Tas, files et plus courts chemins

OPTION INFORMATIQUE - TP n° 3.6 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ définir un tas-min et un tas-max
- ☞ trier un tableau pas tas
- ☞ utiliser un tas pour créer une file de priorité
- ☞ appliquer les tas à l'algorithme de Dijsktra
- ☞ programmer impérativement en OCaml

A Implémentation d'un tas max et tri par tas

A1. Créer une variable `heap_test` de type `Array` contenant les entiers de 1 à 10 dans l'ordre décroissant. On génèrera automatiquement cette variable (pas in extenso). Est-ce un tas ? Si oui, de quel type ?

Solution :

```
let heap_test = Array.init 10 (fun i -> 10 - i);;
val heap_test : int array = [|10; 9; 8; 7; 6; 5; 4; 3; 2; 1|]
```

C'est un tas-max.

A2. Écrire une fonction de signature `swap : 'a array -> int -> int -> unit` qui échange la place de deux éléments dans un type `Array`. Tester cette fonction sur le tableau `heap_test`.

Solution :

```
let swap t i j = let tmp = t.(i) in t.(i) <- t.(j); t.(j) <- tmp;;
```

Comme en OCaml les `Array` sont indexés à partir de 0, on choisit de placer la racine du tas en 0. Puis, si le père est k alors les fils seront en $2k+1$ et $2k+2$. Si un fils est en k, alors le père est en $\lfloor \frac{k-1}{2} \rfloor$.
Il faut noter qu'un tas peut ne pas être rempli. Dans ce cas, on prendra soin de relever l'indice de la première case non remplie.

A3. Écrire une fonction de signature `up : 'a array -> int -> unit` qui, si cela est possible, fait monter un élément d'après son indice dans le tas tout en préservant la structure du tas.

Solution :

```
let up heap k =
  let rec aux i = match i with
    | 0 -> ()
    | _ -> let f = (i - 1)/2 in
           if heap.(i) > heap.(f) then (swap heap i f ; aux f)
  in  aux k;;
```

A4. Écrire une fonction de signature `down : 'a array -> int -> int -> unit` qui, si cela est possible, fait descendre un élément dans un tas max tout en préservant la structure du tas. Le premier entier de la signature est l'indice de la première case non remplie du tas, le second l'indice de l'élément à faire descendre.

Solution :

```
let down heap first_not_used k =
    let rec aux i = match i with
        | n when 2*n + 1 >= first_not_used -> ()
                            (* Out of bounds, it is done *)
        | n when 2*n + 1 = (first_not_used - 1) -> if heap.(n) < heap.(2*n +
             1) then swap heap (2*n + 1) n
                            (* just one last child *)
        | _ -> begin   (* two children *)
                let f = if heap.(2*i + 1) > heap.(2*i + 2) then  2*i +  1
                    else 2*i + 2 in
                if heap.(i) < heap.(f) then (swap heap i f; aux f;)
                 end
    in aux k;;
```

On cherche à créer un tas de deux manière différente :

- soit en considérant que le premier élément du tableau est un tas que l'on fait grossir avec les éléments de droite du tableau,
- soit en considérant que les feuilles sont des tas à un élément que l'on fait grossir au fur avec les éléments de gauche du tableau. On remarque dans ce cas que quelle que soit la taille n du tas, l'élément d'indice $n/2$ est toujours la première feuille. On va donc insérer les $n/2$ premiers éléments dans leurs feuilles et faire apparaître la structure de tas.

A5. Effectuer à la main la transformation du tableau `[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]` en tas-max par les deux méthodes.

A6. Écrire une fonction de signature `heap_make_up : 'a array -> unit` qui transforme un tableau en un tas en faisant monter les éléments. Quelle la complexité de cette fonction ?

Solution :

```
let heap_make_up t =  for k = 1 to Array.length t - 1 do up t k done;;
```

La complexité au pire de cette fonction est linéaire en $O(n \log n)$ si n est la taille du tas. En effet, dans le pire des cas, on fait remonter les n nœuds jusqu'à la racine, c'est à dire toute la hauteur de l'arbre qui vaut $\log n$.

A7. Écrire une fonction de signature `heap_make_down : 'a array -> unit` qui transforme un tableau en un tas en faisant descendre les éléments. Quelle est la complexité de cette fonction [1] ?

Solution :

```
let heap_make_down t = let size = Array.length t in
    for k = size/2 - 1 downto 0 do down t size k done;;
```

En considérant que le nombre de nœuds ayant la hauteur h est majoré par $\lceil \frac{n}{2^{h+1}} \rceil \leq 1 + \frac{nh}{2^{h+1}}$, on montre que :

$$\sum_{h=1}^{p} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \leqslant \frac{p(p+1)}{2} + n \sum_{h=1}^{p} \frac{h}{2^{h+1}} \tag{1}$$

Comme la série $\sum_h \frac{h}{2^{h+1}}$ converge et que $p = \lfloor \log n \rfloor$, on peut écrire que :

$$C(n) = O((\log n)^2 + n) = O(n) \tag{2}$$

La complexité de la création d'un tas est linéaire si on fait descendre.

A8. Écrire une fonction de signature `heap_sort : 'a array -> unit` qui effectue le tri par tas d'un tableau. Quelle la complexité de cette fonction ?

Solution : La complexité du tri par tas vaut : $C(n) = O(n + n \log_2 n) = O(n \log_2 n)$. On compte la création du tas (linéaire) et la descente ($\log_2 n$) des n éléments dans le tas.

```
let heap_sort t =
  heap_make_down t;
  let size = Array.length t in
    for k = size - 1  downto 0 do (swap t 0 k; down t k 0) done;;
```

A9. Tester le tri par tas sur un tableau de 10000 valeurs entières choisies aléatoirement entre 0 et 1000000.

Solution :

```
Random.init 42;;
let test = Array.init 100000 (fun i -> Random.full_int 1000000);;
heap_sort test;;
test;;
```

---

1. On suppose que le nombre de nœuds ayant la hauteur h est majoré par $\lceil \frac{n}{2^{h+1}} \rceil$