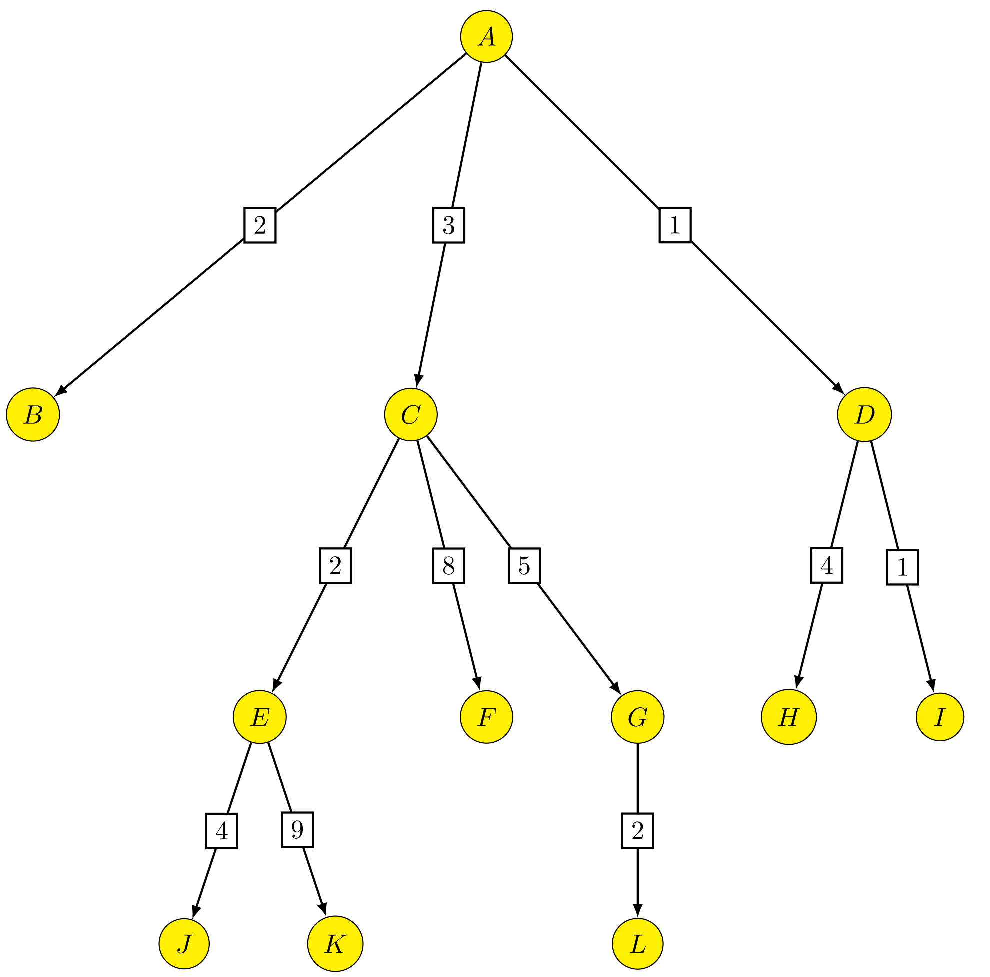
Les **arbres** sont des structures de données hiérarchiques très utilisés dans la pratique. Dans une première partie de ce cours, nous allons construire un type abstrait arbre puis nous verrons certains algorithmes sur les arbres.

Un arbre est constitué de **noeuds** qui peuvent avoir des enfants, qui sont d'autres noeuds. Si un noeud n'a pas d'enfants, c'est une **feuille.** Le sommet de l'arbre est appelé **racine.** Les noeuds autre que les feuilles et la racine sont des **noeuds internes.** Une **branche** est une suite finie de noeuds consécutifs de la racine à une feuille. Un arbre a donc autant de branches que de feuilles. On relie entre eux les noeuds par des **arêtes** (lorsqu'il n'y a pas de sens) ou **arcs** (lorsque l'arête est orientée. On peut ajouter sur l'arête un **coût** entre 2 noeuds, en écrivant une valeur positive ou négative, cela dépendra de la modélisation du problème.

Une arbre peut-être caractérisé par :

- **son arité :** le nombre maximal d'enfants qu'un noeud peut avoir.
- **sa taille :** le nombre de noeuds qui le compose.
- **sa hauteur :** la profondeur à laquelle il faut descendre pour trouver la feuille la plus éloignée de la racine. La racine est le niveau 0 et il est ancêtre de tous les noeuds.

Je vous propose l'arbre suivant :



Dans cet exemple :

- Quel est la racine ? ....
- Combien de feuilles ? ....
- Combien de noeuds internes ? ....
- Quel est l'arité du noeud racine ? ....

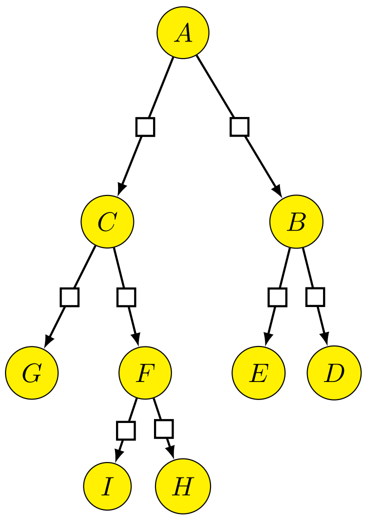
- Quelle est sa taille ? ....
- Quelle est sa hauteur ? ....
- Combien de niveaux ? ....
- Quels sont le chemin et le coût entre les noeuds A et L ? ....

1 Les arbres binaires

Les arbres binaires sont des arbres d'arité 2. Chaque noeud ne peut donc avoir 0,1 ou 2 enfants. Nous allons encore apprendre du vocabulaire de base sur les arbres binaires. puis une application suivra.

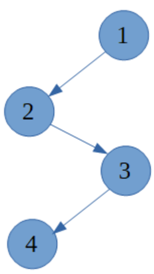
- A partir du noeud racine, nous avons 2 sous-arbres binaires à droite et à gauche : **Sous-Arbre Gauche : SAG** et **Sous-Arbre Droit : SAD**
- Un arbre **complet** : un arbre binaire dont les noeuds possèdent 2 enfants obligatoirement et les feuilles sont au même niveau.
- Un arbre **localement complet** : un arbre binaire dont les noeuds possèdent 2 enfants ou aucun.
- Un arbre **dégénéré** : un arbre binaire dont les noeuds possèdent qu'un ou aucun enfant.

Soit l'arbre binaire :



- A partir du noeud C, délimiter le **Sous-Arbre Gauche : SAG** et le **Sous-Arbre Droit : SAD**
- Cet arbre est-il **complet**, **dégénéré**, **localement complet** ? ....

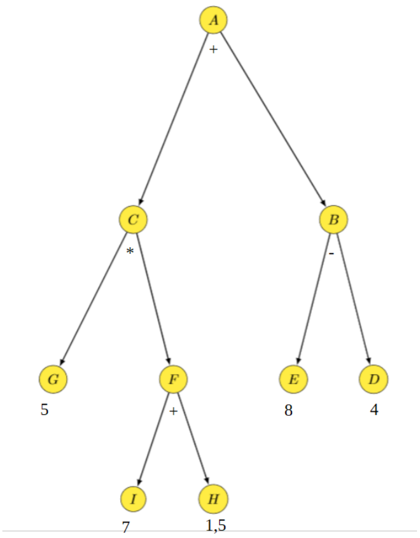
Soit l'arbre binaire :



- A partir du noeud 2, délimiter le **Sous-Arbre Gauche : SAG** et le **Sous-Arbre Droit : SAD**
- Cet arbre est-il **complet**, **dégénéré**, **localement complet** ? ....

Dans un arbre binaire, les noeuds ont des noms mais ils peuvent être étiquetés : on parle d'**arbre étiqueté**.

Voici un exemple, écrire à droite de l'arbre l'expression aritmétique que représente cet arbre étiqueté ?



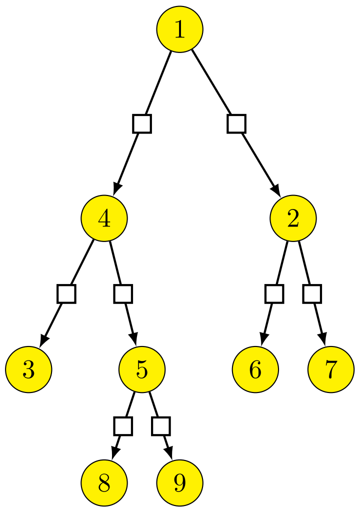
Type abstrait Arbre : Comme pour les structures linéaires, on peut proposer une structure abstraite pour les arbres.

Un arbre est défini de façon récursive sur des éléments de type primitif ou pas, type T car un sous-arbre est aussi un arbre :

- Soit un arbre est vide
- Soit il est composé d'un élément de type T, d'un sous-arbre gauche et d'un sous-arbre droit.

Nous pourrions fournir les opérations de base suivante :

- **creer_arbre_vide()** : retourne un objet de type Arbre, un arbre vide est souvent noté **NIL** : du latin Nihil, un pointeur qui n'a pas été affecté, pointe vers Null.
- **creer_arbre(e,Ag,Ad)** : retourne un objet de type Arbre, racine = e, SAG= Ag et SAD = Ad
- **creer_arbre_feuille(e)** : retourne un objet de type Arbre, racine = e, SAG= vide et SAD = vide
- **racine(A)** : retourne un objet de type T, racine de l'arbre A
- **est_vide(A)** : retourne un objet de type booléen
- **sad(A)** : retourne un objet de type Arbre, sous-arbre droit de l'arbre A
- **sag(A)** : retourne un objet de type Arbre, sous-arbre droit de l'arbre A
- **taille(A)** : retourne la taille de l'arbre A,
  si est_vide(A) alors $taille = 0$ sinon $taille = 1 + taille\_sag(A) + taille\_sad(A)$
- **hauteur(x)** : retourne la hauteur d'un noeud x de l'arbre A,
  si x est racine de l'arbre A alors $hauteur(x) = 0$ sinon $hauteur(x) = 1 + hauteur(y)$ où y est le père de x
- **hauteur(A)** : retourne la hauteur de l'arbre A,
  si hauteur(A) = Max(hauteur(x))

Arbre	Instructions pour le construire :
	a= creer_arbre_feuille(8) b= creer_arbre_feuille(9) c=creer_arbre(5,a,b) d=creer_arbre_feuille(3) e= creer_arbre(4,d,c) f= creer_arbre_feuille(6) g= creer_arbre_feuille(7) h= creer_arbre(2,f,g) j= creer_arbre(1,e,h)

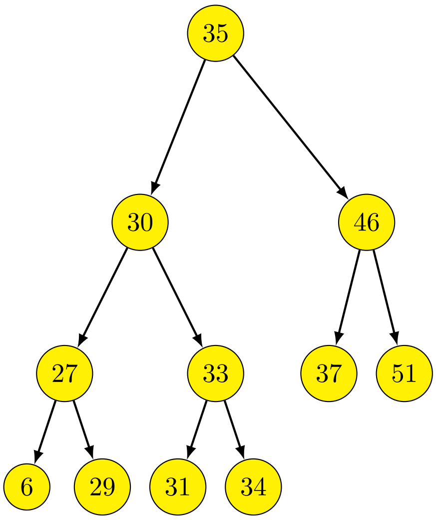
Les arbres binaires de recherche :

L'étiquette d'un noeud est alors appelé **clé**. Un arbre binaire de recherche satisfait à 2 conditions :

- Les clès de tous les noeuds du sous-arbre gauche d'un noeud N sont inférieures ou égales à la clé de N
- Les clès de tous les noeuds du sous-arbre droit d'un noeud N sont supérieures ou égales à la clé de N

Pour passer d'un arbre binaire à un arbre binaire de recherche, on prend un noeud puis on insère par comparaisons successives depuis la racine.

Voici un exemple d'arbre binaire de recherche :



2 Algorithmes sur les arbres :

Nous allons voir 4 algorithmes de parcours d'un arbre, c'est-à-dire comment seront visités tous les noeuds d'un arbre. Puis 2 algorithmes sur les arbres binaire de recherche.

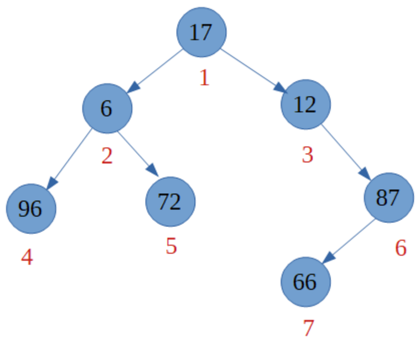
On distingue essentiellement 2 types **le parcours en largeur** (de gauche à droite) et **le parcours en profondeur** (de haut en bas).

Pour le parcours en profondeur, nous avons 3 manières de visiter les noeuds de haut en bas : **le parcours préfixe, le parcours infixe, le parcours postfixe.**

1.Le parcours en largeur :

Le parcours en largeur se programme à l'aide d'une file (FIFO) que nous avons déjà vu.

J'ai mis en étiquette sur cet exemple d'arbre le numéro de visite :



voici l'algorithme en pseudo-code :

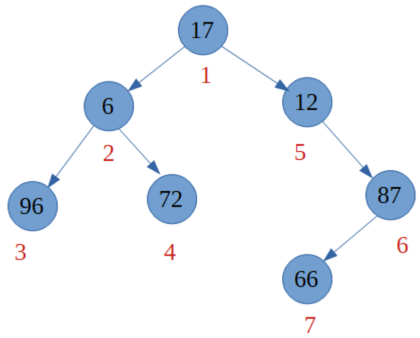
Algorithme 1 : ParcoursLargeur(a)

```
listRes=[]
si A.est_vide() =Faux alors
    F=File()
    F.enfiler(a)
    tant que F.est_vide=Faux faire
        Arbre_courant=F.defiler()
        listeRes= listeRes + Arbre_courant.Racine.cle
        si Arbre_courant.sag.est_vide()= Faux alors
            F.enfiler(Arbre_courant.sag)
        si Arbre_courant.sad.est_vide()= Faux alors
            F.enfiler(Arbre_courant.sad)
Retourner listRes
```

2.Le parcours en profondeur préfixe :

Le parcours se fait en commençant par la racine puis l'arbre de gauche récursivement et enfin l'arbre de droite de façon récursive. On prend chaque noeud que l'on rencontre la première fois.

J'ai mis en étiquette sur cet exemple d'arbre le numéro de visite :



voici l'algorithme en pseudo-code :

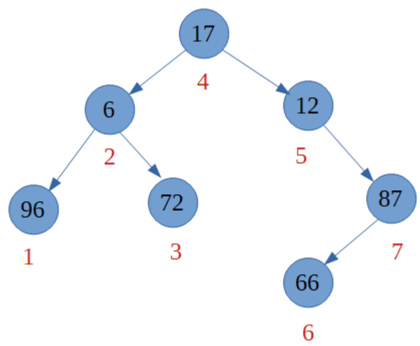
Algorithme 2 : ParcoursProfondeurPrefixe(list,a)

list=[]
si *a.est_vide() =Faux* **alors**
    list=list+a.Racine.Cle
    ParcoursProfondeurPrefixe(list,a.sag)
    ParcoursProfondeurPrefixe(list,a.sad)
Retourner list

3.Le parcours en profondeur infixe :

Le parcours se fait en commençant par le sous-arbre de gauche récursivement puis la racine et enfin le sous-arbre de droite de façon récursive. On prend chaque noeud ayant un fils gauche la seconde fois qu'on le voit et chaque noeud sans fils gauche la première fois qu'on le voit.

J'ai mis en étiquette sur cet exemple d'arbre le numéro de visite :



voici l'algorithme en pseudo-code :

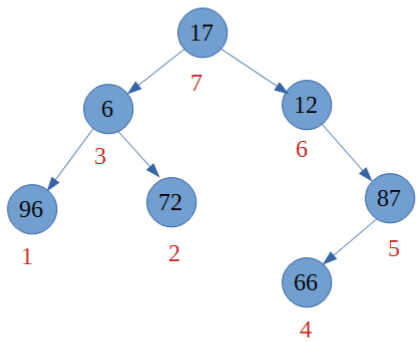
Algorithme 3 : ParcoursProfondeurInfixe(list,a)

```
list=[]
si a.est_vide() =Faux alors
    ParcoursProfondeurInfixe(list,a.sag)
    list=list+a.Racine.Cle
    ParcoursProfondeurInfixe(list,a.sad)
Retourner list
```

4.Le parcours en profondeur postfixe :

Le parcours se fait en commençant par le sous-arbre de gauche récursivement puis le sous-arbre de droite de façon récursive et enfin on ajoute la racine. On prend chaque noeud la dernière fois qu'on le rencontre.

J'ai mis en étiquette sur cet exemple d'arbre le numéro de visite :



voici l'algorithme en pseudo-code :

Algorithme 4 : ParcoursProfondeurPostfixe(list,a)

```
list=[]
si a.est_vide() =Faux alors
    ParcoursProfondeurInfixe(list,a.sag)
    ParcoursProfondeurInfixe(list,a.sad)
    list=list+a.Racine.Cle
Retourner list
```

5.Insérer un noeud dans un ABR :

Le nouveau noeud à insérer sera une feuille. On commence à chercher une clé par comparaison à partir de la racine jusqu'à atteindre une feuille alors on ajoute ce noeud en tant qu'enfant de cette feuille.

voici l'algorithme en pseudo-code :

Algorithme 5 : insererNoeud(a,n)

```
tant que a.est_vide() =Faux faire
    arbre_courant = a
    si n.cle <= arbre_courant.racine.cle alors
        a=a.sag
    sinon
        a=a.sad
si n.cle <= arbre_courant.racine.cle alors
    arbre_courant.sag=arbre(n.cle)
sinon
    arbre_courant.sad= arbre(n.cle)
```

6.rechercher une clé dans un ABR :

Du fait de la relation d'ordre entre les noeuds de notre ABR, il est facile et rapide de retrouver une clé. On commence par la racine, si la clé est présente à la racine on retourne vrai. Si la clé est inférieure, on recommence sur le sous-arbre de gauche et ainsi de suite, si la clé est supérieure on recommence avec le sous-arbre de droite et si la clé n'est pas trouvée on renvoie faux. La complexité dans le pire des cas de cet algorithme qui divise en 2 à chaque fois le problème de recherche est du même ordre que la recherche dichotomique vu en première NSI soit $\Theta(log_2(n))$

voici l'algorithme en pseudo-code :

Algorithme 6 : rechercheCle(a,cle)

```
si a.est_vide=vrai alors
    retourner Faux
sinon
    si a.racine.cle=cle alors
        Retourner vrai
    sinon
        si cle<a.racine.cle alors
            Retourner Recherche(a.sag,cle)
        sinon
            Retourner Recherche(a.sad,cle)
```

remarque importante pour le TP :

Parcourir en profondeur Infixe un ABR (arbre binaire de recherche) vous retourne la liste trié des clés des noeuds.

Récapitulatif des compétences attendues pour ce cours et les exercices qui y sont liés
(**en gris foncé**) :

Compétences	Commentaires	Conseils
Analyser et modéliser un problème en terme de flux et de traitement d'informations	Au préalable, il faut formaliser le problème avec des schémas, des algorithmes, un langage,...	S'approprier le contexte en lisant la totalité du sujet, prendre un brouillon et y mettre quelques idées avant de résoudre sur ordinateur, vous y gagnerez du temps
Décomposer un problème en sous-problèmes, reconnaître des situations déjà analysées et réutiliser des solutions	Décomposer le problème en sous-problèmes que l'on pourra résoudre plus facilement	Faire un état de l'existant : qu'avons-nous à notre disposition ? Quelles sont les technologies disponibles dans ce contexte ? Sur quel existant puis-je me baser ?
Concevoir des solutions algorithmiques	Un algorithme est une suite d'instructions logiques, d'étapes permettant de résoudre un problème. La conception va donner une solution, il est nécessaire de connaître des méthodes.	Il existe bien souvent plusieurs algorithmes qui donnent une même solution. La performance de l'algorithme interviendra post-bac.
Traduire un algorithme dans un langage de programmation, en spécifier les interfaces et les intéractions, comprendre et réutiliser les codes existants, développer des processus de mise au point et de validation de programme	Une fois l'algorithme écrit, il faut le traduire en une suite d'instructions comprises par une machine : programmation de l'algorithme en Python puis de vérifier sa correction = il produit bien le résultat attendu.	Comme il existe plusieurs algorithmes, il en est autant pour nos programmes !
Mobiliser les concepts et les technologies utiles pour assurer les fonctions d'acquisition, de mémorisation, de traitement et de diffusion des informations	Dans le monde actuel, la récolte, le stockage et la diffusion des informations est devenue primordiale	Le programme de NSI ne requiert pas de connaissance d'un système d'information (SI) en particulier mais les SGBD sont liès au programme
Développer des capacités d'abstraction et de généralisation	L'écriture d'algorithmes et de programmes informatiques va développer chez vous une forte capacité d'abstraction qui vous sera utile en mathématiques, en physiques, chimie, svt,...	Cette démarche d'abstraction vous sera utile dans toutes les autres matières où vous devez souvent parvenir à une formule littérale plutôt que de raisonner sur des valeurs numériques, physiques.

Sources :

A - Cours POLYTECH Paris-UPCM

B - Cours algorithme et programmation CNAM (Programmation Java)

C - Spécialité NSI Tle -Edition : Ellipses Bonnefoy / Petit

D - Algoritmique edition : Dunod Cormen / Leiserson / Rivest / Stein