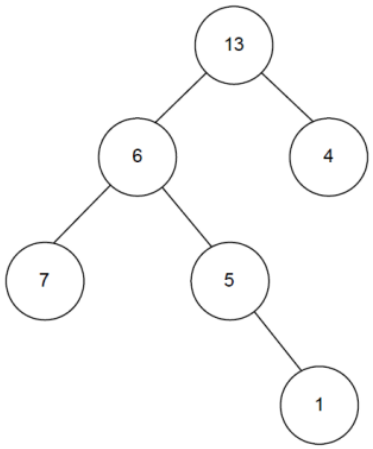
Table des matières

1 Polynésie J2 - 2023

Exercice 1

Cet exercice porte sur les arbres binaires, les arbres binaires de recherche, la programmation orientée objet et la récursivité.

Q1. On considère l'arbre ci-dessous :



(a) Justifier que cet arbre est un arbre binaire.

(b) Indiquer si l'arbre ci-dessus est un arbre binaire de recherche (ABR). Justifier votre réponse.

Q2. On considère la classe `Noeud`, nous permettant de définir les arbres binaires, définie de la manière suivante en `python` :

```
class Noeud:
    def __init__(self, g, v, d):
        """crée un noeud d'un arbre binaire"""
        self.gauche = g
        self.valeur = v
        self.droit = d
```

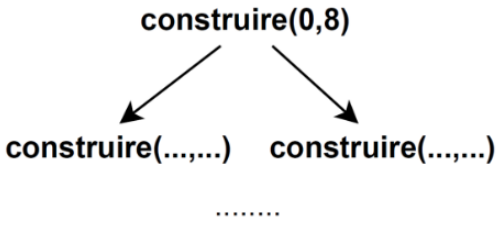
On considère également la fonction `construire` ci-dessus qui prend en paramètre deux entiers `mini` et `maxi` et qui renvoie un arbre.

```
1  def construire(mini, maxi):
2      """mini, maxi: entiers respectant mini <= maxi"""
3      assert isinstance(mini, int) and isinstance(..., ...) and ...
4      if maxi - mini == 1 or maxi - mini == 0:
5          return Noeud(None, mini, None)
6      elif maxi - mini == 2:
7          return Noeud(None, (mini+maxi)//2, None)
8      else:
9          sag = construire(mini, (mini+maxi)//2)
10         sad = construire((mini+maxi)//2, maxi)
11         return Noeud(sag, (mini+maxi)//2, sad)
```

La fonction `isinstance(obj, t)` renvoie `True` si l'objet `obj` est du type `t`, sinon `False`.

(a) Recopier et compléter sur votre copie la ligne 3 de l'assertion de la fonction `construire` de manière à vérifier les conditions sur les paramètres `mini` et `maxi`.

(b) On exécute l'instruction `construire(0, 8)`. Déterminer quels sont les différents appels récursifs de la fonction `construire` exécutés. On représentera ces appels sous forme d'une arborescence, par exemple :



(c) Dessiner l'arbre renvoyé par l'instruction `construire(0, 8)`.

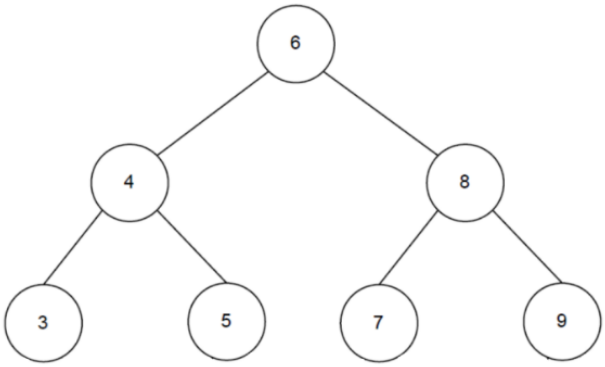
(d) Dessiner l'arbre renvoyé par l'instruction `construire(0, 3)`.

(e) Donner le résultat d'un parcours infixe sur l'arbre obtenu à la question **Q2.(c)**.
Expliquer pourquoi ce parcours permet d'affirmer que l'arbre est un arbre binaire de recherche.

(f) La fonction récursive `maximum` ci-dessous prend en paramètre un arbre binaire de recherche `abr` et renvoie la valeur maximale de ses nœuds. Recopier et compléter les lignes 5 et 7 de cette fonction.

```
1  def maximum(abr):
2      if abr is None:
3          return None
4      elif abr.droit is None:
5          return ...
6      else :
7          return ...
```

Q3. On donne l'arbre binaire de recherche `abr_7_noeuds` suivant :



On donne également ci-dessous la fonction `mystere` qui prend en paramètres un arbre binaire de recherche `abr`, un entier `x` et une liste `liste`.

```
def mystere(abr, x, liste):
    """
    abr -- arbre binaire de recherche
    x -- int
    liste -- list
    Renvoie -- list"""
    if abr is None:
        return []
    else:
        liste.append(abr.valeur)
        if x == abr.valeur:
            return liste
        elif x < abr.valeur:
            return mystere(abr.gauche, x, liste)
        else:
            return mystere(abr.droit, x, liste)
```

(a) Donner les résultats obtenus lorsque l'on exécute les instructions : `mystere(abr_7_noeuds,5,[])`, puis `mystere(abr_7_noeuds,6,[])` puis `mystere(abr_7_noeuds,2,[])`.

(b) Décrire quel peut être le rôle de la fonction `mystere`.

2 France J2 - 2023

Exercice 2

Cet exercice porte sur les arbres binaires, les files et la programmation orientée objet.
Cet exercice comporte deux parties indépendantes.

PARTIE 1

Une entreprise stocke les identifiants de ses clients dans un arbre binaire de recherche.

On rappelle qu'un arbre binaire est composé de nœuds, chacun des nœuds possédant éventuellement un sous-arbre gauche et éventuellement un sous-arbre droit.

La taille d'un arbre est le nombre de nœuds qu'il contient. Sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles. On convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1 et celle de l'arbre vide vaut 0.

Dans cet arbre binaire de recherche, chaque nœud contient une valeur, ici une chaîne de caractères, qui est, avec l'ordre lexicographique (celui du dictionnaire) :

- ▷ strictement supérieure à toutes les valeurs des nœuds du sous-arbre gauche ;
- ▷ strictement inférieure à toutes les valeurs des nœuds du sous-arbre droit.

Ainsi les valeurs de cet arbre sont toutes distinctes.

On considère l'arbre binaire de recherche suivant :

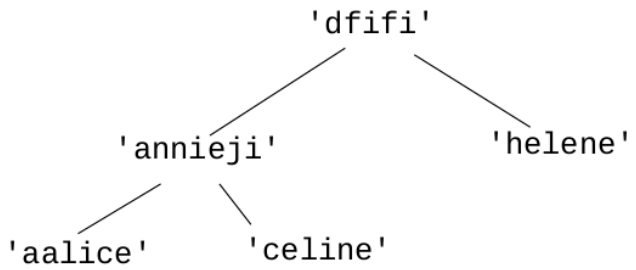


Figure 1. Arbre binaire de recherche.

Q1. Donner la taille et la hauteur de l'arbre binaire de recherche de la figure 1.

Q2. Recopier cet arbre après l'ajout des identifiants suivants : 'davidbg' et 'papicoeur' dans cet ordre.

Q3. On décide de parcourir cet arbre pour obtenir la liste des identifiants dans l'ordre lexicographique.

Recopier la lettre correspondant au parcours à utiliser parmi les propositions suivantes :

A - Parcours en largeur d'abord
B - Parcours en profondeur dans l'ordre préfixe
C - Parcours en profondeur dans l'ordre infixe
D - Parcours en profondeur dans l'ordre suffixe (ou postfixe)

Q4. Pour traiter informatiquement les arbres binaires, nous allons utiliser une classe `ABR`.

Un arbre binaire de recherche, nommé `abr` dispose des méthodes suivantes :

- ▷ `abr.est_vide()` : renvoie `True` si `abr` est vide et `False` sinon.
- ▷ `abr.racine()` : renvoie l'élément situé à la racine de `abr` si `abr` n'est pas vide et renvoie `None` sinon.
- ▷ `abr.sg()` : renvoie le sous-arbre gauche de `abr` s'il existe et renvoie `None` sinon.
- ▷ `abr.sd()` : renvoie le sous-arbre droit de `abr` s'il existe et `None` sinon.

On a commencé à écrire une méthode récursive present de la classe ABR, où le paramètre identifiant est une chaîne de caractères et qui retourne True si identifiant est dans l'arbre et False sinon.

```
1    def present(self, identifiant):
2        if self.est_vide():
3            return False
4        elif self.racine() == identifiant:
5            return ...
6        elif self.racine() < identifiant:
7            return self.sd(). ...
8        else:
9            return ...
```

Recopier et compléter les lignes 5, 7 et 9 de cette méthode.

PARTIE 2

On considère une structure de données file que l'on représentera par des éléments en ligne, l'élément à droite étant la tête de la file et l'élément à gauche étant la queue de la file.

On appellera f1 la file suivante :

	'bac'	'nsi'	'2023'	'file'	

On suppose que les quatre fonctions suivantes ont été programmées préalablement en langage python :

- ▷ `creer_file()` : renvoie une file vide ;
- ▷ `est_vide(f)` : renvoie `True` si la file `f` est vide et `False` sinon ;
- ▷ `enfiler(f, e)` : ajoute l'élément `e` à la queue de la file `f` ;
- ▷ `defiler(f)` : renvoie l'élément situé à la tête de la file `f` et le retire de la file.

Q5. Les trois questions suivantes sont indépendantes.

(a) Donner le résultat renvoyé après l'appel de la fonction `est_vide(f1)`.

(b) Représenter la file `f1` après l'exécution du code `defiler(f1)`.

(c) Représenter la file `f2` après l'exécution du code suivant :

```
1 f2 = creer_file()
2 liste = ['castor', 'python', 'poule']
3 for elt in liste :
4     enfiler(f2, elt)
```

Q6. Recopier et compléter les lignes 4, 6 et 7 de la fonction `longueur` qui prend en paramètre une file `f` et qui renvoie le nombre d'éléments qu'elle contient.
Après un appel à la fonction, la file `f` doit retrouver son état d'origine.

```
1  def longueur(f):
2      resultat = 0
3      g = creer_file()
4      while ... :
5          elt = defiler(f)
6          resultat = ...
7          enfiler(... , ...)
8      while not(est_vide(g)):
9          enfiler(f, defiler(g))
10     return resultat
```

Q7. Un site impose à ses clients des critères sur leur mot de passe.
Pour cela il utilise la fonction `est_valide` qui prend en paramètre une chaîne de caractères `mot` et qui retourne `True` si mot correspond aux critères et `False` sinon.

```
def est_valide(mot):
    if len(mot) < 8:
        return False
    for c in mot:
        if c in ['!', '#', '@', ';', ':']:
            return True
    return False
```

Parmi les mots de passe suivants, recopier celui qui sera validé par cette fonction.

A - 'best@' **B -** 'paptap23' **C -** '2!@59fgds'

Q8. Le tableau suivant montre, sur deux exemples, l'évolution d'une file `f3` après l'exécution de l'instruction `ajouter_mot(f3, 'super')` :

	état initial de `f3`	état de `f3` après l'instruction `ajouter_mot(f3, 'super')`
Exemple 1	`'bac'` `'nsi'` `'2023'`	`'super'` `'bac'` `'nsi'`
Exemple 2	`'test'` `'info'`	`'super'` `'test'` `'info'`

Écrire le code de cette fonction `ajouter_mot` qui prend en paramètres une file `f` (qui a au plus 3 éléments) et une chaîne de caractères valide `mdp`. Cette fonction met à jour la file de stockage `f` des mots de passe en y ajoutant `mdp` et en défilant, si nécessaire, pour avoir au maximum trois éléments dans cette file.

On pourra utiliser la fonction `longueur` de la question **Q6.**

Q9. Pour intensifier sa sécurité, le site stocke les trois derniers mots de passe dans une file et interdit au client lorsqu'il change son mot de passe d'utiliser l'un des mots de passe stockés dans cette file.

Recopier et compléter les lignes 7 et 8 de la fonction `mot_file` :

- ▷ qui prend en paramètres une file `f` et `mdp` de type chaîne de caractères ;
- ▷ qui renvoie `True` si le mot de passe est un élément de la file `f` et `False` sinon.

Après un appel à cette fonction, la file `f` doit retrouver son état d'origine.

```
def mot_file(f, mdp):
    g = creer_file()
    present = False
    while not(est_vide(f)):
        elt = defiler(f)
        enfiler(g, elt)
        if ...:
            present = ...
    while not(est_vide(g)):
        enfiler(f, defiler(g))
    return present
```

Q10. Écrire une fonction modification qui prend en paramètres une file `f` et une chaîne de caractères `nv_mdp`. Si le mot de passe `nv_mdp` répond bien aux **deux** exigences des questions **Q7.** et **Q9.**, alors elle modifie la file des mots de passe stockés et renvoie `True`. Dans le cas contraire, elle renvoie `False`.
On pourra utiliser les fonctions `mot_file`, `est_valide` et `ajouter_mot`.

3 Liban J2 - 2023

Exercice 3

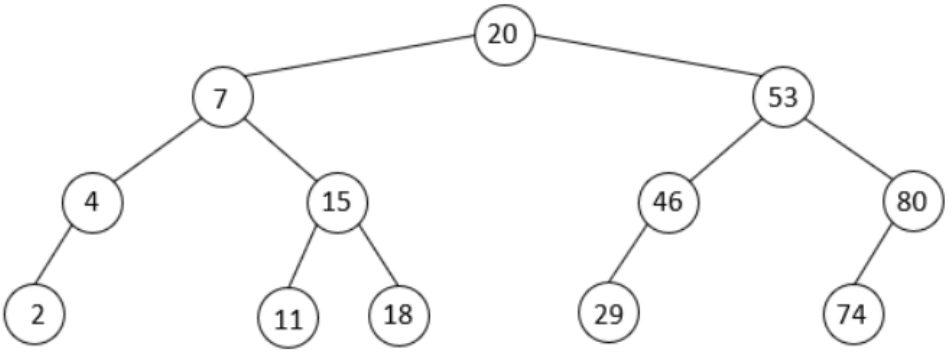
Cet exercice traite des arbres et de l'algorithmique.

Dans cet exercice, la taille d'un arbre est égale au nombre de ses nœuds et on convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1.

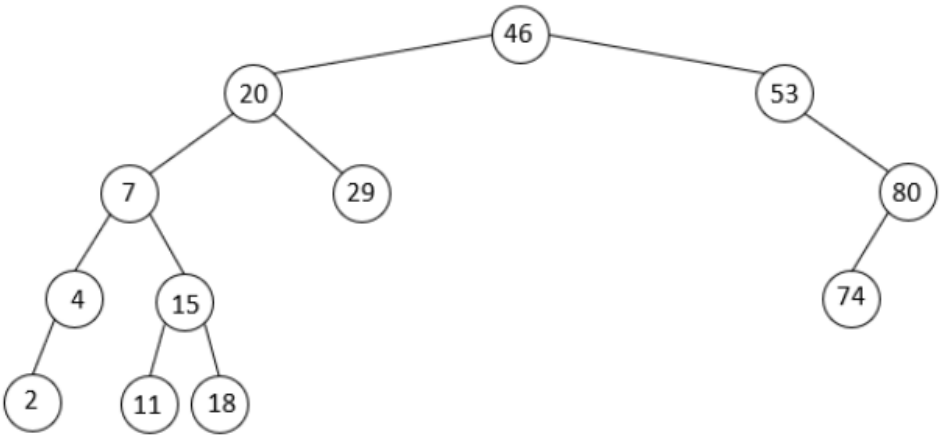
On utilisera la définition suivante : un arbre binaire de recherche est un arbre binaire, dans lequel

- ▷ on peut comparer les valeurs des nœuds : ce sont par exemple des nombres entiers, ou des lettres de l'alphabet ;
- ▷ si x est un nœud de cet arbre et y est un nœud du sous-arbre gauche de x, alors il faut que `y.valeur < x.valeur`
- ▷ si x est un nœud de cet arbre et y est un nœud du sous-arbre droit de x, alors il faut que `y.valeur ⩾ x.valeur`

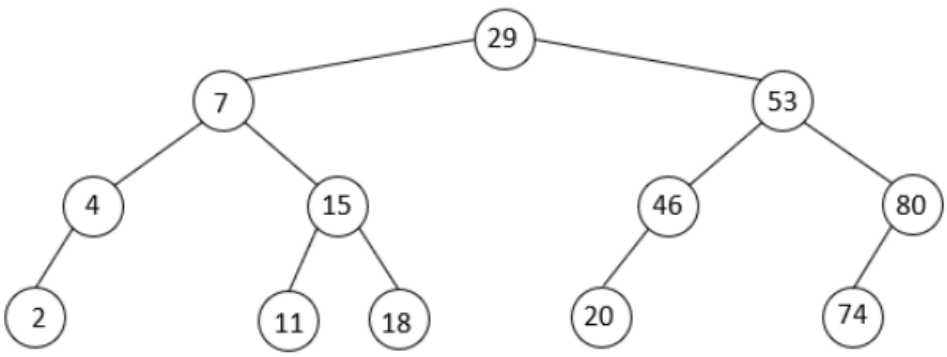
Q1 Parmi les trois arbres dessinés ci-dessous, recopier sur la copie le numéro correspondant à celui qui n'est pas un arbre binaire de recherche. Justifier.



Arbre 1



Arbre 2



Arbre 3

Une classe ABR, qui implémente une structure d'arbre binaire de recherche, possède l'interface suivante :

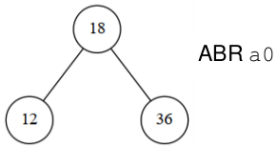
```
1  class ABR:
2      def __init__(self, valeur, sa_gauche, sa_droit):
3          self.valeur = valeur # valeur de la racine
4          self.sa_gauche = sa_gauche # sous-arbre gauche
5          self.sa_droit = sa_droit # sous-arbre droit
6
7      def inserer_noued(self, valeur):
8          """Renvoie un nouvel ABR avec le noeud de valeur 'valeur'
9          inséré comme nouvelle feuille à sa position correcte"""
10         # code non étudié dans cet exercice
```

On prendra la valeur `None` pour représenter un sous-arbre vide.

Q2. La construction d'un ABR se fait en insérant progressivement les valeurs à partir de la racine : la méthode `inserer_noeud` (dont le code n'est pas étudié dans cet exercice) place ainsi un nœud à sa « bonne place » comme feuille dans la structure, sans modifier le reste de la structure. On admet que la position de cette feuille est unique.

(a) En utilisant les méthodes de la classe ABR :

- ▷ écrire l'instruction python qui permet d'instancier un objet a0, de type ABR, ayant un seul nœud (la racine) de valeur 18.
- ▷ écrire une séquence d'instructions qui permet ensuite d'insérer dans l'objet a0 les deux feuilles de l'arbre de valeurs 12 et 36.

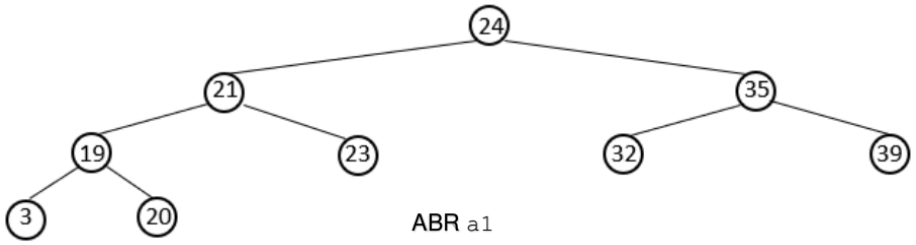


ABR a0

Selon l'ordre dans lequel les valeurs sont insérées, on construit des ABR ayant des structures différentes.

Voilà par exemple ci-dessous un ABR (nommé a1) obtenu en créant une instance de type ABR ayant un seul nœud (la racine) de valeur 24 puis en insérant successivement les valeurs dans l'ordre suivant :

21 ; 35 ; 19 ; 23 ; 32 ; 39 ; 3 ; 20



ABR a1

(b) Dessiner sur la copie l'ABR (nommé a2) que l'on obtiendrait en créant une instance de type ABR ayant un seul nœud (la racine) de valeur 3 puis en insérant successivement les valeurs dans l'ordre suivant :

20 ; 19 ; 21 ; 23 ; 32 ; 24 ; 35 ; 39

(c) Donner la hauteur des ABR a1 et a2.

(d) On complète la classe ABR avec une méthode `calculer_hauteur` qui renvoie la hauteur de l'arbre.
Recopier sur la copie les lignes 10 et 13 en les complétant par des commentaires et la ligne 14 en la complétant par une instruction dans le code ci-après de cette méthode.

On pourra utiliser la fonction python `max` qui prend en paramètres deux nombres et renvoie le maximun de ces deux nombres.

```
    def calculer_hauteur(self):
        """Renvoie la hauteur de l'arbre"""
        if self.sa_droit is None and self.sa_gauche is None:
        # l'arbre est réduit à une feuille
            return 1
        elif self.sa_droit is None:
        # arbre avec une racine et seulement un sous-arbre gauche
            return 1 + self.sa_gauche.calculer_hauteur()
        elif self.sa_gauche is None:
        # à compléter
            return 1 + self.sa_droit.calculer_hauteur()
        else:
        # à compléter
            return # à compléter

```

Q3. La différence de hauteur entre l'ABR `a1` et l'ABR `a2` aura des conséquences lors de la recherche d'une valeur dans l'ABR.

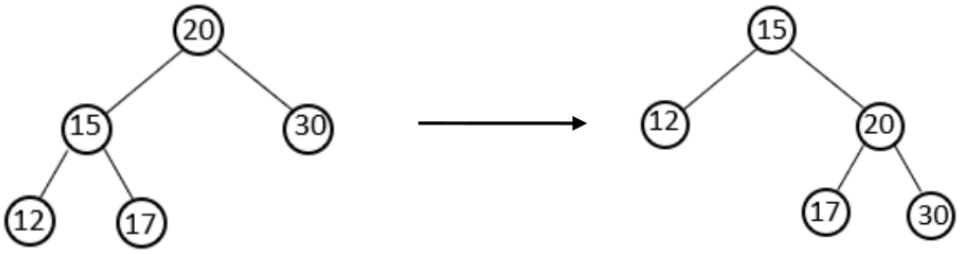
(a) Recopier et compléter sur la copie les lignes 6, 8, 11 et 13 du code ci-dessous de la méthode `rechercher_valeur`, qui permet de tester la présence ou l'absence d'une valeur donnée dans l'ABR :

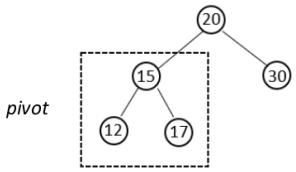
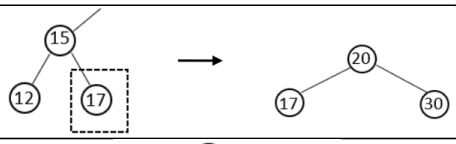
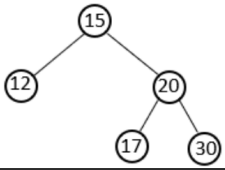
```
    def rechercher_valeur(self, v):
        """
        Renvoie True si la valeur v est trouvée dans l'ABR,
        False sinon
        """
        if # à compléter
            return True
        elif  #à compléter and self.sa_gauche is not None:
            return self.sa_gauche.rechercher_valeur(v)
        elif v > self.valeur and self.sa_droit is not None:
            return # à compléter
        else:
            return # à compléter
```

(b) On admet que le nombre de fois où la méthode `rechercher_valeur` est appelée pour rechercher la valeur 39 dans l'ABR `a2` est 7.
Donner le nombre de fois où la méthode `rechercher_valeur` est appelée pour rechercher la valeur 20 dans l'ABR `a1`.

Q4. Il existe des algorithmes pour modifier la structure d'un ABR , afin par exemple de diminuer la hauteur d'un ABR ; on s'intéresse aux algorithmes appelés rotation, consistant à faire « pivoter » une partie de l'arbre autour d'un de ses nœuds.

L'exemple ci-dessous permet d'expliquer l'algorithme pour réaliser une rotation droite d'un ABR autour de sa racine :



On appelle *pivot* le sous-arbre gauche de la racine de l'arbre	
Le sous-arbre droit du pivot devient le sous-arbre gauche de la racine	
La racine ainsi modifiée devient le sous-arbre droit du pivot et la racine du pivot devient la nouvelle racine de l'ABR	

On admet que ces transformations conservent la propriété d'ABR de l'arbre.

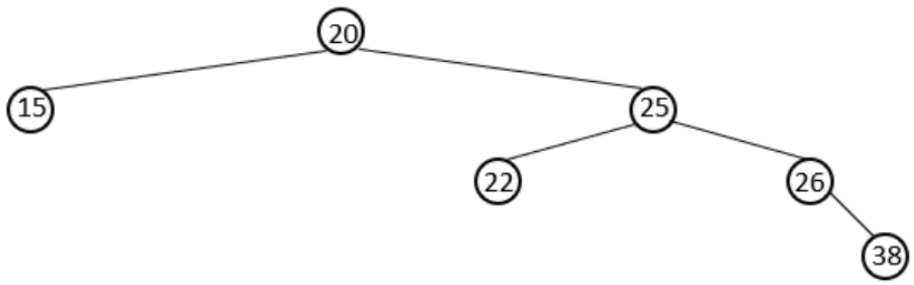
La méthode `rotation_droite` ci-après renvoie une nouvelle instance de type ABR, correspondant à une rotation droite de l'objet de type ABR à partir duquel elle est appelée :

```
    def rotation_droite(self):
        """ Renvoie une instance d'un ABR apres uns rotation droite
            On suppose qu'il existe un sous-arbre gauche"""
        pivot = self.sa_gauche
        self.sa_gauche = self.sa_droit
        pivot.sa_droit = self
        return ABR(pivot.valeur, pivot.sa_gauche, pivot.sa_droit)
```

Pour réaliser une rotation gauche, on suivra alors l'algorithme suivant :

- ▷ on appelle *pivot* le sous-arbre droit de la racine de l'arbre,
- ▷ le sous-arbre gauche du pivot devient le sous-arbre droit de la racine,
- ▷ la racine ainsi modifiée devient le sous-arbre gauche du pivot et la racine du pivot devient la nouvelle racine de l'ABR

(a) En suivant les différentes étapes de cet algorithme, dessiner l'arbre obtenu après une rotation gauche de l'ABR suivant :



(b) Écrire le code d'une méthode python `rotation_gauche` qui réalise la rotation gauche d'un ABR autour de sa racine.

4 Liban J1 - 2023

Exercice 4

Cet exercice traite des piles, des arbres et de l'algorithmique.

Dans cet exercice, on s'intéresse à la notation polonaise inversée (NPI) d'une expression mathématique. Dans cette notation, l'opérateur est placé après les nombres sur lesquels il s'applique. On se limitera aux expressions faisant intervenir des nombres entiers et les quatre opérateurs : $+$, $-$,$\times$, $/$.

Par exemple, l'expression $4 \times (5 + 7)$ s'écrit, en NPI : $457+\times$.

L'évaluation de l'expression $12/2-4+5\times3$, écrite en NPI : 12 2 / 4 − 5 3 × + est détaillée ci-dessous. Cette expression s'évalue à 17 de la manière suivante :

- ▷ lorsqu'on rencontre un premier opérateur ($+$, $-$, $/$, $\times$), on évalue l'opération avec les deux nombres situés juste avant cet opérateur :

$$\underbrace{12\ 2\ /}_{12/2=6}\ 4\ -\ 5\ 3\ \times\ +$$

- ▷ on remplace cette opération par le résultat :

$$6\ 4\ -\ 5\ 3\ \times\ +$$

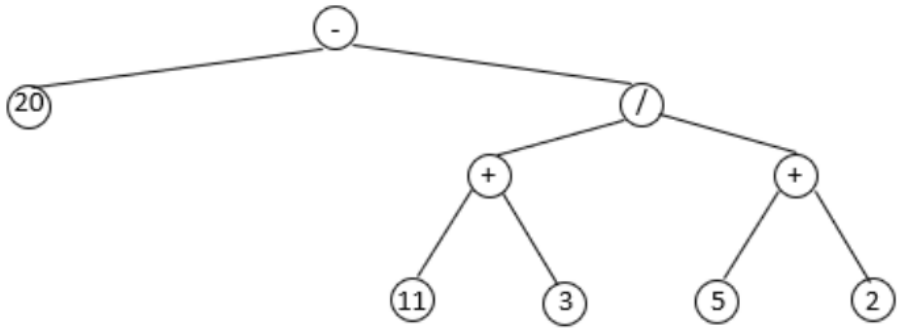
- ▷ on continue la lecture de l'expression :

$$\underbrace{6\ 4\ -}_{6-4=2}\ 5\ 3\ \times\ +\ \text{devient}\ 2\ \underbrace{5\ 3\ \times}_{5\times3=15}\ +\ \text{devient}\ 2\ 15\ +\ \text{qui vaut 17.}$$

Q1. Donner la valeur de l'expression suivante écrite en NPI :

$$15\ 5\ -\ 4\ 12\ +\ \times$$

Q2. On donne l'arbre binaire suivant :



Indiquer, parmi les quatre différents parcours d'arbre ci-dessous celui qui permet d'obtenir la succession des symboles correspondant à la notation NPI suivante :

$$20\ 11\ 3\ +\ 5\ 2\ +\ /\ -$$

- ▷ un parcours en largeur ;
- ▷ un parcours en profondeur préfixe ;
- ▷ un parcours en profondeur infixe ;
- ▷ un parcours en profondeur postfixe (ou suffixe).

Q3. On utilisera une liste de symboles pour écrire la notation polonaise inversée d'une expression mathématique. Par exemple, 6 2 3 + × 94 1 − + sera représentée par la liste :

```
[6, 2, 3, +, x, 94, 1, -, +]
```

On considère à présent l'algorithme ci-dessous, écrit en pseudo-code, qui reçoit une liste de symboles correspondant à la notation polonaise inversée d'une expression et renvoie l'arbre binaire associé.

Algorithme `créer_arbre(liste_symboles)`

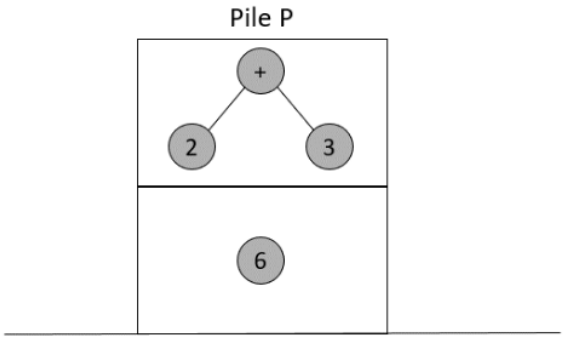
```
P ← créer une pile vide
pour chaque élément de liste_symboles :
    si élément est un entier :
        a ← construire l'arbre de racine élément et n'ayant pas
            sous-arbre droit ni gauche
    sinon : # élément est le symbole d'un opérateur
        a_droit ← dépiler P
        a_gauche ← dépiler P
        a ← construire l'arbre de racine élément, ayant pour sous-arbre
            gauche a_gauche et pour sous-arbre droit a_droit
    fin si
    empiler a dans P
fin pour
a ← dépiler P
renvoyer a
```

L'algorithme précédent nécessite l'utilisation d'une pile, dont les éléments sont des arbres.
On précise que :

- ▷ l'action empiler a dans P insère l'élément a en haut de la pile P ;
- ▷ l'action dépiler P renvoie la valeur de l'élément en haut de la pile P et le supprime de la pile.

(a) Des deux acronymes LIFO et FIFO, indiquer lequel permet de décrire, de manière générale, la structure de pile. Donner la signification des quatre lettres qui composent cet acronyme.

(b) On applique l'algorithme précédent sur la liste `[6, 2, 3, +, x, 94, 1, -, +]`.
À la fin de l'exécution du quatrième tour de la boucle pour, l'état de la pile est le suivant :



Indiquer le nombre d'élément(s) contenu(s) dans la pile P et dessiner l'état de la pile, à la fin de l'exécution du cinquième tour de la boucle `pour`.

(c) Dessiner l'arbre binaire représentant l'expression mathématique dont la notation polonaise inversée est la liste `[6, 2, 3, +, x, 94, 1, -, +]`.

Q4. On souhaite écrire un e fonction en langage Python qui permet d'évaluer une expression mathématique écrite avec la notation polonaise inversée NPI. On suppose qu'une telle expression est représentée par un arbre binaire obtenu à l'aide de l'algorithme précédent.

On dispose pour cela de quatre fonctions :

- ▷ `est_vide(arb)` qui renvoie `True` si l'arbre binaire `arb` est vide, `False` sinon ;
- ▷ `racine(arb)` qui renvoie la valeur de la racine de l'arbre binaire `arb` ;
- ▷ `gauche(arb)` qui renvoie le sous-arbre gauche de l'arbre binaire `arb` ;
- ▷ `droit(arb)` qui renvoie le sous-arbre droit de l'arbre binaire `arb`.

La fonction `evaluer` donnée ci-dessous est écrite en `python`.

Cette fonction prend en paramètre un arbre binaire `arb` représentant une expression mathématique écrite en NPI et renvoie la valeur de cette expression.

```
def evaluer(arb):
    if est_vide(gauche(arb)) and (instruction1):
        res = (instruction2)
    elif (instruction2) == "+":
        res = evaluer(instruction3) + (instruction4)
    elif (instruction2) == "-":
        res = evaluer(instruction3) - (instruction4)
    elif (instruction2) == "*":
        res = evaluer(instruction3) * (instruction4)
    else:
        res = evaluer(instruction3) / (instruction4)
    return res
```

Quatre instructions seulement permettent de compléter ce code : `instruction1`, `instruction2`, `instruction3` et `instruction4`.

Écrire sur la copie le code de chacune de ces quatre instructions.

5 Centres-Etrangers J2 - 2023

Exercice 5

Cet exercice porte sur la programmation en `python`*, la manipulation des chaînes de caractères, les arbres binaires de recherche et le parcours de liste.*

Q1. On rappelle ici quelques notions sur la manipulation des chaînes de caractères en `python`.
Une chaîne de caractères se comporte comme un tableau de caractères que l'on ne peut pas modifier.

Par exemple, on a le comportement suivant :

```
>>> une_chaine = 'Bonjour'
>>> une_chaine[3]
'j'
>>> une_chaine[3] = 'z'
TypeError: 'str' object does not support item assignment
```

On peut aussi utiliser l'opérateur de concaténation `+`.

```
>>> une_chaine = 'a' + 'b'
>>> une_chaine
'ab'
>>> une_chaine = une_chaine + 'c'
>>> une_chaine
'abc'
```

On définit la fonction bonjour par le code suivant :

```
def bonjour(nom) :
    return 'Bonjour ' + nom + ' !'
```

(a) Donner le résultat de l'exécution de `bonjour('Alan')`.

On exécute le programme suivant :

```
une_chaine='Bonjour'
x = (une_chaine[2] == une_chaine[3])
y = (une_chaine[4] == une_chaine[1])
```

(b) Donner le type et les valeurs des variables `x` et `y`.

(c) Écrire une fonction `occurrences_lettre(une_chaine, une_lettre)` prenant en paramètres une chaîne `une_chaine` et une lettre `une_lettre` et renvoyant le nombre d'occurrences de `une_lettre` dans `une_chaine`.

Q2. On rappelle qu'un arbre binaire de recherche est un arbre binaire pour lequel chaque nœud possède une étiquette dont la valeur est supérieure ou égale à toutes les étiquettes des nœuds de son fils gauche et strictement inférieure à celles des nœuds de son fils droit.
Sa taille est son nombre de nœuds; sa hauteur est le nombre de niveaux qu'il contient.
On rappelle aussi que l'on peut comparer des cha înes de caractères en utilisant l'ordre alphabétique.
On a par exemple :

```
>>> une_chaine = 'Bonjour'
>>> une_chaine[3]
'j'
>>> une_chaine[3] = 'z'
TypeError: 'str' object does not support item assignment
```

On considère la liste de mots :
`animaux = ['python', 'chameau', 'pingouin', 'renard', 'gnou']`

(a) Dessiner un arbre binaire de recherche contenant tous les mots de la liste `animaux` et de hauteur minimale.

(b) Dessiner un arbre binaire de recherche contenant tous ces mots et de hauteur maximale.

Q3. On considère l'implémentation objet suivante d'un arbre binaire de recherche :
On dispose d'une classe `Abr` contenant notamment les méthodes et attributs suivants :

- ▷ Si `un_abr` est une instance d'`Abr` alors `un_abr.est_vide()` renvoie `True` si l'arbre est vide et `False` sinon.
- ▷ Si `un_abr` est une instance d'`Abr` et si `un_abr.est_vide()` renvoie `False`, alors `un_abr.valeur` contient une chaîne de caractères représentant la valeur de la racine de l'arbre.
- ▷ Si `un_abr` est une instance d'`Abr` et si `un_abr.est_vide()` renvoie `False`, alors `un_abr.sous_arbre_gauche` et `un_abr.sous_arbre_droit` contiennent chacun une instance d'`Abr`.

On considère que la variable `liste_mots_francais` est une liste de 336 531 mots en français et que la variable `abr_mots_francais` est une instance d'`Abr` contenant les mots de la liste.

On considère la fonction suivante :

```
1 def mystere(un_abr):
2     if un_abr.est_vide():
3         return 0
4     else:
5         return 1 + mystere(un_abr.sous_arbre_gauche) \
6             + mystere(un_abr.sous_arbre_droit)
```

Remarque : on rappelle que le caractère \ en fin de ligne 5 indique que l'instruction se poursuit sur la ligne suivante.

(a) Donner le résultat de `mystere(abr_mots_francais)`, en justifiant le résultat.

On veut calculer la hauteur de `abr_mots_francais`.

(b) Donner le code d'une fonction `hauteur(un_abr)` permettant de faire ce calcul, en vous inspirant du code précédent.

Q4. Dans cette question, nous nous servirons uniquement de `liste_mots_francais` et plus de `abr_mots_francais`.

Pour aider à la résolution de mots croisés, on a décidé d'écrire une fonction `chercher_mots(liste_mots, longueur, lettre, position)` où `liste_mots` est une liste de mots français, `longueur` est la taille du mot recherché, `lettre` est une lettre du mot se trouvant à l'indice `position`.

Par exemple `chercher_mots(liste_mots_francais,3,'x',2)` renverra
`['aux', 'box', 'dix', 'eux', 'fax', 'fox', 'lux', 'max', 'six']`.

(a) Recopier et compléter la ligne 4 de la fonction ci-dessous :

```
1 def chercher_mots(liste_mots,longueur,lettre,position):
2     res = []
3     for i in range(len(liste_mots)):
4         if ... and ... :
5             res.append(liste_mots[i])
6     return res
```

(b) Expliquer ce que donne la commande suivante.

```
>>> chercher_mots(chercher_mots(liste_mots_francais,3,'x',2),3,'a',1)
```

On cherche un mot de 5 lettres dont on connait la fin `'ter'`.

(c) Écrire la commande permettant de chercher les mots candidats dans `liste_mots_francais`.

6 Sujet 0-b - 2023

Exercice 6

Cet exercice est consacré aux arbres binaires de recherche et à la notion d'objet.

Q1. Voici la définition d'une classe nommée `ArbreBinaire`, en python :

```
class ArbreBinaire:
    """ Construit un arbre binaire """

    def __init__(self, valeur):
        """ Crée une instance correspondant
        à un état initial """
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None

    def insert_gauche(self, valeur):
        """ Insère le paramètre valeur
        comme fils gauche """
        if self.enfant_gauche is None:
            self.enfant_gauche = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_gauche = self.enfant_gauche
            self.enfant_gauche = new_node

    def insert_droit(self, valeur):
        """ Insère le paramètre valeur
        comme fils droit """
        if self.enfant_droit is None:
            self.enfant_droit = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_droit = self.enfant_droit
            self.enfant_droit = new_node

    def get_valeur(self):
        """ Renvoie la valeur de la racine """
        return self.valeur

    def get_gauche(self):
        """ renvoie le sous arbre gauche """
        return self.enfant_gauche

    def get_droit(self):
        """ renvoie le sous arbre droit """
        return self.enfant_droit
```

(a) En utilisant la classe définie ci-dessus, donner un exemple d'attribut, puis un exemple de méthode.

(b) Après avoir défini la classe `ArbreBinaire`, on exécute les instructions `python` suivantes :

```
r = ArbreBinaire(15)
r.insert_gauche(6)
r.insert_droit(18)
a = r.get_valeur()
b = r.get_gauche()
c = b.get_valeur()
```

Donner les valeurs associées aux variables `a` et `c` après l'exécution de ce code.

On utilise maintenant la classe `ArbreBinaire` pour implémenter un arbre binaire de recherche.

On utilisera la définition suivante : un arbre binaire de recherche est un arbre binaire, dans lequel :

- ▷ on peut comparer les valeurs des nœuds : ce sont par exemple des nombres entiers, ou des lettres de l'alphabet.
- ▷ si `x` est un nœud de cet arbre et `y` est un nœud du sous-arbre gauche de `x`, alors il faut que `y.valeur <= x.valeur`.
- ▷ si `x` est un nœud de cet arbre et `y` est un nœud du sous-arbre droit de `x`, alors il faut que `y.valeur => x.valeur`.

Q2. On exécute le code `python` suivant. Représenter graphiquement l'arbre ainsi obtenu.

```
racine_r = ArbreBinaire(15)
racine_r.insert_gauche(6)
racine_r.insert_droit(18)
r_6 = racine_r.get_gauche()
r_6.insert_gauche(3)
r_6.insert_droit(7)
r_18 = racine_r.get_droit()
r_18.insert_gauche(17)
r_18.insert_droit(20)
r_3 = r_6.get_gauche()
r_3.insert_gauche(2)
```

Q3. On a représenté sur la figure 1 ci-dessous un arbre. Justifier qu'il ne s'agit pas d'un arbre binaire de recherche. Redessiner cet arbre sur votre copie en conservant l'ensemble des valeurs 2 ; 3 ; 5 ; 10 ; 11 ; 12 ; 13 pour les nœuds afin qu'il devienne un arbre binaire de recherche.

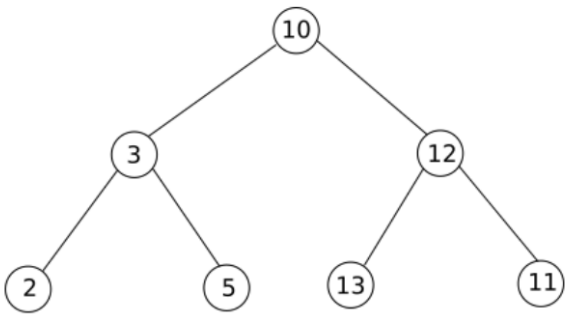


Figure 1

Q4. On considère qu'on a implémenté un objet `ArbreBinaire` nommé A représenté sur la figure 2.

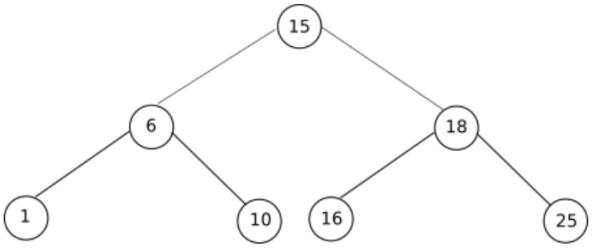


Figure 2

On définit la fonction `parcours_infixe` suivante, qui prend en paramètre un objet `ArbreBinaire` `T` et un second paramètre `parcours` de type liste.

```
def parcours_infixe(T, parcours):
    """ Affiche la liste des valeurs de l'arbre """
    if T is not None:
        parcours_infixe(T.get_gauche(), parcours)
        parcours.append(T.get_valeur())
        parcours_infixe(T.get_droit(), parcours)
    return parcours
```

Donner la liste renvoyée par l'appel suivant :`parcours_infixe(A,[ ])`.

7 Sujet 0-a - 2023

Exercice 7

L'exercice porte sur les arbres binaires de recherche et la programmation objet.

Dans un entrepôt de e-commerce, un robot mobile autonome exécute successivement les tâches qu'il reçoit tout au long de la journée.
La mémorisation et la gestion de ces tâches sont assurées par une structure de données.

Q1. Dans l'hypothèse où les tâches devraient être extraites de cette structure (pour être exécutées) dans le même ordre qu'elles ont été mémorisées, préciser si ce fonctionnement traduit le comportement d'une file ou d'une pile. Justifier.

En réalité, selon l'urgence des tâches à effectuer, on associe à chacune d'elles, lors de la mémorisation, un indice de priorité (nombre entier) distinct : il n'y a pas de valeur en double.

Plus cet indice est faible, plus la tâche doit être traitée prioritairement.

La structure de données retenue est assimilée à un arbre binaire de recherche (ABR) dans lequel chaque nœud correspond à une tâche caractérisée par son indice de priorité.

> **Rappel** : Dans un arbre binaire de recherche, chaque nœud est caractérisé par une valeur (ici l'indice de priorité), telle que chaque nœud du sous-arbre gauche a une valeur strictement inférieure à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une valeur strictement supérieure à celle-ci.
> Cette structure de données présente l'avantage de mettre efficacement en œuvre l'insertion ou la suppression de nœuds, ainsi que la recherche d'une valeur.

Par exemple, le robot a reçu successivement, dans l'ordre, des tâches d'indice de priorité 12, 6, 10, 14, 8 et 13. En partant d'un arbre binaire de recherche vide, l'insertion des différentes priorités dans cet arbre donne la figure 1.

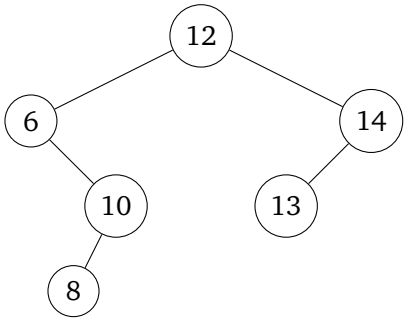


FIGURE 1 – Exemple d'un arbre binaire

Q2. En utilisant le vocabulaire couramment utilisé pour les arbres, préciser le terme qui correspond :

(a) au nombre de tâches restant à effectuer, c'est-à-dire le nombre total de nœuds de l'arbre ;

(b) au nœud représentant la tâche restant à effectuer la plus ancienne ;

(c) au nœud représentant la dernière tâche mémorisée (la plus récente).

Q3. Lorsque le robot reçoit une nouvelle tâche, on déclare un nouvel objet, instance de la classe `Noeud`, puis on l'insère dans l'arbre binaire de recherche (instance de la classe `ABR`) du robot. Ces deux classes sont définies comme suit :

```
1  class Noeud:
2      def __init__(self, tache, indice):
3          self.tache = tache   #ce que doit accomplir le robot
4          self.indice = indice #indice de priorité (int)
5          self.gauche = ABR()  #sous-arbre gauche vide (ABR)
6          self.droite = ABR()  #sous-arbre droit vide (ABR)
7
8
9  class ABR:
10     #arbre binaire de recherche initialement vide
11     def __init__(self):
12         self.racine = None #arbre vide
13         #Remarque : si l'arbre n'est pas vide, racine est
14         #une instance de la classe Noeud
15
16     def est_vide(self):
17         """renvoie True si l'arbre autoréférencé est vide,
18         False sinon"""
19         return self.racine == None
20
21     def insere(self, nouveau_noeud):
22         """insere un nouveau noeud, instance de la classe
23         Noeud, dans l'ABR"""
24         if self.est_vide():
25             self.racine = nouveau_noeud
26         elif self.racine.indice ... nouveau_noeud.indice:
27             self.racine.gauche.insere(nouveau_noeud)
28         else:
29             self.racine.droite.insere(nouveau_noeud)
```

(a) Donner les noms des attributs de la classe `Noeud`.

(b) Expliquer en quoi la méthode `insere` est dite récursive et justifier rapidement qu'elle se termine.

(c) Indiquer le symbole de comparaison manquant dans le test à la **ligne 26** de la méthode `insere` pour que l'arbre binaire de recherche réponde bien à la définition de l'encadré « **Rappel** » de l'énoncé.

(d) On considère le robot dont la liste des tâches est représentée par l'arbre de la figure 1. Ce robot reçoit, successivement et dans l'ordre, des tâches d'indice de priorité 11, 5, 16 et 7, sans avoir accompli la moindre tâche entretemps.
Recopier et compléter la figure 1 après l'insertion de ces nouvelles tâches.

Q4. Avant d'insérer une nouvelle tâche dans l'arbre binaire de recherche, il faut s'assurer que son indice de priorité n'est pas déjà présent.
Écrire une méthode `est_present` de la classe `ABR` qui répond à la description :

```
41     def est_present(self, indice_recherche) :
42         """renvoie True si l'indice de priorité indice_recherche
43         (int) passé en paramètre est déjà l'indice d'un noeud
44         de l'arbre, False sinon"""
```

Q5. Comme le robot doit toujours traiter la tâche dont l'indice de priorité est le plus petit, on envisage un parcours infixe de l'arbre binaire de recherche.

(a) Donner l'ordre des indices de priorité obtenus à l'aide d'un parcours infixe de l'arbre binaire de recherche de la figure 1.

(b) Expliquer comment exploiter ce parcours pour déterminer la tâche prioritaire.

Q6. Afin de ne pas parcourir tout l'arbre, il est plus efficace de rechercher la tâche du nœud situé le plus à gauche de l'arbre binaire de recherche : il correspond à la tâche prioritaire.
Recopier et compléter la méthode récursive `tache_prioritaire` de la classe `ABR` :

```
61    def tache_prioritaire(self):
62        """renvoie la tache du noeud situé le plus
63        à gauche de l'ABR supposé non vide"""
64        if self.racine. ... .est_vide(): #pas de noeud plus à gauche
65            return self.racine. ...
66        else:
67            return self.racine.gauche. ...()
```

Q7. Une fois la tâche prioritaire effectuée, il est nécessaire de supprimer le nœud correspondant pour que le robot passe à la tâche suivante :

- ▷ si le nœud correspondant à la tâche prioritaire est une feuille, alors il est simplement supprimé de l'arbre (cette feuille devient un arbre vide)
- ▷ si le nœud correspondant à la tâche prioritaire a un sous-arbre droit non vide, alors ce sous-arbre droit remplace le nœud prioritaire qui est alors écrasé, même s'il s'agit de la racine.

Dessiner alors, pour chaque étape, l'arbre binaire de recherche (seuls les indices de priorités seront représentés) obtenu pour un robot, initialement sans tâche, et qui a, successivement dans l'ordre :

- ▷ étape 1 : reçu une tâche d'indice de priorité 14 à accomplir
- ▷ étape 2 : reçu une tâche d'indice de priorité 11 à accomplir
- ▷ étape 3 : reçu une tâche d'indice de priorité 8 à accomplir
- ▷ étape 4 : accompli sa tâche prioritaire
- ▷ étape 5 : reçu une tâche d'indice de priorité 12 à accomplir
- ▷ étape 6 : accompli sa tâche prioritaire
- ▷ étape 7 : accompli sa tâche prioritaire
- ▷ étape 8 : reçu une tâche d'indice de priorité 15 à accomplir
- ▷ étape 9 : reçu une tâche d'indice de priorité 19 à accomplir
- ▷ étape 10 : accompli sa tâche prioritaire