1. Priority Queue (Cola de Prioridad)

Ejercicio 1: Implementación de Priority Queue con Listas

Se implementará el TAD PriorityQueue utilizando una lista ordenada de menor a mayor como representación.

Haskell

```
module PriorityQueue
 ( PriorityQueue
 , emptyPQ
 , isEmptyPQ
 , insertPQ
 , findMinPQ
 , deleteMinPQ
 )
where

-- INVARIANTE DE REPRESENTACIÓN: La lista interna se mantiene siempre ordenada
-- de menor a mayor.
data PriorityQueue a = PQ [a] deriving Show

-- Propósito: Devuelve una priority queue vacía.
-- Costo: O(1).
emptyPQ :: PriorityQueue a
emptyPQ = PQ []

-- Propósito: Indica si la priority queue está vacía.
-- Costo: O(1).
isEmptyPQ :: PriorityQueue a -> Bool
isEmptyPQ (PQ xs) = null xs

-- Propósito: Inserta un elemento en la priority queue, manteniendo el orden.
-- Costo: O(n), donde n es la cantidad de elementos en la cola. En el peor caso,
-- debe recorrer toda la lista para insertar el elemento al final.
insertPQ :: Ord a => a -> PriorityQueue a -> PriorityQueue a
insertPQ x (PQ xs) = PQ (insertarOrdenado x xs)

-- Subtarea para insertar un elemento en una lista ordenada.
insertarOrdenado :: Ord a => a -> [a] -> [a]
insertarOrdenado x [] = [x]
insertarOrdenado x (y:ys) =
 if x <= y
  then x : y : ys
  else y : insertarOrdenado x ys

-- Propósito: Devuelve el elemento más prioritario (el mínimo) de la priority queue.
-- Precondición: La priority queue no puede estar vacía.
-- Costo: O(1), ya que el mínimo es siempre el primer elemento de la lista.
findMinPQ :: Ord a => PriorityQueue a -> a
findMinPQ (PQ xs) =
 if null xs
  then error "La Priority Queue está vacía."
  else head xs

-- Propósito: Devuelve una priority queue sin el elemento más prioritario (el mínimo).
-- Precondición: La priority queue no puede estar vacía.
-- Costo: O(1), solo necesita devolver la cola de la lista.
deleteMinPQ :: Ord a => PriorityQueue a -> PriorityQueue a
```

```
deleteMinPQ (PQ xs) =
 if null xs
  then error "La Priority Queue está vacía."
  else PQ (tail xs)
```

Ejercicio 2: heapSort

Se implementa la función de ordenamiento heapSort utilizando la PriorityQueue anterior como estructura auxiliar.

Haskell

```
-- Se asume la importación del módulo PriorityQueue.
import PriorityQueue

-- Propósito: Dada una lista, la ordena de menor a mayor.
-- Costo: O(n^2), donde n es la longitud de la lista de entrada.
-- Justificación:
-- 1. `listaAPq`: Para construir la PQ, se insertan n elementos. Cada inserción
--    cuesta O(i), donde i es el tamaño actual de la PQ. La suma de 1 a n
--    da un costo total de O(n^2).
-- 2. `pqALista`: Se extraen n elementos. Cada extracción combina findMinPQ (O(1))
--    y deleteMinPQ (O(1)). Esto se repite n veces, dando un costo de O(n).
-- El costo total es la suma de ambos pasos: O(n^2) + O(n) = O(n^2).
heapSort :: Ord a => [a] -> [a]
heapSort xs = pqALista (listaAPq xs)

-- Subtarea para convertir una lista en una Priority Queue.
listaAPq :: Ord a => [a] -> PriorityQueue a
listaAPq []     = emptyPQ
listaAPq (x:xs) = insertPQ x (listaAPq xs)

-- Subtarea para convertir una Priority Queue en una lista ordenada.
pqALista :: Ord a => PriorityQueue a -> [a]
pqALista pq =
 if isEmptyPQ pq
  then []
  else findMinPQ pq : pqALista (deleteMinPQ pq)
```

---

2. Map (Diccionario)

Ejercicio 4: Implementación de Map

Primero, se implementa el TAD Map según la variante 1, que será utilizada por las funciones de usuario.

Haskell

```
module Map
 ( Map
 , emptyM
 , assocM
 , lookupM
 , deleteM
 , keys
 )
where

-- INVARIANTE DE REPRESENTACIÓN: La lista de pares no contiene claves repetidas.
data Map k v = M [(k, v)] deriving Show

-- Propósito: Devuelve un map vacío.
-- Costo: O(1).
emptyM :: Map k v
```

```
emptyM = M []

-- Propósito: Agrega una asociación clave-valor al map. Si la clave ya existe,
-- su valor es reemplazado.
-- Costo: O(n), donde n es la cantidad de claves. `filter` recorre toda la lista.
assocM :: Eq k => k -> v -> Map k v -> Map k v
assocM k v (M kvs) = M ((k, v) : filter (\(key, _) -> key /= k) kvs)

-- Propósito: Encuentra un valor dado una clave.
-- Costo: O(n), en el peor caso `lookup` recorre toda la lista.
lookupM :: Eq k => k -> Map k v -> Maybe v
lookupM k (M kvs) = lookup k kvs

-- Propósito: Borra una asociación dada una clave.
-- Costo: O(n), `filter` recorre toda la lista.
deleteM :: Eq k => k -> Map k v -> Map k v
deleteM k (M kvs) = M (filter (\(key, _) -> key /= k) kvs)

-- Propósito: Devuelve las claves del map.
-- Costo: O(n), `clavesL` recorre toda la lista de pares.
keys :: Map k v -> [k]
keys (M kvs) = clavesL kvs

-- Subtarea para obtener las claves de una lista de pares.
clavesL :: [(k, v)] -> [k]
clavesL [] = []
clavesL ((k,v):kvs) = k : clavesL kvs
```

Ejercicio 3: Funciones como Usuario de Map

Estas funciones se implementan utilizando la interfaz del TAD Map definido anteriormente.

Haskell

```
-- Se asume la importación del módulo Map.
import Map

-- 1. values
-- Propósito: Obtiene los valores asociados a cada clave del map.
-- Costo: O(n^2). `keys` cuesta O(n). Por cada una de las n claves, se hace un
-- `lookupM` que cuesta O(n). El costo total es n * O(n) = O(n^2).
values :: Eq k => Map k v -> [Maybe v]
values m = lookups (keys m) m

-- Subtarea para buscar una lista de claves en un map.
lookups :: Eq k => [k] -> Map k v -> [Maybe v]
lookups [] _ = []
lookups (k:ks) m = lookupM k m : lookups ks m

-- 2. todasAsociadas
-- Propósito: Indica si en el map se encuentran todas las claves dadas.
-- Costo: O(n*m), donde n es la longitud de la lista de claves y m es el tamaño
-- del map. Por cada una de las n claves se hace un lookupM de costo O(m).
todasAsociadas :: Eq k => [k] -> Map k v -> Bool
todasAsociadas [] _ = True
todasAsociadas (k:ks) m =
  let maybeV = lookupM k m
  in estaDefinido maybeV && todasAsociadas ks m
```

```
-- Subtarea para chequear si un Maybe tiene un valor.
estaDefinido :: Maybe a -> Bool
estaDefinido Nothing = False
estaDefinido (Just _) = True

-- 3. listToMap
-- Propósito: Convierte una lista de pares clave-valor en un map.
-- Costo: O(n^2), donde n es la longitud de la lista. Se hacen n llamadas a
-- `assocM`, y cada una cuesta O(i) donde i es el tamaño actual del map.
listToMap :: Eq k => [(k, v)] -> Map k v
listToMap [] = emptyM
listToMap ((k,v):kvs) = assocM k v (listToMap kvs)

-- 4. mapToList
-- Propósito: Convierte un map en una lista de pares clave-valor.
-- Costo: O(n^2). `keys` cuesta O(n). Por cada clave, se realiza un `lookupM`
-- (costo O(n)) para obtener el valor. Total: n * O(n) = O(n^2).
mapToList :: Eq k => Map k v -> [(k, v)]
mapToList m = armarPares (keys m) m

-- Subtarea para construir la lista de pares.
armarPares :: Eq k => [k] -> Map k v -> [(k,v)]
armarPares [] _ = []
armarPares (k:ks) m =
 let Just v = lookupM k m -- Asumimos que la clave siempre existe.
 in (k, v) : armarPares ks m

-- 5. agruparEq
-- Propósito: Agrupa los valores de los pares que comparten la misma clave.
-- Costo: O(n^2), donde n es la longitud de la lista. Por cada par (n pares),
-- se hace un lookupM (O(i)) y un assocM (O(i)).
agruparEq :: Eq k => [(k, v)] -> Map k [v]
agruparEq [] = emptyM
agruparEq ((k,v):kvs) =
 let mapAgrupado = agruparEq kvs
     maybeVs = lookupM k mapAgrupado
 in agruparPar (k,v) maybeVs mapAgrupado

-- Subtarea que decide cómo asociar el nuevo par.
agruparPar :: Eq k => (k, v) -> Maybe [v] -> Map k [v] -> Map k [v]
agruparPar (k,v) Nothing m = assocM k [v] m
agruparPar (k,v) (Just vs) m = assocM k (v:vs) m
```

Ejercicio 5: Más Funciones como Usuario de Map

```
Haskell
-- Se asume la importación del módulo Map.
import Map

-- indexar
-- Propósito: Relaciona cada elemento de una lista con su posición.
-- Costo: O(n^2), donde n es la longitud de la lista. La subtarea `indexarDesde`
-- hace n llamadas a `assocM`, cada una con costo O(i).
indexar :: [a] -> Map Int a
indexar xs = indexarDesde 0 xs emptyM

-- Subtarea para indexar recursivamente con un map acumulador.
```

```
indexarDesde :: Int -> [a] -> Map Int a -> Map Int a
indexarDesde _ [] m = m
indexarDesde i (x:xs) m =
 let mMap = assocM i x m
 in indexarDesde (i+1) xs mMap

-- ocurrencias
-- Propósito: Cuenta las ocurrencias de cada caracter en un string.
-- Costo: O(n^2), donde n es la longitud del String. Por cada caracter (n),
-- se realiza un `lookupM` (O(i)) y un `assocM` (O(i)).
ocurrencias :: String -> Map Char Int
ocurrencias s = contarOcurrencias s emptyM

-- Subtarea para contar recursivamente con un map acumulador.
contarOcurrencias :: String -> Map Char Int -> Map Char Int
contarOcurrencias "" m = m
contarOcurrencias (c:cs) m =
 let maybeCount = lookupM c m
 in contarOcurrencias cs (actualizarConteo c maybeCount m)

-- Subtarea para actualizar el conteo de un caracter.
actualizarConteo :: Char -> Maybe Int -> Map Char Int -> Map Char Int
actualizarConteo c Nothing m = assocM c 1 m
actualizarConteo c (Just n) m = assocM c (n+1) m
```

---

3. MultiSet (Multiconjunto)
Ejercicio 6: Implementación y Uso de MultiSet
1. Implementar MultiSet usando Map

Haskell

```
-- Se asume la importación del módulo Map.
import Map

-- Representación: Se usa un Map donde la clave es el elemento y el valor es
-- el número de ocurrencias.
data MultiSet a = MS (Map a Int) deriving Show

-- Propósito: Denota un multiconjunto vacío.
-- Costo: O(1), el costo de emptyM.
emptyMS :: MultiSet a
emptyMS = MS emptyM

-- Propósito: Agrega una ocurrencia de un elemento al multiconjunto.
-- Costo: O(n), donde n es el número de elementos distintos en el multiset.
-- El costo está dominado por `lookupM` y `assocM` del Map subyacente.
addMS :: (Ord a, Eq a) => a -> MultiSet a -> MultiSet a
addMS x (MS m) =
 let maybeCount = lookupM x m
 in MS (actualizarConteo x maybeCount m)

-- Subtarea de la práctica anterior reutilizada.
actualizarConteo :: Eq a => a -> Maybe Int -> Map a Int -> Map a Int
actualizarConteo c Nothing m = assocM c 1 m
actualizarConteo c (Just n) m = assocM c (n+1) m

-- Propósito: Indica la cantidad de apariciones de un elemento.
```

```
-- Costo: O(n), el costo de `lookupM`.
ocurrencesMS :: (Ord a, Eq a) => a -> MultiSet a -> Int
ocurrencesMS x (MS m) =
  let maybeCount = lookupM x m
  in obtenerConteo maybeCount

-- Subtarea para devolver 0 si el elemento no existe.
obtenerConteo :: Maybe Int -> Int
obtenerConteo Nothing = 0
obtenerConteo (Just n) = n

-- Propósito: Devuelve una lista con los elementos y su cantidad de ocurrencias.
-- Costo: O(n^2), el costo de `mapToList`.
multiSetToList :: (Ord a, Eq a) => MultiSet a -> [(a, Int)]
multiSetToList (MS m) = mapToList m
```

2. Reimplementar ocurrencias como usuario de MultiSet

Haskell

```
-- Se asume la importación del módulo MultiSet.
import MultiSet

-- Propósito: Dado un string, devuelve un multiconjunto con las ocurrencias
-- de cada caracter.
-- Costo: O(n^2), donde n es la longitud del string. Se realizan n llamadas a
-- `addMS`, y cada una cuesta O(i), siendo i el número de caracteres
-- distintos encontrados hasta el momento.
ocurrenciasMultiSet :: String -> MultiSet Char
ocurrenciasMultiSet "" = emptyMS
ocurrenciasMultiSet (c:cs) = addMS c (ocurrenciasMultiSet cs)
```