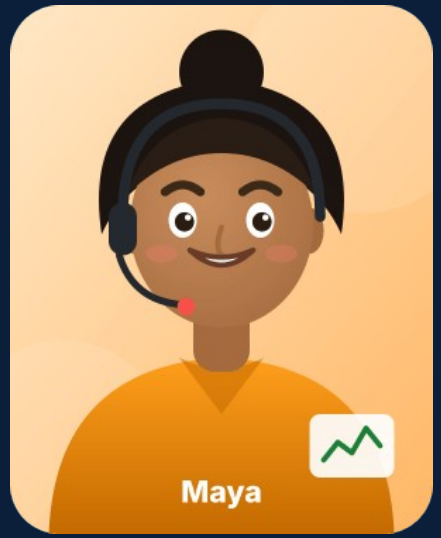
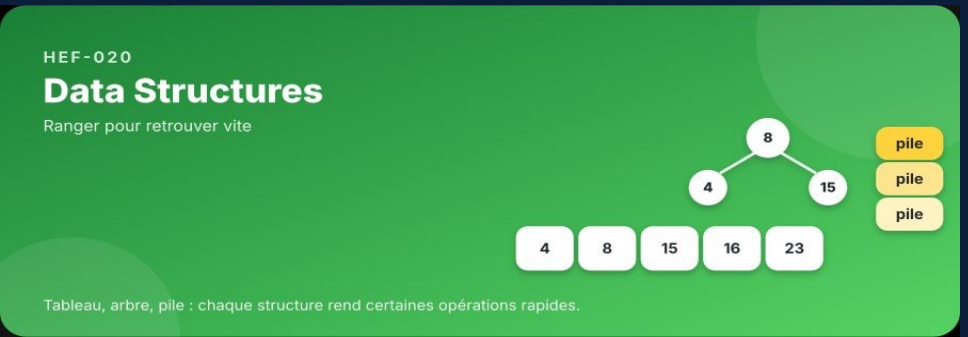
HEF-020 · MISSION 06 / 10

Arbres et arbres de recherche

Quand les données sont organisées en niveaux, un arbre s'impose.





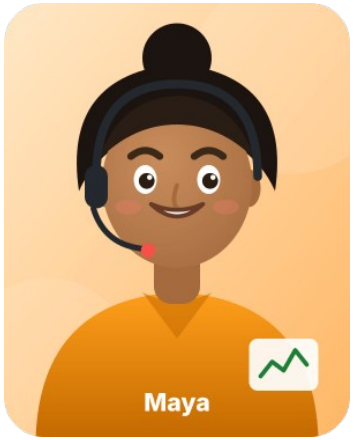
Badge : Jardinier de données

80 min · débutant

L'histoire

Le catalogue de formations de HashBox est organisé en catégories, sous-catégories, cours, chapitres. Kodi veut aussi un **annuaire trié** qui reste rapide même quand on ajoute des membres en continu.

> *« Les deux problèmes se résolvent avec des arbres, dit Sam. Un arbre général pour la hiérarchie, un arbre de recherche pour l'annuaire. »*



Tu découvres une structure qui te servira dans presque tous les domaines de l'informatique.

Ta mission

À la fin, je serai capable de…

1. décrire un arbre avec racine, nœud, feuille, hauteur
2. parcourir un arbre récursivement
3. expliquer la règle d'un arbre binaire de recherche
4. insérer et rechercher dans un BST
5. expliquer pourquoi un BST déséquilibré devient lent

Défi d'échauffement

Défi d'échauffement

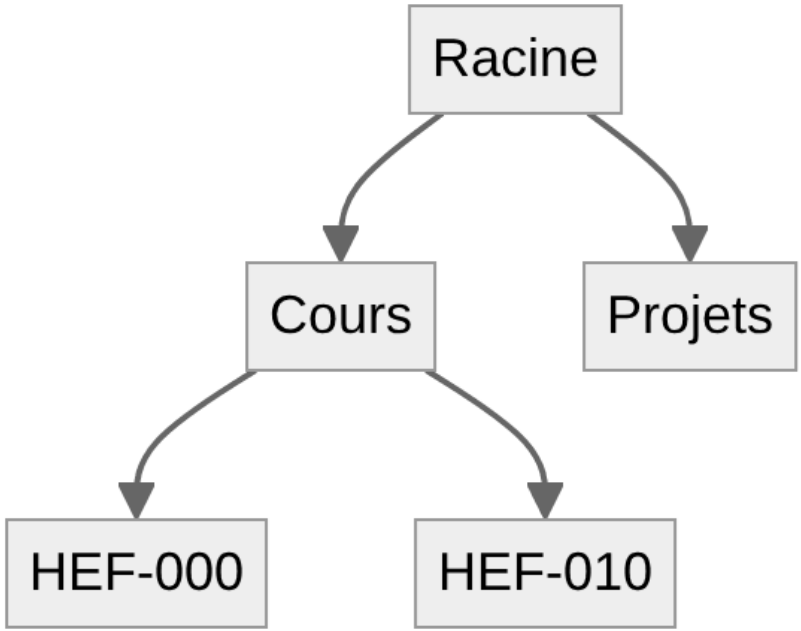
Dessine l'arbre généalogique de ta famille (sur 3 générations ou plus). Qui est la « racine » ? Qui sont les « feuilles » ? Comment compterais-tu tous les descendants d'une personne ?

Le concept

Vocabulaire

Un **arbre** est composé de **nœuds** reliés. Il y a une **racine** (en haut) et chaque nœud a des **enfants**. Un nœud sans enfant est une **feuille**. La **hauteur** est la longueur du plus long chemin de la racine à une feuille.

Vocabulaire (suite)



Le concept

Vocabulaire (suite)

Description du schéma : la racine a deux enfants, « Cours » et « Projets ». « Cours » a deux enfants, HEF-000 et HEF-010, qui sont des feuilles.

Parcours récursif

Un arbre est une structure **récursive** : chaque enfant est lui-même un arbre. Les algorithmes s'écrivent naturellement avec la récursion.

```
class Noeud:
    def __init__(self, valeur, enfants=None):
        self.valeur = valeur
        self.enfants = enfants or []

def compter(noeud):
    return 1 + sum(compter(e) for e in noeud.enfants)

def hauteur(noeud):
```

Le concept

Parcours récursif (suite)

```
    if not noeud.enfants:
        return 0
    return 1 + max(hauteur(e) for e in noeud.enfants)
```

Arbre binaire de recherche (BST)

Un **arbre binaire** a au plus 2 enfants (gauche, droit). Un **BST** respecte une règle :

> *Pour chaque nœud, tout ce qui est dans le sous-arbre de **gauche** est **plus petit**, tout ce qui est à **droite** est **plus grand**.*

Cette règle permet de chercher comme en recherche binaire : à chaque nœud, on élimine la moitié.

Le concept

Arbre binaire de recherche (BST) (suite)

```
class Arbre:
    def __init__(self):
        self.racine = None

    def inserer(self, valeur):
        self.racine = self._inserer(self.racine, valeur)

    def _inserer(self, noeud, valeur):
        if noeud is None:
            return [valeur, None, None]    # [valeur, gauche, droite]
        if valeur < noeud[0]:
            noeud[1] = self._inserer(noeud[1], valeur)
        elif valeur > noeud[0]:
            noeud[2] = self._inserer(noeud[2], valeur)
        return noeud

    def contient(self, valeur):
        n = self.racine
        while n is not None:
```

Le concept

Arbre binaire de recherche (BST) (suite)

```
            if valeur == n[0]:
                return True
            n = n[1] if valeur < n[0] else n[2]
        return False
```

Parcours en ordre : trier gratuitement

Parcourir gauche → nœud → droite d'un BST donne les valeurs **dans l'ordre croissant**.

Le danger : l'arbre déséquilibré

Si on insère `1, 2, 3, 4, 5` dans cet ordre, l'arbre devient une chaîne : la hauteur est n et la recherche redevient lente. Les **arbres équilibrés** (AVL, rouge-noir) corrigent cela automatiquement ; on retient l'idée sans les implémenter.

Exemple pas à pas

Exemple pas à pas

Problème : construire un BST avec `[50, 30, 70, 20, 40, 60, 80]` et lister les valeurs triées.

```
class Arbre:
    def __init__(self):
        self.racine = None

    def inserer(self, valeur):
        self.racine = self._inserer(self.racine, valeur)

    def _inserer(self, noeud, valeur):
        if noeud is None:
            return [valeur, None, None]
        if valeur < noeud[0]:
            noeud[1] = self._inserer(noeud[1], valeur)
        elif valeur > noeud[0]:
            noeud[2] = self._inserer(noeud[2], valeur)
        return noeud
```

Exemple pas à pas (suite)

```
    def en_ordre(self):
        resultat = []
        def visiter(n):
            if n is None:
                return
            visiter(n[1])
            resultat.append(n[0])
            visiter(n[2])
        visiter(self.racine)
        return resultat

a = Arbre()
for v in (50, 30, 70, 20, 40, 60, 80):
    a.inserer(v)
print(a.en_ordre())   # [20, 30, 40, 50, 60, 70, 80]
```

Arbre obtenu :

Exemple pas à pas

Exemple pas à pas (suite)



Pour chercher 60 : 50 → à droite → 70 → à gauche → 60. **3 comparaisons** pour 7 valeurs.

À toi, avec aide

Écris deux fonctions sur l'arbre `Arbre` de l'exemple :

1. `hauteur()` : hauteur de l'arbre (arbre vide : -1, un seul nœud : 0).
2. `minimum()` : la plus petite valeur (aller toujours à gauche).

Teste avec les valeurs `[50, 30, 70, 20, 40, 60, 80]` (hauteur 2, minimum 20) puis avec `[1, 2, 3, 4]` (hauteur 3).

À toi, avec aide · réponse

Indices progressifs

Indice 1

La hauteur d'un nœud vaut 1 + max(hauteur gauche, hauteur droite).

Indice 2

Pour un nœud vide, renvoie -1.

Indice 3

Le minimum : descends à gauche tant que possible.

À toi, avec aide · réponse

Réponse

```
class Arbre:
    def __init__(self):
        self.racine = None

    def inserer(self, valeur):
        self.racine = self._inserer(self.racine, valeur)

    def _inserer(self, noeud, valeur):
        if noeud is None:
            return [valeur, None, None]
        if valeur < noeud[0]:
            noeud[1] = self._inserer(noeud[1], valeur)
        elif valeur > noeud[0]:
            noeud[2] = self._inserer(noeud[2], valeur)
        return noeud

    def hauteur(self):
        def h(n):
            if n is None:
```

Réponse (suite)

```
                return -1
            return 1 + max(h(n[1]), h(n[2]))
        return h(self.racine)

    def minimum(self):
        n = self.racine
        while n[1] is not None:
            n = n[1]
        return n[0]

a = Arbre()
for v in (50, 30, 70, 20, 40, 60, 80):
    a.inserer(v)
print(a.hauteur(), a.minimum())   # 2 20

b = Arbre()
for v in (1, 2, 3, 4):
    b.inserer(v)
print(b.hauteur())                # 3
```

À toi, en solo

Dans l'arbre `Noeud` (enfants multiples), écris :

1. `chemin(noeud, cible)` : renvoie la liste des valeurs de la racine jusqu'à la cible, ou `None` si absente ;
2. `afficher(noeud)` : affiche l'arbre avec une indentation par niveau.

Teste sur l'arbre du catalogue : racine « Catalogue », enfants « Fondations » (HEF-000, HEF-010) et « Systèmes » (HEF-040).

À toi, en solo · réponse

Pistes de correction

```
class Noeud:
    def __init__(self, valeur, enfants=None):
        self.valeur = valeur
        self.enfants = enfants or []

def chemin(noeud, cible):
    if noeud.valeur == cible:
        return [noeud.valeur]
    for e in noeud.enfants:
        c = chemin(e, cible)
        if c is not None:
            return [noeud.valeur] + c
    return None

def afficher(noeud, niveau=0):
    print("  " * niveau + str(noeud.valeur))
    for e in noeud.enfants:
        afficher(e, niveau + 1)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
catalogue = Noeud("Catalogue", [
    Noeud("Fondations", [Noeud("HEF-000"), Noeud("HEF-010")]),
    Noeud("Systèmes", [Noeud("HEF-040")]),
])
print(chemin(catalogue, "HEF-040"))   # ['Catalogue', 'Systèmes', 'HEF-040']
afficher(catalogue)
```

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Oublier le cas de l'arbre vide	Erreur sur `None`	Traiter `None` en premier
Insérer dans l'ordre trié	L'arbre devient une chaîne, lent	Mélanger ou utiliser un arbre équilibré
Doublons dans un BST	Règle ambiguë	Décider : ignorer, compter, ou placer à droite
Confondre hauteur et nombre de nœuds	Mauvaise estimation du coût	La hauteur est la longueur du plus long chemin

Mini-boss

Mini-boss

Écris un **annuaire de membres** basé sur un BST où chaque nœud contient `(nom, ville)`.

1. `ajouter(nom, ville)` (si le nom existe, mets à jour la ville) ;
2. `chercher(nom)` ;
3. `lister()` : tous les membres triés par nom ;
4. `entre(a, b)` : membres dont le nom est entre `a` et `b` (inclus) ;
5. Compare la hauteur obtenue en insérant 500 noms triés et 500 noms mélangés.

Mini-boss · réponse

Pistes de correction

Un bon travail réutilise la structure du BST avec un nœud `[cle, valeur, gauche, droite]`. `entre(a, b)` fait un parcours en ordre qui ne descend pas dans les sous-arbres inutiles. La comparaison montre une hauteur de ~499 pour les noms triés et de l'ordre de 20 pour les noms mélangés.

Quiz éclair

Quiz éclair

1. Dans un BST, le sous-arbre de gauche contient :

A. des valeurs plus grandes
B. des valeurs plus petites
C. des valeurs au hasard

2. Un parcours en ordre d'un BST donne :

A. les valeurs inversées
B. les feuilles
C. les valeurs triées

3. Un BST déséquilibré a une recherche :

A. lente (comme une liste)

Quiz éclair

Quiz éclair (suite)

B. très rapide
C. impossible

4. 🔁 *Révision mission 04* : Une table de hachage associe :

A. des nœuds à des feuilles
B. des clés à des valeurs
C. des listes à des tris

5. 🔁 *Révision mission 05* : `a & b` pour deux ensembles donne :

A. tous les éléments
B. les éléments de a seulement
C. les éléments communs

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Quelle hiérarchie de ta vie ressemble à un arbre ?
2. Pourquoi l'ordre d'insertion compte-t-il dans un BST ?

À retenir

À retenir

1. Arbre = nœuds, racine, feuilles, hauteur.
2. Les algorithmes d'arbres sont naturellement récursifs.
3. BST : gauche plus petit, droite plus grand.
4. Parcours en ordre = valeurs triées.
5. Un arbre déséquilibré perd son avantage.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Jardinier de données

Prochaine mission : Tas et files de priorité

