Spécialité NSI en terminale
Structures non linéaires : les arbres

Les listes, les piles et les files sont des structures de données linéaires. Des structures non linéaires peuvent être intéressantes dans la représentation des données lorsque celles-ci sont hiérarchisées et particulièrement si leur nombre est important. Parmi celles-ci, on trouve les arbres et en particulier les arbres binaires. On parle d'arbre, de structure arborescente ou d'arborescence.

On trouve des arbres dans des domaines divers : arbre généalogique, arbre lexicographique (qui présente un ensemble de mots) où les préfixes communs à plusieurs mots n'apparaissent qu'une seule fois, organigramme d'une société avec directeur, sous-directeurs, secrétaires, manutentionnaires, agents comptables, etc.

Le contenu d'un livre peut être présenté sous la forme d'un arbre avec les dépendances entre les différents chapitres et les différentes sections ou sous-sections.

Sur un écran d'ordinateur, le système d'exploitation nous présente, à l'aide d'un explorateur, les données sous forme arborescente. Cette forme de présentation nous permet d'accéder rapidement à une donnée particulière.

Un exemple rencontré dans le programme de NSI en première : le DOM (document object model) en JavaScript dans une page HTML. Il s'agit d'un arbre qui permet de trouver un nœud, (un élément), par exemple à l'aide de querySelector.

1 Définitions

1.1 Arbre

Un *arbre* peut être défini par divers éléments : des nœuds, des racines, des feuilles, des branches. Une branche relie deux nœuds. Une racine et une feuille sont des nœuds particuliers. On peut aussi le définir sur le modèle des liens de parenté avec des enfants et des parents, des descendants, des ancêtres.

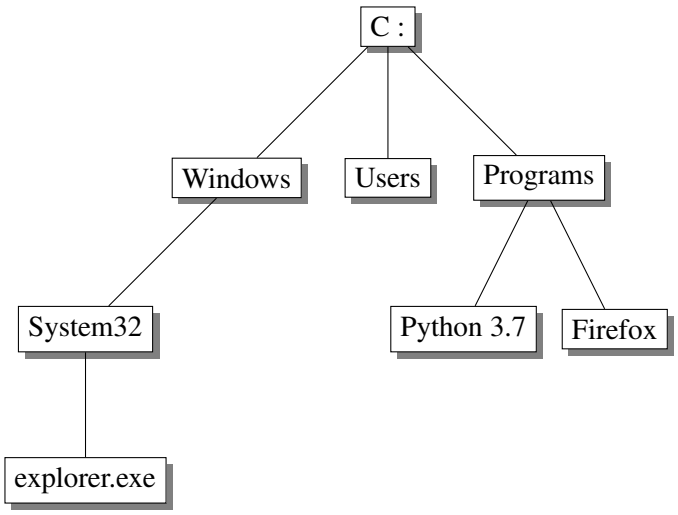
Les branches permettent d'aller de la racine jusqu'à chaque feuille.

Une manière simple de définir un arbre est récursive. Un arbre est un ensemble de un ou plusieurs nœuds tel que : un nœud particulier est appelé la racine et les autres nœuds partitionnés en sous-ensembles disjoints constituent des arbres qui sont appelés les sous-arbres de la racine.

Une autre façon de présenter un arbre est de dire qu'il est composé de nœuds et que chaque nœud excepté le nœud racine n'a qu'un seul parent. La racine n'a pas de parent. Un nœud qui a un parent p est un enfant de p. Une feuille est un nœud qui n'a pas d'enfant.

On peut également utiliser le vocabulaire des graphes présentés dans la section suivante et parler de sommets, d'arêtes et de chemins.

On peut représenter un arbre avec un schéma comme celui-ci :



Par exemple, l'explorateur de fichiers sous Windows propose une structure arborescente. Nous voyons la racine d'une partition qui est notée `C:`. C'est la racine de l'arbre. Ensuite des répertoires comme `Programmes` ou `Windows`. Ce sont des nœuds. Chaque répertoire contient éventuellement des sous-répertoires qui sont aussi des nœuds et qui eux-mêmes peuvent contenir des sous-répertoires, et ainsi de suite. Chaque répertoire ou sous-répertoire peut contenir des fichiers qui sont les feuilles.

Un nœud peut avoir un nombre arbitraire d'enfants.

1.2 Arbre binaire

Définition : un *arbre binaire* est un ensemble de nœuds, soit vide, soit constitué d'un nœud appelé racine et de nœuds constituant deux arbres binaires, l'un appelé sous-arbre gauche de la racine et l'autre sous-arbre droit de la racine.

On peut utiliser la structure de liste en Python pour implémenter une structure d'arbre.
Par exemple, dans le cas d'un arbre binaire :

- une liste vide pour un arbre vide ;
- une liste contenant trois éléments, la valeur ou clé de la racine, le sous-arbre gauche et le sous-arbre droit. Les deux sous-arbres sont représentés par des listes.

Il est nécessaire de définir des fonctions pour :

- créer un arbre vide ou un arbre dont la racine est donnée ;
- savoir si un arbre est vide ou pas ;
- obtenir le sous-arbre gauche ou le sous-arbre droit à partir d'une racine ;
- insérer un nœud.

Considérons l'implémentation d'un arbre binaire. Il y a plusieurs manières d'insérer un nœud dans cet arbre. L'une des plus simples à programmer, si on dispose d'une relation d'ordre sur les valeurs, satisfait la règle suivante qui est appliquée récursivement : si la valeur à insérer est inférieure à la valeur de la racine, on l'insère dans le sous-arbre gauche, sinon on l'insère dans le sous-arbre droit.

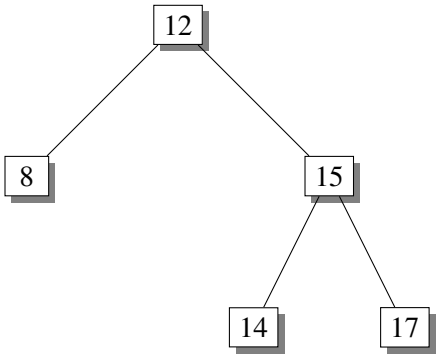
```
def creer_arbre(r=None):
    """renvoie un arbre vide ou un arbre de racine r"""
    if r is None:
        return []
    else:
        return [r, [], []]

def arbre_vide(a):
    return a == []

def ss_arbre_gauche(a):
    if not arbre_vide(a):
        return a[1]

def ss_arbre_droit(a):
    if not arbre_vide(a):
        return a[2]

def insere(a, val):
    if arbre_vide(a):
        a.append(val)
        a.append([])
        a.append([])
```

```
    elif val <= a[0]:
        insere(a[1], val)
    else:
        insere(a[2], val)
```

Test des fonctions :

```
>>> a = creer_arbre(12)
>>> arbre_vide(a)
False
>>> insere(a, 15)
>>> insere(a, 14)
>>> insere(a, 8)
>>> insere(a, 17)
>>> a
[12, [8, [], []], [15, [14, [], []], [17, [], []]]]
```

Le schéma de l'arbre obtenu est le suivant :



2 Notions de hauteur et de profondeur

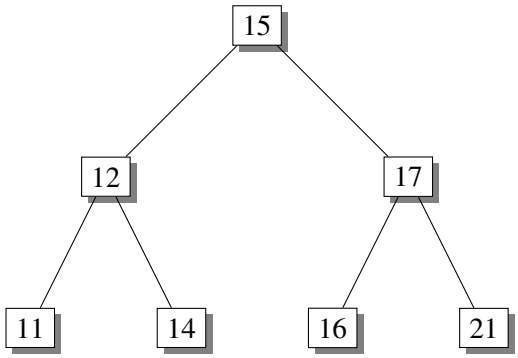
On définit un niveau pour chaque nœud. La racine a le niveau 0, ses enfants ont le niveau 1 et ainsi de suite :un nœud de niveau k est l'enfant d'un nœud de niveau $k-1$. La *profondeur* d'un nœud est le niveau du nœud. La *hauteur* ou la profondeur d'un arbre est égale au plus grand niveau.

Un arbre réduit à la racine a donc pour hauteur 0 et on décide qu'un arbre vide a pour hauteur -1.

Ce qui suit se généralise à un arbre quelconque mais est plus facilement abordable avec un arbre binaire. Supposons que chaque parent, hormis les feuilles, a exactement deux enfants et que les feuilles ont toutes le même niveau. On dit dans ce cas que l'arbre est *complet*. Une hauteur égale à 3 signifie que nous avons 4 niveaux, racine comprise. Alors au niveau 0, celui de la racine, nous avons un nœud, au niveau suivant nous avons deux nœuds, puis quatre nœuds et enfin huit nœuds qui sont des feuilles. Le total est donc de 8 feuilles (2^3) pour un total de 15 nœuds ($1+2^1+2^2+2^3=2^4-1$). De manière générale, si h est la hauteur, alors le nombre de feuilles est 2^h et le nombre de nœuds est $2^{h+1}-1$.

Remarque : on peut aussi définir la hauteur d'un arbre comme étant le nombre de niveaux. Dans ce cas, il y a un décalage de 1 par rapport à la précédente définition (un arbre réduit à la racine a pour hauteur 1 et un arbre vide a pour hauteur 0).

Ci-dessous, un arbre complet de hauteur $h=2$:



Remarque : on peut définir cet arbre binaire complet avec une simple liste.

```
>>> arbre = [15, 12, 17, 11, 14, 16, 21]
```

Dans ce cas, les deux enfants du nœud d'indice `i` se trouvent aux indices `2i+1` et `2i+2`.

3 Arbre binaire de recherche

Un *arbre binaire de recherche* est un arbre binaire : chaque nœud a au plus deux enfants, donc au plus un sous-arbre gauche et un sous-arbre droit. Les descendants à gauche d'un nœud ont des valeurs inférieures à celles du nœud et les descendants à droite d'un nœud ont des valeurs supérieures à celles du nœud. Il faut donc disposer d'une relation d'ordre sur l'ensemble des valeurs. On suppose que toutes les valeurs sont distinctes. L'arbre représenté avant le début de cette section est un arbre binaire de recherche.

Il y a plusieurs façons d'implémenter un arbre binaire de recherche. Nous pouvons utiliser l'implémentation d'un arbre binaire présentée dans la section précédente ou une structure de classe qui peut servir également à représenter un arbre binaire quelconque.

Un nœud a deux enfants et, exceptée la racine, a un parent. Chaque nœud a une valeur. On crée donc une classe `Nœud` avec quatre attributs : une valeur, un parent et deux enfants.

```
class Noeud:
    def __init__(self, valeur):
        self.valeur = valeur
        self.parent = None
        self.gauche = None
        self.droite = None

    def __str__(self):
        return str(self.valeur)
```

Les principales opérations à effectuer sur un tel arbre sont l'insertion et la suppression d'un nœud, ainsi que la recherche d'une valeur. On peut aussi rechercher un minimum, un maximum, un prédécesseur ou un successeur. Pour cela on est amené à parcourir l'arbre. Par exemple la valeur la plus petite se trouve à gauche. Donc pour trouver cette valeur, à partir de la racine on choisit toujours le sous-arbre gauche. Si un nœud n'a pas de sous-arbre ou seulement un sous-arbre droit, c'est terminé. Pour trouver la valeur la plus grande, on considère le sous-arbre droit à la place du gauche. Ces différentes opérations seront présentées plus tard dans la partie Algorithmique.

Voici par exemple l'implémentation de l'ajout d'un nœud. Pour cela, nous comparons sa valeur avec celle de la racine, puis avec celle de chaque nœud rencontré. Si cette valeur est inférieure, on continue en descendant à gauche, sinon en descendant à droite. Le dernier nœud rencontré est le parent du nouveau nœud.

```
class Noeud:
    ...

    def ajoute(self, valeur):
        if valeur < self.valeur:
            if self.gauche is None:
                self.gauche = Noeud(valeur)
                self.gauche.parent = self
            else:
                self.gauche.ajoute(valeur)
        elif valeur > self.valeur:
            if self.droite is None:
                self.droite = Noeud(valeur)
                self.droite.parent = self
            else:
                self.droite.ajoute(valeur)
```

Pour créer un arbre, on commence par créer une racine, par exemple :

```
arbre = Noeud(17)
```

Puis on ajoute des nœuds :

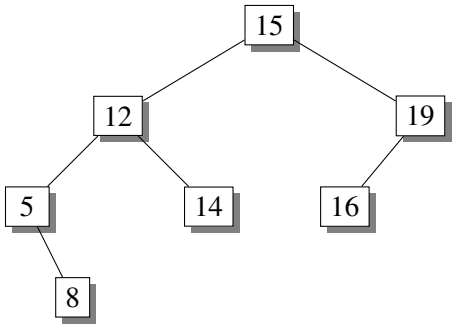
```
arbre.ajoute(10)
arbre.ajoute(23)
```

Dans l'implémentation précédente, les deux enfants d'un nœud sont des nœuds. Une autre manière d'implémenter un arbre binaire de recherche, de manière récursive, est de dire que les deux enfants d'un nœud sont deux arbres. Voici un exemple d'implémentation :

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

    def inserer(self, val):
        if val < self.valeur:
            if self.gauche is not None:
                self.gauche.inserer(val)
```

```
            else:
                self.gauche = ABR(val)
        else:
            if self.droit is not None:
                self.droit.inserer(val)
            else:
                self.droit = ABR(val)

# Test
arbre = ABR(15)
liste = [12, 5, 8, 19, 14, 16]
for n in liste:
    arbre.inserer(n)
```

On obtient l'arbre suivant :



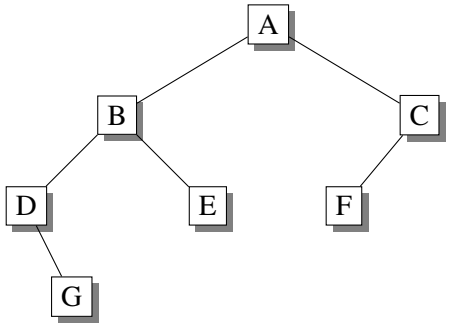
D'autres méthodes peuvent être ajoutées. Le code précédent permet seulement de créer un arbre. Les algorithmes que l'on utilise avec les arbres peuvent nécessiter les notions de hauteur et de taille, ainsi que différentes manières de parcourir un arbre, en largeur ou en profondeur, suivant un parcours infixe, préfixe, ou suffixe. Tout ceci sera traité plus tard.

4 Exercice

Dessiner un arbre binaire de recherche contenant les valeurs 11, 13, 14, 15, 17, 18, 19 satisfaisant la condition donnée dans chacun des deux cas.

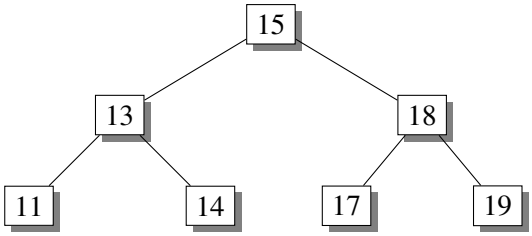
On appelle hauteur le nombre de niveaux.

1. Dans le premier cas, la hauteur de l'arbre est 3.
2. Dans le deuxième cas, l'arbre ressemble à l'arbre ci-dessous :



Solution

1. 1er cas :



2. 2e cas :