Arbres

Structures de données hiérarchiques

1 Introduction aux arbres

Un **arbre** est une structure de données hiérarchique composée de **nœuds** reliés par des **arêtes**. Un arbre ne contient aucun **cycle** : entre deux nœuds, il existe un seul chemin. On verra dans le chapitre sur les graphes qu'un arbre est un graphe connexe (d'un seul tenant) et sans cycle.

Exemple. On retrouve des arbres dans de nombreuses situations :

a) L'arborescence d'un **système de fichiers**
b) Un **arbre généalogique**
c) La structure d'un **document HTML** (DOM)
d) L'organisation hiérarchique d'une **entreprise**

2 Vocabulaire des arbres

Nœuds particuliers

Dans un arbre, on distingue trois types de nœuds selon leur position :

— la **racine** : unique nœud sans parent, situé au sommet de l'arbre ;

— les **feuilles** : nœuds sans fils, situés aux extrémités ;

— les **nœuds internes** : nœuds ayant au moins un fils.

Relations entre nœuds

Les relations s'expriment avec un vocabulaire familial :

— Le **père** (ou parent) d'un nœud est le nœud directement au-dessus de lui.

— Les **fils** (ou enfants) d'un nœud sont les nœuds directement en dessous.

Caractéristiques chiffrées

Plusieurs mesures permettent de caractériser la structure :

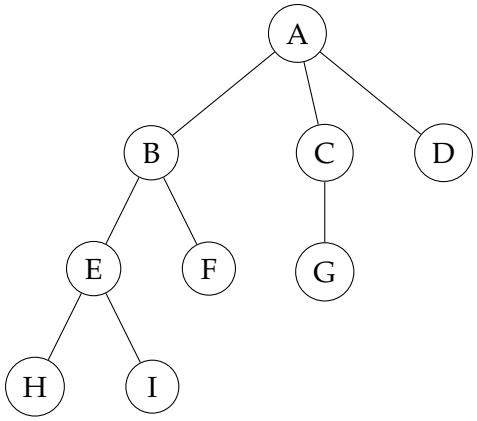
— Le **degré** d'un nœud est son nombre de fils.

— La **profondeur** d'un nœud est le nombre d'arêtes entre la racine et ce nœud. La racine a une profondeur de 0.

— La **hauteur** d'un arbre est la profondeur maximale de ses feuilles. Un arbre vide a une hauteur de -1, un arbre réduit à sa racine a une hauteur de 0.

— La **taille** d'un arbre est son nombre total de nœuds.

Exercice 1 *Analyse d'un arbre* Soit l'arbre suivant :



1. Identifier les nœuds particuliers :

   Racine :

   Feuilles :

   Nœuds internes :

2. Identifier les relations :

   Père de E :

   Fils de B :

3. Calculer les caractéristiques :

   Degré de A : Degré de B :

   Profondeur de G : Profondeur de I :

   Hauteur : Taille :

③ Arbres binaires

Un **arbre binaire** est un arbre où chaque nœud possède **au plus deux fils**, appelés **fils gauche** et **fils droit**.

C'est une structure définie de manière récursive. Un arbre binaire peut être :

— **vide** (aucun nœud) ;

— **non vide** : composé d'une racine et de deux sous-arbres (gauche et droit), eux-mêmes des arbres binaires.

3.1 Interface d'un arbre binaire

Opération	Description	Dans l'implémentation ci-dessous
`est_vide(a)`	Teste si l'arbre est vide	`a is None`
`racine(a)`	Renvoie la valeur de la racine	`a.valeur`
`gauche(a)`	Renvoie le sous-arbre gauche	`a.gauche`
`droit(a)`	Renvoie le sous-arbre droit	`a.droit`
`est_feuille(a)`	Teste si la racine est une feuille	`a.est_feuille()`

L'**interface** décrit les opérations disponibles sur un arbre binaire, indépendamment de la façon dont il est programmé ; l'**implémentation** ci-dessous les réalise avec une classe `Arbre` dont les attributs sont `valeur`, `gauche` et `droit`, l'arbre vide étant représenté par `None`.

Précondition : les opérations `racine`, `gauche`, `droit` et `est_feuille` ne s'appliquent qu'à un arbre **non vide**. Une autre implémentation garderait la même interface : les algorithmes écrits avec ces opérations resteraient valables.

3.2 Implémentation en Python

```python
class Arbre:
    """Implementation d'un arbre binaire."""

    def __init__(self, valeur, gauche=None, droit=None):
        """Constructeur : cree un noeud avec ses sous-arbres."""
        self.valeur = valeur
        self.gauche = gauche  # Sous-arbre gauche (Arbre ou None)
        self.droit = droit    # Sous-arbre droit (Arbre ou None)

    def __repr__(self):
        """Affiche l'arbre sous la forme : valeur-(gauche,droit)"""
        if self.gauche is None and self.droit is None:
            return str(self.valeur)
        return str(self.valeur) + "-(" + str(self.gauche) + "," + str(self.droit) + ")"

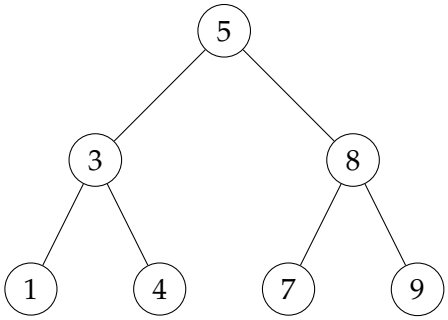
    def est_vide(self):
        """Renvoie True si l'arbre est vide."""
        return False  # Un objet Arbre n'est jamais vide

    def est_feuille(self):
        """Renvoie True si le noeud est une feuille."""
        return self.gauche is None and self.droit is None
```

Remarque. Dans cette implémentation, un arbre vide est représenté par `None`, et un arbre non vide par un objet `Arbre`. La méthode `est_vide()` renvoie toujours `False` car elle n'est appelée que sur un objet existant.

Exercice 2 *Construction d'un arbre binaire*

1. Écrire le code Python pour construire l'arbre binaire suivant :



2. Que renvoient les expressions suivantes ?

`a.valeur` :

`a.gauche.valeur` :

`a.droit.gauche.valeur` :

`a.gauche.est_feuille()` :

`a.gauche.gauche.est_feuille()` :

④ Algorithmes récursifs sur les arbres

La structure récursive des arbres binaires permet d'écrire des algorithmes élégants. Pour chaque fonction, on traitera d'abord le cas de base (arbre vide), puis l'étape de récurrence.

Exercice 3 *Implémentation de la taille* Compléter la fonction `taille(arbre)` :

```
def taille(arbre):
    """Renvoie le nombre de noeuds de l'arbre."""
    if arbre is None:
        return .....
    return ..........................................................................................
```

Exercice 4 *Implémentation de la hauteur* Compléter la fonction `hauteur(arbre)` :

```
def hauteur(arbre):
    """Renvoie la hauteur de l'arbre."""
    if arbre is None:
        return .....
    return ..........................................................................................
```

Exercice 5 *Nombre de feuilles* Écrire une fonction `nb_feuilles(arbre)` qui renvoie le nombre de feuilles.

5 Parcours d'arbres binaires

Un **parcours** permet de visiter tous les nœuds d'un arbre dans un certain ordre.

5.1 Parcours en profondeur (DFS)

On distingue trois types de parcours selon l'ordre de traitement de la racine.

— **Préfixe** : nœud, gauche, droit — on traite le nœud **avant** ses fils.

— **Infixe** : gauche, nœud, droit — on traite le nœud **entre** ses fils.

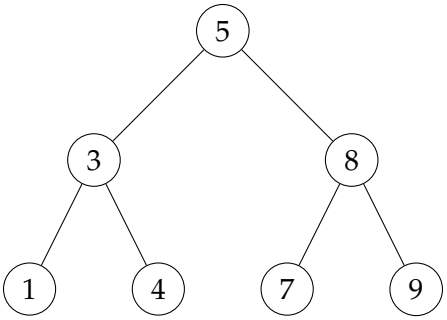
— **Suffixe** : gauche, droit, nœud — on traite le nœud **après** ses fils.

```python
def prefixe(arbre):
    """Parcours prefixe : noeud, gauche, droit."""
    if arbre is not None:
        print(arbre.valeur, end=' ')  # Traiter
        prefixe(arbre.gauche)         # Gauche
        prefixe(arbre.droit)          # Droit

def infixe(arbre):
    """Parcours infixe : gauche, noeud, droit."""
    if arbre is not None:
        infixe(arbre.gauche)          # Gauche
        print(arbre.valeur, end=' ')  # Traiter
        infixe(arbre.droit)           # Droit

def suffixe(arbre):
    """Parcours suffixe : gauche, droit, noeud."""
    if arbre is not None:
        suffixe(arbre.gauche)         # Gauche
        suffixe(arbre.droit)          # Droit
        print(arbre.valeur, end=' ')  # Traiter
```

Exercice 6 *Parcours en profondeur* Donner l'ordre de visite pour chaque parcours de l'arbre suivant :



Préfixe :

Infixe :

Suffixe :

5.2 Parcours en largeur (BFS)

Le parcours en largeur visite les nœuds **niveau par niveau**, de gauche à droite. Il utilise une **file** (FIFO : First In, First Out).

```python
def largeur(arbre):
    """Parcours en largeur avec une file."""
    if arbre is None:
        return
    file = [arbre]  # Initialisation avec la racine
    while file != []:
        noeud = file.pop(0)  # Defiler (premier element)
        print(noeud.valeur, end=' ')
        if noeud.gauche is not None:
            file.append(noeud.gauche)  # Enfiler fils gauche
        if noeud.droit is not None:
            file.append(noeud.droit)   # Enfiler fils droit
```

Exercice 7 *Trace du parcours en largeur* Compléter la trace d'exécution du parcours en largeur sur l'arbre précédent :

Étape	Contenu de la file	Nœud traité
0	[5]	—
1		5
2		
3		
4		
5		
6		
7		

⑥ Arbres binaires de recherche (ABR)

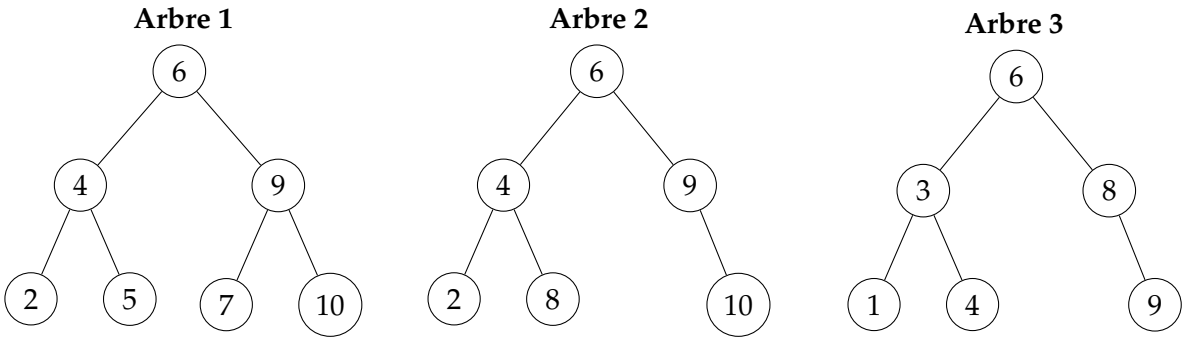
Un **ABR** est un arbre binaire tel que pour **tout** nœud :

— toutes les valeurs du sous-arbre **gauche** sont **strictement inférieures** à la valeur du nœud ;

— toutes les valeurs du sous-arbre **droit** sont **strictement supérieures** à la valeur du nœud.

Une propriété fondamentale est que le parcours **infixe** d'un ABR donne les valeurs dans l'**ordre croissant**.

Exercice 8 *Reconnaissance d'ABR* Parmi ces arbres, lesquels sont des ABR ? Justifier.



6.1 Recherche dans un ABR

```python
def rechercher(arbre, x):
    """Recherche x dans l'ABR. Renvoie True si trouve, False sinon."""
    if arbre is None:
        return False
    if x == arbre.valeur:
        return True
    elif x < arbre.valeur:
        return rechercher(arbre.gauche, x)
    else:
        return rechercher(arbre.droit, x)
```

Complexité

La recherche suit un seul chemin de la racine vers une feuille : son coût dépend donc de la hauteur de l'arbre. Un arbre est **équilibré** lorsque, pour chaque nœud, les hauteurs des sous-arbres gauche et droit sont proches ; sa hauteur est alors de l'ordre de $\log_2 n$.

— **Arbre équilibré** : $O(\log n)$, car on élimine environ la moitié des nœuds restants à chaque étape.

— **Pire cas** (arbre dégénéré) : $O(n)$, car l'arbre se comporte comme une liste.

6.2 Insertion dans un ABR

Exercice 9 Écrire la fonction `inserer(arbre, x)` qu insère l'élément `x` dans l'ABR `arbre`.

Exercice 10 ***Construction d'ABR*** On insère successivement les valeurs 5, 2, 8, 1, 3, 7, 9 dans un ABR initialement vide.

1. Dessiner l'ABR obtenu après chaque insertion :

2. Quelle serait la forme de l'arbre si on insérait les valeurs dans l'ordre 1, 2, 3, 5, 7, 8, 9 ?

3. Comparer les hauteurs des deux arbres et expliquer la conséquence sur les performances.