Arbres Binaires de Recherche (ABR) 2 : Implémentation

1. Représentation en python

- Un ABR est un arbre binaire. On le représente donc à l'aide des classes et méthodes précédemment définies.On supposera que les conditions d'ordre sur les valeurs des noeuds sont vérifiées dans les arbres qui seront créés.
- Rappel : Un arbre binaire de recherche est un arbre binaire dont les noeuds contiennent des valeurs qui peuvent être comparées entre elles et tel que que pour tout noeud de l'arbre, toutes les valeurs situées dans le sous-arbre gauche (ou droit) sont plus petite (ou plus grande) que la valeur située dans le noeud.
- En particulier, le parcours infixe d'un ABR affiche les valeurs contenues dans l'arbre par ordre croissant.

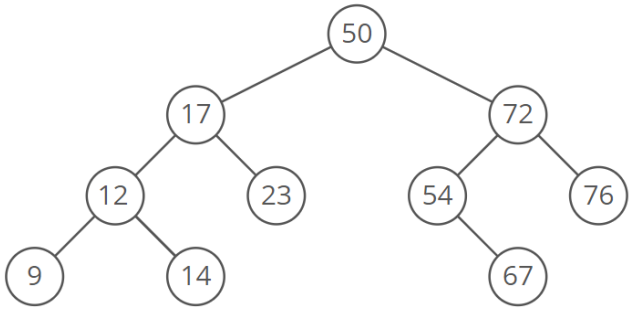
In [2]:

```
class Noeud:
    '''Classe définissant un noeud d'arbre binaire'''
    def __init__(self,gauche,valeur,droite):
        self.gauche=gauche
        self.valeur=valeur
        self.droite=droite

    def __str__(self):
        '''Renvoie la valeur du noeud (str)'''
        return str(self.valeur)

    def ajoute(self,x):
        '''ajoute un noeud en modifiant
        les liens des noeuds entre eux'''
        if self.valeur is None:
            self.valeur=x
        else:
            if x > self.valeur:
                if self.droite is None:
                    self.droite=Noeud(None,x,None)
                else:
                    self.droite.ajoute(x)
            else:
                if self.gauche is None:
                    self.gauche=Noeud(None,x,None)
                else:
                    self.gauche.ajoute(x)


class ABR:
    '''Classe définissant un Arbre Binaire de Recherche'''
    def __init__(self, racine=None):
        self.racine=racine

    def est_vide(self):
        '''renvoie True si et seulement si l'arbre est vide'''
        return self.racine is None

    def hauteur(self):
        '''renvoie la hauteur de l'arbre'''
        if self.racine is None:
            return 0
        else:
            return max(ABR(self.racine.gauche).hauteur(),ABR(self.racine.droite).hauteur())+1

    def taille(self):
        '''renvoie la taille de l'arbre'''
        if self.racine is None:
            return 0
        else:
            return ABR(self.racine.gauche).taille() + ABR(self.racine.droite).taille()+1

    def parcours_infixe(self):
        '''Affiche les noeuds de l'arbre
        parcouru en profondeur préfixe'''

        if self.racine is None:
            return # on sort de la fonction

        ABR(self.racine.gauche).parcours_infixe()
        print(self.racine, end=' ')
        ABR(self.racine.droite).parcours_infixe()

    #Ex 2 : recherche
    def recherche(self, x):
        '''Renvoie True si x appartient à l'arbre
        False sinon'''
        pass

    def ajouter(self,x):
        '''ajoute le noeud de valeur x
        dans l'arbre binaire de recherche'''
```

```
        if self.racine is None:
            self.racine=Noeud(None,x,None)
        else:
            self.racine.ajoute(x)

    #Ex 4 :
    def minimum(self):
        '''Renvoie la valeur minimale
        contenue dans l'arbre'''
        pass

    #Ex 4 :
    def maximum(self):
        '''Renvoie la valeur maximale
        contenue dans l'arbre'''
        pass


    #Ex 6:
    def recherche_detail(self, x):
        '''Renvoie True si x appartient à l'arbre
        False sinon.
        Affiche aussi les sommets visités lors de la recherche'''
        pass
```

Exercice 1:

Construire l'ABR ci-contre, vérifier sa taille, sa hauteur et le résultat de son parcours infixe à l'aide des méthodes contenues dans le code précédent.



In [13]:
```
#Réponse
```

In [3]:
```
print(abr.hauteur())
```

```
4
```

In [4]:
```
print(abr.taille())
```

```
10
```

In [5]:
```
abr.parcours_infixe()
```

```
9 12 14 17 23 50 54 67 72 76
```

2. Recherche

Pour chercher une valeur dans un ABR, il suffit de la comparer à la valeur à la racine puis, si elle est différente, de se diriger vers un seul des deux sous-arbres. On élimine ainsi complètement la recherche dans l'autre sous-arbre.

Exercice 2 :

Dans le code de la classe, compléter la méthode `recherche(self, x)` qui renvoie `True` si la valeur x est contenue dans l'arbre et `False` sinon.

Principe :

- Si l'arbre est vide, on renvoie `False`
- Sinon :
  - Si la valeur de la racine est x, on renvoie True
  - Si x est inférieur à la valeur de la racine, on cherche de façon récursive dans le sous-arbre gauche.
  - Si x est supérieur à la valeur de la racine, on cherche de façon récursive dans le sous-arbre droit.

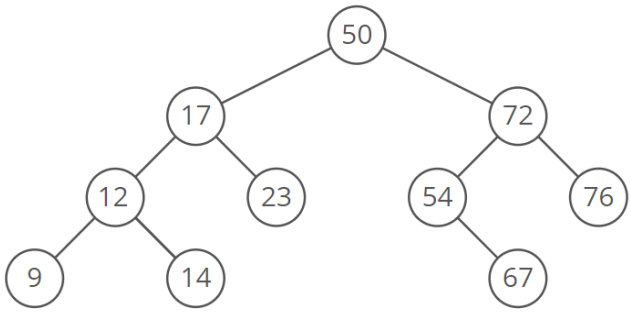
In [6]:
```
#test recherche dans un ABR
assert(abr.recherche(50)==True)
assert(abr.recherche(8)==False)
assert(abr.recherche(54)==True)
assert(abr.recherche(67)==True)
```

3. Ajout

- Pour pouvoir construire un ABR, nous allons utiliser la méthode `ajouter(self,x)` qui ajoute l'élément x dans l'arbre. Ainsi , nous allons construire un ABR par ajout successifs avec cette méthode à partir d'un arbre éventuellement vide.
- En principe, ajouter un nouvel élément dans un ABR n'est pas plus compliqué 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 un arbre vide, on ajoute un nouveau noeud.
- En pratique, c'est un peu plus compliqué, car l'ajout dans un arbre vide pose problème. Dans le code des classes, la classe `Noeud` est également modifiée pour pouvoir redéfinir le lien entre les noeuds lors d'un ajout.

Exercice 3 :

Construire l'ABR représenté ci-dessous en utilisant la méthode `ajouter` .



In [7]:
```
#Réponse
abr=ABR()
```

In [8]:
```
#test ajouter dans un ABR
print(abr.taille()) #10
print(abr.hauteur()) #4
print(abr.racine.gauche) #17
abr.parcours_infixe() #9 12 14 17 23 50 54 67 72 76
```

```
10
4
17
9 12 14 17 23 50 54 67 72 76
```

4. Exercices

Exercice 4:

Dans la classe `ABR` :

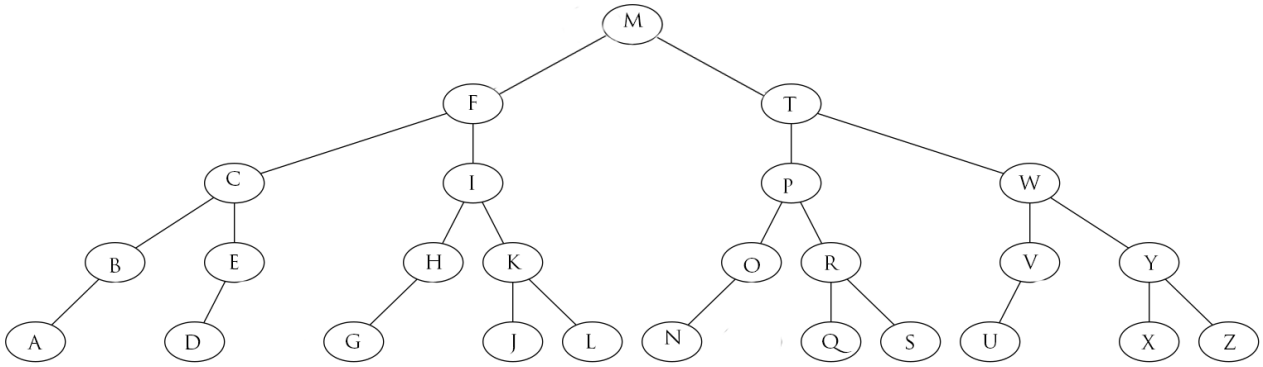
1. Ecrire la méthode `minimum(self)` qui renvoie la valeur minimale contenue dans un ABR.
2. Ecrire la méthode `maximum(self)` qui renvoie la valeur maximale contenue dans un ABR.

Dans les deux cas, si l'arbre est vide, la méthode renvoie `None` .On pourra écrire de façon récursive ou pas...

In [11]:
```
#tests minimum et maximum sur l'arbre de l'exercice 3
assert(ABR().minimum()==None)
assert(abr.minimum()==9)
assert(ABR().maximum()==None)
assert(abr.maximum()==76)
```

Exercice 5 :

On considère l'arbre binaire de recherche ci-dessous permettant de faciliter une recherche alphabétique :



1. Construire cet arbre.
2. Vérifier :
   - Que son parcours infixe affiche bien les lettres dans l'ordre alphabétique.
   - Sa taille
   - Sa hauteur

In [27]:
```
#1.



#2.
```

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
26
5
```

Exercice 6:

Dans la classe `ABR` , Ecrire la méthode `recherche_detail(self,x)` qui en plus de renvoyer `True` ou `False` (selon si x est dans l'arbre ou pas) affiche aussi le nombre de noeuds visités lors de la recherche de la clé `x` ainsi que la valeur de chaque noeud visité.

In [37]:
```
# Tests recherche détail
alphabet.recherche_detail('H')
```

```
M
F
I
H
```

Out[37]: True

In [38]:
```
alphabet.recherche_detail('R')
```

```
M
T
P
R
```

Out[38]: True

In [ ]: