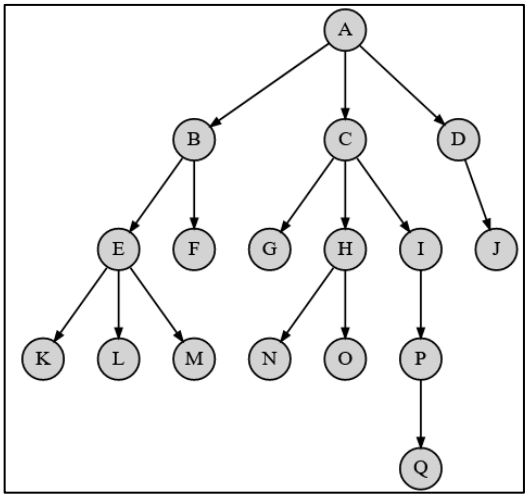
Arbres & Arbres binaires

I : Introduction : notion d'arborescence

Les structures arborescentes ont depuis très longtemps été utilisées pour représenter des informations présentant des *dépendances hiérarchiques* entre elles (une information stockée dans une arborescence a *une seule* information *juste au-dessus* d'elle). En annexe ont été indiqués quelques exemples :

- arbre phylogénétique en sciences du vivant,
- organisation d'un système de fichiers,
- représentation Document Object Model pour les fichiers HTML ou XML,
- descendance d'un individu (Clovis en l'occurrence),
- ascendance d'un individu (Antoine de Gargan en l'occurrence) etc.

Une autre utilité des arborescences est de pouvoir représenter les évolutions de certaines situations logiques (voir l'exemple du jeu du morpion en annexe). Pour de tels exemples, si on dispose d'algorithmes pour explorer des arborescences alors on est en mesure d'anticiper ce qui peut se passer si on effectue tel ou tel choix. Autrement dit, si on sait explorer des arborescences, on est en mesure de prendre la meilleure décision (ou la moins mauvaise).



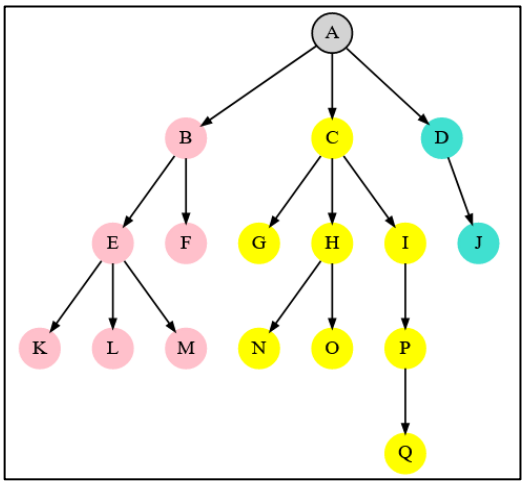
Voyons maintenant comment a été formalisée cette notion d'arborescence.

La première remarque que l'on se fait est qu'il n'y a aucune régularité : on a donc tendance à se dire que ce sera difficile à formaliser : le nombre de branches à chaque nœud est variable, la longueur des branches est variable, bref on a l'impression qu'une arborescence, ça part dans tous les sens (c'est d'ailleurs ce que vous avez en tête quand vous jouez au morpion : vous réfléchissez et vous sentez bien qu'il y a plein de possibilités qui s'offrent à vous à chaque coup).

A contrario, pour les structures de données (SDD) que l'on connait jusqu'ici :

- Dans les tableaux, les listes : les données sont bien rangées en ligne, on peut les parcourir avec une boucle en itératif (ou en récursif pour les listes).

- Dans les tables ou les matrices, les informations sont bien rangées également (plutôt en carré qu'en ligne), et on arrive facilement à penser des algorithmes itératifs opérant sur ces SDD.



La seconde remarque – qui permet d'appréhender correctement les arborescences – est la suivante :

> Une arborescence ce n'est jamais qu'une racine (le nœud A sur l'exemple ci-contre) à laquelle sont attachées d'autres arborescences (les trois arborescences ayant pour racines B, C et D sur l'exemple ci-contre). Autrement dit une arborescence est – comme les listes – intrinsèquement récursive et son élément de base est le Nœud (tout comme la Cellule est l'élément de base des listes).

Vocabulaire – Définitions :

Une *arborescence* est un ensemble *non vide* de *nœuds* qui sont organisés de la façon suivante :

- Un nœud particulier R est appelé la *racine* de l'arborescence.
- Les autres nœuds sont partagés en *n* sous-ensembles distincts qui forment autant d'arborescences appelées *sous-arborescences*.
- Le nœud racine R est relié aux *n* nœuds racines des sous-arborescences, *n* nœuds qui sont appelés les fils de la racine R.

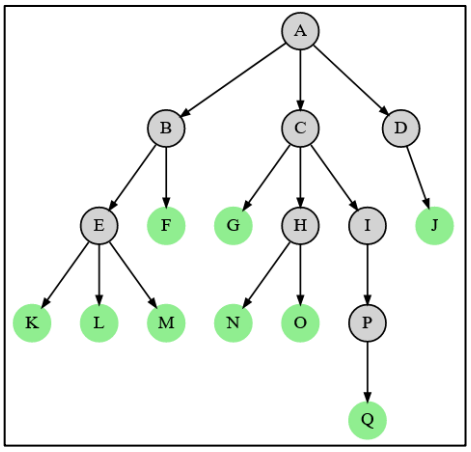
Sur l'exemple ci-dessus, l'arborescence comporte 3 sous-arborescences : BEFKLM, CGHINOPQ et DJ et le nœud racine A possède trois fils : les nœuds B, C et D.

- Les informations portées par les nœuds sont appelées les *étiquettes* : on parle parfois de *nœuds étiquetés*. Sur l'arborescence ci-contre, la racine a pour étiquette A.

- Un nœud qui ne possède aucun fils est appelé une *feuille* : sur l'arborescence ci-contre les nœuds K, L, M, F, G, N, O, P, Q et J sont des feuilles.

- La *hauteur* d'une arborescence est le plus grand nombre de nœuds rencontrés en descendant de la racine jusqu'à une feuille : l'arborescence ci-contre a pour hauteur 5.

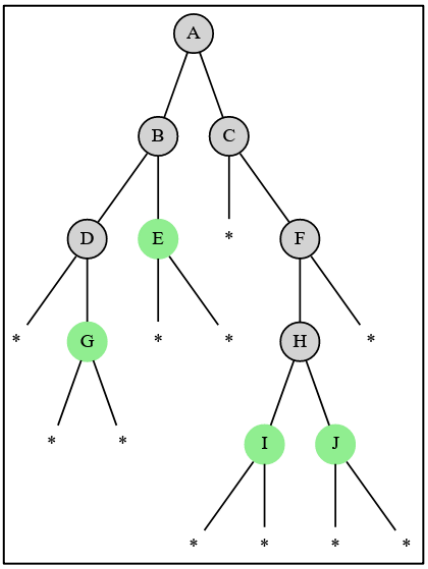
- La *taille* d'une arborescence est son nombre de nœuds : l'arborescence ci-contre est de taille 17.



Remarque 1 : la hauteur d'une arborescence est parfois définie comme "la longueur du plus grand chemin possible en descendant de la racine jusqu'à une feuille". Dans ce cas l'arborescence ci-dessus aurait pour hauteur 4.
Remarque 2 : il faut avoir conscience que sur la figure ci-contre il y a 17 nœuds mais aussi 17 arborescences.

II : Notion d'arbre binaire

En NSI, nous travaillerons que sur des arbres binaires qui présentent quelques différences avec la notion arborescences. Ces différences rendent plus simple l'implémentation des arbres binaires et des algorithmes associés.



II . 1 : Vocabulaire - Définition :

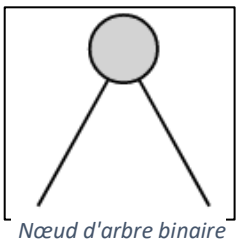
Un *arbre binaire* est :

- Soit l'arbre vide, c'est-à-dire qu'il ne contient aucun nœud,
- Soit un arbre non vide auquel cas :
    - Un nœud particulier R est appelé la *racine* de l'arbre binaire.
    - Les autres nœuds sont séparés en *deux* sous-ensembles qui forment récursivement deux *sous-arbres* appelés respectivement *sous-arbre gauche* et *sous-arbre droit*.
    - Le nœud racine R est relié à *deux* sous-arbres gauche et droit et, plus précisément, lorsqu'ils ne sont pas vides à leurs nœuds racines qui sont appelés les *fils* de la racine R.

On définit les notions de feuille, de hauteur et de taille comme pour les arborescences (l'arbre vide est de hauteur 0).

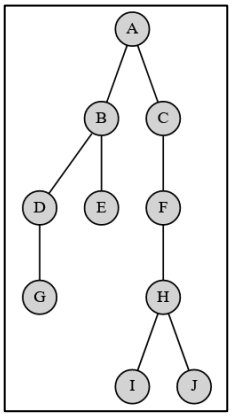
Différences avec la notion d'arborescence :

- Un arbre binaire peut être vide.
- Un nœud possède systématiquement deux sous-arbres, éventuellement vides.
- Un nœud possède toujours 0, 1 ou 2 nœuds fils.
- La position gauche-droite des nœuds a une importance alors que ce n'était pas le cas pour les arborescences.

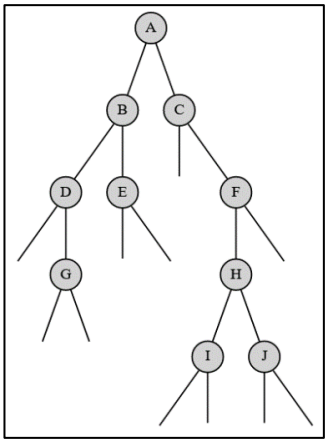

Nœud d'arbre binaire

Remarque :
Pour bien distinguer les sous-arbres droit et gauche des nœuds, on peut représenter les deux liaisons qui descendent d'un nœud même lorsqu'elles conduisent à un sous-arbre vide.

Cela évite de se demander si tel ou tel nœud est le fils gauche ou le fils droit du nœud situé au-dessus. Par exemple sur la figure de droite, il est impossible de savoir si le nœud G est le fils gauche ou le fils droit du nœud D. Au contraire c'était tout à fait possible sur la figure située plus haut.



Exercice :
- Donner la hauteur, la taille et les noms des feuilles de l'arbre binaire ci-contre.



- Quels sont les deux nœuds qui, si on les échange, modifient l'arbre binaire mais pas l'arborescence associée ?

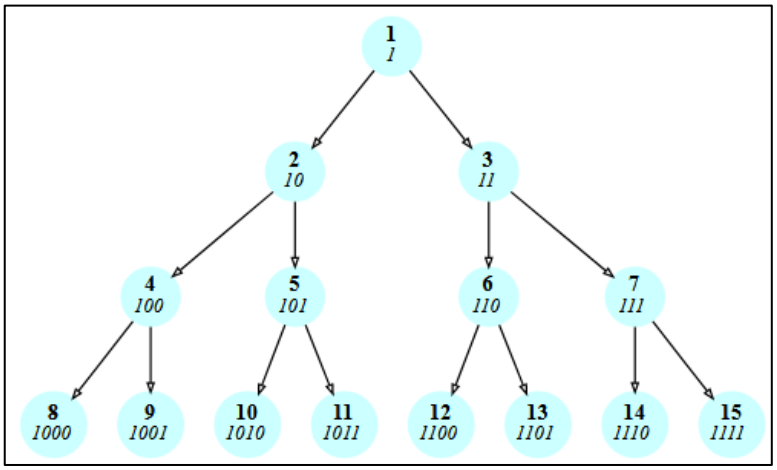
I . 2 : Evaluation de quelques mesures des arbres binaires

Soit N la taille d'un arbre binaire et h sa hauteur.
On sait que N vérifie :

$$h \leq N < 2^h$$

Ce qui revient à dire que h vérifie :

$$\log_2 N < h \leq N$$



Démonstration :
Pour : $h \leq N$, c'est immédiat par définition de la hauteur.

Pour : $N < 2^h - 1$, regardons l'arbre représenté ci-contre. On y a numéroté les sommets en binaire en commençant à 1 pour la racine et en rajoutant 0 à l'écriture binaire pour les fils gauche et en rajoutant 1 à l'écriture binaire pour les fils droit.

Ainsi sur un arbre binaire de hauteur h complètement rempli comme celui-ci, le nombre maximal N_{max} de nœuds est :

$$N_{max} = \underbrace{111 \ldots 111}_{h \text{ bits}}$$

On a donc : $N \leq N_{max} = 2^h - 1 < 2^h$ puisque le plus grand nombre codable sur h bits est $2^h - 1$.
Ou alors : $\log_2 N \leq \log_2 N_{max} < h$ puisqu'en général pour coder un entier X il faut k bits avec : $\log_2 X < k \leq \log_2 X$.

Remarque : si la définition de la hauteur est celle où la hauteur de la racine est zéro (alors que nous avons retenu la définition où la hauteur de la racine est 1), alors on a : $h + 1 \leq N < 2^{h+1}$ et : $\log_2 N < h + 1 \leq N$

III : Implémentation d'une classe pour des arbres binaires

Nous avons retenu une méthode élémentaire mais assez efficace en nous basant, comme pour les listes, sur une classe Nœud qui dispose de trois attributs : valeur, gauche et droit. Voici un exemple d'utilisation de cette classe.

On voit qu'on peut procéder de deux façons :

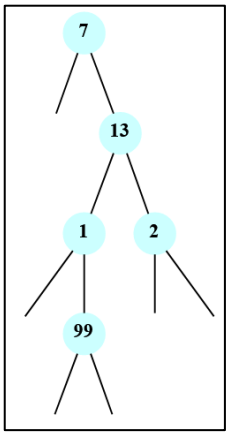
- On peut construire un arbre binaire en une seule instruction assez longue,

```
class Noeud:

    def __init__(self, v, g, d):
        self.valeur = v
        self.gauche = g
        self.droit = d
```

- On peut aussi muter un arbre binaire déjà construit en modifiant les attributs `gauche` ou `droit` d'un de ses nœuds. Cette façon de faire permet de construire un arbre au fur et à mesure, ce qui est nécessaire en pratique (lorsqu'on génère programmatiquement un arbre).

```
mon_arbre_b = Noeud(7,
                    None,
                    Noeud(13,
                          Noeud(1,None,None),
                          Noeud(2,None,None)))
n = Noeud(99, None, None)
mon_arbre_b.droit.gauche.droit = n
```



IV : Algorithmes élémentaires sur les arbres binaires

Les algorithmes sur les arbres binaires reposent fondamentalement sur leur structure récursive. En pratique ces algorithmes s'appuient très souvent sur :

- Le cas de base qui est l'arbre binaire vide,
- Une formule de récursivité du type :

```
algo(arbre) <--> arbre.valeur, algo(arbre.gauche), algo(arbre.droite)
```

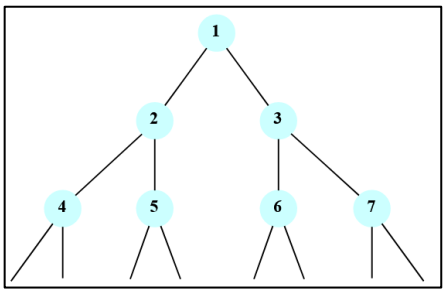
Par conséquent lorsqu'on cherche à programmer un algorithme sur un arbre binaire il faut chercher à relier le résultat qu'on obtiendrait pour l'arbre entier aux deux résultats qu'on obtiendrait pour ses deux sous-arbres.

Voici quelques exemples vus en TD/TP que l'on complètera à la main :

Calcul de la taille d'un arbre binaire	Calcul de la hauteur d'un arbre binaire
`def taille(arbre):`	`def hauteur(arbre):`
Calcul de la valeur maximale d'un arbre binaire d'entiers	**Calcul de la somme des valeurs d'un arbre binaire d'entiers**
`def maximum(arbre):`	`def somme(arbre):`

Nous avons par ailleurs vu en TP/TD un algorithme de copie d'un arbre binaire qui permet par exemple de construire un gros arbre binaire à partir de plusieurs copies d'un petit arbre binaire de sorte que les différentes copies sont indépendantes les unes des autres (sans copies indépendantes, une mutation d'un seul nœud du gros arbre binaire peut entraîner une mutation sur d'autres nœuds).

V : parcours des arbres binaires (en profondeur)



On prend l'exemple de l'arbre ci-contre que l'on notera `arbre_1234567`. Son sous-arbre droit sera par exemple noté `arbre_367`.

La fonction `afficher` ci-dessous permet de parcourir l'arbre binaire passé en argument et d'afficher les valeurs de tous les sommets de l'arbre. Un arbre peut-être parcouru dans l'ordre :

- *préfixe* : on traite d'abord le cas de sa valeur puis on traite les sous-arbres gauche et droit,
L'affichage est alors effectué dans cet ordre : 1, 2, 4, 5, 3, 6, 7.

```
def afficher(arbre):
    if arbre != None:
        print(arbre.valeur)
        afficher(arbre.gauche)
        afficher(arbre.droit)
```

- *infixe* : on traite le sous-arbre gauche, puis le cas de sa valeur puis le sous-arbre droit,
L'affichage est alors effectué dans cet ordre : 4, 2, 5, 1, 6, 3, 7.

```
def afficher(arbre):
    if arbre != None:
        afficher(arbre.gauche)
        print(arbre.valeur)
        afficher(arbre.droit)
```

- *postfixe* : on traite d'abord les sous-arbres gauche et droit puis on traite le cas de sa valeur.
L'affichage est alors effectué dans cet ordre : 4, 5, 2, 6, 3, 7, 1.

```
def afficher(arbre):
    if arbre != None:
        afficher(arbre.gauche)
        afficher(arbre.droit)
        print(arbre.valeur)
```

Cela est lié à la pile d'appels. Par exemple pour l'ordre infixe (les notations sont là pour vous faire comprendre ce qui se passe et ne correspondent pas à la réalité du code) :

`afficher(arbre_1234567)`	`afficher(arbre_1234567)`
	`afficher(arbre_245)`
		`afficher(arbre_1234567)`
		`afficher(arbre_245)`
		`afficher(arbre_4)`

`afficher(arbre_1234567)`
`afficher(arbre_245)`
`afficher(arbre_4)`
`afficher(None)`
	`afficher(arbre_1234567)`
	`afficher(arbre_245)`
	`afficher(arbre_4)`
	`print(4)`
		`afficher(arbre_1234567)`
		`afficher(arbre_245)`
		`afficher(arbre_4)`
		`afficher(None)`

`afficher(arbre_1234567)`
`afficher_arbre(245)`
`print(2)`
	`afficher(arbre_1234567)`
	`afficher(arbre_245)`
	`afficher(arbre_5)`
		`afficher(arbre_1234567)`
		`afficher(arbre_245)`
		`afficher(arbre_5)`
		`afficher(None)`

`afficher(arbre_1234567)`
`afficher(arbre_245)`
`afficher(arbre_5)`
`print(5)`
	`afficher(arbre_1234567)`
	`afficher(arbre_245)`
	`afficher(arbre_5)`
	`afficher(None)`
		`afficher(arbre_1234567)`
		`print(1)`

`afficher(arbre_1234567)`
`afficher(arbre_367)`
	`afficher(arbre_1234567)`
	`afficher(arbre_367)`
	`afficher(arbre_6)`
		`afficher(arbre_1234567)`
		`afficher(arbre_367)`
		`afficher(arbre_6)`
		`afficher(None)`

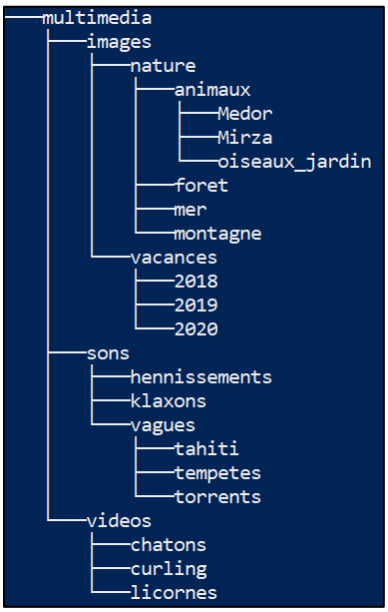
`afficher(arbre_1234567)`
`afficher(arbre_367)`
`afficher(arbre_6)`
`print(6)`
	`afficher(arbre_1234567)`
	`afficher(arbre_367)`
	`afficher(arbre_6)`
	`afficher(None)`
		`afficher(arbre_1234567)`
		`afficher(arbre_367)`
		`print(3)`

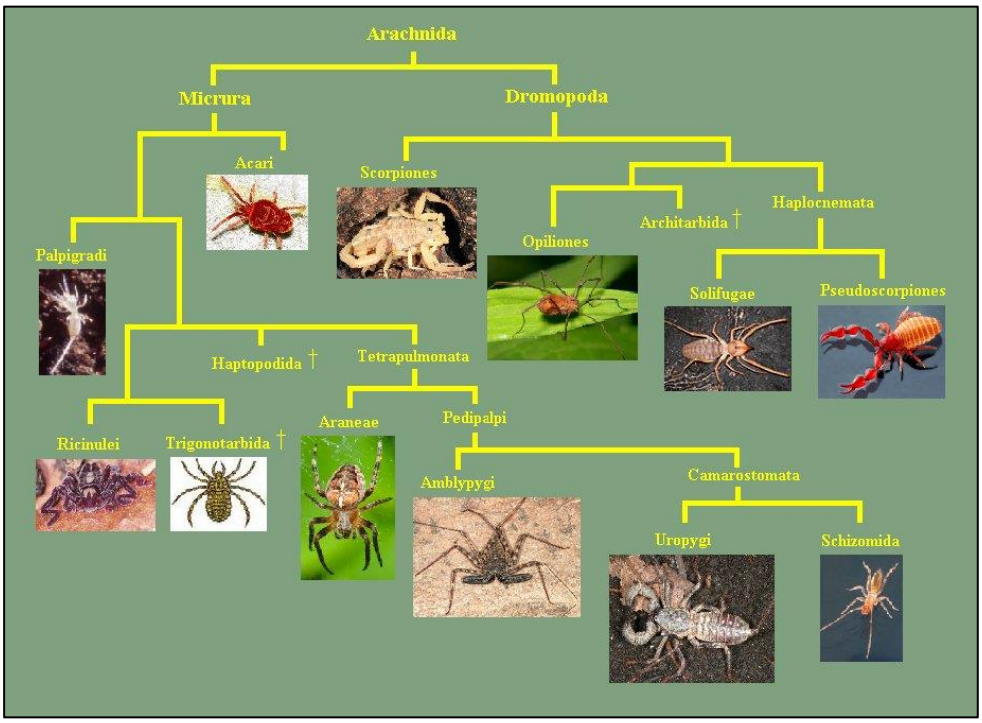
`...`		

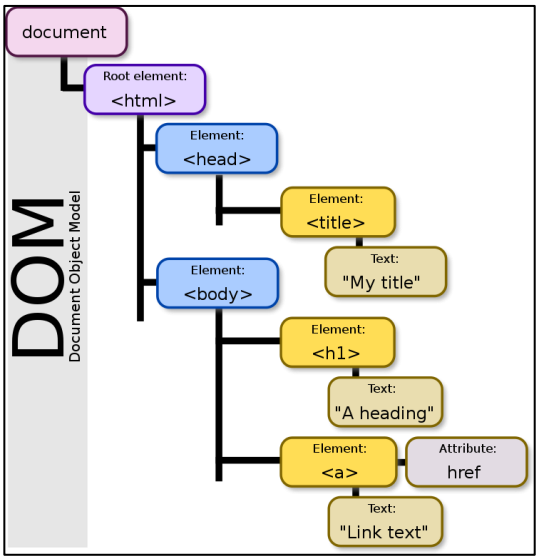
REMARQUE QUALITATIVE : lorsque la pile d'appel diminue cela revient à revenir en arrière/vers le haut dans le parcours d'arbre.

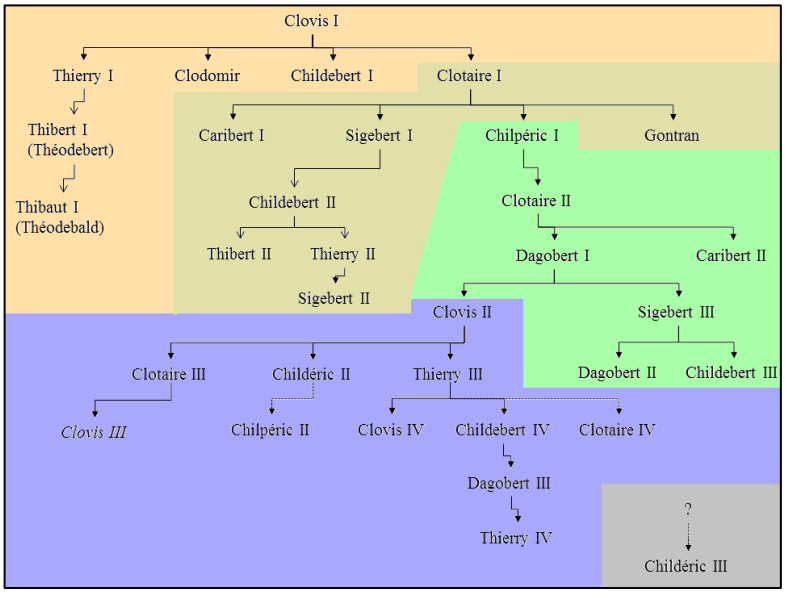
Ces trois parcours sont des *parcours en profondeur* : on va d'abord le plus profondément possible puis on remonte puis on repart le plus profondément possible puis on remonte puis on repart le plus profondément possible etc.
Nous verrons dans le cours suivant sur les arbres qu'il est également possible de faire un *parcours en largeur* pour lequel les nœuds sont parcourus dans l'ordre suivant : 1, 2, 3, 4, 5, 6, 7 (Formidable !).

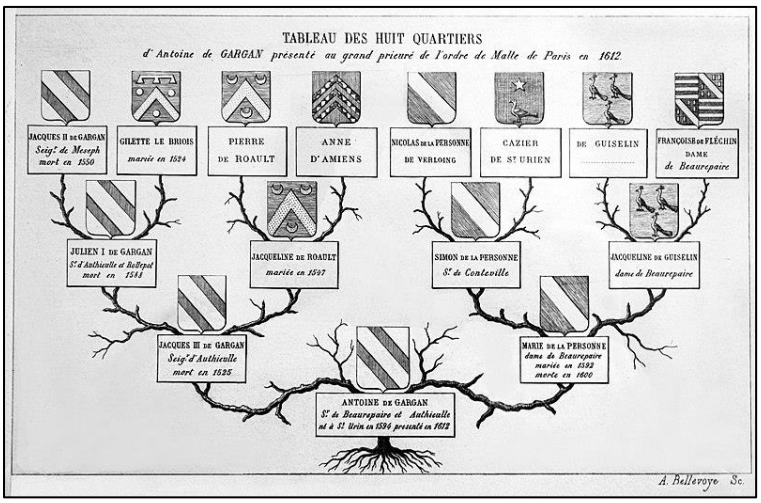
Annexe : images d'arbres divers (source : *Wikipédia Commons*)

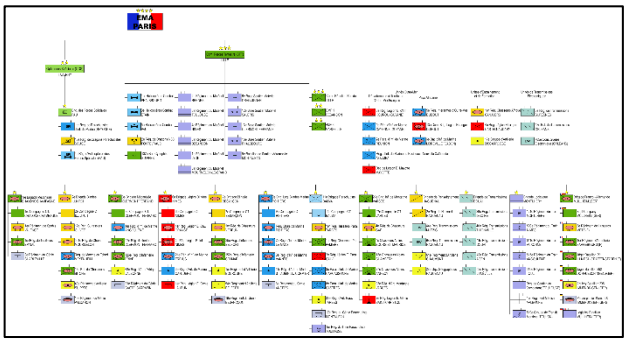






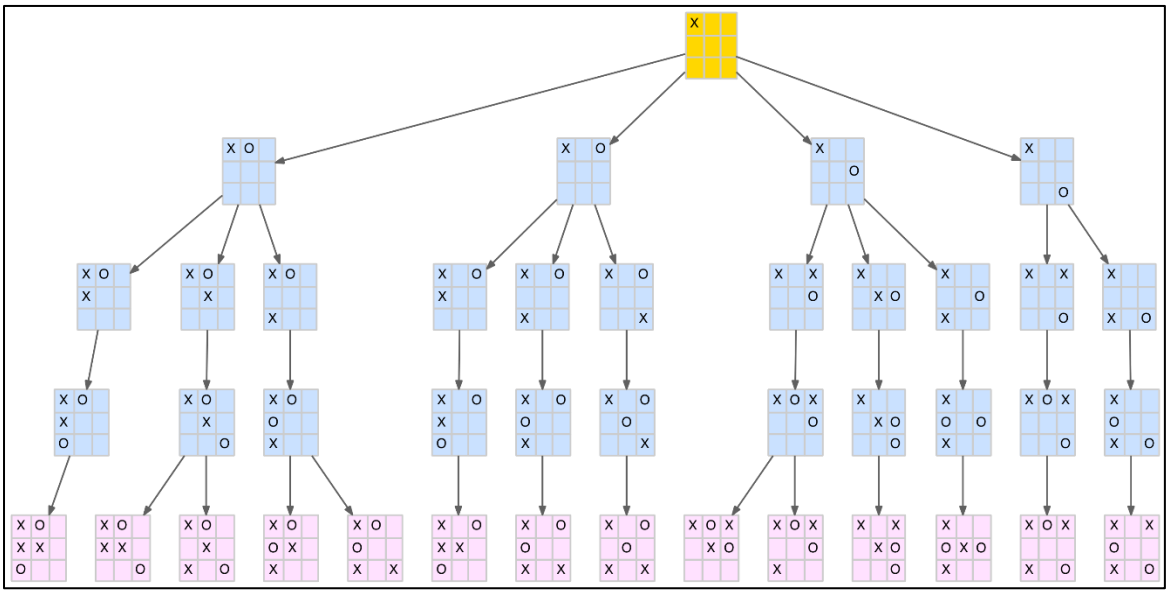




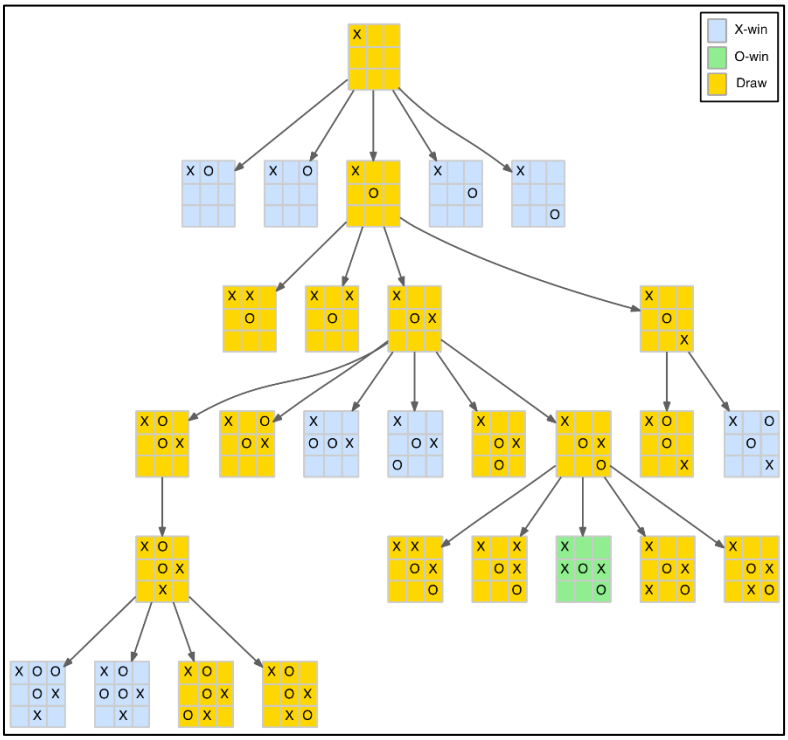


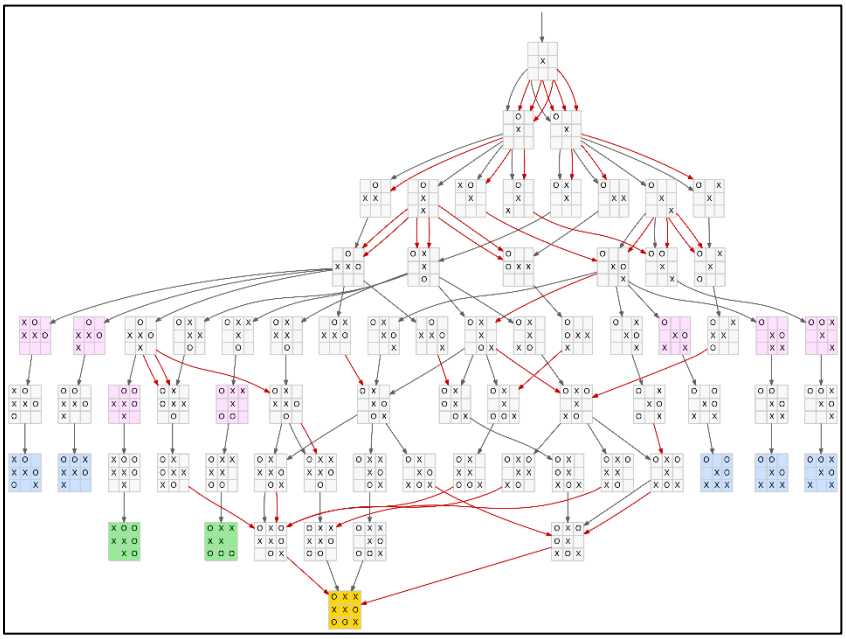
Organisation des unités de l'armée française

Source : http://www.occasionalenthusiast.com



Les évolutions possibles du jeu du Morpion lorsque le premier joueur X joue dans un coin et que le second joueur O joue ailleurs qu'au centre (figure de gauche) ou joue au centre (figure ci-dessous).





Ci-contre à gauche les cas possibles lorsque le premier joueur joue au centre.

Les flèches rouges désignent des opérations de rotation du plateau.

Par ailleurs sur cette image l'arborescence a été *"réduite"* : pour ne pas redessiner plusieurs fois les mêmes morceaux de l'arborescence, l'auteur fusionne les plateaux de jeu lorsqu'ils sont identiques ce qui explique que plusieurs flèches peuvent conduire au même nœud.

Cette image de gauche correspond donc en réalité à un graphe, structure de données qui sera étudiée plus tard (ici pour réduire l'arbre, on fusionne les cas identiques ce qui conduit à un graphe).