TP Chapitre 5 : Arbres (ABR)

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

— reconnaître une situation nécessitant une structure de données arborescente et utiliser le vocabulaire des arbres (racine, nœud, feuille, père, enfant, branche) ;

— définir la classe `Noeud` (valeur, sous-arbre gauche, sous-arbre droit) et construire un arbre binaire ;

— calculer la taille et la hauteur d'un arbre binaire à l'aide de fonctions récursives ;

— parcourir un arbre en largeur (avec une file) et en profondeur (préfixe, infixe, postfixe) ;

— reconnaître et vérifier un arbre binaire de recherche (ABR) ;

— rechercher une clé dans un ABR et y insérer une clé (fonctions récursives) ;

— expliquer le coût d'une recherche dans un ABR : logarithmique si l'arbre est équilibré, linéaire s'il est dégénéré.

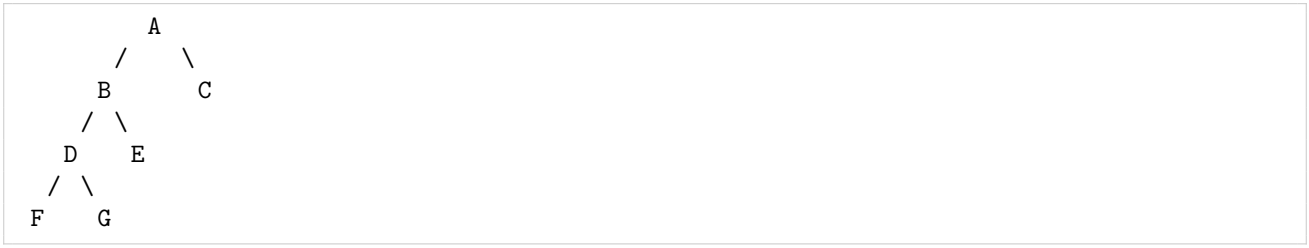
Consignes générales. Tous les exercices se traitent en Python (console ou fichier). La classe `Noeud` utilisée dans tout le TP est celle du cours : chaque nœud possède une valeur, un sous-arbre gauche et un sous-arbre droit, un sous-arbre vide étant représenté par `None`. Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

Exercice A1 — L'arbre de décision d'achat (vocabulaire et classe `Noeud`)

Un site de comparaison d'achats propose un arbre de décision : chaque nœud est une question, chaque feuille une décision d'achat. L'arbre ci-dessous est étiqueté par des lettres ; la légende donne la signification économique de chaque étiquette.



Légende : A : « Acheter ? » ; B : « Budget OK » ; C : « Besoin faible » ; D : « Besoin réel ? » ; E : « Attendre » ; F : « Acheter » ; G : « Louer ».

1. Quelle est la racine de cet arbre ?
2. Quelles sont les feuilles de cet arbre ?
3. Quel est le père du nœud E ? Quels sont les enfants du nœud D ?
4. Quelle est la taille de cet arbre ? Quelle est sa hauteur en nombre d'arêtes ? Combien de niveaux compte-t-il ?

5. Écrire la classe `Noeud` utilisée dans tout le chapitre : un nœud possède une valeur, un sous-arbre gauche et un sous-arbre droit (un sous-arbre vide est représenté par `None`).

```
class Noeud:
    def __init__(self):
        self.valeur = None
        # A COMPLETER : sous-arbre gauche et sous-arbre droit
```

Point de validation

La racine est le nœud A ; les feuilles sont C, E, F et G ; le père de E est B ; les enfants de D sont F et G. La taille vaut 7, la hauteur en arêtes vaut 2 (3 niveaux). La classe `Noeud` complétée doit créer des nœuds avec trois attributs `valeur`, `gauche` et `droite`, tous initialisés à `None`.

Exercice A2 — Le catalogue arborescent de prix (taille et hauteur)

Une librairie en ligne range les prix de ses ouvrages (en euros) dans un arbre binaire : chaque nœud porte un prix, et les sous-arbres regroupent les prix des sous-catégories. Voici le catalogue, construit avec la classe `Noeud` (c'est l'arbre de référence du cours).

```
        1
      /   \
     2     3
    / \     \
   4   5     6
      / \   /
     7   8 9
```

1. Écrire la fonction récursive `taille(Arbre)` qui renvoie le nombre de nœuds d'un arbre binaire. On rappelle : la taille d'un arbre vide vaut 0 ; celle d'un arbre non vide vaut 1 plus la taille de ses deux sous-arbres.

```
def taille(Arbre):
    if Arbre == None:
        return 0
    else:
        # A COMPLETER : 1 + taille(sous-arbre gauche) + taille(sous-
            arbre droit)
```

2. Écrire la fonction récursive `hauteur(Arbre)` — convention du cours : un arbre vide a une hauteur nulle, un arbre non vide a pour hauteur 1 plus la hauteur maximale de ses deux sous-arbres (la racine compte pour 1 niveau).

```
def hauteur(Arbre):
    if Arbre == None:
        return 0
    else:
        # A COMPLETER : 1 + max(hauteur(gauche), hauteur(droite))
```

3. Combien d'arêtes compte le plus long chemin de la racine à une feuille de ce catalogue ?

Point de validation

`taille(Arbre)` doit renvoyer `9` ; `hauteur(Arbre)` doit renvoyer `4` (la fonction compte les niveaux, racine comprise). Le plus long chemin comporte 3 arêtes.

Exercice A3 — Les transactions du jour (parcours infixe et préfixe)

Un commerçant enregistre chaque jour les montants de ses ventes (en euros) dans un arbre binaire de recherche : chaque nœud est un montant, tous les montants du sous-arbre gauche sont inférieurs ou égaux, tous ceux du sous-arbre droit sont supérieurs ou égaux. Voici l'ABR des ventes du jour, construit avec la classe `Noeud`.

```
          13
        /    \
      11       14
     /  \
    9    12
   / \
  0   10
```

1. Écrire la fonction récursive `parcours_infixe(Arbre)` qui affiche les valeurs une par une, dans l'ordre infixe (GND : sous-arbre gauche, nœud, sous-arbre droit).

```
def parcours_infixe(Arbre):
    if Arbre != None:
        # A COMPLETER : sous-arbre gauche, affichage, sous-arbre droit
```

2. Écrire la fonction récursive `parcours_prefixe(Arbre)` qui affiche les valeurs dans l'ordre préfixe (NGD : nœud, sous-arbre gauche, sous-arbre droit).

```
def parcours_prefixe(Arbre):
    if Arbre != None:
        # A COMPLETER : affichage, sous-arbre gauche, sous-arbre droit
```

3. Que constate-t-on sur l'affichage du parcours infixe ? Comment l'expliquer avec la définition d'un ABR ?

Point de validation

Le parcours infixe doit afficher : 0, 9, 10, 11, 12, 13, 14 : les montants sont triés dans l'ordre croissant (propriété des ABR). Le parcours préfixe doit afficher : 13, 11, 9, 0, 10, 12, 14.

Exercice A4 — Retrouver un prix dans le catalogue (recherche dichotomique)

Un client demande si un montant figure dans l'ABR des ventes du jour (exercice A3). La recherche suit le principe de la recherche dichotomique : on compare le montant cherché à la valeur du nœud courant ; s'ils sont égaux, la clé est trouvée ; sinon on descend dans le sous-arbre gauche ou droit — à chaque étape, un sous-arbre entier est écarté.

Écrire la fonction récursive `rechercher(Arbre, element)` qui renvoie `True` si `element` figure dans l'arbre, `False` sinon.

```
def rechercher(Arbre, element):
    if Arbre == None:
        return False
    else:
        x = Arbre.valeur
        if x == element:
            return True
        elif element < x:
            # A COMPLETER : continuer dans le sous-arbre gauche
        else:
            # A COMPLETER : continuer dans le sous-arbre droit
```

Point de validation

Sur l'ABR de l'exercice A3 : `rechercher(ABR, 12)` doit renvoyer `True`, `rechercher(ABR, 1)` `False`, `rechercher(ABR, 14)` `True`, `rechercher(ABR, 7)` `False`.

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

Exercice B1 — Un catalogue bien rangé ? (vérifier qu'un arbre est un ABR)

Une boutique en ligne stocke les prix de ses articles (en euros) dans un arbre binaire. Avant d'autoriser la recherche, le programme doit vérifier que l'arbre est bien un arbre binaire de recherche : chaque nœud doit avoir une valeur strictement supérieure à celle de son fils gauche (si ce fils existe) et strictement inférieure à celle de son fils droit.

Énoncé. Écrire un programme complet qui définit la classe `Noeud`, construit l'arbre des prix de la boutique ci-dessous, puis vérifie avec une fonction récursive `est_de_recherche(Arbre)` que cet arbre est un ABR, et affiche le résultat.

```
        20
      /    \
    10      30
   /  \    /  \
  5   15  25  35
```

Spécification.

— *Entrées* : aucune saisie clavier ; l'arbre est construit dans le programme (valeurs : 20, 10, 30, 5, 15, 25, 35).

— *Traitement* : fonction récursive `est_de_recherche` : un arbre vide est un ABR ; sinon on vérifie la valeur du nœud par rapport à ses deux fils, puis on vérifie récursivement les deux sous-arbres.

— *Sorties* : `True` si l'arbre est un ABR, `False` sinon.

— *Contraintes* : classe `Noeud` ; fonction récursive ; aucun `input()`.

Pour t'aider — questions de conception (sans donner le code)

— Que doit renvoyer `est_de_recherche` pour un arbre vide ? Pourquoi ?

— Pourquoi comparer chaque nœud à ses deux fils directs suffit-il ici ? (la version du cours ne vérifie la propriété que localement)

— Que renverrait la fonction si l'on échangeait les fils gauche et droit du nœud 30 ?

Point de validation

Le programme doit afficher `True`.

Exercice B2 — Enregistrer les transactions du jour (`inserer` et parcours infixe)

Chaque soir, un commerçant enregistre les montants de ses ventes du jour (en euros) dans un arbre binaire de recherche, pour pouvoir les consulter triés. Les montants arrivent dans l'ordre chronologique.

Énoncé. Écrire un programme complet qui définit la classe `Noeud`, la fonction `inserer` (version fonctionnelle : elle renvoie un nouvel arbre, sans modifier l'arbre donné), construit un ABR en insérant successivement les montants de la liste `[120, 45, 200, 30, 90, 150, 250, 60]` (en euros), puis affiche les montants dans l'ordre croissant grâce au parcours infixe.

Spécification.

— *Entrées* : aucune saisie clavier ; la liste des montants est fixée dans le programme.

— *Traitement* : pour chaque montant de la liste, `abr = inserer(montant, abr)` en partant de `abr = None` ; puis affichage par parcours infixe.

— *Sorties* : les montants dans l'ordre croissant, un par ligne.

— *Contraintes* : version fonctionnelle de `inserer` (l'arbre donné n'est pas modifié) ; parcours infixe récursif ; montants entiers.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi `inserer` doit-elle renvoyer un nouvel arbre ? Que deviendrait `abr` si l'on oubliait de récupérer la valeur de retour ?

— Que devient l'arbre d'origine après un appel à `inserer` ?

— Pourquoi le parcours infixe affiche-t-il les montants dans l'ordre croissant ?

— Si les montants arrivaient déjà triés, quelle forme prendrait l'arbre ? (penser à la remarque du cours)

Point de validation

Le programme doit afficher : 30, 45, 60, 90, 120, 150, 200, 250 (un montant par ligne, dans cet ordre).

Exercice B3 — Retrouver un montant et compter les comparaisons

Le commerçant veut savoir si un montant figure dans le registre de la journée, et combien de comparaisons la recherche a coûtées : c'est le coût de la recherche dans l'ABR.

Énoncé. Écrire un programme complet qui définit la classe `Noeud`, construit le même ABR que l'exercice B2 (liste `[120, 45, 200, 30, 90, 150, 250, 60]`), et définit une fonction `rechercher_comparaisons(Arbre, element)` qui renvoie un couple `(trouve, nb_comparaisons)` : `trouve` vaut `True` si la clé est dans l'arbre, `False` sinon ; `nb_comparaisons` compte les comparaisons effectuées (chaque comparaison entre `element` et la valeur d'un nœud visité compte pour 1). Le programme effectue trois recherches — 90, 250 et 130 — et affiche pour chacune le résultat et le nombre de comparaisons.

Spécification.

— *Entrées* : aucune saisie clavier ; l'ABR est construit avec la liste ci-dessus.

— *Traitement* : fonction récursive renvoyant un couple `(trouve, nb)` ; trois recherches (90, 250, 130) ; affichage de chaque résultat.

— *Sorties* : pour chaque recherche, le montant, `True`/`False` et le nombre de comparaisons.

— *Contraintes* : récursivité ; le nombre de comparaisons doit être renvoyé (par exemple dans un couple), pas affiché au fil de l'eau ; pas de variable globale.

Pour t'aider — questions de conception (sans donner le code)

— Que doit renvoyer la fonction pour un arbre vide ? (combien de comparaisons ?)

— Comment transmettre le nombre de comparaisons d'un appel récursif à l'autre ? Pourquoi une variable globale est-elle déconseillée ?

— Pour un ABR de hauteur h (nombre de niveaux), combien de comparaisons la recherche effectue-t-elle au plus ?

Point de validation

Recherche de 90 : `True`, 3 comparaisons ; recherche de 250 : `True`, 3 comparaisons ; recherche de 130 : `False`, 3 comparaisons.

Exercice B4 — La hiérarchie des dépenses (parcours en largeur)

Un foyer organise son budget dans un arbre binaire : la racine est le poste « Total », chaque nœud interne est une catégorie de dépense, chaque feuille un sous-poste. Pour un tableau de bord, on veut afficher les dépenses étage par étage, de haut en bas et de gauche à droite : c'est le parcours en largeur.

```
            Total
          /       \
     Logement     Alimentation
     /     \       /        \
  Loyer   EDF  Courses    Transport
                          /       \
                     Essence   Assurance
```

Le nœud `Transport`, fils droit de `Alimentation`, a pour enfants `Essence` et `Assurance`.

Énoncé. Écrire un programme complet qui définit la classe `Noeud`, construit l'arbre du budget ci-dessus (les étiquettes sont des chaînes de caractères), définit la classe `File` (structure FIFO vue au chapitre 2, rappelée dans le cours), puis affiche les étiquettes dans l'ordre du parcours en largeur à l'aide d'une file.

Spécification.

— *Entrées* : aucune saisie clavier ; l'arbre est construit dans le programme.

— *Traitement* : classe `File` (`enfiler`, `defiler`, `est_vide`) ; parcours en largeur : enfiler la racine, puis tant que la file n'est pas vide : défiler un nœud, l'afficher, enfiler ses enfants non vides.

— *Sorties* : une étiquette par ligne, dans l'ordre du parcours en largeur.

— *Contraintes* : structure FIFO ; on n'enfile que les enfants non vides (`None` exclus) ; l'affichage a lieu au moment du défilement.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi une file (FIFO) et pas une pile pour ce parcours ? Que donnerait une pile ?

— Pourquoi faut-il enfiler les enfants d'un nœud avant de défiler le nœud suivant ?

— Que se passe-t-il si l'on enfile un enfant vide (`None`) sans le tester ?

— Quel intérêt pour le tableau de bord d'afficher les dépenses étage par étage plutôt qu'en profondeur ?

Point de validation

Le programme doit afficher, dans cet ordre : `Total`, `Logement`, `Alimentation`, `Loyer`, `EDF`, `Courses`, `Transport`, `Essence`, `Assurance`.

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

Registre de transactions : l'ordre d'arrivée fait-il le coût ?

Contexte. Un artisan enregistre chaque jour les montants (en euros) de ses ventes dans un arbre binaire de recherche, pour retrouver rapidement si un montant a déjà été encaissé (litige, double paiement…). Les montants arrivent dans l'ordre chronologique, au fil de la journée. Une question se pose : la forme de l'arbre, et donc le coût de la recherche, dépend-elle de l'ordre d'arrivée des montants ?

Objectif. Écrire un programme qui construit trois ABR contenant exactement les mêmes montants — `[200, 50, 300, 25, 120, 250, 400, 30, 90]` (en euros) — mais insérés dans trois ordres différents :

1. l'ordre chronologique : `[200, 50, 300, 25, 120, 250, 400, 30, 90]` (insertions successives avec `inserer`) ;
2. l'ordre trié croissant : `[25, 30, 50, 90, 120, 200, 250, 300, 400]` (insertions successives) ;
3. un ordre équilibré : insérer d'abord la médiane (120), puis les médianes des sous-listes restantes, et ainsi de suite — méthode « diviser pour régner ».

Pour chacun des trois arbres, le programme affiche : sa hauteur (nombre de niveaux, fonction du cours), son parcours en largeur, et le nombre de comparaisons nécessaires pour retrouver le montant 400 (fonction de l'exercice B3). Enfin, il compare les trois résultats et conclut sur l'influence de l'ordre d'arrivée.

Questions à se poser avant de coder.

— Quelles fonctions du TP réutilises-tu ? (`inserer`, `hauteur`, parcours en largeur, recherche avec compteur)

— Comment construire l'ordre équilibré par « diviser pour régner » ? (médiane d'abord, puis médianes des moitiés gauche et droite, récursivement)

— Pourquoi le parcours en largeur permet-il de « voir » la forme de l'arbre ?

— Que vaut la hauteur minimale d'un arbre binaire de 9 nœuds ? (rappeler l'encadrement $\log_2(n+1) \leq h \leq n$ du cours)

— Comment vérifier tes résultats ? (le parcours infixe doit toujours afficher la même liste triée, quel que soit l'ordre d'insertion)

Étapes suggérées.

1. Étape 1 : construire l'arbre chronologique ; afficher sa hauteur, son parcours en largeur et le nombre de comparaisons pour 400.
2. Étape 2 : construire l'arbre trié ; afficher les mêmes mesures. Constater la forme en chaîne (arbre dégénéré).
3. Étape 3 : construire l'arbre équilibré (médianes) ; afficher les mêmes mesures.
4. Étape 4 : comparer et conclure. Bonus : pour un ABR équilibré de 1 000 000 de clés, estimer le nombre de comparaisons pour rechercher une clé ($2^{10} = 1024$, $2^{20} \approx 1\,048\,576$) ; même question pour un ABR dégénéré.

Contraintes. Les montants doivent réellement être insérés un par un avec la fonction `inserer` (sauf l'ordre équilibré, construit par médianes) ; programme testé et commenté ; les messages peuvent être écrits sans accents (ex. `hauteur`, `comparaisons`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.