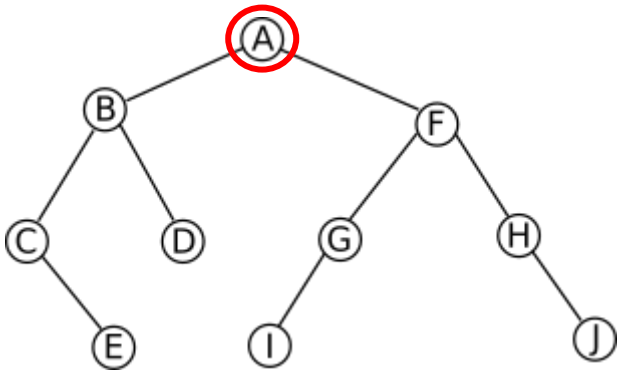
Algorithme sur les arbres binaires

Introduction :

Avant d'entrer dans le vif du sujet (les algorithmes), nous allons un peu approfondir la notion d'arbre binaire (revoir aussi l'activité-1 du thème « 04_Donnees-Structurees-1 ».

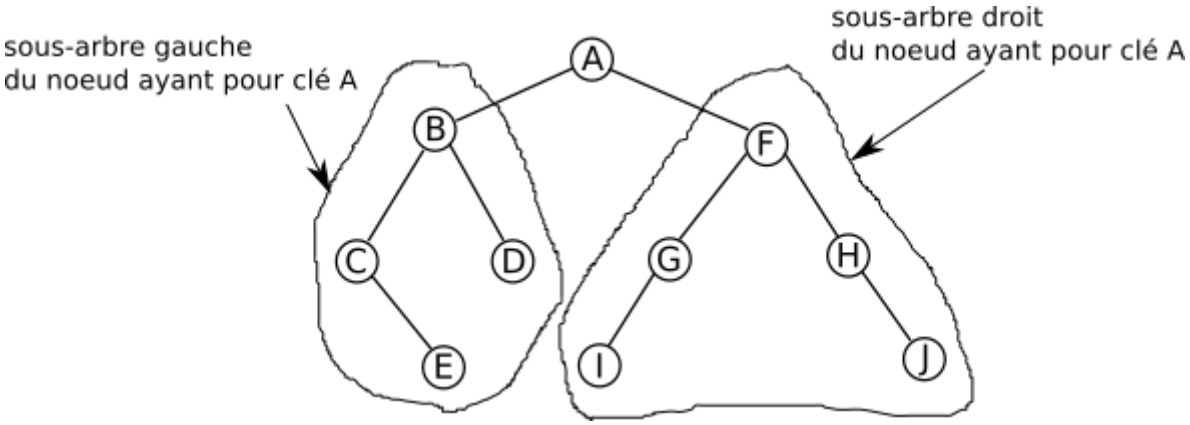
À chaque nœud d'un arbre binaire, on associe une **clé** ("valeur" associée au nœud on peut aussi utiliser le terme "valeur" à la place de clé), **un "sous-arbre gauche"** et **un "sous-arbre droit"**.

Soit l'arbre binaire suivant :



Par exemple, si on prend le nœud ayant pour clé (valeur) A (le nœud racine de l'arbre) on a :

- le sous-arbre gauche est 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 droit 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



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 (droit ou gauche) est un arbre (même s'il contient un seul nœud ou pas de nœud de tout (NIL)).

Dans la suite de cette activité, nous utiliserons les notations suivantes :

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

Pour 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.droit sont des arbres vides (NIL)

1. Calcul de 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 :

Soit l'algorithme suivant:

```
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
```

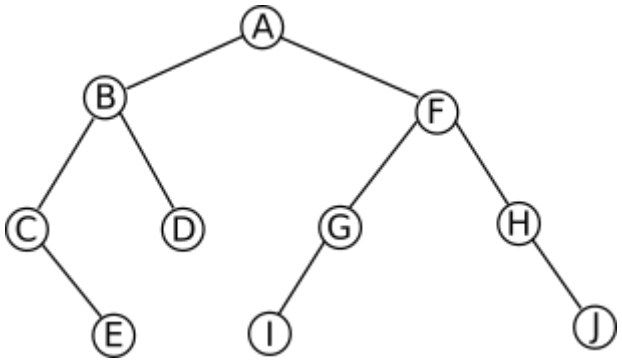
Remarque : La fonction « max(x,y) » renvoie la plus grande valeur des 2 valeurs passées en paramètre (exemple : max(5,6) renvoie 6).

Tout d'abord, rappelons la définition de la **hauteur d'un arbre** : On appelle **hauteur** d'un arbre la **profondeur maximale** des nœuds de l'arbre.

Et donc celle de la **profondeur d'un nœud** : On appelle **profondeur** d'un nœud **le nombre de nœuds du chemin qui va de la racine à ce nœud**. On considère que la racine d'un arbre est à une profondeur 1, donc, la profondeur d'un nœud est égale à la profondeur de son prédécesseur plus 1.

Essayez d'appliquer cet algorithme sur l'arbre binaire ci-dessous.

Cet algorithme est loin d'être simple, n'hésitez pas à écrire votre raisonnement sur une feuille de brouillon.



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

A B C NIL E D F G I H J

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)))) =

1+max(1+max(1+max(0, 1), 1), 1+max(1+max(1, 0), 1+max(0, 1))) =

1+max(1+max(1+1, 1), 1+max(1+1, 1+1)) =

1+max(1+max(2, 1), 1+max(2, 2)) =

1+max(1+2, 1+2) =

1+max(3, 3) =

1+3 = 4

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.

2. Calcul de la taille d'un arbre

Soit l'algorithme suivant:

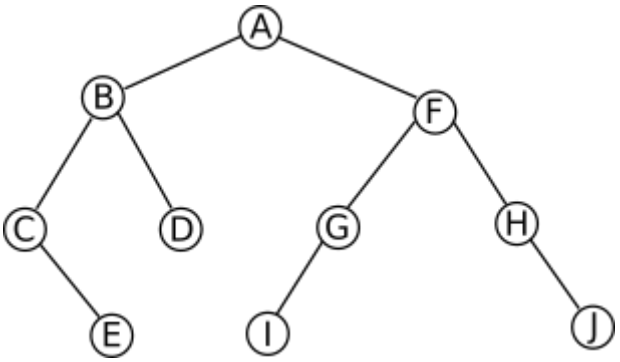
```
VARIABLE
T : arbre
x : nœud

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 « hauteur d'un arbre » son étude ne devrait donc pas vous poser de problème, si vous avez bien compris l'algorithme précédent.

Tout d'abord, rappelons la définition de la **taille d'un arbre** : On appelle **taille** d'un arbre binaire le **nombre de ses nœuds**.

Appliquez cet algorithme sur l'arbre binaire ci-dessous.



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

3. Parcourir un arbre dans l'ordre infixe

Dans le parcours infixe, le traitement de la racine est fait entre les appels sur les sous-arbres gauche et droit.

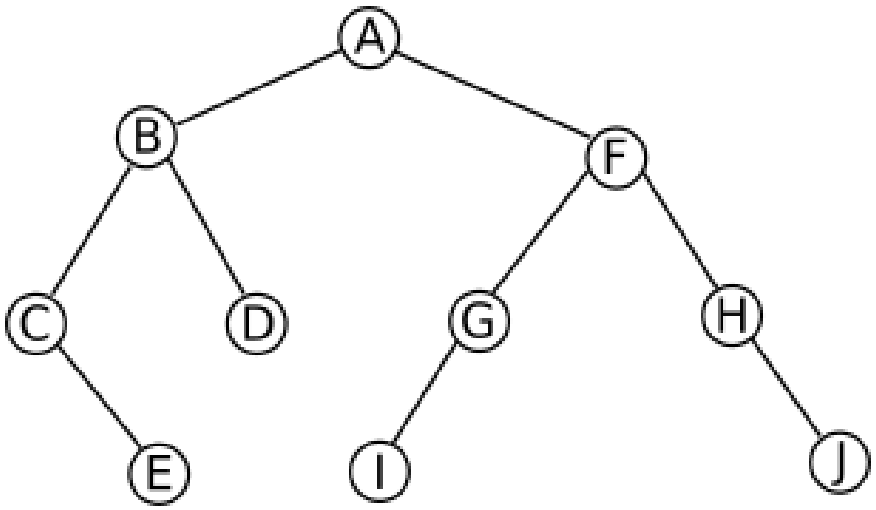
Cela revient à lister chaque sommet ayant un fils gauche la seconde fois qu'on le voit et chaque sommet sans fils gauche la première fois qu'on le voit.

Soit l'algorithme suivant:

```
VARIABLE
T : arbre
x : nœud

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

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



4. Parcourir un arbre dans l'ordre préfixe

Dans le parcours préfixe, la racine est traitée avant les appels récursifs sur les sous-arbres gauche et droit.

Cela revient à lister chaque sommet la première fois qu'on le rencontre en balayant l'arbre.

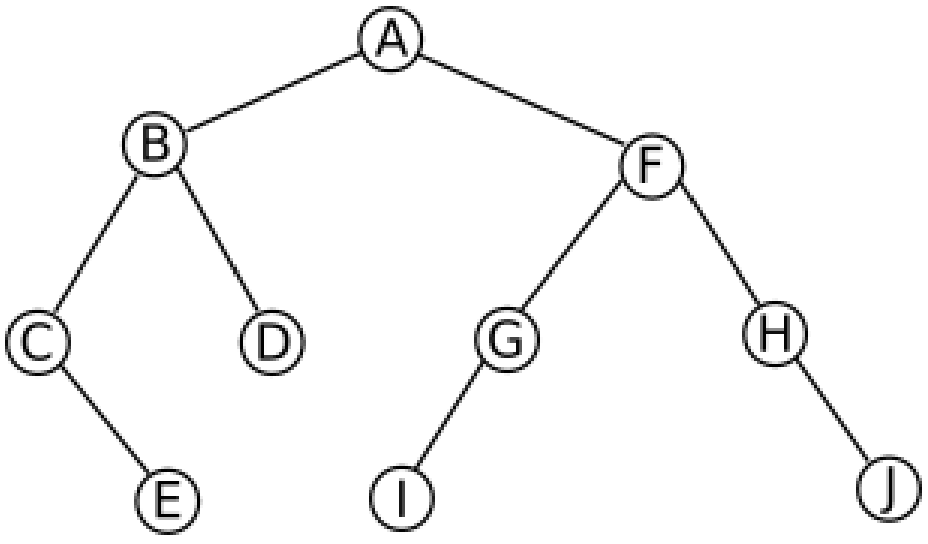
Dans le cas du parcours préfixe, un nœud est affiché avant d'aller visiter ces enfants

Soit l'algorithme suivant:

```
VARIABLE
T : arbre
x : nœud

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.



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

Dans le parcours suffixe ou postfixe, la racine est traitée après les appels récursifs sur les sous-arbres gauche et droit.

Cela revient à lister chaque sommet la dernière fois qu'on le rencontre en balayant l'arbre.

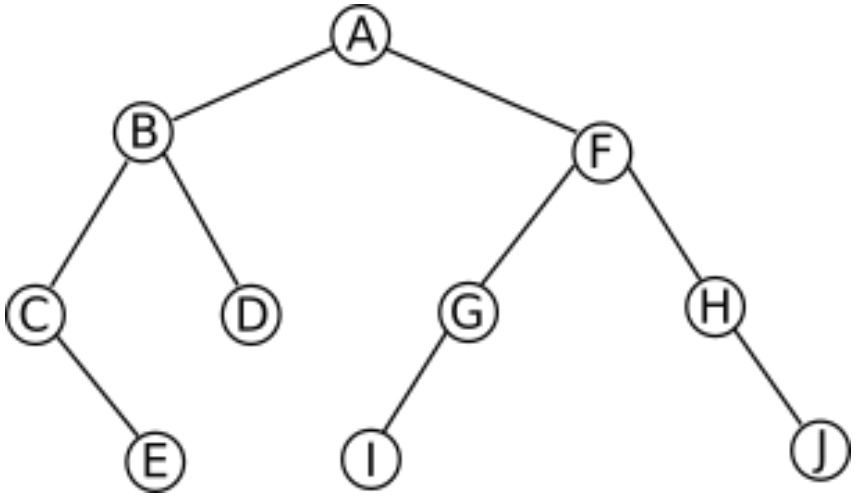
Dans le cas du parcours suffixe, on affiche chaque nœud après avoir affiché chacun de ses fils

Soit l'algorithme suivant:

```
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.



6. Parcourir un arbre en largeur

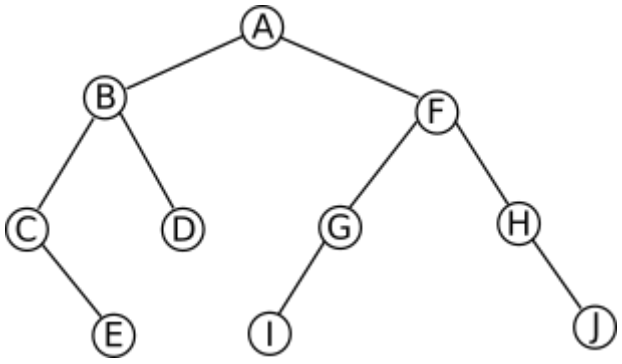
Soit l'algorithme suivant:

```
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 ← défiler(f)
            affiche x.clé
            si x.gauche ≠ NIL :
                    Tg ← x.gauche
                    enfiler(Tg.racine,f)
            fin si
            si x.droit ≠ NIL :
                    Td ← x.droit
                    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.