Structures de données arborescentes

1 Introduction et motivations

1.1 Un arbre pour un objet

Pour les éléments d'un ensemble construit par induction, il est approprié d'utiliser une représentation par arbres puisque ces objets ont intrinsèquement une structure arborescente.

Exemples : On peut par exemple citer :
- les expressions booléennes, qui sont construites par induction à partir des constructeurs :

$$\texttt{true}|^0_{\{\cdot\}},\ \texttt{false}|^0_{\{\cdot\}},\ \text{var}|^0_{\Sigma^*},\ \texttt{not}|^1_{\{\cdot\}},\ \texttt{\&\&}|^2_{\{\cdot\}},\ \texttt{||}|^2_{\{\cdot\}}$$

- les expressions arithmétiques en OCaml (*cf.* DM n°2, partie 2 – "expressions arithmétiques")
- les expressions de type en OCaml, qui peuvent être simples ou composées comme l'illustrent les deux arbres ci-dessous :

1.2 Un arbre pour une collection d'objets

On cherche à stocker une collection d'objets sans multiplicité et dont l'ordre relatif n'est pas significatif (typiquement à la manière d'un ensemble au sens mathématique). On suppose que les éléments sont tous de même type `elem` et qu'ils sont identifiés de manière unique par une clé (c'est-à-dire une sous-partie de l'élément permettent l'identification).

Remarque : Un élément de type `elem` est donc supposé contenir lui-même sa clé : celle-ci peut être vue comme un diminutif de l'élément, et on la déduit facilement si l'on a l'élément.

1.2.1 La structure d'ensemble

On peut s'intéresser par exemple à la structures de données abstraite d'ensemble :

ensemble	• creer_ens_vide : () → ens • est_ens_vide : ens → bool • appartient : ens × cle → bool • trouve_elem : ens × cle → elem • ajoute_elem : ens × elem → ens / () • supprime_elem : ens × cle → ens / ()

En-voici quelques implémentations possibles, avec leurs complexités pour les fonctions élémentaires.

∗ **Implémentation par liste**
- recherche : $\Theta(n)$
- appartenance : $\Theta(n)$
- suppression : $\Theta(n)$ ($\Theta(n)$ pour trouver l'élément et $\Theta(1)$ pour le supprimer)
- ajout : $\Theta(1)$

∗ **Implémentation par tableau trié** – si l'ensemble des valeurs de type `cle` est ordonné
- recherche : $\Theta(\log(n))$
- appartenance : $\Theta(\log(n))$
- suppression : $\Theta(n)$

- ajout : $\Theta(n)$

∗ **Implémentation par arbres binaires de recherche (ABR)** – *cf.* plus loin dans ce chapitre
- recherche : $\Theta(\log(n))$ au mieux$^{(*)}$
- appartenance : $\Theta(\log(n))$ au mieux$^{(*)}$
- suppression : $\Theta(\log(n))$
- ajout : $\Theta(\log(n))$

Remarque : L'opération de recherche et le test appartenance dans un ABR est en $\Theta(\log(n))^{(*)}$ si l'arbre est globalement complet, *i.e.* son hauteur est logarithmique par rapport à son nombre de nœuds (*cf.* TD n°9).

Après comparaison de ces différentes complexités, on observe que selon le contexte, une certaine implémentation sera plus efficace ou adaptée qu'une autre. Plus précisément :
- l'implémentation par liste est mieux adaptée pour un ensemble dans lequel on ne cherche pas à effectuer de recherche ou de vérification d'appartenance.
- celle par tableau trié est adaptée aux ensembles stables, c'est-à-dire peu sujets à modification

En revanche, les arbres semblent, dans les bonnes conditions, permettre de rassembler les deux avantages précédents, ce qui justifie que nous nous y intéresserons dans la partie suivante.

1.2.2 La structure de tas

On peut également se pencher sur la structure de données abstraite de tas, qui n'est qu'un cas particulier de la structure de file de priorité. Les éléments sont alors munis d'une priorité.

tas	
	• creer_tas_vide : () → `tas`
	• est_tas_vide : `tas` → `bool`
	• min : `tas` → `elem`
	• ajout : `tas` × `elem` → `tas` / ()
	• supprime_min : `tas` → `tas` / ()

Remarque : Ce qu'on appelle minimum d'un tas n'est pas l'élément de clé minimum, mais celui de priorité minimum.

2 Arbres binaires

2.1 Définition de l'ensemble $\mathscr{A}_B$

Définition (*ensemble des arbres étiquetés par un ensmeble*) :

Soit $\mathcal{S}$ un ensemble. On définit l'ensemble des arbres binaires étiquetés par $\mathcal{S}$, noté $\mathscr{A}_B(\mathcal{S})$, comme l'ensemble construit par induction à partir des règles de construction :
- $\mathrm{V}|^0_{\{\_\}}$ (arbre vide)
- $N|^2_{\mathcal{S}}$ (nœud binaire)

Exemple : Pour $\mathcal{S} = \{1, 2, 3, 4\}$, les arbres suivants sont des éléments de $\mathscr{A}_B(\mathcal{S})$:
$t_0 = (\mathrm{V}, \_)$
$t_1 = (N, 1, (\mathrm{V}, \_), (\mathrm{V}, \_))$
$t_2 = \Big(N, 1, \big(N, 2, (\mathrm{V}, \_), (\mathrm{V}, \_)\big), \big(N, 3, (N, 4, (\mathrm{V}, \_), (\mathrm{V}, \_)), (\mathrm{V}, \_)\big)\Big)$.

Remarque : En français, $\mathscr{A}_B(\mathcal{S})$ est le plus petit ensemble qui contient l'arbre vide V, ainsi que l'arbre $N(x, g, d)$ pour tout $x \in \mathcal{S}$ et $(g, d) \in \mathscr{A}_B(\mathcal{S})^2$.

2.2 Feuilles

On fixe $\mathcal{S}$ un ensemble d'étiquettes pour tout le reste de cette partie.

Définition (*arbre réduit à une feuille*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})$. On dit que t est réduit à une feuille s'il existe $x \in \mathcal{S}$ tel que $t = N(x, \mathrm{V}, \mathrm{V})$.

Propriétés (*éléments minimaux de $\mathscr{A}_B(\mathcal{S})$ et $\mathscr{A}_B(\mathcal{S})\backslash\{\mathrm{V}\}$*) :

Notons $\leqslant$ la relation d'ordre sur $\mathscr{A}_B(\mathcal{S})$ associée à sa définition inductive. Alors :

i. V est le seul élément minimal de $(\mathscr{A}_B(\mathcal{S}), \leqslant)$
ii. les éléments minimaux de $(\mathscr{A}_B(\mathcal{S})\backslash\{V\}, \leqslant)$ sont des feuilles
iii. réciproquement, toutes les feuilles sont des éléments minimaux de $(\mathscr{A}_B(\mathcal{S})\backslash\{V\}, \leqslant)$.

Preuve :

ii. Soit $t \in \mathscr{A}_B(\mathcal{S})\backslash\{V\}$. Il existe $x \in \mathcal{S}$, $(g, d) \in \mathscr{A}_B(\mathcal{S})^2$ tels que $t = N(x, g, d)$.
On suppose que t est minimal dans $\mathscr{A}_B(\mathcal{S})\backslash\{V\}$. Si on note $\mathcal{R}$ la relation dont $\leqslant$ est la clôture réflexive transitive (*cf.* chapitre 8 – "ordre et induction"), alors on a :

$$\begin{cases} g \,\mathcal{R}\, t \\ d \,\mathcal{R}\, t \end{cases} \text{ donc } \begin{cases} g \leqslant t \\ d \leqslant t \end{cases}$$

De plus, $g, d \neq t$ donc on a $g < t$ et $d < t$. Par minimalité de t dans $\mathscr{A}_B(\mathcal{S})\backslash\{V\}$, on en déduit que $g = \mathrm{V}$ et $d = \mathrm{V}$, donc $t = N(x, \mathrm{V}, \mathrm{V})$ est bien une feuille.

iii.

Corollaire :

On peut définir une fonction récursive à partir des éléments minimaux.

2.3 Chemins dans un arbre binaire

On pose $\Sigma = \{0, 1\}$ et on note $\cdot$ la concaténation sur $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$.

Définition (*ensemble des chemins d'un arbre*) :

On définit par induction sur $\mathscr{A}_B(\mathcal{S})$ l'ensemble des chemins admissibles d'un arbre binaire :

$$\mathrm{ch} = \begin{pmatrix} \mathscr{A}_B(\mathcal{S}) \to \mathcal{P}_f(\Sigma^*) \\ t \mapsto \begin{cases} \emptyset \text{ si } t = \mathrm{V} \\ \{\varepsilon\} \cup 0 \cdot \mathrm{ch}(g) \cup 1 \cdot \mathrm{ch}(d) \text{ si } t = N(x, g, d) \end{cases} \end{pmatrix}$$

où $\mathcal{P}_f(\Sigma^*$ est l'ensemble des parties finies de Σ^*.

Un chemin admissible décrit ainsi la "position" d'un nœud dans l'arbre.

Définition (*nœuds formels et profondeur*) :

Soit $A \in \mathscr{A}_B(\mathcal{S})$. On appelle nœud de A un élément n de $\mathrm{ch}(A)$.
Sa profondeur est alors sa longueur en tant que mot de Σ^*, c'est-à-dire $|n|$.

Définition (*taille d'un arbre*) :

Soit $A \in \mathscr{A}_B(\mathcal{S})$. La taille de A, notée $s(A)$, est le nombre de nœuds de A :

$$s(A) = \mathrm{Card}(\mathrm{ch}(A))$$

Exercice : Donner une définition inductive de s.

Correction : En s'aidant de la définition de ch, on a facilement :

$$s = \begin{pmatrix} \mathscr{A}_B(\mathcal{S}) \to \mathbb{N} \\ t \mapsto \begin{cases} 0 \text{ si } t = \mathrm{V} \\ 1 + s(g) + s(d) \text{ si } t = N(x,g,d) \end{cases} \end{pmatrix}$$

2.4 Étiquetage

Définition (*étiquetage d'un arbre*) :

On appelle étiquetage d'un arbre non vide la fonction qui à un nœud de l'arbre associe son étiquette. Formellement, cet étiquetage est défini inductivement comme suit :

$$\mathrm{et} = \begin{pmatrix} \mathscr{A}_B(\mathcal{S})\backslash\{\mathrm{V}\} \to \mathcal{F}_p(\Sigma^*, \mathcal{S}) \\ N(x,\mathrm{V},\mathrm{V}) \mapsto \begin{pmatrix} \{\varepsilon\} \to \mathcal{S} \\ \varepsilon \mapsto x \end{pmatrix} \\ N(x,g,\mathrm{V}) \mapsto \begin{pmatrix} \mathrm{ch}(N(x,g,\mathrm{V})) \to \mathcal{S} \\ \varepsilon \mapsto x \\ 0 \cdot u \mapsto \mathrm{et}(g)(u) \end{pmatrix} \\ N(x,\mathrm{V},d) \mapsto \begin{pmatrix} \mathrm{ch}(N(x,\mathrm{V},d)) \to \mathcal{S} \\ \varepsilon \mapsto x \\ 1 \cdot u \mapsto \mathrm{et}(d)(u) \end{pmatrix} \\ N(x,g,d) \mapsto \begin{pmatrix} \mathrm{ch}(N(x,g,d)) \to \mathcal{S} \\ \varepsilon \mapsto x \\ 0 \cdot u \mapsto \mathrm{et}(g)(u) \\ 1 \cdot u \mapsto \mathrm{et}(d)(u) \end{pmatrix} \end{pmatrix}$$

$\mathcal{F}_p(\Sigma^*, \mathcal{S})$ étant l'ensemble des fonctions partiellement définies sur Σ^* et à valeurs dans $\mathcal{S}$.

Pour $A \in \mathscr{A}_B(\mathcal{S})\backslash\{\mathrm{V}\}$, $\mathrm{et}(A)$ est appelée l'étiquetage de A.

Définition (*étiquette d'un nœud dans un arbre*) :

Soit $A \in \mathscr{A}_B(\mathcal{S})\backslash\{\mathrm{V}\}$. Pour $n \in \mathrm{ch}(A)$, l'étiquette de n dans A est $(\mathrm{et}(A))(n)$.

2.5 Vocabulaire

Définition (*racine, feuille, nœud interne*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})$ et $n \in \mathrm{ch}(t)$. On dit que :

- n est racine de t ssi $n = \varepsilon$
- n est une feuille de t ssi $\forall\, u \in \Sigma^*,\ n \cdot u \in \mathrm{ch}(t) \Longrightarrow u = \varepsilon$, autrement dit il n'existe pas de prolongement strict de n dans $\mathrm{ch}(t)$
- n est un nœud interne de t ssi $\exists\, u \in \Sigma^*\backslash\{\varepsilon\},\ n \cdot u \in \mathrm{ch}(t)$, *i.e.* n n'est pas une feuille.

Remarque : En fonction de la taille de l'arbre, la racine est tantôt une feuille, tantôt un

nœud interne.

Définition (*père, fils, descendant*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})\backslash\{\mathrm{V}\}$ et $(n, n') \in \mathrm{ch}(t)^2$. Alors :
- n' est le fils gauche ou l'enfant gauche de n ssi $n' = n \cdot 0$
- n' est le fils droit ou enfant droit de n ssi $n' = n \cdot 1$
- n' est un fils de n ssi c'est un fils gauche ou un fils droit, n est alors le père de n'
- n' est un descendant de n ssi il existe $u \in \Sigma^*$ tel que $n' = n \cdot u$.

Remarque : La racine est le seul nœud sans père, et les feuilles sont les seuls sans fils.

Définition (*branche*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})\backslash\{V\}$ et $(n_i)_{i\in[0..k]} \in \mathrm{ch}(t)^{k+1}$. On dit que $(n_i)_{i\in[0..k]}$ est une branche de t ssi :

$$\begin{cases} n_0 = \varepsilon \\ \forall\, i \in [0..k[,\ n_i \text{ est le père de } n_{i+1} \end{cases}$$

Définition (*sous-arbres*) :

Soit $(t, t') \in \mathscr{A}_B(\mathcal{S})^2$.
- t' est sous-arbre gauche de t ssi il existe $d \in \mathscr{A}_B(\mathcal{S})$ et $x \in \mathcal{S}$ tels que $t = N(x, t', d)$
- t' est sous-arbre droit de t ssi il existe $g \in \mathscr{A}_B(\mathcal{S})$ et $x \in \mathcal{S}$ tels que $t = N(x, g, t')$
- t' est un sous-arbre de t ssi $t' \leqslant t$, $\leqslant$ étant la relation d'ordre induite par la construction de l'ensemble par induction.

Remarque : Comme $\leqslant$ est une relation d'ordre, donc en particulier transitive, t' peut être sous-arbre de t sans qu'il soit en nécessairement sous-arbre gauche ni sous-arbre droit.

2.6 Hauteur

Définition (*hauteur d'un arbre*) :

On définit la hauteur d'un arbre binaire étiqueté par $\mathcal{S}$ par la fonction :

$$h = \left(\begin{array}{rl} \mathscr{A}_B(\mathcal{S}) \to & \mathbb{N} \cup \{-1\} \\ t \mapsto & \begin{cases} -1 \text{ si } t = \mathrm{V} \\ 1 + \max(h(g), h(d)) \text{ si } t = N(x, g, d) \end{cases} \end{array} \right)$$

Propriétés (*caractérisations de la hauteur*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})$. Alors, on a :

i. $h(t) = \max\limits_{n\in\mathrm{ch}(t)} \mathrm{prof}(n)$ si $t \neq \mathrm{V}$ (plus généralement, $h(t) = \max\left(\max\limits_{n\in\mathrm{ch}(t)} \mathrm{prof}(n), -1\right)$)

ii. $h(t)$ est la longueur maximale d'une branche de t, moins 1.

Remarque : Une branche atteingant un tel maximum débute nécessairement à la racine.

Propriété : Pour tout $t \in \mathscr{A}_B(\mathcal{S})$, on a $h(t) + 1 \leq s(t) \leq 2^{h(t)+1} - 1$.

Preuve :
Montrons la première inégalité.

2.7 Parcours

Définition (*parcours dans un arbre*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})\backslash\{\mathrm{V}\}$ et $a = (a_i)_{i\in[1..s(t)]} \in \mathcal{S}^{s(t)}$.
On dit que a est un parcours de t ssi il existe $\varphi \in \mathcal{F}(\mathrm{ch}(t), [1..s(t)])$ bijective telle que :

$$\forall\, n \in \mathrm{ch}(t),\ a_{\varphi(n)} = \mathrm{et}(t)(n) \ \text{ soit encore } \ \forall\, i \in [1..s(t)],\ a_i = \mathrm{et}(t)(\varphi^{-1}(i))$$

Définition (*parcours préfixe, infixe, post-fixe*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})\backslash\{V\}$. Pour $n \in \mathrm{ch}(t)$, on note :

- $\mathcal{G}(n) = \{n \cdot 0 \cdot u \,|\, u \in \Sigma^*\} \cap \mathrm{ch}(t)$
- $\mathcal{D}(n) = \{n \cdot 1 \cdot u \,|\, u \in \Sigma^*\} \cap \mathrm{ch}(t)$

Soit $\varphi \in \mathscr{B}ij(\mathrm{ch}(t), [1..s(t)])$. Alors :

- $(\mathrm{et}(t)(\varphi^{-1}(i)))_{i\in[1..s(t)]}$ est un parcours préfixe de t si
$$\forall\, n \in \mathrm{ch}(t),\ \varphi(n) \leq \min_{g\in\mathcal{G}(n)} \varphi(g) \text{ et } \max_{g\in\mathcal{G}(n)} \varphi(g) \leq \min_{d\in\mathcal{D}(n)} \varphi(d)$$
- $(\mathrm{et}(t)(\varphi^{-1}(i)))_{i\in[1..s(t)]}$ est un parcours infixe de t si
$$\forall\, n \in \mathrm{ch}(t),\ \max_{g\in\mathcal{G}(n)} \varphi(g) \leq \varphi(n) \leq \min_{d\in\mathcal{D}(n)} \varphi(d)$$
- $(\mathrm{et}(t)(\varphi^{-1}(i)))_{i\in[1..s(t)]}$ est un parcours post-fixe de t si
$$\max_{g\in\mathcal{G}(n)} \varphi(g) \leq \min_{d\in\mathcal{D}(n)} \varphi(d) \text{ et } \min_{g\in\mathcal{G}(n)} \varphi(g) \leq \varphi(n)$$

Remarque : De façon alternative, si $\forall\, n \in \mathrm{ch}(t), \forall\,(g,d) \in \mathcal{G}(n)\times\mathcal{D}(n), \varphi(n) \leq \varphi(g) \leq \varphi(d)$, alors $(\mathrm{et}(t)(\varphi^{-1}(i)))_{i\in[1..s(t)]}$ est un parcours préfixe de t (*idem* pour les autres parcours).

Propriété (*unicité des parcours préfixe, infixe, post-fixe*) :

Il y a unicité des parcours préfixe, infixe et post-fixe dans un arbre non vide.

Exemple :

Définition (*parcours en longueur*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})$ et $\varphi \in \mathscr{B}ij(\mathrm{ch}(t), [1..s(t)])$.
On dit que $(\mathrm{et}(t)(\varphi^{-1}(i)))_{i\in[1..s(t)]}$ est un parcours en longueur de t si :

$$\forall\,(n,n') \in \mathrm{ch}(t)^2,\ \mathrm{prof}(n) \leq \mathrm{prof}(n') \Longrightarrow \varphi(n) \leq \varphi(n')$$

Remarque : Il existe aussi une notion de parcours en profondeur, qui est hors-programme.

3 Arbres de recherche

Dans cette partie, $\mathcal{E}$ désignera un ensemble muni d'une relation d'ordre totale notée $\preccurlyeq$.

Définition (*arbre binaire de recherche*) :

Soit $t \in \mathscr{A}_B(\mathcal{E})\backslash\{\mathrm{V}\}$. On note e pour désigner la fonction $\mathrm{et}(t) \in \mathcal{F}(\mathrm{ch}(t), \mathcal{E})$.
t est un arbre binaire de recherche, ou ABR, si et seulement si

$$\forall\, n \in \mathrm{ch}(t),\ \max_{g \in \mathcal{G}(n)} e(g) \preccurlyeq e(n) \prec \min_{d \in \mathcal{D}(n)} e(d)$$

où $\mathcal{G}(n)$ et $\mathcal{D}(n)$ sont définis comme précédemment.

De plus, on convient que V est un ABR.

Remarque : Le parcours infixe d'un ABR donne une suite croissante d'éléments de $(\mathcal{E}, \preccurlyeq)$.

3.1 Recherche d'élément

Profitant de la structure ordonnée de l'ABR, on procède par dichotomie pour la recherche d'un élément (de son étiquette en réalité). On propose le pseudo-code suivant :

Pseudo-code – est_présent (ABR t, $e \in \mathcal{E}$) $\to$ `bool`

Si $t = \mathrm{V}$ alors renvoyer Faux
Si $t = N(x, g, d)$ alors
    Si $e = x$ alors Vrai
    Si $e \prec x$ alors est_présent(g, e)
    Sinon est_présent(d, e)

Propriété :

Pour $t \in \mathscr{A}_B(\mathcal{E})$ et $e \in \mathcal{E}$, on note $c(t, e)$ le nombre de comparaisons effectuées lors de l'appel est_présent(t, e). Alors :

i. $\forall\, t \in \mathscr{A}_B(\mathcal{E})$, $\forall\, e \in \mathcal{E}$, on a $c(t, e) \leq 2(h(t) + 1)$
ii. $\forall\, t \in \mathscr{A}_B(\mathcal{E})$, $\forall\, n \in \mathrm{ch}(t)$, on a $\mathrm{uni}(n, t) \implies c(t, \mathrm{et}(t)(n)) = 2\,\mathrm{prof}(n) + 1$

où $\mathrm{uni}(n, t)$ désigne la proposition $\forall\, n' \in \mathrm{ch}(t)$, $\mathrm{et}(t)(n') = \mathrm{et}(t)(n) \implies n' = n$ (ce qui revient à dire que n est le seul nœud de t portant son étiquette).

Preuve :
i.

Corollaire :

Soit $t \in \mathscr{A}_B(\mathcal{E})\backslash\{\mathrm{V}\}$. Il existe $e \in \mathcal{E}$ tel que $c(t, e) = 2h(t) + 1$.

Preuve :
D'après une caractérisation de la hauteur, pour $t \in \mathscr{A}_B(\mathcal{E})\backslash\{\mathrm{V}\}$ on a $h(t) = \max_{n \in \mathrm{ch}(t)} \mathrm{prof}(n)$.
Soit $n^* \in \mathrm{ch}(t)$ tel que $\mathrm{prof}(n^*) = \mathrm{ch}(t)$. D'après la propriété précédente :

$$c(t, \mathrm{et}(t)(n^*)) = 2\,\mathrm{prof}(n^*) + 1 = 2h(t) + 1$$

donc $e = \mathrm{et}(t)(n^*)$ convient.

3.2 Ajout en feuille

Voici le pseudo-code d'une fonction permettant d'ajouter un élément en feuille dans un ABR.

Pseudo-code – ajout (ABR t, $e \in \mathcal{E}$) $\to$ ABR

Si $t = \mathrm{V}$ alors $N(e, \mathrm{V}, \mathrm{V})$
Si $t = N(x, g, d)$ alors
    Si $e \preccurlyeq x$ alors
        $N(x, \mathsf{ajout}(g, e), d)$
    Sinon
        $N(x, g, \mathsf{ajout}(d, e))$

Propriété (*conservation de la structure d'ABR*) :

Si t est un ABR et $e \in \mathcal{E}$, alors $\mathsf{ajout}(t, e)$ est encore un ABR.

Propriété : La complexité de $\mathsf{ajout}(t, \cdot)$ est en $\Theta(h(t))$.

3.3 Suppression

On donne ci-dessous le pseudo-code de l'opération de suppression d'élément pour un arbre non vide.

Pseudo-code – supp (ABR t, $e \in \mathcal{E}$) $\to$ ABR

hyp : $\mathsf{est\_présent}(t, e)$ et $t \neq \mathrm{V}$

Si $t = N(x, \mathrm{V}, \mathrm{V})$ alors V *// nécessairement, $x = e$*
Si $t = N(x, g, d)$ alors
    Si $x = e$ alors
        Si $g \neq \mathrm{V}$ alors
            $m = \mathsf{max\_e}(g)$
            $N(m, \mathsf{supp}(g, m), d)$
        Sinon d
    Si $e \prec x$ alors $N(x, \mathsf{supp}(g, e), d)$
    Si $e \succ x$ alors $N(x, g, \mathsf{supp}(d, e))$

$\mathsf{max\_e}$ désignant ici une fonction donnant le plus grand élément du sous-ensemble de $\mathcal{E}$ formé des étiquettes de l'arbre pris en argument.

Propriété (*conservation de la structure d'ABR*) :

Si t est un ABR et $e \in \mathrm{et}(t)(\mathrm{ch}(t)) = \{\mathrm{et}(t)(n) \mid n \in \mathrm{ch}(t)\}$, alors $\mathsf{supp}(t, e)$ est encore un ABR. De plus, si $\mathrm{Card}\left((\mathrm{et}(t))^{-1}(\{e\})\right) = 1$, alors e n'est plus présent dans $\mathsf{supp}(t, e)$.

3.4 Limitation de la hauteur

Définition (*arbre localement complet*) :

Un arbre binaire est dit localement complet si chaque nœud interne a deux fils non vides.

Exemples :

Définition (*arbre complet*) :

Un arbre binaire de hauteur h est complet si :

$$h < 1 \text{ ou bien } \begin{cases} h \geq 1 \\ \text{il y a } 2^{h-1} \text{ nœuds de profondeur } h-1 \\ \text{les nœuds de profondeur } h \text{ sont le plus à gauche possible} \end{cases}$$

Toutes les feuilles sont alors de profondeur h ou $h-1$.

Propriété (*unicité et propriétés de l'arbre parfait à n nœuds*) :

Si l'on omet les étiquettes, alors pour tout $n \in \mathbb{N}^*$, il existe un unique arbre binaire parfait à n nœuds. De plus, pour un tel arbre :

i. sa hauteur est $\lfloor \log_2(n) \rfloor$
ii. $\forall\, k \in [0..h-1]$, il y a 2^k nœuds de profondeur k
iii. il y a $n - \sum_{k=0}^{h-1} 2^k$, c'est-à-dire $n - (2^h - 1)$ nœuds de profondeur h.

Propriété :

Soit $n \in \mathbb{N}^*$. L'arbre parfait à n nœuds minimise la somme des profondeurs des nœuds parùi les arbres binaires à n nœuds (mais ce n'est pas nécessairement le seul qui atteint ce minimum).

Un constat que l'on peut établir vis-à-vis de l'arbre parfait est qu'il manque de souplesse. En effet, l'opération d'ajout en feuille vue précédemment modifie complètement sa structure, qu'il est pourtant souhaitable de conserver puisqu'elle permet une recherche dichotomique en $\Theta(\log(n))$ (car elle est toujours en $\Theta(h)$). Mais cela nécessite alors d'avoir une opération d'insertion peu efficace.

Dans la partie suivante, on va donc manipuler un autre type d'arbres où l'on s'autorise à avoir trois clés au maximum par nœud, ce qui a pour effet de rendre la structure d'arbre parfait plus variable tout en maintenant ses propriétés avantageuses.

4 Arbres 2-3-4 et arbres rouge-noir

Dans cette partie, on fixe $\mathcal{S}$ un ensemble totalement ordonné.

4.1 Arbres 2-3-4

4.1.1 Définitions

Définition (*ensemble $\mathscr{A}_{2,3,4}(\mathcal{S})$*) :

On définit l'ensemble $\mathscr{A}_{2,3,4}(\mathcal{S})$ par induction à partir des règles de construction suivantes :
- $\mathrm{V}|^0_{\{\cdot\}}$ (arbre vide)
- $N^2|^2_{\mathcal{S}}$ (nœud binaire)
- $N^3|^3_{\mathcal{S}^{\leqslant}}$ (nœud ternaire)
- $N^4|^4_{\mathcal{S}^{\leqslant\leqslant}}$ (nœud quaternaire)

où $\mathcal{S}^{\leqslant} = \{(i,j) \in \mathcal{S}^2 \mid i \leqslant j\}$ et $\mathcal{S}^{\leqslant\leqslant} = \{(i,j,k) \in \mathcal{S}^3 \mid i \leqslant j \leqslant k\}$.

On peut étendre les définitions données pour $\mathscr{A}_B(\mathcal{S})$ à $\mathscr{A}_{2,3,4}(\mathcal{S})$, notamment :

· les nœuds comme des chemins, c'est-à-dire des mots sur $\Sigma = \{0, 1, 2, 3\}$, et en particulier :

$$\mathrm{ch} = \begin{pmatrix} \mathscr{A}_{2,3,4}(\mathcal{S}) & \to & \mathcal{P}_f(\Sigma^*) \\ \mathrm{V} & \mapsto & \{\varepsilon\} \\ N^2(x, a_1, a_2) & \mapsto & \{\varepsilon\} \cup (0 \cdot \mathrm{ch}(a_1)) \cup (1 \cdot \mathrm{ch}(a_2)) \\ N^3(x, a_1, a_2, a_3) & \mapsto & \{\varepsilon\} \cup (0 \cdot \mathrm{ch}(a_1)) \cup (1 \cdot \mathrm{ch}(a_2)) \cup (2 \cdot \mathrm{ch}(a_3)) \\ N^4(x, a_1, a_2, a_3, a_4) & \mapsto & \{\varepsilon\} \cup (0 \cdot \mathrm{ch}(a_1)) \cup (1 \cdot \mathrm{ch}(a_2)) \cup (2 \cdot \mathrm{ch}(a_3)) \cup (3 \cdot \mathrm{ch}(a_4)) \end{pmatrix}$$

· les feuilles
· la profondeur
· la hauteur (par induction ou bien comme la profondeur maximale d'un nœud)
· la taille, c'est-à-dire le nombre de nœuds
· les étiquettes d'un arbre
· les arbres "de recherche", prenant la forme suivante :

On ajoute de plus la définition de 2-nœud, 3-nœud et 4-nœud.

Définition (*arbre parfaitement équilibré*) :

On dit qu'un arbre de $\mathscr{A}_{2,3,4}(\mathcal{S})$ est parfaitement équilibré si toutes ses branches ont la même longueur.

Définition (*arbre 2-3-4*) :

On dit qu'un arbre de $\mathscr{A}_{2,3,4}(\mathcal{S})$ est un arbre 2-3-4 s'il est de recherche et parfaitement équilibré.

Remarque : La structure d'arbre de recherche assure une recherche d'élements en $\Theta(h(t))$. Puis, le caractère équilibré implique que $h(t) \in \Theta(\log(s(t)))$: en effet, en supprimant des nœuds dans t, on peut fabriquer $t' \in \mathscr{A}_B(\mathcal{S})$ parfaitement équilibré et de même hauteur, on a donc $s(t) \geq s(t') = 2^{h(t')+1} - 1 = 2^{h(t)+1} - 1$.

4.1.2 Opération de scission d'un 4-nœud

4.1.3 Opération d'insertion dans un arbre 2-3-4

4.1.4 Suppression

cf. ARN (arbres rouge-noir).

4.2 ARN

4.2.1 Définition

Définition (*arbre rouge-noir*) :

Un arbre rouge-noir ou ARN est un arbre binaire de recherche dans lequel les nœuds ont une couleur, rouge ou noir, de sorte que la racine est noire, un nœud rouge a exactement deux fils noirs ou bien deux fils vides, et chaque branche contient exactement le même nombre de nœuds noirs.

Proposition (*lien avec les arbres 2-3-4*) :

Remarque : La hauteur d'un ARN est logarithmique en le nombre de nœuds.

Définition (*branche (???)*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})$ et $(n_i)_{i\in[0..N]} \in \mathrm{ch}(t)^{N+1}$. $(n_i)_{i\in[0..N]}$ est une branche de t ssi :

$$\begin{cases} n_0 = \varepsilon \\ \forall\, i \in [0..N[,\ n_i \text{ est le père de } n_{i+1} \\ n_N \cdot 0 \notin \mathrm{ch}(t) \text{ ou } n_N \cdot 1 \notin \mathrm{ch}(t) \text{ (c'est-à-dire } n_N \text{ a un fils vide)} \end{cases}$$

4.2.2 Scission des 4-nœuds

4.2.3 Rotation simple

Comme illustré ci-dessus, les opérations de scission des 4-nœuds vues précédemment reposent sur des opérations intermédiaires dites de rotation simple. On en donne donc ici un pseudo-code :

Pseudo-code – rotation (nœuds u, v, p, orientation o_u) $\to$ ()

hyp : p.fg $= u$ si $o_u =$ gauche et p.fd $= u$ si $o_u =$ droite

```
u.fg ← v.fd
v.fd ← u
Si o_u = gauche alors p.fg ← v
Sinon p.fd ← v
```

Le schéma ci-dessous illustre le principe de cette opération dans le cas d'une orientation quelconque.

4.2.4 Suppression

Pseudo-code – suppression (ARN t, clé c) $\to$ ()

hyp : c est présent dans t

```
Trouver z le nœud de clé c
Si z.fg = V alors
    remplacer z par z.fd
    Si z était racine alors
        z.fd devient noir
    Sinon si z était noir alors
        répare(z.fd)
Sinon si z.fd = V alors
    remplacer z par z.fg
    Si z était racine alors
        z.fg devient noir
    Sinon si z était noir alors
        répare(z.fg)
Sinon    // alors, z.fg, z.fd ≠ V
    y = min_e(z.fd)    // y.fg = V nécessairement
    remplacer y par y.fd
    z.clé ← y.clé
    z.donnée ← y.donnée
    Si y était noir alors répare(y.fd)
```

où min_e donne la plus grande étiquette dans un arbre et répare est une fonction dont on donne le pseudo-code dans la sous-section qui suit.

4.2.5 Réparation post-suppression

Suite à la suppression d'un nœud noir qui menaçait l'équilibre de l'ARN, le nœud x doit porter une surcharge noire ("en souvenir du nœud disparu" en quelques sortes), car si on omet cette surcharge, il manque un nœud noir sur les branches passant par x. On cherche à absorber cette surcharge pour régulariser la situation.

Pseudo-code – répare (nœud x) $\to$ ()

```
Si x est rouge alors
    alors rendre x noir
Sinon
    Si x est racine alors
        rien à faire
    Sinon
        p ← père de x
        w ← frère de x
        Si w est rouge alors
            rotation p-w
            p devient rouge
            w devient noir
            w ← nouveau frère de x     // nécessairement noir car ex-fils du w rouge
        Si w est noir avec deux fils noirs alors
            w devient rouge
            répare(p)
        Sinon
            Si w est noir avec un fils noir et un fils rouge y tel que x-p // y-w
                rotation y-w
                y devient noir
                w devient rouge
                w ← nouveau frère de x (i.e. y)
            Si w est noir avec un fils noir et un fils rouge z tel que x-p # z-w
                rotation p-w
                w prend la couleur de p
                z devient noir
                et c'est fini
```

On illustre ci-dessous les quatre cas de la grande branche conditionnelle, les surcharges noires étant représentées par des icônes de fantômes.

4.3 Transition

Les ABR et les ARN nous ont permis d'implémenter le type abstrait **ensemble**, encore appelé **dictionnaire**, avec une complexité de recherche, d'insertion et de suppression logarithmique en le nombre d'élements. Les éléments étaient positionnés dans la structure en fonction de la valeur de leur clé relativement à celles des autres.

Dans la prochaine section, on envisage de positionner les éléments de la structure en fonction de la seule valeur de leur clé et non relativement à celles des autres, en vue d'avoir une procédure de recherche en temps constant (c'est-à-dire indépendante du nombre d'éléments de la structure).

5 Hachage

On suppose que l'on cherche à implémenter un ensemble dont les clés sont à valeurs dans un ensemble $\mathcal{C}$.

On se propose de stocker les éléments (ou du moins l'adresse des données) dans un tableau T de taille m à l'aide d'une fonction $f \in \mathcal{F}(\mathcal{C}, [0..m-1])$ de sorte qu'un élément $c \in \mathcal{C}$ soit stocké dans la case d'indice $f(c)$ de T.

Une telle fonction f est appelée fonction de hachage.

Une "bonne" fonction de hachage est :
- déterministe (la même clé donne la même case)
- efficace (opérations en temps constant)
- elle doit répartir au mieux les différentes clés

Pour le dernier point, on peut en particulier chercher à minimiser l'une des quantités suivantes :
- $\max_{i \in [0..m-1]} \text{Card}(f^{-1}\{i\})$ (nombre maximal de collisions dans une même case)
- $\sum_{c \in \mathcal{C}} p_c \,\text{Card}\, f^{-1}\{f(c)\}$ (nombre moyen de collisions, connaissant la distribution des clés $(p_c)_{c \in \mathcal{C}}$)
- $\frac{1}{m} \sum_{i=0}^{m-1} \text{Card}\, f^{-1}\{i\})$ (nombre moyen de collisions dans le cas d'une distribution uniforme)

6 Implémentation du tas

6.1 Arbres tournoi

*Dans cette section, on cherche à implémenter le type abstrait **tas (minimal)** grâce à des arbres binaires. Devinant qu'il faut limiter la hauteur de ces arbres pour limiter la complexité pire cas de ses opérations, on se restreint à des arbres binaires parfaits, sachant qu'ils ont une hauteur en $\Theta(\log(n))$ pour n nœuds (cf. définition dans le TD sur les arbres).*

En vue d'avoir accès au minimum en temps constant, on maintiendra la clé minimale à la racine. Cependant, cette seule contrainte n'est pas suffisante car elle ne donne aucune information quant à la position du second minimum dans l'arbre, ce qui impliquerait au pire une recherche exhaustive lors de la suppression du minimum, et donc une complexité en $\Theta(n)$.

C'est pourquoi on introduit la définition suivante.

Définition (*arbre tournoi*) :

Soit $t \in \mathscr{A}_B(\mathcal{S})$ où $\mathcal{S}$ est totalement ordonné. On dit que t est un (arbre) tournoi ssi

$$\forall\, (n, n') \in \text{ch}(t)^2,\ n' \text{ est fils de } n \implies \text{et}(t)(n') \geq \text{et}(t)(n)$$

Remarque : On obtient une définition équivalente en remplaçant la proposition "n' est fils de n" par "n' est descendant de n".

6.2 Opérations en pseudo-code

6.2.1 Minimum

Pseudo-code – min (ABPT (c-à-d tas) t) → `elem`

Retourner l'étiquette de la racine de t

Dans cette implémentation des tas sous forme d'arbres binaires parfaits tournois (ce que l'on a abrégé en ABPT), on a clairement accès au minimum en en $\Theta(1)$.

6.2.2 Insertion

Pseudo-code – insere (ABPT t, `elem` e, priorité p) → ()

Soit pere le premier nœud(*) ayant un fils vide(**)
Si pere.fg = V alors
    ajouter un nœud d'étiquette (e, p) en fils gauche de pere
Sinon   *// dans ce cas,* **pere.fd = V**
    ajouter un nœud n d'étiquette (e, p) en fils droit de pere
Tant que (n.prio < pere.prio) et (pere ≠ t.racine)
    échanger les étiquettes de n et pere
    $n \leftarrow$ pere
    pere $\leftarrow n$.pere
Si n.prio < pere.prio alors
    échanger les étiquettes de n et pere

Remarque : (*)On ordonne ici les nœuds par profondeur croissante et de gauche à droite.

Remarque : (**)Cette ligne peut s'effectuer en $\Theta(1)$ grâce à une information que l'on peut accessoirement choisir de stocker dans le tas (ou plutôt dans la structure l'implémentant).

6.2.3 Suppression du minimum

Pseudo-code – supprime_min (ABPT t) → ()

hyp : t non vide

Soit n le dernier nœud
Remplacer ce nœud par V dans t(*)
Remplacer l'étiquette de la racine par celle de n
$n \leftarrow t$.racine
Tant que (n.prio > n.fg.prio) ou (n.prio > n.fd.prio)(**)
    Si n.fg.prio < n.fg.prio alors
        échanger les étiquettes de n et n.fg
        $n \leftarrow n$.fg
    Sinon
        échanger les étiquettes de n et n.fd
        $n \leftarrow n$.fd

Remarque : (*)Ceci peut encore une fois se faire en $\Theta(n)$ ou en $\Theta(\log(n))$ à partir d'une information stockée dans le tas.

Remarque : (**)Il faut supposer ici que V.prio = $+\infty$ de façon à ce que la boucle termine.

En utilisant ce pseudo-code, on peut relativement facilement implémenter des tas :

- par tableau, en en retenant la première case vide (ou la dernière non vide)
- avec des structures représentant les nœuds et un pointeur vers le champ correspondant au premier vide.

On propose une troisième implémentation de tas en utilisant un type récursif OCaml pour les arbres binaires et une liste de booléens indiquant le chemin du premier vide dans l'arbre.

On définit ainsi les fonctions suivantes, qui peuvent être utiles pour les opérations élémentaires :

$$\text{suivant} = \begin{pmatrix} \Sigma^* \to \Sigma^+ \\ \varepsilon \mapsto 0 \\ u \cdot 0 \mapsto u \cdot 1 \\ u \cdot 1 \mapsto \text{suivant}(u) \cdot 0 \end{pmatrix} \qquad \text{prec} = \begin{pmatrix} \Sigma^+ \to \Sigma^* \\ 0 \mapsto \varepsilon \\ v \cdot 1 \mapsto v \cdot 0 \\ v \cdot 0 \mapsto \text{prec}(v) \cdot 1 \end{pmatrix}$$