```
module ListDict = struct
type ('a,'b) dict = Dict of ('a * 'b) list
let empty = Dict []
let insert key value old_dict = match old_dict with | Dict xs -> Dict ((key,value) :: xs)
let rec remove key old_dict = match old_dict with | Dict [] -> Dict [] | Dict ((a , b) :: xs)
when key = a -> Dict xs | Dict ((a,b) :: xs) -> remove key (Dict xs)
let rec find key old_dict = match old_dict with | Dict [] -> failwith "!" | Dict ((a,b) :: xs)
when a = key -> b | Dict((a,b) :: xs) -> find key (Dict xs)
let rec find_opt key old_dict = match old_dict with |(Dict []) -> None | Dict((a,b) :: xs) when a
= key -> Some b | Dict((a,b) :: xs) -> find_opt key (Dict xs)
let to_list old_dict = match old_dict with | Dict xs -> xs
end;;


module type KDICT = sig
type key
type 'b dict
val empty : 'b dict
val insert : key -> 'b -> 'b dict -> 'b  dict
val remove : key ->  'b  dict ->  'b  dict
val find_opt : key -> 'b dict -> 'b option
val find : key ->  'b  dict -> 'b
val to_list : 'b dict -> (key * 'b ) list
end

module MakeListDict (M : Map.OrderedType) = struct
type key = M.t
type 'b dict = Dict of  (key*'b) list
let empty = Dict []
let insert key value old_dict = match old_dict with | Dict xs -> Dict ((key,value) :: xs)
let rec remove key old_dict = match old_dict with | Dict [] -> Dict [] | Dict ((a , b) :: xs)
when key = a -> Dict xs | Dict ((a,b) :: xs) -> remove key (Dict xs)
let rec find key old_dict = match old_dict with | Dict [] -> failwith "!" | Dict ((a,b) :: xs)
when a = key -> b | Dict((a,b) :: xs) -> find key (Dict xs)
let rec find_opt key old_dict = match old_dict with |(Dict []) -> None | Dict((a,b) :: xs) when a
= key -> Some b | Dict((a,b) :: xs) -> find_opt key (Dict xs)
let to_list old_dict = match old_dict with | Dict xs -> xs
end;;

module CharListDict : (KDICT with type key = char) = MakeListDict (struct type t = char;; let
compare a b = Char.compare a b;;  end);;

module MakeMapDict (M : Map.OrderedType) = struct
type key = M.t
module KeyMap = Map.Make(M)
type 'b dict = Dict of 'b KeyMap.t
let  empty = Dict KeyMap.empty
let insert key value old_dict = match old_dict with | Dict old_map -> Dict (KeyMap.add key value
old_map)
let remove key old_dict =  match old_dict with | Dict old_map ->  Dict (KeyMap.remove key
old_map)
let find key old_dict =   match old_dict with | Dict old_map -> KeyMap.find key old_map
let find_opt key old_dict =   match old_dict with | Dict old_map -> KeyMap.find_opt key old_map
let to_list old_dict = match old_dict with | Dict old_map -> KeyMap.bindings old_map
end;;

module CharMapDict : (KDICT with type key = char) = MakeMapDict(Char);;

module Freq (D : KDICT) = struct
let freq (xs : D.key list) : (D.key * int) list = let rec build ys = match ys with | [] ->
D.empty | y :: ys' -> let u = build ys' in (match (D.find_opt y u) with | None -> D.insert y 1 u
| Some c -> D.remove y u |> D.insert y (c+1)) in D.to_list (build xs)
end;;

module CharFreq = Freq (CharMapDict);;
let list_of_string s = String.to_seq s |> List.of_seq;;

let char_freq str = CharFreq.freq (list_of_string str);;

module LeftistHeap = struct
type ('a , 'b ) heap =
| HLeaf
| HNode of int * ('a , 'b ) heap * 'a * 'b * ('a , 'b ) heap
let rank = function HLeaf -> 0 | HNode (n , _ , _ , _ , _ ) -> n
let heap_ordered p = function
| HLeaf -> true
| HNode (_ , _ , p', _ , _ ) -> p <= p'
let rec is_valid = function
| HLeaf -> true
| HNode (n , l , p , v , r ) ->
rank r <= rank l
&& rank r + 1 = n
&& heap_ordered p l
&& heap_ordered p r
&& is_valid l
&& is_valid r
let make_node p v l r = if rank l >= rank r then HNode((1 + rank r), l , p, v, r) else
HNode((1+rank l),r,p,v,l)
let rec heap_merge h1 h2 = match (h1, h2) with | (HLeaf, h2) -> h2 | (h1, HLeaf) -> h1 |
(HNode(n1, l1, p1, e1, r1), HNode(n2, l2, p2, e2, r2)) -> if p1 > p2 then make_node p2 e2 l2
(heap_merge r2 h1) else make_node p1 e1 l1 (heap_merge r1 h2)
end;;

module type PRIO_QUEUE = sig
  type ('a, 'b) pq
  val empty : ('a, 'b) pq
  val insert : 'a -> 'b -> ('a, 'b) pq -> ('a, 'b) pq
  val pop : ('a, 'b) pq -> ('a, 'b) pq
  val min : ('a, 'b) pq -> 'b
  val min_prio : ('a, 'b) pq -> 'a
end

module Leftist_Pq : PRIO_QUEUE = struct
type ('a,'b) pq = ('a,'b) LeftistHeap.heap
let empty = LeftistHeap.HLeaf
let insert a b old_heap = LeftistHeap.(heap_merge (make_node a b HLeaf HLeaf) old_heap)
let pop old_heap = LeftistHeap.(match old_heap with | HLeaf -> failwith "!" |
HNode(n1,l1,p1,v1,r1) -> heap_merge l1 r1)
let min old_heap =LeftistHeap.(match old_heap with | HLeaf -> failwith "!" |
HNode(n1,l1,p1,v1,r1) ->  v1)
let min_prio old_heap =LeftistHeap.(match old_heap with | HLeaf -> failwith "!" |
HNode(n1,l1,p1,v1,r1) ->  p1)
end;;

module PqSort (Pq : PRIO_QUEUE) = struct
let pqsort xs = let rec build xs = match xs with | [] -> Pq.empty | y :: ys -> Pq.insert y y
(build ys) in let rec debuild prio = Pq.(match prio with | a when a = empty -> [] | a -> min prio
:: debuild (pop prio)) in debuild (build xs)
end;;


module LeftistPqSort = PqSort(Leftist_Pq);;
```