

Chapitre 9

ARBRES

| Notions | Commentaires |
|--|--|
| Définition inductive du type arbre binaire. Vocabulaire : noeud, noeud interne, racine, feuille, fils, père, hauteur d'un arbre, profondeur d'un noeud, étiquette, sous-arbre. | La hauteur de l'arbre vide est -1 . On mentionne la représentation d'un arbre complet dans un tableau. |
| Arbre. Conversion d'un arbre d'arité quelconque en un arbre binaire. | La présentation donne lieu à des illustrations au choix du professeur. Il peut s'agir par exemple d'expressions arithmétiques, d'arbres préfixes (<i>trie</i>), d'arbres de décision, de dendrogrammes, d'arbres de classification, etc. |
| Parcours d'arbre. Ordre préfixe, infixe et postfixe. | On peut évoquer le lien avec l'empilement de blocs d'activation lors de l'appel à une fonction réursive. |
| [...] Arbre binaire de recherche. Arbre bicolore. | On note l'importance de munir l'ensemble des clés d'un ordre total. |

Extrait de la section 3.3 du programme officiel de MP2I : « Structures de données hiérarchiques ».

| Notions | Commentaires |
|---|--|
| Implémentation interne [des fichiers] : blocs et noeuds d'index (<i>inode</i>). | On présente le partage de blocs (avec liens physiques ou symboliques) et l'organisation hiérarchique de l'espace de nommage. |

Extrait de la section 5.2 du programme officiel de MP2I : « Gestion des fichiers et entrées-sorties ».

SOMMAIRE

| | |
|---|------------|
| 0. Exemples introductifs et vocabulaire..... | 165 |
| 0. Les mobiles | 165 |
| 1. Les expressions arithmétiques | 166 |
| 2. Arbre d'appels récursifs | 166 |
| <i>Tours de Hanoï (p. 166). Retour sur trace (p. 167).</i> | |
| 3. Exemple : les tries | 168 |
| 4. Exemple : les arbres binaires de recherche | 168 |
| 5. Exemple et parenthèse : gestion des fichiers Unix | 169 |
| <i>Inodes et fichiers (p. 169). Inodes et dossiers (p. 169). Liens physiques et symboliques (p. 170).</i> | |
| 6. Vrac de vocabulaire | 171 |
| 7. Construction inductive | 172 |
| 1. Arbres binaires et implémentation..... | 173 |
| 0. Arbres binaires | 173 |
| <i>Définition (p. 173). Implémentation (p. 176).</i> | |
| 1. Arbres binaires stricts | 176 |
| <i>Définitions (p. 176). Implémentation (p. 177).</i> | |
| 2. Arbres parfaits et complets | 178 |
| <i>Définitions (p. 178). Implémentation des arbres binaires complets (p. 180).</i> | |

| | |
|--|------------|
| 3. Peigne | 180 |
| 4. Arbres quelconques | 181 |
| <i>Transformation LCRS (p. 181). Type général (p. 182).</i> | |
| 2. Parcours d'arbres..... | 183 |
| 0. Parcours en profondeur | 183 |
| <i>Ordre préfixe (p. 184). Parcours infixe (p. 185). Parcours postfixe (p. 186). Écriture itérative (p. 186).</i> | |
| 1. Parcours en largeur | 187 |
| <i>Écriture itérative (p. 187). (Mi-HP) Preuve du pseudo-code (p. 188). (Mi-HP) Preuve, suite et fin (p. 189).</i> | |
| 3. (HP) Propriétés avancées | 189 |
| 0. Lemme de lecture unique | 189 |
| 1. Dénombrement des arbres binaires stricts | 190 |
| 4. Arbres binaires de recherche | 194 |
| 0. Définition, caractérisation | 194 |
| 1. Opérations | 196 |
| <i>Recherche (p. 196). Insertion (p. 197). Suppression (p. 198). Rotation (p. 200). Tri par ABR (p. 201).</i> | |
| 2. Arbres Rouge-Noir | 202 |
| <i>Définition (p. 202). Insertion (p. 204). Suppression (p. 206).</i> | |
| 3. Pour aller plus loin | 206 |

0 Exemples introductifs et vocabulaire

Définition 1 (Arbres).

Un **arbre** est une structure de données hiérarchique constituée de $n \in \mathbb{N}$ **noeuds**, éventuellement étiquetés. S'il n'est pas vide ($n \geq 1$), les noeuds sont structurés de la manière suivante :

- Un noeud particulier r est appelé la **racine** de l'arbre.
- Les $n - 1$ noeuds restants sont partitionnés en $k \geq 0$ sous-ensemble disjoints qui forment k arbres, appelés **sous-arbres** de r .
- La racine r est reliée à la racine de chacun des sous-arbres. Ces liens sont appelés des **arêtes**.
- S'il n'y a pas de sous-arbres (si $k = 0$), on dit que r est une **feuille**.

FIGURE 9.1 – Un arbre

Remarque.

- Autrement dit, un arbre est un type inductif! Un arbre est soit constitué d'une racine reliée à des sous-arbres (éventuellement vides), soit vide.
- Il n'y a pas une seule définition des arbres, mais plein de variations autour de cette définition.

0.0 Les mobiles

Voici un type pour les mobiles de Calder :

```
4 type objet = int
5 type mobile = Obj of objet | Bar of mobile * mobile
```



Ici, les noeuds sont les barres ou les objets. Vous noterez qu'il n'autorise pas les mobiles vides : le cas de base est l'objet, qui est donc une feuille.

FIGURE 9.2 – Un mobile de Calder décomposé comme un arbre

0.1 Les expressions arithmétiques

Voici un type pour des expressions arithmétiques (sans division) :

```

4 type expr =
5   | Int of int
6   | Var of string
7   | Add of expr * expr
8   | Sub of expr * expr
9   | Mul of expr * expr
10  | Div of expr * expr
11  | Mod of expr * expr
12

```

 arith.ml

Ici, les noeuds sont des entiers, variables ou des opérations. Les feuilles correspondent aux entiers et aux variables. Notez que l'on pourrait fusionner tous les cas inductifs en un seul :

```

19 type expr =
20   | Int of int
21   | Var of string
22   | Fun of (int -> int -> int) * expr * expr
23     (* ex : Fun ((+), Int 3, Int 2)
24        correspond à Add (Int 3, Int 2) *)

```

 arith.ml

FIGURE 9.3 – Une expression arithmétique décomposée comme un arbre

0.2 Arbre d'appels récursifs

0.2.0 Tours de Hanoï

Voici un pseudo-code résolvant les tours de Hanoï :

Fonction Hanoï

Entrées : i : la tige de départ ; j : la tige d'arrivée ; n : le nombre de disques à déplacer

```

1 si n > 0 alors
2   k ← tige autre que i ou j
3   HANOÏ(i, k, n-1)
4   Déplacer 1 disque de i à j
5   HANOÏ(k, j, n-1)

```

Et voici la forme de l'arbre d'appels associé pour $n = 3$:

FIGURE 9.4 – Arbre d'appels pour $n = 3$

On remarque que le déroulé de l'algorithme récursif consiste à se déplacer dans l'arbre. On appelle cela un **parcours** de l'arbre.

0.2.1 Retour sur trace

Voici l'arbre des plateaux des n dames explorés par un retour sur trace qui place les dames ligne par ligne et rejette dès que deux dames sont en prise :

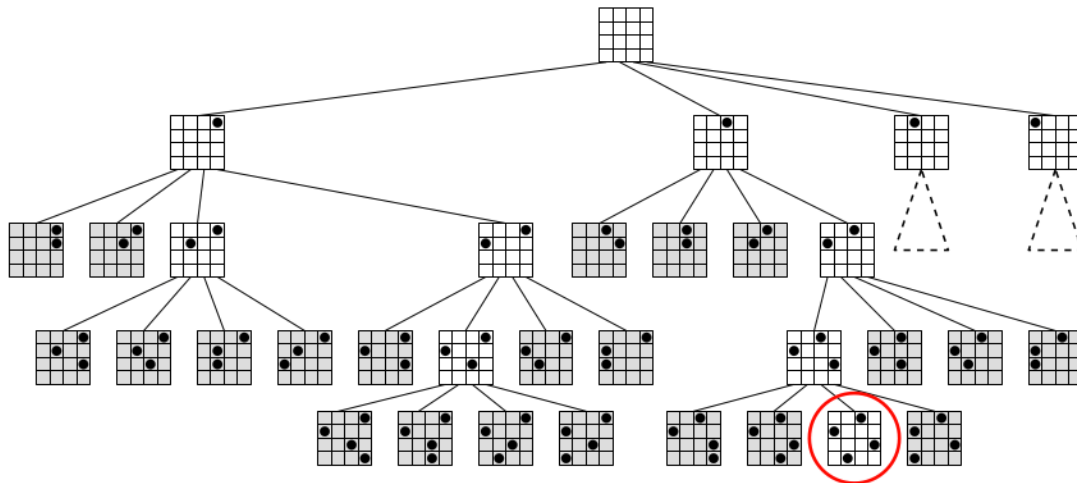


FIGURE 9.5 – Exploration en plaçant d'abord sur la première ligne, puis sur la seconde, etc. En gris les solutions partielles rejetées. La moitié non-représentée de la figure est symétrique. Src : J.B. Bianquis

Remarque.

- Si l'on code l'algorithme de sorte à ce qu'il s'arrête dès qu'il a trouvé une solution, on arrête le parcours de l'arbre dès que l'on rencontre le plateau valide entouré en rouge.
- Le rejet dans un retour sur trace est aussi appelé un **élagage** : on « coupe » des « branches » de l'arbre.

0.3 Exemple : les tries

Un **trie** est un arbre qui permet de représenter un ensemble de mots, en fusionnant leurs préfixes : on étiquette les arêtes de l'arbre par des lettres, et un mot stocké dans le trie correspondant à un chemin (descendant) de la racine jusqu'à une feuille.

FIGURE 9.6 – Trie contenant { dragon, drame, drap, droide, droite, drone, demon, de, degat, demo, degel, demi }

0.4 Exemple : les arbres binaires de recherche

Un arbre binaire de recherche est un arbre où chaque noeud a au plus deux sous-arbres. On range des valeurs dans les noeuds. L'étiquette d'un noeud est plus grande que celles de son sous-arbre gauche et plus petite que celles de son sous-arbre droit : cela permet d'y effectuer des recherches plutôt efficaces, « par dichotomie ».

(a) Un arbre binaire de recherche pour E .

(b) Un autre arbre binaire de recherche pour E .

FIGURE 9.7 – Deux arbres binaires de recherche pour $E = \llbracket 1; 12 \rrbracket$

0.5 Exemple et parenthèse : gestion des fichiers Unix

0.5.0 Inodes et fichiers

Un disque dur sert à contenir des fichiers, et est naturellement linéaire : les différentes zones d'un disque dur sont correspondent à différentes « adresses », les unes après les autres. Plus précisément, un disque dur est une suite de **blocs**, et un fichier est réparti sur plusieurs blocs.

Chaque fichier correspond en fait à :

- Des métadonnées (e.g. date de dernière modification)
- L'adresse des blocs du disque dur contenant le contenu du fichier.

Ces informations dans une structure qui s'appelle un **inode** (information **node**).

FIGURE 9.8 – Schéma (très simplifié) d'un inode indiquant des blocs d'un disque dur.

Remarque.

- Le nom d'un fichier n'est pas stocké dans l'inode du fichier, mais dans celui du dossier parent. Cf suite du cours.
- Une conséquence intéressante de cette façon de gestion des fichiers est que faire un couper-coller est quasi-immédiat : on ne recopie absolument pas tout le fichier, on ne déplace que l'inode (qui est très petit). Ainsi, couper-coller un fichier de 20 Go et un fichier de 10ko est à peu près aussi long.
- La remarque précédente ne s'applique pas à la copie : on veut que la copie d'un fichier soit physiquement différente de l'original (c'est à dire qu'il ne s'agisse pas des mêmes blocs du disque dur), de sorte à ce que modifier la copie ne modifie pas l'original (et réciproquement).
- En réalité, rien ne garantit que les blocs d'un fichier soient consécutifs : un inode pointe sur tous les blocs. Si un seul inode n'est pas assez long pour contenir tous les inodes, il y a une gestion par liste chaînée d'inodes¹ : un inode indique où trouver les premiers blocs du fichier, et où trouver l'inode de la suite.

0.5.1 Inodes et dossiers

Les dossiers sont des fichiers particuliers : les dossiers sont des inodes dont les blocs pointés sont... les inodes du contenu du dossier².

FIGURE 9.9 – Schéma (très simplifié et un peu faux) de l'inode d'un dossier.

1. Je simplifie. Plus généralement, toute cette sous-partie sur les inodes tient plus de la vulgarisation ; et a avant tout pour but de vous faire comprendre que la représentation et l'organisation des fichiers sur un disque dur est quelque chose qui s'étudie et s'optimise.

2. En réalité, c'est faux. L'inode d'un dossier indique un fichier spécial qui lui-même redirige vers les inodes. Je simplifie à outrance.

Dans un inode de dossier, on trouve des informations supplémentaires sur le contenu du dossier, et notamment... les noms de fichiers/sous-dossiers ! En effet, le nom d'un fichier ne fait pas partie de ses méta-données mais des données de son dossier parent.

Remarque.

- Un dossier contient des fichiers ou des dossiers qui contiennent eux-même... etc. On parle de l'arborescence des fichiers : l'organisation des dossiers/fichiers est un arbre. D'ailleurs, le tout premier dossier est appelé la *racine* du système. Sur Unix, il s'agit du dossier `/`.
- Puisque cet arbre correspond aux successions d'inodes qui indiquent des inodes des sous-dossiers, etc, il s'agit d'un exemple d'arbre implémenté comme une succession de « pointeurs »³. C'est une implémentation courante pour des arbres.

0.5.2 Liens physiques et symboliques

Un lien est un raccourci. Il y a deux types de liens :

- Un lien physique (« hard link ») : un lien physique `l` vers un fichier `f` correspond à un inode qui indique le fichier `f`. Autrement dit, c'est vraiment le même fichier ! En particulier, il a le même poids que `f` !
En bref, un lien physique permet de faire en sorte que le même fichier soit accessible depuis plusieurs endroits.
⚠ On ne peut pas faire un lien physique vers un dossier, uniquement vers un fichier. Cela garantit que l'arborescence reste un arbre (on pourrait avoir un cycle sinon).
- Un lien symbolique (« soft link ») : c'est un lien vers (donc un inode qui redirige vers⁴) un élément de l'arborescence. Il est donc très léger, et a l'avantage de pouvoir pointer vers un dossier.

Remarque.

- Un lien physique correspond au même inode que l'origine. Pourtant, un lien physique peut avoir un nom différent de l'original : c'est pour cela que le nom n'est pas stocké dans l'inode du fichier !
- Corollaire intéressant des liens physiques : l'espace utilisé sur votre disque dur n'est pas la somme des poids des éléments présents dans votre arborescence, puisque certains de ces éléments peuvent être un lien physique vers un autre élément de l'arborescence.
- Autre corollaire intéressant : déplacer le fichier `f` à un autre endroit de l'arborescence ne modifiera pas son hard link `l` : en effet, `l` pointe vers les blocs du disque dur, et non vers un endroit de l'arborescence. Plus fort encore, supprimer `f` ne modifiera pas `l` : en effet, supprimer `f` revient simplement à supprimer son inode. Comme les blocs du disque dur sont encore pointé par `l`, eux ne sont pas effacés.

On peut utiliser `ls -l` pour lister le contenu d'un dossier et afficher la nature de ses éléments (ainsi que leur date de dernière modification). En ajoutant `-sh`, on peut obtenir les poids des éléments :

```
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Bureau
lrwxrwxrwx 1 antoine antoine 28 oct. 10 11:35 CPGE -> /home/antoine/Documents/CPGE
drwxr-xr-x 17 antoine antoine 4,0K déc. 16 16:58 Documents
drwxrwxr-x 6 antoine antoine 4,0K déc. 29 22:35 foo
-rw-rw-r-- 1 antoine antoine 99K déc. 28 15:15 guidelines.pdf
drwxr-xr-x 12 antoine antoine 4,0K nov. 12 14:57 Images
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Modèles
lrwxrwxrwx 1 antoine antoine 33 oct. 10 11:35 MP2I-tex -> /home/antoine/Documents/CPGE/MP2I
lrwxrwxrwx 1 antoine antoine 41 oct. 10 11:35 MPI-tex -> /home/antoine/Documents/CPGE/MPI-Poitiers
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Musique
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Public
drwx----- 6 antoine antoine 4,0K févr. 5 09:00 snap
drwxr-xr-x 12 antoine antoine 20K févr. 4 11:28 Téléchargements
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Vidéos
```

FIGURE 9.10 – Contenu de mon dossier home obtenu via `ls -sh`.

3. Je pense que tout-e spécialiste du sujet s'étranglerait en m'entendant en parler comme des pointeurs, mais c'est l'idée. À outrance j'ai dit !

4. Vers un fichier spécial qui redirige vers

Sur la capture d'écran de la figure 9.10, notez que :

- MP2I-tex et MPI-tex sont des liens symboliques pointant vers un autre endroit de mon arborescence. Ils sont *extrêmement* légers.
- Les dossiers en eux-même sont très légers : en effet, un dossier est un nom ainsi qu'une redirection vers son contenu. En particulier, le poids d'un dossier ne comprend pas le poids de son contenu ! Notez que la raison pour laquelle ils sont malgré tout aussi lourd (4K) est parce qu'il y a une taille minimale. Je soupçonne que la raison pour laquelle Téléchargements est aussi lourd est parce qu'il ne tient pas dans 4K (il doit pointer vers beaucoup d'éléments, je n'ai pas fait le ménage depuis longtemps), et qu'il a donc pris la « prochaine » taille minimale.
- guidelines.pdf est lourd : c'est normal, c'est un vrai fichier, sa taille est donc la taille de son contenu (de ses blocs).

On peut également demander à `ls` d'afficher le numéro des inodes : il s'agit de l'option `-li` :

```
22154963 100K -rw-rw-r-- 2 antoine antoine 99K déc. 28 15:15 guide-hard
22154963 100K -rw-rw-r-- 2 antoine antoine 99K déc. 28 15:15 guidelines.pdf
22160938 0 lrwxrwxrwx 1 antoine antoine 14 févr. 5 10:49 guide-symb -> guidelines.pdf
```

FIGURE 9.11 – Un fichier `guidelines.pdf`, un lien physique `guide-hard` et un lien symbolique `guide-symb` vers lui. Affichage obtenu via `ls -lshi`.

Sur la capture d'écran de la figure 9.11, une première coordonnée s'est ajoutée : il s'agit du numéro de l'inode des éléments concernés. Notez que :

- Le lien symbolique et sa cible n'ont pas le même inode. Ils n'ont pas non plus le même poids, ni la même dernière date de modification (la dernière modification du lien est la création du lien). Le lien symbolique est indiqué comme étant un lien.
- Le lien physique et sa cible sont indiscernables. Et c'est normal, ils sont rigoureusement identiques ! Notez que la dernière modification du lien physique correspond à la dernière modification du fichier d'origine, puisque ce sont les mêmes ; et en particulier la dernière modification du lien physique n'est pas « sa création ». Le lien physique n'est pas indiqué comme étant un lien ; et d'ailleurs ce n'en est pas vraiment un... c'est juste le fichier d'origine.

0.6 Vrac de vocabulaire

Définition 2 (Vrac de vocabulaire).

- Chaque noeud a un certain nombre (éventuellement nul) d'**enfants** qui sont les racines des sous-arbres.
- L'**arité** d'un noeud est son nombre d'enfants.
- Un noeud d'arité 0 (sans enfants) est appelé **feuille**, un noeud d'arité non-nulle est appelé **noeud interne**.
- Si x est un enfant de y , on dit que y est le **parent** de x . La **racine** est le seul noeud qui n'a pas de père.
- Les enfants d'un même parent sont des **adelphe**s.
- Les **ancêtres** d'un noeud sont ses parents successifs jusqu'à la racine (parent, parent du parent, etc).
- La **profondeur** d'un noeud est la longueur du chemin (en nombre d'arêtes) qui relie la racine à ce noeud. La racine a profondeur 0, ses enfants 1, etc.
- L'étage p **étage** d'un arbre est l'ensemble des noeuds à profondeur p .
- La **hauteur** de l'arbre est la profondeur maximale d'un noeud.

FIGURE 9.12 – Illustration du vocabulaire

Remarque. Les termes « père / frère / fils » au lieu de « parent / adelphe / enfant » sont aussi utilisés. Ils ont tendance à l'être de moins en moins, notamment en anglais où l'on préfère très clairement « parent/sibling/child ».

0.7 Construction inductive

Théorème 3 (Induction structurelle).

Pour raisonner sur les arbres, on peut utiliser des inductions structurelles. Autrement dit, on procède généralement ainsi pour prouver une propriété vraie sur un arbre :

- Initialisation : la prouver vraie sur les feuilles.
- Hérédité : prouver que si elle est vraie pour les sous-arbres, alors elle est vraie pour l'arbre

Démonstration. Les arbres sont définis inductivement. □

Remarque. On peut aussi à la place faire des récurrences sur la hauteur ou sur le nombre de noeuds.

Théorème 4 (Fonctions inductives).

Pour définir une fonction sur les arbres, on peut procéder inductivement. Autrement dit, on procède généralement pour définir une fonction f qui prend en entrée un arbre :

- Initialisation : définir f sur les feuilles.
- Hérédité : définir f sur l'arbre à l'aide de f sur les sous-arbres.

Démonstration. Idem. □

1 Arbres binaires et implémentation

Dans toute cette partie, on considère des arbres dont les noeuds contiennent une valeur appelée « étiquette ». On note \mathcal{E} l'ensemble des étiquettes possibles.

1.0 Arbres binaires

1.0.0 Définition

Définition 5 (Arbres binaires, informel).

Un arbre binaire est un arbre où l'arité de tous les noeuds vaut au plus 2.

Définition 6 (Arbres binaires, formel).

L'ensemble $\mathcal{A}(\mathcal{E})$ des arbres binaires (non-stricts) étiquetés par \mathcal{E} est défini inductivement comme suit :

- L'arbre vide \perp (auss appelé Nil) est un arbre de $\mathcal{A}(\mathcal{E})$.
- Relier deux arbres binaires par une racine donne un arbre binaire : pour tout $g, d \in \mathcal{A}(\mathcal{E})$ et $x \in \mathcal{E}$, le noeud $N(g, x, d)$ est un arbre de $\mathcal{A}(\mathcal{E})$.
 g est appelé l'**enfant gauche** de ce noeud et d l'**enfant droit**. x est appelé l'**étiquette** du noeud.

Remarque.

- J'abrégérai parfois $N(g, x, d)$ en (g, x, d) .
- Quand un noeud n'a que des enfants \perp , on dit souvent qu'il n'a pas d'enfants. De même, s'il a un seul enfant non- \perp , on dit souvent qu'il a un seul enfant. Ainsi, une feuille est un noeud de la forme $N(\perp, x, \perp)$; et les autres noeuds non- \perp sont les noeuds internes.
- Un arbre binaire est un arbre dans lequel chaque noeud a au plus deux enfants.
- Quand on représente un arbre binaire, on omet souvent les Nil pour alléger le tout :

(a) Un arbre binaire

(b) Le même arbre binaire

FIGURE 9.13 – Deux façons de représenter un arbre binaire.

- Le fait d'être enfant gauche ou droit à son importance. Ainsi, les deux arbres ci-dessous ne sont pas les mêmes :

(a) $N(N(2, Nil, Nil), 1, Nil)$ (b) $N(Nil, 1, N(2, Nil, Nil))$

FIGURE 9.14 – Deux arbres binaires distincts.

Définition 7 (Hauteur, formel).

La **hauteur** h d'un arbre se formalise ainsi :

- $h(\perp) = -1$
- Pour tout $(g, d) \in \mathcal{A}(\mathcal{E})$ et $x \in \mathcal{E}$, on pose $h(N(g, x, d)) = 1 + \max(h(g), h(d))$

Pourquoi cette définition ? Parce que la hauteur est la longueur du plus long chemin descendant de la racine à une feuille. D'où le cas récursif. Mais pourquoi ce cas de base ? Parce que l'on a envie de dire que dans l'arbre qui est réduit à une feuille, le plus long chemin descendant est de longueur 0. Il faut pour cela poser $h(\perp) = -1$. Au fond, tout cela provient du fait que l'on veut ignorer les \perp lorsque l'on raisonne sur des dessins.

(a) Un arbre binaire de hauteur 2.

(b) Un arbre binaire de hauteur 4.

FIGURE 9.15 – Hauteur dans un arbre binaire

Définition 8 (Étage entièrement rempli).

On dit que l'étage p d'un arbre binaire est entièrement rempli s'il contient 2^p noeuds.

Proposition 9.

L'étage p d'un arbre binaire est au plus entièrement rempli.

Démonstration. Soit A un arbre binaire. Montrons qu'un étage p de A a au plus 2^p noeud. Procédons par récurrence⁵ sur $p \in (\mathbb{N}, \leq)$.

5. Pour une (rare) fois, ce n'est pas une induction structurelle ! En effet, le cas de base de la profondeur est la racine et non les feuilles.

- Initialisation : par définition, un noeud à profondeur 0 a un chemin vide depuis la racine : autrement dit, c'est la racine. Il y a dans A au plus une racine, or $1 = 2^0$ d'où l'initialisation.
- Hérédité : soit $p \in \mathbb{N}^*$ tel que la propriété est vraie au rang $p - 1$, montrons-la vraie au rang p . Tout noeud de A à profondeur p est l'enfant d'un (et d'un seul) noeud de A à profondeur $p - 1$. Réciproquement, chaque noeud à profondeur $p - 1$ a au plus deux enfants car l'arbre est binaire. Il s'ensuit que le nombre de noeuds à profondeur p est au plus le double du nombre de noeuds à profondeur $p - 1$. En appliquant l'hypothèse de récurrence et en multipliant par deux, on obtient l'hérédité.
- Conclusion : On a prouvé la propriété vraie pour toute profondeur p dans A . Puisque A était un arbre binaire quelconque, on a montré la propriété vraie pour toute profondeur de tout arbre binaire.

□

Remarque. Pour que cette définition soit pertinente, il faudrait aussi prouver que la borne peut-être atteinte. Les arbres parfaits fourniront un tel exemple.

Proposition 10 (Nombre de noeud et hauteur).

Un arbre binaire (quelconque) de hauteur h à n noeuds. On a :

$$n \leq 2^{h+1} - 1$$

Démonstration. Le cas d'égalité sera prouvé dans la suite du cours. Montrons que $n \leq 2^{h+1} - 1$, par induction structurale sur l'arbre.

- Initialisation : \perp est de hauteur -1 et a $n = 0$ noeuds. On a bien $0 \leq 2^{-1+1} - 1 = 1 - 1 = 0$, d'où l'initialisation.
- Hérédité : Soit $N(g, x, d)$ un arbre tel que g et d vérifient la propriété, montrons que l'arbre la vérifie aussi. On a :

$$h(g, x, d) = 1 + \max(h(g), h(d))$$

$$n(g, x, d) = 1 + n(g) + n(d)$$

D'où $h(g) \leq h(g, x, d) - 1$ et de même pour $h(d)$, et donc :

$$\begin{aligned} n(g, x, d) &= 1 + n(g) + n(d) \\ &\leq 1 + (2^{h(g)+1} - 1) + (2^{h(d)+1} - 1) && \text{par hypothèse d'induction} \\ &\leq -1 + 2^{h(g,x,d)} + 2^{h(g,x,d)} && \text{car } h(g) \leq h(g, x, d) - 1 \\ &\leq -1 + 2^{h(g,x,d)+1} \end{aligned}$$


D'où l'hérédité.

- Conclusion : On a prouvé que pour tout arbre binaire, $n \leq 2^{h+1} - 1$.

□

1.0.1 Implémentation


Voici des types pour implémenter de tels arbres en C et OCaml. On veut pour chaque noeud stocker son sous-arbre gauche, son étiquette (« key » en anglais) et son sous-arbre droit.



```

1 struct tree_s {
2     struct tree_s *left;
3     int key;
4     struct tree_s *right;
5 };
6 typedef struct tree_s tree;

```



```

1 type 'a tree =
2     Nil
3   | Node of 'a tree * 'a * 'a tree

```

Exemple. Les légendes deux arbres de la figure 9.14 sont des `int tree` avec le type ci-dessus (avec \perp pour Nil et N pour Node).

Lorsque les étiquettes de l'arbre sont en bijection avec $\llbracket 0; n \rrbracket$ (où n est le nombre de noeuds), on peut aussi utiliser une représentation par tableau : dans `arr[i]`, on stocke l'étiquette du parent du noeud d'étiquette i .

(a) Un arbre binaire aux étiquettes en bijection avec $\llbracket 0; 8 \rrbracket$.

(b) Le tableau correspondant.

FIGURE 9.16 – Représentation par tableau de parenté

Remarque. Cette représentation servira surtout en MPI, pour implémenter l'algorithme Union-Find.

1.1 Arbres binaires stricts

1.1.0 Définitions

Définition 11 (Arbres binaires stricts, V0).

Un arbre binaire strict est un arbre binaire non-vide où l'arité de tous les noeuds internes vaut exactement 2.

Définition 12 (Arbres binaires stricts, V1).

L'ensemble $\mathcal{S}(\mathcal{E})$ des arbres binaires stricts est défini inductivement par :

- Les feuilles sont des arbres : pour tout $x \in \mathcal{E}$, $F(x)$ est un arbre de $\mathcal{S}(\mathcal{E})$.
- Relier deux arbres stricts par une racine donne un arbre strict : pour tout $g, d \in \mathcal{S}(\mathcal{E})$ et $x \in \mathcal{E}$, le noeud $N(g, x, d)$ est un arbre de $\mathcal{S}(\mathcal{E})$.

Démonstration. Prouvons rapidement qu'un arbre défini par la déf V1 vérifie le fait que tout noeud interne est d'arité 2, par induction structurelle :

- Une feuille n'est pas un noeud interne.
- Un noeud $N(g, x, d)$ vérifie le fait que ni g ni d ne sont vides. Il a donc pour arité 2, et comme ce n'est pas une feuille c'est un noeud interne. Par hypothèse d'induction, les noeuds de ses sous-arbres vérifient également V1.

Ainsi, cette définition construit bien des arbres binaires stricts correspondant à la première définition.

Prouvons maintenant que tout arbre binaire de définition V0 correspond à la définition V1. On procède par récurrence forte sur la hauteur $h \in \llbracket 0; +\infty \rrbracket$, de A un arbre binaire non-vide dont tous les noeuds internes sont d'arité 2 :

- **Initialisation** : Si $h(A) = 0$, alors A est une feuille : c'est bien un cas de la définition V1.
- **Hérédité** : Soit $h > 0$ tel que la propriété est vraie pour tout $h' < h$ et A un arbre de hauteur h . Comme $h > 0$, A est de la forme $N(g, x, d)$ avec g et d de hauteur inférieure. Par hypothèse A est d'arité 2 et pas une feuille, g et d sont non-vides. On peut donc appliquer l'hypothèse de récurrence à g et d : g et d correspondent à la définition v1. $N(g, x, d)$ correspond alors à la définition V1. D'où l'hérédité.

□

Remarque.

- On peut aussi prendre la définition des arbres binaires et y imposer dans le cas inductif que si un des enfant est \perp , alors l'autre doit également l'être.
- On peut aussi les définir en prenant deux ensembles d'étiquettes : un pour les feuilles et un pour les noeuds internes. Encore une fois, plein de variantes existent !
- Dans certains cours, « arbre binaire » signifie « arbre binaire strict ». Ce n'est pas le cas ici, notamment parce que j'ai envie de dire qu'un arbre binaire de recherche est un arbre binaire.

(a) Un arbre binaire **pas** strict.

(b) Un arbre binaire strict.

FIGURE 9.17 – Illustration des arbres binaires stricts

Proposition 13 (Noeud et feuilles dans un arbre binaire strict).

Dans un arbre binaire strict à n_i noeuds internes et f feuilles, on a :

$$f = n_i + 1$$

Démonstration. Nous l'avons déjà prouvé sur les mobiles de Calder, qui sont des arbres binaires stricts déguisés. □

1.1.1 Implémentation

Pour implémenter de tels arbres en C, on reprend le type des arbres binaires et on code de sorte à ce que si l'un des deux enfants est vide alors l'autre aussi. En OCaml, on peut aussi procéder ainsi mais on peut également traduire la définition récursive :

```

1 type 'a strict =
2   Leaf of 'a
3   | Node of 'a strict * 'a * 'a strict

```



1.2 Arbres parfaits et complets

1.2.0 Définitions

Définition 14 (Arbre parfait, informel).

Un arbre parfait est un arbre dont tous les étages non-vides sont entièrement remplis.

Proposition 15 (Nombre de noeuds d'un arbre parfait).

Un arbre binaire de hauteur h est parfait si et seulement si il a $2^{h+1} - 1$ noeuds.

Démonstration. L'implication directe se traite à peu près comme dans la preuve de la propriété 10 ; sauf qu'ici on ne majore pas mais on montre l'égalité : on prouve par récurrence sur $p \in (\llbracket 0; h \rrbracket)$ que l'étage p contient exactement 2^p noeuds (tout noeud de l'étage $p + 1$ est enfant d'un noeud de l'étage p , et tout noeud de l'étage p a 2 enfants afin de remplir le plus possible l'étage $p + 1$; on conclut par récurrence). On termine l'implication directe en sommant les 2^p .

Le sens réciproque est lui aussi très similaire : un arbre de hauteur h a h étages qui contiennent chacun au plus 2^p noeuds. Si la somme vaut $2^{h+1} - 1$, alors toutes ces majorations sont atteintes : chaque étage p a 2^p noeuds, donc est entièrement plein. \square

Définition 16 (Arbre parfait, formel).

L'ensemble $\mathcal{P}(\mathcal{E})$ des arbres parfaits est défini inductivement par :

- L'arbre vide \perp est un arbre parfait de $\mathcal{P}(\mathcal{E})$.
- Relier deux arbres parfaits de même hauteur par une racine donne un arbre binaire : pour tout $g, d \in \mathcal{P}(\mathcal{E})$ tels que $h(g) = h(d)$ et $x \in \mathcal{E}$, le noeud $N(g, x, d)$ est un arbre parfait de $\mathcal{P}(\mathcal{E})$.

Démonstration. Montrons que cette seconde définition implique la première : prouvons qu'un arbre ainsi défini a bien tous ses étages entièrement remplis. Procédons par induction structurelle :

- Initialisation : Immédiat puisqu'il n'y a aucun étage non-vide.
- Hérédité : Soit $N(g, x, d) \in \mathcal{P}(\mathcal{E})$ tel que g et d vérifient la propriété.

L'étage de profondeur 0 de p est entièrement rempli puisqu'il contient 1 noeud : la racine.

Par définition de la profondeur, un noeud de profondeur p dans g est de profondeur $p + 1$ dans $N(g, x, d)$ (et de même pour d). Donc l'étage de profondeur $p > 0$ est constitué des noeuds de profondeur $p - 1$ dans g et dans d . Or par hypothèse d'induction, il y en a 2^{p-1} dans g et de même dans d . D'où l'hérédité.

FIGURE 9.18 – Schéma de la preuve

- Conclusion : on a prouvé vraie la propriété par induction structurelle.

Le sens réciproque est un jeu d'écriture qui est technique (prouver qu'un objet peut correspondre à une définition inductive est souvent compliqué à rédiger de manière claire, concise et rigoureuse). Je vais simplement montrer un morceau de la preuve : si un arbre non-vide est parfait (définition informelle), alors ses deux sous-arbres ont la même hauteur. Notons $N(g, x, d)$ un tel arbre et h sa hauteur. Par définition, $h(g) \leq h - 1$ et de même pour d . L'étage de profondeur h de $N(g, x, d)$ est plein par hypothèse, et composé de l'étage $h - 1$ de g et de l'étage $h - 1$ de d . Pour qu'il soit plein, il faut que ces deux étages des sous-arbres soient plein, et en particulier que ces deux sous-arbres soient de même hauteur $h - 1$. \square

Exemple.

FIGURE 9.19 – Un arbre parfait à 15 noeuds

Proposition 17 (Strict vs parfait).

Un arbre binaire strict A est parfait si et seulement si toutes ses feuilles sont à même profondeur $h(A)$

Démonstration. Prouvons d'abord le sens direct, par induction structurelle sur A un arbre binaire strict (V1) qui est également parfait.

- Initialisation : c'est immédiat pour la feuille.
- Hérédité : Soit A un arbre binaire strict parfait de la forme $N(g, x, d)$ avec g et d binaires stricts vérifiant la propriété. Or, les feuilles de A sont les feuilles de g et de d . Celles-ci sont à profondeur respectives $h(g)$ dans g et $h(d)$ dans d , donc $h(g) + 1$ et $h(d) + 1$ dans A . Or, par construction des arbres parfaits, $h(g) = h(d) = h(A) - 1$. D'où l'hérédité.
- Conclusion : on a prouvé par induction structurelle que le sens direct est de l'implication est vrai. Pour le sens réciproque, on procède de même. \square

Définition 18 (Arbre binaire complet).

Un arbre binaire est dit complet si tous es étages non-vides sont entièrement remplis sauf éventuellement le dernier, et que celui-ci est rempli de gauche à droite.

Remarque. Ce dernier point n'est pas présent dans toutes les définitions. Je le mets ici pour simplifier la définition des tas.

Exemple.

FIGURE 9.20 – Un arbre binaire complet de hauteur 3

Proposition 19.

Un arbre binaire parfait est complet.

Démonstration. Immédiat. □

1.2.1 Implémentation des arbres binaires complets

On peut représenter les arbres binaires complets inductivement, comme des arbres binaires. Mais on peut également les stocker dans un tableau : l'idée est d'écrire dans le tableau les étiquettes des noeuds de l'arbre dans le sens de lecture (de gauche à droite et de haut en bas).

(a) Sens de lecture.

(b) Représentation dans un tableau.

FIGURE 9.21 – Représentation dans un tableau de l'arbre binaire complet précédent.

Proposition 20 (Indexation d'un arbre binaire complet dans un tableau).

Avec cet représentation :

- L'étiquette de la racine est stockée dans la case d'indice 0.
- Les enfants gauches et droits du noeud d'indice i correspondent respectivement aux indices $2i + 1$ et $2i + 2$.
- Le parent du noeud d'indice i correspond à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$

Démonstration. Admis (pour l'instant). □

Remarque.

- D'expérience, vous avez du mal à retenir cette formule. Deux possibilités : faire un effort particulier et régulier pour l'apprendre par coeur (comme les formules trigos en maths), ou être capable de la retrouver très rapidement sur des dessins. Dans tous les cas, comme tout le cours, il faut la connaître !
- Certaines implémentations proposent de plutôt ne pas utiliser la case d'indice 0 et commencer à l'indice 1. Cela fait que les enfants sont $2i$ et $2i + 1$, et le parent $\left\lfloor \frac{i}{2} \right\rfloor$. C'est en général avec ces formules-ci que vous confondez.

Pro-tip : Si vous utilisez cette représentation dans un code, commencez par vous définir des fonctions `gauche`, `droite` et `parent` qui renvoie les bons indices. Sinon, vous ferez tout ou tard une erreur d'inattention.

1.3 Peigne

Définition 21.

Un **peigne gauche** (resp. **peigne droit**) est un arbre binaire dont tous les noeuds internes sont de la forme $N(g, x, \perp)$ (resp. $N(\perp, x, d)$).

Exemple.

(a) Peigne gauche.

(b) peigne droit.

FIGURE 9.22 – Les deux peignes

Remarque.

- Un peigne est, fondamentalement, une liste chaînée.
- Ces arbres correspondent à un déséquilibre maximal entre le sous-arbre gauche et le sous-arbre droit. Ils nous donnent ainsi des pires cas de beaucoup d’algorithmes sur les arbres.

1.4 Arbres quelconques

1.4.0 Transformation LCRS

Définition 22 (Transformation LCRS).

La transformation LCRS (« **L**eft **C**hild **R**ight **S**ibling ») permet de transformer un arbre quelconque en arbre binaire. Elle fonctionne ainsi :

- L’arbre binaire a les mêmes noeuds que l’arbre d’origine, mais pas les mêmes arêtes.
- La racine de l’arbre binaire est la racine de l’arbre d’origine.
- Pour les autres noeuds, on leur donne comme enfant gauche la racine de leur premier sous-arbre et comme enfant droit leur prochain adelphe (« le noeud qui est à droite d’eux dans l’arbre d’origine »).

Exemple.

(a) Un arbre et les liens utiles à LCRS

(b) Transformation LCRS de l’arbre

FIGURE 9.23 – Transformation LCRS

Proposition 23.

La transformation LCRS est injective, et on peut également coder la transformation inverse.

Démonstration. Injection : admis. Inverse : cf TP. □

Remarque.

- À la main sur des exemples, la transformation et la dé-transformation se calculent assez bien. Les programmer est un bon exercice étoilé de programmation.
- Prouver l'injection se fait assez bien, mais il faudrait pour cela formaliser la transformation par un programme. Nous le ferons peut-être en TD étoilé après le TP.

1.4.1 Type général

Voici un type OCaml pour des arbres binaires quelconques. L'idée est que pour chaque noeud on stocke la liste de ses sous-arbres.

```
1 type 'a tree =
2   Nil
3   | Node of 'a * 'a tree list
```



Exemple. Le code OCaml ci-dessous correspond à l'arbre de la figure 9.24 :

```
17 let leaf x = Node (x, [])
18
19 let general =
20   let t0 = Node (1, [Node (2, [leaf 3; leaf 4]);
21                       Node (5, [leaf 6])
22                     ]) in
23   let t1 = Node (7, [leaf 8; leaf 9; leaf 10; leaf 11]) in
24   let t2 = Node (12, [leaf 13;
25                      Node (14, [leaf 15; leaf 16]);
26                      leaf 17;
27                      Node (18, [leaf 19])
28                    ]) in
29   Node (0, [t0; t1; t2])
```



FIGURE 9.24 – L'arbre general

2 Parcours d'arbres

Définition 24.

Un **parcours d'arbre** est la visite des noeuds d'un graphe dans un ordre particulier, pour leur appliquer un certain « traitement ».

Remarque.

- Par exemple, on peut vouloir afficher les noeuds, calculer l'expression associée à un arbre, reconstituer les appels récursifs qui ont lieu, etc. En fait, la plupart des algorithmes qui utilisent l'entiereté d'un arbre sont des parcours.
- Dans ce cours, on traitera les enfants « de gauche à droite » pour simplifier. On peut bien sûr généraliser.

2.0 Parcours en profondeur

Définition 25 (Parcours en profondeur).

Le **parcours en profondeur** (« Depth First Search ») consiste à se déplacer dans le graphe en descendant le plus possible jusqu'à une feuille, puis en remontant jusqu'à pouvoir descendre à nouveau, et ainsi de suite.

Exemple.

FIGURE 9.25 – Parcours en profondeur d'un arbre A_{dfs}

Remarque.

- C'est un parcours naturellement récursif : on descend dans les enfants les uns après les autres.
- L'exploration exhaustive consiste à parcourir en profondeur l'arbre des extensions des solutions partielles jusqu'à trouver une solution valide (et s'arrêter alors).

FIGURE 9.26 – Exploration exhaustive pour SUBSET-SUM avec $E = \{2; 8; 3\}$ et $t = 5$

- Comme dans le retour sur trace, la « remontée » est correspond par le fait de quitter les appels récursifs : il n'y a pas à la coder.

Voici une façon d'implémenter un tel parcours sur un arbre d'arité quelconque⁶ (cf fin de la section précédente). On va y appliquer la fonction `visite` à chacun des noeuds. On y utilise la fonction `List.iter` qui permet d'évaluer une fonction sur tous les éléments d'une liste :

```

8  (** Applique la fonction f à chacune des étiquettes *)
9  let rec dfs visite tree =
10     match tree with
11     | Nil -> ()
12     | Node (etq, children) ->
13         let _ = visite etq in
14         List.iter (dfs visite) children

```



Exemple. Avec `general` l'arbre de la fin de la section précédente, l'expression `dfs (Printf.printf "%d ") general` a pour effet secondaire d'afficher 0 ... 19.

Remarque. Dans un arbre binaire :

- Le parcours en profondeur est plus simple à écrire (on peut éviter le `List.iter`).
- On passe trois fois par chaque noeud : avant le parcours de son enfant gauche, entre le parcours de ses deux enfants, et après le parcours de son enfant droit. Cela mène à trois notions : parcours préfixe, infix et postfixe.

2.0.0 Ordre préfixe

Définition 26 (Ordre préfixe pour un DFS).

Dans un parcours en profondeur d'un arbre, l'**ordre préfixe** consiste à effectuer le traitement d'un noeud *avant* d'explorer ses enfants.

Exemple.

- **Cet ordre est défini peu importe l'arité, pas uniquement pour les arbres binaires !**
- La fonction `dfs` précédente affiche l'étiquette d'un noeud avant d'explorer récursivement les enfants ; c'est donc un parcours en profondeur préfixe.
- Le parcours préfixe de l'arbre A_{dfs} traite les noeuds dans l'ordre 1, 2, 4, 5, 8, 9, 3, 6, 7. Notez que l'on visite un noeud « lorsque le trait du chemin passe à gauche ».

Dans un arbre binaire, un parcours en profondeur préfixe qui applique f à tous les éléments peut s'écrire ainsi :

```

44 let rec prefixe visite tree =
45     match tree with
46     | Nil -> ()
47     | Node (gauche, x, droite) ->
48         let _ = visite x in
49         let _ = prefixe visite gauche in
50         prefixe visite droite

```



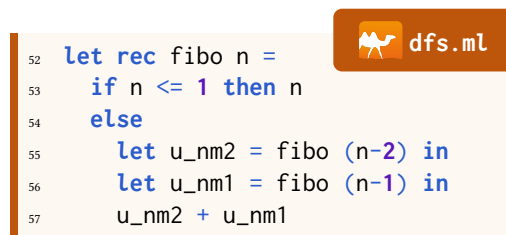
6. Bien sûr, les valeurs renvoyées et d'autres points peuvent changer selon le problème à résoudre. Typiquement, ignorer la sortie de `visite` n'est pas toujours pertinent.

Proposition 27 (Dates d'ouverture dans une fonction récursive).

Lors de l'exécution d'une fonction récursive, l'*empilement* des blocs d'activation des appels sur la pile mémoire correspond à un parcours en profondeur préfixe de l'arbre d'appels.

Rappelons que le bloc d'activation d'une fonction est créé (et donc empilé) lorsque l'appel débute, et supprimé (donc dépilé) lorsqu'il se termine.

Exemple.



```

52 let rec fibo n =
53   if n <= 1 then n
54   else
55     let u_nm2 = fibo (n-2) in
56     let u_nm1 = fibo (n-1) in
57     u_nm2 + u_nm1

```

(a) Une fonction récursive `fibo`

(b) Arbre d'appels de `fibo 3`

(c) Les états successifs de la pile mémoire

FIGURE 9.27 – Arbre d'appel et pile mémoire

Remarque. Dans utop, `#trace fibo;;` permet d'avoir un affichage des ouvertures et fermetures des appels récursifs.

2.0.1 Parcours infixe

Définition 28 (Ordre infixe pour un DFS).

Dans un parcours en profondeur d'un arbre binaire, l'**ordre infixe** consiste à effectuer le traitement d'un noeud *entre* les traitements de ses deux enfants.

Exemple.

- On peut essayer de généraliser aux arbres quelconques en disant que le traitement a lieu entre le traitement de deux enfants (avec un cas particulier pour les noeuds d'arité 1).
- Le parcours infixe de A_{dfs} traite les noeuds dans l'ordre 4, 2, 8, 5, 9, 1, 6, 3, 7. Notez que l'on visite un noeud « lorsque le trait du chemin passe *en dessous* ».
- Les mouvements de la résolution récursive des tours de Hanoï ont lieu dans l'ordre infixe : d'abord on fait un premier appel récursif, puis on effectue un mouvement, puis un second appel récursif.
- Nous verrons que le parcours infixe tri un arbre binaire de recherche.

Dans un arbre binaire, un parcours infixe ressemble à ceci :

```

60 let infixe visite tree =
61   match tree with
62   | Nil -> ()
63   | Node (gauche, x, droite) ->
64     let _ = prefixe visite gauche in
65     let _ = visite x in
66     prefixe visite droite

```



2.0.2 Parcours postfixe

Définition 29 (Ordre postfixe pour un DFS).

Dans un parcours en profondeur d'un arbre, l'**ordre postfixe** consiste à effectuer le traitement d'un noeud *après* les traitements de deux enfants.

Exemple.

- Le parcours postfixe sur A_{dfs} traite les noeuds dans l'ordre 4, 8, 9, 5, 2, 6, 7, 3, 1. Notez que l'on visite un noeud « lorsque le trait du chemin passe à droite ».
- La fonction ci-dessous qui évalue une expression arithmétique (cf section 1) effectue les calculs dans l'ordre postfixe :

```

27 let rec eval e =
28   match e with
29   | Int n -> n
30   | Var _ -> failwith "Je ne connais pas la valeur des variables."
31   | Fun (f, e0, e1) -> f (eval e0) (eval e1)

```



Proposition 30 (Dates de fermeture dans une fonction récursive).

Lors de l'exécution d'une fonction récursive, le *dépilement* des blocs d'activation des appels sur la pile mémoire correspond à un parcours en profondeur postfixe de l'arbre d'appels.

Exemple. Cf figure 9.27

2.0.3 Écriture itérative

Il est possible d'écrire un parcours en profondeur d'arbre de manière itérative. Cela peut permettre un contrôle un peu plus fin⁷, mais est plus difficile d'écrire sans bug. L'idée est de remplacer la récursivité par une pile :

Algorithme 12 : Parcours en profondeur d'arbre, itératif

Entrées : A un arbre; et $VISITE$ une fonction de traitement

```

1   $P \leftarrow$  pile vide
2  EMPILER la racine de  $A$  sur  $P$ 
3  tant que  $P$  est non-vide faire
4     $n \leftarrow$  DÉPILER( $P$ )
5     $VISITE(n)$ 
6    pour chaque enfant  $e \neq \perp$  de  $n$  faire
7      EMPILER  $e$  sur  $P$ 

```

7. Et encore, l'utilisation d'exceptions dans une fonction récursive permet d'implémenter des comportements exceptionnels.

Remarque. Cette version-ci correspond à un DFS préfixe, on peut bien sûr l'adapter aux autres ordres.

Exemple.

(a) L'arbre parcouru

(b) États de la pile P en débuts d'itération

FIGURE 9.28 – Algorithme de parcours en profondeur itératif

2.1 Parcours en largeur

Définition 31 (Parcours en largeur).

Le **parcours en largeur** (« **Breadth First Search** ») consiste à parcourir les sommets par profondeur croissante.

Exemple.

FIGURE 9.29 – Parcours en largeur d'un arbre A_{bfs}

2.1.0 Écriture itérative

Le parcours en largeur n'a pas, contrairement au DFS, une écriture récursive simple. En fait, il n'est pas basé sur un ordre LIFO (comme la récursivité) mais FIFO :

Algorithme 13 : Parcours en largeur d'arbre

Entrées : A un arbre; et $VISITE$ une fonction de traitement

```

1  $F \leftarrow$  file vide
2 ENFILER la racine de  $A$  dans  $F$ 
3 tant que  $F$  est non-vide faire
4    $n \leftarrow$  DÉFILER( $F$ )
5    $VISITE(n)$ 
6   pour chaque enfant  $e \neq \perp$  de  $n$  faire
7     ENFILER  $e$  dans  $F$ 
```

Exemple.

(a) L'arbre parcouru

(b) États de la file F en débuts d'itération

FIGURE 9.30 – Algorithme de parcours en largeur

2.1.1 (Mi-HP) Preuve du pseudo-code

Prouvons que ce pseudo-code est bien un parcours en largeur. C'est un bon entraînement⁸.

Théorème 32.

Le pseudo-code proposé est bel et bien un parcours en largeur, c'est à dire qu'il parcourt les sommets par profondeur croissante.

Démonstration. Le pseudo-code utilise une file et n'effectue que 4 opérations qui appartiennent au type abstrait : obtenir une file vide, tester si une file est vide, ENFILER et DÉFILER. Nous allons la considérer implémentée à l'aide de deux piles : cette implémentation était totalement correcte, faire ce choix ne modifiera pas le déroulé de l'algorithme. De plus, nous allons introduire une variable p qui stocke la profondeur en cours. Celle-ci n'interagira pas avec les autres variables, aussi l'ajouter ne modifie pas non plus le déroulé. On obtient le pseudo-code suivant :

```

1  $P_{in} \leftarrow$  pile vide
2  $P_{out} \leftarrow$  pile vide
3  $p \leftarrow -1$ 
4 EMPILER la racine de  $A$  sur  $P_{in}$ 
5 tant que  $P_{out}$  ou  $P_{in}$  est non-vide faire
6   si  $P_{out}$  est vide alors
7     RENVERSER  $P_{in}$  sur  $P_{out}$ 
8      $p \leftarrow p + 1$ 
9    $n \leftarrow$  DÉPILER( $P_{out}$ )
10  VISITE( $n$ )
11  pour chaque enfant  $e \neq \perp$  de  $n$  faire
12    EMPILER  $e$  sur  $P_{in}$ 

```

Pour prouver la correction partielle de l'algorithme, nous allons prouver les invariants suivants :

- I_{out} : « Tous les noeuds de P_{out} sont à profondeur p . »
- I_{in} : « Tous les noeuds de P_{in} sont à profondeur $p+1$. »

Expliquons tout d'abord à quoi servent ces invariants. À chaque itération, le sommet visité est un sommet de P_{out} donc d'après I_{in} à profondeur p ; or comme p est croissant on en déduit la correction de l'algorithme⁹. I_{in} sert à prouver I_{out} .

⁸. La somme des entiers ça va bien 5 minutes.

⁹. On pourrait formaliser cela avec des invariants supplémentaires... je préfère alléger le tout. Le coeur de la preuve porte sur I_{in} et I_{out} .

Avant la 1ère itération : I_{out} est trivial. De plus, P_{in} contient un seul noeud : la racine, qui est à profondeur 0. Or, $p = -1$: d'où l'initialisation de I_{in} .

Conservation : Supposons les deux invariants vrais au début d'une itération, montrons-les vrais à la fin. On utilisera les notations prime.

On va distinguer deux cas :

- Si P_{out} n'est pas vide. Alors on ne rentre pas dans le Si des lignes 6-8. Donc $p' = p$.
La seule modification faite à P_{out} est un DEPILE : ainsi, tous les éléments de P_{out}' étaient dans P_{out} . Comme $p = p'$, I_{out} entraîne I_{out}' .
Enfin, les seules modifications faites à P_{in} sont les EMPILE des enfants de noeud n . Or, noeud est à profondeur p donc ses enfants à $p+1$. Ainsi, les éléments de P_{in}' sont soit ces enfants à profondeur $p+1$, soient des éléments de P_{in} qui sont à profondeur $p+1$ d'après I_{in} . Comme $p = p'$, on conclut de même que I_{in}' est vrai.
- Si P_{out} est vide. Alors on rentre dans le Si, et donc :
 - $p' = p+1$
 - Après la ligne 7, le contenu de P_{out} est l'ancien contenu de P_{in} (en ordre renversé). En particulier, d'après I_{in} ce sont des noeuds de profondeur $p+1 = p'$. De même que dans le cas précédent, on en déduit I_{out}' .
 - Après la ligne 7, P_{in} est vidée (et ne sera re-remplie qu'à la 12).
 Enfin, P_{in}' contient uniquement les enfants de du noeud n . Or, n est à profondeur p' , donc ses enfants à $p'+1$: d'où I_{in}' .

Conclusion : On a prouvé que les quatre invariants étaient vrais à tout moment de l'exécution de la boucle Tant Que. Comme annoncé précédemment, on déduit de I_{out} et de la monotonie de p que les noeuds sont visités par profondeur croissante. □

2.1.2 (Mi-HP) Preuve, suite et fin

Je vous ai eu ! Nous n'avons pas prouvé *tout* ce qu'il fallait prouver. Nous avons prouvé que les sommets visités le sont par profondeur croissante... mais pas qu'ils sont tous visités. Prouvons cela :

Démonstration. Les visites correspondent aux défilement : on veut en fait prouver que tous les sommets sont défilés. Mais puisque la boucle termine lorsque la file est vide, cela revient à prouver que tous les sommets sont enfilés.

Montrons par récurrence sur la profondeur p d'un noeud quelconque qu'il est bien enfilé.

Initialisation : le seul noeud à profondeur $p = 0$ est la racine. Elle est enfilée à la ligne 2 (ou 4 dans la version modifiée).

Hérédité : supposons que tous les noeuds à profondeur p sont enfilés (peu importe quand), et montrons que tous ceux à profondeur $p + 1$ le sont. Un noeud à profondeur $p + 1$ a un parent à profondeur p , qui a été enfilé. En particulier, ce parent sera défilé. Or, quand on défile un noeud, on enfile tous ses enfants (ligne 11-12). Ainsi, les noeuds à profondeur $p + 1$ seront bien enfilés, d'où l'hérédité.

Conclusion : tous les noeuds sont bien enfilés à un moment ou un autre. On en déduit comme annoncé qu'ils seront bien tous visités. □

3 (HP) Propriétés avancées

3.0 Lemme de lecture unique

Dans le TP sur la notation polonaise inversée (RPN), j'ai affirmé que toute expression pouvait s'écrire ainsi, et sous-entendu que l'écriture d'une notation était unique. Or, la RPN correspond au parcours postfixe de l'arbre d'une expression : autrement dit, le parcours postfixe d'une expression définit.

On appelle cette propriété un lemme de lecture unique :

Théorème 33 (Lemme de lecture unique).

Un arbre (quelconque) est entièrement défini par son parcours préfixe, si celui-ci contient les arités des noeuds en plus des étiquettes.

De même avec le parcours postfixe.

Démonstration. Admis. □

Pour comprendre un peu mieux cela, essayons de reconstruire un arbre à partir d'un tel parcours. On note le résultat du parcours ainsi :

- $prefix(N(x, []))$ est $(x, 0)$
- $prefix(N(x, [a_1; \dots; a_k]))$ est $(x, k) \ prefix(a_1) \ \dots \ prefix(a_k)$

Remarque. J'ai ajouté des parenthèses et des virgules dans (x, k) ; mais on peut les enlever : dans la notation finale alternent étiquette et arité. En particulier, on ne va pas « tricher » en utilisant des parenthèses imbriquées pour reconstruire l'arbre.

Exemple.

FIGURE 9.31 – Arbre correspondant à « 1 3 2 2 5 0 6 0 3 0 4 1 7 0 »

Remarque. La preuve complète repose sur un lemme : « en tout point de la notation débute un unique facteur qui correspond à un arbre ». Mais comme toutes les preuves de lemme de lecture unique, c'est plus lourd que pédagogique... le TP RPN et la figure 9.31 ci-dessus nous suffiront largement.

3.1 Dénombrement des arbres binaires stricts

Dénombrons les arbres binaires stricts. On ignore le contenu des étiquettes : on s'intéresse uniquement à la « forme » de l'arbre binaire strict. On va montrer que leur nombre est lié à la suite de Catalan :

Définition 34 (Nombre de Catalan).

La suite $(c_n)_{n \in \mathbb{N}}$ des nombres de Catalan est définie par récurrence par :

- $c_0 = 1$
- $c_{n+1} = \sum_{k=0}^n c_k c_{n-k}$

Remarque. Les nombres de Catalan sont aussi connus pour être le nombre de mots de Dyck, c'est à dire le nombre de façon de placer n parenthèses ouvrantes et n fermantes de sorte à ce que le tout soit bien parenthésé.¹⁰

¹⁰. C'est un exercice classique de dénombrement dans des concours difficiles ; je vous laisse vous renseigner si cela vous intéresse.

(a) Les $c_3 = 5$ arbres binaires stricts à 3 noeuds internes

(b) Les $c_4 = 14$ arbres binaires stricts à 4 noeuds internes

FIGURE 9.32 – Dénombrement pour $n = 3$ et $n = 4$

Théorème 35.

Il y a c_n arbres binaires stricts non-étiquetés à n noeuds internes.

Remarque. J'insiste, *internes* !

Démonstration. Procédons par récurrence sur le nombre de noeuds internes.

- Il y a exactement 1 arbre binaire strict à 0 noeuds internes : la feuille.
- Supposons la propriété vraie jusqu'au rang n , montrons-la vraie au rang $n + 1$. Un arbre binaire strict à $n + 1$ noeuds internes est constitué de :
 - une racine (qui est un noeud interne).
 - un sous-arbre gauche possédant k noeuds internes avec $0 \leq k \leq n$ (il ne peut pas y en avoir $n + 1$ car la racine est un noeud interne).
 - un sous-arbre droit qui a donc $n + 1 - 1 - k = n_k$ noeuds internes.

Par hypothèse de récurrence, pour chaque k il y a c_k façons de construire un sous-arbre gauche, c_{n-k} de construire un sous-arbre droit, et donc $c_k c_{n-k}$ arbres binaires stricts à $n + 1$ noeuds internes dont le sous-arbre gauche a k noeuds internes. En sommant sur k , on obtient l'hérédité. \square

La suite des nombres de Catalan croît vite, aussi ce théorème nous apprend qu'il y a *beaucoup* binaires stricts, et donc encore plus d'arbres binaires.

Une autre application intéressante de ce théorème est qu'elle peut aider à calculer une expression plus agréable pour c_n :

Lemme 36.

Pour tout $n \in \mathbb{N}$:

$$(n + 2)c_{n+1} = 2(2n + 1)c_n$$

Démonstration. Nous allons interpréter les deux membres de l'égalité comme le cardinal de deux ensembles en bijection :

- À gauche : $(n + 2)c_{n+1}$ est le nombre de paires (A, f) avec A un arbre binaire strict à $n + 1$ noeuds internes et f une feuille de A . En effet, il y a c_{n+1} possibilités pour A , et comme A a $(n + 1) + 1$ feuilles, il reste $n + 2$ possibilités pour f .
- À droite : $2(2n + 1)c_n$ est le nombre de triplets (B, x, ϵ) où B est un arbre strict à n noeuds internes (c_n possibilités), x un noeud quelconque de B ($n + (n + 1)$ possibilités), et $\epsilon \in \{\leftarrow, \rightarrow\}$ (2 possibilités).

Montrons que ces deux ensembles sont en bijection en définissant une fonction bijective φ qui à (A, f) associe (B, x, ϵ) :

- B est obtenu à partir de A en supprimant la feuille f ainsi que son parent. L'adelphe de f est « remonté » à la place de son parent.
- x est le noeud qui a été remonté.
- ϵ vaut \leftarrow si f était enfant gauche, et \rightarrow si elle était enfant droit.

FIGURE 9.33 – Définition de φ

Pour prouver qu'il s'agit d'une bijection, exhibons sa réciproque φ^{-1} . À un triplet (B, x, ϵ) elle associe :

- A est obtenu en insérant un noeud y entre x et son parent. Si $\epsilon = \leftarrow$, on crée une nouvelle feuille enfant gauche de y et le sous-arbre de x devient enfant droit de y ; sinon symétrique.
- f est la feuille qui a été créée.

On peut vérifier qu'il s'agit bien d'une réciproque, d'où la bijection¹¹. □

Ce lemme, obtenu grâce aux arbres, permet d'obtenir une expression non-récursive des nombres de Catalan :

Proposition 37.

Pour tout $n \in \mathbb{N}$:

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

Démonstration. Procédons par récurrence sur \mathbb{N} :

- Pour $n = 0$, on a $c_0 = 1 = \frac{1}{0+1} \binom{0}{0}$.
- Supposons la propriété vraie au rang n , montrons-la vraie au rang $n+1$. On a :

$$\begin{aligned} (n+2)c_{n+1} &= 2(2n+1)c_n && \text{d'après le lemme} \\ &= \frac{2(2n+1)}{n+1} \binom{2n}{n} && \text{par H.R.} \end{aligned}$$

Or :

$$\begin{aligned} \binom{2n+2}{n+1} &= \frac{(2n+2)!}{(n+1)!(n+1)!} \\ &= \frac{(2n+2)(2n+1)(2n)!}{(n+1)^2 n!} \\ &= \frac{(2n+2)(2n+1)}{(n+1)^2} \binom{2n}{n} \end{aligned}$$

Donc :

$$\begin{aligned} (n+2)c_{n+1} &= \frac{2(2n+1)}{n+1} \binom{2n}{n} \\ &= \frac{2(2n+1)}{n+1} \frac{(n+1)^2}{(2n+2)(2n+1)} \binom{2n+2}{n+1} \\ &= \binom{2n+2}{n+1} \end{aligned}$$

□

11. C'est très simple de s'en convaincre; et plus embêtant qu'autre chose d'en donner une preuve formelle (on montre que $\varphi \circ \varphi^{-1} = \varphi^{-1} \circ \varphi = Id$). Cette partie étant déjà du bonus hors-programme, je me limite au plus intéressant.

4 Arbres binaires de recherche

Dans toute cette section, on considère des arbres binaires dont les étiquettes appartiennent à un ensemble *totale*ment ordonné (\mathcal{E}, \leq) . On rappelle que g, x, d est un raccourci pour $N(g, x, d)$.

Les arbres binaires de recherche sont une structure de donnée qui vise à stocker des éléments et pouvoir efficacement :

- Rechercher si un élément est dedans.
- Insérer un nouvel élément dedans.
- Supprimer un élément dedans.

En bonus, on va également obtenir une recherche efficace du minimum et du maximum d'un ensemble.

Remarque.

- Précisons tout de suite : les arbres binaires de recherche en eux-même ne sont pas efficaces. Pour garantir leur efficacité, il faut garantir qu'ils ne soient pas trop déséquilibrés : c'est à cela que serviront les arbres rouge-noir (ARN).
- Les ARN sont assez efficace pour être utilisés dans des sections critiques du noyau Linux ! L'ordonnanceur (le programme qui décide quelle tâche faire avancer en priorité) est codé à l'aide d'un ARN.

4.0 Définition, caractérisation

Pour les preuves, il sera pratique d'avoir une notation pour l'ensemble des étiquettes des noeuds d'un arbre.

Définition 38 (Ensemble d'étiquettes).

On définit l'ensemble $S(A)$ des étiquettes des noeuds d'un arbre binaire A par :

- $S(\perp) = \emptyset$
- $S(g, x, d) = \{x\} \cup S(g) \cup S(d)$

Définition 39 (Arbres binaires de recherche).

L'ensemble $ABR(\mathcal{E})$ des **arbres binaires de recherche** sur \mathcal{E} est défini inductivement par :

- $\perp \in ABR(\mathcal{E})$
- $N(g, x, d) \in ABR(\mathcal{E})$ lorsque :
 - $g \in ABR(\mathcal{E})$ et $d \in ABR(\mathcal{E})$
 - et $\max(S(g)) \leq x \leq \min(S(d))$

Remarque.

- Pour simplifier, dans la suite de ce cours on supposera les étiquettes deux à deux distinctes. En particulier, les inégalités larges seront strictes. De plus, cela permet de confondre sans ambiguïté un noeud et son étiquette, ce qui allège des notations.
- En maths, on pose $\max \emptyset = -\infty$ et $\min \emptyset = +\infty$. En informatique, dans les ABR on considèrera similairement que pour tout $x \in \mathcal{E}$, $\max \emptyset < x < \min \emptyset$.
- La définition d'un ABR n'est **pas** locale : il ne suffit pas que chaque noeud ait les bonnes comparaisons avec les racines de ses enfants !! La figure 9.34 ci-dessous donne un contre-exemple.

Exemple.

- (a) Un ABR pour {3; 5; 6; 7; 8; 9; 13; 20}
- (b) **Pas** un ABR, malgré le fait que chaque noeud est supérieur à la racine de gauche et inférieur à celle de droite

FIGURE 9.34 – Définition d'un arbre binaire de recherche

Théorème 40 (Caractérisation des ABR).

Soit A un arbre. On a :

A est un ABR si et seulement si son parcours infixe est trié.

Démonstration. On notera $\text{infixe}(A)$ la suite des noeuds visités par le parcours infixe d'un arbre A . Ainsi $\text{infixe}(\perp) = \emptyset$ et $\text{infixe}(g, x, d)$ est la concaténation de $\text{infixe}(g)$, x et $\text{infixe}(d)$; que l'on notera $\text{infixe}(g) @ x @ \text{infixe}(d)$ pour simplifier.

Montrons d'abord l'implication directe. On montre par induction structurelle sur A un ABR que $\text{infixe}(A)$ est trié.

- Initialisation : Si A est vide, c'est immédiat.
- Hérédité : Si $A = N(g, x, d)$ avec g et d vérifiant l'implication, montrons que $\text{infixe}(g, x, d)$ est trié. On a :

$$\text{infixe}(A) = \underbrace{\text{infixe}(g)}_{\text{trié par H.I.}} @ x @ \underbrace{\text{infixe}(d)}_{\text{trié par H.I.}}$$

Par H.I., les deux sous-parcours infixes sont triés par ordre croissant. Or, $\max S(g) < x < \min S(d)$; donc x est bien placé. Il s'ensuit que le tout est trié, d'où l'hérédité.

- Conclusion : Le parcours infixe d'un ABR est trié.

Montrons maintenant l'implication réciproque, par induction structurelle sur A un arbre binaire.

- Initialisation : Si A est vide, c'est immédiat.
- Hérédité : Si $A = N(g, x, d)$ est un arbre binaire avec g et d vérifiant l'implication réciproque, montrons que A est un ABR. Comme $\text{infixe}(A)$ est trié, ses sous-suites $\text{infixe}(g)$ et $\text{infixe}(d)$ le sont aussi : donc par H.I., g et d sont des ABR. Mais comme $\text{infixe}(A)$ est trié, $\max S(g) < x < \min S(d)$. Donc A est un ABR, d'où l'hérédité.
- Conclusion : Un arbre binaire trié par parcours infixe est un ABR.

□

Remarque. Une preuve de l'équivalence par induction structurelle aurait aussi été possible.

4.1 Opérations

4.1.0 Recherche

Les arbres binaires de recherche sont pensés pour faire des dichotomies dedans :

Exemple.

FIGURE 9.35 – Recherche dans un ABR

Cela donne le OCaml suivant

```

40  (** Renvoie [true] ssi [x] est présent dans [arbre] *)
41  let rec search x arbre =
42    match arbre with
43    | Nil -> false
44    | Node (g, etq, d) ->
45      if x = etq then true
46      else if x < etq then search x g
47      else search x d

```



Proposition 41 (Complexité de la recherche dans un ABR).

La recherche dans un ABR de hauteur h s'effectue en $\Theta(h)$ comparaisons.

Démonstration. La recherche se déplace dans un chemin de la racine à une feuille, en effectuant 2 comparaisons¹² par noeuds. D'où une complexité en $O(h)$ comparaisons.

Chercher x dans le peigne gauche ayant pour étiquettes $x + h \dots x + 1$ (de la racine à la feuille) prouve le besoin de $\Omega(h)$ comparaisons.

FIGURE 9.36 – Un pire cas de la recherche

□

12. En fait, en codant bien, c'est une seule : une fonction de comparaison totale (comme `Stdlib.compare` en OCaml) a trois issues possibles : « strictement inférieur », « égal » et « strictement supérieur ». On peut avec cela fusionner le test d'égalité et de strictement inférieur en un seul.

Remarque.

- On peut formaliser cette preuve ; mais elle est si simple que le gain n'est pas clair.
- Il est très important que l'ordre utilisé soit *total* : sinon, on ne sait pas dans quel sous-arbre aller !

4.1.1 Insertion

Pour insérer dans un ABR, on insère le nouvel élément au niveau des feuilles. Autrement dit, on recherche l'élément dans l'arbre : s'il est déjà présent, rien à faire¹³ ; sinon on l'insère à la place du \perp où la recherche termine.

Exemple.

FIGURE 9.37 – Insertions successives dans un ABR

Cela donne le code OCaml ci-dessous. On propose ici une implémentation fonctionnelle des ABR, donc l'insertion ne modifie pas l'arbre mais en renvoie un nouveau où l'insertion a eu lieu.

```

50 (** Renvoie l'abr obtenu en insérant [x] dans [arbre] *)
51 let rec insere x arbre =
52   match arbre with
53   | Nil ->
54     (* Créer la feuille contenant x *)
55     Node(Nil, x, Nil)
56   | Node (g, etq, d) ->
57     if x = etq then
58       (* Rien à faire, x est déjà là *)
59       arbre
60     else if x < etq then
61       (* Insérer à gauche *)
62       Node (insere x g, etq, d)
63     else
64       (* Insérer à droite *)
65       Node (g, etq, insere x d)

```



Proposition 42 (Complexité de l'insertion dans un ABR).

L'insertion dans un ABR de hauteur h s'effectue en $\Theta(h)$ comparaisons.

Démonstration. C'est la même que pour la recherche. □

13. Rappel : on s'est placé dans le cadre simplificateur des ABR avec éléments deux à deux distincts.

Proposition 43 (Variation de la hauteur lors de l'insertion dans un ABR).

Lors de l'insertion dans un ABR, la hauteur augmente au plus de 1.

Démonstration. L'insertion rajoute une feuille, qui rallonge donc un chemin de la racine aux feuilles de 1. Si ce chemin définissait la hauteur elle augmente de 1 ; sinon elle reste inchangée. \square

Remarque. On peut aussi prouver par induction structurelle sur A que $h(\text{insere}(x, A)) \leq h(A) + 1$

4.1.2 Suppression

Cette opération est plus délicate. On distingue quatre cas pour supprimer x de A :

- (i) Si $x \notin A$: rien à faire.
- (ii) Si x est une feuille : il suffit de la supprimer.
- (iii) Si x est un noeud interne avec un seul enfant : il suffit de le supprimer et de remonter son enfant à sa place.

FIGURE 9.38 – Suppression dans le cas où x a un seul enfant

- (iv) Sinon : x est un noeud interne avec deux enfants : c'est plus compliqué. On va se ramener au point précédent. Pour cela, on va écraser l'étiquette x avec l'étiquette du maximum de l'enfant gauche de x , puis aller ce dernier.

Proposition 44 (Maximum dans un ABR).

Le maximum d'un ABR A :

- n'a pas d'enfant droit.
- peut être trouvé en se déplaçant toujours à droite dans l'arbre.

Démonstration. • Considérons $N(g, m, d)$ le noeud qui contient le maximum. Si d est non-vide, tout noeud qu'il contient est strictement supérieur à m .

- Procédons par induction structurelle sur un abr A non-vide :
 - Si $A = N(g, x, \perp)$, par définition des ABR $\max S(g) < x$ donc x est le maximum de A .
 - Sinon, $A = N(g, x, d)$ avec $d \neq \perp$, par définition des ABR $\max S(g) < x < \min S(d)$. Il s'ensuit que $\max S(A) = \max S(d)$.

\square

Remarque.

- On a une propriété similaire pour le minimum.
- En corollaire, il est toujours simple de supprimer le maximum : on applique le cas

On peut en déduire une (mauvaise) façon de supprimer un noeud $N(g, x, d)$: le supprimer, remonter à sa place g et brancher d sous le maximum de g .

FIGURE 9.39 – Suppression dans un ABR, version peu efficace

Le problème de cette façon de faire est qu'elle déséquilibre fortement l'arbre, et risque de faire exploser la hauteur. Or, la hauteur est le paramètre déterminant dans la complexité des opérations ! À la place, nous allons plutôt procéder ainsi : remplacer x par le maximum du sous-arbre gauche, puis supprimer celui-ci.

FIGURE 9.40 – Suppression dans un ABR, version efficace !

Cela correspond au code OCaml ci-dessous, où `maximum` calcule le maximum d'un ABR :

```

85 (** Renvoie l'abr obtenu en supprimant [x] de [arbre] *)
86 let rec supprime x arbre =
87   match arbre with
88   | Nil -> Nil
89   | Node (g, etq, Nil) ->
90     if x = etq then g else Node (supprime x g, etq, Nil)
91   | Node (Nil, etq, d) ->
92     if x = etq then d else Node (Nil, etq, supprime x d)
93   | Node (g, etq, d) ->
94     if x = etq then
95       let maxi_g = maximum g in
96       Node (supprime maxi_g g, maxi_g, d)
97     else if x < etq then
98       Node (supprime x g, etq, d)
99     else
100      Node (g, etq, supprime x d)

```



Remarque. On pourrait rendre ce code plus efficace en faisant une fonction qui simultanément trouve et le maximum d'un ABR. Cela permettrait de parcourir une seule fois le sous-arbre gauche au lieu de deux dans les lignes 95-96.

Proposition 45 (Complexité de la suppression dans un ABR).

La suppression dans un ABR s'effectue en $\Theta(h)$ comparaisons.

Démonstration. Informellement, l'algorithme est en $O(h)$ car la suppression consiste en trois phases :

- Trouver x (complexité d'une recherche)
- Puis trouver le maximum de son sous-arbre gauche (linéaire en sa hauteur)
- Puis supprimer celui-ci (cas (iii) de la suppression, linéaire en la hauteur)

Pour prouver $\Omega(h)$, on considère exactement le même cas que pour la recherche ou l'insertion. \square

Exercice. Formaliser la preuve du $O(h)$. Pour cela, commencer par prouver que dans le cas (iii) de la suppression, la complexité est bel et bien linéaire en la hauteur. Ensuite, écrire l'équation de récurrence vérifiée par la complexité, et conclure (on peut admettre que la complexité est croissante en h pour simplifier).

4.1.3 Rotation

Les rotations gauches et droites sont des transformations d'un arbre binaire qui sont parfois utilisés dans certains algorithmes sur les ABR (notamment dans les ARN). J'ai réussi à écrire ce cours sans les utiliser, mais il est bon de savoir ce dont il s'agit :

Définition 46 (Rotation gauche et droite).

Les rotations gauche et droite d'un arbre sont les transformations suivantes :

FIGURE 9.41 – Rotation gauche et droite

Remarque.

- La rotation gauche ne peut pas s'appliquer à n'importe quel arbre (il faut que le sous-arbre gauche ait la bonne forme), et de même pour la rotation droite.
- La rotation est définie sur tout arbre binaire, pas uniquement sur les ABR.

Proposition 47.

Une rotation s'effectue en temps constant.

Démonstration. Implémentation-dépendant. \square

Proposition 48.

A est un ABR si et seulement si sa rotation (gauche ou droite) est un ABR.

Démonstration. Assez agréable est d'utiliser le parcours infixe. Laissé en exercice. \square

4.1.4 Tri par ABR

On déduit de ce qui précède un algorithme pour trier un tableau ou une liste via un ABR :

Algorithme 14 : Tri par ABR

Entrées : x_0, \dots, x_{n-1} à trier

Sorties : Une version triée de l'entrée

```

1  $A \leftarrow \perp$ 
2 pour chaque  $x_i$  faire
3    $A \leftarrow \text{INSERE}(x_i, A)$ 
4 renvoyer  $\text{infixe}(A)$ 
```

Proposition 49 (Complexité du tri par ABR).

En nombre de comparaisons, le tri par ABR a pour complexité :

- $\Theta(n^2)$ dans le pire des cas.
- $O(n \log_2 n)$ dans le cas moyen

Démonstration. Le parcours infixe d'un arbre à n peut-être écrit en complexité $\Theta(n)$ (cf question bonus TP). La question est donc la complexité de la boucle des lignes 2-3.

- Dans le pire des cas : chaque insertion coûte $O(h(A))$. Or, la hauteur augmente au plus de 1 à chaque insertion, donc la boucle coûte $O\left(\sum_{i=0}^{n-1} (-1 + i)\right) = O(n^2)$. De plus, dans le cas où les x_i sont déjà triés par ordre croissant ou décroissant, la hauteur augmente d'exactly 1 à chaque insertion : cette borne est atteinte, d'où le $\Theta(n^2)$.
- La complexité moyenne est admise. Rappelons simplement que la complexité moyenne pour n est la moyenne sur tous les x_0, \dots, x_{n-1} des complexités. Ici, comme la complexité ne dépend pas des valeurs précises x_i mais des comparaisons entre eux, on moyenne sur les permutations de \mathcal{S}_n .

□

Ce pire des cas fournit un bon exemple de la limitation de nos ABR actuels : la complexité des opérations dépend de la *hauteur*, or rien ne garantit que celle-ci soit faible. On voudrait garantir que les ABR ne soient pas « trop » déséquilibrés pour que la hauteur reste logarithmique.

4.2 Arbres Rouge-Noir

Les arbres Rouge-Noir (abrégés ARN), aussi appelés arbres bicolores, sont une façon de garantir l'équilibrage d'un arbre binaire de recherche.

4.2.0 Définition

Définition 50 (Arbre Rouge-Noir).

Un **arbre Rouge-Noir** est un arbre binaire de recherche où chaque noeud est colorié, en Rouge ou bien en Noir. On impose de plus que :

- (1) Tout noeud Rouge a un parent Noir.
- (2) Tous les chemins de la racine à un \perp contiennent autant de noeuds noirs.

Remarque.

- La propriété (1) équivaut à « Un noeud rouge n'a que des enfants noirs, et la racine est noire ».
- Rien n'interdit d'avoir plusieurs noeuds noirs à la suite !

Exemple.

(a) Un arbre Rouge-Noir à 8 noeuds

(b) Un arbre Rouge-Noir à 12 noeuds

FIGURE 9.42 – Deux arbres Rouge-Noir

Proposition 51.

Soit A est un arbre binaire colorié (qu'il soit ou non un ARN). S'équivalent :

- A vérifie le point (2) de la définition des ARN.
- A vérifie (2') : « Pour tout noeud x d'un arbre, tous les chemins de ce noeud x à un \perp contiennent autant de noeuds noirs ».

Démonstration. Procédons par double implication.

\Leftarrow) L'implication réciproque est immédiate puisque la racine est un noeud.

\Rightarrow) Pour l'implication directe, procédons par induction structurelle sur un arbre bicolorié A :

- Initialisation : Si $A = \perp$, c'est immédiat.
- Hérédité : Soit $A = N(g, x, d)$ tel que g et d vérifient l'implication, et montrons-la pour A . Supposons donc que A vérifie (2), et montrons qu'il vérifie (2'). Notons que (2') est immédiatement vrai pour la racine d'après (2). Pour les autres noeuds, remarquons que les chemins dans A de la racine à un \perp sont de la forme $x, \text{racine}(g), \dots, \perp$ ou $x, \text{racine}(d), \dots, \perp$

FIGURE 9.43 – Les chemins de la racine aux feuilles

D'après (2), ces chemins contiennent tous autant de noeuds noirs. En particulier, tous les chemins $racine(g), \dots, \perp$ et $racine(d), \dots, \perp$ ont le même nombre de noeuds noirs : en appliquant l'hypothèse d'induction à g et d , on en déduit que (2') est valide sur tous les noeuds de g et d . D'où l'hérédité.

- Conclusion : On a prouvé par induction structurelle que (2) \implies (2').

□

Définition 52 (Hauteur noire).

Dans un ARN, la **hauteur noire** d'un noeud x , notée $bh(x)$ est le nombre de noeuds noirs sur un chemin de x à un \perp .

Démonstration. Pour que la hauteur noire soit bien définie, il faut qu'elle soit indépendante du chemin (la définition n'impose aucun chemin particulier). C'est le cas d'après (2'). □

Remarque. Contrairement à la hauteur, la hauteur noire est un nombre de noeuds et non d'arêtes !

Lemme 53.

Soit A un ARN, A_x un sous-arbre non-vide de A dont on note x la racine. Alors A_x a au moins $2^{bh(x)} - 1$ noeuds internes.

Démonstration. Procédons par récurrence sur la hauteur $h(x)$ de x (et non sur la hauteur noire !!) :

- Initialisation : $h(x) = -1$ est impossible car A_x est non-vide, donc prenons $h(x) = 0$ comme cas de base. Alors A_x est juste une feuille donc a 0 noeuds internes, d'où l'initialisation.
- Hérédité : Si $h(x) > 0$ (et que la propriété est vraie pour tout $h' < h(x)$), alors A_x est de la forme $A_x = N(g, x, d)$ avec g et d non-vides.

Par définition de bh , on a $bh(g) \in \{bh(x) - 1, bh(x)\}$ et de même pour $bh(d)$. Par H.R., on sait que g a au moins $2^{bh(g)} - 1$ noeuds internes, et de même pour d . Donc en notant n_i le nombre de noeuds internes de A_x :

$$\begin{aligned} n_i &\geq (2^{bh(g)} - 1) + (2^{bh(d)} - 1) + 1 \\ &\geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 && \geq 2^{bh(x)} - 1 \end{aligned}$$

□

Théorème 54 (Un ARN est à peu près équilibré).

Soit A un arbre Rouge-Noir à n noeuds. Alors :

$$h(A) = O(\log_2 n)$$

Plus précisément, avec n_i le nombre de noeuds internes :

$$h(A) \leq 2 \log_2(n_i + 1)$$

Démonstration. La majoration de déduit de l'inégalité puisque $n \geq n_i + 1$ et que \log est croissant. Montrons donc l'inégalité.

D'après la propriété (1) des ARN, dans tout chemin de la racine à \perp au moins la moitié des noeuds sont noirs (un rouge ne peut venir qu'après un noir donc il y a au moins autant de noirs que de rouges). Donc :

$$bh(\text{racine}(A)) \geq \frac{h}{2}$$

Or d'après le lemme, $2^{bh(\text{racine}(A))} - 1 \leq n_i$, donc $2^{\frac{h}{2}} \leq n_i + 1$. On en déduit le résultat par croissance de \log_2 . \square

On vient de prouver qu'un ABR est à peu près équilibré ! Une autre façon de comprendre la preuve de ce théorème est la suivante : un chemin de la racine à \perp contenant bh noeuds noirs a au moins bh noeuds (s'ils sont tous noirs), et au plus $2bh + 1$ (alternance noir-rouge). Ainsi, il y a au plus un facteur 2 d'écart sur la longueur des différents chemins de la racine à un \perp , et l'arbre ne peut donc pas être trop déséquilibré !

FIGURE 9.44 – Interprétation de la preuve du théorème en termes de hauteur vs hauteur noire

Remarque. D'après la propriété 10, $h = \Omega(\log_2 n)$. Donc l'ordre de grandeur atteint par les ARN est optimal !

Proposition 55 (Complexité de la recherche dans un ARN).

La recherche dans un ARN s'effectue en $\Theta(\log_2 n)$ comparaisons.

Démonstration. Il suffit d'appliquer le même algorithme que dans un ABR, puisqu'un ARN est un ABR. Le théorème 54 conclut. \square

4.2.1 Insertion

Insérer dans un ABR est facile... mais dans un ARN, on doit garantir que les propriétés (1) et (2) restent respectées !

On va procéder ainsi :

- On insère comme dans un ABR, en coloriant la feuille créée en Rouge.
- Si cette insertion a rompu la règle (1) (tout Rouge a un parent Noir), on « fait remonter » l'erreur.
- Lorsque la violation de (1) est à la racine, pour la réparer il suffit de colorier la racine en Noir.

En particulier, (2) est toujours vérifiée et il n'y a pas à la réparer ! De plus, en faisant « remonter » la violation de (1), on garantit qu'il y a au plus une violation (1).

Voyons maintenant comment faire cette remontée : on considère donc l'unique violation de (1), qui n'a pas lieu à la racine. La violation est donc un noeud Rouge qui a un parent Rouge. Comme il y a au plus une violation de (1), le grand-parent est Noir. En nommant $x \leq y \leq z$ les 3 noeuds concernés, cela donne 4 cas qui peuvent tous être résolus de la même façon :

(a) Les 4 enchainements Noir-Rouge-Rouge possibles

(b) Une façon unique de tous les réparer

Cette façon de réparer a recolorié la racine du sous-arbre en rouge (alors qu'elle était noire). Si son parent était rouge, il y a toujours une violation de (1), mais moins profonde ! Ainsi, on fait « remonter » la violation jusqu'à la racine, où il suffit de recolorier la racine en Noir pour terminer.

Exercice. Vérifier que cette façon de réparer ne rompt pas (2), c'est à dire qu'elle ne casse pas la hauteur noire.

Exemple.

FIGURE 9.46 – Des insertions successives dans un ARN

Proposition 56 (Complexité de l'insertion dans un ARN).

L'insertion dans un ARN s'effectue en temps $\Theta(\log_2 n)$.

Démonstration. Sans code sous les yeux, je me contenterais d'une preuve très schématique. L'insertion est composée de deux phases :

- Insertion comme dans un ABR, en $O(h)$
- Puis remontée de la violation jusqu'à la racine. Une étape de la remontée se fait en temps constant (c'est un nombre fini de rotation), et on remonte vers la racine donc on fait au plus autant d'étapes que la hauteur.

Comme la hauteur est logarithmique en n , on conclut. \square

Remarque. En particulier, avec des ARN, le tri par ABR/ARN est en temps $O(n \log n)$!

4.2.2 Suppression

La suppression dans un ARN est plus compliquée.

Proposition 57 (Complexité de la suppression dans un ARN).

La suppression dans un ARN s'effectue en temps $\Theta(\log_2 n)$.

Démonstration. Admis. \square

La suppression fonctionne ainsi :

- Supprimer comme dans un ABR, en supprimant le maximum d'un sous-arbre gauche.
- Si le noeud supprimé était rouge, tout va bien.
- Sinon, on « ajoute » la couleur noir du noeud supprimé à celle de son parent. Cela crée éventuellement un noeud « doublé noir », qu'il faut ensuite faire remonter jusqu'à la racine.

4.3 Pour aller plus loin

Si vous voulez en savoir plus, vous pouvez aller voir :

- À propos de la suppression dans un ARN : <https://www.irif.fr/~carton/Enseignement/Algorithmique/Programmation/RedBlackTree/> . La définition des ARN qui y est utilisée est légèrement différente, mais cela ne change pas fondamentalement l'algo. Vous trouverez également sur cette page une façon un peu plus efficace de coder l'insertion (car elle arrive parfois à réparer le Rouge-Rouge localement, sans le faire remonter jusqu'à la racine).
- Un résumé très clair des grandes idées de ce cours¹⁴ : deuxième leçon de <https://www.college-de-france.fr/fr/agenda/cours/structures-de-donnees-persistantes> . Les ARN (et leur suppression) y sont traités, mais vous y trouverez aussi une autre façon d'équilibrer les ABR : les AVL !

14. Leçon de Xavier Leroy, le fondateur d'OCaml!