Algoritmos y Estructuras de Datos II

Tipos Abstractos de Datos: Pilas y Colas

Clase de hoy

1. Tipos abstractos de datos
2. TAD Pila
   - Generalización de paréntesis balanceados
   - Especificación del TAD Pila
   - Resolviendo el problema
3. TAD Cola
   - Problema del Buffer de datos
   - Cola
   - Especificación del TAD Cola
   - Resolviendo el problema

Tipos abstractos de datos (TADs)

Tipos abstractos.

- Separación entre **especificación** e **implementación**.
- Se definen especificando constructores y operaciones.
- Podemos tener varias implementaciones para un mismo tipo.
- En general surgen de analizar un problema a resolver.
- El problema evidencia qué necesitamos representar y qué operaciones tener.

TAD Contador
Paréntesis balanceados

- Problema:
  - Dar un algoritmo que tome una expresión,
  - dada, por ejemplo, por un arreglo de caracteres,
  - y devuelva verdadero si la expresión tiene sus paréntesis correctamente balanceados,
  - y falso en caso contrario.

TAD Contador

Solución conocida

- Recorrer el arreglo de izquierda a derecha y *contar* paréntesis que abren y cierran.
- Necesitamos *iniciar* el contador,
- *incrementarlo* cada vez que se encuentra un paréntesis que abre,
- *decrementarlo* (comprobando previamente que no sea nulo en cuyo caso **no están balanceados**) cada vez que se encuentra un paréntesis que cierra,
- Al finalizar, *comprobar* que la cuenta esté en el estado inicial.

TAD Contador

- El contador se define por lo que sabemos de él: sus cuatro operaciones
  - inicializar
  - incrementar
  - comprobar si su valor es el inicial
  - decrementar si no lo es
- Notamos que las operaciones **inicializar** e **incrementar** son capaces de generar todos los valores posibles del contador, por lo que serán nuestros **constructores**.
- **comprobar** en cambio solamente examina el contador,
- **decrementar** no genera más valores que los obtenibles por **inicializar** e **incrementar**

Generalización de paréntesis balanceados

- Problema:
  - Dar un algoritmo que tome una expresión,
  - dada, por ejemplo, por un arreglo de caracteres,
  - y devuelva verdadero si la expresión tiene sus paréntesis, corchetes, llaves, etc. correctamente balanceados,
  - y falso en caso contrario.

Usando contadores

- ¿Alcanza con un contador?
  - “(1+2)”
  - “{1+(18-[4*2])}”
  - “(1+2}”
- ¿Alcanza con tres (o n) contadores?
  - “(1+2}”
  - “(1+[3-1)+4]”

Conclusión

- No alcanza con saber cuántos delimitadores restan cerrar,
- también hay que saber en qué orden deben cerrarse,
- o lo que es igual
- en qué orden se han abierto,
- mejor dicho,
- ¿cuál fue el último que se abrió? (de los que aún no se han cerrado)
- ¿y antes de ése?
- etc.
- Hace falta una “constancia” de cuáles son los delimitadores que quedan abiertos, y en qué orden deben cerrarse.

Solución posible

- Recorrer el arreglo de izquierda a derecha,
- utilizando dicha “constancia” de delimitadores aún abiertos **inicialmente vacía**,
- **agregarle** obligación de cerrar un paréntesis (resp. corchete, llave) cada vez que se encuentra un paréntesis (resp. corchete, llave) que abre,
- **removerle** obligación de cerrar un paréntesis (resp. corchete, llave) (**comprobando** previamente que la constancia no sea vacía y que la **primera** obligación a cumplir sea justamente la de cerrar el paréntesis (resp. cochete, llave)) cada vez que se encuentra un paréntesis (resp. cochete, llave) que cierra,
- Al finalizar, **comprobar** que la constancia está vacía.

Pila

- Hace falta **algo**, una “constancia,” con lo que se pueda
  - inicializar vacía,
  - agregar una obligación de cerrar delimitador,
  - comprobar si quedan obligaciones,
  - examinar la primera obligación,
  - quitar una obligación.
- La última obligación que se agregó, es la primera que debe cumplirse y quitarse de la constancia.
- Esto se llama **pila**.

TAD Pila

- La pila se define por lo que sabemos: sus cinco operaciones
  - inicializar en vacía
  - apilar una nueva obligación (o elemento)
  - comprobar si está vacía
  - examinar la primera obligación (si no está vacía)
  - quitarla (si no está vacía).
- Nuevamente las operaciones **inicializar** y **agregar** son capaces de generar todas las pilas posibles,
- **comprobar** y **examinar**, en cambio, solamente examinan la pila,
- **quitarla** no genera más valores que los obtenibles por **inicializar** y **agregar**.

Especificación del TAD Pila

spec Stack **of** T **where**

constructors

fun empty_stack() **ret** s : Stack **of** T
{- *crea una pila vacía.* -}

proc push (**in** e : T, **in**/**out** s : Stack **of** T)
{- *agrega el elemento* e *al tope de la pila* s. -}

Especificación del TAD Pila

operations

fun is_empty_stack(s : Stack **of** T) **ret** b : Bool
{- *Devuelve* True *si la pila es vacía* -}

fun top(s : Stack **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra en el tope de* s. -}
{- **PRE:** not is_empty_stack(s) -}

proc pop (**in**/**out** s : Stack **of** T)
{- *Elimina el elemento que se encuentra en el tope de* s. -}
{- **PRE:** not is_empty_stack(s) -}

De aquí en adelante omitiremos escribir las operaciones de destrucción y copia que incluimos en todo TAD, pero las asumimos especificadas.

Algoritmo de control de delimitadores balanceados

```
fun matching_delimiters (a: array[1..n] of char) ret b: bool
    var i: nat
    var p: stack of char
    b:= true
    p := empty_stack()
    i:= 1
    do i ≤ n ∧ b → if left(a[i]) → push(match(a[i]),p)
                      right(a[i]) ∧ (is_empty(p) ∨ top(p) ≠ a[i]) → b:= false
                      right(a[i]) ∧ ¬is_empty(p) ∧ top(p) = a[i] → pop(p)
                      otherwise → skip
                   fi
                   i:= i+1
    od
    b:= b ∧ is_empty(p)
    destroy(p)
end fun
```

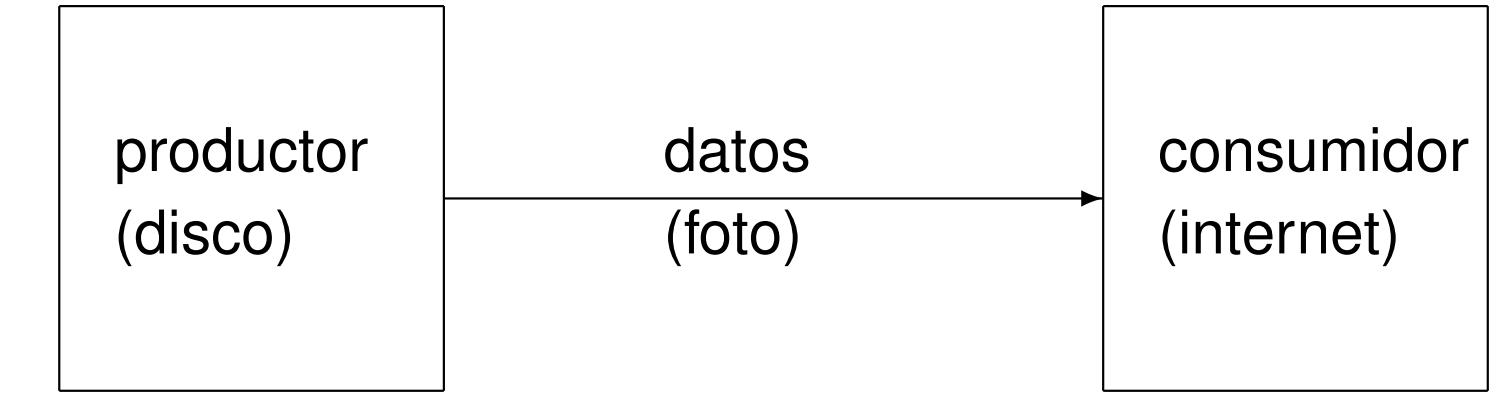
Este algoritmo asume, además de la implementacion de pila,

- una función **match** tal que match('(') = ')', match('[') = ']', match('{') = '}', etc.
- una función **left**, tal que left('('), left('['), left('{'), etc son verdadero, en los restantes casos left devuelve falso.
- una función **right**, tal que right(')'), right(']'), right('}'), etc son verdadero, en los restantes casos, right devuelve falso.

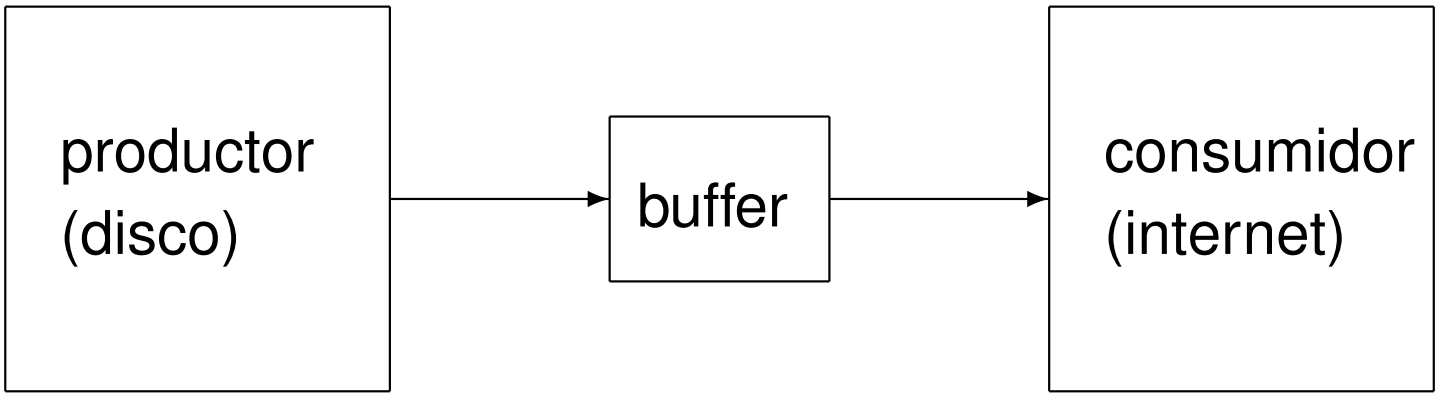
Buffer de datos

- Imaginemos cualquier situación en que ciertos datos deben transferirse desde una unidad a otra,
- por ejemplo, datos (¿una foto?) que se quiere subir a algún sitio de internet desde un disco,
- un agente suministra o produce datos (el disco) y otro que los utiliza o consume (el sitio de internet),
- esta relación se llama **productor-consumidor**
- para amortiguar el impacto por la diferencia de velocidades, se puede introducir un buffer entre ellos,
- un buffer recibe y almacena los datos a medida que se producen y los emite en el mismo orden, a medida que son solicitados.

Gráficamente



se interpone un buffer



Interés

- El programa que realiza la subida de datos puede liberar más rápidamente la lectora del disco.
- El proceso que realizaba la lectura se desocupa antes.
- El productor se ocupa de lo suyo.
- El consumidor se ocupa de lo suyo.
- El buffer se ocupa de la interacción entre ambos.

Uso del buffer

- La interposición del buffer no debe afectar el orden en que los datos llegan al consumidor.
- El propósito es sólo permitir que el productor y el consumidor puedan funcionar cada uno a su velocidad sin necesidad de sincronización.
- El buffer inicialmente está **vacío**.
- A medida que se van **agregando** datos suministrados por el productor, los mismos van siendo alojados en el buffer.
- Siempre que sea necesario enviar un dato al consumidor, habrá que comprobar que el buffer no se encuentre **vacío** en cuyo caso se enviará el **primero** que llegó al buffer y se lo **eliminará** del mismo.

Cola

- Es **algo**, con lo que se pueda
  - inicializar **vacía**,
  - agregar o **encolar** un dato,
  - comprobar si quedan datos en el buffer, es decir, si **es** o no **vacía**
  - examinar el **primer** dato (el más viejo de los que se encuentran en el buffer),
  - quitar o **decolar** un dato.
- El primer dato que se agregó, es el primero que debe enviarse y quitarse de la **cola**.
- Por eso se llama **cola** o también **cola FIFO** (First-In, First-Out).

Tad cola

- La cola se define por lo que sabemos: sus cinco operaciones
  - inicializar en **vacía**
  - **encolar** un nuevo dato (o elemento)
  - comprobar si **está vacía**
  - examinar el **primer** elemento (si no está vacía)
  - **decolarlo** (si no está vacía).
- Las operaciones **vacía** y **encolar** son capaces de generar todas las colas posibles,
- **está vacía** y **primero**, en cambio, solamente examinan la cola,
- **decolarla** no genera más valores que los obtenibles por **vacía** y **apilar**.

Especificación del TAD Cola

spec Queue **of** T **where**

constructors

fun empty_queue() **ret** q : Queue **of** T
{- *crea una cola vacía.* -}

proc enqueue (**in**/**out** q : Queue **of** T, **in** e : T)
{- *agrega el elemento* e *al final de la cola* q. -}

Especificación del TAD Cola

operations

fun is_empty_queue(q : Queue **of** T) **ret** b : Bool
{- *Devuelve* True *si la cola es vacía* -}

fun first(q : Queue **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

proc dequeue (**in**/**out** q : Queue **of** T)
{- *Elimina el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

Algoritmo de transferencia de datos con buffer

```
proc buffer ()
    var d: data
    var q: queue of data
    empty_queue(q)
    do (not finish())
        if there_is_product() → d := get_product()
                                enqueue(q,d)
          there_is_demand() ∧¬ is_empty(q) → d:= first(q)
                                             consume(d)
        fi
    od
end proc
```

- Hemos asumido que hay varias funciones definidas:
- finish(). Devuelve true cuando el servicio de productor-consumidor haya finalizado.
- there_is_product() y there_is_demand() devuelven si hay producción o demanda respectivamente.
- get_product() obtiene un producto desde el productor, y consume(d) le envía al consumidor el producto d.

Utilización

- El TAD cola tiene numerosas aplicaciones.
- Siempre que se quieran atender pedidos, datos, etc. en el orden de llegada.
- Una aplicación interesante es el algoritmo de ordenación llamado Radix Sort.