Algoritmos y Estructuras de Datos II

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

Examen Final

12 de abril 2023

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

Explique la relacion entre el invariante de representacion y complejidad algoritmica, y entre funcion de abstraccion y la demostracion de que el diseño es correcto con respecto a la especificacion.

Resolucion

La relacion entre el invariante de representacion y la complejidad algoritmica se encuentra en que el invariante al ser un predicado que da *true* cuando recibe una instancia valida, determina los casos o el dominio de las funciones que tendran que ser implementadas en los algoritmos. Esto determina en parte la complejidad algoritmica porque define que casos se van a considerar y en la eleccion de la estructura para implementar los algoritmos.

Luego, la relacion entre la funcion de abstraccion y la demostracion de que el diseño es correcto con respecto a la especificacion es que la funcion de abstraccion es una herramienta que permite demostrar que el diseño es correcto con respecto a la especificacion. La funcion de abstraccion iguala los observadores basicos definidos en la especificacion con algun componente de la estructura de representacion elegida. Si ambos reflejan observacionalmente lo mismo, entonces podemos afirmar que el diseño es correcto con respecto a la especificacion siempre y cuando considerando que se cumpla el invariante de representacion.

2. Ejercicio 2

Justifique por que las colas de prioridad son ineficientes para implementar busquedas aun a pesar de ser arboles balanceados. Explique por que no pueden modificarse para que sean eficientes en esto sin perder una de sus propiedades fundamentales ¿Que estructura propondria para implementar busquedas generales y de minimo eficientes?

Resolucion

Las colas de prioridad son ineficientes para implementar busquedas porque a pesar de ser arboles balanceados, los elementos en este estan ordenados de acuerdo a una prioridad y a prori, no se sabe cual es la ubicacion de cada uno de ellos. Solo el elemento de maxima prioridad puede ser accedido en tiempo $O(1)$ ya que se encuentra en la raiz pero cuando se realiza una busqueda, no se conoce la prioridad de un elemento. Es decir, es una estructura muy util para realizar *sorting* y no *searching*.

Una buena modificacion a esta estructura seria poder almacenar los elementos que contengan claves menores al valor raiz en el subarbol izquierdo y almacenar los elementos que contengan claves mayores a la raiz en el subarbol derecho, es decir, convertir al heap es un ABB lo cual lograria una complejidad en la busqueda de orden $O(log(n))$ en lugar de $O(n)$ pero sabemos que un heap no es un ABB y en este caso, no se preservaria el invariante de heap donde no necesariamente tiene que haber elementos menores a la raiz del subarbol derecho y elementos mayores a la raiz del subarbol derecho.

Ahora, voy a proponer dos estructuras para implementar busquedas generales. Una de ellas son los **arboles AVL** que permiten realizar busqueda en tiempo logaritmico dado que son arboles balanceados, es decir, son arboles que cumplen el invariante de ABB mencionado anteriormente pero ademas el factor de balanceo para cada nodo es menor o igual a 1 en modulo, es decir, $-1 \leq altura(der) - altura(izq) \leq 1$. Otra estructura eficiente para implementar busquedas eficientes son los **tries**. Esta estructura utiliza componentes de las claves para realizar la busquedas. Es decir, si las claves son numeros enteros, utiliza sus digitos para realizar comparaciones y realizar la busqueda; y si las claves son string, utiliza sus caracteres para realizar comparaciones y realizar la busqueda. Esto resulta eficiente porque se pueden implementar sobre arboles donde la complejidad en peor caso va a ser $O(long(T))$ donde $long(T)$ es la longitud de la clave mas larga. Su altura tambien esta determinada por $long(T)$. Es mas, si las claves se encuentran acotadas, se van a poder realizar busquedas en tiempo $O(1)$ dado que en todos los casos se va a recorrer una cantidad de veces constante el arbol para realizar la busqueda.

Con respecto a la busqueda de minimo, una estructura eficiente que realiza esto son los **min-heap**. En este caso, el minimo siempre se encuentra en la raiz del arbol y esta operacion tiene costo $O(1)$. Otra estructura eficiente que realiza la busqueda de minimo son los **Arboles AVL** dado que el minimo elemento es aquel que se encuentra en el nodo "mas a la izquierda" del arbol y como el arbol se encuentra balanceado, coincide con su altura, es decir, tiene complejidad $O(log(n))$. Por ultimo, una estructura que podria ser eficiente para la busqueda de minimo, podrian ser los ABB pero estos al no estar necesiaramente balanceados, podriamos tener una secuencia de inserciones en orden decreciente, lo cual acceder al minimo tendria costo $O(n)$, es decir, habria que recorrer todos los nodos y justamente es lo que queremos evitar.

3. Ejercicio 3

Justifique detalladamente la existencia de una cota inferior para la complejidad temporal asociada a ordenar un arreglo de numeros naturales sobre los que no se tiene ninguna hipotesis adicional.

Resolucion

La existencia de una cota inferior para la complejidad temporal asociada a ordenar un arreglo de numeros naturales sobre los que no se tiene ninguna hipotesis adicional indica que cualquier algoritmo que se proponga va a tener una complejidad de, al menos, la de la cota superior propuesta.

Esto favorece a la hora de implementar el algoritmo de sorting porque gracias a la cota inferior, tenemos una posible cantidad de estructuras cuya complejidad en el ordenamiento van a cumplir la de la cota inferior propuesta. Es mas, con los algoritmos vistos en clase, podriamos chequear cuales de esos algoritmos tienen complejidad que cumplan con la cota inferior y aplicarlos en la implementacion. Por ejemplo, si tenemos una cota inferior de n^2, o sea, de $\Omega(n^2)$, todos los algoritmos que implementemos tienen que tener complejidad al menos cuadratica y podriamos utilizar SelectionSort o InsertionSort. Si tenemos una cota inferior de $nlog(n)$, o sea, de $\Omega(nlog(n))$, todos los algoritmos que implementemos tienen que tener complejidad al menos $n \,.\, log(n)$ y podriamos utilizar Mergesort o Quicksort.

Otro aspecto importante sucede que al no tener ninguna hipotesis adicional de como estan ditribuidos los elementos en el arreglo pero al tener una cota inferior, al encarar la implementacion del ordenamiento sabemos cual es la complejidad algoritmica que puede tener el algoritmo como maximo. Es decir, no tenemos informacion sobre la distribucion de los elementos pero sabemos cual es la complejidad maxima y eso nos puede dar indicios de como encarar la implementacion.

4. Ejercicio 4

Responder verdadero o falso. Justificar o dar un contraejemplo respectivamente.

a) Sea S arreglo de claves representado por max-heap. Sean $S[i]$ y $S[j]$ claves del heap tal que $i < j$ y $S[i] < S[j]$, entonces el arreglo intercambiando $S[i]$ y $S[j]$ sigue siendo un max-heap. **Falso**

Lo muestro con un contrajemplo.

Supongamos que $S = [89, 67, 84, 66, 65, 82, 83, 1, 43, 21, 5, 79, 70]$. Ahora, tomo $i = 9$ y $j = 12$ con $S[i] = 21 < S[j] = 70$.

Al realizar el intercambio propuesto, obtenemos $S = [89, 67, 84, 66, 65, 82, 83, 1, 43, 70, 5, 79, 21]$ y no sigue siendo un max-heap porque el nodo 65 ahora tiene como hijo al nodo 70 y como $70 > 65$, se rompe la condicion de max-heap.

b) Sea S arreglo de claves representado por max-heap. Sean $S[i]$ y $S[j]$ claves del heap tal que $i < j$ y $S[i] > S[j]$. Entonces, el arreglo obtenido intercambiar de $S[i]$ y $S[j]$ sigue siendo max-heap. **Falso**

Supongamos que $S = [89, 67, 84]$. Ahora, tomo $i = 0$ y $j = 1$ con $S[i] = 89 > S[j] = 67$.

Al realizar el intercambio propuesto, obtenemos $S = [67, 89, 84]$ y no sigue siendo un max-heap porque el nodo 67 ahora tiene como hijo al nodo 89 y como $89 > 67$, se rompe la condicion de max-heap.

c) Un arbol B con exactamente 2 hijos por nodo, es igual a un AVL. **Falso**

La afirmacion es falsa porque a pesar de tener 2 hijos por nodo donde cada uno va a contener una sola clave y de acuerdo al invariante de arbol B, el subarbol izquierdo va a contener las claves menores al valor del nodo y el subarbol derecho va a contener las claves mayores al valor del nodo, no va a implicar que el arbol B sea izquierdista. Por ejemplo, si realizo las inserciones en un arbol vacio primero del 20 y luego del 24, voy a tener el ultimo nivel que no es izquierdista y rompe la condicion de AVL.

d) Un AVL perfectamente balanceado es igual a un arbol B. **Verdadero**

En este caso, la afirmacion es verdadera porque al ser perfectamente balanceado, las inserciones en el arbol B se van a realizar de la misma manera. Estamos considerando un arbol B con exactamente 2 hijos por nodo. Si el arbol B es un arbol-3 o un arbol-4, la afirmacion resultaria falsa.

5. Ejercicio 5

Suponiendo que un programador tiene un error en una implementacion de un diccionario con hashing doble, de modo que una de las dos funciones devuelve siempre el mismo valor (distinto de 0), describir lo que sucede en cada situacion (cuando es la primera funcion la que esta mal y cuando lo es la segunda).

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 c.

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

El primer hasheo de cualquier clave siempre resulta c pues $i = 0$. Por lo tanto, una vez que insertamos la primer clave, todas las futuras claves colisionan 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 c y el tamaño $|T|$).

. Continuamos analizando el caso donde h_2 es una constante c.

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

La funcion de hash genera una especie de barrido lineal porque 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 las claves.

Existe la posibilidad de que haya 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 h_1 o h_2 es una constante en una funcion de hashing doble, se degrada la funcion a hashing simple con barridos lineales.