

## 5.2. Arbres binaires de recherche



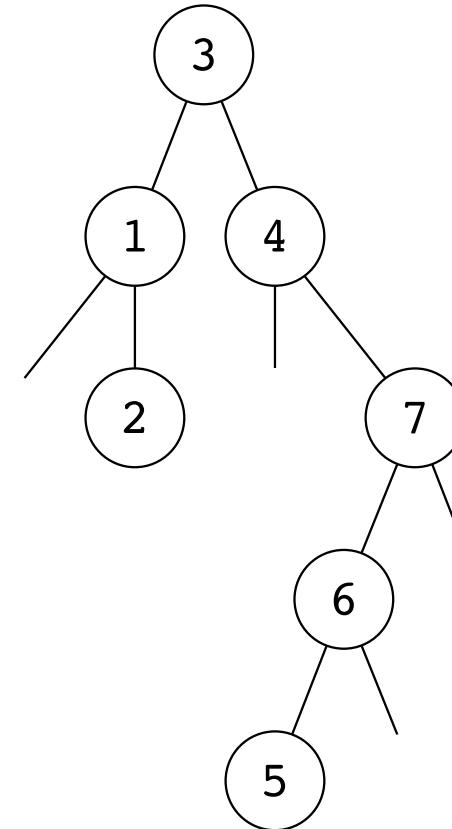
# 9. Structure et parcours





# Définition

- Arbre binaire contenant des clés comparables

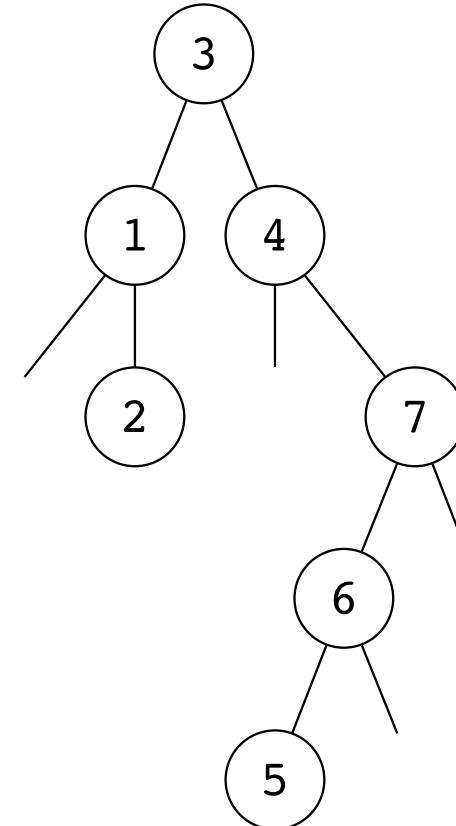




# Définition

- Arbre binaire contenant des clés comparables
- Pour tout noeud **g** du sous-arbre gauche de **r** et tout noeud **d** du sous-arbre droit de **r**,

**g.clé < r.clé < d.clé**

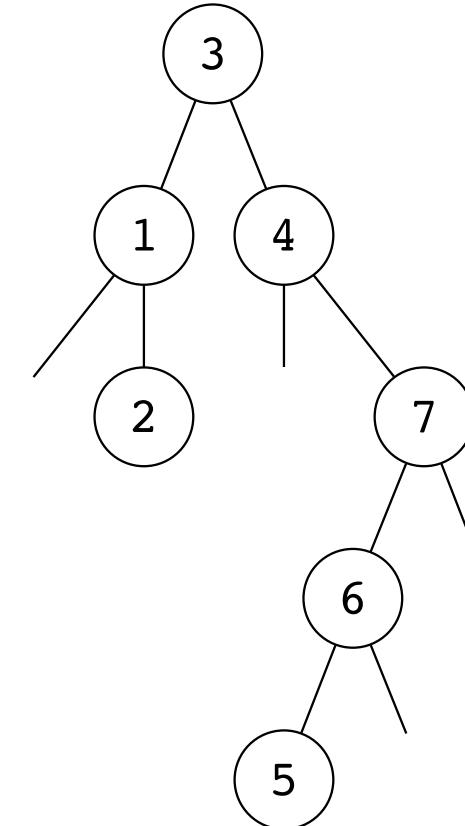




# Définition

- Arbre binaire contenant des clés comparables
- Pour tout noeud **g** du sous-arbre gauche de **r** et tout noeud **d** du sous-arbre droit de **r**,

$$\mathbf{g.clé} < \mathbf{r.clé} < \mathbf{d.clé}$$

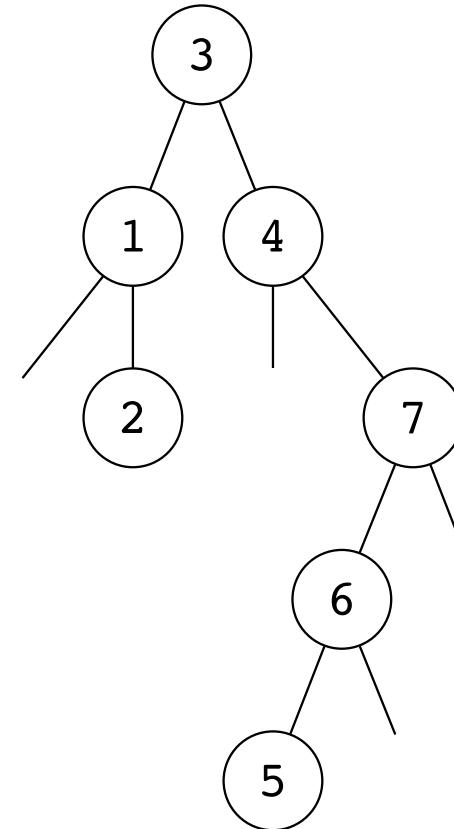


- Les clés sont uniques



# Utilité

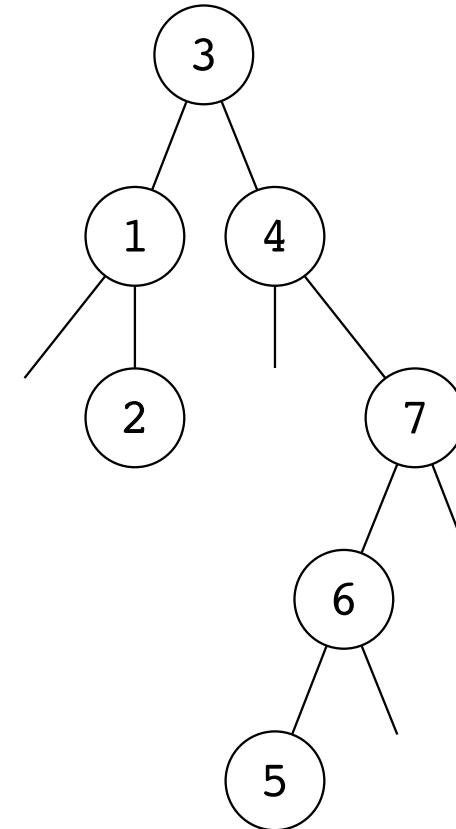
- Pour **chercher** une clé, on ne doit parcourir que son chemin, pas tout l'arbre





# Utilité

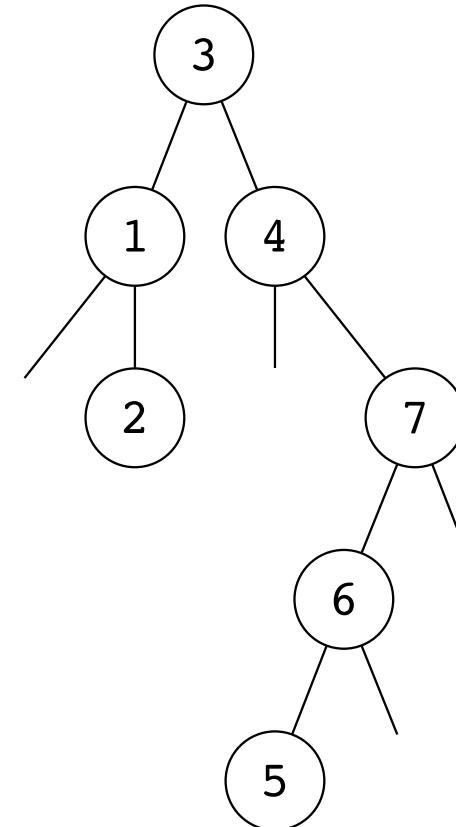
- Pour **chercher** une clé, on ne doit parcourir que son chemin, pas tout l'arbre
- Si l'arbre n'est pas trop dégénéré, la complexité sera  **$O(\log(n))$**  pour **n** clés





# Utilité

- Pour **chercher** une clé, on ne doit parcourir que son chemin, pas tout l'arbre
- Si l'arbre n'est pas trop dégénéré, la complexité sera  **$O(\log(n))$**  pour **n** clés
- **Insérer / supprimer** une clé ont des complexités similaires





# Représentation

- Au minimum, on stocke la **clé** et les liens vers les **enfants** gauche et droit
- La **valeur** est la partie des données qui n'est pas utilisée comme clé, i.e. pas comparée et pas unique
- Pour pouvoir itérer, il faut inclure un lien vers le noeud **parent**
- Stocker taille, hauteur, équilibre, couleur, ... permet d'offrir des fonctionnalités supplémentaires et de garantir des complexités logarithmiques

```
structure Noeud<K, V>
```

```
// obligatoires
```

```
K clé
```

```
Noeud* gauche
```

```
Noeud* droit
```



# Représentation

- Au minimum, on stocke la **clé** et les liens vers les **enfants** gauche et droit
- La **valeur** est la partie des données qui n'est pas utilisée comme clé, i.e. pas comparée et pas unique
- Pour pouvoir itérer, il faut inclure un lien vers le noeud **parent**
- Stocker taille, hauteur, équilibre, couleur, ... permet d'offrir des fonctionnalités supplémentaires et de garantir des complexités logarithmiques

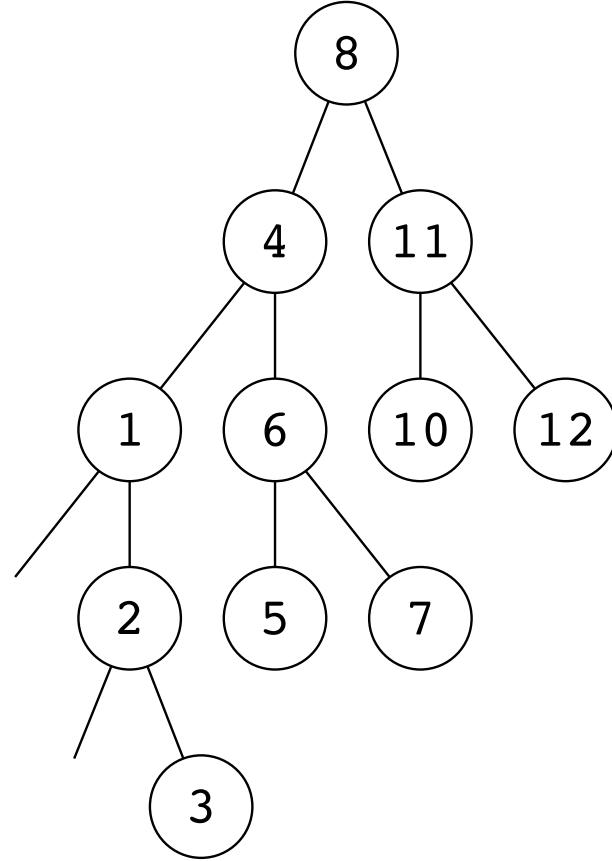
```
structure Noeud<K, V>
    // obligatoires
    K clé
    Noeud* gauche
    Noeud* droit

    // optionnels
    V valeur
    Noeud* parent
    Entier64 taille
    Entier8 hauteur
    Entier2 équilibre
    Booléen couleur
```



# Recherche

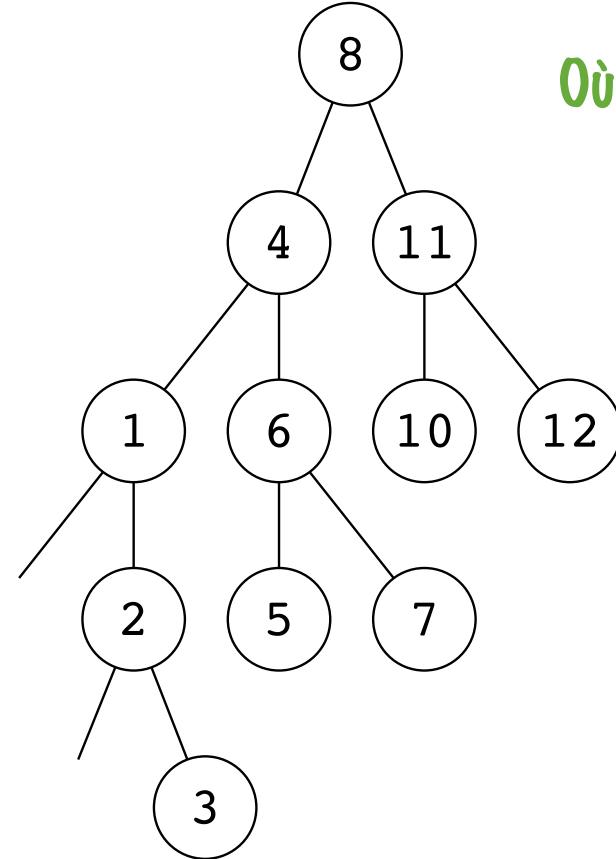
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

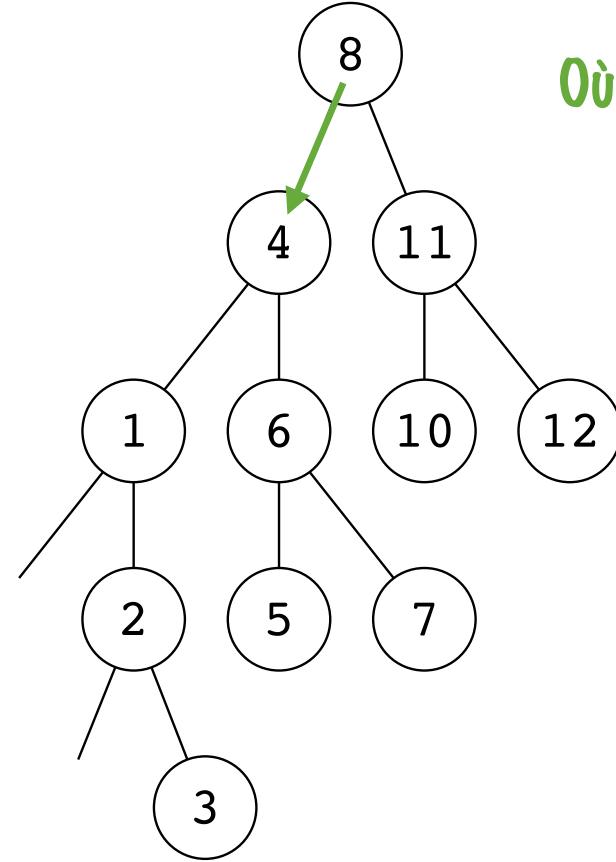
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

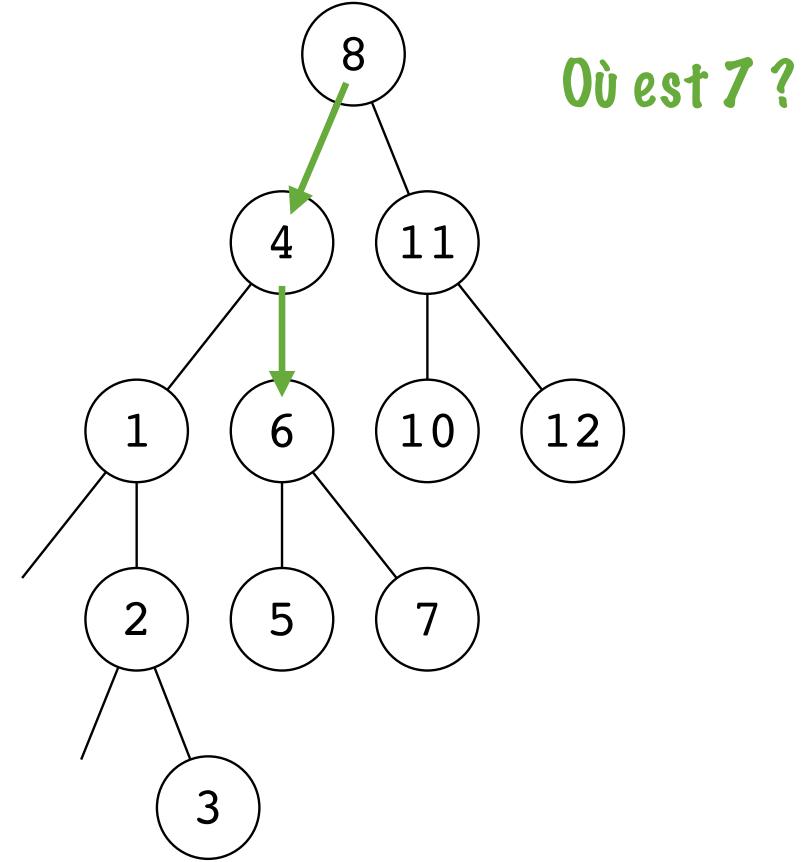
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

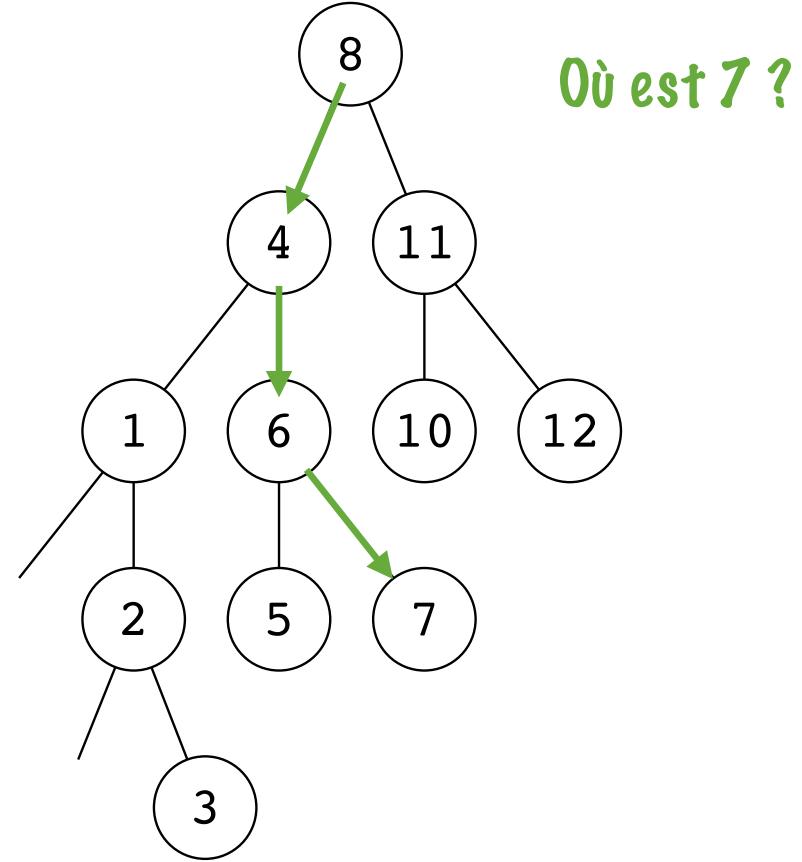
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

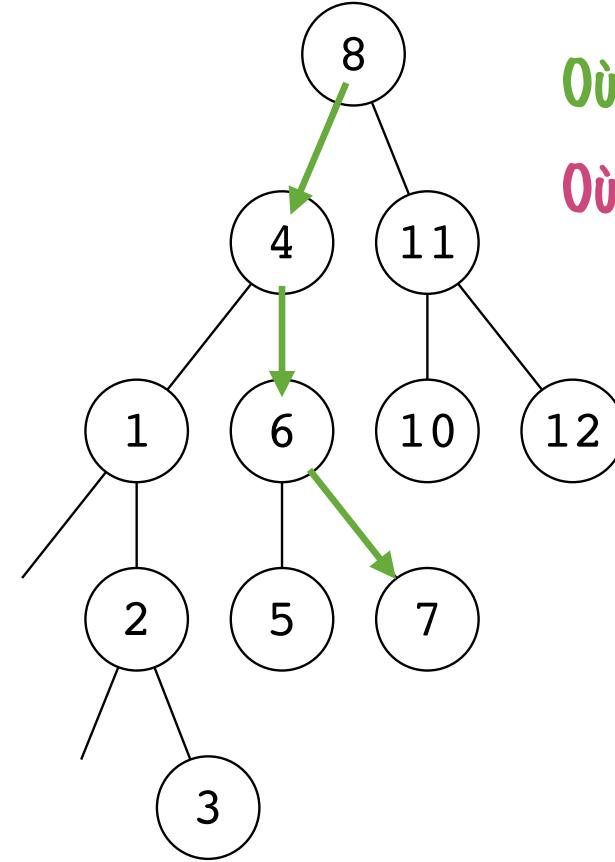
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente



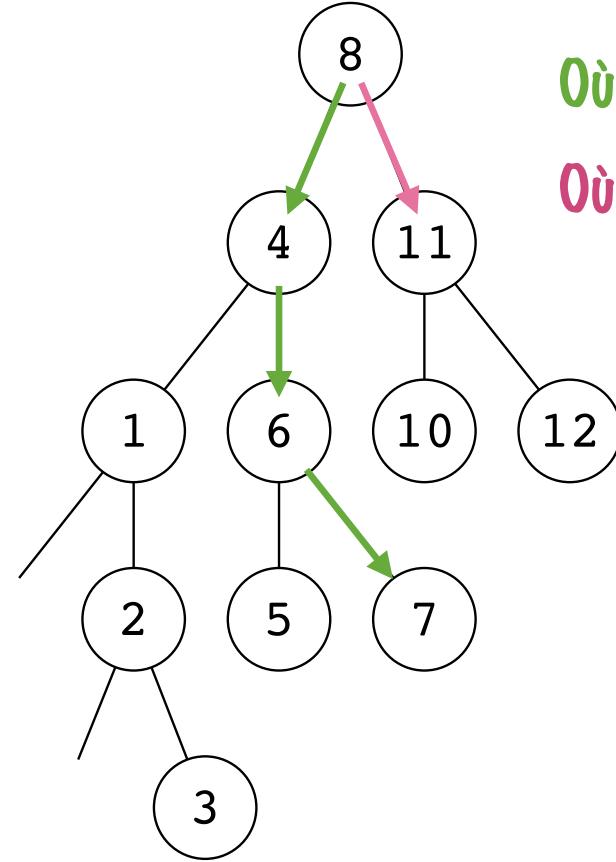
Où est 7 ?

Où est 9 ?



# Recherche

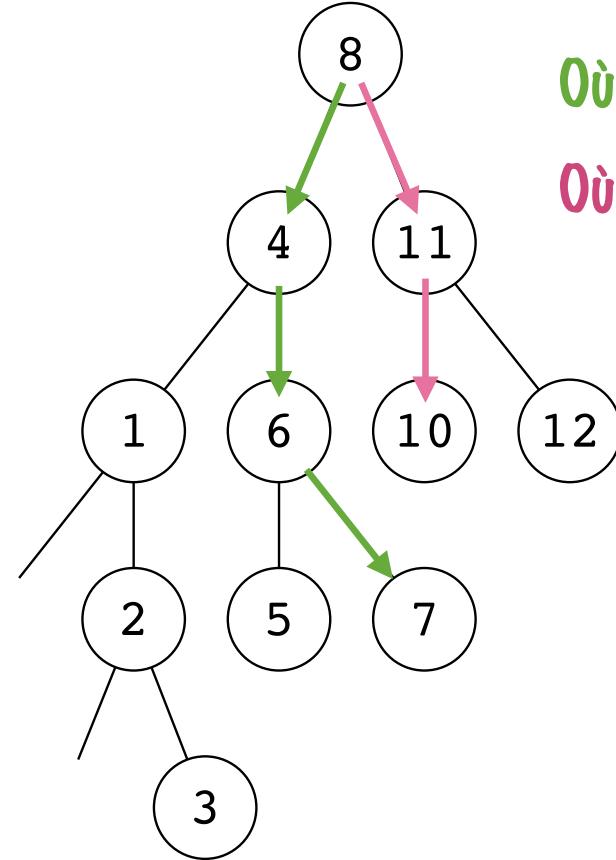
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente



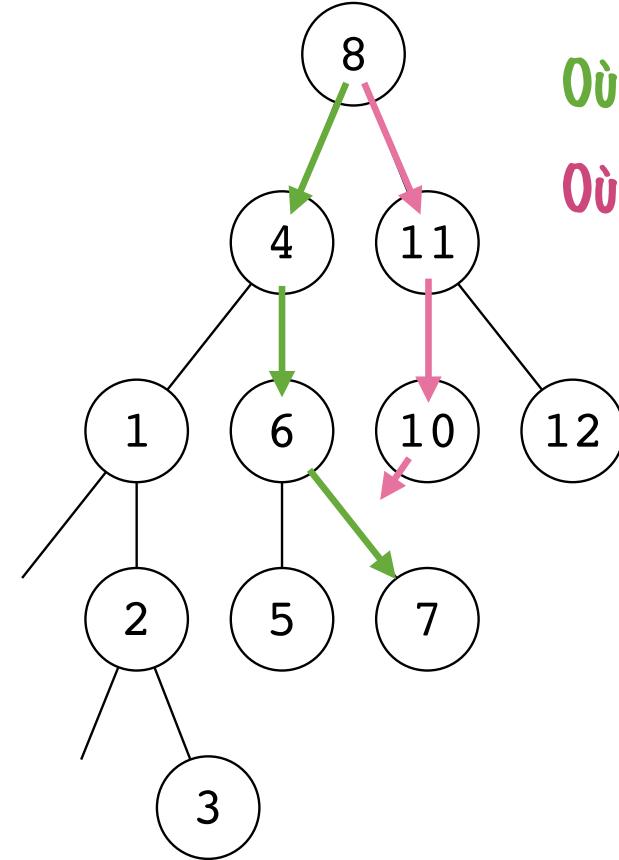
Où est 7 ?

Où est 9 ?



# Recherche

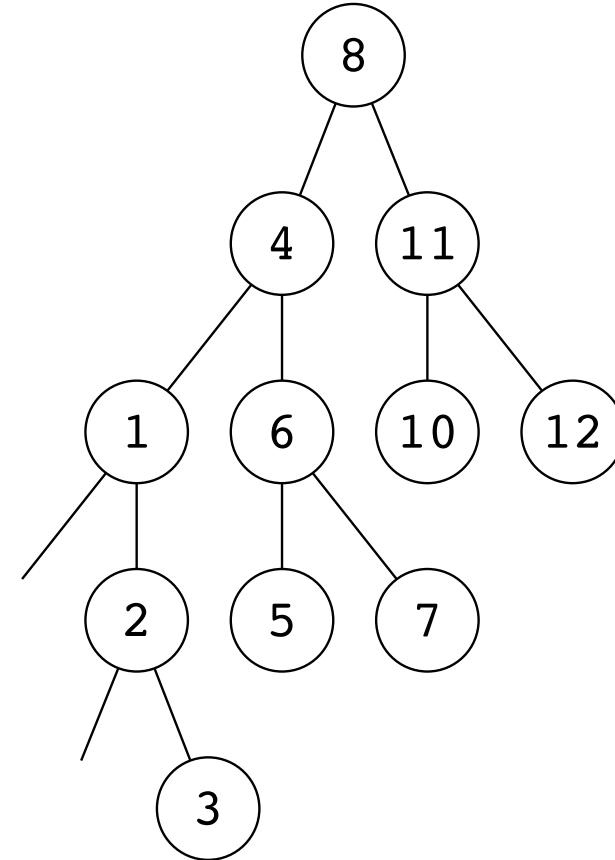
- Comparer la clé cherchée à la racine
- Si égale, on a trouvé
- Sinon, on sait de quel côté chercher
- Si on arrive à un noeud vide, la clé est absente





# Recherche

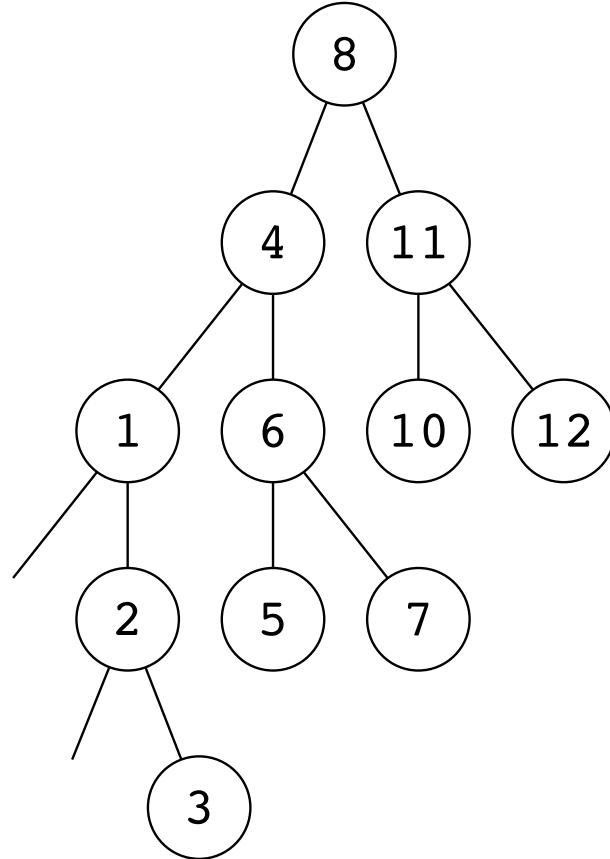
```
fonction chercher (r, k)
    si r est ø
        k n'est pas dans l'arbre
    sinon, si k == r.clé
        r est le noeud cherché
    sinon, si k < r.clé
        chercher(r.gauche, k)
    sinon // k > r.clé
        chercher(r.droit, k)
```





# Insertion

- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...

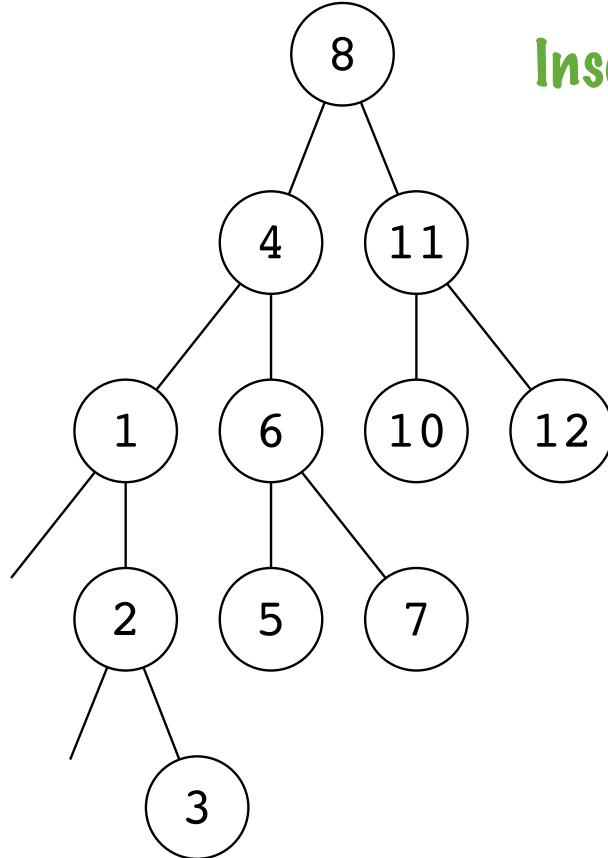




# Insertion

Insérer 5.5

- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...

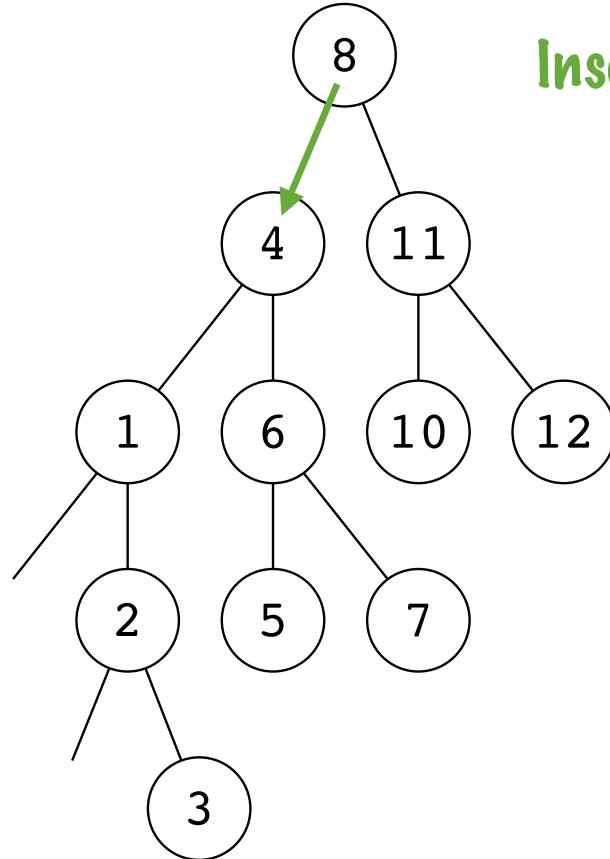




# Insertion

Insérer 5.5

- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...

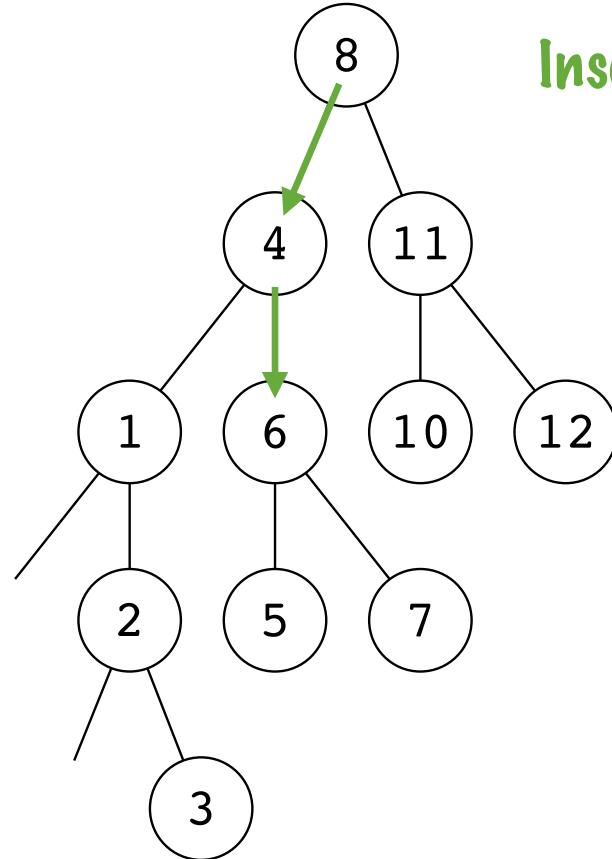




# Insertion

Insérer 5.5

- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...

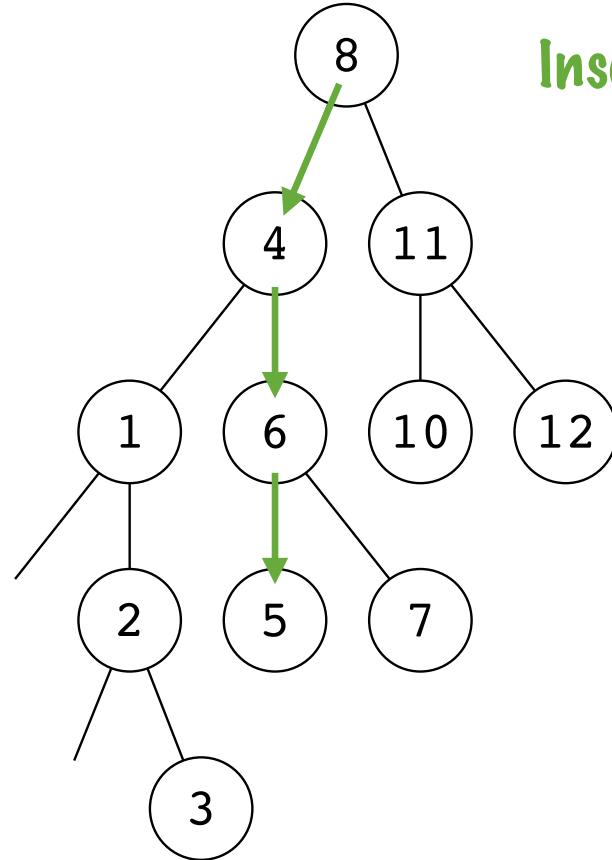




# Insertion

Insérer 5.5

- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...

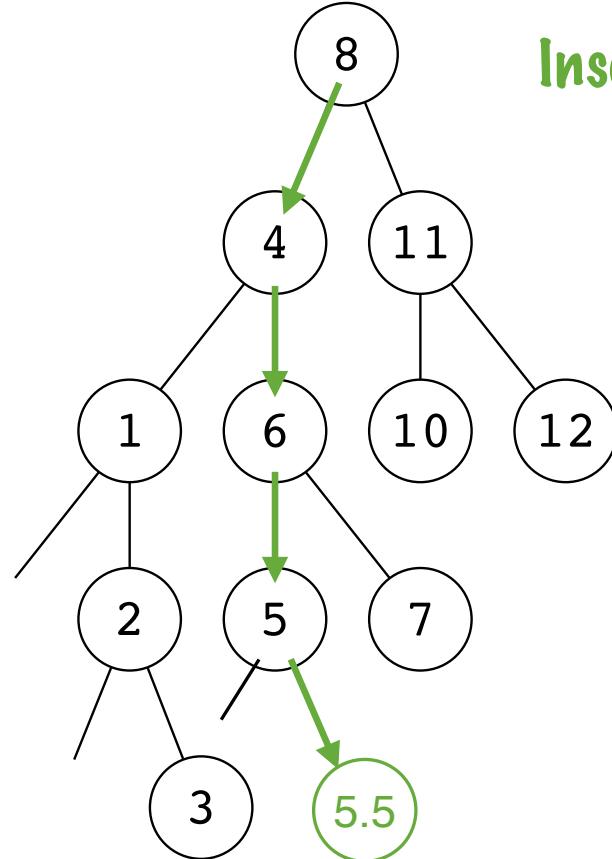




# Insertion

Insérer 5.5

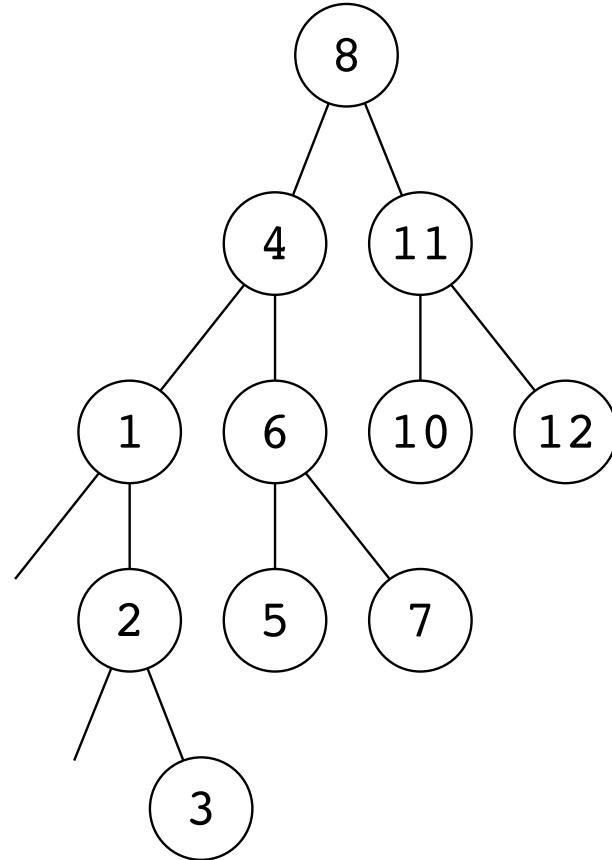
- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...





# Insertion

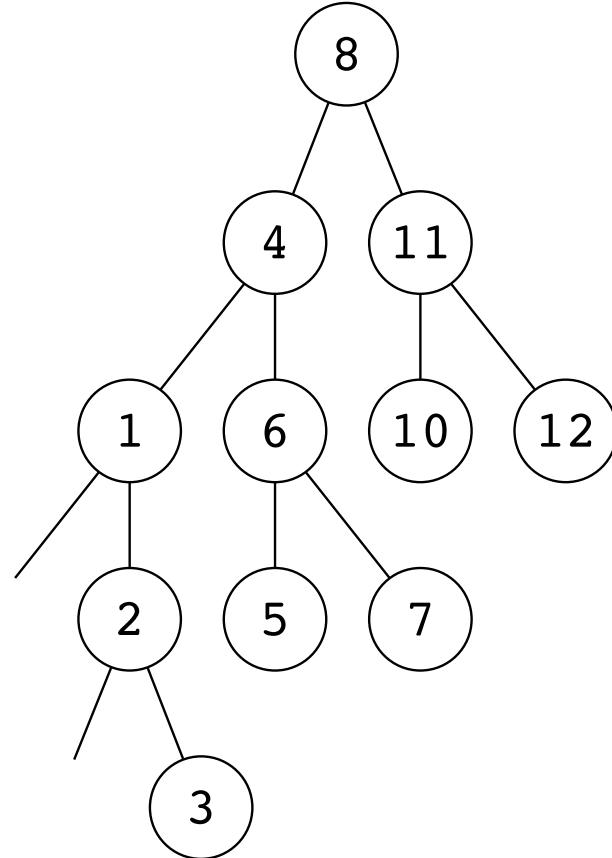
```
fonction insérer (ref r, k)
    si r est Ø
        r ← nouveau noeud de clé k
    sinon, si k == r.clé
        k est déjà présent. L'action
        dépend du TDA mis en oeuvre
    sinon, si k < r.clé
        insérer(r.gauche, k)
    sinon // k > r.clé
        insérer(r.droite, k)
```





# Insertion

```
fonction insérer (ref r, k)
    si r est ø
        r ← nouveau noeud de clé k
    sinon, si k == r.clé
        k est déjà présent. L'action
        dépend du TDA mis en oeuvre
    sinon, si k < r.clé
        insérer(r.gauche, k)
    sinon // k > r.clé
        insérer(r.droite, k)
```

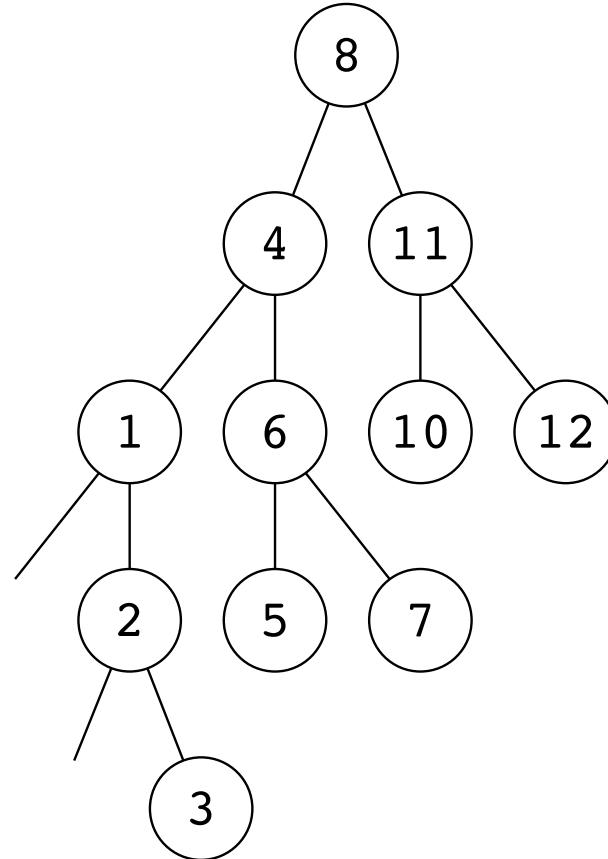


Notons que la racine est passée par référence



# Parcours ordonné

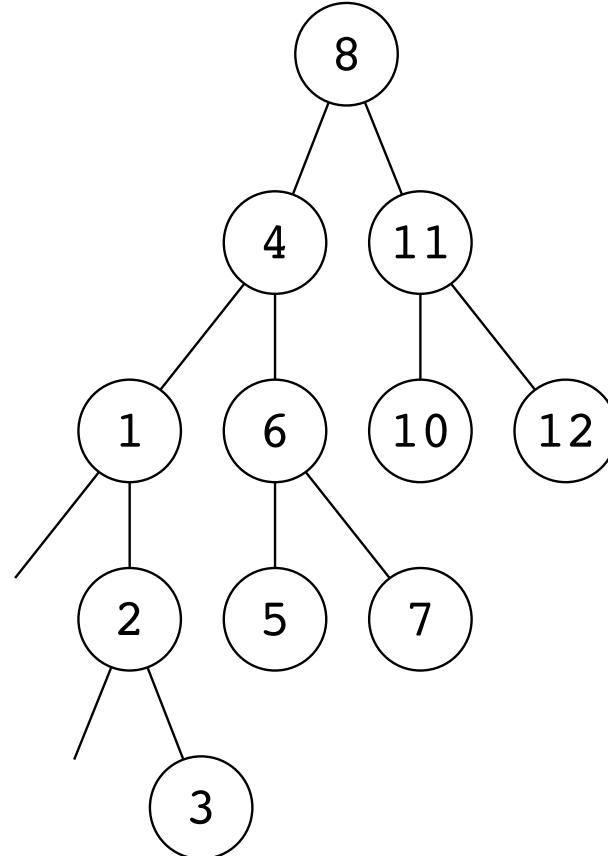
- Parcours par clé croissante





# Parcours ordonné

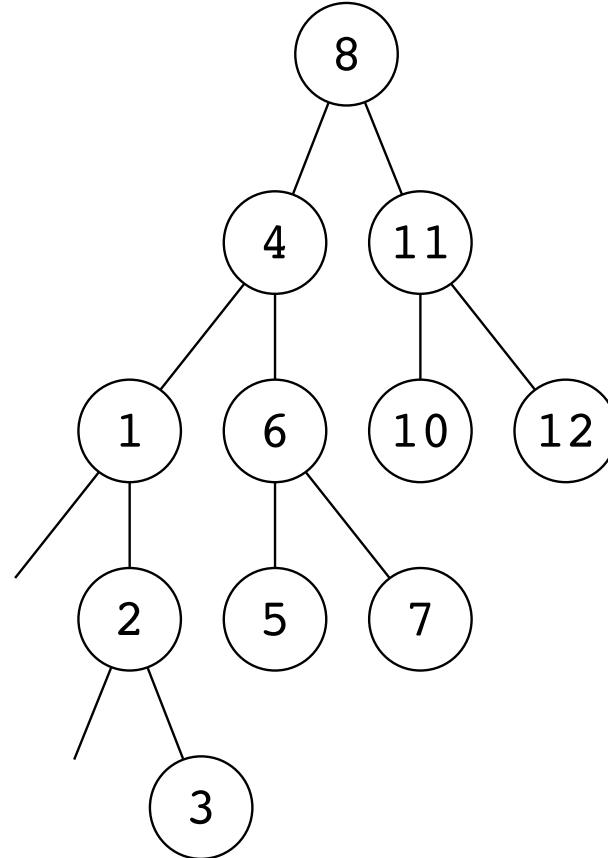
- Parcours par clé croissante
  - Sous-arbre gauche avant la racine
  - Sous-arbre droit après la racine





# Parcours ordonné

- Parcours par clé croissante
  - Sous-arbre gauche avant la racine
  - Sous-arbre droit après la racine
- Parcours symétrique

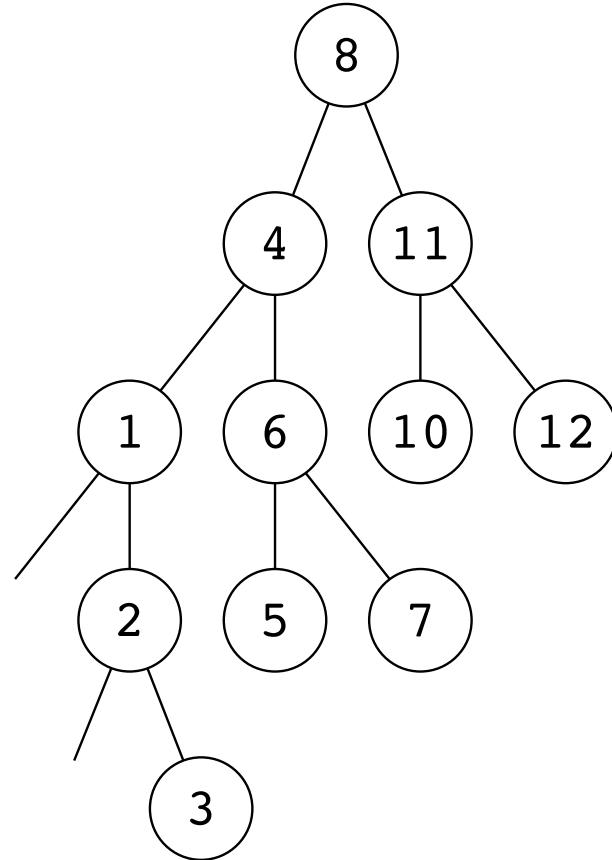




# Parcours ordonnés

```
fonction croissant (r, fn)
    si r != Ø
        croissant(r.gauche, fn)
        fn(r)
        croissant(r.droit, fn)
```

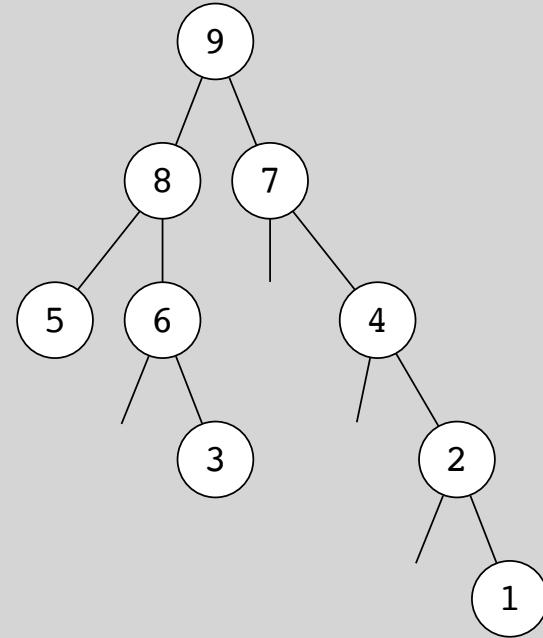
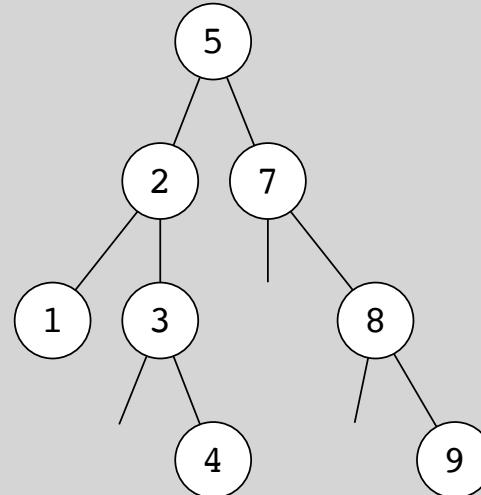
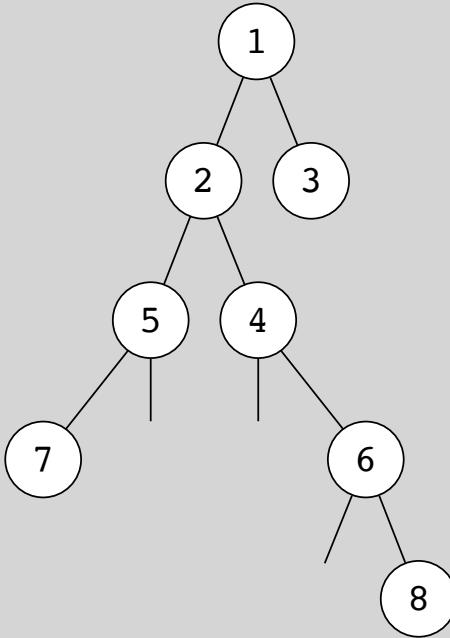
```
fonction décroissant (r, fn)
    si r != Ø
        décroissant(r.droit, fn)
        fn(r)
        décroissant(r.gauche, fn)
```





# Exercice 1

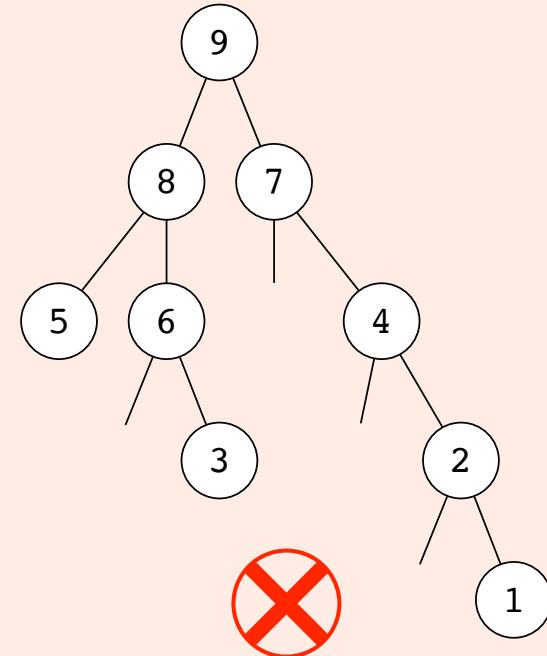
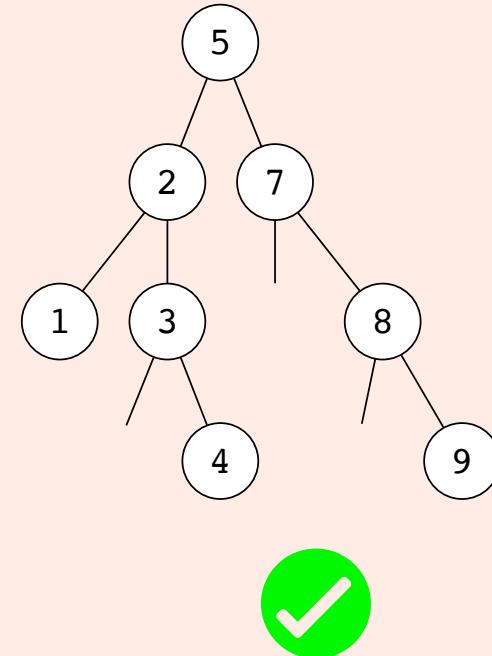
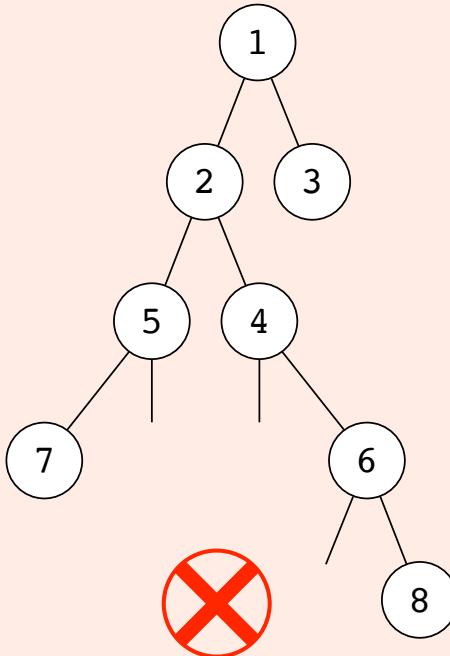
- Parmi les arbres suivants, le(s)quel(s) respecte(nt) la condition d'arbre binaire de recherche ?





# Solution 1

- Condition ABR : la clé de toute racine est plus grande que les clés à gauche et plus petite que les clés à droite.





# Exercice 2

- En partant d'un ABR vide, on insère les valeurs suivantes dans cet ordre

6, 1, 3, 5, 7, 9, 8, 4, 2

- Dessinez l'arbre résultant

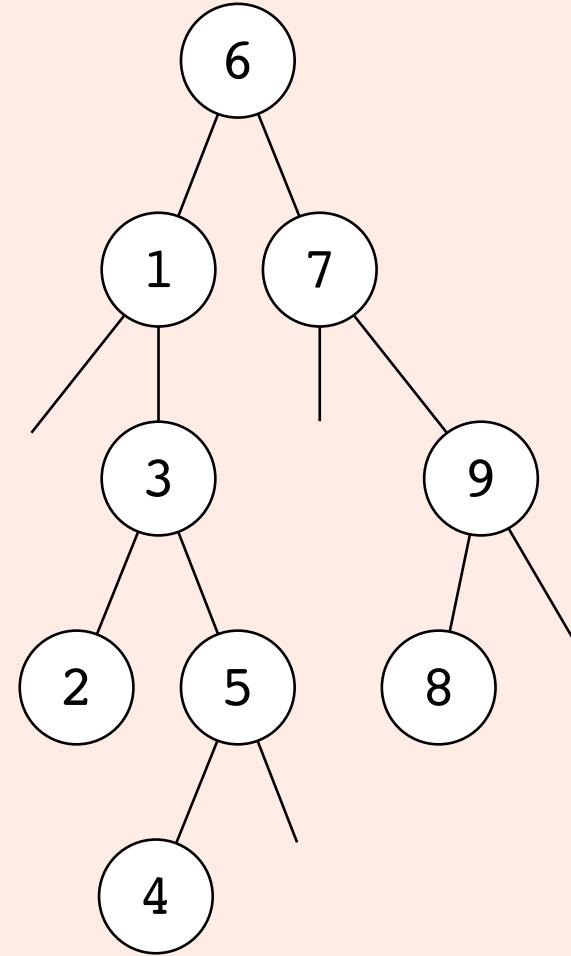


# Solution 2

- En partant d'un ABR vide, on insère les valeurs suivantes dans cet ordre

6, 1, 3, 5, 7, 9, 8, 4, 2

- L'arbre résultant est ...



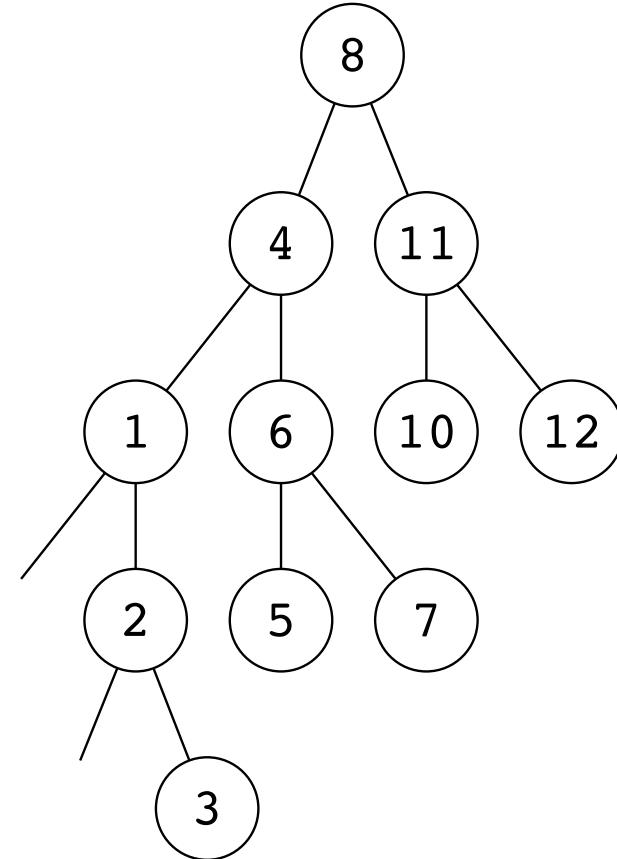
# 10. Suppression d'éléments





# Recherche du minimum

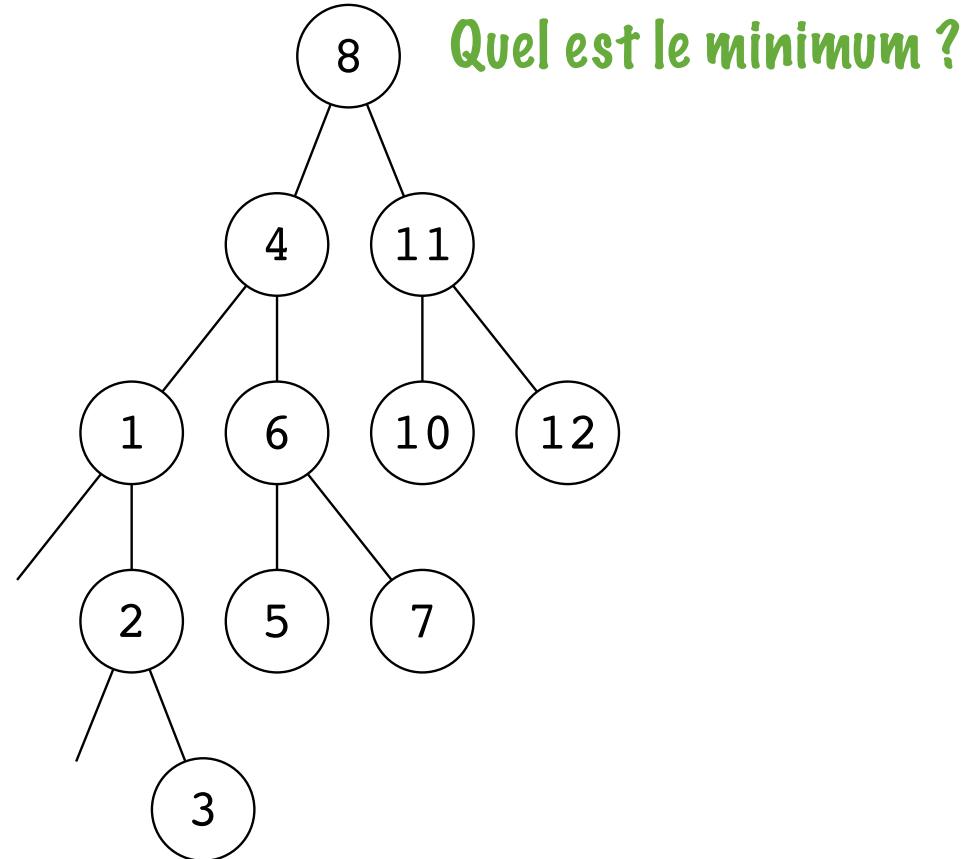
- Si la racine a un sous-arbre gauche, c'est là que se trouve le minimum
- Si la racine n'a pas de sous-arbre gauche, elle est le minimum





# Recherche du minimum

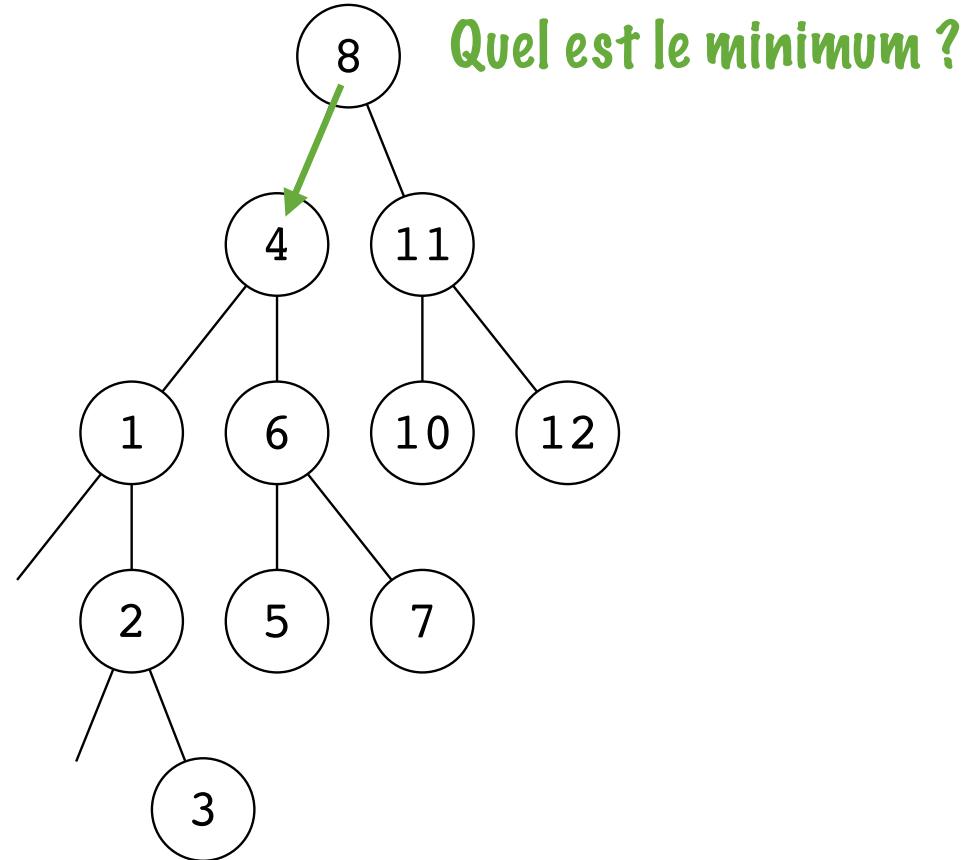
- Si la racine a un sous-arbre gauche, c'est là que se trouve le minimum
- Si la racine n'a pas de sous-arbre gauche, elle est le minimum





# Recherche du minimum

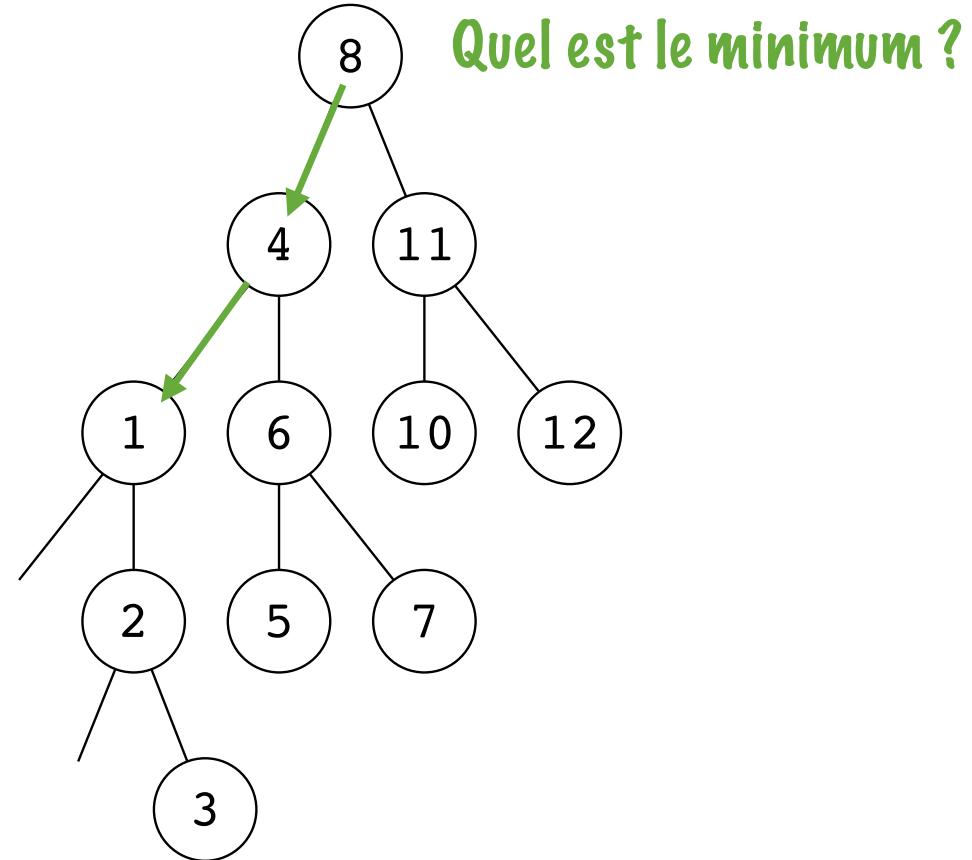
- Si la racine a un sous-arbre gauche, c'est là que se trouve le minimum
- Si la racine n'a pas de sous-arbre gauche, elle est le minimum





# Recherche du minimum

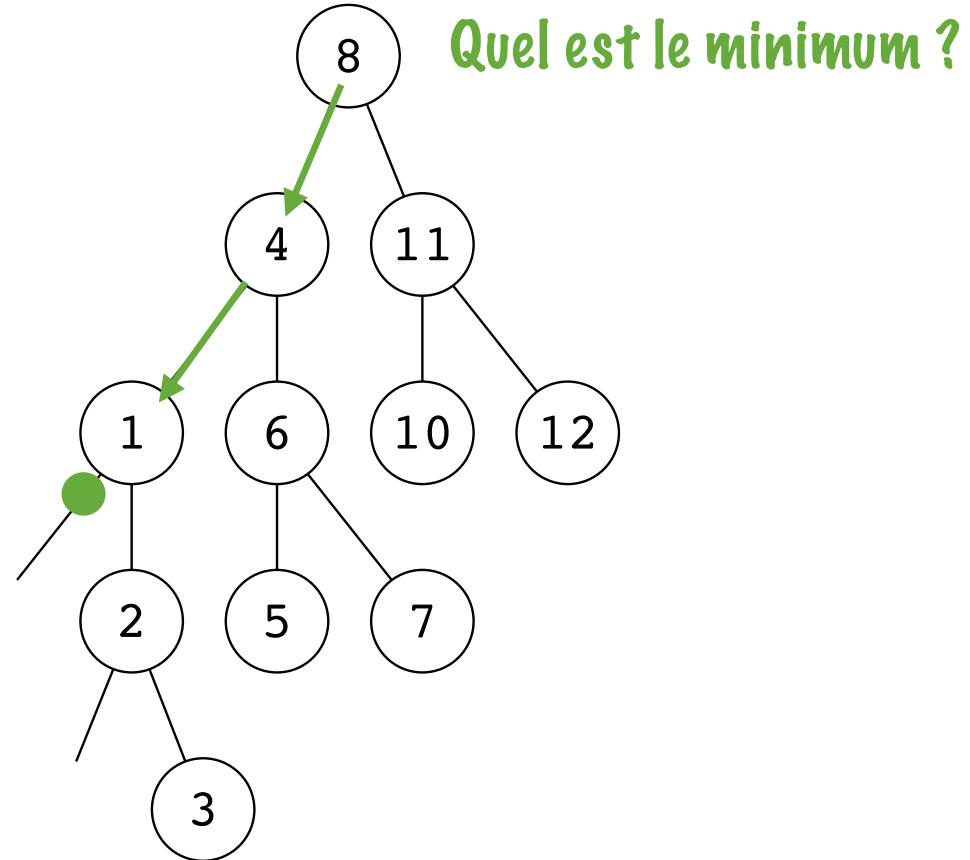
- Si la racine a un sous-arbre gauche, c'est là que se trouve le minimum
- Si la racine n'a pas de sous-arbre gauche, elle est le minimum





# Recherche du minimum

- Si la racine a un sous-arbre gauche, c'est là que se trouve le minimum
- Si la racine n'a pas de sous-arbre gauche, elle est le minimum





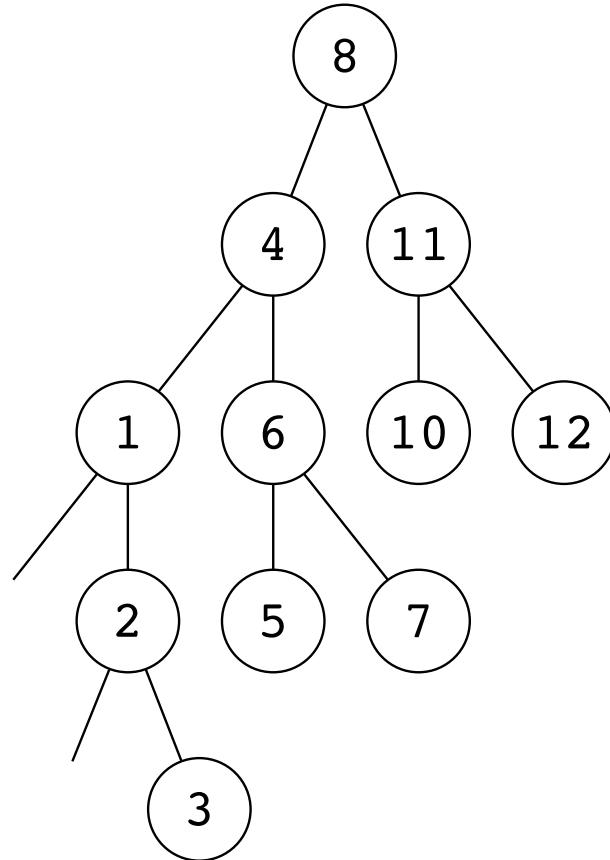
# Recherche du minimum

- Récursivement

```
fonction min (r)
    si r.gauche != Ø
        retourner min(r.gauche)
    sinon
        retourner r
```

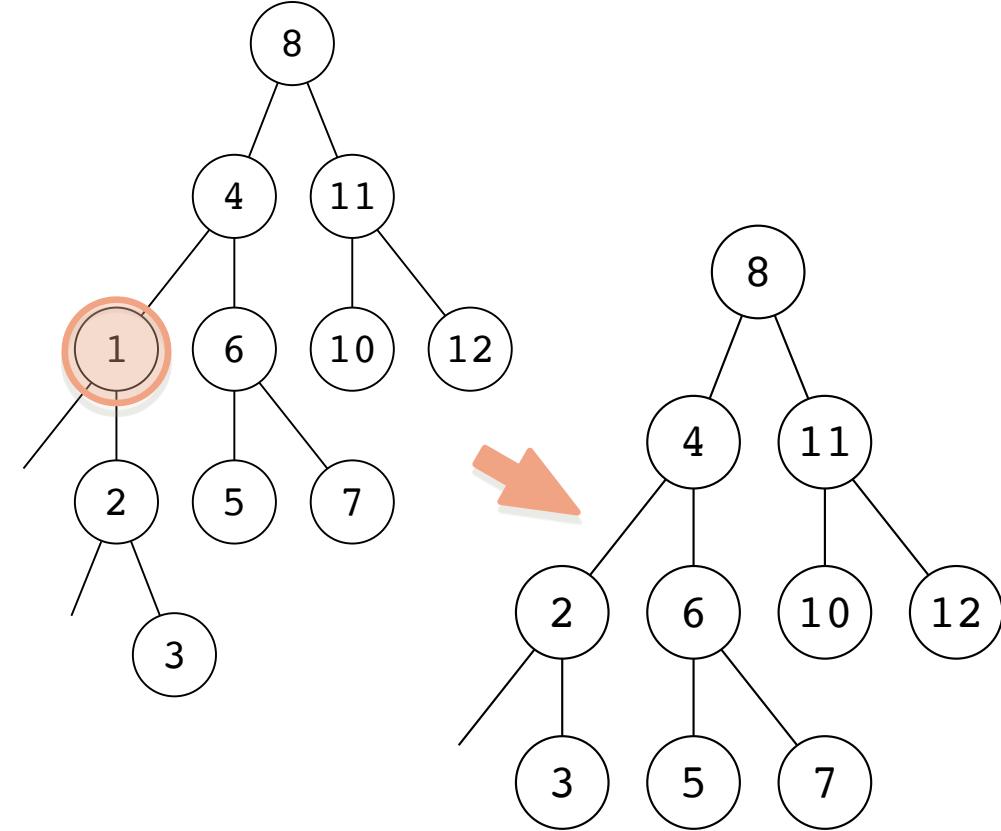
- Itérativement

```
fonction min (r)
    m ← r
    tant que m.gauche != Ø
        m ← m.gauche
    retourner m
```





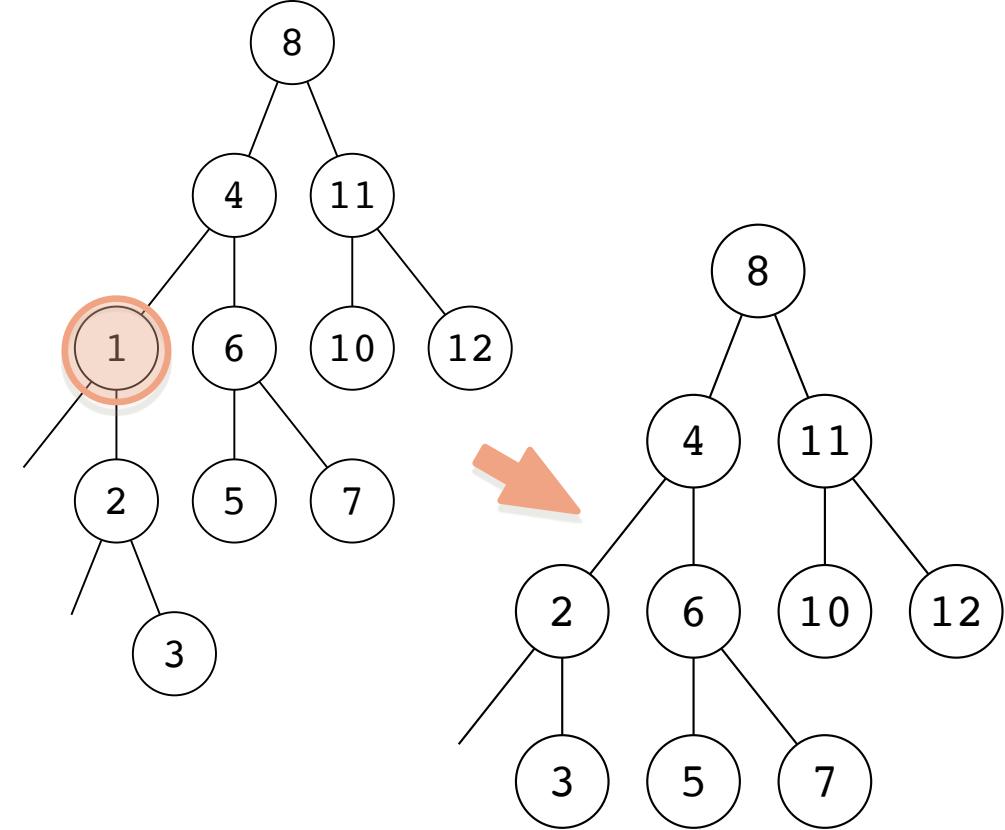
# Suppression du minimum





# Suppression du minimum

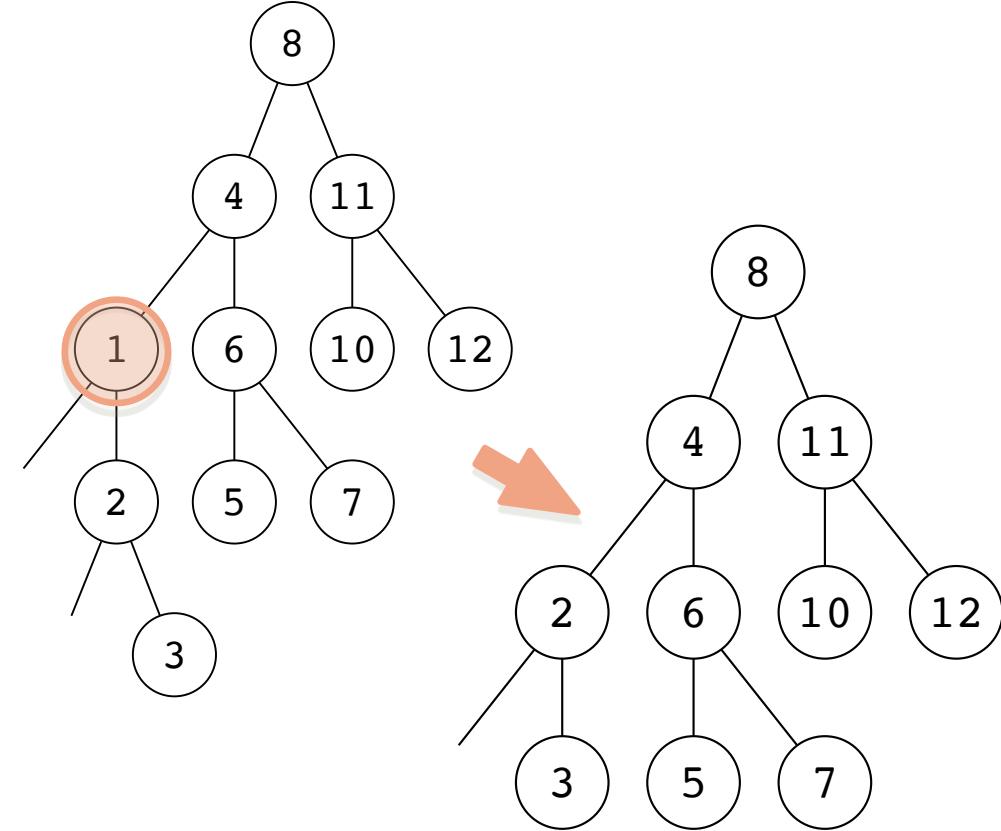
- Trouver le minimum





# Suppression du minimum

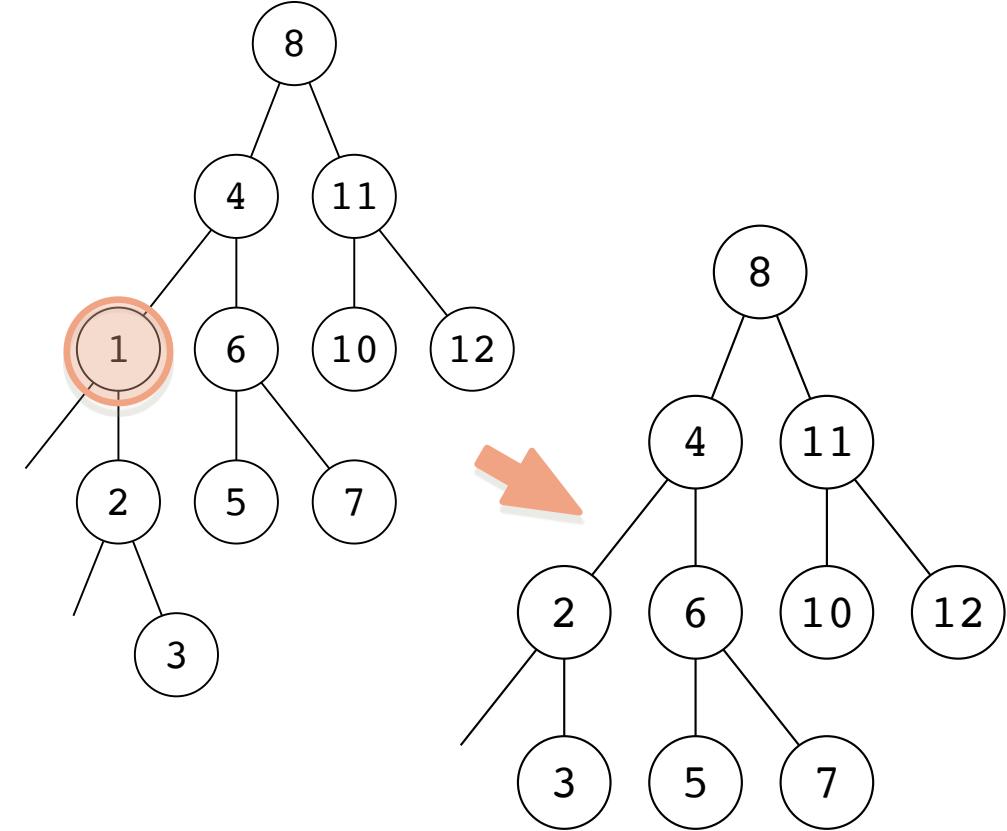
- Trouver le minimum
- Oter le noeud de l'arbre en raccrochant ses descendants à son parent





# Suppression du minimum

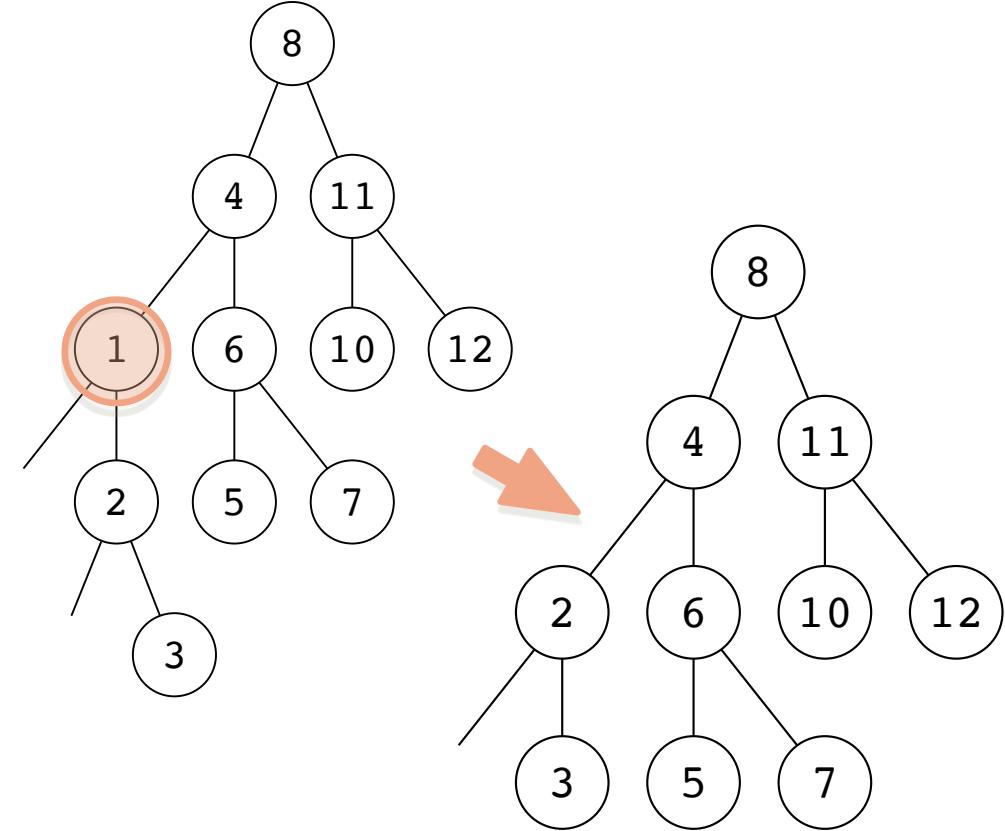
- Trouver le minimum
- Oter le noeud de l'arbre en raccrochant ses descendants à son parent
- Effacer le noeud de la mémoire





# Suppression du minimum

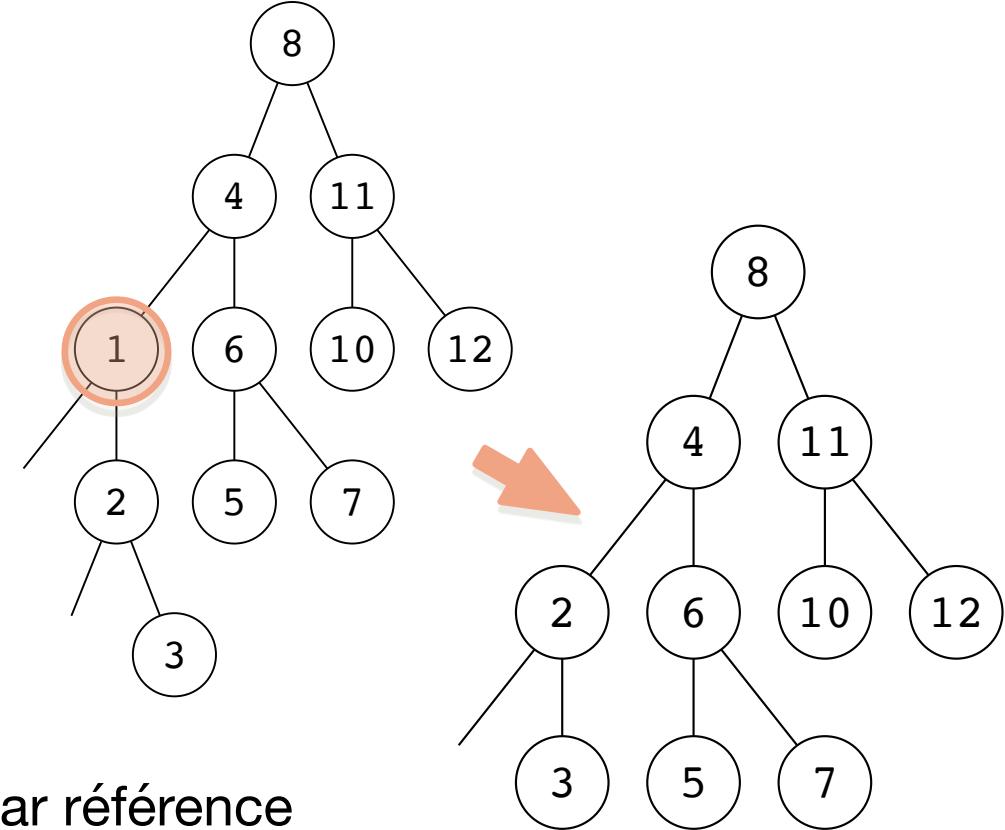
```
fonction supprimer_min (ref r)
```





# Suppression du minimum

```
fonction supprimer_min (ref r)
```

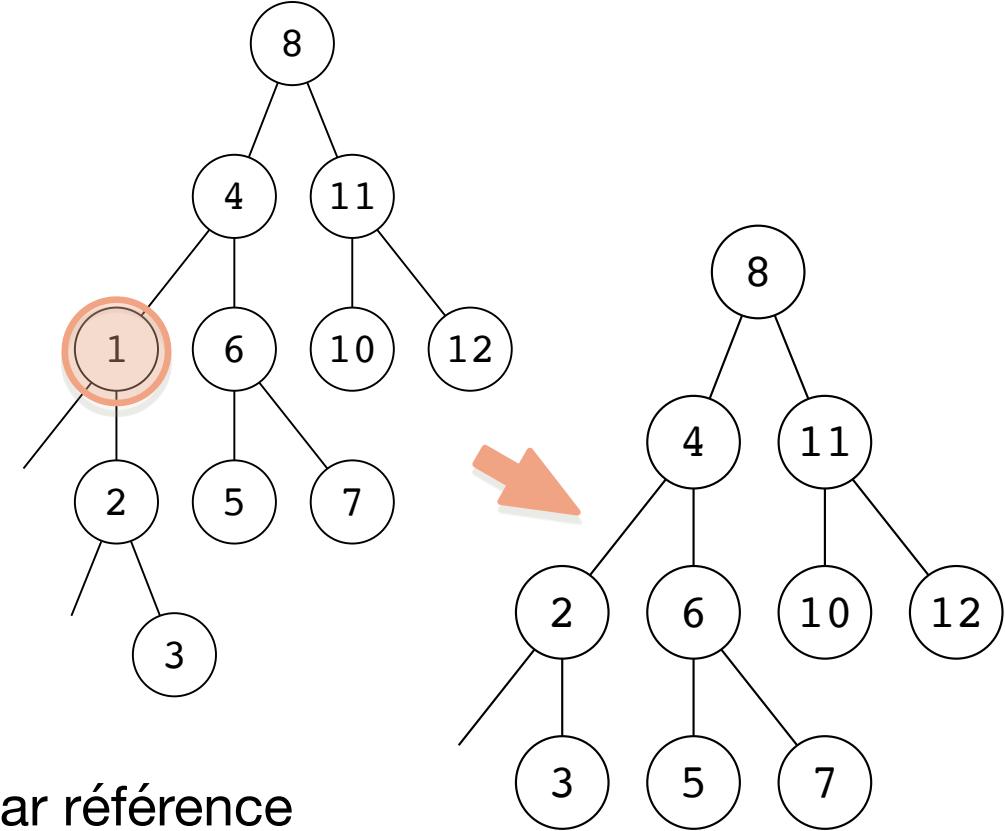


- Notons que la racine est passée par référence



# Suppression du minimum

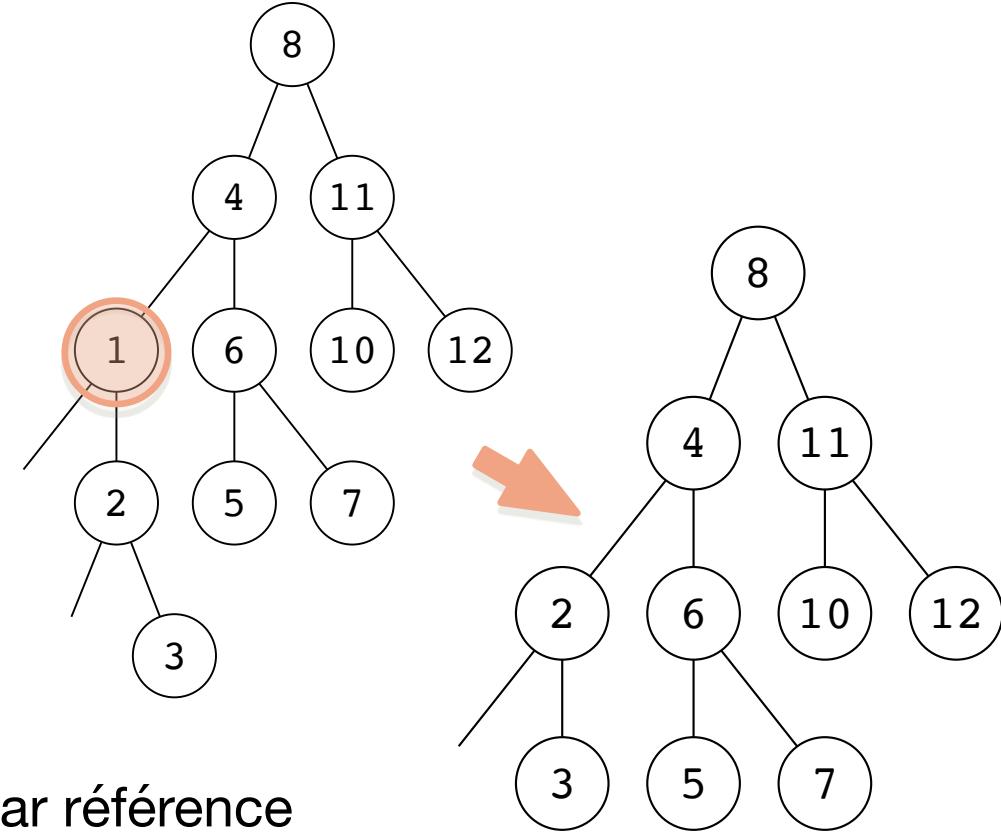
```
fonction supprimer_min (ref r)
    si r == Ø
        signaler erreur
```



- Notons que la racine est passée par référence

# Suppression du minimum

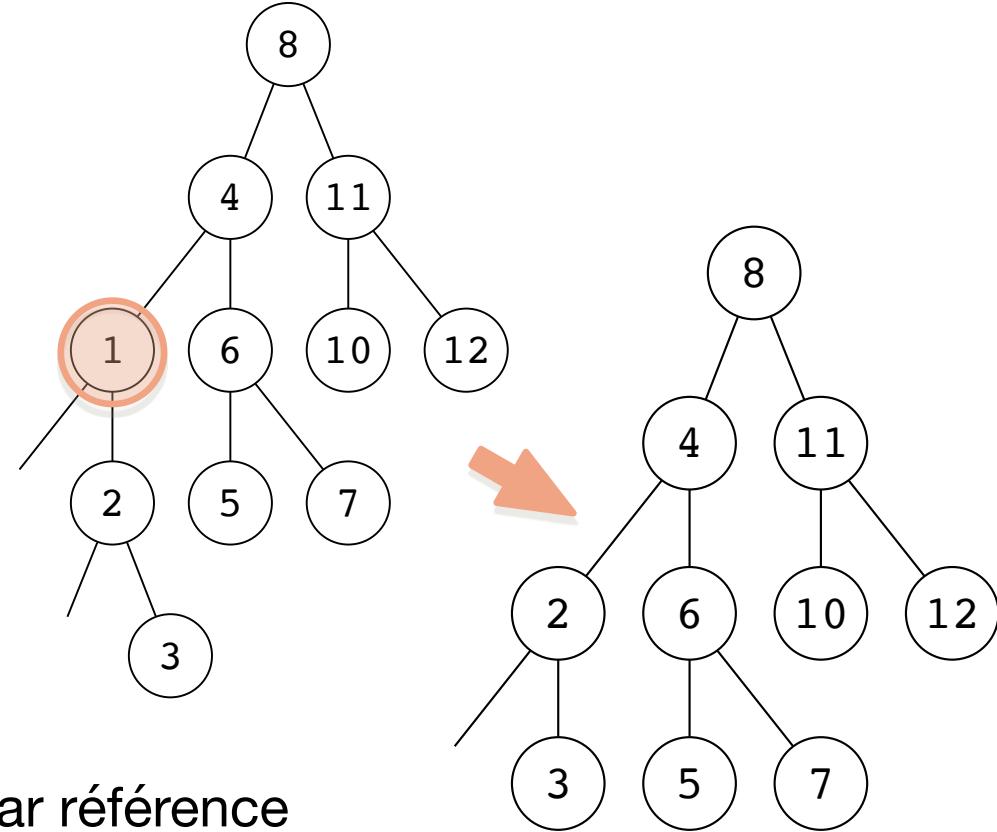
```
fonction supprimer_min (ref r)  
  
    si r == Ø  
        signaler erreur  
    sinon, si r.gauche != Ø  
        supprimer_min (r.gauche)
```



- Notons que la racine est passée par référence

# Suppression du minimum

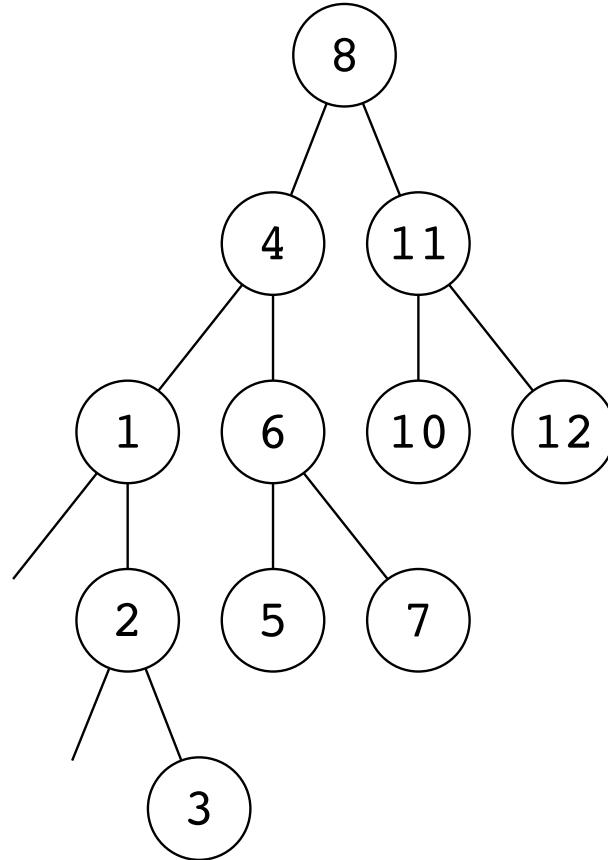
```
fonction supprimer_min (ref r)
    si r == Ø
        signaler erreur
    sinon, si r.gauche != Ø
        supprimer_min (r.gauche)
    sinon, // r est le minimum
        d ← r.droit
        effacer r
        r ← d
```



- Notons que la racine est passée par référence



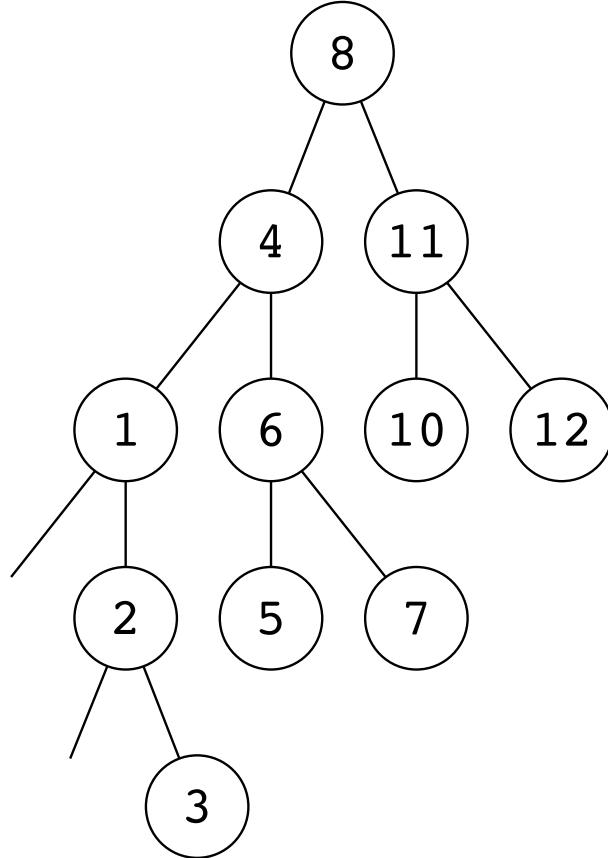
# Suppression de la clé k





# Suppression de la clé k

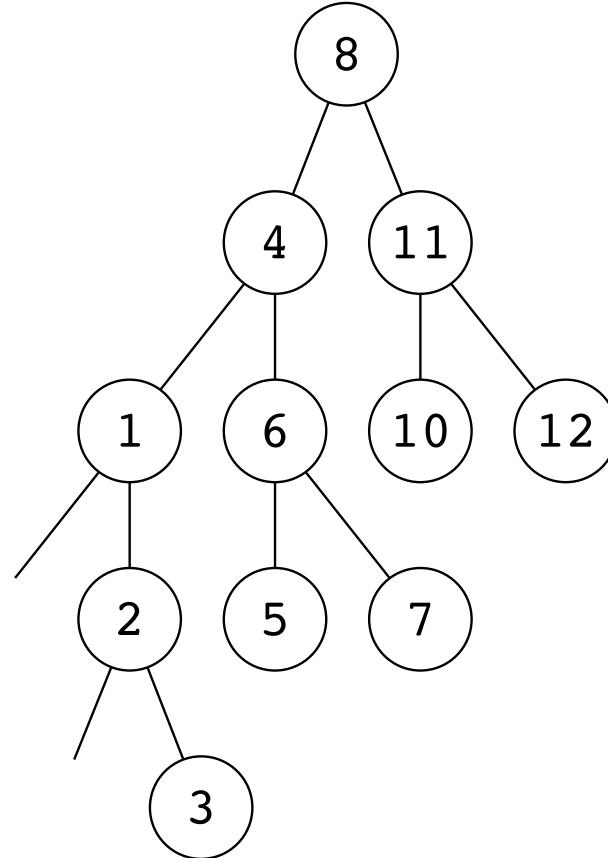
- Trouver le noeud de clé k





# Suppression de la clé k

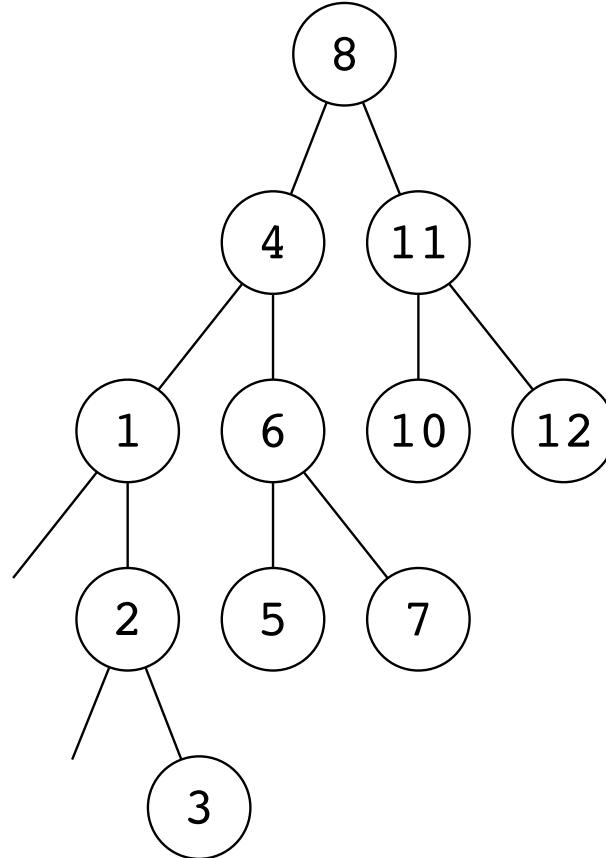
- Trouver le noeud de clé k
- Oter le noeud de l'arbre en raccrochant ses descendants à son parent





# Suppression de la clé k

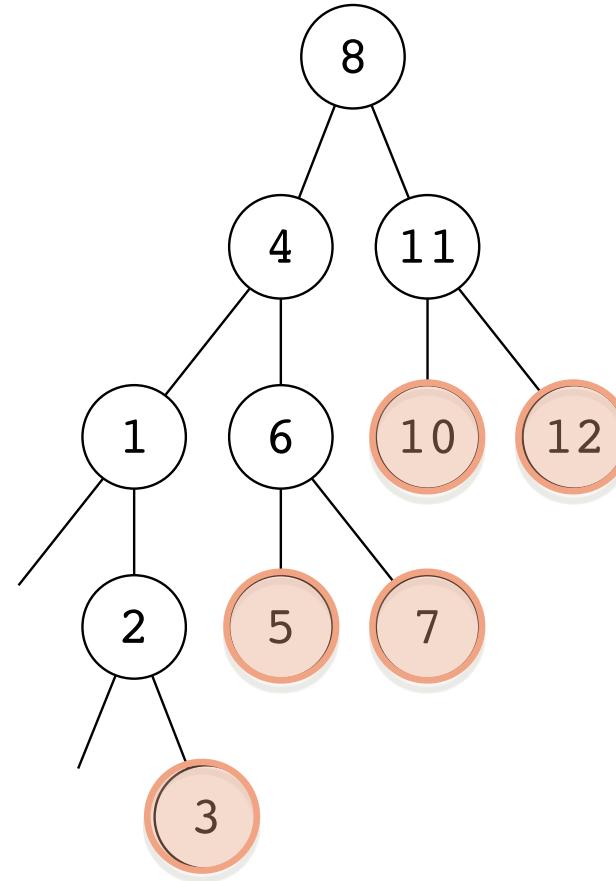
- Trouver le noeud de clé k
- Oter le noeud de l'arbre en raccrochant ses descendants à son parent
- Effacer le noeud de la mémoire





# Raccrocher les descendants au parent ...

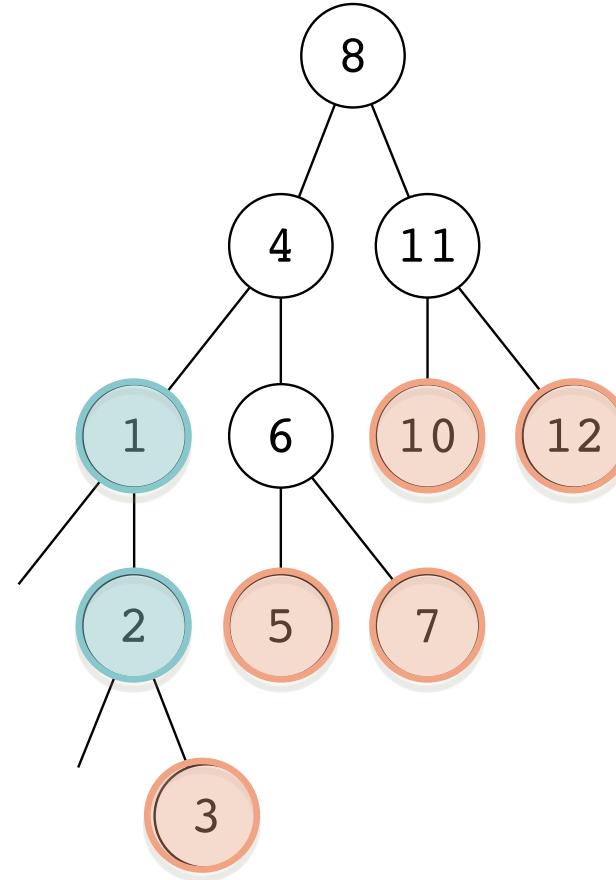
- Feuilles ... rien à faire





# Raccrocher les descendants au parent ...

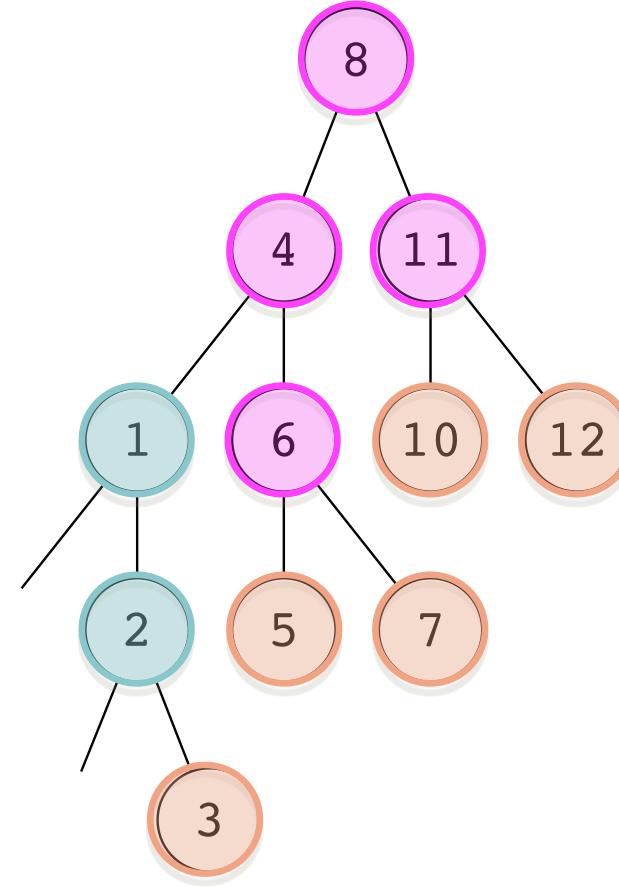
- Feuilles ... rien à faire
- Degré 1 ... comme pour le minimum





# Raccrocher les descendants au parent ...

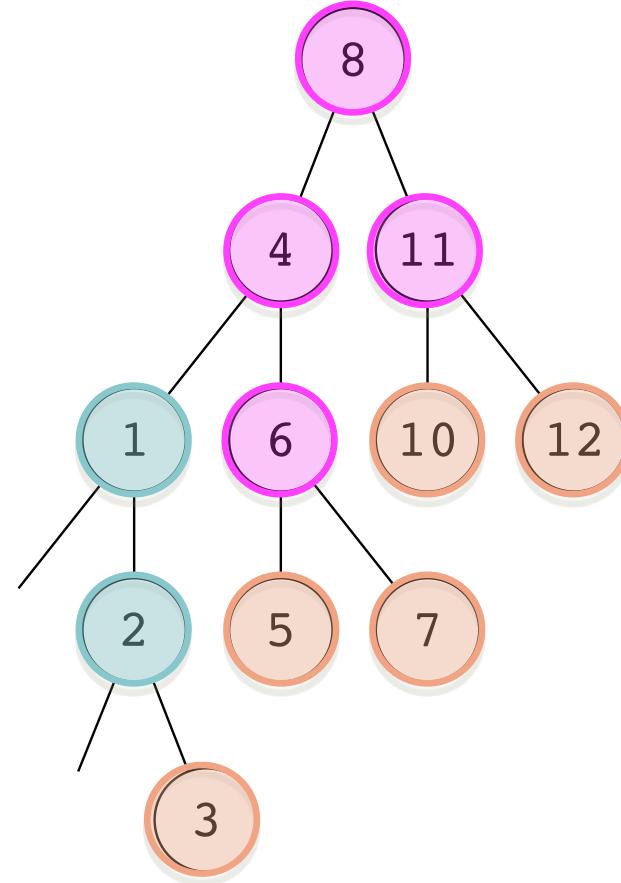
- Feuilles ... rien à faire
- Degré 1 ... comme pour le minimum
- Degré 2 ... choisir un des noeuds descendant comme racine du sous-arbre à raccrocher.





# Raccrocher les descendants au parent ...

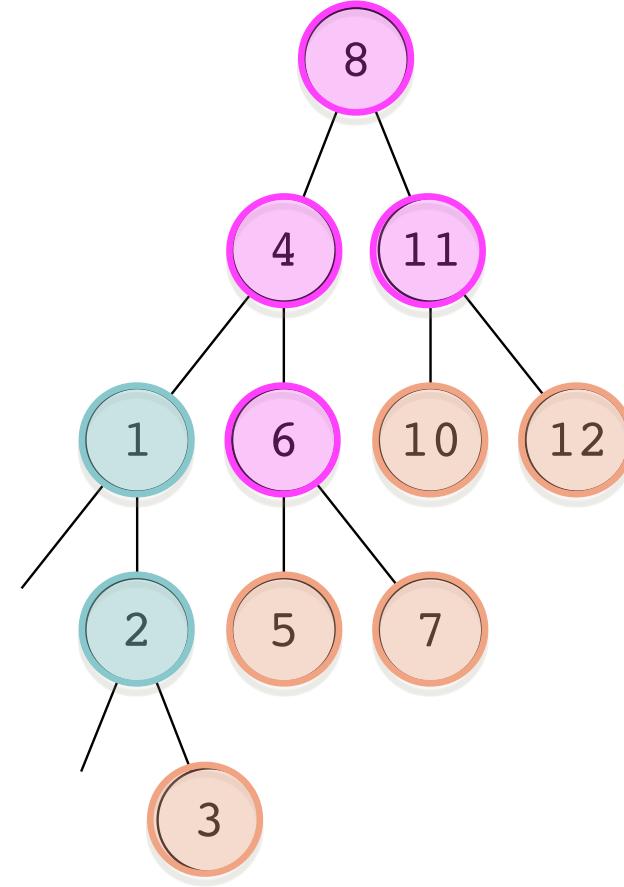
- Feuilles ... rien à faire
- Degré 1 ... comme pour le minimum
- Degré 2 ... choisir un des noeuds descendant comme racine du sous-arbre à raccrocher.
- Minimum du sous-arbre droit ou maximum du sous-arbre gauche





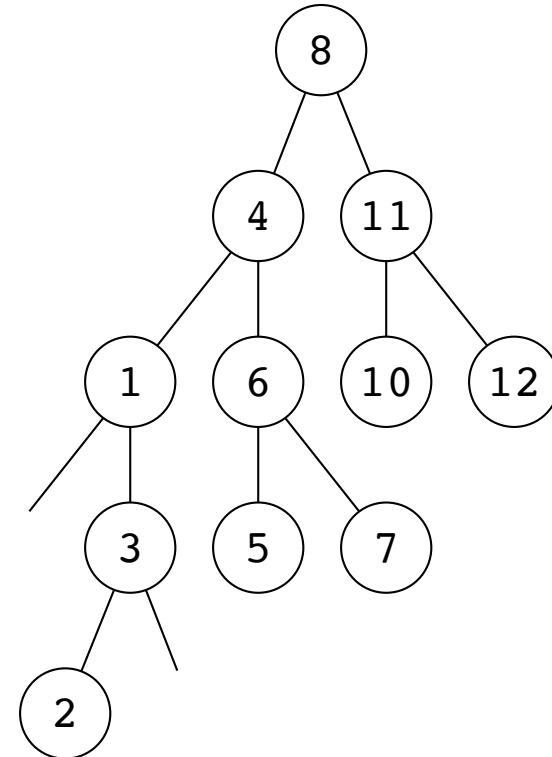
# Raccrocher les descendants au parent ...

- Feuilles ... rien à faire
- Degré 1 ... comme pour le minimum
- Degré 2 ... choisir un des noeuds descendant comme racine du sous-arbre à raccrocher.
- Minimum du sous-arbre droit ou maximum du sous-arbre gauche
- Technique de Thomas N. Hibbard



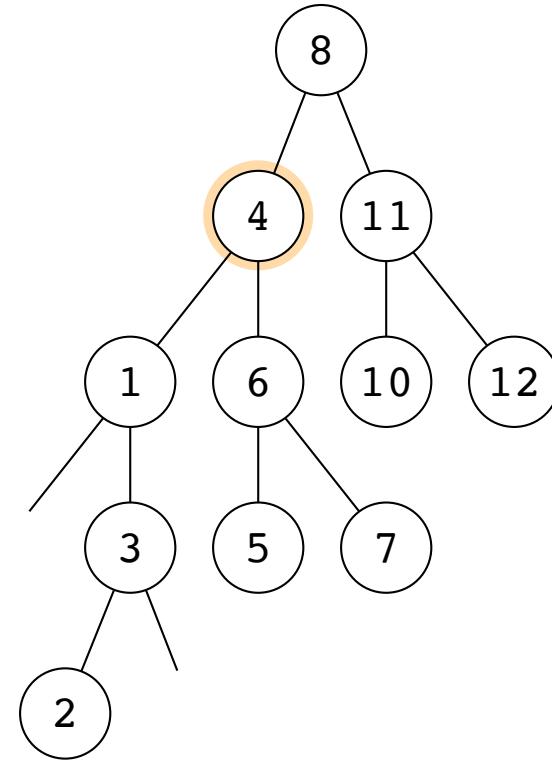


# Exemple : suppression de la clé 4



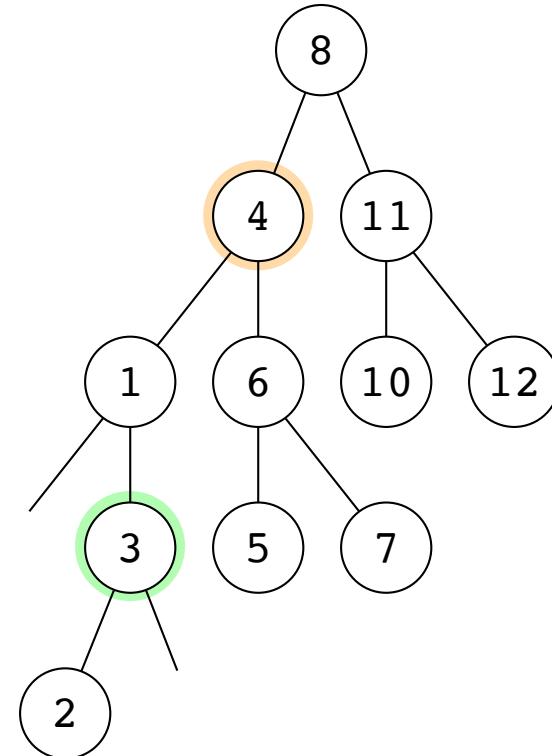


# Exemple : suppression de la clé 4



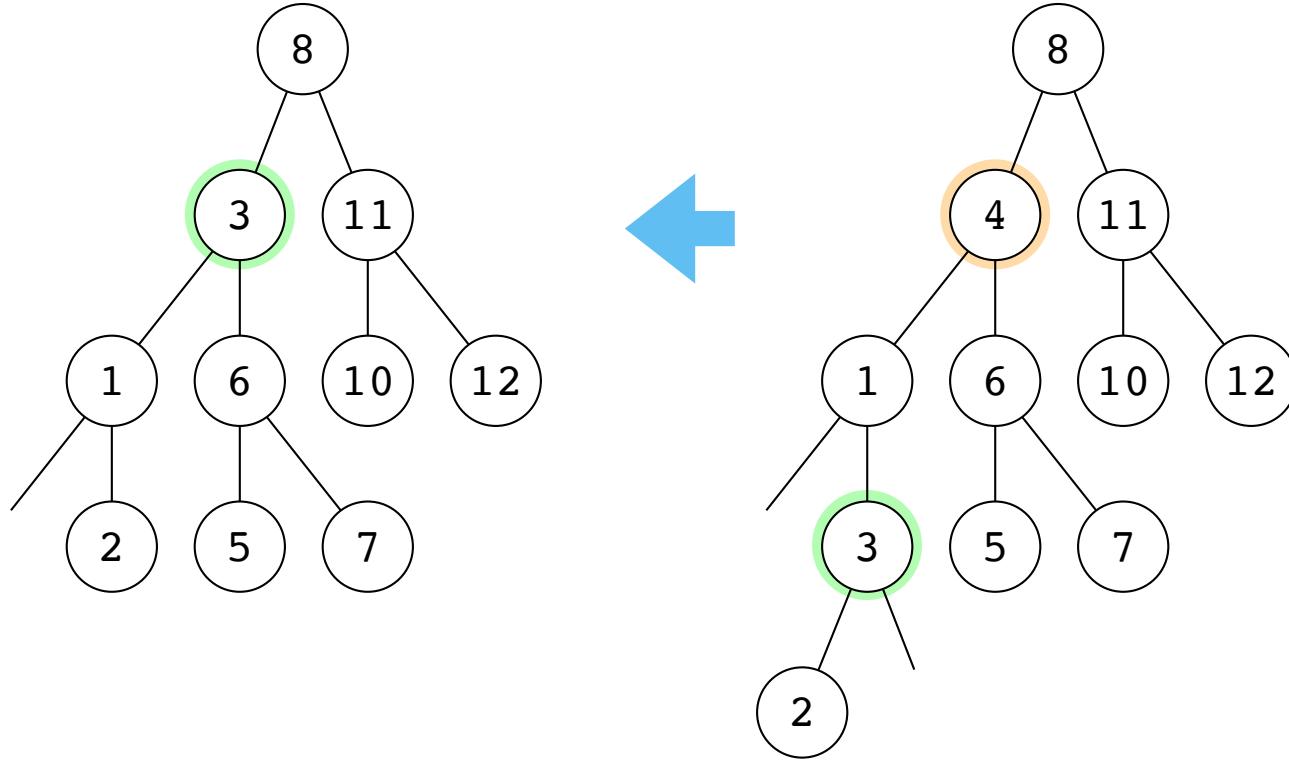


# Exemple : suppression de la clé 4



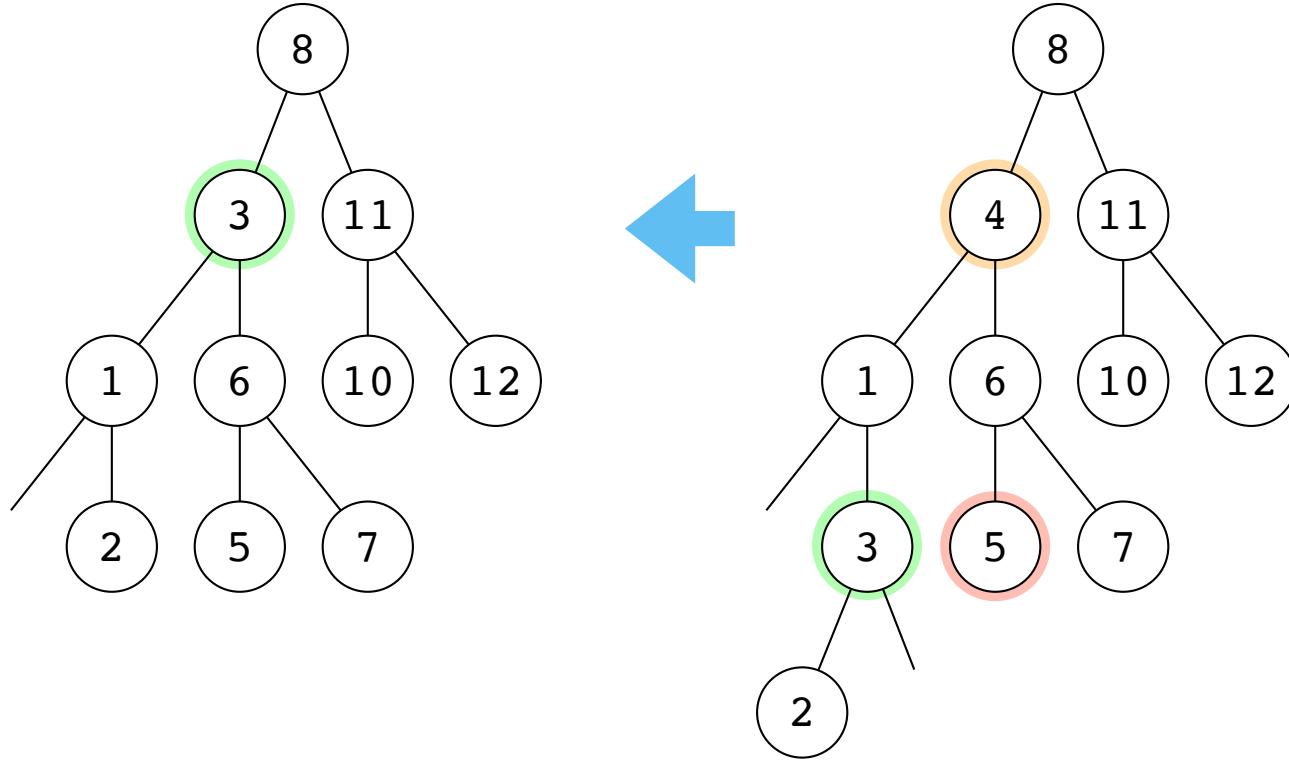


# Exemple : suppression de la clé 4



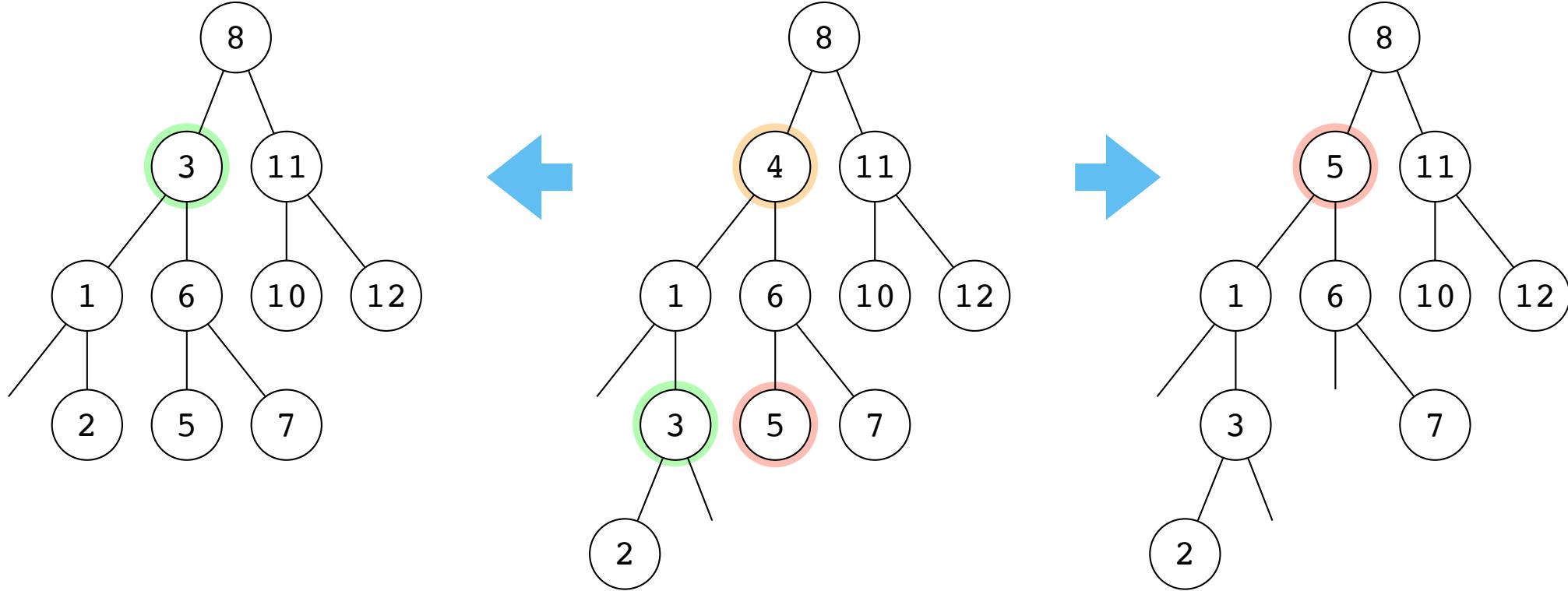


# Exemple : suppression de la clé 4





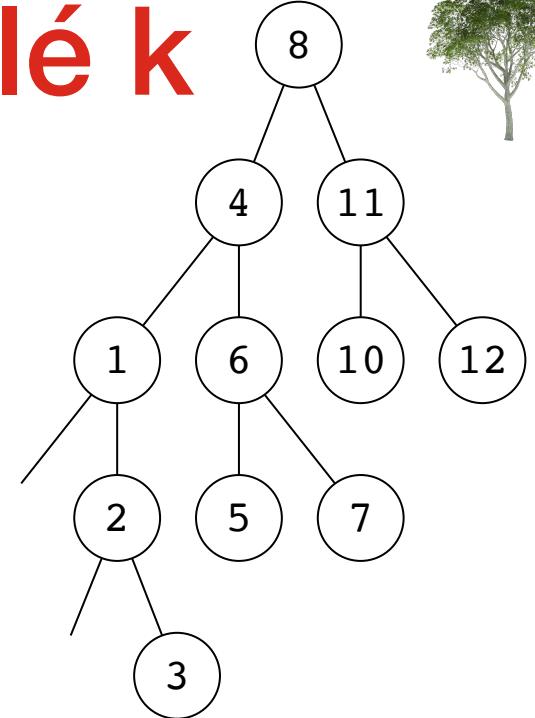
## Exemple : suppression de la clé 4



# Suppression de la clé k



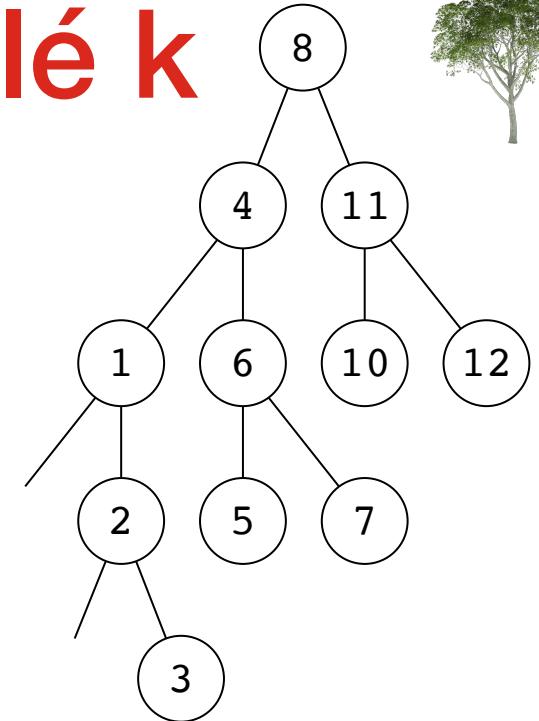
```
fonction supprimer (ref r, k)
```



# Suppression de la clé k



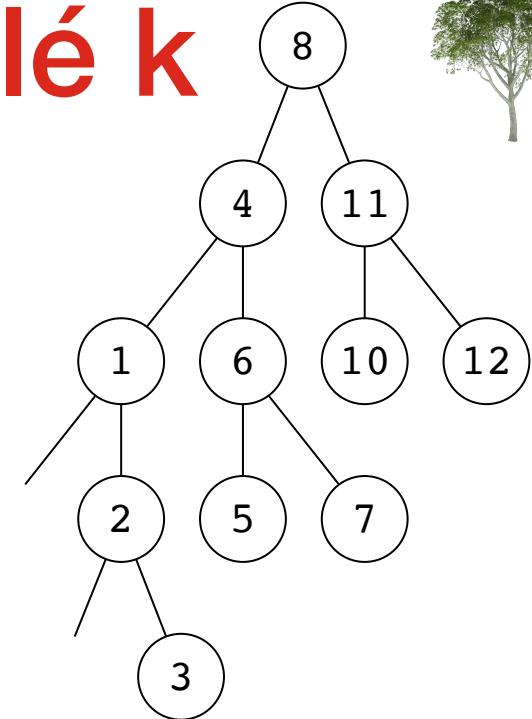
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
```





# Suppression de la clé k

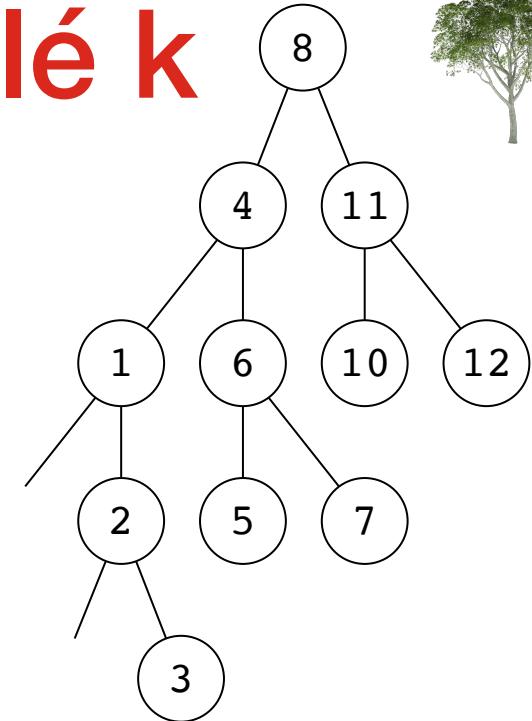
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
```





# Suppression de la clé k

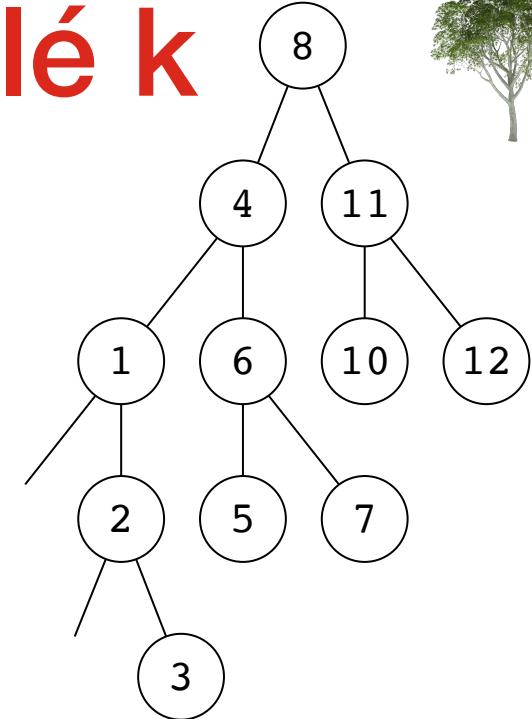
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
```





# Suppression de la clé k

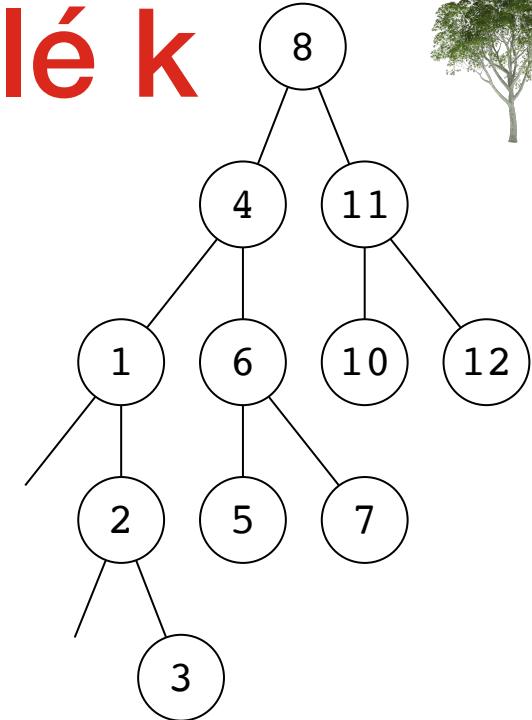
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
```





# Suppression de la clé k

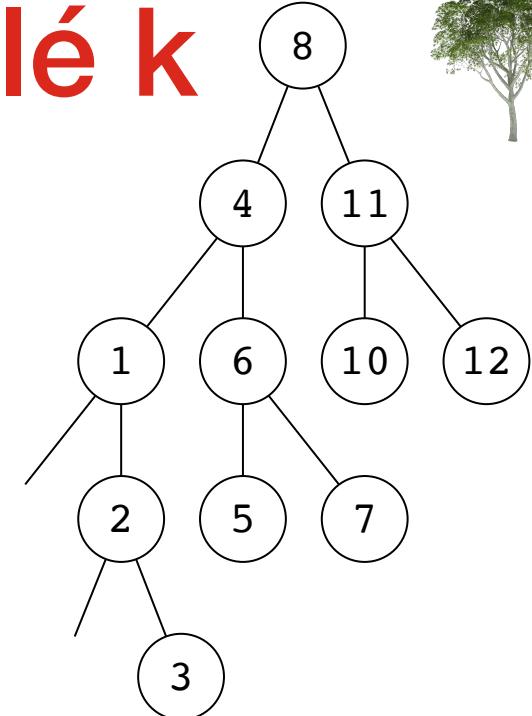
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
```





# Suppression de la clé k

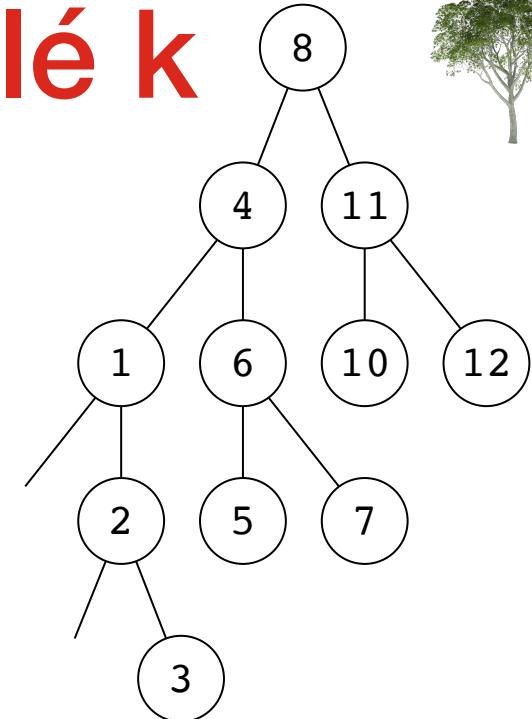
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
        sinon, si r.droit == Ø
            r ← r.gauche
```





# Suppression de la clé k

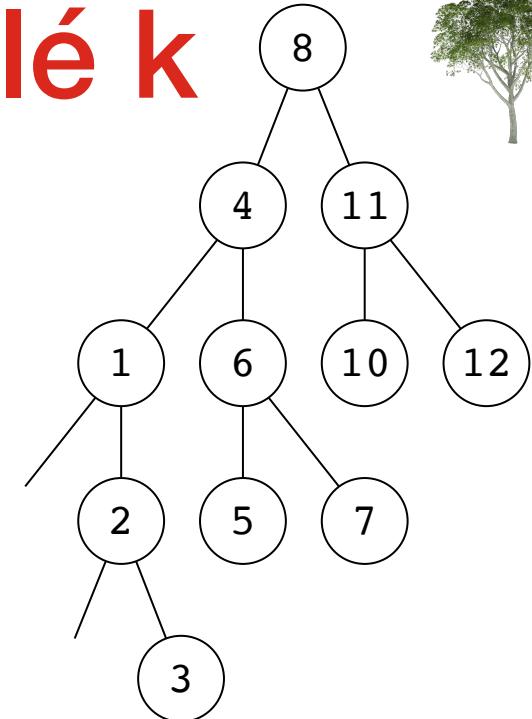
```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
        sinon, si r.droit == Ø
            r ← r.gauche
        sinon, // Hibbard
            m ← sortir_min (r.droit)
            m.droit ← r.droit
            m.gauche ← r.gauche
            r ← m
```





# Suppression de la clé k

```
fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
        sinon, si r.droit == Ø
            r ← r.gauche
        sinon, // Hibbard
            m ← sortir_min (r.droit)
            m.droit ← r.droit
            m.gauche ← r.gauche
            r ← m
    effacer tmp
```



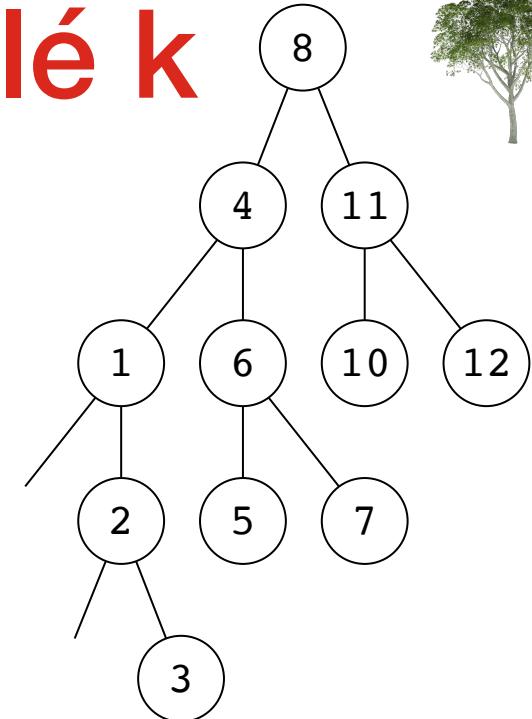


# Suppression de la clé k

```

fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
        sinon, si r.droit == Ø
            r ← r.gauche
        sinon, // Hibbard
            m ← sortir_min (r.droit)
            m.droit ← r.droit
            m.gauche ← r.gauche
            r ← m
    effacer tmp

```



```
fonction sortir_min (ref r)
```

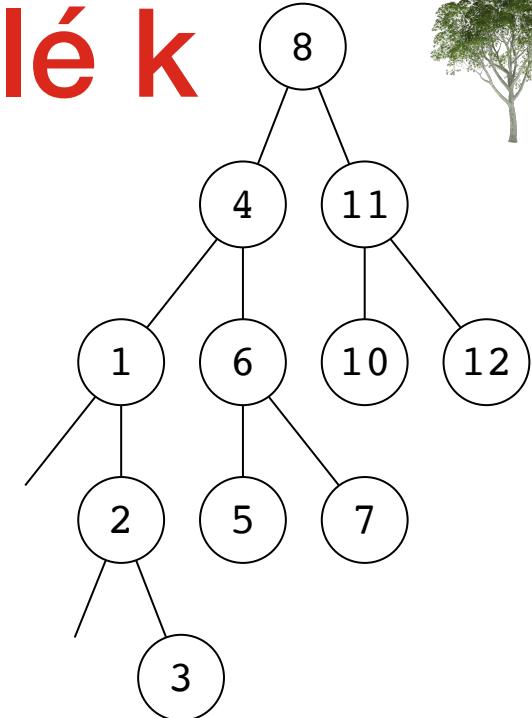


# Suppression de la clé k

```

fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
        sinon, si r.droit == Ø
            r ← r.gauche
        sinon, // Hibbard
            m ← sortir_min (r.droit)
            m.droit ← r.droit
            m.gauche ← r.gauche
            r ← m
    effacer tmp

```



```

fonction sortir_min (ref r)
    si r.gauche != Ø
        retourner sortir_min (r.gauche)

```

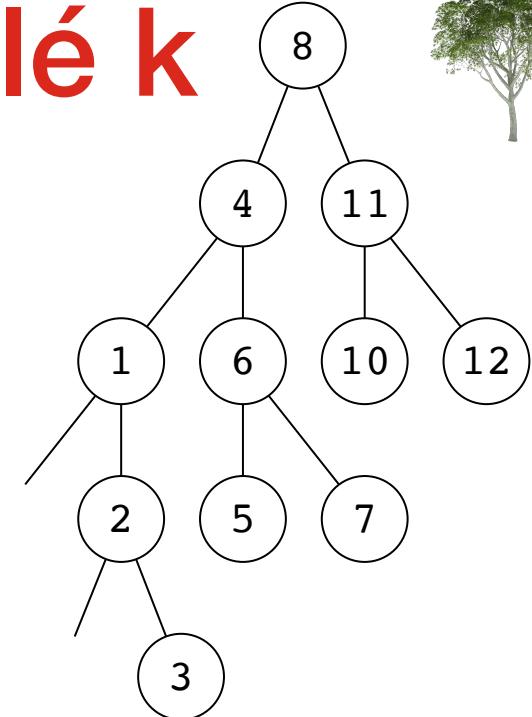


# Suppression de la clé k

```

fonction supprimer (ref r, k)
    si r == Ø
        // k est absent
    sinon, si k < r.clé
        supprimer (r.gauche, k)
    sinon, si k > r.clé
        supprimer (r.droit, k)
    sinon, // k est trouvé
        tmp ← r
        si r.gauche == Ø
            r ← r.droit
        sinon, si r.droit == Ø
            r ← r.gauche
        sinon, // Hibbard
            m ← sortir_min (r.droit)
            m.droit ← r.droit
            m.gauche ← r.gauche
            r ← m
    effacer tmp

```



```

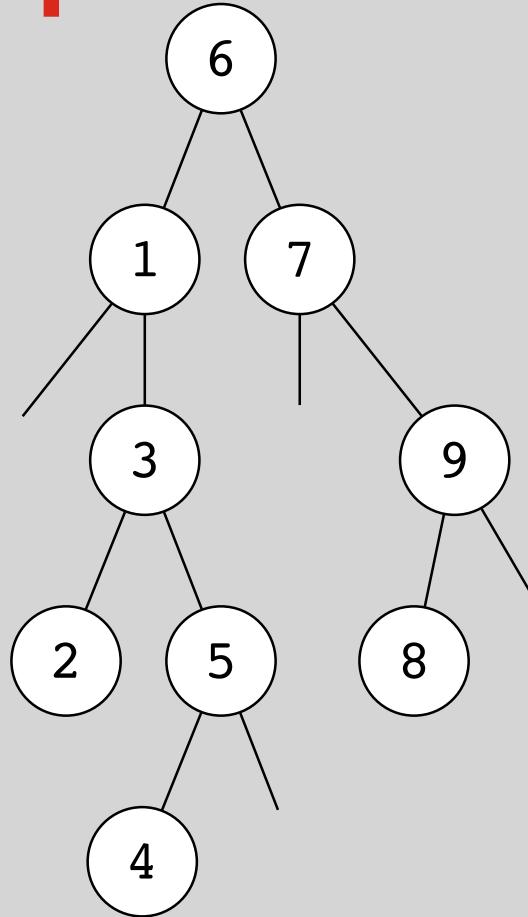
fonction sortir_min (ref r)
    si r.gauche != Ø
        retourner sortir_min (r.gauche)
    sinon, // r est le minimum
        tmp ← r
        r ← r.droit
    retourner tmp

```



# Exercice 1

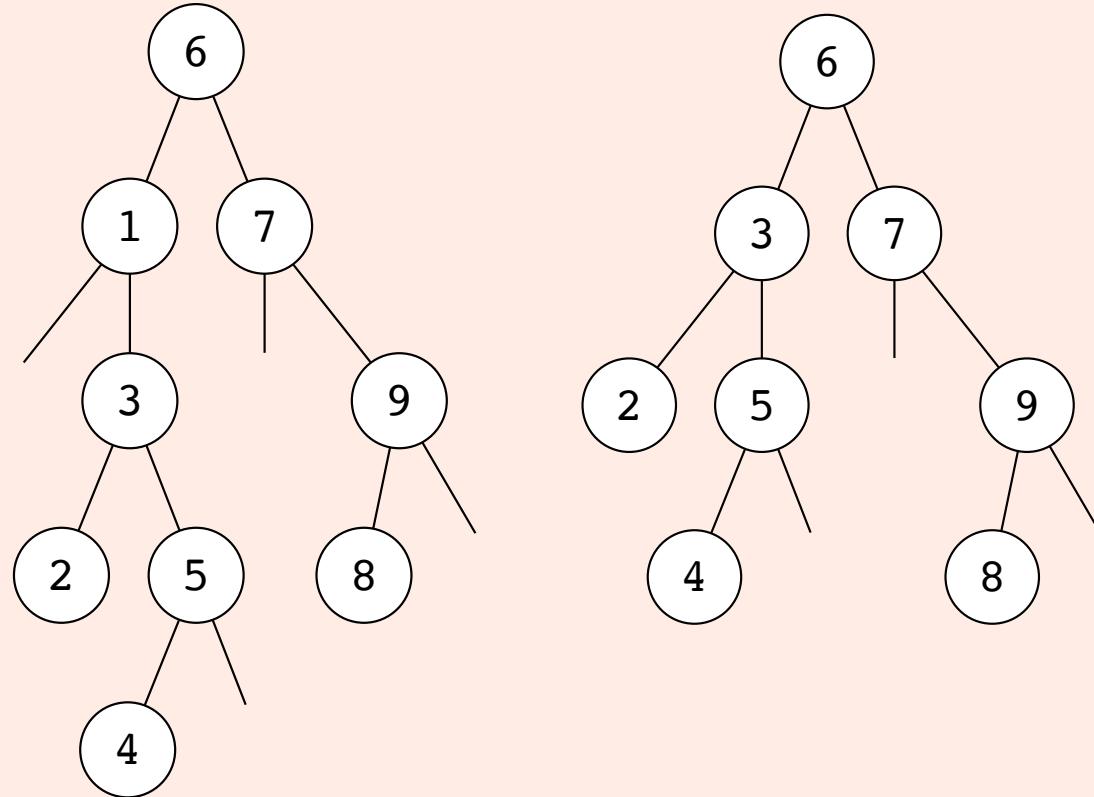
- On supprime le minimum de l'ABR ci-contre. Dessinez l'arbre résultant





# Solution 1

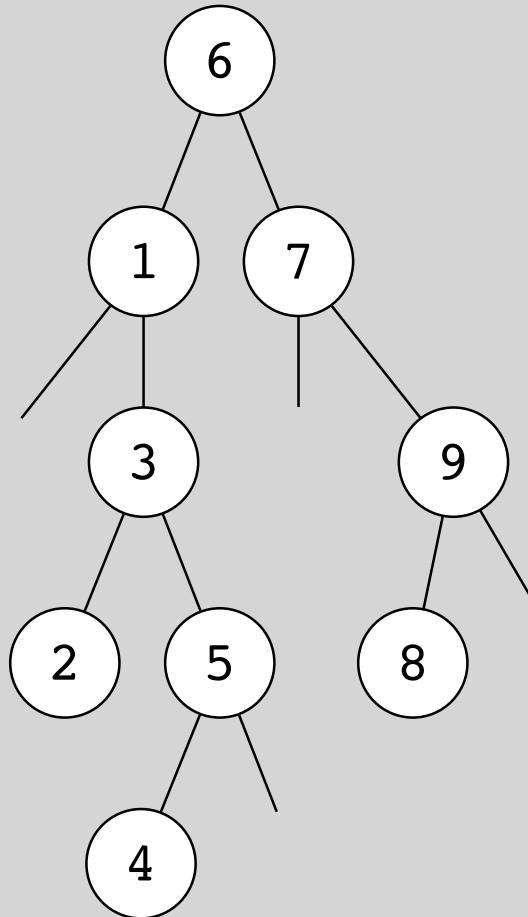
- En supprimant le minimum de l'ABR de gauche, on obtient celui de droite





# Exercice 2

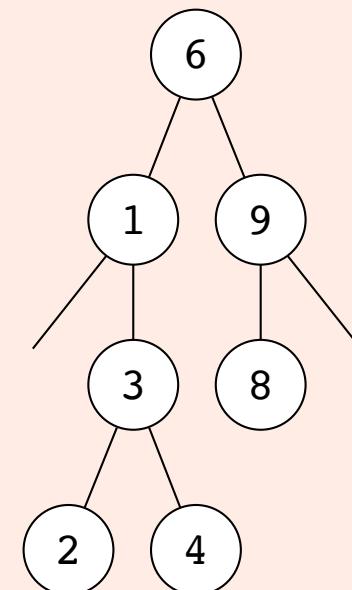
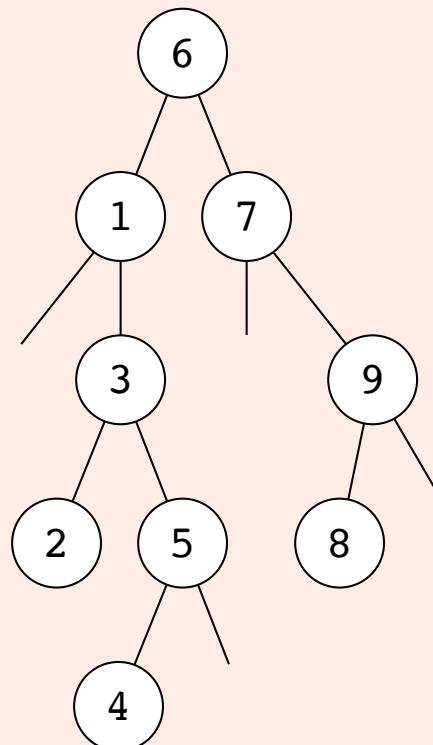
- On supprime les 2 noeuds de clé 5 et de clé 7 de l'arbre ci-contre. Dessinez l'arbre résultant





# Solution 2

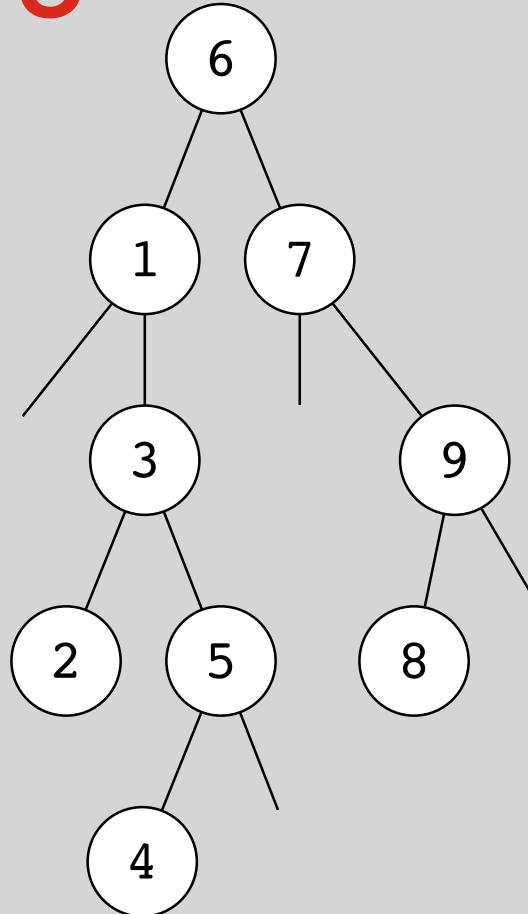
- En supprimant les 2 noeuds de clé 5 et de clé 7 de l'arbre de gauche, on obtient l'arbre de droite.





# Exercice 3

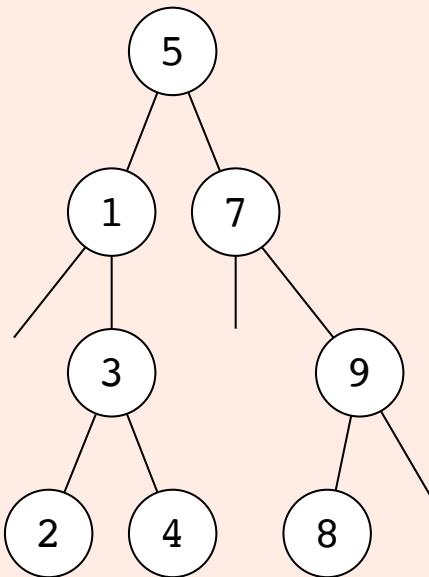
- On supprime le noeud de clé 6 de l'arbre ci-contre. Dessinez les deux arbres résultants possibles en appliquant la technique de Hibbard.



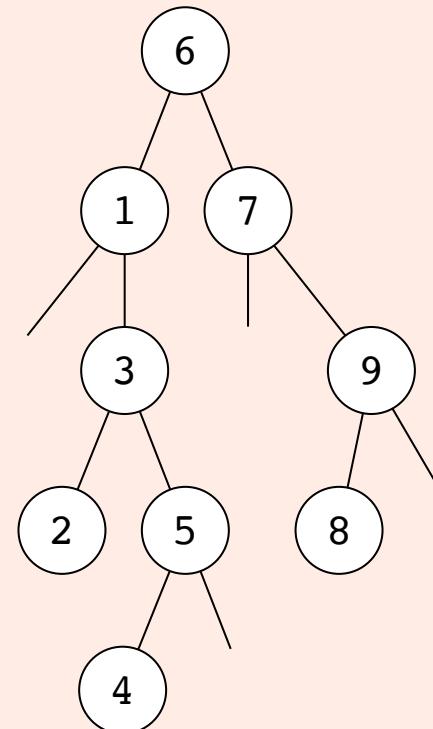
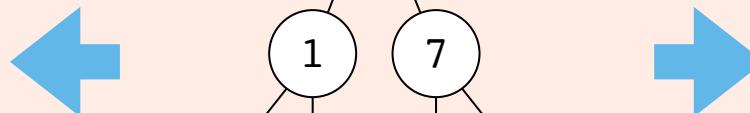


# Solution 3

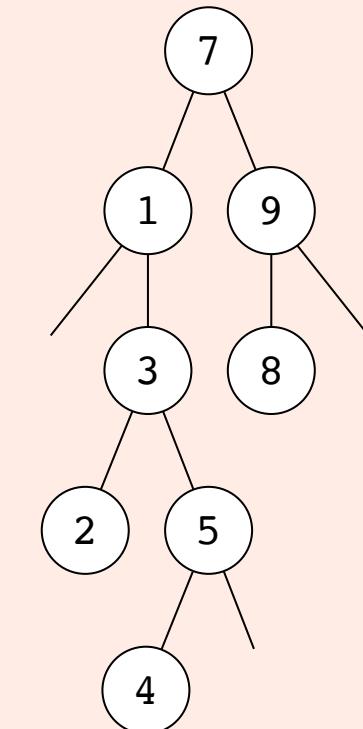
En remontant le maximum de gauche



Arbre dont on veut supprimer la clé 6



En remontant le minimum de droite



# 11. Itération sur un ABR





# Itérer sur un arbre (rappel)

Pré-ordre

```
fonction début(r)
    retourner r
```

```
fonction fin(r)
    retourner Ø
```

```
fonction suivant(r)
    si ainé(r) != Ø
        retourner ainé(r)
    sinon
        tant que puiné(r) = Ø
            et parent(r) != Ø
            r ← parent(r)
        retourner puiné(r)
```

Post-ordre

```
fonction début(r)
    tant que ainé(r) != Ø
        r ← ainé(r)
    retourner r
```

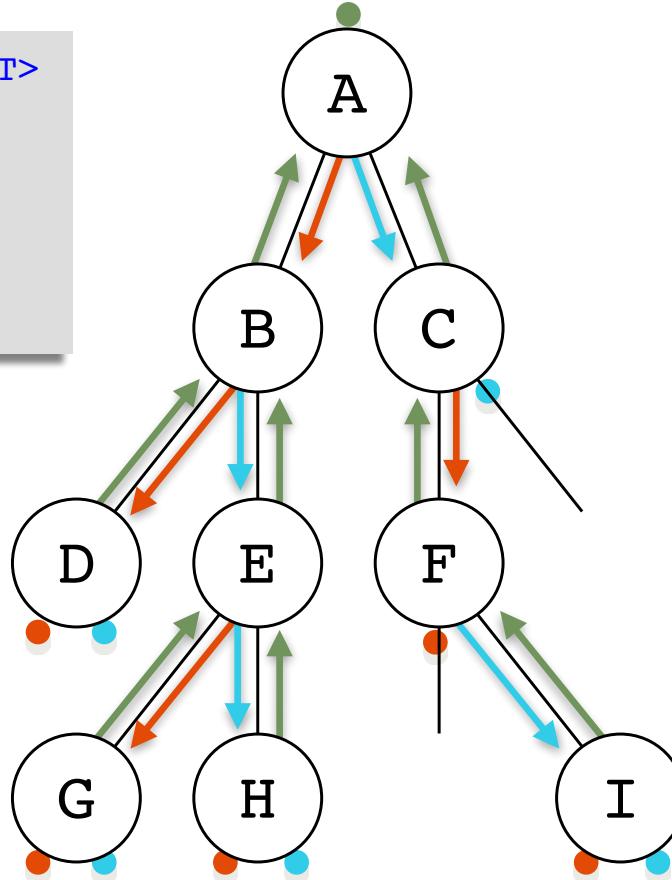
```
fonction fin(r)
    retourner Ø
```

```
fonction suivant(r)
    si puiné(r) != Ø
        retourner début(puiné(r))
    sinon
        retourner parent(r)
```



# parent, ainé, puiné dans un ABR

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Noeud* gauche
    Noeud* droit
```

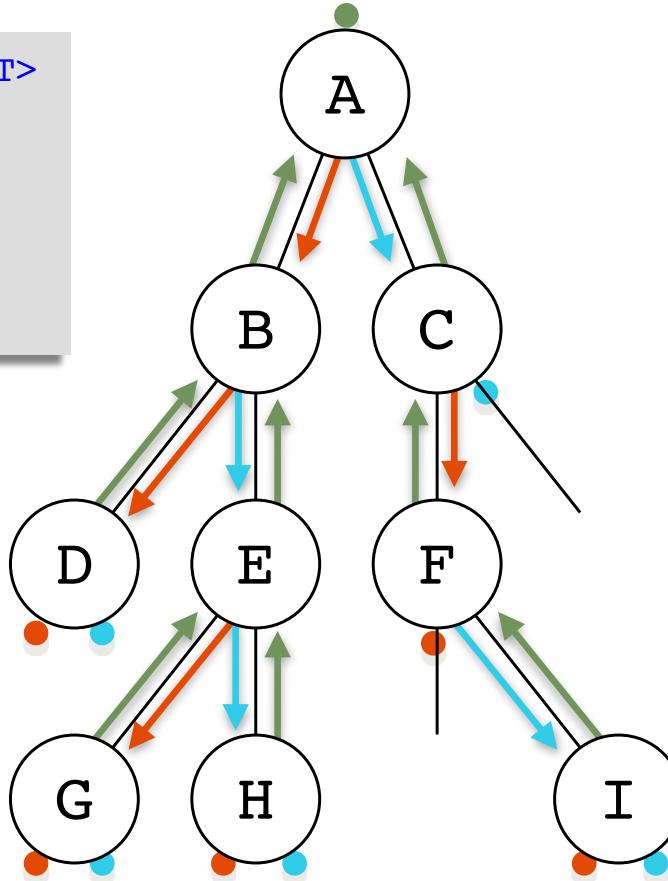




# parent, ainé, puiné dans un ABR

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Noeud* gauche
    Noeud* droit
```

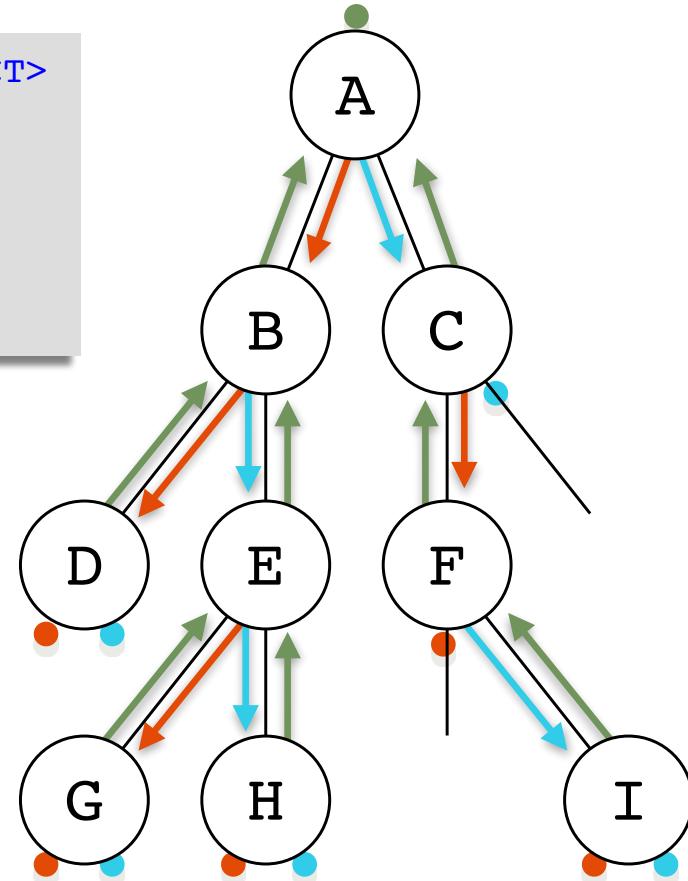
```
fonction parent (r)
    retourner r.parent
```





# parent, ainé, puiné dans un ABR

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Noeud* gauche
    Noeud* droit
```



```
fonction parent (r)
```

```
    retourner r.parent
```

```
fonction ainé (r)
```

```
    si r.gauche != Ø,
```

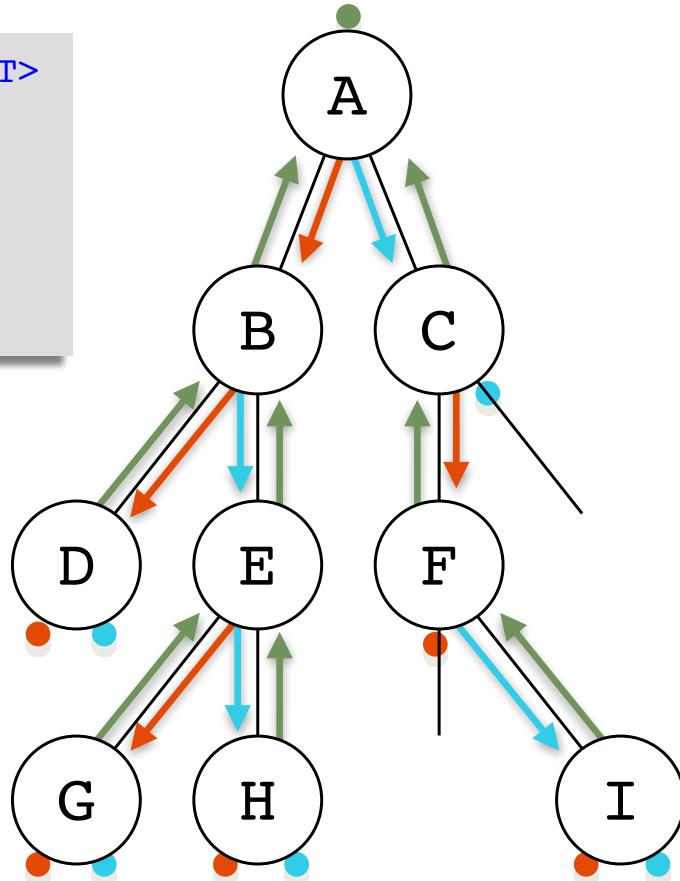
```
        retourner r.gauche
```

```
    sinon, retourner r.droit
```



# parent, ainé, puiné dans un ABR

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Noeud* gauche
    Noeud* droit
```



**fonction parent (r)**

retourner r.parent

**fonction ainé (r)**

si r.gauche !=  $\emptyset$ ,  
retourner r.gauche  
sinon, retourner r.droit

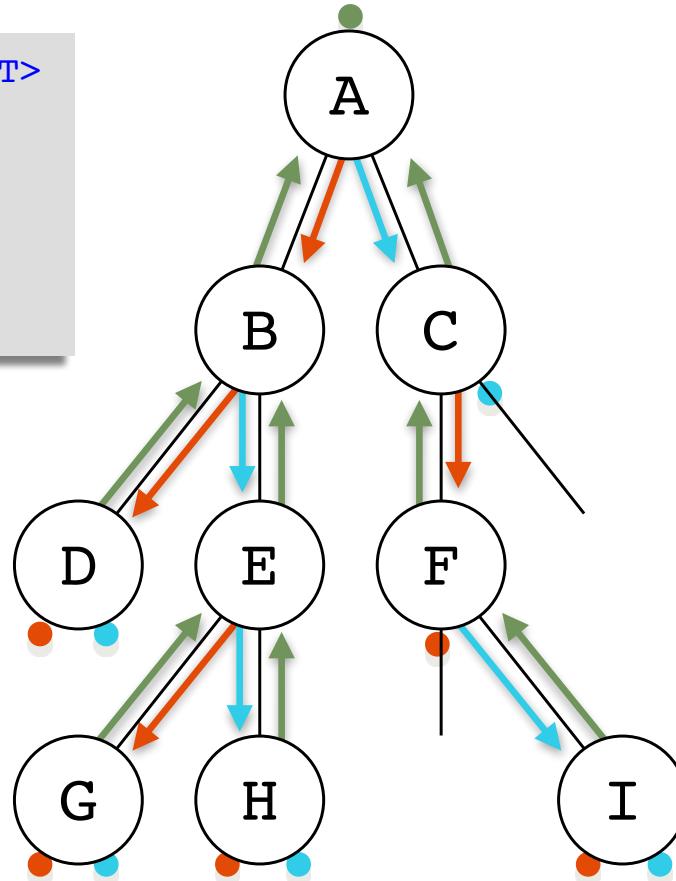
**fonction puiné (r)**

si est\_enfant\_gauche(r),  
retourner r.parent.droit  
sinon, retourner  $\emptyset$



# parent, ainé, puiné dans un ABR

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Noeud* gauche
    Noeud* droit
```



```
fonction parent (r)
```

```
    retourner r.parent
```

```
fonction ainé (r)
```

```
    si r.gauche != Ø,
        retourner r.gauche
    sinon, retourner r.droit
```

```
fonction puiné (r)
```

```
    si est_enfant_gauche(r),
        retourner r.parent.droit
    sinon, retourner Ø
```

```
fonction est_enfant_gauche (r)
```

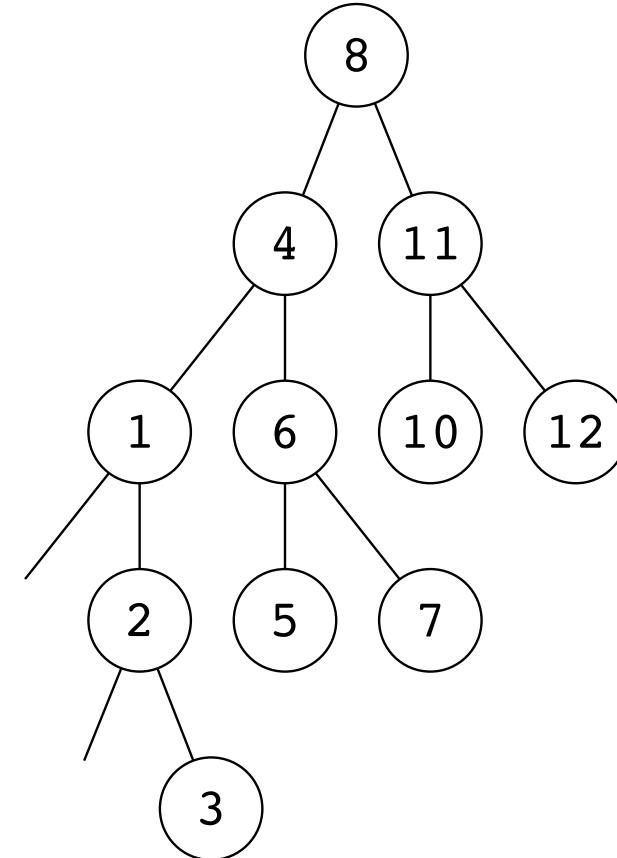
```
    p ← r.parent
```

```
    retourner p != Ø et r == p.gauche
```



# Itération en ordre symétrique

```
fonction symétrique (r, fn)
n ← début(r)
tant que n != fin(n)
    fn(n)
    n ← suivant(n)
```



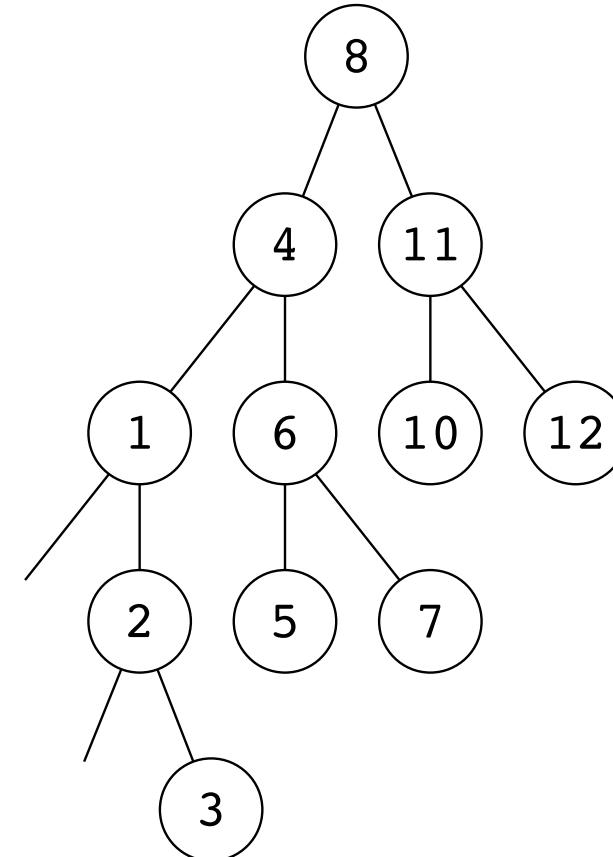


# Itération en ordre symétrique

```
fonction symétrique (r, fn)
n ← début(r)
tant que n != fin(n)
    fn(n)
    n ← suivant(n)
```

```
fonction début(r)
    retourner min(r)
```

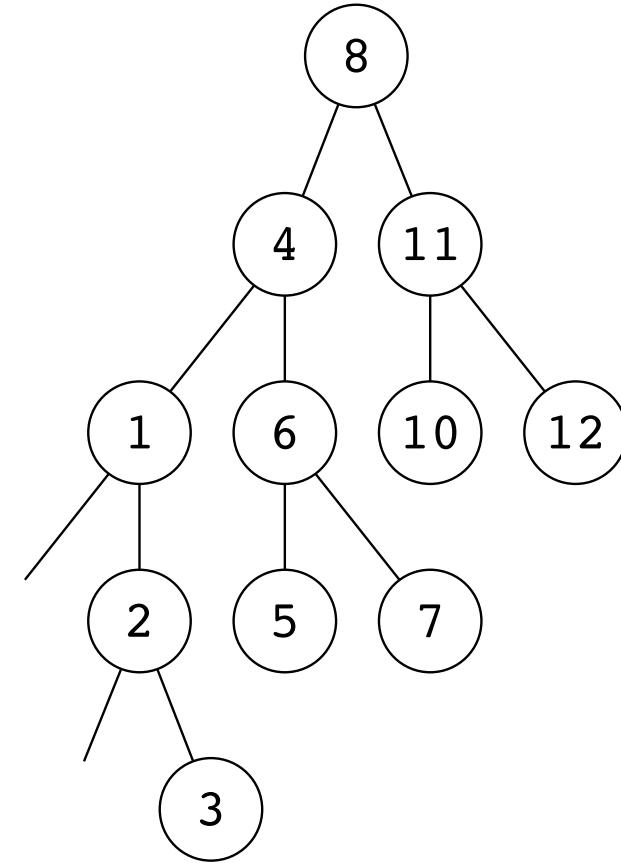
```
fonction fin(r)
    retourner ∅
```





# suivant en ordre symétrique

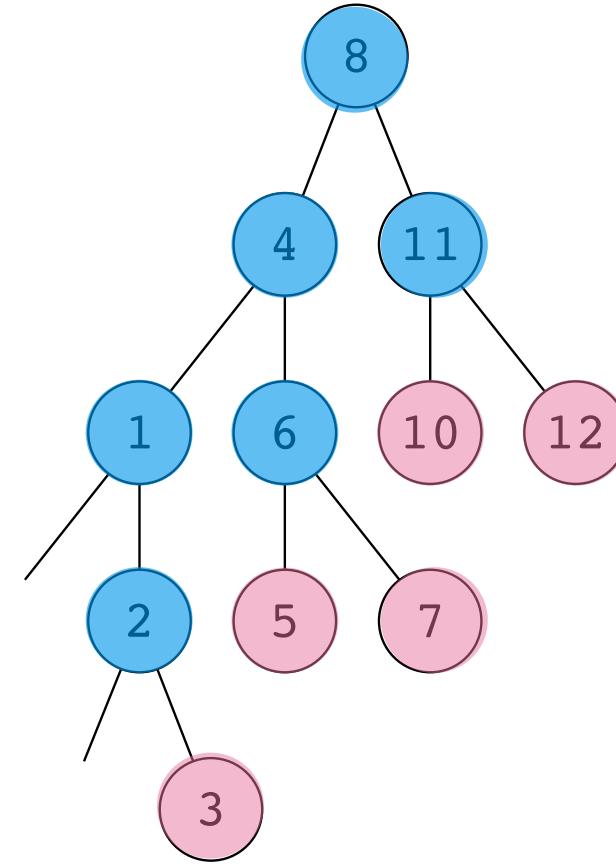
- $r$  a un sous-arbre droit
  - $\text{suivant}(r)$  est le minimum de ce sous-arbre
- $r$  n'a pas de sous-arbre droit
  - $\text{suivant}(r)$  est le premier ancêtre donc  $r$  appartient au sous-arbre gauche





# suivant en ordre symétrique

- $r$  a un sous-arbre droit
  - $\text{suivant}(r)$  est le minimum de ce sous-arbre
- $r$  n'a pas de sous-arbre droit
  - $\text{suivant}(r)$  est le premier ancêtre donc  $r$  appartient au sous-arbre gauche

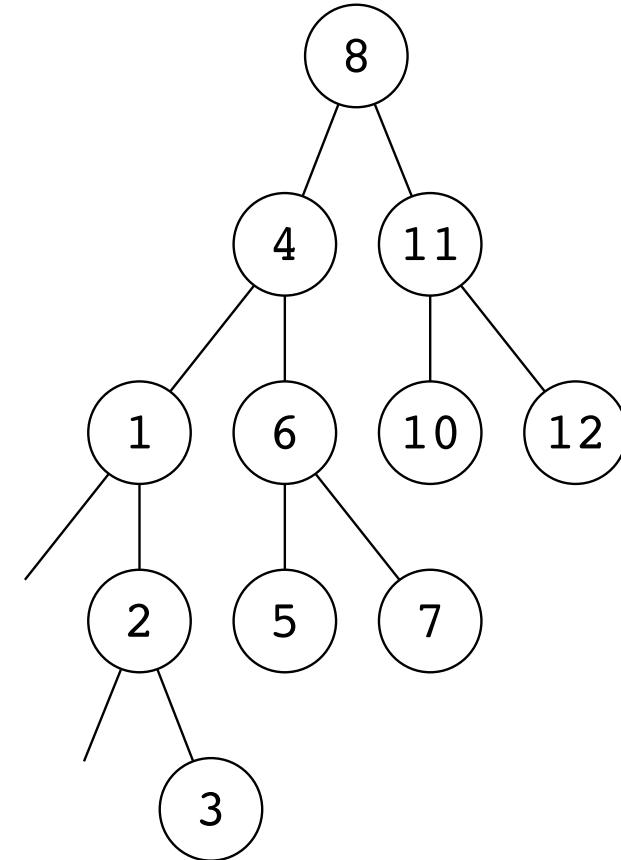




# suivant en ordre symétrique

```
fonction suivant(r)
    si r.droit != Ø
        retourner début(r.droit)
    sinon
        n ← r
        tant que est_enfant_droit(n),
            n ← n.parent
    retourner n.parent
```

```
fonction est_enfant_droit (r)
    p ← r.parent
    retourner p != Ø et r == p.droit
```

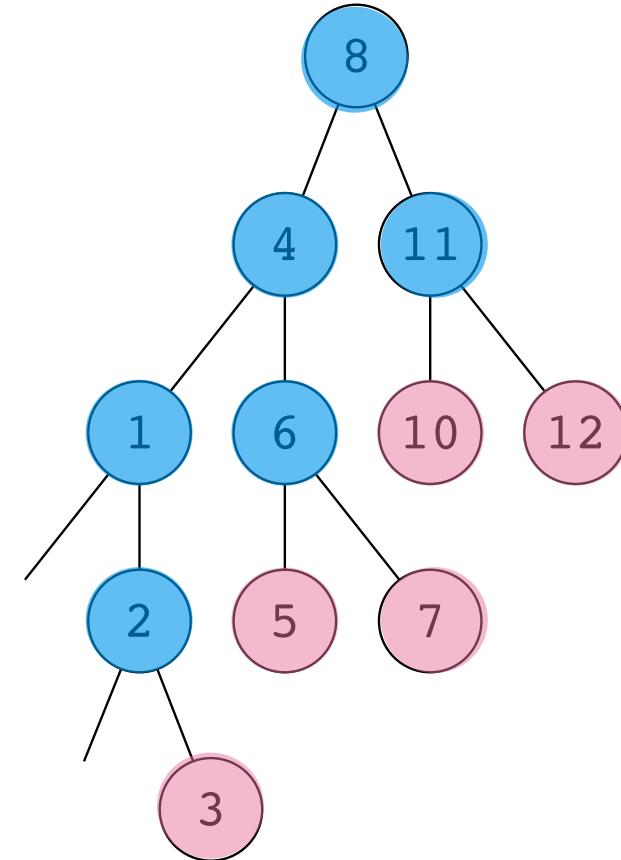




# suivant en ordre symétrique

```
fonction suivant(r)
    si r.droit != Ø
        retourner début(r.droit)
    sinon
        n ← r
        tant que est_enfant_droit(n),
            n ← n.parent
        retourner n.parent
```

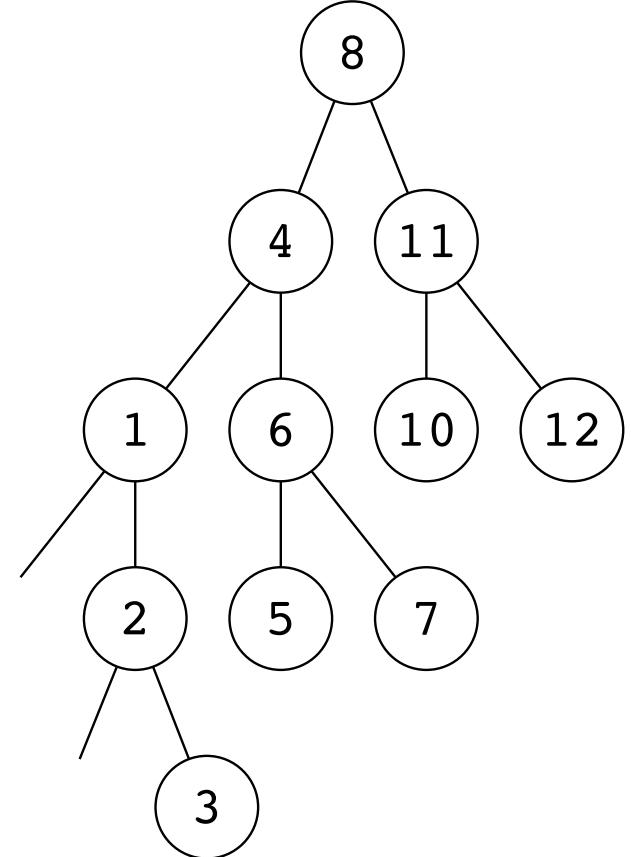
```
fonction est_enfant_droit (r)
    p ← r.parent
    retourner p != Ø et r == p.droit
```





# Trace des 3 itérations

- Itération en profondeur
- Pré-ordre
- Post-ordre
- Symétrique





# Trace des 3 itérations

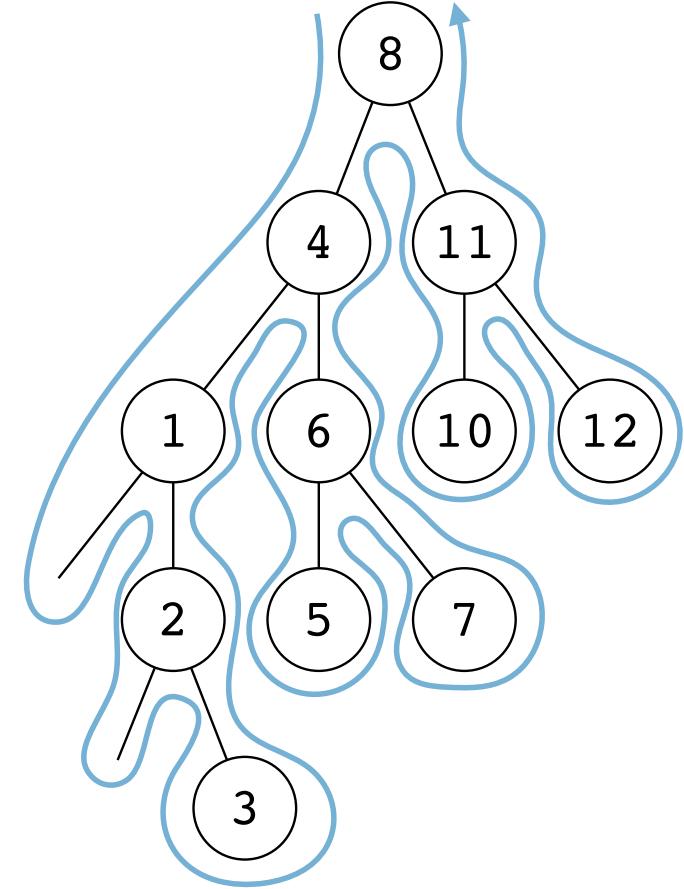
- Itération en profondeur

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Pré-ordre

- Post-ordre

- Symétrique





# Trace des 3 itérations

- Itération en profondeur

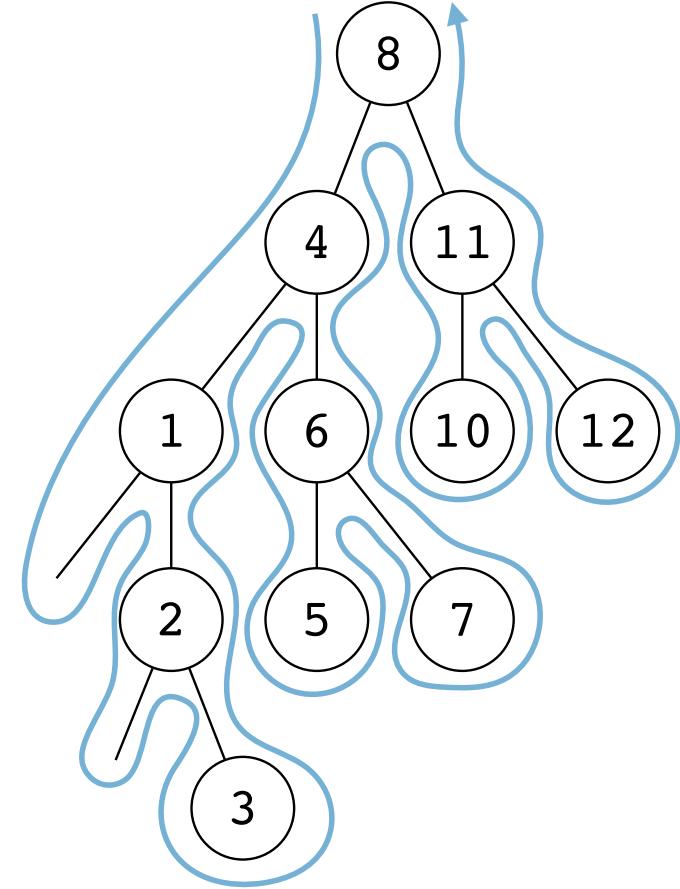
8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Pré-ordre

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Post-ordre

- Symétrique





# Trace des 3 itérations

- Itération en profondeur

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

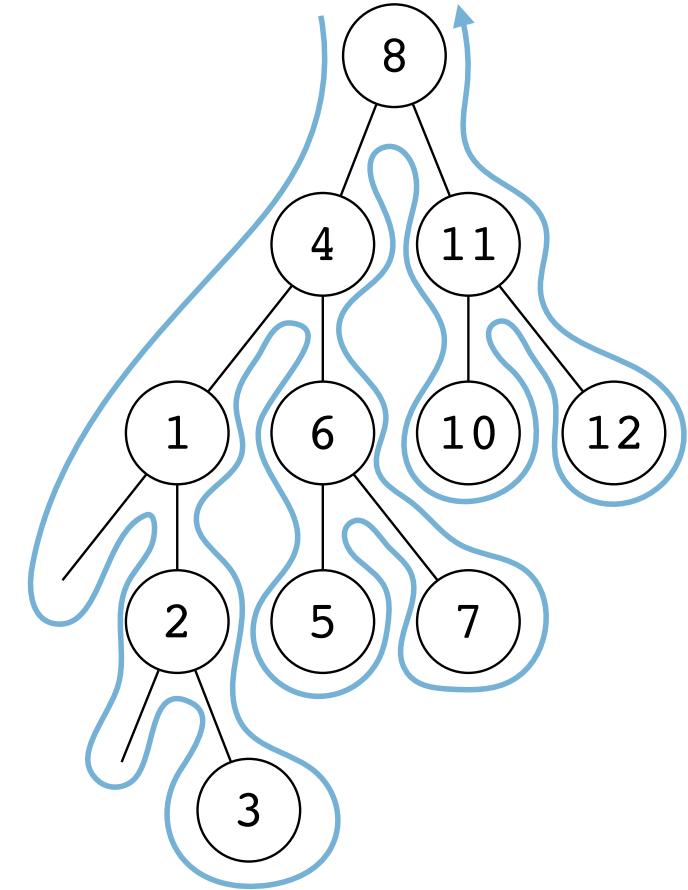
- Pré-ordre

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Post-ordre

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Symétrique





# Trace des 3 itérations

- Itération en profondeur

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Pré-ordre

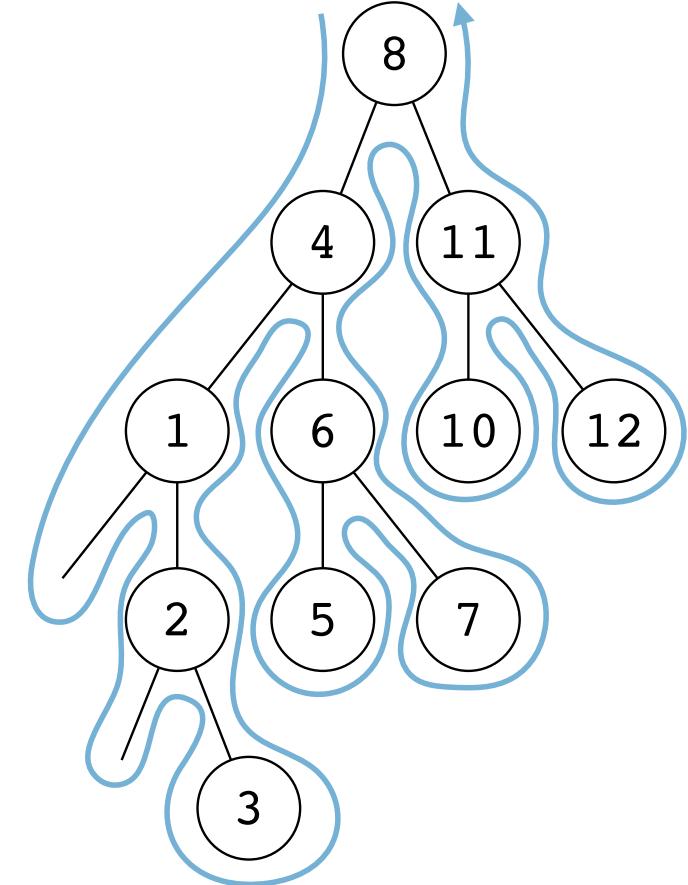
8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Post-ordre

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Symétrique

8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅



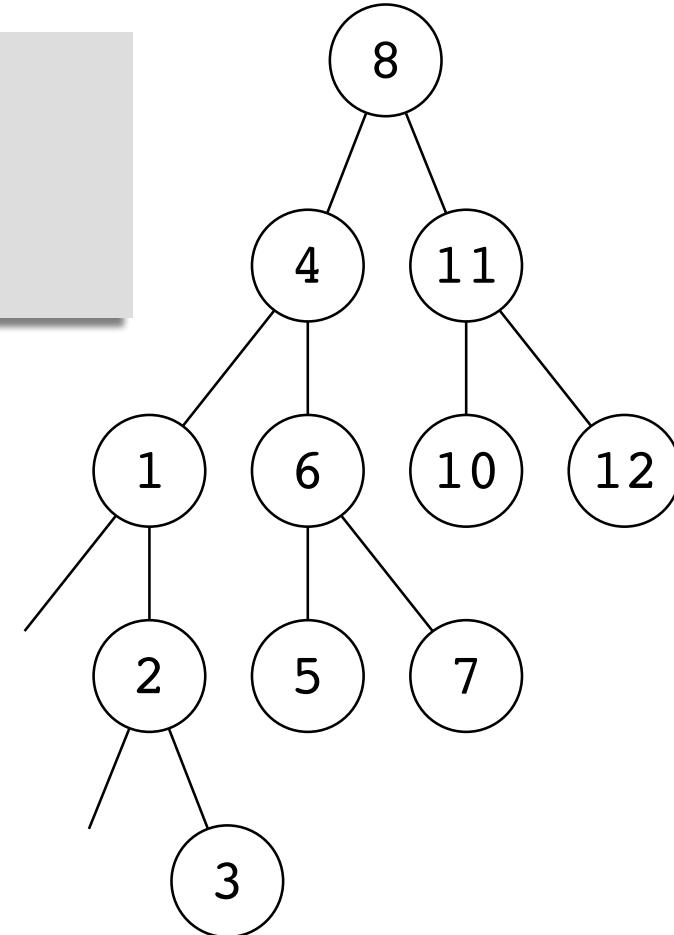
## 12. Taille, rang, nième élément





# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
```

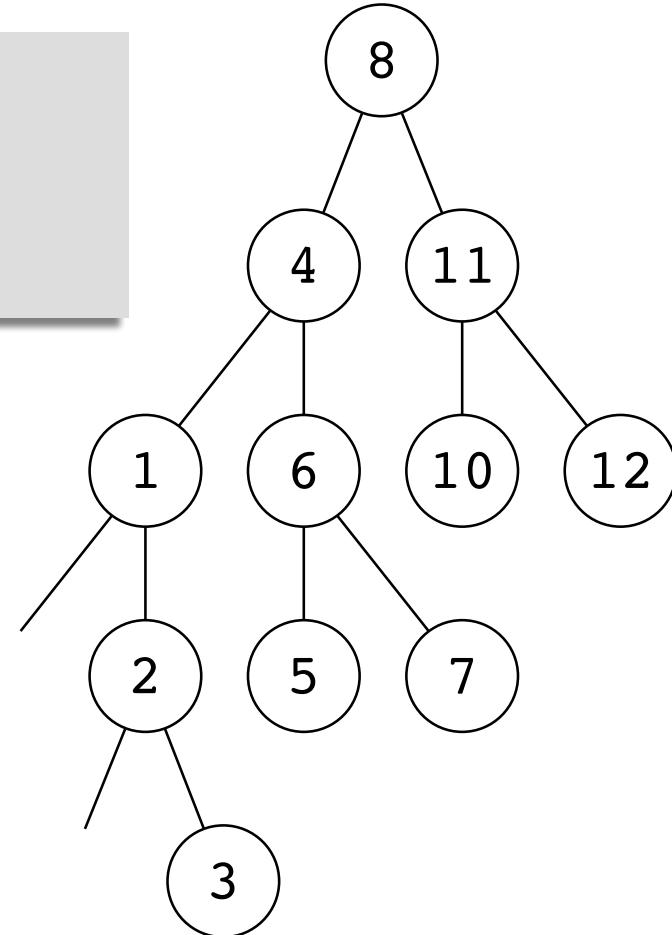




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
```

- Calcul récursif

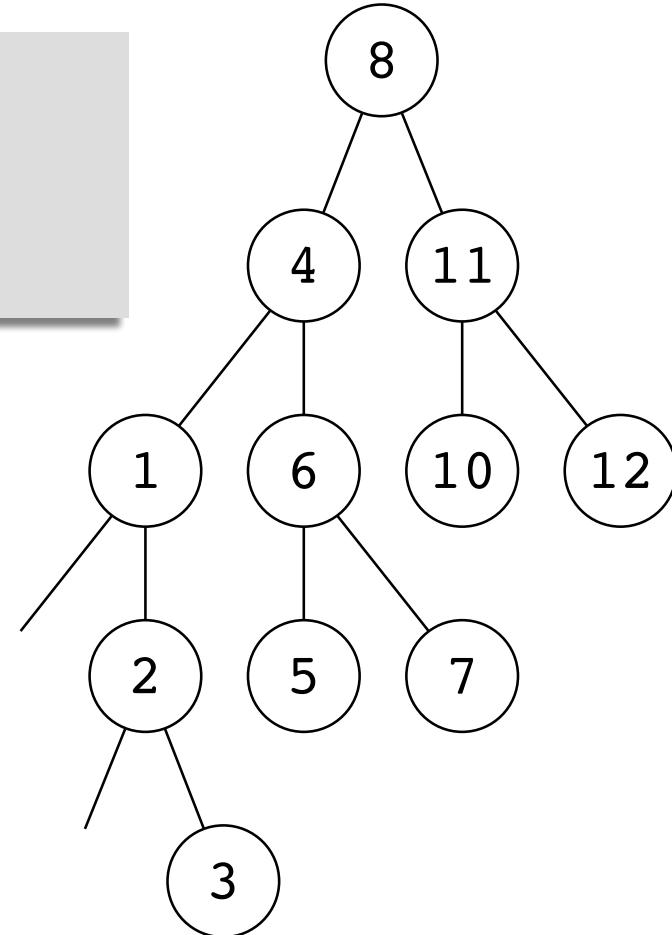




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
```

- Calcul récursif
  - Cas trivial : arbre vide

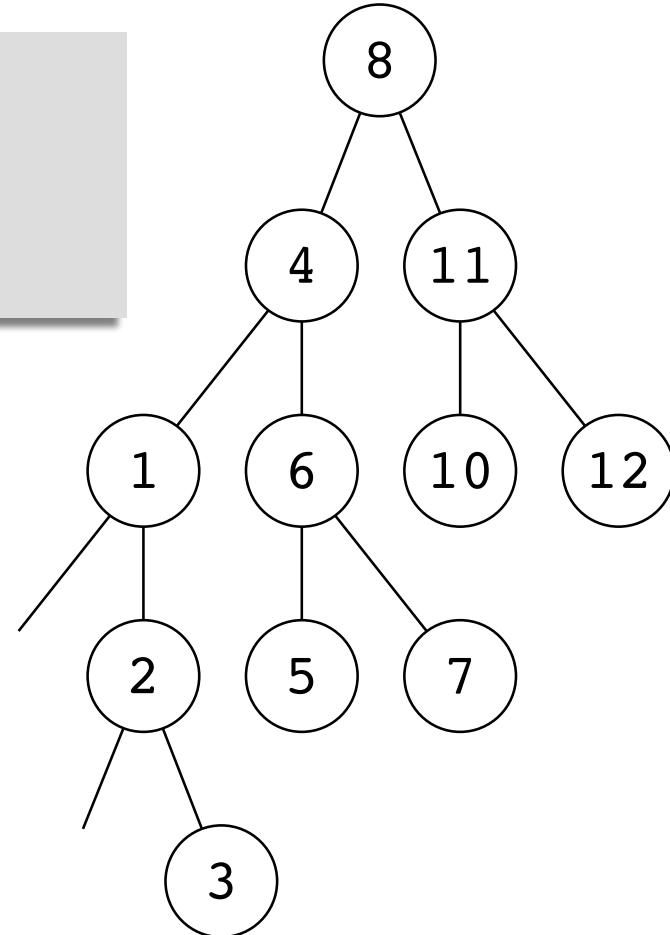




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
```

- Calcul récursif
  - Cas trivial : arbre vide

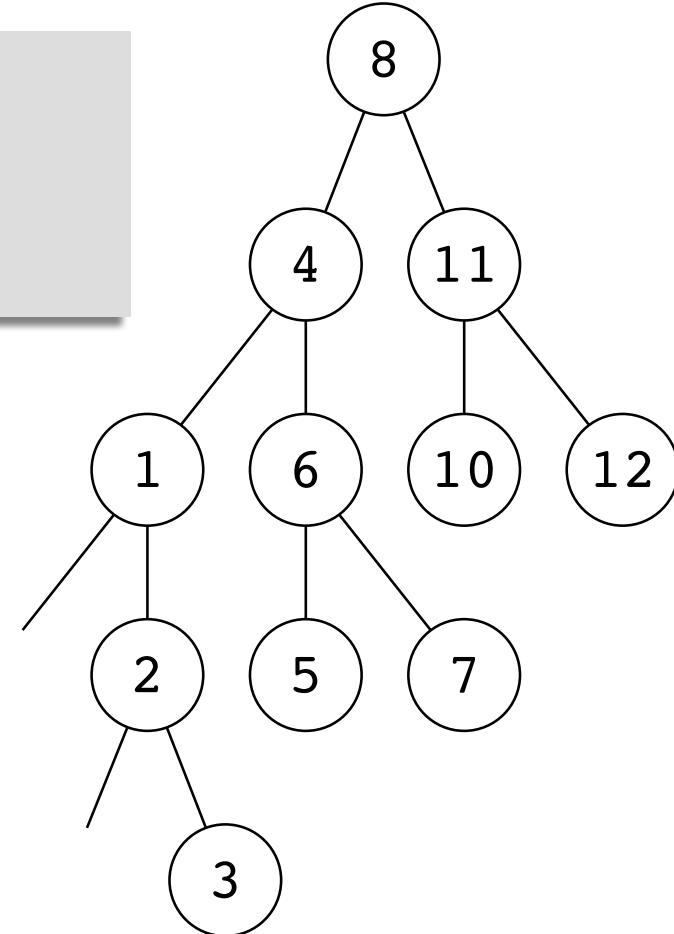




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

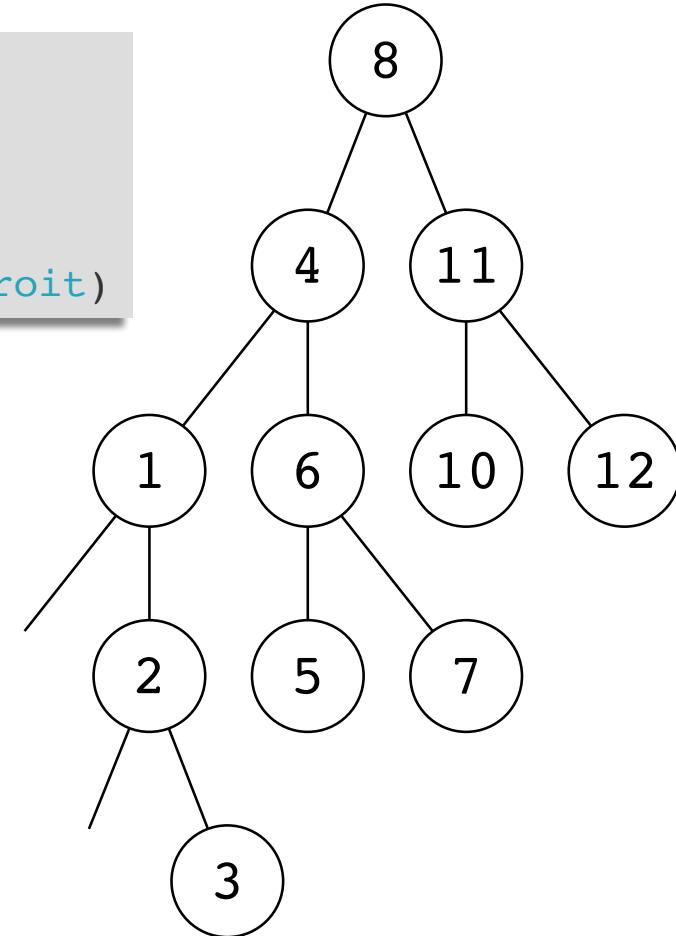




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

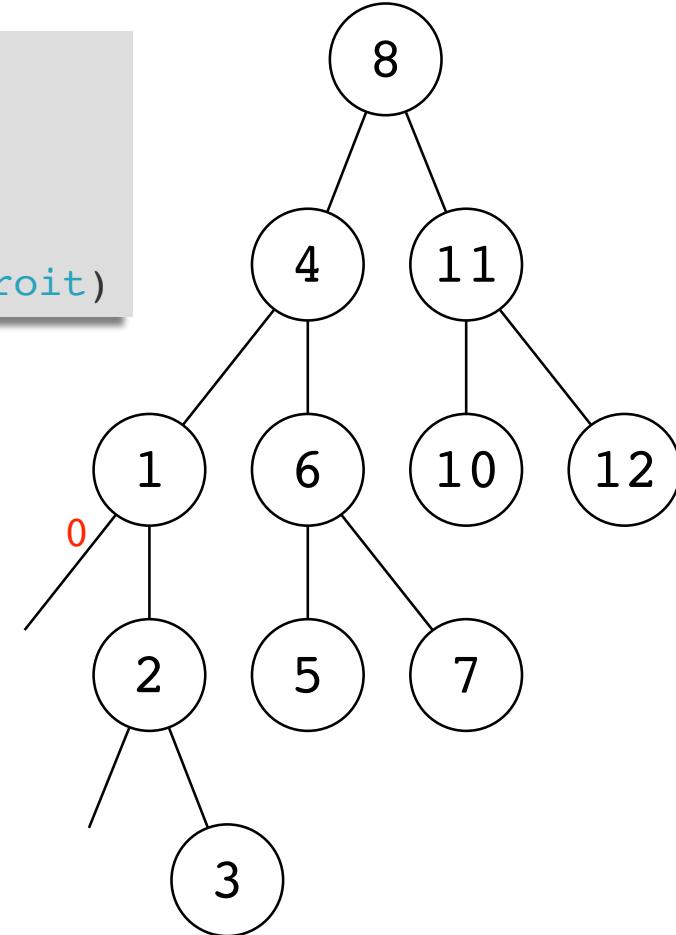




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

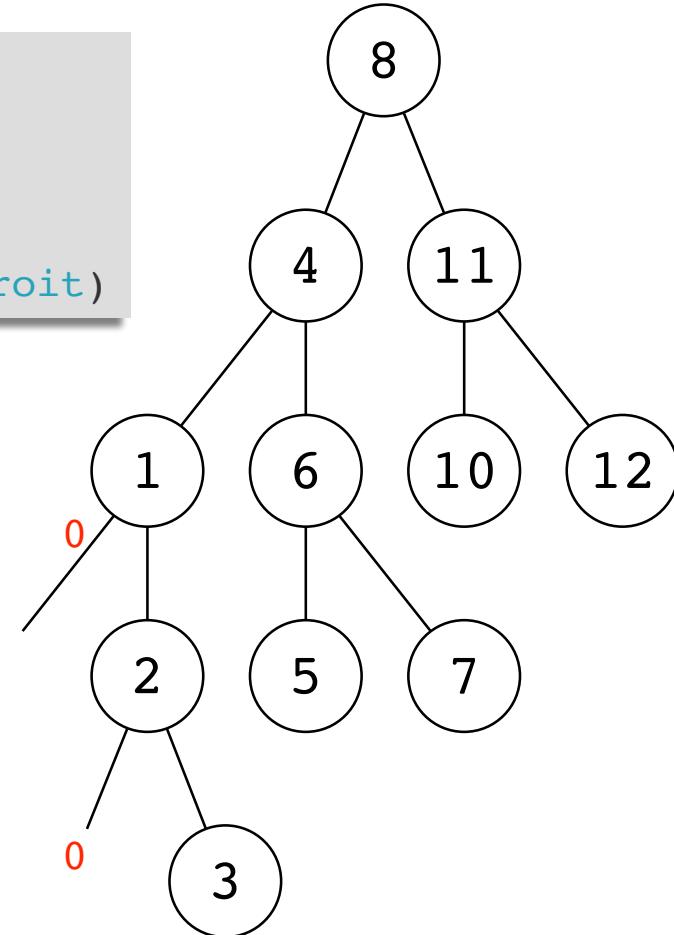




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

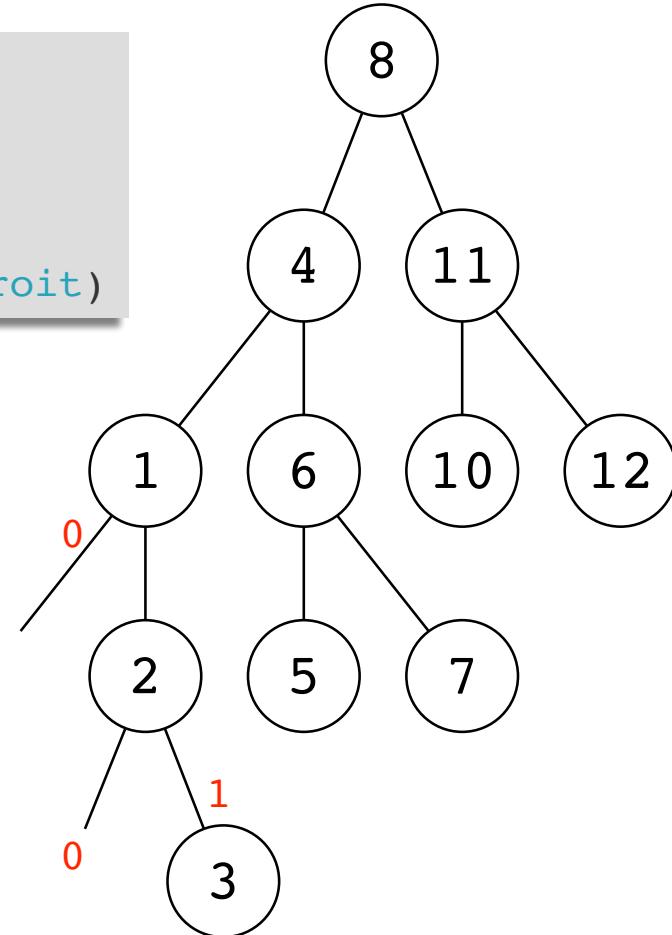




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

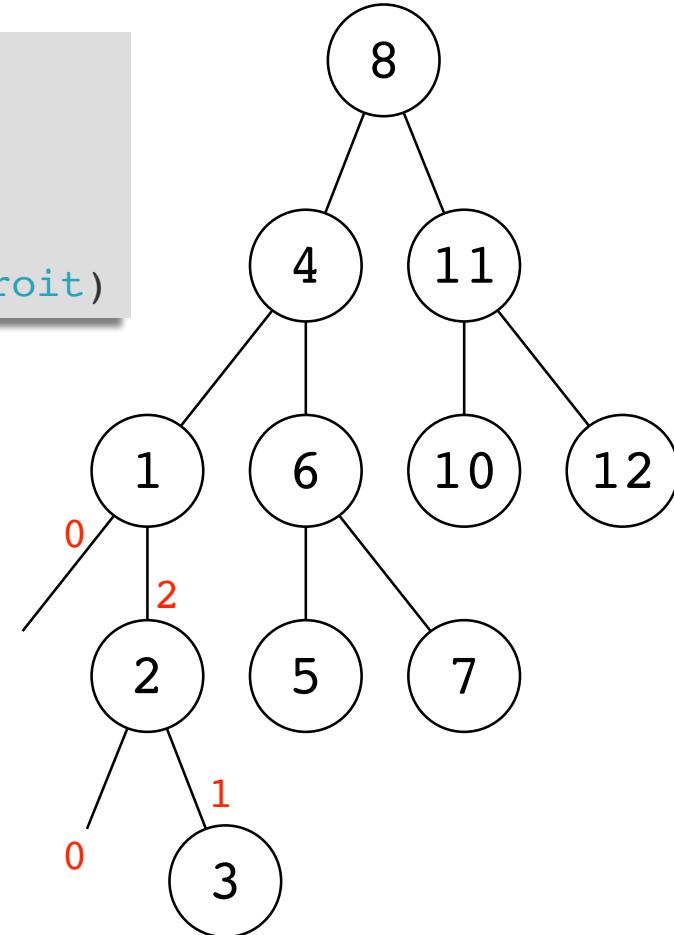




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

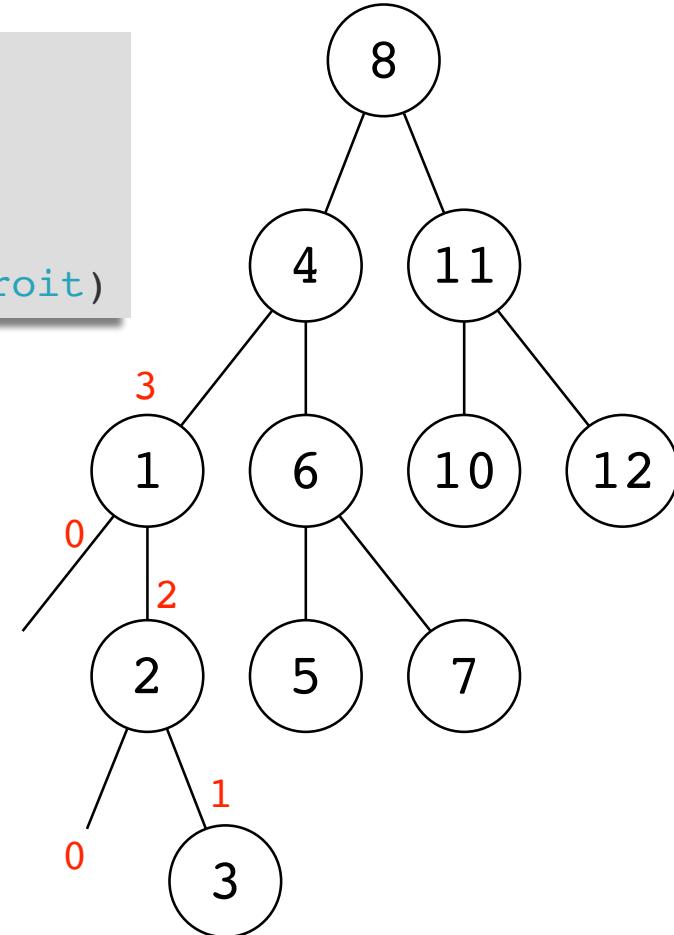




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

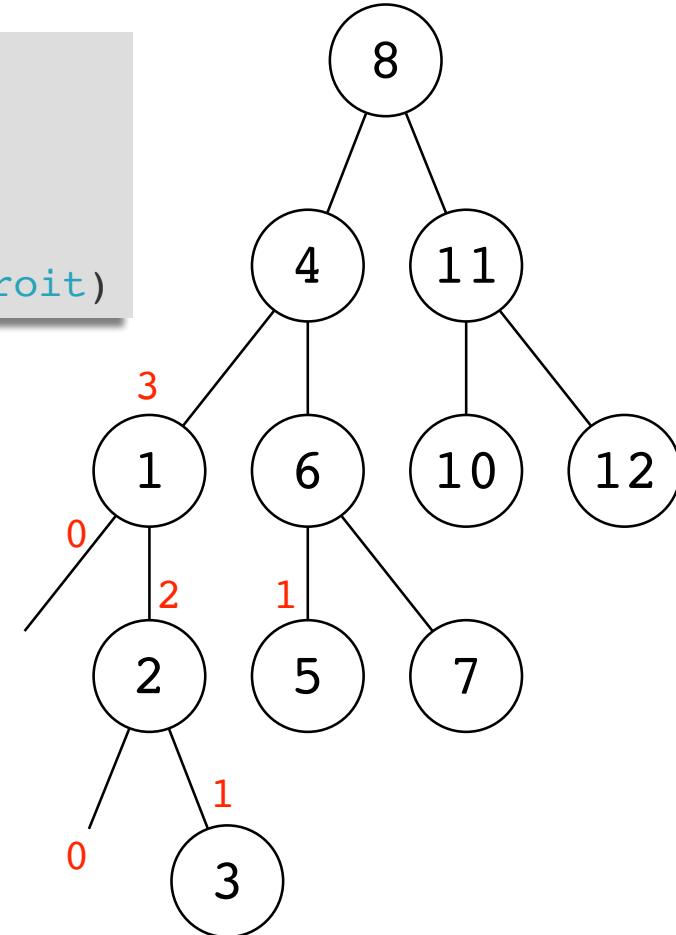




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

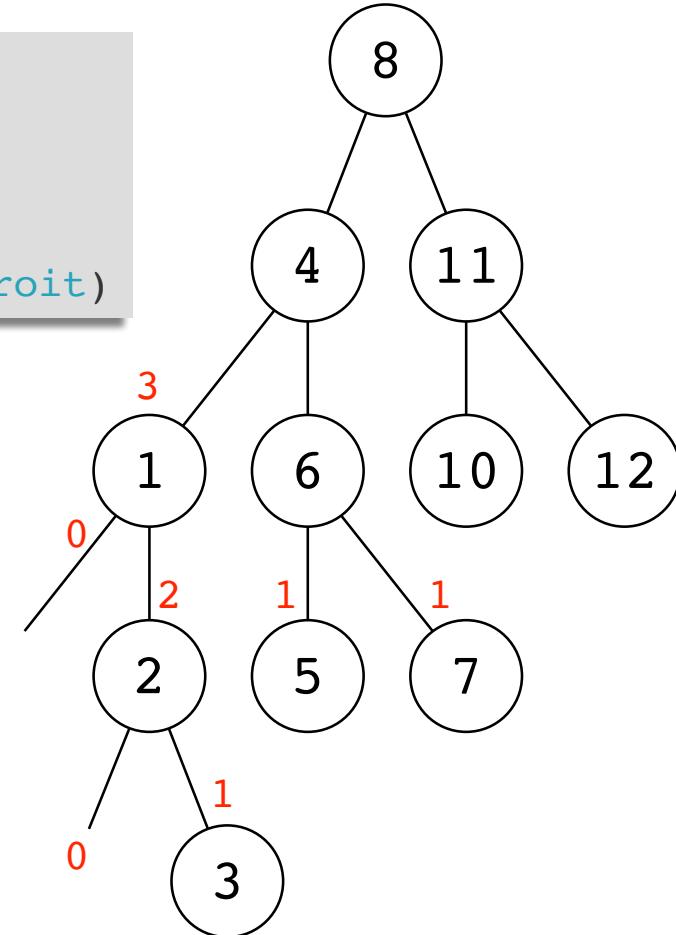




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

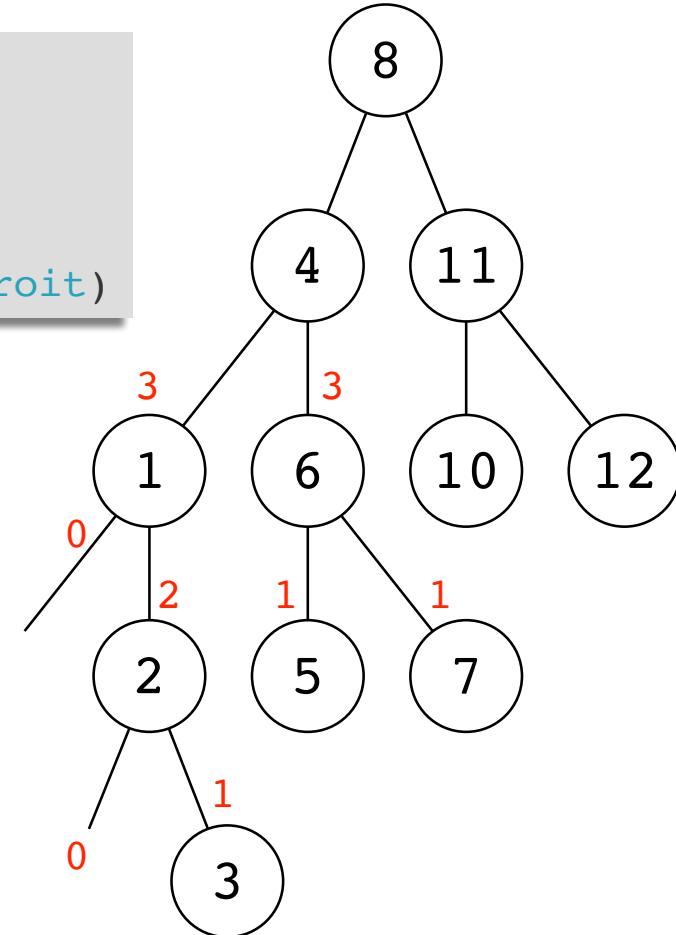




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

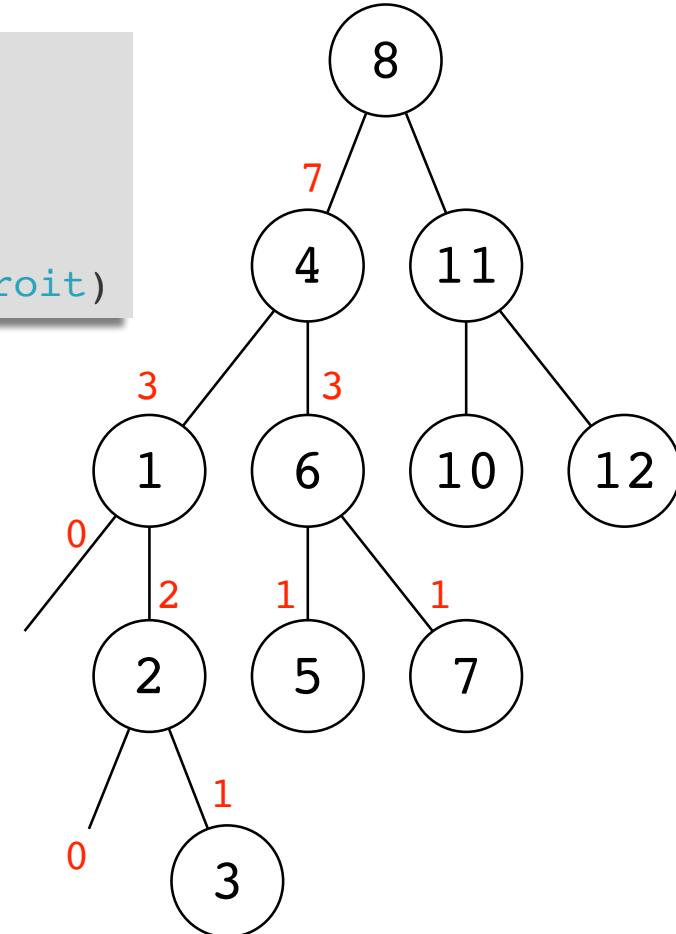




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

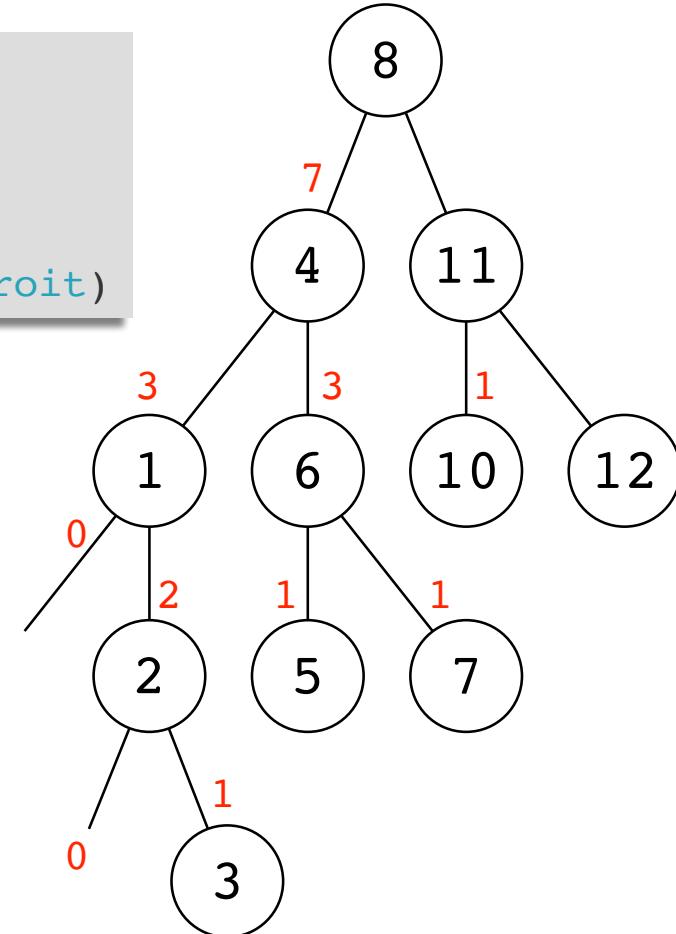




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

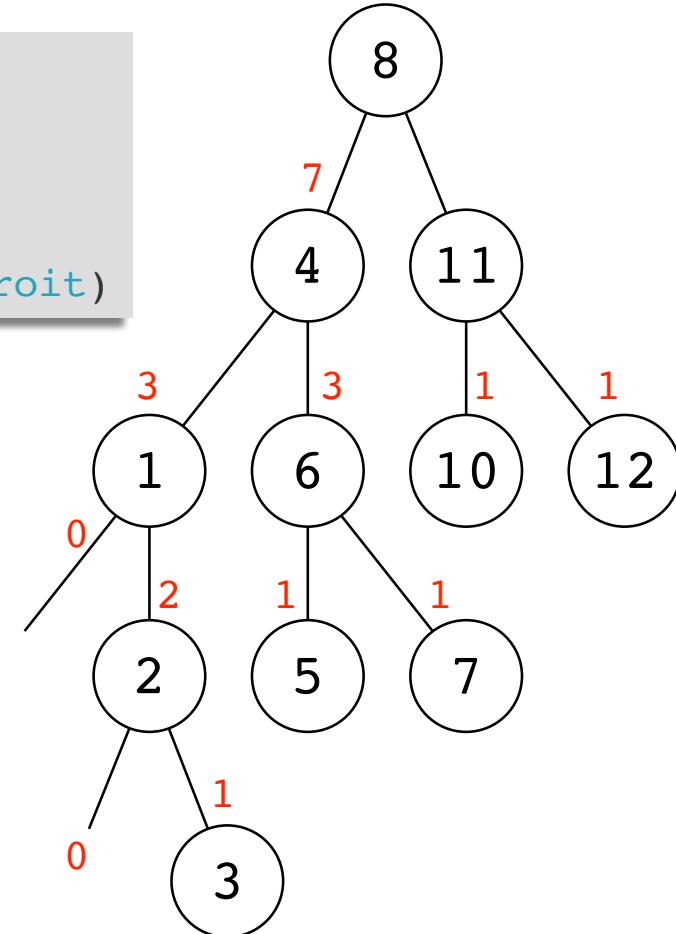




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

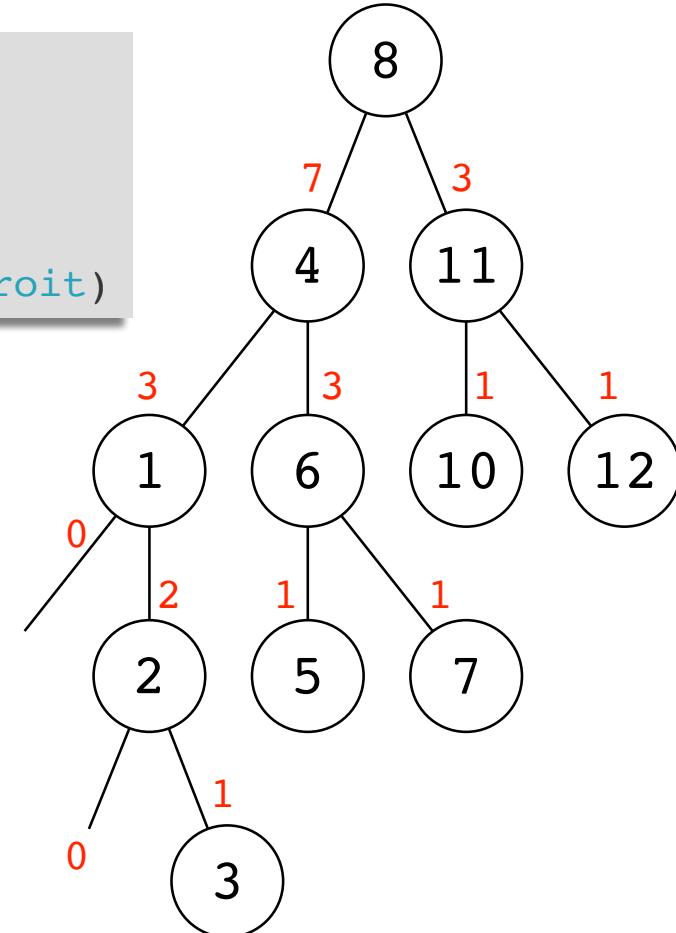




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine

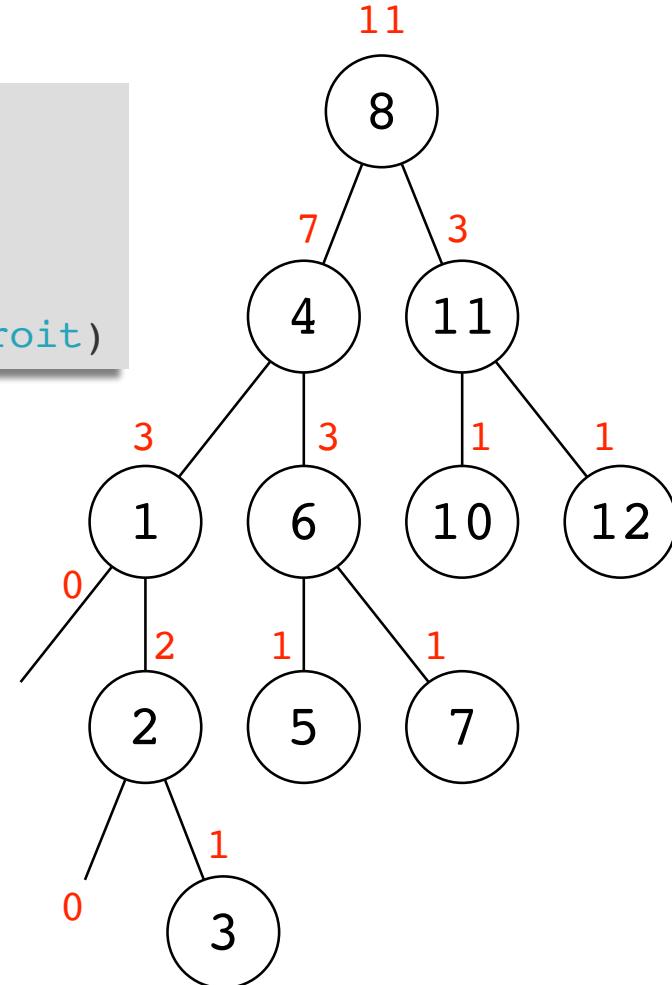




# Taille = nombre de noeuds d'un arbre

```
fonction taille (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

- Calcul récursif
  - Cas trivial : arbre vide
  - Cas général : somme des tailles des sous-arbres + 1 pour la racine





# Taille stockée

- L'approche récursive a une complexité  $O(n)$
- Parfois, on voudrait  $O(1)$



# Taille stockée

- L'approche récursive a une complexité  $O(n)$
- Parfois, on voudrait  $O(1)$ 
  - Stocker la taille dans les noeuds

```
structure Noeud<K, V>
```

```
K clé
```

```
Noeud* gauche
```

```
Noeud* droit
```

```
Entier taille
```



# Taille stockée

- L'approche récursive a une complexité  $O(n)$
- Parfois, on voudrait  $O(1)$ 
  - Stocker la taille dans les noeuds
  - Ecrire une fonction taille robuste

```
structure Noeud<K, V>
    K clé
    Noeud* gauche
    Noeud* droit
    Entier taille
```

```
fonction taille (r)
    si r == Ø, retourner 0
    sinon,      retourner r.taille
```



# Taille stockée

- L'approche récursive a une complexité  $O(n)$
- Parfois, on voudrait  $O(1)$ 
  - Stocker la taille dans les noeuds
  - Ecrire une fonction taille robuste
  - Maintenir à jour l'attribut taille dans les modificateurs en retour de récursion

```
structure Noeud<K, V>
```

```
K clé
```

```
Noeud* gauche
```

```
Noeud* droit
```

```
Entier taille
```

```
fonction taille (r)
```

```
si r == Ø, retourner 0
```

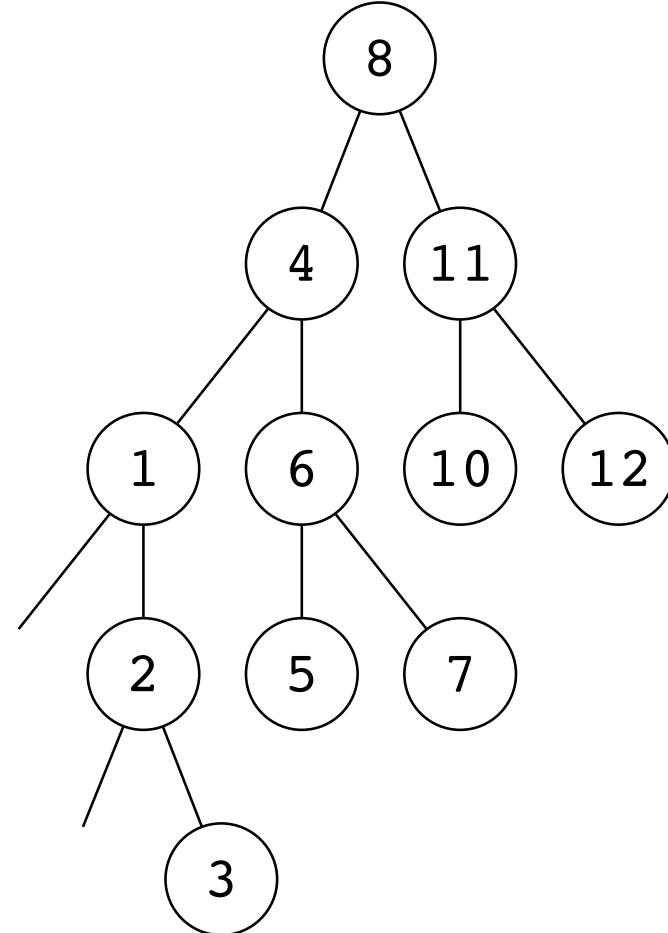
```
sinon,      retourner r.taille
```

```
r.taille ← taille(r.gauche)  
          + taille(r.droit)  
          + 1
```



# Rang d'une clé k (1)

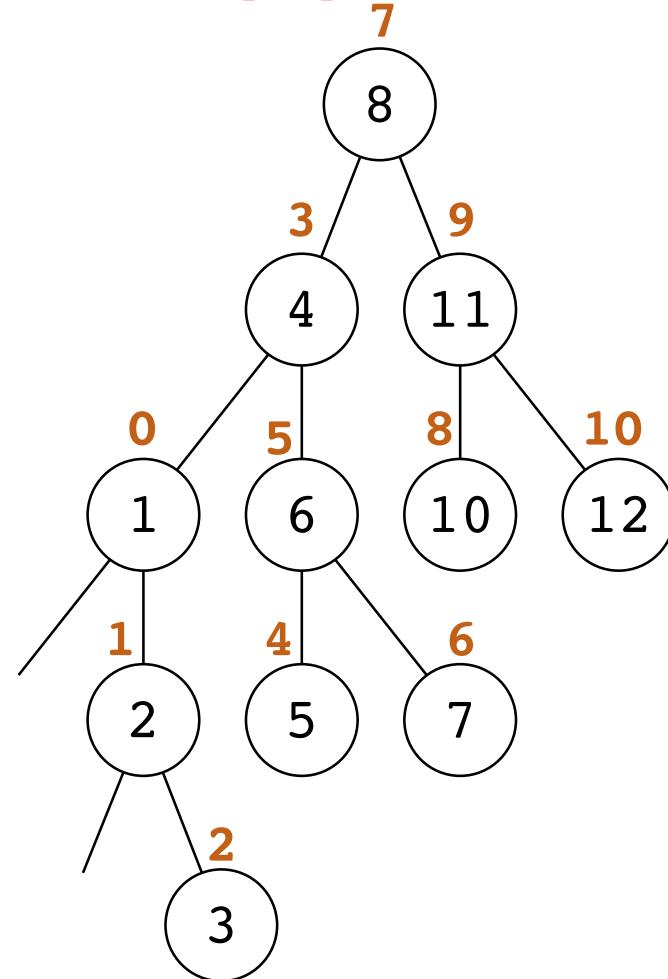
- Nombre d'éléments strictement plus petits que la clé





# Rang d'une clé k (1)

- Nombre d'éléments strictement plus petits que la clé

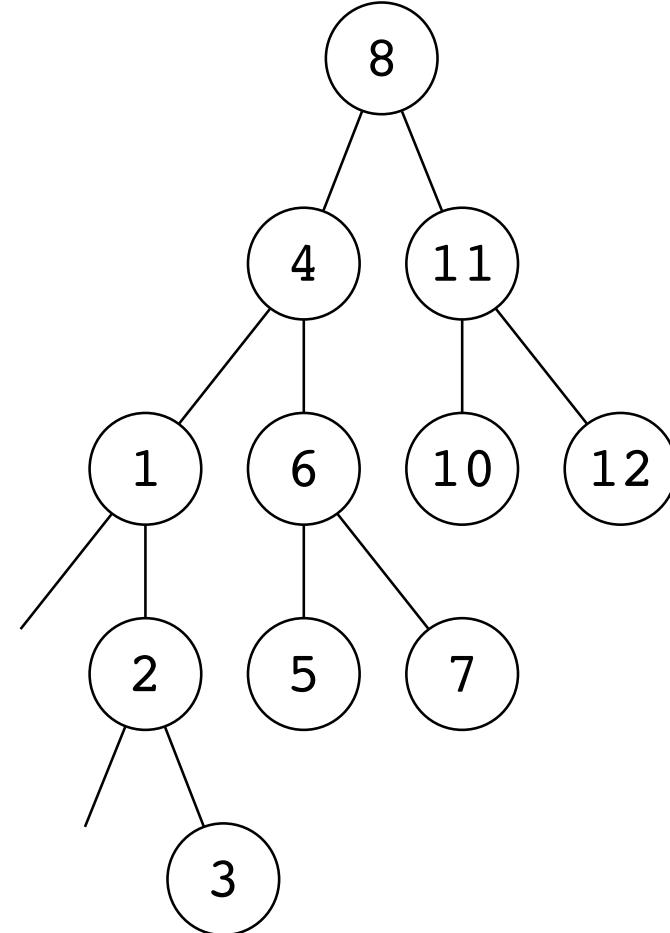




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
```

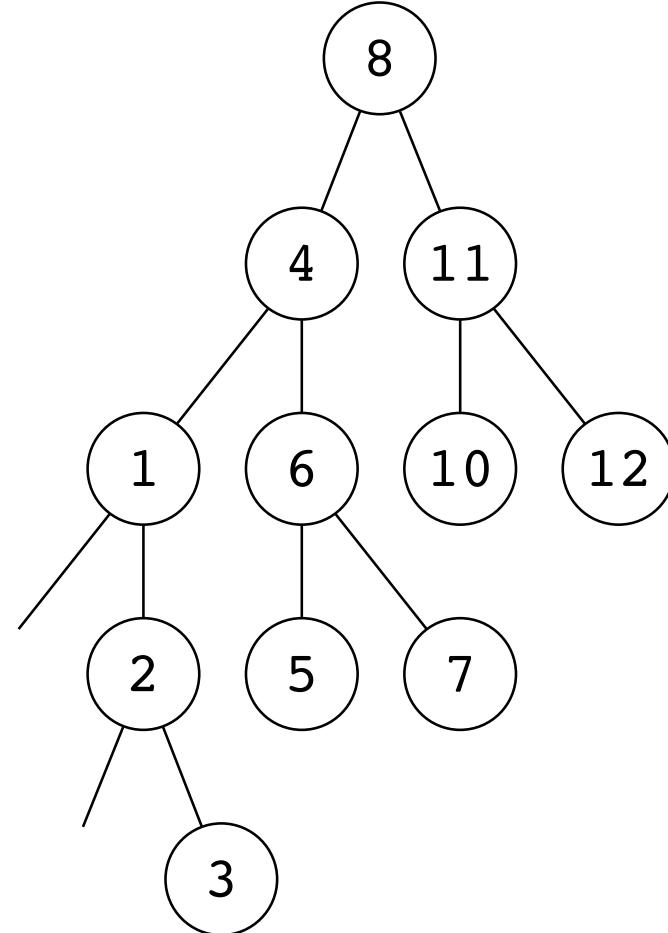




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
```

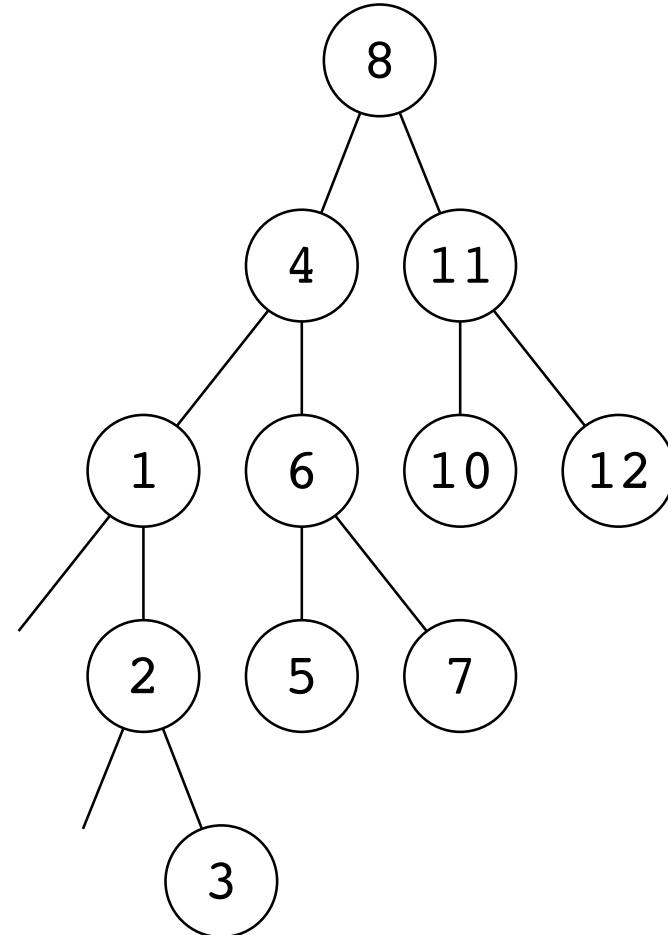




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

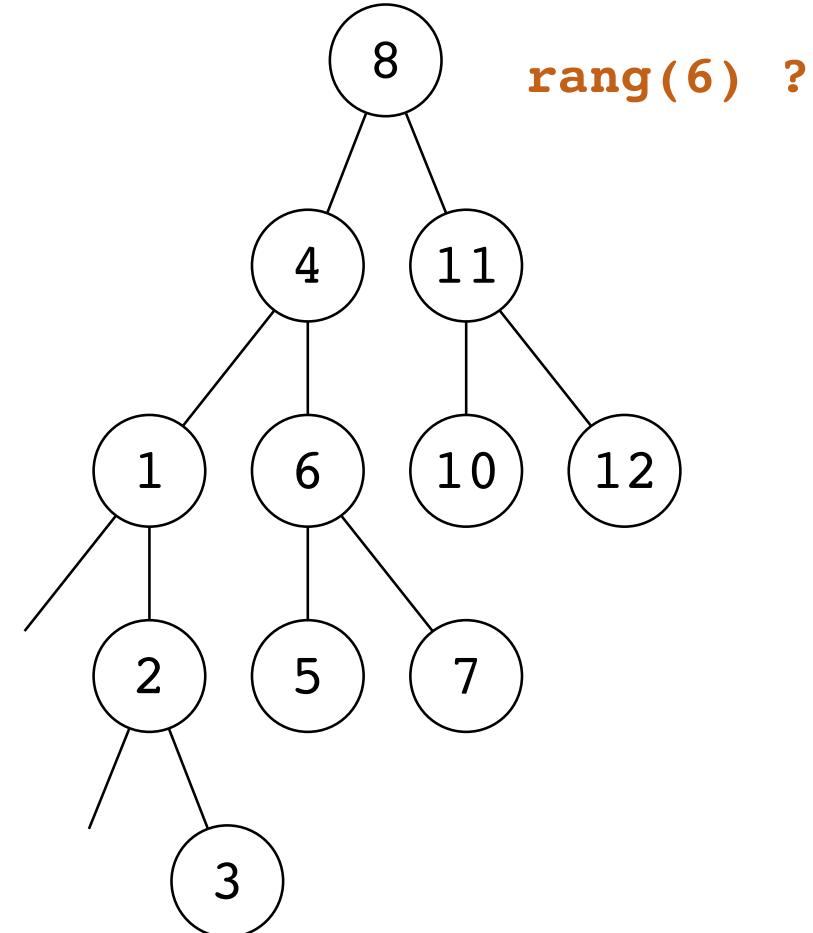




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

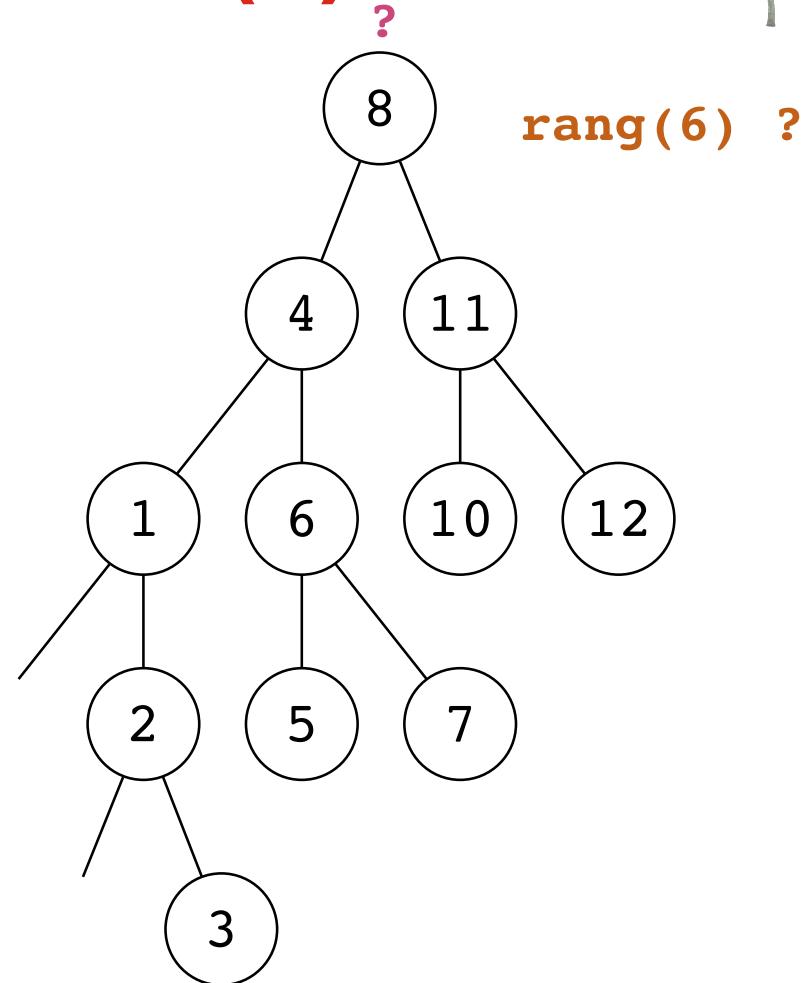




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

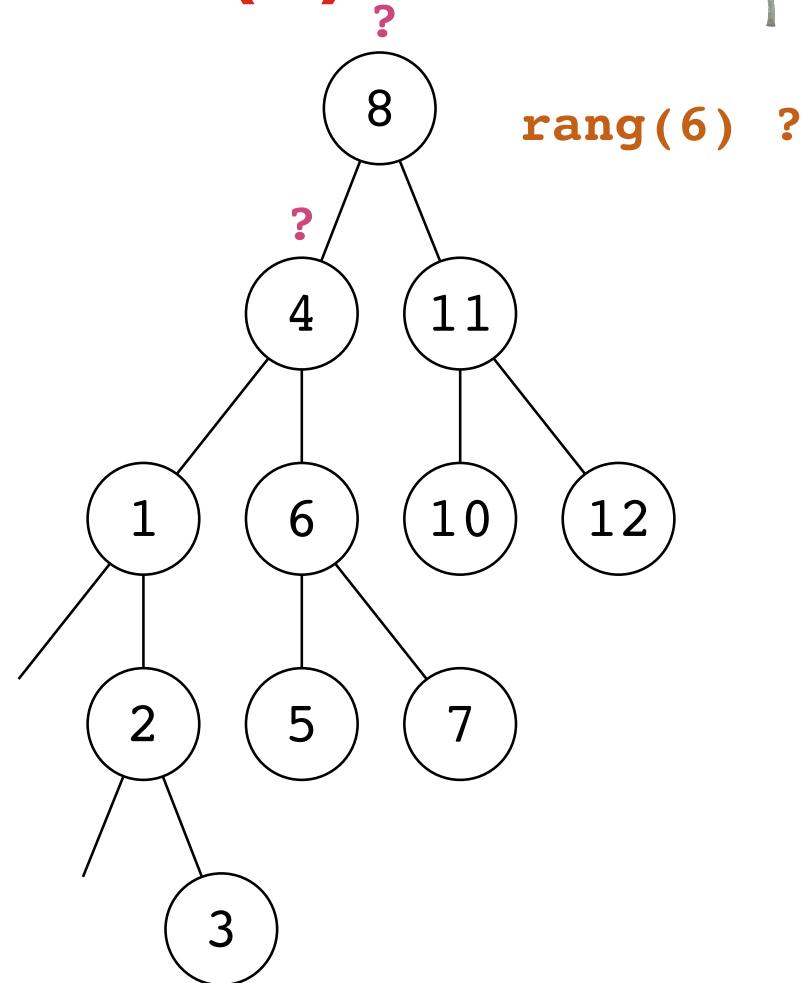




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

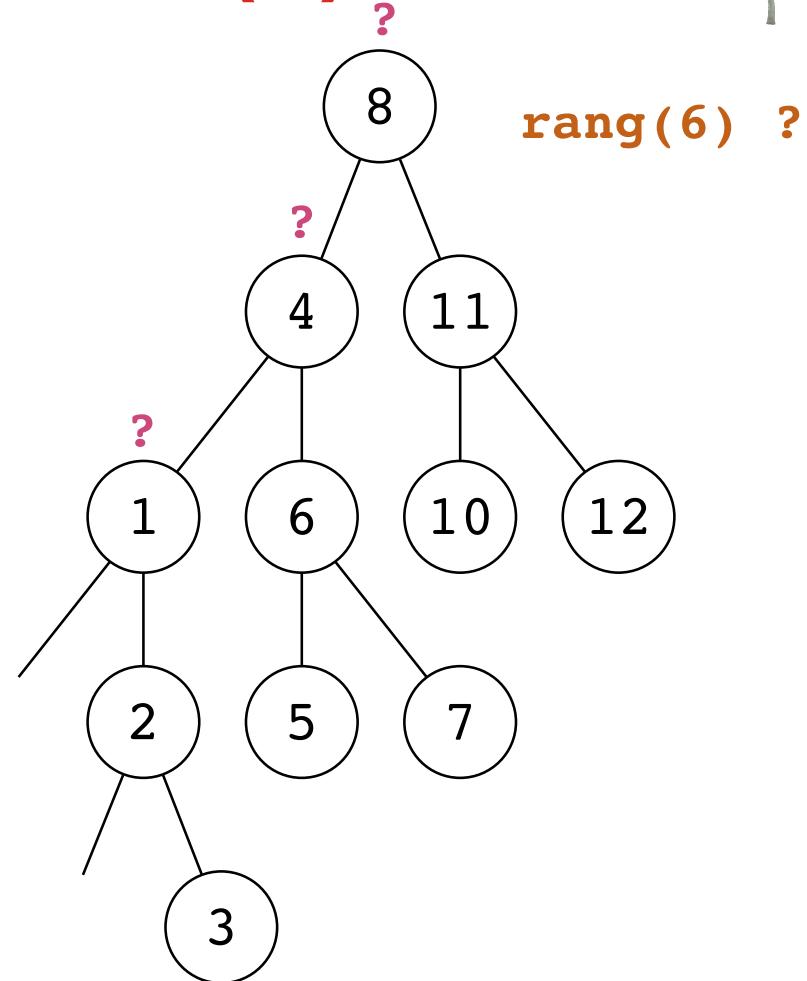




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

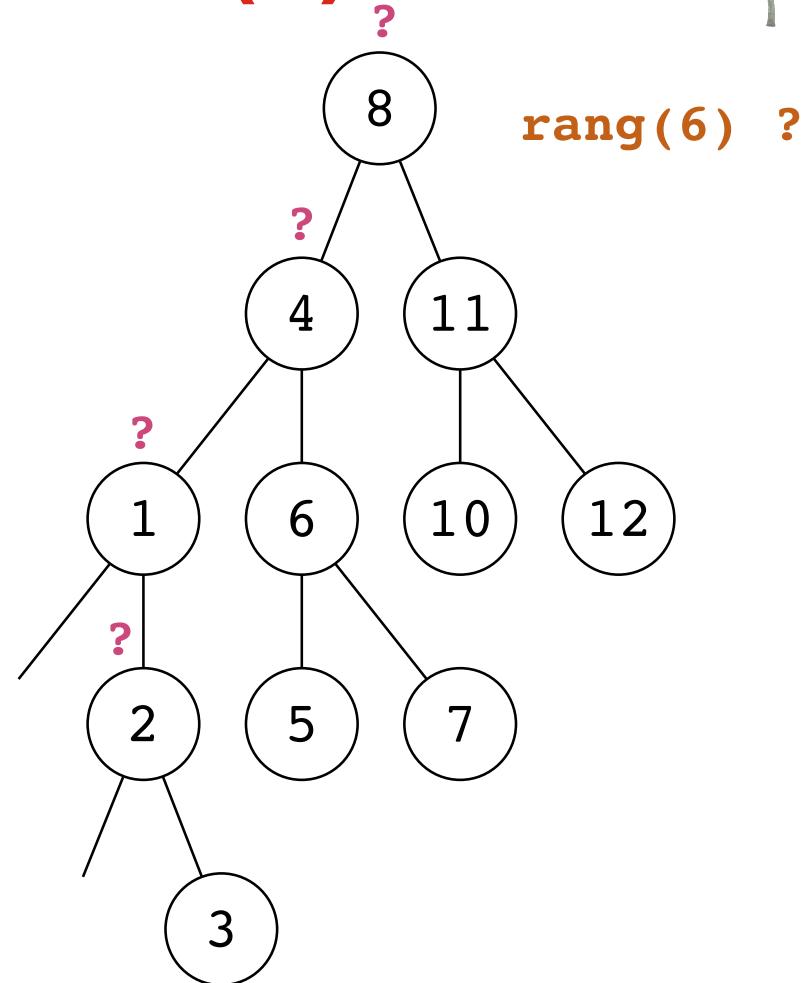




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

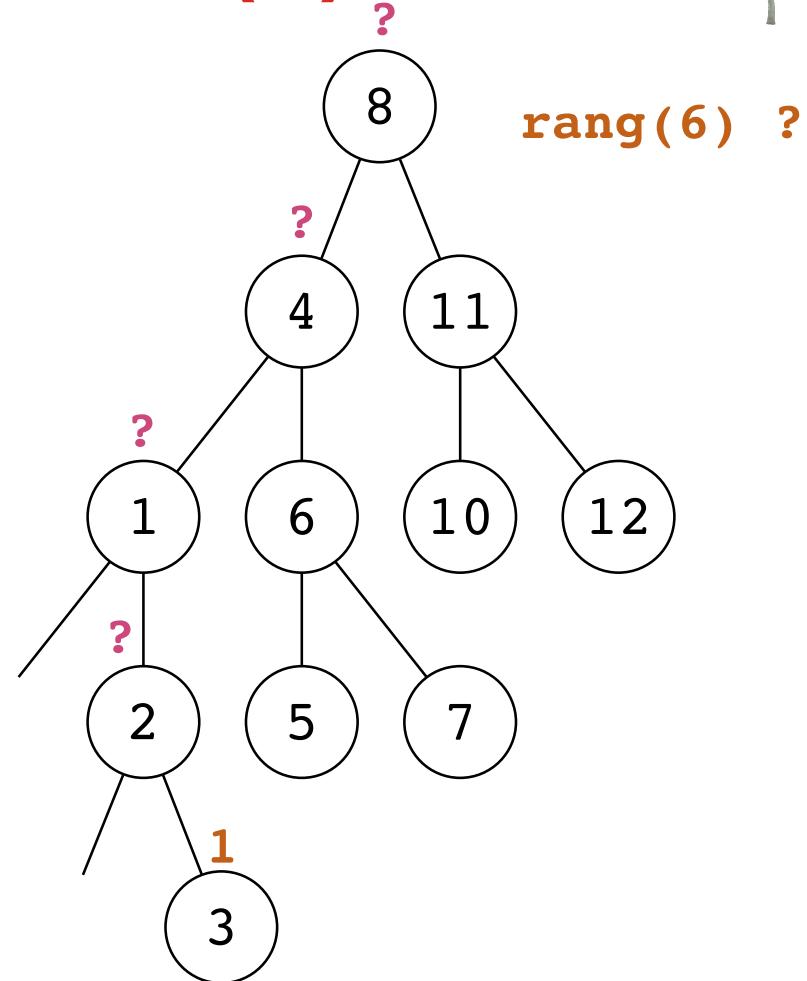




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

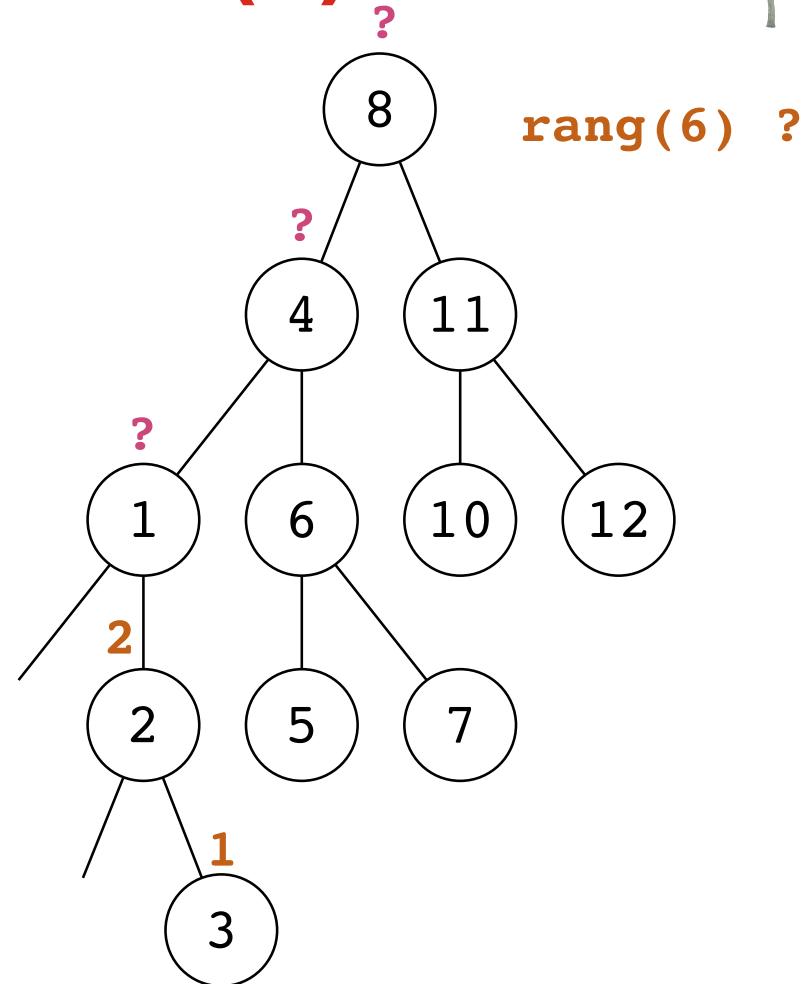




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

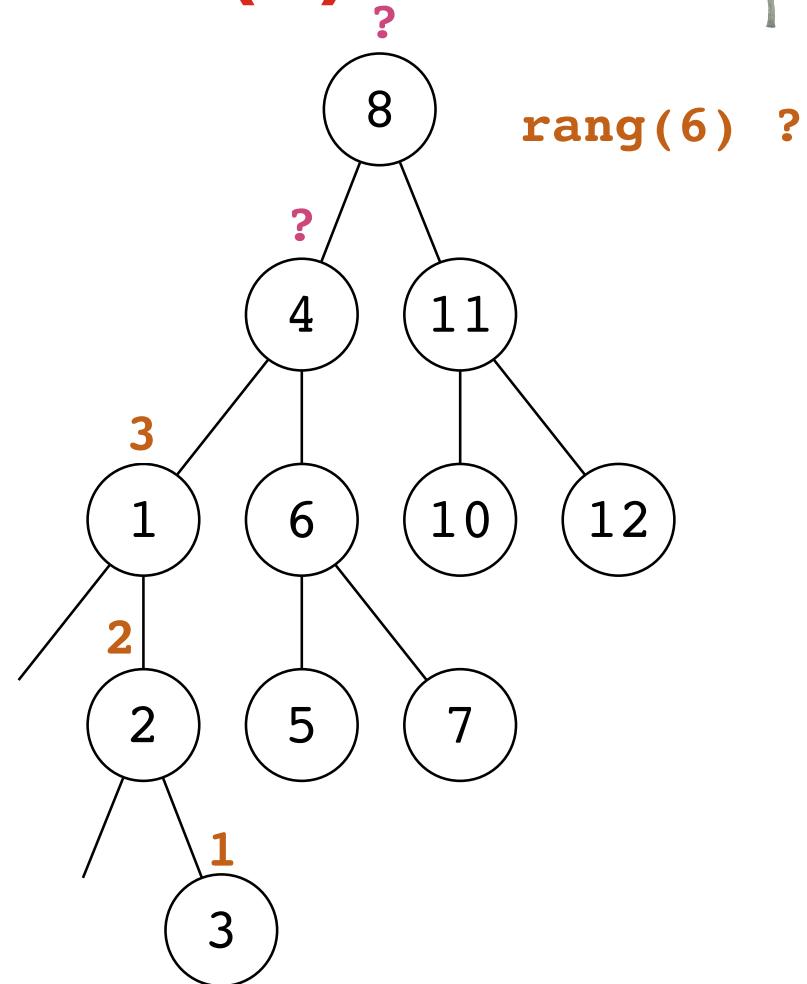




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

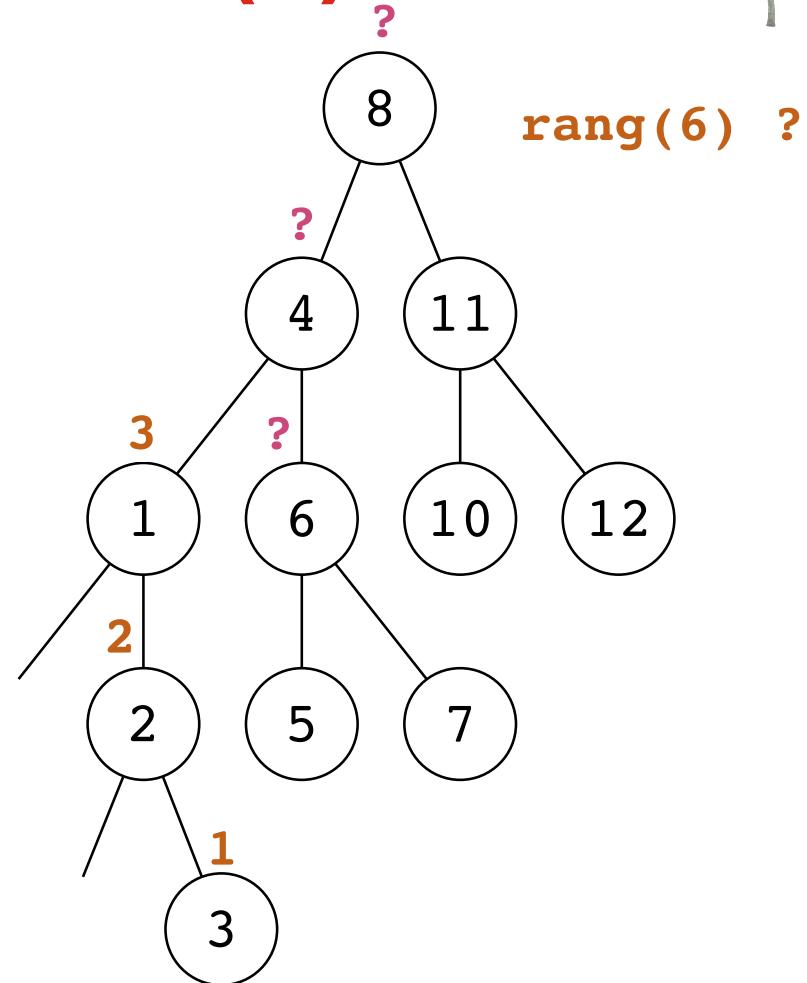




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

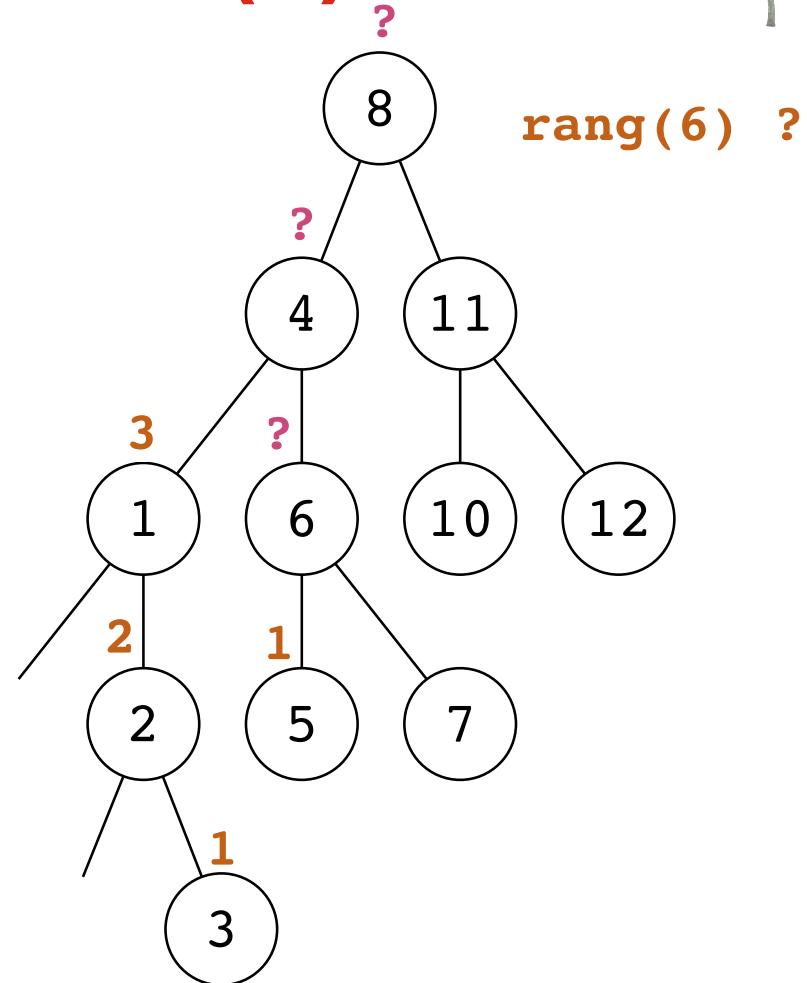




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

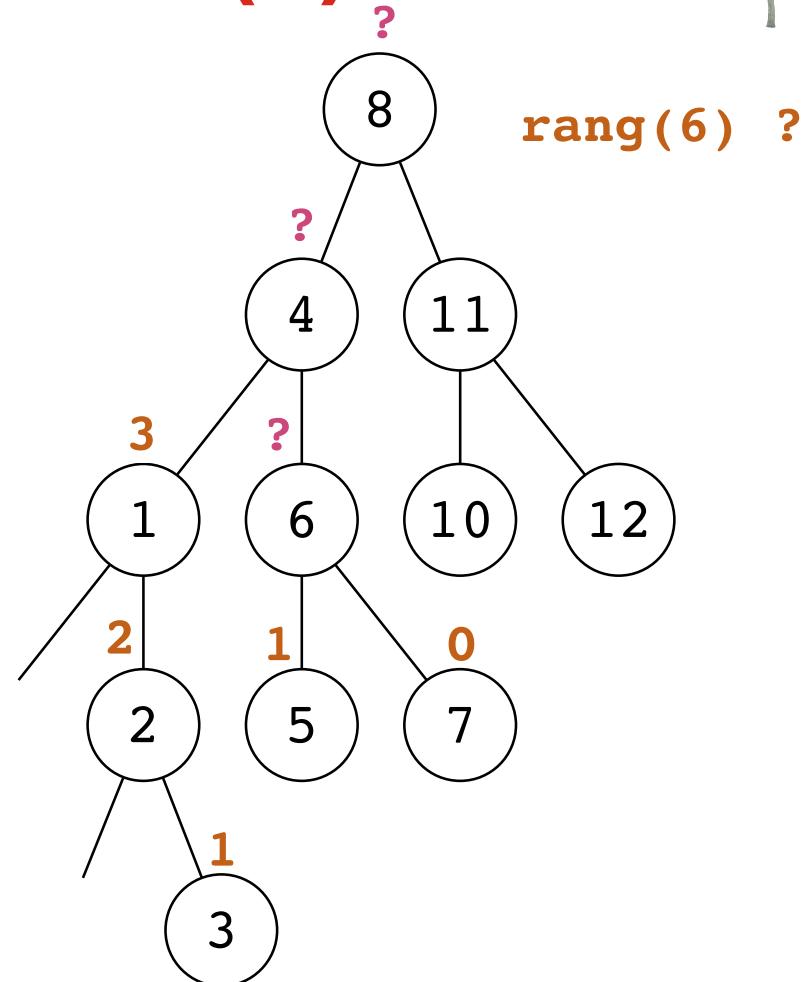




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```



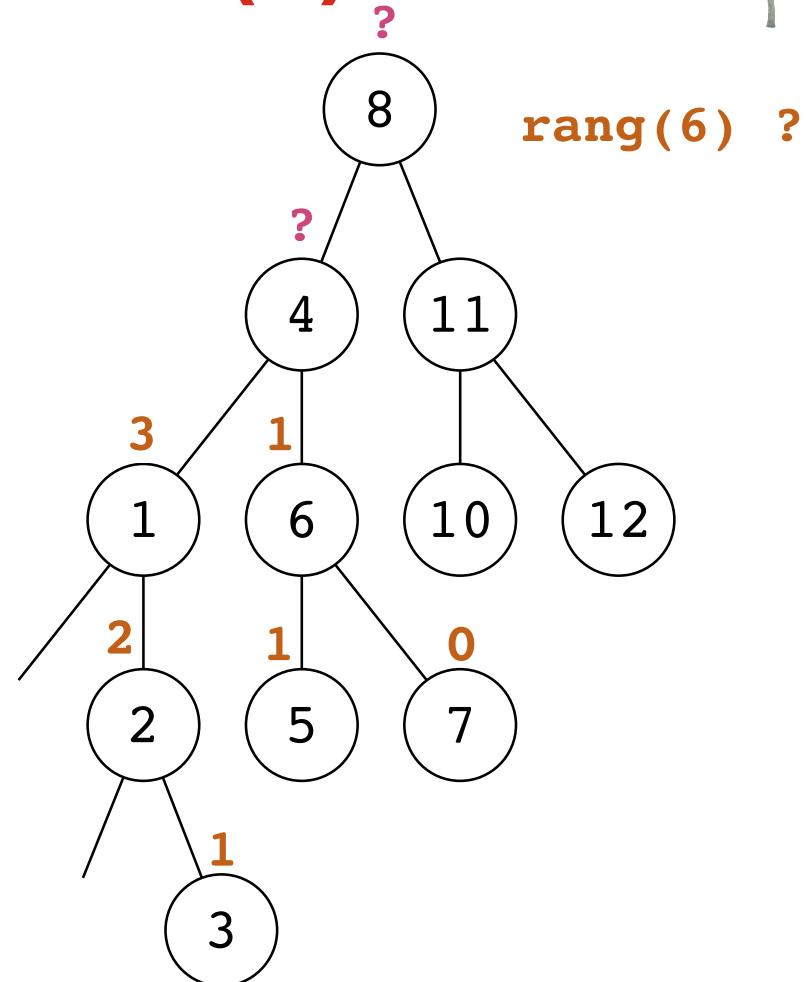


# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```

fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
    
```

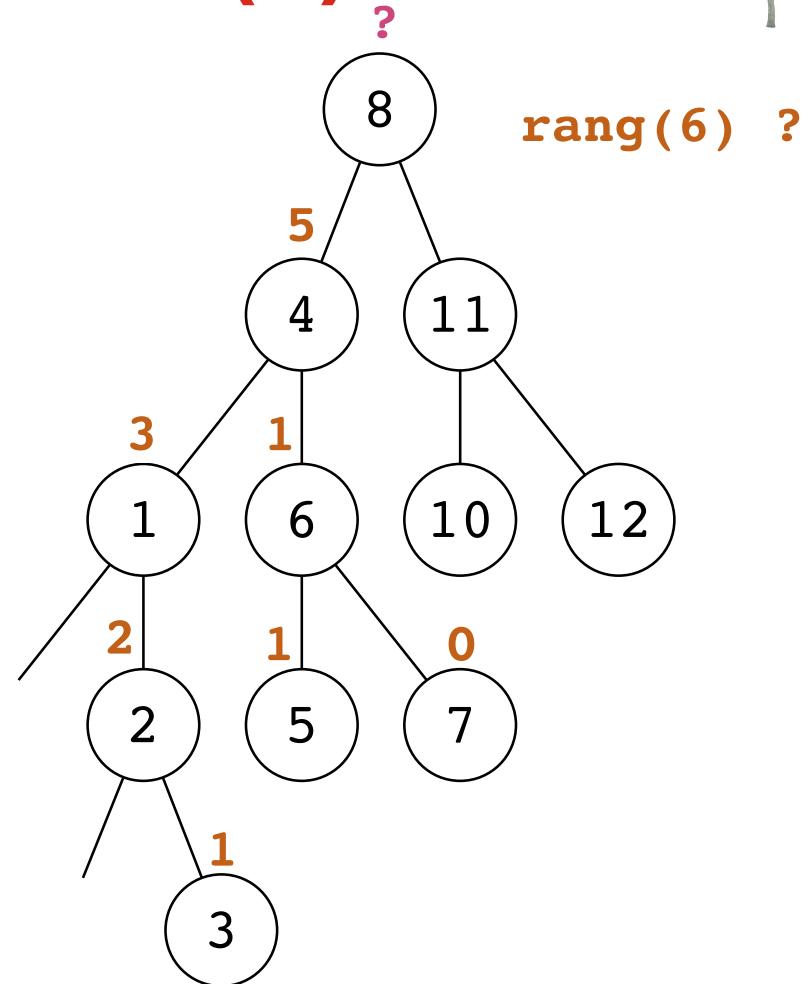




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

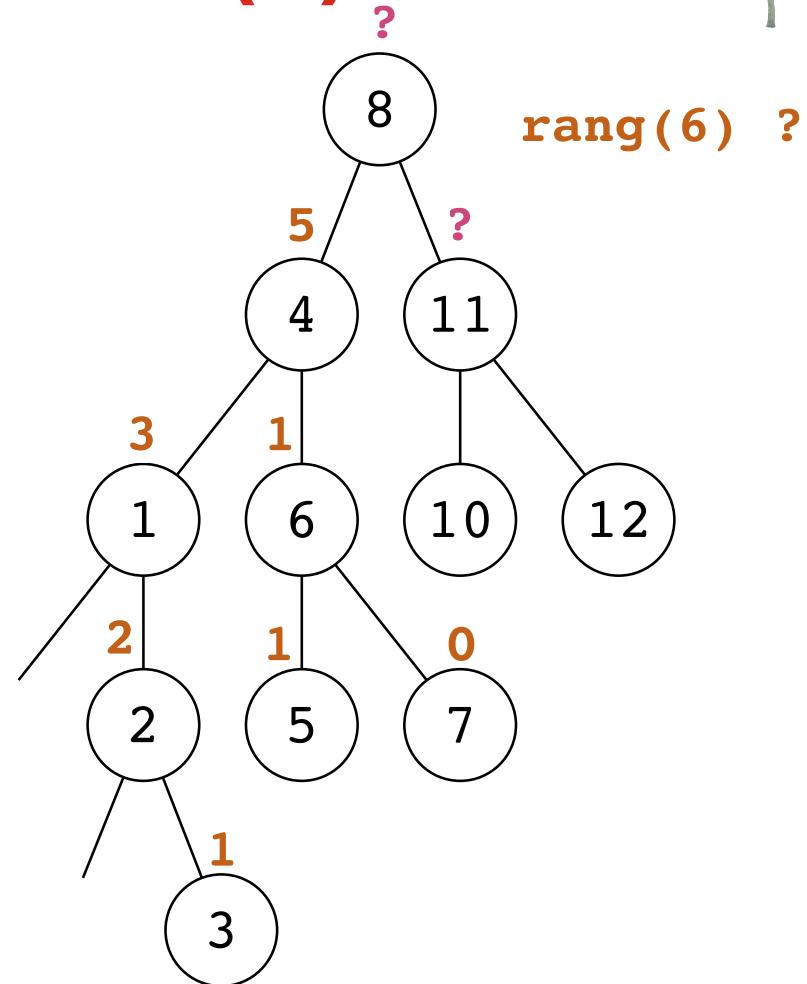




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

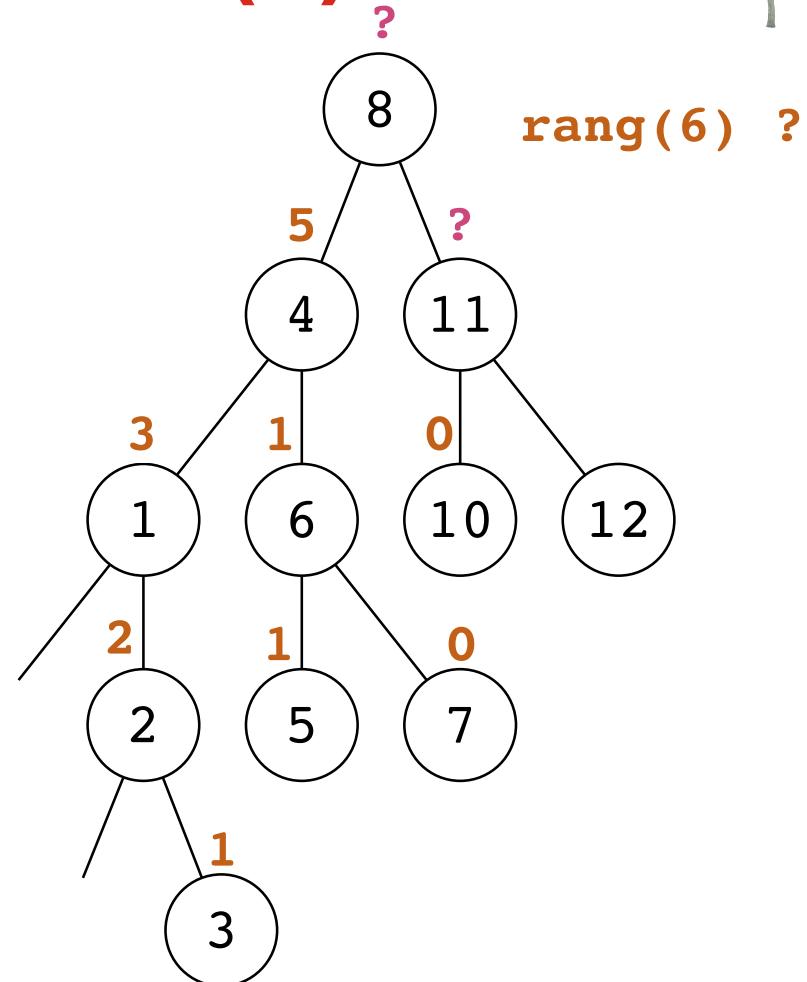




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```

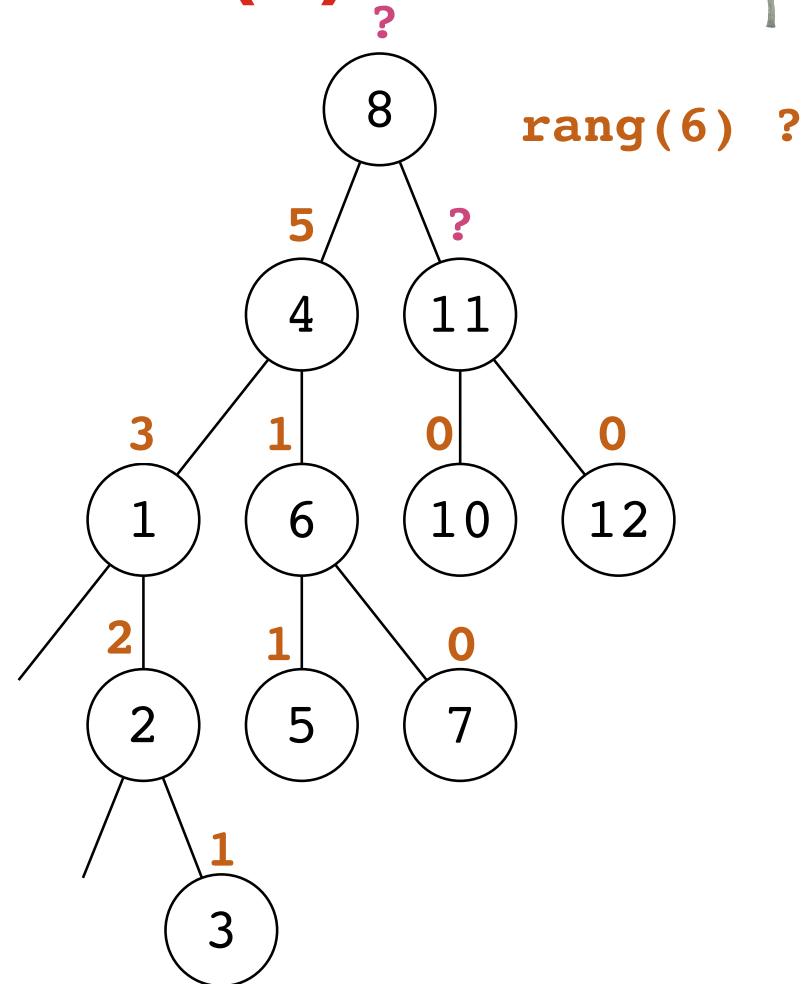




# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
```



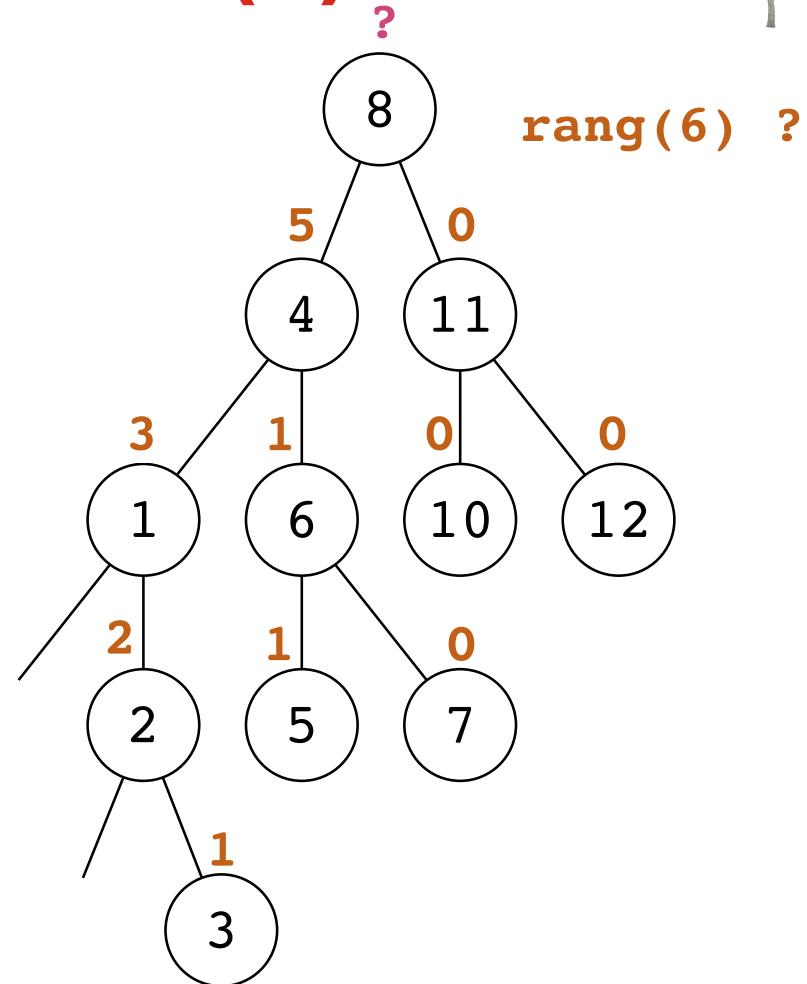


# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```

fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
    
```



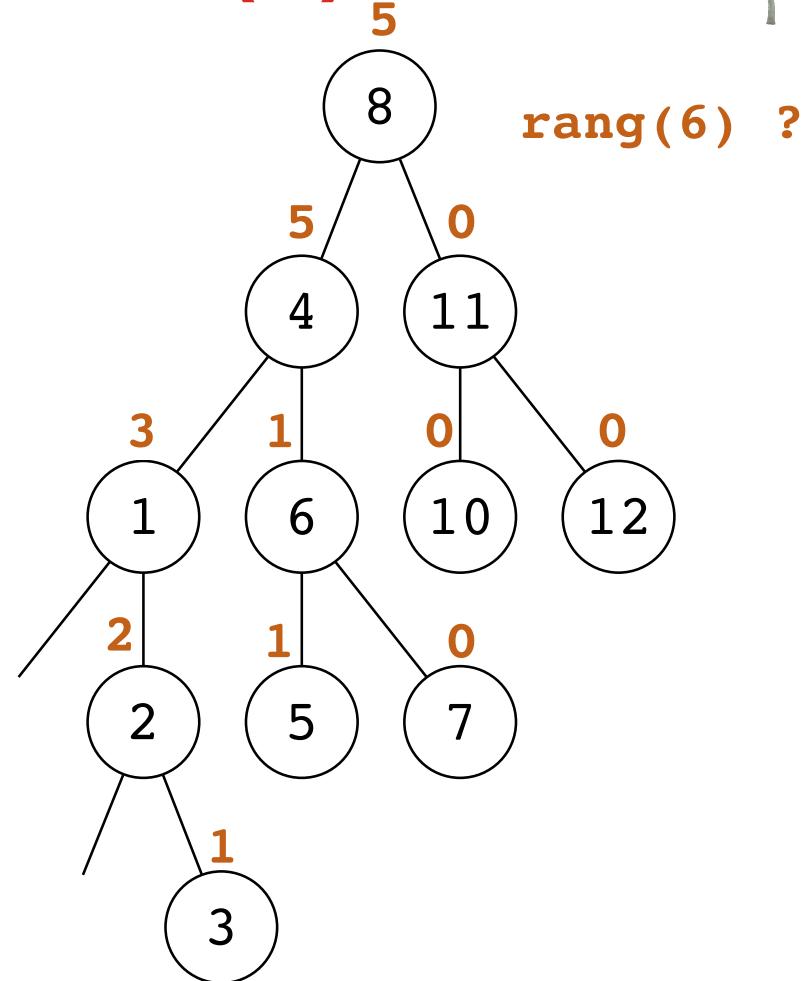


# Rang d'une clé k (2)

- Parcourir tout l'arbre et compter combien de clés sont < k

```

fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        retourner rang(r.gauche, k)
            + rang(r.droit, k)
            + (r.clé < k) ? 1 : 0
    
```

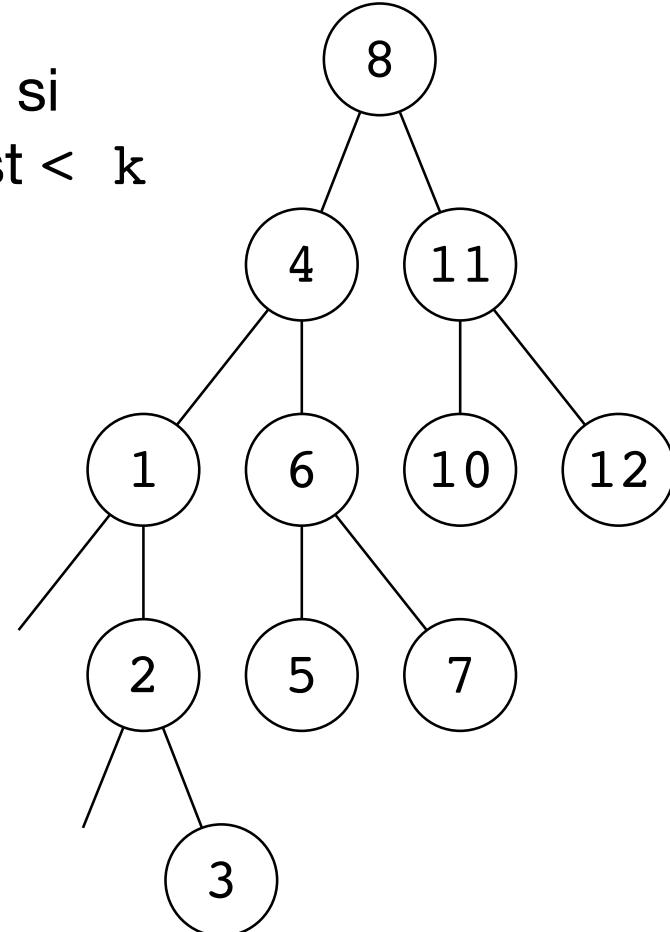




# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
```

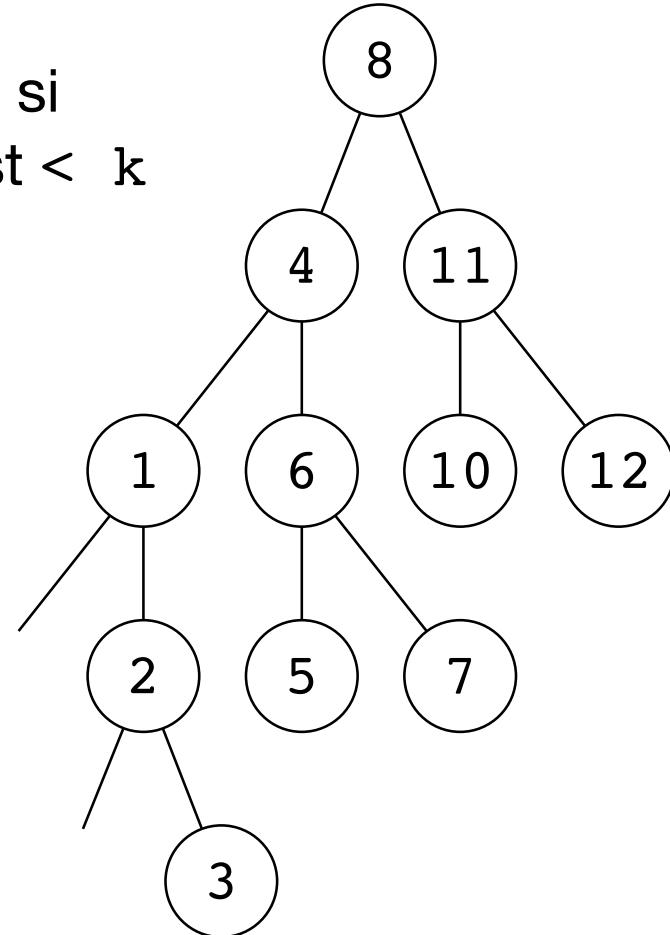




# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



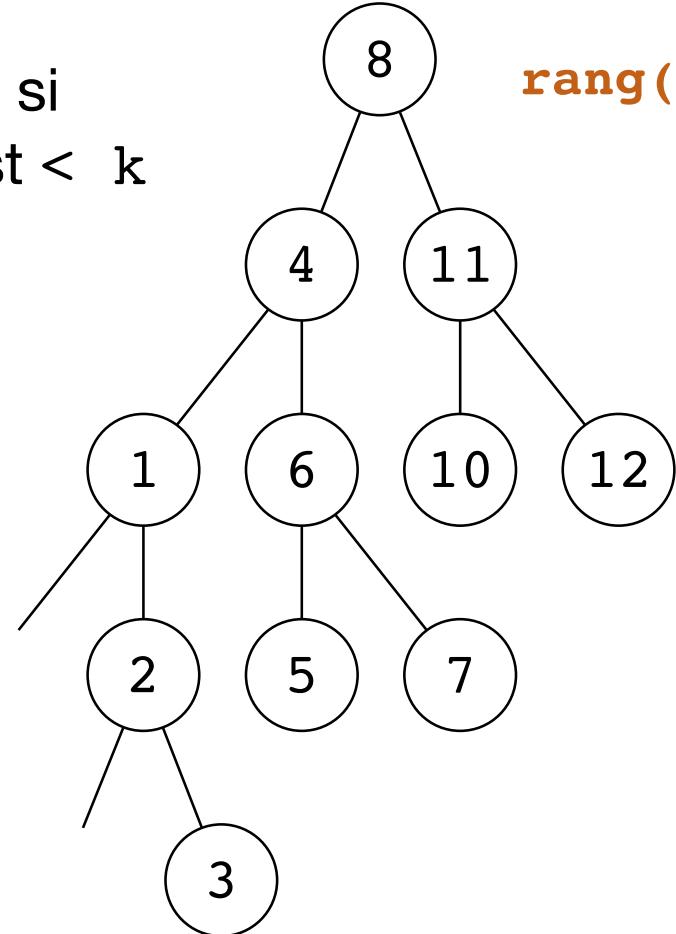


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



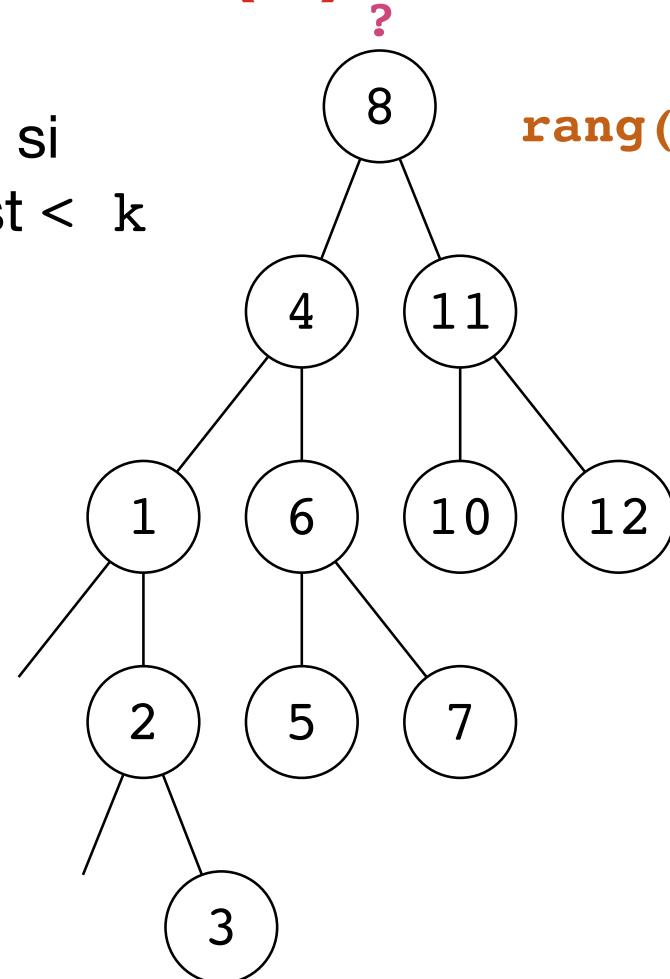


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



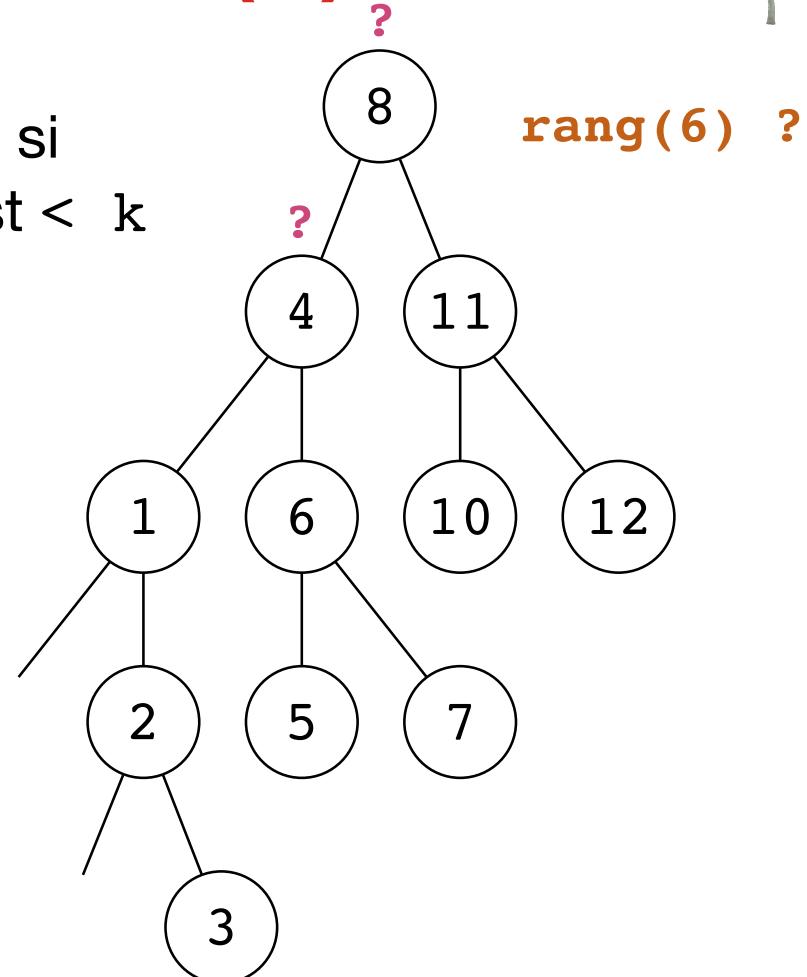


# Rang d'une clé k (3)

Appeler *rang(r.droit, k)* est inutile si *r.clé >= k*, puisqu'aucune clé n'y est < k

**rang(6) ?**

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



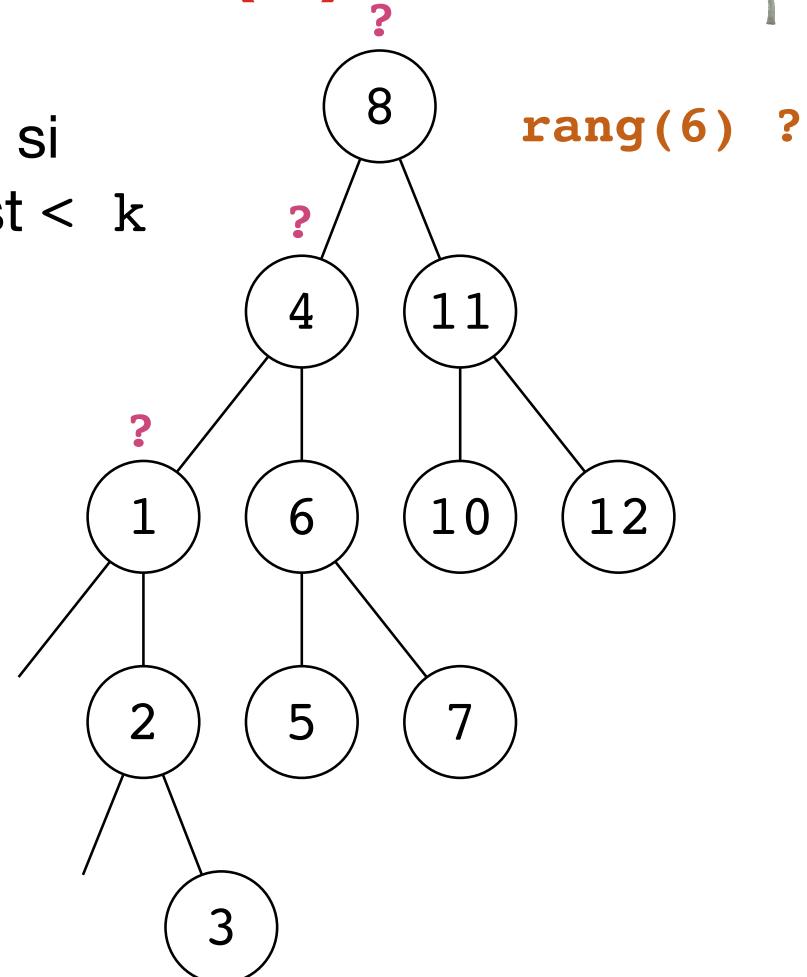


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



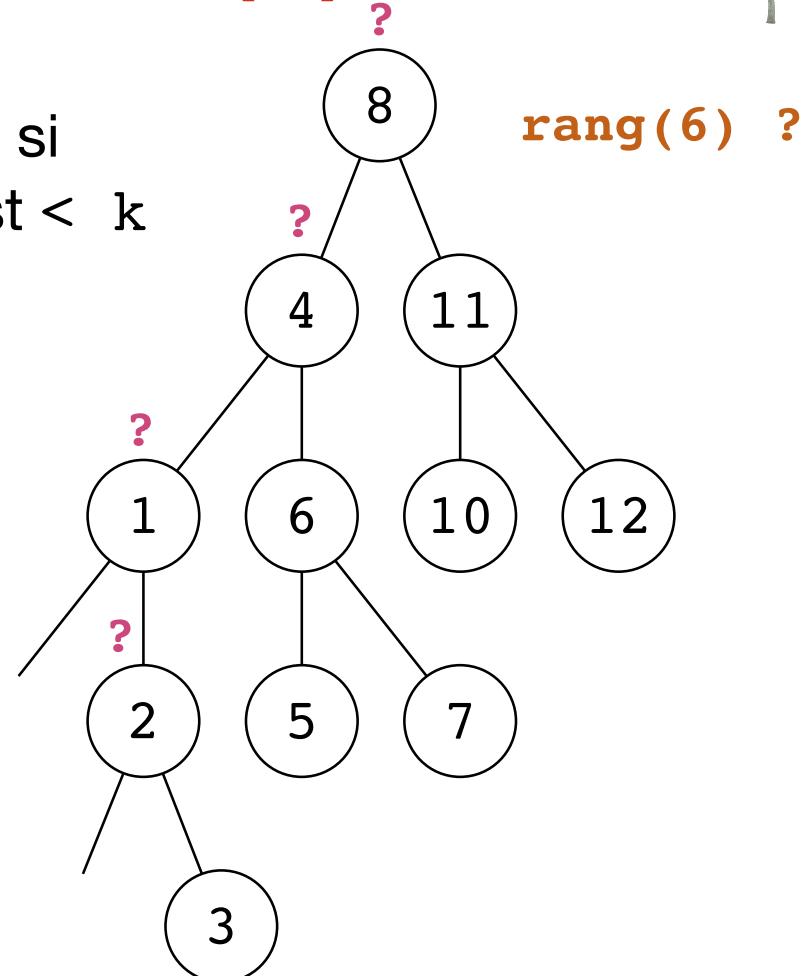


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



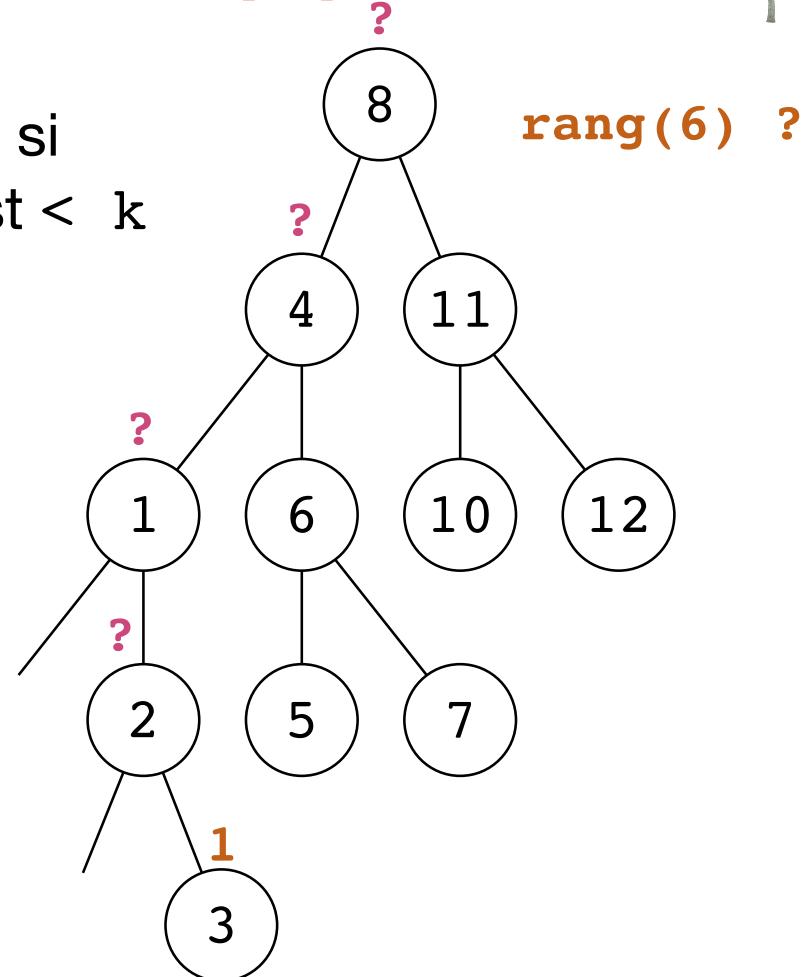


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



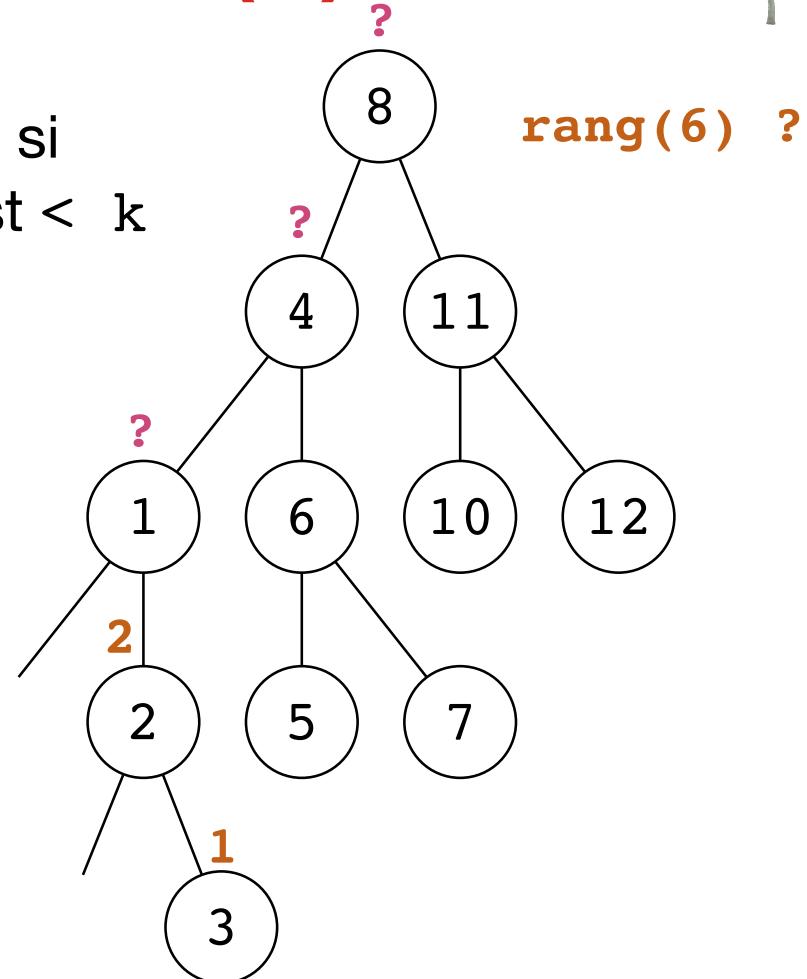


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



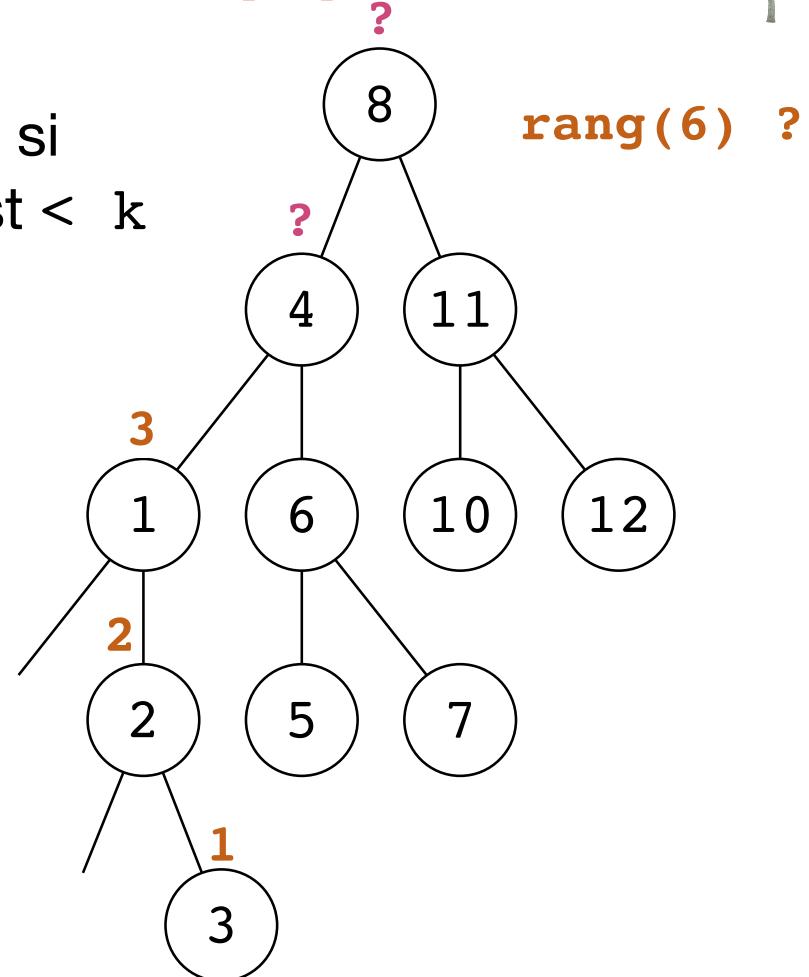


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



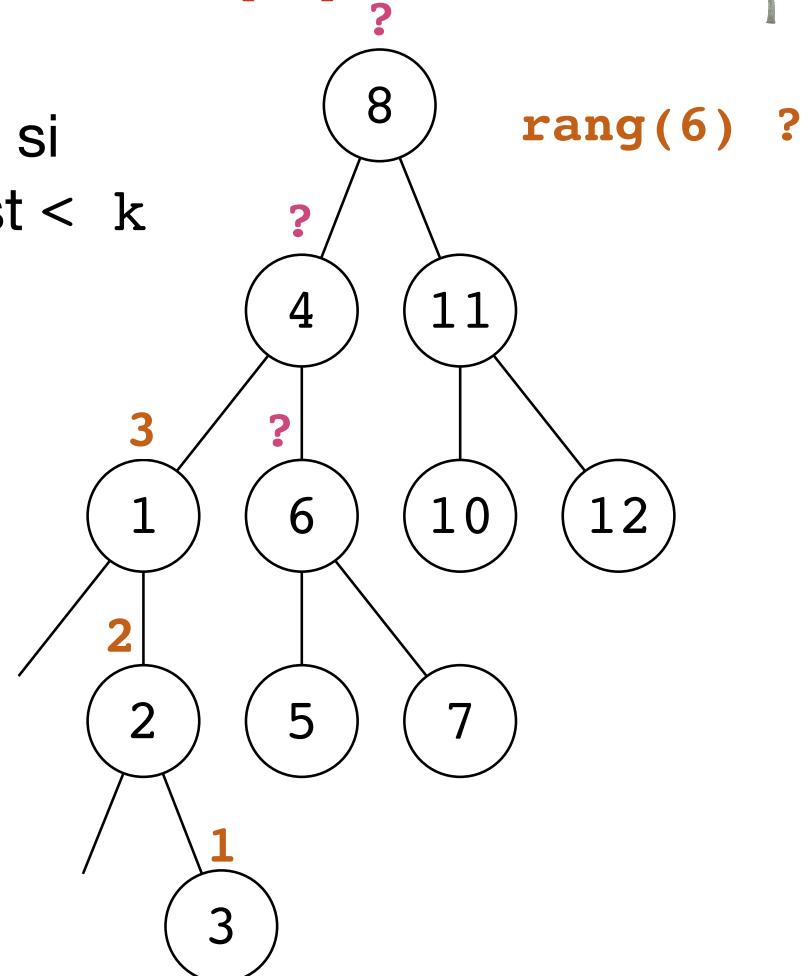


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



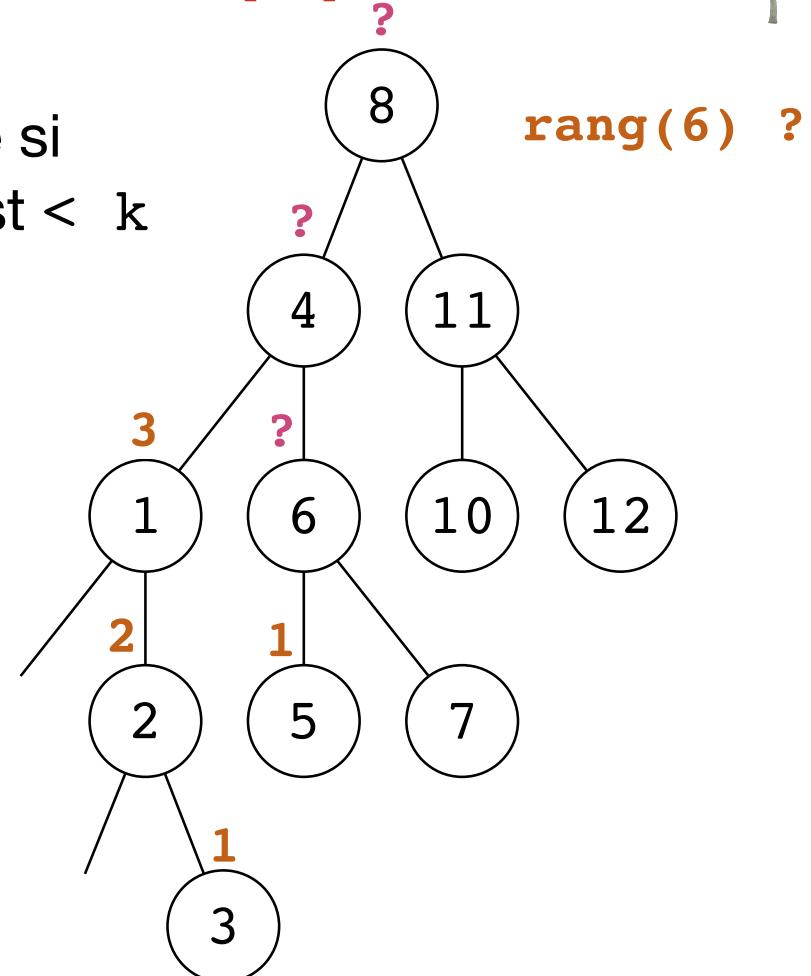


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



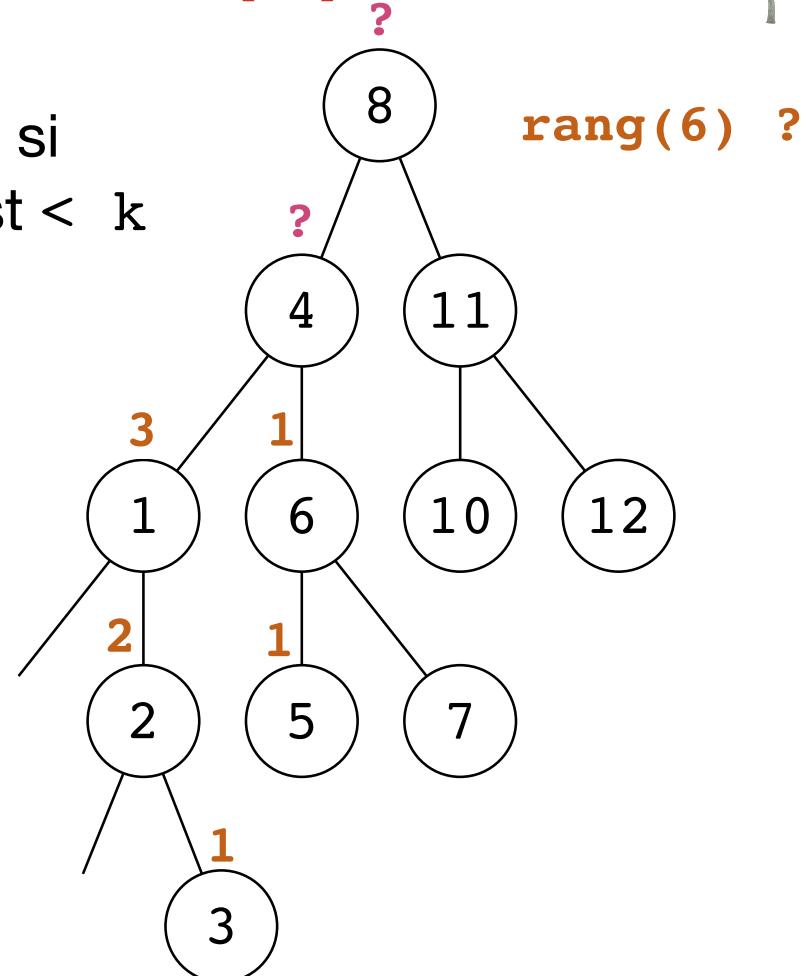


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



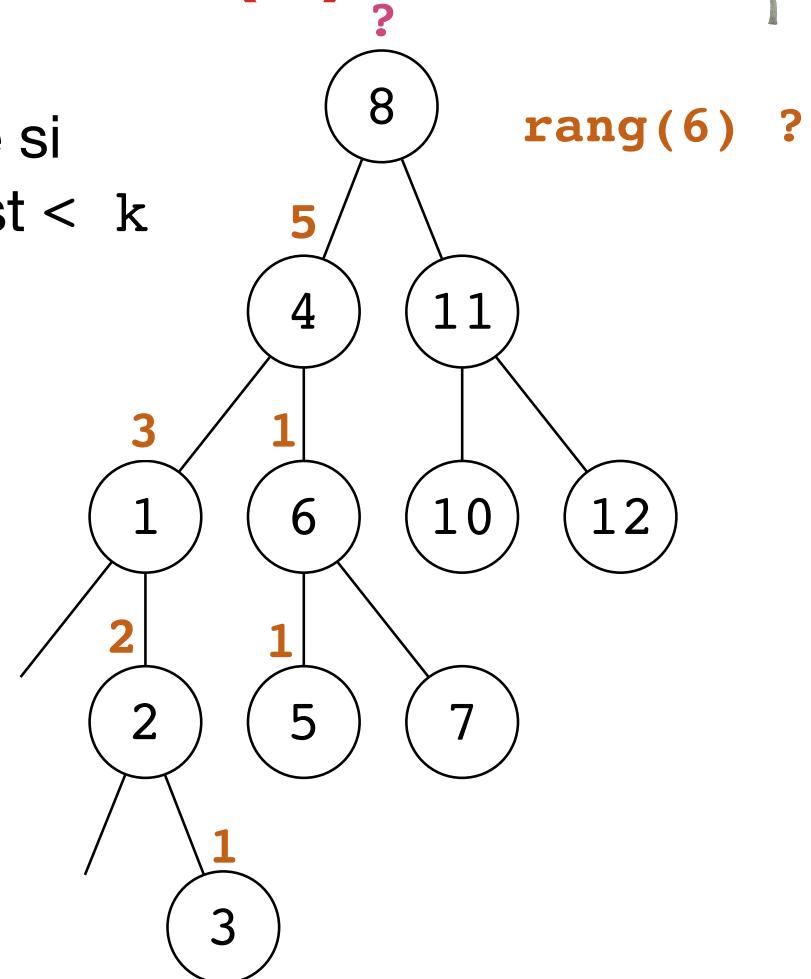


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```



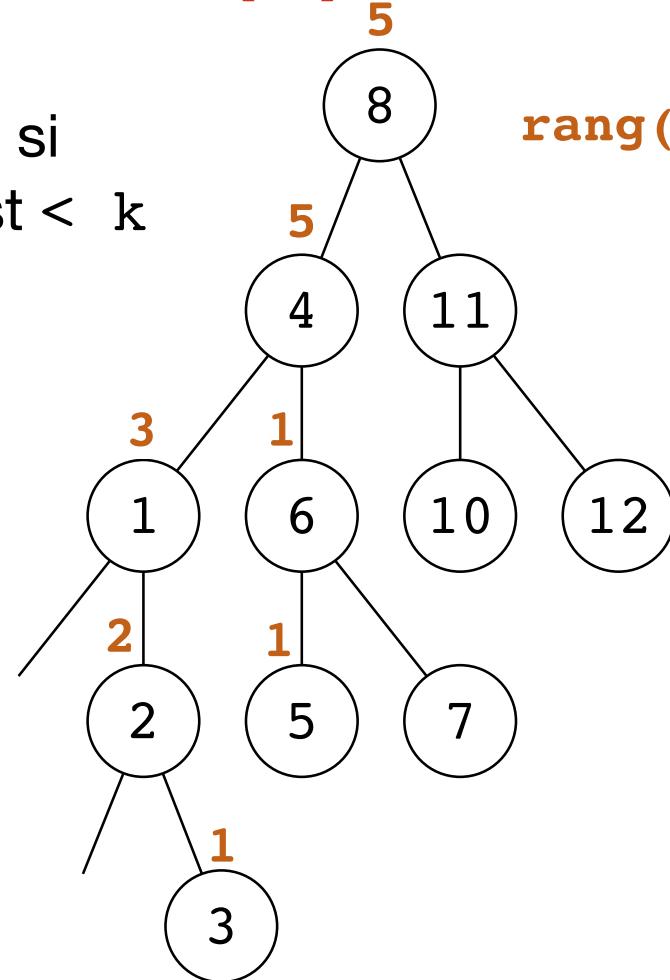


# Rang d'une clé k (3)

Appeler `rang(r.droit, k)` est inutile si `r.clé >= k`, puisqu'aucune clé n'y est  $< k$

`rang(6) ?`

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        rg ← rang(r.gauche, k)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
    retourner rg
```

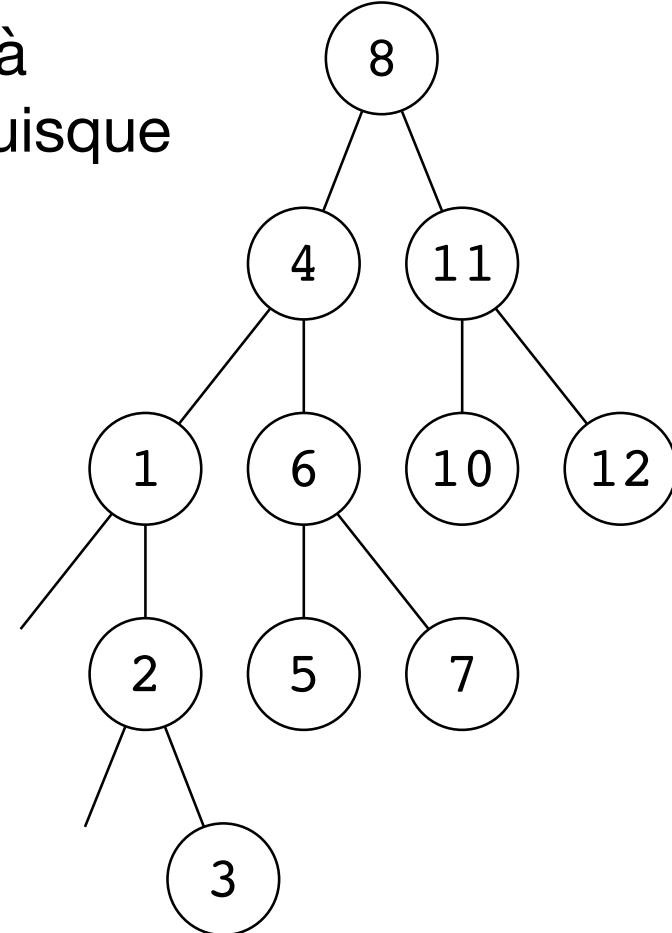




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.clé \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
```

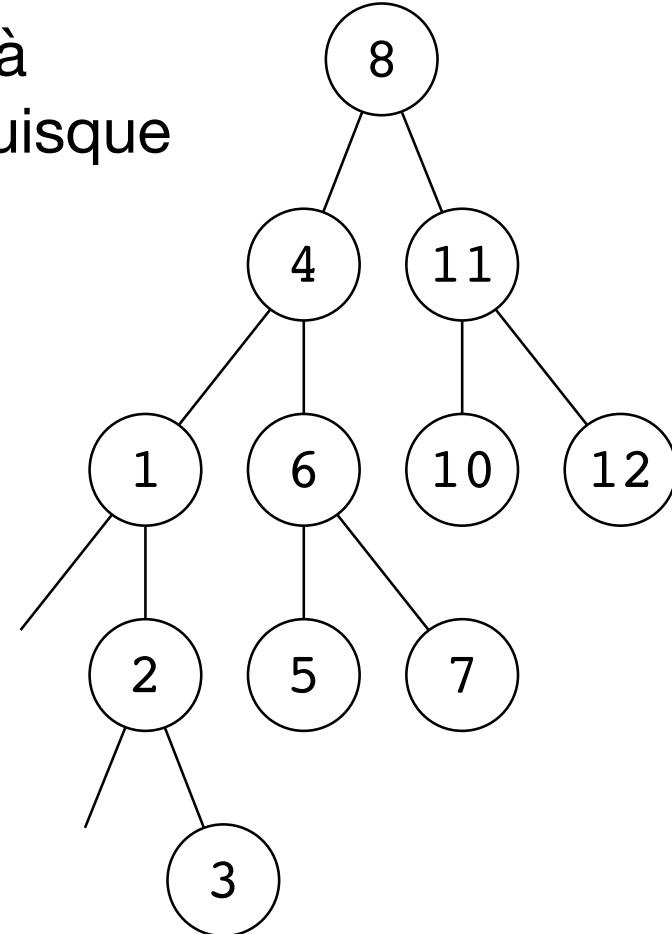




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        si r.clé > k,
            rg ← rang(r.gauche, k)
```

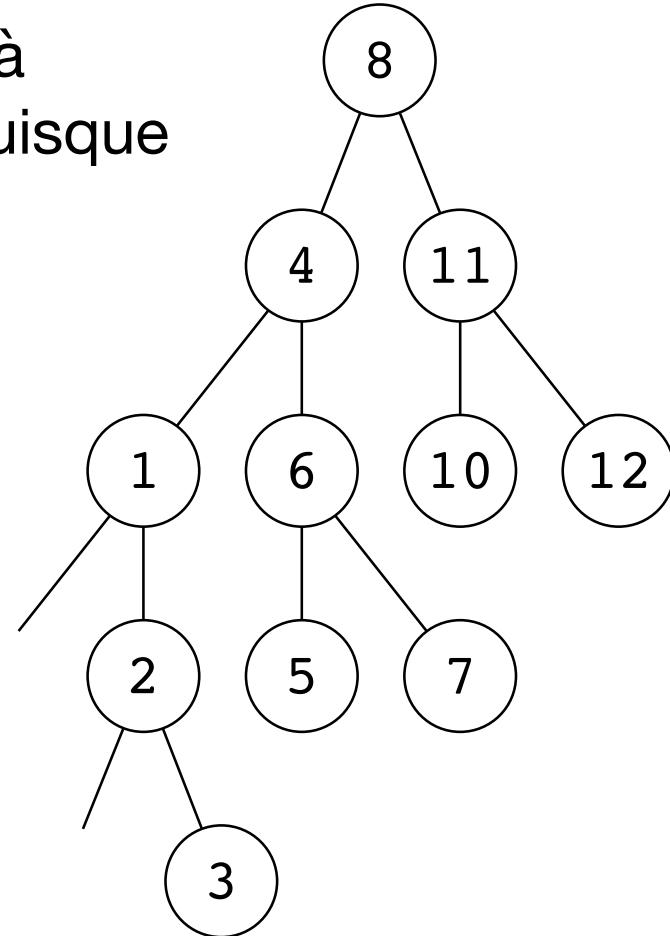




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        si r.clé > k,
            rg ← rang(r.gauche, k)
        sinon,
            rg ← taille(r.gauche)
            si r.clé < k,
                rg ← rg + rang(r.droit, k) + 1
    retourner rg
```

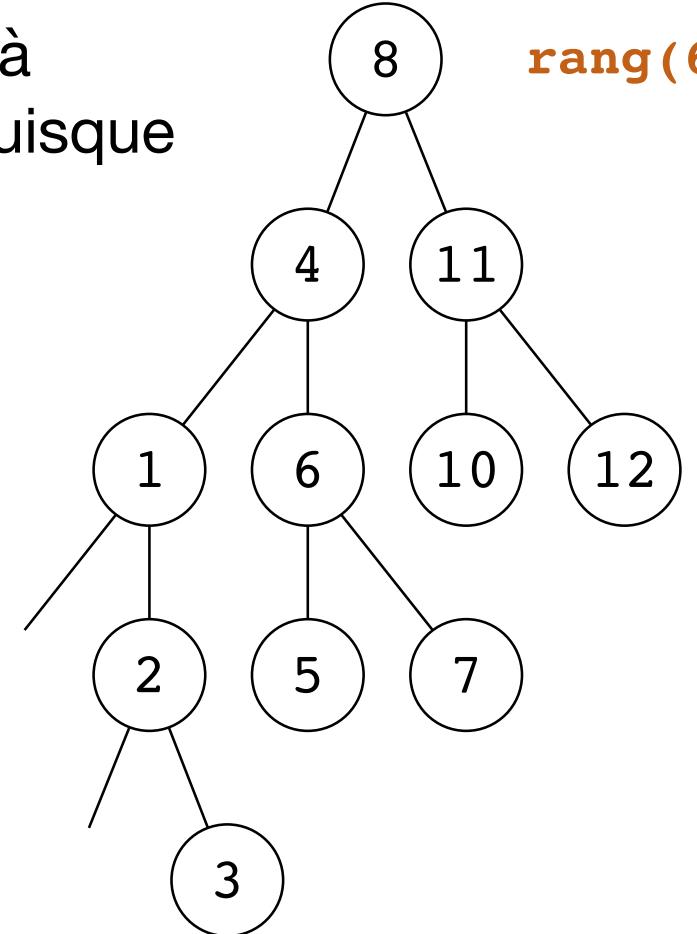




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        si r.clé > k,
            rg ← rang(r.gauche, k)
        sinon,
            rg ← taille(r.gauche)
            si r.clé < k,
                rg ← rg + rang(r.droit, k) + 1
    retourner rg
```





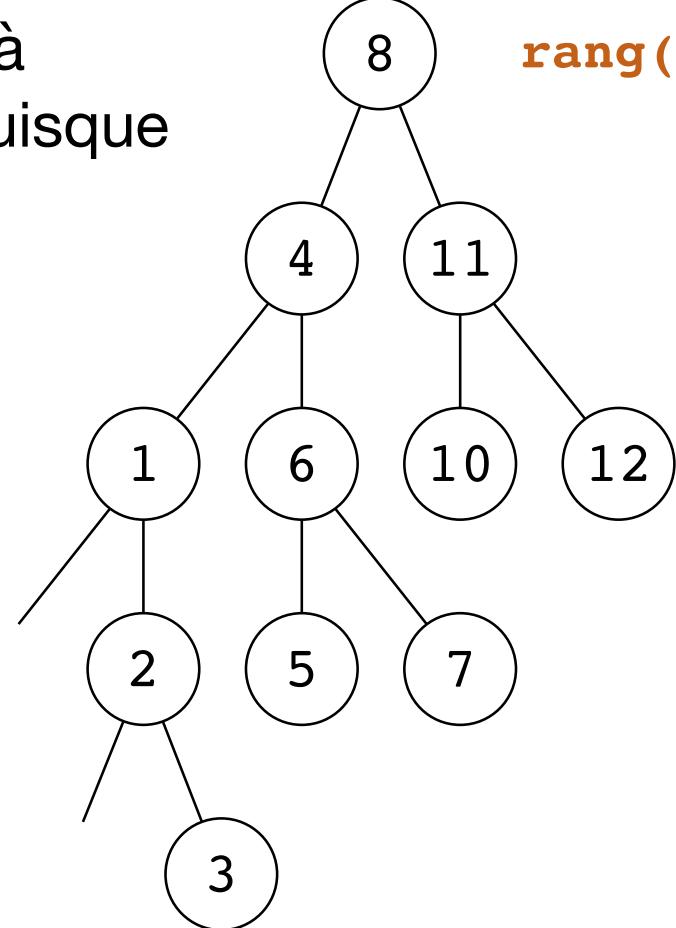
# Rang d'une clé k (4)

?

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

rang(6) ?

```
fonction rang (r, k)
    si r == Ø,
        retourner 0
    sinon,
        si r.clé > k,
            rg ← rang(r.gauche, k)
        sinon,
            rg ← taille(r.gauche)
            si r.clé < k,
                rg ← rg + rang(r.droit, k) + 1
    retourner rg
```

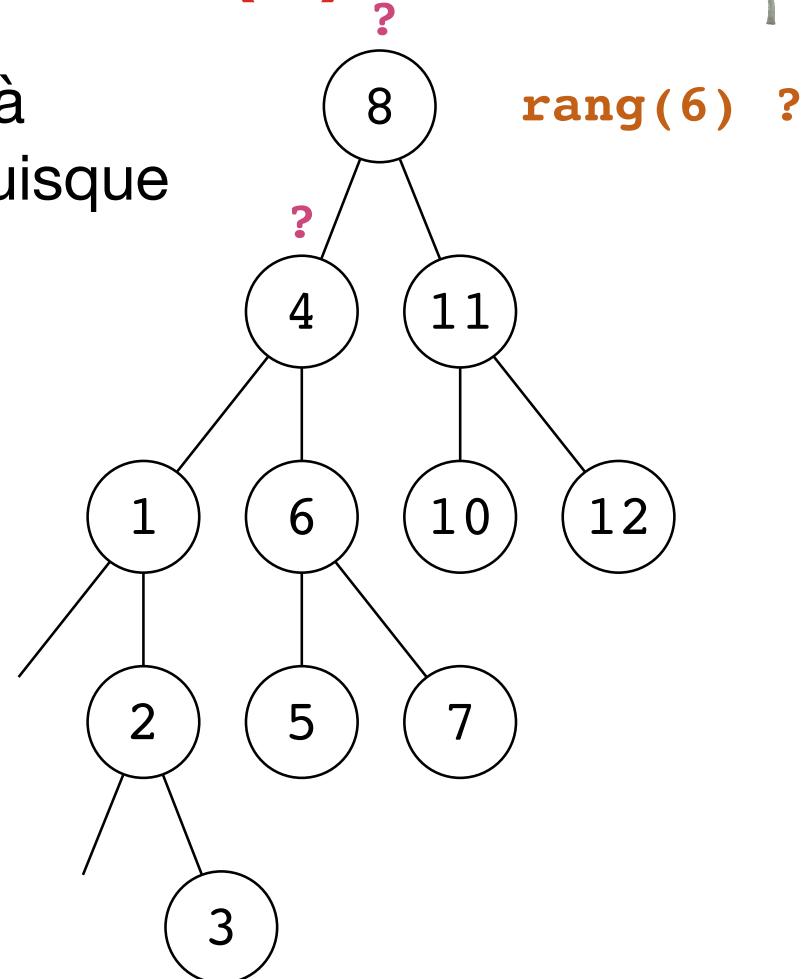




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.clé \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
```



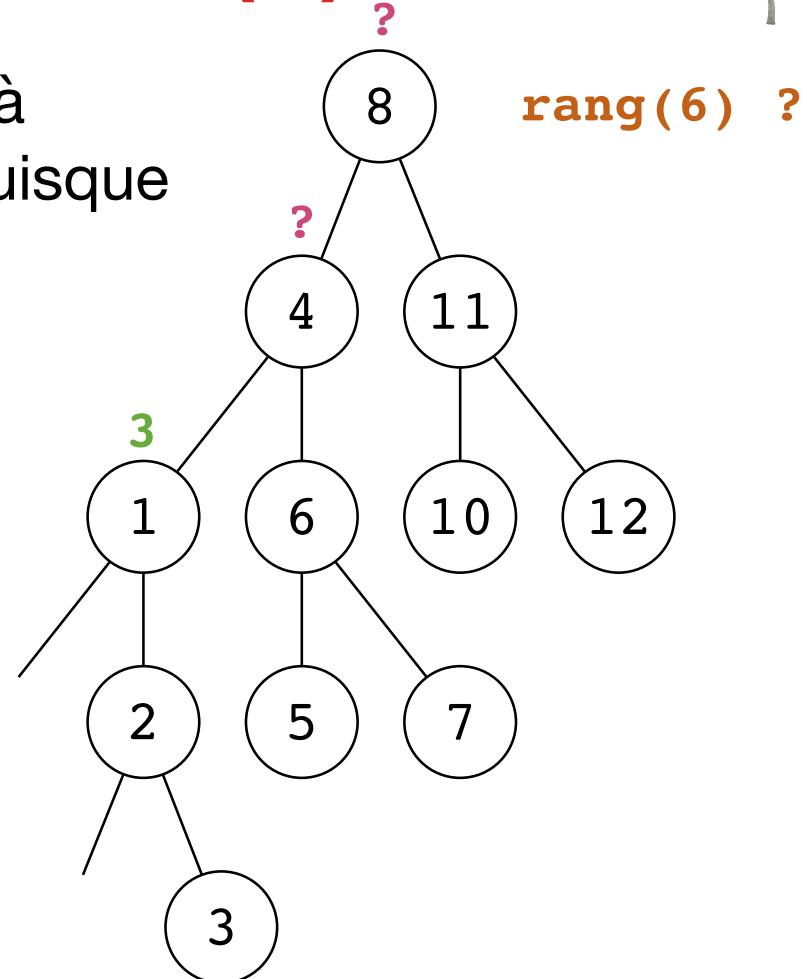


# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

```

fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
    
```

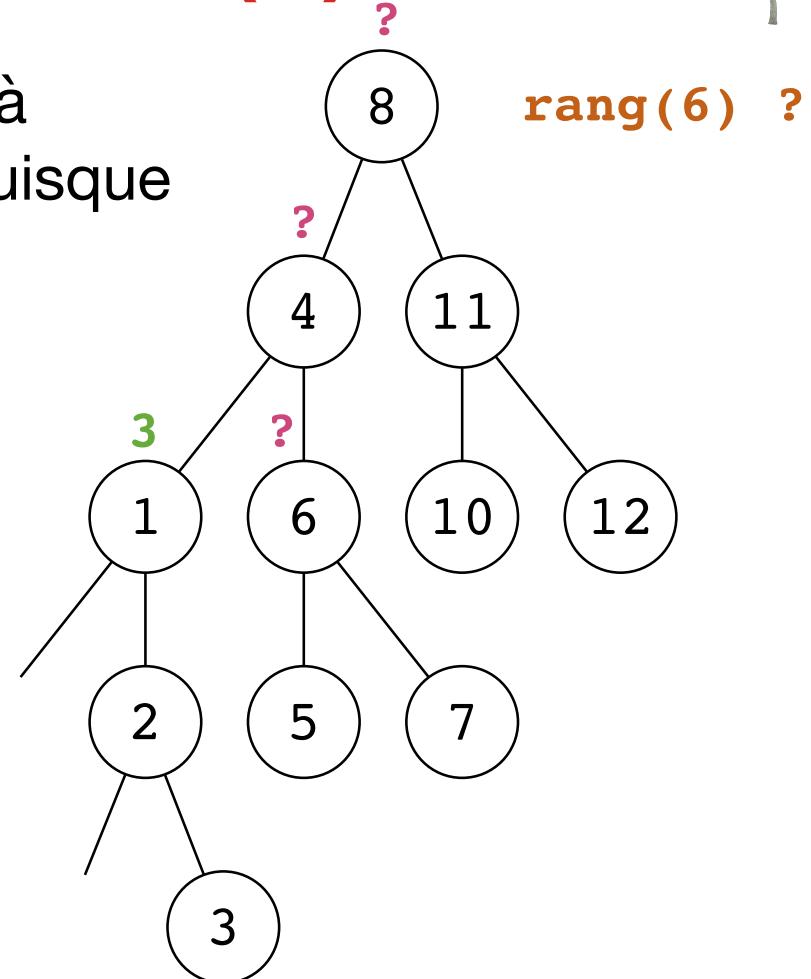




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.clé \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
```



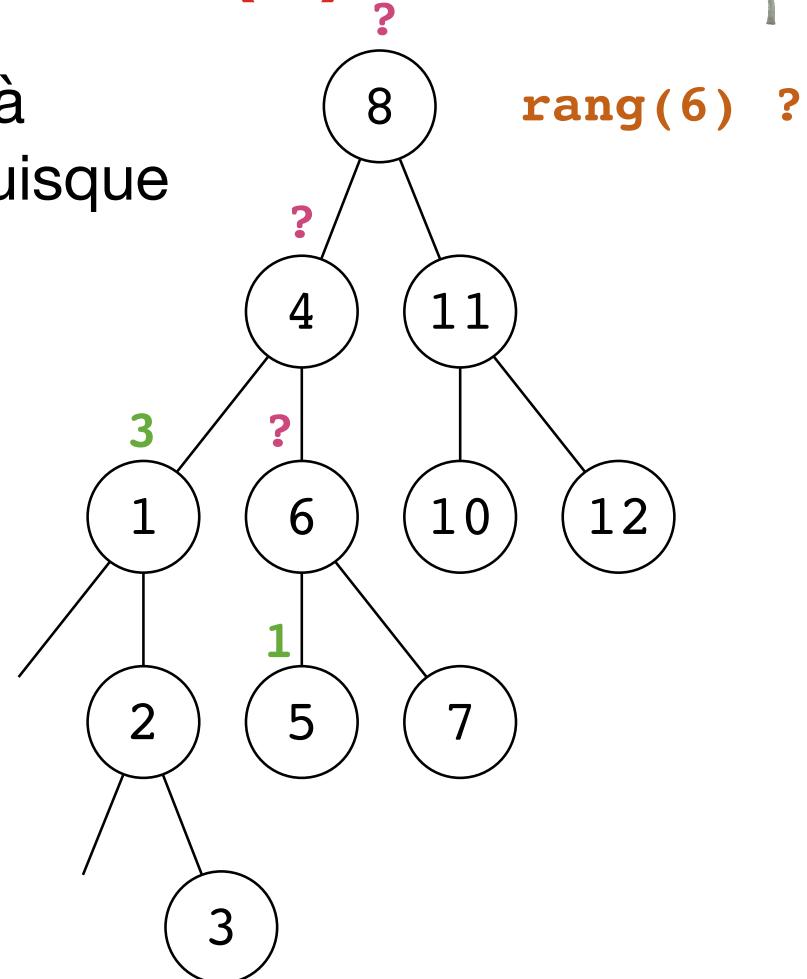


# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.clé \leq k$ , puisque  
toutes les clés y sont  $< k$

```

fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
    
```



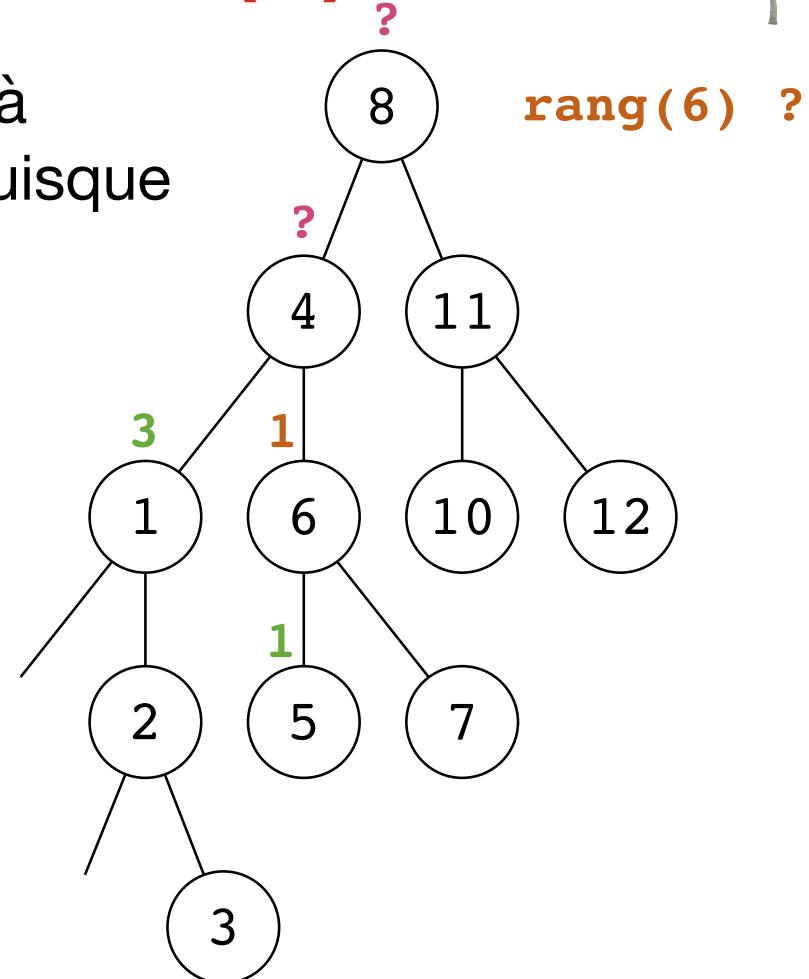


# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.clé \leq k$ , puisque  
toutes les clés y sont  $< k$

```

fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
    
```

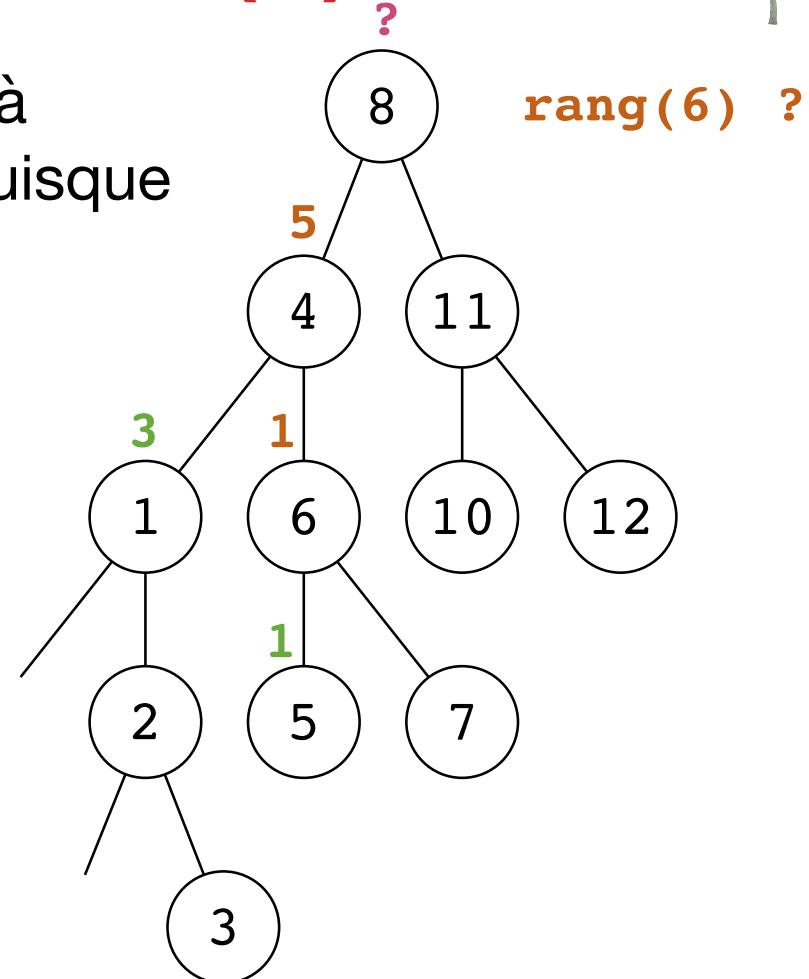




# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

```
fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
```



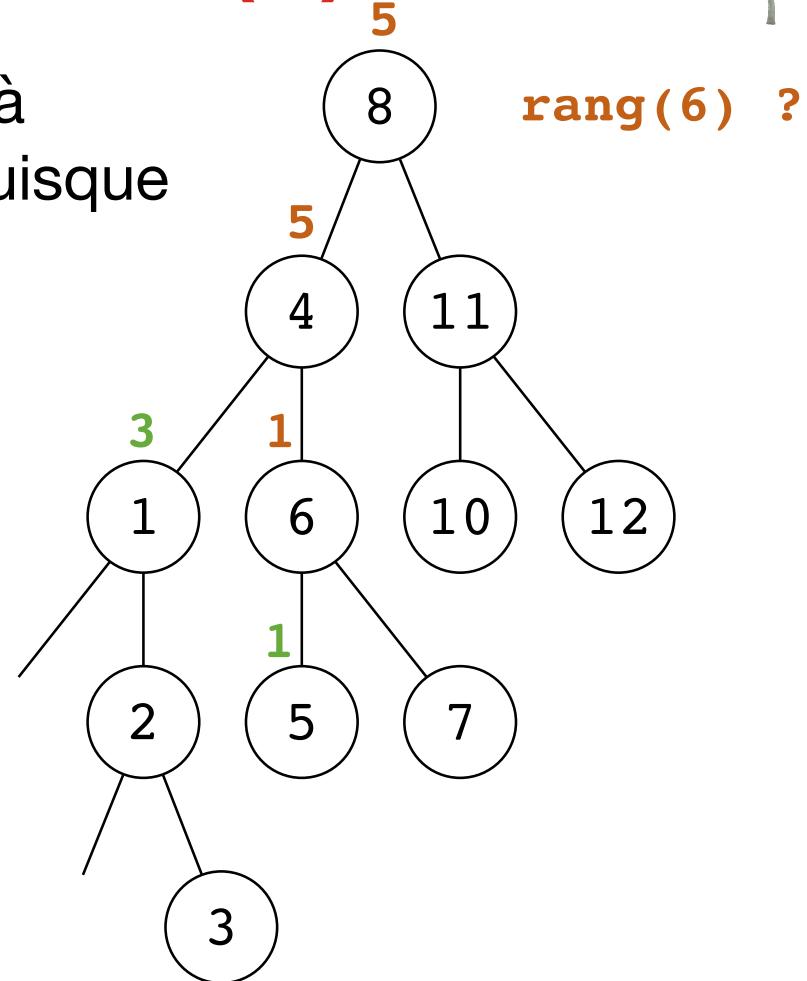


# Rang d'une clé k (4)

*rang(r.gauche, k)* est équivalent à  
*taille(r.gauche)* si  $r.\text{clé} \leq k$ , puisque  
toutes les clés y sont  $< k$

```

fonction rang (r, k)
si r == Ø,
    retourner 0
sinon,
    si r.clé > k,
        rg ← rang(r.gauche, k)
    sinon,
        rg ← taille(r.gauche)
        si r.clé < k,
            rg ← rg + rang(r.droit, k) + 1
retourner rg
    
```





# Elément de rang n (1) - sélection

- Fonction inverse de la précédente

```
rang(r, élément_n(r, n).clé) == n
```



# Elément de rang n (1) - sélection

- Fonction inverse de la précédente

```
rang(r, élément_n(r, n).clé) == n
```

- Effectuer les n premiers pas d'un parcours symétrique...



# Elément de rang n (1) - sélection

- Fonction inverse de la précédente

```
rang(r, élément_n(r, n).clé) == n
```

- Effectuer les n premiers pas d'un parcours symétrique...
- Mais on évite de parcourir les sous-arbres gauches si c'est juste pour en mesurer la taille...



# Elément de rang n (1) - sélection

- Fonction inverse de la précédente

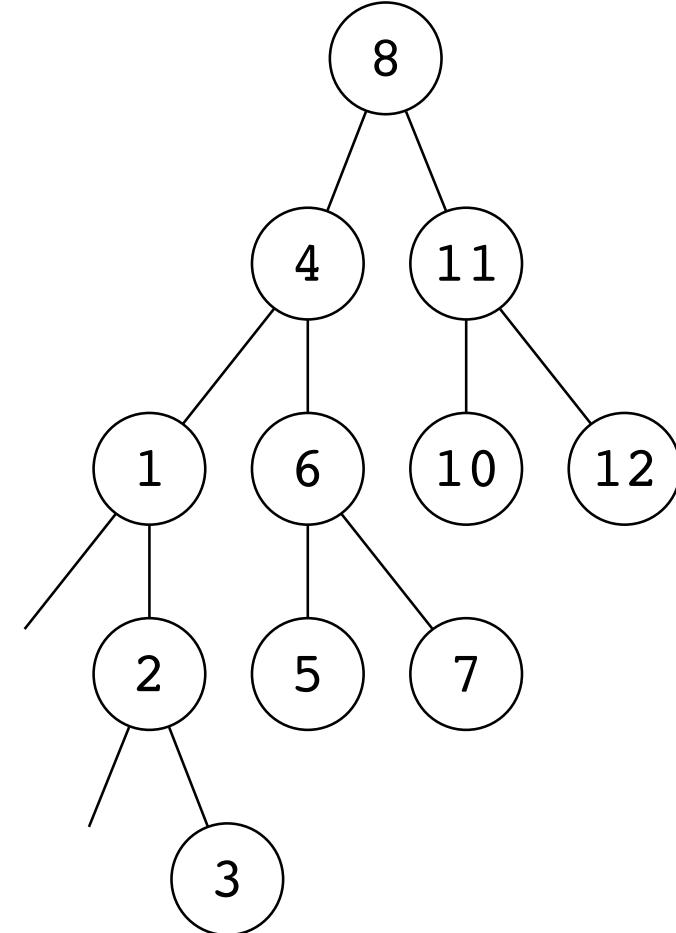
```
rang(r, élément_n(r, n).clé) == n
```

- Effectuer les n premiers pas d'un parcours symétrique...
- Mais on évite de parcourir les sous-arbres gauches si c'est juste pour en mesurer la taille...
- Équivalent à la recherche d'une clé, mais décision sur le nombre d'éléments plus petits plutôt que le clé



# Elément de rang n (2)

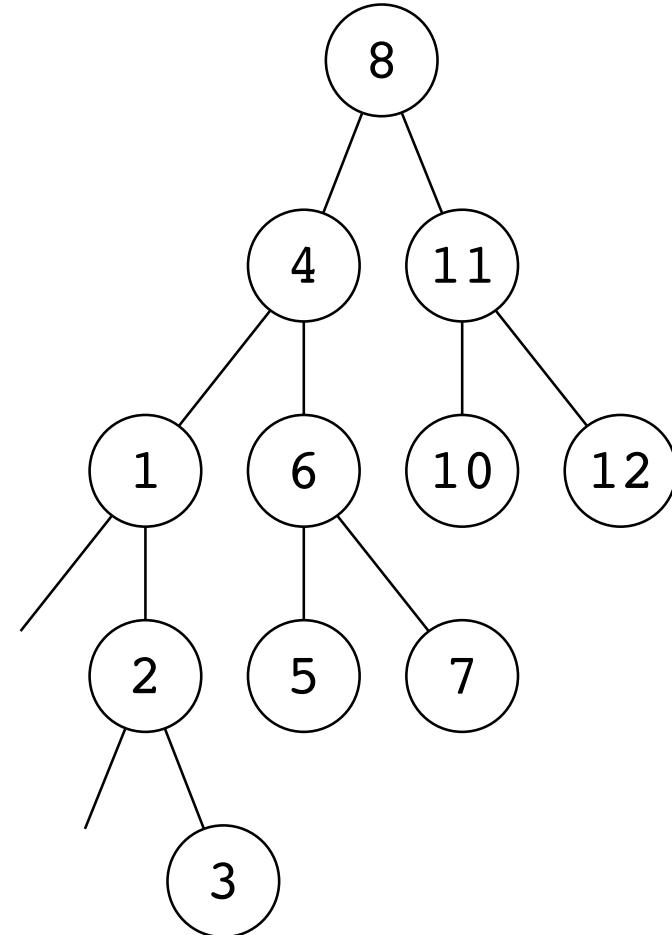
```
fonction élément_n (r, n)
```





# Elément de rang n (2)

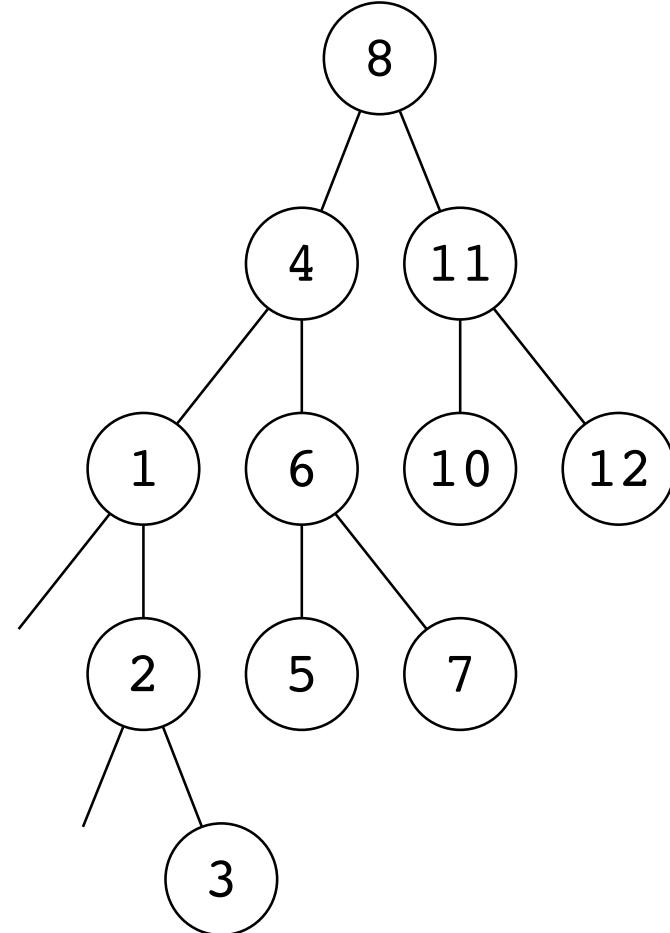
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
```





# Elément de rang n (2)

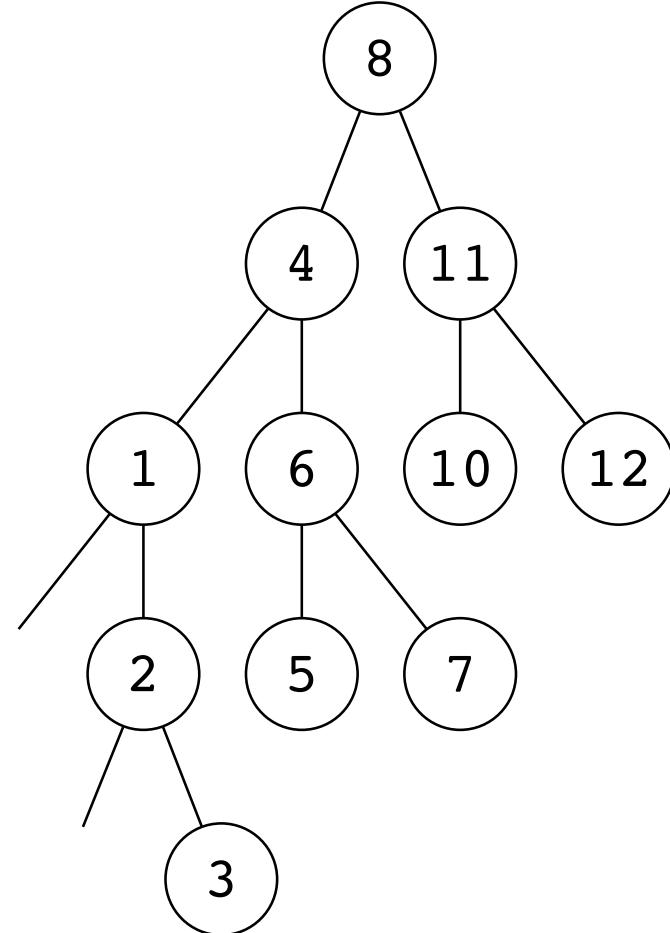
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
```





# Elément de rang n (2)

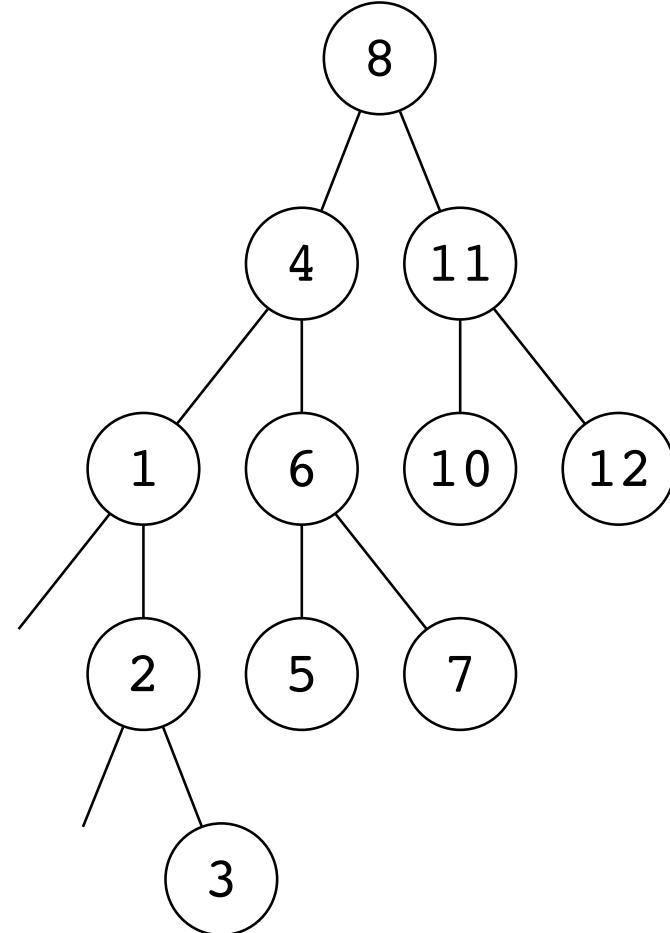
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
```





# Elément de rang n (2)

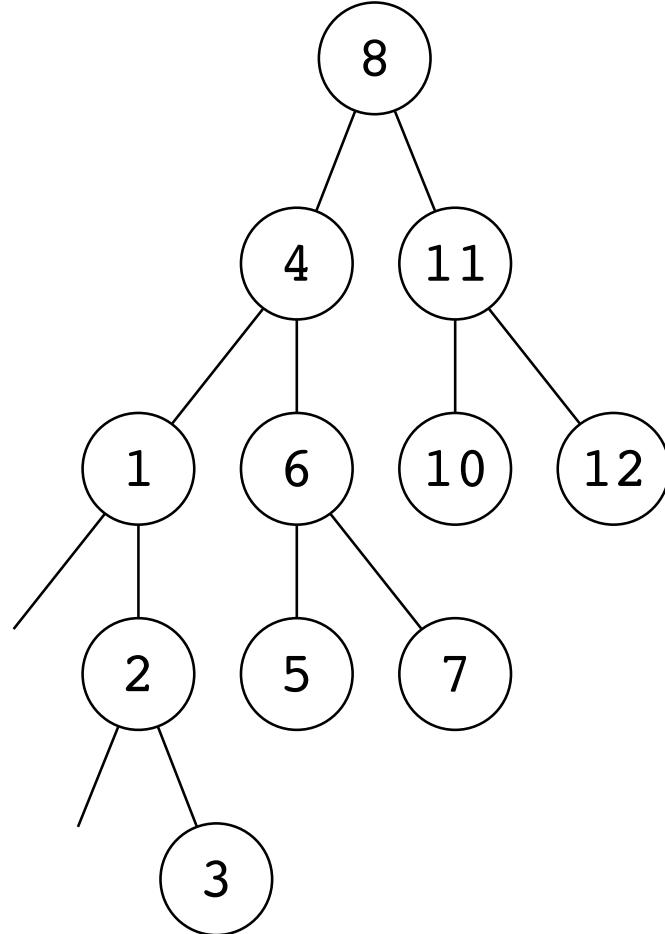
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
```





# Elément de rang n (2)

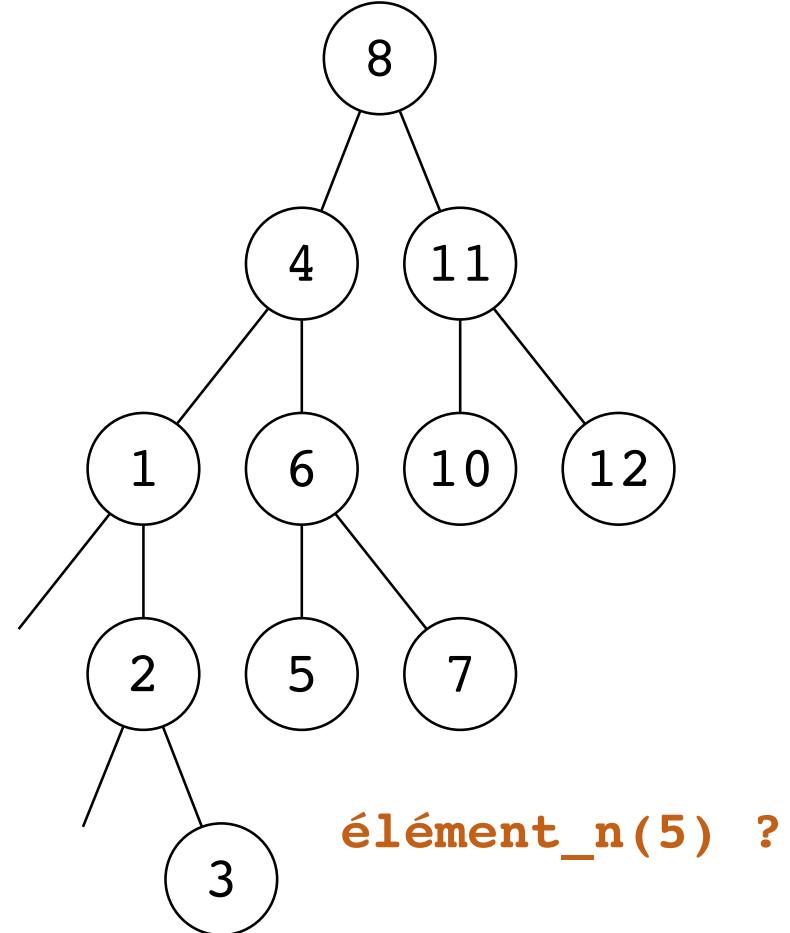
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche,n)
        sinon, (n > t)
            retourner élément_n(r.droit,n-t-1)
```





# Elément de rang n (2)

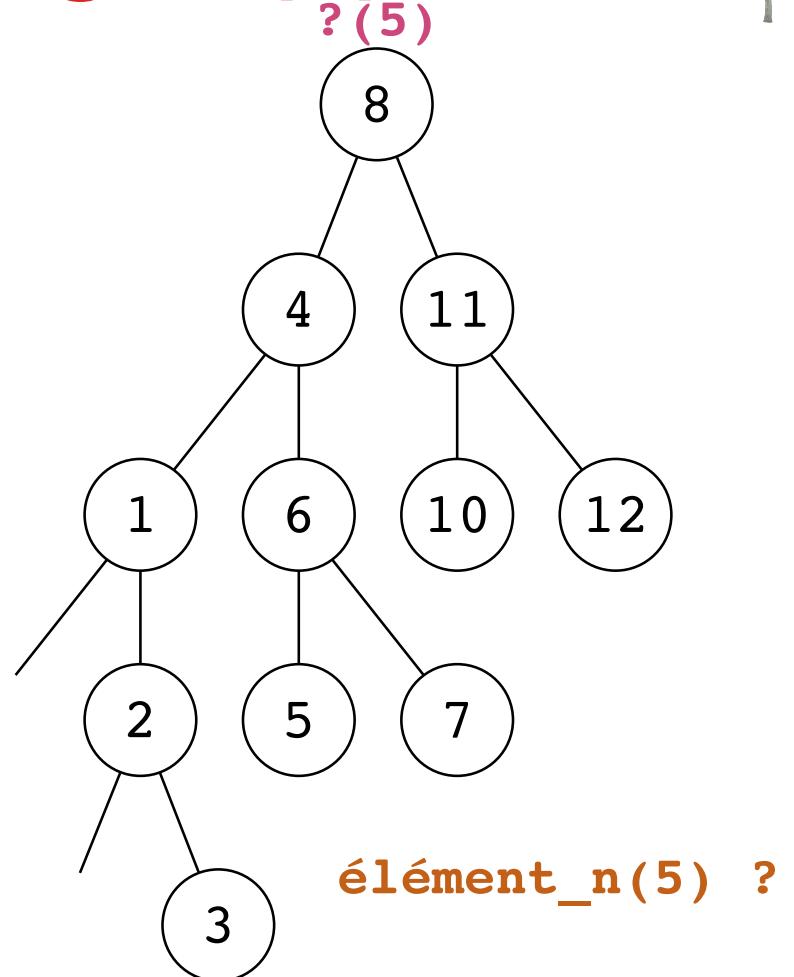
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche,n)
        sinon, (n > t)
            retourner élément_n(r.droit,n-t-1)
```





# Elément de rang n (2)

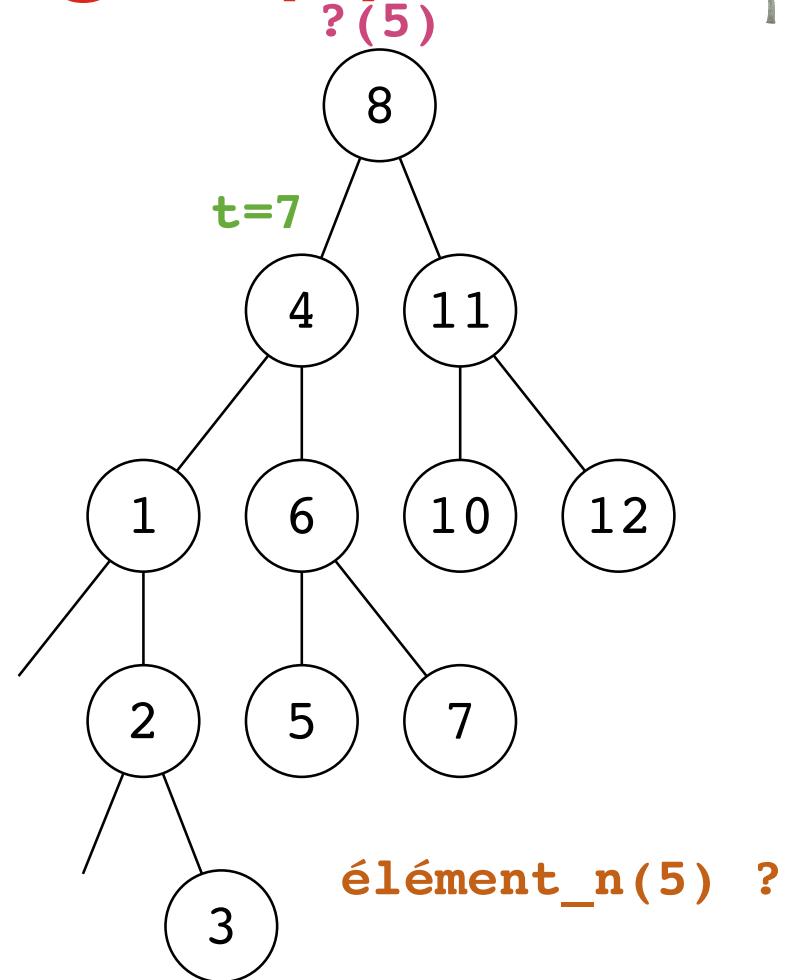
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# Elément de rang n (2)

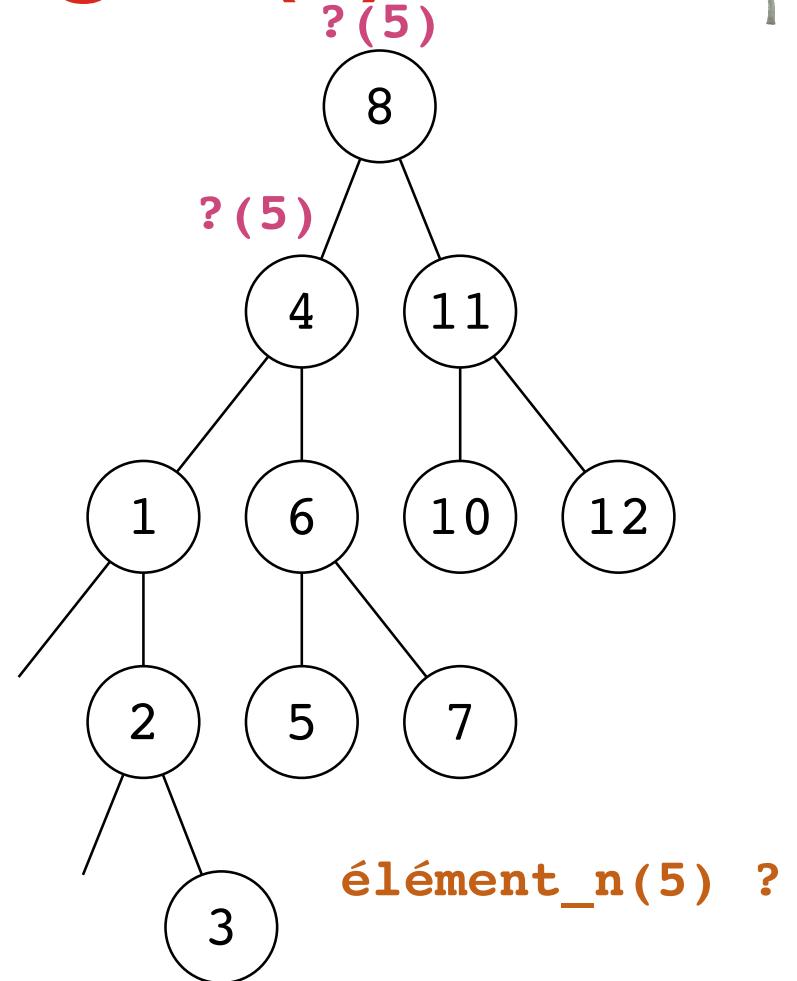
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# Elément de rang n (2)

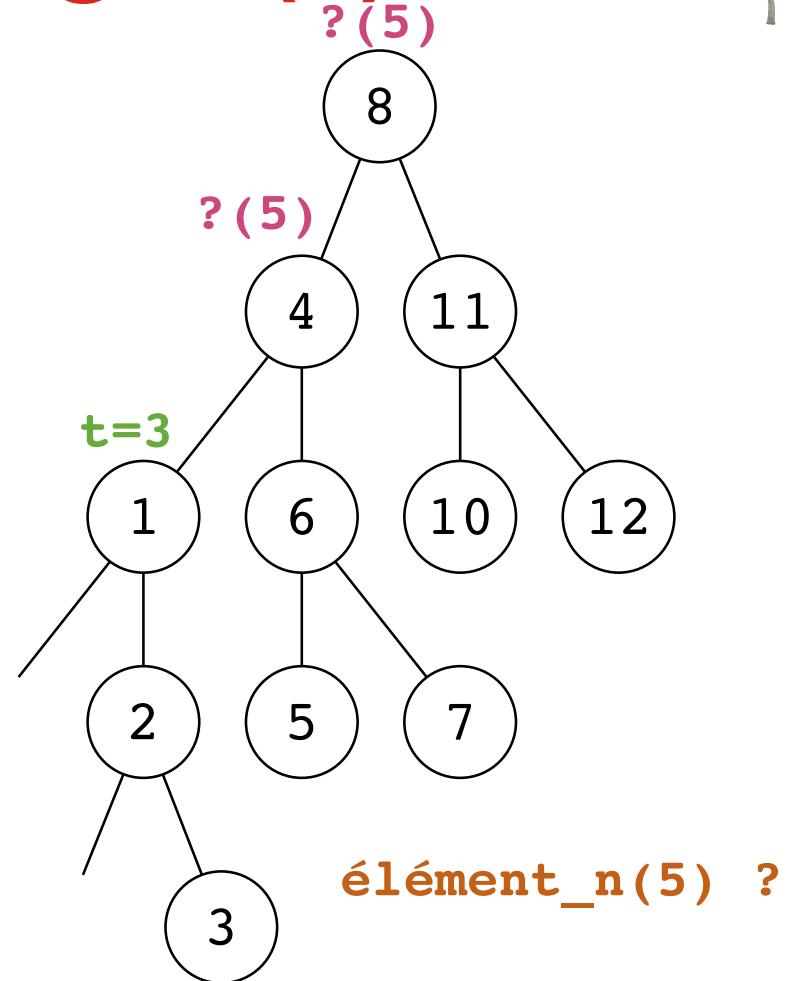
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# Elément de rang n (2)

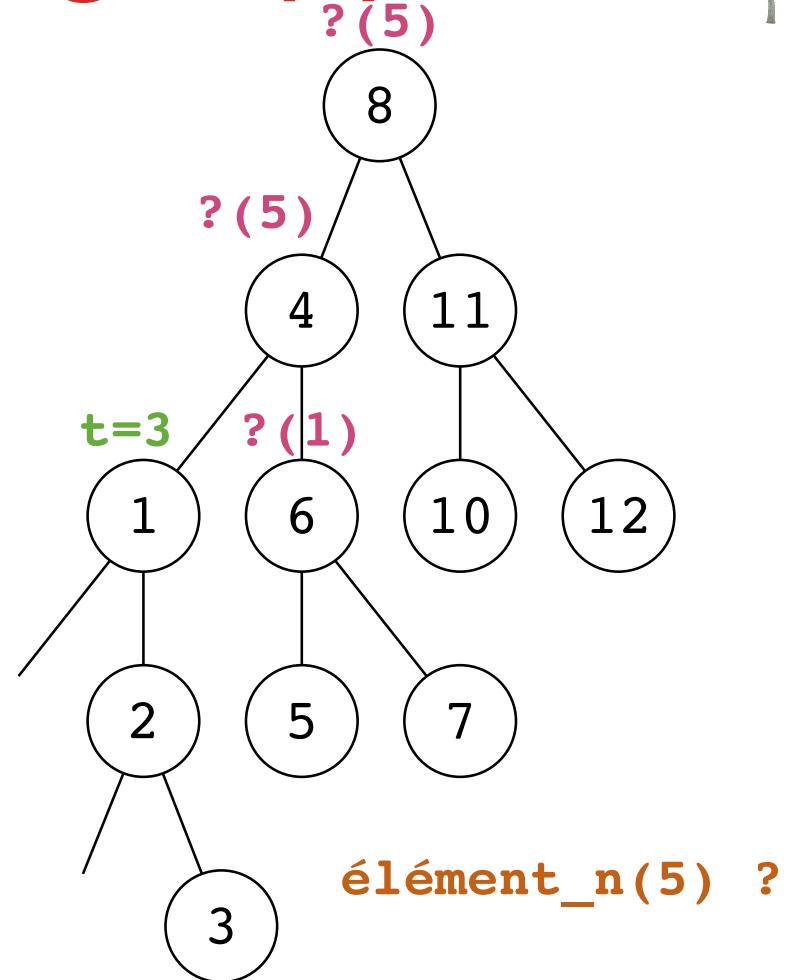
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# Elément de rang n (2)

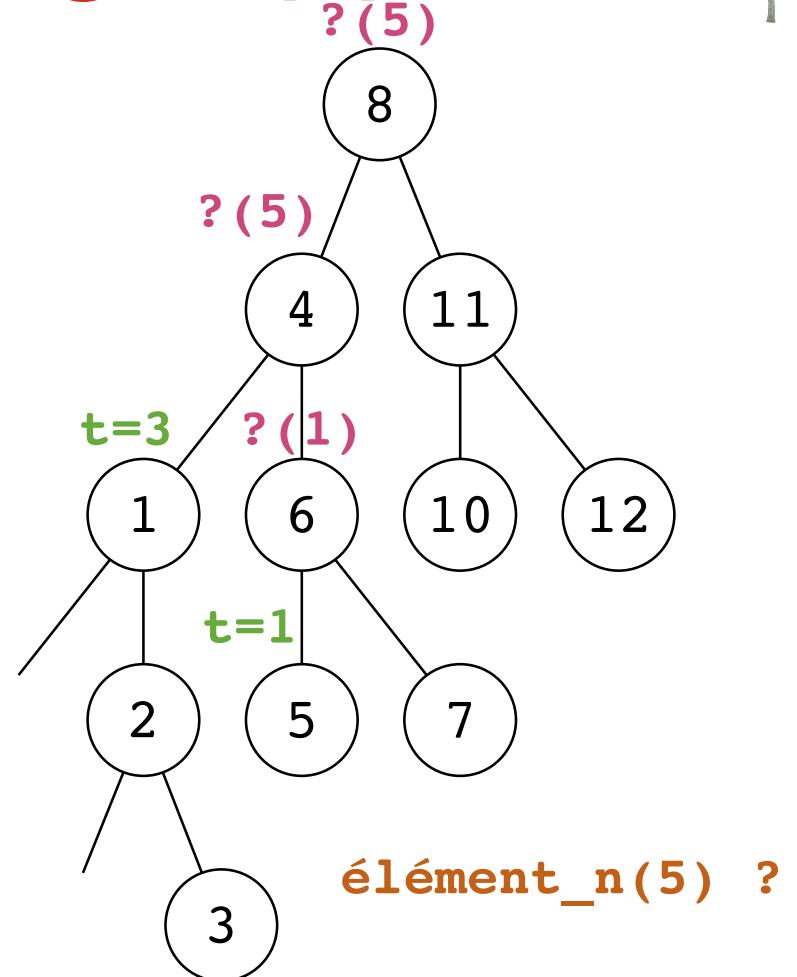
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# Elément de rang n (2)

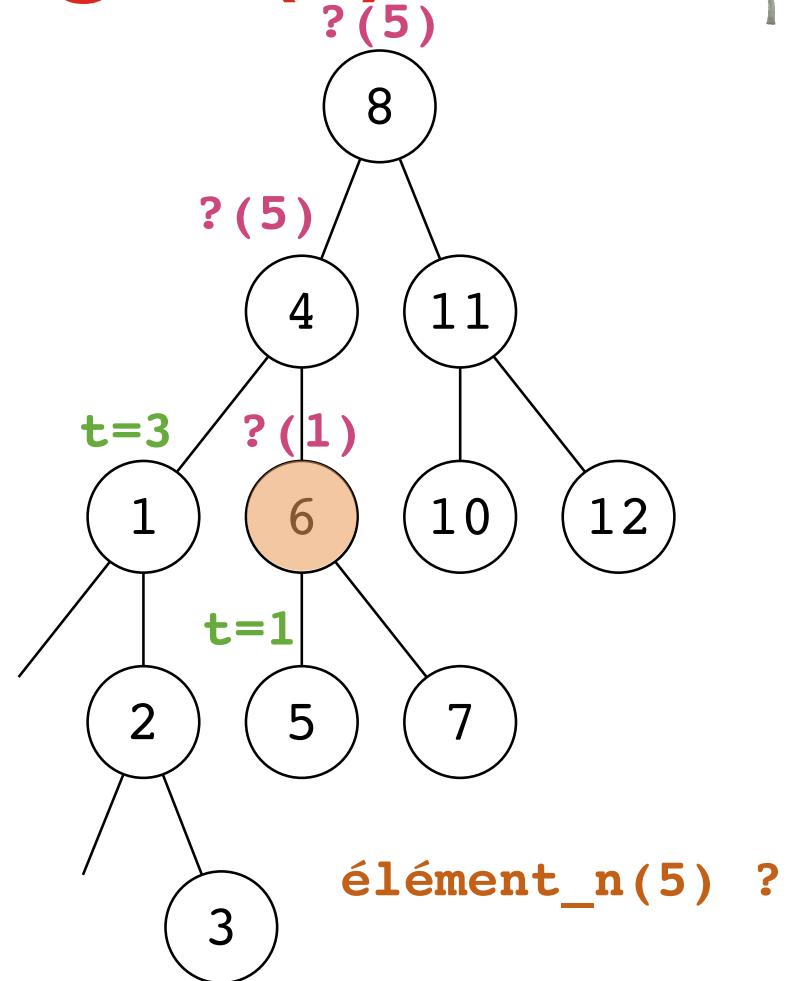
```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# Elément de rang n (2)

```
fonction élément_n (r, n)
    si n < 0 ou n ≥ taille(r)
        signaler erreur
    sinon,
        t ← taille(r.gauche)
        si n == t,
            retourner r
        sinon, si n < t,
            retourner élément_n(r.gauche, n)
        sinon, (n > t)
            retourner élément_n(r.droit, n-t-1)
```





# TDA Order statistic tree

- ABR fournissant les opérations
  - Calcul du rang
  - sélection par le rang ( $n^{\text{ième}}$  élément)

# TDA Order statistic tree

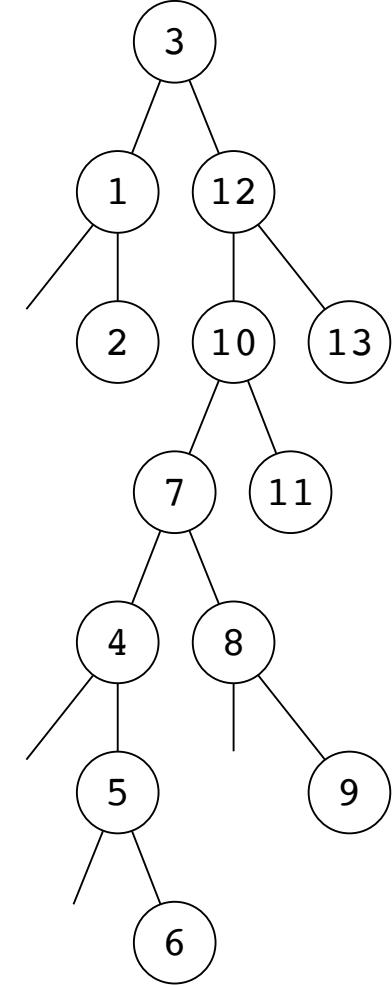


- ABR fournissant les opérations
  - Calcul du rang
  - sélection par le rang ( $n^{\text{ième}}$  élément)
- Ce TDA n'est pas fourni par la STL en C++. Si vous en avez besoin, il faudra le programmer vous-même.

# 13 . Complexité des opérations



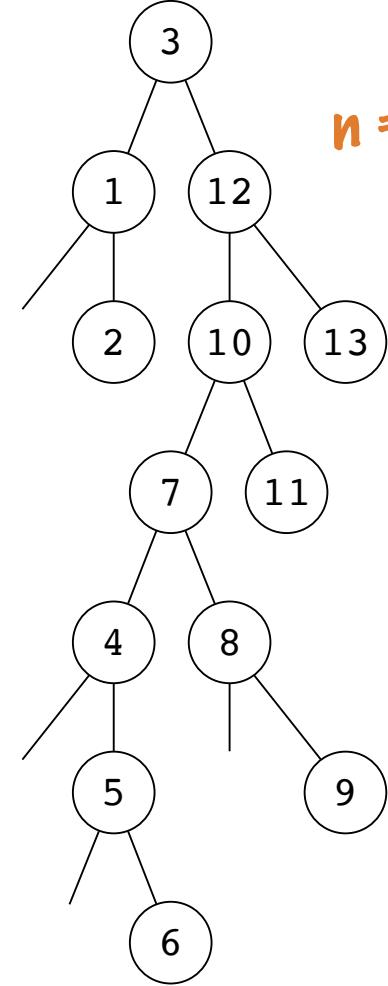
# Complexités pour un arbre de taille n et hauteur h



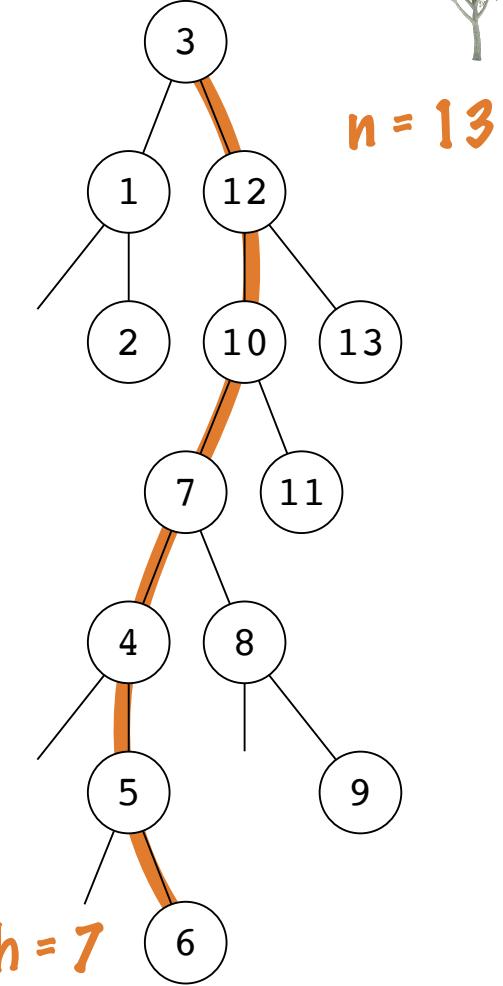
# Complexités pour un arbre de taille n et hauteur h



$n = 13$



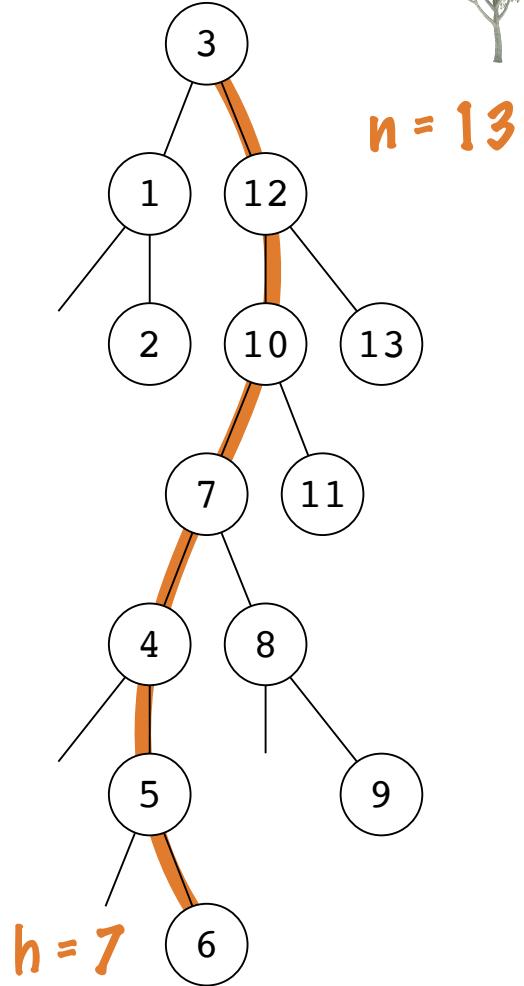
# Complexités pour un arbre de taille n et hauteur h



# Complexités pour un arbre de taille n et hauteur h



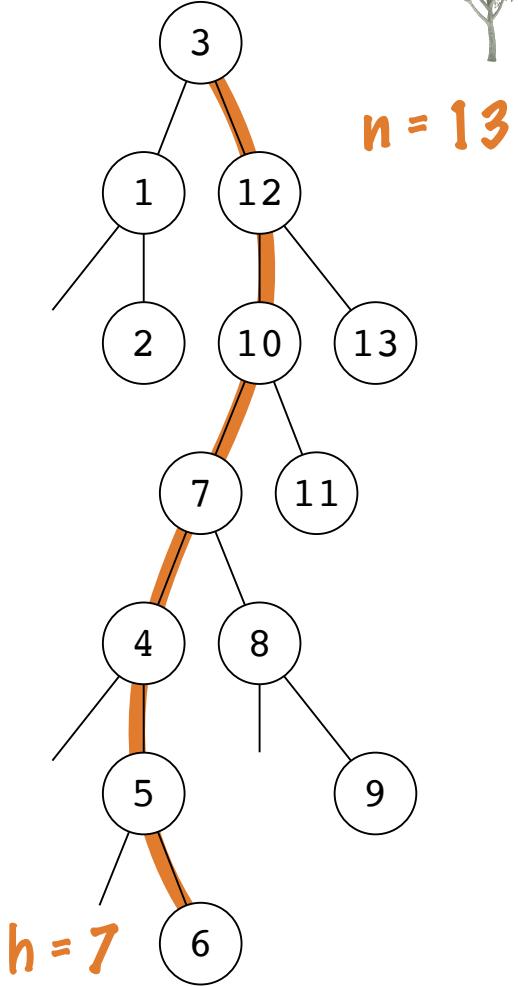
- Parcours en  $O(n)$



# Complexités pour un arbre de taille n et hauteur h



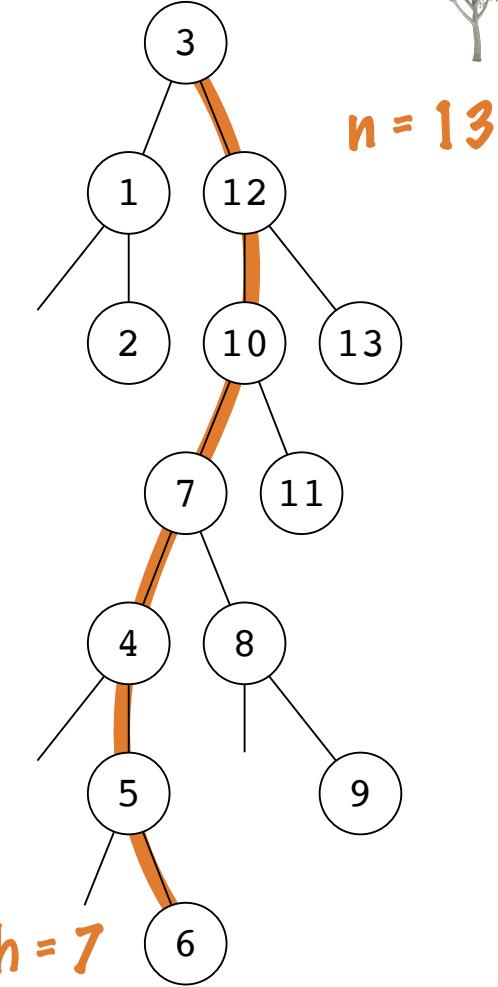
- Parcours en  $O(n)$
- Itération (avec lien parent)
  - sur tout l'arbre en  $O(n)$
  - suivant en  $O(1)$  en moyenne,  $O(h)$  au pire



# Complexités pour un arbre de taille n et hauteur h



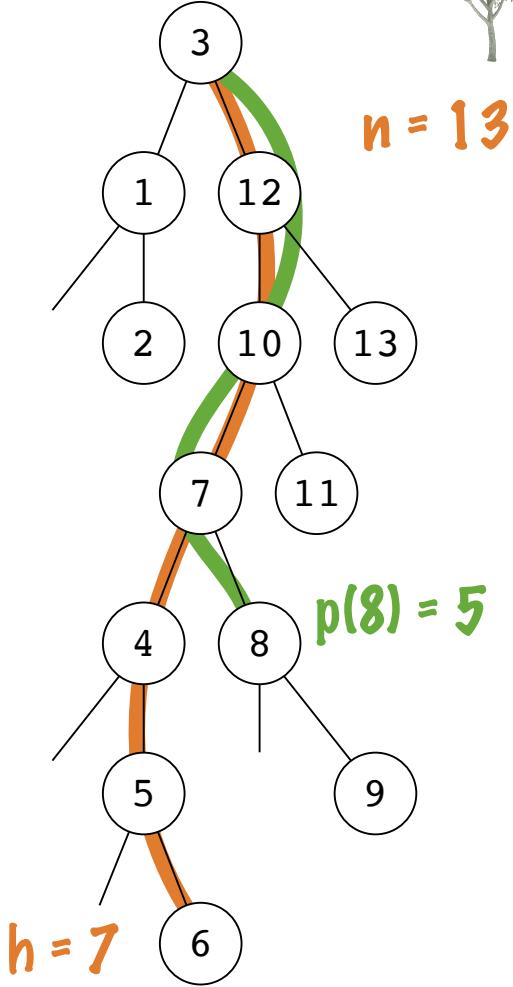
- Parcours en  $O(n)$
- Itération (avec lien parent)
  - sur tout l'arbre en  $O(n)$
  - suivant en  $O(1)$  en moyenne,  $O(h)$  au pire
- Recherche, insertion, suppression, minimum et maximum en  $O(p)$  avec  $p$  le niveau du noeud pertinent



# Complexités pour un arbre de taille n et hauteur h



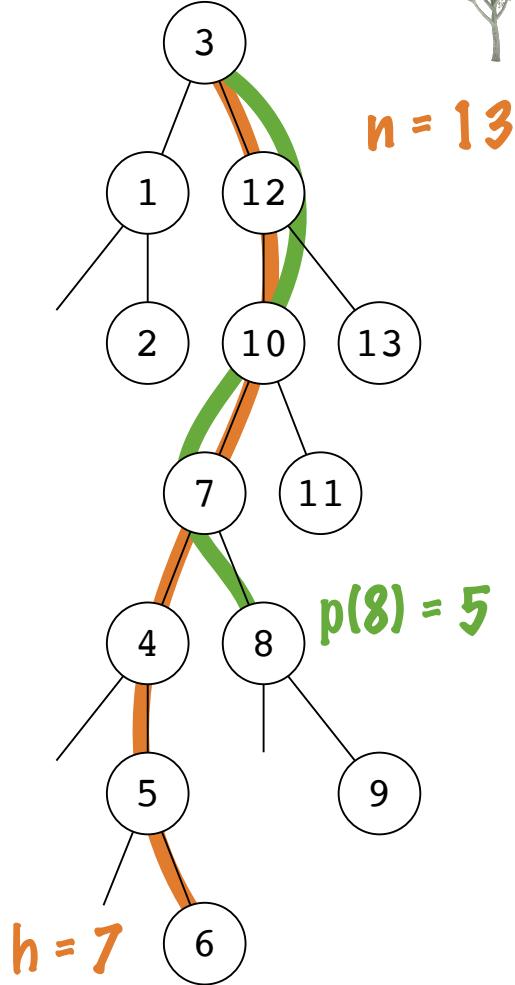
- Parcours en  $O(n)$
- Itération (avec lien parent)
  - sur tout l'arbre en  $O(n)$
  - suivant en  $O(1)$  en moyenne,  $O(h)$  au pire
- Recherche, insertion, suppression, minimum et maximum en  $O(p)$  avec  $p$  le niveau du noeud pertinent



# Complexités pour un arbre de taille n et hauteur h



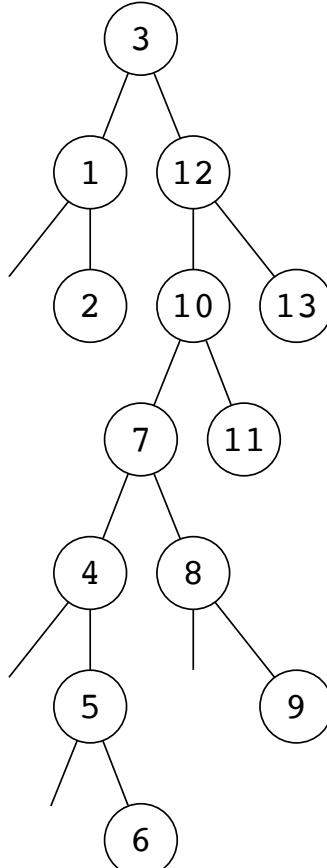
- Parcours en  $O(n)$
- Itération (avec lien parent)
  - sur tout l'arbre en  $O(n)$
  - suivant en  $O(1)$  en moyenne,  $O(h)$  au pire
- Recherche, insertion, suppression, minimum et maximum en  $O(p)$  avec  $p$  le niveau du noeud pertinent
- Quelle sont les relations entre  $n$ ,  $h$  et  $p$  ?



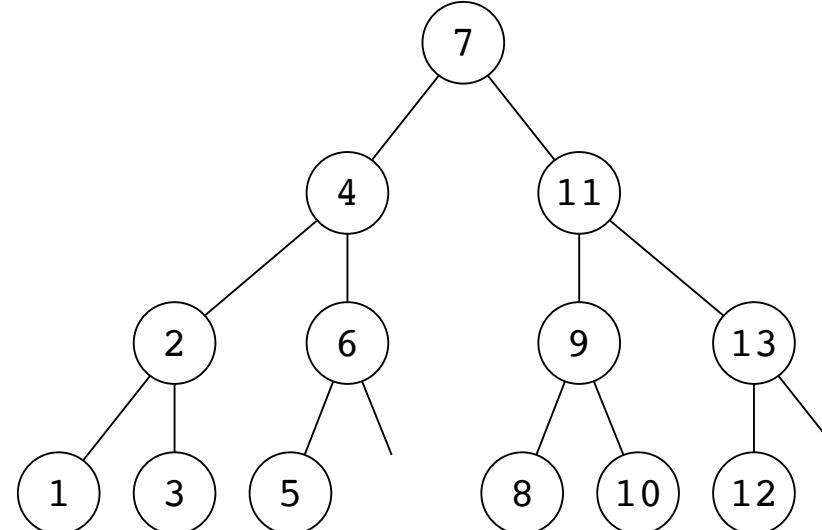


# n, h et p

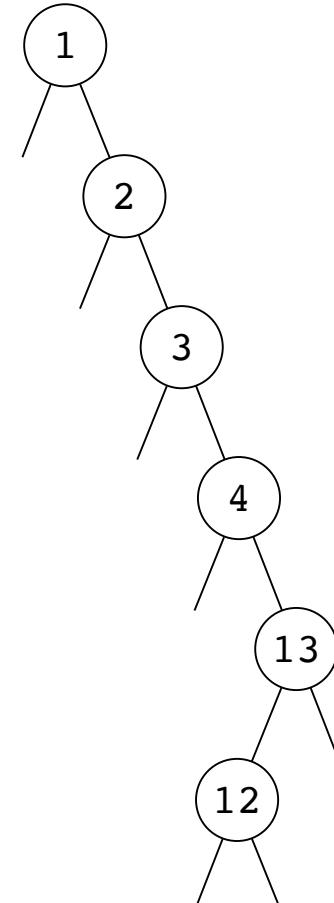
- Quelconque



- Plein



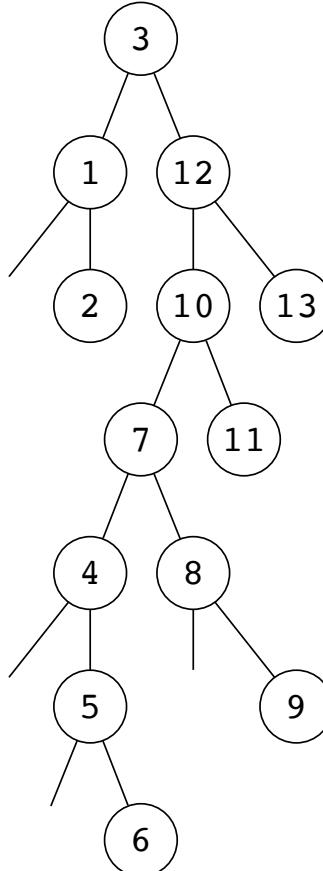
- Dégénéré



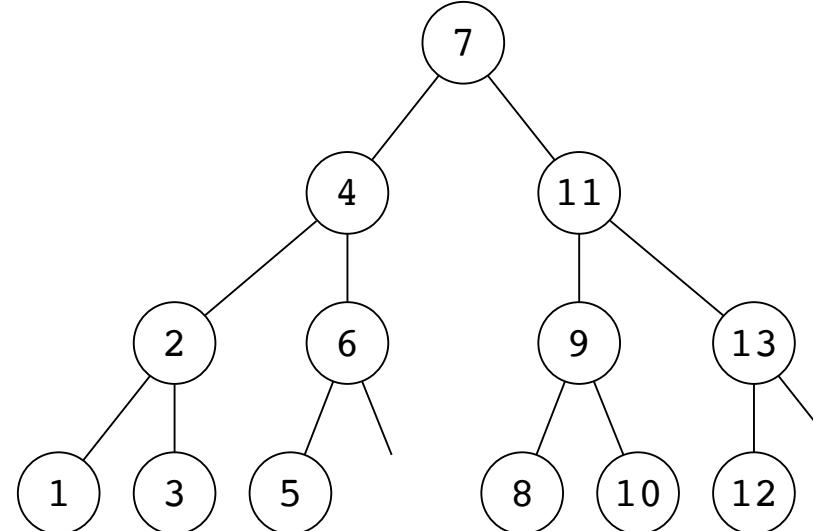


# n, h et p

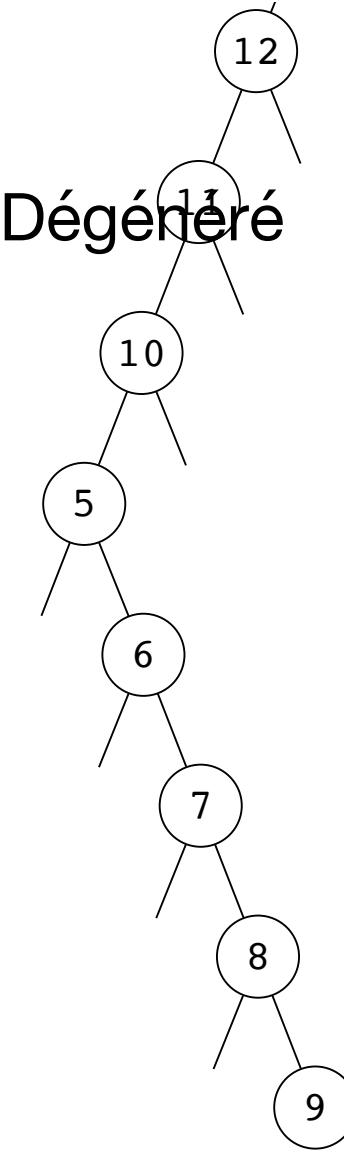
- Quelconque



- ## • Plein



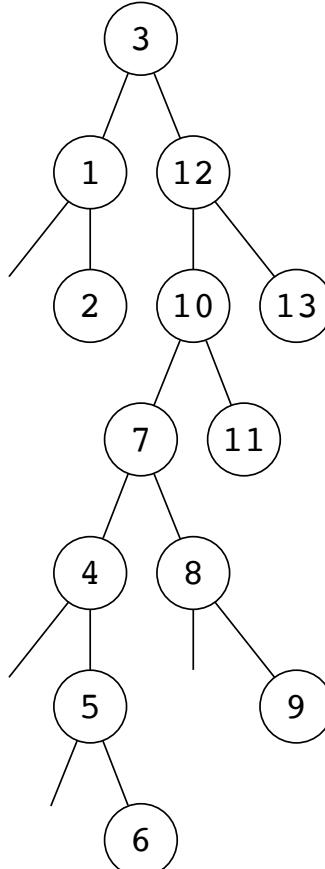
- ## • Dégéné<sup>re</sup>



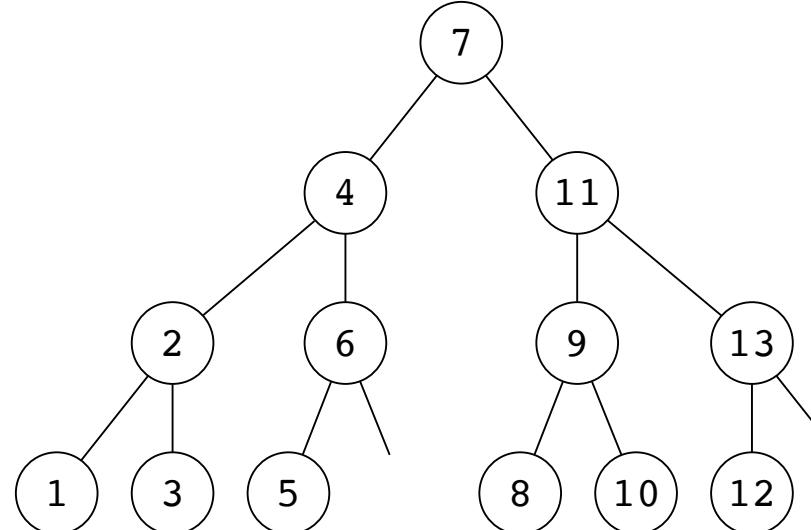


# n, h et p

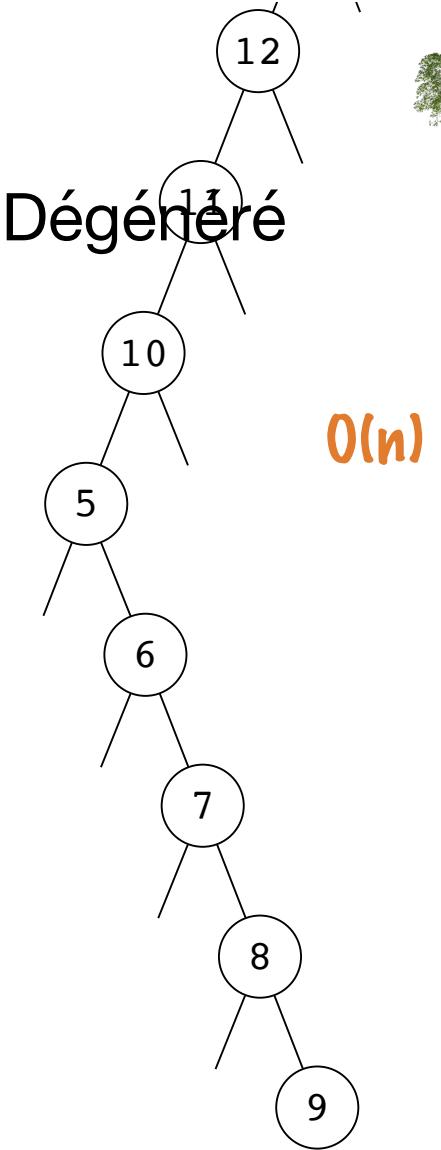
- Quelconque



- Plein



- Dégénéré

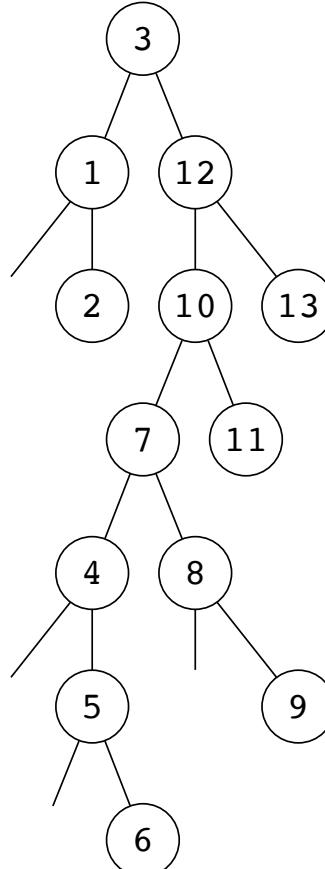


$O(n)$

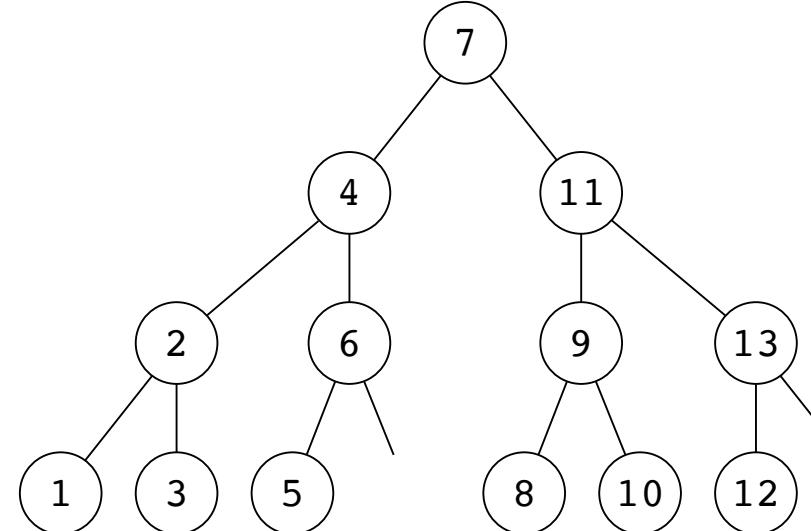


# n, h et p

- Quelconque

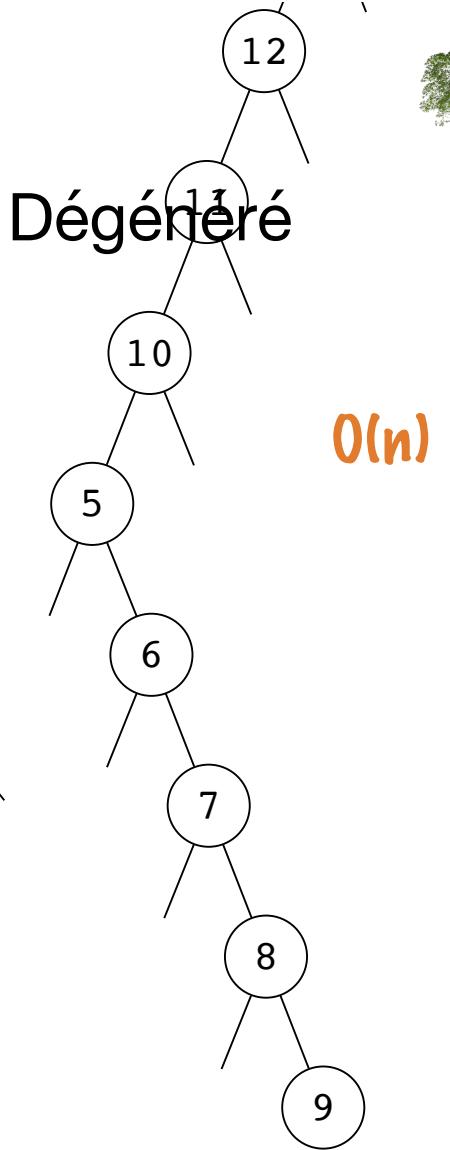


- Plein



$O(\log(n))$

- Dégénéré

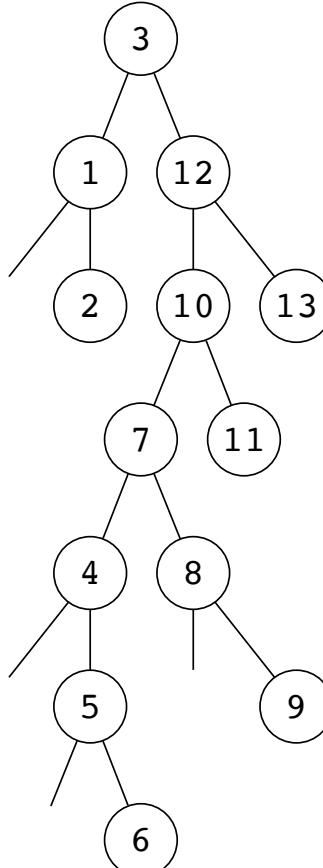


$O(n)$

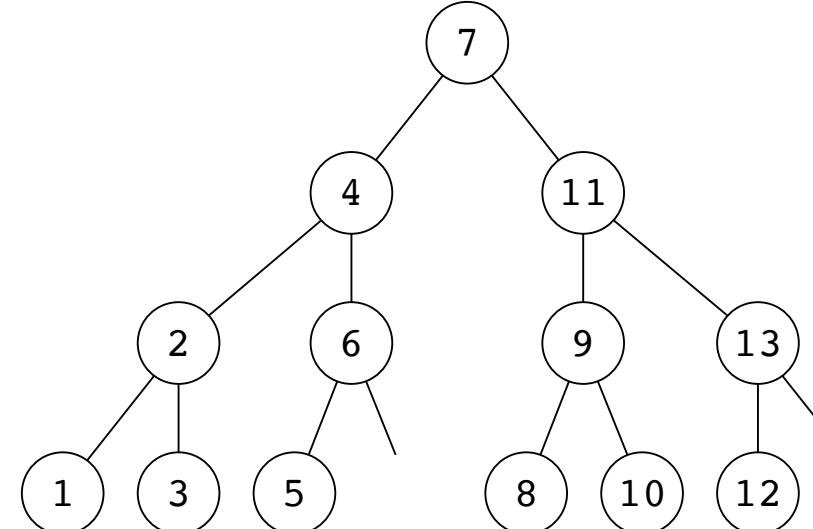


# n, h et p

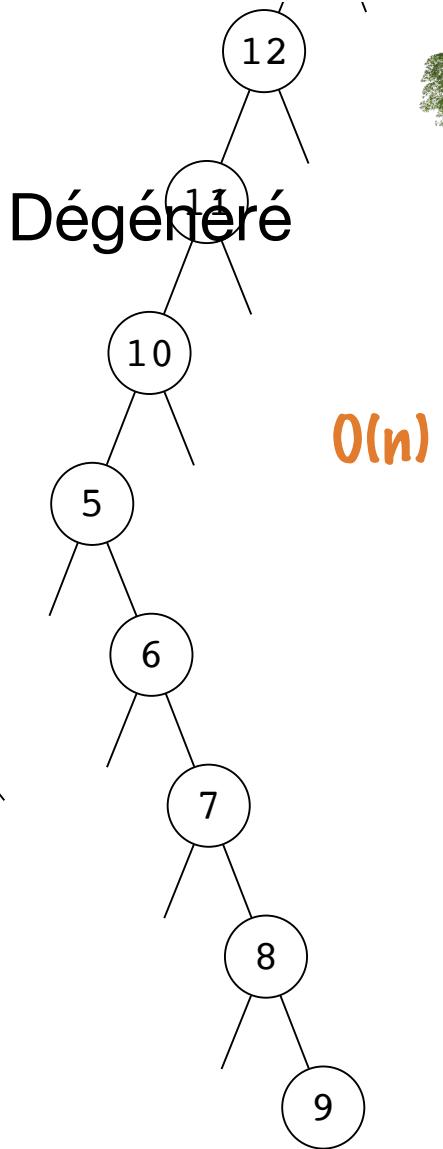
- Quelconque



- Plein



- Dégénéré

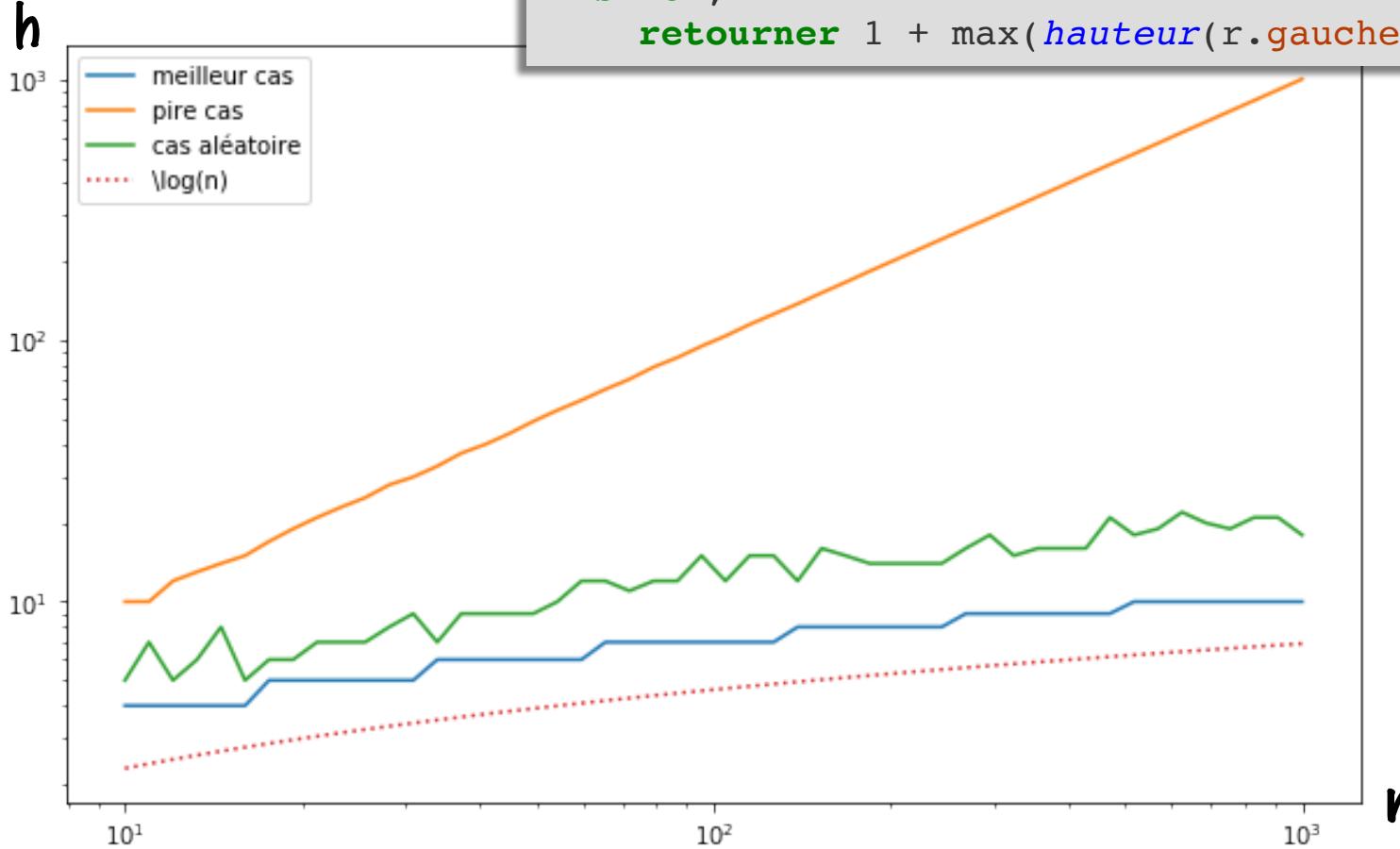




# Hauteurs

```
fonction hauteur (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + max(hauteur(r.gauche), hauteur(r.droit))
```

```
fonction hauteur (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + max(hauteur(r.gauche), hauteur(r.droit))
```



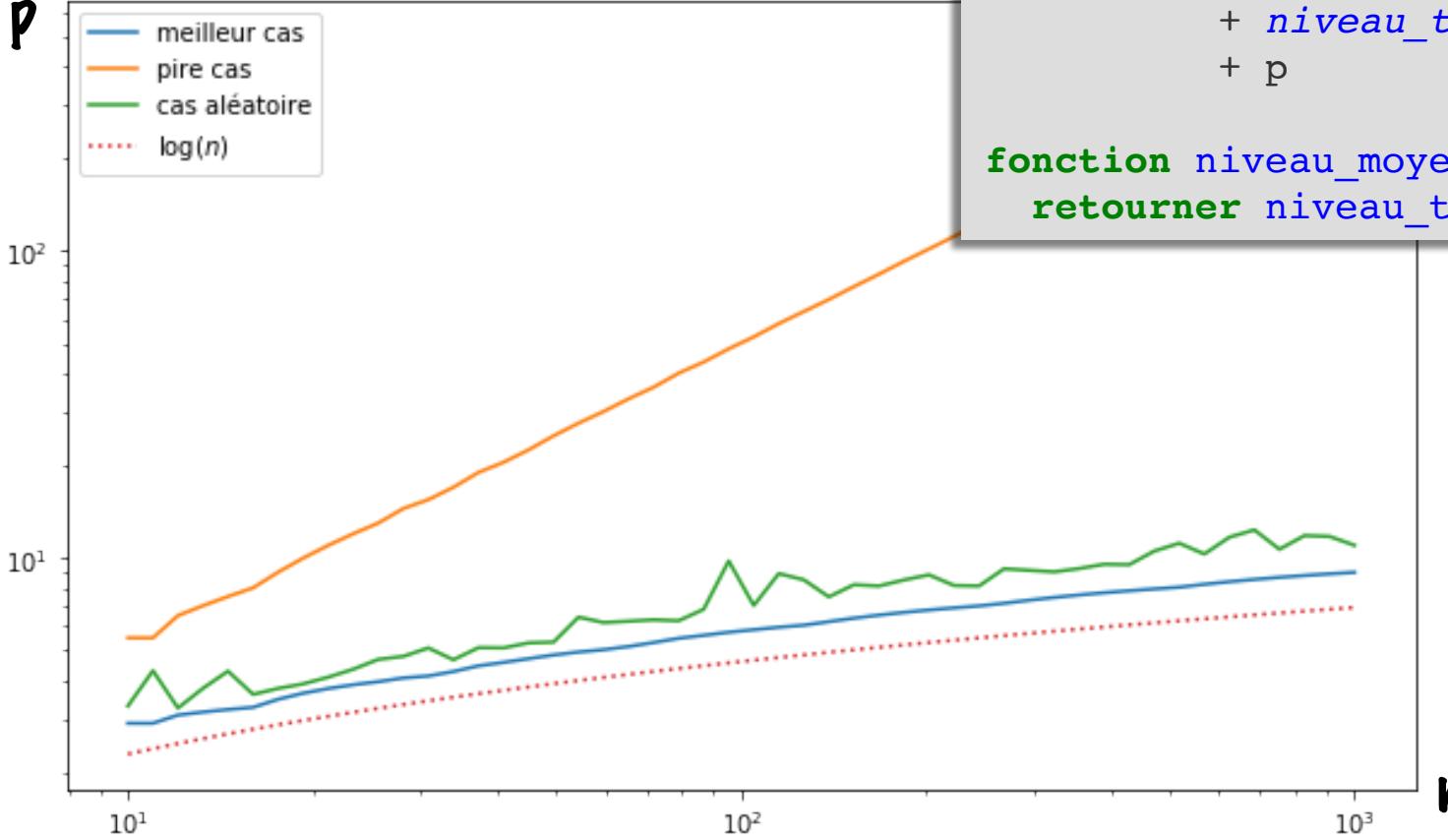


# Niveaux moyens

```
fonction niveau_total (r, p)
    si r == Ø,
        retourner 0
    sinon,
        retourner niveau_total(r.gauche, p+1)
            + niveau_total(r.droit, p+1)
            + p

fonction niveau_moyen (r)
    retourner niveau_total(r,1) / taille(r)
```

# Niveaux



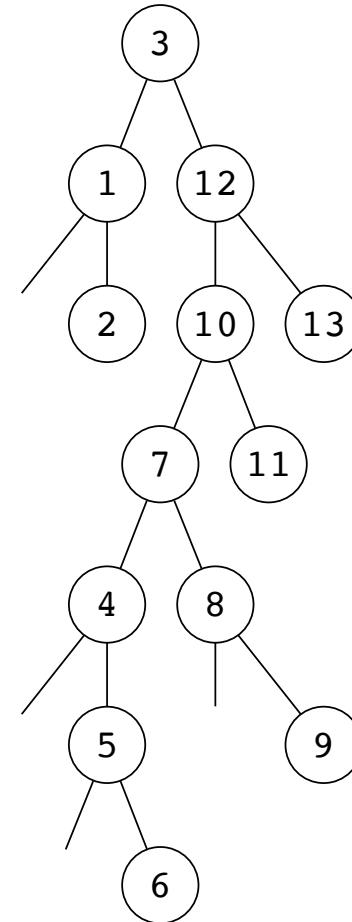
```
fonction niveau_total (r, p)
    si r == Ø,
        retourner 0
    sinon,
        retourner niveau_total(r.gauche, p+1)
            + niveau_total(r.droit, p+1)
            + p

fonction niveau_moyen (r)
    retourner niveau_total(r,1) / taille(r)
```



# O(log(n))

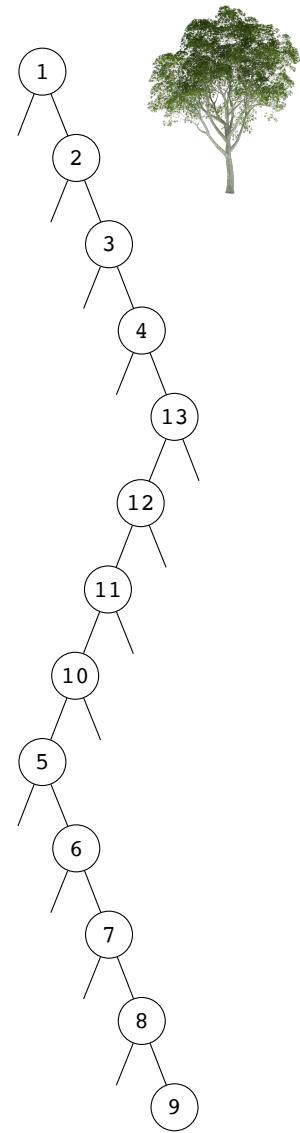
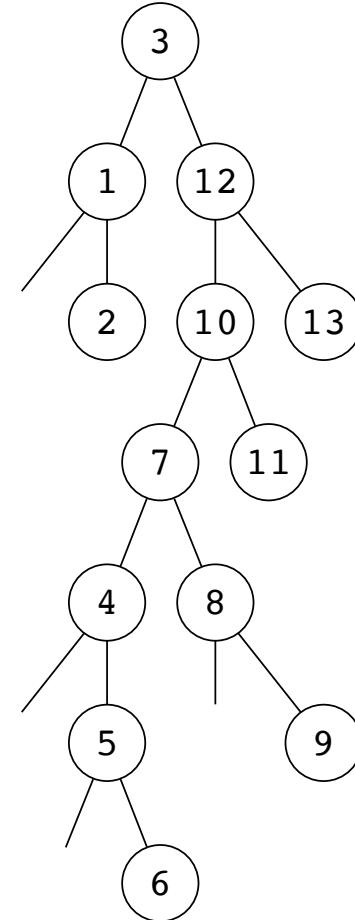
- En moyenne, h et p sont de l'ordre log(n)



# O(log(n))

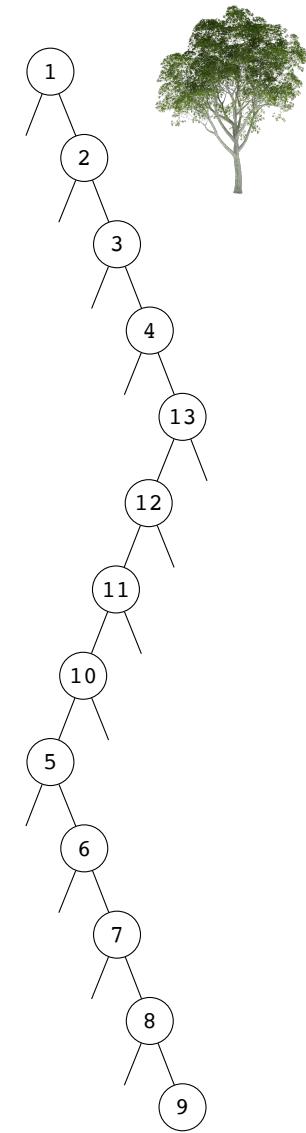
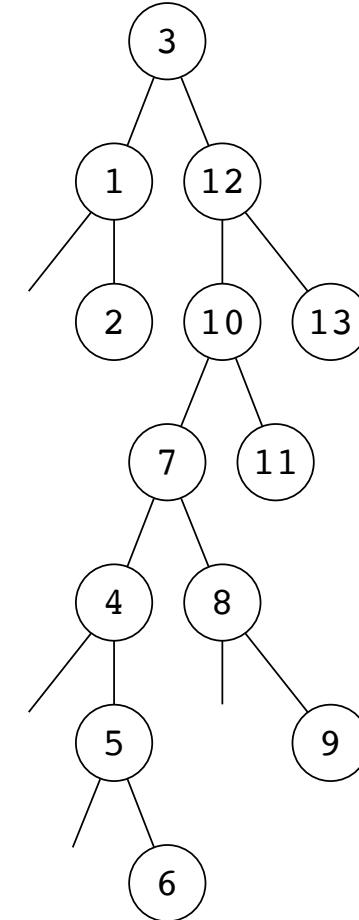
- En moyenne, h et p sont de l'ordre log(n)
- Au pire, ils sont de l'ordre de n

...



# O(log(n))

- En moyenne, h et p sont de l'ordre  $\log(n)$
- Au pire, ils sont de l'ordre de  $n$
- ...
- Mais le pire cas est un cas courant
  - Entrée triée
  - Entrée à l'envers
  - Entrés triée par morceaux
  - ...

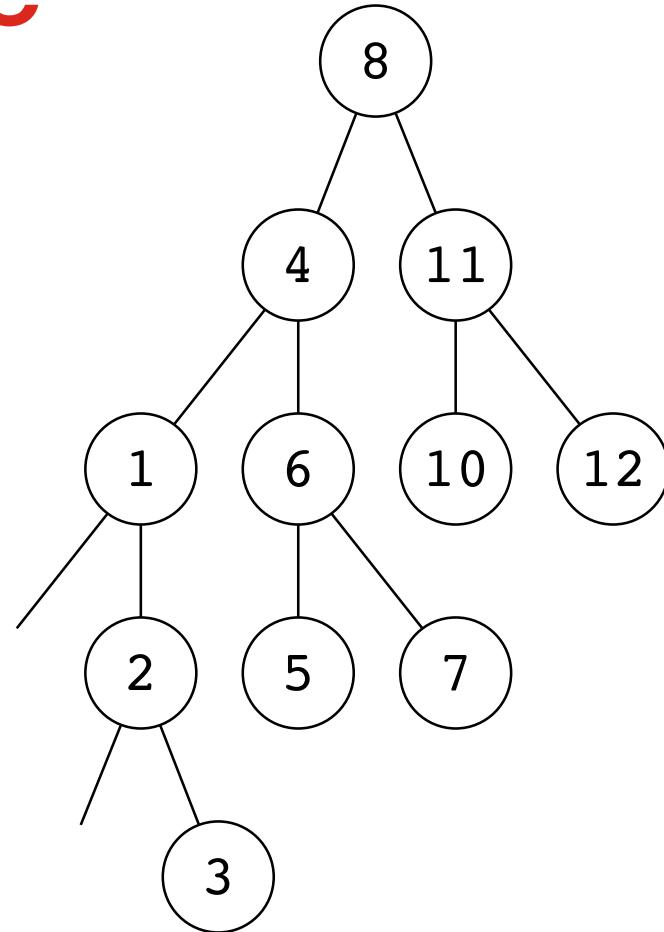




# Equilibre

- Équilibre d'un noeud

```
fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
```

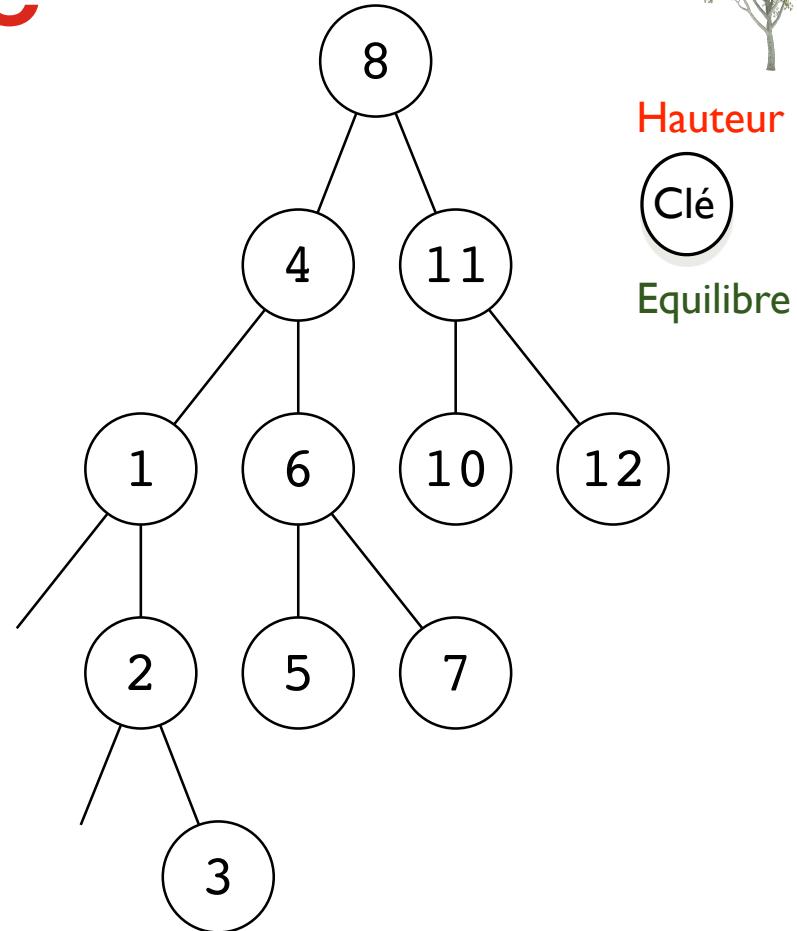




# Equilibre

- Équilibre d'un noeud

```
fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
```

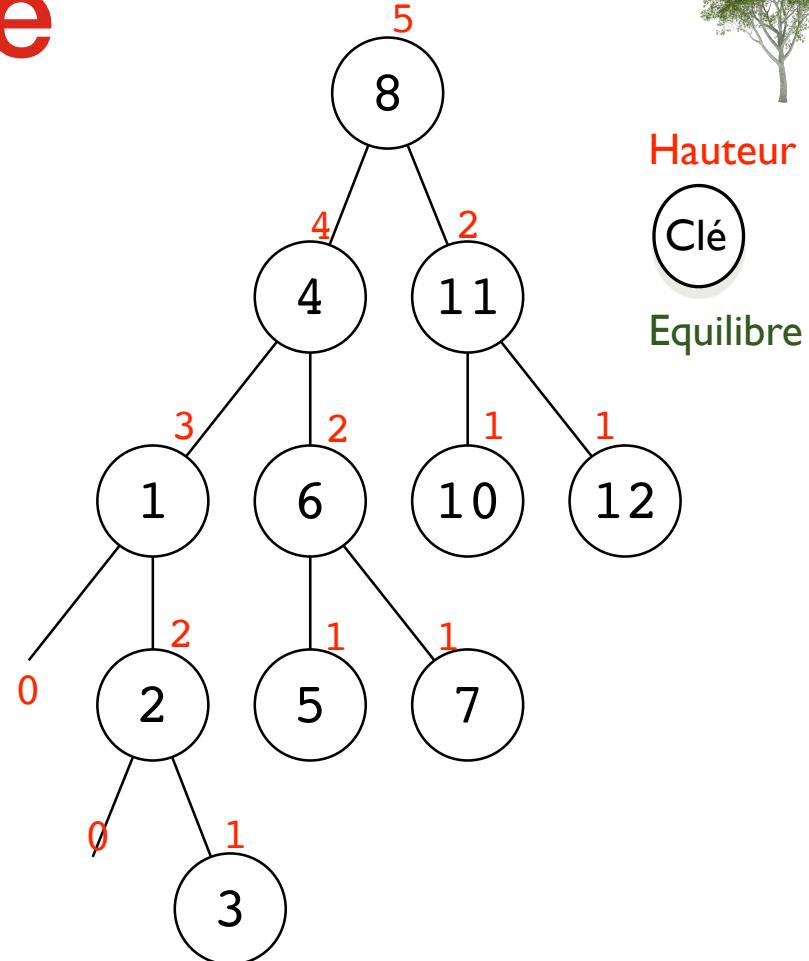




# Equilibre

- Équilibre d'un noeud

```
fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
```

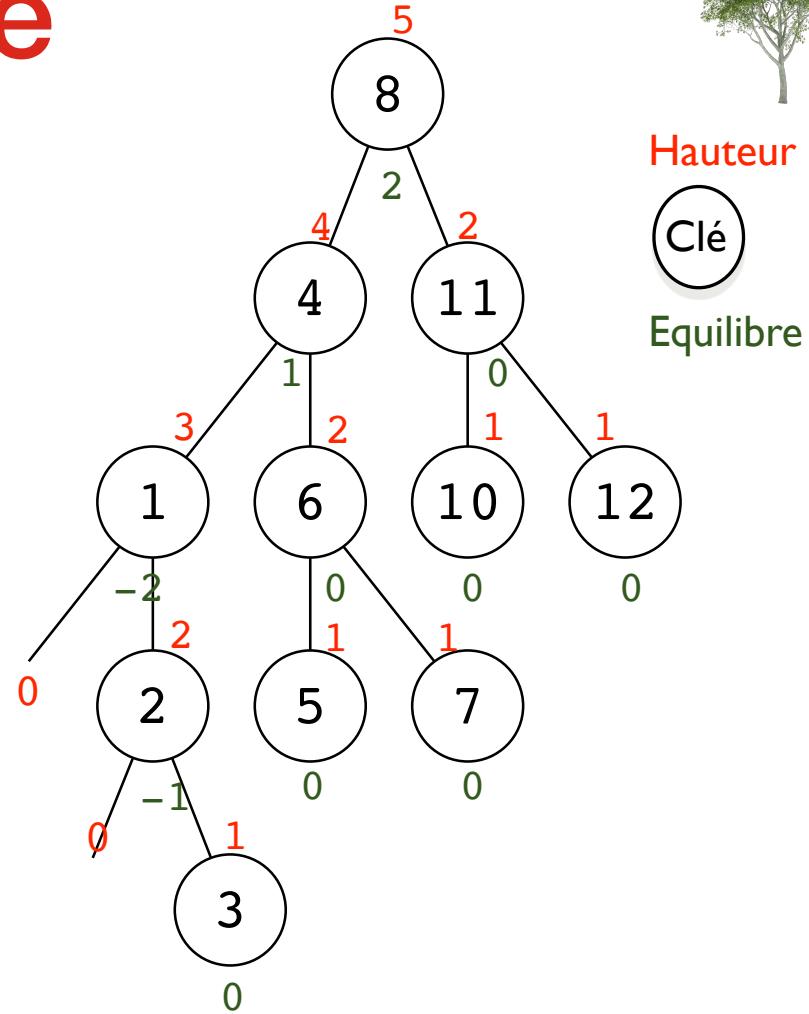




# Equilibre

- Équilibre d'un noeud

```
fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
```





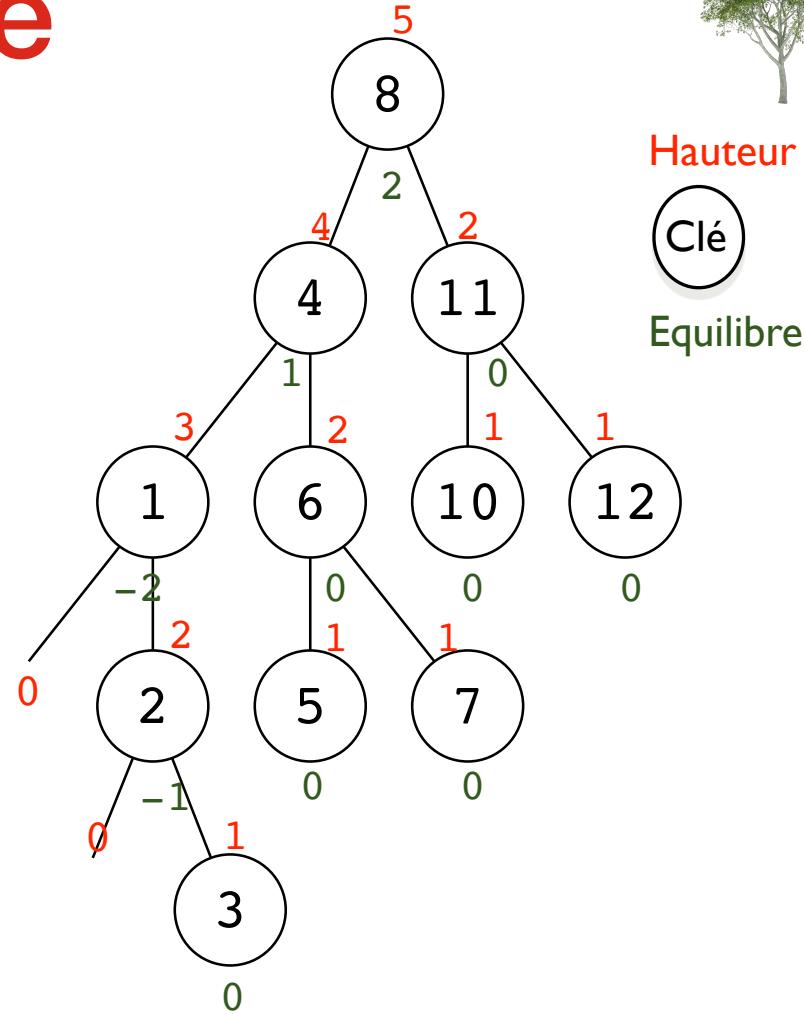
# Equilibre

- Équilibre d'un noeud

```

fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
    
```

- Un arbre est équilibré si  
 $-1 \leq \text{équilibre}(n) \leq 1$





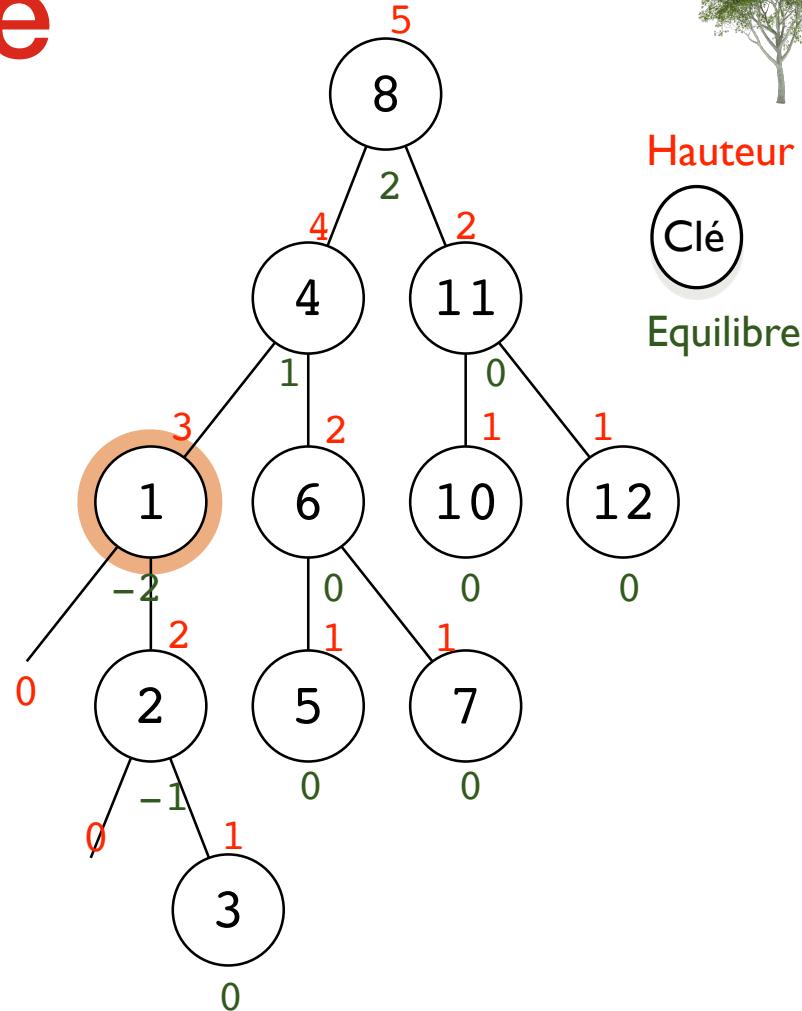
# Equilibre

- Équilibre d'un noeud

```

fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
    
```

- Un arbre est équilibré si  
 $-1 \leq \text{équilibre}(n) \leq 1$





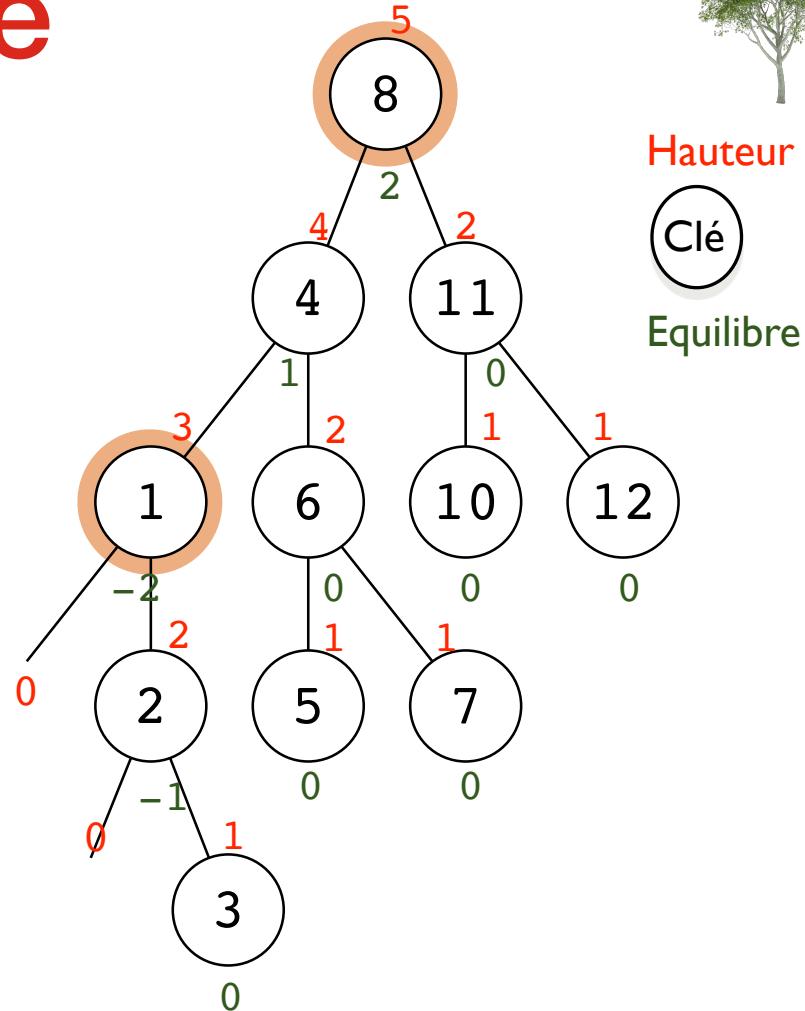
# Equilibre

- Équilibre d'un noeud

```

fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche)
            - hauteur(r.droit)
    
```

- Un arbre est équilibré si  
 $-1 \leq \text{équilibre}(n) \leq 1$



# Techniques d'équilibrage





# Techniques d'équilibrage

- Sur demande

# Techniques d'équilibrage



- Sur demande
- Automatiquement à chaque modification - arbres AVL, arbres rouge-noir, arbres d'Andersson, ...



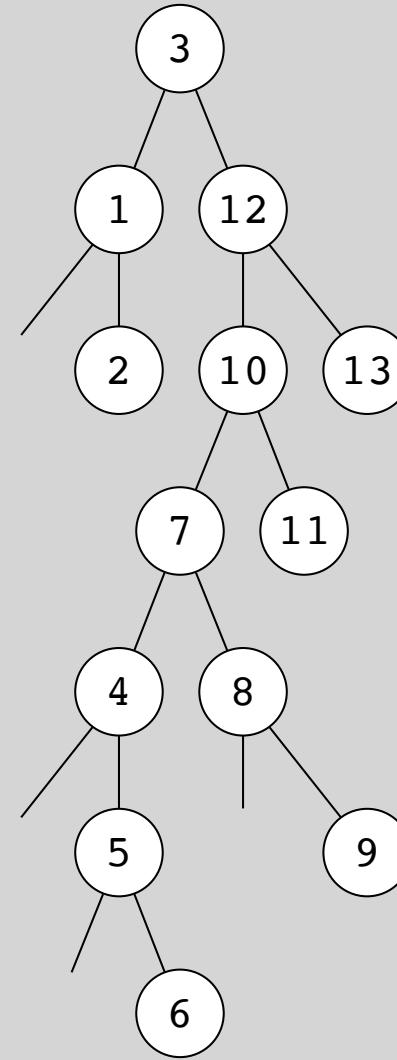
# Techniques d'équilibrage

- Sur demande
- Automatiquement à chaque modification - arbres AVL, arbres rouge-noir, arbres d'Andersson, ...
- Equilibré par construction mais nombre d'enfants variables - arbres 2-3, arbres 2-3-4, b-trees, ...



# Exercice

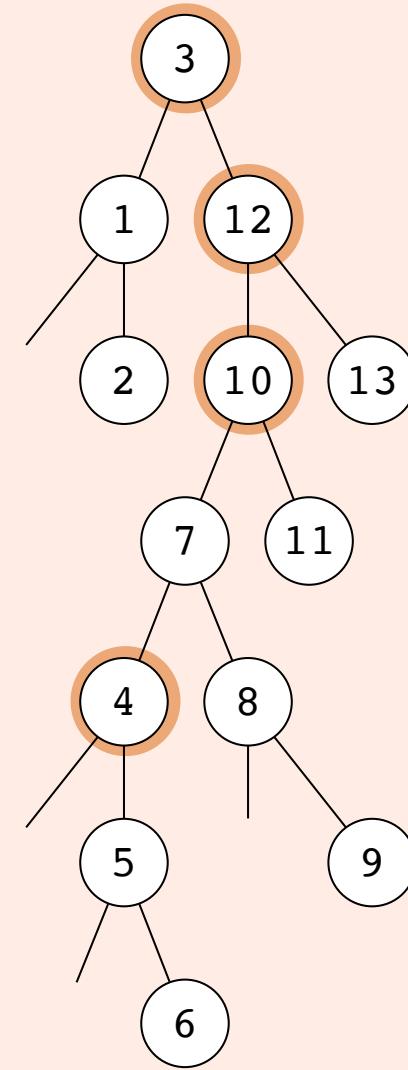
- Cet arbre est-il équilibré ?
- Si non, quels sont les noeuds où l'équilibre est rompu ?





# Solution

- Cet arbre n'est pas équilibré
- Les noeuds suivants rompent l'équilibre :
  - $\text{Eq}(3) = -4$
  - $\text{Eq}(12) = 4$
  - $\text{Eq}(10) = 3$
  - $\text{Eq}(4) = -2$

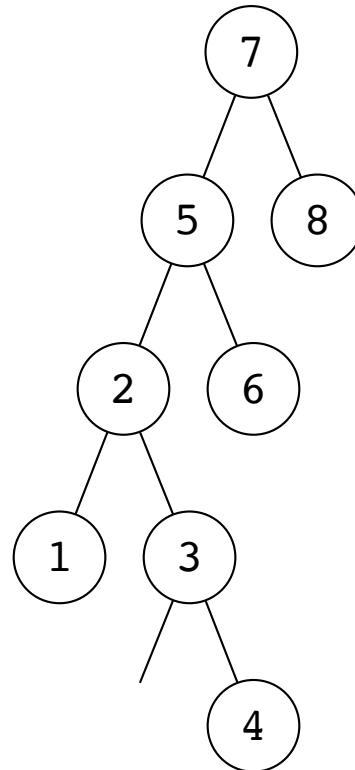


# 14 . Equilibrage par linéarisation et arborisation



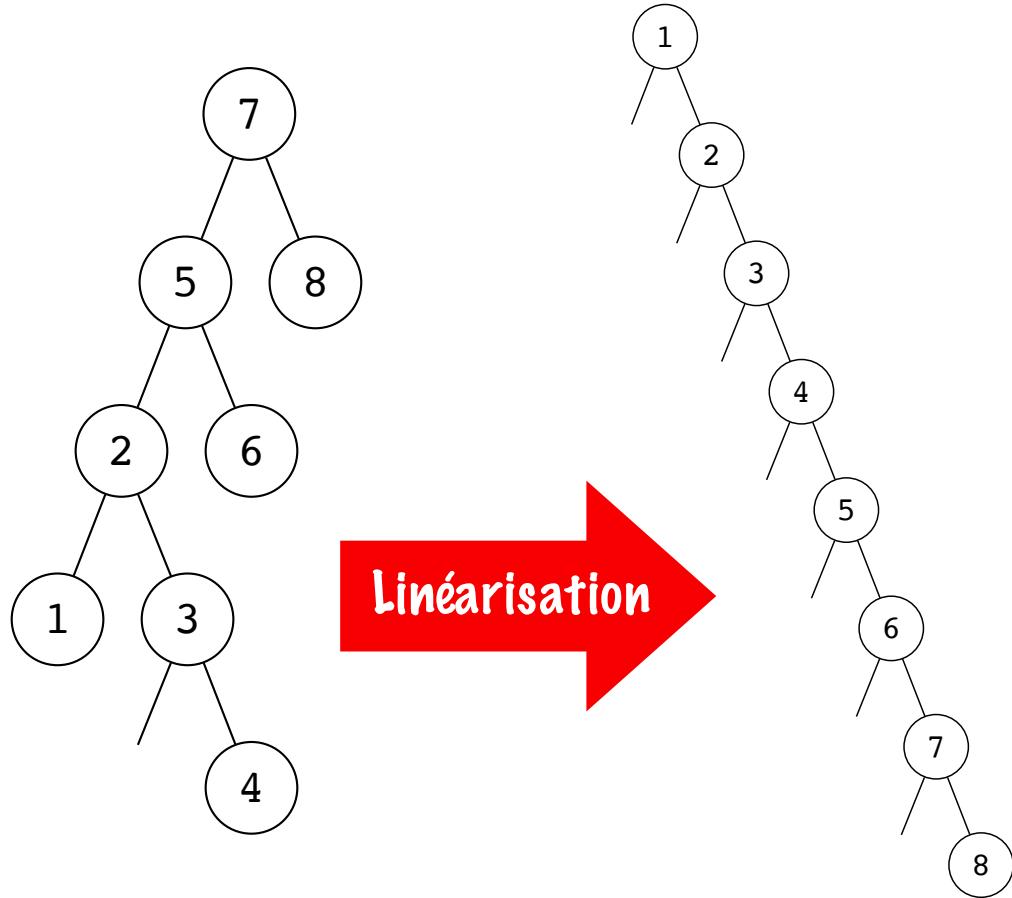


# Equilibrage à la demande



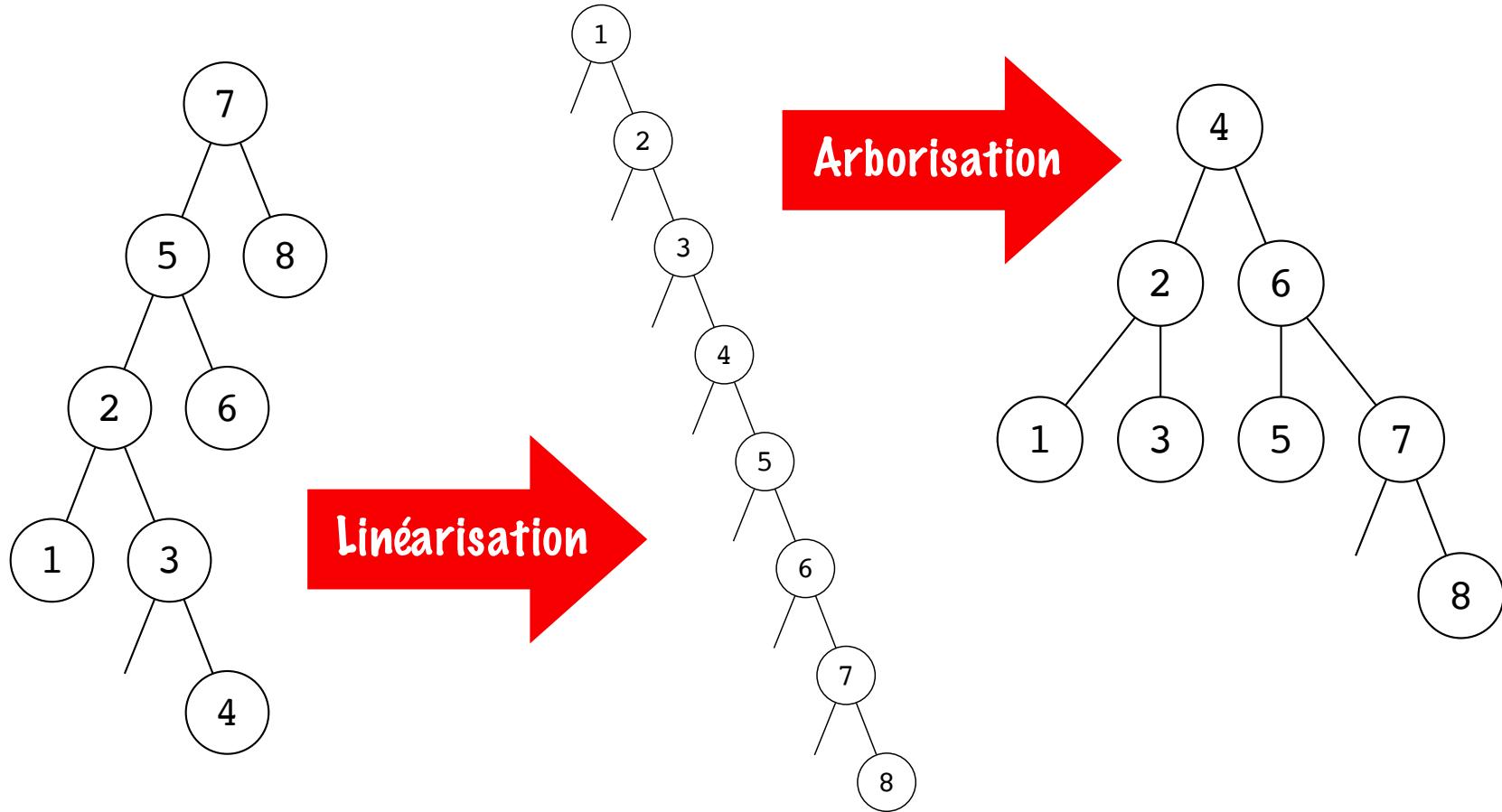


# Equilibrage à la demande





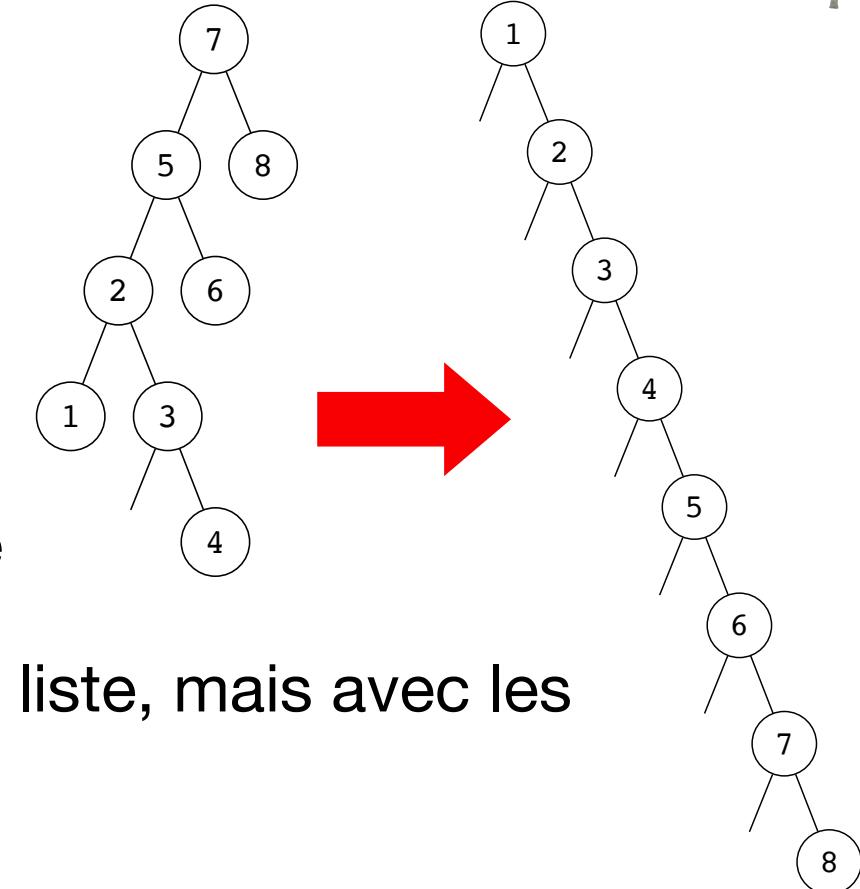
# Equilibrage à la demande





# Linéarisation

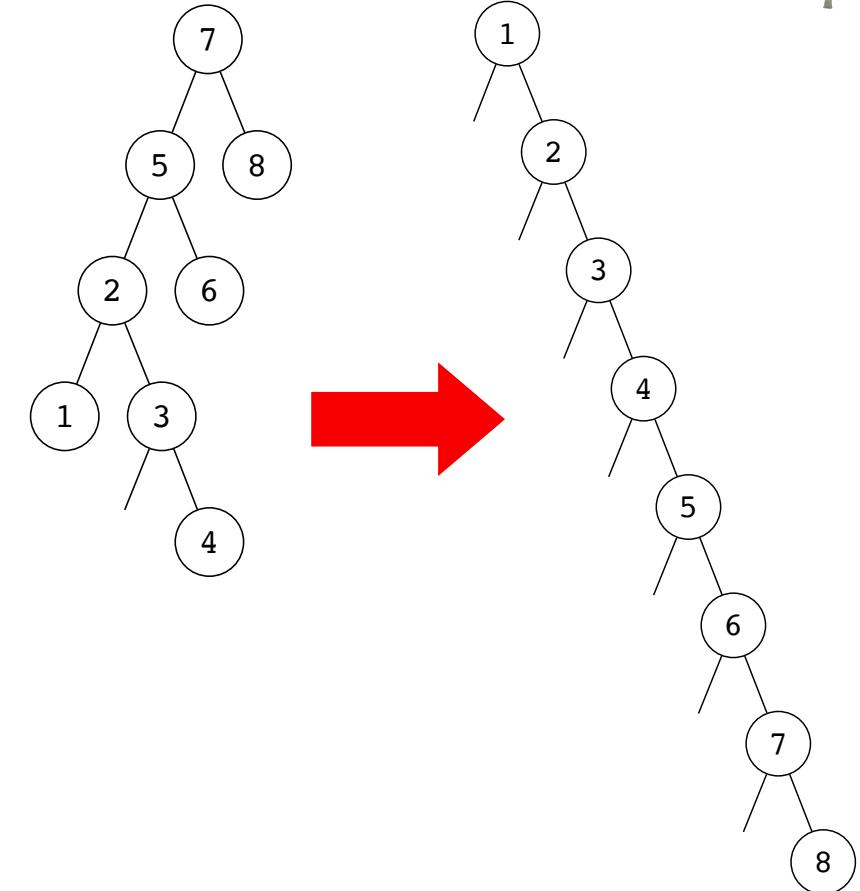
- Ré-organise l'arbre
- Ne touche qu'aux liens
- Arbre linéarisé sans enfant gauche
- Topologiquement équivalent à une liste, mais avec les noeuds d'arbre originaux





# Linéarisation

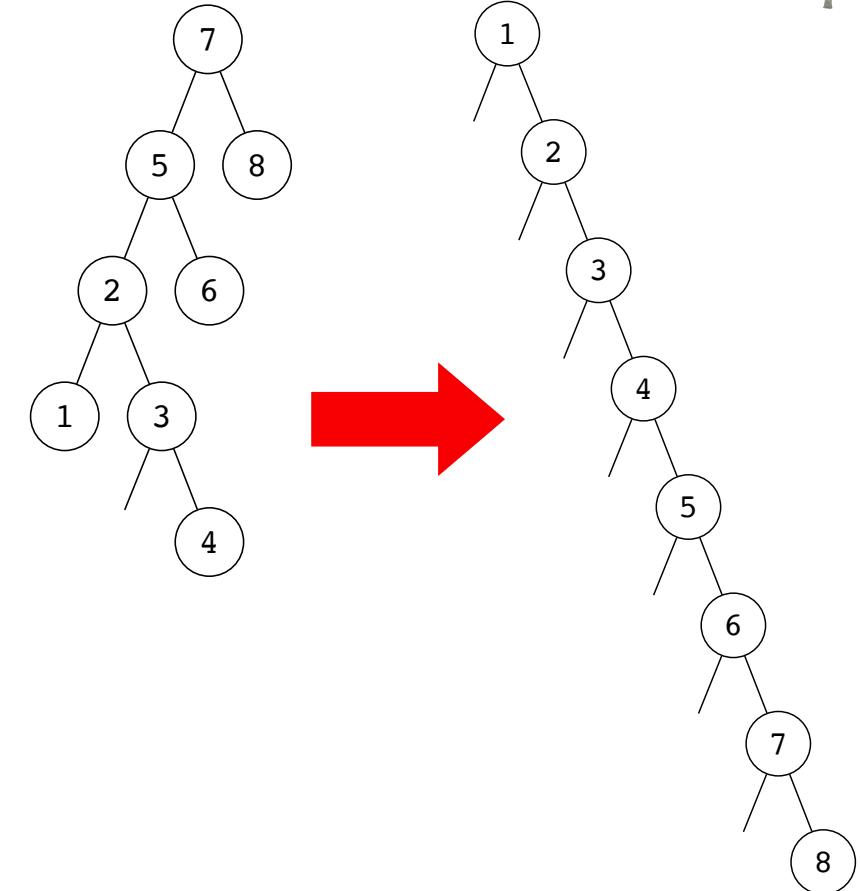
- Paramètres:





# Linéarisation

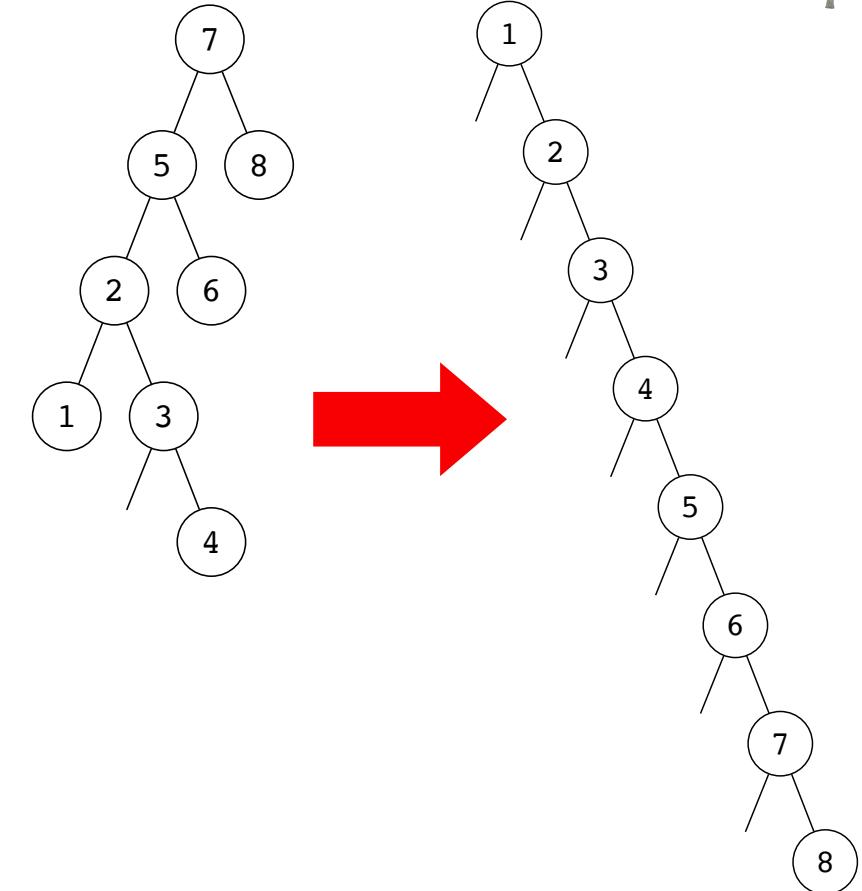
- Paramètres:
  - Un lien  $r$  vers la racine de l'arbre en entrée





# Linéarisation

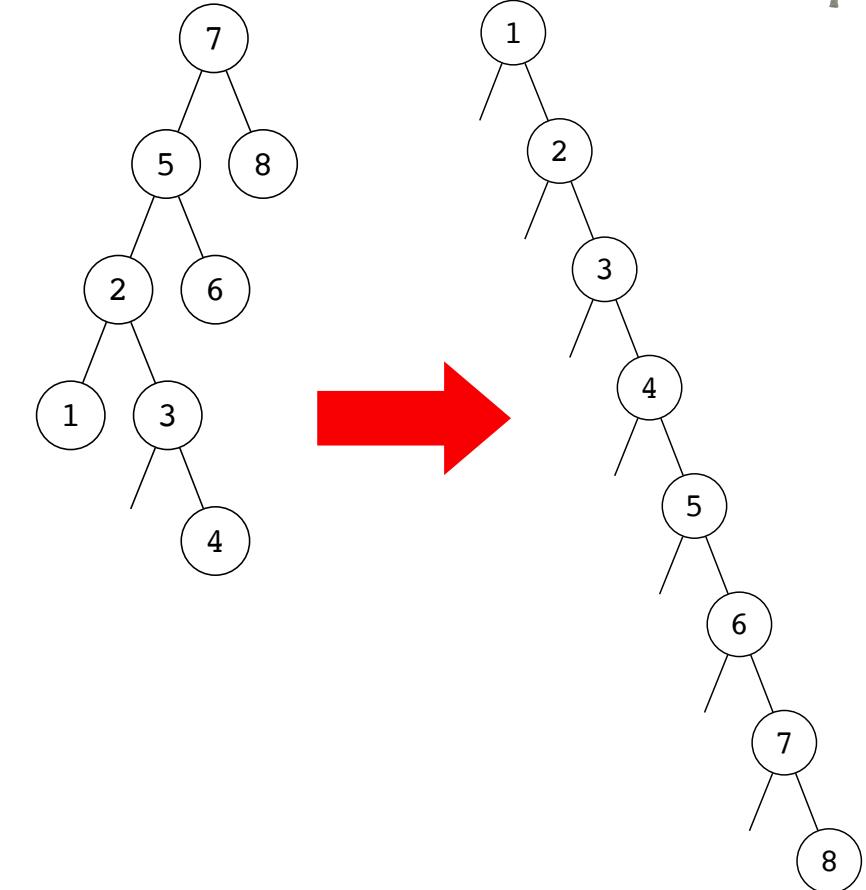
- Paramètres:
  - Un lien  $r$  vers la racine de l'arbre en entrée
  - Un lien  $L$  vers la racine de l'arbre en sortie (liste)





# Linéarisation

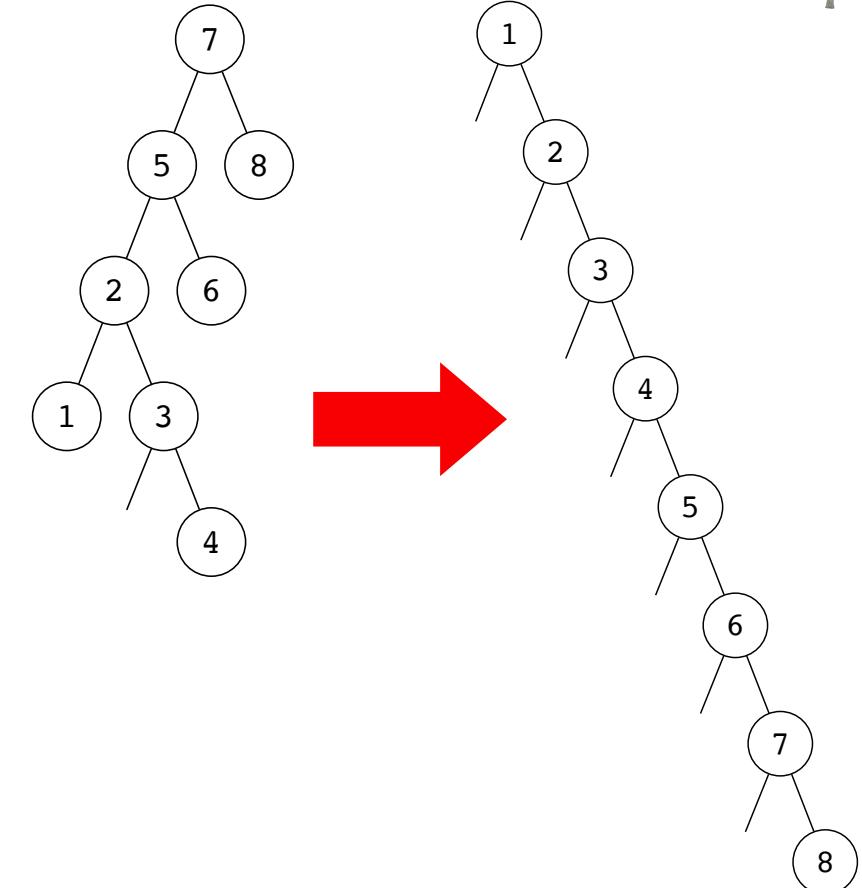
- Paramètres:
  - Un lien  $r$  vers la racine de l'arbre en entrée
  - Un lien  $L$  vers la racine de l'arbre en sortie (liste)
  - Un compteur  $n$  du nombre de noeuds dans l'arbre





# Linéarisation

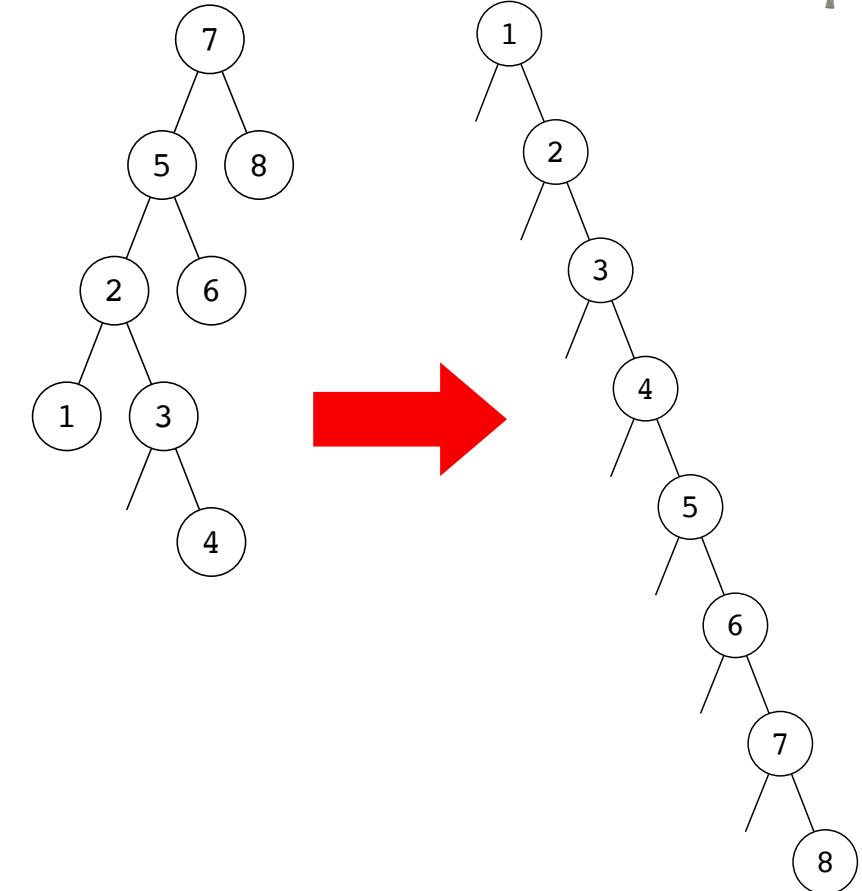
- Paramètres:
  - Un lien  $r$  vers la racine de l'arbre en entrée
  - Un lien  $L$  vers la racine de l'arbre en sortie (liste)
  - Un compteur  $n$  du nombre de noeuds dans l'arbre
- On parcours  $r$  pour insérer dans le bon ordre les noeuds en tête de  $L$ , donc on parcours  $r$  en ordre symétrique inverse





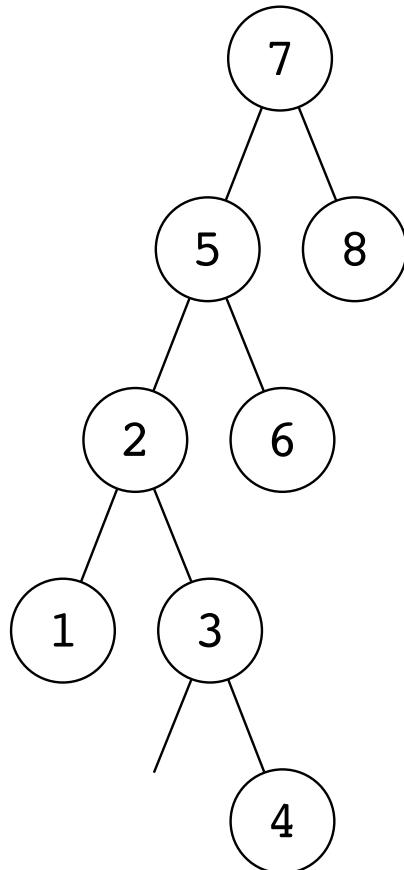
# Linéarisation

```
fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø
```





# Linéariser

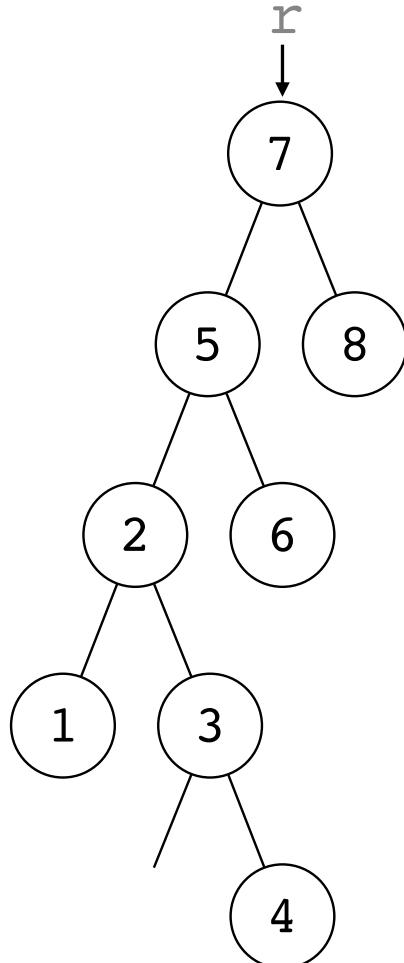


```
linéariser(r=7, L=∅, n=0)
linéariser(r=8, L=∅, n=0)
8.droit ← ∅, L ← 8, n ← 1
```

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



# Linéariser



```
linéariser(r=7, L=∅, n=0)
```

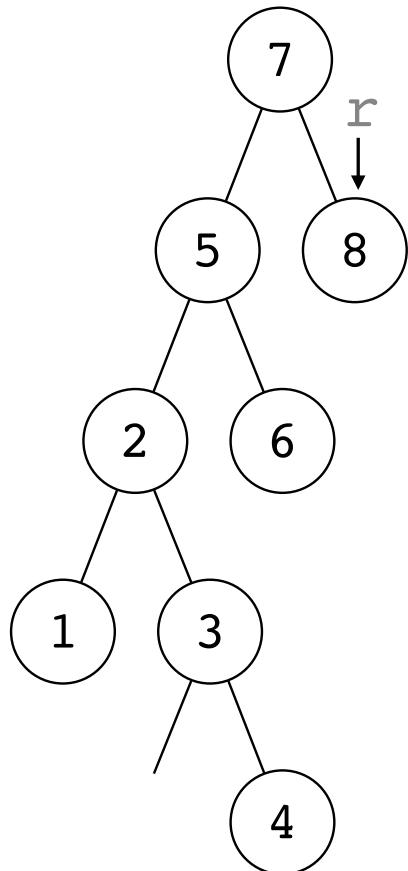
```
linéariser(r=8, L=∅, n=0)
```

```
8.droit ← ∅, L ← 8, n ← 1
```

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



# Linéariser



```
linéariser(r=7, L=∅, n=0)
```

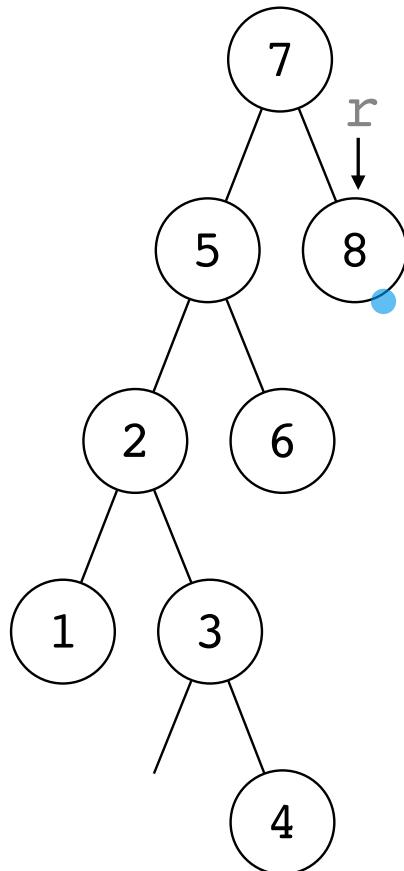
```
linéariser(r=8, L=∅, n=0)
```

```
8.droit ← ∅, L ← 8, n ← 1
```

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



# Linéariser



```
linéariser(r=7, L=∅, n=0)
```

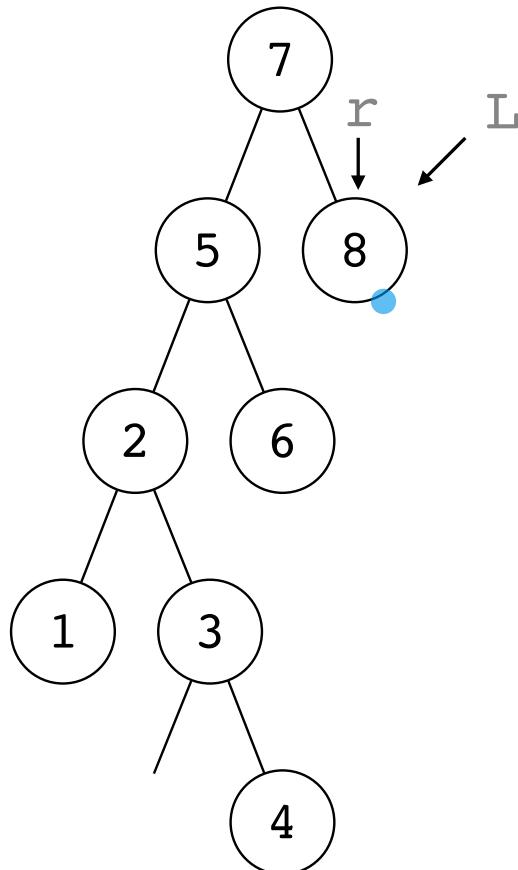
```
linéariser(r=8, L=∅, n=0)
```

```
8.droit ← ∅, L ← 8, n ← 1
```

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



# Linéariser

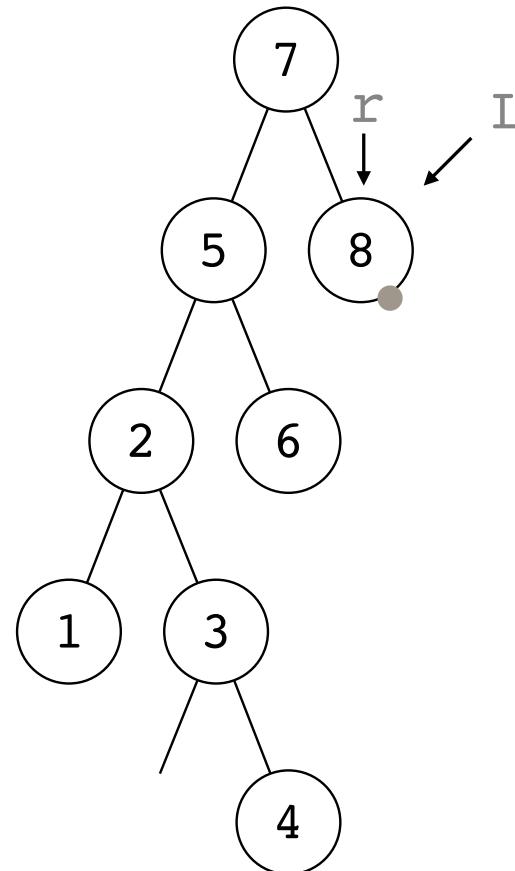


```
linéariser(r=7, L=∅, n=0)
```

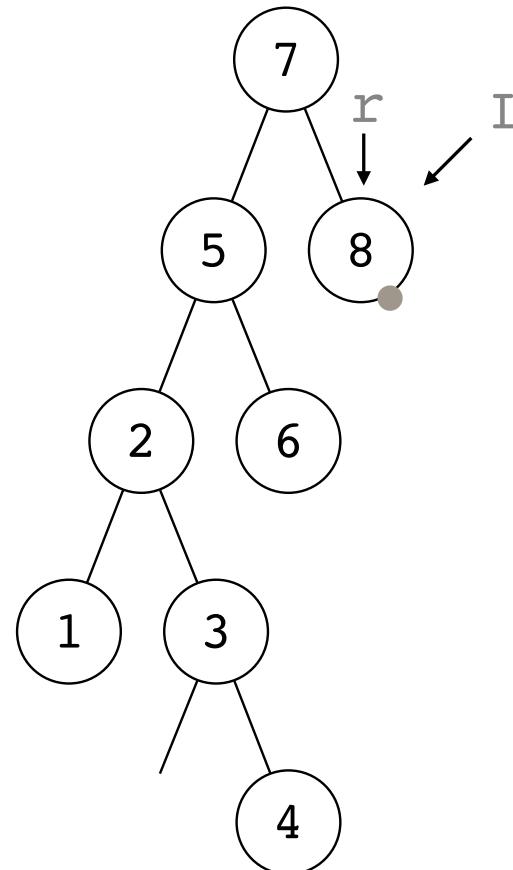
```
linéariser(r=8, L=∅, n=0)
```

```
8.droit ← ∅, L ← 8, n ← 1
```

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



```
linéariser(r=7, L=∅, n=0)
linéariser(r=8, L=∅, n=0)
    8.droit ← ∅, L ← 8, n ← 1
    8.gauche ← ∅
    7.droit ← 8, r ← 7, n ← 2
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



linéariser( $r=7$ ,  $L=\emptyset$ ,  $n=0$ )

linéariser( $r=8$ ,  $L=\emptyset$ ,  $n=0$ )

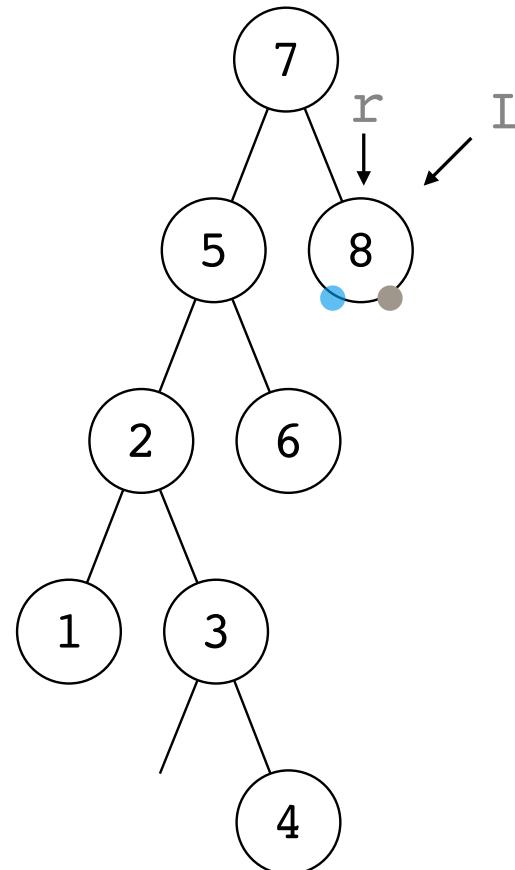
$8.\text{droit} \leftarrow \emptyset$ ,  $L \leftarrow 8$ ,  $n \leftarrow 1$

$8.\text{gauche} \leftarrow \emptyset$

$7.\text{droit} \leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

```

fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow L$ , L  $\leftarrow r$ , n  $\leftarrow n + 1$ 
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow \emptyset$ 
    
```



```
linéariser(r=7, L=∅, n=0)
```

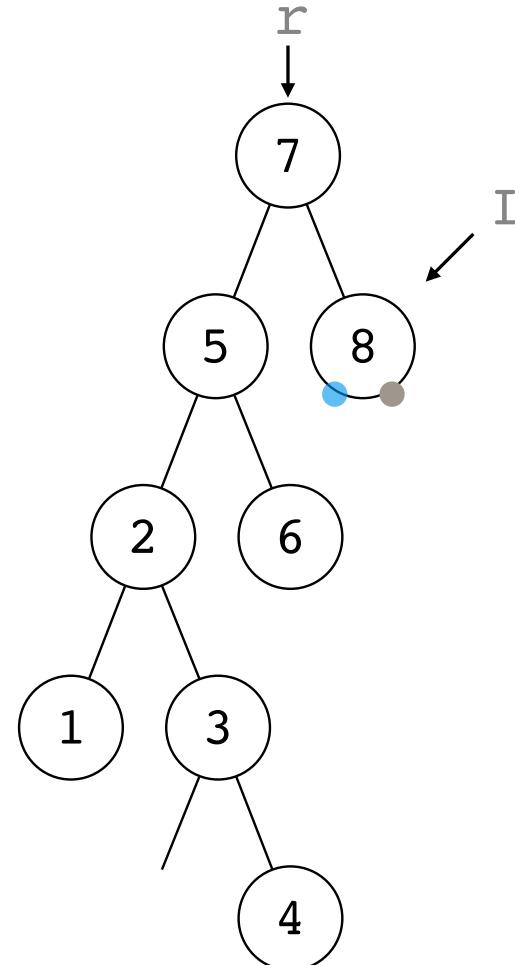
```
linéariser(r=8, L=∅, n=0)
```

```
8.droit ← ∅, L ← 8, n ← 1
```

```
8.gauche ← ∅
```

```
7.droit ← 8, L ← 7, n ← 2
```

```
fonction linéariser (r, ref L, ref n)  
    si r != ∅,  
        linéariser(r.droit, L, n)  
        r.droit ← L, L ← r, n ← n + 1  
        linéariser(r.gauche, L, n)  
        r.gauche ← ∅
```



linéariser( $r=7, L=\emptyset, n=0$ )

linéariser( $r=8, L=\emptyset, n=0$ )

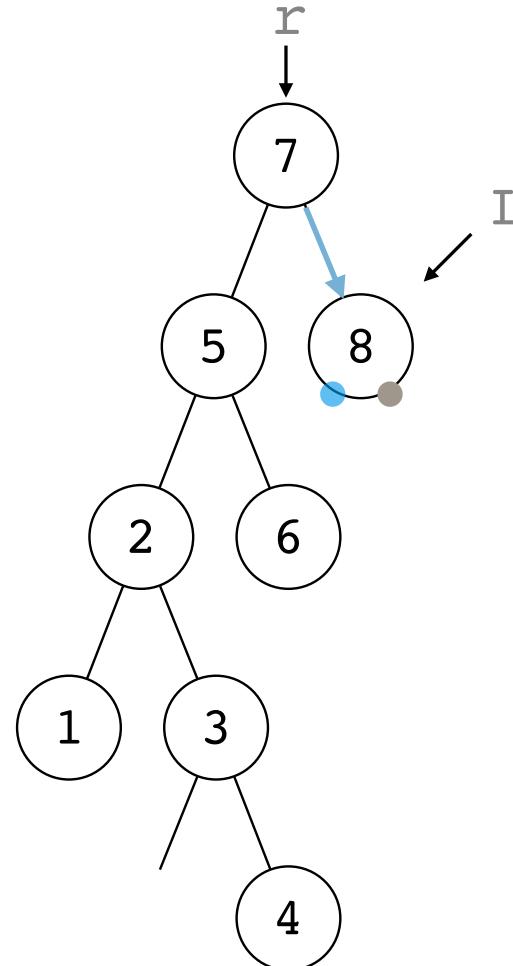
$8.\text{droit} \leftarrow \emptyset, L \leftarrow 8, n \leftarrow 1$

$8.\text{gauche} \leftarrow \emptyset$

$7.\text{droit} \leftarrow 8, L \leftarrow 7, n \leftarrow 2$

```

fonction linéariser (r, ref L, ref n)
  si r != ∅,
    linéariser(r.droit, L, n)
    r.droit ← L, L ← r, n ← n + 1
    linéariser(r.gauche, L, n)
    r.gauche ← ∅
  
```



linéariser( $r=7, L=\emptyset, n=0$ )

linéariser( $r=8, L=\emptyset, n=0$ )

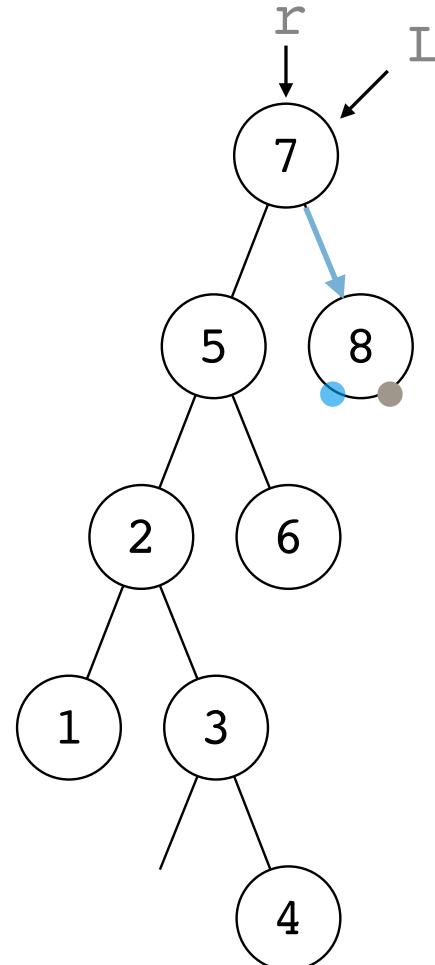
$8.\text{droit} \leftarrow \emptyset, L \leftarrow 8, n \leftarrow 1$

$8.\text{gauche} \leftarrow \emptyset$

$7.\text{droit} \leftarrow 8, L \leftarrow 7, n \leftarrow 2$

```

fonction linéariser ( $r, \text{ref } L, \text{ref } n$ )
  si  $r \neq \emptyset,$ 
    linéariser( $r.\text{droit}, L, n$ )
     $r.\text{droit} \leftarrow L, L \leftarrow r, n \leftarrow n + 1$ 
    linéariser( $r.\text{gauche}, L, n$ )
     $r.\text{gauche} \leftarrow \emptyset$ 
  
```



```
linéariser(r=7, L=∅, n=0)
```

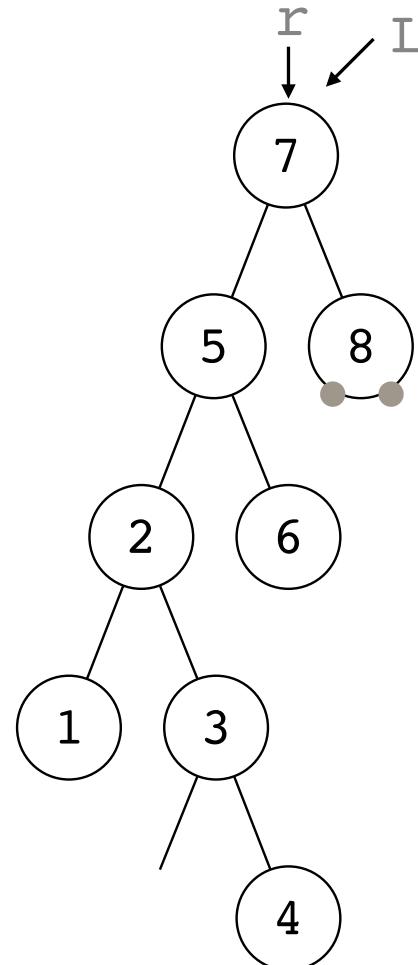
```
linéariser(r=8, L=∅, n=0)
```

```
8.droit ← ∅, L ← 8, n ← 1
```

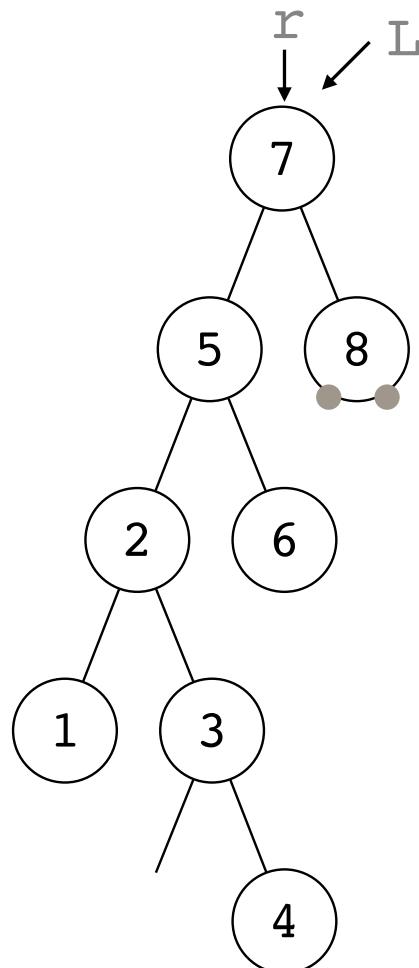
```
8.gauche ← ∅
```

```
7.droit ← 8, L ← 7, n ← 2
```

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



```
linéariser(r=7, L=∅, n=0)
linéariser(r=8, L=∅, n=0)
    8.droit ← ∅, L ← 8, n ← 1
    8.gauche ← ∅
    7.droit ← 8, L ← 7, n ← 2
linéariser(r=5, L=8, n=2)
linéariser(r=6, L=8, n=2)
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



`linéariser(r=7, L=Ø, n=0)`

`linéariser(r=8, L=Ø, n=0)`

`8.droit ← Ø, L ← 8, n ← 1`

`8.gauche ← Ø`

`7.droit ← 8, L ← 7, n ← 2`

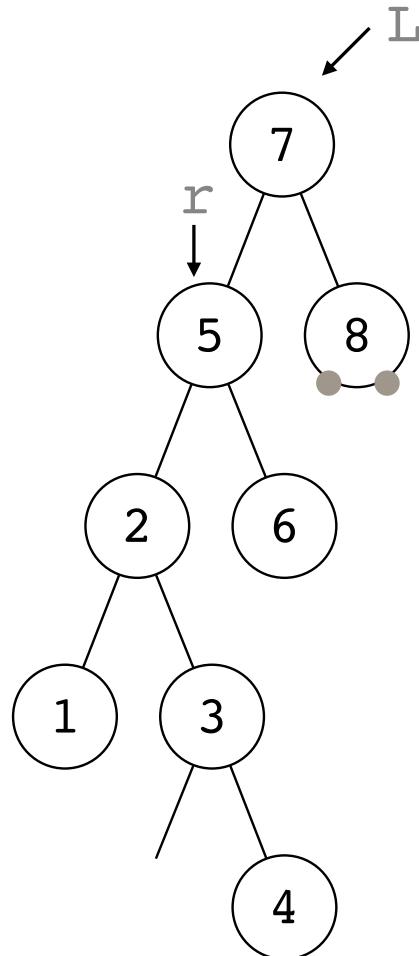
`linéariser(r=5, L=8, n=2)`

`linéariser(r=6, L=8, n=2)`

`6.droit ← 7, L ← 6, n ← 3`

```

fonction linéariser (ref r, ref L, ref n)
  si r != Ø,
    linéariser(r.droit, L, n)
    r.droit ← L, L ← r, n ← n + 1
    linéariser(r.gauche, L, n)
    r.gauche ← Ø
  
```



`linéariser(r=7, L=Ø, n=0)`

`linéariser(r=8, L=Ø, n=0)`

`8.droit ← Ø, L ← 8, n ← 1`

`8.gauche ← Ø`

`7.droit ← 8, L ← 7, n ← 2`

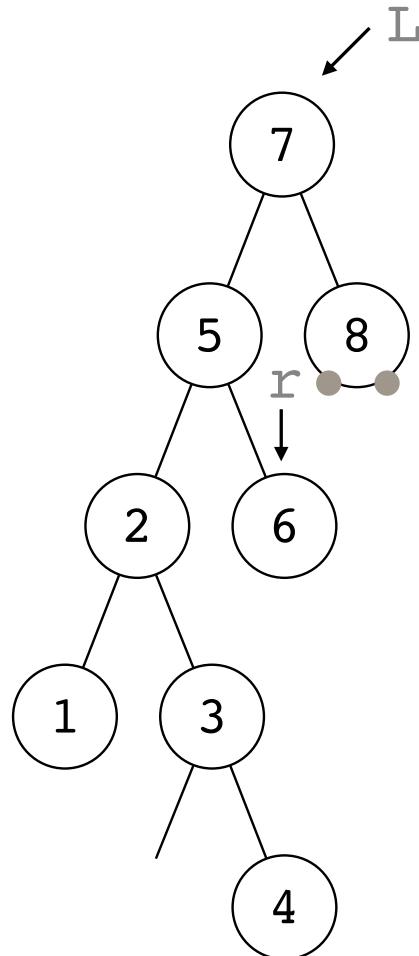
`linéariser(r=5, L=8, n=2)`

`linéariser(r=6, L=8, n=2)`

`6.droit ← 7, L ← 6, n ← 3`

```

fonction linéariser (ref r, ref L, ref n)
  si r != Ø,
    linéariser(r.droit, L, n)
    r.droit ← L, L ← r, n ← n + 1
    linéariser(r.gauche, L, n)
    r.gauche ← Ø
  
```



`linéariser(r=7, L=Ø, n=0)`

`linéariser(r=8, L=Ø, n=0)`

`8.droit ← Ø, L ← 8, n ← 1`

`8.gauche ← Ø`

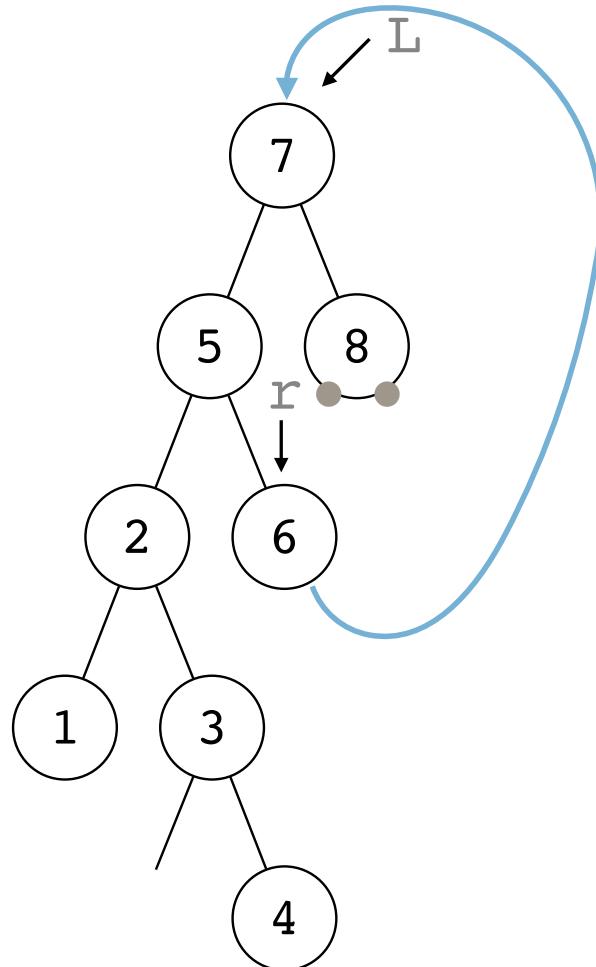
`7.droit ← 8, L ← 7, n ← 2`

`linéariser(r=5, L=8, n=2)`

`linéariser(r=6, L=8, n=2)`

`6.droit ← 7, L ← 6, n ← 3`

```
fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø
```



`linéariser(r=7, L=Ø, n=0)`

linéariser( r=8 , L=Ø , n=0 )

8.droit  $\leftarrow \emptyset$ , L  $\leftarrow 8$ , n  $\leftarrow 1$

8 .gauche  $\leftarrow$  Ø

7.  $\text{droit} \leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

`linéariser(r=5, L=8, n=2)`

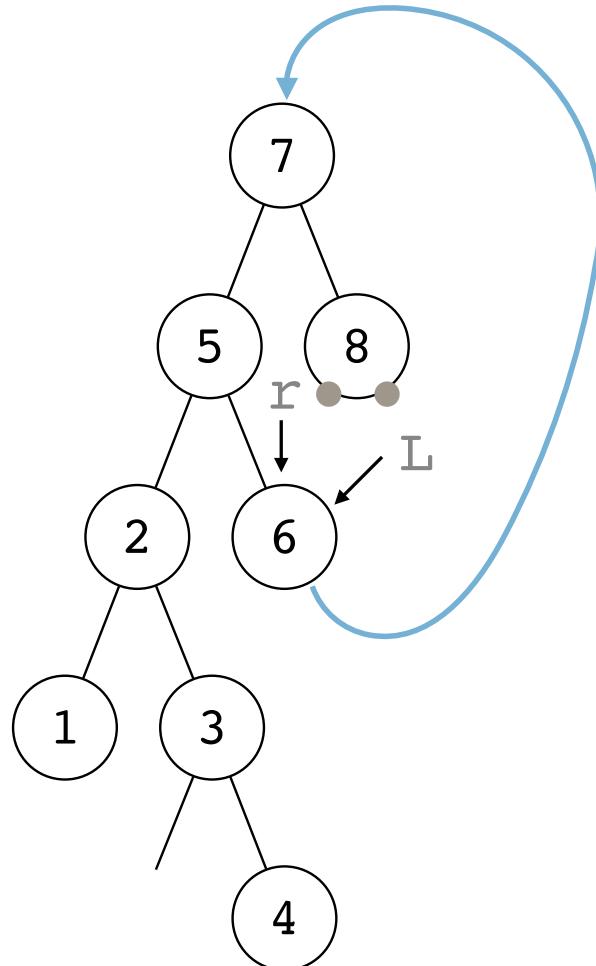
**linéariser( r=6 , L=8 , n=2 )**

6. `droit`  $\leftarrow$  7, `L`  $\leftarrow$  6, `n`  $\leftarrow$  3

```

fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø

```



`linéariser(r=7, L=Ø, n=0)`

`linéariser(r=8, L=Ø, n=0)`

`8.droit ← Ø, L ← 8, n ← 1`

`8.gauche ← Ø`

`7.droit ← 8, L ← 7, n ← 2`

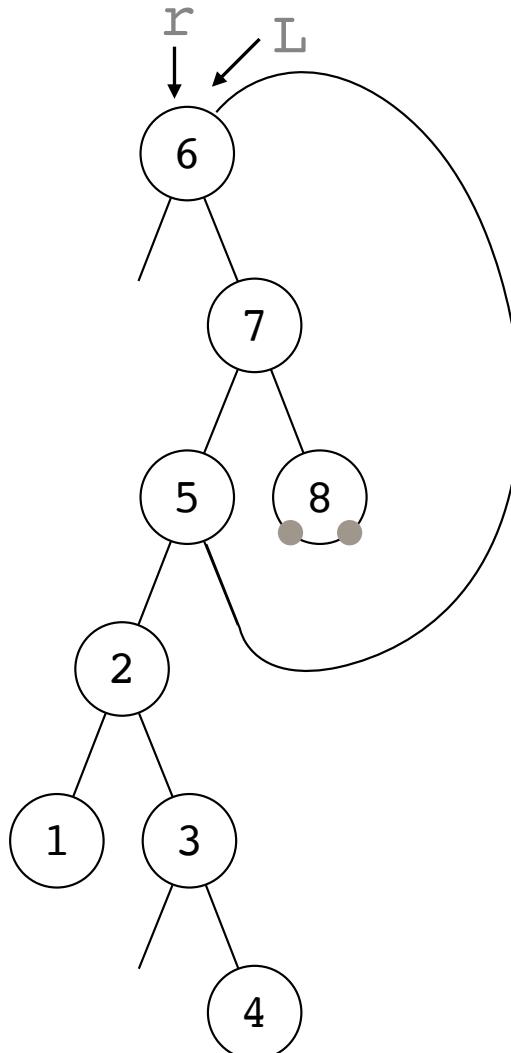
`linéariser(r=5, L=8, n=2)`

`linéariser(r=6, L=8, n=2)`

`6.droit ← 7, L ← 6, n ← 3`

```

fonction linéariser (ref r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø
    
```



linéariser( $r=7$ ,  $L=\emptyset$ ,  $n=0$ )

linéariser( $r=8$ ,  $L=\emptyset$ ,  $n=0$ )

8.droit  $\leftarrow \emptyset$ ,  $L \leftarrow 8$ ,  $n \leftarrow 1$

7.droit  $\leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

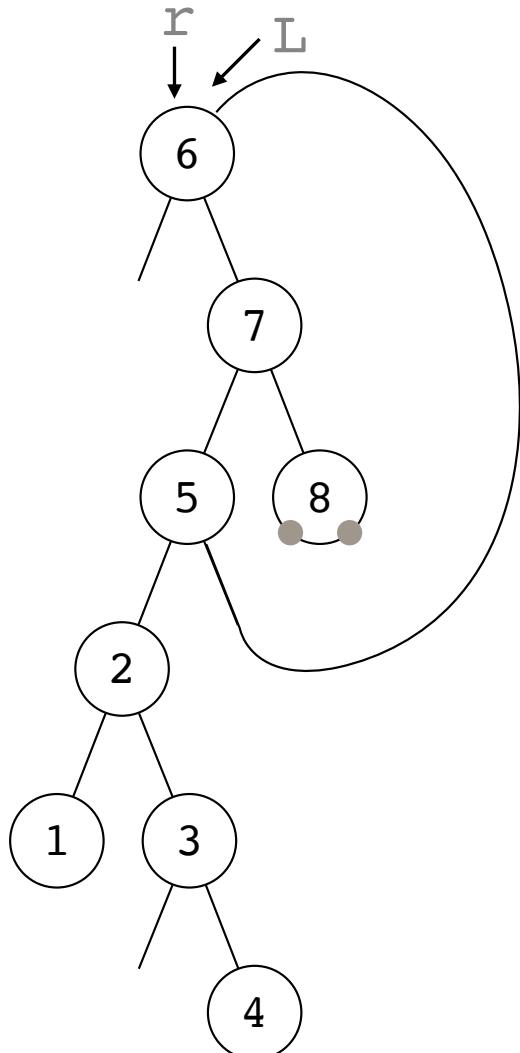
6.droit  $\leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

```

fonction linéariser (r, ref L, ref n)
  si r != ∅,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow L$ ,  $L \leftarrow r$ ,  $n \leftarrow n + 1$ 
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow \emptyset$ 
  
```



linéariser( $r=8$ ,  $L=\emptyset$ ,  $n=0$ )

8.droit  $\leftarrow \emptyset$ ,  $L \leftarrow 8$ ,  $n \leftarrow 1$

7.droit  $\leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

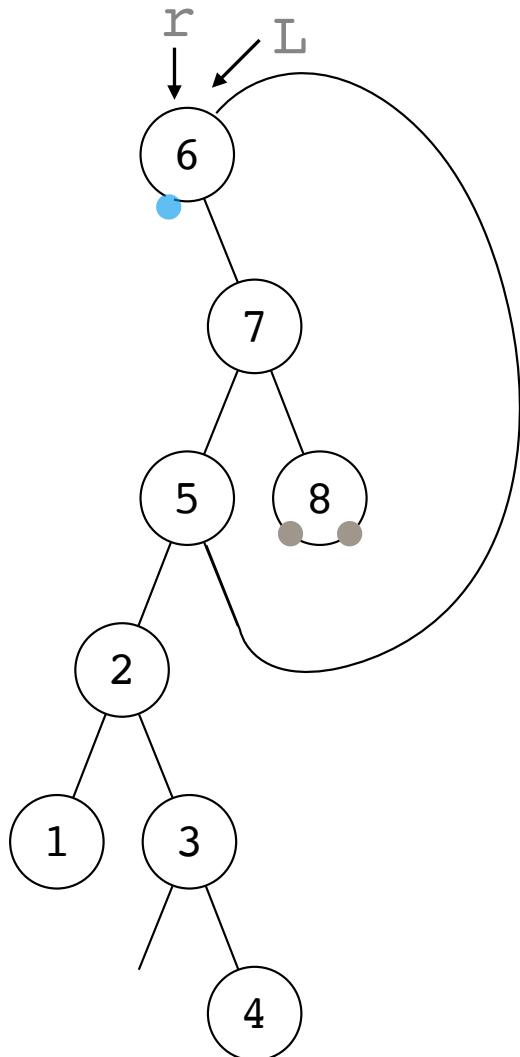
6.droit  $\leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

```

fonction linéariser (r, ref L, ref n)
  si r != ∅,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow L$ ,  $L \leftarrow r$ ,  $n \leftarrow n + 1$ 
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow \emptyset$ 
  
```



linéariser( $r=8$ ,  $L=\emptyset$ ,  $n=0$ )

8.droit  $\leftarrow \emptyset$ ,  $L \leftarrow 8$ ,  $n \leftarrow 1$

7.droit  $\leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

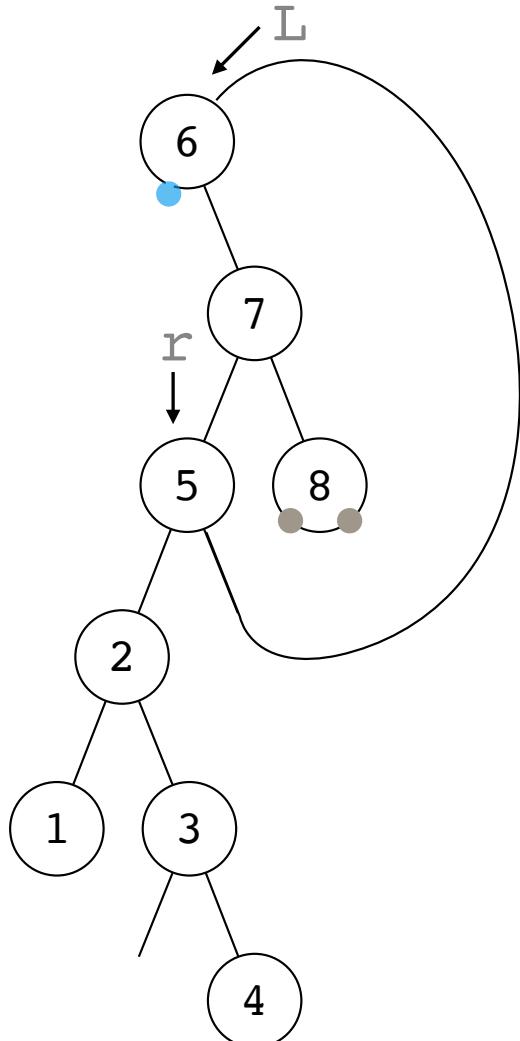
linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

6.droit  $\leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

```
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



linéariser( $r=8, L=\emptyset, n=0$ )

8.droit  $\leftarrow \emptyset, L \leftarrow 8, n \leftarrow 1$

7.droit  $\leftarrow 8, L \leftarrow 7, n \leftarrow 2$

linéariser( $r=5, L=8, n=2$ )

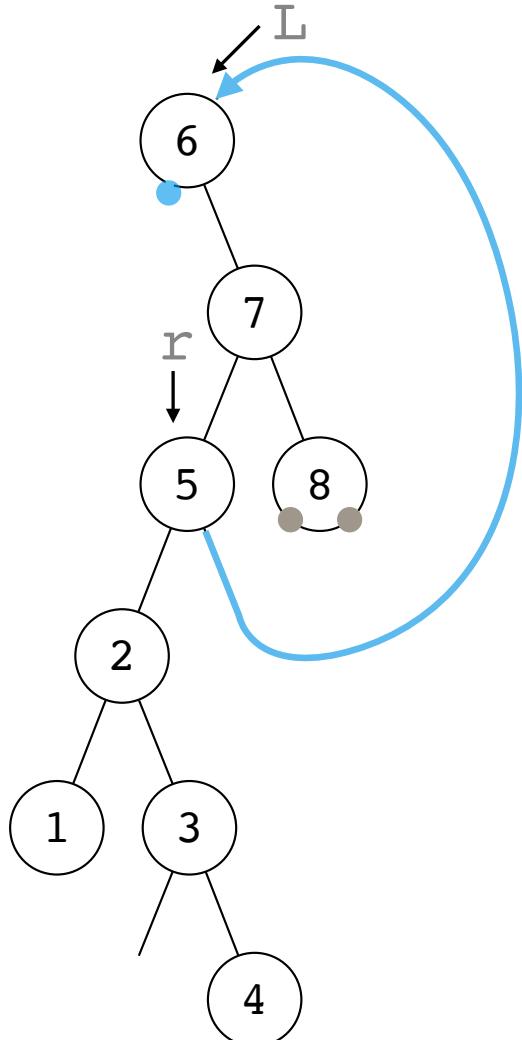
linéariser( $r=6, L=8, n=2$ )

6.droit  $\leftarrow 7, L \leftarrow 6, n \leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6, L \leftarrow 5, n \leftarrow 4$

```
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow L, L \leftarrow r, n \leftarrow n + 1$ 
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow \emptyset$ 
```



linéariser( $r=8, L=\emptyset, n=0$ )

8.droit  $\leftarrow \emptyset, L \leftarrow 8, n \leftarrow 1$

7.droit  $\leftarrow 8, L \leftarrow 7, n \leftarrow 2$

linéariser( $r=5, L=8, n=2$ )

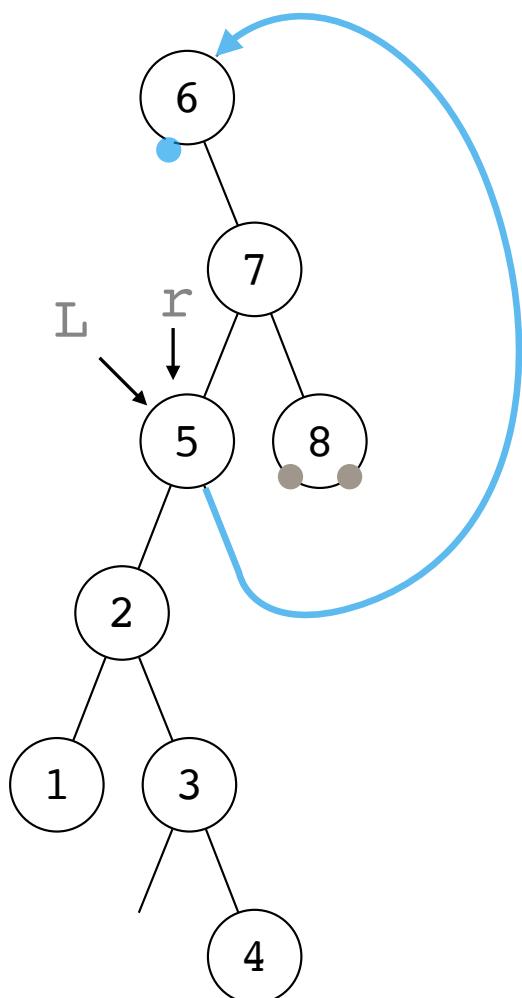
linéariser( $r=6, L=8, n=2$ )

6.droit  $\leftarrow 7, L \leftarrow 6, n \leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6, L \leftarrow 5, n \leftarrow 4$

```
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



linéariser( $r=8$ ,  $L=\emptyset$ ,  $n=0$ )

8.droit  $\leftarrow \emptyset$ ,  $L \leftarrow 8$ ,  $n \leftarrow 1$

7.droit  $\leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

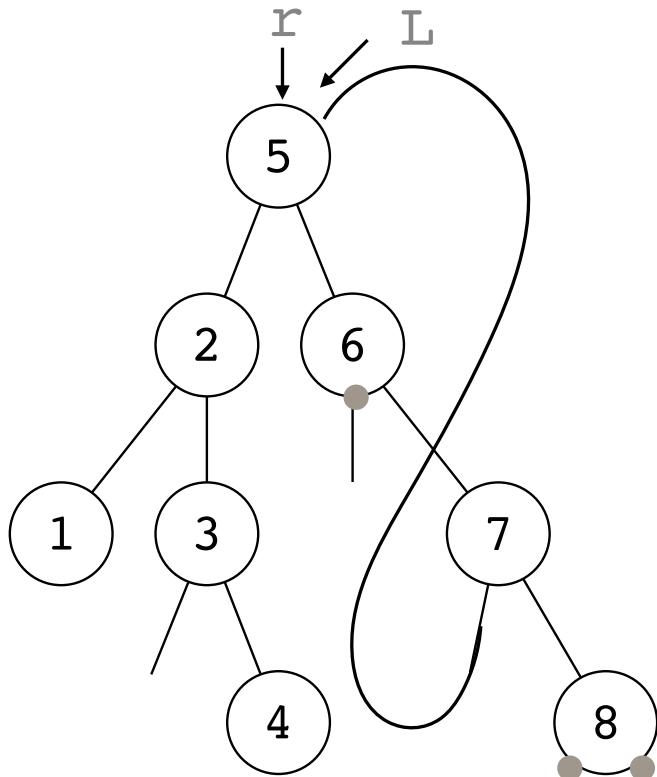
linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

6.droit  $\leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

```
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



linéariser( $r=7$ ,  $L=\emptyset$ ,  $n=0$ )

linéariser( $r=8$ ,  $L=\emptyset$ ,  $n=0$ )

$8.\text{droit} \leftarrow \emptyset$ ,  $L \leftarrow 8$ ,  $n \leftarrow 1$

$7.\text{droit} \leftarrow 8$ ,  $L \leftarrow 7$ ,  $n \leftarrow 2$

linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

$6.\text{droit} \leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

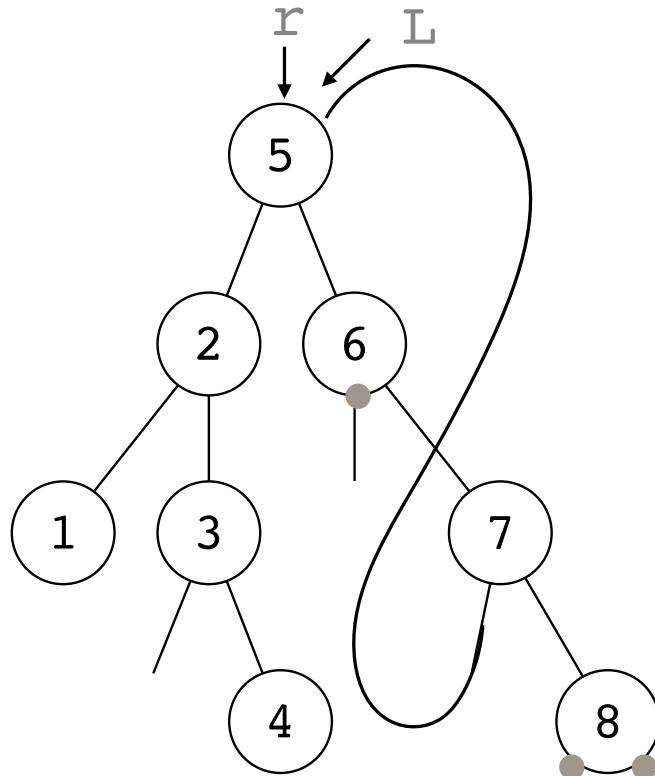
$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

linéariser( $r=2$ ,  $L=5$ ,  $n=4$ )

linéariser( $r=3$ ,  $L=5$ ,  $n=4$ )

```
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
```



linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

$6.\text{droit} \leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

linéariser( $r=2$ ,  $L=5$ ,  $n=4$ )

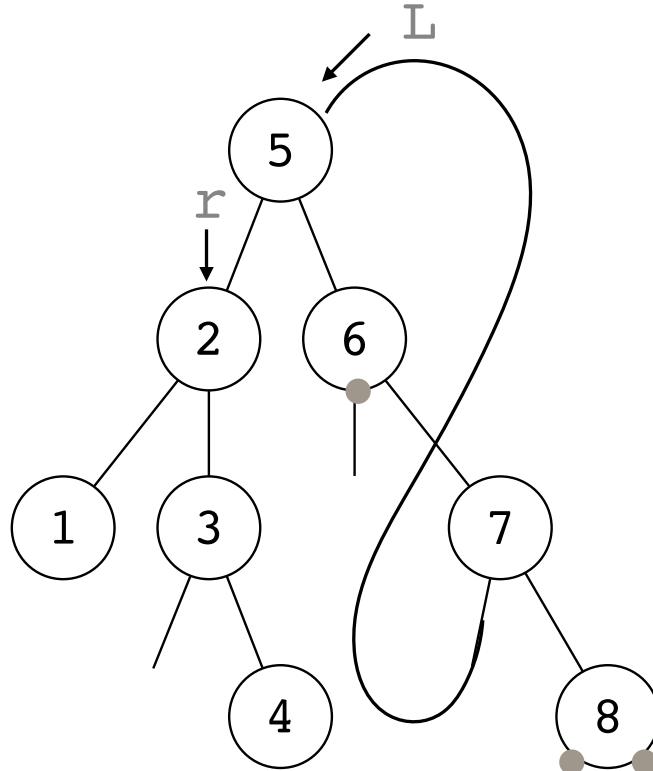
linéariser( $r=3$ ,  $L=5$ ,  $n=4$ )

linéariser( $r=4$ ,  $L=5$ ,  $n=4$ )

$4.\text{droit} \leftarrow 5$ ,  $L \leftarrow 4$ ,  $n \leftarrow 5$

```

fonction linéariser (r, ref L, ref n)
  si r != ∅,
    linéariser(r.droit, L, n)
    r.droit ← L, L ← r, n ← n + 1
    linéariser(r.gauche, L, n)
    r.gauche ← ∅
  
```



linéariser( $r=5, L=8, n=2$ )

linéariser( $r=6, L=8, n=2$ )

$6.\text{droit} \leftarrow 7, L \leftarrow 6, n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6, L \leftarrow 5, n \leftarrow 4$

linéariser( $r=2, L=5, n=4$ )

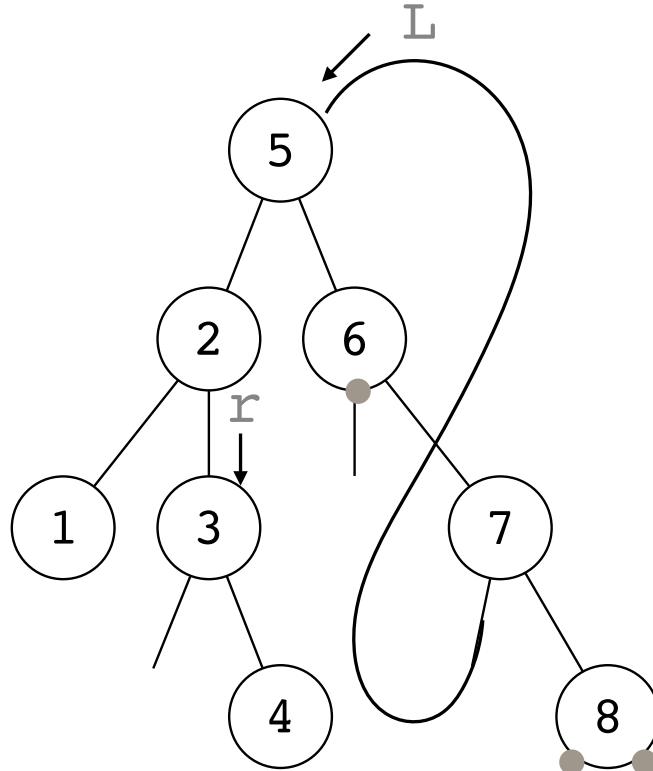
linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=5, L=8, n=2$ )

linéariser( $r=6, L=8, n=2$ )

$6.\text{droit} \leftarrow 7, L \leftarrow 6, n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6, L \leftarrow 5, n \leftarrow 4$

linéariser( $r=2, L=5, n=4$ )

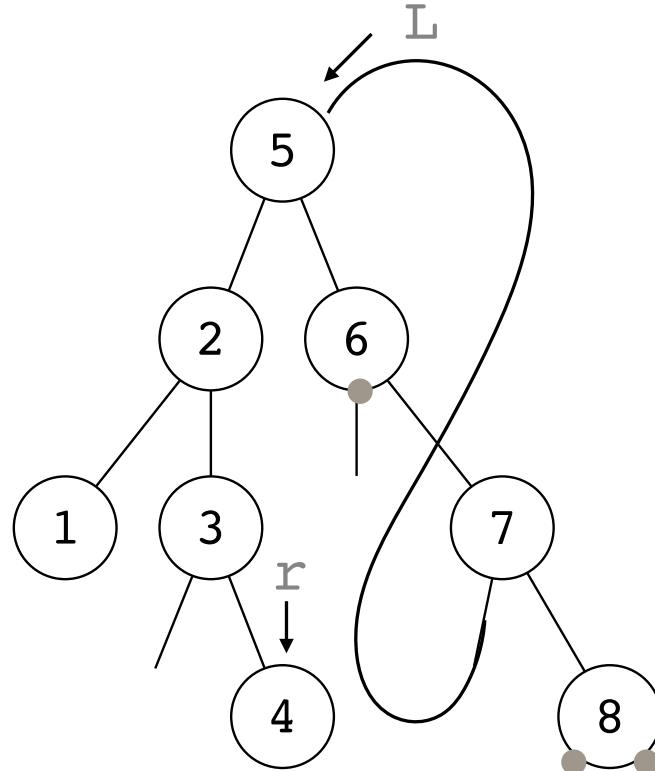
linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=5, L=8, n=2$ )

linéariser( $r=6, L=8, n=2$ )

$6.\text{droit} \leftarrow 7, L \leftarrow 6, n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6, L \leftarrow 5, n \leftarrow 4$

linéariser( $r=2, L=5, n=4$ )

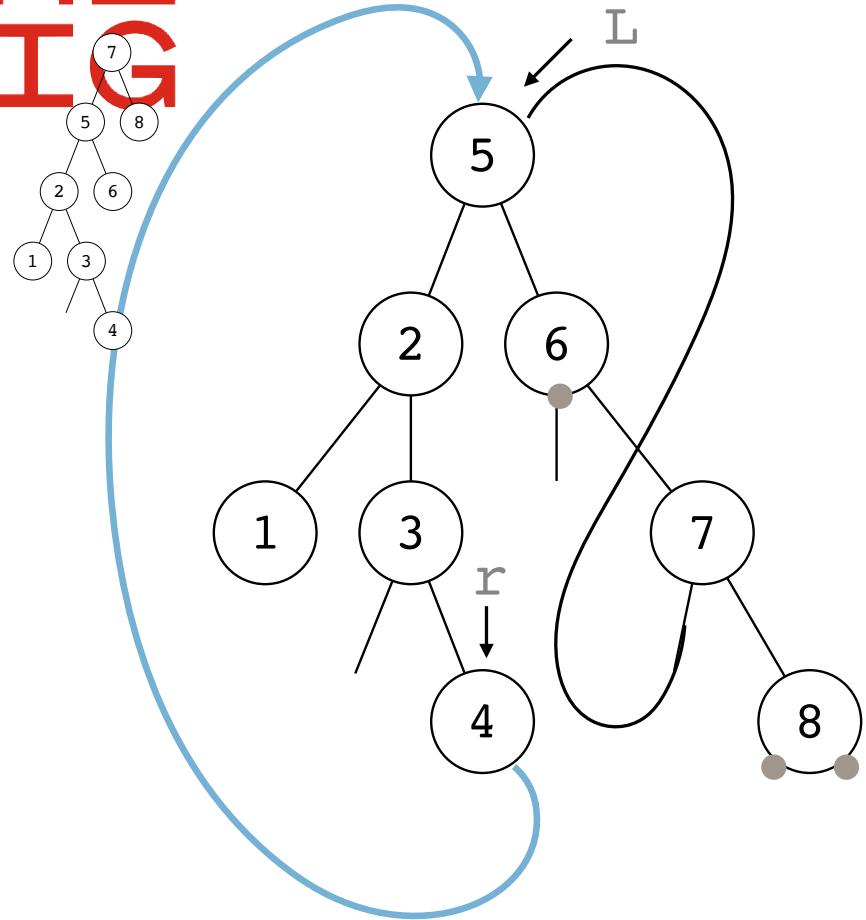
linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=5$ ,  $L=8$ ,  $n=2$ )

linéariser( $r=6$ ,  $L=8$ ,  $n=2$ )

$6.\text{droit} \leftarrow 7$ ,  $L \leftarrow 6$ ,  $n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6$ ,  $L \leftarrow 5$ ,  $n \leftarrow 4$

linéariser( $r=2$ ,  $L=5$ ,  $n=4$ )

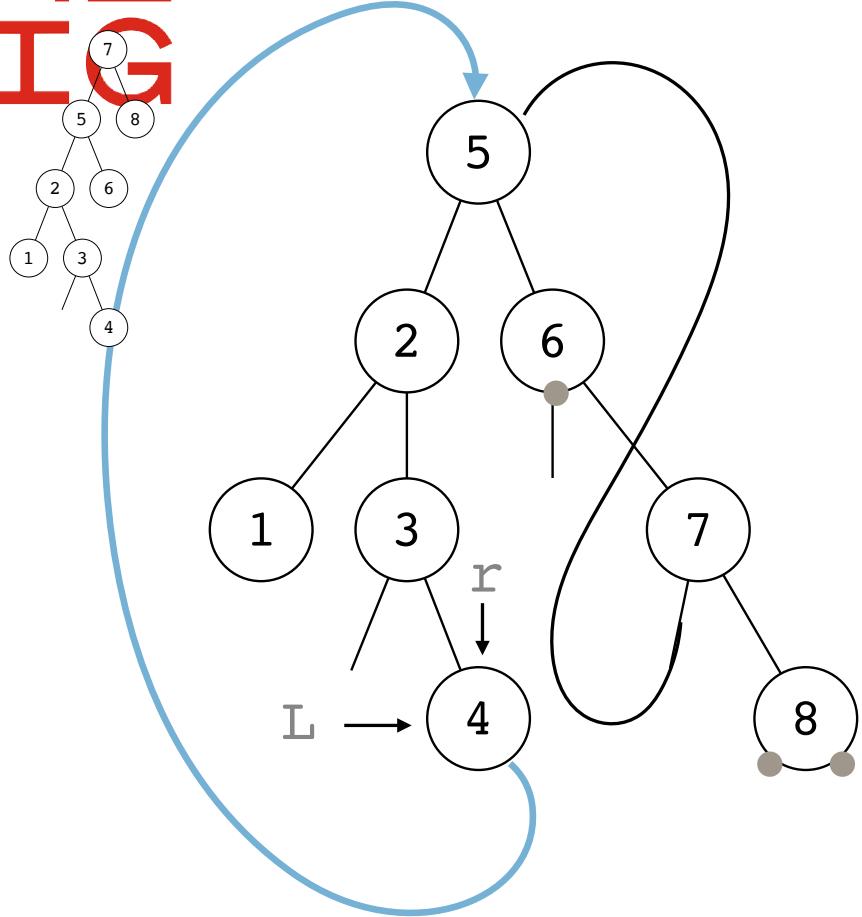
linéariser( $r=3$ ,  $L=5$ ,  $n=4$ )

linéariser( $r=4$ ,  $L=5$ ,  $n=4$ )

$4.\text{droit} \leftarrow 5$ ,  $L \leftarrow 4$ ,  $n \leftarrow 5$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=5, L=8, n=2$ )

linéariser( $r=6, L=8, n=2$ )

$6.\text{droit} \leftarrow 7, L \leftarrow 6, n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6, L \leftarrow 5, n \leftarrow 4$

linéariser( $r=2, L=5, n=4$ )

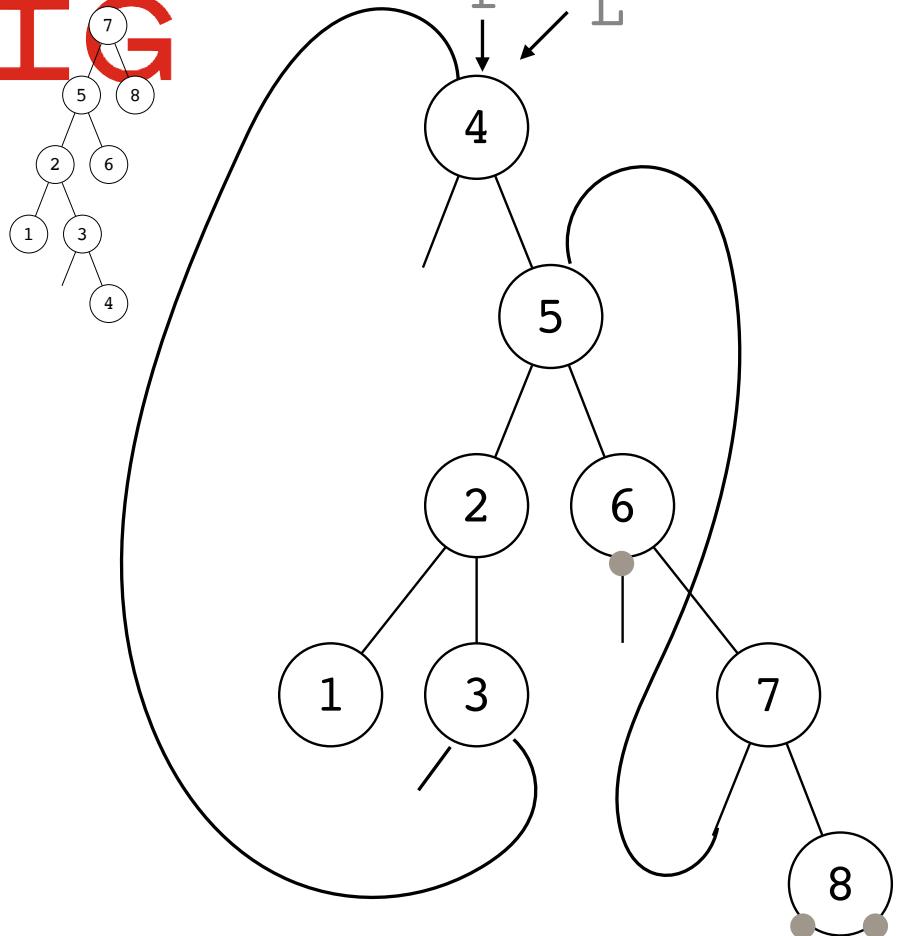
linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=5, L=8, n=2$ )

linéariser( $r=6, L=8, n=2$ )

$6.\text{droit} \leftarrow 7, L \leftarrow 6, n \leftarrow 3$

$6.\text{gauche} \leftarrow \emptyset$

$5.\text{droit} \leftarrow 6, L \leftarrow 5, n \leftarrow 4$

linéariser( $r=2, L=5, n=4$ )

linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

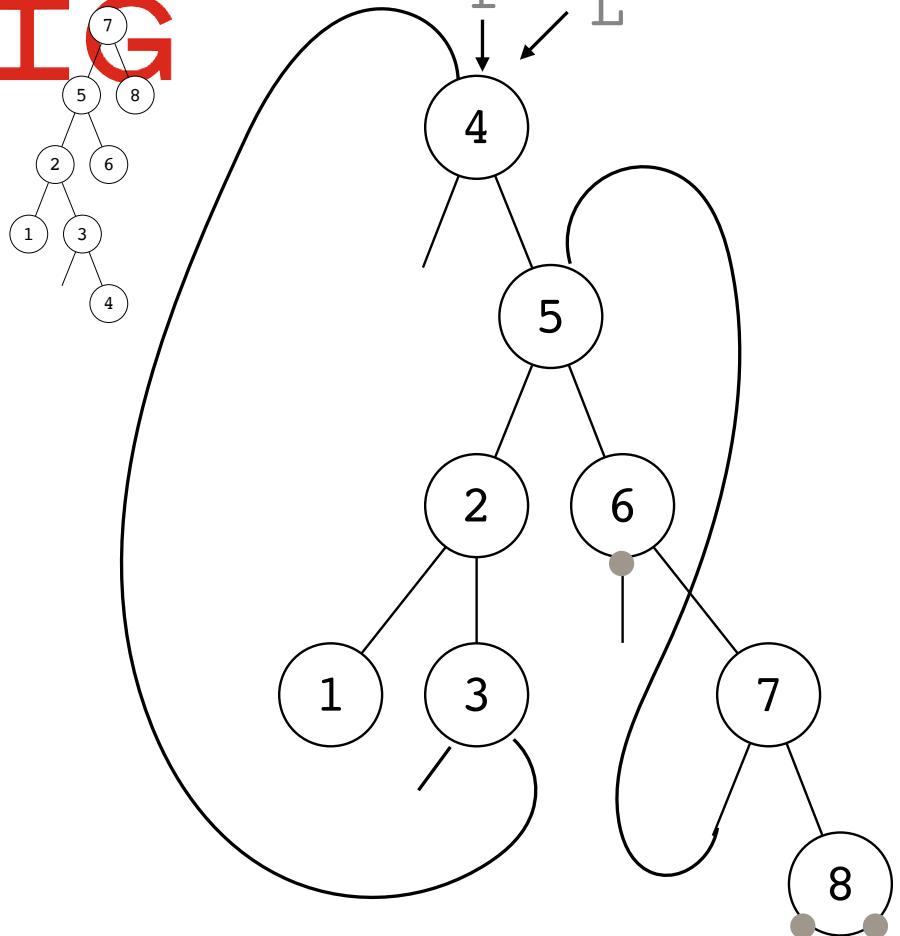
$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

$4.\text{gauche} \leftarrow \emptyset$

$3.\text{droit} \leftarrow 4, L \leftarrow 3, n \leftarrow 6$

```

fonction linéariser (ref r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
    
```



6.droit  $\leftarrow$  7, L  $\leftarrow$  6, n  $\leftarrow$  3

6.gauche  $\leftarrow$   $\emptyset$

5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4

linéariser(r=2, L=5, n=4)

linéariser(r=3, L=5, n=4)

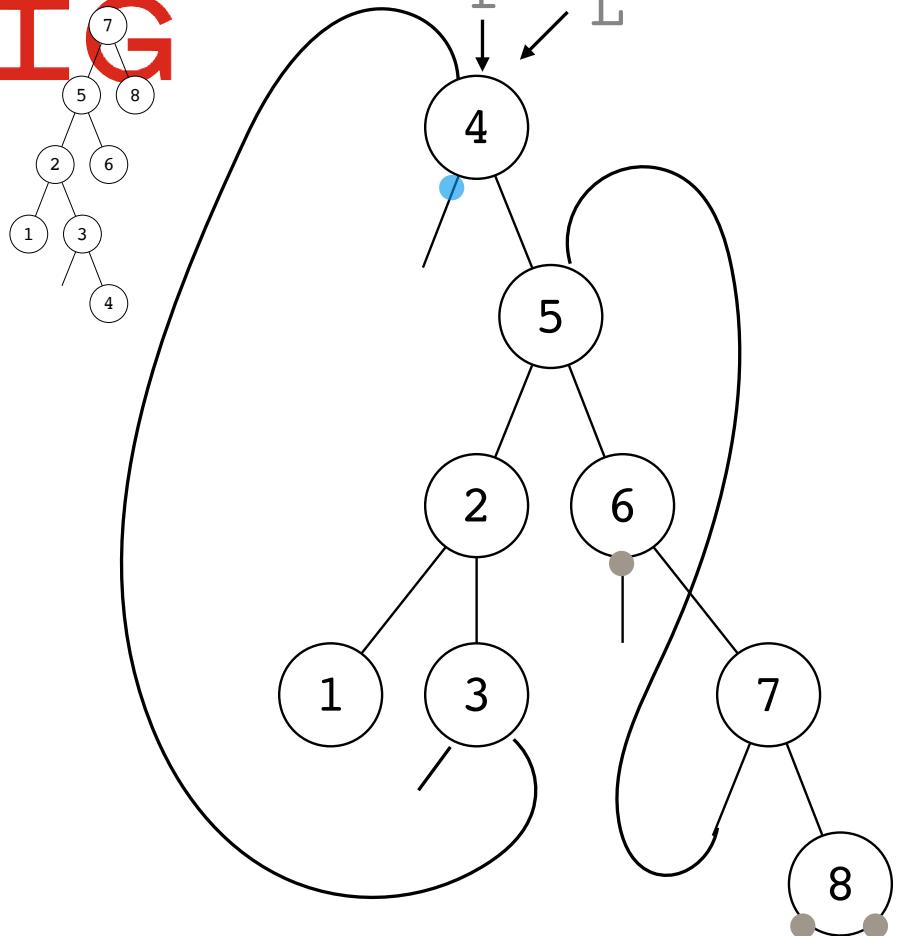
linéariser(r=4, L=5, n=4)

4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5

4.gauche  $\leftarrow$   $\emptyset$

3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6

```
fonction linéariser (ref r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



6.droit  $\leftarrow 7$ , L  $\leftarrow 6$ , n  $\leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ , L  $\leftarrow 5$ , n  $\leftarrow 4$

linéariser(r=2, L=5, n=4)

linéariser(r=3, L=5, n=4)

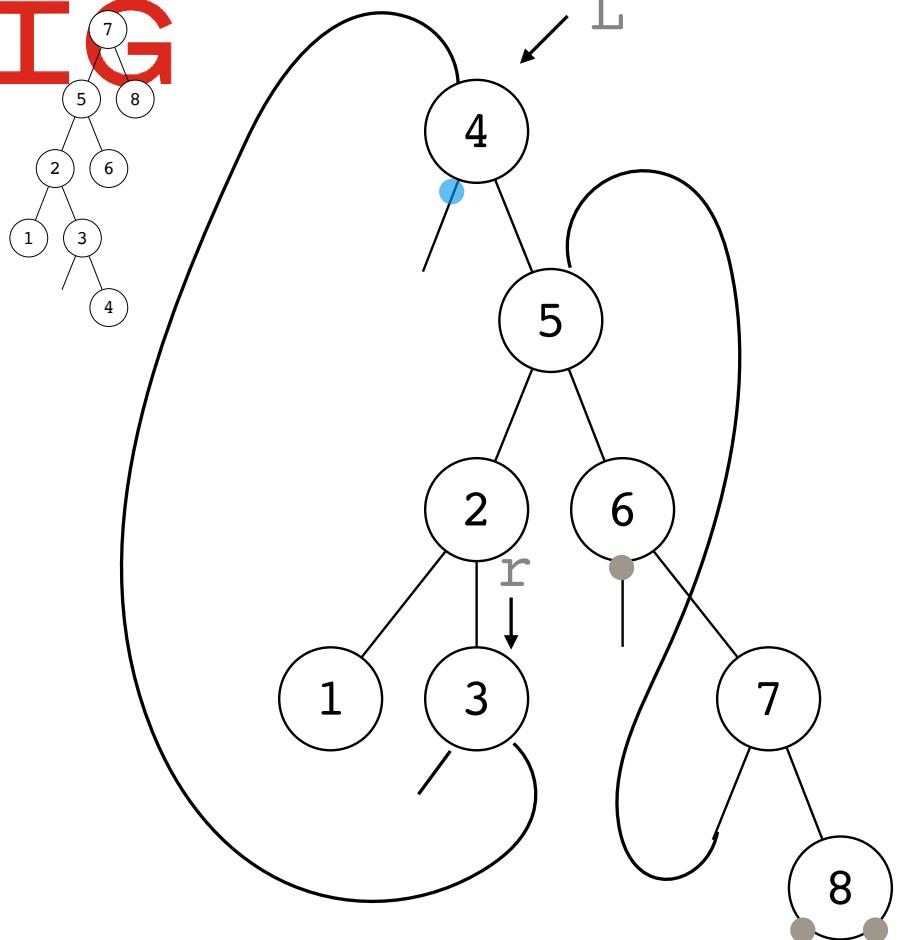
linéariser(r=4, L=5, n=4)

4.droit  $\leftarrow 5$ , L  $\leftarrow 4$ , n  $\leftarrow 5$

4.gauche  $\leftarrow \emptyset$

3.droit  $\leftarrow 4$ , L  $\leftarrow 3$ , n  $\leftarrow 6$

```
fonction linéariser (ref r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



6.droit  $\leftarrow$  7, L  $\leftarrow$  6, n  $\leftarrow$  3

6.gauche  $\leftarrow$   $\emptyset$

5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4

linéariser(r=2, L=5, n=4)

linéariser(r=3, L=5, n=4)

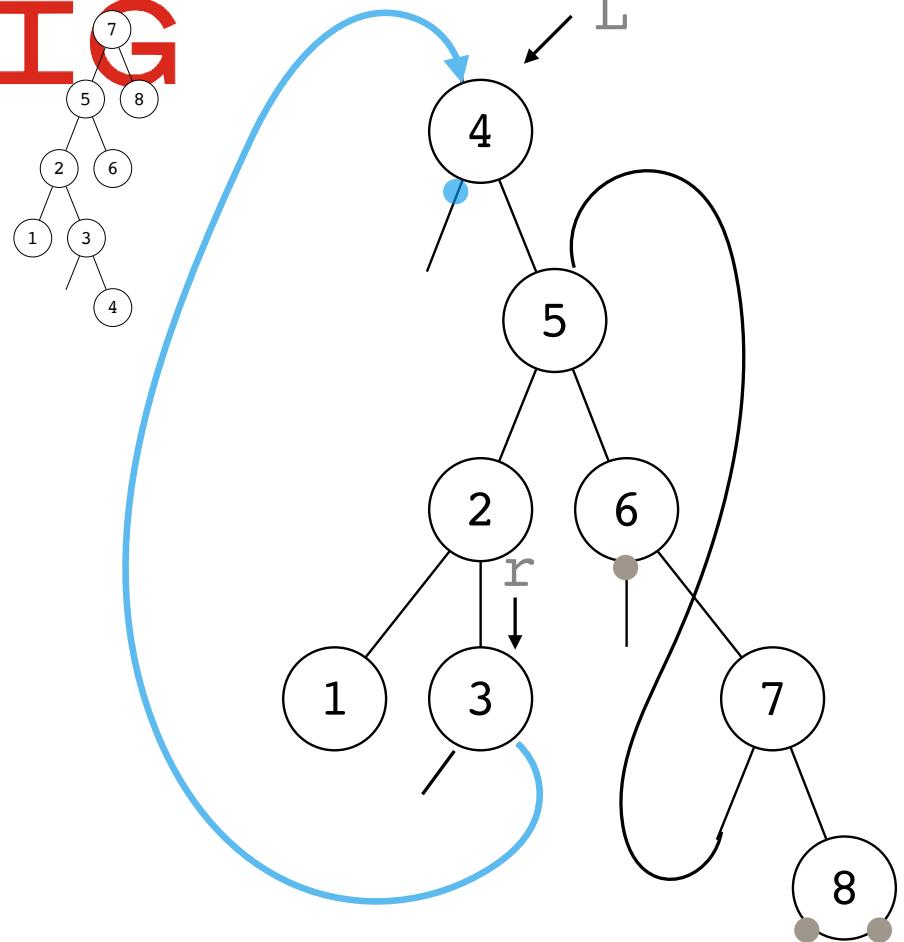
linéariser(r=4, L=5, n=4)

4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5

4.gauche  $\leftarrow$   $\emptyset$

3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6

```
fonction linéariser (ref r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



6.droit  $\leftarrow 7$ , L  $\leftarrow 6$ , n  $\leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ , L  $\leftarrow 5$ , n  $\leftarrow 4$

linéariser(r=2, L=5, n=4)

linéariser(r=3, L=5, n=4)

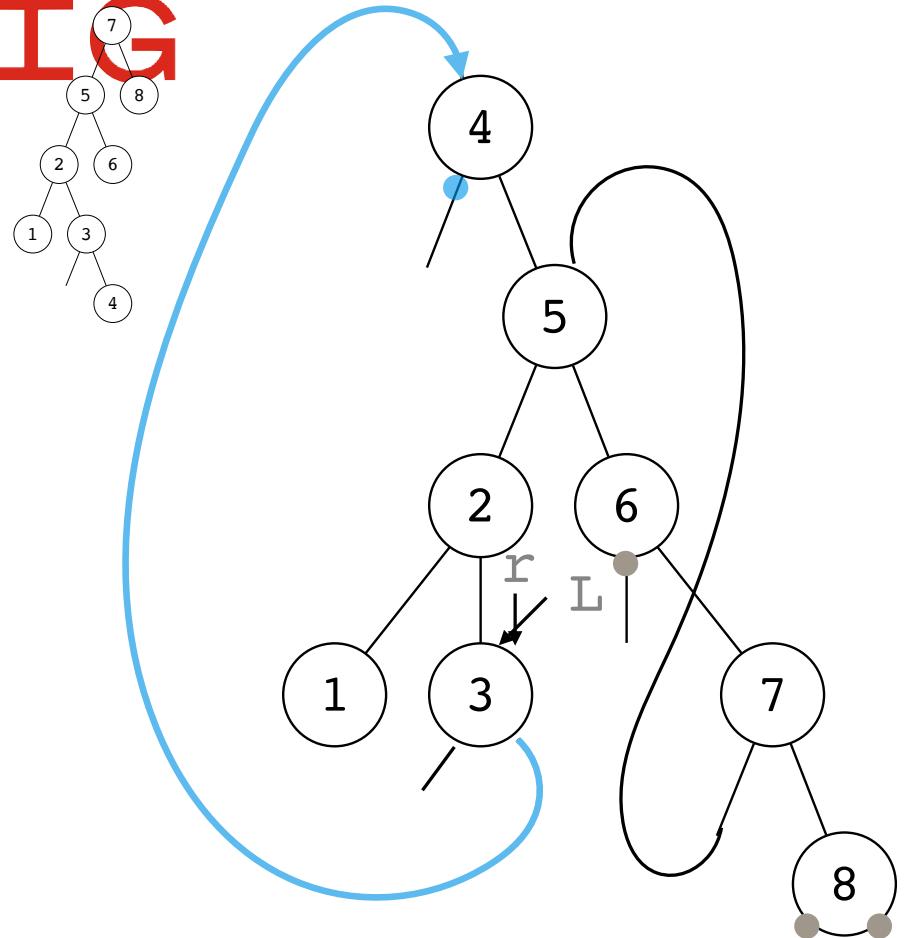
linéariser(r=4, L=5, n=4)

4.droit  $\leftarrow 5$ , L  $\leftarrow 4$ , n  $\leftarrow 5$

4.gauche  $\leftarrow \emptyset$

3.droit  $\leftarrow 4$ , L  $\leftarrow 3$ , n  $\leftarrow 6$

```
fonction linéariser (ref r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



6.droit  $\leftarrow 7$ , L  $\leftarrow 6$ , n  $\leftarrow 3$

6.gauche  $\leftarrow \emptyset$

5.droit  $\leftarrow 6$ , L  $\leftarrow 5$ , n  $\leftarrow 4$

linéariser(r=2, L=5, n=4)

linéariser(r=3, L=5, n=4)

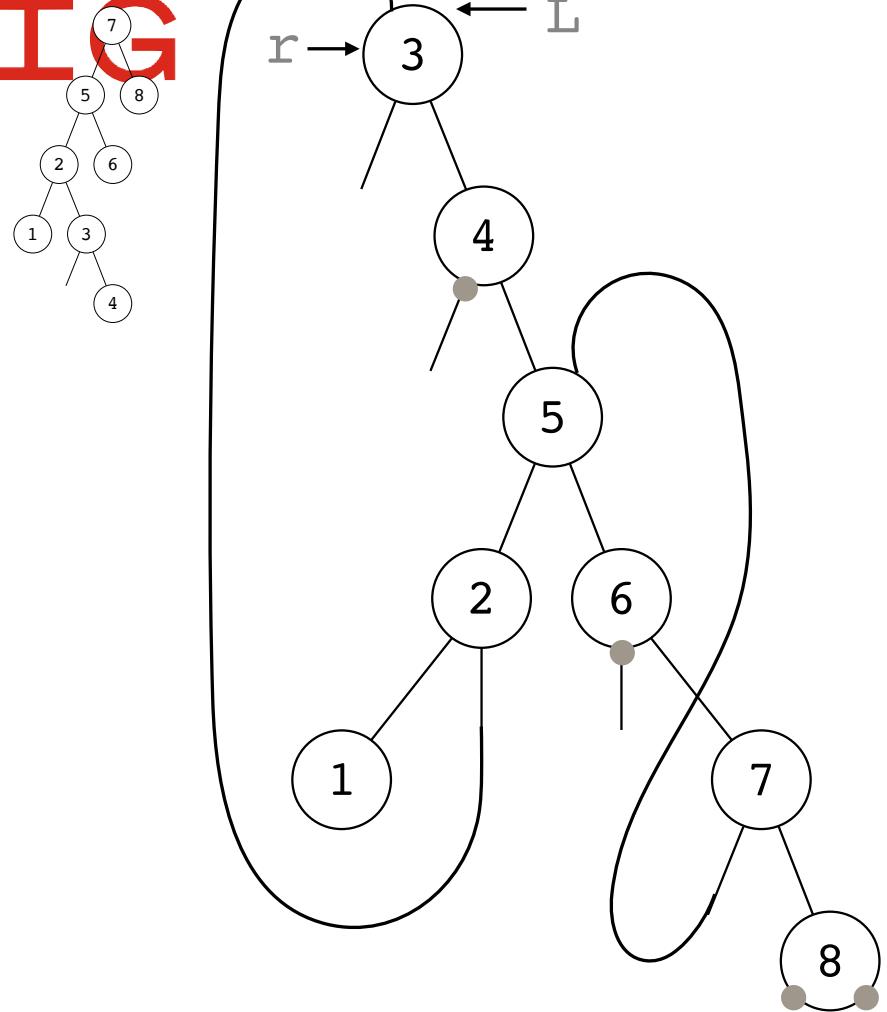
linéariser(r=4, L=5, n=4)

4.droit  $\leftarrow 5$ , L  $\leftarrow 4$ , n  $\leftarrow 5$

4.gauche  $\leftarrow \emptyset$

3.droit  $\leftarrow 4$ , L  $\leftarrow 3$ , n  $\leftarrow 6$

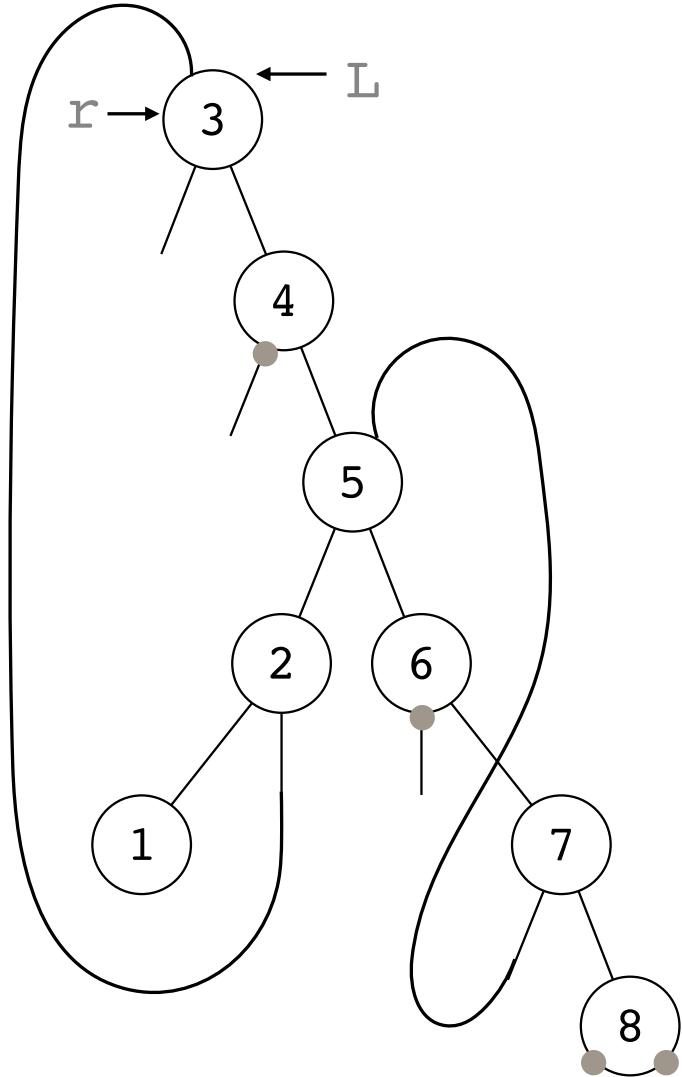
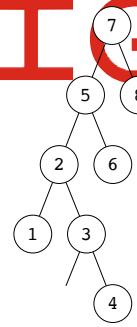
```
fonction linéariser (ref r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



6.droit  $\leftarrow$  7, L  $\leftarrow$  6, n  $\leftarrow$  3  
 6.gauche  $\leftarrow$   $\emptyset$   
 5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4  
 linéariser(r=2, L=5, n=4)  
 linéariser(r=3, L=5, n=4)  
 linéariser(r=4, L=5, n=4)  
 4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5  
 4.gauche  $\leftarrow$   $\emptyset$   
 3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow$   $\emptyset$   
 2.droit  $\leftarrow$  3, L  $\leftarrow$  2, n  $\leftarrow$  7

```

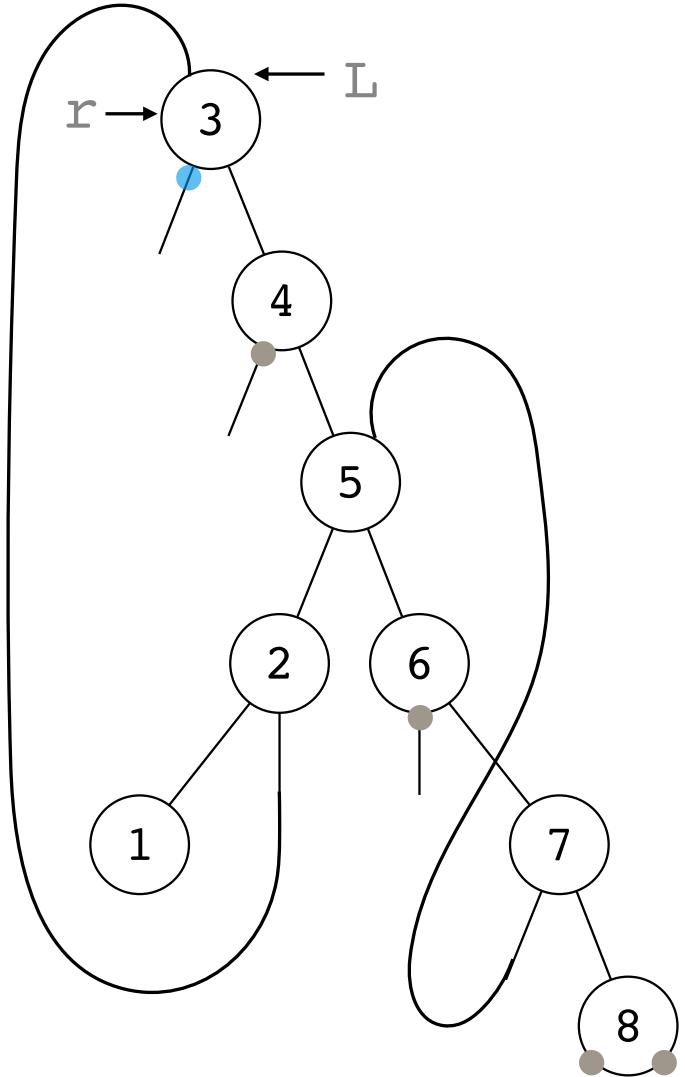
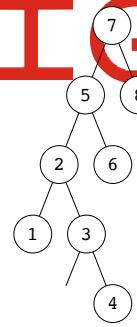
fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow$   $\emptyset$ 
  
```



5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4  
 linéariser(r=2, L=5, n=4)  
 linéariser(r=3, L=5, n=4)  
 linéariser(r=4, L=5, n=4)  
 4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5  
 4.gauche  $\leftarrow \emptyset$   
 3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow$  3, L  $\leftarrow$  2, n  $\leftarrow$  7

```

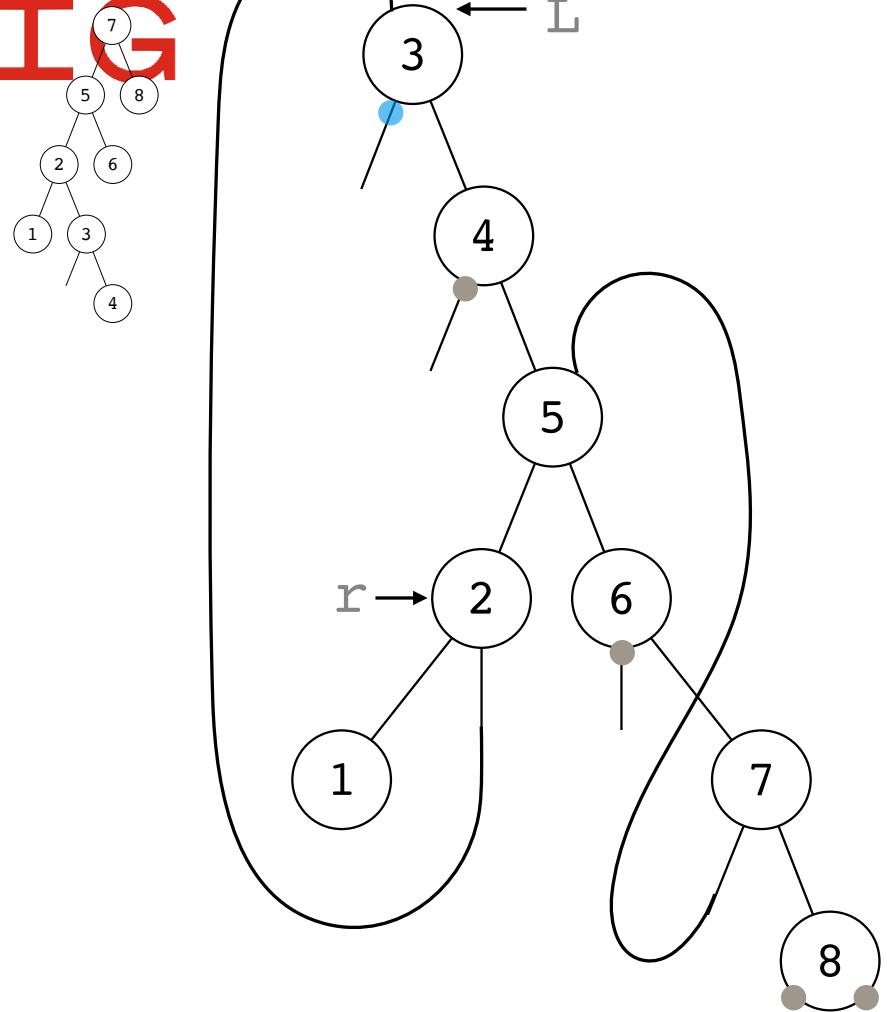
fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow \emptyset$ 
  
```



5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4  
 linéariser(r=2, L=5, n=4)  
 linéariser(r=3, L=5, n=4)  
 linéariser(r=4, L=5, n=4)  
 4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5  
 4.gauche  $\leftarrow$   $\emptyset$   
 3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow$   $\emptyset$   
 2.droit  $\leftarrow$  3, L  $\leftarrow$  2, n  $\leftarrow$  7

```

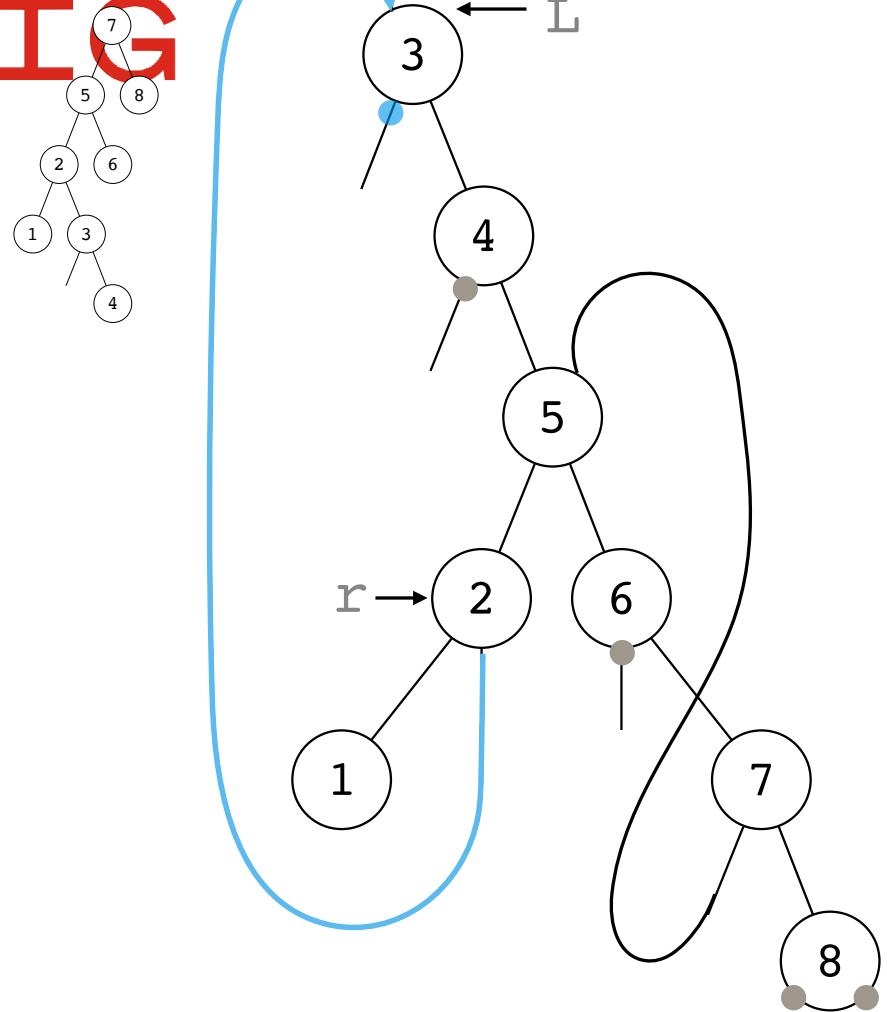
fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow$   $\emptyset$ 
  
```



5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4  
 linéariser(r=2, L=5, n=4)  
 linéariser(r=3, L=5, n=4)  
 linéariser(r=4, L=5, n=4)  
 4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5  
 4.gauche  $\leftarrow \emptyset$   
 3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow$  3, L  $\leftarrow$  2, n  $\leftarrow$  7

```

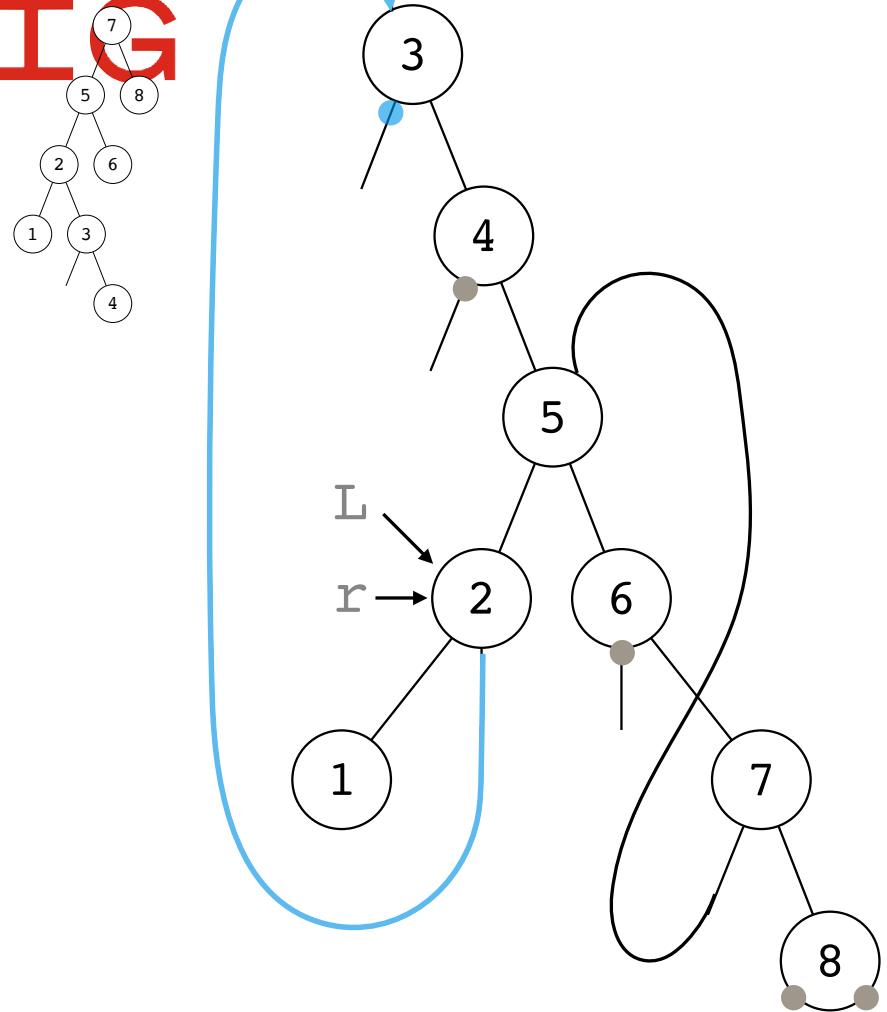
fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow \emptyset$ 
  
```



5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4  
 linéariser(r=2, L=5, n=4)  
 linéariser(r=3, L=5, n=4)  
 linéariser(r=4, L=5, n=4)  
 4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5  
 4.gauche  $\leftarrow \emptyset$   
 3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow$  3, L  $\leftarrow$  2, n  $\leftarrow$  7

```

fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow \emptyset$ 
  
```



5.droit  $\leftarrow 6$ , L  $\leftarrow 5$ , n  $\leftarrow 4$

linéariser(r=2, L=5, n=4)

linéariser(r=3, L=5, n=4)

linéariser(r=4, L=5, n=4)

4.droit  $\leftarrow 5$ , L  $\leftarrow 4$ , n  $\leftarrow 5$

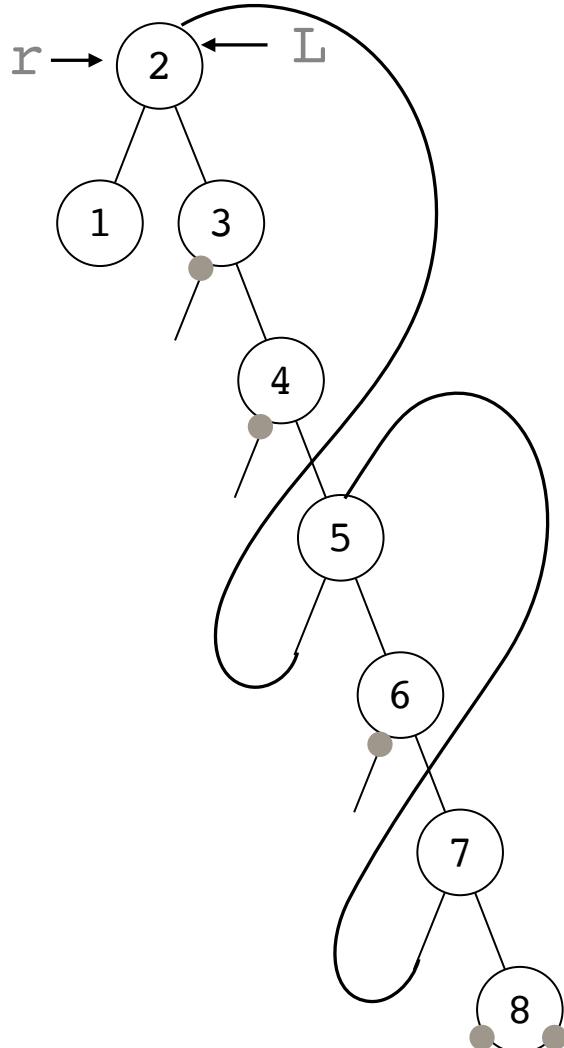
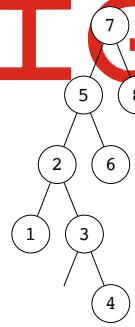
4.gauche  $\leftarrow \emptyset$

3.droit  $\leftarrow 4$ , L  $\leftarrow 3$ , n  $\leftarrow 6$

3.gauche  $\leftarrow \emptyset$

2.droit  $\leftarrow 3$ , L  $\leftarrow 2$ , n  $\leftarrow 7$

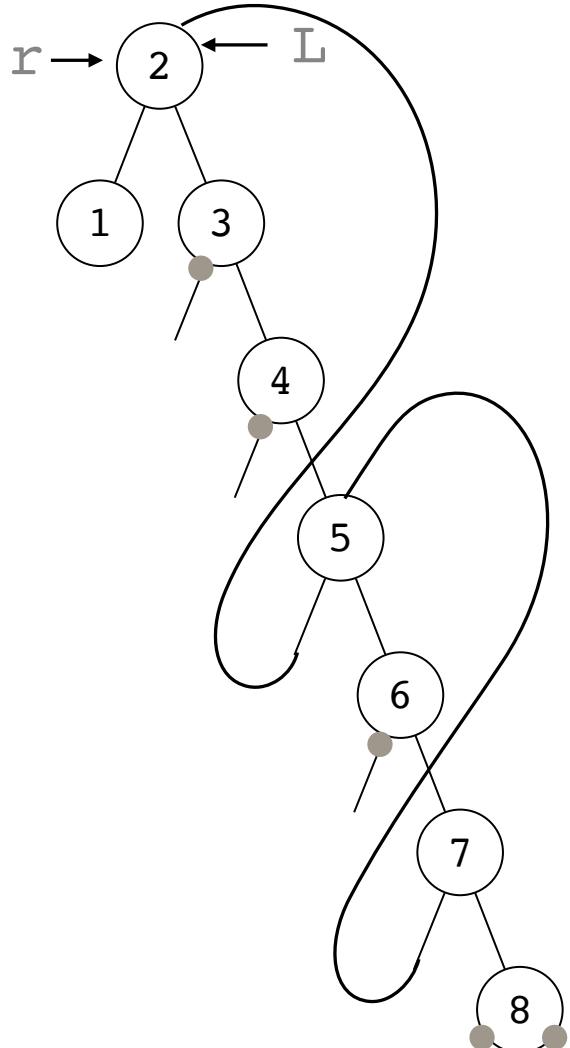
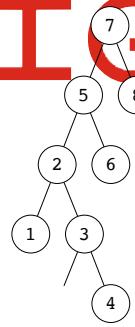
```
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
```



5.droit  $\leftarrow$  6, L  $\leftarrow$  5, n  $\leftarrow$  4  
**linéariser(r=2, L=5, n=4)**  
**linéariser(r=3, L=5, n=4)**  
**linéariser(r=4, L=5, n=4)**  
 4.droit  $\leftarrow$  5, L  $\leftarrow$  4, n  $\leftarrow$  5  
 4.gauche  $\leftarrow$   $\emptyset$   
 3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow$   $\emptyset$   
 2.droit  $\leftarrow$  1, L  $\leftarrow$  2, n  $\leftarrow$  7  
**linéariser(r=1, L=2, n=7)**  
 1.droit  $\leftarrow$  2, L  $\leftarrow$  1, n  $\leftarrow$  7

```

fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow$   $\emptyset$ 
  
```



linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

$4.\text{gauche} \leftarrow \emptyset$

$3.\text{droit} \leftarrow 4, L \leftarrow 3, n \leftarrow 6$

$3.\text{gauche} \leftarrow \emptyset$

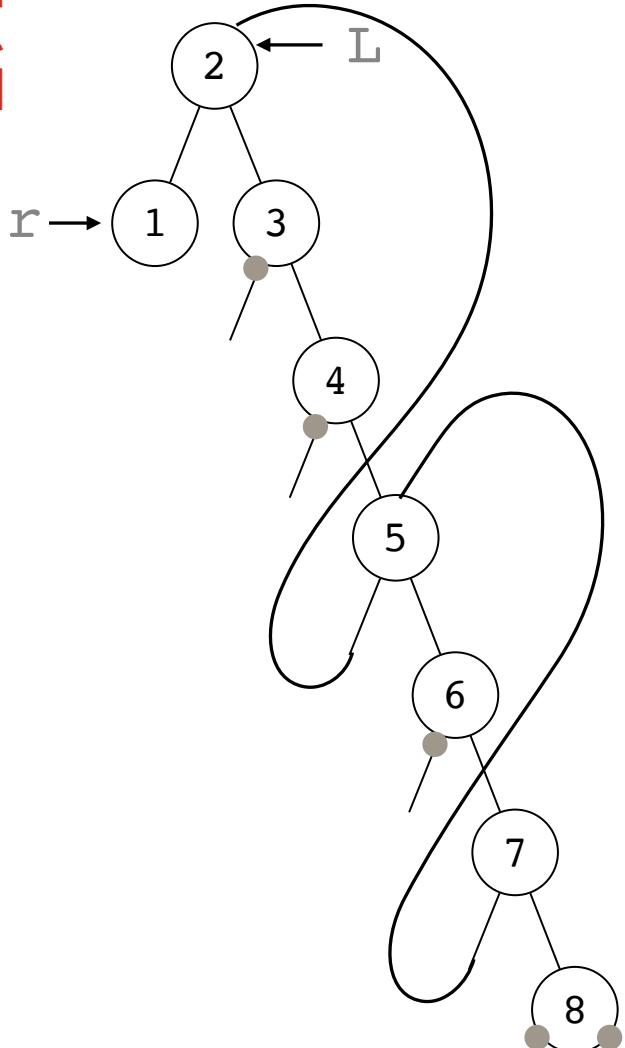
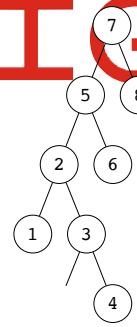
$2.\text{droit} \leftarrow 1, L \leftarrow 2, n \leftarrow 7$

linéariser( $r=1, L=2, n=7$ )

$1.\text{droit} \leftarrow 2, L \leftarrow 1, n \leftarrow 7$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

$4.\text{gauche} \leftarrow \emptyset$

$3.\text{droit} \leftarrow 4, L \leftarrow 3, n \leftarrow 6$

$3.\text{gauche} \leftarrow \emptyset$

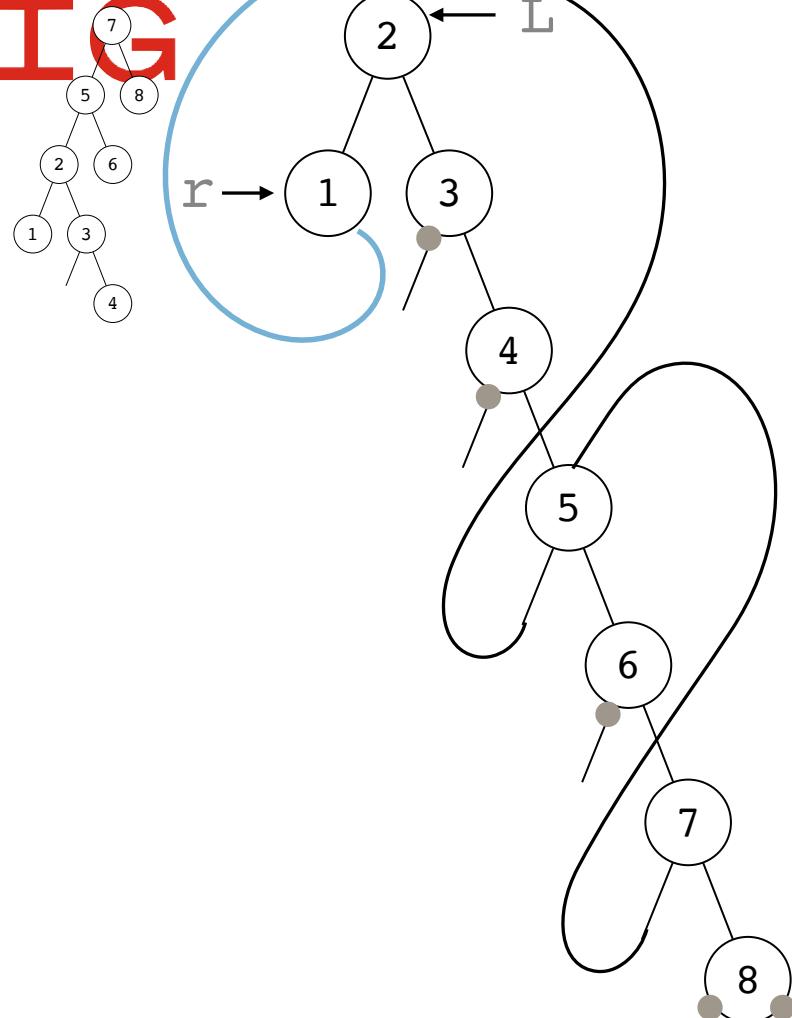
$2.\text{droit} \leftarrow 1, L \leftarrow 2, n \leftarrow 7$

linéariser( $r=1, L=2, n=7$ )

$1.\text{droit} \leftarrow 2, L \leftarrow 1, n \leftarrow 7$

```

fonction linéariser (r, ref L, ref n)
  si r != ∅,
    linéariser(r.droit, L, n)
    r.droit ← L, L ← r, n ← n + 1
    linéariser(r.gauche, L, n)
    r.gauche ← ∅
  
```



linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

$4.\text{gauche} \leftarrow \emptyset$

$3.\text{droit} \leftarrow 4, L \leftarrow 3, n \leftarrow 6$

$3.\text{gauche} \leftarrow \emptyset$

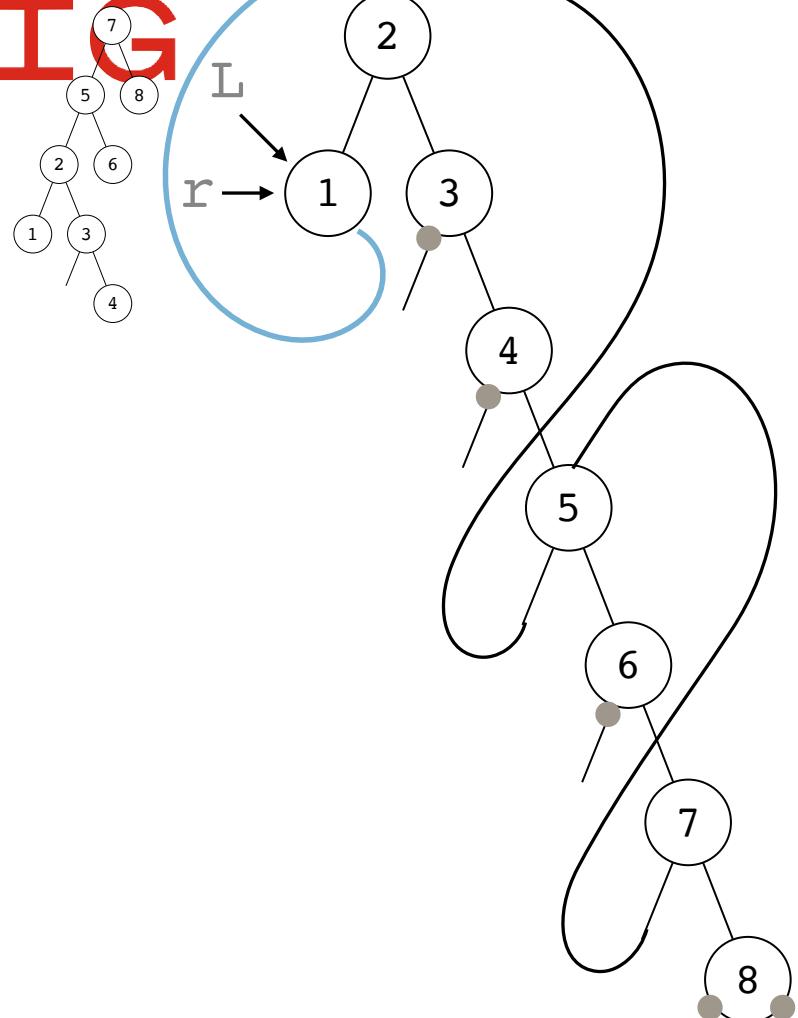
$2.\text{droit} \leftarrow 1, L \leftarrow 2, n \leftarrow 7$

linéariser( $r=1, L=2, n=7$ )

$1.\text{droit} \leftarrow 2, L \leftarrow 1, n \leftarrow 7$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=3, L=5, n=4$ )

linéariser( $r=4, L=5, n=4$ )

$4.\text{droit} \leftarrow 5, L \leftarrow 4, n \leftarrow 5$

$4.\text{gauche} \leftarrow \emptyset$

$3.\text{droit} \leftarrow 4, L \leftarrow 3, n \leftarrow 6$

$3.\text{gauche} \leftarrow \emptyset$

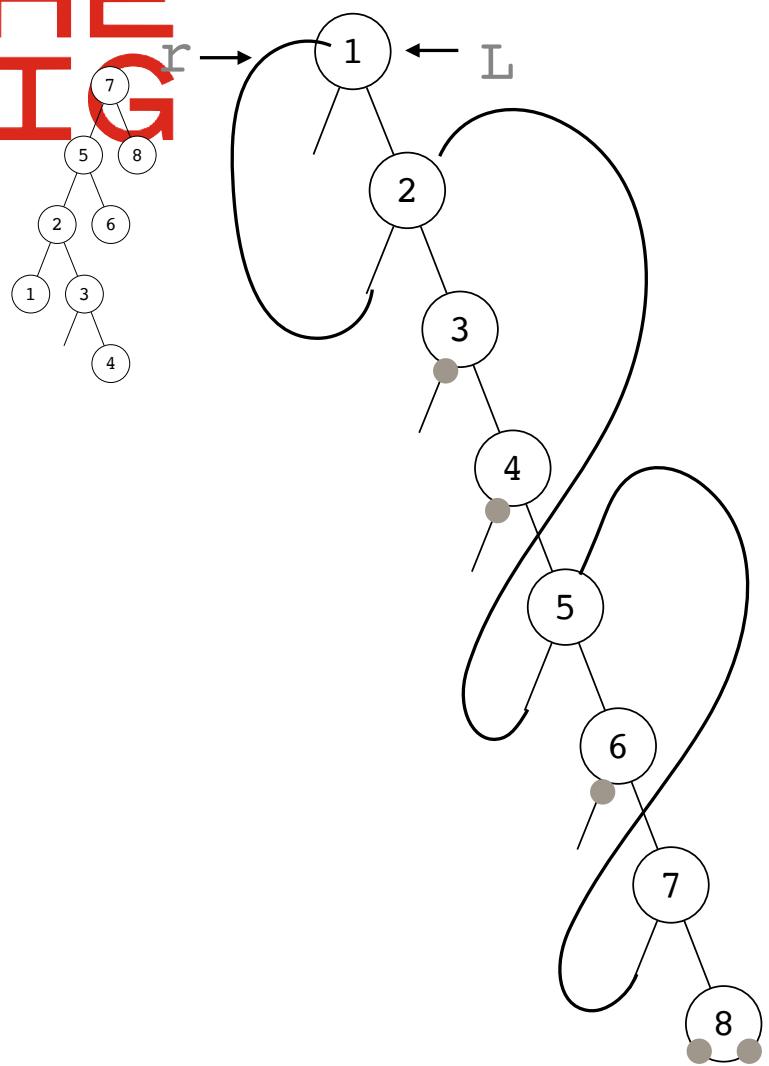
$2.\text{droit} \leftarrow 1, L \leftarrow 2, n \leftarrow 7$

linéariser( $r=1, L=2, n=7$ )

$1.\text{droit} \leftarrow 2, L \leftarrow 1, n \leftarrow 7$

```

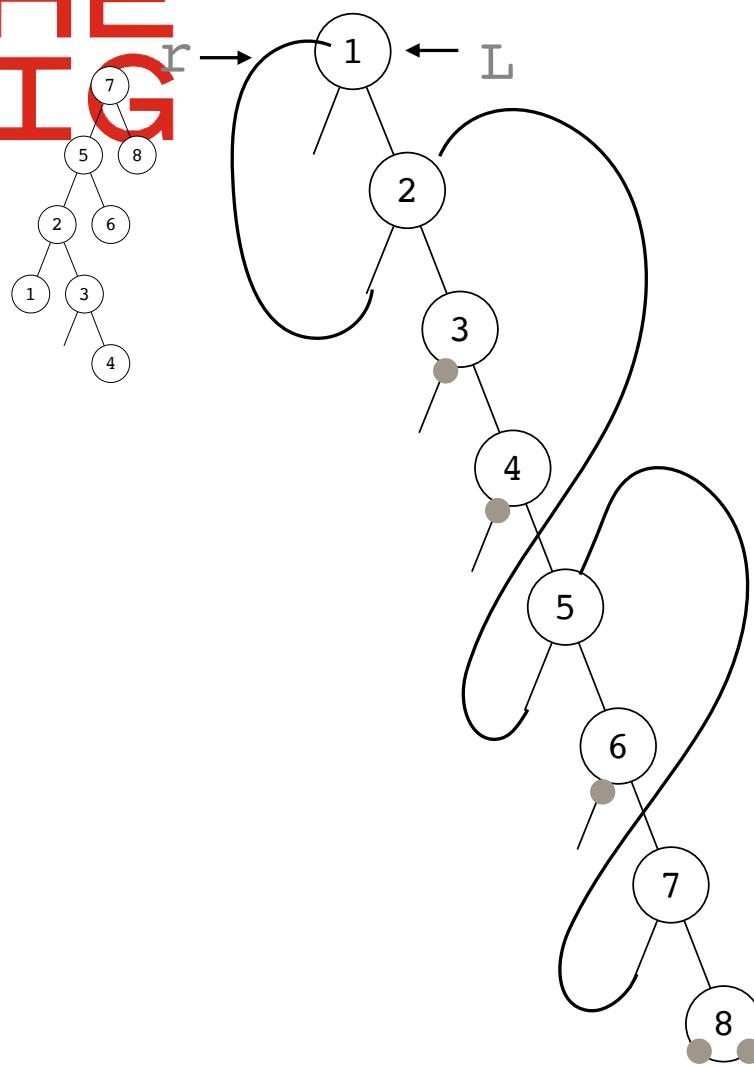
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit ← L, L ← r, n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← ∅
    
```



linéariser( $r=3$ ,  $L=5$ ,  $n=4$ )  
 linéariser( $r=4$ ,  $L=5$ ,  $n=4$ )  
 4.droit  $\leftarrow 5$ ,  $L \leftarrow 4$ ,  $n \leftarrow 5$   
 4.gauche  $\leftarrow \emptyset$   
 3.droit  $\leftarrow 4$ ,  $L \leftarrow 3$ ,  $n \leftarrow 6$   
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow 1$ ,  $L \leftarrow 2$ ,  $n \leftarrow 7$   
 linéariser( $r=1$ ,  $L=2$ ,  $n=7$ )  
 1.droit  $\leftarrow 2$ ,  $L \leftarrow 1$ ,  $n \leftarrow 7$   
 1.gauche  $\leftarrow \emptyset$   
 2.gauche  $\leftarrow \emptyset$

```

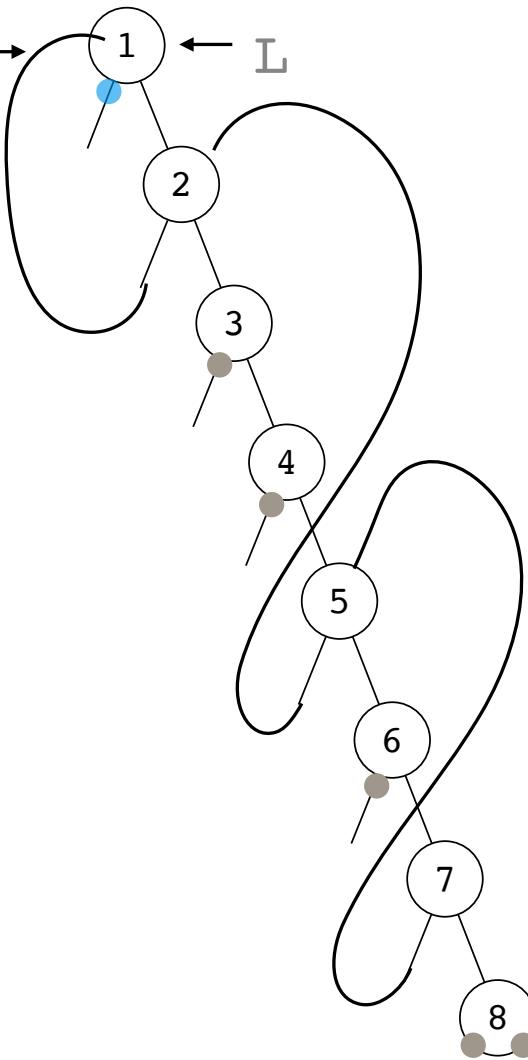
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow L$ ,  $L \leftarrow r$ ,  $n \leftarrow n + 1$ 
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow \emptyset$ 
    
```



3.droit  $\leftarrow 4, L \leftarrow 3, n \leftarrow 6$   
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow 1, L \leftarrow 2, n \leftarrow 7$   
 linéariser( $r=1, L=2, n=7$ )  
 1.droit  $\leftarrow 2, L \leftarrow 1, n \leftarrow 7$   
 1.gauche  $\leftarrow \emptyset$   
 2.gauche  $\leftarrow \emptyset$   
 5.gauche  $\leftarrow \emptyset$   
 7.gauche  $\leftarrow \emptyset$

```

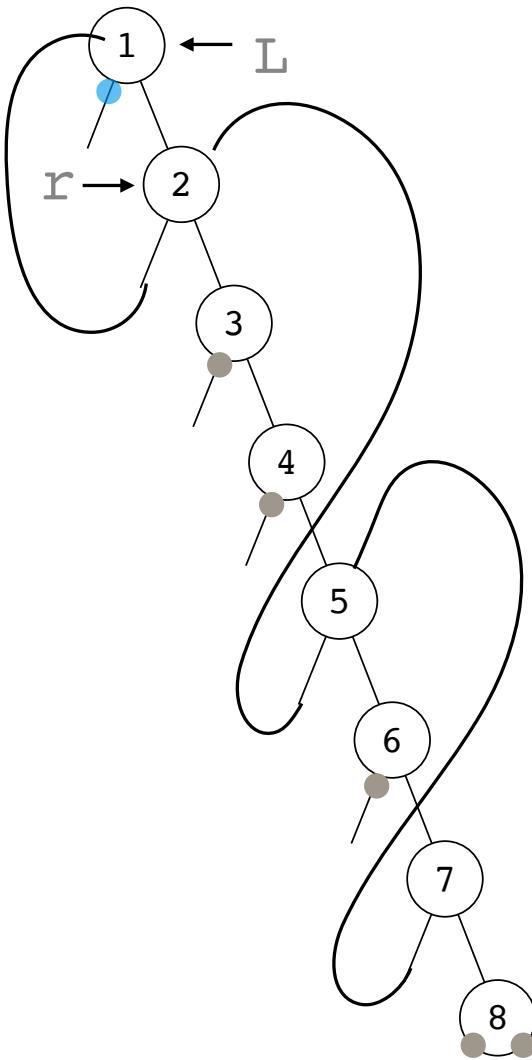
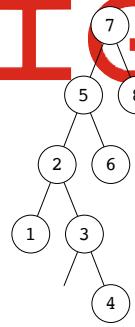
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow L, L \leftarrow r, n \leftarrow n + 1$ 
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow \emptyset$ 
    
```



3.droit  $\leftarrow 4$ , L  $\leftarrow 3$ , n  $\leftarrow 6$   
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow 1$ , L  $\leftarrow 2$ , n  $\leftarrow 7$   
 linéariser(r=1, L=2, n=7)  
 1.droit  $\leftarrow 2$ , L  $\leftarrow 1$ , n  $\leftarrow 7$   
 1.gauche  $\leftarrow \emptyset$   
 2.gauche  $\leftarrow \emptyset$   
 5.gauche  $\leftarrow \emptyset$   
 7.gauche  $\leftarrow \emptyset$

```

fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$  ∅
    
```



```

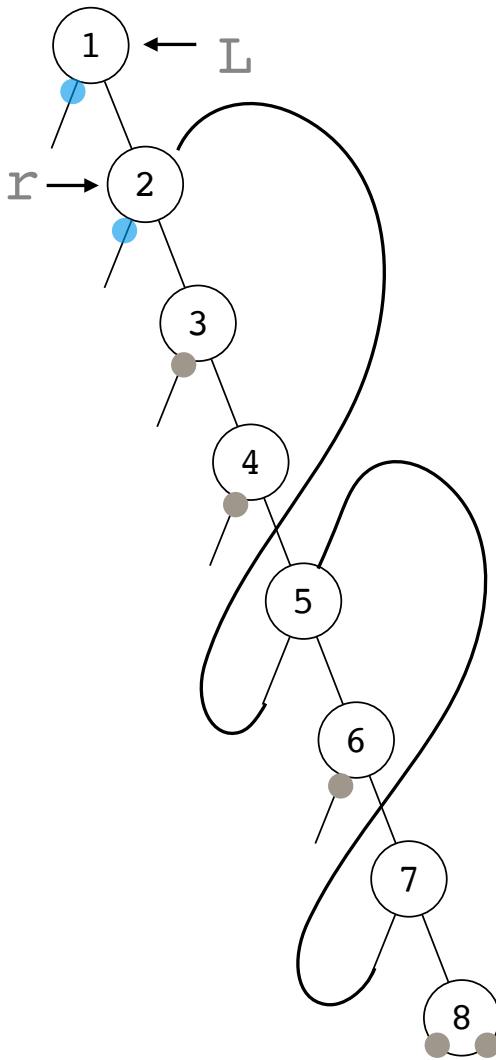
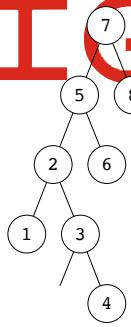
3.droit ← 4,   L ← 3,   n ← 6
3.gauche ← Ø
2.droit ← 1,   L ← 2,   n ← 7
linéariser(r=1, L=2, n=7)
1.droit ← 2,   L ← 1,   n ← 7
1.gauche ← Ø
2.gauche ← Ø
5.gauche ← Ø
7.gauche ← Ø

```

```

fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L,   L ← r,   n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø

```



```

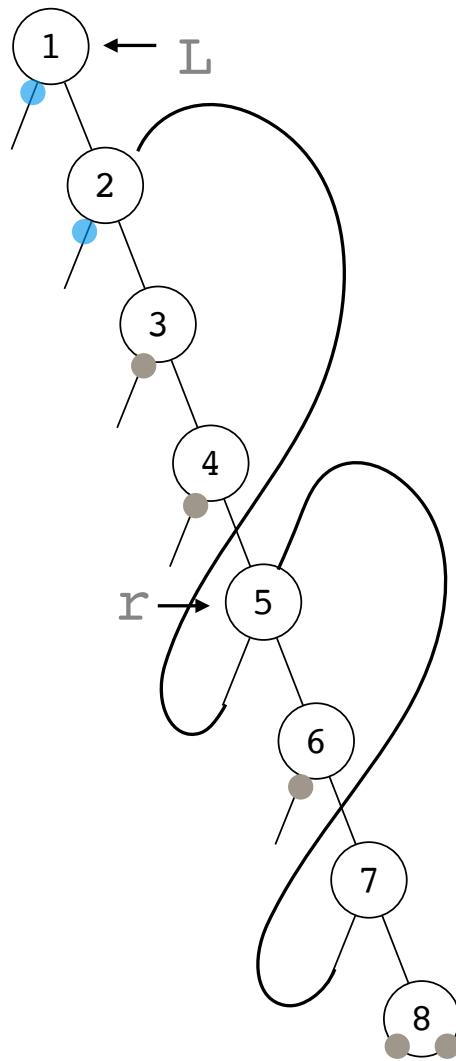
3.droit ← 4,   L ← 3,   n ← 6
3.gauche ← Ø
2.droit ← 1,   L ← 2,   n ← 7
linéariser(r=1, L=2, n=7)
1.droit ← 2,   L ← 1,   n ← 7
1.gauche ← Ø
2.gauche ← Ø
5.gauche ← Ø
7.gauche ← Ø

```

```

fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L,   L ← r,   n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø

```



```

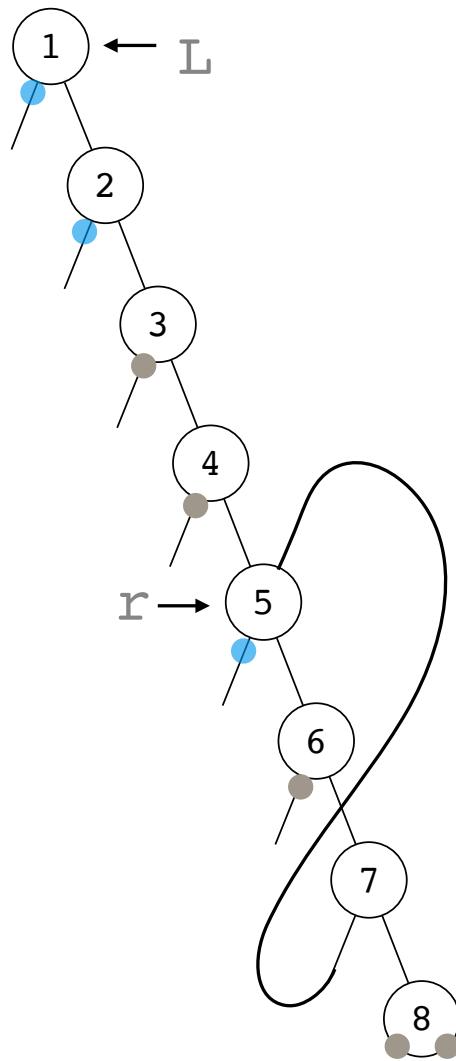
3.droit ← 4,   L ← 3,   n ← 6
3.gauche ← Ø
2.droit ← 1,   L ← 2,   n ← 7
linéariser(r=1, L=2, n=7)
1.droit ← 2,   L ← 1,   n ← 7
1.gauche ← Ø
2.gauche ← Ø
5.gauche ← Ø
7.gauche ← Ø

```

```

fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L,   L ← r,   n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø

```



```

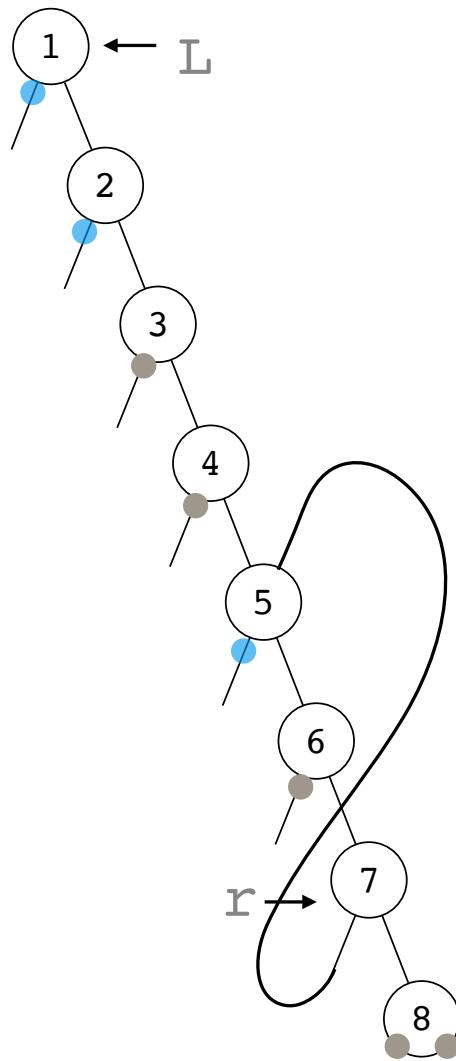
3.droit ← 4,   L ← 3,   n ← 6
3.gauche ← Ø
2.droit ← 1,   L ← 2,   n ← 7
linéariser(r=1, L=2, n=7)
1.droit ← 2,   L ← 1,   n ← 7
1.gauche ← Ø
2.gauche ← Ø
5.gauche ← Ø
7.gauche ← Ø

```

```

fonction linéariser (r, ref L, ref n)
    si r != Ø,
        linéariser(r.droit, L, n)
        r.droit ← L,   L ← r,   n ← n + 1
        linéariser(r.gauche, L, n)
        r.gauche ← Ø

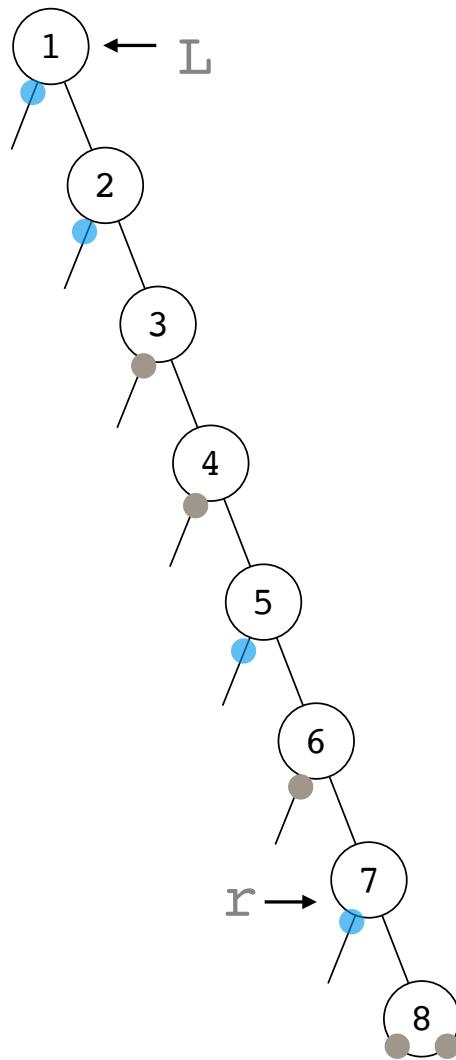
```



3.droit  $\leftarrow 4, L \leftarrow 3, n \leftarrow 6$   
 3.gauche  $\leftarrow \emptyset$   
 2.droit  $\leftarrow 1, L \leftarrow 2, n \leftarrow 7$   
 linéariser(r=1, L=2, n=7)  
 1.droit  $\leftarrow 2, L \leftarrow 1, n \leftarrow 7$   
 1.gauche  $\leftarrow \emptyset$   
 2.gauche  $\leftarrow \emptyset$   
 5.gauche  $\leftarrow \emptyset$   
 7.gauche  $\leftarrow \emptyset$

```

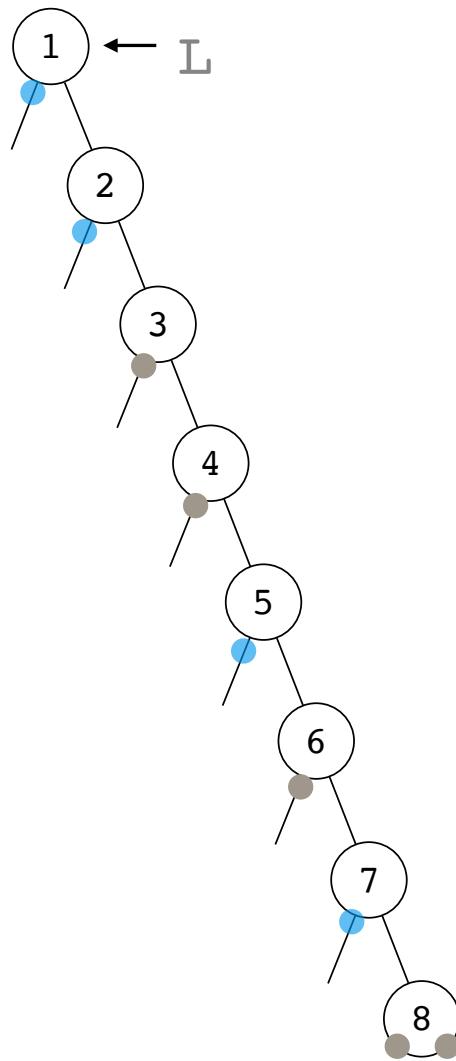
fonction linéariser (r, ref L, ref n)
    si r != ∅,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow L, L \leftarrow r, n \leftarrow n + 1$ 
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow \emptyset$ 
    
```



3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow$   $\emptyset$   
 2.droit  $\leftarrow$  1, L  $\leftarrow$  2, n  $\leftarrow$  7  
 linéariser(r=1, L=2, n=7)  
 1.droit  $\leftarrow$  2, L  $\leftarrow$  1, n  $\leftarrow$  7  
 1.gauche  $\leftarrow$   $\emptyset$   
 2.gauche  $\leftarrow$   $\emptyset$   
 5.gauche  $\leftarrow$   $\emptyset$   
 7.gauche  $\leftarrow$   $\emptyset$

```

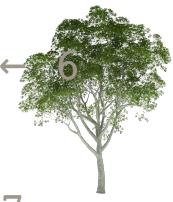
fonction linéariser (r, ref L, ref n)
  si r !=  $\emptyset$ ,
    linéariser(r.droit, L, n)
    r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
    linéariser(r.gauche, L, n)
    r.gauche  $\leftarrow$   $\emptyset$ 
  
```



3.droit  $\leftarrow$  4, L  $\leftarrow$  3, n  $\leftarrow$  6  
 3.gauche  $\leftarrow$   $\emptyset$   
 2.droit  $\leftarrow$  1, L  $\leftarrow$  2, n  $\leftarrow$  7  
 linéariser(r=1, L=2, n=7)  
 1.droit  $\leftarrow$  2, L  $\leftarrow$  1, n  $\leftarrow$  7  
 1.gauche  $\leftarrow$   $\emptyset$   
 2.gauche  $\leftarrow$   $\emptyset$   
 5.gauche  $\leftarrow$   $\emptyset$   
 7.gauche  $\leftarrow$   $\emptyset$

```

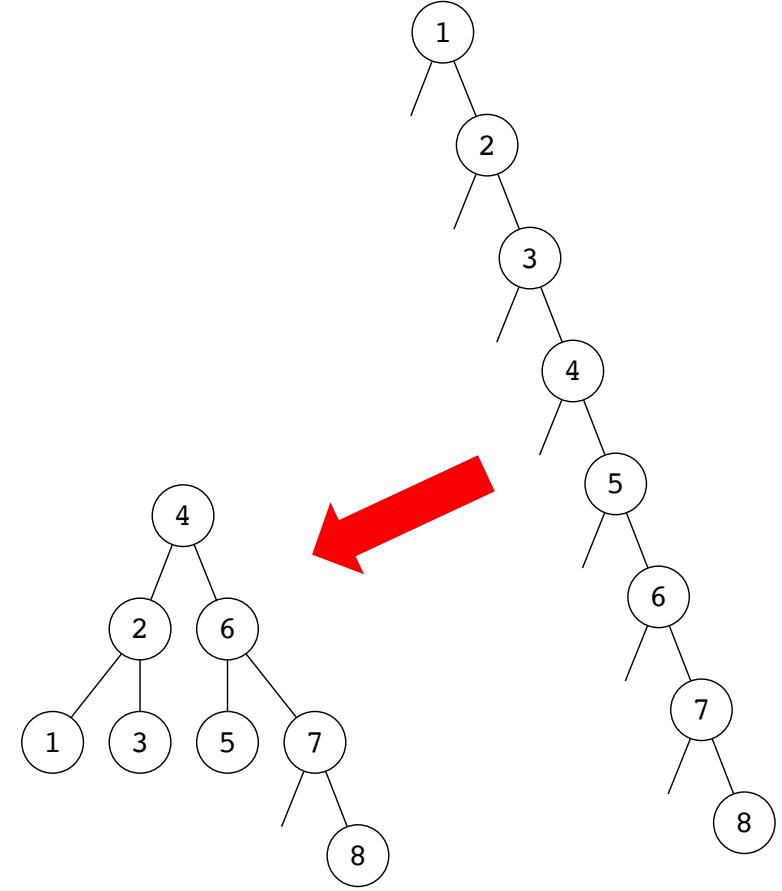
fonction linéariser (r, ref L, ref n)
    si r !=  $\emptyset$ ,
        linéariser(r.droit, L, n)
        r.droit  $\leftarrow$  L, L  $\leftarrow$  r, n  $\leftarrow$  n + 1
        linéariser(r.gauche, L, n)
        r.gauche  $\leftarrow$   $\emptyset$ 
    
```





# Arborisation

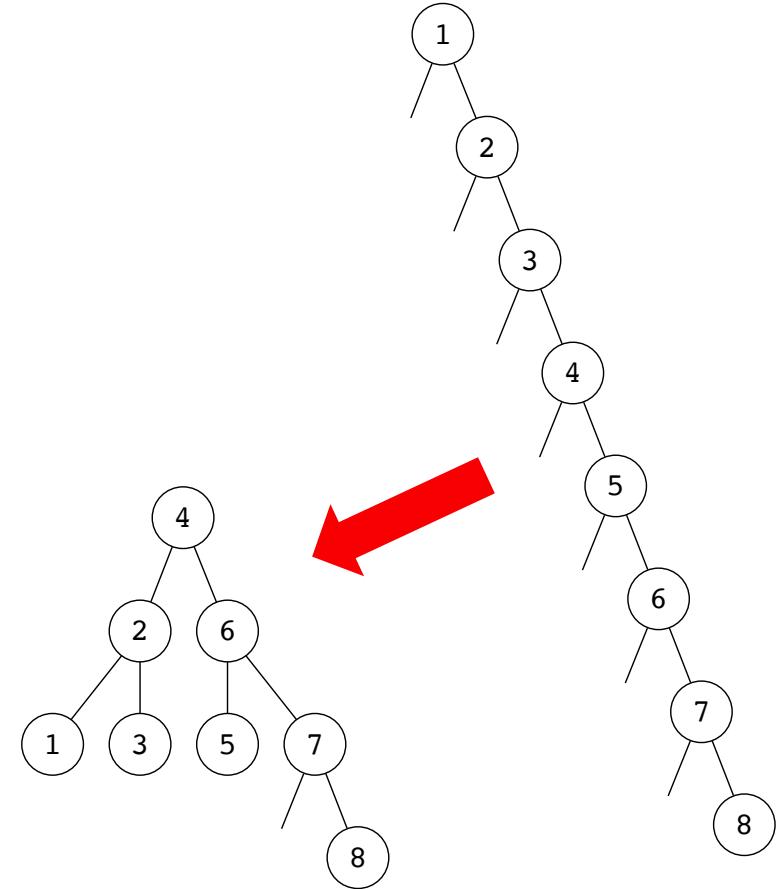
- Réorganise l'arbre dégénéré à droite
- Prend l'élément médian comme racine
- Arborise récursivement les sous-arbres gauche et droit





# Arborisation

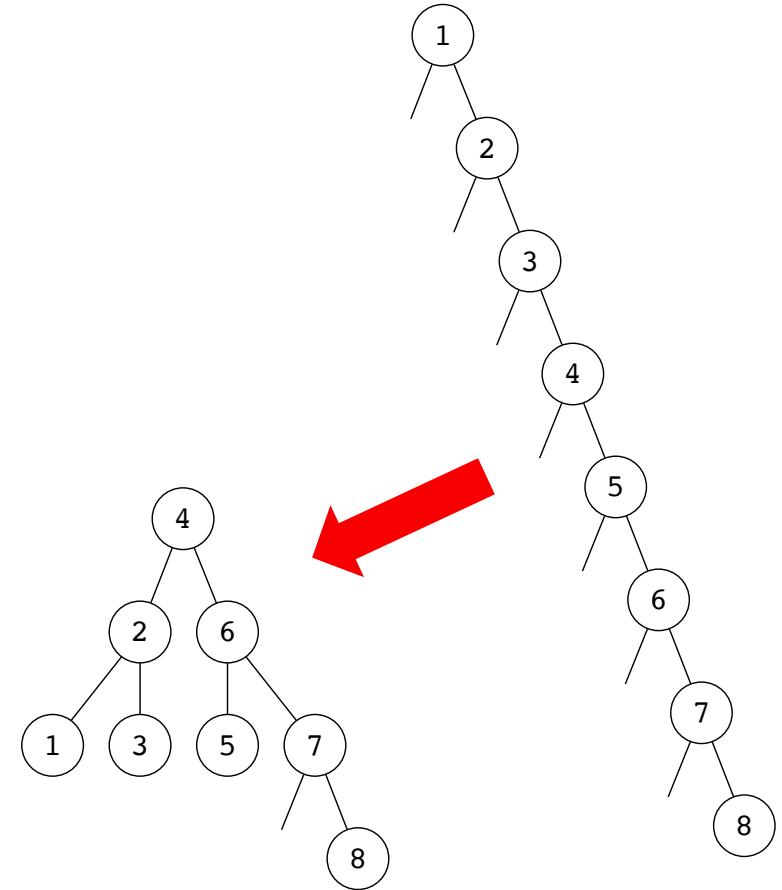
- Paramètres d'entrée





# Arborisation

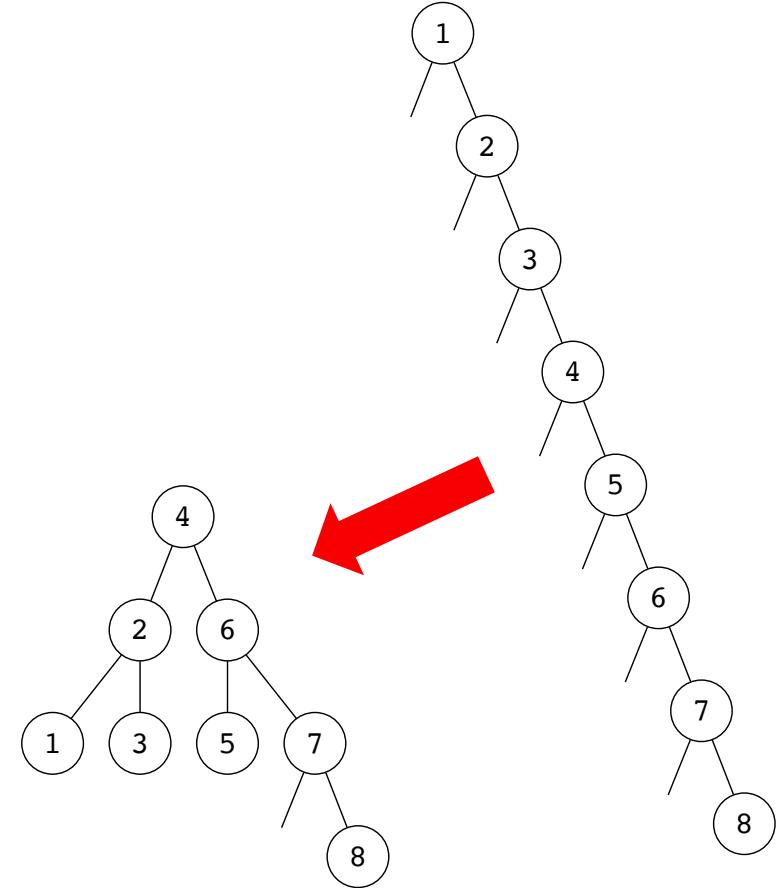
- Paramètres d'entrée
  - l'arbre dégénéré L dont tous les enfants gauches sont vides





# Arborisation

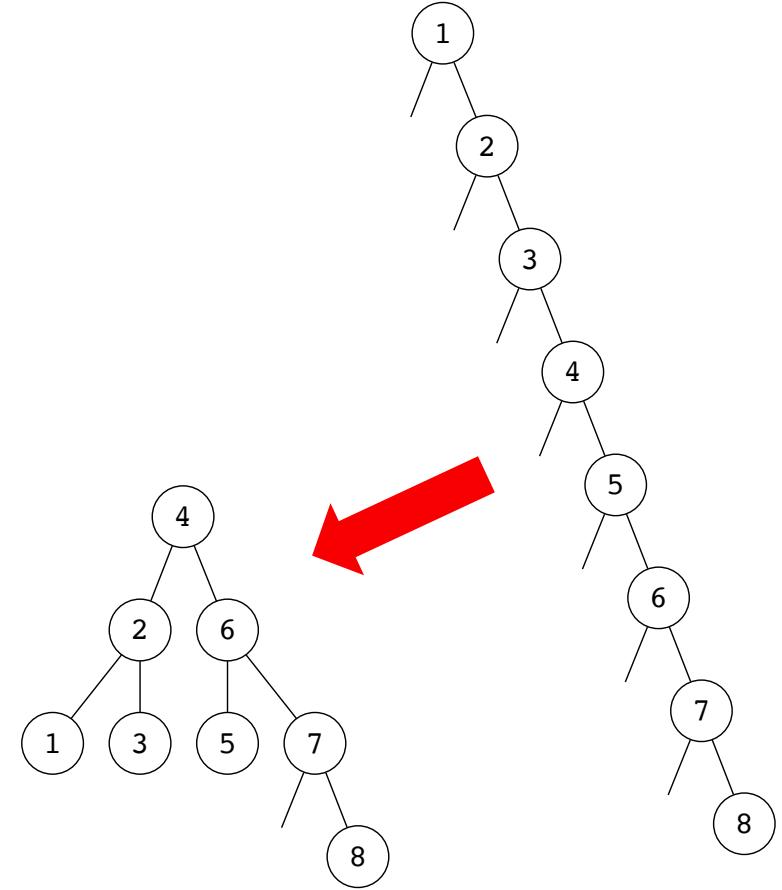
- Paramètres d'entrée
  - l'arbre dégénéré L dont tous les enfants gauches sont vides
  - Le nombre n d'éléments à traiter dans L





# Arborisation

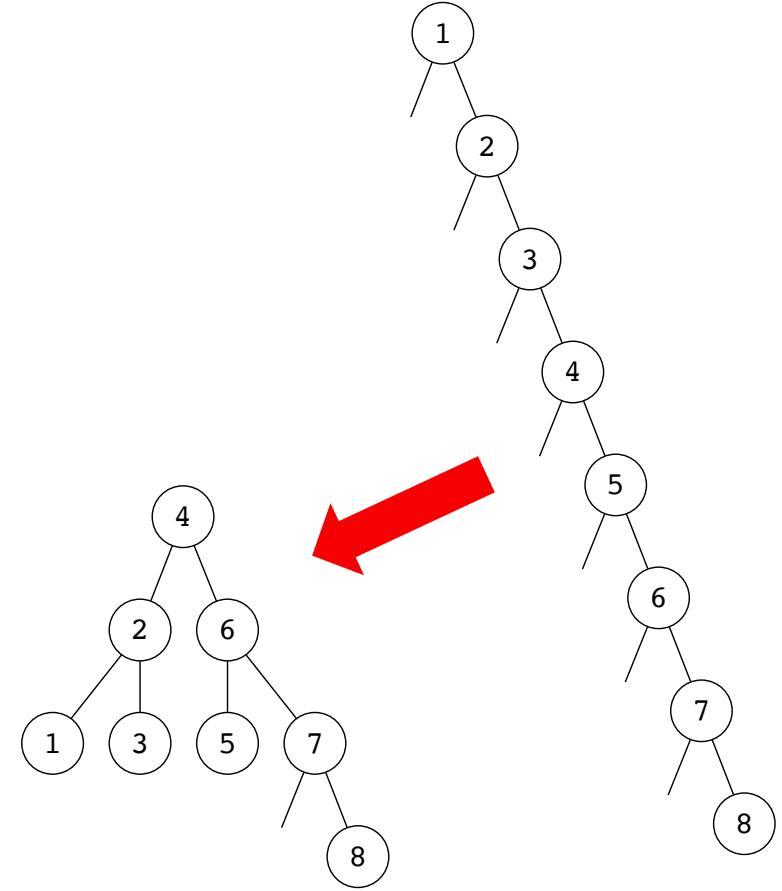
- Paramètres d'entrée
  - l'arbre dégénéré L dont tous les enfants gauches sont vides
  - Le nombre n d'éléments à traiter dans L
- Retourne l'élément médian comme racine de l'arbre arborisé





# Arborisation

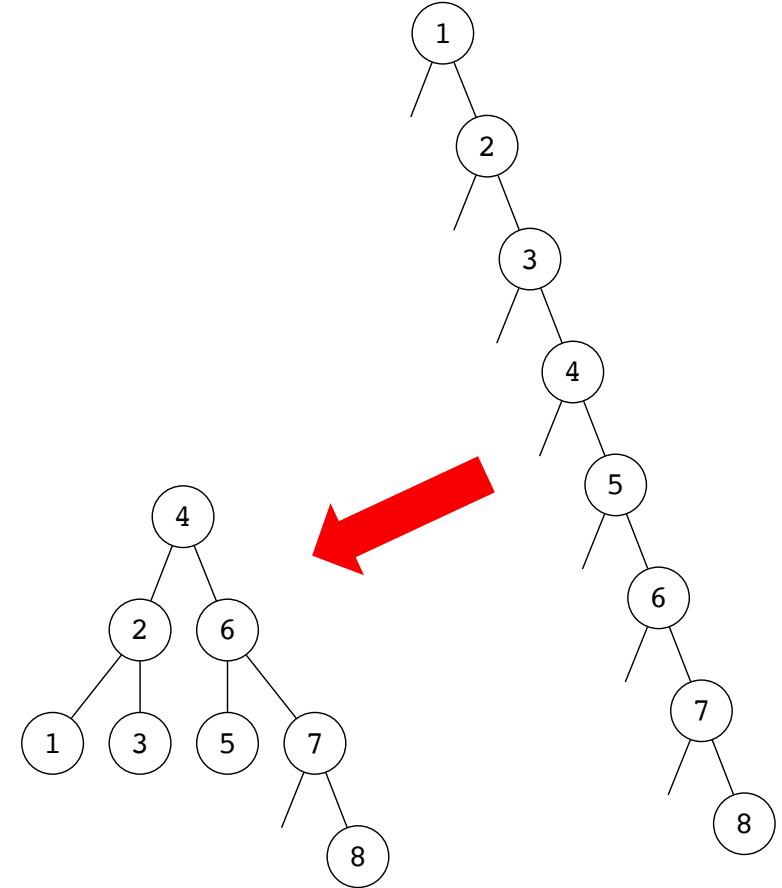
- Paramètres d'entrée
  - l'arbre dégénéré L dont tous les enfants gauches sont vides
  - Le nombre n d'éléments à traiter dans L
- Retourne l'élément médian comme racine de l'arbre arborisé
- Pour atteindre l'élément médian de la liste, il faut d'abord traiter les  $(n-1)/2$  éléments de la liste qui forment le sous-arbre droit arborisé.





# Arborisation

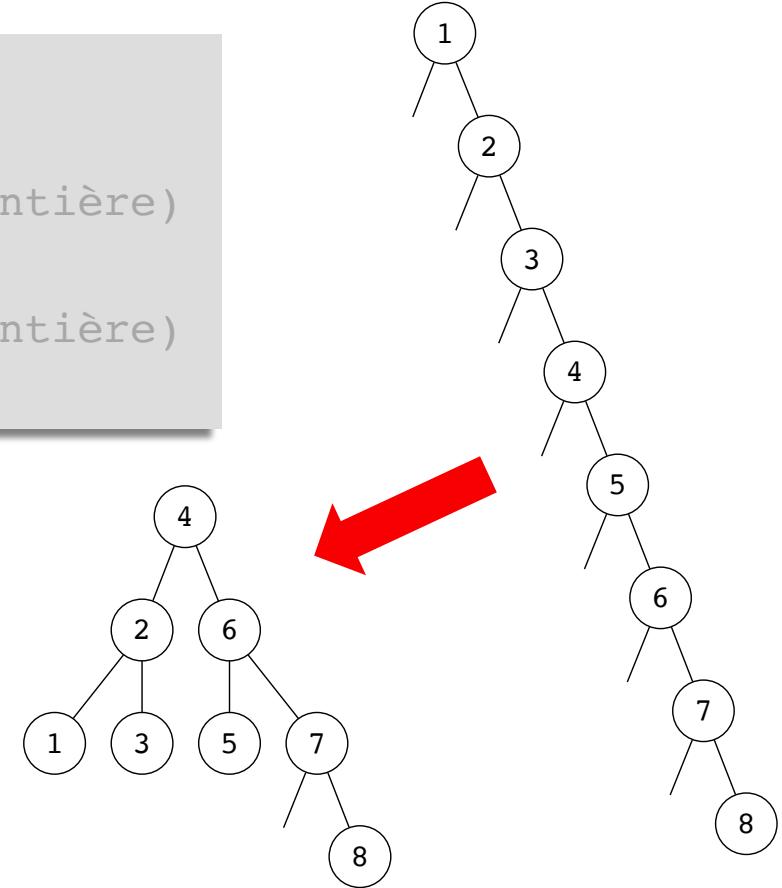
- Paramètres d'entrée
  - l'arbre dégénéré L dont tous les enfants gauches sont vides
  - Le nombre n d'éléments à traiter dans L
- Retourne l'élément médian comme racine de l'arbre arborisé
- Pour atteindre l'élément médian de la liste, il faut d'abord traiter les  $(n-1)/2$  éléments de la liste qui forment le sous-arbre droit arborisé.
- Parcours symétrique de l'arbre résultat





# Arborisation

```
fonction arboriser (ref L, n)
    si n != 0,
        rg ← arboriser(L, (n-1)/2)      (division entière)
        r ← L,  r.gauche ← rg,  L ← L.droit
        r.droit ← arboriser(L, n/2)      (division entière)
    retourner r
```





# Equilibrage

- Linéarisation puis arborisation

```
fonction équilibrer (r)
    L ← Ø,    n ← 0
    linéariser(r, L, n)
    retourner arboriser(L, n)
```



# Equilibrage

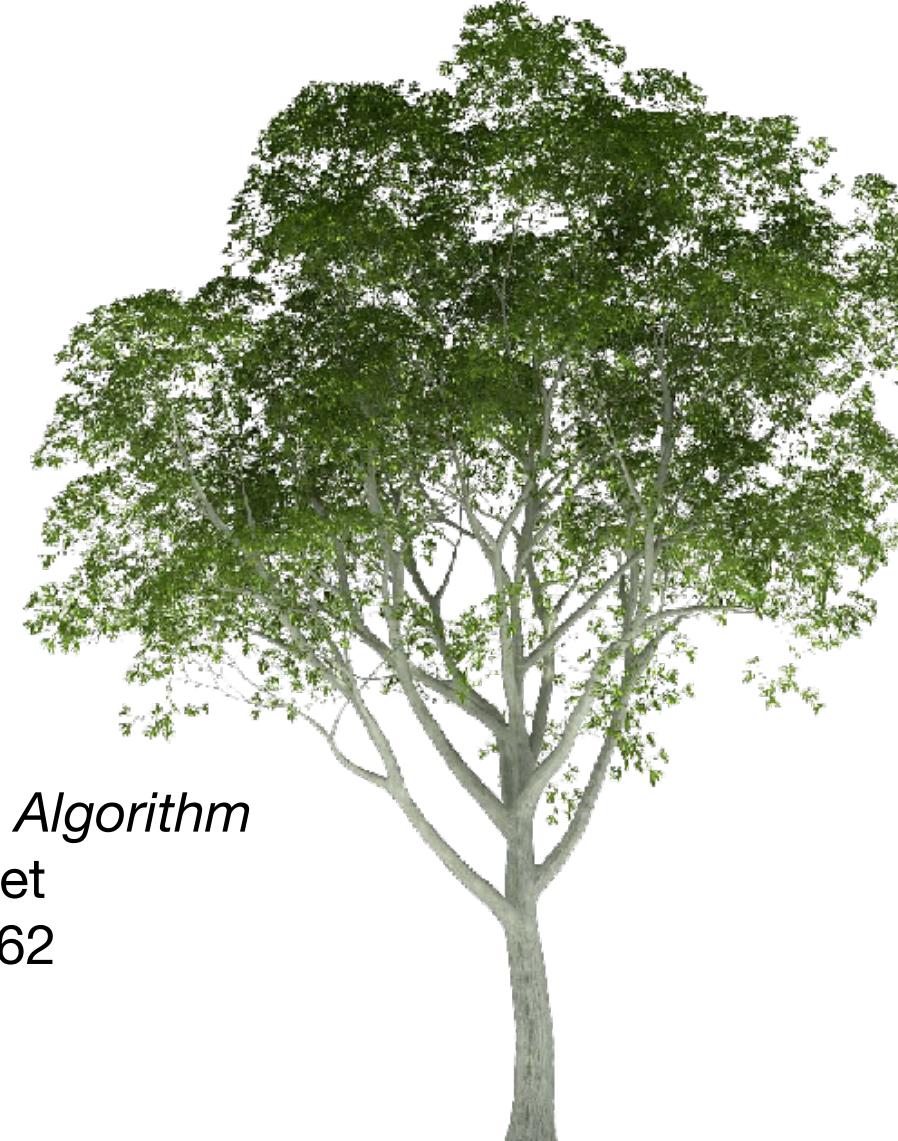
- Linéarisation puis arborisation

```
fonction équilibrer (r)
    L ← Ø,    n ← 0
    linéariser(r, L, n)
    retourner arboriser(L, n)
```

- Les 2 algorithmes sont des parcours de l'arbre, donc de complexité  $O(n)$

# 15. Arbres AVL

**G. Adelson-Velskii et E. M. Landis,** *An Algorithm for the Organization of Information.* Soviet Mathematics Doklady, 3:1259–1263, 1962





# Définition

- Un arbre AVL est
  - un arbre binaire de recherche
  - qui garantit qu'aucun noeud n'a un déséquilibre autre que -1, 0 ou 1
  - en se ré-équilibrant à chaque opération d'**insertion / suppression**

Déséquilibre : différence de hauteur entre les sous-arbres gauche et droit d'un noeud.



# Mesure de l'équilibre

- Rappel: on sait mesurer hauteur et équilibre en  $O(n)$

```
fonction hauteur (r)
    si r == Ø,
        retourner 0
    sinon,
        retourner 1 + max(hauteur(r.gauche), hauteur(r.droit))
```

```
fonction équilibre (r)
    si r == Ø, retourner 0
    sinon,
        retourner hauteur(r.gauche) - hauteur(r.droit)
```

- Pour le faire en  $O(1)$ , il faut modifier la structure des noeuds



# Noeuds

- Plus simple : stocker la hauteur du sous-arbre dont le noeud est la racine
  - mis à jour à chaque insertion / suppression / équilibrage
  - 8 bits suffisent, le nombre d'éléments stockés croît exponentiellement avec la hauteur
- Plus compact : stocker le déséquilibre seul sur 2 bits dans la racine, sur 1 bit dans ses enfants. (pas étudié ici)



# Noeuds

- Plus simple : stocker la hauteur du sous-arbre dont le noeud est la racine
  - mis à jour à chaque insertion / suppression / équilibrage
  - 8 bits suffisent, le nombre d'éléments stockés croît exponentiellement avec la hauteur
  - Plus compact : stocker le déséquilibre seul sur 2 bits dans la racine, sur 1 bit dans ses enfants. (pas étudié ici)

```
structure Noeud<K>
    K clé
    Noeud* gauche
    Noeud* droit
    Entier hauteur
```



# Hauteur

- La fonction **hauteur** est de complexité O(1)

```
fonction hauteur (r)
    si r == Ø, retourner 0
    sinon,     retourner r.hauteur
```

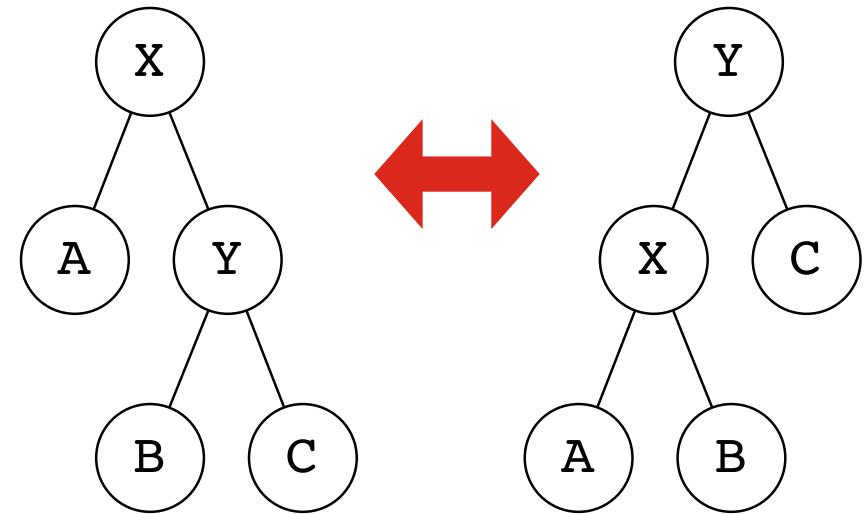
- Après toute opération qui modifie la hauteur d'un enfant de  $r$ , il faudra recalculer  $r.hauteur$  (en sortie de récursion)

```
fonction calculer_hauteur (r)
    si r != Ø,
        r.hauteur = 1 + max(hauteur(r.gauche), hauteur(r.droit))
```



# Principe de l'équilibrage

- Equilibrer dès qu'un déséquilibre de  $+2$  ou  $-2$  est détecté.
- Le ramener dans  $[-1, 1]$
- Aucun déséquilibre hors de  $[-2, 2]$  n'est donc possible
- Equilibrer se fait localement, par **rotation**
- On détecte et corrige d'abord les noeuds les plus profonds, puis leurs parents en **retour de récursion**



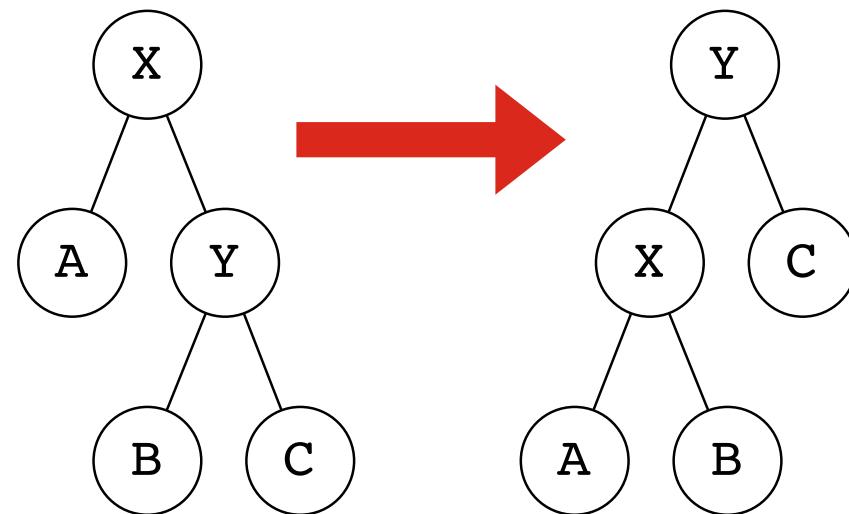


# Rotation gauche

- Faire monter l'enfant droit Y à la place de son parent X
- En conservant la relation d'arbre binaire de recherche

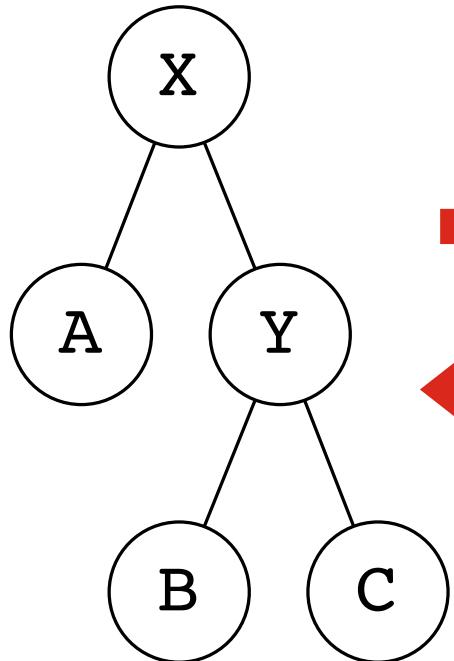
A < X < B < Y < C

- Rotation droite : l'inverse

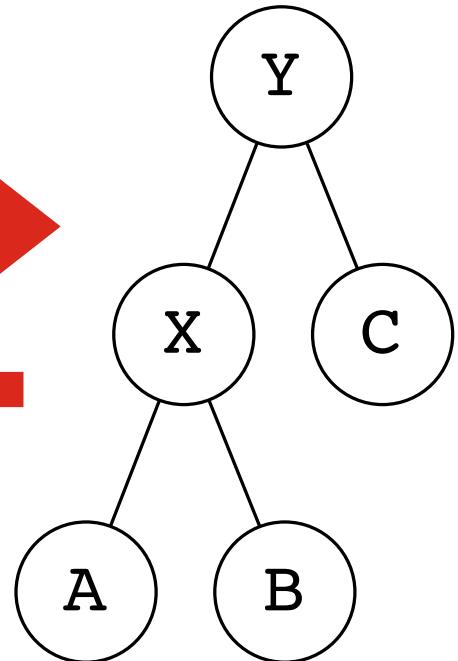




# Rotations



```
fonction rotation_gauche (ref r) // X
    t ← r.droit // Y
    r.droit ← t.gauche // B
    t.gauche ← r // X
    r ← t // Y
```

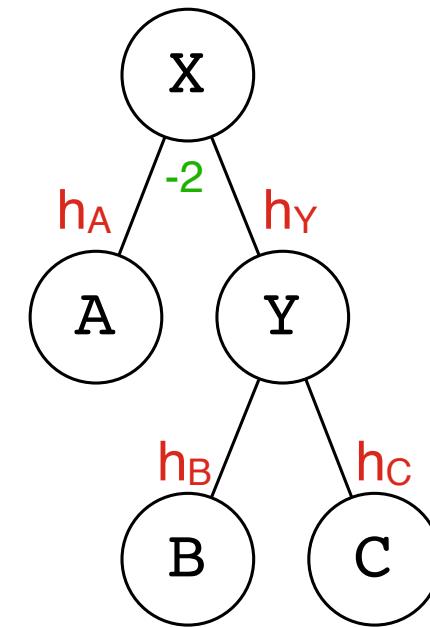


```
fonction rotation_droite (ref r) // Y
    t ← r.gauche // X
    r.gauche ← t.droit // B
    t.droit ← r // Y
    r ← t // X
```



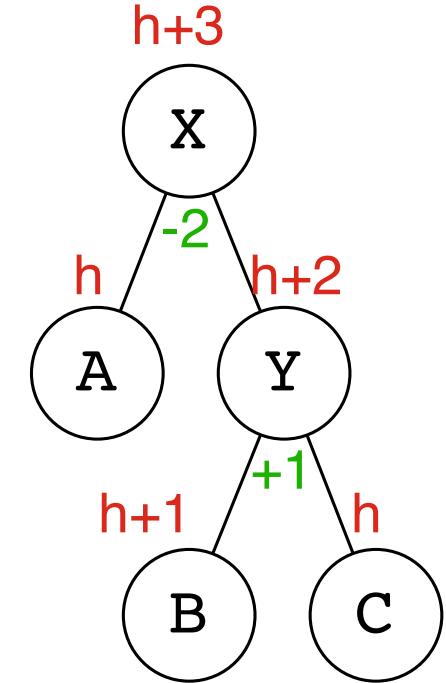
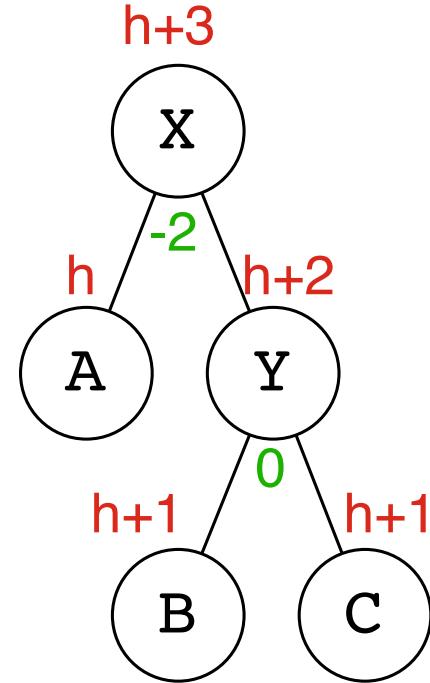
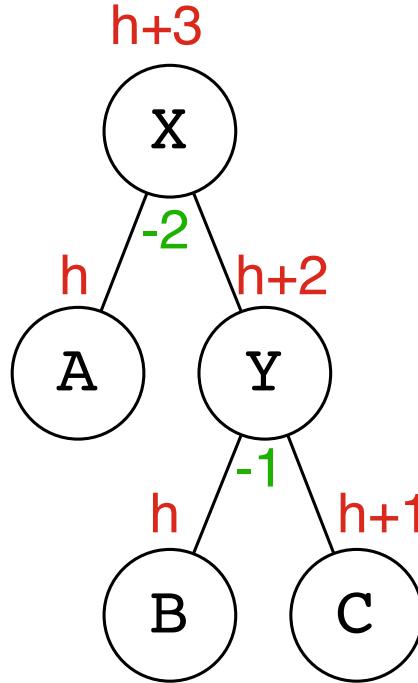
# Effet de la rotation

- A, B et C sont les racines de sous arbres de hauteur  $h_A, h_B, h_C$
- On effectue une rotation gauche pour corriger un déséquilibre à droite, donc
  - $\text{équilibre}(X) = h_A - h_Y = -2$
  - $h_Y = h_A + 2$
  - Par définition,  $h_Y = 1 + \max(h_B, h_C)$
  - Y n'étant pas déséquilibré, on a
    - $\text{équilibre}(Y) = h_B - h_C = +1, 0 \text{ ou } -1$





## 3 cas

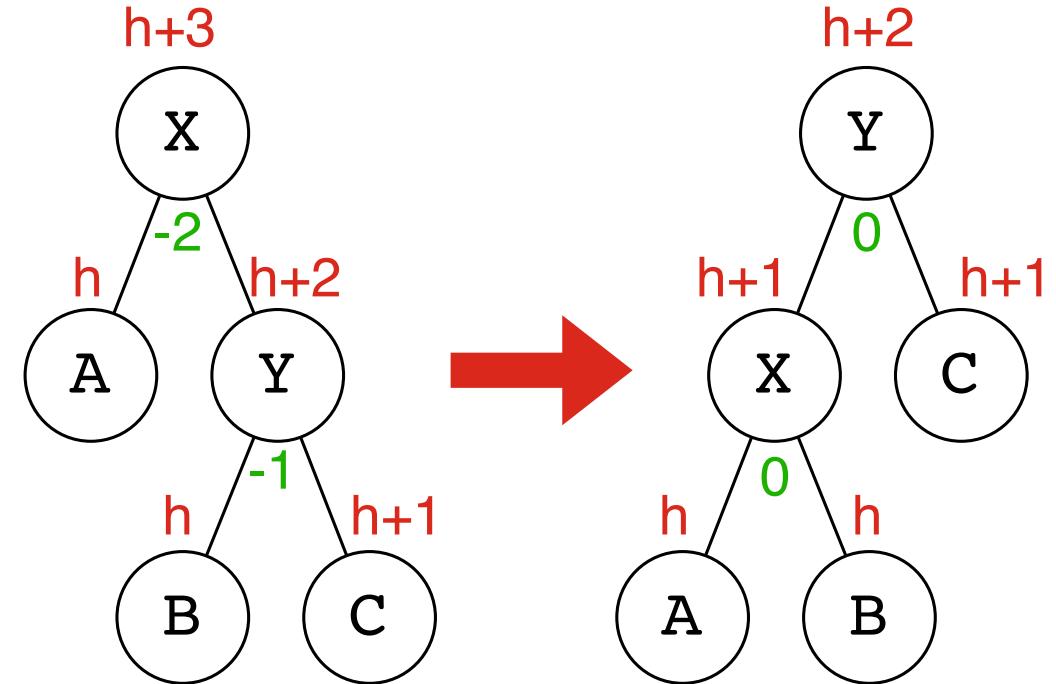




# Cas 1: équilibre(Y) = -1

- Après rotation ...
  - X et Y sont équilibrés
  - $h_X$  et  $h_Y$  changent

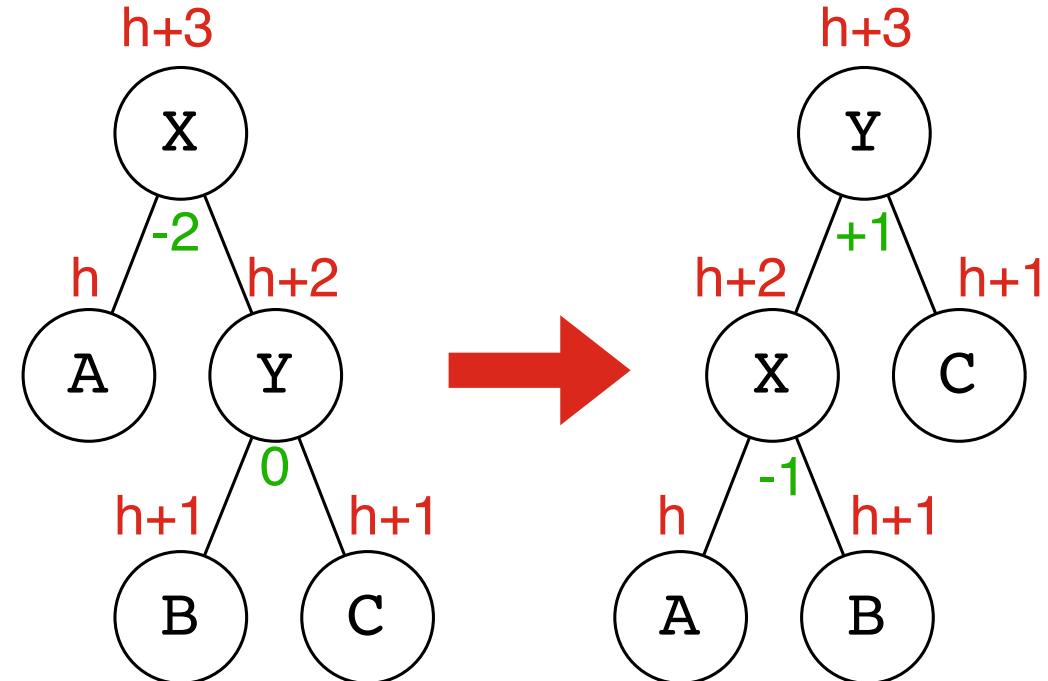
```
fonction rotation_gauche (ref r)
    t ← r.droit
    r.droit ← t.gauche
    t.gauche ← r
    r ← t
    calculer_hauteur(r.gauche)
    calculer_hauteur(r)
```





# Cas 2: équilibre(Y) = 0

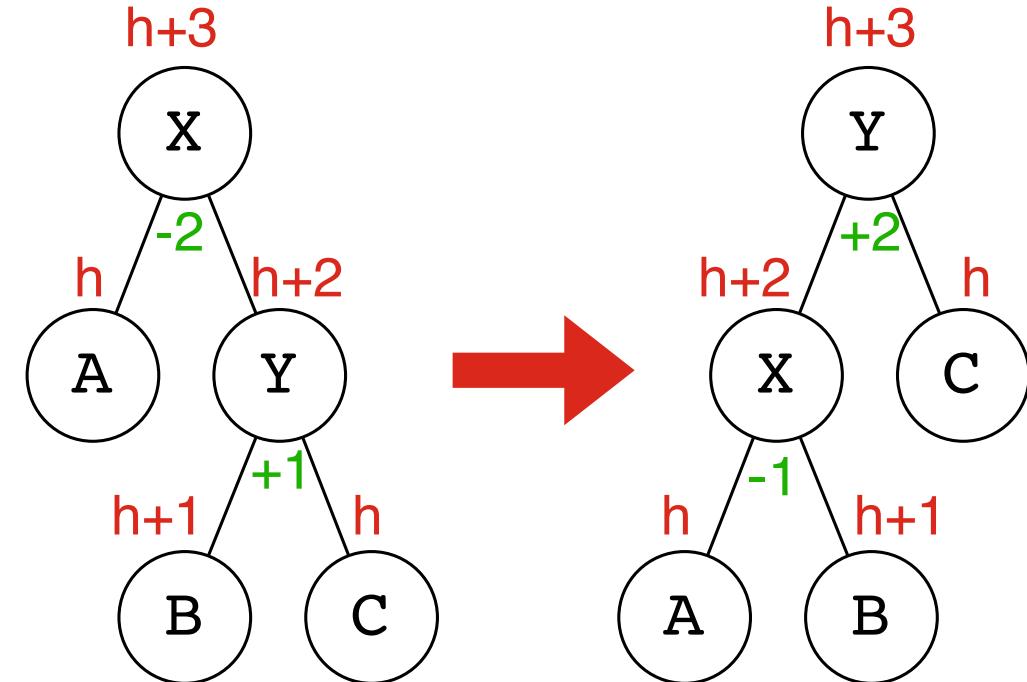
- Après rotation ...
  - Les équilibres de X et Y sont dans [-1,1]
  - La hauteur ne change pas. Pas besoin de vérifier plus haut dans l'arbre





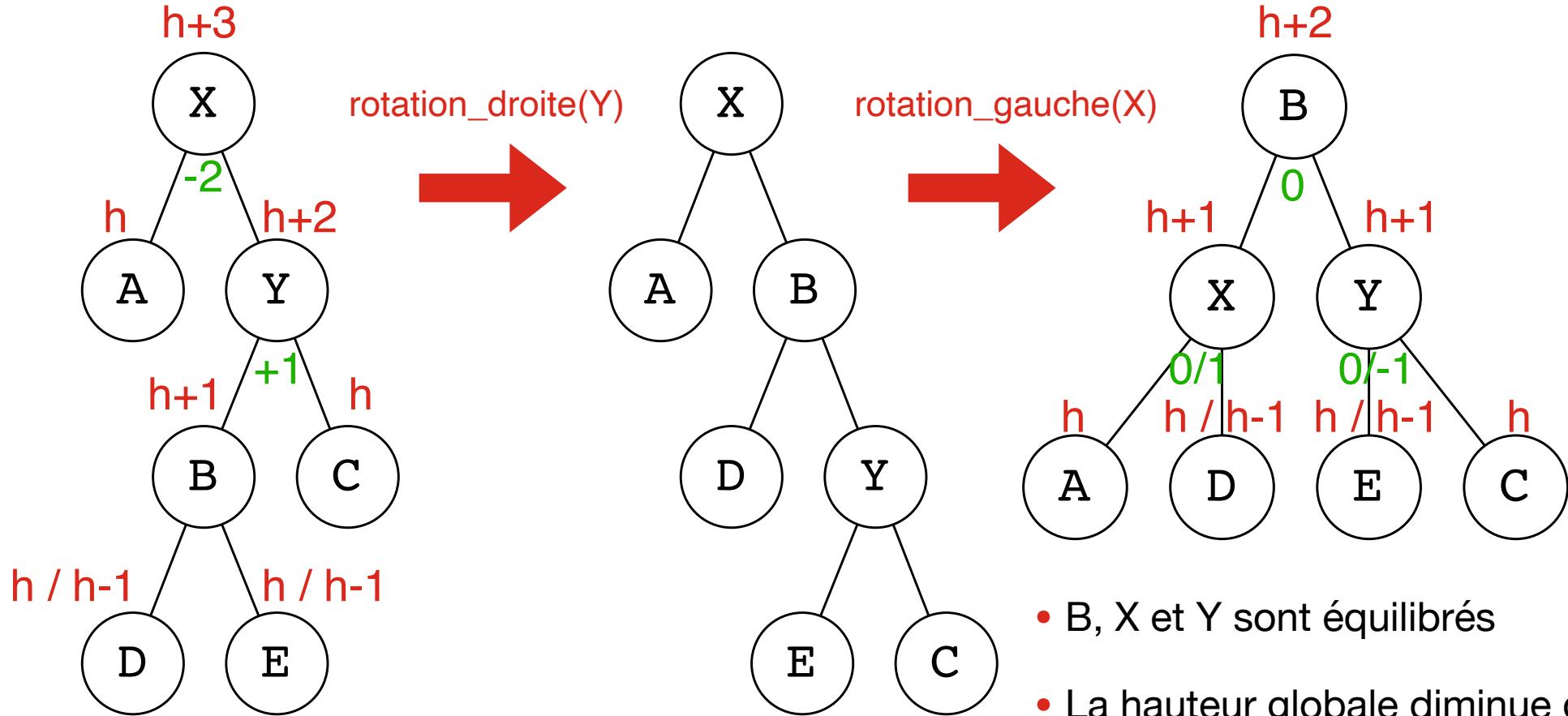
# Cas 3: équilibre(Y) = +1

- Après rotation ...
  - $\text{équilibre}(Y) = +2$
  - la rotation simple ne résout pas le problème
- Solution : une rotation **droite** de Y avant la rotation **gauche** de X, dans ce cas 3 uniquement





# Cas 3bis : double rotation





# En résumé, pour équilibrer

- Si le noeud  $r$  penche à droite ( $\text{équilibre}(r) = -2$ ),
  - si  $r.\text{droit}$  est équilibré ou penche à droite (cas 1 et 2), une rotation gauche suffit
  - si  $r.\text{droit}$  penche à gauche (cas 3), il faut effectuer une rotation droite de  $r.\text{droit}$  avant la rotation gauche de  $r$
- L'équilibrage conserve la hauteur de  $r$  si  $r.\text{droit}$  était équilibré, la diminue de 1 sinon
- Si le noeud  $r$  penche à gauche, idem par symétrie



# Algorithme d'équilibrage

```
fonction rétablir_équilibre (ref r)
    si r == Ø, retourner

    si équilibre(r) < -1,           // penche à droite
        si équilibre(r.droit) > 0,
            rotation_droite(r.droit)
            rotation_gauche(r)

        sinon, si équilibre(r) > 1,    // penche à gauche
            si équilibre(r.gauche) < 0,
                rotation_gauche(r.gauche)
                rotation_droite(r)

        sinon, calculer_hauteur(r)
```



# Avec pour rappel ...

```
fonction hauteur (r)
    si r == Ø, retourner 0
    sinon,      retourner r.hauteur
```

```
fonction calculer_hauteur (r)
    si r != Ø,
        r.hauteur ← 1 +
        max(hauteur(r.gauche), hauteur(r.droite))
```

```
fonction équilibre (r)
    si r == Ø, retourner 0
    sinon, retourner
        hauteur(r.gauche) - hauteur(r.droite)
```

```
fonction rotation_gauche (ref r)
    t ← r.droit
    r.droit ← t.gauche
    t.gauche ← r
    r ← t
    calculer_hauteur(r.gauche)
    calculer_hauteur(r)
```

```
fonction rotation_droite (ref r)
    t ← r.gauche
    r.gauche ← t.droit
    t.droit ← r
    r ← t
    calculer_hauteur(r.droit)
    calculer_hauteur(r)
```



# Insertion dans un arbre AVL

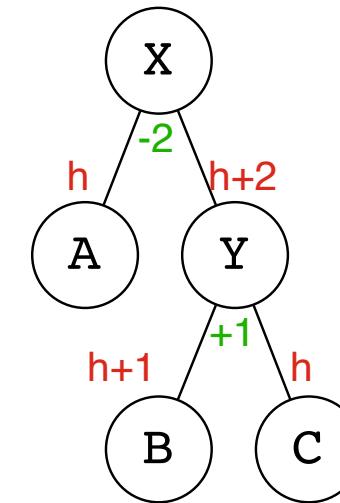
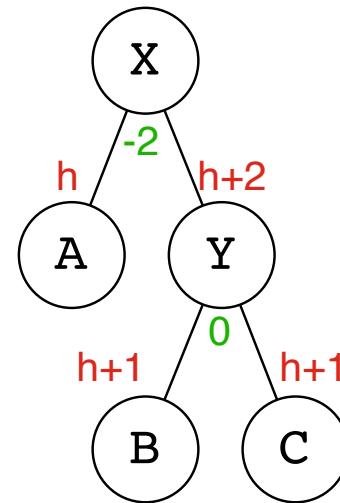
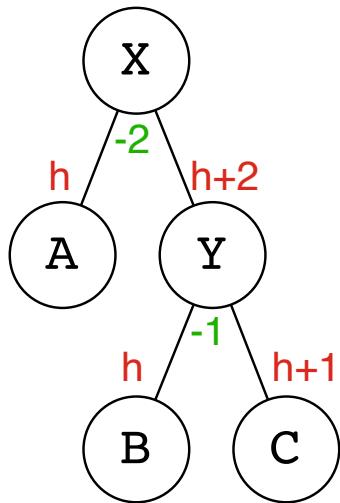
- Le plus simple :
  - code identique à celui de l'insertion dans un ABR
  - avec appel à `rétablir_équilibre(r)` après un appel récursif
- Plus efficace :
  - Utiliser le fait qu'**une seule rotation** sera nécessaire
  - Nécessite plus de communication entre noeud ou une approche itérative

```
fonction insérer (ref r, k)  
    si r est Ø  
        r ← nouveau noeud de clé k  
    sinon, si k == r.clé  
        k est déjà présent. L'action  
        dépend du TDA mis en oeuvre  
    sinon,  
        si k < r.clé  
            insérer(r.gauche, k)  
        sinon // k > r.clé  
            insérer(r.droite, k)  
            rétablir_équilibre(r)
```



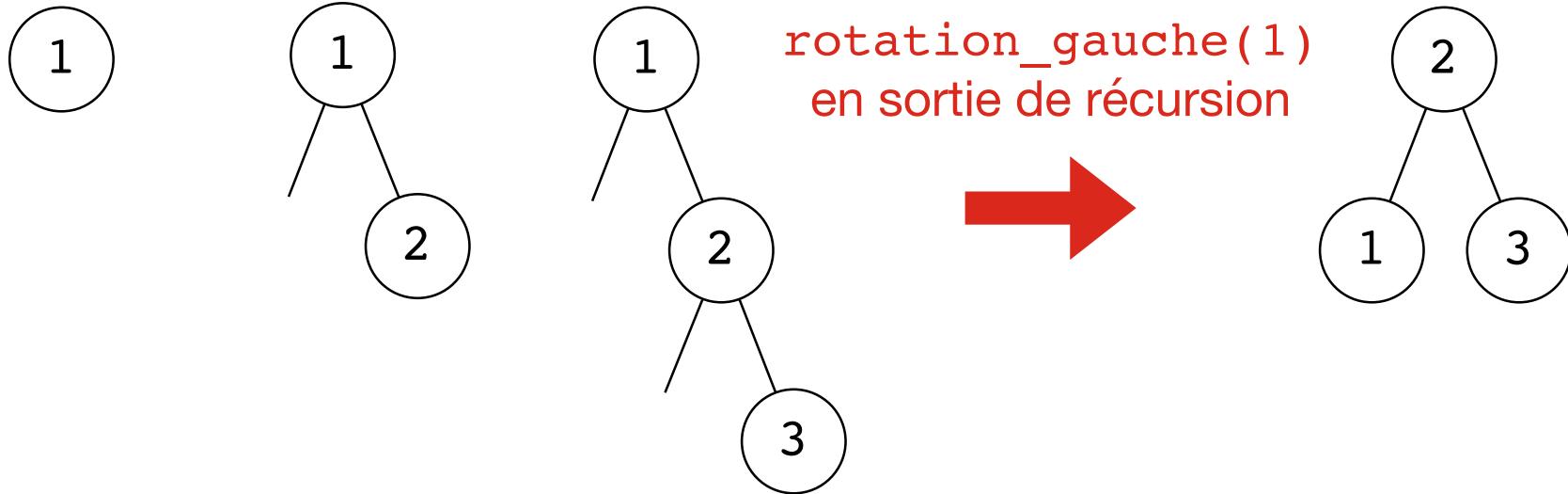
# Pourquoi une seule rotation ?

- Si le déséquilibre de X provient d'une insertion, elle a eu lieu dans les sous-arbres B ou C
- Seuls les cas 1 et 3 sont possibles. L'équilibrage de X diminue donc sa hauteur de 1, i.e. rétablit sa hauteur avant insertion



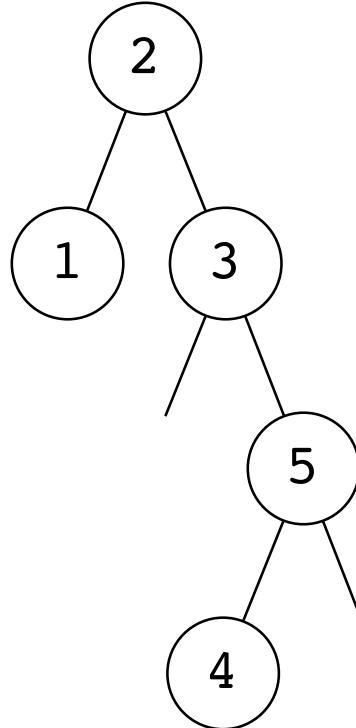
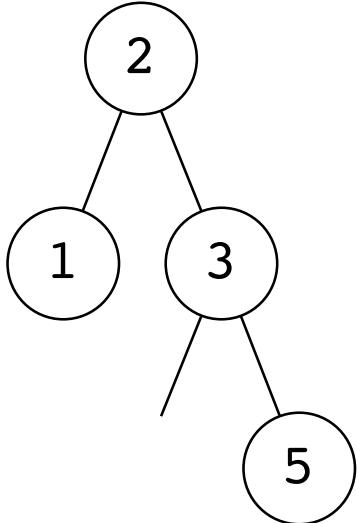


# Exemple : insérer 1, 2, 3

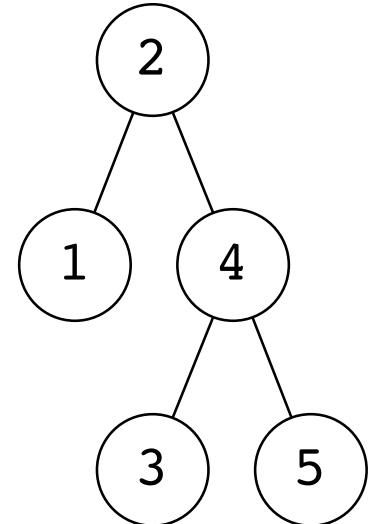
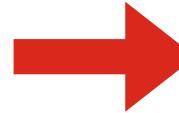




# Exemple : insérer 5, 4

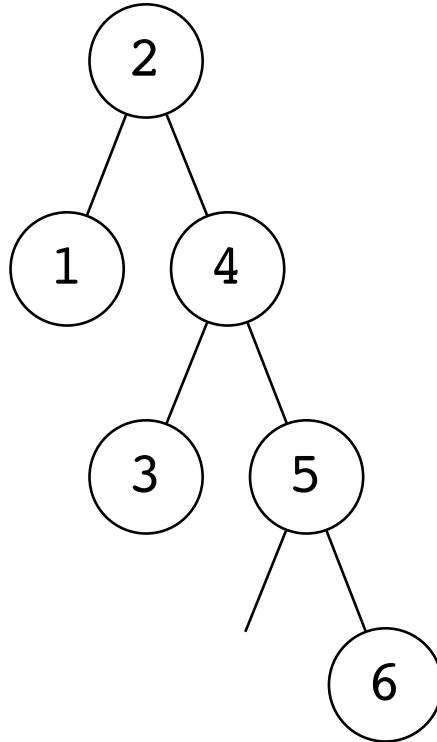


double rotation droite(5)  
puis gauche(3) pour ré-  
équilibrer 3 en sortie de  
récursion

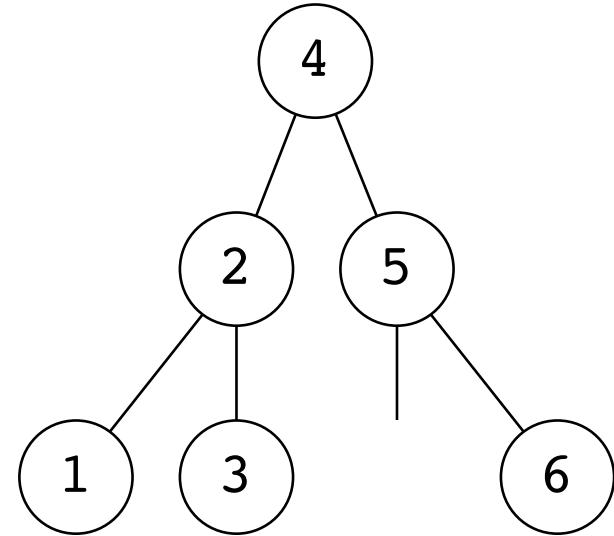
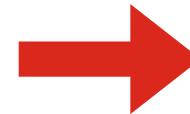




# Exemple : insérer 6

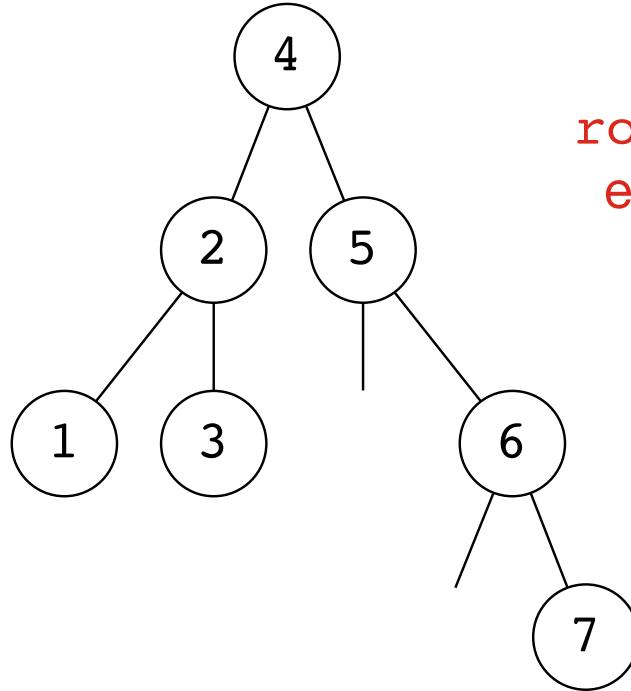


rotation\_gauche(2)  
en sortie de récursion

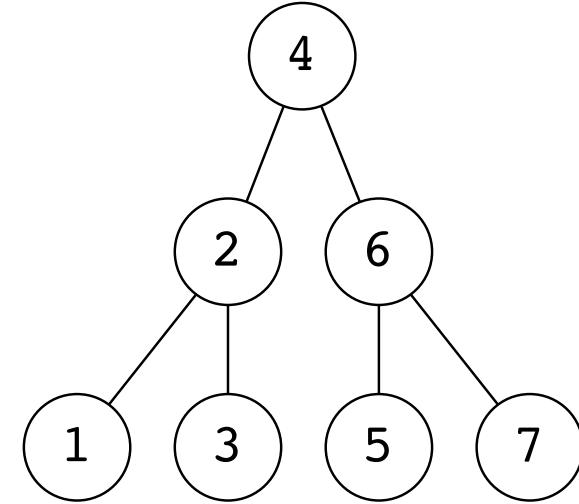
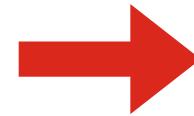




# Exemple : insérer 7



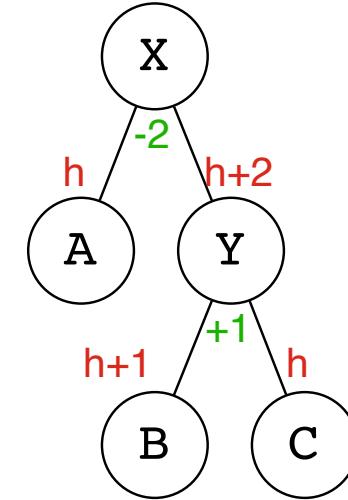
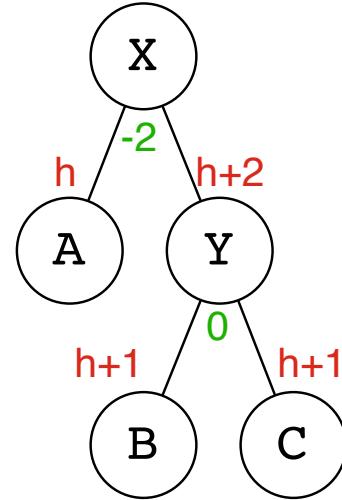
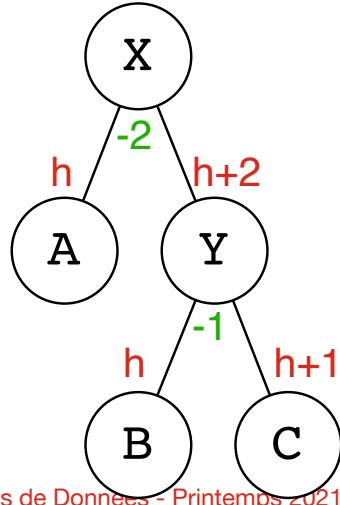
rotation\_gauche(5)  
en sortie de récursion





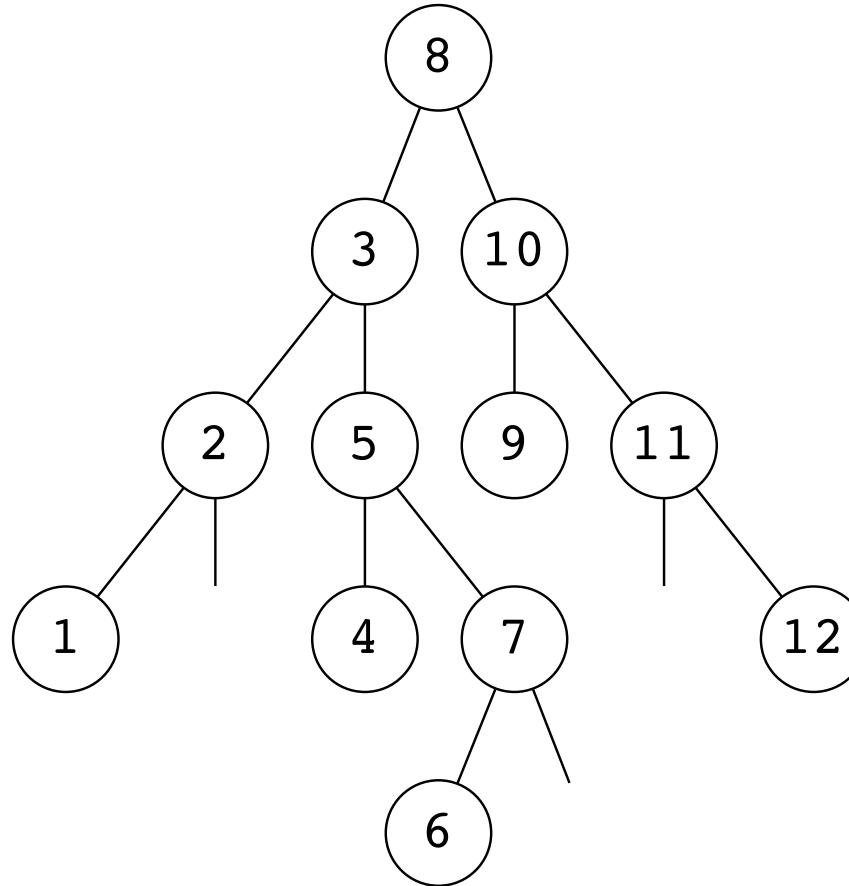
# Suppression d'un arbre AVL

- Comme pour l'insertion, il suffit d'ajouter un équilibrage en sortie de récursion
- Un déséquilibre ne survient que si l'élément a été supprimé du sous-arbre A.
- Dans les cas 1 et 3, la hauteur de X diminue de 1, ce qui propage éventuellement le déséquilibre à son parent



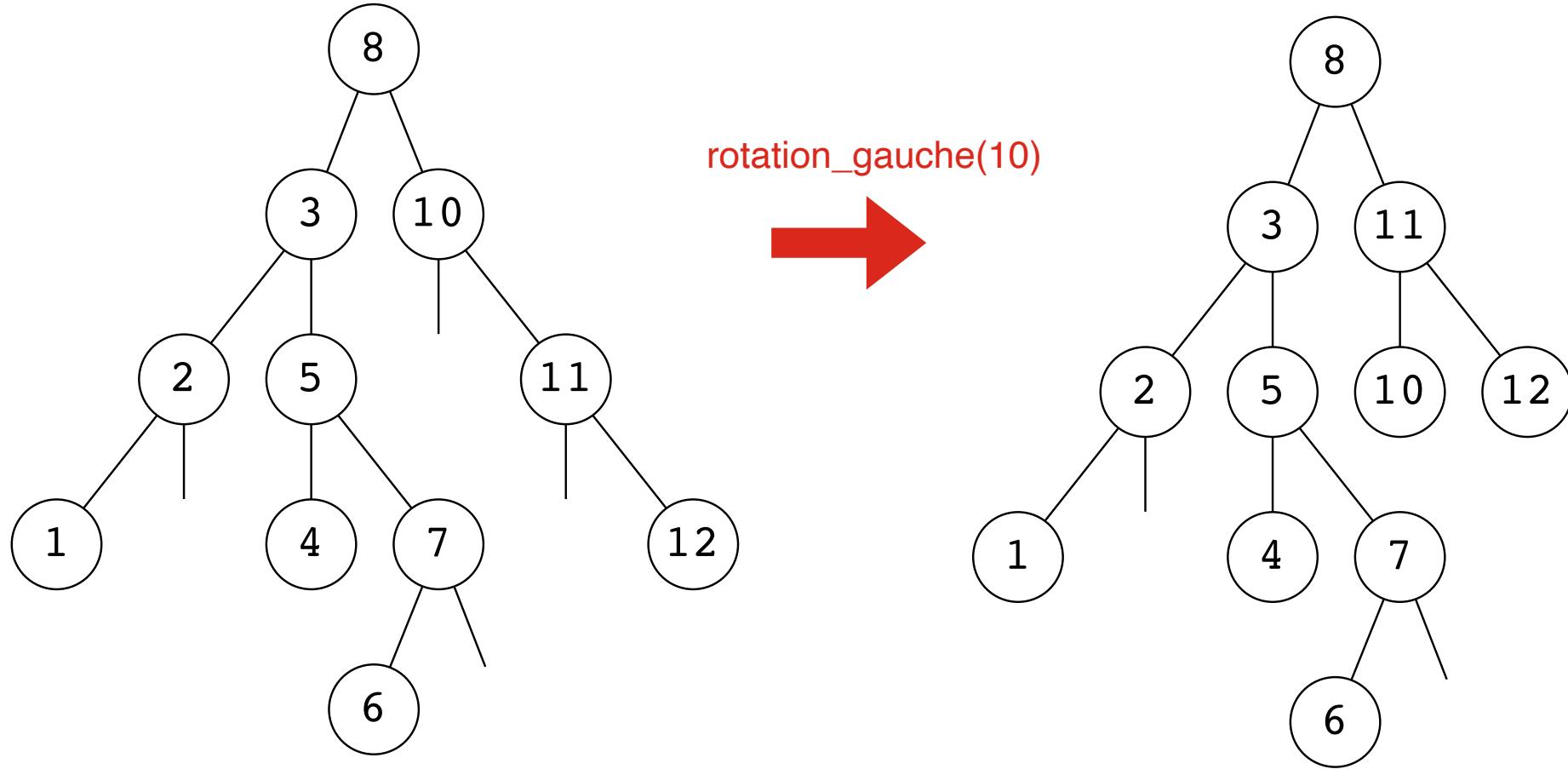


# Exemple : supprimer 9



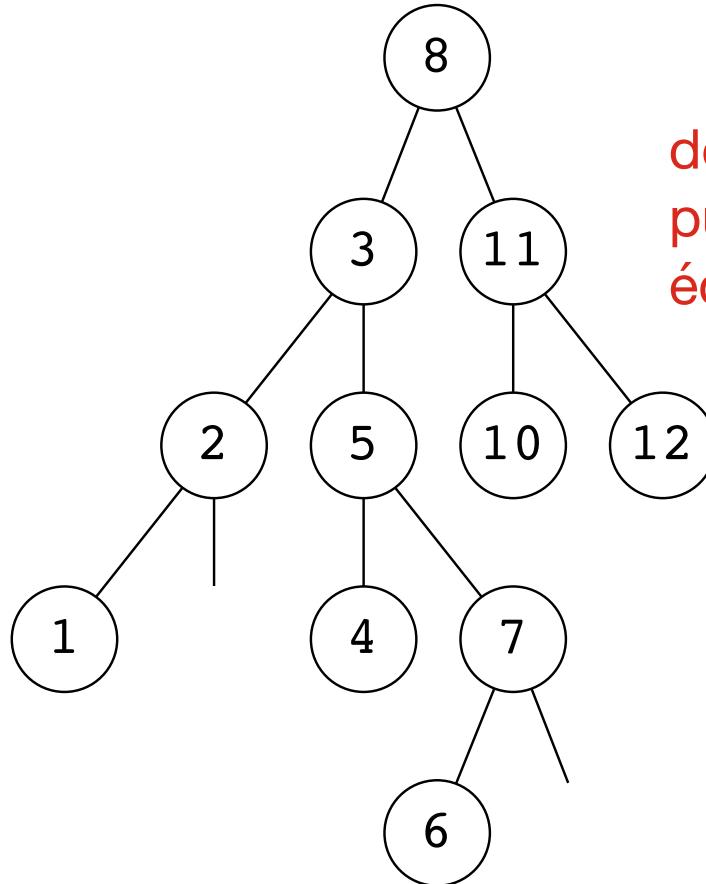


# Déséquilibre en 10

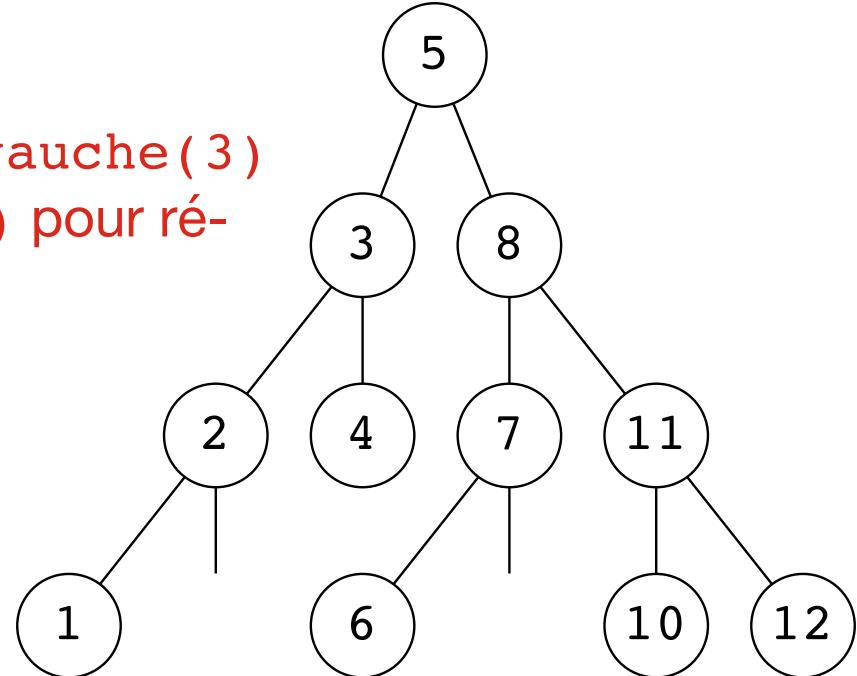




# Déséquilibre en 8



double rotation gauche(3)  
puis droite(8) pour ré-  
équilibrer 8





# Complexité

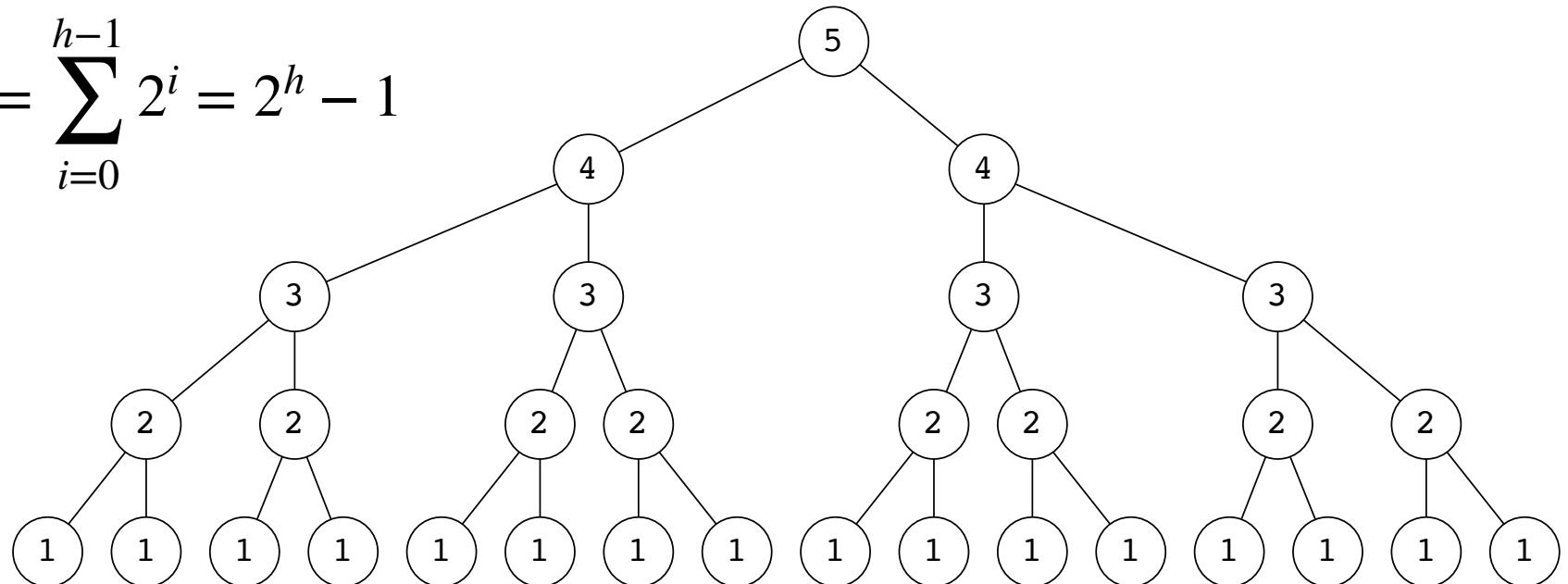
- Comme pour tout ABR, la complexité des opérations dépend de la profondeur du noeud cherché / inséré / supprimé
- Cette profondeur est bornée par la hauteur de l'arbre
- Etudions la relation  $n(h)$  entre cette hauteur  $h$  et le nombre  $n$  d'éléments stockés dans l'arbre
  - Dans le meilleur cas
  - Dans le pire cas



# Meilleur cas

- L'arbre binaire est complet, donc parfaitement équilibré en tous ses noeuds

$$n(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

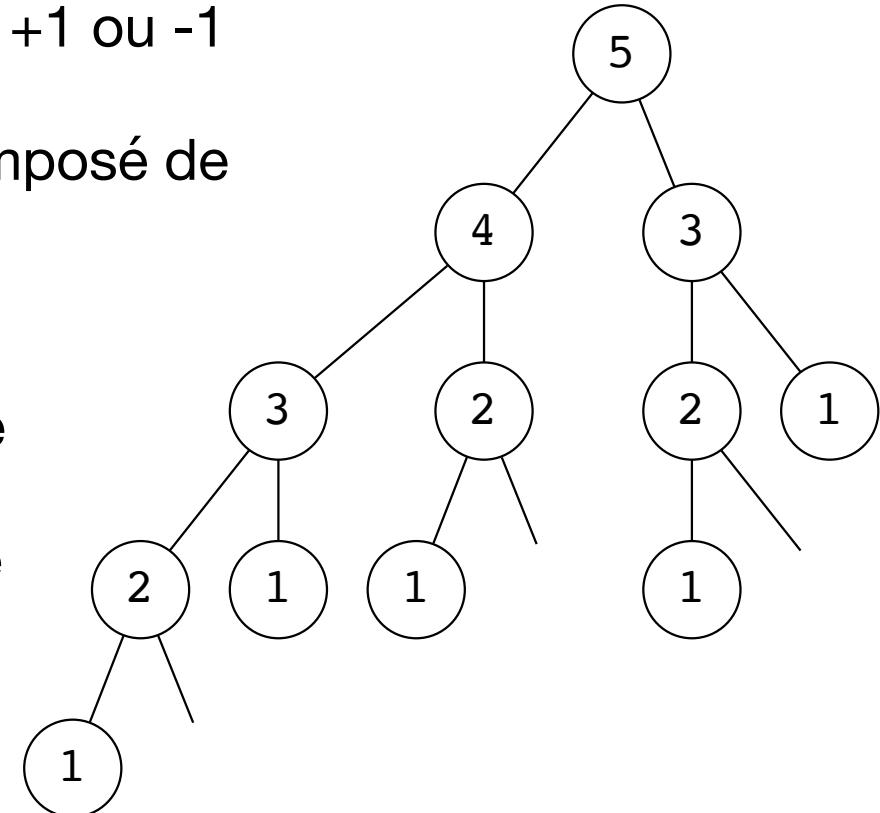


On utilise ici les hauteurs comme label des noeuds



# Pire cas

- Tous les noeuds ont un déséquilibre de +1 ou -1
- Le pire arbre de hauteur  $h$  est donc composé de
  - Sa racine
  - Le pire arbre de hauteur  $h-1$  d'un côté
  - Le pire arbre de hauteur  $h-2$  de l'autre



On utilise ici les hauteurs comme label des noeuds



# Pire cas (2)

- Le nombre  $n(h)$  d'éléments du pire arbre AVL de hauteur  $h$  se calcule donc en résolvant la récurrence

$$n(0) = 0 \quad n(1) = 1$$

$$n(h) = 1 + n(h - 1) + n(h - 2)$$

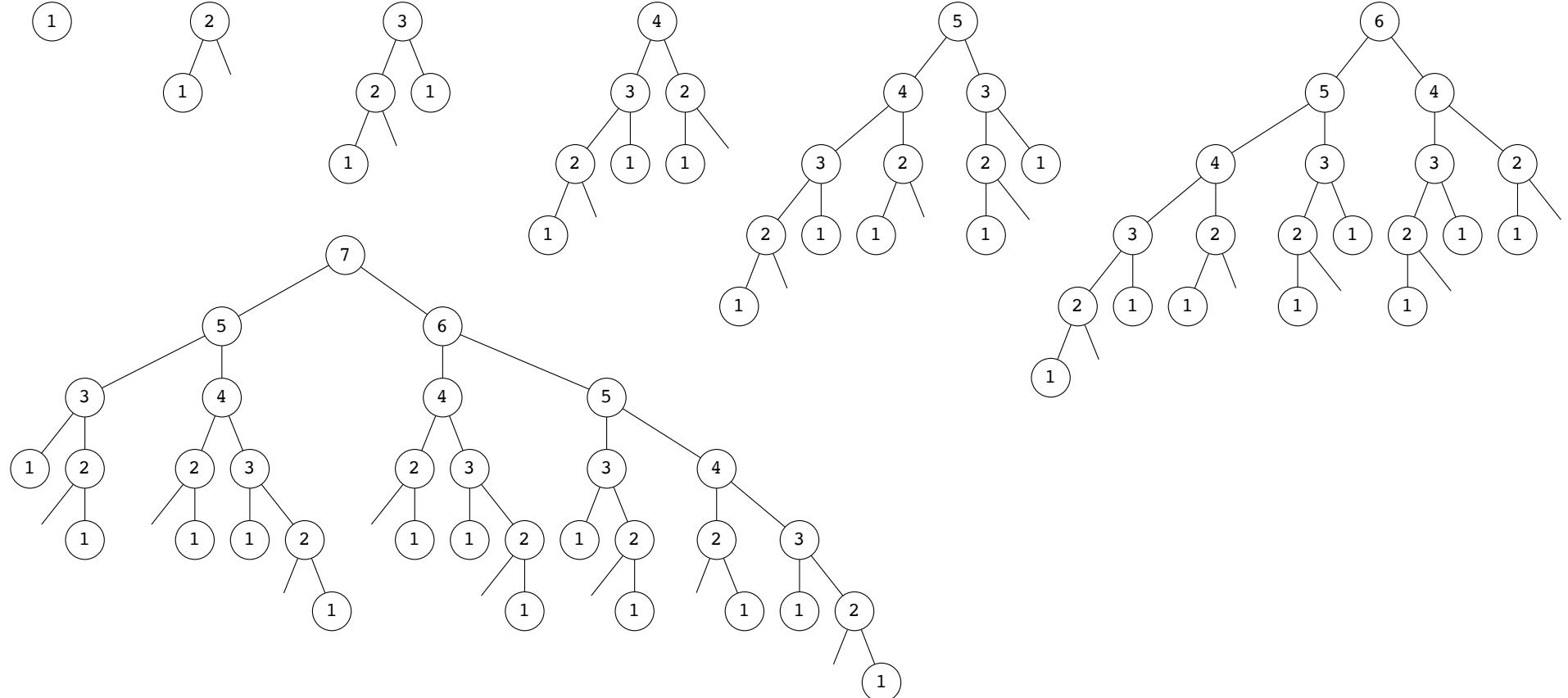
- Les premières valeurs de cette suite sont 0, 1, 2, 4, 7, 12, 20, 33, 54, 88, ... i.e. la suite des nombres de Fibonacci moins 1

- $n(h)$  croit donc comme  $\phi^h$  avec  $\phi = \frac{1 + \sqrt{5}}{2}$

- Réciiproquement,  $h(n) \approx 1.4404 \cdot \log_2(n)$



# Les pires arbres AVL...



# 16. TDA Ensemble





# TDA Ensemble

- TDA qui stocke les données sans répétition.



# TDA Ensemble

- TDA qui stocke les données sans répétition.
- Opérations
  - Insérer un élément
  - Chercher si un élément est présent
  - Supprimer un élément
  - Parcourir, itérer, ...



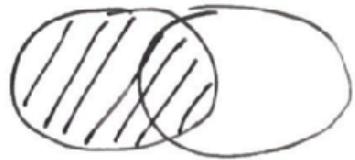
# TDA Ensemble

- TDA qui stocke les données sans répétition.
- Opérations
  - Insérer un élément
  - Chercher si un élément est présent
  - Supprimer un élément
  - Parcourir, itérer, ...
- Un ensemble est dit ordonné si on peut le parcourir dans l'ordre. C'est le cas quand il est mis en oeuvre avec un **arbre binaire de recherche**.

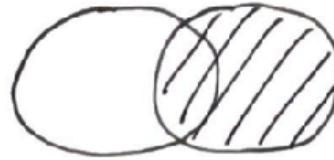
# Opérations sur les ensembles



E1



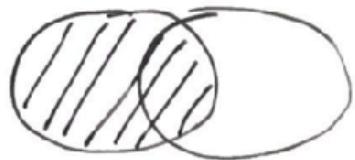
E2



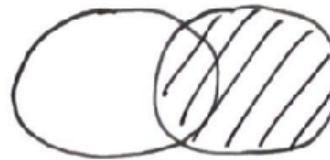


# Opérations sur les ensembles

E1

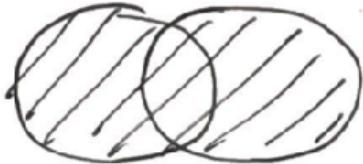


E2



$$A \cup B = \{x: x \in A \vee x \in B\}$$

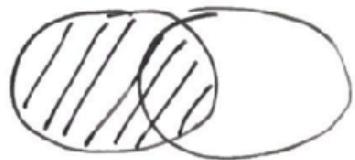
E1 ∪ E2



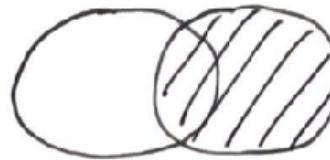


# Opérations sur les ensembles

E1

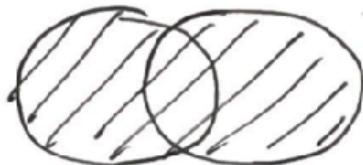


E2

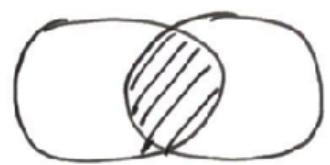


$$A \cup B = \{x: x \in A \vee x \in B\}$$

E1 ∪ E2



E1 ∩ E2

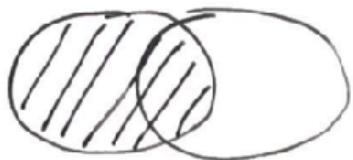


$$A \cap B = \{x: x \in A \wedge x \in B\}$$

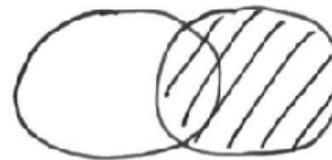


# Opérations sur les ensembles

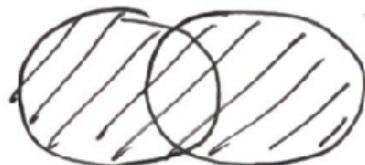
E1



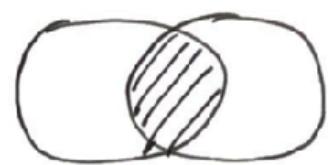
E2



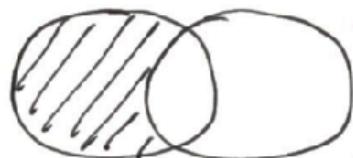
E1 ∪ E2



E1 ∩ E2



E1 \ E2



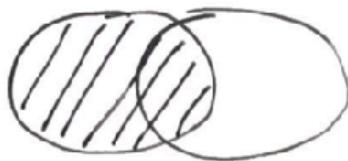
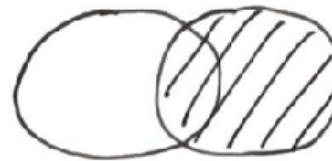
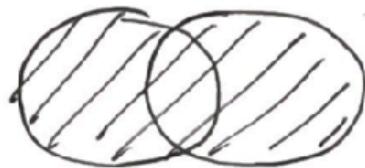
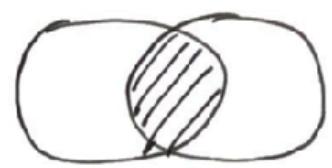
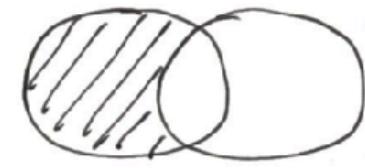
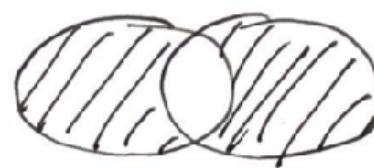
$$A \cup B = \{x: x \in A \vee x \in B\}$$

$$A \cap B = \{x: x \in A \wedge x \in B\}$$

$$A \setminus B = \{x: x \in A \wedge x \notin B\}$$



# Opérations sur les ensembles

 $E_1$  $E_2$  $E_1 \cup E_2$  $E_1 \cap E_2$  $E_1 \setminus E_2$  $E_1 \Delta E_2$ 

$$A \cup B = \{x: x \in A \vee x \in B\}$$

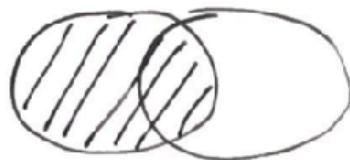
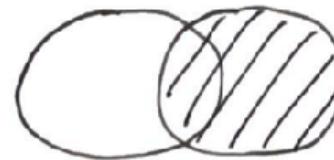
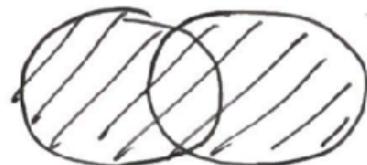
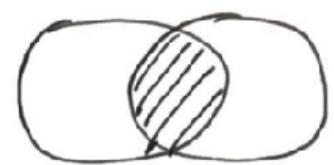
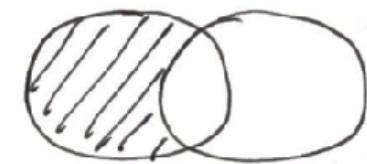
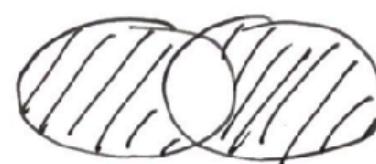
$$A \cap B = \{x: x \in A \wedge x \in B\}$$

$$A \setminus B = \{x: x \in A \wedge x \notin B\}$$

$$A \Delta B = (A \cup B) \setminus (A \cap B)$$



# Opérations sur les ensembles

 $E_1$  $E_2$  $E_1 \cup E_2$  $E_1 \cap E_2$  $E_1 \setminus E_2$  $E_1 \Delta E_2$ 

$$A \cup B = \{x: x \in A \vee x \in B\}$$

$$A \cap B = \{x: x \in A \wedge x \in B\}$$

$$A \setminus B = \{x: x \in A \wedge x \notin B\}$$

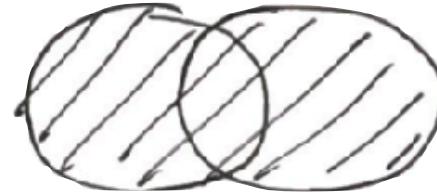
$$A \Delta B = (A \cup B) \setminus (A \cap B)$$

$$A \subseteq B \text{ si } \forall x \in A, x \in B$$



# Union

E1 U E2

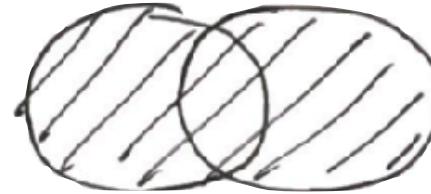


- Algorithme naïf
  - Copier un ensemble
  - Insérer les éléments de l'autre dans l'ensemble copié



# Union

$E_1 \cup E_2$



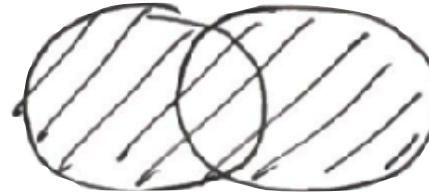
- Algorithme naïf
- Copier un ensemble
- Insérer les éléments de l'autre dans l'ensemble copié

```
fonction union (A, B)
    U  $\leftarrow$  copier(A)
    b  $\leftarrow$  début(B)
    tant que b != fin(B),
        insérer(U, b.clé)
        b  $\leftarrow$  suivant(b)
    retourner U
```



# Union

$E_1 \cup E_2$



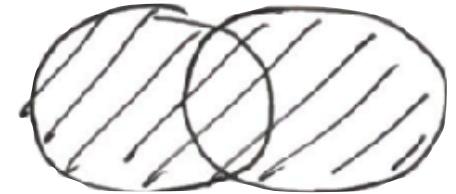
- Algorithme naïf
  - Copier un ensemble
  - Insérer les éléments de l'autre dans l'ensemble copié
- Complexité  $O(n+m \cdot \log(n+m))$  si les ensembles ont  $n$  et  $m$  éléments

```
fonction union (A, B)
    U  $\leftarrow$  copier(A)
    b  $\leftarrow$  début(B)
    tant que b != fin(B),
        insérer(U, b.clé)
        b  $\leftarrow$  suivant(b)
    retourner U
```

# Union



E<sub>1</sub> ∪ E<sub>2</sub>

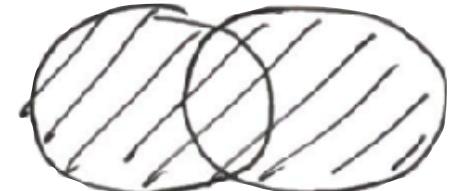


- Algorithme en  $O(n+m)$ , similaire à l'algorithme de fusion du tri fusion



# Union

E<sub>1</sub> ∪ E<sub>2</sub>

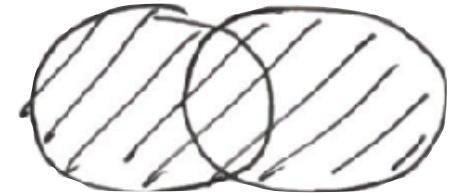


- Algorithme en  $O(n+m)$ , similaire à l'algorithme de fusion du tri fusion
- Itérer sur les deux arbres simultanément par ordre croissant



# Union

E1 U E2

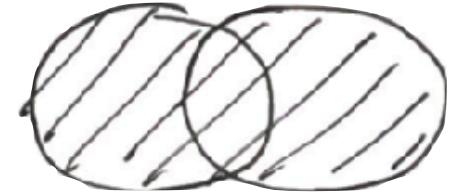


- Algorithme en  $O(n+m)$ , similaire à l'algorithme de fusion du tri fusion
- Itérer sur les deux arbres simultanément par ordre croissant
- N'avancer que dans l'arbre ayant l'élément le plus petit



# Union

E1 U E2

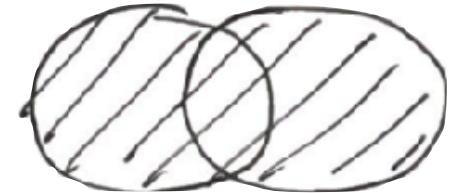


- Algorithme en  $O(n+m)$ , similaire à l'algorithme de fusion du tri fusion
- Itérer sur les deux arbres simultanément par ordre croissant
- N'avancer que dans l'arbre ayant l'élément le plus petit
- Insérer en tête de l'arbre union, pour créer un arbre dégénéré sans enfant droit



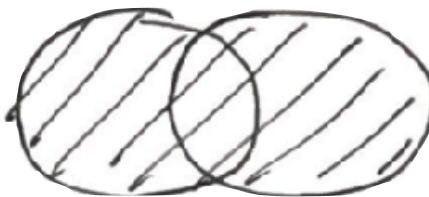
# Union

E1 U E2



- Algorithme en  $O(n+m)$ , similaire à l'algorithme de fusion du tri fusion
- Itérer sur les deux arbres simultanément par ordre croissant
- N'avancer que dans l'arbre ayant l'élément le plus petit
- Insérer en tête de l'arbre union, pour créer un arbre dégénéré sans enfant droit
- Arboriser le résultat final

$E_1 \cup E_2$



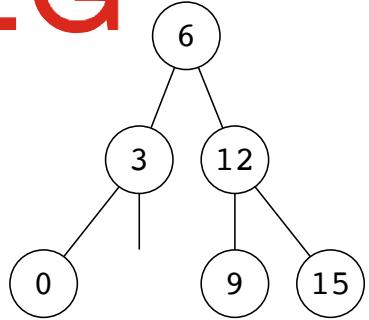
```

fonction union (A, B)
    U ← Ø
    a ← début(A), b ← début(B)
    tant que a != fin(A) et b != fin(B),
        si a.clé < b.clé,
            k ← a.clé, a ← suivant(a)
        sinon, si a.clé > b.clé,
            k ← b.clé, b ← suivant(b)
        sinon,
            k ← a.clé, a ← suivant(a), b ← suivant(b)
    n ← nouveau noeud de clé k
    n.gauche ← U, U ← n
    tant que a != fin(A),
        n ← nouveau noeud de clé a.clé
        n.gauche ← U, U ← n, a ← suivant(a)
    tant que b != fin(B),
        n ← nouveau noeud de clé b.clé
        n.gauche ← U, U ← n, b ← suivant(b)
    retourner U

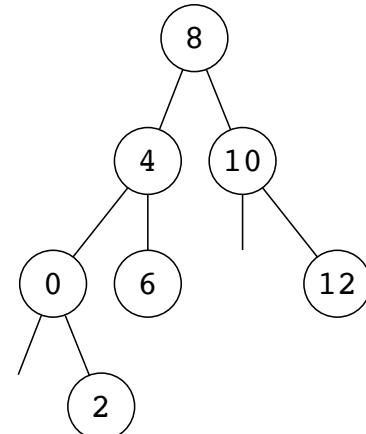
```



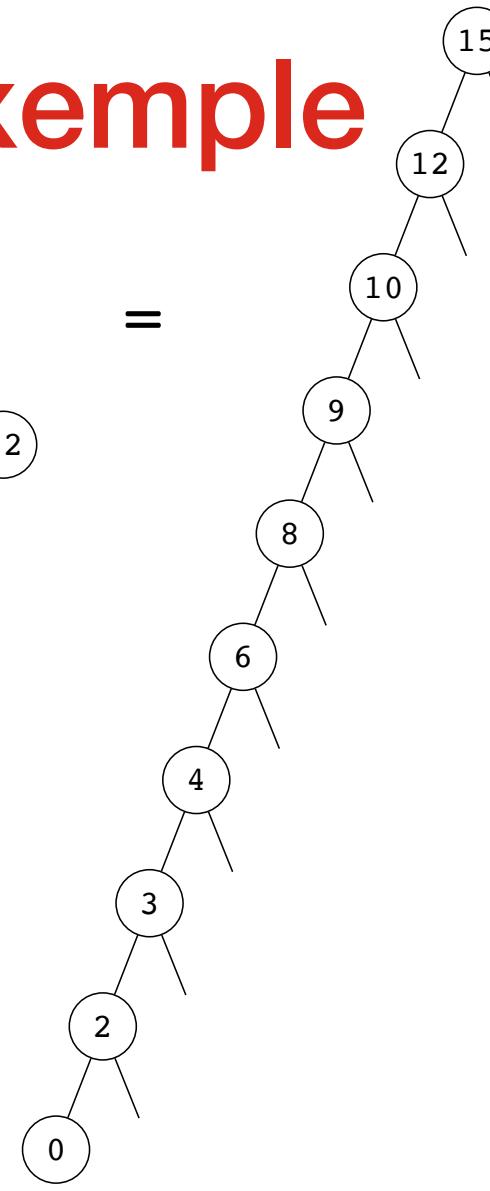
# Exemple



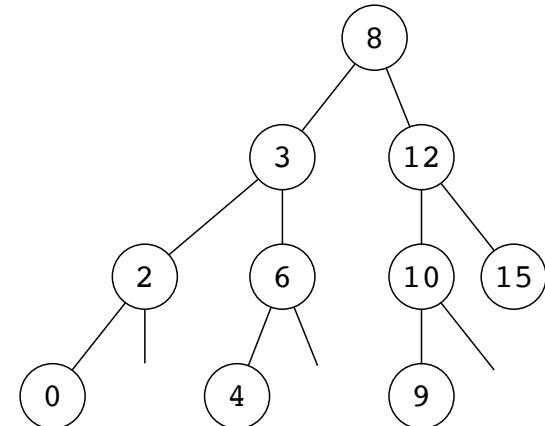
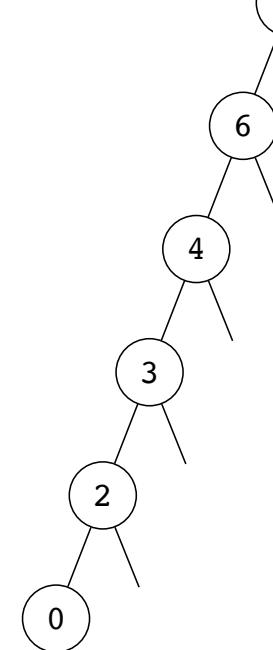
U



=

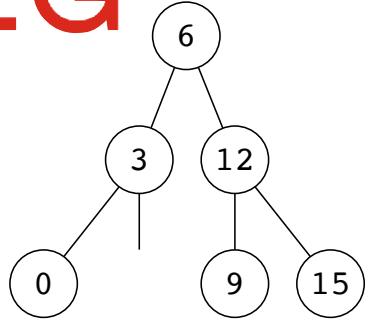


$E_1 \cup E_2$

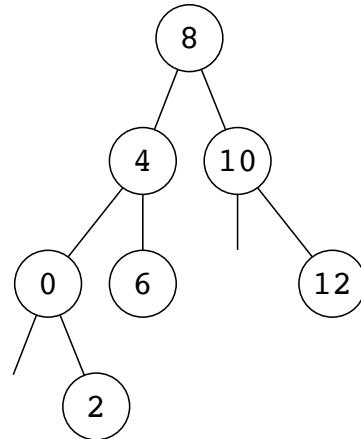




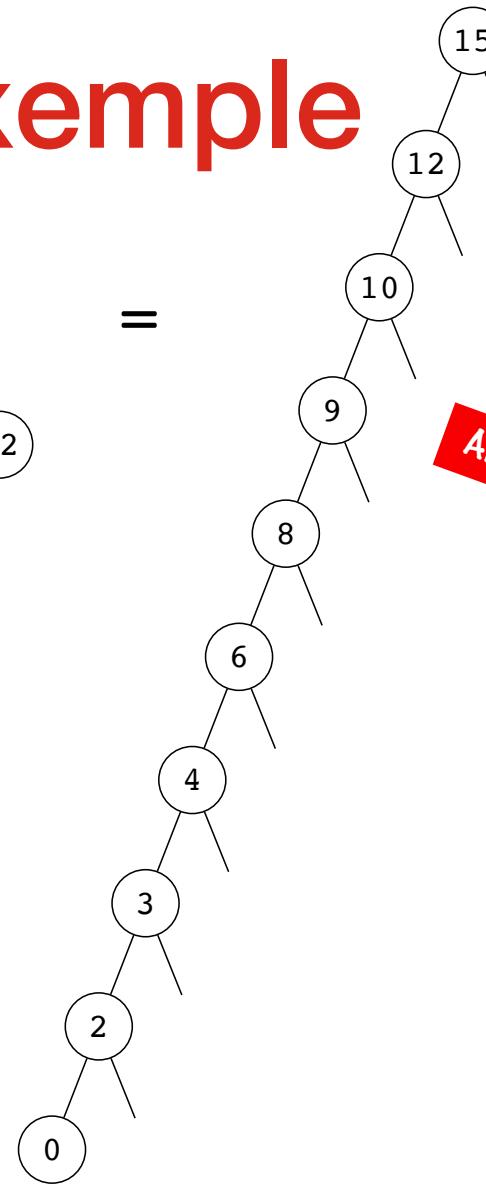
# Exemple



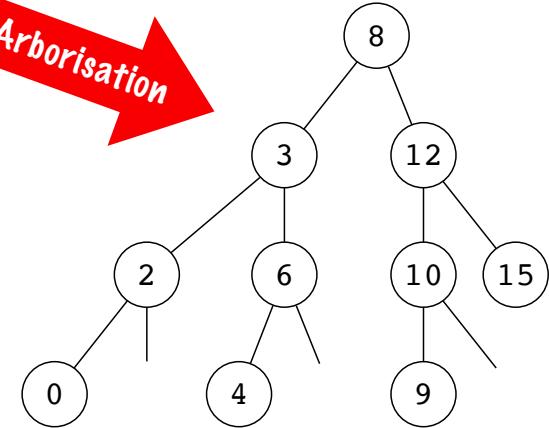
U



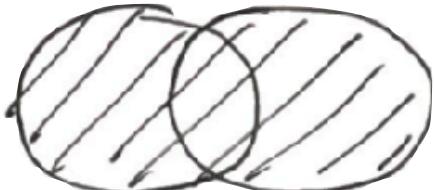
=



Arborisation



$E_1 \cup E_2$

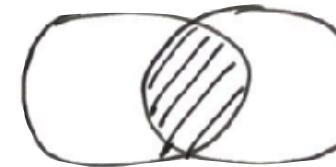


# Intersection

- Algorithme similaire en  $O(m+n)$
- Itération simultanée sur les deux arbres par ordre croissant
- N'insérer que les clés présentent dans les deux arbres
- Arboriser le résultat final

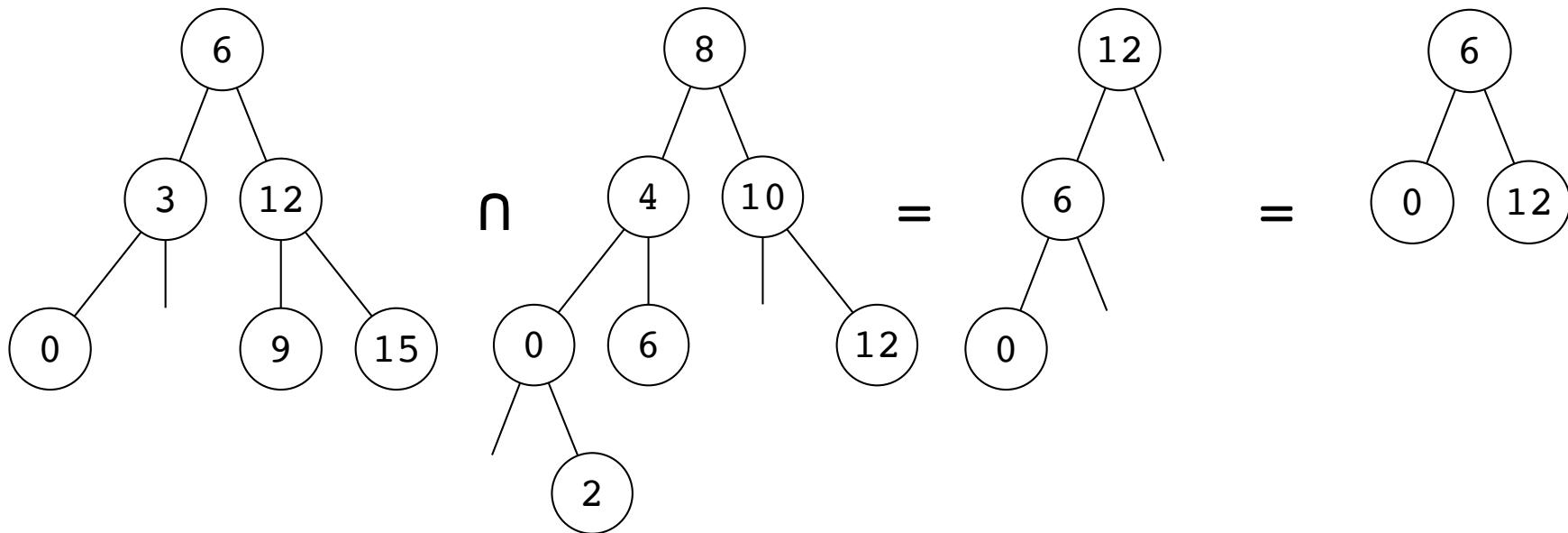
```
fonction intersection (A, B)
    I ← Ø
    a ← début(A), b ← début(B)
    tant que a != fin(A) et b != fin(B),
        si a.clé < b.clé,
            a ← suivant(a)
        sinon, si a.clé > b.clé,
            b ← suivant(b)
        sinon,
            n ← nouveau noeud de clé a.clé
            n.gauche ← I, I ← n
            a ← suivant(a), b ← suivant(b)
    retourner I
```

$$E_1 \cap E_2$$





# Exemple



# 17. TDA Tableau Associatif





# TDA Tableau associatif

- Aussi appelé dictionnaire

# TDA Tableau associatif



- Aussi appelé dictionnaire
- Associe des clés uniques et des valeurs

# TDA Tableau associatif



- Aussi appelé dictionnaire
- Associe des clés uniques et des valeurs
- Généralisation du TDA tableau avec des indices de type arbitraire



# TDA Tableau associatif

- Aussi appelé dictionnaire
- Associe des clés uniques et des valeurs
- Généralisation du TDA tableau avec des indices de type arbitraire
- Permet d'écrire

```
age( "Chiara" ) = 7;
```



# TDA Tableau associatif

- Aussi appelé dictionnaire
- Associe des clés uniques et des valeurs
- Généralisation du TDA tableau avec des indices de type arbitraire
- Permet d'écrire

```
age( "Chiara" ) = 7;
```

- Expression correcte en C++ si age est de type

```
std::map < std::string, unsigned >
```



# Opérations

- Ajouter - associer nouvelle clé à une valeur



# Opérations

- Ajouter - associer nouvelle clé à une valeur
- Modifier - associer nouvelle valeur à une clé existante



# Opérations

- Ajouter - associer nouvelle clé à une valeur
- Modifier - associer nouvelle valeur à une clé existante
- Supprimer - une clé et la valeur associée



# Opérations

- Ajouter - associer nouvelle clé à une valeur
- Modifier - associer nouvelle valeur à une clé existante
- Supprimer - une clé et la valeur associée
- Chercher - si une clé existe et quelle est la valeur associée



# Opérations

- Ajouter - associer nouvelle clé à une valeur
- Modifier - associer nouvelle valeur à une clé existante
- Supprimer - une clé et la valeur associée
- Chercher - si une clé existe et quelle est la valeur associée
- Parcourir, itérer, ...



# Mise en oeuvre

- Arbre binaire de recherche dont les noeuds stockent clé et valeur
- Tableau associatif trié
- Complexités en  $O(\log(n))$



# Mise en oeuvre

- Arbre binaire de recherche dont les noeuds stockent clé et valeur
- Tableau associatif trié
- Complexités en  $O(\log(n))$
- Tables de hachage (ASD2)
- Non trié
- Complexité moyenne en  $O(1)$ , mais  $O(n)$  au pire

# 18. std::set





# Conteneurs et algorithmes

## Containers class templates

### Sequence containers:

<a href="#">array</a> <small>C++11</small>	Array class (class template )
<a href="#">vector</a>	Vector (class template )
<a href="#">deque</a>	Double ended queue (class template )
<a href="#">forward_list</a> <small>C++11</small>	Forward list (class template )
<a href="#">list</a>	List (class template )

### Container adaptors:

<a href="#">stack</a>	LIFO stack (class template )
<a href="#">queue</a>	FIFO queue (class template )
<a href="#">priority_queue</a>	Priority queue (class template )

### Associative containers:

<a href="#">set</a>	Set (class template )
<a href="#">multiset</a>	Multiple-key set (class template )
<a href="#">map</a>	Map (class template )
<a href="#">multimap</a>	Multiple-key map (class template )

### Unordered associative containers:

<a href="#">unordered_set</a> <small>C++11</small>	Unordered Set (class template )
<a href="#">unordered_multiset</a> <small>C++11</small>	Unordered Multiset (class template )
<a href="#">unordered_map</a> <small>C++11</small>	Unordered Map (class template )
<a href="#">unordered_multimap</a> <small>C++11</small>	Unordered Multimap (class template )



Container adaptors:

stack

LIFO stack (class template )

queue

FIFO queue (class template )

priority\_queue

Priority queue (class template )

## Associative containers:

set

Set (class template )

multiset

Multiple-key set (class template )

map

Map (class template )

multimap

Multiple-key map (class template )

## Unordered associative containers:

unordered\_set C++11

Unordered Set (class template )

unordered\_multiset C++11

Unordered Multiset (class template )

unordered\_map C++11

Unordered Map (class template )

# std::set<T, Compare>



- TDA ensemble trié
  - Éléments non modifiables, seulement insertion, recherche et suppression

# std::set<T, Compare>



- TDA ensemble trié
  - Éléments non modifiables, seulement insertion, recherche et suppression
  - Arbre rouge-noir, une sorte d'arbre binaire de recherche équilibré



# std::set<T, Compare>

- TDA ensemble trié
  - Éléments non modifiables, seulement insertion, recherche et suppression
- Arbre rouge-noir, une sorte d'arbre binaire de recherche équilibré
- 3 pointeurs par noeud (parent, gauche, droite) pour permettre une itération efficace



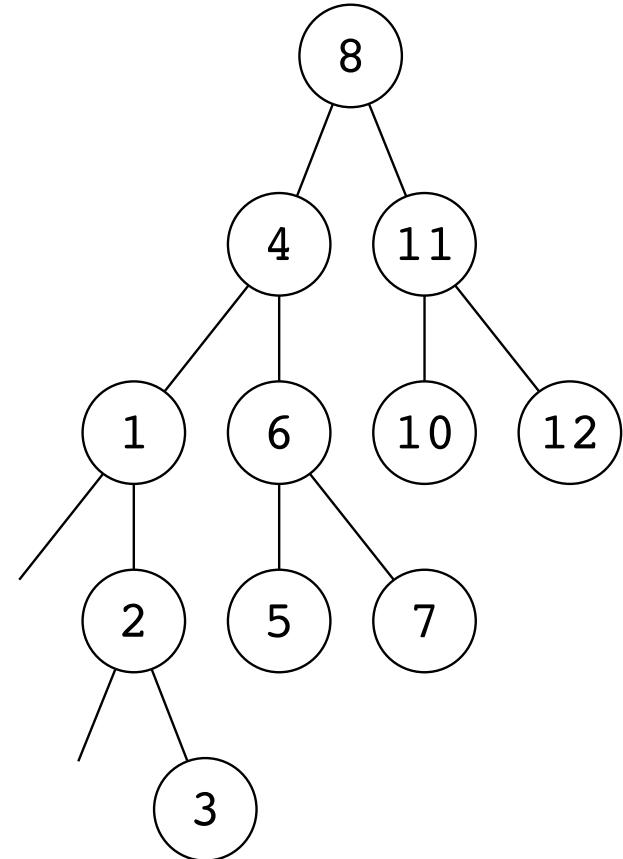
# std::set<T, Compare>

- TDA ensemble trié
  - Éléments non modifiables, seulement insertion, recherche et suppression
  - Arbre rouge-noir, une sorte d'arbre binaire de recherche équilibré
  - 3 pointeurs par noeud (parent, gauche, droite) pour permettre une itération efficace
  - Ne stocke pas la taille dans les noeuds, donc n'offre pas le TDA order statistics tree



# Noeud final

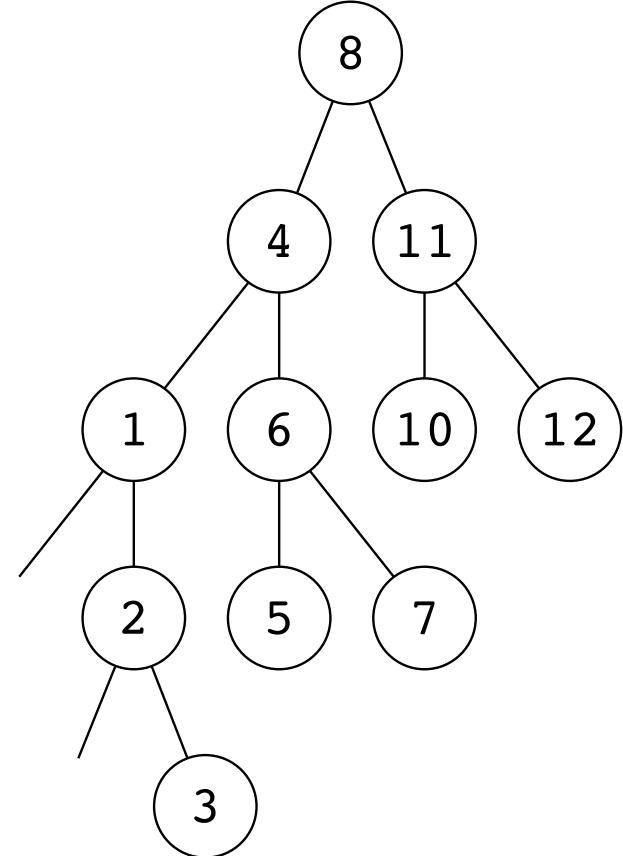
- Pour simplifier la mise en oeuvre des itérateurs





# Noeud final

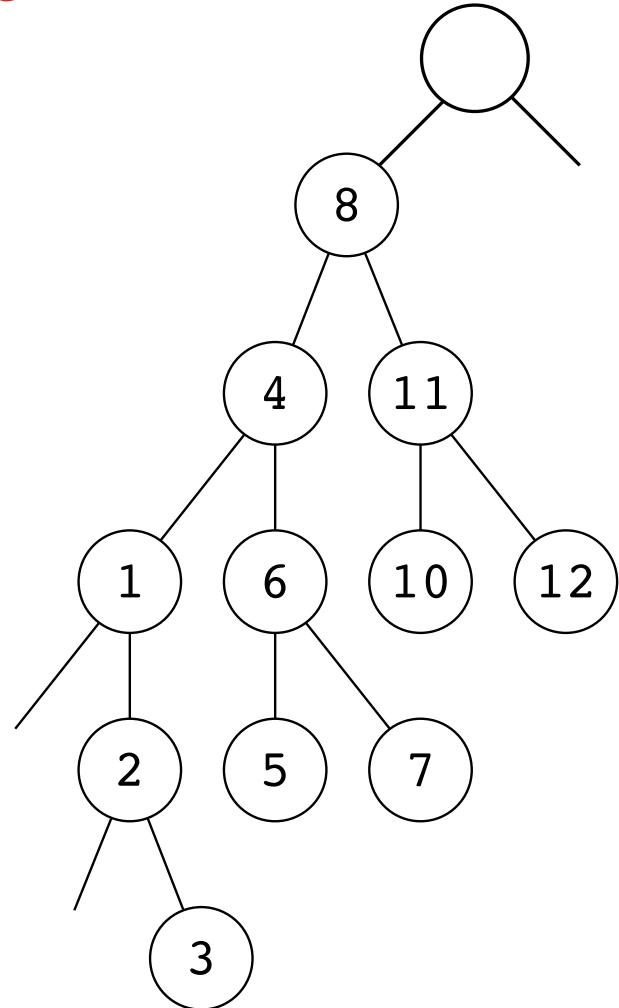
- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide





# Noeud final

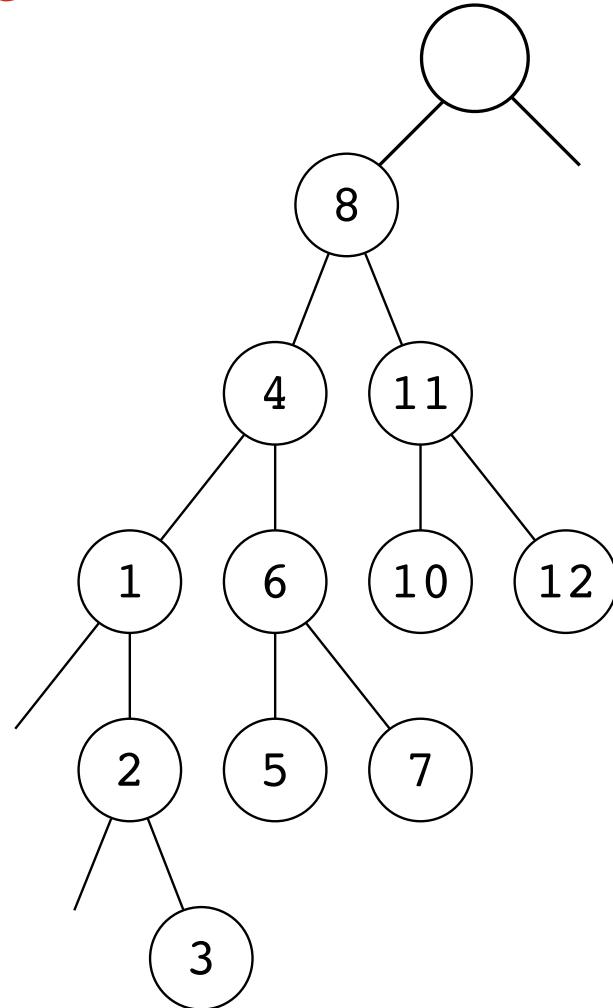
- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide





# Noeud final

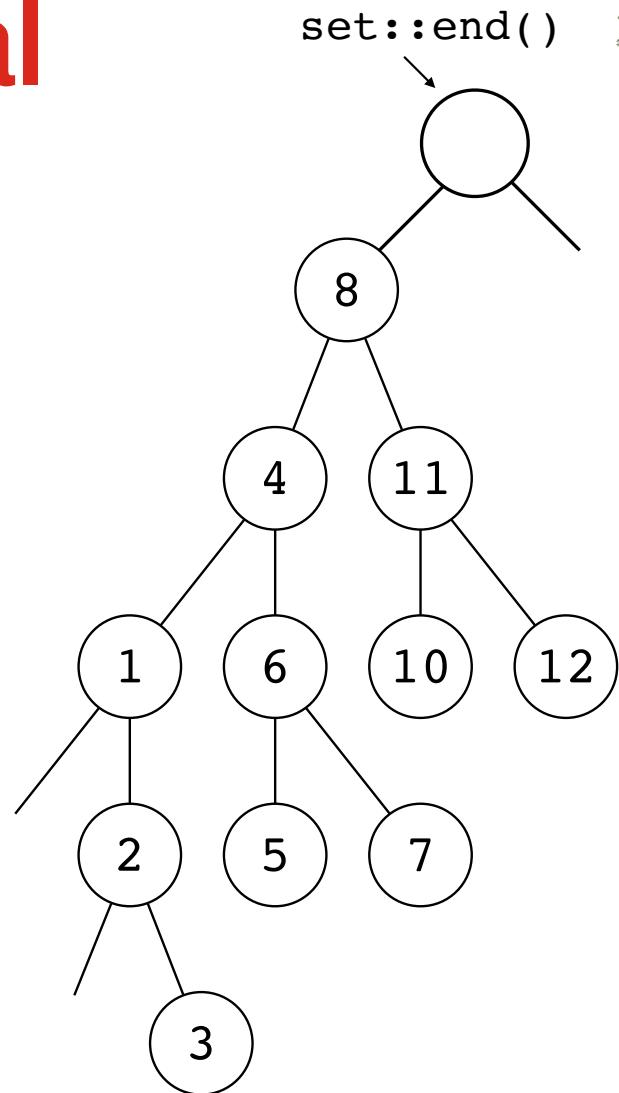
- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide
- L'itérateur `set::end()` pointe vers ce noeud final





# Noeud final

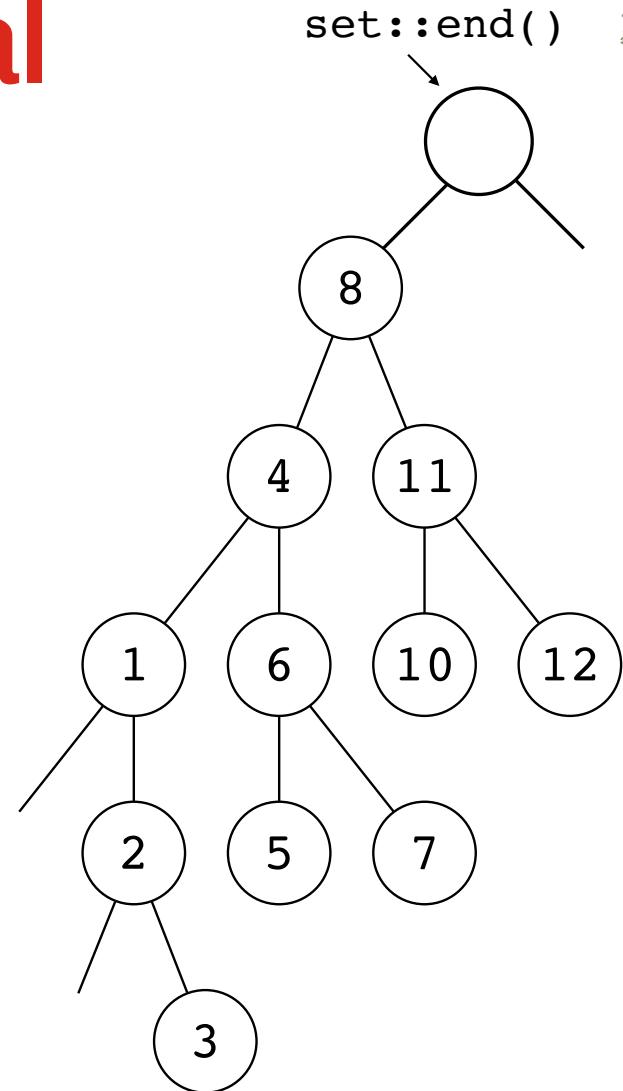
- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide
- L'itérateur `set::end()` pointe vers ce noeud final





# Noeud final

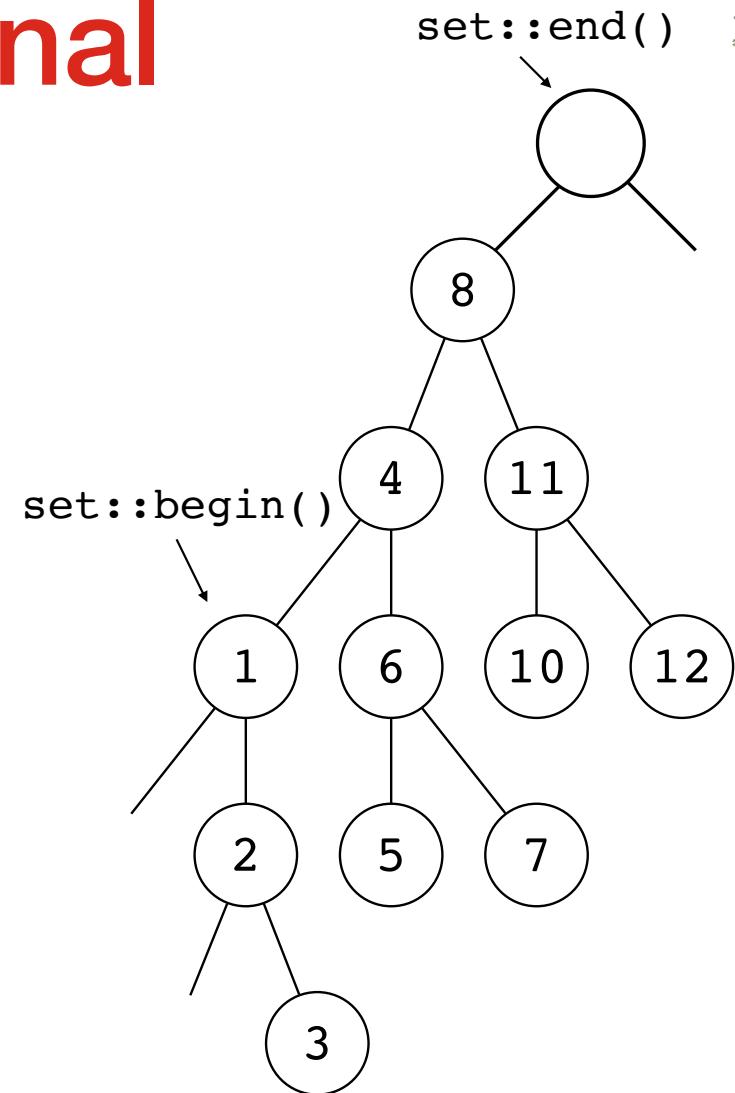
- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide
- L'itérateur `set::end()` pointe vers ce noeud final
- Rend possible une opération telle que `std::prev(s.end())`





# Noeud final

- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide
- L'itérateur `set::end()` pointe vers ce noeud final
- Rend possible une opération telle que `std::prev(s.end())`





# Insertion

```
(1) pair<iterator,bool> insert (const value_type& val);
    pair<iterator,bool> insert (value_type&& val);
    pair<iterator,bool> emplace (Args&&... args);

(2) iterator insert (const_iterator p, const value_type& val);
    iterator insert (const_iterator p, value_type&& val);
    iterator emplace_hint( const_iterator p, Args&&... args);

(3) void insert (InputIterator first, InputIterator last);

(4) void insert (initializer_list<value_type> il);
```



# Insertion

- Insertion par la clé en  $O(\log(n))$

```
set<int> s {4, 2, 1, 3};
pair<set<int>::iterator, bool> p;
p = s.insert(5); // *(p.first) = 5
                  // p.second = true
p = s.insert(3); // *(p.first) = 3
                  // p.second = false
for (auto e : s) cout << e; // 12345
```



# Insertion

- Insertion par la clé en  $O(\log(n))$
- Insertion avec indice en  $O(1)$  s'il est correct

```
set<int> s {4, 2, 1, 3};
pair<set<int>::iterator, bool> p;
p = s.insert(5); // *(p.first) = 5
                  // p.second = true
p = s.insert(3); // *(p.first) = 3
                  // p.second = false
for (auto e : s) cout << e; // 12345
```

```
set<int> s {1, 2, 3};
set<int>::iterator i;
i = s.insert(s.end(), 5);      // *i = 5
i = s.insert(i, 4);           // *i = 4
i = s.insert(s.begin(), 0);   // *i = 0
for (auto e : s) cout << e; // 012345
```



# Insertion

- Insertion par la clé en  $O(\log(n))$
- Insertion avec indice en  $O(1)$  s'il est correct
- Insertion d'une plage de  $n$  clés dans un set de taille  $s$  en  $O(n * \log(n+s))$

```
set<int> s {4, 2, 1, 3};  
pair<set<int>::iterator, bool> p;  
p = s.insert(5); // *(p.first) = 5  
// p.second = true  
p = s.insert(3); // *(p.first) = 3  
// p.second = false  
for (auto e : s) cout << e; // 12345
```

```
set<int> s {1, 2, 3};  
set<int>::iterator i;  
i = s.insert(s.end(), 5); // *i = 5  
i = s.insert(i, 4); // *i = 4  
i = s.insert(s.begin(), 0); // *i = 0  
for (auto e : s) cout << e; // 012345
```

```
set<int> s {0, 6, 3};  
list<int> L{4, 1, 2, 3};  
s.insert(L.begin(), L.end());  
for (auto e : s) cout << e; // 012346
```



# Insertion

- Insertion par la clé en  $O(\log(n))$
- Insertion avec indice en  $O(1)$  s'il est correct
- Insertion d'une plage de  $n$  clés dans un set de taille  $s$  en  $O(n * \log(n+s))$
- Insertion d'une liste d'initialisation

```
set<int> s {4, 2, 1, 3};
pair<set<int>::iterator, bool> p;
p = s.insert(5); // *(p.first) = 5
                  // p.second = true
p = s.insert(3); // *(p.first) = 3
                  // p.second = false
for (auto e : s) cout << e; // 12345
```

```
set<int> s {1, 2, 3};
set<int>::iterator i;
i = s.insert(s.end(), 5); // *i = 5
i = s.insert(i, 4); // *i = 4
i = s.insert(s.begin(), 0); // *i = 0
for (auto e : s) cout << e; // 012345
```

```
set<int> s {0, 6, 3};
list<int> L{4, 1, 2, 3};
s.insert(L.begin(), L.end());
for (auto e : s) cout << e; // 012346
```

```
set<int> s {0, 3, 6};
s.insert({1, 3, 5});
for (auto e : s) cout << e; // 01356
```



# Recherche

- 3 fonctions disponibles

```
const_iterator find (const value_type& val) const;  
const_iterator lower_bound (const value_type& val) const;  
const_iterator upper_bound (const value_type& val) const;
```



# Recherche

- 3 fonctions disponibles

```
const_iterator find (const value_type& val) const;
const_iterator lower_bound (const value_type& val) const;
const_iterator upper_bound (const value_type& val) const;
```

- Valeurs de retour :

```
set<int> s { 0, 2, 4, 6, 8 };
set<int>::iterator i;

i = s.find(2);           // *f = 2
i = s.lower_bound(2);   // *f = 2
i = s.upper_bound(2);   // *f = 4

i = s.find(3);           // f = s.end()
i = s.lower_bound(3);   // *f = 4
i = s.upper_bound(3);   // *f = 4
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin()),3));  
for (auto e : s) cout << e; // 024789
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789  
  
s.erase(s.lower_bound(2),s.upper_bound(8));  
for (auto e : s) cout << e; // 09
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

- Complexités en ...

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789  
  
s.erase(s.lower_bound(2),s.upper_bound(8));  
for (auto e : s) cout << e; // 09
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

- Complexités en ...
  - $O(1)$

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789  
  
s.erase(s.lower_bound(2),s.upper_bound(8));  
for (auto e : s) cout << e; // 09
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

- Complexités en ...
  - $O(1)$
  - $O(\log(n))$

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789  
  
s.erase(s.lower_bound(2),s.upper_bound(8));  
for (auto e : s) cout << e; // 09
```



# Suppression

- 3 interfaces

```
(1) iterator erase (const_iterator position);  
(2) size_type erase (const value_type& val);  
(3) iterator erase (const_iterator first, const_iterator last);
```

- Complexités en ...
  - $O(1)$
  - $O(\log(n))$
  - $O(\text{distance}(\text{first}, \text{last}))$

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789  
  
s.erase(s.lower_bound(2),s.upper_bound(8));  
for (auto e : s) cout << e; // 09
```

# 19. std::map



# std::map <Key, Value, Compare>



- Met en oeuvre le TDA tableau associatif

# std::map <Key, Value, Compare>



- Met en oeuvre le TDA tableau associatif
- Essentiellement un std::set contenant des éléments de type

`std::pair<const Key, Value>`

# std::map <Key, Value, Compare>



- Met en oeuvre le TDA tableau associatif
- Essentiellement un std::set contenant des éléments de type

`std::pair<const Key, Value>`

- Dont la fonction de comparaison ne tient compte que des clés



# std::map <Key, Value, Compare>

- Met en oeuvre le TDA tableau associatif
- Essentiellement un std::set contenant des éléments de type

```
std::pair<const Key, Value>
```

- Dont la fonction de comparaison ne tient compte que des clés
- Et qui fournit l'opérateur crochet

```
Value& map<Key, Value>::  
operator[ ](const Key& k)
```



# map<K, V>::operator[ ]

- Selon la documentation, l'opérateur s'écrit...

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    return (*((this->insert(make_pair(k,Value()))).first)).second;
}
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,v()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,V()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    V                                v = V();
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,v()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    V v = V();
    pair<K,V> p = make_pair(k,v);
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,v()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    V                                v = V();
    pair<K,V>                      p = make_pair(k,v);
    pair<map<K,V>::iterator,bool> r = this->insert(p);
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,v()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    V                                v = V();
    pair<K,V>                      p = make_pair(k,v);
    pair<map<K,V>::iterator,bool> r = this->insert(p);
    map<K,V>::iterator              i = r.first;
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,v()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    V                                v = V();
    pair<K,V>                      p = make_pair(k,v);
    pair<map<K,V>::iterator,bool> r = this->insert(p);
    map<K,V>::iterator              i = r.first;
    pair<K,V>&                     q = *i;
```



# map<K,V>::operator[] en détail

```
return (*((this->insert(make_pair(k,v()))).first)).second;
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    V                                v = V();
    pair<K,V>                      p = make_pair(k,v);
    pair<map<K,V>::iterator,bool> r = this->insert(p);
    map<K,V>::iterator              i = r.first;
    pair<K,V>&                     q = *i;
    V&                            w = q.second;
    return w;
}
```



# map<K, V>::operator[ ] , utilisation

```
string chaine = "abracadabra";
std::map<char, size_t> m;
```



# map<K, V>::operator[ ] , utilisation

```
string chaine = "abracadabra";
std::map<char, size_t> m;

size_t i = 0;
for(char c : chaine) m[c] = i++;
```



# map<K, V>::operator[ ] , utilisation

```
string chaine = "abracadabra";
std::map<char, size_t> m;

size_t i = 0;
for(char c : chaine) m[c] = i++;

for(auto p : m)
    cout << "m[" << p.first << "]=" << p.second << "\n";
```



# map<K, V>::operator[ ] , utilisation

```
string chaine = "abracadabra";
std::map<char, size_t> m;

size_t i = 0;
for(char c : chaine) m[c] = i++;

for(auto p : m)
    cout << "m[" << p.first << "]=" << p.second << "\n";
```

m[a]=10  
m[b]=8  
m[c]=4  
m[d]=6  
m[r]=9



# Attention, lire `m[c]` insère un élément si la clé `c` est absente

```
string chaine = "abracadabra";
std::map<char, size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;
```



# Attention, lire `m[c]` insère un élément si la clé `c` est absente

```
string chaine = "abracadabra";
std::map<char,size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";
```



# Attention, lire `m[c]` insère un élément si la clé `c` est absente

```
string chaine = "abracadabra";
std::map<char,size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";
```

m.size(): 5



# Attention, lire $m[c]$ insère un élément si la clé $c$ est absente

```
string chaine = "abracadabra";
std::map<char,size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";

for(char c = 'c'; c <= 'e'; ++c)
    cout << "m[" << c << "]=" << m[c] << "\n";
```

`m.size(): 5`



# Attention, lire `m[c]` insère un élément si la clé `c` est absente

```
string chaine = "abracadabra";
std::map<char,size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";

for(char c = 'c'; c <= 'e'; ++c)
    cout << "m[" << c << "]=" << m[c] << "\n";
```

`m.size(): 5`  
`m[c]=4`  
`m[d]=6`  
`m[e]=0`



# Attention, lire `m[c]` insère un élément si la clé `c` est absente

```
string chaine = "abracadabra";
std::map<char,size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";

for(char c = 'c'; c <= 'e'; ++c)
    cout << "m[" << c << "]=" << m[c] << "\n";

cout << "m.size(): " << m.size() << "\n";
```

`m.size(): 5`

`m[c]=4`

`m[d]=6`

`m[e]=0`



# Attention, lire `m[c]` insère un élément si la clé `c` est absente

```
string chaine = "abracadabra";
std::map<char,size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";

for(char c = 'c'; c <= 'e'; ++c)
    cout << "m[" << c << "]=" << m[c] << "\n";

cout << "m.size(): " << m.size() << "\n";
```

`m.size(): 5`

`m[c]=4`

`m[d]=6`

`m[e]=0`

`m.size(): 6`

# map<K,V>::find(K const& k)



// On peut l'éviter en utilisant la fonction find.

```
std::cout << "m.size(): " << m.size() << "\n";
```

m.size(): 6

# map<K,V>::find(K const& k)



// On peut l'éviter en utilisant la fonction find.

m.size(): 6

```
std::cout << "m.size(): " << m.size() << "\n";  
  
for(char c = 'd'; c <= 'f'; ++c)  
{
```

# map<K,V>::find(K const& k)



// On peut l'éviter en utilisant la fonction find.

m.size(): 6

```
std::cout << "m.size(): " << m.size() << "\n";  
  
for(char c = 'd'; c <= 'f'; ++c)  
{  
    auto it = m.find(c);
```

# map<K,V>::find(K const& k)



// On peut l'éviter en utilisant la fonction find.

m.size(): 6

```
std::cout << "m.size(): " << m.size() << "\n";\n\nfor(char c = 'd'; c <= 'f'; ++c)\n{\n    auto it = m.find(c);\n    if(it != m.end())\n        std::cout << "m[" << c << "]=" << (*it).second << "\n";\n}
```



# map<K,V>::find(K const& k)

// On peut l'éviter en utilisant la fonction find.

m.size(): 6

```
std::cout << "m.size(): " << m.size() << "\n";\n\nfor(char c = 'd'; c <= 'f'; ++c)\n{\n    auto it = m.find(c);\n    if(it != m.end())\n        std::cout << "m[" << c << "]=" << (*it).second << "\n";\n    else\n        std::cout << "m[" << c << "] est absent\n";\n}
```

# map<K,V>::find(K const& k)



```
// On peut l'éviter en utilisant la fonction find.

std::cout << "m.size(): " << m.size() << "\n";

for(char c = 'd'; c <= 'f'; ++c)
{
    auto it = m.find(c);
    if(it != m.end())
        std::cout << "m[" << c << "]=" << (*it).second << "\n";
    else
        std::cout << "m[" << c << "] est absent\n";
}
```

m.size(): 6

m[d]=6

m[e]=0

m[f] est absent

# map<K,V>::find(K const& k)



```
// On peut l'éviter en utilisant la fonction find.

std::cout << "m.size(): " << m.size() << "\n";

for(char c = 'd'; c <= 'f'; ++c)
{
    auto it = m.find(c);
    if(it != m.end())
        std::cout << "m[" << c << "]=" << (*it).second << "\n";
    else
        std::cout << "m[" << c << "] est absent\n";
}

std::cout << "m.size(): " << m.size() << "\n";
```

m.size(): 6

m[d]=6

m[e]=0

m[f] est absent

# map<K,V>::find(K const& k)



```
// On peut l'éviter en utilisant la fonction find.

std::cout << "m.size(): " << m.size() << "\n";

for(char c = 'd'; c <= 'f'; ++c)
{
    auto it = m.find(c);
    if(it != m.end())
        std::cout << "m[" << c << "]=" << (*it).second << "\n";
    else
        std::cout << "m[" << c << "] est absent\n";
}

std::cout << "m.size(): " << m.size() << "\n";
```

m.size(): 6

m[d]=6

m[e]=0

m[f] est absent

m.size(): 6



# multiset et multimap

- Comme set et map, mais permettent de stocker plusieurs fois la même clé



# multiset et multimap

- Comme set et map, mais permettent de stocker plusieurs fois la même clé
- Pas d'operator [ ] pour multimap, il serait ambigu



# multiset et multimap

- Comme set et map, mais permettent de stocker plusieurs fois la même clé
- Pas d'operator [ ] pour multimap, il serait ambigu
- La méthode equal\_range retourne toutes les clés équivalentes

```
template <typename T>
pair<multiset<T>::iterator, multiset<T>::iterator>
multiset<T>::equal_range (const T& k)
{
    return make_pair(lower_bound(k),upper_bound(k));
}
```

# Empreintes mémoire de set et map





# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille

`2*sizeof(T*) +  
1*sizeof(size_t)`



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille
- Coût par noeud :
  - 2 pointeurs vers les enfants
  - 1 pointeur vers le parent
  - ... pour l'information d'équilibrage : 1 bit pour les arbres rouge-noir
  - Et évidemment la mémoire pour stocker la clé (et la valeur pour `std::map`)
  - Plus éventuellement des bytes d'alignement, mais nous n'en tenons pas compte en ASD1

```
2*sizeof(T*) +  
1*sizeof(size_t)
```



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille
- Coût par noeud :
  - 2 pointeurs vers les enfants
  - 1 pointeur vers le parent
  - ... pour l'information d'équilibrage : 1 bit pour les arbres rouge-noir
  - Et évidemment la mémoire pour stocker la clé (et la valeur pour `std::map`)
  - Plus éventuellement des bytes d'alignement, mais nous n'en tenons pas compte en ASD1

```
2*sizeof(T*) +  
1*sizeof(size_t)
```



# Empreintes mémoire de set et map

- Coût fixe :
  - 1 pointeur vers le noeud minimum `set::begin()`
  - 1 pointeur vers le noeud final `set::end()`
  - 1 `size_t` pour stocker la taille
- Coût par noeud :
  - 2 pointeurs vers les enfants
  - 1 pointeur vers le parent
  - ... pour l'information d'équilibrage : 1 bit pour les arbres rouge-noir
  - Et évidemment la mémoire pour stocker la clé (et la valeur pour `std::map`)
  - Plus éventuellement des bytes d'alignement, mais nous n'en tenons pas compte en ASD1

`2*sizeof(T*) +  
1*sizeof(size_t)`

`3*sizeof(void*) + ... + sizeof(Key) +  
sizeof(Value)`

# 19 . STL set theory operations + output iterators





# <algorithm>

**Merge** (operating on sorted ranges):

<b>merge</b>	Merge sorted ranges (function template )
<b>inplace_merge</b>	Merge consecutive sorted ranges (function template )
<b>includes</b>	Test whether sorted range includes another sorted range (function template )
<b>set_union</b>	Union of two sorted ranges (function template )
<b>set_intersection</b>	Intersection of two sorted ranges (function template )
<b>set_difference</b>	Difference of two sorted ranges (function template )
<b>set_symmetric_difference</b>	Symmetric difference of two sorted ranges (function template )



# Opérations ensemblistes

```
template <class InputIterator1, class InputIterator2,  
         class OutputIterator>  
OutputIterator set_xxx ( InputIterator1 first1, InputIterator1 last1,  
                         InputIterator2 first2, InputIterator2 last2,  
                         OutputIterator result);
```



# Opérations ensemblistes

```
template <class InputIterator1, class InputIterator2,  
         class OutputIterator>  
OutputIterator set_xxx ( InputIterator1 first1, InputIterator1 last1,  
                         InputIterator2 first2, InputIterator2 last2,  
                         OutputIterator result);
```

- En entrée, des plages d'éléments triés



# Opérations ensemblistes

```
template <class InputIterator1, class InputIterator2,  
         class OutputIterator>  
OutputIterator set_xxx ( InputIterator1 first1, InputIterator1 last1,  
                         InputIterator2 first2, InputIterator2 last2,  
                         OutputIterator result);
```

- En entrée, des plages d'éléments triés
- Ecrivent dans un itérateur de sortie



# Opérations ensemblistes

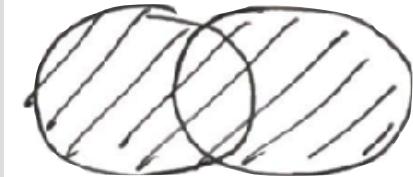
```
template <class InputIterator1, class InputIterator2,  
         class OutputIterator>  
OutputIterator set_xxx ( InputIterator1 first1, InputIterator1 last1,  
                         InputIterator2 first2, InputIterator2 last2,  
                         OutputIterator result);
```

- En entrée, des plages d'éléments triés
- Ecrivent dans un itérateur de sortie
- Retournent l'itérateur suivant le dernier écrit



# set\_union

$E_1 \cup E_2$

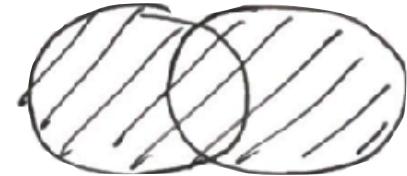


```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)
{
```



# set\_union

$E_1 \cup E_2$



```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)
{
    while (true)
    {
```

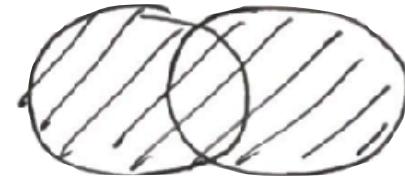


# set\_union

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)

{
    while (true)
    {
        if (first1==last1) return std::copy(first2,last2,result);
        if (first2==last2) return std::copy(first1,last1,result);
```

E1 ∪ E2





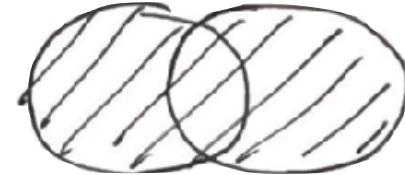
# set\_union

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)

{
    while (true)
    {
        if (first1==last1) return std::copy(first2,last2,result);
        if (first2==last2) return std::copy(first1,last1,result);

        if (*first1<*first2) { *result = *first1; ++first1; }
```

E1 ∪ E2





# set\_union

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)

{
    while (true)
    {
        if (first1==last1) return std::copy(first2,last2,result);
        if (first2==last2) return std::copy(first1,last1,result);

        if (*first1<*first2) { *result = *first1; ++first1; }
        else if (*first2<*first1) { *result = *first2; ++first2; }
```



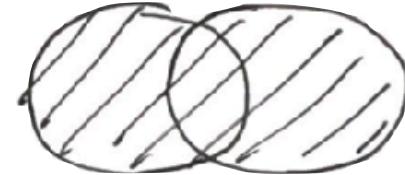
# set\_union

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)

{
    while (true)
    {
        if (first1==last1) return std::copy(first2,last2,result);
        if (first2==last2) return std::copy(first1,last1,result);

        if (*first1<*first2) { *result = *first1; ++first1; }
        else if (*first2<*first1) { *result = *first2; ++first2; }
        else { *result = *first1; ++first1; ++first2; }
    }
}
```

E1 ∪ E2





# set\_union

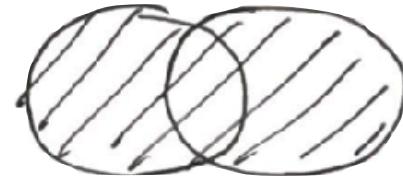
```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)

{
    while (true)
    {
        if (first1==last1) return std::copy(first2,last2,result);
        if (first2==last2) return std::copy(first1,last1,result);

        if (*first1<*first2) { *result = *first1; ++first1; }
        else if (*first2<*first1) { *result = *first2; ++first2; }
        else { *result = *first1; ++first1; ++first2; }

        ++result;
    }
}
```

E1 ∪ E2

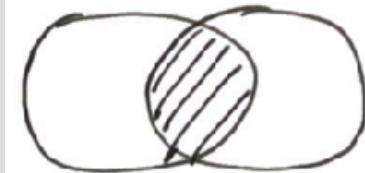




# set\_intersection

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                           InputIter2 first2, InputIter2 last2,
                           OutputIter result)
{
```

$$E_1 \cap E_2$$

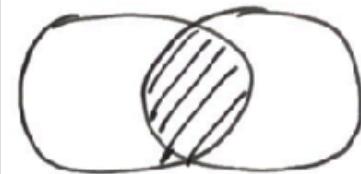




# set\_intersection

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                           InputIter2 first2, InputIter2 last2,
                           OutputIter result)
{
    while (first1!=last1 && first2!=last2)
    {
```

$$E_1 \cap E_2$$

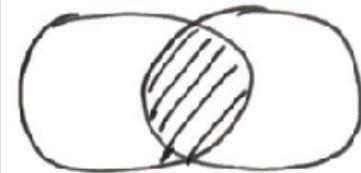




# set\_intersection

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                           InputIter2 first2, InputIter2 last2,
                           OutputIter result)
{
    while (first1!=last1 && first2!=last2)
    {
        if (*first1<*first2) ++first1;
```

$$E_1 \cap E_2$$

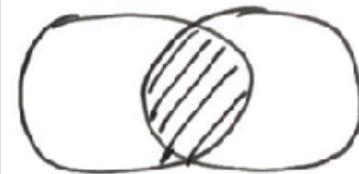




# set\_intersection

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                           InputIter2 first2, InputIter2 last2,
                           OutputIter result)
{
    while (first1!=last1 && first2!=last2)
    {
        if (*first1<*first2) ++first1;
        else if (*first2<*first1) ++first2;
```

$$E_1 \cap E_2$$

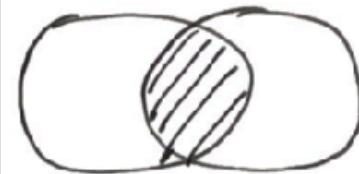




# set\_intersection

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                           InputIter2 first2, InputIter2 last2,
                           OutputIter result)
{
    while (first1!=last1 && first2!=last2)
    {
        if (*first1<*first2) ++first1;
        else if (*first2<*first1) ++first2;
        else {
            *result = *first1;
            ++result; ++first1; ++first2;
        }
    }
}
```

$$E_1 \cap E_2$$

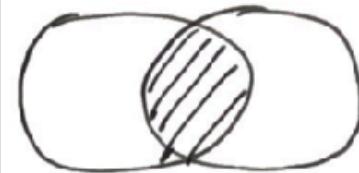




# set\_intersection

```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                           InputIter2 first2, InputIter2 last2,
                           OutputIter result)
{
    while (first1!=last1 && first2!=last2)
    {
        if (*first1<*first2) ++first1;
        else if (*first2<*first1) ++first2;
        else {
            *result = *first1;
            ++result; ++first1; ++first2;
        }
    }
    return result;
}
```

$$E_1 \cap E_2$$





# En entrée ?

```
#include <set>
std::set<int> E1 = {5, 10, 15, 20, 25, 5, 10};
// E1 contient {5, 10, 15, 20, 25}
```



# En entrée ?

```
#include <set>
std::set<int> E1 = {5, 10, 15, 20, 25, 5, 10};
// E1 contient {5, 10, 15, 20, 25}
```

```
#include <list>
std::list<int> E2 = { 50, 40, 10, 20, 30, 10, 40 };
E2.sort();
E2.unique();                                // E2 contient {10, 20, 30, 40, 50}
```



# En entrée ?

```
#include <set>
std::set<int> E1 = {5, 10, 15, 20, 25, 5, 10};
// E1 contient {5, 10, 15, 20, 25}
```

```
#include <list>
std::list<int> E2 = { 50, 40, 10, 20, 30, 10, 40 };
E2.sort();
E2.unique(); // E2 contient {10, 20, 30, 40, 50}
```

```
#include <forward_list>
std::forward_list<int> E3 = { 5, 10, 15, 20, 25, 5, 10};
E3.sort();
E3.unique(); // E3 contient {5, 10, 15, 20, 25}
```



# En entrée ?

```
#include <set>
std::set<int> E1 = {5, 10, 15, 20, 25, 5, 10};
// E1 contient {5, 10, 15, 20, 25}
```

```
#include <list>
std::list<int> E2 = { 50, 40, 10, 20, 30, 10, 40 };
E2.sort();
E2.unique(); // E2 contient {10, 20, 30, 40, 50}
```

```
#include <forward_list>
std::forward_list<int> E3 = { 5, 10, 15, 20, 25, 5, 10};
E3.sort();
E3.unique(); // E3 contient {5, 10, 15, 20, 25}
```

```
#include <vector>
#include <algorithm>
std::vector<int> E4 = { 50, 40, 10, 20, 30, 10, 40 };
std::sort(E4.begin(),E4.end());
auto last = std::unique(E4.begin(),E4.end());
E4.erase(last,E4.end()); // E4 contient {10, 20, 30, 40, 50}
```



# En sortie ?

- La seule approche que nous connaissons consiste à allouer suffisamment (trop) avant l'appel et à effacer l'excédent après.



# En sortie ?

- La seule approche que nous connaissons consiste à allouer suffisamment (trop) avant l'appel et à effacer l'excédent après.

```
vector<int> U(E1.size()+E2.size());
```



# En sortie ?

- La seule approche que nous connaissons consiste à allouer suffisamment (trop) avant l'appel et à effacer l'excédent après.

```
vector<int> U(E1.size()+E2.size());  
auto lastU = std::set_union(E1.begin(), E1.end(),  
                           E2.begin(), E2.end(),  
                           U.begin());
```



# En sortie ?

- La seule approche que nous connaissons consiste à allouer suffisamment (trop) avant l'appel et à effacer l'excédent après.

```
vector<int> U(E1.size()+E2.size());
auto lastU = std::set_union(E1.begin(), E1.end(),
                           E2.begin(), E2.end(),
                           U.begin());
U.erase(lastU,U.end());
```



# En sortie ?

- La seule approche que nous connaissons consiste à allouer suffisamment (trop) avant l'appel et à effacer l'excédent après.

```
vector<int> U(E1.size() + E2.size());
auto lastU = std::set_union(E1.begin(), E1.end(),
                            E2.begin(), E2.end(),
                            U.begin());
U.erase(lastU, U.end());
```

- Comment faire avec `list`, `forward_list`, `set` ?



# Itérateurs de sortie

- `back_inserter`,  
`front_inserter`,  
`inserter`



# Itérateurs de sortie

- `back_inserter`,  
`front_inserter`,  
`inserter`
- Conservent une référence  
au conteneur qu'ils itèrent



# Itérateurs de sortie

- `back_inserter`,  
`front_inserter`,  
`inserter`
- Conservent une référence  
au conteneur qu'ils itèrent
- Appellent `push_back`,  
`push_front` ou `insert`  
quand ils sont référencés à  
gauche d'une affectation



# Itérateurs de sortie

- `back_inserter`,  
`front_inserter`,  
`inserter`
- Conservent une référence  
au conteneur qu'ils itèrent
- Appellent `push_back`,  
`push_front` ou `insert`  
quand ils sont référencés à  
gauche d'une affectation

Expression	Propriété
<code>x b(a);</code> <code>b = a;</code>	Constructible par copie, assignable et destructible
<code>*a = t;</code>	Déréférable à gauche, seulement à gauche d'une affectation, et une seule fois entre deux incrémentations
<code>++a;</code> <code>a++;</code> <code>*a++ = t;</code>	Incrémentable
<code>swap(a,b)</code>	Les lvalues sont échangeables



# Itérateurs de sortie

```
#include <iterator>
list<int> I;      // ou vector<int> ou deque<int>
std::set_intersection(E1.begin(), E1.end(),
                      E2.begin(), E2.end(),
                      std::back_inserter(I));
```



# Itérateurs de sortie

```
#include <iterator>
list<int> I;      // ou vector<int> ou deque<int>
std::set_intersection(E1.begin(), E1.end(),
                      E2.begin(), E2.end(),
                      std::back_inserter(I));
```

```
forward_list<int> D;
std::set_difference(E1.begin(), E1.end(),
                     E2.begin(), E2.end(),
                     std::front_inserter(D));
D.reverse();      (insérer au début inverse le résultat)
```



# Itérateurs de sortie

```
#include <iterator>  
  
list<int> I;      // ou vector<int> ou deque<int>  
std::set_intersection(E1.begin(), E1.end(),  
                      E2.begin(), E2.end(),  
                      std::back_inserter(I));
```

```
forward_list<int> D;  
  
std::set_difference(E1.begin(), E1.end(),  
                     E2.begin(), E2.end(),  
                     std::front_inserter(D));  
  
D.reverse();    (insérer au début inverse le résultat)
```

```
set<int> SD;  
  
std::set_symmetric_difference(E1.begin(), E1.end(),  
                           E2.begin(), E2.end(),  
                           std::inserter(SD, SD.end()));
```