EJEMPLOS

Ejemplo 1 – Variante de implementación: Dada la siguiente definición del TDA cola (que prioriza velocidad de proceso), modifique las operaciones del *TDA cola* de modo que se adapten a la implementación propuesta.

```
CONSTANTES
          MAXC=10
          MAXI=2
TIPOS
   contenedor=ARREGLO [1..MAXC] de CARACTERES
   indicadores=ARREGLO [1..MAXI] de ENTEROS

   tcola=REGISTRO
           datos:contenedor
           indice:indicadores
         FIN_REGISTRO
```

```
const int MAXC=10;
const int MAXI=2;

typedef char contenedor[MAXC];
typedef int indicadores[MAXI];

typedef struct tcola{
                  contendor datos;
                  indicadores índice;
                };
```

Adaptación de las operaciones del TDA cola a la variante de implementación propuesta

En la variante de implementación propuesta los indicadores de la cola se almacenan en un arreglo de 2 posiciones. En este caso, la primera posición se destina al indicador *frente* y la segunda al indicador *final*. Teniendo en cuenta esto, la operación *iniciar_cola* asigna los valores de inicialización a los elementos del campo *índice*.

```
PROCEDIMIENTO iniciar_cola(E/S q:tcola)
INICIO
   q.indice[1]←MAXC // frente
   q.indice[2]←MAXC // final
FIN
```

```
void iniciar_cola(tcola &q)
{
  q.indice[0]=MAXC-1; // frente
  q.indice[1]=MAXC-1; // final
}
```

La operación *cola_vacia* comprueba si la cola está vacía o no. En este caso, la comprobación se realiza mediante la comparación de los indicadores *frente* y *final* cuyos valores se encuentran almacenados en el arreglo *índice* (campo del registro *tcola*). Si *frente* y *final* son iguales entonces la cola está vacía.

```
FUNCIÓN cola_vacia(E q:tcola):LOGICO
INICIO
   cola_vacia←q.indice[1]=q.indice[2]
FIN
```

```
bool cola_vacia(tcola q)
{
  return q.indice[0]==q.indice[1];
}
```

La operación *cola_llena* comprueba si la cola está llena o no. En este caso, la comprobación se realiza mediante la comparación de los indicadores *frente* y *final* cuyos valores se encuentran almacenados en el arreglo *índice* (campo del registro *tcola*). Si la próxima posición a la que debe apuntar *final* coincide con *frente* entonces la cola está llena.

```
FUNCIÓN cola_llena(E q:tcola):LOGICO
INICIO
   cola_llena←q.indice[1]=sig(q.indice[2])
FIN
```

```
bool cola_llena(tcola q)
{
  return q.indice[0]==sig(q.indice[1]);
}
```

La operación *sig* calcula, a partir del valor actual de un índice, la siguiente posición a la que se hará referencia teniendo en cuenta un almacenamiento circular. Nótese que al alcanzar la última posición del arreglo, la siguiente posición es la primera.

```
FUNCIÓN sig(E n:ENTERO):ENTERO
INICIO
   SI (n = MAX) ENTONCES
      n←1
   SINO
      n←n+1
   FIN_SI
   sig←n
FIN
```

```
int sig(int n)
{ if (n == MAX-1)
    n=0;
  else
    n++;
  return n;
}
```

La operación *agregar_cola* permite añadir un nuevo elemento a la cola siempre que exista espacio. Para adaptar esta operación a la implementación propuesta simplemente se utilizó *q.indice[segunda_posición]* como indicador *final* de la cola en las instrucciones que hacen referencia a él.

```
PROCEDIMIENTO agregar_cola(E/S q:tcola,E nuevo:carácter)
INICIO
   SI (cola_llena(q) = V) ENTONCES
      ESCRIBIR "COLA LLENA"
   SINO
      q.indice[2]←sig(q.indice[2])
      q.datos[q.indice[2]]←nuevo
   FIN_SI
FIN
```

```
void agregar_cola(tcola &q, char nuevo)
{ if (cola_llena(q)==true)
    cout << "COLA LLENA" << endl;
  else
    { q.indice[1]=sig(q.indice[1]);
      q.datos[q.indice[1]]=nuevo;
    }
}
```

La operación *quitar_cola* permite extraer un elemento de la cola siempre que no esté vacía. Para adaptar esta operación a la implementación propuesta simplemente se utilizó *q.indice[primera_posición]* como indicador *frente* de la cola en las instrucciones que hacen referencia a él.

```
FUNCIÓN quitar_cola(E/S q:tcola):CARACTER
VARIABLES
   extraido:CARACTER
INICIO
   SI (cola_vacia(q) = V) ENTONCES
      extraido←' '
   SINO
      q.indice[1]←sig(q.indice[1])
      extraido←q.datos[q.indice[1]]
   FIN_SI
   quitar_cola←extraido
FIN
```

```
char quitar_cola(tcola &q)
{char extraido;
  if (cola_vacia(q)==true)
    extraido=' ';
  else
    { q.indice[0]=sig(q.indice[0]);
      extraido=q.datos[q.indice[0]];
    }
  return extraido;
}
```

La operación *primero* permite consultar el elemento de la cola será el próximo en salir. Para adaptar esta operación a la implementación propuesta simplemente se utilizó *q.indice[primera_posición]* como indicador *frente* de la cola en las instrucciones que hacen referencia a él. Nótese que *frente* no indica el primer elemento de la cola sino uno que ya se extrajo (según se definió en la clase de teoría), por esta razón se utiliza la función *sig* para apuntar al primer elemento válido de la cola.

```
FUNCIÓN primero(E/S q:tcola):CARACTER
VARIABLES
   pri:CARACTER
INICIO
   SI (cola_vacia(q) = V) ENTONCES
      pri←' '
   SINO
      pri←q.datos[sig(q.indice[1])]
   FIN_SI
   primero←pri
FIN
```

```
char primero(tcola &q)
{char pri;
  if (cola_vacia(q)==true)
    pri=' ';
  else
     pri=q.datos[sig(q.indice[0])];
  return pri;
}
```

La operación *último* permite consultar el último elemento agregado a la cola. Para adaptar esta operación a la implementación propuesta simplemente se utilizó *q.indice[segunda_posición]* como indicador *final* de la cola en las

instrucciones que hacen referencia a él. El indicador *final* apunta al último elemento de la cola, por lo que no es necesario hacer ningún ajuste.

```
FUNCIÓN ultimo(E/S q:tcola):CARACTER
VARIABLES
   ulti:CARACTER
INICIO
   SI (cola_vacia(q) = V) ENTONCES
      ulti←' '
   SINO
      ulti←q.datos[q.indice[2]]
   FIN_SI
   ultimo←ulti
FIN
```

```
char ultimo(tcola &q)
{char ulti;
  if (cola_vacia(q)==true)
    ulti=' ';
  else
     ulti=q.datos[q.indice[1]];
  return ulti;
}
```

Ejemplo 2 – Aplicación del TDA cola: Sabiendo que el cociente entre 2 valores enteros puede calcularse mediante restas sucesivas, desarrolle un algoritmo que aplique el TDA cola y sus operaciones básicas para resolver el problema propuesto. Indique además la definición de las estructuras de datos utilizadas en la solución planteada.

Solución propuesta

A fin de resolver el problema planteado (calcular el cociente entre 2 valores enteros) se propone diseñar una función entera que, utilizando el concepto de pila y restas sucesivas, calcule el cociente de una división entera.

```
FUNCIÓN cociente(a:ENTERO, b:ENTERO):ENTERO
VARIABLES
   coc:ENTERO
   q:tcola
INICIO
   iniciar_cola(q)
   MIENTRAS (a >= b) ENTONCES
      agregar_cola(q,1)
      a←a-b
   FIN_MIENTRAS
   coc←0
   MIENTRAS (cola_vacia(q) = F) HACER
     coc<-coc + quitar_cola(q)
   FIN_MIENTRAS
   cociente←coc
FIN
```

```
int cociente(int a, int b)
{int coc;
 tcola q;
  iniciar_cola(q);
  while (a >= b)
  { agregar_cola(q,1);
    a=a-b; }
  coc=0;
  while (cola_vacia(q)==false)
    coc=coc + quitar_cola(q);
  return coc;
}
```

El primer bucle *MIENTRAS* ejecuta la resta sucesiva del dividendo (*a*) respecto del divisor (*b*), almacenando un valor 1 en la cola por cada resta realizada. El segundo bucle extrae los valores almacenados en la cola acumulándolos en la variable *coc*, la que finalmente se asigna a la función cociente.

Respecto a la **definición** del *TDA cola*, a continuación se indican 3 alternativas (existen muchas más) que podrían utilizarse en este problema. Es importante destacar que, más allá de la implementación utilizada, el algoritmo anterior hace referencia de **forma general** al TDA cola y sus operaciones por lo que emplear una u otra resulta indistinto.

Alternativa 1

```
CONSTANTES
     MAX=10
TIPOS
tcontenedor=ARREGLO [1..MAX] de ENTEROS
tcola=REGISTRO
        datos:tcontenedor
        frente,final:ENTERO
      FIN_REGISTRO
```

Alternativa 1

```
const int MAX=10;

typedef int tcontenedor[MAX];
typedef struct tcola{
                     tcontenedor datos;
                     int frente,final;
                    };
```

Alternantiva 2

```
CONSTANTES
     MAX=12
TIPOS
tcola=ARREGLO [1..MAX] de ENTEROS
```

Alternantiva 2

```
const int MAX=12;

typedef int tcola[MAX];
```

Alternativa 3

```
CONSTANTES
          MAXC=10
          MAXI=3
TIPOS
   contenedor=ARREGLO [1..MAXC] de ENTEROS
   indicadores=ARREGLO [1..MAXI] de ENTEROS
   tcola=REGISTRO
          datos:contenedor
          indice:indicadores
         FIN_REGISTRO
```

Alternativa 3

```
const int MAXC=10;
const int MAXI=3;

typedef int contenedor[MAXC];
typedef int indicadores[MAXI];

typedef struct tcola{
                    contendor datos;
                    indicadores índice;
                   };
```

Ejemplo 3 - Implementación del TDA bicola: Utilizando listas simples, con punteros de *inicio* y *final*, implemente el *TDA bicola* (con salida restringida, que almacene caracteres) y las operaciones *iniciar_bicola*, *agregar_bicola*, *quitar_bicola, primero* y *último*.

Definición de la estructura

Para implementar el *TDA bicola* utilizando listas simples es necesario definir los nodos que formarán la *bicola* y los indicadores que permitirán manejarla. En este caso, se optó por hacer corresponder *frente* con el *inicio* de la lista y *final* con el *final* de la lista, aunque podría haberse planteado al revés sin ningún problema (extraer datos del final (*frente* de la bicola) y agregar datos al inicio (*final* de la bicola)).

```
typedef pnodo *tnodo;
typedef struct tnodo {
                      char dato;
                      pnodo sig;
                     };
typedef struct tbicola{
                       pnodo frente; // inicio de la lista
                       pnodo final; // final de la lista
                      };
```

La operación *iniciar_bicola* es simplemente la operación básica *iniciar_lista*. En esta operación los indicadores *frente* y *final* de la bicola se inicializan en NULL, lo que genera una bicola vacía.

```
void iniciar_bicola(tbicola &bq)
{
  bq.frente=NULL;
  bq.final=NULL;
}
```

La operación *agregar_bicola*, que debe trabajar sobre ambos extremos de la estructura, se construye combinando las operaciones básicas *agregar_inicio* y *agregar_final*. El extremo sobre el que se realizará el agregado se determina en función de la variable lógica *primero* (*true* para agregar por el frente, *false* para agregar por el final).

```
void agregar_bicola(tbicola &bq, pnodo nuevo,bool primero)
{
  if (bq.frente==NULL)
    { bq.frente=nuevo;
      bq.final=nuevo; }
  else
    if (primero==true)
     { nuevo->sig=bq.frente;
       bq.frente=nuevo; }
    else
     { bq.final->sig=nuevo;
       bq.final=nuevo;
     }
}
```

Bicola vacía

Agregar inicio

Agregar final

La operación *quitar_bicola*, que sólo opera con el frente de la bicola, es simplemente la operación básica *quitar_inicio*. Esta operación contempla 3 posibles casos: 1) una bicola vacía, 2) una bicola con un único elemento (se modifican los punteros *frente* y *final*) y 3) una bicola con 2 o más elementos (se modifica únicamente el puntero *frente*).

```
pnodo quitar_bicola(tbicola &bq)
{ pnodo extraido;
  if (bq.frente==NULL)
    extraido=NULL;
  else
    if (bq.frente==bq.final)
     { extraido=bq.frente;
      bq.frente=NULL;
      bq.final=NULL;
     }
    else
     { extraido=bq.frente;
       bq.frente=extraido->sig;
       extraido->sig=NULL;
     }
  return extraido;
}
```

Bicola vacía

Único elemento

2 o más elementos

La operación *primero* devuelve la dirección del primer elemento de la bicola (referido por el puntero *frente*) mientras que la operación *ultimo* devuelve la dirección del último elemento de la bicola (referido por el puntero *final*). Nótese que si la bicola está vacía, ambas operaciones retornan nulo.

```
pnodo primero(tbicola bq)
{
  return bq.frente;
}
```

```
pnodo ultimo(tbicola bq)
{
  return bq.final;
}
```

EJERCICIOS

1) De acuerdo a la definición del *TDA cola*, implemente el TDA y sus operaciones fundamentales, considerando:
   - *TDA cola* requiere un contenedor de datos e indicadores del primer y último elemento de la cola.
   - Una operación de inicialización que permita crear (inicializar) una cola vacía.
   - Una operación de inserción que permita agregar un nuevo elemento a la cola (siempre como último elemento).
   - Una operación que determine si el contenedor de datos está completo.
   - Una operación que extraiga elementos de la cola (siempre el elemento que está al principio de la cola).
   - Una operación que determine si la cola no contiene elementos (cola vacía).
   - Una operación que permita consultar el elemento que está al principio (frente) de la cola.
   - Una operación que permita consultar el elemento que está al final de la cola.
   - Una operación que permita consultar la cantidad de elementos almacenados en la cola.

   Suponga que la implementación corresponde a una cola de números caracteres, realizándose en 2 variantes:
   a) implementación *TDA cola* que priorice velocidad de procesamiento
   b) implementación *TDA cola* que priorice espacio de almacenamiento
2) Modifique la definición de la estructura y operaciones del TDA cola del ítem b) de modo que el almacenamiento y recuperación de datos se realice recorriendo el arreglo forma inversa. Considere que la función *siguiente* se renombra como *anterior*.
3) Modifique la implementación del TDA cola del ítem a) considerando que ÚNICAMENTE cuenta con 2 arreglos de 10 posiciones (cada uno) para construir el TDA. Utilice las posiciones iniciales del primer arreglo para almacenar los indicadores de la cola.

4) Suponiendo que la definición del tamaño de arreglos está restringida a 5 elementos, modifique la implementación (definición de la estructura y operaciones) que *prioriza espacio de almacenamiento* de modo que se pueda construir colas de 15 elementos. ¿Cómo se modificaría esta implementación si las últimas posiciones del primer arreglo se utilizan para almacenar los indicadores de la cola? ¿Cuántos datos podrán almacenarse en esta segunda implementación?

5) Dadas las siguientes definiciones del *TDA cola* implemente las operaciones básicas *iniciar_cola*, *agregar_cola*, *anterior* y *cola_vacía* para cada variante

Cola que prioriza espacio de almacenamiento

```
CONSTANTES
          MAX=15
TIPOS
   tcola=ARREGLO [1..MAX] de ENTEROS
```

Considere que los indicadores ocupan las posiciones centrales del arreglo.

Cola que prioriza velocidad de proceso

```
CONSTANTES
          MAX=10
TIPOS
   tcontenedor=ARREGLO [1..MAX] de CARACTERES
   tindicadores=REGISTRO
            ind1:ENTERO
            ind2:ENTERO
         FIN_REGISTRO
   tcola=REGISTRO
            datos:tcontenedor
            indicador:tindicadores
         FIN_REGISTRO
```

Cola que prioriza velocidad de proceso

```
CONSTANTES
          MAX=5
TIPOS
  tcontenedor=ARREGLO [1..MAX] de ENTEROS
  tindicadores=ARREGLO [1..2] de ENTEROS
  tcola=REGISTRO
          datos1:tcontenedor
          datos2:tcontenedor
          indice:tindicadores
        FIN_REGISTRO
```

Cola que prioriza espacio de almacenamiento

```
CONSTANTES
          MAX=6
TIPOS
   tcontenedor=ARREGLO [1..MAX] de ENTEROS
   tcola=REGISTRO
            datos1:tcontenedor
            datos2:tcontenedor
            datos3:tcontenedor
         FIN_REGISTRO
```

Considere que los indicadores ocupan las posiciones finales del segundo arreglo.

6) Utilizando listas enlazadas implemente un *TDA cola* de caracteres y sus operaciones básicas. Para ello, y teniendo en cuenta que sólo podrán almacenarse 20 elementos, considere las siguientes variantes:

   a) listas simples (queda a libre elección el uso de 1 o 2 punteros a la lista)
      - la operación de **inserción** se realiza por el **final** de la lista mientras que la **eliminación** se realizan por el **principio** de la lista. ¿Qué extremo funciona como frente de la cola? ¿Qué extremo funciona como final de la cola?
      - la operación de **inserción** se realiza por el **principio** de la lista mientras que la **eliminación** se realizan por el **final** de la lista. ¿Qué extremo funciona como frente de la cola? ¿Qué extremo funciona como final de la cola?

   b) listas dobles (queda a libre elección el uso de 1 o 2 punteros a la lista)
      - la operación de **inserción** se realiza por el **final** de la lista mientras que la **eliminación** se realizan por el **principio** de la lista. ¿Qué extremo funciona como frente de la cola? ¿Qué extremo funciona como final de la cola?
      - la operación de **inserción** se realiza por el **principio** de la lista mientras que la **eliminación** se realizan por el **final** de la lista. ¿Qué extremo funciona como frente de la cola? ¿Qué extremo funciona como final de la cola?

7) Aplicando el *TDA cola* implementado en el ejercicio 6.a, diseñe un algoritmo que permita convertir una cadena ingresada por el usuario a mayúsculas/minúsculas según elección de éste. Por ejemplo:

```
Cadena ingresada: Soy un programador del 2024
Selección de conversión M (mayúsculas) o m (minúsculas): M
Cadena en mayúsculas: SOY UN PROGRAMADOR DEL 2024
```

8) Desarrolle un algoritmo que aplique el **TDA cola** y sus operaciones básicas para convertir una cadena de caracteres que almacena dígitos de un valor decimal a su correspondiente valor numérico. Para ello, tenga en cuenta que un valor fraccionario puede expresarse mediante la siguiente ecuación:

$$0{,}5327 = 5x10^{-1} + 3x10^{-2} + 2x10^{-3} + 7x10^{-4}$$ (note que los exponentes son negativos)

Para desarrollar la solución utilice el TDA cola implementado en el ejercicio 6.b.

Nota: SUPONGA QUE LOS VALORES A CONVERTIR SIEMPRE TIENEN PARTE ENTERA CERO.

9) Aplicando el TDA cola y sus operaciones básicas desarrolle algoritmos que permitan
   a) calcular un término cualquiera de la serie de Fibonacci
   b) determinar si una palabra o frase es capicúa
   c) invertir los dígitos de un número entero

   Compruebe el funcionamiento de cada programa utilizando una librería de cola estática y otra dinámica.

10) Una variante del ***TDA cola es la bicola de salida restringida*** cuya implementación modifica la operación de inserción de elementos de modo que ésta se realice por el *frente* o el *final* de la estructura según el valor de un parámetro de opción. Teniendo en cuenta esto, realice la definición de datos del *TDA bicola* e implemente sus operaciones básicas considerando las siguientes variantes:
   a) la bicola prioriza el espacio de almacenamiento y se construye únicamente con un arreglo (20 elementos), ubicando los indicadores en las posiciones iniciales del arreglo.
   b) la bicola prioriza la velocidad de proceso y se construye a partir de la definición básica de colas (implementación con arreglos). Suponga que la bicola almacena caracteres.
   c) la bicola se implementa con listas simples (queda a libre elección el uso de 1 o 2 punteros). Suponga que la bicola almacena números enteros y que su capacidad está limitada a 20 elementos.
   d) la bicola se implementa con listas dobles (queda a libre elección el uso de 1 o 2 punteros). Suponga que la bicola almacena valores reales.

   Nota: Genere archivos de librerías *hpp* para las variantes implementadas.

11) Una variante del ***TDA cola es la bicola de entrada restringida*** cuya implementación modifica la operación de eliminación de elementos de modo que ésta se realice por el *frente* o el *final* de la estructura según el valor de un parámetro de opción. Teniendo en cuenta esto, realice la definición de datos del *TDA bicola* e implemente sus operaciones básicas considerando las siguientes variantes:
   a) la bicola prioriza velocidad de proceso y se construye únicamente con dos arreglos (10 elementos), ubicando los indicadores en las posiciones iniciales del segundo arreglo.
   b) la bicola prioriza la velocidad de proceso construyéndose el contenedor de datos con 3 arreglos (5 elementos) y los indicadores por separado. Suponga que la bicola almacena caracteres.
   c) la bicola se implementa con listas simples (queda a libre elección el uso de 1 o 2 punteros). Suponga que la bicola almacena números reales.
   d) la bicola se implementa con listas dobles (queda a libre elección el uso de 1 o 2 punteros). Suponga que la bicola almacena caracteres y que su capacidad está limitada a 15 elementos.

   Nota: Genere archivos de librerías *hpp* para las variantes implementadas.

12) Aplicando el ***TDA bicola*** y sus operaciones básicas, elija el tipo de *bicola* más apropiado y diseñe los algoritmos que permitan resolver los siguientes problemas:
   a) determinar si un número entero es capicúa o no
   b) calcular un término cualquiera de la serie de Fibonacci
   c) calcular el resultado de una expresión posfija

   Compruebe el funcionamiento de cada programa utilizando una librería de bicola estática y otra dinámica.

13) Dado el siguiente módulo *agregar_bicola*, ¿puede reconstruir la definición de datos que le corresponde? ¿de qué tipo de bicola se trata? ¿cómo sería la operación *quitar_bicola*?

```
void agregar_bicola (tbicola &bq, char nuevo)
{ if (bicola_llena(bq)==true)
     cout << "BICOLA LLENA" << endl;
  else
    { bq.ind[1]=siguiente(bq.ind[1]);
      bq.ind[2]++;
      if (bq.ind[1]>=2*MAX)
        bq.datos3[bq.ind[1]-2*MAX]=nuevo;
      else
        if (bq.ind[1]>=MAX)
           bq.datos2[bq.ind[1]-MAX]=nuevo;
        else
           bq.datos1[bq.ind[1]]=nuevo;

    }
}
```