Algoritmos y Estructura de Datos 2

Parcial 3

Diseño

Alumno: Leandro Carreira
LU: 669/18

Link a documento online con barra lateral para rápida navegación en el documento:
https://docs.google.com/document/d/1JmkPqFtOgO0nGiyoBgCpiURlJLkQ6DOz7-YnH1v-xI0/edit?usp=sharing

1. Elección de estructuras

Todes les que estamos trabajando desde nuestra casa, sabemos de la pesadilla que es que se acumulen los platos rápidamente por estar todo el día en casa. Por eso, les docentes de Algo 2 queremos aportar para mejorar la situación. Hicimos un TAD para modelar una pileta y les pedimos a ustedes que lo diseñen.

El sistema PILETA (nos matamos con el nombre) permite dejar una pila de platos personalizada, así cada persona lava lo suyo y nadie tiene que estar lavando lo de las demás (bien individualista la cosa). Para dejar una pila de platos se llama a la operación dejarPlatos. Si una persona llama a dejarPlatos muchas veces, su pila de platos se acumula (o sea, llamar a dejarPlatos(p, "Pepe", [1,2]) y después a dejarPlatos(p, "Pepe", [3,4]) es equivalente a llamar a dejarPlatos(p, "Pepe", [3,4,1,2]), siempre lo primero es lo último que se agrega). Los nombres de las personas tienen longitud acotada (se puede asumir que la operación de copiar o guardar un nombre tiene tiempo constante)

Como no queremos que ninguna pila crezca demasiado, sólo puede lavar quien tenga la pila más numerosa. Si hay varias personas que tienen la pila de igual tamaño, se determina quién lava de alguna manera. También nos piden saber quien es le que más lavó históricamente y quien es le que tiene más vajilla por lavar actualmente.

A continuación les dejamos la especificación del problema:

TAD VAJILLA ES NAT
TAD PERSONA ES STRING

TAD PILETA

géneros pileta

observadores básicos

personasConPila : pileta $\longrightarrow$ conj(persona)

porLavar : pileta p $\times$ persona per $\longrightarrow$ secu(vajilla) $\{per \in \text{personasConPila}(p)\}$

cuántoLavó : pileta p $\times$ persona per $\longrightarrow$ nat

generadores

nueva : $\longrightarrow$ pileta

dejarPlatos : pileta p $\times$ persona per $\times$ secu(vajilla) vs $\longrightarrow$ pileta
$\{(\forall\, per' : \text{persona})(\forall\, v : \text{vajilla})\ per' \in \text{personasConPila}(p) \land \text{está?}(v, vs) \Rightarrow_L \neg\text{está?}(v, \text{porLavar}(p, per'))\}$

lavar : pileta $\longrightarrow$ pileta

otras operaciones

elQueMásLavó : pileta p $\longrightarrow$ persona $\{(\exists per : \text{persona})\ \text{cuántoLavó}(per) > 0\}$

elQueMásTieneParaLavar : pileta p $\longrightarrow$ persona $\{\neg\text{vacio?}(\text{personasConPila}(p))\}$

Fin TAD

Dadas las siguientes operaciones con las complejidades temporales de peor caso indicadas (teniendo en cuenta que P es el conjunto de personas que pasaron por el sistema y s es la vajilla que se agrega a la pila):

- PERSONASCONPILA(**in** p: pileta, **out** ps: conj(persona))
  Devuelve las personas que tengan platos pendientes de lavar.
  Complejidad: $O(1)$
- PORLAVAR(**in** p: pileta, **in** per: persona, **out** vs: secu(vajilla))
  Devuelve los platos por lavar de la persona per.
  Complejidad: $O(\log(\#P))$
- DEJARPLATOS(**in**/**out** p: pileta, **in** per: persona, **in** s: secu(vajilla))
  Agrega una pila de platos a los de esa persona.
  Complejidad: $O(\log(\#P) + \text{len}(s))$
- LAVAR(**in**/**out** p: pileta)
  Elimina la primer vajilla de la secuencia más larga.
  Complejidad: $O(\log(\#P))$
- ELQUEMASLAVO(**in** p: pileta) $\longrightarrow$ per : persona
  Devuelve la persona que más vajillas lavó en total.
  Complejidad: $O(1)$
- ELQUEMASTIENEPARALAVAR(**in** p: pileta) $\longrightarrow$ per : persona
  Devuelve la persona que más vajillas tiene actualmente por lavar.
  Complejidad: $O(1)$

Se pide:

1. Dar una estructura de representación para un módulo PILETA (que provee las operaciones mencionadas), explicando detalladamente qué información se guarda en cada parte, las relaciones entre las partes, y las estructuras de datos subyacentes.
2. Justificar detalladamente de qué manera es posible implementar cada una de las operaciones para cumplir con las complejidades pedidas. Escribir el algoritmo para la operación LAVAR.

Para la resolución del ejercicio no esta permitido utilizar módulos implementados con tabla hash de base. Esto se debe a dos motivos: uno porque queremos que combinen el resto de las estructuras vistas, y otro porque los peores casos de la tabla hash exceden los que se piden en los ejercicios (por ejemplo, duplicar el espacio en una tabla cuesta $O(n)$ en el peor caso).

Donde se menciona 'secu' se puede usar alguno de los módulos vistos que se explican con el TAD Secuencia, de acuerdo a lo que está en el apunte de Módulos Básicos.

Representación:

```
Pileta se representa con estr
donde estr es tupla <
          pilas:      diccTrie(Persona, Pila(Nat)),
          personas:   conjLineal(Persona),
          historial:  listaMaxHeap(Nat),
          cantidades: diccMaxHeap(Nat, <itDiccTrie, itConjLineal, itListaMaxHeap>*),
```

>

* Escrito de forma explícita:

< itDiccTrie<Persona, Pila(Nat)>,
itConjLineal<Persona> ,
itListaMaxHeap<Nat> >

Interpretación de "Vajilla":

Vajilla es Nat, pero qué representa este Nat no es claro: Puede ser un **ID** que de alguna forma identifica a la vajilla, como también puede ser una cantidad de vajillas de algún tipo en la pila , por ejemplo:

Agregar una secuencia [1, 3, 2] a una Pila vacía puede ser dejar 1 olla, 3 cubiertos (tenedor, cuchillo, cuchara) y debajo de todo, 2 platos.

Asumo el primer caso (cada Nat es un ID) en las explicaciones que siguen, ya que la operación Lavar habla de *secuencia más larga*, y no de *mayor cantidad de vajillas a lavar*.

A continuación, el detalle de cada una de las operaciones para esta estructura.

Operación **PersonasConPila**:

Se devuelve una **referencia** en **O(1)** al Conjunto Lineal que representa el conjunto de personas que actualmente tienen pilas de vajilla **sin** lavar en la pileta.

Para su mantenimiento (en detalle en las próximas operaciones):

1. Cada vez que una persona **agrega su pila** a Pileta, se debe **buscar si existe** en el diccionario e.pilas en O(|P|) dado por la **búsqueda en un Trie**.
   Como |P| se puede acotar por una constante pues la longitud del String (Persona) está acotada, **O(|P|) ≡ O(1)**.

   Si existe, no hace falta hacer nada para mantenerlo.
   Si **no** existe, se agrega al Conjunto Lineal con la operación AgregarRápido O(copia(P)) ≡ O(1), aprovechando que sabemos que **no habrá repetidos**.

2. Cada vez que una persona lava **todos** sus platos, debe **eliminarse** del conjunto de personas.

Operación **PorLavar**

Se devuelve una **referencia** a la Pila de Nat (Vajilla) correspondiente a la persona en esa clave del diccionario Trie.

Esta búsqueda se logra en O(|P|) que por la razón ya explicada, se acota por **O(1)**.

Complejidad: O(1) (que es menor a la complejidad de peor caso requerida de O(log #P))

Operación **DejarPlatos**:

Dos casos: La persona ya tenía una pila o no.

1. Caso *"Ya tenía pila de platos"*:

   Se busca la persona en el diccionario e.pilas, y al encontrarla en O(|P|) ≡ O(1) se agregan (Apilan) 1 a 1 los elementos de la secuencia **s** de entrada (secuencia de Vajilla) **comenzando por el último** de ellos..

   Se asume que la secuencia de vajilla se implementa con una **Lista Enlazada**. Cada una de estas operaciones de copia de la Lista Enlazada a la Pila es O(copia(Nat)) ≡ O(1), por lo que agregar (Apilar) los |s| elementos (*len(s)*) de la secuencia a la pila costará **O(|s|)**.

   También debe **actualizarse** el diccionario Max Heap e.cantidades, que tiene costo **O(log #P)** para incrementar la clave en tantas unidades como cantidad de vajillas hay en la secuencia de entrada, operación que cuesta **O(|s|)**, dada por las |s|+1 operaciones de suma.

   Juntanto todo, el costo total será de:

   O(log #P) + O(|s|) + O(|s|) ≡
   ≡ O(log #P) + 2 * O(|s|)
   ≡ O(log #P) + O(2 * |s|)
   ≡ O(log #P) + O(|s|)
   ≡ **O(log #P + |s|)**

   Costo total: **O(log #P + |s|)**

2. Caso *"No tenía pila de platos"*:
   Se busca la persona en el diccionario e.pilas y no se encuentra: O(|P|) ≡ O(1)

   Se agrega en la misma pasada, inicializando una Pila vacía (O(1)) y apilando uno a uno los elementos de la secuencia comenzando por el último como se explicó en el caso 1 (O(|s|))

   También debe **actualizarse** el diccionario Max Heap e.cantidades **agregando** un nuevo elemento, cuya complejidad dada por Max Heap también es **O(log #P) + O(|s|)** por sumar los |s|+1 elementos.

   Costo total: **O(log #P + |s|)**

   Observación: Puede darse el caso que el elemento a agregar (Nat (cantidad de vajillas)) ya exista en e.cantidades. En ese caso se agrega de nuevo y se

establece la convención de que el valor repetido **más antiguo** en el Max Heap será el de mayor prioridad, con el objetivo de evitar pilas de platos que duren para siempre (un asco).

Operación **Lavar**:

Encontrar al de la mayor cantidad de vajillas a lavar, cuesta **O(1)** siendo la raíz del Max Heap que representa e.cantidades.

Este nodo del Max Heap posee un iterador al diccionario de pilas, teniendo acceso a la Persona y su Pila en **O(1)**.

Lavar la última vajilla de esta pila cuesta **O(1)** dado por la operación Desapilar del módulo Pila de la persona.

A partir de la referencia del iterador en la tercera componente de la tupla significado de e.cantidades, se actualiza el historial e.historial de la cantidad de vajillas lavadas por persona, que al ser una lista implementada con un Max Heap, cuesta **O(log #P)** pues **incrementa** su valor**.**

Aquí se dan otros dos casos:

1. Era la última vajilla a lavar:
   La persona no tiene más vajilla por lavar, por lo que debe ser **borrada** del **conjunto de personas** con pila de vajillas en la Pileta: **O(1)** pues tengo la referencia al elemento del conjunto a partir del iterador en la segunda componente de la tupla del valor del Max Heap e.cantidades.

   NO voy a elimiar a la persona del diccTrie e.pilas y e.cantidades pues las necesito para computar el record de la persona que más lavó.

   Costo caso 1: **O(1)**

2. No era la última vajilla a lavar:
   Se desapila el primer elemento de la Pila de la Persona ya encontrada: O(1).
   La cantidad de vajillas a lavar en e.cantidades de esa persona debe **reducirse** en 1, por lo que borro el elemento anterior O(log #P) y vuelvo a insertarlo reducido en O(log #P), ya que no tengo operación “**Decrease**-key” en un **Max**-heap.

   Costo caso 1: **O(log #P) + O(log #P) ≡ O(log #P)**

Costo Total: O(log #P)

Operación **ElQueMásLavó**:

Al estar en la raíz del Max Heap e.historial, puedo devolverlo en O(1) usando la referencia de la primera componente de la tupla que apunta al diccionario de pilas de cada persona. Como este diccionario contiene a todas las personas que alguna vez tuvieron una Pila en la Pileta, puedo estar seguro de que la persona existe como clave del mismo.

Operación **ElQueMásTieneParaLavar**:

Similarmente a la operación anterior, el que más tiene para lavar está en la raíz del Max heap e.cantidades, y por ende puedo devolver su nombre en O(1) usando la referencia a la clave del diccionario e.pilas.

Algoritmo:

Asumo que tener un iterador apuntando a un elemento de una estructura, usar Siguiente(itEstruct) devuelve **el valor de la clave** en el caso de un diccionario, y no su significado, por lo que para obtener su significado, debo usar Significado(diccionario, Siguiente(itEstruct)).

```
Lavar(in/out p: Pileta)
    // De la raíz del Max Heap, extraigo iteradores
    // Asumo que en la implementación, la raiz es el primer elemento de una lista
    // con valor Nat y significado tupla de los 3 elementos correspondientes
    itPilas     ← π1(Prim(p.cantidades).significado)                    O(1)
    itPersonas  ← π2(Prim(p.cantidades).significado)                    O(1)
    itHistorial ← π3(Prim(p.cantidades).significado)                    O(1)

    // Borro elemento de la pila de la persona con más vajillas
    Desapilar(Significado(p.pilas, Siguiente(itPilas)))                 O(1)

    // Aumento en 1 la cantidad de vajillas lavadas por esa persona en el historial
    Siguiente(itHistorial) ← Siguiente(itHistorial) + 1                 O(1)

    if EsVacía?(Significado(p.pilas, Siguiente(itPilas))) then          O(1)
        // Ya no tiene más vajillas por lavar, lo elimino del Conjunto
        EliminarSiguiente(itPersonas)                                   O(1)
    end if

    // Actualizo Max Heap e.cantidades
    // Como debo reducir la clave del nodo del max heap, lo borro y vuelvo a agregar
    // Pues no tengo operación "Reducir", solo "Incrementar"
    newClave       ← Prim(p.cantidades).clave - 1                       O(1)
    // Copio tupla de 3 referencias
    newSignificado ← Prim(p.cantidades).significado                     O(1)
    EliminarRaíz(p.cantidades)                                          O(log #P)
    Agregar(p.cantidades, newClave, newSignificado)                     O(log #P)
```

Complejidad: O(log #P)
