Ejercicio 1: Costo de Heapsort

Se analiza el costo de la función heapsort asumiendo una PriorityQueue implementada con un **Heap**, que ofrece costos logarítmicos.

- **Propósito**: Ordenar una lista de elementos.
- **Implementación conceptual**:
  1. Tomar una lista de N elementos e insertarlos uno por uno en una PriorityQueue vacía.
  2. Extraer el elemento mínimo de la PriorityQueue N veces y construir la lista ordenada.
- **Análisis de Costo**:
  - **Fase de Inserción**: Se realizan N inserciones. El costo de insertar en un heap es O(log K), donde K es el tamaño del heap en ese momento. La suma de los costos es log(1) + log(2) + ... + log(N). Esta sumatoria es O(N log N).
  - **Fase de Extracción**: Se realizan N extracciones. El costo de extraer el mínimo de un heap es O(log K). De manera similar a la inserción, el costo total de esta fase es O(N log N).
  - **Costo Total**: El costo total de heapsort es la suma de ambas fases, resultando en **O(N log N)**.

Ejercicio 2: Funciones sobre BST (Versión Refactorizada)

Estas son las implementaciones sobre un Árbol Binario de Búsqueda (BST) utilizando las estructuras de control solicitadas.

Haskell

```
-- Definición base para los ejercicios
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
```

---

1. belongsBST

Haskell

```
-- Propósito: Dado un BST, dice si el elemento pertenece o no al árbol.
belongsBST :: Ord a => a -> Tree a -> Bool
belongsBST e t =
 case t of
  EmptyT -> False
  NodeT x ti td ->
   if e == x then
    True
   else if e < x then
    belongsBST e ti
   else
    belongsBST e td
```

- **Análisis de Costo: O(log N)** La función recorre una única rama del árbol. En cada paso, una expresión if-else descarta la mitad del árbol restante. El número de operaciones es proporcional a la altura del árbol, que en un árbol balanceado es log N.

---

2. insertBST

Haskell

```
-- Propósito: Dado un BST, inserta un elemento en el árbol.
insertBST :: Ord a => a -> Tree a -> Tree a
insertBST e t =
 case t of
  EmptyT -> NodeT e EmptyT EmptyT
  NodeT x ti td ->
   if e == x then
```

```
    NodeT x ti td -- El elemento ya existe, no se hace nada.
  else if e < x then
    NodeT x (insertBST e ti) td
  else
    NodeT x ti (insertBST e td)
```

- **Análisis de Costo: O(log N)** La función busca la ubicación correcta descendiendo por una sola rama (costo O(log N)). Luego, al volver de la recursión, reconstruye los nodos en ese mismo camino. El costo total sigue siendo proporcional a la altura del árbol.

---

3. deleteBST

Haskell

```
-- Propósito: Dado un BST, borra un elemento en el árbol.
deleteBST :: Ord a => a -> Tree a -> Tree a
deleteBST e t =
 case t of
  EmptyT -> EmptyT
  NodeT x ti td ->
   if e < x then
    NodeT x (deleteBST e ti) td
   else if e > x then
    NodeT x ti (deleteBST e td)
   else
    rearmarBST ti td

-- Subtarea para rearmar el árbol después de borrar un nodo.
rearmarBST :: Ord a => Tree a -> Tree a -> Tree a
rearmarBST ti td =
 case ti of
  EmptyT -> td
  _    -> let (max, ti') = splitMaxBST ti in NodeT max ti' td
```

- **Análisis de Costo: O(log N)** La función primero busca el elemento a borrar (O(log N)). Al encontrarlo, si es necesario, llama a la subtarea rearmarBST, que a su vez utiliza splitMaxBST (costo O(log N)). El costo total es la suma de estos recorridos logarítmicos, que sigue siendo O(log N).

---

4. splitMinBST

Haskell

```
-- Propósito: Dado un BST, devuelve un par con el mínimo elemento y el árbol sin el mismo.
-- Precondición: El árbol no puede ser vacío.
splitMinBST :: Ord a => Tree a -> (a, Tree a)
splitMinBST t =
 case t of
  EmptyT -> error "splitMinBST: el arbol no puede ser vacio"
  NodeT x ti td ->
   case ti of
    EmptyT -> (x, td)
    _    -> let (minElem, ti') = splitMinBST ti
            in (minElem, NodeT x ti' td)
```

- **Análisis de Costo: O(log N)** Para encontrar el mínimo, la función desciende recursivamente por la rama izquierda. El costo es proporcional a la altura del árbol, O(log N).

5. splitMaxBST

Haskell

```
-- Propósito: Dado un BST, devuelve un par con el máximo elemento y el árbol sin el mismo.
-- Precondición: El árbol no puede ser vacío.
splitMaxBST :: Ord a => Tree a -> (a, Tree a)
splitMaxBST t =
 case t of
  EmptyT -> error "splitMaxBST: el arbol no puede ser vacio"
  NodeT x ti td ->
   case td of
    EmptyT -> (x, ti)
    _    -> let (maxElem, td') = splitMaxBST td
            in (maxElem, NodeT x ti td')
```

- **Análisis de Costo: O(log N)** Análogo a splitMinBST, pero descendiendo por la rama derecha. El costo es O(log N).

---

6. esBST

Haskell

```
-- Propósito: Indica si el árbol cumple con los invariantes de BST.
esBST :: Ord a => Tree a -> Bool
esBST t =
 case t of
  EmptyT -> True
  NodeT x ti td ->
   cumpleInvarianteNodo x ti td && esBST ti && esBST td

-- Subtarea: Verifica el invariante para un único nodo.
cumpleInvarianteNodo :: Ord a => a -> Tree a -> Tree a -> Bool
cumpleInvarianteNodo x ti td =
 elementosSonMenoresA x (treeToList ti) &&
 elementosSonMayoresA x (treeToList td)

-- Subtarea: Convierte un árbol a una lista (ineficiente aquí).
treeToList :: Tree a -> [a]
treeToList EmptyT = []
treeToList (NodeT x ti td) = treeToList ti ++ [x] ++ treeToList td

-- Subtareas para verificar propiedades de listas.
elementosSonMenoresA :: Ord a => a -> [a] -> Bool
elementosSonMenoresA _ [] = True
elementosSonMenoresA n (x:xs) = x < n && elementosSonMenoresA n xs

elementosSonMayoresA :: Ord a => a -> [a] -> Bool
elementosSonMayoresA _ [] = True
elementosSonMayoresA n (x:xs) = x > n && elementosSonMayoresA n xs
```

- **Análisis de Costo: O(N^2)** Para cada nodo del árbol (N nodos), la subtarea cumpleInvarianteNodo convierte sus subárboles a listas. treeToList cuesta O(N) en el peor caso. Esta operación costosa se repite en cada nodo, llevando a un costo total de O(N^2).

---

7. elMaximoMenorA

Haskell

```
-- Propósito: Dado un BST y un elemento, devuelve el máximo elemento que sea menor al elemento dado.
elMaximoMenorA :: Ord a => a -> Tree a -> Maybe a
elMaximoMenorA e t =
 case t of
  EmptyT -> Nothing
  NodeT x ti td ->
   if e <= x then
    -- Si el elemento es menor o igual a la raíz, el resultado SÓLO puede estar a la izquierda.
    elMaximoMenorA e ti
   else
    -- Si el elemento es mayor, la raíz 'x' es un candidato.
    -- Buscamos si existe un candidato mejor (más grande, pero aún menor que 'e') en la derecha.
    let resultadoDerecha = elMaximoMenorA e td
    in case resultadoDerecha of
       Just mejorCandidato -> Just mejorCandidato
       Nothing             -> Just x
```

- **Análisis de Costo: O(log N)** La función recorre una única rama. En cada nodo, la estructura if-else decide si ir a la izquierda o a la derecha, sin explorar ambos subárboles. El costo es proporcional a la altura del árbol.

---

8. elMinimoMayorA

Haskell

```
-- Propósito: Dado un BST y un elemento, devuelve el mínimo elemento que sea mayor al elemento dado.
elMinimoMayorA :: Ord a => a -> Tree a -> Maybe a
elMinimoMayorA e t =
 case t of
  EmptyT -> Nothing
  NodeT x ti td ->
   if e >= x then
    -- Si el elemento es mayor o igual a la raíz, el resultado SÓLO puede estar a la derecha.
    elMinimoMayorA e td
   else
    -- Si el elemento es menor, la raíz 'x' es un candidato.
    -- Buscamos si existe un candidato mejor (más chico, pero aún mayor que 'e') en la izquierda.
    let resultadoIzquierda = elMinimoMayorA e ti
    in case resultadoIzquierda of
       Just mejorCandidato -> Just mejorCandidato
       Nothing             -> Just x
```

- **Análisis de Costo: O(log N)** La lógica es simétrica a elMaximoMenorA y recorre una sola rama, resultando en un costo proporcional a la altura del árbol.

---

9. balanceado

Haskell

```
-- Propósito: Indica si el árbol está balanceado.
balanceado :: Tree a -> Bool
balanceado t =
```

```
 case t of
  EmptyT -> True
  NodeT _ ti td ->
   -- Un árbol está balanceado si el nodo actual está balanceado Y sus hijos también lo están.
   if nodoEstaBalanceado t then
    balanceado ti && balanceado td
   else
    False

-- Subtarea: Calcula la altura de un árbol.
heightT :: Tree a -> Int
heightT t =
 case t of
  EmptyT -> 0
  NodeT _ ti td -> 1 + max (heightT ti) (heightT td)

-- Subtarea: Verifica si un único nodo cumple la condición de balanceo AVL.
nodoEstaBalanceado :: Tree a -> Bool
nodoEstaBalanceado t =
 case t of
  EmptyT -> True
  NodeT _ ti td ->
   let diff = heightT ti - heightT td
   in diff >= -1 && diff <= 1
```

- **Análisis de Costo: O(N^2)** La función balanceado recorre cada nodo. Para cada nodo, la subtarea nodoEstaBalanceado llama a heightT, que recorre todos los descendientes de ese nodo. Esto provoca que los nodos se recorran múltiples veces, resultando en un costo cuadrático.

Ejercicios 4 y 5: Implementación del TAD Empresa

El código para estos ejercicios ya utilizaba subtareas y no dependía de guardas, por lo que su estructura se mantiene. A continuación se presenta el código y su análisis de costos correspondiente.

Invariante de Representación para Empresa

Para una empresa = ConsE mapSectores mapCuil:

1. **Consistencia de Empleados**: Todo empleado en un Set de mapSectores debe existir como clave en mapCuil.
2. **Consistencia de Sectores**: Si un empleado en mapCuil tiene asignado un sector s, debe pertenecer al Set de empleados del sector s en mapSectores.

Funciones del TAD Empresa (Ejercicio 4)

Haskell

```
-- Propósito: Agrega un empleado a la empresa, que trabajará en dichos sectores y tendrá el CUIL dado.
agregarEmpleado :: [SectorId] -> CUIL -> Empresa -> Empresa
agregarEmpleado ss c (ConsE mapSectores mapCuil) =
  let nuevoEmp = crearEmpleadoConSectores ss c
      mapCuil' = assocM c nuevoEmp mapCuil
      mapSectores' = agregarEmpleadoASectores ss nuevoEmp mapSectores
  in ConsE mapSectores' mapCuil'
```

```
-- Subtarea: Crea un empleado y le asigna una lista de sectores.
crearEmpleadoConSectores :: [SectorId] -> CUIL -> Empleado
crearEmpleadoConSectores ss c = foldr incorporarSector (consEmpleado c) ss

-- Subtarea: Agrega un empleado a los sets de múltiples sectores.
agregarEmpleadoASectores :: [SectorId] -> Empleado -> Map SectorId (Set Empleado) -> Map
SectorId (Set Empleado)
agregarEmpleadoASectores [] _ mapS = mapS
agregarEmpleadoASectores (s:ss) emp mapS =
  let setActual = fromMaybe emptyS (lookupM s mapS)
    setActualizado = addS emp setActual
    mapParcialmenteActualizado = assocM s setActualizado mapS
  in agregarEmpleadoASectores ss emp mapParcialmenteActualizado
```

- **Análisis de Costo: O(L * (log S + log E_s) + log E)** Donde L es la cantidad de sectores a agregar. La creación del empleado cuesta O(L * log L). Actualizar el mapa de CUILs cuesta O(log E). La subtarea agregarEmpleadoASectores se ejecuta L veces, y en cada paso realiza operaciones O(log S) y O(log E_s). El costo total es dominado por este último proceso.

Funciones de Usuario de Empresa (Ejercicio 5)

Haskell

```
-- Propósito: Dado un CUIL de empleado le asigna todos los sectores de la empresa.
convertirEnComodin :: CUIL -> Empresa -> Empresa
convertirEnComodin c empresa =
  let todosLosS = todosLosSectores empresa
  in asignarSectoresAEmpleado todosLosS c empresa

-- Subtarea para realizar la asignación iterativa.
asignarSectoresAEmpleado :: [SectorId] -> CUIL -> Empresa -> Empresa
asignarSectoresAEmpleado sectores cuil emp = foldr (\s accEmpresa -> agregarASector s cuil
accEmpresa) emp sectores
```

- **Análisis de Costo: O(S * (log E + log S))** Primero, todosLosSectores obtiene una lista de S sectores (O(S)). Luego, la subtarea asignarSectoresAEmpleado ejecuta agregarASector (O(log E + log S)) por cada uno de los S sectores.