Spécialité NSI en terminale
Arbres binaires de recherche

1 Arbres binaires de recherche

Une arbre binaire de recherche est un arbre binaire qui vérifie la propriété : pour un sommet quelconque, les valeurs de son sous-arbre gauche sont inférieures à la valeur du sommet et les valeurs de son sous-arbre droit sont supérieures à la valeur du sommet. On suppose toutes les valeurs distinctes.

On reprend l'implémentation d'un arbre avec la classe `Arbre` présentée plus haut. Dans cette classe les deux méthodes `ajout_gauche` et `ajout_droit` permettent d'ajouter à un sommet un enfant au choix, soit à gauche, soit à droite, sans se soucier de la valeur. Nous supprimons ces deux méthodes et implémentons une méthode `ajoute` qui permet d'ajouter une valeur en respectant la propriété d'un arbre binaire de recherche. Cette insertion d'une valeur s'obtient après une recherche.

```
class ABR:
    def __init__(self, val):
        self.valeur = val
        self.gauche = None
        self.droit = None

    def ajoute(self, valeur):
        if valeur < self.valeur:
            if self.gauche is None:
                self.gauche = ABR(valeur)
            else:
                self.gauche.ajoute(valeur)
        elif valeur > self.valeur:
            if self.droit is None:
                self.droit = ABR(valeur)
            else:
                self.droit.ajoute(valeur)

arbre = ABR(17)
arbre.ajoute(10)
arbre.ajoute(23)
arbre.ajoute(19)
arbre.ajoute(25)
arbre.ajoute(20)
```

On parle de valeur ou de clé et donc d'insertion d'une valeur ou d'une clé.

Pour la recherche d'une clé, on écrit une méthode dans la classe `ABR` en suivant le même principe que pour l'insertion :

```
class ABR:
    ...

    def recherche(self, val):
```

```
        if val < self.valeur:
            return self.gauche.recherche(val) if self.gauche else False
        elif val > self.valeur:
            return self.droit.recherche(val) if self.droit else False
        return True
```

On peut tester la méthode `recherche` avec l'arbre défini précédemment.

```
>>> arbre.recherche(19)
True
>>> arbre.recherche(18)
False
```

Le principe utilisé est celui de la dichotomie. Suivant la valeur du nœud où l'on se trouve, on descend vers la gauche si cette valeur est plus grande que la valeur cherchée et vers la droite sinon. La recherche est terminée si la valeur est trouvée ou si on atteint une branche qui n'a pas de fils gauche ou de fils droit, suivant celui que l'on souhaiterait.

La même méthode peut s'écrire en version itérative.

```
class ABR:
    ...

    def recherche(self, val):
        while self.valeur is not None:
            if val < self.valeur:
                if not self.gauche: return False
                self = self.gauche
            elif val > self.valeur:
                if not self.droit: return False
                self = self.droit
            else: return True
```

Il apparaît clairement que si on insère les valeurs par exemple dans l'ordre 5, 8, 11, 13, 15, 18, on obtient une structure avec 6 niveaux semblable à une structure linéaire. Alors que si on insère les valeurs dans l'ordre 13, 8, 5, 11, 15, 18, on obtient seulement 3 niveaux.

Il existe des méthodes pour équilibrer l'arbre si c'est nécessaire, ce qui permet de diminuer significativement le coût d'une recherche.

Résultat : si l'arbre de recherche est équilibré et possède n nœuds, alors cette recherche a un coût de l'ordre du logarithme de n, c'est-à-dire du nombre de chiffres utilisés dans l'écriture de n.

D'autres fonctions sont utiles, par exemple pour la recherche d'un minimum ou d'un maximum, et peuvent être implémentées dans la classe.

```
class ABR:
    ...
```

```
    def min(self):
        s = self
        while s.gauche:
            s = s.gauche
        return s.valeur

    def max(self):
        s = self
        while s.droit:
            s = s.droit
        return s.valeur
```

Test des méthodes `min` et `max` avec l'arbre exemple :

```
>>> arbre.min()
10
>>> arbre.max()
25
```

Coût d'une recherche et d'une insertion

De manière générale, dans un arbre binaire de recherche, les fonctions de recherche d'une valeur, d'un maximum, d'un minimum, d'un successeur et d'un prédécesseur ont un coût de l'ordre de h où h est la hauteur de l'arbre. Il en est de même pour l'insertion d'une valeur.

Compléments

Pour terminer cette partie, quelques méthodes sont proposées simplement pour l'utilité qu'elles peuvent apporter dans certains cas.

Un parcours infixe permet d'afficher dans l'ordre croissant ou décroissant, suivant que l'on commence par les sous-arbres gauches ou les sous-arbres droits, les valeurs contenues dans l'arbre. Cet affichage a un coût de l'ordre de n si n est le nombre de nœuds.

On peut alors écrire une méthode pour afficher d'une manière plus visuelle un arbre, à l'aide d'un parcours en profondeur suivant l'ordre infixe. L'arbre est affiché couché, la racine à gauche.

```
class ABR:
    ...

    def affiche(self, niveau=0):
        if self.droit:
            self.droit.affiche(niveau + 1)
        print(niveau * "\t", self.valeur)
        if self.gauche:
            self.gauche.affiche(niveau + 1)
```

L'écriture pour la recherche d'une valeur est un peu lourde. On écrit `arbre.recherche(k)`. Il est possible de simplifier cette syntaxe et d'écrire par exemple `arbre[k]` qui a la valeur `True` si la valeur `k` est présente dans l'arbre et `False` sinon.

Pour obtenir ce résultat, on écrit une méthode `__getitem__` dans la classe `ABR`.

```
class ABR:
    ...

    def __getitem__(self, k):
        return self.recherche(k)
```

On définit un arbre pour tester cette nouvelle méthode.

```
>>> arbre = ABR(17)
>>> arbre.ajoute(10)
>>> arbre.ajoute(23)
>>> arbre.ajoute(19)
>>> arbre.ajoute(25)
>>> arbre.ajoute(20)
>>> arbre[19]
True
>>> arbre[21]
False
```

Afin de parcourir les valeurs des différents sommets d'un arbre, on peut souhaiter utiliser une boucle `for` dans un programme avec une syntaxe habituelle : `for k in arbre`.

Pour cela il suffit de rendre l'arbre « itérable » en définissant une méthode `__iter__` dans la classe `ABR`.

```
class ABR:
    ...

    def __iter__(self):
        if self.gauche:
            for n in self.gauche:
                yield n # yield est un mot clé du langage Python
        yield self.valeur
        if self.droit:
            for n in self.droit:
                yield n
```

On utilise le même arbre que précédemment pour tester la méthode.

```
>>> for k in arbre:
        print(k)

10
17
19
20
```

```
23
25
```

2 Application

Les applications sont nombreuses et diverses. Un exemple est dans le codage de Huffman.

Prenons un texte quelconque. Si nous utilisons le codage ASCII étendu, à 256 caractères pour avoir par exemple les lettres accentuées françaises, chaque caractère du texte peut être codé sur un octet. Donc un texte de 1000 caractères nécessite 1000 octets. Le principe du codage de Huffman est le suivant : plutôt que coder chaque caractère sur 8 bits, on utilise moins de bits pour les caractères les plus fréquents et plus de bits pour les caractères les moins fréquents.

Considérons un exemple simple. Un texte ne contient que les quatre caractères "A", "Z", "E" et "R". Le "E" est présent 500 fois, le "R" 300 fois, le "A" 150 fois et le "Z" 50 fois.

Nous commençons par le caractère le plus fréquent et codons le "E" par 0. Nous ne pouvons pas coder un autre caractère, par exemple le "R", avec le seul 1 ou avec 01. En effet une suite débutant par 01 est alors ambigüe : est-ce une chaîne qui commence par "ER" ou par "E" suivi d'un autre caractère dont le codage commence par 1 ? Nous codons donc le "R" par 10. Une condition nécessaire apparaît : le codage d'un caractère ne doit pas être le préfixe du codage d'un autre caractère. Donc après le codage du "E" et du "R", le codage des deux autres caractères doit commencer par 11. Nous choisissons 110 pour le "A" et 111 pour le "Z". Finalement le codage est le suivant :

"A"	"Z"	"E"	"R"
110	111	0	10

Considérons une suite quelconque, par exemple 100110111010 et vérifions que le système de codage ne génère aucune ambiguïté possible. La seule possibilité est la séparation : 10 0 110 111 0 10. Le texte est donc "REAZER".

Pour le texte de 1000 caractères, nous obtenons en nombre de bits : 500×1 bits pour les "E", 300×2 bits pour les "R", 150×3 bits pour les "A", et 50×3 bits pour les "Z". Le total est donc de 1700 bits au lieu des 8000 avec un codage sur 1 octet pour chaque caractère.

Le gain est très important sur cet exemple.

Cependant, avec seulement quatre caractères, on aurait pu envisager de les coder chacun sur 2 bits, soit 00, 01, 10 et 11. Dans ce cas le total serait de $1000 \times 2 = 2000$ bits pour le texte. Ce codage est donc moins performant que le précédent.

Remarque : le rapport 8000/1700 ou 2000/1700 s'appelle le taux de compression. Dans le premier cas il est d'environ 4,7 et dans le second cas d'environ 1,18.

Nous avons procédé sur cet exemple en choisissant un codage pour le caractère le plus fréquent, puis le caractère suivant dans l'ordre des fréquences décroissantes. En pratique, on construit un arbre en suivant le chemin inverse. Chaque caractère représente une feuille. On part des deux caractères les moins fréquents que l'on fusionne pour obtenir un nœud. À chaque étape, on procède à une fusion.