Chapitre 2

Algorithmes sur les arbres binaires

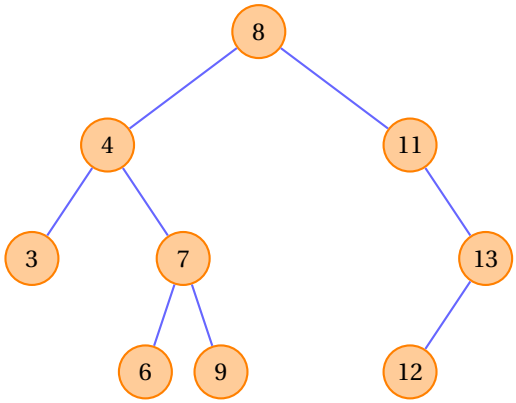
Le seul arbre non binaire de ce chapitre

2.1 Représentation d'un arbre binaire

Dans le cours sur les arbres, nous avons découvert plusieurs manières de représenter un arbre binaire : sous forme d'une liste (représentation contigüe), d'un tableau ou encore de façon abstraite. Il est temps de transposer toutes ces représentations en Python.

2.1.1 Sous forme d'une liste

La représentation contigüe apparaît comme la plus simple en apparence : on énumère les valeurs (ou étiquettes) de notre arbre dans une liste, et les fils non existants ou « fantômes » sont signalés par des `None`.



L'arbre ci-contre est représenté par la liste :

```
[8, 4, 11, 3, 7, None, 13, None, None, 6, 9,
↪  None, None, 12, None]
```

La seule difficulté de cette représentation réside dans détermination des indices des fils gauche et droit d'un noeud d'indice donné.

Par exemple, le noeud 7, d'indice 4 dans la liste, possède deux fils, situés aux indices 9 et 10.

Question 01 Pour un arbre de hauteur 4, quelle est la taille de la liste associée ?

Question 02 Même question pour un arbre de hauteur h quelconque.

Question 03 Quel est l'indice du noeud 4 dans l'exemple ci-dessus ? Quels sont les indices de ses noeuds fils ?

Question 04 Quel est l'indice du noeud 13 dans l'exemple ci-dessus ? Quels sont les indices de ses noeuds fils ?

Question 05 Soit L une liste représentant un arbre de façon contigüe. Soit N un noeud de cet arbre.
Si N se trouve à l'indice i dans la liste L, à quels indices se trouvent ses fils gauche et droit ?

☞ *Dans les questions suivantes, on supposera que les valeurs de l'arbre binaire considéré sont uniques*

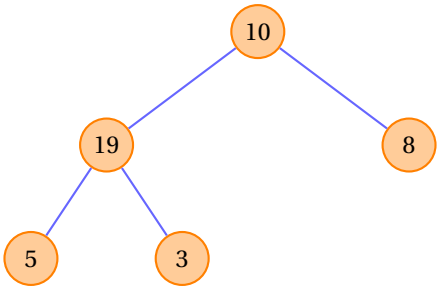
Question 06 Écrire une fonction `valeursFils(liste, valeur)` qui retourne la valeur des fils gauche et droit du noeud d'étiquette `valeur`, sous la forme d'une liste `[fils_gauche, fils_droit]`.

Question 07 Écrire une fonction `estFeuille(liste, valeur)` qui retourne `True` si le noeud d'étiquette `valeur` est une feuille de l'arbre binaire représenté de façon contigüe par `liste`.

Question 08 Écrire une fonction `parent(liste, valeur)` qui retourne la valeur du parent du noeud d'étiquette `valeur`.

2.1.2 Sous forme d'un dictionnaire

La représentation d'un arbre binaire sous forme de dictionnaire, plus abstraite, fait apparaître une certaine récursivité. Elle mélange la représentation d'un arbre sous forme de tableau et la représentation abstraite. Considérons l'exemple suivant :



Sous forme de tableau, l'arbre ci-dessus s'écrit :

Noeud	Étiquette	Noeud du SAG	Noeud du SAD
A	10	B	C
B	19	D	E
C	8		
D	5		
E	3		

On représente alors chaque noeud sous la forme d'un dictionnaire.
En notant respectivement les fils gauche et droit `fg` et `fd`, on a par exemple :

```
A = {"val": 10, "fg": {}, "fd": {}}
```

Les fils de A étant B et C, on les définit ensuite et on les précise comme fils de A :

```
B = {"val": 19, "fg": {}, "fd": {}}
C = {"val": 8, "fg": {}, "fd": {}}

A['fg'] = B
A['fd'] = C
```

On peut également définir directement le noeud A avec ses deux fils, mais on perd en lisibilité :

```
A = {"val": 10, "fg": {"val": 19, "fg": {}, "fd": {}}, "fd": {"val": 8, "fg": {}, "fd": {}}}
```

Au final, voici comment représenter l'arbre précédent :

```
# Création des noeuds

A = {"val": 10, "fg": {}, "fd": {}}
B = {"val": 19, "fg": {}, "fd": {}}
C = {"val": 8, "fg": {}, "fd": {}}
D = {"val": 5, "fg": {}, "fd": {}}
E = {"val": 3, "fg": {}, "fd": {}}

# Affiliations

A['fg'] = B
A['fd'] = C
B['fg'] = D
B['fd'] = E
```

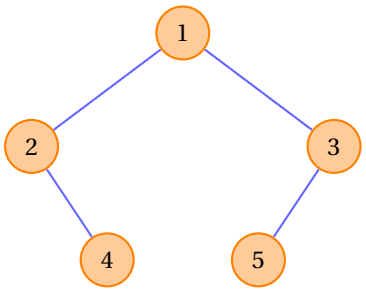
Une autre façon de faire est de commencer par les feuilles, et de « remonter » :

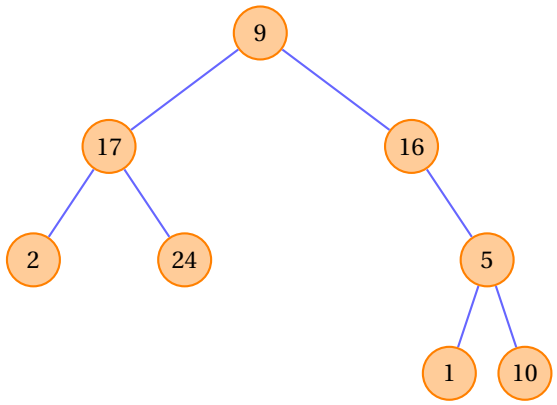
```
C = {"val": 8, "fg": {}, "fd": {}}
D = {"val": 5, "fg": {}, "fd": {}}
E = {"val": 3, "fg": {}, "fd": {}}
B = {"val": 19, "fg": D, "fd": E}
A = {"val": 10, "fg": B, "fd": C}
```

Dans un cas comme dans l'autre, la variable A correspond à notre arbre tout entier - A étant la racine de l'arbre :

```
>>> A
{'val': 10, 'fg': {'val': 19, 'fg': {'val': 5, 'fg': {}, 'fd': {}}, 'fd': {'val': 3, 'fg': {}, 'fd':
 ↪  {}}, 'fd': {'val': 8, 'fg': {}, 'fd': {}}}
```

Question 09 Représenter les arbres suivants à l'aide d'un dictionnaire :





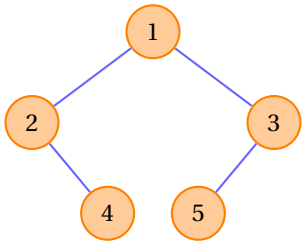
2.1.3 Sous forme d'objet

La représentation sous forme d'objet est de loin la plus abstraite mais également la plus adapatée.
Voici une classe `Arbre` permettant de représenter... un arbre !

☞ *Afin de ne pas rendre l'exercice trop difficile, on ignore ici l'encapsulation*

```
class Arbre:

    def __init__(self, valeur): # On définit un arbre en précisant la valeur de sa racine
        self.valeur = valeur
        self.SAG = None              # SAG : Sous-Arbre Gauche
        self.SAD = None              # SAD : Sous-Arbre Droit
```

Reprenons cet arbre, vu précédemment :

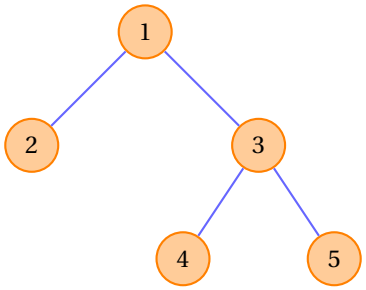


Avec le type `Arbre` défini plus haut, on peut le représenter de plusieurs façons :

```
# En définissant les noeuds un à un

A = Arbre(1)
B = Arbre(2)
C = Arbre(3)
D = Arbre(4)
E = Arbre(5)

A.SAG = B
A.SAD = C
B.SAD = D
C.SAG = E
```

```
# Relativement à la racine

A = Arbre(1)
A.SAG = Arbre(2)
A.SAD = Arbre(3)
A.SAG.SAD = Arbre(4)
A.SAD.SAG = Arbre(5)
```

Question 10 Représenter l'arbre suivant en objet :



Telle quelle, la représentation sous forme d'objet ressemble beaucoup à la représentation sous forme de dictionnaire. Mais en objet, on peut définir des méthodes, permettant d'analyser et de modifier les attributs (ou propriétés) de notre objet. C'est là que l'objet prend tout son sens. Par exemple, pour savoir si un noeud est une feuille, on peut définir une méthode idoine :

```
def estFeuille(self):
    if self.SAG == None and self.SAD == None:
        return True
    else:
        return False
```

Arbre vide

Un **arbre vide** est un arbre qui n'est constitué d'aucun noeud. C'est le cas des fils d'une feuille : on peut dire qu'une feuille ne dispose d'aucun enfant, ou alors que les enfants d'une feuille sont des arbres vides. Il est ainsi préférable de définir un autre type d'objet, `ArbreVide`, permettant de représenter ce cas de figure.
Au final, on obtient les deux classes `Arbre` et `ArbreVide` suivantes, qui sont loin d'être complètes :

```
class Arbre:

    def __init__(self, valeur):
        self.valeur = valeur
        self.SAG = ArbreVide()  # Par défaut, les sous-arbres sont
        self.SAD = ArbreVide()  # des objets de type ArbreVide

    def estFeuille(self):
        if self.SAG.estVide() and self.SAD.estVide(): # Attention au changement ici !
            return True
        else:
            return False

    def estVide(self):
        return False

class ArbreVide:

    def __init__(self):
        self.valeur = None

    def estVide(self):
        return True
```

Dans la foulée, on a défini une méthode `estVide` pour chacun des deux objets : pour un « vrai » arbre, la méthode renvoie `False`, et dans le cas d'un arbre vide, elle renvoie `True`.

▷ ***Quel est l'intérêt de créer un objet avec un seul attribut égal à `None` ? Ça paraît compliqué !***

Cette façon de faire semble complexe, mais nous verrons dans la suite que cela résoud certains problèmes et rend le code plus lisible.

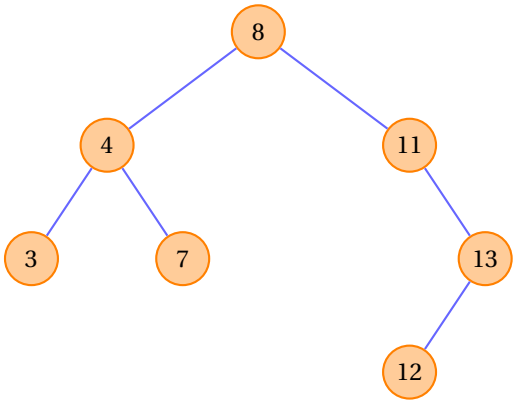
2.2 Taille et hauteur d'un arbre binaire

La **taille** d'un arbre est le nombre de noeuds le composant. Un algorithme permettant de déterminer la taille d'un arbre est le suivant (donné ici en langage naturel) :

```
T : arbre binaire

TAILLE(T):
  Si T est un arbre vide:
    Retourner 0
  Sinon:
    Retourner 1 + TAILLE(T.sous_arbre_gauche) + TAILLE(T.sous_arbre_droit)
```

Question 11 Cet algorithme est-il récursif ? Justifier.

Question 12 Appliquer cet algorithme sur l'arbre ci-contre afin de saisir son fonctionnement :



La **hauteur** d'un arbre est la profondeur maximale des noeuds qui le composent. Un algorithme permettant de déterminer la hauteur d'un arbre est le suivant (donné ici en langage naturel) :

```
T : arbre binaire

HAUTEUR(T):
  Si T est un arbre vide:
    Retourner 0
  Sinon:
    Retourner 1 + max(HAUTEUR(T.sous_arbre_gauche), HAUTEUR(T.sous_arbre_droit))
```

Question 13 Reprendre les deux questions précédentes.

Question 14 Implémenter les méthodes `taille` et `hauteur` dans la classe `Arbre` définie précédemment.

☞ *On pourra également implémenter ces méthodes dans la classe* `ArbreVide` ...

2.3 Parcours d'un arbre binaire

Parcourir un arbre, c'est lister les noeuds de ce dernier dans un certain ordre. On peut voir l'action de parcourir un arbre comme une fonction qui, à un arbre donné, associe une liste de ses noeuds. On distingue deux types de parcours :

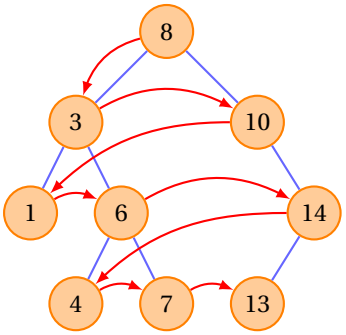
- le **parcours en largeur** (de gauche à droite)
- le **parcours en profondeur** (de haut en bas), dans lequel on distingue à nouveau trois parcours :
  - le parcours préfixe
  - le parcours infixe
  - le parcours postfixe

2.3.1 Parcours en largeur

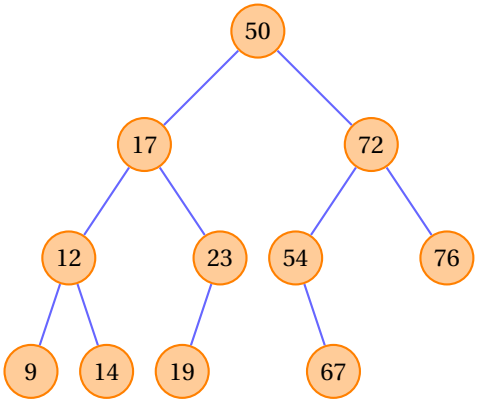
Le **parcours en largeur** consiste à parcourir l'arbre niveau par niveau. Les noeuds de niveau 0 sont d'abord parcourus, puis les noeuds de niveau 1 et ainsi de suite. Dans chaque niveau, les noeuds sont parcourus de la gauche vers la droite.

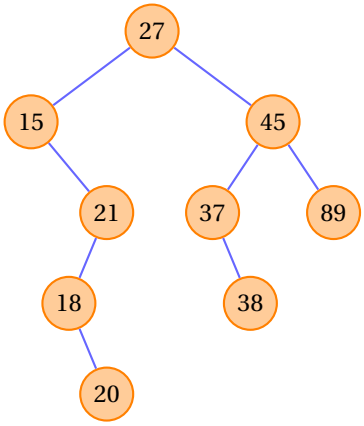
Prenons l'arbre ci-contre. Son parcours en largeur donne la liste :

```
[8, 3, 10, 1, 6, 14, 4, 7, 13]
```



Question 15 Décrire le parcours en largeur des arbres binaires suivants :



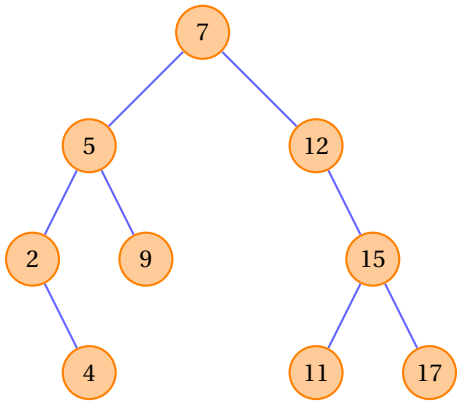


Le parcours en largeur se programme à l'aide d'une **file**, que nous avons déjà découvert dans un précédent chapitre :

```
T : arbre binaire

PARCOURS_LARGEUR(T):
  L = Liste vide

  SI T n'est pas vide:
    F = File vide
    F.enfiler(T)

    Tant que F n'est pas vide:
      arbre_courant = F.defiler()
      Ajouter la racine de arbre_courant à L

      Si arbre_courant.sous_arbre_gauche n'est pas vide:
        F.enfiler(arbre_courant.sous_arbre_gauche)

      Si arbre_courant.sous_arbre_droit n'est pas vide:
        F.enfiler(arbre_courant.sous_arbre_droit)

  Retourner L
```

Question 16 Implémenter la méthode `parcours_largeur` dans la classe `Arbre` précédente.

Question 17 Représenter l'arbre suivant en objet, et vérifier l'implémentation de la méthode `parcours_largeur` .



2.3.2 Parcours en profondeur

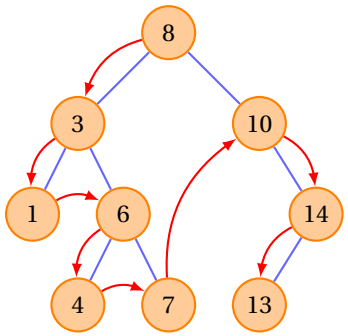
Parcours préfixe : Racine - SAG - SAD

Dans le **parcours préfixe**, la racine est traitée avant les appels récursifs sur les sous-arbres gauches et droit (faits dans cet ordre). L'algorithme est le suivant :

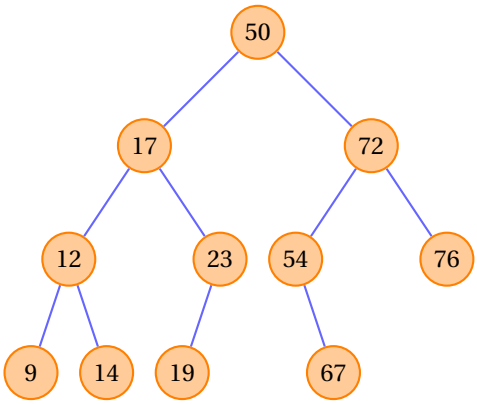
```
T : arbre binaire
L : liste (initialement vide)

PARCOURS_PREFIXE(T, L):
  Si T est non vide:
    Ajouter la racine de T à la liste L
    PARCOURS_PREFIXE(T.sous_arbre_gauche, L)
    PARCOURS_PREFIXE(T.sous_arbre_droit, L)
```

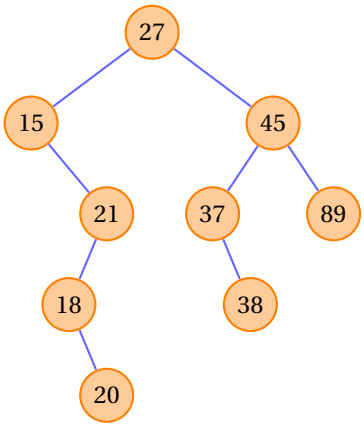
Prenons l'arbre ci-contre. Son parcours préfixe donne la liste :

```
[8, 3, 1, 6, 4, 7, 10, 14, 13]
```



Question 18 Décrire le parcours préfixe des arbres binaires suivants :





Question 19 Implémenter la méthode `parcours_prefixe` et vérifier son fonctionnement sur les exemples précédents.

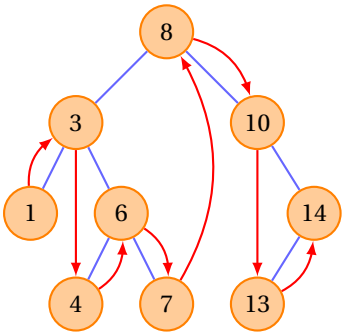
Parcours infixe : SAG - Racine - SAD

Dans le **parcours infixe**, le traitement de la racine se fait entre les appels sur les sous-arbres gauche et droit. L'algorithme est le suivant :

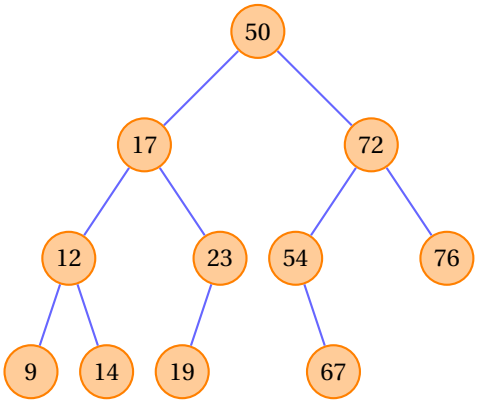
```
T : arbre binaire
L : liste (initialement vide)

PARCOURS_INFIXE(T, L):
  Si T est non vide:
    PARCOURS_INFIXE(T.sous_arbre_gauche, L)
    Ajouter la racine de T à la liste L
    PARCOURS_INFIXE(T.sous_arbre_droit, L)
```

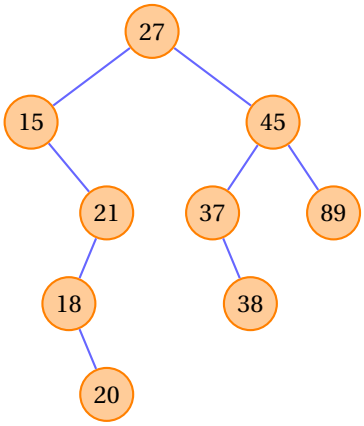
Prenons l'arbre ci-contre. Son parcours infixe donne la liste :

```
[1, 3, 4, 6, 7, 8, 10, 13, 14]
```



Question 20 Décrire le parcours infixe des arbres binaires suivants :





Question 21 Implémenter la méthode `parcours_infixe` et vérifier son fonctionnement sur les exemples précédents.

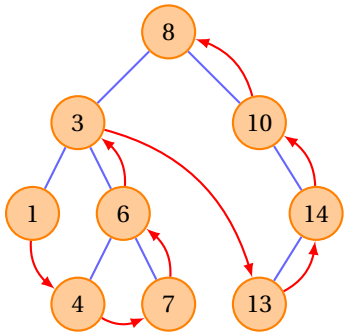
Parcours postfixe : SAG - SAD - Racine

Dans le **parcours postfixe**, la racine est traitée après les appels récursifs sur les sous-arbres gauche et droit (dans cet ordre). L'algorithme est le suivant :

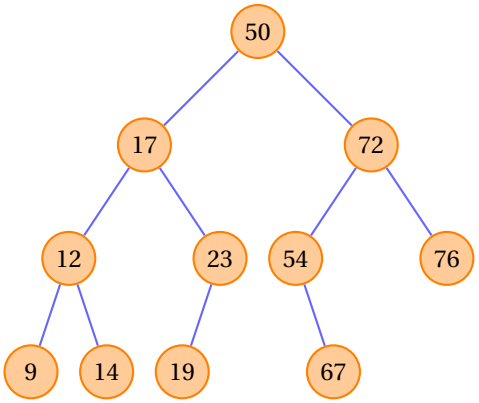
```
T : arbre binaire
L : liste (initialement vide)

PARCOURS_POSTFIXE(T, L):
  Si T est non vide:
    PARCOURS_POSTFIXE(T.sous_arbre_gauche, L)
    PARCOURS_POSTFIXE(T.sous_arbre_droit, L)
    Ajouter la racine de T à la liste L
```

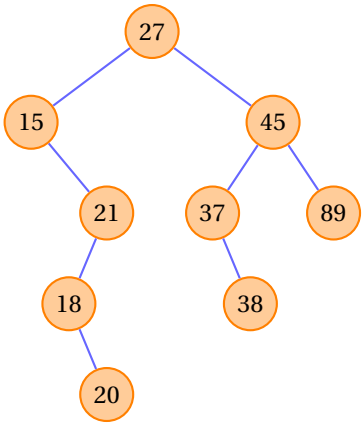
Prenons l'arbre ci-contre. Son parcours infixe donne la liste :

```
[1, 3, 4, 6, 7, 8, 10, 13, 14]
```



Question 22 Décrire le parcours postfixe des arbres binaires suivants :

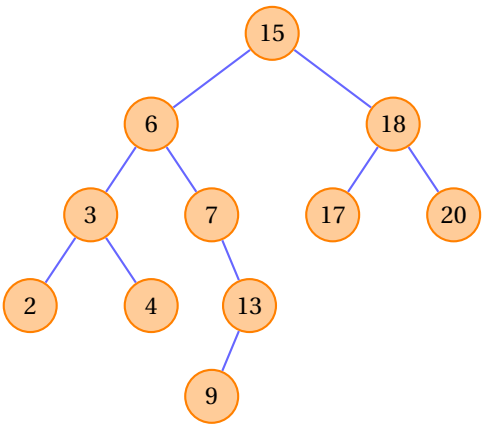




Question 23 Implémenter la méthode `parcours_postfixe` et vérifier son fonctionnement sur les exemples précédents.

2.4 Avec des arbres binaires de recherche

Un arbre binaire de recherche est un arbre dans lequel chaque clé d'un noeud est supérieure ou égale à la clé de son fils gauche, et strictement inférieure à la clé de son fils droit. Un arbre binaire de recherche est intéressant puisqu'il est toujours possible de connaître dans quelque branche de l'arbre se trouve un élément et, de proche en proche, le localiser dans l'arbre. Considérons l'arbre binaire de recherche suivant :



Question 24 Représenter cet arbre sous la forme d'un objet de type `Arbre`.

Question 25 Implémenter la méthode `dessiner` suivante, qui permet de représenter graphiquement un arbre à l'aide du module `turtle` (qu'il ne faut pas oublier d'importer !).

Question 26 Dessiner votre arbre !

```
1  def dessiner(self, t = None, x = 0, y = 300, d = 0, debut = True):
2      if t == None:
3          t = turtle.Turtle()
4          t.up()
5          t.ht()
6          d = 10 * 2**(self.hauteur())
7
8      if not self.SAG.estVide():
9          t.goto(x,y)
10         t.down()
11         t.goto(x-d, y-40)
12         t.up()
13         self.SAG.dessiner(t, x-d, y - 40, d // 2, False)
14     if not self.SAD.estVide():
15         t.goto(x,y)
16         t.down()
17         t.goto(x+d, y-40)
18         t.up()
19         self.SAD.dessiner(t, x+d, y - 40, d // 2, False)
20
21     t.goto(x,y)
22     t.dot(30, "black")
23     t.dot(25, "white")
24     t.goto(x+1,y-7.5)
25     t.write(self.valeur, align="center", font=("Verdana", 10, "normal"))
26
27     if debut:
28         turtle.done()
```

☞ *Ne pas oublier l'instruction* `import turtle`, *placée tout en haut de votre fichier*

2.4.1 Recherche dans un ABR

Pour chercher une valeur dans un arbre binaire de recherche, on commence par la comparer avec la racine. Si la valeur cherchée est égale à la clé de la racine, on retourne `True`. Si la valeur est strictement supérieure à la clé de la racine, on recherche alors cette valeur dans le sous-arbre droit. Sinon, on la cherche dans le sous-arbre gauche. Si, arrivé à une feuille, la valeur n'est pas trouvée, l'algorithme se termine et retourne `False`.

```
T : arbre binaire de recherche
cle : valeur cherchée

RECHERCHER(T, cle):
  Si T est vide:
    Retourner Faux
  Sinon:
    Si cle = T.valeur:
      Retourner Vrai
    Sinon si cle > T.valeur:
      Retourner RECHERCHER(T.SAD, cle)
    Sinon:
      Retourner RECHERCHER(T.SAG, cle)
```

Question 24 Quelle est la taille de l'ABR précédent ? Au maximum, combien de comparaisons faut-il effectuer pour savoir si une valeur se trouve dans l'arbre ?

Question 25 Au maximum, combien de comparaisons faut-il effectuer pour chercher une valeur dans une liste contenant les mêmes valeurs (non triée) ?s

Question 26 Ajouter une méthode `rechercher(self, valeur)` à la classe `Arbre` implémentant cet algorithme.

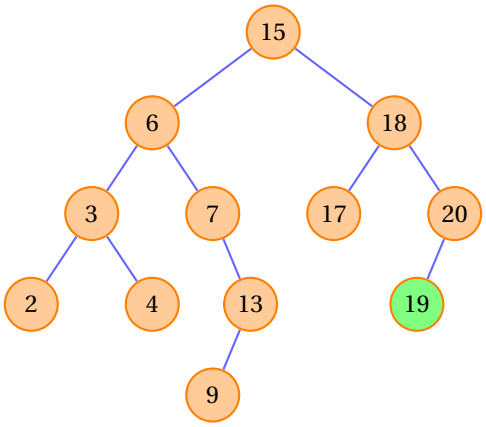
Remarque. Cet algorithme de recherche ressemble à la recherche dichotomique dans une liste triée vue en classe de Première. L'avantage de ces méthodes est la compléxité en temps dans le pire des cas. Elle est de $O(n)$ pour une liste non triée (on parle de **compléxité linéaire** : doubler la taille de la liste double le nombre d'opérations effectuées par le programme, et donc le temps d'exécution) alors qu'elle est de $O(\log_2 n)$ pour un arbre binaire de recherche (on parle de **compléxité logarithmique** : doubler la taille de la liste ajoute seulement une opération !).

2.4.2 Insertion dans un ABR

Pour insérer une clé dans un ABR, encore faut-il savoir à quel endroit la placer !
Premièrement, un nouveau noeud sera toujours inséré dans une feuille. On commence à chercher la valeur à insérer jusqu'à atteindre une feuille. Une fois la feuille trouvée, on compare la valeur à ajouter à la clé de la feuille : si la valeur est supérieure stricte à la clé, on crée un enfant à droite de valeur égale à la valeur à ajouter, sinon on crée un enfant à gauche.

```
T : arbre binaire de recherche
cle : valeur à ajouter

INSERER(T, cle):
  Si T est une feuille:
    Si cle <= T.valeur:
      T.SAG = Arbre(cle)
    Sinon:
      T.SAD = Arbre(cle)
  Sinon:
    Si cle <= T.valeur:
      T.SAG.INSERER(cle)
    Sinon:
      T.SAD.INSERER(cle)
```

En appliquant cet algorithme, la valeur 19 serait placée de cette manière dans l'ABR précédent :



Question 27 Avec cet algorithme, où serait insérée la valeur 11 dans l'ABR précédent ?

Question 28 Avec cet algorithme, où serait insérée une valeur déjà présente dans l'arbre ?

Question 29 Ajouter une méthode `inserer(self, valeur)` à la classe `Arbre` implémentant cet algorithme.

Question 30 Écrire une fonction `creerABR(liste)` qui transforme une liste en un arbre binaire de recherche.