```
(* for ML Lecture 4 *)

(* Definition of the structure MultiSet *)

module MultiSet =
  struct
    exception Empty_Set
    exception Not_Found

    type 'a set = Leaf | Node of 'a * 'a set * 'a set

    let empty = Leaf

    let rec add set elt =
      match set with
        Leaf -> Node(elt, Leaf, Leaf)
      | Node(e, left, right) ->
          if e < elt
          then Node(e, left, add right elt)
          else Node(e, add left elt, right)

    let rec remove_biggest = function
        Leaf -> raise Empty_Set
      | Node(v, left, Leaf) -> (v, left)
      | Node(v, left, right) ->
          let (m, t) = remove_biggest right in
          (m, Node(v, left, t))

    let remove_top = function
        Leaf -> raise Empty_Set
      | Node(_, left, Leaf) -> left
      | Node(_, Leaf, right) -> right
      | Node(_, left, right) ->
          let (t, newl) = remove_biggest left in
          Node(t, newl, right)

    let rec remove set elt = match set with
      Leaf -> raise Not_Found
    | Node(top, left, right) as whole ->
        if top = elt then remove_top whole
        else if (elt < top) then Node(top, remove left elt, right)
        else Node(top, left, remove right elt)

    let rec member set elt = match set with
      Leaf -> false
    | Node(top, left, right) ->
        if top = elt then true
        else if (elt < top) then member left elt
        else member right elt

  end


(*
 > module MultiSet :
 >   sig
 >     exception Empty_Set
 >     exception Not_Found
 >     type 'a set = Leaf | Node of 'a * 'a set * 'a set
 >     val empty : 'a set
 >     val add : 'a set -> 'a -> 'a set
 >     val remove_biggest : 'a set -> 'a * 'a set
```



```
 >     val remove_top : 'a set -> 'a set
 >     val remove : 'a set -> 'a -> 'a set
 >     val member : 'a set -> 'a -> bool
 >   end
*)

(* Definition of the signature MULTISET *)

module type MULTISET =
  sig
    type 'a set (* abstract : implementation is hidden *)
    val empty : 'a set
    val add : 'a set -> 'a -> 'a set
    val remove : 'a set -> 'a -> 'a set
    val member : 'a set -> 'a -> bool
    exception Not_Found
  end

(*
 > module type MULTISET =
 >   sig
 >     type 'a set
 >     val empty : 'a set
 >     val add : 'a set -> 'a -> 'a set
 >     val remove : 'a set -> 'a -> 'a set
 >     val member : 'a set -> 'a -> bool
 >     exception Not_Found
 >   end
*)

(* Applying signature to structure *)

module AbstractMultiSet = (MultiSet : MULTISET)

(*
 > module AbstractMultiSet : MULTISET
*)

(* cf: apply signature at the definition time *)

module type ExampleSig1 =
  sig
    val pub : unit -> int
  end

module Example1 : ExampleSig1 =
  struct
    let pub () = 3
  end

module Example2 : sig
  val pub : unit -> int
end =
  struct
    let pub () = 1
    let hidden () = 2
  end
```

```
(* for ML lecture 4 *)

type comparison = Less | Equal | Greater

(* ORDERED_TYPE : signature for the types with *)
(*                  equality and comparability *)

module type ORDERED_TYPE =
  sig
    type t
    val cmp : t -> t -> comparison
  end

(* MultiSet2: functor from any structure of ORDERED_TYPE *)
(*            to the definition of multiple set of that type *)

module MultiSet2 =
  functor (Elt: ORDERED_TYPE) ->
  struct
    exception Not_Found

    type elem = Elt.t
    type t = Leaf | Node of elem * t * t

    let empty = Leaf

    let rec add set elt =
      match set with
        Leaf -> Node(elt, Leaf, Leaf)
      | Node(e, left, right) as whole ->
          match Elt.cmp e elt with
            Less -> Node(e, left, add right elt)
          | _ -> Node(e, add left elt, right)

    let rec remove_biggest = function
        Leaf -> failwith "remove_biggest"
      | Node(v, left, Leaf) -> (v, left)
      | Node(v, left, right) ->
          let (m, t) = remove_biggest right in
          (m, Node(v, left, t))

    let remove_top = function
        Leaf -> failwith "remove_top"
      | Node(_, left, Leaf) -> left
      | Node(_, Leaf, right) -> right
      | Node(_, left, right) ->
          let (t, newl) = remove_biggest left in
          Node(t, newl, right)

    let rec remove set elt = match set with
      Leaf -> raise Not_Found
    | Node(top, left, right) as whole ->
        match Elt.cmp elt top with
          Equal -> remove_top whole
        | Less -> Node(top, remove left elt, right)
        | Greater -> Node(top, left, remove right elt)

    let rec member set elt = match set with
      Leaf -> false
    | Node(top, left, right) ->
        match Elt.cmp elt top with
          Equal -> true
        | Less -> member left elt
        | Greater -> member right elt
  end

(*
 > module MultiSet2 :
 >   functor (Elt : ORDERED_TYPE) ->
 >     sig
 >       exception Not_Found
 >       type elem = Elt.t
 >       and t = Leaf | Node of elem * t * t
 >       val empty : t
 >       val add : t -> elem -> t
 >       val remove_top : t -> t
 >       val remove : t -> Elt.t -> t
 >       val member : t -> Elt.t -> bool
 >     end
 >
*)

(* OrderedString : string with normal ordering *)

module OrderedString =
  struct
    type t = string
    let cmp x y =
      if x = y then Equal
      else if x < y then Less
      else Greater
  end

(*
 > module OrderedString :
 >   sig type t = string val cmp : 'a -> 'a -> comparison end
*)

(* StringSet: multiple set for string *)

module StringSet = MultiSet2(OrderedString)
(*
 > module StringSet :
 >   sig
 >     exception Not_Found
 >     type elem = OrderedString.t
 >     and t = MultiSet2(OrderedString).t = Leaf | Node of elem * t * t
 >     val empty : t
 >     val add : t -> elem -> t
 >     val remove_top : t -> t
 >     val remove : t -> OrderedString.t -> t
 >     val member : t -> OrderedString.t -> bool
 >   end
*)

(* SETFUNCTOR : signature for the functor MultiSet2 *)

module type SETFUNCTOR =
    functor (Elt: ORDERED_TYPE) ->
  sig
    type elem = Elt.t (* not abstract *)
    type t            (* abstract *)
    val empty : t
    val add : t -> elem -> t
```

```
    val remove : t -> elem -> t
    val member : t -> elem -> bool
    exception Not_Found
  end

(* AbstractSet2 : abstract version of MultiSet2 *)

module AbstractSet2 = (MultiSet2 : SETFUNCTOR)

(*
 > module AbstractSet2 : SETFUNCTOR
*)

module AbstractStringSet = AbstractSet2(OrderedString)
(*
 > module AbstractStringSet :
 >   sig
 >     type elem = OrderedString.t
 >     and t = AbstractSet2(OrderedString).t
 >     val empty : t
 >     val add : t -> elem -> t
 >     val remove : t -> elem -> t
 >     val member : t -> elem -> bool
 >     exception Not_Found
 >   end
*)

(* NCString: case-ignoring string *)

module NCString =
  struct
    open String
    type t = string
    let cmp x y =
      let (x', y') = (lowercase x, lowercase y) in
      if x' = y' then Equal
      else if x' < y' then Less
      else Greater
  end
(*
 > module NCString :
 >   sig type t = string val cmp : string -> string -> comparison end
*)

module NCStringSet = AbstractSet2(NCString)
(*
 > module NCStringSet :
 >   sig
 >     type elem = NCString.t
 >     and t = AbstractSet2(NCString).t
 >     val empty : t
 >     val add : t -> elem -> t
 >     val remove : t -> elem -> t
 >     val member : t -> elem -> bool
 >     exception Not_Found
 >   end
*)
```



```
(* interface definition for multiSet.ml *)

type 'a set (* abstract : implementation is hidden  *)

val empty : 'a set
val add : 'a set -> 'a -> 'a set
val member_count : 'a set -> 'a -> int
val to_list : 'a set -> ('a * int) list
```

Apr 26, 04 23:10 **mySet.ml** Page 1/1

```
(* Implementation for multiset *)

exception Empty_Set
exception Not_Found

type 'a set = Leaf | Node of 'a * int * 'a set * 'a set

let empty = Leaf

let rec add set elt =
  match set with
    Leaf -> Node(elt, 1, Leaf, Leaf)
  |     Node(e, cnt, left, right) as whole ->
      if e = elt then Node(e, cnt + 1, left, right)
      else if e < elt then Node(e, cnt, left, add right elt)
      else Node(e, cnt, add left elt, right)

let rec member_count set elt = match set with
  Leaf -> 0
| Node(top, cnt, left, right) ->
    if top = elt then cnt
    else if (elt < top) then member_count left elt
    else member_count right elt

let rec remove_top = function
    Leaf -> raise Empty_Set
  | Node(_, _, left, Leaf) -> left
  | Node(_, _, Leaf, right) -> right
  | Node(_, _, (Node(ltop, lcnt, _, _) as left),
         (Node(rtop, rcnt, _, _) as right)) ->
           if (ltop < rtop)
           then Node(ltop, lcnt, remove_top left, right)
           else Node(rtop, rcnt, left, remove_top right)

let rec remove set elt = match set with
  Leaf -> raise Not_Found
| Node(top, cnt, left, right) as whole ->
    if top = elt then begin
      if cnt > 1 then
        Node(top, cnt - 1, left, right)
      else
        remove_top whole
    end
    else if (elt < top) then Node(top, cnt, remove left elt, right)
    else Node(top, cnt, left, remove right elt)

let rec to_list l = function
    Leaf -> l
  | Node(top, cnt, left, right) -> to_list ((top,cnt)::(to_list l right)) left

let to_list set = to_list [] set
```

Apr 26, 04 23:10 **uniq.ml** Page 1/1

```
open MySet

let s = ref empty

let main () =
  try
    while true do
      s := add !s (read_line ())
    done
  with
    End_of_file ->
      let l = (to_list !s) in
      let rec iter = function
          [] -> ()
        | (item, count)::tl -> (
            Printf.printf "%7d %s" count item;
            print_newline ();
            iter tl
          ) in
      iter l

let _ = main ()
```

```
all: mySet.cmo myuniq

mySet.cmo: mySet.ml mySet.cmi
	ocamlc -c mySet.ml

mySet.cmi: mySet.mli
	ocamlc -c mySet.mli

uniq.cmo: uniq.ml
	ocamlc -c uniq.ml

myuniq: mySet.cmo uniq.cmo
	ocamlc -o myuniq mySet.cmo uniq.cmo

clean:
	rm -f *.cmo *.cmi myuniq
```



```
(* For ML lecture 4 *)

(* Question 2: simple bank account module *)

module BankImpl =
  struct
    type account = string * int ref
    exception Underflow
    exception InvalidPassword
    exception InvalidOperation

    let accounts = ref []
    let banktreasure = ref 10000

    let do_cashier (_, cur : account) amount =
      cur := !cur + amount;
      banktreasure := !banktreasure + amount

    let pass_check (p, _ : account) pass =
      if (p = pass) then () else raise InvalidPassword

    let balance_int (_, cur : account) =
      !cur

    let new_account password initial =
      if initial <= 0 then raise InvalidOperation else begin
        let instance = (password, ref 0 : account) in
        accounts := instance :: !accounts;
        do_cashier instance initial;
        instance
      end

    let balance acc pass =
      pass_check acc pass;
      balance_int acc

    let deposit acc amount =
      if amount <= 0 then raise InvalidOperation
      else do_cashier acc amount

    let withdraw acc pass amount =
      if amount <= 0 then raise InvalidOperation
      else begin
        pass_check acc pass;
        if balance_int acc < amount then raise Underflow
        else do_cashier acc (-amount)
      end

    let bank_statistics () =
      List.length !accounts, !banktreasure
  end

(*
  # open BankImpl;;
  # let a1 = new_account "pass1" 50;;
  val a1 : BankImpl.account = ("pass1", {contents = 50})
  # bank_statistics ();;
  - : int * int = (1, 10050)
  # withdraw a1 "foobar" 30;;
  Exception: BankImpl.InvalidPassword.
  # withdraw a1 "pass1" 30;;
  - : unit = ()
```

```
  # withdraw a1 "pass1" 100;;
  Exception: BankImpl.Underflow.
  # deposit a1 50;;
  - : unit = ()
  # balance a1 "pass1";;
  - : int = 70
  # let a2 = new_account "pass2" 1000;;
  val a2 : BankImpl.account = ("pass2", {contents = 1000})
  # bank_statistics ();;
  - : int * int = (2, 11070)
*)
```