Structures de données arborescentes

1 Motivations

1.1 Un arbre pour un objet

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

Exemple : Les expressions booléennes, arithmétiques, de type.

1.2 Un arbre pour une collection d'objet

On cherche à stocker une collection d'objets sans multiplicité, et dont l'ordre relatif n'est pas significatif (comme dans le cas d'un ensemble).

On suppose que tous les éléments sont du type `elem`, et qu'ils sont identifiés de manière unique par une clé, c'est à dire une sous-partie permettant l'identification.

On a alors les méthodes suivantes :

- `creer_ens_vide : () -> ens`
- `est_ens_vide : ens -> bool`
- `appartient : ens×elem -> bool`
- `ajoute_elem : ens×elem -> ens`
- `supprime_elem : ens×clé -> ens`
- `trouve_elem : ens×clé -> elem`

Remarque : On peut aussi imaginer des fonctions `ajoute_elem` et `supprime_elem` qui modifieraient directement l'ensemble donné en entrée, plutôt que de renvoyer un nouvel ensemble.

1.3 Implémentations

On peut stocker les éléments dans une liste. On peut aussi associer chaque élément à une clé (pour un ensemble de caractères par exemple, leurs valeurs ascii), qui permet de les comparer rapidement. Si l'on peut ordonner ces clés, on peut alors classer les éléments par ordre croissant dans un tableau.

Opération	Complexité (Liste)	Complexité (Tableau ordonné)
`appartient`	$\theta(n)$	$\theta(\log(n))$ (dichotomie)
`ajoute_elem`	$\theta(1)$	$\theta(n)$
`supprime_elem`	$\theta(n)$	$\theta(n)$
`trouve_elem`	$\theta(n)$	$\theta(\log(n))$

Remarque : On voit selon le contexte qu'une certaine implémentation sera plus efficace qu'une autre : si on doit faire beaucoup d'insertions sans trop chercher d'éléments, le plus efficace sera la liste. Par contre, si on n'insère que rarement des éléments mais que l'on est souvent ammené à chercher dans les éléments, le tableau trié sera à préférer.

ll existe cela dit une structure qui permet d'avoir une complexité d'ajout / suppression et de recherche en $\theta(\log(n))$, sous certaines conditions : il s'agit des **arbres binaires de recherche (ABR)**.

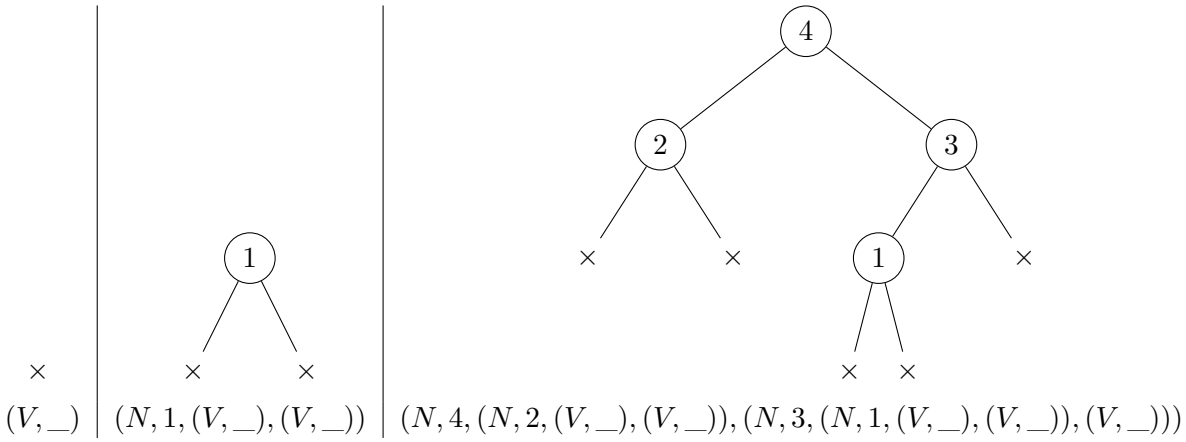
2 Arbre binaires

2.1 Ensemble $\mathscr{A}_B$

Soit $\mathcal{S}$ un ensemble. On définit l'ensemble des arbres binaires étiquetés par $\mathcal{S}$ comme l'ensemble construit par induction à partir de ces deux règles :

- $V\Big|_{\{-\}}^{0}$
- $N\Big|_{\mathcal{S}}^{2}$

Exemple : Soit $\mathcal{S} = \{1, 2, 3, 4\}$.



2.2 Feuilles

On fixe désormais $\mathcal{S}$ un ensemble d'étiquettes. On notera par ailleurs $N(x, g, d) = (N, x, g, d)$ et $V = (V, \_)$ pour alléger l'écriture.

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

Propriété : *Pour la relation d'ordre $\leq$ associée à la définition inductive de $\mathscr{A}_B(\mathcal{S})$, on a :*

1. *V est le seul élément minimal de $(\mathscr{A}_B(\mathcal{S}), \leq)$.*
2. *t élément minimal de $\mathscr{A}_B(\mathcal{S}) \setminus \{V\} \Leftrightarrow t$ est réduit à une feuille.*

▷ Le 1) découle du fait que V soit le seul cas de base des règles d'induction de $\mathscr{A}_B(\mathcal{S})$.

▷ Soit t minimal dans $\mathscr{A}_B(\mathcal{S}) \setminus \{V\}$. Il existe par définition $x \in \mathcal{S}$ et $(g, d) \in \mathscr{A}_B(\mathcal{S})^2$ tels que $t = N(x, g, d)$. Comme $g \leq t$ et $d \leq t$, et d'autre part $g \neq t$ et $d \neq t$, on en déduit que $g < t$ et $d < t$. Par minimalité de t dans $\mathscr{A}_B(\mathcal{S}) \setminus \{V\}$, on en déduit que $g = d = V$, donc que t est réduit à une feuille.

2.3 Chemins

Considérons l'alphabet $\Sigma = \{0, 1\}$. On définit par induction sur $\mathscr{A}_B(\mathcal{S})$ l'ensemble des chemins admissibles d'un arbre binaire t, noté $\mathrm{ch}(t)$, par :

$$\forall t \in \mathscr{A}_B(\mathcal{S}) : \quad \mathrm{ch}(t) = \begin{cases} \varnothing \text{ si } t = V \\ \{\varepsilon\} \cup \{0.\mathrm{ch}(g)\} \cup \{1.\mathrm{ch}(d)\} \text{ si } t = N(x, g, d) \end{cases}$$

Soit $t \in \mathscr{A}_B(\mathcal{S})$. Un **noeud** $\mathcal{N}$ de t est un élément de $\mathrm{ch}(t)$. Sa profondeur, notée $|\mathcal{N}|$ est alors sa longueur en tant que mot de Σ^*. La taille de t, notée $s(t)$, est alors le nombre de noeuds de t, autrement dit $\mathrm{cardch}(t)$.

Remarque : Un chemin admissible décrit la "position" d'un "noeud" dans l'arbre.

Remarque : Sans compter les étiquettes, il est possible de construire un arbre à partir de l'ensemble de ses chemins admissibles.

Exercice 1: Définir par induction une fonction qui prend en entrée l'ensemble des chemins admissibles d'un arbre et renvoie un arbre ayant le même ensemble des chemins admissibles, dont tous les noeuds sont étiquettés par 1.

Exercice 2: Donner une définition inductive de $s(t)$.

On appelle **étiquetage** d'un arbre non vide la fonction qui à un noeud de l'arbre associe son étiquette.

Formellement, l'étiquetage est défini inductivement comme suit (on note $\mathcal{E}$ l'ensemble des étiquetages des arbres, c'est à dire des fonctions d'une partie de Σ^* dans $\mathcal{S}$) :

$$\text{etiq}\left(\begin{array}{rcl} \mathscr{A}_B(\mathcal{S})\setminus\{V\} & \to & \mathcal{E} \\ N(x,V,V) & \mapsto & \left(\begin{array}{rcl} \text{ch}(x,V,V)=\{\varepsilon\} & \to & \mathcal{S} \\ \varepsilon & \mapsto & x \end{array}\right) \\ N(x,g,V) & \mapsto & \left(\begin{array}{rcl} \text{ch}(N,x,g,V) & \to & \mathcal{S} \\ \varepsilon & \mapsto & x \\ 0\cdot u & \mapsto & \text{etiq}(g)(u) \end{array}\right) \\ N(x,V,d) & \mapsto & \left(\begin{array}{rcl} \text{ch}(N,x,V,d) & \to & \mathcal{S} \\ \varepsilon & \mapsto & x \\ 1\cdot u & \mapsto & \text{etiq}(d)(u) \end{array}\right) \\ N(x,g,d) & \mapsto & \left(\begin{array}{rcl} \text{ch}(N,x,g,d) & \to & \mathcal{S} \\ \varepsilon & \mapsto & x \\ 0\cdot u & \mapsto & \text{etiq}(g)(u) \\ 1\cdot u & \mapsto & \text{etiq}(d)(u) \end{array}\right) \end{array}\right)$$

Soit $t \in \mathscr{A}_B(\mathcal{S})$, etiq$(t)$ est l'étiquetage de t. Si $n \in \text{ch}(t)$, alors l'étiquette de n dans t est $(\text{etiq}(t))\,(n)$

Remarque : Il est possible de reconstruire un arbre à partir de son ensemble des chemins admissibles et sa fonction d'étiquetage.

Exercice 3: Définir une fonction qui reconstruit un arbre à partir de son ensemble des chemins admissibles et sa fonction d'étiquetage.

2.4 Vocabulaire

Soit $t \in \mathscr{A}_B(\mathcal{S})$. Soit $n \in \text{ch}(t)$.

- n est une **feuille** de t si et seulement si pour tout $u \in \Sigma^*$, $n \cdot u \in \text{ch}(t) \Rightarrow u = \varepsilon$
- n est **racine** de t si et seulement si $n = \varepsilon$
- n est un **noeud interne** de t si et seulement si n n'est pas une feuille.

Soient n et m deux chemins admissibles pour t un arbre binaire non vide. m est le **fils gauche** (resp. **fils droit**) de n si et seulement si $m = n \cdot 0$ (resp. $m = n \cdot 1$). Dans les deux cas, n est alors le **père** de m.

On dit que m est un **descendant** de n si et seulement si il existe $u \in \Sigma^*$ tel que $m = n \cdot u$. Dans ce cas n est un **ascendant** de m.

Remarque : Les feuilles sont les noeuds sans enfant, et la racine est le seul noeud sans père.

Soit $(n_i)_{i\in[\![0,k]\!]} \in \text{ch}(t)^{k+1}$. On dit que (n_i) est une **branche** de t si et seulement si $n_0 = \varepsilon$ et n_i est le père de n_{i+1} pour tout $i \in [\![0, k[\![$.

Soient t et t' deux arbres binaires sur $\mathcal{S}$. On dit que t' est le **sous-arbre droit** (resp **sous-arbre gauche**) de t si et seulement si il existe $g \in \mathscr{A}_B(\mathcal{S})$ (resp. $d \in \mathscr{A}_B(\mathcal{S})$) tel que $t = N(x, g, t')$ (resp. $t = N(x, t', d)$). On dit que t' est un **sous-arbre** de t si et seulement si $t' \leq t$.

Remarque : Notons qu'un arbre binaire peut-être sous-arbre d'un autre tout en étant ni un sous-arbre droit, ni un sous-arbre gauche.

2.5 Hauteur

On définit par induction la hauteur $h(t)$ d'un arbre binaire $t \in \mathscr{A}_B(\mathcal{S})$ comme valant -1 si $t = V$ et $1 + \max(h(g), h(d))$ si $t = N(x, g, d)$.

Propriété : *Soit $t \in \mathscr{A}_B(\mathcal{S}) \setminus \{V\}$. Alors $h(t) = \max_{n \in ch(t)} \text{prof}(n)$. $h(t)+1$ est alors la longueur maximale d'une branche.*

▷ Montrons-le par induction sur t : c'est bien le cas pour $t = N(x, V, V)$ car $h(t) = 0$, le seul chemin admissible pour cet arbre est ε qui est de longueur 0, et la seule branche de t est alors (ε), qui est de taille 1.

▷ Soit $t = N(x, g, d) \in \mathscr{A}_B(\mathcal{S})$ avec g et d deux arbres binaires non vides vérifiant la propriété. On a

$$\max_{n \in \text{ch}(t)} \text{prof}(n) = \max(\max_{0 \cdot n \in \text{ch}(t)} \text{prof}(n), \max_{1 \cdot n \in \text{ch}(t)} \text{prof}(n), 0)$$

en séparant les chemins admissibles de t selon leur première lettre. Par définition des chemins admissibles et de la profondeur, on a alors

$$\max_{n \in \text{ch}(t)} \text{prof}(n) = \max(1 + \max_{n \in \text{ch}(g)} \text{prof}(n), 1 + \max_{n \in \text{ch}(d)} \text{prof}(n), 1)$$

soit, par hypothèse sur g et d,

$$\max_{n \in \text{ch(t)}} \text{prof}(n) = 1 + \max(h(g), h(d), 0) = 1 + \max(h(g), h(d)) = h(t)$$

car g et d ne sont pas vides et ont donc une hauteur plus grande que 0. On établit le résultat sur la longueur maximale des branches de la même manière.

▷ Les cas $t = N(x, g, V)$ et $t = N(x, V, d)$ se traitent de la même manière (la séparation des maximums donne alors un maximum d'un ensemble vide, soit $-\infty$, qui est neutre pour le maximum, ce qui traduit l'inexistance de branches / noeuds à droite ou à gauche de la racine).

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

▷ Montrons-le par induction sur t : pour $t = V$, on a $h(t) + 1 = 0$, $s(t) = 0$ et $2^{h(t)+1} - 1 = 0$.

▷ Soit $t = N(x, g, d) \in \mathscr{A}_B(\mathcal{S})$, où g et d respectent la propriété énoncée. Alors

$$h(t) + 1 = 2 + \max(h(d), h(g)) \leq 2 + h(d) + h(g)$$

$$1 + (h(g) + 1) + (h(d) + 1) = 3 + h(g) + h(d) \leq 1 + s(g) + s(d)$$

$$s(g) + s(d) + 1 \leq 2^{h(t)+1} - 1 + 2^{h(d)+1} \leq 2^{\max(h(t),h(d))+1} - 1 = 2^{h(t)+1} - 1$$

En combinant ces inégalités, il vient le résultat attendu.

2.6 Parcours

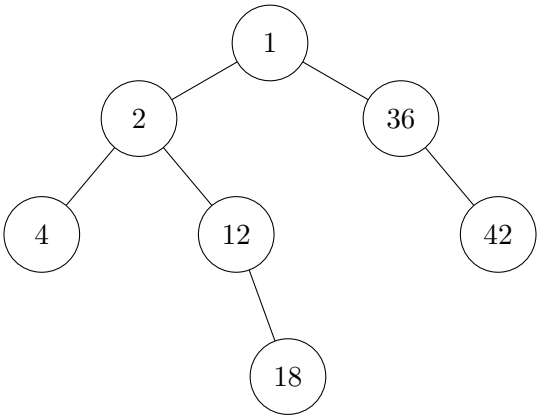
Soit $t \in \mathscr{A}_B(\mathcal{S}) \setminus \{V\}$. $(a_i)_{i \in [\![1, s(t)]\!]}$ est un **parcours** de t si et seulement si il existe $\varphi \in \mathcal{F}(\text{ch}(t), [\![1, s(t)]\!])$ bijective telle que $\forall n \in \text{ch}(t), a_{\varphi(n)} = \text{etiq}(t)(n)$.

Pour $n \in \text{ch}(t)$ on note

- $\mathscr{G}(n) = \{n \cdot 0 \cdot u \,|\, u \in \Sigma^*\} \cap \text{ch}(t)$ l'ensemble des descendants gauches de n ;
- $\mathscr{D}(n) = \{n \cdot 1 \cdot u \,|\, u \in \Sigma^*\} \cap \text{ch}(t)$ l'ensemble des descendants droits de n.

Un parcours $(\text{etiq}(t)(\varphi^{-1}(i)))_{i \in [\![1, s(t)]\!]}$ (où φ est une bijection de $\text{ch}(t)$ dans $[\![1, s(t)]\!]$) est dit **préfixe** (resp. **postfixe**, **infixe**) si et seulement si pour tout $n \in \text{ch}(t)$, pour tout $g \in \mathscr{G}(n)$ et $d \in \mathscr{D}(n)$, on a $\varphi(n) \leq \varphi(g) \leq \varphi(d)$ (resp. $\varphi(g) \leq \varphi(d) \leq \varphi(n)$, $\varphi(g) \leq \varphi(n) \leq \varphi(d)$).

Exemple : Pour l'arbre suivant (les noeuds vides n'ont pas été représentés) :



- Un parcours préfixe est 1 ; 2 ; 4 ; 12 ; 18 ; 36 ; 42 ;
- Un parcours postfixe est 4 ; 18 ; 12 ; 2 ; 42 ; 36 ; 1 ;
- Un parcours infixe est 4 ; 2 ; 12 ; 18 ; 1 ; 36 ; 42.

Propriété : *Il y a unicité des parcours préfixes, infixes et postfixes.*

▷ En exercice (4).

On définit aussi le **parcours en largeur**, pour lequel $(\text{prof}\,(\varphi^{-1}(i)))_{i\in[\![1,s(t)]\!]}$ doit être croissante, et de parcours en profondeur (hors programme dans le cadre des arbres, mais cela suit le même principe que les parcours en profondeur dans les graphes).

3 Arbres binaires de recherches (ABR)

Dans cette partie, E désigne un ensemble muni d'une relation d'ordre totale, notée $\preceq$.

Soit $t \in \mathscr{A}_B(E) \setminus \{V\}$. On pose $e = \text{etiq}(t)$. t est un arbre binaire de recherche (abgrégé ABR) si et seulement si pour tout $n \in \text{ch}(t)$, $\max\limits_{g\in\mathscr{G}(n)} e(g) \leq e(n) < \min\limits_{d\in\mathscr{D}(n)} e(n)$. Par convention, l'arbre vide est un arbre de recherche.

3.1 Recherche d'éléments

Profitons de la structure ordonnée de l'ABR. On procède par dichotomie pour la recherche d'un élément (d'une étiquette).

Algorithme 1 : Recherche d'élément dans un ABR

Entrées : $t \in \mathscr{A}_B(E)$ un ABR à étiquettes dans E, $e \in E$ l'élément à rechercher dans t.
Sorties : Vrai s'il existe un noeud dans t ayant pour étiquette e, faux sinon.

```
si t = V alors
    Renvoyer faux
sinon
    Poser t = N(x, g, d) ;
    si e = x alors
        Renvoyer vrai
    sinon
        si e < x alors
            Rechercher e dans g
        sinon
            Rechercher e dans d
        fin
    fin
fin
```

($\mathbb{B}$ désigne l'ensemble des booléens : vrai ou faux)

Propriété : *Pour tout noeud n d'un arbre binaire de recherche $t \in \mathscr{A}_B(E)$, s'il n'existe pas d'autre noeud dans t ayant la même étiquette que n, alors le nombre de comparaisons (entre éléments de E) effectuées lors de la recherche de* $\text{etiq}(t)(n)$ *dans t vaut* $2\text{prof}(n) + 1$.

▷ Montrons-le par récurrence sur la profondeur du noeud n : le prédicat à prouver est, pour $k \in \mathbb{N}$

$\mathcal{P}(k)$: pour tout noeud de taille k d'un arbre $t \in \mathscr{A}_B(E)$, s'il n'existe pas d'autre noeud dans t de même étiquette que n, alors le nombre de comparaison lors de la recherche de $e = \text{etiq}(t)(n)$ dans t vaut $2k + 1$.

▷ Pour $k = 0$, on a nécessairement $n = \varepsilon$, ce qui implique que l'arbre sur lequel on appelle la fonction n'est pas vide, et que sa racine est étiquetée par e : ainsi, on effectue une seule comparaison (l.5) avant de renvoyer vrai. D'où $\mathcal{P}(0)$.

▷ Soit $k \in \mathbb{N}$. Supposons $\mathcal{P}(k)$. Soit n un chemin de profondeur $k + 1$ d'un arbre binaire de recherche t. Lors de l'appel initial à la recherche, le test d'égalité ligne 5 échoue (sinon on aurait deux noeuds étiquettés par e, ce qui est exclu par hypothèse). On engendre donc un appel au sous-arbre gauche ou droit (cela dépend du premier caractère de $n = 0 \cdot m$ ou $n = 1 \cdot m$) après une seconde comparaison. Dans ce sous-arbre, le noeud étiqueté par e dans l'arbre initial est m (par définition inductive de l'étiquetage), et est toujours le seul à être étiqueté par e. Comme m est de profondeur k, on sait d'après $\mathcal{P}(k)$ que la suite de la recherche va alors engendrer $2k + 1$ nouvelles comparaisons. Sommées avec les comparaisons déjà effectuées, on obtient un total de $2k + 1 + 2 = 2(k + 1) + 1$ comparaisons, d'où $\mathcal{P}(k + 1)$.

Corrolaire : *Pour tout $e \in E$, la recherche de x dans t engendrera au plus $2h(t) + 2$ comparaisons.*

▷ Si e n'est pas dans t, on le prouve par une induction élémentaire sur t. Sinon, soit n un noeud de profondeur minimale parmi tous les noeuds étiquetés par e. En reprenant la preuve précédente en utilisant la minimalité de la profondeur de n plutôt que le fait que n soit le seul noeud à être étiquetté par x, on prouve que l'algorithme effectue $2\text{prof}(n) + 1$ comparaisons. D'après la remarque sur le lien entre profondeur des noeuds est hauteur, le nombre total de comparaisons effectuées est bien majoré par $2h(t) + 1$.

3.2 Ajout en feuille

Algorithme 2 : Ajout en feuille dans un arbre binaire de recherche

Entrées : $t \in \mathscr{A}_B(E)$ un ABR, $e \in E$ un élément à ajouter dans t
Sorties : t' un ABR contenant tous les éléments de t (avec multiplicité) et e.

```
1  si t = V alors
2  |  Renvoyer N(e, V, V)
3  sinon
4  |  Poser t = N(x, g, d) ;
5  |  si e ≤ x alors
6  |  |  Ajouter e en feuille à g. On note g' l'arbre obtenu ;
7  |  |  Renvoyer N(x, g', d)
8  |  sinon
9  |  |  Ajouter e en feuille à d. On note d' l'arbre obtenu ;
10 |  |  Renvoyer N(x, g, d')
11 |  fin
12 fin
```

Propriété : *Si t est un ABR, l'arbre obtenu en ajoutant à t un élément en feuille est toujours un ABR*

▷ En exercice (5)(par induction sur t).

Propriété : *La complexité temporelle de l'ajout en feuille est en $O(h(t))$.*

▷ On montre par induction sur t que l'ajout en feuille engendre au plus $h(t) + 1$ comparaisons entre éléments de E.

3.3 Suppression

Algorithme 3 : Suppression dans un ABR

Entrées : $t \in \mathscr{A}_B(E)$ un ABR, e un élément de E à supprimer de t. On suppose que e est dans t (dans ce cas $t \neq V$).

Sorties : Un arbre binaire de recherche t' avec pour chaque élément de E le même nombre de noeuds étiquettés par cet élément que dans t, sauf pour e pour lequel il y en aura un de moins.

```
1  si t = N(x, V, V) (nécessairement x = e) alors
2  |  Renvoyer V
3  sinon
4  |  Poser t = N(x, g, d) ;
5  |  si x = e alors
6  |  |  Poser m = g ;
7  |  |  tant que m ≠ V faire
8  |  |  |  Poser m = N(x', g', d') ;
9  |  |  |  m ← d'
10 |  |  fin
11 |  |  Supprimer x' de g ; on note g' l'arbre obtenu ;
12 |  |  Renvoyer N(x', g', d)
13 |  sinon
14 |  |  si x < e alors
15 |  |  |  Supprimer e dans g ; on note g' l'arbre obtenu ;
16 |  |  |  Renvoyer N(x, g', d)
17 |  |  sinon
18 |  |  |  Supprimer e dans d ; on note d' l'arbre obtenu ;
19 |  |  |  Renvoyer N(x, g, d')
20 |  |  fin
21 |  fin
22 fin
```

Remarque : Les lignes 6 à 10 permettent de trouver le fils le plus à droit du sous-gauche droit du noeud supprimé, fils dont l'étiquette peut remplacer celle du noeud supprimé sans perturber le caractère "de recherche" de l'arbre. On peut ensuite supprimer facilement ce fils en "remontant d'un cran" son sous-arbre gauche.

Remarque : Les opérations lignes 6 à 10 peuvent aussi consister à trouver le fils le plus à gauche du sous-arbre droit. Le seul problème est que l'on pourra potentiellement passer plusieurs fois par x' l'élément minimal du sous-arbre droit en descendant pour trouver le fils le plus à gauche, engendrant potentiellement bien plus d'appels récursifs et donnant une complexité en $\mathcal{O}(h^2)$ (prendre l'exemple d'un arbre étiqueté uniquement par x contenant une racine avec à sa droite un peigne s'étendant sur la gauche et un noeud vide à sa gauche. Le cas symétrique n'est pas possible à cause de l'inégalité stricte imposée dans la définition d'un ABR).

Propriété : *(Correction de l'algorithme de suppression) Si t est un ABR contenant e (i.e $e \in \mathrm{etiq}(t)(\mathrm{ch}(t))$), alors l'arbre t' obtenu après application de l'algorithme de suppression de e à t est aussi un ABR. De plus,*

$$\mathrm{card}\{c \in \mathrm{ch}(t') \,|\, \mathrm{etiq}(t)(c) = e\} = \mathrm{card}\{c \in \mathrm{ch}(t) \,|\, \mathrm{etiq}(t)(c) = e\} - 1$$

et pour tout $x \in E \setminus \{e\}$,

$$\mathrm{card}\{c \in \mathrm{ch}(t') \,|\, \mathrm{etiq}(t)(c) = x\} = \mathrm{card}\{c \in \mathrm{ch}(t) \,|\, \mathrm{etiq}(t)(c) = x\}.$$

- ▷ Par induction sur t : c'est immédiat si $t = N(e, V, V)$. Soit alors $t = N(x, g, d)$ un ABR. g et d sont alors eux-mêmes des ABR. Supposons que la propriété soit vraie pour g et d.
- ▷ Si $e < x$, comme t est un ABR, on sait que le plus grand élément étiquetant un noeud de g est plus petit que x, et que x est dans g. En notant g' l'arbre obtenu en supprimant e dans g (qui est donc

par hypothèse un ABR), celui-ci ne contiendra (toujours par hypothèse d'induction) que des éléments inférieurs à x : ainsi $t' := N(x, g', d)$ est bien un ABR. De plus, on a, pour tout $y \in E \setminus x$:

$$\text{card}\{c \in \text{ch}(t') \,|\, \text{etiq}(t')(c) = y\} - \text{card}\{c \in \text{ch}(t) \,|\, \text{etiq}(t)(c) = y\}$$

$$= \text{card}\{c \in \text{ch}(g') \,|\, \text{etiq}(g')(c) = y\} - \text{card}\{c \in \text{ch}(g) \,|\, \text{etiq}(g)(c) = y\}$$

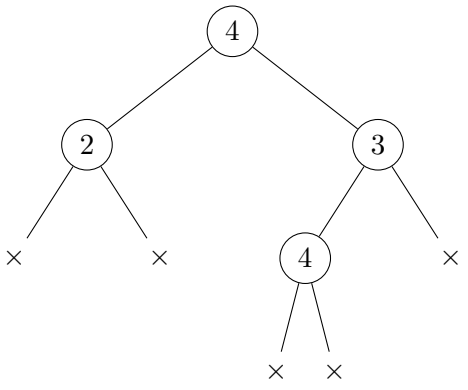
(en séparant entre l'élément étiquetant la racine, les éléments dans g/g' et dans d, ces derniers étant les mêmes dans t comme dans t'). On en déduit la seconde partie du résultat par hypothèse sur g. Le cas $e > x$ se traite de manière similaire.

▷ Pour le cas $x = e$, il est clair par la structure d'ABR de g que x' désigne à la ligne 11 la valeur de la plus grande étiquette dans g. À l'issue de la ligne 11, g' est alors par hypothèse un ABR dont les étiquettes sont toutes inférieures ou égales à x. Par ailleurs, t étant un ABR, pour tout y étiquettant un noeud de $\mathscr{D}(\varepsilon)$ (soit un noeud du sous arbre-droit de t), $x' < y$: ainsi $t' = N(x', g', d)$ est bien un ABR. Le nombre de noeuds étiquetés par x' diminue de 1 par hypothèse sur g entre g et g', mais augmente ensuite de 1 dans l'arbre renvoyé (x' étiquette la racine). e n'étiquettant alors plus la racine, il y a un noeud étiquetté par e de moins entre t et t'. Les autres égalités sur les nombres de noeuds étiquettés par chaque élément de E découlent immédiatement de l'hypothèse d'induction sur g.

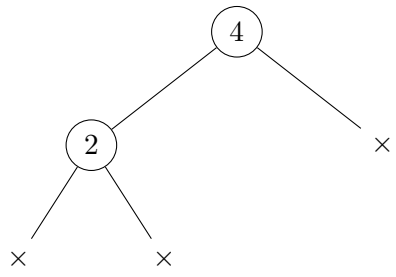
3.4 Limitation de la hauteur

Un arbre binaire est dit **localement complet** ou **strict** si et seulement si chaque noeud interne a 2 fils non vides.

Exemple :



Arbre localement complet



Arbre non localement complet

On dit qu'un arbre binaire t de hauteur h est **presque complet** ou **complet** si et seulement si $h < 1$ ou $h \geq 1$ et toutes les feuilles sont de profondeur h ou $h-1$, dont 2^{h-1} de profondeur $h-1$, et les noeuds de profondeur h sont "le plus à gauche possible" (i.e il n'existe pas un chemin de longeur h qui ne soit pas un noeud de t dont la valeur en base 2 soit inférieure à celle d'un noeud de t de profondeur h).

Remarque : On croise parfois l'appelation "arbre parfait" pour "arbre complet". Dans ce cours, un arbre parfait est arbre où toutes les racines ont la même profondeur.

Lemme : *Un arbre binaire complet de hauteur $h \in \mathbb{N}^*$ a au moins 2^h noeuds*

▷ Dans un arbre binaire t, si c est un chemin admissible de t, alors tous les préfixes de c sont des chemins de t. Comme un arbre binaire complet a 2^{h-1} noeuds de profondeur $h-1$, tout mot de $\{0,1\}^{h-1}$ est un chemin admissible de t : En comptant tous les préfixes de ces mots, il y a donc au moins $\sum_{k=0}^{h-1} 2^k = 2^h - 1$ noeuds dans t. De plus, comme t est de hauteur h, il existe au moins un noeud de hauteur h, non compté parmi les précédents : t contient au moins 2^h noeuds.

Propriété : *Aux étiquettes près, il existe pour tout entier $n \in \mathbb{N}^*$ un unique arbre binaire complet à n noeuds (i.e les ensembles des chemins admissibles pour deux arbres binaires complets avec le même nombre de noeuds sont identiques). Sa hauteur est $h = \lfloor \log_2(n) \rfloor$ et $\forall k \in [\![0, h-1]\!]$, il y a 2^k noeuds de profondeur k, et il y a $n - 2^h + 1$ noeuds de profondeur h.*

▷ Soit $n \in \mathbb{N}$. Soit t un arbre binaire complet à n noeuds. Notons h sa hauteur. Si $h = 0$, l'arbre est nécessairement réduit à la racine, et vérifie dans ce cas toutes les propriétés annoncées. Sinon, $h \geq 1$: dans ce cas, on sait que $2^h \leq n$ d'après le lemme, et $n < 2^{h+1} - 1$. En prenant la partie entière du logarithme en base 2, il vient $h \leq \lfloor \log_2(n) \rfloor \leq h$, d'où $h = \lfloor \log_2(n) \rfloor$.

▷ D'après la remarque effectuée dans le lemme, en hautant h la hauteur de t l'arbre binaire complet étudié, on sait que tous pour tout $k \in [\![0, h-1]\!]$, $\{0,1\}^k \subset \mathrm{ch}(t)$: ainsi t a exactement 2^k noeuds de profondeur k pour $k \in [\![0, h-1]\!]$.

▷ Tout noeud de t est de profondeur au plus h : le nombre de noeuds de profondeur h de t est donc égal au nombre total de noeuds, moins le nombre de noeuds de profondeur inférieure ou égale à $h-1$: ainsi d'après le résultat précédent, il y a $n - \sum_{k=0}^{h-1} 2^k = n - (2^h - 1) = n - 2^h + 1$.

▷ Soit t un arbre binaire complet à n noeuds. Notons h sa hauteur. Pour tout $k \in [\![0, h-1]\!]$, $\{0,1\}^k \subset \mathrm{ch}(t)$. Notons alors $C = \mathrm{ch}(t) \setminus \bigcup_{k=0}^{h-1} \{0,1\}^k$. Comme t est de hauteur h, on a $C \subset \{0,1\}^h$. En notant $p = n - 2^h + 1 = \mathrm{card}(C)$, il est immédiat d'après la définition d'un arbre binaire complet que C est l'ensemble des écritures en base 2 à h chiffres des nombres de $[\![0, p[\![$, d'où l'unicité de l'ensemble des chemins pour un arbre binaire complet à n noeuds.

Propriété : *L'arbre complet à n noeuds minimise la somme des profondeurs les noeuds parmi les arbres binaires à n noeuds.*

▷ On sait qu'il y a au plus 2^p noeuds de profondeur p dans un arbre binaire (c'est le cardinal de $\{0,1\}^p$). De plus, on a intérêt à "remplir" tous les niveaux de profondeur : si pour un arbre binaire t à n noeuds et de hauteur h noeuds il existe un chemin $c \in \{0,1\}^p \setminus \mathrm{ch}(t)$ (pour $p \in [\![0, h-1]\!]$), la somme des hauteurs de l'arbre obtenu en supprimant un noeud de profondeur h de t et en ajoutant le noeud c à t, on obtient un arbre avec le même nombre de noeuds mais une somme des profondeurs strictement plus petite. On établit alors une B borne inférieure de la somme des profondeurs des noeuds d'un arbre binaire à n noeuds, en notant $K = \min\{p \in \mathbb{N} \,|\, \sum_{k=0}^{p} 2^p < n\} = h - 1$:

$$B = \sum_{k=0}^{K} 2^k \times k + (n - \sum_{k=0}^{K} 2^k) \times h = 2\left(x \mapsto \sum_{k=0}^{K} x^k\right)'(2) + nh - h \times (2^h - 1)$$

en remarquant que $2^k k = 2k2^{k-1} = 2\left(x \mapsto x^k\right)'(2)$. Alors

$$B = 2\left(x \mapsto \frac{x^h - 1}{x - 1}\right)'(2) + nh - h \times (2^h - 1) = 2\left(x \mapsto \frac{hx^{h-1}(x-1) - (x^h - 1)}{(x-1)^2}\right)(2) + nh - h \times (2^h - 1)$$

$$= 2^h h - 2^{h+1} + 2 + nh - h2^h + h = (n+1)h - 2(2^h - 1)$$

Cette borne est atteinte pour un arbre binaire complet (en exercice (6)), d'où le résultat.

4 Arbres 2-3-4

On fixe dans cette partie $\mathcal{S}$ un ensemble totalement ordonnée.

4.1 Définitions

On définit par induction l'ensemble $\mathscr{A}_{2,3,4}(\mathcal{S})$ à partir des règles de construction suivantes :

- $V\Big|_{\{-\}}^{0}$
- $N^2\Big|_{\mathcal{S}}^{2}$
- $N^3\Big|_{\mathcal{S}^{\leq}}^{3}$
- $N^2\Big|_{\mathcal{S}^{\leq\leq}}^{4}$

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

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

- Les noeuds comme des chemins / mots sur $\Sigma = \{0, 1, 2, 3\}$;
- ch (en exercice (7)) ;
- la hauteur (par induction ou par hauteur maximale d'un noeud) ;
- les feuilles ;
- la taille (le nombre de noeuds) ;
- la profondeur ;
- la notion d'arbre "de recherche".

On ajoute la notion de 2-noeud ($N^2(...)$), de 3-noeud ($N^3(...)$) et de 4-noeud($N^4(...)$).

Un arbre de $\mathscr{A}_{2,3,4}(\mathcal{S})$ est dit **parfaitement équilibré** si et seulement si toutes ses feuilles ont la même profondeur. On appelle alors **arbre 2-3-4** tout arbre de recherche de $\mathscr{A}_{2,3,4}(\mathcal{S})$ parfaitement équilibré. En généralisant les propriétés sur les arbres binaires de recherches, un arbre 2-3-4 a alors une hauteur logarithmique en son nombre d'éléments.

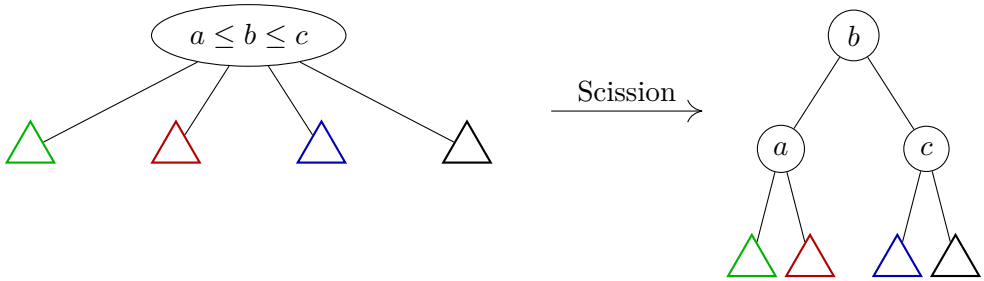
L'intérêt des arbres 2-3-4 réside dans la possibilité de "stocker" plusieurs clés dans un noeud, permettant de conserver (temporairement) la structure d'arbre 2-3-4. Une fois que 3 clés sont stockées dans un noeud, il est possible de "convertir" le 4-noeud de manière à obtenir des 2-noeud / des trois noeuds sans perdre la structure d'arbre 2-3-4.

4.2 Scission d'un 4-noeud

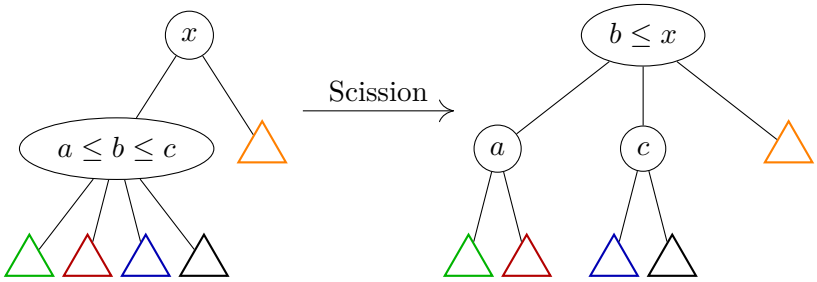
La scission d'un 4-noeud n de clés $a \leq b \leq c$ consiste à le transformer en deux 2-noeuds de clés a et c et à faire remonter b dans le père de n :

- si n n'a pas de père, alors un 2-noeud est créé pour la clé b : la hauteur augmente de 1 ;
- si le père de n est un 2-noeud, il devient un 3-noeud, et si c'est un 3-noeud, il devient un 4-noeud ;
- si le père de n est un 4-noeud, l'opération est impossible (il faut préalablement effectuer la scission du père) ;

Dans tous les cas, l'arbre reste équilibré et de recherche.



Scission d'un 4-noeud racine. Les triangles représentent des sous-arbres : le vert contient les éléments x vérifiant $x \leq a$, le rouge $a < x \leq b$, le bleu $b < x \leq c$, et le noir $c < x$.



Scission d'un 4-noeud fils d'un 2-noeud Les triangles représentent des sous-arbres avec les mêmes conventions que précédemment (le triangle noir contient des éléments entre c exclu et x inclus, et le triangle orange est sous-arbre avec des éléments strictement supérieurs à x).

Exercice 8: Représenter l'opération de scission d'un 4-noeud fils droit d'un 2-noeud, et d'un 4-noeud fils d'un 3-noeud (à gauche, au milieu ou à droite).

4.3 Insertion dans un arbre 2-3-4

Pour insérer une clé $e \in \mathcal{S}$ dans t un arbre 2-3-4 de racine n, on procède comme suit :

- si n est un 4-neoud, alors on scinde n puis on insère e dans le sous-arbre adapté de l'arbre obtenu ;
- sinon, si n est une feuille, on ajoute la clé e dans le noeud n ;
- sinon, on insère e dans le sous-arbre adapté à l'arbre obtenu.

Ces opérations conservent l'équilibre parfait et le caractère de recherche d'un arbre 2-3-4.

5 Arbres rouge-noir

5.1 Définition

Un arbre rouge-noir (ARN) est un arbre binaire de recherche dans lequel les noeuds ont une couleur (rouge ou noir) et :

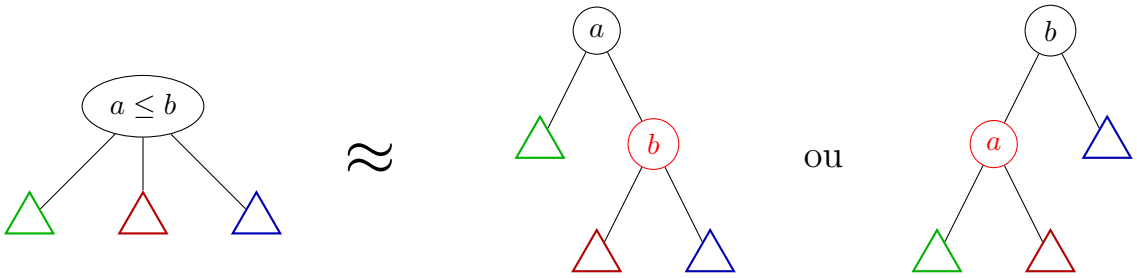
- la racine est noire ;
- chaque noeud rouge a 2 fils noirs ou 2 fils vides ;
- chaque branche contient exactement le même nombre de noeuds noirs que les autres.

On pourra stocker la couleur dans l'étiquette.

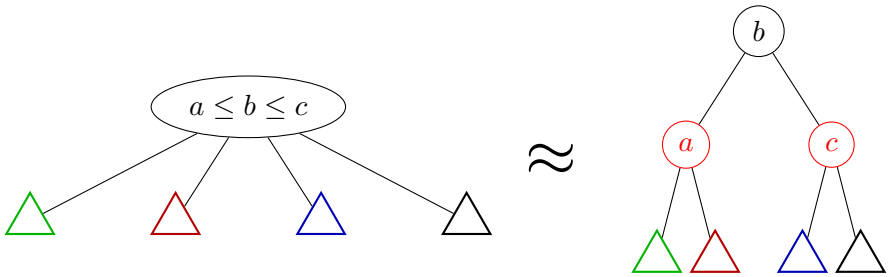
Remarque : La hauteur d'un ARN est logarithmique en le nombre de noeuds.

Remarque : Ces arbres permettent de présenter les arbres 2-3-4 sous formes d'arbres binaires : on a en effet la correspondance :

- pour 2-noeuds, ce sont les noeuds habituels ;
- pour les 3-noeuds, 2 types possibles :



- pour les 4-noeuds :



Remarque : L'opération sur les ARN passant d'une représentation du 3-noeud à l'autre s'appelle rotation simple entre les noeuds a et b (ici abusivement identifiés par leur étiquette).

Remarque : Il est nécessaire d'avoir les noeuds a et c rouges dans la représentation du 4-noeud pour conserver le nombre de noeuds noirs entre les branches. Le passage à l'ARN où a et c sont noirs correspond à la scission du 4-noeud racine.

On peut alors définir par analogie avec arbres 2-3-3 les opérations de scission de 4-noeuds et insertion (en exercice (9)).

5.2 Suppression dans un ARN

L'algorithme de suppression dans un ABR peut être appliqué à un ARN, mais il peut perturber les conditions sur la couleur de la racine, les fils d'un noeud rouge ou le nombre de nouds noirs dans une branche. Il faut donc le "réparer" après. Notons que cela n'est nécessaire que quand le noeud supprimé x était noir (et qu'il n'était pas racine). On peut ensuite appliquer l'algorithme suivant (où x est le noeud supprimé) :

Algorithme 4 : Réparation post-suppression dans un ARN

Entrées : t' un ARN auquel on a appliqué l'algorithme de suppression dans un ABR, obtenant un ABR t x un noeud de t' (qu'on a éventuellement supprimé dans t)

Sorties : Modifie t à partir de x en t'' un ARN contenant le même nombre de noeuds étiquettés par e que t pour tout $e \in E$.

```
1  si x est rouge alors
2  |  Rendre x noir.
3  sinon
4  |  si x n'est pas racine (x ≠ ε) alors
5  |  |  p ← père de x ;
6  |  |  f ← frère de x (autre fils de p) ;
7  |  |  si f est rouge alors
8  |  |  |  effectuer la rotation p − f ;
9  |  |  |  rendre p rouge ;
10 |  |  |  rendre f noir f ← nouveau frère de x (qui sera nécessairement noir car fils de f rouge).
11 |  |  fin
12 |  |  si f est noir avec deux fils noirs ou 2 fils vides alors
13 |  |  |  rendre f rouge ;
14 |  |  |  réparer t à partir de p
15 |  |  sinon
16 |  |  |  si f est un fils noir avec 1 fils noir et un fils rouge y tel que x est le fils du même côté de p
   |  |  |   que le côté y en tant que fils de f alors
17 |  |  |  |  effectuer la rotation y − f ;
18 |  |  |  |  rendre y noir ;
19 |  |  |  |  rendre f rouge ;
20 |  |  |  |  f ← nouveau frère de x (i.e f ← y)
21 |  |  |  fin
22 |  |  |  si f est noir avec un fils rouge z tel que x n'est pas le fils du même côté de p que le côté de
   |  |  |   z en tant que fils de f alors
23 |  |  |  |  effectuer la rotation p − w ;
24 |  |  |  |  rendre z noir
25 |  |  |  fin
26 |  |  fin
27 |  fin
28 fin
```

Remarque : Cet algorithme est assez difficile à appréhender. Une manière de le comprendre est de considérer qu'après la suppression d'un noeud noir, l'équilibre du nombre de noeuds noirs entre les branches de l'ARN est perturbé : toutes les branches qui passent par le noeud supprimé manquent un noeud noir, ce que l'on modélise par une "décharge" noire portée par ledit noeud. On peut alors éliminer cette décharge de deux manières : en la faisant remonter jusqu'à la racine ou en l'éliminant (l.16-25 ou 22-25).

Remarque : Les cas traités par l'algorithme de réparation sont tous "symétriques" (on ne regarde pas la valeur des étiquettes, on n'a donc aucune raison de s'intéresser au fils gauche ou au fils droit en particulier).

Remarque : Le plus difficile est d'établir les différents cas à traiter et de discriminer les cas impossibles. Pour cela, on commencer par représenter tous les cas de déséquilibre possibles pour la section de l'arbre contenant x, son père et son frère, sans s'intéresser à la couleur de x (ni le fait qu'il soit fils droit ou fils gauche), et atteignables en supprimant un noeud noir : on obtient 3 configurations. La configuration où le

frère est rouge est facile à traiter, les deux autres (père rouge et frère noir, père noir et frère noir) sont plus retorses.

Exercice 10: Représenter chacun des cas rencontrés par l'algorithme et leur évolution durant son exécution.

6 Hachage

Les ABR et ARN nous ont permis d'implémenter le type abstrait ensemble / multi-ensemble / dictionnaire avec une complexité de recherche / suppression / insertion en temps logarithmique en le nombre d'éléments stockés. Les éléments étaient positionnés dans la structure en fonction de la valeur de leur clé relativement aux autres éléments dans l'arbre. Dans cette section, on envisage de positioner les éléments dans la structure en fontion 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 dans la structure.

On suppose que l'on cherche à implémenter un ensemble dont les clés sont à valeurs dans $\mathcal{C}$. On se propose de stocker les éléments (ou du moins l'ensemble des données) dans un tableau T de taille m, via une fonction de $\mathcal{C} \to [\![0, m[\![$, de sorte qu'un élément $c \in \mathcal{C}$ soit stocké dans la case $T[f(c)]$.

Une "bonne" fonction de hachage :

- est déterministe (la même clé donne la même case) ;
- est efficace (opérations en temps constant)
- doit répartir au mieux les différentes clés

Pour le dernier point, on peut en particulier chercher à minimiser l'une des quantités suivantes :

- $\max_{i \in [\![0,m[\![} \text{card}(f^{-1}(i))$ (le nombre maximal de collisions dans une même case) ;
- $\sum_{c \in \mathcal{C}} \text{card}(f^{-1}(\{f(c)\}) \times \mathbb{P}(c)$ (nombre moyen de collisions, connaissant la distribution des clés modélisée par une probabilité $\mathbb{P}$ sur $\mathcal{C}$) ;
- $\frac{1}{m} \sum_{i=0}^{m-1} \text{card}(f^{-1}(i))$ (nombre moyen de collisions dans le cas d'une distribution uniforme).

7 Implémentation des tas

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 noeuds.

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 $\theta(n)$.

Soit $t \in \mathscr{A}_B(E)$ où E est totalement ordonné. On dit que t est un **arbre tournoi** si et seulement si $\forall (n, m) \in \text{ch}(t)^2, m$ est fils de $n \Rightarrow \text{etiq}(t)(m) \geq \text{etiq}(t)(n)$

Remarque : On obtient une définition équivalente en remplaçant "fils" par "descendant".

7.1 Opérations en pseudo-code

L'accès au minimum se fait en temps constant :

Algorithme 5 : Minimum d'un tas

Entrées : $t \in \mathscr{A}_B(E)$ un arbre binaire complet tournoi (ABCT) à étiquettes dans E
Sorties : Le plus petit élément de E dans t

```
1 Renvoyer l'étiquette de la racine de t
```

Pour l'insertion, on "trie" les noeuds par profondeur puis par valeur en base 2 (cela correspond à compter

les noeuds dans le sens de lecture, c'est à dire de gauche à droite puis de haut en bas).

Algorithme 6 : Insertion dans un tas

Entrées : $t \in \mathscr{A}_B(E)$ un ABPCT à étiquettes dans E, $e \in E$ l'élément à insérer dans t.
Sorties : t' un ABPT vérifiant

$$\mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = e\}) = \mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = e\}) + 1$$

$$\forall x \in E \setminus \{e\},\ \mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = x\}) = \mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = x\})$$

1 Trouver p le premier noeud ayant un fils vide ;
2 Ajouter un noeud n d'étiquette e en fils du père à la place du fils vide (ou d'un des fils vides s'il y en a deux) ;
3 **tant que** n a un père ***et*** l'étiquette de n a une priorité plus petite que celle de son père p **faire**
4 Échanger les étiquettes de n et p ;
5 $n \leftarrow p$
6 **fin**

Remarque : L'opération de recherche du premier noeud ayant un fils vide peut paraître coûteuse, mais elle peut se faire en temps constant en stockant dans la structure de tas la position de ce noeud. Si on ne fait que les opérations usuelles, sa mise à jour se fera aussi en temps constant. Cette astuce peut aussi être utilisée pour repérer le dernier noeud dans l'algorithme de suppression suivant :

Algorithme 7 : Suppression du minimum

Entrées : $t \in \mathscr{A}_B(E)$ un ABCT non vide
Sorties : En notant $e = \min\limits_{c \in \mathrm{ch}(t)} \{(\mathrm{etiq}(t))\,(c)\}$, modifie t en un ABCT t' vérifiant :

$$\mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = e\}) = \mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = e\}) - 1$$

$$\forall x \in E \setminus \{e\},\ \mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = x\}) = \mathrm{card}(\{c \in \mathrm{ch}(t') \mid (\mathrm{etiq}(t'))\,(c) = x\})$$

1 Soit n le dernier noeud de t ; on note v sa valeur ;
2 Remplacer n par Vide dans t ;
3 Remplacer l'étiquette de la racine par v ;
4 $n \leftarrow$ racine de t ;
5 **tant que** la priorité de l'étiquette n est strictement supérieure à la priorité de l'étiquette de l'un de ses fils **faire**
6 Échanger l'étiquette de n et celle de son fils ayant l'étiquette de priorité minimale entre les deux fils ;
7 $n \leftarrow$ le fils avec lequel on a fait l'échange
8 **fin**

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

- par tableau, en retenant la première case vide tu tableau ;
- avec des structures représentant les noeuds et un pointeur vers le champ correspondant au premier noeud vide.

Remarque : Dans le cadre d'une implémentation par tableau, il est possible de créer un tas à partir d'un tableau non trié en temps linéaire : pour cela, on descend la racine du tas (c'est-à-dire que l'on échange la valeur de la racine avec celle de l'enfant ayant la plus petite priorité si celle-ci est plus petite que celle du noeud de la racine, puis on opère récursivement sur le fils avec lequel on a fait l'échange jusqu'à arriver en bas ou ne plus faire d'échange) engendré par chaque sous-noeud de l'avant-dernier niveau du tas (2 fois au besoin, une pour chaque fils), puis on fait la même chose sur le niveau au-dessus, et ainsi de suite jusqu'à la racine.

Exercice 11: Proposer une implémentation en C d'un tas comme tableau (où les valeurs du tableau sont des entiers représentant la priorité de chaque noeud) avec les opérations d'insertion, suppression et accès au minimum (on pourra utiliser des tableaux dynamiques).

Exercice 12: Ajouter à l'implémentation précédente une fonction `void descend_racine(Tas tas, int position)` qui implémente la descente de racine dans le sous-tas engendré par le noeud en position `position` dans `tas`. Implémenter alors la fonction `Tas cree_tas(int taille, int *elements)` qui crée un tas à partir d'un tableau d'entiers non ordonné.

Exercice 13: Justifier que l'implémentation de la remarque de la création du tas s'exécute bien en temps linéaire.