Numérique et sciences informatiques
Classe de terminale

II Structures de données
B Structures de données hiérarchiques
1. Les arbres binaires

I Les arbres binaires

I.1 La structure de données arbre

• On souhaite disposer d'une structure de données qui exprime des données *hiérarchisées*.

• Pour cela, nous allons définir ce que nous appellerons une structure arborescente ou un arbre. Voici quelques exemples de structure arborescente : organisation des fichiers dans un ordinateur, un livre, une page web, une formule logique, etc.

• On peut se représenter une **structure arborescente,** ou **arbre,** comme un ensemble fini d'éléments, appelés **nœuds** reliés entre eux par un *lien orienté*. C'est la façon dont ces nœuds sont reliés entre eux qui constitue une structure arborescente. Autrement dit, l'organisation des liens entre les nœuds doit vérifier certaines conditions pour qu'on puisse parler d'une structure arborescente.

• Un lien orienté entre deux nœuds permet d'attribuer des statuts différents à chacun d'entre eux :
  * le nœud de départ est appelé le **père** (ou le parent) ;
  * le nœud d'arrivé est appelé le **fils** (ou l'enfant).

• Une structure arborescente respecte les règles suivantes :
  * Un nœud donné ne peut pas avoir plus qu'un seul père mais il peut avoir plusieurs fils.
  * Tous les nœuds sauf un possède exactement un père. Celui qui ne possède pas de père s'appelle **la racine**.
  * Si on suit les liens orientés de père à fils, on finit en un nombre fini d'étapes par aboutir sur un nœud qui n'a pas de fils Un tel nœud est appelé **une feuille**.

Illustration

Remarque : On représente un arbre binaire en suivant les liens orientés de haut en bas. Paradoxalement, on place donc la racine au sommet de la représentation.

I.2 Les arbres binaires

• On va présenter ici un cas particulier de structure arborescente : les **arbres binaires**. Dans un arbre binaire, un nœud possède au maximum deux fils. On attribue un *statut différent* à chacun des deux fils éventuels en les appelant **fils gauche** et **fils droit**. Un nœud peut n'avoir qu'un seul fils qui peut être *un fils gauche ou un fils droit*.

• On peut définir un arbre binaire de façon récursive de la façon suivante. Un arbre binaire est :

* soit vide ;
* soit composé de trois éléments :
  - un noeud appelé **racine** de cet arbre ;
  - *deux arbres binaires* respectivement appelés **sous-arbre gauche** et **sous-arbre droit**. Ces deux sous-arbres peuvent être vides (tous les deux ou bien un seul d'entre eux) ou bien être formés de trois éléments. Si les deux sous arbres sont vides, alors on est arrivé à une feuille.

• Voici la représentation d'un arbre binaire : on représente la racine d'un arbre comme reliée à la racine de chacun des deux sous-arbres.

• Vocabulaire sur les arbres :

* chaque élément de l'arbre est un **nœud** d'un arbre binaire :
* chacun de ses nœuds est soit la racine de l'ensemble de l'arbre soit la racine de l'un de ses arbres de niveau inférieur ;
* notion de **fils** et **de père** d'un nœud ;
*notion de **feuille** d'un arbre binaire ;
* notion de **nœud interne** d'un arbre : un nœud qui n'est ni la racine ni une feuille de l'arbre ;
* les **descendants** et les **ancêtres** d'un nœud ;
* une **branche** de l'arbre.

• Un arbre binaire respecte les conditions suivantes :

* un arbre binaire structure un ensemble *fini* de nœuds ;
* il n'y a pas de cycle : un nœud ne peut pas être à la fois descendant et ancêtre d'un même autre nœud ;
* il n'y a pas de nœud partagé par deux sous-arbres distincts.

I.3 Le stockage de l'information dans un arbre binaire

• Un arbre binaire étant une structure de données, on « stocke de l'information » dans un arbre binaire en attribuant une **valeur** (ou une **clé)** à chaque nœud**.** Selon les arbres binaires considérés, il est possible, ou non, que deux nœuds distincts possèdent la même valeur. Il peut arriver que seules les feuilles possèdent une valeur.

• Comme pour les piles, les files et les listes chaînées, les données stockées dans un arbre binaire seront généralement supposées **homogènes**, c'est-à-dire qu'elles seront toutes de même type.

I.4 Mesures d'un arbre binaire

• On définit les notions numériques suivantes :
  * la **taille** d'un arbre : le nombre de ses nœuds ;
  * la **profondeur d'un nœud** donné : le nombre de nœuds entre la racine et ce nœud (en comptant la racine et le nœud lui-même) ;
  * la **hauteur de l'arbre** est la profondeur de sa racine.

Remarque : attention, les définitions pour la hauteur et la profondeur varient d'un manuel à l'autre. Lisez donc attentivement celle(s) donnée(s) dans votre énoncé d'exercice.

I.5 Quelques cas particuliers et un encadrement de la taille et de la hauteur

• On distinguera trois cas particuliers d'arbre binaire suivants :
  * Un **arbre dégénéré** : les nœuds n'ont jamais plus d'un seul fils.
  * Un arbre **localement complet** : les nœuds sont, soient des feuilles, soient possèdent *exactement deux* fils.
  * Un arbre **complet** : il est localement complet et toutes les feuilles ont la même hauteur.

• Exemples :

Un arbre dégénéré	un arbre localement complet	Un arbre complet

• Si on considère un arbre binaire de hauteur h et de taille N alors :

  * s'il est dégénéré alors :

$$N = h$$

  * s'il est complet alors :

$$N = 1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$

Un encadrement à connaître absolument

Dans le cas général, on a l'encadrement de N suivant :

$$\boldsymbol{h \leq N \leq 2^h - 1}$$

D'où on déduit l'encadrement de h :

$$\boldsymbol{\log_2 (N + 1) \leq h \leq N}$$

En effet la fonction logarithme de base 2 notée $\log_2$ est strictement croissante sur $]0\ ;\ +\infty[$ et pour tout réel x, on a $\log_2(2^x) = x$.

I.6 Le type abstrait Arbre Binaire

• On peut définir le type `ArbreBinaire` *de façon récursive* comme formé des attributs suivants :

* `valeur` d'un type donné **`N`** ,
* `sa_gauche` et `sa_droit` : de type **`ArbreBinaire`**.

• On lui associera les opérations pour *créer* la structure arborescente :

* `creerArbreVide()` qui renvoie un arbre vide ;
* `creerFeuille(val)` qui crée un arbre dont la racine aura pour valeur `val` et dont les deux sous-arbres seront vides ;
* `creerArbre(val, g, d)` qui crée un arbre dont la racine aura pour valeur `val` et dont les deux sous-arbres seront respectivement `g` et `d`.

• On lui ajoutera des opérations pour *lire* les informations contenues dans cette structure :

* `racine(arb)` qui renvoie la valeur de la racine de l'arbre binaire `arb`. Ceci suppose que cet arbre soit non vide ;
* `sousA_gauche(arb)` qui renvoie le sous-arbre gauche de l'arbre binaire `arb` ;
* `sousA_droit(arb)` qui renvoie le sous-arbre droit de l'arbre binaire `arb` ;
* `est_vide(arb)` qui renvoie `True` si l'arbre binaire arb est vide et `False` sinon ;

• On aura aussi besoin d'opérations pour obtenir des informations numériques indiquant *différentes mesures* de la structure :

* `taille(arb)` qui renvoie la taille de l'arbre binaire `arb`, c'est-à-dire le nombre de ses nœuds ;
* `hauteur(arb)` qui renvoie la hauteur de l'arbre binaire `arb`.

II Une implémentation d'un arbre binaire en Python

II.1 Définition des premiers éléments

• Définition de la classe et du constructeur :

```
class ArbreBinaire :
    def __init__(self, val=None, g=None, d=None):
        """ On peut créer un arbre vide. """
        self.cle = val
        self.sa_gauche = g
        self.sa_droit = d
```

Remarque : Il s'agit d'une définition de classe récursive. En effet, les attributs `sa_gauche` et `sa_droit` doivent eux-mêmes avoir pour valeur *un arbre binaire*. Ce constructeur met en œuvre dans le même temps les opérations suivantes :

* `creerArbreVide()` si l'on appelle le constructeur sans argument par une instruction de la forme

```
unArbre = ArbreBinaire()
```

* `creerFeuille(val)` si l'on appelle le constructeur avec un argument par une instruction de la forme

```
unArbre = ArbreBinaire(val)
```

* `creerArbre(val, g, d)` si l'on appelle le constructeur avec trois arguments par une instruction de la forme

```
unArbre = ArbreBinaire(val, sa_g, sa_d)
```

II.2 Méthodes de lecture de la classe `ArbreBinaire`

• On peut alors écrire de façon très simple quatre méthodes correspondant aux trois opérations de lecture décrites plus haut :

```
    def est_vide(self) :
        return self.cle == None

    def racine(self) :
        assert not self.est_vide(), "L'arbre est vide"
        return self.cle

    def gauche(self) :
        assert not self.est_vide(), "L'arbre est vide"
        return self.sa_gauche

    def droit(self) :
        assert not self.est_vide(), "L'arbre est vide"
        return self.sa_droit
```

II.3 Algorithmes récursifs pour mesurer la taille et la hauteur d'un arbre binaire

a) Ecrire des fonctions

• La fonction `taille(arb)` prend pour argument soit `None` pour un arbre vide, soit un objet de la classe `ArbreBinaire` qui représente un arbre *qui peut être vide*. On applique une programmation récursive en distinguant les deux cas suivants :

* si l'arbre est vide alors la taille est égale à 0 ;
* sinon la taille de l'arbre est égale à

1 + la taille du sous-arbre gauche + la taille du sous-arbre droit.

On obtient donc le code récursif suivant :

```
def taille(arb) :
    if arb == None or arb.est_vide() :
        return 0
    tg = taille(arb.sa_gauche)
    td = taille(arb.sa_droit)
    return 1 + tg + td
```

• De même, la fonction `hauteur(arb)` a pour argument soit `None`, soit un objet de la classe `ArbreBinaire` qui représente un arbre binaire *qui peut être vide*. On distingue deux cas :

* si l'arbre est vide la hauteur est 0 (selon la convention adoptée plus haut) ;
* si l'arbre n'est pas vide la hauteur est égale à

1 + *le maximum entre* la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit.

On obtient donc le code récursif suivant :

```
def hauteur(arb) :
    if arb == None or arb.est_vide() :
        return 0
    hg = hauteur(arb.sa_gauche)
    hd = hauteur(arb.sa_droit)
    return 1 + max(hg, hd)
```

b) Écrire des méthodes de classe

• Ecrire une définition récursive pour une *méthode de classe* (et non pas une fonction) pose une difficulté supplémentaire. En effet, le code récursif suivant, bien qu'il paraisse naturel, *est incorrect* :

```
    def taille(self) :
        if self == None or self_est_vide():
            return 0
        tg = self.sa_gauche.taille()
        td = self.sa_droit.taille()
        return 1 + td +td
```

En effet, la variable `self` désigne nécessairement un objet de la classe `ArbreBinaire` et ne peut donc jamais être égale à `None`. Inversement, l'objet Python `None` n'est pas de la classe `ArbreBinaire` et on ne peut donc pas lui appliquer la méthode `taille()`.

Dans le cas où le sous-arbre gauche est égal à `None`, l'appel de méthode `self.sa_gauche.taille()` ne fonctionnera donc pas. Bien entendu, on peut faire la même remarque pour le cas où le sous-arbre droit est `None`.

• La solution, un peu plus lourde, pour la méthode `taille()` est la suivante :

```
    def taille(self) :
        if self.est_vide() :
            return 0
        if self.sa_gauche == None :
            tg = 0
        else :
            tg = self.sa_gauche.taille()
        if self.sa_droit == None :
            td = 0
        else :
            td = self.sa_droit.taille()
        return 1 + tg + td
```

• Voici une solution analogue pour la méthode `hauteur()` :

```
    def hauteur(self):
        if self.est_vide():
            return 0
        if self.sa_gauche == None :
            hg = 0
        else :
            hg = self.sa_gauche.hauteur()
        if self.sa_droit == None :
            hd = 0
        else :
            hd = self.sa_droit.hauteur()
        return 1 + max(hg, hd)
```

• Le problème est donc résolu en traitant dans des structures conditionnelles `if ... else ...` *distinctes* les deux situations qui posent une difficulté : celles où le sous-arbre droit ou le sous-arbre gauche est égal à `None`. Bien entendu, cette structure permet aussi de traiter convenablement le cas où *les deux* sous-arbres sont égaux à `None` ainsi que celui où *aucun des deux* n'est est égal à `None`.

c) La complexité de ces algorithmes

Intéressons-nous à la complexité de ces deux algorithmes, c'est-à-dire à la relation entre le nombre d'instructions exécutées par l'algorithme et la taille des données en entrée, ici la taille de l'arbre binaire sur lequel on effectue les calculs.

• Il est clair que ces deux algorithmes conduisent à exécuter un nombre *fixe* d'instructions *pour chaque nœud de l'arbre* et que *l'on traitera chaque nœud une seule fois*. Nous pouvons en déduire que, dans les deux cas, *le nombre d'instructions exécutée est proportionnel à la taille de l'arbre* : la complexité de l'algorithme est **linéaire** et on utilise la notation $\mathcal{O}(n)$**.**

III Parcourir un arbre binaire

III.1 Les différents parcours possibles

• Qu'est-ce que « parcourir un arbre » ? On souhaite consulter ou afficher les clés associées à chacun des nœuds de notre arbre binaire ou bien encore numéroter chacun des nœuds (en commençant par le nombre 1) en n'oubliant aucun de ces nœuds. Pour cela il nous faut « nous déplacer » dans l'arbre de nœud en nœud *en suivant un ordre précis* qui nous assure de *rencontrer chaque nœud une et une seule fois*. Contrairement à une structure linéaire comme la liste chaînée, l'ordre du parcours d'un arbre, et notamment d'un arbre binaire, ne va pas de soi : il y a de multiples façons possibles de parcourir l'arbre et aucune n'est clairement plus naturelle que les autres.

• Dans quel but peut-on souhaiter parcourir un arbre binaire ? Pour rechercher une valeur dans cet arbre, ou bien compter le nombre de ses occurrences, ou bien pour afficher à l'écran les valeurs de tous les nœuds, ou pour rechercher la valeur maximum ou la valeur minimum, etc. Il existe donc de nombreuses questions sur l'information contenue dans l'arbre qui nécessitera de consulter tous ses nœuds un par un.

• Nous allons distinguer deux grandes façons classiques de parcours d'un arbre :

* le **parcours en profondeur d'abord** (en anglais **DFS** pour *Depth-First Search*) ;
* le **parcours en largeur d'abord** (en anglais **BFS** pour *Breadth-First Search*).

a) Le parcours en profondeur d'abord

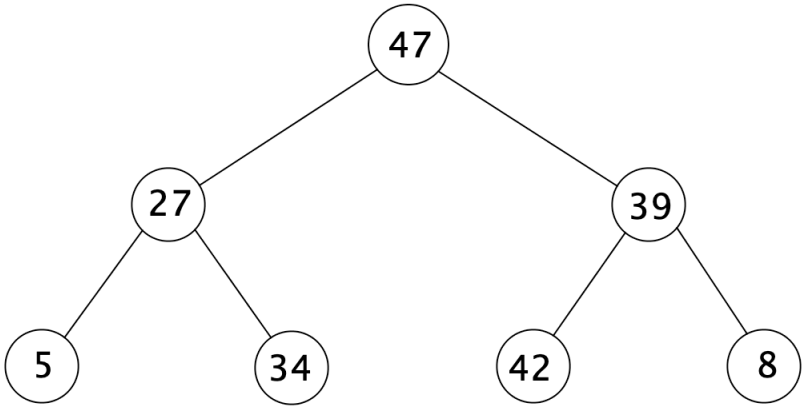
• Le principe du parcours en profondeur consiste à parcourir entièrement chaque branche de l'arbre *avant* d'explorer la branche voisine. Il reste toutefois à décider dans quel ordre on traite la racine, le sous-arbre gauche et le sous-arbre droit.

• On distingue alors les trois choix classiques suivants :

* le parcours en **profondeur préfixe** : on consulte *la racine d'abord*, puis on parcourt le sous-arbre gauche, puis le sous-arbre droit ;
* le parcours en **profondeur postfixe** (ou **suffixe**) : on parcourt le sous-arbre gauche, puis le sous-arbre droit et *finalement on consulte la racine* ;
* le parcours en **profondeur infixe** : on parcourt le sous-arbre gauche, puis on consulte la racine, et enfin on parcourt le sous-arbre droit.

Il en existe d'autres parcours moins courants et pour lesquels on ne définira pas de dénomination particulière.

• Considérons par exemple l'arbre binaire suivant :



• On aura les parcours suivants dans les trois cas :

* en profondeur préfixe : 47 – 27 – 5 – 34 – 39 – 42 – 8 ;

* en profondeur postfixe (ou suffixe) : 5 – 34 – 27 – 42 – 8 – 39 – 47 ;

* en profondeur infixe : 5 – 27 – 34 - 47 – 42 – 39 – 8.

b) Le parcours en largeur d'abord

• Le parcours en largeur consiste à traiter la racine puis tous les fils de la racine (en commençant par le fils gauche), puis tous les fils des fils, etc. D'une façon générale, si `h` est la hauteur de l'arbre, on traite le nœud de profondeur 1, puis tous les nœuds de profondeur 2, puis les nœuds de profondeur 3, etc. On finit donc en traitant les feuilles « les plus éloignées » de la racine.

• Si on reprend l'exemple précédent, le parcours en largeur donnera donc le parcours suivant :

47 – 27 – 39 – 5 – 34 – 42 – 8.

III.2 Implémentation des parcours en profondeur

• Dans cette partie, nous allons définir dans notre classe `ArbreBinaire` les méthodes permettant d'afficher à l'écran les valeurs de tous les nœuds d'un arbre *en suivant à chaque fois un parcours particulier*. Nous supposerons que l'arbre binaire à traiter n'est pas vide.

• Nous avons besoin d'une méthode pour afficher la clé associée à la racine d'un arbre :

```
    def affiche_cle(self) :
       """l'arbre ne peut pas être vide"""
        print(self.cle, end=" - ")
```

On remarquera ici l'argument `end=" - "` dans l'appel de la fonction `print` qui permet de faire en sorte que l'affichage suivant soit effectué sur la même ligne que celui-ci tout en séparant les deux affichages par la chaîne de caractère `' - '`.

a) Les parcours en profondeur d'abord

• Pour effectuer un parcours en profondeur, le plus simple est d'utiliser une programmation récursive.

• Le **parcours préfixe** : on affiche la valeur de la racine avant de parcourir les deux sous-arbres selon la même méthode. On a donc le code suivant :

```
    def affiche_pre(self) :
        self.affiche_cle()
        if self.sa_gauche != None :
            self.sa_gauche.affiche_pre()
            # on parcourt le sous-arbre gauche
        # sinon on n'affiche rien
        if self.sa_droit != None :
            self.sa_droit.affiche_pre()
            # on parcourt le sous-arbre gauche
        # sinon on n'affiche rien
```

• Le **parcours postfixe** (ou **suffixe**) : on affiche la valeur d'une racine <u>après avoir parcouru les deux sous-arbres</u> selon la même méthode :

```
    def affiche_post(self) :
        if self.sa_gauche != None :
            self.sa_gauche.affiche_post()
        if self.sa_droit != None :
            self.sa_droit.affiche_post()
        self.affiche_cle()
```

• Le **parcours infixe** : on parcoure le sous-arbre gauche, puis on affiche la valeur de la racine, puis on parcoure le sous-arbre droit :

```
    def affiche_in(self) :
        if self.sa_gauche != None :
            self.sa_gauche.affiche_in()
        self.affiche_cle()
        if self.sa_droit != None :
            self.sa_droit.affiche_in()
```

• Une fois ces trois méthodes récursives écrites, il est souhaitable de les **encapsuler** dans des méthodes qui permettent <u>de finaliser l'affichage</u> :

```
    def affiche_prefixe(self) :
        print("-- Affichage en profondeur préfixe --")
        self.affiche_pre()
        print() # retour à la ligne
        print("-----------")

    def affiche_suffixe(self) :
        print("-- Affichage en profondeur postfixe --")
        self.affiche_post()
        print() # retour à la ligne
        print("-----------")

    def affiche_infixe(self) :
        print("-- Affichage en profondeur infixe --")
        self.affiche_in()
        print() # retour à la ligne
        print("-----------")
```

III.3 Implémentation du parcours en largeur

• Le parcours en largeur pose des difficultés particulières de programmation car il ne se prête pas à une programmation récursive. Pour l'implémenter, nous allons utiliser une programmation itérative et avoir recours à une file dans laquelle on placera l'arbre et ses sous-arbres de différents niveaux que nous traiterons successivement.

• SI on importe la classe `File` que nous avons écrite dans un chapitre précédent, nous pouvons définir la méthode d'affichage suivante :

```
    def affiche_largeur(self) :
        print("-- Affichage en largeur --")
        F = File()
        F.enfiler(self)
        while not F.est_vide() :
            a = F.defiler()
            a.affiche_cle()
            if a.sa_gauche != None :
                F.enfiler(a.sa_gauche)
            if a.sa_droit != None :
                F.enfiler(a.sa_droit)
        print()
        print("-----------")
```

III.4 La complexité de ces algorithmes

• Comme pour les algorithmes permettant de déterminer la taille et la hauteur d'un arbre, ces différents algorithmes exécutent *un nombre fixe d'instructions pour chaque nœud de l'arbre* et traitent chaque nœud une et une seule fois. Nous pouvons en déduire que, là encore, la complexité des algorithmes est **linéaire**, donc en $\mathcal{O}(n)$**.**