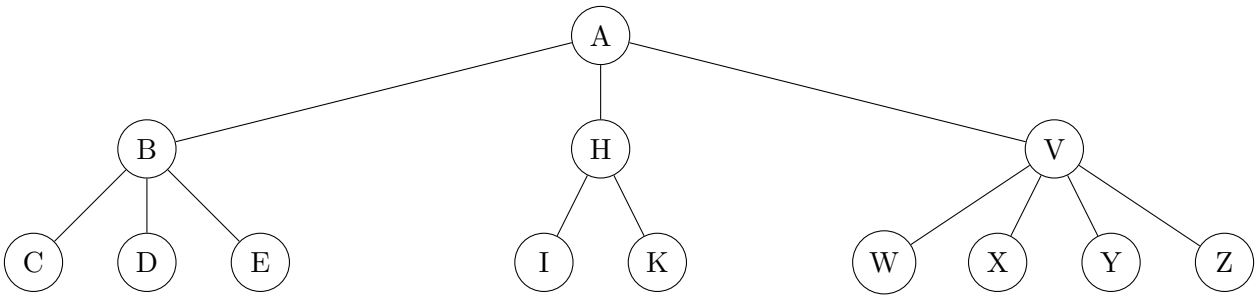
Arbres Binaires

Structure, Métriques, et Manipulation

Ce document a pour objectif de vous familiariser avec une nouvelle structure de données algorithmique abstraite : les *arbres* (en anglais : *trees*). En particulier, nous y aborderons les *arbres binaires* (en anglais : *binary trees*), précisément, les *arbres binaires de recherche* ou *ABR* (en anglais : *binary search tree* ou *BST*).

Définition informelle d'une structure de données[1] : « *En informatique, une structure de données est une manière d'organiser les données pour les traiter plus facilement. Une structure de données est une mise en œuvre concrète d'un type abstrait* ».

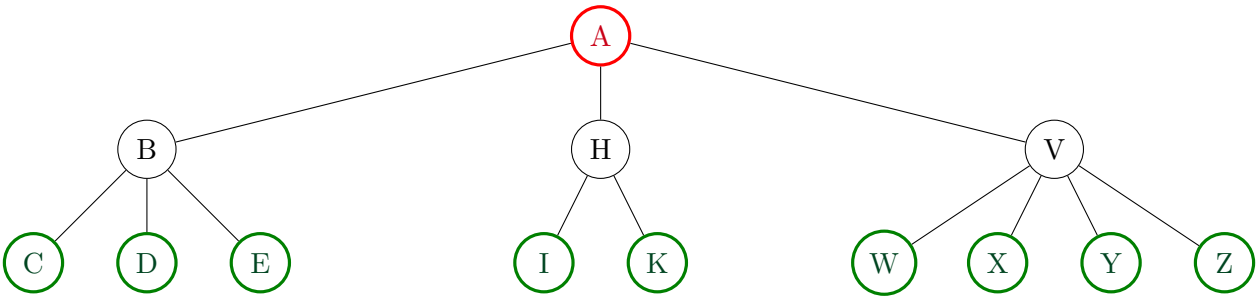
Les arbres, aussi appelés *arborescences*, sont des structures de données dites hiérarchiques : les données sont organisées sur plusieurs niveaux par des relations *parent-enfant*. La seule contrainte importante : il n'existe aucun lien pouvant créer de boucle (d'un fils à un autre du même niveau, ou vers un niveau au dessus).
Voici un exemple d'*arbre général* de hauteur 2 :



1 Terminologie

1.1 Arbres

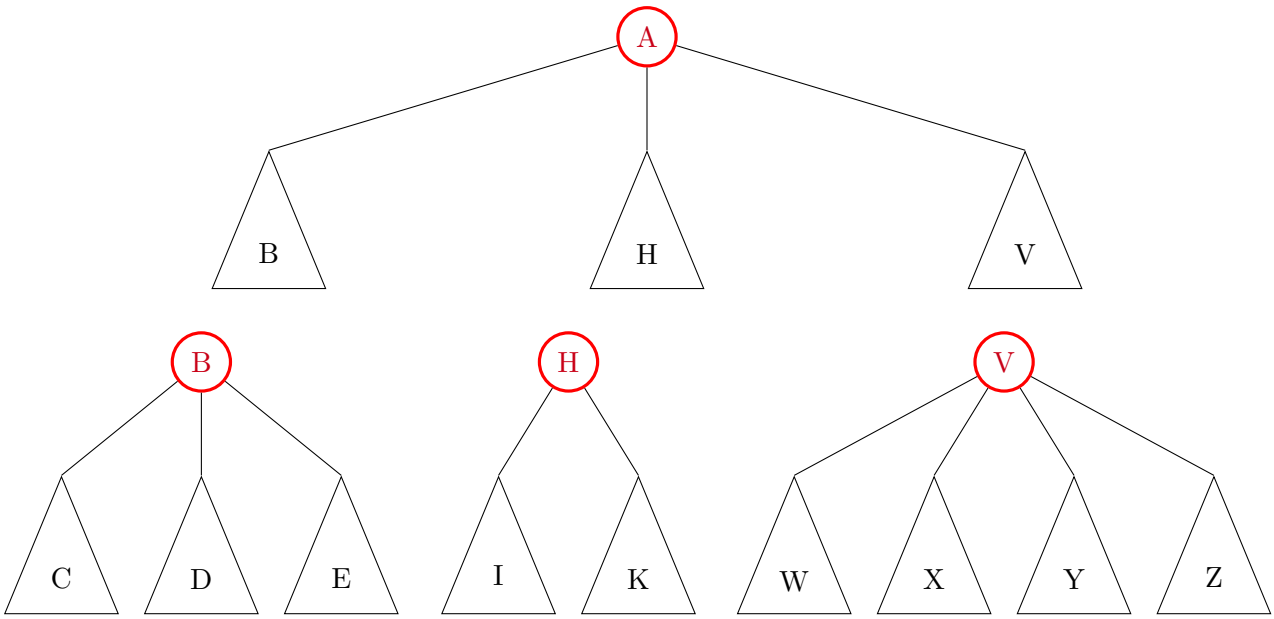
Les arbres sont composés de *nœuds* (les cercles dans le schéma, en anglais *nodes*) et de *liens* (les traits entre les cercles, en anglais *edges*). Avant d'aborder la terminologie précise, nous utiliserons une vision simplifiée des termes *racine* et *feuilles* pour mieux comprendre les arbres. La *racine* est le nœud tout en haut (le nœud en rouge sur le schéma, c'est-à-dire A) duquel descendent tous les autres, et les *feuilles* sont les nœuds tout en bas (les nœuds verts, c'est-à-dire C, D, E, I, K, W, X, Y, Z).



Vous avez très probablement déjà manipulé l'arborescence UNIX où les fichiers sont donc les feuilles d'un arbre dont la racine se nomme « / » . Dans l'arborescence Windows, chaque disque dur est présenté comme un arbre dictinct avec sa propre racine, et où les fichiers sont également des feuilles.

1. Wikipedia : Structure de données

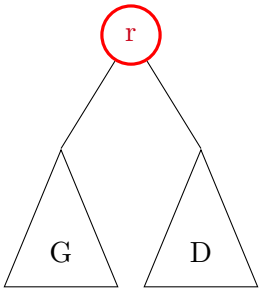
Les arbres sont également considérés comme des structures de données *récursives* dans le sens où la structure se répète plusieurs fois en se contenant elle-même. En effet, on peut observer une répétition de la structure d'arbre : l'arbre que l'on manipule est simplement un nœud servant de racine auquel d'éventuels liens renvoient vers des *sous-arbres*, eux-mêmes composés d'une racine chacun et d'éventuels sous-arbres.



Cette vision récursive des arbres permet de voir les caractéristiques essentielles pour former la structure : il faut un nœud racine, et des liens vers des sous-arbres appelés *fils* (en anglais : *children)*. La racine d'un nœud est également appelée *père* (en anglais : *parent*). Chaque nœud a au plus un père (pour éviter de créer des boucles).
La quantité de fils que chaque nœud peut porter au maximum s'appelle *l'arité* (ou le *degré*) : un arbre *N-aire* (de *degré N*) a des nœuds pouvant contenir jusqu'à N fils. Vous pouvez trouver dans la littérature des mentions d'octrees dans le cas d'un arbre 8-aire/de degré 8 où un nœud peut avoir jusqu'à 8 fils, ou d'autres types d'arbres (quadtrees, arbres ternaires, ...).

1.2 Arbres Binaires

Dans le cadre de ce cours, nous nous concentrerons sur les *arbres binaires* (en anglais : *binary trees*), c'est-à-dire les arbres d'arité 2 (ou de degré 2), donc disposant au maximum de 2 fils par nœud. Ainsi, nous nous intéresserons en particulier aux arbres de cette forme :



Les arbres binaires sont donc formés d'une racine, et de nœuds comportant 2, 1, ou 0 fils. On utilise couramment les dénominations *fils gauche* et *fils droit* pour parler de chacun des fils d'un nœud.

Pour préciser la terminologie précédente, un arbre est composé de :

— *feuilles*, c'est-à-dire des nœuds sans aucun fils
— *nœuds internes*, c'est-à-dire des nœuds avec un ou plusieurs fils

La racine devenant ainsi un cas spécial de nœud interne n'ayant aucun parent.

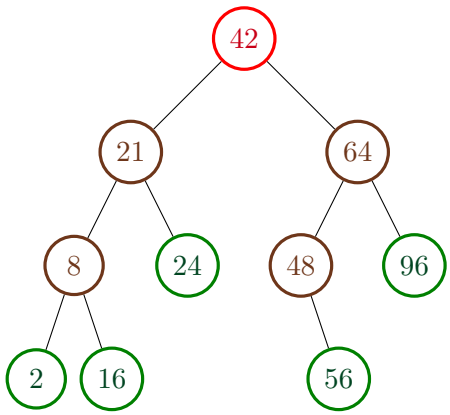
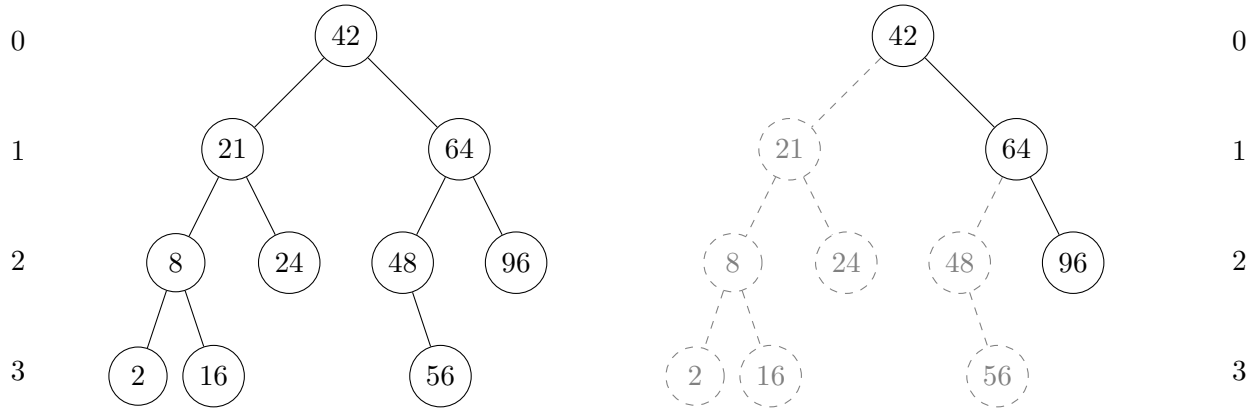


Fig.1 : Arbre binaire

Dans l'exemple Fig.1, on trouvera donc en nœuds internes : 8, 21, 42, 48, 64, sachant que 42 est également la racine, et on trouvera en feuilles : 2, 16, 24, 56, 96.
Toujours dans cet exemple, vous constaterez que tous les nœuds ne sont pas au même *niveau*. Cela ne retire pas à l'arbre sa qualité d'*arbre binaire* : chacun de ses nœuds a *au plus* 2 fils.

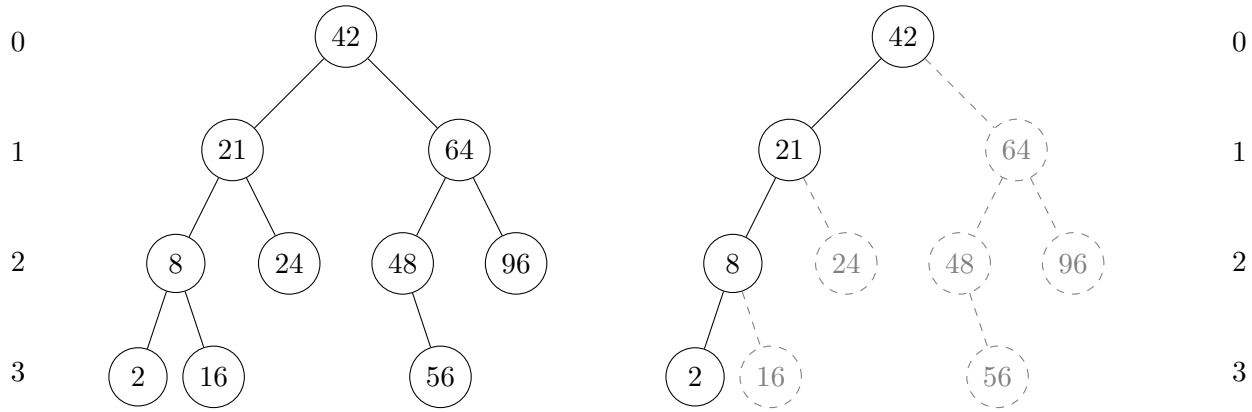
1.2.1 Profondeur

La *profondeur* (en anglais : *depth*) d'un nœud est le niveau auquel il se trouve (profondeur = niveau). Plus précisément, il s'agit de la différence entre le niveau du nœud et celui de la racine. La racine se trouve généralement au niveau 0 (mais certaines conventions peuvent indiquer 1).
Ainsi, la profondeur de 42 est 0, la profondeur de 64 est 1, la profondeur de 96 est 2, etc.



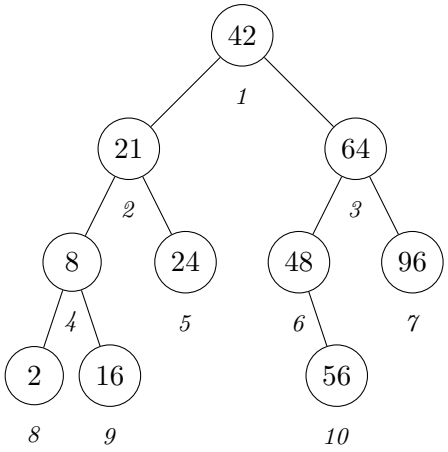
1.2.2 Hauteur

La *hauteur* (en anglais : *height*) d'un arbre est le niveau du plus profond de ses nœuds. Un arbre ne contenant qu'un seul nœud (la racine) a une hauteur de 0 (ou 1 dans certaines conventions).
Ainsi, la hauteur de l'arbre exemple est de 3.



1.2.3 Taille

La *taille* (en anglais : *size*) d'un arbre est le nombre de nœuds qu'il possède.
Ainsi, la taille de l'arbre est de 10.



1.3 Résumé

— Arbre/Arborescence (tree) : structure de données récursive organisée comme une hiérarchie, parfois aussi appelée *arbre enraciné*
— Nœud (node) : élément stocké dans la structure
— Lien (edge) : lien de parenté entre deux nœuds

— Père/Parent (parent) : nœud hiérarchiquement au dessus
— Fils/Enfant(s) (children) : nœud(s) hiérarchiquement en dessous
— Arité/Degré (arity/degree) : nombre maximum de fils que chaque nœud peut avoir dans l'arbre

— Feuille (leaf) : nœud sans aucun fils
— Nœud interne (internal node) : nœud avec au moins un fils
— Racine (root) : nœud sans parent *(peut être une feuille si l'arbre n'a qu'un seul élément)*

— Profondeur/Niveau (depth/level) d'un nœud : différence entre le niveau d'un nœud et le niveau de la racine *(la racine étant généralement au niveau 0)*, ou plus simplement, le nombre de liens depuis la racine
— Hauteur (height) de l'arbre : niveau du nœud le plus profond de l'arbre
— Taille (size) de l'arbre : nombre de nœuds dans l'arbre

2 Typologie d'Arbres Binaires

Les arbres binaires sont donc des arbres de degré 2, c'est-à-dire où chaque nœud a au plus 2 fils.
Il existe plusieurs cas spécifiques d'arbres binaires que nous allons voir avec des illustrations.

2.1 Arbre binaire strict - Arbre binaire localement complet

Un *arbre binaire strict* ou *arbre binaire localement complet* (en anglais : *full binary tree*, *proper binary tree*, *plane binary tree*, ou encore *strict binary tree*) désignent un arbre dont tous les nœuds ont exactement 0 ou 2 fils.
Chaque nœud de l'arbre est effectivement complet dans le sens où il dispose de 2 fils au maximum, ou aucun, mais toutes les feuilles ne sont pas nécessairement au même niveau. Aucun nœud ne dispose que d'un seul fils.
Avec un point de vue récursif, on se rend compte qu'un arbre binaire localement complet dispose donc de sous-arbres ayant exclusivement 2 ou aucun fils.

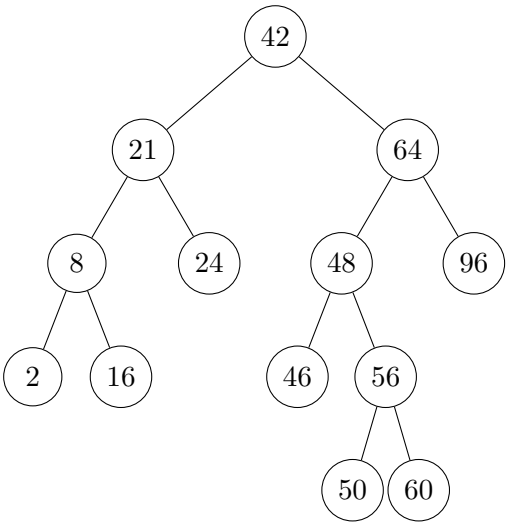


Fig.2 : Arbre binaire strict / Arbre binaire localement complet

2.2 Arbre binaire parfait

Un *arbre binaire parfait* (en anglais : *perfect binary tree*) désigne un arbre dont tous les nœuds ont exactement 0 ou 2 fils, et où toutes les feuilles sont au même niveau.
Il s'agit donc d'un arbre binaire localement complet, mais, avec toutes les feuilles au même niveau. Une autre façon d'exprimer ce cas serait que tous les niveaux de l'arbre sont remplis.

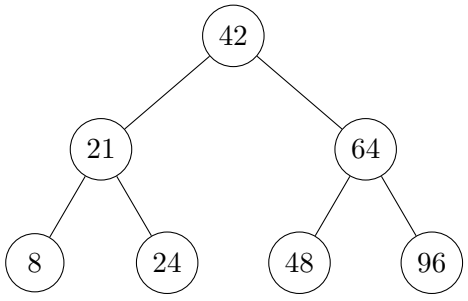


Fig.3 : Arbre binaire parfait

On notera que chaque niveau de l'arbre dispose au maximum de 2^{niveau} nœuds. Ainsi, au niveau 0 seule la racine peut exister, au niveau 1 il y aura les 2 fils de la racine, et ainsi de suite.

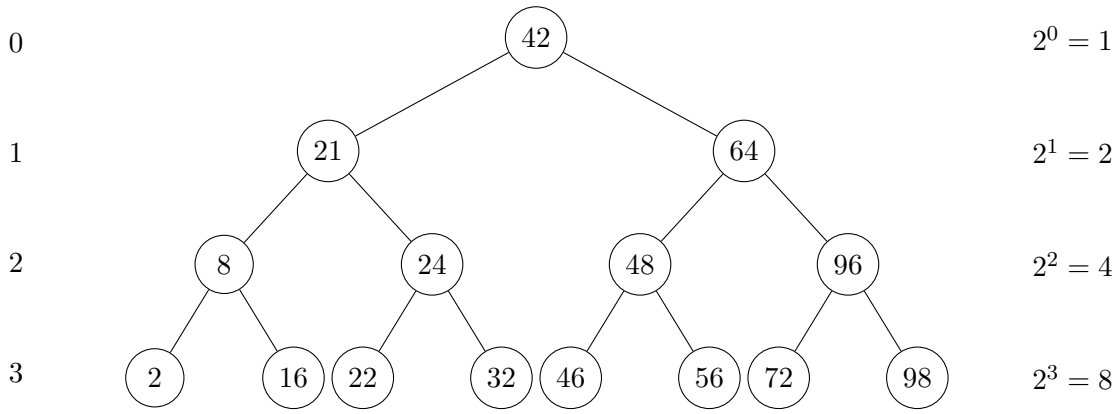


Fig.4 : Nombre de nœuds par niveau = 2^{niveau}

2.3 Arbre binaire (presque) complet

Un *arbre binaire complet* ou *arbre binaire presque complet* (en anglais : *complete binary tree* ou *almost complete binary tree*) désigne un arbre parfait dont le dernier niveau est éventuellement partiellement rempli, mais, dont les feuilles sont alignées à gauche.
Par définition, un arbre binaire parfait est donc un arbre binaire presque complet.
Il est important de noter la différence avec les abres binaires stricts/localement complets : ici, tous les niveaux sont remplis sauf *éventuellement* le dernier, et on peut n'avoir qu'un seul fils à un nœud de l'avant dernier niveau.

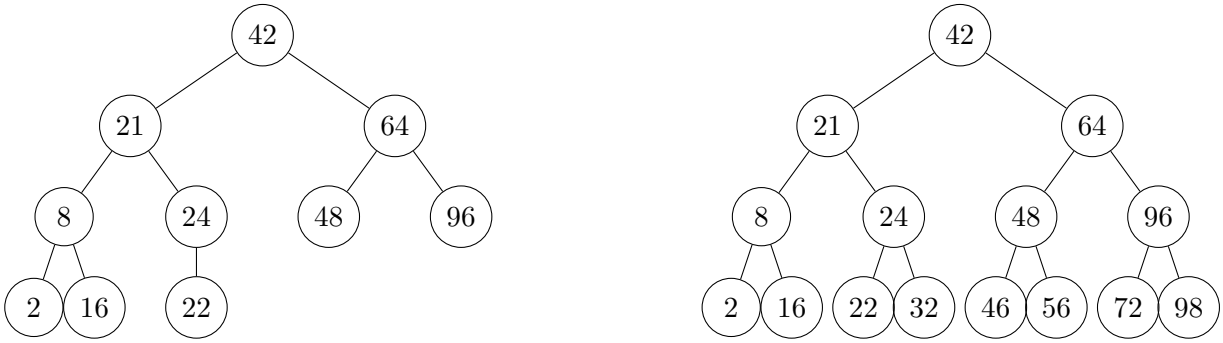


Fig.5 : Arbres binaires presque complets

2.4 Arbre filiforme

Un *arbre filiforme* ou *arbre dégénéré* (en anglais : *degenerated tree* ou *pathological tree*) désigne un arbre dont tous les nœuds sont des *points simples* (c'est-à-dire que chaque nœud a au plus 1 seul fils). De ce fait, la taille d'un arbre filiforme est exactement sa hauteur + 1.

Fig.6 : Arbres filiformes

2.5 Peignes

Les *peignes* (en anglais : *skewed trees*) désignent deux cas proches des arbres filiformes où l'arbre n'est développé que d'un seul côté jusqu'aux feuilles, mais tous les nœuds disposent de 2 fils.
L'arbre est donc localement complet, mais n'est développé en profondeur que sur sa gauche ou sa droite.
Le *peigne droit* (en anglais : *right skewed tree*) atteint sa profondeur maximale sur les fils droits uniquement, tout en ayant un fils gauche à chaque niveau. Le *peigne gauche* (en anglais : *left skewed tree*) est l'équivalent symétrique à gauche.

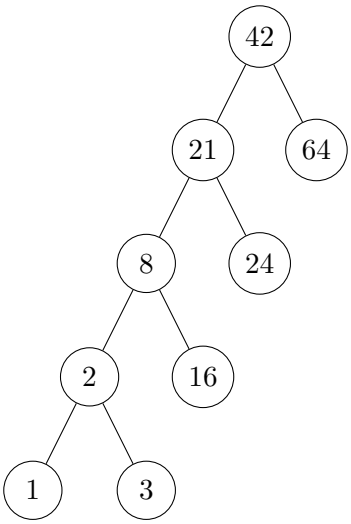


Fig.7 : Peigne gauche

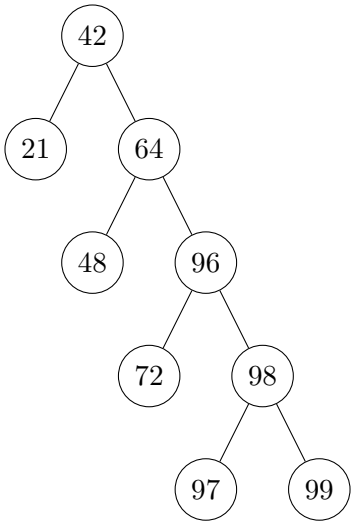
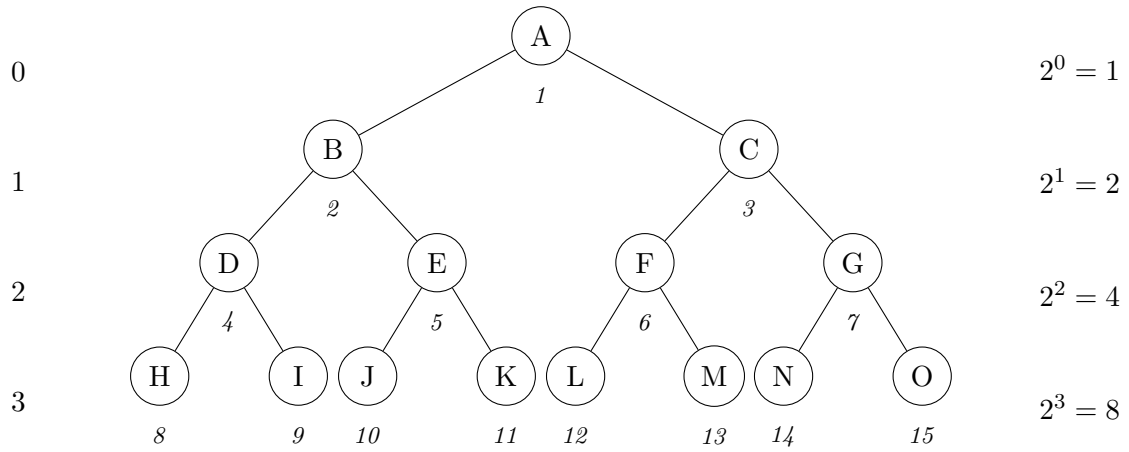


Fig.8 : Peigne droit

3 Implémentations des Arbres Binaires

3.1 Tableaux

La représentation sous forme de tableaux s'appuie sur la numérotation hiérarchique des nœuds, et donc des puissances de 2 pour représenter les niveaux. Ainsi la position de chaque nœud va permettre de déduire sa profondeur, et donc le numéro de la case où il se trouve dans le tableau.



position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
valeur	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
profondeur	*0*	*1*	*1*	*2*	*2*	*2*	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*

Une façon plus intuitive est de lire ce problème dans l'autre sens : lorsqu'un arbre est représenté sous la forme d'un tableau, on retrouve la position du nœud dans l'arbre à partir du numéro de sa case et des puissances de 2 (et donc du logarithme base 2).

— Le niveau 0 contient 1 nœud (la position 1, donc case 0)

— Le niveau 1 contient 2 nœuds (positions 2 3, cases 1 2)

— Le niveau 2 contient 4 nœuds (positions 4 5 6 7, cases 3 4 5 6)

— Le niveau 3 contient 8 nœuds (positions 8 à 15, cases 7 à 14)

La profondeur se retrouve avec un logarithme base 2 : niveau$(N) = \lfloor log_2(\text{position-hiérarchique}(N)) \rfloor$
Le nœud en position 11 se trouve niveau 3 : $log_2(11) = 3,45943 \Rightarrow \lfloor log_2(11) \rfloor = 3 \Rightarrow 2^3 \leq 11 < 2^4$
Il est le 4^e nœud du niveau (car on le décale de $11 - 2^3 = 3$ depuis le premier élément du niveau).

Encore plus intuitivement, il suffit de diviser par 2 le numéro de la position hiérarchique pour obtenir la position du nœud père. Pour un nœud en position hiérarchique i :

— son fils gauche est en position $2i$

— son fils droit est en position $2i + 1$

— son père est en position $\lfloor i \div 2 \rfloor$ *(partie entière par défaut de la division par 2)*

Attention : les numéros des cases du tableau démarrent à 0 et non pas 1, pensez au décalage de 1 entre le numéro de la case et la position hiérarchique

Par exemple pour F :

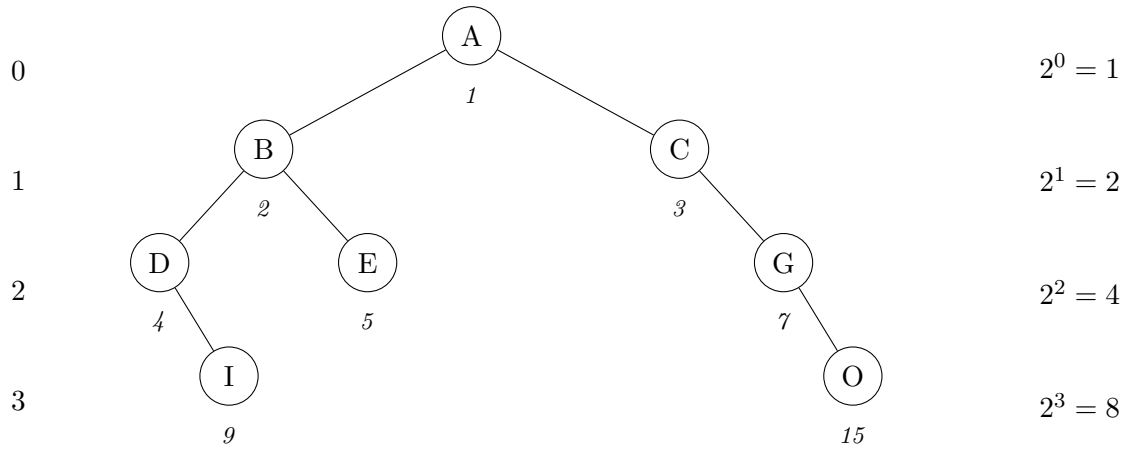
a) F est dans la case 5 du tableau, donc en position 6 (fils gauche)

b) $\lfloor 6 \div 2 \rfloor = 3$ le père de F est en position 3 (case 2)

c) $\lfloor 3 \div 2 \rfloor = 1$ le grand-père de F est en position 1 : la racine (case 0)

d) le fils gauche de F est en position 12 (case 11), et son fils droit en position 13 (case 12)

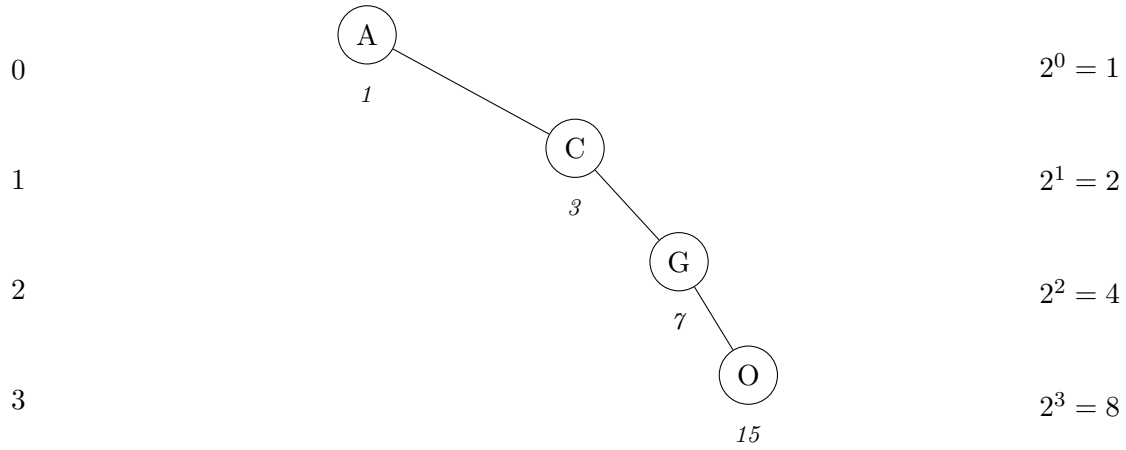
Comme vous le constatez, il s'agit d'une représentation idéale pour les arbres parfaits : comme ceux-ci ont tous leurs nœuds utilisés, le tableau est plein. Cependant, dans le cas où un arbre est incomplet, il est nécessaire de choisir une valeur spéciale qui n'est pas utilisée par les éléments stockés (par exemple, si on travaille avec des entiers naturels, on peut utiliser des valeurs négatives pour indiquer que le nœud n'existe pas).



position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
valeur	A	B	C	D	E	*-1*	G	*-1*	I	*-1*	*-1*	*-1*	*-1*	*-1*	O
profondeur	*0*	*1*	*1*	*2*	*2*	*2*	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*

Dans ce cas, on se rend compte que la perte d'espace n'est pas négligeable, et peut devenir pire si l'arbre est relativement profond : l'arbre est de taille 8, mais il consomme tout de même 15 cases en mémoire.

Dans le cas d'un arbre filiforme, étant donné qu'un seul fils est utilisé à chaque fois, la perte est extrême : une seule case par niveau n'est utilisée en pratique (pour un arbre de hauteur 3, on aura donc seulement 4 cases utilisées sur 15).



position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
valeur	A	*-1*	C	*-1*	*-1*	*-1*	G	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	O
profondeur	*0*	*1*	*1*	*2*	*2*	*2*	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*

3.2 Pointeurs

La représentation la plus commune des arbres binaires s'appuie sur les pointeurs en utilisant une structure copiant le modèle récursif :

```
struct node
{
  int          key;
  struct node *left_child;
  struct node *right_child;
};
```

On définit un arbre comme étant un nœud contenant une *clé* (l'élément à stocker), et des pointeurs vers le sous-arbre gauche et le sous-arbre droit (eux-mêmes des nœuds).

Vous pouvez également tout à fait définir un type *node* comme conteneur concret, et un type plus abstrait *bin_tree* pointant vers un nœud qui servira de racine et pouvant éventuellement contenir des informations utiles.

```
struct node
{
  int          key;
  struct node *left_child;
  struct node *right_child;
};

struct bin_tree
{
  struct node *root;
};
```

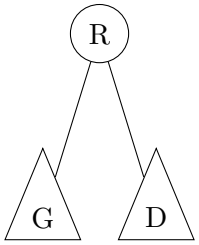
Il existe généralement deux façons de traiter les arbres binaires avec le point de vue récursif : une vision où l'on traite 2 cas (un arbre vide, ou un arbre disposant de deux sous-arbres), et une vision avec plusieurs cas plus explicites (racine vide, nœud sans fils, nœud avec fils gauche, nœud avec fils droit, nœud avec deux fils). Lorsque vous écrirez des algorithmes sur les arbres, vous devrez vous appuyer sur un de ces points de vue.

Dans la première vision, il s'agit donc de traiter deux cas typiques des traitements récursifs :

1. l'arbre courant est *NULL*
2. l'arbre courant existe et dispose donc de liens vers des sous-arbres (pouvant être *NULL* ou pas)

Ø

Arbre vide

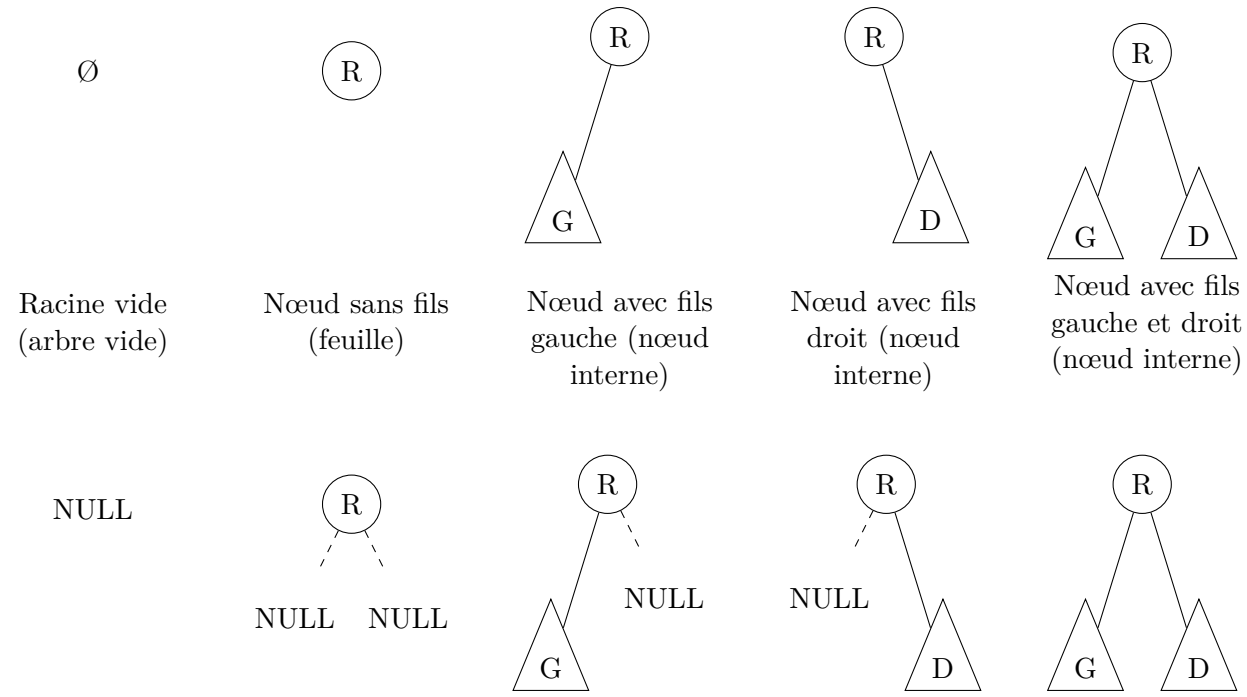


Arbre non-vide

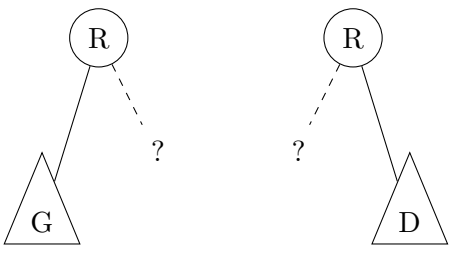
Dans la deuxième vision, il s'agit de traiter plusieurs cas avec un certain recul. On teste d'abord si l'arbre est vide, et ensuite seulement, on observe l'état du nœud courant et de ses éventuels fils pour y descendre par la suite :

1. **Racine vide** : comme dans les traitements avec une fonction chapeau, si l'arbre est totalement vide, on a probablement rien à faire (donc on traite ce cas avant tous les autres)
2. **Nœud sans fils** : le nœud courant est une feuille, il y a donc peut être un traitement particulier pour les nœuds externes
3. **Nœud avec un fils gauche** : le nœud courant dispose d'un fils gauche (nœud interne)
4. **Nœud avec un fils droit** : le nœud courant dispose d'un fils droit (nœud interne)
5. **Nœud avec fils gauche et droit** : le nœud courant dispose de ses deux fils (parfois ce cas est induit par les deux précédents, parfois il s'agit d'un cas à traiter en priorité)

Attention : tous ces cas n'ont pas à être énumérés dans tous les algorithmes s'appuyant sur cette vision ! Il s'agit seulement de tous les cas possibles, mais on peut par exemple avoir des traitements exclusifs aux nœuds ne disposant que d'un seul fils, ou au contraire à ceux disposant de deux fils.



Une dernière variante dans les algorithmes est de traiter chaque fils indépendamment de l'existence de l'autre. Si le fils gauche existe, on le traite, puis, si le fils droit existe, on le traite. Cette variante n'exclue pas les précédents cas : faire ces deux traitements n'induisent pas l'existence des deux fils en même temps.



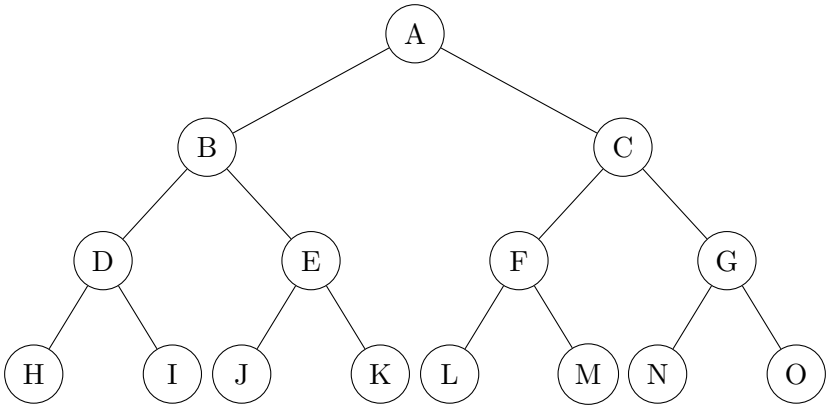
Ces différentes visions et ces différents cas ne sont présentés que pour vous aider à comprendre les algorithmes liés aux arbres, vous ne devez pas les apprendre par cœur.

4 Parcours

Les parcours d'arbres impliquent de passer dans chacun des nœuds pour accéder à leurs valeurs. Néanmoins, on peut visiter de plusieurs façons le même arbre : en descendant le plus profondément en priorité, ou au contraire, en passant prioritairement sur chaque nœud de chaque niveau.

4.1 Parcours Profondeur

Le *parcours profondeur* (en anglais : *depth-first search* ou *DFS*) consiste à descendre le plus profondément possible en priorité. Néanmoins, plusieurs parcours profondeurs sont également possibles : démarre-t-on par le fils gauche ou droit ?
Usuellement, on démarre par le fils gauche, ce qui en fait un *parcours profondeur main gauche* : tant que l'on peut descendre dans un fils gauche, on le fait, si un nœud n'a de fils qu'à droite, on y descend, si un nœud n'a pas de fils, alors on remonte d'un niveau, et on essaye le fils droit.



Lors d'un parcours profondeur, on passe sur chaque nœud 3 fois :

1. *Ordre préfixe* (*pre-order*) : avant de descendre vers le sous-arbre gauche
2. *Ordre infixe* ou *symétrique* (*in-order*) : à la remontée du sous-arbre gauche et avant de descendre vers le sous-arbre droit
3. *Ordre suffixe* ou *postfixe* (*post-order*) : à la remontée du sous-arbre droit

Le choix de l'ordre peut avoir une importance selon le problème étudié : doit-on traiter un nouvel élément dès sa première rencontre (parcours préfixe) ? ou au contraire la dernière fois que l'on passera dessus (parcours suffixe) ?

Le parcours profondeur main gauche pour chaque ordre donnera ces résultats :

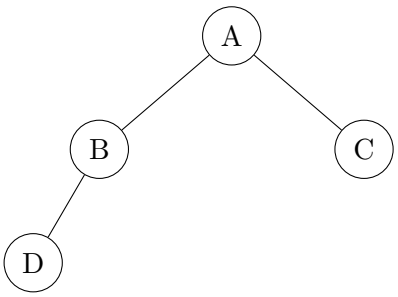
ordre préfixe : A-B-D-H-I-E-J-K-C-F-L-M-G-N-O
ordre infixe : H-D-I-B-J-E-K-A-L-F-M-C-N-G-O
ordre suffixe : H-I-D-J-K-E-B-L-M-F-N-O-G-C-A

Il est très important de noter que l'ordre préfixe n'est pas l'inverse de l'ordre suffixe.

Le parcours profondeur dans lequel on applique les 3 ordres donnerait :

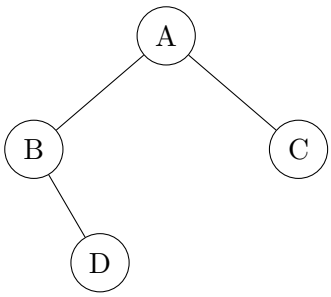
A- B- D- H- H- H- D- I- I- I- D- B- E- J- J- J- E- K- K- K- E- B- A-C- F- L- L- L- F- M- M- M- F- C- G- N- N- N- G- O- O- O- G- C- A

On remarque que les feuilles sont visitées 3 fois successivement : étant donné qu'il n'y a aucun fils, on arrive la première fois sur le nœud, on remonte du sous-arbre gauche vide, on remonte du sous-arbre droit vide, puis on repart immédiatement vers le père. Il faut néanmoins faire attention au cas des points simples : étant donné qu'un point simple a un seul fils, il faut vérifier s'il s'agit du fils gauche ou droit.



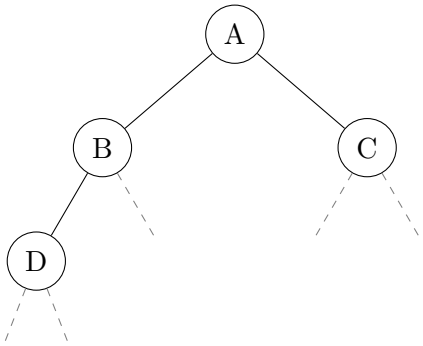
ordre préfixe : A-B-D-C
ordre infixe : D-B-A-C
ordre suffixe : D-B-C-A

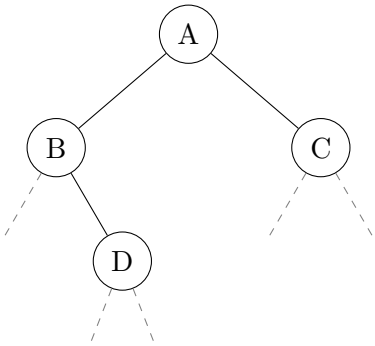
A- B- D- D- D- B- B- A- C- C- C- A



ordre préfixe : A-B-D-C
ordre infixe : B-D-A-C
ordre suffixe : D-B-C-A

A- B- B- D- D- D- B- A- C- C- C- A





On remarque dans cet exemple que la variation sera l'ordre de lecture de B et D dans l'ordre infixe.

L'implémentation la plus directe du parcours profondeur main gauche est récursive : on teste si le nœud est vide, puis on descend dans le sous-arbre gauche, puis on descend dans le sous-arbre droit.

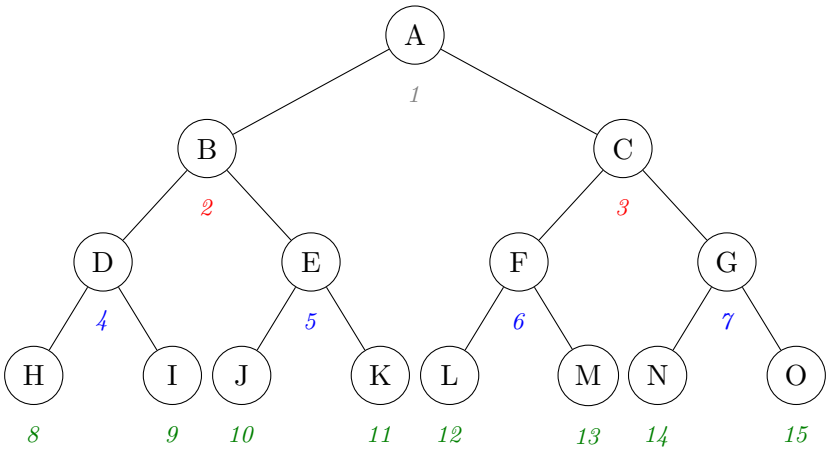
```
void Parc_Prof_Gauche(node *N)
{
  if (N == NULL)
  {
    printf("Le noeud est vide\n");
    return ();
  }

  printf("%d\n", N->key); /* Prefixe / avant fils gauche */

  Parc_Prof_Gauche(N->left_child);

  printf("%d\n", N->key); /* Infixe / ap. fils gauche & av. fils droit */

  Parc_Prof_Gauche(N->right_child);

  printf("%d\n", N->key); /* Suffixe / apres fils droit */
}
```

L'implémentation itérative implique l'usage d'une pile : on empile d'abord la racine, puis, tant que la pile n'est pas vide et selon l'ordre choisi, on traite l'élément en tête de la pile ou on empile les fils. L'ordre d'empilement entre les deux fils est important ! Pour respecter le principe du parcours profondeur main gauche, on empile d'abord le fils droit, puis ensuite seulement le fils gauche, afin d'accéder au fils gauche en premier lors du dépilement.

4.2 Parcours Largeur

Le *parcours largeur* ou *parcours par niveaux* (en anglais : *breadth-first search* ou *BFS*) consiste à parcourir les nœuds niveau par niveau. Tout comme pour le parcours profondeur : on peut démarrer à gauche ou à droite.
Usuellement, on démarre à gauche pour respecter *l'ordre hiérarchique*.

Comme vous l'aurez reconnu, l'ordre hiérarchique est celui employé par l'implémentation des arbres sous forme de tableaux. Ainsi, parcourir le tableau du premier au dernier élément revient à effectuer un parcours largeur.



L'implémentation du parcours largeur s'appuie sur une file : on enfile la racine, puis, tant que la file n'est pas vide, on défile la tête de file pour enfiler les fils dans l'ordre choisi (gauche puis droit, en général).

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en mars 2023. Certains exemples et illustrations sont inspirés des supports de cours de Nathalie "Junior" BOUQUET, et Christophe "Krisboul" BOULLAY, ainsi que de Wikipédia.