Algoritmos y Estructuras de Datos

Segundo Parcial – Viernes 28 de Junio de 2024

LU	Apellido y Nombre	#Hojas	E1	E2	E3	E4	Nota	Corrigió
363/23	Martínez, Fausto Nicolás	4	20	29	13	30	92	Camilo

- Es posible tener una hoja (2 carillas), escrita a mano, con los anotaciones que se deseen, más los apuntes del campus
- Cada ejercicio debe entregarse en **hojas separadas**
- Incluir en cada hoja nombre y apellido, número de libreta y número de hoja
- El parcial se aprueba con 70 puntos

A

E1. Complejidad [20 pts]

Responder, justificando adecuadamente su respuesta:

a) Decimos que la complejidad de insertar en una lista enlazada ordenada es $O(n)$ en el peor caso, pero eso sólo es cierto si podemos comparar dos elementos en $O(1)$ y lo mismo para la operación de copia. Calcule la complejidad de insertar en una lista enlazada ordenada *sin aliasing* (es decir, copiando en la lista el elemento a insertar y no haciendo una referencia al mismo) si comparar es $O(\log n)$ y copiar es $O(n)$.

b) Si la complejidad en el peor caso de un algoritmo es $\Omega(n)$, ¿es verdad que la complejidad de mejor caso no puede ser $O(1)$?

c) Indique si la siguiente expresión es verdadera o falsa y justifique: Si $O(f(n)) \cap \Omega(g(n)) = \emptyset$, entonces $O(g(n)) \cap \Omega(f(n)) = \emptyset$.

E2. Elección de estructuras e invariante de representación [30 pts]

Una relación parcial es una correspondencia entre dos conjuntos en la cual a cada elemento del primer conjunto le corresponde cero o más elementos del segundo conjunto. Especificamos una relación parcial con el siguiente TAD:

```
TAD Relacion<T1,T2> {
    obs elems: conj<tupla<T1, T2>>

    proc relacionar(inout r: Relacion<T1, T2>, in t1: T1, in t2: T2)
        requiere {r = R0}
        asegura {r = R0 ∪ {<t1,t2>}}

    proc relacionadosDer(in r: Relacion<T1, T2>, in t1: T1): Conjunto<T2>
        asegura {(∀ t2 : T2)(<t1,t2> ∈ r.elems ↔ t2 ∈ res)}

    proc relacionadosIzq(in r: Relacion<T1, T2>, in t2: T2): Conjunto<T1>
        asegura {(∀ t1 : T1)(<t1,t2> ∈ r.elems ↔ t1 ∈ res)}
}
```

Queremos las siguientes complejidades para las operaciones:

- relacionar debe ser $O(\log n + \log m)$.
- relacionadosIzq debe ser $O(\log m)$.
- relacionadosDer debe ser $O(\log n)$.

donde n es la cantidad de elementos de $T1$ que tienen al menos un elemento de $T2$ relacionado, y m es la cantidad de elementos de $T2$ que tienen al menos un elemento de $T1$ relacionado.

a) Proponga una estructura para implementar este TAD cumpliendo esas complejidades.

b) Se quiere agregar la operación `másRelacionadoIzq` que dada una relación devuelve el `T1` que tiene más `T2` relacionados (en caso de haber más de un `T1` con la cantidad máxima de relacionados, se puede devolver cualquiera de ellos). Dicha operación debe tener complejidad $O(1)$. Explique cómo modificaría la estructura propuesta en el punto anterior para también resolver este nuevo problema.

c) Escribir en castellano pero en forma precisa el invariante de representación de la estructura (incluyendo los cambios realizados en el punto b)

d) Escribir en castellano, también en forma precisa, la función de abstracción para dicha estructura.

E3. Estructuras [20 pts]

¿Qué estructura elegiría para implementar cada uno de los siguientes conjuntos? Justifique su respuesta indicando qué ventajas tiene la estructura que eligió.

a) un conjunto de a lo sumo 100 números de 64 bits (entre 1 y 18446744073709551615)
b) un conjunto de mediciones en el que se agregan nuevas mediciones muy frecuentemente y se consulta muy poco
c) un conjunto de palabras sobre un alfabeto no acotado en el que se inserta poco y se consulta mucho

E4. Sorting [30 pts]

Dado un arreglo de pares <estudiante, nota>, queremos devolver un arreglo de pares <estudiante, promedio>, ordenado en forma descendente por promedio y, en caso de empate, por número de libreta.

Los estudiantes se representan con un número de libreta (puede considerar que es un número de a lo sumo 10 dígitos), y la nota es un número entre 1 y 100. La cantidad de estudiantes se considera no acotada.

Se pide dar un algoritmo que resuelva el problema con complejidad $O(n + m \log m)$, donde n es la cantidad total de notas (es decir, el tamaño del arreglo de entrada), y m es la cantidad total de estudiantes.

Ejemplo:

```
Entrada: <12395, 72>, <45615, 81>, <12395, 94>, <45615, 100>, <78920, 81>,
         <12395, 90>, <45615, 71>, <78920, 50>
Salida: <12395, 85.3333>, <45615, 84>, <78920, 65.5>
```

a) Describa cada etapa del algoritmo con sus palabras, y justifique por qué cumple con las complejidades pedidas.
b) Escriba el algoritmo en pseudocódigo. Puede utilizar (sin reescribir) todos los algoritmos y estructuras de datos vistos en clase.

Fausto Martinez - 2º Parcial
Algoritmos y Estructuras de Datos II

1) Complejidad

a) Un esbozo del algoritmo de ordenar en una lista enlazada ordenada es el siguiente:

```
it es Iterador de listaEnlazada
while it.haySiguiente() { //Se ejecuta n veces en peor caso
    if it.siguiente() > elemAInsertar then → O(log n)
        Meto el elemento → O(n)
    else (y termino la ejecución del while)
        sigo
    endif
```

Cuando asumimos la comparación en O(1) y la copia en O(n), el peor caso, que es que el elemento vaya al final de la lista, nos da O(n)

Ahora, si comparar es O(log n) y copiar el elem es O(n)

Vemos que en el peor caso, que sigue siendo el final, tengo O(n log n) por las n comparaciones y O(n) por la única vez que meto al elemento a la listaEnlazada

La complejidad resulta O(n log n + n)

= O(n log n) ☒

b) La afirmación es falsa, un algoritmo puede tardar muchísimo en su peor caso y ser Ω(n) y el mejor caso, ser tan rápido como O(1)

~~Este no cumple por ejemplo porque en el peor caso tiene complejidad resultante cuadrática y sin embargo en el mejor caso puede~~

Por ejemplo, algún algoritmo de búsqueda puede ser Ω(n) en su peor caso si el elemento no está en la estructura de datos pero en el mejor caso, podría tener al elemento por ejemplo en la primer posición del array y encontrarlo llevaría O(1).

c) La afirmación es falsa.

Contraejemplo $f(n) = n$, $g(n) = n^3$

$O(f(n)) \cap \Omega(g(n))$ es vacío,
pues no existe ninguna h tal que

$\exists C_1 > 0, n_0 \in \mathbb{N} \mid n \geq n_0 \Rightarrow h \leq C_1 n$.

$\exists C_2 > 0, n_1 \in \mathbb{N} \mid n \geq n_1 \Rightarrow h \geq C_2 n^3$

simultaneamente

(Es decir, no hay funciones "mas chicas" que n y "mas grandes" que n^3 simultaneamente)

En cambio, $n^2 \in \Omega(f(n)) = \Omega(n) = \Omega(n)$

$n^2 \in O(n^3)$

$\Rightarrow \Omega(n) \cap O(n^3) \neq \emptyset$

$\Rightarrow \Omega(f(n)) \cap O(g(n)) \neq \emptyset$ ⊠

↑ pues n^2 pertenece a ambos.

(Es decir, sin embargo sí existen funciones "más grandes" que n y "mas chicas" que n^3 a la vez)

Fausto Martínez - 2° Parcial
Algoritmos y Estructuras de Datos II

2) Elección de estructuras e invariante de representación

a) La estructura propuesta para implementar el TAD es:

diccRelacionesDer : diccLog<T1, conjLog<T2>>
diccRelacionesIzq : diccLog<T2, conjLog<T1>>

Veamos que, al agregar hago

diccRelacionesDer.obtener(t1) [O(log n)] . agregar(t2) [O(log m)]

y

diccRelacionesIzq.obtener(t2) [O(log m)] . agregar(t1) [O(log n)]

Al pedir relacionadosDer hago

diccRelacionesDer.obtener(t1) O(log n)

y lo mismo con relacionadosIzq

diccRelacionesIzq.obtener(t2) O(log m)

Así, cumplo todas las complejidades pedidas.

b) Como debe tener complejidad O(1), propongo el observador o atributo

masRelIzq: tupla<T1, int>

que tendrá el elemento más relacionado y su cantidad de relaciones.

La idea para que esto funcione es que cada vez que ingreso una nueva relación me fijo si el t1 se convirtió o no en el más relacionado. Sería algo así:

en agregar: ....
diccRelacionesDer.obtener(t1).agregar(t2).
if diccRelacionesDer.obtener(t1).tamaño() [O(1)] > masRelIzq1
masRelIzq0 = t1
masRelIzq1 = diccRelacionesDer.obtener(t1).tamaño.
endif.
:

en masRelacionadoIzq: return masRelIzq0. // O(1)

c) Invariante de Representación:

El elemento de tipo T_1 que está en masRelIzq es efectivamente el más relIzq de todos, es decir: está en diccRelacionesDer y su conjunto asociado tiene tamaño masRelIzq1, que es mayor a cualquier otro tamaño de conjuntoLog en diccRelacionesDer.

Luego, para toda clave $t_1 : T_1$ en diccRelacionesDer pasa que si entramos a cada $t_2 : T_2$ en su conjLog asociado y con esos t_2 los usamos como clave en diccRelacionesIzq, obtendremos un conjLog en el que t_1 está.

Lo mismo pasa en sentido contrario.

Todo ese trabalenguas quiere decir que si t_1 está relacionado con t_2, t_2 lo está con t_1.

d) Función de abstracción

Acá relacionamos TAD con Módulo.

si hay una tupla $\langle t_1, t_2 \rangle$ en r.elems. (es decir, para cada $\langle t_1, t_2 \rangle$)

$\Rightarrow t_1 \in r'.\text{diccRelacionesDer} \wedge t_2 \in r'.\text{diccRelDer.obtener}(t_1)$

Relación $t_2 \in r'.\text{diccRelacionesIzq} \wedge t_1 \in r'.\text{diccRelIzq.obtener}(t_2)$

para cada t_1 clave de diccRelDer, las tuplas formadas por $\langle t_1, j \rangle$, $j \in r'.\text{diccRelDer.obtener}(t_1)$ pertenecen a r.elems.

para cada t_2 clave de diccRelIzq, las tuplas formadas por $\langle j, t_2 \rangle$ $j \in r'.\text{diccRelIzq.obtener}(t_2)$ pertenecen a r.elems.

NOTA

Fausto Martinez - 2º Parcial
Algoritmos y Estructuras de Datos II

3) Estructuras

a) En el primer caso, como ya sé que no va a haber más de 100 registros, me va a convenir usar un Array de 100 posiciones. Esta estructura es la más rápida en muchos sentidos y la que más eficiente va a ser. Me va a permitir agregar, consultar, eliminar todo muy rápidamente. Su única desventaja es la capacidad acotada, pero como sabemos que no tenemos más de 100 registros, nos viene bomba.

b) No hay una única manera de interpretar este inciso, pero voy a asumir que no nos importa el orden de las mediciones.

En tal caso, la estructura que voy a elegir va a ser un Conjuntolineal. Este tiene la ventaja de permitirme agregar Rapido en $O(n)$, que es lo que necesito, ya que como se agregan mediciones frecuentemente, me interesa que eso se haga rápido. pertenece, sacar son $O(n)$, bastante pesado, lento, pero no me interesa pues se consulta poco.

c) Creo que estaríamos tentados a elegir un Diccionario Digital para guardar las palabras, pero que el alfabeto no sea acotado nos priva de sus ventajas.

De esta manera, optaré por elegir un Vector, ya que podemos hacer operaciones de consulta, como obtener, rápidamente en $O(1)$.

Tenemos como desventaja de esa estructura, que al agregar toma

$O(f(n))$, con $f(n) = \begin{cases} n & \text{si } n = 2^k \text{ para } k \in \mathbb{Z} \\ 1 & \text{si no.} \end{cases}$

O sea que agregar podría tomar $O(n)$ en algún caso. Pero esto no solo será poco frecuente porque insertamos poco en el ~~Array~~ Vector, si no que encima esas veces que tarda $O(n)$ el Vector se duplica en capacidad y la próxima vez que tardará $O(n)$ al insertar será cada vez más lejana.

En conclusión: Nos sirve porque es muy eficiente para nuestros propósitos.

Un vector puede en O(1), pero busca (find) en O(n). No nos aporta mucho si no podemos saber la posición del elemento a priori. Un tree modificado hubiera sido ideal, pero cualquier estructura logarítmica hubiera estado bien

Fausto Martinez - 2° Parcial
Algoritmos y Estructuras de Datos II

4) Sorting

a) Primero, recorro el array que nos dieron, y a partir de eso creo un diccionario digital con clave estudiante y valor una tupla <sumaNotas, cantNotas>

Esto me llevará O(n) por recorrer el arreglo original y porque, como la clave (Estudiante) está acotada, definir lleva O(1).

A partir de ese diccDigital, recorriendolo en O(m), obtengo un array de <Estudiante, Prom> desordenado.

Acá ordeno primero por el criterio menos relevante (Estudiante), a quienes, aprovechandome de que tienen 1 a lo sumo 10 digitos puedo ordenar en O(10m) = O(m) con el radix sort visto en clase.

Al promedio, lo ordeno con mergeSort que me lleva O(m log m)

Así, obtengo el array pedido en

$$O(n+m+m+m\log m) = O(n+m\log m)_{//}$$

b) estudiante es int(no), nota es int(100), sumaNota es int, promedio es float.

Algoritmo ordenarPorPromedio(entrada array<tupla<estudiante, nota>>) : array<tupla<estudiante, promedio>>

DiccionarioDigital<Estudiante, tupla<sumaNota, int>> tmeAux := new. diccVacio();

for each tupla in entrada → O(n)

if tmeAux.esta(tupla$_0$) then → O(1)

~~tmeAux.obtener(tupla$_0$)$_0$:= tmeAux.obtener(tupla$_0$)$_0$ + tupla$_1$~~

~~tmeAux.obtener(tupla$_0$)$_1$:= tmeAux.obtener(tupla$_0$)$_1$ + 1~~

→ O(1)

tmeAux.definir(tupla$_0$, <tmeAux.obtener(tupla$_0$)$_0$ + tupla$_1$, tmeAux.obtener(tupla$_0$)$_1$ + 1>);

else

tmeAux.definir(tupla$_0$, <tupla$_1$, 1>); → O(1)

endif.

endfor

```
Array<tupla<Estudiante, promedio>> res = new Array(treAux.tamaño)
int i = 0                                          ⟶ O(1)
for each estudiante in treAux.claves → O(m)
    res[i] := <estudiante, treAux.obtener(estudiante)> / treAux.obtener(estudiante)>
    i++;
endfor
res := radixSortNatEnPrimeraCoord(res);   O(10m)
res := mergeSortEnSegundaCoord(res);      O(m log m)
return res;
```

Complejidad: $O(n + m + 10m + m \log m)$

$= O(n + m \log m)$ //

Aclaración: El for que recorre las claves de treAux es $O(m)$ pues no hay estudiantes repetidos en el Trie, tenemos la cantidad justa, m.