Cours Lumière
enko schools network

Classe : TleNSI

STRUCTURES DE DONNEES : LES ARBRES & ALGORITHMES SUR LES ARBRES BINAIRES

LES ARBRES

Objectif :

Contenus	Capacités attendues	Commentaires
Arbres : structures hiérarchiques. Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	Identifier des situations nécessitant une structure de données arborescente. Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).	On fait le lien avec la rubrique « algorithmique ».

I- NOTION D'ARBRE

Nous allons travailler sur la structure de donnée "arbre". Cette structure n'est pas linéaire mais **hiérarchique**.

Un organisateur de tournoi de rugby recherche la meilleure solution pour afficher les potentiels quarts de final, demi-finales et finale :

Au départ nous avons 4 poules de 4 équipes. Les 4 équipes d'une poule s'affrontent dans un mini championnat (3 matchs par équipe). À l'issue de cette phase de poule, les 2 premières équipes de chaque poule sont qualifiées pour les quarts de finale.

Dans ce qui suit, on désigne les 2 qualifiés par poule par :

- Poule 1 => 1er Eq1 ; 2e Eq8
- Poule 2 => 1er Eq2 ; 2e Eq7
- Poule 3 => 1er Eq3 ; 2e Eq6
- Poule 4 => 1er Eq4 ; 2e Eq5

En quart de final on va avoir :

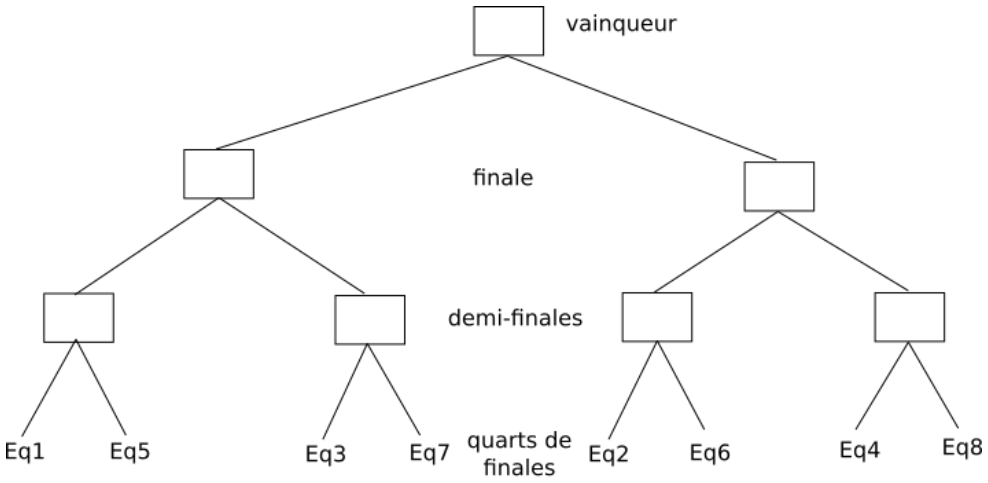
- quart de finale 1 => Eq1 contre Eq5
- quart de finale 2 => Eq2 contre Eq6
- quart de finale 3 => Eq3 contre Eq7
- quart de finale 4 => Eq4 contre Eq8

Pour les demi-finales on aura :

- demi-final 1 => vainqueur quart de finale 1 contre vainqueur quart de finale 3
- demi-final 2 => vainqueur quart de finale 2 contre vainqueur quart de finale 4

L'organisateur du tournoi affiche les informations ci-dessus le jour du tournoi. Malheureusement, la plupart des spectateurs se perdent quand ils cherchent à déterminer les potentielles demi-finales (et ne parlons pas de la finale !)

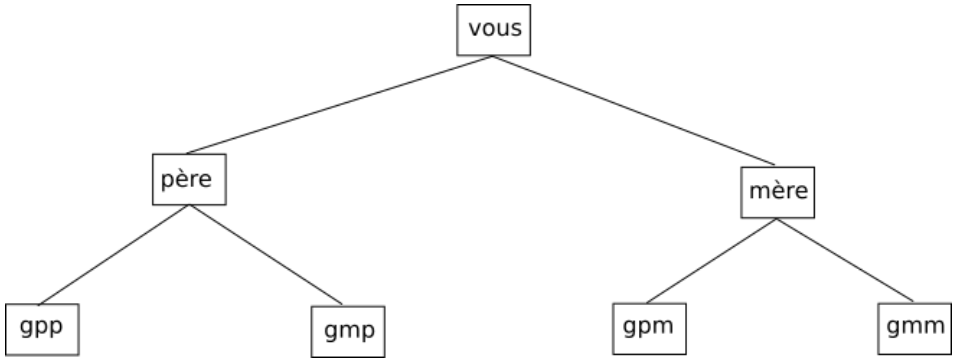
Pourtant, un simple graphique aurait grandement simplifié les choses :



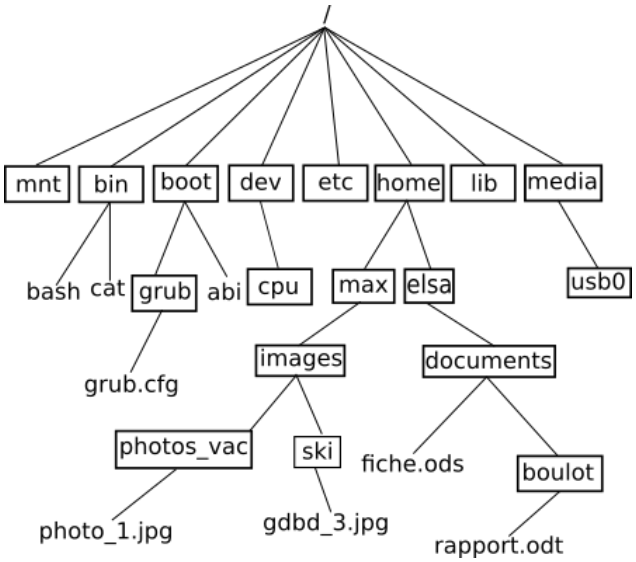
Poule 1 => 1er Eq1 ; 2e Eq8
Poule 2 => 1er Eq2 ; 2e Eq7
Poule 3 => 1er Eq3 ; 2e Eq6
Poule 4 => 1er Eq4 ; 2e Eq5

Les spectateurs peuvent alors recopier sur un bout de papier ce schéma et ensuite se livrer au jeu des pronostics.

Nous avons ci-dessous ce que l'on appelle une structure en arbre. On peut aussi retrouver cette même structure dans un arbre généalogique :



Dernier exemple, les systèmes de fichiers dans les systèmes de type UNIX ont aussi une structure en arbre (notion vue l'année dernière)



Système de fichiers de type UNIX

Définition :

Un arbre est une structure de données constituée de **nœuds**.

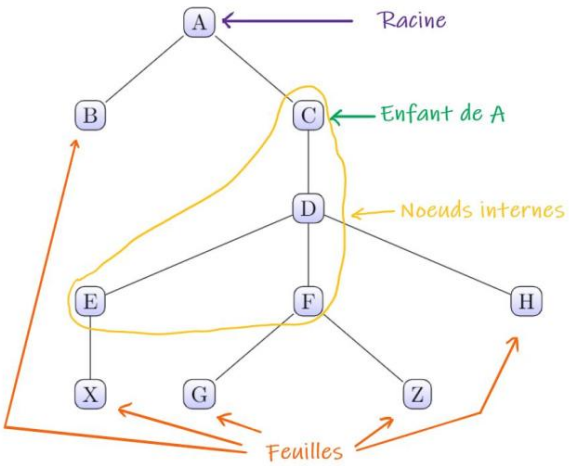
Le sommet de l'arbre s'appelle la **racine**.

Le nœud B situé sous un nœud A est appelé **enfant** du nœud A

Un nœud qui ne possède pas d'enfant est appelé **feuille**.

Les nœuds autre que la racine et les feuilles sont appelés **nœuds internes**.

Une **branche** est une suite de nœuds consécutifs de la racine vers une feuille.



Dans cet arbre:

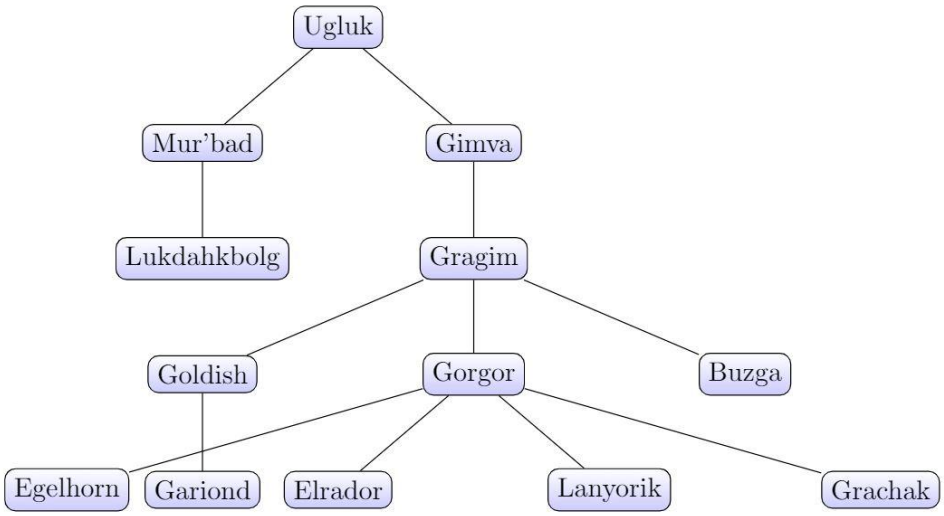
- il y a 10 nœuds.
- Le nœud A est la racine.
- Il y a 5 feuilles donc 5 branches.
- Le nœud D est l'enfant du nœud C.
- Il y a 4 nœuds internes.

Les arbres sont des types abstraits très utilisés en informatique. On les utilise notamment quand on a besoin d'une structure hiérarchique des données : dans l'exemple ci-dessous le fichier grub.cfg ne se trouve pas au même niveau 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).

On trouve souvent dans cette hiérarchie une notion de temps (les quarts de finale sont avant les demi-finales ou encore votre grand-mère paternelle est née avant votre père), mais ce n'est pas une obligation (voir l'arborescence du système de fichiers).

Activité 1

On donne l'arbre suivant :



1. Quelle est la racine de cet arbre ?
2. Combien y-a-t-il de nœuds ?
3. Combien y-a-t-il de feuilles ?
4. Combien y-a-t-il de branches ?

5. L'ensemble des nœuds internes compte combien d'éléments ?

6. Quels sont les enfants de Gragim ?

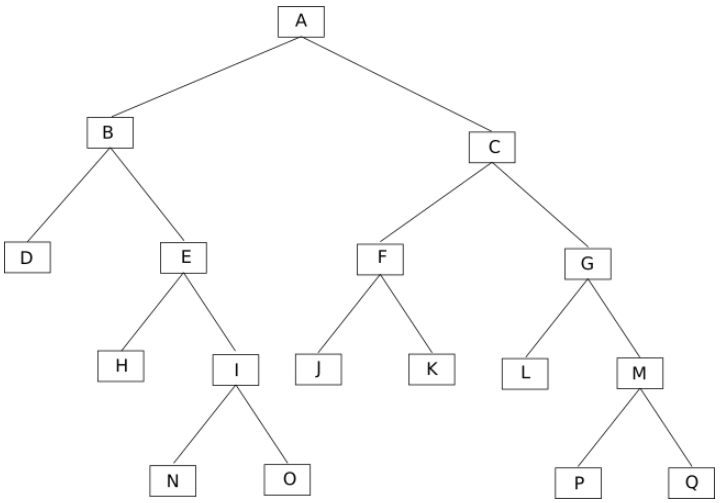
II- LES ARBRES BINAIRES

a) INTRODUCTION

Les arbres binaires sont des cas particuliers d'arbre : l'arbre du tournoi de rugby et l'arbre "père, mère..." sont des arbres binaires, en revanche, l'arbre représentant la structure du système de fichier n'est pas un arbre binaire. Dans un arbre binaire, on a au maximum 2 **arête** qui partent d'un élément (pour le système de fichiers, on a 7 branches qui partent de la racine : ce n'est donc pas un arbre binaire).

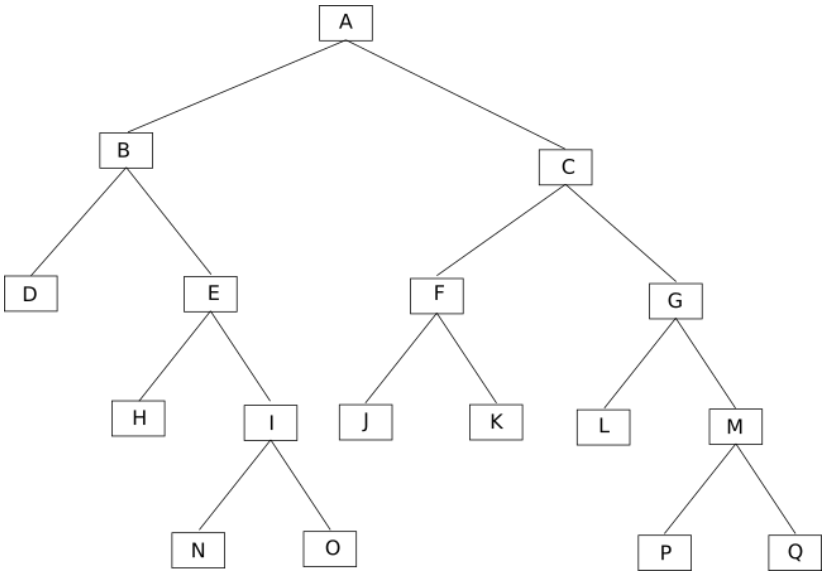
Dans la suite nous allons uniquement travailler sur les arbres binaires.

Soit l'arbre binaire suivant :



Chaque élément de l'arbre est appelé nœud (par exemple : A, B, C, D,...,P et Q sont des nœuds)

b) UN PEU DE VOCABULAIRE



- le nœud initial (A) est appelé **nœud racine** ou plus simplement **racine**
- On dira que le nœud E et le nœud D sont **les fils** du nœud B. 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)
- On appelle **arête** le segment qui relie 2 nœuds.
- On appelle **taille** d'un arbre le nombre de nœuds présents dans cet arbre
- On appelle **profondeur** d'un nœud ou d'une feuille dans un arbre binaire 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

  ATTENTION : on trouve aussi dans certains livres la profondeur de la racine égale à 0 (on trouve alors : profondeur de B = 1 ; profondeur de I = 3 ; profondeur de P = 4). Les 2 définitions sont valables, il faut juste préciser si vous considérez que la profondeur de la racine est de 1 ou de 0.

- On appelle **hauteur** d'un arbre la profondeur maximale des nœuds de l'arbre.

  Exemple : la profondeur de P = 5, c'est un des nœuds les plus profonds, donc la hauteur de l'arbre est de 5.

  ATTENTION : comme on trouve 2 définitions pour la profondeur, on peut trouver 2 résultats

différents pour la hauteur : si on considère la profondeur de la racine égale à 1, on aura bien une hauteur de 5, mais si l'on considère que la profondeur de la racine est de 0, on aura alors une hauteur de 4.

!!! Ainsi un *arbre vide* a pour hauteur 0.

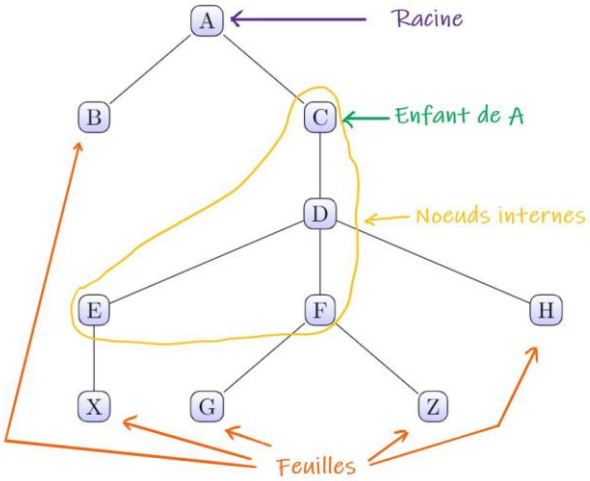
Définition	Hauteur de l'arbre avec un nœud	Hauteur de l'arbre vide
Par le nombre d'arêtes	0	-1
Par le nombre de nœuds	1	0

On peut caractériser un arbre par différentes caractéristiques :

- Son arité : le nombre maximal d'enfants qu'un nœud peut avoir.

Exemple :

On reprend l'arbre de la définition d'un arbre :



- L'arité de cet arbre est 3.
- La taille de cet arbre est 10.
- La hauteur de cet arbre est 4.

Remarque : On appelle **arbre binaire** un arbre d'arité inférieure ou égale à 2.

c) STRUCTURE RECURSIVE

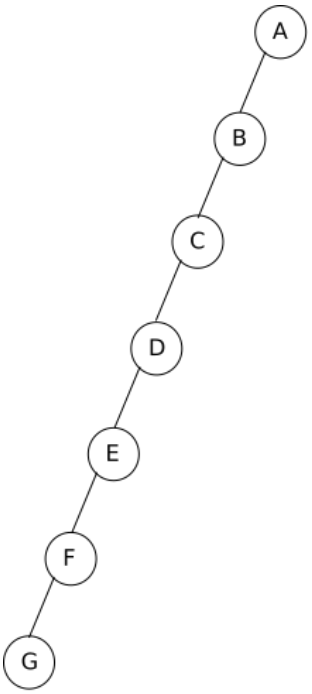
Il est aussi important de bien noter que l'on peut aussi voir les arbres comme des structures récursives : les fils d'un nœud sont des arbres (sous-arbre gauche et un sous-arbre droite dans le cas d'un arbre binaire), ces arbres sont eux-mêmes constitués d'arbres.

On peut alors définir un **arbre binaire** de façon *récursive* :

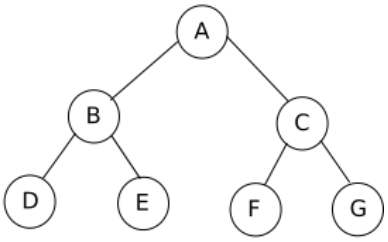
- soit c'est **l'arbre vide** s'il ne contient aucun noeud
- soit il est constitué d'un noeud spécial appelé **racine** de l'arbre dont le *fils droit* et le *fils gauche* sont des **arbres binaires**

d) ENCADREMENT DE LA HAUTEUR D'UN ARBRE

Il est possible d'avoir des arbres binaires de même taille mais de "forme" très différente :



Arbre 1



Arbre 2

Sur le schéma ci-dessus l'arbre 1 est dit " **filiforme**" alors que l'arbre 2 est dit "**complet**" (on dira qu'un arbre binaire est complet si tous les nœuds possèdent 2 fils et que toutes les feuilles se situent à la même profondeur). On pourra aussi dire que l'arbre 1 est **déséquilibré** alors que l'arbre 2 est équilibré.

Si on prend un arbre filiforme de taille n, on peut dire que la hauteur de cet arbre est égale à $n-1$ (si on prend la définition de la hauteur d'un arbre où la racine a une profondeur 0)
Si on prend un arbre complet de taille n, on peut démontrer que la hauteur de cet arbre est égale à la partie entière de $log_2(n)$ (on arrondit à l'entier immédiatement inférieur le $log_2(n)$)). Dans le cas de l'arbre 2,
Nous avons $log_2(7) = 2{,}8$ donc en prenant la partie entière on a bien la hauteur de l'arbre 2 égale à 2.

Un arbre filiforme et un arbre complet étant deux cas extrêmes, on peut affirmer que pour un arbre binaire quelconque

$$\lfloor log_2(n) \rfloor \leq h \leq n-1$$

Avec n la taille de l'arbre et h la hauteur de l'arbre ($\lfloor log_2(n) \rfloor$ permet de prendre la partie entière du logarithme base 2 de n)

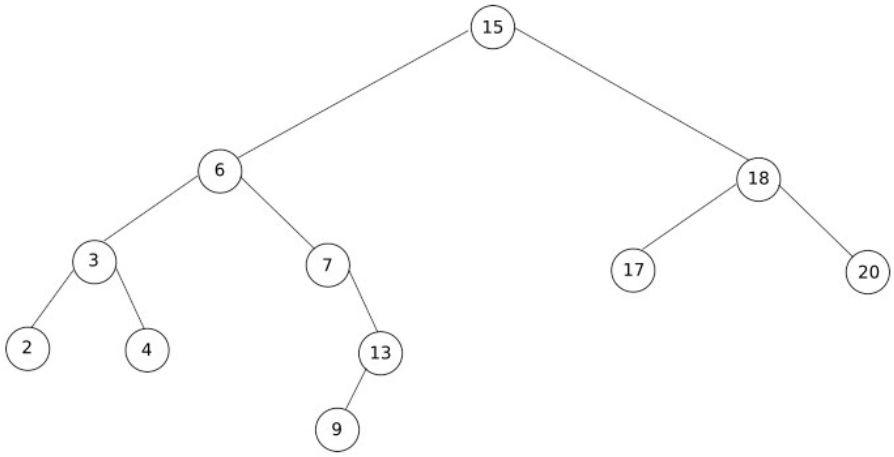
III- les arbres binaires de recherche

Un arbre binaire de recherche est un cas particulier d'arbre binaire.

Un **arbre binaire de recherche** est un arbre binaire dans lequel l'étiquette d'un nœud est appelé **clé** et est un entier. L'arbre binaire vérifie des propriétés suivantes :

- Les clés de tous les nœuds du sous-arbre gauche d'un nœud x sont inférieures ou égales à la clé de x.
- il faut que les clés de nœuds composant l'arbre soient ordonnables (on doit pouvoir classer les nœuds, par exemple, de la plus petite clé à la plus grande)
- soit x un nœud d'un arbre binaire de recherche. Si y est un nœud du sous-arbre gauche de x, alors il faut que **y.clé ⩽ x.clé**. Si y est un nœud du sous-arbre droit de x, il faut alors que **x.clé ⩽ y.clé**
- Les clés de tous les nœuds du sous-arbre droit d'un nœud x sont strictement supérieures à la clé de x.

Exemple d'arbre binaire de recherche :

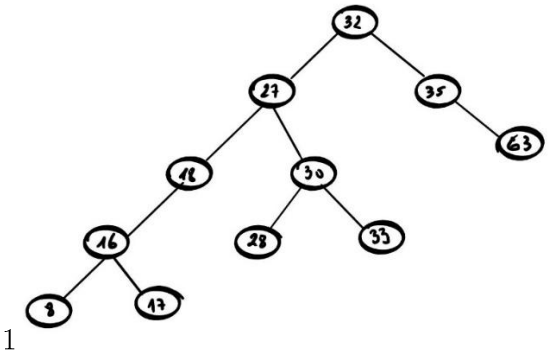


Vous pouvez vérifier que le fils gauche d'un nœud a une valeur plus petite que son père (par exemple $3 < 6$) et que le fils droit d'un nœud a une valeur plus grande que son père (par exemple $7 > 6$)

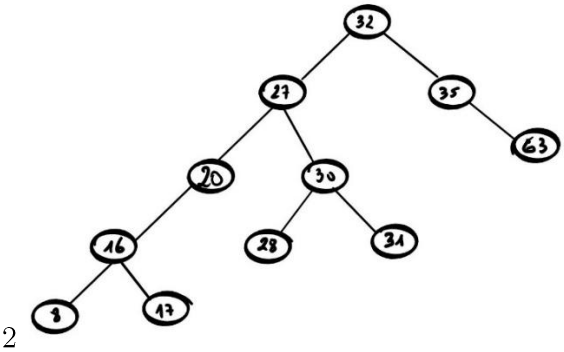
Attention : pour un nœud donné A, tous les nœuds de l'arbre gauche de A auront des valeurs plus petites que la valeur du nœud A et tous les nœuds de l'arbre droit de A auront des valeurs plus grandes que la valeur du nœud A.

Exemple :

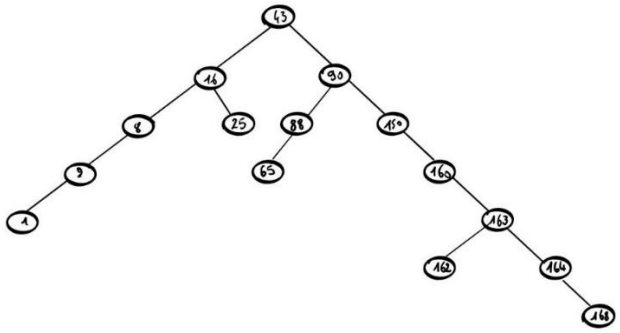
Quels sont parmi les arbres proposés ci-dessous les arbres binaires de recherche?



1



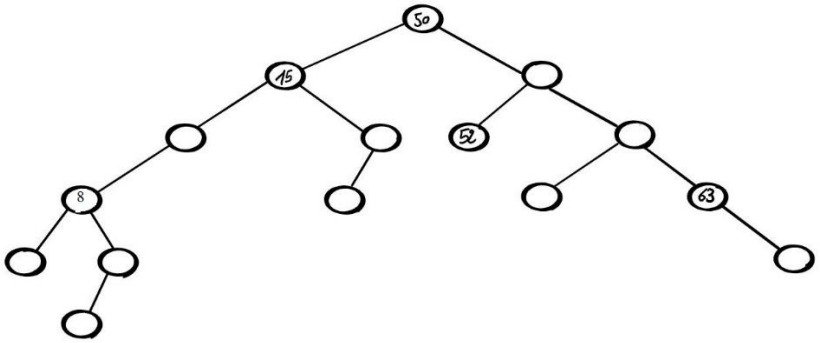
2



3

Exercice

Compléter cet arbre pour que ce soit un arbre binaire de recherche :

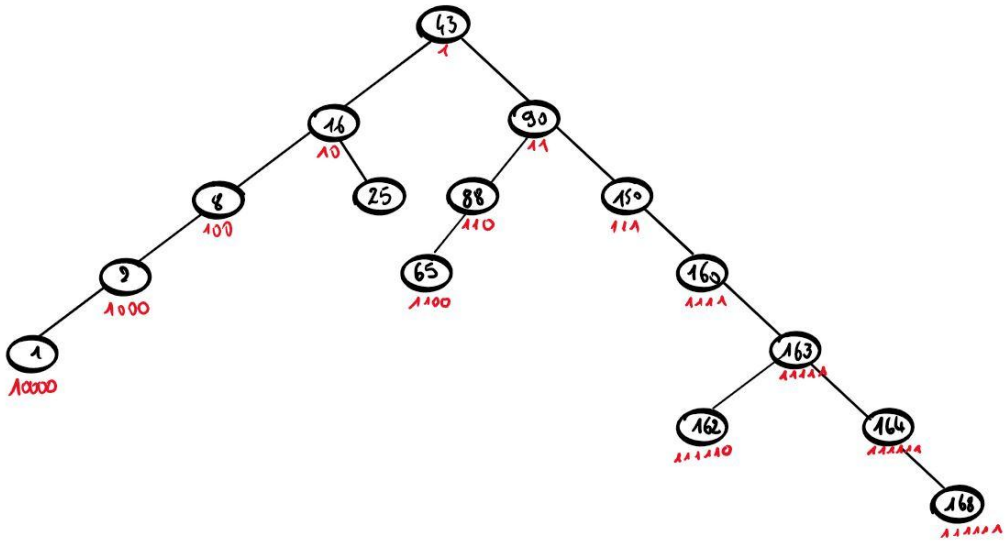


Propriété :

Un **étiquetage intéressant** d'un arbre de recherche est le suivant :

1. La racine est étiquetée 1.
2. La premier nœud du sous arbre gauche prend l'étiquette de son père auquel on ajoute un 0.
3. La premier nœud du sous arbre droit prend l'étiquette de son père auquel on ajoute un 1.

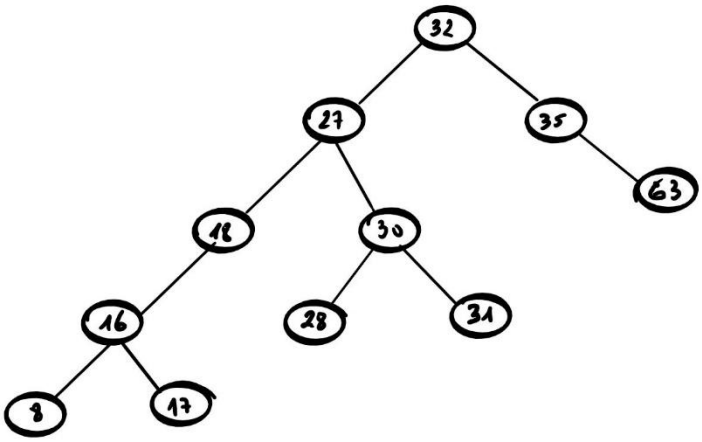
Exemple :



Quelle est l'étiquette de 25 dans l'arbre précédent ?

Exercice :

Etiqueter comme l'exemple précédent l'arbre suivant :



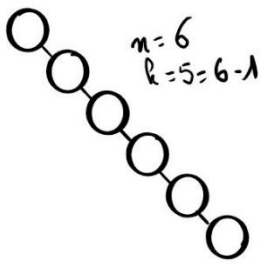
Propriété :

Pour un arbre binaire de recherche de hauteur h et de taille n les inégalités : $\lfloor log_2(n) \rfloor \leq h \leq n-1$

Pour démontrer ce résultat nous allons observer le cas où l'arbre est le plus profond et le cas où l'arbre est le moins profond.

- Cas où l'arbre est le plus profond :

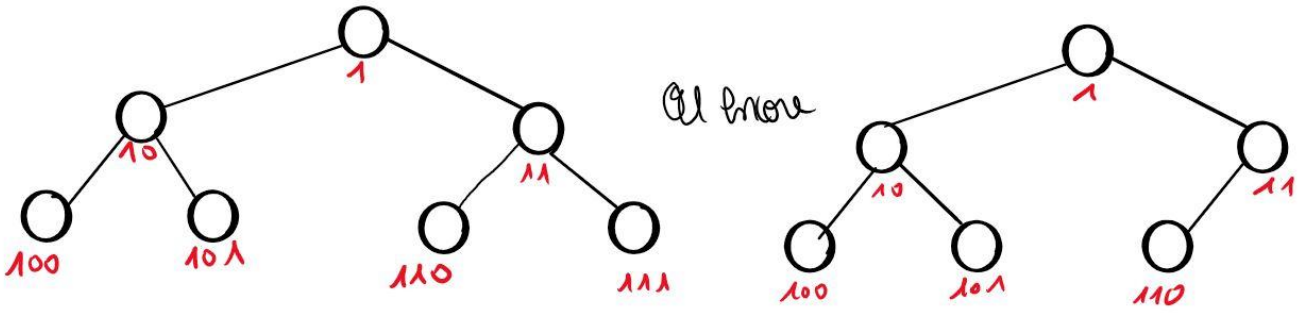
Pour une taille n fixée, la hauteur maximale d'un arbre binaire est $h = n - 1$, qu'on obtient avec des arbres « filiformes » comme cet arbre :



Ainsi $h \leq n - 1$

- Cas où l'arbre est le moins profond

Les arbres de taille n de hauteur minimale sont les arbres comme ces arbres :



Les clés des feuilles sont supérieures ou égales en binaire à 100...0100...0 où 0 est répété p fois

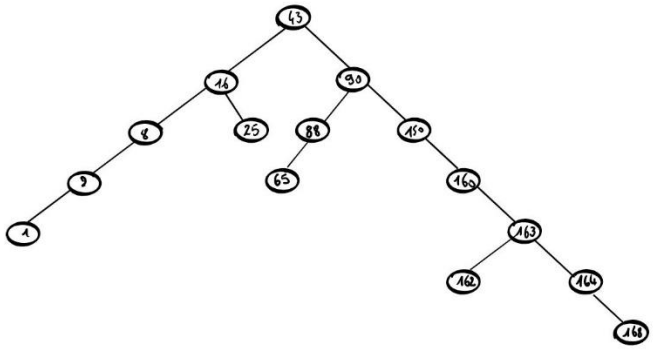
Dans ce cas la hauteur h est égale à p, il y a au moins $n=2^p$ nœuds

Or $log_2(2^p) = p$

Ainsi $log_2(n) \leq p$

Exemple :

Vérifions cette inégalité sur cet arbre :



La taille de cet arbre est 15.

La hauteur est 6. $log_2(15) = 3{,}9$ à 10^{-1} près

$\lfloor 3{,}9 \rfloor = 3$

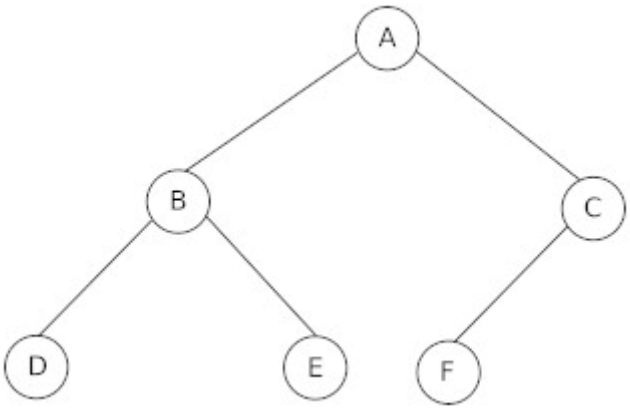
On a bien : $3 \leq 6 \leq 14$

Activité 1

Trouvez un autre exemple de données qui peuvent être représentées par un arbre binaire (dans le domaine de votre choix). Dessinez au moins une partie de cet arbre binaire. Déterminez la hauteur et la taille de l'arbre que vous aurez dessiné.

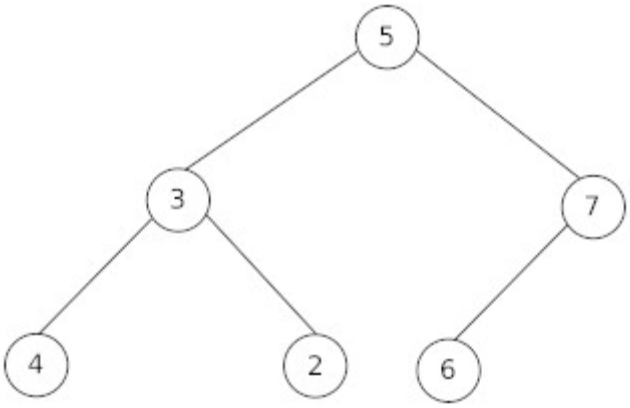
Activité 2

Soit l'arbre binaire suivant :



1. Cet arbre est-il un arbre binaire ? Justifiez votre réponse.
2. Donnez la clé (valeur) de la racine de cet arbre.
3. Quels sont les fils du nœud B.
4. Donnez l'arbre droit du nœud A.
5. Le nœud C est-il une feuille ? Justifiez votre réponse.
6. Donnez la taille de cet arbre.
7. Donnez la profondeur du nœud B (on prendra la profondeur de la racine égale à 0).
8. Donnez la hauteur de cet arbre (on prendra la profondeur de la racine égale à 0).

Activité 3



1. Expliquez pourquoi cet arbre binaire n'est pas un arbre binaire de recherche.
2. Modifiez cet arbre pour le transformer en arbre binaire de recherche.

Activité 4

Soit les valeurs suivantes : 14, 22, 8, 47, 42, 13, 1, 24, 33, 74.

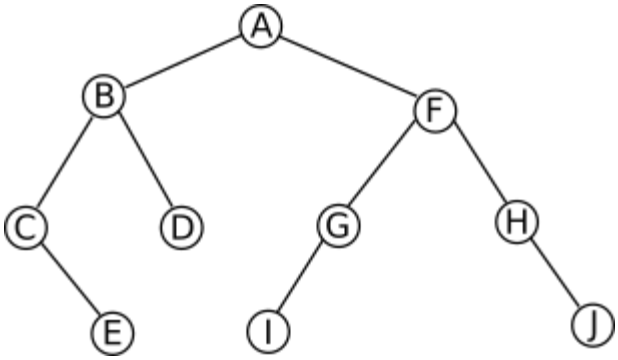
Construisez un arbre binaire de recherche à partir de ces valeurs.

I- NOTATIONS UTILISEES

Avant d'entrer dans le vif du sujet, nous allons un peu approfondir la notion d'arbre binaire :

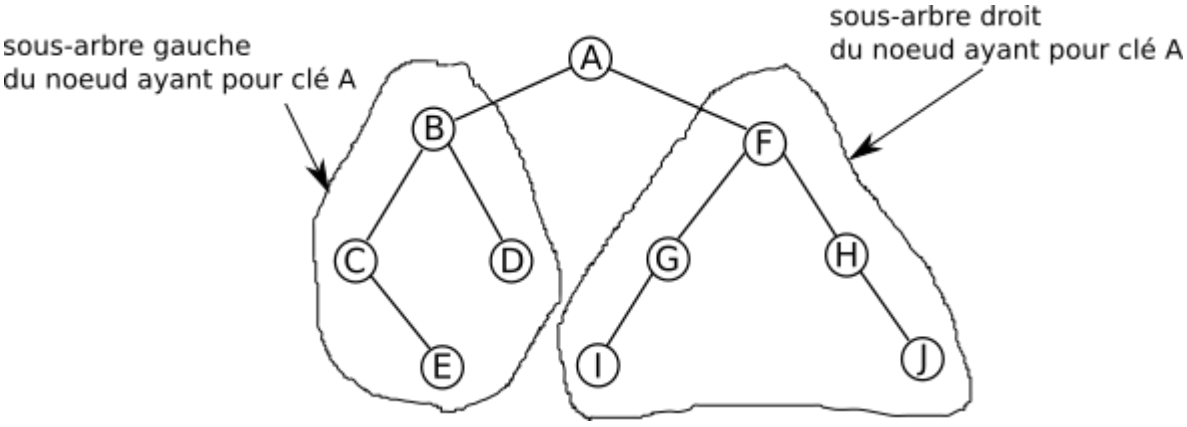
À chaque nœud d'un arbre binaire, on associe une clé ("valeur" associée au nœud on peut aussi utiliser le terme "valeur" à la place de clé), un "sous-arbre gauche" et un "sous-arbre droit"

Soit l'arbre binaire suivant :



si on prend le noeud ayant pour clé A (le noeud racine de l'arbre) on a :

- le sous-arbre gauche est composé du noeud ayant pour clé B, du noeud ayant pour clé C, du noeud ayant pour clé D et du noeud ayant pour clé E
- le sous-arbre droit est composé du noeud ayant pour clé F, du noeud ayant pour clé G, du noeud ayant pour clé H, du noeud ayant pour clé I et du noeud ayant pour clé J



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

- le sous-arbre gauche est composé du nœud ayant pour clé C et du nœud ayant pour clé E
- le sous-arbre droit est uniquement composé du nœud ayant pour clé D

Un arbre (ou un sous-arbre) vide est noté NIL ou NONE (NIL est une 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 (NONE)

Il faut bien avoir en tête qu'un sous-arbre (droite ou gauche) est un arbre (même s'il contient un seul nœud ou pas de nœud de tout (NONE)).

Soit un arbre T : T.racine correspond au nœud racine de l'arbre T

Soit un nœud x :

- x.gauche correspond au sous-arbre gauche du nœud x
- x.droit correspond au sous-arbre droit du nœud x
- x.clé correspond à la clé du nœud x

Il faut noter que si le nœud x est une feuille, x.gauche et x.droite sont des arbres vides (NONE)

A retenir

Un noeud est constitué d'un élément, d'un fils gauche et d'un fils droit donc on peut représenter un noeud par un objet d'une classe Noeud avec trois attributs : element, gauche et droite.

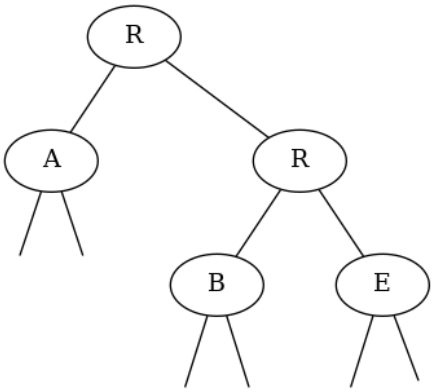
On choisit de représenter l'absence de noeud par la valeur spéciale None

```
class Noeud:

    """Noeud pour arbre binaire"""

    def __init__(self, g, e, d):
        self.gauche = g       # lien vers fils gauche g éventuellement vide
(None)
        self.element = e     # élément e stocké dans le noeud
        self.droit = d        # lien vers fils droit d éventuellement vide
(None)
```

Cette classe Noeud permet déjà de construire des arbres binaires. Par exemple l'expression :

```
Noeud(Noeud(None, 'A', None), 'R', Noeud(Noeud(None, 'B', None), 'R', Noeud(None, 'E', None)))
```

permet de construire l'arbre binaire ci-dessous :



Exercice 1

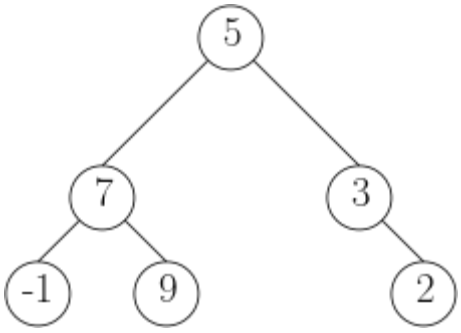
1. Représenter le noeud construit par l'instruction suivante :

```
ac = Noeud(None, 'C', None)
```

2. On complète le code précédent. Représenter les deux arbres binaires construits par la séquence d'instructions :

```
ab = Noeud(ac, 'B', None)
aa = Noeud(None, 'A', ab)
```

3. Donner une séquence d'instructions permettant de construire avec la classe Noeud l'arbre binaire ci-dessous.



II- INTERFACE ET IMPLEMENTATION D'ARBRE BINAIRE IMMUABLE

Pour le type abstrait arbre binaire on a besoin de représenter :

- l'arbre binaire vide : on peut choisir une valeur spéciale comme None
- un noeud par exemple avec un objet de la classe Noeud définie précédemment.

On peut alors définir un ensemble de fonctions constituant une interface fonctionnelle minimale pour le type abstrait arbre binaire. Le paramètre arbre désigne un arbre qui est de type Noeud s'il est non vide ou qui vaut None sinon.

Fonction	Signature	Description
arbre_vide	`arbre_vide()`	Renvoie un arbre vide représenté par `None`
est_vide	`est_vide(arbre)`	Renvoie un booléen indiquant si `arbre` est vide
element_racine	`element_racine(arbre)`	Renvoie l'élément à la racine de l'arbre s'il est non vide
gauche	`gauche(arbre)`	Renvoie le sous-arbre fils gauche de l'arbre s'il est non vide
droit	`droit(arbre)`	Renvoie le sous-arbre fils droit de l'arbre s'il est non vide
creer_arbre	`creer_arbre(g, e, d)`	Renvoie un noeud constitué de l'élément `e`, du sous-arbre gauche `g` et du sous-arbre droit `d`

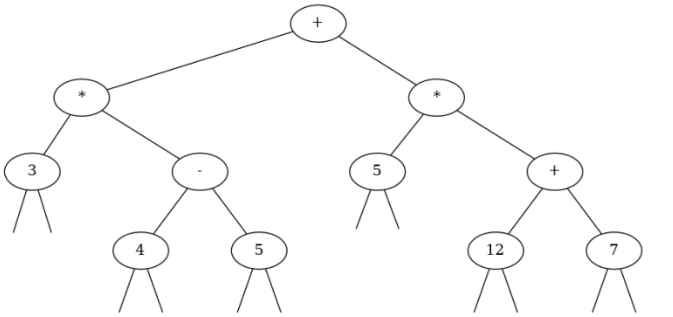
A partir de cette interface fonctionnelle, on peut donner une implémentation d'**arbre binaire immuable** c'est-à-dire que chaque fonction renvoie un nouvel arbre binaire et ne modifie jamais l'arbre binaire passé en paramètre :

```
def arbre_vide():
    """Renvoie un arbre vide représenté par None"""
    return None

def est_vide(arbre):
    """Teste si un arbre est vide, renvoie un booléen"""
    return arbre is None

def gauche(arbre):
    """Renvoie le sous-arbre fils gauche de arbre
    Provoque une erreur si arbre est vide"""
    assert not est_vide(arbre), "Arbre vide"
    return arbre.gauche

def droit(arbre):
    """Renvoie le sous-arbre fils droit de arbre
    Provoque une erreur si arbre est vide"""
    assert not est_vide(arbre), "Arbre vide"
    return arbre.droit

def element_racine(arbre):
    """Renvoie l'élément à la racine de arbre
    Provoque une erreur si arbre est vide"""
    assert not est_vide(arbre), "Arbre vide"
    return arbre.element

def creer_arbre(g, e, d):
    """Construit et renvoie l'arbre binaire dont la racine est constituée
    par le noeud d'élément e, g sous-arbre gauche et d sous-arbre droit"""
    return Noeud(g, e, d)

# extension de l'interface pour afficher un arbre
def afficher_arbre(arbre):
    """Affichage syntaxiquement correct d'un arbre :
    vide ou avec le constructeur de  Noeud"""
    if arbre is None:
        return repr(None)
    return      f"Noeud({afficher_arbre(arbre.gauche)},      {repr(arbre.element)},
{afficher_arbre(arbre.droit)})"
```

Exercice

Représenter l'arbre binaire construit dans la variable a à la fin de la séquence d'instructions ci-dessous :

```
sag = creer_arbre(None, '4', None)
sad = creer_arbre(None, '5', None)
sad = creer_arbre(sag, '-', sad)
sag = creer_arbre(None, '3', None)
sag = creer_arbre(sag, '*', sad)

sag2 = creer_arbre(None, '12', None)
sad2 = creer_arbre(None, '7', None)
sad2 = creer_arbre(sag2, '+', sad2)
sag2 = creer_arbre(None, '5', None)
sad2 = creer_arbre(sag2, '*', sad2)

a = creer_arbre(sag, '+', sad2)
```



III- CALCULER LA HAUTEUR D'UN ARBRE

Nous allons commencer à travailler sur les algorithmes en nous intéressant à l'algorithme qui permet de calculer la hauteur d'un arbre :

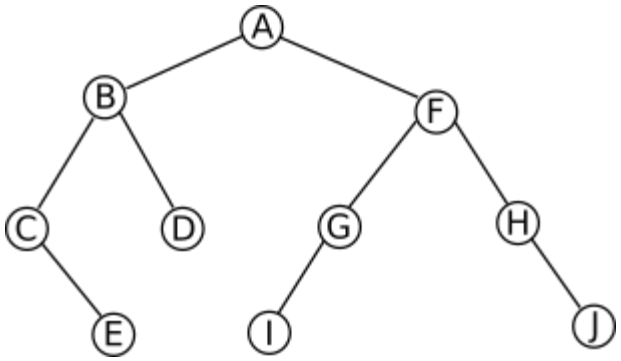
Activité 1

Étudiez cet algorithme :

```
VARIABLE
T : arbre
x : nœud

DEBUT
HAUTEUR(T) :
  si T ≠ NONE :
    x ← T.racine
    renvoyer 1 + max(HAUTEUR(x.gauche), HAUTEUR(x.droit))
  sinon :
    renvoyer 0
  fin si
FIN
```

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

Cet algorithme est loin d'être simple, n'hésitez pas à écrire votre raisonnement sur une feuille de brouillon. Vous pourrez par exemple essayer d'appliquer cet algorithme sur l'arbre binaire ci-dessous. N'hésitez pas à poser des questions si nécessaires.



Comme vous l'avez sans doute remarqué, nous avons dans l'algorithme ci-dessus une fonction récursive. Vous aurez l'occasion de constater que c'est souvent le cas dans les algorithmes qui travaillent sur des structures de données telles que les arbres.

IV- CALCULER LA TAILLE D'UN ARBRE

Nous allons maintenant étudier un algorithme qui permet de calculer le nombre de nœuds présents dans un arbre.

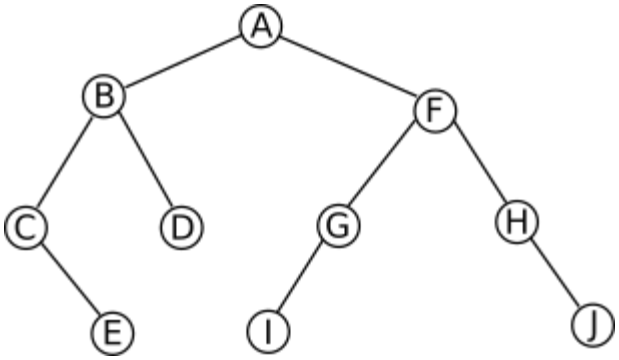
Activité 2

Étudiez cet algorithme :

```
VARIABLE
T : arbre
x : noeud

DEBUT
TAILLE(T) :
  si T ≠ NONE :
    x ← T.racine
    renvoyer 1 + TAILLE(x.gauche) + TAILLE(x.droit)
  sinon :
    renvoyer 0
  fin si
FIN
```

Cet algorithme ressemble beaucoup à l'algorithme étudié au "À faire vous-même 1", son étude ne devrait donc pas vous poser de problème.

Appliquez cet algorithme à l'exemple suivant :



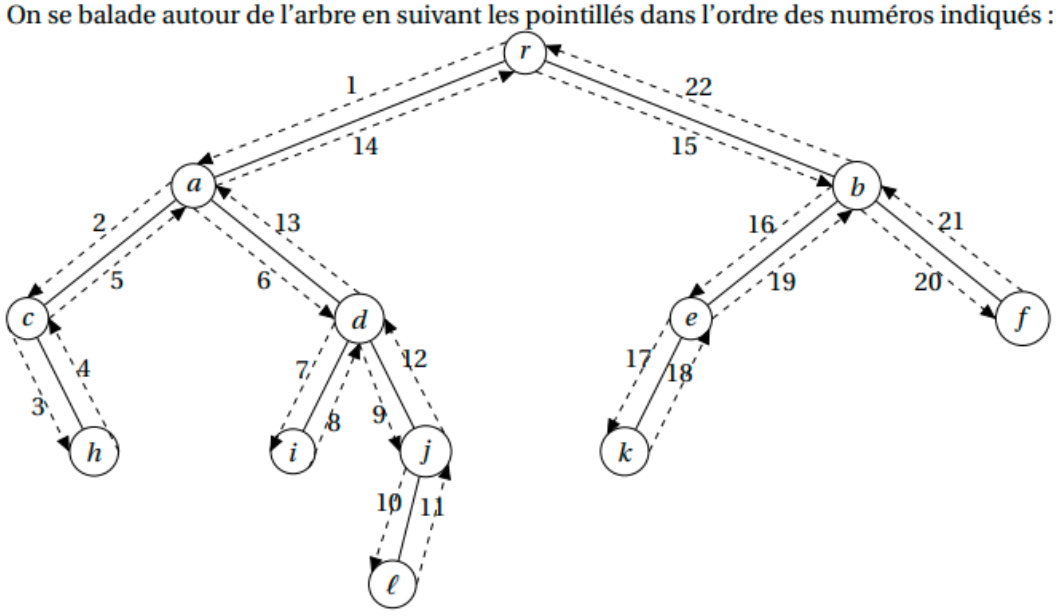
Il existe plusieurs façons de parcourir un arbre (parcourir un arbre = passer par tous les nœuds), nous allons en étudier quelques-unes :

V- PARCOURIR UN ARBRE

a) Introduction

Il existe plusieurs façons de parcourir un arbre (parcourir un arbre = passer par tous les nœuds), nous allons en étudier quelques-unes. Le choix du parcours dépend du problème à traiter

On se balade autour de l'arbre en suivant les pointillés dans l'ordre des numéros indiqués :



A partir de ce contour, on définit trois parcours des sommets de l'arbre :

1. l'ordre préfixe : on liste chaque sommet la première fois qu'on le rencontre dans la balade.

2. l'ordre suffixe : on liste chaque sommet la dernière fois qu'on le rencontre.

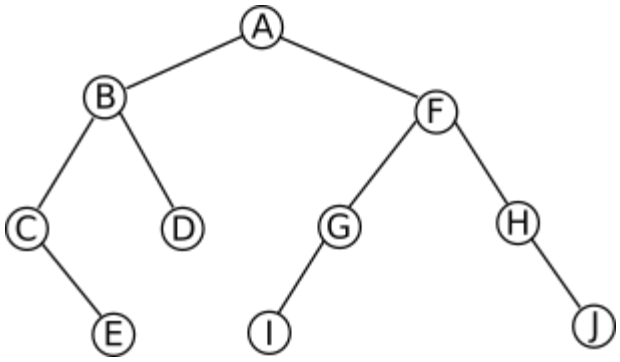
3. l'ordre infixe : on liste chaque sommet ayant un fils gauche la seconde fois qu'on le voit et chaque sommet sans fils gauche la première fois qu'on le voit.

b) Parcourir un arbre dans l'ordre infixe

Voici l'algorithme qui va permettre de parcourir un arbre dans l'ordre infixe :

```
VARIABLE
T : arbre
x : noeud
DEBUT
PARCOURS-INFIXE(T) :
  si T ≠ NONE :
    x ← T.racine
    PARCOURS-INFIXE(x.gauche)
    affiche x.clé
    PARCOURS-INFIXE(x.droit)
  fin si
FIN
```

Dans le cas du parcours infixe, pour un nœud A donné, on parcourra le sous-arbre gauche de A, puis on affichera la clé de A puis enfin, on parcourra le sous-arbre droite de A



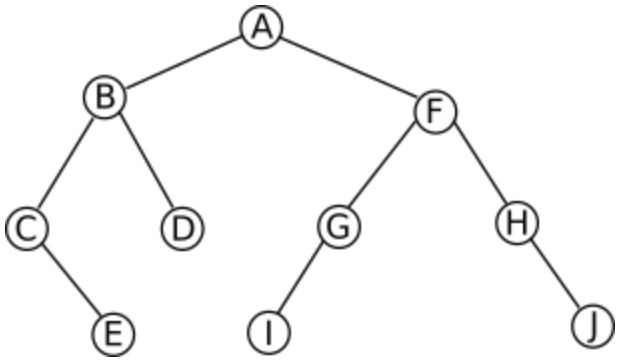
c) Parcourir un arbre dans l'ordre préfixe

Activité 3

Étudiez cet algorithme :

```
VARIABLE
T : arbre
x : noeud

DEBUT
PARCOURS-PREFIXE(T) :
  si T ≠ NONE :
    x ← T.racine
    affiche x.clé
    PARCOURS-PREFIXE(x.gauche)
    PARCOURS-PREFIXE(x.droit)
  fin si
FIN
```

Vérifiez qu'en appliquant l'algorithme ci-dessus, l'arbre ci-dessous est bien parcouru dans l'ordre suivant : A, B, C, E, D, F, G, I, H, J



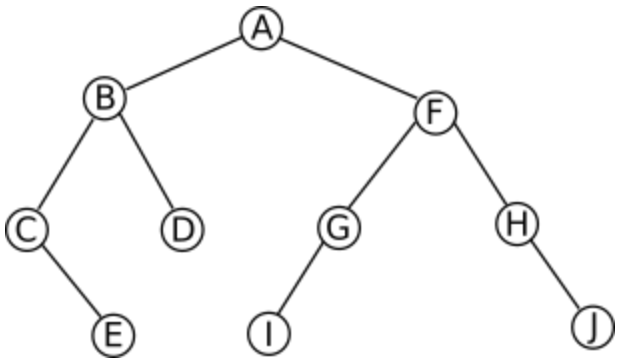
d) Parcourir un arbre dans l'ordre suffixe

Activité 4

Étudiez cet algorithme :

```
VARIABLE
T : arbre
x : nœud

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

Vérifiez qu'en appliquant l'algorithme ci-dessus, l'arbre ci-dessous est bien parcouru dans l'ordre suivant : E, C, D, B, I, G, J, H, F, A



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***

e) Parcourir un arbre en largeur d'abord

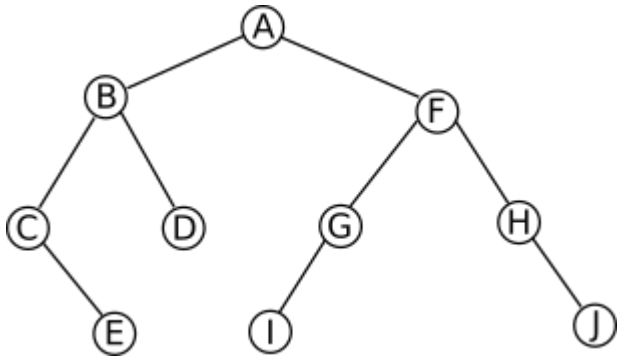
Activité 5

Étudiez cet algorithme :

```
VARIABLE
T : arbre
Tg : arbre
Td : arbre
x : nœud
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 ≠ NONE :
      Tg ← x.gauche
      enfiler(Tg.racine, f)
    fin si
    si x.droit ≠ NONE :
      Td ← x.droite
      enfiler(Td.racine, f)
    fin si
  fin tant que
FIN
```

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

Vérifiez qu'en appliquant l'algorithme ci-dessus, l'arbre ci-dessous est bien parcouru dans l'ordre suivant : A, B, F, C, D, G, H, E, I, J



Selon vous, pourquoi parle-t-on de parcours en largeur ?

Il est important de bien noter l'utilisation d'une file (FIFO) pour cet algorithme de parcours en largeur. Vous noterez aussi que cet algorithme n'utilise pas de fonction récursive.

Dans le cas d'un parcours en largeur d'abord on affiche tous les nœuds situés à une profondeur n avant de commencer à afficher les nœuds situés à une profondeur n+1.

NB : ***Parcours en profondeurs***

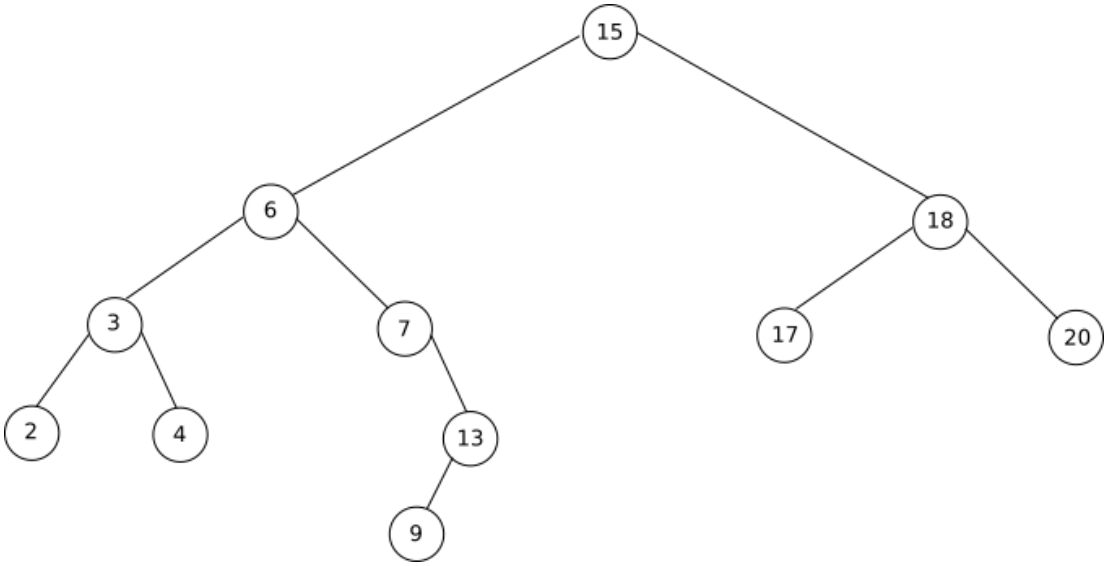
Les parcours en profondeur se définissent de manière récursive sur les arbres. Le parcours d'un arbre consiste à traiter la racine de l'arbre et à parcourir récursivement les sous-arbres gauche et droit de la racine. Les parcours préfixe, infixe et suffixe se distinguent par l'ordre dans lequel sont faits ces traitements.

VI- ARBRE BINAIRE DE RECHERCHE

Un arbre binaire de recherche est un cas particulier d'arbre binaire. Pour avoir un arbre binaire de recherche :

- il faut avoir un arbre binaire !
- il faut que les clés de nœuds composant l'arbre soient ordonnables (on doit pouvoir classer les nœuds, par exemple, de la plus petite clé à la plus grande)
- soit x un nœud d'un arbre binaire de recherche. Si y est un nœud du sous-arbre gauche de x, alors il faut que y.clé $\leqslant$ x.clé. Si y est un nœud du sous-arbre droit de x, il faut alors que x.clé $\leqslant$ y.clé

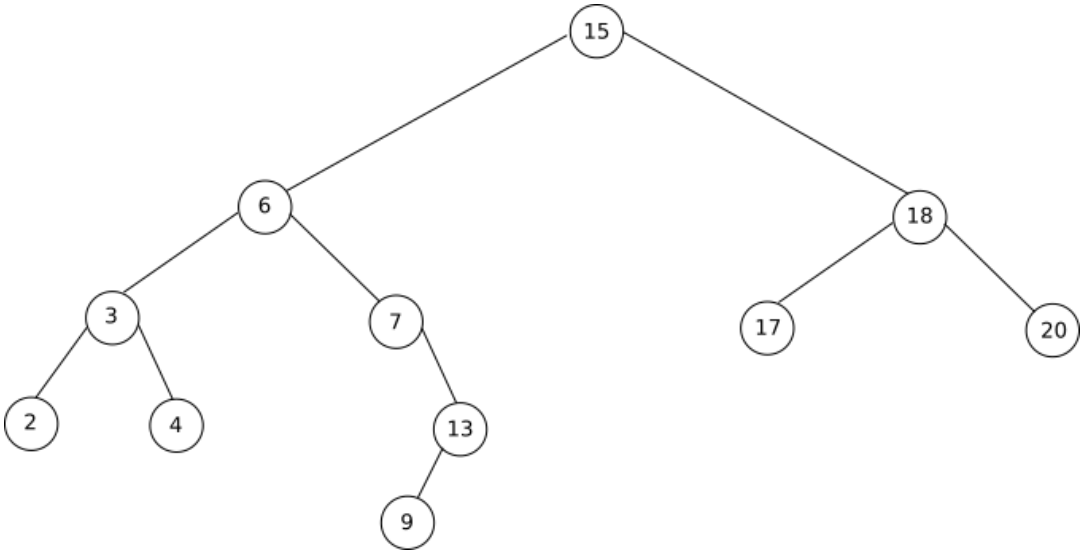
Activité 6



Vérifiez que l'arbre ci-dessus est bien un arbre binaire de recherche.

Activité 7

Appliquez l'algorithme de parcours infixe sur l'arbre ci-dessous :



Que remarquez-vous ?

a) 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.

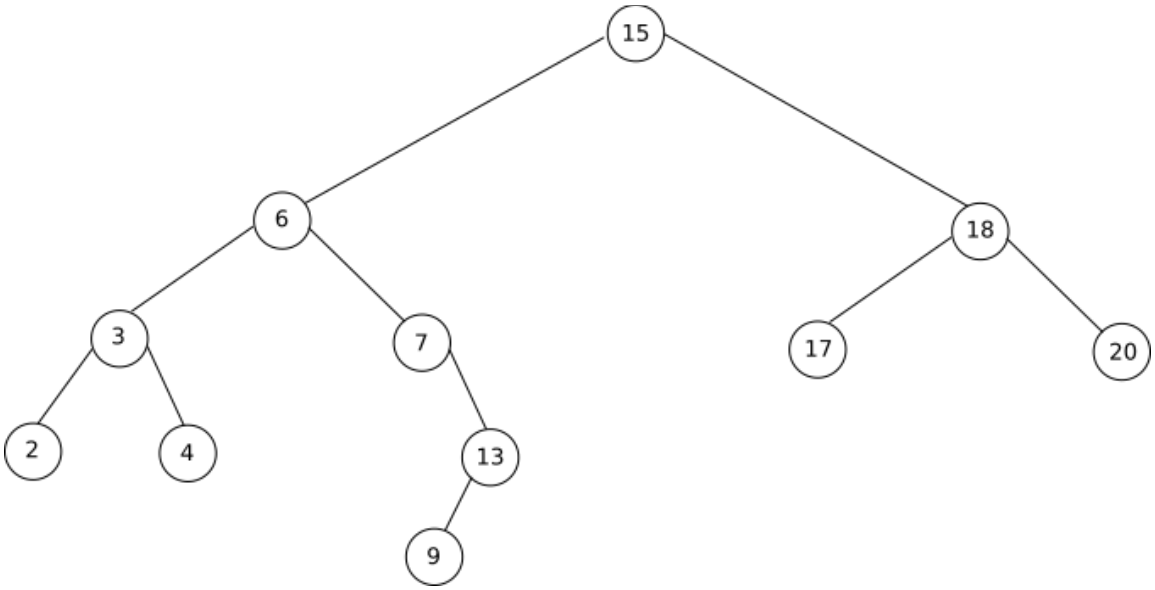
Activité 8

Étudiez l'algorithme suivant:

```
VARIABLE
T : arbre
Tg : arbre
Td : arbre
x : nœud
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 ≠ NONE :
      Tg ← x.gauche
      enfiler(Tg.racine, f)
    fin si
    si x.droit ≠ NONE :
      Td ← x.droite
      enfiler(Td.racine, f)
    fin si
  fin tant que
FIN
```

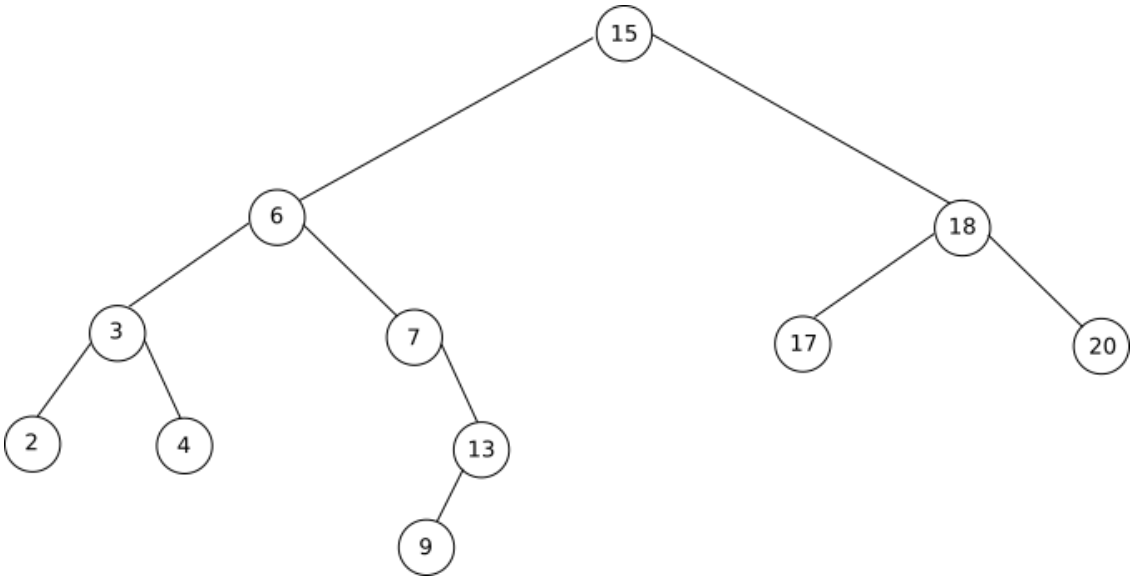
Activité 9

Appliquez l'algorithme de recherche d'une clé dans un arbre binaire de recherche sur l'arbre ci-dessous. On prendra k = 13.



Activité 10

Appliquez l'algorithme de recherche d'une clé dans un arbre binaire de recherche sur l'arbre ci-dessous. On prendra k = 16.



Cet algorithme de recherche d'une clé dans un arbre binaire de recherche ressemble beaucoup à la recherche dichotomique vue en première dans le cas où l'arbre binaire de recherche traité est équilibré. La complexité en temps dans le pire des cas de l'algorithme de recherche d'une clé dans un arbre binaire de recherche équilibré est donc ***O(log2(n))***. Dans le cas où l'arbre est filiforme, la complexité est ***O(n)***. Rappelons qu'un algorithme en ***O(log2(n))*** est plus "efficace" qu'un algorithme en ***O(n)***.

À noter qu'il existe une version dite "itérative" (qui n'est pas récursive) de cet algorithme de recherche :

Activité 11

Étudiez l'algorithme suivant:

```
VARIABLE
T : arbre
x : noeud
k : entier
DEBUT
ARBRE-RECHERCHE_ITE(T,k) :
  x ← T.racine
  tant que T ≠ NIL et k ≠ x.clé :
    x ← T.racine
    si k < x.clé :
      T ← x.gauche
    sinon :
      T ← x.droit
    fin si
```

```
  fin tant que
  si k == x.clé :
    renvoyer vrai
  sinon :
    renvoyer faux
  fin si
FIN
```

b) 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) :

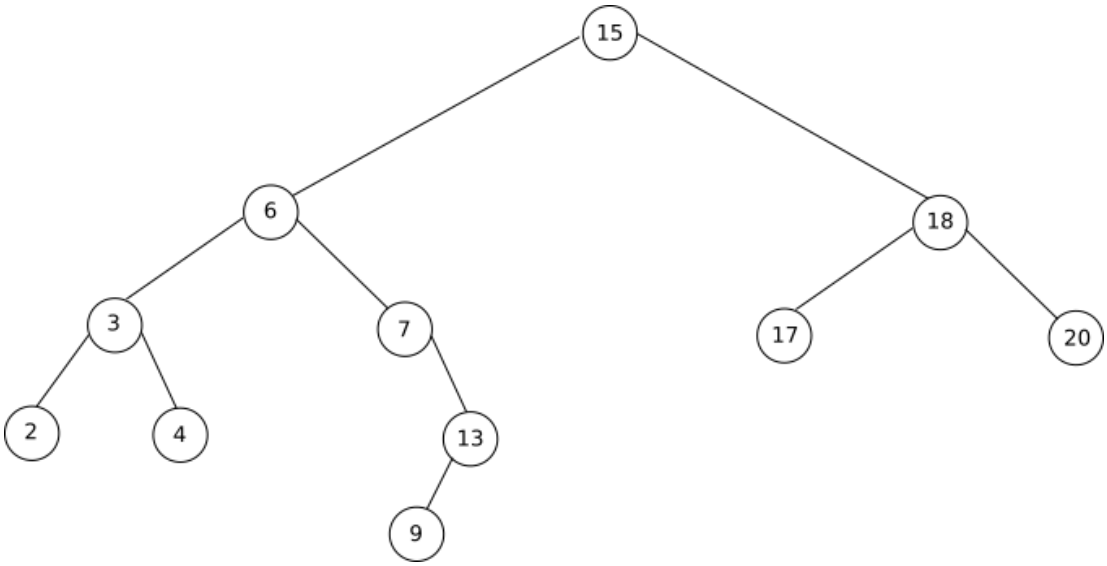
Activité 12

Étudiez l'algorithme suivant :

```
VARIABLE
T : arbre
x : nœud
y : nœud
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
```

Activité 13

Appliquez l'algorithme d'insertion d'un nœud y dans un arbre binaire de recherche sur l'arbre ci-dessous. On prendra y.clé = 16.



c) arbre binaire de recherche et parcours infixe

Il est important de noter qu'un parcours infixe d'un arbre binaire de recherche permet d'obtenir les valeurs des nœuds de l'arbre binaire de recherche dans l'ordre croissant.

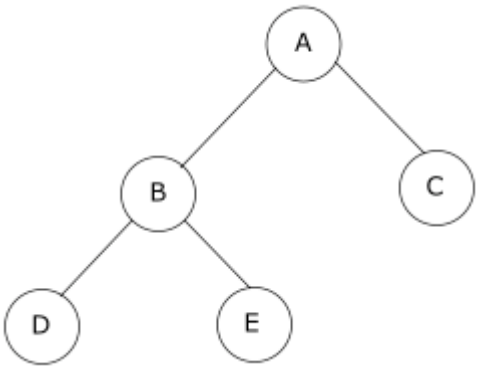
Activité 14

Dans cette activité, nous allons implémenter des arbres binaires en Python en utilisant des dictionnaires .

L'idée est relativement simple : chaque nœud est modélisé à l'aide d'un dictionnaire, ces dictionnaires seront composés de 3 clés (et donc 3 valeurs) : une clé "valeur", une clé "arbre_gauche" et une clé "arbre_droit".

La valeur associée à la clé "valeur" sera tout simplement la valeur du nœud. La valeur associé à la clé "arbre_gauche" sera un nœud (donc un autre dictionnaire) si l'arbre gauche existe et None dans le cas contraire. La valeur associée à la clé "arbre_droit" sera un nœud (donc un autre dictionnaire) si l'arbre droit existe et None dans le cas contraire.

L'arbre binaire suivant :



sera implémenté en Python avec le dictionnaire suivant :

```
arbre_1 = {"valeur" : "A", "arbre_gauche" : {"valeur" : "B", "arbre_gauche":
{"valeur" : "D", "arbre_gauche": None, "arbre_droit": None}, "arbre_droit":
{"valeur" : "E", "arbre_gauche": None, "arbre_droit": None}}, "arbre_droit" :
{"valeur" : "C", "arbre_gauche": None, "arbre_droit": None}}
```

Que l'on peut aussi représenter comme ceci afin d'améliorer la visibilité :

```
arbre_1 = {"valeur":"A",
           "arbre_gauche":
                          {"valeur" : "B",
                          "arbre_gauche":
                                         {"valeur" : "D",
                                         "arbre_gauche": None,
                                         "arbre_droit": None},
                          "arbre_droit": {"valeur" : "E",
                                         "arbre_gauche": None,
                                         "arbre_droit": None}},
           "arbre_droit" : {"valeur" : "C",
                          "arbre_gauche": None,
                          "arbre_droit": None}}
```

1. Écrire en Python une fonction taille qui prend en paramètre un arbre arb(arbre implémenté sous forme de dictionnaire) et qui renvoie la taille de l'arbre arb.

2. Écrire en Python une fonction hauteur qui prend en paramètre un arbre arb(arbre implémenté sous forme de dictionnaire) et qui renvoie la hauteur de l'arbre arb.

3. Écrire en Python une fonction parcours_prefixe qui prend en paramètre un arbre arb(arbre implémenté sous forme de dictionnaire) et qui affiche les nœuds de l'arbre arb dans l'ordre préfixe.

4. Écrire en Python une fonction parcours_infixe qui prend en paramètre un arbre arb(arbre implémenté sous forme de dictionnaire) et qui affiche les nœuds de l'arbre arb dans l'ordre infixe.

5. Écrire en Python une fonction parcours_suffixe qui prend en paramètre un arbre arb(arbre implémenté sous forme de dictionnaire) et qui affiche les nœuds de l'arbre arb dans l'ordre suffixe.

6. Écrire en Python une fonction parcours_largeur qui prend en paramètre un arbre arb(arbre implémenté sous forme de dictionnaire) et qui affiche les nœuds de l'arbre arb en respectant le parcours en largeur.

7. Implémenter en Python un arbre binaire de recherche 'arbre_2' constitué des nombres suivants : 30, 0, 10, 40 et 20

8. Écrire en Python une fonction récursive arbre_recherche_rec qui prend en paramètre un arbre binaire de recherche arb(arbre implémenté sous forme de dictionnaire) et un entier k et qui renvoie True si k est bien présent dans l'arbre et False dans le cas contraire.

9. Écrire en Python une fonction non récursive arbre_recherche_it qui prend en paramètre un arbre binaire de recherche arb(arbre implémenté sous forme de dictionnaire) et un entier k et qui renvoie True si k est bien présent dans l'arbre et False dans le cas contraire.

10. Écrire en Python une fonction non récursive insertion qui prend en paramètre un arbre binaire de recherche arb(arbre implémenté sous forme de dictionnaire) et un entier k et qui place l'entier k dans l'arbre binaire de recherche arb.

Ce qu'il faut savoir

- connaître l'algorithme qui permet de calculer la hauteur d'un arbre (voir cours)
- connaître l'algorithme qui permet de calculer la taille d'un arbre (voir cours)
- connaître les algorithmes qui permettent de parcourir un arbre : ordre infixe, ordre préfixe, ordre suffixe, en largeur d'abord (voir cours)
- connaître l'algorithme qui permet de rechercher une clé dans un arbre binaire de recherche (voir cours), savoir que cet algorithme à une complexité en $O(log_2(n))$ dans le cas d'un arbre binaire de recherche équilibré et $O(n)$ dans le cas d'un arbre binaire de recherche filiforme.
- connaître l'algorithme qui permet d'insérer une clé dans un arbre binaire de recherche (voir cours)

Ce qu'il faut savoir faire

Vous devez être capable d'implémenter tous ces algorithmes en Python (voir activité)

Exercices du bac

- Sujet 1 2021 Exercice 3
- Sujet 2 2021 Exercice 1
- Sujet 6 2021 Exercice 3
- Sujet 8 2021 Exercice 4
- Sujet 1 2022 Exercice 5
- Sujet 4 2022 Exercice 3
- Sujet 5 2022 Exercice 4
- Sujet 9 2022 Exercice 2

- Sujet 10 2022 Exercice 4
- Sujet 11 2022 Exercice 4
- Sujet 13 2022 Exercice 3
- Sujet 14 2022 Exercice 1