

1. Introduction

Nous avons déjà rencontré des arbres à la fin du module précédent, pour représenter une expression algébrique ou une liste. Nous avons pu nous contenter d'une approche intuitive, car cette notion vous est déjà certainement familière (qu'on pense par exemple aux arbres généalogiques ou à l'organisation de tournois sportifs, voire à l'organisation des répertoires et fichiers de nos ordinateurs). Mais c'est aussi une structure de donnée extrêmement riche, dont nous allons maintenant explorer quelques aspects.

1.1 Terminologie

C'est en théorie des graphes, qui sera abordée en seconde année, que l'on trouve une définition précise d'un arbre : *un arbre est un graphe simple non orienté, acyclique et connexe*.

- Un graphe *simple* est un graphe dans lequel il n'existe pas d'arête reliant un sommet à lui-même et dans lequel deux sommets distincts sont reliés par au plus une arête ;
- un graphe *acyclique* est un graphe dans lequel il n'existe pas de chemin fermé ;
- un graphe *connexe* est un graphe dans lequel il existe toujours un chemin reliant deux sommets distincts.

Une conséquence de ces deux dernières propriétés est que pour tout couple de sommets (r, s) il existe un unique chemin qui conduit de r à s , et que le choix de r induit une orientation implicite du graphe. Dans ce cas, on dit que l'arbre est *enraciné* et que r est la *racine* de l'arbre.

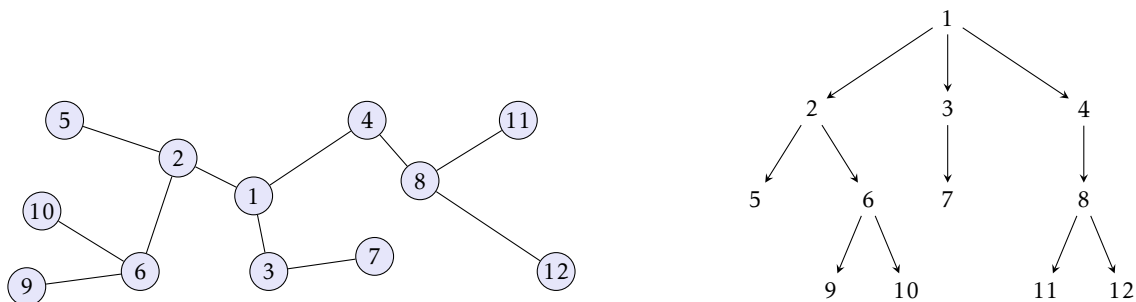


FIGURE 1 – Un arbre et son enracinement à partir du sommet 1.

Dans l'exemple ci-dessus, le sommet 1 est la *racine* de l'arbre.

Si une arête mène du sommet i au sommet j , on dit que i est le *père* de j , et en conséquence que j est le *fil* de i . Par exemple, le sommet 3 est le père de 7, le sommet 6 est le fils de 2.

Il est d'usage de dessiner un arbre en plaçant un père au dessus de ses fils, si bien que l'on peut sans ambiguïté représenter les arêtes par des traits simples à la place des flèches, ce que l'on fera par la suite.

Quand un graphe est un arbre on parle plus facilement de *nœud* que de sommet. On distingue les *nœuds externes* (les *feuilles*) qui n'ont pas de fils des *nœuds internes* qui en ont. Dans l'exemple ci-dessus, les feuilles sont les sommets 5, 7, 9, 10, 11, et 12 et les nœuds internes les sommets 1, 2, 3, 4, 6 et 8.

Enfin, on notera que chaque nœud est la racine d'un arbre constitué de lui-même et de l'ensemble de ses descendants ; on parle alors de *sous-arbre* de l'arbre initial. Par exemple, les sommets 2, 5, 6, 9 et 10 constituent un sous-arbre enraciné en 2 de l'arbre représenté ci-dessus. Il arrivera fréquemment qu'on identifie le nœud et le sous-arbre dont il est la racine.

1.2 Définition d'une structure de données

En réalité, il n'existe pas *une* structure de données associée aux arbres, mais *des* structures de données, que l'on définit en fonction de l'organisation que l'on souhaite. En outre, l'information peut être stockée dans les nœuds, dans les feuilles, voire dans les arêtes. Par la suite, on appellera *étiquette* l'information attachée à un nœud, une feuille (ou une arête).

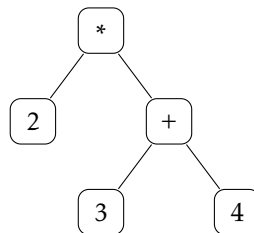
On appelle *arité* d'un nœud le nombre de ses fils. Ainsi, les feuilles sont les nœuds d'arité 0. Un *arbre binaire strict* est un arbre dans lequel chaque nœud interne a pour arité 2, ce qui nous amène à la définition du type de données abstrait suivant :

$$\text{Arbre} = \text{feuille} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

Si on choisit en outre d'attacher une étiquette aussi bien aux nœuds internes qu'aux feuilles, ceci conduit à définir le type suivant :

```
type ('a, 'b) arbre =
  | Feuille of 'a
  | Noeud of 'b * ('a, 'b) arbre * ('a, 'b) arbre ;;
```

On notera que l'information stockée dans les feuilles (de type '*a*') et de nature différente de celle attachée aux nœuds internes (de type '*b*'). Ceci permet par exemple de définir une structure de donnée adaptée à la représentation des expressions n'utilisant que des opérateurs binaires. Par exemple, l'expression $2 * (3 + 4)$ est représentée par l'arbre dessiné ci-dessous :



et définie en CAML de la façon suivante :

```
# let arb = Noeud ("*", Feuille 2, Noeud ("+", Feuille 3, Feuille 4)) ;;
arb : (int, string) arbre = ...
```

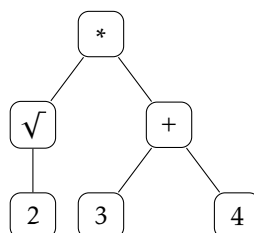
Plus généralement, un *arbre binaire* est un arbre dont chaque nœud a une arité égale à 0, 1 ou 2. Pour définir la structure de donnée associée, il est pratique de convenir que le graphe vide est un arbre et ainsi à définir le type :

$$\text{Arbre} = \text{nil} + \text{Arbre} \times \text{nœud} \times \text{Arbre}$$

```
type 'a arbre =
  | Nil
  | Noeud of 'a * 'a arbre * 'a arbre ;;
```

On notera que cette structure de données ne permet plus de distinguer le type d'information portée par les nœuds internes et par les feuilles : les feuilles sont les nœuds dont les deux fils sont l'arbre vide, les nœuds d'arité 1 ceux dont l'un des deux fils uniquement est l'arbre vide, les nœuds d'arité 2 ceux dont aucun des deux fils n'est l'arbre vide.

Par exemple, l'expression $\text{sqrt}(2) * (3 + 4)$ est représentée par l'arbre dessiné ci-dessous :



et définie en CAML par :

```
# let arb = Noeud ("*", Noeud ("sqrt", Noeud ("2", Nil, Nil), Nil),
                  Noeud ("+", Noeud ("3", Nil, Nil),
                        Noeud ("4", Nil, Nil))) ;;

arb : string arbre = ...
```

Enfin, pour permettre la représentation d'arbres dont les arités peuvent prendre des valeurs quelconques, une solution (ce n'est pas la seule) consiste à représenter l'ensemble des descendants d'un nœud par une liste. On définit ainsi le type :

```
type 'a arbre = Noeud of 'a * ('a arbre list) ;;
```

Par exemple, l'arbre représenté figure 1 peut être défini en écrivant :

```
# let arb = Noeud (1, [ Noeud (2, [ Noeud (5, []);
                                   Noeud (6, [ Noeud (9, []);
                                                Noeud (10, []) ]) ]);
                  Noeud (3, [ Noeud (7, []) ]);
                  Noeud (4, [ Noeud (8, [ Noeud (11, []);
                                           Noeud (12, []) ]) ]) ]);

arb : int arbre = ...
```

On observera qu'avec cette définition il n'est pas possible de représenter l'arbre vide. Pour le représenter il faudrait définir le type :

```
type 'a arbre = Nil | Noeud of 'a * ('a arbre list) ;;
```

On retiendra de cette section qu'il n'existe pas une manière canonique de représenter un arbre en machine, le choix de la représentation étant fortement tributaire du type d'arbre destiné à supporter les données.

2. Algorithmes sur les arbres binaires

Dans la suite de ce chapitre et dans un souci de simplification nous allons maintenant nous restreindre aux arbres binaires et utiliser uniquement le type :

```
type 'a btree = Nil | Node of 'a * 'a btree * 'a btree
```

Il va de soit que les notions que nous allons aborder sont pour la plus-part d'entre-elles généralisables au cas d'un arbre quelconque ; on se reportera à l'exercice 4 pour rédiger certaines de ces fonctions dans le cas d'un arbre n -aire.

2.1 Nombre de feuilles et de nœuds

La définition récursive des arbres incite tout naturellement à procéder par filtrage pour agir sur ceux-ci. Par exemple, pour calculer le nombre de feuilles ou de nœuds d'un arbre binaire, il suffit d'écrire :

```
let rec nb_feuilles = function
| Nil                -> 0
| Node (_, Nil, Nil) -> 1
| Node (_, fils_g, fils_d) -> (nb_feuilles fils_g) + (nb_feuilles fils_d) ;;
```

```
let rec nb_noeuds = function
| Nil                -> 0
| Node (_, fils_g, fils_d) -> 1 + (nb_noeuds fils_g) + (nb_noeuds fils_d) ;;
```

En général il n'existe pas de relation simple entre nombre de feuilles et nombre de nœuds, sauf dans le cas d'un arbre binaire strict. Dans ce cas, on dispose du résultat suivant :

THÉORÈME. — *Si un arbre binaire strict possède n nœuds internes et f feuilles alors $f = n + 1$ (et donc $2f - 1$ nœuds).*

Preuve. Deux preuves sont possibles. La première se contente de raisonner par récurrence sur le nombre de nœuds total de l'arbre :

- si cet arbre est constitué d'une unique feuille, alors $n = 0$ et $f = 1$ et le résultat est bien vérifié ;
- dans le cas contraire, cet arbre est constitué d'une racine ayant deux fils. Notons n_1 et f_1 (respectivement n_2 et f_2) le nombre de nœuds internes et de feuilles du premier fils (respectivement du second), et supposons $f_1 = n_1 + 1$ et $f_2 = n_2 + 1$. Alors $n = 1 + n_1 + n_2$ et $f = f_1 + f_2$ donc $f = (n_1 + 1) + (n_2 + 1) = n_1 + n_2 + 2 = n + 1$.

Mais il est aussi possible de raisonner directement en observant que le nombre de sommets ayant un père est égal à $n + f - 1$ (seule la racine n'a pas de père) et que chaque père a exactement deux fils (il s'agit d'un arbre binaire strict). Or seuls les nœuds internes sont des pères, donc $2n = n + f - 1$, soit $n = f - 1$. \square

2.2 Hauteur d'un arbre

La *profondeur* d'un nœud est la distance qui sépare ce nœud de la racine, et la *hauteur* de l'arbre est la profondeur maximale d'un nœud. Bien entendu, cette profondeur maximale ne peut être atteinte que pour une feuille. Par exemple, la hauteur de l'arbre dessiné figure 1 est égale à 3 ; celle des deux arbres binaires de la page précédente égale à 2.

On conviendra que la hauteur de l'arbre vide est égale à -1 .

On calcule la hauteur d'un arbre binaire à l'aide de la fonction :

```
let rec hauteur = fonction
| Nil -> -1
| Node (_, fils_g, fils_d) -> 1 + max (hauteur fils_g) (hauteur fils_d) ;;
```

Attention à ne pas confondre la profondeur d'un nœud (la distance qui le sépare de la racine) de sa hauteur, sachant que tout nœud est la racine de l'arbre de sa descendance.

THÉORÈME. — Si h désigne la hauteur d'un arbre binaire, le nombre de feuilles est majoré par 2^h .

Preuve. Raisonnons par récurrence sur la hauteur de l'arbre.

Un arbre de hauteur nulle est réduit à une feuille. Un arbre de hauteur $h \geq 1$ possède une racine avec deux fils. Chacun de ces deux fils est un arbre de hauteur inférieure ou égale à $h - 1$, donc par hypothèse de récurrence chacun possède au plus 2^{h-1} feuilles. L'arbre initial possède donc au plus $2 \times 2^{h-1} = 2^h$ feuilles. \square

COROLLAIRE. — Un arbre binaire possédant n feuilles a pour hauteur minimale $\lceil \log_2 n \rceil$.

Preuve. En effet, $n \leq 2^h \iff \log_2 n \leq h \iff h \geq \lceil \log_2 n \rceil$ puisque h est un entier. \square

Remarque. Un arbre binaire de hauteur h dont le nombre de feuilles est égal à 2^h est appelé un arbre binaire *complet*. Il est facile de prouver (en raisonnant par récurrence sur h) qu'un arbre binaire complet est nécessairement strict et que toutes les feuilles sont à la même profondeur.

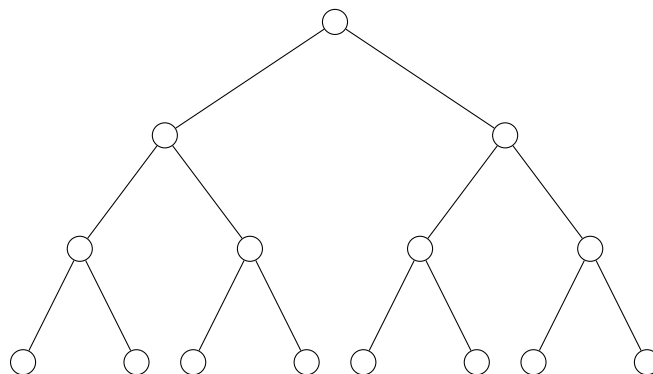


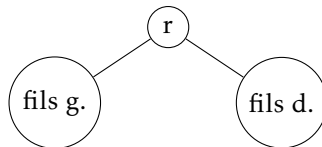
FIGURE 2 – Un exemple d'arbre binaire complet.

2.3 Parcours d'arbres binaires

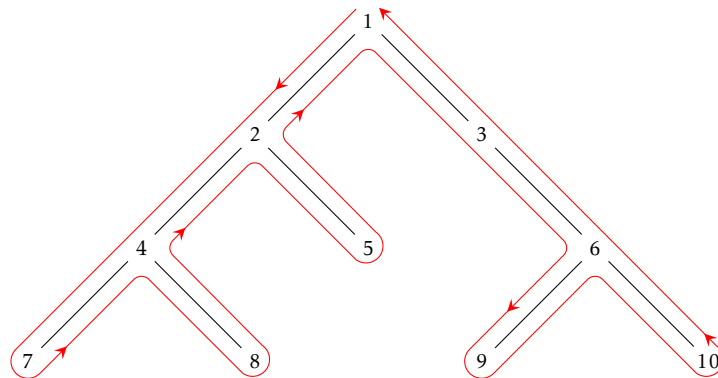
Parcourir un arbre, c'est ordonner la liste des nœuds et feuilles de celui-ci en vue d'effectuer un certain traitement (par exemple imprimer la liste des étiquettes de cet arbre). On distingue deux types de parcours : les parcours en profondeur et les parcours hiérarchiques, que nous allons maintenant décrire.

• Parcours en profondeur

Dans le parcours en profondeur d'un arbre constitué d'une racine et de deux sous-arbres, chaque sous-arbre est exploré dans son entier avant d'explorer le sous-arbre suivant.



Le parcours préfixe consiste à parcourir l'arbre suivant l'ordre : $r \rightarrow \text{fils g.} \rightarrow \text{fils d.}$. Dans l'exemple qui suit, il correspond au parcours de l'arbre suivant l'ordre 1 - 2 - 4 - 7 - 8 - 5 - 3 - 6 - 9 - 10 :



c'est à dire suivant le premier passage à *gauche* d'un nœud lorsqu'on parcourt le trajet représenté ci dessus.

Par exemple, pour afficher les étiquettes d'un arbre de type *int arbre* en suivant un parcours en profondeur préfixe, il faudrait définir :

```

let rec parcours_prefixe = fonction
| Nil          -> ()
| Node (r, fils_g, fils_d) -> print_int r ;
                                parcours_prefixe fils_g ;
                                parcours_prefixe fils_d ;;
  
```

Le parcours suffixe consiste à parcourir l'arbre suivant l'ordre : $\text{fils g.} \rightarrow \text{fils d.} \rightarrow r$. Dans l'exemple précédent, il correspond au parcours 7 - 8 - 4 - 5 - 2 - 9 - 10 - 6 - 3 - 1, c'est à dire suivant le premier passage à *droite* d'un nœud.

Pour afficher les étiquettes d'un arbre de type *int arbre* en suivant un parcours en profondeur suffixe, il suffit de modifier très légèrement la fonction précédente :

```

let rec parcours_suffixe = fonction
| Nil          -> ()
| Node (r, fils_g, fils_d) -> parcours_suffixe fils_g ;
                                parcours_suffixe fils_d ;
                                print_int r ;;
  
```

Le parcours infixé (ou parcours symétrique) consiste à parcourir l'arbre suivant l'ordre : $\text{fils g.} \rightarrow r \rightarrow \text{fils d.}$. Dans l'exemple précédent, il correspond au parcours 7 - 4 - 8 - 2 - 5 - 1 - 3 - 9 - 6 - 10, c'est à dire suivant le premier passage *sous* un nœud.

Pour afficher les étiquettes d'un arbre de type *int arbre* en suivant un parcours en profondeur infixé, on définit la fonction :

```

let rec parcours_infixe = fonction
| Nil          -> ()
| Node (r, fils_g, fils_d) -> parcours_infixe fils_g ;
                               print_int r ;
                               parcours_infixe fils_d ;;

```

Remarque. On observera que dans le cas d'un arbre représentant une expression arithmétique, les parcours préfixe et postfixe de l'arbre correspondent respectivement à la représentation préfixe et postfixe de l'expression.

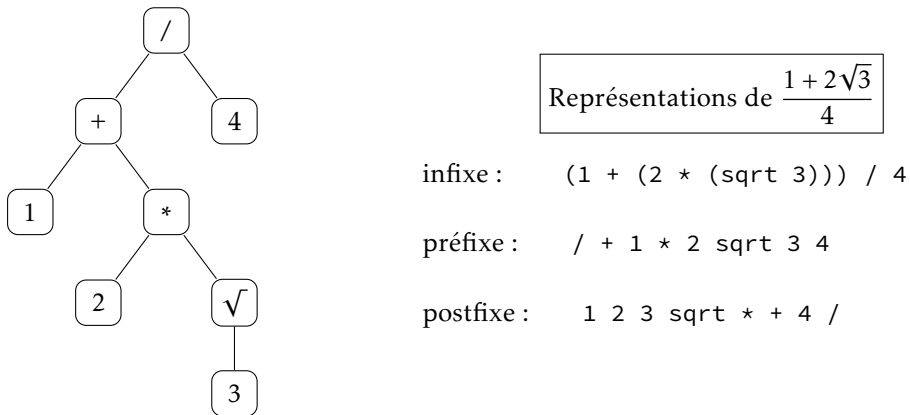
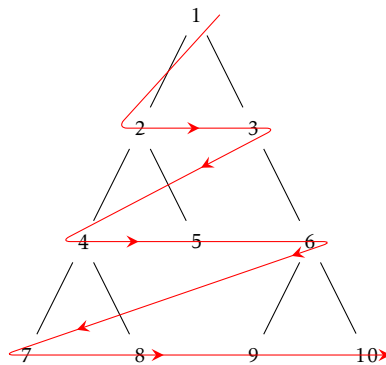


FIGURE 3 – Représentations d'une expression arithmétique. Les représentations préfixe et postfixe sont non ambiguës, l'usage des parenthèses est superflu.

• Parcours hiérarchique

Le parcours hiérarchique, ou parcours en largeur, consiste à parcourir les nœuds et feuilles de l'arbre en les classant par profondeur croissante (et en général de la gauche vers la droite pour une profondeur donnée). Dans l'exemple qui suit, il correspond à l'ordre 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 :



Nous ne décrivons pas dans l'immédiat la fonction de parcours hiérarchique d'un arbre car celle-ci utilise une structure de donnée qui sera étudiée plus tard, les *files d'attente*.

3. Exercices

Exercice 1 On définit le type *arbre* suivant :

```

type arbre = Nil | Noeud of arbre * arbre ;;

```

Rédiger une fonction **genere_complet** de type $int \rightarrow arbre$ qui retourne un arbre complet de hauteur n .

On appelle *cheminement* d'un arbre la somme des profondeurs de chacune de ses feuilles. Rédiger une fonction **cheminement** de type $arbre \rightarrow int$ qui calcule le cheminement d'un arbre quelconque. À quoi est égal le cheminement d'un arbre complet de hauteur n ?

Exercice 2 La numérotation de Sosa-Stradonitz d'un arbre binaire strict, utilisée notamment en généalogie, consiste à attribuer un numéro à chaque nœud (interne ou feuille) d'un arbre binaire strict, en suivant les règles suivantes :

- le numéro de la racine est égal à 1 ;
- si un nœud est de numéro n , son fils gauche se verra attribuer le numéro $2n$ et son fils droit le numéro $2n + 1$.

On définit le type suivant :

```
type 'a arbre = Feuille of 'a | Noeud of 'a * 'a arbre * 'a arbre ;;
```

Écrire une fonction de type `'a arbre -> (int * 'a) arbre` qui ajoute à chaque nœud son numéro.

Exercice 3 On représente les arbres binaires à étiquettes entières par le type :

```
type arbre = Nil | Noeud of int * arbre * arbre ;;
```

Écrire une fonction de type `arbre -> int -> int list` qui, à partir d'un arbre binaire et d'un entier n , calcule la liste de toutes les étiquettes des nœuds de la profondeur n de l'arbre.

Exercice 4 On représente les arbres n -aires par le type :

```
type 'a arbre = Noeud of 'a * ('a arbre list) ;;
```

(À chaque nœud sont associées son étiquette et la liste de ses fils).

Rédiger les fonctions `nb_feuilles` et `hauteur` de type `'a arbre -> int` qui calculent respectivement le nombre de feuilles et la hauteur d'un arbre n -aire.

Exercice 5 Le nombre de STRAHLER d'un arbre binaire strict est une mesure de la complexité de cet arbre ; il est défini de la manière suivante :

- le nombre de STRAHLER d'une feuille est égal à 1 ;
- si i et j désignent les nombres de STRAHLER des fils gauche et droit d'un nœud, alors le nombre de Stralher de ce dernier sera égal à $\max(i, j)$ si $i \neq j$, et à $i + 1$ si $i = j$.

On définit le type suivant :

```
type arbre = Feuille | Noeud of arbre * arbre ;;
```

- a) Définir une fonction de type `arbre -> int` qui détermine le nombre de STRAHLER d'un arbre binaire strict.
- b) Quels sont, pour une hauteur donnée, les arbres de complexité maximale ? de complexité minimale ?
- c) Les arbres de complexité minimale sont appelés des arbres *filiformes*. Combien y-en-a-t'il pour une hauteur donnée ?

Écrire une fonction de type `int -> arbre list` qui détermine la liste de tous les arbres filiformes de hauteur donnée.



Exercice 6 On définit le type d'arbre binaire strict suivant :

```
type arbre = Feuille | Noeud of arbre * arbre ;;
```

et on définit par récurrence la suite des arbres de FIBONACCI en convenant que :

- A_0 et A_1 sont des feuilles ;
- si $n \geq 2$, A_n est l'arbre binaire dont le sous-arbre gauche est A_{n-1} et le sous-arbre droit, A_{n-2} .

- a) Dessiner les arbres A_2 , A_3 , A_4 et A_5 , puis définir une fonction CAML qui génère les arbres de FIBONACCI.
- b) Quelle est la hauteur de A_n ? Montrer que les arbres de FIBONACCI sont H-équilibrés, c'est à dire qu'en tout nœud de l'arbre les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.
- c) Déterminer le nombre de feuilles et de nœuds de A_n à l'aide des nombres de la suite de FIBONACCI.
- d) Quel est le nombre de STRAHLER de l'arbre A_n ?

Exercice 7 On note c_n le nombre d'arbres binaires à n nœuds, en faisant abstraction de leur étiquette. Ainsi, $c_0 = 1$ (l'arbre vide), $c_1 = 1$ (l'arbre constitué d'un unique nœud) $c_2 = 2$ ( et ) et $c_3 = 5$:

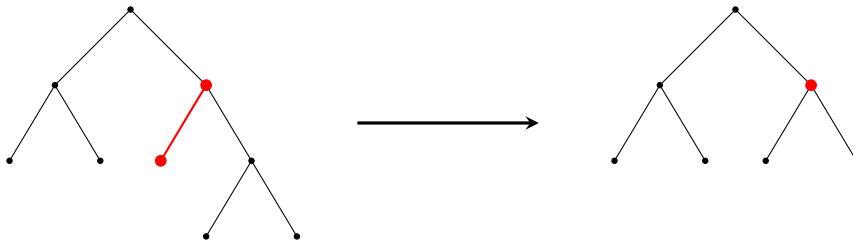


a) Justifier la relation de récurrence suivante : $\forall n \in \mathbb{N}, c_{n+1} = \sum_{k=0}^n c_k c_{n-k}$.

La suite $(c_n)_{n \in \mathbb{N}}$ définie par la valeur $c_0 = 1$ et la relation de récurrence énoncée ci-dessus est bien connue : il s'agit de la suite de CATALAN ; elle intervient dans de nombreux problèmes de dénombrement (nous la rencontrerons à plusieurs reprises dans ce cours). La résolution usuelle de cette relation de récurrence utilise la notion de série génératrice, elle figure dans de nombreux ouvrages. Nous allons en voir une autre, basée sur une seconde relation.

b) Montrer que c_n est aussi le nombre d'arbres binaires stricts ayant $n + 1$ feuilles.

c) Considérons un arbre binaire strict à $n + 1$ feuilles, ainsi qu'une de ses feuilles, et effectuons la transformation suivante :



on élimine la feuille en question ainsi que son père, et le deuxième fils prend la place de ce dernier. On obtient ainsi un arbre binaire strict ayant n feuilles.

Déduire de cette transformation la relation suivante : $(n + 1)c_n = 2(2n - 1)c_{n-1}$, et prouver alors que

$$c_n = \frac{1}{n+1} \binom{2n}{n}.$$

Exercice 8 Soient x et y deux nœuds d'un même arbre. Montrer que x est un ascendant de y si et seulement si x précède y suivant l'ordre préfixe et succède à y suivant l'ordre suffixe.