Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

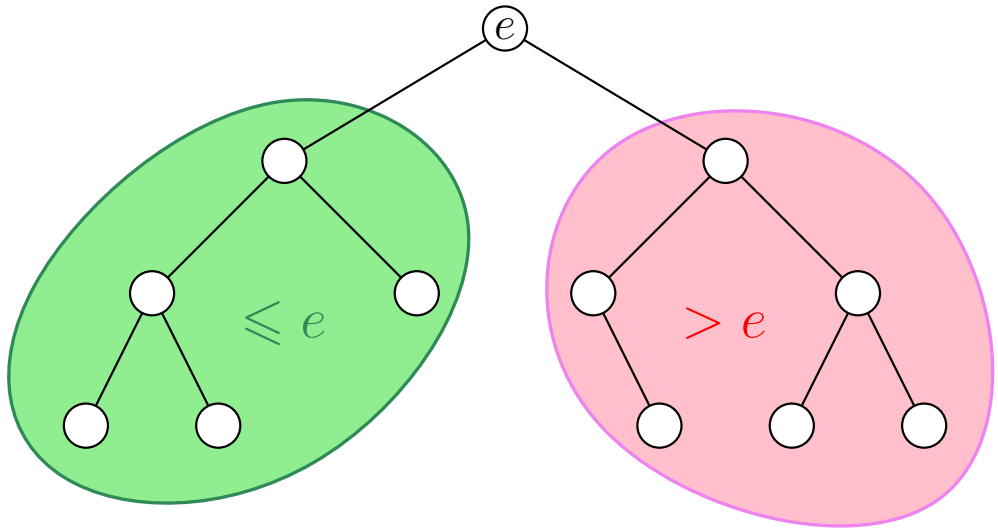
Arbres Binaires de Recherche

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)*

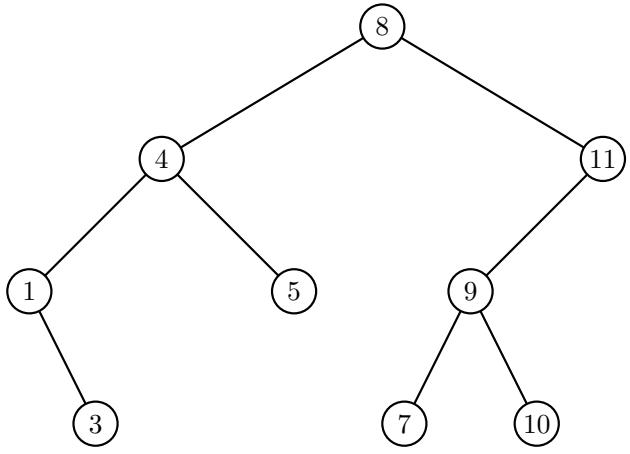


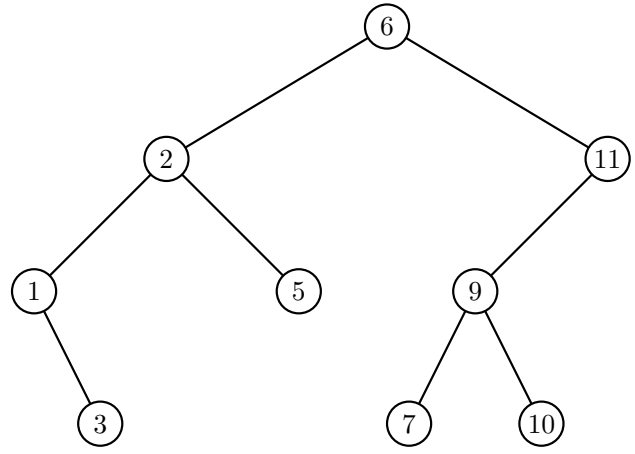
Attention !!!

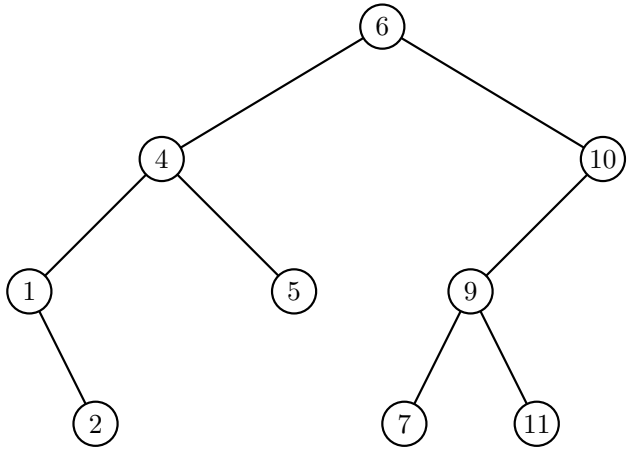
localement, *la définition entraîne* que chaque nœud a une étiquette comprise entre celle de son fils gauche et celle de son fils droit *mais ceci ne suffit pas*

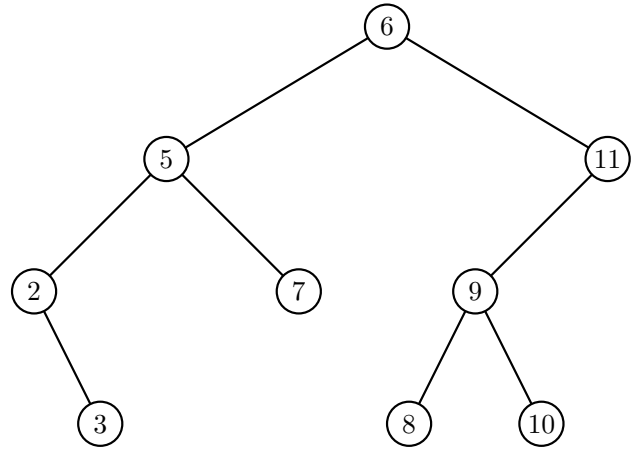
ABR or not ?

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



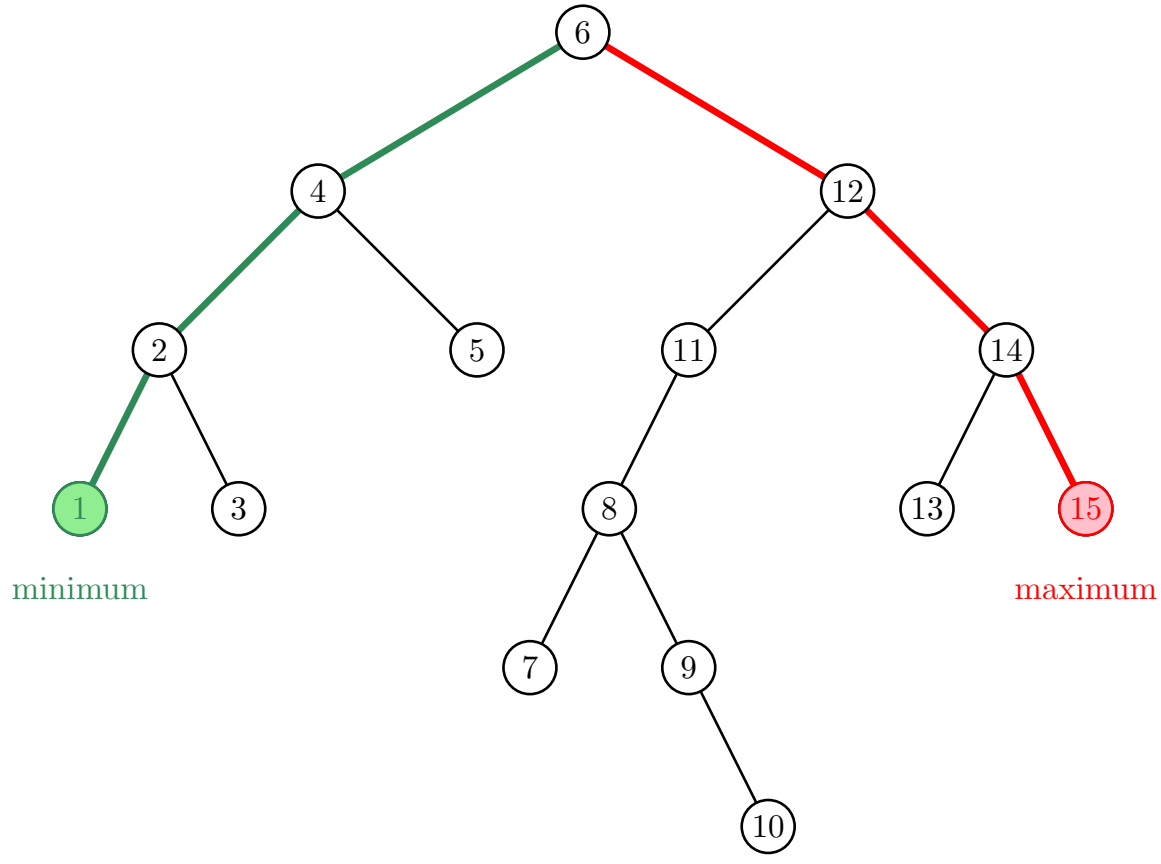






Minimum et maximum d'un ABR

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



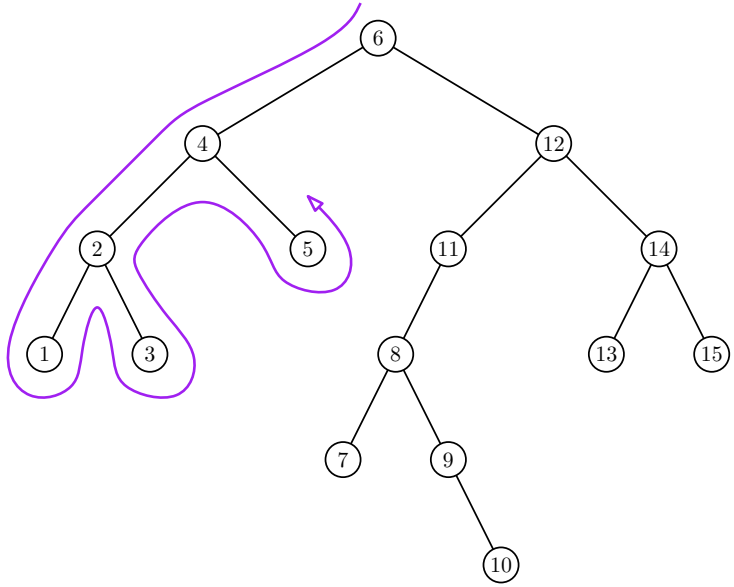
Plus généralement, la « disposition spatiale » des éléments suit leur ordre : plus un élément est à gauche, plus il est petit. L'élément de rang 2 de l'ABR est le minimum du sous-arbre droit du minimum (s'il en a un), ou son père sinon, etc.

Ordre dans un ABR

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

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 $\Theta(n)$.

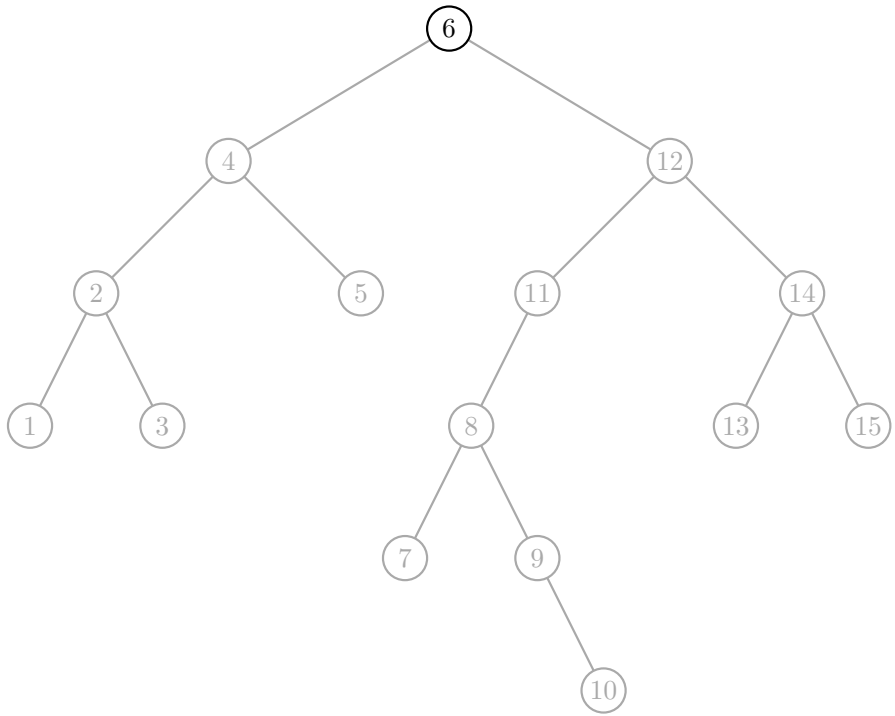
Corollaire

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

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.

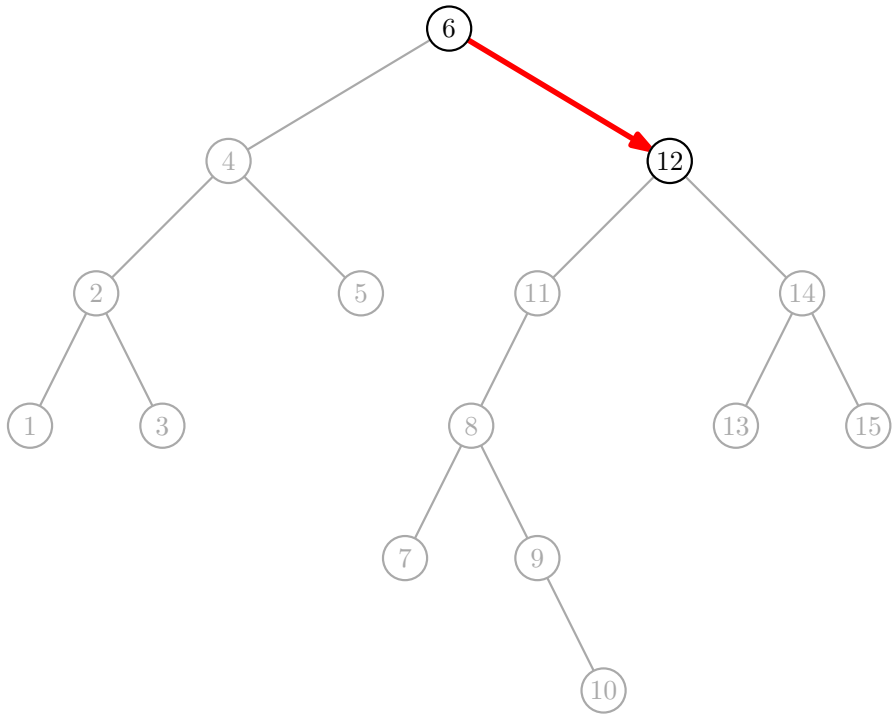
Exemple – recherche de 9 :



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.

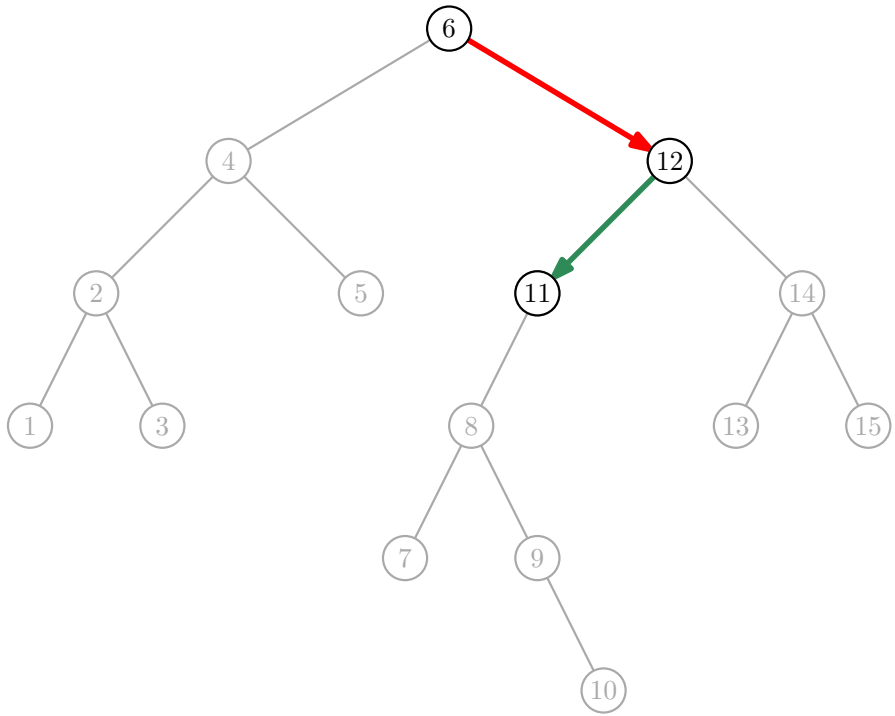
Exemple – recherche de 9 :



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.

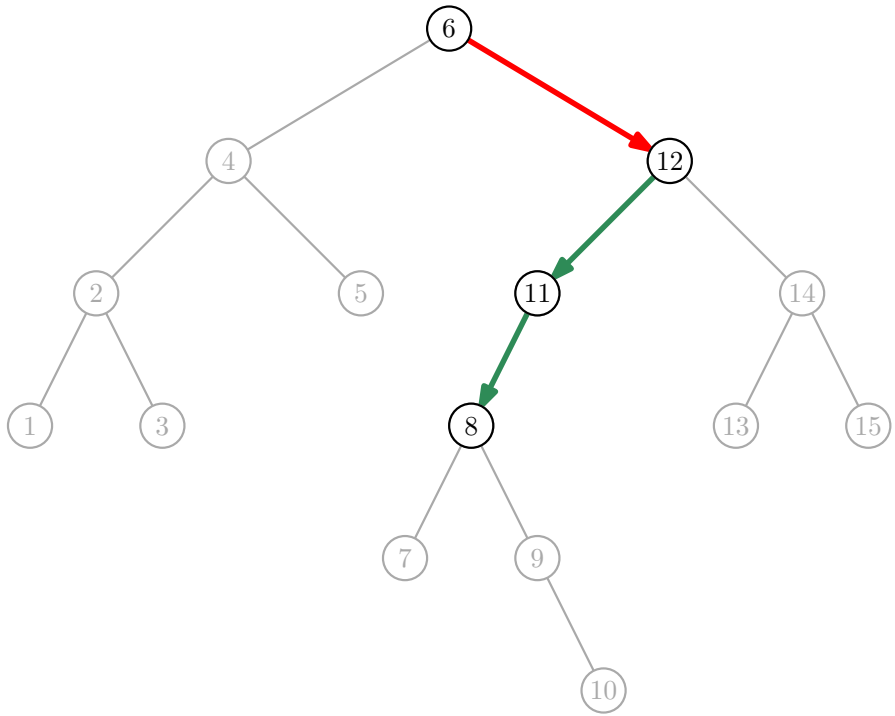
Exemple – recherche de 9 :



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.

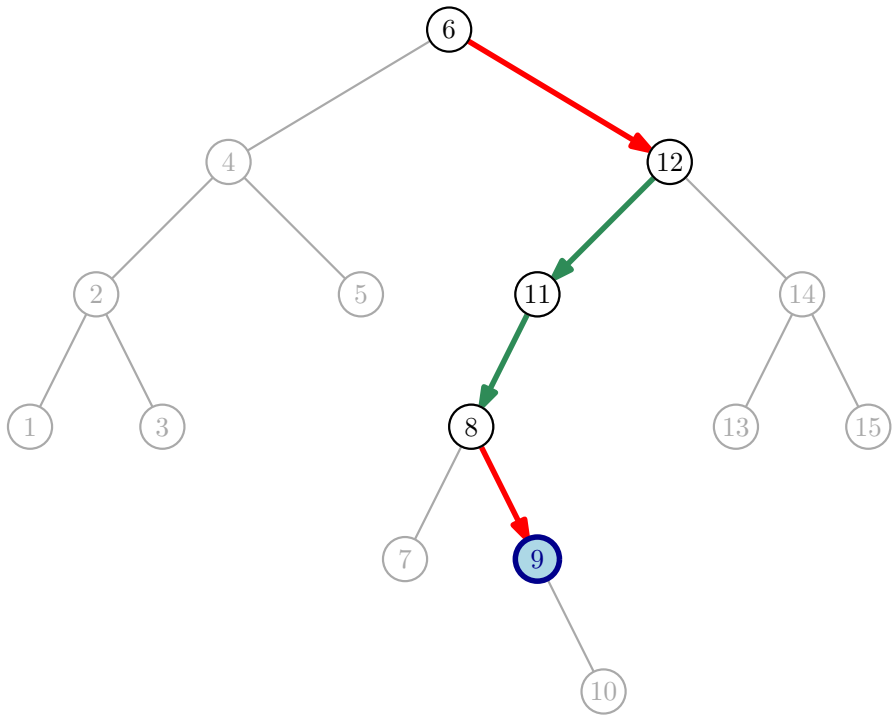
Exemple – recherche de 9 :



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 :



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 très simple à dérécursiver, puisqu'au plus un appel récursif, terminal, est effectué.

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

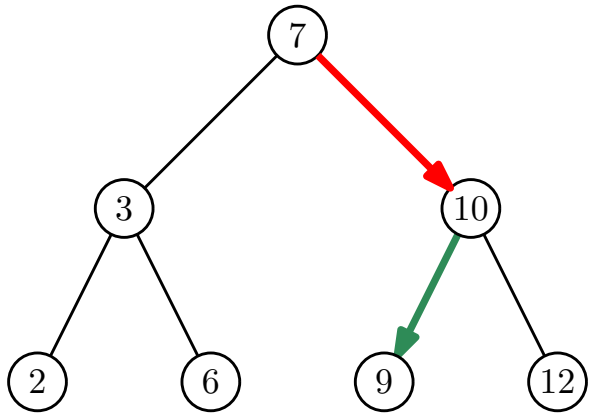
Théorème

`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

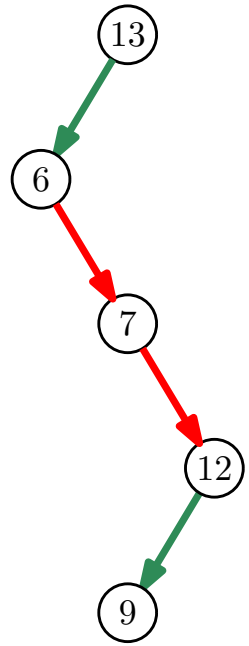
La recherche (et plus généralement tous les algorithmes sur les ABR) se comportent radicalement différemment selon la forme de l'arbre, qui détermine le rapport entre sa taille et sa hauteur.

Cas sympathique : ABR « parfait »



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

Cas désagréable : ABR « filiforme »



Hauteur n, recherche aussi inefficace que la recherche dans une liste chaînée

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

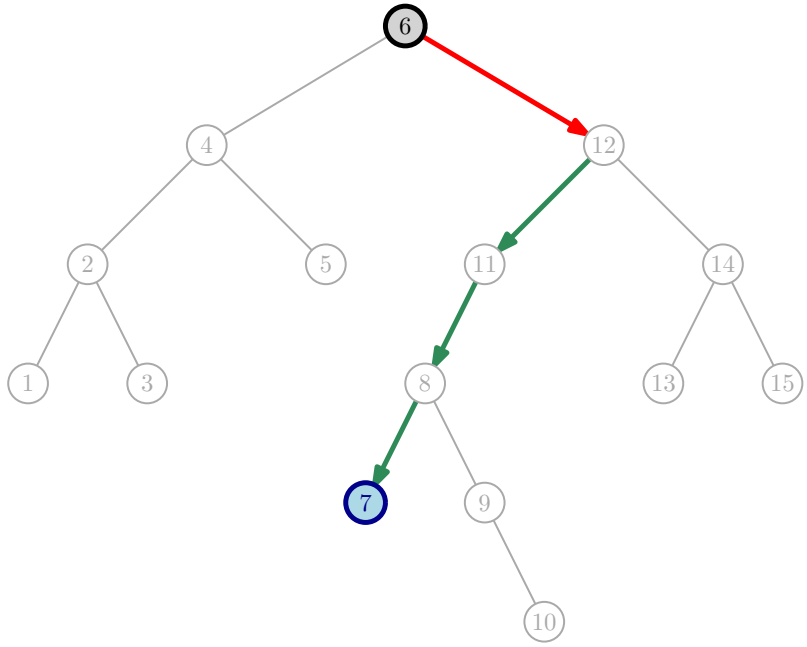
`minimum(r)` détermine le plus petit élément dans l'ABR de racine `r` en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

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

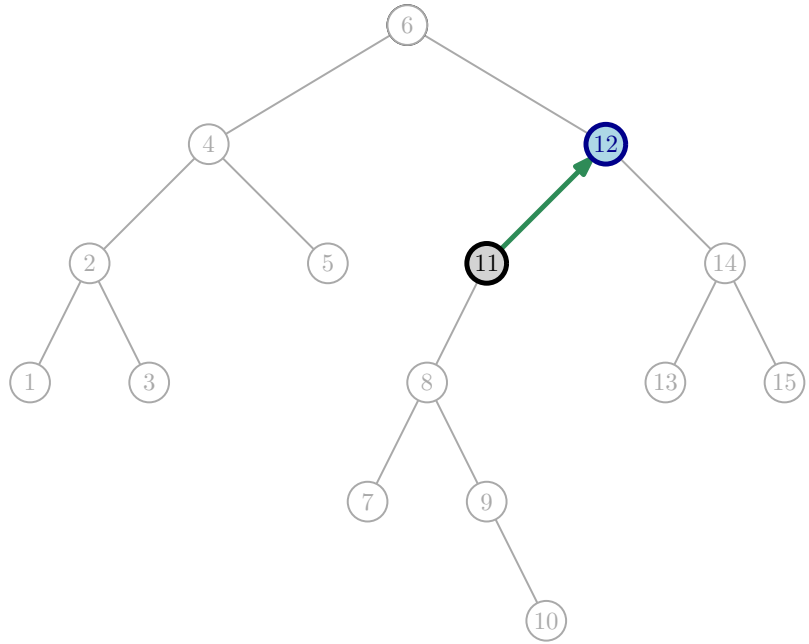


$\Longrightarrow$ le successeur est le minimum du sous-arbre droit

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

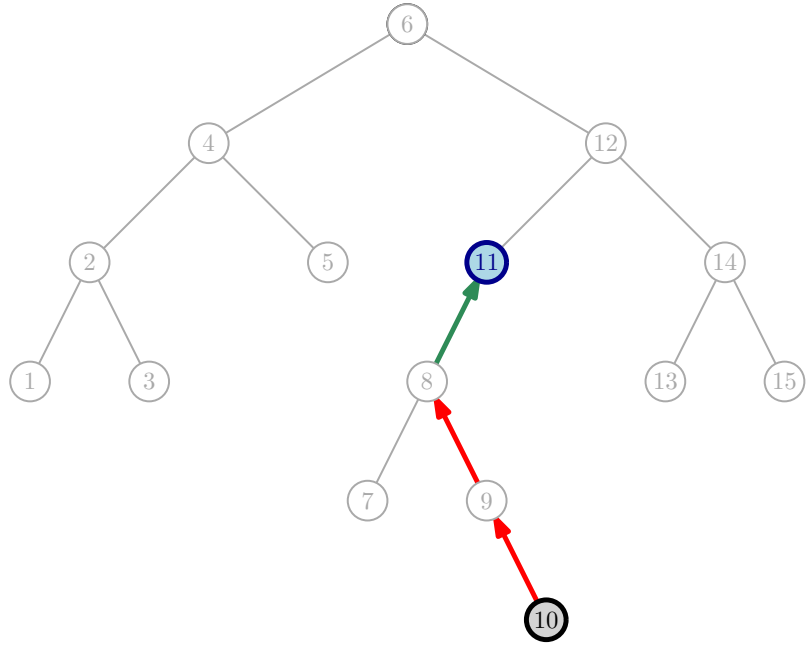


$\Longrightarrow$ le successeur est le premier ancêtre supérieur à l'élément – donc le premier vers lequel on remonte depuis la gauche

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.