Resolución Práctica: Random Access List (RAList)
Este documento contiene la resolución completa de los ejercicios del parcial modelo 2018s2, enfocados en la implementación del TAD RAList.

---

a) Invariantes de Representación
Dada la representación data RAList a = MkR Int (Map Int a) (Heap a), se deben mantener los siguientes invariantes para asegurar la consistencia de la estructura:

1. **Consistencia de Tamaño**: El Int que representa la próxima posición a ocupar debe ser igual a la cantidad de elementos en el Map y en el Heap. Si la lista está vacía, este valor debe ser 0.
2. **Consistencia de Contenido**: El conjunto de valores a contenidos en el Map debe ser exactamente el mismo que el conjunto de elementos almacenados en el Heap.
3. **Integridad de Índices**: Las claves del Map deben ser un conjunto de enteros consecutivos que comiencen en 0 y terminen en N-1, donde N es el valor del Int (la cantidad de elementos).

---

b) Implementación de la Interfaz
A continuación se implementan las funciones de la interfaz del TAD RAList.
emptyRAL

- **Propósito**: Devuelve una lista vacía.
- **Eficiencia**: O(1).

Haskell
```
emptyRAL :: RAList a
emptyRAL = MkR 0 emptyM emptyH
```

- **Análisis de costo**: La construcción se realiza a partir de emptyM y emptyH, ambas operaciones de costo O(1).

isEmptyRAL

- **Propósito**: Indica si la lista está vacía.
- **Eficiencia**: O(1).

Haskell
```
isEmptyRAL :: RAList a -> Bool
isEmptyRAL (MkR nextPos _ _) = nextPos == 0
```

- **Análisis de costo**: La operación consiste en acceder al primer campo y compararlo con 0, lo cual es de costo O(1).

lengthRAL

- **Propósito**: Devuelve la cantidad de elementos.
- **Eficiencia**: O(1).

Haskell
```
lengthRAL :: RAList a -> Int
lengthRAL (MkR nextPos _ _) = nextPos
```

- **Análisis de costo**: La operación accede directamente al campo que almacena el tamaño, siendo de costo O(1).

get

- **Propósito**: Devuelve el elemento en el índice dado.
- **Precondición**: El índice debe existir.
- **Eficiencia**: O(logN).

Haskell
```
get :: Int -> RAList a -> a
get index (MkR _ mapV _) = fromJust (lookupM index mapV)
```

- **Análisis de costo**: La operación principal es lookupM, que tiene un costo de O(logK), donde K es N. fromJust es O(1). El costo total es O(logN).

minRAL

- **Propósito**: Devuelve el mínimo elemento de la lista.
- **Precondición**: La lista no está vacía.
- **Eficiencia**: O(1).

Haskell
```
minRAL :: Ord a => RAList a -> a
minRAL (MkR _ _ heapV) = findMin heapV
```

- **Análisis de costo**: Se utiliza findMin del Heap, que es una operación de costo O(1).

add

- **Propósito**: Agrega un elemento al final de la lista.
- **Eficiencia**: O(logN).

Haskell
```
add :: Ord a => a -> RAList a -> RAList a
add elem (MkR nextPos mapV heapV) =
 let newMap = assocM nextPos elem mapV  -- O(log N)
    newHeap = insertH elem heapV     -- O(log N)
 in MkR (nextPos + 1) newMap newHeap
```

- **Análisis de costo**: assocM tiene un costo de O(logN) y insertH tiene un costo de O(logN). El costo total es O(logN).

elems

- **Propósito**: Transforma una RAList en una lista, respetando el orden de los elementos.
- **Eficiencia**: O(NlogN).

Haskell
```
elems :: Ord a => RAList a -> [a]
elems ral = elems' 0 (lengthRAL ral) ral

-- Subtarea para construir la lista recursivamente usando la interfaz del TAD.
-- Costo: N llamadas a 'get', cada una O(log N). Total O(N log N).
elems' :: Int -> Int -> RAList a -> [a]
elems' currentIndex len ral =
 if currentIndex == len
 then []
 else (get currentIndex ral) : (elems' (currentIndex + 1) len ral)
```

- **Análisis de costo**: La función se implementa mediante una subtarea recursiva elems' que se invoca N veces. En cada invocación, se llama a la función get, que tiene un costo de O(logN). Por lo tanto, el costo total es la suma de N llamadas a get, resultando en $N*O(logN)=O(NlogN)$.

remove

- **Propósito**: Elimina el último elemento de la lista.
- **Precondición**: La lista no está vacía.
- **Eficiencia**: O(NlogN).

Haskell
```
remove :: Ord a => RAList a -> RAList a
remove (MkR nextPos mapV _) =
 let lastIndex = nextPos - 1
    -- Se elimina la última clave del mapa
    newMap = deleteM lastIndex mapV -- O(log N)
    -- Como la interfaz de Heap no permite borrar un elemento arbitrario,
    -- es necesario reconstruir el Heap con los elementos restantes.
    -- Se asume una función valuesM :: Map k v -> [v] de costo O(N).
    remainingElems = valuesM newMap -- O(N)
    newHeap = heapFromList remainingElems -- O(N log N)
```

```
 in MkR lastIndex newMap newHeap
```

- **Análisis de costo**: El costo está dominado por la reconstrucción del Heap a partir de los N-1 elementos restantes, que es O(NlogN). deleteM tiene un costo de O(logN), que es menor.

set

- **Propósito**: Reemplaza el elemento en la posición dada.
- **Precondición**: El índice debe existir.
- **Eficiencia**: O(NlogN).

Haskell

```
set :: Ord a => Int -> a -> RAList a -> RAList a
set index newVal (MkR nextPos mapV _) =
 let -- Se actualiza el valor en el mapa
    newMap = assocM index newVal mapV -- O(log N)
    -- Al igual que en remove, un cambio en un valor requiere
    -- reconstruir el Heap para mantener su invariante.
    allElems = valuesM newMap -- O(N)
    newHeap = heapFromList allElems -- O(N log N)
 in MkR nextPos newMap newHeap
```

- **Análisis de costo**: assocM cuesta O(logN). Sin embargo, la reconstrucción del Heap es la operación más costosa con O(NlogN), determinando la eficiencia total.

addAt

- **Propósito**: Agrega un elemento en la posición dada, desplazando los posteriores.
- **Precondición**: El índice debe estar entre 0 y la longitud de la lista.
- **Eficiencia**: O(NlogN).

Haskell

```
-- Subtarea para correr los elementos del Map.
-- Costo: (maxPos - currentPos) * O(log N), que en el peor caso es O(N log N).
shiftRight :: Int -> Int -> Map Int a -> Map Int a
shiftRight fromPos currentPos map =
 if currentPos < fromPos
 then map
 else let Just elemToMove = lookupM currentPos map
        mapWithMoved = assocM (currentPos + 1) elemToMove map
     in shiftRight fromPos (currentPos - 1) mapWithMoved

addAt :: Ord a => Int -> a -> RAList a -> RAList a
addAt index elem (MkR nextPos mapV _) =
 let -- Se desplazan los elementos desde el final hasta el índice.
    shiftedMap = shiftRight index (nextPos - 1) mapV -- O(N log N)
    -- Se inserta el nuevo elemento en la posición liberada.
    finalMap = assocM index elem shiftedMap -- O(log N)
    -- Se reconstruye el Heap con la nueva colección de elementos.
    allElems = valuesM finalMap -- O(N)
    newHeap = heapFromList allElems -- O(N log N)
 in MkR (nextPos + 1) finalMap newHeap
```

- **Análisis de costo**: El desplazamiento de elementos en el Map (shiftRight) tiene un costo de O(NlogN) en el peor caso. La reconstrucción del Heap también es O(NlogN). Por lo tanto, la eficiencia total es O(NlogN).