Examen Final

Algoritmos y Estructuras de Datos II

20 de diciembre de 2016

Para aprobar el final es necesario aprobar al menos dos ejercicios.

Ejercicio 1

Describa la **estructura de representación** y la **implementación** de una cola de prioridad indexada. Una *cola de prioridad indexada* es una cola de prioridad que asocia cada elemento con una *clave*. Esta clave, que se autogenera en la inserción, *identifica* unívocamente al elemento dentro de la cola y se puede usar para cambiar su prioridad. La cola de prioridad indexada debe permitir las suguientes operaciones:

- Insertar un elemento en $O(\log n)$; retorna su **clave**.
- Acceder al tope de la cola en $O(1)$.
- Borrar el tope de la cola en $O(\log n)$.
- Cambiar la prioridad del elemento identificado por una **clave** en $O(\log n)$.

En todos los casos n representa la cantidad de elementos en la cola, y la complejidad es en peor caso. La descripción puede ser en castellano, con el suficiente nivel de detalle como para ser entendido por alguien que ya conoce los temas de AED2. En particular, puede hacer referencia a las estructuras básicas vistas en AED2 sin contar cómo se implementan, salvo que las mismas deban ser modificadas (en cuyo caso alcanza con contar las modificaciones).

Ejercicio 2

Describa la **interfaz** (en castellano) y **estructura de representación** de un módulo que implementa el TAD diccionario con claves ordenadas, describiendo una familia de operaciones suficientes como para poder implementar todas las operaciones esperadas en un tiempo razonable. En particular, el diccionario se debe poder recorrer iterativamente (es decir, clave a clave, en orden) en tiempo lineal consumiendo $O(1)$ memoria adicional, mientras que la búsqueda, inserción y borrado de elementos deben costar $O(\log n)$. Recuerde describir los aspectos de aliasing y las posibilidades de modificación de los valores de entrada y salida.

Ejercicio 3

Determine cuáles de las siguientes ecuaciones de recurrencia pueden ser resueltas con el teorema maestro visto en AED2. Para aquellas que pueden ser resueltas, determine cuál es el valor de la recurrencia, justificando qué caso del teorema maestro aplica. Para el resto, explique el motivo por el que el teorema maestro no aplica. En todos los casos, $T(1) = 1$ y $f(n) = n^3$ cuando n es par y n^2 en caso contrario.

(1) $T(n) = 4T(n/2) + 3n^2$ (2) $T(n) = 2^n T(n/8) + 1$

(3) $T(n) = 3T(n/2) + n^2 \log n$ (4) $T(n) = 3T(n/2) + f(n)$

Ejercicio 4

Considere la siguiente especificación de un TAD Conjunto.

TAD Conjunto(α)

igualdad observacional
$(\forall c, c' : \text{set}(\alpha))\ (c =_{obs} c' \iff (\forall x : \alpha)(\text{esta}(x, \text{toSecu}(c)) = \text{esta}(x, \text{toSecu}(c'))))$

gros set(α)

observadores básicos

toSecu	: set(α)	$\longrightarrow$ secu(α)	

generadores

$\emptyset$	:	$\longrightarrow$ set(α)	
Add	: $\alpha \times$ set(α)	$\longrightarrow$ set(α)	

otras operaciones

esPermutacion	: set(α) $\times$ secu(α)	$\longrightarrow$ bool	
$\emptyset$?	: set(α)	$\longrightarrow$ bool	
$\bullet - \{\bullet\}$	: set(α) $\times$ α	$\longrightarrow$ set(α)	$\{\neg\emptyset?(c)\}$
oneOf	: set(α) c	$\longrightarrow$ α	$\{\neg\emptyset?(c)\}$
woOne	: set(α) c	$\longrightarrow$ set(α)	
...			
toSet	: secu(α)	$\longrightarrow$ set(α)	

axiomas ...

esPermutacion($\emptyset$, s)	$\equiv$ vacia?(s)
esPermutacion(Add(x, c), s)	$\equiv$ esta(x, s) $\wedge$ esPermutacion($c - \{x\}$, sinElemento(x, s))
$\emptyset$?(c)	$\equiv$ vacia?(toSecu(c))
$c - \{x\}$	$\equiv$ toSet(sinElemento(x, toSecu(c))
oneOf(c)	$\equiv$ prim(toSecu(c))
woOne(c)	$\equiv$ $c - \{\text{oneOf}(c)\}$
...	
toSet(s)	$\equiv$ **if** vacia?(s) **then** $\emptyset$ **else** Ag(prim(s), toSet(fin(s))) **fi**
...	
esPermutacion(c, toSecu(c))	$\equiv$ true //Nota: axioma para definir toSecu

Fin TAD

(a) ¿Esta especificación es correcta de acuerdo a la descripción usual de conjunto? **Justifique.**

(b) Suponiendo que la especificación fuera correcta,

I. ¿es mejor que la especificación vista en la materia? **Justifique**, explicitando qué considera mejor.

II. ¿se podrá demostrar que ambos TADs son "equivalentes"? Formalmente, ¿se podrá demostrar que

$$\text{Ag}(x_1, \ldots \text{Ag}(x_k, \emptyset) \ldots) =_{obs} \text{Ag}(y_1, \ldots \text{Ag}(y_j, \emptyset), \ldots)$$

si y sólo si

$$\text{Add}(x_1, \ldots (\text{Add}(x_k, \emptyset) \ldots) =_{obs} \text{Add}(y_1, \ldots (\text{Add}(y_j, \emptyset), \ldots)?$$

No hace falta hacer la demostración o encontrar un contraejemplo, sólo **justificar**.

III. ¿se podrá demostrar la siguiente identidad? **Justificar.**

$$\text{dameUno}(\text{Ag}(x_1, \ldots \text{Ag}(x_k, \emptyset) \ldots)) =_{obs} \text{oneOf}(\text{Add}(x_1, \ldots \text{Add}(x_k, \emptyset) \ldots))$$

1/7 [redacted]

Formas 9 (nueva)

Libreta: [redacted]

1	2	3	4
B	B-	B	B

e. cola sobre?

Mestruclera de Representacion

para la estructura usaría primero un Arbol binario, el cual tendrá un Invariante Heap (puede ser Max-Heap o Min Heap segun nos convenga) para poder encolar en log(n) y desencolar en log(n) y pedir el próximo O(1) pero cuando encolamos en la e. cola nos devuelva un iterador al nodo encolado, pero además tendría un e. DiccOrdenado el cual tendra la estructura de un AVL Asi podremos buscar, Agregar y eliminar en log(n) (ya que por cada elemento agregado tendremos una clave asociada por lo tanto n tambien seria la cantidad de claves) entonces como significa no tendra el iterador a la al nodo que esta e. cola (acordemonos que buscar en un heap cuesta O(n) y nosotros con el contexto de us. podido no podemos buscar el elemento ya que no nos daria la completitud), y ademas debemos tener e.total que sera un Nat y no servira como guia para poder Asignar las claves

colaPriorInd se representa con estr

donde estr es tupla < e.cola: colaprior(α), e: DiccOrdenado : diccOrd(clave, iteradorDeCola), e: total: Nat >

Funcion Insertar: Bueno Agarro encolamos en la colaprioridad una tupla que sera <elemento, e.total-1> eso toma log(n) y nos guardamos el iterador, la clave que usaremos sera segun el elemento que fueron Agregados (osea la idea seria es ir Asignando clave del ~~0 en adelante~~ cero en adelante)

entonces ~~una estaba corriendo~~ ~~la buscar~~ Despues
definimos e.DiccOrdenado como clave ~~en~~ e.totales y
como significado el iterador ~~p~~, ~~a~~ esto tomaría log(n)
y por ultimo Hacemos e.totales ~~++~~ la cantidad
que ya tenia que sume uno mas y le tornamos
e.totales - 1.
entonces la complejidad seria Log(n).

Función tope: Bueno Aqui solo le pedimos el proximo
a e.colo esto tomaria O(1)
entonces la complejidad seria O(1)

Función Desencolar: para poder desencolar primero
pido el proximo esto toma O(1) y despues de Hi
Aplicamos desencolar en la cola de prioridad
esto toma log(n) y despues como nos guardamos el
proximo y sabemos que es una tupla ~~donde~~ podemos
pedirle que nos de ~~el~~ π_2 ~~[redacted]~~ el cual ~~es~~
~~la~~ contendra la clave y con esa clave podemos
- Borrar de e.diccordenado en log(n).

entonces esto tomaria en complejidad log(n)
función
- Cambiar prioridad: Bueno la idea para poder
cambiarle de prioridad es poder Buscarlo en
e.DiccOrdenado ~~en~~ ~~la~~ esto nos cuesta log(n) y como
significado nos devuelve el iterador y despues con el
iterador (Acordamos que el iterador dentro de su estruc
tura tiene la ~~esta~~ e.cola) ~~Bueno~~ podemos Hacer el

2/7

Libreta [redacted]

Subir o bajar A si vemos donde hacen le corresponde estar elemento en la cola de prioridad y esto toma $\log(n)$ (y hay que acordarnos que como mínimo puede pasar porque el elemento era raíz y hay que hacer una hoja o al nieves y como la altura del arbol es $\log(n)$ ~~[illegible]~~ y listo. Cabe aclarar que no es necesario notificar el iterador ya que no da error cambiar de lugar por lo tanto el iterador lo sigue apuntando).

∴ La complejidad sera $O(\log(n))$

3/7

Libreta: [redacted]

B=

2) para la Interfaz del TAD diccionario con claves ordenadas sería:

Vacio ( ) : el cual devolvería un diccionario vacio, la precondición sería true, su poscondicion sería $\widehat{res}$ = obs Vacio(), su complejidad sería O(1) y no tendríamos NADA de Aliasing

No es apropiado para discurso escrito

Inserción : ~~Bueno~~ aquí primero debemos decir que hay un efecto colateral ya que el Diccionario que recibimos como parámetro será modificado puesto tanto en la precondicion dice $d = do$ (usamos la notacion de cambio de estado) y postcondición $\hat{d}$ = obs definir($\hat{k}$, $\hat{s}$, do), la complejidad sería log(n) (por el contexto uso que nos pidieron) y diremos en Aliasing que si K,S el copiable lo cuesta O(1) lo Hacemos por copia, pero sino lo Hacemos por referencia.

Falta definido?

Busqueda : ~~Bueno~~ Aquí no hay efectos colaterales, y que en la precondición es siempre true y en la postcondición $\widehat{res}$ = obs buscar($\hat{k}$, $\hat{d}$) (en la especificacion del tad diccionario con claves ordenadas debería estar la operación buscar Asi la podemos MAPear por medio de la función ^ sombrerito), complejidad sería log(n) (por el contexto de uso) y en la parte de Aliasing debemos decir que el significado se devuelve por referencia.

Borrado: Aquí tenemos un efecto colateral en el diccionario pasando por ~~para~~ valor por lo tanto precondición va decir $d = do$ (cabe aclarar que si el elemento a borrar está o no está en el diccionario eso no lo dice la especificación yo tomo la decisión que pueda borrar elementos que no están) y, en la poscondición debe decir $\hat{d} = Borrar(\hat{K}, do)$, la complejidad sería $\log(n)$ (por el contexto nevera).

para poder recorrer el diccionario voy a usar un iterador unidireccional el cual tendrá funciones como Hay Siguiente, Siguiente, Avanzar, crear it.

Crear it: no hay efectos colaterales, la precondición es siempre true, y en la post-condición debe decir que Alias (¿Permutación? (Dicc Seq($\hat{d}$), res))
(osea que de alguna forma estamos diciendo que relación hay entre el diccionario y el iterador), la complejidad va a ser $O(1)$, el iterador se invalida si casualmente se borra un elemento ~~de~~ a, ~~en~~ el cual si yo hacía Siguiente(it) = a

Hay Siguiente: no hay efectos colaterales, esta precondición es siempre true y en la poscondición $\hat{res} = hay\ Siguiente(\hat{it})$ complejidad va a ser $O(1)$.

Siguiente: acá si hay efectos colaterales y que se va modificar el iterador entonces en la precondición

4/7

Li Bueto [redacted]

Dice it = ito ∧ HaySiguiente(it) y en la post condicion nos dice que it = Siguiente(it₀), la complejidad sera O(1)

Avanzar: Acá Hay efectos colaterales ya que el iterador se va a modificar, entonces en la precondicion tenemos it = ito ∧ HaySiguiente(it) y para la post condicion tenemos que it = Avanzar(ito), la complejidad sera O(1).

estructura de representación

Diccionario se va a representar sobre un AVL ya que sabemos que en un AVL el insertar, borrar y buscar un elemento es log(n) donde n es la cantidad de elementos y la estructura del iterador voy a usar una pila donde voy a tener el pila, voy a hacer un pila de punteros así mientras voy recorriendo el árbol va ir sacando los punteros (ojo yo no quiero que los saque todos los nodos en la pila, sino la idea es cuando se crea el iterador la pila solo tenga apilando el puntero a la raiz del árbol AVL entonces cuando se aplique la funcion hay siguiente se fija si la pila es vacia o no, la funcion siguiente lo unico que hace es pedirle el dato al tope de la pila y por último la funcion Avanzar lo que va a ser es pedir el tope de la pila, desapilar la pila y despues

como tenemos guardado el tope de la pila (la cual desapilamos), solo apilamos su hijo der (si lo tiene) y su hijo izq (si lo tiene) y listo. Avanzar, cabe aclarar que con esta forma nunca vamos a tener almacenados todos los elementos del arbol AVL por lo tanto la complejidad temporal sería O(n), ya que solo recorremos una vez cada nodo y la complejidad espacial es O(1)). y tendríamos e.arbol que sería un puntero a la raiz del AVL (así puedo ~~acceder~~ sacar la raiz del arbol al crear el iterador)

Esto requiere O(log n) de memoria

estructura del iterador

iter se representa con estriter donde estriter es tupla <(e.pila : pila(puntero(nodo))), e.arbol : puntero(nodo)>

5/7

Libreta: [redacted]

3) B-

Para aplicar el método maestro debemos cumplir ciertas condiciones: $a \geq 1$, $b > 1$ (donde a y b constante) y $f(n)$ debe ser una función asintóticamente positiva, y se aplica a estas funciones de recurrencia

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(\frac{n}{b}) + f(n) & n > 1 \end{cases}$$

y nos da 3 casos en los cuales puede caer (cabe aclarar hay casos en los que no funciona)

$$\begin{cases} T(n) \subseteq \Theta(n^{\log_b a}), & \text{si } \exists \varepsilon > 0 / f(n) \subseteq O(n^{\log_b a - \varepsilon}) \\ T(n) \subseteq \Theta(n^{\log_b a} \log(n)), & \text{si } f(n) \subseteq \Theta(n^{\log_b a}) \\ T(n) \subseteq \Theta(f(n)), & \text{si } \exists \varepsilon > 0 / f(n) \subseteq \Omega(n^{\log_b a + \varepsilon}) \end{cases}$$

y si además cumple la condición de regularidad que dice $\exists c < 1 / \forall n \geq n_0$ se cumple que

$$a f(\frac{n}{b}) \geq c f(n)$$

1) $T(n) = 4T(\frac{n}{2}) + 3n^2$

$a = 4 \quad b = 2 \quad f(n) = n^2$

$n^2 \subseteq \Theta(n^{\log_2 4}) \Rightarrow n^2 \Theta(n^2)$ esto quiere decir que cae en el segundo caso del teorema maestro por lo tanto $T(n) \subseteq \Theta(n^2 \log(n))$ ✓

2) $T(n) = 2^n T(n/8) + 1$

$a = 2^n \quad f(n) = 1$

$b = 8$

en este caso no podemos aplicar el metodo maestro ya que $a = 2^n$ no es una constante.

3) $T(n) = 3T(\frac{n}{2}) + n^2 \log(n)$

$a = 3 \quad f(n) = n^2 \log(n)$

$b = 2$

Vemos claramente que quiere caer en el tercer caso bueno pasemos a analizarlo

$n^2 \log(n) \in \Omega(n^{\log_2 3 + \varepsilon}) \Rightarrow n^2 \log(n) \in \Omega(n^{1,4+\varepsilon})$

$\varepsilon = 0,4$ esto Vemos que cumple, pero nos falta ver si cumple la condición de regularidad que dice $\exists\, c<1 \;/\; n \geq$ no se cumple que

$$a f(\tfrac{n}{b}) \geq c f(n) \Rightarrow 3(\tfrac{n}{2})^2 \log(n/2) \geq c n^2 \log(n)$$

$$\Rightarrow \frac{3}{4} n^2 \log(\tfrac{n}{2}) \geq c n^2 \log(n)$$

Vemos que esto cumple para $c < \frac{3}{4}$

$$T(n) \in \Theta(n^2 \log(n))$$

6/7

Alumno: [redacted]

4) $T(n) = 3T\left(\frac{n}{2}\right) + f(n)$ donde $f(n) = \begin{cases} n^3 & \text{es par} \\ n^2 & \text{es impar} \end{cases}$

$a = 3$

$b = 2$

$f(n) = \begin{cases} n^3 \text{ es par} \\ n^2 \text{ es impar} \end{cases}$

por lo que vemos esto entra en el tercer caso analizamos

$f(n) \in \Omega\left(n^{\log_b a + \varepsilon}\right) \Rightarrow f(n) \in \Omega\left(n^{\log_2 3 + \varepsilon}\right)$

$\varepsilon = 0{,}4$ tenemos $f(n) \in \Omega(n^2)$ esto cumple pero falta ver si cumple la condición de regularidad que dice $\exists\, c < 1 / \forall\, n \geq n_0$ se cumple que $a f\left(\frac{n}{b}\right) \geq c f(n)$

Veamos el caso primero el caso impar

$3 \frac{n^2}{4} \geq c n^2$ esto vale $\forall\, c < \frac{3}{4}$

Veamos el caso segundo donde n es par

$3 \frac{n^3}{8} \geq c n^3$ esto vale $\forall\, c < \frac{3}{8}$

~~cualquier~~ ~~$\frac{3}{8}$~~ ~~que cumple~~

$\frac{3}{8} < 1$

con esto vemos que el teorema no sirve no nos dice nada ya que vemos que en cada caso necesitamos una constante distinta.

7|7

Libreta: [redacted]

4) a) No, ya que Hay algunas cosas que estan mal

I) La función OneOf(c) esta sobreespecificando ya que dice que siempre me devuelve primero elemento de la secuencia y ~~esto esto~~ cuando quiera ya sea la etapa de diseño me va a limitar la forma implementar el tad conjunto.

b) I) No, ya que esta especificacion esta ~~muy~~ Resolviendo el qué del conjunto, pero de un otro lado esta haciendo el como, lo cual deja, que cuando lo queramos ~~usar~~ para el diseño, tengo muchas limitaciones, en cambio el tad conjunto de la catedra te da muchas mas posibilidades a la hora de querer ~~diseñar~~ diseñarlo y nos da una amplia variedad de soluciones.

II) No, ya que dame Uno como esta axiomatizado en la catedra nos dice solo que nos da un elemento que pertenece al conjunto, pero no sabemos cual es, en cambio la función OneOf nos da un elemento que pertenece al conjunto y ademas sabemos que elemento nos va devolver (ya que siempre devuelve el primer elemento de toSecu) por lo tanto la demostracion no se podria demostrar

II) Si, ya que los dos generadores tanto A_g, A_{dd} Hacen el mismo comportamiento (el cual es Agregar un elemento más al conjunto), cabe aclarar que lo Igualan observa como el pertenece como observador Siempre utilizan el pertenece creados tanto por lo tanto las instancias del generador add van a por lo tanto los generadores A_g como el generador add van a tener la misma semántica, por lo tanto la demostración se podría demostrar formalmente