Khôlles d'informatique, Saison 1.

$$\text{Février} \xrightarrow[\text{programme de colle} \to \text{fin}]{} \text{Mars}$$

Résumé

Dans cet article de sciences scientifiques, on se propose de résoudre quelques exercices posés par Mme Ines Klimann (aka. inesKKK).

Questions de cours :
- — Algorithme du tri par tas, complexité.
- — Définir un tas, algorithme linéaire pour construire un tas, démontrer qu'il est en $O(n)$.
- — Compléxité dans le pire des cas d'un tri par comparaisons est en $\Omega(n \log n)$.
- — Percoler vers le bas, terminaison, complexité, correction.
- — Tri rapide, terminaison, correction, complexité, complexité en moyenne.
- — Tri rapide avec doublons.
- — Le parcours infixe d'un ABR donne la liste triée de ses éléments.
- — Parcours d'arbres.
- — Taille maximale d'un arbre de hauteur h, taille d'un arbre parfait.
- — Relation entre le nombre de feuilles et noeuds internes dans un arbre binaire strict.
- — Propriétés et fonctions sur les ARN.

Plus affichées au programme de colles :
- — Algorithme du codage de Huffman.

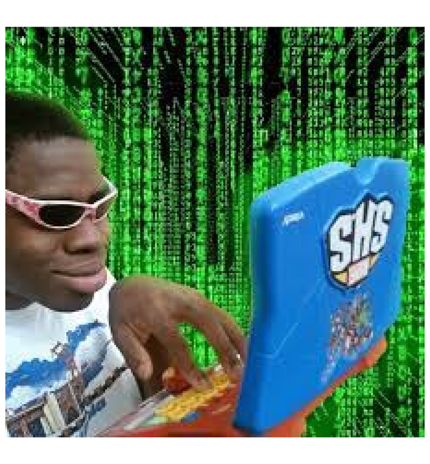

FIGURE 1 – Comment les MP2I qui écrivent en LaTeX se croient

Dénombrement d'arbres.

(Posé à Ibrahim)

Énoncé :
Dénombrer les arbres binaires de taille $n \in \mathbb{N}$, à étiquettes dans $[\![1, n]\!]$ deux-à-deux distinctes et vérifiant la condition de tas.

Lemme 0.1. *Le nombre de noeuds vides d'un arbre binaire de taille n est $n+1$.*

Preuve.
Pour un arbre réduit à sa racine, il y a deux fils possibles à la racine.
On suppose que pour un n fixé, il y a $n+1$ noeuds vides pour un arbre de taille n.
On fabrique un arbre de taille $n+1$ en remplaçant l'un de ses noeuds vides par un noeud non-vide.
Il nous reste donc n noeuds, plus les 2 fils vides de ce nouveau noeud : $n+2$ noeuds vides.
Par récurrence, le nombre de noeuds vides d'un arbre binaire de taille n est $n+1$.

Théorème 0.2. *Soit $n \in \mathbb{N}$. Le nombre N_n d'arbres distincts de taille n, d'éléments numérotés dans $[\![1, n]\!]$ deux-à-deux distincts et vérifiant la condition de tas est donné par la relation de récurrence suivante : $N_{n+1} = (n+1)N_n$, et la relation directe : $N_n = n!$.*

Preuve.
On considère, sans perte de généralité, le cas du tas min. Procédons par récurrence sur n.
$\Diamond$ *Initialisation.*
Il y a 1 arbre de taille 0 et 1 arbre de taille 1, ces arbres vérifient la condition de tas par définition et sont distincts.
On a bien que $N_1 = (0+1)N_0 = 1$.

$\Diamond$ *Hérédité.*
Soit $n \in \mathbb{N}$ tel que $N_n = nN_{n-1}$. Montrons que $N_{n+1} = (n+1)N_n$.
Soit $\mathcal{T}$ un tas de taille n. D'après le Lemme 0.1, ce tas a $n+1$ noeuds vides.
Pour créer un tas $\mathcal{T}'$ de taille $n+1$ à partir du tas $\mathcal{T}$, il suffit de remplacer l'un de ses noeuds vides par le noeud d'étiquette $n+1$.
On est assuré que ce nouvel arbre est un tas min puisque $n+1$ est plus élevé que n'importe quelle autre étiquette de $\mathcal{T}$.
Cette opération crée donc $n+1$ tas distincts de taille $n+1$ à partir de $\mathcal{T}$.
Cependant, par hypothèse de récurrence, il existe N_n arbres distincts vérifiant l'énoncé.
De surcroît, on peut créer $n+1$ tas de taille $n+1$ à partir de chacun d'entre-eux.
On a donc $N_{n+1} = (n+1)N_n$.

$\Diamond$ *Conclusion.*
Cette récurrence nous permet d'obtenir le terme général sur la suite du nombre de ces arbres.
On en conclut que $\forall n \in \mathbb{N},\ N_n = n!$

Arbres de Fibonacci (TD 14)

(Posé à Alex)

Énoncé :
On définit la suite d'arbres de Fibonacci $(F_n)_{n\in\mathbb{N}}$ par :

- F_0 et F_1 sont des feuilles.
- si $n \geq 0, F_{n+2}$ est l'arbre binaire possédant F_n et F_{n+1} comme sous-arbres.

1. Donner la hauteur de F_n.
2. Montrer que pour tout $n \geq 2$ et en tout noeud de F_n,
les hauteurs des sous-arbres gauche et droit différent d'au plus 1.
3. Déterminer le nombre de feuilles et de noeuds internes de F_n.

Notations : Pour un arbre $\mathcal{A}$, on note :
$>$ $\mathcal{H}(A)$ la hauteur de $\mathcal{A}$,
$>$ $\mathcal{F}(A)$ son nombre de feuilles,
$>$ $\mathcal{N}(A)$ son nombre de noeuds internes.

1. Montrons par récurrence que $\forall n \in \mathbb{N}^*,\ \mathcal{H}(F_n) = n-1$.
$\Diamond$ *Initialisation.*
On obtient immédiatement que $\mathcal{H}(F_1) = 0$.
De plus, $\mathcal{H}(F_2) = 1$, puisque F_2 a pour fils droit et gauche des feuilles (F_0 et F_1).

$\Diamond$ *Hérédité.*
Soit $n \in \mathbb{N}^*$ tel que $F_{n-2} = n-3$ et $F_{n-1} = n-2$. Montrons que $F_n = n-1$.
On a que F_n a pour fils gauche F_{n-2} et pour fils droite F_{n-1}, de hauteurs respectives $n-3$ et $n-2$.
Or, $\mathcal{H}(F_n) = \max(\mathcal{H}(F_{n-2}), \mathcal{H}(F_{n-1})) + 1 = n-2+1 = n-1$.

$\Diamond$ *Conclusion.*
Par récurrence, $\forall n \in \mathbb{N}^*, F_n = n-1$ et $F_0 = 0$.

2. Cas trivial pour $n \leq 1$.
Soit $n \geq 2$.
L'arbre F_n a pour fils gauche F_{n-2} et pour fils droit F_{n-1}.
D'après la question 1, on a : $\mathcal{H}(F_{n-2}) \leq n-2$ et $\mathcal{H}(F_{n-1}) \leq n-1$.
Ainsi, $\mathcal{H}(F_{n-1}) - \mathcal{H}(F_{n-2}) \leq n-1-n+2 \leq 1$.
Pour tout $n \geq 2$, la hauteur des sous-arbres gauche et droit de F_n diffère d'au plus 1.
De plus, pour tout noeud $\mathcal{A}$ de F_n, il existe $m \in \mathbb{N}$ tel que $\mathcal{A} = F_m$, cette propriété s'y applique.

3. Soit $(f_n)_{n\in\mathbb{N}}$ la suite de fibonacci numérique.
Pour $n \in \mathbb{N}$, notons $\mathcal{P}_n$: « $\mathcal{F}(F_n) = f_n$ et $\mathcal{N}(F_n) = f_n - 1$».
$\Diamond$ *Initialisation.*
Trivial pour F_0 et F_1.

$\Diamond$ *Hérédité.*
Soit $n \in \mathbb{N}$ tel que $\mathcal{P}_{n-1}$ et $\mathcal{P}_{n-2}$.
On a d'abord que $\mathcal{F}(F_n) = \mathcal{F}(F_{n-2}) + \mathcal{F}(F_{n-1}) = f_{n-2} + f_{n-1} = f_n$ car on somme les nombres de feuilles des sous-arbres droit et gauche de F_n.
De plus, $\mathcal{N}(F_n) = 1 + \mathcal{N}(F_{n-2}) + \mathcal{N}(F_{n-1}) = 1 + f_{n-2} - 1 + f_{n-1} - 1 = f_n - 1$.
En effet, la racine de F_n est un noeud interne, et on somme les nombres de noeuds internes du sous-arbre droit et gauche.

$\Diamond$ *Conclusion.*
On a montré $\mathcal{P}_n$ par récurrence pour tout $n \in \mathbb{N}$.

ABR : successeur, prédecesseur

(Posé à Aïcha)

Énoncé.
Dans un ABR, montrer que si un noeud a deux fils alors le successeur de sa valeur n'a pas de fils gauche, et son prédecesseur n'a pas de fils droit.

Soit un noeud d'ABR N, de valeur n. On suppose qu'il a deux fils.
Soit un noeud S, contenant la valeur s, successeur de n.
On suppose que ce noeud a un fils gauche, de valeur g, alors $g \leq s$ et $g \geq n$ car le noeud de g est à droite de N.
Ainsi, g est le successeur de n, ce qui est absurde car c'est s.
Soit un noeud P, contenant la valeur p, prédecesseur de n.
On suppose que ce noeud a un fils droit, de valeur d, alors $d \geq p$ et $d \leq n$ car le noeud de g est à gauche de N.
Ainsi, d est le prédecesseur de n, ce qui est absurde car c'est p.

Complexité de la construction d'un ABR

(Posé à Morgan)

Énoncé.
Montrer que tout algorithme construisant un ABR à partir d'une liste de taille n a une complexité dans le pire des cas en $\Omega(n \log n)$.

Soit A un algorithme de construction d'un ABR et T une liste de taille n.
On sait qu'un arbre est un ABR si et seulement si son parcours infixe donne la liste triée de ses éléments.
On veut donc ordonner T de façon à ce que ses éléments soient dans le bon ordre pour le parcours infixe, cela ramène au résultat du cours sur la complexité d'un tri par comparaisons : $\Omega(n \log n)$.

Nombre d'ABR filiformes

(Posé à Benjamin) (Je suis pas sûr de l'énoncé)

Énoncé.
Combien y a-t-il d'ordres initiaux pour un ABR dans lequel on insère tous les entiers de 1 à n ?

Un seul, filiforme vers la droite puisque à tout rang n, on insère le noeud noeud suivant en tant que fils droit du précédent car sa valeur est $n+1$.

Questions de cours :

Constuire tas.
Un tas est un arbre binaire tel que chacun de ses noeuds vérifie une certaine relation d'ordre avec ses fils.

Algorithme 1 : Construire tas
```
Entrées : Un tableau 𝒯
Sorties : 𝒯 tel qu'il vérifie la condition de tas.
pour i allant de ⌊n/2⌋ à 1 faire
 |  percoler_vers_le_bas(𝒯, i)
fin
```
Compléxité.
Soit H la hauteur du tas. On admet que percoler_vers_le_bas est en $O(H)$.
À une certaine profondeur p, il y a au plus 2^p noeuds.
De plus, ces noeuds sont à hauteur soit $h = H - p$, soit $h = H - p - 1$, donc $p = H - h$ ou $p = H - h - 1$.
La complexité de percoler_vers_le_bas sur un noeud de hauteur h est donc de αh, $\alpha \in \mathbb{R}$.
Ainsi, il y a au plus 2^{H-h} noeuds de hauteur h.
Alors, dans le pire des cas :

$$\sum_{h=0}^{H} \alpha h \cdot 2^{H-h} = \alpha 2^H \sum_{h=0}^{H} \frac{h}{2^h}$$

On pose $f : x \mapsto \sum_{h=0}^{H} x^h$, alors $f(x) = \sum_{h=0}^{H} x^h = \frac{x^{H+1}-1}{x-1}$.
De plus, on pose $g : x \mapsto xf'(x)$, et pour $x < 1$: $f'(x) = \frac{Hx^{H+1} - Hx^H + x^H + 1}{(x-1)^2} = \frac{1}{(x-1)^2} + Hx^H \frac{x-1-\frac{1}{H}}{(x-1)^2} \to \frac{1}{(x-1)^2}$.
Alors $g(x) \leq \frac{x}{(x-1)^2} + \beta$ en particulier pour $\frac{1}{2}$.

$$\sum_{h=0}^{H} \alpha h 2^{H-h} \leq \alpha 2^H (2 + \beta)$$

Alors $\alpha 2^H \sum_{h=0}^{H} \frac{h}{2^h} = O(2^H = n)$.

Tri par tas.

Algorithme 2 : Tri par tas
```
Entrées : Un tableau T
Sorties : T trié
construire_tas(T)
pour i allant de n à 1 faire
 |  échanger les cases 1 et i
 |  percoler_vers_le_bas(T, 1, T[i])
fin
```
Complexité :
Soit n la taille de T. On sait que construire_tas s'effectue en $O(n)$ et percoler_vers_le_bas en $O(\log n)$.
L'échange des cases et la décrémentation s'effectuent en $O(1)$.
La boucle for se termine, le variant est $n-1$, chaque itération se termine aussi et il y a n itérations.
On en déduit que cet algorithme est en $O(n \log n)$.

Codage de Huffman.

Algorithme 3 : Codage de Huffman
```
Entrées : Un texte
Sorties : Son encodage
mettre les couples (caractère, fréquence dans une chaîne de priorité)
tant que file contient plusieurs éléments faire
 |  extraire les 2 éléments de plus basses fréquences
 |  créer un noeud binaire dont ces éléments sont les fils
 |  insérer ce noeud dans la file, avec pour fréquence la somme des fréquences de ses fils
fin
renvoyer l'élément de la file
```

Complexité dans le pire des cas d'un tri par comparaisons.
Soit A un algorithme de tri par comparaisons et T un tableau de taille n et de hauteur h d'éléments supposés deux-à-deux distincts.
L'action de A est d'appliquer une permutation sur T en fonction de l'ordre relatif des éléments de T.
On modélise cela par un arbre binaire strict représentant le graphe de flot de contrôle de A sur T.
Cet arbre possède $n!$ feuilles, le nombre de permutations de T, de plus, on arrive à une feuille si le tri est terminé.
La complexité du tri est proportionnelle à la hauteur de l'arbre.
On a $n! \leq 2^h \Rightarrow \log_2(n!) \leq h$.
Et : $\log_2(n!) = \sum_{k=1}^{n} \log_2(k) \geq \sum_{k=n/2}^{n} \log_2(k) \geq \sum_{k=n/2}^{n} \log_2(\frac{n}{2}) = \frac{n}{2}\log_2(\frac{n}{2})$
Alors $h = \Omega(n \log_2(n))$.

Taille maximale d'un arbre de hauteur h.
Soit A_h un arbre de hauteur h. On note $\mathcal{N}(A_h)$ le nombre de noeuds de A_h.
Montrons par récurrence que $\mathcal{N}(A_h) \leq 2^{h+1} - 1$.
$\Diamond$ *Initialisation.* Pour $h = 0$, il n'y a que la racine alors $\mathcal{N}(A_0) = 1 \leq 2^{0+1} - 1 = 1$.
$\Diamond$ *Hérédité.* Soit $h \in \mathbb{N} \mid \forall n < h\ \mathcal{N}(A_n \leq 2^{h+1} - 1)$.
On note A_g et A_d son fils gauche et droit, de hauteurs inférieures à $h-1$.
Par hypothèse : $\mathcal{N}(A_h) = 1 + \mathcal{N}(A_g) + \mathcal{N}(A_d) \leq 1 + 2^h - 1 + 2^h - 1 = 2^{h+1} - 1$.
Par le principe de raisonnement par récurrence, la taille maximale d'un arbre de hauteur h est $2^{h+1} - 1$.

Taille d'un arbre parfait.
Soit A_h un arbre parfait de hauteur h. Montrons par récurrence que $\mathcal{N}(A_h) = 2^{h+1} - 1$.
$\Diamond$ *Initialisation.* Un arbre de hauteur 0 est réduit à sa racine et de taille $1 = 2^{0+1} - 1$.
$\Diamond$ *Hérédité.* Soit $h \in \mathbb{N}$ tel que $\mathcal{N}(A_{h-1}) = 2^h - 1$.
Les sous-arbres gauche et droite de A_h sont des arbres de hauteur $n-1$ par hypothèse d'arbre parfait.
Ainsi, $\mathcal{N}(A_h) = 1 + 2 \cdot (2^h - 1) = 1 + 2^{h+1} - 2 = 2^{h+1} - 1$
Par récurrence, la taille de tout arbre parfait de hauteur h est de $2^{h+1} - 1$.

Relation nombre de feuilles et nombre de noeuds internes dans un arbre binaire strict.
Nombre de liaisons père -> fils : $2 \cdot |\{\text{noeuds internes}\}|$.
Nombre de liaisons fils -> père : $|\{\text{noeuds}\}| - 1$.
Alors $2|\text{noeuds internes}| = |\{\text{noeuds}\}| - 1$.
Or $|\{\text{feuilles}\}| = |\{\text{noeuds}\}| - |\{\text{noeuds internes}\}|$.
Alors : $|\{\text{feuilles}\}| = |\{\text{noeuds internes}\}| + 1$

Parcours infixe d'un ABR.
Soit A un ABR de hauteur h et P_h :«Le parcours infixe de A donne une liste triée».
Initialisation. Pour $h = 0$ c'est trivial car l'arbre est réduit à sa racine.
Hérédité. Supposons P_h vrai pour toute hauteur strictement inférieure à h. Montrons P_h.
On note A_g le fils gauche de A, A_d son fils droit et r sa racine.
Par supposition, le parcours est correct sur A_g et A_d car ils sont de hauteurs strictement inférieures à h.
Le parcours infixe parcourt d'abord A_g, puis r, puis A_g.
On sait que tout élément de A_g est inférieur à r et que tout élément de A_d est supérieur à r par propriété des ABR.
Donc le parcours de A_g puis r puis A_d est dans l'ordre croissant.
P_h est donc vrai et par récurrence, pour tout h, P_h est vraie.

Percoler vers le bas.

Algorithme 4 : Percoler vers le bas
```
Entrées : Un tas A, un indice i et une valeur v
Sorties : Un tas similaire à A, tel que A[i] = v
A[i] ← v
max ← indice du noeud d'étiquette maximale entre A[i] et ses fils
si max ≠ i alors
 |  A[i] ← A[max]
 |  percoler_vers_le_bas(A, max, v)
fin
```
Terminaison.
Les lignes $1, 2, 3, 4$ se terminent. Le variant d'appel est la hauteur du noeud d'indice i.
Complexité.
Notons $T(h)$ le nombre d'opérations élémentaires pour une certaine hauteur h.
On a $T(0) = \alpha$ et $T(h) = \alpha + T(h-1)$ donc $T(h) = \alpha h = O(h)$.
Or $2^h \leq n \leq 2^{h+1} - 1$ avec n le nombre de noeuds de l'arbre.
Donc $h \leq \log_2(n) \leq h+1$: complexité dans le pire des cas en $O(n)$.
Correction.
Soit h la hauteur d'un noeud d'indice i.
Si $h = 0$, alors c'est une feuille et $max = i$: il n'y a pas d'appel.
Supposons que l'appel est correct pour une certaine hauteur $h-1$. Montrons que l'appel sur h fonctionne.
On prend un indice i d'un noeud de hauteur h.
Si $max = i$ alors par hypothèse de récurrence, les sous-arbres de i sont des tas et l'appel est correct.
Si $max \neq i$ alors on remplace $A[i]$ par le max et la condition est donc vérifiée entre i et ses fils.
Par hypothèse de récurrence, on sait que l'appel récursif est correct.

Tri rapide.

Algorithme 5 : Partitionner
```
Entrées : Tableau 𝒯, entier debut, entier fin
Sorties : 𝒯 partitionné et indice du pivot
pivot ← 𝒯[debut]
inf ← debut+1
sup ← fin
tant que true faire
 |  tant que 𝒯[sup] ≥ pivot et sup > debut faire
 |   |  sup ← sup - 1
 |  fin
 |  tant que 𝒯[inf] < pivot et inf < fin faire
 |   |  inf ← inf +1
 |  fin
 |  si inf ≥ sup alors
 |   |  break
 |  fin
 |  echanger(𝒯, inf, sup)
fin
echanger(𝒯, debut, sup)
renvoyer sup
```

Algorithme 6 : Tri rapide
```
Entrées : Tableau 𝒯, entier debut, entier fin
Sorties : 𝒯 trié
si debut < fin alors
 |  pivot ← partitionner(𝒯, debut, fin)
 |  tri_rapide(𝒯, debut, pivot-1)
 |  tri_rapide(𝒯, pivot+1, fin)
fin
```
Terminaison.
Les boucles while internes de partitionner se terminent car $O(1)$ et sup et inf sont variants. La boucle while externe se termine car $sup - inf$ est un variant. Les autres instructions sont en $O(1)$.
Alors partitionner se termine.
Les appels à partitionner se terminent donc tri_rapide aussi.
Correction.
Montrons la correction de partitionner.
Prédicat : «les cases d'indices debut+1 à inf-1 ont des valeurs strictement inférieurs au pivot, les cases d'indices sup+1 à fin ont des valeurs strictement supérieures au pivot».
Avant la boucle, l'ensemble des cases est vide, donc le prédicat est vrai.
Supposons que le prédicat est vrai au début d'une itération.
Les deux boucles internes gardent le prédicat, tout comme le reste des instructions.
C'est bien un invariant, il est vérifié en fin de boucle.
L'échange final permet de mettre le pivot au bon endroit.
Partitionner fonctionne donc correctement, et par récurrence sur fin-debut, tri_rapide aussi.
Complexité.
On prend la comparaison comme opération élémentaire.
On note S le nombre de noeuds de l'arbre des appels.
Alors $h = \Omega(\log S)$ donc $h = \Omega(\log n)$.
et $h = O(S)$ donc $h = O(n)$.
Le meilleur des cas correspond à la plus petite valeur de h : $\beta \log n$ alors on a une complexité en $O(n \log n)$.
Le pire des cas correspond à la plus grande valeur de h : βn, alors on a une complexité en $O(n^2)$.
Complexité en moyenne.
Si les données de départ dans un ordre aléatoire uniforme, alors c'est le cas de tout sous-tableau de l'entrée.
Soit $T(n)$ le nombre d'opérations élémentaires sur une entrée de taille n et $\overline{T}(n)$ le nombre moyen.

$$\begin{aligned}
&T(n) = n + 1 + T(k) + T(n-k-1) \\
&\overline{T}(n) = n + 1 + \frac{1}{n}\sum_{k=0}^{n-1} \overline{T}(k) + \overline{T}(n-k-1) = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} \overline{T}(k) \\
&n\overline{T}(n) = n(n+1) + 2\sum_{k=0}^{n-1} \overline{T}(k) \\
&(n-1)\overline{T}(n-1) = n(n-1) + 2\sum_{k=0}^{n-2} \overline{T}(k) \qquad (n \leftarrow n-1) \\
&n\overline{T}(n) - (n-1)\overline{T}(n-1) = n(n+1) - n(n-1) + 2\overline{T}(n-1) \\
&n\overline{T}(n) = 2n + (n+1)\overline{T}(n-1) \\
&\frac{1}{n+1}\overline{T}(n) = \frac{2}{n+1} + \frac{1}{n}\overline{T}(n-1) \\
&\frac{\overline{T}(n)}{n+1} = \overline{T}(0) + 2\sum_{k=0}^{n+1} \frac{1}{k} = O(\log n) \qquad \text{téléscopage} \\
&\overline{T}(n) = O(n \log n)
\end{aligned}$$

Tri rapide avec doublons.

Algorithme 7 : Tri rapide avec doublons
```
Entrées : Tableau 𝒯, entier debut, entier fin
Sorties : 𝒯 trié
si debut ≥ fin alors
 |  fin
fin
pivot ← 𝒯[debut]
inf ← debut
sup ← fin
egal ← debut
tant que egal ≤ sup faire
 |  si pivot > 𝒯[egal] alors
 |   |  echanger(𝒯, inf, egal)
 |   |  inf ← inf + 1
 |   |  egal ← egal + 1
 |  fin
 |  sinon si pivot < 𝒯[egal] alors
 |   |  echanger(𝒯, sup, egal)
 |   |  sup ← sup - 1
 |  fin
 |  sinon
 |   |  egal ← egal + 1
 |  fin
fin
tri_rapide(𝒯, debut, inf-1)
tri_rapide(𝒯, sup+1, fin)
```

Parcours d'arbres.
Soit A un arbre.
Dans le parcours préfixe, on traite A, puis récursivement son fils gauche, puis son fils droit.
Dans le parcours infixe, on traite son fils gauche, puis A, puis son fils droit.
Dans le parcours postfixe, on traite son fils gauche, puis droit, puis A.

Algorithme 8 : Parcours en largeur
```
Entrées : Arbre A
Sorties : Traitement en largeur de A
pile ← pile vide
empiler(pile, racine de A)
tant que pile non vide faire
 |  noeud ← depiler(pile)
 |  traiter(noeud)
 |  empiler(pile, A.droite)
 |  empiler(pile, A.gauche)
fin
```

☼☼ Fin du sujet ! ☼☼