```
(* ******************************************************************************* *)
(* Tas de Tarjan / union-find *)
type element = Representant | Reductible of int
type classes = element array

let isoles n = Array.make n Representant

let rec find classes elt = match classes.(elt) with
    | Representant -> elt
    | Reductible i ->
        let representant = find classes i in
        classes.(elt) <- Reductible representant;
        representant

let union classes elt1 elt2 =
  let r1 = find classes elt1 in
  let r2 = find classes elt2 in
  if r1 <> r2 then classes.(r1) <- Reductible r2

(* test *)
let foo = isoles 10;;
union foo 6 3; foo;;
union foo 3 0; foo;;
union foo 9 6; foo;;
union foo 8 5; foo;;
union foo 5 2; foo;;
find foo 8;;
foo;;

(* version recursive terminale de find *)
let find classes elt =
  let rec find_rec elt accu = match classes.(elt) with
    | Representant -> List.iter (fun i -> classes.(i) <- Reductible elt) accu; elt
    | Reductible i -> find_rec i (elt :: accu) in
  find_rec elt []


(* ********************************************************************** *)
(* Arbre aleatoire *)
type tree = Leaf | Node of tree * tree

let random_tree_vect n =
  let result = Array.make (2*n +1) None in
  result.(0) <- Some (1,2);
  for i = 1 to pred n do
    let index = Random.int (2*i +1) in
    let node = result.(index) in
    if Random.bool ()
    then (result.(2*i +1) <- None; result.(2*i +2) <- node)
    else (result.(2*i +1) <- node; result.(2*i +2) <- None);
    result.(index) <- Some (2*i+1, 2*i+2)
  done;
  result

let tree_of_vect v =
  let rec parcours i = match v.(i) with
    | None -> Leaf
    | Some (j,k) -> Node (parcours j, parcours k) in
  parcours 0

let random_tree n = tree_of_vect (random_tree_vect n)

#load "graphics.cma"
#load "draw_tree.cmo"
let rec ab_of_tree = function
  | Leaf -> Draw_tree.Feuille ()
  | Node (a,b) -> Draw_tree.Noeud ((), ab_of_tree a, ab_of_tree b)

let _ =
  Graphics.open_graph " 1024x768";
  for i=1 to 10 do
    Graphics.clear_graph ();
    Draw_tree.drawt_points (ab_of_tree (random_tree 700));
    ignore (Graphics.read_key ())
  done;
  Graphics.close_graph ()

(* ********************************************************************** *)
(* Listes doublement chainees *)
type ’a cell = {
  mutable a:’a;
  mutable prec:’a cell option;
  mutable suiv:’a cell option}
type ’a liste = {
  mutable deb:’a cell option;
  mutable fin:’a cell option}

exception Empty

let empty () = {deb = None; fin = None};;

let singleton elt =
  let cell = Some {a = elt; prec = None; suiv = None} in
  {deb = cell; fin = cell}

let add_deb liste elt =
  match liste.deb with
    | None ->
        assert (liste.fin = None);
        let cell = Some {a = elt; prec = None; suiv = None} in
        liste.deb <- cell;
        liste.fin <- cell
    | Some deb ->
        let cell = Some {a = elt; prec = None; suiv = liste.deb} in
        deb.prec <- cell;
        liste.deb <- cell

let add_fin liste elt =
  match liste.fin with
    | None ->
        assert (liste.deb = None);
        let cell = Some {a = elt; prec = None; suiv = None} in
        liste.deb <- cell;
        liste.fin <- cell
    | Some fin ->
        let cell = Some {a = elt; prec = liste.fin; suiv = None} in
        fin.suiv <- cell;
        liste.fin <- cell

let take_deb liste = match liste.deb with
  | None -> raise Empty
  | Some {a=result; suiv = s} ->
      liste.deb <- s;
      (match s with
         | Some cell -> cell.prec <- None
         | None -> liste.fin <- None);
      result

let take_fin liste = match liste.fin with
  | None -> raise Empty
  | Some {a=result; prec = p} ->
      liste.fin <- p;
      (match p with
         | Some cell -> cell.suiv <- None
         | None -> liste.deb <- None);
      result

let to_list =
  let rec to_list_cell accu = function
    | {prec = Some prec; a = a} -> to_list_cell (a::accu) prec
    | {prec = None; a = a} -> a::accu in
function
  | {fin = None} -> []
  | {fin = Some fin} -> to_list_cell [] fin

let rec of_list = function
  | [] -> empty ()
  | a::q ->
      let result = of_list q in
      add_deb result a; result

(* version recursive terminale: *)
let of_list liste =
  let rec of_list accu = function
    | [] -> accu
    | a::q ->
        add_fin accu a;
        of_list accu q in
  of_list (empty ()) liste

let rev_liste liste =
  let rec rev_cell_option = function
    | None -> ()
    | Some ({suiv=s} as c) ->
        c.suiv <- c.prec; c.prec <- s;
        rev_cell_option s in
  let deb = liste.deb in
  rev_cell_option deb;
  liste.deb <- liste.fin; liste.fin <- deb
```

```
let a = empty () in
add_deb a 3; add_fin a 4;
add_deb a 2; add_deb a 1;
to_list a

let b = singleton 42 in
add_deb b 12; add_deb b 12;
Printf.printf "%i\n%!" (take_fin b);
add_fin b 32;
Printf.printf "%i, %i\n%!" (take_deb b) (take_deb b);
to_list b

let c = of_list [1;2;3;4;5] in
add_deb c 0;
to_list c

let d = of_list [0;1;2;3;4;5] in
rev_liste d;
to_list d
```