Les arbres binaires

Résumé

L'arborescence de fichiers sur un ordinateur, l'organigramme d'une entreprise, l'arbre généalogique d'une famille, l'expression arithmétique $3 \times (2 + 5)$: toutes ces structures ont la forme d'un **arbre**. En informatique, un arbre est une structure hiérarchique où chaque élément (appelé nœud) peut avoir des enfants, mais un seul parent. L'**arbre binaire** (au plus deux enfants par nœud) est la variante la plus étudiée : c'est la base des algorithmes de recherche rapide (arbres binaires de recherche) et des parcours systématiques (préfixe, infixe, suffixe, largeur).

Prérequis : La récursivité, Programmation orientée objet.

① Mémo

Définitions 1 (Vocabulaire)

Arbre binaire Structure hiérarchique dans laquelle chaque nœud a au plus deux enfants, un à gauche et un à droite.
Racine Nœud au sommet de l'arbre (sans parent).
Feuille Nœud sans enfant.
Nœud interne Nœud ayant au moins un enfant.
Hauteur Longueur du plus long chemin de la racine à une feuille. Un arbre réduit à un nœud a une hauteur de 0. L'arbre vide a une hauteur de -1 (par convention).
Taille Nombre total de nœuds.
Profondeur d'un nœud Distance (nombre d'arêtes) entre la racine et ce nœud.

Propriété 1 (Propriétés fondamentales)

Pour un arbre binaire de hauteur h et de taille n :

- $h + 1 \leqslant n \leqslant 2^{h+1} - 1$;
- un arbre complet de hauteur h a exactement $2^{h+1} - 1$ nœuds ;
- $\lfloor \log_2(n) \rfloor \leqslant h \leqslant n - 1$.

Définition 1 (Arbre binaire de recherche (ABR))

Un ABR est un arbre binaire où, pour chaque nœud :

- toutes les valeurs du sous-arbre gauche sont **inférieures** ;
- toutes les valeurs du sous-arbre droit sont **supérieures**.

La recherche, l'insertion et la suppression sont en $O(h)$, soit $O(\log n)$ si l'arbre est équilibré.

Parcours d'un arbre binaire

Préfixe (préordre) Racine, sous-arbre gauche, sous-arbre droit.
Infixe (inordre) Sous-arbre gauche, racine, sous-arbre droit. Pour un ABR, donne les valeurs dans l'ordre croissant.
Suffixe (postordre) Sous-arbre gauche, sous-arbre droit, racine.
En largeur (BFS) Niveau par niveau, de gauche à droite (utilise une file).

Erreurs classiques

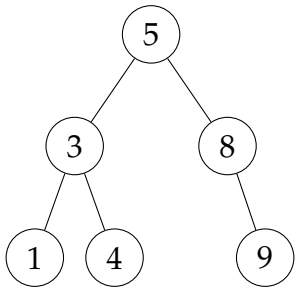
Code erroné	Code correct	Explication
Confondre hauteur et profondeur	Hauteur = plus long chemin **descendant**, profondeur = distance à la **racine**	La hauteur est une propriété de l'arbre (ou d'un sous-arbre) ; la profondeur est une propriété d'un nœud. Ce sont deux mesures distinctes.
Vérifier un ABR en comparant seulement enfant gauche < racine < enfant droit	Propager un intervalle]mini ; maxi[à chaque nœud	Un nœud peut être inférieur à son parent mais supérieur à un ancêtre plus haut : il faut vérifier la contrainte sur *tout* le sous-arbre.
`def taille(arbre): return 1 + taille( arbre.gauche) + ...`	Ajouter `if arbre is None: return 0`	Sans le cas de base `None`, l'accès à `arbre.gauche` sur l'arbre vide provoque un `AttributeError`.
Insérer des valeurs triées dans un ABR	Utiliser un arbre équilibré (AVL, rouge-noir)	L'insertion de valeurs triées dégénère l'ABR en liste chaînée (hauteur = $n-1$) : la recherche passe de $O(\log n)$ à $O(n)$.

② Exemples

Programme 1 · Implémentation d'un arbre binaire

```
class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

arbre = Arbre(5,
    Arbre(3, Arbre(1), Arbre(4)),
    Arbre(8, None, Arbre(9))
)
```



Programme 2 · Taille et hauteur (récursif)

```
class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

arbre = Arbre(5,
    Arbre(3, Arbre(1), Arbre(4)),
    Arbre(8, None, Arbre(9))
)

def taille(arbre):
    if arbre is None:
        return 0
    return 1 + taille(arbre.gauche) + taille(arbre.droit)

def hauteur(arbre):
    if arbre is None:
        return -1
    return 1 + max(hauteur(arbre.gauche), hauteur(arbre.droit))
```

Programme 3 · Parcours

```
class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

arbre = Arbre(5,
    Arbre(3, Arbre(1), Arbre(4)),
    Arbre(8, None, Arbre(9))
)

def prefixe(arbre):
    if arbre is None:
        return []
    return [arbre.valeur] + prefixe(arbre.gauche) + prefixe(arbre.droit)

def infixe(arbre):
    if arbre is None:
        return []
    return infixe(arbre.gauche) + [arbre.valeur] + infixe(arbre.droit)

def suffixe(arbre):
    if arbre is None:
        return []
    return suffixe(arbre.gauche) + suffixe(arbre.droit) + [arbre.valeur]

# Pour l'arbre ci-dessus :
# prefixe donne [5, 3, 1, 4, 8, 9]
# infixe  donne [1, 3, 4, 5, 8, 9]  (ordre croissant car ABR)
# suffixe donne [1, 4, 3, 9, 8, 5]
```

Programme 4 · Parcours en largeur

```
from collections import deque

class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

arbre = Arbre(5,
    Arbre(3, Arbre(1), Arbre(4)),
    Arbre(8, None, Arbre(9))
)

def largeur(arbre):
    if arbre is None:
```

```
        return []
    file = deque([arbre])
    resultat = []
    while file:
        noeud = file.popleft()
        resultat.append(noeud.valeur)
        if noeud.gauche:
            file.append(noeud.gauche)
        if noeud.droit:
            file.append(noeud.droit)
    return resultat
# largeur donne [5, 3, 8, 1, 4, 9]
```

Programme 5 · Recherche dans un ABR

```
class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

arbre = Arbre(5,
    Arbre(3, Arbre(1), Arbre(4)),
    Arbre(8, None, Arbre(9))
)

def rechercher_abr(arbre, x):
    if arbre is None:
        return False
    if x == arbre.valeur:
        return True
    elif x < arbre.valeur:
        return rechercher_abr(arbre.gauche, x)
    else:
        return rechercher_abr(arbre.droit, x)
```

Programme 6 · Insertion dans un ABR

```
class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

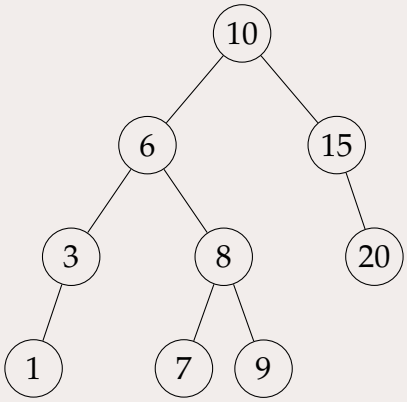
arbre = Arbre(5,
    Arbre(3, Arbre(1), Arbre(4)),
    Arbre(8, None, Arbre(9))
)

```

```python
def inserer_abr(arbre, x):
    if arbre is None:
        return Arbre(x)
    if x < arbre.valeur:
        arbre.gauche = inserer_abr(arbre.gauche, x)
    elif x > arbre.valeur:
        arbre.droit = inserer_abr(arbre.droit, x)
    return arbre
```

③ Exercices

Exercice 1 *Propriétés d'un arbre* Soit l'arbre binaire suivant :



1. Donner la taille, la hauteur et les feuilles de cet arbre.
2. Est-ce un ABR ? Justifier.
3. Donner le résultat des quatre parcours (préfixe, infixe, suffixe, largeur).

Exercice 2 *Fonctions récursives* Écrire les fonctions suivantes :

1. `nb_feuilles(arbre)` qui renvoie le nombre de feuilles.
2. `somme(arbre)` qui renvoie la somme de toutes les valeurs.
3. `est_abr(arbre)` qui vérifie si un arbre est un ABR.

Exercice 3 *Construction d'un ABR* On insère successivement les valeurs 5, 2, 8, 1, 3, 7, 9 dans un ABR initialement vide.

1. Dessiner l'arbre obtenu.
2. Quel serait l'arbre si l'on insérait les mêmes valeurs dans l'ordre 1, 2, 3, 5, 7, 8, 9 ? Quel problème cela pose-t-il ?

Solutions des exercices

Corrigé de l'exercice 1.

1. Taille : 9 nœuds. Hauteur : 3 (chemin 10-6-3-1). Feuilles : 1, 7, 9, 20.
2. Oui, c'est un ABR. Pour chaque nœud, les valeurs à gauche sont inférieures et à droite supérieures. Par exemple, 10 a {1,3,6,7,8,9} à gauche et {15,20} à droite.
3. Parcours :
   - Préfixe : 10, 6, 3, 1, 8, 7, 9, 15, 20.
   - Infixe : 1, 3, 6, 7, 8, 9, 10, 15, 20 (ordre croissant, confirmation que c'est un ABR).
   - Suffixe : 1, 3, 7, 9, 8, 6, 20, 15, 10.
   - Largeur : 10, 6, 15, 3, 8, 20, 1, 7, 9.

Corrigé de l'exercice 2.

```
def nb_feuilles(arbre):
    if arbre is None:
        return 0
    if arbre.gauche is None and arbre.droit is None:
        return 1
    return nb_feuilles(arbre.gauche) + nb_feuilles(arbre.droit)

def somme(arbre):
    if arbre is None:
        return 0
    return arbre.valeur + somme(arbre.gauche) + somme(arbre.droit)

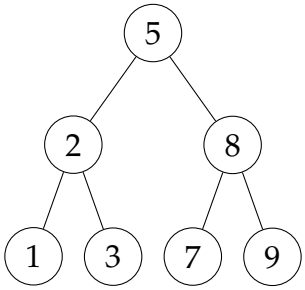
def est_abr(arbre, mini=float('-inf'), maxi=float('inf')):
    if arbre is None:
        return True
    if arbre.valeur <= mini or arbre.valeur >= maxi:
        return False
    return (est_abr(arbre.gauche, mini, arbre.valeur) and
            est_abr(arbre.droit, arbre.valeur, maxi))
```

Principe de `est_abr` : on propage un intervalle]mini ; maxi[que chaque nœud doit respecter :

- le sous-arbre gauche doit avoir des valeurs dans]mini ; valeur[;
- le sous-arbre droit dans]valeur ; maxi[.

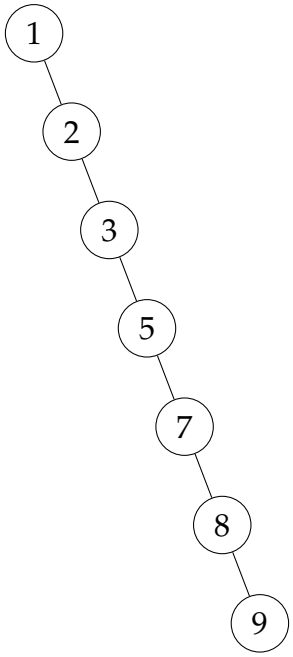
Attention : vérifier uniquement que `gauche.valeur < valeur` et `droit.valeur > valeur` ne suffit pas ! Il faut vérifier que *tous* les nœuds respectent la contrainte.

Corrigé de l'exercice 3. 1. Insertion de 5, 2, 8, 1, 3, 7, 9 :



Arbre équilibré de hauteur 2. La recherche est en $O(\log n)$.

2. Insertion de 1, 2, 3, 5, 7, 8, 9 :



L'arbre dégénère en une liste chaînée (hauteur = $n - 1 = 6$). La recherche est en $O(n)$, ce qui annule l'avantage de l'ABR.

C'est pourquoi on utilise des arbres équilibrés (AVL, arbres rouge-noir) dans les applications réelles.