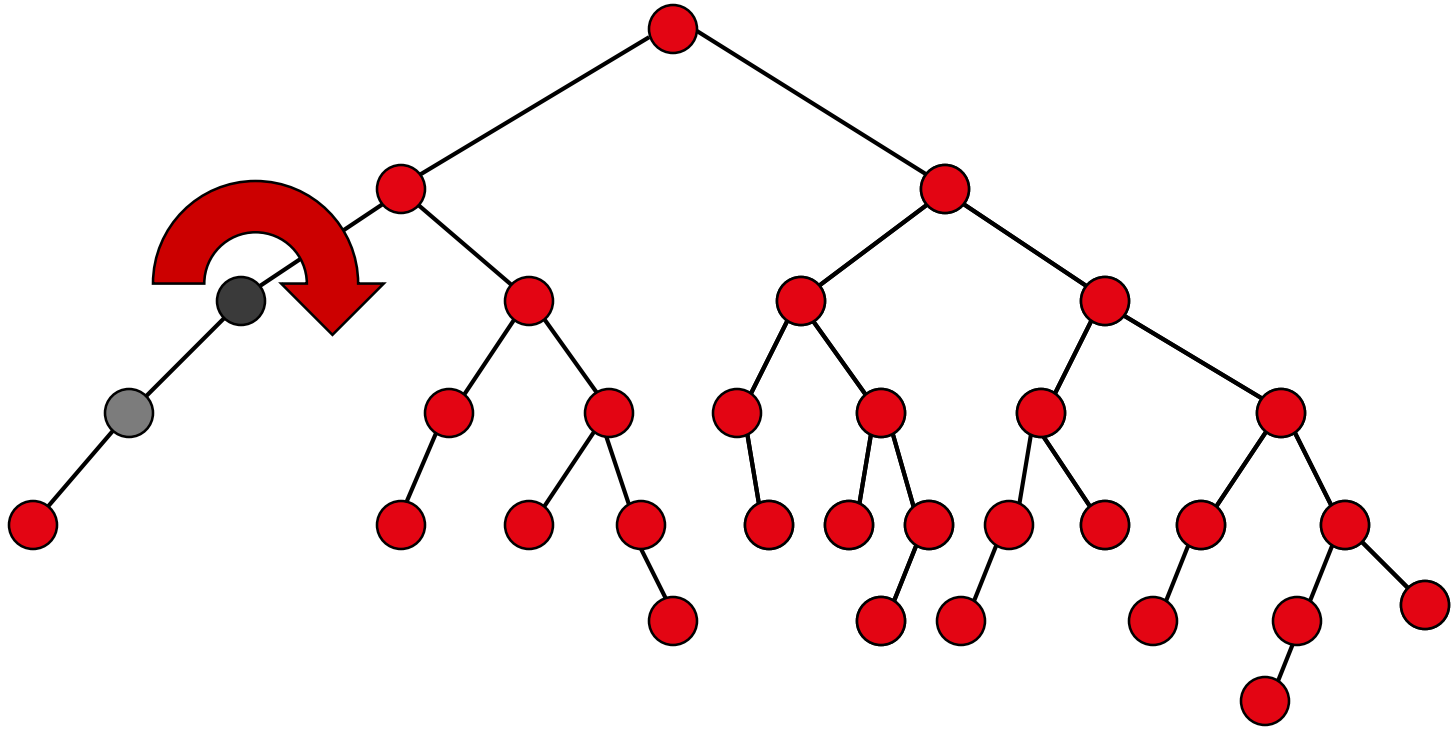
Enlèvement balance : HB[1]



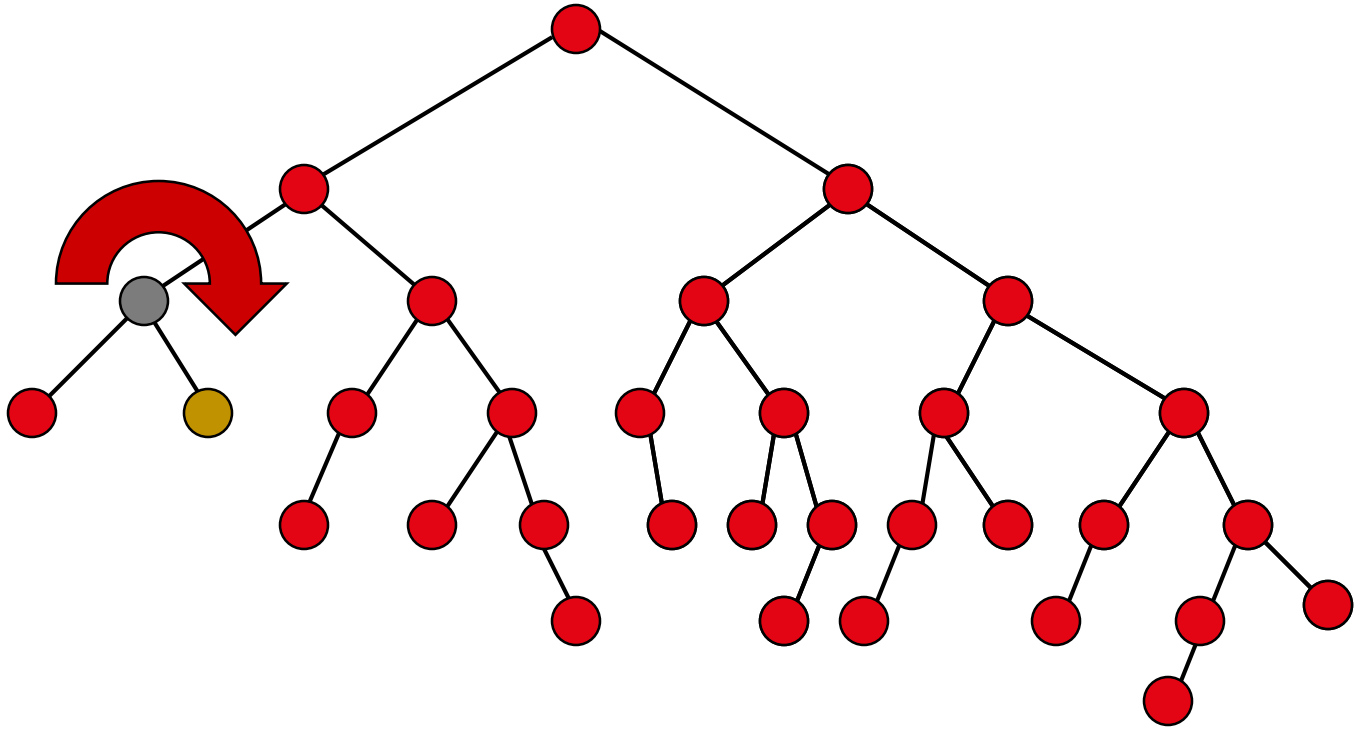
Enlèvement balance : HB[1]



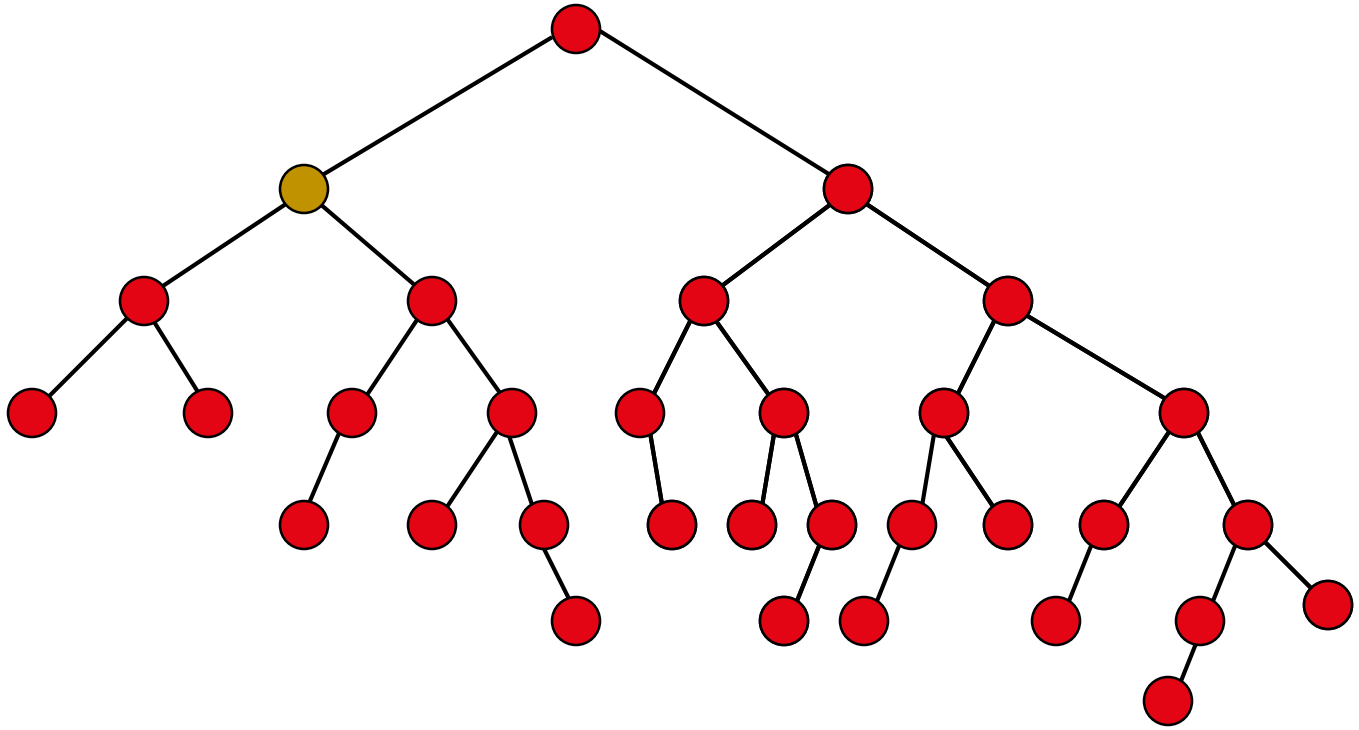
Enlèvement balance : HB[1]



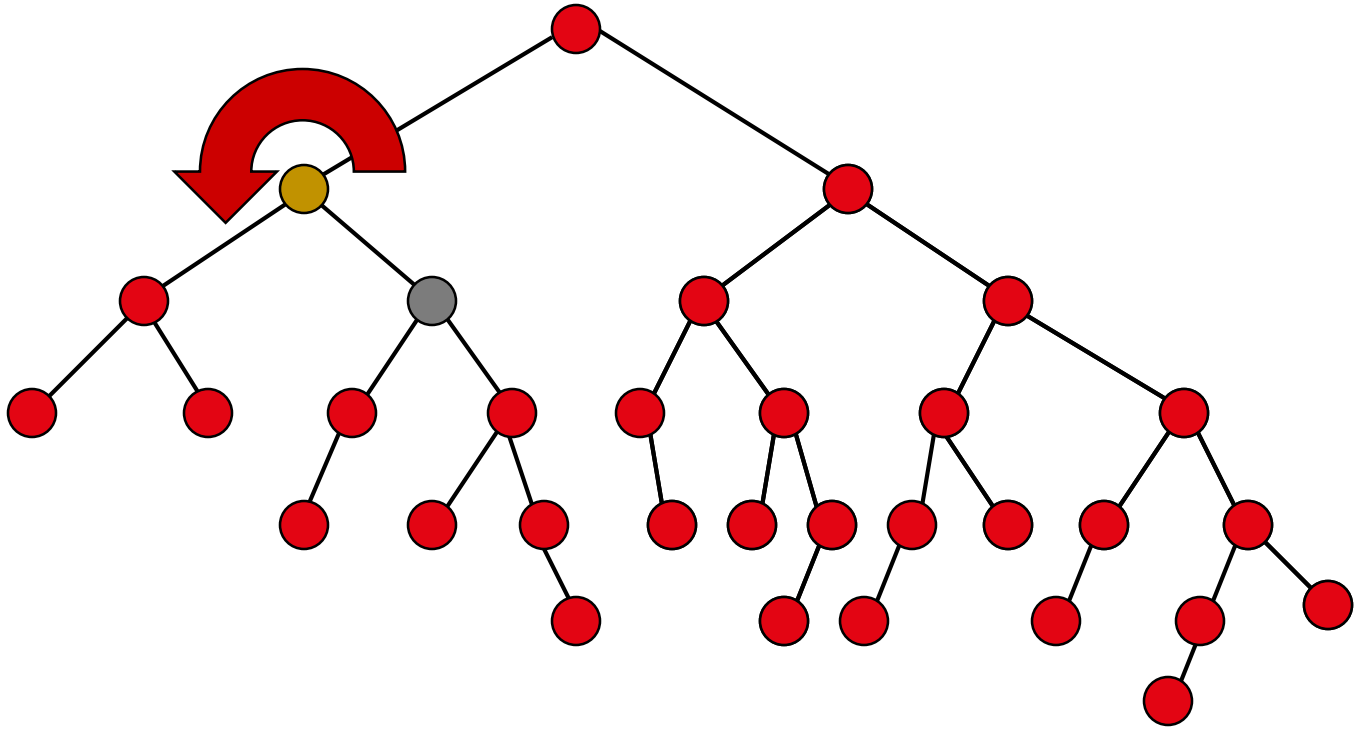
Enlèvement balance : HB[1]



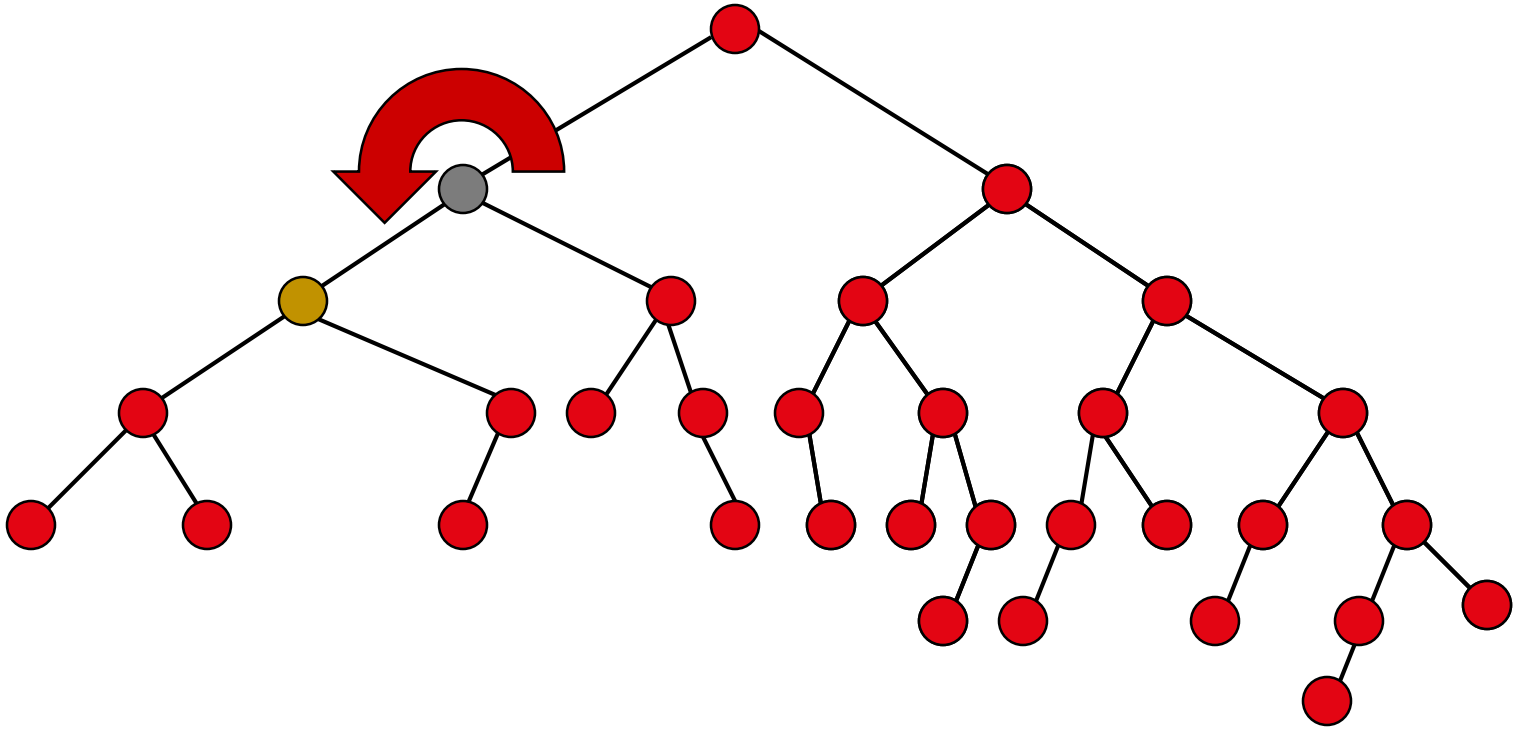
Enlèvement balance : HB[1]



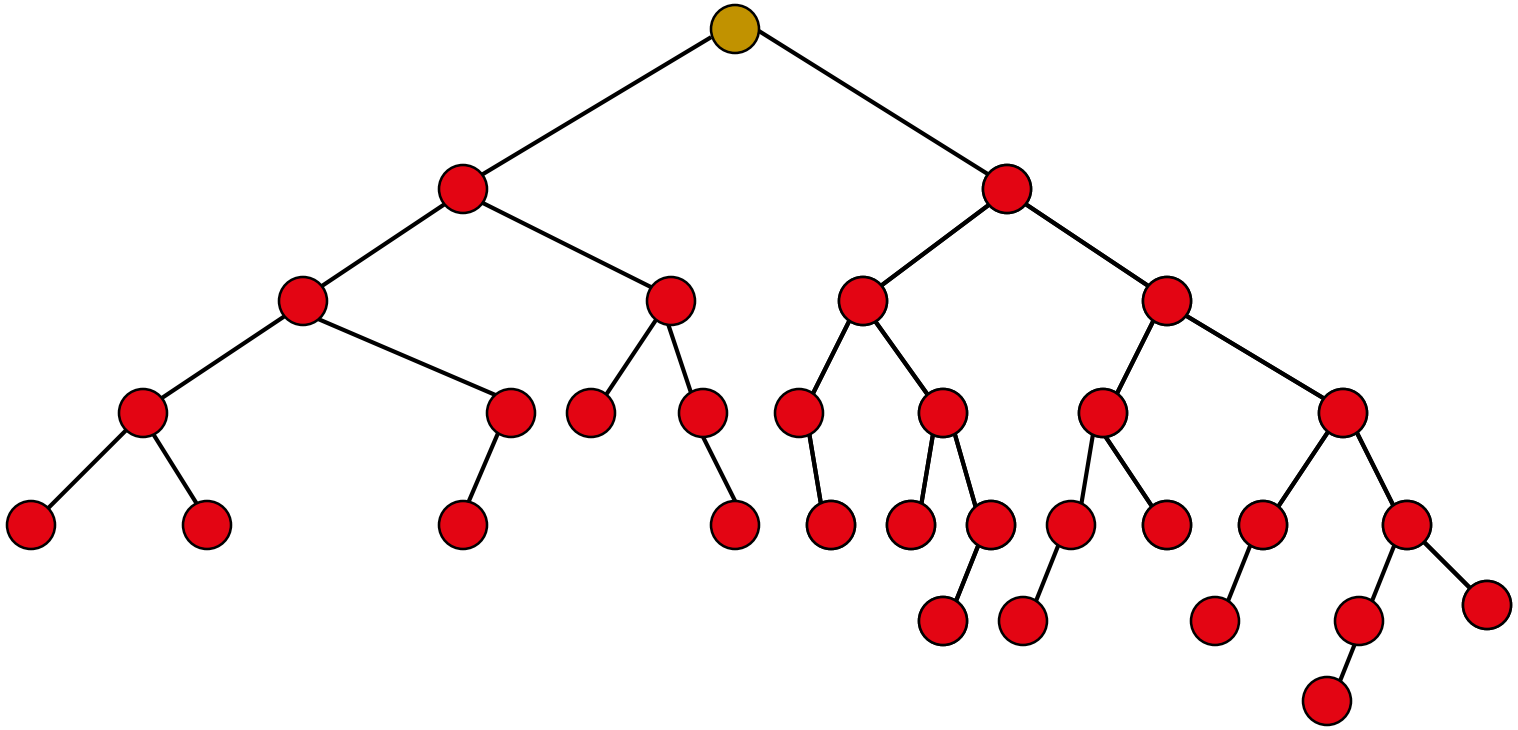
Enlèvement balance : HB[1]



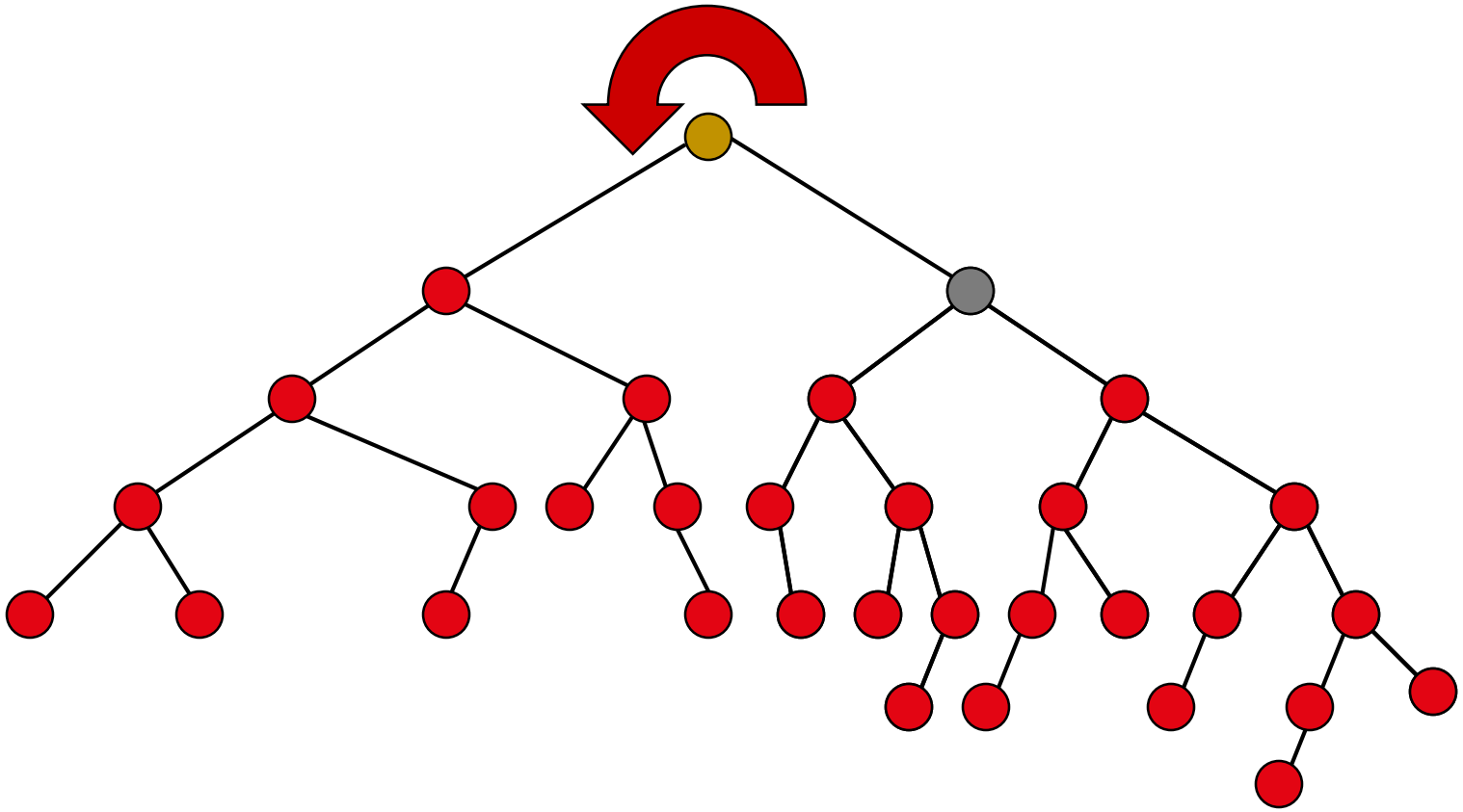
Enlèvement balance : HB[1]



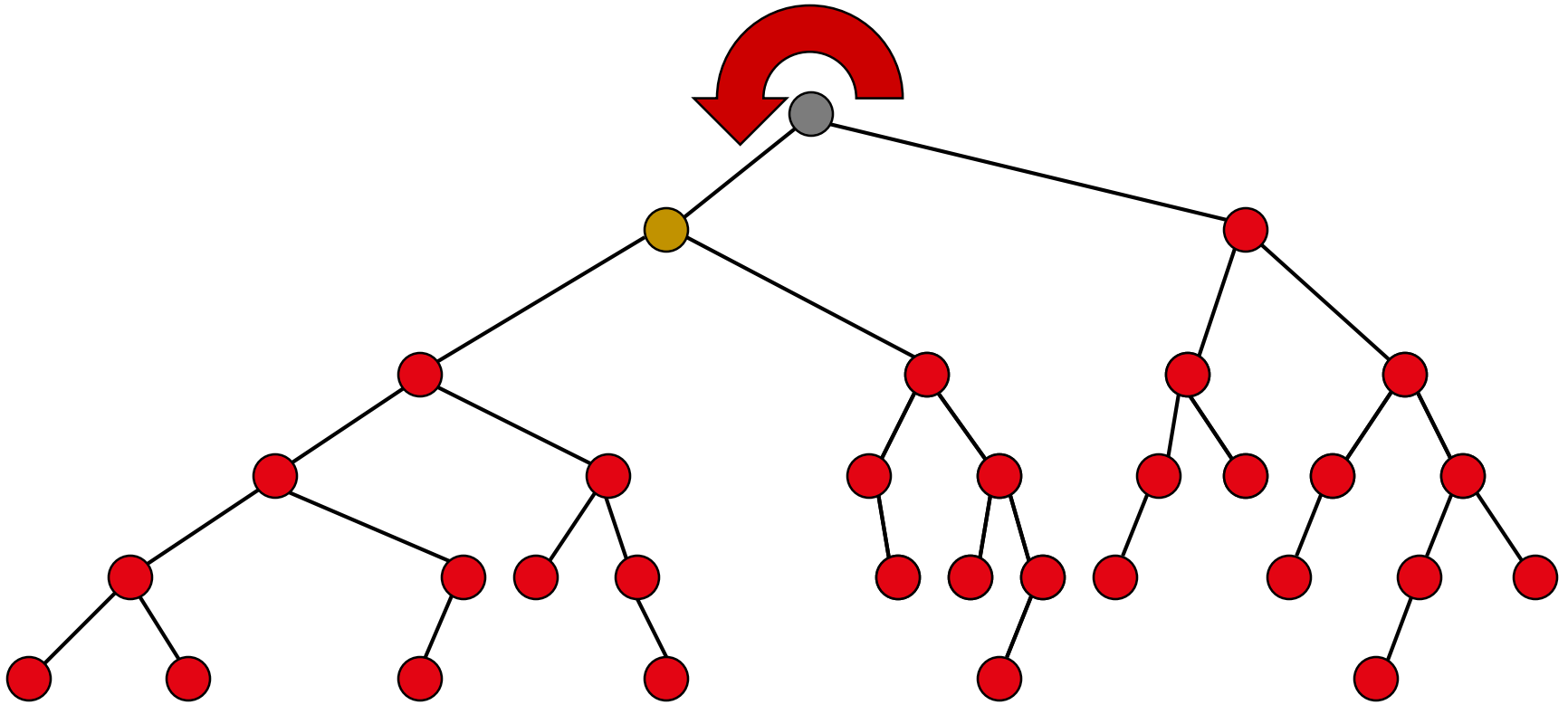
Enlèvement balance : HB[1]



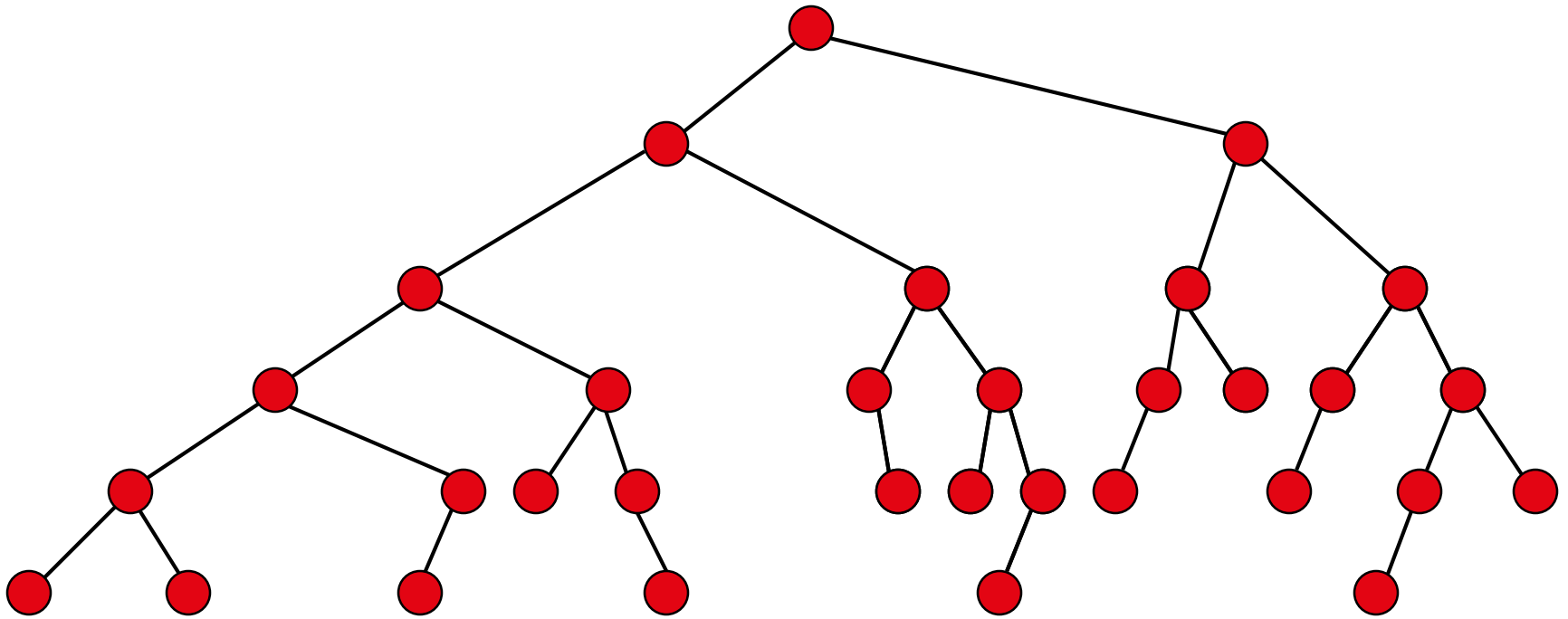
Enlèvement balance : HB[1]



Enlèvement balance : HB[1]



Enlèvement balance : HB[1]



Enlèvement dans un arbre AVL

```
template <typename E>
void Arbre<E>::enleverAVL(const E & data)
{
_auxEnleverAVL(racine, data); //voir page suivante
}
```

```
template <typename E>
void Arbre<E>::_auxEnleverAVL(Noeud*& noeud, const E& valeur)
{
	if (noeud==0) throw logic_error("Tentative d'enlever une donnée absente");
	if (valeur < noeud->data)  _auxEnleverAVL(noeud->gauche, valeur);
	else if(noeud->data < valeur) _auxEnleverAVL(noeud->droite, valeur);
	// valeur == noeud->data: ici on doit enlever le noeud pointé par noeud
	else if(noeud->gauche != 0 && noeud->droite != 0)
	{//Cas complexe: chercher le successeur minimal droit et l'enlever
		_enleverSuccMinDroite(noeud);
	}
	else
	{ //Cas simples: un ou zéro enfant
		Noeud* vieuxNoeud = noeud;
		noeud = (noeud->gauche != 0) ? noeud->gauche : noeud->droite;
		delete vieuxNoeud;
		--nb_noeuds;
	}
	_balancer(noeud); //dans tous les cas: rebalancer et mise à jour des hauteurs
}
```

Enlèvement dans un arbre AVL

```
template <typename E>
void Arbre<E>::_enleverSuccMinDroite(Noeud* noeud)
{
	Noeud* temp = noeud->droite;
	Noeud* parent = noeud;
	while ( temp->gauche != 0)
	{
		parent = temp;
		temp = temp->gauche;
	}
	noeud->data = temp->data; //écrasement par le successeur minimal à droite
	// enlever nœud (cas simple)
	if (temp == parent->gauche)
		_auxEnleverAVL(parent->gauche, temp->data);
	else
		_auxEnleverAVL(parent->droite, temp->data);
}
```

Analyse: complexité de l'enlèvement d'un noeud

- Soit $h(n)$ = hauteur de l'arbre de n nœuds.
- Pour enlever un nœud il faut éventuellement se positionner sur un noeud ayant, au plus, un enfant et l'enlever; ce qui nécessite un temps $O(h(n))$ dans tous les cas.
  - Car la profondeur de la feuille la moins profonde dans un arbre AVL de hauteur h est ≥ $\lceil \boldsymbol{h/2} \rceil$ (voir exercices).
- Chaque re-balancement se fait en $O(1)$ mais il faut le faire en remontant jusqu'à la racine; ce qui se fait donc en $O(h(n))$.
- Conclusion: la suppression dans un arbre AVL (avec re-balancement) se fait en $O(\log n)$ car on a que $h(n)$ est en $O(\log n)$ pour un arbre AVL.

Faut-il toujours re-balancer?

- Cela vaut-il la peine de re-balancer à chaque insertion et suppression comme on le fait pour les arbres AVL?
- Pas nécessaire de le faire à chacune de ces opérations pour conserver $h(n)$ en $O(\log n)$,
  - ➢ changer les algorithmes d'insertions et de suppression.

- Il existe d'autres algorithmes plus complexes (ex: arbres rouge-noir) effectuant moins de rotations lors des insertions et suppressions et qui sont un peu plus performants en pratique, mais ils ont la même croissance asymptotique du temps d'exécution que celui des arbres AVL.

Recherche d’un élément

```
template<typename E>
bool Arbre<E>::appartient(const E& data) const
{
	return _auxAppartient(racine, data) != 0;
}
```

Recherche d'un élément

```
template<typename E>
typename Arbre<E>::Noeud* Arbre<E>::_auxAppartient(Noeud* arbre, const E& data) const
{
	if (arbre == 0)
		return 0; //si data n'est pas dans l'arbre

	if (data < arbre->data)
		return _auxAppartient(arbre->gauche, data);
	else if (arbre->data < data)
		return _auxAppartient(arbre->droite, data);
	else
		return arbre; //car arbre->data == data
}
```

Trouver l'élément maximal

- L'élément maximal se trouve en $O(\log n)$ dans un arbre AVL
- Car la profondeur de la feuille la moins profonde dans un arbre AVL de hauteur h est ≥ $\lceil h/2 \rceil$ (voir exercices).

```
template<typename E>
const E& Arbre<E>::max() const
{
	if(nb_noeuds<=0) throw logic_error("Doit être non vide");
	Noeud* temp = racine;
	while (temp->droite!=0)
		temp = temp->droite;
	return temp->data;
}
```

Synthèse

- Analyse hauteur d'un arbre AVL
  - Insertion dans un arbre AVL en $O(h(n))$
  - $h \in O(\log n)$ pour un arbre AVL de n nœuds
  - Insertion dans un arbre AVL en $O(\log n)$
- Supprimer un nœud dans un arbre AVL
  - Cas simple: le nœud à supprimer est une feuille
  - Cas simple: le nœud à supprimer possède un seul enfant
  - Cas compliqué: le nœud à supprimer possède deux enfants
    - ✓ successeur minimal à droite
  - On remonte jusqu'à la racine et l'on rebalance l'arbre au besoin le long du parcours
  - La suppression dans un arbre AVL (avec re-balancement) se fait en $O(\log n)$
- Cela vaut-il la peine de re-balancer à chaque insertion et suppression?
  - arbres rouge-noir
- Recherche d'un élément en $O(\log n)$
- Trouver l'élément maximal en $O(\log n)$

Arbres binaires de recherche dans la STL

- Utilise les arbres binaires de recherche pour mettre en œuvre les conteneurs set, map, multiset et multimap.
- Ces conteneurs supportent les opérations **insert**, **erase** et **find**.
  - ➢ Le temps est logarithmique en pire cas pour set et map
    - ✓ Pour y arriver, on utilise plus souvent les arbres rouge-noir au lieu des arbres AVL
- Le conteneur set est un ensemble ordonné d'éléments avec la contrainte que chaque élément peut être présent une seule fois
  - ➢ Les duplicatas sont permis dans un multiset
- Le conteneur **map** est un ensemble ordonné des paires (clé,valeur)
  - ➢ Les éléments sont ordonnés selon la clé
  - ➢ Chaque clé doit être unique dans un map
  - ➢ Les duplicatas de clé sont permis dans un multimap

Le conteneur set

- La valeur d'un élément dans un set est son identifiant et doit être unique et non modifiable.
- Les élément sont ordonnés selon operator< défini par le type T des éléments du set
  - Il est possible d'utiliser un foncteur lors de la création du set pour définir l'opérateur de comparaison
- On utilise **iterator** pour accéder aux élements
  - la méthode **begin** () retourne un itérateur pointant sur le premier (i.e. le plus petit) élément et **end**() retourne un iterator pointant sur un élément (sentinelle) après le dernier

Le conteneur set

- On utilise **insert(const T&)** pour ajouter un élément et ça retourne un objet de type **pair<iterator,bool>** où le membre bool nous indique si l'insertion a échoué (élément déjà dans le set) ou a été effectué avec succès (dans ce cas iterator pointe sur l'élément inséré).
- On utilise **erase(const T&)** ou **erase(iterator)** pour enlever un élément du set.
- On utilise **find(const T&)** pour obtenir un iterator pointant sur l'élément recherché s'il est trouvé ou pointant sur la sentinelle à la fin du set si cet élément n'est pas dans le set.

Le conteneur set : exemple

```
#include <iostream>
#include <set>
int main()
{
	std::set<int> myset;
	std::set<int>::iterator it;
	myset.insert(30);
	myset.insert(10);
	myset.insert(20);
	myset.insert(5);
	myset.insert(15);
	it = myset.begin();
	myset.erase(it); //enlève le plus petit élément
	it = myset.end();
	myset.erase(--it); //enlève le plus grand
	std::cout << "myset contains:";
	for (it = myset.begin(); it != myset.end(); ++it)
		std::cout << ' ' << *it; //affiche 10 15 20
	std::cout << std::endl;
	return 0;
}
```

Le conteneur map

- Est un ensemble ordonné de paires de type **pair<KeyType,ValueType>**
- Les élément sont ordonnés selon **operator<** défini par le type **KeyType**
  - Il est possible d'utiliser un foncteur lors de la création du map pour définir l'opérateur de comparaison
- Pour accéder aux éléments, on peut utiliser un **iterator** ou, plus simplement, l'opérateur d'indexation [ ] dont le prototype est:
  - **ValueType & operator[] (const KeyType & key)**

Le conteneur map

- Fonctionnement: Si **key** est dans le map alors une référence à la valeur associé à **key** est retournée. Si **key** n'est pas dans le map, **key** est inséré dans la map avec une valeur associé par défaut (défini par le constructeur sans paramètre de **ValueType**).
  - **operator[]** est un modificateur: il est donc inutilisable dans une fonction où le **map** est passé par référence constante.
- Il est donc généralement plus prudent de tester d'abord si une clé est dans le **map** avant de lui associer une valeur. Pour cela on **utilise find(const KeyType & key)** qui retourne un itérateur pointant sur la **paire pair<key,value>** si key est dans le map (et value est la valeur associé à key). Sinon, l'itérateur retourné pointe sur un élément de map passé le dernier.

Le conteneur map (exemple)

```
std::map<string, unsigned int> tailles;
tailles["Julie"] = 162; //insère éventuellement une paire <''Julie'',162> dans tailles
std::cout << tailles["Julie"] << endl; //affiche 162
std::cout << tailles["Robert"] << endl; //insère une paire <''Robert'',0>
                                        //dans tailles et affiche 0
```

- Le deuxième énoncé: tailles[''Julie''] insère d'abord <''Julie'',0> dans tailles et retourne par référence la valeur associée de 0 . Cette valeur est ensuite écrasée par 162 avec l'opérateur d'affectation =.
- Le dernier énoncé insère <''Robert'',0> dans tailles et affiche 0

Le conteneur map (exemple)

```
std::map<std::string, unsigned int>::const_iterator itr;
itr = tailles.find("Robert");
if(itr==tailles.end())
	std::cout << "Absent de notre base de donnée" << endl;
else
	std::cout << itr->second << endl; //affiche la valeur associée à la clé Robert
```

- À la place de ce dernier énoncé, on peut d'abord vérifier si ''Robert'' est dans le map et, si c'est le cas, afficher sa taille:

Synthèse

- Dans la STL
  - set, map, multiset et multimap
    - ✓ insert erase find Iterator begin end
    - ✓ Clés non modifiables
    - ✓ Arbres rouge et noir
- set (duplicatas interdits), multiset (duplicatas autorisés)
  - operator< défini par le type T (data)
- map (duplicatas interdits), multimap (duplicatas autorisés)
  - pair<KeyType,ValueType>
  - operator< défini par le type KeyType
  - operator[]