File de priorité

Informatique MP2I

Victor ROBERT

1. Définitions

Définition

Une *file de priorité* est une structure de données abstraite.

Une file de priorité stocke un multiensemble (répetitions acceptées) d'élements totalement ordonnés.

Les données sont "ordonnées" par valeur, avec priorité au plus petit (ou grand) élement.

Remarque

On nommera d'une file de priorité qu'elle est *min* (resp. *max*) si elle priorise le plus petit (resp. grand)

Signature

- créer une file de priorité vide: $créer : () \to F$
- ajouter un élement: $ajouter : F \to x \to () \mid F$
- retirer le *max* ou le *min*: $retirer : F \to () \mid F$
- tester si c'est vide: $estVide : F \to \mathbb{B}$
- détruire la file: $détruire : F \to ()$

Exemple

```
let xs = [1; 10; 7; 5; 2; 2; 11]
and f = creer () in
    let () = List.iter (ajouter f) xs in
    let () = ajouter f 3
             (* [1; 10; 7; 5; 3; 2; 2; 11] *)
    and x = retirer f
             (* [10; 7; 5; 3; 2; 2; 11] et x = 1 *)
    and y = retirer f
             (* [10; 7; 5; 3; 2; 11] et y = 2 *)
```

1.1. Implémentation naïve

Avec une liste:

```
type 'a file = 'a list;;

let creer (): 'a file = []

and ajouter (f: 'a file) (x: 'a): 'a file = x :: f

and retirer (f: 'a file): 'a file * 'a =
    let rec trouver_min xs m = match xs with
    | [] -> m
    | h :: tl when m < h -> trouver_min xs m
    | h :: tl -> trouver_min xs h
    and retirer_valeur xs v = match xs with
    | [] -> []
    | h :: tl when h = v -> tl
    | h :: tl -> h :: (retire_valeur tl v)
    in match f with
    | [] -> failwith "rip bozo"
    | h :: tl -> let m = trouver_min tl h in (retire_valeur l m, m)
;;
```

Remarque

La fonction `retirer_valeur` est en $O(n)$ avec n le nombre d'élements stockés dans la file de priorité.

2. Tas

2.1. Structure et propriétés

Définition

Un *tas max* (resp. *tas min*) est un arbre binaire étiqueté par un ensemble totalement ordonnée et qui vérifie deux propriétés.

- il est complet
- il est tournoi, c'est à dire que chaque noeud a une étiquette inférieure (resp. supérieure) à celle de son père.

Propriété

La racine porte l'étiquette maximale dans un tas max et minimale dans un tas minimal.

Propriété

Les sous-arbres d'un tas sont des tas.

2.2. Ajouter une étiquette

On veut ajouter un élement à un tas en conservant les propriétés du tas.

Définition

La *percolation vers le haut* consiste à échanger l'étiquette d'un noeud avec celle de son père tant que:

- l'étiquette n'est pas sur la racine
- l'étiquette et celle de son père ne sont pas bien ordonnancées

Preuve de l'agorithme

Un variant est la profondeur du noeud qui porte l'étiquette qu'on déplace, donc ça termine.

On ajoute toujours une feuille de sorte à ce que l'arbre reste complet et l'algorithme de percolation vers le haut ne modifie que les étiquettes et pas la structure de l'arbre. Donc l'arbre est complet à la fin.

On rappelle qu'on suppose que avant de rajouter l'étiquette e, l'arbre était un tas. Alors la propriété *"le sous-arbre enraciné en le noeud d'étiquette e est un tas."* est un invariant, on l'admet.

Avant la percolation c'est vrai puisque l'arbre enraciné en e est la feuille (e), qui est un tas.

Ainsi l'invariant est vrai à la fin de la boucle et nous dit que l'arbre enraciné en x est un tas.

- Si l'algorithme s'est arrêté car e est la racine, l'arbre total est un tas.
- Sinon on peut montrer que la propriété de tournoi est vrai en tous les noeuds.

2.2.1. Complexité de la percolation vers le haut

Si on percole vers le haut une étiquette x qui est initialement à la profondeur p, alors la complexité est en $O(p)$. Donc percoler vers le haud un noeud quelconque dans $\mathcal{A}$ se fait en $O(h(\mathcal{A}))$.

La complexité ne prend pas en compte la recherche de x.

Rechercher dans un tas se fait en regardant tous les noeuds. Dans le cas où l'étiquette n'est pas présente, on ne peut pas s'en rendre compte avant la fin.

2.3. Retirer un élement

On ne retirera que la racine.

1. On supprime la dérnière feuille et on met son étiquette sur la racine.
2. On percole la nouvelle étiquette de la racine vers le bas pour rétablir la propriété de tournoi.

Algorithme de percolation vers la bas

Tant que x est mal ordonnée avec ses deux enfants et que x n'est pas une feuille: échanger x avec le plus grand des deux fils c'est un tas max (le plus petit sinon)

Complexité

Percoler vers le bas l'étiquette x qui se trouve initialement à la profondeur p coûte au maximum $h(\mathcal{A}) - p$ opérations ce qui est un $O(h(\mathcal{A}))$

Création à partir d'un tableau

Le but est de créer un tas qui contient exactement les valeurs données par un tableau t de taille n.

Méthode n°1 Créer un tas vide, ajouter 1 à 1 les élements dedans.

Méthode n°2 Créer un arbre binaire complet étiqueté par les valeurs du tableau.
On parcourt chaque niveau de l'arbre (profondeur constante) et on percole tous les noeuds vers le bas, de droit à gauche.

2.4. Complexité

On admet que l'opération d'ajout et l'opération de retrait de la racine s'effectuent en $O(h(\mathcal{A}))$.

Propriété

La hauteur d'un arbre complet à n noeuds est en $O(\log_2(n))$

Preuve

On note h_n la hauteur d'un arbre complet à n noeuds.

L'arbre complet à n noeuds à même hauteur que l'arbre parfait qu'on obtient en complétant sa dérnière ligne des avec feuilles.

Cet arbre parfait a forcément $2^{h_n+1} - 1$ noeuds. Donc

$$n \leq 2^{h_n+1} - 1$$

On sait aussi que l'arbre complet à n noeuds est de hauteur $+1$ par rapport à l'arbre parfait à n lignes. D'où:

$$2^{h_n} - 1 \leq n \leq 2^{h_n+1} - 1$$

et donc

$$h_n \leq \log_2(n+1) \leq h_n + 1$$

Finalement:

$$\boxed{h_n = O(\log_2(n))}$$

Corollaire

Les opérations d'ajout et de retrait sont en $O(\log_2(n)))$

Complexité de la méthode 1

- Créer un arbre vide coute $O(1)$
- Pour chaque élement, inserer l'élement coûte $O(n \log_2(n))$ car c'est n fois une opération qui coûte $O(n)$.

Au final c'est un $O(n \log_2(n))$

Complexité de la méthode 2

- Créer un arbre complet avec les valeurs du tableau en $O(n)$
- Percoler les noeuds vers le bas, de bas en haut, de droite à gauche.

Pour un noeud x à la profondeur p la percolation vers le bas coûte au plus $h(\mathcal{A}) - p$, au total on a donc moins de:

$$\sum_{p=0}^{h(\mathcal{A})} 2^p \cdot \Big(h(\mathcal{A}) - p\Big) = h(\mathcal{A}) \cdot \sum_{p=0}^{h(\mathcal{A})} 2^p - \sum_{p=0}^{h(\mathcal{A})} p \cdot 2^p = 2^{h(\mathcal{A})+1} - h(\mathcal{A}) - 2$$

C'est donc un $O(n)$ car $h(\mathcal{A}) = O(n \log_2(n))$.

2.5. Implémentation dans un tableau.

Les enfants du noeud situé à la case i sont situés dans les cases $2i+1$ et $2i+2$ et le père de ce noeud est situé dans la case $\lfloor \frac{i-1}{2} \rfloor$. On peut programmer assez facilement les fonctions d'ajout et de retrait pour des tas dans des tableaux.

2.6. Tri par tas

Le tri pas tas est un application des tas (stockés dans des tableaux). C'est un algorithme optimal de tri de tableau en place.

Principe

- On prend en entrée un tableau τ de taille n
- On utilise la **Méthode 2** pour le transformer en un tas.
- Pour i allant de $n-1$ à 1
  - On échange $\tau.(0)$ et $\tau.(i)$
  - On considère que le tableau est de taille i
  - On percole $\tau.(0)$ vers le bas.

2.6.1. Correction

A la fin d'un tour de boucle i, la partie du tableau située entre les indices 0 et $i-1$ représente un tas et la partie située entre les indices i et $n-1$ est un tableau trié.

De plus les élements entre les indices 0 et $i-1$ sont tous plus grands (ou plus petits) que ceux entre les indices 1 et $n-1$.

2.7. Implémentation en Ocaml

Comment trouver la première feuille qui n'existe pas ?

On suppose qu'on connait le nombre de noeuds de l'arbre (on peut le stocker dans la structure).
On rappelle qu'on a:

$$2^h < n \leq 2^{h-1}$$

d'où

$$h \leq \log_2(n) < h + 1$$

donc

$$h = \left\lfloor \log_2(n) \right\rfloor$$

On regarde le sous-arbre gauche. Il est parfait si le rectangle contient plus de $\frac{2^h}{2}$ sommets, c'est à dire 2^{h-1}.

Il faut aller à droite dès qu'on a plus au sens large de $2^h - 1 + 2^{h-1}$ sommets.

Si g est parfait, alors:

- g contient $2^h - 1$ sommets
- d contient $n - (2^h - 1) - 1 = n - 2^h$ sommets

Sinon, c'est d qui est parfait.

- g contient $n - 2^{h-1}$ sommets
- d contient $2^{h-1} - 1$ sommets

Erratum Pour le cas $(n = 3, h = 1)$, pour trouver la première feuille qui n'existe pas, on va à gauche si $n \geq 2^{h+1} - 1$.

Comment trouver la dernière feuille qui existe ?

C'est pareil, il faut aller à droite dès qu'on a plus au sens strict de $2^h - 1 + 2^{h+1}$ sommets dans l'arbre.

2.8 Fonction ajout

C'est une fonction récursive qui:

- en descendant cherche si la première feuille qui n'existe pas est à gauche ou à droite.
- en remontant percole la valeur si besoin. *(Cela fais plus de percolations que l'on voudrait mais pas plus que $h(\mathcal{A})$)*

```
let rec ajoute (a: 'a arbre) (e: 'a) (n: int): 'a arbre =
match a with
| Vide -> N(Vide, e, Vide)
| N(g, v, d) ->
    let h = n |> float_of_int |> log2 |> int_of_float in
    let p = int_of_float (2. **. float_of_int h) in

    if
        n < (p - 1) + (p / 2) || n >= (p * 2) - 1
    then (* on va à gauche *)
        let N(gg, vg, gd) = ajoute g e (n - p/2) in
        if vg < v then (* on percole *)
            N(N(gg, v, gd), vg, d)
        else (* on ne percole pas *)
            N(N(gg, vg, gd), v, d)
    else (* on va à droite *)
        let N(dg, vd, dd) = ajoute d e (n - p) in
        if vd < v then (* on percole *)
            N(g, vd, N(dg, v, dd))
        else (* on ne percole pas *)
            N(g, v, N(dg, vd, dd))
;;
```

2.9. Fonction retrait

```
let rec percole_bas: tas -> tas = function
    | N(Vide, v, N(dg, vd, dd)) when v > vd ->
        N(Vide, vd, percole_bas N(dg, v, dd))
    | N(N(gg, vg, gd), v, Vide) when v > vg ->
        N(percole_bas N(gg, v, gd), vg, Vide)
    | N(N(gg, vg, gd) as g, v, N(dg, vd, dd) as d)
      when (min vg vd) < v ->
        if vg = min vg vd then
            N(percole_bas N(gg, v, gd), vg, d)
        else
            N(g, vd, percole_bas N(dg, v, dd))
    | x -> x

and retire_feuille (a: tas) (n: int): tas * int =
    match a with
    | Vide -> failwith "rip bozo"
    | N(Vide, f, Vide) -> (Vide, f)
    | N(g, v, d) ->
        let h = n |> float_of_int |> log2 |> int_of_float in
        let p = int_of_float (2. **. float_of_int h) in

        if n <= p - 1 + p/2 then
            let g', f = retire_feuille g (n - p/2) in
                (N(g', v, d), f)
        else
            let d', f = retire_feuille d (n - p) in
                (N(g, v, d'), f)

and retrait (a: tas) (n: int): tas = let a', f = retire_feuille a n in
    match a' with
    | Vide -> N(Vide, f, Vide)
    | N(g, v, d) -> percole_bas N(g, f, d)
;;
```

Fin du chapitre