Terminale - Spécialité NSI

Les arbres binaires

Le but de cette première partie va être de modéliser un **arbre binaire** à l'aide de la **programmation objet**.

Exercice 1 : Création de la classe NoeudBinaire *

Dans ce premier exercice, vous allez devoir définir la base de la classe **NoeudBinaire** qui servira à construire les arbres. Nous viendrons améliorer cette classe avec différentes implémentations de méthodes dans les prochains exercices.
Pour le moment, la classe **NoeudBinaire** est définie par :

- Un attribut **"valeur"** désignant la valeur "contenue" par le noeud.
- Un attribut **"fils_gauche"**, de type **NoeudBinaire** désignant le descendant à gauche du noeud.
- Un attribut **"fils_droite"**, de type **NoeudBinaire** désignant le descendant à droite du noeud.

Vous l'aurez remarqué, comme les deux fils sont aussi des noeuds binaires, nous sommes donc en train de manipuler une **structure récursive**. L'arbre que nous manipulerons sera en fait **le noeud racine de l'arbre**.
Définissez la classe NoeudBinaire et implémentez son constructeur qui aura pour paramètres (dans l'ordre) : La valeur du noeud, son fils de gauche puis son fils de droite.

```
class NoeudBinaire:
    def __init__(self, valeur, fils_gauche, fils_droite):
        ......
        ......
        ......
```

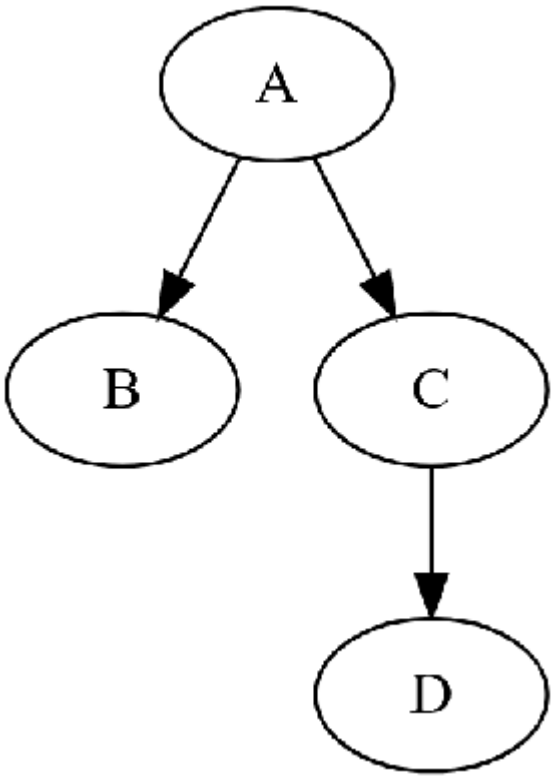
Test

Afin de facilitez vos tests, vous allez importez une fonction (afficher_arbre(arbre)) qui vous permettra de visualiser facilement votre arbre. Veillez à **bien avoir respecté le nom de la classe NoeudBinaire et de ses attributs**. Placez

le fichier **"afficheur_arbre.py"** dans le même dossier que votre fichier python courant.
Importez le module au début de votre fichier ainsi :

```
from afficheur_arbre import afficher_arbre
```

```
#Test
>>> D = NoeudBinaire("D", None, None)
>>> C = NoeudBinaire("C", D, None)
>>> B = NoeudBinaire("B", None, None)
>>> A = NoeudBinaire("A", B, C)
>>> mon_arbre = A
>>> afficher_arbre(mon_arbre)
```



Pour la suite des exercices de cette section, nous nous baserons sur l'arbre suivant, que nous nommerons **"arbre_binaire"**:

Exercice 2 : Création de l'arbre *

A l'aide des lignes de codes utilisées pour tester votre classe **NoeudBinaire** lors de l'exercice précédent, **créez l'arbre ci-dessus (arbre_binaire) en**

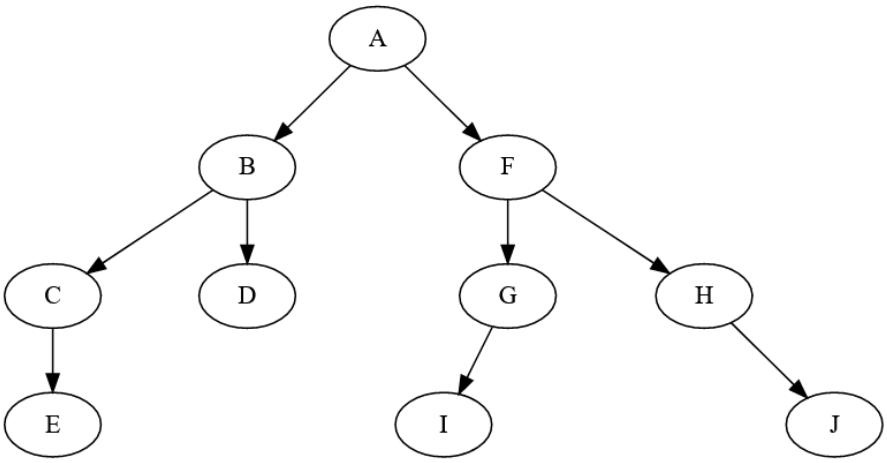


Figure 1: Un arbre binaire "arbre_binaire"

python.
Gardez le dans votre fichier python en le stockant dans une variable **arbre_binaire** (donc pas dans la console) car vous le réutiliserez plus tard.
Enfin, **visualisez le résultat** pour valider en utilisant (cette fois-ci dans la console) la fonction **afficher_arbre**.

Exercice 3 : Taille de l'arbre **

La **taille** d'un arbre binaire correspond au **nombre de noeuds** de l'arbre.
1. Donnez la taille de l'arbre "arbre_binaire".
D'un point de vue algorithmique, la méthode **récursive** permettant de déterminer la taille d'un arbre à partir du noeud racine est la suivante :

1. On initialise une variable, comptant le nombre de noeuds, à 1 (car on compte le noeud courant)
2. Si le noeud possède un fils à gauche, on demande de calculer la taille du sous-arbre ayant pour racine ce fils **(récursivité)** et on incrémente notre variable initiale avec la taille de ce sous-arbre.
3. Idem avec le fils à droite.
4. On retourne la variable comptant le nombre de noeuds.

2. Implémentez et testez (sur **arbre_binaire**) la méthode **"taille(self)"** de la classe **NoeudBinaire** qui renvoie la taille de l'arbre ayant pour racine le noeud sur lequel la méthode est appelée..

```
def taille(self):
```

```
.........
.........
.........
```

Exercice 4 : Hauteur de l'arbre **

La **hauteur** d'un arbre binaire correspond à **la profondeur maximale** de l'arbre, c'est à dire, le niveau où se trouve le noeud (feuille) le plus éloigné de la racine.

1. Donnez la hauteur de l'arbre "arbre_binaire".

D'un point de vue algorithmique, la méthode **récursive** permettant de déterminer la hauteur d'un arbre à partir du noeud racine est la suivante :

1. On initialise deux variables (à 0) qui stockeront la hauteur du sous-arbre gauche puis droit qui descendent de la racine (noeud courant sur lequel l'algorithme est appelé).
2. Si le noeud possède un fils à gauche, on stocke dans la variable associée la hauteur du sous-arbre ayant pour racine ce fils **(récursivité)**.
3. Idem avec le fils à droite.
4. On détermine la plus grande hauteur entre celle du sous-arbre gauche et le sous-arbre droit (**indice : fonction "max" de python**).
5. On retourne 1 + cette dernière valeur (car on compte le noeud courant + ma hauteur max des sous-arbres).

2. Implémentez et testez (sur **arbre_binaire**) la méthode **"hauteur(self)"** de la classe **NoeudBinaire** qui renvoie la hauteur de l'arbre ayant pour racine le noeud sur lequel la méthode est appelée.

```
def hauteur(self):
    .........
    .........
    .........
```

Exercice 5 : Feuilles d'un arbre **

Une **feuille d'un arbre** est un noeud qui ne possède **aucun fils** (à gauche et à droite).

1. Donnez le nombre de feuilles de l'arbre "arbre_binaire".

2. Implémentez et testez la méthode **"est_feuille(self)"** de la classe **NoeudBinaire** qui renvoie **True** sir le noeud est une feuille et **False** sinon.

```
def est_feuille(self):
    .........
```

```
.........
.........
```

D'un point de vue algorithmique, la méthode **récursive** permettant de déterminer la le nombre de feuilles d'un arbre à partir du noeud racine est la suivante :

1. Cas de base : Si le noeud courant est une feuille, on retourne 1.
2. Sinon, :
   (a) On initialise une variable, comptant le nombre de feuilles, à 0
   (b) Si le noeud possède un fils à gauche, on demande de calculer le nombre de feuilles dans le sous-arbre ayant pour racine ce fils **(récursivité)** et on incrémente notre variable initiale avec ce nombre.
   (c) Idem avec le fils à droite.
   (d) On retourne la variable comptant le nombre de feuilles.

3. Implémentez et testez (sur **arbre_binaire**) la méthode **"nombre_feuilles(self)"** de la classe **NoeudBinaire** qui renvoie le nombre de feuilles contenues dans l'arbre ayant pour racine le noeud sur lequel la méthode est appelée.

```
def nombre_feuilles(self):
    .........
    .........
    .........
```

Parcourir les arbres

Le "parcours" d'un arbre désigne un ordre d'affichage des valeurs des noeuds de l'arbre, à partir du noeud racine.
Nous allons voir quatre types de parcours :

- Le parcours **préfixe**.
- Le parcours **suffixe**.
- Le parcours **infixe**.
- Le parcours **en largeur**.

Dans les différentes méthodes que vous allez coder, **l'affichage de la valeur du noeud** sera réalisée avec la fonction **print**.

Exercice 6 : Le parcours préfixe *

Le parcours **préfixe** d'un arbre se déroule ainsi :

1. On affiche le noeud racine.

2. Si le noeud possède un fils à gauche, on réalise le parcours préfixe du sous-arbre gauche.
3. Idem pour le fils à droite.

Implémentez et testez (sur **arbre_binaire**) la méthode **"parcours_prefixe(self)"** de la classe **NoeudBinaire** qui réalise le **parcours préfixe** sur l'arbre ayant pour racine le noeud sur lequel la méthode est appelée. La fonction développée ne renvoie rien, elle se contente d'afficher les valeurs des noeuds.
Le **parcours préfixe** de **arbre_binaire** devrait vous donner : A, B, C, E, D, F, G, I, H, J

```
def parcours_prefixe(self):
    .........
    .........
    .........
```

Exercice 7 : Le parcours suffixe *

Le parcours **suffixe** d'un arbre se déroule ainsi :

1. Si le noeud (racine) possède un fils à gauche, on réalise le parcours suffixe du sous-arbre gauche.
2. Idem pour le fils à droite.
3. On affiche le noeud racine.

Implémentez et testez (sur **arbre_binaire**) la méthode **"parcours_suffixe(self)"** de la classe **NoeudBinaire** qui réalise le **parcours suffixe** sur l'arbre ayant pour racine le noeud sur lequel la méthode est appelée. La fonction développée ne renvoie rien, elle se contente d'afficher les valeurs des noeuds.
Le **parcours suffixe** de **arbre_binaire** devrait vous donner : E, C, D, B, I, G, J, H, F, A

```
def parcours_suffixe(self):
    .........
    .........
    .........
```

Exercice 8 : Le parcours infixe *

Le parcours **infixe** d'un arbre se déroule ainsi :

1. Si le noeud (racine) possède un fils à gauche, on réalise le parcours suffixe du sous-arbre gauche.
2. On affiche le noeud racine.
3. Si le noeud (racine) possède un fils à droite, on réalise le parcours suffixe du sous-arbre droit.

Implémentez et testez (sur **arbre_binaire**) la méthode **"parcours_infixe(self)"** de la classe **NoeudBinaire** qui réalise le **parcours infixe** sur l'arbre ayant pour racine le noeud sur lequel la méthode est appelée. La fonction développée ne renvoie rien, elle se contente d'afficher les valeurs des noeuds.
Le **parcours infixe** de **arbre_binaire** devrait vous donner : C, E, B, D, A, I, G, F, H, J

```
def parcours_infixe(self):
    .........
    .........
    .........
```

Exercice 9 : Le parcours en largeur **

Le parcours **en largeur** diffère des parcours précédents.
Nous n'utiliserons pas d'algorithme récursif ici, mais une structure adaptée que vous connaissez bien : La file.
La méthode pour effectuer un parcours **en largeur** est la suivante :

1. On crée une file dans laquelle on ajoute le noeud (racine)
2. On boucle tant qu'il y a des éléments à traiter dans la file
   (a) On retire un élément de la file qui constitue le noeud courant et on l'affiche.
   (b) Si le noeud (courant) possède un fils à gauche, on ajoute ce fils à la file.
   (c) Idem pour le fils à droite.

1. Pourquoi parle-t-on de parcours en largeur ?
Importez la classe File issue du dernier TP (programamtion objet) dans votre programme. La classe doit se nommer "File" et doit se trouver dans un fichier **"File.py"**. Vous pouvez utiliser le fichier **"File.py"** corrigé et fourni par le professeur.

```
from File import File
```

2. Implémentez et testez (sur **arbre_binaire**) la méthode **"parcours_largeur(self)"** de la classe **NoeudBinaire** qui réalise le **parcours en largeur** sur l'arbre ayant pour racine le noeud sur lequel la méthode est appelée. La fonction développée ne renvoie rien, elle se contente d'afficher les valeurs des noeuds.
Le **parcours en largeur** de **arbre_binaire** devrait vous donner : A, B, F, C, D, G, H, E, I, J

```
def parcours_largeur(self):
    .........
    .........
    .........
```

3. On souhaite déterminer la **complexité en espace** du **parcours en largeur**. Pour rappel, la complexité en espace consiste à mesurer l'espace (mémoire) utilisé par un algorithme, en fonction de ses entrées. Par exemple, pour un algorithme qui prendrait un nombre n en entrée et qui pour chaque nombre "x" de 0 à n stockerait dans une liste x et -x, la complexité en espace serait donc de O(2n) car il y aurait 2n nombres dans la liste.
Dans notre cas, l'entrée est donc un arbre binaire possédant plusieurs noeuds.

- (a) Sur quel(s) élément(s) peut-on se baser pour évaluer la **complexité en espace** ?
- (b) Combien de noeuds a-t-on (au maximum) à la profondeur 1 (racine), 2, 3 et 4 ?
- (c) Pour un niveau de profondeur **n**, déterminez le nombre maximum de noeuds présents à cette profondeur.
- (d) Déduisez en **la complexité en espace du parcours en largeur**.

Arbres binaires de recherche

Un **arbre binaire de recherche** est un type **d'arbre binaire** qui respecte les 2 règles.
A partit d'un noeud de l'arbre :

- La valeur du noeud du fils gauche est **inférieure (strictement)** à la valeur du noeud.
- La valeur du noeud du fils droit est **supérieure ou égale** à la valeur du noeud.

Comme la structure est **récursive** on peut généraliser ainsi :
A partit d'un noeud de l'arbre :

- **L'ensemble des valeurs** des noeuds du sous-arbre gauche sont **inférieures (strictement)** à la valeur du noeud.
- **L'ensemble des valeurs** des noeuds du sous-arbre droit sont **supérieures ou égales** à la valeur du noeud.

Tant qu'il existe une **relation d'ordre** entre les éléments (c'est à dire qu'on peut comparer les éléments entre eux), on peut stocker n'importe quelle type de valeur dans un **arbre binaire de recherche**.

Exercice 10 : Création de la classe NBR (pour Noeud binaire de recherche) *

Dans cette exercice, vous allez devoir définir la base de la classe **NBR** qui servira à construire les **arbres binaires de recherche**. Nous viendrons améliorer cette classe avec différentes implémentations de méthodes dans les prochains exercices.

Pour le moment, la classe **NBR** est définie par les **mêmes attributs que la classe NoeudBinaire**
Définissez la classe NoeudBinaire et implémentez son constructeur qui aura un seul paramètre : La valeur "contenue" dans le noeud. Les deux autres noeuds (fils_gauche et fils_droite) seront initialisés à **None** dans le constructeur. En effet, comme l'insertion dans un **arbre binaire de recherche**, l'insertion d'un nouveau noeud est particulière, les fils ne sont pas passés en paramètre au constructeur.
Vous écrirez la déclaration de la classe en rajoutant, (entre parenthèses) **"NoeudBinaire" après NBR**, de cette manière : **class NBR(NoeudBinaire)**. Ce mécanisme est appelé **héritage**. Cette notion ne sera pas travaillée cette année, mais, dans les grandes lignes, cela permet à la classe **NBR** de possèder les **mêmes méthodes (code)** que la classe **NoeudBinaire** sans avoir besoin de récrire le code dans la nouvelle classe. Ainsi, avec **NBR**, vous créez en fait une extension de **NoeudBinaire**.

```
class NBR(NoeudBinaire):
    def __init__(self, valeur):
        ......
        ......
        ......

#Même si on n'a pas défini ces méthodes dans la nouvelle classe on peut faire :
>>> abr = NBR(5)
>>> abr.hauteur() # Renvoie 1
>>> abr.taille() # Renvoie 1
```

Test

Comme l'insertion nécessite une fonction que nous allons développer juste après, vous ne pourrez pas vraiment tester à cette étape. Vous pouvez toujours créer un noeud unique et le visualiser.

```
>>> abr = NBR(10)
>>> afficher_arbre(abr)
```

Pour la suite des exercices de cette section, nous nous baserons sur l'arbre suivant, que nous nommerons **"arbre_binaire_recherche"**:

Exercice 11 : Insertion d'une valeur dans un arbre binaire de recherche **

Quand on veut insérer une nouvelle valeur dans un **arbre binaire de recherche**, à partir de sa racine, il faut chercher son emplacement tout en respectant les règles de l'arbre.
Comme on est dans une structure récursive, l'algorithme pour l'ajouter est plutôt simple.

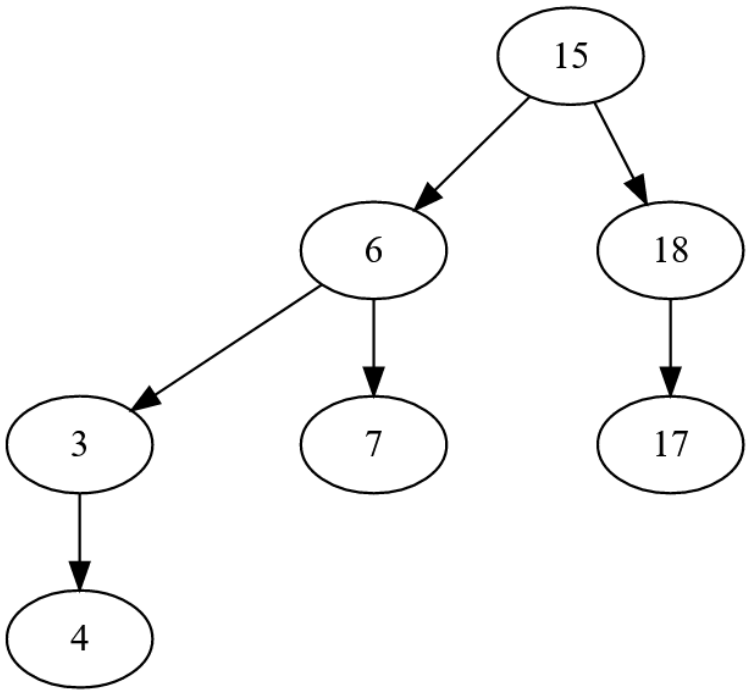


Figure 2: Un arbre binaire de recherche "arbre_binaire_recherche"

Il faut donc, à partir de la valeur que l'on veut ajouter :

1. Si la valeur est plus petite (strictement) que la valeur du noeud racine, deux cas :
   (a) Le noeud possède un fils à gauche, on insère donc la valeur dans le sous-arbre gauche (récursivité)
   (b) Le noeud n'a pas de fils à gauche, on a donc localisé la position de la valeur. On crée un nouveau Noeud binaire de recherche (NBR) qui deviendra le fils gauche du noeud.
2. Si la valeur est plus grande ou égale que la valeur du noeud racine, on fait la même chose mais à droite.

Par exemple, si on veut ajouter la valeur **"14"** dans "arbre_binaire_recherche", celle si sera placée comme fils droit du noeud contenant la valeur "7".

1. On souhaite ajouter les valeurs 2, 13, 9 et 20 dans "arbre_binaire_recherche". Déterminez les positions des nouveaux noeuds.

2. Implémentez la méthode **"inserer_valeur(self, x)"** de la classe **NBR** qui ajoute la valeur **x** dans l'arbre binaire de recherche ayant pour racine le noeud sur lequel la méthode est appelée.

```
def inserer_valeur(self, x):
    ..........
    ..........
    ..........
```

3. Pour tester, à l'aide de "inserer_valeur" **créez l'arbre "arbre_binaire_recherche"** présenté plus tôt en le stockant dans une variable du même nom. **Attention à bien respecter l'ordre d'ajout** (valeurs de haut en bas). Visualisez votre nouvel arbre avec la fonction **afficher_arbre**.

```
#Debut du test...
arbre_binaire_recherche = NBR(15)
arbre_binaire_recherche.inserer_valeur(6)
#A vous de compléter...
```

4. Ajoutez les valeurs 2, 13, 9 et 20 dans votre arbre (numérique cette fois) et vérifier l'exactitude de vos réponses à la question 1.

Exercice 12 : Recherche d'une valeur **

Nous aimerions maintenant développer une méthode pour savoir si un arbre contient **valeur x donnée**. Pour cela, on pourrait par exemple vérifier tous les noeuds de l'arbre un par un, mais cela ne serait pas efficace. En effet, pour déterminer si une valeur est contenue dans l'arbre binaire de recherche, il n'est **pas nécessaire de passer par tous les noeuds.**

1. Déterminez un algorithme permettant de vérifier la présence d'une valeur "x" dans un arbre binaire de recherche (sans passer par tous les noeuds!). Cet algorithme renvoie **True** si la valeur est contenue dans l'arbre et **False** sinon.

2. Implémentez et testez (sur **arbre_binaire_recherche**) la méthode **"contient_valeur(self, x)"** de la classe **NBR** (où **x** est la valeur recherchée) qui implémente votre algorithme.

```
def contient_valeur(self, x):
    ..........
    ..........
    ..........
```

Test

```
>>> arbre_binaire_recherche.contient_valeur(15)
True
>>> arbre_binaire_recherche.contient_valeur(7)
True
>>> arbre_binaire_recherche.contient_valeur(25)
False
>>> arbre_binaire_recherche.contient_valeur(-1)
False
```

Exercice 13 : Trace de recherche ***

On reprend la méthode de recherche **développée lors de l'exercice 12** sauf qu'au lieu de renvoyer "True" ou "False" si la valeur est présente ou non dans l'arbre binaire de recherche, on veut réaliser une **trace de recherche**.
Le but est d'afficher chaque étape, quelle valeur on est en train de vérifier, par quel noeud on passe...L'affichage s'arrête quand la valeur est trouvée (et affichée) ou que la valeur est introuvable (on affiche alors "Introuvable").

Exemple

```
>>> arbre_binaire_recherche.trace_recherche(7)
15
Gauche
6
Droite
7
>>> arbre_binaire_recherche.trace_recherche(16)
15
Droite
18
Gauche
```

```
17
Gauche
Introuvable
```

Implémentez et testez (sur **arbre_binaire_recherche**) la méthode **"trace_recherche(self, x)"** de la classe **NBR** qui effectue la recherche d'une valeur en affichant une trace (comme ci-dessus). Cette méthode ne renvoie rien, elle se contente d'afficher.

```
def trace_recherche(self, x):
    ..........
    ..........
    ..........
```

Exercice 14 : Recherche des extremums **

On souhaite développer deux algorithmes permettant de trouver la plus petite ou la plus grande valeur de l'arbre binaire, encore une fois, **sans passer par tous les noeuds de l'arbre**.

1. Déterminez un algorithme (récursif) permettant de trouver **la plus petite valeur** d'un arbre binaire de recherche (sans passer par tous les noeuds!). Cet algorithme renvoie cette valeur.

2. Faites de même avec la plus grande valeur de l'arbre.

3. Implémentez et testez (sur **arbre_binaire_recherche**) la méthode **"valeur_minimale(self)"** de la classe **NBR** qui implémente votre algorithme de recherche de la plus petite valeur contenue dans l'arbre.

4. Implémentez et testez (sur **arbre_binaire_recherche**) la méthode **"valeur_maximale(self)"** de la classe **NBR** qui implémente votre algorithme de recherche de la plus grande valeur contenue dans l'arbre.

```
def valeur_minimale(self):
    ..........
    ..........
    ..........

def valeur_maximale(self):
    ..........
    ..........
    ..........
```

Test

```
>>> arbre_binaire_recherche.valeur_minimale()
2 #Ou 3 si vous n'avez pas sauvegardé l'ajout de 2 (exercice 11)
>>> arbre_binaire_recherche.valeur_maximale()
20 #Ou 18 si vous n'avez pas sauvegardé l'ajout de 20 (exercice 11)
```

Exercice 15 : Une structure triée ***

Nous avons vu que, de par ses règles de composition, un arbre binaire de recherche est une structure triée.

1. Déterminez quel type de parcours d'arbre permet d'afficher la liste des valeurs de l'arbre dans l'ordre croissant.

2. Après avoir identifié ce parcours, implémentez une méthode (dans NoeudBinaire, et du nom de votre choix) qui réalise **le même parcours d'arbre** (identifié à la question 1) mais qui, au lieu d'afficher les valeurs dans la console, **stocke celles-ci dans une liste** qui sera renvoyée à la fin.
La méthode **"extend"** (utilisable sur les listes) vous sera sûrement utile.

3. Implémentez et testez en dehors des classes une **fonction "tri_liste_par_arbre(l)"** qui prend une **liste "l"** en entrée et **renvoie une nouvelle liste triée**. Le but l'algorithme implémenté est de stocker toutes les valeurs de la liste dans un nouvelle arbre binaire et de renvoyer la liste des éléments triés avec **la méthode développée à la question 2**.

```
def tri_liste_par_arbre(l):
    ..........
```

Exercice 16 : Valeur la plus proche ****

Pour une valeur **x** donnée, il se peut qu'un arbre binaire de recherche ne la contienne pas. On aimerait développer un algorithme qui permettrait de donner **la valeur (contenue dans l'arbre) la plus proche (mathématiquement) de celle recherchée**.

L'algorithme n'est pas extrêmement compliqué, mais n'est pas totalement évidemment non plus. La principale difficulté est que "le plus proche (géographiquement) dans l'arbre" ne signifie pas forcément "la valeur la plus proche (mathématiquement)". Si on trouve une valeur qui ne correspond pas, il ne faut pas immédiatement la rejeter, car c'est peut-être la plus proche. ici aussi, **il n'y a pas besoin de parcourir tous les noeuds** pour déterminer cette valeur.

Implémentez et testez la méthode **"valeur_la_plus_proche(self, x)"** de la classe **NBR** qui retourne la valeur contenu dans l'arbre la plus proche (mathématiquement) de la valeur x passée en paramètre.

Vous pouvez vous aider de la méthode **contient_valeur** développée lors de l'exercice 12. (reprendre son code pour développer la nouvelle méthode).

```
def valeur_la_plus_proche(self):
    ..........
    ..........
    ..........
```

Test

```
>>> arbre_binaire_recherche.valeur_la_plus_proche(15)
15
>>> arbre_binaire_recherche.valeur_la_plus_proche(5)
4
>>> arbre_binaire_recherche.valeur_la_plus_proche(4)
4
>>> arbre_binaire_recherche.valeur_la_plus_proche(14)
15
>>> arbre_binaire_recherche.valeur_la_plus_proche(30)
20 #Ou 18 si vous n'avez pas sauvegardé l'ajout de 20 (exercice 11)
>>> arbre_binaire_recherche.valeur_la_plus_proche(11)
13 #Ou 7 si vous n'avez pas sauvegardé l'ajout de 20 (exercice 11)
```