安定した整列 merge_sort

```
let rec merge l1 l2 =
  match l1, l2 with
  | a::l1', b::l2' ->
      if a < b then a :: merge l1' l2 else b :: merge l1 l2'
  | l1, [] -> l1
  | [], l2 -> l2
val merge : 'a list -> 'a list -> 'a list = <fun>

let rec split l =
  match l with
  | a::b::l ->
      let (l1, l2) = split l in
      (a :: l1, b :: l2)
  | l -> (l, [])
val split : 'a list -> 'a list * 'a list = <fun>

let rec merge_sort l =
  match l with
  | [] -> []
  | [a] -> l
  | [a;b] -> if a <= b then l else [b;a]
  | l ->
      let (l1, l2) = split l in
      merge (merge_sort l1) (merge_sort l2)
val merge_sort : 'a list -> 'a list = <fun>
```

5 探索アルゴリズム

直線探索

```
let rec assoc (x : 'a) (l : ('a * 'b) list) =
  match l with
    [] -> raise Not_found
  | (a,b)::l ->
      if x = a then b else assoc x l
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc 2 [3,"c"; 1,"a"; 2,"b"];;
- : string = "b"
```

長所： 要素が簡単に追加できる $O(1)$
短所： 探索が遅い $O(n)$

二分探索

```
let search (x : int) (arr : ('a * 'b) array) =
  let rec search first last =
    if first = last then
      let (a,b) = arr.(first) in
      if x = a then b else raise Not_found
    else
      let middle = (first + last) / 2 in
      let (a,_) = arr.(middle) in
      if x <= a then search first middle
      else search (middle+1) last
  in search 0 (Array.length arr + 1)
val search : int -> (int * 'a) array -> 'a = <fun>
# search 3 [|1,"a"; 2,"b"; 3,"c"; 4,"d"|] ;;
- : string = "c"
```

長所： 探索が早い $O(n \log n)$
短所： 追加の度に整列しなければならない $O(n)$

二分木探索

```
type 'a btree = Empty | Node of 'a node
and 'a node = { left: 'a btree; key: int; data: 'a; right: 'a btree }

let rec search x t =
  match t with
    Empty -> raise Not_found
  | Node n ->
      if x < n.key then search x n.left else
      if x > n.key then search x n.right else
      n.data
val search : int -> 'a btree -> 'a = <fun>

let rec add x d t =
  match t with
    Empty -> Node {left = Empty; key = x; data = d; right = Empty}
  | Node n ->
      if x < n.key then Node {n with left = add x d n.left} else
      if x > n.key then Node {n with right = add x d n.right} else
      Node {n with data = d}
val add : int -> 'a -> 'a btree -> 'a btree = <fun>
```

長所： バランスが取れていれば，追加も検索も早い $O(\log n)$
短所： バランスがくずれると遅くなる 最悪 $O(n)$

平衡二分木

```
type 'a avl_tree = Empty | Node of 'a avl_node
and 'a avl_node =
    { left: 'a avl_tree; key: int; data: 'a; right: 'a avl_tree; height: int }
let height t =
  match t with
    Empty -> 0
  | Node n -> n.height
val height : 'a avl_tree -> int = <fun>

let create l x d r =
  Node { left = l; key = x; data = d; right = r;
         height = max (height l) (height r) + 1 }
val create : 'a avl_tree -> int -> 'a -> 'a avl_tree -> 'a avl_tree = <fun>

let bal l x d r =
  let hl = height l and hr = height r in
  match l, r with
    Node n, _ when hl = hr + 2 ->
      begin match n.right with
        Node n' when n'.height > height n.left ->
          create (create n.left n.key n.data n'.left)
            n'.key n'.data (create n'.right x d r)
      | _ -> create n.left n.key n.data (create n.right x d r)
      end
  | _, Node n when hr = hl + 2 ->
      begin match n.left with
        Node n' when n'.height > height n.right ->
          create (create l x d n'.left)
            n'.key n'.data (create n'.right n.key n.data n.right)
      | _ -> create (create l x d n.left) n.key n.data n.right
      end
  | _ -> create l x d r
val bal : 'a avl_tree -> int -> 'a -> 'a avl_tree -> 'a avl_tree = <fun>

let rec add x d t =
  match t with
    Empty -> create Empty x d Empty
  | Node n ->
      if x < n.key then bal (add x d n.left) n.key n.data n.right else
      if x > n.key then bal n.left n.key n.data (add x d n.right) else
      Node {n with data = d}
val add : int -> 'a -> 'a avl_tree -> 'a avl_tree = <fun>
```

長所： バランスを取りながら $O(\log n)$

実習課題 (5 回目)

1. 以下の関数で二分木に値を入れる

```
let rec add_list t l =
  match l with
    [] -> t
  | (k,d)::l -> add_list (add k d t) l
val add_list : 'a btree -> (int * 'a) list -> 'a btree
```

   二分木の形が最悪になる要素のリストを考えて下さい．その場合の探索時間が $O(n)$ であることも確認して下さい．

2. 同じ要素のリストを平衡二分木に入れるとどうなりますか．

3. 整列されたリストから最適な二分木 (高さが $\log n$) を作る関数を書きなさい．

```
val btree_of_list : (int * 'a) list -> 'a btree
```

4. 整列されたリストを並べ替えて，上記の add_list に通せば最適な二分木になるような関数を書きない．

```
val balance_list : (int * 'a) list -> (int * 'a) list
let btree_of_list l = add_list Empty (balance_list l)
val btree_of_list : (int * 'a) list -> 'a btree
```

5. 平衡二分木から値を除く関数を書きなさい．

   remove : int -> 'a avl_tree -> 'a avl_tree

   (let t' = remove n t をすると，t' の中では n が含まれていない．)