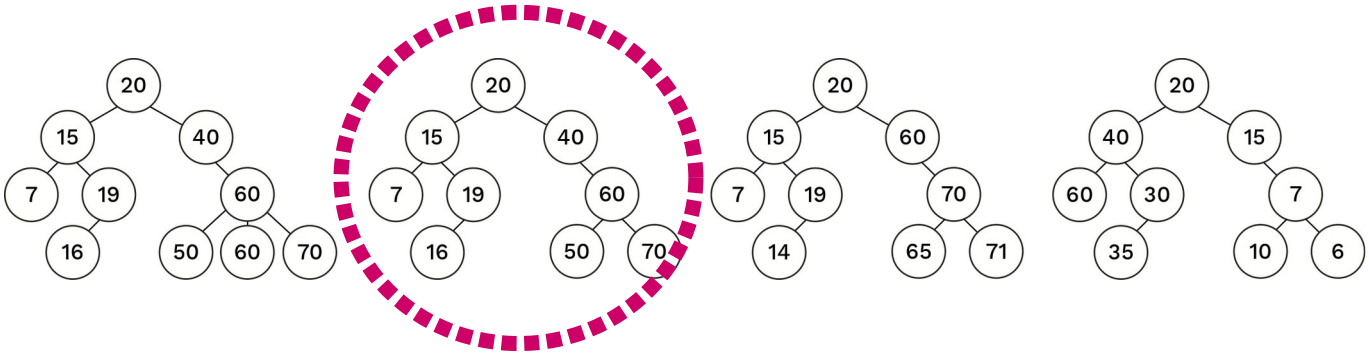
Arbres binaires de recherche

En informatique, un arbre binaire de recherche ou *ABR* (en anglais, *binary search tree* ou *BST*) est une structure de données représentant tableau associatif (tel qu'un dictionnaire en Python). Un arbre binaire de recherche permet des opérations rapides pour rechercher une clé, insérer ou supprimer une clé.

Construction d'un arbre binaire de recherche

Un ABR est un arbre binaire dans lequel toutes les valeurs dans le sous-arbre gauche d'un nœud sont inférieures à la valeur à la racine de l'arbre et toutes les valeurs dans le sous-arbre droit d'un nœud sont supérieures ou égales à la valeur à la racine de l'arbre.

1. **Entourer** dans les exemples suivants, les structures répondant aux caractéristiques d'un ABR.



Recherche d'un élément dans un ABR

La recherche dans un arbre binaire d'un nœud ayant une clé particulière est un procédé récursif. On commence par examiner la racine. Si sa clé est la clé recherchée, l'algorithme se termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même si la clé recherchée est strictement supérieure à la clé de la racine, la recherche continue dans le sous-arbre droit.
Si on atteint une feuille dont la clé n'est pas celle recherchée, on sait alors que la clé recherchée n'appartient à aucun nœud, elle ne figure donc pas dans l'arbre de recherche. On peut comparer l'exploration d'un arbre binaire de recherche avec la recherche par dichotomie qui procède à peu près de la même manière.
L'implémentation en langage Python de cet algorithme peut être effectuée en programmation objet de la façon suivante :

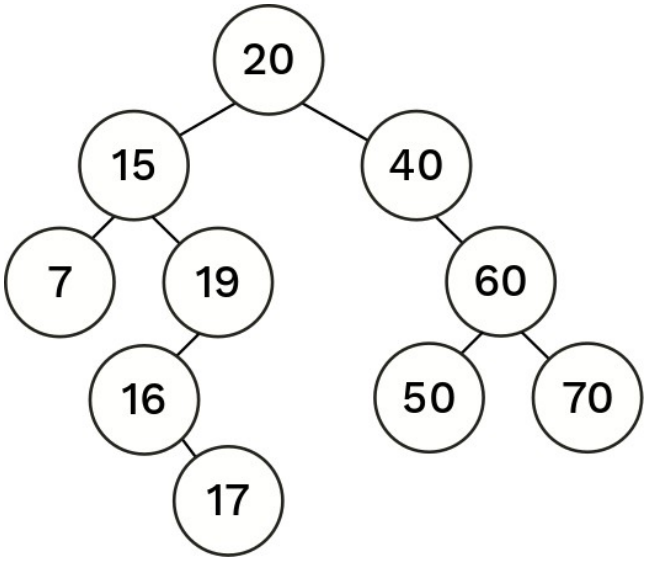
```python
def rechercher(self, n : int) :
    '''Recherche la presence de n dans l'arbre'''
    if self.noeud() == n : return True
    elif self.noeud() > n :
          if self.sag() == None : return False
          else : return self.sag().rechercher(n)
    else :
          if self.sad() == None : return False
          else : return self.sad().rechercher(n)
```

Programme complet disponible sur \donnee\NSI\Abr\codes\Abr.py

Test de la méthode rechercher()

Le but de cette partie est de définir un jeu de tests afin de valider la méthode rechercher()

2. **Construire** sur papier un arbre binaire de recherche de hauteur 4. **Programmer** cet arbre avec la classe Abr.




```python
#Construction d'un arbre de test
abr = Abr(20, Abr(15, Abr(7), Abr(19,Abr(16,None,Abr(17)),None)),\
              Abr(40,None,Abr(60,Abr(50),Abr(70))))
```

3. **Définir** sur papier, un jeu de 4 tests à effectuer sur cet arbre de référence (2 vrais et 2 faux) afin de valider la méthode rechercher(). **Tester** et **valider** la méthode rechercher().

abr.rechercher(16) → True
abr.rechercher(70) → True
abr.rechercher(55) → False
abr.rechercher(14) → False

Automatisation du test

Pour valider un programme il est souvent nécessaire de le tester avec un grand jeu de test. Dans ce cas, il est inutile de poursuivre la série de tests dès qu'un test est faux. Sous Python l'instruction assert permet de tester une réponse attendue à un test et renvoyer une erreur si ce test est faux.
Exemple :

```
def indice_max_tableau(t) :
    '''Renvoie l'indice du maximum de tableau t
    t est supposé non vide'''
    assert len(t) >0, 'Le tableau est vide'
```

```
>>> indice_max_tableau([])
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/media/herve/RV DROUGARD/TG_NSI/Cours/7_arbres/code/Abr/Abr.py", line 47,
in indice_max_tableau
    assert len(t) >0, ' Le tableau est vide '
AssertionError:  Le tableau est vide
```

La syntaxe de l'instruction assert est :

assert *test*, *Message d'erreur*

Dans cet exemple, on voit que l'instruction assert a pour effet d'arrêter le programme en levant une erreur de type assert et affiche le message d'erreur souhaité. Comparé à l'instruction raise, assert combine dans une même instruction, un test et une levée d'erreur.

4. **Compléter** sur le fichier abr.py et en respectant la spécification, la fonction test_rechercher_ABR().

```
def test_rechercher_Abr(abr,list_tests, list_resultats) :
    ''' Test automatique de la méthode Abr.rechercher()
    abr : arbre binaire de recherche
    list_tests : liste des valeurs (int) à rechercher
    list_resulats : lsite des resultats (booleens) attendus
    Retour : Pas de retour attendu -> Levée d'erreur par assertion si test faux'''
    for i in range(len(list_tests)):
    assert abr.rechercher(list_tests[i]) == list_resultats[i], 'erreur sur  \
          '+str(list_tests[i])
```

Ajout d'un élément dans un ABR

Le principe est le même que pour la recherche. Un nouveau nœud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arêtée.

5. **Programmer** dans la classe Abr, la méthode inserer(val) qui insère une valeur dans un ABR. **Décrire** sur feuille un test permettant de valider la méthode inserer() à partir de la fonction test_rechercher_ABR(). **Tester** et **valider** la méthode inserer()

Il suffit de créer un arbre binaire avec la méthode insérer() puis vérifier avec test_rechercher_Abr(a,l,l) si ce test arrive à retrouver les valeurs. Si le test ne renvoie pas d'erreur cela signifie que l'ABR est bien construit.

```python
def inserer(self, n :int) :
    '''Insere n dans l'arbre de recherche'''
    if self.noeud() > n :
            if self.sag() == None : self.set_sag(Abr(n))
            else : self.sag().inserer(n)
    else :
            if self.sad() == None : self.set_sad(Abr(n))
            else : self.sad().inserer(n)
```

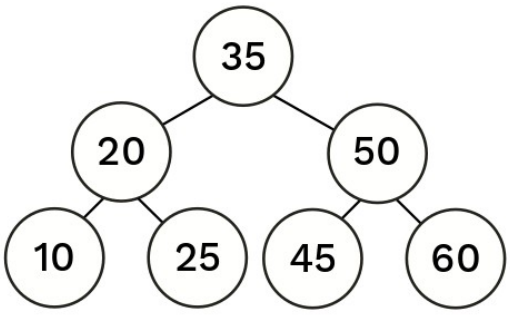
Analyse de performance d'un ABR

Le but de cette partie est d'analyser la performance de l'algorithme de recherche d'une valeur dans un ABR en fonction de la forme de l'arbre.

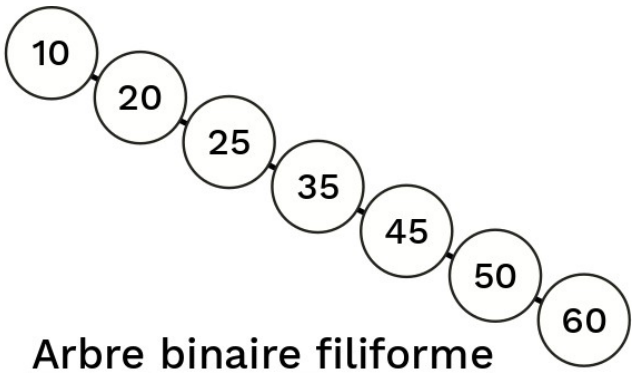
Construisons deux arbres binaires de recherche distincts avec la méthode inserer(). Ces deux arbres contiennent des valeurs identiques mais l'ordre d'appel de la méthode d'insertion d'un nœud diffère. Par exemple, considérons deux arbres contenant les valeurs 10, 20, 25, 35, 45 50, 60. Ces deux arbres sont construits en suivant l'ordre suivant :

Nom de l'arbre	Ordre d'insertion des nœuds dans l'arbre
abr1	10, 20, 25, 35, 45 50, 60
abr2	35, 20, 50, 10, 25, 45, 60

6. **Dessiner** ces deux arbres abr1 et abr2 puis **décrire** leurs différences.



Arbre binaire complet équilibré



Arbre binaire filiforme

7. **Déterminer** l'arbre dont la structure permet de faire une recherche d'élément le plus rapidement. **Indiquer** la complexité de l'algorithme de recherche rechercher() dans ces deux cas.

La recherche sera plus rapide sur l'arbre binaire équilibré. Les complexités dans ces deux cas extrêmes sont :

Abr équilibré: $O(\log_2(n))$

Abr filiforme : $O(n)$

8. **Conclure** en décrivant la forme que doit conserver l'arbre de recherche pour maintenir une recherche efficace ainsi que les moyens d'y parvenir.

Pour améliorer la recherche il faut conserver une forme équilibrée de l'arbre. Ceci peut être réalisé en modifiant les racines des sous arbres au fur et mesure de l'insertion d'un nouvel élément.