APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 2: ESTRUCTURAS DE DATOS

Tipos Abstractos de Datos (TADs): Implementaciones elementales

Hemos planteado tres problemas de programación: control de paréntesis balanceados; su generalización al caso de múltiples delimitadores (paréntesis, corchetes, llaves, etc.) y la implementación de un buffer entre un productor y un consumidor de datos. Del análisis de cada uno de estos problemas surgieron diferentes tipos abstractos que fueron oportunamente especificados: los TADs contador, pila y cola. Luego de su especificación, se asumió que se contaba con una implementación del mismo, lo que permitió resolver los problemas planteados.

Resta entonces encontrar implementaciones adecuadas de los tres TADs mencionados.

Implementación del TAD contador. Para verificar que una expresión tenga sus paréntesis balanceados se observó la necesidad de utilizar un TAD que denominamos contador y cuya especificación es:

TAD contador
constructores
inicial : contador
incrementar : contador → contador
operaciones
es_cero : contador → booleano
decrementar : contador → contador {se aplica sólo a un contador que no sea inicial}
ecuaciones
es_inicial(inicial) = verdadero
es_inicial(incrementar(c)) = falso
decrementar(incrementar(c)) = c

Para escribir el programa que verifica que los paréntesis estén balanceados, asumimos que contábamos con una implementación del mismo:

type counter = ... {- no sabemos aún cómo se implementará -}
proc init (**out** c: counter) {Post: c ∼ inicial}
{Pre: c ∼ C} **proc** inc (**in**/**out** c: counter) {Post: c ∼ incrementar(C)}
{Pre: c ∼ C ∧¬is_init(c)} **proc** dec (**in**/**out** c: counter) {Post: c ∼ decrementar(C)}
fun is_init (c: counter) **ret** b: **bool** {Post: b = (c ∼ inicial)}

Eventualmente el TAD contador debe implementarse de alguna manera. Esta parte del desarrollo es independiente del algoritmo ya dado, puede ser realizado por otro programador y pueden proponerse varias implementaciones. La más natural es la que utiliza un entero o un natural:

type counter = **nat**

El constructor inicial se implementa con el procedimiento init, que tiene como parámetro un contador. El parámetro sólo se utiliza para que el procedimiento devuelva un valor, por ello es un parámetro de salida (**out**) del procedimiento:

proc init (**out** c: counter)
    c:= 0
end proc
{Post: c ∼ inicial}

La poscondición c ∼ inicial indica que al finalizar el procedimiento, c es la representación del valor inicial del contador. Este procedimiento no tiene precondición, puede aplicarse en cualquier momento y al finalizar la poscondición vale.

El constructor incrementar se implementa con el procedimiento inc que tiene como parámetro un contador que se utiliza como de entrada y salida (**in**/**out**):

{Pre: c ∼ C}
proc inc (**in**/**out** c: counter)
    c:= c+1
end proc
{Post: c ∼ incrementar(C)}

La precondición es c ∼ C y la poscondición es c ∼ incrementar(C). Esto indica que si c es la representación del contador C, luego de ejecutar el procedimiento inc será la representación del contador incrementar(C).

La operación decrementar se implementa con el procedimiento dec que tiene como precondición que su parámetro sea un contador diferente del inicial:

{Pre: c ∼ C ∧¬is_init(c)}
proc dec (**in**/**out** c: counter)
    c:= c-1
end proc
{Post: c ∼ decrementar(C)}

La operación es_inicial se implementa con la función is_init que devuelve verdadero sii c es la representación del contador inicial. Los parámetros de las **funciones** son siempre de entrada (**in**) solamente, por eso no es necesario anotarlo.

fun is_init (c: counter) **ret** b: **bool**
    b:= (c = 0)
end fun
{Post: b = (c ∼ inicial)}

Es fácil comprobar que las cuatro operaciones son *constantes* con esta implementación.

Implementación del TAD pila. El siguiente problema planteado fue generalizar el problema de controlar si los paréntesis están balanceados al caso en que hay diferentes tipos de delimitadores. Para resolverlo, descubrimos el tipo abstracto pila, que se especificó de la siguiente manera:

TAD pila[elem]
constructores
vacía : pila
apilar : elem × pila → pila
operaciones
primero : pila → elem {se aplica sólo a una pila no vacía}
desapilar : pila → pila {se aplica sólo a una pila no vacía}
es_vacía : pila → booleano
ecuaciones
primero(apilar(e,s)) = e
desapilar(apilar(e,s)) = s
es_vacía(vacía) = verdadero
es_vacía(apilar(e,s)) = falso

Para resolver el problema planteado, hemos asumido que contábamos con una implementación del mismo:

type stack = ... {- no sabemos aún cómo se implementará -}
proc empty(**out** p:stack) {Post: p ∼ vacía}
{Pre: p ∼ P} **proc** push(**in** e:elem,**in**/**out** p:stack) {Post: p ∼ apilar(e,P)}
{Pre: p ∼ P ∧¬is_empty(p)} **fun** top(p:stack) **ret** e:elem {Post: e ∼ primero(P)}
{Pre: p ∼ P ∧¬is_empty(p)} **proc** pop(**in**/**out** p:stack) {Post: p ∼ desapilar(P)}
fun is_empty(p:stack) **ret** b:**bool** {Post: b = (p ∼ vacía)}

Existen al menos dos maneras naturales de implementar el TAD pila: utilizando listas o utilizando arreglos.

Implementación de pilas con listas. La implementación de pilas utilizando listas es tan directa que no requiere explicaciones:

type stack = [elem]

proc empty(**out** p:stack)
p:= [ ]
end
{Post: p ∼ vacía}

{Pre: p ∼ P }
proc push(**in** e:elem,**in**/**out** p:stack)
p:= (e ▷ p)
end
{Post: p ∼ apilar(e,P)}

{Pre: p ∼ P ∧¬is_empty(p)}
fun top(p:stack) **ret** e:elem
e:= head(p)
end
{Post: e ∼ primero(P)}

{Pre: p ∼ P ∧¬is_empty(p)}
proc pop(**in**/**out** p:stack)
    p:= tail(p)
end
{Post: p ∼ desapilar(P)}

fun is_empty(p:stack) **ret** b:**Bool**
    b:= (p = [ ])
end
{Post: b = (p ∼ vacía)}

Observemos que todas las operaciones de esta implementación son *constantes*: el tiempo de ejecución de ninguna de estas operaciones depende del número de elementos de la pila.[9]

Implementación de pilas con arreglos. Si debe implementarse una pila en un lenguaje que no tenga el tipo concreto lista, una posibilidad digna de consideración es la utilización de un arreglo, a pesar de que esto trae aparejado una restricción en el número de elementos de la pila ya que los arreglos tienen un tamaño establecido en el momento de su creación. Como, a diferencia del arreglo, la pila puede aumentar o disminuir de tamaño, el arreglo será del tamaño máximo que se estime tendrá la pila en el caso del problema a resolver. La definición del tipo stack, además del arreglo contará con un dato adicional que indique el tamaño de la pila, ya que éste varía y no siempre (quizá nunca) coincidirá con el número de celdas del arreglo, N. Para ello utilizamos tuplas:

type stack = **tuple**
        elems: **array**[1..N] **of** elem
        size: **nat**
    end

Con esta representación, el campo elems es el arreglo en el que se alojan los elementos de la pila y el campo size indica cuántos elementos se encuentran alojados actualmente en la pila. Un **invariante** que debe satisfacer esta **representación** es que el campo size debe ser un valor entre 0 y N. Intuitivamente, la pila está formada por los valores del campo elems desde la posición 1 hasta la posición indicada por el campo size. El primero de la pila (cuando no es vacía) se encuentra en la posición indicada por el campo size. La pila p se inicializa como la pila vacía, asignando 0 al campo size de p.

proc empty(**out** p:stack)
    p.size:= 0
end
{Post: p ∼ vacía}

El primer elemento que se agrega a la pila p se aloja en p.elems[1], el segundo en p.elems[2], etc. A medida que se agregan dichos elementos, el campo p.size adopta los valores 1, 2, etc. Es fácil observar que el campo size indicará la última celda del arreglo

[9] Asumiendo algo razonable: que las listas han sido implementadas de manera de que las operaciones que agregan o examinan o quitan el primer elemento de una lista sean constantes.

elems ocupada por la pila. En efecto, los elementos de la pila p se encontrarán en las posiciones p.elems[1], p.elems[2], ..., p.elems[p.size] en el orden en que ingresaron a p.

Implementada la pila de esta manera, para agregarle un elemento es necesario que quede espacio en el arreglo. Eso se indica a continuación con la precondición ¬is_full(p).[10] Al agregar un elemento a la pila, su número de elementos se incrementa en 1, lo que debe consignarse incrementando el campo size. Como el campo size indicaba la última celda del arreglo elems ocupada por la pila, ahora que fue incrementada indica la primera celda libre. Allí es donde se aloja el nuevo elemento e.

```
{Pre: p ∼ P ∧¬is_full(p)}
proc push(in e:elem,in/out p:stack)
      p.size:= p.size+1
      p.elems[p.size]:= e
end
{Post: p ∼ apilar(e,P)}
```

La función top debe devolver el elemento más nuevo de la pila. Como se observó, éste se encontrará en la celda indicada por el campo size.

```
{Pre: p ∼ P ∧¬is_empty(p)}
fun top(p:stack) ret e:elem
    e:= p.elems[p.size]
end
{Post: e ∼ primero(P)}
```

El procedimiento pop debe suprimir el elemento más nuevo de la pila. Esto se realiza decrementando el campo size. Efectivamente, esto no sólo indica que la pila se ha achicado, sino también que la última celda ocupada por la pila se encuentra una posición antes de lo que se encontraba en la pila original. De este modo, el elemento que se pretendía suprimir permanece en el arreglo pero no en la parte del arreglo ocupado por la pila. Es decir, dicho elemento ha sido suprimido de la pila.

```
{Pre: p ∼ P ∧¬is_empty(p)}
proc pop(in/out p:stack)
      p.size:= p.size-1
end
{Post: p ∼ desapilar(P)}
```

Para comprobar si una pila es vacía o no, alcanza con observar su campo size. Se trata de una pila vacía sii ese campo es 0.

```
fun is_empty(p:stack) ret b:bool
    b:= (p.size = 0)
end
{Post: b = (p ∼ vacía)}
```

Por último, esta implementación impone una restricción al tamaño máximo de la pila. Se implementa una función is_full que comprueba si el arreglo está lleno, en cuyo caso

[10] Esta condición no estaba en la especificación del TAD pila, es una condición propia de esta manera de implementar pilas. En efecto, esta implementación modifica (fortalece) la precondición del procedimiento push.

no se pueden agregar elementos a la pila hasta que vuelva a generarse espacio mediante el procedimiento pop o empty:

fun is_full(p:stack) **ret** b:**bool**
    b:= (p.size = N)
end
{Post: b = (p.size = N)}

Cada una de las operaciones presentadas realizan un número fijo de operaciones elementales, independientemente de cuántos elementos tenga la pila. Son operaciones de tiempo *constante*.

Observar que, en realidad, no se requería un natural o un entero para el campo size de la pila. Alcanzaba con un contador. ¿Cómo implementaría el tipo pila con un contador?[11] ¿Qué orden tendrían las operaciones?

Esta es una implementación bastante habitual del tipo pila que, no obstante, impone una restricción artificial en su tamaño, fortalece la precondición del procedimiento push y requiere la definición de la función auxiliar is_full. En este sentido se desvía un poco de la especificación dada, no es una implementación totalmente fiel.

Implementación del TAD cola. Seguidamente se planteó el problema de implementar un buffer entre un productor y un consumidor de datos, lo que dió lugar al TAD cola, que se especificó de la siguiente manera:

TAD cola[elem]
constructores
    vacía : cola
    encolar : cola × elem → cola
operaciones
    primero : cola → elem {se aplica sólo a una cola no vacía}
    decolar : cola → cola {se aplica sólo a una cola no vacía}
    es_vacía : cola → bool
ecuaciones
    primero(encolar(vacía,e)) = e
    primero(encolar(encolar(q,e'),e)) = primero(encolar(q,e'))
    decolar(encolar(vacía,e)) = vacía
    decolar(encolar(encolar(q,e'),e)) = encolar(decolar(encolar(q,e')),e)
    es_vacía(vacía) = verdadero
    es_vacía(encolar(q,e)) = falso

Para resolver el problema planteado, hemos asumido que contábamos con una implementación del TAD cola:

type queue = ... {- no sabemos aún cómo se implementará -}
proc empty(**out** q:queue) {Post: q ∼ vacía}
{Pre: q ∼ Q} **proc** enqueue(**in**/**out** q:queue,**in** e:elem) {Post: q ∼ encolar(Q,e)}
{Pre: q ∼ Q ∧¬is_empty(q)} **fun** first(q:queue) **ret** e:elem {Post: e ∼ primero(Q)}
{Pre: q ∼ Q ∧¬is_empty(q)} **proc** dequeue(**in**/**out** q:queue) {Post: q ∼ decolar(Q)}
fun is_empty(q:queue) **ret** b:**bool** {Post: b = (q ∼ vacía)}

[11] Puede ser necesaria una versión un poco diferente del TAD contador.

Al igual que las pilas, las colas también pueden implementarse utilizando listas o arreglos.

Implementación de colas con listas. Nuevamente la implementación de colas usando listas es directa.

type queue = [elem]

proc empty(**out** q:queue)
    q:= [ ]
end
{Post: q ∼ vacía}

{Pre: q ∼ Q}
proc enqueue(**in**/**out** q:queue; **in** e:elem)
    q:= (q ◁ e)
end
{Post: q ∼ encolar(Q,e)}

{Pre: q ∼ Q ∧¬is_empty(q)}
fun first(q:queue) **ret** e:elem
    e:= head(q)
end
{Post: e ∼ primero(Q)}

{Pre: q ∼ Q ∧¬is_empty(q)}
proc dequeue(**in**/**out** q:queue)
    q:= tail(q)
end
{Post: q ∼ decolar(Q)}

fun is_empty(q:queuee) **ret** b:**Bool**
    b:= (q = [ ])
end
{Post: b = (q ∼ vacía)}

Las operaciones así implementadas son constantes, salvo la de encolar, que puede resultar lineal en algunas implementaciones habituales de listas. De todas maneras, es posible implementar las listas de modo que incluso el procedimiento enqueue sea constante.

Implementación de colas con arreglos. Así como pilas pueden implementarse con arreglos, también las colas se pueden implementar con arreglos. Nuevamente el tamaño de la cola no podrá exceder al del arreglo.

La primera idea que se podría intentar es la de imitar en todo lo posible la implementación de pilas usando arreglos. La diferencia es que en el caso de la cola se elimina el elemento que ingresó primero. Suponiendo que se encuentran k elementos en la cola q en las posiciones q.elems[1], q.elems[2], . . . , q.elems[k], al eliminar un elemento habría que eliminar el que se encuentra en q.elems[1] a diferencia de lo que ocurría en el caso

de la pila, donde se eliminaba el que se encontraba en q.elems[k]. Deben permanecer en la cola los siguientes k-1 elementos: q.elems[2], ..., q.elems[k]. Éstos pueden reubicarse en las posiciones q.elems[1], ..., q.elems[k-1] haciendo las asignaciones

q.elem[1]:= q.elems[2]
q.elems[2]:= q.elems[3]
...
q.elems[k-1]:= q.elems[k]

que se pueden expresar simplemente usando un **for** adecuado, pero demandaría, como se ve, k-1 asignaciones. Esto volvería al algoritmo lineal en vez de constante como han sido todas las operaciones de pilas que se implementaron recientemente.

¿Pueden implementarse las colas sobre arreglos de modo de que las operaciones sean constantes? La respuesta es afirmativa. Para ello debemos pensar en algo más ingenioso que reubicar los elementos luego de eliminar el primero.

Una posibilidad es dejarlos donde están y agregar, además del campo size, un campo fst que señale la celda donde se encuentra el primer elemento de la cola.

Así, tendremos entonces que los elementos de la cola q estarán en las posiciones q.elems[q.fst], q.elems[q.fst+1], ..., q.elems[q.fst+q.size-1]. Una desventaja que se observa fácilmente es que las celdas anteriores a q.elems[q.fst] quedan desaprovechadas y, por ello, el tamaño máximo de la cola limitado aún más.

Para no desaprovechar esas celdas, se piensa a q.elems como un **arreglo circular**. Es decir, si el último elemento de la cola ocupa la celda q.elems[N], el siguiente elemento que ingrese a la cola debería alojarse en la celda q.elems[1] (si ésta ya se desocupó). El siguiente se debería alojar en q.elems[2], etc.

Para lograr ese comportamiento circular, es conveniente indexar el arreglo elems con números de 0 a N-1 en vez de números de 1 a N y utilizar aritmética módulo N para acceder a las celdas del arreglo.

type queue = **tuple**
elems: **array**[0..N-1] **of** elem
size: **nat**
fst: **nat**
end

Un **invariante** que debe satisfacer esta **representación** es que el campo size debe ser un valor entre 0 y N y el campo fst, un valor entre 0 y N-1.

La cola q se inicializa como la cola vacía asignando 0 al campo size de q. También debe asignarse cualquier valor entre 0 y N-1 a su campo fst para que el mismo satisfaga el invariante de la representación.

proc empty(**out** q:queue)
q.size:= 0
q.fst:= 0
end
{Post: q $\sim$ vacía}

Como con esta representación la cola puede llenarse, antes de agregar un elemento debemos asegurarnos que la cola no esté llena. El elemento nuevo se agrega al final, es

decir, en la primera celda libre de q.elems "después" de la última celda ocupada por la cola. Dicha celda se encuentra q.size lugares "después" de la primer celda ocupada por la cola, es decir, de q.elems[q.fst]. Las comillas se deben a que "después" debe interpretarse teniendo en cuenta que se trata de un arreglo circular. La celda que se encuentra q.size lugares "después" de q.elems[q.fst] es la celda q.elems[(q.fst+q.size) mod N]. Luego de alojar e en dicha posición se consigna que la cola tiene un elemento más incrementando el campo size de q.

```
{Pre: q ∼ Q ∧¬is_full(q)}
proc enqueue(in/out q:queue,in e:elem)
      q.elems[(q.fst+q.size) mod N]:= e
      q.size:= q.size+1
end
{Post: q ∼ encolar(Q,e)}
```

La función que devuelve el primer elemento de la cola no tiene más que buscarlo donde se sabe que está: en la posición de q.elems indicada por q.fst.

```
{Pre: q ∼ Q ∧¬is_empty(q)}
fun first(q:queue) ret e:elem
    e:= q.elems[q.fst]
end
{Post: e ∼ primero(Q)}
```

El procedimiento que elimina el primer elemento de la cola q debe decrementar el campo size de q. Además, una vez eliminado, el primer elemento de la cola pasa a ser el que anteriormente era el segundo, que se encuentra un lugar "después" de q.elems[q.fst], es decir en q.elems[(q.fst+1) mod N]. Se debe modificar q.fst para que señale esta posición.

```
{Pre: q ∼ Q ∧¬is_empty(q)}
proc dequeue(in/out q:queue)
      q.size:= q.size-1
      q.fst:= (q.fst+1) mod N
end
{Post: q ∼ decolar(Q)}
```

La función que determina si una cola es vacía no necesita más que comprobar el tamaño de la misma inspeccionando el campo size.

```
fun is_empty(q:queue) ret b:Bool
    b:= (q.size = 0)
end
{Post: b = (q ∼ vacía)}
```

De la misma forma, una cola está llena cuando todas las celdas del arreglo están ocupadas por la cola, es decir, cuando el número de celdas ocupadas por la cola es N.

```
fun is_full(q:queue) ret b:Bool
    b:= (q.size = N)
end
```

Cada una de las operaciones presentadas realizan un número fijo de operaciones elementales, por lo que todas ellas son de tiempo constante.

Nuevamente se puede observar que no se requieren naturales o enteros para los campos size y first de la cola. Para el campo size podría utilizarse una variante del tipo contador. Para el campo fst, en cambio, es necesario un TAD un poco más sofisticado que podríamos llamar índice circular.

Hay algunas variantes para implementar colas usando arreglos circulares. Por ejemplo, se puede reemplazar el campo size por otro entero que indique cuál es la primer celda libre. Sin embargo, esto lleva a desaprovechar una celda del arreglo o a complicar un poco la implementación.

Como en el caso de la pila, la implementación de la cola con arreglos impone una restricción artificial en su tamaño, fortalece la precondición del procedimiento enqueue y requiere la definición de la función auxiliar is_full. En este sentido se desvía un poco de la especificación dada, no es una implementación totalmente fiel.