> Version pdf

Synthèse du cours Arbre binaire de recherche

Propriété d'arbre binaire de recherche

"Point de cours 1 : propriété d'arbre binaire de recherche"

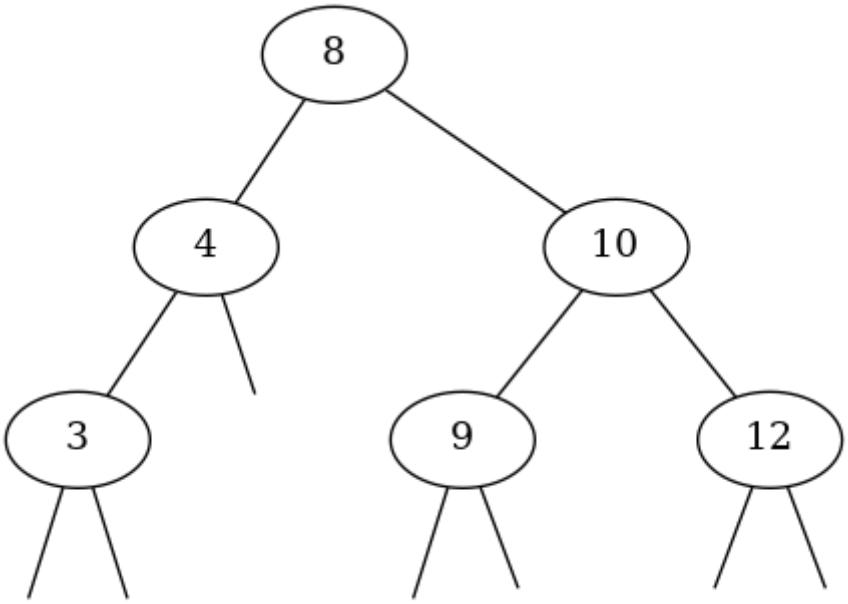
Un **arbre binaire de recherche** (ou ABR) est un arbre binaire vérifiant certaines propriétés.

- **Premier cas :** un **arbre binaire de recherche** peut être *vide*
- **Second cas :** un **arbre binaire** non vide est un **arbre binaire de recherche** s'il vérifie les conditions suivantes :
  - **(C1) :** tous les éléments stockés dans les noeuds sont de même type et *comparables* deux à deux
  - **(C2) :** pour tous les *noeuds* de l'arbre binaire, l'élément stocké dans un noeud est *supérieur ou égal* [1] à tous les éléments stockés dans son *fils/sous-arbre gauche* (s'il est non vide) et *inférieur ou égal* [1:1] à tous les éléments stockés dans son *fils/sous-arbre droit* (s'il est non vide).

"Exemple 1"

L'arbre binaire ci-dessous est non vide et vérifie la propriété d'arbre binaire de recherche :

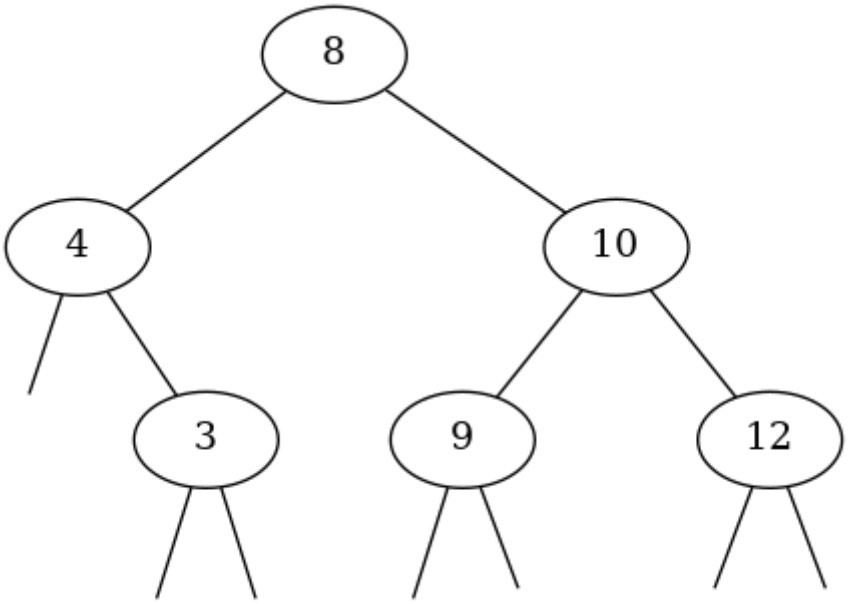
- tous les éléments sont des entiers comparables deux à deux ;
- pour tous les noeuds, l'élément stocké dans le noeud est supérieur à tous les éléments stockés dans son sous-arbre gauche et inférieur à tous les éléments stockés dans son sous-arbre droit



"Exemple 2"

L'arbre binaire ci-dessous est non vide et contient des éléments entiers comparables deux à deux, mais il ne vérifie pas la propriété d'arbre binaire de recherche :

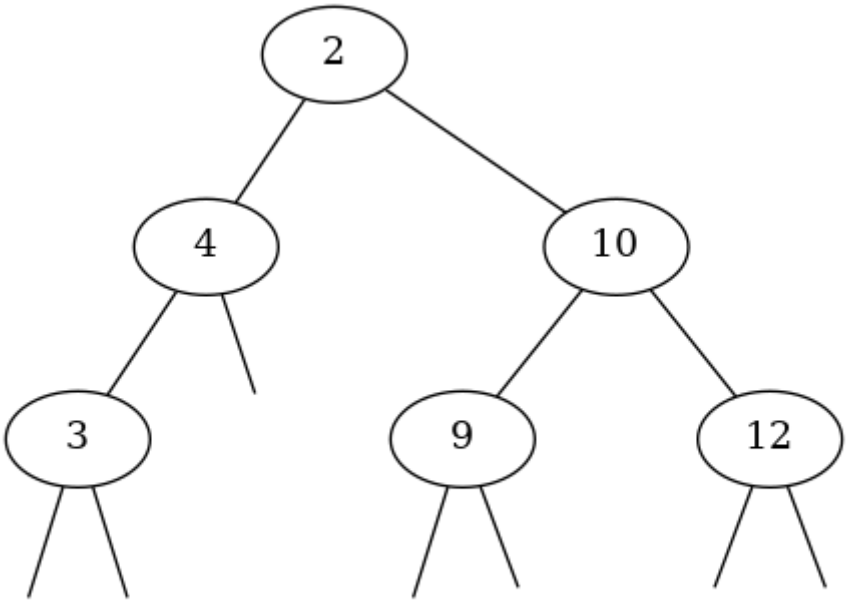
- l'élément 4 n'est pas inférieur ou égal à l'élément 3 qui se trouve dans le sous-arbre droit du noeud où il est stocké.



"Exemple 3"

L'arbre binaire ci-dessous est non vide et contient des éléments entiers comparables deux à deux, mais il ne vérifie pas la propriété d'arbre binaire de recherche :

- l'élément 2 n'est pas supérieur ou égal aux éléments 3 et 4 qui se trouvent dans le sous-arbre gauche du noeud où il est stocké.



Propriétés

"Point de cours 2 : maximum ou minimum dans un arbre binaire de recherche"

Soit un arbre binaire vérifiant la propriété d'**arbre binaire de recherche**.

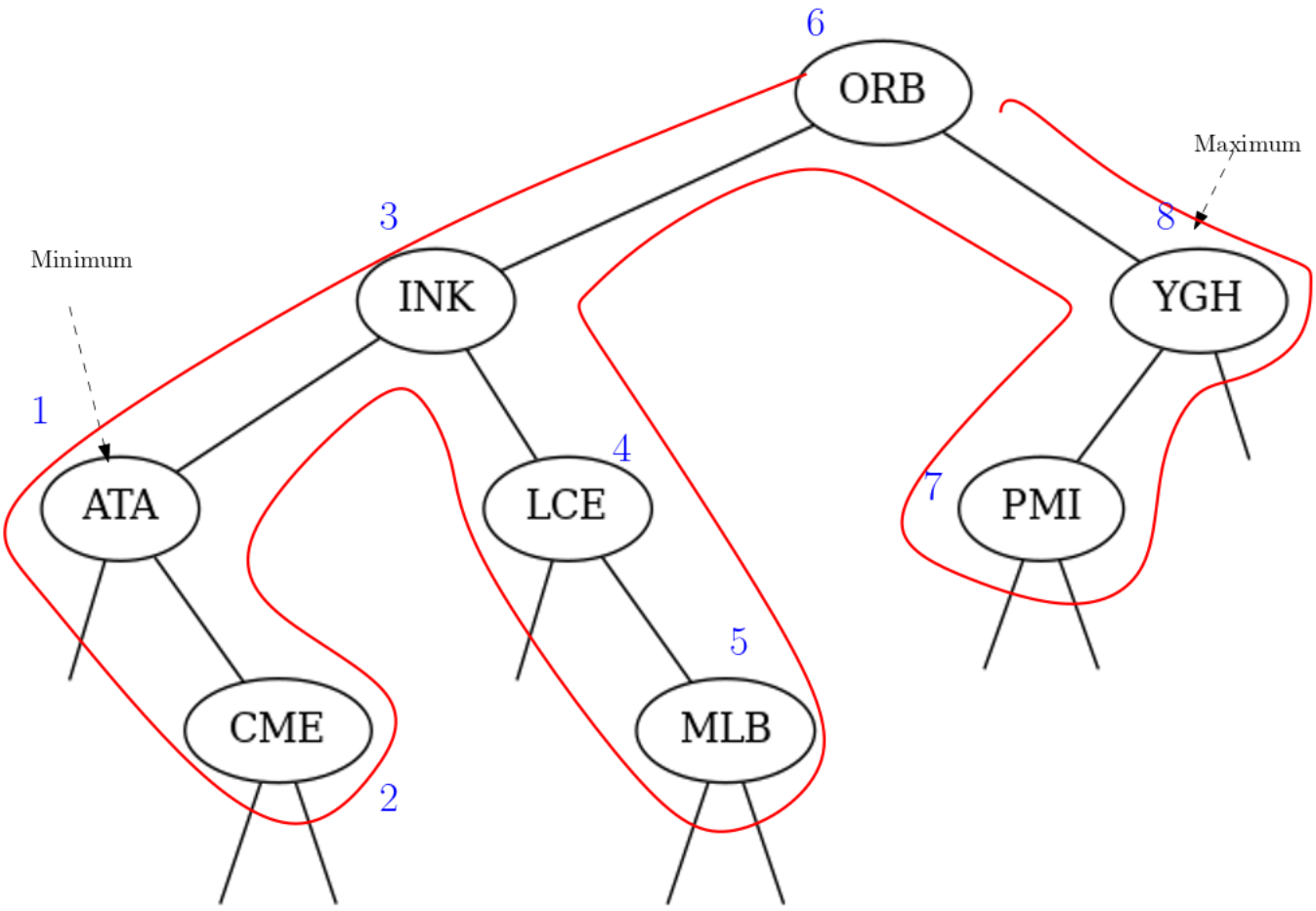
- *Minimum :* Pour déterminer le **minimum** des éléments stockés dans les noeuds de l'arbre il faut et il suffit de partir du noeud racine et de *toujours descendre dans le fils/sous-arbre gauche* tant que celui-ci est non vide. Le **minimum** est alors l'élément stocké dans le premier noeud atteint dont le fils/sous-arbre gauche est vide.
- *Maximum :* Pour déterminer le **maximum** des éléments stockés dans les noeuds de l'arbre il faut et il suffit de partir du noeud racine et de *toujours descendre dans le fils/sous-arbe droit* tant que celui-ci est non vide. Le **maximum** est alors l'élément stocké dans le premier noeud atteint dont le fils/sous-arbre droit est vide.

"Point de cours 3 : parcours infixe d'un arbre binaire de recherche"

Le **parcours infixe** d'un arbre binaire vérifiant la propriété d'**arbre binaire de recherche** donne une **énumération dans l'ordre croissant** des éléments stockés dans les noeuds.

"Exemple"

On donne ci-dessous un arbre binaire de recherche dont les noeuds portent des chaînes de caractère avec les éléments maximum et minimum et l'ordre d'énumération infixe du parcours en profondeur. Celui-ci donne bien la séquence des éléments dans l'ordre croissant.



Interface et implémentation

"Point de cours 4"

Un **arbre binaire de recherche** est un arbre binaire vérifiant la *propriété d'arbre binaire de recherche*.

Pour implémenter la structure de données **arbre binaire de recherche** on peut donc reprendre une implémentation d'arbre binaire. La *propriété d'arbre binaire de recherche* sera un *invariant* de la structure préservé par l'ajout ou la suppression d'un nouvel élément dans l'arbre.

À partir des implémentations proposées pour la structure d'arbre binaire on va donner deux implémentations d'**arbre binaire de recherche** en étendant l'interface d'arbre binaire avec de nouvelles fonctions :

- une implémentation d'arbre binaire de recherche *immuable* à partir d'une classe `Noeud` et d'une *interface fonctionnelle* : chaque fonction renvoie un nouvel arbre sans modifier en place l'arbre binaire auquel elle s'applique
- une implémentation d'arbre binaire de recherche *mutable* à partir de deux classes `Noeud` et `ABR` : les méthodes de la classe `ABR` modifient en place l'arbre binaire.

"Méthode 1 : arbre binaire de recherche immuable"

```
class Noeud:
    """Noeud pour arbre binaire"""

    def __init__(self, g, e, d):
        self.gauche = g # lien vers fils gauche g éventuellement vide (None)
        self.element = e # élément e stocké dans le noeud
        self.droit = d # lien vers fils droit d éventuellement vide (None)
```

```python
# Interface fonctionnelle minimale pour Arbre Binaire de Recherche
def abr_vide():
    """Renvoie un arbre binaire de recherche vide représenté par None"""
    return None

def est_vide(abr):
    """Teste si un arbre binaire de recherche est vide, renvoie un booléen"""
    return abr is None

def gauche(abr):
    """Renvoie le sous-arbre fils gauche de  l'arbre binaire de recherche abr
    Provoque une erreur si arbre est vide"""
    assert not est_vide(abr), "Arbre vide"
    return abr.gauche

def droit(abr):
    """Renvoie le sous-arbre fils droit de  l'arbre binaire de recherche abr
    Provoque une erreur si arbre est vide"""
    assert not est_vide(abr), "Arbre vide"
    return abr.droit

def element_racine(abr):
    """Renvoie l'élément à la racine de l'arbre binaire de recherche abr
    Provoque une erreur si arbre est vide"""
    assert not est_vide(abr), "Arbre vide"
    return abr.element

# Extension de l'interface
def hauteur(abr):
    """Renvoie la hauteur de l'arbre  binaire de recherche abr"""
    if est_vide(abr):
        return 0
    return 1 + max(hauteur(droit(abr)), hauteur(gauche(abr)))

def taille(abr):
    """Renvoie la hauteur de l'arbre  binaire de recherche abr"""
    if est_vide(abr):
        return 0
    return 1 + taille(droit(abr)) + taille(gauche(abr))

def maximum(abr):
    """Renvoie le maximum de  l'arbre  binaire de recherche abr"""
    assert not est_vide(abr), "arbre vide"
```

```
    if est_vide(abr.droit):
        return abr.element
    return maximum(abr.droit)

def minimum(abr):
    """Renvoie le minimum de  l'arbre  binaire de recherche abr"""
    assert not est_vide(abr), "arbre vide"
    if est_vide(abr.gauche):
        return abr.element
    return minimum(abr.gauche)


def parcours_infixe(abr, tab):
    """Renvoie une trace du parcours infixe de l'arbre binaire de recherche
    dans le tableau dynamique tab"""
    if est_vide(abr):
        return
    parcours_infixe(abr.gauche, tab)
    tab.append(abr.element)
    parcours_infixe(abr.droit, tab)
```

"Méthode 2 : arbre binaire de recherche mutable"

```python
class Noeud:
    """Noeud pour arbre binaire"""

    def __init__(self, g, e, d):
        self.gauche = g # lien vers fils gauche g éventuellement vide (None)
        self.element = e # élément e stocké dans le noeud
        self.droit = d # lien vers fils droit d éventuellement vide (None)


class ABR:
    """Classe d'arbre binaire de recherche mutable"""

    def __init__(self):
        """Constructeur, self.racine pointe vers None si arbre vide
        ou le noeud racine"""
        self.racine = None

    def est_vide(self):
        """Teste si l'arbre est vide, renvoie un booléen"""
        return self.racine is None

    def droit(self):
        """Renvoie le sous-arbre (de type Arbre) fils droit de l'arbre
        Provoque une erreur si arbre est vide"""
        assert not self.est_vide()
        return self.racine.droit
```

```
    def gauche(self):
        """Renvoie le sous-arbre (de type ABR)  gauche de l'arbre
        Provoque une erreur si arbre est vide"""
        assert not self.est_vide()
        return self.racine.gauche

    def element_racine(self):
        """Renvoie l'élément stocké dans le noeud racine de l'arbre
        Provoque une erreur si arbre est vide"""
        assert not self.est_vide()
        return self.racine.element

    # extension de l'interface
    def parcours_infixe(self, tab):
        """Renvoie une trace du parcours infixe de l'arbre binaire de recherche
        dans la tableau dynamique tab
        """
        if self.est_vide():
            return
        self.gauche().parcours_infixe(tab)
        tab.append(self.element_racine())
        self.droit().parcours_infixe(tab)

    def minimum(self):
        assert not  self.est_vide(), "arbre vide"
        if self.gauche().est_vide():
            return self.element_racine()
        else:
            return self.gauche().minimum()

    def maximum(self):
        assert not  self.est_vide(), "arbre vide"
        if self.racine.droit.est_vide():
            return self.racine.element
        else:
            return self.racine.droit.maximum()

    def taille(self):
        """Renvoie la taille de l'arbre binaire de recherche"""
        if self.est_vide():
            return 0
        return 1 + self.gauche().taille() +  self.droit().taille()
```

```
    def hauteur(self):
        """Renvoie la hauteur de l'arbre binaire de recherche"""
        if self.est_vide():
            return 0
        return 1 + max(self.gauche().hauteur(), self.droit().hauteur())
```

Recherche et ajout d'élément

"Avertissement"

Dans cette partie on considère des **arbres binaires de recherche** tels que pour chacun des sous-arbres l'élément stocké dans le noeud racine est *supérieur strictement* à tous les éléments stockés dans les noeuds de son sous-arbre gauche et *inférieur ou égal* à tous les éléments stockés dans les noeuds de son sous-arbre droit.

"Point de cours 4"

Pour rechercher un élément dans un arbre binaire présentant la **propriété d'arbre binaire de recherche** on exploite cette propriété pour procéder *récursivement* à l'instar d'une *recherche dichotomique dans un tableau trié* en éliminant une partie des noeuds restants chaque fois que la recherche doit se poursivre :

- **Étape 1 :** Si l'arbre est vide, alors on termine la recherche et on renvoie `False`, sinon on compare l'élément stocké dans le noeud racine à l'élément cherché et on passe à l'étape 2.
- **Étape 2 :** Trois alternatives sont possibles en fonction de la comparaison de l'élément stocké dans le noeud racine avec l'élément cherché.
  - S'ils sont *égaux*, alors on termine la recherche et on renvoie `True`.
  - Si l'élément cherché est *inférieur* à l'élément stocké dans le noeud racine, alors on poursuit la recherche dans le *fils/sous-arbre gauche* et on revient à l'étape 1 (appel récursif).
  - Sinon, alors on poursuit la recherche dans le *fils/sous-arbre droit* et on revient à l'étape 1 (appel récursif).

"⏱️ Complexité"

> ⚠️ Pour simplifier, on considère que le coût de la comparaison de deux éléments est constant, dans notre étude de complexité on ne prend donc en compte que le nombre de comparaisons.

Le nombre de comparaisons effectuées lors de la recherche d'un élément dans un *arbre binaire* de recherche est au plus égal au nombre de noeuds dans le plus long chemin reliant la racine à une feuille. La *complexité en temps de la recherche* est donc majorée par une constante fois la *hauteur* de l'arbre binaire de recherche.
On rappelle que la hauteur h d'un arbre binaire de taille n vérifie l'inégalité $\log_2(n) < h \leqslant n$.
La *complexité en temps de la recherche* dans un arbre binaire de recherche dépend donc de la forme de l'arbre, avec deux cas extrêmes :

- *Pire forme :* dans un *arbre binaire dégénéré*, par un exemple un peigne, on aura une *complexité linéaire* en $O(n)$ comme pour une recherche séquentielle dans une liste chaînée.
- *Meilleure forme :* dans un *arbre binaire presque complet* ou *parfait*, on aura une *complexité logarithmique*, en $O(\log_2(n))$ donc bien meilleure.

Forme de l'arbre binaire	**Complexité de la recherche d'un élément par rapport à la taille**
dégénéré	linéaire
presque complet ou parfait	logarithmique

"Méthode 3 : recherche avec une interface fonctionnelle d'ABR immuable"

```python
def recherche(abr, elt):
    """Renvoie un booléen indiquant si elt est stocké dans un noeud
    de l'arbre binaire de recherche abr
    """
    if est_vide(abr):
        return False
    elif elt < abr.element:
        return recherche(abr.gauche, elt)
    elif elt > abr.element:
        return recherche(abr.droit, elt)
    else:
        return True
```

"Méthode 4 : recherche avec une interface POO d'ABR mutable "

```python
def recherche(self, elt):
    """
    Renvoie True si element dans l'arbre binaire de recherche
    et False sinon
    """
    if self.est_vide(): # cas de l'arbre vide
        return False
    elif elt < self.element_racine():
        return self.gauche().recherche(elt)
    elif elt > self.element_racine():
        return self.droit().recherche(elt)
    else:
        return True
```

"Point de cours 5 : ajout dans un ABR"

Pour ajouter un élément dans un arbre binaire présentant la **propriété d'arbre binaire de recherche** on effectue la même descente dans l'arbre que pour la recherche. Deux situations finales sont possibles :

- *Situation 1 :* on atteint un arbre dont le noeud racine contient déjà l'élément stocké, dans ce cas :
  - *si on ne veut pas de doublons* dans l'arbre binaire initial : on termine sans rien faire

- *si on accepte les doublons* dans l'arbre binaire initial : on poursuit alors l'ajout dans le *fils/sous-arbre gauche*

- *Situation 2 :* on atteint un arbre vide :
  - on crée alors à cet emplacement libre un arbre binaire dont le noeud racine contient l'élément

"⏱️ Complexité"

> ⚠️ Pour simplifier, on considère que le coût de la comparaison de deux éléments est constant, dans notre étude de complexité on ne prend donc en compte que le nombre de comparaisons.

Le nombre de comparaisons effectuées lors de la descente dans l'arbre est le même que pour la recherche. La *complexité en temps de l'ajout* est donc majorée par une constante fois la *hauteur* de l'arbre binaire de recherche.

Forme de l'arbre binaire	Complexité de l'ajout d'un élément par rapport à la taille
dégénéré	linéaire
presque complet ou parfait	logarithmique

L'ajout (comme la suppression) d'éléments dans l'arbre va modifier sa forme. Par exemple si on ajoute des éléments dans l'ordre croissant, on obtiendra un peigne droit. Or on veut maintenir une hauteur proche de l'optimum $\log_2(n)$ où n est la taille de l'arbre. Différentes techniques de réarrangement des noeuds au cours d'un ajout (ou d'une suppression) permettent de maintenir un arbre *équilibré*, ou presque, tout en gardant une complexité logarithmique. [2]

"Méthode 5 : ajout avec une interface fonctionnelle d'ABR immuable"

Premier cas : on accepte les éléments en doublons dans l'arbre.

```
def ajoute(abr, elt):
    """Renvoie un nouvel arbre binaire de recherche construit par ajout de  elt
    """
    if abr is None:
        return Noeud(None, elt, None)
    elif elt <= abr.element:
        return Noeud(ajoute(abr.gauche, elt), abr.element, abr.droit)
    else:
        return Noeud(abr.gauche, abr.element, ajoute(abr.droit, elt))
```

Deuxième vas : on n'accepte pas les éléments en doublons dans l'arbre, si l'élément est déjà présent on renvoie une copie superficielle de l'arbre.

```
def ajoute(abr, elt):
    """Renvoie un nouvel arbre binaire de recherche construit par ajout de  elt
    """
    if abr is None:
        return Noeud(None, elt, None)
    elif elt < abr.element:
        return Noeud(ajoute(abr.gauche, elt), abr.element, abr.droit)
    elif elt > abr.element:
        return Noeud(abr.gauche, abr.element, ajoute(abr.droit, elt))
    else:
        return Noeud(abr.gauche, abr.element, abr.droit)
```

"Méthode 6 : ajout avec une interface POO d'ABR mutable"

Premier cas : on accepte les éléments en doublons dans l'arbre.

```
def ajoute(self, elt):
    """Ajoute elt dans une feuille de l'arbre binaire de recherche
    Maintient la propriété d'arbre binaire de recherche."""
    if self.est_vide():
        self.racine = Noeud(ABR(), elt, ABR())
    elif elt <= self.element_racine():
        self.gauche().ajoute(elt)
    else:
        self.droit().ajoute(elt)
```

Deuxième vas : on n'accepte pas les éléments en doublons dans l'arbre, si l'élément est déjà présent on ne fait rien.

```
def ajoute(self, elt):
    """Ajoute elt dans une feuille de l'arbre binaire de recherche
    Maintient la propriété d'arbre binaire de recherche."""
    if self.est_vide():
        self.racine = Noeud(ABR(), elt, ABR())
    elif elt < self.element_racine():
        self.gauche().ajoute(elt)
    elif elt > self.element_racine():
        self.droit().ajoute(elt)
```

1. L'inégalité peut devenir stricte si on veut exclure les doublons pour stocket un ensemble. ↩ ↩
2. Ces techniques d'équilibrage sont hors-programme :arbres rouge-noir ou AVL. ↩