Práctica de ejercicios # 6 - Priority Queue, Map y Multiset

Estructuras de Datos, Universidad Nacional de Quilmes

12 de mayo de 2022

Aclaraciones:

- ***Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados.*** *No se saltee ejercicios sin consultar antes a un docente.*
- ***Recuerde que puede aprovechar en todo momento las funciones que ha definido****, tanto las de esta misma práctica como las de prácticas anteriores.*
- ***Pruebe todas sus implementaciones****, al menos en una consola interactiva.*
- *Es sumamente aconsejable* ***resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en videos publicados o clases presenciales****, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Priority Queue (cola de prioridad)

Ejercicio 1

La siguiente interfaz representa colas de prioridad, llamadas priority queue, en inglés. La misma posee operaciones para insertar elementos, y obtener y borrar el mínimo elemento de la estructura. Implementarla usando listas, e indicando el costo de cada operación.

- `emptyPQ :: PriorityQueue a`
  Propósito: devuelve una priority queue vacía.
- `isEmptyPQ :: PriorityQueue a -> Bool`
  Propósito: indica si la priority queue está vacía.
- `insertPQ :: Ord a => a -> PriorityQueue a -> PriorityQueue a`
  Propósito: inserta un elemento en la priority queue.
- `findMinPQ :: Ord a => PriorityQueue a -> a`
  Propósito: devuelve el elemento más prioriotario (el mínimo) de la priority queue.
  Precondición: parcial en caso de priority queue vacía.
- `deleteMinPQ :: Ord a => PriorityQueue a -> PriorityQueue a`
  Propósito: devuelve una priority queue sin el elemento más prioritario (el mínimo).
  Precondición: parcial en caso de priority queue vacía.

Ejercicio 2

Implementar la función `heapSort :: Ord a => [a] -> [a]`, que dada una lista la ordena de menor a mayor utilizando una Priority Queue como estructura auxiliar. ¿Cuál es su costo?

OBSERVACIÓN: el nombre `heapSort` se debe a una implementación particular de las Priority Queues basada en una estructura concreta llamada Heap, que será trabajada en la siguiente práctica.

2. Map (diccionario)

Ejercicio 3

La interfaz del tipo abstracto Map es la siguiente:

- `emptyM :: Map k v`
  Propósito: devuelve un map vacío
- `assocM :: Eq k => k -> v -> Map k v -> Map k v`
  Propósito: agrega una asociación clave-valor al map.
- `lookupM :: Eq k => k -> Map k v -> Maybe v`
  Propósito: encuentra un valor dado una clave.
- `deleteM :: Eq k => k -> Map k v -> Map k v`
  Propósito: borra una asociación dada una clave.
- `keys :: Map k v -> [k]`
  Propósito: devuelve las claves del map.

Implementar como usuario del tipo abstracto Map las siguientes funciones:

1. `valuesM :: Eq k => Map k v -> [Maybe v]`
   Propósito: obtiene los valores asociados a cada clave del map.
2. `todasAsociadas :: Eq k => [k] -> Map k v -> Bool`
   Propósito: indica si en el map se encuentran todas las claves dadas.
3. `listToMap :: Eq k => [(k, v)] -> Map k v`
   Propósito: convierte una lista de pares clave valor en un map.
4. `mapToList :: Eq k => Map k v -> [(k, v)]`
   Propósito: convierte un map en una lista de pares clave valor.
5. `agruparEq :: Eq k => [(k, v)] -> Map k [v]`
   Propósito: dada una lista de pares clave valor, agrupa los valores de los pares que compartan la misma clave.
6. `incrementar :: Eq k => [k] -> Map k Int -> Map k Int`
   Propósito: dada una lista de claves de tipo k y un map que va de k a Int, le suma uno a cada número asociado con dichas claves.
7. `mergeMaps:: Eq k => Map k v -> Map k v -> Map k v`
   Propósito: dado dos maps se agregan las claves y valores del primer map en el segundo. Si una clave del primero existe en el segundo, es reemplazada por la del primero.

Indicar los ordenes de complejidad en peor caso de cada función implementada, justificando las respuestas.

Ejercicio 4

Implemente las siguientes variantes del tipo Map, indicando los costos obtenidos para cada operación, justificando las respuestas:

1. Como una lista de pares-clave valor sin claves repetidas
2. Como una lista de pares-clave valor con claves repetidas
3. Como dos listas, una de claves y otra de valores, donde la clave ubicada en la posición i está asociada al valor en la misma posición, pero de la otra lista.

Ejercicio 5

Implemente estas otras funciones como usuario de Map:

- `indexar :: [a] -> Map Int a`
  Propósito: dada una lista de elementos construye un map que relaciona cada elemento con su posición en la lista.
- `ocurrencias :: String -> Map Char Int`
  Propósito: dado un string, devuelve un map donde las claves son los caracteres que aparecen en el string, y los valores la cantidad de veces que aparecen en el mismo.

Indicar los ordenes de complejidad en peor caso de cada función del usuario en base a la implementación elegida, justificando las respuestas.

3. MultiSet (multiconjunto)

Ejercicio 6

Un *MultiSet* (multiconjunto) es un tipo abstracto de datos similar a un *Set* (conjunto). A diferencia del último, cada elemento posee una cantidad de apariciones, que llamaremos *ocurrencias* del elemento en el multiset. Su interfaz es la siguiente:

- `emptyMS :: MultiSet a`
  Propósito: denota un multiconjunto vacío.
- `addMS :: Ord a => a -> MultiSet a -> MultiSet a`
  Propósito: dados un elemento y un multiconjunto, agrega una ocurrencia de ese elemento al multiconjunto.
- `ocurrencesMS :: Ord a => a -> MultiSet a -> Int`
  Propósito: dados un elemento y un multiconjunto indica la cantidad de apariciones de ese elemento en el multiconjunto.
- `unionMS :: Ord a => MultiSet a -> MultiSet a -> MultiSet a` (opcional)
  Propósito: dados dos multiconjuntos devuelve un multiconjunto con todos los elementos de ambos multiconjuntos.
- `intersectionMS :: Ord a => MultiSet a -> MultiSet a -> MultiSet a` (opcional)
  Propósito: dados dos multiconjuntos devuelve el multiconjunto de elementos que ambos multiconjuntos tienen en común.
- `multiSetToList :: MultiSet a -> [(a, Int)]`
  Propósito: dado un multiconjunto devuelve una lista con todos los elementos del conjunto y su cantidad de ocurrencias.

1. Implementar el tipo abstracto *MultiSet* utilizando como representación un Map. Indicar los ordenes de complejidad en peor caso de cada función de la interfaz, justificando las respuestas.
2. Reimplementar como usuario de *MultiSet* la función `ocurrencias` de ejercicios anteriores, que dado un string cuenta la cantidad de ocurrencias de cada caracter en el string. En este caso el resultado será un multiconjunto de caracteres.