Application des arbres en algorithmique

Chapitre 02

1 Arbres binaires de recherche

Définition 1

Un arbre binaire de recherche est un arbre binaire tel que pour tout sommet s dont les fils sont f_g et f_d, pour tout sommet u de f_g, $u < s$; pour tout sommet v de f_d, $s < v$.

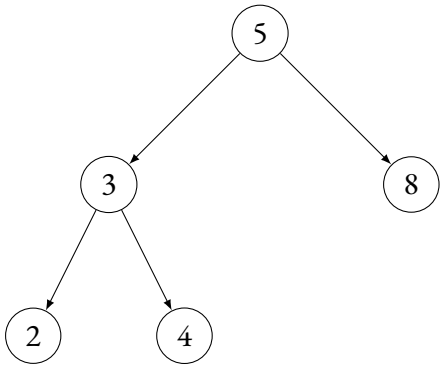
L'implémentation d'arbre binaire en OCaml choisie dans ce cours est la suivante :

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

C'est l'implémentation la plus classique des arbres binaires en OCaml. On remarquera que dans cette implémentation, on fait la distinction entre des nœuds internes étiquetés, et des feuilles non-étiquetées.

L'arbre ci-contre est un ABR : on dit même qu'il est *strict*, car tout nœud a 0 ou 2 fils.

```
let abr = N(5,
            N(3,
              N(2,V,V),
              N(4,V,V)),
            N(8,V,V));;
```



1.1 Fonctions élémentaires

```
let rec est_dedans elem arb =
    match arb with
    | V -> false
    | N(x,_,_) when x = elem -> true
    | N(x,a1,_) when elem < x -> est_dedans elem a1
    | N(x,_,a2) -> est_dedans elem a2;;
```

Le programme à gauche est récursif terminal ; si on note h la hauteur de l'arbre, la complexité temporelle est en $O(h)$ (et complexité spatiale $O(1)$).

Propriété 2 : hauteur et nombre de nœuds internes d'un arbre

Soit h la hauteur d'un arbre, et N le nombre de nœuds (internes ou non) de l'arbre. Alors $\log_2(N) \leqslant h \leqslant N$. Les deux cas sont atteints.

```
let rec ajouter elem arb =
    match arb with
    | V -> N(elem,V,V)
    | N(x,a1,a2) when elem <= x -> N(x,ajouter elem a1,a2)
    | N(x,a1,a2) -> N(x,a1,ajouter elem a2);;
```

Le programme à gauche n'est pas récursif terminal ; la complexité temporelle est en $O(h)$ (et complexité spatiale $O(1)$).

```
let rec hauteur arb =
    match arb with
    | V -> 0
    | N(x,a1,a2) -> if hauteur a1 >  hauteur a2
        then 1 + hauteur a1
        else 1 + hauteur a2;;
```

Le programme à gauche n'est pas récursif terminal ; la complexité temporelle est en $O(N)$ (et complexité spatiale $O(1)$).

```
let rec nb_nds arb =
    match arb with
    | V -> 0
    | N(x,a1,a2) -> 1 + (nb_nds a1 + nb_nds a2);;
```

Le programme à gauche n'est pas récursif terminal ; la complexité temporelle est en $O(N)$ (et complexité spatiale $O(1)$).

Les deux dernières complexités spatiales ne prennent pas en compte la taille de la pile d'appels, qui est en $O(N)$.

```
let rec min arb = (* : 'a arbre -> 'a * 'a arbre*)
  match arb with
  | V -> failwith "oups"
  | N(x,a1,a2) when a1=V -> (x,a2)
  | N(x,a1,a2) -> let (m,a3) = min a1 in (m,N(x,a3,a2));;

let rec supprimer elem arb =
  match arb with
  | V -> V
  | N(x,V,V) when x = elem -> V
  | N(x,a1,a2) when elem < x -> N(x,supprimer elem a1,a2)
  | N(x,a1,a2) when elem > x -> N(x,a1,supprimer elem a2)
  | N(x,a1,a2) -> let (m,a3) = min a2 in N(m,a1,a3);;
```

La suppression d'un élément est une opération plus délicate, pour laquelle plusieurs stratégies existent. La stratégie utilisée ici consiste à remplacer un nœud à supprimer par le nœud le plus à gauche de son fils de droite.
Les programmes ne sont pas récursifs terminaux, de complexités temporelles $O(h)$ tous les deux.

1.2 ABR et dictionnaires

Une utilisation des arbres binaires de recherche concerne l'implémentation de dictionnaires. Pour rappel : les dictionnaires forment une structure de données, avec les opérations élémentaires suivantes :

- `recherche(cle)` : renvoie la valeur associée à une clé ;
- `ajout(cle,valeur)` : ajoute une association clé-valeur ;
- `suppr(cle)` : supprime une clé, et la valeur associée ;
- `modif(cle,nv_valeur)` : modifie la valeur associée à une clé.

Un dictionnaire peut être implémenté comme une liste de couples `(clé,valeur)`, qui rend l'ensemble des opérations linéaires en la taille de la liste. Une autre implémentation est sous la forme d'un ABR, ordonné selon les clés :

```
type ('a,'b) dico_abr = V | N of ('a,'b) dico_abr * ('a,'b) dico_abr ;;
```

Dans cette implémentation, `'a` est le type des clés, et `'b` celui des valeurs. Ici, tout nœud stocke une clé et la valeur associée.

Propriété 3 : dictionnaire par ABR

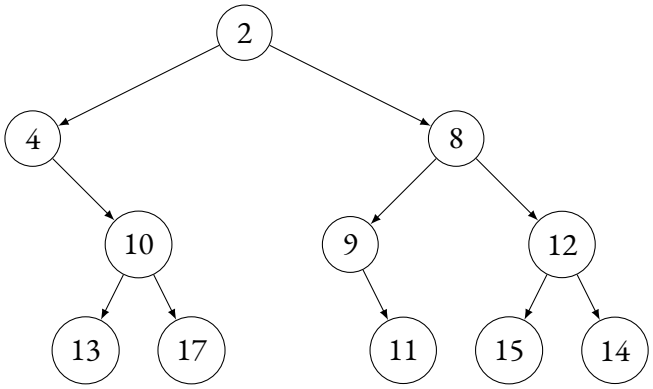
Les quatre opérations élémentaires sur les dictionnaires sont de complexité temporelles $O(h)$.

2 Tas binaires

Définition 4 : tas min

Un tas min est un arbre binaire tel que pour tout sommet s dont les fils sont f_g et f_d, $s < f_g$ et $s < f_d$.

Il existe la notion symétrique de tas max.



La figure de gauche est un tas min : chaque nœud est plus petit que ses fils.

```
let tas = N(2,
N(4,V,
    N(10,N(13,V,V),N(17,V,V))),
N(8,N(9,V,N(11,V,V)),
    N(12,N(15,V,V),N(14,V,V))));;
```

Propriété 5

Si un arbre binaire A est un tas min, alors les sous-arbres sont aussi des tas min. La racine de A est son plus petit élément.

Démonstration.

La première est simplement une restriction de la condition de tas min à un sous-arbre; la seconde se montre par induction.

2.1 Implémentation des tas en OCaml

Il existe une implémentation des arbres binaires (et notamment des tas) différente de l'implémentation inductive : elle se fait à l'aide d'un tableau.

```
type 'a tas = {mutable taille : int ; mutable tab : 'a array}
```

Pour rappel, il s'agit d'un type enregistrement, où un objet est caractérisé par différents champs. Ici, un tas est caractérisé par deux attributs :

— `taille`, qui donne le nombre d'éléments dans le tas;

— `tab`, qui stocke les valeurs du tas.

La représentation d'un arbre binaire dans un tableau repose sur l'idée très simple que, pour tout entier n, on peut facilement lui associer deux nouveaux entiers : $2n$ et $2n + 1$. Mieux encore, cette opération peut être fait dans le sens retour.

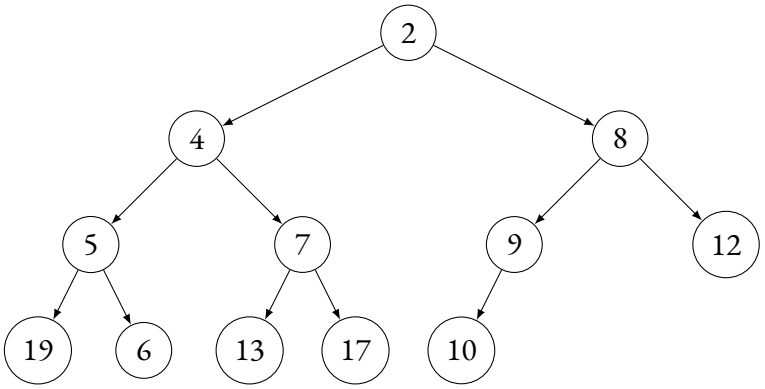
En OCaml, la première case d'un tableau commence à l'indice 0. Afin de simplifier la manipulation des indices dans ce cours, on maintiendra donc une case inutile en 0. Ce choix n'est pas nécessairement standard.

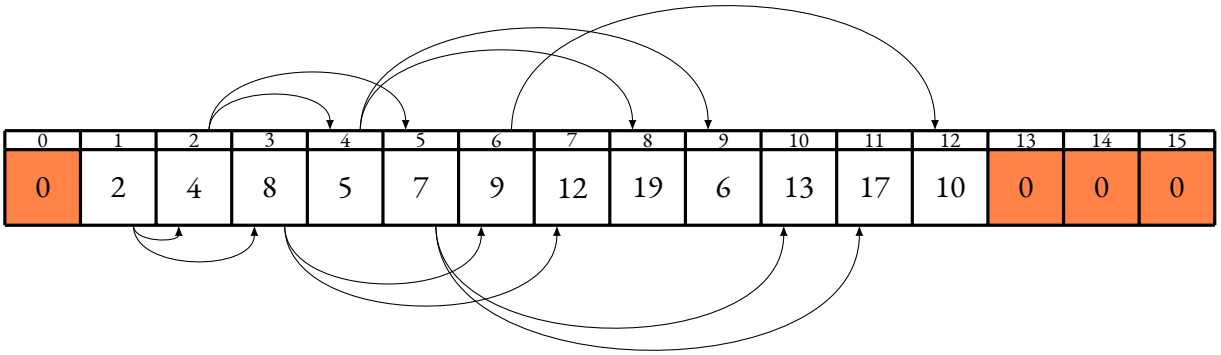
Dans cette implémentation des tas, on rajoute deux contraintes supplémentaires :

— les niveaux (sauf le dernier) du tas doivent être remplis;

— le dernier niveau est rempli le plus à gauche possible.

Il faudra prendre en compte ces deux contraintes en écrivant nos fonctions.






```
let t = {taille = 12 ; tab = [|0;2;4;8;5;7;9;12;19;6;13;17;10;0;0;0|]};;
```

Propriété 6 : équilibre des tas

Un tas remplissant les conditions supplémentaires de remplissage est équilibré.

Démonstration.

Le tas vide est équilibré, on traite le cas du tas non vide.
Soit $k \geqslant 1$ le nombre de niveaux complets dans le tas, et N le nombre de nœuds dans le tas. Alors :

$$\sum_{i=0}^{k-1} 2^i \leqslant N < \sum_{i=0}^{k} 2^i$$

$$2^k - 1 \leqslant N < 2^{k+1} - 1$$

1er cas : si $N = 2^k - 1$, alors tous les niveaux sont complets, et la hauteur de l'arbre est $k - 1$.
Donc $k - 1 \leqslant \log_2(2^{k-1}) \leqslant \log_2(2^k - 1) \leqslant \log_2(N)$.
2nd cas : si $N > 2^k - 1$, alors le dernier niveau est incomplet, et la hauteur de l'arbre est k.
Donc $k = \log_2(2^k) \leqslant \log_2(N)$.

En préservant les conditions de remplissage, on s'assurera donc que nos fonctions sont efficaces.

Les cases extrêmales servent 2 rôles différents :

- la case la plus à gauche sert à décaler les indices pour faciliter nos manipulations, et fournit une « valeur par défaut » pour les cases inutiles à droite ;
- les cases les plus à droite servent à stocker de futures valeurs qu'on pourrait rajouter, et ne représentent pas de « vraies » valeurs du tas.

Par ailleurs, dans cette implémentation, `t.tab.(t.taille)` renvoie la dernière « vraie » case du tas.

2.2 Files de priorité

Définition 7 : file de priorité

La structure de données des files de priorité est une structure disposant de trois opérations élémentaires :

- `creer` : crée une file de priorité vide.
- `ajouter(valeur)` : ajoute une valeur à la file de priorité ;
- `defiler` : renvoie la plus petite valeur de la file de priorité, en la retirant de la structure.

Les files de priorité sont une structure abstraite, qui disposent de plusieurs implémentations.

On pourrait par exemple implémenter les files de priorité par de simples listes :

```
type 'a fdp_liste = 'a list;;
```

Les trois fonctions fondamentales pourraient alors être les fonctions suivantes :

```
let creer () = ([] : 'a fdp_liste) ;;
```

```
let ajouter (f : 'a fdp_liste) (x : 'a) = ((x::f) : 'a fdp_liste);;
```

```
let defiler (f : 'a fdp_liste) =
  let rec aux g h m =
    match g with
    | [] -> ((m, h) : 'a * 'a fdp_listes)
    | t::q when t < m -> aux q (m::h) t
    | t::q -> aux q (t::h) m
  in
  match f with
  | [] -> failwith "file vide !"
  | t::q -> aux q [] t;;
```

Les fonctions `creer` et `ajouter` sont en temps $O(1)$. La fonction `defiler` consiste à déterminer le minimum de la liste en passant les éléments un par un, et en les stockant dans une liste auxiliaire qu'on renverra à la fin : sa complexité temporelle est en $O(|f|)$.

Malgré leurs noms, les files et les files de priorité n'ont pas forcément grand chose à voir.

On s'intéresse désormais à créer des fonctions implémentant les files de priorité en passant par les tas.

2.2.1 Création d'une file de priorité

```
let creer x_0 = {taille = 0 ; tab = [|x_0|]};;
```

Ici, `creer` nécessite un argument `x_0`, qui sera ce qui est contenu dans la case inutile la plus à gauche.

Un autre intérêt est que `x_0` fixe définitivement le type contenu dans le tas considéré.

Si on veut avoir une fonction `creer` sans argument (qu'on écrira donc `let creer () = ...`), une bonne méthode consiste à passer par les options (`None` et `Some`). Cela demande à modifier le type utilisé, les fonctions qui suivent, à rester cohérent dans la syntaxe tout le long. L'un des intérêts est notamment qu'en utilisant des options, les cases inutilisées sont d'une taille minimale.

2.2.2 Ajout d'une valeur dans le tas

L'ajout d'une valeur dans un tas, avec le type choisi, rencontre deux principales difficultés :

— parfois, il faudra augmenter la taille du tableau `tab`, car il sera plein ;

— où mettre la valeur supplémentaire ? faudra-t-il modifier le tas en conséquence ?

```
let ajouter v h =
```

Agrandissement du tableau

D'abord, commençons par agrandir le tableau s'il est plein.

```
if tas.taille = Array.length h.tab - 1 then
```

Nous allons alors doubler le nombre de cases du tas (ou en créer une, si le tas est vide).

```
let nb_cases = ref 0 in
nb_cases := 2*h.taille+2;
(* Ici, le +1 représente la case inutile à gauche. *)
let t = Array.make !nb_cases h.tab.(0) in
for i=0 to h.taille do
    t.(i) <- h.tab.(i)
done;
h.tab <- t;
```

Le doublement du nombre de cases dans `tab` a un coût temporel (et spatial) en $O(|\texttt{taille}|)$.

Ajout de la valeur et modification du tas associée

Maintenant, on a au moins une case libre. Le remplissage des tas contraint la case à remplir :

```
h.tab.(h.taille+1) <- v;
```

Tout se passe bien si, en faisant cette affectation, le nouveau fils est effectivement plus petit que son nouveau père (ou que le nœud ajouté est la racine) :

```
let ind = ref (h.taille+1) in
while !ind <> 1 && h.tab.(!ind) < h.tab.(!ind/2) do
    ?
done;
```

Si on est rentré dans la boucle `while`, c'est qu'il y a un problème : un fils est strictement plus petit que son père. On doit donc remonter le fils. Regardons un peu notre sous-arbre de plus près :

Les deux premiers cas sont possibles : mais les deux autres ne le sont pas, car h était au préalable un tas min rempli vers la gauche. On observe donc qu'il n'y a besoin de modifier que deux nœuds pour corriger notre sous-arbre : le père et le fils.

```
let temp = h.tab.(!ind) in
h.tab.(!ind) <- h.tab.(!ind/2);
h.tab.(!ind/2) <- temp;
ind := !ind/2
```

Cet ajout de la valeur, et la modification du tas associée, sont en temps $O(\text{hauteur})$: chaque passage de boucle est en $O(1)$, et on fait au plus un passage de boucle par niveau de l'arbre. Par équilibre du tas, cette opération est donc en $O(\log(\texttt{taille}))$.

Complexité dans le pire cas, complexité amortie

Quelle est la complexité temporelle de notre fonction `ajouter` ? Dans le pire cas, c'est simple : il s'agit du cas où on doit doubler le nombre de cases dans le tableau, et où on doit faire remonter la valeur ajoutée jusqu'à la racine. La première opération est en $O(|\texttt{taille}|)$, la seconde en $O(\log(|\texttt{taille}|))$: au total, on est donc en $O(|\texttt{taille}|)$, c'est peu efficace. On observe en particulier que ce qui coûte aussi cher est l'agrandissement du tableau.

Toutefois, cette opération arrive relativement peu souvent. Supposons effectuer N ajouts consécutifs ; alors le coût total de tous les agrandissements de tableaux est :

$$C_N = 1 + \sum_{k=1}^{N} k \times \delta(k) \qquad \text{avec } \delta(k) = \begin{cases} 1 \text{ si } k \text{ est une puissance de 2} \\ 0 \text{ sinon} \end{cases}$$

Ici, le 1+ représente l'ajout de la toute première vraie case du tableau ($k = 0$). On a donc effectué N opérations : on peut alors s'intéresser à C_N/N, la moyenne du coût de chaque opération.

Propriété 8

$C_N/N = O(1)$.

Démonstration.

Le pire cas correspond au moment où on a terminé par un agrandissement du tableau, c'est-à-dire si $N = 2^n$ pour un certain $n \in \mathbb{N}$. Alors :

$$\begin{aligned} C_{2^n} &= 1 + \sum_{k=1}^{2^n} k\delta(k) = 1 + \sum_{j=0}^{n} 2^j \\ &= 1 + 2^{n+1} - 1 = 2^{n+1} \end{aligned}$$

Donc $C_N/N = 2^{n+1}/2^n = 2$: dans tous les cas, $C_N/N = O(1)$.

Attention, cette « complexité moyennée » n'est pas la complexité moyenne : ici, on ne moyenne pas sur les différentes entrées possibles, mais sur des exécutions successives.

On s'est concentré ici sur la complexité temporelle : tout ce raisonnement est aussi valable dans l'étude de la complexité spatiale.

2.2.3 Défilement et extraction de la racine d'un tas

Le défilement sert à obtenir la valeur minimale stockée dans un tas. En soi, cette valeur minimale est très facile à obtenir : il s'agit de la racine du tas. La difficulté concerne la mise à jour du tas, en maintenant les exigences de remplissage. Pour ce faire, nous allons introduire quelques fonctions auxiliaires simples qui augmenteront la lisibilité de la fonction principale.

Vérification de l'ordre entre un père et ses fils

On étudie deux cas, selon qu'un père a un ou deux fils. Si un père a un seul fils, c'est que son fils est le dernier nœud.

```
let est_pere_petit h ind =
    if ind = h.taille/2
        then h.tab.(ind) <= h.tab.(2*ind)
    else
        h.tab.(ind) <= h.tab.(2*ind) &&
        h.tab.(ind) <= h.tab.(2*ind+1)
```

Indice du plus petit fils

On fait encore la distinction selon la présence d'un ou deux fils.

```
let ind_fils_petit h ind =
  if ind = h.taille/2 then 2*ind
  else begin
  if h.tab.(2*ind) < h.tab.(2*ind+1) then 2*ind
  else 2*ind+1 end
```

Extraction de la racine

Maintenant, réfléchissons à notre fonction de défilement. Ce qu'on renverra à la fin est accessible simplement : c'est la première vraie valeur du tableau. Il faut toutefois faire un test au préalable, au cas où notre tas est vide.

```
let defiler h =
    if h.taille = 0 then failwith "Tas vide !";
    let r = h.tab.(1) in
```

La difficulté consiste à maintenir la structure de tas après avoir retiré la racine, en maintenant les critères sur le remplissage du tas. Ce dernier critère nous donne en fait une contrainte très simple : il faut ranger la dernière vraie valeur du tas au bon endroit. Où pourrait-on mettre cette valeur ? Une case vient de se libérer, celle de la racine : plaçons notre valeur ici temporairement.

```
let v = h.tab.(h.taille) in
let ind = ref 1 in
h.taille <- h.taille -1 ;
h.tab.(1) <- !v ;
```

Maintenant, nous allons procéder selon la logique suivante : si `ind` correspond à un indice de père, et que ce père est plus grand que l'un de ses fils, alors il faut procéder à une modification du tas.

```
while !ind <= h.taille/2 && not (est_pere_petit h !ind) do
```

Comment modifier le tas ? On va simplement procéder à un échange entre le père trop grand et son fils le plus petit. Pour rappel, `v` contient la valeur à descendre. On mettra à jour `ind`, qui est notre indice de travail.

```
let ind2 = ind_fils_petit h !ind in
h.tab.(!ind) <- h.tab.(ind2);
h.tab.(ind2) <- v ;
ind := ind2;
```

Une fois terminé, il suffit de renvoyer `r`.

Complexité

La complexité spatiale est $O(1)$: on a besoin de quelques variables, et les modifications se font en place dans le tableau.

Pour la complexité temporelle : il suffit de remarquer que chaque passage de la boucle `while` est en $O(1)$, et qu'on procède au plus à un passage par niveau de l'arbre : la complexité temporelle est donc en $O(\text{hauteur})$, et comme le tas est équilibré, cela donne $O(\log(|\texttt{taille}|))$.

2.2.4 Code entier

```
type 'a tas = {mutable taille : int; mutable tab : 'a array};;

let creer x_0 =
  {taille = 0 ; tab = [|x_0|]};;

let ajouter v h =
  if h.taille = Array.length h.tab - 1 (* le tableau est plein*)
  then
    begin
      let nb_cases = ref 0 in
      nb_cases := 2*h.taille+2;
      let t = Array.make !nb_cases h.tab.(0) in
      for i=0 to h.taille do
        t.(i) <- h.tab.(i)
      done;
      h.tab <- t;
    end;
  h.tab.(h.taille + 1) <- v;
  let ind = ref (h.taille+1) in
  while !ind <> 1 && h.tab.(!ind) < h.tab.(!ind/2) do
    let temp = h.tab.(!ind) in
    h.tab.(!ind) <- h.tab.(!ind/2);
    h.tab.(!ind/2) <- temp;
    ind := !ind/2
  done;
  h.taille <- h.taille+1;;

let est_pere_petit h ind =
if ind = h.taille/2
    then h.tab.(ind) <= h.tab.(2*ind)
else
    h.tab.(ind) <= h.tab.(2*ind) &&
    h.tab.(ind) <= h.tab.(2*ind+1);;
```

```
let ind_fils_petit h ind =
  if ind = h.taille/2 then 2*ind
  else begin
  if h.tab.(2*ind) < h.tab.(2*ind+1) then 2*ind
  else 2*ind+1 end ;;

let defiler h =
  if h.taille = 0 then failwith "Tas vide !";
  let r = h.tab.(1) in
  let v = h.tab.(h.taille) in
  let ind = ref 1 in
  h.taille <- h.taille -1 ;
  h.tab.(1) <- v ;
  while !ind <= h.taille/2 && not (est_pere_petit h !ind) do
    let ind2 = ind_fils_petit h !ind in
    h.tab.(!ind) <- h.tab.(ind2);
    h.tab.(ind2) <- v ;
    ind := ind2;
  done;
  r;;
```

2.3 Tri par tas

L'implémentation des files de priorité par les tas est plutôt efficace : on peut alors exploiter cette efficacité pour obtenir un nouvel algorithme de tri.

Supposons vouloir trier une liste `l`. On peut alors utiliser les files de priorité de la façon suivante :

— créer une file de priorité vide `f` ;

— ajouter à `f` les éléments de `l` petit à petit (en vidant `l` au passage);

— extraire successivement de `f` ses racines, et les stocker au fur et à mesure dans `l`.

```
let tri_par_tas l =
  match l with
  | [] -> []
  | a::b -> let h = creer a in
    let rec aux li =
      match li with
      | [] -> ()
      | t::q -> ajouter t h; aux q
    in
    aux l;
    let rec aux2 t =
      match t.taille with
      | 0 -> []
      | _ -> let x = defiler t in
        x::(aux2 t)
    in
    aux2 h;;
```

Dans le cas du tri par tas, l'implémentation des files de priorité choisie est celle des tas : cela permet d'atteindre une bonne complexité.

Créer `f` (ici, jouée par le tas `h`) et remplir `f` avec les éléments de `l` nécessite $O(n\log(n))$ opérations (avec $n = |\texttt{l}|$); défiler les racines de `f` nécessite aussi $O(n\log(n))$ opérations.

> Ici, on a étudié le cas où on veut trier une liste. Lorsqu'on cherche à trier un tableau, il est possible d'adapter nos fonctions de telle sorte que le tri par tas soit en place (le tableau à trier prendra la place du `tab` d'un tas). Le tri par tas est le tri implémenté par défaut en OCaml pour les tableaux.