1. Stack (Pila)
Implementación utilizando una lista. Las operaciones push, pop y top actúan sobre la cabeza de la lista, lo que les da una eficiencia de O(1).
Haskell

```
module Stack (Stack, emptyS, isEmptyS, push, top, pop, lenS) where

-- Representación: Se usa una lista de elementos 'a'.
data Stack a = Stk [a] deriving Show

-- Crea una pila vacía.
emptyS :: Stack a
emptyS = Stk []

-- Verifica si la pila está vacía.
isEmptyS :: Stack a -> Bool
isEmptyS (Stk []) = True
isEmptyS _      = False

-- Agrega un elemento a la pila.
push :: a -> Stack a -> Stack a
push x (Stk xs) = Stk (x:xs)

-- Devuelve el elemento en el tope de la pila (sin quitarlo).
-- Precondición: La pila no puede estar vacía.
top :: Stack a -> a
top (Stk (x:_)) = x
top (Stk [])    = error "La pila está vacía."

-- Quita el elemento del tope de la pila.
-- Precondición: La pila no puede estar vacía.
pop :: Stack a -> Stack a
pop (Stk (_:xs)) = Stk xs
pop (Stk [])     = error "La pila está vacía."

-- Devuelve la cantidad de elementos en la pila. (Costo O(n))
lenS :: Stack a -> Int
lenS (Stk xs) = length xs
```

---

2. Queue (Cola)
Implementación utilizando dos listas para mayor eficiencia. Una lista (front) se usa para dequeue y la otra (back) para enqueue. Cuando front se vacía, se revierte back y se pasa a front, logrando un costo amortizado eficiente.
Haskell

```
module Queue (Queue, emptyQ, isEmptyQ, enqueue, firstQ, dequeue) where

-- Representación: Dos listas, una para el frente y otra para el fondo.
-- Invariante de representación: Si la lista 'front' está vacía,
-- entonces la lista 'back' también debe estarlo.
data Queue a = Q [a] [a] deriving Show
```

```haskell
-- Crea una cola vacía.
emptyQ :: Queue a
emptyQ = Q [] []

-- Verifica si la cola está vacía.
isEmptyQ :: Queue a -> Bool
isEmptyQ (Q [] []) = True
isEmptyQ _       = False

-- Agrega un elemento al final de la cola.
enqueue :: a -> Queue a -> Queue a
enqueue x (Q front back) = Q front (x:back)

-- Devuelve el primer elemento de la cola.
-- Precondición: La cola no puede estar vacía.
firstQ :: Queue a -> a
firstQ (Q (x:_) _) = x
firstQ (Q [] back) =
 if null back
  then error "La cola está vacía."
  else last back

-- Quita el primer elemento de la cola.
-- Precondición: La cola no puede estar vacía.
dequeue :: Queue a -> Queue a
dequeue (Q (_:xs) back) =
 if null xs && not (null back)
  then Q (reverse back) []
  else Q xs back
dequeue (Q [] []) = error "La cola está vacía."
```

---

3. Set (Conjunto)

Implementación con una lista que mantiene el invariante de no tener elementos repetidos. Esto hace que la operación adds tenga un costo de O(n) porque debe verificar la preexistencia del elemento.

Haskell

```haskell
module Set (Set, emptyS, addS, belongs, sizeS, removeS, unionS, setToList) where

-- Representación: Una lista sin elementos repetidos.
data Set a = S [a] deriving Show

-- Crea un conjunto vacío.
emptyS :: Set a
emptyS = S []

-- Agrega un elemento al conjunto, si no existe ya.
addS :: Eq a => a -> Set a -> Set a
addS x (S xs) =
 if x `elem` xs
  then S xs
  else S (x:xs)
```

```
-- Verifica si un elemento pertenece al conjunto.
belongs :: Eq a => a -> Set a -> Bool
belongs x (S xs) = x `elem` xs

-- Devuelve la cantidad de elementos.
sizeS :: Set a -> Int
sizeS (S xs) = length xs

-- Elimina un elemento del conjunto.
removeS :: Eq a => a -> Set a -> Set a
removeS x (S xs) = S (filter (/= x) xs)

-- Realiza la unión de dos conjuntos.
unionS :: Eq a => Set a -> Set a -> Set a
unionS (S xs) (S ys) = S (xs ++ filter (`notElem` xs) ys)

-- Convierte el conjunto a una lista.
setToList :: Set a -> [a]
setToList (S xs) = xs
```

---

4. Priority Queue (Cola de Prioridad)

Implementación utilizando una lista ordenada. findMinPQ es O(1), pero insertPQ es O(n) ya que debe mantener el orden de la lista.

Haskell

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

-- Representación: Una lista ordenada de menor a mayor.
data PriorityQueue a = PQ [a] deriving Show

-- Crea una cola de prioridad vacía.
emptyPQ :: PriorityQueue a
emptyPQ = PQ []

-- Verifica si la cola está vacía.
isEmptyPQ :: PriorityQueue a -> Bool
isEmptyPQ (PQ []) = True
isEmptyPQ _      = False

-- Inserta un elemento manteniendo el orden.
insertPQ :: Ord a => a -> PriorityQueue a -> PriorityQueue a
insertPQ x (PQ xs) = PQ (insert' x xs)
 where
  insert' e [] = [e]
  insert' e (y:ys) =
   if e <= y
     then e : y : ys
     else y : insert' e ys
```

```
-- Devuelve el elemento con mayor prioridad (el mínimo).
-- Precondición: La cola no puede estar vacía.
findMinPQ :: Ord a => PriorityQueue a -> a
findMinPQ (PQ (x:_)) = x
findMinPQ (PQ [])    = error "La cola de prioridad está vacía."

-- Elimina el elemento con mayor prioridad.
-- Precondición: La cola no puede estar vacía.
deleteMinPQ :: Ord a => PriorityQueue a -> PriorityQueue a
deleteMinPQ (PQ (_:xs)) = PQ xs
deleteMinPQ (PQ [])     = error "La cola de prioridad está vacía."
```

---

5. Map (Diccionario)

Implementación con una lista de tuplas (clave, valor), sin claves repetidas. Las operaciones principales tienen un costo de O(n).

Haskell

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

-- Representación: Una lista de pares (clave, valor) sin claves repetidas.
data Map k v = M [(k, v)] deriving Show

-- Crea un mapa vacío.
emptyM :: Map k v
emptyM = M []

-- Asocia una clave a un valor. Si la clave ya existe, actualiza el valor.
assocM :: Eq k => k -> v -> Map k v -> Map k v
assocM k v (M kvs) = M ((k, v) : filter ((/= k) . fst) kvs)

-- Busca un valor a partir de una clave.
lookupM :: Eq k => k -> Map k v -> Maybe v
lookupM k (M kvs) = lookup k kvs

-- Elimina una clave y su valor asociado.
deleteM :: Eq k => k -> Map k v -> Map k v
deleteM k (M kvs) = M (filter ((/= k) . fst) kvs)

-- Devuelve una lista con todas las claves.
keys :: Map k v -> [k]
keys (M kvs) = map fst kvs
```

---

6. Multiset (Multiconjunto)

Implementación utilizando un Map para asociar cada elemento con su número de ocurrencias (un entero). La eficiencia de sus operaciones dependerá de la del Map subyacente (en este caso, O(n)).

Haskell

```
-- Se asume que la implementación del TAD Map está disponible.
-- module Map (Map, emptyM, assocM, lookupM, deleteM, keys) where ...

module Multiset(Multiset, emptyMS, addMS, occurrencesMS) where
```

```
-- Representación: Un Map que asocia elementos a su cantidad de ocurrencias.
data Multiset a = MS (Map a Int) deriving Show

-- Crea un multiconjunto vacío.
emptyMS :: Multiset a
emptyMS = MS emptyM

-- Agrega una ocurrencia de un elemento.
addMS :: (Ord a, Eq a) => a -> Multiset a -> Multiset a
addMS x (MS m) =
  let count = maybe 0 id (lookupM x m)
  in MS (assocM x (count + 1) m)

-- Devuelve el número de ocurrencias de un elemento.
occurrencesMS :: (Ord a, Eq a) => a -> Multiset a -> Int
occurrencesMS x (MS m) = maybe 0 id (lookupM x m)
```