Brown tree

On rappelle que la taille $|t|$ d'un arbre binaire t est définie comme son nombre de nœuds non vides. Si $\mathcal{E}$ est un ensemble d'étiquettes, on définit par induction structurelle l'ensemble $\mathcal{B}(\mathcal{E})$ des *arbres de Braun* étiquetés par $\mathcal{E}$ de la manière suivante :

— L'arbre vide $\perp$ est un arbre de Braun.

— Si g et d sont des arbres de Braun tels que $|d| \leqslant |g| \leqslant |d| + 1$ et $x \in \mathcal{E}$, alors $\mathcal{N}(x, g, d)$ est un arbre de Braun.

La seconde partie, mettant un place un algorithme efficace du calcul de la taille d'un arbre de Brown, est indépendante de la troisième partie.

1 Propriétés élémentaires

1. Montrer, en ignorant les étiquettes, qu'il existe un unique arbre de Braun de taille n pour tout entier $n \in \mathbb{N}$.
2. Dessiner la forme des arbres de Braun de taille 1 à 6.
3. On définit la hauteur d'un arbre binaire de la manière usuelle, en posant notamment $h(\perp) := -1$. Montrer que si t est un arbre de Braun de taille $n \geqslant 1$, alors $h(t) = \lfloor \log_2 n \rfloor$.

Pour la programmation, on choisit de se limiter aux arbres de Braun à étiquettes entières. On utilise donc le type suivant :

```
type braun =
  | E
  | N of int * braun * braun
```

4. Écrire une fonction `height` calculant la hauteur d'un arbre de Braun en temps logarithmique en la taille de l'arbre.

```
height : braun -> int
```

2 Calcul de la taille

L'objectif de cette partie est d'écrire une fonction permettant de calculer la taille $n = |t|$ d'un arbre de Braun en temps $o(n)$.

1. Si l'on sait que t est un arbre de Braun vérifiant $2k \leqslant |t| \leqslant 2k + 1$, où $k \geqslant 1$, quelles sont les tailles possibles pour son sous-arbre gauche et pour son sous-arbre droit ?
2. Même question si t vérifie $2k + 1 \leqslant |t| \leqslant 2k + 2$.
3. En déduire une fonction `diff` ayant la spécification suivante :

   Entrées : un arbre de Braun t et un entier n.

   Précondition : $n \leqslant |t| \leqslant n + 1$.

   Sortie : $|t| - n$, qui sera donc égal à 0 ou 1 suivant les cas.

   Cette fonction devra avoir une complexité en $O(h)$.

```
diff : braun -> int -> int
```

4. Écrire à présent une fonction `size` calculant de manière efficace la taille d'un arbre de Braun.
5. Déterminer la complexité de la fonction `size`.

3 Réalisation d'un tas fonctionnel par un arbre de Braun

On appelle *tas de Braun* un arbre de Braun vérifiant la condition d'ordre des tas : l'étiquette d'un nœud est toujours inférieure ou égale à celles de ses fils éventuels. On souhaite programmer, de manière efficace, les trois opérations élémentaires sur les tas :

```
get_min : braun -> int
insert : braun -> int -> braun
extract_min : braun -> int * braun
```

L'appel `extract_min t` renverra un couple (m, t') où m est le minimum de t et t' est un tas de Braun contenant les mêmes étiquettes que t moins une occurrence de m.

1. Écrire la fonction `get_min`. On renverra `max_int` si jamais l'arbre est vide.
2. Écrire la fonction `insert`. On exige une complexité logarithmique en la taille de l'arbre. *Attention, il faut bien sûr que l'arbre renvoyé soit un tas de Braun et donc en particulier un arbre de Braun.*
3. Supposons que l'on ait écrit une fonction `merge : braun -> braun -> braun` ayant le comportement suivant :
   — Elle prend en entrée deux tas de Braun g et d vérifiant $|d| \leqslant |g| \leqslant |d| + 1$.
   — Elle renvoie un tas de Braun contenant les éléments de g ainsi que ceux de d.
   Écrire alors une fonction `extract_min` ayant la spécification donnée plus haut.

Il nous reste à écrire cette fonction `merge`. On peut commencer par traiter les cas simples.

4. Comment programmer `merge g d` si `d` est vide ? si $\min g \leqslant \min d$?

Le cas délicat est donc celui où la racine de d est strictement plus petite que celle de g : on voudrait prendre la racine de d comme racine « globale », mais on ne peut pas diminuer le nombre d'éléments dans d, puisqu'on violerait alors la condition d'équilibre des arbres de Braun.

5. Écrire une fonction `extract_element` qui prend en entrée un tas de Braun non vide t et renvoie un couple (x, t') tel que :
   — x est un élément quelconque de t.
   — t' est un tas de Braun contenant les éléments de t moins une occurrence de x.

```
extract_element : braun -> int * braun
```

6. Écrire une fonction `replace_min` tel que l'appel `replace_min t x` renvoie un tas de Braun t' contenant les mêmes éléments que t, moins une occurrence du minimum de t, plus une occurrence de x.

```
replace_min : braun -> int -> braun
```

7. Écrire à présent la fonction `merge`.
8. Déterminer la complexité de `extract_min`.