Interface for module Ptset

1. Sets of integers implemented as Patricia trees. The following signature is exactly *Set.S* with type *elt* = *int*, with the same specifications. This is a purely functional data-structure. The performances are similar to those of the standard library's module *Set*. The representation is unique and thus structural comparison can be performed on Patricia trees.

```
include Set.S with type elt = int
```

2. Warning: *min_elt* and *max_elt* are linear w.r.t. the size of the set. In other words, *min_elt t* is barely more efficient than *fold min t (choose t)*.

3. Additional functions not appearing in the signature *Set.S* from ocaml standard library. *intersect u v* determines if sets *u* and *v* have a non-empty intersection.

```
val intersect : t → t → bool
```

4. Big-endian Patricia trees

```
module Big : sig
  include Set.S with type elt = int
  val intersect : t → t → bool
end
```

5. Big-endian Patricia trees with non-negative elements. Changes: - *add* and *singleton* raise *Invalid_arg* if a negative element is given - *mem* is slightly faster (the Patricia tree is now a search tree) - *min_elt* and *max_elt* are now O(log(N)) - *elements* returns a list with elements in ascending order

```
module BigPos : sig
  include Set.S with type elt = int
  val intersect : t → t → bool
end
```

Module Ptset

6. Sets of integers implemented as Patricia trees, following Chris Okasaki and Andrew Gill's paper *Fast Mergeable Integer Maps* (`http://www.cs.columbia.edu/~cdo/papers.html#ml98maps`). Patricia trees provide faster operations than standard library's module *Set*, and especially very fast *union*, *subset*, *inter* and *diff* operations.

7. The idea behind Patricia trees is to build a *trie* on the binary digits of the elements, and to compact the representation by branching only one the relevant bits (i.e. the ones for which there is at least on element in each subtree). We implement here *little-endian* Patricia trees: bits are processed from least-significant to most-significant. The trie is implemented by the following type t. *Empty* stands for the empty trie, and *Leaf* k for the singleton k. (Note that k is the actual element.) *Branch* (m, p, l, r) represents a branching, where p is the prefix (from the root of the trie) and m is the branching bit (a power of 2). l and r contain the subsets for which the branching bit is respectively 0 and 1. Invariant: the trees l and r are not empty.

```
type t =
  | Empty
  | Leaf of int
  | Branch of int × int × t × t
```

8. Example: the representation of the set $\{1, 4, 5\}$ is

```
Branch (0, 1, Leaf 4, Branch (1, 4, Leaf 1, Leaf 5))
```

The first branching bit is the bit 0 (and the corresponding prefix is 0_2, not of use here), with $\{4\}$ on the left and $\{1, 5\}$ on the right. Then the right subtree branches on bit 2 (and so has a branching value of $2^2 = 4$), with prefix $01_2 = 1$.

9. Empty set and singletons.

```
let empty = Empty

let is_empty = function Empty → true | _ → false

let singleton k = Leaf k
```

10. Testing the occurrence of a value is similar to the search in a binary search tree, where the branching bit is used to select the appropriate subtree.

```
let zero_bit k m = (k land m) ≡ 0

let rec mem k = function
  | Empty → false
  | Leaf j → k ≡ j
  | Branch (_, m, l, r) → mem k (if zero_bit k m then l else r)

let find k s = if mem k s then k else raise Not_found
```

11. The following operation *join* will be used in both insertion and union. Given two non-empty trees *t0* and *t1* with longest common prefixes *p0* and *p1* respectively, which are supposed to disagree, it creates the union of *t0* and *t1*. For this, it computes the first bit m

where *p0* and *p1* disagree and create a branching node on that bit. Depending on the value of that bit in *p0*, *t0* will be the left subtree and *t1* the right one, or the converse. Computing the first branching bit of *p0* and *p1* uses a nice property of twos-complement representation of integers.

```
let lowest_bit x = x land (-x)

let branching_bit p0 p1 = lowest_bit (p0 lxor p1)

let mask p m = p land (m - 1)

let join (p0, t0, p1, t1) =
  let m = branching_bit p0 p1 in
  if zero_bit p0 m then
    Branch (mask p0 m, m, t0, t1)
  else
    Branch (mask p0 m, m, t1, t0)
```

12. Then the insertion of value k in set t is easily implemented using *join*. Insertion in a singleton is just the identity or a call to *join*, depending on the value of k. When inserting in a branching tree, we first check if the value to insert k matches the prefix p: if not, *join* will take care of creating the above branching; if so, we just insert k in the appropriate subtree, depending of the branching bit.

```
let match_prefix k p m = (mask k m) ≡ p

let add k t =
  let rec ins = function
    | Empty → Leaf k
    | Leaf j as t →
        if j ≡ k then t else join (k, Leaf k, j, t)
    | Branch (p, m, t0, t1) as t →
        if match_prefix k p m then
          if zero_bit k m then
            Branch (p, m, ins t0, t1)
          else
            Branch (p, m, t0, ins t1)
        else
          join (k, Leaf k, p, t)
  in
  ins t

let of_list = List.fold_left (fun s x → add x s) empty
```

13. The code to remove an element is basically similar to the code of insertion. But since we have to maintain the invariant that both subtrees of a *Branch* node are non-empty, we use here the "smart constructor" *branch* instead of *Branch*.

```
let branch = function
  | (_, _, Empty, t) → t
  | (_, _, t, Empty) → t
  | (p, m, t0, t1) → Branch (p, m, t0, t1)

let remove k t =
  let rec rmv = function
    | Empty → Empty
    | Leaf j as t → if k ≡ j then Empty else t
    | Branch (p, m, t0, t1) as t →
        if match_prefix k p m then
          if zero_bit k m then
            branch (p, m, rmv t0, t1)
          else
            branch (p, m, t0, rmv t1)
        else
          t
  in
  rmv t
```

14. One nice property of Patricia trees is to support a fast union operation (and also fast subset, difference and intersection operations). When merging two branching trees we examine the following four cases: (1) the trees have exactly the same prefix; (2/3) one prefix contains the other one; and (4) the prefixes disagree. In cases (1), (2) and (3) the recursion is immediate; in case (4) the function *join* creates the appropriate branching.

When comparing branching bits, one has to be careful with the leftmost bit (which is negative), so we introduce function *unsigned_lt* below.

```
let unsigned_lt n m = n ≥ 0 ∧ (m < 0 ∨ n < m)

let rec merge = function
  | Empty, t → t
  | t, Empty → t
  | Leaf k, t → add k t
  | t, Leaf k → add k t
  | (Branch (p, m, s0, s1) as s), (Branch (q, n, t0, t1) as t) →
      if m ≡ n ∧ match_prefix q p m then
        (* The trees have the same prefix. Merge the subtrees. *)
        Branch (p, m, merge (s0, t0), merge (s1, t1))
      else if unsigned_lt m n ∧ match_prefix q p m then
```

```
        (* q contains p. Merge t with a subtree of s. *)
        if zero_bit q m then
          Branch (p, m, merge (s0, t), s1)
        else
          Branch (p, m, s0, merge (s1, t))
      else if unsigned_lt n m ∧ match_prefix p q n then
        (* p contains q. Merge s with a subtree of t. *)
        if zero_bit p n then
          Branch (q, n, merge (s, t0), t1)
        else
          Branch (q, n, t0, merge (s, t1))
      else
        (* The prefixes disagree. *)
        join (p, s, q, t)

let union s t = merge (s, t)
```

15. When checking if *s1* is a subset of *s2* only two of the above four cases are relevant: when the prefixes are the same and when the prefix of *s1* contains the one of *s2*, and then the recursion is obvious. In the other two cases, the result is false.

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ → true
  | _, Empty → false
  | Leaf k1, _ → mem k1 s2
  | Branch _, Leaf _ → false
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        subset l1 l2 ∧ subset r1 r2
      else if unsigned_lt m2 m1 ∧ match_prefix p1 p2 m2 then
        if zero_bit p1 m2 then
          subset l1 l2 ∧ subset r1 l2
        else
          subset l1 r2 ∧ subset r1 r2
      else
        false
```

16. To compute the intersection and the difference of two sets, we still examine the same four cases as in *merge*. The recursion is then obvious.

```
let rec inter s1 s2 = match (s1, s2) with
  | Empty, _ → Empty
  | _, Empty → Empty
  | Leaf k1, _ → if mem k1 s2 then s1 else Empty
  | _, Leaf k2 → if mem k2 s1 then s2 else Empty
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        merge (inter l1 l2, inter r1 r2)
      else if unsigned_lt m1 m2 ∧ match_prefix p2 p1 m1 then
        inter (if zero_bit p2 m1 then l1 else r1) s2
      else if unsigned_lt m2 m1 ∧ match_prefix p1 p2 m2 then
        inter s1 (if zero_bit p1 m2 then l2 else r2)
      else
        Empty

let rec diff s1 s2 = match (s1, s2) with
  | Empty, _ → Empty
  | _, Empty → s1
  | Leaf k1, _ → if mem k1 s2 then Empty else s1
  | _, Leaf k2 → remove k2 s1
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        merge (diff l1 l2, diff r1 r2)
      else if unsigned_lt m1 m2 ∧ match_prefix p2 p1 m1 then
        if zero_bit p2 m1 then
          merge (diff l1 s2, r1)
        else
          merge (l1, diff r1 s2)
      else if unsigned_lt m2 m1 ∧ match_prefix p1 p2 m2 then
        if zero_bit p1 m2 then diff s1 l2 else diff s1 r2
      else
        s1
```

17. All the following operations (*cardinal*, *iter*, *fold*, *for_all*, *exists*, *filter*, *partition*, *choose*, *elements*) are implemented as for any other kind of binary trees.

```
let rec cardinal = function
  | Empty → 0
  | Leaf _ → 1
  | Branch (_, _, t0, t1) → cardinal t0 + cardinal t1
```

```
let rec iter f = function
  | Empty → ()
  | Leaf k → f k
  | Branch (_, _, t0, t1) → iter f t0; iter f t1

let rec fold f s accu = match s with
  | Empty → accu
  | Leaf k → f k accu
  | Branch (_, _, t0, t1) → fold f t0 (fold f t1 accu)

let rec for_all p = function
  | Empty → true
  | Leaf k → p k
  | Branch (_, _, t0, t1) → for_all p t0 ∧ for_all p t1

let rec exists p = function
  | Empty → false
  | Leaf k → p k
  | Branch (_, _, t0, t1) → exists p t0 ∨ exists p t1

let rec filter pr = function
  | Empty → Empty
  | Leaf k as t → if pr k then t else Empty
  | Branch (p, m, t0, t1) → branch (p, m, filter pr t0, filter pr t1)

let partition p s =
  let rec part (t, f as acc) = function
    | Empty → acc
    | Leaf k → if p k then (add k t, f) else (t, add k f)
    | Branch (_, _, t0, t1) → part (part acc t0) t1
  in
  part (Empty, Empty) s

let rec choose = function
  | Empty → raise Not_found
  | Leaf k → k
  | Branch (_, _, t0, _) → choose t0 (* we know that t0 is non-empty *)

let elements s =
  let rec elements_aux acc = function
    | Empty → acc
    | Leaf k → k :: acc
    | Branch (_, _, l, r) → elements_aux (elements_aux acc l) r
  in
  (* unfortunately there is no easy way to get the elements in ascending order with little-
```

endian Patricia trees *)

```
  List.sort Pervasives.compare (elements_aux [] s)

let split x s =
  let coll k (l, b, r) =
    if k < x then add k l, b, r
    else if k > x then l, b, add k r
    else l, true, r
  in
  fold coll s (Empty, false, Empty)
```

18. There is no way to give an efficient implementation of *min_elt* and *max_elt*, as with binary search trees. The following implementation is a traversal of all elements, barely more efficient than *fold min t* (*choose t*) (resp. *fold max t* (*choose t*)). Note that we use the fact that there is no constructor *Empty* under *Branch* and therefore always a minimal (resp. maximal) element there.

```
let rec min_elt = function
  | Empty → raise Not_found
  | Leaf k → k
  | Branch (_, _, s, t) → min (min_elt s) (min_elt t)

let rec max_elt = function
  | Empty → raise Not_found
  | Leaf k → k
  | Branch (_, _, s, t) → max (max_elt s) (max_elt t)
```

19. Another nice property of Patricia trees is to be independent of the order of insertion. As a consequence, two Patricia trees have the same elements if and only if they are structurally equal.

```
let equal = (=)

let compare = compare
```

20. Additional functions w.r.t to *Set.S*.

```
let rec intersect s1 s2 = match (s1, s2) with
  | Empty, _ → false
  | _, Empty → false
  | Leaf k1, _ → mem k1 s2
  | _, Leaf k2 → mem k2 s1
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
```

```
      intersect l1 l2 ∨ intersect r1 r2
    else if unsigned_lt m1 m2 ∧ match_prefix p2 p1 m1 then
      intersect (if zero_bit p2 m1 then l1 else r1) s2
    else if unsigned_lt m2 m1 ∧ match_prefix p1 p2 m2 then
      intersect s1 (if zero_bit p1 m2 then l2 else r2)
    else
      false
```

21. Big-endian Patricia trees

```
module Big = struct

  type elt = int

  type t_ = t
  type t = t_

  let empty = Empty

  let is_empty = function Empty → true | _ → false

  let singleton k = Leaf k

  let zero_bit k m = (k land m) ≡ 0

  let rec mem k = function
    | Empty → false
    | Leaf j → k ≡ j
    | Branch (_, m, l, r) → mem k (if zero_bit k m then l else r)

  let find k s = if mem k s then k else raise Not_found

  let mask k m = (k lor (m − 1)) land (lnot m)
```

we first write a naive implementation of *highest_bit* only has to work for bytes

```
  let naive_highest_bit x =
    assert (x < 256);
    let rec loop i =
      if i = 0 then 1 else if x lsr i = 1 then 1 lsl i else loop (i − 1)
    in
    loop 7
```

then we build a table giving the highest bit for bytes

```
  let hbit = Array.init 256 naive_highest_bit
```

to determine the highest bit of x we split it into bytes

```
let highest_bit_32 x =
  let n = x lsr 24 in if n ≢ 0 then hbit.(n) lsl 24
  else let n = x lsr 16 in if n ≢ 0 then hbit.(n) lsl 16
  else let n = x lsr 8 in if n ≢ 0 then hbit.(n) lsl 8
  else hbit.(x)

let highest_bit_64 x =
  let n = x lsr 32 in if n ≢ 0 then (highest_bit_32 n) lsl 32
  else highest_bit_32 x

let highest_bit = match Sys.word_size with
  | 32 → highest_bit_32
  | 64 → highest_bit_64
  | _ → assert false

let branching_bit p0 p1 = highest_bit (p0 lxor p1)

let join (p0, t0, p1, t1) =
      let m = branching_bit p0 p1 (*EXP (m t0) (m t1) *) in
  if zero_bit p0 m then
    Branch (mask p0 m, m, t0, t1)
  else
    Branch (mask p0 m, m, t1, t0)

let match_prefix k p m = (mask k m) ≡ p

let add k t =
  let rec ins = function
    | Empty → Leaf k
    | Leaf j as t →
        if j ≡ k then t else join (k, Leaf k, j, t)
    | Branch (p, m, t0, t1) as t →
        if match_prefix k p m then
          if zero_bit k m then
            Branch (p, m, ins t0, t1)
          else
            Branch (p, m, t0, ins t1)
        else
          join (k, Leaf k, p, t)
  in
  ins t

let of_list = List.fold_left (fun s x → add x s) empty
```

```
let remove k t =
  let rec rmv = function
    | Empty → Empty
    | Leaf j as t → if k ≡ j then Empty else t
    | Branch (p, m, t0, t1) as t →
        if match_prefix k p m then
          if zero_bit k m then
            branch (p, m, rmv t0, t1)
          else
            branch (p, m, t0, rmv t1)
        else
          t
  in
  rmv t

let rec merge = function
  | Empty, t → t
  | t, Empty → t
  | Leaf k, t → add k t
  | t, Leaf k → add k t
  | (Branch (p, m, s0, s1) as s), (Branch (q, n, t0, t1) as t) →
      if m ≡ n ∧ match_prefix q p m then
        (* The trees have the same prefix. Merge the subtrees. *)
        Branch (p, m, merge (s0, t0), merge (s1, t1))
      else if unsigned_lt n m ∧ match_prefix q p m then
        (* q contains p. Merge t with a subtree of s. *)
        if zero_bit q m then
          Branch (p, m, merge (s0, t), s1)
        else
          Branch (p, m, s0, merge (s1, t))
      else if unsigned_lt m n ∧ match_prefix p q n then
        (* p contains q. Merge s with a subtree of t. *)
        if zero_bit p n then
          Branch (q, n, merge (s, t0), t1)
        else
          Branch (q, n, t0, merge (s, t1))
      else
        (* The prefixes disagree. *)
        join (p, s, q, t)

let union s t = merge (s, t)
```

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ → true
  | _, Empty → false
  | Leaf k1, _ → mem k1 s2
  | Branch _, Leaf _ → false
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        subset l1 l2 ∧ subset r1 r2
      else if unsigned_lt m1 m2 ∧ match_prefix p1 p2 m2 then
        if zero_bit p1 m2 then
          subset l1 l2 ∧ subset r1 l2
        else
          subset l1 r2 ∧ subset r1 r2
      else
        false

let rec inter s1 s2 = match (s1, s2) with
  | Empty, _ → Empty
  | _, Empty → Empty
  | Leaf k1, _ → if mem k1 s2 then s1 else Empty
  | _, Leaf k2 → if mem k2 s1 then s2 else Empty
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        merge (inter l1 l2, inter r1 r2)
      else if unsigned_lt m2 m1 ∧ match_prefix p2 p1 m1 then
        inter (if zero_bit p2 m1 then l1 else r1) s2
      else if unsigned_lt m1 m2 ∧ match_prefix p1 p2 m2 then
        inter s1 (if zero_bit p1 m2 then l2 else r2)
      else
        Empty

let rec diff s1 s2 = match (s1, s2) with
  | Empty, _ → Empty
  | _, Empty → s1
  | Leaf k1, _ → if mem k1 s2 then Empty else s1
  | _, Leaf k2 → remove k2 s1
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        merge (diff l1 l2, diff r1 r2)
      else if unsigned_lt m2 m1 ∧ match_prefix p2 p1 m1 then
        if zero_bit p2 m1 then
```

```
        merge (diff l1 s2, r1)
      else
        merge (l1, diff r1 s2)
    else if unsigned_lt m1 m2 ∧ match_prefix p1 p2 m2 then
      if zero_bit p1 m2 then diff s1 l2 else diff s1 r2
    else
      s1
```

same implementation as for little-endian Patricia trees

```
let cardinal = cardinal
let iter = iter
let fold = fold
let for_all = for_all
let exists = exists
let filter = filter

let partition p s =
  let rec part (t, f as acc) = function
    | Empty → acc
    | Leaf k → if p k then (add k t, f) else (t, add k f)
    | Branch (_, _, t0, t1) → part (part acc t0) t1
  in
  part (Empty, Empty) s

let choose = choose

let elements s =
  let rec elements_aux acc = function
    | Empty → acc
    | Leaf k → k :: acc
    | Branch (_, _, l, r) → elements_aux (elements_aux acc r) l
  in
  (* we still have to sort because of possible negative elements *)
  List.sort Pervasives.compare (elements_aux [] s)

let split x s =
  let coll k (l, b, r) =
    if k < x then add k l, b, r
    else if k > x then l, b, add k r
    else l, true, r
  in
  fold coll s (Empty, false, Empty)
```

could be slightly improved (when we now that a branch contains only positive or only negative integers)

```
let min_elt = min_elt
let max_elt = max_elt

let equal = (=)

let compare = compare

let make l = List.fold_right add l empty

let rec intersect s1 s2 = match (s1, s2) with
  | Empty, _ → false
  | _, Empty → false
  | Leaf k1, _ → mem k1 s2
  | _, Leaf k2 → mem k2 s1
  | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
      if m1 ≡ m2 ∧ p1 ≡ p2 then
        intersect l1 l2 ∨ intersect r1 r2
      else if unsigned_lt m2 m1 ∧ match_prefix p2 p1 m1 then
        intersect (if zero_bit p2 m1 then l1 else r1) s2
      else if unsigned_lt m1 m2 ∧ match_prefix p1 p2 m2 then
        intersect s1 (if zero_bit p1 m2 then l2 else r2)
      else
        false

end
```

22. Big-endian Patricia trees with non-negative elements only

```
module BigPos = struct

  include Big

  let singleton x = if x < 0 then invalid_arg "BigPos.singleton"; singleton x

  let add x s = if x < 0 then invalid_arg "BigPos.add"; add x s

  let of_list = List.fold_left (fun s x → add x s) empty
```

Patricia trees are now binary search trees!

```
  let rec mem k = function
    | Empty → false
    | Leaf j → k ≡ j
    | Branch (p, _, l, r) → if k ≤ p then mem k l else mem k r
```

```
let rec min_elt = function
  | Empty -> raise Not_found
  | Leaf k -> k
  | Branch (_, _, s, _) -> min_elt s

let rec max_elt = function
  | Empty -> raise Not_found
  | Leaf k -> k
  | Branch (_, _, _, t) -> max_elt t
```

we do not have to sort anymore

```
let elements s =
  let rec elements_aux acc = function
    | Empty -> acc
    | Leaf k -> k :: acc
    | Branch (_, _, l, r) -> elements_aux (elements_aux acc r) l
  in
  elements_aux [] s

end
```

23. EXPERIMENT: Big-endian Patricia trees with swapped bit sign

```
module Bigo = struct

  include Big
```

swaps the sign bit

```
  let swap x = if x < 0 then x land max_int else x lor min_int

  let mem x s = mem (swap x) s

  let add x s = add (swap x) s

  let of_list = List.fold_left (fun s x -> add x s) empty

  let singleton x = singleton (swap x)

  let remove x s = remove (swap x) s

  let elements s = List.map swap (elements s)

  let choose s = swap (choose s)

  let iter f = iter (fun x -> f (swap x))

  let fold f = fold (fun x a -> f (swap x) a)

  let for_all f = for_all (fun x -> f (swap x))
```

```
  let exists f = exists (fun x → f (swap x))

  let filter f = filter (fun x → f (swap x))

  let partition f = partition (fun x → f (swap x))

  let split x s = split (swap x) s

  let rec min_elt = function
    | Empty → raise Not_found
    | Leaf k → swap k
    | Branch (_, _, s, _) → min_elt s

  let rec max_elt = function
    | Empty → raise Not_found
    | Leaf k → swap k
    | Branch (_, _, _, t) → max_elt t

end

let test empty add mem =
  let seed = Random.int max_int in
  Random.init seed;
  let s =
    let rec loop s i =
      if i = 1000 then s else loop (add (Random.int max_int) s) (succ i)
    in
    loop empty 0
  in
  Random.init seed;
  for i = 0 to 999 do assert (mem (Random.int max_int) s) done
```

Interface for module Ptmap

24. Maps over integers implemented as Patricia trees. The following signature is exactly *Map.S* with type *key* = *int*, with the same specifications.

```
include Map.S with type key = int
```

25. Warning: *min_binding* and *max_binding* are linear w.r.t. the size of the map. They are barely more efficient than a straightforward implementation using *fold*.

Module Ptmap

26. Maps of integers implemented as Patricia trees, following Chris Okasaki and Andrew Gill's paper *Fast Mergeable Integer Maps* (`http://www.cs.columbia.edu/~cdo/papers.html#ml98maps`). See the documentation of module *Ptset* which is also based on the same data-structure.

```
type key = int

type α t =
  | Empty
  | Leaf of int × α
  | Branch of int × int × α t × α t

let empty = Empty

let is_empty t = t = Empty

let zero_bit k m = (k land m) ≡ 0

let rec mem k = function
  | Empty → false
  | Leaf (j, _) → k ≡ j
  | Branch (_, m, l, r) → mem k (if zero_bit k m then l else r)

let rec find k = function
  | Empty → raise Not_found
  | Leaf (j, x) → if k ≡ j then x else raise Not_found
  | Branch (_, m, l, r) → find k (if zero_bit k m then l else r)

let find_opt k m = try Some (find k m) with Not_found → None

let lowest_bit x = x land (−x)

let branching_bit p0 p1 = lowest_bit (p0 lxor p1)

let mask p m = p land (m − 1)

let join (p0, t0, p1, t1) =
  let m = branching_bit p0 p1 in
  if zero_bit p0 m then
    Branch (mask p0 m, m, t0, t1)
  else
    Branch (mask p0 m, m, t1, t0)

let match_prefix k p m = (mask k m) ≡ p
```

```
let add k x t =
  let rec ins = function
    | Empty → Leaf (k, x)
    | Leaf (j, _) as t →
        if j ≡ k then Leaf (k, x) else join (k, Leaf (k, x), j, t)
    | Branch (p, m, t0, t1) as t →
        if match_prefix k p m then
          if zero_bit k m then
            Branch (p, m, ins t0, t1)
          else
            Branch (p, m, t0, ins t1)
        else
          join (k, Leaf (k, x), p, t)
  in
  ins t

let singleton k v =
  add k v empty

let branch = function
  | (_, _, Empty, t) → t
  | (_, _, t, Empty) → t
  | (p, m, t0, t1) → Branch (p, m, t0, t1)

let remove k t =
  let rec rmv = function
    | Empty → Empty
    | Leaf (j, _) as t → if k ≡ j then Empty else t
    | Branch (p, m, t0, t1) as t →
        if match_prefix k p m then
          if zero_bit k m then
            branch (p, m, rmv t0, t1)
          else
            branch (p, m, t0, rmv t1)
        else
          t
  in
  rmv t

let rec cardinal = function
  | Empty → 0
  | Leaf _ → 1
  | Branch (_, _, t0, t1) → cardinal t0 + cardinal t1
```

```
let rec iter f = function
  | Empty → ()
  | Leaf (k, x) → f k x
  | Branch (_, _, t0, t1) → iter f t0; iter f t1

let rec map f = function
  | Empty → Empty
  | Leaf (k, x) → Leaf (k, f x)
  | Branch (p, m, t0, t1) → Branch (p, m, map f t0, map f t1)

let rec mapi f = function
  | Empty → Empty
  | Leaf (k, x) → Leaf (k, f k x)
  | Branch (p, m, t0, t1) → Branch (p, m, mapi f t0, mapi f t1)

let rec fold f s accu = match s with
  | Empty → accu
  | Leaf (k, x) → f k x accu
  | Branch (_, _, t0, t1) → fold f t0 (fold f t1 accu)

let rec for_all p = function
  | Empty → true
  | Leaf (k, v) → p k v
  | Branch (_, _, t0, t1) → for_all p t0 ∧ for_all p t1

let rec exists p = function
  | Empty → false
  | Leaf (k, v) → p k v
  | Branch (_, _, t0, t1) → exists p t0 ∨ exists p t1

let rec filter pr = function
  | Empty → Empty
  | Leaf (k, v) as t → if pr k v then t else Empty
  | Branch (p, m, t0, t1) → branch (p, m, filter pr t0, filter pr t1)

let partition p s =
  let rec part (t, f as acc) = function
    | Empty → acc
    | Leaf (k, v) → if p k v then (add k v t, f) else (t, add k v f)
    | Branch (_, _, t0, t1) → part (part acc t0) t1
  in
  part (Empty, Empty) s
```

```
let rec choose = function
  | Empty → raise Not_found
  | Leaf (k, v) → (k, v)
  | Branch (_, _, t0, _) → choose t0 (* we know that t0 is non-empty *)

let split x m =
  let coll k v (l, b, r) =
    if k < x then add k v l, b, r
    else if k > x then l, b, add k v r
    else l, Some v, r
  in
  fold coll m (empty, None, empty)

let rec min_binding = function
  | Empty → raise Not_found
  | Leaf (k, v) → (k, v)
  | Branch (_, _, s, t) →
      let (ks, _) as bs = min_binding s in
      let (kt, _) as bt = min_binding t in
      if ks < kt then bs else bt

let rec max_binding = function
  | Empty → raise Not_found
  | Leaf (k, v) → (k, v)
  | Branch (_, _, s, t) →
      let (ks, _) as bs = max_binding s in
      let (kt, _) as bt = max_binding t in
      if ks > kt then bs else bt

let bindings m =
  fold (fun k v acc → (k, v) :: acc) m []
```

we order constructors as Empty ¡ Leaf ¡ Branch

```
let compare cmp t1 t2 =
  let rec compare_aux t1 t2 = match t1, t2 with
    | Empty, Empty → 0
    | Empty, _ → −1
    | _, Empty → 1
    | Leaf (k1, x1), Leaf (k2, x2) →
        let c = compare k1 k2 in
        if c ≠ 0 then c else cmp x1 x2
    | Leaf _, Branch _ → −1
    | Branch _, Leaf _ → 1
```

```
    | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
        let c = compare p1 p2 in
        if c ≠ 0 then c else
        let c = compare m1 m2 in
        if c ≠ 0 then c else
        let c = compare_aux l1 l2 in
        if c ≠ 0 then c else
        compare_aux r1 r2
  in
  compare_aux t1 t2

let equal eq t1 t2 =
  let rec equal_aux t1 t2 = match t1, t2 with
    | Empty, Empty → true
    | Leaf (k1, x1), Leaf (k2, x2) → k1 = k2 ∧ eq x1 x2
    | Branch (p1, m1, l1, r1), Branch (p2, m2, l2, r2) →
        p1 = p2 ∧ m1 = m2 ∧ equal_aux l1 l2 ∧ equal_aux r1 r2
    | _ → false
  in
  equal_aux t1 t2

let merge f m1 m2 =
  let add m k = function None → m | Some v → add k v m in
  (* first consider all bindings in m1 *)
  let m = fold
    (fun k1 v1 m → add m k1 (f k1 (Some v1) (find_opt k1 m2))) m1 empty in
  (* then bindings in m2 that are not in m1 *)
  fold (fun k2 v2 m → if mem k2 m1 then m else add m k2 (f k2 None (Some v2)))
    m2 m
```

Index