Estructuras de Datos – UNQ

Parcial – Registro Vehicular

RECORDATORIOS

- Poner apellido y nombre, número de hoja y cantidad total de hojas en CADA hoja entregada.
- Dejar MEDIA carilla vacía en la 1era hoja, para poder realizar correcciones.

Descripción

El objetivo de esta evaluación es implementar una parte de un Registro Vehicular. El objetivo es mantener un registro de fotos de vehículos que fueron tomadas automáticamente, de forma de poder responder ciertas consultas. Cada foto del registro de vehículos se identifica mediante un Código Único Identificador de Foto (CUIF).

Para representar a los elementos involucrados se usarán dos TADs auxiliares: el tipo `CUIF` para representar códigos de fotos, y el tipo `Vehiculo` para representar a los vehículos. Ambos tipos se pueden comparar por igualdad y por orden (poseen funciones `(==)`, `(<=)` y `(<)`), siendo éstas las únicas operaciones relevantes de estos TADs para este trabajo. También se utilizará el tipo algebraico `Foto`, dado por

```
data Foto = MkF CUIF (Set Vehiculo)
```

que representa el tipo de las fotos tomadas para registro, indicando qué vehículos aparecen en ella.

El tipo de datos abstracto de Registro Vehicular, `RV`, cuenta con la siguiente interfaz. En esta interfaz se utilizan los números *F*, la cantidad total de fotos del Registro Vehicular, *V*, la cantidad de vehículos diferentes que aparecen en el Registro y *vf* la cantidad máxima de vehículos en alguna foto.

- `nuevoRV :: RV`
  Propósito: describe un Registro Vehicular vacío.
  Eficiencia: *O(1)*

- `agregarFoto :: RV -> Foto -> RV`
  Propósito: dado un Registro y una foto de registro, describe el Registro con la foto agregada.
  Precondición: el identificador de la foto que se agrega no fue usado previamente en el Registro.
  Eficiencia: *O(log F + vf^2 * log vf)*

- `todasLasFotos :: RV -> [CUIF]`
  Propósito: describe una lista con todas y cada una de los identificadores de fotos del Registro.
  Eficiencia: *O(F)*

- `quienesAparecen :: RV -> CUIF -> Set Vehiculo`
  Propósito: describe el conjunto de vehículos que aparecen en la foto dada.
  Precondición: el CUIF debe corresponder a una foto del Registro.
  Eficiencia: *O(log F)*

- `aparecenJuntos :: RV -> Vehiculo -> Vehiculo -> Bool`
  Propósito: dado un Registro y 2 vehículos, indica si los mismos fueron registrados juntos alguna vez.
  Precondición: los vehículos deben ser distintos.
  Eficiencia: *O(log V + log vf)*

del TAD

Implementar las siguientes funciones como usuario del TAD `RV`, respetando los costos dados y justificándolos.

a. `esInnecesaria :: RV -> CUIF -> Bool` que indica si la foto fue un falso positivo, es decir, no incluye ningún vehículo.
Eficiencia: *O(log F)*

b. `juntos :: Vehiculo -> Vehiculo -> RV -> Set CUIF` que dado dos vehículos y un Registro, describe el conjunto de aquellas fotos del Registro en las que los vehículos aparecen juntos. Se puede suponer que los dos vehículos dados son distintos.
Eficiencia: *O(F * (log F + log vf))*

2. **Implementación del TAD**
Implementar el TAD `RV` suponiendo el siguiente tipo de representación:

```
data RV = MkRV (Map CUIF Foto)                    -- Fotos del registro
               (Map Vehiculo (Set Vehiculo))
                -- Vehículos que aparecen juntos en alguna foto
```

a. Escribir los invariantes de representación para poder crear elementos válidos del TAD.
b. Implementar las funciones de la interfaz, respetando las restricciones de eficiencia pedidas, justificándolas.

3. **Variante del TAD**
Modificar la implementación del punto anterior suponiendo que en la interfaz se agrega una nueva operación:

- `masCargada :: RV -> Maybe CUIF`
Propósito: dado un Registro describe una de las fotos del mismo con más vehículos, si hay alguna.
Eficiencia: *O(1)*

Observación 1: no hace falta reimplementar aquellas operaciones para las que solamente cambia el PM, sin afectar el código de la misma.

Observación 2: para aquellas operaciones existentes que deban cambiar su implementación, dicha modificación NO debe afectar el costo ya dado.

de interfaces

, siendo K la cantidad de claves distintas en el map:

```
emptyM  :: Map k v                          O(1)
assocM  :: Ord k => k -> v -> Map k v -> Map k v   O(log K)
lookupM :: Ord k => k -> Map k v -> Maybe v        O(log K)
deleteM :: Ord k => k -> Map k v -> Map k v        O(log K)
keysM   :: Map k v -> [k]                          O(K)
sizeM   :: Map k v -> Int                          O(1)
```

Heap, siendo H la cantidad de datos en el heap:

```
emptyH     :: Heap a                        O(1)
isEmptyH   :: Heap a -> Bool                O(1)
insertH    :: Ord k => a -> Heap a -> Heap a   O(log H)
findMaxH   :: Ord k => Heap a -> a          O(1)
deleteMaxH :: Ord k => Heap a -> Heap a     O(log H)
```

Set, siendo S la cantidad de elementos del conjunto:

```
emptyS    :: Set a                          O(1)
isEmptyS  :: Set a -> Bool                  O(1)
addS      :: Eq a => a -> Set a -> Set a    O(log S)
belongsS  :: Eq a => a -> Set a -> Bool     O(log S)
deleteS   :: Eq a => a -> Set a -> Set a    O(log S)
unionS    :: Eq a => Set a -> Set a -> Set a   O(S * log S)
sizeS     :: Set a -> Int                   O(1)
setToList :: Set a -> [a]                   O(S)
```