Estructuras de Datos y Algoritmos – IIC2133

I1

5 abril 2013

1. La evaluación computacional de expresiones aritméticas *infix* simples, en que cada operador aparece entre sus dos argumentos, tal como

5 * (((9 + 8) * (4 * 6)) + 7)

puede hacerse a partir de representar la expresión en notación *postfix*: cada operador aparece *después* que sus dos argumentos, en lugar de entre ellos, y los paréntesis no son necesarios:

5 9 8 + 4 6 * * 7 + *

a) **[2 puntos]** Da un algoritmo, escrito en un pseudo lenguaje de programación similar a C++, para evaluar expre-siones postfix simples, como la anterior. Tu algoritmo debe leer la expresión postfix elemento a elemento de izquierda a derecha: si es un número, debe guardarlo para procesarlo más adelante; si es un operador, debe realizar la operación correspondiente con los números guardados, y debe guardar el resultado para procesarlo más adelante (si fuera necesario).

El algoritmo está prácticamente descrito en el enunciado. Supongamos que la expresión viene en un archivo (o en un arreglo) de elementos de algún tipo (o clase) **T**, y que tenemos un stack **s** inicialmente vacío. La clase **T** cuenta con métodos para saber si un objeto de tipo **T** es un número (en ese caso se puede calcular su valor) o un operador ('+' o '*').

```
while (!file.eof())
       item = file.read()
       if (item.isNumber())
              s.push(item.value())
       if (item.isPlus())
              s.push(s.pop()+s.pop())
       if (item.isTimes())
              s.push(s.pop()*s.pop())
print(s.pop())
```

b) **[3 puntos]** Da un algoritmo, escrito en un pseudo lenguaje de programación similar a C++, para convertir ex-presiones infix simples en expresiones postfix.

Observando el ejemplo, notamos que los números aparecen en la expresión postfix exactamente en el mismo orden en que aparecen el la expresión infix; por lo tanto, cada vez que leemos un número, lo escribimos. También notamos que cada vez que en la expresión infix aparece un ')', en la expresión postfix aparece el primer operador a la izquierda del ')' que aún no había aparecido en la expresión postfix. Por lo tanto, cada vez que leemos un operador, lo ponemos en un stack, y cada vez que leemos un ')', escribimos el operador que está en el tope del stack (y lo sacamos del stack). Finalmente, ignoramos los '('. Para simplificar un poquito el algoritmo, conviene poner toda la expresión infix entre paréntesis.

Supongamos que la expresión infix viene en un archivo (o en un arreglo) de elementos de algún tipo (o clase) **T**, que vamos a escribir la expresión postfix resultante en otro archivo, y que tenemos un stack **s** inicialmente vacío. La clase **T** cuenta con métodos para saber si un objeto de tipo **T** es un número, un paréntesis ('(' o ')'), o un operador ('+' o '*').

```
while (!fileA.eof())
       item = fileA.read()
       if (item.isNumber())
              fileB.write(item)
       if (item.isPlus() || item.isTimes())
              s.push(item)
       if (item.isLeft())
              —lo ignoramos
       if (item.isRight())
              fileB.write(s.pop())
```

c) **[1 punto]** ¿Cuáles son las complejidades de tus algoritmos de **a**) y **b**), en función del número de elementos de la expresión procesada?

Ambos algoritmos son O(*n*), en que *n* es el número de elementos en la expresión procesada. Leemos cada elemento de la expresión una vez, y, por cada elemento leido, realizamos un número constante de pasos: primero, averiguamos el tipo del elemento (tres o cuatro opciones); y luego, hacemos uno o dos push's, pop's o write's.

2. En el caso de hashing con encadenamiento, propón una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una *lista ligada de casilleros disponibles*. Para esto, considera que cada casillero puede almacenar un *boolean* y, ya sea, un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y las que manejan la lista debieran correr en tiempo $O(1)$ en promedio. Específicamente, explica lo siguiente:

a) **[0,5 puntos]** El papel del *boolean*.

El boolean es para saber si la casilla tiene un elemento y un puntero a otro elemento (o *null*), o si tiene dos punteros (a las casillas delante y detrás en la lista **doblemente ligada** de casillas disponibles).

b) **[4 puntos]** ¿Cómo se implementan las operaciones de diccionario: inserción, eliminación y búsqueda?

Llamemos **lista de colisiones** a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento) y **lista disponible** a la lista de casillas disponibles.

[2 puntos] Inserción: Se aplica la función de hash al nuevo elemento *x*; supongamos que da *k*. Si la casilla *k* está disponible (su boolean vale *true*), la sacamos de la lista (en tiempo O(1) porque está doblemente ligada) y ponemos ahí *x*: cambiamos el boolean a *false* y ponemos el puntero en *null*.

Si la casilla *k* está ocupada (su boolean vale *false*) por un elemento *z*, hay dos posibilidades: *z* hace hash a *k*; o *z* hace hash a un valor distinto de *k* (es decir, es parte de otra lista de colisiones). En el primer caso, hay que agregar *x* en el segundo lugar de la misma lista de *z*, usando una casilla de la lista disponible. En el segundo caso, hay que mover *z* a una casilla disponible y poner *x* en la casilla dejada por *z*, actualizando apropiadamente los punteros involucrados.

[1,25] Eliminación: Sea *k* la casilla a la que hace hash el elemento *x* a eliminar. Si *x* es el único elemento en la lista de colisiones que empieza en la casilla *k*, hay que agregar esta casilla a la lista disponible. Si *x* es el primer elemento, pero no único, en su lista de colisiones, hay que mover el segundo elemento *z* a la casilla *k* y agregar la casilla en que estaba *z* a la lista disponible. Si *x* no es el primer elemento en su lista de colisiones, hay que agregar la casilla que ocupa *x* a la lista disponible, actualizando apropiadamente los punteros.

[0,75 puntos] Búsqueda: Hay que revisar la casilla a la cual el elemento *x* hace hash. Si es una casilla disponible, entonces *x* no está en la tabla. De lo contrario, si *x* no está en la casilla, hay que seguir los punteros.

c) **[1,5 puntos]** ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo $O(1)$ en promedio?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) en promedio.

Las operaciones sobre la lista de casillas disponibles corren en tiempo O(1) gracias a que la lista es doblemente ligada (si no, el único problema ocurre cuando se saca una casilla de la lista).

3. Considera la siguiente implementación de un cola priorizada. Los elementos que están en la cola ocupan posiciones consecutivas en un arreglo *a*, a partir de la casilla *a*[0] —el arreglo se mantiene compacto. Cuando ingresamos un nuevo elemento a la cola, simplemente lo ponemos a continuación del elemento que ocupa la última casilla. Cuando sacamos un elemento de la cola, buscamos secuencialmente a partir de *a*[0] el elemento con mayor prioridad y lo sacamos; para mantener el arreglo compacto, ponemos el elemento que está en la última casilla en el lugar del que sacamos.

a) **[2 puntos]** Compara esta implementación con la que estudiamos en clase, basada en un heap binario. En parti-cular, ¿cuál es la complejidad para las operaciones *xMax( )*, *insertObject( )* e *incrementKey( )* en cada una de estas implementaciones? ¿Bajo qué condiciones conviene usar la implementación descrita en el párrafo anterior?

Supongamos que la cola tiene *n* elementos. En el caso del heap binario, las tres operaciones son O(log *n*), como vimos en clase.

En cambio, en la implementación de más arriba, *insertObject()* e *incrementKey()* son O(1), pero *xMax()* es O(*n*). Recordemos que *incrementKey()* recibe como parámetro el objeto al que se le va a incrementar su prioridad, además del valor del incremento; por lo tanto, no es necesario buscar este objeto en la cola.

Conviene usar esta nueva implementación si la gran mayoría de las operaciones son *insertObject()* e *incrementKey()*, y sólo se hace uno que otro *xMax()*.

b) **[2 puntos]** ¿Cambian tus respuestas si en lugar del arreglo *a* usamos una lista doblemente ligada? ¿Por qué?

No cambian. En particular, *insertObject()* ahora puede insertar al comienzo de la lista doblemente ligada (en lugar de a continuación del último elemento), pero sigue siendo O(1); y *xMax()* no necesita "rellenar" el espacio dejado por el elemento sacado de la cola (las listas doblemente ligadas se mantienen naturalmente compactas), pero sigue siendo O(*n*).

c) **[2 puntos]** Otra operación sobre colas priorizadas es *join( )*, que consiste en unir dos colas priorizadas, compues-tas por elementos del mismo tipo, en una nueva cola priorizada; las prioridades de cada elemento se mantienen, pero la posición relativa de la prioridad de un elemento en la nueva cola puede cambiar. ¿Cuál es la complejidad de *join( )* en cada una de las tres implementaciones mencionadas? Explica.

En el caso de las listas doblemente ligadas, *join()* toma tiempo O(1), ya que basta conectar el último elemento de una de las colas al primero de la otra, cambiando un número fijo de punteros.

En el caso del arreglo descrito en el enunciado, *join()* toma tiempo O(*n*), en que *n* es el número de elementos de la cola más corta, ya que hay que pasar cada uno de estos elementos a la otra cola: hay que ejecutar *n insertObject's*.

En el caso del heap binario, *join()* toma tiempo O(*n*), en que *n* es el número total de elementos an ambas colas: primero, hay que traspasar los elementos de la cola más corta a la otra cola, en tiempo proporcional al número de elementos de la cola más corta; luego, como vimos en clase, armar un heap binario a partir de un arreglo (desordenado) de *n* elementos es O(*n*). Si la cola más corta es mucho más corta, se puede hacer *insertObject()* de cada uno de sus elementos en la otra cola, en tiempo total O(*n'* log *n*), en que *n'* es el número de elementos de la cola más corta.

Tiempo: 105 minutos