```
type ('a,'b) arbre =
  | Noeud of 'a * ('a,'b) arbre * ('a,'b) arbre
  | Feuille of 'b

(* Le plus petit élément ne peut avoir de père, c'est donc la racine.
   Le plus grand ne peut avoir de fils, c'est donc une feuille.
   Mais il n'est pas nécessairement à la profondeur maximale. *)

let etiquette = function
  | Noeud (result,_,_) -> result
  | Feuille result -> result

let test_racine = function
  | Feuille _ -> true
  | Noeud (e,fils_g,fils_d) -> e <= etiquette fils_g && e <= etiquette fils_d

let rec is_heap = function
  | Feuille _ -> true
  | Noeud (_,fils_g,fils_d) as arbre ->
      test_racine arbre && is_heap fils_g && is_heap fils_d

(* Pour ceux qui veulent optimiser en limitant le nombre de filtrages: *)
exception Pas_tas
let is_heap arbre =
  (* renvoie l'étiquette de la racine si c'est un tas,
     soulève l'exception Pas_tas sinon *)
  let rec parcours = function
    | Feuille result -> result
    | Noeud (result,fils_g,fils_d) ->
        if result > parcours fils_g || result > parcours fils_d then raise Pas_tas;
        result in
  try ignore (parcours arbre); true
  with Pas_tas -> false

let rec parcours_infixe = function
  | Feuille e -> Printf.printf "Feuille %i\n%!" e
  | Noeud (e,fils_g,fils_d) ->
      parcours_infixe fils_g;
      Printf.printf "Noeud interne %i\n%!" e;
      parcours_infixe fils_d

type noeud = Node of int | Leaf of int

let parcours_postfixe arbre=
  let result = ref [] in
  (* Cette liste va accumuler les éléments au fur et à mesure de leur visite.
  Elle contiendra donc les éléments dans l'ordre inverse de leur visite,
  d'où le List.rev à la fin de la fonction.*)
  let rec parcours = function
    | Feuille e -> result:= Leaf e :: !result
    | Noeud (e,fils_g,fils_d) ->
        parcours fils_g;
        parcours fils_d;
        result:= Node e :: !result in
  parcours arbre;
  List.rev !result

(* Pour évaluer des expressions en notation postfixe, on utilise une pile.
Chaque fois qu'on rencontre une feuille
(correspondant à une constante ou une variable dans une expression), on l'empile.
Chaque fois qu'on rencontre un noeud
(correspondant à une fonction dans une expression),
on va chercher ses arguments sur la pile et on empile le résultat.
Exemple : l'expression postfixe
1 2 + 3 * 4 5 6 + * -
correspond à l'expression mathématique (infixe !)
(1+2)*3 - 4*(5+6)
. On voit d'ailleurs que l'on a besoin de parenthèses,
ce qui répond à la question 3.5.*)
let reconstruit liste =
  let pile = Stack.create () in
  let reste = ref liste in
  let continue = ref true in
  while !continue do match !reste with
    | [] -> continue := false
    | Leaf e :: q ->
        Stack.push (Feuille e) pile;
        reste := q
    | Node e :: q ->
        let fils_d = Stack.pop pile in
        let fils_g = Stack.pop pile in
        Stack.push (Noeud (e,fils_g,fils_d)) pile;
        reste:= q
  done;
  Stack.pop pile
```

```
(* 3.5 On peut écrire l'équivalent de reconstruit pour un parcours préfixe car
la fonction "parcours_prefixe" est toujours injective. En revanche le parcours
infixe n'est pas injectif : il renvoit la même liste pour les arbres
  N(4,
    N(2, F 1, F 3),
    N(6, F 5, F 7))
et
  N(2,
    F 1,
    N(4,
      F 3,
      N(6, F 5, F 7)))
.*)

(* 4.1 Un élément entre dans la file au moment où son père en sort, ou bien au
début pour la racine. La racine est donc traitée une fois et une seule. Une
récurrence immédiate montre que chaque élément est traité exactement une
fois. D'où la terminaison. *)

let parcours_largeur f g t =
  let a_traiter = Queue.create () in
  Queue.add t a_traiter;
  try while true do
    match Queue.take a_traiter with
      | Feuille e -> g e
      | Noeud (e,fils_g,fils_d) ->
          f e; Queue.add fils_g a_traiter; Queue.add fils_d a_traiter
  done with Queue.Empty -> ()
;;


(* Question bonus : parcours des graphes.
Le principe est le même, mais il y a deux difficultés supplémentaires:
 - Il peut y avoir plusieurs chemins entre le sommet de départ
   (la racine dans le cas des arbres) et un sommet donné.
   Il faut donc mémoriser si un sommet a déjà été visité ou non, ce qui est fait
   dans le tableau [visited]. Un sommet n'est traité que s'il n'a pas déjà été
   visité.
 - Il peut n'y avoir aucun chemin entre le sommet de départ et un sommet
   donné, d'où la boucle [for] qui prend un à un tous les sommets comme sommet
   de départ.*)

type graphe = int list array
(* graphe.(i) est la liste des voisins du sommet i. *)

let parcours_profondeur f graphe =
  let n = Array.length graphe in
  let visited = Array.make n false in
  let rec parcours_composante_connexe sommet =
    if not visited.(sommet) then (
      visited.(sommet) <- true;
      f sommet;
      List.iter (parcours_composante_connexe) graphe.(sommet)) in
  for i = 0 to pred n do parcours_composante_connexe i done

let parcours_largeur f graphe =
  let n = Array.length graphe in
  let visited = Array.make n false in
  let a_traiter = Queue.create () in
  let parcours_composante_connexe sommet =
    Queue.add sommet a_traiter;
    try while true do
      let sommet = Queue.take a_traiter in
      if not visited.(sommet) then (
        f sommet;
        List.iter (fun voisin -> Queue.add voisin a_traiter) graphe.(sommet);
        visited.(sommet) <- true)
    done with Queue.Empty -> () in
  for i = 0 to pred n do parcours_composante_connexe i done
```