Classe de terminale

II Structures de données

B Les arbres

2. Les arbres binaires de recherche

I La structure abstraite « Arbre binaire de recherche »

I.1 Qu'est-ce qu'un arbre binaire de recherche ?

• Lorsque l'on utilise une structure de données qui enregistre un très grand nombre de valeurs, il est souhaitable de disposer d'une structure qui *permette une recherche très efficace parmi ces valeurs*, par exemple pour déterminer si une certaine valeur est présente ou non dans la structure de données.

• Nous avons vu par exemple l'année dernière qu'une recherche par la méthode par dichotomie dans un tableau *trié* est beaucoup plus efficace qu'une recherche séquentielle dans un tableau *non trié*. En effet, dans le premier cas le coût de l'algorithme est logarithmique alors qu'il est linéaire dans le second.

• Dans le même esprit, nous allons découvrir un type particulier d'arbres binaires qui permet une recherche très efficace parmi les valeurs qu'il contient : il s'agit des **arbres binaires de recherche** (**ABR** en abrégé).

• Un arbre binaire de recherche est donc un arbre binaire qui comporte trois contraintes :

* il est **homogène**, c'est-à-dire que les clés (ou valeurs) associées aux différents nœuds sont toutes d'un même type, et le type de ces valeurs est **doté d'une relation d'ordre total**. On peut toujours dire si une valeur est inférieure ou supérieure à une autre valeur. Par exemple, en Python, les types `int`, `float` ou `str` comportent un telle relation d'ordre puisqu'on peut leur appliquer les opérateurs $>$ et $<$. Si Python ne fournit pas de façon native une relation d'ordre entre les valeurs d'un certain type (par exemple des tuples ou des dictionnaires) il faudra alors en définir une.

* pour tout un arbre binaire de recherche, comme pour chacun de ses sous-arbres de niveau inférieur,

  - les clés associées aux nœuds du sous-arbre gauche sont toutes inférieures à la clé associée à la racine

  - et la clé associée à la racine est elle-même inférieure aux clés associées aux nœuds du sous-arbre droit.

• Par exemple, parmi les différents arbres binaires ci-dessous :

Le premier et le deuxième ne sont pas des ABR tandis que les deux suivants en sont.

• L'intérêt d'une telle organisation des données est de faciliter la recherche d'une valeur dans l'arbre binaire. Un ABR est donc à un arbre binaire quelconque ce qu'est un tableau trié à un tableau quelconque.

• On rencontrera deux types de situation :

* celles où toutes les clés d'un même arbre binaire de recherche doivent être toutes distinctes ;

* celles où on acceptera qu'une même clé soit associée à plusieurs nœuds d'un même arbre.

• Pour chacun de ces deux cas, on adoptera une convention différente :

* Dans le premier cas, il faudra que les valeurs du sous-arbre gauche seront *strictement* inférieures à celle du nœud correspondant et les valeurs du sous-arbre droit seront *strictement* supérieures à celle du nœud.

* Dans le deuxième cas, Il faudra prendre une convention pour décider où sont rangées des valeurs égales. On peut, par exemple, décider que le sous-arbre gauche contient des valeurs inférieures ou égales à celle de la racine alors que le sous-arbre droit contient les valeurs strictement supérieures à celle de la racine. On pourrait bien entendu aussi choisir la convention inverse selon laquelle une valeur égale peut se trouver dans le sous-arbre droit.

• Puisqu'un ABR est un arbre binaire, nous pouvons utiliser le vocabulaire introduit dans le chapitre précédent et notamment les notions de feuille, de taille N et de hauteur h. Nous pourrons aussi utiliser les résultats numériques déjà indiqués et notamment les deux encadrements :

$$h \leq N \leq 2^h - 1$$
$$\text{et } \log_2(N+1) \leq h \leq N$$

• Il faut enfin remarque que, d'après cette définition, le sous-arbre gauche et le sous-arbre droit d'un ABR *sont eux-mêmes des ABR*.

I.2 Interface d'une classe `ABR` en programmation objet

• Un ABR pourra comporter les mêmes attributs qu'un arbre binaire. On peut par exemple le définir de façon récursive comme un objet d'une classe `ABR` qui comporte trois attributs :

* `cle` qui est du type, muni d'une relation d'ordre total, des valeurs à enregistrer ;

* `sa_gauche` et `sa_droit` qui sont de type `ABR`.

• Ce sont les méthodes associées à cette nouvelle classe `ABR` qui vont *garantir que la contrainte concernant l'emplacement des valeurs est toujours bien respectée* et que nous n'avons pas un arbre binaire « ordinaire ».

• Ceci implique que *nous ne pourrons pas utiliser un constructeur similaire à celui que nous avons créé pour les arbres binaires*. En effet, il n'est pas possible de créer sans précaution un ABR à partir d'une nouvelle valeur pour sa racine et de deux ABR déjà créés pour former son sous-arbre gauche et son sous-arbre droit. En effet, il est tout à fait possible qu'un arbre binaire construit de cette façon ne soit pas un ABR.

• Nous allons donc lui donner les méthodes suivantes :

* un constructeur `ABR(val)` qui permet de **créer un ABR vide ou bien une feuille**, c'est-à-dire un ABR comportant une valeur à la racine et dont les deux sous-arbres sont vides ;

* on utilisera les méthodes `taille()` et `hauteur()` comme pour un arbre binaire ordinaire ;

* une méthode `afficher()` qui permet d'afficher à l'écran le contenu de notre ABR ;

* une méthode `inserer(val)` qui permet d'ajouter une valeur `val` dans notre ABR tout en veillant à respecter la contrainte des ABR ;

* une méthode `rechercher(val)` qui permet de dire si une valeur donnée `val` est présente ou non dans notre ABR.

• On pourrait y ajouter une méthode `supprimer` permettant de supprimer une valeur dans notre ABR.

• Pour la méthode `afficher`, nous allons privilégier **le parcours en profondeur infixe**. En effet, on admettra et on retiendra la propriété très importante suivante :

Propriété : L'application du parcours d'un ABR en suivant un ordre en profondeur infixe permet d'afficher les clés par ordre croissant.

II Une implémentation des ABR

II.1 Définition de la classe et constructeur

• Nous pouvons créer la classe et son constructeur de la façon suivante :

```
class ABR :
    def __init__(self, val=None):
        """On peut créer un arbre vide ou une feuille."""
        self.cle = val
        self.sa_gauche = None
        self.sa_droit = None
```

On remarque ici qu'au moment de la construction, on affectera `None` aux attributs `sa_gauche` et `sa_droit`. Pour éviter le problème évoqué plus haut, notre constructeur ne permet pas d'attribuer directement un objet de type `ABR` à l'attribut `sa_gauche` ou `sa_droit`. L'ajout d'une nouvelle valeur dans un objet de la classe ABR se fera donc systématiquement par l'intermédiaire de la méthode `inserer`.

• On peut y ajouter une méthode `est_vide()` identique à celle de la classe `ArbreBinaire` :

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

II.2 Les méthodes identiques à celles des arbres binaires ordinaires

a) Les méthodes `taille()` et `hauteur()`

• Rappelons brièvement les méthodes `taille` et `hauteur` déjà implémentées pour la classe `ArbreBinaire` :

```
    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

    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)
```

b) L'affichage d'un ARB

• Nous allons définir la méthode `afficher()` d'un arbre binaire en suivant le *parcours en profondeur infixe* :

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

    def afficher_in(self) :
        if self.sa_gauche != None :
            self.sa_gauche.afficher_in()
        self.afficher_cle()
        if self.sa_droit != None :
            self.sa_droit.afficher_in()

    def afficher(self) :
        print("-- Affichage en profondeur infixe --")
        self.affiche_in()
        print()
        print("-----------")
```

• Nous pouvons y ajouter une méthode d'impression `afficher_largeur()` par parcours en largeur :

```
    def afficher_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("-----------")
```

Bien entendu on suppose ici qu'au début de notre fichier nous avons importé la classe `File` déjà implémentée.

II.3 La méthode `inserer(val)` et la méthode `appartient(val)`

a) Algorithme et code des méthodes

• Nous en arrivons maintenant à la méthode `inserer(val)` qui est caractéristique des arbres binaires de recherche. Nous avons choisi d'écrire une méthode qui accepte de placer plusieurs clés identiques dans un même `ARB` et qui place les clés égales à celle d'un nœud dans son sous-arbre gauche. On remarquera que l'on a adopté une méthode de **programmation récursive** :

```
    def inserer(self, val):
        """en cas d'égalité, on place la valeur
           dans le sous-arbre gauche"""
        if self.est_vide() :
            self.cle = val
        elif val <= self.cle : # on accepte l'égalité
            if self.sa_gauche == None :
                self.sa_gauche = ABR(val)
            else :
                self.sa_gauche.inserer(val)
        else :
            if self.sa_droit == None :
                self.sa_droit = ABR(val)
            else :
                self.sa_droit.inserer(val)
```

• Sur ce modèle, nous pouvons aussi introduire la méthode `appartient(val)`. Il s'agit cependant ici d'un accesseur et non d'un mutateur (on interroge la structure de données sans la modifier). On n'oublie donc pas d'utiliser le mot-clé `return` :

```
    def appartient(self, val) :
        if self.est_vide() :
            return False
        elif val < self.cle :
            if self.sa_gauche == None :
                return False
            else :
                return self.sa_gauche.appartient(val)
        elif val > self.valeur :
            if self.sa_droit == None :
                return False
            else :
                return self.sa_droit.appartient(val)
        else :
            return True # on a trouvé !
```

• Nous pouvons remarquer que cet algorithme suit une logique très similaire à celle qui guide la recherche dichotomique dans un tableau trié.

b) Application de l'algorithme d'insertion dans un ABR sur un exemple

• Remarquons que la structure de l'ABR obtenu **dépend de l'ordre dans lequel les différentes clés sont insérées**. Vérifions-le sur un exemple.

• Reconstituons la structure de l'arbre binaire de recherche obtenue après exécution de la séquence d'instructions suivantes :

```
abr1 = ABR()
abr1.inserer(25)
abr1.inserer(23)
abr1.inserer(31)
abr1.inserer(20)
abr1.inserer(24)
abr1.inserer(12)
abr1.inserer(27)
```

• Voyons la structure obtenue pour un arbre binaire de recherche obtenu après insertion *des mêmes valeurs dans un ordre différent* :

```
abr1 = ABR()
abr1.inserer(20)
abr1.inserer(23)
abr1.inserer(27)
abr1.inserer(24)
abr1.inserer(12)
abr1.inserer(31)
abr1.inserer(25)
```

c) Complexité de l'algorithme de recherche

• Nous pouvons remarquer que, dans la méthode `appartient (self, val)`, le nombre des tests effectués pour déterminer si la valeur `val` se trouve dans l'arbre binaire de recherche est inférieur ou égal au nombre de nœuds dans la branche parcourue. Le nombre de tests effectués est donc inférieur à la hauteur h.

• La recherche d'une valeur sera **d'autant plus efficace que la hauteur de notre ABR sera faible**. Pour une recherche efficace, il faut donc s'efforcer de construire un ABR *le plus équilibré possible*, c'est-à-dire qui minimise la hauteur h pour une taille N donnée.

• Le pire des cas est celui d'un ABR *dégénéré* pour lequel la taille N est égale par la hauteur h de l'arbre. Même dans ce cas très défavorable, la recherche est au pire **de complexité linéaire** soit en $\boldsymbol{\mathcal{O}(n)}$. Le plus souvent, elle sera bien meilleure.

• Le meilleur des cas est celui d'un ABR *complet* pour lequel la hauteur h de l'arbre est égale à $\log_2(N+1)$ (où N est la taille). Dans ce cas particulièrement favorable d'un ABR très bien équilibré, la recherche est alors **de complexité logarithmique** donc en $\boldsymbol{\mathcal{O}(\log(n))}$**.** On obtient alors la même complexité que pour une recherche dichotomique dans un tableau trié. Ceci est logique car, dans le cas d'une recherche dans un ABR bien équilibré, chaque test conduit à éliminer à peu près la moitié des possibilités restantes.