3　検索 (続)

要素の削除

```
let rec get_max t =
  match t with
    Empty -> raise Not_found
  | Node (t1, d, t2) ->
      if t2 = Empty then d else get_max t2
val get_max : 'a tree -> 'a
let rec remove x t =
  match t with
    Empty -> raise Not_found
  | Node (t1, d, t2) ->
      if d.key = x then
        if t2 = Empty then t1 else
        if t1 = Empty then t2 else
        let d' = get_max t1 in bal (remove d'.key t1) d' t2
      else if x < d.key then
        bal (remove x t1) d t2
      else
        bal t1 d (remove x t2)
val remove : 'a -> ('a, 'b) node tree -> ('a, 'b) node tree
```

4　グラフ・アルゴリズム

グラフの表現

```
let distances =
  [ "Nagoya", ["Sakae", 2; "Hisaya", 3];
    "Sakae", ["Nagoya", 2; "Hisaya", 1; "Imaike", 3];
    "Hisaya", ["Nagoya", 3; "Sakae", 1; "Imaike", 3; "Motoyama", 11];
    "Imaike", ["Motoyama", 3; "Sakae", 3; "Hisaya", 3];
    "Motoyama", ["Imaike", 3; "Hisaya", 11; "Daigaku", 1];
    "Daigaku", ["Motoyama", 1] ]
val distances : (string * (string * int) list) list = ...

type 'a graph = {names: 'a list; vertices: (int*int) list array}

let rec index x l =
  match l with
    [] -> raise Not_found
  | a::l -> if x = a then 0 else 1 + index x l
val index : 'a -> 'a list -> int

let add_edge g a b w =
  let v = g.vertices in
  let i = index a g.names and j = index b g.names in
  v.(i) <- (j,w) :: v.(i);                              (* 配列の破壊的代入 *)
  v.(j) <- (i,w) :: v.(j)
val add_edge : 'a graph -> 'a -> 'a -> int -> unit
```

```
let build_graph ll =
  let len = List.length ll and names = List.map fst ll in
  let g = {names = names; vertices = Array.create len []} in
  List.iter
    (fun (a, neighbours) ->
      List.iter (fun (b, w) -> add_edge g a b w) neighbours)
    ll;
  g
val build_graph : ('a * ('a * int) list) list -> 'a graph

let graph = build_graph distances
val graph : string graph =
  {names = ["Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"; "Daigaku"];
   vertices = [|[(2, 3); (1, 2); (2, 3); (1, 2)]; ...|]}
```

最短経路問題

Dijkstra によるアルゴリズム．ノードを確定 (Reached)，処理中 (Fringe)，未到達 (Unseen) に分類して，毎回，処理中で最も近いノードを Reached に格上げし，そこから辿れるノードを Fringe に追加する．

```
type status = Reached of int * int | Fringe of int * int | Unseen

(* source の距離が srcdist と確定されたとき，隣接するノードを処理する *)
(* "~" で始まる変数名はラベル付き引数．型に名前が現われる *)
let visit_neighbour ~status ~source ~srcdist (next, dist) =
  match status.(next) with
  | Reached _ -> ()
  | Fringe (d, src) ->
      let d' = srcdist + dist in
      if d' < d then status.(next) <- Fringe (d', source)
  | Unseen ->
      status.(next) <- Fringe (srcdist + dist, source)
val visit_neighbour :
  status:status array -> source:int -> srcdist:int -> int * int -> unit

(* 最も近い Fringe のノードを見付ける．一個もなければ 0 を返す．*)
let closest_fringe status =
  let closest = ref 0 in
  Array.iteri
    (fun i s ->
      match status.(!closest), s with
      | Fringe (d, _), Fringe (di, _) ->
          if di < d then closest := i
      | _, Fringe _ -> closest := i
      | _ -> ())
    status;
  !closest
val closest_fringe : status array -> int

let rec shortest_rec ~graph ~status =
  let next = closest_fringe status in
  match status.(next) with
    Fringe (d, src) ->
      status.(next) <- Reached (d, src);                (* Reached に格上げ *)
      List.iter (visit_neighbour ~status ~source:next ~srcdist:d)
        graph.vertices.(next);
      shortest_rec ~graph ~status                         (* 処理を続ける *)
  | _ -> ()                        (* Fringe がなくなったときに処理が終わる *)
val shortest_rec : graph:'a graph -> status:status array -> unit

(* start から辿れる全てのノードの距離を計算し，status として返す *)
let shortest_paths ~graph ~start =
  let status = Array.create (Array.length graph.vertices) Unseen in
  status.(index start graph.names) <- Fringe (0, -1);
  shortest_rec ~graph ~status;
  status
val shortest_paths : graph:'a graph -> start:'a -> status array

let shortest_path ~graph ~start ~dest =
  let status = shortest_paths ~graph ~start in
  let v = index dest graph.names in
  match status.(v) with
  | Reached (d, _) -> Some d
  | _ -> None
;;
val shortest_path : graph:'a graph -> start:'a -> dest:'a -> int option

# shortest_path graph "Nagoya" "Daigaku";;
- : int option = Some 9
```

練習問題 4.1 *1.* 状態が Reached になると，距離と一緒に一歩前に通ったノードが分かる．この情報を利用すれば，逆向きに辿ることで経路を読み出せる．以下の型の関数を定義し，それを shortest_path で利用しなさい．

```
val read_path :
  graph:'a graph -> status:status array -> dest:'a -> 'a list
```

2. 上記のプログラムでは closest_fringe が全てのノードを調べている．本当なら，*Fringe* の要素をリストとして持って，closest_fringe の処理効率を上げたい．必要な関数の型が以下のようになる．

```
val remove : 'a -> 'a list -> 'a list
val visit_neighbour :
  status:status array -> fringe:int list ->
  source:int -> srcdist:int -> int * int -> int list
val closest_fringe : status array -> int list -> int
val shortest_rec :
  graph:'a graph -> status:status array -> fringe:int list -> unit
```