

Arbres Binaires de Recherche



9. Structure et parcours



Arbre Binaire de Recherche

Définition



Arbre binaire tel que

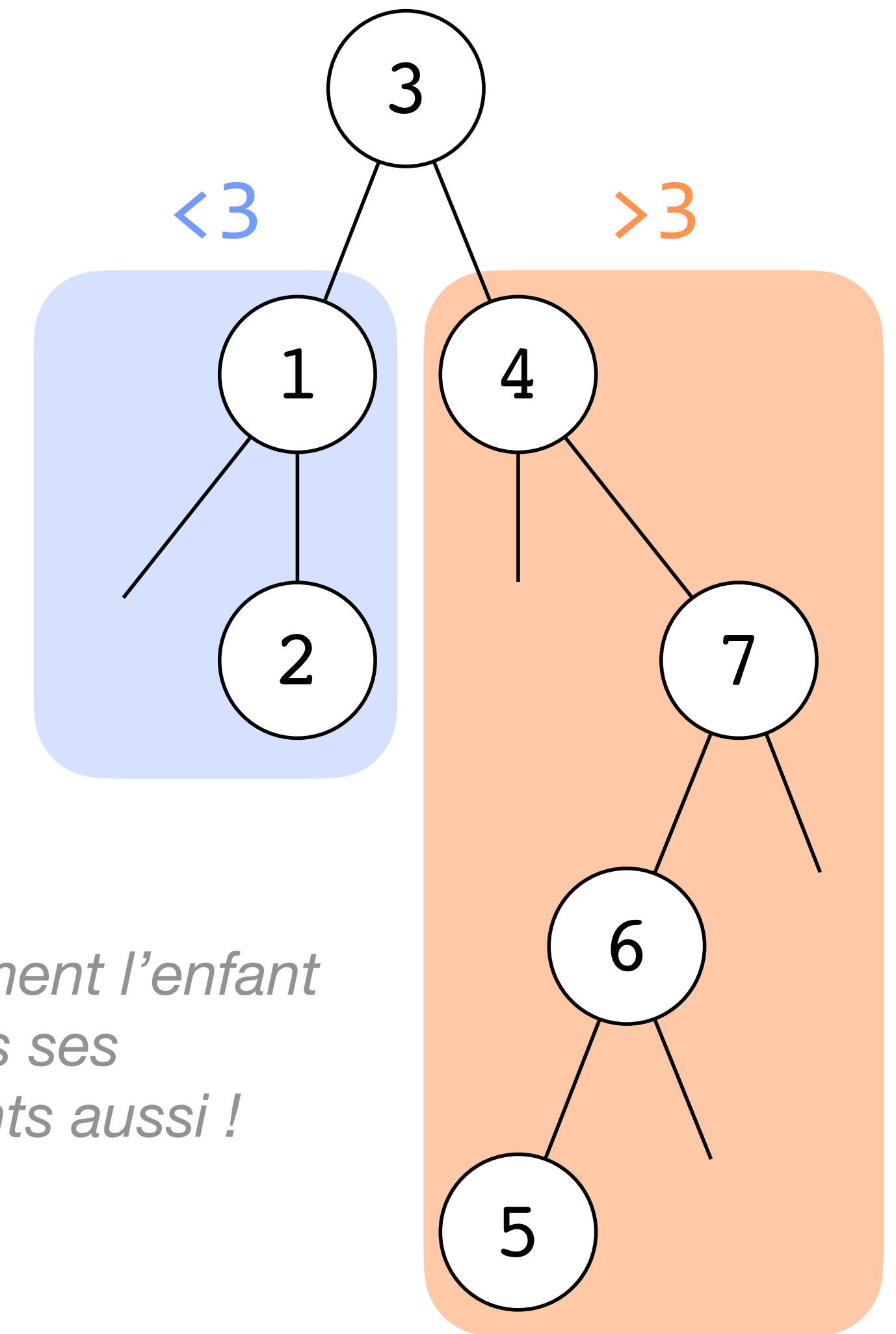
- Chaque noeud contient une **clé unique**, *en plus de l'étiquette.*

- Pour tout noeud **r**,
tout noeud **g** du sous-arbre gauche,
tout noeud **d** du sous-arbre droit,

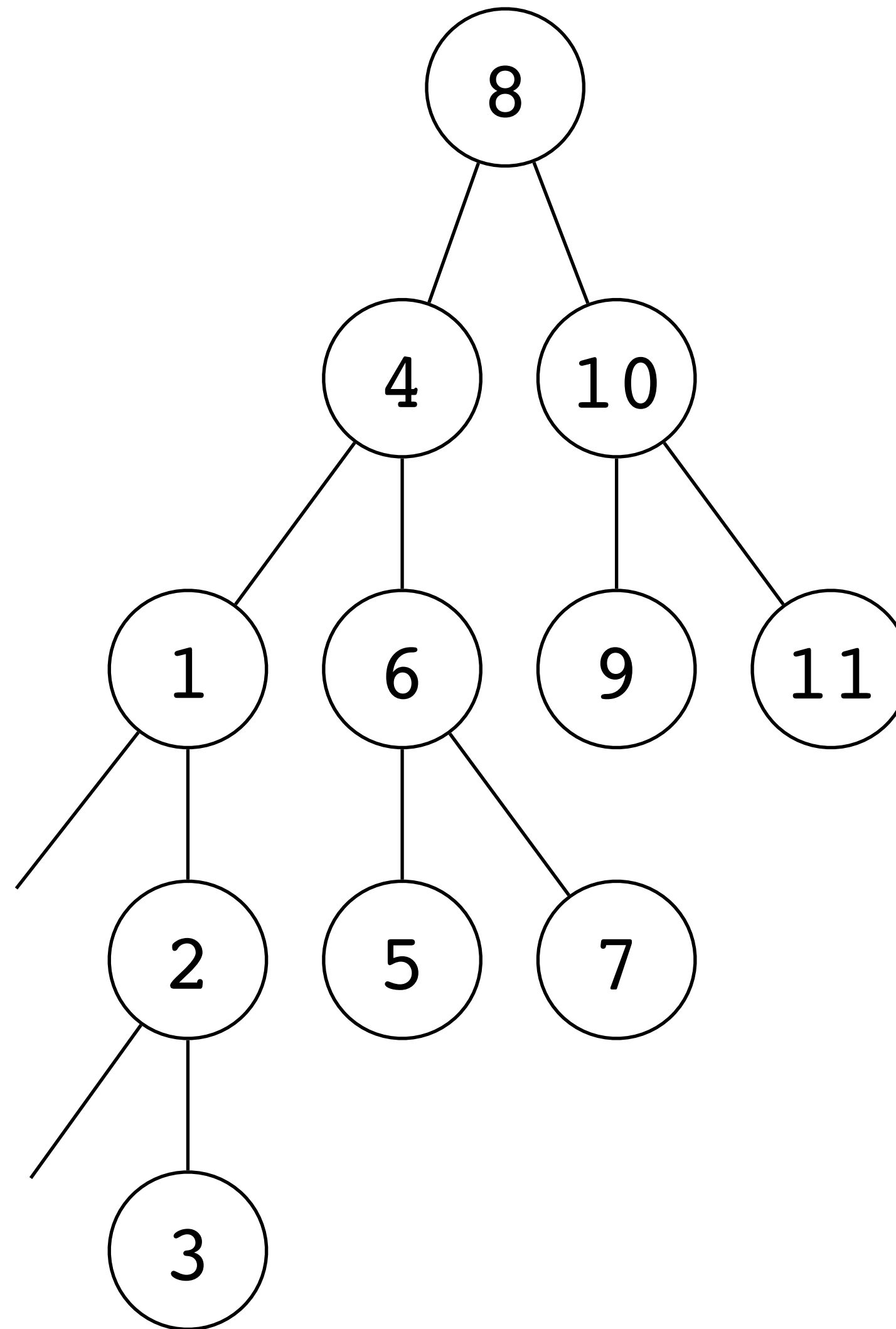
$$\mathbf{g.clé} < \mathbf{r.clé} < \mathbf{d.clé}$$



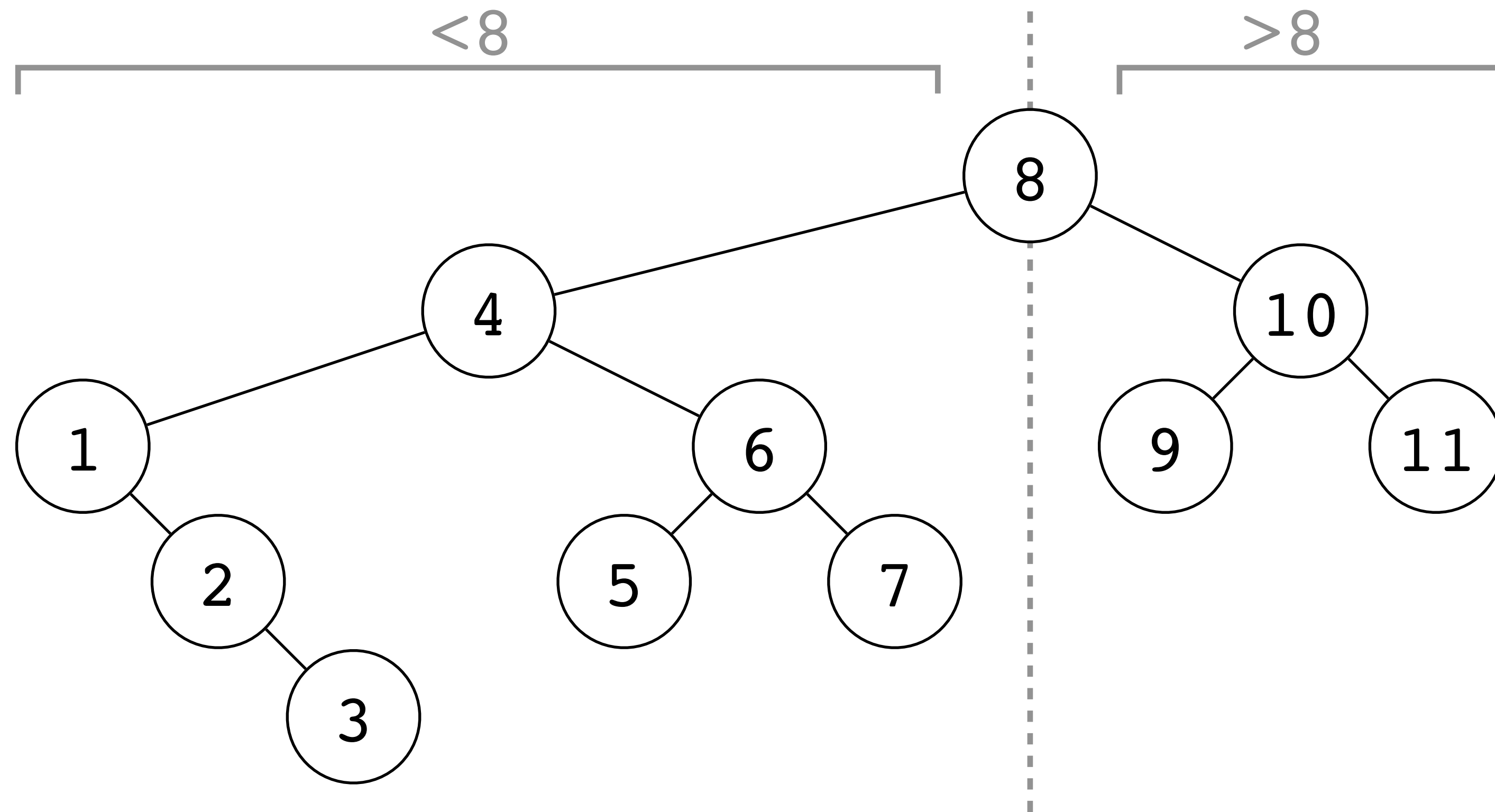
Pas seulement l'enfant direct, tous ses descendants aussi !



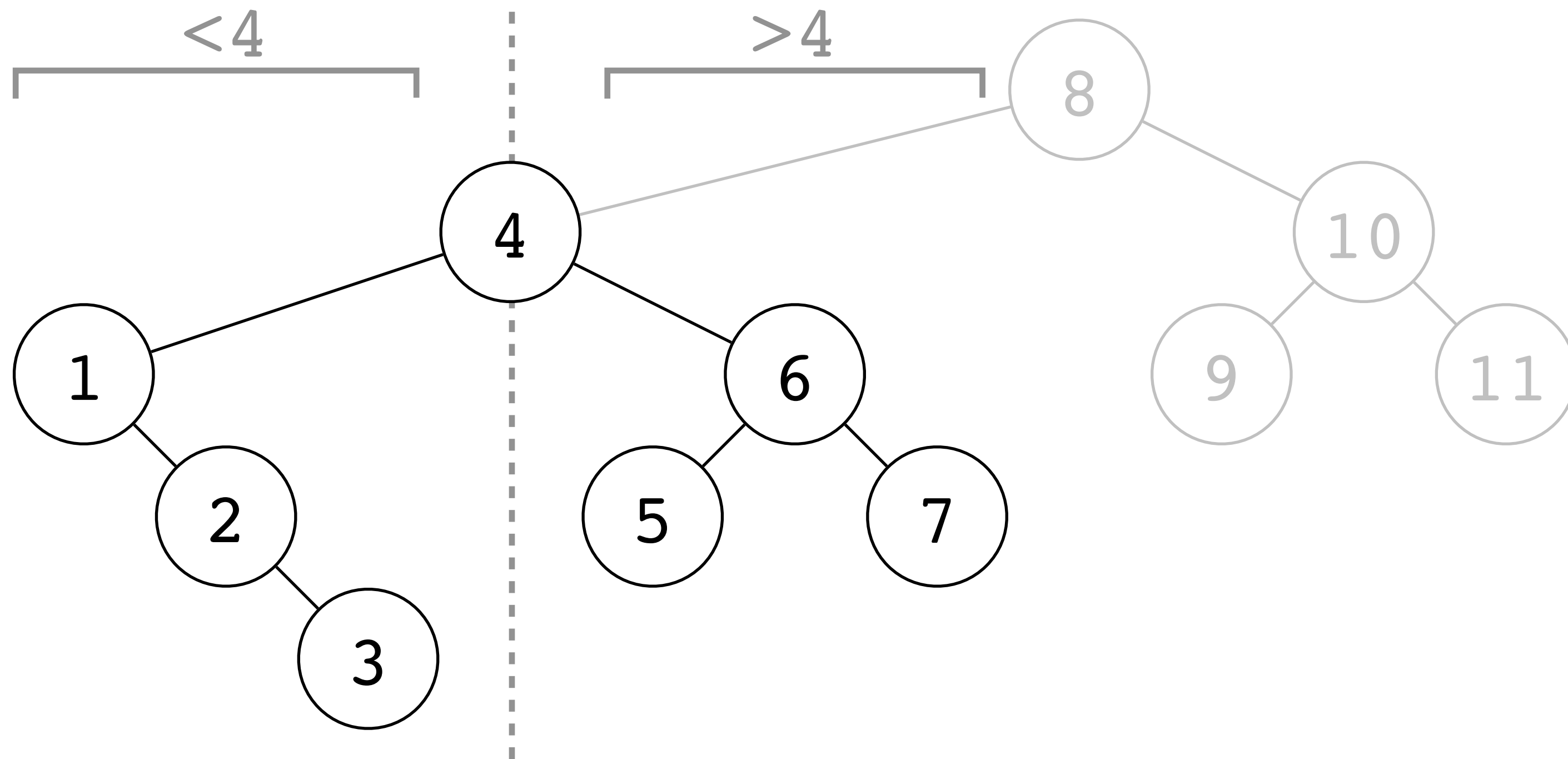
(Représentation visuelle)



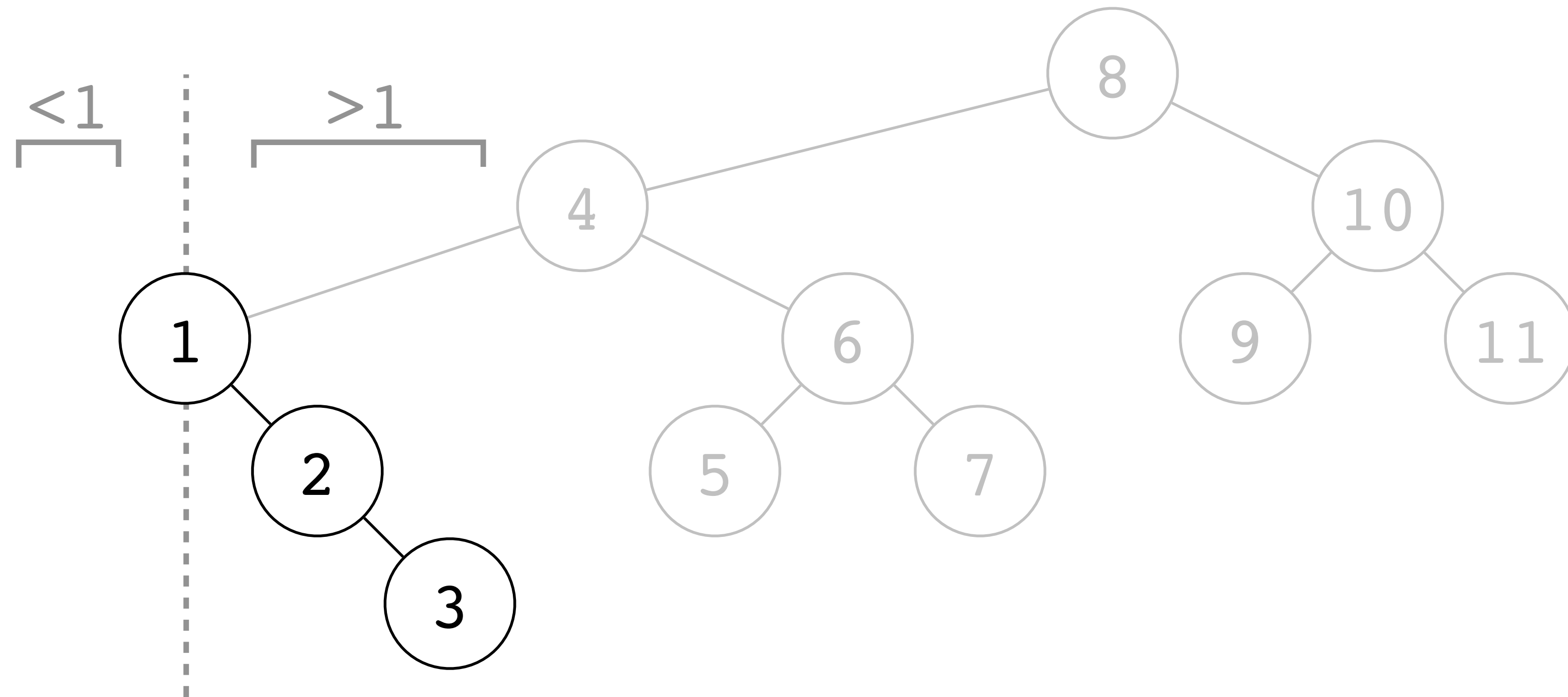
(Représentation visuelle)



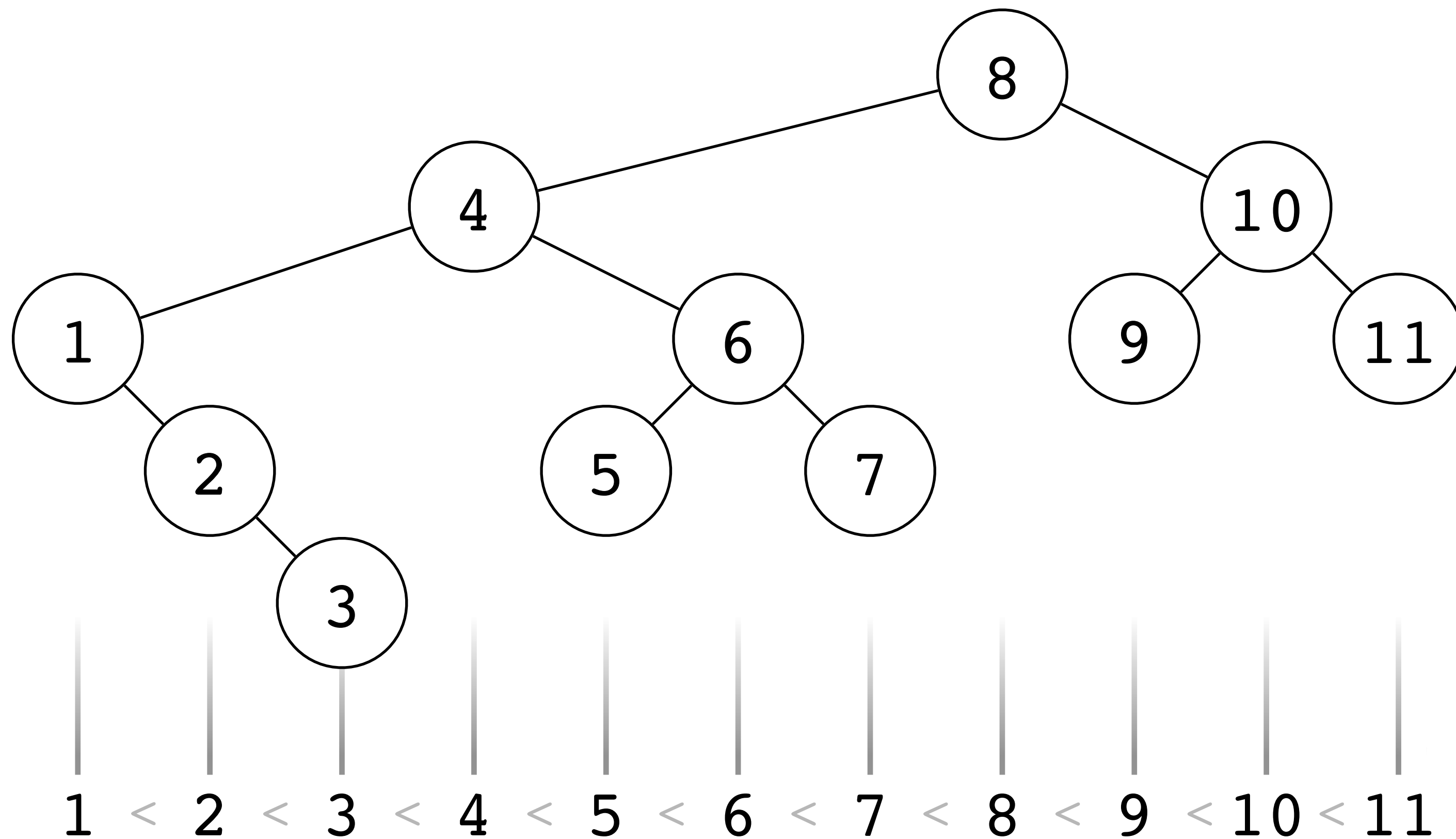
(Représentation visuelle)



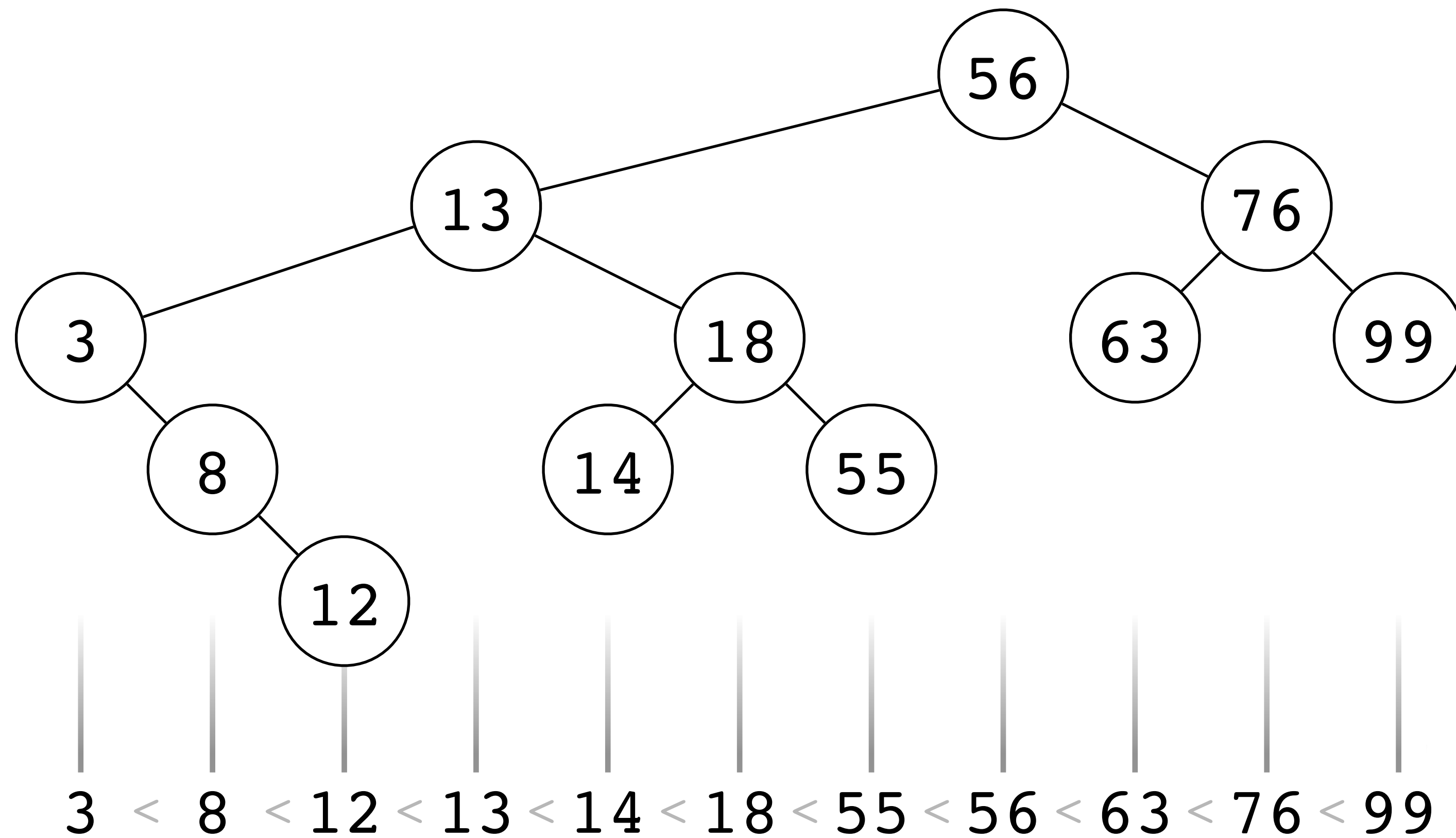
(Représentation visuelle)



(Représentation visuelle)

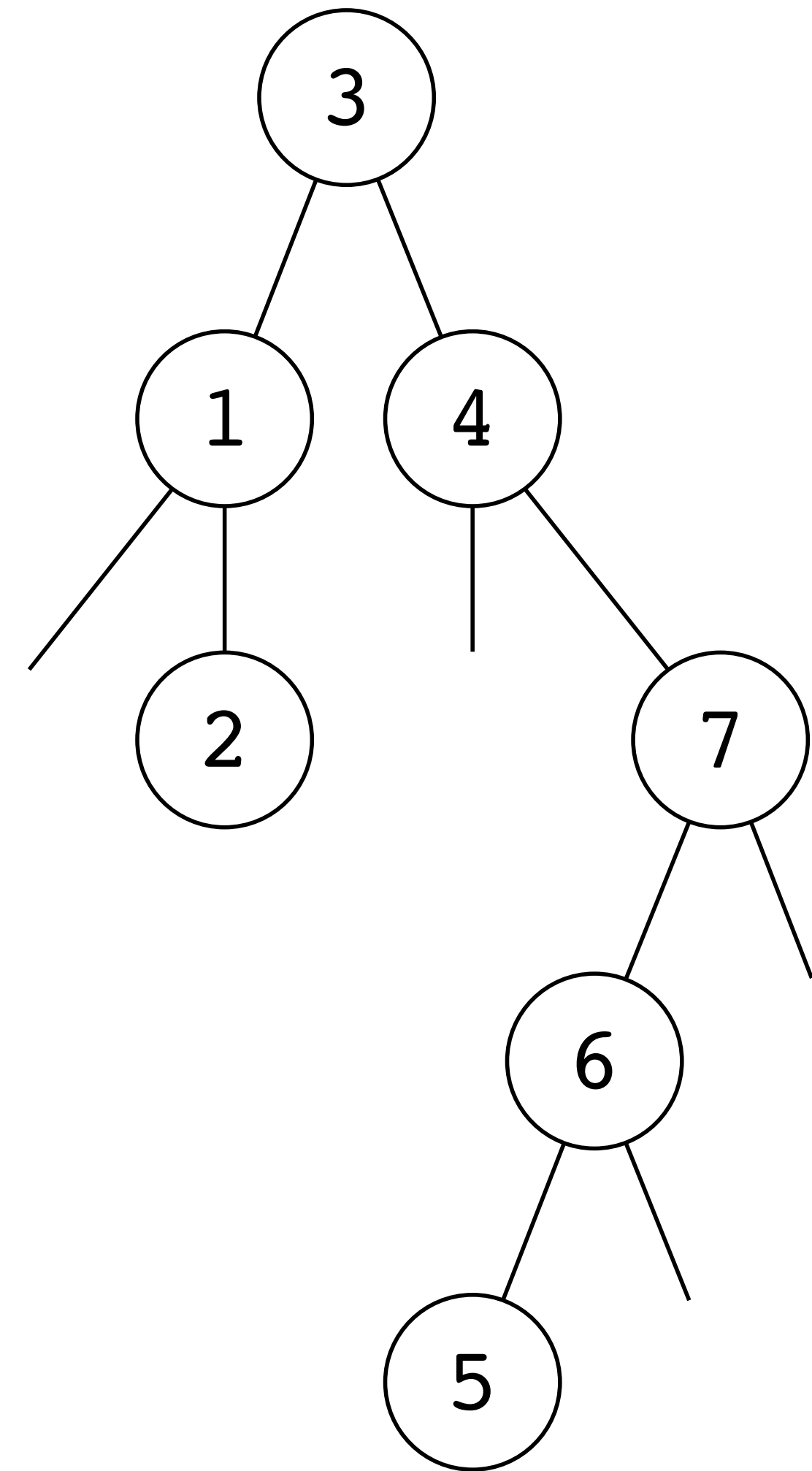


(Représentation visuelle)





- Pour **chercher** une clé donnée on sait directement de quel coté aller
- 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





- Au minimum, on stocke la **clé** et les liens vers les **enfants** gauche et droit.
- La **valeur** (a.k.a. étiquette) 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
```

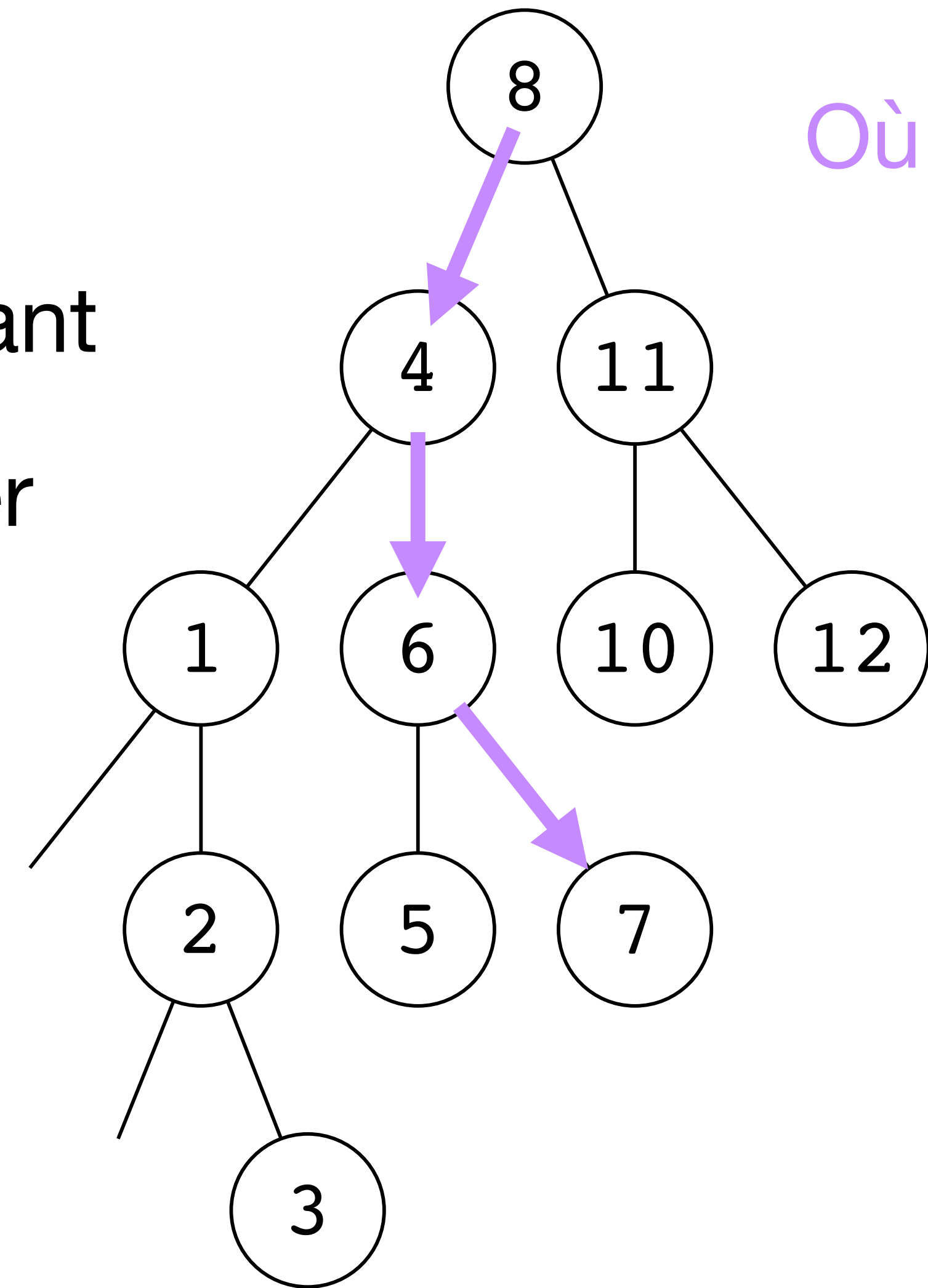


Recherche



Récurivement :

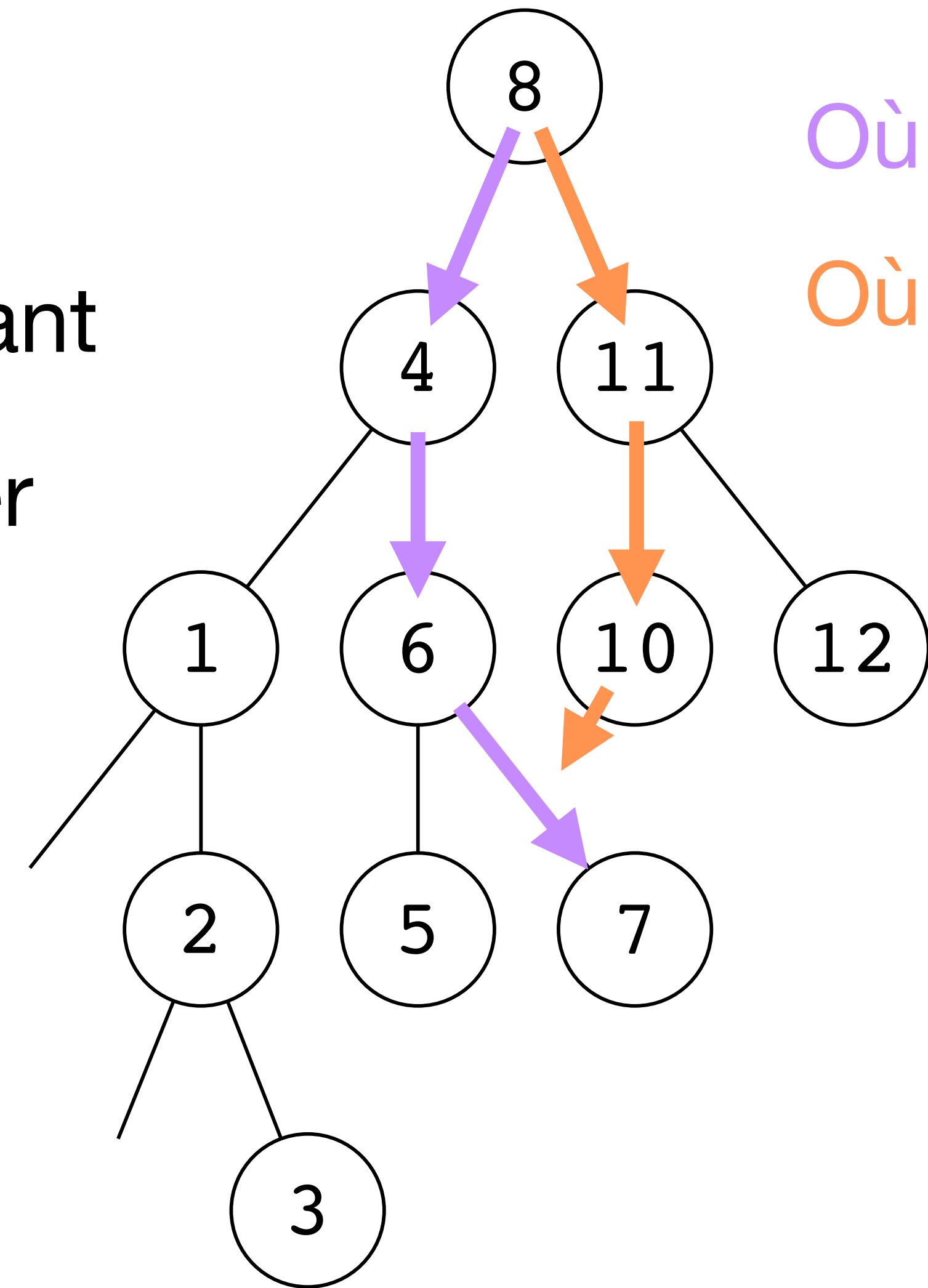
- Comparer la clé cherchée au noeud courant
- Si différente, on sait de quel coté chercher
- Si égale, on a trouvé





Récurivement :

- Comparer la clé cherchée au noeud courant
- Si différente, on sait de quel coté chercher
- Si égale, on a trouvé
- Si on arrive à un noeud vide, la clé est absente

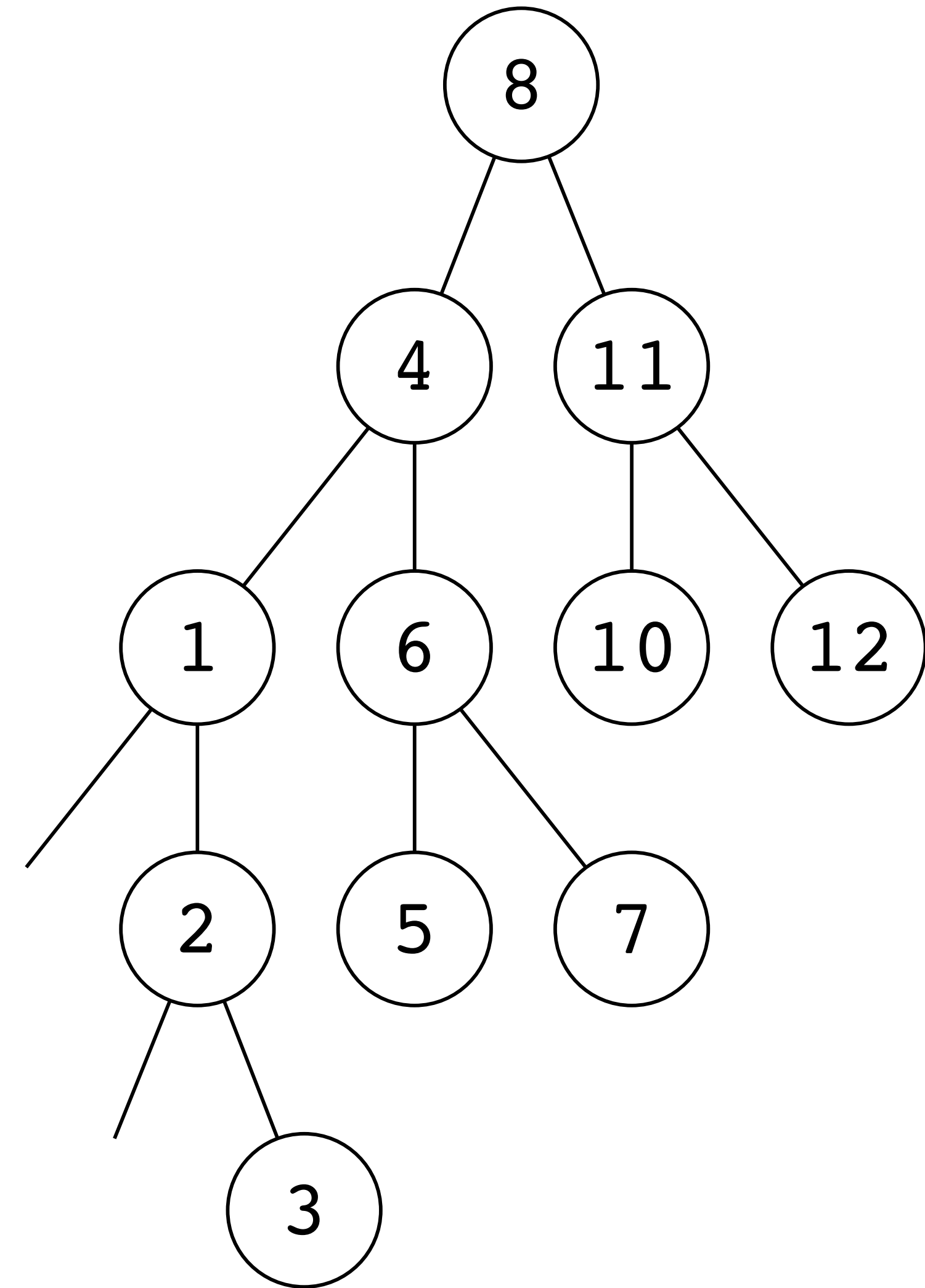


Où est 7 ?

Où est 9 ?



```
fonction chercher (r, k)
  si r est ∅
    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)
```

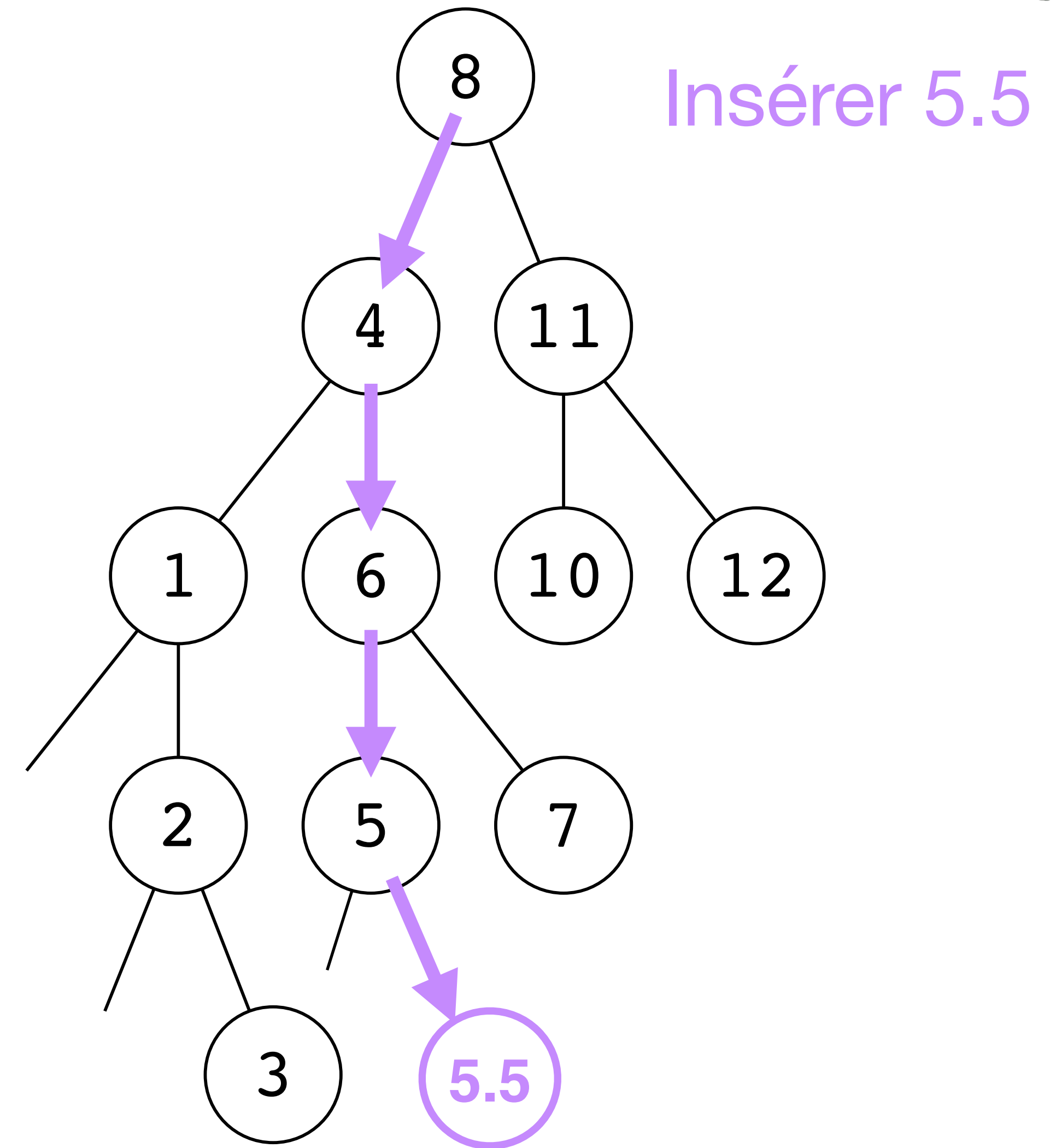




Insertion



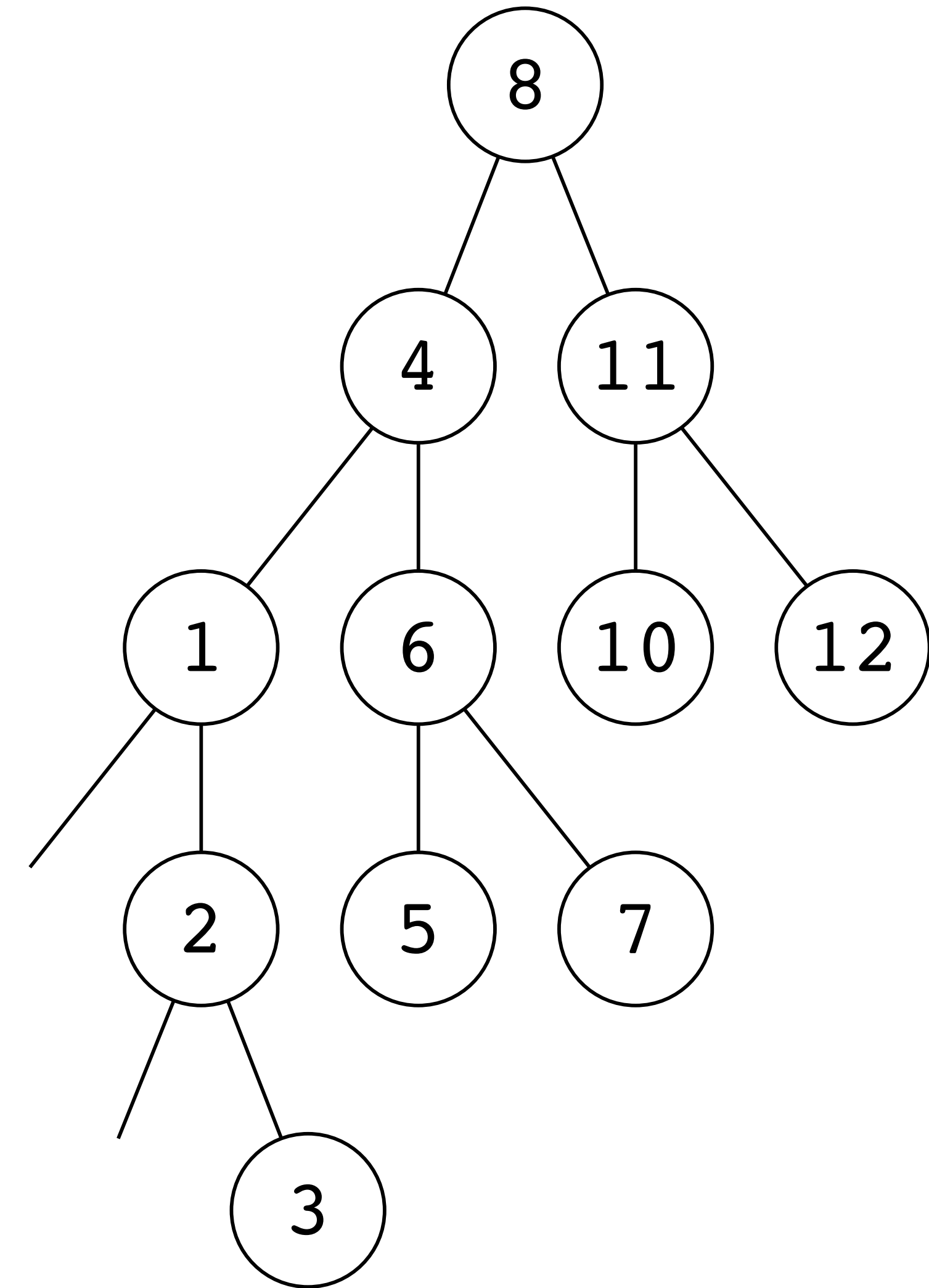
- Chercher l'emplacement où devrait se trouver la clé
- Si elle est absente, remplacer le noeud vide par un nouveau noeud
- Si elle est déjà présente, cela dépend...



Insertion



```
fonction insérer (ref r, k)
  si r est ∅
    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

Insertion



```
fonction insérer (ref r, k)
    si r est ∅
        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)
```

```
fonction chercher (r, k)
    si r est ∅
        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)
```

Notons que la racine est passée par référence

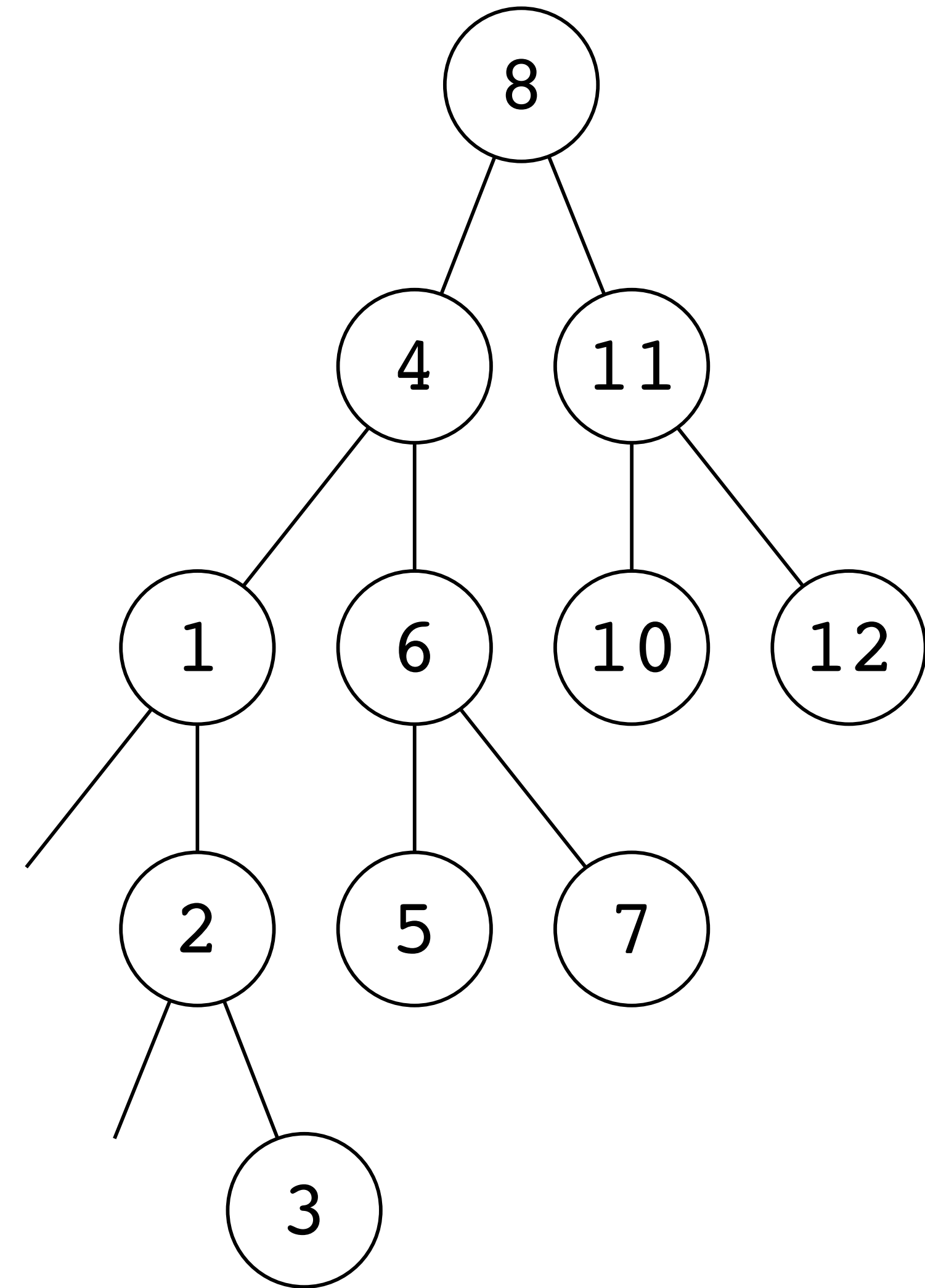


Parcours

Parcours ordonné



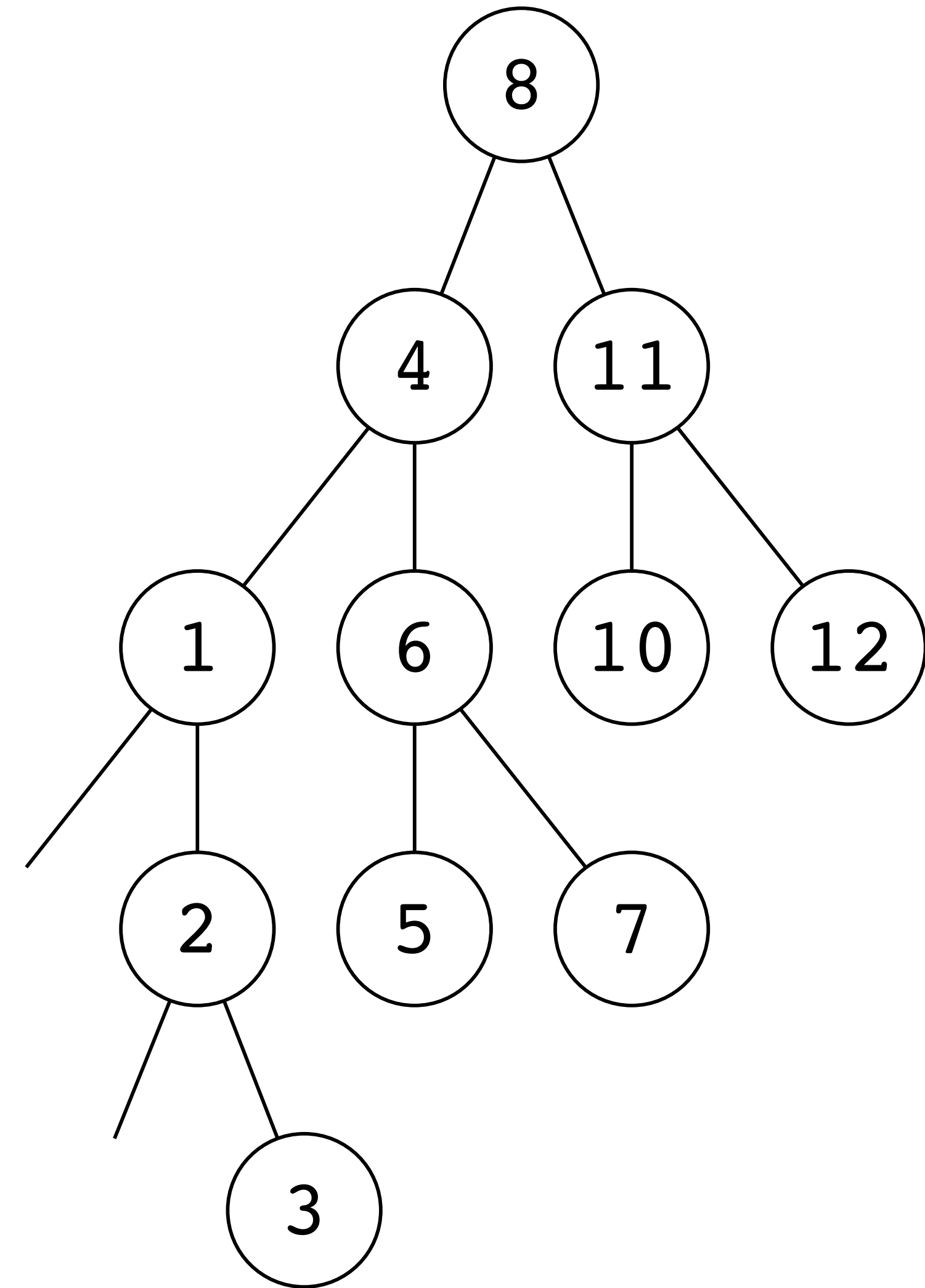
- Parcours par clé croissante



Parcours ordonné



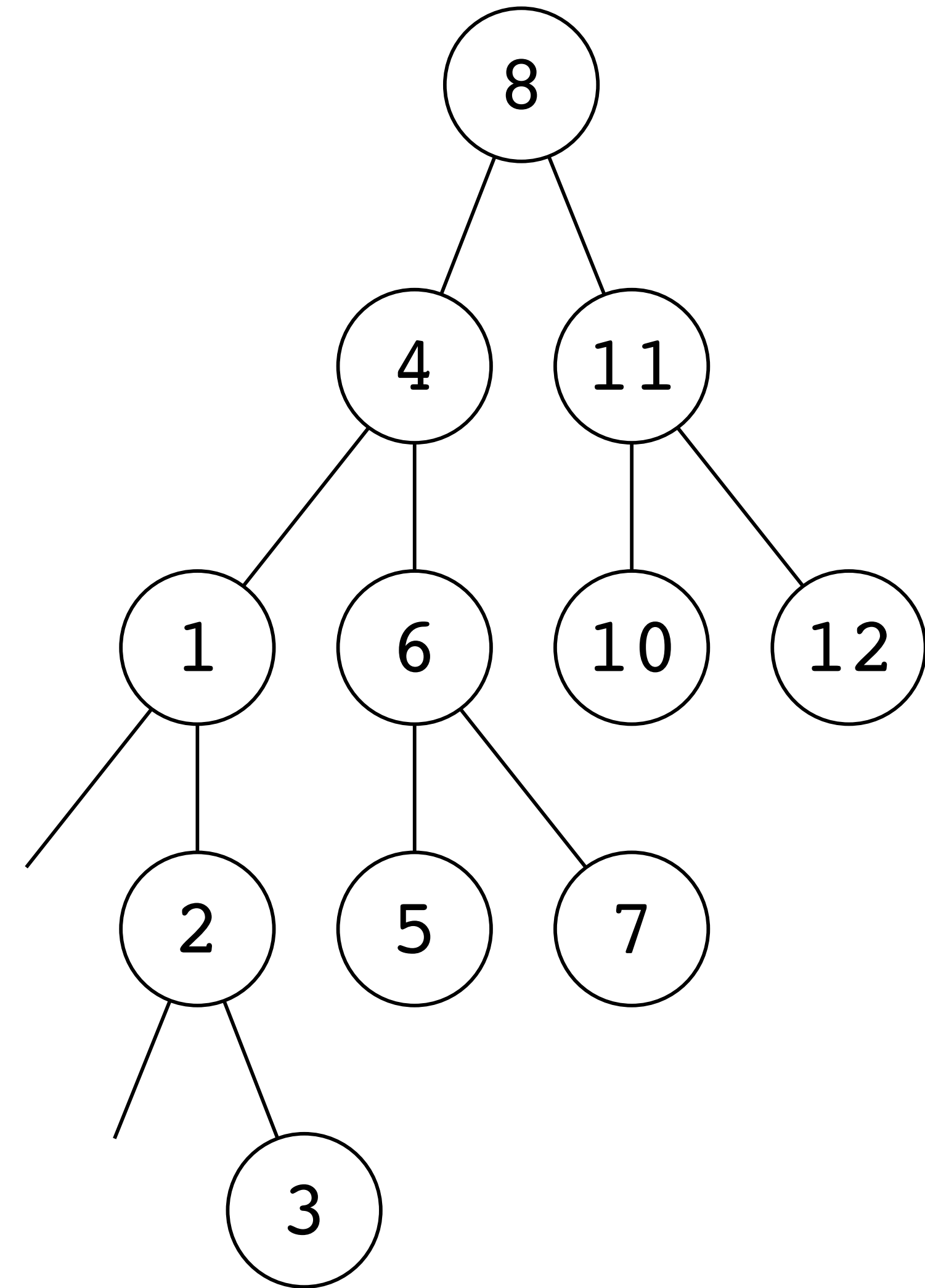
- Parcours par clé croissante
- Sous-arbre gauche avant la racine
- Sous-arbre droit après la racine
- Parcours symétrique !



Parcours ordonné



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

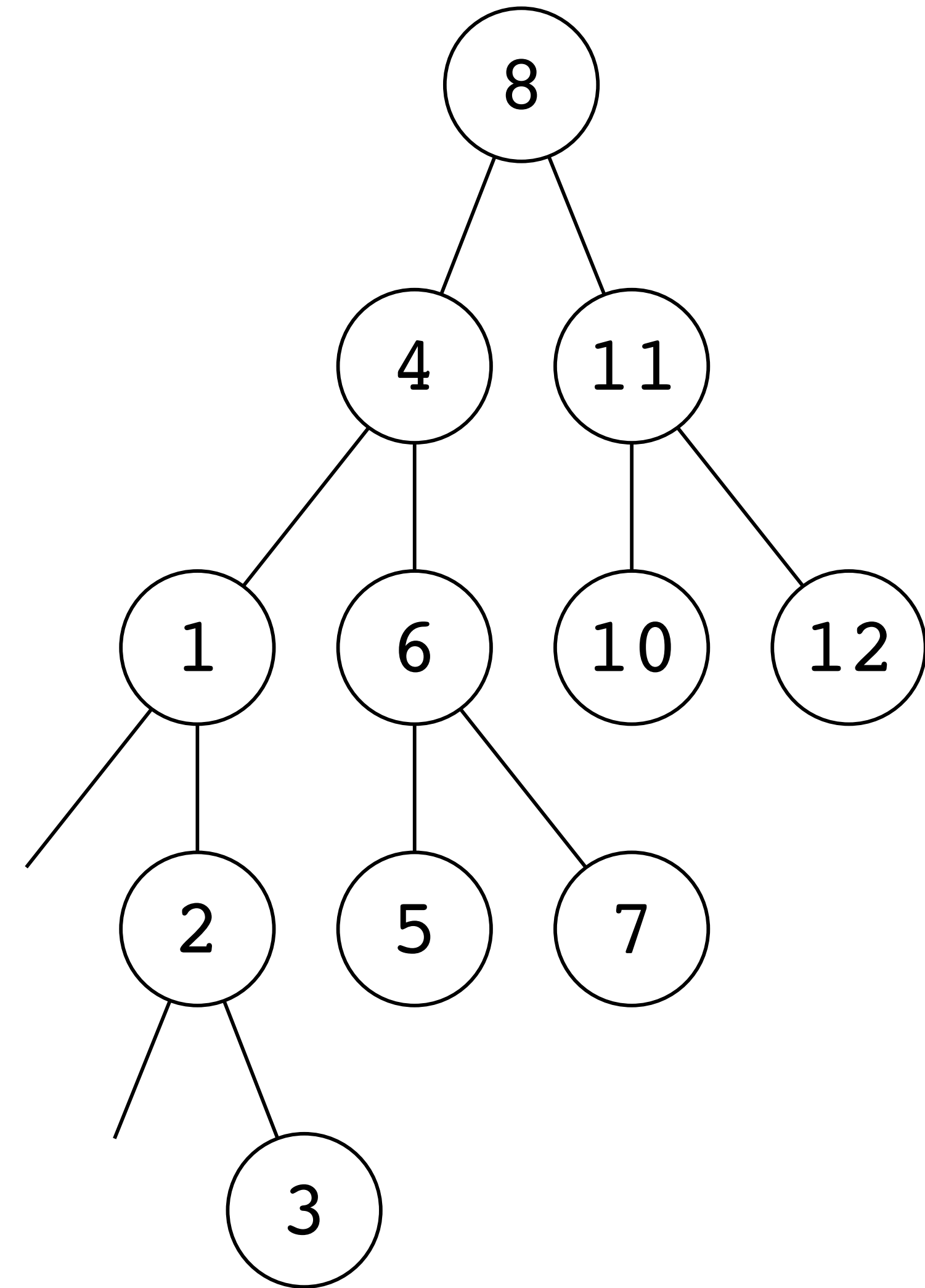


Parcours ordonné



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

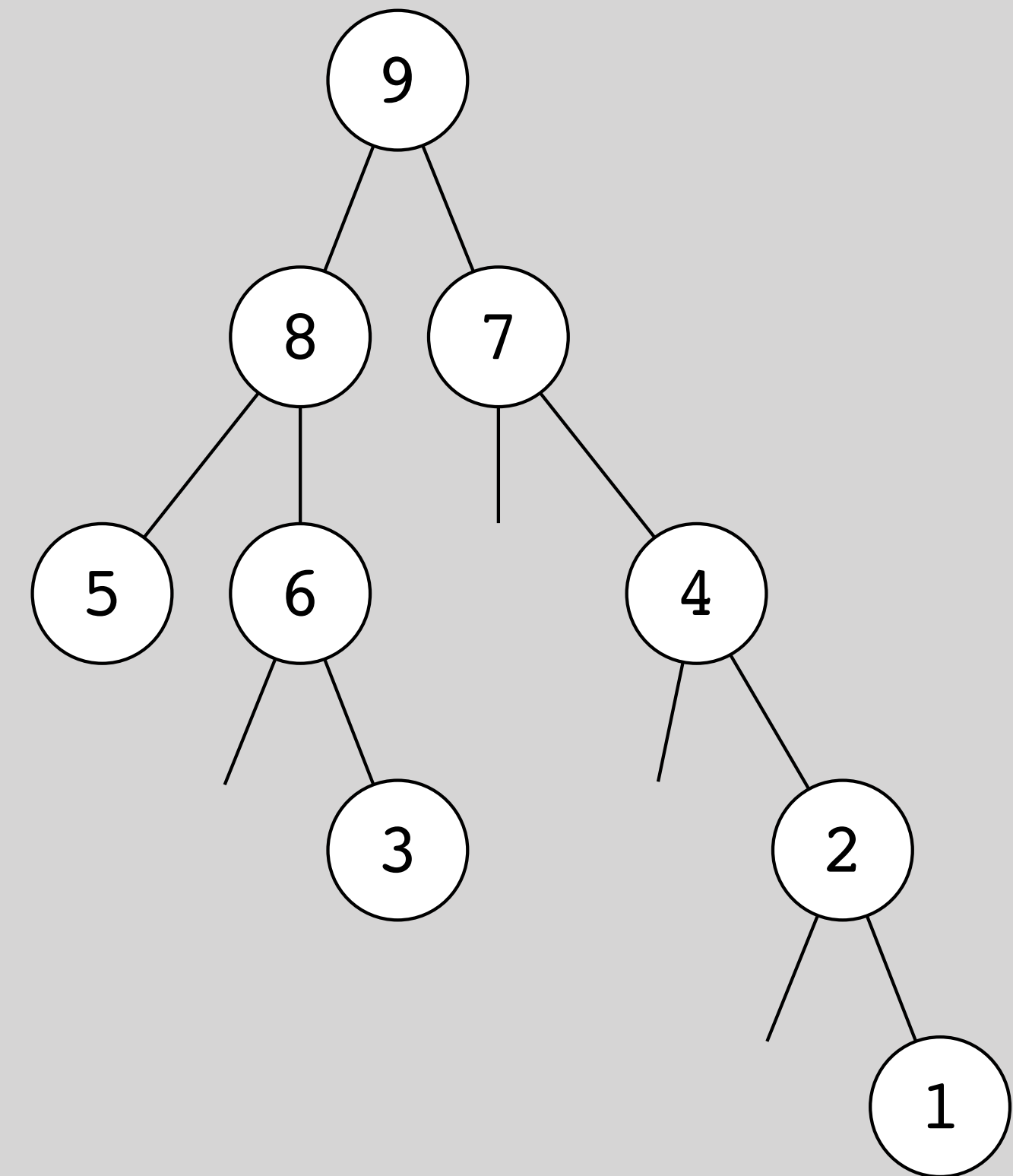
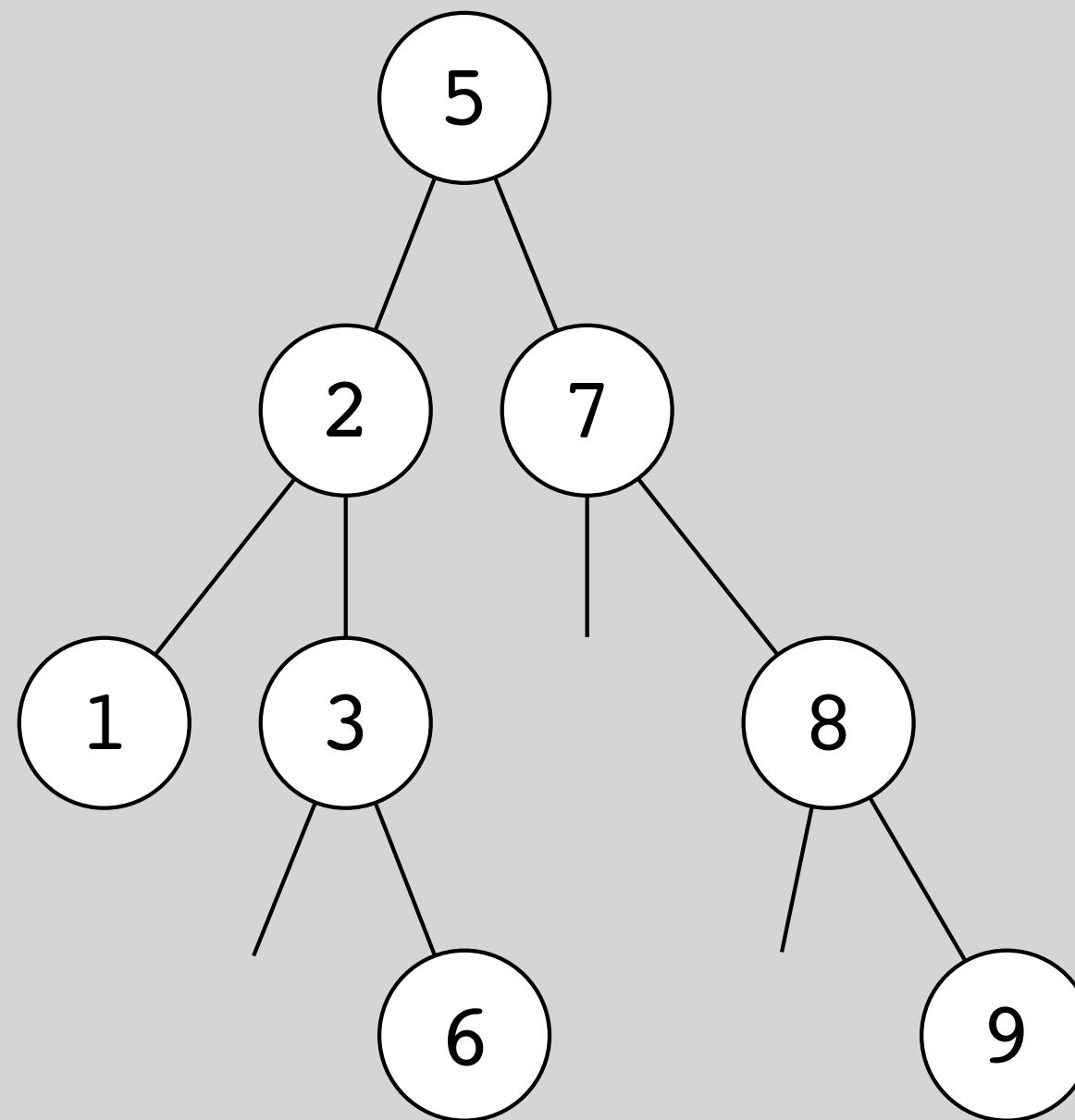
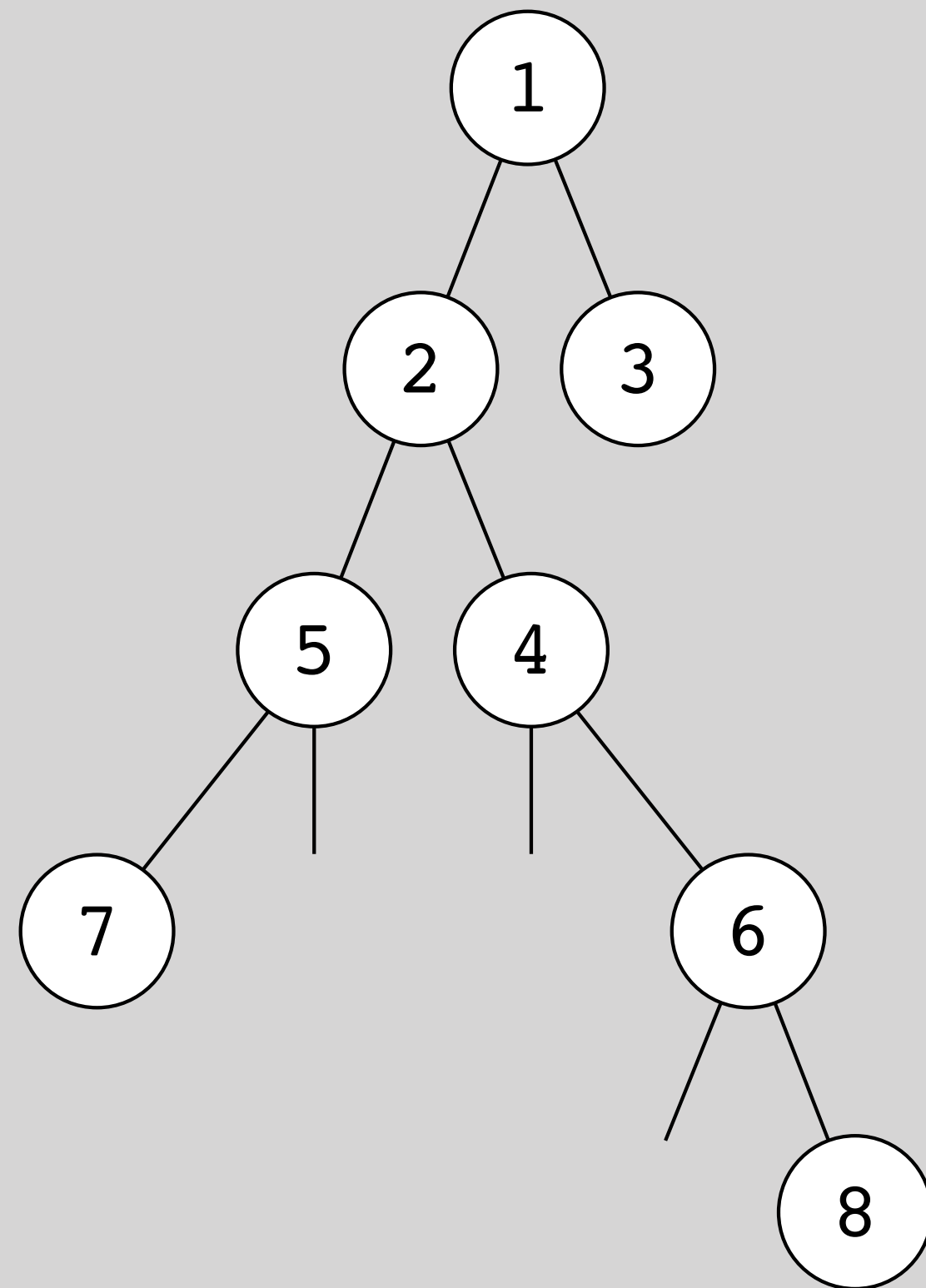
```
fonction décroissant (r, fn)
  si r != ∅
    décroissant(r.droit, fn)
    fn(r)
    décroissant(r.gauche, fn)
```



Exercice 1



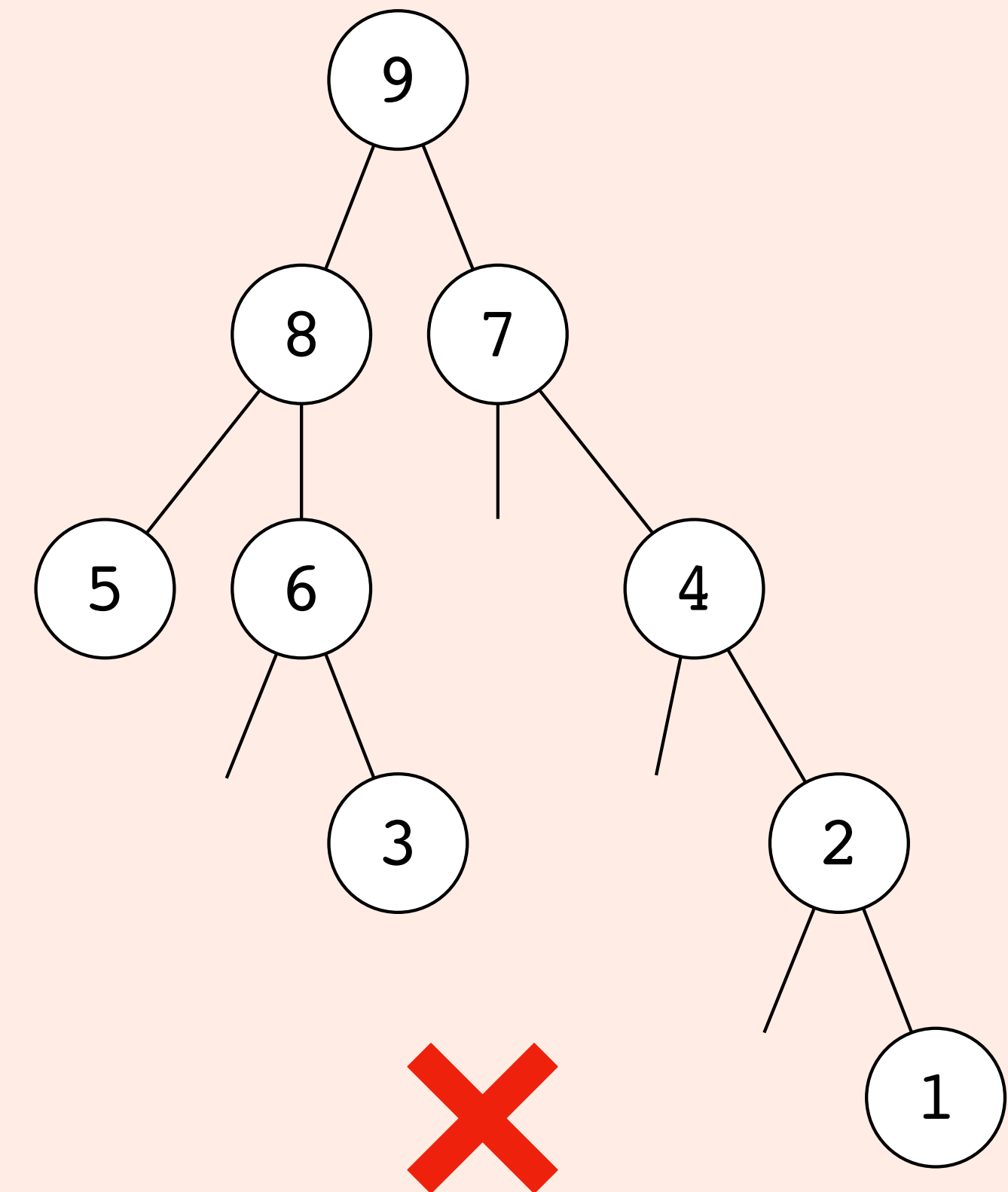
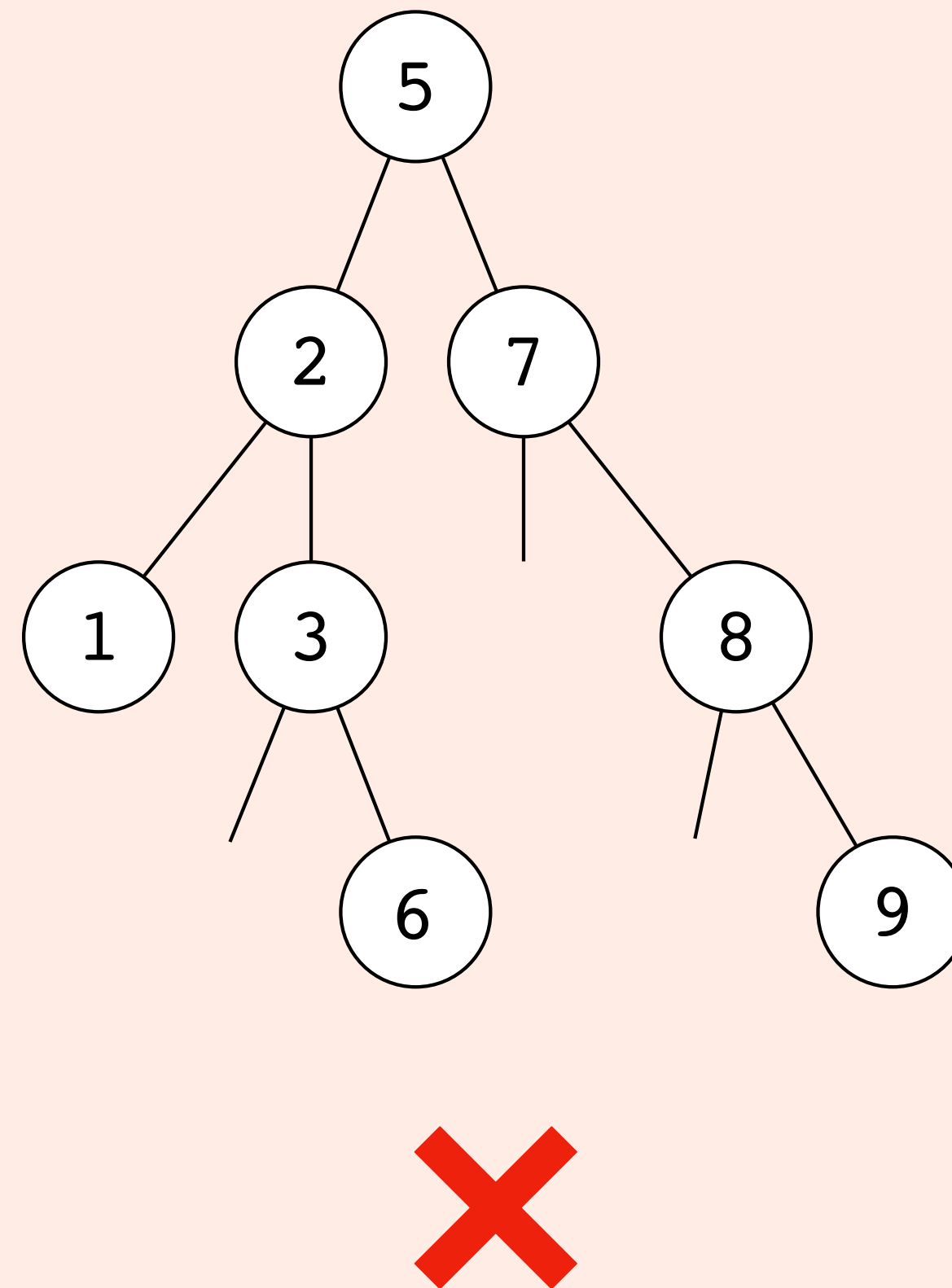
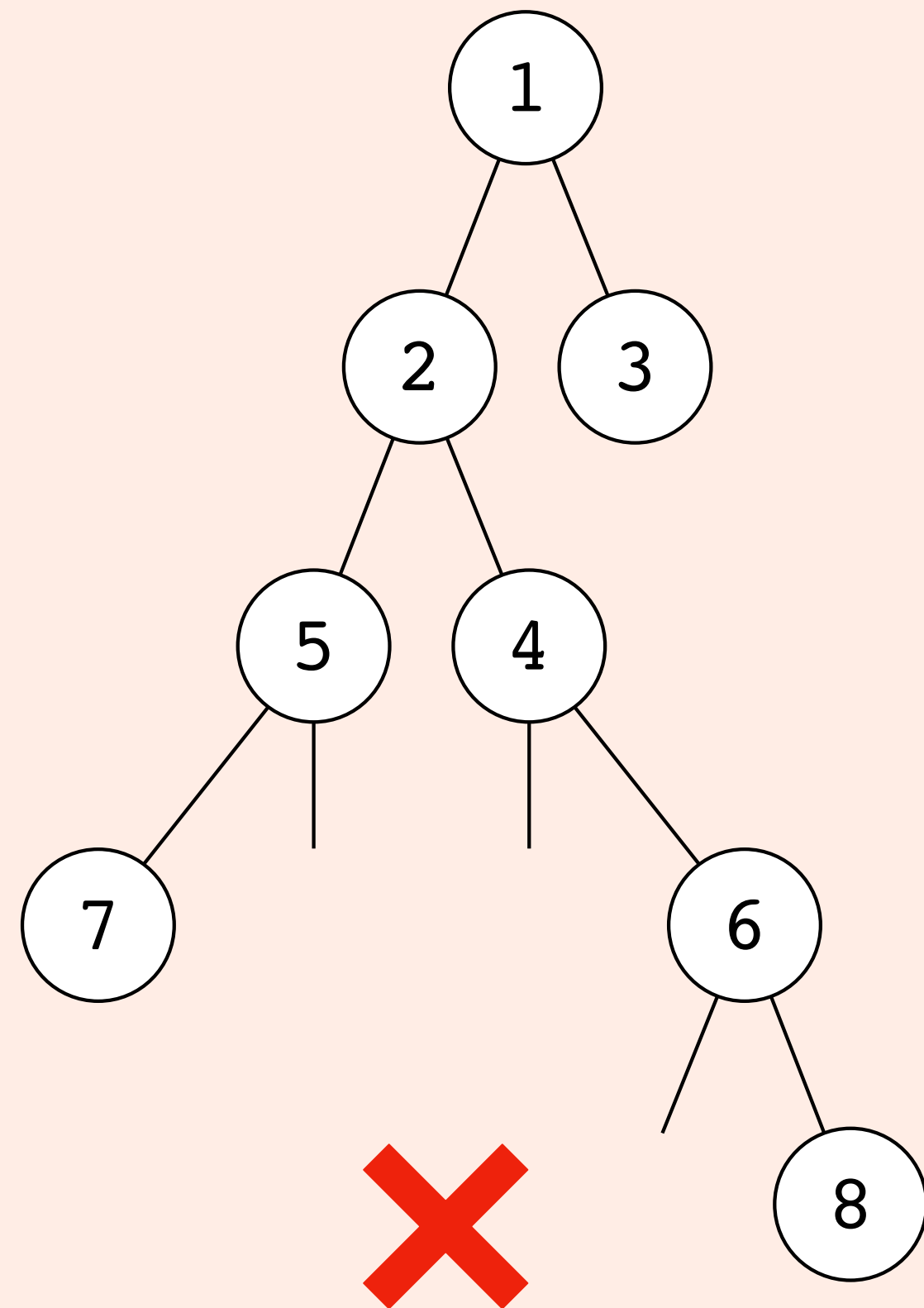
- Parmi les arbres suivants, le(s)quel(s) respecte(nt) la condition d'arbre binaire de recherche ?



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

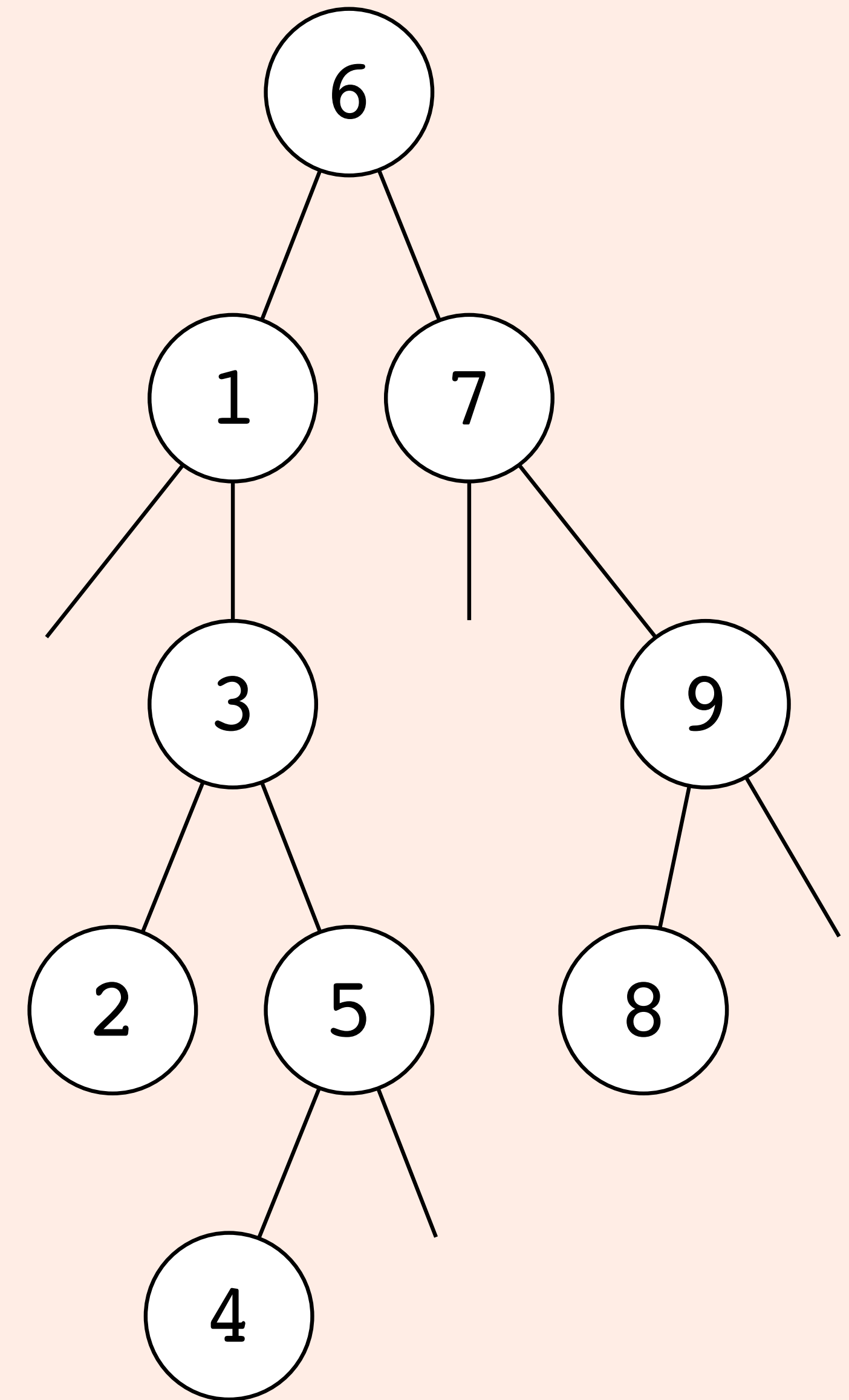
Solution 2



- 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 ...



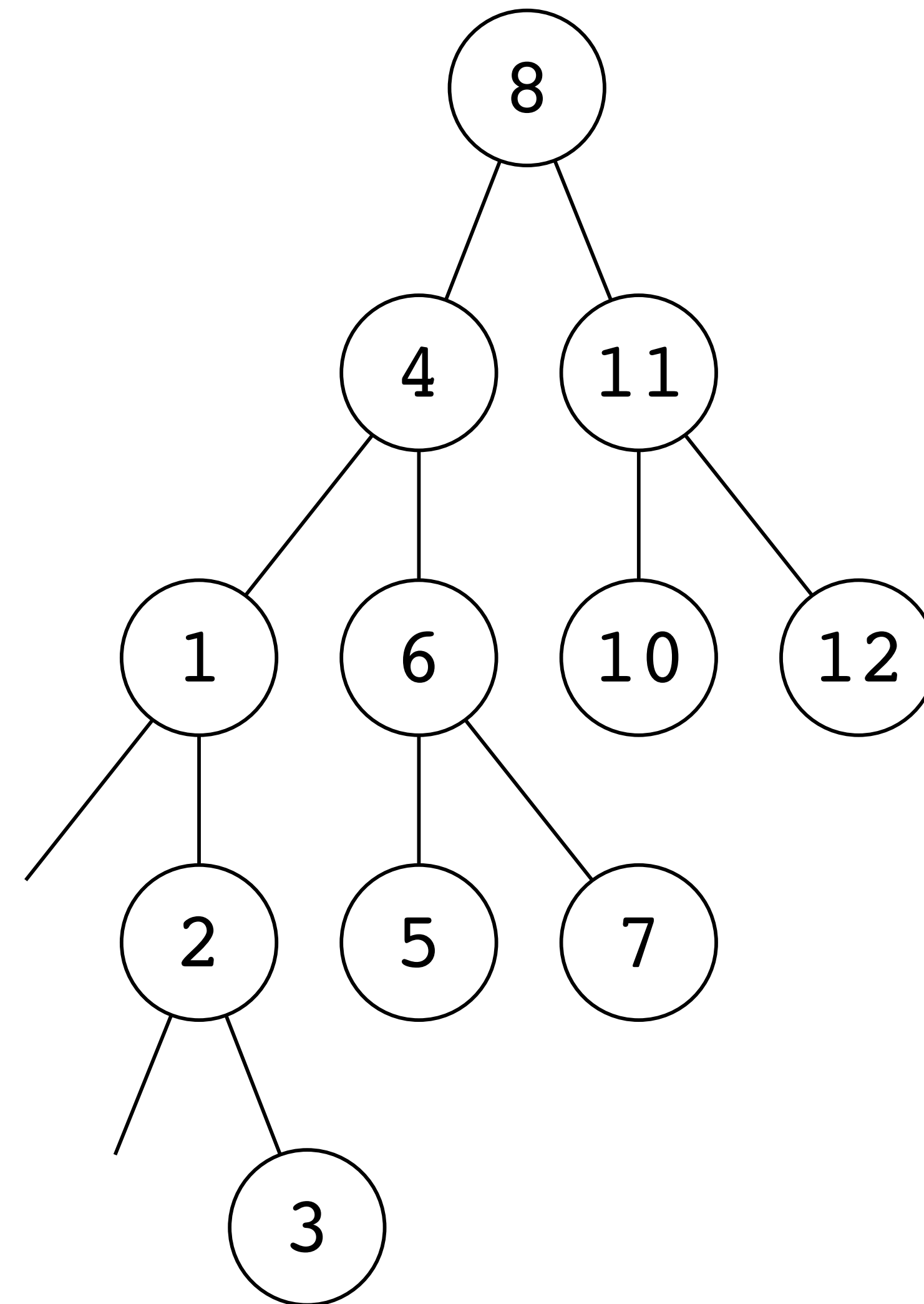
10. Suppression d'éléments





Recherche du minimum

Recherche du minimum

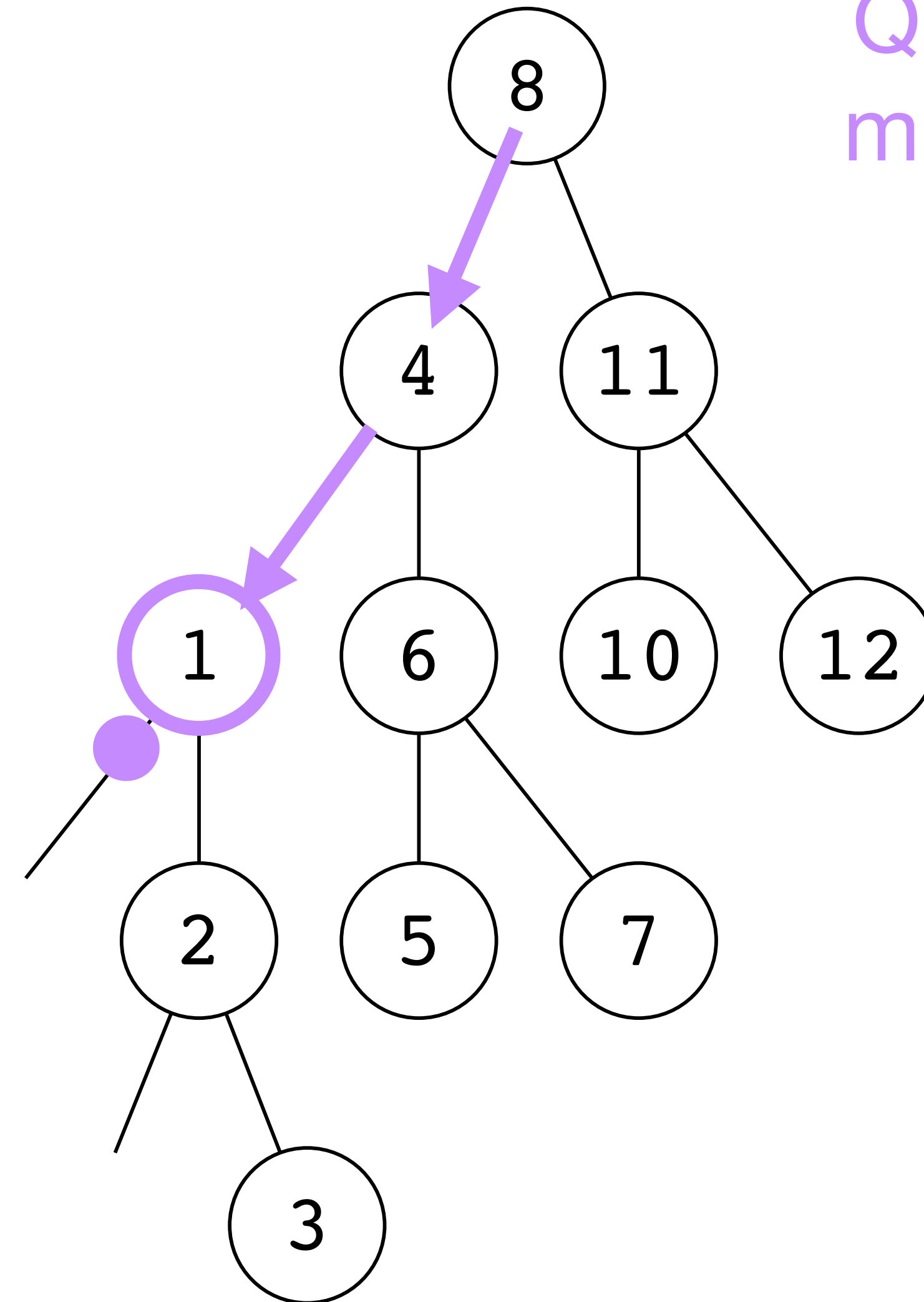


Recherche du minimum



Quel est le 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



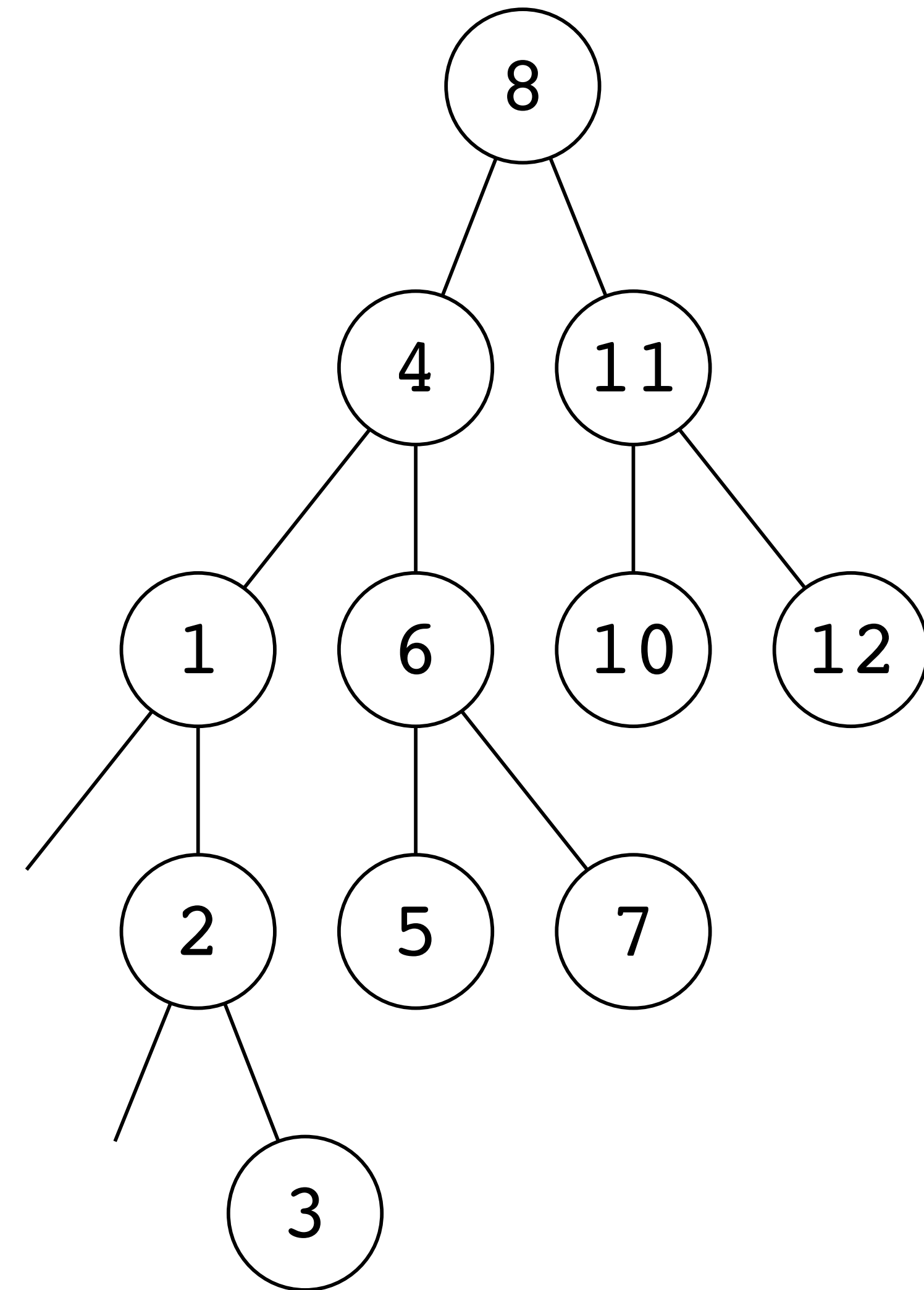


- Récursivement

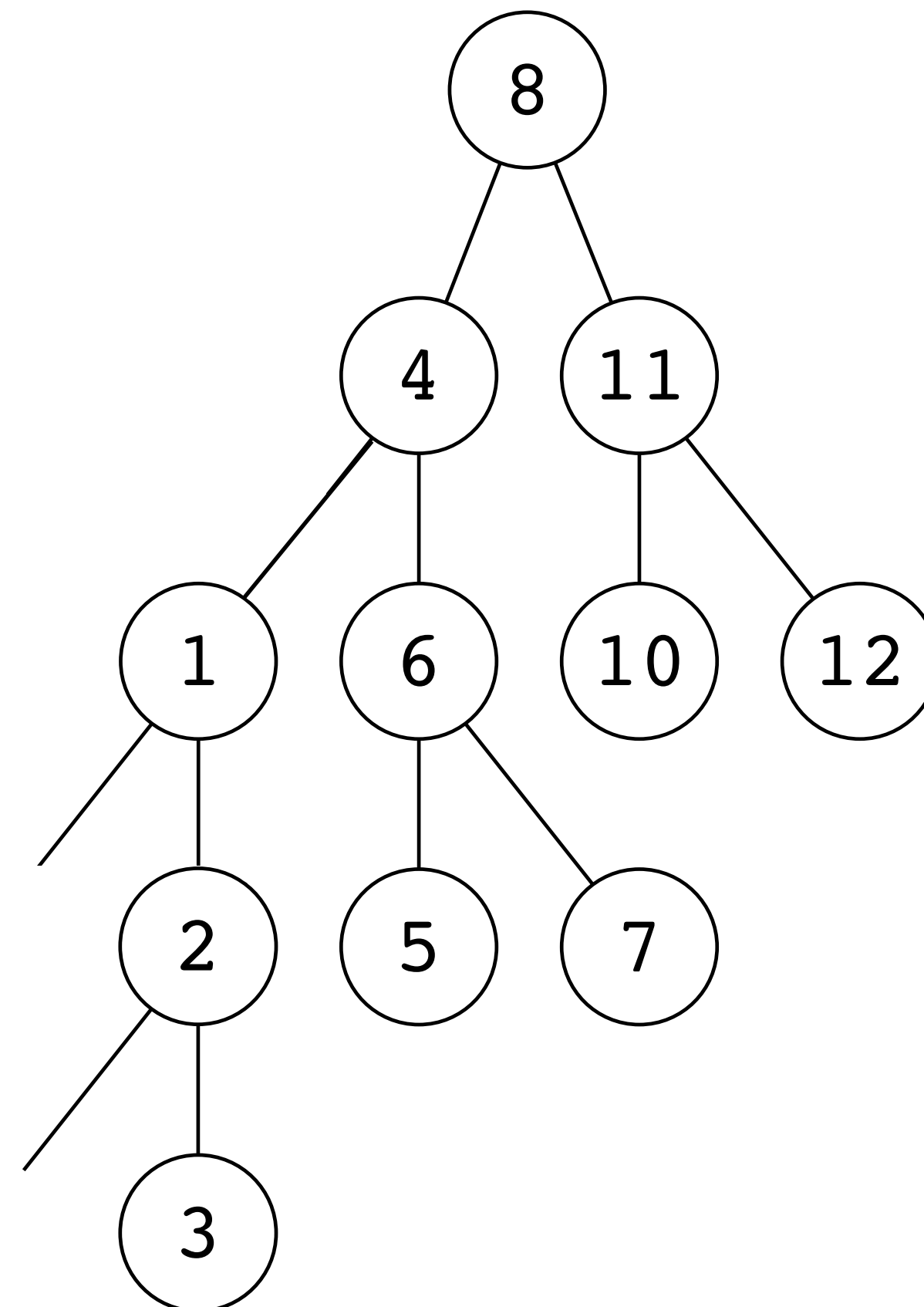
```
fonction min (r)
  si r.gauche != ∅
    retourner min(r.gauche)
  sinon
    retourner r
```

- Itérativement

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



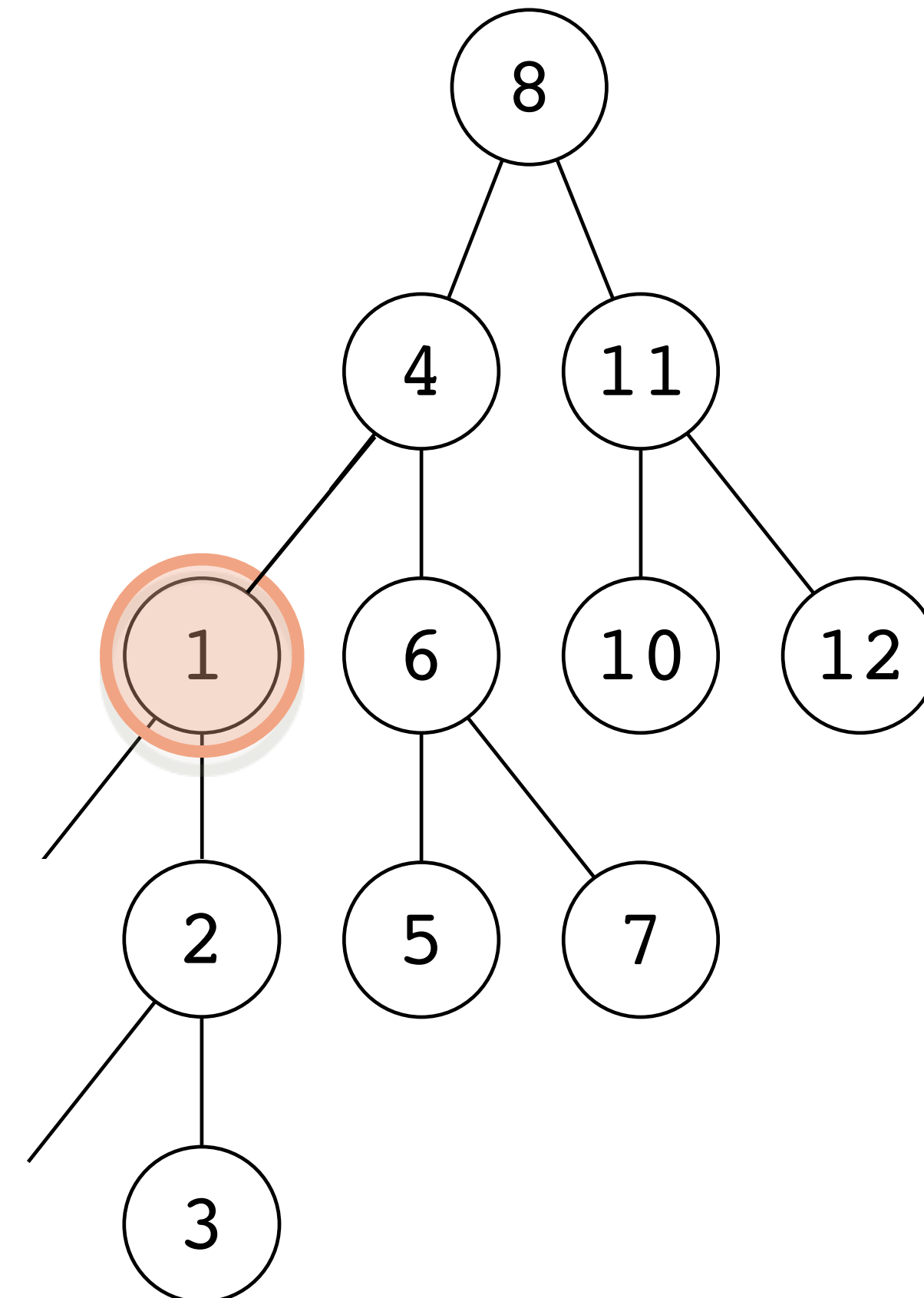
Suppression du minimum



Suppression du minimum



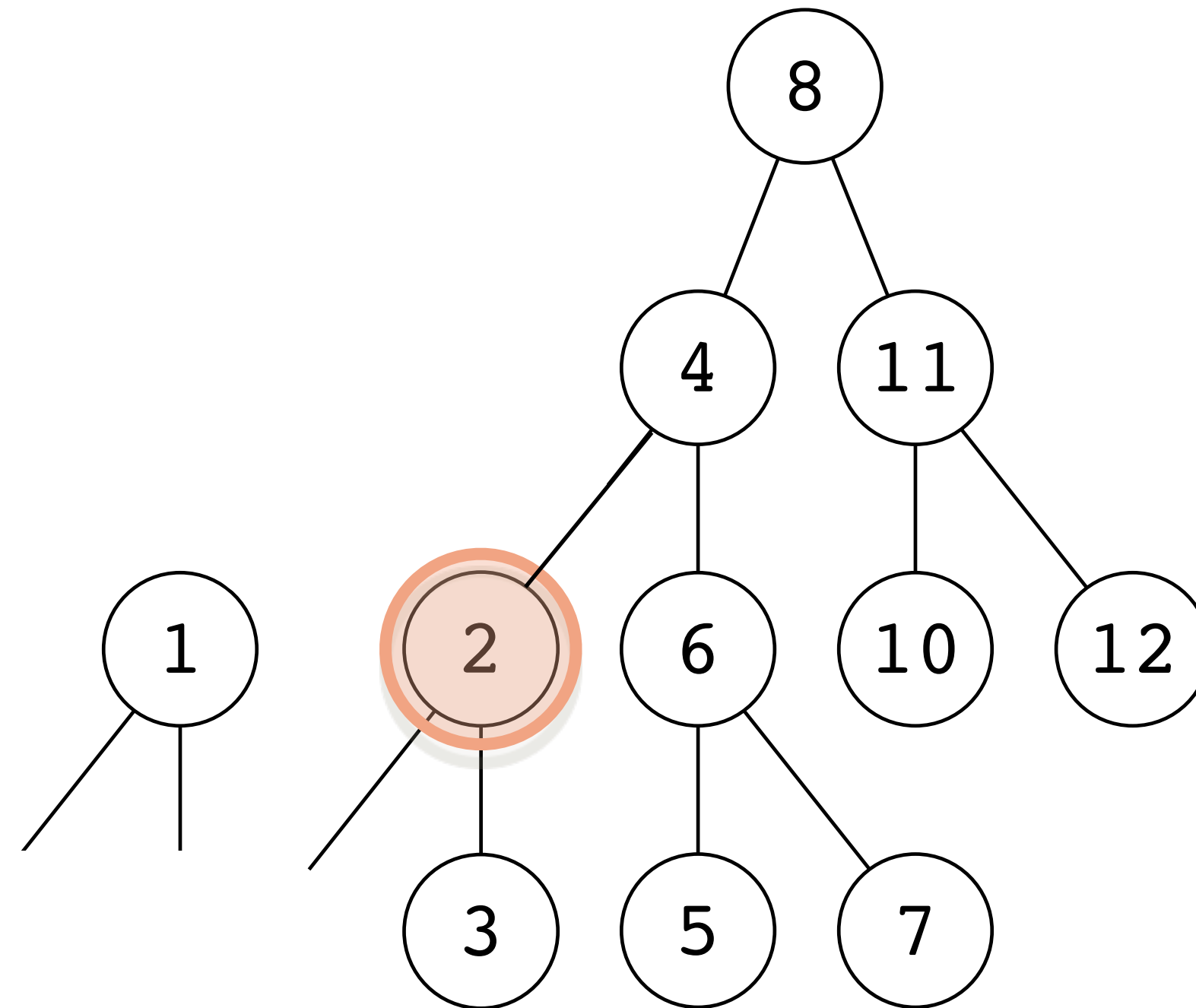
- Trouver le minimum



Suppression du minimum



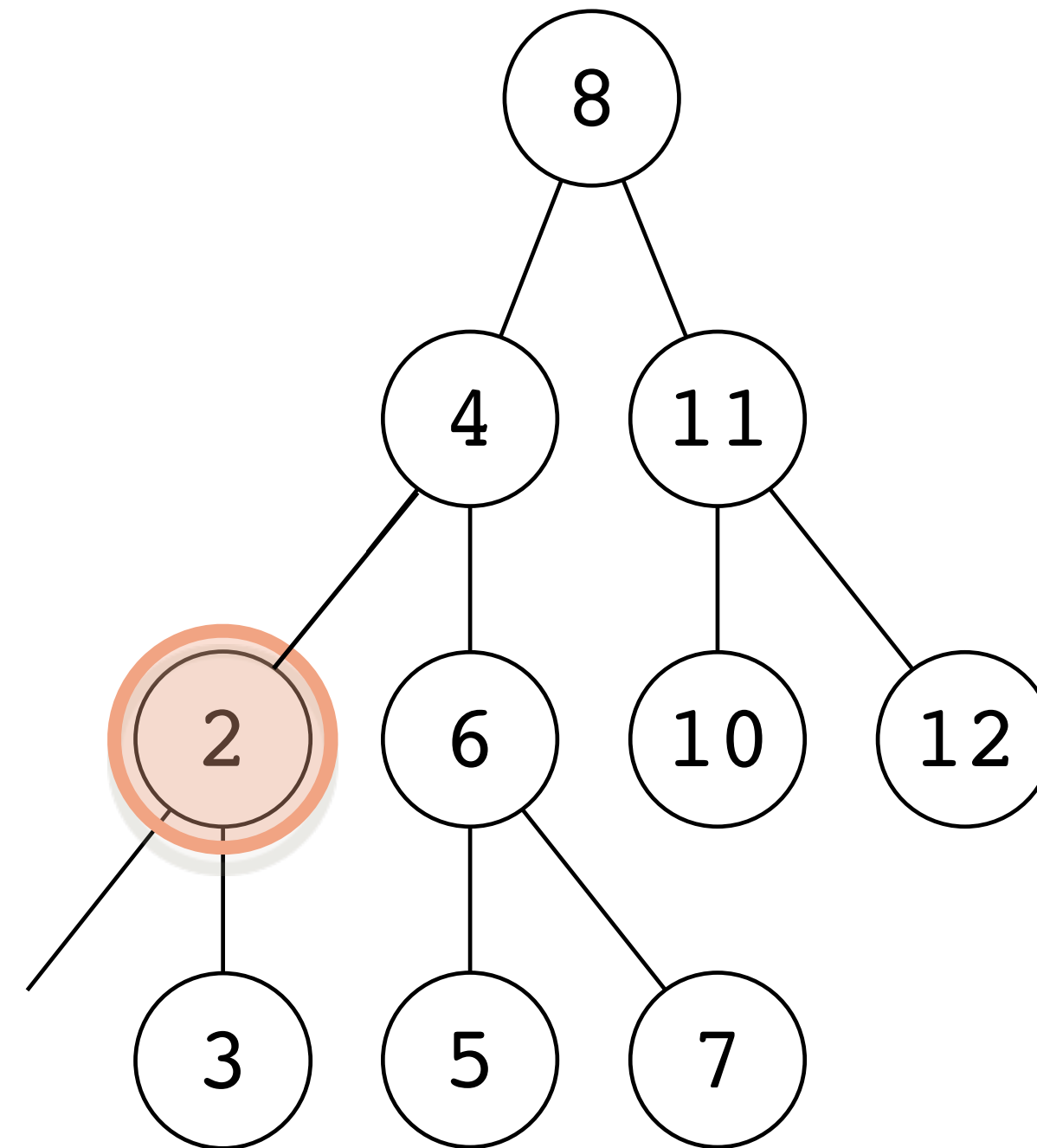
- Trouver le minimum
- Le remplacer par son enfant de droite



Suppression du minimum



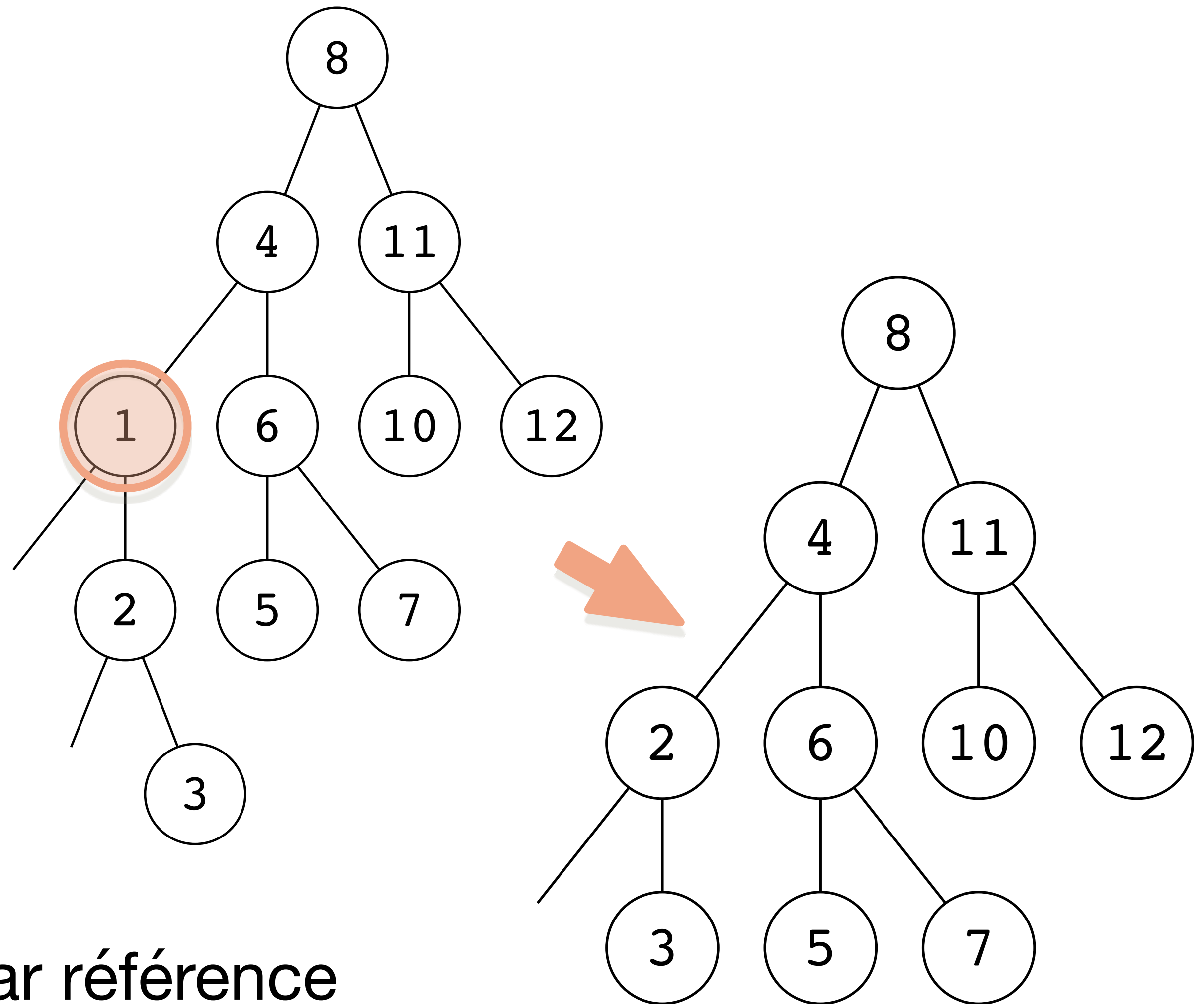
- Trouver le minimum
- Le remplacer par son enfant de droite
- Effacer le noeud de la mémoire



Suppression du minimum



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



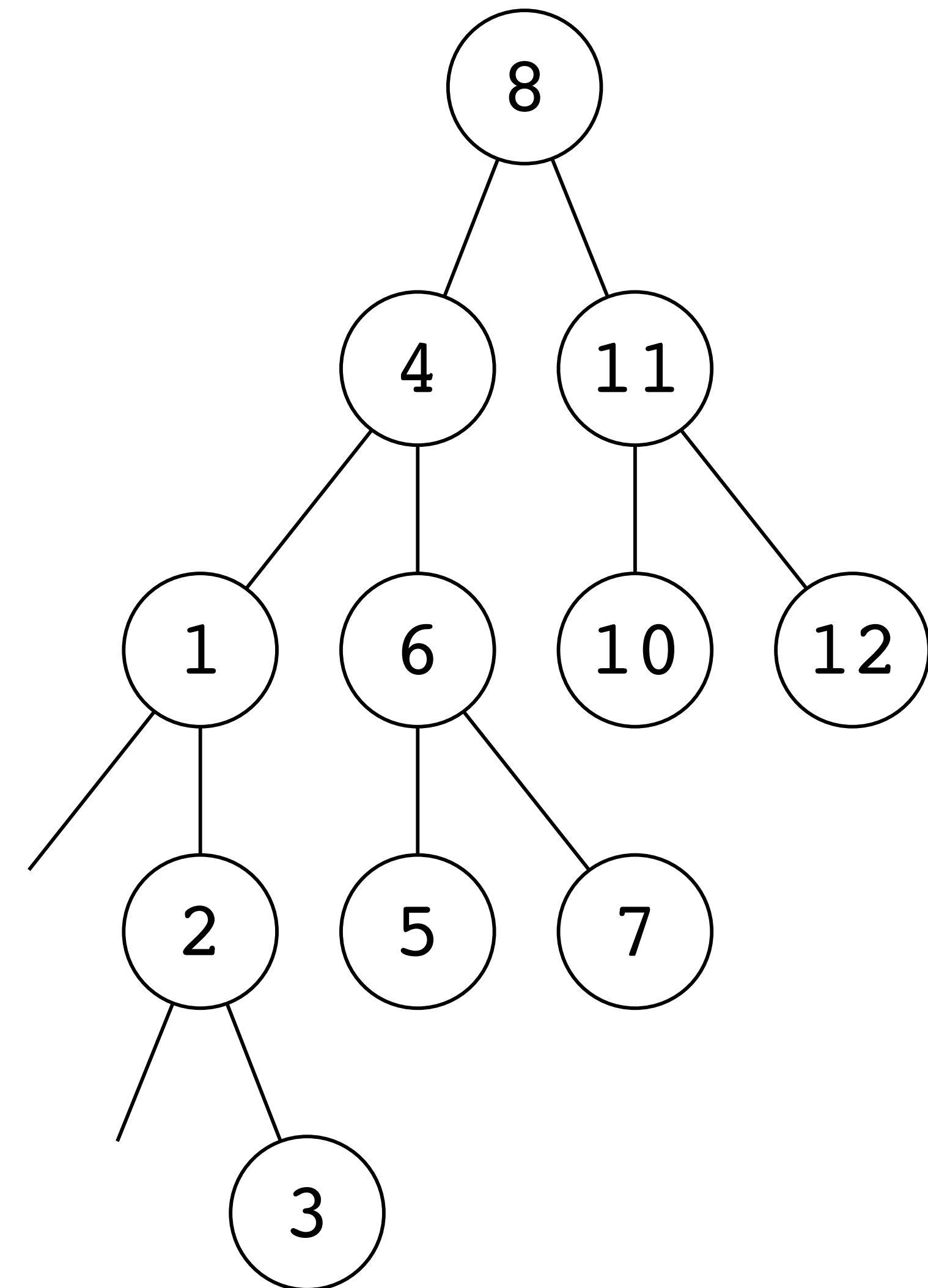
- Notons que la racine est passée par référence

Suppression du maximum



Exactement symétrique à la suppression du minimum.

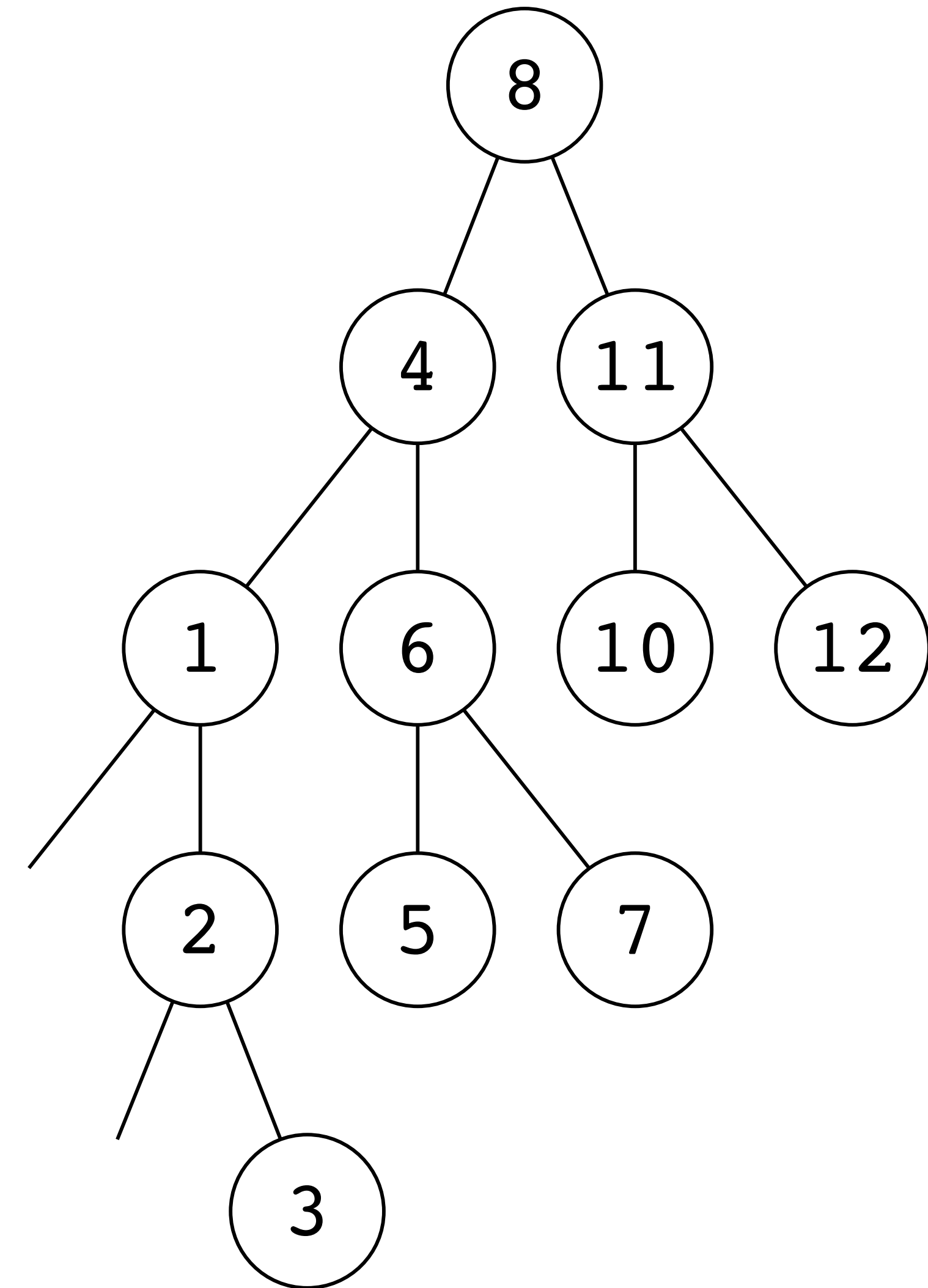
Suppression de la clé k



Suppression de la clé k



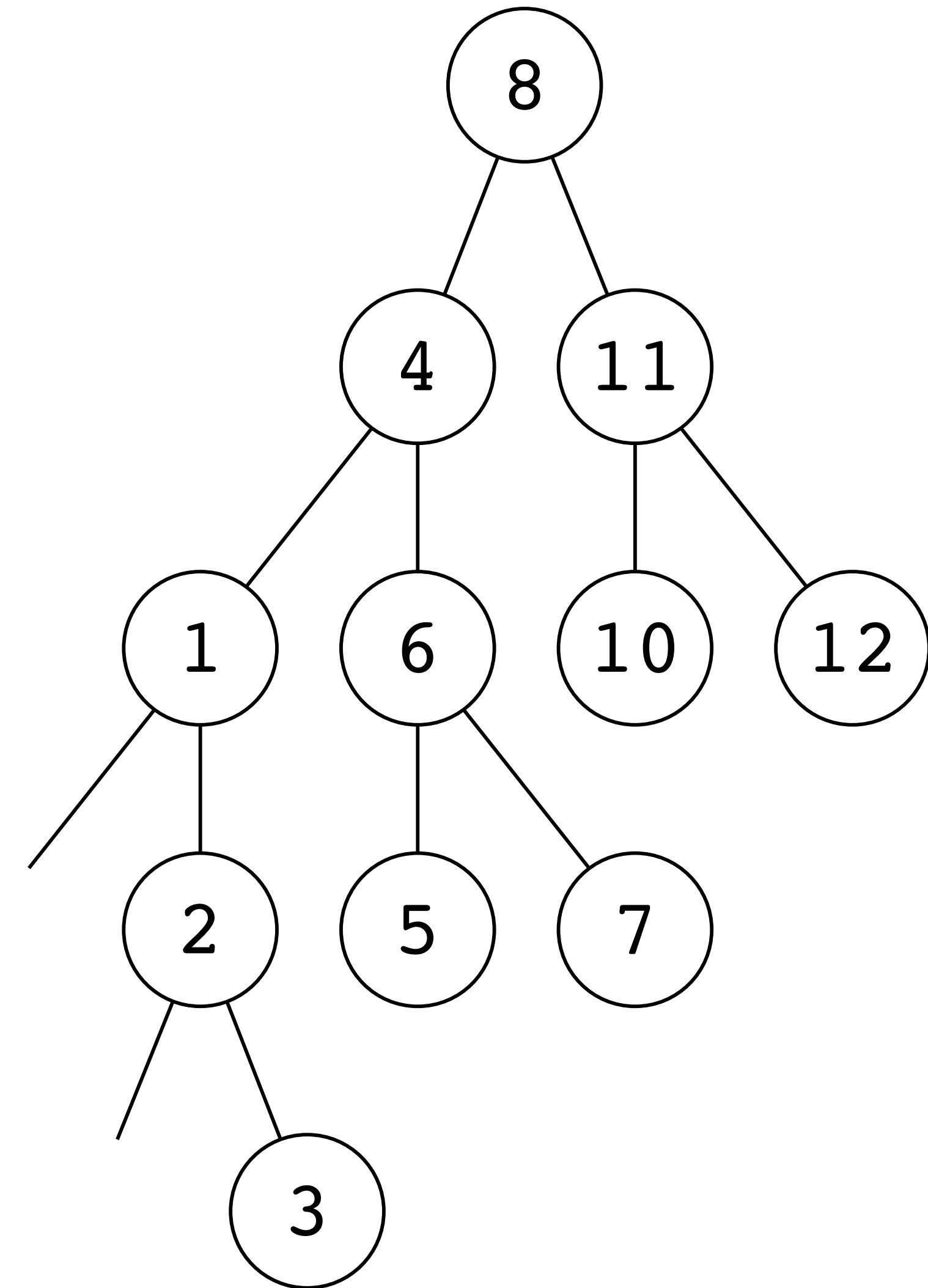
- Trouver le noeud de clé k
- Oter le noeud de l'arbre
- Raccrocher ses enfants à son parent
- Effacer le noeud de la mémoire



Suppression de la clé k



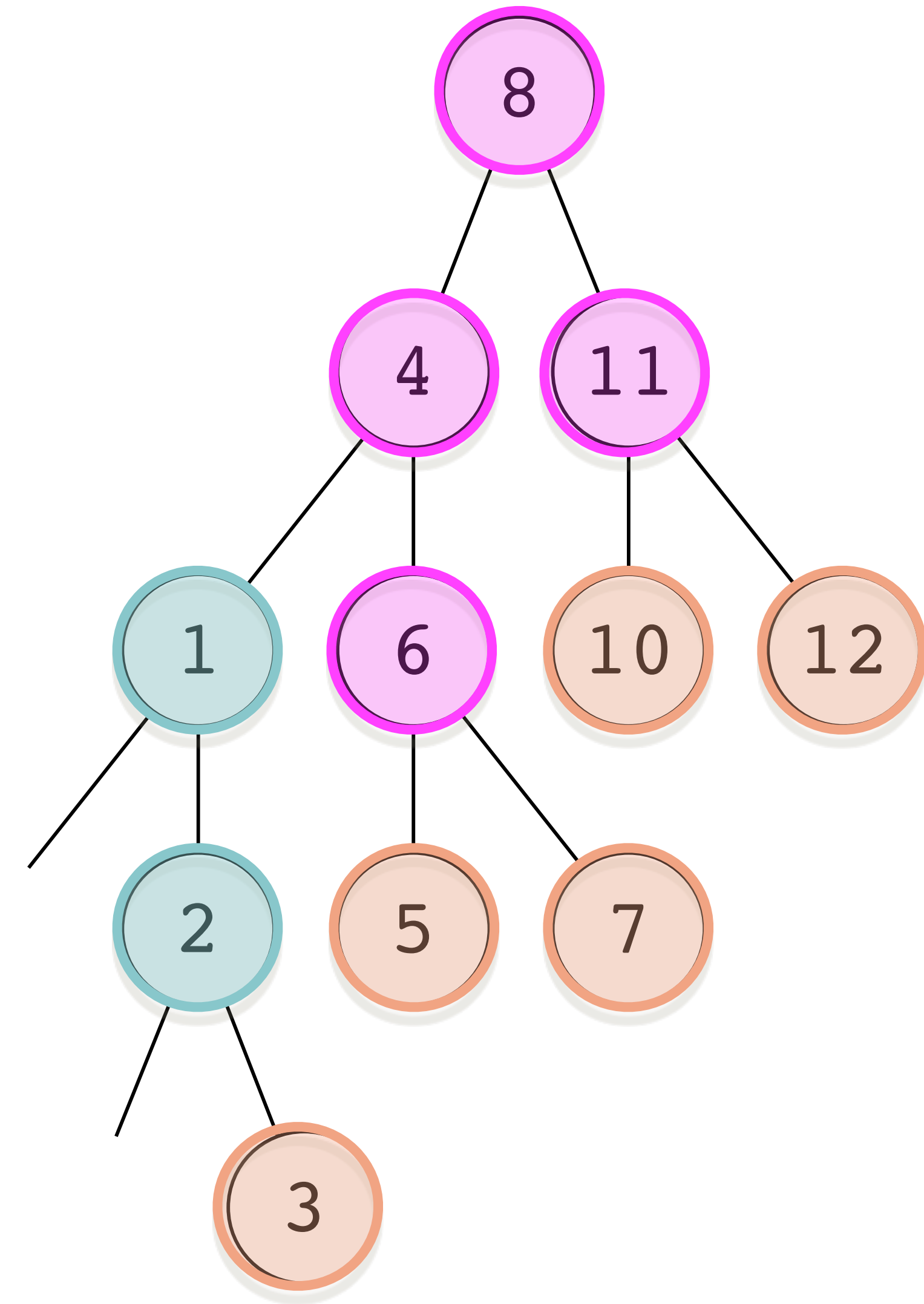
- Trouver le noeud de clé k
- Oter le noeud de l'arbre
- **Raccrocher ses enfants à son parent ???**
- Effacer le noeud de la mémoire



Raccrocher les descendants au parent ...

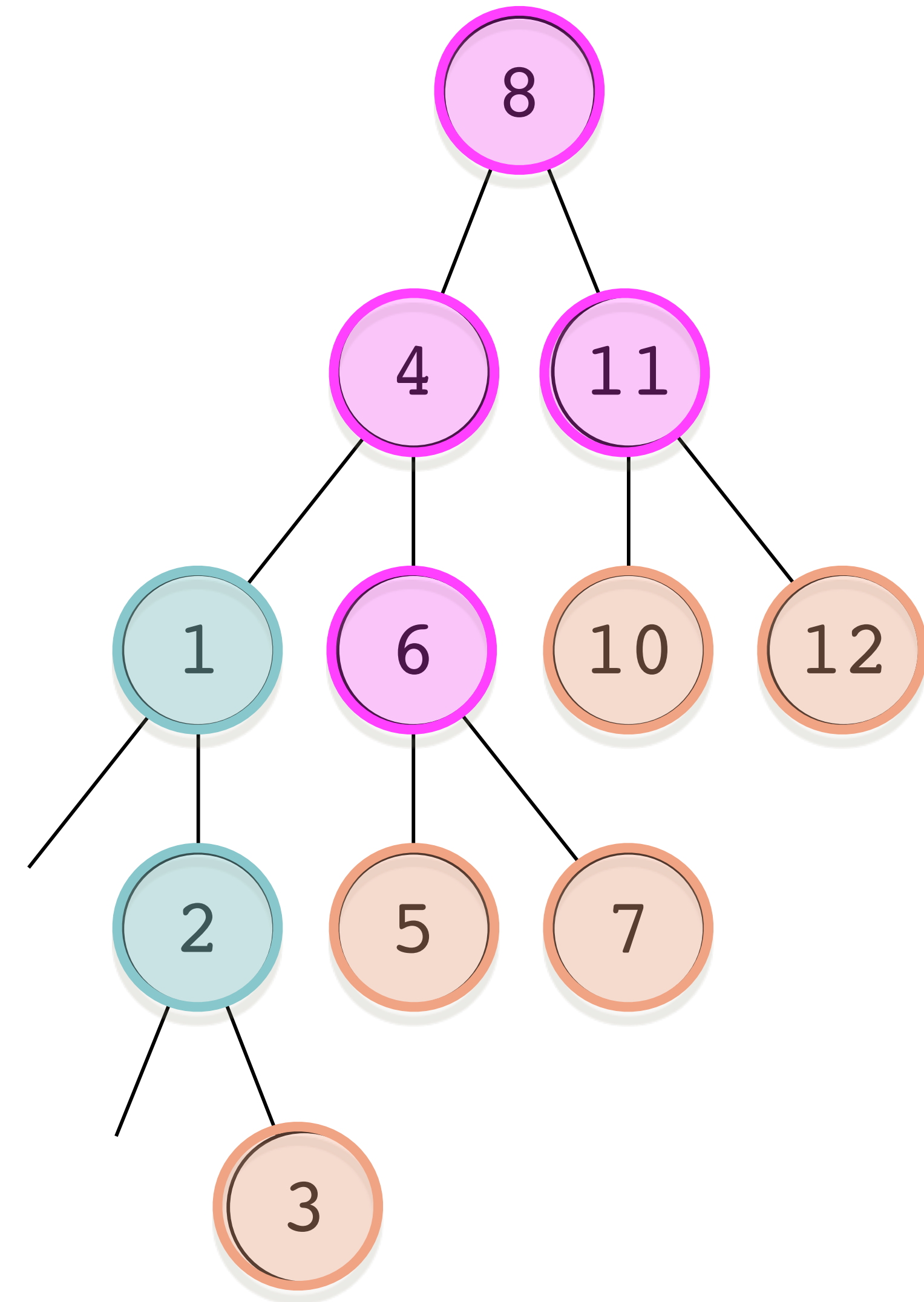


- Feuilles ... rien à faire
- Degré 1 ... comme pour le minimum
- Degré 2 ... trouver une nouvelle racine parmi les descendants.

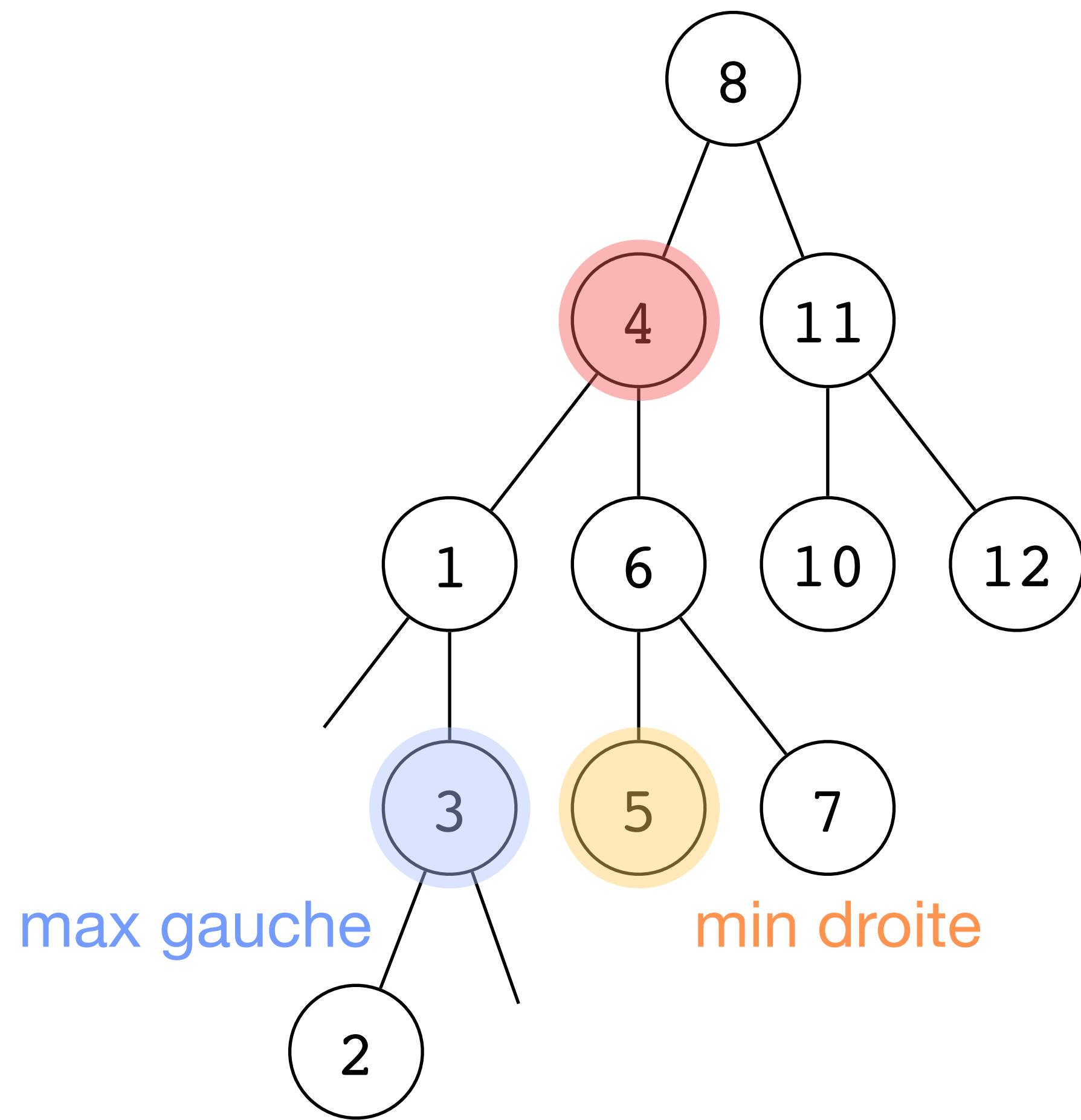




- Feuilles ... rien à faire
- Degré 1 ... comme pour le minimum
- Degré 2 ... trouver une nouvelle racine parmi les descendants.
(Technique de Thomas N. Hibbard)

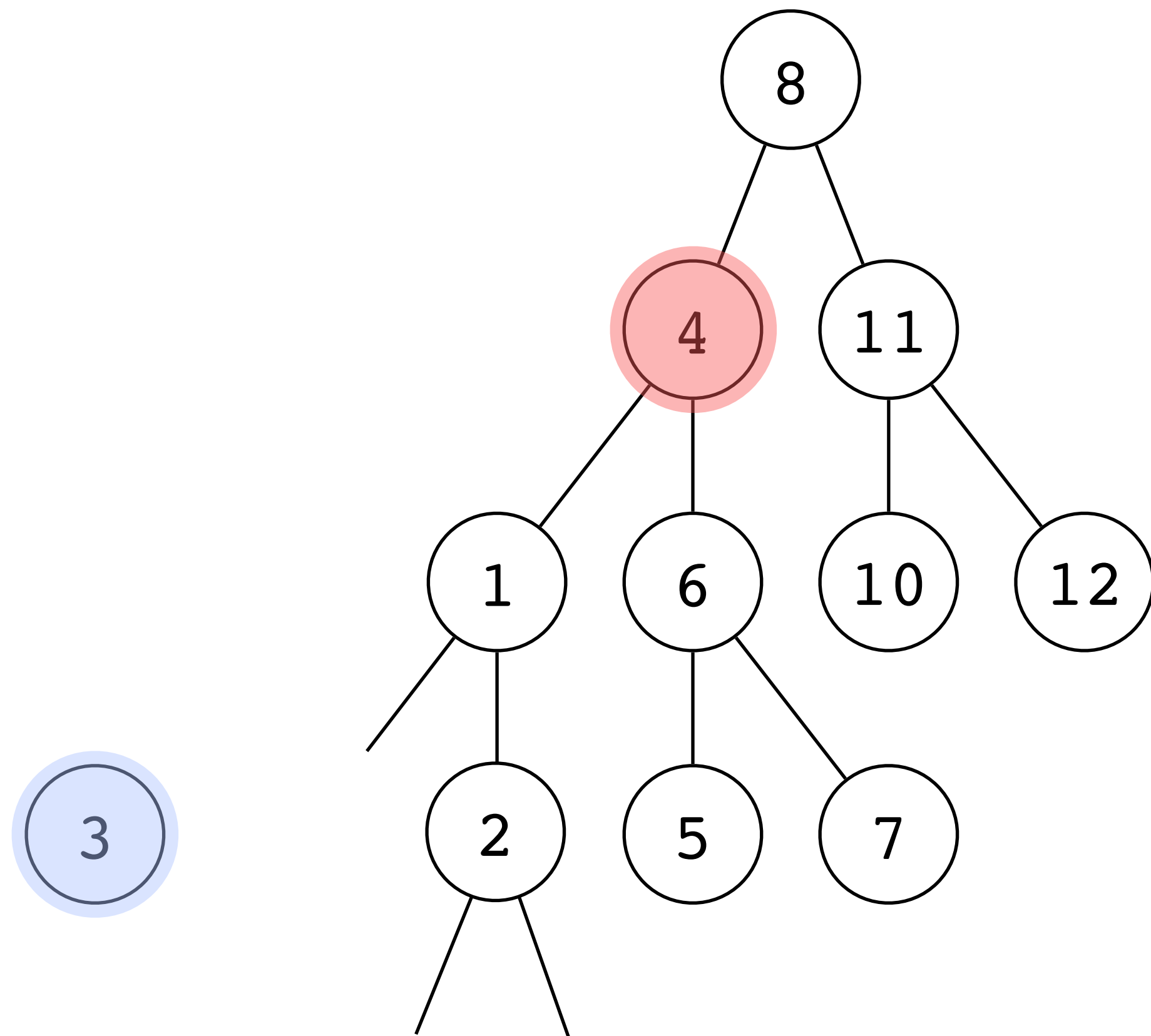


Exemple : suppression de la clé 4





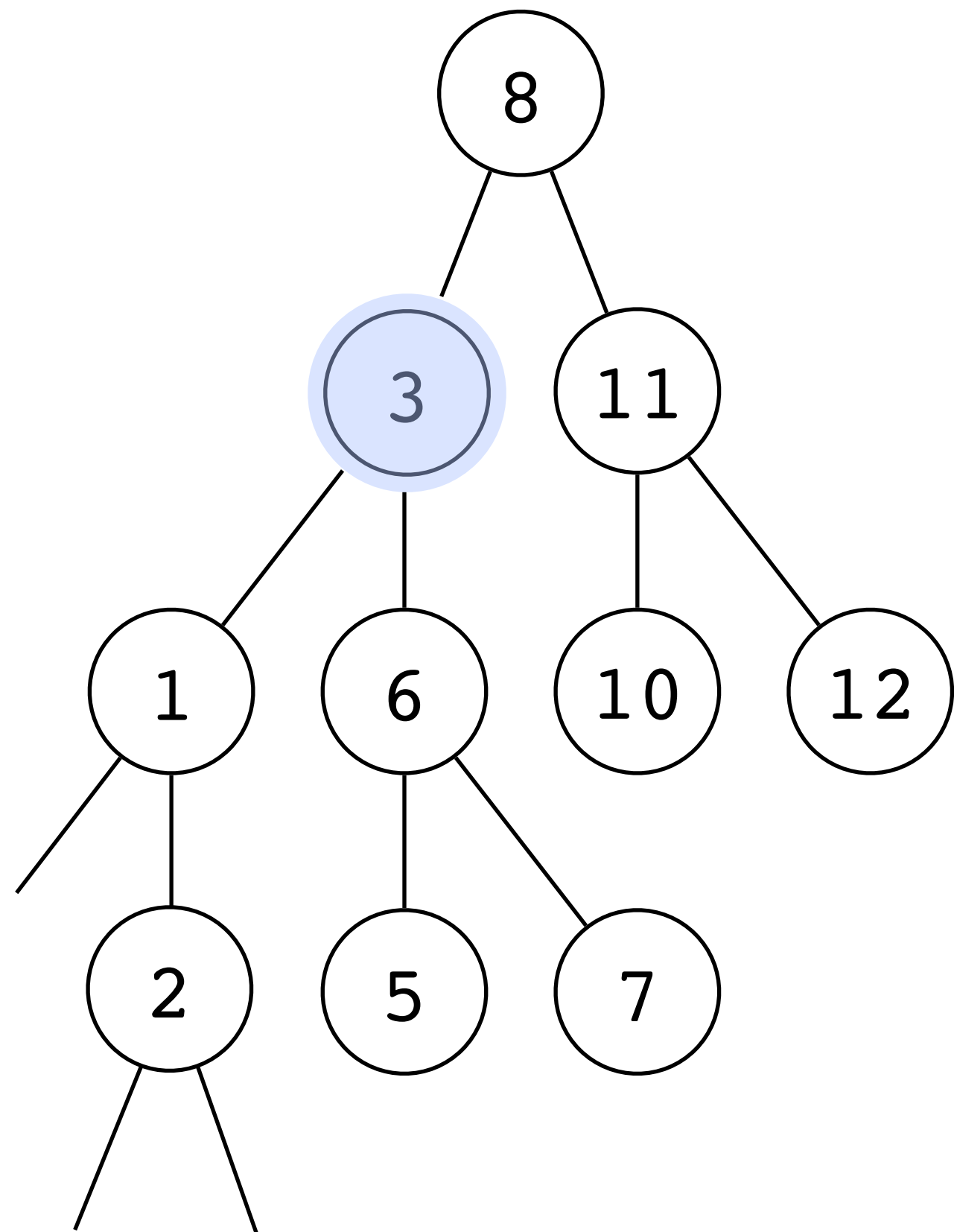
Exemple : suppression de la clé 4



- Extraction du max gauche

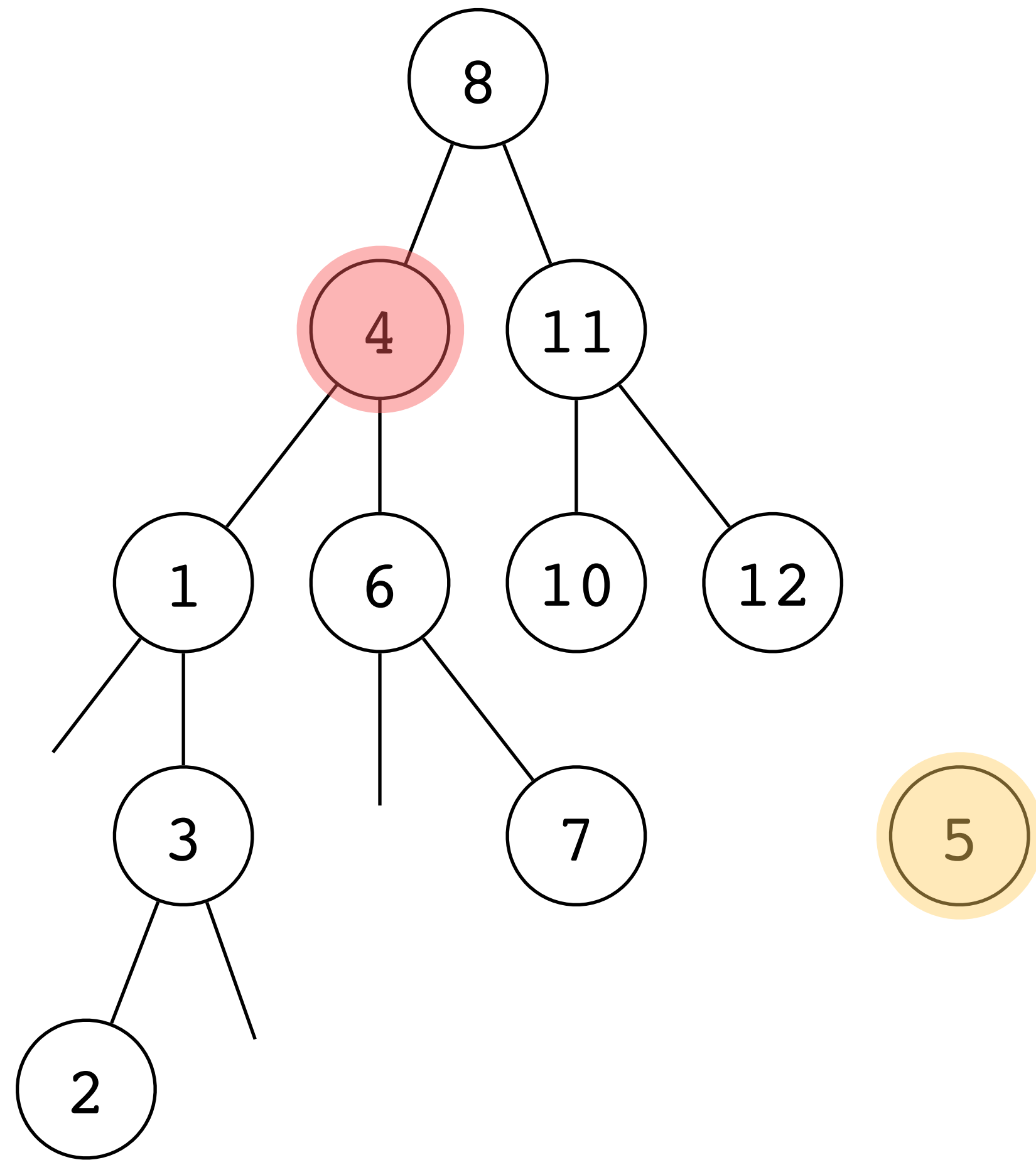


Exemple : suppression de la clé 4



- Extraction du max gauche
- Remplacement de l'élément à supprimer par le max gauche.

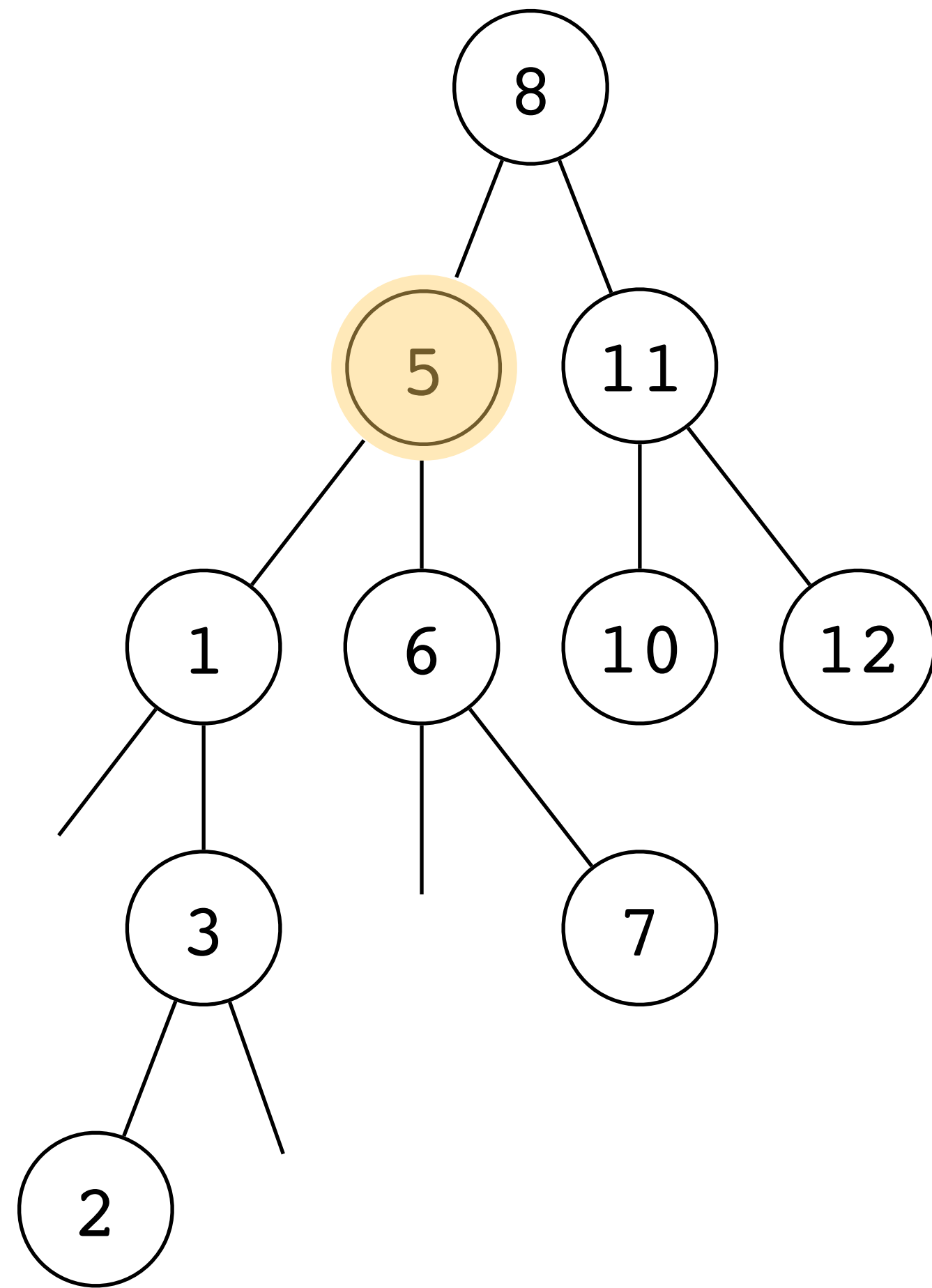
Exemple : suppression de la clé 4



- Extraction du min droite



Exemple : suppression de la clé 4



- Extraction du min droite
- Remplacement de l'élément à supprimer par le min droite.

Suppression de la clé k



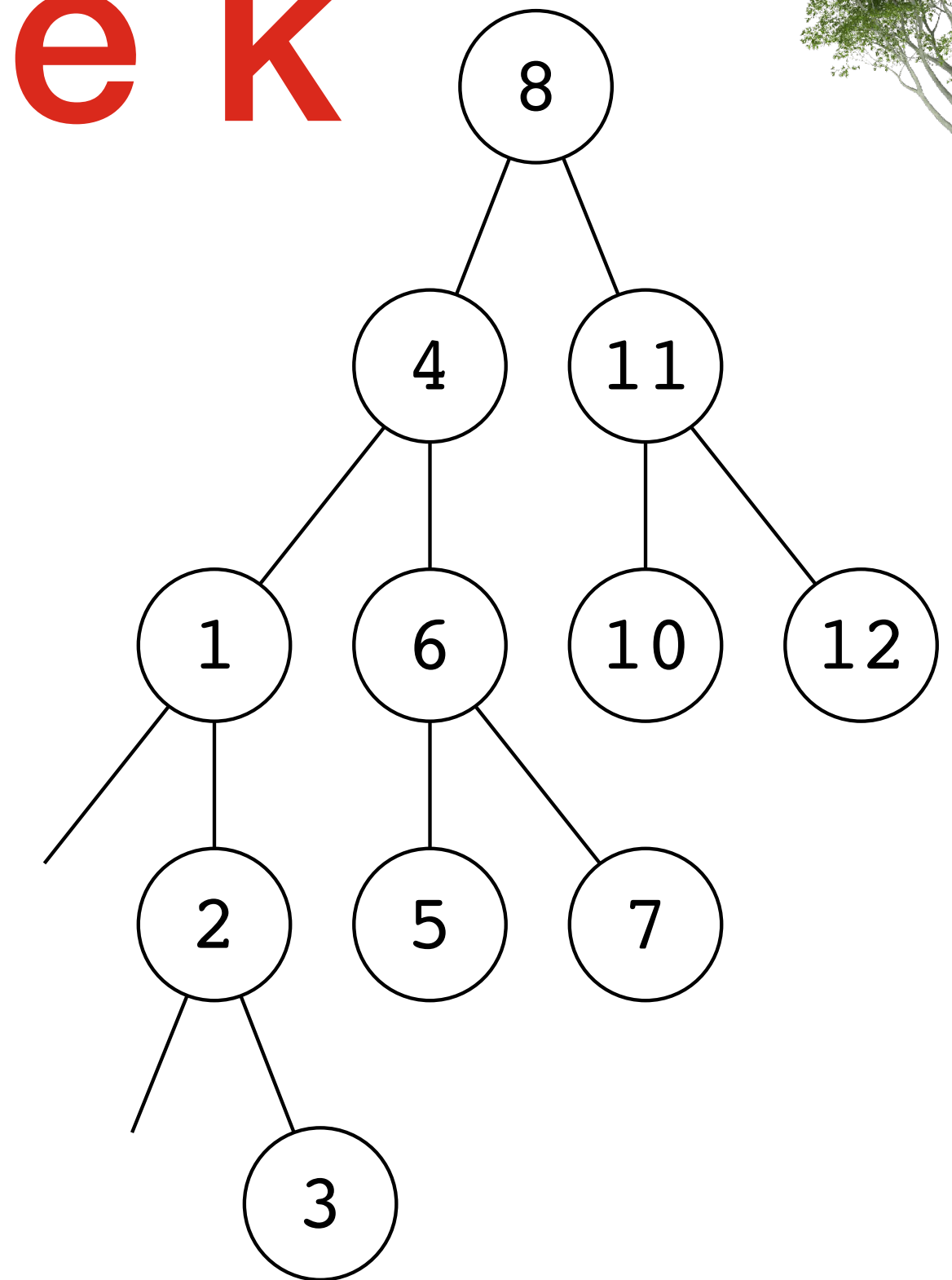
```

fonction supprimer (ref r, k)
  si r == ∅
    // 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 == ∅
      r ← r.droit
    sinon, si r.droit == ∅
      r ← r.gauche
    sinon, // Hibbard
      m ← sortir_min (r.droit)
      m.droit ← r.droit
      m.gauche ← r.gauche
      r ← m
    effacer tmp
  
```

Recherche de k

Cas simple : degré ≤ 1

Cas avancé : degré == 2



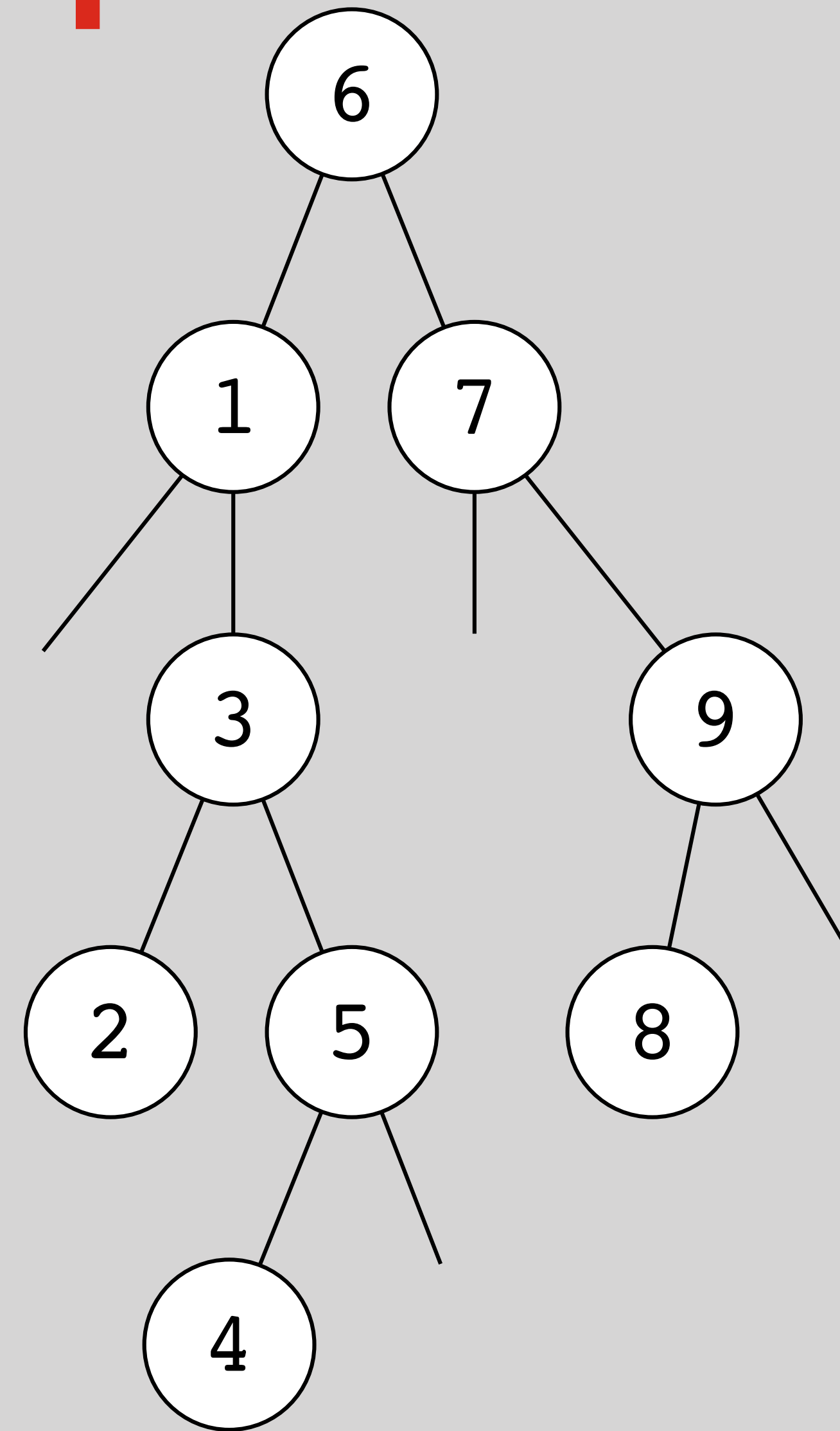
```

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

Exercice 1



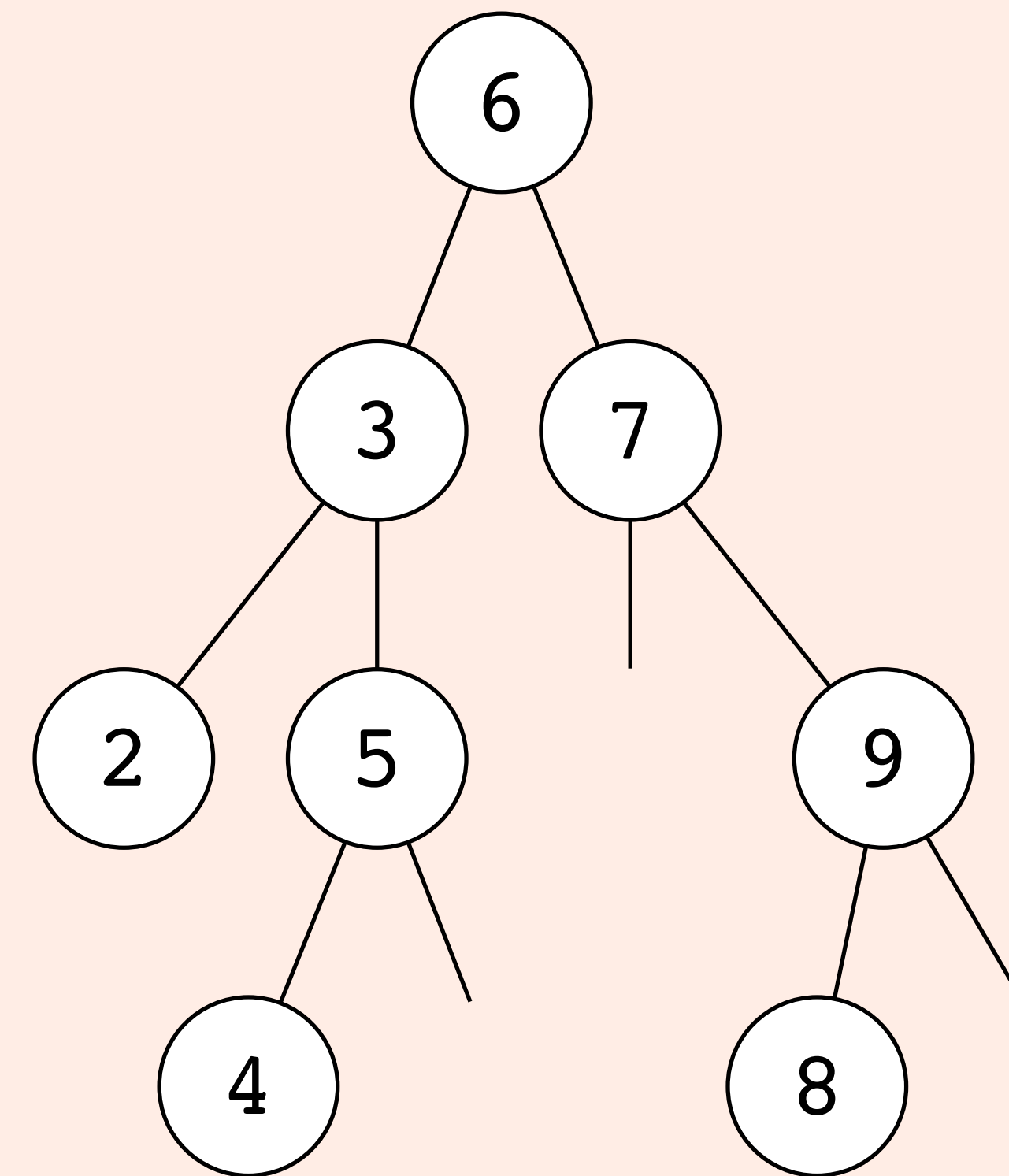
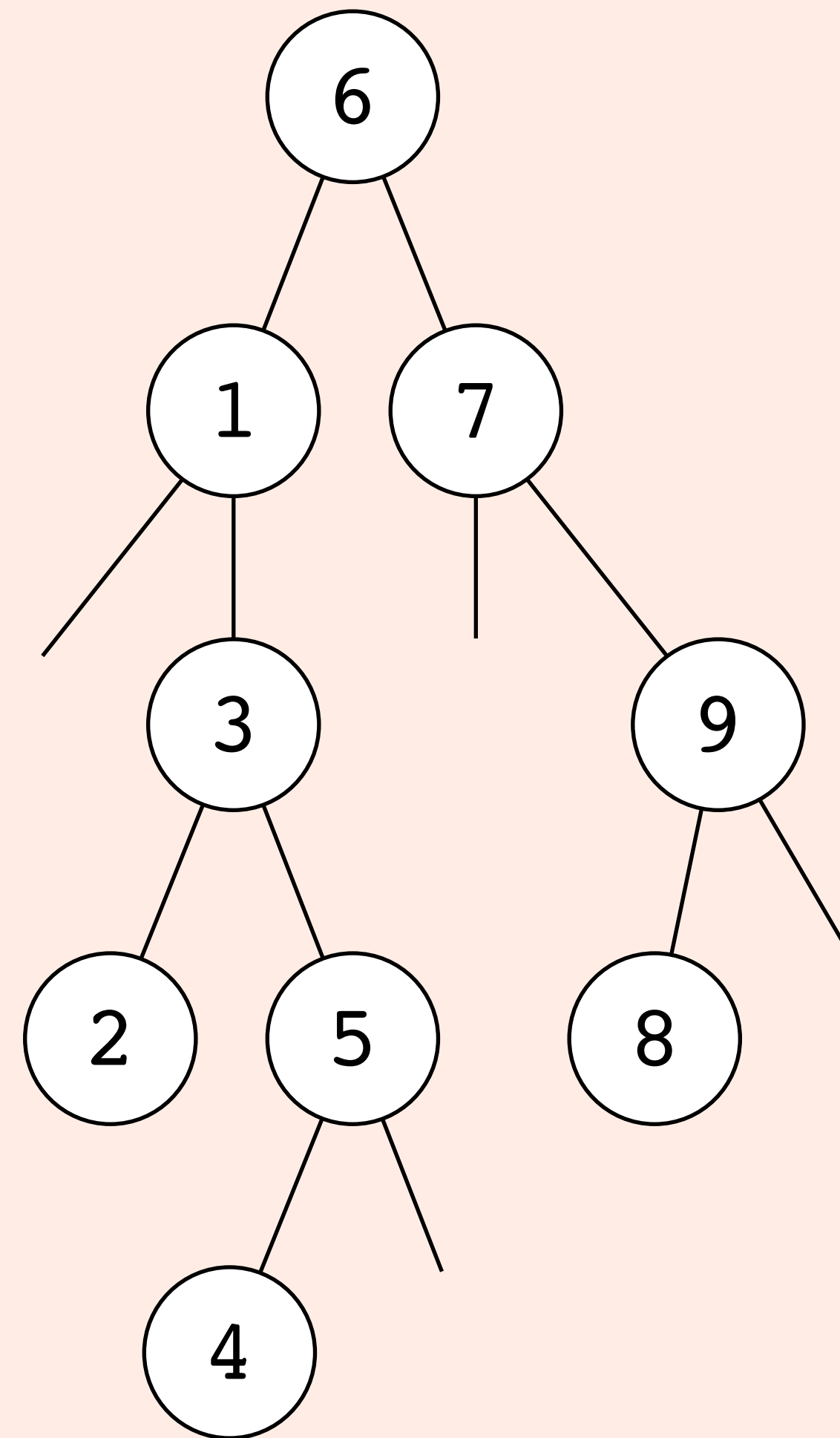
- On supprime le minimum de l'ABR ci-contre. Dessinez l'arbre résultant



Solution 1



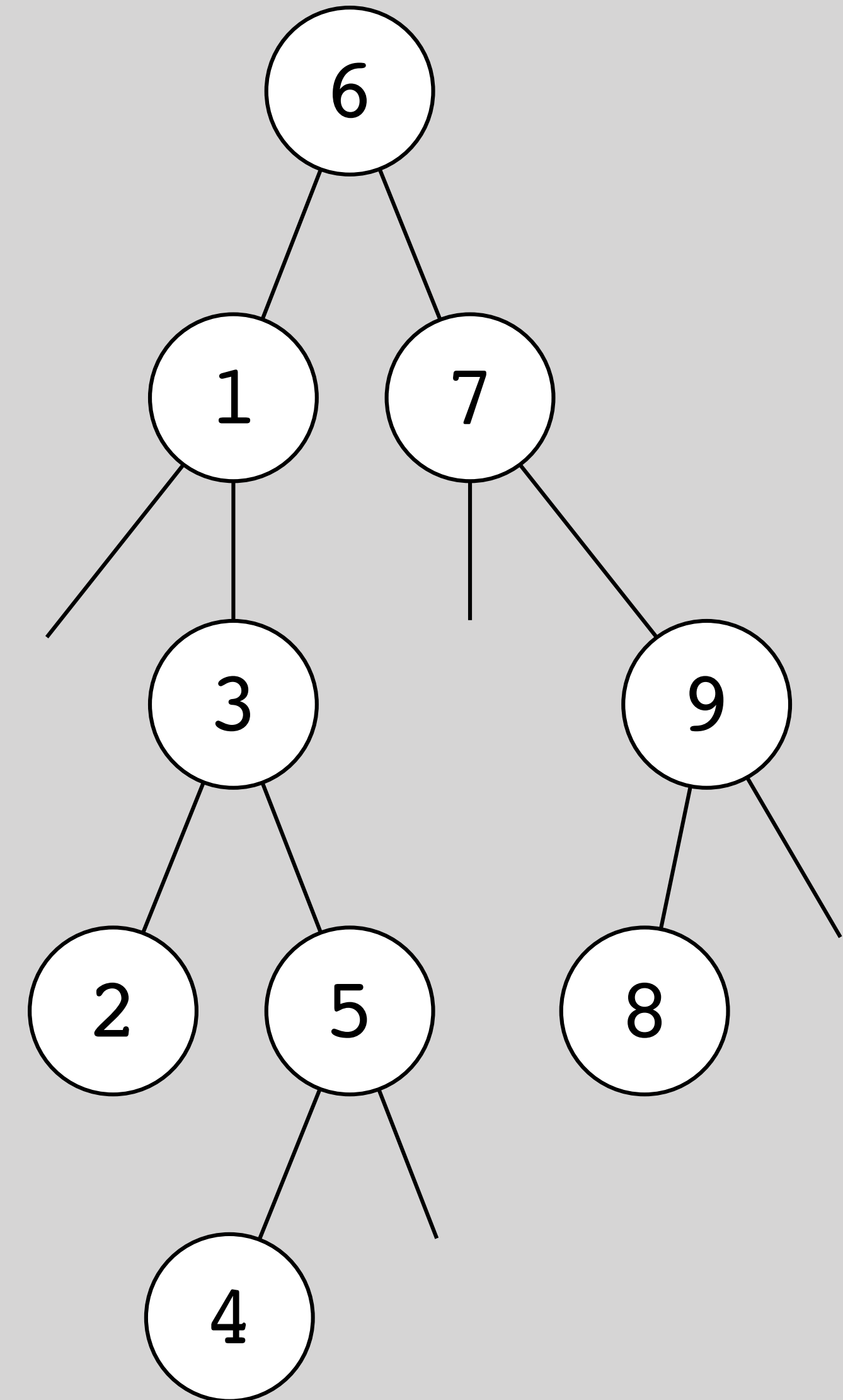
- En supprimant le minimum de l'ABR de gauche, on obtient celui de droite



Exercice 2



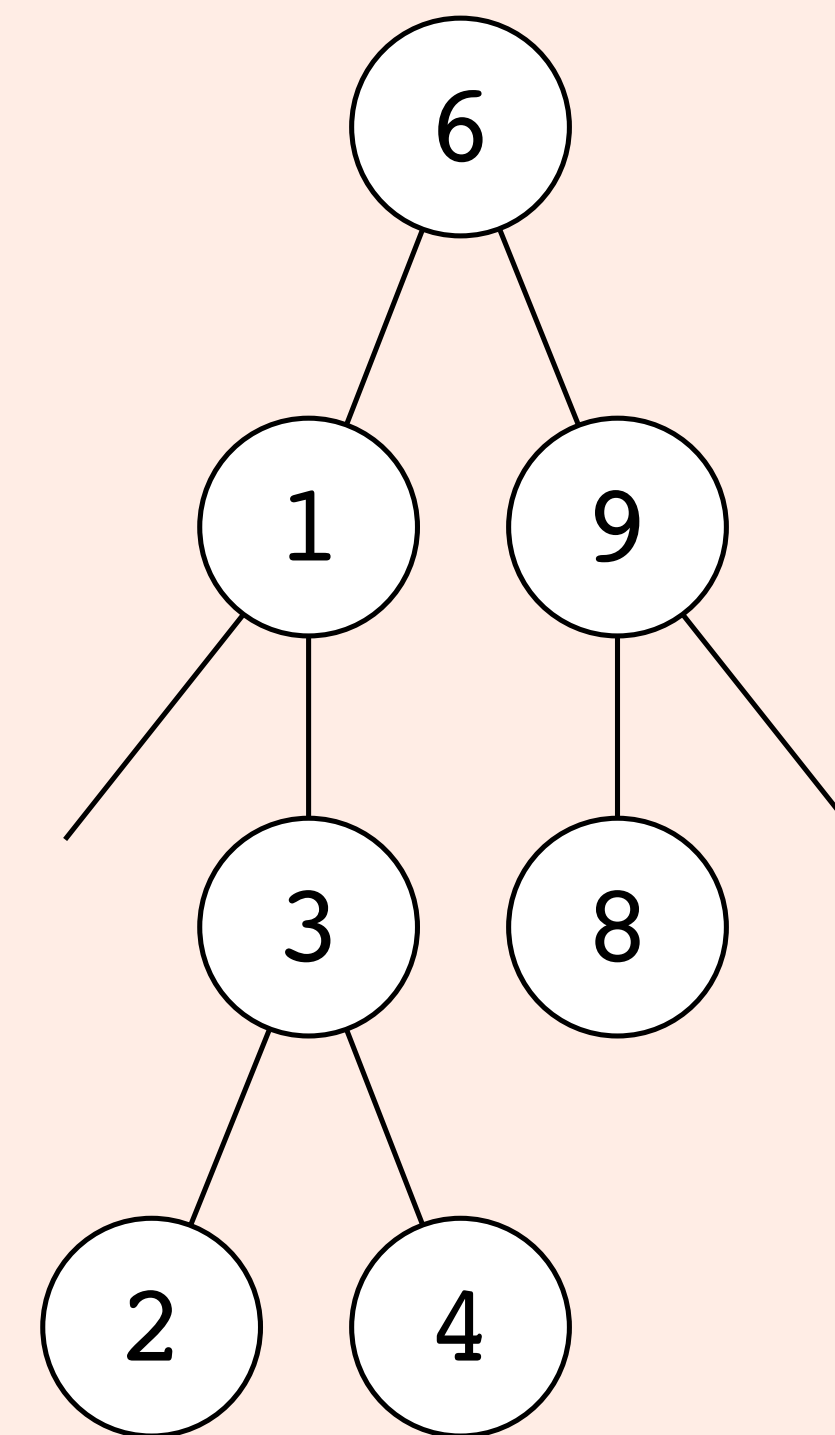
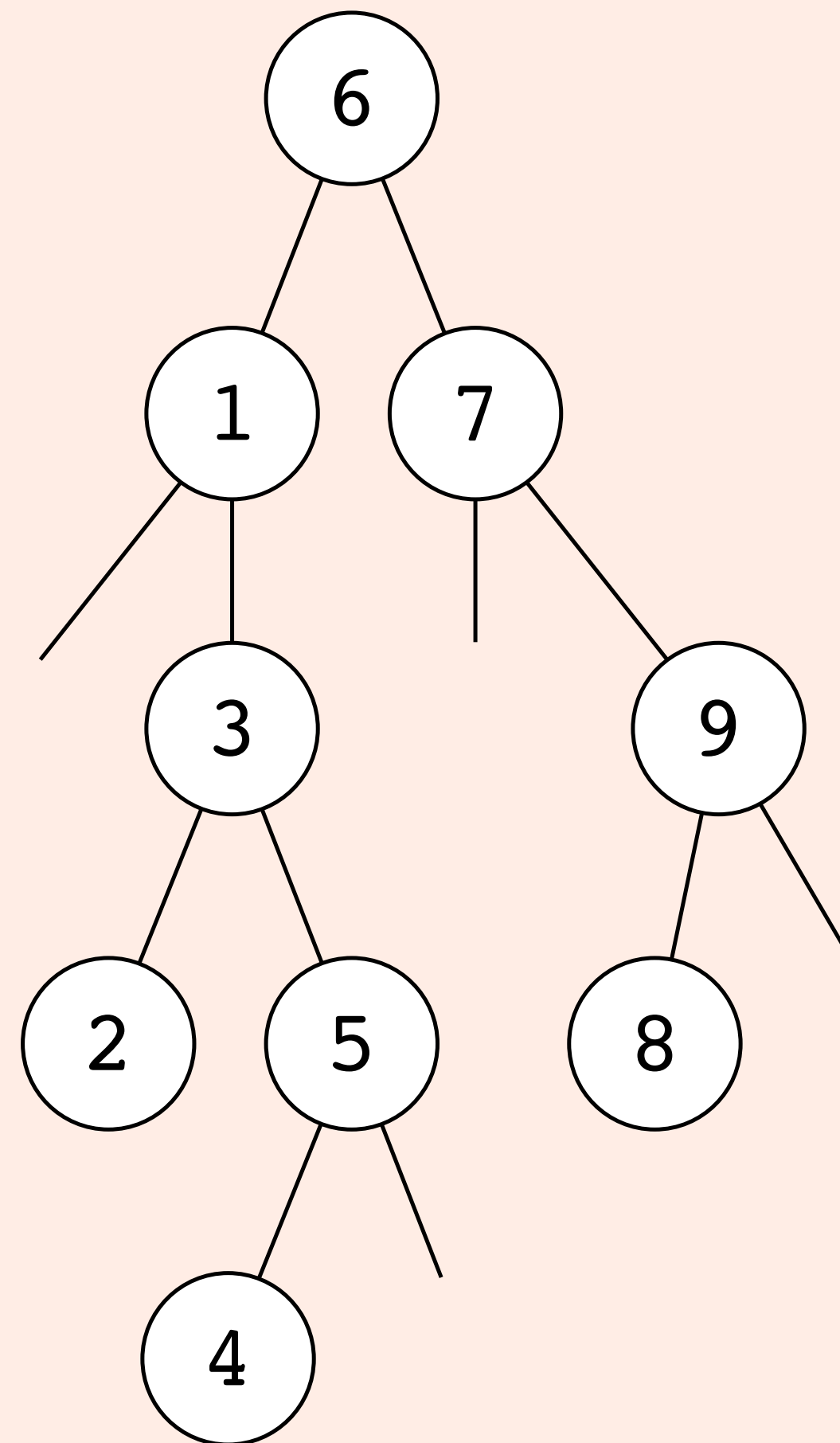
- On supprime les 2 noeuds de clé 5 et de clé 7 de l'arbre ci-contre. Dessinez l'arbre résultant



Solution 2



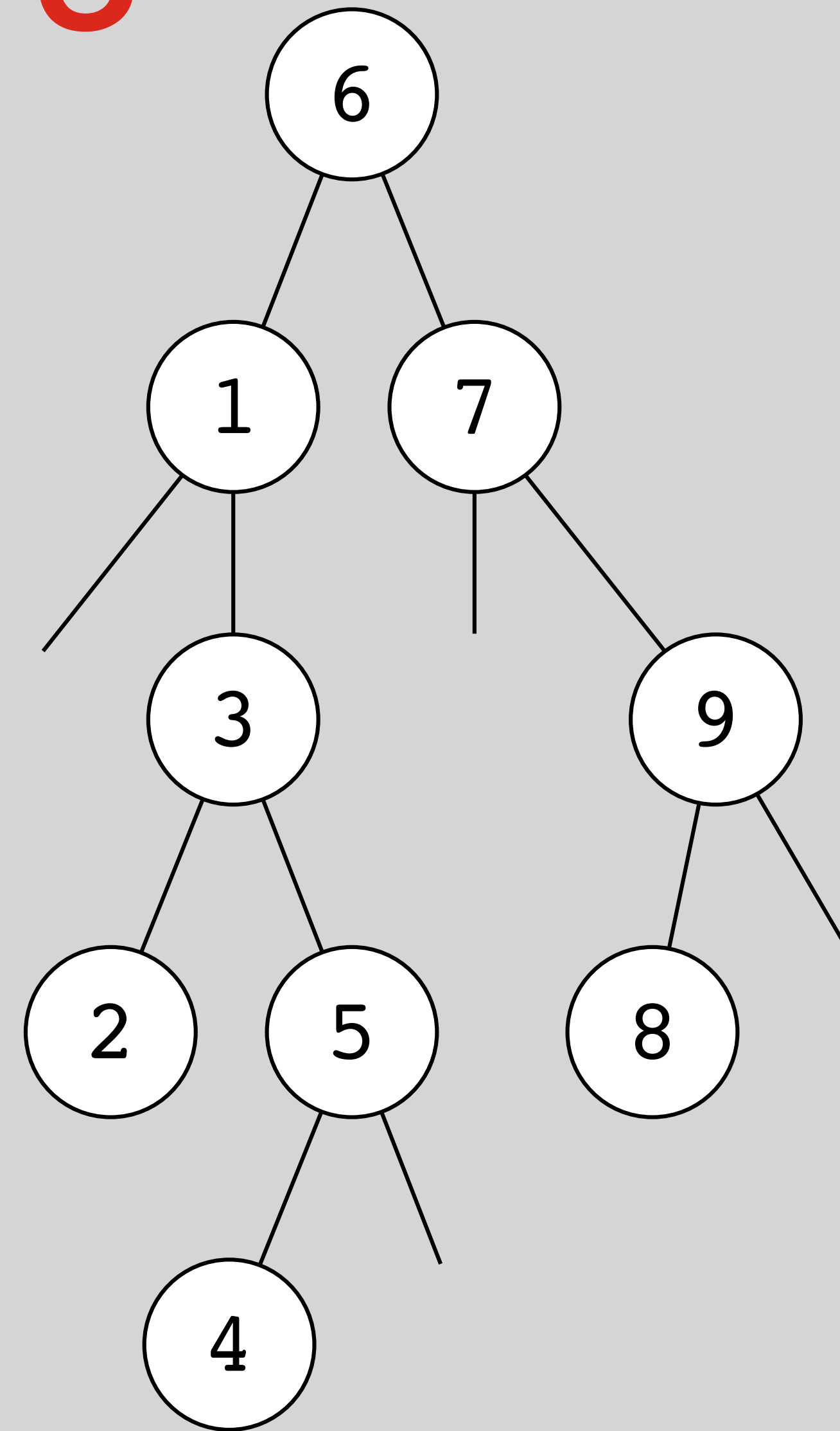
- En supprimant les 2 noeuds de clé 5 et de clé 7 de l'arbre de gauche, on obtient l'arbre de droite.



Exercice 3



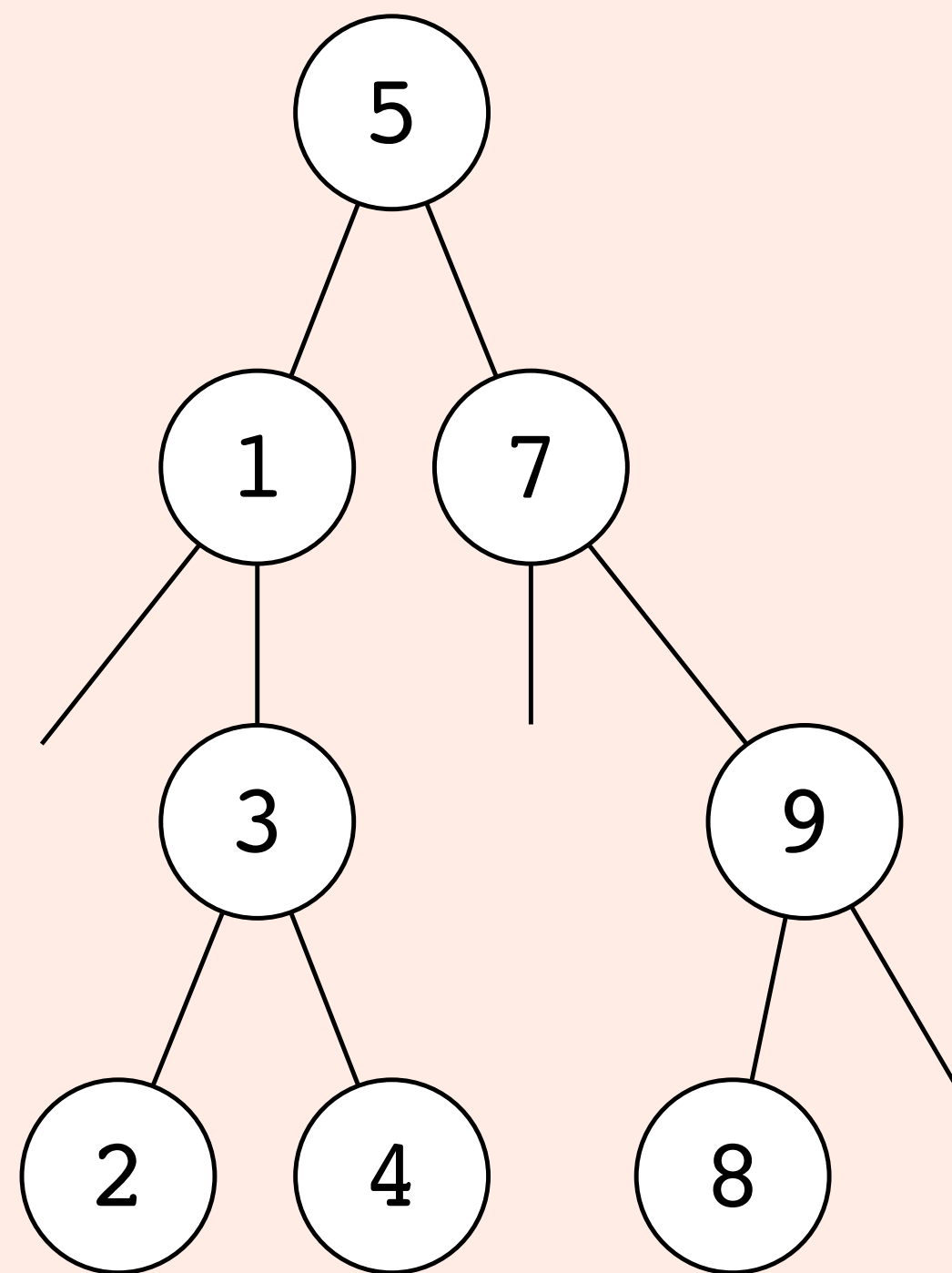
- On supprime le noeud de clé 6 de l'arbre ci-contre. Dessinez les deux arbres résultants possibles en appliquant la technique de Hibbard.



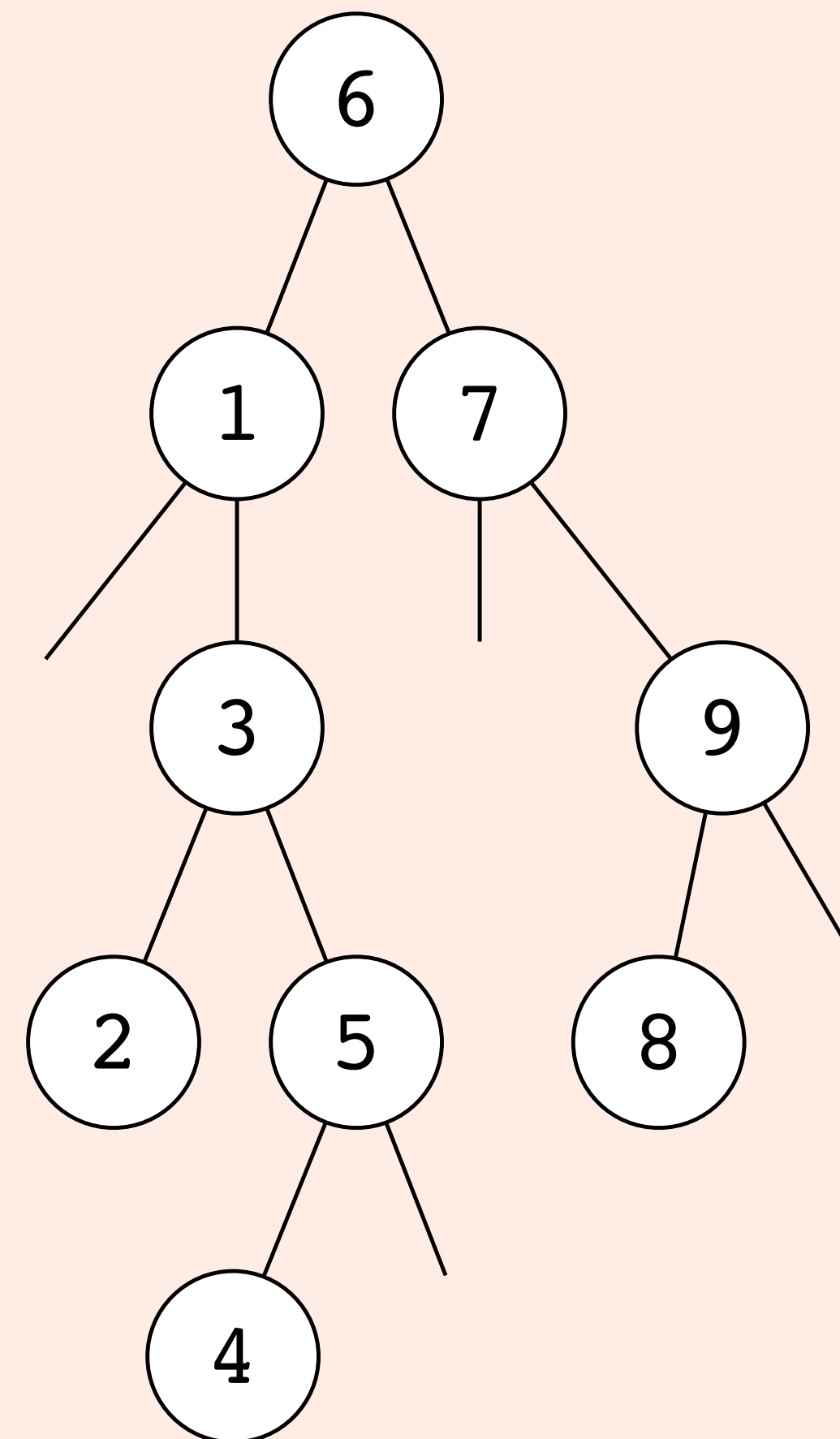


Solution 3

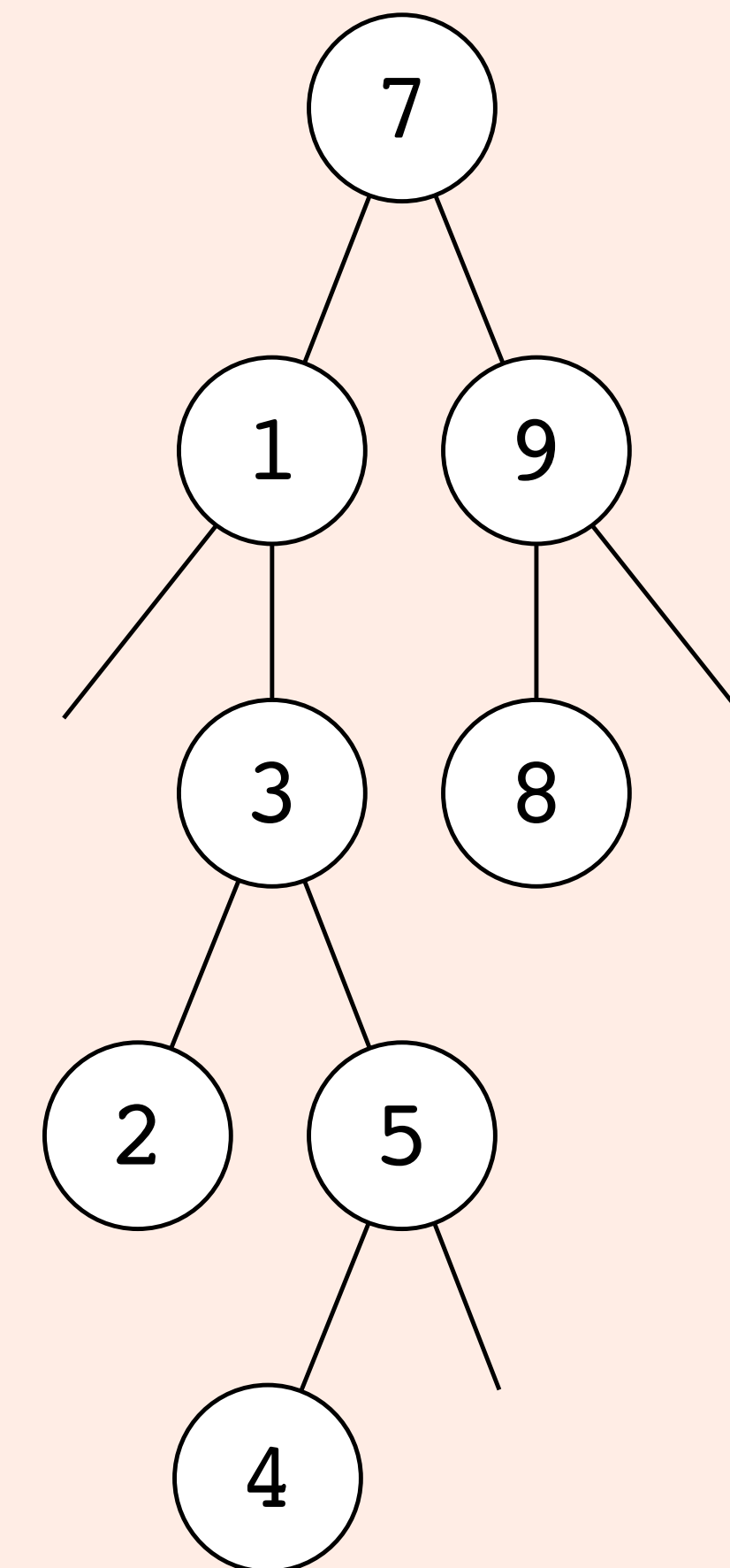
En remontant le
maximum de gauche



Arbre dont on veut
supprimer la clé 6



En remontant le
minimum de droite



11. Itération sur un ABR



Rappel : itérer sur un arbre



Pré-ordre

```
fonction début(r)
    retourner r

fonction fin(r)
    retourner ∅

fonction suivant(r)
    si aîné(r) ≠ ∅
        retourner aîné(r)
    sinon
        tant que puiné(r) = ∅
            et parent(r) ≠ ∅
                r ← parent(r)
        retourner puiné(r)
```

Post-ordre

```
fonction début(r)
    tant que aîné(r) ≠ ∅
        r ← aîné(r)
    retourner r

fonction fin(r)
    retourner ∅

fonction suivant(r)
    si puiné(r) ≠ ∅
        retourner début(puiné(r))
    sinon
        retourner parent(r)
```

parent, aîné, puiné dans un ABR



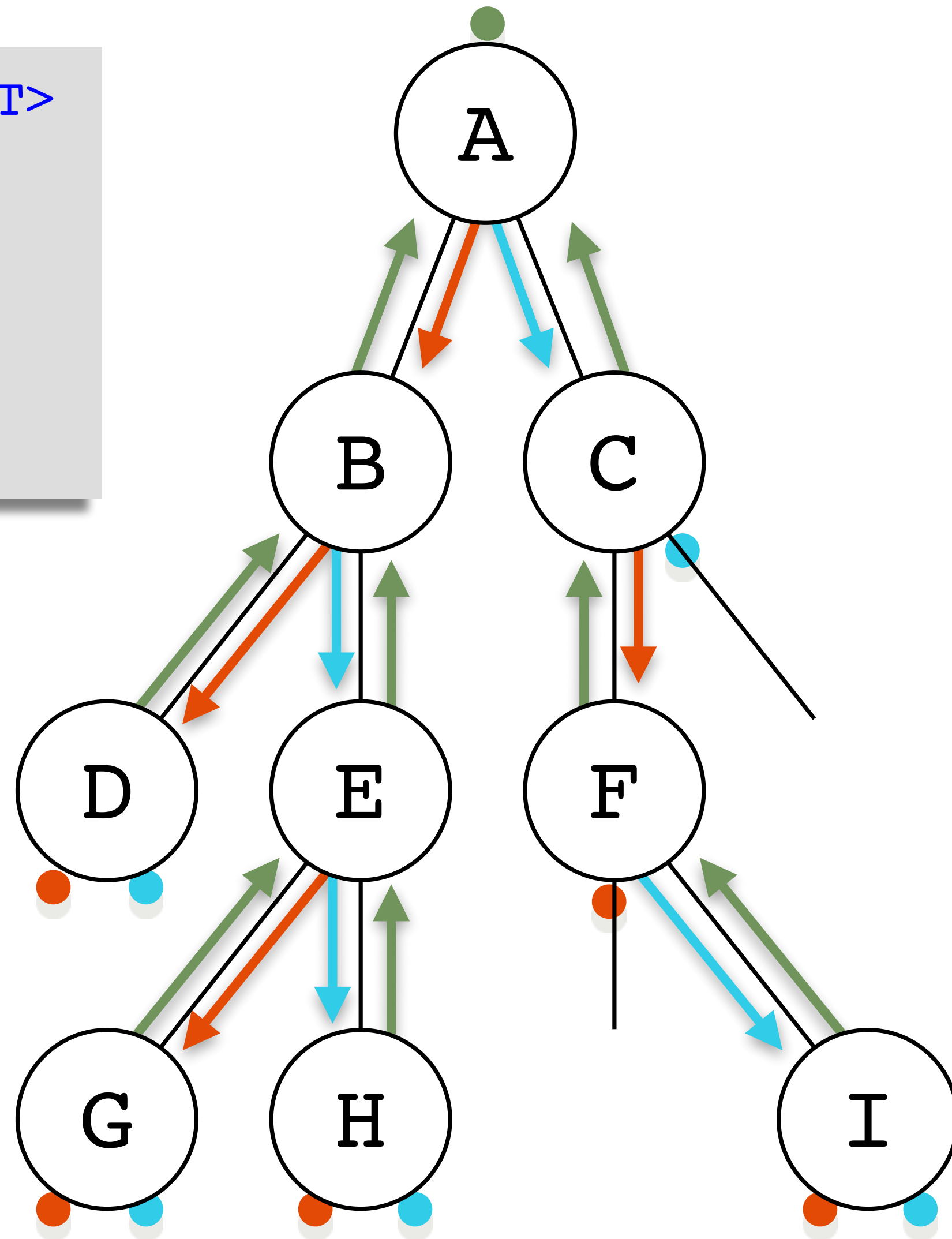
structure Noeud<T>

T étiquette

Noeud* parent

Noeud* gauche

Noeud* droit



parent, aîné, puiné dans un ABR



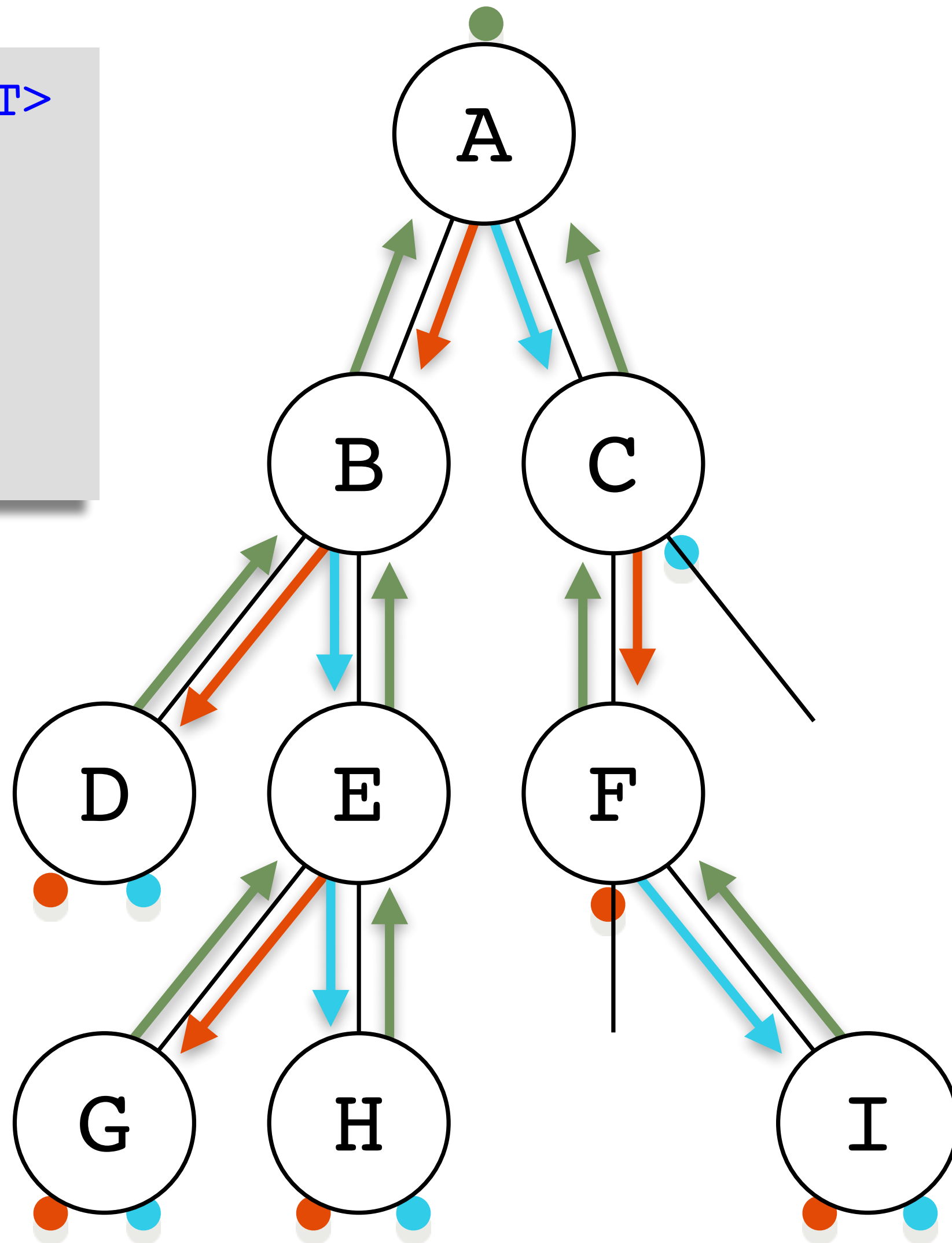
```
structure Noeud<T>
```

```
  T étiquette
```

```
  Noeud* parent
```

```
  Noeud* gauche
```

```
  Noeud* droit
```



```
fonction parent (r)
```

```
  retourner r.parent
```

```
fonction aîné (r)
```

```
  si r.gauche != ∅,
```

```
    retourner r.gauche
```

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

```
fonction puiné (r)
```

```
  si est_enfant_gauche(r),
```

```
    retourner r.parent.droit
```

```
  sinon, retourner ∅
```

```
fonction est_enfant_gauche (r)
```

```
  p ← r.parent
```

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

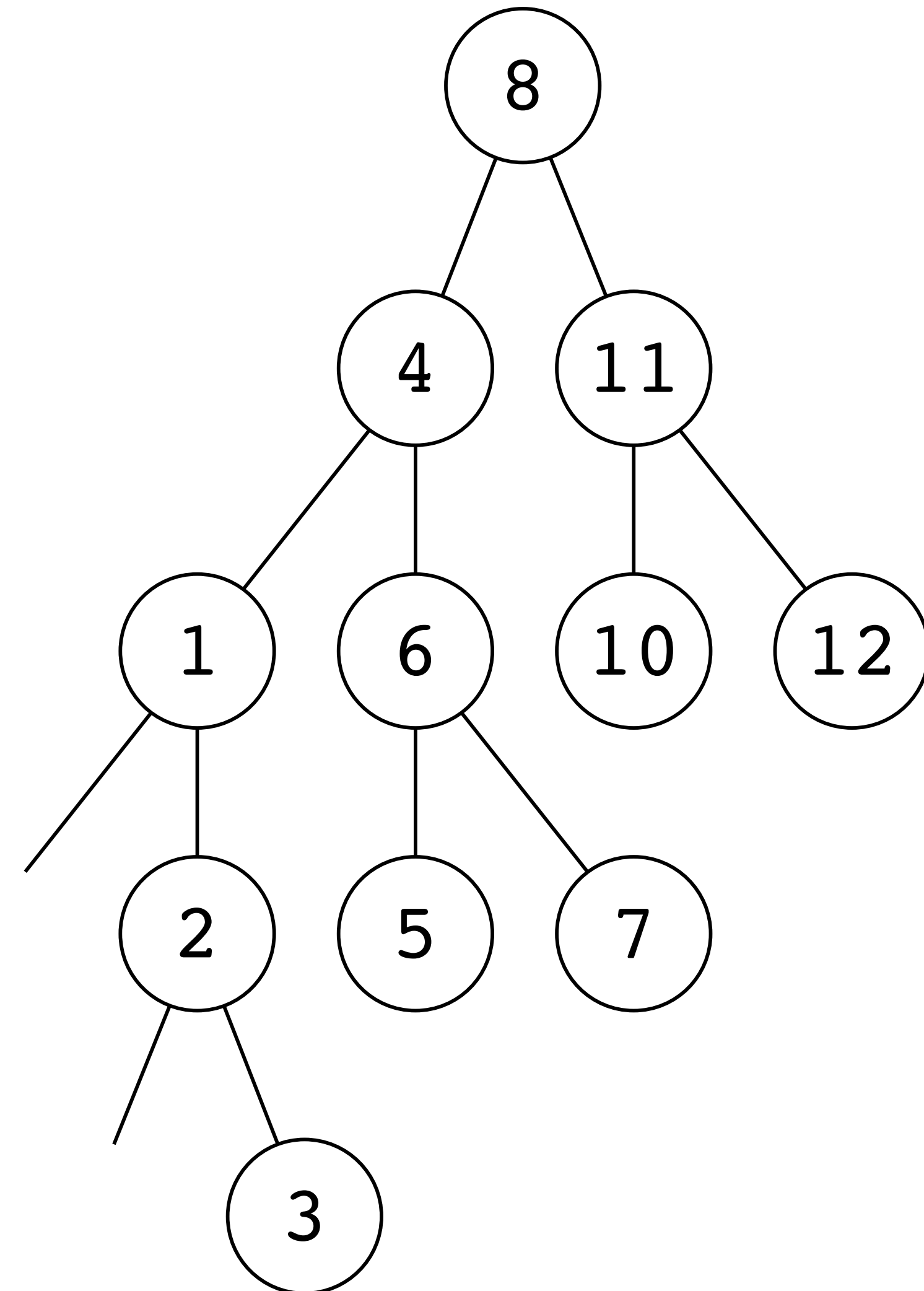
Itération en ordre symétrique



```

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

(à définir)



Itération en ordre symétrique



```

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

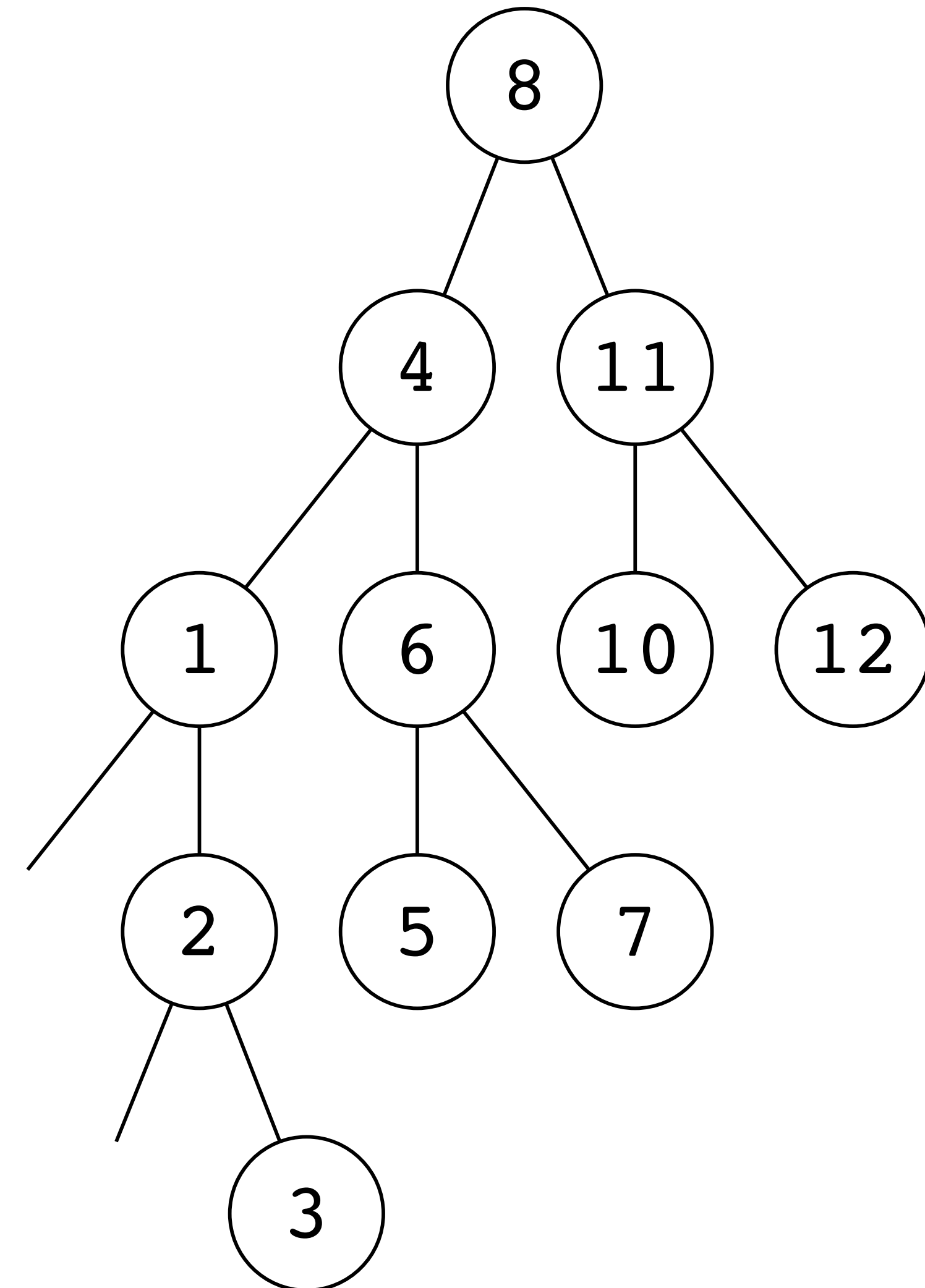
(à définir)

```

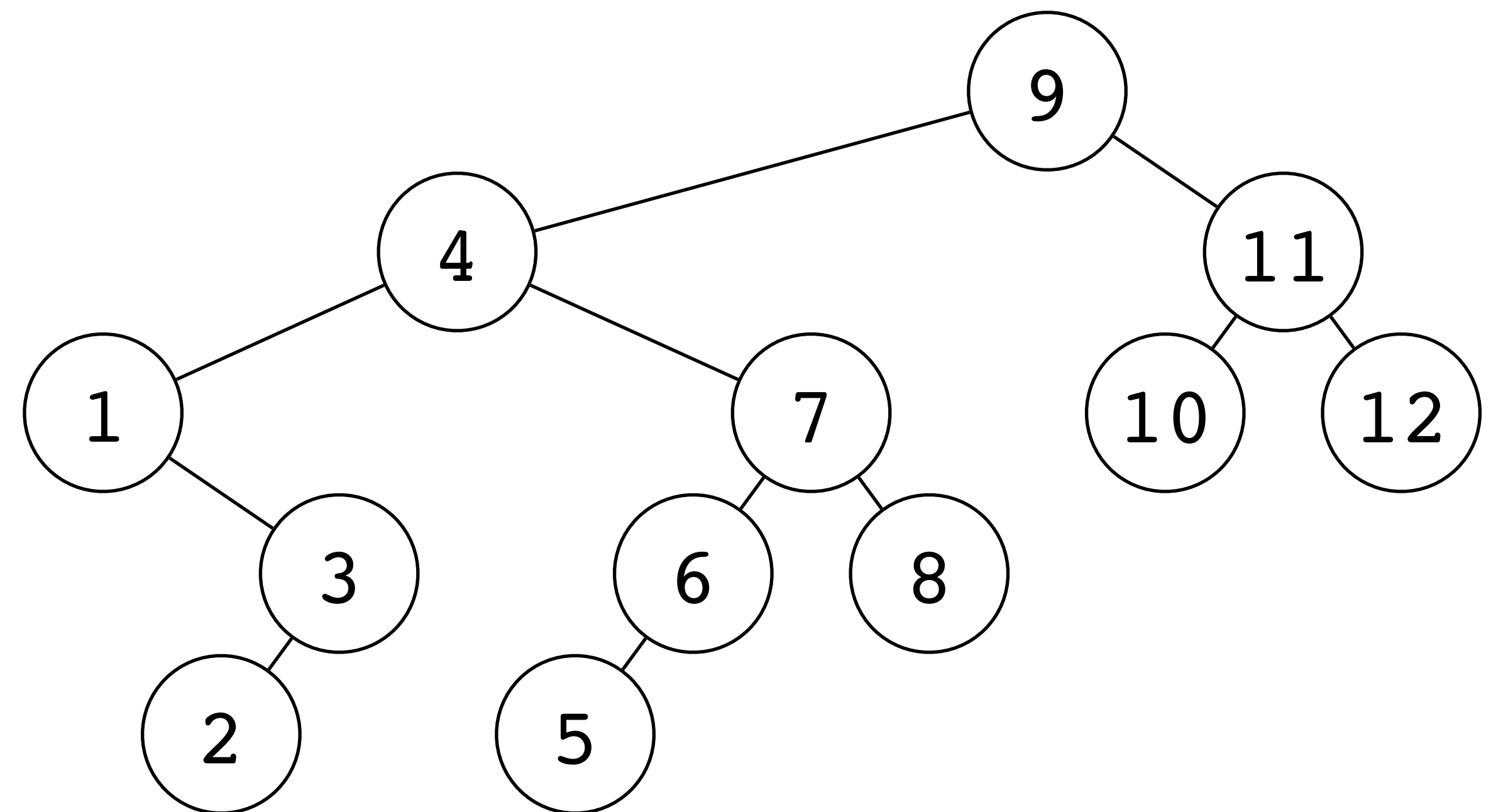
fonction début(r)
  retourner min(r)
  
```

```

fonction fin(r)
  retourner ∅
  
```



suivant(r) en ordre symétrique



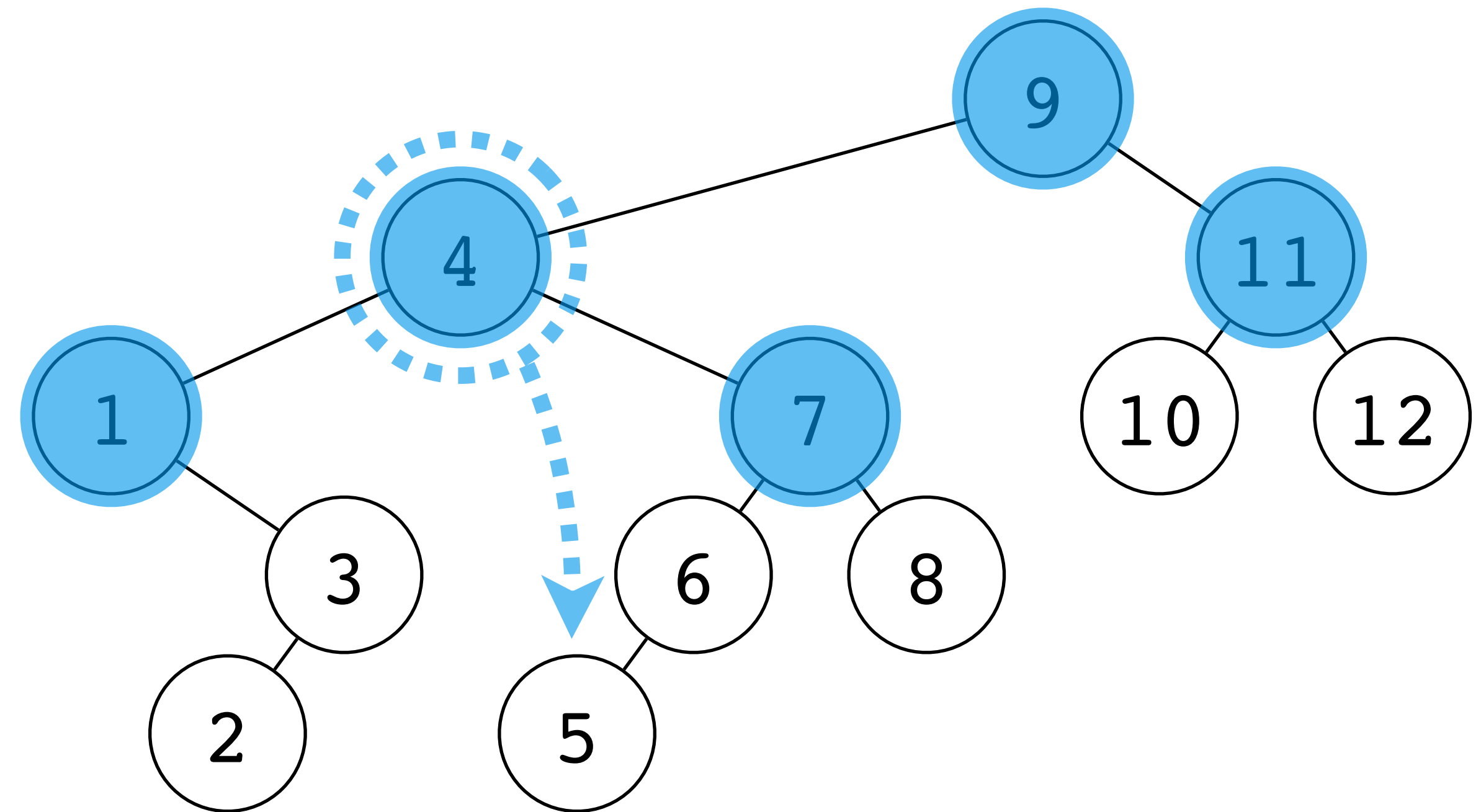
suivant(r)

en ordre symétrique



Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit



suivant(r) en ordre symétrique

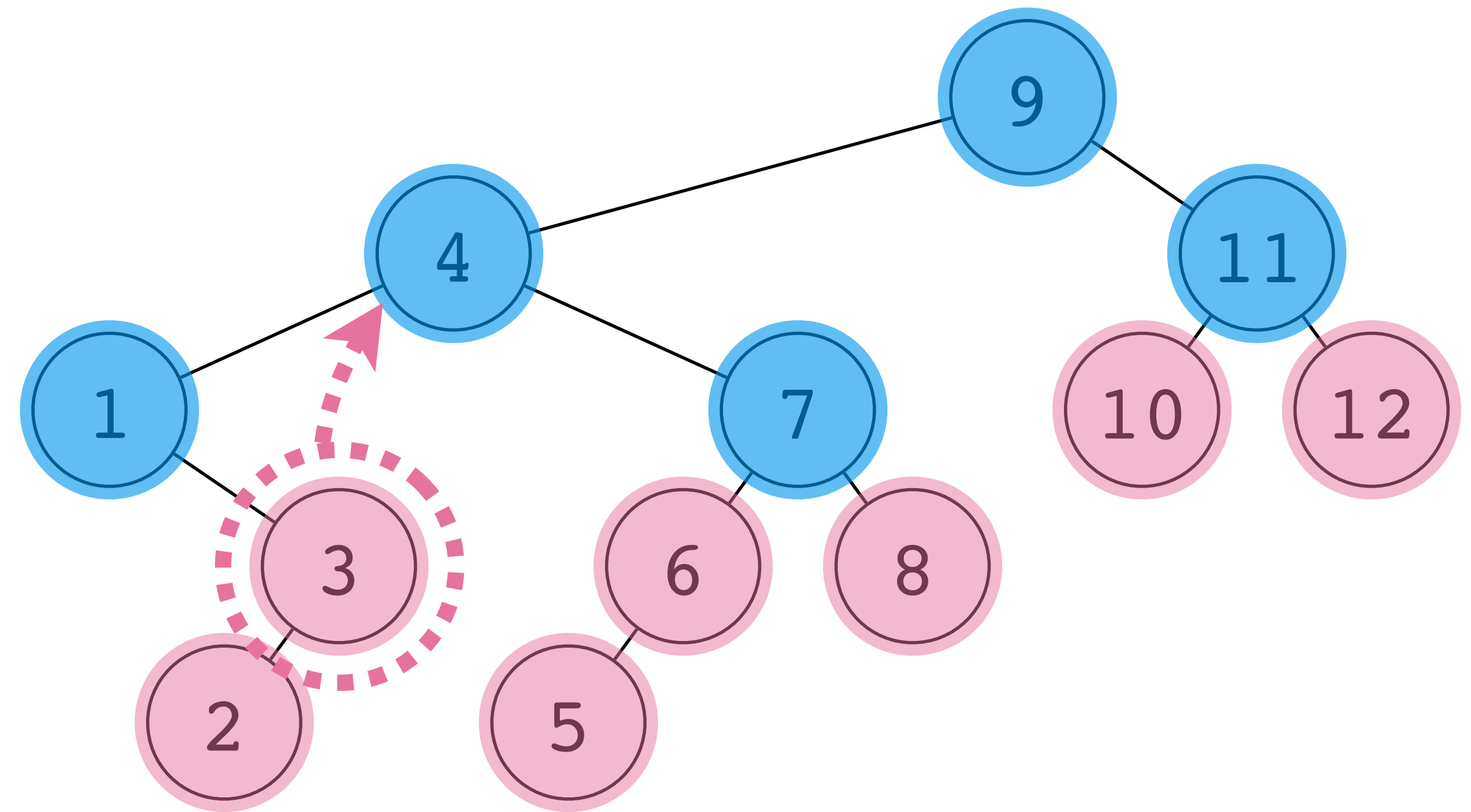


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r)

en ordre symétrique

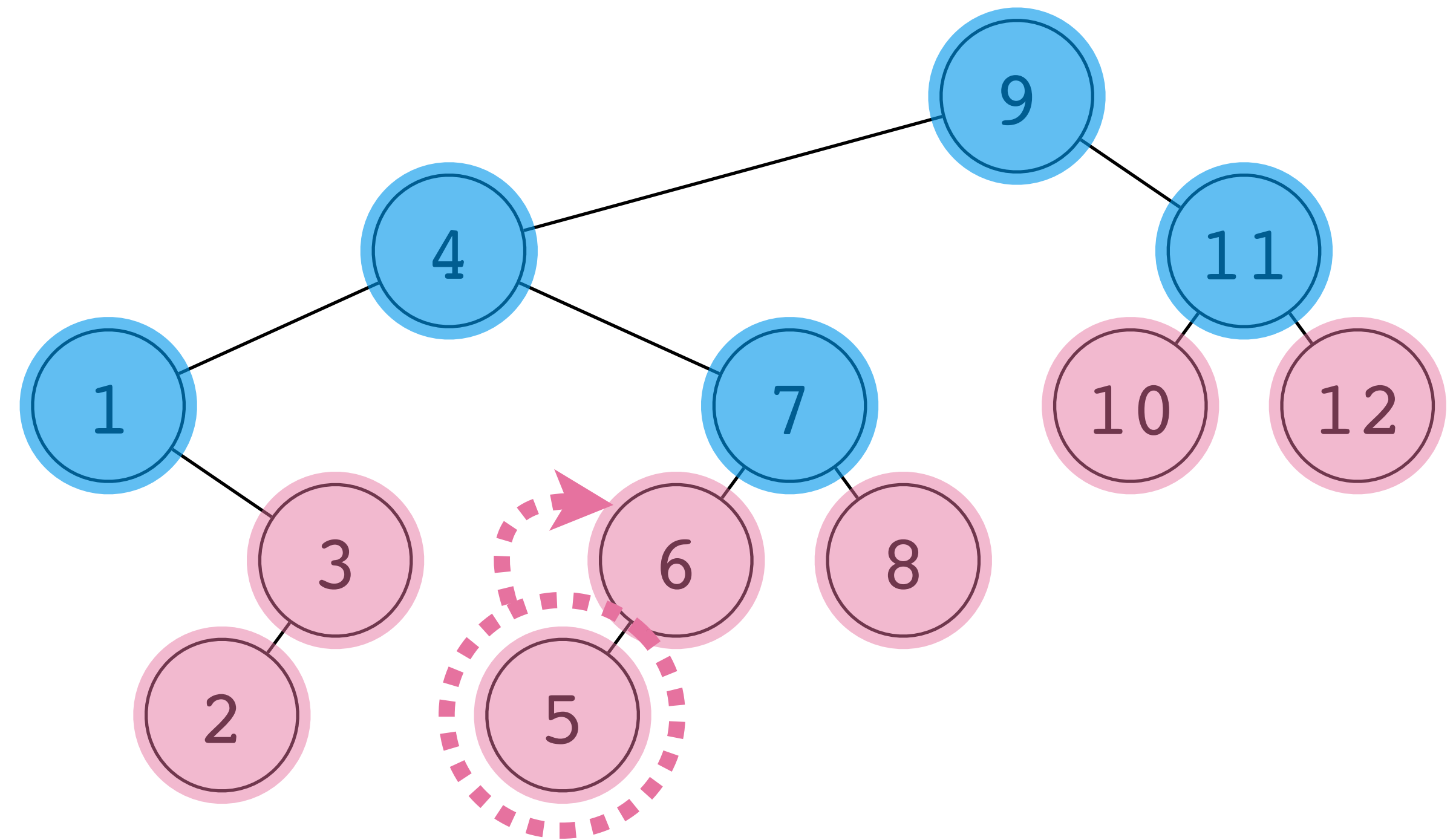


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r) en ordre symétrique

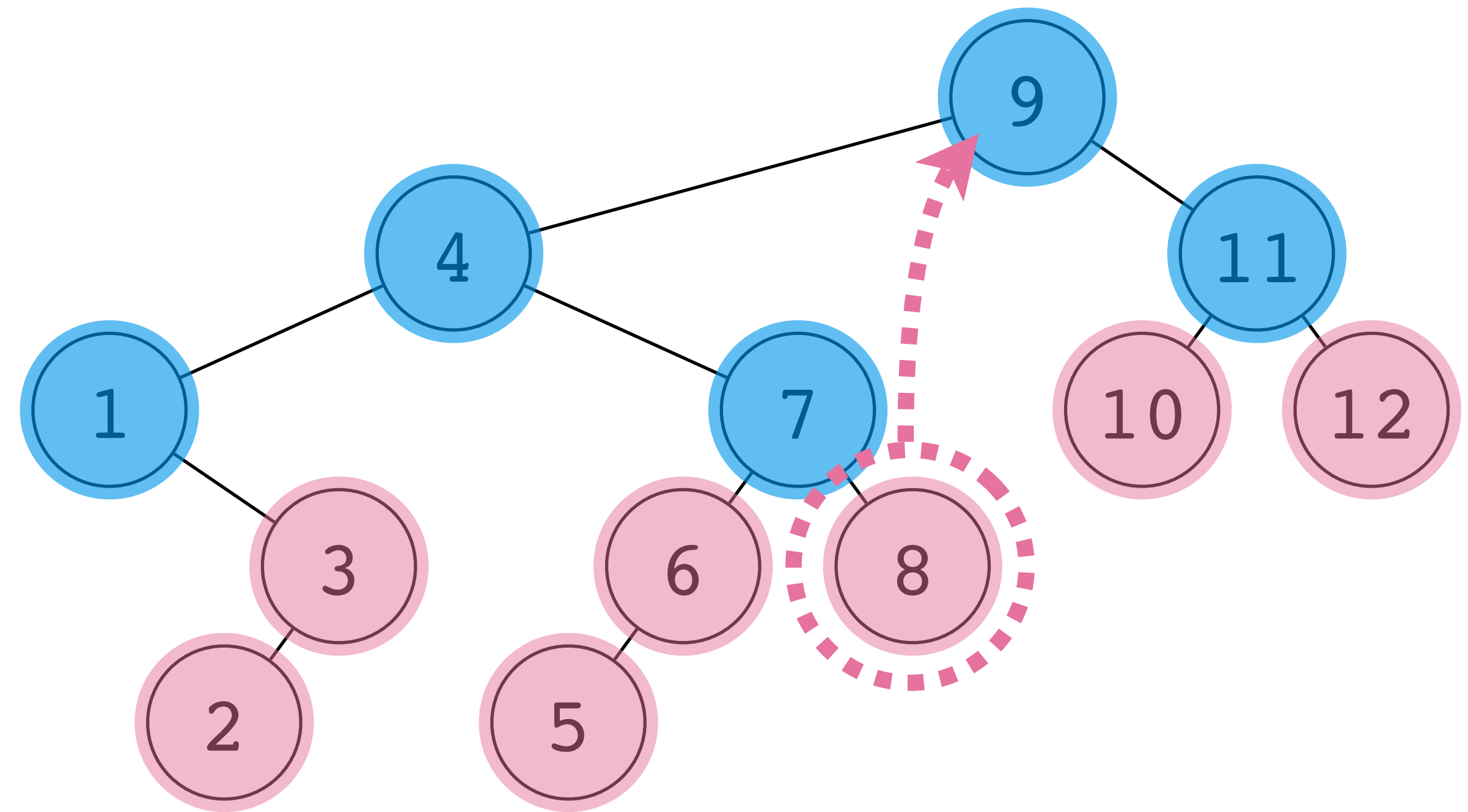


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r) en ordre symétrique

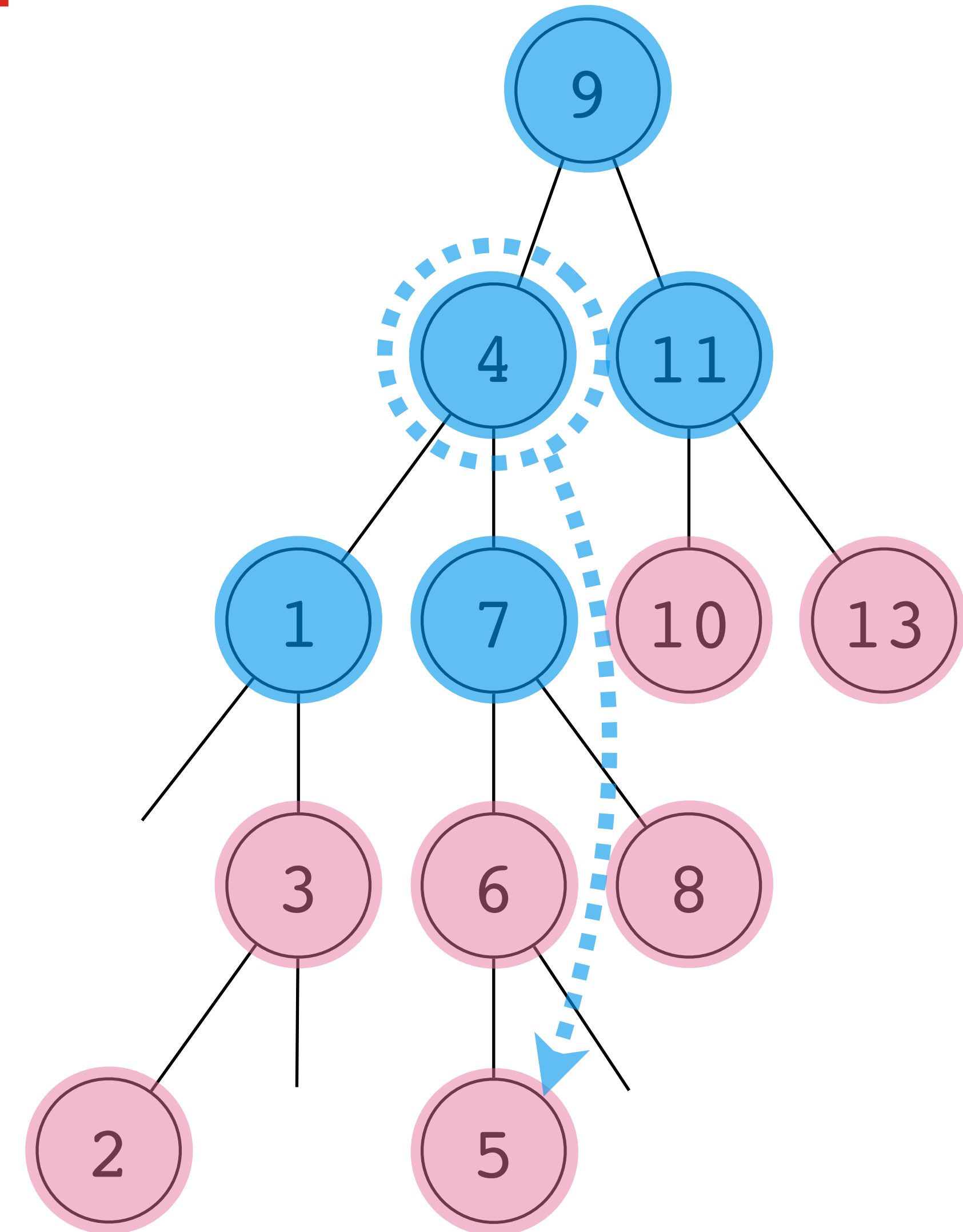


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r) en ordre symétrique

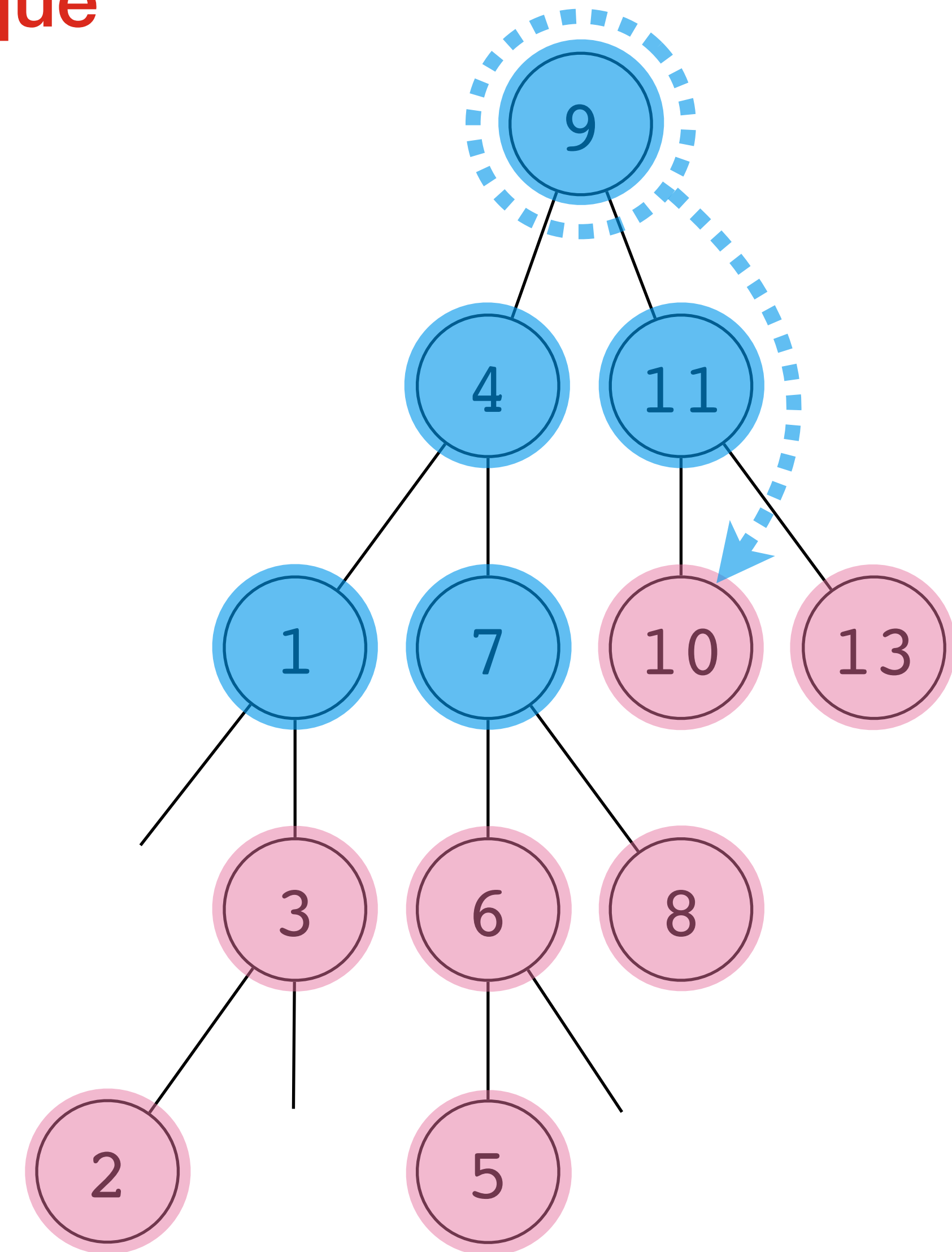


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r)

en ordre symétrique

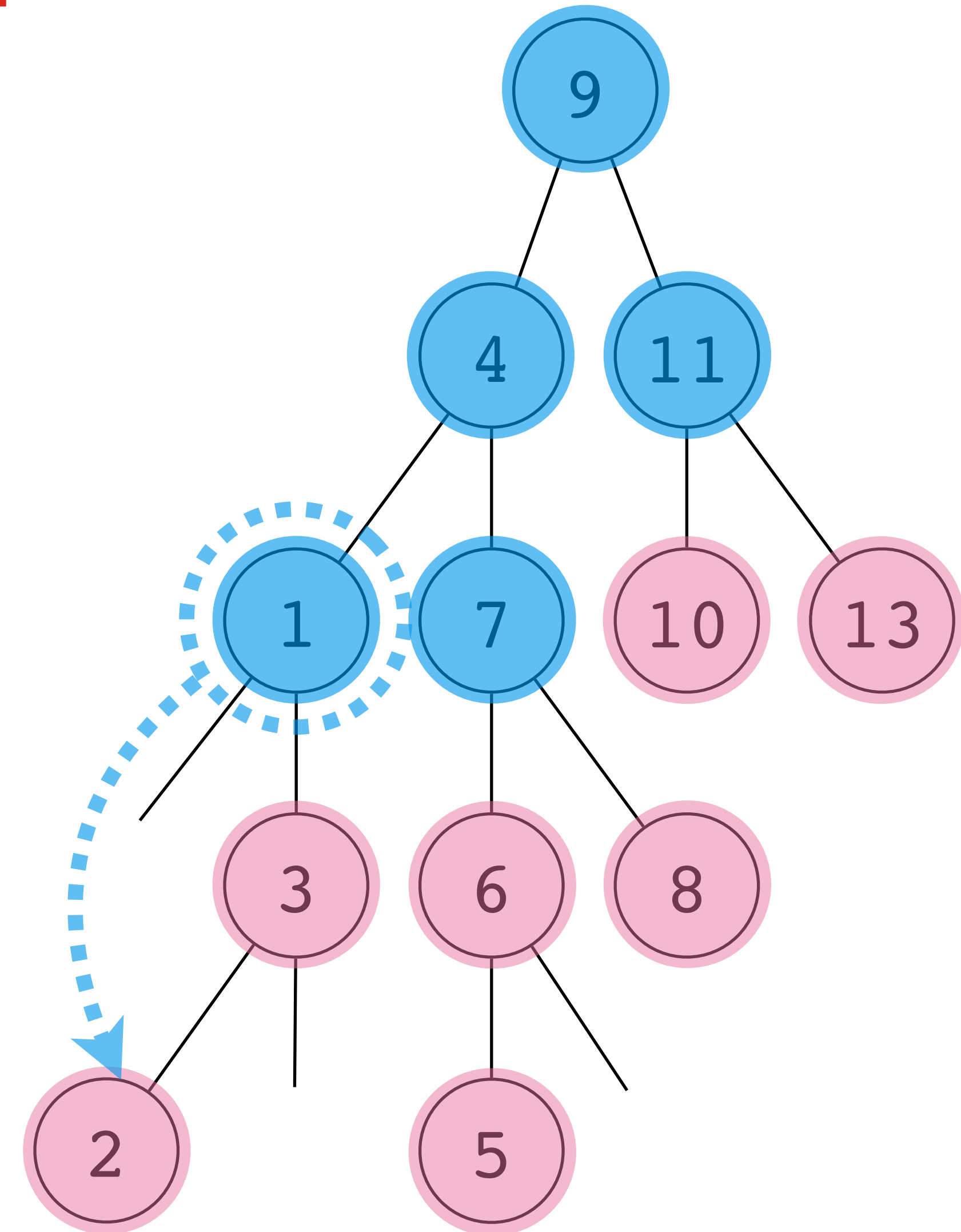


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r) en ordre symétrique

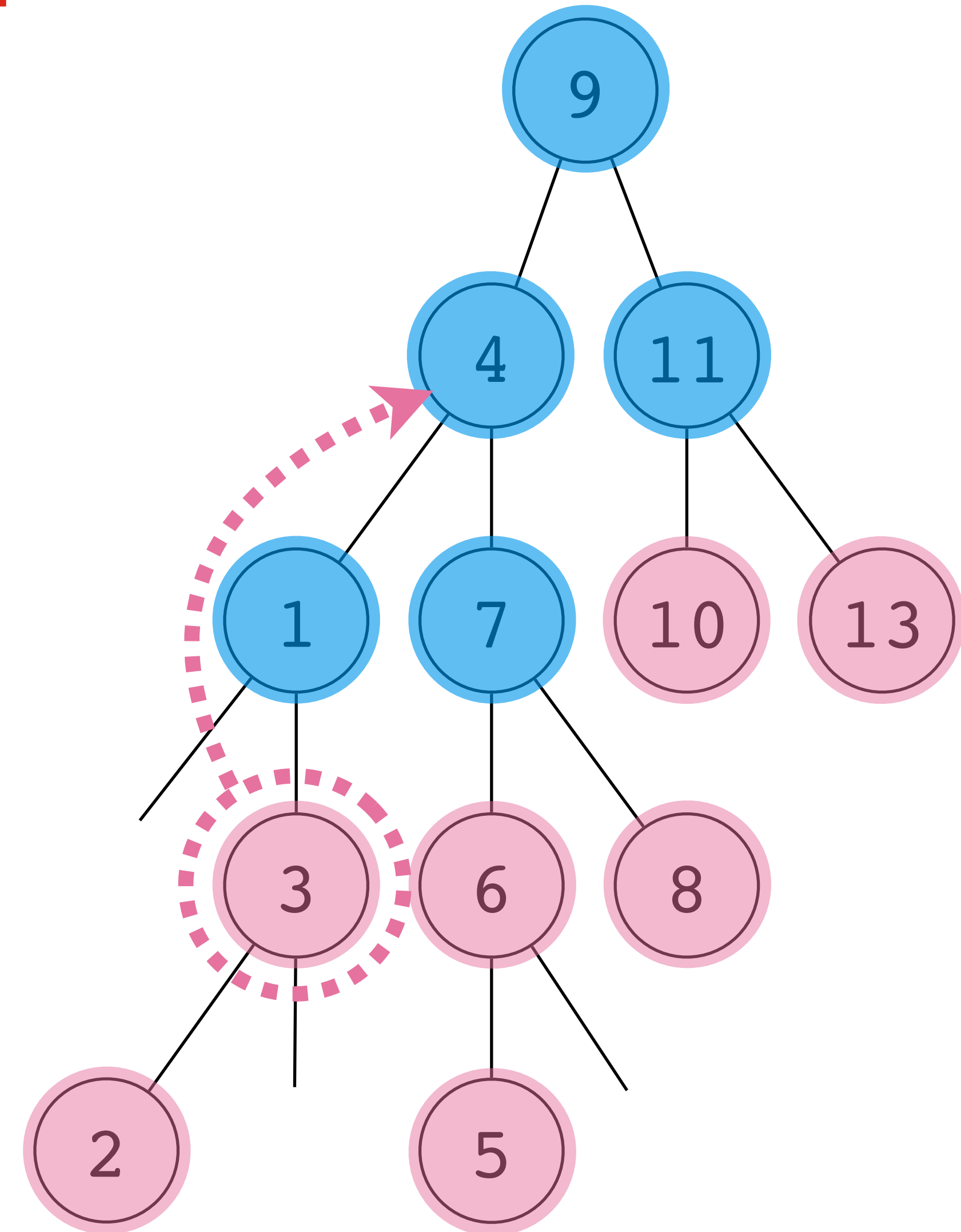


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r) en ordre symétrique

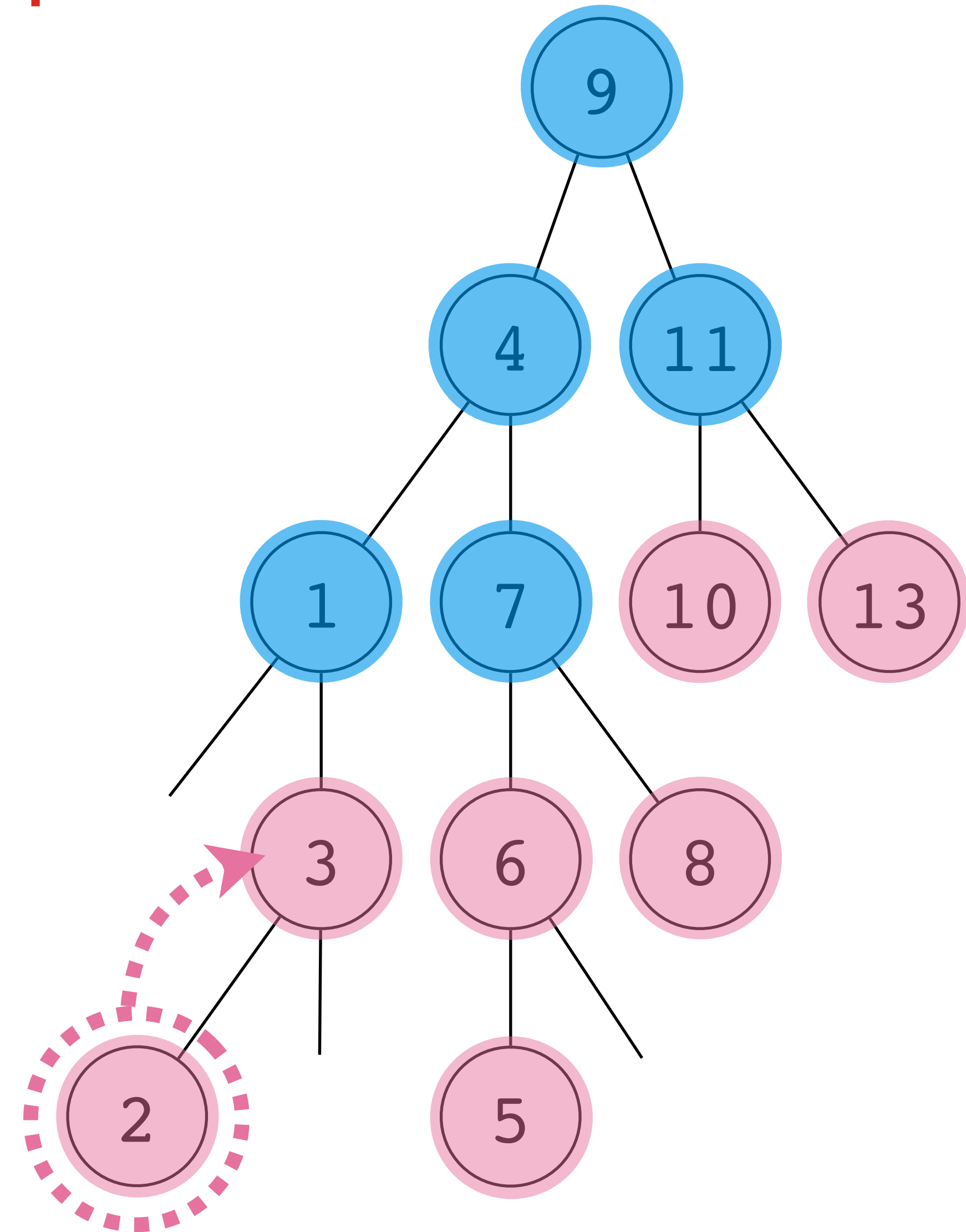


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche



suivant(r) en ordre symétrique

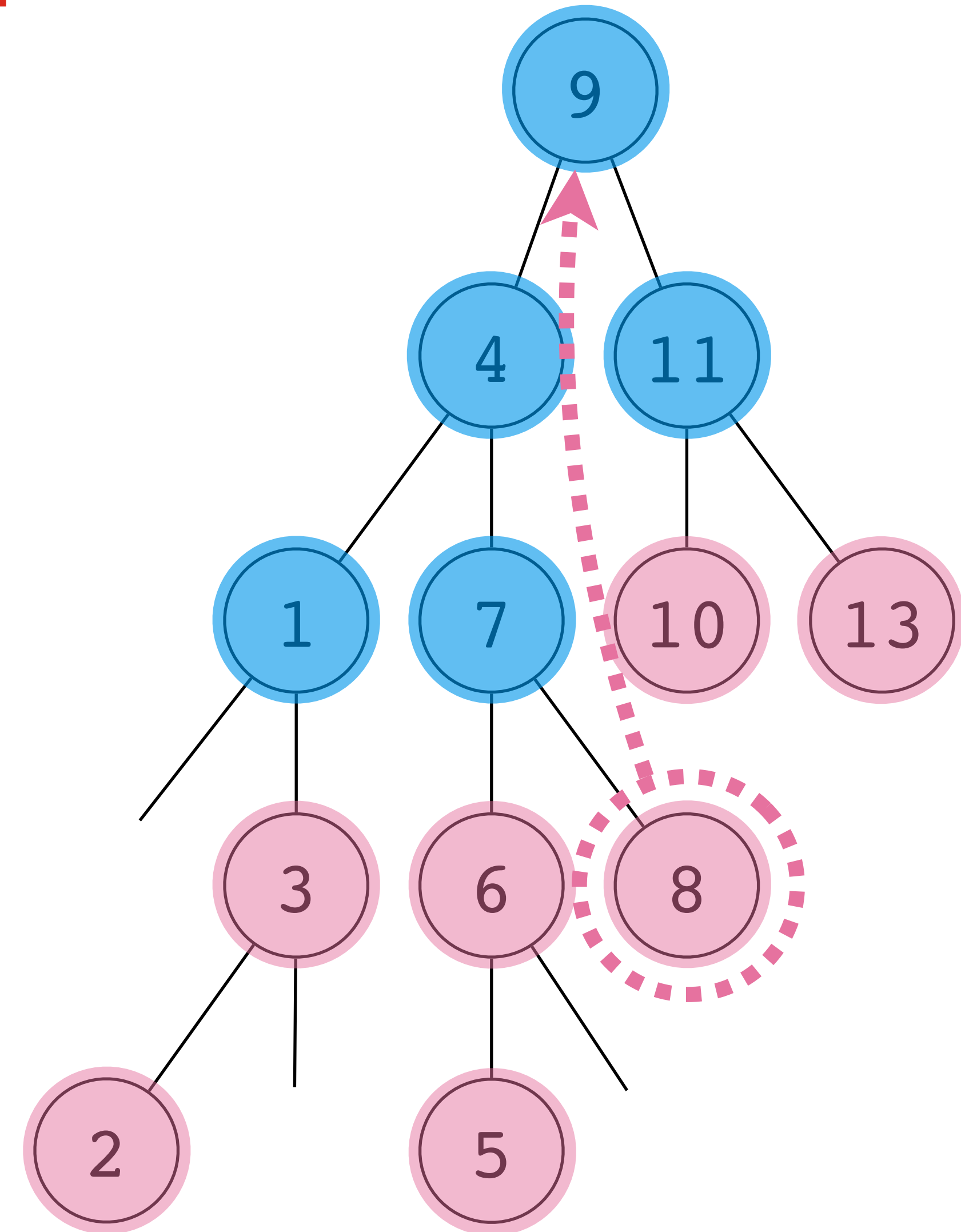


Si r a un sous-arbre droit

- $\text{suivant}(r)$ est le minimum du sous-arbre droit

Si r n'a pas de sous-arbre droit

- $\text{suivant}(r)$ est le premier ancêtre dont r est un descendant du sous-arbre gauche

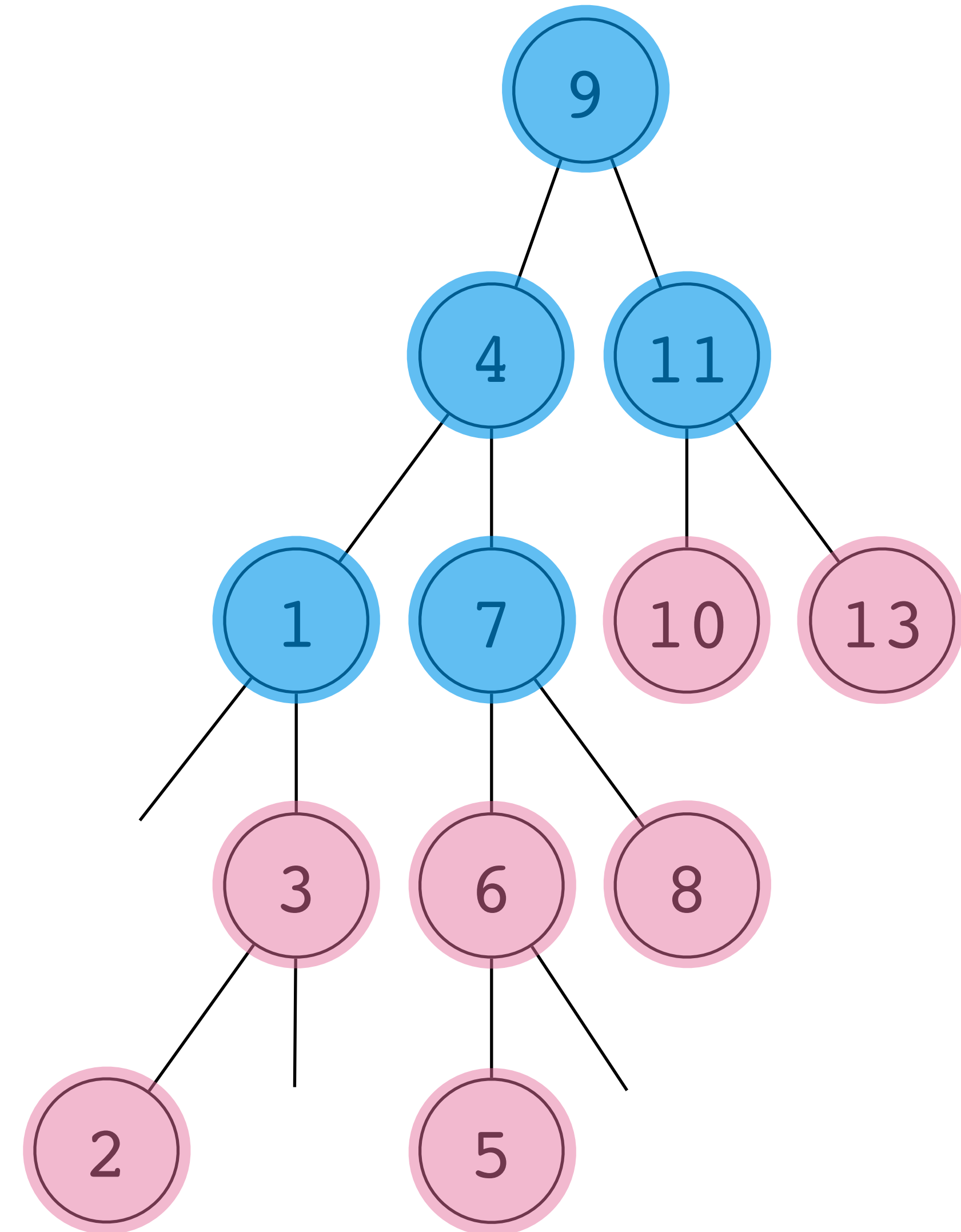


suivant en ordre symétrique



```
fonction suivant(r)
  si r.droit != ∅
    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 != ∅ et r == p.droit
```



Trace des 3 itérations



- Noeuds traversés par itération en profondeur

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

- Noeuds itérés en pré-ordre

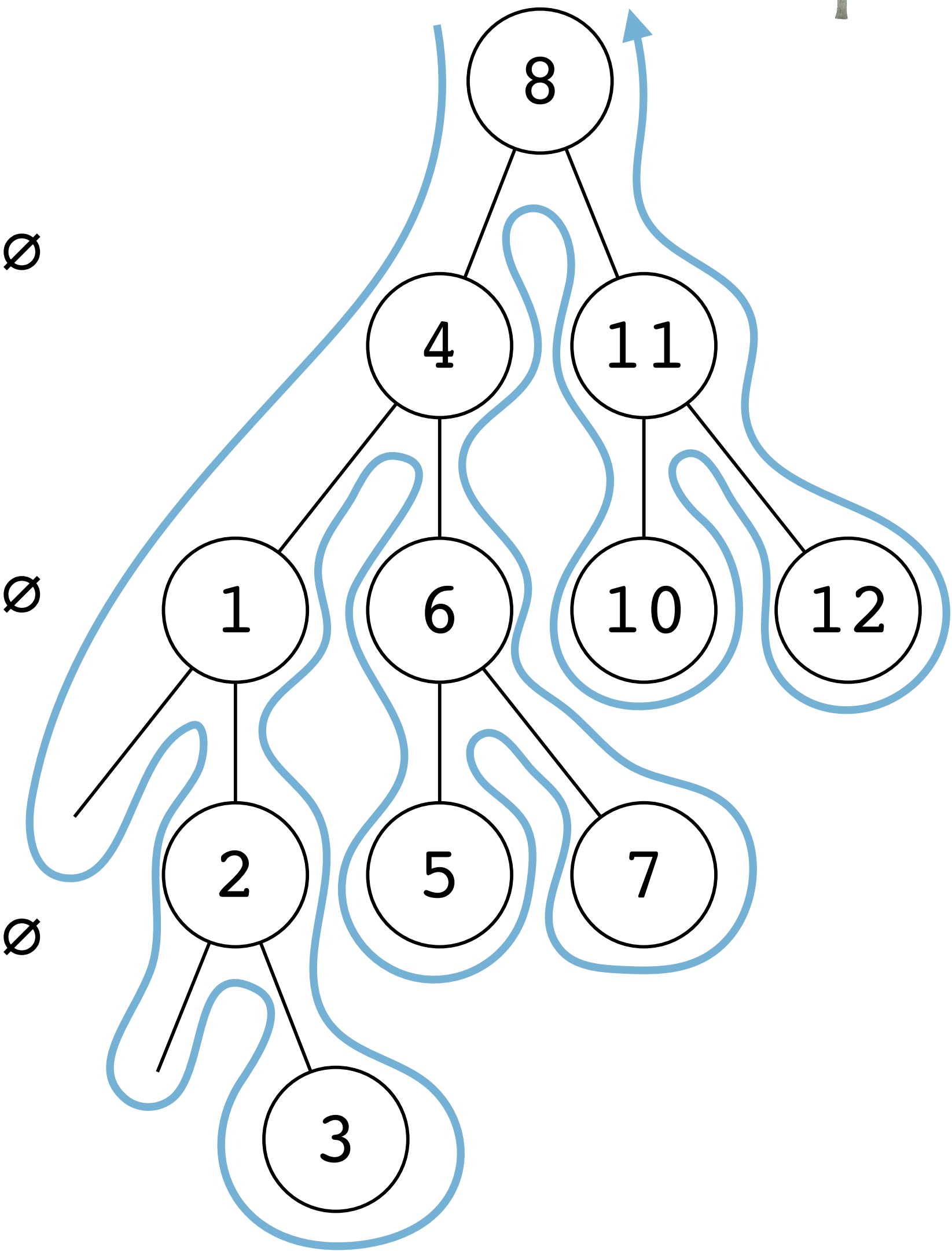
8-4-1-2-3-2-1-4-6-5-6-7-6-4-8-11-10-11-12-11-8-∅

- Noeuds itérés en post-ordre

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

- Noeuds itérés en 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, n^{ième} élément

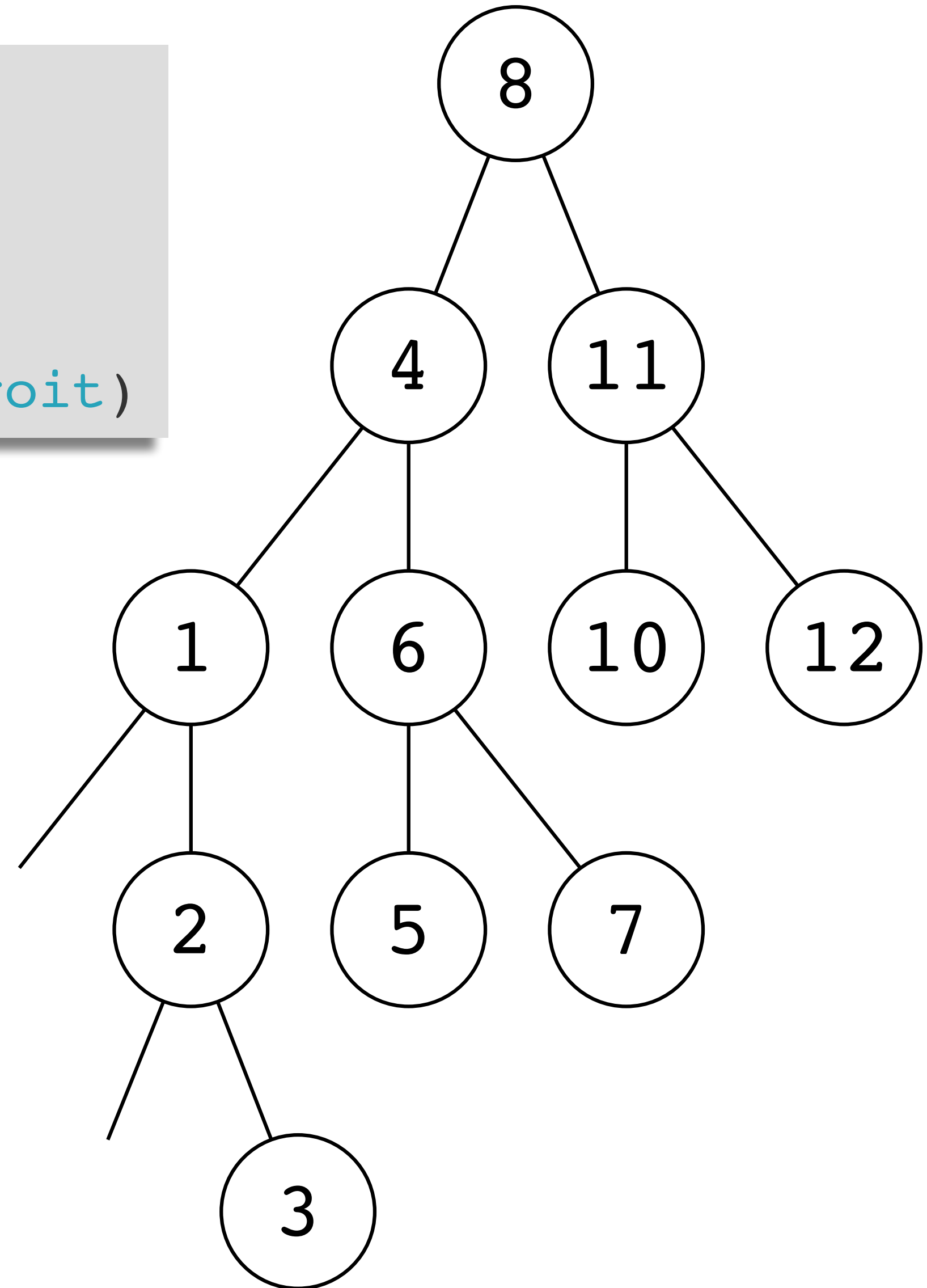


Taille = nombre de noeuds d'un arbre

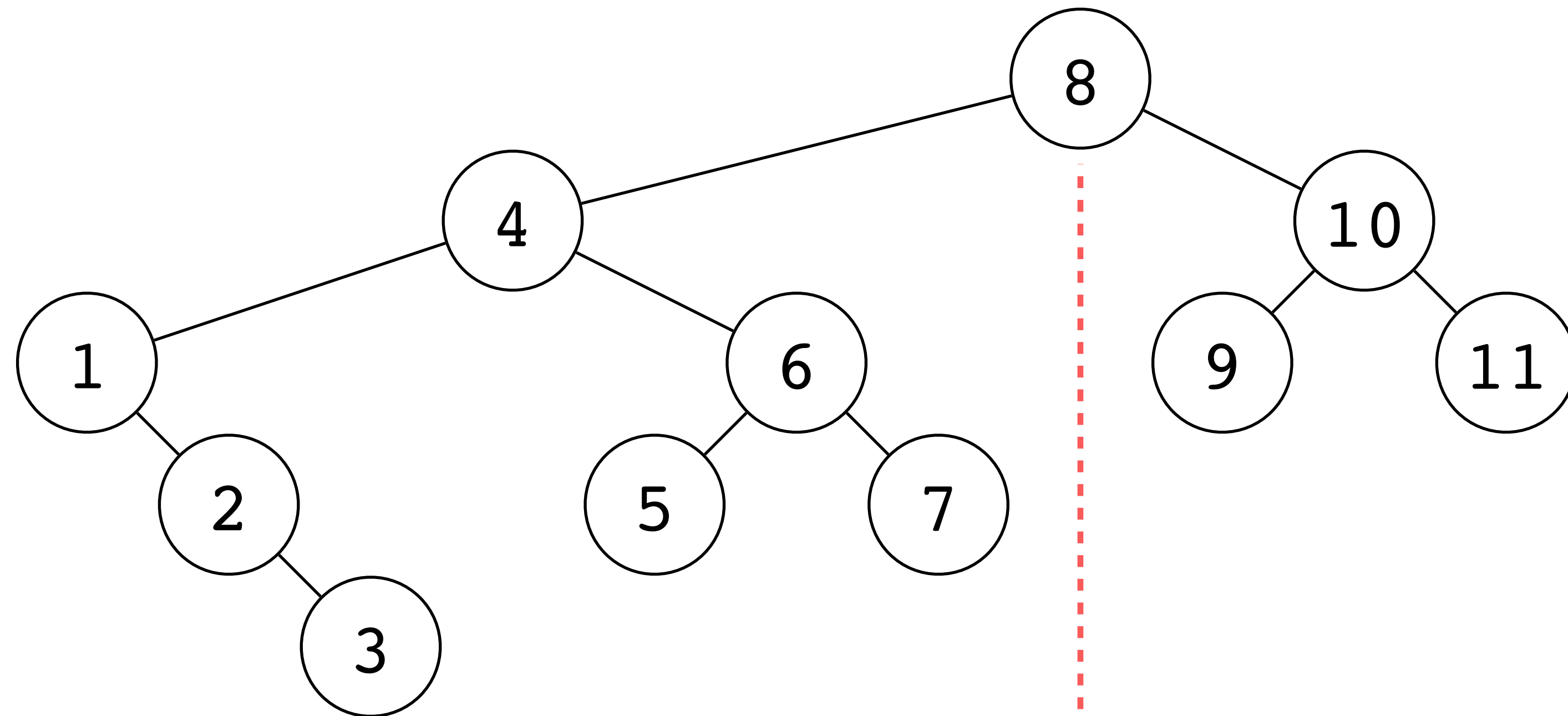


```
fonction taille (r)
  si r == ∅,
    retourner 0
  sinon,
    retourner 1 + taille(r.gauche) + taille(r.droit)
```

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

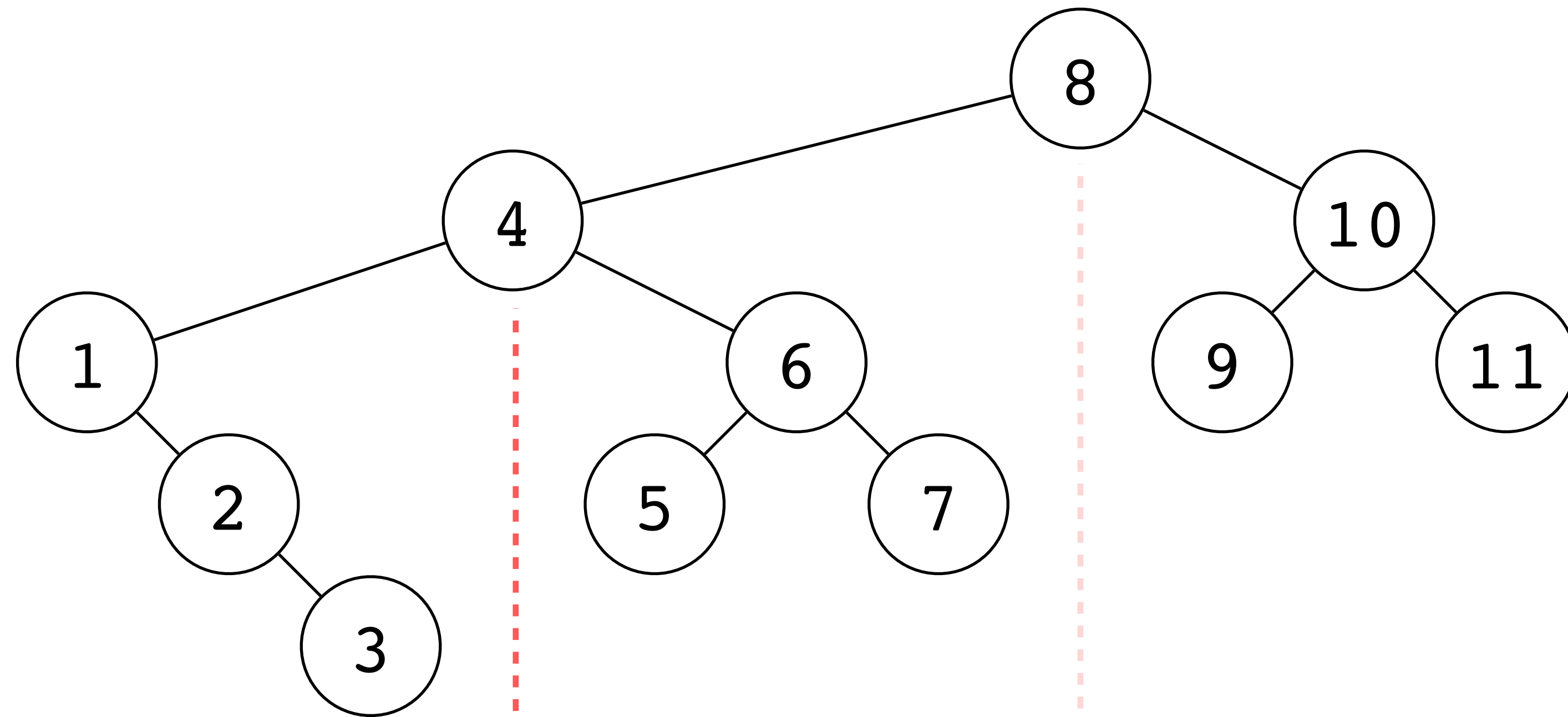


Représentation visuelle



$$Taille(8) = \underbrace{\hspace{15em}}_{Taille(4)} + \underbrace{1}_{1} + \underbrace{\hspace{10em}}_{Taille(10)}$$

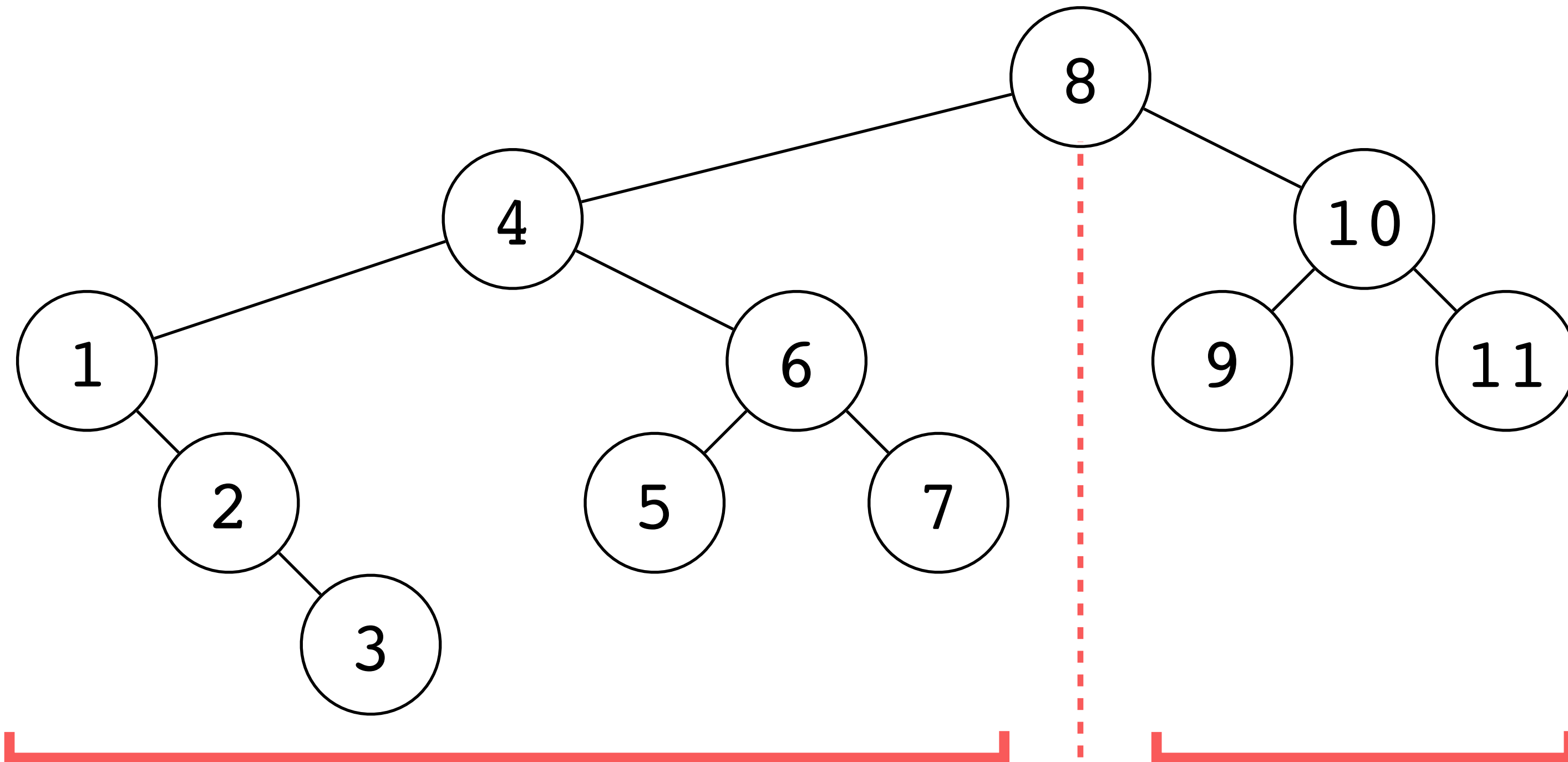
Représentation visuelle



$$Taille(8) = \underbrace{\hspace{10em}}_{Taille(4)} + 1 + \underbrace{\hspace{10em}}_{Taille(10)}$$

$$Taille(4) = \underbrace{\hspace{10em}}_{Taille(1)} + 1 + \underbrace{\hspace{10em}}_{Taille(6)}$$

Représentation visuelle



$$\text{Taille}(8) = \text{Taille}(4) + 1 + \text{Taille}(10)$$

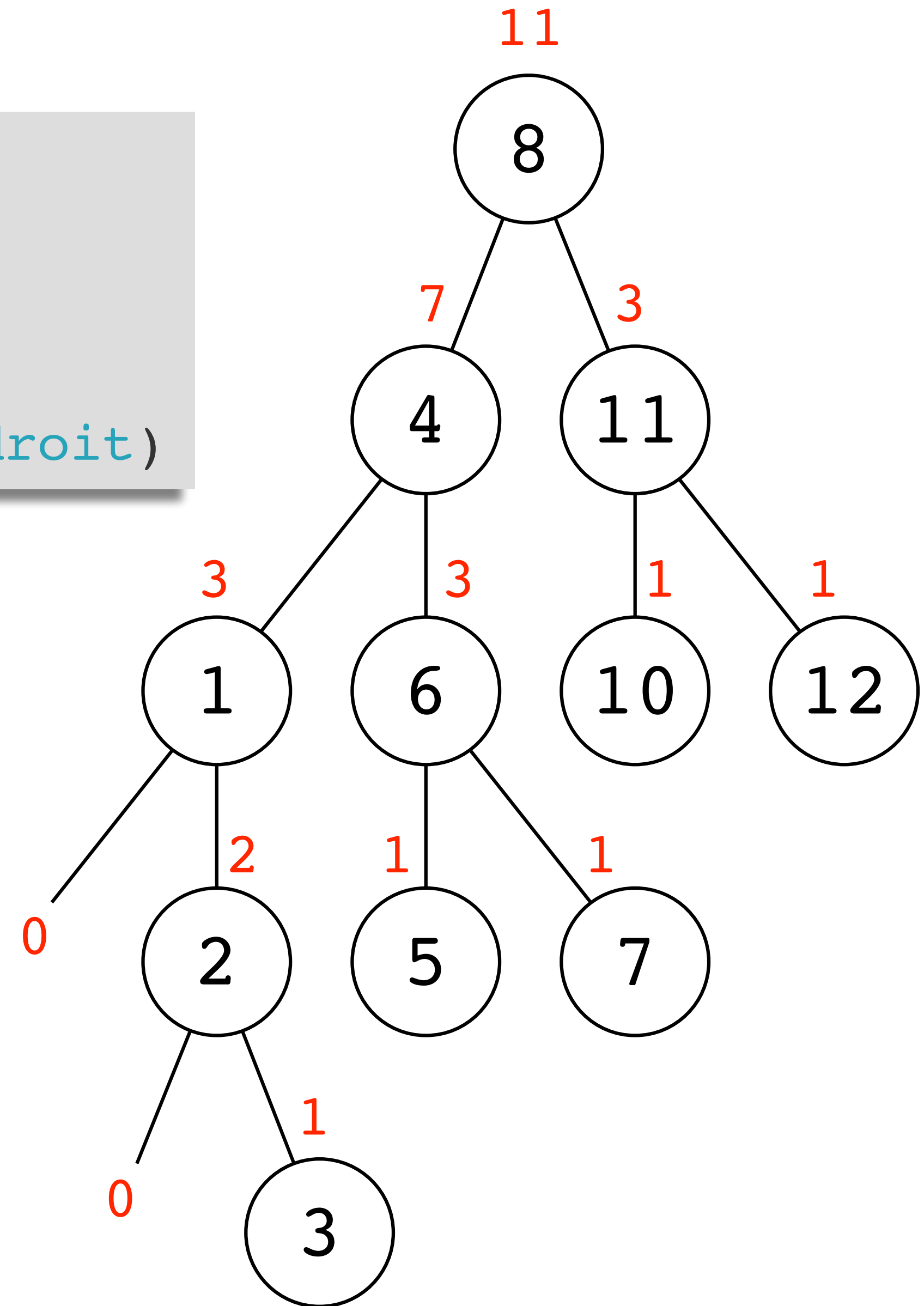
```
fonction taille (r)
    si r == ∅,
        retourner 0
    sinon,
        retourner 1 + taille(r.gauche) + taille(r.droit)
```

Taille = nombre de noeuds d'un arbre



```
fonction taille (r)
  si r == ∅,
    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





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

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

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

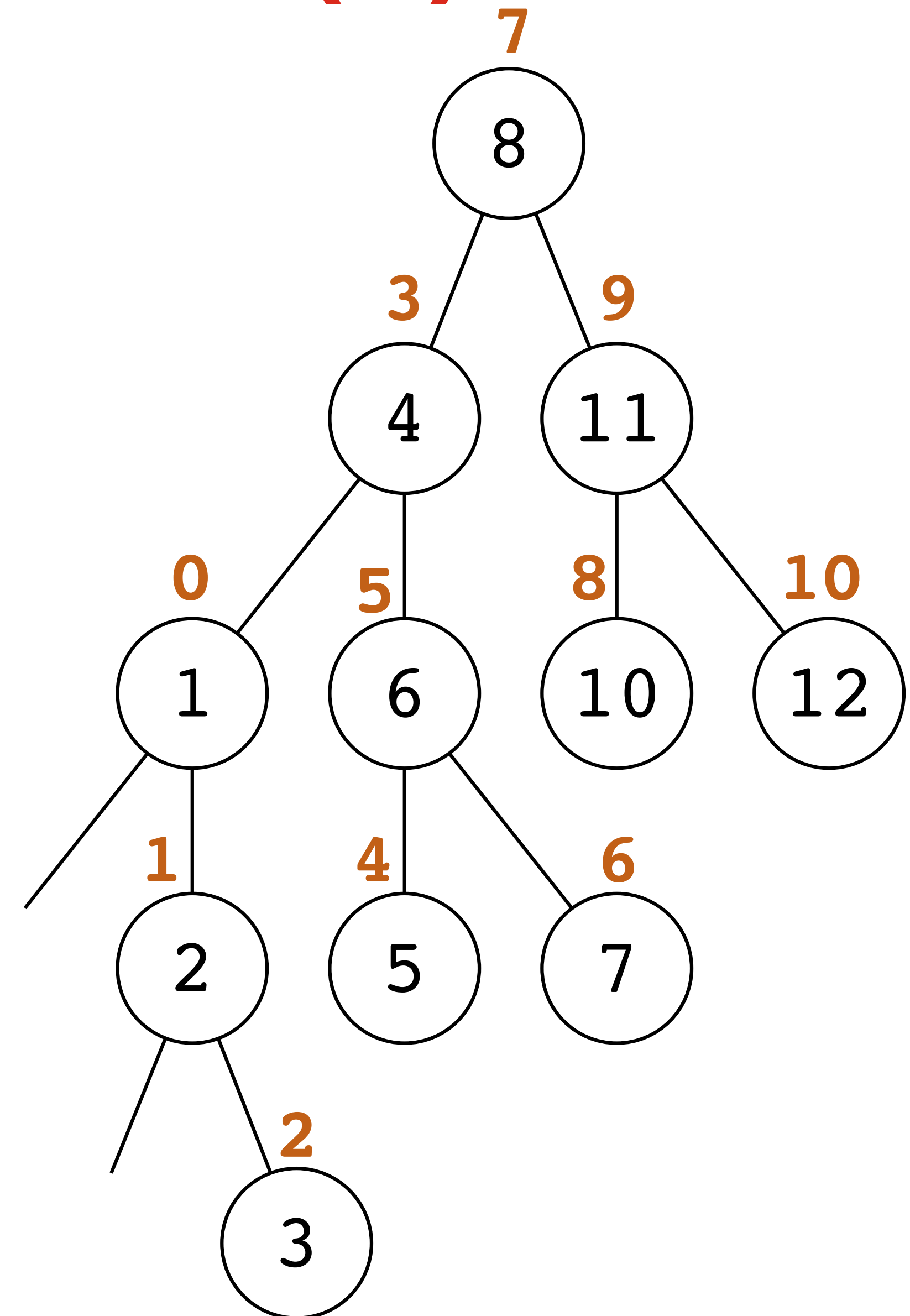
```
r.taille ← taille(r.gauche)  
          + taille(r.droit)  
          + 1
```

Rang d'une clé k (1)

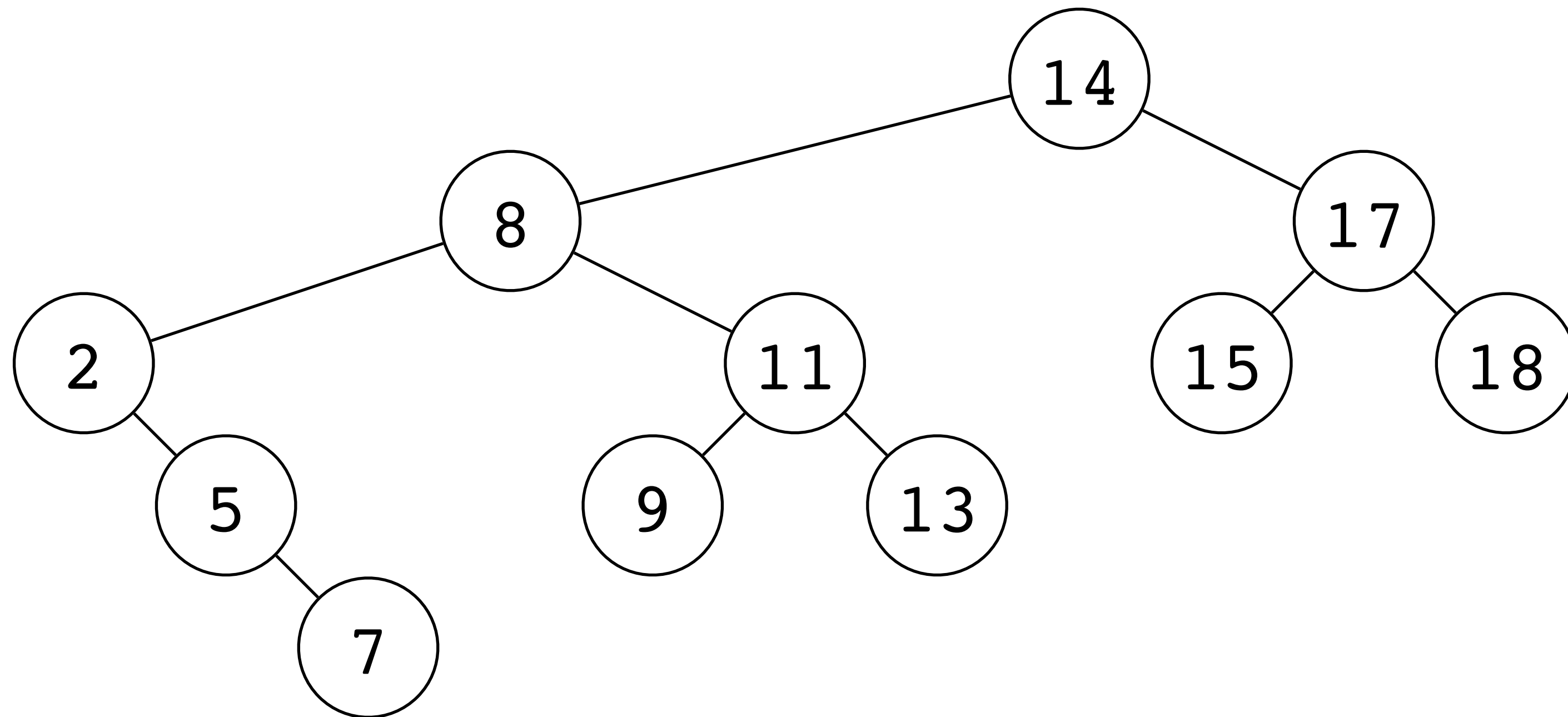


Définition - Rang

Nombre d'éléments strictement plus petits que la clé

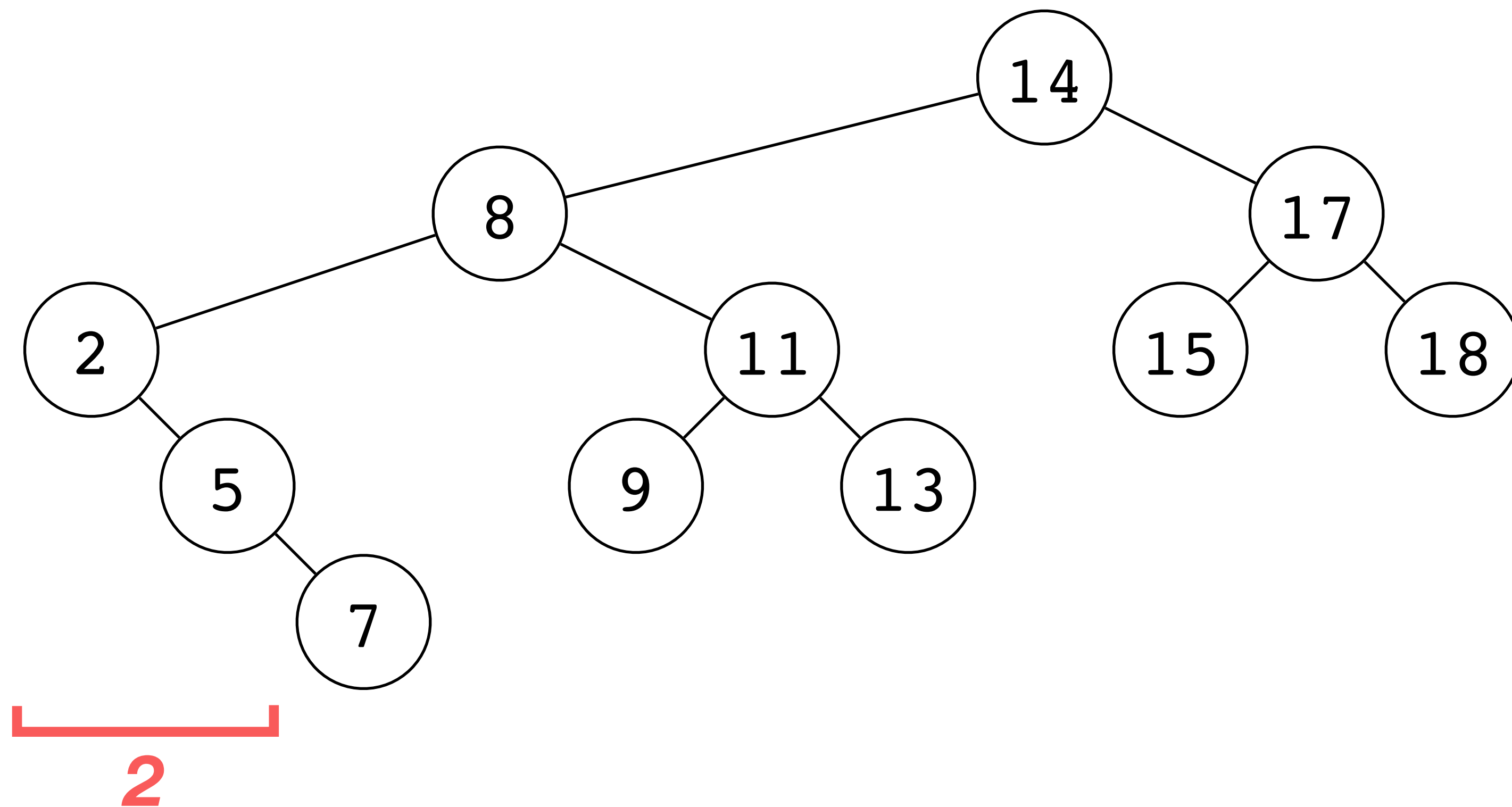


Représentation visuelle



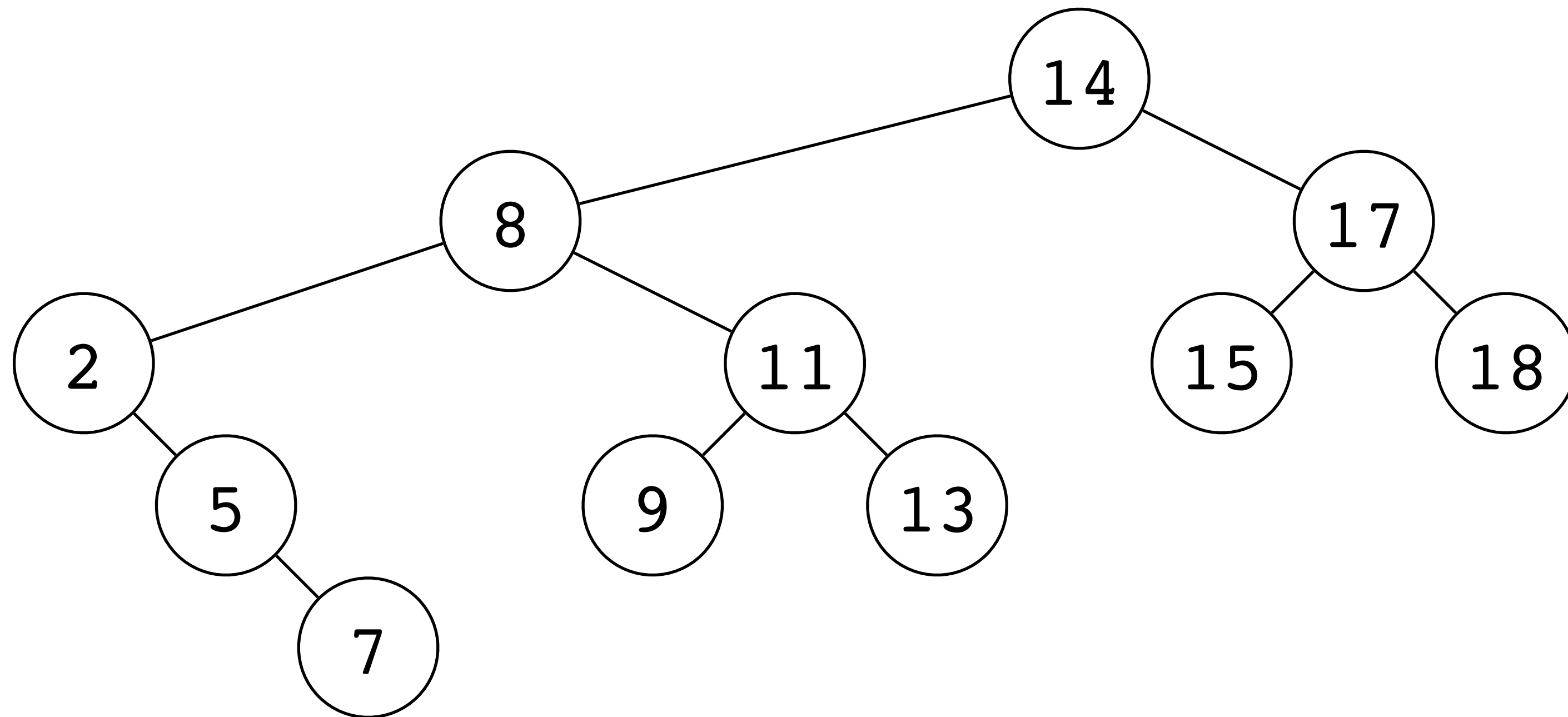
Rang de 7 : ? clés plus petites que 7

Représentation visuelle



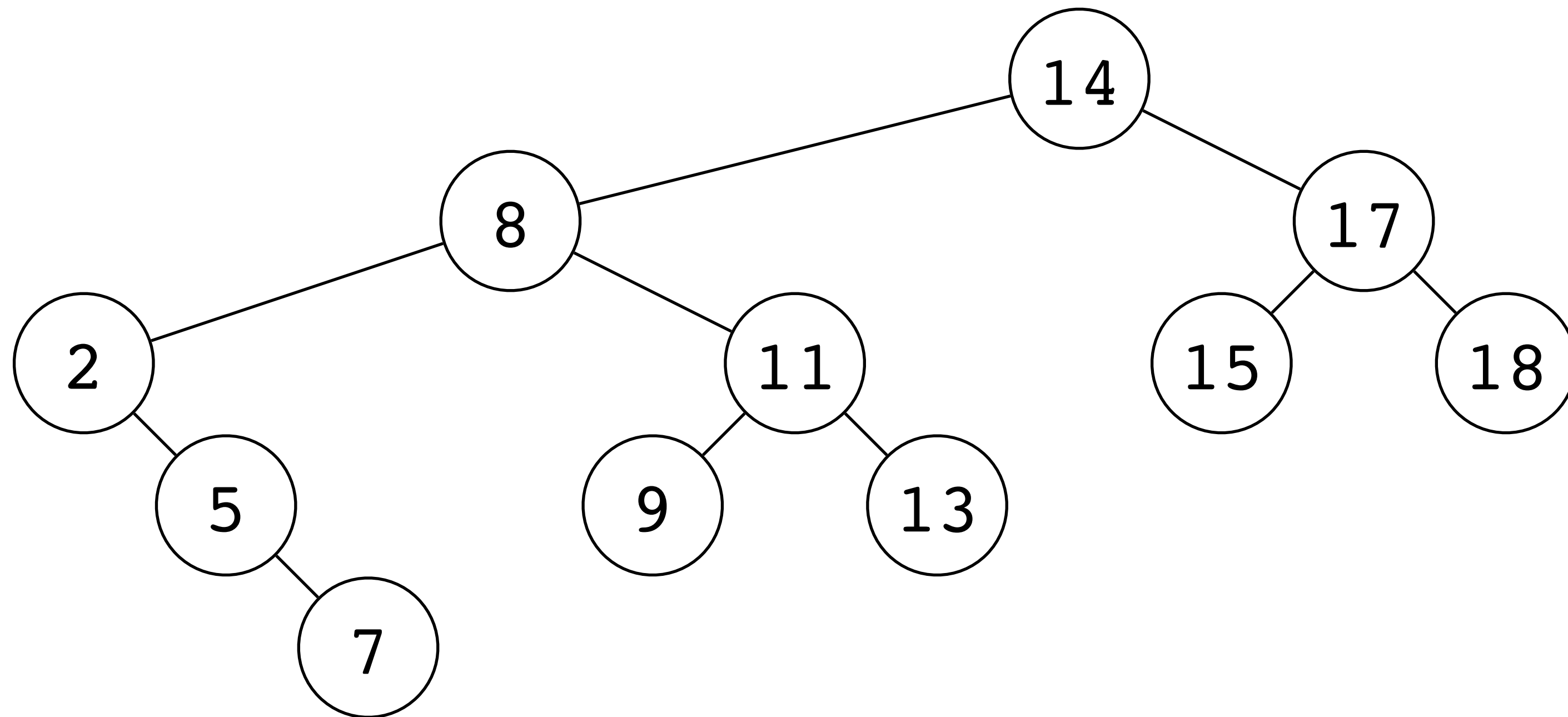
Rang de 7 : 2 clés plus petites que 7

Représentation visuelle



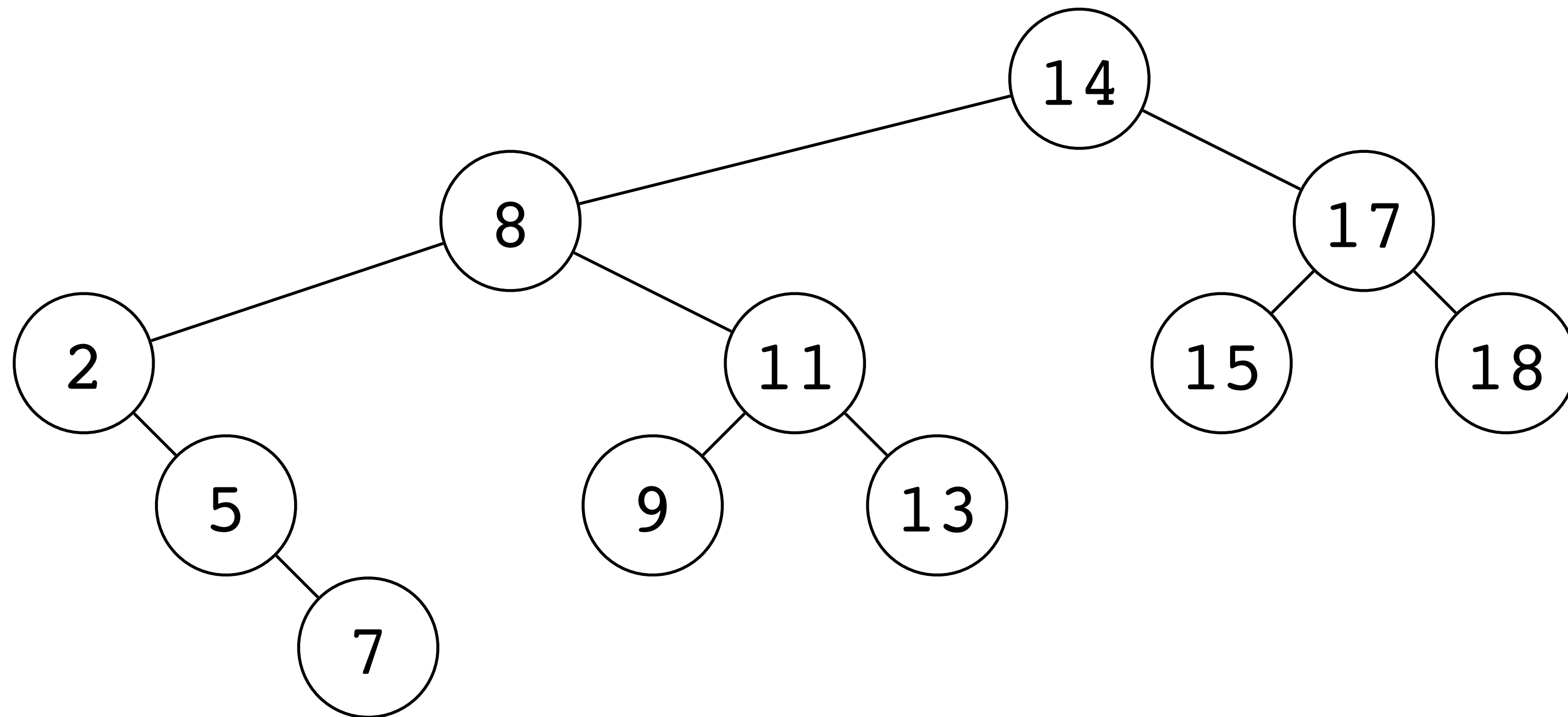
Rang de 2 : ? clés plus petites que 2

Représentation visuelle



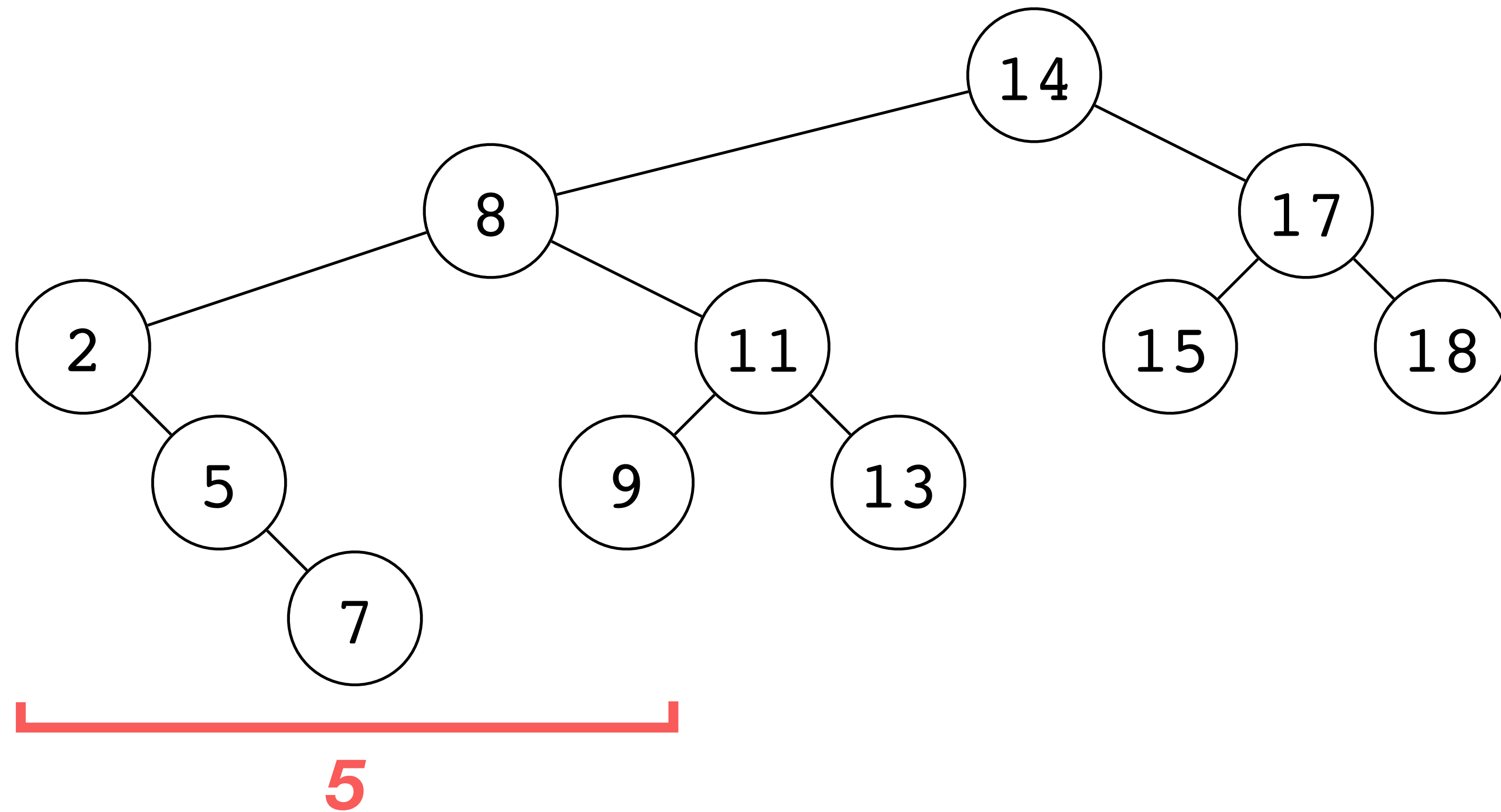
0

Rang de 2 : 0 clés plus petites que 2



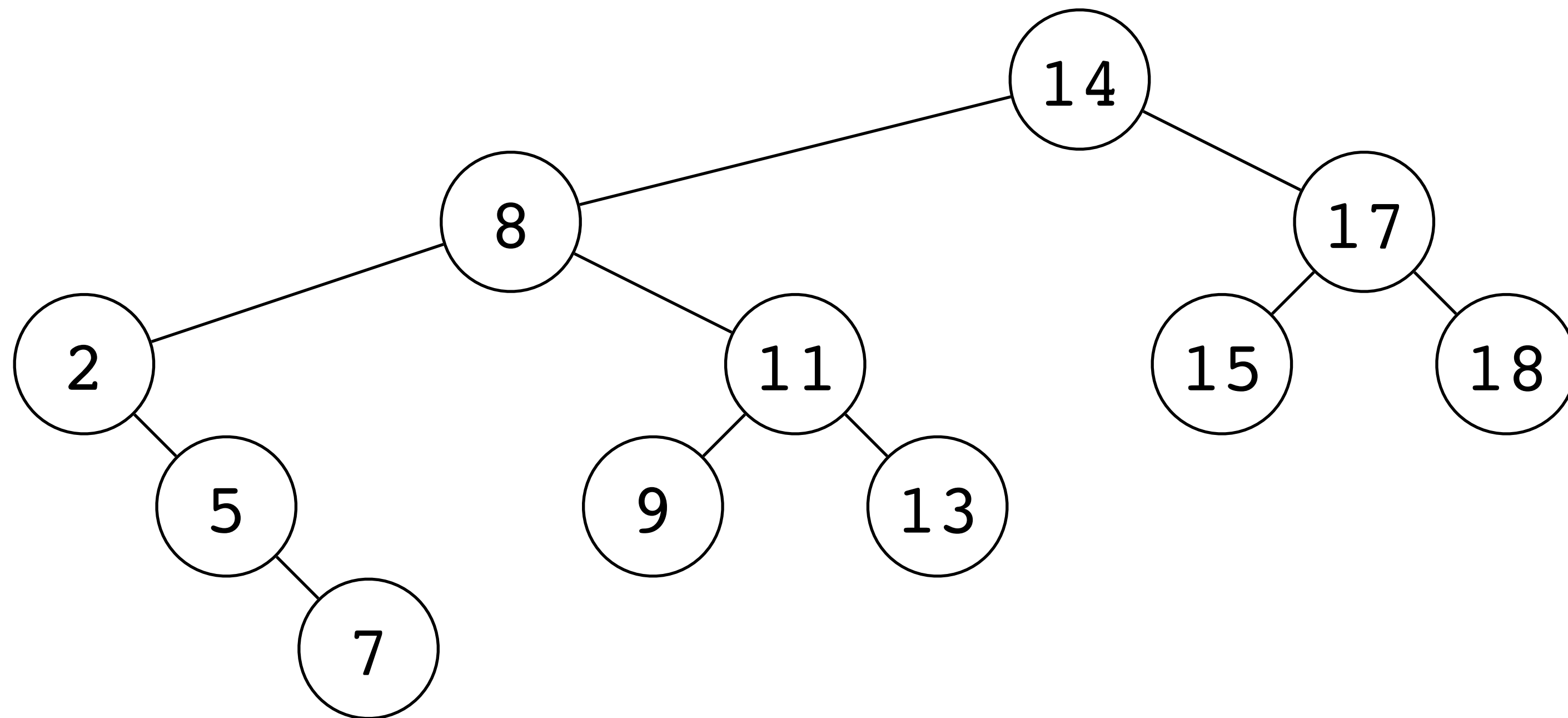
Rang de 11 : ? clés plus petites que 11

Représentation visuelle



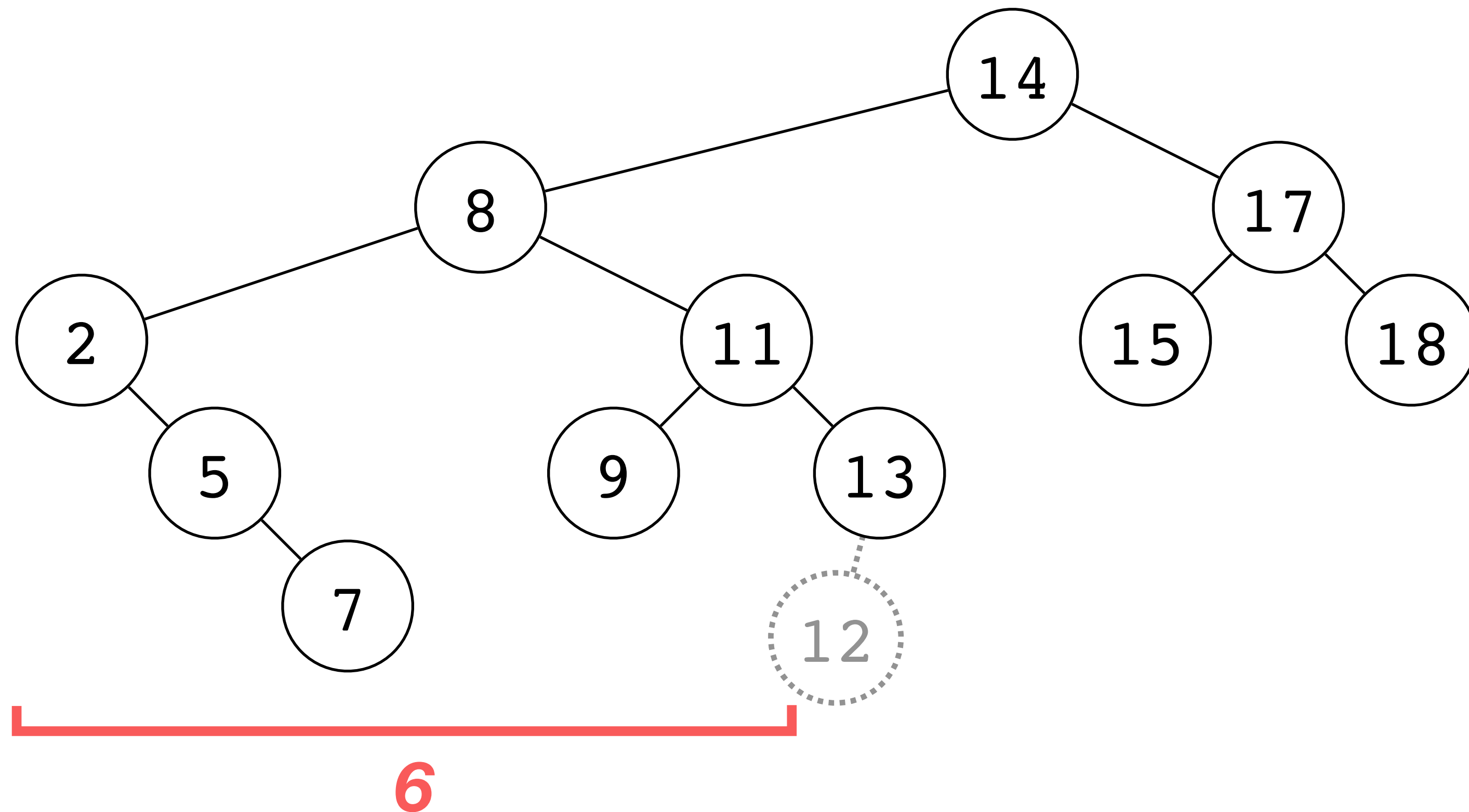
Rang de 11 : 5 clés plus petites que 11

Représentation visuelle



Rang de 12 : ? clés plus petites que 12

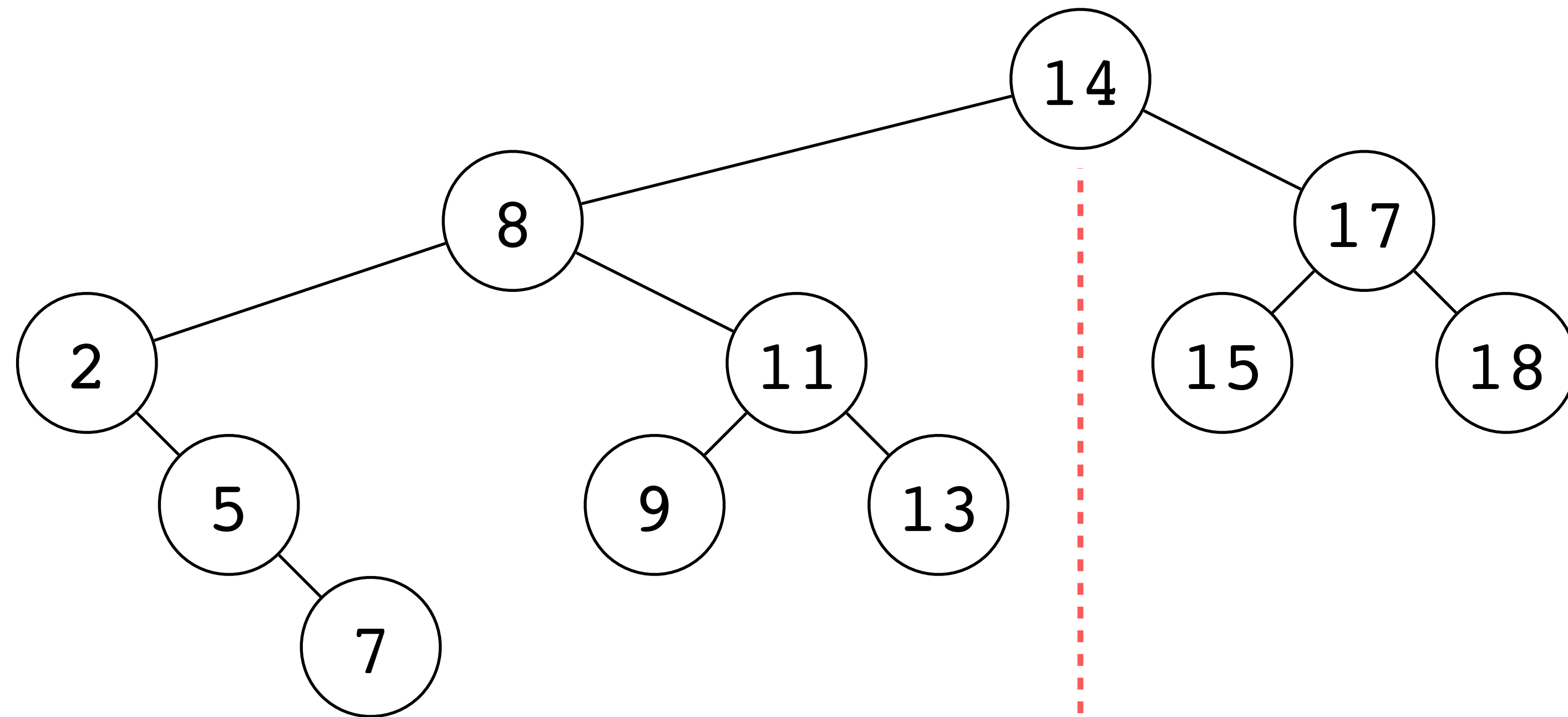
Représentation visuelle



Rang de 12 : 6 clés plus petites que 12

Toute clé a un rang, même si elle n'est pas présente dans l'arbre.

Comment calculer le rang ?



Racine
↓

Clé
↓

$Rang(14, x) =$

“Nombre de clés $< x$
dans le sous-arbre **14**”

$Rang(8, x)$

“Nombre de clés $< x$
dans le sous-arbre **8**”

+

$1 + Rang(17, x)$

ssi $14 < x$
(0 sinon)

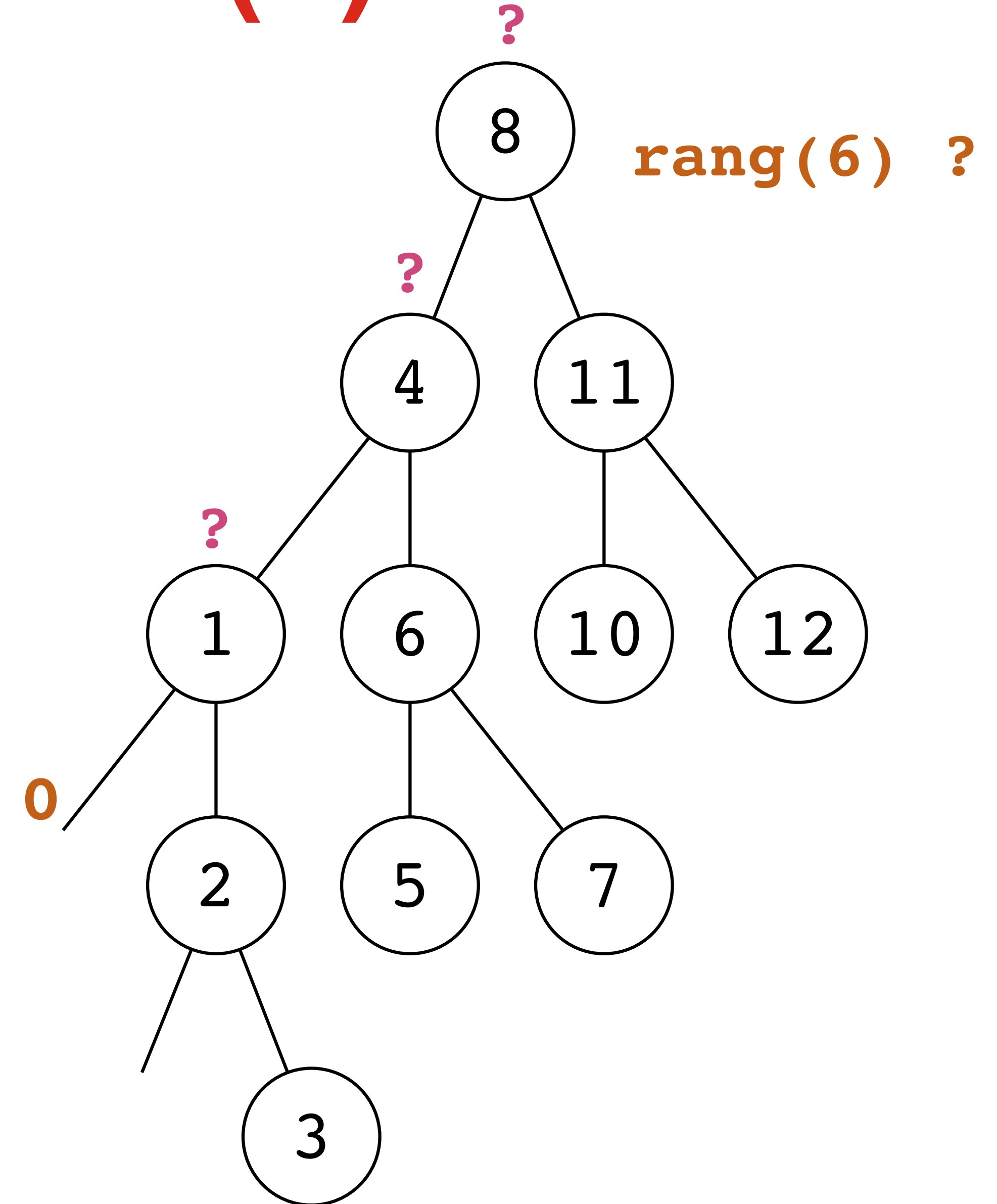
“Nombre de clés $< x$
dans le sous-arbre **17**”

Rang d'une clé k (2)



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

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

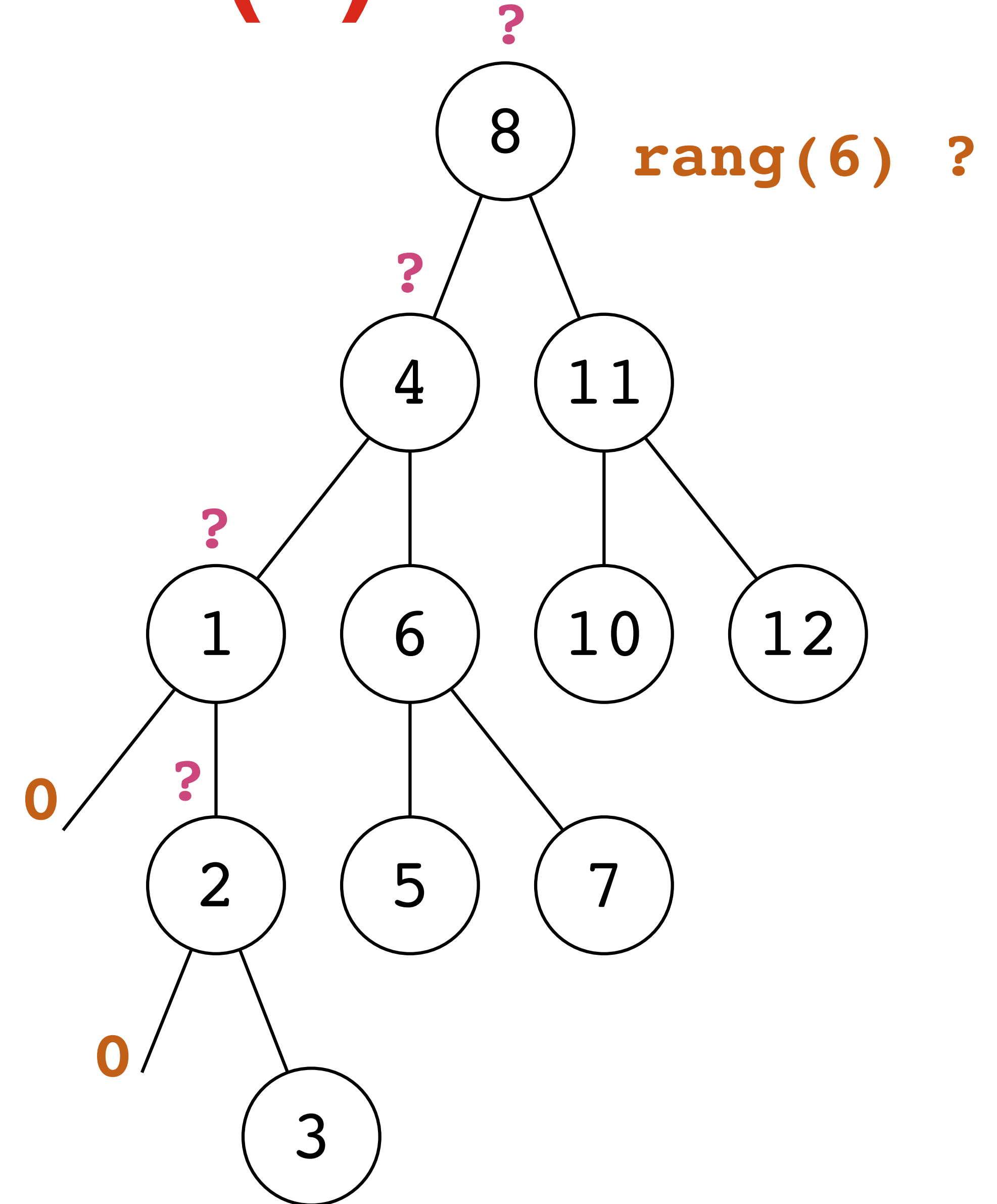


Rang d'une clé k (2)



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

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

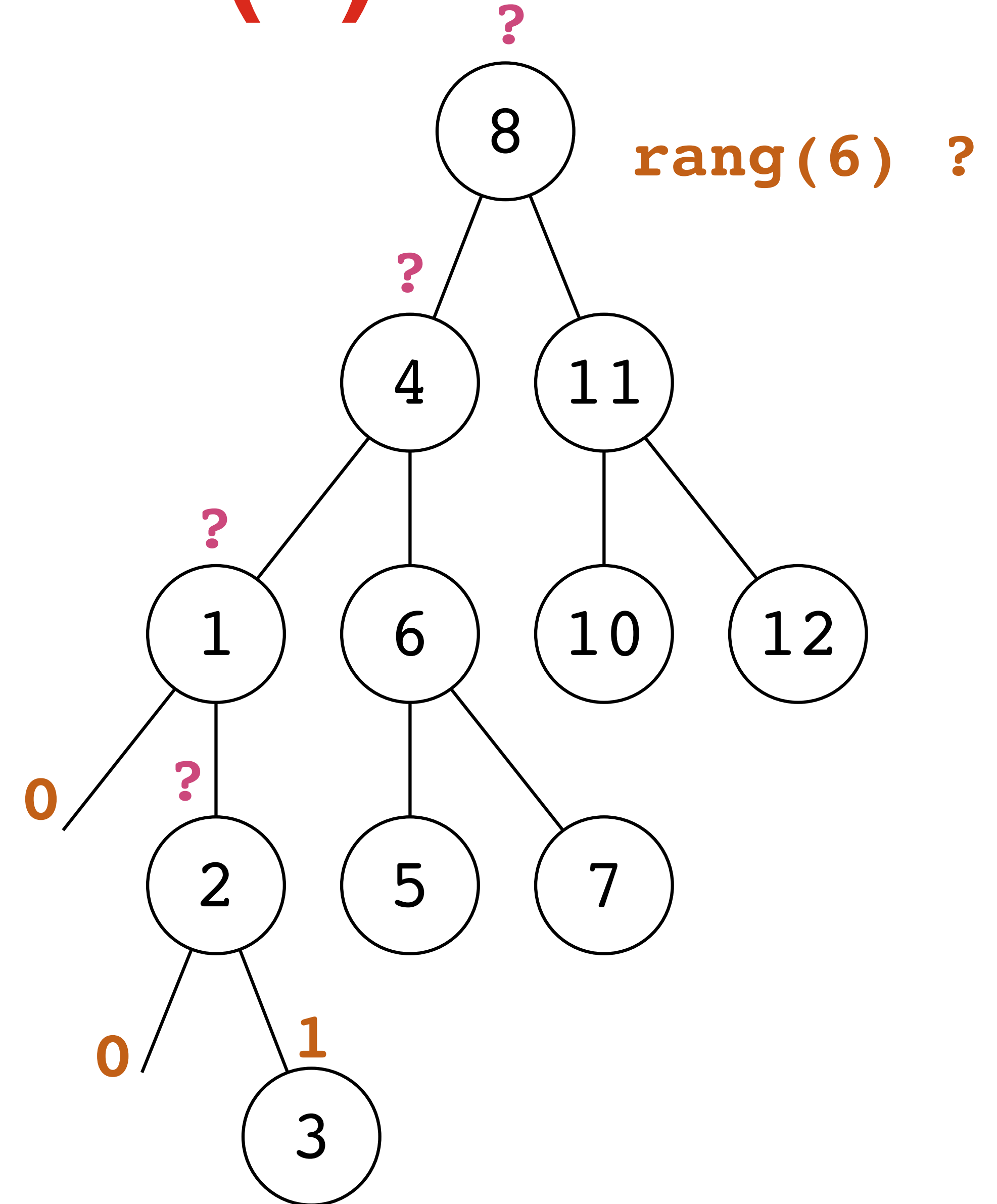


Rang d'une clé k (2)



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

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

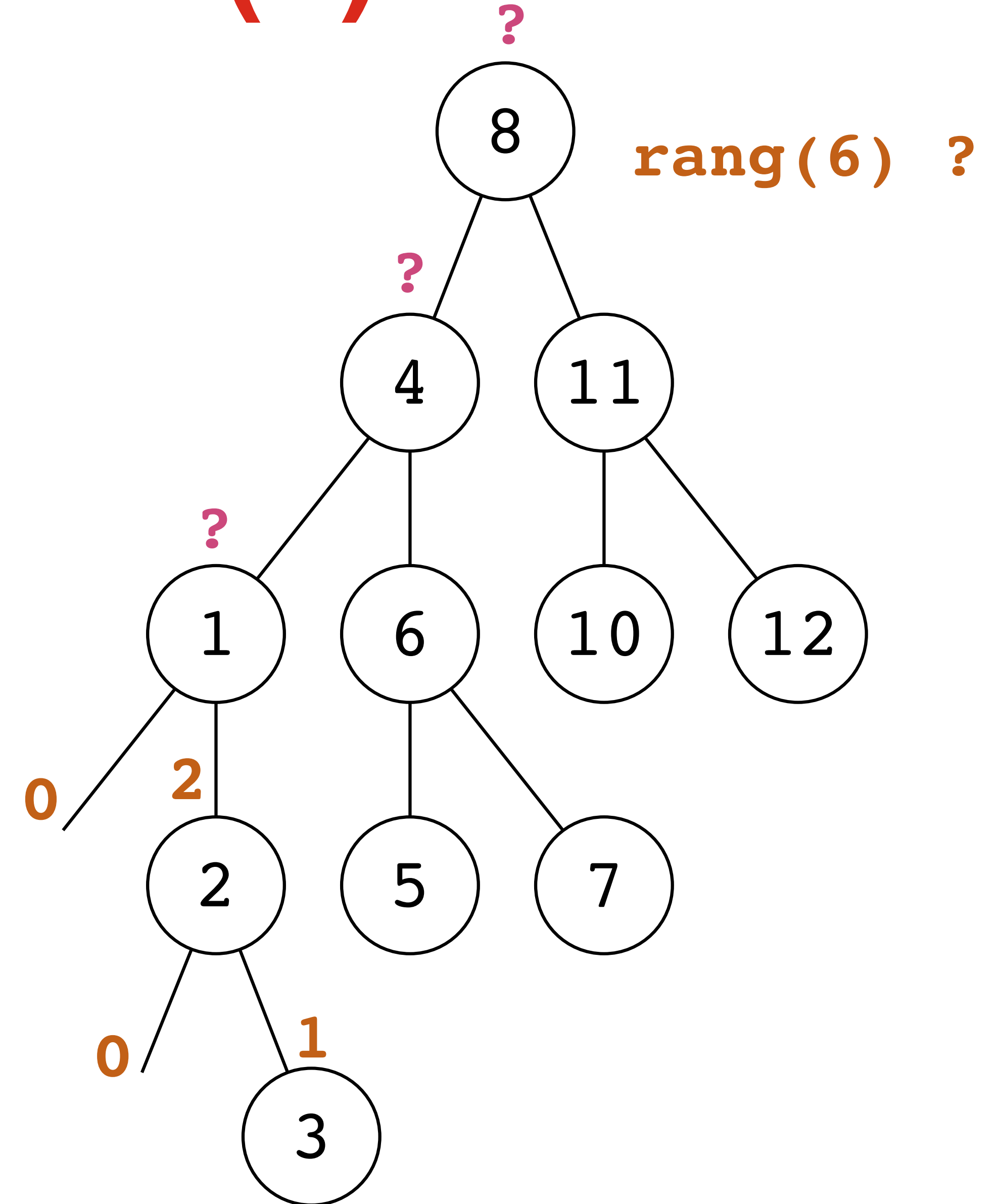


Rang d'une clé k (2)



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

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

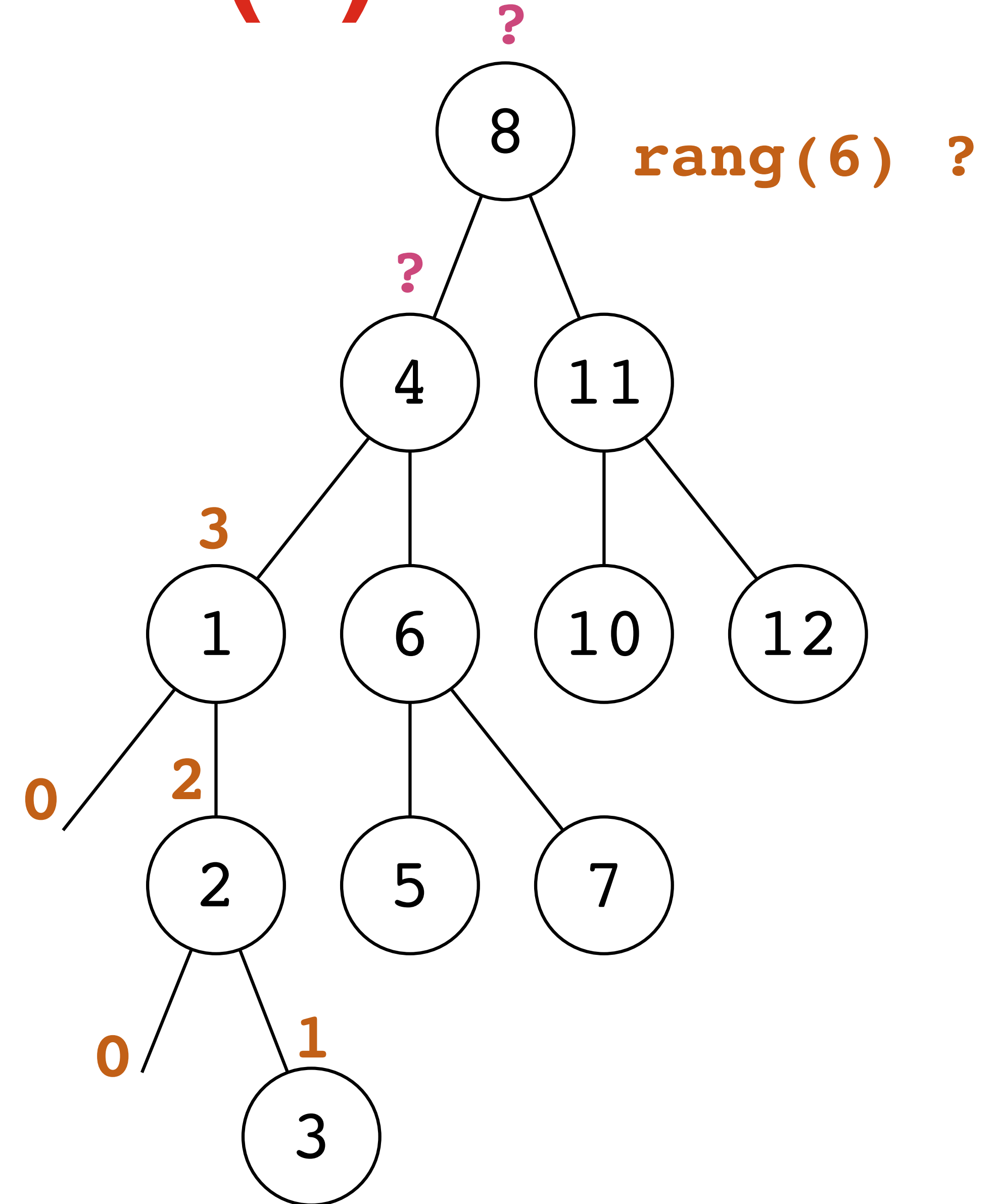


Rang d'une clé k (2)



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

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

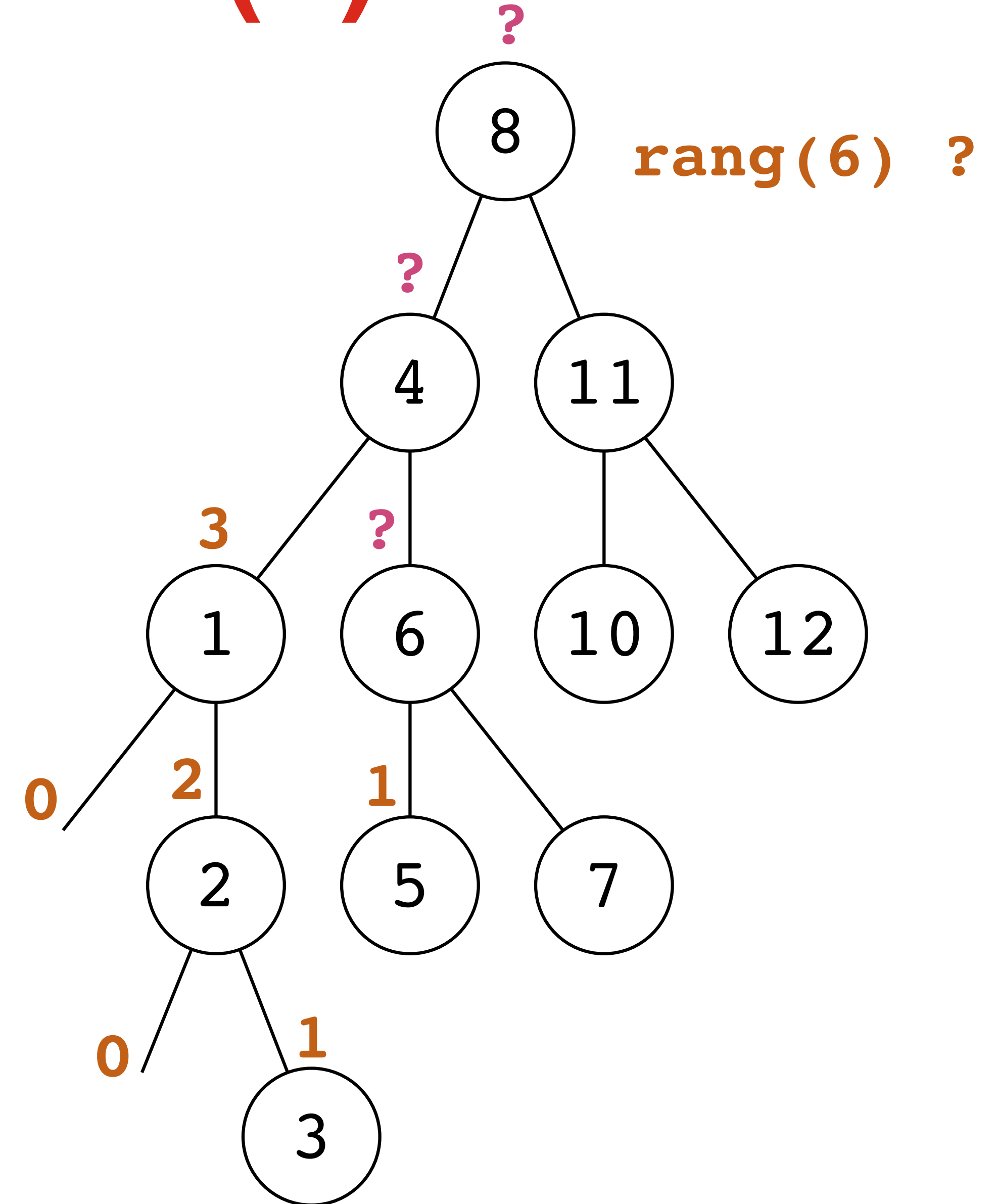


Rang d'une clé k (2)



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

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



rang(6) ?

```

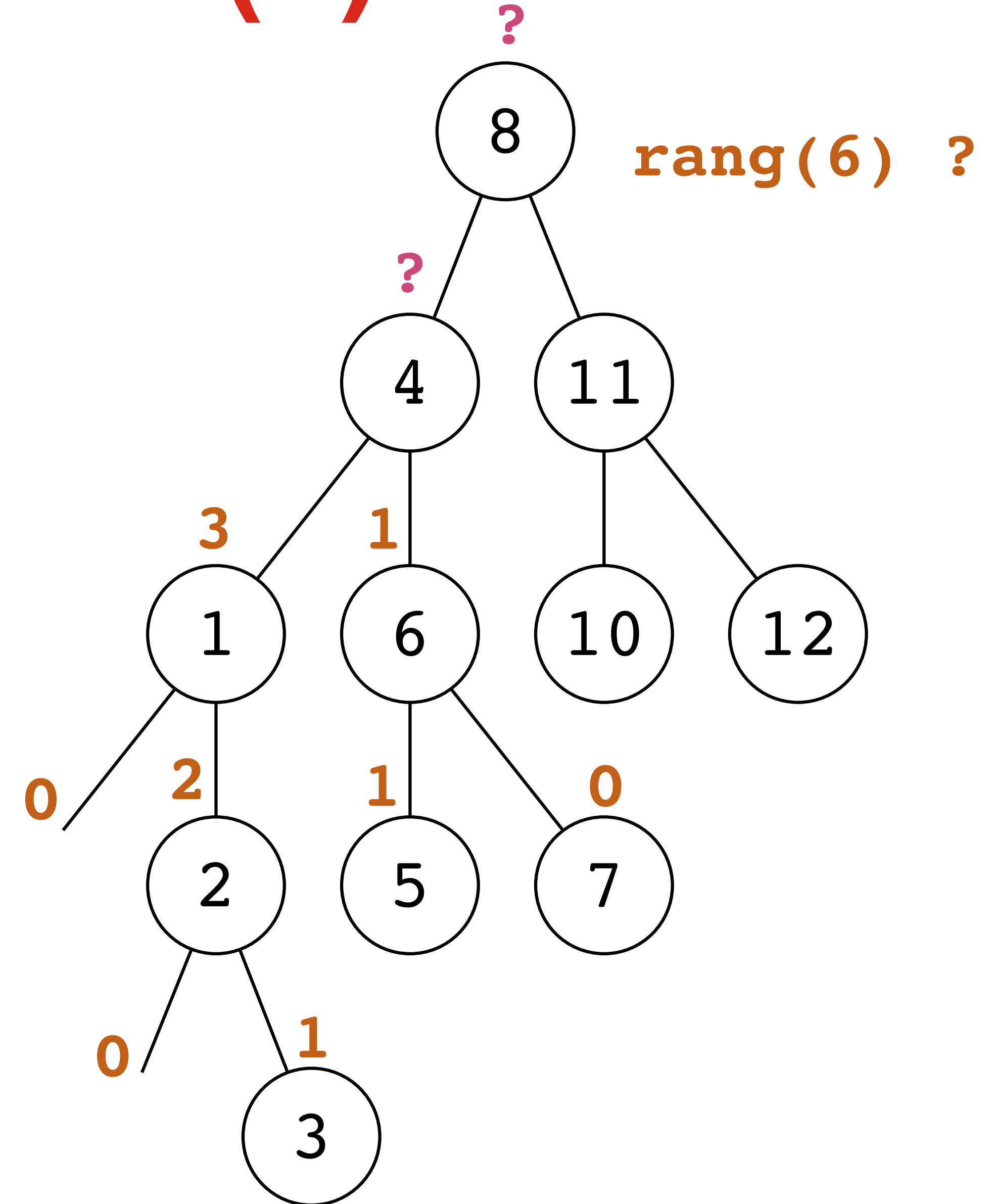
graph TD
    8((8)) --- 4((4))
    8 --- 11((11))
    4 --- 1((1))
    4 --- 6((6))
    1 --- 2((2))
    2 --- 0((0))
    2 --- 3((3))
    6 --- 5((5))
    6 --- 7((7))
    11 --- 10((10))
    11 --- 12((12))
  
```

Rang d'une clé k (2)



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

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

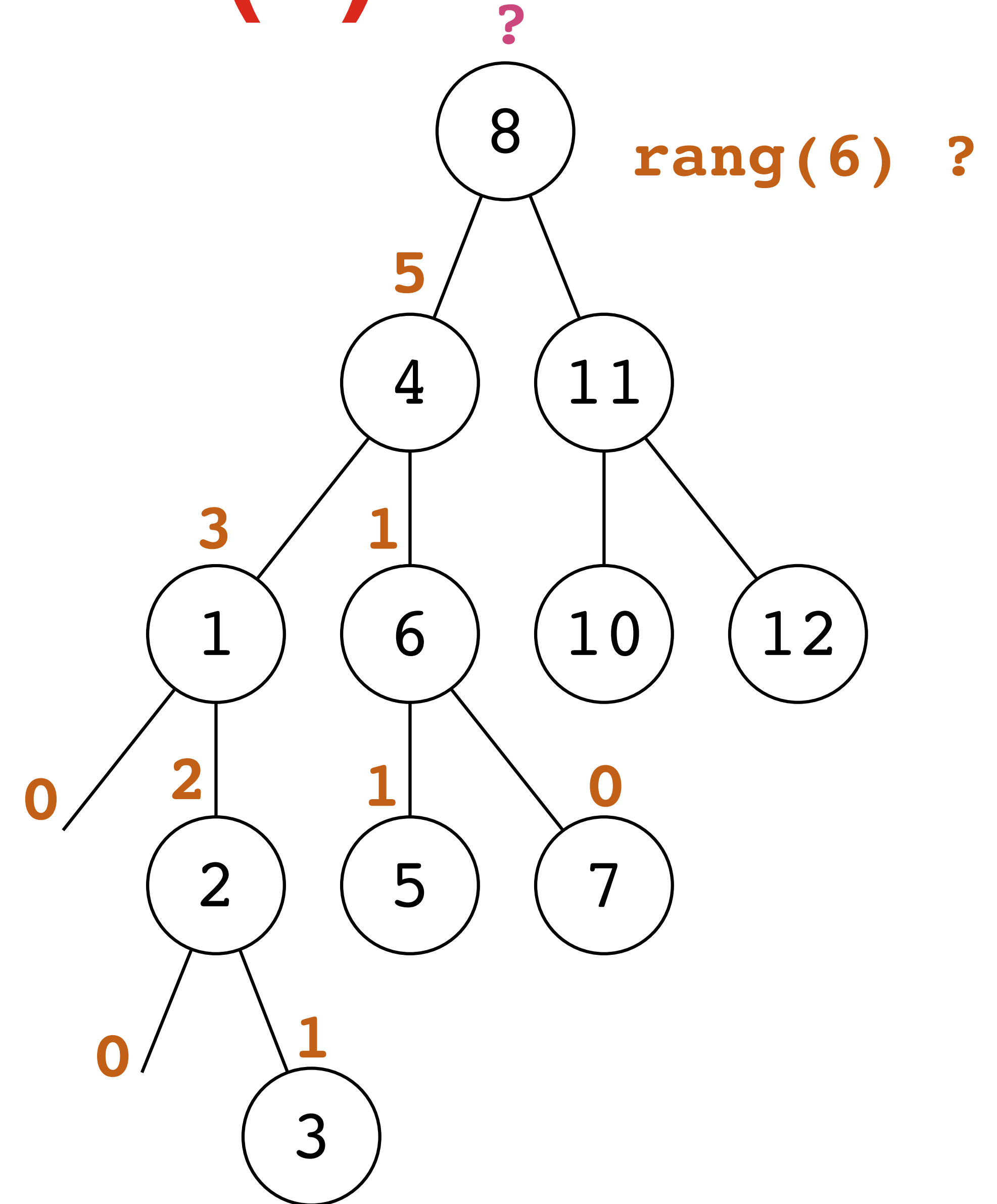


Rang d'une clé k (2)



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

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

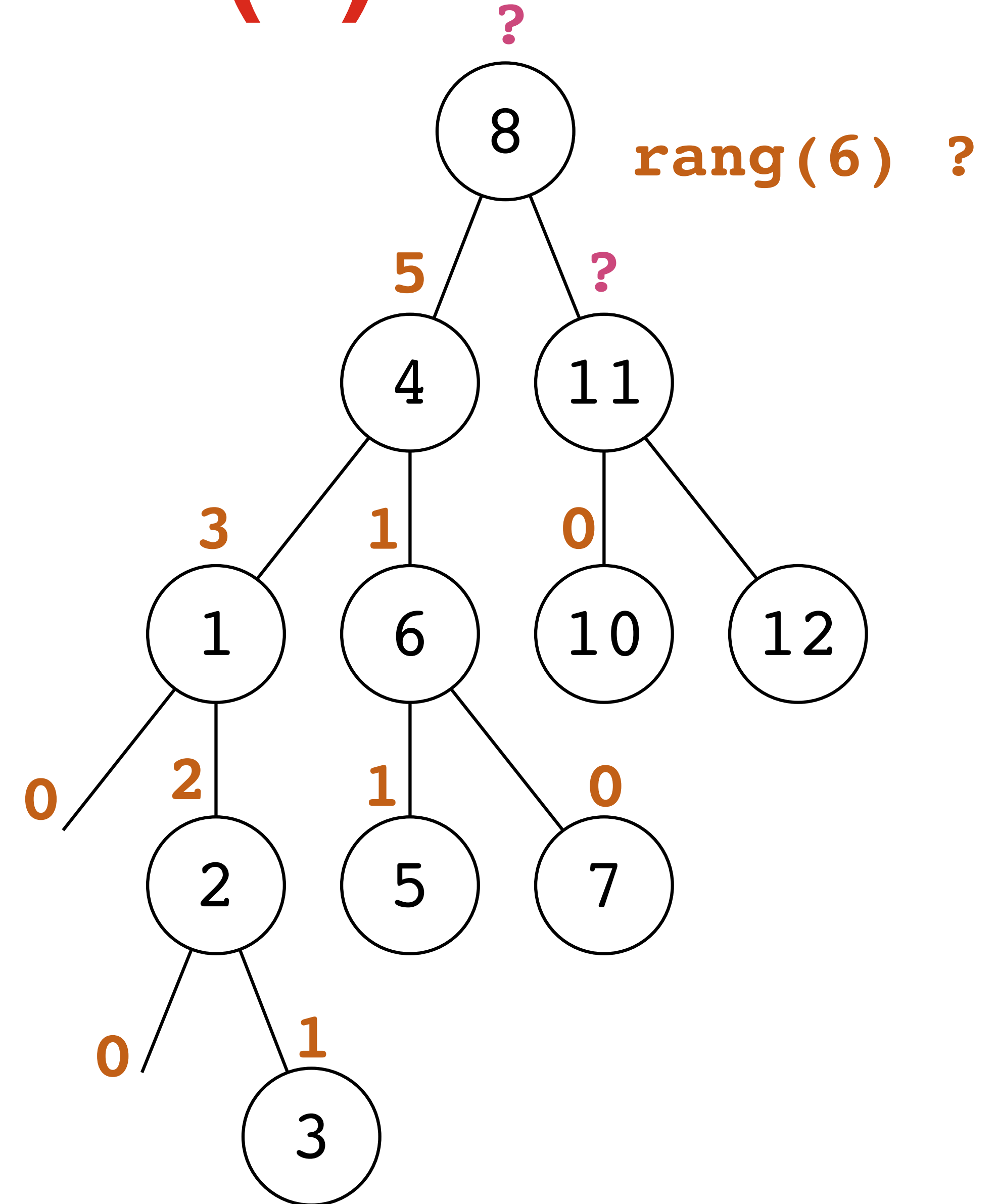


Rang d'une clé k (2)



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

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

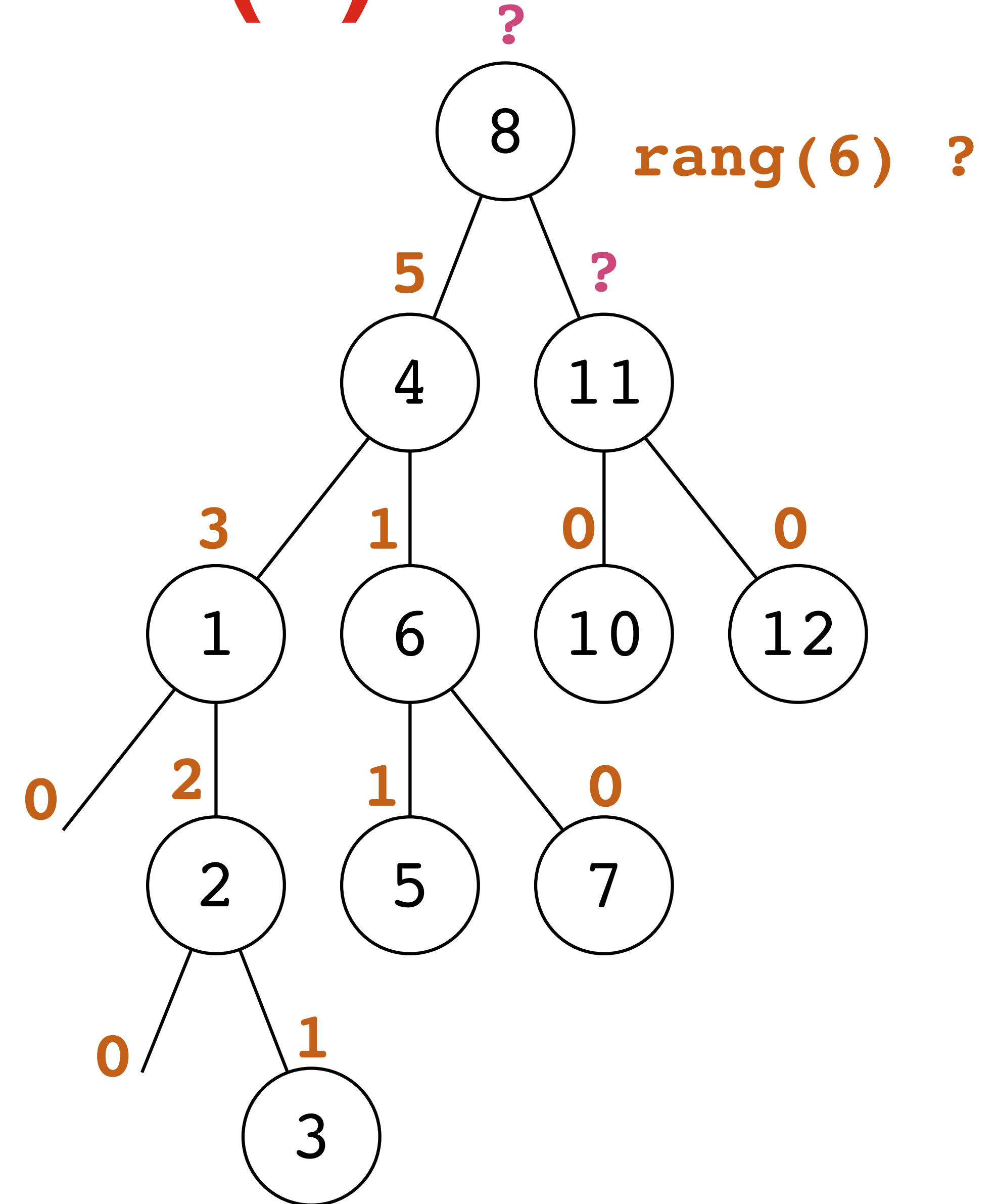


Rang d'une clé k (2)



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

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

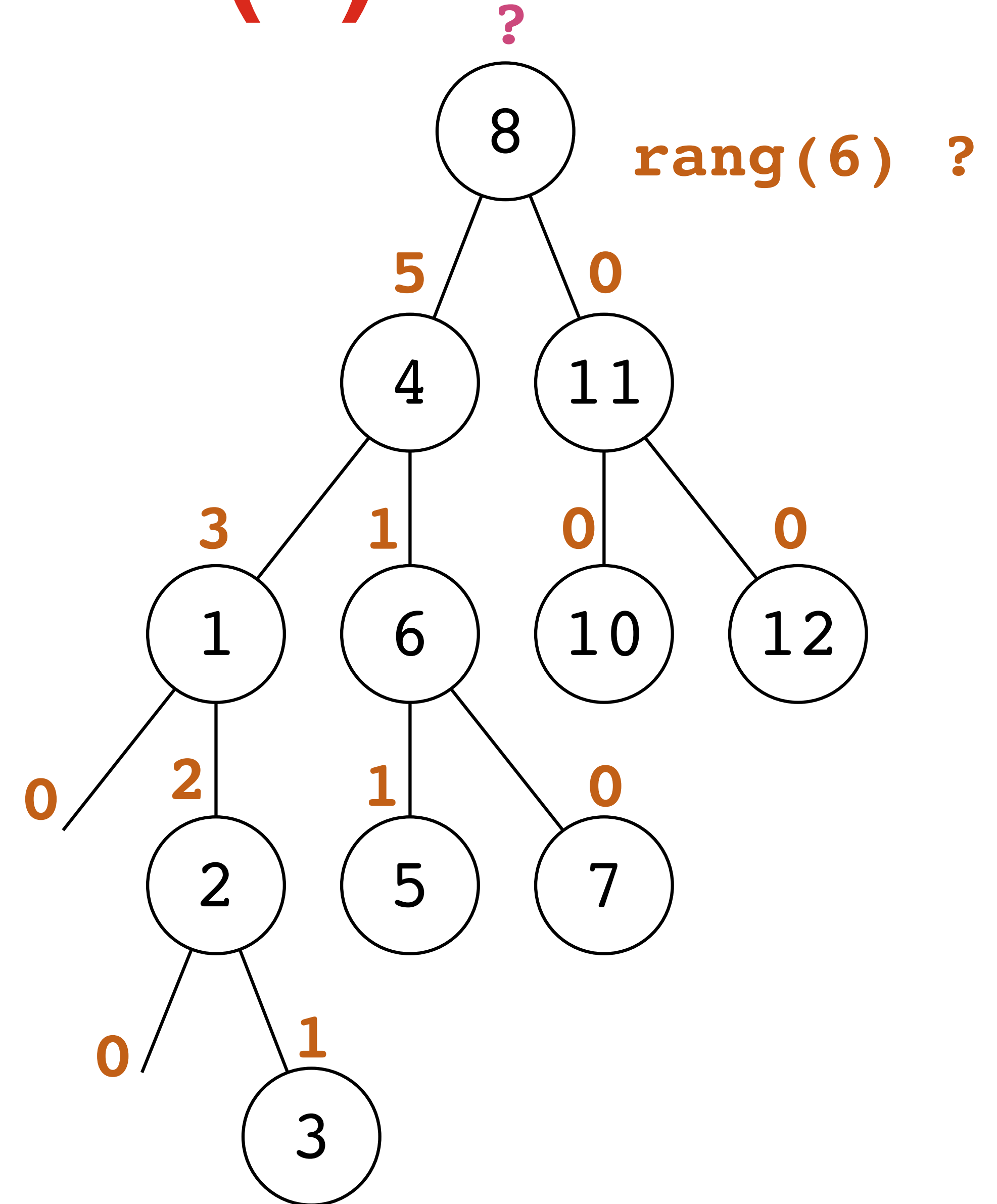


Rang d'une clé k (2)



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

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



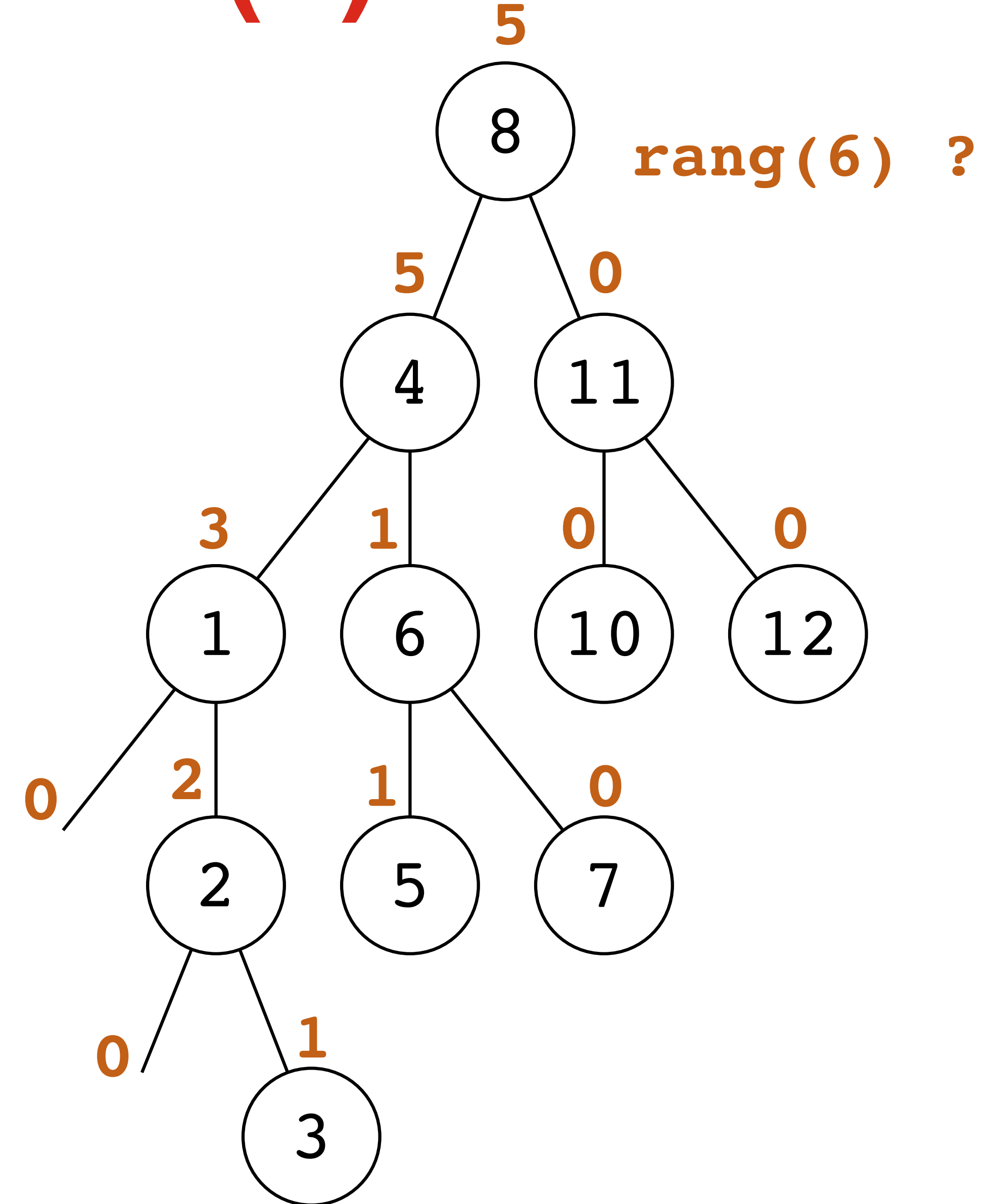
Rang d'une clé k (2)



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

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

Remarque ?



Rang d'une clé k (2)

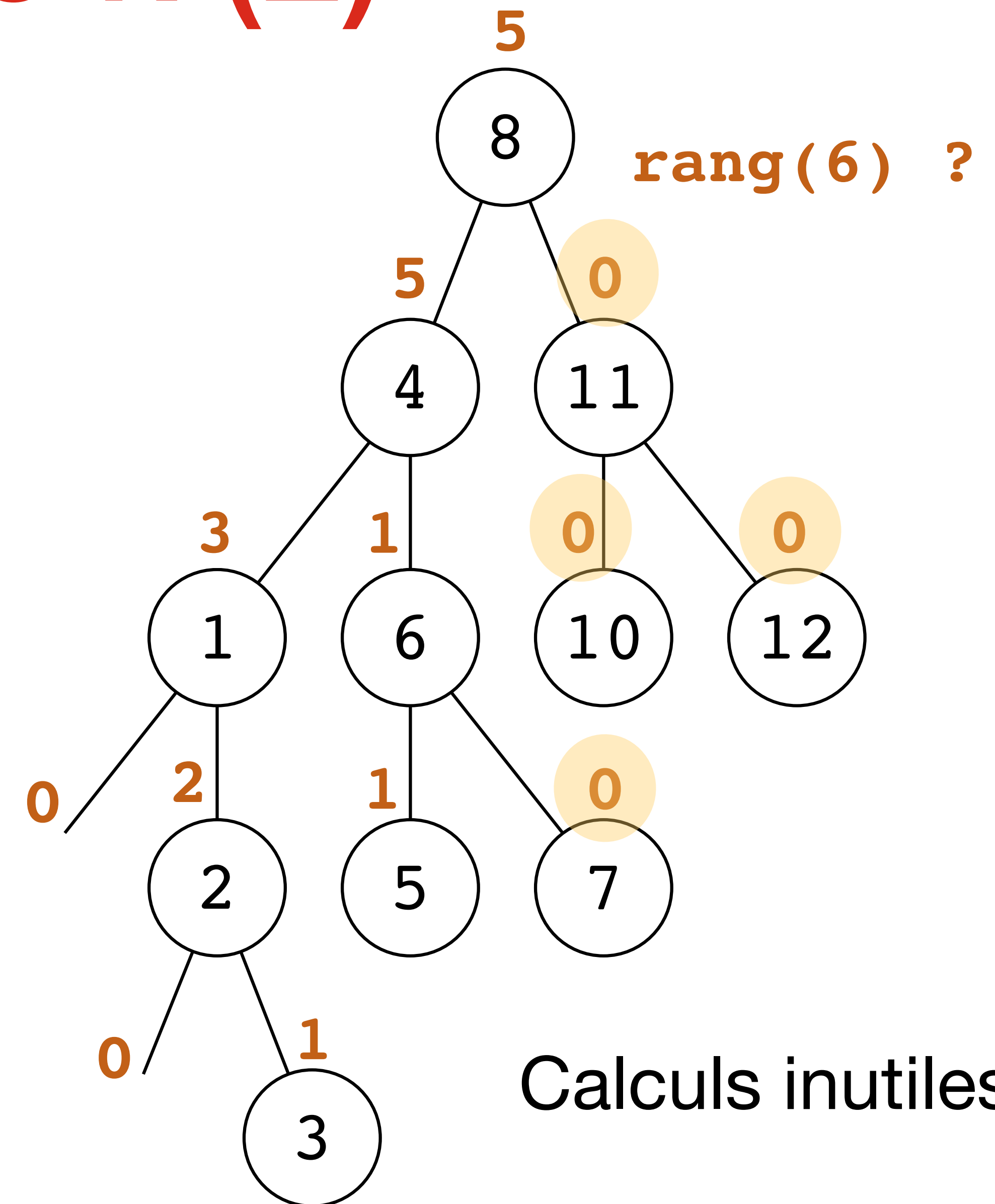


Parcourir tout l'arbre et compter combien de clés sont $< k$

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

Remarque ?

Si $r.clé \geq k$,
tous les noeuds de $r.droit$ seront $> k$,
donc $\text{rang}(r.droit, k)$ vaudra zéro.



Rang d'une clé k (3)

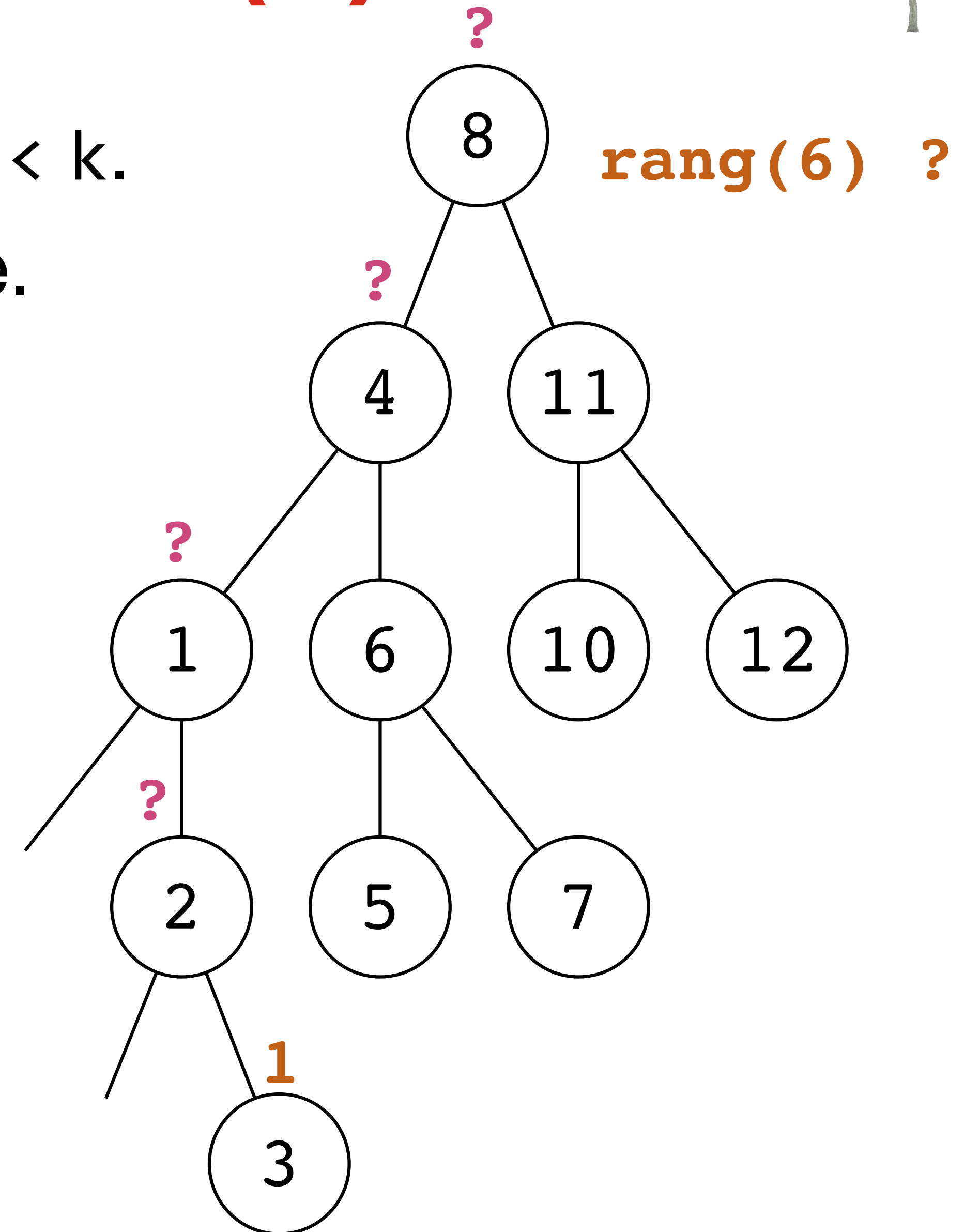


Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```

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



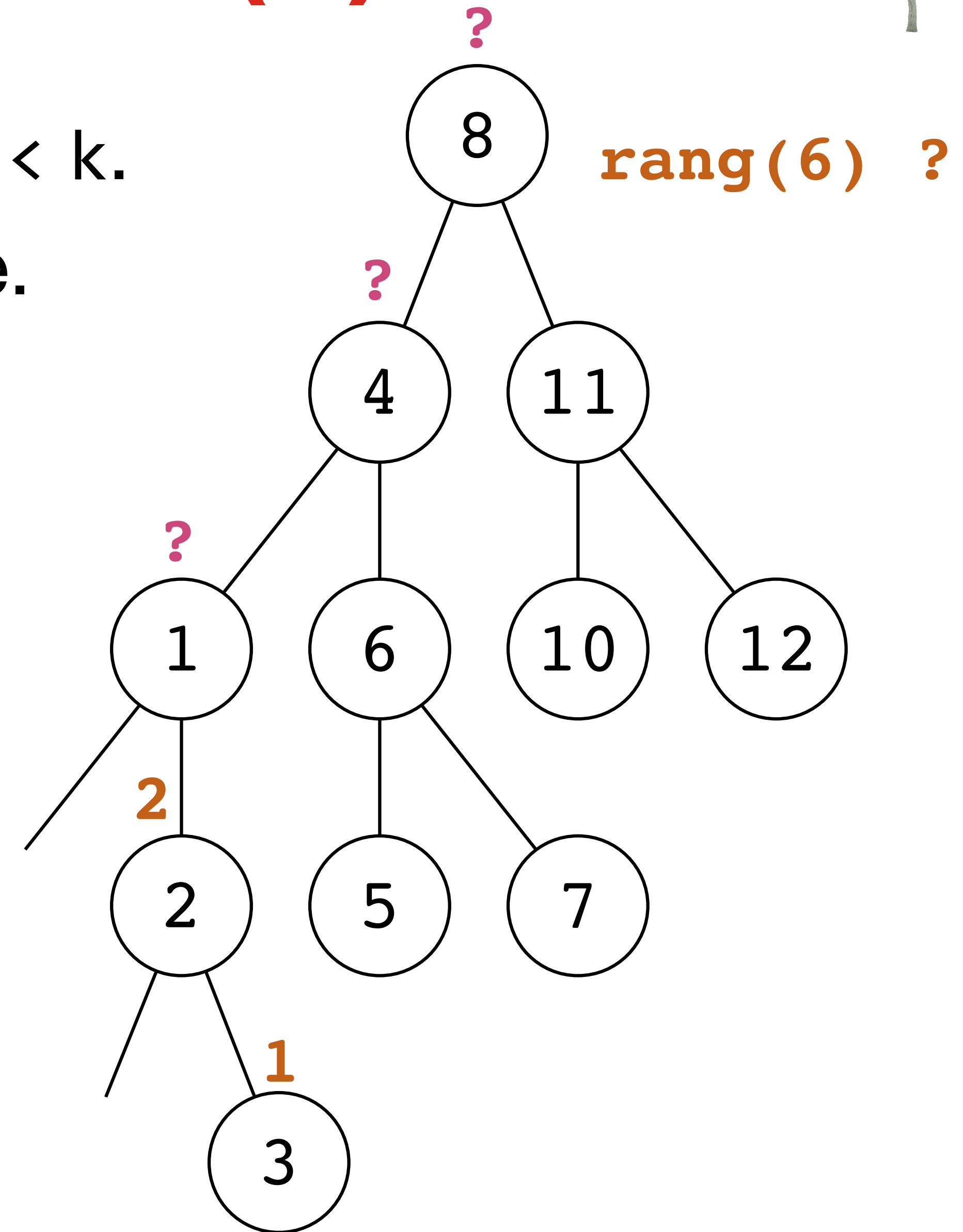
Rang d'une clé k (3)



Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```
fonction rang (r, k)
  si r == ∅,
    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)

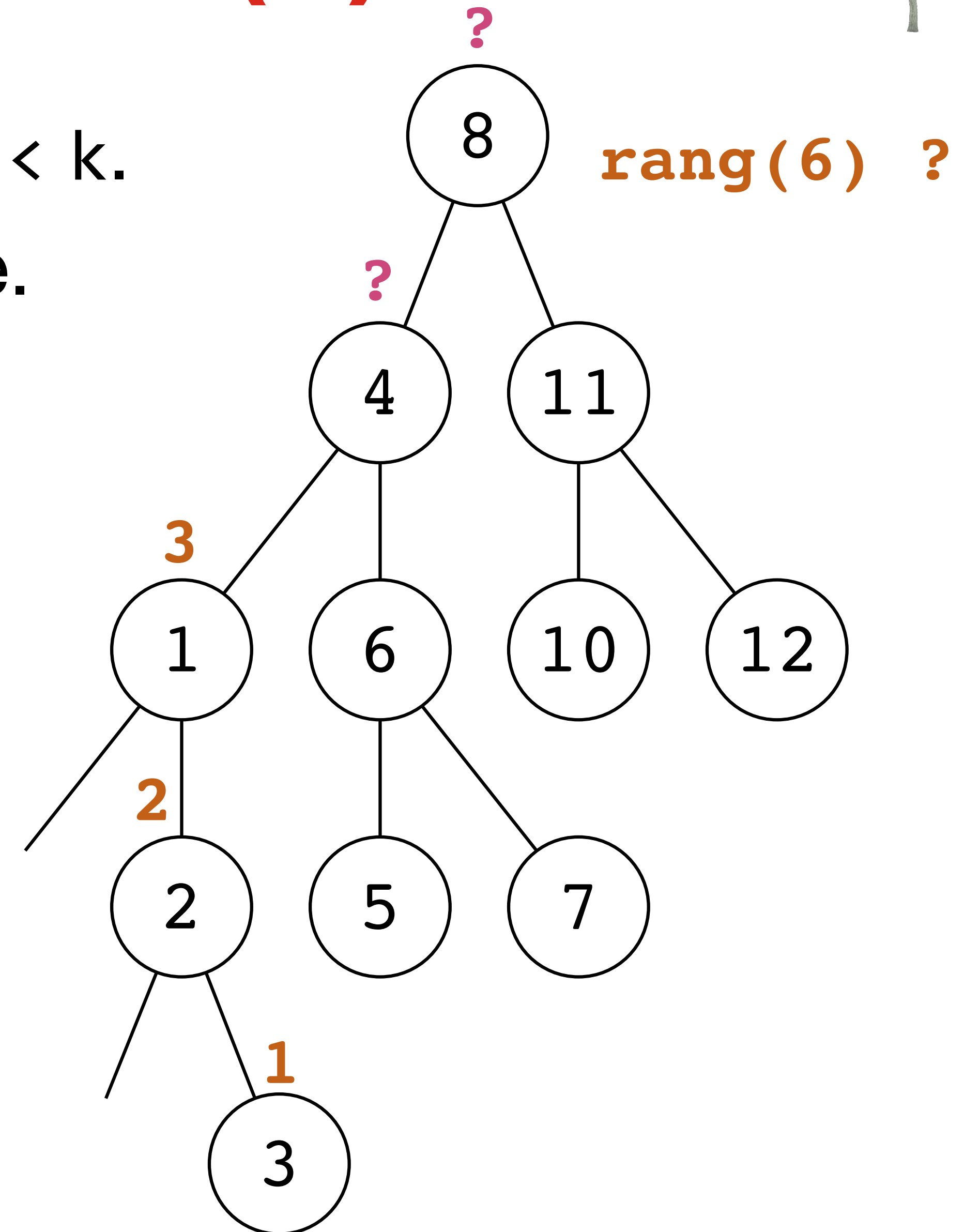


Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```

fonction rang (r, k)
  si r == ∅,
    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)

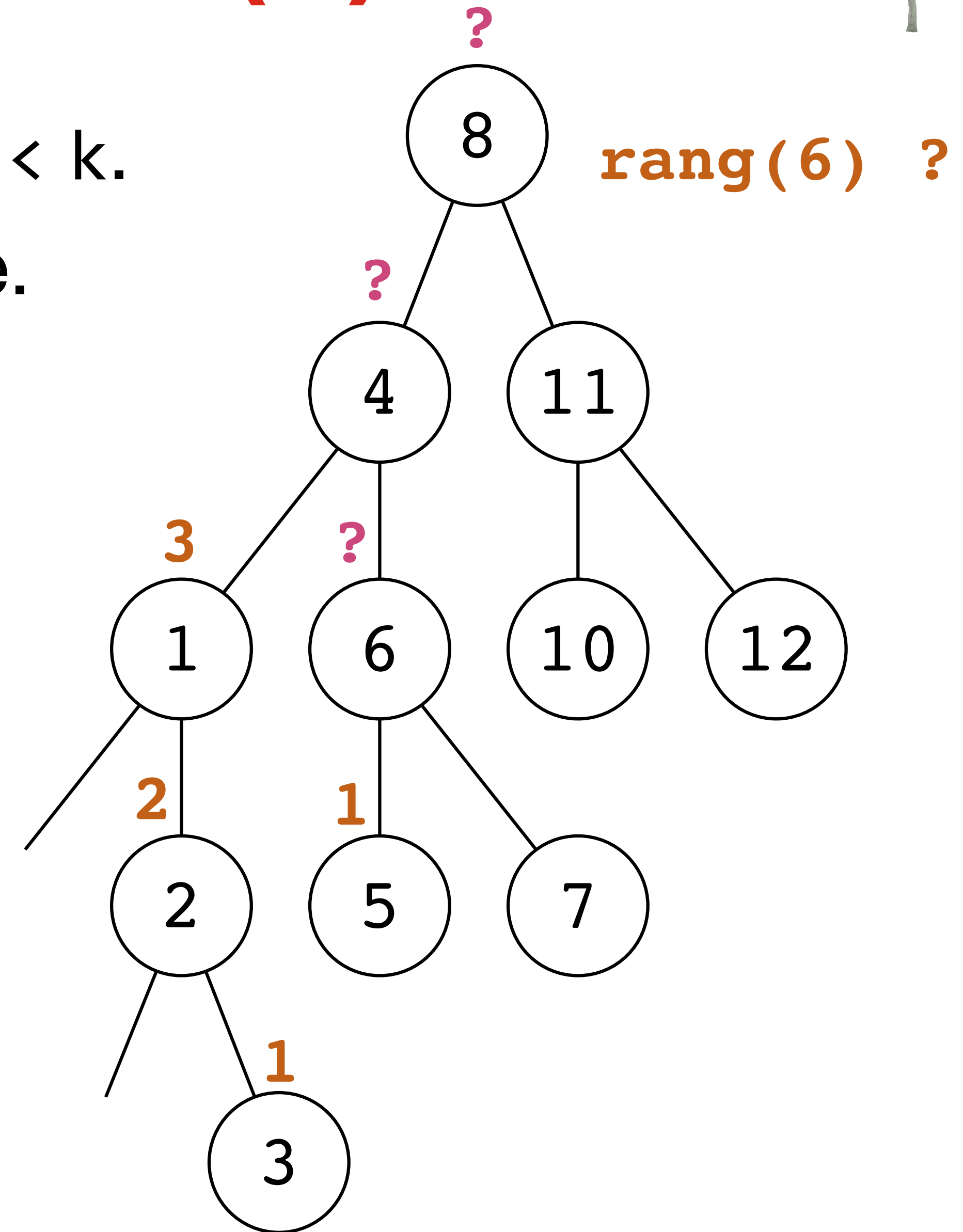


Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```

fonction rang (r, k)
    si r == ∅,
        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)

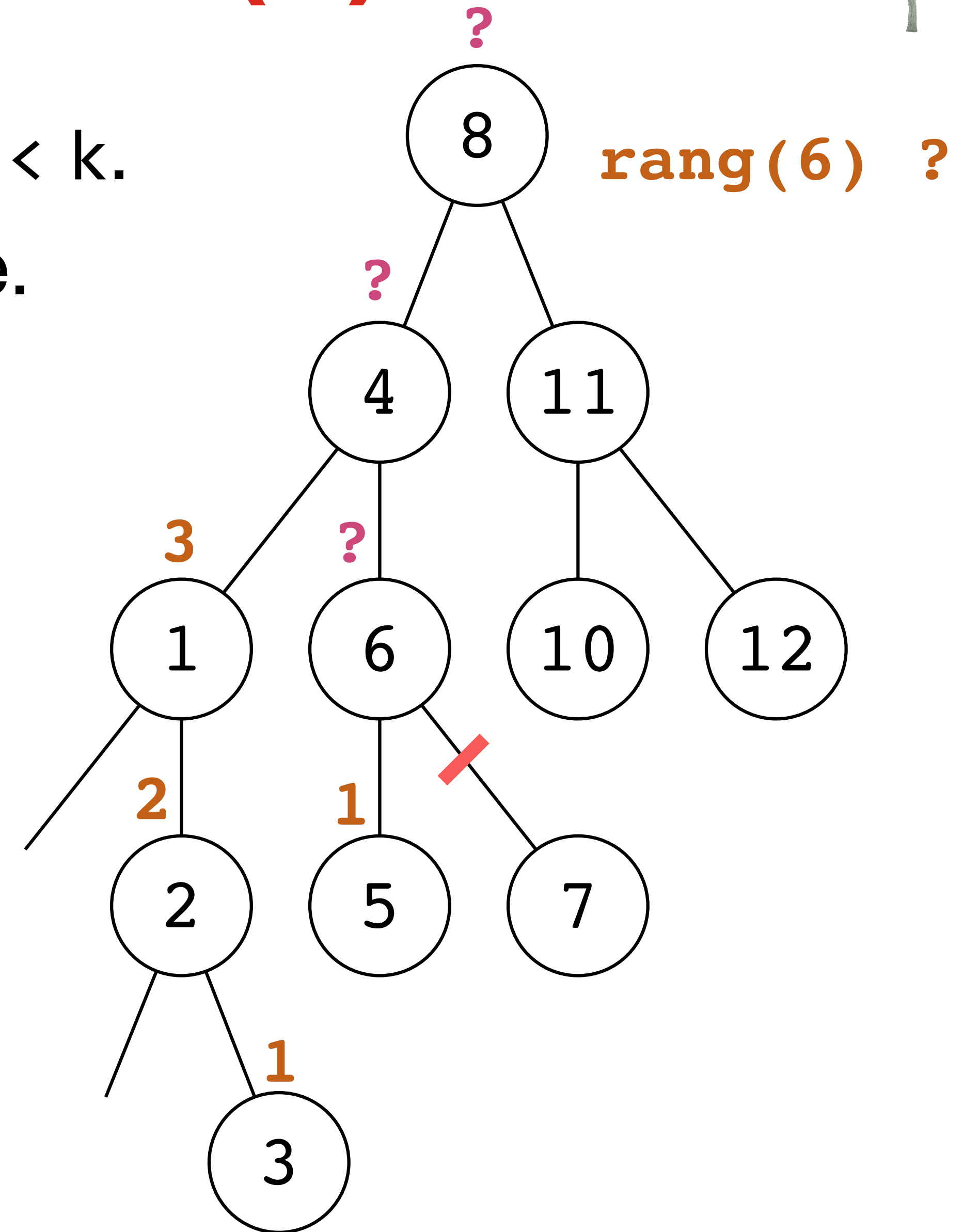


Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```

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



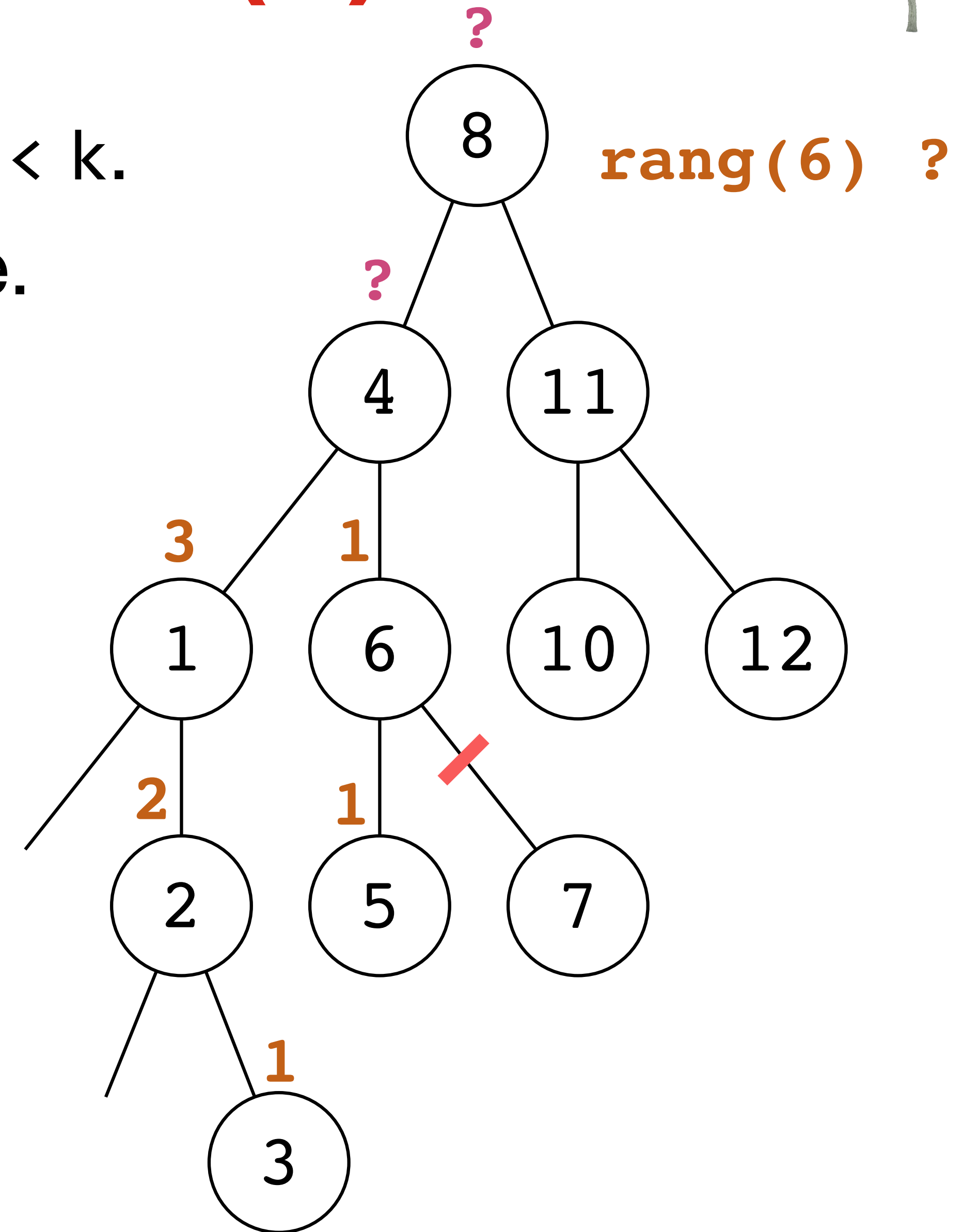
Rang d'une clé k (3)



Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

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



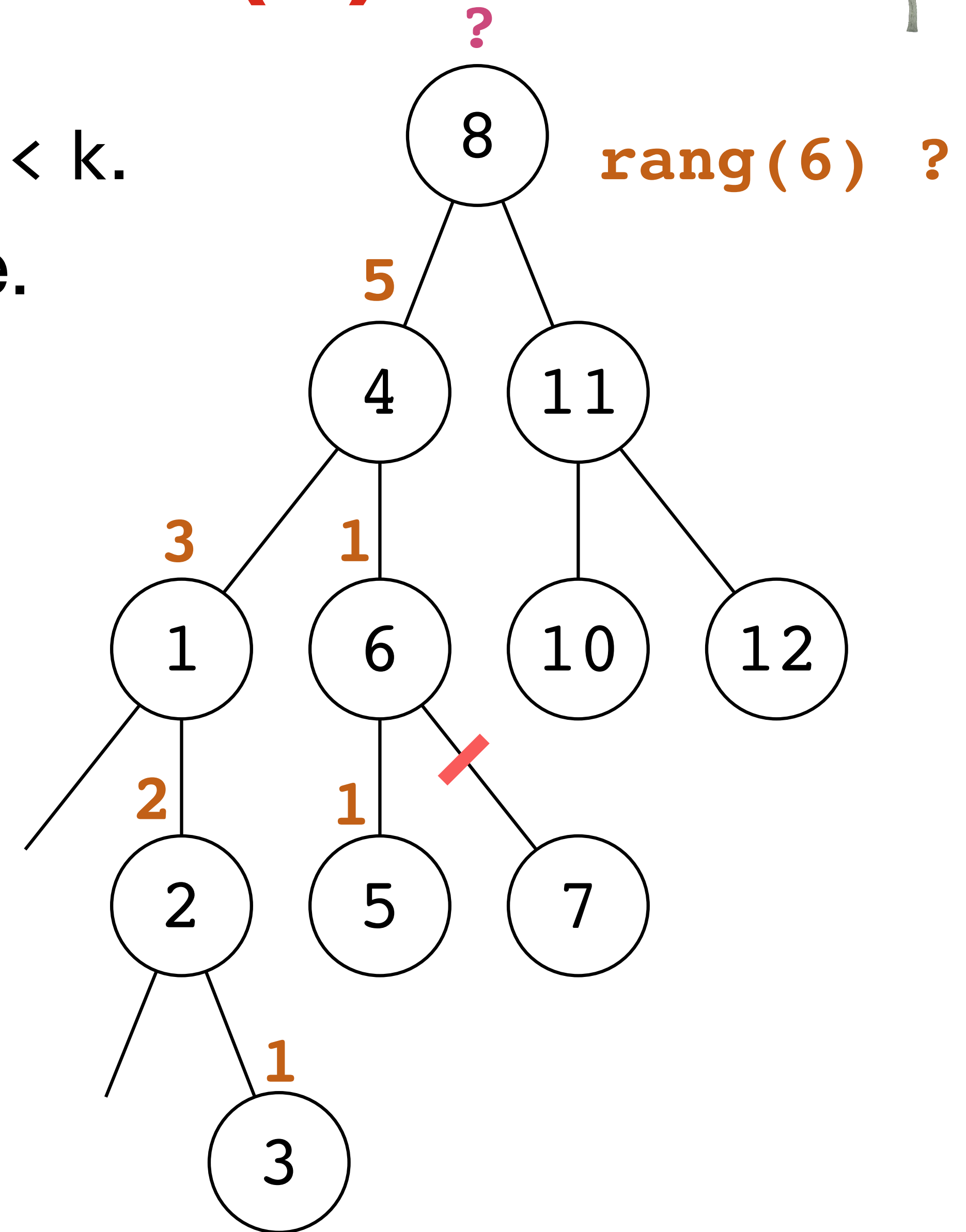
Rang d'une clé k (3)



Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

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



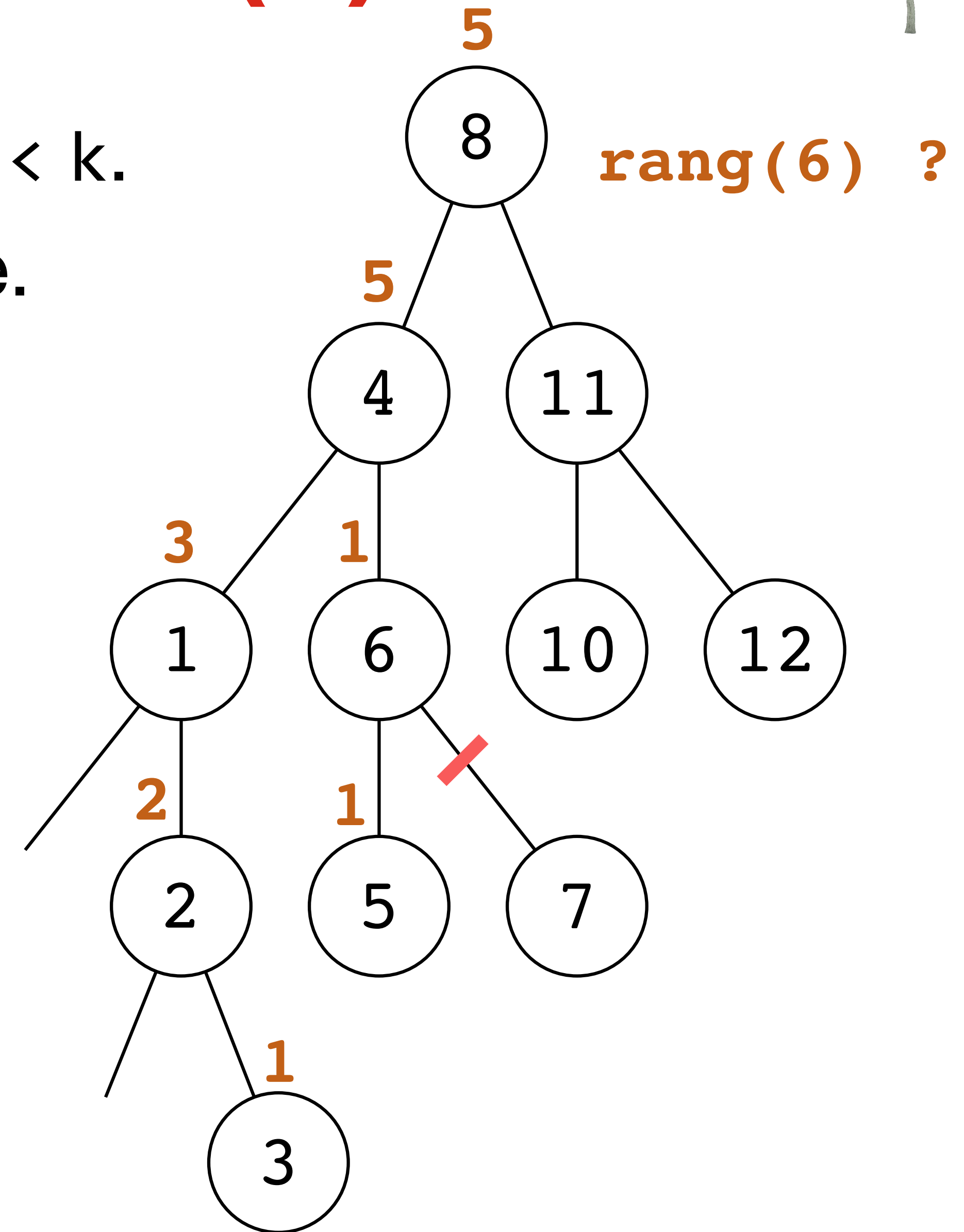
Rang d'une clé k (3)



Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```
fonction rang (r, k)
  si r == ∅,
    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)



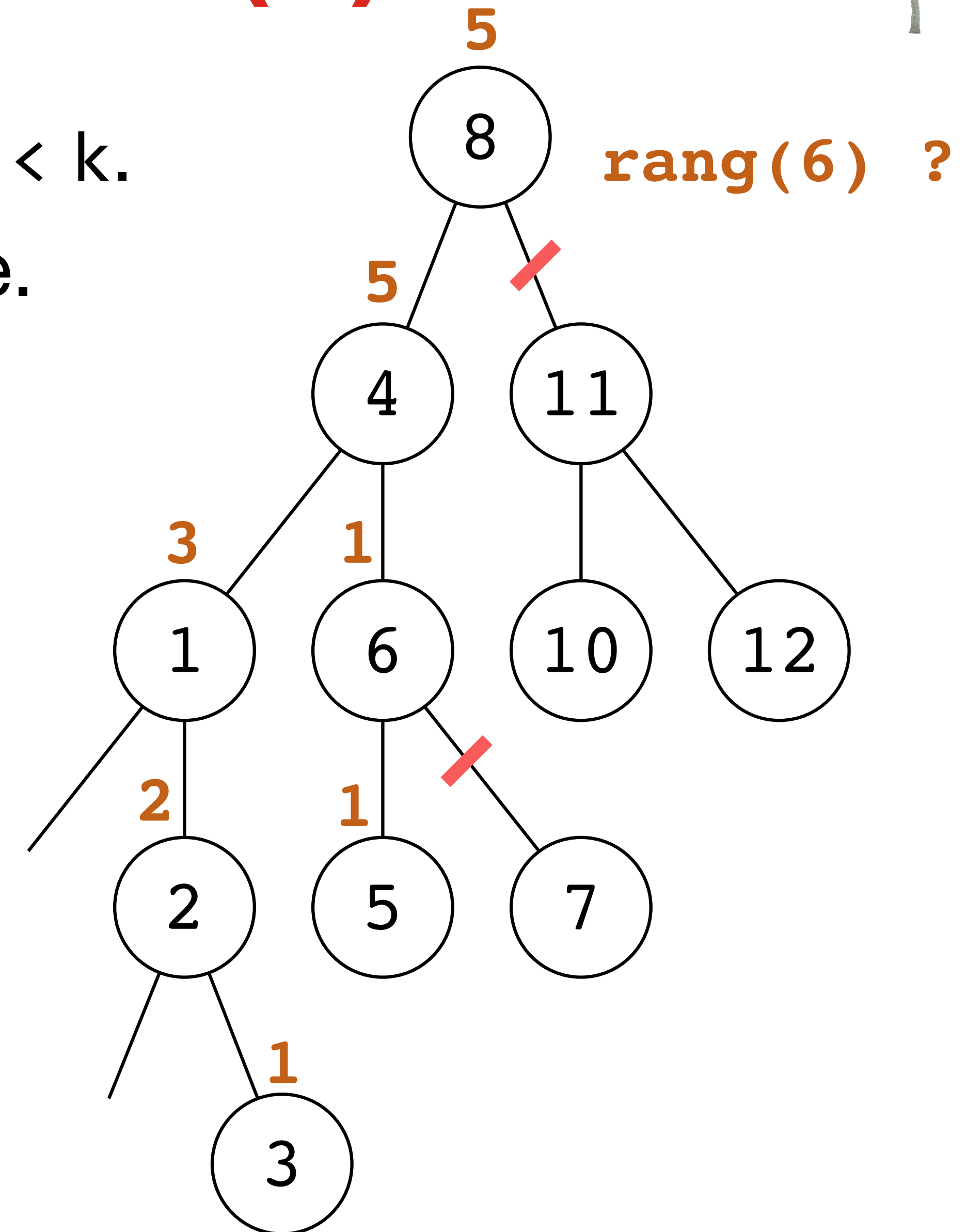
Si $r.\text{clé} \geq k$, aucune clé de $r.\text{droit}$ n'est $< k$.

L'appel $\text{rang}(r.\text{droit}, k)$ est donc inutile.

```

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

Autre remarque ?



Rang d'une clé k (3)

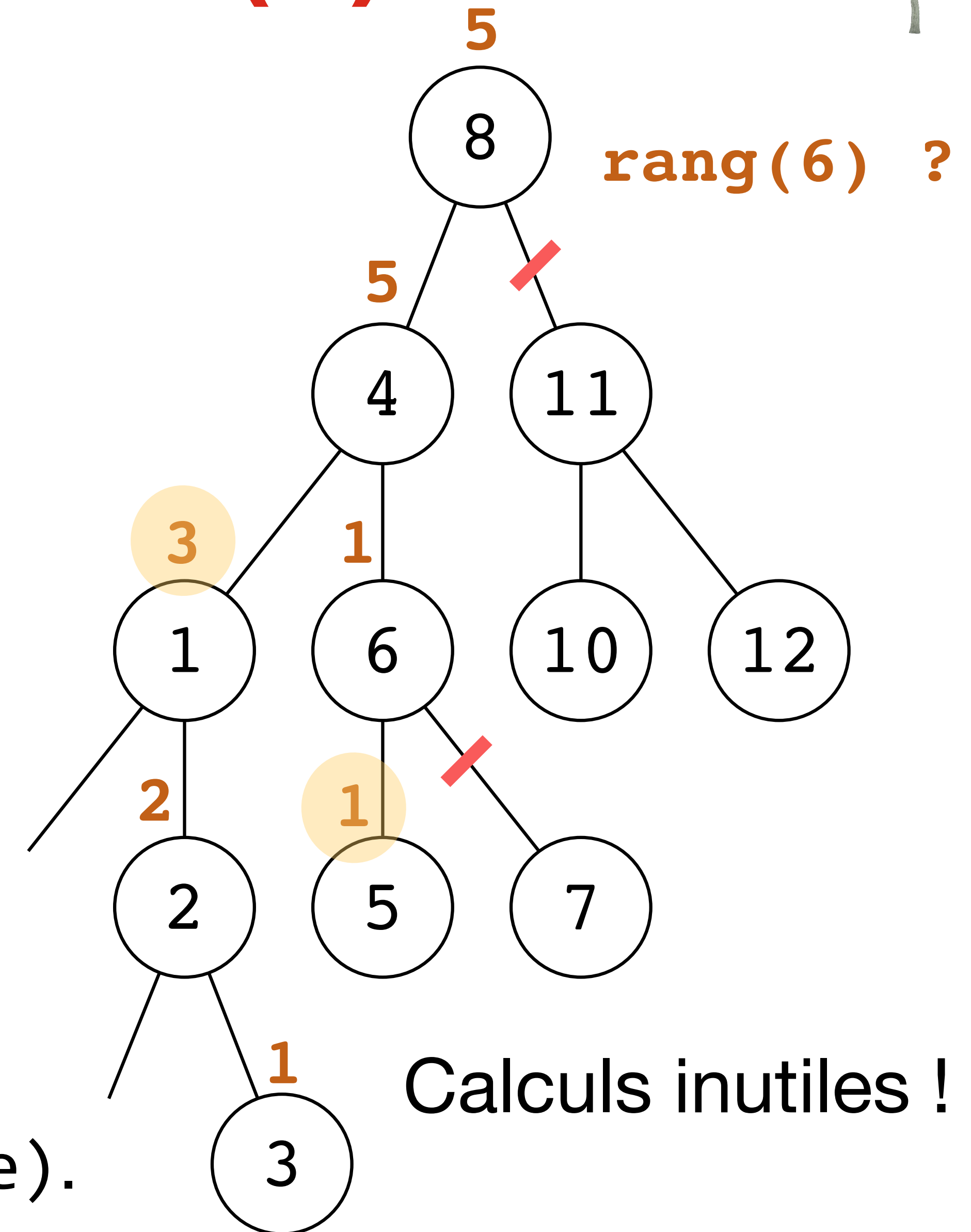


```

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

Autre remarque ?

Si $r.clé \leq k$,
tous les noeuds de $r.gauche$ seront $< k$, donc
 $rang(r.gauche, k)$ vaudra $taille(r.gauche)$.



Rang d'une clé k (4)

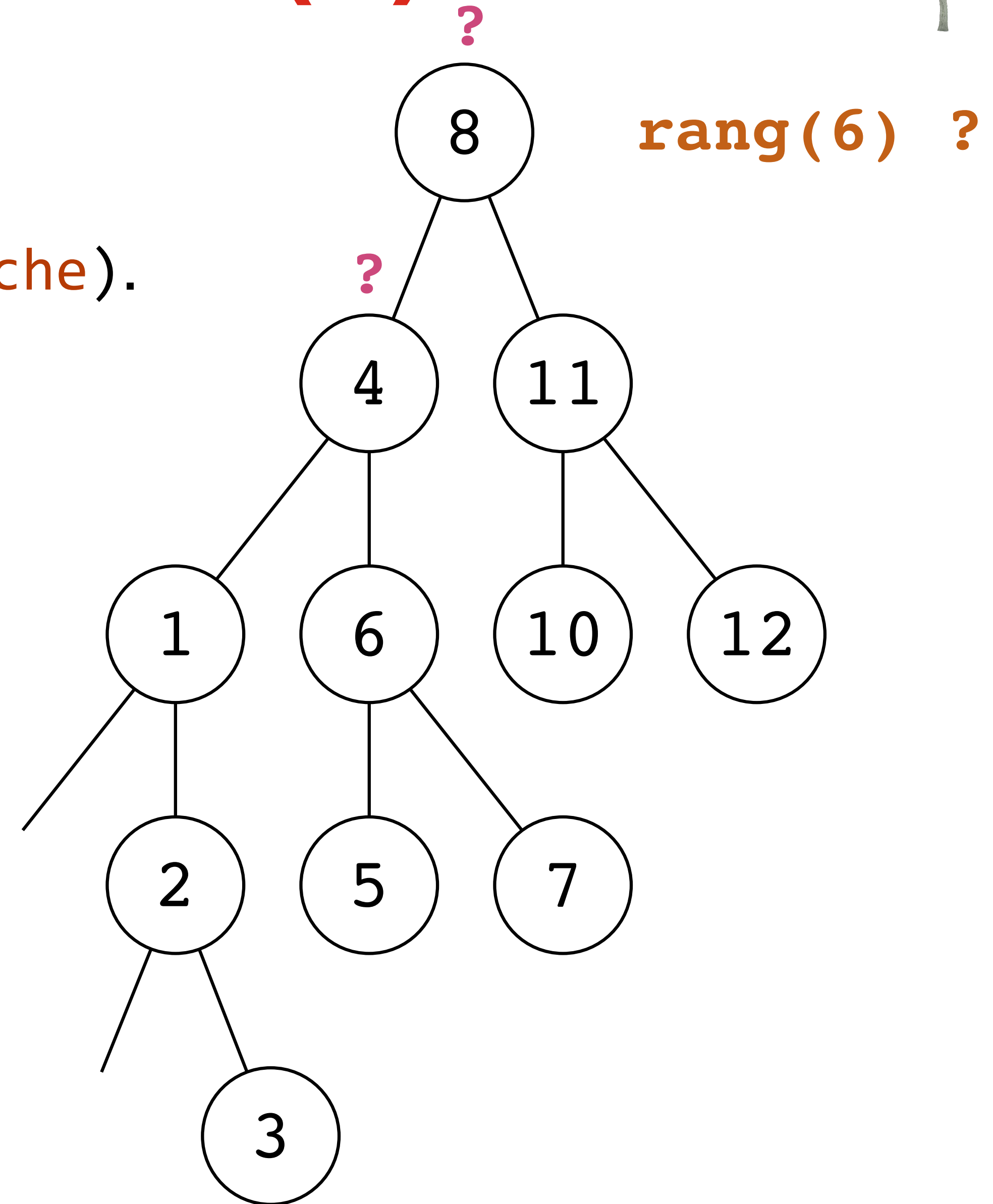


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

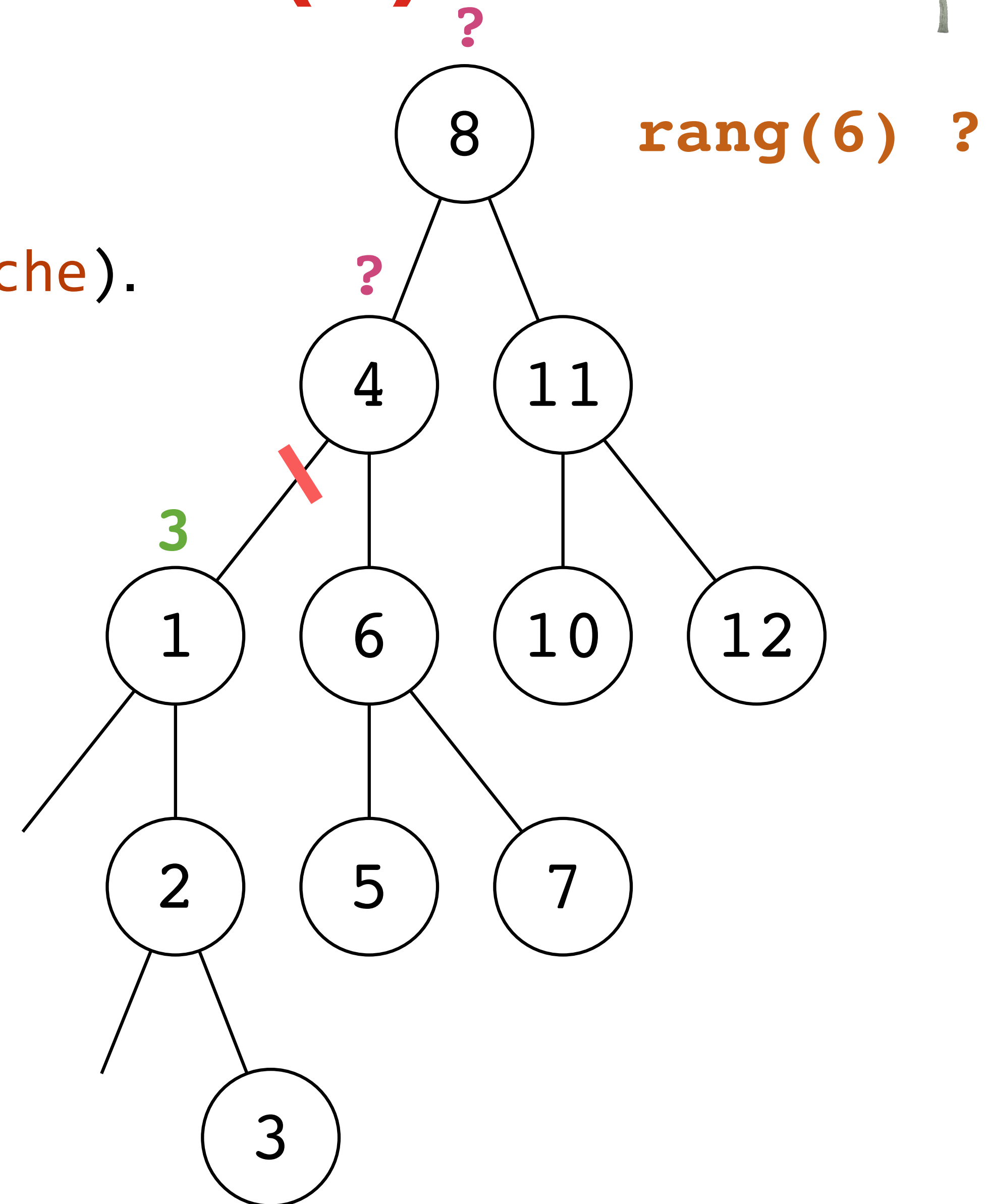


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

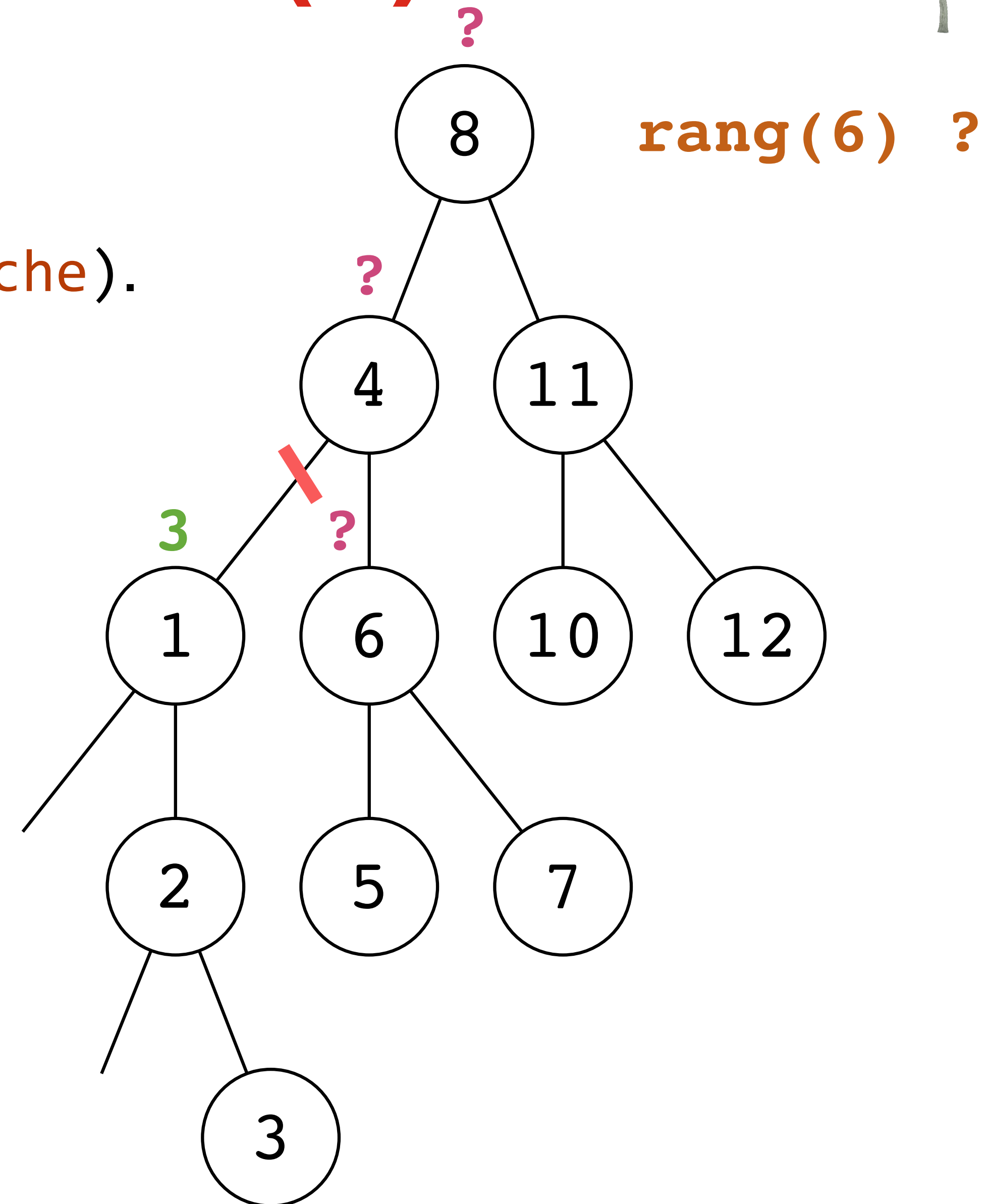


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

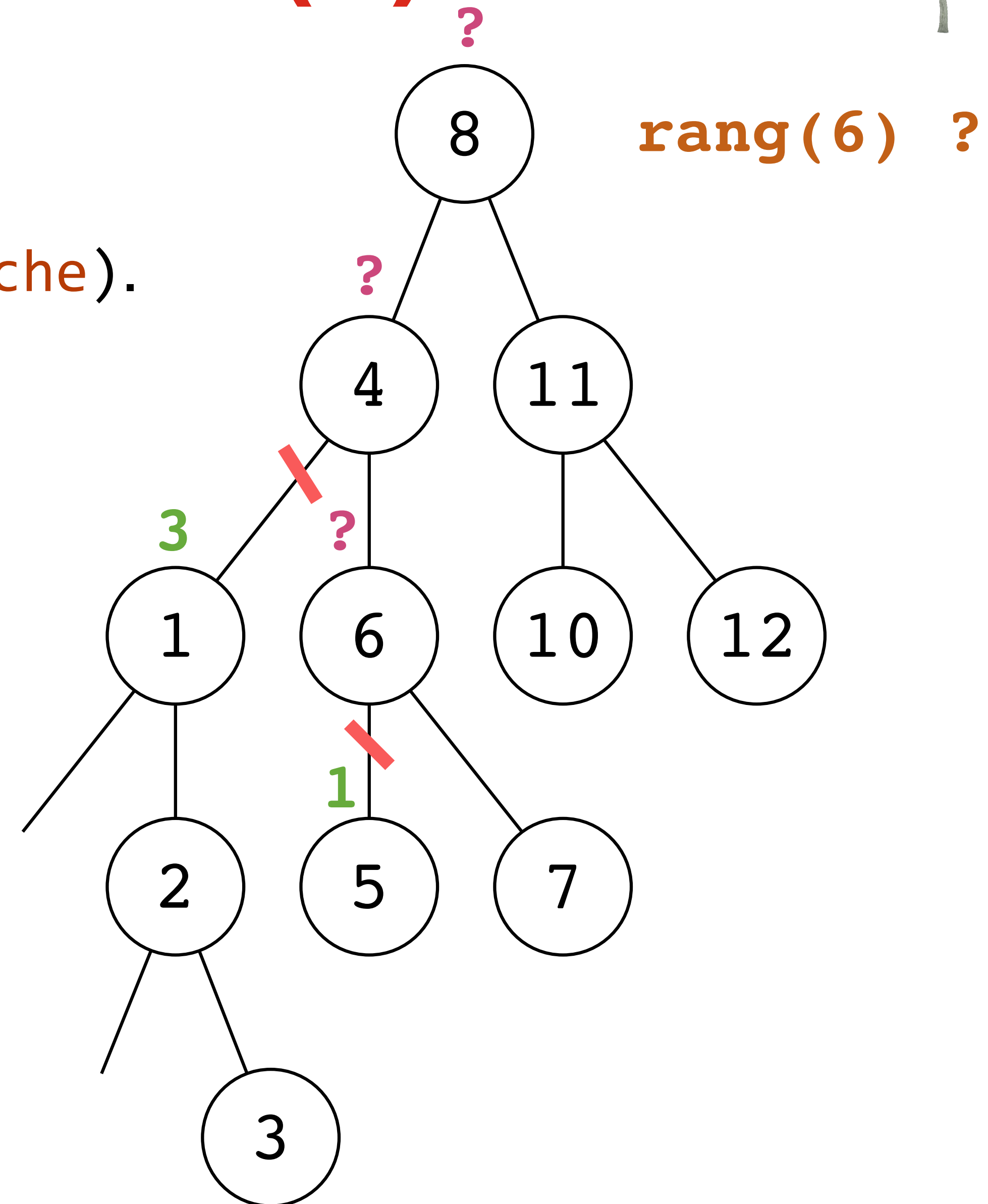


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

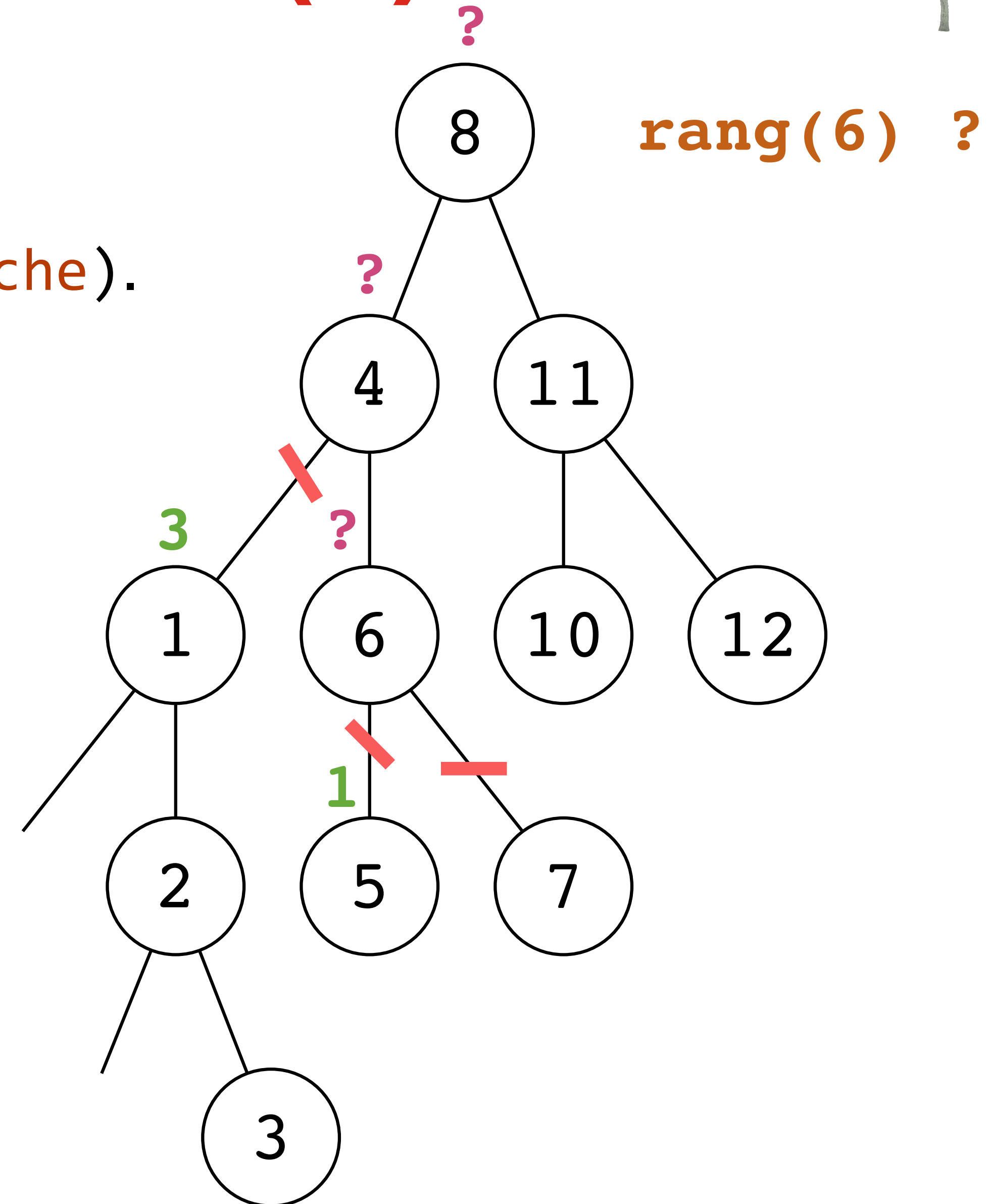


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

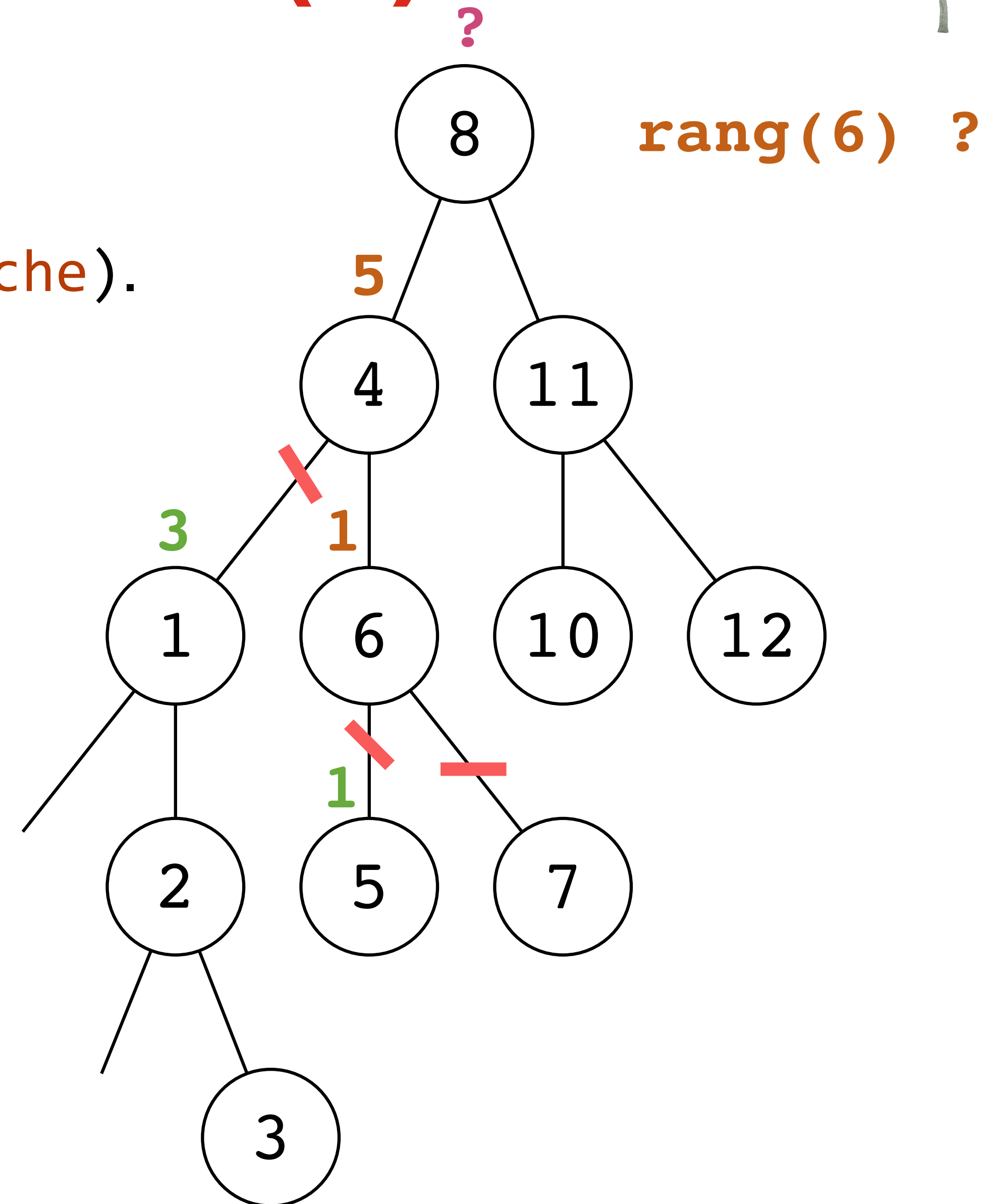


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

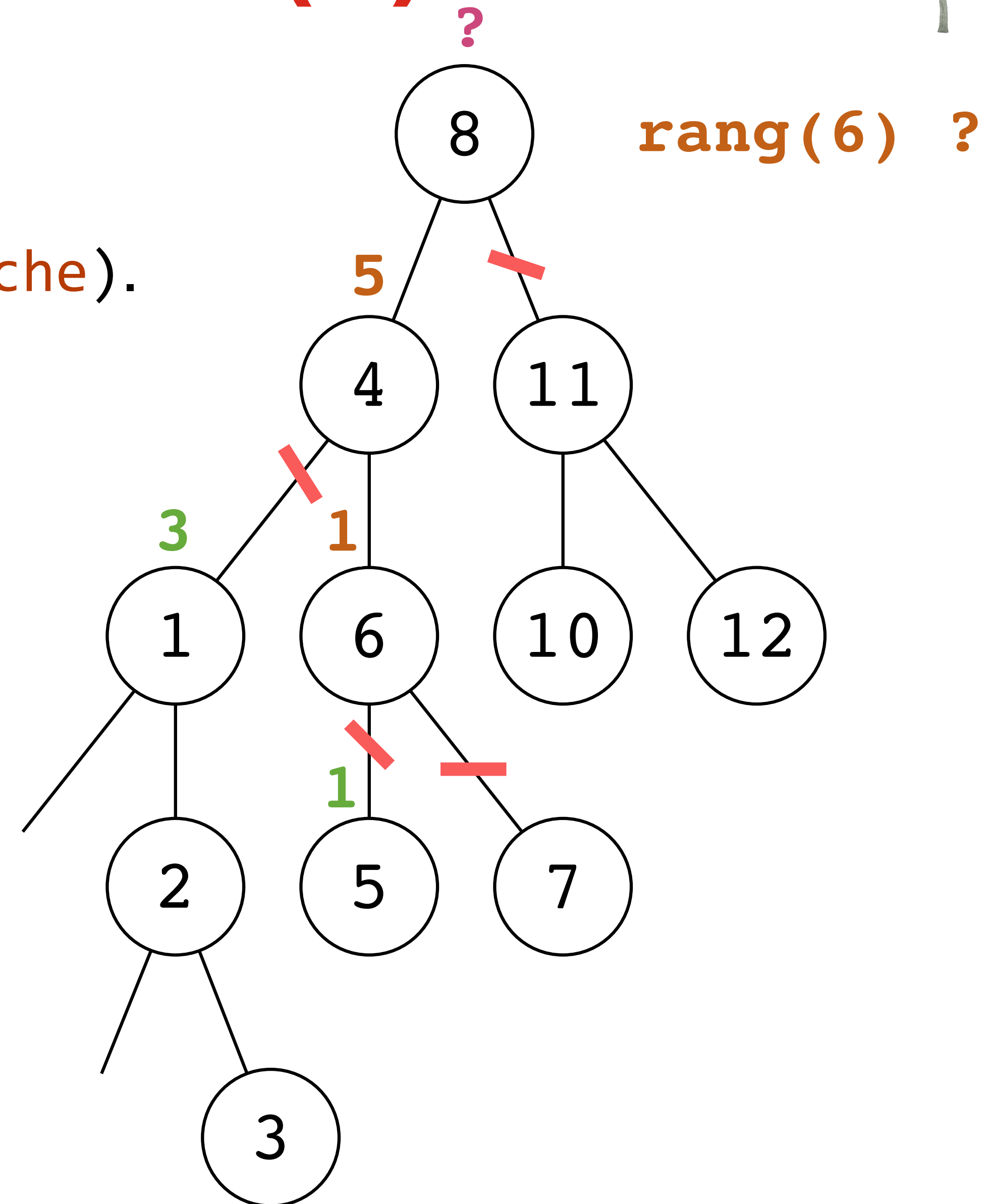


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

fonction rang (r, k)
  si r == ∅,
    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)

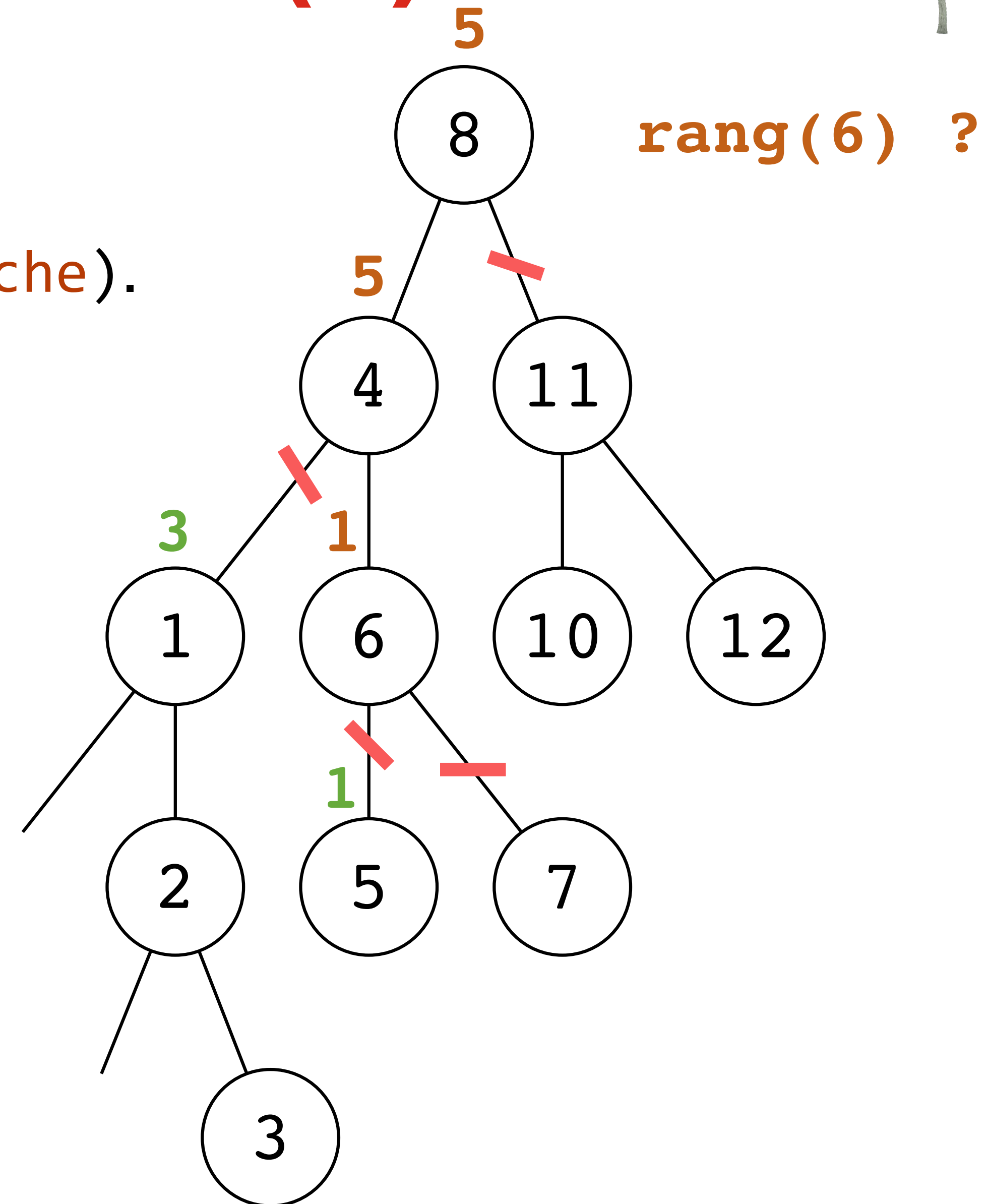


Si $r.\text{clé} \leq k$, toutes les clés de $r.\text{gauche}$ sont $< k$.

Donc $\text{rang}(r.\text{gauche}, k)$ est égal à $\text{taille}(r.\text{gauche})$.

```

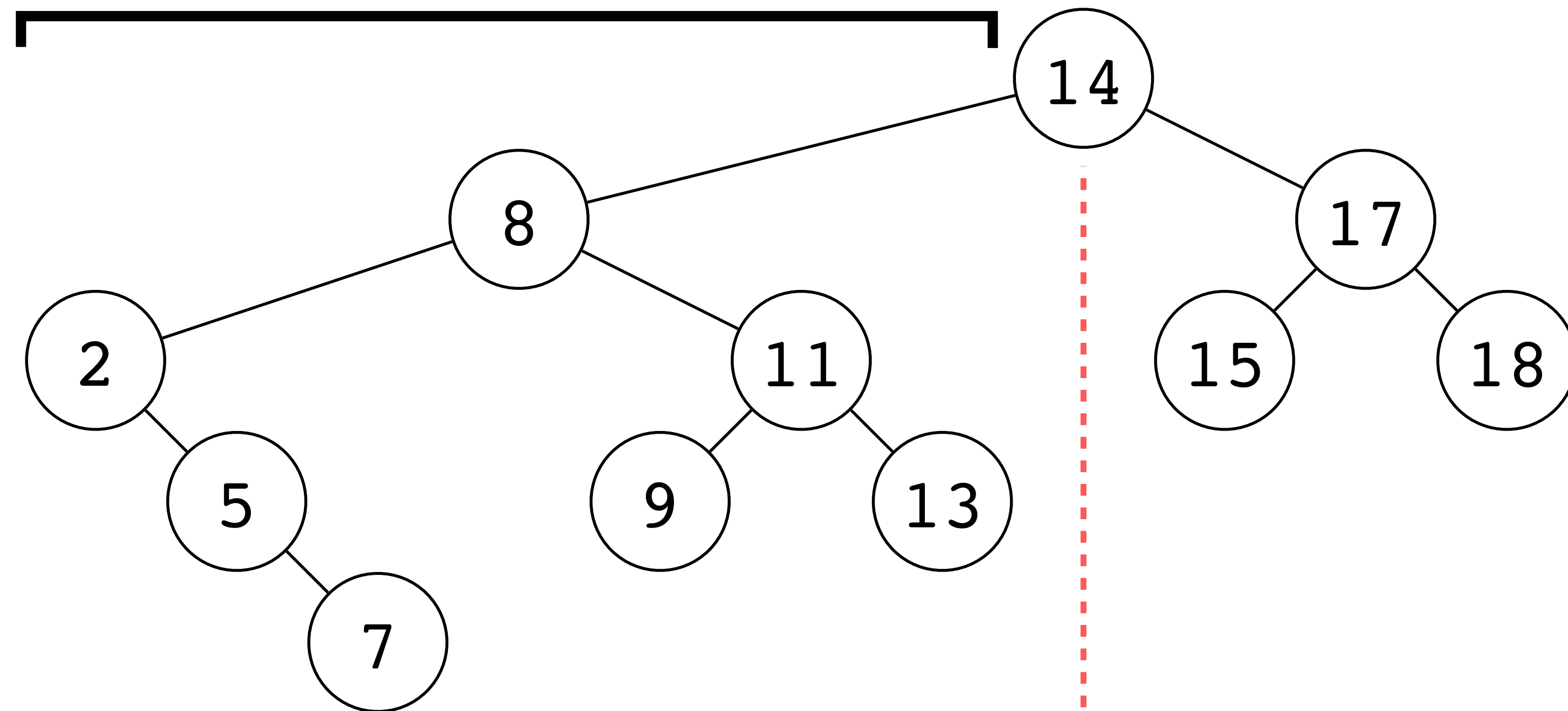
fonction rang (r, k)
  si r == ∅,
    retourner 0
  sinon,
    si r.clé > k,
      rg ← rang(r.gauche, k)
    sinon,
      rg ← taille(r.gauche)
      si r.clé < k,
        rg ← rg + rang(r.droit, k) + 1
    retourner rg
  
```



En résumé



Si la clé x se trouve ici
(car elle est $< r.clé$)



$Rang(14, x) =$

$Rang(8, x)$

+

0

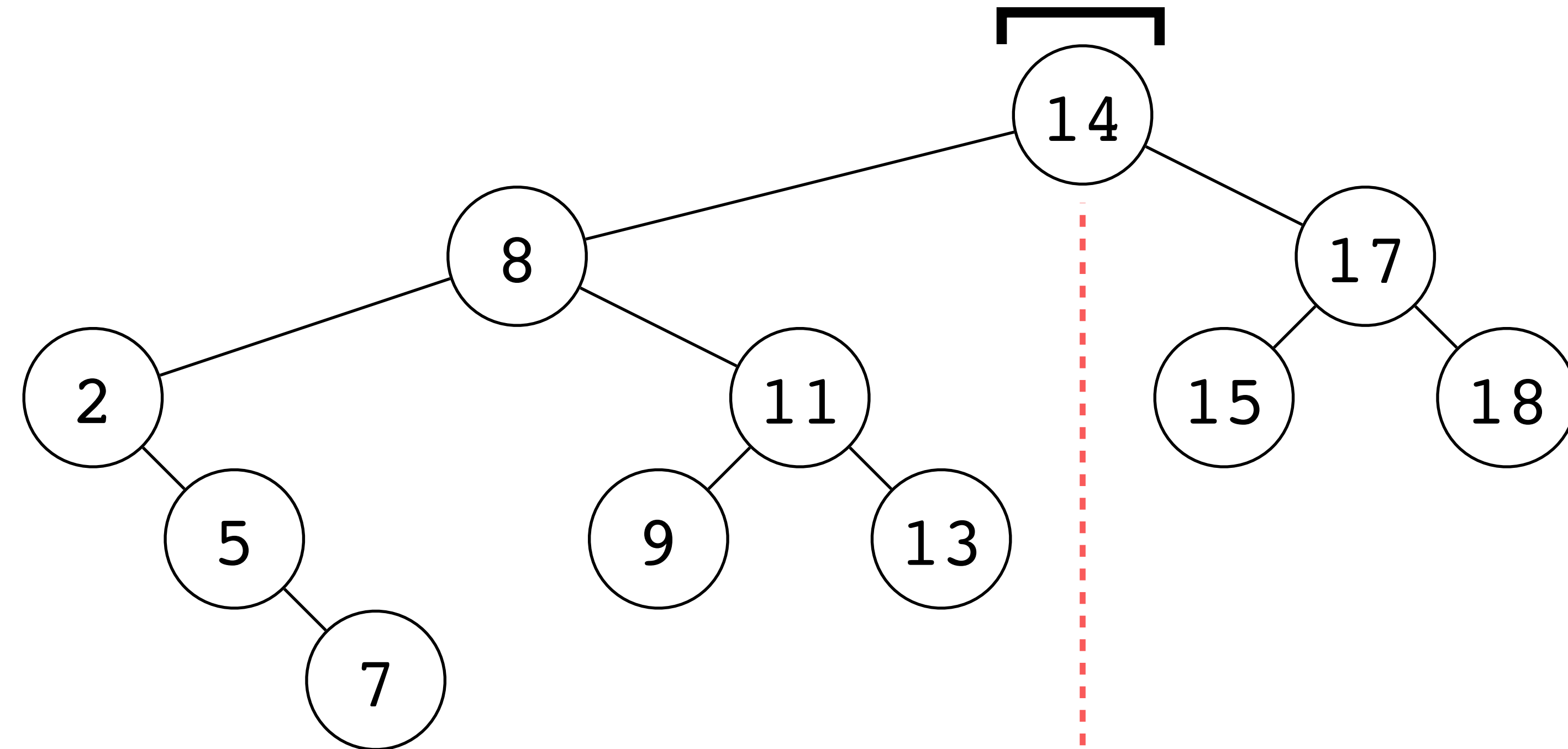
+

0

En résumé



Si la clé x est ici
(car elle est = $r.clé$)



$Rang(14,x) =$

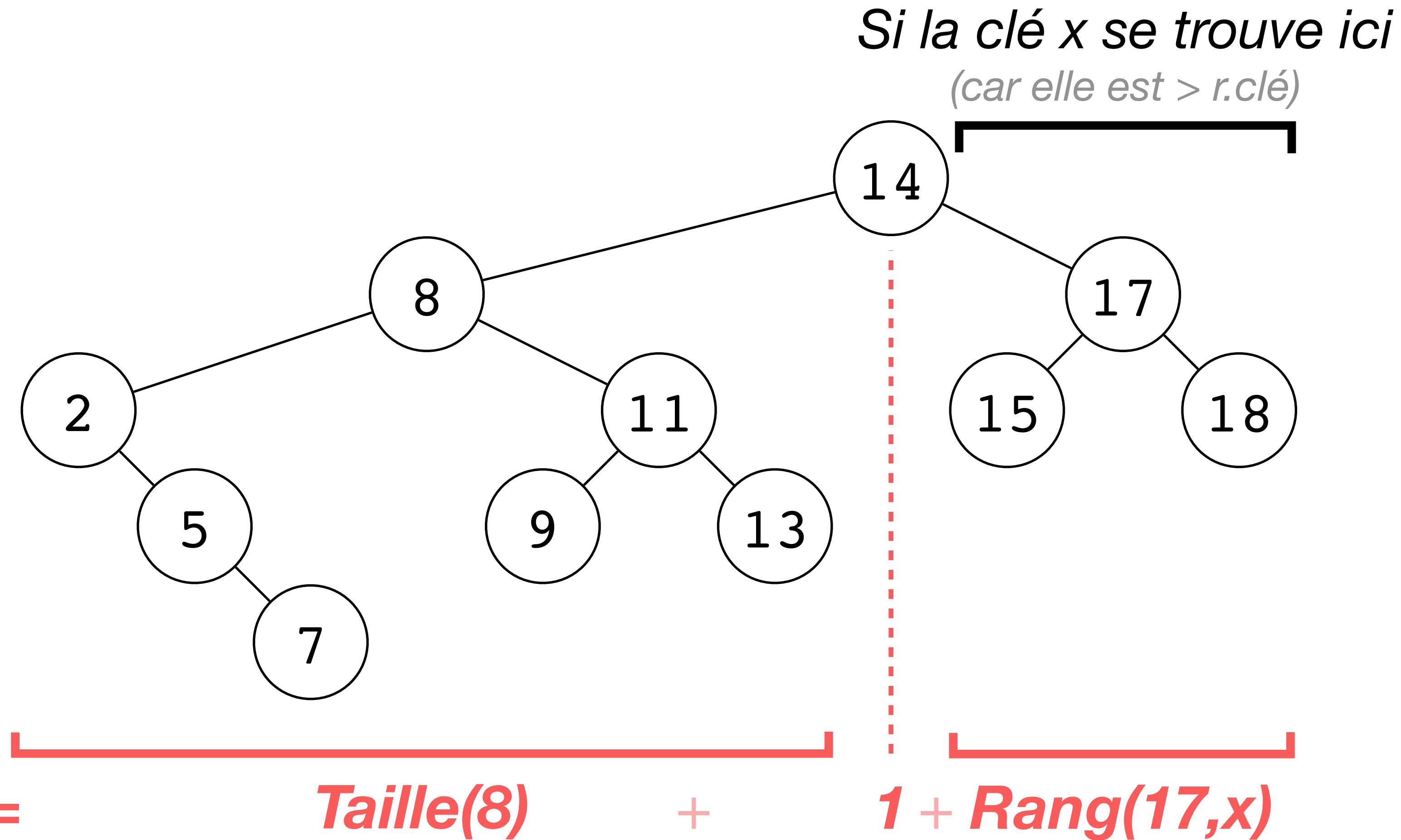
$Taille(8)$

+

1

+

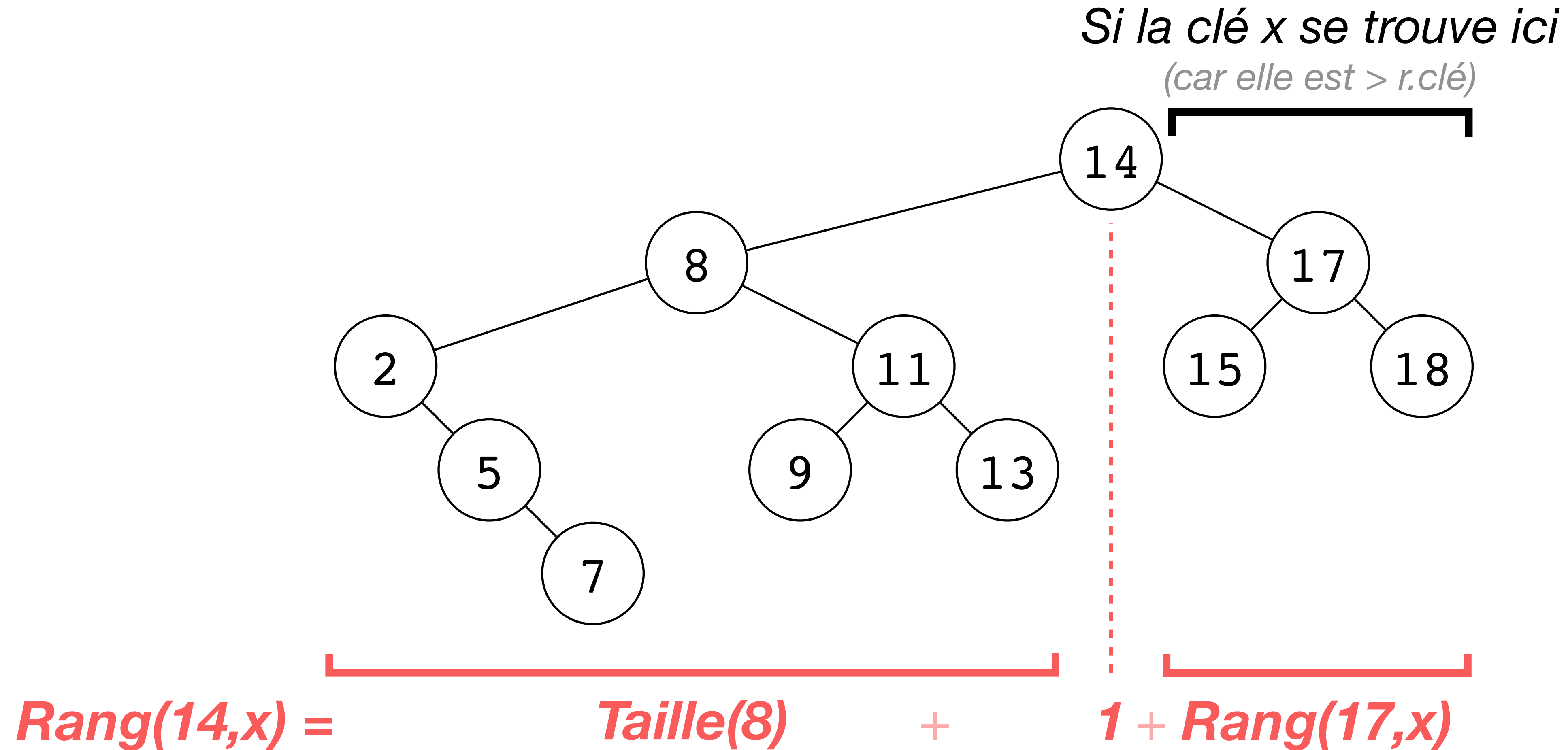
0



Remarque : un seul appel récursif par appel. Donc, complexité de



En résumé



Remarque : un seul appel récursif par appel. Donc, complexité de **$O(hauteur)$** .

Élément de rang n (1) - sélection



Définition

Étant donnée une racine et un n , trouver l'élément de rang n .

Là où `rang` donnait le rang d'une clé donnée,
`élément_n` donne l'élément ayant le rang donné.

C'est la fonction inverse de `rang`

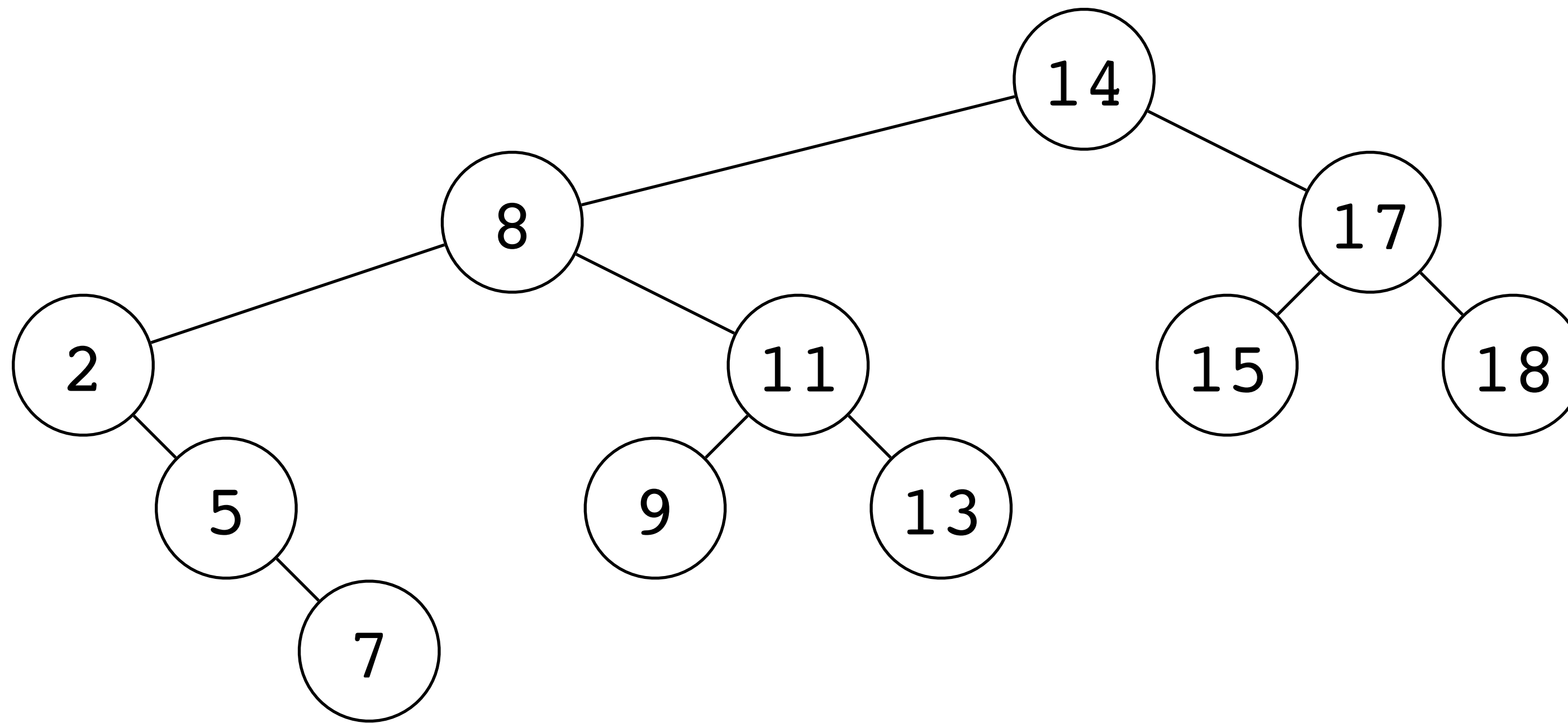
$$\text{rang}(r, \text{élément_n}(r, n).clé) == n$$

Comment le calculer ?



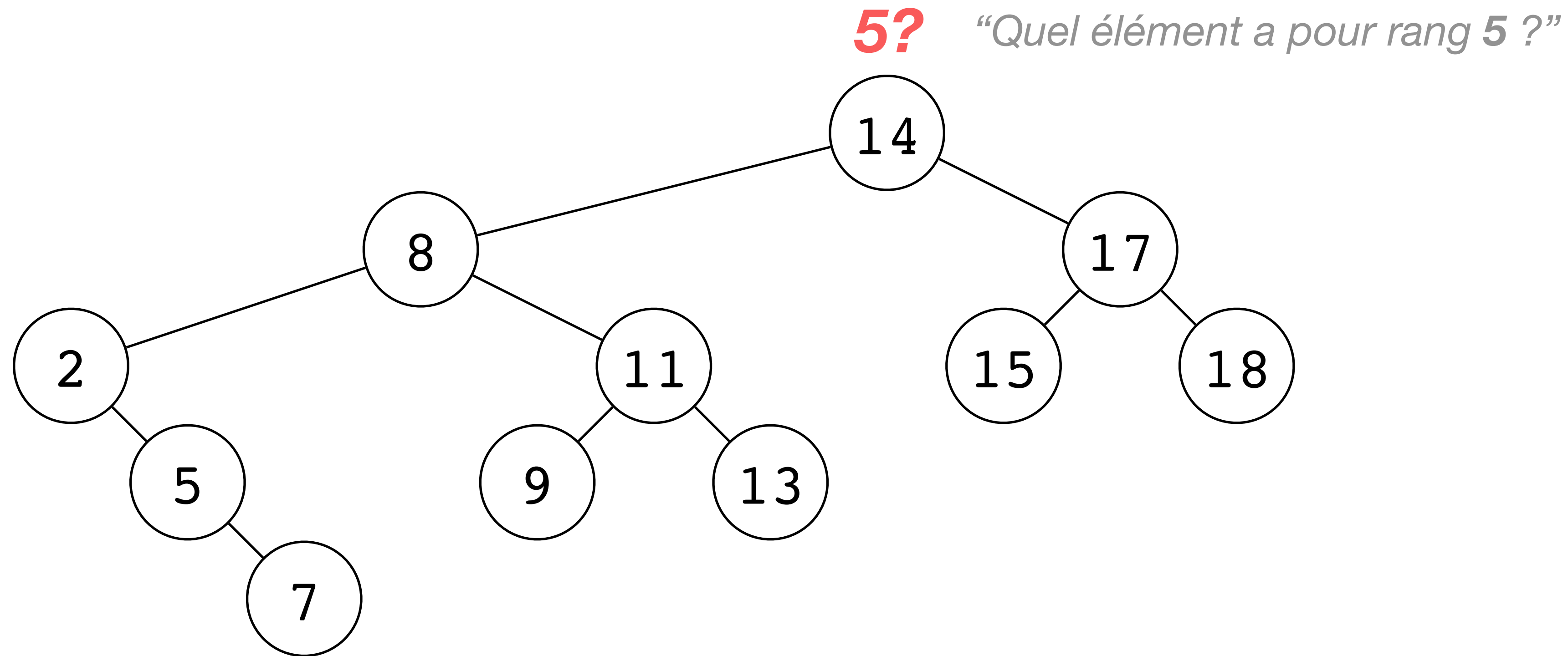
5?

“Quel élément a pour rang 5 ?”





Comment le calculer ?

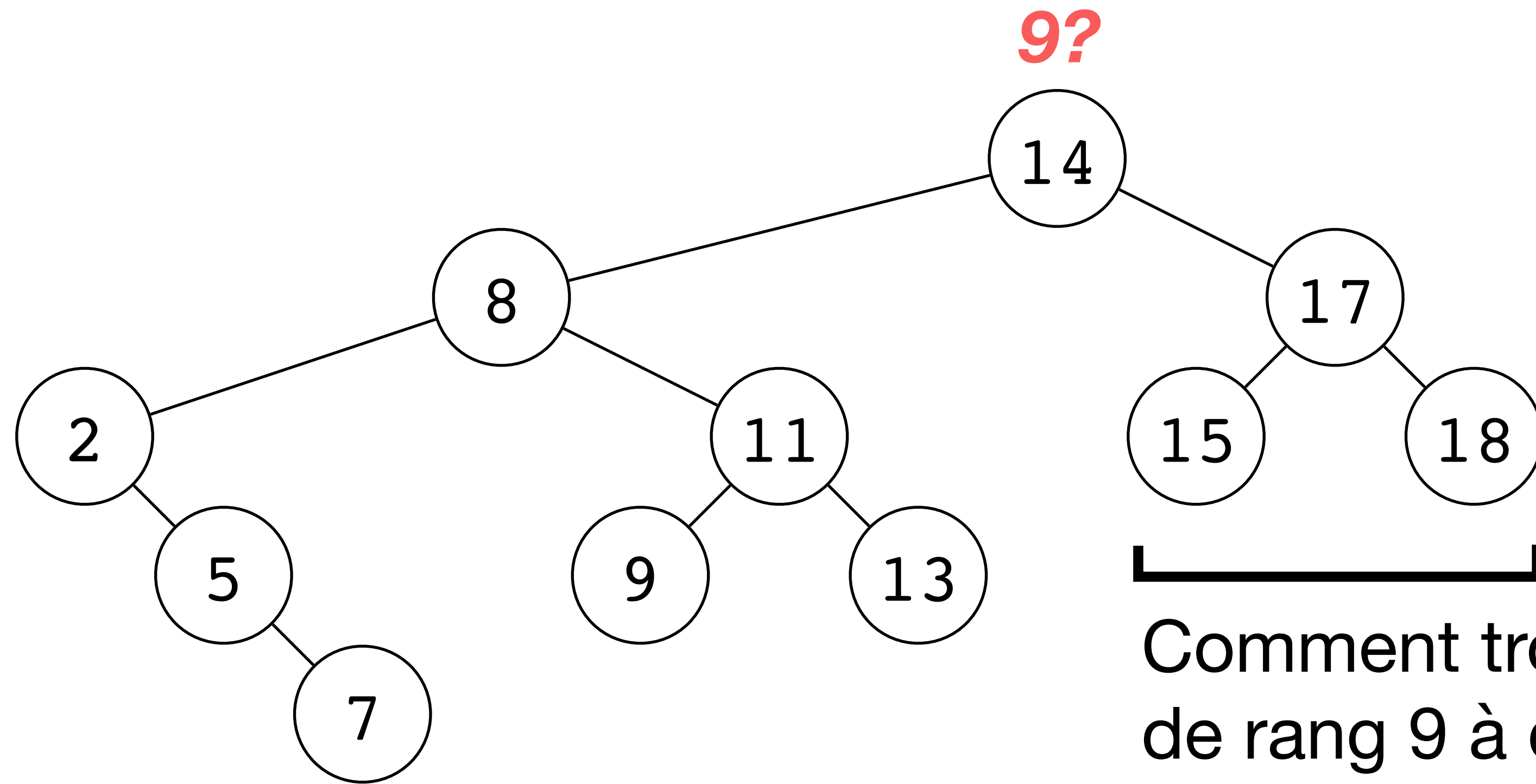


Taille(8) = 7

Rang 5 forcément parmi les 7 premiers.

Appel récursif à gauche

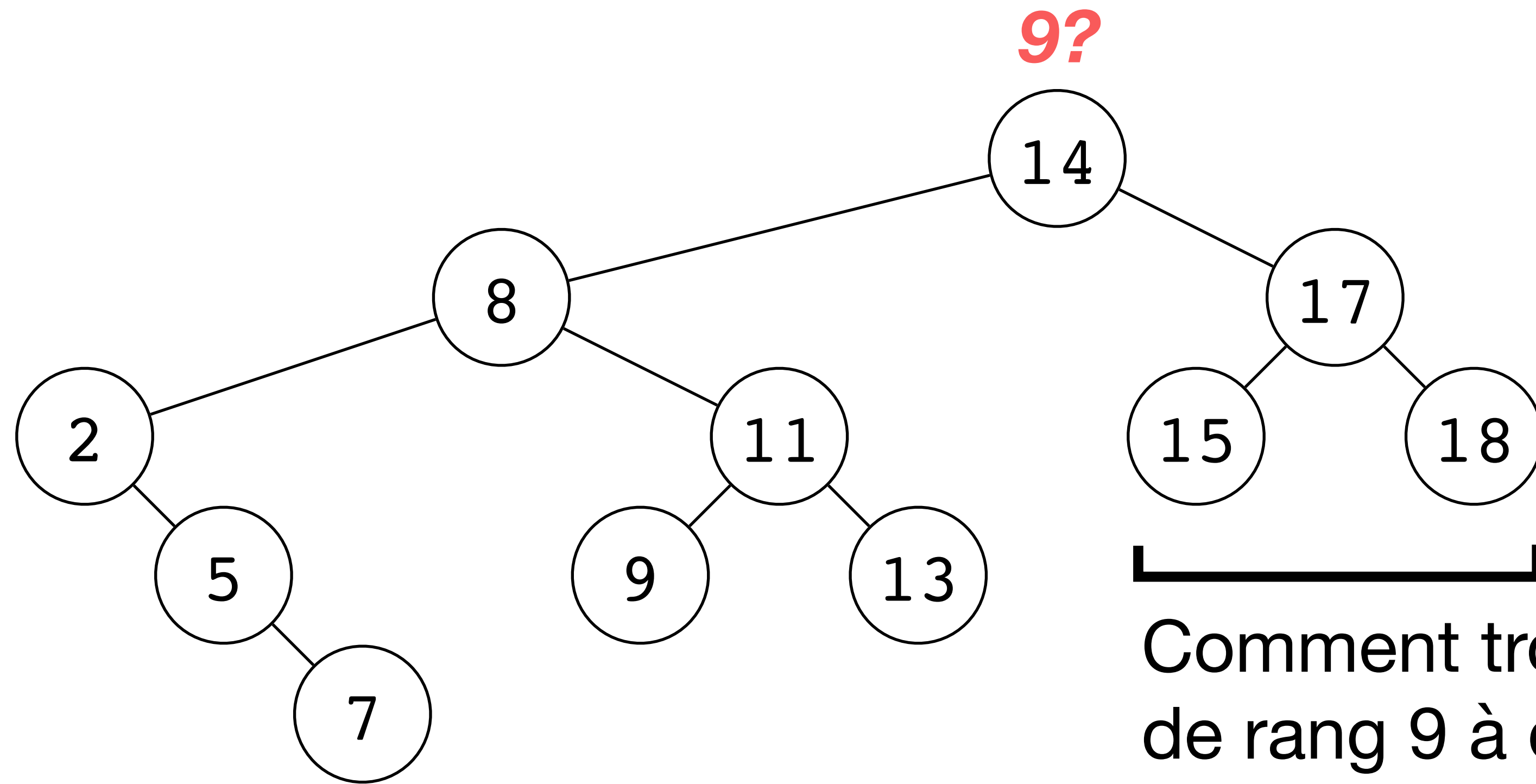
Représentation visuelle



Taille(8) = 7

Rang 9 forcément dans l'autre sous-arbre

Représentation visuelle



Taille(8) = 7

Rang 9 forcément dans l'autre sous-arbre

Comment trouver l'élément
de rang 9 à droite ?

*Dans le sous-arbre 17,
c'est l'élément de rang 1*

(9 - taille(8) - 1)

Appel récursif pour rang 1 à droite

Élément 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)
```

] Trouvé !

] Chercher à gauche

] Chercher à droite

Élément 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)
```

On fait en fait les n premiers pas d'un parcours symétrique en évitant les sous-arbres qui ne nous intéressent pas.

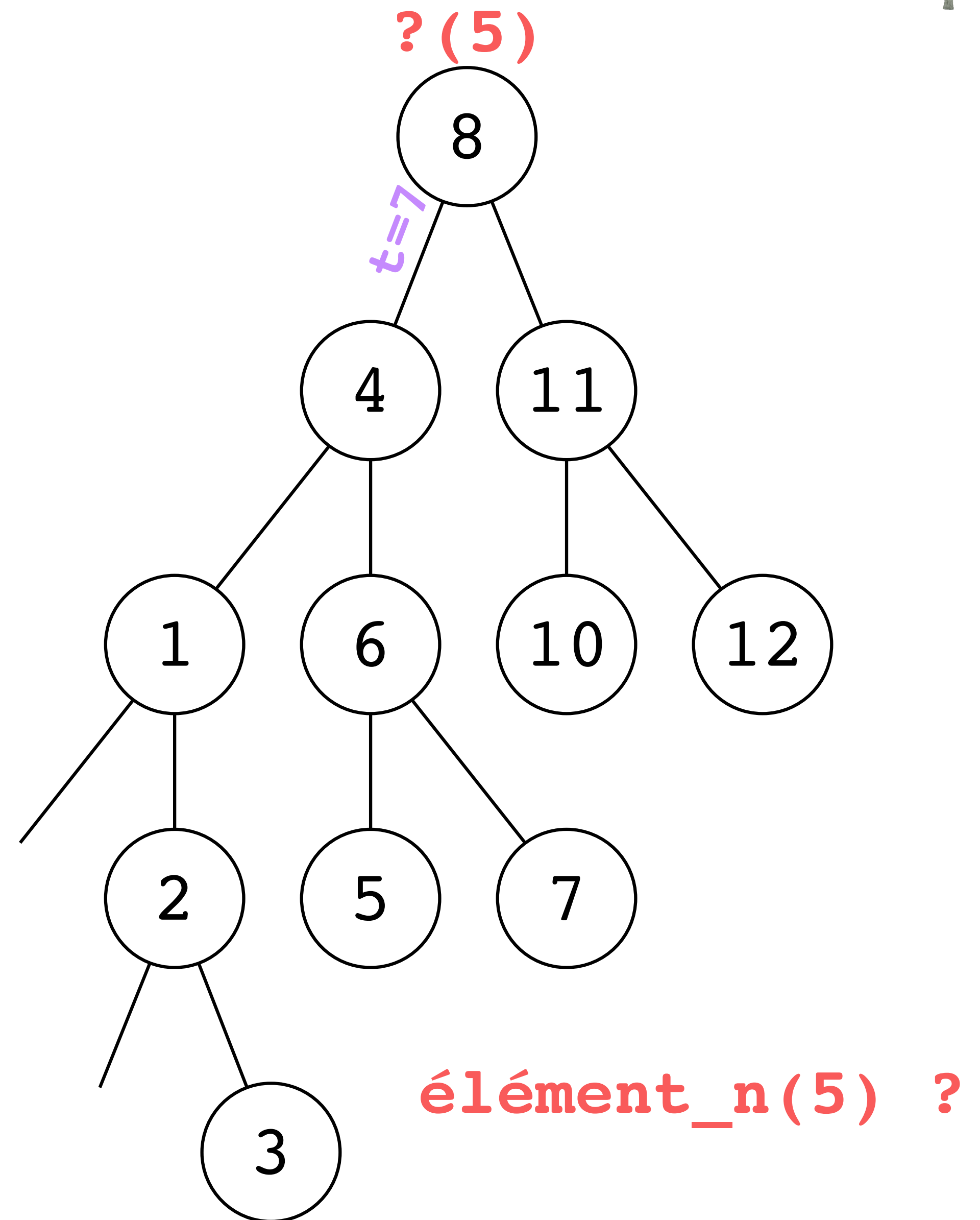
Même approche que pour le calcul du rang d'une clé, mais décisions en fonction du nombre d'éléments plus petits, plutôt que de la clé.

Élément 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)
  
```

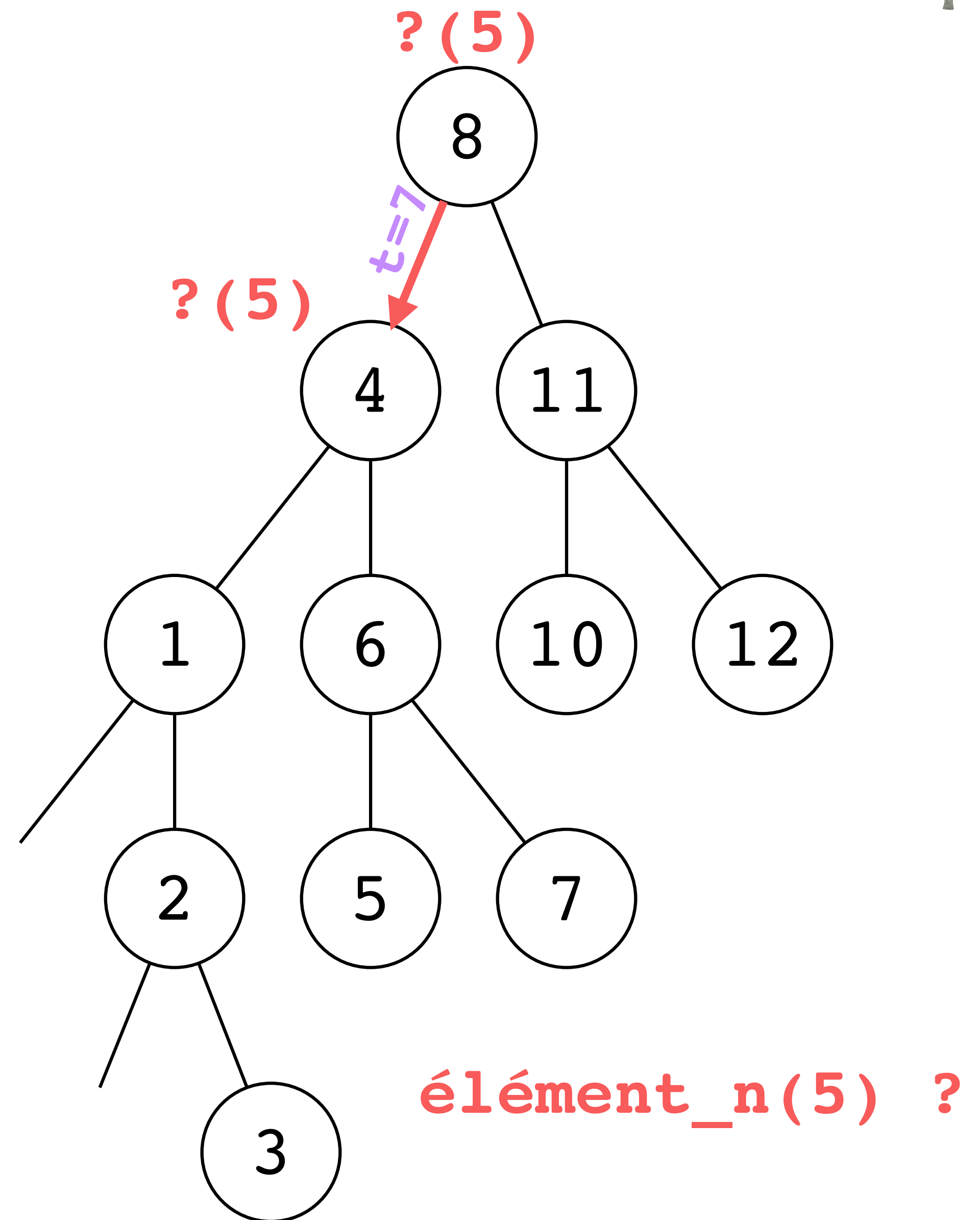


Élément de rang n (2)



```

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)
  
```

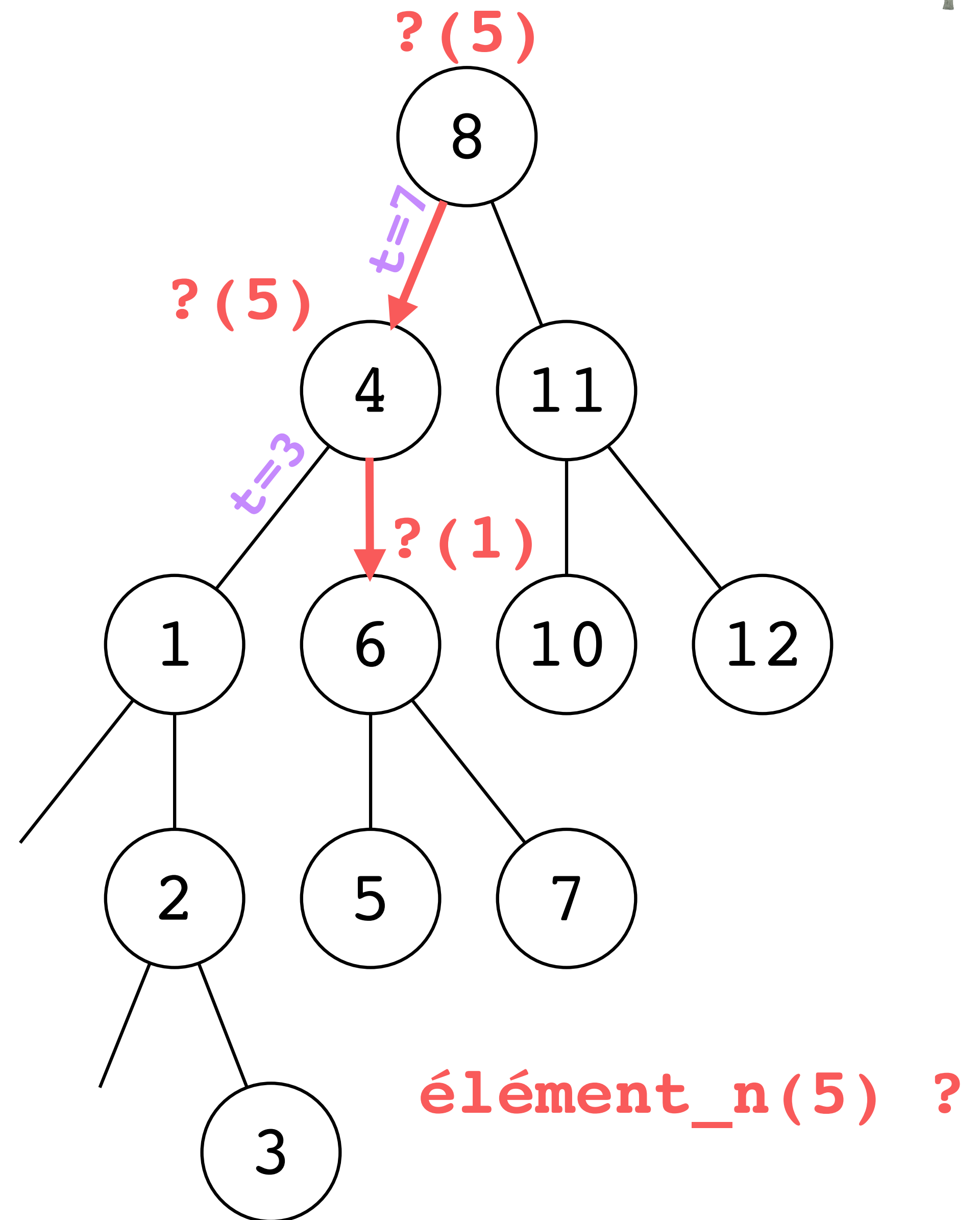


Élément de rang n (2)



```

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)
  
```

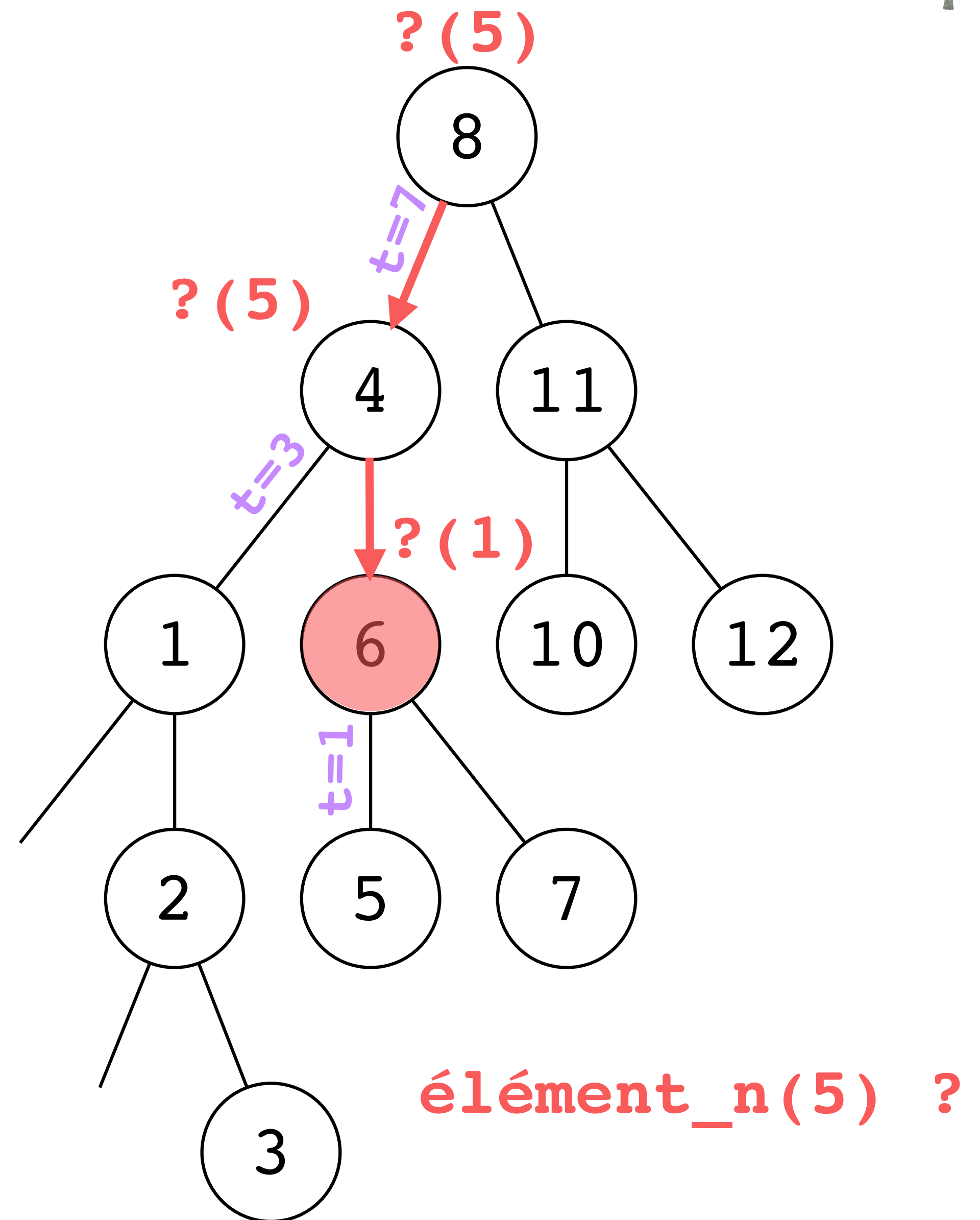


Élément de rang n (2)



```

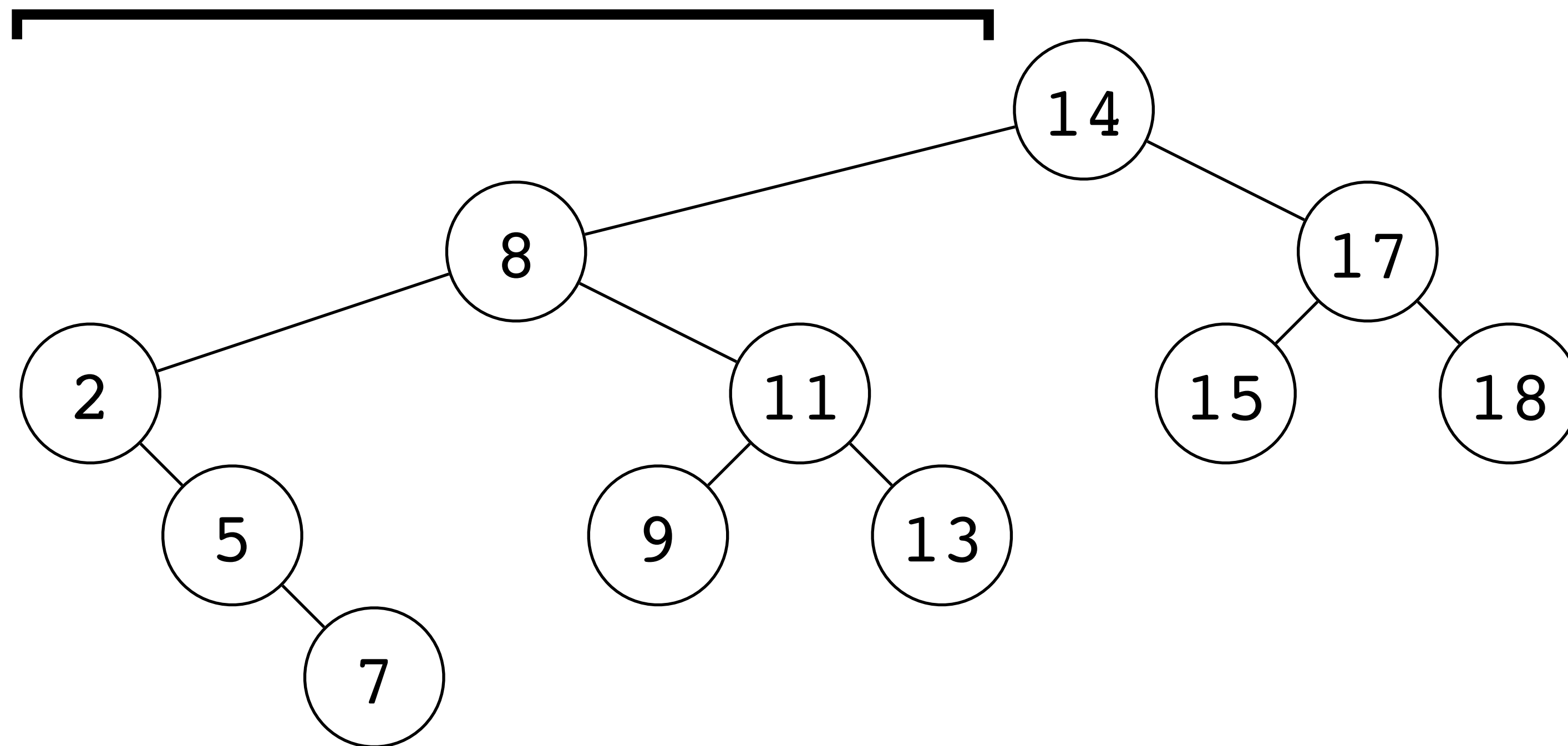
fonction élément_n (r, n)
  si n < 0 ou n ≥ taille(r)
    signaler erreur
  sinon,
    t ← taille(r.gauche)
    si n == t,
      retourner r
    sinon, si n < t,
      retourner élément_n(r.gauche, n)
    sinon, (n > t)
      retourner élément_n(r.droit, n-t-1)
  
```



En résumé



Si l'élément est ici
(car $n < \text{Taille}(8)$)



Racine ↓ *Rang* ↓
↓ ↓

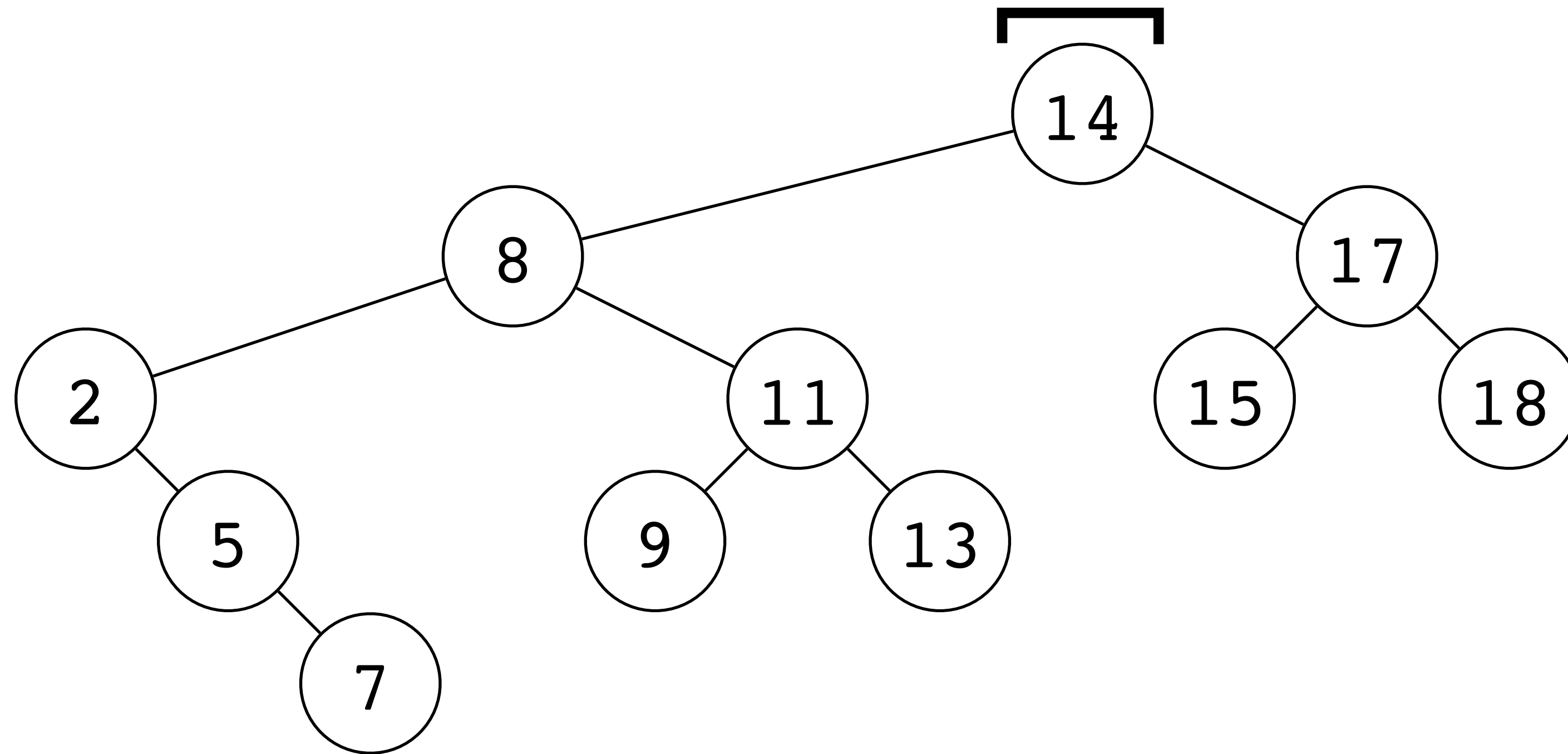
elem_n(14,n) =

elem_n(8,n)

En résumé



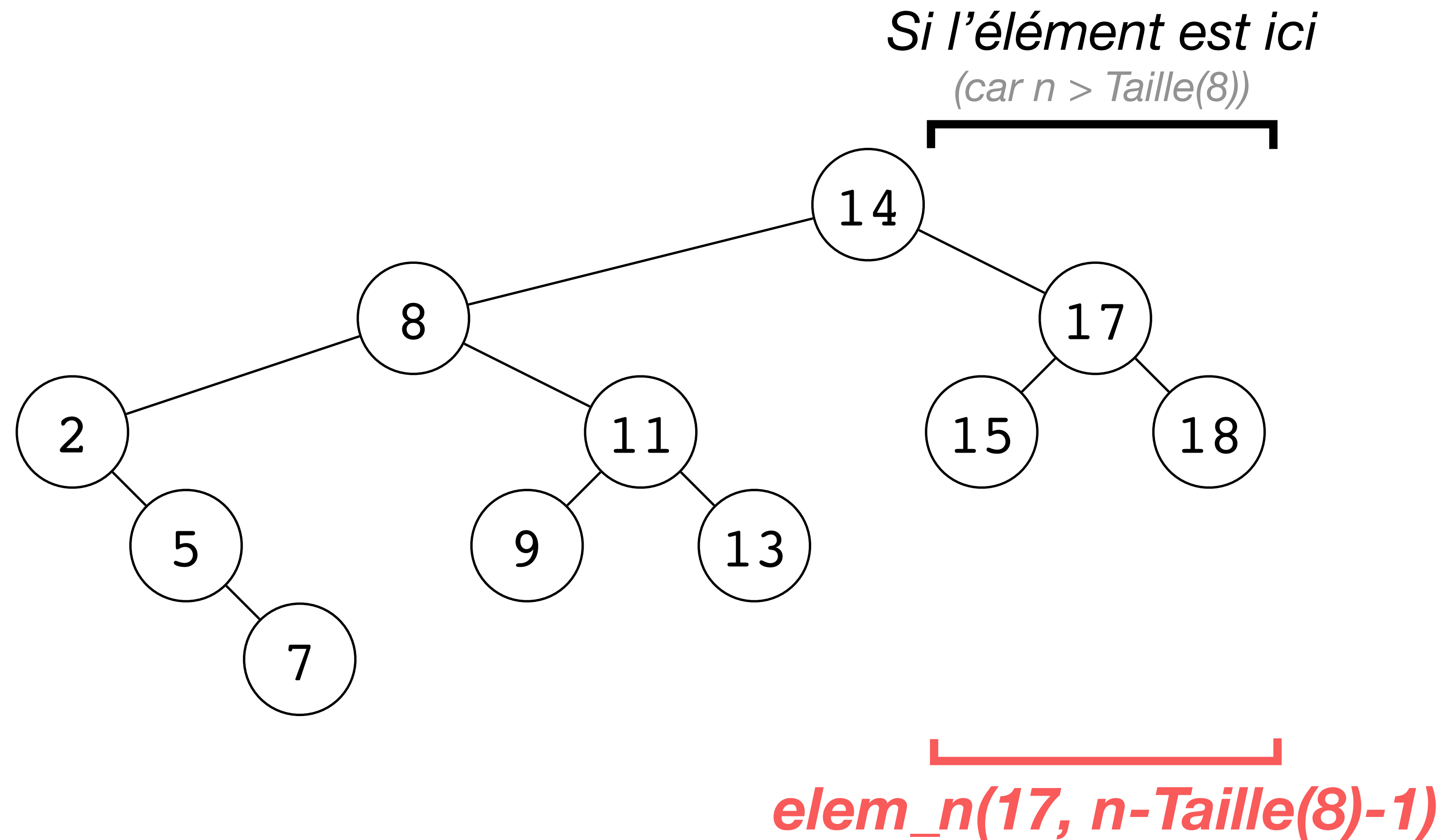
Si l'élément est ici
(car $n = \text{Taille}(8)$)



elem_n(14,n) =

14

En résumé





- Un **Order Statistic Tree** est un 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.