Séquence sur les arbres

Table des matières

Compétences cibles :

- 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.).

Julien Mondary
Benjamin Mousset

Les arbres – COURS

I. Structures arborescentes

En informatique on utilise souvent les structures en forme d'arbre : en effet les informations sont très souvent hiérarchisées. Cela tient également au fait que ces structures arborescentes permettent de stocker un grand nombre de données de sorte que les leurs accès soient efficaces.

1. Les arbres dans la vie courante

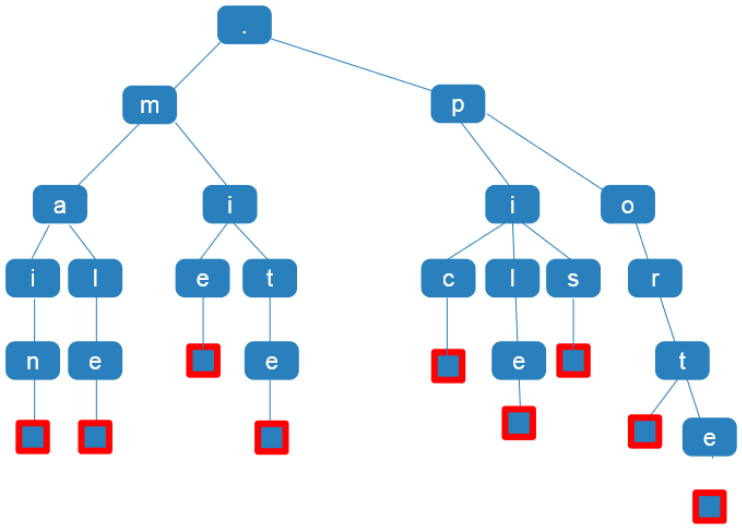
a. Arbre généalogique

Un arbre généalogique représente la descendance d'une personne ou d'un couple. Les nœuds de l'arbre sont étiquetés par les membres de la famille et leurs conjoints. L'arborescence est construite à partir des liens de parenté (les enfants du couple).

Exercice : faire l'arbre généalogique de Napoléon Bonaparte.

b. Exemple des mots (arbre lexicographique.)

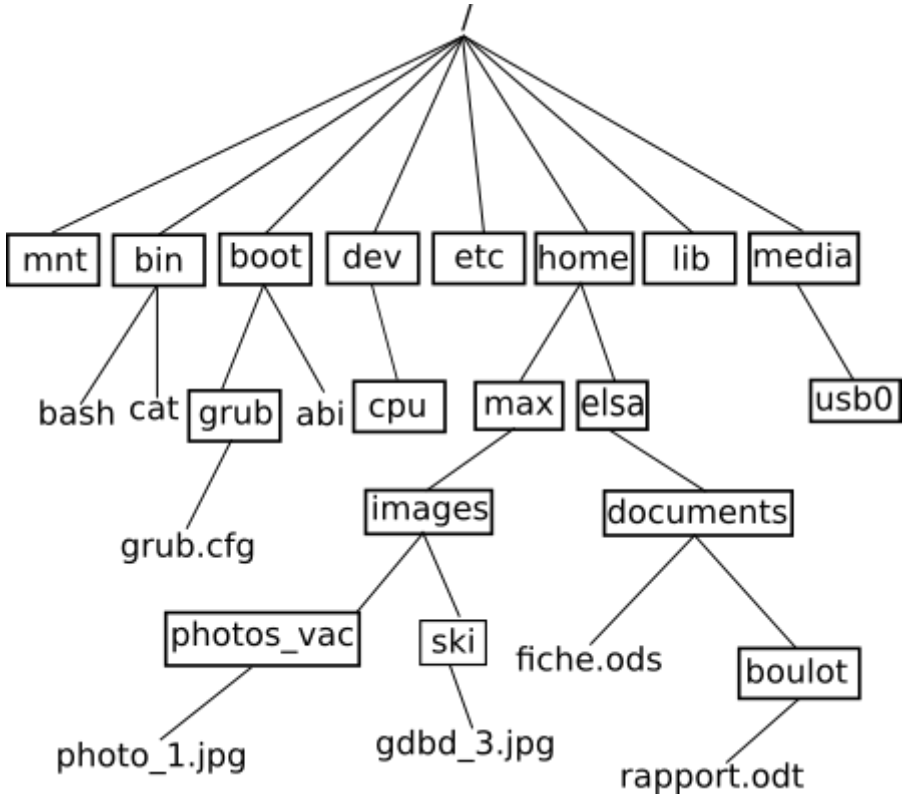
Un arbre lexicographique, ou dictionnaire, représente un ensemble de mots. Les préfixes communs à plusieurs mots apparaissent une seule fois dans l'arbre, ce qui se traduit par un gain d'espace mémoire. De plus la recherche d'un mot est assez efficace, puisqu'il suffit de parcourir une branche de l'arbre en partant de la racine, en cherchant à chaque niveau parmi les fils du nœud courant la lettre du mot de rang correspondant.



Exercice :

1. Ecrivez tous les mots que donne cet arbre.
2. Rajoutez les mots « mal », « portail » et « portable »

2. Exemples en informatique



Source : https://pixees.fr/informatiquelycee/n_site/nsi_term_structDo_arbre.html

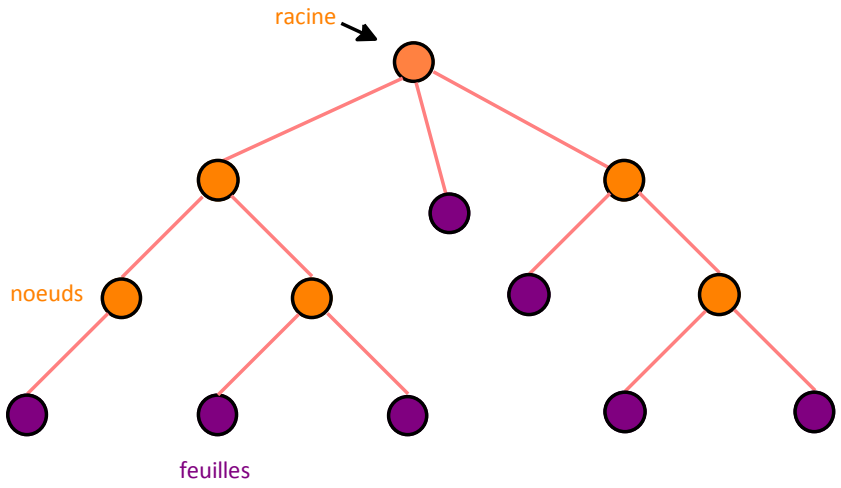
II. Les arbres

1. Vocabulaire

Définition : Un arbre est un ensemble organisé de *nœuds* dans lequel chaque *nœud* a un père et un seul, sauf un nœud que l'on appelle la racine.

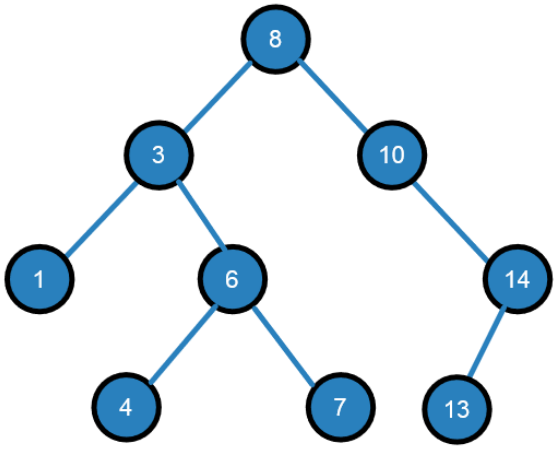
Si le nœud n_1 est le père du nœud n_2 alors on peut dire que le nœud n_2 est le fils du nœud n_1.
Si un nœud n'a pas de fils on dit alors que c'est une ***feuille***. Chaque nœud porte une ***étiquette*** ou ***valeur*** ou ***clé***.

On a l'habitude, lorsqu'on dessine un arbre, de le représenter avec la tête en bas, c'est-à-dire que la racine est tout en haut, et les nœuds-fils sont représentés en-dessous du nœud-père.



Un nœud est défini par son étiquette et ses sous-arbres. On peut donc représenter un arbre par un n-uplet e, $a_1, \ldots, a_k$ dans lesquels e est l'étiquette portée par le nœud, et $a_1, \ldots, a_k$ sont ses sous-arbres.

Exercice : Dans l'arbre ci-dessous identifier les nœuds et les feuilles.
(On considère que la racine est le nœud 8)



III. Arbres binaires

1. Définition

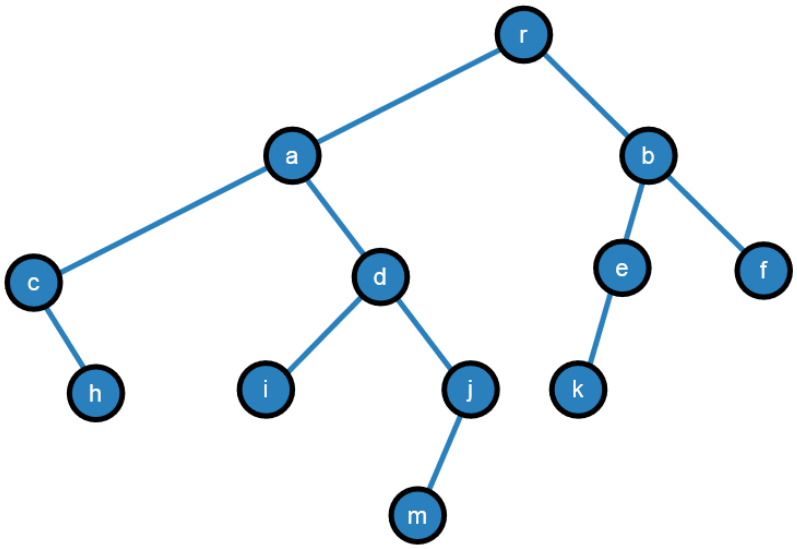
Il faut faire la différence entre les arbres binaires et les arbres généraux. Les noeuds dans les arbres binaires ont une particularité : chaque nœud a un fils **gauche** et un fils **droit**.
Ce fils gauche ou ce fils droit peut être un arbre vide, que l'on notera NUL ou NIL.
Ainsi dans un arbre binaire : un nœud a toujours deux fils.

Les arbres binaires (*AB*) forment une structure de données qui peut être définie récursivement de la manière suivante.

Un arbre binaire est :
- soit vide,
- soit composé d'une racine portant une étiquette (clé) et d'une paire d'arbres binaires, appelés fils gauche et droit (ou sous-arbre gauche et sous-arbre droit)

On peut aussi le dire de la façon suivante :

Un arbre binaire est un arbre avec racine dans lequel tout nœud possède **au plus deux fils** : un éventuel fils gauche et un éventuel fils droit.



Exemple d'arbre binaire

2. Parcours en profondeur

Document 1 :

L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit. Il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin. On peut faire l'analogie avec une exploration d'un labyrinthe.
L'exploration s'arrête quand tous les sommets depuis S ont été visités. Bref, l'exploration progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S.

Références : https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur
Animation : https://fr.wikipedia.org/wiki/Fichier:MAZE_30x20_DFS.ogv

3. Parcours en hauteur

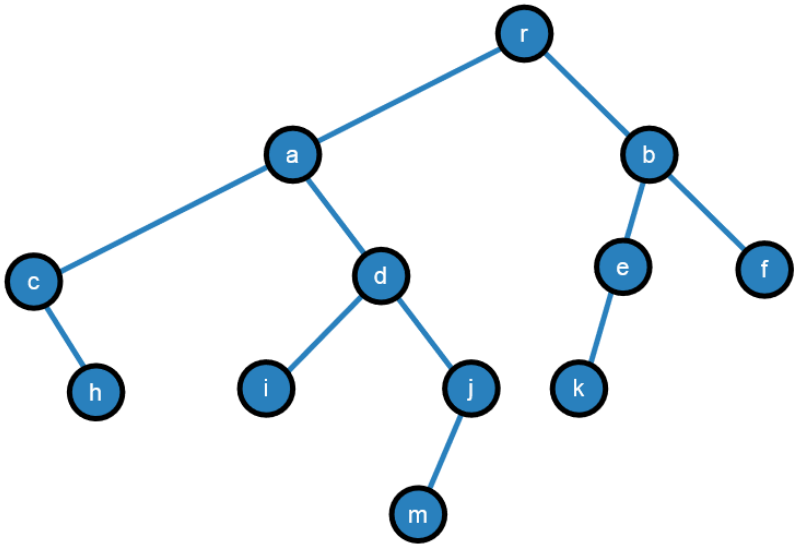
Document 2 :

Cet algorithme diffère de l'algorithme de parcours en profondeur par le fait que, à 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.
Les nœuds déjà visités sont marqués afin d'éviter qu'un même nœud soit exploré plusieurs fois. Dans le cas particulier d'un arbre, le marquage n'est pas nécessaire.
Étapes de l'algorithme :

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.

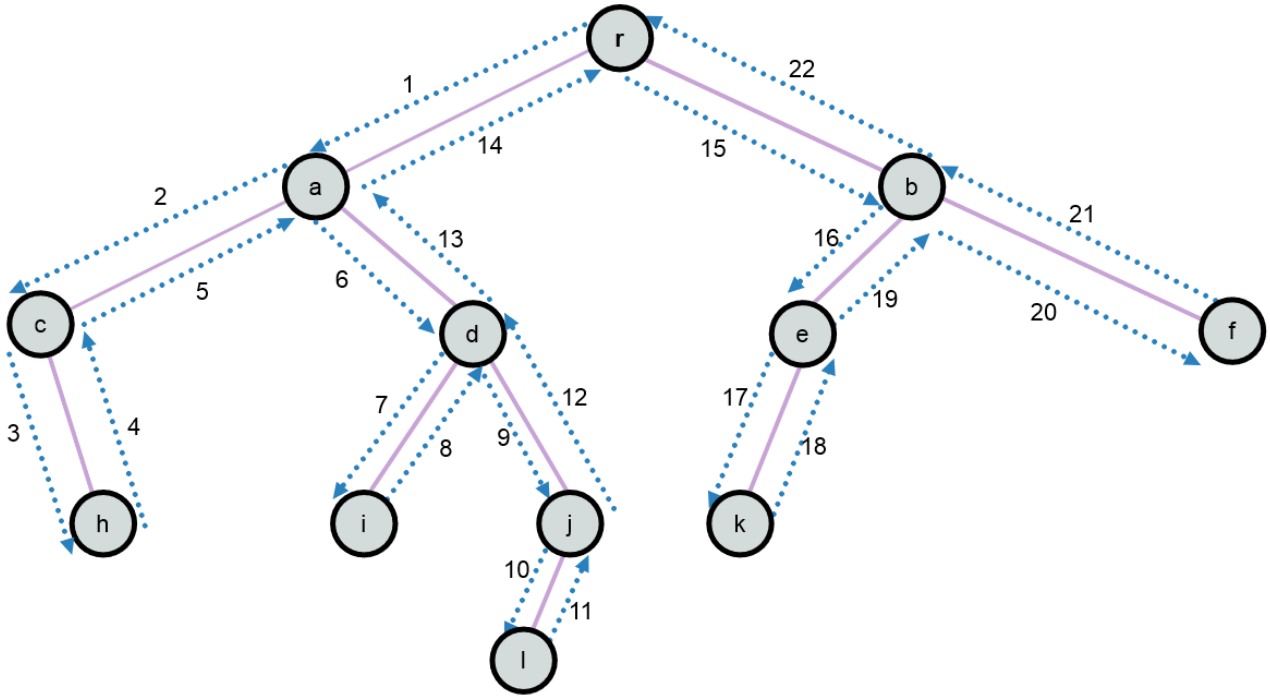
Références : https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur
Animation : https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur#/media/Fichier:Animated_BFS.gif

Exercice : à l'aide l'arbre ci-dessous écrire les étapes du parcours en profondeur et en hauteur.



4. Différents parcours autour de l'arbre

On se déplace 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 **postfixe** : 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.

Exercice :

Donnez :

1. L'ordre préfixe de cet arbre.
2. L'ordre postfixe de cet arbre.
3. L'ordre infixe de cet arbre.

Corrigé

1. r,a,c,h,d,i,j,l,b,e,k,f
2. h,c,i,l,j,d,a,k,e,f,b,r
3. c,h,a,i,d,l,j,r,k,e,b,f.

IV. Les arbres binaires de recherche

1. Qu'est-ce qu'un arbre binaire de recherche (ABR) ?

Un arbre binaire de recherche est organisé comme un arbre binaire. En plus du champ clé et des données satellites, chaque nœud contient les champs gauches, droite et qui pointent sur les nœuds correspondant respectivement à son enfant de gauche, à son enfant de droite et à son parent. Si le parent, ou l'un des enfants, est absent, le champ correspondant contient la valeur NIL. Le nœud racine est le seul nœud de l'arbre dont le champ parent vaut NIL.

2. Propriétés fondamentales des arbres binaires de recherche :

Pour tout nœud n_1 d'arbre binaire de recherche, pour tout nœud n_2, nœud du sous-arbre gauche de n_1 on a les propriétés suivantes : la clé du nœud n_2 est plus petite ou égale à la clé du nœud n_2.
Si n_2 est un nœud de droite alors la clé du nœud n_1 est plus petite ou égale à la clé de nœud n_2.

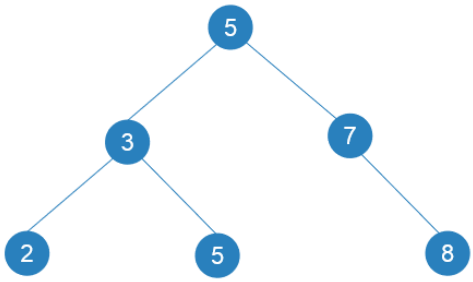
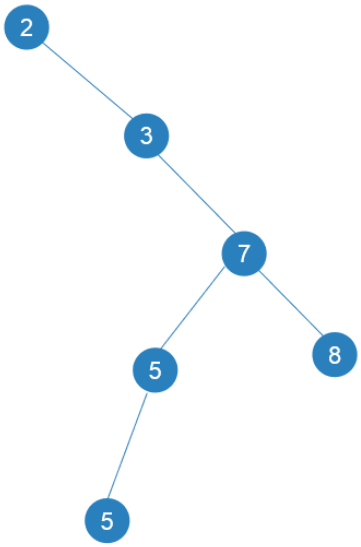


Figure ①



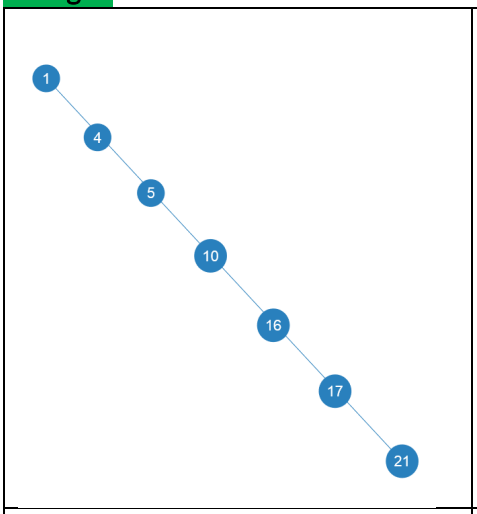
Figure②

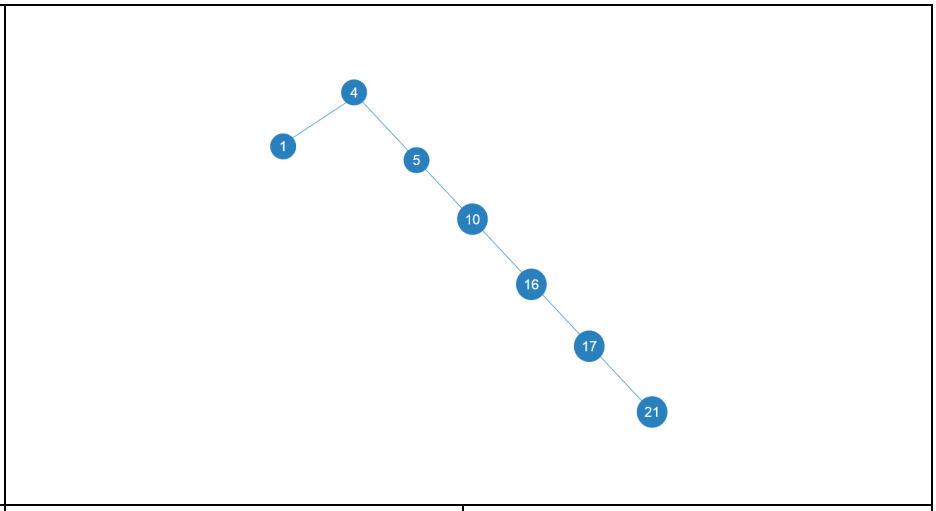
Voici 2 exemples d'arbres binaires de recherche.
Pour un nœud x, les clés du sous-arbre de gauche de x valent au plus clé[x] et celles du sous-arbre de droite valent au moins clé[x].
La figure ① est un arbre binaire de recherche de 6 nœuds et de hauteur 2.
La figure ② est un arbre binaire de recherche de 6 nœuds aussi mais de hauteur 4.
Le temps d'exécution est dans le cas le plus défavorable proportionnel à la hauteur de l'arbre

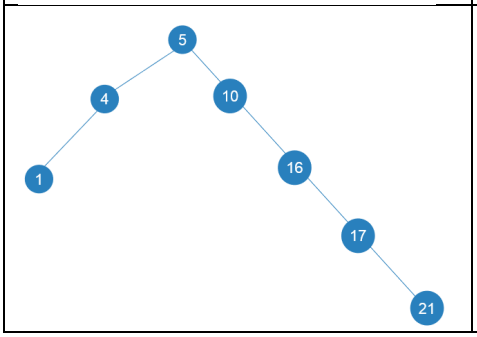
Exercice :

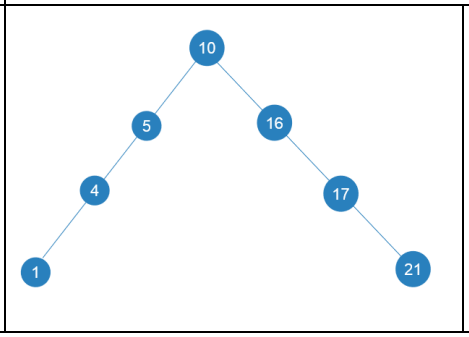
Dessiner des arbres binaires de recherche de hauteur 2,3,4,5 et 6 pour le même ensemble de clés {1,4,5,10,16,17,21}.

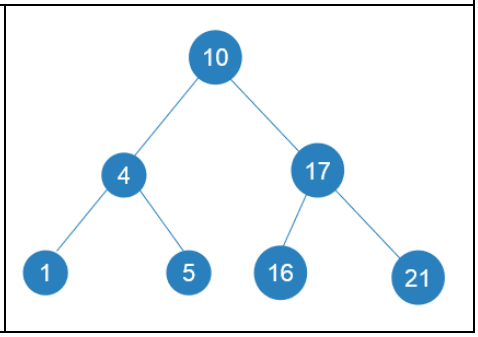
Corrigé :


1
4
5
10
16
17
21


4
1
5
10
16
17
21


5
4
10
1
16
17
21


10
5
16
4
17
1
21


10
4
17
1
5
16
21

Exercices sur les arbres

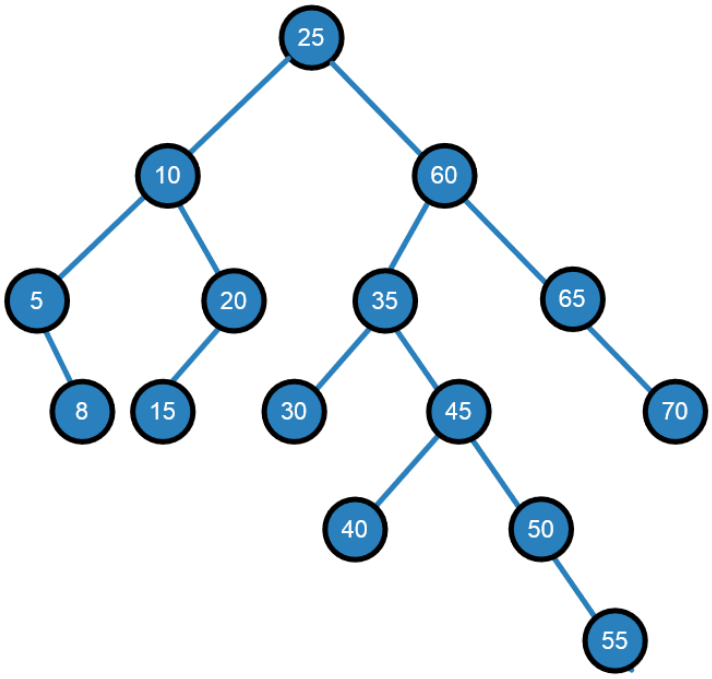
Exercice 1 :

On donne une liste aléatoire de 16 nombres entiers :

25	60	35	10	5	20	65	45	70	40	50	55	30	15	8

Construire dans l'ordre de la liste l'arbre binaire de recherche associé.

Corrigé :



Exercice 2 :

Soit le tableau suivant qui représente un arbre binaire T en triplets (nœud, fils gauche, fils droit) :

23	2	3	5	7	11	13	37	41	19
NIL	7	5	NIL	NIL	19	NIL	41	13	NIL
NIL	11	23	NIL	NIL	NIL	3	2	NIL	NIL

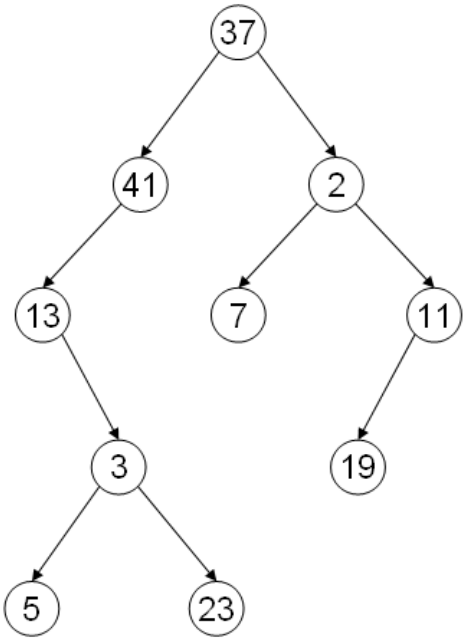
La valeur NIL indique l'absence d'un fils gauche ou droit.

Comment lire le tableau :
La première colonne (indice 0) représente le nœud dont le champ info est 23 (valeur du nœud), le champ gauche est NIL (indice du fils gauche) et le champ droit est NIL (indice du fils droit),
La seconde colonne (indice 1) représente le nœud dont le champ info est 2, le champ gauche est 5 et le champ droit est 0, et ainsi de suite.

1. Dessiner l'arbre binaire T et donner sa taille.
2. Donner sa hauteur.
3. Donner le résultat du parcours de l'arbre T selon l'ordre infixe.
4. Donner le résultat du parcours de l'arbre T selon l'ordre préfixe.
5. Donner le résultat du parcours de l'arbre T selon l'ordre postfixe.

Corrigé :

1. Dessin de l'arbre binaire T (taille = 10) :



2. H= 4
3. Résultat en ordre infixe : 5, 3, 23, 13, 41, 37, 7, 2, 19, 11
4. Résultat en ordre préfixe : 37, 41, 13, 3, 5, 23, 2, 7, 11, 19
5. Résultat en ordre postfixe : 5, 23, 3, 13, 41, 7, 19, 11, 2, 37

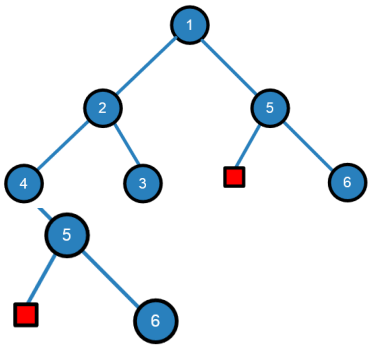
Exercice 3 :

On donne la suite d'instruction suivant :

```
A=creer_arbre(2, creer_arbre_feuille(4), creer_arbre_feuille(3))
B=creer_arbre(5, creer_arbre_vide(), creer_arbre_feuille(6))
C=creer_arbre(1,A,B)
```

1. Représenter l'arbre
2. Donner l'arbre correspondant à l'instruction D=arbredroit(C)
3. Quelle est la valeur retournée par l'instruction suivante : r= racine(B) ?

Corrigé :

1. 

2.

3. C'est 5

TD sur les arbres

Objectifs :

- Utiliser les algorithmes pour calculer la hauteur et la taille d'un arbre.
- Utiliser les algorithmes des parcours infixe, suffixe et postfixe.
- Utiliser les algorithmes de parcours en profondeur.
- Utiliser les algorithmes pour insérer un nœud.
- Utiliser les algorithmes pour rechercher une clé.

I. Calculer la hauteur d'un arbre.

Soit un arbre ab : ab.racine correspond au nœud racine de l'arbre ab
Soit un nœud ***x*** :

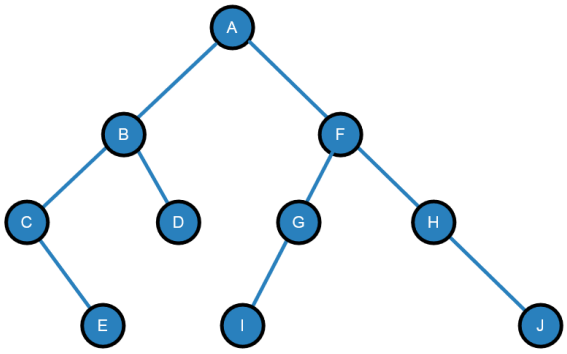
- x.gauche correspond au sous-arbre gauche du nœud ***x***
- x.droit correspond au sous-arbre droit du nœud ***x***
- x.clé correspond à la clé du nœud ***x***

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

Voici l'algorithme de calcul de la hauteur d'un arbre :

```
VARIABLE
ab : arbre
x : noeud

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

Exercice :



1. Calculer la taille de cet arbre.
2. Déroulez l'algorithme.

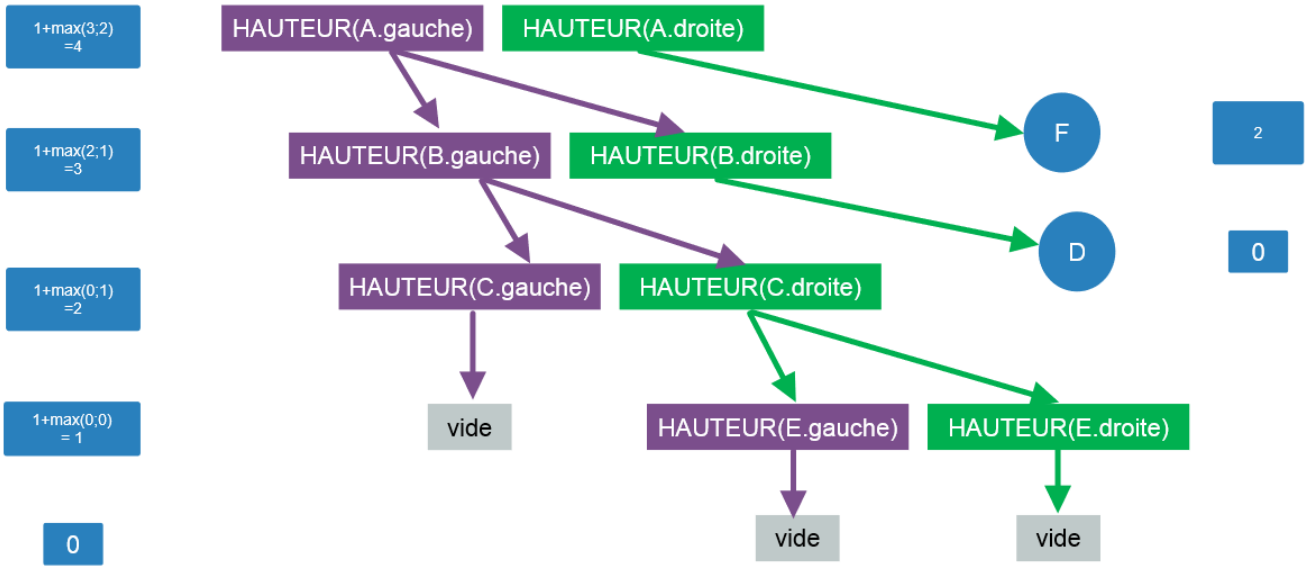
Corrigé

On part de A

x= A

on renvoie (1 + max (Hauteur(A.gauche),Hauteur(A.droit)) et ainsi de suite

Ce qui donne schématiquement :

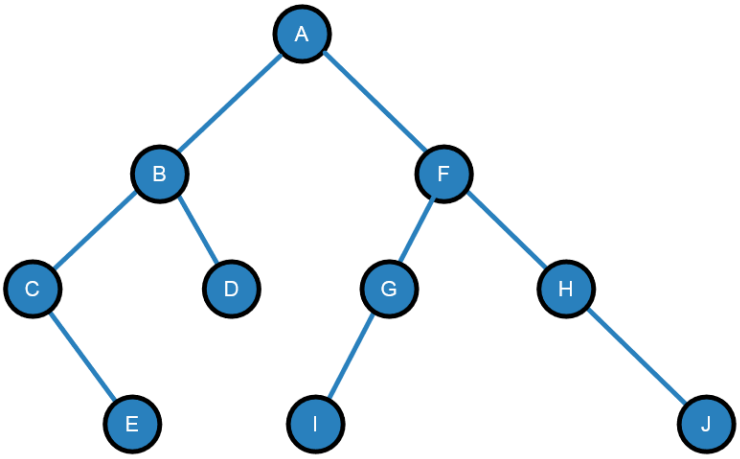


II. Calculer la taille d'un arbre

Voici un algorithme qui permet de calculer le nombre de nœuds dans un arbre :

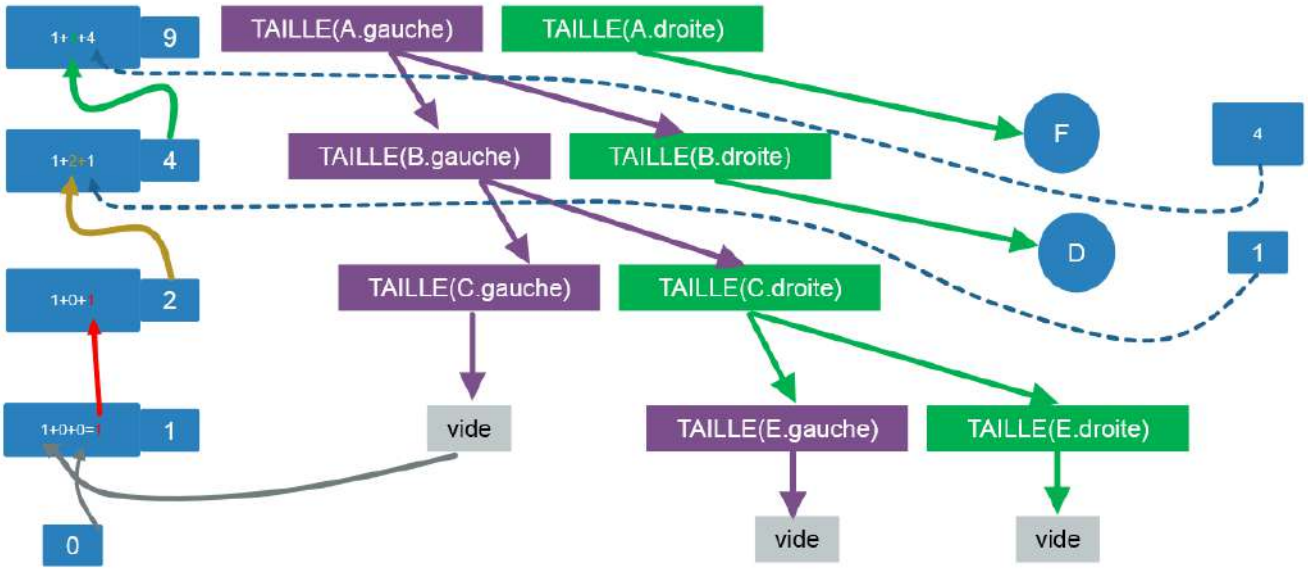
```
VARIABLE
ab : arbre
x : noeud

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

Exercice : appliquez l'algorithme à cet arbre



Corrigé

Taille = 9



III. Parcourir un arbre

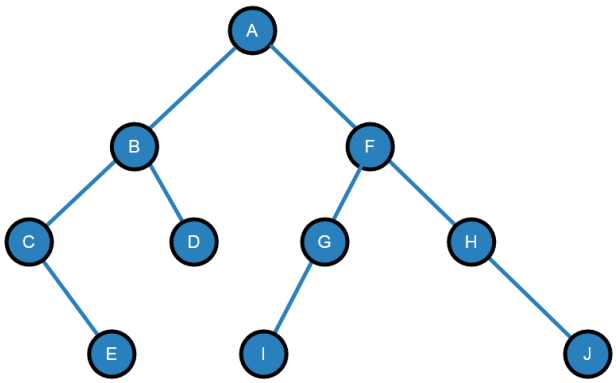
Il existe plusieurs façons de parcourir un arbre. Il existe l'ordre infixe, préfixe et suffixe, en largeur.

1. Parcours infixe.

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.

```
VARIABLE
ab : arbre
x : noeud

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

Exercice : appliquez l'algorithme à cet arbre



Corrigé :

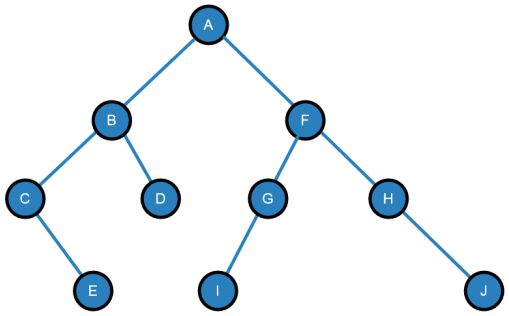
C–E–D-B–A-I-G–F–H-I

2. Parcours préfixe.

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

```
VARIABLE
abr : arbre
x : noeud

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

Exercice : appliquez l'algorithme à cet arbre



Corrigé :

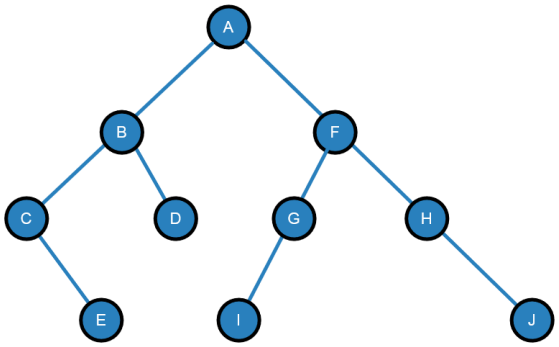
A-B-C-E-D-F-G-I-H-J

3. Parcours postfixe.

L'ordre postfixe : on liste chaque sommet la dernière fois qu'on le rencontre.

```
VARIABLE
abr : arbre
x : noeud

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

Exercice : appliquez l'algorithme à cet arbre



Corrigé :

E-C-D-B-I-G-J-H-F-A

4. Insérer un nœud

Un nouveau nœud est toujours inséré dans une feuille (sinon on n'a pas un arbre binaire de recherche). Il faut commencer par rechercher une valeur de clé par comparaison à partir de la racine jusqu'à atteindre une feuille. Une fois la feuille trouvée le noeux nouveau est ajouté en tant qu'enfant de la feuille.

```
VARIABLE
abr : arbre
n : nouveau nœud

DEBUT
inserer(abr,n) :
  tant que abr ≠ NIL :
       arbre_courant=abr
       si n.cle <= arbre_courant.racine.cle :
             b=b.gauche
       sinon :
             b=b.droit
       fin si
       fin tanque
  si n.cle<=arbre_courant.racine.cle :
       arbre_courant.gauche=Arbre(n.cle)
  sinon :
       arbre_courant.droit=Arbre(n.cle)
  fin si
 FIN
```

5. Rechercher une clé

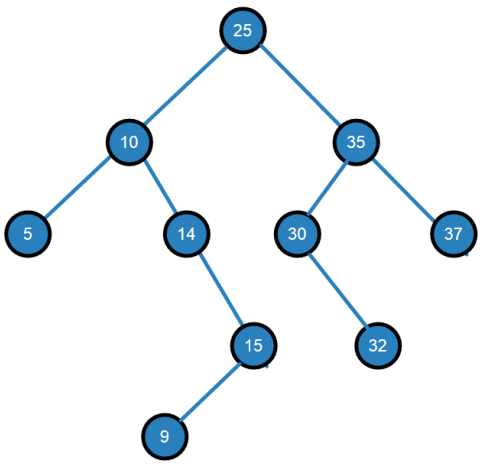
On considère un arbre binaire de recherche. On sait alors que chaque clé d'un nœud est supérieure ou égale à la clé de son fils droit et strictement inférieure à la clé de son fils droit. Cette structure nous permet de savoir dans quelle branche de l'arbre se trouve un élément et de proche en proche de le localiser dans l'arbre.
Pour rechercher une clé dans un arbre binaire de recherche :
On compare d'abord à la racine, si la clé est présente à la racine on renvoie vrai. Si la clé recherchée est supérieure à la racine on recommence pour le sous-arbre droit. Sinon on recommence pour le sous-arbre gauche. Si la clé n'a pas été trouvée on revoie faux

```
VARIABLE
abr : arbre
cle : cle recherchée

DEBUT
rechercher(abr,clé) :
  si abr = NIL :
       renvoyer Faux
  sinon :
       si la cle est à la racine :
             renvoyer Vrai
       sinon :
             si (cle <= racine(abr))
                   recherche(abr.gauche, cle)
             sinon :
                   recherche(abr.droit, cle)
             fin si
       fin si
   fin si
FIN
```

Commentaire sur la complexité : cet algorithme ressemble beaucoup à celui de la recherche dichotomique. On peut donc dire que la complexité en temps, dans le pire des cas, est égale à O *log(n)*

Exercice :

1. Déroulez l'algorithme sur l'exemple ci-dessus, pour trouver la clé 15.
2. Combien d'opérations sont nécessaires ?



6. Tri et arbres : tri rapide (tri du bijoutier)

C'est un exemple classique. Ici il faudra l'arbre binaire nœud après nœud.

Pour trier un tas de diamants, un bijoutier utilise un tamis qui permet de séparer le tas de départ en deux tas. On recommence ensuite avec les deux tas obtenus avec un tamis plus fin. La bonne qualité du tri dépend alors des tamis utilisés à chaque étape, de façon que chaque tas soit séparé en deux parties à peu près égales. Si les mailles des tamis sont trop grandes ou trop petites, le tri n'aura rien de rapide.

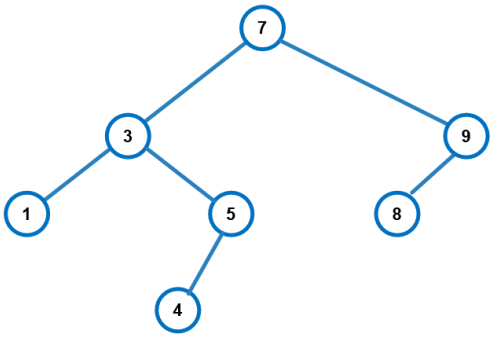
En informatique, cela se traduit de la façon suivante :

- On part d'une liste de nombres à trier.
- On prend le premier nombre, ce sera la racine de l'arbre. Ce nœud, comme tous ceux qui vont suivre, aura deux branche (une à droite et une à gauche).
- On prend le deuxième nombre. S'il est plus petit que le premier, on l'accroche sous la racine à gauche sinon on le met à droite. Et l'on continue ainsi de suite.
- Chaque nombre est descendu dans l'arbre, en allant à gauche s'il est plus petit que le nombre du nœud où il passe, et à droite s'il est plus grand.
- Il prend finalement la première place vide sur laquelle il tombe.

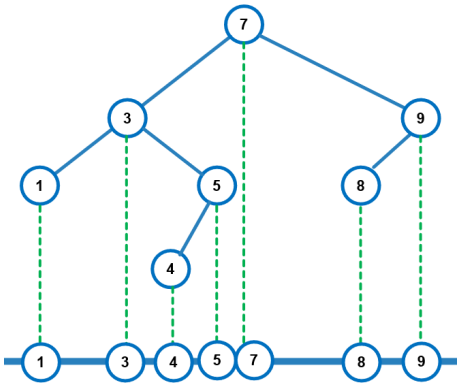
Exercice :

1. Appliquez cette méthode avec les valeurs suivantes : 7,9,3,5,4,1,8.
2. L'un des parcours postfixe, infixe, préfixe de la liste trie la liste. Lequel ?
3. Dans la construction de l'arbre pour une liste de ***n*** nombres, quel est le nombre de comparaisons effectuées dans le pire des cas ?
4. Quel est le nombre de comparaisons effectuées si l'arbre final est un arbre binaire complet (arbre binaire dans lequel tout nœud autre qu'une feuille a deux fils et dans lequel les feuilles sont tous des nœuds de même profondeur).

Corrigé

1. 

2. Le parcours infixe trie la liste. Les éléments de gauche sont en effet par construction plus petits qu'un nœud et sont affichés avant le nœud dans l'ordre infixe et les éléments de droite qui sont, par construction, plus grands sont affichés dans l'ordre infixe après le nœud. Par "récurrence", on a donc un affichage des éléments de la liste dans l'ordre.
   On peut le montrer visuellement en « projetant » l'arbre sur une ligne horizontale telle que les nœud s'écrasent sur cette ligne.



3. Le pire des cas correspond aux cas où la liste est triée (ordre croissant ou décroissant). Le nombre de comparaisons à effectuer est alors de $1 + 2 + \cdots + (n-1) = \frac{1}{2}n(n-1)$
4. Pour ajouter un nœud au niveau de profondeur *p* (la racine étant au niveau de profondeur 0), on effectue *p* comparaisons. Si la profondeur est *h*, on aura effectué une comparaison pour chacun des deux nœuds de profondeur 1 (2×1), deux comparaisons pour chacun des 2^2 nœuds de profondeur 2 (total : 2^1+2x2^2), trois comparaisons pour chacun des 2^3 nœuds de profondeur 3 (total : $2^1 +2\times2^2 + 3\times2^3$). Ainsi le nombre de comparaisons pour une profondeur *h* est :

$$(h-1) \times 2^{h+1} + 2 = \sum_{f=1}^{h} j \times 2^f$$

Avec *n* nœuds (c'est à dire *n* nombres à trier), on a $1 + 2 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$

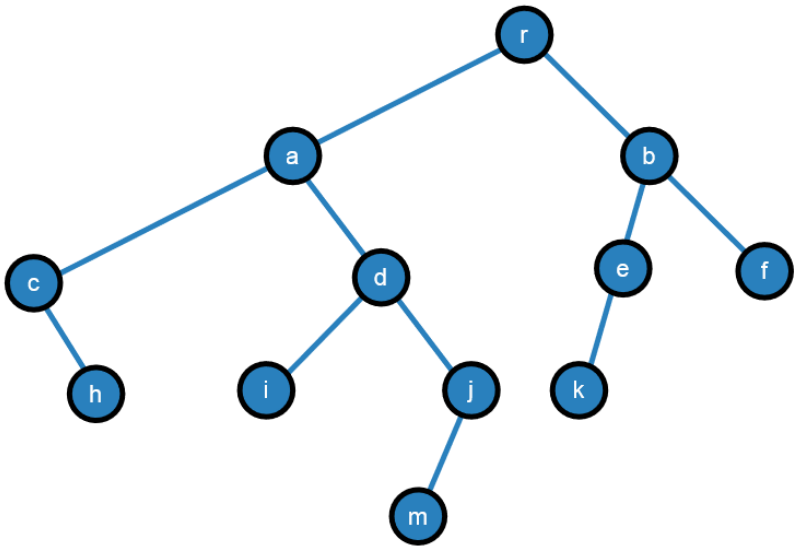
$$(h-1) \times 2^{h+1} + 2 = \sum_{f=1}^{h} j \times 2^f = (\log(n+1) - 1) \times (n+1) + 2$$

On a donc un cas optimal en $O(n \log(n))$ et on peut montrer (comme pour le *quick* sort) que la hauteur moyenne d'un arbre binaire de recherche construit aléatoirement à partir de *n* clefs est $O(\log(n))$

TP sur les arbres avec python (sans Programmation Orientée Objet)

I. Coder un arbre binaire de façon simple.

On utilise l'arbre suivant :



- r est la racine de l'arbre, son fils gauche est a, son fils droit est b.
- *j* a un fils gauche (*m*) et pas de fils droit.
- *h* n'a pas de fils, c'est une feuille de l'arbre.

On décide de coder naïvement cet arbre dans un premier temps de la façon suivante :

```
def noeud(nom, fg = None, fd = None) :
    return {'racine': nom, 'fg' : fg, 'fd': fd}


k = noeud('k')
f = noeud('f')
e = noeud('e', k, None)
b = noeud('b', e, f)
m = noeud('m')
j = noeud('j', m, None)
i = noeud('i')
d = noeud('d', i, j)
h = noeud('h')
c = noeud('c', None, h)
a = noeud('a', c, d)
A = noeud('r', a, b)
```

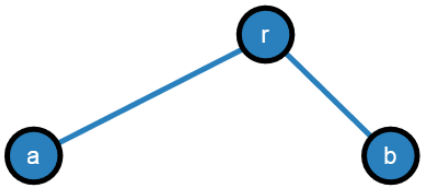
A partir de ce premier codage, on aimerait obtenir une représentation de l'arbre définie comme suit :
L'arbre vide est représenté par [].

L'arbre 

, codé par :

```
A = noeud('r')
```

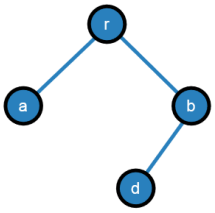
est représenté par `['r', [], []]`

L'arbre 

codé par :

```
b = noeud('b')
a = noeud('a')
A = noeud('r', a, b)
```

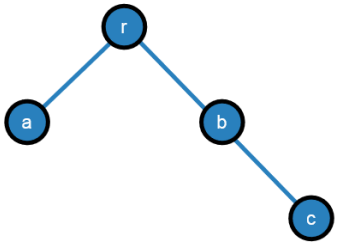
est représenté par `['r', ['a', [], []], ['b', [], []]]`

L'arbre , 

codé par :

```
d = noeud('d')
b = noeud('b',None,d)
a = noeud('a')
A = noeud('r',a,b)
```

est représenté par `['r', ['a', [], []], ['b', ['d', [], []], []]]`

L'arbre , 

codé par :

```
c = noeud('c')
b = noeud('b',c, None)
a = noeud('a')
A = noeud('r',a,b)
```

est représenté par `['r', ['a', [], []], ['b', [], ['c', [], []]]]`

Exercice 1 :

Dessiner l'arbre représenté par
`['r', ['1', [x], [y]], ['2', [v], ['5', [7], []]]]`

Exercice 2 :

Écrire une fonction python récursive nommée **`construit`** permettant d'obtenir, à partir du codage proposé, un arbre représenté sous forme d'une liste comme défini dans les exemples précédents.
L'arbre vide sera noté [] et les arbres comme des listes.

Corrigé

```
def construit(arbre) :
      if arbre == None : return []
      else : return [arbre['racine'],  construit(arbre['fg']),
construit(arbre['fd']) ]
```

II. Représentation visuelle

On voudrait pour que cela soit plus lisible avoir une représentation visuelle de l'arbre.
Par exemple l'arbre représenté par `['r', ['a', [], []], ['b', [], ['d', [], []]]]` donnera :

```
r
-a
--*
--*
-b
--*
--d
---d
---*
---*
```

Exercice 3 :

Pour cela écrire une fonction récursive appelée **`represente`** qui réalise cet affichage.

Corrigé

```
def represente(arbre, p = 0) :
    if arbre==[] : print('*')
    else :
        print(arbre[0])
        p += 1
        print('-' * p, end ='')
        represente(arbre[1], p)

        print('-' * p, end ='')
        represente(arbre[2], p)
```

III. Calcul de la hauteur

Rappel sur la hauteur : voir TD

Exercice 4 :

Ecrire la fonction `hauteur(arbre)` qui permet de calculer la hauteur de l'arbre.

Corrigé :

```
def hauteur(arbre) :
      if arbre == []  :
            return -1
      else :
            h1 = 1 + hauteur(arbre[1])
            h2 = 1 + hauteur(arbre[2])
            return max(h1, h2)
```

IV. Parcours

Rappel sur les parcours infixe et suffixe : voir TD

Exercice 5 :

Ecrire la fonction `parcour_infixe(arbre)` qui permet d'afficher le parcours infixe de l'arbre.

Ecrire la fonction `parcours_postfixe(arbre)` qui permet d'afficher le parcours suffixe de l'arbre.

Corrigé :

```
def parcourspostfixe(arbre) :
    if(arbre !=[]) :
        parcoursPostfixe(arbre[1])
        parcoursPostfixe(arbre[2])
        print(arbre[0], end = ',')

def parcoursInfixe(arbre) :
    if(arbre !=[]) :
        parcoursInfixe(arbre[1])
        print(arbre[0], end = ',')
        parcoursInfixe(arbre[2])
```

V. Arbre avec un dictionnaire

On peut aussi représenter l'arbre précédent avec un dictionnaire.

```
A = {'r':['a','b'], 'a':['c','d'], 'b':['e','f'],

      'c':['','h'], 'd':['i', 'j'], 'e':['k',''], 'f':['',''],

  'h':['',''], 'i':['',''], 'j':['m',''], 'k':['',''], 'm':['','']}
```

Ecrire les fonctions précédentes avec les dictionnaires.

a. `hauteur(arbre, nœud)`
b. `parcours_prefixe(arbre, nœud)`
c. `parcours_infixe(arbre, nœud)`

Corrigé :

```
def hauteur(arbre, noeud='r') :
        if arbre[noeud][0] == '' and arbre[noeud][1] == '' : return 0
        elif arbre[noeud][0] == '' : return 1 + hauteur(arbre, arbre[noeud][1])
        elif arbre[noeud][1] == '' : return 1 + hauteur(arbre, arbre[noeud][0])
        else : return 1 + max( hauteur(arbre, arbre[noeud][0]), hauteur(arbre,
arbre[noeud][1]) )


def parcours_prefixe(arbre, noeud='r') :
    if(noeud != '') :
        print(noeud, end = ',')
        parcoursPrefixe(arbre, arbre[noeud][0])
        parcoursPrefixe(arbre, arbre[noeud][1])


def parcours_infixe(arbre, noeud='r') :
    if(noeud != '') :
        parcoursInfixe(arbre, arbre[noeud][0])
        print(noeud, end = ',')
        parcoursInfixe(arbre, arbre[noeud][1])
```

TP sur les arbres binaires (Programmation orientée objet.)

L'idée de ce type de TP est de faire le lien avec les différents algorithmes vus précédemment sur les arbres (hauteur, taille, parcours…)

I. Rappels sur la POO

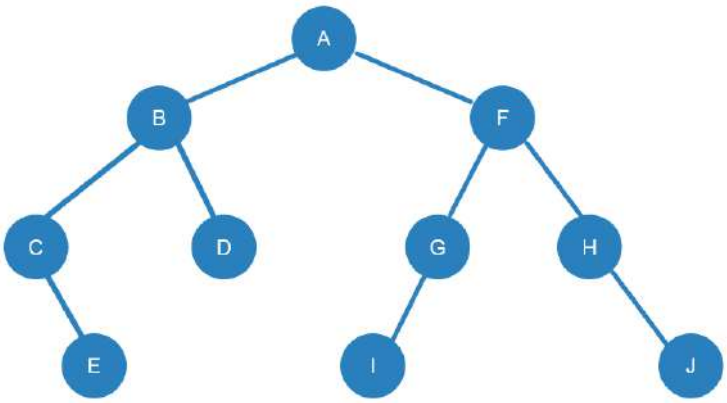
On donne le code suivant qui permet de créer les objets arbres.

```
class ArbreBinaire:
    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None

    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
```

1. Expliquez la méthode **`insert_gauche.`**
2. Expliquez la méthode **`get_valeur.`**
3. Expliquez la méthode **`get_gauche.`**

II. Construction d'un arbre

On construit un premier arbre **`arbre1`** suivant :



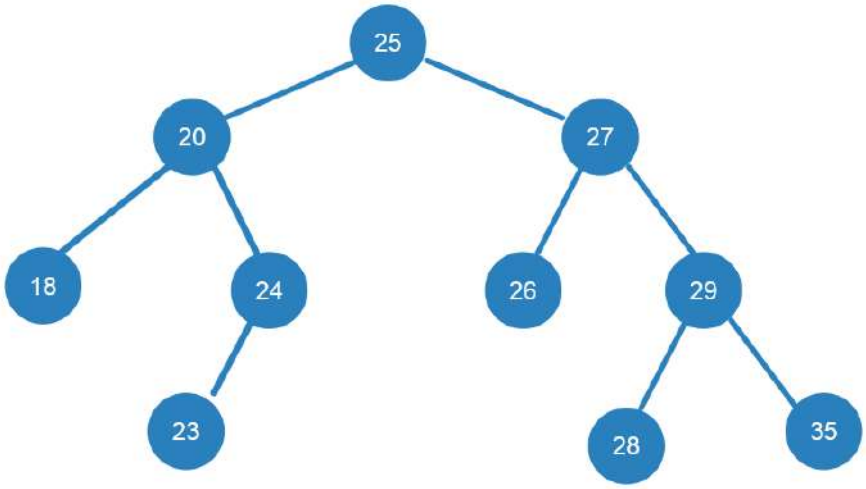
Avec le code suivant :

```
arbre1 = ArbreBinaire('A')
arbre1.insert_gauche('B')
arbre1.insert_droit('F')

Nœud_B = racine.get_gauche()
Nœud_B.insert_gauche('C')
Nœud_B.insert_droit('D')

Nœud_F = racine.get_droit()
Nœud_F.insert_gauche('G')
Nœud_F.insert_droit('H')

Nœud_C = Nœud_F.get_gauche()
Nœud_C.insert_droit('E')

Nœud_G = Nœud_F.get_gauche()
Nœud_G.insert_gauche('I')

Nœud_H = Nœud_F.get_droit()
Nœud_H.insert_droit('J')
```

Exercice : Dessiner le schéma de l'arbre **arbre3** construit par le code suivant :

```
#arbre3
arbre3=ArbreBinaire(25)
arbre3.insert_gauche(20)
arbre3.insert_droit(27)

noeud_20=arbre3.get_gauche()
noeud_20.insert_gauche(18)
noeud_20.insert_droit(24)

noeud_24=noeud_20.get_droit()
noeud_24.insert_gauche(23)

noeud_27=arbre3.get_droit()
noeud_27.insert_gauche(26)
noeud_27.insert_droit(29)

noeud_29=noeud_27.get_droit()
noeud_29.insert_gauche(28)
noeud_29.insert_droit(35)
```

Corrigé



On peut créer une fonction affiche qui affiche l'arbre

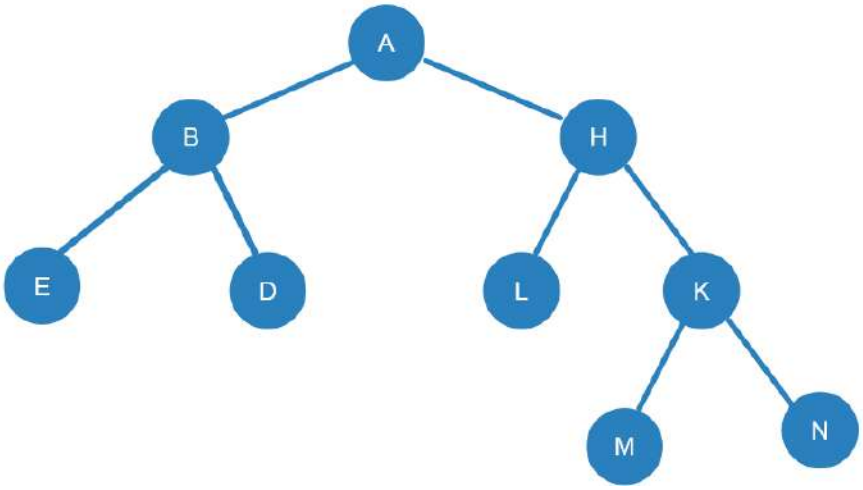
```
def affiche(arbre):
    if arbre != None:
        return (arbre.get_valeur(),affiche(arbre.get_gauche()),
affiche(arbre.get_droit()))
```

Par exemple pour l'arbre3 on obtient :

```
(25, (20, (18, None, None), (24, (23, None, None), None)), (27, (26,
None, None), (29, (28, None, None), (35, None, None))))
```

Question : Que peut-on dire sur le type de l'affichage retourné.
C'est un tuple de tuples de tuple….
Chaque sous-arbre est représenté par un tuple.

III. A vous de jouer !



1. Implémentez le code pour construire cet arbre.
2. Créer la fonction **`hauteur(arbre)`** qui permet de calculer la hauteur de cet arbre.
3. Créer la fonction **`taille(arbre)`** qui permet de calculer la taille de cet arbre.
4. Créer la fonction **`parcours_infixe(arbre)`** qui permet de suivre le parcours infixe de cet arbre.
5. Créer la fonction **`parcours_suffixe(arbre)`** qui permet de suivre le parcours suffixe de cet arbre.

Corrigé partiel :

```
def HAUTEUR(arbre) :
    if arbre!=None:
        rac= arbre.get_valeur()
        n1= arbre.get_gauche()
        n2= arbre.get_droit()
        return(1+max(HAUTEUR(n1),HAUTEUR(n2)))
    else:
        return 0
```

```
def taille(arbre) :
  if arbre!=None:
    n1= arbre.get_gauche()
    n2= arbre.get_droit()
    return( 1 + taille(n1) + taille(n2))
  else:
    return 0
```

Liens et ressources

Sites internet :

- https://pixees.fr/informatiquelycee/n_site/nsi_term_structDo_arbre.html
- https://pixees.fr/informatiquelycee/n_site/nsi_term_paraProg_poo.html
- https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_arbre.html
- https://www.youtube.com/watch?v=WvGiuIE1ktc
- http://math.univ-lyon1.fr/irem/IMG/pdf/parcours_arbre_avec_solutions.pdf
- http://math.univ-lyon1.fr/irem/Formation_ISN/formation_recursivite/arborescence/arbres.html