二分木の生成

```
let rec split n l =
  match l with
  | a :: l' when n > 0 ->
      let (l1, l2) = split (n-1) l' in
      (a::l1, l2)
  | l -> ([], l)
val split : int -> 'a list -> 'a list * 'a list = <fun>
let rec balance_list l =
  let n = List.length l in
  if n = 0 then l else
  match split (n/2) l with
    (l1, a::l2) -> a :: balance_list l1 @ balance_list l2
  | (l1, []) -> assert false
val balance_list : 'a list -> 'a list = <fun>
let btree_of_list l = add_list Empty (balance_list l)
val btree_of_list : (int * 'a) list -> 'a btree = <fun>
```

要素の削除

```
let rec min_binding t =
  match t with
    Empty -> raise Not_found
  | Node n ->
      match n.left with
        Empty -> (n.key, n.data, n.right)
      | l ->
          let (x, d, t) = min_binding l in
          (x, d, bal t n.key n.data n.right)
val min_binding : 'a avl_tree -> int * 'a * 'a avl_tree = <fun>
let rec remove x t =
  match t with
    Empty -> t
  | Node n ->
      if x < n.key then bal (remove x n.left) n.key n.data n.right else
      if x > n.key then bal n.left n.key n.data (remove x n.right) else
      match n.left, n.right with
        Empty, r -> r
      | l, Empty -> l
      | l, r ->
          let (x, d, r') = min_binding r in
          bal l x d r'
val remove : int -> 'a avl_tree -> 'a avl_tree = <fun>
```

6 グラフ・アルゴリズム

グラフの表現

```
let distances : (string * (string * int) list) list =
  [ "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 = ...

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 = <fun>
let matrix_of_graph ll =
  let len = List.length ll in
  let m = Array.create_matrix len len (-1) in
  let names = List.map fst ll in
  let line = ref 0 in
  List.iter
    (fun (name, neighbours) ->
      let i = index name names in
      List.iter (fun (name, w) -> m.(i).(index name names) <- w)
        neighbours)
    ll;
  (names, m)
val matrix_of_graph : ('a * ('a * int) list) list -> 'a list * int array array
let matrix = matrix_of_graph distances
val matrix : string list * int array array =
  (["Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"; "Daigaku"],
   [|[|-1; 2; 3; -1; -1; -1|]; [|2; -1; 1; 3; -1; -1|];
     [|3; 1; -1; 3; 11; -1|]; [|-1; 3; 3; -1; 3; -1|];
     [|-1; -1; 11; 3; -1; 1|]; [|-1; -1; -1; -1; 1; -1|]|])

type 'a node = {data: 'a; mutable neighbours: ('a node * int) list}
let rec assoc_node x l =
  match l with
    [] -> raise Not_found
  | n::l -> if n.data = x then n else assoc_node x l
val assoc_node : 'a -> 'a node list -> 'a node = <fun>
let nodes_of_graph ll =
  let nodes = List.map (fun (n,_) -> {data = n; neighbours = []}) ll in
  List.iter2
    (fun n (_,l) ->
      n.neighbours <- List.map (fun (x,w) -> (assoc_node x nodes, w)) l)
    nodes ll;
  nodes
val nodes_of_graph : ('a * ('a * int) list) list -> 'a node list = <fun>
let nodes = nodes_of_graph distances ;;
val nodes : string node list =
  [{data = "Nagoya"; neighbours =
     [({data = "Sakae"; neighbours = ...})]}]
```

最短経路問題

単純な幅優先アルゴリズムによる計算.

```
let merge old current l =
  let known = current @ old in
  let l =                                                    (* 短かいものだけを残す *)
    List.filter
      (fun (n,w) -> try List.assoc n known > w with Not_found -> true)
      l
  in                                                         (* 過去の長いものを捨てる *)
  l @ List.filter (fun (n,_) -> not (List.mem_assoc n l)) current
val merge : ('a * 'b) list -> ('a * 'b) list -> ('a * 'b) list -> ('a * 'b) list
let rec bests old current extra =
  match extra with
    [] -> current
  | l :: extra -> bests old (merge old current l) extra
val bests :
  ('a * 'b) list -> ('a * 'b) list -> ('a * 'b) list list -> ('a * 'b) list
let shortest_path n1 n2 =                                    (* n1 から n2 への最短経路 *)
  let rec shortests old added  =                          (* n1 からの最短経路を全て計算する *)
    if added = [] then old else                              (* added は前回追加されたもの *)
    let extra =                                           (* 一歩先のものを全て計算してみる *)
      List.map (fun (n,w) -> List.map (fun (n',w') -> (n',w+w')) n.neighbours)
        added
    and old = added @ old in
    let added = bests old [] extra in                     (* 新しいもので短かいものを選ぶ *)
    let old = List.filter (fun (n,_) -> not (List.mem_assoc n added)) old in
    shortests old added
  in List.assoc n2 (shortests [n1,0] n1.neighbours)
val shortest_path : 'a node -> 'a node -> int = <fun>
# shortest_path (assoc_node "Nagoya" nodes) (assoc_node "Daigaku" nodes);;
- : int = 9
```

実習課題 (6 回目)

1. shortest_path を元のリストによる表現に対して使えるように直しなさい．型が以下のようになる.

   ```
   val shortest_path :
     (string * (string * int) list) list -> string -> string -> int
   ```

   ヒント: merge と bests は直す必要はない．shortest_path も三個所の訂正で済む.

2. 経路 (途中駅) も返すように定義を変えなさい.

   ```
   val shortest_path : 'a node -> 'a node -> (int * string list)
   ```

3. shortest_path を行列表現に対して定義しなさい．リストを使わないやりかたが望ましい.

22日までに 感想や課題のできた分だけを computer-lecture-2005-aw-4@math.nagoya-u.ac.jp 宛に送って下さい.