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

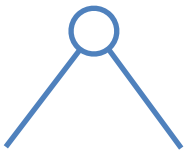
Définition : Un arbre binaire est un ensemble fini de noeuds correspondant à un des deux cas suivants :

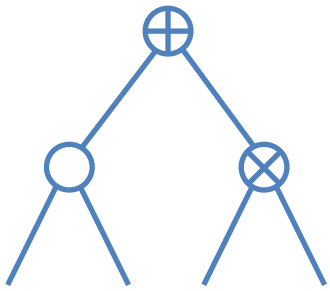
- soit l'arbre est vide ie. il ne contient **aucun** noeud.
- soit l'arbre n'est pas vide et ses noeuds sont structurés de la façon suivante :
  - un noeud est appelé la racine de l'arbre ;
  - les noeuds restants sont séparés en deux sous-ensembles, qui forment **récursivement deux sous-arbres** appelés 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).

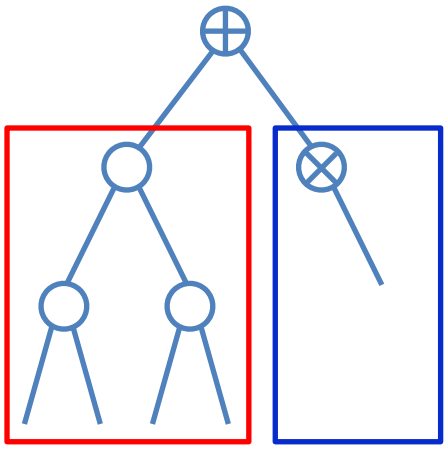
Rem : On peut établir un parallèle entre un noeud d'un arbre binaire et une cellule d'une liste chainée :

- 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 segments terminaux.

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.

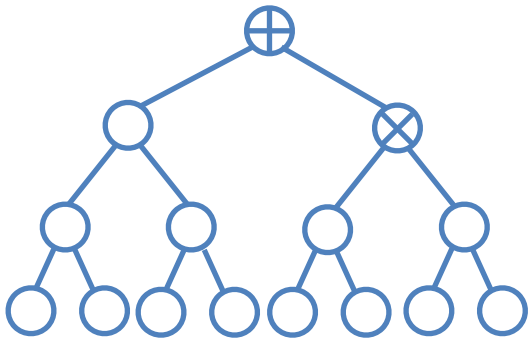
Alors, on a l'inégalité : $h \leq N \leq 2^h - 1$

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-1.

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



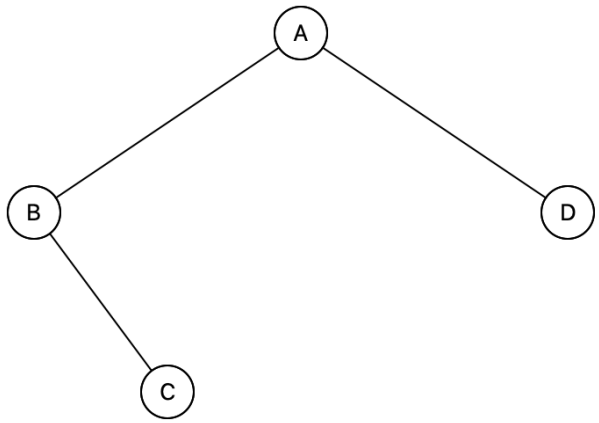
II. Algorithmes de parcours d'arbres

1) Implémentation possible des arbres en Python

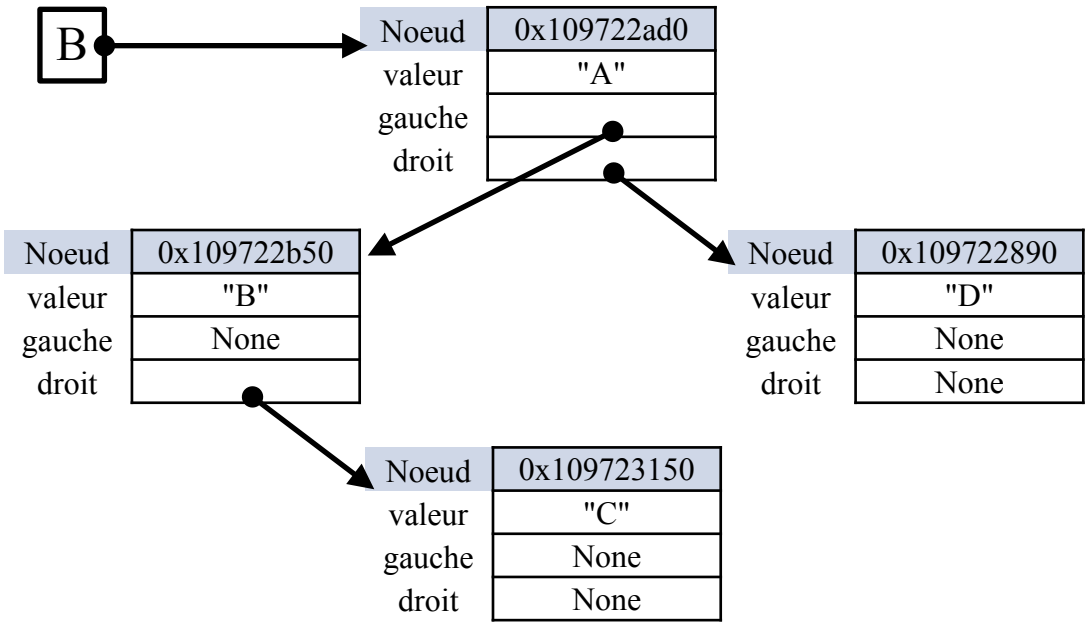
Il existe un grand nombre d'implémentations possibles pour les arbres en Python. Traditionnellement, chaque noeud d'un arbre est représenté à l'aide d'une classe appelée **Noeud**. Cette classe contiendra trois attributs : **gauche, valeur, droit. gauche** représente le sous-arbre gauche, **valeur** 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 :



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**.

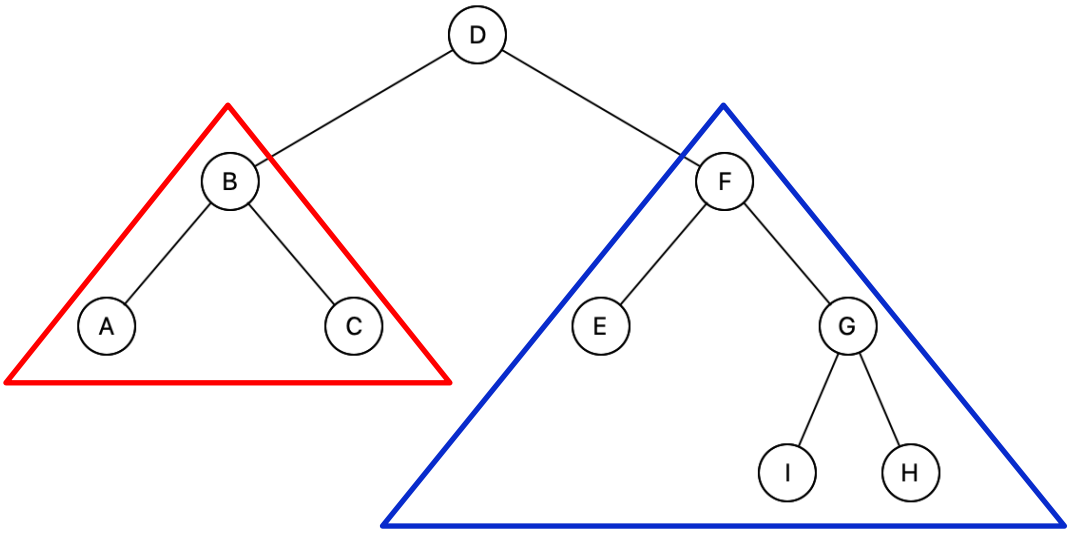
❐ — Exercice **2** —

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.

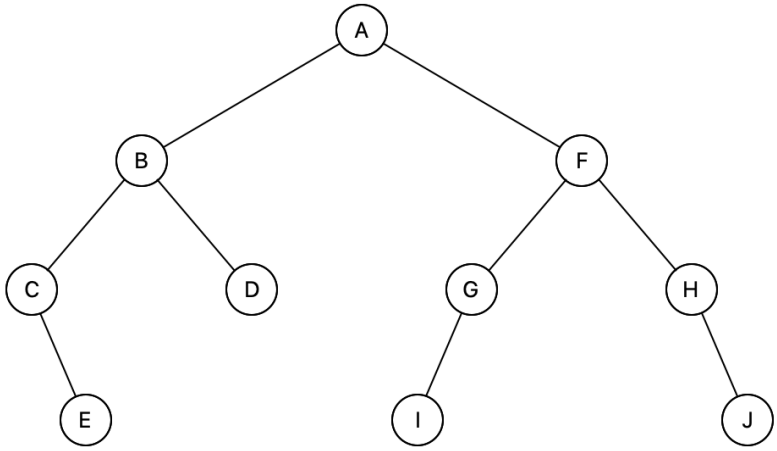
❐ — Exercice **4** —

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 :

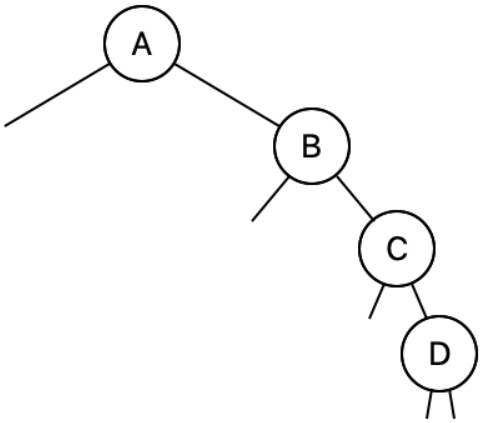
```
VARIABLE
T, Tg, Td : arbre, arbre, arbre
x : noeud
f : structure de données initialement vide

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.droit
      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 à l'extérieur de la classe. 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.

Exercice :

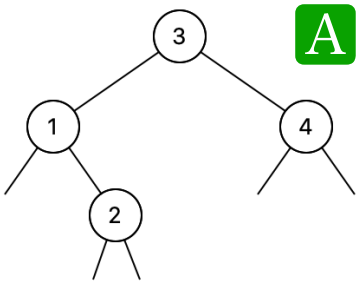
a) Créez un arbre de recherche correspondant à la situation ci-dessus.
b) Dans votre arbre, ajoutez la séquence OLI. Où pouvez-vous mettre le nouveau noeud afin de respecter le principe de l'arbre binaire de recherche ?
c) Dans l'arbre, est-ce que KOALA peut se trouver à gauche de OLI ?

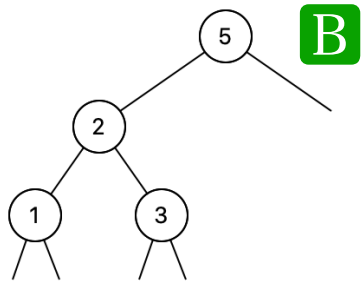
Conclusion : quand on se dirige du côté gauche, on trouve des mots plus petits (inférieurs) dans l'ordre alphabétique et quand on se dirige du côté droit, on trouve des mots plus grands (supérieurs) dans l'ordre alphabétique. Cette propriété, valable pour tous les noeuds de l'arbre, nous permettre de définir un **arbre binaire de recherche**.

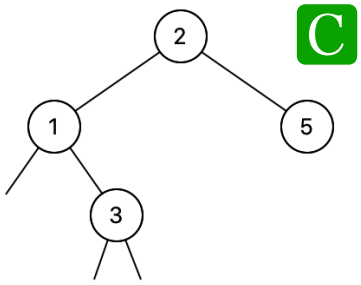
Définition : un arbre binaire de recherche (nommé ABR par la suite) est un arbre binaire dont les noeuds contiennent des valeurs pouvant être comparées entre elles et tel que, **pour tout noeud de l'arbre**, toutes les valeurs du sous-arbre gauche (respectivement droit) sont plus petites (resp. grandes) que la valeur du noeud.

Remarque : certains ABR peuvent contenir deux fois la même valeur. En cas d'égalité, la relation d'ordre utilisée n'est pas stricte et le doublon peut être trouvé à gauche ou à droite. Ces arbres représentent des objets mathématiques appelés **multi-ensembles**. Si la relation d'ordre interdit les doublons, les objets mathématiques sont des **ensembles**.

Exemple :
Les arbres A et B sont des arbres binaires de recherche : expliquez pourquoi.
L'arbre C n'est pas un arbre binaire de recherche : expliquez pourquoi.







Exercice :

a) Dessinez trois ABR différents contenant 6 noeuds. Les valeurs seront : 13, 3, 7, 2, 5, 11
b) On va créer un ABR dont les valeurs seront classées en fonction de la longueur des chaines de caractères.
Dessinez trois ABR différents contenant 5 noeuds : python, quotient, function, set, boolean

2) Algorithmes de recherche et d'ajout

Les algorithmes permettant de déterminer la taille et la hauteur d'un arbre ne changent pas pour les ABR. De même, les algorithmes de parcours infixe, suffixe, préfixe ou en largeur ne sont pas modifiés pour les ABR. Nous réutiliserons donc toutes ces fonctions dans la suite du cours.

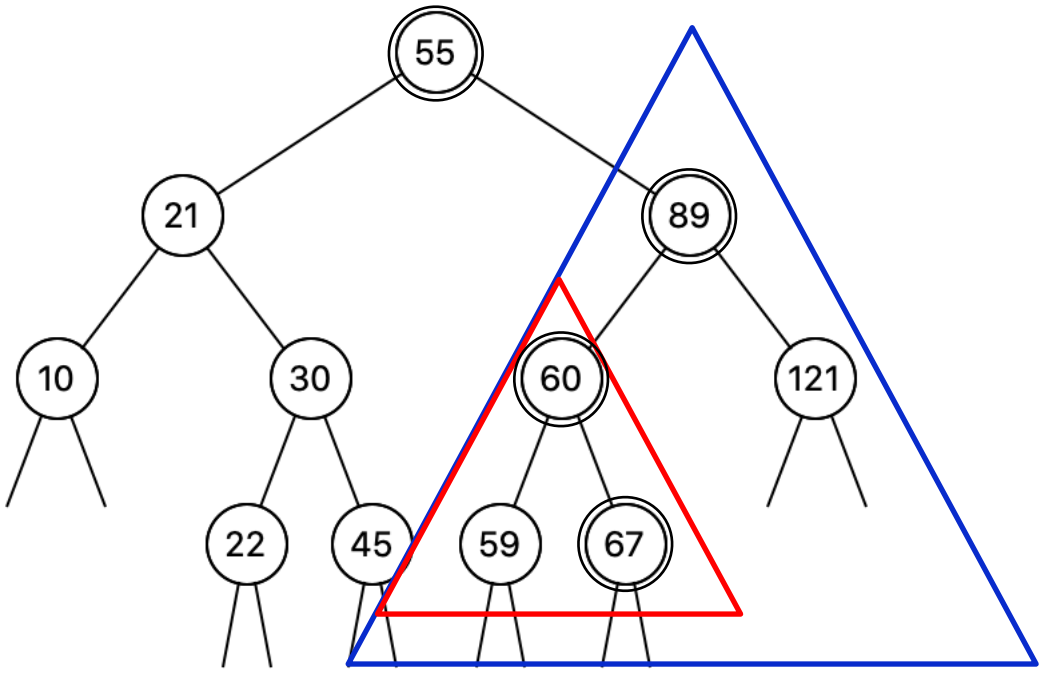
L'introduction a souligné l'intérêt premier des ABR : ceux-ci sont parfaitement adaptés à la recherche d'une valeur parmi un ensemble d'autres valeurs. Nous allons donc commencer par créer une fonction **appartient** nous permettant de tester si une valeur est présente dans un ensemble de valeurs.

❐ — Exercice **6** —

On dispose d'un ABR. On suppose donc que celui-ci a été correctement construit à partir d'objets Noeud dont les attributs sont toujours **valeur**, **gauche** et **droit**: un sous-arbre gauche (resp. droit) ne contient que des valeurs inférieures (resp. supérieures) à la racine.

Le but de cet exercice est d'écrire la fonction récursive **appartient(x, A)** qui indique si une valeur x appartient à un ABR A. Elle renvoie True si $x \in A$ et False sinon.

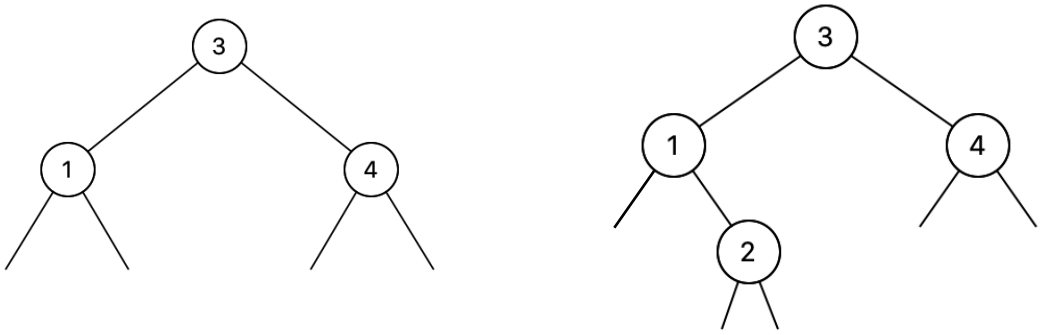
- Étude d'un arbre : on commence la lecture d'un (sous-)ABR à la racine. L'ABR ci-dessous peut donc être décomposé en **1** noeud racine et **deux** sous-arbres non-vides.
  Imaginez un algorithme que l'on doit appliquer pour trouver **67** en répétant toujours la même opération sur des sous-arbres de plus en plus petits. On pourra s'aider du schéma ci-dessous.



- ❖ Condition d'arrêt : On a maintenant une ébauche de fonction récursive. Il reste à déterminer la condition d'arrêt. Imaginez que vous cherchez la valeur 17. Quel test va vous permettre de renvoyer False et d'arrêter votre fonction récursive ?
- ❖ Grâce aux deux questions précédentes, écrire la fonction récursive **appartient(x, A)** prenant en argument une valeur x et un arbre A constitué de Noeuds et renvoyant True si $x \in A$ et False sinon.

Nous allons maintenant discuter de la **création** d'un ABR au travers de l'implémentation d'une fonction **ajoute(x, A).** Cette fonction va permettre de construire un ABR par l'ajout successif de Noeuds d'une valeur x.
Ajouter un nouvel élément dans une ABR n'est en principe pas plus complexe que de le chercher : s'il est plus petit, on va à gauche; s'il est plus grand, on va à droite. Quand on arrive à l'arbre vide, on ajoute le nouveau Noeud.

Exemple :
On rajoute un Noeud dont la valeur est 2 dans l'arbre de gauche. 2 est inférieur à 3 donc, on va dans le sous-arbre gauche. 2 est supérieur à 1 donc, on va dans le sous-arbre droit. Le sous-arbre droit est actuellement à None. On va placer le Noeud de valeur 2 à cet endroit.



Cette idée simple est assez complexe à mettre en oeuvre si l'on souhaite **conserver** l'ABR précédemment créé et le modifier sur place. La fonction **ajoute(x, A)** que nous nous proposons d'écrire ne va donc pas modifier un ABR mais simplement en créer un (ou des) nouveau(x) à chaque ajout.

❐ — Exercice **7** —

En vous inspirant de la fonction **appartient**, écrire la fonction **ajoute(x, A)** qui ajoute x à l'arbre A et renvoie un nouvel arbre.

Contrairement à la fonction **appartient**, lorsque l'arbre A est vide, on ajoutera Noeud dans les deux sous-arbres sont vides et dont la valeur vaut x.

Tester votre fonction ajoute à l'aide des instructions ci-dessous puis explorer votre arbre avec les algorithmes de parcours vus précédemment :

```
A = ajoute(30, None)
print(A.valeur, A.gauche, A.droit)
A = ajoute(10, A)
# affichage avec print
A = ajoute(40, A)
# affichage avec print
```

Remarque sylvestre:

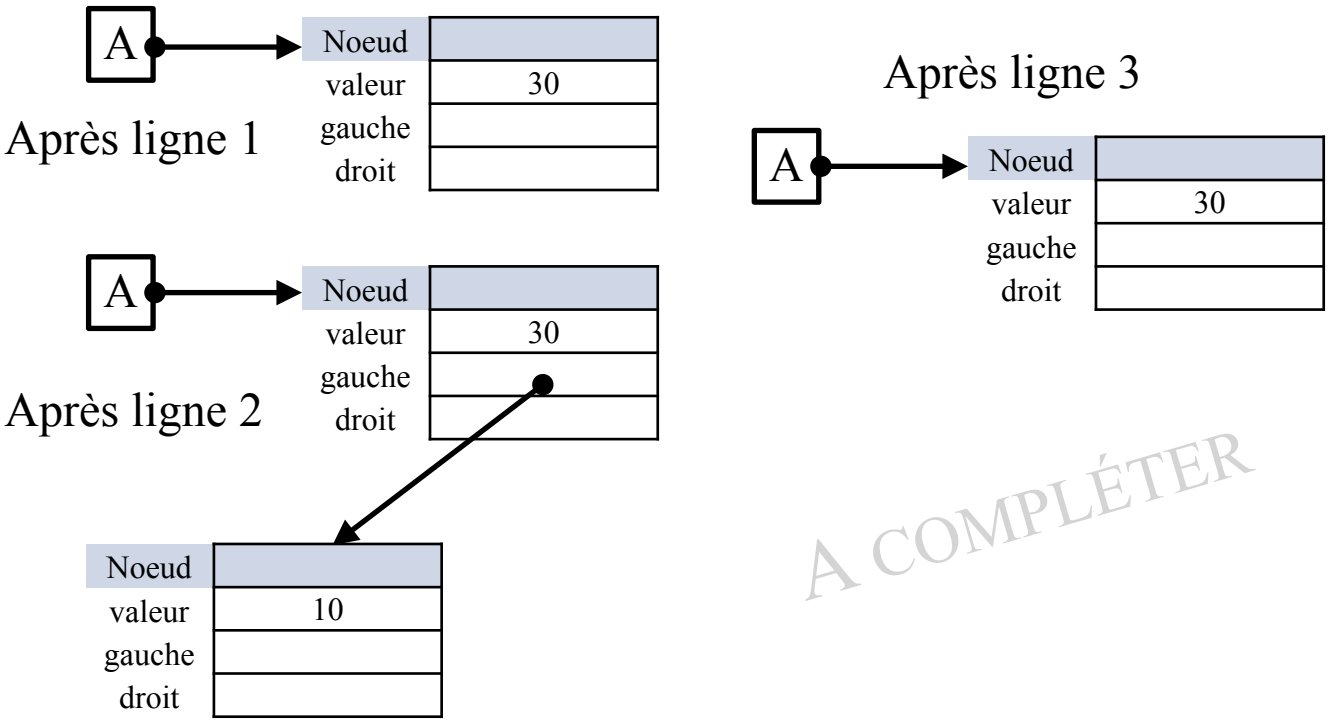
Il est bon de noter que notre arbre A n'est pas complètement recréé à chaque fois. En effet, seuls les Noeuds situés du bon côté seront recréés. Les Noeuds des sous-arbres où l'on ne descend pas seront partagés.

Pour se rendre compte de ces subtilités sylvestres, il faut travailler sur les adresses de l'objet Noeud.

Exercice 8

Nous allons reprendre l'exemple du "à faire vous-même 7".

- En affichant grâce à des print les différentes adresses des noeuds de l'arbre au fur et à mesure de sa construction et en reliant les différents noeuds ci-dessous, expliquez quel(s) sous-arbre(s) ont été dupliqués et quel(s) Noeud(s) ont été recréés



- ❖ On souhaite ajouter un Noeud dont la valeur vaut 31 puis un Noeud dont la valeur vaut 45. Essayez de prédire les créations ou les partages de Noeud qui auront lieu lors de cette opération. Vérifiez ensuite grâce à des affichages d'adresses.

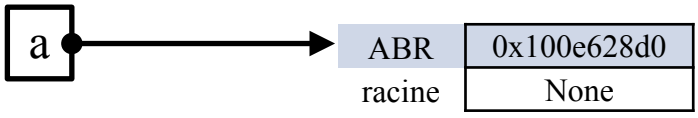
Remarque : Cette dernière partie est très importante pour ceux qui souhaitent continuer l'informatique plus tard. La gestion de la mémoire est en effet une question critique dans les applications actuelles : notre fonction **ajoute** recrée en effet des sous-arbres à chaque appel. Que deviennent les anciens arbres ? Ils sont stockés dans une adresse mémoire jusqu'à ce que l'application crashe[1]. Cela peut mener à une surcharge de la RAM et à un plantage.

3) Encapsulation dans un objet

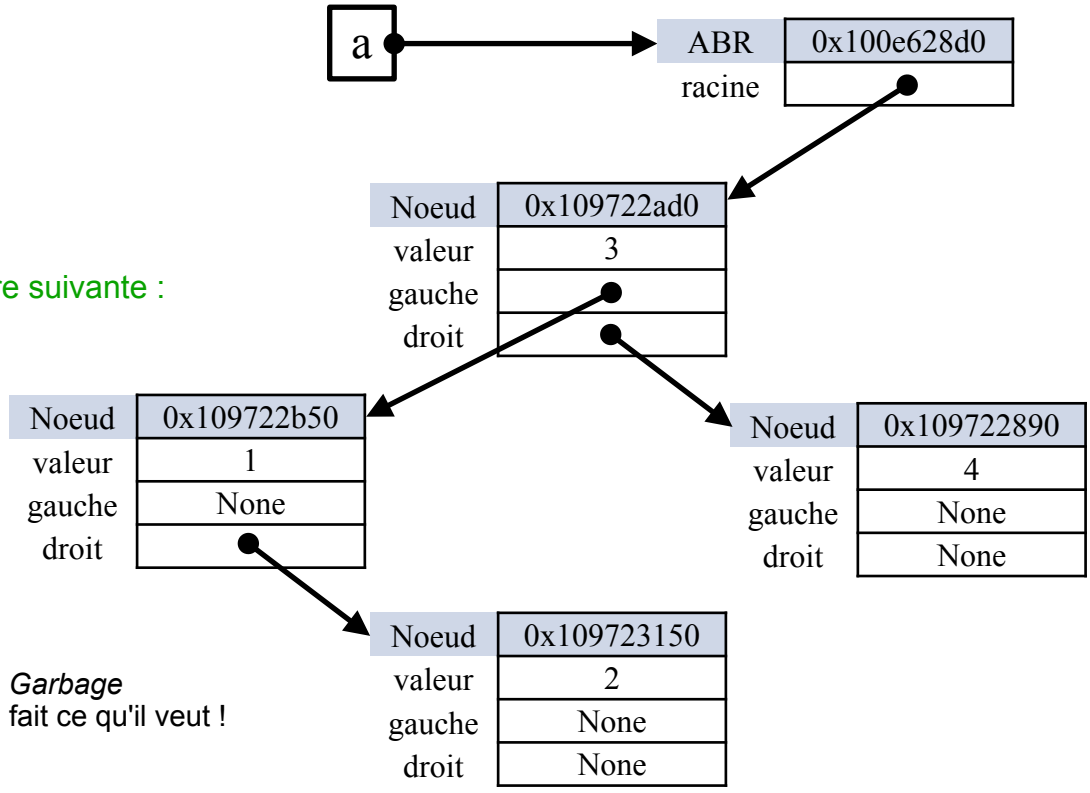
L'encapsulation d'un ABR dans un objet est simple. En effet, un ABR est avant tout un ensemble de noeuds ordonné selon une relation d'ordre (est inférieur ou égal à, lexicographique…). Chaque objet de la classe ABR que nous allons créer va contenir un unique attribut nommé **racine**. Cet attribut représente l'arbre de recherche.

Pour pouvoir construire tous les ABR possibles (y compris l'ABR vide), le constructeur de notre classe doit initialiser l'attribut racine à **None**. On ajoute ensuite grâce aux fonctions précédentes des noeuds à l'arbre de recherche.

Exemple :
a = ABR() instancie un objet avec un attribut racine :



a = ABR()
a.ajouter(3)
a.ajouter(1)
a.ajouter(4)
a.ajouter(2)
donne la structure suivante :



[1] Ils sont gérés par le *Garbage Collector (GC)* qui en fait ce qu'il veut !

❒ — Exercice **9** —

Écrire en Python la classe ABR.
On créera :

- le constructeur tel que décrit ci-dessus ;
- une fonction **ajouter(self, x)** permettant d'ajouter un noeud de valeur x à l'ABR. On appellera la fonction **ajoute** créée précédemment.
- une fonction **contient(self, x)** permettant de savoir si un noeud de valeur x appartient à l'ABR. On appellera la fonction **appartient** créée précédemment.

Testez votre classe en instanciant un objet A auquel vous ajouterez les nombres 3, 1, 4 et 2. Vous devez retrouver l'ABR représenté ci-dessus.

❒ — Exercice **10** —

On dispose maintenant d'une classe ABR que nous allons enrichir de nombreuses fonctions utiles à la manipulation des ABR. Toutes les fonctions seront donc encapsulées dans la classe ABR.

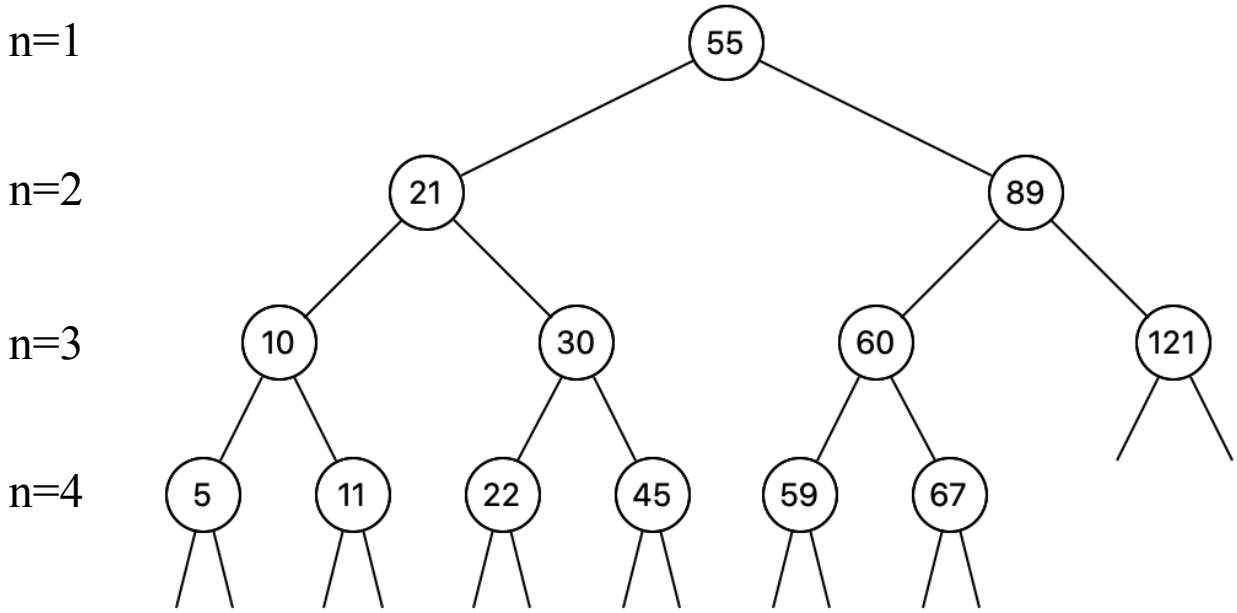
- Encapsulez les fonctions taille, hauteur et parcoursInfixe dans la classe ABR.
- Créez une fonction **min(A)** puis une fonction **max(A)** permettant de trouver le minimum de l'ABR ainsi que le maximum.
- Créez une fonction **supprimerArbre(A)** qui réinitialise la racine de l'arbre à None. Demandez-vous si vous avez bien libéré de la mémoire en "supprimant" l'arbre A.
- Créez une fonction **estValide(A)** qui vérifie si un arbre est un ABR ou non. La fonction renvoie True si l'arbre est un ABR et False sinon.

4) Efficacité des arbres de recherche et d'ajout

Le cout de la recherche et de l'ajout dans un ABR dépend de sa structure. Il est borné par la hauteur de l'ABR. En effet, dans le pire des cas, l'arbre peut être complètement linéaire (tous les sous-arbres sont à gauche par exemple) et le cout d'une recherche est alors **proportionnel au nombre d'éléments de l'arbre**. Cela n'a donc aucun intérêt : autant faire une liste chaînée!

L'intérêt des arbres intervient lorsque ceux-ci sont **équilibrés** : grâce à des algorithmes qui ne sont pas au programme (méthode AVL par exemple), on peut s'assurer que lors de sa construction, tous les noeuds d'un ABR seront à la même hauteur ou à une différence d'une unité.

De manière schématique, dans ces ABR équilibrés, la recherche va permettre **d'éliminer la moitié des Noeuds** de l'arbre à chaque comparaison. Cela va mener à une efficacité logarithmique très rapide.



Dans l'exemple ci-dessus, on s'aperçoit que nous avons environ 2^4 noeuds. Nous avons en fait moins de noeuds mais nous nous plaçons dans le cas le plus défavorable. Dans le pire des cas, l'élément que l'on cherche est une noeud terminal. Dans ce cas, nous devons faire 4 comparaisons, soit l'exposant du nombre de noeuds.

Pour 2^4 noeuds : 4 comparaisons au pire
Pour 2^5 noeuds : 5 comparaisons au pire
…
Pour $N = 2^k$ noeuds : k comparaisons au pire

$$h \leq C \,.\, \log_2(N)$$

hauteur de l'arbre

Conclusion :

Les arbres équilibrés constituent de très bonnes structures de données pour stocker et rechercher de l'information.