Tecnología de la programación
Sesión 20

Deberes para la evaluación continua

1. Subalgoritmo para comprobar si dos pilas son iguales de forma iterativa y sin destruir las pilas (ya mandado en la sesión anterior)
2. Construir un subalgoritmo que calcule la suma de todos los elementos almacenados en una cola de números reales y no destruya la cola (transparencia 36).

Objetivos de la sesión

1. TAD Cola: implementación estática y dinámica (transparencias 30-50)

Guion

CORRECCIÓN DEL EJERCICIO DE COMPROBAR SI DOS PILAS SON IGUALES

El ejercicio de comprobar si dos pilas son iguales se pondrá aquí corregido a partir de mañana. Estaba mandado de deberes desde antes de semana santa en la sesión anterior. He abierto una entrega, cuyo plazo acaba el día hoy 22 de abril a las 23:59. Habrá que subirlo con una foto al cuaderno (es en pseudocódigo). Se implementa como USUARIOS.

TAD COLA

Si ya hemos entendido el TAD Pila, todo lo que viene ahora es "más de lo mismo". Vamos a estudiar el TAD Cola. De nuevo vamos a hacer la especificación, implementación y lo usaremos (sin necesidad de saber cómo está implementado, es decir, basándonos solo en la especificación).

Podéis imaginaros una cola como una fila de personas. Todo el mundo es educado y nadie se cuela: es decir, solo podemos añadir de uno en uno (al final del todo) y atender al que está primero (al principio).

Es decir, hablando de elementos de una cola genérica, el primer elemento que llega es el primero que sale. En inglés se suele decir que es una estructura FIFO (first in, first out). Resumiendo:

- Los elementos se insertan de uno en uno (al final de la cola): operación añadir.
- Los elementos se extraen de uno en uno (el primero de la cola): operación eliminar.
- Se puede "observar" al primero de la cola: operación primero.

Al igual que en el caso de pilas, podemos crear colas de cualquier cosa: cola de enteros, cola de booleanos, cola de caracteres o incluso colas de cualquier otro TAD (por ejemplo, una cola de trabajos a imprimir). Para considerar el caso general, diremos que nuestra cola es de elementos de tipo tElemento. De nuevo, habrá que tener cuidado si nuestro tipo tElemento es un tipo primitivo y otro TAD. Recordad que si tElemento es un tipo primitivo (entero, booleano, carácter), podremos asignarlo, devolverlo, compararlo, etc. Pero si tElemento es un TAD, esas operaciones las tendremos que realizar con las operaciones propias del TAD.

Como operaciones del TAD, tendremos constructores (iniciar cola vacía, añadir), accesores (acceder al primero), deconstructores (eliminar operaciones básicas (mostrar, copiar, crear una cola a partir de una secuencia) y auxiliares (saber si una cola está vacía).

Es decir, nuestra especificación sería la siguiente (usando tElemento como tipo primitivo):

ESPECIFICACIÓN TAD COLA

```
TAD Pila(tElemento) // Cola formada por elementos de tipo tElemento
usa
  tElemento
genero
  cola
operaciones
  acción iniciarCola (sal cola c)
  {Pre: }
  {Post: inicia c como una cola vacía}

  acción añadir (e/s cola c, ent tElemento d)
  {Pre: c es una cola que ha sido iniciada previamente}
  {Post: añade en c el elemento d}

  función colaVacia (cola c) dev booleano
  {Pre: c es una cola que ha sido iniciada previamente}
  {Post: dev verdad si c está vacía y falso en caso contrario}

  función primero (cola c) dev tElemento
  {Pre: c es una cola no vacía}
  {Post: devuelve el elemento más antiguo de c y no la modifica}

  acción eliminar (e/s cola c)
  {Pre: c es una cola no vacía}
  {Post: modifica la cola c, eliminando el elemento más antiguo}

  acción crearCola (s/ cola c)
   {Pre: el usuario introducirá por pantalla una secuencia de
elementos con una marca de finalización}
```

Aquí ponemos que usa tElemento, para decir que es sobre un tipo genérico cualquiera

```
   {Post: la cola c contiene la secuencia elementos pedida al usuario
introducida en ese orden}

  acción copiarCola(e/ cola c, s/ cola c2)
  {Pre: c es una cola que ha sido iniciada previamente}
  {Post: c2 es una copia de c, c no se destruye}

  acción mostrarCola(e/ cola c)
  {Pre: c es una cola que ha sido iniciada previamente}
  {Post: muestra los elementos de c en el orden en el que se han
introducido en la cola, c no se destruye}

fin_especificación
```

USO TAD COLA

Una vez conocida la especificación, ya podríamos usarla.

Deberes (para hacer HOY, es decir, fecha límite de entrega el 22 de abril a las 23:59): construye un subalgoritmo que calcule la suma de todos los elementos almacenados en una cola de números reales y no destruya la cola (transparencia 36).

La solución se pondrá en este mismo documento a partir de la siguiente sesión..

Recordemos una cosa muy importante de la sesión anterior: cuando usamos un TAD en general, no sabemos cómo está implementado. Si utilizamos una **representación dinámica (punteros)** para el TAD Cola, **eliminar liberará memoria**. Por lo tanto, **aunque trabajemos con parámetros de entrada, la cola podría quedar destruida.** Por eso hay que tener cuidado con los parámetros a la hora de hacer subalgoritmos con TADs. De hecho, la recomendación con los TADs es usar siempre acciones con parámetros de entrada/salida para evitar problemas. Solo si estamos muy seguros, usaremos funciones.

Vamos a implementar este TAD de dos formas:

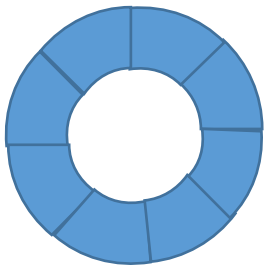
1. Versión estática (basada en vectores)
2. Versión dinámica (basada en punteros)

NOTA: Con la primera versión estaremos limitados a poner un tamaño máximo a la cola.

IMPLEMENTACIÓN ESTÁTICA DEL TAD PILA

La implementación a priori podría parecer muy fácil si la implementamos como un vector donde en la componente 0 está el primer elemento. Sin embargo, el problema llegaría cuando eliminamos el primer elemento, ya que tendríamos que mover el resto de elementos una componente hacia la izquierda. Esto es muy ineficiente.

Para evitarlo, vamos a trabajar como un vector, pero visto circularmente:



El dibujo anterior representa un vector que puede almacenar como máximo 8 elementos.

La implementación estática consistirá en un registro con cuatro campos:

1. Un vector donde vamos a ir almacenando los elementos (hasta un máximo MAXCOLA)
2. Una variable (num) de tipo entero donde se guarda el número de elementos que hay en la cola en ese momento.
3. Una variable de tipo entero llamada "primero", que representa la componente donde se encuentra el primer elemento de la cola (es decir, el índice del vector donde está el primer elemento).
4. Una variable de tipo entero llamada "ultimo", que representa la componente donde se encuentra el primer último de la cola (es decir, el índice del vector donde está el último elemento).

Lo bueno de esta representación es que primero NO siempre vale 0. Hay que pensar que los elementos del vector estarán entre las componentes "primero" y "último". Y ojo, porque "primero" no tiene por qué ser menor que "último". Por ejemplo, si la cola tuviese 6 elementos podríamos tenerlos entre las componentes 6 y 2:

La cola tiene los elementos {95,96,97,98,99} empezando en la posición 6 y terminando en la 2.

A esa situación podríamos haber llega[illegible]os empezado con una cola con 8 elementos {89,90,91,92,93,94,95,96}, ocupa[illegible]as posiciones de 0 a 7 (primero vale 0 y último vale 7), eliminando los 5 primeros elementos (el 89, 90, 91, 92, 93 y 94 que ocupan las posiciones desde 0 hasta 4) y añadiendo después tres elementos (el 97, 98 y 99).

Lo que habrá que hacer en la implementación es aritmética modular, es decir, "modulo MAXCOLA".

Representación constante

```
      MAXCOLA = … // representa tamaño máximo de la cola
tipo
      tipodef tElemento = tvector[MAXCOLA]
      cola = registro
            tvector datos
            entero primero, último, num
      freg
```

Interpretación

Num representa el número de elementos de la cola, los cuales se encuentran almacenados entre las componentes primero y último en el vector datos, estando en la componente primero el elemento más antiguo de la cola y en último el más reciente

Operaciones

```
acción iniciarCola (sal cola c)
{Pre: }
{Post: inicia c como una cola vacía}
principio
      c.num = 0
      c.primero = 0
      c.ultimo = -1
fin

función colaVacia (cola c) dev booleano
{Pre: c es una cola que ha sido iniciada previamente}
{Post: dev verdad si c está vacía y falso en caso contrario}
principio
      devuelve(c.num == 0)
fin
```

Esta función auxiliar es muy importante, es la que nos asegura que al avanzar una posición nos quedemos siempre dentro del rango de elementos del vector (es decir, en un índice válido).

```
función sumaUno (entero n) dev entero
{Pre: n>=0}
{Post: suma uno a la posición n en el sentido
circular de las agujas del reloj, con tamaño
máximo MAXCOLA}
principio
      dev((n+1) MOD MAXCOLA)
fin

acción añadir (e/s cola c, ent tElemento d)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: añade en c el elemento d}
principio
      si c.num < MAXCOLA entonces
            c.num = c.num+1
            c.ultimo = sumaUno(c.ultimo)
            c.datos[c.ultimo] = d
      fsi
fin
```

```
función primero (cola c) dev tElemento
{Pre: c es una cola no vacía}
{Post: devuelve el elemento más antiguo de c y no la modifica}
principio
      devuelve(c.datos[c.primero])
fin

acción eliminar (e/s cola c)
{Pre: c es una cola no vacía}
{Post: modifica la cola c, eliminando el elemento más antiguo}
principio
      c.num = c.num - 1
      c.primero = sumaUno(c.primero)
fin

acción crearCola (s/ cola c)
{Pre: el usuario introducirá por pantalla una secuencia de elementos
con una marca de finalización}
{Post: la cola c contiene la secuencia elementos pedida al usuario
introducida en ese orden}
variables
    tElemento n
principio
    c.num = 0
    c.primero = 0
    c.ultimo = -1
    escribir("Introduce la secuencia")
    leer(n)
    mientras que n != 0 AND c.num < MAXCOLA hacer
            c.num = c.num+1
            c.ultimo = sumaUno(c.ultimo)
            c.datos[c.ultimo] = n
         leer(n)
    fmq
fin

acción copiarCola(e/ cola c, s/ cola c2)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: c2 es una copia de c, c no se destruye}
variables
    entero i
principio
    c2.num = c.num
    c2.primero = c.primero
    c2.ultimo = c.ultimo
    i=c.primero
    mientras que i != sumaUno(c.ultimo)
        c2.datos[i] = c.datos[i]
        i = sumaUno(i)
    fmq
fin


acción mostrarCola(e/ cola c)
```

```
{Pre: c es una cola que ha sido iniciada previamente}
{Post: muestra los elementos de c en el orden en el que se han
introducido en la cola, c no se destruye}
variables
      entero i
principio
    i=c.primero
    mientras que i != sumaUno(c.ultimo)
        escribir(c.datos[i])
        i = sumaUno(i)
    fmq
fin
```

fin implementación

Hay que prestar atención a la hora de implementar estáticamente usando esta representación. Es bastante fácil equivocarse. Por ejemplo, parece natural implementar el copiarCola de la siguiente manera:

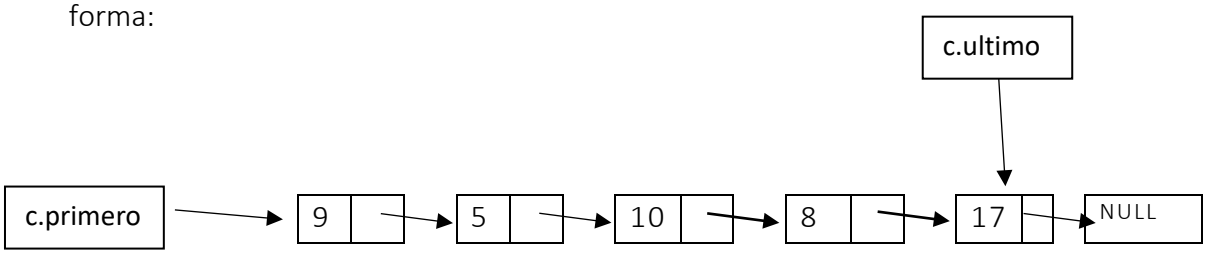
```
principio
    c2.num = c.num
    c2.primero = c.primero
    c2.ultimo = c.ultimo
    para i = c.primero hasta c.ultimo
             c2.datos[i] = c.datos[i]
    fpara
fin
```

Pero eso está mal, porque c.primero no tiene por qué ser un número más pequeño que c.ultimo (como hemos visto en un ejemplo anteriormente). Sí que hay otras alternativas para implementar esa operación que son válidas.

IMPLEMENTACIÓN DINÁMICA DEL TAD PILA

Para no tener la restricción de un número máximo de elementos (la constante MAXCOLA), podemos hacer una implementación basada en punteros. La modelaremos como una especie de "lista enlazada de nodos" pero donde solo podamos añadir al final y eliminar al principio. Para evitar tener que recorrer todos los elementos cada vez que queramos añadir, tendremos un puntero que apuntará al último elemento. Y con otro puntero, apuntaremos al primer elemento, donde la cola empieza. Es decir, que cola lo podemos representar como un registro con dos punteros: uno que apunta al principio y otro que apunta al final.

Es decir, que si queremos modelar una cola que tiene los elementos 9, 5, 10, 8 y 17 (añadidos en ese orden), entonces podremos representarlo con punteros y nodos de esta forma:



Implementación TAD Cola

Representación

```
tipo
      Nodo = registro
            tElemento dato
            puntero a Nodo sig
      freg
      Cola = registro
            puntero a Nodo primero, ultimo
      freg
```

Interpretación

Primero es un puntero a Nodo en el que está el primer elemento de la cola y un puntero a un nodo en el que está el siguiente elemento. El último nodo de la lista apunta a NULL. Ultimo es un puntero que apunta al último elemento de la cola.

Operaciones

```
acción iniciarCola (sal cola c)
{Pre: }
{Post: inicia c como una cola vacía}
principio
      c.primero = NULL
      c.ultimo = NULL
fin

función colaVacia (cola c) dev booleano
{Pre: c es una cola que ha sido iniciada previamente}
{Post: dev verdad si c está vacía y falso en caso contrario}
principio
      devuelve(c.primero == NULL)
fin
acción añadir (e/s cola c, ent tElemento d)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: añade en c el elemento d}
variables
      puntero a Nodo nuevo
principio
      nuevo = reservar(Nodo)
      dest(nuevo).dato = d
      dest(nuevo).sig = NULL
```

```
      si c.primero == NULL entonces
            c.primero = nuevo
      sino
            dest(c.ultimo).sig = nuevo
      fsi
      c.ultimo = nuevo
fin

función primero (cola c) dev tElemento
{Pre: c es una cola no vacía}
{Post: devuelve el elemento más antiguo de c y no la modifica}
principio
      devuelve(dest(c.primero).dato)
fin

acción eliminar (e/s cola c)
{Pre: c es una cola no vacía}
{Post: modifica la cola c, eliminando el elemento más antiguo}
variables
      puntero a Nodo aux
principio
      aux = c.primero
      c.primero = dest(c.primero).sig
      liberar(aux)
      si c.primero == NULL entonces
            c.ultimo = NULL
      fsi
fin

acción crearCola (s/ cola c)
variables
    tElemento n,
    puntero a nodo nuevo
principio
    c.primero = NULL
    c.ultimo = NULL
    escribir("Introduce la secuencia")
    leer(n)
    mientras que n != 0 hacer
        nuevo = reservar(Nodo)
         dest(nuevo).dato = n
         dest(nuevo).sig = NULL
         si c.primero == NULL entonces
            c.primero = nuevo
         sino
             dest(c.ultimo).sig = nuevo
         fsi
         c.ultimo = nuevo
        leer(n)
    fmq
fin

acción copiarCola(e/ cola c, s/ cola c2)
variables
```

```
    puntero a Nodo nuevo, aux
principio
    si c.primero == NULL
        c2.primero = NULL
        c2.ultimo = NULL
    si_no
        nuevo = reservar(Nodo)
        dest(nuevo).dato = dest(c.primero).
        dest(nuevo).sig = NULL
        c2.primero = nuevo
        c2.ultimo = nuevo
        aux = dest(c.primero).sig
        mientras que aux != NULL
            nuevo = reservar(Nodo)
            dest(nuevo).dato = dest(aux).dato
            dest(nuevo).sig = NULL
            dest(c2.ultimo).sig = nuevo
            c2.ultimo = nuevo
            aux = dest(aux).sig
        fmq
    fsi
fin

acción mostrarCola(e/ cola c)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: muestra los elementos de c en el orden en el que se han
introducido en la cola, c no se destruye}
variables
    puntero a Nodo aux
principio
    aux = p
    mientras que aux != NULL
        escribir(dest(aux).dato)
        aux = dest(aux).sig
    fmq
fin

fin_implementación
```

¿En caso de que **copiarCola, mostrarCola y crearCola** no perteneciesen a la especificación, cómo las implementamos?

Habría que implementarlas como usuarios, es decir, sin saber la implementación del TAD Cola (no sabríamos si es con vectores, punteros u otra forma). Lo haríamos usando simplemente las operaciones de la especificación del TAD (añadir, eliminar, primero, colaVacia, etc).

Por ejemplo, para copiar una cola, podríamos hacer lo siguiente:

```
acción copiarCola(e/s cola c1, e/s cola c2)
variables
      cola aux
principio
```

```
      iniciarCola(aux)
      iniciarCola(c2)
      mientras que NOT colaVacia(c1)
          añadir(c2, primero(c1))
          añadir(aux,primero(c1))
          eliminar(c1)
      fmq
      mientras que NOT colaVacia(aux)
         añadir(c1,primero(aux))
         eliminar(aux)
      fmq
fin
```

Hay que darse cuenta de que la cola original se modifica (aunque se reconstruya), y por tanto el parámetro es de entrada/salida.