Module EA4 – Éléments d'Algorithmique

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

Rappel – contrôle continu

- interrogation n° 2 *mercredi 1er avril*
  (à la place de l'amphi)
- TP n° 6 *pour tous les groupes* les 14 et 15 avril
- Interrogation n° 3 : TP noté les 5 et 6 mai
- Dernier cours : mercredi 6 mai

Quelle structure de données pour représenter un ensemble ?

Quelle structure de données pour représenter un ensemble ?

Solution 1 : une liste de ses éléments, sans doublon

Quelle structure de données pour représenter un ensemble ?

Solution 1 : une liste de ses éléments, sans doublon

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

Quelle structure de données pour représenter un ensemble ?

Solution 1 : une liste de ses éléments, sans doublon

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
insertion	$+\,\Theta(1)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$

Quelle structure de données pour représenter un ensemble ?

Solution 1 : une liste de ses éléments, sans doublon

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
insertion	$+\,\Theta(1)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

Quelle structure de données pour représenter un ensemble ?

Solution 1 : une liste de ses éléments, sans doublon

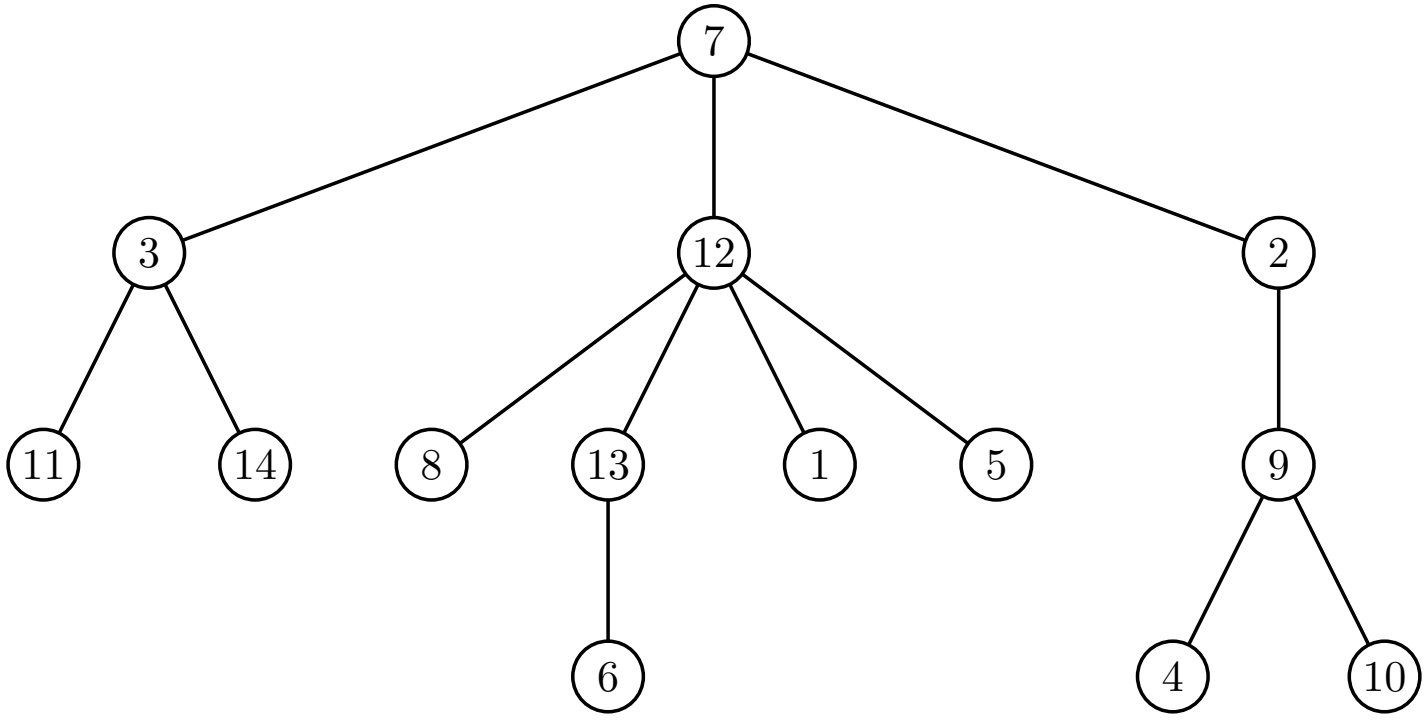
	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
insertion	$+\,\Theta(1)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
sélection du k^e	$\Theta(kn)$	$\Theta(1)$	$\Theta(kn)$	$\Theta(k)$

Quelle structure de données pour représenter un ensemble ?

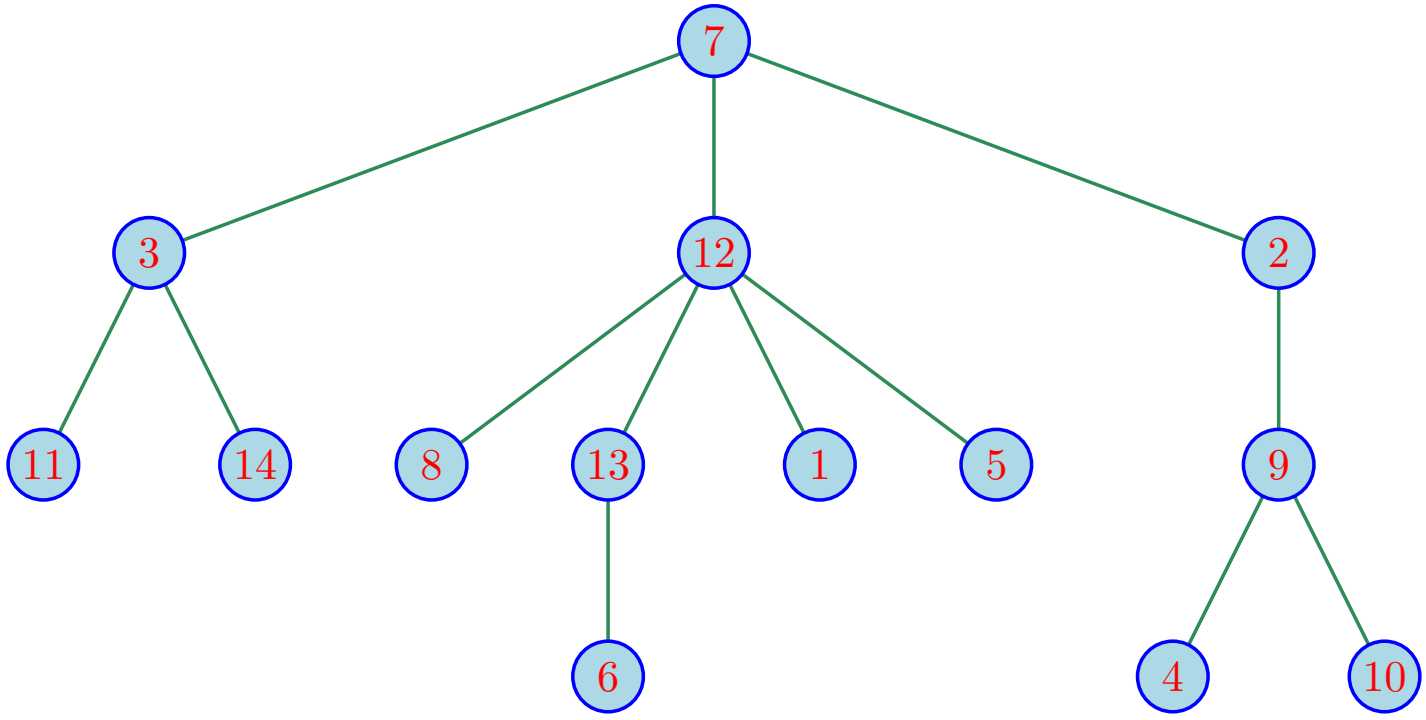
Solution 1 : une liste de ses éléments, sans doublon

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
insertion	$+\Theta(1)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
sélection du k^e	$\Theta(kn)$	$\Theta(1)$	$\Theta(kn)$	$\Theta(k)$
union	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$

Idée : utiliser des arbres

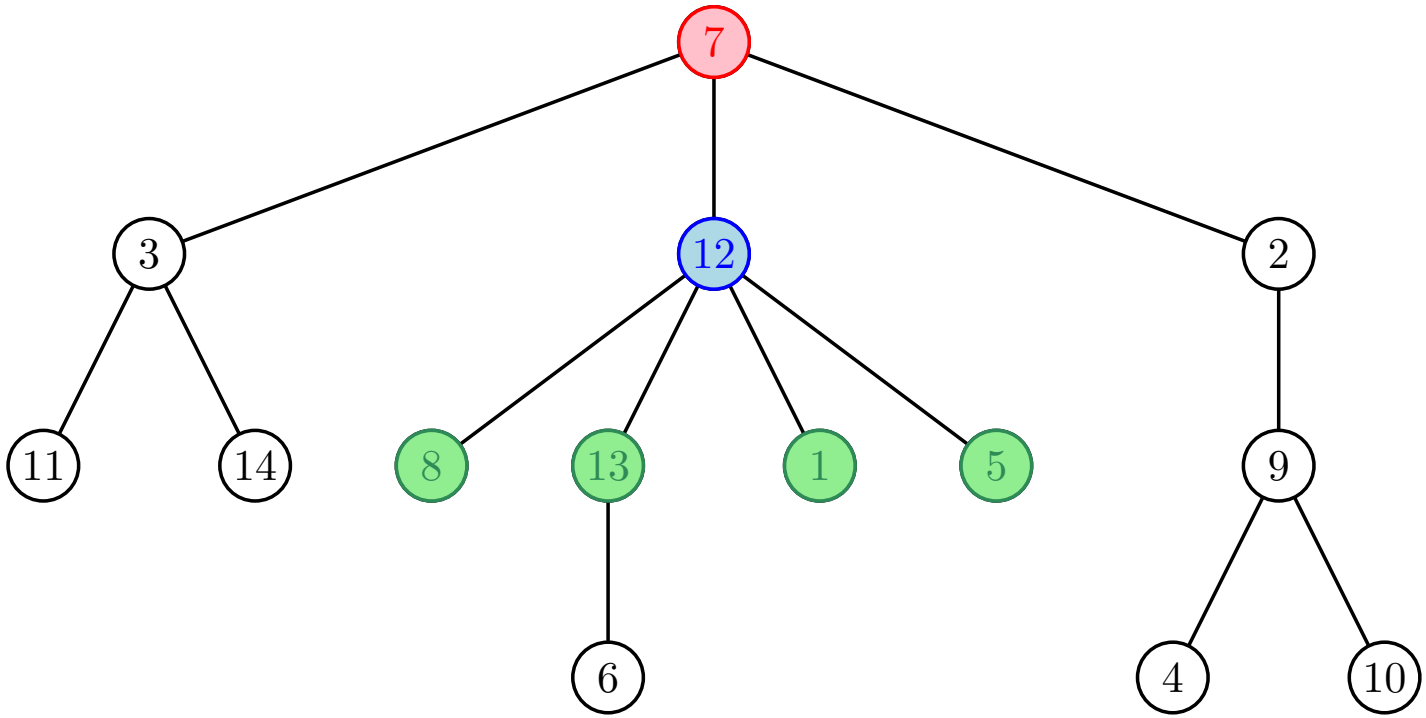


Idée : utiliser des arbres



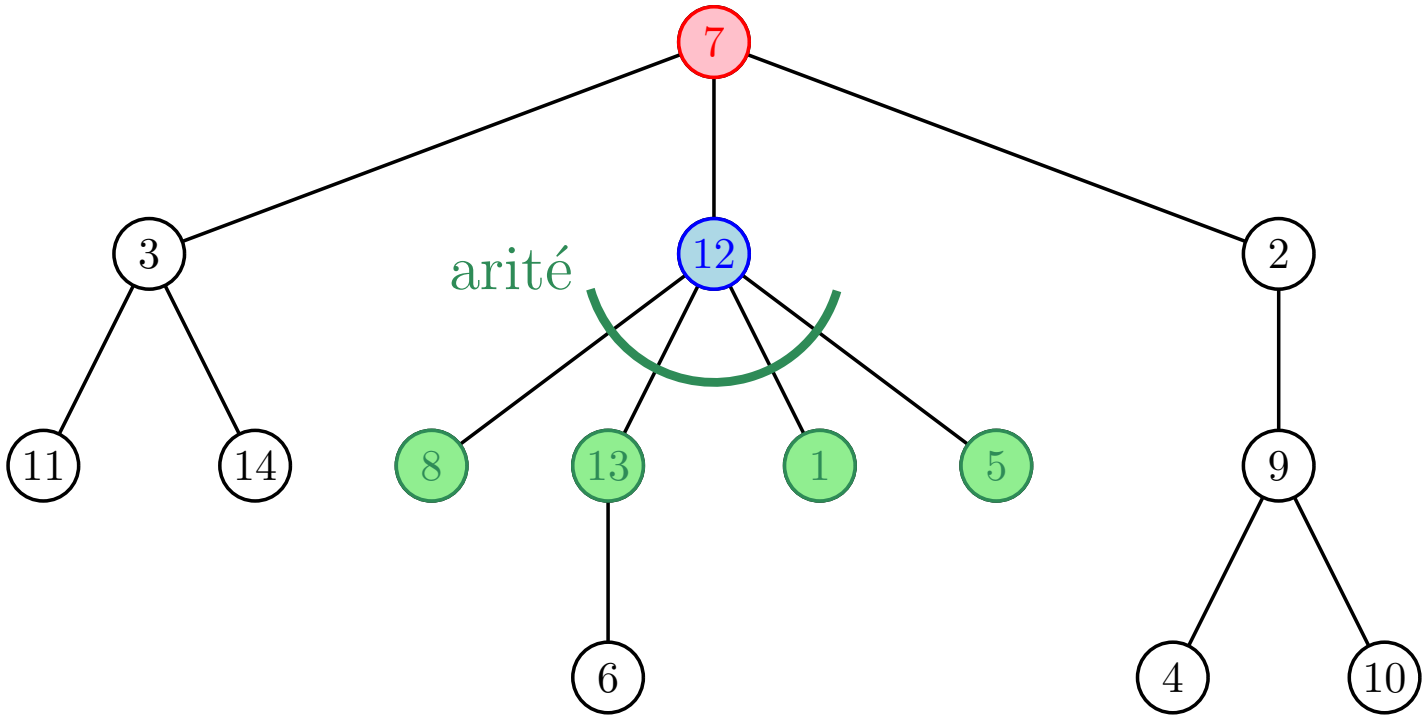
sommets contenant des étiquettes reliés par des arêtes

Idée : utiliser des arbres



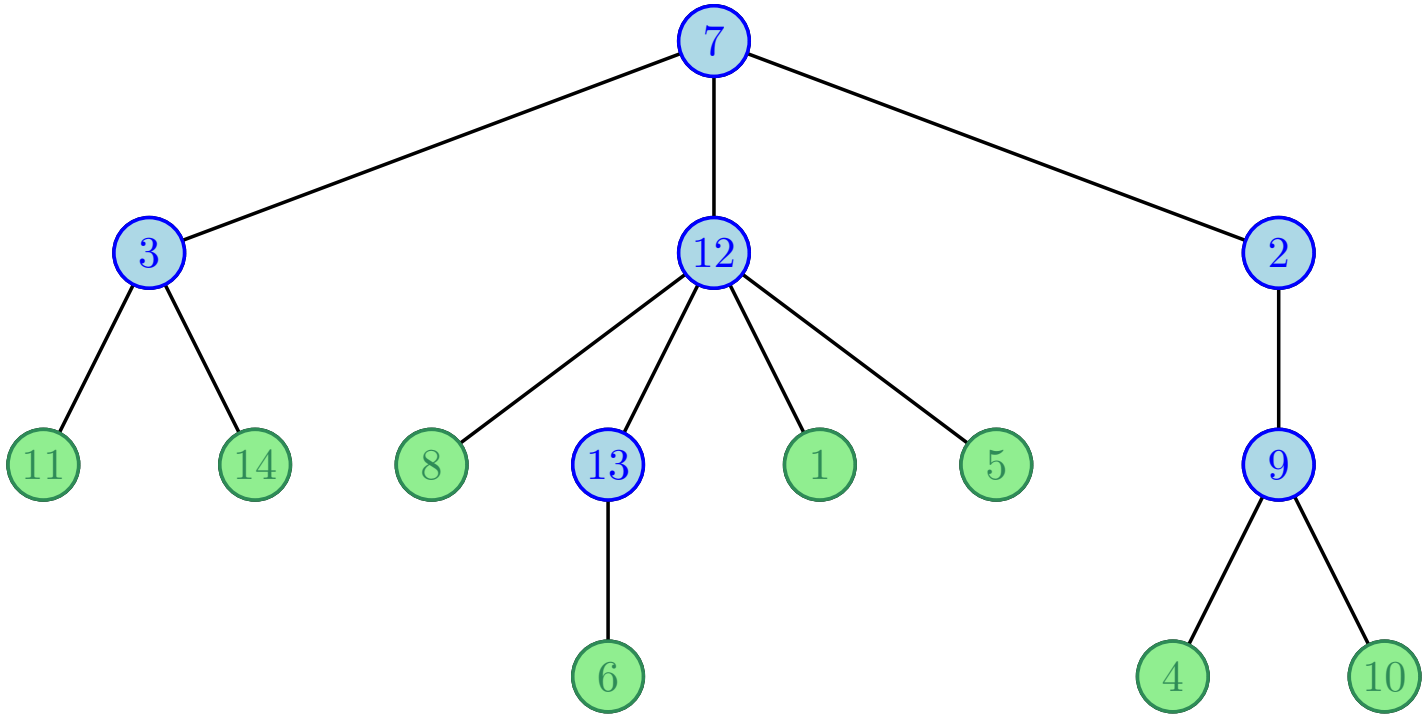
hiérarchie entre les sommets : père , fils

Idée : utiliser des arbres



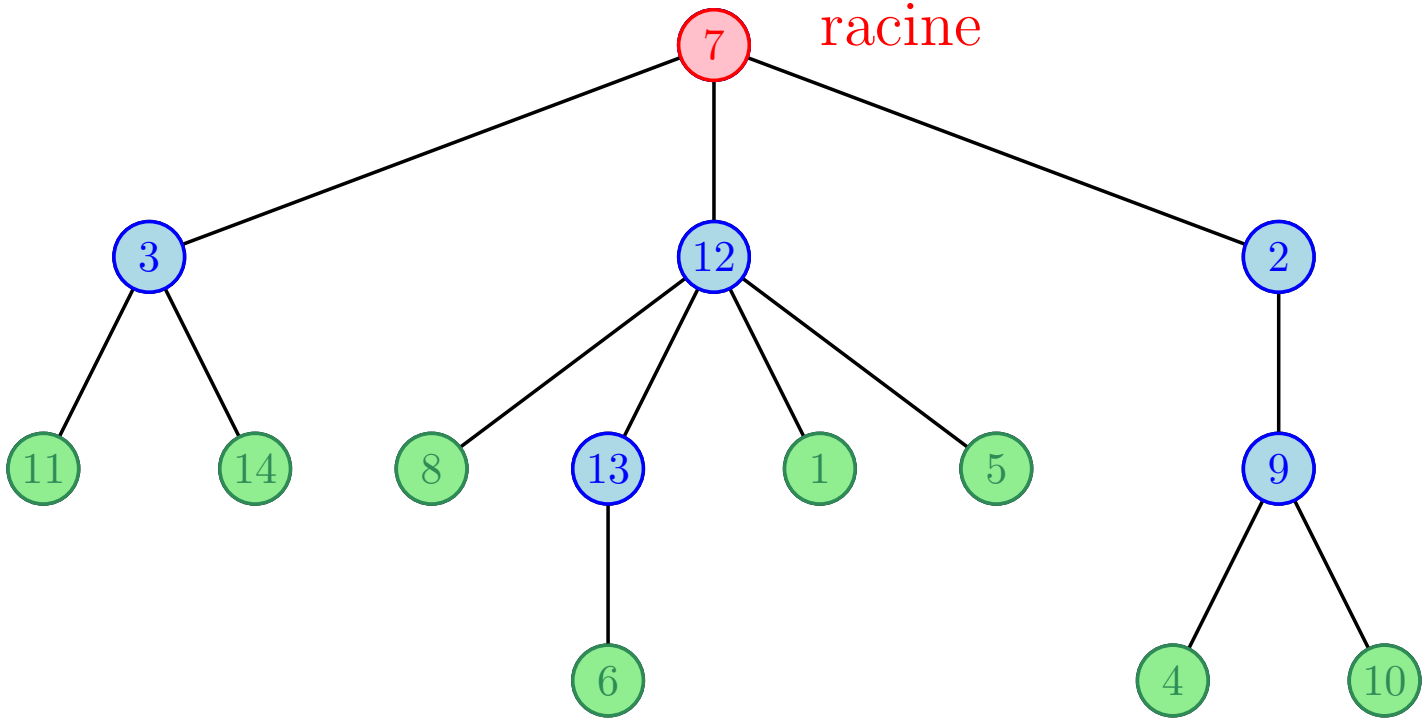
hiérarchie entre les sommets : père , fils

Idée : utiliser des arbres



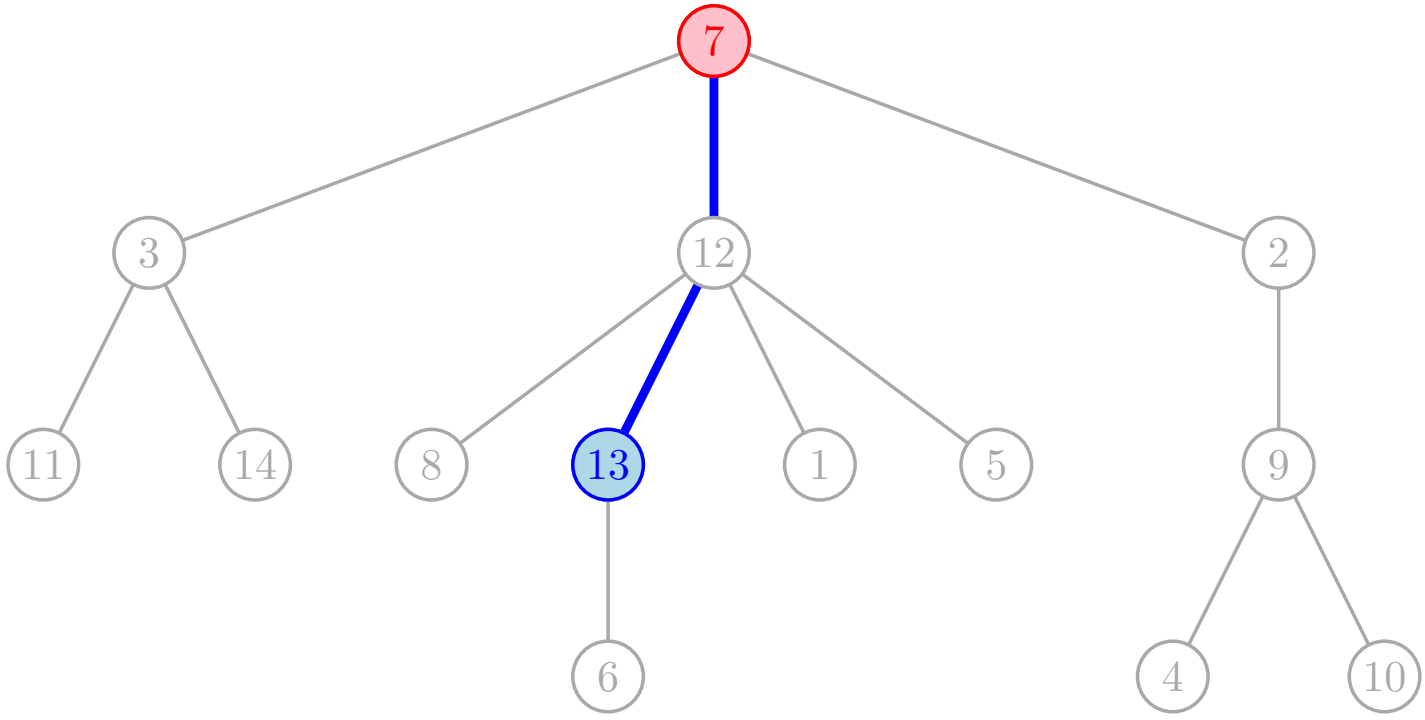
sommet = nœud ou feuille

Idée : utiliser des arbres



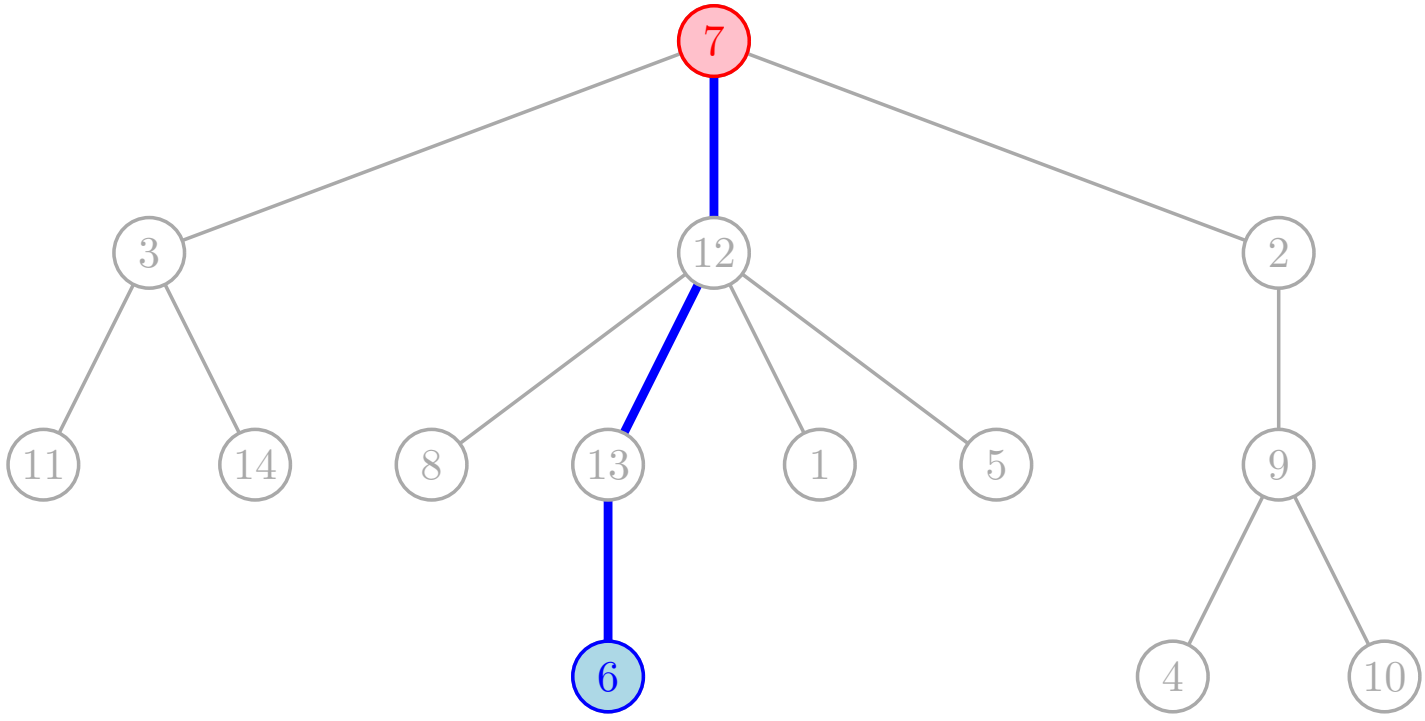
sommet = nœud ou feuille

Idée : utiliser des arbres



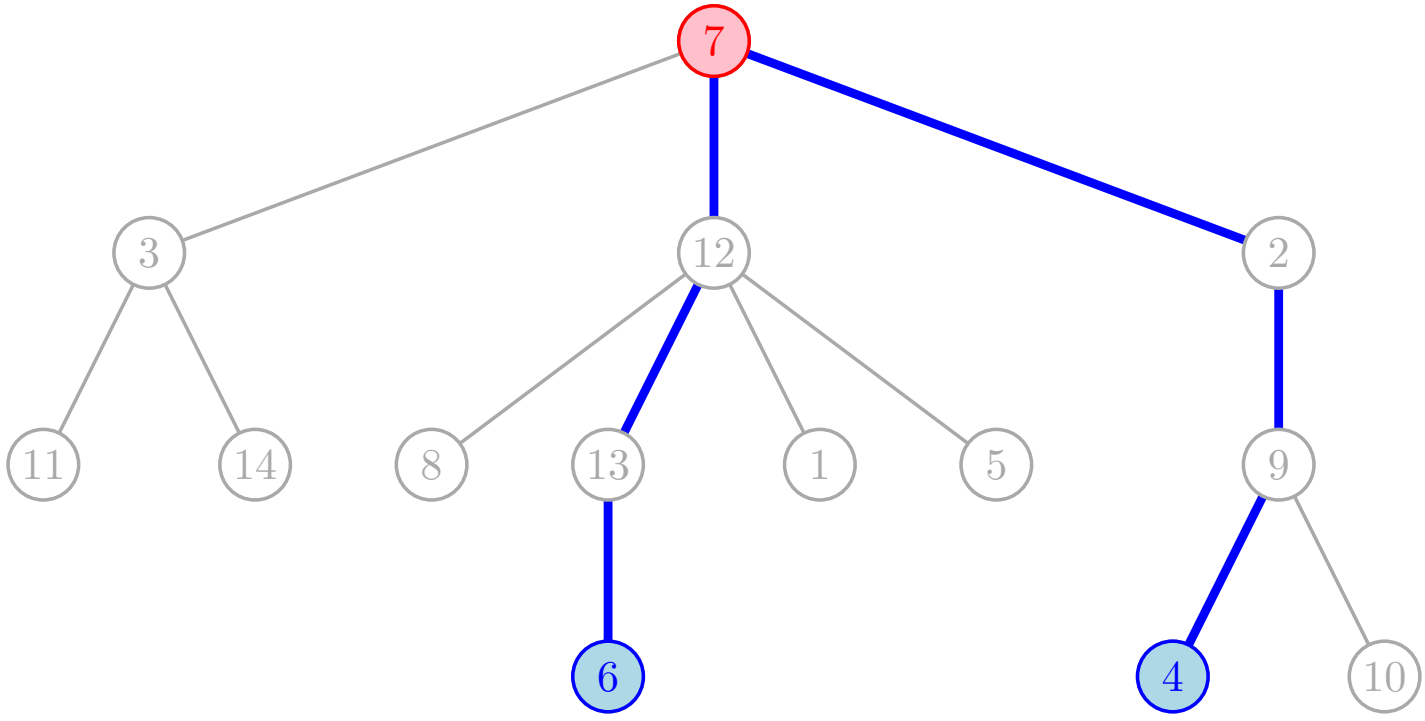
profondeur d'un sommet = distance à la racine

Idée : utiliser des arbres



hauteur de l'arbre = profondeur maximale

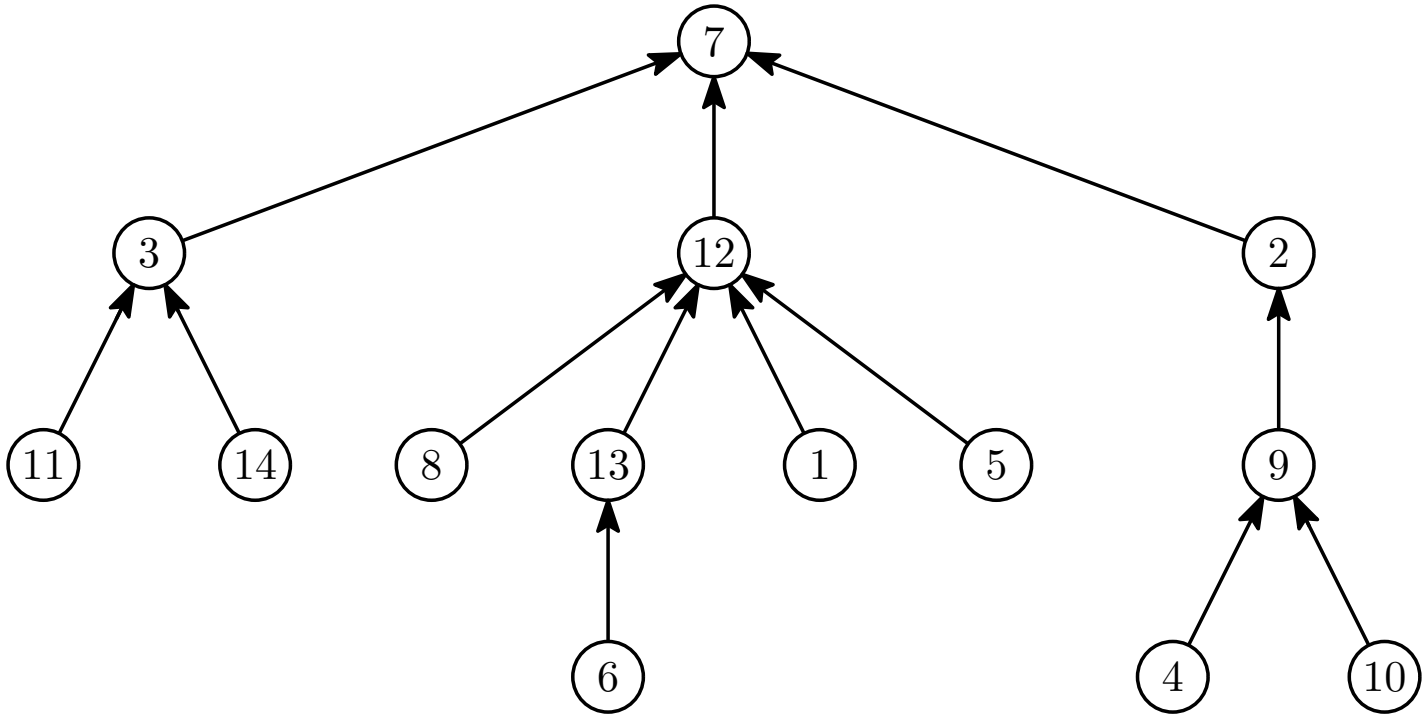
Idée : utiliser des arbres



hauteur de l'arbre = profondeur maximale

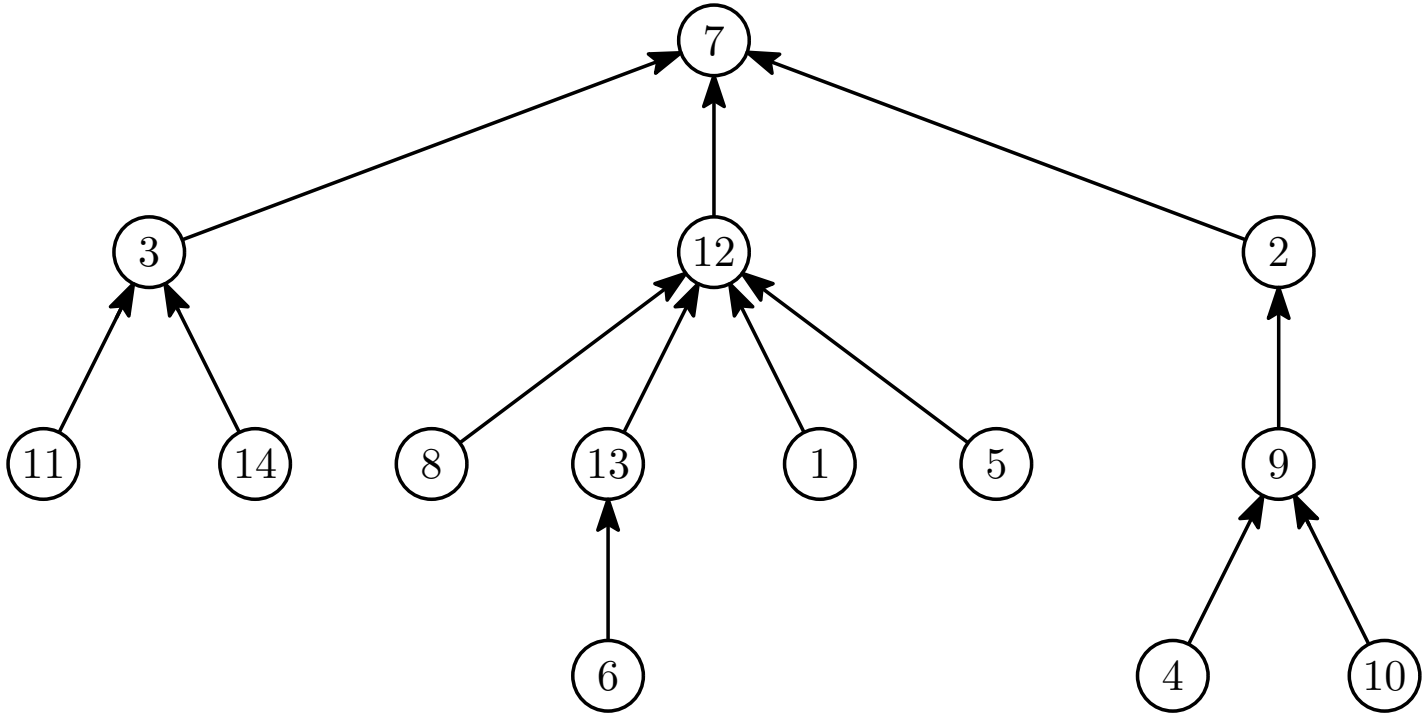
Représentation des arbres

pointeur vers le père (dans chaque nœud, ou dans un tableau)



Représentation des arbres

pointeur vers le père (dans chaque nœud, ou dans un tableau)



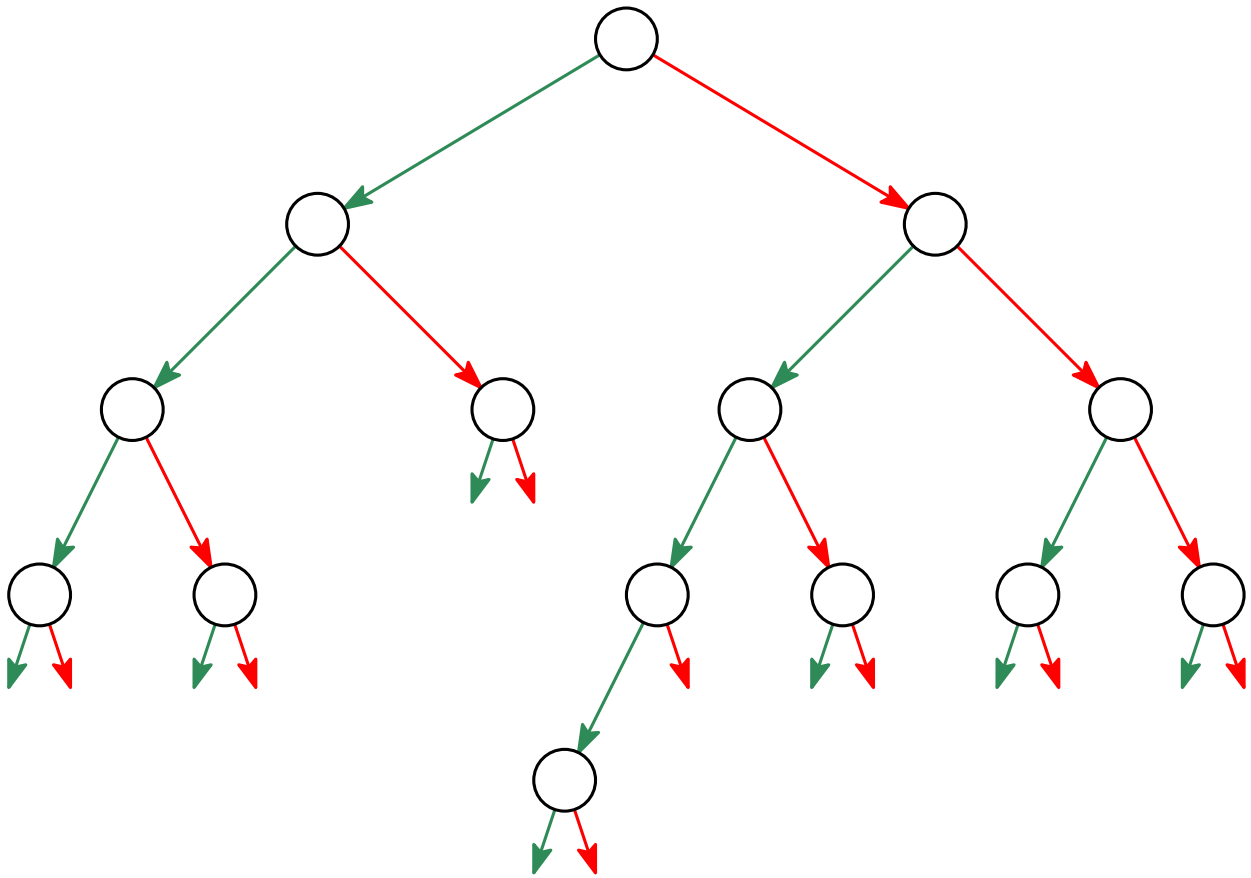
avantage : représentation très compacte

inconvénient : ne peut être parcouru que de bas en haut

$\Longrightarrow$ utilisée pour représenter une partition en sous-ensembles (structure *union-find*)

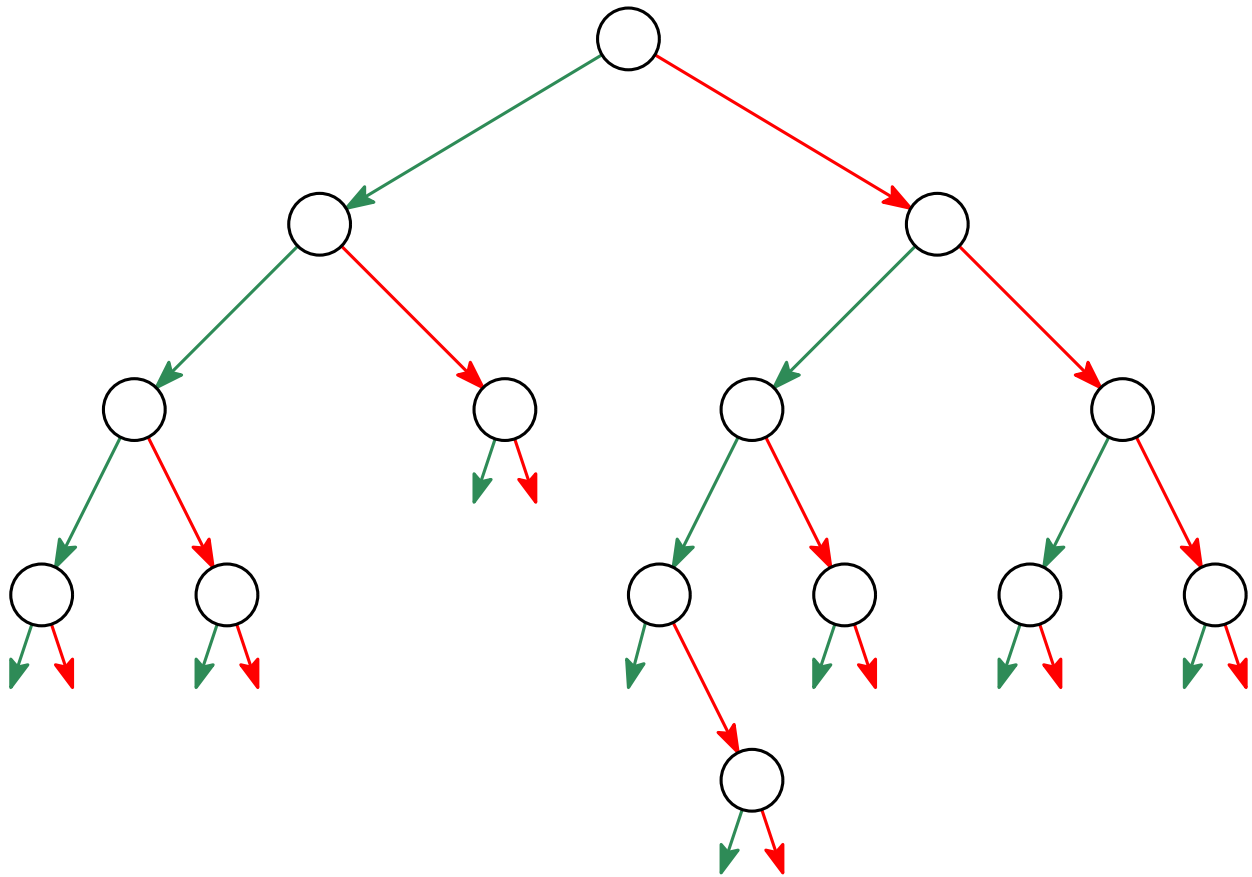
Représentation des arbres

cas des arbres à arité fixe (en particulier les arbres binaires) :
un pointeur vers chaque fils (plus éventuellement le père)



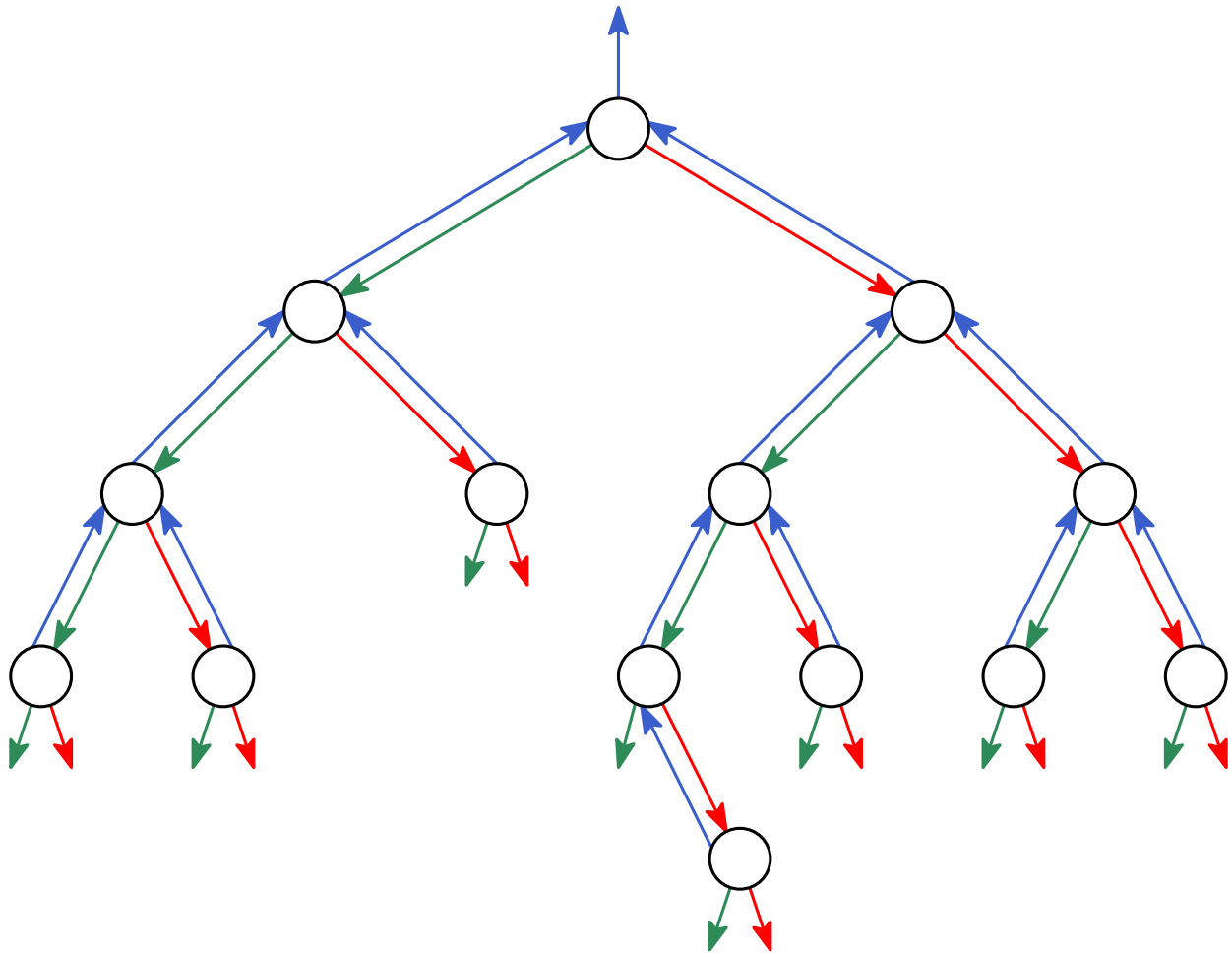
Représentation des arbres

cas des arbres à arité fixe (en particulier les arbres binaires) :
un pointeur vers chaque fils (plus éventuellement le père)



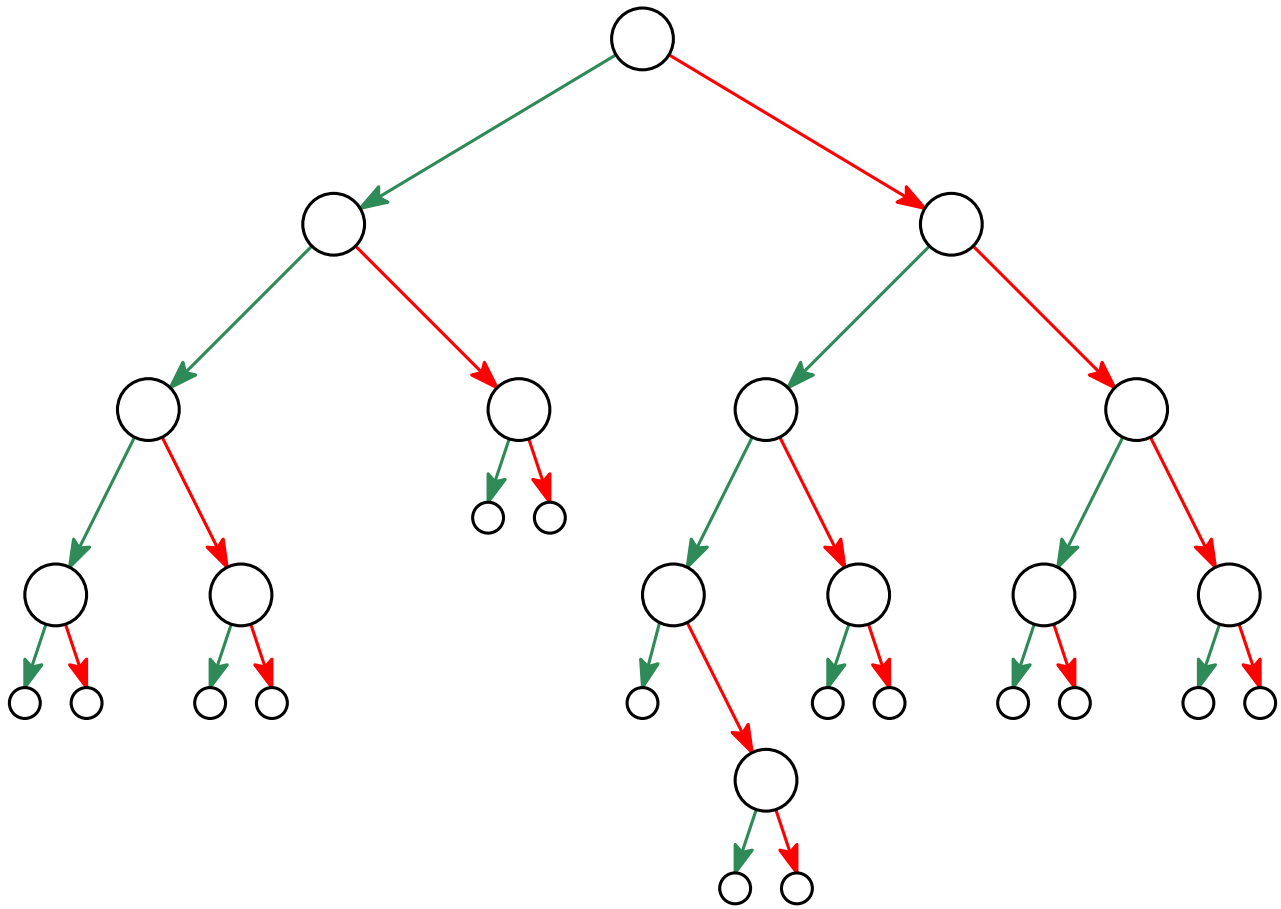
Représentation des arbres

cas des arbres à arité fixe (en particulier les arbres binaires) :
un pointeur vers chaque fils (plus éventuellement le père)



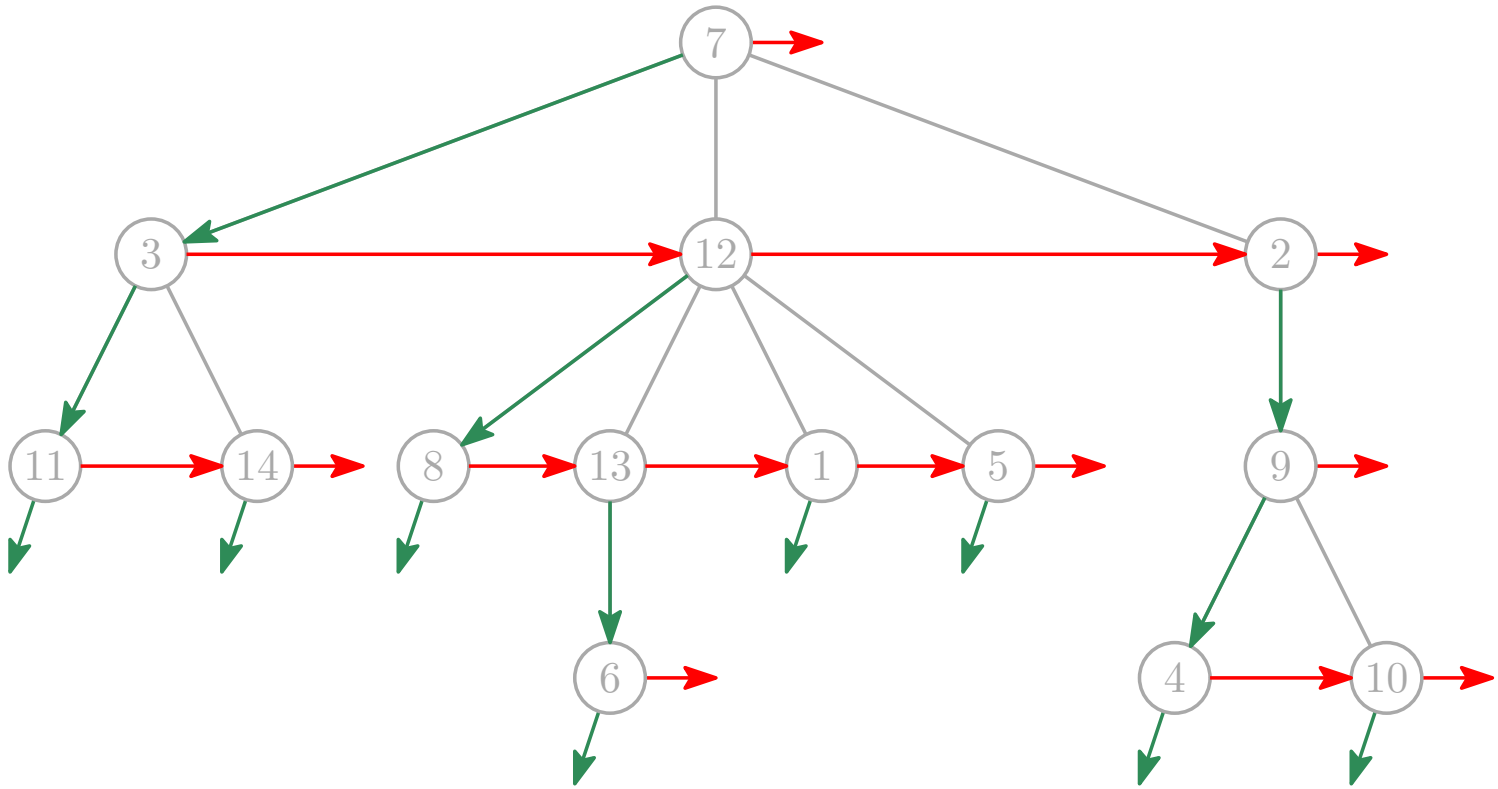
Représentation des arbres

cas des arbres à arité fixe (en particulier les arbres binaires) :
un pointeur vers chaque fils (plus éventuellement le père)



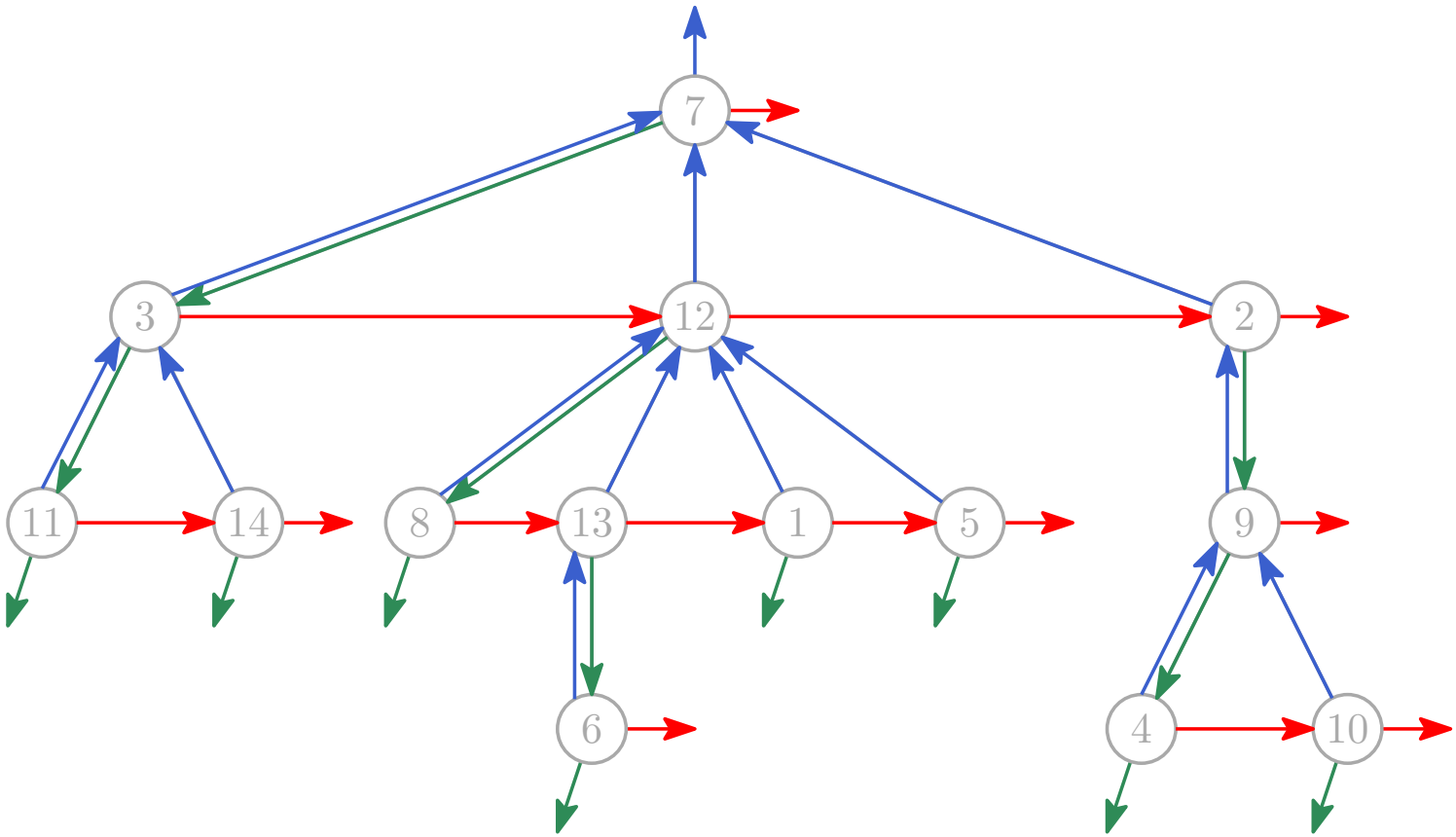
Représentation des arbres

cas général : pointeurs vers le fils aîné et le frère cadet



Représentation des arbres

cas général : pointeurs vers le fils aîné et le frère cadet (plus éventuellement le père)

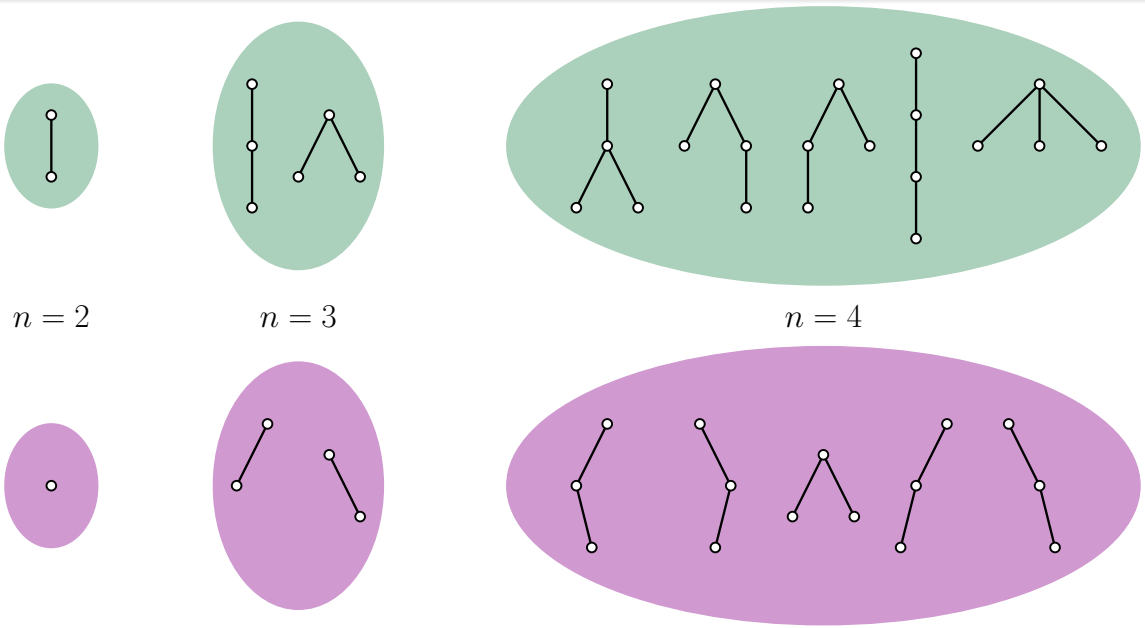


Théorème

Les arbres à n sommets sont en bijection avec les arbres binaires à $n-1$ sommets

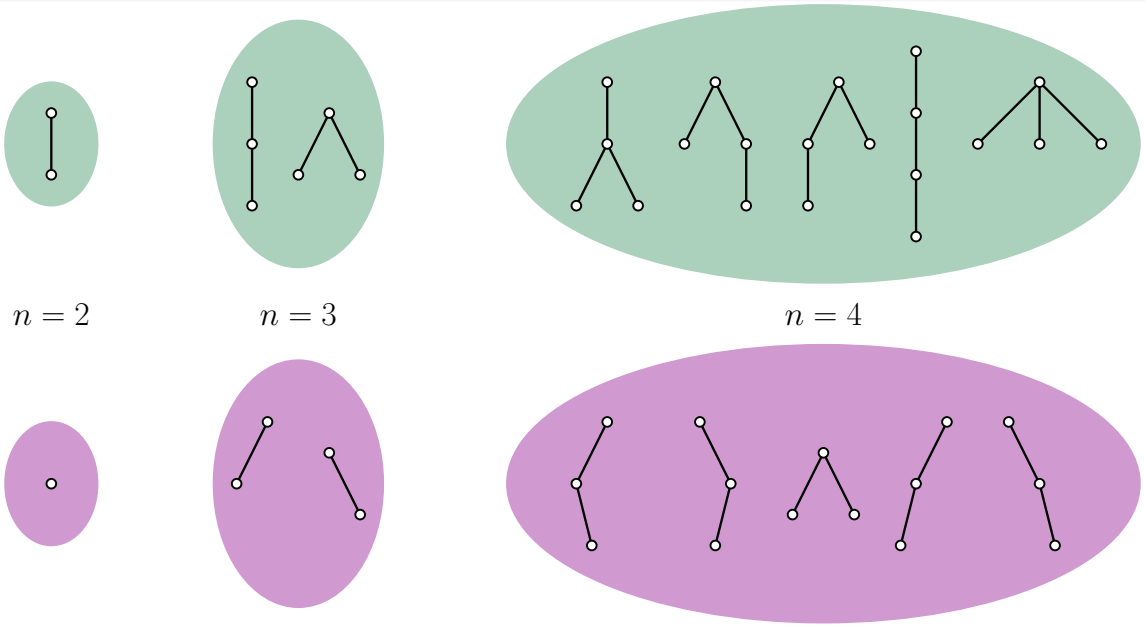
Théorème

Les arbres à n sommets sont en bijection avec les arbres binaires à $n-1$ sommets



Théorème

Les arbres à n sommets sont en bijection avec les arbres binaires à $n-1$ sommets



Théorème (*admis*)

Le nombre d'arbres à n sommets est $\frac{1}{n+1}\binom{2n}{n}$

Représentation des arbres

À partir de maintenant, on suppose qu'on dispose des fonctions suivantes, dont le code dépend de la représentation choisie :

- `pere(noeud)`
- `liste_des_fils(noeud)`
- `etiquette(noeud)`
- `gauche(noeud)` et `droite(noeud)`

Parcours en profondeur générique

```
def parcours(racine) :
  pre_traitement(racine)
  for i, noeud in enumerate(liste_des_fils(racine)) :
    parcours(noeud)
    post_traitement(i, racine)
```

Parcours en profondeur générique

```
def parcours(racine) :
  pre_traitement(racine)
  for i, noeud in enumerate(liste_des_fils(racine)) :
    parcours(noeud)
    post_traitement(i, racine)
```

Théorème

`parcours(racine)` visite tous les nœuds de l'arbre enraciné en `racine`, en temps $\Theta(n)$ si chaque traitement est de coût $\Theta(1)$

Parcours en profondeur générique

```
def parcours(racine) :
  pre_traitement(racine)
  for i, noeud in enumerate(liste_des_fils(racine)) :
    parcours(noeud)
    post_traitement(i, racine)
```

Théorème

`parcours(racine)` visite tous les nœuds de l'arbre enraciné en `racine`, en temps $\Theta(n)$ si chaque traitement est de coût $\Theta(1)$

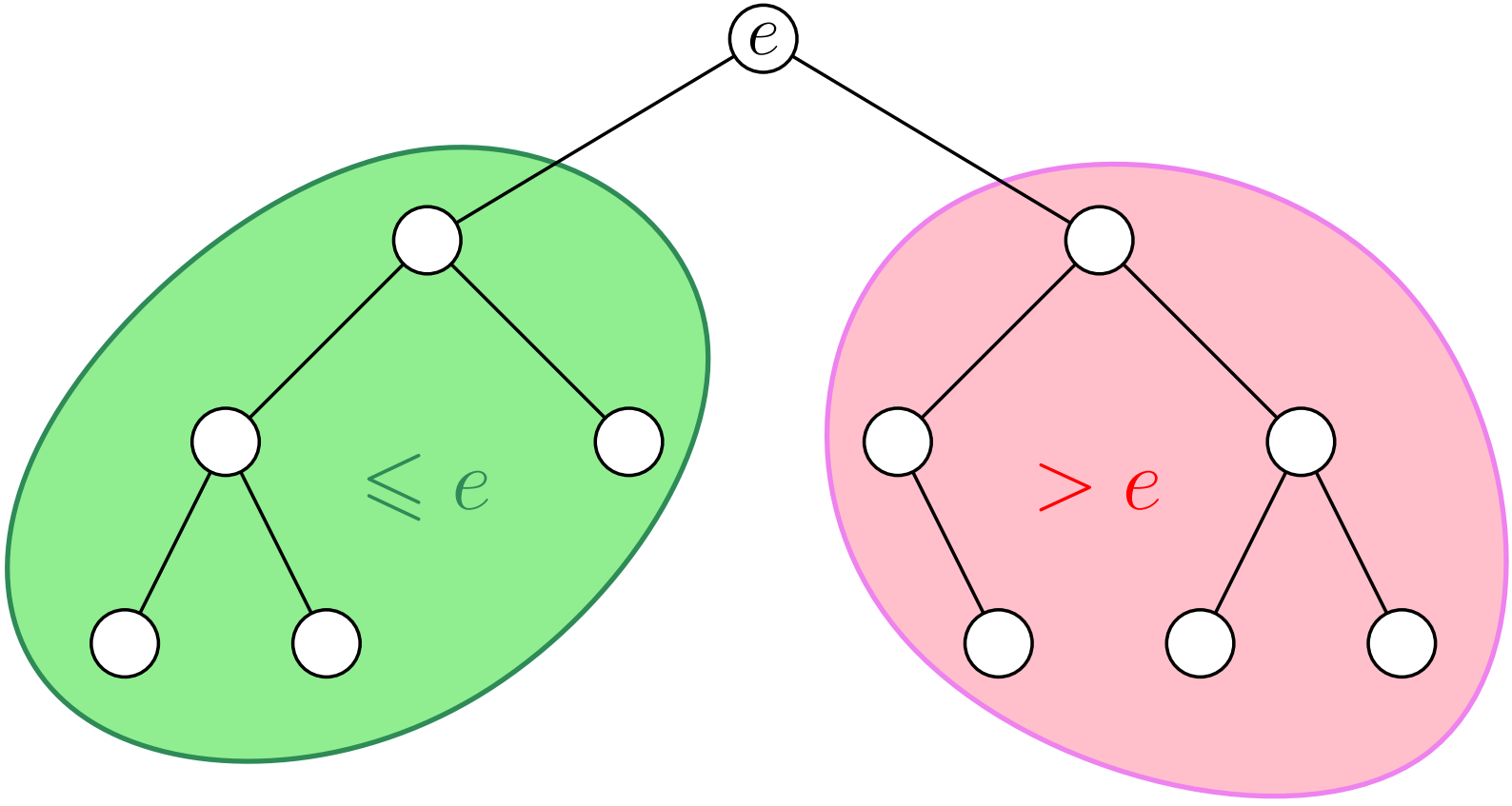
variation selon les traitements intermédiaires :

- s'il y a seulement un prétraitement : *parcours préfixe*
- s'il y a seulement un posttraitement : *parcours postfixe*
- dans le cas binaire, s'il y a seulement un traitement intermédiaire : *parcours infixe*

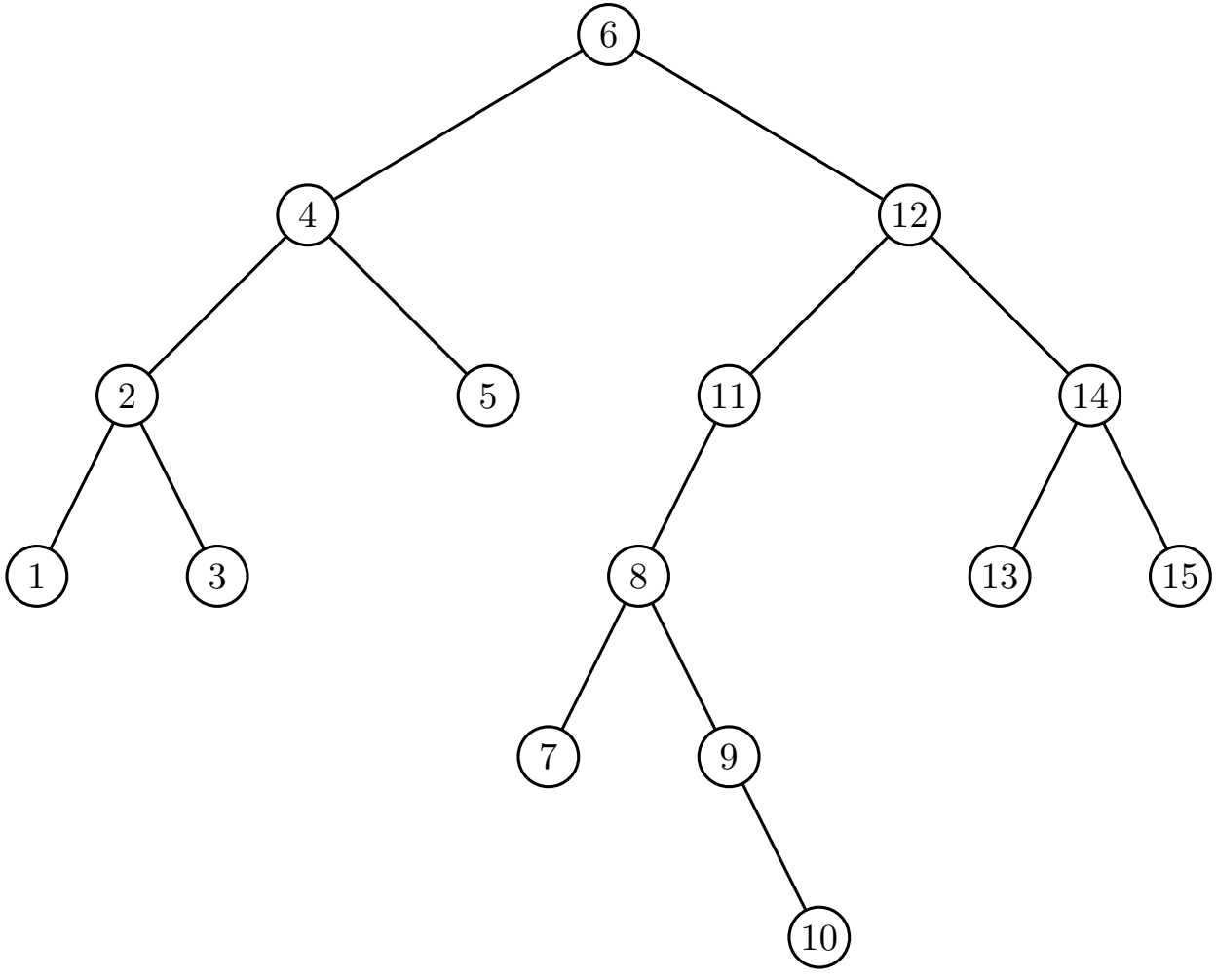
les arbres binaires de recherche (ABR)

en chaque nœud, l'étiquette est comprise entre

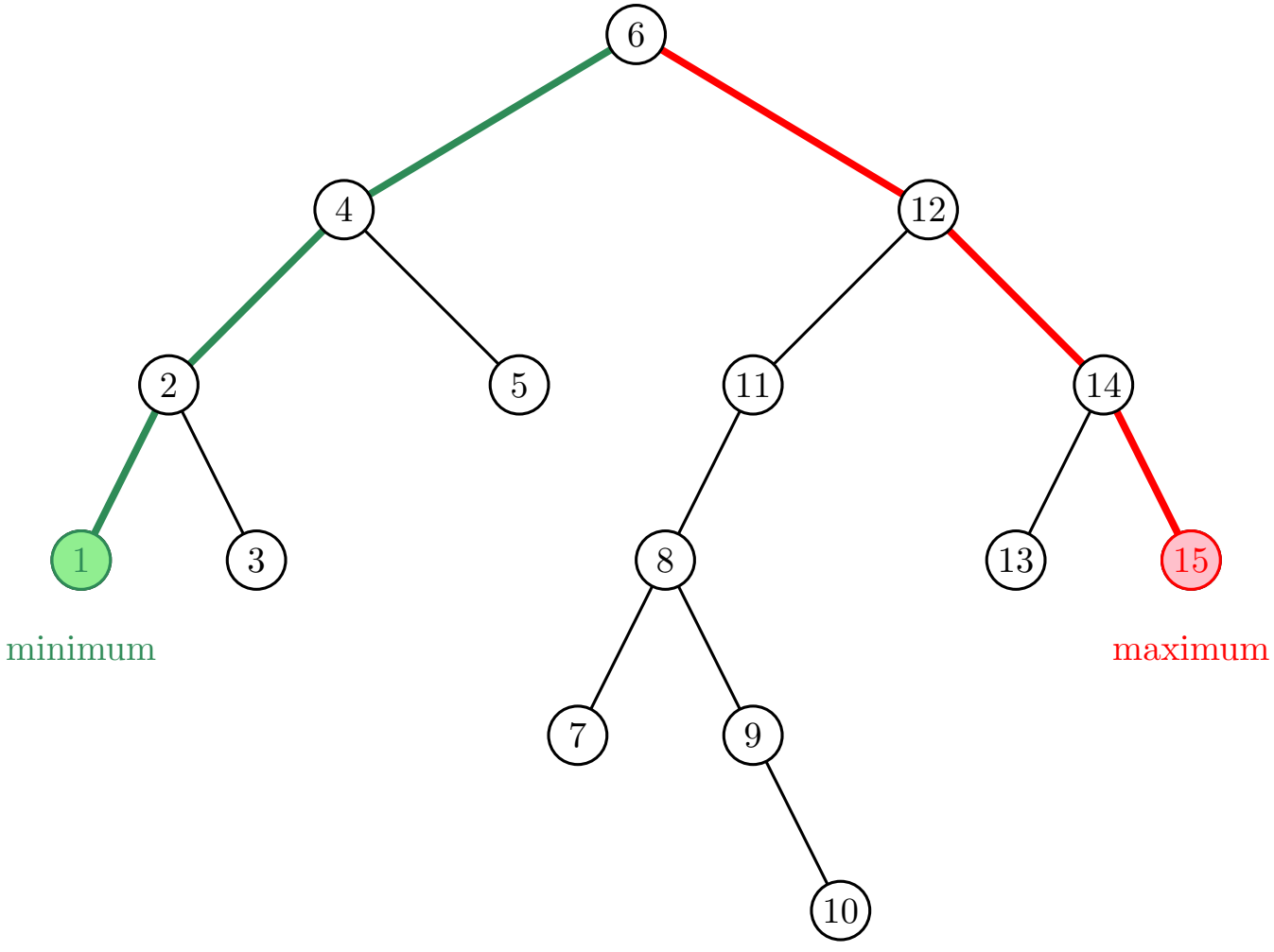
- les étiquettes du sous-arbre gauche (plus petites) et
- celles du sous-arbre droit (plus grandes)



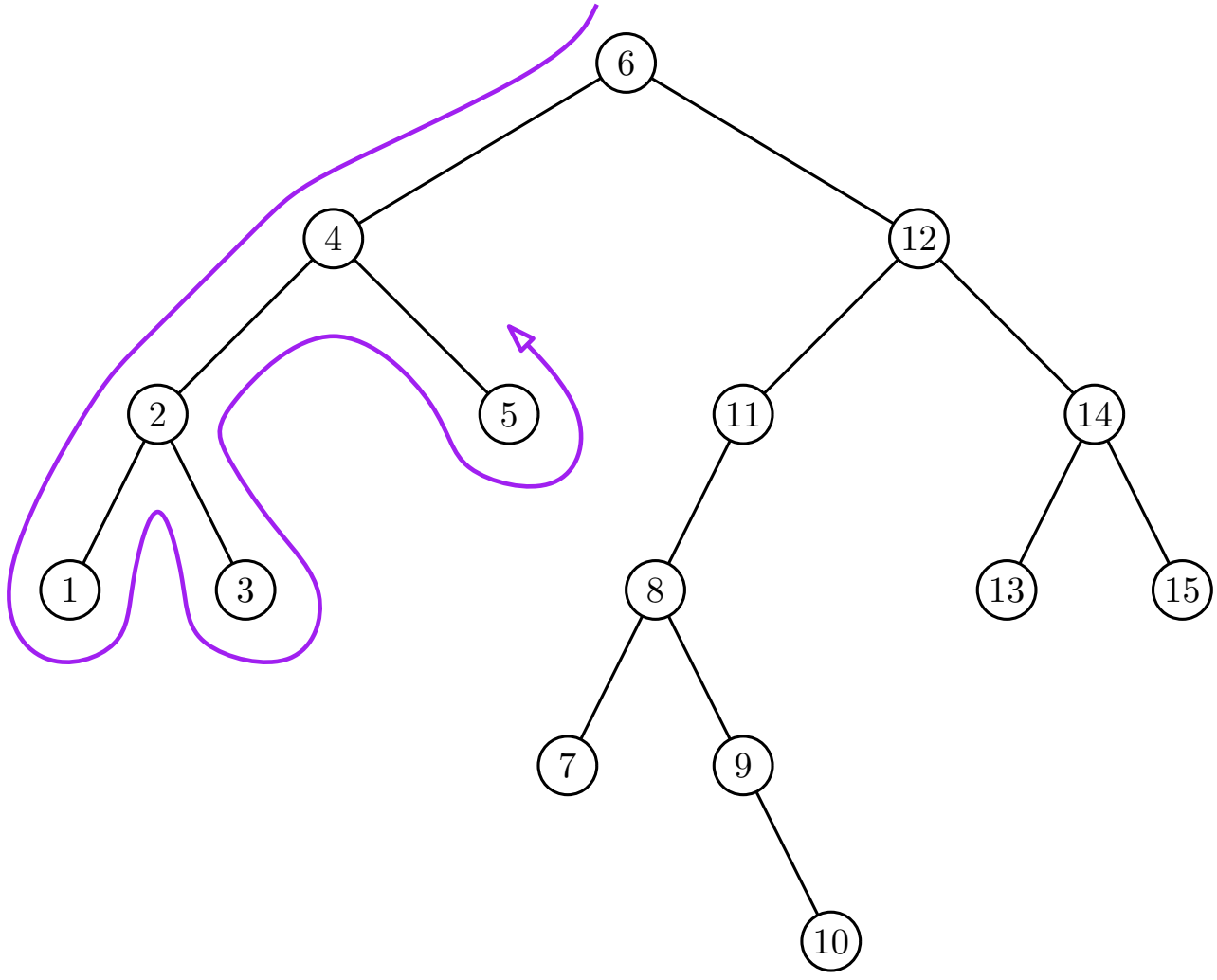
Arbre binaire de recherche



Arbre binaire de recherche



Arbre binaire de recherche



Ordre dans un ABR

```
def liste_triee(noeud) :
  res = []
  if noeud != None :
    res = liste_triee(gauche(noeud))
    res += [ etiquette(noeud) ]
    res += liste_triee(droit(noeud))
  return res
```

Ordre dans un ABR

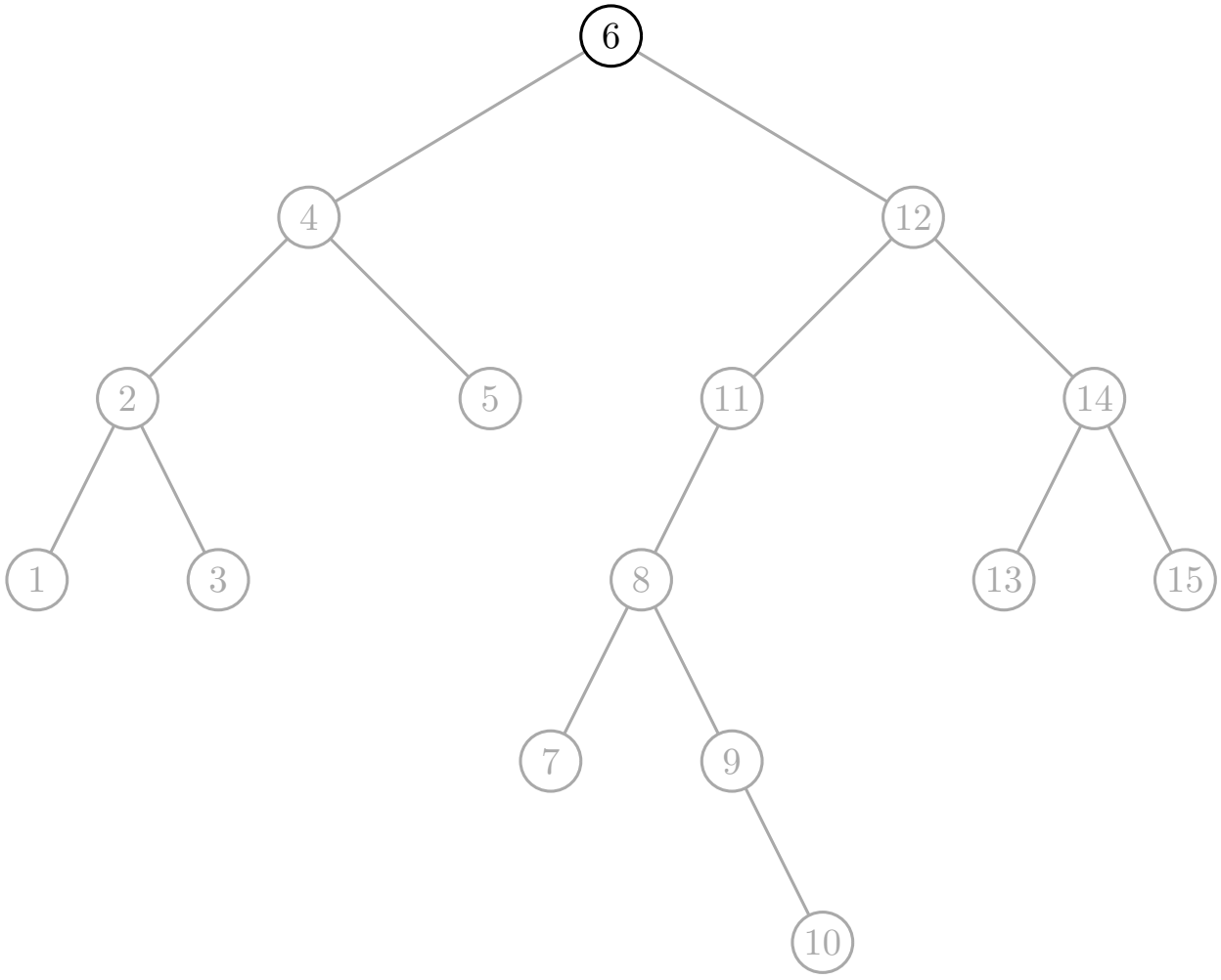
```
def liste_triee(noeud) :
  res = []
  if noeud != None :
    res = liste_triee(gauche(noeud))
    res += [ etiquette(noeud) ]
    res += liste_triee(droit(noeud))
  return res
```

Théorème

le parcours infixe d'un ABR à n nœuds produit la liste triée de ses éléments en temps $\Theta(n)$.

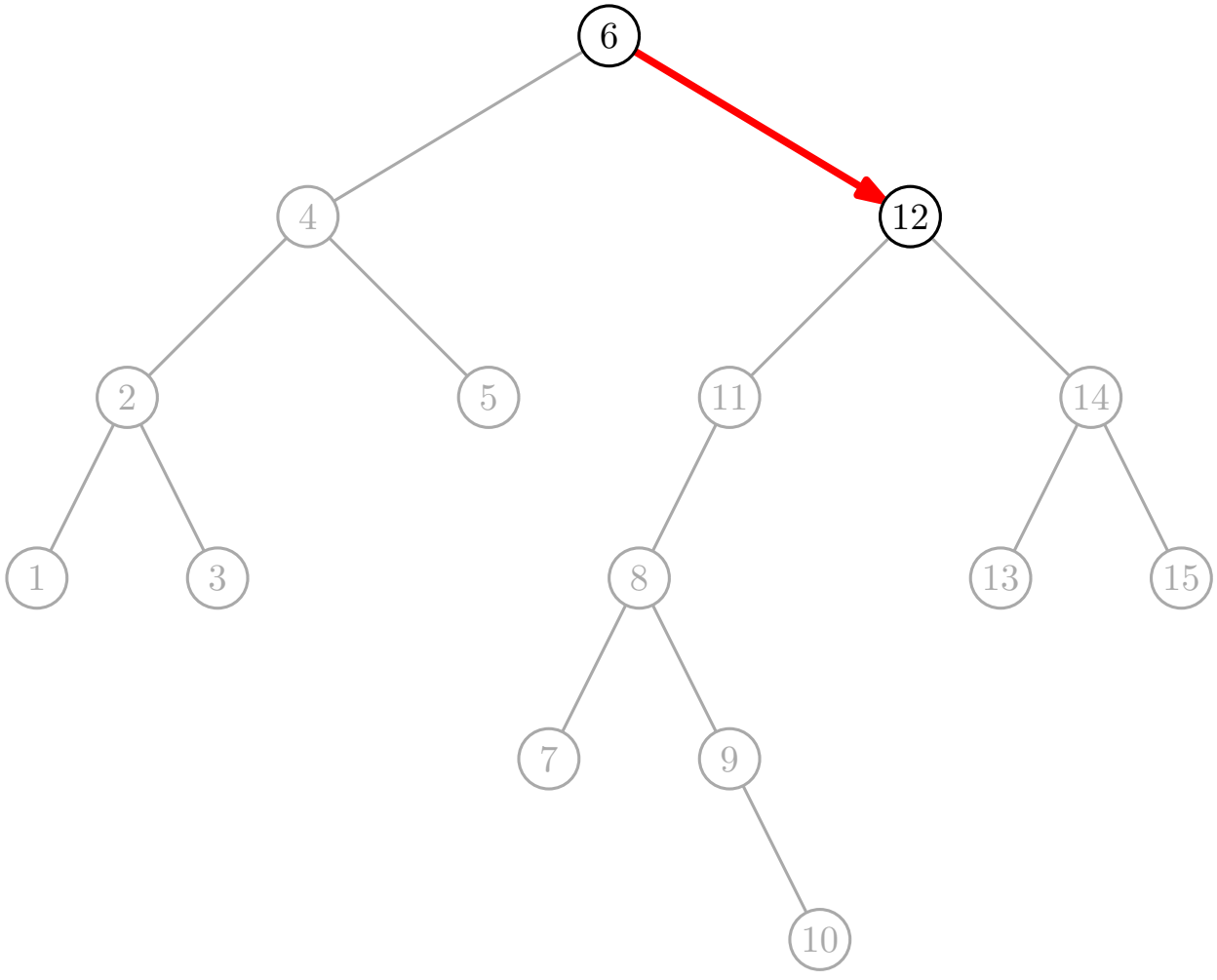
Recherche dans un ABR

Exemple – recherche de 9 :



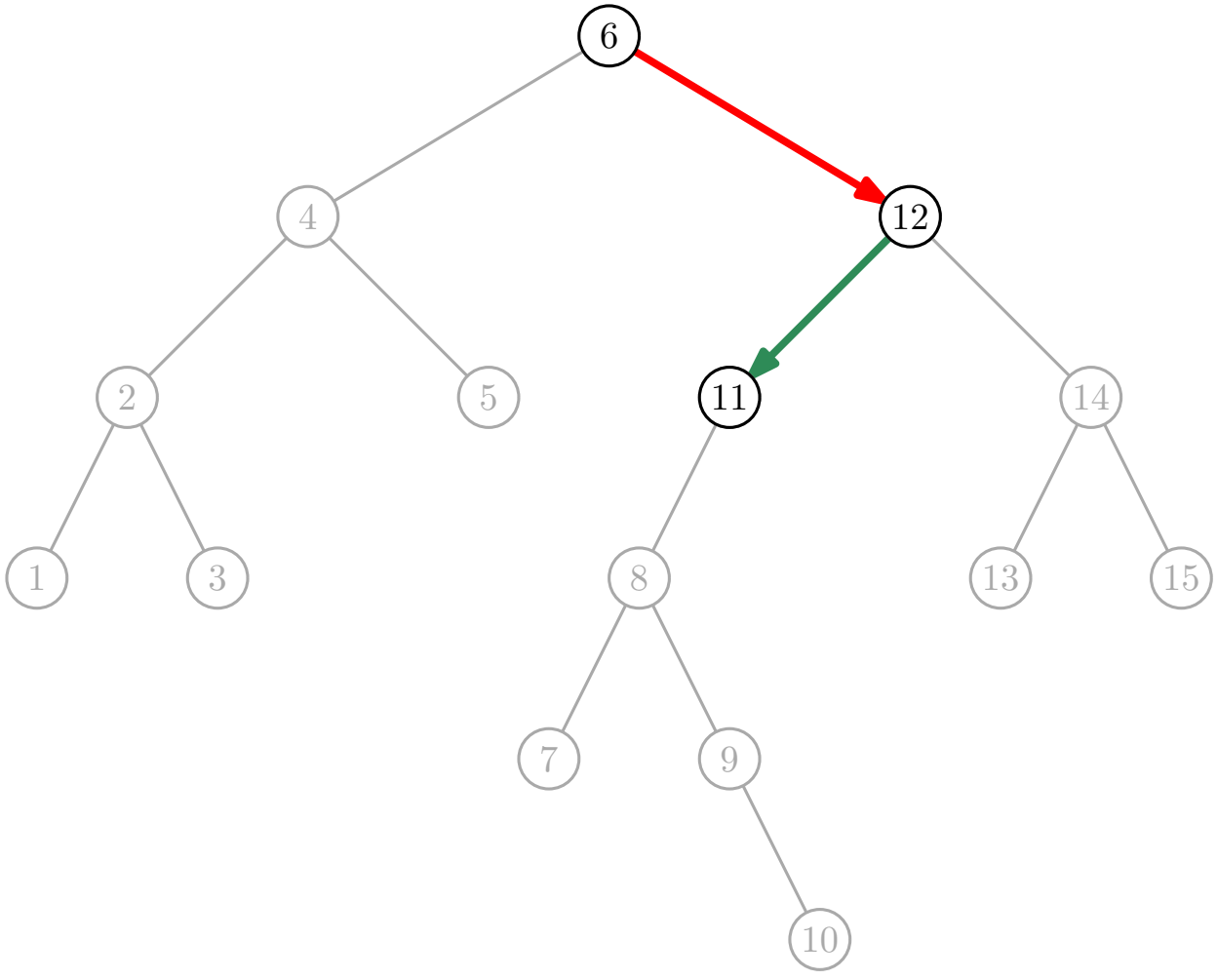
Recherche dans un ABR

Exemple – recherche de 9 :



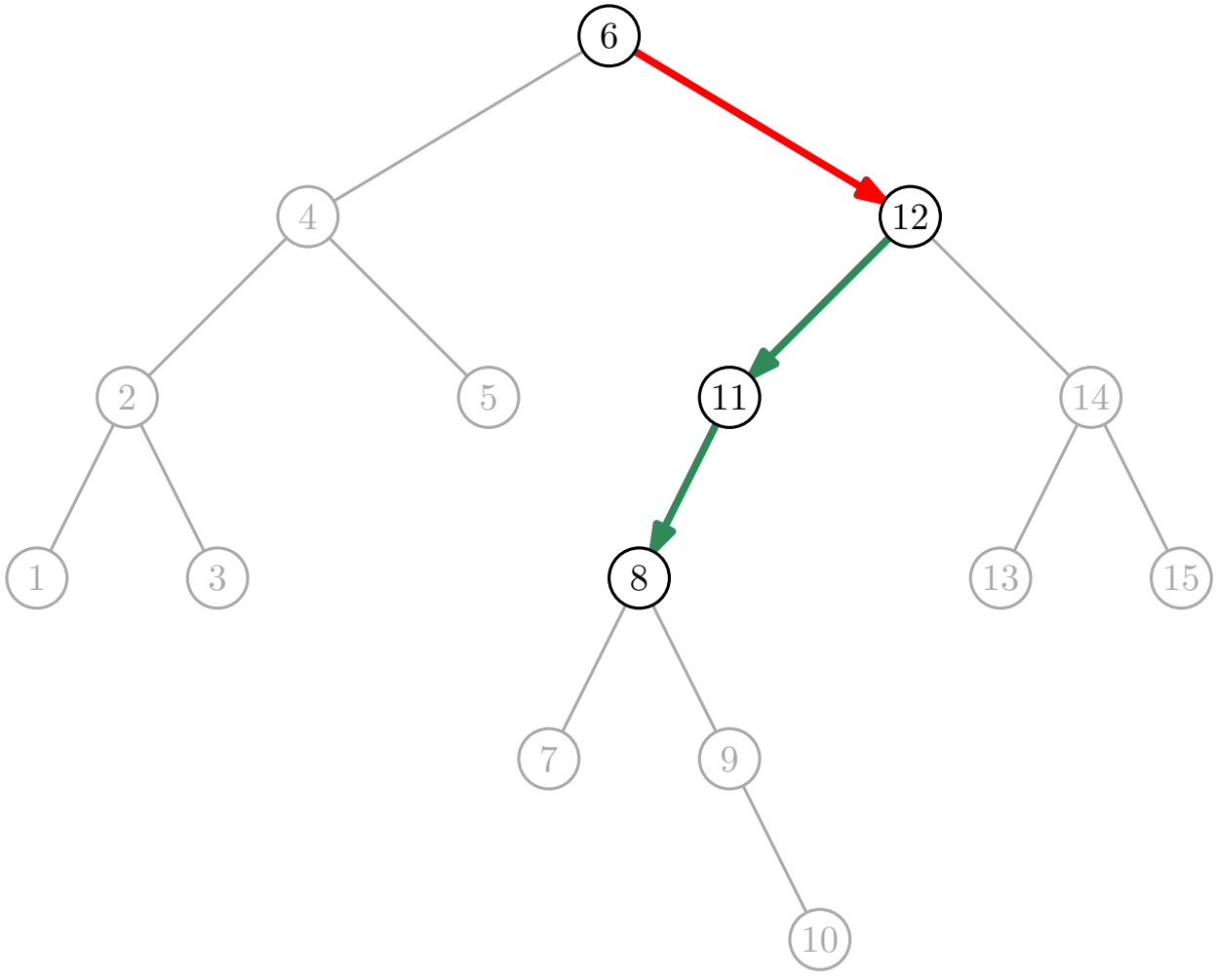
Recherche dans un ABR

Exemple – recherche de 9 :



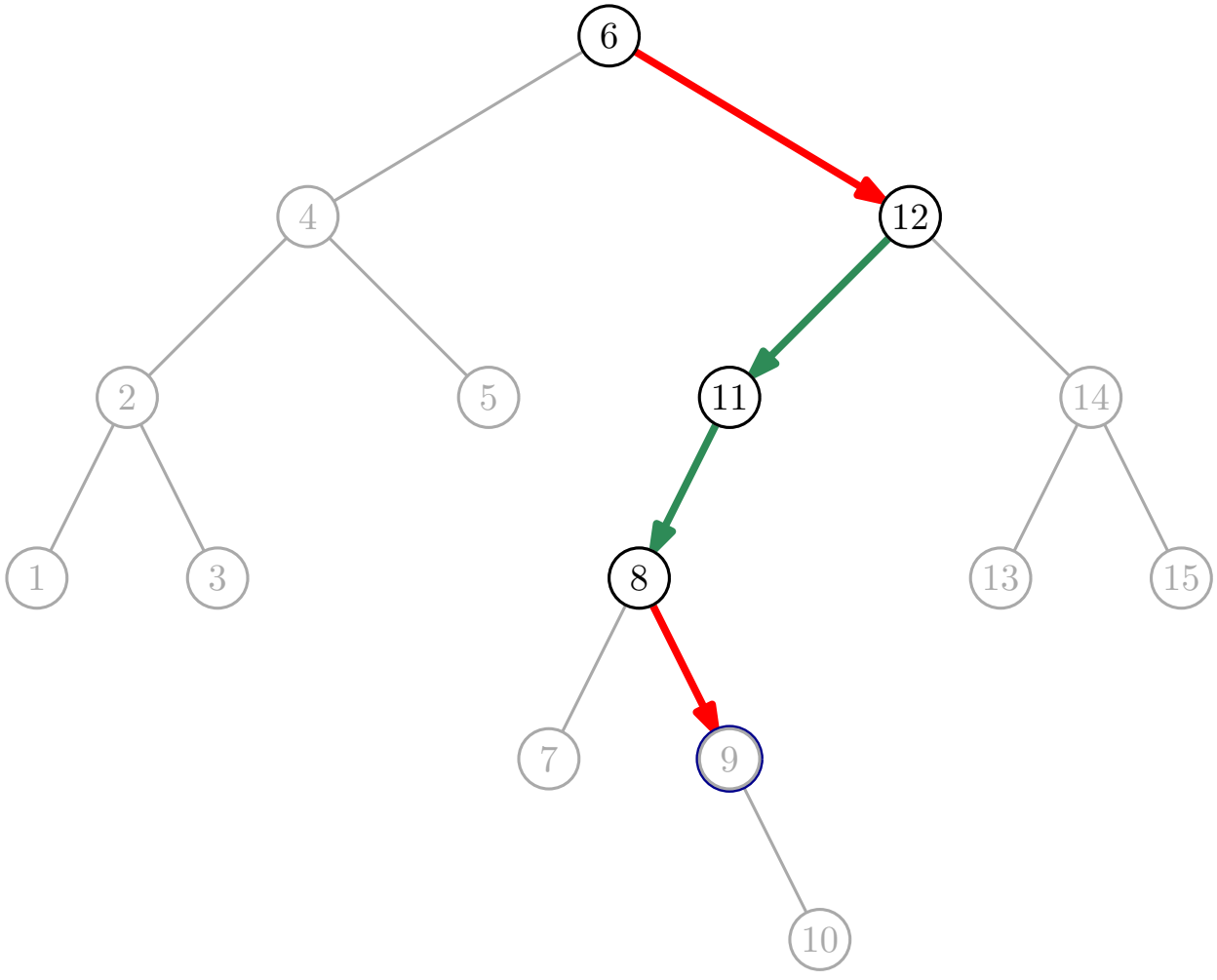
Recherche dans un ABR

Exemple – recherche de 9 :



Recherche dans un ABR

Exemple – recherche de 9 :



Recherche dans un ABR

```
def recherche(noeud, x) : # version récursive
  if noeud == None : return None
  if etiquette(noeud) == x : return noeud
  if etiquette(noeud) > x : return recherche(gauche(noeud))
  return recherche(droit(noeud))
```

Recherche dans un ABR

```
def recherche(noeud, x) : # version récursive
  if noeud == None : return None
  if etiquette(noeud) == x : return noeud
  if etiquette(noeud) > x : return recherche(gauche(noeud))
  return recherche(droit(noeud))
```

Théorème

`recherche(r, x)` effectue la recherche d'un élément `x` dans l'ABR de racine `r` en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

Cas particuliers : minimum/maximum

```
def minimum(noeud) : # version récursive
  if gauche(noeud) == None : return noeud
  return minimum(gauche(noeud))
```

Cas particuliers : minimum/maximum

```
def minimum(noeud) : # version récursive
  if gauche(noeud) == None : return noeud
  return minimum(gauche(noeud))

def minimum(noeud) : # version itérative
  while gauche(noeud) != None :
    noeud = gauche(noeud)
  return noeud
```

Cas particuliers : minimum/maximum

```
def minimum(noeud) : # version récursive
  if gauche(noeud) == None : return noeud
  return minimum(gauche(noeud))

def minimum(noeud) : # version itérative
  while gauche(noeud) != None :
    noeud = gauche(noeud)
  return noeud
```

Théorème

minimum(r) détermine le plus petit élément dans l'ABR de racine r en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

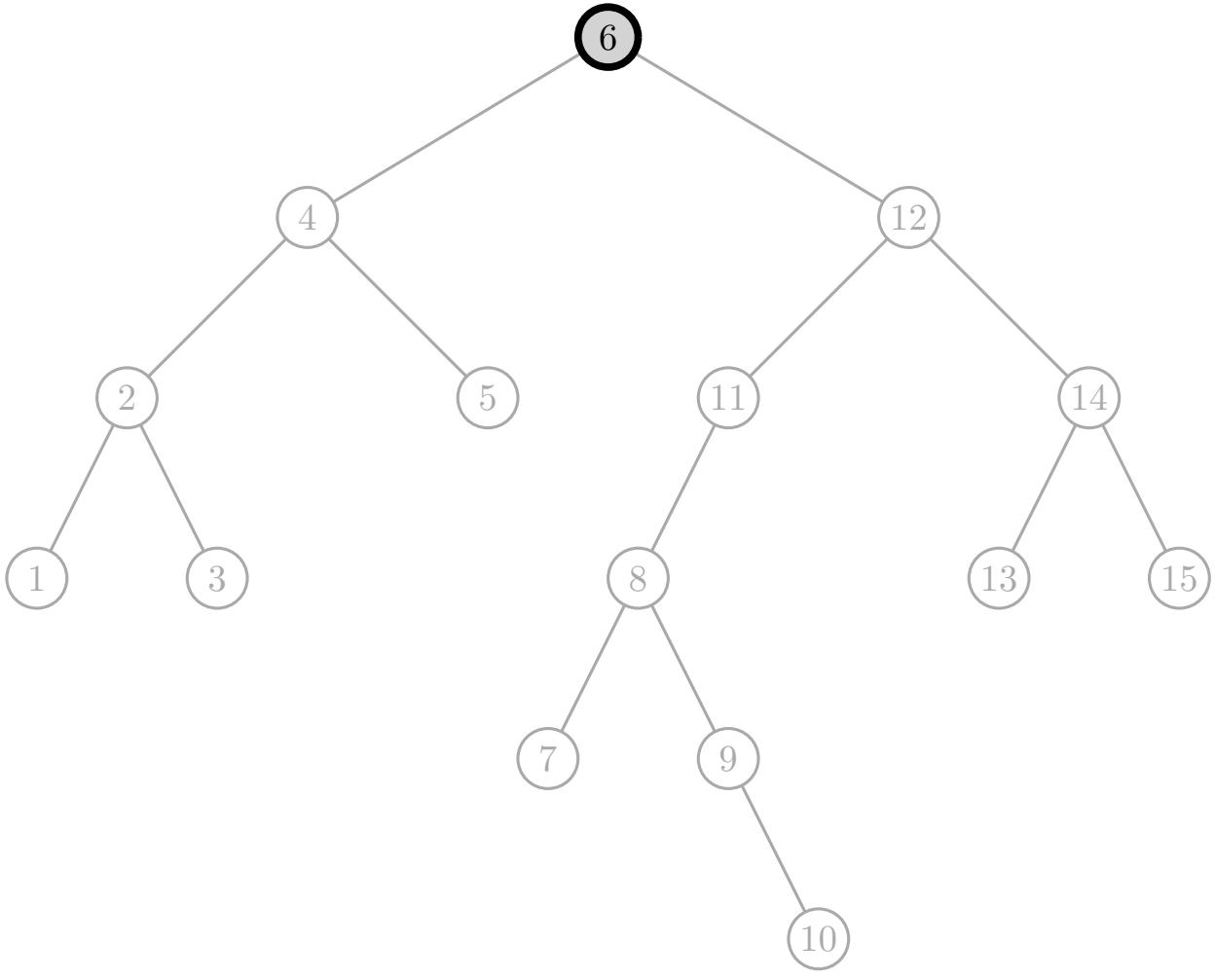
Successeur d'un élément

successeur(n)

étant donné un nœud `n` d'un ABR, d'étiquette `e`, déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à `e`.

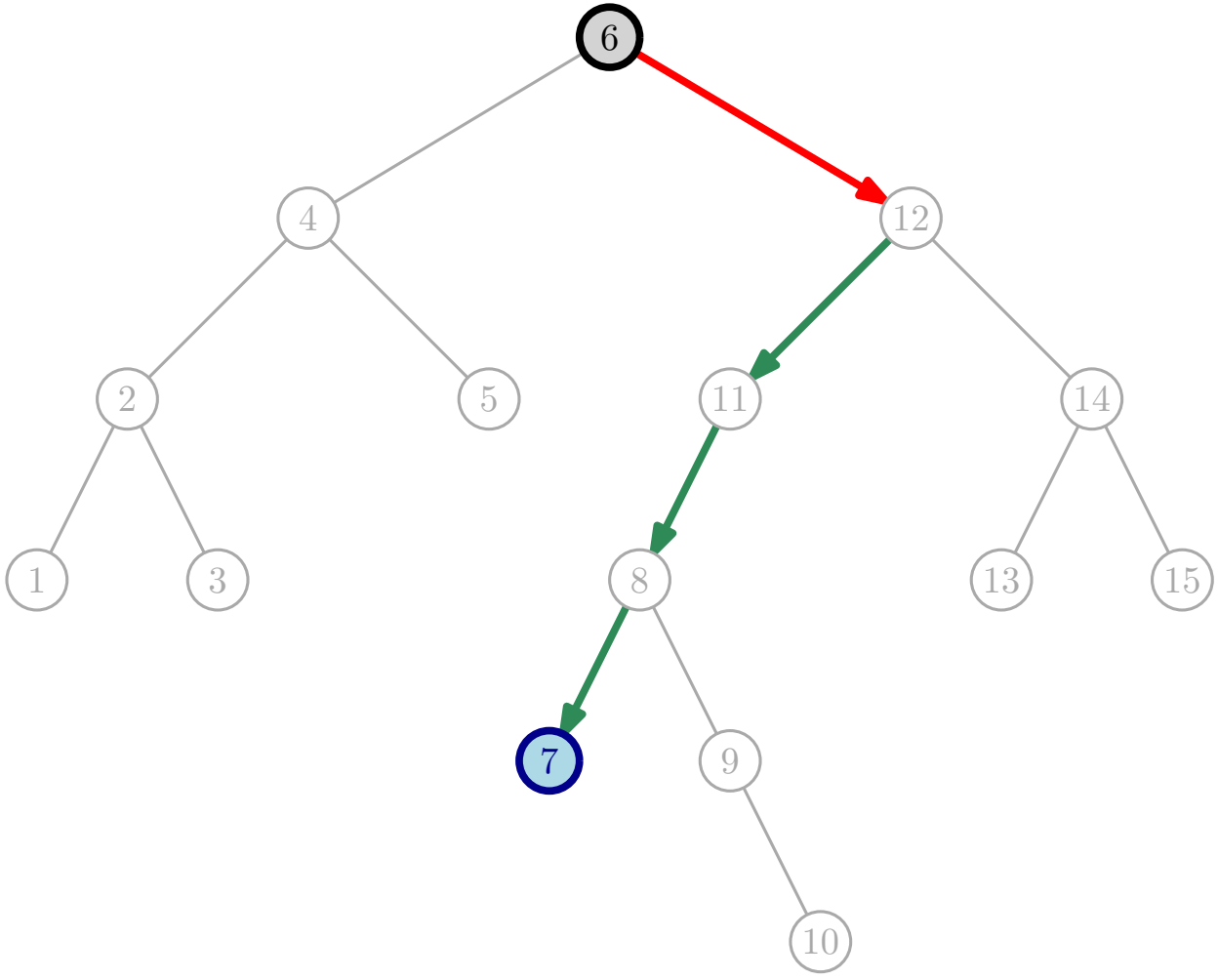
Successeur d'un élément

si le nœud a un fils droit



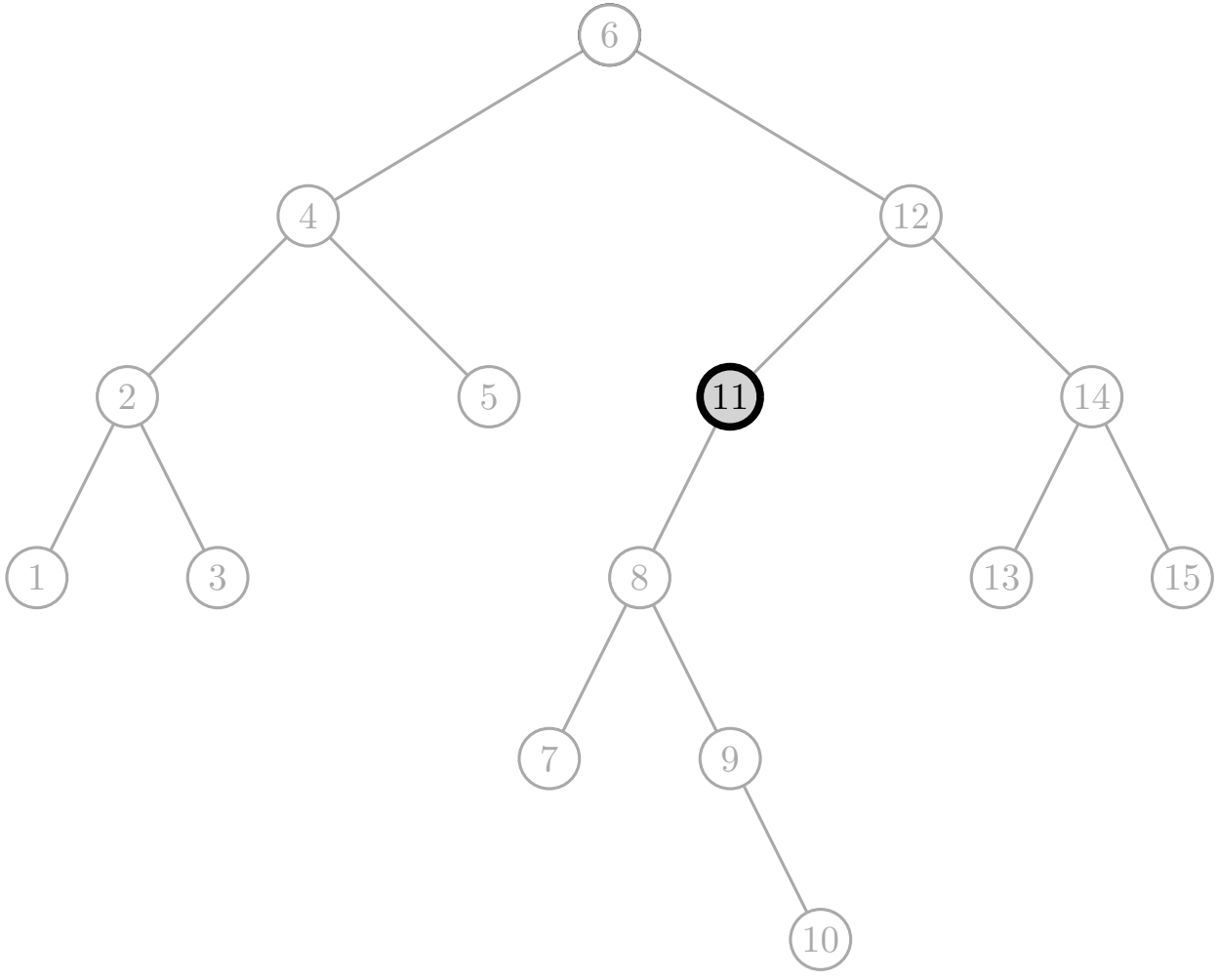
Successeur d'un élément

si le nœud a un fils droit



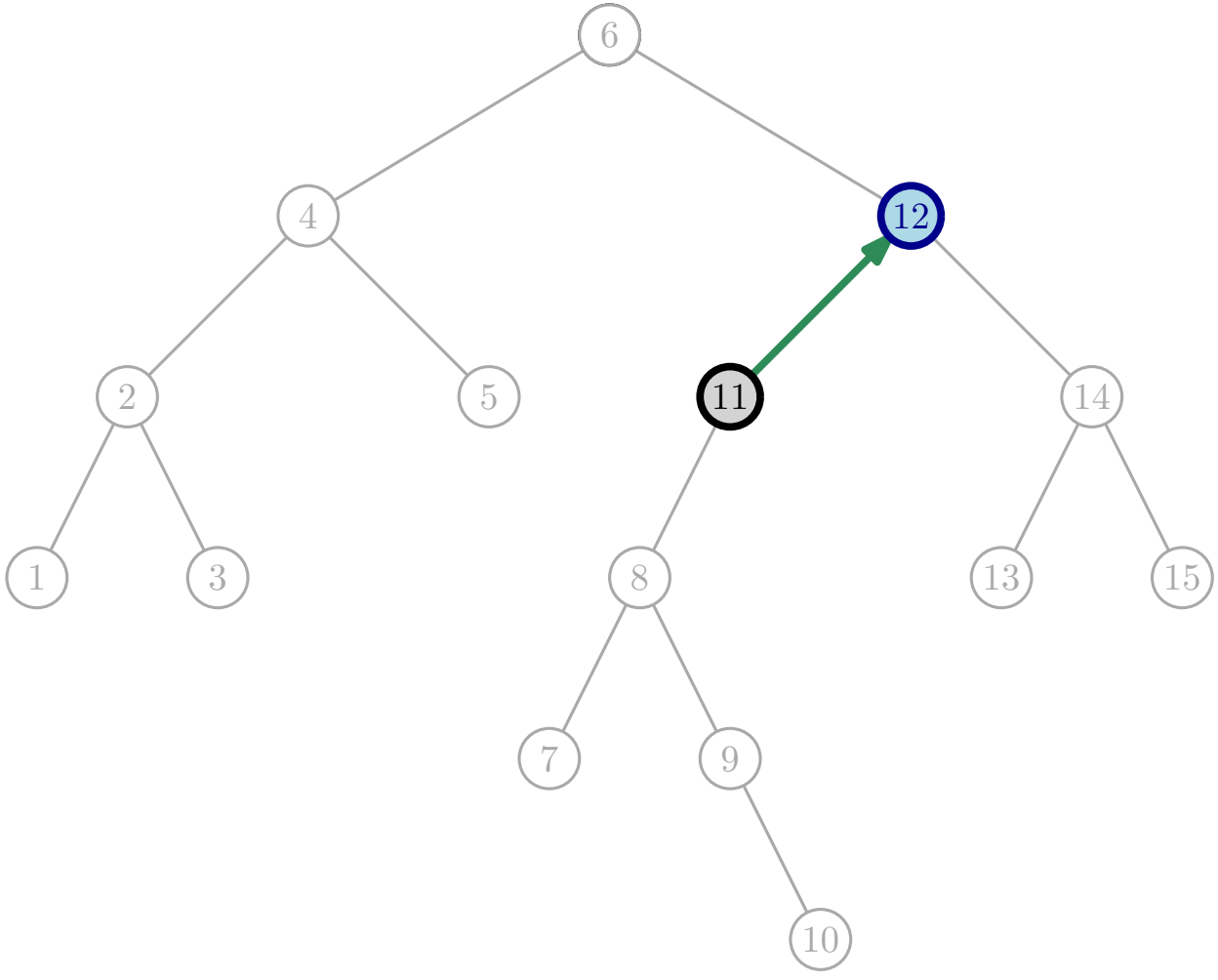
Successeur d'un élément

si le nœud n'a pas de fils droit



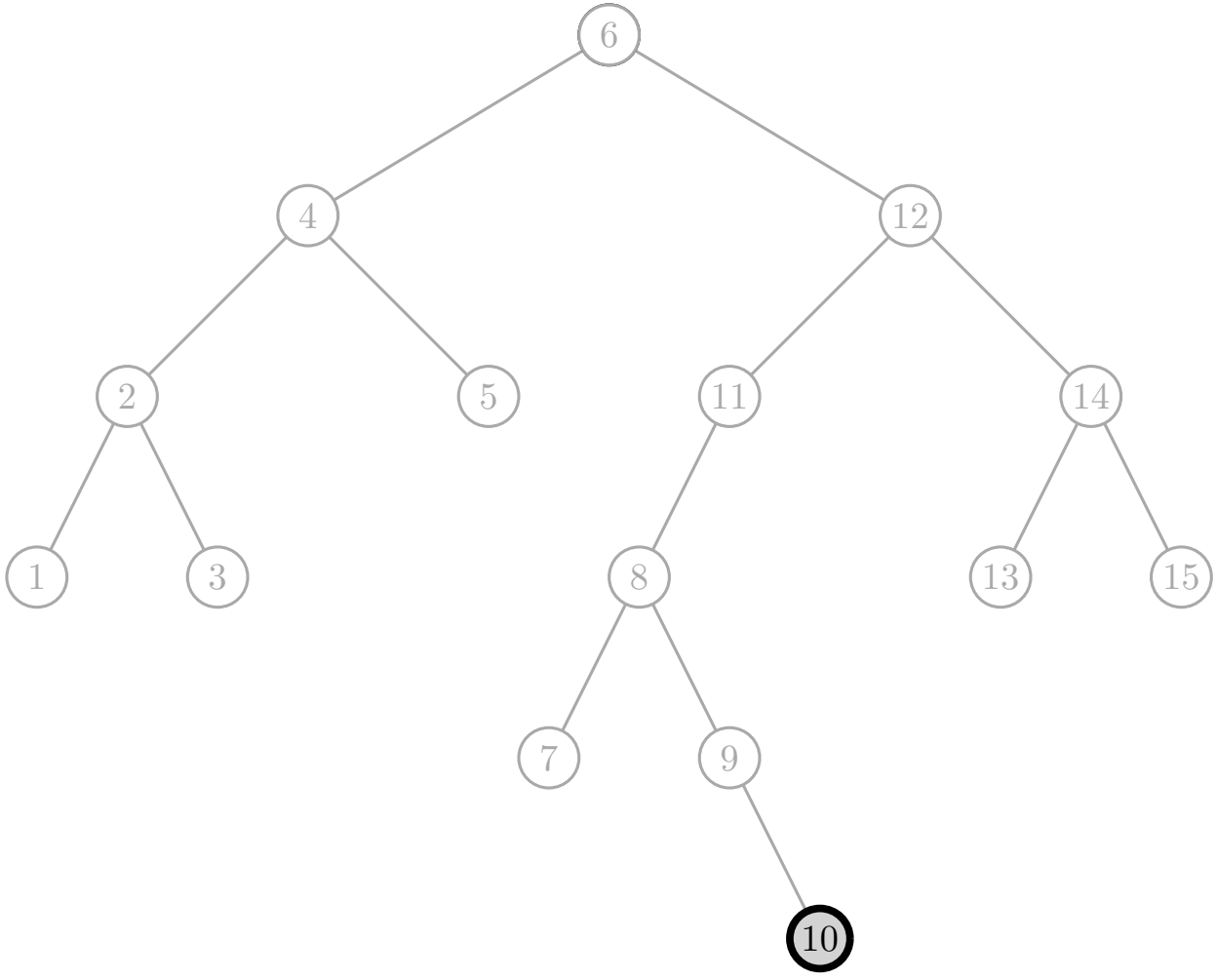
Successeur d'un élément

si le nœud n'a pas de fils droit



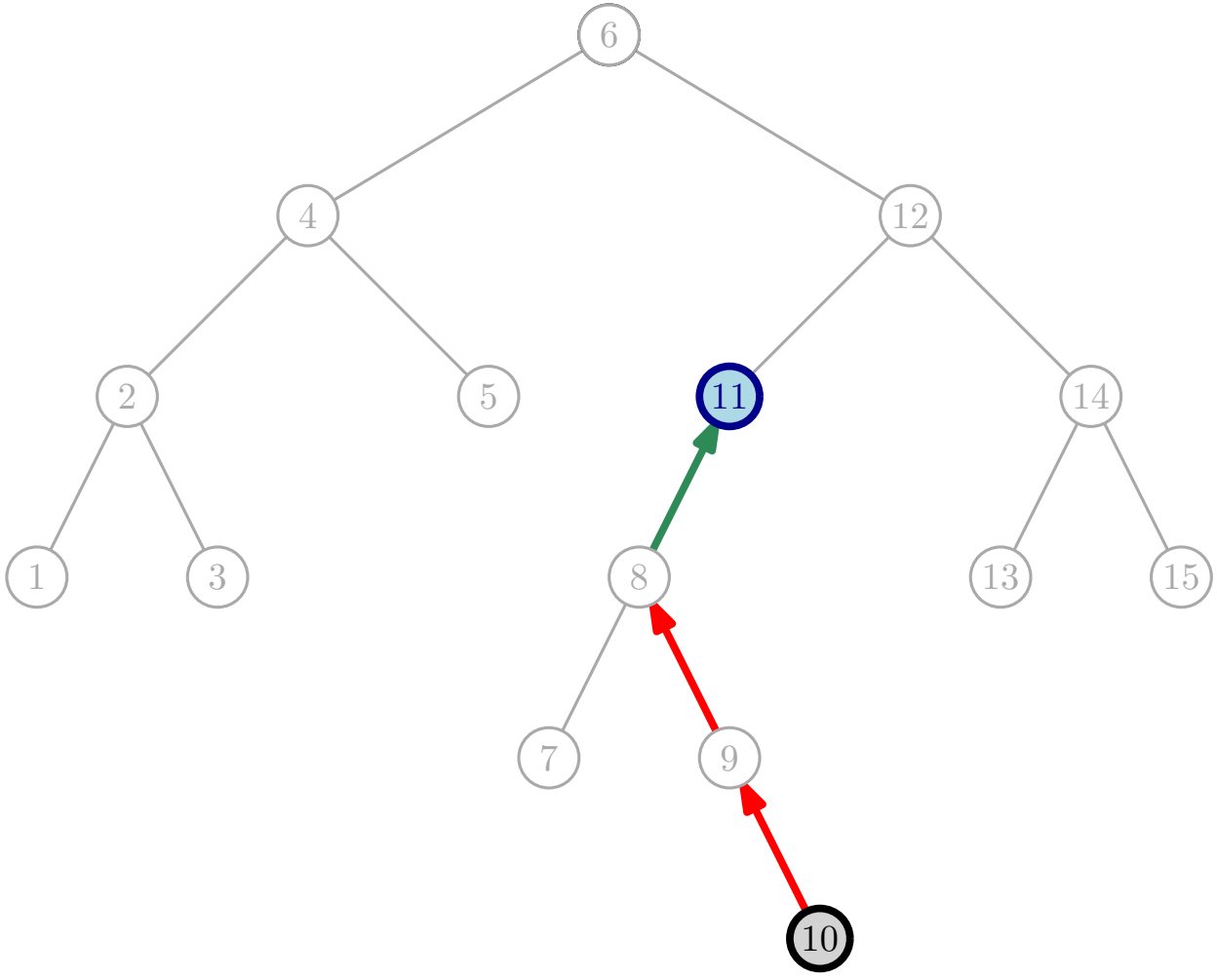
Successeur d'un élément

si le nœud n'a pas de fils droit



Successeur d'un élément

si le nœud n'a pas de fils droit



Successeur d'un élément

```
def successeur(noeud) :
  if droit(noeud) != None :
    return minimum(droit(noeud))
  while pere(noeud) != None and est_fils_droit(noeud) :
    noeud = pere(noeud)
  return pere(noeud)
```

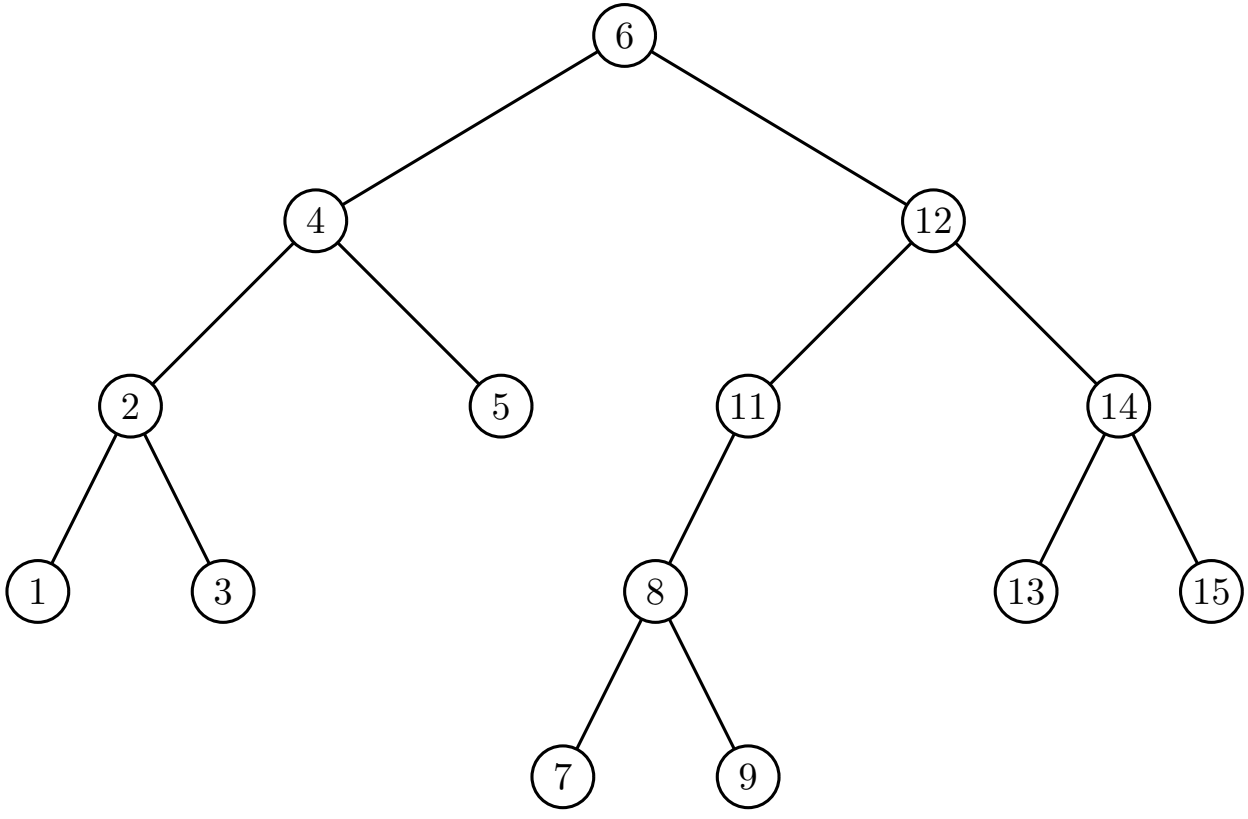
Successeur d'un élément

```
def successeur(noeud) :
  if droit(noeud) != None :
    return minimum(droit(noeud))
  while pere(noeud) != None and est_fils_droit(noeud) :
    noeud = pere(noeud)
  return pere(noeud)
```

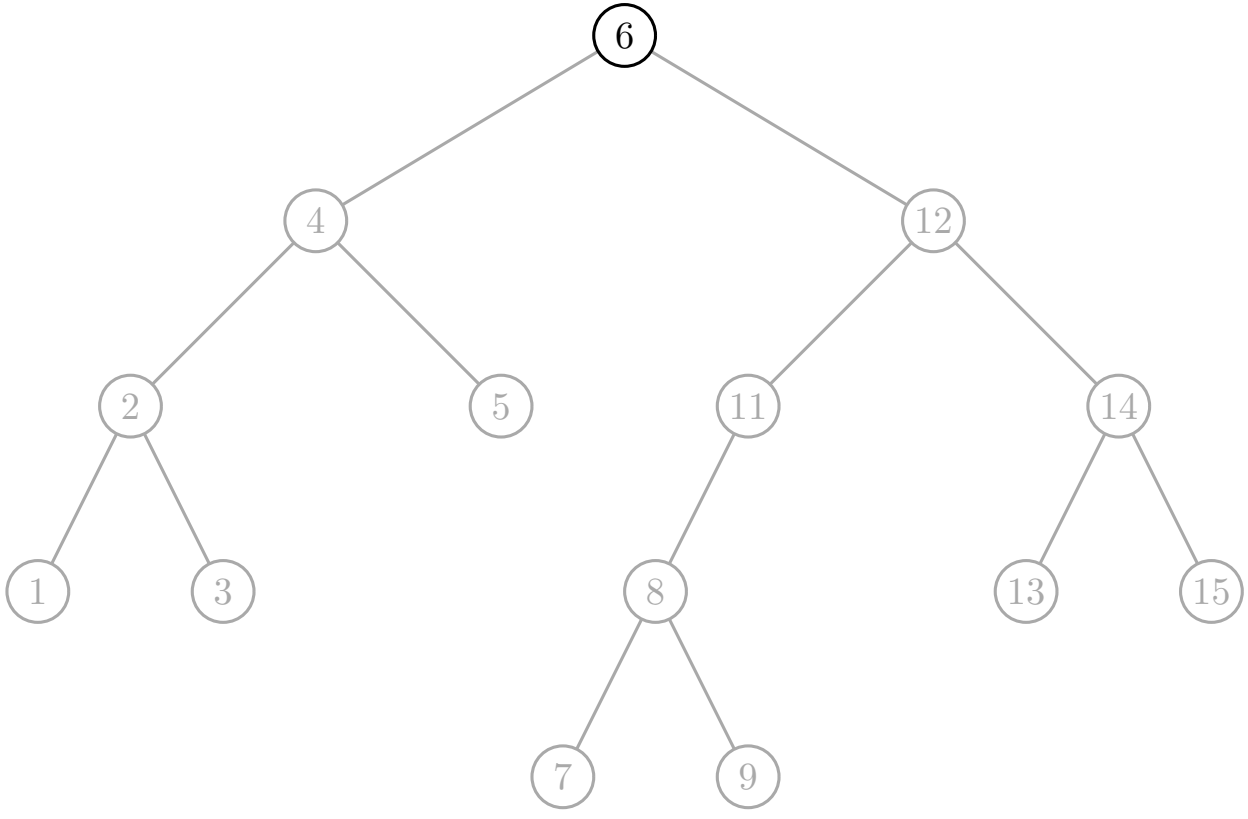
Théorème

successeur(noeud) détermine le successeur d'un noeud d'un ABR en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

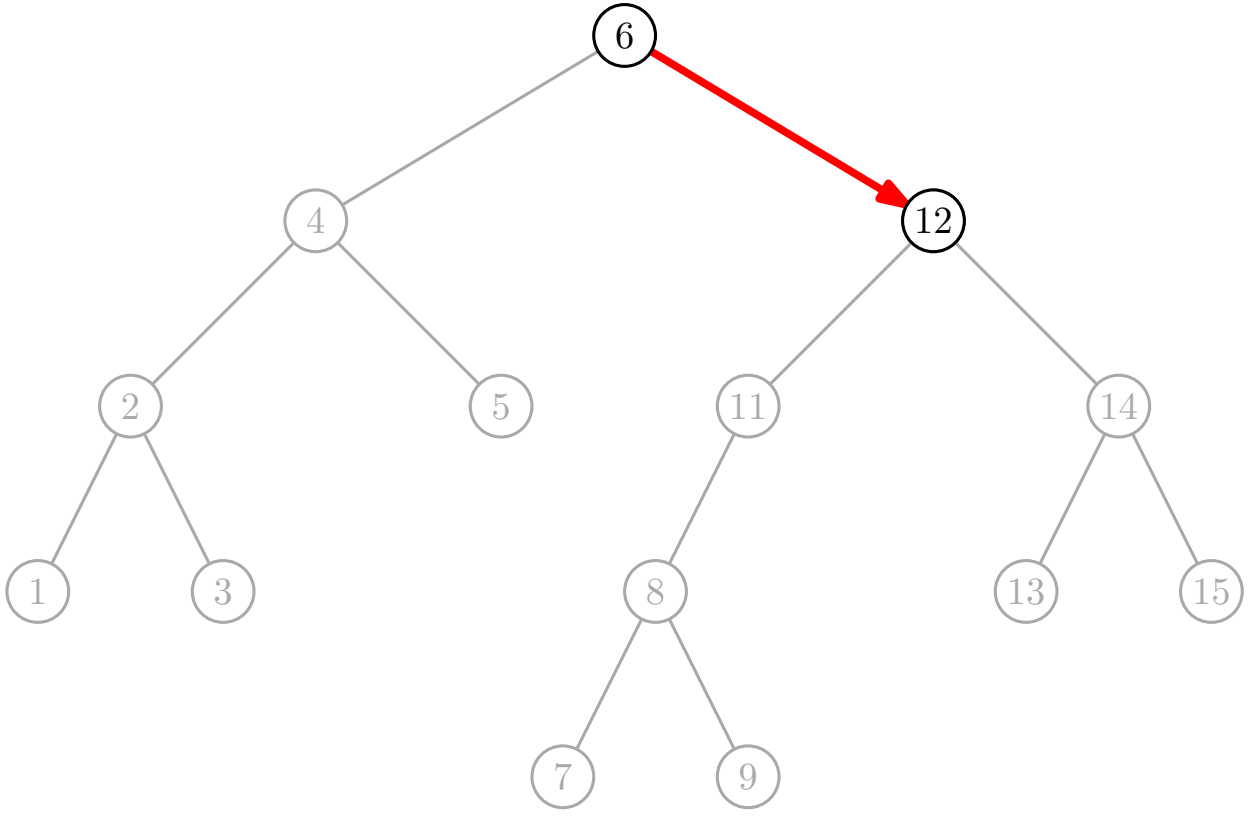
Insertion dans un ABR



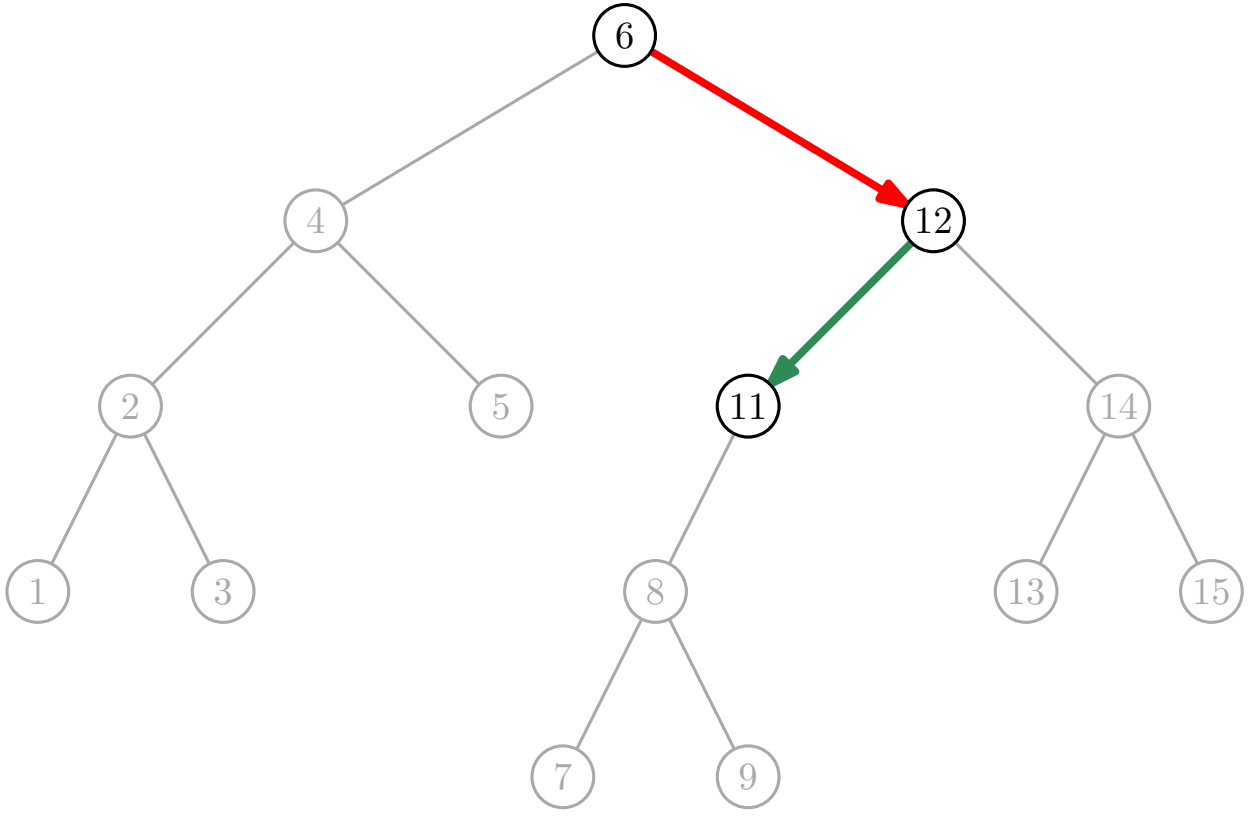
Insertion dans un ABR



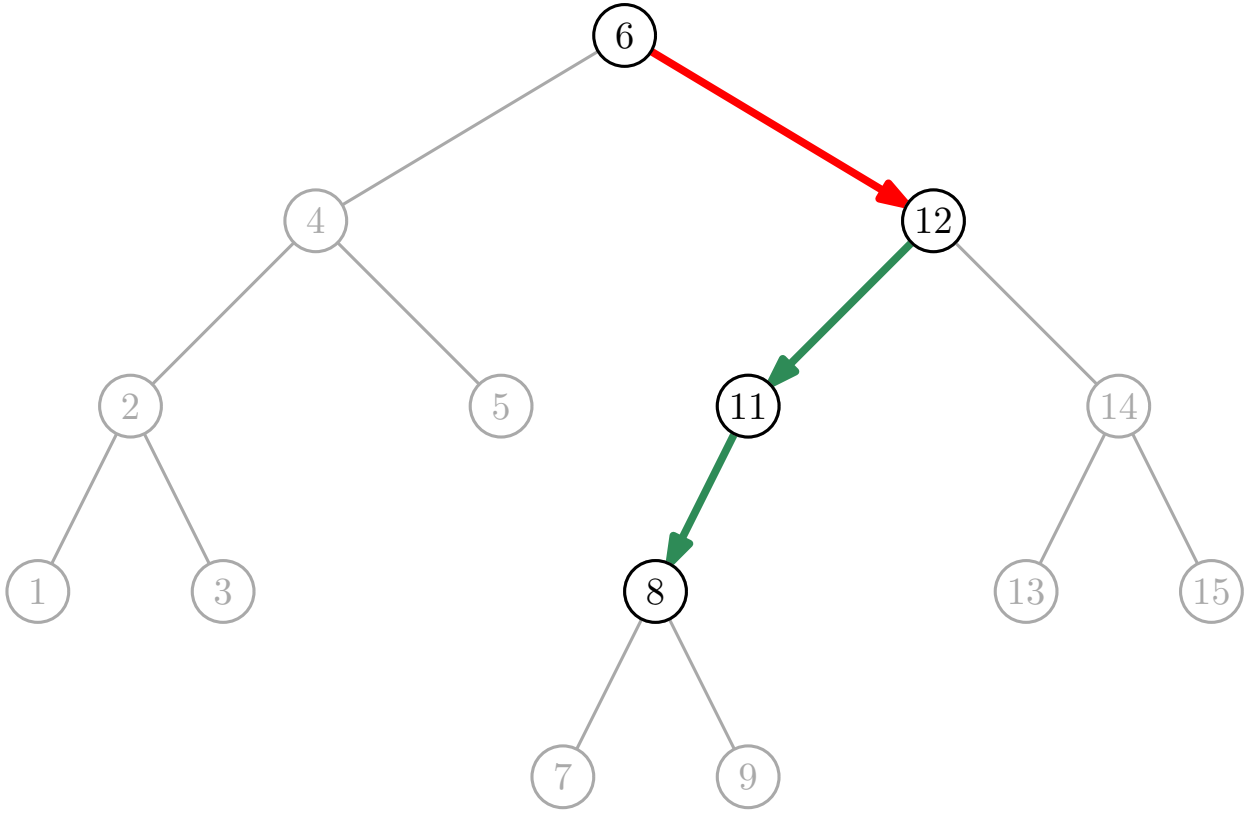
Insertion dans un ABR



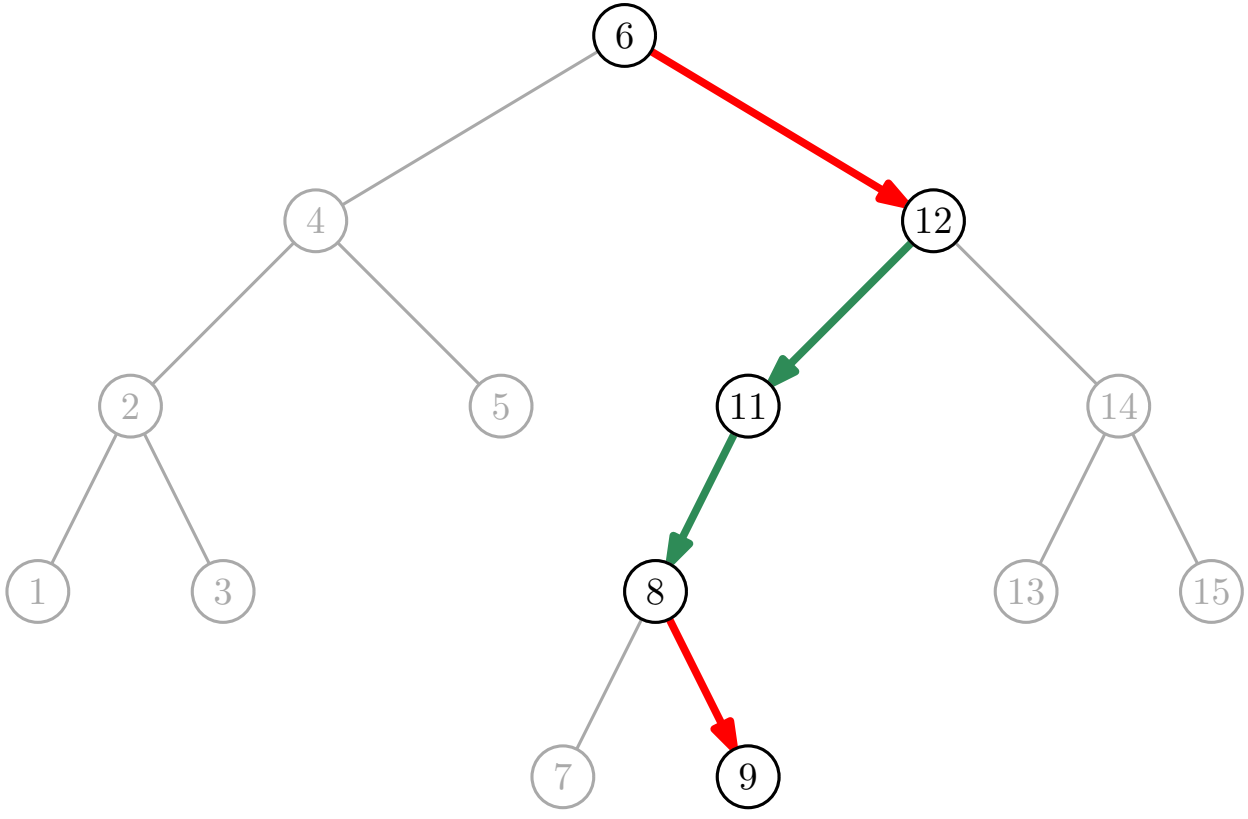
Insertion dans un ABR



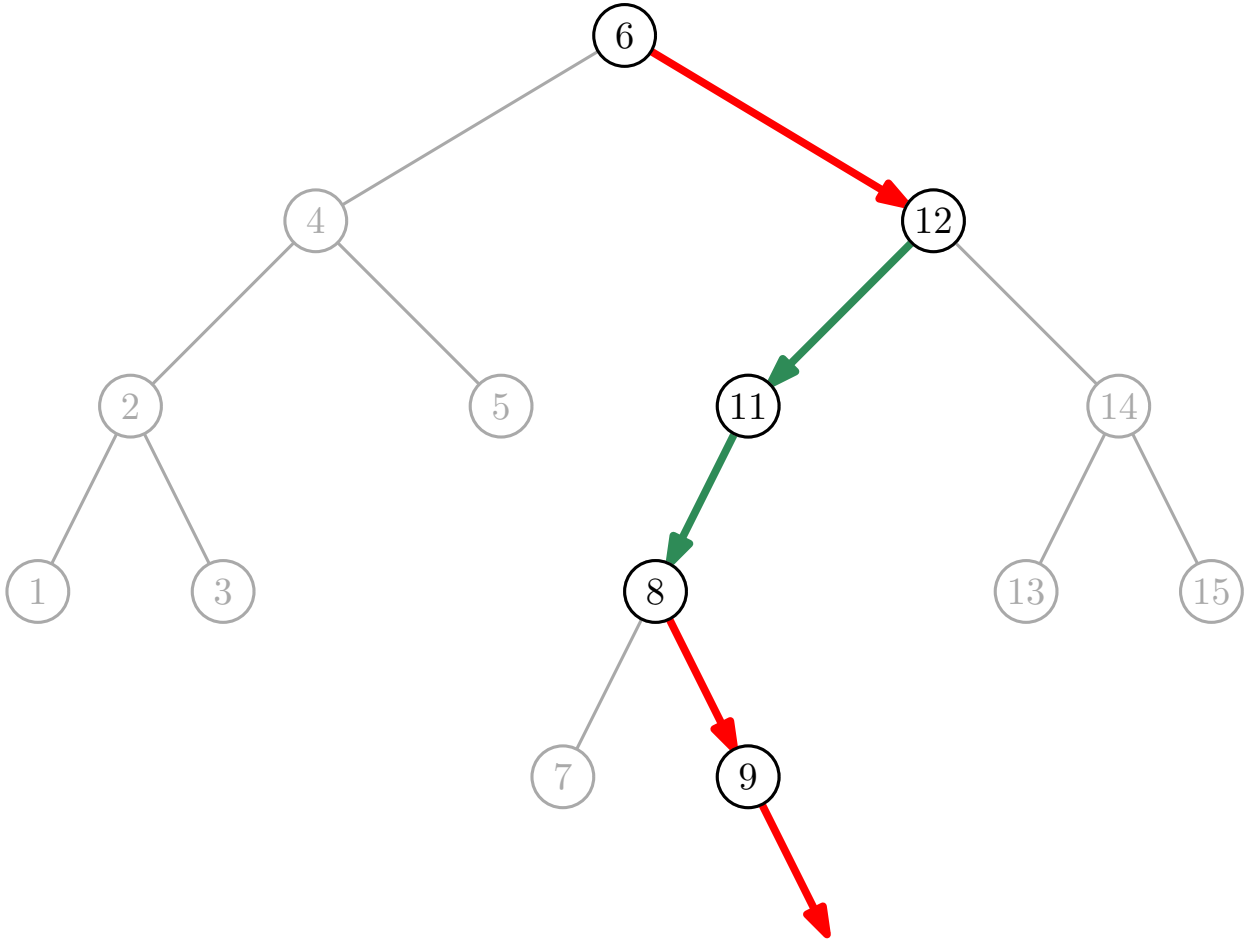
Insertion dans un ABR



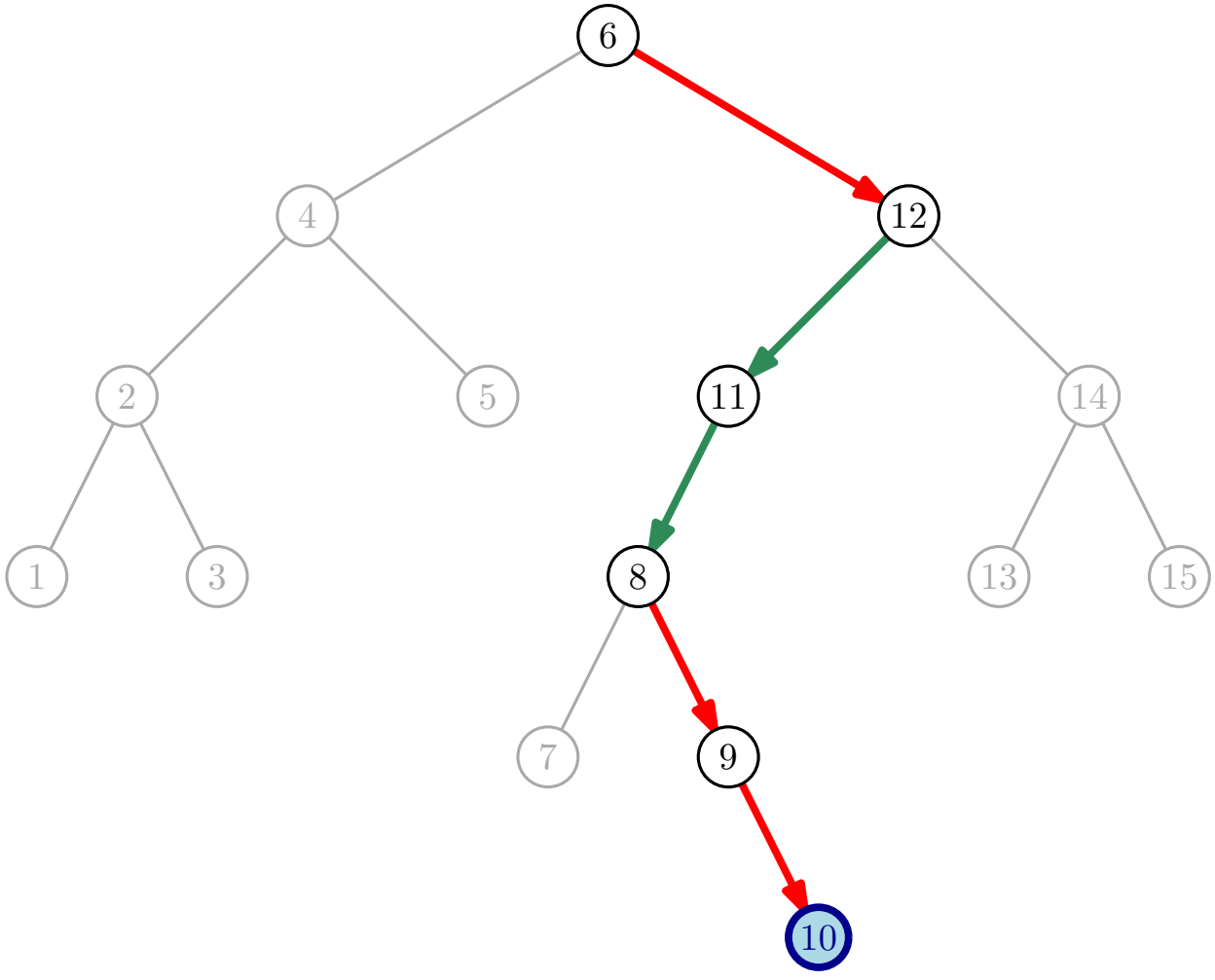
Insertion dans un ABR



Insertion dans un ABR



Insertion dans un ABR

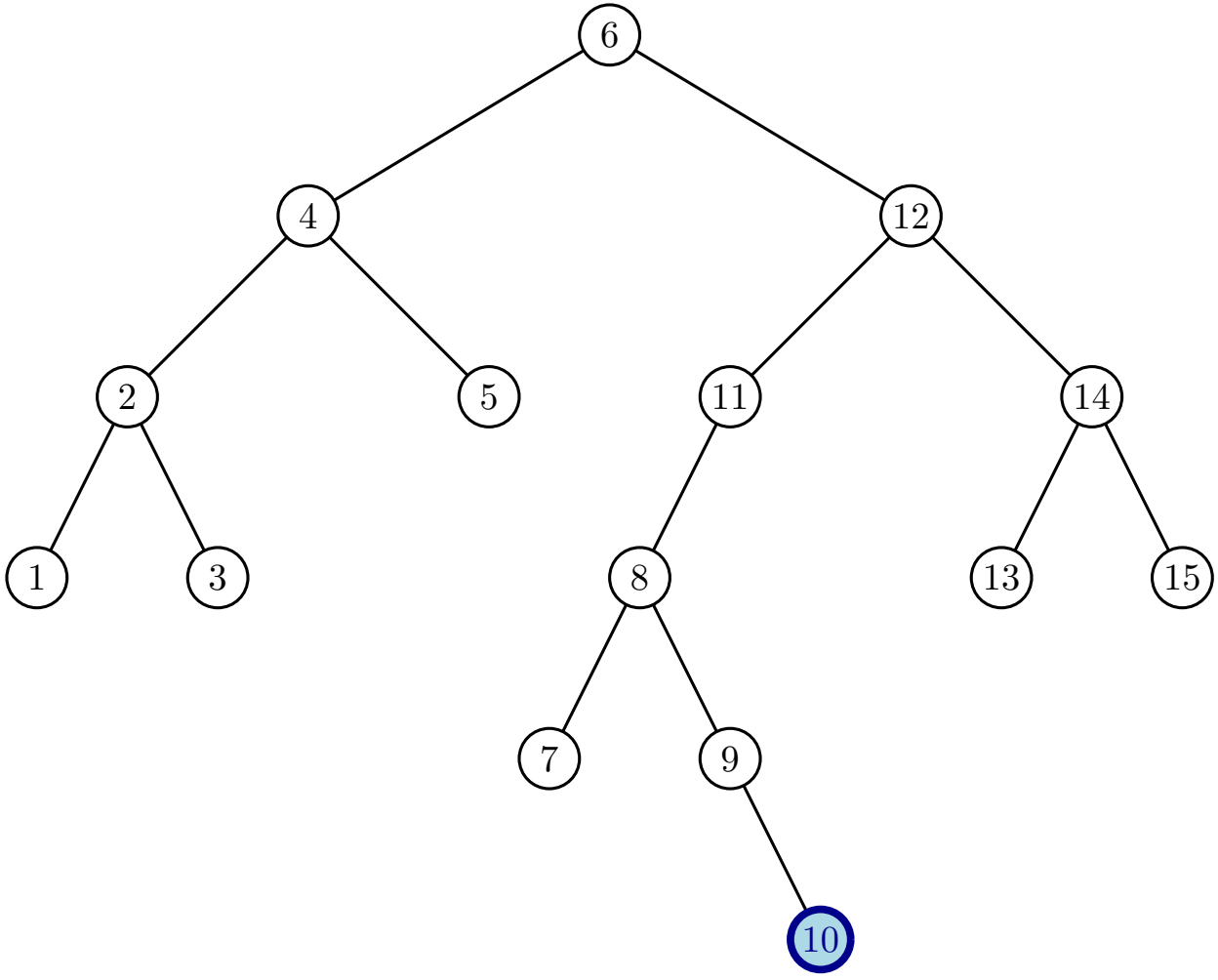


Théorème

L'insertion d'un nouvel élément dans un ABR de hauteur h *peut se faire en temps* $\Theta(h)$ *au pire.*

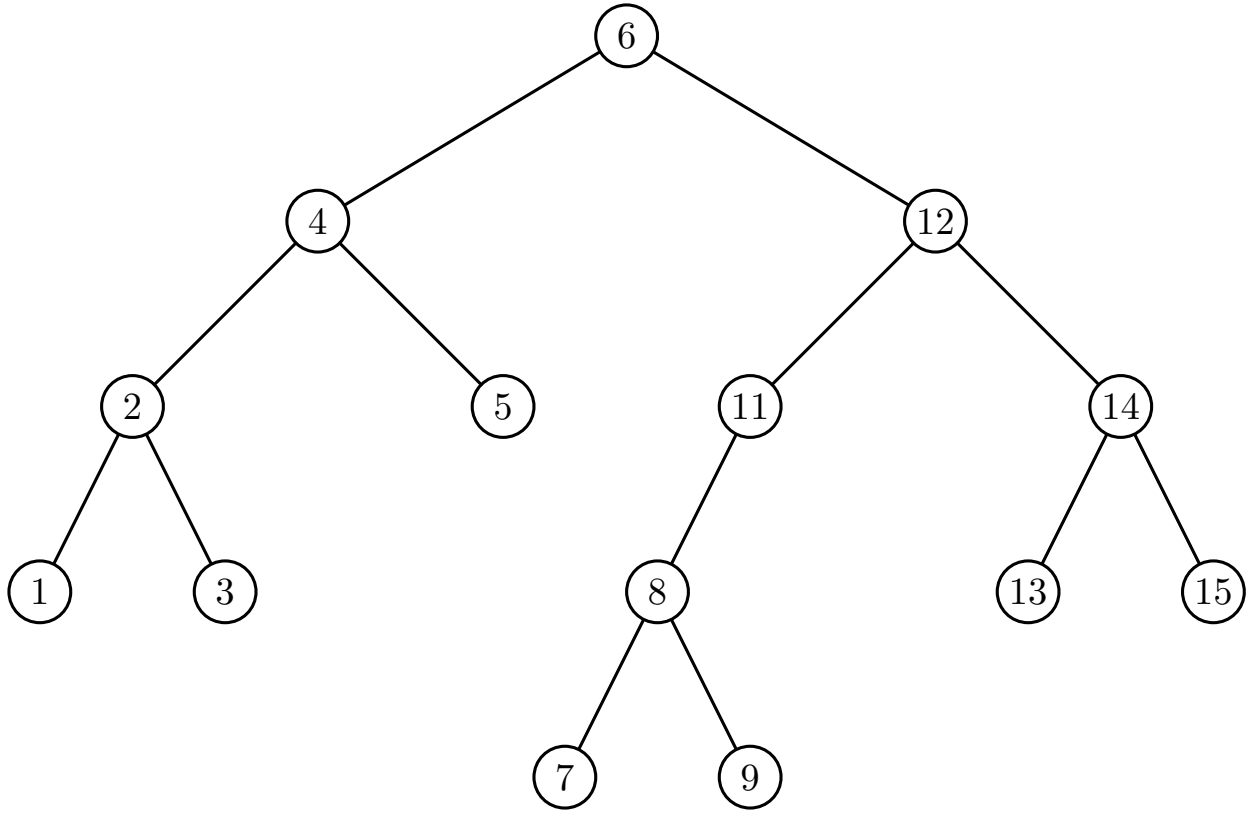
Suppression dans un ABR

si le nœud à supprimer n'a pas d'enfant



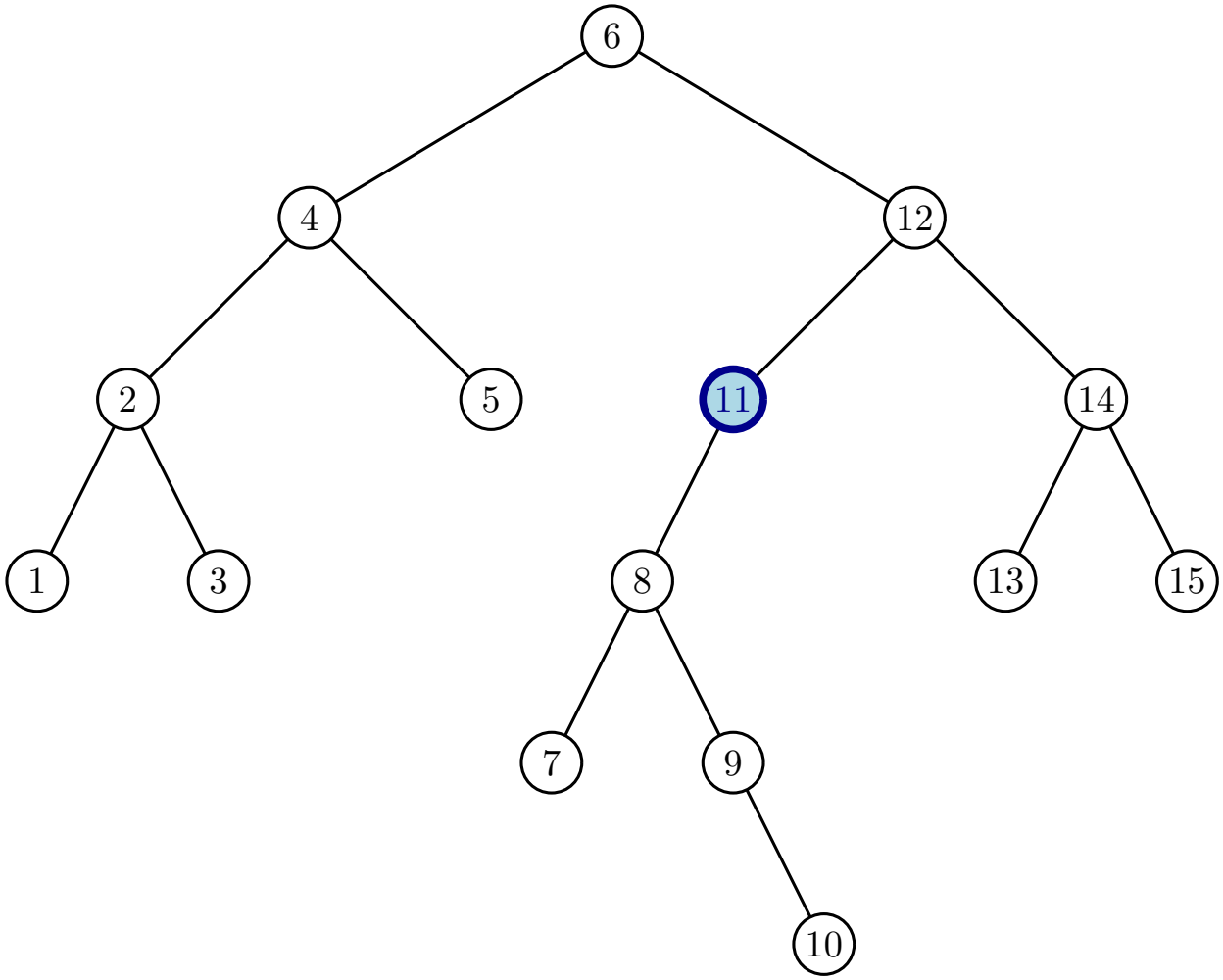
Suppression dans un ABR

si le nœud à supprimer n'a pas d'enfant



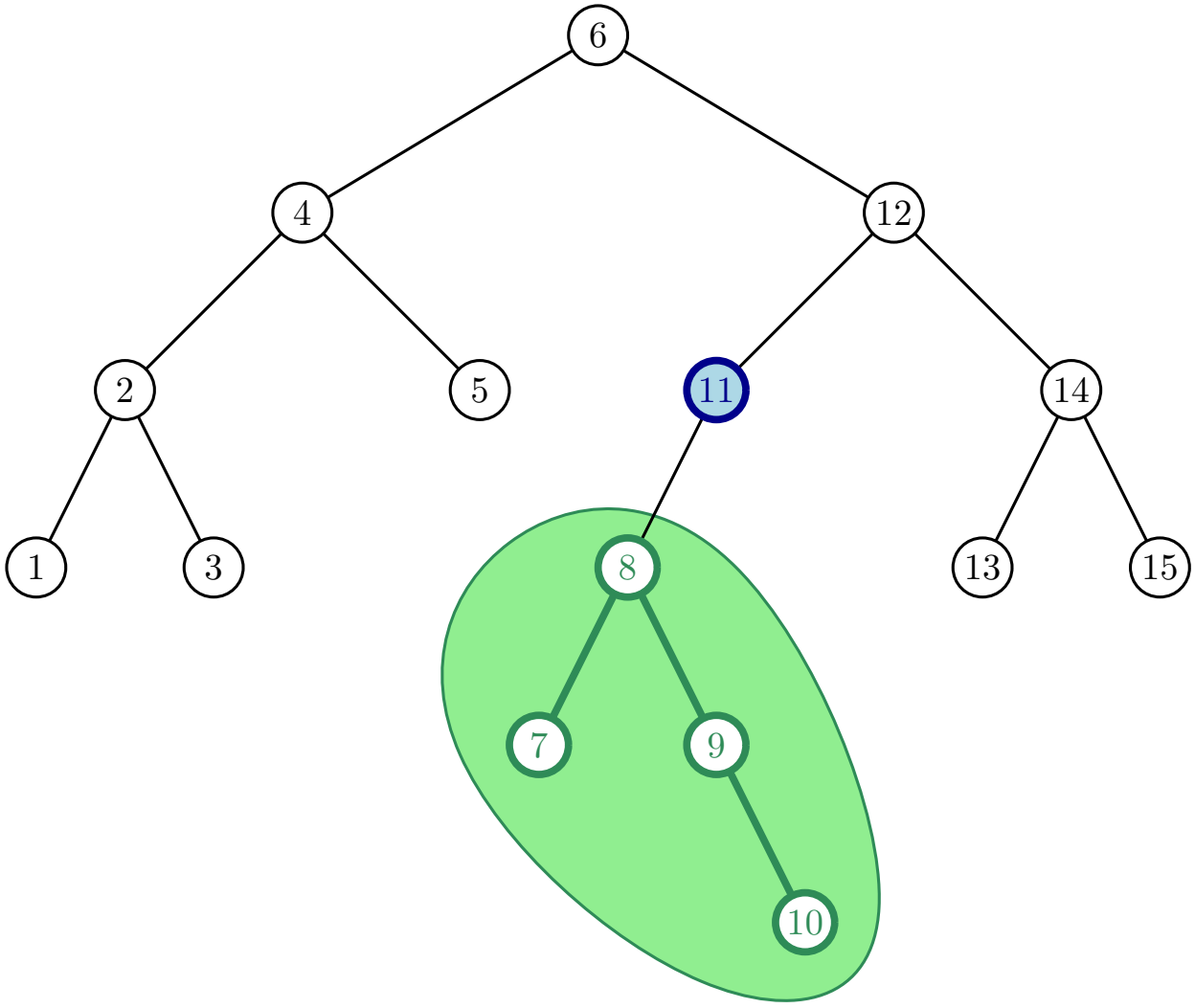
Suppression dans un ABR

si le nœud à supprimer n'a qu'un enfant



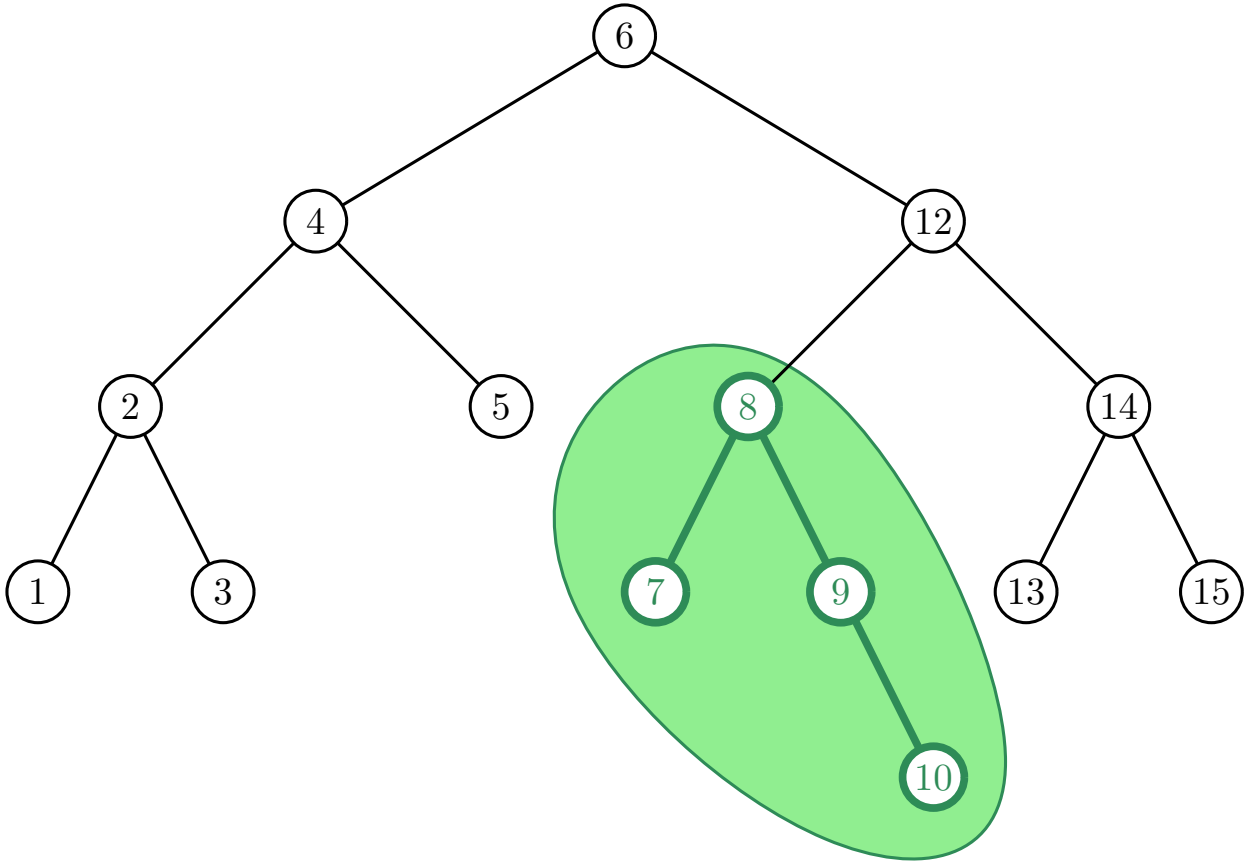
Suppression dans un ABR

si le nœud à supprimer n'a qu'un enfant



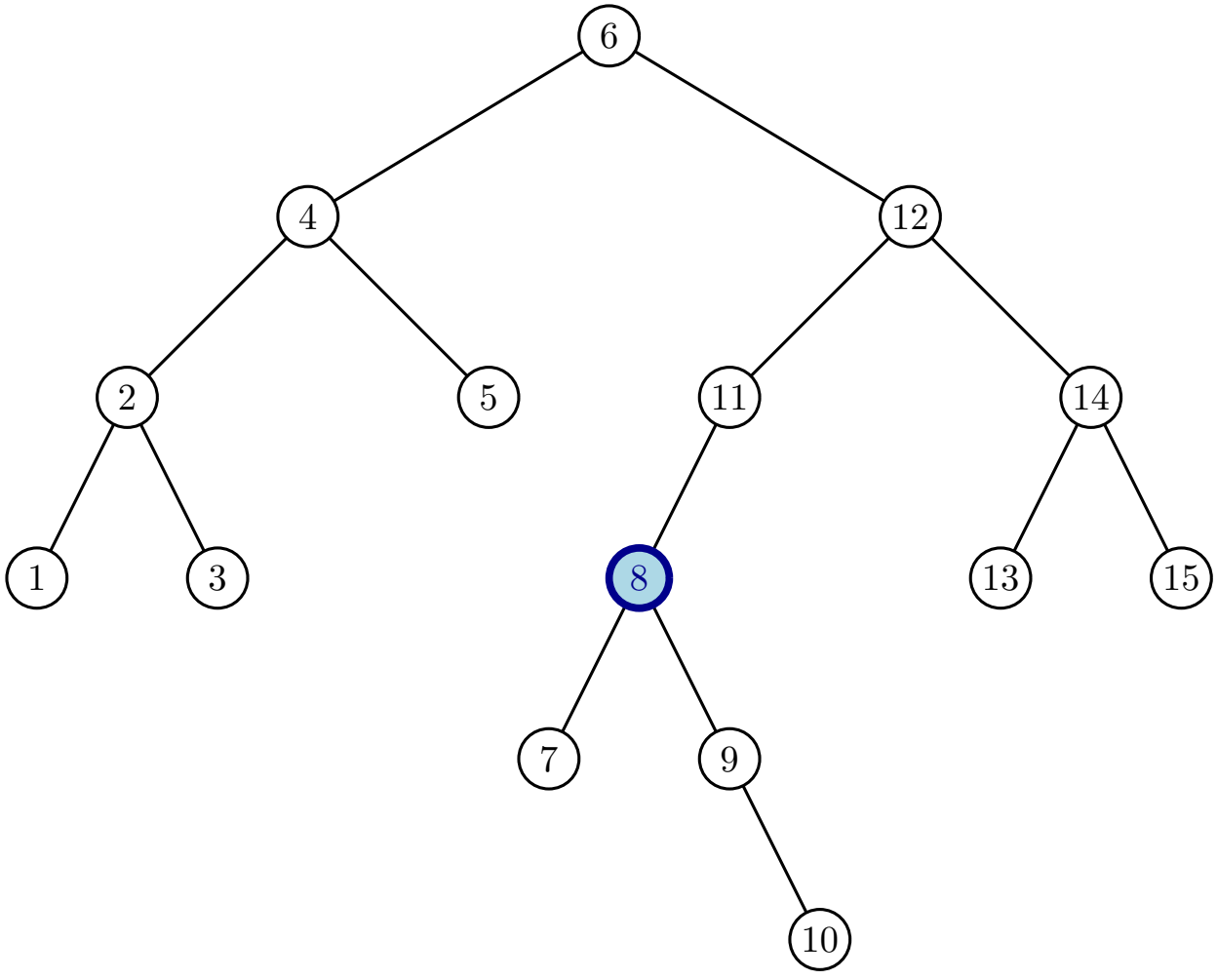
Suppression dans un ABR

si le nœud à supprimer n'a qu'un enfant



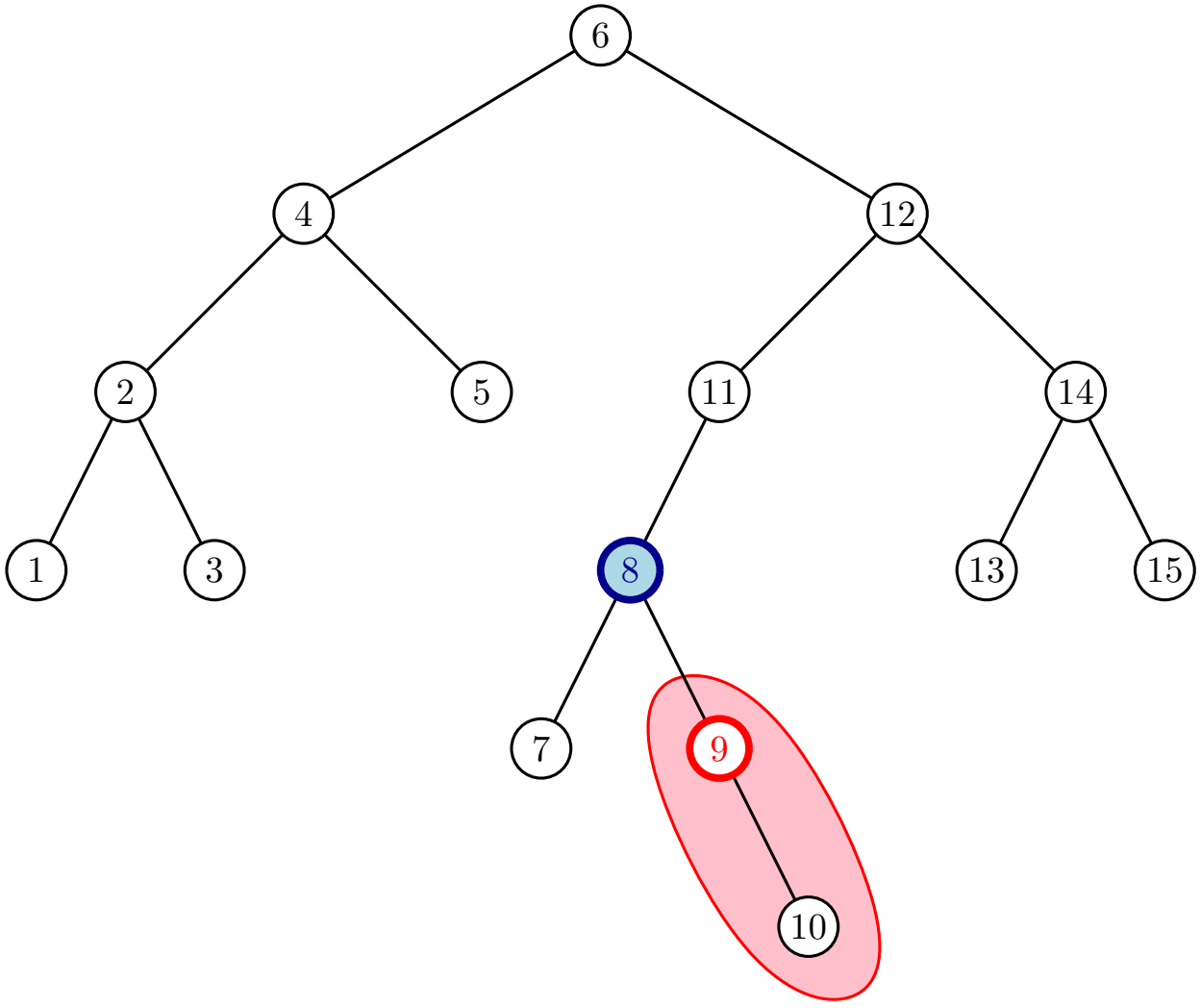
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



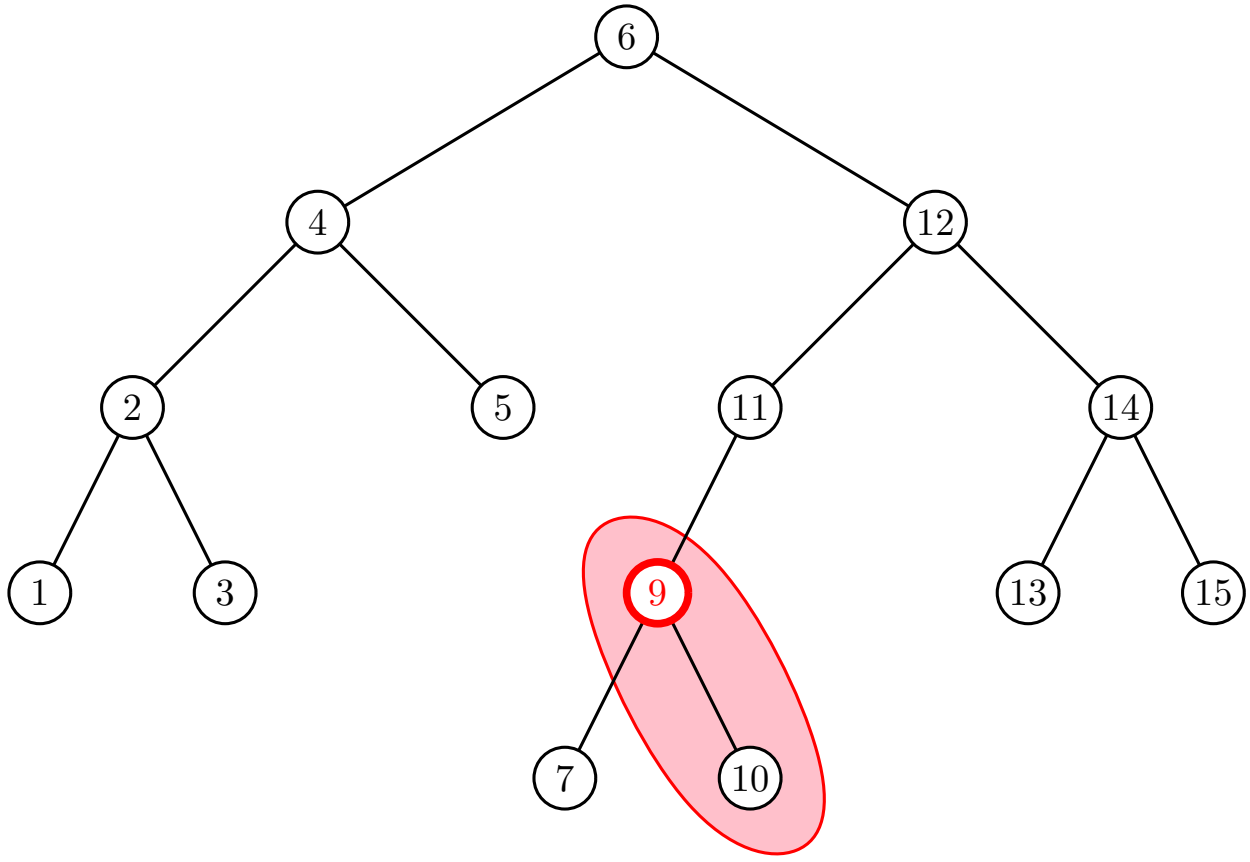
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



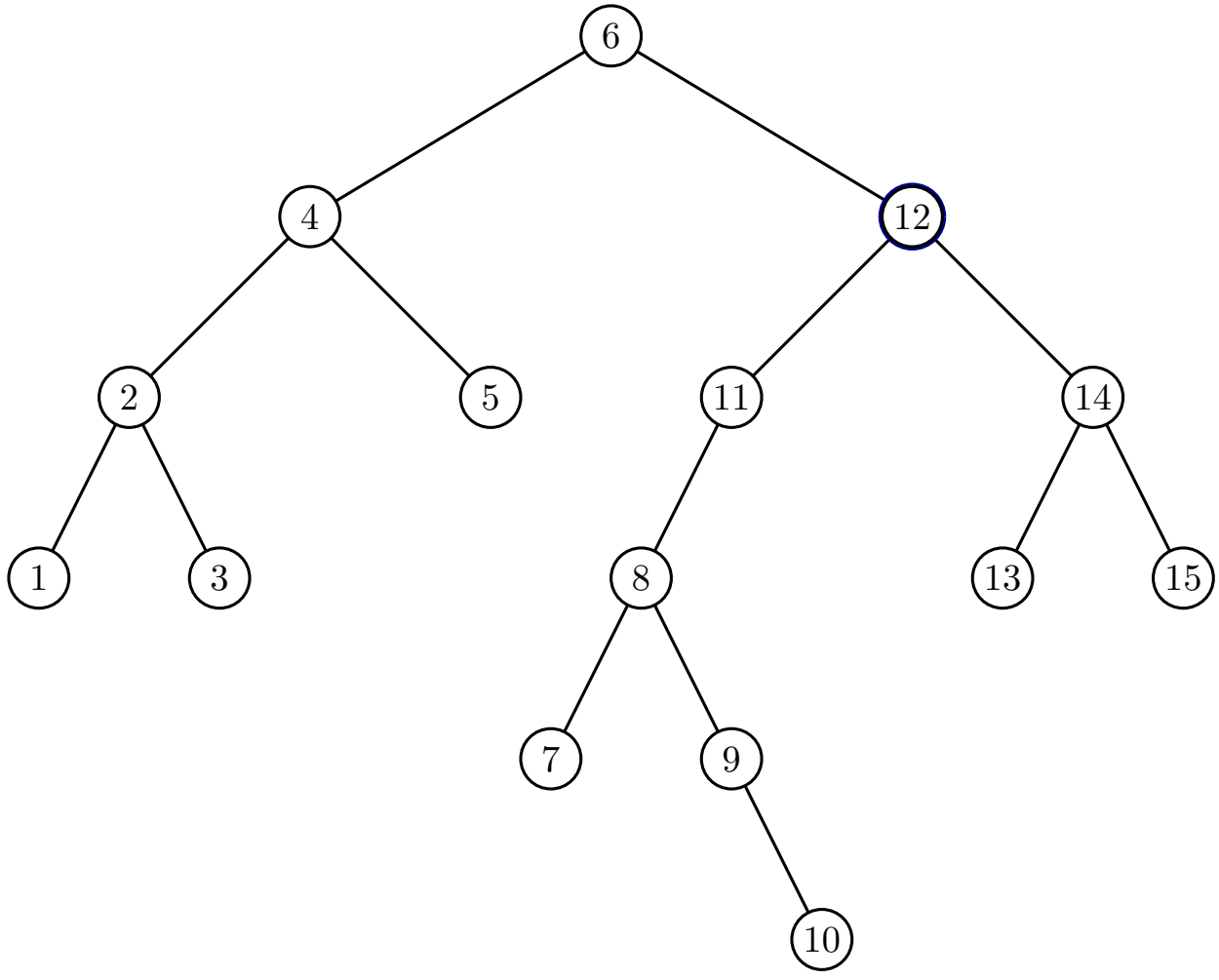
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



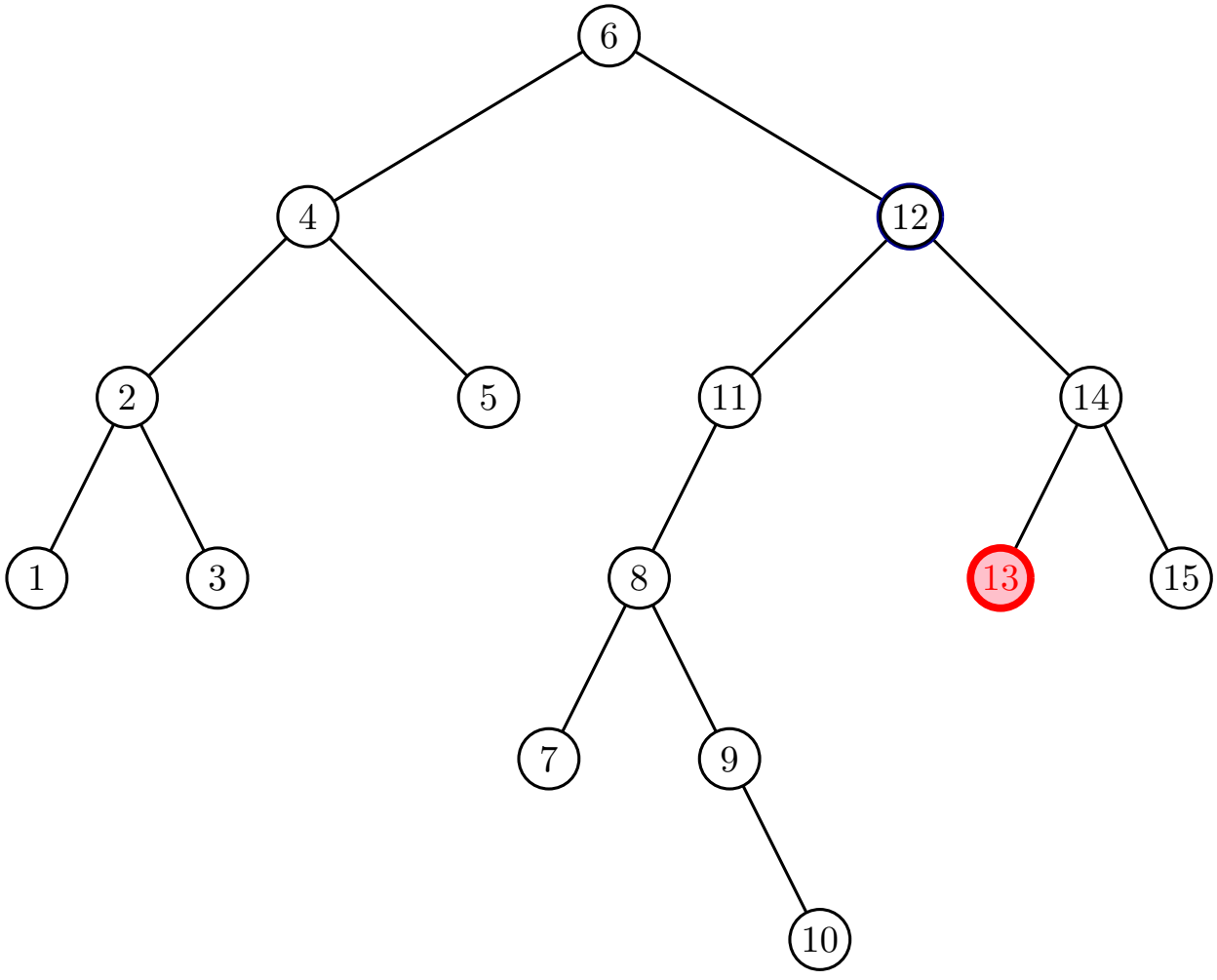
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



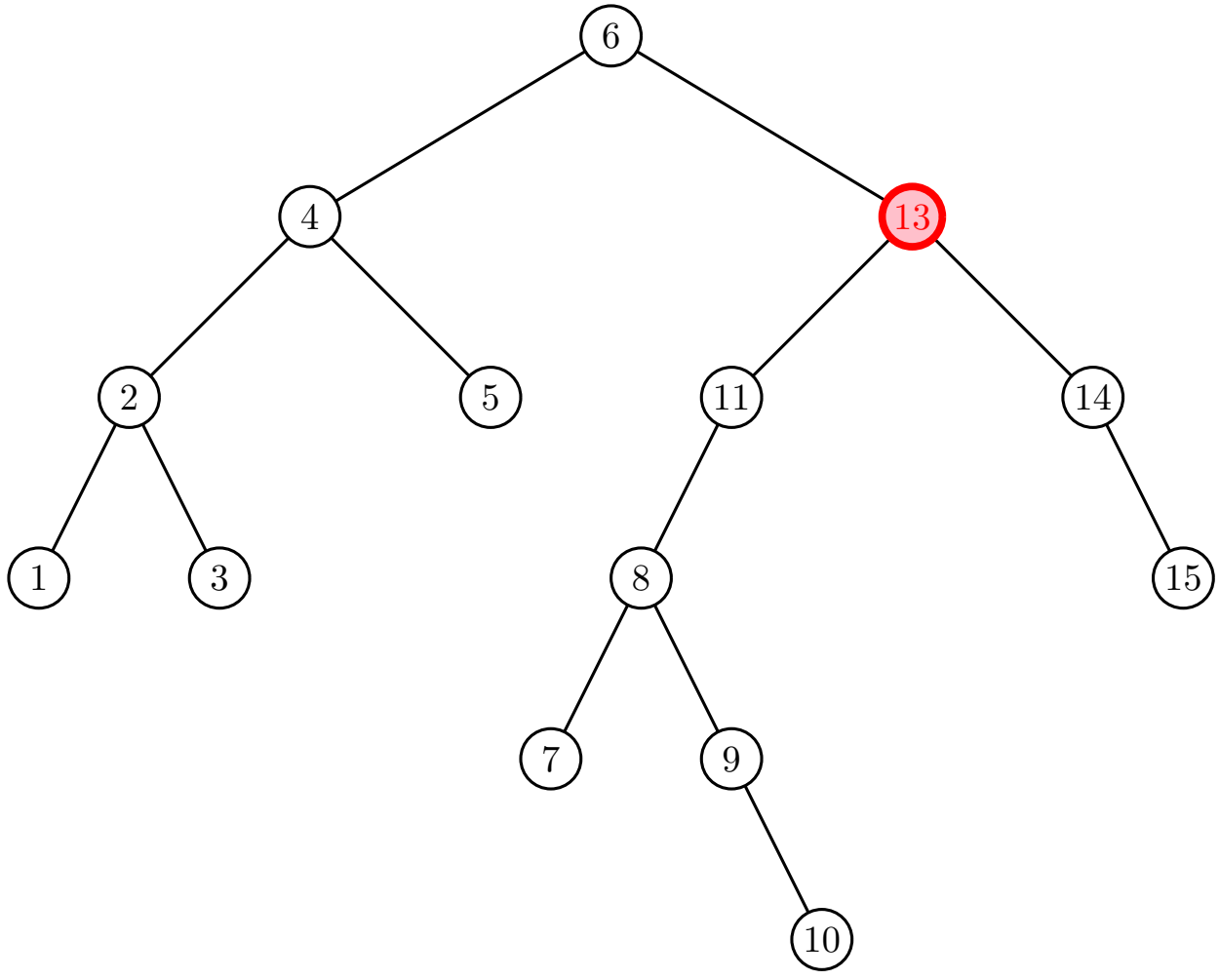
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



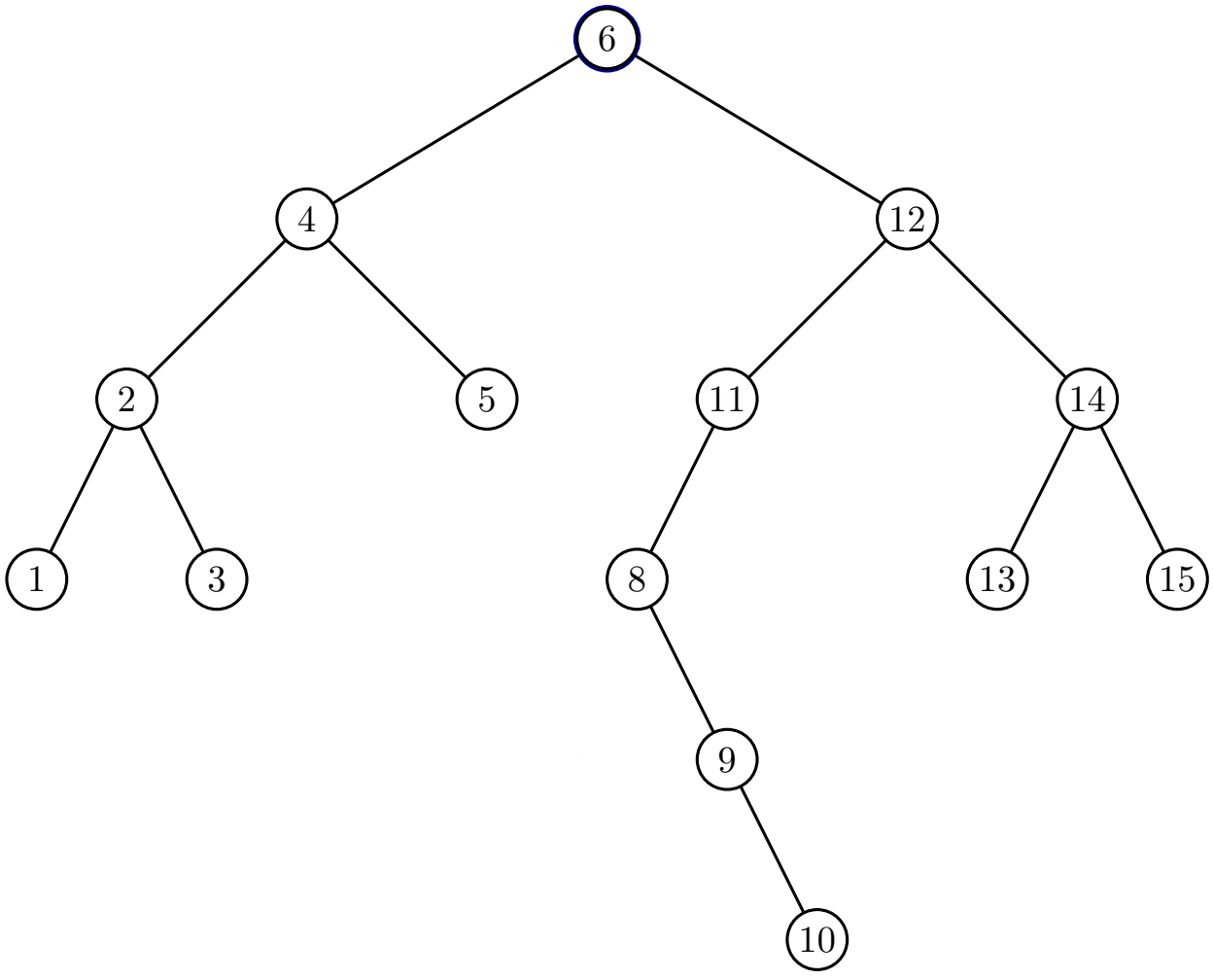
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



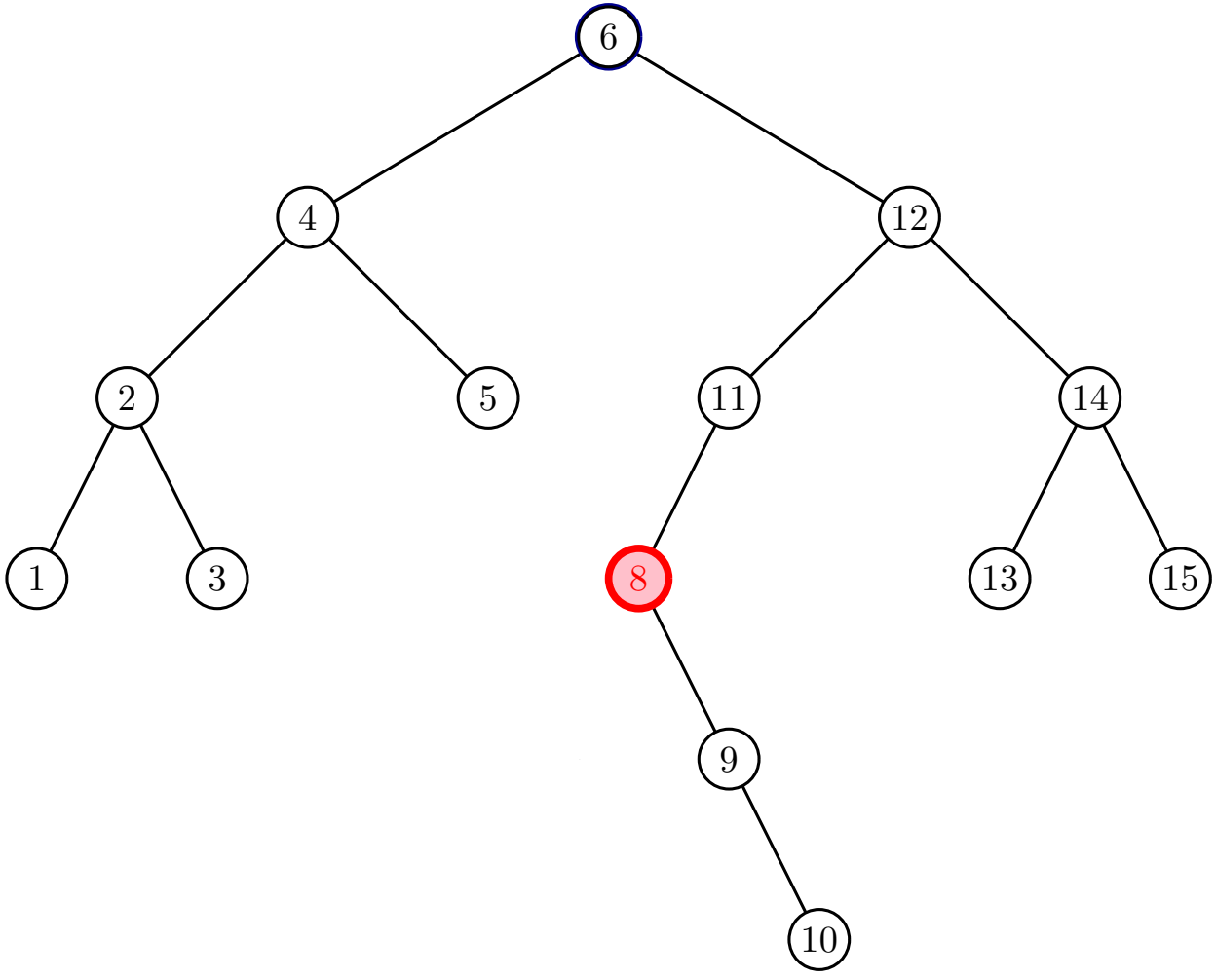
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



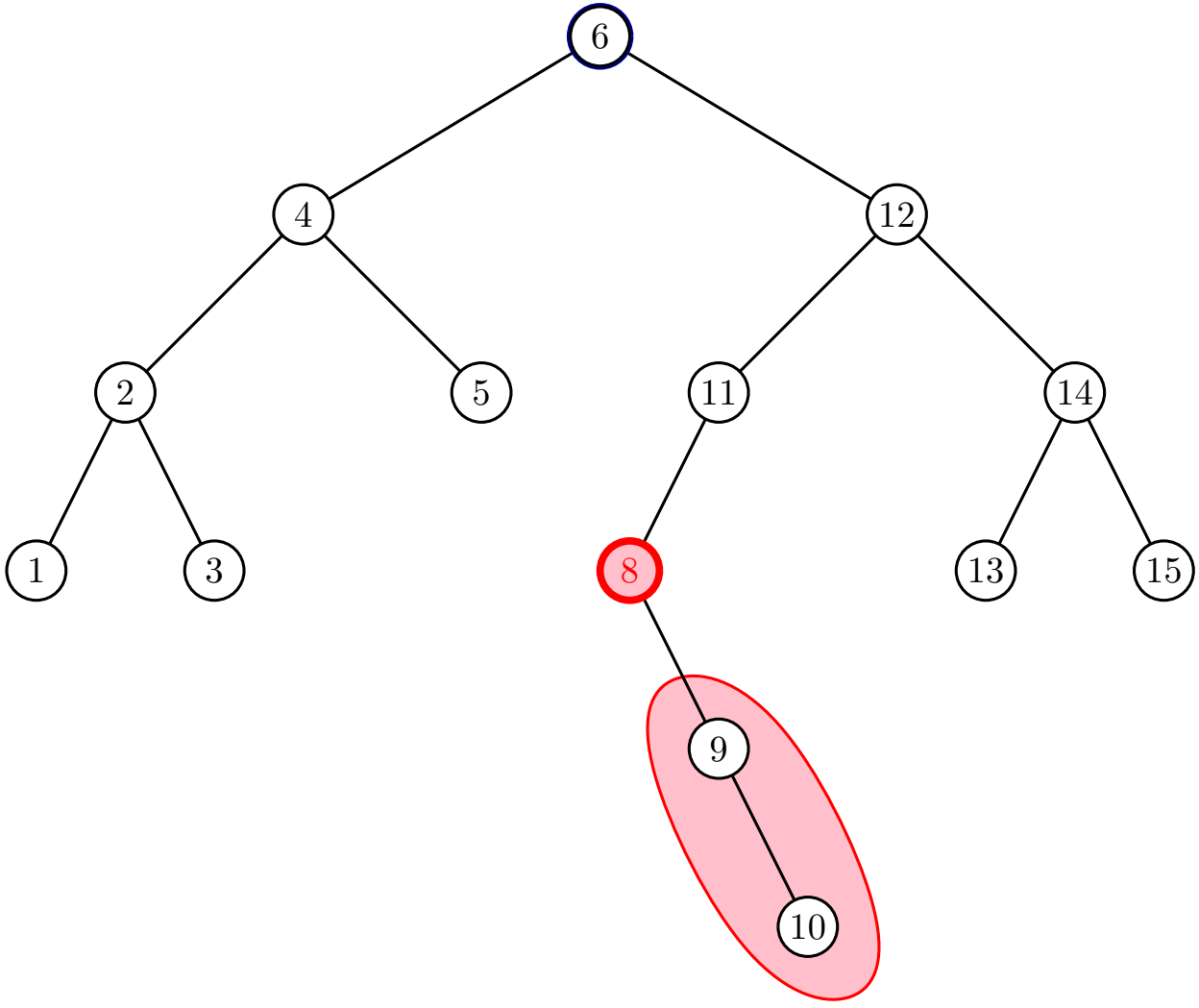
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



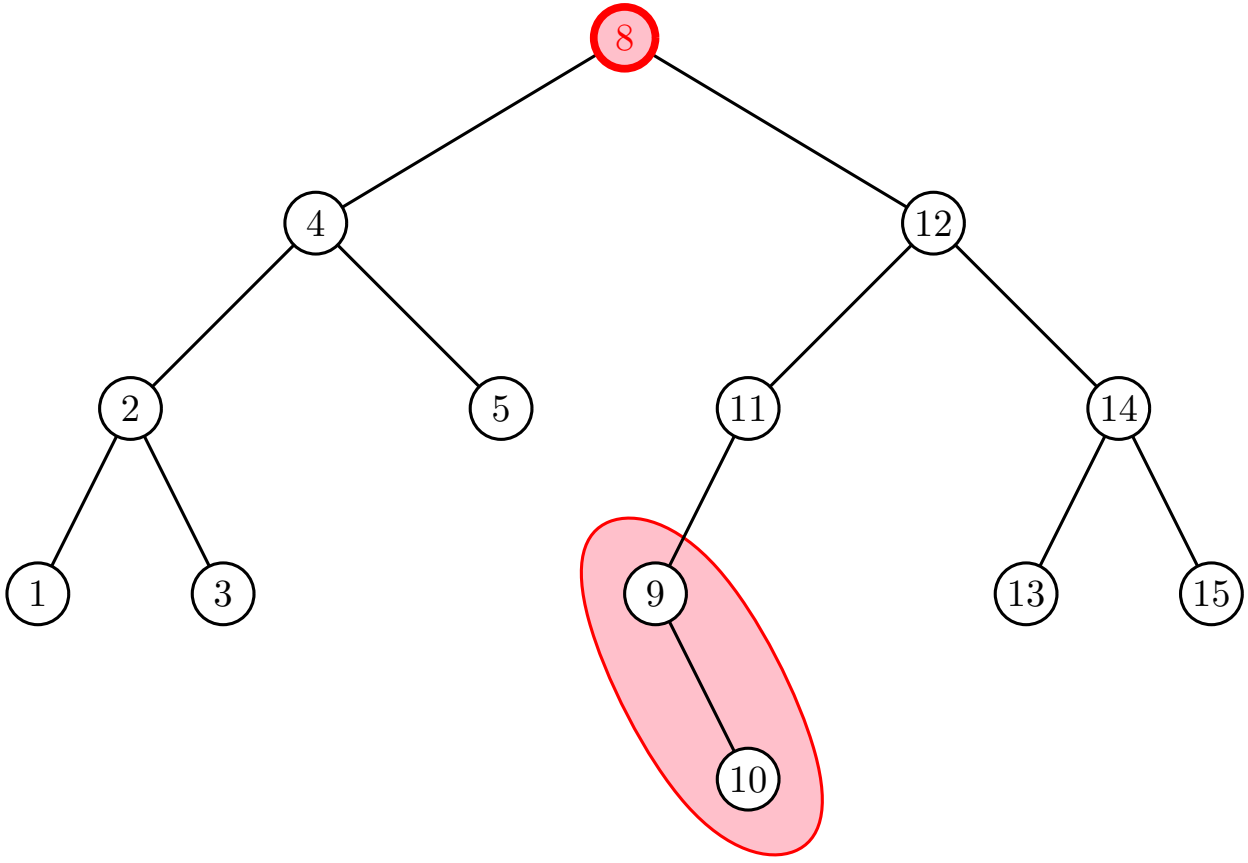
Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



Suppression dans un ABR

sinon : remplacer le nœud à supprimer par son successeur



Suppression dans un ABR

cas d'une feuille : suppression simple

cas d'un nœud à un seul fils : l'autre fils remonte d'un niveau

cas où le successeur est le fils droit : le fils droit remonte d'un niveau et adopte son frère

autres cas : le noeud est remplacé par son successeur, dont l'unique fils (droit) remonte d'un niveau

Suppression dans un ABR

cas d'une feuille : suppression simple

cas d'un nœud à un seul fils : l'autre fils remonte d'un niveau

cas où le successeur est le fils droit : le fils droit remonte d'un niveau et adopte son frère

autres cas : le noeud est remplacé par son successeur, dont l'unique fils (droit) remonte d'un niveau

remarque : la même manipulation peut être faite avec le prédécesseur plutôt que le successeur

Suppression dans un ABR

cas d'une feuille : suppression simple

cas d'un nœud à un seul fils : l'autre fils remonte d'un niveau

cas où le successeur est le fils droit : le fils droit remonte d'un niveau et adopte son frère

autres cas : le noeud est remplacé par son successeur, dont l'unique fils (droit) remonte d'un niveau

Théorème

la suppression d'un nœud d'un ABR de hauteur h *se fait en temps* $\Theta(h)$ *au pire.*