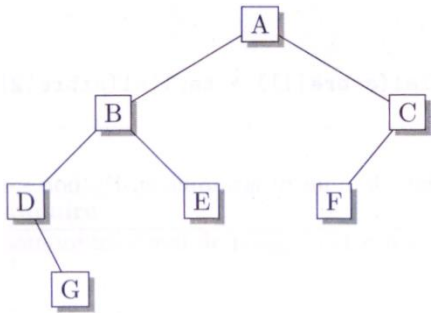
SDD_Arbres_Programmation

I/ Arbres binaires

On utilisera l'arbre à gauche comme exemple concret



Une possibilité est d'utiliser une classe *Tree*. Voici ici un exemple possédant également les méthodes *size(self)* et *height(self)* permettant de déterminer la **taille** de l'arbre (nombre de nœuds) et sa **hauteur** (nombre de sauts maximal entre un nœud et la racine de l'arbre.
On choisira ici par convention que la hauteur de la racine est de 1.

1/ Définition d'un arbre binaire

Voici un exemple de programme :

```
# Définition de la classe `Tree`
class Tree :
    # Constructeur de la classe `Tree`
    def __init__(self,val) :
        self.value = val
        self.left = None   # Fils à gauche par défaut
        self.right = None  # Fils à droite par défaut

    # On ajoute un sous-arbre à gauche
    def add_left(self,val) :
        self.left = Tree(val)

    # On ajoute un sous-arbre à gauche
    def add_right(self,val) :
        self.right = Tree(val)
```

Construction de l'arbre de l'exemple

```
# Jeu de tests
# Remplissage "en largeur"
my_tree = Tree("A")
my_tree.add_left("B")
my_tree.add_right("C")
my_tree.left.add_left("D")
my_tree.left.add_right("E")
my_tree.right.add_left("F")
my_tree.left.left.add_right("G")
```

2/ Taille d'un arbre binaire

La **récursivité** s'avère très utile pour parcourir l'ensemble des nœuds.
On compte la racine et le nombre total de nœuds du sous-arbre gauche **puis** celui du sous-arbre droite **(et non les deux en même temps)**.

Voici un exemple de programmation de la méthode *size(self)* :

```python
# Taille de l'arbre
def size(self) :
    # Taille des sous arbres gauche et droit
    sz_left, sz_right = 0, 0

    # Si un sous-arbre à gauche existe, on calcule sa taille
    # sinon elle est de zéro.
    if self.left is not None :
        sz_left = self.left.size()

    # Si un sous-arbre à droite existe, on calcule sa taille
    # sinon elle est de zéro.
    if self.right is not None :
        sz_right = self.right.size()

    # Renvoi de la taille totale de l'arbre (racine + sous-arbres)
    return 1 + sz_left + sz_right
```

```python
# Jeu de tests
# Remplissage "en largeur"
my_tree = Tree("A")
my_tree.add_left("B")
my_tree.add_right("C")
my_tree.left.add_left("D")
my_tree.left.add_right("E")
my_tree.right.add_left("F")
my_tree.left.left.add_right("G")

print(my_tree.size())   # Attendu : 7
```

3/ Hauteur d'un arbre binaire

Le principe est le même. On renvoie cette fois la **hauteur maximale** détectée entre les nœuds de l'arbre gauche et ceux de l'arbre droit augmentée d'un avec la racine (selon la convention adoptée).

Voici un exemple de programmation de la méthode *height(self)* :

```python
# Hauteur de l'arbre
def height(self) :
    # Hauteur des sous arbres gauche et droit
    hg_left, hg_right = 0, 0

    # Si un sous-arbre à gauche existe, on calcule sa hauteur
    # sinon elle est de zéro.
    if self.left is not None :
        hg_left = self.left.height()

    # Si un sous-arbre à droite existe, on calcule sa hauteur
    # sinon elle est de zéro.
    if self.right is not None :
        hg_right = self.right.height()

    # Renvoi de la hauteur totale de l'arbre
    # (racine + hauteur maximale des sous-arbres)
    return 1 + max(hg_left,hg_right)
```

```python
# Jeu de tests
# Remplissage "en largeur"
my_tree = Tree("A")
my_tree.add_left("B")
my_tree.add_right("C")
my_tree.left.add_left("D")
my_tree.left.add_right("E")
my_tree.right.add_left("F")
my_tree.left.left.add_right("G")

print(my_tree.size())   # Attendu : 7
print(my_tree.height()) # Attendu : 4
```

4/ Parcours d'un arbre binaire

Il y a deux types de parcours, le **parcours en profondeur** et le **parcours en largeur**.

Principe du parcours en profondeur :

- On explore de manière récursive son sous-arbre gauche puis son sous-arbre droit.
- Si un sous-arbre est vide, on passe au suivant s'il existe, sinon l'algorithme s'arrête.

Il y a trois types de parcours en profondeur :

- Si la **racine est traitée avant ses deux sous-arbres**, il s'agit d'un ordre préfixe.
- Si la **racine est traitée après son sous-arbre gauche mais avant son sous-arbre droit**, il s'agit d'un ordre infixe.
- Si la **racine est traitée après ses deux sous-arbres**, il s'agit d'un ordre postfixe (appelé suffixe également).

Voici un exemple de programmation des différents parcours en profondeur :

```python
# Parcours préfixe de l'arbre
def pref_print(tree) :
    if tree is None :
        return

    print(tree.value, end = " ")
    pref_print(tree.left)
    pref_print(tree.right)
```

```python
# Parcours infixe de l'arbre
def inf_print(tree) :
    if tree is None :
        return

    inf_print(tree.left)
    print(tree.value, end = " ")
    inf_print(tree.right)
```

```python
# Parcours postfixe de l'arbre
def postf_print(tree) :
    if tree is None :
        return

    postf_print(tree.left)
    postf_print(tree.right)
    print(tree.value, end = " ")
```

```python
# 3 différents parcours
pref_print(my_tree)    # Attendu : A B D G E C F
print("\n")
inf_print(my_tree)     # Attendu : D G B E A F C
print("\n")
postf_print(my_tree)   # Attendu : G D E B F C A
```

Résultats obtenus

```
A B D G E C F

D G B E A F C

G D E B F C A
```

On constate qu'il suffit de **permuter l'ordre des instructions** en fonction du **type de parcours en profondeur** choisi : la programmation par récursivité permet une meilleure compréhension de l'algorithme.

A savoir : ces parcours sont à connaître !

On peut intégrer ces parcours dans la classe *`Tree`* et en faire des *méthodes*. **On fera attention à vérifier si les sous-arbres gauche et droit ne sont pas vides**.
Voici un exemple :

```python
def pref_print(self) :
    if self is None :
        return

    print(self.value,end = " ")

    if self.left is not None :
        self.left.pref_print()

    if self.right is not None :
        self.right.pref_print()
```

```python
def inf_print(self) :
    if self is None :
        return

    if self.left is not None :
        self.left.inf_print()

    print(self.value, end = " ")

    if self.right is not None :
        self.right.inf_print()
```

```python
def postf_print(self) :
    if self is None :
        return

    if self.left is not None :
        self.left.postf_print()

    if self.right is not None :
        self.right.postf_print()

    print(self.value, end = " ")
```

On obtient bien sûr les mêmes résultats que précédemment :

```python
# 3 différents parcours (méthodes)
my_tree.pref_print()
print("\n")
my_tree.inf_print()
print("\n")
my_tree.postf_print()
```

Résultats obtenus

```
A B D G E C F

D G B E A F C

G D E B F C A
```

La **complexité** d'un parcours en profondeur est **linéaire**, chaque nœud n'étant visité qu'une seule fois.

Aller plus loin : le parcours en largeur

Le **parcours en largeur** est bien moins intuitif puisqu'il faut afficher les nœuds par niveau et de gauche à droite.

Méthode 1 : par récursivité

- **Déterminer** la **hauteur** de l'arbre et parcourir chaque niveau à l'aide d'une **boucle *for***.
- Par **récursivité**, parcourir alors les fils de chaque nœud (s'ils existent, cas d'arrêt) en affichant ceux du niveau correspondant (cas d'arrêt).

Voici un exemple de parcours en largeur par récursivité :

```
def width_print(tree):
    # Récupération de la hauteur de l'arbre
    h = tree.height()

    # Affichage des noeuds par niveau
    for i in range(1, h+1):
        level_print(tree, i)
```

```
def level_print(tree, level):
    # Cas d'arrêt
    if tree is None:
        return

    if level == 1:
        print(tree.value, end = " ")
        return # Inutile mais pour la compréhension

    # Cas général
    else :
        # Parcours des sous-arbres de gauche à droite
        # jusqu'au niveau souhaité
        level_print(tree.left, level-1)
        level_print(tree.right, level-1)
```

Résultats obtenus

```
width_print(my_tree) # Attendu : A B C D E F G
```

```
A B C D E F G
```

Méthode 2 : à l'aide d'une ***file***.

- On place l'arbre dans la file.
- Tant que la file n'est pas vide, on défile un élément, qui est un arbre, on affiche la valeur de la racine et on place dans la file chacun de ses deux arbres s'ils ne sont pas vides.

```
from queue import *

def level_print_bis(tree) :
    # Crée une file de la taille de l'arbre
    my_file = Queue(tree.size())
    my_file.put(tree)

    # Tant que la file n'est pas entièrement dépilée
    while not my_file.empty() :
        # On retire le dernier élément de la file
        # et on l'affiche
        u_tree = my_file.get()
        print(u_tree.value, end=" ")

        # On place dans la file les valeurs des sous_arbres
        # s'ils existent
        if u_tree.left is not None :
            my_file.put(u_tree.left)
        if u_tree.right is not None :
            my_file.put(u_tree.right)
```

```
# Jeu de tests
# Remplissage "en largeur"
my_tree = Tree("A")
my_tree.add_left("B")
my_tree.add_right("C")
my_tree.left.add_left("D")
my_tree.left.add_right("E")
my_tree.right.add_left("F")
my_tree.left.left.add_right("G")

# Attendu : A B C D E F G
level_print_bis(my_tree)
```

Résultats obtenus

```
A B C D E F G
```

Déroulement du script	*Contenu de la file*
➢ On place 'A' dans la file.	'A'
➢ On retire et affiche 'A' de la file.	vide
➢ On place 'B' et 'C' dans la file.	'B' ; 'C'
➢ On retire et affiche 'B' de la file.	'C'
➢ On place 'D' et 'E' dans la file.	'C' ; 'D' ; 'E'
➢ On retire et affiche 'C' de la file.	'D' ; 'E'
➢ On place 'F' dans la file.	'D' ; 'E' ; 'F'
➢ On retire et affiche 'D' de la file.	'E' ; 'F'
➢ On place 'G' dans la file.	'E' ; 'F' ; 'G'
➢ On retire et affiche 'E' de la file.	'F' ; 'G'
➢ On retire et affiche 'F' de la file.	'G'
➢ On retire et affiche 'G' de la file.	vide

Fin de l'algorithme

La **complexité** du **parcours en largeur** est aussi **linéaire** avec cette méthode.

Aller plus loin : différents parcours d'arbres binaires avec les complexités et démonstrations.
http://math.univ-lyon1.fr/irem/IMG/pdf/parcours_arbre_avec_solutions-2.pdf

II/ Arbres binaires de recherche

Un arbre binaire de recherche (noté ABR) est un arbre vérifiant la propriété suivant : *« pour un nœud quelconque, les valeurs de son sous-arbre gauche lui sont inférieures et celles de son sous-arbre droit lui sont supérieures ».*
En règle générale, les valeurs sont différentes. Le cas échéant, on les placerait aléatoirement dans le sous-arbre gauche ou droite correspondant.

1/ Ajout d'une clé dans un ABR

L 'ajout d'une **valeur** (ou **clé**) se fait alors en **fonction de celle du nœud père** et ce par récursivité. La **racine** doit donc être **initialisée**.

Voici un exemple de programme :

```
# Définition de la classe de l'arbre binaire de recherche.
class ABR :
    # Attribut de la classe `ABR`
    def __init__(self, val) :
        self.value = val
        self.left = None    # Valeur par défaut
        self.right = None   # Valeur par défaut

    # Ajout d'une valeur dans l'ABR
    def add(self, val) :
        # Cas où la valeur se situera dans le sous-arbre gauche.
        if val < self.value :
            # Cas d'arrêt
            if self.left is None : # S'il s'agit d'une feuille.
                self.left = ABR(val)
            # Cas général
            else :
                self.left.add(val)

        # Cas où la valeur se situera dans le sous-arbre droit.
        else :
            # Cas d'arrêt
            if self.right is None : # S'il s'agit d'une feuille.
                self.right = ABR(val)
            # Cas général
            else :
                self.right.add(val)
```

```
    # Le parcours infixe permet de vérifier qu'il s'agit bien
    # d'un arbre binaire de recherche (ordre croissant).
    def inf_print(self) :
        if self is None :
            return

        if self.left is not None :
            self.left.inf_print()

        print(self.value, end = " ")

        if self.right is not None :
            self.right.inf_print()


# Jeu de tests
tree = ABR(15)
tree.add(10)
tree.add(23)
tree.add(19)
tree.add(25)
tree.add(20)

tree.inf_print() # Attendu : tri dans l'ordre croissant
```

Résultats obtenus

```
10 15 19 20 23 25
```

On parcourt l'arbre selon ses sous-arbres gauche et droite en comparant systématiquement la clé à ajouter avec les celles des nœuds visités.

Complexité : pour chaque clé à insérer, on parcourt l'arbre selon sa **hauteur** notée *h*. Il y un nombre ***n*** de clés à ajouter soit une **complexité égale à n x h**.

Rappel :
- Si l'arbre est équilibré (la hauteur de chaque sous-arbres diffère au plus d'un), **h** est de l'ordre de $\mathbf{\log_2(n)}$.
- Si l'arbre est dégénéré (sous forme d'une liste), **h** est de l'ordre de **n**.

La complexité de l'ajout de tous les éléments varie entre $\mathbf{n \times \log_2(n)}$ et $\mathbf{n^2}$.

Aller plus loin : **tri d'une liste**.

On remarque que le parcours infixe d'un ABR (de complexité **n**) permet de trier ses clés dans l'ordre croissant avec une complexité comprise entre **$n \times \log_2(n)$** et **n^2**.

Si l'ABR est équilibré, alors on obtient un **tri d'une grande efficacité**, il est appelé « tri rapide » (*quick sort* en anglais). La grande difficulté est donc d'obtenir un tel arbre !

L'équilibrage des ABR est un sujet difficile et passionnant. De nombreuses techniques ont été proposées (arbres rouge-noir, arbres AVL, arbres déployés, . . .). Parmi celles-ci, une technique est simple à mettre en œuvre : si on insère des éléments dans un ordre aléatoire dans un ABR initialement vide, on obtient avec grande probabilité un ABR équilibré.

A noter : La technique des arbres AVL sera vue en pratique.

2/ Recherche d'une clé dans un ABR

Pour la recherche d'une clé dans un ABR, on applique le même principe que pour l'insertion d'une clé.

Voici un exemple de programme :

```python
def search(self, val) :
    # Parcours du sous-arbre gauche si la valeur est
    # inférieure à celle du noeud courant
    if val < self.value :
        # S'il reste un sous-arbre gauche à parcourir
        if self.left is not None :
            return self.left.search(val)
        # Sinon il s'agit d'une feuille : il n'y aura
        # de valeur plus petite, on renvoie `False`
        else :
            return False

    # Parcours du sous-arbre droit si la valeur est
    # supérieure à celle du noeud courant
    elif val > self.value :
        # S'il reste un sous-arbre droit à parcourir
        if self.right is not None :
            return self.right.search(val)
        # Sinon il s'agit d'une feuille : il n'y aura
        # de valeur plus grande, on renvoie `False`
        else :
            return False

    # On a val = self.value : on renvoie `True`
    else :
        return True
```

```python
# Jeu de tests
tree = ABR(15)
tree.add(10)
tree.add(23)
tree.add(19)
tree.add(25)
tree.add(20)

print(tree.search(23))  # Attendu : True
print(tree.search(160))  # Attendu : False
```

Résultats obtenus

```
True
False
```

La **complexité** est également proportionnelle à la **hauteur** de l'arbre. En effet, à chaque appel récursif, on cherche au niveau supérieur de l'ABR.
Cette recherche est donc très efficace (du niveau de celle d'une liste triée) du moins ***si l'ABR est équilibré***.