Interface for module Rbset

1. Sets implemented as Red-Black trees. Interface copied from Ocaml's *Set*.

```
module type OrderedType =
  sig
    type t
    val compare : t → t → int
  end

module type S =
  sig
    type elt
    type t
    val empty : t
    val is_empty : t → bool
    val mem : elt → t → bool
    val add : elt → t → t
    val singleton : elt → t
    val remove : elt → t → t
    val union : t → t → t
    val inter : t → t → t
    val diff : t → t → t
    val compare : t → t → int
    val equal : t → t → bool
    val subset : t → t → bool
    val for_all : (elt → bool) → t → bool
    val exists : (elt → bool) → t → bool
    val filter : (elt → bool) → t → t
    val partition : (elt → bool) → t → t × t
    val cardinal : t → int
    val elements : t → elt list
    val min_elt : t → elt
    val max_elt : t → elt
    val choose : t → elt
    val split : elt → t → t × bool × t
    val iter : (elt → unit) → t → unit
    val fold : (elt → α → α) → t → α → α
  end

module Make(Ord : OrderedType) : (S with type elt = Ord.t)
```

Module Rbset

2. Sets interface.

```
module type OrderedType =
  sig
    type t
    val compare : t → t → int
  end

module type S =
  sig
    type elt
    type t
    val empty : t
    val is_empty : t → bool
    val mem : elt → t → bool
    val add : elt → t → t
    val singleton : elt → t
    val remove : elt → t → t
    val union : t → t → t
    val inter : t → t → t
    val diff : t → t → t
    val compare : t → t → int
    val equal : t → t → bool
    val subset : t → t → bool
    val for_all : (elt → bool) → t → bool
    val exists : (elt → bool) → t → bool
    val filter : (elt → bool) → t → t
    val partition : (elt → bool) → t → t × t
    val cardinal : t → int
    val elements : t → elt list
    val min_elt : t → elt
    val max_elt : t → elt
    val choose : t → elt
    val split : elt → t → t × bool × t
    val iter : (elt → unit) → t → unit
    val fold : (elt → α → α) → t → α → α
  end
```

3. Sets implemented as reb-black trees.

```
module Make(Ord : OrderedType) : (S with type elt = Ord.t) = struct
  type elt = Ord.t
  type t = Empty | Black of t × elt × t | Red of t × elt × t
```

Invariants: (1) a red node has no red son, and (2) any path from the root to a leaf has the same number of black nodes

Note the use of two constructors *Black* and *Red* to save space (resulting in longer code at a few places, e.g. in function *remove*). These red-black trees saves 20% of space w.r.t Ocaml's AVL, which store the height into a fourth argument.

4. For debug only: checks whether a tree is properly colored

```
exception Bad
```

check_aux s checks invariants and returns the black height

```
let rec check_aux = function
  | Empty →
      0
  | Red (Red _, _, _) | Red (_, _, Red _) →
                                   raise Bad
  | Black (l, _, r) →
      let h = check_aux l in
      if check_aux r ≠ h then raise Bad;
      succ h
  | Red (l, _, r) →
      let h = check_aux l in
      if check_aux r ≠ h then raise Bad;
      h

let check s = try ignore (check_aux s); true with Bad → false
```

5. Implementation of the set operations; *empty*, *is_empty*, *mem* and *singleton* are trivial.

```
let empty = Empty

let is_empty = function Empty → true | _ → false

let rec mem x = function
  | Empty → false
  | Black (l, v, r) | Red (l, v, r) →
      let c = Ord.compare x v in
      c = 0 ∨ mem x (if c < 0 then l else r)
```

Note: the variant for *mem* proposed in Okasaki's "Purely Functional Data Structures" is useless in the case of a ternary comparison function.

```
let singleton x = Black (Empty, x, Empty)
```

6. Insertion

```
let lbalance x1 x2 x3 = match x1, x2, x3 with
  | Red (Red (a,x,b), y, c), z, d ->
      Red (Black (a,x,b), y, Black (c,z,d))
  | Red (a, x, Red (b,y,c)), z, d ->
      Red (Black (a,x,b), y, Black (c,z,d))
  | a,x,b ->
      Black (a,x,b)

let rbalance x1 x2 x3 = match x1, x2, x3 with
  | a, x, Red (Red (b,y,c), z, d) ->
      Red (Black (a,x,b), y, Black (c,z,d))
  | a, x, Red (b, y, Red (c,z,d)) ->
      Red (Black (a,x,b), y, Black (c,z,d))
  | a,x,b ->
      Black (a,x,b)

let add x s =
  let rec ins = function
    | Empty ->
        Red (Empty, x, Empty)
    | Red (a, y, b) as s ->
        let c = Ord.compare x y in
        if c < 0 then Red (ins a, y, b)
        else if c > 0 then Red (a, y, ins b)
        else s
    | Black (a, y, b) as s ->
        let c = Ord.compare x y in
        if c < 0 then lbalance (ins a) y b
        else if c > 0 then rbalance a y (ins b)
        else s
  in
  match ins s with
    | Black _ as s -> s
    | Red (a, y, b) -> Black (a, y, b)
    | Empty -> assert false
```

7. Removal

unbalanced_left repares invariant (2) when the black height of the left son exceeds (by 1) the black height of the right son

```
let unbalanced_left = function
  | Red (Black (t1, x1, t2), x2, t3) →
      lbalance (Red (t1, x1, t2)) x2 t3, false
  | Black (Black (t1, x1, t2), x2, t3) →
      lbalance (Red (t1, x1, t2)) x2 t3, true
  | Black (Red (t1, x1, Black (t2, x2, t3)), x3, t4) →
      Black (t1, x1, lbalance (Red (t2, x2, t3)) x3 t4), false
  | _ →
      assert false
```

unbalanced_right repares invariant (2) when the black height of the right son exceeds (by 1) the black height of the left son

```
let unbalanced_right = function
  | Red (t1, x1, Black (t2, x2, t3)) →
      rbalance t1 x1 (Red (t2, x2, t3)), false
  | Black (t1, x1, Black (t2, x2, t3)) →
      rbalance t1 x1 (Red (t2, x2, t3)), true
  | Black (t1, x1, Red (Black (t2, x2, t3), x3, t4)) →
      Black (rbalance t1 x1 (Red (t2, x2, t3)), x3, t4), false
  | _ →
      assert false
```

remove_min $s = (s', m, b)$ extracts the minimum m of s, s' being the resulting set, and indicates with b whether the black height has decreased

```
let rec remove_min = function
  | Empty →
      assert false
  (* minimum is reached *)
  | Black (Empty, x, Empty) →
      Empty, x, true
  | Black (Empty, x, Red (l, y, r)) →
      Black (l, y, r), x, false
  | Black (Empty, _, Black _) →
      assert false
  | Red (Empty, x, r) →
      r, x, false
  (* minimum is recursively extracted from l *)
  | Black (l, x, r) →
      let l', m, d = remove_min l in
      let t = Black (l', x, r) in
      if d then
```

```
        let t, d' = unbalanced_right t in t, m, d'
      else
        t, m, false
  | Red (l, x, r) →
      let l', m, d = remove_min l in
      let t = Red (l', x, r) in
      if d then
        let t, d' = unbalanced_right t in t, m, d'
      else
        t, m, false
```

```
let blackify = function
  | Red (l, x, r) → Black (l, x, r), false
  | s → s, true
```

remove_aux x s $=$ (s', b) removes x from s and indicates with b whether the black height has decreased

```
let remove x s =
  let rec remove_aux = function
    | Empty →
        Empty, false
    | Black (l, y, r) →
        let c = Ord.compare x y in
        if c < 0 then
          let l', d = remove_aux l in
          let t = Black (l', y, r) in
          if d then unbalanced_right t else t, false
        else if c > 0 then
          let r', d = remove_aux r in
          let t = Black (l, y, r') in
          if d then unbalanced_left t else t, false
        else (* x = y *)
          (match r with
             | Empty →
                 blackify l
             | _ →
                 let r', m, d = remove_min r in
                 let t = Black (l, m, r') in
                 if d then unbalanced_left t else t, false)
    | Red (l, y, r) →
        let c = Ord.compare x y in
```

```
          if c < 0 then
            let l', d = remove_aux l in
            let t = Red (l', y, r) in
            if d then unbalanced_right t else t, false
          else if c > 0 then
            let r', d = remove_aux r in
            let t = Red (l, y, r') in
            if d then unbalanced_left t else t, false
          else (* x = y *)
            (match r with
              | Empty ->
                  l, false
              | _ ->
                  let r', m, d = remove_min r in
                  let t = Red (l, m, r') in
                  if d then unbalanced_left t else t, false)
    in
    let s', _ = remove_aux s in s'
```

8. The sorted list of elements

```
  let rec elements_aux accu = function
    | Empty ->
        accu
    | Black (l, v, r) | Red (l, v, r) ->
        elements_aux (v :: elements_aux accu r) l

  let elements s =
    elements_aux [] s
```

9. The functions *union*, *inter*, *diff* and *compare* are implemented over the lists of elements. So we need first a function to build a set from a list.

10. Building a red-black tree from a sorted list in reverse order. The result is a complete binary tree, where all nodes are black, except the bottom line which is red.

```
  let log2 n = truncate (log (float n) /. log 2.)
```

```
let of_list sl =
  let rec build sl n k =
    if k = 0 then
      if n = 0 then
        Empty, sl
      else match sl with
        | [] ->
            assert false
        | x :: sl ->
            Red (Empty, x, Empty), sl
    else
      let n' = (n - 1) / 2 in
      match build sl n' (k - 1) with
        | _, [] ->
            assert false
        | l, x :: sl ->
            let r, sl = build sl (n - n' - 1) (k - 1) in
            Black (r, x, l), sl
  in
  let n = List.length sl in
  fst (build sl n (log2 n))
```

11. Merges two sorted lists, into a sorted list in reverse order

```
let union_list l1 l2 =
  let rec merge_aux acc = function
    | [], l2 ->
        List.rev_append l2 acc
    | l1, [] ->
        List.rev_append l1 acc
    | (x1 :: r1 as l1), (x2 :: r2 as l2) ->
        let c = Ord.compare x1 x2 in
        if c < 0 then merge_aux (x1 :: acc) (r1, l2)
        else if c > 0 then merge_aux (x2 :: acc) (l1, r2)
        else merge_aux (x1 :: acc) (r1, r2)
  in
  merge_aux [] (l1, l2)

let union s1 s2 =
  of_list (union_list (elements s1) (elements s2))
```

12. Intersects two sorted lists, into a sorted list in reverse order

```
let inter_list l1 l2 =
  let rec inter_aux acc = function
    | [], _ | _, [] →
        acc
    | (x1 :: r1 as l1), (x2 :: r2 as l2) →
        let c = Ord.compare x1 x2 in
        if c = 0 then inter_aux (x1 :: acc) (r1, r2)
        else if c < 0 then inter_aux acc (r1, l2)
        else (* c ¿ 0 *) inter_aux acc (l1, r2)
  in
  inter_aux [] (l1, l2)

let inter s1 s2 =
  of_list (inter_list (elements s1) (elements s2))
```

13. Difference of two sorted lists, into a sorted list in reverse order

```
let diff_list l1 l2 =
  let rec diff_aux acc = function
    | [], _ →
        acc
    | l1, [] →
        List.rev_append l1 acc
    | (x1 :: r1 as l1), (x2 :: r2 as l2) →
        let c = Ord.compare x1 x2 in
        if c = 0 then diff_aux acc (r1, r2)
        else if c < 0 then diff_aux (x1 :: acc) (r1, l2)
        else (* c ¿ 0 *) diff_aux acc (l1, r2)
  in
  diff_aux [] (l1, l2)

let diff s1 s2 =
  of_list (diff_list (elements s1) (elements s2))
```

14. Comparison. Uses lists, but could be optimized following Ocaml's *Set*.

```
let rec compare_list = function
  | [], [] → 0
  | [], _ → −1
  | _, [] → 1
  | x1 :: r1, x2 :: r2 →
      let c = Ord.compare x1 x2 in
      if c ≠ 0 then c else compare_list (r1, r2)
```

```
let compare s1 s2 = compare_list (elements s1, elements s2)

let equal s1 s2 = compare s1 s2 = 0
```

15. Subset. Copied from Ocaml's sets

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ →
      true
  | _, Empty →
      false
  | (Black (l1, v1, r1) | Red (l1, v1, r1)),
    (Black (l2, v2, r2) | Red (l2, v2, r2) as t2) →
      let c = Ord.compare v1 v2 in
      if c = 0 then
        subset l1 l2 ∧ subset r1 r2
      else if c < 0 then
        subset (Black (l1, v1, Empty)) l2 ∧ subset r1 t2
      else
        subset (Black (Empty, v1, r1)) r2 ∧ subset l1 t2
```

16. Other functions

```
let rec for_all p = function
  | Empty → true
  | Black (l, v, r) | Red (l, v, r) → p v ∧ for_all p l ∧ for_all p r

let rec exists p = function
  | Empty → false
  | Black (l, v, r) | Red (l, v, r) → p v ∨ exists p l ∨ exists p r

let filter p s =
  let rec filt accu = function
    | Empty → accu
    | Black (l, v, r) | Red (l, v, r) →
        filt (filt (if p v then add v accu else accu) l) r
  in
  filt Empty s

let partition p s =
  let rec part (t, f as accu) = function
    | Empty → accu
    | Black (l, v, r) | Red (l, v, r) →
        part (part (if p v then (add v t, f) else (t, add v f)) l) r
```

```
    in
    part (Empty, Empty) s
  let rec cardinal = function
    | Empty → 0
    | Black (l, _, r) | Red (l, _, r) → cardinal l + 1 + cardinal r
  let rec min_elt = function
    | Empty → raise Not_found
    | Black (Empty, v, _) | Red (Empty, v, _) → v
    | Black (l, _, _) | Red (l, _, _) → min_elt l
  let rec max_elt = function
    | Empty → raise Not_found
    | Black (_, v, Empty) | Red (_, v, Empty) → v
    | Black (_, _, r) | Red (_, _, r) → max_elt r
  let choose = min_elt
  let rec iter f = function
    | Empty → ()
    | Black (l, v, r) | Red (l, v, r) → iter f l; f v; iter f r
  let rec fold f s accu = match s with
    | Empty → accu
    | Black (l, v, r) | Red (l, v, r) → fold f l (f v (fold f r accu))
  let split x s =
    let coll k (l, b, r) =
      let c = Ord.compare k x in
      if c < 0 then add k l, b, r
      else if c > 0 then l, b, add k r
      else l, true, r
    in
    fold coll s (Empty, false, Empty)
end
```