Diseño e implementación de TADs lineales

Marco Antonio Gómez es el autor principal de este tema

Facultad de Informática - UCM

6 de marzo de 2014

Estructuras de datos lineales

- Estructuras básicas
  - Vector de elementos.
  - Listas enlazadas.
- Las implementaciones de los TADs lineales pueden hacer uso de una u otra estructura de datos;
- La elección de una u otra podrá influir en la complejidad de sus operaciones.

Vectores de elementos.

Definición de tipos: Se 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.

```
private:
    /** Tamaño inicial del vector dinámico. */
   static const int TAM_INICIAL = 10;
    /** Puntero al array que contiene los datos. */
    T *_v;
    /** Tamanyo del vector _v. */
    unsigned int _tam;
    /** Numero de elementos reales guardados. */
    unsigned int _numElems;
};
```

Vectores de elementos.

Si se utilizan *vectores dinámicos*; cuando el vector se llena se puede construir un nuevo *array* más grande, y copiar los elementos.

```
 /** 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;
}
```

Al duplicar el tamaño del vector, el coste amortizado de la operación de añadir un elemento es constante.

Listas enlazadas

- Cada elemento es almacenado en un espacio de memoria independiente (un *nodo*) y la colección completa se mantiene utilizando punteros.
- 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.
- Todas las implementaciones tendrán la definición de la clase `Nodo` que almacena el elemento y los punteros al nodo siguiente y al nodo anterior (sólo en listas doblemente enlazadas).
- Este curso implementaremos la clase `Nodo` como una clase interna del TAD que estemos definiendo.

Lista enlazada simple.

```
template <class T>
...
private:
    class Nodo {
    public:
        Nodo() : _sig(NULL) {}
        Nodo(const T &elem) :
            _elem(elem), _sig(NULL) {}
        Nodo(const T &elem, Nodo *sig) :
            _elem(elem), _sig(sig) {}

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

Operaciones con listas enlazadas simples.

```
class xxxx {
   class Nodo {......}
   Nodo * _ini;
   ....
   public: ....
};
```

Añadir un nodo por el comienzo de una lista.

```
void xxxx::AnadePrincipio(const T &e) {
   Nodo *nuevo = new Nodo(e, _ini);
   _ini = nuevo;
}
```

Operaciones con listas enlazadas simples.

Eliminar el nodo del comienzo de la lista.

```
 void xxxx::elimPrincipio() {
    if (_ini == NULL) throw ListaVacia;
    Nodo *aux = _ini;;
    _ini = _ini->sig;
    delete aux;
}
```

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

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

Operaciones con listas enlazadas simples.

Añadir un nodo a continuación de un cierto elemento dado. Si el elemento no existe el nodo no se añade.

```
void xxxx::Anade(const T &e, const T & v ) {
   Nodo * aux = _ini;
   while (aux != NULL && aux->_elem != v)
            aux = aux->_sig;
   if (aux != NULL) {
      Nodo *nuevo = new Nodo(e, aux->_sig);
      aux->_sig  = nuevo;
   }
}
```

Operaciones con listas enlazadas simples.

Eliminar un elemento. Si el elemento no existe la lista no se modifica.

```
void xxxx::Elimina(const T &e) {
 if (_ini != NULL) {
  if (_ini->_elem == e) {
     Nodo* rem = _ini;
     _ini = _ini->sig;
     delete rem;  }
  else {
    Nodo * aux = _ini;
    while (aux->sig != NULL && aux->_sig->_elem != e)
          aux = aux->_sig;
    if (aux->_sig != NULL) {
          Nodo *rem = aux->_sig;
          aux->_sig = aux->_sig->_sig;
          delete rem;
    }
 }
}
```

Lista enlazada doble.

```
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;
};
```

Operaciones con listas enlazadas dobles.

Añadir un nodo a continuación de un cierto elemento dado. Si el elemento no existe el nodo no se añade.

```
void yyyy::Anade(const T &e, const T & v ) {
   Nodo * aux = _ini;
   while (aux != NULL && aux ->_elem != v)
            aux = aux->_sig;
   if (aux != NULL) {
      Nodo *nuevo = new Nodo(aux,e,aux->_sig);
      aux->_sig->_ant = nuevo;
      aux->_sig  = nuevo;
   }
}
```

Operaciones con listas enlazadas dobles.

Eliminar un elemento. Si el elemento no existe la lista no se modifica.

```
void yyyy::Elimina(const T &e) {
    Nodo * aux = _ini;
    while (aux != NULL && aux ->_elem != e)
          aux = aux->_sig;
    if (aux != NULL) {
          Nodo *rem = aux;
          if (aux->_sig != NULL)
                aux->_sig->_ant = aux->ant;
          if (aux->_ant != NULL)
                aux->_ant->_sig = aux->_sig;
          else _ini = _ini->_sig;
          delete rem;
    }
 }
}
```

En el mundo real...

- La librería de C++ tiene implementadas dos estructuras de datos llamadas *contenedores* (`std::vector` y `std::list`).
- Las implementaciones de los TADs se parametrizan con el tipo de contenedor que se quiera.
- En esta asignatura no nos preocuparemos de la reutilización utilizando las estructuras descritas en los apuntes (inaceptable en el mundo real).
- La principal pega de los vectores dinámicos es el consumo de memoria: los vectores crecen indefinidamente, nunca decrecen.
- En la librería de C++ existe otro tipo de contenedor (`std::dequeue`) que no sufre este problema.

Pilas

- Una pila (stack) representa una colección de valores donde sólo es posible acceder al último elemento añadido.
- Operaciones:
  - Crear una pila vacía.
  - Apilar un nuevo elemento en una pila `apila` (`push`).
  - Desapilar el último elemento (parcial) `desapila`(`pop`).
  - Acceder al último elemento añadido (parcial) `cima`(`top`).
  - Averiguar si una pila tiene elementos `esVacia`(`isempty`).
- Las pilas tienen muchas utilidades, como por ejemplo "dar la vuelta" a una secuencia de datos.

Implementación de pilas con vectores dinámicos

- Tipo representante: puntero al vector (`_v`), número de elementos almacenados (`_numElems`) y el tamaño máximo del vector (`_tam`). Cima de la pila: `_v[_numElems-1]`.
- Invariante de la representación
  $0 \leq$ p._numElems $\leq$ p._tam
  $\forall i : 0 \leq i <$ p._numElems $p._v[i]$ cumple el invariante de la representación del tipo de sus elementos.
- Relación de equivalencia Dos pilas son iguales si el número de elementos almacenados coincide y sus valores respectivos, uno a uno, también.

```
// Tratamiento de excepciones
class ExcepcionTAD {
public:
  ExcepcionTAD() {}
  ExcepcionTAD(const std::string &msg) : _msg(msg) {}
  const std::string msg() const { return _msg; }
  friend std::ostream &operator<<
         (std::ostream &out, const ExcepcionTAD &e){
         out << e._msg; return out; }
protected:
  std::string _msg;
};

// Excepción generada por algunos métodos de Pila. */
class EPilaVacia : public ExcepcionTAD {
public:
    EPilaVacia() {};
    EPilaVacia(const std::string &msg) :
        ExcepcionTAD(msg) {}
};
```

```
template <class T>
class Pila {
public:
    Pila() : _v(new T[TAM_INICIAL]), _tam(TAM_INICIAL),
             _numElems(0) { }

    ~Pila() { delete [] _v;  }

//  Apila un elemento. Operación generadora.
// @param elem Elemento a apilar.
void apila(const T &elem) {
    if (_numElems == _tam)  amplia();
    _v[_numElems] = elem;
    _numElems++;
}
```

```
// Desapila un elemento. Modificadora parcial,
// falla si la pila está vacía.
void desapila() {
    if (esVacia()) throw EPilaVacia();
    --_numElems;
}

// Devuelve el elemento en la cima de la pila.
// observadora parcial, falla si la pila está vacía.
const T &cima() const {
    if (esVacia()) throw EPilaVacia();
    return _v[_numElems - 1];
}
```

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

private:
    T *_v;
    // Tamanyo del vector _v.
    unsigned int _tam;
   // Numero de elementos reales guardados.
    unsigned int _numElems;
};
```

Implementación con listas enlazadas

- Tipo representante: 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.
- Invariante de la representación: 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:
- Relación de equivalencia: 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):

```
template <class T>
class PilaLE {
public:
    PilaLE() : _cima(NULL) {}

   // Constructor copia
   PilaLE(const PilaLE<T> &other) : _cima(NULL) {
      copia(other); }

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

    void apila(const T &elem) {
        _cima = new Nodo(elem, _cima); }

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

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

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

// Operador de asignacion
PilaLE<T> &operator=(const PilaLE<T> &other) {
    if (this != &other) {
      libera(_cima);
      copia(other);
   }
   return *this;
}
```

```
// Operador de comparacion.
bool operator==(const PilaLE<T> &rhs) const {
   Nodo *cima1 = _cima;
   Nodo *cima2 = rhs._cima;
   while ((cima1 != NULL) && (cima2 != NULL) &&
       cima1->_elem == cima2->_elem) {
         cima1 = cima1->_sig;
         cima2 = cima2->_sig;
   }
   return (cima1 == NULL) && (cima2 == NULL);
}
```

```
private:
  class Nodo {.....}
  Nodo *_cima;
  void libera(Nodo* prim) {...}
  void copia(const PilaLE &other) {
    if (other.esVacia()) {_cima = NULL;}
    else {
      Nodo *act = other._cima;
      Nodo *ant;
      _cima = new Nodo(act->_elem);
      ant = _cima;
      while (act->_sig != NULL) {
        act = act->_sig;
        ant->_sig = new Nodo(act->_elem);
        ant = ant->_sig;
      }
    }
  }
};
```

- La complejidad de las operaciones de ambas implementaciones es similar:

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`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Colas

- Las colas (queue) 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).
- Se las conoce como estructuras FIFO (*first in, first out*), el primer elemento que entra es el primero que saldrá.
- Operaciones:
  - `ColaVacia`: genera una cola vacía.
  - `PonDetras`: añade un nuevo elemento a la cola (`enqueue`)
  - `quitaPrim`: modificadora parcial que elimina el primer elemento de la cola. Falla si la cola está vacía (`dequeue`).
  - `primero`: observadora parcial que devuelve el primer elemento de la cola (el más antiguo). Falla si la cola está vacía.
  - `esVacia`: observadora, permite averiguar si la cola tiene elementos.

Implementación de colas con un vector

Tipo representante:

- Vector dinámico, el primer elemento de la cola está siempre en la posición 0 del vector.
- Inconveniente: el coste de la operación `quitaPrim` está en $\mathcal{O}(n)$, ya que se deben desplazar todos los elementos válidos una posición a la izquierda.
- Solución: Colas circulares.

Implementación de colas con una lista enlazada

- Tipo representante:
  - una lista enlazada simple en la que el primer nodo contiene el elemento que hay en la cabecera de la cola,
  - un puntero al primer nodo y
  - otro puntero al último nodo.
  - La cola vacía se representa con los dos punteros a NULL.
- Invariante de la representación: similar al de las pilas.
- Relación de equivalencia: similar a las pilas.

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

template <class T>
class Cola {
private:
    class Nodo {...}
    // Puntero al primer y último elemento
    Nodo *_prim, *_ult;
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.
// @param elem Elemento a añadir.
void ponDetras(const T &e) {
    Nodo *nuevo = new Nodo(e);
    _ult->_sig = nuevo;
     _ult = nuevo;
     if (_prim == NULL)  _prim = nuevo;
}

// Devuelve el primer elemento de la cola. Operación
// observadora parcial, falla si la cola está vacía.
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.
void quitaPrim() {
   if (esVacia()) throw EColaVacia();
   Nodo *aBorrar = _prim;
   _prim = _prim->_sig;
    delete aBorrar;
    if (_prim == NULL) _ult = NULL;
 }

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

- Para evitar el caso especial de la cola vacía en la implementación de las funciones se puede utilizar un *nodo fantasma* o *cabecera* que no guarda ningún elemento.
- La complejidad de las operaciones no varía, pero su programación es más sencilla.
- 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`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Colas dobles

- Las colas dobles son una generalización de las colas que permiten operar en los dos extremos. 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). Falla si la cola está vacía.
  - `quitaUlt`: modificadora parcial que elimina el último elemento de la cola. Falla si la cola está vacía.
  - `ultimo`: observadora parcial que devuelve el último elemento de la cola. Falla si la cola está vacía.
  - `esVacia`: observadora, permite saber si la cola tiene elementos.

- Tipo representante:
  - Lista circular doblemente enlazada con nodo cabecera.
  - Un puntero al nodo cabecera.
  - La cola vacía esta representada por un nodo cabecera que no contiene ningún elemento y cuyos punteros anterior y siguiente apuntan a él mismo.
  - La implementación hará que *el siguiente* al nodo fantasma sea el primero de la cola (la cabecera), mientras que el *anterior* será el último.
- Relación de equivalencia dos objetos son iguales si las listas enlazadas contienen el mismo numero de elementos y sus valores uno a uno coinciden, empezando en el nodo cabecera y exceptuando este.

- Invariante de la representación:
  - 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.

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

template <class T>
class DCola {
private:
    class Nodo {...}
    // Puntero al nodo fantasma
    Nodo *_fantasma;
    ...
public:
    // Constructor
    DCola() {
        _fantasma = new Nodo();
        _fantasma->_sig = _fantasma;
        _fantasma->_ant = _fantasma;
    }
```

```
// Destructor; elimina la lista.
~DCola() {
// 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.
void ponDetras(const T &e) {
  Nodo* nuevo = new Nodo(_fantasma->_ant, e, _fantasma);
  _fantasma->_ant->_sig = nuevo;
  _fantasma->_ant = nuevo;
}
```

```
//Devuelve el primer elemento de la cola;
// Observadora parcial
const T &primero() const {
   if (esVacia()) throw EDColaVacia();
   return _fantasma->_sig->_elem;
}

// Elimina el primer elemento de la doble cola.
// Operación modificadora parcial,
void quitaPrim() {
  if (esVacia()) throw EDColaVacia();
  Nodo* aux = _fantasma->_sig;
  _fantasma->_sig = _fantasma->_sig->_sig;
  aux->_sig->_ant = _fantasma;
  delete aux;
}
```

```
// Añade un elemento delante de una doble cola.
void ponDelante(const T &e) {
  Nodo* nuevo = new Nodo(_fantasma, e, _fantasma->_sig);
  _fantasma->_sig->_ant = nuevo;
  _fantasma->_sig = nuevo;
}

// Devuelve el último elemento de la doble cola. Es
// un error preguntar por el último si está vacía.
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.
void quitaUlt() {
  if (esVacia()) throw EDColaVacia();
  Nodo* aux = _fantasma->_ant;
  aux->_ant->_sig = _fantasma;
  _fantasma->_ant = aux->_ant;
  delete aux;
}

// Operación observadora para saber si una doble cola
// tiene o no elementos.
bool esVacia() const {
  return _fantasma->_sig == _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`	$\mathcal{O}(1)$

PARTE I

Listas

PARTE 2

- Las listas son los TADs lineales más generales posibles.
- Permiten la consulta y modificación de los dos extremos (como las colas dobles) pero también acceder a cualquier punto intermedio.
- Operaciones:
  - `ListaVacia`. Generadora. Construye la lista vacía.
  - `Cons`. Generadora. Añade un elemento en la cabeza de la lista ("parte izquierda" en las posibles figuras).
  - `ponDr`. Modificadora. Añade un elemento en la cola de la lista ("parte derecha").
  - `primero`. Observadora parcial. Devuelve el primer elemento.
  - `resto`. Modificadora parcial. Quita el primer elemento.
  - `ultimo`. Observadora parcial. Devuelve el último elemento.
  - `inicio`. Modificadora parcial. Elimina el último elemento.
  - `esVacia`. Observadora.
  - `numElems`. Observadora.
  - `elem`. Observadora. Devuelve el elemento i-ésimo de la lista (0..numElems-1).

- Notación: `x` e `y` representan *un elemento* de la lista, `xs` o `ys` representan listas:

`ListaVacia`	$\equiv$	`[]`
`Cons(x, ListaVacia)`	$\equiv$	`[x]`
`Cons(x, xs)`	$\equiv$	`[x|xs]`
`ponDr(xs, x)`	$\equiv$	`[xs#x]`
`concatena(xs, ys)`[1]	$\equiv$	`xs++ys`

- Las listas pueden implementarse utilizando vectores o listas enlazadas.
- Con listas enlazadas se penaliza la operación *elem* ($\mathcal{O}(n)$), pero se consigue una complejidad constante para el resto de operaciones.

- Tipo representante: lista doblemente enlazada (sin nodo cabecera y no circular) y dos punteros, uno al primer nodo y otro al último.
  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.
- Función de abstracción y la relación de equivalencia son similares a las vistas para las colas dobles

Implementación

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

template <class T>
class Lista {
private:
    class Nodo {...}
    Nodo *_prim, *_ult;  // primer y último elemento
    unsigned int _numElems;     // Número de nodos
public:
    Lista() : _prim(NULL), _ult(NULL), _numElems(0) {}
    ~Lista() {  // destructor. elimina la lista
        libera(_prim);
        _prim = NULL;  _ult = NULL;
    }
```

```
// Añade un nuevo elemento en la cabeza de la lista.
// @param elem Elemento que se añade
void Cons(const T &elem) {
    _numElems++;
    Nodo * nuevo = new Nodo(NULL,e,_prim);
    if (_prim != NULL) _prim->_ant = nuevo;
                no vacía
    _prim = nuevo;
     if (_ult == NULL) _ult = _prim;
     }          vacía

// Añade un elemento al final de la lista.
void ponDr(const T &elem) {
    _numElems++;
    Nodo * nuevo = new Nodo(_ult,e,NULL);
    if (_ult != NULL) _ult->_sig = nuevo;
                no vacía
    _ult = nuevo;
    if (_prim == NULL) _prim = _ult;
}           vacía
```

```
// Devuelve el valor almacenado en la cabecera
const T &primero() const {
    if (esVacia()) throw EListaVacia();
    return _prim->_elem;
 }

// Devuelve el valor almacenado en la última posición
const T &ultimo() const {
    if (esVacia()) throw EListaVacia();
    return _ult->_elem;
}
```

```
// Elimina el primer elemento de la lista.
void resto() {
   if (esVacia()) throw EListaVacia();
   Nodo *aBorrar = _prim;
   _prim = _prim->_sig;
   if (_prim == NULL) _ult = NULL;
   else _prim->_ant = NULL;
    --_numElems;
   delete aBorrar;
}

// Elimina el último elemento de la lista.
void inicio() {
   if (esVacia()) throw EListaVacia();
   Nodo *aBorrar = _ult;
   _ult = _ult->_ant;
  if (_ult == NULL) _prim = NULL;
  else _ult->_sig = NULL;
  --_numElems;
  delete aBorrar;
}
```

```
// devuelve si la lista es vacia
bool esVacia() const {
   return _prim == NULL;
}

// Devuelve el número de elementos de la lista.
unsigned int numElems() const {
   return _numElems;
}

// Devuelve el elemento i-ésimo de la lista
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;
}
```

- 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`	$\mathcal{O}(1)$

Recorridos de listas

- Problema de las listas anteriores. Un bucle que escribe uno a uno todos los elementos tiene coste cuadrático.

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

- Solución: utilizar un *iterador*.

- Un *iterador* es 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* 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.

- Se extiende el TAD lista con dos operaciones:
  - `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*.
- La forma de recorrer una lista es:

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

Implementación de un iterador básico

- Se define una clase interna `Iterador` que tiene 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);
    }

    ...
};
```

Iteradores para modificar elementos

- El iterador puede utilizarse para cambiar elementos. En este caso no declararemos la operación `principio` como `const`, ya que la lista puede resultar alterada, aunque no sea en la función.
- La nueva clase `IteradorMutable` es una copia de la clase `Iterador` con un método nuevo

```
class IteradorMutable {
public: ...
    void pon(const T &elem) {
        if (_act == NULL) throw AccesoInvalido();
        _act->_elem = elem;
    }
};
```

- Para utilizar este nuevo iterador se deben declarar las operaciones `principioMutable` y `finalMutable` en el TAD de las listas.

Usando iteradores para insertar elementos

- El TAD lista puede extenderse para permitir insertar elementos en medio de la lista.
- La operación recibe un iterador mutable situado en el punto de la lista donde se desea insertar un elemento. 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 en `finalMutable()`, el elemento insertado será el nuevo último elemento de la lista, y el iterador sigue apuntando a `finalMutable()`, es decir, por el hecho de insertar, la posición del iterador no cambia.

- Ejemplo, función que 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` recibe el elemento a insertar y el iterador que marca el lugar de la inserción.

```
void insertar(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);
    }
    else { // Caso normal
    Nodo* nuevo = new Nodo(it._act->_ant,elem,it._act);
    it._act->_ant = nuevo;
    it._act = nuevo;
    }
}
```

Usando iteradores para eliminar elementos

- Operación que elimina un elemento interno de la lista. Recibe un iterador situado en el punto de la lista que se desea borrar.
- Se devuelve un nuevo iterador que puede utilizarse para continuar el recorrido.
- Por ejemplo, eliminar 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();
    }
}
```

Implementación de la función que borra un elemento.

```
IteradorMutable borra(const IteradorMutable &it) {
    if (it._act == NULL) throw AccesoInvalido();
    if (it._act == _prim) { // caso especial
        resto();
        return IteradorMutable(_prim);
    } else if (it._act == _ult) { // caso especial
        inicio();
        return IteradorMutable(NULL);
    } else { // caso general
    // El elemento a borrar es interno a la lista.
        --_numElems;
        Nodo *sig = it._act->_sig;
        sig->_ant = it._act._ant;
        it._act._ant._sig = sig->_sig;
        delete it._act;
        return IteradorMutable(sig);
    }
}
```

Peligros de los iteradores

- El uso de iteradores conlleva un riesgo debido a la existencia de *efectos laterales* en las operaciones, ya que un iterador abre la puerta a acceder a los elementos de la lista *desde fuera* de la propia lista.
- Cambios que ocurran en la lista pueden afectar al resultado de las operaciones del iterador.
- Por ejemplo el código siguiente generará 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
```

En el mundo real...

- 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 T>
int sumaTodos(T it, T fin) {
    int ret = 0;
    while (it != fin) {
        ret += it.elem();
        it.avanza();  }
    return ret; }
```

- El tipo T debe instanciarse con un iterador, que tenga los métodos `elem` y `avanza`.

- En la librería de C++, los métodos `elem` y `avanza` se referencian con el operador `*` para acceder al elemento y `++` para el incremento. 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;
}
```

- Se puede abstraer *la dirección* del recorrido: el método `avanza` podría avanzar *hacia atrás* en la colección.

Para terminar...Ejemplo

- 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"

```
Lista<char> codifica(Lista<char> &mensaje) {
// Primera fase; metemos el resultado en una doble cola
    DCola<char> resFase1; Pila<char> aInvertir;
    Lista<char>::Iterador it = mensaje.principio();

    while (it != mensaje.final()) {
        char c = it.elem();   it.avanza();
        // No vocal, metemos el caracter en la pila
        if (!esVocal(c)) aInvertir.apila(c);
        else { // vocal: damos la vuelta a las cons.
            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;
    }
```