Arbres bouc-émissaires

Dans ce sujet on s'intéresse à une variante des arbres binaires de recherche (ABR) appelée les ***arbres bouc-émissaires***. Étant donné que le temps d'exécution des principales opérations sur les ABR (recherche, ajout et suppression d'un nœud) dépend principalement de la hauteur de l'arbre, il est intéressant de maintenir cette hauteur aussi petite que possible. Les arbres bouc-émissaires permettent de faire cela en exploitant une idée assez simple : lorsque la hauteur de l'arbre devient trop grande, on reconstruit partiellement, voir complètement l'arbre.

Lien vers l'article original : `http://people.csail.mit.edu/rivest/pubs/GR93.pdf`

1 Préliminaires

Pour rappel, en OCaml vous pouvez utiliser les fonctions « `Array.make: int -> 'a -> 'a array` » et « `Array.length: 'a array -> int` ». L'expression « `Array.make n x` » s'évalue en un tableau de taille `n` dont toutes les cases contiennent `x`. L'expression « `Array.length tab` » s'évalue en la taille de `tab`.

1.1 Définitions

Arbres bouc-émissaires. Tous les arbres qui nous intéressent ici sont des arbres binaires définis en OCaml par le type « `'a arbre` ». Étant donné un arbre, on appellera ***taille*** son nombre de nœuds. Afin d'implémenter les différentes opérations sur les arbres bouc-émissaires, il sera utile de pouvoir accéder en temps constant aux tailles de l'arbre et de ses sous-arbres. Ainsi, chaque nœud x est étiqueté par un couple d'entiers (c, n) où c est la ***clé*** stockée dans ce nœud et n est la taille du sous-arbre enraciné en x. Les arbres bouc-émissaires sont donc représentés par le type `abe`.

```
type 'a arbre =
  | Vide
  | N of 'a * 'a arbre * 'a arbre;;
```

```
type abe = (int * int) arbre;;
```

Voici un exemple d'arbre bouc-émissaire et sa représentation en OCaml :

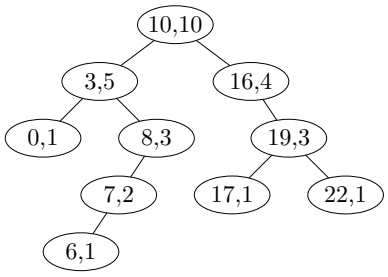

FIGURE 1

```
N((10,10),
  N((3,5),
    N((0,1),Vide,Vide),
    N((8,3),
      N((7,2),
        N((6,1),Vide,Vide),Vide),Vide)),
  N((16,4),Vide,
    N((19,3),
      N((17,1),Vide,Vide),
      N((22,1),Vide,Vide))));;
```

En résumé, tout objet de type `abe` doit vérifier deux propriétés :

- Les clés de l'arbre sont distinctes deux à deux et organisées comme dans un ABR : pour tout nœud x, une clé stockée dans le sous-arbre gauche (resp. droit) de x est nécessairement strictement inférieure (resp. strictement supérieure) à la clé stockée dans x.
- Le second élément de l'étiquette d'un nœud x est égal à la taille du sous-arbre enraciné en x.

Ainsi, lorsqu'une fonction prend en entrée un objet de type `abe`, vous pouvez supposer qu'il vérifie les propriétés ci-dessus. De plus, lorsqu'une fonction renvoie un objet de type `abe`, vous devez faire en sorte que ces propriétés soient vérifiées.

Arbres α-équilibrés en poids. Soit $\alpha \in [\frac{1}{2}\,;1[$ un réel. Un nœud x est dit ***α-équilibré en poids*** si :

$$n_G \leqslant \alpha\, n \qquad \text{et} \qquad n_D \leqslant \alpha\, n$$

où n est la taille de l'arbre enraciné en x, n_G est la taille du sous-arbre gauche de x et n_D est la taille du sous-arbre droit de x. Considérons dans l'exemple de la figure 1 le nœud x étiqueté par $(3, 5)$. On a alors $n = 5$, $n_G = 1$, $n_D = 3$, et donc x est α-équilibré pour $\alpha \in [\frac{3}{5}\,;1[$, mais pas pour $\alpha \in [\frac{1}{2}\,;\frac{3}{5}[$.

Un arbre est dit ***α-équilibré en poids*** si tout nœud de cet arbre est α-équilibré en poids.

Un arbre est dit ***α-déséquilibré en poids*** s'il n'est pas α-équilibré en poids.

Arbres α-équilibrés en hauteur. Pour $\alpha \in [\frac{1}{2}\,;1[$, on note $h_\alpha : \mathbb{N}^* \to \mathbb{N}$ la fonction définie par :

$$h_\alpha(n) = \left\lfloor \log_{1/\alpha}(n) \right\rfloor$$

où $\log_{1/\alpha}(n)$ est le logarithme en base $1/\alpha$ de n.

Un arbre est dit ***α-équilibré en hauteur*** s'il est vide ou bien si sa taille n et sa hauteur h vérifient $h \leqslant h_\alpha(n)$.
Un arbre est dit ***α-déséquilibré en hauteur*** s'il n'est pas α-équilibré en hauteur.

1.2 Quelques fonctions utiles

Le but de cette partie est d'implémenter des fonctions que vous pourrez réutiliser librement dans la suite.

1. Écrire une fonction « `get_taille : abe -> int` » qui renvoie la taille de l'arbre bouc-émissaire donné en entrée. Votre fonction devra s'exécuter en temps constant.
2. (a) Écrire une fonction « `arbre_of_abe : abe -> int arbre` » qui prend en entrée un arbre bouc-émissaire A et renvoie A dans lequel les tailles ont été supprimées des étiquettes (vous devez donc remplacer le couple (c, n) par c). Le temps d'exécution devra être linéaire en la taille de l'arbre.
   (b) Écrire une fonction « `abe_of_arbre : int arbre -> abe` » qui prend entrée un arbre A et renvoie l'arbre bouc-émissaire ayant la même forme que A et contenant les mêmes clés. En d'autres termes, votre fonction doit calculer les tailles des sous-arbres et les ajouter dans les étiquettes de A. Le temps d'exécution devra être linéaire en la taille de l'arbre.

Afin de rechercher si un nœud est présent dans un arbre bouc-émissaire, il suffit de procéder comme pour un ABR habituel en ignorant les tailles présentes dans les étiquettes.

3. Expliquer comment rechercher si une clé est présente dans un ABR.

Plus tard dans le sujet, il sera utile de connaître la profondeur d'une clé dans un arbre bouc-émissaire. Pour cela, on peut compter le nombre d'étapes nécessaires à la recherche de la clé dans l'arbre.

4. (a) Écrire une fonction « `get_profondeur : int -> abe -> int` » qui renvoie la profondeur d'une clé dans un arbre bouc-émissaire. Si la clé n'est pas présente, votre fonction déclenchera une exception.
   (b) Donner en le justifiant le temps d'exécution de votre fonction.

2 Arbres $\frac{1}{2}$-équilibrés en poids

Dans cette partie, on s'intéresse aux arbres α-équilibrés en poids dans le cas particulier où $\alpha = \frac{1}{2}$. Notez qu'un tel arbre est β-équilibré en poids pour tout $\beta \in [\frac{1}{2}\,;1[$.

5. (a) Soit A un arbre non vide $\frac{1}{2}$-équilibré en poids, soit x un nœud de A et $n \geqslant 1$ la taille du sous-arbre enraciné en x. En fonction de n, quelles sont les valeurs possibles pour les tailles des sous-arbres gauche et droit de x ?
   (b) Écrire une fonction « `abe_of_array : int array -> abe` » qui renvoie un arbre bouc-émissaire $\frac{1}{2}$-équilibré en poids dont les clés sont les éléments du tableau `tab` donné en entrée. Les éléments de `tab` sont supposés distincts deux à deux et rangés par ordre croissant. Le temps d'exécution devra être linéaire en la taille du tableau.
   (c) Écrire une fonction « `equilibrer : abe -> abe` » qui prend en entrée un arbre A, et renvoie un arbre $\frac{1}{2}$-équilibré en poids contenant les mêmes clés que A. Le temps d'exécution devra être linéaire en la taille de l'arbre.
6. Montrer que si un arbre est $\frac{1}{2}$-équilibré en poids, aucun arbre de la même taille n'est de hauteur strictement inférieure.

3 Insertion dans un arbre bouc-émissaire

À partir de maintenant, on fixe un réel $\alpha \in]\frac{1}{2}\,;1[$ (donc $\alpha \neq \frac{1}{2}$). Les arbres bouc-émissaires manipulés seront α-équilibrés en hauteur. Notre but est de conserver cette propriété lors de l'ajout ou de la suppression d'une clé.

Soit A_0 un ABR de taille $n-1$ et A_1 l'ABR obtenu après insertion d'un nouveau nœud x dans A_0 (la procédure d'insertion est la procédure habituelle vue en cours). Si A_0 est α-équilibré en hauteur, mais que A_1 ne l'est pas, alors c'est que x est à profondeur strictement supérieure à $h_\alpha(n)$. Dans ce cas, on reconstruit en partie l'arbre afin d'en diminuer la hauteur. Pour cela, on détermine un nœud α-déséquilibré en poids noté y et on remplace l'arbre enraciné en y par un arbre $\frac{1}{2}$-équilibré en poids à l'aide de la fonction `equilibrer`. Le nœud y est appelé le ***nœud bouc-émissaire***.

7. (a) Soit A un arbre α-déséquilibré en hauteur de taille n et x un nœud dont la profondeur est strictement supérieure à $h_\alpha(n)$. Montrer que l'un des ancêtres de x est α-déséquilibré en poids. On rappelle qu'un ***ancêtre*** est un nœud qui se trouve sur le chemin reliant x à la racine de l'arbre.

   (b) Soit A un arbre non vide α-équilibré en poids de taille n et de hauteur h. Montrer que $h = \mathcal{O}\Big(\log_2(n)\Big)$.

Dans la suite de cette partie, nous allons implémenter la procédure d'insertion (question 8), puis montrer que l'arbre obtenu en sortie est α-équilibré en hauteur (question 10) et enfin analyser le temps d'exécution (question 12).

Implémentation de la procédure d'insertion. Nous sommes maintenant prêts à ajouter une clé c dans un arbre bouc-émissaire. Soit A_0 un arbre α-équilibré en hauteur de taille $n-1$:

- On commence par insérer c dans A_0 comme dans un ABR standard. Soit A_1 l'arbre obtenu et x le nœud de A_1 dont la clé est c.
- On vérifie si A_1 est α-équilibré en hauteur en comparant la profondeur de x avec $h_\alpha(n)$, puis :
  - Si A_1 est α-équilibré en hauteur, c'est la fin de la procédure d'insertion.
  - Sinon, on parcourt les noeuds de A_1 en direction de x jusqu'à rencontrer un nœud y qui n'est pas α-équilibré en poids. On remplace alors le sous-arbre enraciné en y par un arbre $\frac{1}{2}$-équilibré en poids. Dans la suite, on notera A_2 l'arbre ainsi obtenu.

8. (a) Écrire une fonction « `ajout_abr : int -> abe -> abe` » qui prend en entrée c ainsi que A_0 et renvoie A_1. Dans le cas où c est déjà présente dans A_0, votre fonction déclenchera une erreur.

   (b) Dans cette question, on suppose que l'arbre A_1 n'est pas α-équilibré en hauteur.
   Écrire une fonction « `equilibrer_be : int -> abe -> float -> abe` » qui prend en entrée c, A_1 ainsi que α, et renvoie l'arbre A_2.

   (c) Écrire une fonction « `ajout_abe : int -> abe -> float -> abe` » qui prend en entrée c, A_0 ainsi que α et renvoie l'arbre obtenu après insertion de la clé c dans A_0.

Correction de la procédure d'insertion. Il reste à montrer que si l'arbre A_0 est α-équilibré en hauteur alors l'arbre obtenu après l'insertion est également α-équilibré en hauteur.

9. Soit A un arbre dont la racine est α-déséquilibrée en poids.

   (a) Montrer que le sous-arbre gauche de A contient au moins deux nœuds de plus que le sous-arbre droit (ou inversement).

   (b) Supposons que A ne contienne qu'un seul nœud à profondeur h où h est la hauteur de A. On note A' l'arbre obtenu après application de la fonction `equilibrer` sur l'arbre A. Montrer que la hauteur de A' est strictement inférieure à la hauteur de A.

10. Montrer que si l'arbre A_0 est α-équilibré en hauteur alors l'arbre obtenu après une insertion est également α-équilibré en hauteur.

Complexité de la procédure d'insertion. Lors de la procédure d'insertion, il peut arriver que l'arbre soit reconstruit entièrement. La complexité d'une insertion peut donc être linéaire en la taille de l'arbre. Toutefois, lors d'une suite de plusieurs insertions, le fait de reconstruire l'arbre entièrement est un phénomène assez rare. On va montrer que le temps nécessaire pour effectuer n insertions dans un arbre initialement vide est en $\mathcal{O}\Big(n\log_2(n)\Big)$. Ainsi chaque insertion aura pris en moyenne un temps $\mathcal{O}\Big(\log_2(n)\Big)$, on parle de ***complexité amortie***.

Soit B_0 l'arbre vide et $c_1, \dots c_n$ des clés que l'on va insérer successivement dans cet arbre. Pour $i \in [\![1; n]\!]$, on note B_i l'arbre obtenu après insertion de la clé c_i dans l'arbre B_{i-1} à l'aide de la fonction `ajout_abe`. Cette étape sera appelée la "$i^{\text{ème}}$ insertion" et on note T_i son temps d'exécution. Notez que la question 10 assure que les arbres B_i sont tous α-équilibrés en hauteur.

Soit A un arbre, x un nœud de A, n_G le nombre de nœuds dans le sous-arbre droit de x et n_D le nombre de nœuds dans le sous-arbre gauche de x. On définit l'entier $\Delta_A(x)$ appelé le ***potentiel*** de x dans A par :

$$\Delta_A(x) = \begin{cases} 0 & \text{si } |n_G - n_D| \leqslant 1, \\ |n_G - n_D| & \text{sinon.} \end{cases}$$

On note également $\Delta(A)$ la somme des $\Delta(x)$ où x parcourt tous les nœuds de A.
Soit $k_0 > 0$ une constante quelconque (sa valeur sera fixée plus tard). Le ***coût amorti*** de l'insertion numéro i noté C_{i,k_0} est définie par :

$$C_{i,k_0} = T_i + k_0\Big(\Delta(B_i) - \Delta(B_{i-1})\Big)$$

11. Soit x la racine d'un arbre A de taille n. On suppose que x n'est pas α-équilibré en poids. Montrer que :

$$\Delta_A(x) > (2\alpha - 1)n + 1$$

12. (a) Supposons que la $i^{\text{ème}}$ insertion ne nécessite pas d'appel à la fonction `equilibrer_be`. Montrer que quelle que soit la valeur de $k_0 > 0$: $C_{i,k_0} = \mathcal{O}\Big(\log_2(i)\Big)$.

    (b) Supposons que la $i^{\text{ème}}$ insertion nécessite un appel à la fonction `equilibrer_be`. Montrer qu'il existe une valeur de $k_0 > 0$, pour laquelle il existe une constante $k_1 > 0$ telle que : $C_{i,k_0} \leqslant k_1 \times \log_2(i)$.

    (c) Conclure que $\displaystyle\sum_{i=1}^{n} T_i = \mathcal{O}\Big(n \log_2(n)\Big)$.

4 Suppression dans un arbre bouc-émissaire

Lors de la suppression d'une étiquette dans un arbre bouc-émissaire, il faudra parfois reconstruire entièrement l'arbre à l'aide de la fonction `equilibrer`. Dans la suite, le fait de reconstruire entièrement l'arbre lors d'une suppression sera appelé une ***réinitialisation***.

Avant de décrire la procédure de suppression, il faut ajouter une information dans la structure de donnée : la taille maximale qu'a eu l'arbre depuis la dernière réinitialisation. Cette information sera stockée dans un champ `taille_max`.

```
type abe2 = {
    arbre: abe;
    taille_max: int
  };;
```

13. Écrire une fonction « `ajout_abe2 : int -> abe2 -> float -> abe2` » qui prend en entrée une clé c, un arbre A de type `abe2` ainsi que le paramètre α et qui ajoute la clé c dans A en suivant la procédure de la partie 3.

Voici les étapes pour supprimer un nœud dans un arbre bouc-émissaire :

- On supprime d'abord le nœud comme dans un ABR standard (avec l'algorithme vu en cours).
- Soit $n_{\max}$ la valeur contenu dans le champ `taille_max`.
  - Si $n \geqslant \alpha\, n_{\max}$, on ne fait rien.
  - Sinon, on remplace l'arbre par un arbre $\frac{1}{2}$-équilibré en hauteur et on réinitialise le champ `taille_max`.

14. Écrire une fonction « `suppression_abe2 : int -> abe2 -> float -> abe2` » qui prend en entrée une clé c, un arbre A de type `abe2` ainsi que le paramètre α, et qui supprime la clé c dans A en suivant la procédure décrite ci-dessus. Si c n'apparaît pas dans l'arbre, votre fonction déclenchera une erreur.

Remarque. Considérons un arbre vide sur lequel on effectue une suite de n opérations (une opération est soit une insertion, soit une suppression), alors en utilisant le même genre d'arguments que dans la partie 3, on peut montrer que :

- Le temps d'exécution de ces n opérations est en $\mathcal{O}\Big(n \log_2(n)\Big)$.
- La hauteur h et la taille n de l'arbre final vérifient $h \leqslant h_\alpha(n) + 1$.