Jean-Baptiste Bianquis et Lilian Besson Si besoin : mp2i@besson.link

ARBRES DE BRAUN

1 Définition et propriétés élémentaires

Définition XXVIII.1 – Arbres de Braun

On définit la taille $|t|$ d'un arbre binaire t comme son nombre de nœuds non vides. Un *arbre de Braun* est alors :

- soit l'arbre vide $\perp$;
- soit de la forme $N(x, g, d)$, où g et d sont des arbres de Braun tels que $|d| \leqslant |g| \leqslant |d| + 1$.

► **Question 1** Montrer que, en ignorant les étiquettes, il existe un unique arbre de Braun de taille n pour chaque entier $n \geqslant 0$.

► **Question 2** Dessiner la forme des arbres de Braun de taille 1 à 6.

► **Question 3** On définit la hauteur d'un arbre binaire de la manière usuelle (avec $h(\perp) = -1$). 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 (cela simplifiera légèrement l'écriture de certaines fonctions). On utilise donc le type suivant :

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

► **Question 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 d'un arbre de Braun en temps $o(n)$.

► **Question 5** Si l'on sait que t est un arbre de Braun vérifiant $2n \leqslant |t| \leqslant 2n + 1$ (avec $n \geqslant 1$), quelles sont les tailles possibles pour son sous-arbre gauche et pour son sous-arbre droit ?

► **Question 6** Même question si t vérifie $2n + 1 \leqslant |t| \leqslant 2n + 2$.

► **Question 7** 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 : $|t| - n$ (0 ou 1 suivant les cas, donc).
Cette fonction devra avoir une complexité en $O(h)$.

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

▶ **Question 8** Écrire à présent une fonction `size` calculant de manière efficace la taille d'un arbre de Braun.

▶ **Question 9** 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.

▶ **Question 10** Écrire la fonction `get_min`. On renverra `max_int` si jamais l'arbre est vide (ce sera pratique par la suite).

▶ **Question 11** 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).*

▶ **Question 12** Supposons que l'on ait écrit une fonction `merge : braun -> braun -> braun` ayant le comportement suivant :

- elle prend en entrée deux tas de Braun t et t' vérifiant $|t'| \leqslant |t| \leqslant |t'| + 1$;
- elle renvoie un tas de Braun contenant les élément de t plus ceux de t'.

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 :

▶ **Question 13** Que doit valoir `merge l r` si r est vide ? si $\min l \leqslant \min r$?

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

▶ **Question 14** 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
```

▶ **Question 15** 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
```

▶ **Question 16** Écrire à présent la fonction `merge`.

▶ **Question 17** Déterminer la complexité de `extract_min`.

Solutions

▸ **Question 1** Par récurrence forte sur n :

- pour $n = 0$, seul l'arbre vide convient ;
- si $n = 2k+1$, un arbre de Braun est forcément de la forme (g, d) avec $|g| = |d| = k$. Par hypothèse de récurrence, il existe exactement un arbre de Braun non étiqueté B_k de taille k, et (B_k, B_k) est donc l'unique arbre de Braun de taille $2k+1$;
- si $n = 2k+2$, on a nécessairement $|g| = k+1$ et $|d| = k$. On a alors de même (B_{k+1}, B_k) comme unique arbre de Braun de taille $2k+2$.

▸ **Question 2**

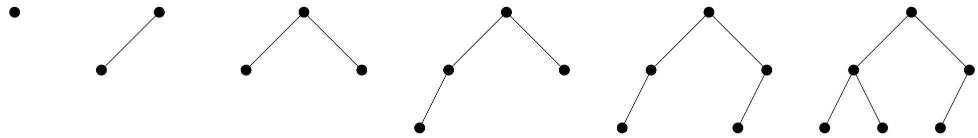

Figure XXVIII.1 – Arbres de Braun à $1, 2, \ldots, 6$ nœuds.

▸ **Question 3** On procède par récurrence forte sur n.

- Pour $n = 1$, on a bien $h(B_1) = 0 = \lfloor \log_2 1 \rfloor$.
- Si $n = 2k+1$ avec $k \geqslant 1$ alors $B_n = (B_k, B_k)$, donc $h(B_n) = 1 + h(B_k) = 1 + \lfloor \log_2 k \rfloor$ par hypothèse de récurrence. Or $1 + \lfloor \log_2 k \rfloor = \lfloor 1 + \log_2 k \rfloor = \lfloor \log_2(2k) \rfloor$. Il reste à remarquer que l'on a nécessairement $\lfloor \log_2(2k) \rfloor = \lfloor \log_2(2k+1) \rfloor$, puisque $2k+1$ ne peut pas être une puissance de 2. Finalement, on a donc bien $h(B_{2k+2}) = \lfloor \log_2(2k+1) \rfloor$.
- Si $n = 2k+2$ avec $k \geqslant 1$, alors $B_{2k+2} = (B_{k+1}, B_k)$, donc $h(B_{2k+1}) = 1 + \max(h(B_{k+1}), h(B_k)) = 1 + \lfloor \log_2(k+1) \rfloor$ par hypothèse de récurrence. Or $1 + \lfloor \log_2(k+1) \rfloor = \lfloor \log_2(2k+2) \rfloor$, ce qui achève la preuve.

▸ **Question 4** La hauteur d'un arbre de Braun est une fonction croissante de sa taille (d'après la question précédente), et le sous-arbre gauche a toujours une taille supérieure ou égale à celle du sous-arbre droit. On en déduit la fonction suivante :

```
let rec height = function
  | E -> -1
  | N (_, l, _) -> 1 + height l
```

Il est immédiat que la complexité est en $O(h(t))$, et donc en $O(\log |t|)$ d'après la question précédente.

▸ **Question 5**

Notons $t = (g, d)$ (t ne peut être vide puisque $n \geqslant 1$).

- Si $|t| = 2n$, alors $|g| = n$ et $|d| = n-1$.
- Si $|t| = 2n+1$, alors $|g| = n$ et $|d| = n$.

On a donc $|g| = n$ dans tous les cas, et $|d| \in \{n-1, n\}$.

▸ **Question 6** Cette fois, on a $|g| \in \{n, n+1\}$ et $|d| = n$.

▸ **Question 7** D'après les deux questions précédentes, la taille de l'un des sous-arbres est directement connue (le gauche ou le droit suivant la parité de n), et celle de l'autre est connue à un près et peut donc être calculée par un appel récursif. On obtient le code suivant :

```
let rec diff t n =
  match t, n with
  | E, 0 -> 0
  | N (_, E, E), 0 -> 1
  | N (_, l, r), _ ->
    if n mod 2 = 0 then diff r (n / 2 - 1)
    else diff l (n / 2)
  | _ -> failwith "impossible"
```

On effectue un travail constant à chaque niveau de l'arbre, la complexité est bien en $O(h) = O(\log n)$.

▸ **Question 8** On commence par calculer la taille n du sous-arbre droit par un appel récursif. On sait que celle du sous-arbre gauche vaut n ou $n+1$, on peut donc utiliser `diff` pour terminer le calcul.

```
let rec size t =
  match t with
  | E -> 0
  | N (_, l, r) ->
    let n = size r in
    2 * n + 1 + diff l n
```

▸ **Question 9** Notons $\varphi(h)$ le nombre d'opérations élémentaires (à une constante multiplicative près) effectuées par `size` sur un arbre de hauteur h. On a $\varphi(h+1) = 1 + \varphi(h) + \psi(h)$ où $\psi(h)$ est le nombre d'opérations faites par `diff` sur un arbre de hauteur h. Or $\psi(h) = h$, donc $\varphi(h+1) = h + 1 + \varphi(h)$. En sommant la relation $\varphi(h+1) - \varphi(h) = h+1$, on obtient $\varphi(h) - \varphi(0) = \frac{h(h+1)}{2}$, et la complexité est donc en $O\left(h^2\right) = O\left((\log n)^2\right)$.

▸ **Question 10** C'est immédiat :

```
let get_min = function
  | E -> max_int
  | N (x, _, _) -> x
```

▸ **Question 11** Dans le cas intéressant, on a $t = (y, g, d)$. Pour respecter la contrainte d'ordre des tas, il faudra prendre comme racine $m = \min(x, y)$ et insérer $M = \max(x, y)$ dans l'un des deux sous-arbres. Pour respecter la contrainte de structure des arbres de Braun, on choisit d'insérer dans d et d'inverser d et g. On a alors $t' = (m, \mathtt{insere}(d, M), g)$, et $|\mathtt{insere}(d, M)| = 1 + |d| \in \{|g|, |g| + 1\}$, ce qui est exactement ce que l'on veut.

```
let rec insert t x =
  match t with
  | E -> N (x, E, E)
  | N (y, l, r) ->
    if x < y then N (x, insert r y, l)
    else N (y, insert r x, l)
```

▸ **Question 12** Il suffit de faire :

```
let extract_min t =
  match t with
  | E -> failwith "empty queue"
  | N (x, l, r) -> x, merge l r
```

▸ **Question 13** Si r est vide, on renvoie bien sûr l. Si $\min l \leqslant \min r$ et que l n'est pas vide ($l = (x, ll, lr)$), alors la racine de $merge(l, r)$ sera x. On peut fusionner récursivement ll et lr et obtenir ainsi l' tel que $|l'| = |l| - 1$. De manière symétrique à `insert`, on peut alors conserver la structure d'arbre de Braun en faisant de l' le fils *droit* du nouvel arbre. Autrement dit, on a alors $merge(l, r) = (x, r, merge(ll, lr))$.

▸ **Question 14** On extrait un élément à gauche et l'on inverse fils gauche et fils droit.

```
let rec extract_element t =
  match t with
  | E -> failwith "extract_element from empty queue"
  | N (x, E, E) -> x, E
  | N (x, l, r) ->
    let y, l' = extract_element l in
    y, N (x, r, l')
```

▸ **Question 15** Pas de difficulté majeure ici, le nombre d'éléments de l'arbre ne change pas donc les contraintes de structure sont automatiquement respectées.

```
let rec replace_min t x =
  match t with
  | E -> failwith "nope"
  | N (y, l, r) ->
    if x <= get_min l && x <= get_min r then N (x, l, r)
    else if get_min l <= get_min r then
      N (get_min l, replace_min l x, r)
    else
      N (get_min r, l, replace_min r x)
```

▸ **Question 16** Tout le travail a été fait :

```
let rec merge l r =
  match l, r with
  | _, E -> l
  | N (lx, ll, lr), N (rx, rl, rr) ->
    if lx <= rx then
      N (lx, r, merge ll lr)
    else
      let x, l' = extract_element l in
      N (rx, replace_min r x, l')
  | _ -> failwith "merge"
```

▸ **Question 17** La complexité de `extract_min` est évidemment la même que celle de `merge`, et les complexités de `extract_element` et `replace_min` sont clairement en $O(h)$. Un appel à `merge` descend le long d'une branche de l'arbre en faisant un temps constant à chaque nœud (pour un coût $O(h)$ au total pour cette phase), jusqu'à arriver dans le cas `lx > lr`. À ce moment, on fait un appel à `extract_min` et un à `replace_min`, pour un coût $O(h)$ (et l'on arrête les appels récursifs à `merge`). Au total, on a donc du $O(h) = O(\log n)$.