COURS DE NSI (Numérique et Sciences Informatiques)

Classe : terminale	LYCÉE INTERNATIONAL COURS LUMIÈRE
séquence 2	**ALGORITHMES SUR LES ARBRES BINAIRES**
Mr BANANKO K.	

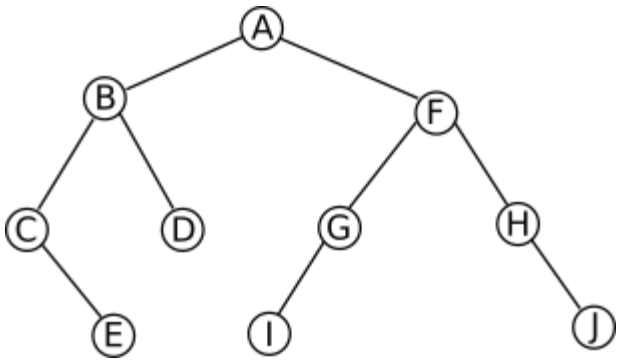
I- NOTATIONS UTILISEES

Il est conseillé de relire au moins une fois le cours consacré aux arbres.

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

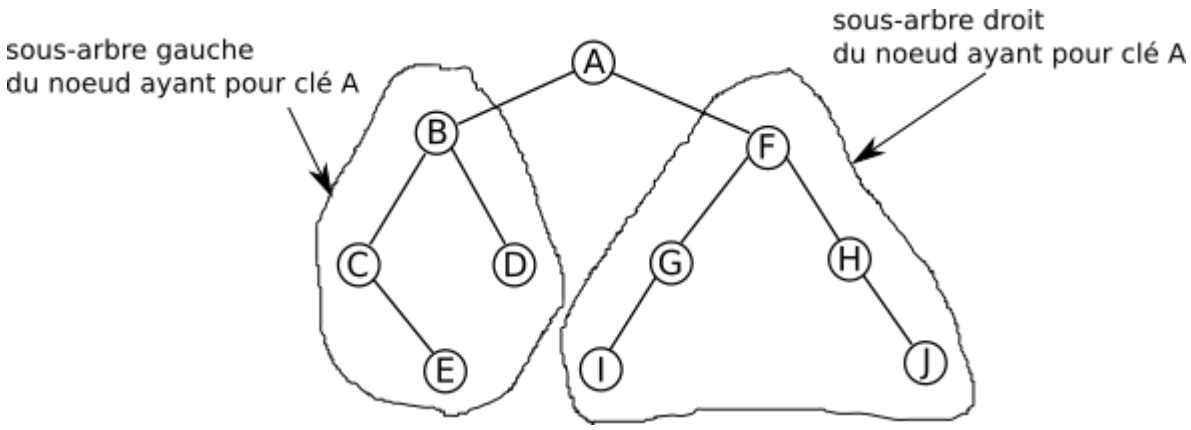
À chaque nœud d'un arbre binaire, on associe une clé ("valeur" associée au nœud on peut aussi utiliser le terme "valeur" à la place de clé), un "sous-arbre gauche" et un "sous-arbre droit"

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 (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 (NIL)

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 (NIL)).

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 (NIL)

II- 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 :

À faire vous-même

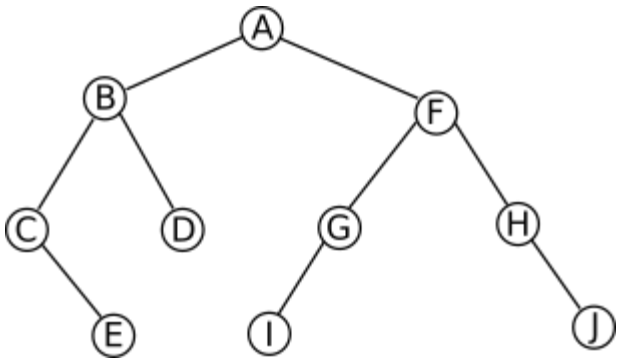
Étudiez cet algorithme :

```
VARIABLE
```

```
T : arbre
x : nœud

DEBUT
HAUTEUR(T) :
  si T ≠ NIL :
    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 nécessaire.



Si vraiment vous avez des difficultés à comprendre le fonctionnement de l'algorithme sur l'arbre ci-dessus, vous trouverez ici un petit calcul qui pourrait vous aider.

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.

III- CALCULER LA TAILLE D'UN ARBRE

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

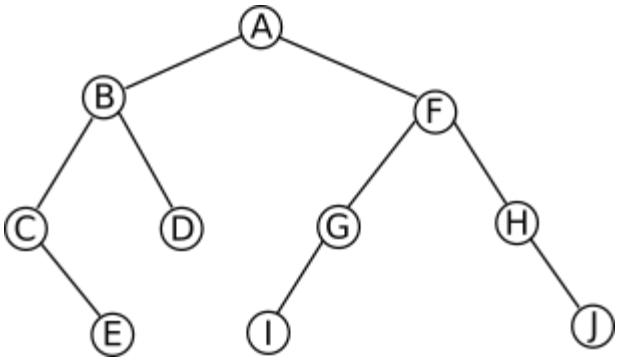
À faire vous-même 2

Étudiez cet algorithme :

```
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
```

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 :



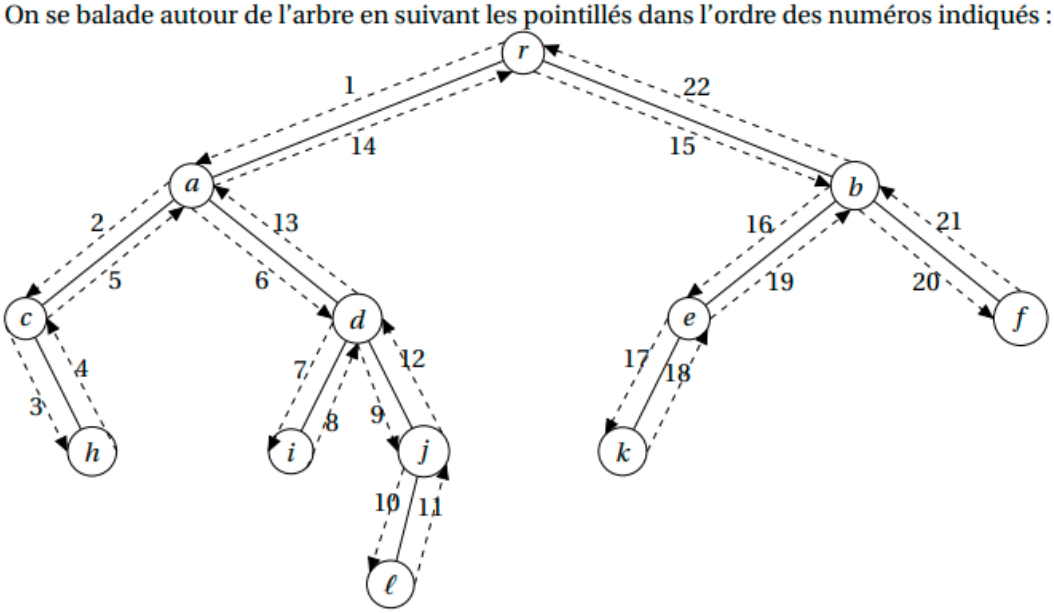
Il existe plusieurs façons de parcourir un arbre (parcourir un arbre = passer par tous les nœuds), nous allons en étudier quelques-unes :

IV- 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.

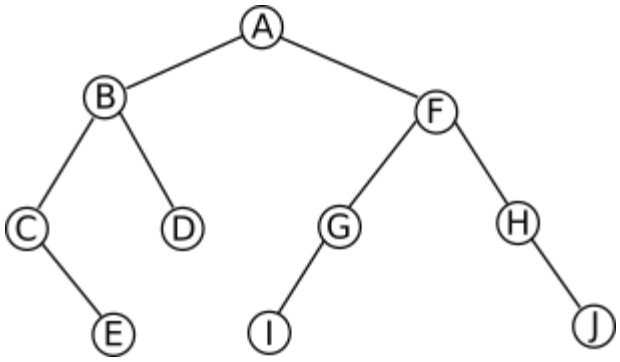
b) parcourir un arbre dans l'ordre infixe

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 ≠ NIL :
```

```
    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

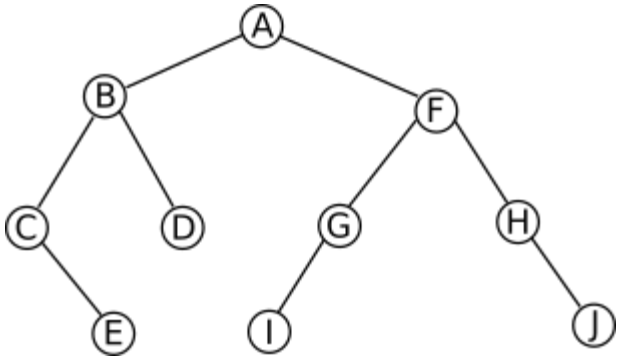
À faire vous-même 4

Étudiez cet algorithme :

```
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
```

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



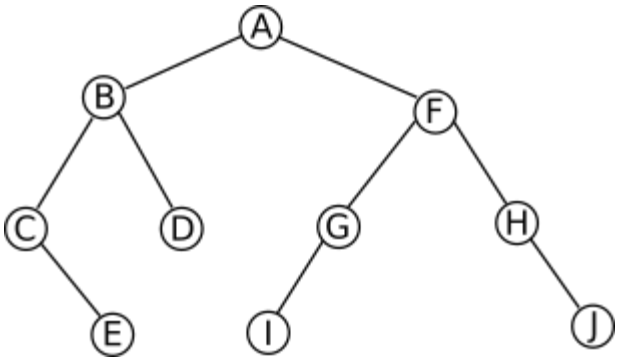
d) Parcourir un arbre dans l'ordre suffixe

À faire vous-même 5

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

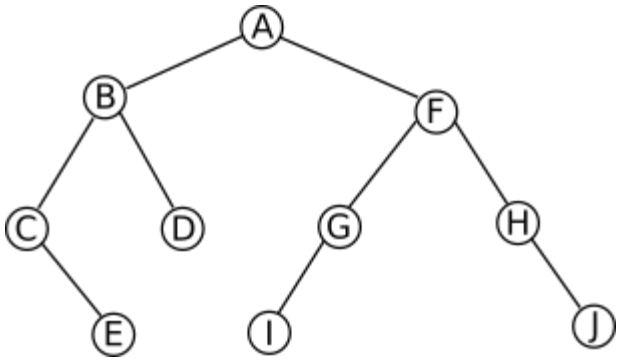
À faire vous-même 6

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 ≠ 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
```

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.

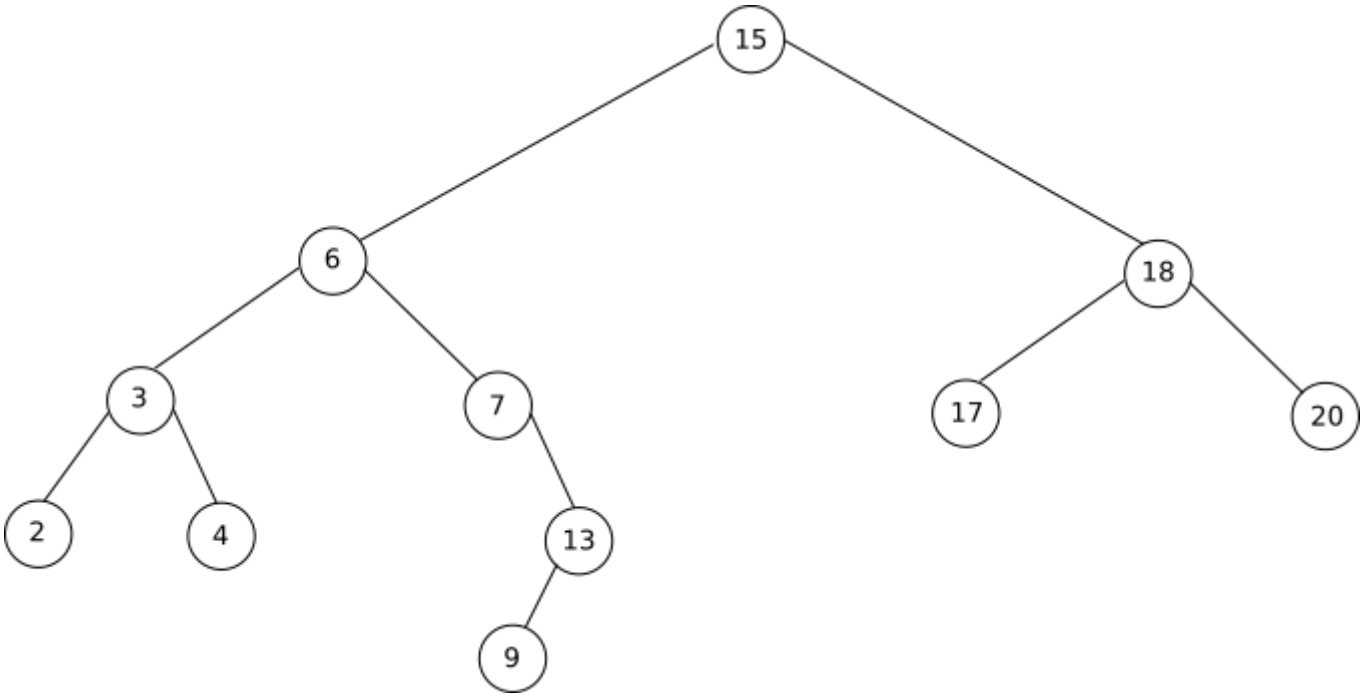
V- ARBRE BINAIRE DE RECHERCHE

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 ⩽ x.clé. Si y est un nœud du sous-arbre droit de x, il faut alors que x.clé ⩽ y.clé

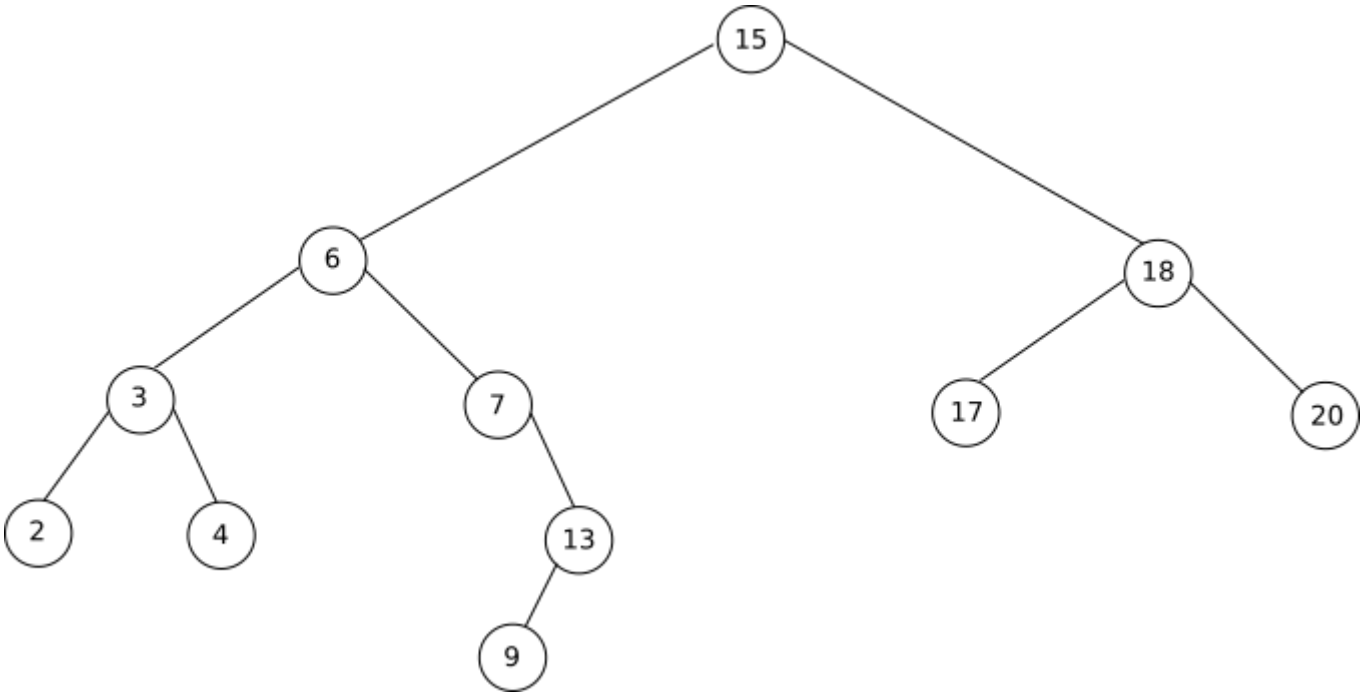
À faire vous-même 7



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

À faire vous-même 8

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.

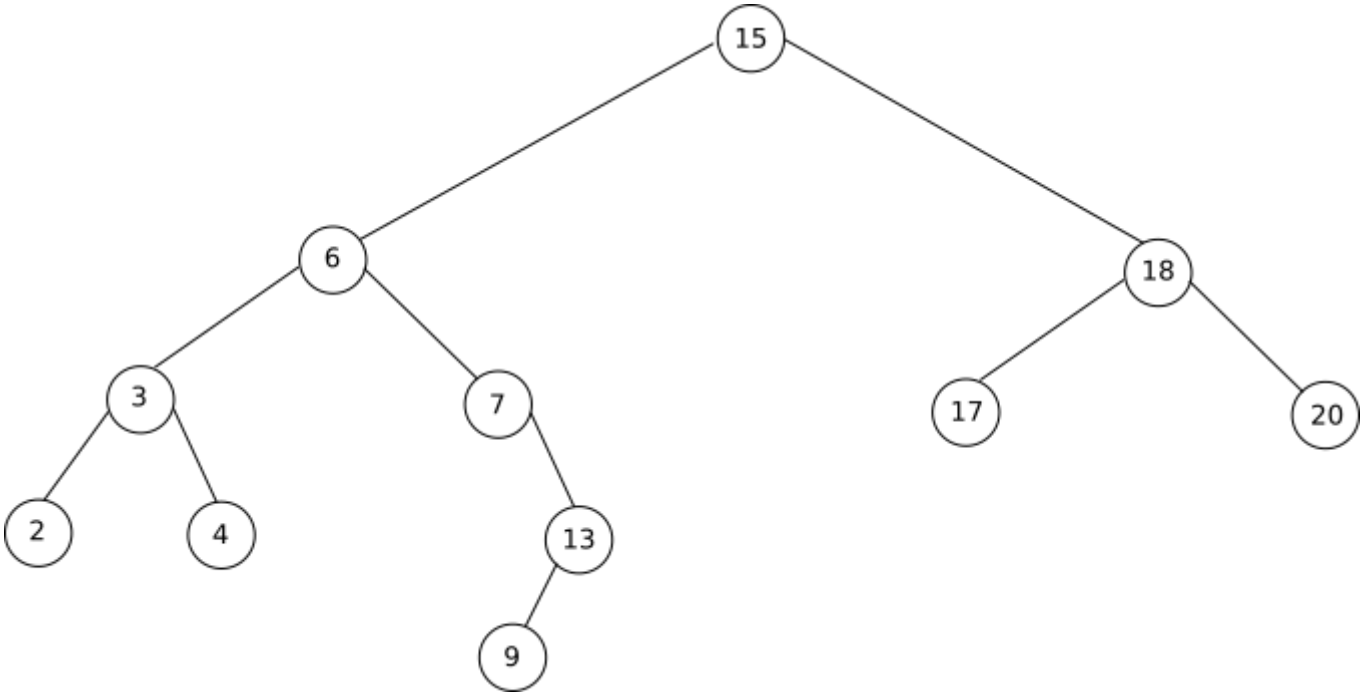
À faire vous-même 9

Étudiez l'algorithme suivant:

```
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 :
    renvoyer ARBRE-RECHERCHE(x.gauche,k)
  sinon :
    renvoyer ARBRE-RECHERCHE(x.droit,k)
  fin si
FIN
```

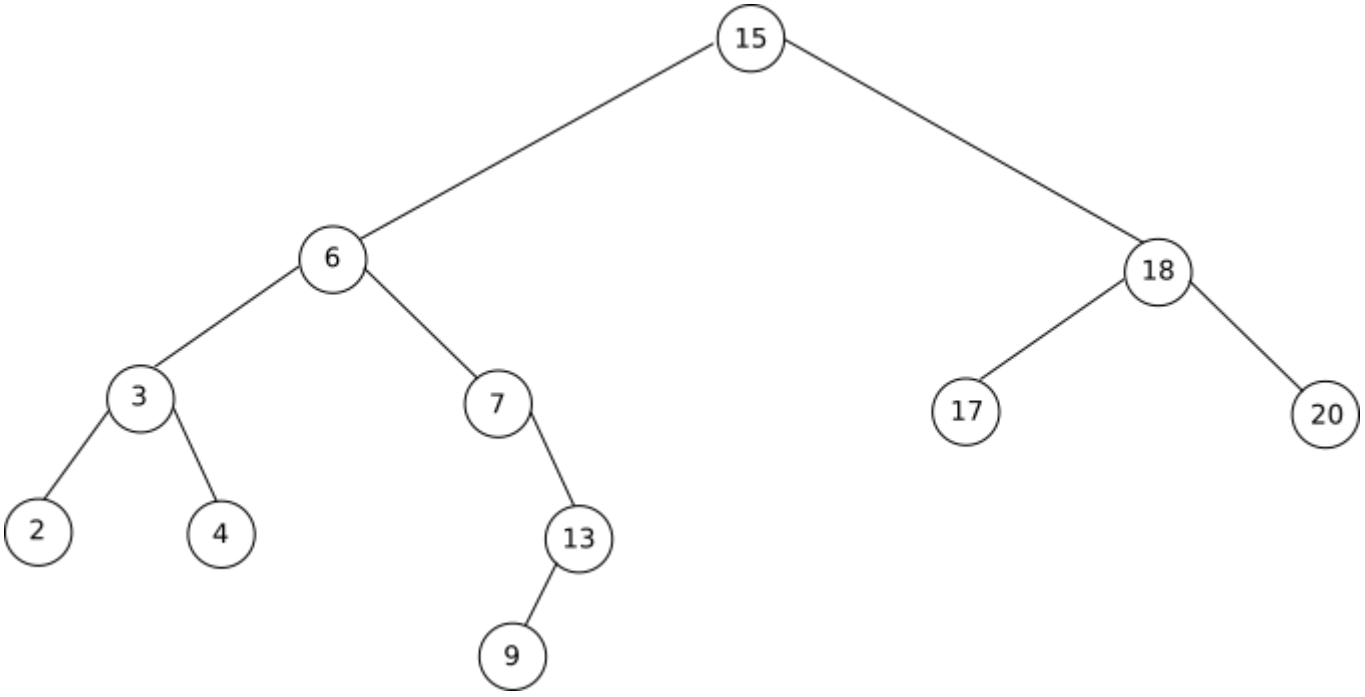
À faire vous-même 10

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



À faire vous-même 11

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 $O(\log_2(n))$. Dans le cas où l'arbre est filiforme, la complexité est $O(n)$. Rappelons qu'un algorithme en $O(\log_2(n))$ 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 :

À faire vous-même 12

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) :

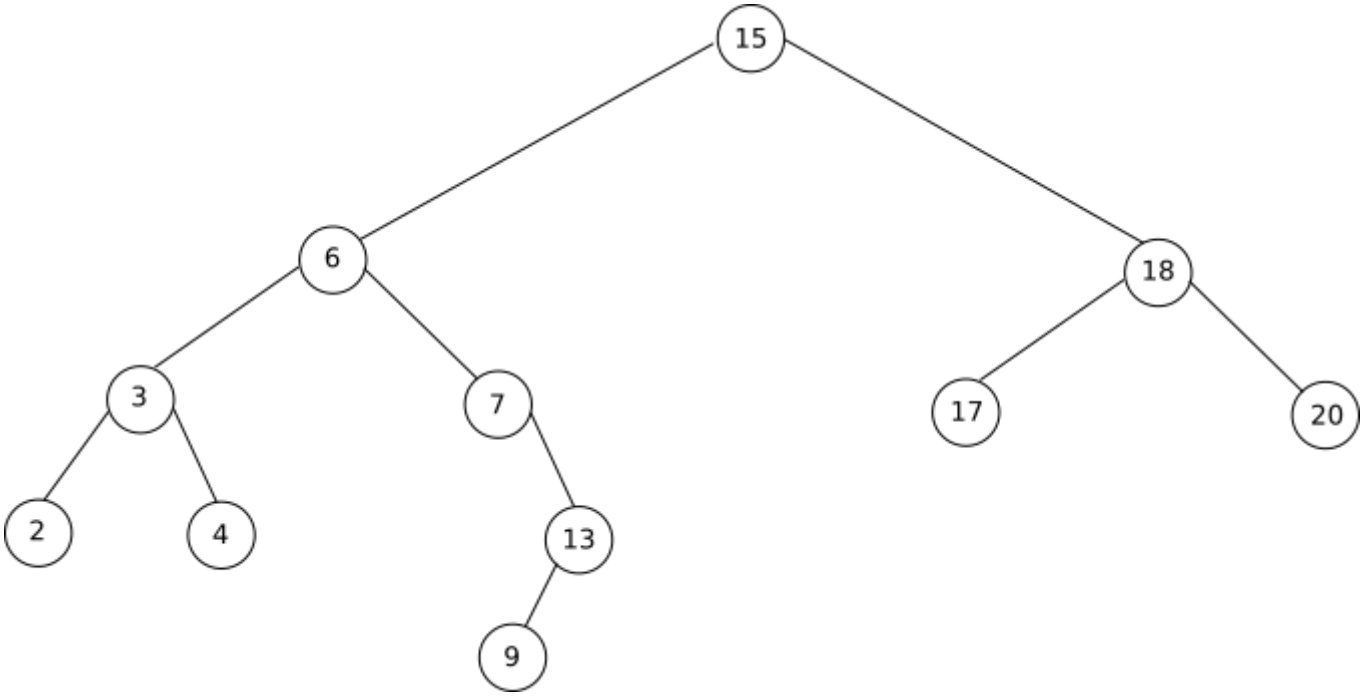
À faire vous-même 13

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
```

À faire vous-même 14

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.

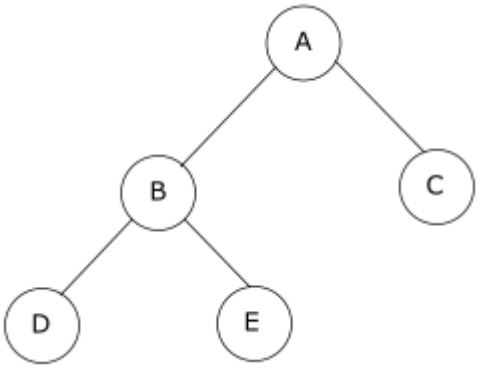
Activité

Dans cette activité, nous allons implémenter des arbres binaires en Python en utilisant des dictionnaires (nous verrons un peu plus tard dans l'année une autre façon de procéder).

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 $\boldsymbol{O(log_2(n))}$ dans le cas d'un arbre binaire de recherche équilibré et $\boldsymbol{O(n)}$ 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)