TEMA 4
TIPOS ABSTRACTOS DE DATOS CON ESTRUCTURA LINEAL

---

1. Pilas
2. Colas
3. Colas dobles
4. Listas
5. Secuencias

Bibliografía:

Fundamentals of Data Structures in C++
E. Horowitz, S. Sahni, D. Mehta
Computer Science Press, 1995

Data Abstraction and Problem Solving with C++, Second Edition
Carrano, Helman y Veroff

4.1 Pilas

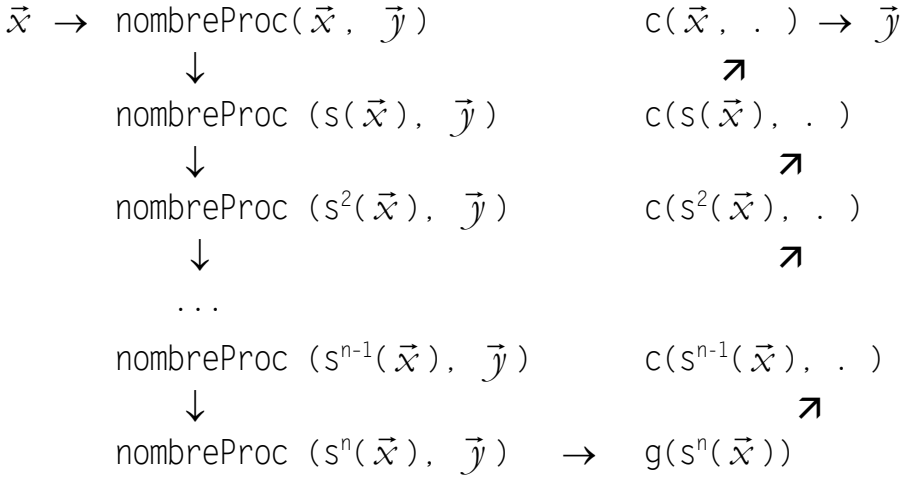
- En el tema anterior ya hemos estudiado las dos implementaciones más habituales de las pilas. En este aparato simplemente veremos una de sus aplicaciones: la transformación a iterativo de algoritmos recursivos lineales.

4.1.1 Eliminación de la recursión lineal no final

- El esquema de la recursión simple

```
void nombreProc ( τ1 x1 , … , τn xn , δ1 & y1 , … , δm & ym ) {
// Precondición
// declaración de constantes
  τ1 x1' ; ... ; τn xn' ;  // x'
  δ1 y1' ; ... ; δm ym' ;  // y'

  if ( d(x) )
     y = g(x);
  else if ( ¬d(x) ) {
     x' = s(x);
     nombreProc(x', y');
     y = c(x, y');
  }
// Postcondición
}
```

La ejecución de *nombreProc*($\vec{x}$, $\vec{y}$) se puede ver como un "bucle descendente" seguido de un "bucle ascendente"



- La transformación a iterativo se basa en ese esquema, se utilizan dos bucles compuestos secuencialmente, de forma que
  - en el primero se van obteniendo las descomposiciones recursivas hasta llegar al caso base, y
  - en el segundo se aplica sucesivamente la función de combinación.

  Nótese que en el segundo bucle, para ir aplicando sucesivamente la función de combinación se debe disponer de los parámetros correspondientes a esa llamada: $s^{n-i}(\vec{x})$.

- Según la forma de obtener los valores de $s^{n-i}(\vec{x})$ en el bucle ascendente distinguimos tres casos, que vienen dados por la forma de la función de descomposición recursiva:
  - Caso especial. La función s de descomposición recursiva, posee una función inversa calculable s^{-1}.

    Los datos $s^{n-i}(\vec{x})$ pueden calcularse sobre la marcha, usando s^{-1}.

  - Caso general. La función s no posee inversa –o, aunque la posea, ésta no es calculable, o su cálculo resulta demasiado costoso–.

    Los datos $s^{n-i}(\vec{x})$ se irán almacenando en una pila en el curso del bucle descendente y se irán recuperando de ésta en el curso del bucle ascendente.

  - Combinación de los casos especial y general.
    - En cada iteración del bucle descendente no es necesario apilar $s^{n-i}(\vec{x})$ sino que es suficiente con apilar un dato más simple $\vec{u}_{n-i}$, de forma que
    - en el bucle ascendente es fácil calcular $s^{n-i}(\vec{x})$ a partir de $\vec{u}_{n-i}$, obtenido de la pila, y $s^{n-i+1}(\vec{x})$ obtenido en la iteración anterior.

Caso especial

- El esquema de la transformación

```
void nombreProcit ( τ1 X1 , … , τn Xn , δ1 & y1 , … , δm & ym ) {
// Precondición
  τ1 X1' ; ... ; τn Xn' ;  // x'

  x' = x;
  while ( ¬d(x') )
    x' = s(x');
  y = g(x');
  while ( x' != x ) {
    x' = s-1(x');
    y = c(x', y);
  }
// Postcondición
}
```

- Transformación de la versión recursiva no final de la función factorial

```
s(n) = n-1     s-1(n) = n+1

int fact ( int n ) {
// Pre : n >= 0 }

  int r, nAux;

  nAux = n;
  while ( nAux != 0 )
    nAux = nAux - 1;
  r = 1;
  while ( nAux != n ) {
    nAux = nAux + 1;
    r = r * nAux;
  }
  return r;
// Post: devuelve n!
}
```

Es obvio que en este caso no se necesita el bucle descendente.

Caso general

- Este es el caso donde nos ayudamos de una pila para almacenar los sucesivos valores del parámetro $\vec{x}$

```
void nombreProcit ( τ1 x1 , … , τn xn , δ1 & y1 , … , δm & ym ) {
// Precondición
  τ1 x1' ; ... ; τn xn' ;  // x⃗'
  TPila< TTupla<τ1, ..., τn> > xs;

  x⃗' = x⃗;
  while ( ¬d(x⃗') ) {
    xs.apila(x⃗');
    x⃗' = s(x⃗');
  }
  y⃗ = g(x⃗');
  while ( ! xs.esVacio() ) {
    x⃗' = xs.cima();
    y⃗ = c(x⃗', y⃗);
    xs.desapila();
  }
// Postcondición
}
```

Para utilizar la implementación genérica de las pilas en este algoritmo, es necesario implementar una clase que represente a las tuplas de *n* elementos

$$TTupla<\tau_1, ..., \tau_n>$$

excepto cuando $n = 1$ en cuyo caso los elementos de la pila serán directamente de tipo τ_1.

- Como ejemplo, veamos cómo se aplica este esquema a la función *bin* que obtiene la representación binaria de un número decimal

```
int bin( int n ) {
// Pre: n >= 0

  int r;

  if ( n < 2 )
    r = n;
  else if ( n >= 2 )
    r = 10 * bin(n / 2) + (n % 2);

  return r;

// Post: devuelve la representación binaria de n
}
```

La transformación a iterativo

```
int binIt( int n ) {
// Pre: n >= 0

  int r, nAux;
  TPilaDinamica<int> ns;

  nAux = n;
  while ( nAux >= 2 ) {
    ns.apila(nAux);
    nAux = nAux / 2;              // n' = s(n')
  }
  r = nAux;
  while ( ! ns.esVacio() ) {
    nAux = ns.cima();
    r = 10 * r + nAux % 2;        // r = c(n',r)
    ns.desapila();
  }

  return r;

// Post: devuelve la representación binaria de n
}
```

Combinación de los casos especial y general

- En este caso sólo es necesario apilar una parte de la información que contienen los parámetros $\vec{x}$ de forma que luego sea posible reconstruir dicho parámetro a partir de la información apilada y s($\vec{x}$).

  Formalmente, hemos de encontrar dos funciones *f* y *h* que verifiquen:

  $\neg d(x) \Rightarrow \vec{u} = f(\vec{x})$ está definido y cumple que $\vec{x} = h(\vec{u}, s(\vec{x}))$

  La versión iterativa queda entonces

```
void nombreProcit ( τ1 X1 , … , τn Xn , δ1 & y1 , … , δm & ym ) {
// Precondición
  τ1 X1' ; ... ; τn Xn' ;  // x'
  σ1 u1 ;  ... ; σp up ;
  TPila< TTupla<σ1, ..., σp> > us;

  x' = x;
  while ( ¬d(x') ) {
    u = f(x');
    us.apila(u);
    x' = s(x');
  }
  y = g(x');
  while ( ! us.esVacio() ) {
    u = us.cima();
    x' = h(u, x');
    y = c(x', y);
    us.desapila();
  }
// Postcondición
}
```

  Para utilizar la implementación genérica de las pilas en este algoritmo, es necesario implementar una clase que represente a las tuplas de *p* elementos

  $TTupla<\sigma_1, ..., \sigma_p>$

  excepto cuando $p = 1$ en cuyo caso los elementos de la pila serán directamente de tipo σ_1.

- En la función *bin* la descomposición recursiva es:

```
s(n) = n / 2;
```

Aquí no se puede aplicar la técnica del caso especial porque no existe la inversa de esta función: para obtener *n* a partir de *n /2* tenemos que saber si *n* es par o impar. Sin embargo, en realidad no hace falta apilar *n*, basta con apilar su paridad, pues a partir de *n / 2* y la paridad de *n* podemos obtener *n*.

```
f(n) = par(n) = u

                ⎧ 2 * s(n)          si     u
h(u, s(n)) =    ⎨
                ⎩ 2 * s(n) + 1      si NOT u
```

Con lo que la versión iterativa aplicando este nuevo esquema quedará:

```
int binIt2( int n ) {
// Pre: n >= 0

  int r, nAux;
  bool u;
  TPilaDinamica<bool> us;

  nAux = n;
  while ( nAux >= 2 ) {
    u = (nAux % 2) == 0;          // u = f(n')
    us.apila(u);
    nAux = nAux / 2;              // n' = s(n')
  }
  r = nAux;
  while ( ! us.esVacio() ) {
    u = us.cima();
    if ( u )                      ⎫
      nAux = 2 * nAux;            ⎪
    else                          ⎬ // n' = h(u, n')
      nAux = 2 * nAux + 1;        ⎭
    r = 10 * r + nAux % 2;        // r = c(n',r)
    us.desapila();
  }
  return r;
// Post: devuelve la representación binaria de n
}
```

4.2 Colas

- Este TAD representa una colección de datos del mismo tipo donde las operaciones de acceso hacen que su comportamiento se asemeje al de una cola de personas esperando a ser atendidas: los elementos se añaden por el final y se eliminan por el principio, o dicho de otra forma, el primero en llegar es el primero en salir –en ser atendido–: FIFO –*first in first out*–.

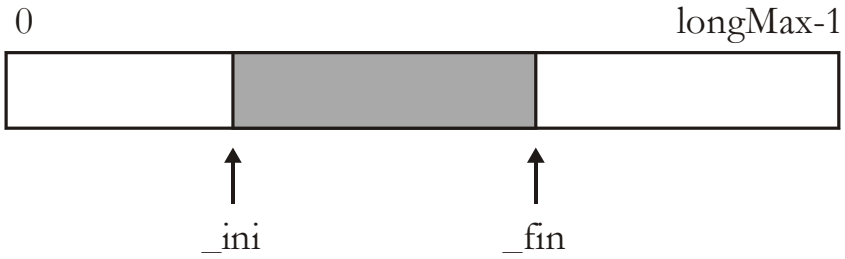
Especificación

```
tad COLA [E :: ANY]
  usa
    BOOL
  tipo
    Cola[Elem]
  operaciones
    Nuevo: → Cola[Elem]                                  /* gen */
    PonDetras: (Elem, Cola[Elem]) → Cola[Elem]           /* gen */
    quitaPrim: Cola[Elem] - → Cola[Elem]                 /* mod */
    primero: Cola[Elem] - → Elem                         /* obs */
    esVacio: Cola[Elem] → Bool                           /* obs */
  ecuaciones
    ∀ x : Elem : ∀ xs : Cola[Elem] :
    def quitaPrim(PonDetras(x, xs))
    quitaPrim(PonDetras(x, xs)) = Nuevo                  si esVacio(xs)
    quitaPrim(PonDetras(x, xs)) =
                PonDetras(x, quitaPrim(xs))              si NOT esVacio(xs)
    def primero(PonDetras(x, xs))
    primero(PonDetras(x, xs))   = x                      si esVacio(xs)
    primero(PonDetras(x, xs))   = primero(xs)            si NOT esVacio(xs)
    esVacio(Nuevo)              = Cierto
    esVacio(PonDetras(x, xs))   = Falso
  errores
    avanza(Nuevo)
    primero(Nuevo)
ftad
```

Implementación estática basada en un vector

Tipo representante

- La idea de la representación es almacenar los datos en posiciones consecutivas de un array, indicando con dos índices el principio, *ini*, y el final, *fin*, de la zona utilizada. Inicialmente *ini* y *fin* están al principio del vector, pero tras un cierto número de invocaciones a *Añade* y *avanza* llegaremos a un estado como este:



- La estructura de datos:

```
template <class TElem>
class TColaEstatica {
  public:
  // Tamaño máximo
     static const int longMax = 100;
  ...
  private:
  // Variables privadas
     int _ini, _fin;
     TElem _espacio[longMax];
  ...
};
```

- Invariante de la representación

  Dada *xs : TColaEstatica*

$$
\begin{aligned}
& R(xs) \\
\Leftrightarrow_{def}\ & 0 \le xs.\_ini \le longMax \wedge -1 \le xs.\_fin \le longMax-1 \wedge \\
& xs.\_ini \le xs.\_fin+1 \wedge \\
& \forall\ i : xs.\_ini \le i \le xs.\_fin : R(xs.\_espacio[i])
\end{aligned}
$$

  Nótese que una cola vacía se caracteriza por $xs.\_ini = xs.\_fin+1$.

Interfaz de la implementación

```
template <class TElem>
class TColaEstatica {
  public:

  // tamaño máximo
     static const int longMax = 5;

  // Constructoras y operador de asignación
     TColaEstatica( );
     TColaEstatica( const TColaEstatica<TElem>& );
     TColaEstatica<TElem>& operator=( const TColaEstatica<TElem>& );

  // Operaciones de las colas
     void ponDetras(const TElem&) throw (EDesbordamiento);
     // Pre: ! esLleno( )
     // Post: Se añade 'elem' al final de la cola
     // Lanza la excepción EDesbordamiento si la cola está llena
     const TElem& primero( ) const throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Devuelve el primer elemento de la cola
     // Lanza la excepción EAccesoIndebido si la cola está vacía
     void quitaPrim( ) throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Elimina el primer elemento de la cola
     // Lanza la excepción EAccesoIndebido si la cola está vacía
     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la cola está o no vacía

  // Limitaciones de la implementación
     bool esLleno( ) const;
     // Pre: true
     // Post: determina si es posible añadir más elementos a la cola

  private:
  // Variables privadas
     int _ini, _fin;
     TElem _espacio[longMax];
  // Operaciones privadas
     void copia(const TColaEstatica<TElem>&);
};
```

Implementación de las operaciones

```
template <class TElem>
TColaEstatica<TElem>::TColaEstatica( ) :
  _ini(0), _fin(-1) { };
// O( longMax * TElem::TElem() ), O(1) sobre tipos predefinidos

template <class TElem>
void TColaEstatica<TElem>::ponDetras(const TElem& elem)
  throw (EDesbordamiento){
  if ( esLleno() )
    throw EDesbordamiento( "Cola llena" );
  _fin++;
  _espacio[_fin] = elem;
};
// O( TElem::operator=(TElem&) ), O(1) sobre tipos predefinidos

template <class TElem>
const TElem& TColaEstatica<TElem>::primero( ) const
  throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido( "Cola vacía");
  return _espacio[_ini];
};
// O(1)

template <class TElem>
void TColaEstatica<TElem>::quitaPrim( ) throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido( "Cola vacía" );
  _ini++;
};
// O(1)

template <class TElem>
bool TColaEstatica<TElem>::esVacio( ) const {
  return _ini == _fin+1;
};
// O(1)

template <class TElem>
bool TColaEstatica<TElem>::esLleno( ) const {
  return _fin == longMax-1;
};
// O(1)
```

- Aunque todos los datos de *TColaEstatica* son estáticos, redefinimos el constructor de copia y el operador de asignación para copiar/asignar tan solo los elementos almacenados y no el array entero.

```
template <class TElem>
void TColaEstatica<TElem>::copia(const TColaEstatica<TElem>& cola) {
  int numElem = cola._fin - cola._ini + 1;
  for ( int i = 0; i < numElem ; i++ )
    _espacio[i] = cola._espacio[cola._ini + i];
  _ini = 0;
  _fin = -1 + numElem;
};
// O( n' * TElem::operator=(TElem&) )
// O(n') sobre tipos predefinidos
// siendo n' el número de elementos de 'cola'

template <class TElem>
TColaEstatica<TElem>::TColaEstatica( const TColaEstatica<TElem>& cola ) {
  copia(cola);
};
// O( longMax · TElem::TElem() + n' * TElem::operator=(TElem&) )
// O(n') sobre tipos predefinidos
// siendo n' el número de elementos de 'cola'

template <class TElem>
TColaEstatica<TElem>&
TColaEstatica<TElem>::operator=( const TColaEstatica<TElem>& cola ) {
  if( this != &cola ) {
    copia(cola);
  }
  return *this;
};
// O( n' * TElem::operator=(TElem&) )
// O(n') sobre tipos predefinidos
// siendo n' el número de elementos de 'cola'
```

El coste de la destructora, que no implementamos, será

```
O( longMax * TElem::~TElem() )
```

o

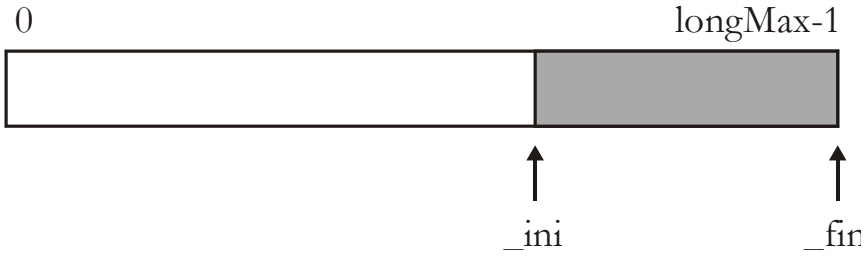
```
O(1)
```

sobre tipos predefinidos.

Desventaja de esta implementación

- El espacio ocupado por la representación de la cola se va desplazando hacia la derecha al ejecutar *ponDetras* y *quitaPrim*.

  Cuando *_fin* alcanzar el valor *longMax*-1 no se pueden añadir nuevos elementos, aunque quede sitio en *_espacio[0.._ini–1]*.

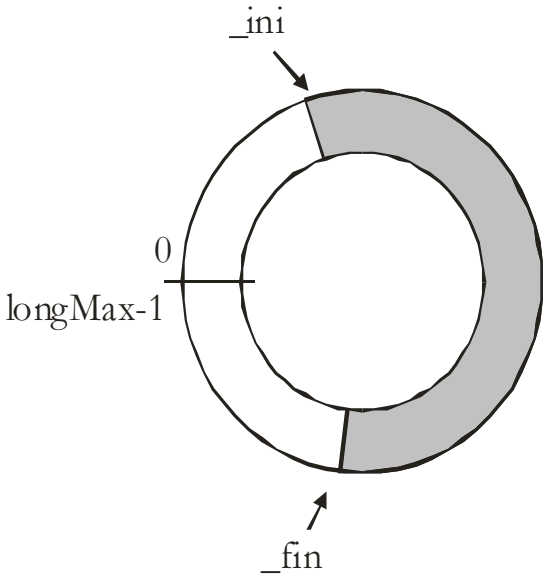


  Una solución sería que cuando llegásemos a esta situación se desplazasen todos los elementos de la pila hacia la izquierda haciendo *_ini=0*. Sin embargo esto penalizaría en esos casos la eficiencia de la operación *ponDetras* –*O*(*n*)–

Implementación basada en un vector circular

- Esta implementación está pensada para resolver el problema de la anterior; cuando *_fin* alcanza el valor *longMax*-1 y queda espacio libre entre 0 e *_ini–1* se utiliza ese espacio para seguir añadiendo elementos.

  La idea es considerar que los extremos del array está unidos



  Hay que tener cuidado porque ahora puede suceder que *_ini* > *_fin*, y en particular *_ini*=*_fin+1* puede implicar que la cola esté llena o vacía.

- Además nos aprovecharemos de la posibilidad de ubicar arrays dinámicamente para aumentar el tamaño del array cuando éste se llene.

  Para ello utilizamos el puntero *_espacio* que apuntará a un array ubicado dinámicamente y la variable *_capacidad* que almacena en cada instante el tamaño del array ubicado.

```
int _capacidad;     // equivalente a longMax
TElem *_espacio;
```

  Cuando estando todas las posiciones del array ocupadas se intenta insertar un nuevo elemento

  – se ubica un array del doble de capacidad,
  – se copia el array antiguo sobre el recién ubicado, y
  – se anula el array antiguo.

Tipo representante

- La estructura de datos

```
template <class TElem>
class TColaEstatica {
  ...

  private:
  // Variables privadas
     int _ini, _fin;
     int _longitud;      // número de elementos de la cola
     int _capacidad;
     TElem *_espacio;    // _espacio[_capacidad]
  ...
};
```

El campo *_longitud* permite distinguir en la situación *_ini=_fin+1* si tenemos una cola vacía o llena.

- Invariante de la representación
  Dado *xs : TColaEstatica*

$$
\begin{aligned}
& R(xs) \\
\Leftrightarrow_{def}\ & (xs.\_ini = 0 \wedge xs.\_longitud = 0 \wedge xs.\_fin = xs.\_capacidad-1) \vee \\
& (0 \le xs.\_longitud \le xs.\_capacidad \wedge \\
& \ 0 \le xs.\_ini \le xs.capacidad-1 \wedge \\
& \ xs.\_fin = (xs.\_ini + xs.\_longitud-1) \bmod xs.\_capacidad \wedge \\
& \ \forall\ i : 0 \le i \le xs.\_longitud-1 : \\
& \qquad R(xs.\_espacio[xs.\_ini+i \bmod xs.\_capacidad]) \qquad)
\end{aligned}
$$

Interfaz de la implementación

```
template <class TElem>
class TColaEstatica {
  public:

     // Constructoras, destructora y operador de asignación
     TColaEstatica( int = 5 );
     // Recibe el tamaño inicial de la cola

     TColaEstatica( const TColaEstatica<TElem>& );
     ~TColaEstatica( );
     TColaEstatica<TElem>& operator=( const TColaEstatica<TElem>& );

  // Operaciones de las colas
     void ponDetras(const TElem&);
     // Pre: true
     // Post: Se añade 'elem' al final de la cola

     const TElem& primero( ) const throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Devuelve el primer elemento de la cola

     void quitaPrim( ) throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Elimina el primer elemento de la cola

     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la cola está o no vacía

  private:
  // Variables privadas
     int _longitud;      // número de elementos de la cola
     int _capacidad;
     TElem *_espacio;    // _espacio[_capacidad]
     int _ini, _fin;

  // Operaciones privadas
     void copia(const TColaEstatica<TElem>&);
     void libera();
};
```

Implementación de las operaciones

```
// Constructoras, destructora y operador de asignación

template <class TElem>
TColaEstatica<TElem>::TColaEstatica( int capacidad ) :
  _capacidad(capacidad), _ini(0), _fin(-1), _longitud(0),
  _espacio(new TElem[_capacidad]) {
};
// O( _capacidad * TElem::TElem() ), O(1) sobre tipos predefinidos


// Operaciones auxiliares de copia y anulación

template <class TElem>
void TColaEstatica<TElem>::libera() {
  delete [] _espacio;
};
// O( _capacidad * TElem::~TElem() ), O(1) sobre tipos predefinidos


template <class TElem>
void TColaEstatica<TElem>::copia(const TColaEstatica<TElem>& cola) {
  _capacidad = cola._capacidad;
  _longitud = cola._longitud;
  _espacio = new TElem[ _capacidad ];
  for ( int i = 0; i < _longitud; i++ )
    _espacio[i] = cola._espacio[(cola._ini+i) % _capacidad];
  _ini = 0;
  _fin = _longitud-1;
};
// O( cola._capacidad * TElem::TElem() + n' * TElem::operator=(TElem&) ),
// O(n') sobre tipos predefinidos
// donde n' es el número de elementos de 'cola'
```

```
template <class TElem>
TColaEstatica<TElem>::TColaEstatica( const TColaEstatica<TElem>& cola ) {
  copia(cola);
};
// O( cola._capacidad * TElem::TElem() + n' * TElem::operator=(TElem&) ),
// O(n') sobre tipos predefinidos
// donde n' es el número de elementos de 'cola'


template <class TElem>
TColaEstatica<TElem>::~TColaEstatica( ) {
  libera();
};
// O( _capacidad * TElem::~TElem() ), O(1) sobre tipos predefinidos

template <class TElem>
TColaEstatica<TElem>&
TColaEstatica<TElem>::operator=( const TColaEstatica<TElem>& cola ) {
  if( this != &cola ) {
    libera();
    copia(cola);
  }
  return *this;
};
// O( _capacidad * TElem::~TElem() +
//    cola._capacidad * TElem::TElem() + n' * TElem::operator=(TElem&) ),
// O(n') sobre tipos predefinidos
// donde n' es el número de elementos de 'cola'
```

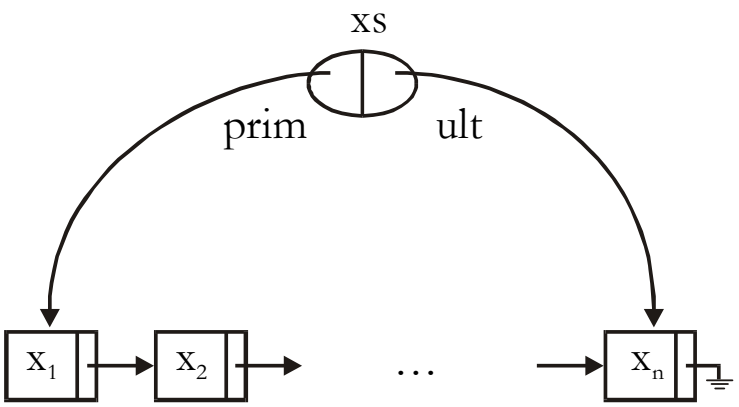
En el operador de asignación se podría distinguir el caso en el que las dos colas tienen la misma *_capacidad*, donde no es necesario liberar y copiar sino que basta con realizar asignaciones.

```
template <class TElem>
void TColaEstatica<TElem>::ponDetras(const TElem& elem) {
  if ( _longitud == _capacidad ){
    _capacidad *= 2;
    TElem* nuevo = new TElem[_capacidad];
    for( int i = 0; i < _longitud; i++ )
      nuevo[i] = _espacio[(_ini+i) % (_capacidad/2)];
    delete [] _espacio; _espacio = nuevo;
    _ini = 0; _fin = _longitud-1;
  }
  _fin = (_fin + 1) % _capacidad ;
  _longitud++;
  _espacio[_fin] = elem;
};
// Si _longitud < _capacidad
//    O( TElem::operator=(TElem&), O(1) sobre tipos predefinidos
// Si _longitud == _capacidad
//    O( 2 * _capacidad * TElem::TElem() +
//       n * TElem::operator=(TElem&) + _capacidad * TElem::~TElem() ),
//    O(n) sobre tipos predefinidos

template <class TElem>
const TElem& TColaEstatica<TElem>::primero( ) const
  throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido( "Cola vacía");
  return _espacio[_ini];
};
// O(1)

template <class TElem>
void TColaEstatica<TElem>::quitaPrim( ) throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido( "Cola vacía" );
  _ini = (_ini + 1) % _capacidad;
  _longitud--;
};
// O(1)

template <class TElem>
bool TColaEstatica<TElem>::esVacio( ) const {
  return _longitud == 0;
};
// O(1)
```

Implementación dinámica

Tipo representante

- La idea es tener acceso directo al principio y al final de la cola para así poder realizar las operaciones de manera eficiente.



- La estructura de datos

```
template <class TElem>
class TNodoCola {
  private:
    TElem _elem;
    TNodoCola<TElem>* _sig;

  ...

};

template <class TElem>
class TColaDinamica {

  ...

  private:
     TNodoCola<TElem> *_prim, *_ult;
  ...

};
```

- Invariante de la representación

  Dada *xs : TCola*

  $R(xs) \Leftrightarrow_{def} R_{CV}(xs) \vee R_{CNV}(xs)$

  donde R_{CV} se refieren a las condiciones de una cola vacía y R_{CNV} a las de una que no está vacía

$$
\begin{array}{l}
\quad R_{CV}(xs) \\
\Leftrightarrow_{def} \\
\quad xs.\_prim = 0 \wedge xs.\_ult = 0
\end{array}
$$

$$
\begin{array}{l}
\quad R_{CNV}(xs) \\
\Leftrightarrow_{def} \\
\quad xs.\_prim \neq 0 \wedge xs.\_ult \neq 0 \wedge \\
\quad buenaCola(xs.\_prim,\ xs.\_ult)
\end{array}
$$

  y el predicado *buenaCola* por su parte

$$
\begin{array}{l}
\quad buenaCola(p,q) \\
\Leftrightarrow_{def} \\
\quad (\ p = q \wedge ubicado(p) \wedge R(p\text{->}\_elem) \wedge p\text{->}\_sig = 0\ ) \vee \\
\quad (\ p \neq q \wedge ubicado(p) \wedge R(p\text{->}\_elem) \wedge \\
\quad\ \ p \notin cadena(p\text{->}\_sig) \wedge buenaCola(p\text{->}\_sig,\ q)\ )
\end{array}
$$

$$
\begin{array}{ll}
cadena(p) =_{def} \varnothing & \textbf{si } p = 0 \\
cadena(p) =_{def} \{p\} \cup cadena(p\text{->}\_sig) & \textbf{si } p \neq 0
\end{array}
$$

  donde se indica que:

  – desde *p* es posible alcanzar *q*
  – todos los nodos entre *p* y *q* están ubicados
  – todos los nodos contienen representantes válidos del tipo de los elementos
  – no hay ciclos en la lista enlazada

Interfaz de la implementación

- Es igual que en la implementación estática

```
template <class TElem>
class TColaDinamica {
  public:

  // Constructoras, destructora y operador de asignación
     TColaDinamica( );
     TColaDinamica( const TColaDinamica<TElem>& );
     ~TColaDinamica( );
     TColaDinamica<TElem>& operator=( const TColaDinamica<TElem>& );

  // Operaciones de las colas
     void ponDetras(const TElem&);
     // Pre: true
     // Post: Se añade 'elem' al final de la cola

     const TElem& primero( ) const throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Devuelve el primer elemento de la cola
     // Lanza la excepción EAccesoIndebido si la cola está vacía

     void quitaPrim( ) throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Elimina el primer elemento de la cima
     // Lanza la excepción EAccesoIndebido si la cola está vacía

     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la pila está o no vacía

  private:
  // Variables privadas
     TNodoCola<TElem> *_prim, *_ult;

  // Operaciones privadas
     void libera();
     void copia(const TColaDinamica<TElem>& pila);
};
```

Implementación de las operaciones

- La clase de los nodos

```
template <class TElem>
class TNodoCola {
  private:
    TElem _elem;
    TNodoCola<TElem>* _sig;
    TNodoCola( const TElem&, TNodoCola<TElem>* = 0 );
  public:
    const TElem& elem() const;
    TNodoCola<TElem> * sig() const;
    friend TColaDinamica<TElem>;
};


template <class TElem>
TNodoCola<TElem>::TNodoCola( const TElem& elem, TNodoCola<TElem>* sig ) :
  _elem(elem), _sig(sig) {
};
// O(TElem::TElem(TElem&)), O(1) sobre tipos predefinidos

template <class TElem>
const TElem& TNodoCola<TElem>::elem() const {
  return _elem;
}
// O(1)

template <class TElem>
TNodoCola<TElem>* TNodoCola<TElem>::sig() const {
  return _sig;
}
// O(1)
```

```
// Constructoras, destructora y operador de asignación

template <class TElem>
TColaDinamica<TElem>::TColaDinamica( ) :
  _prim(0), _ult(0) {
};
// O(1)


template <class TElem>
void TColaDinamica<TElem>::libera() {
  while (_prim != 0) {
    TNodoCola<TElem>* tmp = _prim;
    _prim = _prim->sig();
    delete tmp;
  }
};
// O(n * TElem::~TElem()), O(n) si sobre tipos predefinidos


template <class TElem>
void TColaDinamica<TElem>::copia(const TColaDinamica<TElem>& cola) {
  if ( cola.esVacio() )
    _prim = _ult = 0;
  else {
    TNodoCola<TElem> *antCopia, *actCopia, *act;
    act = cola._prim;
    _prim = new TNodoCola<TElem>( act->elem(), 0 );
    actCopia = _prim;
    while ( act->sig() != 0 ) {
      act = act->sig();
      antCopia = actCopia;
      actCopia = new TNodoCola<TElem>( act->elem(), 0 );
      antCopia->_sig = actCopia;
    }
    _ult = actCopia;
  }
};
// O(n' * TElem::TElem(TElem&)), O(n') sobre tipos predefinidos
// donde n' es el número de elementos de 'cola'
```

```
template <class TElem>
TColaDinamica<TElem>::TColaDinamica( const TColaDinamica<TElem>& cola ) {
  copia(cola);
};
// O(n' * TElem::TElem(TElem&)), O(n') sobre tipos predefinidos
// donde n' es el número de elementos de 'cola'

template <class TElem>
TColaDinamica<TElem>::~TColaDinamica( ) {
  libera();
};
// O(n * TElem::~TElem()), O(n) sobre tipos predefinidos

template <class TElem>
TColaDinamica<TElem>&
TColaDinamica<TElem>::operator=( const TColaDinamica<TElem>& cola ) {
  if( this != &cola ) {
    libera();
    copia(cola);
  }
  return *this;
};
// O(n * TElem::~TElem() + n' * TElem::TElem(TElem&) ),
// O(n+n') sobre tipos predefinidos
// donde n' es el número de elementos de 'cola'
```

```
// operaciones de las colas

template <class TElem>
void TColaDinamica<TElem>::ponDetras(const TElem& elem) {
  TNodoCola<TElem>* p = new TNodoCola<TElem>(elem);
  if( _ult != 0 )
    _ult->_sig = p;
  else
    _prim = p;
  _ult = p;
};
// O(TElem::TElem(TElem&)), O(1) sobre tipos predefinidos

template <class TElem>
const TElem& TColaDinamica<TElem>::primero( ) const
  throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido("Error: no existe el primero de la cola vacía");
  else
    return _prim->elem();
};
// O(1)

template <class TElem>
void TColaDinamica<TElem>::quitaPrim( ) throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido("Error: no existe el primero de la cola vacía");
  else {
    TNodoCola<TElem>* tmp = _prim;
    _prim = _prim->sig();
    if( _prim == 0 )
      _ult = 0;
    delete tmp;
  }
};
// O(TElem::~TElem()), O(1) sobre tipos predefinidos

template <class TElem>
bool TColaDinamica<TElem>::esVacio( ) const {
  return _prim == 0;
};
// O(1)
```

4.3 Colas dobles

- Son una generalización de las colas y las pilas, donde es posible insertar, consultar y eliminar tanto por el principio como por el final.

Especificación

```
tad DCOLA [E :: ANY]
  usa
    BOOL
  tipo
    DCola[Elem]
  operaciones
    Nuevo: → DCola[Elem]                                    /* gen */
    PonDetras: (Elem, DCola[Elem]) → DCola[Elem]            /* gen */
    ponDelante: (Elem, DCola[Elem]) → DCola[Elem]           /* mod */
    quitaUlt: DCola[Elem] - → DCola[Elem]                   /* mod */
    ultimo: DCola[Elem] - → Elem                            /* obs */
    quitaPrim: DCola[Elem] - → DCola[Elem]                  /* mod */
    primero: DCola[Elem] - → Elem                           /* obs */
    esVacio: DCola[Elem] → Bool                             /* obs */
  ecuaciones
    ∀ x, y : Elem : ∀ xs : DCola[Elem] :
    ponDelante(y, Nuevo)             = PonDetras(y, Nuevo)
    ponDelante( y, PonDetras(x, xs)) = PonDetras(x, ponDelante(y, xs))
    def quitaUlt(xs) si NOT esVacio(xs)
    quitaUlt(PonDetras(x,xs))        = xs
    def ultimo(xs) si NOT esVacio(xs)
    ultimo(PonDetras(x, xs))         = x
    def quitaPrim(xs) si NOT esVacio(xs)
    quitaPrim(PonDetras(x, xs)) = Nuevo                 si    esVacio(xs)
    quitaPrim(PonDetras(x,xs)) = PonDetras(x,quitaPrim(xs)) si NOT esVacio(xs)
    def primero(xs) si NOT esVacio(xs)
    primero(PonDetras(x, xs))        = x                si     esVacio(xs)
    primero(PonDetras(x, xs))        = primero(xs)      si NOT esVacio(xs)
    esVacio(Nuevo)                   = Cierto
    esVacio(PonDetras(x, xs))        = Falso
  errores
    quitaUlt(Nuevo)
    ultimo(Nuevo)
    quitaPrim(Nuevo)
    primero(Nuevo)
ftad
```

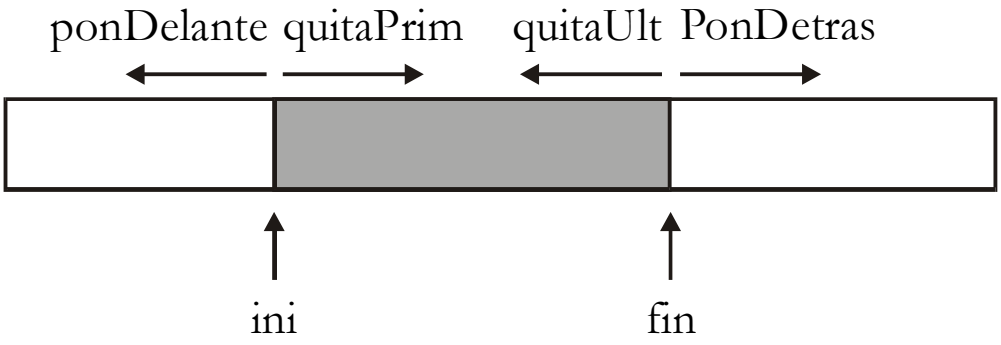
Implementación basada en un vector circular

Tipo representante

- Se utiliza la misma estructura de datos y el mismo invariante de la representación que en la implementación de las colas

Implementación de las operaciones

- Para implementar las operaciones de inserción y supresión debemos decidir cómo evolucionan los índices *ini* y *fin*.



- Observamos que las que tienen una operación análoga en las colas, se implementa de la misma forma.

COLA	DCOLA
Nuevo	Nuevo
PonDetras	PonDetras
quitaPrim	quitaPrim
primero	primero
esVacio	esVacio
	ponDelante
	quitaUlt
	ultimo

- Es necesario modificar la constructora porque ahora la primera inserción puede realizarse con *ponDelante* y no es válido el valor inicial *_fin=–1*.

```
template <class TElem>
TDColaEstatica<TElem>::TDColaEstatica( int capacidad ) :
  _capacidad(capacidad), _ini(0), _fin(_capacidad-1), _longitud(0),
  _espacio(new TElem[_capacidad]) {
};
```

- El resto, se implementan de forma similar

```
template <class TElem>
void TDColaEstatica<TElem>::ponDelante(const TElem& elem) {
// Pre: true
  if( _longitud == _capacidad )
    amplia();
  if ( _ini == 0 )
    _ini = _capacidad - 1;
  else
    _ini--;
  _longitud++;
  _espacio[_ini] = elem;
// Post: Se añade 'elem' al principio de la cola
// Si _longitud < _capacidad
//    O( TElem::operator=(TElem&), O(1) sobre tipos predefinidos
// Si _longitud == _capacidad
//    O( 2 * _capacidad * TElem::TElem() +
//       n * TElem::operator=(TElem&) + _capacidad * TElem::~TElem() ),
//    O(n) sobre tipos predefinidos
};

template <class TElem>
void TDColaEstatica<TElem>::amplia( ) {
  _capacidad *= 2;
  TElem* nuevo = new TElem[_capacidad];
  for( int i = 0; i < _longitud; i++ )
    nuevo[i] = _espacio[(_ini+i) % (_capacidad/2)];
  delete [] _espacio;
  _espacio = nuevo;
  _ini = 0;
  _fin = _longitud-1;
};
```

```
template <class TElem>
const TElem& TDColaEstatica<TElem>::ultimo( ) const
  throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido( "Cola vacía");
  return _espacio[_fin];
// Post: Devuelve el último elemento de la cola
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(1)
};

template <class TElem>
void TDColaEstatica<TElem>::quitaUlt( ) throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido( "Cola vacía" );
  if ( _fin == 0 )
    _fin = _capacidad - 1;
  else
    _fin--;
  _longitud--;
// Post: Elimina el último elemento de la cola
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(1)
};
```

Implementación dinámica

Tipo representante

- Se utiliza la misma estructura de datos y el mismo invariante de la representación que en la implementación de las colas

Implementación de las operaciones

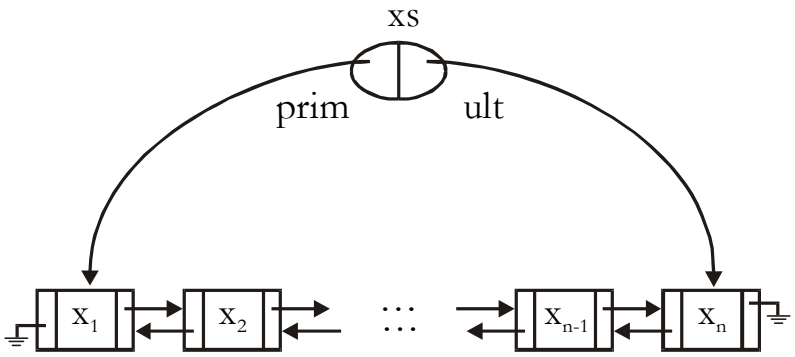
- Las que tienen una operación análoga en las colas, se implementa de la misma forma. *ponDelante* y *ultimo* se implementan de la forma obvia con complejidad $O(1)$, pero no así *quitaUlt*.
- Al eliminar el último nodo, es necesario hacer que *ult* pase a apuntar al penúltimo, pero para acceder a éste es necesario recorrer la lista enlazada, con lo que se obtiene una complejidad $O(n)$.

```
template <class TElem>
void TDColaDinamica<TElem>::quitaUlt( ) throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido("Error al quitar el último de la cola vacía");
  else {
    TNodoCola<TElem>* p = _prim;
    if ( p == _ult ) {
      _prim = _ult = 0;
      delete p;
    }
    else {
      while ( p->sig() != _ult )
        p = p->sig();
      p->_sig = 0;
      delete _ult;
      _ult = p;
    }
  }
// Post: Elimina el último elemento de la cima
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(TElem::~TElem()), O(1) sobre tipos predefinidos
};
```

Implementación dinámica con encadenamiento doble

Tipo representante

- La idea de la representación



- Estructura de datos

```
template <class TElem>
class TNodoCola {
  public:
    friend TDColaDinamica<TElem>;
    ...

  private:
    TElem _elem;
    TNodoCola<TElem> *_sig, *_ant;

  ...
};

template <class TElem>
class TDColaDinamica {

  ...

  private:
     TNodoCola<TElem> *_prim, *_ult;

  ...
};
```

- Invariante de la representación

  Dada *xs : TDColaDinamica*

$$R(xs) \Leftrightarrow_{def} R_{CDV}(xs) \vee R_{CDNV}(xs)$$

donde R_{CDV} se refieren a las condiciones de una cola doble vacía y R_{CDNV} a las de una que no está vacía

$$\begin{array}{l} R_{CDV}(xs) \\ \Leftrightarrow_{def} \\ \quad xs.\_prim = 0 \wedge xs.\_ult = 0 \end{array}$$

$$\begin{array}{l} R_{CDNV}(xs) \\ \Leftrightarrow_{def} \\ \quad xs.\_prim \neq 0 \wedge xs.\_ult \neq 0 \wedge \\ \quad xs.\_prim\text{->}ant = 0 \wedge \\ \quad buenaColaDoble(xs.\_prim, xs.\_ult) \end{array}$$

y el predicado *buenaColaDoble* por su parte

$$\begin{array}{l} buenaColaDoble(p,q) \\ \Leftrightarrow_{def} \\ \quad (p = q \wedge ubicado(p) \wedge R(p\text{->}\_elem) \wedge p\text{->}\_sig = 0) \vee \\ \quad (p \neq q \wedge ubicado(p) \wedge R(p\text{->}\_elem) \wedge \\ \quad\ p\text{->}\_sig\text{->}\_ant = p \wedge \\ \quad\ p \notin cadena(p\text{->}\_sig) \wedge buenaColaDoble(p\text{->}\_sig, q)) \end{array}$$

$$\begin{array}{ll} cadena(p) =_{def} \varnothing & \textbf{si } p = nil \\ cadena(p) =_{def} \{p\} \cup cadena(p\text{->}\_sig) & \textbf{si } p \neq nil \end{array}$$

Implementación de las operaciones

- La clase de los nodos

```
template <class TElem>
TNodoCola<TElem>::TNodoCola( const TElem& elem, TNodoCola<TElem>* sig,
                             TNodoCola<TElem>* ant ) :
  _elem(elem), _sig(sig), _ant(ant) {
// O(TElem::TElem(TElem&), O(1) sobre tipos predefinidos
};

template <class TElem>
const TElem& TNodoCola<TElem>::elem() const {
  return _elem;
// O(1)
}

template <class TElem>
TNodoCola<TElem>* TNodoCola<TElem>::sig() const {
  return _sig;
//O(1)
}

template <class TElem>
TNodoCola<TElem>* TNodoCola<TElem>::ant() const {
  return _ant;
//O(1)
}
```

- Las operaciones de las colas dobles

```
template <class TElem>
TDColaDinamica<TElem>::TDColaDinamica( ) :
  _prim(0), _ult(0) {
};


// constructora de copia, destructora y operador de asignación
// igual que siempre

// operaciones privadas de copia y anulación

template <class TElem>
void TDColaDinamica<TElem>::libera() {
  while (_prim != 0) {
    TNodoCola<TElem>* tmp = _prim;
    _prim = _prim->sig();
    delete tmp;
  }
// O(n * TElem::~TElem()), O(n) sobre tipos predefinidos
};

template <class TElem>
void TDColaDinamica<TElem>::copia(const TDColaDinamica<TElem>& cola) {
  if ( cola.esVacio() )
    _prim = _ult = 0;
  else {
    TNodoCola<TElem> *antCopia, *actCopia, *act;
    act = cola._prim;
    _prim = new TNodoCola<TElem>( act->elem(), 0 );
    actCopia = _prim;
    while ( act->sig() != 0 ) {
      act = act->sig();
      antCopia = actCopia;
      actCopia = new TNodoCola<TElem>( act->elem(), 0, antCopia );
      antCopia->_sig = actCopia;
    }
    _ult = actCopia;
  }
// O(n' * TElem::TElem(TElem&)), O(n') sobre tipos predefinidos
// donde n' es el número de elementos de la cola a copiar
};
```

```
template <class TElem>
void TDColaDinamica<TElem>::ponDetras(const TElem& elem) {
// Pre: true
  TNodoCola<TElem>* p = new TNodoCola<TElem>(elem);
  if( esVacio() )
    _prim = _ult = p;
  else {
    _ult->_sig = p;
    p->_ant = _ult;
    _ult = p;
  }
// Post: Se añade 'elem' al final de la cola
// O(TElem::TElem(TElem&)), O(1) sobre tipos predefinidos
};

template <class TElem>
void TDColaDinamica<TElem>::ponDelante(const TElem& elem) {
// Pre: true
  TNodoCola<TElem>* p = new TNodoCola<TElem>(elem);
  if( esVacio() )
    _prim = _ult = p;
  else {
    _prim->_ant = p;
    p->_sig = _prim;
    _prim = p;
  }
// Post: Se añade 'elem' al principio de la cola
// O(TElem::TElem(TElem&)), O(1) sobre tipos predefinidos
};

template <class TElem>
const TElem& TDColaDinamica<TElem>::primero( ) const
  throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido("Error: no existe el primero de la cola vacía");
  else
    return _prim->elem();
// Post: Devuelve el primer elemento de la cola
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(1)
};
```

```
template <class TElem>
const TElem& TDColaDinamica<TElem>::ultimo( ) const
  throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido("Error: no existe el último de la cola vacía");
  else
    return _ult->elem();
// Post: Devuelve el último elemento de la cola
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(1)
};


template <class TElem>
void TDColaDinamica<TElem>::quitaPrim( ) throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido("Error: no existe el primero de la cola vacía");
  else {
    TNodoCola<TElem>* tmp = _prim;
    _prim = _prim->sig();
    if( _prim == 0 )
      _ult = 0;
    else
      _prim->_ant = 0;
    delete tmp;
  }
// Post: Elimina el primer elemento de la cima
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(TElem::~TElem()), O(1) sobre tipos predefinidos
};
```

```
template <class TElem>
void TDColaDinamica<TElem>::quitaUlt( ) throw (EAccesoIndebido) {
// Pre: ! esVacio( )
  if( esVacio() )
    throw EAccesoIndebido("Error: no existe el último de la cola vacía");
  else {
    TNodoCola<TElem>* tmp = _ult;
    _ult = _ult->ant();
    if ( _ult == 0 )
      _prim = 0;
    else
      _ult->_sig = 0;
    delete tmp;
  }
// Post: Elimina el último elemento de la cima
// Lanza la excepción EAccesoIndebido si la cola está vacía
// O(TElem::~TElem()), O(1) sobre tipos predefinidos
};

template <class TElem>
bool TDColaDinamica<TElem>::esVacio( ) const {
// Pre: true
  return _prim == 0;
// Post: Devuelve true | false según si la pila está o no vacía
// O(1)
};
```

4.4 Listas

- Es un TAD que representa a una colección de elementos de un mismo tipo que generaliza a las colas dobles incluyendo algunas operaciones adicionales

Operaciones de LISTA	Operaciones de DCOLA
Nuevo	Nuevo
PonDelante	ponDelante
ponDetras	PonDetras
primero	primero
quitaPrim	quitaPrim
último	último
quitaUlt	quitaUlt
concatena	—
esVacio	esVacio
numElem	—
elemEn	—

  - *concatena* construye una lista a partir de dos
  - *numElem* devuelve el número de elementos de la lista
  - *elemEn* permite acceder al elemento *i*-ésimo de una lista.

- Debido a estas similitudes, las implementaciones que consideraremos serán similares a las estudiadas para las colas dobles.

Especificación

```
tad LISTA[E :: ANY]
  usa
    BOOL, NAT
  tipo
    Lista[Elem]
  operaciones
    Nuevo: → Lista[Elem]                                        /* gen */
    PonDelante: (Elem, Lista[Elem]) → Lista[Elem]               /* gen */
    ponDetras: (Lista[Elem], Elem) → Lista[Elem]                /* mod */
    primero: Lista[Elem] - → Elem                               /* obs */
    quitaPrim: Lista[Elem] - → Lista[Elem]                      /* mod */
    último: Lista[Elem] - → Elem                                /* obs */
    quitaUlt: Lista[Elem] - → Lista[Elem]                       /* mod */
    concatena: (Lista[Elem], Lista[Elem]) → Lista[Elem]         /* mod */
    esVacio: Lista[Elem] → Bool                                 /* obs */
    numElem: Lista[Elem] → Nat                                  /* obs */
    elemEn: (Lista[Elem], Nat) - → Elem                         /* obs */
  ecuaciones
    ∀ x, y: Elem : ∀ xs, ys : Lista[Elem] : ∀ n : Nat :
    concatena(Nuevo, ys)              = ys
    concatena(PonDelante(x, xs), ys) = PonDelante(x, concatena(xs, ys))
    ponDetras(xs, x)                  = concatena(xs, PonDelante(x, Nuevo))
    esVacio(Nuevo)                    = Cierto
    esVacio(PonDelante(x, xs))        = Falso
    def primero(xs) si NOT esVacio(xs)
    primero(PonDelante(x, xs))        = x
    def quitaPrim(xs)   si NOT esVacio(xs)
    quitaPrim(PonDelante(x, xs))      = xs
    def último(xs)  si NOT esVacio(xs)
    último(PonDelante(x, xs))         = x              si     esVacio(xs)
    último(PonDelante(x, xs))         = último(xs)     si NOT esVacio(xs)
    def quitaUlt(xs)  si NOT esVacio(xs)
    quitaUlt(PonDelante(x, xs))       = Nuevo          si     esVacio(xs)
    quitaUlt(PonDelante(x, xs))       = PonDelante(x, quitaUlt(xs))
                                                       si NOT esVacio(xs)
    numElem(Nuevo)                    = Cero
    numElem(PonDelante(x, xs))        = Suc(numElem(xs))
    def elemEn(xs, n) si Suc(Cero) ≤ n ≤ numElem(xs)
    elemEn(PonDelante(x, xs), Suc(Cero)) = x
    elemEn(PonDelante(x, PonDelante(y, ys)), Suc(Suc(n))
                                      =f elemEn(PonDelante(y,ys), Suc(n))
```

```
  errores
    primero(Nuevo)
    quitaPrim(Nuevo)
    último(Nuevo)
    quitaUlt(Nuevo)
    elemEn(xs, n) si (n == Cero) OR (n > (numElem xs))
ftad
```

Implementación estática basada en un vector circular

Tipo representante

- Se utiliza la misma estructura de datos y el mismo invariante de la representación que en la implementación de las colas dobles.

Implementación de las operaciones

- Las que tienen un equivalente en las colas dobles se implementan exactamente de la misma forma, de forma que sólo resta implementar *concatena*, *numElem* y *elemEn*.

```
template <class TElem>
void TListaEstatica<TElem>::concatena( const TListaEstatica<TElem>& lista )
{
// Pre: true
  if( _longitud + lista._longitud > _capacidad )
    amplia( _longitud + lista._longitud );
  for ( int i = 0; i < lista._longitud; i++ ) {
    _fin = ( _fin + 1 ) % _capacidad;
    _espacio[_fin] = lista._espacio[ (lista._ini + i) % lista._capacidad ];
  }
  _longitud += lista._longitud;
// Post: concatena la lista pasada como parámetro, dejándola vacía
// Si _longitud + lista._longitud <= _capacidad
//    O( n'*TElem::operator=(TElem&) ), O(n') sobre tipos predefinidos
// Si _longitud + lista._longitud > _capacidad
//    O( _longitud + lista._longitud * TElem::TElem() +
//       n * TElem::operator=(TElem&) + _capacidad * TElem::~TElem() +
//       n'*TElem::operator=(TElem&) ),
//    O(n+n') sobre tipos predefinidos
// siendo n' el número de elementos de la lista pasada como parámetro
}
```

Es necesario cambiar la implementación de *amplia* para que reciba la nueva longitud como parámetro.

```
template <class TElem>
void TListaEstatica<TElem>::amplia( int longitud ) {
// Pre: longitud >= _capacidad
  TElem* nuevo = new TElem[longitud];
  for( int i = 0; i < _longitud; i++ )
    nuevo[i] = _espacio[(_ini+i) % (_capacidad)];
  delete [] _espacio;
  _capacidad = longitud;
  _espacio = nuevo;
  _ini = 0;
  _fin = _longitud-1;
// Post: _espacio es una array de 'longitud' elementos
//       los elementos del valor inicial de _espacio ocupan
//       las posiciones 0 .. _ini + _longitud - 1
// O( longitud * TElem::TElem() +
//    n * TElem::operator=(TElem&) +
//    _capacidad * TElem::~TElem() ),
// O(n) sobre tipos predefinidos
};
```

- Las operaciones *numElem* y *elemEn* se implementan de forma trivial.

```
template <class TElem>
int TListaEstatica<TElem>::numElem( ) const {
// Pre: true
  return _longitud;
// Post: Devuelve el número de elementos de la lista
// O(1)
}

template <class TElem>
const TElem& TListaEstatica<TElem>::elemEn( int n ) const
  throw (EAccesoIndebido) {
// Pre: 1 <= n <= numElem()
  if ( ( n < 1 ) || ( n > numElem() ) )
    throw EAccesoIndebido("Error: posición inexistente");
  return _espacio[ (_ini + n - 1) % _capacidad ];
// Post: Devuelve el elemento en la posición n
// O(1)
}
```

Implementación dinámica con encadenamiento doble

Tipo representante

- La estructura de datos es la misma que se utiliza en las colas dobles excepto porque se guarda la longitud, para así poder implementar *numElem* con complejidad $O(1)$.

```
template <class TElem>
class TNodoLista {
  public:
    friend TListaDinamica<TElem>;
    ...

  private:
    TElem _elem;
    TNodoCola<TElem> *_sig, *_ant;

  ...
};

template <class TElem>
class TListaDinamica {

  ...

  private:
     TNodoLista<TElem> *_prim, *_ult;
     int _longitud;

  ...
};
```

- El invariante de la representación es el mismo que en las colas dobles, añadiendo la condición referente al campo longitud:

$$R(xs) \Leftrightarrow_{def} (R_{CDV}(xs) \vee R_{CDNV}(xs)) \wedge xs.\_longitud = longitud(xs.\_prim)$$

$$longitud(p) =_{def} 0 \quad \textbf{si } p = 0$$
$$longitud(p) =_{def} 1 + longitud(p\text{->}\_sig) \quad \textbf{si } p \neq 0$$

Implementación de las operaciones

- Para las operaciones que tienen una operación análoga en las colas dobles, podemos utilizar la implementación correspondiente, añadiendo las asignaciones necesarias para que se cumpla la parte del invariante de la representación que hace referencia al campo *longitud*:
  - al crear una lista vacía inicializamos la longitud a 0
  - al insertar incrementamos en 1 la longitud
  - al eliminar decrementamos en 1 la longitud

- En cuanto a las tres operaciones adicionales

```
template <class TElem>
int TListaDinamica<TElem>::numElem( ) const {
// Pre: true
  return _longitud;
// Post: Devuelve el número de elementos de la lista
// O(1)
}

template <class TElem>
const TElem& TListaDinamica<TElem>::elemEn( int n ) const
  throw (EAccesoIndebido) {
// Pre: 1 <= n <= numElem()
  if ( ( n < 1 ) || ( n > numElem() ) )
    throw EAccesoIndebido("Error: posición inexistente");

  TNodoLista<TElem>* p = _prim;
  for ( int i = 1; i < n; i++ )
    p = p-> sig();
  return p->elem();
// Post: Devuelve el elemento en la posición n
// O(n)
}
```

Como es lógico, al tener una implementación enlazada la complejidad del acceso directo pasa a ser lineal, $O(n)$, debido a que esta operación implica un recorrido de la lista enlazada. Nótese asimismo que la n se refiere a la longitud de la lista y no al parámetro n de la operación.

- Aunque en la implementación estática es inevitable que la concatenación tenga complejidad $O(n)$, la implementación dinámica permite realizarla en $O(1)$ de dos formas distintas:
  - Permitir la compartición de estructura, haciendo que los nodos de la lista pasada como parámetro formen parte de ésta y del resultado de la concatenación. Esta solución presenta los problemas ya comentados sobre compartición de estructura.
  - Implementar la operación *concatena* como un procedimiento que devuelve vacía la lista que recibe como argumento, de esta forma es el cliente del TAD quien decide en cada uso particular si necesita o no realizar una copia de la lista pasada como argumento antes de ejecutar la concatenación.

  Optamos por la segunda alternativa y modificamos la especificación para reflejar este nuevo comportamiento

```
void TListaDinamica<TElem>::concatena( TListaDinamica<TElem>& lista );
// Pre: true
// Post: concatena la lista pasada como parámetro, dejándola vacía
```

  que se implementa trivialmente:

```
template <class TElem>
void TListaDinamica<TElem>::concatena( TListaDinamica<TElem>& lista ) {
// Pre: true
  if ( ! lista.esVacio() ) {
    if ( esVacio() ) {
      _prim = lista._prim;
      _ult = lista._ult;
      _longitud = lista._longitud;
    }
    else {
      _ult->_sig = lista._prim;
      lista._prim->_ant = _ult;
      _ult = lista._ult;
      _longitud += lista._longitud;
    }
    lista._prim = 0;
    lista._ult = 0;
    lista._longitud = 0;
  }
// Post: concatena la lista pasada como parámetro, dejándola vacía
// O(1)
}
```

- De esta forma, si se quiere mantener la lista a concatenar basta con utilizar una copia en la llamada al método *concatena*:

```
TListaDinamica<int> l1, l2;

   ...

   l1.concatena( TListaDinamica<int>(l2) );
```

el objeto resultado de la copia es temporal y como tal se libera automáticamente al terminar la ejecución de *concatena*.

Comparación de las implementaciones

- En términos de la complejidad de las operaciones (sobre tipos predefinidos y sin tener en cuenta los casos en que el array aumenta de tamaño):

Operaciones	Estática	Dinámica
Nuevo	$O(1)$	$O(1)$
PonDelante	$O(1)$	$O(1)$
ponDetras	$O(1)$	$O(1)$
primero	$O(1)$	$O(1)$
quitaPrim	$O(1)$	$O(1)$
último	$O(1)$	$O(1)$
quitaUlt	$O(1)$	$O(1)$
concatena	$O(n)$	$O(1)$
esVacio	$O(1)$	$O(1)$
numElem	$O(1)$	$O(1)$
elemEn	$O(1)$	$O(n)$

Siendo n la longitud de la lista.

4.5 Secuencias

- Las secuencias son colecciones de datos que incluyen operaciones específicas para el recorrido de la colección. Estas operaciones se basan en la existencia de una posición distinguida dentro de la colección: el *punto de interés*. Se incluyen operaciones de inserción, supresión y consulta del elemento distinguido, así como de desplazamiento del punto de interés al principio de la colección y avance del punto de interés.

Especificación

- La especificación se basa en la idea de que podemos representar las secuencias como dos listas, siendo el punto de interés el punto divisorio entre las dos partes y el elemento distinguido el primero de la parte derecha.
  - Nótese que la lista de la derecha puede estar vacía, lo que se interpreta como una colección donde el punto de interés está una posición más allá del último elemento y, por lo tanto, no hay elemento distinguido (esto implica la parcialidad de algunas operaciones).

- Aparecen en esta especificación dos elementos nuevos:
  - El uso privado de un TAD, de forma que los clientes de este TAD no tienen acceso al TAD usado (a nivel de implementación es equivalente a usar una unidad en la parte de implementación en lugar de hacerlo en la de interfaz). Como algunas de las operaciones usadas tienen el mismo nombre que las definidas en este TAD, es necesario cualificarlas.
  - La definición de una generadora privada, una operación que los clientes no pueden invocar pero que facilita la construcción de la especificación.

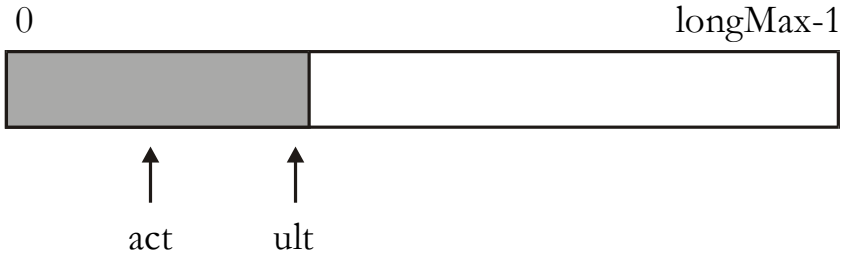
- En la especificación de este TAD es necesario decidir cómo se comportan las operaciones de inserción y supresión con respecto al punto de interés. Las decisiones que tomamos son:
  - Se incluye una operación que permite insertar un elemento a la izquierda del punto de interés, con lo que el elemento distinguido –si lo hay– sigue siendo el mismo que antes de la inserción.
  - Se incluye una operación que permite eliminar el elemento distinguido, si lo hay, de forma que después de la supresión el elemento distinguido, si lo hubiera, pasaría a ser el siguiente al elemento eliminado.

```
tad SEC[E :: ANY]
  usa
    BOOL
  usa privadamente
    LISTA[E]
  tipo
    Sec[Elem]
  operaciones
    Nuevo: → Sec[Elem]                                  /* gen */
    Inserta: (Sec[Elem], Elem) → Sec[Elem]              /* gen */
    borra: Sec[Elem] - → Sec[Elem]                      /* mod */
    actual: Sec[Elem] - → Elem                          /* obs */
    Avanza: Sec[Elem] - → Sec[Elem]                     /* gen */
    Reinicia: Sec[Elem] → Sec[Elem]                     /* gen */
    esVacio: Sec[Elem] → Bool                           /* obs */
    esFin: Sec[Elem] → Bool                             /* obs */
  operaciones privadas
    S : (Lista[Elem], Lista[Elem]) → Sec[Elem]          /* gen */
    piz, pdr, cont: Sec[Elem] → Lista[Elem]             /* obs */
  ecuaciones
    ∀ s : Sec[Elem] : ∀ iz, dr : Lista[Elem] : ∀ x : Elem :
    Nuevo  = S(LISTA.Vacio, LISTA.Vacio)
    Inserta(S(iz, dr), x)           = S(ponDetras(iz, x), dr)
    esFin(S(iz, dr))                = LISTA.esVacio(dr)
    def borra(s)  si NOT esFin(s)
    borra(S(iz, PonDelante(x, dr)))  = S(iz, dr)
    def actual(s) si NOT esFin(s)
    actual(S(iz, PonDelante(x, dr))) = x
    def Avanza(s) si NOT esFin(s)
    Avanza(S(iz, PonDelante(x, dr))) = S(ponDetras(iz, x), dr)
    Reinicia(S(iz, dr))             = S(LISTA.Vacio, concatena(iz, dr))
    esVacio(S(iz, dr))              = LISTA.esVacio(iz) AND
                                      LISTA.esVacio(dr)
    piz(S(iz, dr))                  = iz
    pdr(S(iz, dr))                  = dr
    cont(S(iz, dr))                 = concatena(iz, dr)
  errores
    borra(s)  si esFin(s)
    actual(s) si esFin(s)
    avanza(s) si esFin(s)
ftad
```

Implementación estática

Tipo representante

- Se utiliza una idea similar a la estudiada para la representación de las pilas



Nótese que *act* apunta al primero de la parte derecha, y que cuando el punto de interés esté a la derecha del todo, su valor será *ult+1*.

- La estructura de datos

```
template <class TElem>
class TSecuencia {
  public:
     static const int longMax = 100;
  ...
  private:
  // Variables privadas
     int _ult, _act;
     TElem _espacio[longMax];
};
```

- Invariante de la representación

Dada *xs : TSecuencia*

$$
\begin{aligned}
& R(xs) \\
\Leftrightarrow_{def}\ & -1 \le xs.\_ult \le longMax-1 \wedge 0 \le xs.\_act \le xs.\_ult + 1 \wedge \\
& \forall\ i : 0 \le i \le xs.\_ult : R(xs.\_espacio[i])
\end{aligned}
$$

Nótese que:

– $ult = -1 \Rightarrow act = 0$

En este caso la secuencia está vacía

– $ult = longMax-1 \Rightarrow 0 \le act \le longMax$

– $act = ult + 1 \Rightarrow pdr(xs) = \text{LISTA}.Nuevo$

El punto de interés está más allá del final de la secuencia y no hay elemento distinguido.

Implementación de las operaciones

- Con esta representación, la eficiencia que se puede conseguir para las operaciones realizando las implementaciones obvias:

Operación	Complejidad
Nuevo	$O(1)$
Inserta	$O(n)$
borra	$O(n)$
actual	$O(1)$
Avanza	$O(1)$
Reinicia	$O(1)$
esVacio	$O(1)$
esFin	$O(1)$

Implementación basada en una pareja de listas

Tipo representante

- Seguimos directamente la idea de la especificación del TAD

- Estructura de datos

```
template <class TElem>
class TSecuencia {
  ...
  private:
  // Variables privadas
     TListaDinamica _piz, _pdr;
};
```

- El invariante de la representación queda trivial

  Dada *xs : TSecuencia*

  $R(xs) \Leftrightarrow_{def} R(xs._{piz}) \wedge R(xs._{pdr})$

Implementación de las operaciones

- Haciendo uso de la implementación enlazada de *TLista* podemos conseguir fácilmente una implementación con los siguientes tiempos de ejecución (sin tener en cuenta el coste de las copias y las destrucciones)

Operación	Complejidad	Idea
Nuevo	$O(1)$	*piz.Nuevo*; *pdr.Nuevo*
Inserta	$O(1)$	*piz.ponDetras(x)*
borra	$O(1)$	*pdr.quitaPrim*
actual	$O(1)$	*pdr.primero*
Avanza	$O(1)$	*piz.ponDetras(pdr.primero)*; *pdr.quitaPrim*
Reinicia	$O(1)$	*piz.concatena(pdr); pdr = piz; piz.Nuevo;*
esVacio	$O(1)$	*piz.esVacio and pdr.esVacio*
esFin	$O(1)$	*pdr.esVacio*

El inconveniente de esta implementación radica en las copias y destrucciones innecesarias debido a:

– la asignación *pdr* = *piz* en *Reinicia*,

$O(\ n \cdot$ TElem::TElem(TElem&) + $n \cdot$ TElem::~TElem( ))

– la copia del elemento para luego destruirlo en *Avanza*,

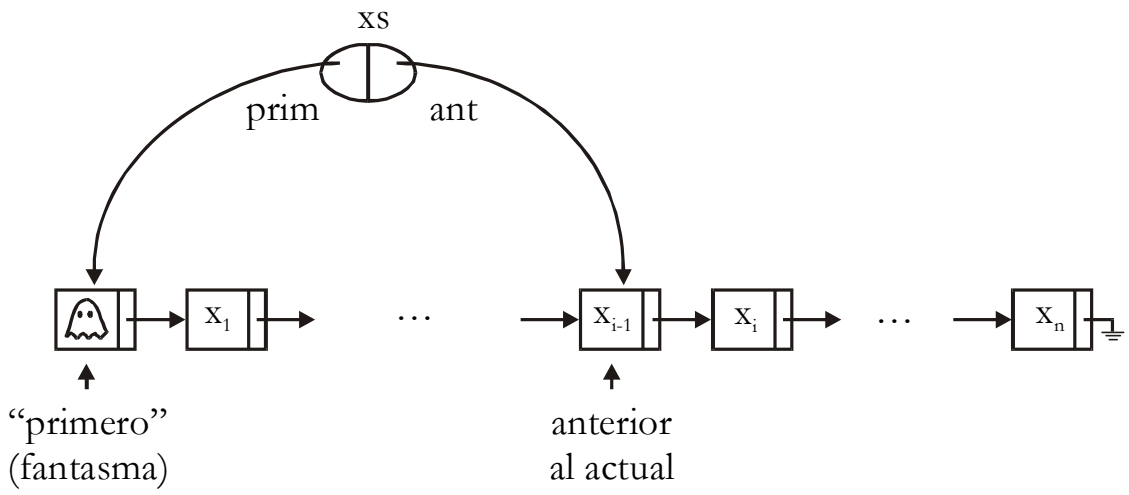
$O(\ $TElem::TElem(TElem&) + TElem::~TElem( ))

El primer problema (el más grave) se podría resolver si las listas dispusiesen de una operación de concatenación por la izquierda que vaciase la lista a concatenar, de forma similar a la operación *concatena*.

Implementación dinámica

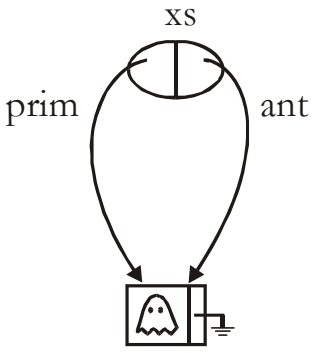
Tipo representante

- La idea de esta representación:



  Elegimos apuntar al anterior al actual –al último de la parte izquierda– porque así podemos realizar todas las operaciones en tiempo $O(1)$.

- Aparece aquí una idea nueva de implementación: el *nodo fantasma*. La existencia de este nodo permite simplificar algunos algoritmos, sin perjudicar gravemente el consumo de espacio. Por ejemplo, de esta forma nos evitamos tratar de manera especial el caso de una secuencia vacía.



- Estructura de datos

```
template <class TElem>
class TNodoSecuencia {
  private:
    TElem _elem;
    TNodoSecuencia<TElem>* _sig;
  ...
};

template <class TElem>
class TSecuenciaDinamica {
  ...

  private:
     TNodoSecuencia<TElem> *_pri, *_ant;
  ...
};
```

- Invariante de la representación

  Dada *xs : TSecuencia*

$$
\begin{array}{l}
\quad R(xs) \\
\Leftrightarrow_{def} \\
\quad xs.\_pri \neq nil \wedge ubicado(xs.\_pri) \wedge \\
\quad xs.\_ant \neq nil \wedge ubicado(xs.\_ant) \wedge \\
\quad xs.\_ant \in cadena(xs.\_pri) \wedge xs.\_pri \notin cadena(xs.\_pri\text{->}\_sig) \wedge \\
\quad cadenaCorrecta(xs.pri\hat{}.sig)
\end{array}
$$

  Donde, como siempre, *cadena(p)* se define como

$$
cadena(p) =_{def} \begin{cases} \varnothing & \textbf{si } p = 0 \\ \{p\} \cup cadena(p\text{->}\_sig) & \textbf{si } p \neq 0 \end{cases}
$$

  Y donde una *cadena* de enlaces es correcta si todos los punteros apuntan a nodos diferentes correctamente construidos

$$
\begin{array}{l}
\quad cadenaCorrecta(p) \\
\Leftrightarrow_{def} \\
\quad p = 0 \vee \\
\quad (p \neq 0 \wedge ubicado(p) \wedge R(p\text{->}\_elem) \wedge \\
\quad\ \ p \notin cadena(p\text{->}\_sig) \wedge cadenaCorrecta(p\text{->}\_sig))
\end{array}
$$

Implementación de las operaciones

- La clase de los nodos

```
template <class TElem>
class TNodoSecuencia {
  private:
    TElem _elem;
    TNodoSecuencia<TElem>* _sig;
    TNodoSecuencia( );
    TNodoSecuencia( const TElem&, TNodoSecuencia<TElem>* = 0);
  public:
    const TElem& elem() const;
    TNodoSecuencia<TElem> * sig() const;
    friend TSecuenciaDinamica<TElem>;
};
```

Es necesaria una constructora sin parámetros para poder construir el nodo fantasma. El campo *_elem* de ese nodo contendrá basura si se trata de un tipo primitivo o el resultado de llamar a la constructora por defecto si no lo es.

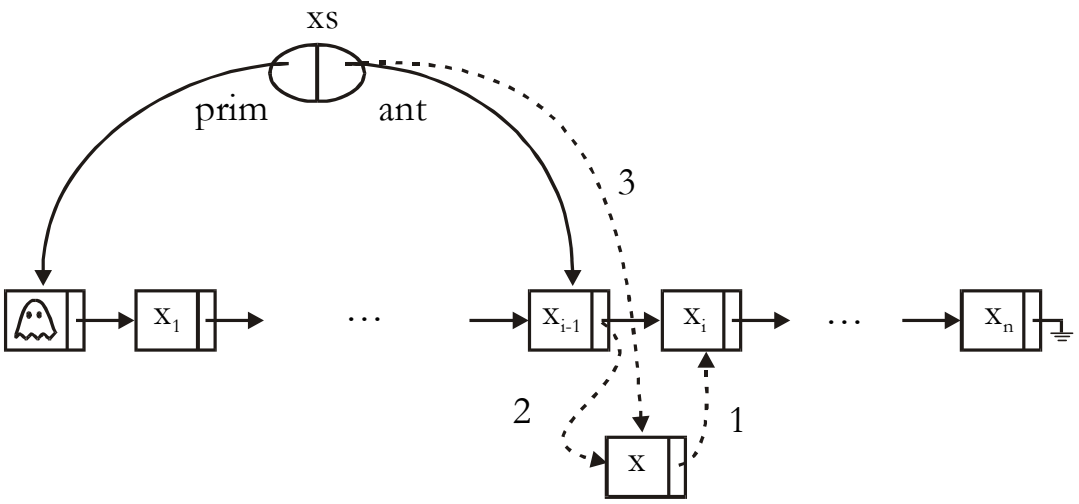
```
template <class TElem>
TNodoSecuencia<TElem>::TNodoSecuencia( ) : _sig(0) {
// O( TElem::TElem() ), O(1) sobre tipos predefinidos
};

template <class TElem>
TNodoSecuencia<TElem>::TNodoSecuencia( const TElem& elem,
                                       TNodoSecuencia<TElem>* sig ) :
  _elem(elem), _sig(sig) {
// O( TElem::TElem(TElem&) ), O(1) sobre tipos predefinidos
};

template <class TElem>
const TElem& TNodoSecuencia<TElem>::elem() const {
  return _elem;
// O(1)
}

template <class TElem>
TNodoSecuencia<TElem>* TNodoSecuencia<TElem>::sig() const {
  return _sig;
// O(1)
}
```

- Construcción, destrucción y asignación

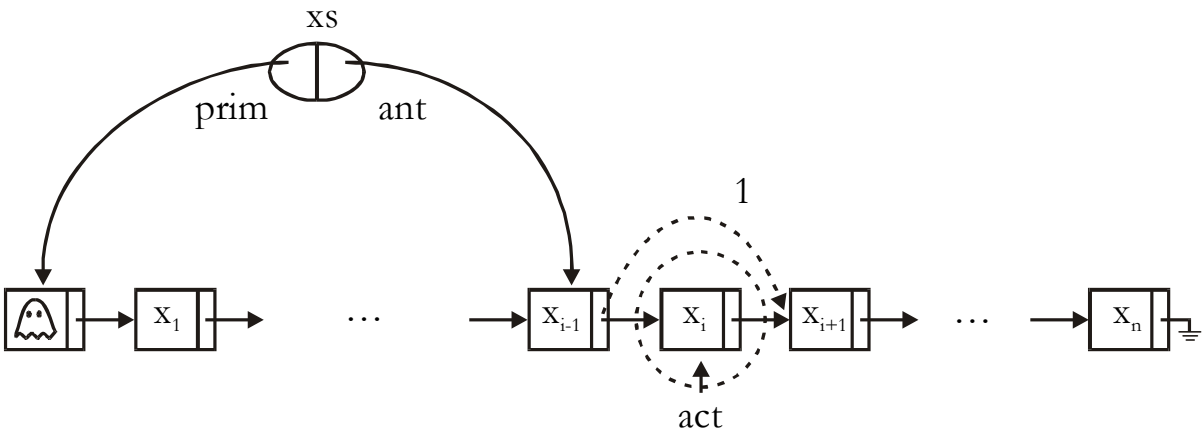
```
template <class TElem>
TSecuenciaDinamica<TElem>::TSecuenciaDinamica( ) {
   _pri = _ant = new TNodoSecuencia<TElem>();
// O( TElem::TElem() ), O(1) sobre tipos predefinidos
};

// constructora de copia, destructora y asignación igual que siempre

template <class TElem>
void TSecuenciaDinamica<TElem>::libera() {
  while ( _pri != 0 ){
    _ant = _pri;
    _pri = _pri->sig();
    delete _ant;
  }
// O( n * TElem::~TElem() ), O(n) sobre tipos predefinidos
};

template <class TElem>
void TSecuenciaDinamica<TElem>::copia(
  const TSecuenciaDinamica<TElem>& secuencia) {
  TNodoSecuencia<TElem> *antCopia, *actCopia, *act;
  act = secuencia._pri;
  actCopia = _ant = _pri = new TNodoSecuencia<TElem>( );
  while ( act->sig() != 0 ) {
    act = act->sig();
    antCopia = actCopia;
    actCopia = new TNodoSecuencia<TElem>( act->elem(), 0 );
    if ( secuencia._ant == act ) _ant = actCopia;
      antCopia->_sig = actCopia;
  }
// O( n' * TElem::TElem(TElem&) ), O(n') sobre tipos predefinidos
// siendo n' el número de elementos de la secuencia a copiar
};
```

- Operaciones de las secuencias

```
template <class TElem>
void TSecuenciaDinamica<TElem>::inserta(const TElem& elem) {
  TNodoSecuencia<TElem>* nuevo =
    new TNodoSecuencia<TElem> ( elem, _ant->sig() );   // 1
  _ant->_sig = nuevo;                                  // 2
  _ant = nuevo;                                        // 3
// O( TElem::TElem(TElem&) ), O(1) sobre tipos predefinidos
};
```




```
template <class TElem>
void TSecuenciaDinamica<TElem>::borra( ) throw (EAccesoIndebido) {
  if( esFin() )
    throw EAccesoIndebido("Error: el punto de interés está al final");
  TNodoSecuencia<TElem>* act = _ant->sig();
  _ant->_sig = act->sig();                    // 1
  delete act;
// O( TElem::~TElem() ), O(1) sobre tipos predefinidos
};
```




```
template <class TElem>
const TElem& TSecuenciaDinamica<TElem>::actual( ) const
  throw (EAccesoIndebido) {
  if( esFin() )
    throw EAccesoIndebido("Error: el punto de interés está al final");
  return _ant->sig()->elem();
// O(1)
};


template <class TElem>
void TSecuenciaDinamica<TElem>::avanza( ) throw (EAccesoIndebido) {
  if( esFin() )
    throw EAccesoIndebido("Error: el punto de interés está al final");
  _ant = _ant->sig();
// O(1)
};


template <class TElem>
void TSecuenciaDinamica<TElem>::reinicia( ) {
  _ant = _pri;
// O(1)
};

template <class TElem>
bool TSecuenciaDinamica<TElem>::esFin( ) const {
  return _ant->sig() == 0;
// O(1)
};
```

```
template <class TElem>
bool TSecuenciaDinamica<TElem>::esVacio( ) const {
  return _pri->sig() == 0;
// O(1)
};
```

- Con esta implementación conseguimos que todas las operaciones tengan complejidad constante (salvo las copias, destrucciones y asignaciones).

Recorrido y búsqueda en una secuencia

- Utilizando las operaciones de las secuencias resulta muy sencillo implementar algoritmos de recorrido y búsqueda secuencial sobre la colección de datos que una secuencia almacena.

Esquema de recorrido de una secuencia

- Un procedimiento genérico de recorrido en términos de un procedimiento genérico de tratamiento *trata*

```
template <class TElem>
void recorre ( TSecuencia<TElem>& xs ) {
// Pre: el punto de interés de xs está al principio
  TElem x;
  while ( ! xs.esFin() ) {
    x = xs.actual();
    trata(x);
    xs.avanza();
  }
// Post: se ha aplicado la función trata sobre
// todos los elementos de xs, y el punto de interés de xs está al final
}
```

Comentarios:

- Como variante de este esquema, podríamos no exigir en la precondición que el punto de interés estuviese al principio, con lo que el recorrido se realizaría desde la posición del punto de interés hasta el final de la secuencia.
- Si nos interesa que la operación *trata* modifica el estado de los elementos de la secuencia, tenemos dos opciones:
  - utilizar punteros como elementos de la secuencia, o
  - tratar cada elemento, borrar el valor antiguo y volver a insertar el nuevo.

Por ejemplo:

```
void incrementa( TSecuenciaDinamica<int>& xs ) {
  int x;
  while ( ! xs.esFin() ) {
    x = xs.actual();
    ++x;
    xs.borra(); xs.inserta(x);
  }
}
```

Esquema de búsqueda de una secuencia

- Un procedimiento genérico de búsqueda en términos de un procedimiento genérico que detecta si un elemento cumple una determinada propiedad *propiedad*

```
template <class TElem>
void busca ( TSecuencia<TElem>& xs, bool& encontrado ) {
// Pre: el punto de interés de xs está al principio
  TElem x;
  encontrado = false;
  while ( ! xs.esFin() && ! encontrado ) {
    x = xs.actual();
    if ( propiedad(x) )
      encontrado = true;
    else
      xs.avanza();
  }
// Post: Determina si algún elemento de la secuencia cumple 'propiedad'
//       Si no lo cumple ninguno, entonces el punto de interés queda
//       al final de la secuencia, si no
//       el punto de interés queda en el primer elemento que lo cumple
}
```

Comentarios:

– Como variante de este esquema, podríamos no exigir en la precondición que el punto de interés estuviese al principio, con lo que la búsqueda se realizaría desde la posición del punto de interés hasta el final de la secuencia.

– Este esquema se puede utilizar tanto para encontrar la primera aparición de un cierto elemento en una secuencia como las apariciones sucesivas, si en cada caso el punto de interés se queda en la posición siguiente a la última aparición encontrada.