Algoritmos y Estructuras de Datos II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Examen Final

12 de septiembre 2022

Integrante	LU	Correo electrónico
Ezequiel Rueda Sanchez	522/16	ezequiel.ruedasanchez@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Ejercicio 1

¿Que implica en la etapa de diseño que la especificación sea: inconsistente, subespecificada, sobreespecificada.?

Resolucion

Un TAD es **inconsitente** cuando la axiomatizacion de alguna operacion produce distintos resultados para la misma entrada. Recordemos que las axiomatizaciones no se "evaluan" en orden. Es decir, valen todos los axiomas al mismo tiempo para todas las posibles combinaciones de argumentos de entrada (no existe una evaluacion secuencial top-down) de los axiomas.

Por ejemplo, esta axiomatizacion produce una inconsistencia ya que la secuencia de entrada `3 • S` es evaluada en ambos axiomas pero una devuelve 3 y otra devuelve 21.

`operacion(3 • S)` $\equiv 21$

`operacion(a • S)` $\equiv$ `a`

Un TAD esta **sobreespecificado** si tenemos varias formas de obtener el resultado de una operacion para una determinada entrada. Es decir, tenemos axiomas "de mas". Esto es legal siempre y cuando el TAD sea consistente (llegamos al mismo resultado utilizando cualquier combinacion de axiomas).

`operacion(3 • S)` $\equiv 3$

`operacion(a • S)` $\equiv$ `a`

Es una sobreespecificacion porque cuando el primer elemento es un 3, hay dos formas de obtener el resultado para dicha entrada.

El problema principal de sobreespecificar es que el TAD puede resultar confuso. Siempre conviene mantener los axiomas lo mas minimal posible, y que haya una forma inequivoca de aplicar los axiomas para obtener el resultado de alguna operacion.

Tambien se sobreespecifica cuando en el TAD se utiliza algun otro tipo de dato que tiene ciertos comportamientos no relevantes para nuestro problema. Por ejemplo, si estamos modelando un carrito de supermercado que contiene productos, estos podrian ser una secuencia o un conjunto ¿Cual conviene usar? Si usamos una secuencia estamos dandole un orden los productos, pero si no importa el orden conviene usar un conjunto para permitir en la etapa de diseño mas opciones de estructuras.

Por ultimo, tenemos varias maneras de **subespecificar**. La mas comun sucede cuando restringimos el dominio de alguna operacion. Por definicion de un TAD, todas las operaciones son totales (estan definidas en todo su dominio). No obstante, es comun que ciertas operaciones simplemente no tengan sentido para un subconjunto del dominio, ya sea porque es imposible determinar el resultado (por ejemplo dividir por 0) porque no son casos relevantes en el contexto de uso. Por ejemplo, una operacion que recibe como argumento un monto de dinero podria solo tener sentido si el monto es ≥ 0.

Cuando aplicamos una restriccion sobre el dominio de una operacion, lo que estamos haciendo es recortar el dominio a los valores relevantes para nuestro problema, y solo considerar esos valores en los axiomas, sin decir que pasa con los valores restringidos. Esto simplifica la axiomatizacion. Durante la etapa de diseño, se debe decidir que hacer al recibir una entrada restringida. Se podria simplemente no hacer nada si la operacion se puede realizar de todas formas, o se fuerza cierto valor valido (si el monto era $<$ 0 lo consideramos como 0) o podemos implementar programacion defensiva y chequear si la entrada es restringida y en ese caso devolver un error.

Similar al caso anterior, otra forma de subespecificar es no axiomatizar para ciertas entradas pero sin poner una restriccion. Esto es un problema porque no sabemos cual es el resultado para ciertas operaciones, y por lo tanto seria imposible diseñar los algoritmos del TAD.

Hay otra forma de subespecificar que es mas sutil pero muy util. Sirve para posponer la definicion de algunos aspectos funcionales de las operaciones hasta la etapa de diseño e implementacion. Cuando estamos escribiendo un TAD, hay partes del problema que quizas no tenemos definicion aun, pero que tampoco son indispensables para modelar el comportamiento.

Por ejemplo, supongamos que estamos modelando un examen final de AEDII el cual se toma de forma oral, y tenemos una

operacion para llamar al siguiente alumno al aula. Durante el modelado, no sabemos aun que criterio se va a utilizar para llamar a los alumnos. Podria ser alfabetico, por DNI, por LU, etc. Si en la axiomatizacion de llamar al siguiente alumno, declaramos `prim(ordenarAlfabeticamente(alumnosQueRinden))` estamos preescribiendo exactamente la forma en que se llaman a los alumnos. Pero aun no sabemos cual es el criterio. Podemos entonces intencionalmente subespecificar el problema y decir `dameUno(alumnosQueRinden)` para luego definir el criterio exacto durante la implementacion.

2. Ejercicio 2

Escribir una versión con el método D&C del algoritmo Heapify que dado un árbol binario completo devuelva un Árbol binario Heap. Dar su complejidad.

Resolucion

---

floydAlgorithm(**in** T : `ab(nat)`) → **out** H : `ab(nat)`

```
1:  if esHoja?(T) then                                   ▷ O(1)
2:      H ← T                                            ▷ O(1)
3:      return H                                         ▷ O(1)
4:  else
5:      raiz(H) ← raiz(T)                                ▷ O(1)
6:      izq(H) ← floydAlgorithm(izq(T))                  ▷ T(n/2)
7:      der(H) ← floydAlgorithm(der(T))                  ▷ T(n/2)
8:      H ← siftDown(H)                                  ▷ O(log(n))
9:      return H                                         ▷ O(1)
10: end if
```

Complejidad: $\Theta(n)$
Justificación: De la linea 1 a la linea 3 tenemos el caso base donde el arbol es una hoja. En ese caso, ya se encuentra heapificado por lo que retornamos a el mismo y esta operacion tiene costo $O(1)$.

Ahora, de la linea 4 a la linea 10, tenemos el caso D&Q. En ese caso, comenzamos heapificando el subarbol izquierdo y derecho y lo guardamos en los subarboles izq y der. Realizamos esta operacion hasta llegar al caso base. Una vez que ambos subarboles se encuentran heapificados, se compara el valor de la raiz de con el valor de las raices de ambos subarboles. En el caso que el maximo de las raices de los subarboles sea mayor al valor de la raiz, se intercambian las posiciones en el arbol. Esta operacion fue definida en las clases teoricas, se llama $siftDown$ y su costo es $O(log(n))$.

Como en cada paso recursivo, partimos el arbol en $c = 2$ subarboles y analizamos los $a = 2$ subarboles derecho e izquierdo. Por lo tanto, si escribimos la funcion en tiempo de peor caso de manera recursiva obtenemos:

$$Heapify(a) = \begin{cases} 1, \text{ si } esHoja?(T) \\ 2T(\frac{n}{2}) + log(n), \text{ en otro caso} \end{cases}$$

Tenemos una recurrencia de la forma $T(n) = aT(\frac{n}{c}) + f(n)$. En este caso, $a = 2$, $c = 2$ y $f(n) = n$. Luego, $log_c(a) = log_2(2) = 1$ y como $f(n) = log(n)$, $f(n) = O(n^{log_c(a)})$, es decir, $f(n) = O(n)$. Concluimos, por el punto 1 del Teorema Maestro que $T(n) = \Theta(n^{log_c(a)}) = \Theta(n^{log_2(2)}) = \Theta(n^1) = \Theta(n)$.

---

3. Ejercicio 3

De los algoritmos de ordenamiento vistos en clase, liste aquellos en los cuales es posible obtener resultados en tiempo de ejecución.

4. Ejercicio 4

Dada la implementacion de una funcion de hash con doble direccionamiento que utiliza las funciones h_1 y h_2, una devuelve siempre el mismo valor distinto de 0. Explique que se puede esperar del comportamiento de la función. Primero si falla h_1 y luego si falla h_2.

Resolucion

Sabemos que en hashing doble tenemos la ecuacion $h(k,i) = (h_1(k) + ih_2(k)) \bmod |T|$ con $|T|$ longitud de la tabla.

. Comenzamos viendo el caso donde h_1 es una constante.

En este caso, $h(k,i) = (c + ih_2(k)) \bmod |T|$.

El primer hasheo de cualquier cualquier clave siempre resulta c pues $i = 0$. Por lo tanto, una vez que hayamos insertado la primer clave, todas las futuras claves van a colisionar en el primer intento y tendremos que realizar un barrido para encontrar su posicion. Dada una clave k que colisiono, $h_2(k)$ resulta constante para esa clave en particular y por lo tanto, el barrido realiza saltos de $h_2(k)$ posiciones.

Puede haber aglomeracion secundaria si h_2 produce el mismo valor para dos claves distintas, ya que en ese caso van a tener la misma secuencia de barrido a partir de la colision inicial con $i = 1$. Pero si h_2 produce valores distintos para 2 claves, ante una colision ya no necesariamente es necesario realizar la misma secuencia de barrido hasta encontrar una posicion libre pues los saltos de los barridos de las 2 claves serian distintos (pero siguen siendo barridos lineales y existe la posibilidad de generar secuencias de barrido iguales dependiendo de la constante usada y el tamaño $|T|$).

. Continuamos analizando el caso donde h_2 es una constante.

En este caso, $h(k,i) = (h_1(k) + ic) \bmod |T|$.

La funcion de hash genera algo muy similar al barrido lineal en donde la posicion inicial esta determinado por $h_1(k)$ y el barrido realiza saltos determinados por la constante $c = h_2$. A diferencia del caso anterior, la clave solo determina la posicion inicial y el salto durante el barrido esta fijo para todas claves por igual.

Puede haber aglomeracion primaria, ya que en cualquier clave que colisiona con la aglomeracion, va a seguir colisionando durante el barrido hasta llegar al "final" de la misma.

Concluimos que si utilizamos una constante para h_1 o h_2 en una funcion de hashing doble, lo que sucede es que degradamos la funcion an hashing simple con barridos lineales. En ambos casos sucede que para la mayoria de las constante no vamos a poder asegurar que la funcion de hash produzca una permutacion de todas las posiciones posibles de T.