Chapitre 8 : Structures abstraites 2
Arbres binaires

Dans le premier chapitre sur les structures abstraites, nous avons discuté des listes chainées. Cette structure est idéale pour structurer des ensembles d'éléments destinés à être énuméré séquentiellement. Toutefois, l'accès à un élément quelconque dans cette liste chainée prend beaucoup de temps : pour améliorer les performances, on introduit une nouvelle structure abstraite qu'est l'arbre binaire.
Dans une seconde partie du chapitre, nous verrons que cette structure permet également de créer des relations d'ordre entre éléments de l'arbre, afin de simplifier la lecture et la hiérarchisation de données.

I. Introduction aux arbres binaires

1) Introduction

Exemple :
Sur votre ordinateur, dans votre dossier **Musique**, vous avez 3 sous-dossiers **Hard**, **Death** et un autre. Dans le sous-dossier **Hard**, vous avez les dossiers : ACDC, Disturbed et Black Sabbath.
Depuis le répertoire ACDC, quelqu'un a tapé la commande UNIX suivante : ls -1 ../../Glam/ , ce qui affiche :
The Darkness/
Beautiful Creatures/

Répondre aux questions suivantes à l'aide d'un graphique simplifiant la situation. Quel commande permet d'aller de The Darkness au répertoire Death ? Quel est le nombre total de sous-sous dossier de Musique ?

Rem : Chaque élément de l'arbre que vous avez dessiné s'appelle un noeud.

Grâce à cet exemple, on voit que les arbres ont une importance considérable lorsque l'on souhaite hiérarchiser des données (par exemple, suivant une notion temporelle pour les tournois de tennis ou une relation d'ordre pour les fichiers dans un système UNIX).
Le principe des structures arborescentes en Informatique est relativement simple : à partir d'un point de départ unique, notre structure chainée va se scinder à chaque étape en plusieurs branches.

2) Vocabulaire

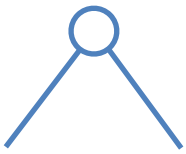
Définition : Un arbre binaire est un ensemble fini de noeuds correspondant à un des deux cas suivants :

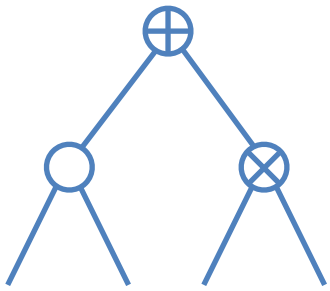
- soit l'arbre est vide ie. il ne contient **aucun** noeud.
- soit l'arbre n'est pas vide et ses noeuds sont structurés de la façon suivante :
  - un noeud est appelé la racine de l'arbre ;
  - les noeuds restants sont séparés en deux sous-ensembles, qui forment **récursivement deux sous-arbres** appelés respectivement sous-arbre gauche et sous-arbre droit ;
  - la racine est reliée à ses deux sous-arbres gauche et droit, et plus précisément à la racine de chacun de ses sous-arbres (lorsqu'ils ne sont pas vides).

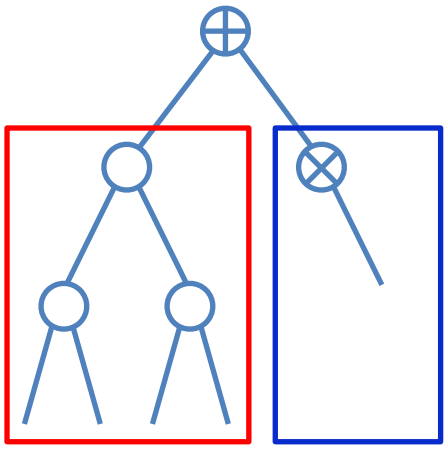
Rem : On peut établir un parallèle entre un noeud d'un arbre binaire et une cellule d'une liste chainée :

- l'arbre vide est alors comme la liste vide ;
- la racine d'un arbre non vide est l'équivalent de la tête d'une liste non vide ;
- les liens vers les sous-arbres gauche et droite correspondent à **deux** chainages "suivants".

Vocabulaire :







À gauche, un arbre à un noeud. Ce noeud est la racine de l'arbre. Généralement, on ne dessine pas les arêtes/branches terminales.

Au centre, un arbre **binaire** à trois noeuds. Le noeud marqué + est le père du noeud marqué x. Le noeud x est le fils du noeud +. Cet arbre est dit **parfait** : **tous les noeuds ont exactement deux fils.**

À droite, un arbre binaire à 5 noeuds. Les noeuds n'ayant aucun fils sont appelés des **feuilles**. En rouge, le sous-arbre gauche de la racine et en bleu, le sous-arbre droit de la racine.
La **hauteur** de cet arbre est 3 : c'est la distance maximale entre la racine et le noeud le plus profond.

Question : Cet arbre est-il parfait ?

Définition :

- La **hauteur** d'un arbre est définie comme le plus grand nombre de noeuds rencontrés en descendant de la racine jusqu'à une feuille. Tous les noeuds sont comptés, y compris la racine et la feuille.
- La **taille** d'un arbre correspond au nombre total de noeuds contenus dans l'arbre.

Question : Comment peut-on définir la hauteur d'un arbre de manière récursive ?

Algorithme :

La hauteur d'un arbre peut se définir récursivement :

- l'arbre vide a pour hauteur 0 ;
- un arbre non vide a pour hauteur le maximum des hauteurs des deux sous-arbres, auquel on ajoute 1.

Propriété :

Soit N la taille d'un arbre binaire et h sa hauteur.

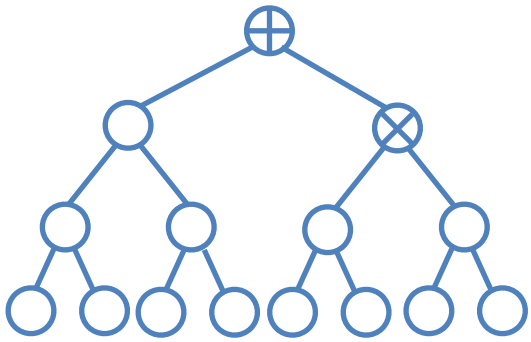
Alors, on a l'inégalité : $h \leq N \leq 2^h - 1$

Démonstration :

- Inégalité de gauche : dans le pire des cas, l'arbre ne comporte que des sous-arbres gauche (ou droit). Dans ce cas, l'égalité de gauche est vérifiée et $h = N$.

- Inégalité de droite : dans le meilleur des cas, l'arbre est complètement équilibré (il est dit **parfait**). Dans ce cas, chaque noeud a exactement deux fils. Chaque niveau contient 2^k noeuds avec k=0 pour la racine de l'arbre. Il s'agit de faire la somme de 2^k pour k allant de 0 à h.

  On peut démontrer que $\sum_{k=0}^{h} 2^k = 2^h - 1$.



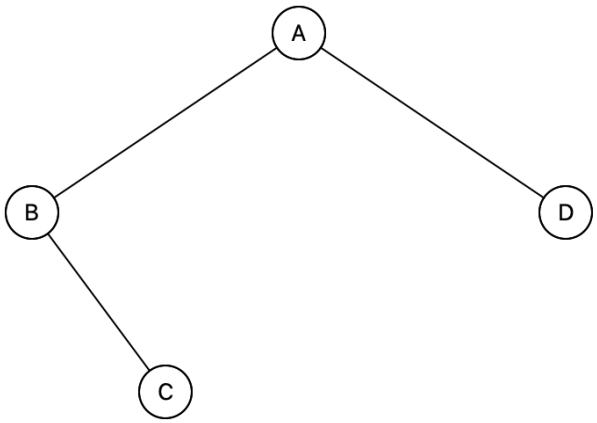
II. Algorithmes de parcours d'arbres

1) Implémentation possible des arbres en Python

Il existe un grand nombre d'implémentations possibles pour les arbres en Python. Traditionnellement, chaque noeud d'un arbre est représenté à l'aide d'une classe appelée **Noeud**. Cette classe contiendra trois attributs : **gauche, valeur, droit. gauche** représente le sous-arbre gauche, **valeur** l'information contenue dans l'arbre, **droit** le sous-arbre droit. L'arbre vide est représenté par la valeur None.

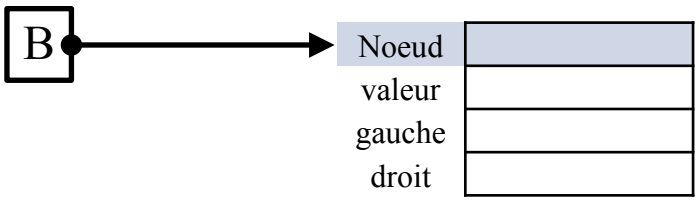
❐ — Exercice **1** —

- Créer la classe Noeud décrite ci-dessus. Cette classe ne comportera qu'un constructeur.
- Tester votre classe en créant un arbre **A** n'ayant qu'un seul noeud et pour valeur la lettre "A". Quelle est la valeur renvoyée par A ? par A.valeur, A.gauche et A.droit ?
- Créer un arbre **B** contenant quatre noeuds et dont la structure est décrite ci-contre :



- En étudiant la valeur des différents attributs de la classe **Noeud,** vérifiez que vous obtenez bien la structure voulue.

Comme pour les listes chainées, cet arbre peut également être interprété en terme d'emplacement mémoire. Complétez la représentation de cet arbre à quatre éléments :



Noeud	
valeur	
gauche	
droit	

Noeud	
valeur	
gauche	
droit	

Noeud	
valeur	
gauche	
droit	

Lorsque vous avez essayé d'accéder à la lettre C à partir de la racine, vous avez du appeler deux fois les attributs des Noeuds : **A.gauche.droit.valeur**. Cela souligne que les arbres binaires sont définies de manière **récursive**. Il est donc naturel d'écrire les opérations sur les arbres binaires en utilisant des fonctions récursives.

Le meilleur exemple consiste en l'algorithme de détermination de la taille d'un arbre **A**.

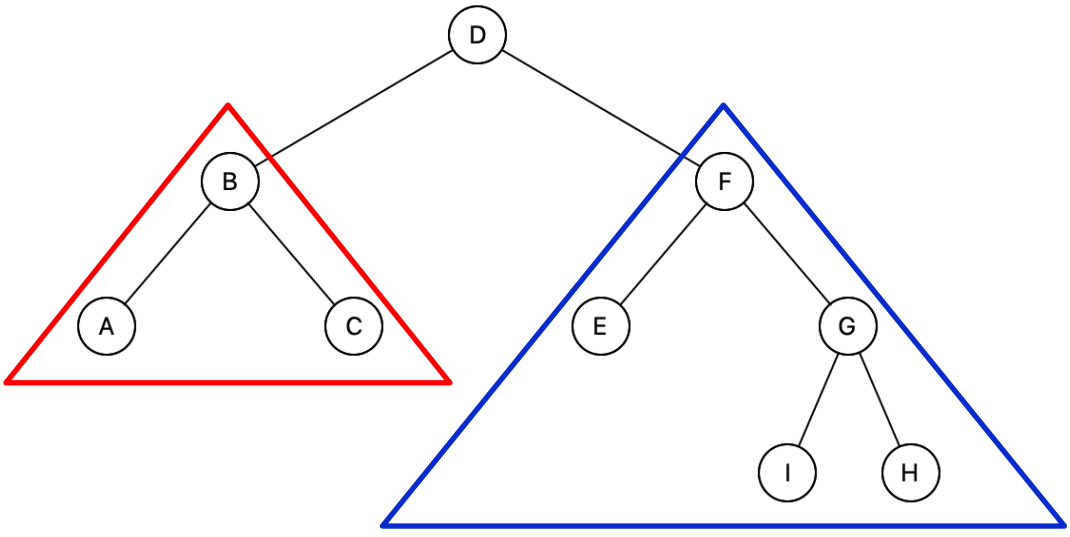
❐ — Exercice **2** —

Nous allons commencer par écrire une fonction récursive **hauteur(a)** prenant en argument un arbre constitué de Noeud et renvoyant la hauteur de l'arbre.
Pour cela, nous allons utiliser la définition et l'algorithme de la hauteur d'un arbre donné à la page 3. Lors de l'écriture de cette fonction, on réfléchira d'abord à la condition d'arrêt puis à l'appel récursif qu'il convient de faire.

❐ — Exercice **3** —

Nous allons écrire un algorithme récursif qui permet de calculer la taille (le nombre de noeuds présents) d'un arbre.

- ❖ <u>Étude d'un arbre :</u> on commence la lecture d'un arbre à la racine. L'arbre ci-dessous peut donc être décomposé en **1** noeud et **deux** sous-arbres non-vides.
  À partir de ce schéma, imaginez ce qu'il faudrait faire pour obtenir, de manière récursive, le nombre de noeuds total. Pour cela, demandez-vous si le procédé ci-dessus peut être répété.



- ❖ <u>Condition d'arrêt :</u> À quelle condition va-t-on s'arrêter de compter ? Pour cela, on pourra isoler les sous-arbres en les traçant avec des triangles.
- ❖ Écrire une fonction récursive **taille(a)** prenant en argument un arbre a constitué de Noeud et renvoyant la taille de l'arbre.

Complexité : Les fonctions taille et hauteur ont tout deux une complexité proportionnelle au nombre de noeuds dans l'arbre. En effet, ces fonctions font un nombre limité d'opérations sur chaque noeud et parcourent une fois et une seule chaque noeud.

2) Lecture d'un arbre : parcours infixe

Les deux fonctions précédentes parcourent tous les noeuds de l'arbre. L'ordre dans lequel ce parcours est effectué n'est pas important : nous pourrions commencer par calculer la taille du sous-arbre droit puis celle du sous-arbre gauche ou l'inverse.

Lorsque nous souhaitons afficher les différents noeuds d'un arbre, l'ordre prend une importance tout particulière. En effet, en l'absence de visualisation graphique, il est important de connaitre précisément notre stratégie d'affichage.

Une solution possible, appelée **parcours infixe**, est la suivante :
- ❖ on parcourt d'abord le sous-arbre gauche ;
- ❖ on affiche la valeur ;
- ❖ on parcourt ensuite le sous-arbre droit.

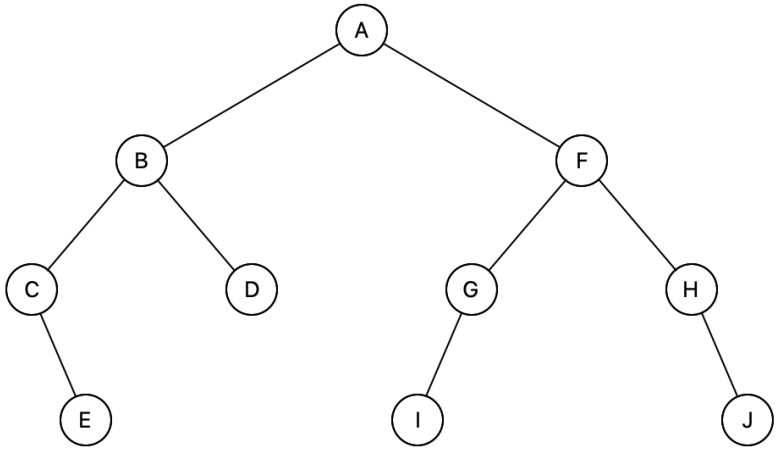
❐ — Exercice **4** —

Un algorithme de parcours infixe a été écrit en langage Python ci-dessous :

```
def parcoursInfixe(self, A):
    if A is None:
        return
    self.parcoursInfixe(A.droit)
    print(A.valeur)
    self.parcoursInfixe(A.gauche)
```

a) Préciser dans quel paradigme cette fonction a été créée. Expliquer quelle pourrait être la raison de ce choix.
b) Cette fonction correspond-elle à l'algorithme du parcours infixe tel que décrit précédemment ? Si non, corrigez la fonction.
c) Expliquer en quoi cette fonction est récursive et détailler le rôle des lignes 2 et 3.
d) On considère l'arbre page suivante.
   À la main (sans programmer le code), vérifiez que cet algorithme appliqué à l'arbre ci-dessous donne bien le parcours suivant : C, E, B, D, A … On complétera le reste du parcours.

Nous verrons dans la prochaine section que le parcours infixe permet en particulier d'afficher des arbres binaires de recherche dans l'ordre croissant pour la relation d'ordre utilisée.



Rem : D'autres algorithmes de parcours existent. Les deux autres algorithmes au programme sont les parcours préfixe et le parcours suffixe que l'on se propose d'étudier en exercice.

3) Lecture d'un arbre : parcours en largeur

Un autre algorithme de parcours d'arbre se révèle particulièrement intéressant en terme algorithmique. En effet, cet algorithme appelé **parcours en largeur** permet d'afficher les noeuds par profondeur croissante dans l'arbre sans utiliser le principe récursif.

❐ — Exercice **5** —

L'algorithme de parcours en largeur peut s'écrire comme suit :

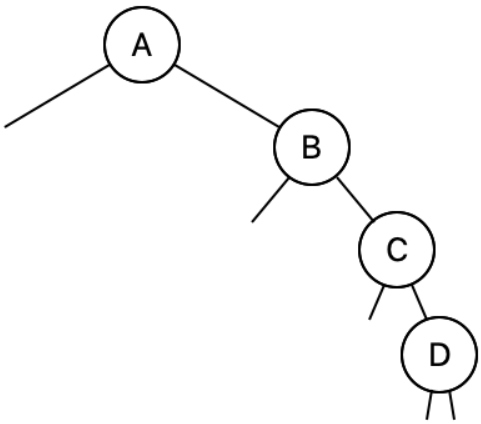
```
VARIABLE
T, Tg, Td : arbre, arbre, arbre
x : noeud
f : structure de données initialement vide

DEBUT
PARCOURS-LARGEUR(T) :
  enfiler(T.racine, f)
  tant que f non vide :
    x ← defiler(f)
    affiche x.clé
    si x.gauche ≠ None :
      Tg ← x.gauche
      enfiler(Tg.racine, f)
    fin si
    si x.droit ≠ None :
      Td ← x.droite
      enfiler(Td.racine, f)
    fin si
  fin tant que
FIN
```

a) Expliquer quelle est la structure de données utilisée pour gérer les racines des différents arbres.
b) À la main (sans programmer le code), vérifiez que cet algorithme appliqué à l'arbre page 7 donne bien le début du parcours suivant : A, B, F, C, D, G, …. . Complétez le parcours.
c) Déduire de la question précédente pourquoi cet algorithme permet **d'afficher les noeuds par profondeur croissante** dans l'arbre.
d) Écrire cet algorithme en Python. La fonction s'appellera **parcoursLargeur**, prendra comme argument un arbre A et affichera les noeuds de l'arbre par un parcours en largeur.
e) En faisant quelques tests et en utilisant au besoin le module Application fournie par votre professeur, expliquez le fonctionnement de cet algorithme avec votre propos mots.

Ce type d'algorithme est couramment utilisé dans les algorithmes d'exploration de possibilités : découverte des cases vides adjacentes au Démineur, intelligence artificielle pour calculer les coups suivants aux échecs…

Exercice : considérons l'arbre ci-contre. Que pensez-vous de l'utilité d'un tel arbre ? Pourrait-on la remplacer par une autre structure de données plus appropriée ? Justifier votre réponse.



III. Arbres binaires de recherche

1) Introduction

Imaginons un énorme dictionnaire qui contiennent tous les mots de la langue française. Ces mots sont répartis sur 17576 pages. Au sommet de chacune de ces pages sont écrits les trois premiers caractères du premier mot de la page.
On souhaite maintenant **rechercher** un mot précis (**KOALA**) dans ce dictionnaire. De manière astucieuse, on va utiliser les trois premiers caractères en sommet de page.

- Par exemple, on ouvre le dictionnaire au hasard : on trouve les lettres MOB. Toutes les pages avant celle-ci (*à gauche*) contiennent des caractères précédant MOB et toutes les pages après celle-ci (*à droite*) contiennent des caractères suivant MOB.
- On ouvre une page à gauche et on trouve les lettres FRI. On sait que toutes les pages à gauche ne nous permettront pas d'avoir KOALA, car à gauche de FRI, il n'y aura que des lettres "inférieures" à FRI.
- On ouvre donc une page à droite et on trouve les lettres JUN. De même, on va continuer à ouvrir à droite car KOA est "supérieur" à JUN.