Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université de Paris
L2 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2020-2021

Arbres Binaires de Recherche

I. Motivations – Rappels

Quelle structure de données pour représenter un ensemble ?

Solution 1 : une liste de ses éléments

Problème : il est naturel de souhaiter que la complexité en espace de la représentation dépende uniquement de l'ensemble représenté
Or ce n'est pas le cas si chaque élément de l'ensemble peut apparaître un nombre quelconque de fois dans la structure qui le représente

Ce n'est pas seulement un problème de complexité en espace : si un ensemble de n *éléments est représenté par une liste, avec répétitions, de longueur* ℓ*, tous les algorithmes de manipulation de l'ensemble auront également une complexité en temps dépendant de* ℓ *et non de* n*. Par exemple, si la liste est non triée, il faudra parcourir les* ℓ *éléments avant d'être sûr qu'un élément ne s'y trouve pas.*

Donc : il est préférable d'interdire les doublons

Quelle structure de données pour représenter un ensemble ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

(dans le pire cas)	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
minimum/maximum		$\Theta(1)$		$\Theta(1)$
sélection du k^e				$\Theta(k)$
union/intersection/...	$\Theta(n^2)$ (sans trier) $\Theta(n \log n)$ (en triant)	$\Theta(n)$	$\Theta(n^2)$ (sans trier) $\Theta(n \log n)$ (en triant)	$\Theta(n)$

Quelle structure de données pour représenter un ensemble ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

(dans le pire cas)	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
minimum/maximum		$\Theta(1)$		$\Theta(1)$
sélection du k^e				$\Theta(k)$
union/intersection/...	$\Theta(n^2)$ (sans trier) $\Theta(n \log n)$ (en triant)	$\Theta(n)$	$\Theta(n^2)$ (sans trier) $\Theta(n \log n)$ (en triant)	$\Theta(n)$

Justification rapide, consulter les cours précédents si nécessaire :

- pour la recherche : linéaire *vs* dichotomique
- pour la sélection : sélection rapide (complexité en moyenne, pas au pire) *vs* accès direct ou parcours des k premiers maillons
- pour l'union ou l'intersection : via la fusion de listes triées

Quelle structure de données pour représenter un ensemble ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

(dans le pire cas)	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
minimum/maximum		$\Theta(1)$		$\Theta(1)$
sélection du k^e				$\Theta(k)$
union/intersection/...	$\Theta(n^2)$ (sans trier) $\Theta(n \log n)$ (en triant)	$\Theta(n)$	$\Theta(n^2)$ (sans trier) $\Theta(n \log n)$ (en triant)	$\Theta(n)$

$\Longrightarrow$ pour ces opérations ***statiques***, supériorité nette du tableau trié

Quelle structure de données pour représenter un ensemble ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

Quid des opérations *dynamiques* ?

(un ensemble a vocation à évoluer par ajout ou suppression d'éléments)

Quelle structure de données pour représenter un ensemble ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

Quid des opérations *dynamiques* ?
(un ensemble a vocation à évoluer par ajout ou suppression d'éléments)

Remarque : chaque insertion/suppression nécessite une recherche préalable

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
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 (bis) : une liste de ses éléments, sans doublon

Quid des opérations *dynamiques* ?
(un ensemble a vocation à évoluer par ajout ou suppression d'éléments)

Remarque : chaque insertion/suppression nécessite une recherche préalable

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

$\Longrightarrow$ coût linéaire pour chaque forme de liste !

*Le tableau est une forme trop **rigide** pour permettre des évolutions peu coûteuses : tout nouvel élément n'a qu'une seule place possible, et lui faire une place au bon endroit ne peut pas se faire uniquement via des modifications locales.*

Quelle structure de données pour représenter un ensemble ?

Moralité : il faudrait trouver une structure ayant à la fois

- la souplesse de la liste chaînée,
- le caractère « ordonné » des listes triées,
- un moyen d'accès rapide à n'importe quel élément

(peut-être pas en $O(1)$, mais pas en $\Theta(n)$)

(et toujours sans doublon pour ne pas manipuler des structures de taille inutilement grosse)

Quelle structure de données pour représenter un ensemble ?

Moralité : il faudrait trouver une structure ayant à la fois

- la souplesse de la liste chaînée,
- le caractère « ordonné » des listes triées,
- un moyen d'accès rapide à n'importe quel élément

(peut-être pas en $O(1)$*, mais pas en* $\Theta(n)$*)*

(et toujours sans doublon pour ne pas manipuler des structures de taille inutilement grosse)

Solution 2 : un arbre binaire, « trié », sans doublon

Quelle structure de données pour représenter un ensemble ?

Moralité : il faudrait trouver une structure ayant à la fois

- la souplesse de la liste chaînée,
- le caractère « ordonné » des listes triées,
- un moyen d'accès rapide à n'importe quel élément

(peut-être pas en $O(1)$, mais pas en $\Theta(n)$)

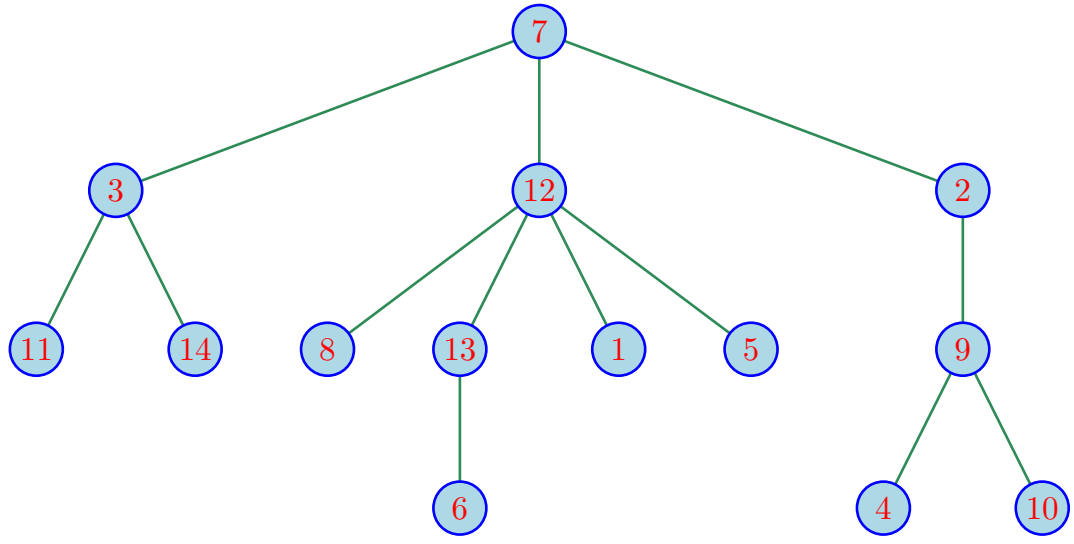
(et toujours sans doublon pour ne pas manipuler des structures de taille inutilement grosse)

Solution 2 : un arbre binaire, « trié », sans doublon

- multiples formes possibles $\Longrightarrow$ souplesse pour ajouter ou supprimer un élément en se contentant de modifications locales,
- on veut ranger les éléments – les petits à gauche, et les grands à droite, pour guider la recherche,
- si l'arbre est « à peu près » équilibré, aucun élément ne sera « très loin » de la racine (on peut espérer une hauteur en $\Theta(\log n)$.

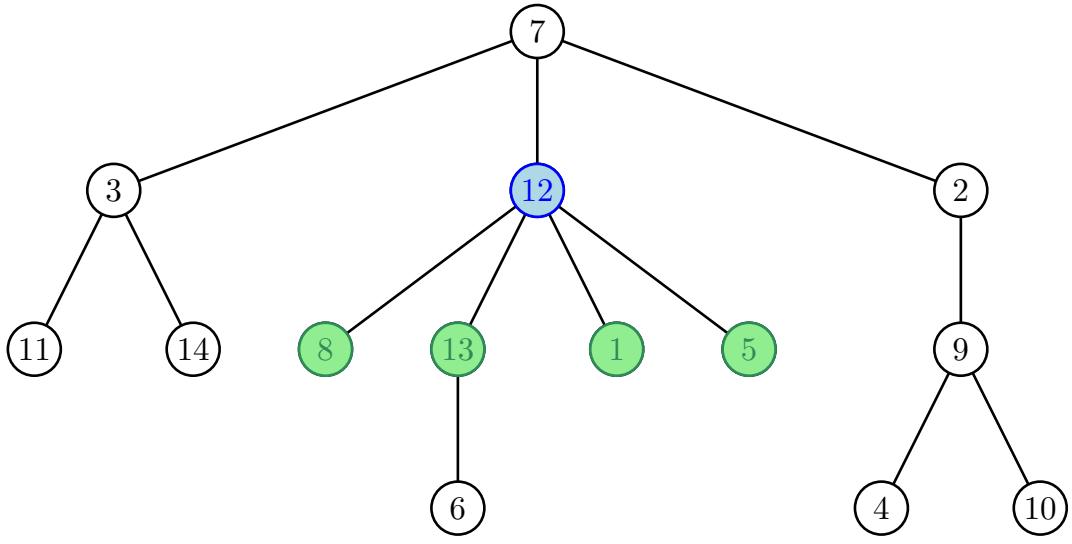
Arbres Binaires de Recherche

II. Généralités sur les arbres



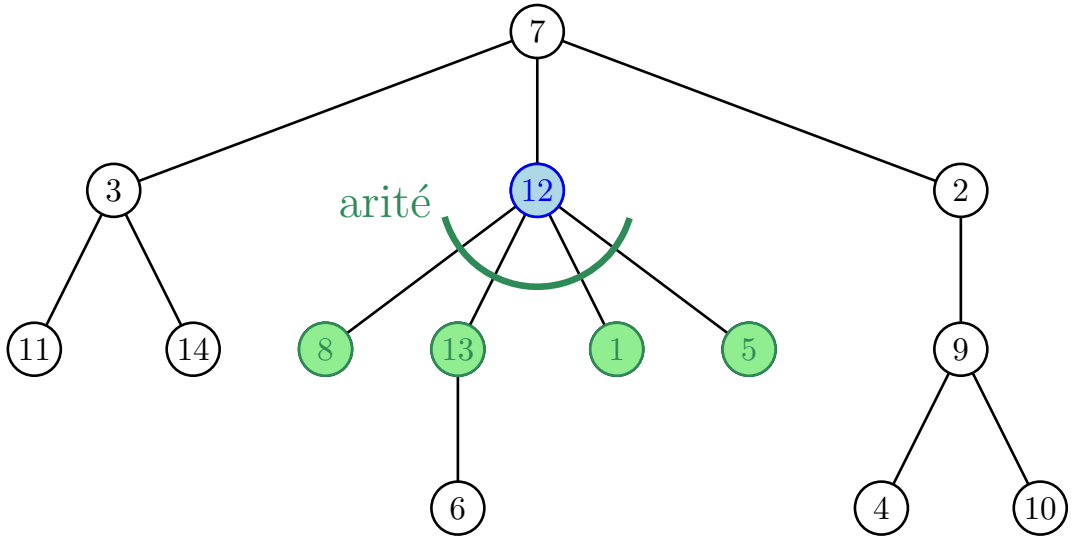
sommets contenant des étiquettes reliés par des arêtes

Ici, 14 sommets, reliés par 13 arêtes, et étiquetés de 1 à 14



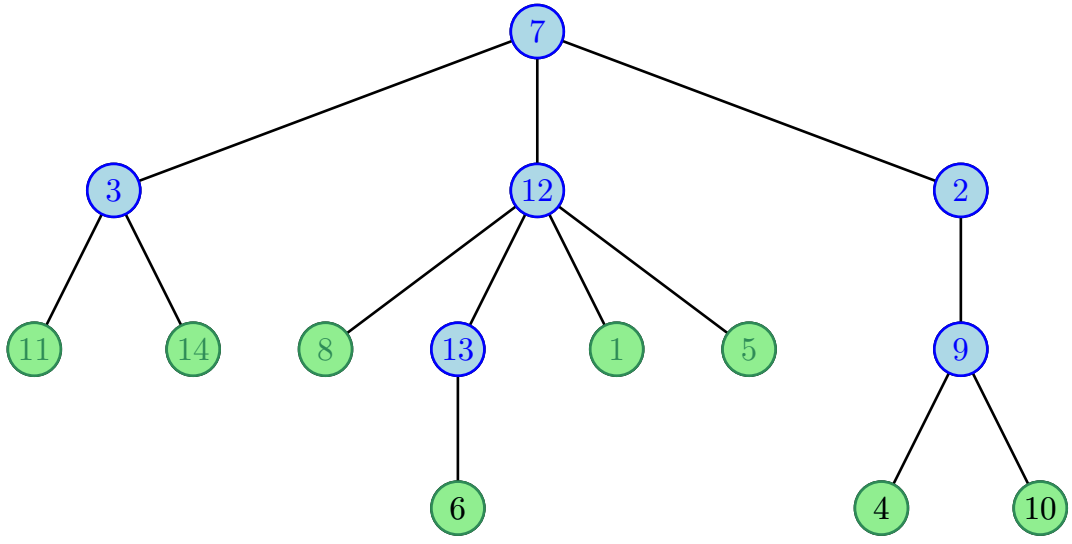
hiérarchie entre les sommets : père , fils

Le sommet d'étiquette 12 a 4 fils (d'étiquettes 8, 13, 1 et 5).



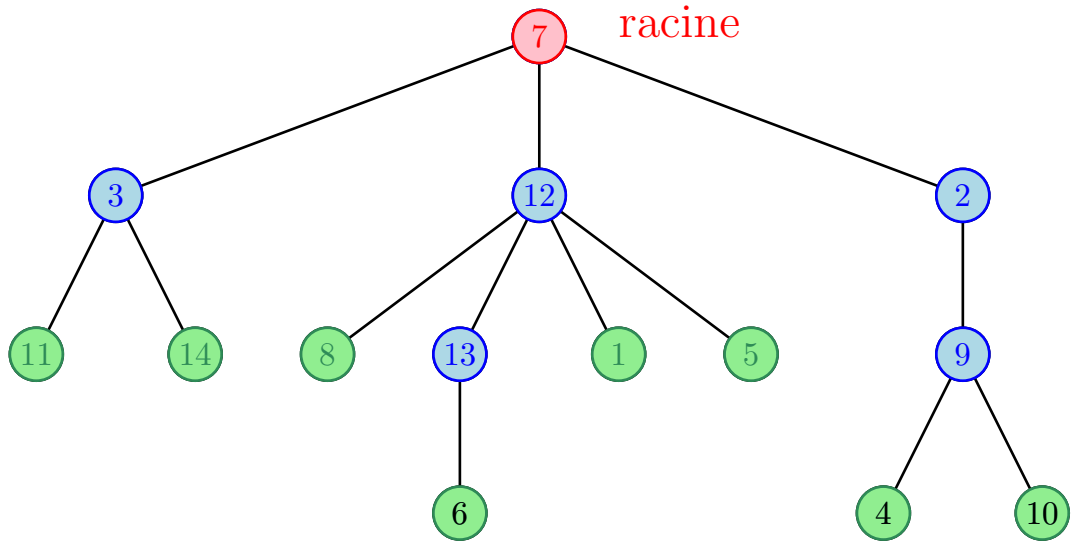
hiérarchie entre les sommets : père , fils

Le sommet d'étiquette 12 a 4 fils (d'étiquettes 8, 13, 1 et 5).
On dit qu'il est d'arité 4, ou que c'est un nœud quaternaire.



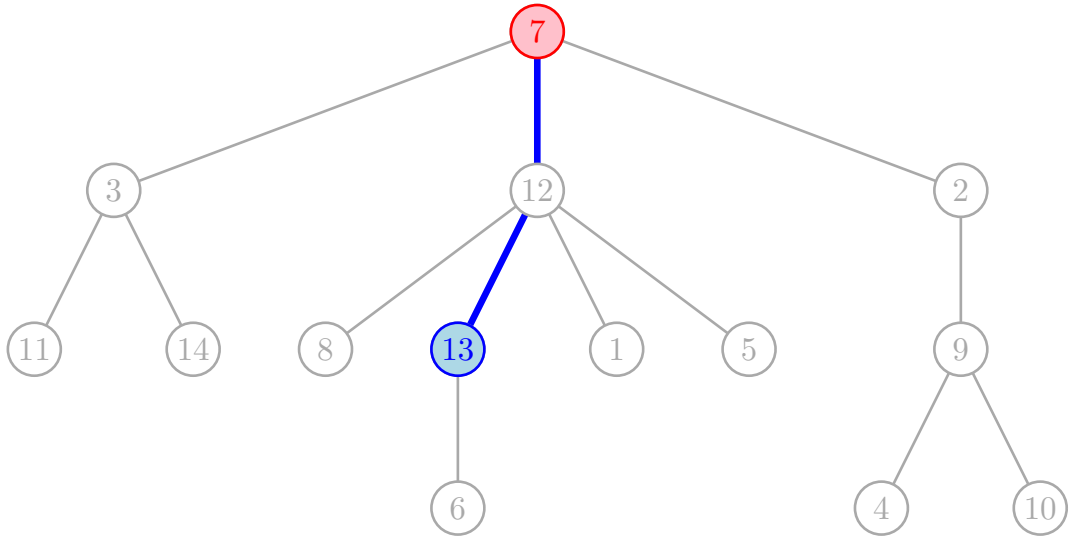
sommet = nœud ou feuille

Les sommets d'arité 0 sont appelés *feuilles* – ici il y en a 8.
Les autres sont les *nœuds* – ici, 6.



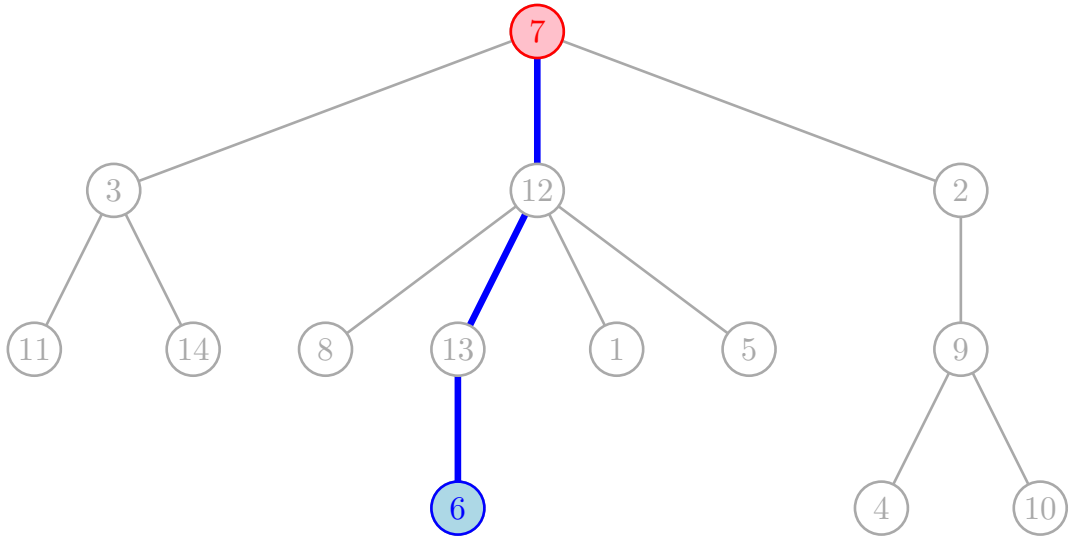
sommet = nœud ou feuille

Les sommets d'arité 0 sont appelés *feuilles* – ici il y en a 8.
Les autres sont les *nœuds* – ici, 6.
Le seul sommet sans père est la racine – c'est en général un nœud.



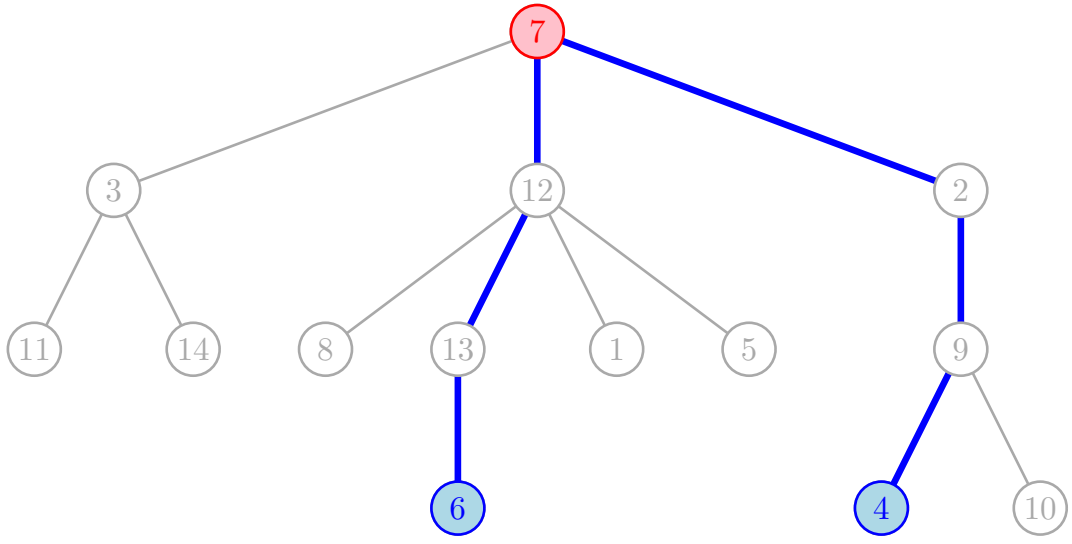
profondeur d'un sommet = distance à la racine

Le sommet d'étiquette 13 est à profondeur 2.



hauteur de l'arbre = profondeur maximale

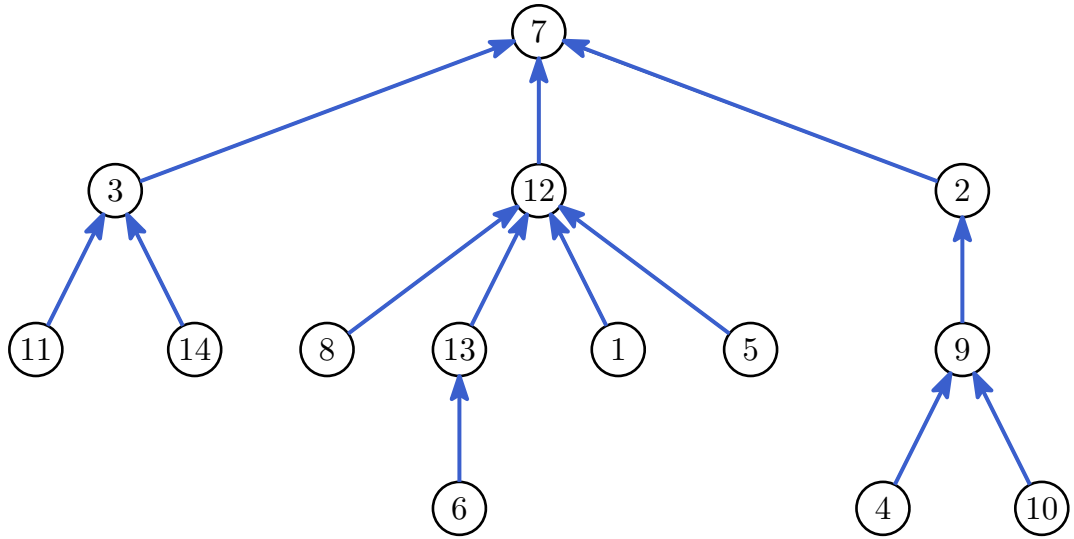
La hauteur de cet arbre est 3.



hauteur de l'arbre = profondeur maximale

La hauteur de cet arbre est 3.
Les sommets à profondeur 3 sont donc des feuilles
(et il peut très bien y en avoir plusieurs)

en gardant pour chaque sommet la référence du père (dans le sommet, ou regroupées dans un tableau)

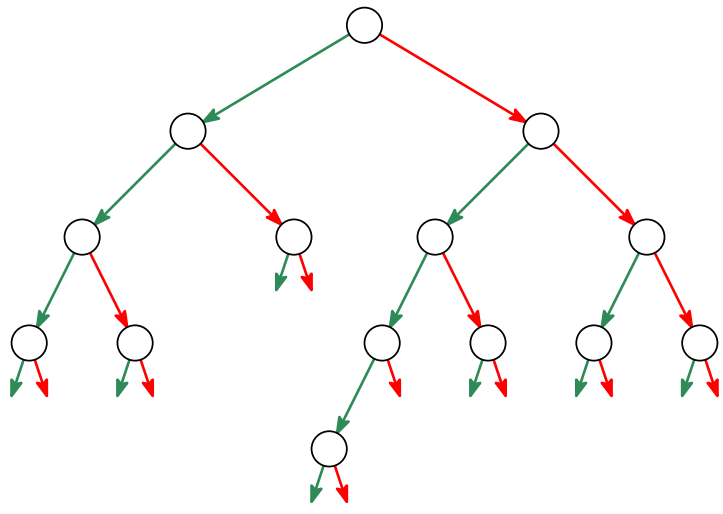


avantage : représentation extrêmement compacte

(gros) inconvénient : l'arbre ne peut être parcouru (efficacement) que de bas en haut... ce qui n'est pas ce qu'on souhaite faire en général

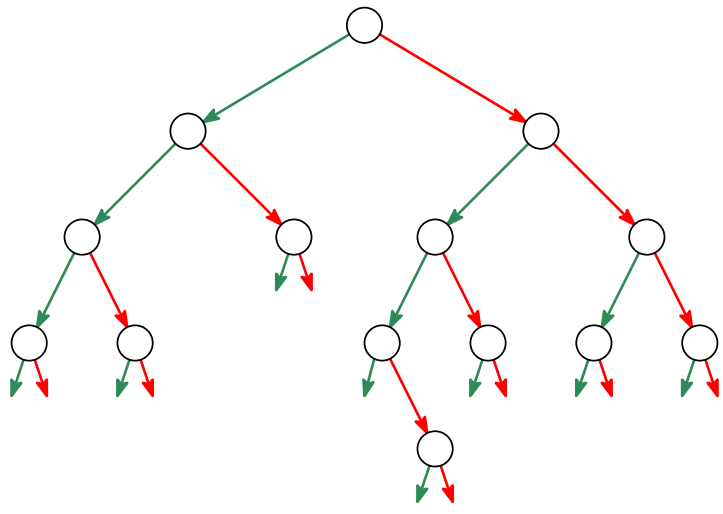
$\Longrightarrow$ essentiellement utilisée pour représenter une partition en sous-ensembles (structure *union-find*, *cf* cours d'algo de L3)

cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils



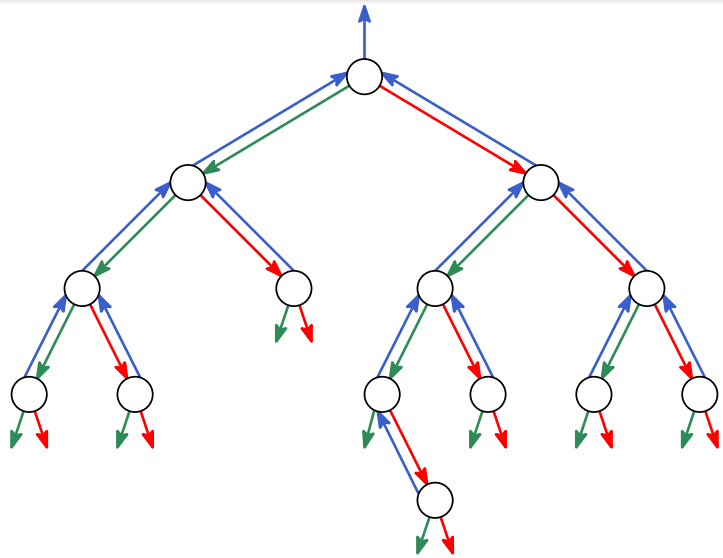
arbre binaire : au plus 2 fils par nœud, avec distinction entre fils gauche et fils droit. *(échanger les deux fils donne un arbre différent)*

cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils



arbre binaire : au plus 2 fils par nœud, avec distinction entre fils gauche et fils droit. *(échanger les deux fils donne un arbre différent)*

cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils (et éventuellement du père)

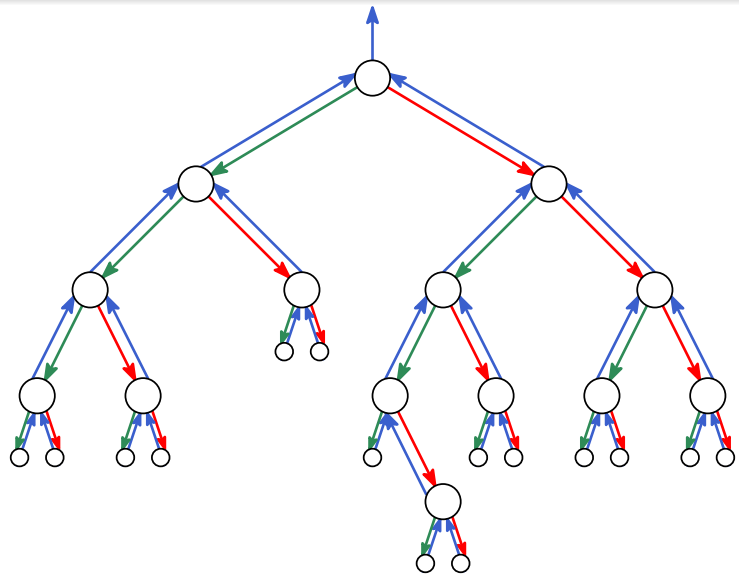


arbre binaire : au plus 2 fils par nœud, avec distinction entre fils gauche et fils droit. *(échanger les deux fils donne un arbre différent)*

Avec le père, chaque sommet stocke donc 3 références/pointeurs/champs. Ce 3e pointeur facilite énormément les manipulations, en particulier les parcours.

Mais certains ne pointent sur « rien » : le père de la racine, les fils absents. Cela peut en fait compliquer les choses : ces « rien » ne savent rien... impossible de distinguer le « non-fils-gauche » d'un sommet du « non-fils-droit » d'un autre.

cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils (et éventuellement du père)



Il est plus commode de *compléter* un arbre binaire en rajoutant une feuille vide à la place de chaque pointeur vers un sommet absent. C'est plus homogène, et cela facilite grandement la programmation des modifications.

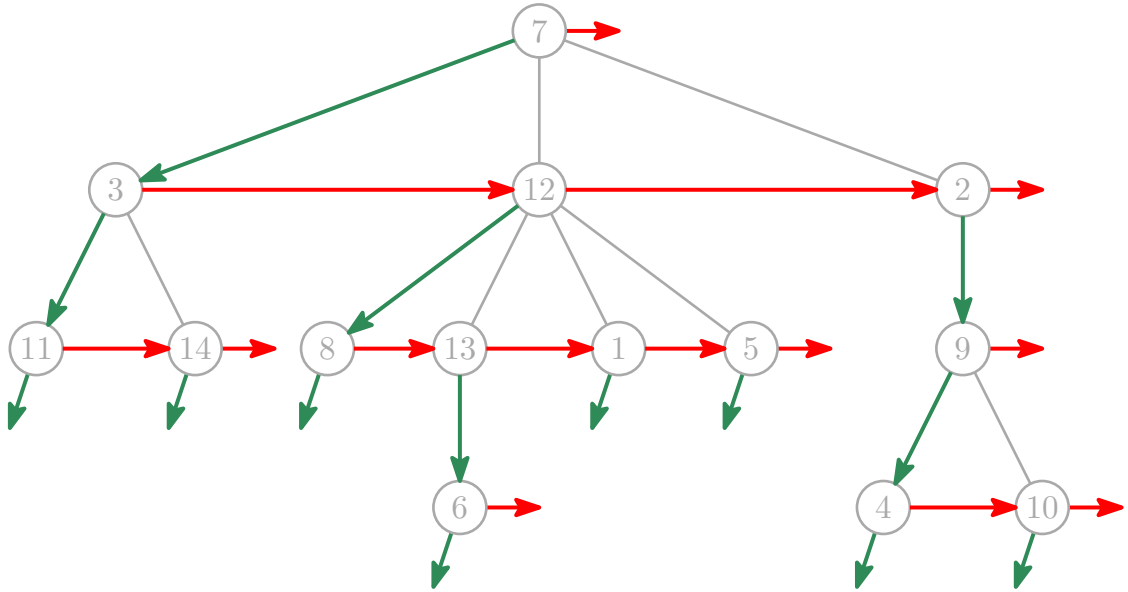
on manipule donc plutôt des arbres binaires *complets* : exactement 2 fils par nœud

la « complétion » met exactement en correspondance les arbres binaires (quelconques) à n sommets et les arbres binaires complets à n nœuds binaires (et $n+1$ feuilles vides).

Représentation des arbres – solution III

(Pour la culture générale, mais pas indispensable pour le cours sur les ABR)

cas général : références du fils aîné et du frère cadet



Le fils aîné sert de référence à la liste chaînée des fils.

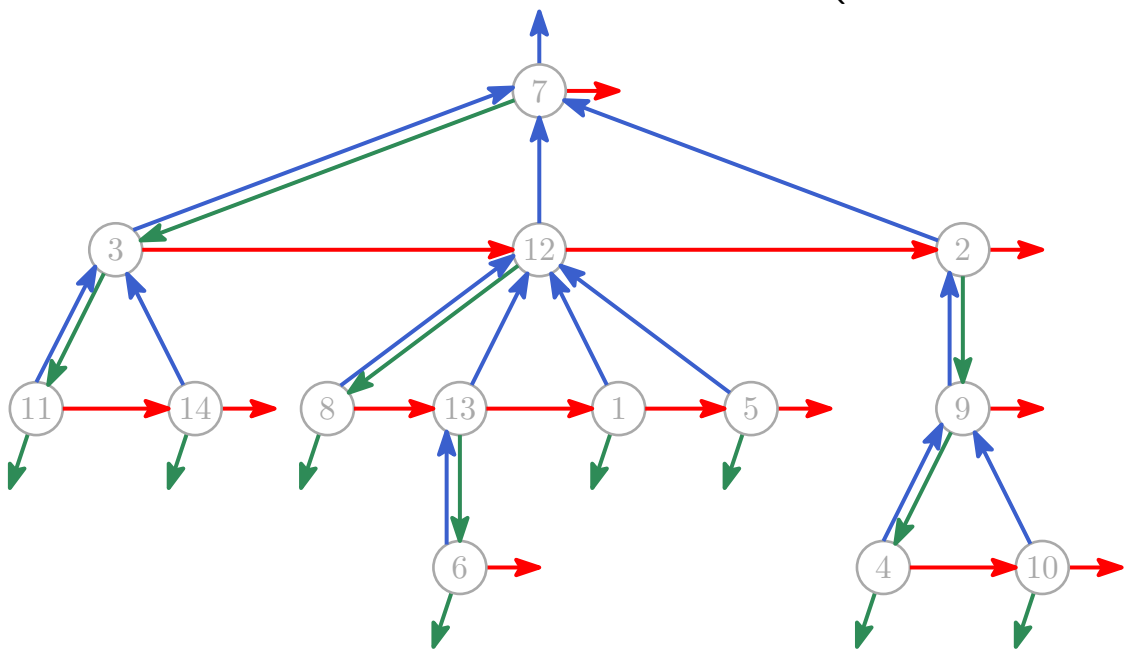
Autre manière de voir les choses, on représente les arbres généraux par des arbres binaires : le fils aîné d'un sommet devient son fils gauche, et son frère cadet devient son fils droit. On obtient un arbre binaire dont la racine n'a pas de fils droit.

Représentation des arbres – solution III

(Pour la culture générale, mais pas indispensable pour le cours sur les ABR)

cas général : références du fils aîné et du frère cadet

(et éventuellement du père)



Le fils aîné sert de référence à la liste chaînée des fils.

Autre manière de voir les choses, on représente les arbres généraux par des arbres binaires : le fils aîné d'un sommet devient son fils gauche, et son frère cadet devient son fils droit. On obtient un arbre binaire dont la racine n'a pas de fils droit.

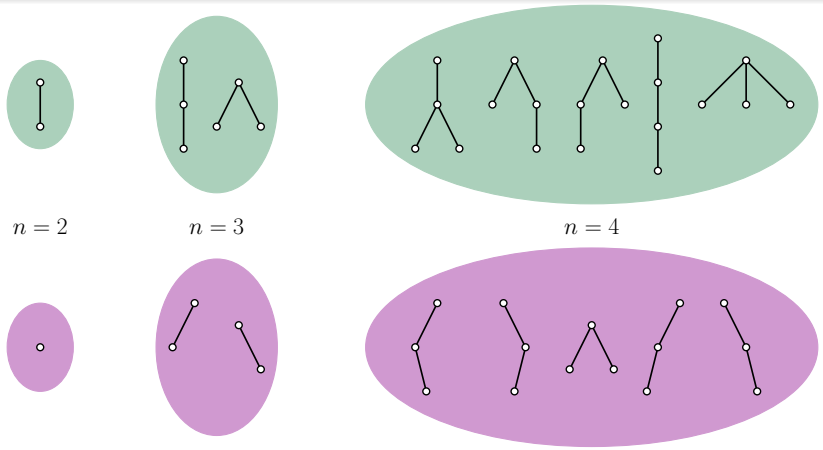
Il est plus commode que le père pointe réellement sur le père (dans l'arbre de départ), pas sur le « père » dans l'arbre binaire correspondant.

Théorème

Les arbres binaires à n sommets sont en bijection avec les arbres binaires complets à n nœuds (binaires) et $n+1$ feuilles.

Théorème

Les arbres à n sommets sont en bijection avec les arbres binaires à $n-1$ sommets (et donc avec les arbres binaires complets à $n-1$ nœuds et n feuilles)



Théorème (*admis*)

Le nombre d'arbres binaires complets à $n-1$ nœuds et n feuilles est

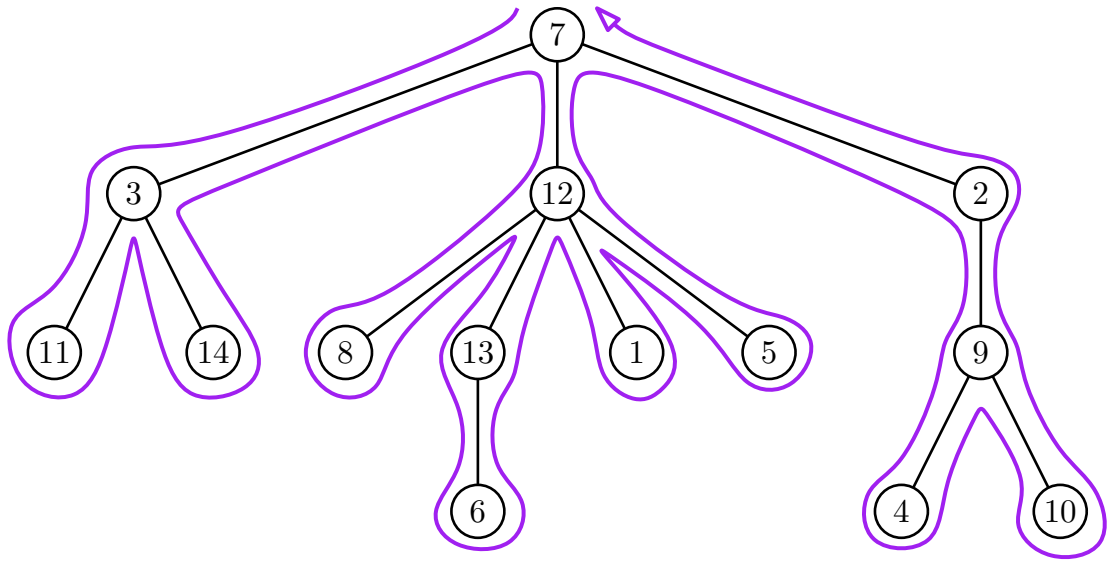
$$\frac{1}{n+1}\binom{2n}{n} \in \Theta\left(\frac{4^n}{n\sqrt{n}}\right)$$

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)`
- (pour les arbres binaires) `gauche(noeud)` et `droite(noeud)`

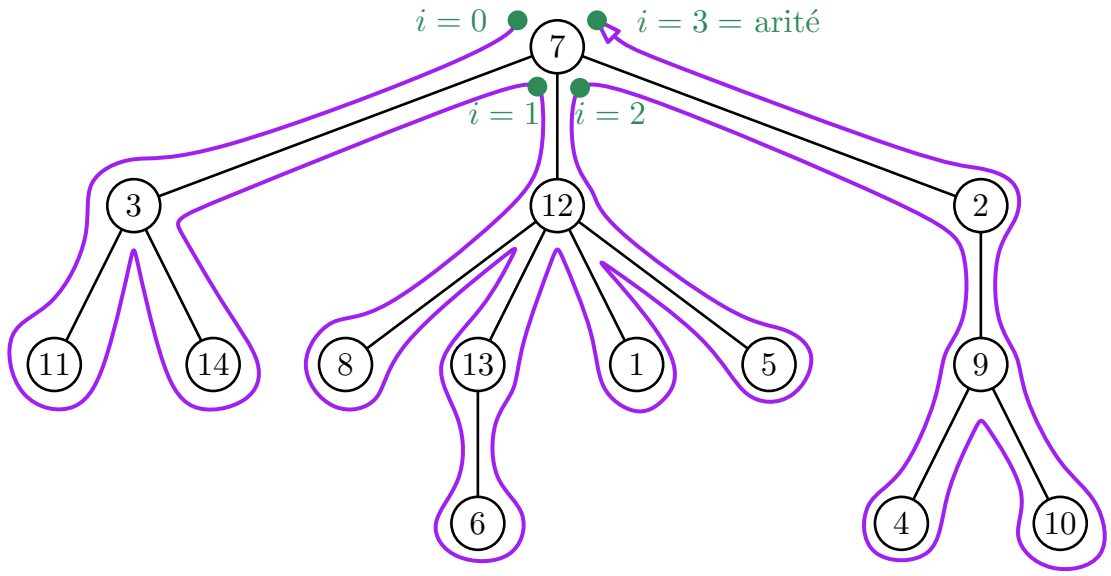
Que peut-on faire avec un arbre ? Par exemple, peut-on parcourir tous les éléments qu'il contient raisonnablement efficacement, comme une liste ?

Une solution : effectuer un parcours en profondeur d'un arbre, c'est-à-dire en faire le tour complet, en partant de la racine et en le tenant fermement de la main gauche :



Parcours en profondeur générique

Une solution : effectuer un parcours en profondeur d'un arbre, c'est-à-dire en faire le tour complet, en partant de la racine et en le tenant fermement de la main gauche :



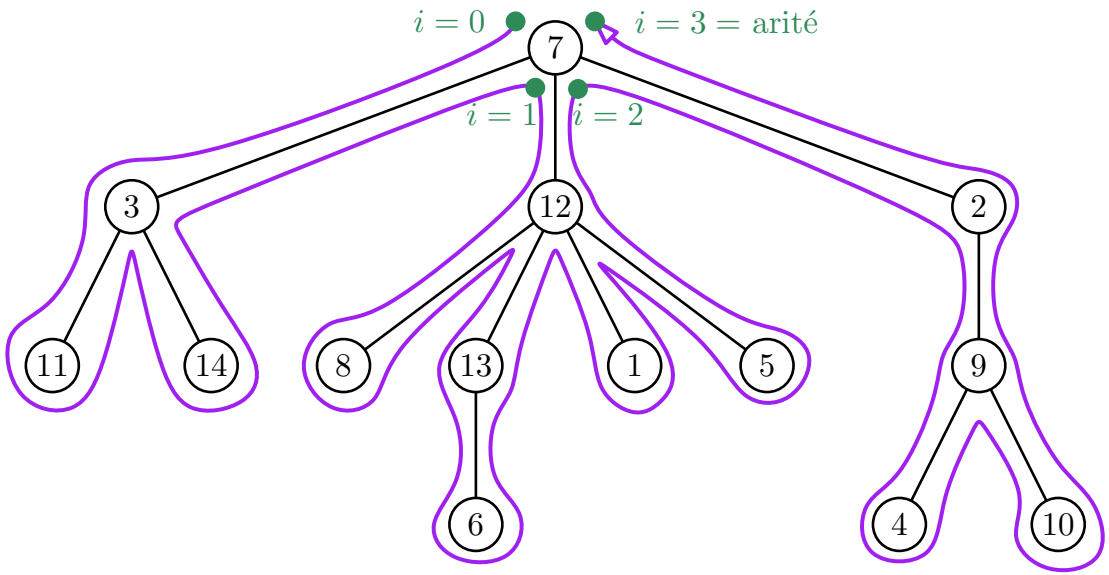
Ce tour passe plusieurs fois en chaque sommet : en arrivant depuis le père et en allant vers le 1er fils, en revenant du 1er fils et en allant vers le 2e, . . ., et finalement après avoir visité le dernier fils et en remontant vers le père.

Chaque passage en un sommet est l'occasion d'effectuer un traitement : un affichage, un stockage, un décompte... ou rien du tout.

Un tel parcours se décrit particulièrement simplement de façon récursive.

Parcours en profondeur générique

```
def parcours_generique(racine) :
  L = liste_des_fils(racine) # éventuellement vide
  for i, noeud in enumerate(L) :
    traitement(i, racine)
    # = traitement après avoir visité i sous-arbres
    parcours_generique(noeud)
  traitement(len(L), racine)
```



```
def parcours_generique(racine) :
  L = liste_des_fils(racine) # éventuellement vide
  for i, noeud in enumerate(L) :
    traitement(i, racine)
    # = traitement après avoir visité i sous-arbres
    parcours_generique(noeud)
  traitement(len(L), racine)
```

Théorème

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

```
def parcours_generique(racine) :
  L = liste_des_fils(racine) # éventuellement vide
  for i, noeud in enumerate(L) :
    traitement(i, racine)
    # = traitement après avoir visité i sous-arbres
    parcours_generique(noeud)
  traitement(len(L), racine)
```

Théorème

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

- si `i = 0` : on parle de ***pré-traitement***
- si `i = len(L)` : on parle de ***post-traitement***

Trois cas particuliers (et particulièrement importants) :

Parcours préfixe : `traitement(i, racine)` vide sauf si `i = 0`

```
def parcours_prefixe(racine) :
  pre_traitement(racine)
  for noeud in liste_des_fils(racine) : parcours(noeud)
```

Parcours postfixe : `traitement(i, racine)` vide sauf si `i = len(L)`

```
def parcours_postfixe(racine) :
  for noeud in liste_des_fils(racine) : parcours(noeud)
  post_traitement(racine)
```

Parcours infixe : cas binaire avec seulement un traitement intermédiaire

```
def parcours_infixe(racine) :
  if racine != None : # vérification nécessaire car les appels récursifs ne l
    parcours(gauche(racine))
    traitement(racine)
    parcours(droite(racine))
```

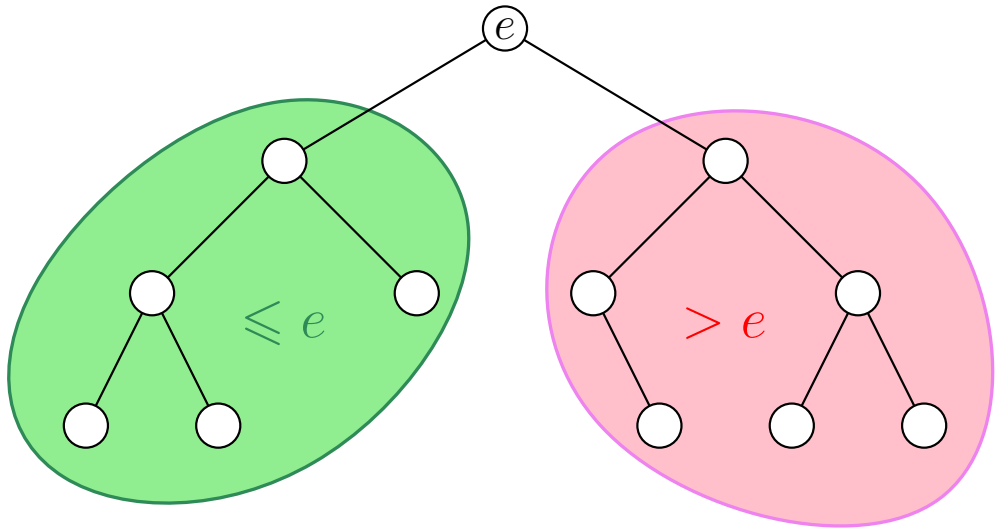
Arbres Binaires de Recherche

III. Définition et manipulations simples

Que peut bien signifier « trier » un arbre ?

Un arbre binaire de recherche (ABR) est un arbre binaire, étiqueté, tel que *l'étiquette de chaque sommet est comprise entre*

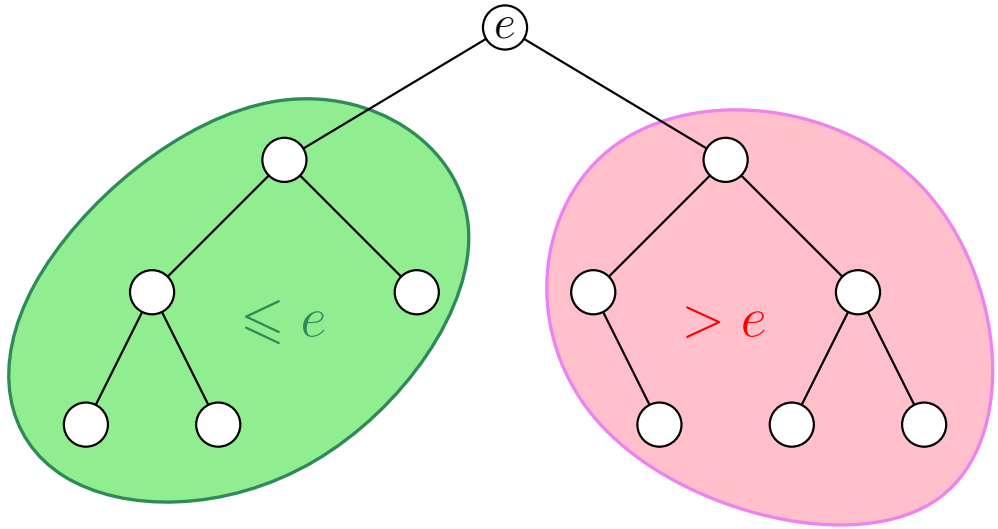
- *toutes les étiquettes du sous-arbre gauche (plus petites)* et
- *toutes les étiquettes du sous-arbre droit (plus grandes)*



Que peut bien signifier « trier » un arbre ?

Un arbre binaire de recherche (ABR) est un arbre binaire, étiqueté, tel que *l'étiquette de chaque sommet est comprise entre*

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

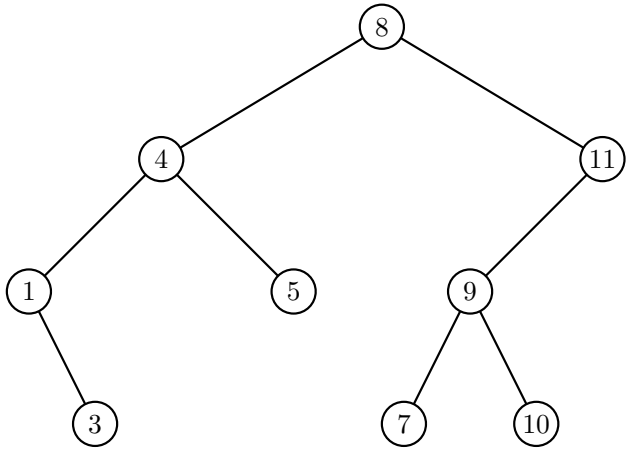


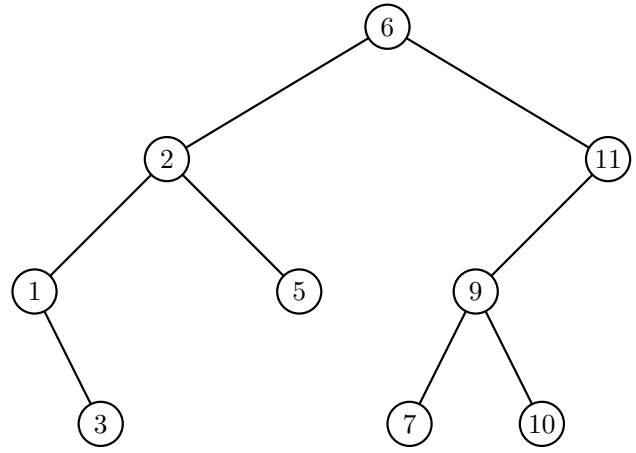
Attention ! ! !

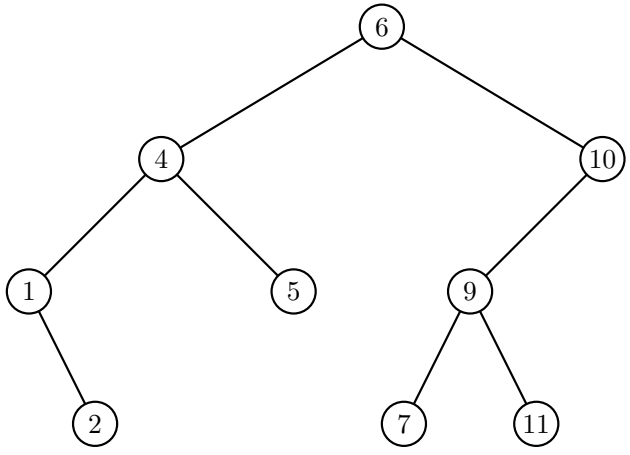
localement, *la définition entraîne* que chaque nœud a une étiquette comprise entre celle de son fils gauche et celle de son fils droit *mais ceci ne suffit pas*

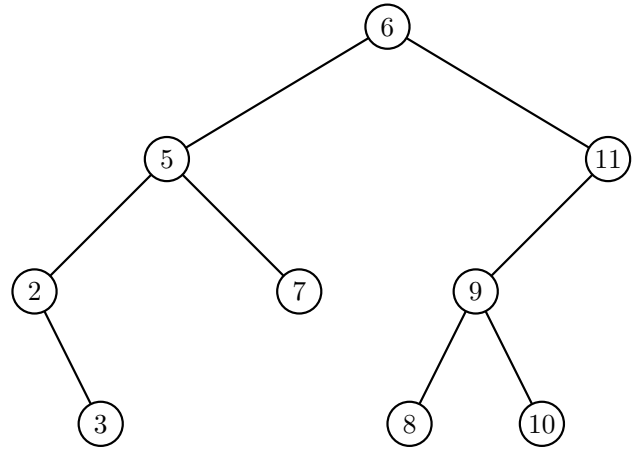
ABR or not ?

Vérifier qu'il n'y a aucun ABR parmi les exemples ci-dessous.



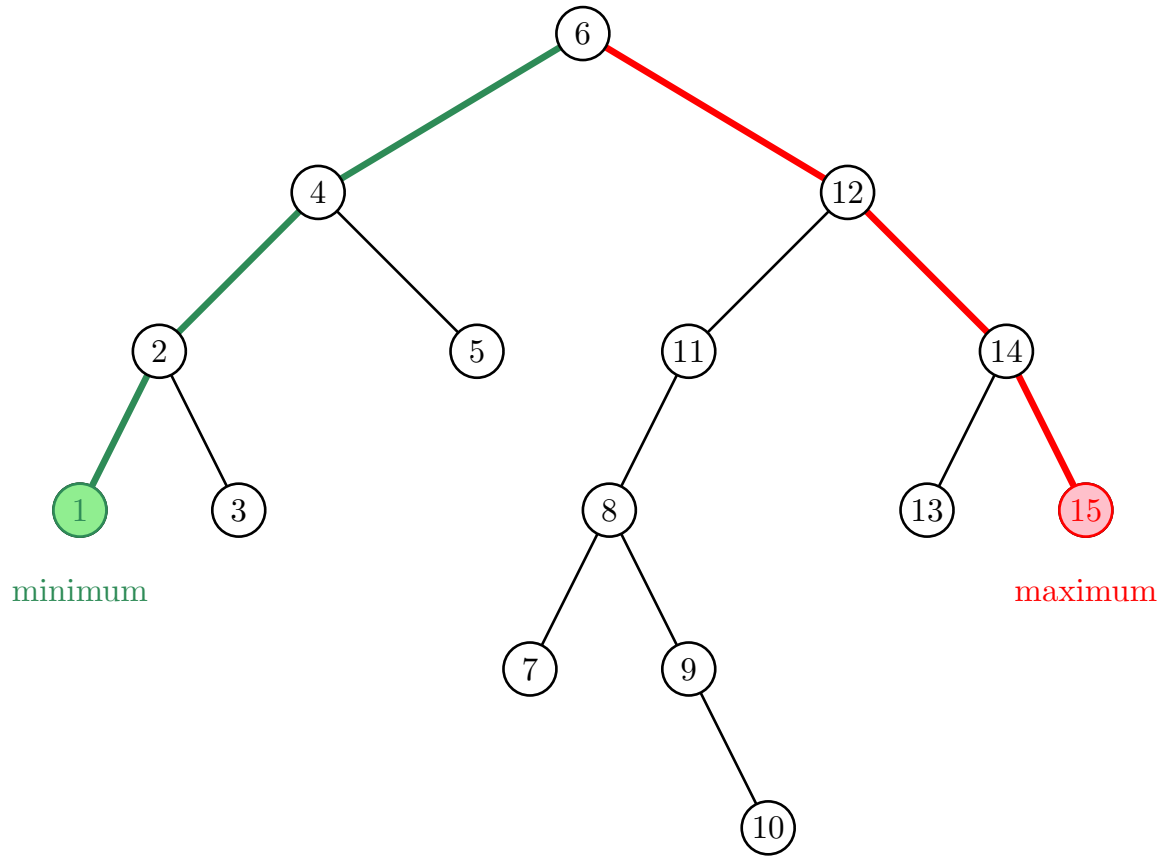






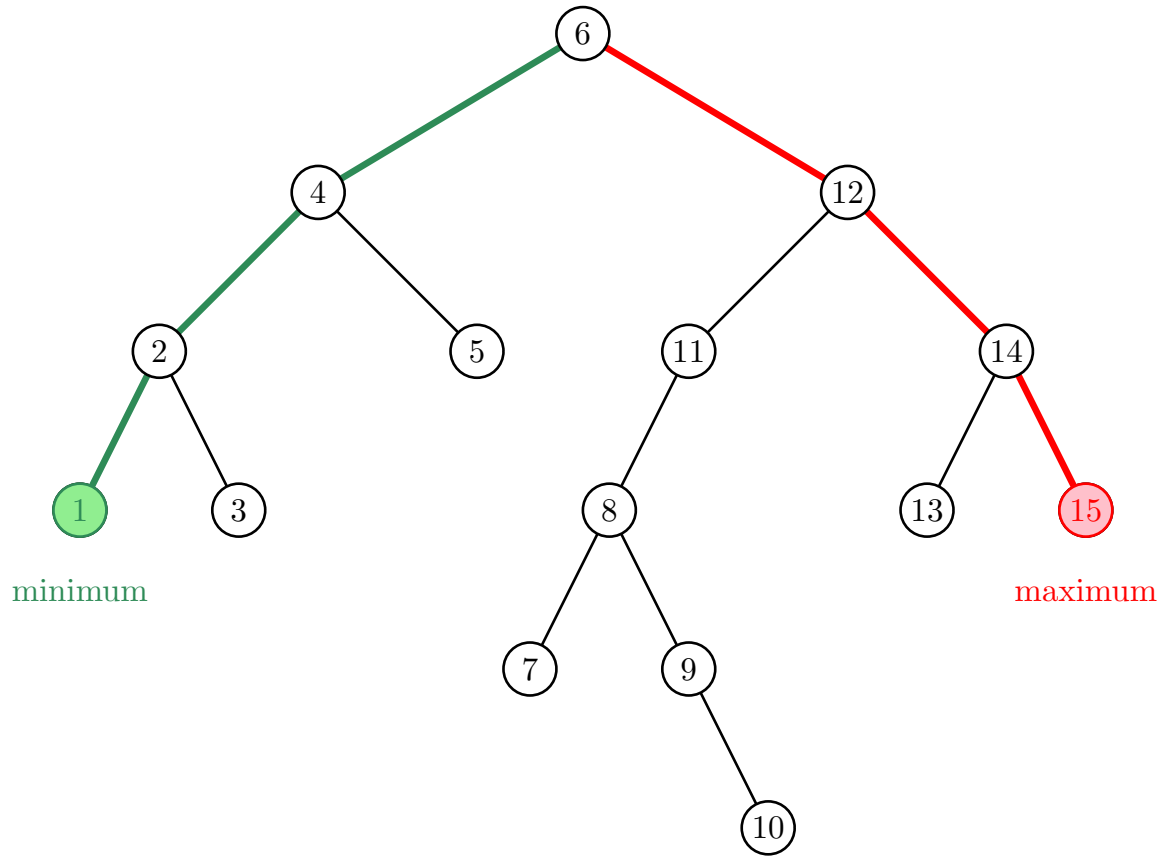
Minimum et maximum d'un ABR

La règle définissant les ABR rend certaines recherches faciles...



Minimum et maximum d'un ABR

La règle définissant les ABR rend certaines recherches faciles...



Plus généralement, la « disposition spatiale » des éléments suit leur ordre : plus un élément est à gauche, plus il est petit. L'élément de rang 2 de l'ABR est le minimum du sous-arbre droit du minimum (s'il en a un), ou son père sinon, etc.

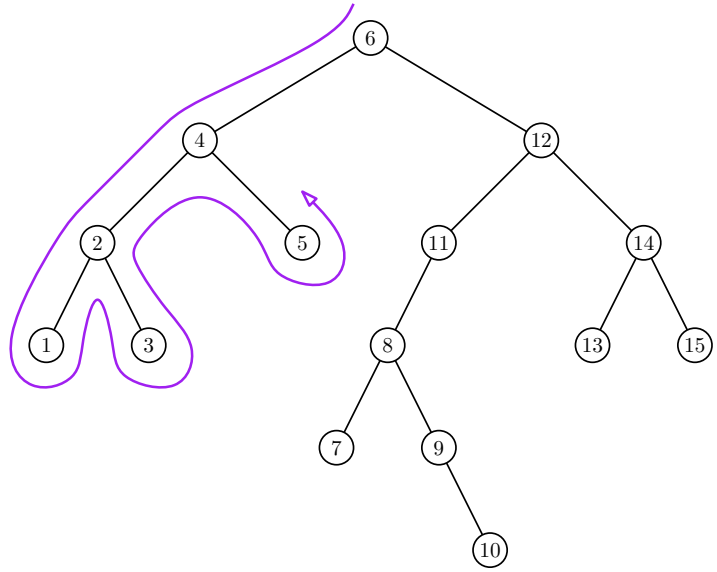
Ordre dans un ABR

Propriété : dans un ABR, chaque sous-arbre contient un intervalle de la liste triée des clés

Propriété : dans un ABR, chaque sous-arbre contient un intervalle de la liste triée des clés

Un ABR est donc « presque » une liste triée :

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



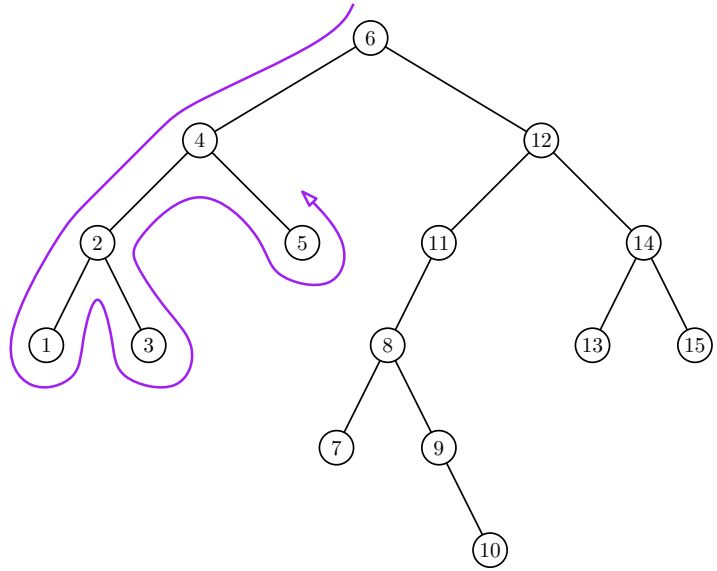
Théorème

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

Propriété : dans un ABR, chaque sous-arbre contient un intervalle de la liste triée des clés

Un ABR est donc « presque » une liste triée :

```
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 $\Theta(n)$.

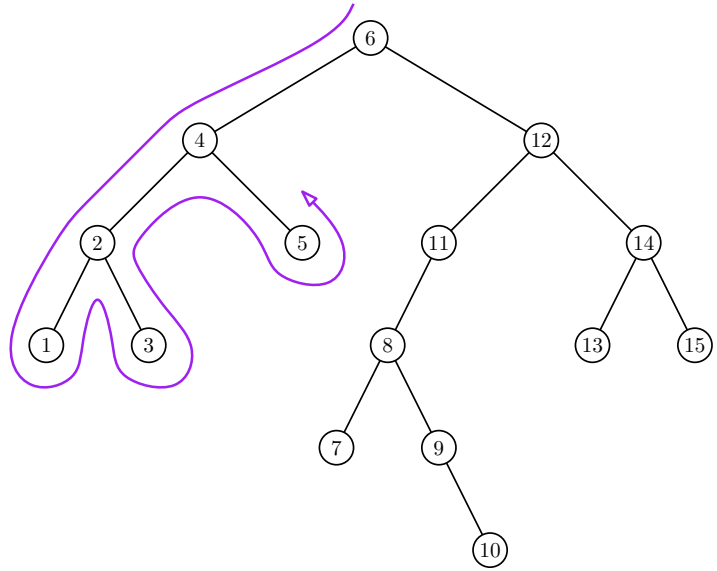
Corollaire

pour déterminer si un arbre est un ABR, il suffit de vérifier si son parcours infixe fournit bien une liste triée

Propriété : dans un ABR, chaque sous-arbre contient un intervalle de la liste triée des clés

Un ABR est donc « presque » une liste triée :

```
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 $\Theta(n)$.

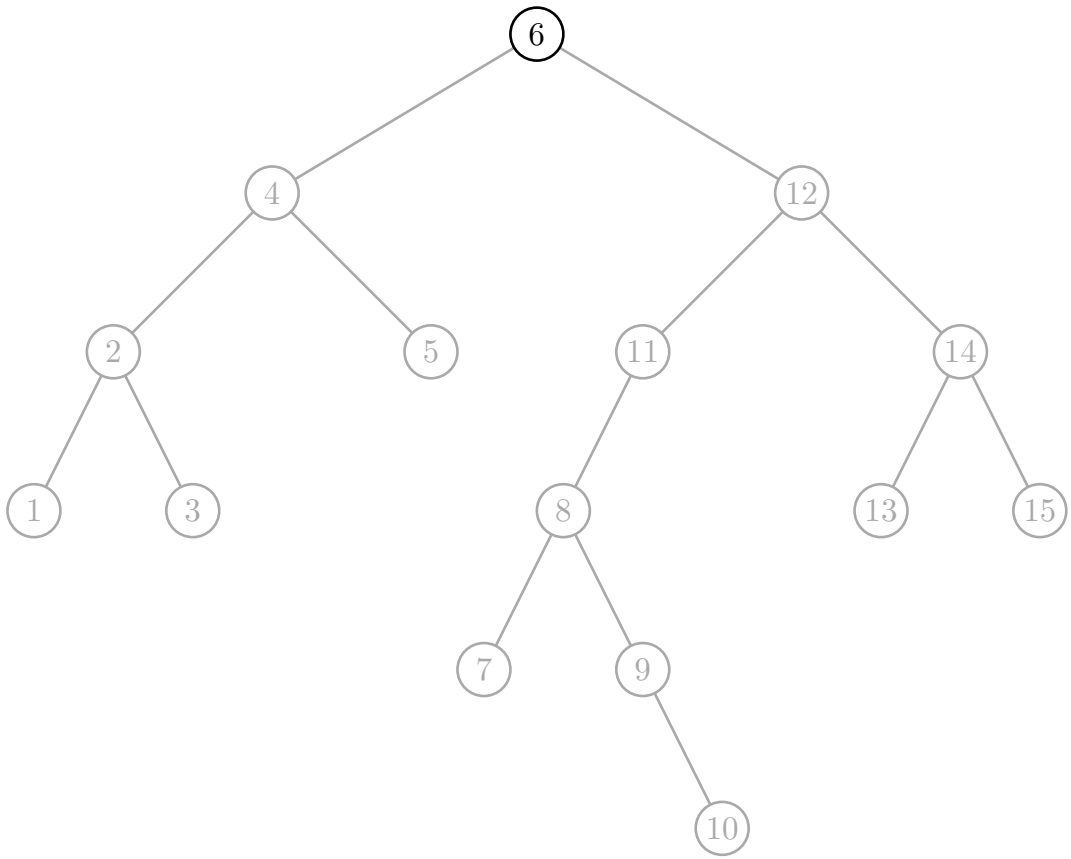
Corollaire

pour déterminer si un arbre est un ABR, il suffit de vérifier si son parcours infixe fournit bien une liste triée (qu'il n'est même pas nécessaire de construire)

Recherche dans un ABR

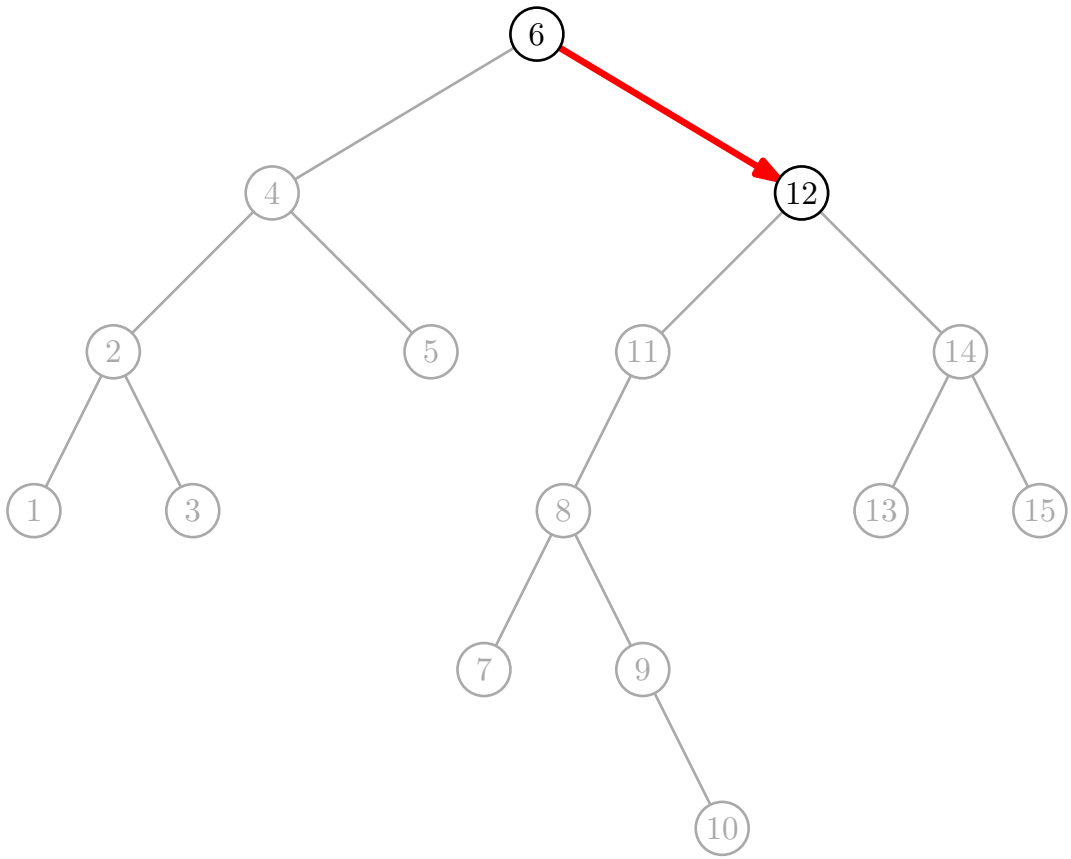
La définition des ABR permet également d'y faire des recherches à la manière de la recherche dichotomique : au lieu de comparer l'élément cherché avec l'élément médian du tableau, on le compare avec l'élément à la racine.

Exemple – recherche de 9 :



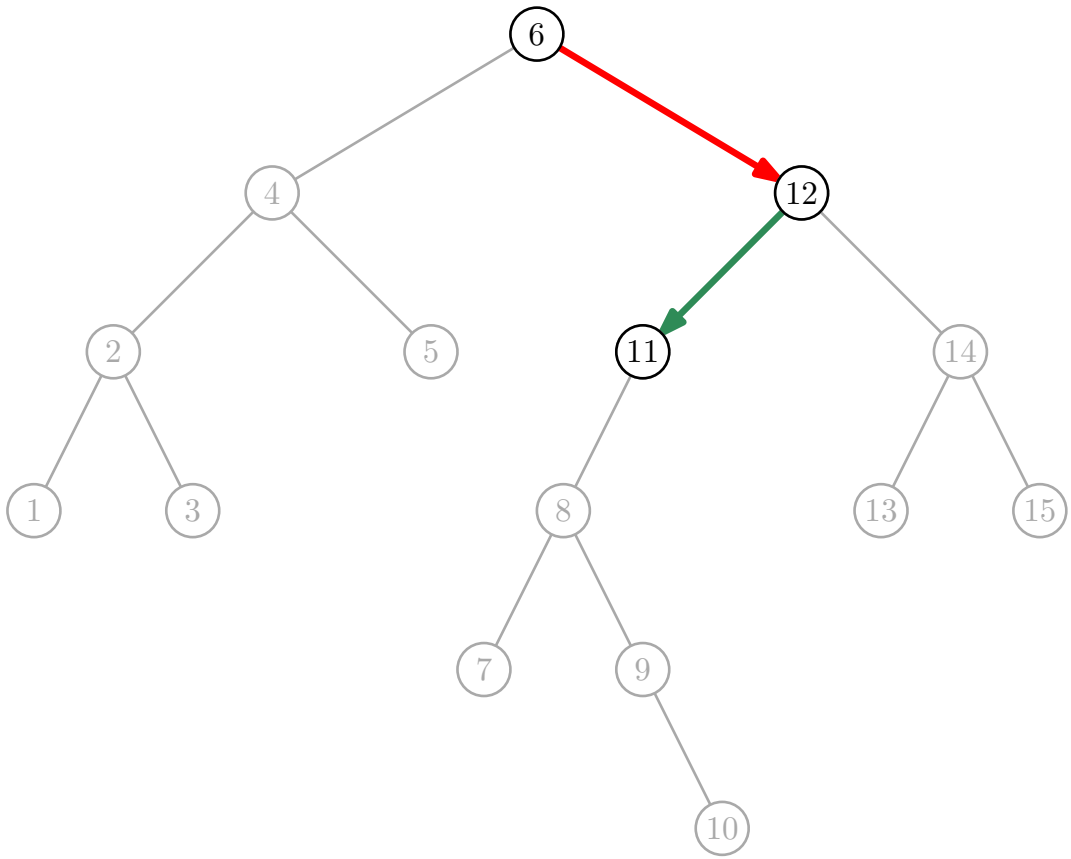
La définition des ABR permet également d'y faire des recherches à la manière de la recherche dichotomique : au lieu de comparer l'élément cherché avec l'élément médian du tableau, on le compare avec l'élément à la racine.

Exemple – recherche de 9 :



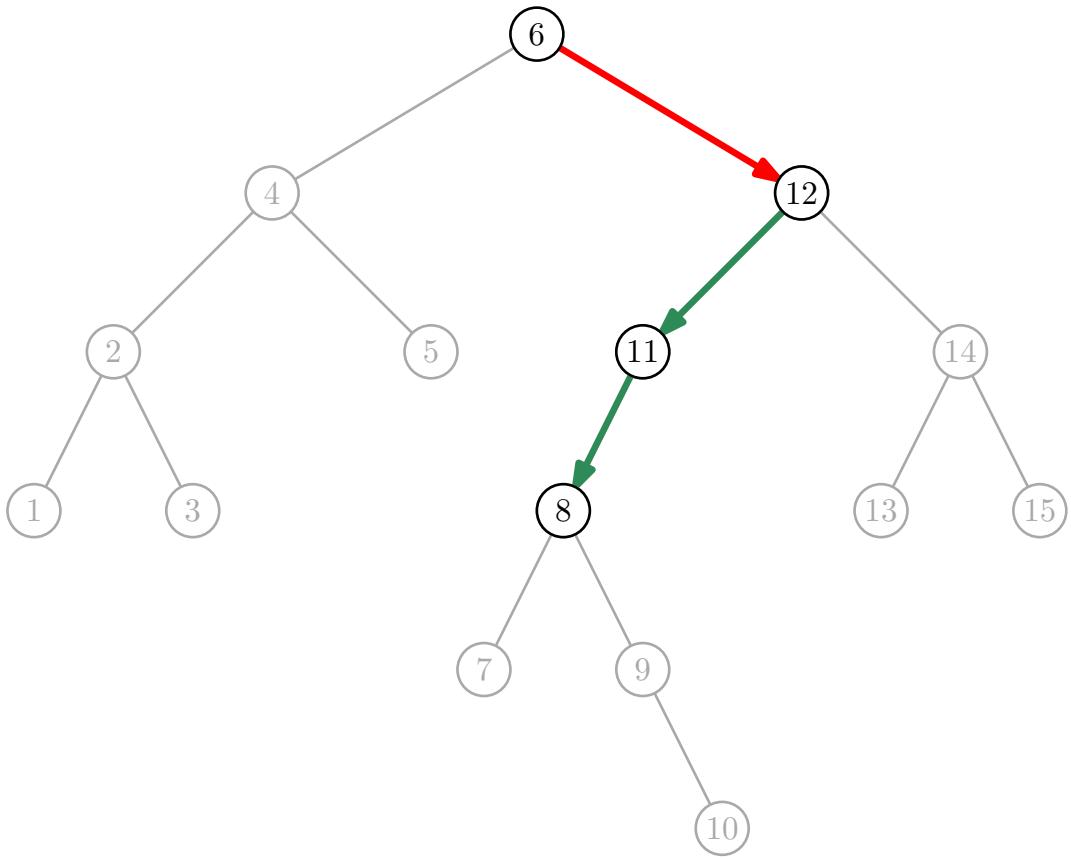
La définition des ABR permet également d'y faire des recherches à la manière de la recherche dichotomique : au lieu de comparer l'élément cherché avec l'élément médian du tableau, on le compare avec l'élément à la racine.

Exemple – recherche de 9 :



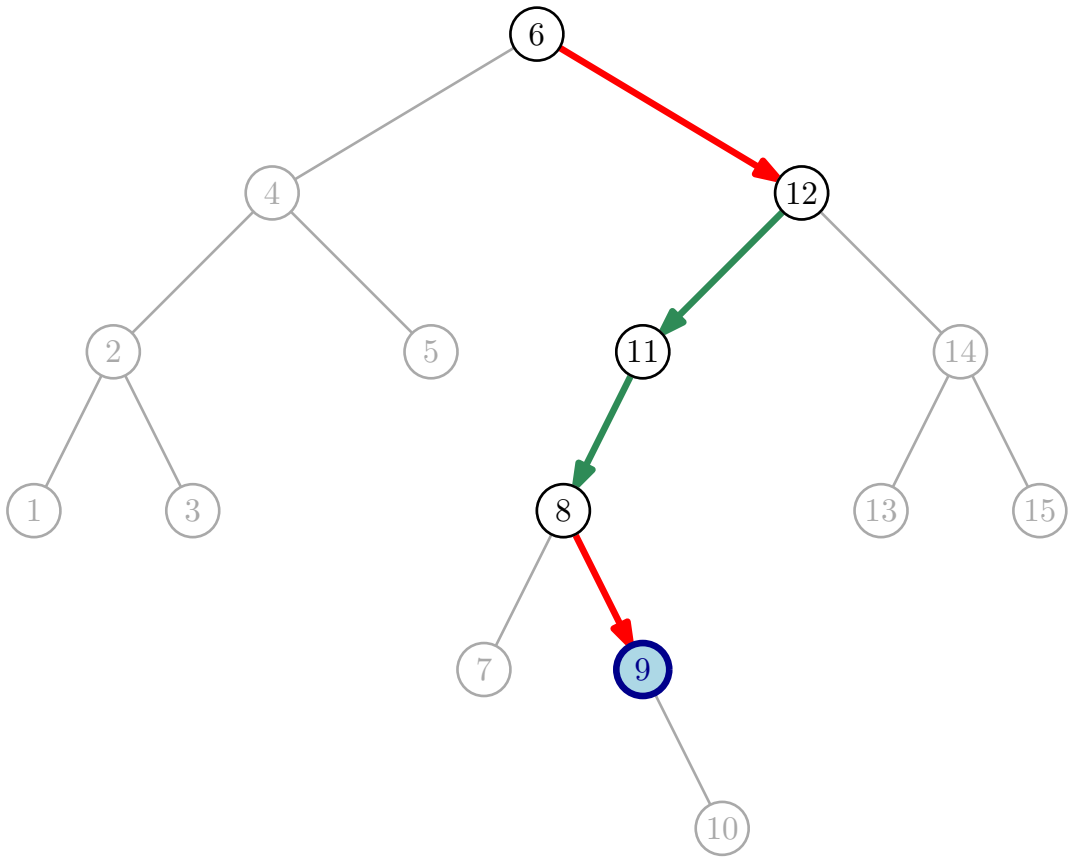
La définition des ABR permet également d'y faire des recherches à la manière de la recherche dichotomique : au lieu de comparer l'élément cherché avec l'élément médian du tableau, on le compare avec l'élément à la racine.

Exemple – recherche de 9 :



La définition des ABR permet également d'y faire des recherches à la manière de la recherche dichotomique : au lieu de comparer l'élément cherché avec l'élément médian du tableau, on le compare avec l'élément à la racine.

Exemple – recherche de 9 :



Cet algorithme s'écrit très simplement de manière récursive :

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

Cet algorithme s'écrit très simplement de manière récursive :

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

(et il est facile à dérécursiver, puisqu'il y a au plus un appel récursif, terminal)

Cet algorithme s'écrit très simplement de manière récursive :

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

(et il est facile à dérécursiver, puisqu'il y a au plus un appel récursif, terminal)

Comme pour la recherche dichotomique, toute une partie de l'ABR est laissée de côté à chaque appel – mais on ne sait pas *a priori* quelle proportion de l'ABR est parcourue, cela dépend fortement de la forme exacte de l'ABR.

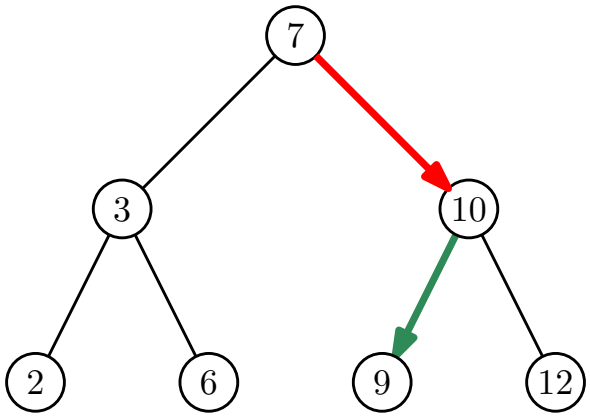
Théorème

`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 extrêmes

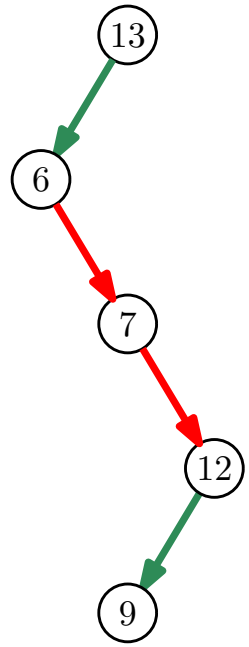
La recherche (et plus généralement tous les algorithmes sur les ABR) se comportent radicalement différemment selon la forme de l'arbre, qui détermine le rapport entre sa taille et sa hauteur.

Cas sympathique : ABR « parfait »



Hauteur $\log n$, recherche aussi efficace que la recherche dichotomique

Cas désagréable : ABR « filiforme »



Hauteur n, recherche aussi inefficace que dans une liste chaînée

```
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

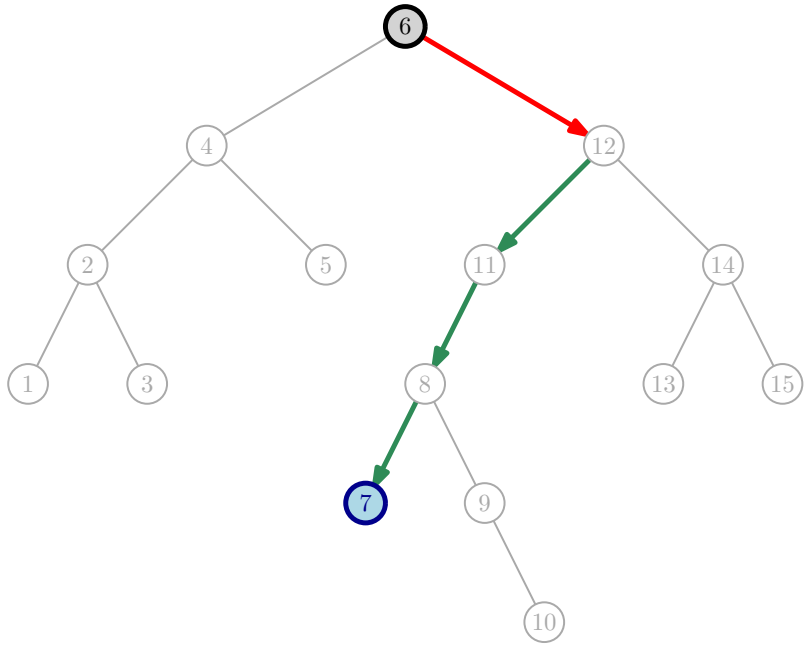
`minimum(r)` détermine le plus petit élément dans l'ABR de racine `r` en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

Un problème un peu plus compliqué :

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`.

Cas n° 1 : si le nœud a un fils droit

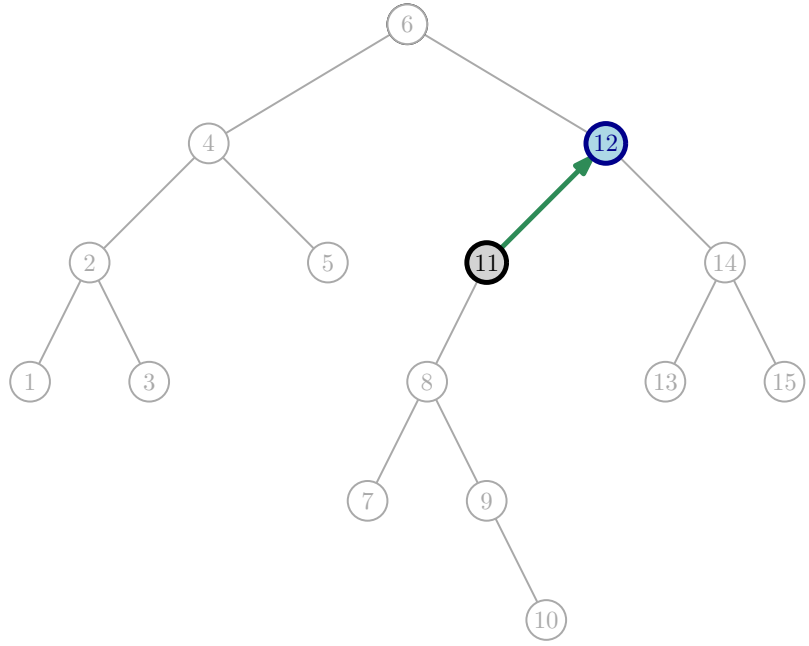


$\Longrightarrow$ le successeur est le minimum du sous-arbre droit

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`.

Cas n° 2 : si le nœud n'a pas de fils droit

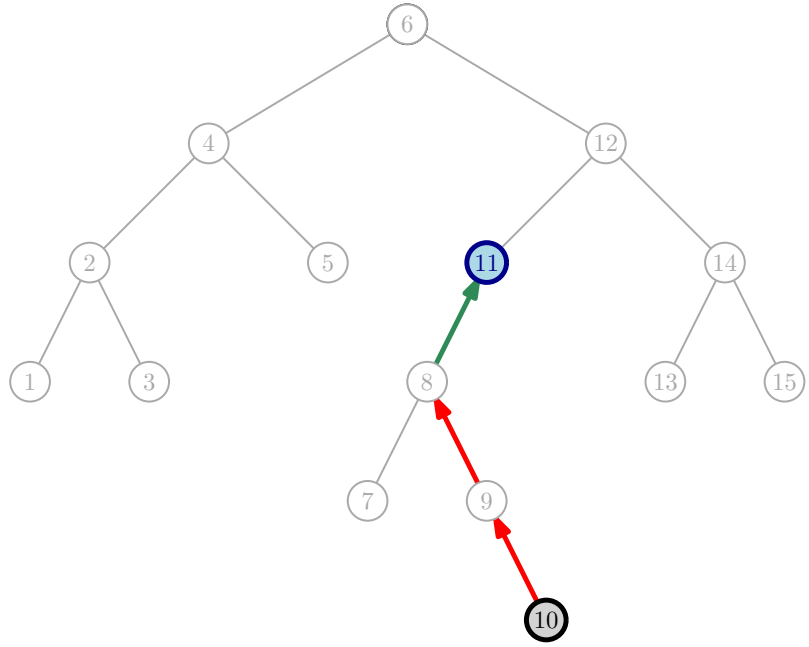


$\Longrightarrow$ le successeur est le premier ancêtre supérieur à l'élément – donc le premier vers lequel on remonte depuis la gauche

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`.

Cas n° 2 : si le nœud n'a pas de fils droit



$\Longrightarrow$ le successeur est le premier ancêtre supérieur à l'élément – donc le premier vers lequel on remonte depuis la gauche

Ce qui donne :

```
def successeur(noeud) :
  if droit(noeud) != None :
    return minimum(droit(noeud))
  while pere(noeud) != None and est_fils_droit(noeud) :
    noeud = pere(noeud)
  # soit pere(noeud) == None : noeud est la racine, le noeud
  # initial était le maximum, et n'a pas de successeur
  # soit noeud est un fils gauche : le successeur est son père
  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.

Arbres Binaires de Recherche

IV. Opérations de modification

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

1. ***l'insertion à la racine***, à la manière d'une insertion en tête de liste chaînée
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

1. ***l'insertion à la racine***, à la manière d'une insertion en tête de liste chaînée
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

Insertion à la racine : c'est peut-être le plus naturel, puisque la racine est le « point d'entrée » de l'arbre, comme la tête d'une liste chaînée [a] ; il s'agirait de créer une nouvelle racine contenant `x`, puis de reconstruire un sous-arbre gauche contenant tous les éléments plus petits que `x`, et un sous-arbre droit contenant tous les éléments plus grands que `x`.

Cela soulève deux problèmes :

- comment partitionner les éléments d'un ABR selon un « pivot » `x` ?
- comment reconstruire deux ABR avec, respectivement, les petits et les grands éléments ?

Non seulement ce n'est pas franchement évident (essayez à la main sur de petits exemples, vous verrez vite), mais en plus, il faudrait faire cela *efficacement* – c'est-à-dire sans parcourir tout l'arbre.

Moralité : essayons autrement !

---

a. et que l'insertion en tête de liste est plus simple que l'insertion en queue… n'est-ce-pas ? Si ce n'est pas clair, revoir les cours de L1 !

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

1. ~~*l'insertion à la racine*, à la manière d'une insertion en tête de liste chaînée~~
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

Insertion aux feuilles : créer une feuille à un emplacement libre, c'est-à-dire l'attacher à un nœud non complet

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

1. ~~*l'insertion à la racine*, à la manière d'une insertion en tête de liste chaînée~~
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

Insertion aux feuilles : créer une feuille à un emplacement libre, c'est-à-dire l'attacher à un nœud non complet

différence majeure aves les listes chaînées : non-unicité des emplacements libres

- $\Longrightarrow$ il faut trouver la bonne feuille,
- mais cela donne de la souplesse : insérer une feuille revient, en terme de liste, à insérer un élément à n'importe quelle position, ***sans effectuer de réorganisation majeure de la structure***

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

1. ~~*l'insertion à la racine*, à la manière d'une insertion en tête de liste chaînée~~
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

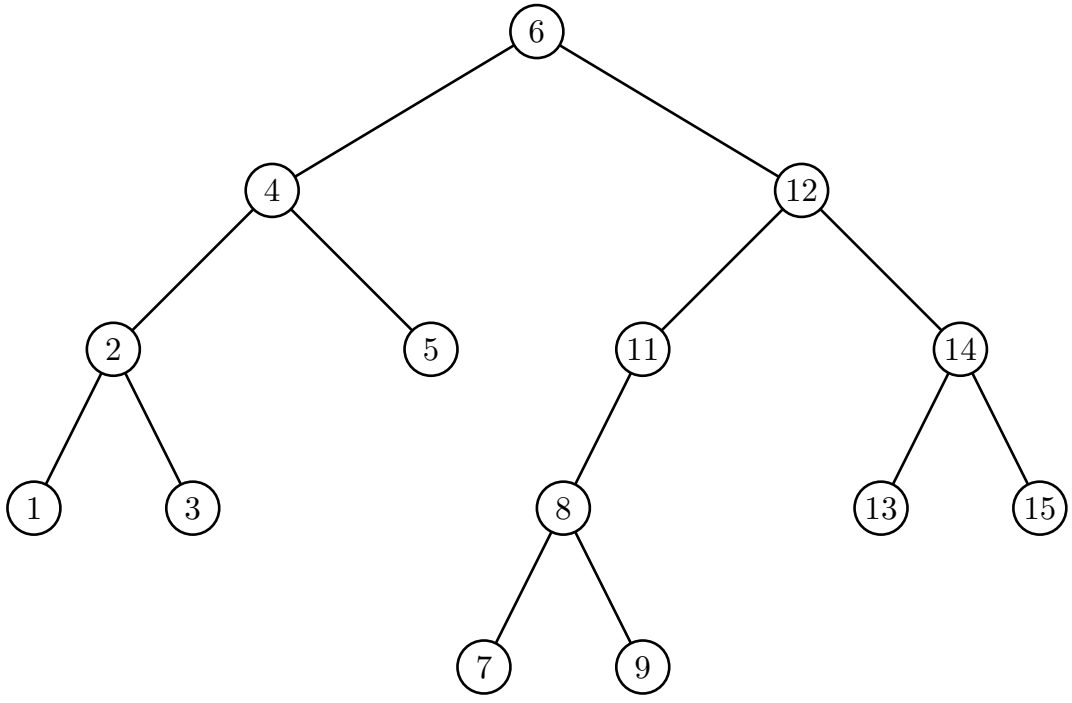
Insertion aux feuilles : créer une feuille à un emplacement libre, c'est-à-dire l'attacher à un nœud non complet

différence majeure aves les listes chaînées : non-unicité des emplacements libres

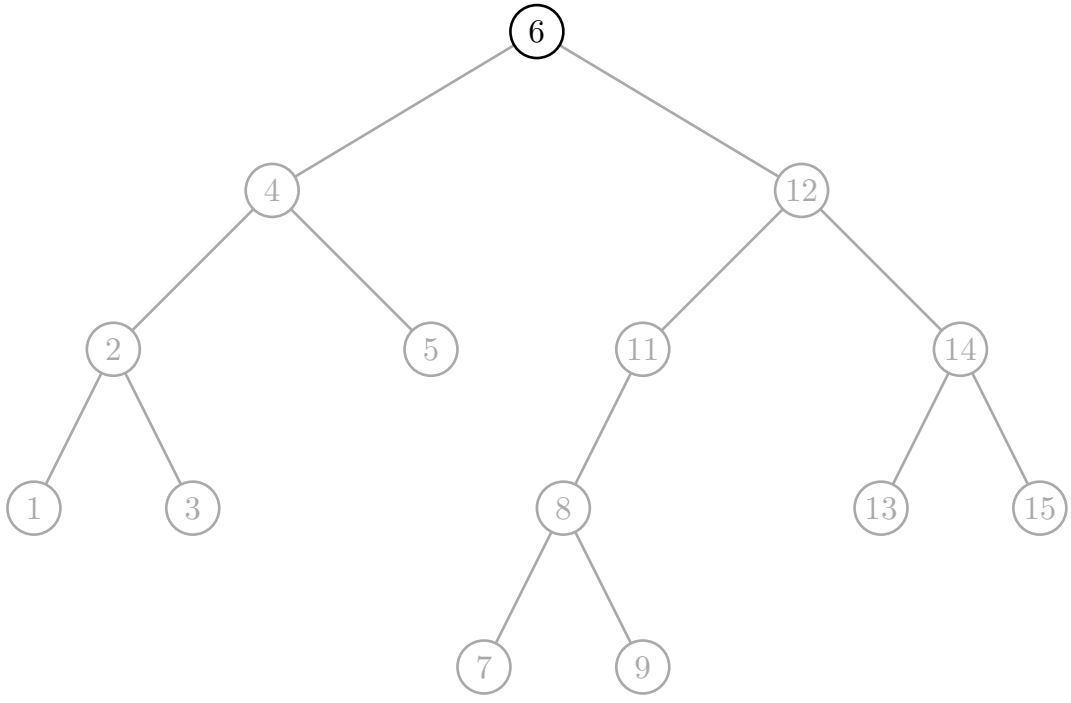
- $\Longrightarrow$ il faut trouver la bonne feuille,
- mais cela donne de la souplesse : insérer une feuille revient, en terme de liste, à insérer un élément à n'importe quelle position, ***sans effectuer de réorganisation majeure de la structure***

Autre point positif : cela permet d'éviter les doublons en faisant précéder toute insertion d'une recherche. Si cette recherche réussit, `x` ne doit pas être inséré ; et si elle échoue... nous avons trouvé l'***unique emplacement*** où `x` aurait pu se trouver, c'est-à-dire l'emplacement où il faut créer une nouvelle feuille pour l'accueillir.

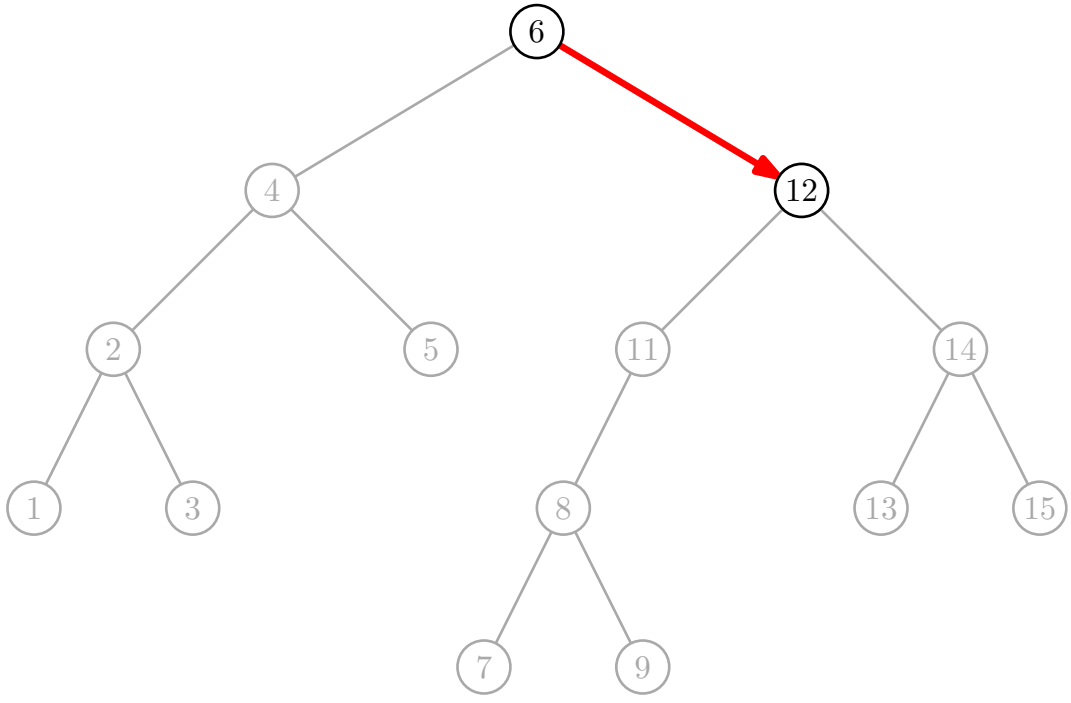
Exemple – insertion de 10 :



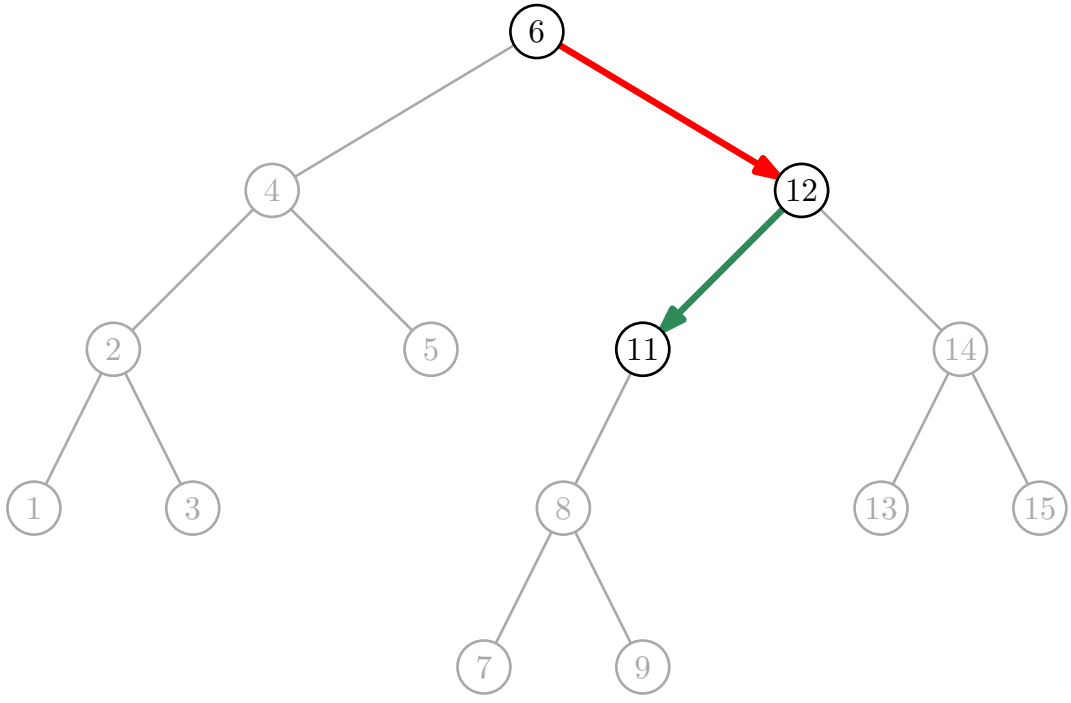
Exemple – insertion de 10 :



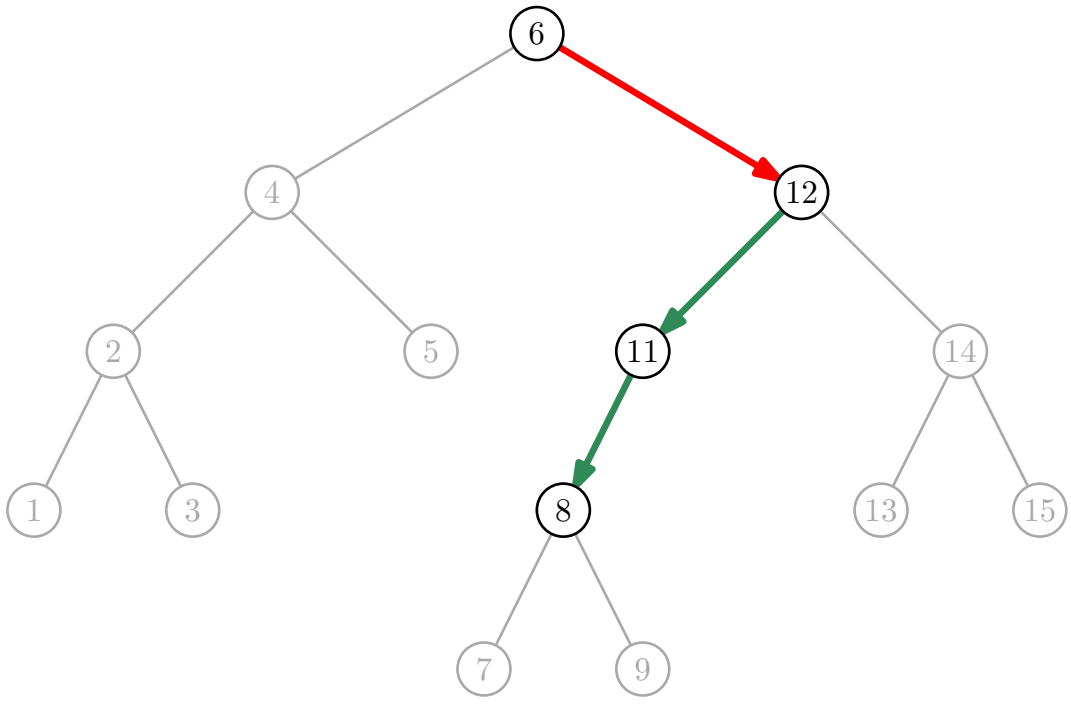
Exemple – insertion de 10 :



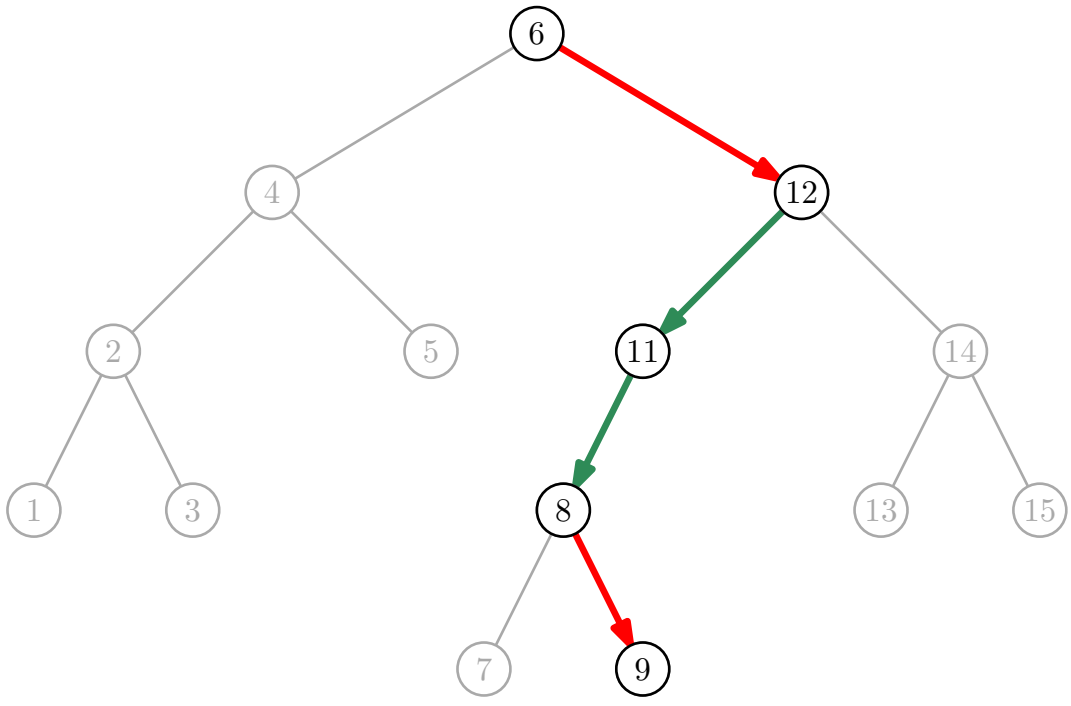
Exemple – insertion de 10 :



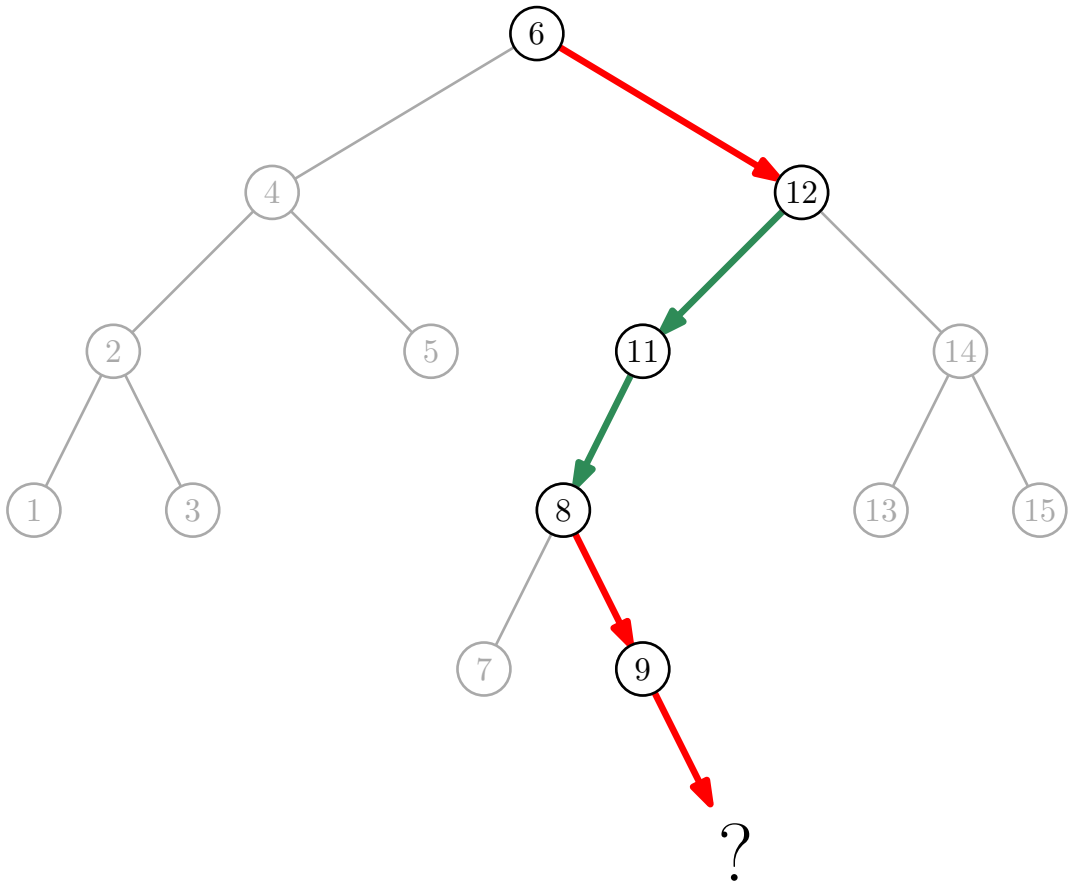
Exemple – insertion de 10 :



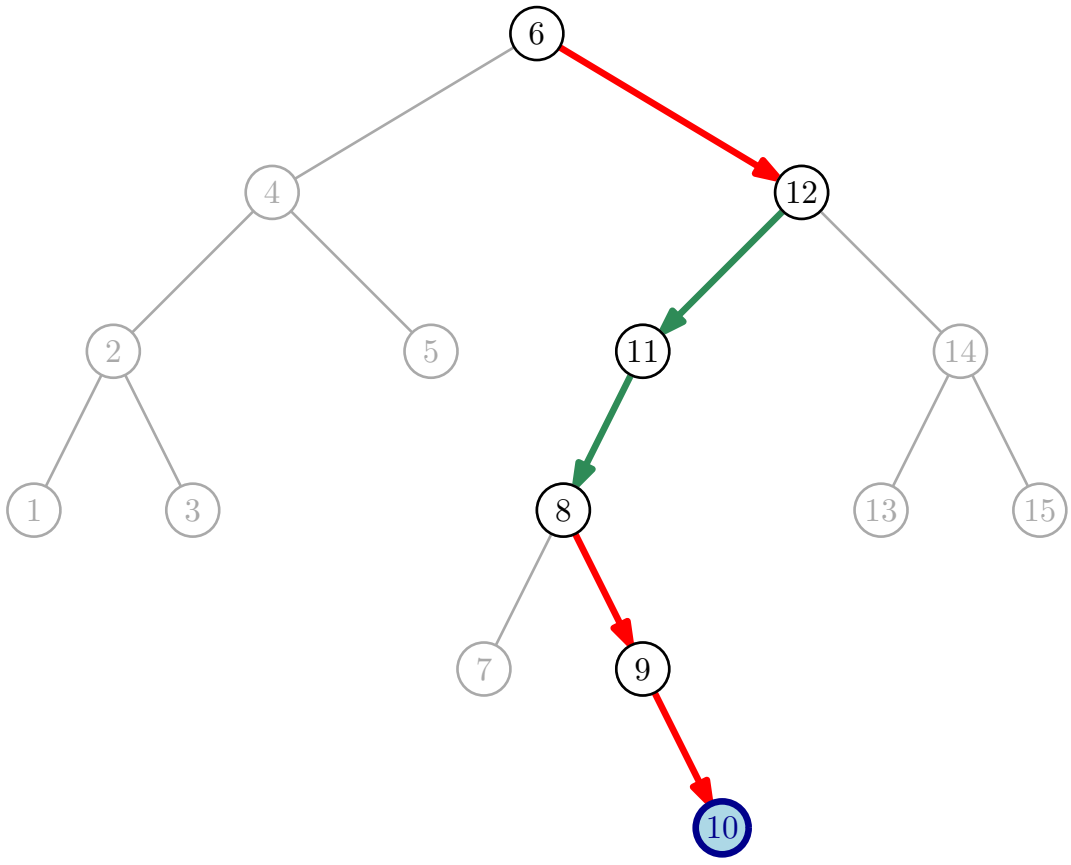
Exemple – insertion de 10 :



Exemple – insertion de 10 :



Exemple – insertion de 10 :



Puisque l'algorithme d'insertion n'est finalement qu'une petite modification de l'algorithme de recherche, sa complexité est exactement la même :

Théorème

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

Puisque l'algorithme d'insertion n'est finalement qu'une petite modification de l'algorithme de recherche, sa complexité est exactement la même :

Théorème

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

Une petite remarque concernant la création de la nouvelle feuille :

- *si l'ABR est représenté uniquement par ses « vrais » sommets, ceux qui contiennent un élément, il faut faire la recherche en ayant constamment un pas de retard, sinon au moment où on arrive « dans » l'absence de feuille, on n'a pas de moyen d'accrocher la nouvelle feuille à son père.*
- *si l'ABR est complété, c'est beaucoup plus simple : en arrivant dans la feuille sans contenu, il suffit de la remplir, et de lui créer deux enfants vides.*

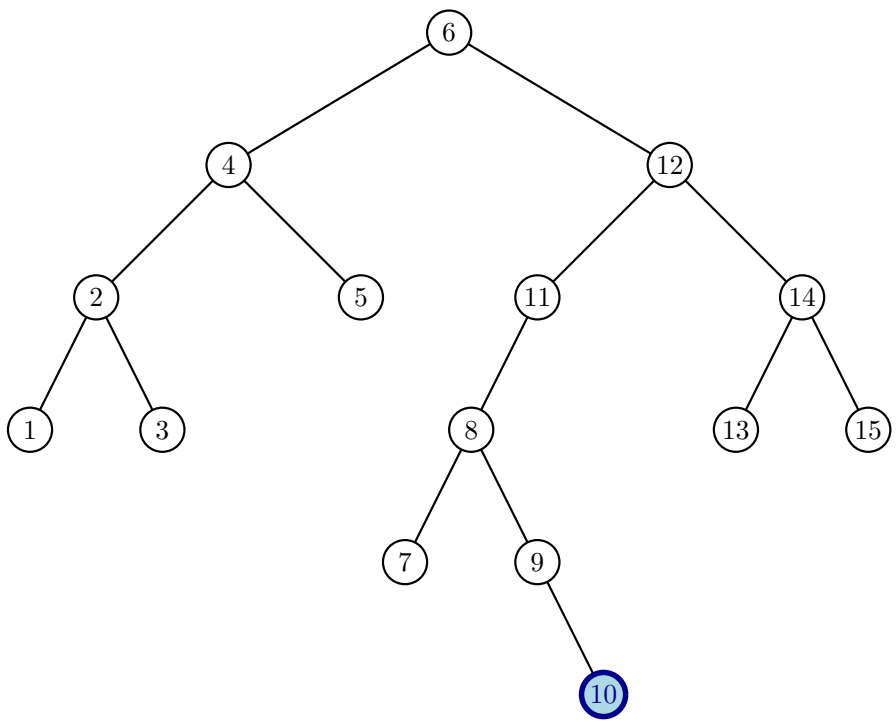
Suppression dans un ABR

La suppression d'un élément x est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque x est contenu dans un nœud vraiment binaire.

La suppression d'un élément x est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque x est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève

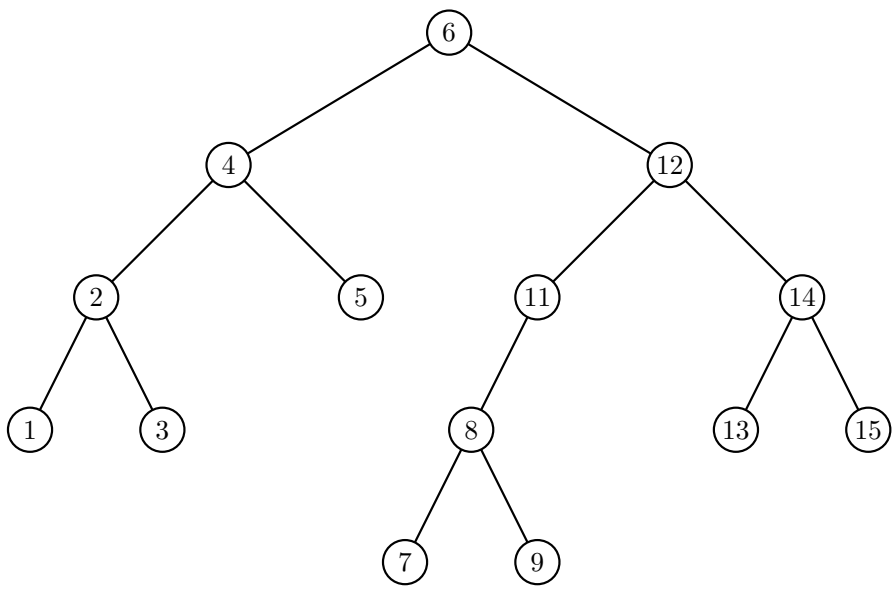


Suppression dans un ABR

La suppression d'un élément x est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque x est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève

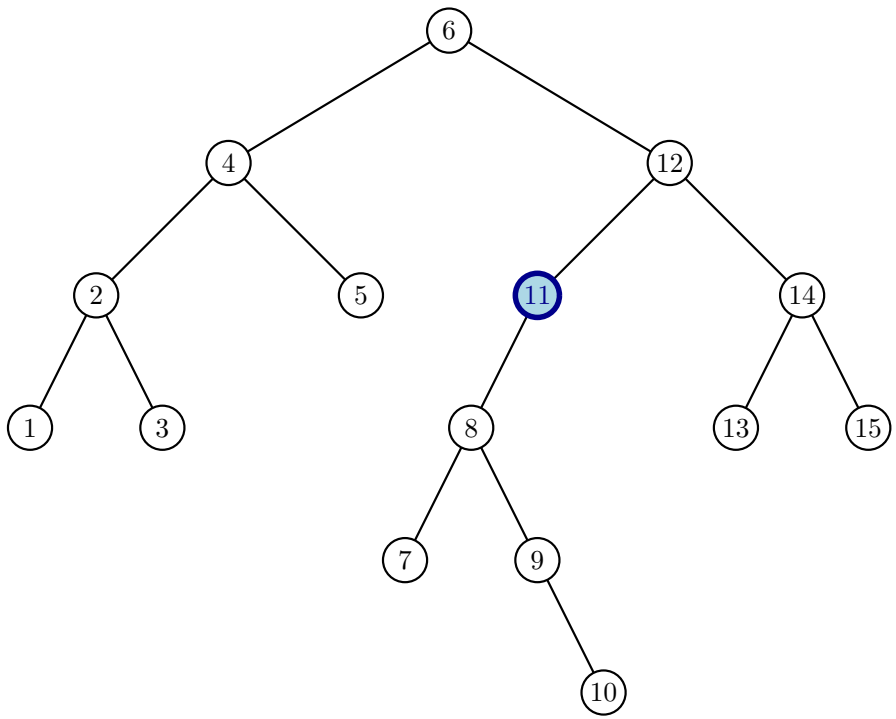


Suppression dans un ABR

La suppression d'un élément x est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque x est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève
2. si le nœud à supprimer n'a qu'un enfant : on « remonte » son unique enfant

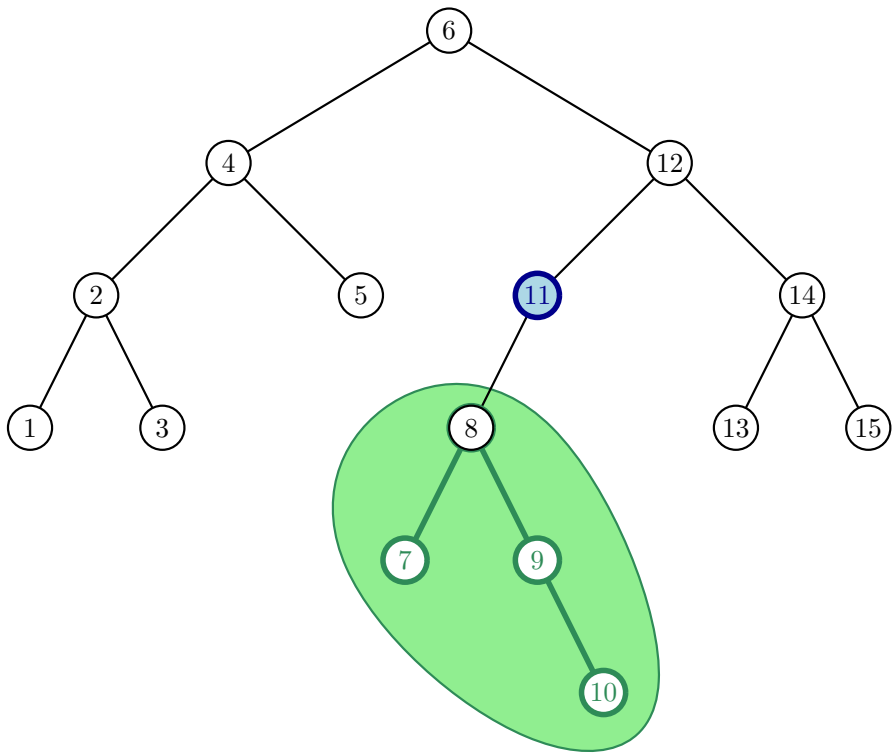


Suppression dans un ABR

La suppression d'un élément x est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque x est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève
2. si le nœud à supprimer n'a qu'un enfant : on « remonte » son unique enfant

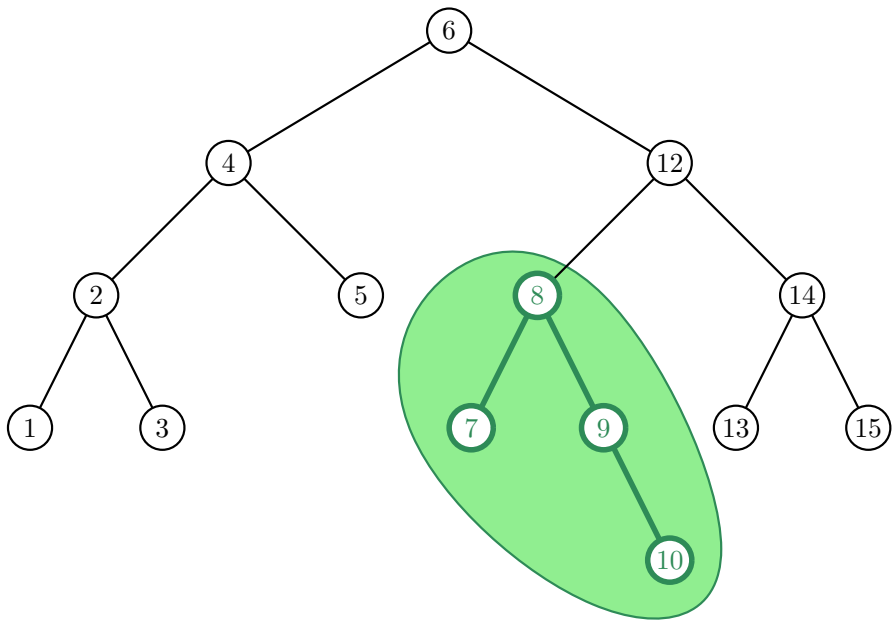


Suppression dans un ABR

La suppression d'un élément x est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque x est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève
2. si le nœud à supprimer n'a qu'un enfant : on « remonte » son unique enfant



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ;*

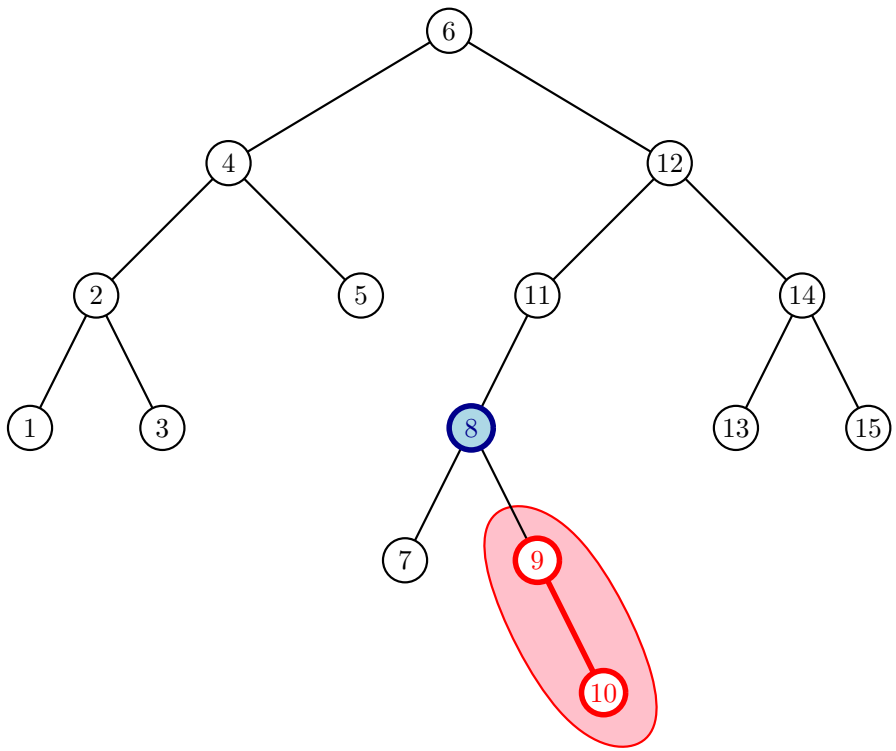
Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

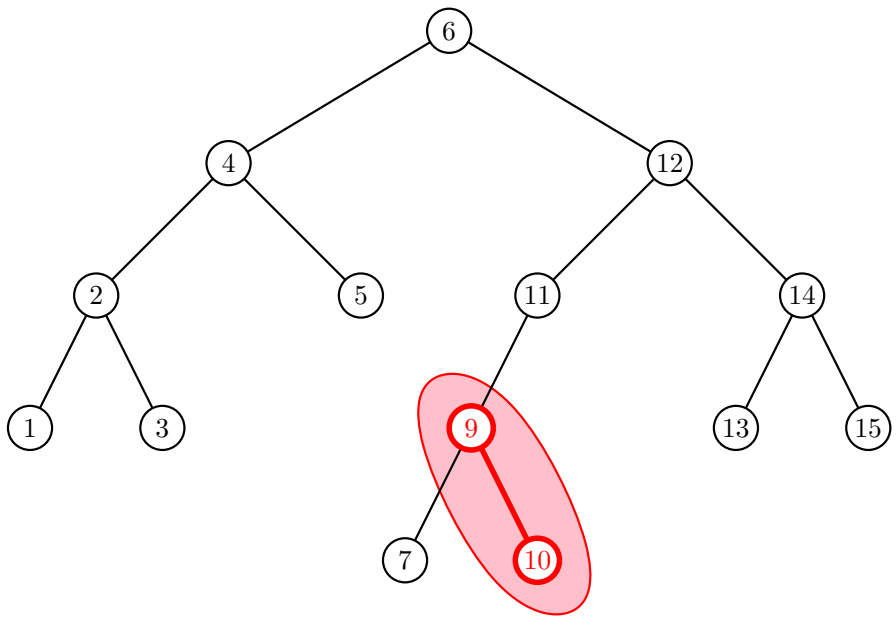
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

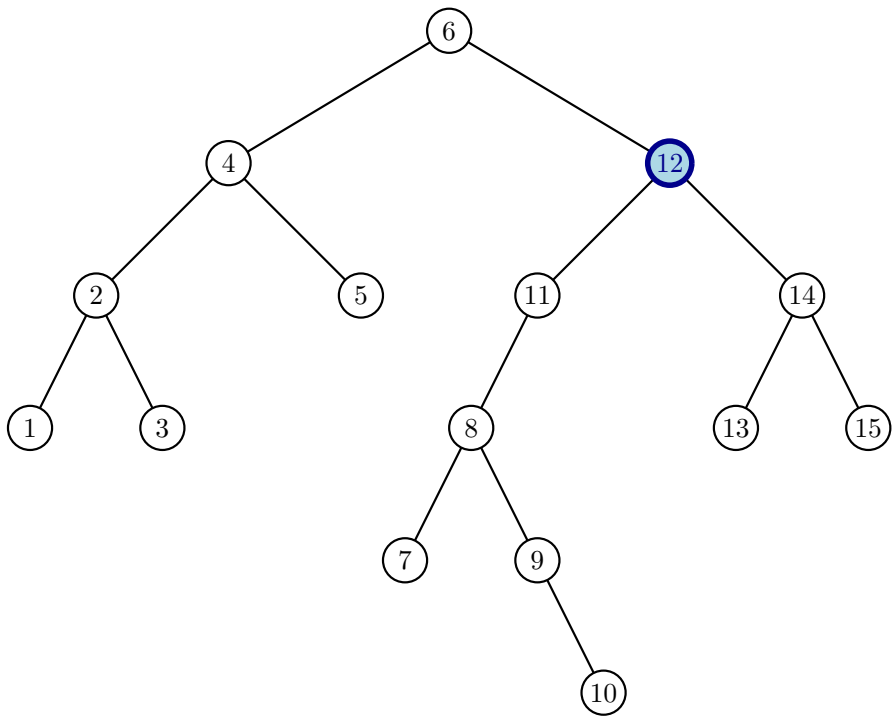
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

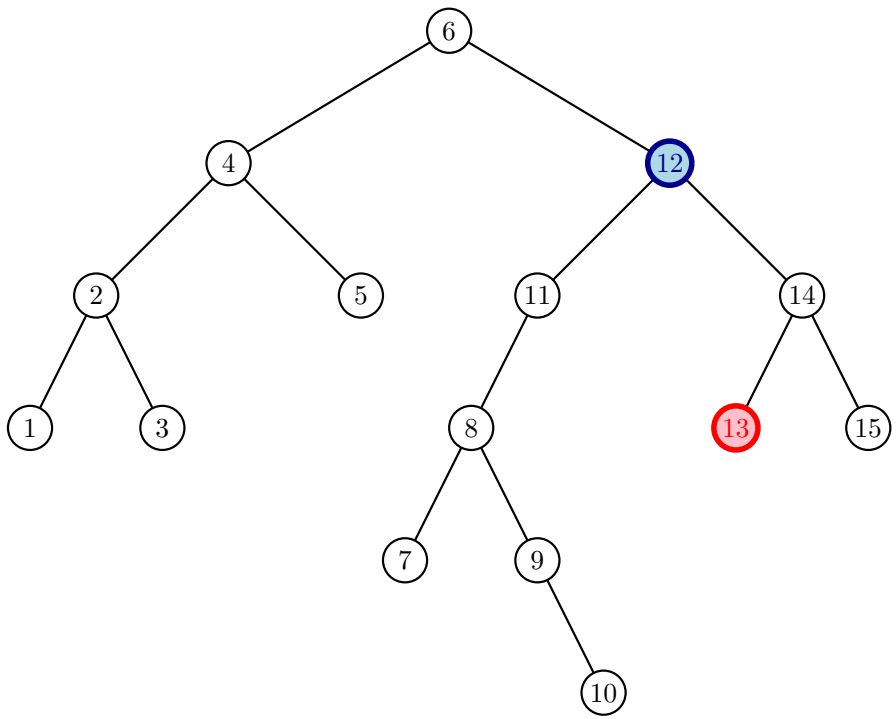
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

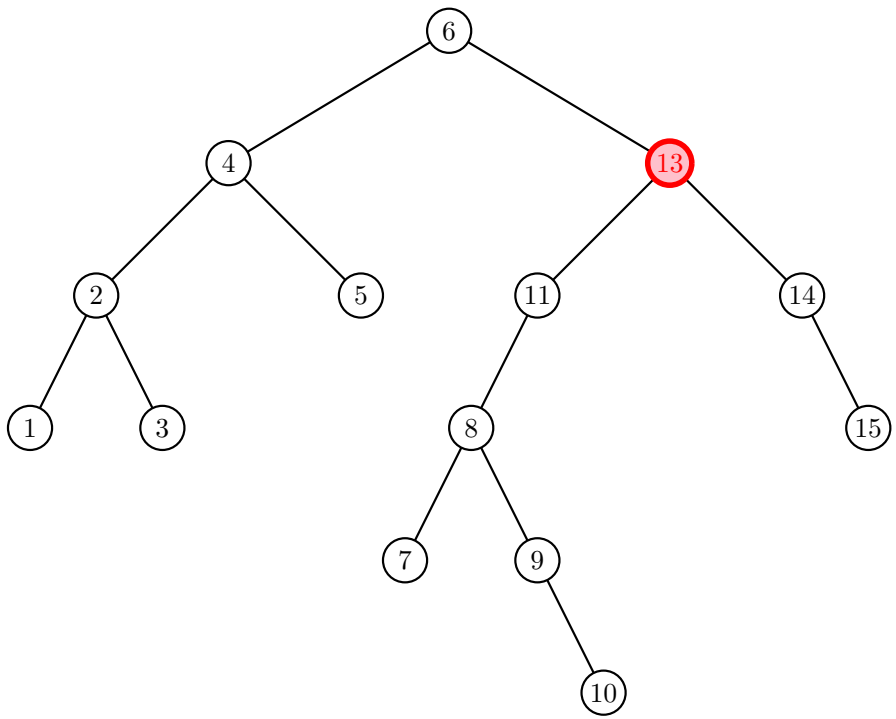
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

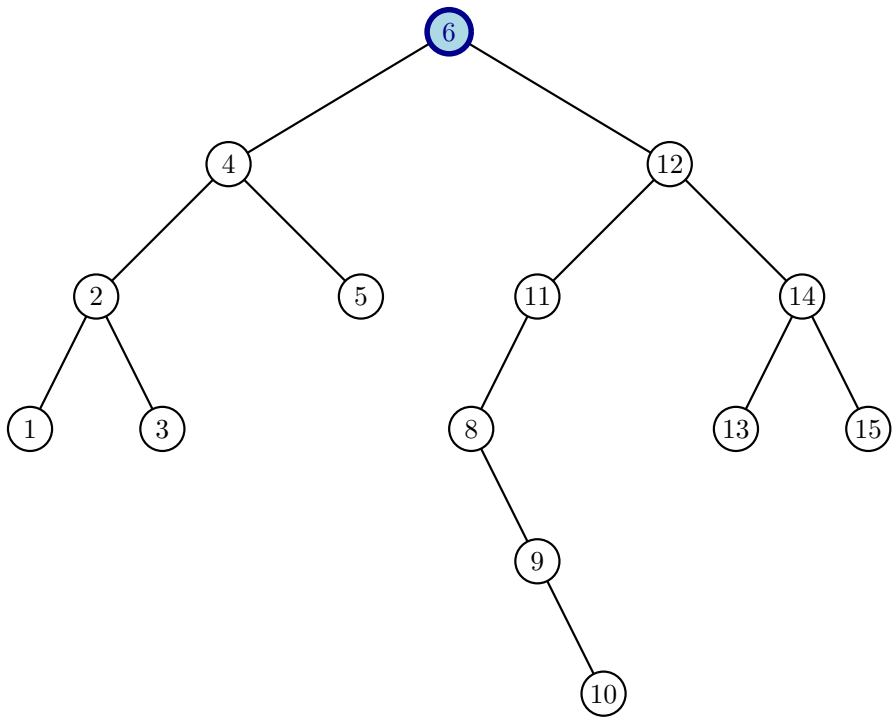
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

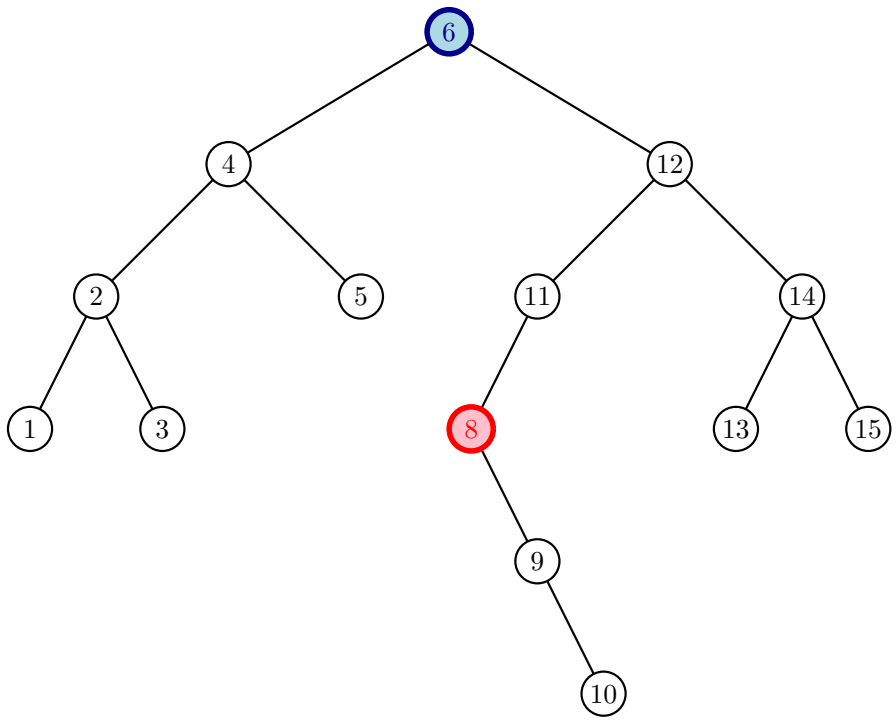
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

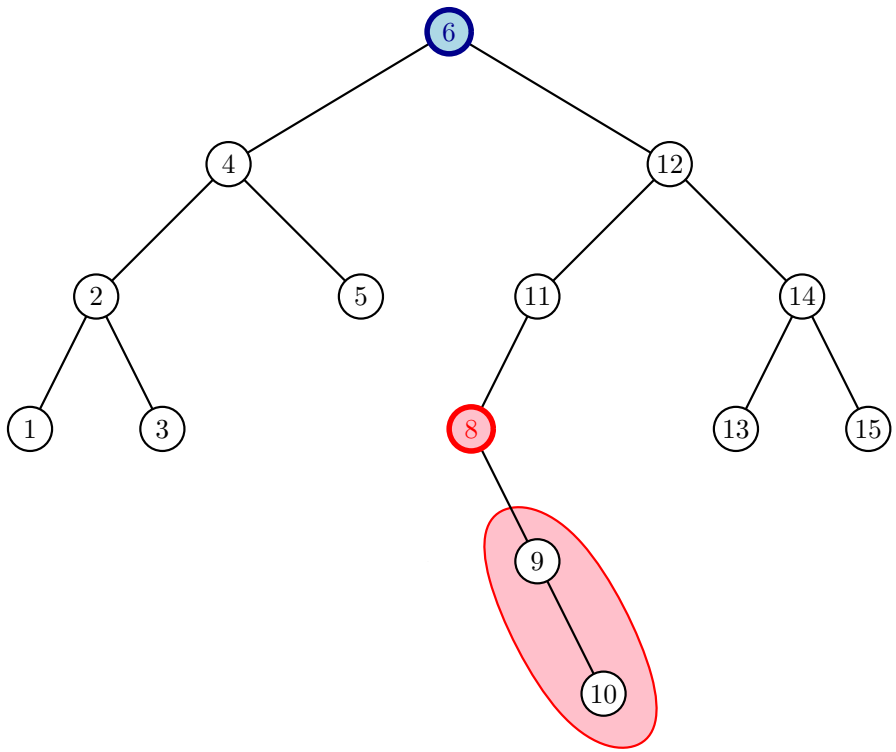
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

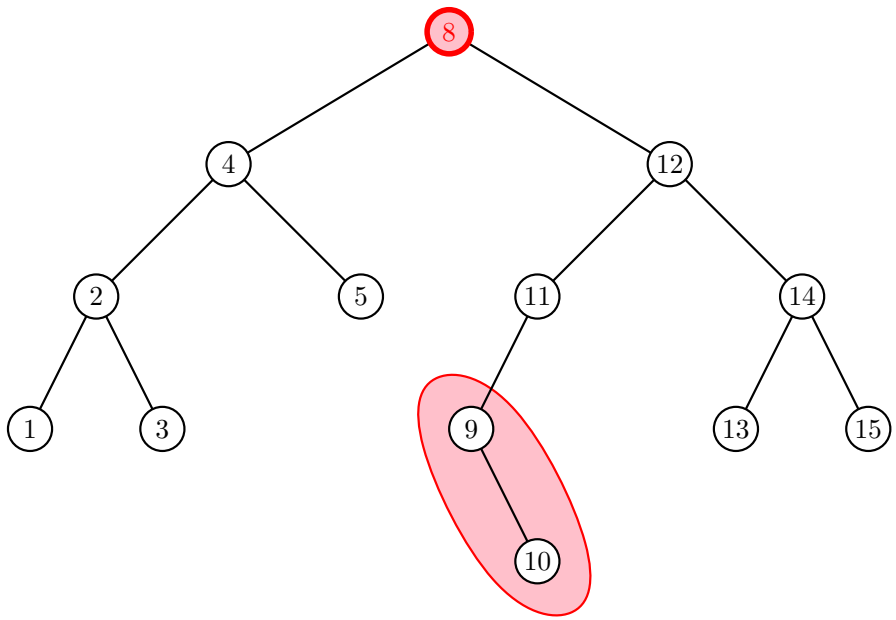
Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

Les deux cas sont symétriques – remplaçons `x` par son successeur.



Sinon(cas générique où le nœud contenant `x` a deux enfants) : *le nœud ne peut pas être supprimé, il faut trouver un autre élément qui puisse prendre sa place ; deux candidats, le prédécesseur et le successeur de* x *: ce sont les seuls éléments qui sont :*

- *plus petits que tous les (autres) éléments plus grands que* x
- *plus grands que tous les (autres) éléments plus petits que* x

Les deux cas sont symétriques – remplaçons `x` par son successeur.

On remarque en fait la propriété suivante :

Lemme

Le successeur d'un nœud à deux enfants n'a lui-même pas de fils gauche.

(forcément, puisque dans ce cas, il s'agit du minimum du sous-arbre droit...)

Donc en résumé :

1. cas d'une feuille : suppression simple
2. cas d'un nœud à un seul fils : l'autre fils remonte d'un niveau
3. cas où le successeur est le fils droit : le fils droit remonte d'un niveau et adopte son frère
4. autres cas : le noeud est remplacé par son successeur, dont l'unique fils (droit) remonte d'un niveau

Remarques :

- le point 3 n'est qu'un cas particulier du point 4
- la même manipulation peut être faite avec le prédécesseur plutôt que le successeur

Et donc, comme la recherche du successeur (ou du prédécesseur) est de complexité $\Theta(h)$ au pire :

Théorème

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

Donc :

Théorème

la liste triée des éléments d'un ABR de taille n peut être obtenue en temps $\Theta(n)$ par un parcours en profondeur infixe.

Théorème

la recherche, l'ajout et la suppression d'un élément dans un ABR de hauteur h se font en temps $\Theta(h)$ au pire.

Corollaire

la construction d'un ABR de taille n par insertion successive de ses éléments a un coût $O(nh)$, si h est la hauteur de l'arbre obtenu.

La clé de l'efficacité de ces opérations réside donc dans la ***hauteur*** de l'ABR considéré en fonction de sa taille.