Claro, aquí tienes la resolución de la práctica de la **Nave Espacial** reajustada. He modificado las implementaciones para que no utilicen funciones genéricas de alto orden como map o filter, sino que se basen en **funciones recursivas auxiliares** que operan utilizando la interfaz del TAD, como solicitaste.

Resolución Práctica 8.1: Nave Espacial (Versión Refactorizada)
Este documento contiene la resolución completa de los ejercicios, con implementaciones que evitan el uso de funciones de alto orden genéricas, enfocándose en un estilo recursivo que utiliza directamente la interfaz del TAD.

---

a) Invariantes de Representación
Dada la representación data Nave = N (Map SectorId Sector) (Map Nombre Tripulante) (MaxHeap Tripulante), los siguientes invariantes deben mantenerse:

1. **Consistencia de Tripulantes**: El conjunto de Nombres que son claves en el Map de Tripulantes debe ser idéntico al conjunto de nombres de los Tripulantes almacenados en el MaxHeap.
2. **Sincronización de Datos de Tripulantes**: Los datos de un tripulante en el Map de Tripulantes deben ser idénticos a los datos del mismo tripulante en el MaxHeap.
3. **Integridad Referencial de Sectores**: Si un Tripulante tiene un SectorId asignado, dicho SectorId **debe** existir como clave en el Map de Sectores.
4. **Integridad Referencial de Tripulantes**: Si un Sector tiene el Nombre de un tripulante asignado, dicho Nombre **debe** existir como clave en el Map de Tripulantes.

Implementación de la Interfaz
b) construir

- **Propósito**: Construye una nave con sectores vacíos, en base a una lista de identificadores de sectores.
- **Eficiencia**: O(SlogS).
- **Nota**: Una implementación con inserciones sucesivas (assocM) tiene un costo logarítmico por inserción.

Haskell
```
construir :: [SectorId] -> Nave
construir ids = N (construirSectores ids emptyM) emptyM emptyH

-- Subtarea recursiva para poblar el mapa de sectores.
-- Costo: S llamadas a assocM. Total O(S log S).
construirSectores :: [SectorId] -> Map SectorId Sector -> Map SectorId Sector
construirSectores [] mapS = mapS
construirSectores (id:ids) mapS =
 let nuevoSector = crearS id -- O(1)
    mapActualizado = assocM id nuevoSector mapS -- O(log S)
 in construirSectores ids mapActualizado
```
c) ingresarT (Sin cambios)

- **Propósito**: Incorpora un tripulante a la nave, sin asignarle un sector.
- **Eficiencia**: O(logT).

Haskell
```
ingresarT :: Nombre -> Rango -> Nave -> Nave
ingresarT nombre rango (N mapS mapT heapT) =
 let nuevoT = crearT nombre rango -- O(1)
    nuevoMapT = assocM nombre nuevoT mapT -- O(log T)
```

```
    nuevoHeapT = insertH nuevoT heapT     -- O(log T)
 in N mapS nuevoMapT nuevoHeapT
```

d) sectoresAsignados (Sin cambios)

- **Propósito**: Devuelve los sectores asignados a un tripulante.
- **Precondición**: Existe un tripulante con dicho nombre.
- **Eficiencia**: O(logT).

Haskell

```
sectoresAsignados :: Nombre -> Nave -> Set SectorId
sectoresAsignados nombre (N _ mapT _) =
 case lookupM nombre mapT of -- O(log T)
  Just tripulante -> sectoresT tripulante -- O(1)
```

e) datosDeSector (Sin cambios)

- **Propósito**: Dado un sector, devuelve los tripulantes y los componentes asignados.
- **Precondición**: Existe un sector con dicho id.
- **Eficiencia**: O(logS).

Haskell

```
datosDeSector :: SectorId -> Nave -> (Set Nombre, [Componente])
datosDeSector sectorId (N mapS _ _) =
 case lookupM sectorId mapS of -- O(log S)
  Just sector -> (tripulantesS sector, componentesS sector) -- O(1) + O(1)
```

f) tripulantesN (Sin cambios)

- **Propósito**: Devuelve la lista de tripulantes ordenada por rango, de mayor a menor.
- **Eficiencia**: O(TlogT).

Haskell

```
tripulantesN :: Nave -> [Tripulante]
tripulantesN (N _ _ heapT) = heapToList heapT

-- Función auxiliar que implementa HeapSort
heapToList :: MaxHeap Tripulante -> [Tripulante]
heapToList heap =
 if isEmptyH heap -- O(1)
 then []
 else let maxT = maxH heap -- O(1)
       restoHeap = deleteMaxH heap -- O(log T)
    in maxT : heapToList restoHeap
```

g) agregarASector

- **Propósito**: Asigna una lista de componentes a un sector de la nave.
- **Eficiencia**: O(C+logS), siendo C la cantidad de componentes.

Haskell

```
agregarASector :: [Componente] -> SectorId -> Nave -> Nave
agregarASector componentes sectorId (N mapS mapT heapT) =
 case lookupM sectorId mapS of -- O(log S)
  Just sector ->
   let sectorActualizado = agregarComponentes componentes sector -- O(C)
       mapSActualizado = assocM sectorId sectorActualizado mapS -- O(log S)
   in N mapSActualizado mapT heapT

-- Subtarea recursiva para agregar los componentes a un sector.
-- Costo: C llamadas a agregarC. Total O(C) ya que agregarC es O(1).
agregarComponentes :: [Componente] -> Sector -> Sector
```

```
agregarComponentes [] sector = sector
agregarComponentes (c:cs) sector =
 let sectorConC = agregarC c sector -- O(1)
 in agregarComponentes cs sectorConC
```

h) asignarASector (Sin cambios)

- **Propósito**: Asigna un sector a un tripulante.
- **Precondición**: El tripulante y el sector existen.
- **Eficiencia**: O(logS+TlogT).

Haskell

```
asignarASector :: Nombre -> SectorId -> Nave -> Nave
asignarASector nombre sectorId (N mapS mapT heapT) =
 let tripulante = fromJust (lookupM nombre mapT)    -- O(log T)
    sector = fromJust (lookupM sectorId mapS)     -- O(log S)
    tripulanteActualizado = asignarS sectorId tripulante -- O(log S)
    sectorActualizado = agregarT nombre sector      -- O(log T)
    mapTActualizado = assocM nombre tripulanteActualizado mapT -- O(log T)
    mapSActualizado = assocM sectorId sectorActualizado mapS    -- O(log S)
    tripulantesActualizados = valuesM mapTActualizado -- O(T)
    heapTActualizado = heapFromList tripulantesActualizados -- O(T log T)
 in N mapSActualizado mapTActualizado heapTActualizado
```

Funciones como Usuario

i) sectores

- **Propósito**: Devuelve todos los sectores no vacíos.
- **Nota**: Se asume una función todosLosSectoresIds :: Nave -> [SectorId] con costo O(S).
- **Eficiencia**: O(SlogS).

Haskell

```
sectores :: Nave -> Set SectorId
sectores nave =
 let ids = todosLosSectoresIds nave -- O(S)
 in sectoresNoVacios ids nave emptyS -- S * O(log S) + S * O(log S)

-- Subtarea recursiva que filtra e inserta en un set los sectores con tripulantes.
sectoresNoVacios :: [SectorId] -> Nave -> Set SectorId -> Set SectorId
sectoresNoVacios [] _ set = set
sectoresNoVacios (id:ids) nave set =
 let (tripulantes, _) = datosDeSector id nave -- O(log S)
 in if sizeS tripulantes > 0 -- O(1)
   then sectoresNoVacios ids nave (addS id set) -- O(log S)
   else sectoresNoVacios ids nave set
```

j) sinSectoresAsignados

- **Propósito**: Devuelve los tripulantes que no poseen sectores asignados.
- **Eficiencia**: O(TlogT).

Haskell

```
sinSectoresAsignados :: Nave -> [Tripulante]
sinSectoresAsignados nave =
 let todosLosTripulantes = tripulantesN nave -- O(T log T)
 in filtrarSinSectores todosLosTripulantes nave [] -- T * O(log T)

-- Subtarea recursiva que filtra los tripulantes sin sectores.
```

```
filtrarSinSectores :: [Tripulante] -> Nave -> [Tripulante] -> [Tripulante]
filtrarSinSectores [] _ acumulados = acumulados
filtrarSinSectores (t:ts) nave acumulados =
  let sectores = sectoresAsignados (nombre t) nave -- O(log T)
  in if sizeS sectores == 0 -- O(1)
     then filtrarSinSectores ts nave (t : acumulados)
     else filtrarSinSectores ts nave acumulados
```

k) barriles

- **Propósito**: Devuelve todos los barriles de los sectores de la nave.
- **Nota**: Se asume todosLosSectoresIds :: Nave -> [SectorId] de costo O(S).
- **Eficiencia**: O(SlogS+Ctotal).

Haskell

```
barriles :: Nave -> [Barril]
barriles nave =
  let ids = todosLosSectoresIds nave -- O(S)
  in barrilesDeSectores ids nave -- S * O(log S)

-- Subtarea que recorre los IDs de sectores y acumula los barriles.
barrilesDeSectores :: [SectorId] -> Nave -> [Barril]
barrilesDeSectores [] _ = []
barrilesDeSectores (id:ids) nave =
  let (_, componentes) = datosDeSector id nave -- O(log S)
     barrilesEnSector = barrilesDeComponentes componentes -- O(C_sector)
  in barrilesEnSector ++ barrilesDeSectores ids nave

-- Subtarea que extrae los barriles de una lista de componentes.
barrilesDeComponentes :: [Componente] -> [Barril]
barrilesDeComponentes [] = []
barrilesDeComponentes (c:cs) =
  let barrilesEnC = case c of
                Almacen b -> b
                _       -> []
  in barrilesEnC ++ barrilesDeComponentes cs
```