|  |  |
| --- | --- |
| Term NSI | AD2 Arbres binaires algorithmes |
| AD :  AD2 Arbres binaires algorithmes [Extrait de pixees](https://pixees.fr/informatiquelycee/n_site/nsi_term.html), Ouvrage Ellipses et Nathan, [vidéo formation ICN](https://www.fun-mooc.fr/courses/inria/41014/session01/about) | |

***Ex 1*** : **Parcours en largeur**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **\*** | **2** | **+** | **3** |  | **+** | **8** | **-** | **5** | **-** |  |  | **4** |  |
| **Niv0** | **Niv1** | | **Niv2** | | | | **Niv3** | | | | | | | |

|  |  |
| --- | --- |
| **Algorithme : *Tous les affichages se feront sur une seule ligne, en python print(……,end='')***  Créer l’arbre avec une liste  **equ** <-  ""  Afficher "le résultat de l'équation : "  Pour chaque élément **i** de la liste :  afficher i toujours sur une même ligne  **equ** <- (**equ** + **i**)  Transformer **equ** en formule **(val(equ))**  Afficher **equ** | **Programme python**  arbre = ["1","\*","2","-","3","","+","8","-","5","-","","","4",""]  a=""  print("le résultat de l'équation : ",end='''' )  for i in arbre :  print(i,end='''')  a=a+i    b=eval(a)  print(" = ",end='''')  print(b) |

***Ex 2*** : **Parcours en largeur**

|  |  |  |
| --- | --- | --- |
| **Compléter les inconnues** | **Exemple 1** | **Exemple 2** |
| **Arbre** |  | **Mettre un \_ pour nœud vide (qui n’existe pas)**  \_  \_  \_  M  E  R  D  I  O  E  Z  T  R  S  N |
| **Représentation liste** | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  | **8** |  |  | **9** | **10** | **11** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  |  |  |  | **12** | **13** |  |  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **N** | **S** | **I** | **\_** | **R** | **O** | **D** | **E** | **Z** | **\_** | **T** | **E** | **R** | **M** | **\_** | |
| **Parcours largeur** | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **N** | **S** | **I** | **\_** | **R** | **O** | **D** | **E** | **Z** | **\_** | **T** | **E** | **R** | **M** | **\_** | |
| **Parcours longueur**  **préfixe** | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **2** | **4** | **8** | **5** | **3** | **6** | **9** | **10** | **12** | **13** | **7** | **11** | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **N** | **S** | **\_** | **E** | **Z** | **R** | **\_** | **T** | **I** | **O** | **E** | **R** | **D** | **M** | **\_** | |
| **Parcours longueur**  **infixe** | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **4** | **8** | **2** | **5** | **1** | **9** | **6** | **12** | **10** | **13** | **3** | **11** | **7** | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **E** | **\_** | **Z** | **S** | **\_** | **R** | **T** | **N** | **E** | **O** | **R** | **I** | **M** | **D** | **\_** | |
| **Parcours longueur**  **postfixe** | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **8** | **4** | **5** | **2** | **9** | **12** | **13** | **10** | **6** | **11** | **7** | **3** | **1** | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **E** | **Z** | **\_** | **\_** | **T** | **R** | **S** | **E** | **R** | **O** | **M** | **\_** | **D** | **I** | **N** | |

Est-il possible de retrouver un arbre dans tous les cas, seulement avec l’un de ces parcours ?Oui**/Non sauf si l’arbre est complet**

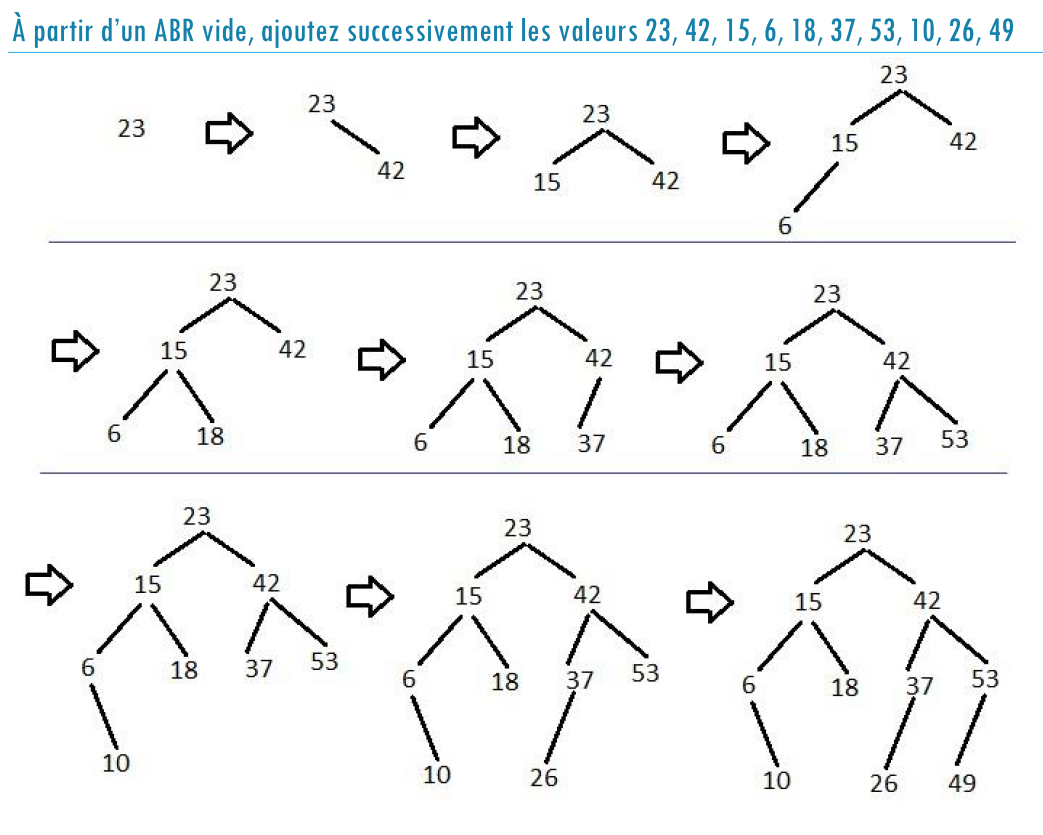
***Ex 3*** : **Arbre binaire de recherche (débranché)**

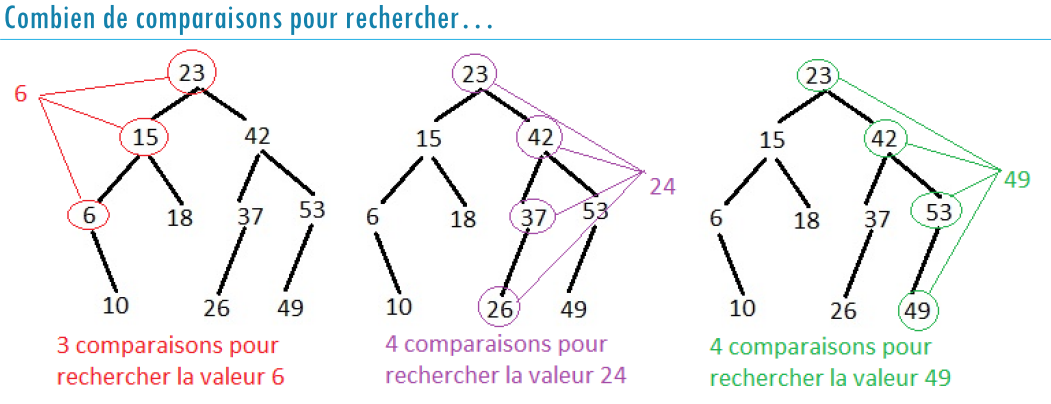
À partir d’un arbre binaire de recherche vide, réaliser les opérations suivantes :

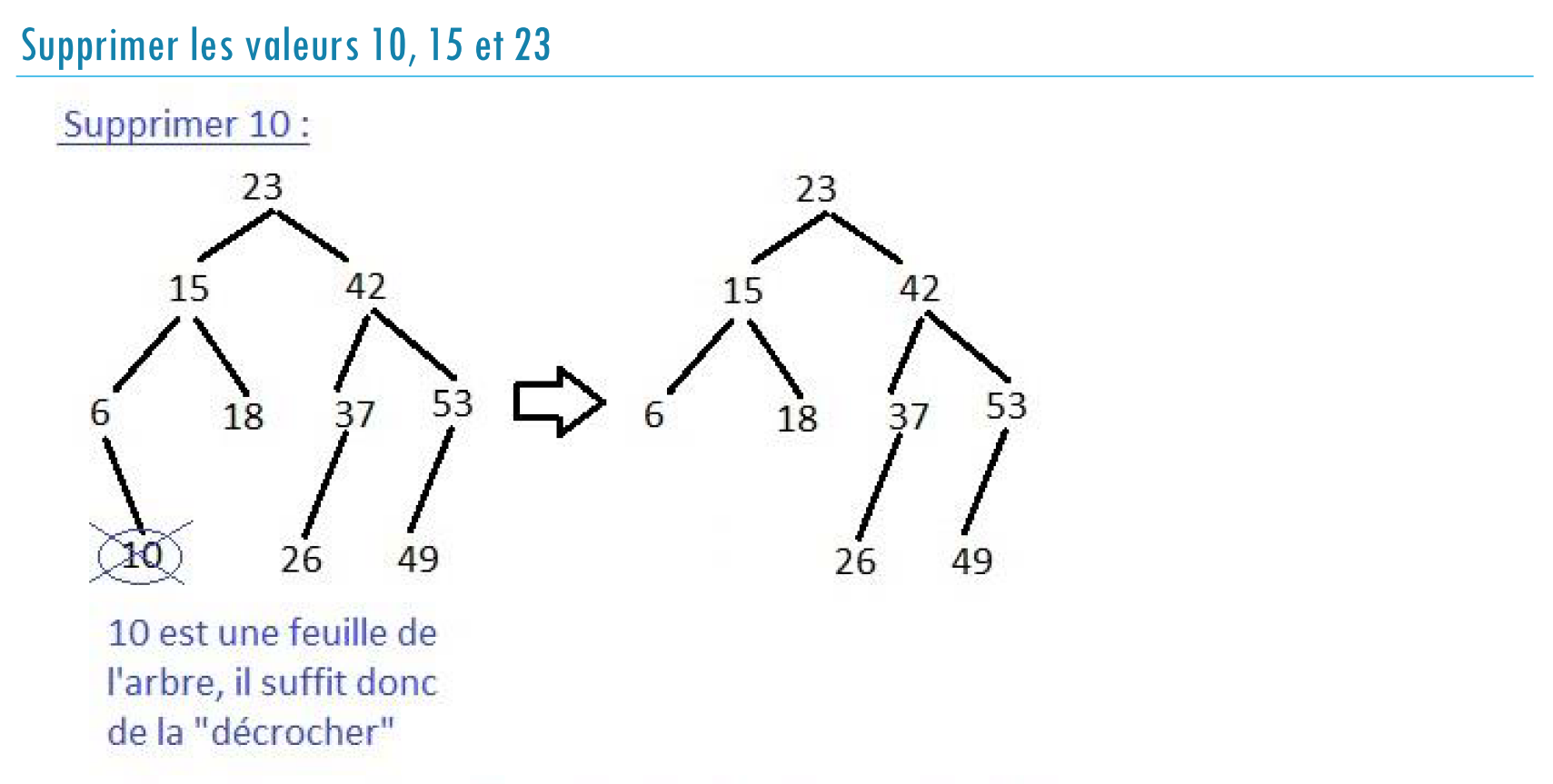
1. Ajouter successivement les valeurs 23, 42, 15, 6, 18, 37, 53, 10, 26, 49

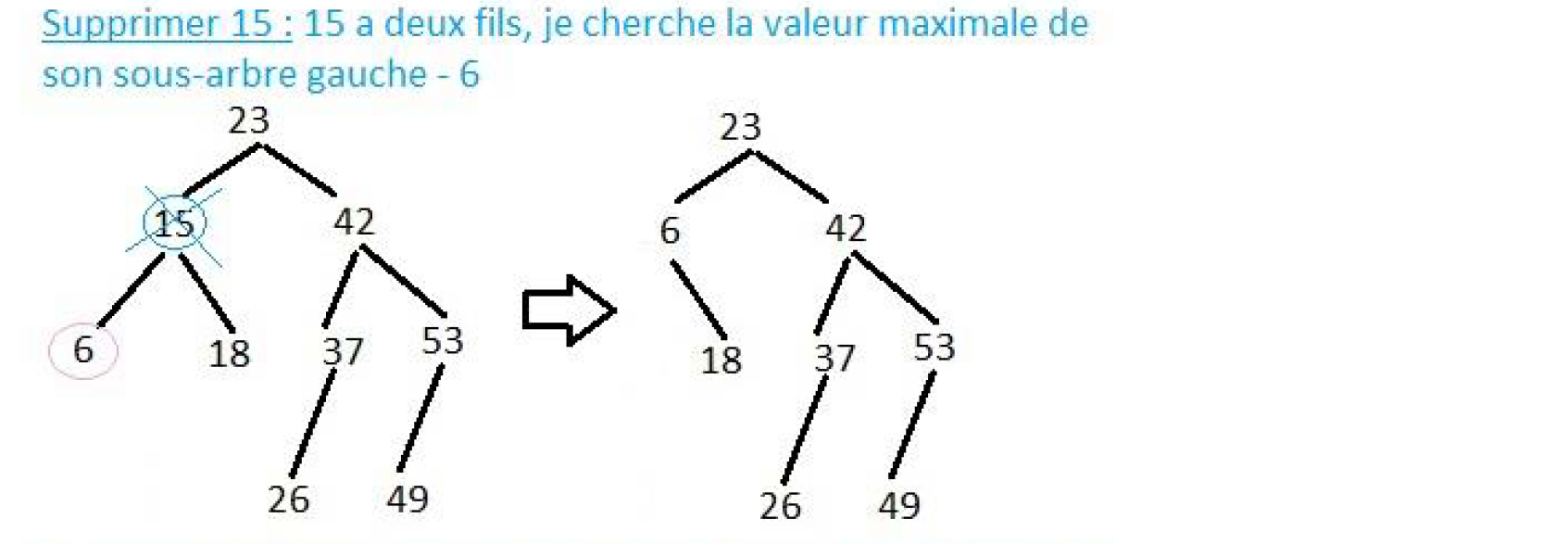
2. Combien de comparaisons effectuez-vous pour rechercher successivement les valeurs 6, 24 et 49 dans cet arbre ?

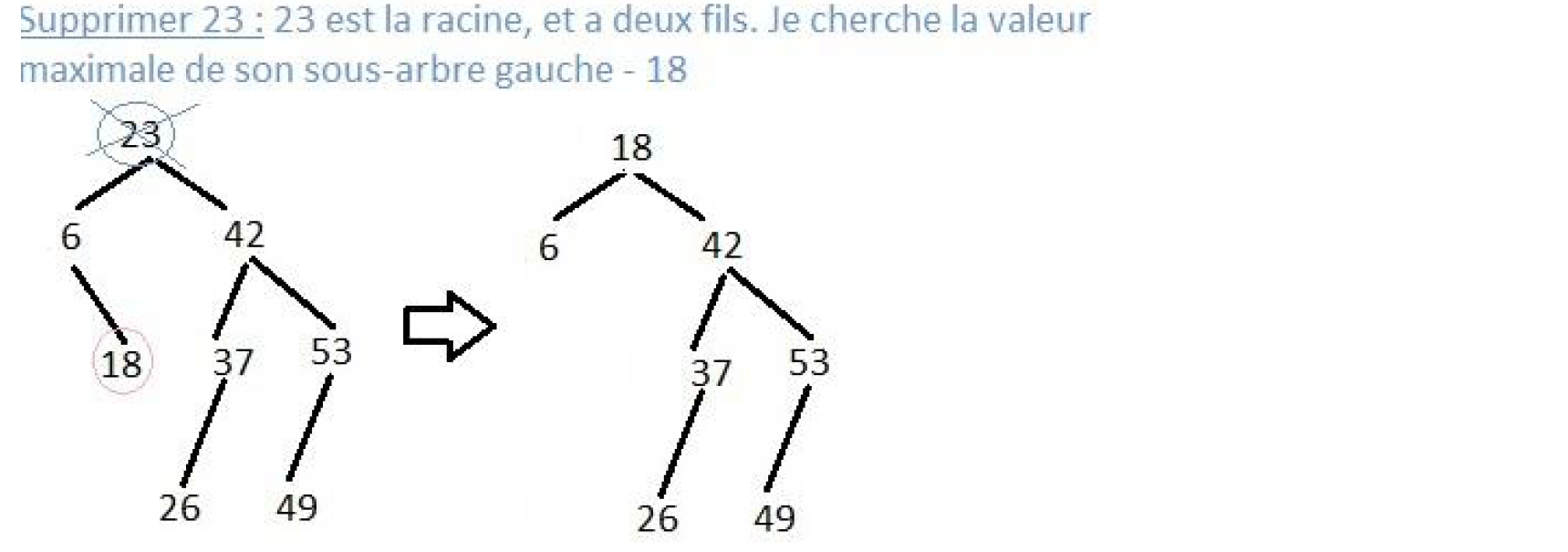
3. Supprimer les valeurs 10,15 et 23.











***Ex 4*** : **Compléter la classe Arbre**

L’objectif est d’implémenter la classe Arbre pour décrire les arbres binaires de recherche. Compléter la classe Arbre

Récupérer le ﬁchier ***SujetABR.py*** .

1. Analyser les fonctions suivantes de la classe arbre ***class Arbre***:

|  |  |
| --- | --- |
| **Méthode** | **Explication** |
| def \_\_init\_\_(self):  self.\_racine = None  self.\_gauche = None  self.\_droit = None | Création de l’objet b = Arbre()   |  |  | | --- | --- | | **Racine de b** | **none** | | **sag** | **none** | | **sad** | **none** | |
| # estVide : Arbre[E] → Booléen  def estVide(self):  return self.\_racine is None | Tester  b=Arbre()  print(b.estVide())   retourne true si vide , false si non vide |
| def racine(self):  assert not self.estVide(), "l'arbre est vide"  return self.\_racine | print(b.racine())  si la racine de b est vide il y a un message de test (AssertionError) l’arbre est vide sinon retourne la valeur de la racine |
| def sag(self): ou sad  assert not self.estVide(), "l'arbre est vide"  return self.\_gauche ou droit | print(b.sad()) si le sous arbre gauche ou droit de b est vide il y a un message de test (AssertionError) l’arbre est vide sinon retourne la valeur du sous arbre gauche ou droit |

2. Compléter la fonction **ajouter(self,x),** qui ajoute l’élément x dans sous arbre droit.   
  
**algorithme méthode de la classe arbre**

|  |  |
| --- | --- |
| **ajouter(self,x) :**  si racine est vide   racine = x  sinon  si x=racine  fin  sinon si x < racine  si sag est vide  sag = Arbre(vide)   sag = valeur  sinon   ajouter(sag,x)  sinon  si sad est vide  sad = Arbre(vide)   sad = valeur  sinon   ajouter(sad,x) | def ajouter(self, valeur):  if self.estVide():  self.\_racine = valeur  else:  if valeur == self.\_racine:  return  elif valeur < self.\_racine:  if self.\_gauche is None:  self.\_gauche = Arbre()  self.\_gauche.\_racine = valeur  else:  self.\_gauche.ajouter(valeur)  else:  if self.\_droit is None:  self.\_droit = Arbre()  self.\_droit.\_racine = valeur  else:  self.\_droit.ajouter(valeur) |

3. Compléter **def construireArbre():**

|  |  |
| --- | --- |
| **Algorithme** | **Méthode** |
| Créer arbre vide a  ajouter le nœud racine15 puis 12,20,20,8,10,14,13,16,18,17,19,21 | def construireArbre():  a = Arbre()  a.ajouter(15)  a.ajouter(12)  a.ajouter(20)  a.ajouter(20)  a.ajouter(8)  a.ajouter(10)  a.ajouter(14)  a.ajouter(13)  a.ajouter(16)  a.ajouter(18)  a.ajouter(17)  a.ajouter(19)  a.ajouter(21)  return a |

Tester avec les commandes   
 a = Arbre.construireArbre()

a.\_draw() , \_draw ne fonctionnera que si la bibliothèque matplot a été importée sinon utiliser :

a.dump()

|  |  |
| --- | --- |
|  |  |

4. Analyser la méthode **parcoursPre**

|  |  |
| --- | --- |
| **Méthode** | **Explication** |
| # parcours préfixé de l'arbre a (ou parcours RGD)  def parcoursPre(self):  if self.estVide():  return []  else:  return self.\_parcoursPre([])  def \_parcoursPre(self, l):  l.append(self.\_racine)  if self.\_gauche:  self.\_gauche.\_parcoursPre(l)  if self.\_droit:  self.\_droit.\_parcoursPre(l)  return l | Test avec 15. parcoursPre()  1 parcoursPre( 15 ):  Si 15 est vide fin ( False )  Sinon renvoyer 15.\_parcoursPre(liste vide)  2 \_parcoursPre( 15 , liste vide )  l=[ 15 ]  Si sag= 12 n’est pas vide  12. \_parcoursPre(l=[ 15 ])  ***Attente : Si sad=\_\_\_ n’est pas vide \_\_\_. \_parcoursPre(l=[\_\_\_]***  ***Attente : Return l=[\_\_\_\_\_\_\_]***  3 \_parcoursPre( 12 , l=[ 15 ])  l=[ 15 , 12 ]  Si sag= 8 n’est pas vide  8. \_parcoursPre(l=[15 , 12])  ***Attente : Si sad=\_\_\_ n’est pas vide \_\_\_. \_parcoursPre(l=[\_\_\_]***  ***Attente : Return l=[\_\_\_\_\_\_\_]***  4 \_parcoursPre( 8 , l=[ 15 , 12 ])  l=[15 , 12 , 8 ]  Si sag=None est vide  Si sad= 10 n’est pas vide  10. \_parcoursPre(l=[ 15 , 12 , 8 ])  ***Attente : Return l=[\_\_\_\_\_\_\_]***  5 \_parcoursPre( 10 , l=[ 15 , 12 , 8])  l=[15 , 12 , 8 , 10 ]  Si sag= None est vide  Si sad= None est vide  Return l=[15 , 12 , 8 , 10] à 4 \_parcoursPre( 8 , l=[15,12 ,8,10])  6(ex\_4) \_parcoursPre(8, l=[ 15 ,12 ,8 ,10 ])  Return l=[15 ,12 ,8 ,10] à 3 \_parcoursPre( 12, l=[15 ,12 ,8 ,10]  7(ex\_3) \_parcoursPre( 12, l=[15 ,12 ,8 ,10]) Si sad= 12 n’est pas vide 14. \_parcoursPre([15 ,12 ,8 ,10])  ***Attente : Return l=[\_\_\_\_\_\_\_]***  8. \_parcoursPre( 14, l=[15 ,12 ,8 ,10])  l=[15 ,12 ,8 ,10 , 14]  Si sag= 13 n’est pas vide 13. \_parcoursPre(l=[15 ,12 ,8 ,10 , 14]  ***Attente :* Si sad=\_\_\_\_\_est vide**  ***Attente : Return l=[\_\_\_\_\_\_\_]***  Nous pourrions ainsi parcourir toute la liste. |

Afficher le parcours préfixe  
print(a. parcoursPre()) l= [ 15 , 12 , 8 , 10 , 14 , 13 , 20 , 16 , 18 , 17 , 19 , 21]  
 

5- Compléter la méthode **parcoursInf**

|  |  |
| --- | --- |
| **Algorithme** | **Méthode** |
| appelRécursif sur sag  + [noeud]   + appelRécursif sur sad | def parcoursInf(self):  if self.estVide():  return []  else:  return self.\_parcoursInf([])  def \_parcoursInf(self, l):  if self.\_gauche:  self.\_gauche.\_parcoursInf(l)  l.append(self.\_racine)  if self.\_droit:  self.\_droit.\_parcoursInf(l)  return l |

Afficher le parcours infixe  
print(a. parcoursInf()) l=[8, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]



6- Compléter la méthode **parcoursPost**

|  |  |
| --- | --- |
| **Algorithme** | **Méthode** |
| appelRécursif sur sag  + appelRécursif sur sad  + [noeud] | def parcoursPost(self):  if self.estVide():  return []  else:  return self.\_parcoursPost([])  def \_parcoursPost(self, l):  if self.\_gauche:  self.\_gauche.\_parcoursPost(l)  if self.\_droit:  self.\_droit.\_parcoursPost(l)  l.append(self.\_racine)  return l |

Afficher le parcours postfixe  
print(a. parcoursPost()) l= [10, 8, 13, 14, 12, 17, 19, 18, 16, 21, 20, 15]



6- Compléter la méthode ***rechercher(self,x)*** qui renvoie Vrai si x est présent dans l’arbre, Faux sinon

|  |  |
| --- | --- |
| **Algorithme** | **Méthode** |
| ***Recherche(A,x) :*** Si l’arbre est vide Renvoyer "l'arbre est vide" Si x < racine(A)   Si nonVide(SAG)  Recherche\_récursive(sag(A),x)  Sinon   Renvoyer faux  Si x > racine(A)   Si nonVide(SAD)   Recherche\_récursive (sad(A),x)  Sinon   Renvoyer faux  Renvoyer x == racine | def rechercher(self, valeur):  if self.estVide():  return "l'arbre est vide"  if valeur < self.\_racine :  if self.\_gauche:  return self.\_gauche.rechercher(valeur)  else :  return False  if valeur > self.\_racine :  if self.\_droit:  return self.\_droit.rechercher(valeur)  else :  return False  return valeur == self.\_racine |

Vérifier en recherchant par exemple 15,19,23

|  |  |
| --- | --- |
| print(a.rechercher(15))  print(a.rechercher(19))  print(a.rechercher(23)) | True  True  False |

7- Analyse la méthode ***hauteurNoeud(self,x)*** qui renvoie la hauteur de la donnée x dans l’arbre

|  |
| --- |
| **Méthode** |
| 1-def hauteurNoeud(self, valeur=13):  2-if self.estVide():  3- assert False, "la valeur n'existe pas dans l'arbre "  4-return self.\_hauteurNoeud(valeur=13)  5-def \_hauteurNoeud(self, valeur):  6-if valeur < self.\_racine:  7-if self.\_gauche is None:  8- assert False, "l'arbre est vide"  8-else:  9-return 1 + self.\_gauche.\_hauteurNoeud(valeur)  10-elif valeur > self.\_racine:  11-if self.\_droit is None:  12- assert False, "la valeur n'existe pas dans l'arbre "  13-else:  14-return 1 + self.\_droit.\_hauteurNoeud(valeur)  15-else:  16-return 0 |
| **Analyse : hauteurNoeud x=13** |
| 1- (Nœud=15,x=13)  4- Return \_Haut(Nœud=15,x=13)= 3  5- (No\_15,x=13) 6- x=13< Nœud=15 8- Return \_Haut=1+\_Haut(SAG(15)=12,x=13) = 1 + 2  5- (Nœud=12,x=13)  10- x=13>Nœud=12 13- Return \_Haut=1+\_Haut(SAD(12)=14,x=13) = 1 + 1    5- (Nœud=14,x=13)  6- x=13<Nœud=12 8- Return \_Haut=1+\_Haut(SAG(14)=13,x=13) = 1 + 0  5- (Nœud=13,x=13)  15- Return \_Haut= 0  Test : print(a.hauteurNoeud(13)) résultat = 3 |
| **Analyse : hauteurNoeud x=22** |
| 1- (Nœud=15,x= 22 )  4- Return \_Haut(Nœud=15,x= 22 )=\_\_\_\_\_\_\_\_  5- (No\_15,x= 22  10- x=22 > Nœud=15 13- Return \_Haut=1+\_Haut(SAD(15)=20 ,x=22) = \_\_\_\_\_\_\_\_\_  5- (Nœud= 20 , x=22)  10- x= 22 >Nœud= 20 13- Return \_Haut=1+\_Haut(SAD( 20 )= 21 ,x= 22 ) =\_\_\_\_\_\_\_\_    5- (Nœud= 21 , x=22)  10- x= 22 >Nœud = 21  11- self.\_droit is None: 12- assert False, "la valeur n'existe pas dans l'arbre "  Arrêt du programme  Test : print(a.hauteurNoeud(22))   résultat = Arrêt du programme AssertionError: la valeur n'existe pas dans l'arbre |
| Conclusion : il faudrait modifier def HauteurNoeud pour vérifier si la valeur existe avec ***def existe***  ***Attention on balaye 2 fois l’arbre complexité 2 \* ln***   |  |  | | --- | --- | | def hauteurNoeud(self, valeur):  if self.estVide():  assert False, "l'arbre est vide"  return self.\_hauteurNoeud(valeur) | def hauteurNoeud(self, valeur):  if self.existe(valeur):  print("la valeur existe")  return self.\_hauteurNoeud(valeur)  else:  return "la valeur n'existe pas" | |

***Autre possibilité ne pas utiliser la récursivité***

8- Analyser la méthode ***\_hauteur(self)*** qui retourne la hauteur de l’arbre, c’est à dire, la longueur du plus long chemin entre la racine et une feuille

|  |  |
| --- | --- |
| **Algorithme** | **Analyse** |
| 1-def hauteur(self):  2-if self.estVide():  3-return 0  4-else:  5-return self.\_hauteur()  6-def \_hauteur(self):  7-if self.\_gauche is None:  8-hg = 0  9-else:  6-\_haut( 8 )  8-hg = 0  14- hd= 1+\_haut( 10 )=1 + 0  ***Suite à faire***  15-return max( 0, 1)= 1  10-hg = 1 + self.\_gauche.\_hauteur()  11-if self.\_droit is None:  12-hd = 0  13-else:  14-hd = 1 + self.\_droit.\_hauteur()  15-return max(hg, hd)  6-\_haut( 10 )  8-hd =0  12-hd =0  15-return max( 0, 0)= 0 | 6-\_haut( 12 )  10-hg= 1+\_haut( 8 )= 1+1  ***Suite à faire***  14- hd= 1+\_haut( 14 )= 1+1  ***Suite à faire***  15-return max( 2, 2)= 2  1-Nœud= 15  2-return \_haut( 15 )  6-\_haut( 15 )  10-hg= 1+\_haut( 12 )= 1+2  ***Suite à faire…….***    6-\_haut( 13 )  8-hd =0  12-hd =0  15-return max( 0, 0)= 0  6-\_haut( 14 )  10-hg= 1+\_haut( 13 )= 1 + 0  ***Suite à faire***  12-hd =0  15-return max( 1, 0)= 1 |

Test : print(a.hauteur ())   
  
résultat = 4

9- Analyser la méthode ***maximum(self)*** et ***minimum(self)*** qui détermine la donnée maximale (respectivement minimale) de l’arbre

|  |  |
| --- | --- |
| **Algorithme** | **Analyse** |
| # minimum : Arbre[E] → Entier  def minimum(self):  if self.estVide():  assert False, "l'arbre est vide"  while self.\_gauche:  self = self.\_gauche  return self.\_racine  # maximum : Arbre[E] → Entier  def maximum(self):  if self.estVide():  assert False, "l'arbre est vide"  while self.\_droit:  self = self.\_droit  return self.\_racine | **Le minimum est la valeur du nœud le plus à gauche**  **SAG-SAG-SAG…..jusqu’à ce que le nœud soit une feuille (while SAG n’est pas vide)**  **Le maximum est la valeur du nœud le plus à gauche**  **SAD-SAD-SAD ….. jusqu’à ce que le nœud soit une feuille (while SAD n’est pas vide)** |