Algoritmos y Estructuras de Datos 2

Final 2022-08-02

Ejercicio 1

Responda verdadero o falso justificando.

1. Lo que hace que una operación sea observador básico es que deba escribirse en base a los generadores.

   Falso. Cualquier operación puede ser axiomatizada sobre los generadores. Una operación es un observador básico si expone alguna característica distintiva de la instancia que permita diferenciarla de cualquier otra instancia. Así, el conjunto de observadores básicos generan las clases de equivalencia del TAD, donde cada clase es el conjunto de instancias que son observacionalmente iguales en el contexto de nuestro problema.

2. Si una operación rompe la congruencia debe ser transformada en observador básico.

   Verdadero. La congruencia se rompe si dado 2 instancias observacionalmente iguales, éstas dejan de serlo luego de aplicar la misma operación a ambas. Si esto sucede, significa que los cambios producidos por alguna otra operación no están siendo observados en su totalidad, y por lo tanto hay alguna característica de la instancia que no se está tomando en cuenta para generar las clases de equivalencia. Al convertir la operación que rompió la congruencia en un observador básico en efecto estamos ahora considerando esa característica para determinar la igualdad observacional, y consecuentemente no romper la congruencia.

3. Dos instancias del mismo TAD pueden ser observacionalmente iguales y aún así ser distinguibles por una operación.

   Falso. Asumiendo que los observadores están bien definidos para modelar el comportamiento deseado, si 2 instancias son observacionalmente iguales, al aplicar la misma operación sobre ambas instancias éstas deberían permanecer en la misma clase de equivalencia.

4. Si un enunciado dice "siempre que sucede A sucede inmediatamente B y B no puede suceder de ninguna otra manera" y la correspondiente axiomatización incluye las operaciones A y B entonces el TAD está mal escrito.

   Verdadero. B es un comportamiento automático, por lo tanto, como el nombre lo indica, los efectos de B deben suceder automáticamente durante la axiomatización de A. Si tenemos una operación explícita para B, estamos delegando al usuario la responsabilidad de aplicar B cada vez que sucede A. Pero así no tenemos forma de garantizar lo que nos piden (que siempre suceda B después de A), ya que el usuario puede simplemente no aplicar B a propósito o por olvido.

Ejercicio 2

Explique detalladamente el rol que juega el invariante de representación en el diseño jerárquico de tipos abstractos de datos. Relaciónelo con el concepto de complejidad algorítmica.

El invariante de representación nos permite determinar si una instancia de la estructura de representación es válida o no. Necesitamos que la estructura sea válida para poder determinar a qué instancia abstracta del TAD representa.

En todas las funciones de la interfaz se asume que el invariante vale en la Pre y en la Post. No necesariamente tiene que valer en el medio de la función, ni en funciones auxiliares que no son exportadas (no están en la interfaz).

¿De qué nos sirve esto? El invariante nos garantiza ciertas propiedades de la estructura que nos permiten diseñar algoritmos eficientes. Por ejemplo, el invariante de un AVL solo permite un factor de desbalanceo F tal que $|F| \leq 1$ y esto es fundamental para poder garantizar la complejidad $O(log(n))$ de sus operaciones. Como el invariante vale en la Pre, los algoritmos de búsqueda, inserción y borrado pueden apoyarse en esta propiedad para asumir el estado inicial del árbol antes de la operación. Es importante que también valga en la Post para que la propiedad siga valiendo en futuras operaciones.

No siempre se trata de garantizar una propiedad interesante para lograr algoritmos eficientes. A veces simplemente tenemos que garantizar que la estructura sea "coherente". Por ejemplo, supongamos una estructura para un TAD que modela un

supermercado con productos, donde tenemos los productos guardados de varias formas: un diccionario sobre un trie que mapea producto → precio, y un conjunto de productos en oferta. El invariante debe garantizar que los productos en oferta estén definidos en el diccionario de precios, pues sino sería un problema tener un producto en oferta para el cual no sabemos su precio.

El invariante y las estructuras utilizadas en los niveles inferiores impactan directamente sobre las complejidades algorítmicas que se podrían conseguir en las funciones de la interfaz. Solo definen una cota inferior, ya que nuestros algoritmos siempre pueden hacer las cosas de forma ineficiente. Por ejemplo no usar nunca la propiedad del AVL y utilizar otra estrategia de balanceo más ineficiente. O en términos de conseguir la mejor complejidad posible, si la estructura utiliza un diccionario sobre lista enlazada, no vamos a poder obtener un significado en menos de $O(n)$ por la naturaleza misma de la lista enlazada (en el peor caso tenemos que recorrer toda la lista para encontrar la clave). Sin embargo, si la estructura utiliza un diccionario sobre un trie, podemos conseguir un significado en $O(|\text{clave}|)$ solo por haber utilizado otra estructura y las propiedades que su invariante garantiza.

Ejercicio 3

Se necesita implementar diccionarios en forma tal que a las operaciones clásicas se le agrega la de devolver el elemento cuya clave es mínima. Analice la performance de los árboles AVL y Heaps. Proponga una estructura que permita implementar TODAS las operaciones en forma óptima. Explique informalmente por qué es óptima.

Los árboles AVL tienen complejidad $O(log(n))$ para las operaciones de búsqueda, inserción y borrado, donde n es la cantidad de nodos en el árbol. Por el invariante de representación del AVL (en realidad por el del ABB), para obtener el mínimo debemos caminar por el árbol desde la raíz siempre tomando la rama izquierda hasta llegar a una hoja. Esta operación tiene un costo de $O(log(n))$ ya que para llegar hasta la hoja de más a la izquierda debemos recorrer toda la altura del árbol, que es $log(n)$.

Un Heap (asumiendo un MinHeap) puede obtener la clave mínima en $O(1)$ ya que ésta está en el nodo raíz. No obstante si queremos borrar el mínimo la complejidad es $O(log(n))$ ya que debemos reestablecer el invariante del Heap. Insertar en el Heap también es $O(log(n))$, pero para buscar, a diferencia del AVL donde en cada nodo decidimos ir por la izquierda o la derecha según la comparación entre la clave buscada y la clave del nodo, en el Heap debemos buscar por ambas ramas, en efecto recorriendo todo el Heap entero en el peor caso lo cual resulta $O(n)$.

Una estructura posible podría ser así:

```
minDicc(K, V) es tupla ⟨ raiz: AVL(K, V), min: AVL(K, V) ⟩
AVL(K, V) es tupla ⟨ clave: K, significado: V, izq: AVL(K, V), der: AVL(K, V) ⟩
```

La componente `raiz` es un puntero a la raíz del AVL, y `min` es un puntero al nodo mínimo dentro del árbol.

De esta forma podemos obtener el mínimo en $O(1)$ al igual que el Heap. Durante las operaciones de inserción y borrado, además de realizar el algoritmo clásico del AVL, tenemos que mantener el puntero `min` actualizado como corresponde. Si insertamos una nueva clave que es menor que el mínimo actual, actualizamos `min` para que apunte al nuevo nodo insertado. Si borramos el mínimo, debemos actualizar `min` con el nuevo mínimo, que podemos obtenerlo al mismo tiempo que buscamos la clave a borrar, o podemos buscarlo luego de borrar el nodo. Ambas operaciones siguen siendo $O(log(n))$ ya que mantener actualizada la referencia al mínimo no demanda mayor complejidad (a lo sumo cambia la constante).

Ejercicio 4

Analice comparativamente ventajas y desventajas de la representación de diccionarios con Hashing Abierto y Cerrado, dando además ejemplos de contextos en los que preferiría una sobre la otra.

Hashing abierto utiliza un direccionamiento cerrado o concatenación para resolver las colisiones. La HashTable no contiene directamente los significados, sino que en cada posición tenemos una lista enlazada de claves y significados (por ejemplo con tuplas). De esta forma, cuando la función de hash produce la misma posición para 2 claves distintas, la colisión se resuelve simplemente concatenando una nueva tupla (clave, significado) a la lista enlazada ubicada en la posición asignada por la función de hash.

Cuando necesitamos obtener el significado de una clave, calculamos su posición en la HashTable y luego debemos iterar por la lista enlazada de esa posición hasta encontrar la clave. Suponiendo que la función de hash realiza una asignación uniforme a todas las posiciones de la HashTable podemos garantizar que en el caso promedio encontrar la clave será $O(1)$.

Hashing cerrado utiliza un direccionamiento abierto (sí, los nombres son confusos). Este direccionamiento asigna una única clave por posición de la HashTable, y en el caso de una colisión, se busca iterativamente nuevas posiciones libres para colocar la clave. Para lograr esto, se introduce un segundo argumento i a la función de hash que indica el número de barrido o número de intento para encontrarle la posición a una clave. La primer posición es calculada con $i = 0$, y en caso de una colisión, se recalcula la posición con $i = 1$, luego con $i = 1$ y así sucesivamente hasta que encontramos una posición libre o detectamos que se llenó la HashTable.

Hay varias formas de direccionamiento abierto. Sea T la HashTable, k la clave, i el número de intento:

- Barrido lineal: $h(k, i) = (h_1(k) + i) \bmod |T|$
- Barrido cuadrático: $h(k, i) = (h_1(k) + i^2) \bmod |T|$
- Hashing doble: $h(k, i) = (h_1(k) + ih_2(k)) \bmod |T|$

Hashing abierto puede ser más conveniente en contextos donde vamos a tener que eliminar claves de la HashTable, ya que esto simplemente requiere eliminar el nodo de la lista enlazada correspondiente (y si además la lista es doblemente enlazada y tenemos una referencia directa al nodo nos evitamos buscarlo para borrarlo).

Con hashing cerrado, borrar claves resulta mucho más trabajoso. No podemos simplemente limpiar la posición que tenía la clave en el HashTable, tenemos que marcar de alguna forma que borramos la clave que estaba en esa posición (y así diferenciarla de las posiciones vacías). Durante el barrido de otras claves, si estamos insertando una clave, podemos permitir reutilizar una posición que está marcada como borrada. Si estamos buscando una clave, no debemos frenar cuando encontramos una posición borrada, tenemos que considerarla como si estuviese ocupada y seguir barriendo hasta encontrar la clave o encontrar una posición que realmente esté vacía. Si frenamos la búsqueda cuando encontramos una posición borrada no estamos permitiendo acceder a todas las claves que se insertaron después de la clave que fue borrada y que además hayan colisionado con ella.

Ejercicio 5

Justifique detalladamente la existencia de una cota inferior para la complejidad temporal asociada a ordenar un arreglo de números naturales sobre los que no se tiene ninguna hipótesis adicional. Relacione con la complejidad de distintos algoritmos de ordenamiento vistos en la materia.

Cualquier algoritmo de ordenamiento basado en comparaciones entre los elementos tiene como cota inferior $\Omega(nlog(n))$ donde n es la cantidad de elementos a ser ordenados.

Podemos representar todas las posibles comparaciones que realiza el algoritmo en un árbol de decisión. Éste es un árbol binario donde cada nodo representa una comparación entre 2 elementos. Como la comparación produce un resultado binario (la comparación es verdadero o falso), cada hijo representa uno de los posibles resultados. Notemos que entonces todos los nodos tienen exactamente 0 o 2 hijos, ya que no pueden haber comparaciones que sean solo verdadero o solo falso. Las hojas ya no modelan una comparación sino que representan una posible permutación del arreglo de entrada.

El algoritmo inicia su ejecución desde la raíz del árbol de decisión, y luego de cada comparación, decide por cuál rama seguir hasta llegar a una hoja. El recorrido que realiza el algoritmo por el árbol determina el orden relativo entre todos los elementos del arreglo, y la hoja en donde termina indica la permutación del arreglo original que produce un arreglo ordenado según el criterio de ordenamiento.

Un arreglo de tamaño n tiene $n!$ permutaciones como máximo. Pueden ser menos si hay elementos repetidos, pero ésto no cambia la cota. Como el árbol de decisión modela todas las posibles permutaciones, vamos a tener $n!$ hojas en el árbol. Si h es la altura del árbol, la cantidad máxima de hojas que puede tener es 2^h, y por lo tanto vale que $n! \leq 2^h$. Utilizando la desigualdad de Stirling:

$log(n!) \leq log(2^h) \iff log(n!) \leq h \iff nlog(n) \leq h$

Esto nos dice que la altura h del árbol de decisión es a lo sumo $nlog(n)$. Como el algoritmo recorre el árbol desde la raíz hasta una hoja para ordenar n elementos, en efecto visita a lo sumo $nlog(n)$ nodos (o realiza $nlog(n)$ comparaciones / pasos). A partir de este resultado podemos concluir que cualquier algoritmo de ordenamiento basado en comparaciones tiene una cota inferior $\Omega(nlog(n))$.

En la práctica no es posible construir este árbol de decisión de forma eficiente y recorrerlo para ordenar el arreglo. Cada algoritmo utiliza diferentes estrategias para realizar la menor cantidad de comparaciones. MergeSort y HeapSort tienen una

complejidad $O(nlog(n))$, y considerando ahora la cota inferior $\Omega(nlog(n))$ podemos decir que tienen complejidad asintótica óptima (no podemos lograr una cota mejor).

Notemos que los algoritmos que logran una cota mejor, como el CountingSort que puede ordenar en tiempo lineal $O(n+k)$, no se basan en comparaciones y necesitan de alguna hipótesis adicional sobre el arreglo de entrada.