Capítulo 1

Diseño e implementación de TADs lineales[1]

Empieza por el principio –dijo el Rey con gravedad – y sigue hasta llegar al final; allí te paras.

Lewis Carroll definiendo, sin pretenderlo, el recorrido de una estructural lineal en Alicia en el país de las maravillas.

Resumen: En este tema se presentan los TADs lineales, dando al menos una implementación de cada uno de ellos. Se presenta también el concepto de iterador que permite recorrer una colección de objetos y se extiende el TAD lista para que soporte recorrido y modificación mediante iteradores.

1. Motivación

El agente 0069 ha inventado un nuevo método de codificación de mensajes secretos. El mensaje original X se codifica en dos etapas:

1. X se transforma en X' reemplazando cada sucesión de caracteres consecutivos que no sean vocales por su imagen especular.
2. X' se transforma en la sucesión de caracteres X'' obtenida al ir tomando sucesivamente: el primer carácter de X', luego el último, luego el segundo, luego el penúltimo, etc.

Ejemplo: para X = "Bond, James Bond", resultan:
X' = "BoJ ,dnameB sodn"
y
X'' = "BnodJo s, dBneam"
¿Serías capaz de implementar los algoritmos de codificación y decodificación?

[1] Marco Antonio Gómez es el autor principal de este tema.

Apostamos que sí; inténtalo. A buen seguro te dedicarás a utilizar vectores de caracteres y enteros a modo de "índice" a los mismos. En este tema aprenderás a hacerlo de una forma mucho más fácil gracias a los TADs lineales. Al final del tema vuelve a implementarlo y compara las dos soluciones.

2. Estructuras de datos lineales

Antes de plantearnos los distintos tipos abstractos de datos lineales nos planteamos cómo podemos guardar en memoria una colección de datos lineal. Hay dos aproximaciones básicas:

- Todos los elementos de forma consecutiva en la memoria: vector de elementos.
- Elementos dispersos en memoria guardando enlaces entre ellos: listas enlazadas.

Cada una de las alternativas tiene sus ventajas y desventajas. Las implementaciones de los TADs lineales que veremos en el tema podrán hacer uso de una u otra estructura de datos; la elección de una u otra podrá influir en la complejidad de sus operaciones.

Veamos cada una de ellas. Es importante hacer notar que estamos hablando aquí de *estructuras de datos* o estrategias para almacenar información en memoria. Por eso *no* planteamos de forma exhaustiva qué operaciones vamos a tener, ni el invariante de la representación ni relación de equivalencia. Introducimos aquí simplemente los métodos típicos que las implementaciones de los TADs que hacen uso de estas estructuras de datos suelen incorporar para el manejo de la propia estructura.

2.1. Vectores de elementos

La idea fundamental es guardar todos los elementos en un vector/array utilizando el tipo primitivo del lenguaje. Dado que un vector *no* puede cambiar de tamaño una vez creado, se impone desde el momento de la creación un *límite* en el número de elementos que se podrán almacenar; de ellos solo los n primeros tendrán información útil (el resto debe verse como espacio reservado para almacenar otros elementos en el futuro).

Para superar la limitación del tamaño fijo es habitual hacer uso de *vectores dinámicos*: se crea un *array* en la memoria dinámica capaz de albergar un número fijo de elementos; cuando el vector se llena se construye un nuevo *array* más grande, se copian los elementos y se elimina el antiguo.

Definición de tipos

Las implementaciones que quieran hacer uso de esta estructura de datos utilizan normalmente tres atributos:

- Puntero al array almacenado en memoria dinámica.
- Tamaño de ese array (o lo que es lo mismo, número de elementos que podría almacenar como máximo).
- Número de elementos ocupados actualmente. Los índices ocupados casi siempre se *condensan* al principio del array.

```
template <class T>
class VectorDinamico {
public:

   ...

private:

   /** Puntero al array que contiene los datos. */
   T *_v;

   /** Tamaño del vector _v. */
   unsigned int _tam;

   /** Número de elementos reales guardados. */
   unsigned int _numElems;
};
```

Creación

La creación consiste en crear un vector con un tamaño inicial. En el código siguiente ese tamaño (definido en una constante) es 10.

```
template <class T>
class VectorDinamico {
public:

   /** Tamaño inicial del vector dinámico. */
   enum { TAM_INICIAL = 10 };

   /** Constructor */
   VectorDinamico() :
           _v(new T[TAM_INICIAL]),
      _tam(TAM_INICIAL),
      _numElems(0) {
   }

   ...

};
```

Operaciones sobre la estructura de datos

Las operaciones relevantes en esta estructura de datos son:

- Método para ampliar el vector: cuando el TAD quiera añadir un nuevo elemento en el vector pero esté ya lleno debe crear un vector nuevo. Para que el coste amortizado de las inserciones sea constante el tamaño del vector *se dobla*[2].

[2]El coste amortizado se utiliza cuando una función presenta costes muy distintos en distintas llamadas y se quiere obtener un coste más preciso que el caso peor de la función. En ese caso se calcula el caso peor de una secuencia de llamadas a la función. Decir que una función requiere un tiempo amortizado constante significa que para cualquier secuencia de n llamadas, el tiempo total de la secuencia está acotado superiormente por cn, para una cierta constante $c > 0$. Se permite por tanto un tiempo excesivo para una

- Al eliminar un elemento intermedio del vector hay que *desplazar* los elementos que quedan a la derecha del eliminado.

```
template <class T>
class VectorDinamico {

   ...

protected:

   /**
    Duplica el tamaño del vector.
    */
   void amplia() {
      T *viejo = _v;
      _tam *= 2;
      _v = new T[_tam];

      for (unsigned int i = 0; i < _numElems; ++i)
         _v[i] = viejo[i];

      delete []viejo;
   }

   /**
    Elimina un elemento del vector; compacta los elementos
    al principio del vector.
    @param pos En el intervalo 0..numElems-1.
    */
   void quitaElem(int pos) {
      assert((0 <= pos) && (pos < _numElems));

      --_numElems;
      for (int i = pos; i < _numElems; ++i)
         _v[i] = _v[i+1];
   }

};
```

Destrucción

La destrucción requiere simplemente eliminar el vector creado en el constructor o en el método `amplia()`.

```
template <class T>
class VectorDinamico {
public:
   ...

   ~VectorDinamico() {
      delete []_v;
```

llamada sólo si se han registrado tiempos muy breves anteriormente. En el caso de la inserción, el vector se dobla tras n inserciones de coste $\mathcal{O}(1)$, por lo que el coste de la secuencia sería $n * \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$. Puede entonces considerarse que el coste amortizado de cada insercion esta en $\mathcal{O}(1)$.

```
    }
    ...
};
```

2.2. Listas enlazadas

En este caso cada elemento es almacenado en un espacio de memoria independiente (un *nodo*) y la colección completa se mantiene utilizando punteros. Hay dos alternativas:

- Listas enlazadas simples (o listas simplemente enlazadas): cada nodo mantiene un puntero al siguiente elemento.
- Listas doblemente enlazadas: cada nodo mantiene dos punteros: el puntero al nodo siguiente y al nodo anterior.

Aquí aparece la implementación de esta segunda opción, por ser más versátil. No obstante en ciertas implementaciones de TADs esa versatilidad no aporta ventajas adicionales (por ejemplo en las pilas), por lo que sería más eficiente (en cuanto a consumo de memoria) el uso de listas enlazadas.

Definición de tipos

Dependiendo del TAD que utilice esta estructura de datos, sus atributos serán distintos. Todas las implementaciones tendrán, eso sí, la definición de la clase `Nodo` que es la que almacena por un lado el elemento y por otro lado los punteros al nodo siguiente y al nodo anterior. Esa clase en C++ la implementaremos como una clase interna.

```
template <class T>
class ListaEnlazada {
public:

    ...

private:

    /**
     Clase nodo que almacena internamente el elemento (de tipo T),
     y dos punteros, uno al nodo anterior y otro al nodo siguiente.
     Ambos punteros podrían ser NULL si el nodo es el primero
     y/o último de la lista enlazada.
     */
    class Nodo {
    public:
        Nodo() : _sig(NULL), _ant(NULL) {}
        Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}
        Nodo(Nodo *ant, const T &elem, Nodo *sig) :
            _elem(elem), _sig(sig), _ant(ant) {}

        T _elem;
        Nodo *_sig;
        Nodo *_ant;
    };
};
```

Creación

Dado que una lista vacía no tiene ningún nodo, el proceso de creación solo implica inicializar los atributos que apuntan al primero/último de la lista a NULL, para indicar la ausencia de elementos.

Operaciones sobre la estructura de datos

Hay dos operaciones: creación de un nuevo nodo y su inserción en la lista enlazada y eliminación.

- La inserción que implementaremos recibe dos punteros, uno al nodo anterior y otro al nodo siguiente al nodo nuevo a añadir; crea un nuevo nodo y lo devuelve. Notar que algunos de los punteros pueden ser NULL (cuando se añade un nuevo nodo al principio o al final de la lista enlazada).
- La operación de borrado recibe únicamente el nodo a eliminar. La implementación utiliza el propio nodo para averiguar cuáles son los nodos anterior y siguiente para modificar sus punteros. Notese que si estuvieramos implementado una lista enlazada (y no doblemente enlazada) la operación necesitaría recibir un puntero al nodo anterior.

```
template <class T>
class ListaEnlazada {

   ...

protected:

   /**
    Inserta un elemento entre el nodo1 y el nodo2.
    Devuelve el puntero al nodo creado.
    Caso general: los dos nodos existen.
       nodo1->_sig == nodo2
       nodo2->_ant == nodo1
    Casos especiales: alguno de los nodos no existe
       nodo1 == NULL y/o nodo2 == NULL
   */
   static Nodo *insertaElem(const T &e, Nodo *nodo1, Nodo *nodo2) {
      Nodo *nuevo = new Nodo(nodo1, e, nodo2);
      if (nodo1 != NULL)
         nodo1->_sig = nuevo;
      if (nodo2 != NULL)
         nodo2->_ant = nuevo;
      return nuevo;
   }

   /**
    Elimina el nodo n. Si el nodo tiene nodos antes
    o después, actualiza sus punteros anterior y siguiente.
    Caso general: hay nodos anterior y siguiente.
    Casos especiales: algunos de los nodos (anterior o siguiente
    a n) no existen.
   */
```

```
   static void borraElem(Nodo *n) {
      assert(n != NULL);
      Nodo *ant = n->_ant;
      Nodo *sig = n->_sig;
      if (ant != NULL)
         ant->_sig = sig;
      if (sig != NULL)
         sig->_ant = ant;
      delete n;
   }

};
```

Destrucción

La destrucción requiere ir recorriendo uno a uno todos los nodos de la lista enlazada y eliminándolos.

```
template <class T>
class ListaEnlazada {

   ...

protected:
   /**
    Elimina todos los nodos de la lista enlazada cuyo
    primer nodo se pasa como parámetro.
    Se admite que el nodo sea NULL (no habrá nada que
    liberar). En caso de pasarse un nodo válido,
    su puntero al nodo anterior debe ser NULL (si no,
    no sería el primero de la lista!).
    */
   static void libera(Nodo *prim) {
      assert((prim == NULL) || (prim->_ant == NULL));

      while (prim != NULL) {
         Nodo *aux = prim;
         prim = prim->_sig;
         delete aux;
      }
   }
};
```

Con esto terminamos el análisis de las dos estructuras de datos que utilizaremos para guardar en memoria los elementos almacenados en los TADs lineales que veremos a lo largo del tema. Aunque en las explicaciones anteriores hemos hecho uso de las clases `VectorDinamico` y `ListaEnlazada`, en la práctica tendremos las clases que implementan los distintos TADs y que tendrán los atributos o clases internas y los métodos que hemos descrito aquí.

2.3. En el mundo real...

Las estructuras de datos que hemos visto en este apartado se utilizarán en la implementación de los TADs que veremos a continuación. Lo mismo ocurre en las librerías de colecciones de lenguajes como C++ o Java. Nosotros no nos preocuparemos de la reutilización aquí, por lo que el código de gestión de estas estructuras estará repetido en todas las implementaciones de los TADs que veamos. En una implementación seria esta aproximación sería inadmisible. Por poner un ejemplo de diseño correcto, la librería de C++ tiene implementadas estas dos estructuras de datos a las que llama *contenedores* (son la clase `std::vector` y `std::list`). Las implementaciones de los distintos TADs son después parametrizadas con el tipo de contenedor que se quiere utilizar. Dependiendo de la elección, la complejidad de cada operación variará.

Desde el punto de vista de la eficiencia de las estructuras de datos el vector dinámico tiene un comportamiento que no querríamos en un desarrollo serio. En concreto, la inocente:

```
_v = new T[_tam];
```

que aparece en el constructor lo que provoca es la llamada al constructor de la clase `T` base, de forma que cuando se construye un vector dinámico *vacío* se *crean* un puñado de elementos. Utilizando técnicas de C++ se puede retrasar la invocación a esos constructores hasta el momento de la inserción.

Peor aún es el momento del borrado de un elemento: cuando se elimina un elemento se desplaza el resto una posición hacia la izquierda pero *ese desplazamiento se realiza mediante copia*, por lo que el último elemento del vector queda *duplicado*, y no se destruirá hasta que no se elimine por completo el vector en el

```
delete []_v;
```

La última pega de los vectores dinámicos es el consumo de memoria. Los vectores crecen indefinidamente, nunca decrecen. Si un vector almacena puntualmente muchos elementos pero después se suprimen todos ellos el consumo de memoria no disminuye[3]. En la librería de C++ existe otro tipo de contenedor (`std::dequeue`) que no sufre este problema.

Por último, en nuestra implementación (y en las implementaciones de los TADs que veremos en las secciones siguientes) hemos obviado métodos recomendables (e incluso necesarias) en las clases de C++ como el constructor por copia, operaciones de igualdad, etc. En las implementaciones proporcionadas como material adicional a estos apuntes aparecerán implementados todos esos métodos, pero no los describiremos aquí por entender que son detalles de C++ no relevantes para la parte de teoría.

3. Pilas

Una pila representa una colección de valores donde es posible acceder al último elemento añadido, implementando la idea intuitiva de *pila* de objetos.

El TAD pila tiene dos operaciones generadoras: la que crea una pila vacía y la que apila un nuevo elemento en una pila dada. Tiene además una operación modificadora que permite desapilar el último objeto (y que es parcial, pues si la pila está ya vacía falla) y al menos dos operaciones observadoras: la que permite acceder al último elemento añadido (también parcial) y la que permite averiguar si una pila tiene elementos.

Las pilas tienen muchas utilidades, como por ejemplo "dar la vuelta" a una secuencia de datos (ver ejercicio 1).

[3] Esta desventaja, no obstante, es un problema de nuestra implementación; para solucionarlo, la solución basta con hacer que cuando el vector pierde un número suficiente de elementos, su tamaño se reduce automáticamente en una función inversa a `amplia()`. Si se hace un análisis del coste amortizado similar al utilizado para la inserción se llegaría a la misma conclusión: las operaciones siguen siendo $\mathcal{O}(1)$.

3.1. Implementación de pilas con vector dinámico

En esta implementación se utiliza como estructura de datos un vector dinámico que almacena en sus primeras posiciones los elementos almacenados en la pila y que duplica su tamaño cuando se llena (según lo visto en el apartado 2.1).

Por lo tanto, el tipo representante tendrá tres atributos: el puntero al vector, el número de elementos almacenados y el tamaño máximo del vector que ya utilizamos en la sección 2.1 llamados `_v`, `_numElems` y `_tam`. Notar que `_v[_numElems-1]` es el elemento de la cima de la pila.

El *invariante de la representación*, o lo que es lo mismo las condiciones que deben cumplir los objetos de la clase para considerarse válidos, para una pila `p` cuyos elementos son de tipo `T` es:

$$
\begin{aligned}
& R_{Pila_T}(\mathsf{p}) \\
\Longleftrightarrow_{def} & \\
& 0 \leq \mathsf{p.\_numElems} \leq \mathsf{p.\_tam} \wedge \\
& \forall i : 0 \leq i < \mathsf{\_numElems} : R_T(\mathsf{p.\_v[i]})
\end{aligned}
$$

También es lícito plantearnos cuándo dos objetos que utilizan la misma implementación representan a pilas idénticas. Es lo que se llama relación de equivalencia que permite averiguar si dos objetos válidos (es decir, que cumplen el invariante de la representación) son iguales. En este caso concreto dos pilas son iguales si el número de elementos almacenados coincide y sus valores respectivos, uno a uno, también:

$$
\begin{aligned}
& \mathsf{p1} \equiv_{Pila_T} \mathsf{p2} \\
\Longleftrightarrow_{def} & \\
& \mathsf{p1.\_numElems} = \mathsf{p2.\_numElems} \wedge \\
& \forall i : 0 \leq i < \mathsf{p1.\_numElems} : \mathsf{p1.\_v[i]} \equiv_T \mathsf{p2.\_v[i]}
\end{aligned}
$$

La implementación es sencilla basándonos en los métodos vistos anteriormente[4]:

```
/** Excepción generada por algunos métodos. */
class EPilaVacia {};

/**
 Implementación del TAD Pila utilizando vectores dinámicos.

 Las operaciones son:

 - PilaVacia: -> Pila. Generadora implementada en el
   constructor sin parámetros.
 - Apila: Pila, Elem -> Pila. Generadora
 - desapila: Pila - -> Pila. Modificadora parcial.
 - cima: Pila - -> Elem. Observadora parcial.
 - esVacia: Pila -> Bool. Observadora.
 */
template <class T>
```

[4] Durante todo el tema utilizaremos métodos que ya han aparecido anteriormente; en una implementación *desde cero* es posible que el código de algunos de esos métodos apareciera directamente integrado en las operaciones en vez de utilizar un método auxiliar.

```
class Pila {
public:

   /** Constructor; operación PilaVacia */
   Pila() : _v(new T[TAM_INICIAL]), _tam(TAM_INICIAL), _numElems(0) {
   }

   /** Destructor; elimina el vector. */
   ~Pila() {
      delete []_v;
   }

   /**
    Apila un elemento. Operación generadora.

    @param elem Elemento a apilar.
   */
   void apila(const T &elem) {
      _v[_numElems] = elem;
      _numElems++;
      if (_numElems == _tam)
         amplia();
   }

   /**
    Desapila un elemento. Operación modificadora parcial,
    que falla si la pila está vacía.

    desapila(Apila(elem, p)) = p
    error: desapila(PilaVacia)
   */
   void desapila() {
      if (esVacia())
         throw EPilaVacia();
      --_numElems;
   }

   /**
    Devuelve el elemento en la cima de la pila. Operación
    observadora parcial, que falla si la pila está vacía.

    cima(Apila(elem, p) = elem
    error: cima(PilaVacia)

    @return Elemento en la cima de la pila.
    */
   const T &cima() const {
      if (esVacia())
         throw EPilaVacia();
      return _v[_numElems - 1];
   }

   /**
    Devuelve true si la pila no tiene ningún elemento.

    esVacia(PilaVacia) = true
```

```
    esVacia(Apila(elem, p)) = false

    @return true si la pila no tiene ningún elemento.
    */
   bool esVacia() const {
      return _numElems == 0;
   }

private:
   ...
};
```

3.2. Implementación de pilas con una lista enlazada

La implementación con listas enlazadas consiste en almacenar como primer elemento de la lista el que aparece en la *cima* de la pila. La base de la pila se guarda en el último elemento de la lista. De esta forma:

- La clase `Pila` necesita un único atributo: un puntero al nodo que contiene la cima (`_cima`). Si la pila está vacía, el puntero valdrá `NULL`.
- Dado que lo único que hacemos con la lista es insertar y borrar el primer elemento las listas enlazadas simples son suficiente.

El invariante debe garantizar que la secuencia de nodos termina en `NULL` (eso garantiza que no hay ciclos) y que todos los nodos deben estar correctamente ubicados y almacenar un elemento del tipo base válido:

$$
\begin{aligned}
& R_{Pila_T}(\mathsf{p}) \\
\Longleftrightarrow_{def} & \\
& \mathsf{null} \in cadena(\mathsf{p.\_cima}) \land \\
& \forall \mathsf{n} \in cadena(\mathsf{p.\_cima}) : \mathsf{n} \neq \mathsf{null} \rightarrow (\mathsf{ubicado(n)} \land R_T(\mathsf{n.\_elem}))
\end{aligned}
$$

Donde ubicado es un predicado que viene a asegurar que se ha pedido memoria para el puntero (y ésta no ha sido liberada) y $cadena(\mathsf{ptr})$ representa el conjunto de todos los nodos que forman la lista enlazada que comienza en ptr, incluido el posible "nodo nulo" representado por el valor null:

$$
\begin{array}{ll}
cadena(\mathsf{ptr}) = \{\mathsf{null}\} & \textbf{si} \quad \mathsf{ptr} = \mathsf{null} \\
cadena(\mathsf{ptr}) = \{\mathsf{ptr}\} \bigcup cadena(\mathsf{ptr.\_sig}) & \textbf{si} \quad \mathsf{ptr} \neq \mathsf{null}
\end{array}
$$

Tras el invariante, definimos la relación de equivalencia en la implementación: dos objetos pila serán iguales si su lista enlazada contiene el mismo número de elementos y sus valores uno a uno coinciden (están en el mismo orden):

$$
\begin{aligned}
& \mathsf{p1} \equiv_{Pila_T} \mathsf{p2} \\
\Longleftrightarrow_{def} & \\
& iguales_T(\mathsf{p1.\_cima}, \mathsf{p2.\_cima})
\end{aligned}
$$

donde

$$
\begin{array}{lll}
iguales(\mathsf{ptr1}, \mathsf{ptr2}) = \mathsf{true} & \mathbf{si} & \mathsf{ptr1} = \mathsf{null} \land \mathsf{ptr2} = \mathsf{null} \\
iguales(\mathsf{ptr1}, \mathsf{ptr2}) = \mathsf{false} & \mathbf{si} & (\mathsf{ptr1} = \mathsf{null} \land \mathsf{ptr2} \neq \mathsf{null}) \lor \\
 & & (\mathsf{ptr1} \neq \mathsf{null} \land \mathsf{ptr2} = \mathsf{null}) \\
iguales(\mathsf{ptr1}, \mathsf{ptr2}) = \mathsf{ptr1.\_elem} \equiv_T \mathsf{ptr2.\_elem} \land & & \\
\qquad iguales(\mathsf{ptr1.\_sig}, \mathsf{ptr2.\_sig}) & \mathbf{si} & (\mathsf{ptr1} \neq \mathsf{null} \land \mathsf{ptr2} \neq \mathsf{null})
\end{array}
$$

La implementación aparece a continuación; el nombre de la clase lo hemos cambiado a `PilaLE` (pila implementada con ***l***istas ***e***nlazadas). En aras de la simplicidad omitimos los comentarios de los métodos (al implementar el mismo TAD y ser una implementación sin limitaciones, coinciden con los de la implementación con vectores dinámicos):

```
template <class T>
class PilaLE {
public:

   PilaLE() : _cima(NULL) {}

   ~PilaLE() {
      libera(_cima);
      _cima = NULL;
   }

   void apila(const T &elem) {
      _cima = insertaElem(elem, NULL, _cima);
   }

   void desapila() {
      if (esVacia())
         throw EPilaVacia;
      Nodo *aBorrar = _cima;
      _cima = _cima->_sig;
      borraElem(aBorrar);
   }

   const T &cima() const {
      if (esVacia())
         throw EPilaVacia;
      return _cima->_elem;
   }

   void esVacia() const {
      return _cima == NULL;
   }

private:
   ...

   Nodo *_cima;
};
```

La complejidad de las operaciones de ambas implementaciones es similar[5]:

[5]En todas las tablas de complejidades de operaciones de TADs que pondremos *asumiremos* que el tipo base tiene operaciones de construcción, destrucción y copia constantes $\mathcal{O}(1)$.

Operación	Vectores	Listas enlazadas
`Pila`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`apila`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`desapila`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`cima`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`esVacia`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

4. Colas

Las colas son TADs lineales que permiten introducir elementos por un extremo (el *final* de la cola) y las consultas y eliminaciones por el otro (el *inicio* de la cola). Es decir son estructuras en las que el primer elemento que entra es el primero que saldrá; de ahí que también se las conozca como estructuras FIFO (del inglés, *first in, first out*).

Las operaciones de las colas son:

- `ColaVacia`: genera una cola vacía.
- `PonDetras`: añade un nuevo elemento a la cola.
- `quitaPrim`: modificadora parcial que elimina el primer elemento de la cola. Falla si la cola está vacía.
- `primero`: observadora parcial que devuelve el primer elemento de la cola (el más antiguo).
- `esVacia`: también observadora, permite averiguar si la cola tiene elementos.

4.1. Implementación de colas con un vector

Una posible implementación con un vector dinámico consistiría en hacer crecer el vector al ir llegando elementos, de forma que el primer elemento de la cola esté siempre en la posición 0 del vector.

La principal pega que tiene esa implementación es el coste de la operación `quitaPrim`: al eliminar el elemento debemos desplazar todos los elementos válidos una posición a la izquierda, elevando el coste de la operación a $\mathcal{O}(n)$.

El invariante de la representación y la relación de equivalencia de esta implementación es análoga a aquella vista para las pilas.

Existe una implementación sobre vectores más eficiente en la que la complejidad de la operación es $\mathcal{O}(1)$; ver el ejercicio 12.

4.2. Implementación de colas con una lista enlazada

Otra posible implementación de las colas es utilizando un esquema similar a aquél visto en las pilas: una lista enlazada en la que el primer nodo contiene el elemento que hay en la cabecera de la cola. Igual que ocurría en las pilas esa lista puede ser simple para ahorrarnos el espacio en memoria ocupado por los punteros a los nodos anteriores que no necesitamos.

Si hiciéramos la implementación nos daríamos cuenta de que la operación `PonDetras` tiene complejidad $\mathcal{O}(n)$ pues debemos recorrer todos los elementos almacenados en la lista para saber dónde colocar el nuevo nodo.

La forma de solucionarlo es hacer que la implementación almacene *dos punteros*: un puntero al primer nodo y otro puntero al último nodo.

El invariante de la representación es similar al visto en la implementación de las pilas, y la relación de equivalencia también.

La implementación es en cierto sentido algo incómoda pues tenemos que tener cuidado con los casos especiales en los que trabajamos con una cola vacía[6].

```
/** Excepción generada por algunos métodos. */
class EColaVacia {};

/**
 Implementación del TAD Cola utilizando una lista enlazada.

 Las operaciones son:

 - ColaVacia: -> Cola. Generadora implementada en el
   constructor sin parámetros.
 - PonDetras: Cola, Elem -> Cola. Generadora
 - quitaPrim: Cola - -> Cola. Modificadora parcial.
 - primero: Cola - -> Elem. Observadora parcial.
 - esVacia: Cola -> Bool. Observadora.
 */
template <class T>
class Cola {
public:

   /** Constructor; operacion ColaVacia */
   Cola() : _prim(NULL), _ult(NULL) {}

   /** Destructor; elimina la lista enlazada. */
   ~Cola() {
      libera(_prim);
      _prim = _ult = NULL;
   }

   /**
    Añade un elemento en la parte trasera de la cola.
    Operación generadora.

    @param elem Elemento a añadir.
   */
   void ponDetras(const T &e) {
      _ult = insertaElem(e, _ult, NULL);
      if (_prim == NULL)
         _prim = _ult;
   }

   /**
    Devuelve el primer elemento de la cola. Operación
    observadora parcial, que falla si la cola está vacía.

    primero(PonDetras(elem, ColaVacia)) = elem
    primero(PonDetras(elem, xs)) = primero(xs) si !esVacia(xs)

    @return El primer elemento de la cola.
```

[6]En el código no se aprecian todos los casos especiales gracias a las operaciones auxiliares implementadas en la sección 2.2 y que no mostramos de nuevo aquí.

```
    */
    const T &primero() const {
        if (esVacia())
            throw EColaVacia();
        return _prim->_elem;
    }

    /**
     Elimina el primer elemento de la cola.
     Operación modificadora parcial, que falla si
     la cola está vacía.

     quitaPrim(PonDetras(elem, ColaVacia)) = ColaVacia
     quitaPrim(PonDetras(elem, xs)) =
             PonDetras(elem, quitaPrim(xs)) si !esVacia(xs)
     error: quitaPrim(ColaVacia)
    */
    void quitaPrim() {
        if (esVacia())
            throw EColaVacia();

        Nodo *aBorrar = _prim;
        _prim = aBorrar->_sig;
        borraElem(aBorrar);
        if (_prim == NULL)
            _ult = NULL;
    }

    /**
     Devuelve true si la cola no tiene ningún elemento.

     esVacia(Cola) = true
     esVacia(PonDetras(elem, p)) = false

     @return true si la cola no tiene ningún elemento.
     */
    bool esVacia() const {
        return _prim == NULL;
    }


private:
    ...

    // Puntero al primer y último elemento
    Nodo *_prim, *_ult;
};
```

4.3. Implementación de colas con una lista enlazada y nodo fantasma

Para evitar los casos especiales que teníamos antes en los que había que tener en cuenta que la cola podía estar o no vacía se puede utilizar un *nodo fantasma*: un nodo extra al principio de la lista enlazada que hace de *cabecera* especial que siempre tenemos, que no guarda ningún elemento, pero nos permite no tener que distinguir el caso en el que `_prim` es `NULL` (cola vacía). La complejidad de las operaciones no varía, pero su programación es más sencilla; puedes comprobarlo realizando la implementación. La misma técnica la utilizaremos posteriormente para las colas dobles.

La complejidad de las operaciones es distinta dependiendo de la estructura utilizada:

Operación	Vectores	Vectores circulares	Listas enlazadas
`Cola`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`ponDetras`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`primero`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`quitaPrim`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`esVacia`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

5. Colas dobles

Las colas dobles son una generalización de las colas en donde se pueden añadir, quitar y consultar elementos en los dos extremos. En concreto las operaciones serán:

- `DColaVacia`: genera una cola doble vacía.
- `PonDetras`: añade un nuevo elemento al final.
- `ponDelante`: añade un nuevo elemento al principio.
- `quitaPrim`: modificadora parcial que elimina el primer elemento de la cola. Falla si la cola está vacía.
- `primero`: observadora parcial que devuelve el primer elemento de la cola (el más antiguo).
- `quitaUlt`: quita el último elemento; también modificadora parcial.
- `ultimo`: devuelve el último elemento de la cola. Igual que `primero` es observadora parcial.
- `esVacia`: también observadora, permite averiguar si la cola tiene elementos.

Dada su semejanza con las colas de la sección anterior existen opciones similares para la implementación, aunque en el caso de la implementación utilizando nodos, es preferible el uso de listas doblemente enlazadas frente a las simples.

No obstante las implementaciones requieren el doble de cuidado que las de el TAD hermano de las colas, pues necesitamos cubrir más casos especiales. Es por eso que la opción de implementación con una lista enlazada circular y nodo fantasma aparece como la mejor candidata.

En concreto la cola vacía estará representada por un nodo fantasma que no contiene ningún elemento y cuyos punteros anterior y siguiente apuntan a él mismo. Ese nodo fantasma, que se crea al principio y no se borrará hasta el final, es apuntado por el único atributo de la clase. La implementación hará que *el siguiente* al nodo fantasma sea el primero de la cola (la cabecera), mientras que el `anterior` será el último.

La relación de equivalencia no es muy diferente de la vista para las pilas y colas; el único cambio es la condición que marca el final de la recursión en el predicado *iguales* que utilizábamos entonces. Ahora no debemos terminar cuando se llega al `null`, sino cuando volvemos al nodo de partida. Para eso extendemos los parámetros de *iguales* para que además de tener los nodos de partida tenga también como parámetros los nodos donde la recursión debe terminar.

$$
\begin{aligned}
& \mathsf{c1} \equiv_{DCola_T} \mathsf{c2} \\
\Longleftrightarrow_{def} & \\
& iguales_T(\\
& \quad \mathsf{c1.\_fantasma.\_sig}, \mathsf{c1.\_fantasma.\_fin}, \\
& \quad \mathsf{c2.\_fantasma.\_sig}, \mathsf{c2.\_fantasma.\_fin} \\
&)
\end{aligned}
$$

donde

$$
\begin{array}{lll}
iguales(\mathsf{i1},\mathsf{e1},\mathsf{i2},\mathsf{e2}) = \mathsf{true} & \textbf{si} & \mathsf{i1} = \mathsf{e1} \wedge \mathsf{i2} = \mathsf{e2} \\
iguales(\mathsf{i1},\mathsf{e1},\mathsf{i2},\mathsf{e2}) = \mathsf{false} & \textbf{si} & (\mathsf{i1} = \mathsf{e1} \wedge \mathsf{i2} \neq \mathsf{e2}) \vee \\
 & & (\mathsf{i1} \neq \mathsf{e1} \wedge \mathsf{i2} = \mathsf{e2}) \\
iguales(\mathsf{i1},\mathsf{e1},\mathsf{i2},\mathsf{e2}) = \mathsf{i1.\_elem} \equiv_T \mathsf{i2.\_elem} \wedge & & \\
\qquad iguales(\mathsf{i1.\_sig},\mathsf{e1},\mathsf{i2.\_sig},\mathsf{e2}) & \textbf{si} & (\mathsf{i1} \neq \mathsf{e1} \wedge \mathsf{i2} \neq \mathsf{e2})
\end{array}
$$

Por su parte el invariante de la representación tiene que incorporar también la circularidad de la lista. Para la definición del invariante utilizamos un conjunto similar al *cadena* utilizado en las pilas pero que indica qué elementos son alcanzables desde un nodo dado si seguimos su puntero `_sig` por un lado y `_ant` por otro. Es claro que:

- El conjunto de nodos alcanzables desde el nodo cabecera por un lado y por otro debe ser el mismo.
- Dado que la lista es circular, el nodo cabecera debe aparecer en el conjunto de nodos alcanzables a partir de él.
- Todos esos nodos deben estar ubicados y tener los enlaces al nodo anterior y al nodo siguiente correctos (lo que implica que si vamos al nodo anterior de n y luego pasamos a su siguiente deberíamos volver a n y al contrario).
- Por último, todos los nodos (excepto el nodo cabecera) deben contener elementos válidos del tipo base.

Con esta idea, el invariante de la representación queda (por comodidad en las definiciones siguientes se entenderá ini como $\mathsf{c.\_fantasma}$):

$$
\begin{aligned}
& R_{DCola_T}(\mathsf{c}) \\
\Longleftrightarrow_{def} & \\
& alcanzables(\mathsf{ini}) = alcanzablesHaciaAtras(\mathsf{ini}) \wedge \\
& \mathsf{ini} \in alcanzables(\mathsf{ini}) \wedge \\
& \forall \mathsf{p} \in alcanzables(\mathsf{ini}) : \\
& \quad (\\
& \quad \mathsf{ubicado(p)} \wedge buenEnlace(\mathsf{p}) \wedge \\
& \quad (\mathsf{p} \neq \mathsf{ini} \rightarrow R_T(\mathsf{p.\_elem})) \\
& \quad)
\end{aligned}
$$

Donde, como hemos dicho antes, *alcanzables* es el conjunto de todos los nodos que pueden alcanzarse desde un nodo dado utilizando el puntero `_sig` y *alcanzablesHaciaAtras* utilizando `_ant`. Por último, *buenEnlace* indica si los punteros son correctos desde el punto de vista de una lista doblemente enlazada:

$$\begin{array}{ll} alcanzables(\mathsf{p}) = \emptyset & \textbf{si}\ \ \mathsf{p} = \mathsf{null} \\ alcanzables(\mathsf{p}) = \{\mathsf{p.\_sig}\} \bigcup alcanzables(\mathsf{p.\_sig}) & \textbf{si}\ \ \mathsf{p} \neq \mathsf{null} \end{array}$$

$$\begin{array}{ll} alcanzablesHaciaAtras(\mathsf{p}) = \emptyset & \textbf{si}\ \ \mathsf{p} = \mathsf{null} \\ alcanzablesHaciaAtras(\mathsf{p}) = \{\mathsf{p.\_ant}\} \bigcup alcanzablesHaciaAtras(\mathsf{p.\_ant}) & \textbf{si}\ \ \mathsf{p} \neq \mathsf{null} \end{array}$$

$$buenEnlace(\mathsf{p}) = \mathsf{p.\_sig.\_ant} = \mathsf{p} \wedge \mathsf{p.\_ant.\_sig} = \mathsf{p}$$

Observa que la implementación crea en el momento de su construcción el nodo "fantasma" y que en la destrucción se rompe la circularidad de la lista para poder utilizar `libera` sin riesgo a entrar en bucles infinitos. Por otro lado verás que la implementación de las operaciones no tienen que preocuparse de casos especiales (más allá de las precondiciones de `esVacia`), haciendo la implementación casi trivial:

```
/** Excepción generada por algunos métodos. */
class EDColaVacia {};

/**
 Implementación del TAD Doble Cola utilizando una lista doblemente
 enlazada circular y con nodo fantasma.

 Las operaciones son:

 - DColaVacia: -> DCola. Generadora implementada en el
   constructor sin parámetros.
 - PonDetras: DCola, Elem -> DCola. Generadora
 - ponDelante: DCola, Elem -> DCola. Modificadora.
 - quitaPrim: DCola - -> DCola. Modificadora parcial
 - primero: DCola - -> Elem. Observadora parcial
 - quitaUlt: DCola - -> DCola. Modificadora parcial
 - ultimo: DCola - -> Elem. Observadora parcial
 - esVacia: DCola -> Bool. Observadora

 */
template <class T>
class DCola {
public:

   /** Constructor; operación DColaVacia. */
   DCola() {
      _fantasma = new Nodo();
      _fantasma->_sig = _fantasma;
      _fantasma->_ant = _fantasma;
   }

   /** Destructor; elimina la lista doblemente enlazada. */
   ~DCola() {
```

```
    // Usamos libera, pero antes quitamos
    // la circularidad para evitar bucle
    // infinito...
    _fantasma->_ant->_sig = NULL;
    _fantasma->_ant = NULL;
    libera(_fantasma);
    _fantasma = NULL;
}

/**
 Añade un elemento por la parte de atrás de la cola.
 Es una operación generadora.
*/
void ponDetras(const T &e) {
    insertaElem(e, _fantasma->_ant, _fantasma);
}

/**
 Devuelve el primer elemento de la cola; es una operación
 observadora parcial, pues es un error preguntar por
 el primer elemento de una doble cola vacía.

 primero(PonDetras(elem, DColaVacia)) = elem
 primero(PonDetras(elem, xs)) = primero(xs) si !esVacia(xs)
 error: primero(DColaVacia)
 */
const T &primero() const {
    if (esVacia())
        throw EDColaVacia();
    return _fantasma->_sig->_elem;
}

/**
 Elimina el primer elemento de la doble cola.
 Operación modificadora parcial, que falla si
 está vacía.

 quitaPrim(PonDetras(elem, DColaVacia)) = DColaVacia
 quitaPrim(PonDetras(elem, xs)) = PonDetras(elem, quitaPrim(xs)) si !esVacia(xs)
 error: quitaPrim(DColaVacia)
*/
void quitaPrim() {
    if (esVacia())
        throw EDColaVacia();

    borraElem(_fantasma->_sig);
}

/**
 Añade un elemento a la parte delantera de una doble cola.
 Operación modificadora.

 ponDelante(elem, DColaVacia) = ponDetras(elem, DColaVacia)
 ponDelante(elem, ponDetras(x, xs)) =
   ponDetras(x, ponDelante(elem, xs))
```

```
     @param e Elemento que se añade
    */
   void ponDelante(const T &e) {
      insertaElem(e, _fantasma, _fantasma->_sig);
   }

   /**
    Devuelve el último elemento de la doble cola. Es
    un error preguntar por el último de una doble cola vacía.

    ultimo(PonDetras(x, xs)) = x
    error: ultimo(DColaVacia)

    @return Último elemento de la cola.
    */
   const T &ultimo() const {
      if (esVacia())
         throw EDColaVacia();
      return _fantasma->_ant->_elem;
   }

   /**
    Elimina el último elemento de la doble cola. Es
    un error quitar el último de una doble cola vacía.

    quitaUlt(PonDetras(x, xs)) = xs
    error: quitaUlt(DColaVacia)
    */
   void quitaUlt() {
      if (esVacia())
         throw EDColaVacia();

      borraElem(_fantasma->_ant);
   }

   /**
    Operación observadora para saber si una doble cola
    tiene o no elementos.

    esVacia(DColaVacia) = true
    esVacia(ponDetras(x, xs)) = false

    @return true si la doble cola no tiene elementos.
    */
   bool esVacia() const {
      return _fantasma->_sig == _fantasma;
   }

private:

   ...

   // Puntero al nodo fantasma
   Nodo *_fantasma;
};
```

La complejidad de las operaciones en esta implementación es:

Operación	Listas enlazadas
`DCola`	$\mathcal{O}(1)$
`ponDetras`	$\mathcal{O}(1)$
`primero`	$\mathcal{O}(1)$
`quitaPrim`	$\mathcal{O}(1)$
`ponDelante`	$\mathcal{O}(1)$
`ultimo`	$\mathcal{O}(1)$
`quitaUlt`	$\mathcal{O}(1)$
`esVacia`	$\mathcal{O}(1)$

6. Listas

Las listas son los TADs lineales más generales posibles[7]. Permiten la consulta y modificación de los dos extremos (como las colas dobles) pero también acceder a cualquier punto intermedio.

- Generadora `ListaVacia`. Construye la lista vacía.
- Generadora `Cons`. Añade un elemento en la cabeza de la lista ("parte izquierda" en las posibles figuras).
- Modificadora `ponDr`: Añade un elemento en la cola de la lista ("parte derecha").
- Observadora `primero`: devuelve el primer elemento (op. parcial).
- Modificadora `resto`: quita el primer elemento (op. parcial).
- Observadora `ultimo`: devuelve el último elemento (op. parcial).
- Modificadora `inicio`: quita el último elemento (op. parcial).
- Observadora `esVacia`: pues eso.
- Observadora `numElems`: pues eso.
- Observadora `elem`: devuelve el elemento i-ésimo de la lista (0..numElems-1).

Las listas son tan habituales que existe una notación muy utilizada mucho más compacta que la anterior. En ella símbolos como `x` o `y` representan *un elemento* de la lista, mientras que `xs` o `ys` representan listas:

```
ListaVacia          ≡ []
Cons(x, ListaVacia) ≡ [x]
Cons(x, xs)         ≡ [x|xs]
ponDr(xs, x)        ≡ [xs#x]
concatena(xs, ys)[8] ≡ xs++ys
```

[7] No confundir el TAD lista con la estructura de datos *lista enlazada*; las listas enlazadas deben verse como un método de organizar en memoria una colección de elementos; el TAD lista es un tipo abstracto de datos con una serie de operaciones que puede implementarse utilizando listas enlazadas pero también otro tipo de estructuras de datos, como los vectores dinámicos.

Igual que en los TADs anteriores podemos plantearnos tanto la implementación utilizando vectores como listas enlazadas. En la práctica la más utilizada son las listas enlazadas que si bien penaliza la operación *elem* ($\mathcal{O}(n)$) consigue una complejidad constante para el resto de operaciones[9].

En vez de utilizar una lista enlazada circular como la usada para la implementación de las colas dobles, utilizaremos una lista doblemente enlzada pero sin el nodo fantasma y no circular, es decir donde el puntero al anterior del primer elemento valdrá `NULL` y el siguiente al último también. El tipo representante guardará dos punteros, uno al primer nodo y otro al último. Además, para poder comprobar la precondición de la operación `elem` (que el índice pasado está en el intervalo válido dependiente del número de elementos añadidos), guardamos también un atributo `numElem` que mantenemos actualizado.

La función de abstracción y la relación de equivalencia de la implementación con listas doblemente enlazadas es similar a la descrita para las colas dobles pero eliminando la componente de circularidad. La implementación es:

```
/* Excepciones generadas por algunos métodos. */
class EListaVacia {};
class EAccesoInvalido {};

/**
 Implementación del TAD Pila utilizando vectores dinámicos.

 Las operaciones son:

 - ListaVacia: -> Lista. Generadora implementada en el
   constructor sin parámetros.
 - Cons: Lista, Elem -> Lista. Generadora.
 - ponDr: Lista, Elem -> Lista. Modificadora.
 - primero: Lista - -> Elem. Observadora parcial
 - resto: Lista - -> Lista. Modificadora parcial
 - ultimo: Lista - -> Elem. Observadora parcial
 - inicio: Lista - -> Lista. Modificadora parcial
 - esVacia: Lista -> Bool. Observadora
 - numElems: Lista -> Elem. Obervadora.
 - elem: Lista, Entero - -> Elem. Observador parcial.

 */
template <class T>
class Lista {
   ...

public:

   /** Constructor; operación ListaVacia. */
   Lista() : _prim(NULL), _ult(NULL), _numElems(0) {}

   /** Destructor; elimina la lista doblemente enlazada. */
   ~Lista() {
      libera(_prim);
      _prim = NULL;
      _ult = NULL;
```

[8] Ver ejercicio 15.

[9] Ver la sección siguiente para hacer recorridos de los elementos de la lista sin la penalización del coste de la operación `elem`.

```
}

/**
 Añade un nuevo elemento en la cabeza de la lista.
 Operación generadora.

 @param elem Elemento que se añade en la cabecera de
 la lista.
*/
void Cons(const T &elem) {
   _numElems++;
   _prim = insertaElem(elem, NULL, _prim);
   if (_ult == NULL)
      _ult = _prim;
}

/**
 Añade un nuevo elemento al final de la lista (a la
 "derecha"). Operación modificadora.

 ponDr(e, ListaVacia) = Cons(e, ListaVacia)
 ponDr(e, Cons(x, xs)) = Cons(x, ponDr(e, xs))
*/
void ponDr(const T &elem) {
   _numElems++;
   _ult = insertaElem(elem, _ult, NULL);
   if (_prim == NULL)
      _prim = _ult;
}

/**
 Devuelve el valor almacenado en la cabecera de la
 lista. Es un error preguntar por el primero de
 una lista vacía.

 primero(Cons(x, xs)) = x
 error primero(ListaVacia)

 @return Elemento en la cabecera de la lista.
 */
const T &primero() const {
   if (esVacia())
      throw EListaVacia();
   return _prim->_elem;
}

/**
 Devuelve el valor almacenado en la última posición
 de la lista (a la derecha).
 Es un error preguntar por el primero de una lista vacía.

 ultimo(Cons(x, xs)) = x            SI esVacia(xs)
 ultimo(Cons(x, xs)) = ultimo(xs)   SI !esVacia(xs)
 error ultimo(ListaVacia)

 @return Elemento en la cola de la lista.
```

```
 */
const T &ultimo() const {
   if (esVacia())
      throw EListaVacia();

   return _ult->_elem;
}

/**
 Elimina el primer elemento de la lista.
 Es un error intentar obtener el resto de una lista vacía.

 resto(Cons(x, xs)) = xs
 error resto(ListaVacia)
*/
void resto() {
   if (esVacia())
      throw EListaVacia();

   Nodo *aBorrar = _prim;
   _prim = _prim->_sig;
   borraElem(aBorrar);
   if (_prim == NULL)
      _ult = NULL;
   --_numElems;
}

/**
 Elimina el último elemento de la lista.
 Es un error intentar obtener el inicio de una lista vacía.

 inicio(Cons(x, ListaVacia)) = ListaVacia
 inicio(Cons(x, xs)) = Cons(x, inicio(xs)) SI !esVacia(xs)
 error inicio(ListaVacia)
*/
void inicio() {
   if (esVacia())
      throw EListaVacia();

   Nodo *aBorrar = _ult;
   _ult = _ult->_ant;
   borraElem(aBorrar);
   if (_ult == NULL)
      _prim = NULL;
   --_numElems;
}

/**
 Operación observadora para saber si una lista
 tiene o no elementos.

 esVacia(ListaVacia) = true
 esVacia(Cons(x, xs)) = false

 @return true si la lista no tiene elementos.
 */
```

```
   bool esVacia() const {
      return _prim == NULL;
   }

   /**
    Devuelve el número de elementos que hay en la
    lista.
    numElems(ListaVacia) = 0
    numElems(Cons(x, xs)) = 1 + numElems(xs)

    @return Número de elementos.
    */
   unsigned int numElems() const {
      return _numElems;
   }

   /**
    Devuelve el elemento i-ésimo de la lista, teniendo
    en cuenta que el primer elemento (primero())
    es el elemento 0 y el último es numElems()-1,
    es decir idx está en [0..numElems()-1].
    Operación observadora parcial que puede fallar
    si se da un índice incorrecto. El índice es
    entero sin signo, para evitar que se puedan
    pedir elementos negativos.

    elem(0, Cons(x, xs)) = x
    elem(n, Cons(x, xs)) = elem(n-1, xs) si n > 0
    error elem(n, xs) si !( 0 <= n < numElems(xs) )
   */
   const T &elem(unsigned int idx) const {
      if (idx >= _numElems)
         throw EAccesoInvalido();

      Nodo *aux = _prim;
      for (int i = 0; i < idx; ++i)
         aux = aux->_sig;

      return aux->_elem;
   }

   ...

private:
   // Puntero al primer y último elemento
   Nodo *_prim, *_ult;

   // Número de elementos (número de nodos entre _prim y _ult)
   unsigned int _numElems;
};
```

Dado que las listas son los TADs lineales más generales es posible desarrollar implementaciones del resto de TADs lineales basándose directamente en las listas. Ver el ejercicio 16.

La complejidad de las operaciones en esta implementación es:

Operación	Listas enlazadas
`lista`	$\mathcal{O}(1)$
`cons`	$\mathcal{O}(1)$
`primero`	$\mathcal{O}(1)$
`resto`	$\mathcal{O}(1)$
`ponDr`	$\mathcal{O}(1)$
`ultimo`	$\mathcal{O}(1)$
`inicio`	$\mathcal{O}(1)$
`elem`	$\mathcal{O}(n)$
`esVacia`	$\mathcal{O}(1)$

Como vemos, existe penalización en la implementación de `elem`. En la sección siguiente se describe una técnica para poder recorrer una lista en coste lineal.

7. Recorridos de listas

Un problema de las listas anteriores es que la complejidad de un bucle tan inocente como el siguiente:

```
Lista<int> l;

...

for (int i = 0; i < l.numElems; ++i)
   std::cout << l.elem(i) << endl;
```

que simplemente escribe uno a uno todos los elementos *no* tiene coste lineal sino cuadrático si utilizamos la implementación con listas enlazadas.

Una solución adoptada por muchas librerías de colecciones es utilizar el concepto de *iterador*. Entenderemos un iterador como un objeto de una clase que:

- Representa un punto intermedio en el recorrido de una colección de datos (una lista en este caso).
- Tiene un método `elem()` que devuelve el elemento por el que va el recorrido (y tendrá el tipo base utilizado en la colección). La operación será *parcial*, pues fallará si el recorrido ya ha terminado.
- Tiene un método `avanza()` que hace que el iterador pase al siguiente elemento del recorrido.
- Tiene implementada la operación de comparación, de forma que se puede saber si dos iteradores son iguales. Dos iteradores son iguales si: representan el mismo punto en el recorrido de una lista concreta o los dos representan el final del recorrido.

Extenderemos el TAD lista para que proporcione dos operaciones adicionales:

- `principio()`: devuelve un iterador inicializado al primer elemento del recorrido.
- `final()`: devuelve un iterador apuntando *fuera* del recorrido, es decir un iterador cuya operación `elem()` *falla*.

Haciendo que la operación `final()` devuelva un iterador *no válido* implica que los elementos válidos de la lista son el *intervalo abierto* [`principio()`, `final()`), y la forma de recorrer una lista será:

```
Lista<int> l;

...

Lista<int>::Iterador it = l.principio();
while (it != l.final()) {
   cout << it.elem() << endl;
   it.avanza();
}
```

7.1. Implementación de un iterador básico

La implementación se basa en la existencia de la clase interna `Iterador` que es devuelta por las operaciones `principio()` y `final()` y que tienen como atributo un puntero al nodo actual en el recorrido. La forma de especificar que el recorrido ya ha terminado es ponerlo a `NULL`.

```
template <class T>
class Lista {
public:

   ...

   class Iterador {
   public:
      void avanza() {
         if (_act == NULL) throw EAccesoInvalido();
         _act = _act->_sig;
      }

      const T &elem() const {
         if (_act == NULL) throw EAccesoInvalido();
         return _act->_elem;
      }

      bool operator==(const Iterador &other) const {
         return _act == other._act;
      }

      bool operator!=(const Iterador &other) const {
         return !(this->operator==(other));
      }
   protected:
      // Para que pueda construir objetos del
      // tipo iterador
      friend class Lista;

      Iterador() : _act(NULL) {}
      Iterador(Nodo *act) : _act(act) {}

      // Puntero al nodo actual del recorrido
```

```
      Nodo *_act;
   };

   Iterador principio() const {
      return Iterador(_prim);
   }

   Iterador final() const {
      return Iterador(NULL);
   }


   ...
};
```

7.2. Iteradores para modificar elementos

En el TAD lista puede tener sentido, además, permitir cambiar elementos utilizando los propios iteradores. De esta forma, podríamos extender la implementación del iterador anterior con una nueva operación `pon` que *cambie* el valor del elemento que se está recorriendo.

Para hacerlo, no obstante, crearemos una clase nueva, `IteradorMutable` que da la idea de que el iterador permite modificar la lista que está recorriendo. De esta forma, podemos asegurar que un usuario que haga uso de la clase `Iterador` descrita anteriormente recorrerá la lista únicamente "para lectura", sin modificarla y por lo tanto la operación `principio()` implementada anteriormente puede ser declarada `const` pues la lista no podrá verse modificada.

La nueva clase interna, `IteradorMutable` es una copia de la clase `Iterador` pero con un método nuevo:

```
class IteradorMutable {
public:

   ...

   void pon(const T &elem) {
      if (_act == NULL) throw AccesoInvalido();
      _act->_elem = elem;
   }

   ...
};
```

El iterador así extendido permite hacer recorridos que vayan alterando la lista. Por ejemplo una función que multiplique por dos el contenido de una lista (ejercicio 30). Para poder realizar los recorridos, se necesita añadir dos operaciones nuevas a las listas:

```
template <class T>
class Lista {
public:

   ...

   /**
```

```
     Devuelve el iterador mutable al principio de la lista.
     @return iterador al principio de la lista;
     coincidirá con final() si la lista está vacía.
     */
   IteradorMutable principioMutable() {
      return IteradorMutable(_prim);
   }

   /**
     @return Devuelve un iterador al final del recorrido
     (fuera de éste).
     */
   IteradorMutable finalMutable() const {
      return IteradorMutable(NULL);
   }

   ...

}
```

7.3. Usando iteradores para insertar elementos

El TAD lista puede extenderse para permitir insertar elementos en medio de la lista. Para eso se puede utilizar el mecanismo de iteradores: la operación recibe un iterador (mutable) situado en el punto de la lista donde se desea insertar un elemento. En concreto, el elemento lo añadiremos *a la izquierda* del punto marcado. Eso significa que si insertamos un elemento a partir de un iterador colocado al principio del recorrido, el nuevo elemento añadido pasará a ser el primero de la lista y el iterador apunta al segundo. Si el iterador está al final del recorrido (en `finalMutable()`), el elemento insertado será el nuevo último elemento de la lista, y el iterador sigue apuntando al `finalMutable()`, es decir por el hecho de insertar, la posición del iterador no cambia.

Por ejemplo, la siguiente función duplica todos los elementos de la lista, de forma que si el contenido inicial era por ejemplo `[1, 3, 4]` al final será `[1, 1, 3, 3, 4, 4]`:

```
void repiteElementos(Lista<int> &lista) {

   Lista<int>::IteradorMutable it = lista.principioMutable();

   while (it != lista.finalMutable()) {
      lista.inserta(it.elem(), it);
      it.avanza();
   }

}
```

La implementación de la operación `inserta` de la lista, por tanto, recibe el elemento a insertar y el iterador que marca el lugar de la inserción. La implementación debe cubrir el caso en el que el elemento insertado se convierte en el primero o el último de la lista para actualizar convenientemente los punteros a `_prim` o `_ult`.

```
void inserta(const T &elem, const IteradorMutable &it) {

   // Caso especial: ¿añadir al principio?
   if (_prim == it._act) {
```

```
      Cons(elem);
   } else
   // Caso especial: ¿añadir al final?
   if (it._act == NULL) {
      ponDr(elem);
   }
   // Caso normal
   else {
      insertaElem(elem, it._act->_ant, it._act);
   }
}
```

7.4. Usando iteradores para eliminar elementos

También se puede extender el TAD Lista para permitir borrar elementos interno a la lista. La operación recibe un iterador situado en el punto de la lista que se desea borrar. En esta ocasión, dado que ese elemento dejará de existir ese iterador recibido *deja de ser válido*. Para poder seguir recorriendo la lista la operación devuelve un nuevo iterador que deberá utilizarse desde ese momento para continuar el recorrido.

Por ejemplo, la siguiente función elimina todos los elementos pares de una lista de enteros:

```
void quitaPares(Lista<int> &lista) {

   Lista<int>::IteradorMutable it = lista.principioMutable();

   while (it != lista.finalMutable()) {
      if (it.elem() \% 2 == 0)
         it = lista.borra(it);
      else
         it.avanza();
   }

}
```

La implementación de la operación `borra` de la lista, por tanto, elimina el elemento y devuelve un iterador apuntando al *siguiente* elemento (o devuelve el iterador que marca el fin del recorrido si no quedan más). Igual que con la inserción se debe cubrir el caso en el que el elemento a borrar es el primero o es el último, para actualizar convenientemente los punteros a `_prim` o `_ult`.

```
IteradorMutable borra(const IteradorMutable &it) {
   if (it._act == NULL)
      throw AccesoInvalido();

   // Cubrimos los casos especiales donde
   // borramos alguno de los extremos
   if (it._act == _prim) {
      resto();
      return IteradorMutable(_prim);
   } else if (it._act == _ult) {
      inicio();
      return IteradorMutable(NULL);
   } else {
```

```
      // El elemento a borrar es interno a la lista.
      --_numElems;
      Nodo *sig = it._act->_sig;
      borraElem(it._act);
      return IteradorMutable(sig);
   }
}
```

7.5. Peligros de los iteradores

El uso de iteradores conlleva un riesgo debido a la existencia de *efectos laterales* en las operaciones. Al fin y al cabo un iterador abre la puerta a acceder a los elementos de la lista *desde fuera* de la propia lista. Eso significa que los cambios que ocurran en ella afectan al resultado de las operaciones del propio iterador. Por ejemplo el código siguiente fallará:

```
Lista<int> lista;
lista.Cons(3);
Lista<int>::Iterador it = lista.principio();
lista.resto(); // Quitamos el primer elemento
cout << it.elem() << endl; // Accedemos a él... CRASH
```

Cuando el iterador permite cambiar el valor y, sobre todo, cuando se pueden borrar elementos utilizando iteradores las posibilidades de provocar funcionamientos incorrectos crecen.

No obstante, las ventajas de los iteradores al permitir recorridos eficientes (y generales, ver ejercicio 28) superan las desventajas. Pero el programador deberá estar atento a los iteradores y ser consciente de que las operaciones de modificación del TAD que está siendo recorrido pueden invalidar sus iteradores.

7.6. En el mundo real...

Como hemos dicho antes, a pesar de los posibles problemas de los iteradores, son muy utilizados (en distintas modalidades) en los lenguajes mayoritarios, como C++, Java o C#. La ventaja de los iteradores es que permiten abstraer el TAD que se recorre y se pueden tener algoritmos genéricos que funcionan bien independientemente de la colección utilizada. Por ejemplo un algoritmo que sume todos los elementos dentro de un intervalo de una colección será algo así:

```
template <class Iterador>
int sumaTodos(Iterador it, Iterador fin) {
   int ret = 0;

   while (it != fin) {
      ret += it.elem();
      it.avanza();
   }

   return ret;
}
```

donde el tipo de los parámetros "`Iterador`" es un parámetro de la plantilla, para poder utilizarlo con cualquier iterador (que tenga los métodos `elem` y `avanza`) independientemente de la colección que se recorre.

En la librería de C++, además, los métodos que aquí hemos llamado `elem` y `avanza` utilizan los operadores que un programador utilizaría si estuviera manejando punteros. En concreto, utilizan el operador `*` para acceder al elemento y `++` para el incremento, de forma que la función anterior se convierte en:

```
template <class Iterador>
int sumaTodos(Iterador it, Iterador fin) {
   int ret = 0;

   while (ini != fin) {
      ret += *it;
      ++it;
   }

   return ret;
}
```

Dada la liberalidad de C y C++ con los tipos y la identificación deliberada que hacen entre punteros, enteros y arrays (lo que se conoce como *dualidad puntero-array*), la sintaxis de iteración utilizada anteriormente permite recorrer incluso un vector sin la necesidad de declarar una clase iterador:

```
int v[100];
...
cout << sumaTodos(&v[0], &v[100]) << endl;

// Versión alternativa, utilizando la dualidad
// puntero-array y aritmética de punteros
// cout << sumaTodos(v, v + 100) << endl;
```

Por último, la librería de C++ también distingue entre los iteradores que permiten modificar la colección y los que no y se habla de los iteradores *constantes* (los que aquí hemos llamado `Iterador`) y los no constantes (`IteradorMutable`). Además, el concepto de iterador permite abstraer *la dirección* del recorrido: el método `avanza` podría avanzar *hacia atrás* en la colección. Es lo que en C++ se conoce como `reverse_iterator`. Ver el ejercicio 28.

8. Para terminar...

Terminamos el tema con la implementación de la función de codificación descrita en la primera sección de motivación. Como se ve, al hacer uso de los TADs ya implementados el código queda muy claro; nos podemos centrar en la implementación del algoritmo de codificación sin preocuparnos del manejo vectores, índices, listas de nodos o estructuras de datos que se llenan y hay que redimensionar.

```
Lista<char> codifica(Lista<char> &mensaje) {

   // Primera fase; el resultado lo metemos
   // en una doble cola para facilitarnos
   // la segunda fase
   DCola<char> resFase1;

   Pila<char> aInvertir;
```

```
    Lista<char>::Iterador it = mensaje.principio();

    while (it != mensaje.final()) {
        char c = it.elem();
        it.avanza();

        // Si no es una vocal, metemos el caracter
        // en la pila para invertirlo posteriormente
        if (!esVocal(c))
            aInvertir.apila(c);
        else {
            // Una vocal: damos la vuelta
            // a todas las consonantes que nos
            // hayamos ido encontrando
            while (!aInvertir.esVacia()) {
                resFase1.ponDetras(aInvertir.cima());
                aInvertir.desapila();
            }
            // Y ahora la vocal
            resFase1.ponDetras(c);
        }
    }

    // Volcamos las posibles consonantes que queden
    // por invertir
    while (!aInvertir.esVacia()) {
        resFase1.ponDetras(aInvertir.cima());
        aInvertir.desapila();
    }

    // Segunda fase de la codificación: seleccionar
    // el primero/último de forma alternativa.
    Lista<char> ret;  // Mensaje devuelto
    while (!resFase1.esVacia()) {
        ret.ponDr(resFase1.primero());
        resFase1.quitaPrim();
        if (!resFase1.esVacia()) {
            ret.ponDr(resFase1.ultimo());
            resFase1.quitaUlt();
        }
    }

    return ret;
}
```

Notas bibliográficas

Gran parte de este capítulo se basa en el capítulo correspondiente de (Rodriguez Artalejo et al., 2011) y de (Peña, 2005). Es interesante ver las librerías de lenguajes como C++, Java o C#. Todas ellas tienen al menos una implementación de cada uno de los TADs lineales vistos en el tema e incluso más de una con distintas complejidades.

Ejercicios

Para probar muchos de los ejercicios siguientes te puede venir bien la función siguiente, que lee de la entrada estandar (`cin`) una línea y devuelve una lista con sus caracteres (devuelve la lista vacía si ya no hay más caracteres en la entrada):

```
#include <string>
#include <iostream>

Lista<char> readLine() {
  Lista<char> ret;

  // Si hemos llegado al final de la entrada...
  if (!std::cin)
    return ret;

  // Leemos la línea
  std::string line;
  std::getline(std::cin, line);

  for (unsigned int i = 0; i < line.size(); ++i)
    ret.ponDr(line[i]);

  return ret;
}
```

1. Implementa, con ayuda de una pila, un procedimiento *no* recursivo que reciba como parámetro un número entero $n \geq 0$ y escriba por pantalla sus dígitos en orden lexicográfico y su suma. Por ejemplo, ante $n = 64323$ escribirá:

   ```
   6 + 4 + 3 + 2 + 3 = 18
   ```

2. Implementa una función que reciba una pila y escriba todos sus elementos desde la base hasta la cima separados por espacios. Haz dos versiones, una versión recursiva y otra iterativa.

   Haz una tercera versión, implementando la funcionalidad como parte de la clase `Pila`.

3. Completa las dos implementaciones de las pilas con una operación nueva, `numElems` que devuelva el número de elementos almacenados en ella. En ambos casos la complejidad debe ser $\mathcal{O}(1)$.

4. Implementa una función que reciba una secuencia de caracteres en una lista que contiene, entre otros símbolos, paréntesis, llaves y corchetes abiertos y cerrados y decida si está equilibrada. Entendemos por secuencia equilibrada respecto a los tres tipos de símbolos si cada uno de ellos tiene el tantos abiertos como cerrados y si cada vez que aparece uno cerrado, el último que apareció fue su correspondiente abierto.

5. Dada una cadena que contiene únicamente los caracteres `<`, `>` y `.`, implementa una función que cuente cuántos *diamantes* podemos encontrar como mucho (`<>`), tras quitar toda la arena (`.`). Ten en cuenta que puede haber diamantes dentro de otros diamantes. Por ejemplo, en la cadena `<..<..>.><..><` hay tres diamantes.

6. Una frase se llama palíndroma si la sucesión de caracteres obtenida al recorrerla de izquierda a derecha (ignorando los espacios) es la misma que de derecha a izquierda. Esto sucede, por ejemplo, con la socorrida frase "*dábale arroz a la zorra el abad*" (ignorando la tilde de la primera palabra; podemos asumir que la entrada no tendrá tildes y todo serán o bien letras o bien espacios). Construye una función *iterativa* ejecutable en tiempo lineal que decida si una frase dada como lista de caracteres es o no palíndroma. Puedes utilizar TADs auxiliares.

7. Implementa una función que reciba una pila como parámetro de E/S y un número n e invierta los n valores almacenados en la pila, comenzando a contar desde la cima.

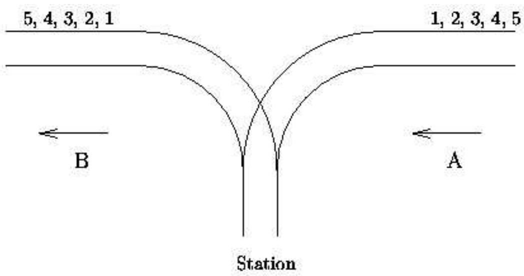
8. Una expresión aritmética construída con los operadores binarios $+$, $-$, $*$ y $/$ y operandos (representados cada uno por un único caracter) se dice que está en forma *postfija* si es o bien un sólo operando o dos expresiones en forma postfija, una tras otra, seguidas inmediatamente de un operador. Lo que sige es un ejemplo de una expresión escrita en notación infija habitual, junto con su forma postfija:

   Forma infija: `(A / (B - C)) * (D + E)`
   Forma postfija: `ABC-/DE+*`

   Diseña un algoritmo iterativo que calcule el valor de una expresión dada en forma postfija (mediante una secuencia de caracteres) por el siguiente método: se inicializa una pila vacía de números y se van recorriendo de izquierda a derecha los caracteres de la expresión. Cada vez que se pasa por un operando, se apila su valor. Cada vez que se pasa por un operador, se desapilan los dos números más altos de la pila, se componen con el operador, y se apila el resultado. Al acabar el proceso, la pila contiene un solo número, que es el valor de la expresión.

9. Repite el ejercicio anterior pero utilizando en lugar de una pila, una cola. Ante un operador se cogen los dos parámetros de la cabecera de la cola y se mete el resultado en la parte trasera de la misma.

10. Dicen las malas lenguas que el alcalde de Dehesapila recibió una buena tajada haciendo un chanchullo en la construcción de la estación de trenes. El resultado fue una estación con forma de "punto ciego" en la que únicamente hay una vía de entrada y una de salida. Para empeorar las cosas según la forma de la figura:



   Todos los trenes llegan desde A y van hacia B, pero reordenando los vagones en un orden distinto. ¿Puedes ayudar a establecer si el tren que entra con los vagones numerados 1, 2, ..., n puede reordenarse para que salga en un orden diferente enganchando y desenganchando vagones?

Por ejemplo, si entran los vagones 1, 2, 3, 4 y 5, se podrá sacarlos en el mismo orden (haciéndolos entrar y salir de uno en uno), en orden inverso (haciéndolos entrar a todos y luego sacándolos), pero no se podrá conseguir que salgan en el orden 5, 4, 1, 2, 3.

Escribe una función que lea de la entrada estándar un entero que indica el número de vagones que entran (n) y luego vaya leyendo en qué orden se quiere que vayan saliendo de la estación (n enteros). La función escribirá `POSIBLE` si el jefe de estación puede ordenar los vagones e `IMPOSIBLE` si no[10].

11. Extiende la función del ejercicio anterior para que en vez de indicar si es posible o no reordenar los vagones como se desea, escriba los movimientos que debe hacer el jefe de estación (meter un vagón en la estación / sacar un vagón de la estación).

12. Realiza una implementación de las colas utilizando vectores (dinámicos) de forma que la complejidad de todas las operaciones sea $\mathcal{O}(1)$. Para eso, en vez de utilizar un único atributo que indica cuántos elementos hay en el vector, utiliza dos atributos que permitan especificar el intervalo de elementos del vector que se están utilizando. Esta implementación se conoce con el nombre de *implementación mediante vectores circulares.*

13. Realiza una implementación con lista enlazada y nodo fantasma de las colas.

14. Completa la implementación de las colas con lista enlazada de nodos con una nueva operación `numElems` cuya complejidad sea $\mathcal{O}(1)$.

15. Extiende la implementación del TAD lista con una nueva operación `concatena` que reciba otra lista y añada sus elementos a la lista original. Por ejemplo, si `xs = [3, 4, 5]` y concatenamos `ys = [5, 6, 7]` con `xs.concatena(ys)`, al final tendremos que `xs` será `[3, 4, 5, 5, 6, 7]`. ¿Qué complejidad tiene la operación? ¿Podríamos conseguir complejidad $\mathcal{O}(1)$ de alguna forma?

16. Implementa los TADs de las pilas, colas y colas dobles utilizando la implementación del TAD lista, de forma que las nuevas implementaciones tengan como único atributo una lista.

17. Dados dos números con un número de dígitos indeterminado expresados en una lista de enteros, donde el primer elemento de cada lista es el dígito más significativo (dígito izquierdo) de cada número, implementa una función que devuelva una lista con la suma de ambos números.

18. El profesor de EDA ha decidido sacar a un alumno a hacer un examen sorpresa. Para seleccionar al "afortunado" ha numerado a cada uno de los n alumnos con un número del 1 al n y los ha colocado a todos en círculo. Empezando por el número 1, va "salvando" a uno de cada dos (es decir, "salva" al 2, luego al 4, luego al 6, etc.), teniendo en cuenta que al ser circular, cuando llega al final sigue por los que quedan sin salvar. ¿Qué número tendrá el alumno "afortunado"?

    Implementa una función:

```
int selecciona(int n);
```

[10] Este ejercicio es una traducción casi directa de `http://uva.onlinejudge.org/index.php?option=onlinejudge&pa` la imagen del enunciado es una copia directa.

que devuelva el número de alumno seleccionado si tenemos n alumnos.

Generaliza la función anterior para el caso en el que el profesor salve a uno de cada m en lugar de a uno de cada 2.

19. Dado un número natural $N \geq 2$, se llaman números afortunados a los que resultan de ejecutar el siguiente proceso: se comienza generando una cola que contiene los números desde 1 hasta N en este orden; se elimina de la cola un número de cada 2 (es decir, los números 1, 3, 5, etc.[11]); de la nueva cola, se elimina ahora un número de cada 3; etc. El proceso termina cuando se va a eliminar un número de cada m y el tamaño de la cola es menor que m. Los números que queden en la cola en este momentos son los afortunados.

    Diseña un procedimiento que reciba N como parámetro y produzca una lista formada por los números afortunados resultantes.

    (Indicación: para eliminar de una cola de n números un número de cada m puede reiterarse n veces el siguiente proceso: extraer el primer número de la cola, y añadirlo al final de la misma, salvo si le tocaba ser eliminado.)

20. Dada una lista de números, se repite el siguiente proceso hasta que queda únicamente uno: se elimina el primer elemento de la lista, y se pone el segundo al final. Implementa una función que, dada la lista, escriba el último número superviviente.

21. Implementa la función de *decodificación* de un mensaje según el algoritmo descrito en la primera sección del tema.

22. ¿Cuál será el texto final resultado de la siguiente pulsación de teclas, y dónde quedará el cursor colocado? `d`, `D`, `<Inicio>`, `<Supr>`, `<FlechaDcha>`, `A`, `<Inicio>`, `E`, `<Fin>`

    Haz una función que reciba una lista con las pulsaciones de teclas y devuelva una lista con el texto resultante.

23. Implementa una función que reciba un vector de enteros de tamaño N y un número K, y escriba el valor mínimo de cada subvector de tamaño K en $\mathcal{O}(n)$.

    Por ejemplo, para el vector `[1, 3, 2, 5, 8, 5]` y $k = 3$, debe escribir `[1, 2, 2, 5`.

Recorridos

24. Implementa una función que reciba una lista e imprima todos sus elementos. La complejidad de la operación debe ser $\mathcal{O}(n)$ incluso para la implementación del TAD lista utilizando listas enlazadas.

25. Implementa utilizando iteradores una función que reciba una secuencia de caracteres y devuelva el número de á'que tiene.

26. Dada una secuencia de enteros, contar cuántas posiciones hay en ella tales que el entero que aparece en esa posición es igual a la suma de todos los precedentes.

27. Implementa una función que reciba dos listas de enteros ordenados y devuelva una nueva lista ordenada con la unión de los enteros de las dos listas.

[11] Observa que este tipo de eliminación es distinto al del ejercicio 18.

28. Extiende el TAD lista implementando dos operaciones nuevas `principioInverso` y `finalInverso` que devuelva sendos `IteradorInverso` que permitan recorrer la lista desde el final al principio.

29. Utilizando el iterador del ejercicio anterior junto con el iterador de las listas, implementa una función que reciba una `Lista` y mire si la lista es palíndroma o no. Recuerda: *no* está permitido usar una pila (ese es el ejercicio 6).

30. Implementa una función que reciba una lista de enteros y duplique (multiplique por dos) todos sus elementos. Haz uso de los iteradores.

31. Implementa:

    - Una función que busque un elemento en una `Lista<T>` y devuelva un iterador mutable al primer elemento encontrado (o a `finalMutable()` si no hay ninguno).
    - Mejora la función anterior para que, en vez de recibir la lista, reciba dos iteradores que marcan el intervalo (abierto) sobre el que hay que buscar el elemento, de forma que la función anterior pueda implementarse como:

      ```
      template <class T>
      Lista<T>::IteradorMutable busca(const T&elem, Lista<T> &l) {
         return busca(elem, l.principio(), l.final());
      }
      ```

    - Implementa una función parecida a la anterior pero que en vez de buscar un elemento *lo borre*. En concreto, borrará todos los elementos que aparezcan en el intervalo (no sólamente el primero).
    - Utilizando las funciones anteriores, implementa una función que reciba una lista de caracteres y elimine todas las '`a`'de la primera palabra.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

BRASSARD, G. y BRATLEY, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. y STEIN, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.

MARTÍ OLIET, N., SEGURA DÍAZ, C. M. y VERDEJO LÓPEZ, J. A. *Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios*. Ibergarceta Publicaciones, 2012.

MARTÍ OLIET, N., ORTEGA MALLÉN, Y. y VERDEJO LÓPEZ, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.

PEÑA, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.

RODRIGUEZ ARTALEJO, M., GONZÁLEZ CALERO, P. A. y GÓMEZ MARTÍN, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.

STROUSTRUP, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201889544.