```
(* Tableaux dynamiques *)
type 'a dynarray = {mutable taille : int; mutable elements : 'a array}

let add_elt dynarray =
  let taille = dynarray.taille in
  if taille < Array.length dynarray.elements
  then (dynarray.elements.(taille) <- elt)
  else (* il faut redimensionner [elements] *)
    (assert (taille = Array.length dynarray.elements);
     let new_elements = Array.make (max 1 (2*taille)) elt in
     Array.blit dynarray.elements 0 new_elements 0 taille;
     dynarray.elements <- new_elements);
  dynarray.taille <- succ taille

let take dynarray =
  assert (dynarray.taille > 0);
  dynarray.taille <- pred dynarray.taille;
  dynarray.elements.(dynarray.taille)

let get dynarray = Array.get dynarray.elements
let set dynarray = Array.set dynarray.elements
let make length elt = {taille = length; elements = Array.make length elt}
let iter f dynarray =
  for i=0 to pred dynarray.taille do f dynarray.elements.(i) done
let dynarray_of_array array = {taille = Array.length array; elements = Array.copy array}


(* Version avec réduction de la taille si nécessaire et tests. *)
let take dynarray =
  let result = take dynarray in
  let half_length = Array.length dynarray.elements /2 in
  if dynarray.taille < half_length
  then dynarray.elements <- Array.sub dynarray.elements 0 half_length;
  result

let get dynarray index =
  assert (0 <= index && index < dynarray.taille);
  get dynarray index

let set dynarray index =
  assert (0 <= index && index < dynarray.taille);
  set dynarray index


(* Arbres binaires *)
let pere i =
  assert (i>0);
  pred i  /2

let fils i = 2*i +1, 2*i +2
(* Autre version pour les fous d'optimisation ou ceux qui aiment le code cryptique: *)
let fils i = succ (i lsl 1), succ i lsl 1

(* Tas *)
let rec percole arbre noeud =
  let valeur_noeud = get arbre noeud in
  let fils_g,fils_d = fils noeud in
  if fils_g < arbre.taille then (* si [noeud] a un fils gauche *)
    let valeur_fils_g = get arbre fils_g in
    if fils_d < arbre.taille then (* si [noeud] a un fils droit *)
      let valeur_fils_d = get arbre fils_d in
      (* Le plus petit des deux fils de [noeud], et sa valeur: *)
      let min_fils, valeur_min_fils =
        if valeur_fils_g <valeur_fils_d then fils_g,valeur_fils_g else fils_d,valeur_fils_d in
      (if valeur_noeud > valeur_min_fils then
         (* sinon il n'y a rien à faire : [arbre] est déjà un tas *)
         (set arbre noeud valeur_min_fils; set arbre min_fils valeur_noeud;
          percole arbre min_fils))
    else (* si [noeud] a seulement un fils gauche *)
      (if valeur_noeud > valeur_fils_g then
         (set arbre noeud valeur_fils_g; set arbre fils_g valeur_noeud;
          percole arbre fils_g))
(* Si [noeud] n'a pas de fils gauche, il n'a pas non plus de fils droit. *)

let take_min arbre =
  let result = get arbre 0 in
  let new_racine = take arbre in
  if arbre.taille > 0 (* Pour n'appeler [set] que si l'arbre n'est pas vide. *)
  then (set arbre 0 new_racine; percole arbre 0);
  result
```

```
let taifie array =
  let arbre = dynarray_of_array array in
  if Array.length array > 1 then
    for i = pere (pred arbre.taille) downto 0 do (* On commence au dernier noeud ayant un fils
*)
      percole arbre i;
    done;
  arbre

let tri array =
  let tas = taifie array in
  Array.init (Array.length array) (fun _ -> take_min tas)
(* Remarque: on pourrait éviter les redimenseionnements du tableau. *)

let echange dynarray a b =
  let temp = get dynarray a in
  set dynarray a (get dynarray b);
  set dynarray b temp

let rec percole_bottom_up arbre noeud =
  if noeud > 0 then
    let pere = pere noeud in
    if get arbre noeud < get arbre pere then
      (echange arbre noeud pere; percole_bottom_up arbre pere)

(* On crée une feuille d'étiquette [elt],
et tant que elt est plus grand que son père on l'échange avec lui,
le faisant ainsi remonter de bas en haut (bottom-up) *)
let insere elt arbre =
  add elt arbre;
  percole_bottom_up arbre (pred arbre.taille)
```