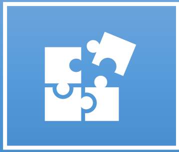


Algorithmique

Les arbres binaires

Objectifs pédagogiques :

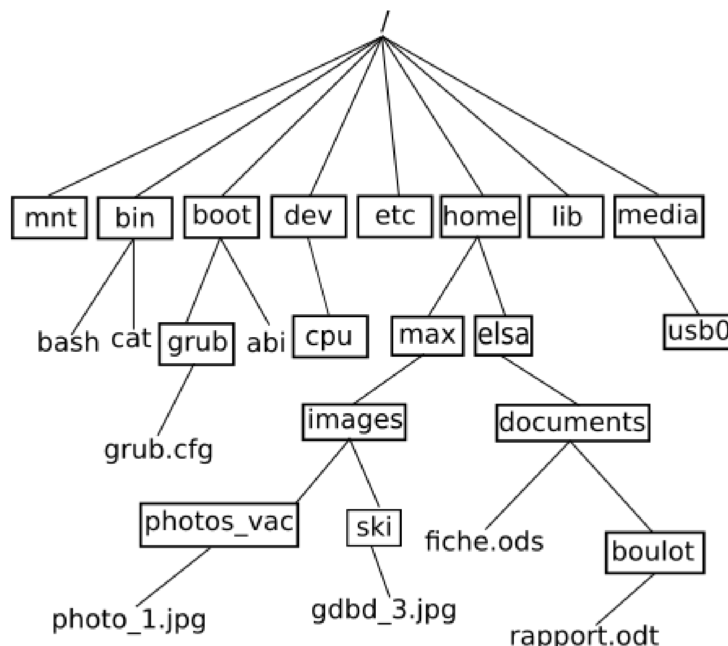
- ✓ Identifier des situations nécessitant une structure de données arborescente.
- ✓ Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).
- ✓ Calculer la taille et la hauteur d'un arbre. Parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur d'abord).
- ✓ Rechercher une clé dans un arbre de recherche, insérer une clé.

1. Qu'est-ce qu'un arbre ?

1.1. Notion d'arbre

Les **structures de données** que nous avons étudié jusqu'à présent (tableaux, listes, piles, files…) sont **linéaires**, dans la mesure où elles stockent les éléments les uns à la suite des autres « à la queue leu leu ». On peut représenter de telles structures d'une manière imagée comme des éléments placés sur une ligne à l'instar d'oiseaux posés sur un fil électrique.

Un **arbre** est une structure constituée de **nœuds**, qui peuvent avoir des **enfants** qui sont eux-mêmes des nœuds. Les systèmes de fichiers dans les systèmes de type UNIX (Linux et Mac OS) ont par exemple une structure en arbre définissant une arborescence :



Système de fichier UNIX (arborescence)

Les arbres sont des structures mathématiques abstraites très utilisés en informatique. On les utilise notamment quand on a besoin d'une structure hiérarchisée des données : dans l'exemple ci-dessus le fichier grub.cfg ne se trouve pas au même niveau de l'arborescence que le fichier rapport.odt (le fichier grub.cfg se trouve "plus proche" de la racine / que le fichier rapport.odt). On ne pourrait pas avec une simple liste qui contiendrait les noms des fichiers et des répertoires, rendre compte de cette hiérarchie (plus ou moins "proche" de la racine).

1.2. Notion d'arbre binaire

Les arbres binaires sont des cas particuliers d'arbre : dans un arbre binaire, on a au maximum 2 branches qui partent d'un élément. Dans la suite de l'activité nous travaillerons uniquement sur les arbres binaires.

■ Un peu de vocabulaire

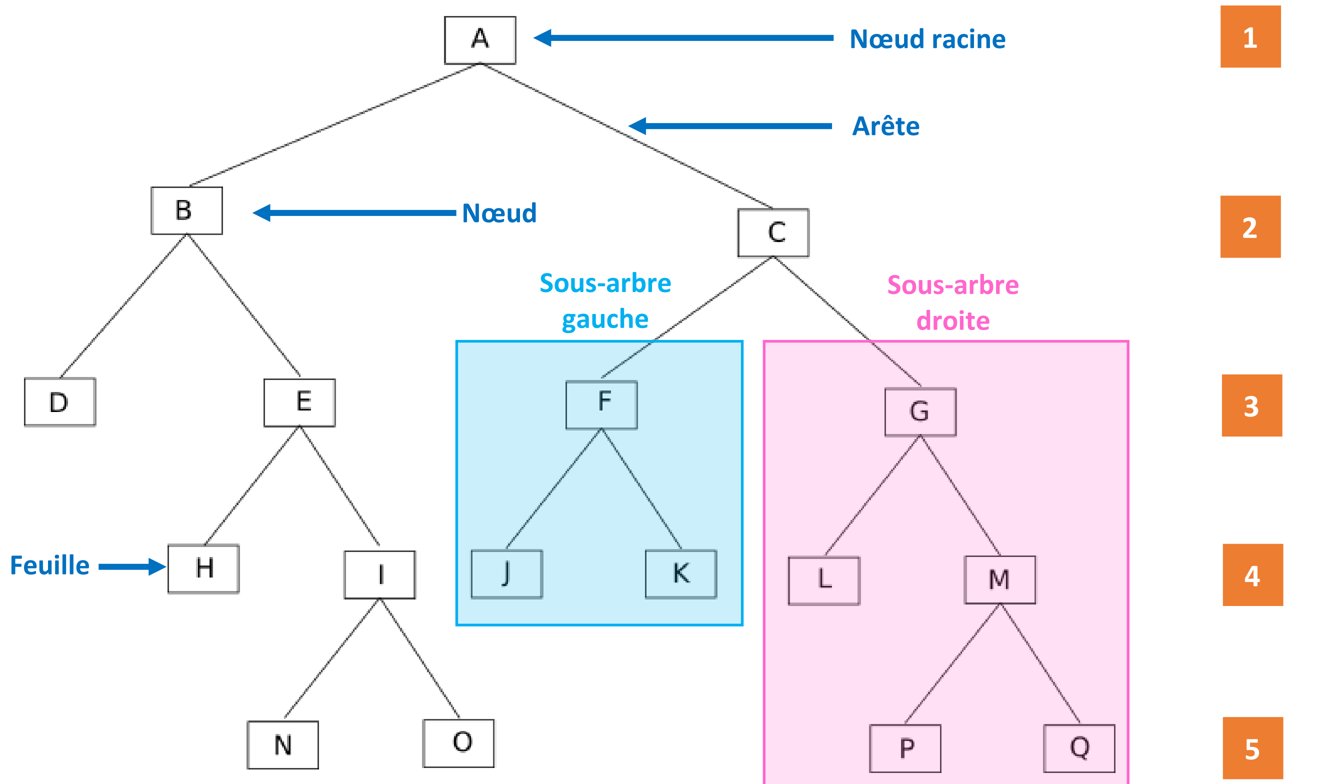
En informatique, un **arbre binaire** est une **structure de données** qui peut se représenter sous la forme d'une **hiérarchie** dont chaque élément est appelé **nœud**, le **nœud initial** étant appelé **racine**. Dans un arbre binaire, chaque élément possède au plus **deux éléments fils** au **niveau inférieur**, habituellement appelés **gauche** et **droit**. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé **père**.

Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a au plus deux nœuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un nœud n'ayant aucun fils est appelé **feuille**. Le nombre de niveaux total, autrement dit la distance entre la feuille la plus éloignée et la racine, est appelé **hauteur** de l'arbre.

Le niveau d'un nœud est appelé **profondeur**.

D'après WIKIPEDIA

Exemple :



■ Quelques précisions concernant l'arbre binaire précédent :

- Chaque élément de l'arbre est appelé **nœud** (par exemple : A, B, C, D,...,P et Q sont des nœuds)
- Le nœud initial (A) est appelé **nœud racine** ou plus simplement **racine**.
- On appelle **arête** le segment qui relie 2 nœuds.
- Le nœud E et le nœud D sont les **fils** du nœud B. Réciproquement, on dira que le nœud B est le **père** des nœuds E et D.
- Dans un arbre binaire, un nœud possède au plus 2 fils.
- Un nœud n'ayant aucun fils est appelé **feuille** (exemples : D, H, N, O, J, K, L, P et Q sont des feuilles).

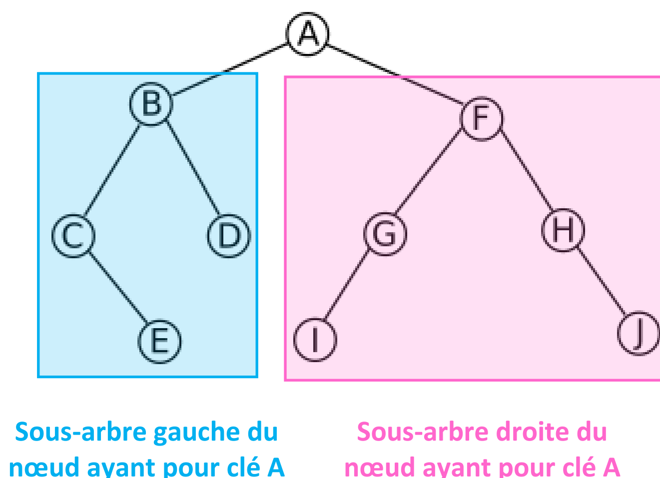
- À partir d'un nœud (qui n'est pas une feuille), on peut définir un **sous-arbre gauche** et un **sous-arbre droite** (exemple : à partir de C on va trouver un sous-arbre gauche composé des nœuds F, J et K et un sous-arbre droit composé des nœuds G, L, M, P et Q).
- La **profondeur** d'un nœud ou d'une feuille dans un arbre binaire est le nombre de nœuds du chemin qui va de la racine à ce nœud. La racine d'un arbre est à une profondeur 1, et la profondeur d'un nœud est égale à la profondeur de son prédécesseur plus 1. Si un nœud est à une profondeur p, tous ses successeurs sont à une profondeur p+1. **Exemples :** profondeur de B = 2 ; profondeur de I = 4 ; profondeur de P = 5.
- La hauteur d'un arbre est la profondeur maximale des nœuds de l'arbre. **Exemple :** la profondeur de P = 5, c'est un des nœuds les plus profond, donc la hauteur de l'arbre est de 5.

Il est important de remarquer que l'on peut aussi voir les arbres comme des structures récursives : les fils d'un nœud sont des arbres (sous-arbre gauche et un sous-arbre droite dans le cas d'un arbre binaire), ces arbres sont eux-mêmes constitués d'arbres...

Remarque : Python ne propose pas de façon native l'implémentation des arbres binaires. Il faudra pour les modéliser faire appel à la programmation orientée objet (P.O.O.) comme nous allons le voir-ci après.

1.3. Arbre binaire et clés

À chaque nœud d'un arbre binaire, on peut associer une clé correspondant au nom du nœud et servant d'étiquette.



Sous-arbre gauche du nœud ayant pour clé A

Sous-arbre droite du nœud ayant pour clé A

En partant du nœud racine dont la clé est A on peut distinguer :

- le sous arbre gauche composé du nœud ayant pour clé B, du nœud ayant pour clé C, du nœud ayant pour clé D et du nœud ayant pour clé E ;
- le sous arbre droite est composé du nœud ayant pour clé F, du nœud ayant pour clé G, du nœud ayant pour clé H, du nœud ayant pour clé I et du nœud ayant pour clé J.

Un **arbre** (ou un **sous-arbre**) **vide** est noté **NIL** (abréviation du latin nihil qui veut dire rien).

Si on prend le nœud ayant pour clé G on a :

- le sous-arbre gauche est uniquement composé du nœud ayant pour clé I
- le sous-arbre droit est vide (NIL)

Il faut bien avoir à l'esprit qu'un sous-arbre (gauche ou droite) est un arbre (même s'il contient un seul nœud ou pas de nœud du tout (NIL)).

2. Hauteur et taille d’un arbre binaire

2.1. Notations utilisées dans les algorithmes

- Soit un **arbre T** : **T.racine** correspond au nœud racine de l'arbre T
- Soit un **noeud x** :
  - **x.gauche** correspond au **sous-arbre gauche** du **noeud x**
  - **x.droit** correspond au **sous-arbre droite** du **noeud x**
  - **x.clé** correspond à la **clé** du **noeud x**

Il faut noter que si le **noeud x** est une feuille, **x.gauche** et **x.droite** sont des arbres vides (NIL)

2.2. Calculer la hauteur d’un arbre

■ **Principe :**

- un arbre vide est de hauteur 0 ;
- un arbre non vide a pour hauteur 1 + la hauteur maximale entre ses fils.

■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
x : noeud

DEBUT
HAUTEUR(T) :
  si T ≠ NIL :
    x ← T.racine
    renvoyer 1 + max(HAUTEUR(x.gauche), HAUTEUR(x.droit))
  sinon :
    renvoyer 0
  fin si
FIN
```

Remarque : la fonction max renvoie la plus grande valeur des 2 valeurs passées en paramètre, par exemple : max(5,6) renvoie 6.

Q1. Quelle est la particularité de la fonction HAUTEUR() ?
Q2. Appliquer l’algorithme ci-dessus sur l’arbre binaire du paragraphe 3.1. afin de calculer sa hauteur en détaillant votre raisonnement. On pourra en particulier compléter le calcul sous la forme suivante :

```
A ? ? ? ? ? ? ? ? ? ?
1+max(1+max(1+max(0, 1+max(0,0)), 1+max(0,0)), 1+max(1+max(1+max(0,0), 0), 1+max(0, 1+max(0,0)))) =
```

2.3. Calculer la taille d’un arbre (nombre de nœuds)

■ **Principe :**

- si l'arbre est vide : renvoyer 0
- sinon renvoyer 1 plus la somme du nombre de nœuds des sous arbres.

■ **Algorithme en pseudo-code:**

```
VARIABLE
T : arbre
x : noeud

DEBUT
TAILLE(T) :
  si T ≠ NIL :
    x ← T.racine
    renvoyer 1 + TAILLE(x.gauche) + TAILLE(x.droit)
  sinon :
    renvoyer 0
  fin si
FIN
```

Q3. Quelle est la particularité de la fonction TAILLE() ?
Q4. En vous inspirant du paragraphe précédent, appliquer l'algorithme ci-dessus sur l'arbre binaire du paragraphe 1.3. afin de calculer sa taille en détaillant votre raisonnement.

3. Parcours en profondeur d'un arbre binaire

3.1. Parcourir un arbre dans l'ordre infixe

■ **Principe :**
Dans un **parcours infixe** (inorder traversal), chaque nœud est visité après son enfant gauche mais avant son enfant droit. Le parcours infixe affiche la racine après avoir traité le sous arbre gauche, après traitement de la racine, on traite le sous arbre droit (c'est donc un parcours de type G R D : Gauche Racine Droite).

■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
x : noeud

DEBUT
PARCOURS-INFIXE(T) :
  si T ≠ NIL :
    x ← T.racine
    PARCOURS-INFIXE(x.gauche)
    affiche x.clé
    PARCOURS-INFIXE(x.droit)
  fin si
FIN
```

Q5. Quelle est la particularité de la fonction PARCOURS-INFIXE() ?
Q6. Dans quel ordre est parcouru l'arbre binaire du paragraphe 1.3. par la fonction PARCOURS-INFIXE() ?

3.2. Parcourir un arbre dans l'ordre préfixe

■ **Principe :**
Dans un **parcours préfixe** (preorder traversal), chaque nœud est visité avant que ses enfants soient visités. Cela signifie que l'on affiche la racine de l'arbre, on parcourt tout le sous arbre de gauche, une fois qu'il n'y a plus de sous arbre gauche on parcourt les éléments du sous arbre droit. Ce type de parcours peut être résumé en trois lettres : R G D (pour Racine Gauche Droit).

■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
x : noeud

DEBUT
PARCOURS-PREFIXE(T) :
  si T ≠ NIL :
    x ← T.racine
    affiche x.clé
    PARCOURS-PREFIXE(x.gauche)
    PARCOURS-PREFIXE(x.droit)
  fin si
FIN
```

Q7. Quelle est la particularité de la fonction PARCOURS-PREFIXE() ?
Q8. Dans quel ordre est parcouru l'arbre binaire du paragraphe 1.3. par la fonction PARCOURS-PREFIXE() ?

3.2. Parcourir un arbre dans l'ordre suffixe (ou postfixe)

■ **Principe :**
Dans un parcours suffixe (ou postfixe pour postorder traversal), chaque nœud est visité après que ses enfants sont visites. Le parcours postfixe effectue schématiquement le parcours suivant : sous arbre gauche, sous arbre droit puis la racine, c'est donc un parcours G D R (Gauche Droite Racine).

■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
x : noeud

DEBUT
PARCOURS-SUFFIXE(T) :
  si T ≠ NIL :
    x ← T.racine
    PARCOURS-SUFFIXE(x.gauche)
    PARCOURS-SUFFIXE(x.droit)
    affiche x.clé
  fin si
FIN
```

Q9. Quelle est la particularité de la fonction `PARCOURS-SUFFIXE()` ?
Q10. Dans quel ordre est parcouru l'arbre binaire du paragraphe 1.3. par la fonction `PARCOURS-SUFFIXE()` ?

Le choix du parcours infixe, préfixe ou suffixe dépend du problème à traiter, on pourra retenir pour les parcours préfixe et suffixe (le cas du parcours infixe sera traité un peu plus loin) que :

- dans le cas du parcours préfixe, un nœud est affiché avant d'aller visiter ces enfants
- dans le cas du parcours suffixe, on affiche chaque nœud après avoir affiché chacun de ses fils

4. Parcours en largeur d'un arbre binaire

■ **Principe :**

L'**algorithme de parcours en largeur** (ou BFS, pour *Breadth First Search* en anglais) permet le parcours d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. A partir d'un nœud source S, il liste d'abord les voisins de S pour ensuite les explorer un par un. Ce mode de fonctionnement utilise donc une file dans laquelle il prend le premier sommet et place en dernier ses voisins non encore explorés. Le principe de l'algorithme est le suivant :

1. Mettre le nœud source dans la file.
2. Retirer le nœud du début de la file pour le traiter.
3. Mettre tous les voisins non explorés dans la file (à la fin).
4. Si la file n'est pas vide reprendre à l'étape 2.

■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
Tg : arbre
Td : arbre
x : noeud
f : file (initialement vide)

DEBUT
PARCOURS-LARGEUR(T) :
  enfiler(T.racine, f) //on place la racine dans la file
  tant que f non vide :
    x ← defiler(f)
    affiche x.clé
    si x.gauche ≠ NIL :
      Tg ← x.gauche
      enfiler(Tg.racine, f)
    fin si
    si x.droit ≠ NIL :
      Td ← x.droite
      enfiler(Td.racine, f)
    fin si
  fin tant que
FIN
```

Q11. Dans quel ordre est parcouru l'arbre binaire du paragraphe 1.3. par la fonction `PARCOURS-LARGEUR()` ?
Q12. Selon vous, pourquoi parle-t-on de parcours en largeur ?

Remarque : on utilise une file (FIFO) pour cet algorithme de parcours en largeur. En outre, cet algorithme n'utilise pas de fonction récursive.

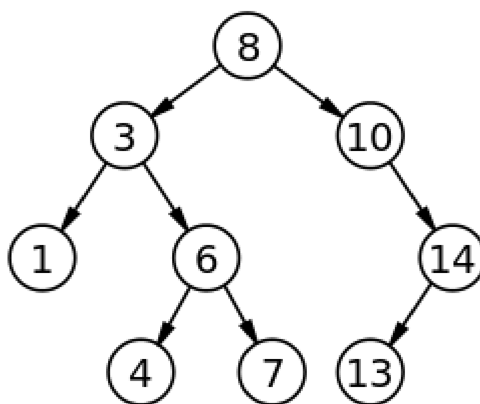
5. Arbre binaire de recherche

5.1. Notion d'arbre binaire de recherche

Les éléments stockés dans un **arbre binaire de recherche ABR** (en anglais, **B**inary **S**earch **T**ree ou **BST)** doivent posséder une **relation d'ordre totale**, c'est-à-dire qu'il doit exister une manière de dire si un élément est plus grand ou plus petit qu'un autre. Par exemple, on peut trier des nombres par valeur ou des personnes par ordre alphabétique de leur nom de famille.

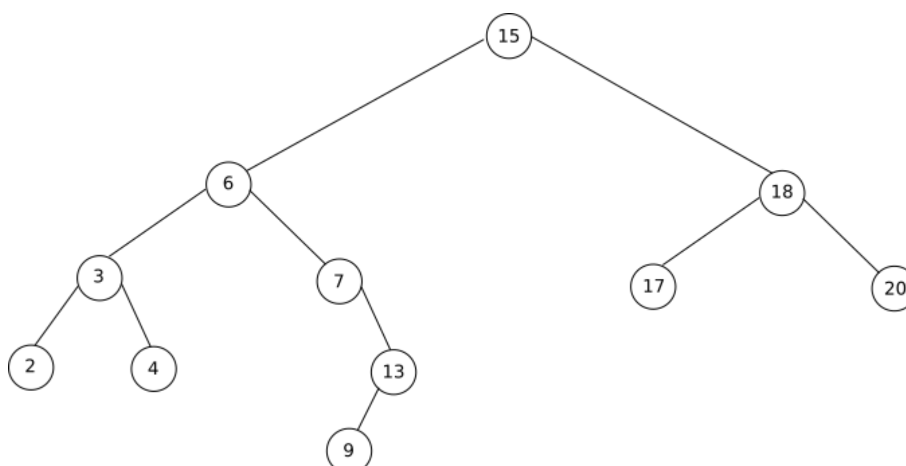
Le premier élément inséré dans l'arbre devient la racine. Ensuite, il suffit de mettre à gauche les éléments plus petits et à droite les éléments plus grands. C'est cette particularité qui rend les BST intéressants : la plupart des opérations réalisées sur les arbres binaires de recherche ont une complexité temporelle logarithmique O(log n) dans le cas moyen. Cela est dû au fait que, grâce à la relation d'ordre liant les valeurs, on peut naviguer dans l'arbre avec une logique rappelant une recherche dichotomique, ce qui est **plus performant** que les listes, par exemple, que l'on doit parcourir élément par élément.

Ainsi la séquence {8 3 10 1 6 14 4 7 13} peut-être traduite sous la forme de l'arbre binaire de recherche suivant :



Exemple d'arbre binaire

Q13. Quelle est la séquence associée à l'arbre binaire de recherche ci-dessous ?



Autre exemple d'arbre binaire de recherche :
Un vétérinaire voudrait stocker les fiches médicales de ses patients, et, plutôt que d'utiliser un tableau ou une liste, on se propose d'utiliser un arbre binaire. La fiche contiendra différentes informations sur l'animal ; on utilisera son nom comme **clé** (c'est-à-dire comme critère pour la relation d'ordre), que l'on triera selon l'ordre alphabétique croissant. Le vétérinaire reçoit sa première patiente, qui répond au nom de **Gaufrette**. Comme sa fiche sera le premier nœud de notre arbre, elle en devient automatiquement la racine. Puis le vétérinaire reçoit les animaux dans l'ordre suivant afin de les soigner : Charlie, Médor, Flipper, Bubulle et Augustin.

Q14. Construire l'arbre binaire de recherche associé à cette séquence.

Q15. Imaginons que la séquence soit maintenant la suivante : {Gaufrette, Charlie, Médor, Flipper, Augustin, Bubulle}, c'est-à-dire que Augustin soit arrivé au rendez-vous médical avant Bubulle. L'arbre binaire de recherche est-il encore le même ? Quelle conclusion peut-on en tirer ?

5.2. Recherche d'une clé dans un arbre binaire de recherche

Nous allons maintenant étudier un algorithme permettant de rechercher une clé de valeur k dans un arbre binaire de recherche. Si k est bien présent dans l'arbre binaire de recherche, l'algorithme renvoie vrai, dans le cas contraire, il renvoie faux.

■ **Principe**
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 sauf qu'elle accède directement à chaque élément d'un tableau au lieu de suivre des liens. La différence entre les deux algorithmes est que, dans la recherche dichotomique, on suppose avoir un critère de découpage de l'espace en deux parties que l'on n'a pas dans la recherche dans un arbre.

■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
x : noeud
k : entier
DEBUT
ARBRE-RECHERCHE(T,k) :
  si T == NIL :
    renvoyer faux
  fin si
  x ← T.racine
  si k == x.clé :
    renvoyer vrai
  fin si
  si k < x.clé :
    ARBRE-RECHERCHE(x.gauche,k)
  sinon :
    ARBRE-RECHERCHE(x.droit,k)
  fin si
FIN
```

Q16. Quelle est la particularité de la fonction ARBRE-RECHERCHE() ?

Q17. Appliquez l'algorithme de recherche d'une clé dans un arbre binaire de recherche sur l'arbre binaire de recherche de la question **Q13**. On prendra k = 13.

Q18. Appliquez l'algorithme de recherche d'une clé dans un arbre binaire de recherche sur l'arbre binaire de recherche de la question **Q13**. On prendra k = 16.

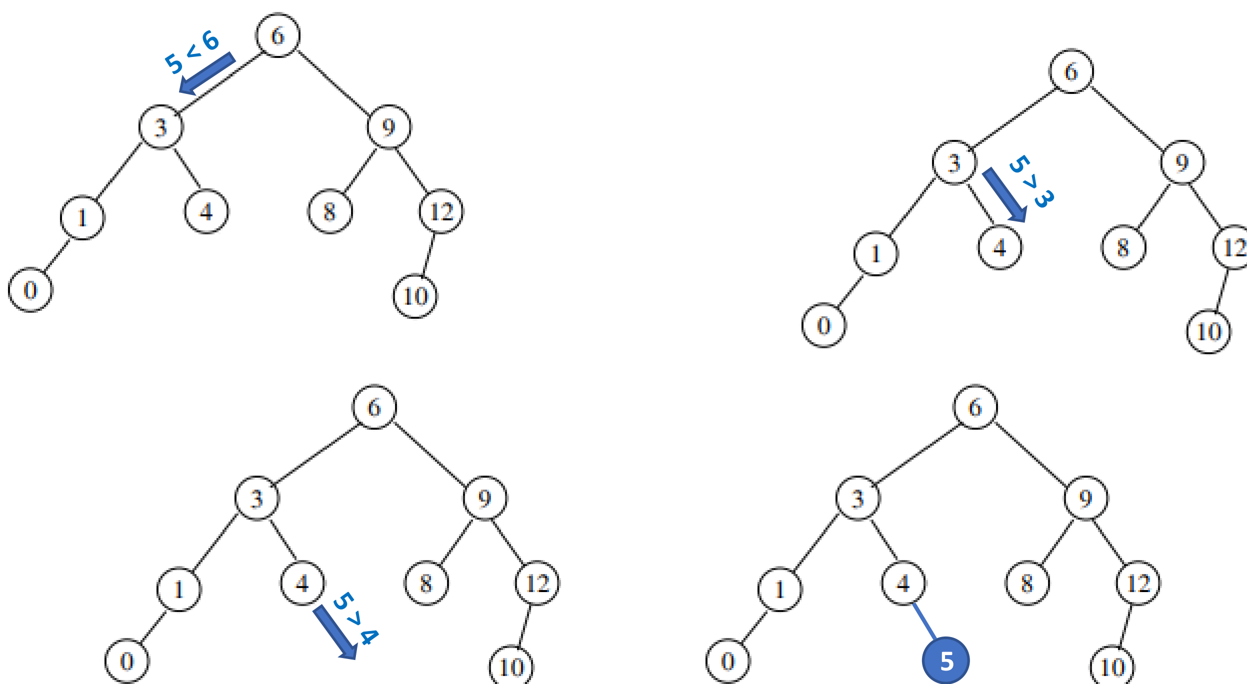
5.3. Insertion d'une clé dans un arbre binaire de recherche

Il est tout à fait possible d'insérer un nœud « y » dans un arbre binaire de recherche (non vide).

■ Principe

L'insertion d'un nœud commence par une recherche : on cherche la clé du nœud à insérer ; lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille : si elle est inférieure, le nouveau nœud sera à gauche ; sinon il sera à droite.

Exemple : insertion de de la valeur 5 dans l'arbre de recherche binaire suivant



■ **Algorithme en pseudo-code :**

```
VARIABLE
T : arbre
x : noeud
y : noeud
DEBUT
ARBRE-INSERTION(T,y) :
  x ← T.racine
  tant que T ≠ NIL :
    x ← T.racine
    si y.clé < x.clé :
      T ← x.gauche
    sinon :
      T ← x.droit
    fin si
  fin tant que
  si y.clé < x.clé :
    insérer y à gauche de x
  sinon :
    insérer y à droite de x
  fin si
FIN
```

Q19. Appliquez l'algorithme d'insertion d'un nœud « y » dans un arbre binaire de recherche sur l'arbre de la question **Q13**. On prendra y.clé = 16.

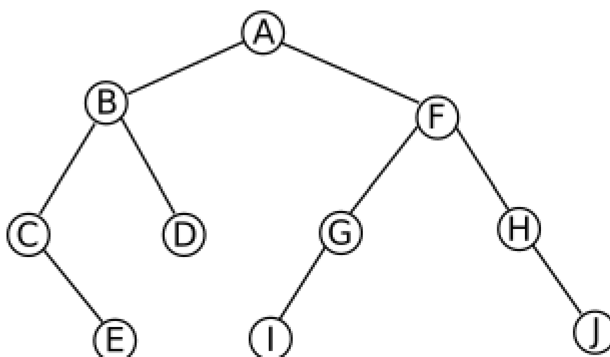
7. Implémentations Python

7.1. Implémentation d'un arbre binaire en POO

Python ne propose pas nativement de structure de données permettant d'implémenter directement les arbres binaires. Il va donc être nécessaire de créer manuellement cette structure. Pour programmer ce type de structure, nous allons utiliser le paradigme de la programmation orienté objet (P.O.O.).
Vous trouverez ci-dessous la classe "ArbreBinaire" qui va nous permettre d'implémenter des arbres binaires.

Entrée [1]:
```
class ArbreBinaire:
    '''
    Classe permettant de construire une arbre binaire.
    Attributs : valeur, enfant_gauche, enfant_droit
    '''
    # Constructeur (un arbre possède à minima un noeud)
    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None

    # Méthodes

    def insert_gauche(self, valeur):
        if self.enfant_gauche == 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):
        if self.enfant_droit == 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):
        return self.valeur

    def get_gauche(self):
        return self.enfant_gauche

    def get_droit(self):
        return self.enfant_droit
```

Q20. Complétez le programme précédent afin qu'il puisse construire l'arbre suivant à l'aide de la classe « ArbreBinaire ». On instanciera pour cela l'objet « racine » appartenant à la classe « ArbreBinaire ».



7.2. Affichage d'un arbre binaire dans la console Python

On peut afficher un arbre binaire dans la console Python en implémentant une fonction "affiche". Cette fonction renvoie une série de tuples de la forme (valeur, arbre_gauche, arbre_droite), comme "arbre_gauche" et "arbre_droite" seront eux-mêmes affichés sous forme de tuples, on aura donc un affichage qui ressemblera à :
(valeur,(valeur_gauche,arbre_gauche_gauche,arbre_gauche_droite),(valeur_droite,arbre_droite_gauche,arbre_droite_droite))

Mais comme "arbre_gauche_gauche" sera lui-même représenté par un tuple... Nous allons donc avoir des tuples qui contiendront des tuples qui eux-mêmes contiendront des tuples et ainsi de suite ...

Entrée [3]:
```
def affiche(T):
   if T != None:
      return (T.get_valeur(),affiche(T.get_gauche()),affiche(T.get_droit()))
```

Q21. Quel est le tuple correspondant au graphe de la question Q20 retourné par fonction affiche() ? Vérifier que la structure en « tuples imbriqués » correspond bien à l'arbre binaire étudié.

Q22. Programmez à l'aide de la classe "ArbreBinaire", l'arbre binaire suivant et vérifier avec la fonction affiche que le programme donne bien le résultat attendu :