

# Estructuras de Datos y Algoritmos – iic2133

## Control 3

17 de abril, 2019

Nombre: \_\_\_\_\_

1) Una tienda quiere premiar a sus  $k$  clientes más rentables. Para ello cuenta con la lista de las  $n$  compras de los últimos años, en que cada compra es una tupla de la forma  $(ID\ Cliente, Monto)$ . La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras.

Explica cómo usar *tablas de hash* y *heaps* para resolver este problema en tiempo esperado —o promedio—  $O(n + n \log k)$ .

**Solución:** El problema se separa en dos partes. Se asignará puntaje por separado para cada una.

### - Calcular la rentabilidad de cada cliente (2pts)

Queremos obtener el set de tuplas  $(ID\ Cliente, Rentabilidad)$ , donde la rentabilidad para el  $ID\ Cliente = i$  es la suma de todos los montos de las tuplas de la forma  $(i, Monto)$ .

Para esto creamos una tabla de hash  $T$  donde se almacenan tuplas  $(ID\ Cliente, Monto)$ . Cada vez que se inserte un  $ID\ Cliente$ :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las  $n$  tuplas hemos efectivamente encontrado la rentabilidad para cada cliente. **[1pt]**

La inserción en esta tabla tiene tiempo esperado  $O(1)$  como se ha visto en clases. Como se realizan  $n$  inserciones, esta parte tiene tiempo esperado  $O(n)$ . **[1pt]**

### - Buscar los $k$ clientes más rentables (4pts)

**Posible solución:**

Sea  $m$  el total de clientes distintos. Creamos un *min-heap* de tamaño  $k$  que contendrá los  $k$  clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los  **$k$  más rentables** encontrados hasta el momento. **[1pt]**

Iteramos sobre las tuplas en  $T$ , insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre  $T$ . Para cada elemento que

veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *sift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior. **[2pts]** (También es posible extraer la cabeza del heap e insertar el nuevo elemento normalmente)

Cada inserción en el heap toma  $O(\log k)$ . En el peor caso insertamos los  $m$  elementos en el heap, por lo que esta parte es  $O(m \log k)$ . Pero como en el peor caso  $m = n$ , esta parte es  $O(n \log k)$ . **[1pt]**

**Para cada sección, no se asignará puntaje si no explican correctamente el método propuesto o este no resuelve correctamente el problema. Para la segunda sección, se asignará como máximo 1pt si no alcanzan la complejidad solicitada.**

2) La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores en existencia en todos los multiversos, y necesita ayuda para hacer más eficientes la atención de los pedidos, ya que recibe millones por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando una pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar *tablas de hash* para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

**Solución:** Necesitamos resolver las siguientes operaciones:

- Registrar pedido (*pizza*, *nombre*). Para una misma *pizza* los nombres deben guardarse por orden de llegada.
- Buscar el siguiente *nombre* para una *pizza* dada.
- Eliminar el pedido del sistema.

Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo *key*. En este caso *key* corresponde al tipo de pizza y un *value* corresponde al nombre de la persona que lo pidió. **[1.5pts]**

Los *values* de un mismo *key* se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea  $O(1)$  y se atienda en orden de llegada. **[1pt]**. (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son  $O(\log n)$ , por lo que sólo obtienen **[0.5pts]**)

La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura. **[1pt]**

Pero como el dominio de las *key* es infinito, debemos ir despejando las celdas de la tabla cuando una *key* se queda sin *values*, ya que no tenemos memoria infinita. **[0.5pts]** Para esto es necesario usar **encadenamiento**, ya que permite eliminar *keys* de la tabla sin perjudicar el rendimiento de esta. **[2pts]**