UNIVERSIDAD COMPLUTENSE MADRID

Tema 5: Tipos de datos lineales

Miguel Gómez-Zamalloa
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Facultad de Informática

Temario

- Análisis de eficiencia
  1. Análisis de eficiencia de los algoritmos ✔
- Algoritmos
  2. Divide y vencerás (DV), o *Divide-and-Conquer* ✔
  3. Vuelta atrás (VA), o *backtracking*
- Estructuras de datos
  4. Especificación e implementación de TADs ✔
  5. **Tipos de datos lineales**
  6. Tipos de datos arborescentes
  7. Diccionarios
  8. Aplicación de TADs

Contenidos

El TAD vector

El TAD vector

- El TAD vector es el contenedor de datos lineal más conocido y utilizado.
- Abstrae a los arrays dotándolos de gestión propia de memoria (redimensión cuando es necesario) y operaciones extra.
- Operaciones básicas: **operator**[], push_back, pop_back, insert , erase, empty, size y resize .
- Representanción: Utilizaremos un array dinámico y dos enteros (contador de elementos y capacidad real del array).

```
template <class T>
class vector {
protected:
  static const int TAM_INI = 10; // tamaño por defecto del array

  int nelems; // número de elementos del vector

  int capacidad; // capacidad del array

  T* array; // array dinámico (redimensionable)
```

El TAD vector

- Constructor, destructor, copia y asignación análogos a **class** Set:

```
// constructor: vector vacío
    vector() : nelems(0), capacidad(TAM_INI), array(new T[capacidad]) {}

    // constructor: vector con n elementos ocupados y capacidad n
    vector(int n) : nelems(n), capacidad(n), array(new T[capacidad]) {}

    // destructor
    ~vector() {
        libera();
    }

    // constructor por copia
    vector(vector<T> const& other) {
        copia(other);
    }

    // operador de asignación
    vector<T>& operator=(vector<T> const& other) {
        if (this != &other) {
            libera();
            copia(other);
        }
        return *this;
    }
```

- Métodos libera, copia y amplia también análogas a **class** Set.

El TAD vector

```
// añade un elemento al final. O(1) amortizado
void push_back(T const& elem) {
    if (nelems == capacidad) amplia(capacidad*2);
    array[nelems] = elem;
    ++nelems;
}

// quita el último elemento. O(1)
void pop_back() {
    if (nelems > 0) --nelems;
}

// devuelve referencia constante a elemento en posición i. O(1)
const T& operator[](int i) const {
    if (i < 0 || i >= nelems) throw std::out_of_range(...);
    return array[i];
}

// devuelve referencia a elemento en posición i. O(1)
T& operator[](int i){
    if (i < 0 || i >= nelems) throw std::out_of_range(...);
    return array[i];
}
```

El TAD vector

```
// inserta e en pos i desplazando para abrir hueco. O(nelems)
void insert(const T& e, int i){
    if (i < 0 || i >= nelems) throw std::out_of_range("...");
    if (nelems == capacidad) amplia(capacidad*2);
    desplazaDchaDesde(i);
    array[i] = e;
    ++nelems;
}

// borra elem de pos i (desplaza para cerrar hueco). O(nelems)
void erase(int i){
    if (i < 0 || i >= nelems) throw std::out_of_range("...");
    desplazaIzdaDesde(i);
    --nelems;
  }
```

El TAD vector

```
// consulta si el vector es vacío. O(1)
bool empty() const {
    return nelems == 0;
}

// consulta el tamaño del vector. O(1)
int size() const {
    return nelems;
}

// cambia el tamaño del vector dando mas capacidad si es
necesario. O(n)
void resize(int n) {
    if (n < 0) throw std::out_of_range("parametro no valido");
    if (n <= nelems) nelems = n; // Se podría redimensionar
    else {
        amplia(n);
        nelems = n;
    }
}
```

Complejidad operaciones

`vector`	$\mathcal{O}(1)$
`push_back`	$\mathcal{O}(1)$*
`pop_back`	$\mathcal{O}(1)$
`operator[]`	$\mathcal{O}(1)$
`insert`	$\mathcal{O}(nelems)$
`erase`	$\mathcal{O}(nelems)$
`empty`	$\mathcal{O}(1)$
`size`	$\mathcal{O}(1)$
`resize`	$\mathcal{O}(n)$

- En (*) la complejidad sería $\mathcal{O}(1)$ excepto cuando redimensiona, que sería $\mathcal{O}(nelems)$. Se dice que es coste amortizado constante.
- Ojo con la complejidad lineal de `insert` e `erase`. Si se tienen que usar masivamente probablemente es que debes usar otro TAD.

Pilas - El TAD stack

Pilas - El TAD stack

- Las pilas son un contenedor de datos lineal que trabajan con política de almacenamiento/acceso de tipo *LIFO* (last-in first-out), es decir, solo se puede acceder/extraer al último elemento que se añadió.
- Operaciones básicas: push, pop, top, empty y size.

```
template <class T>
class stack {
public:
    stack(); // Construye pila vacía

    void push(T const& elem); // Apila un elemento

    T const& top() const; // Devuelve cima. Lanza excepción si vacía

    void pop(); // Desapila el elemento en la cima

    bool empty() const; // Consulta si la pila está vacía

    int size() const; // Consulta el tamaño de la pila
}
```

Pilas - El TAD stack

- Aplicaciones:
  - Estructura auxiliar para algoritmos.
  - Compiladores e intérpretes (máquinas virtuales).
  - Evaluación de expresiones.
  - Transformación de algoritmos recursivos a iterativos.
  - Recorridos de árboles y grafos.
- Podemos trabajar con dos representaciones distintas:
  1. Array dinámico redimensionable o vector.
  2. Nodos enlazados.

TAD stack usando array dinámico

- Misma idea que vimos en el TAD Set.
- Habría que decidir si la cima va al comienzo o al final del array.
- Claramente nos interesa que vaya al final $\Rightarrow$ Si no habría que andar abriendo y cerrando huecos lo que implica recorridos innecesarios.

```
template <class T>
class stack {
protected:
    static const int TAM_INICIAL = 10; // tamaño inicial del array

    int nelems; // número de elementos en la pila

    int capacidad; // tamaño del array

    T* array; // Puntero al array dinámico (redimensionable)
```

TAD stack usando array dinámico

- Constructor, destructor, copia y asignación análogos a **class** Set:

```
public:
    // Construye pila vacía
    stack() : nelems(0), capacidad(TAM_INICIAL), array(new T[capacidad]) {}

    // Destructor
    ~stack() {
        libera();
    }

    // Constructor por copia
    stack(stack<T> const& other) {
        copia(other);
    }

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

- Métodos libera, copia y amplia análogos también a **class** Set.

TAD stack usando array dinámico

```
// Apila elemento elem. O(1), salvo cuando redimensiona
void push(T const& elem) {
    array[nelems] = elem;
    ++nelems;
    if (nelems == capacidad)
        amplia();
}

// Devuelve referencia a la cima. O(1)
T const& top() const {
    if (empty())
        throw std::domain_error("la pila vacia no tiene cima");
    return array[nelems - 1];
}
```

TAD stack usando array dinámico

```
    // Desapila la cima. Lanza excepción si pila vacía. O(1)
    void pop() {
        if (empty())
            throw std::domain_error("desapilando de la pila vacia");
        --nelems;
    }

    // Consulta si la pila está vacía. O(1)
    bool empty() const {
        return nelems == 0;
    }

    // Consulta el tamaño de la pila. O(1)
    int size() const {
        return nelems;
    }
}
```

TAD stack usando nodos enlazados

- Podemos usar nodos enlazados.
- Basta con enlaces simples (unidireccionales) y un único puntero al nodo cima, de manera que la cima apunta al siguiente y así sucesivamente.

```
template <class T>
class LinkedListStack {
protected:
  /** Clase nodo que almacena internamente el elemento (de tipo T),
  y un puntero al nodo siguiente (nullptr si es el último) */
  class Nodo {
  public:
      Nodo() : sig(nullptr) {}
      Nodo(const T& e, Nodo* s = nullptr) : elem(e), sig(s) {}
          T elem;
          Nodo* sig;
  };

  Nodo* cima;
  int nelems;
};
```

TAD stack usando nodos enlazados

```
LinkedListStack() : cima(nullptr), nelems(0) {}

~LinkedListStack() {
    libera();
    cima = nullptr;
}

void push(const T& elem) {
    cima = new Nodo(elem, cima);
    nelems++;
}

void pop() {
    if (empty())
        throw std::domain_error("desapilando de la pila vacia");
    Nodo* aBorrar = cima;
    cima = cima->sig;
    delete aBorrar;
    --nelems;
}
```

TAD stack usando nodos enlazados

```
const T& top() const {
    if (empty())
        throw std::domain_error("la pila vacia no tiene cima");
    return cima->elem;
}

bool empty() const {
    return cima == nullptr;
}

// consultar el tamaño de la pila
int size() const {
    return nelems;
}
```

TAD stack usando nodos enlazados

- Métodos protegidos para la copia y liberación:

```
protected:
void copia(const LinkedListStack& other) {
    if (other.empty()) {
        cima = nullptr;
        nelems = 0;
    } else {
        Nodo* act = other.cima;
        Nodo* ant;
        cima = new Nodo(act->elem);
        ant = cima;
        while (act->sig != nullptr) {
            act = act->sig;
            ant->sig = new Nodo(act->elem);
            ant = ant->sig;
        }
        nelems = other.nelems;
    }
}

void libera() {
    while (cima != nullptr) {
        Nodo* aux = cima;
        cima = cima->sig;
        delete aux;
    }
}
```

Conclusiones

Operación	Array dinámico	Nodos enlazados
`stack`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`push`	$\mathcal{O}(1)$*	$\mathcal{O}(1)$
`pop`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`top`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`size`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

- En (*) la complejidad sería $\mathcal{O}(1)$ excepto cuando redimensiona, que sería $\mathcal{O}(n)$. Se dice que es coste amortizado constante.
- Aunque los nodos enlazados requieren un cierto overhead tanto en memoria como en tiempo (los **new** no son gratis).
- Conclusión: Ambas representaciones se utilizan en la práctica.

Colas - El TAD queue

Colas - El TAD queue

- Las colas son un contenedor de datos lineal que trabajan con política de almacenamiento/acceso de tipo *FIFO* (first-in first-out), es decir, se inserta por el final y se extrae/accede por el principio.
- Operaciones básicas: push, pop, front, empty y size.

```
template <class T>
class queue {
public:
    queue(); // Construye cola vacía

    void push(T const& elem); // Añade elem al final de la cola

    T const& front() const; // Devuelve primero (excepción si vacía)

    void pop(); // Elimina el primero de la cola

    bool empty() const; // Consulta si la cola está vacía

    int size() const; // Consulta el tamaño de la cola
}
```

Colas - El TAD queue

- Aplicaciones:
  - Estructura auxiliar para algoritmos.
  - Gestión de eventos/tareas (scheduling).
  - Recorridos de árboles y grafos.
- Posibles representaciones:
  1. Array dinámico redimensionable o vector:
     - Problema importante con operación pop: En principio requeriría desplazamiento para cerrar el hueco $\Rightarrow \mathcal{O}(n)$
     - Solución: No cerrar el hueco, llevar índice indicando donde comienzan los datos y manejar el array de manera circular (ver enlace).
  2. Nodos enlazados.
     - Es la representación habitual y es la que veremos.

TAD queue usando nodos enlazados

- En este caso es necesario mantener un puntero al primer nodo y otro puntero al último.
- Sigue bastando con enlaces simples (unidireccionales) del primero al segundo, del segundo al tercero y así sucesivamente.

```
template <class T>
class queue {
protected:
    // punteros al primer y último elemento
    Nodo* prim;
    Nodo* ult;

    // número de elementos en la cola
    int nelems;

public:
    // constructor: cola vacía
    queue() : prim(nullptr), ult(nullptr), nelems(0) {}

    // destructor
    ~queue() { libera(); }
```

TAD queue usando nodos enlazados

```
// añadir un elemento al final de la cola. O(1)
void push(T const& elem) {
    Nodo* nuevo = new Nodo(elem);

    if (ult != nullptr)
        ult->sig = nuevo;
    ult = nuevo;
    if (prim == nullptr) // la cola estaba vacía
        prim = nuevo;
    ++nelems;
}

// consultar el primero de la cola. O(1)
T const& front() const {
    if (empty())
        throw std::domain_error("la cola vacia no tiene primero");
    return prim->elem;
}
```

TAD queue usando nodos enlazados

```
    // eliminar el primero de la cola. O(1)
    void pop() {
        if (empty())
            throw std::domain_error("eliminando de una cola vacia");
        Nodo* a_borrar = prim;
        prim = prim->sig;
        if (prim == nullptr) // la cola se ha quedado vacía
            ult = nullptr;
        delete a_borrar;
        --nelems;
    }

    // consultar si la cola está vacía. O(1)
    bool empty() const {
        return nelems == 0;
    }

    // consultar el tamaño de la cola. O(1)
    int size() const {
        return nelems;
    }
};
```

Conclusiones

- Todas las operaciones tienen complejidad constante.
- La representación con array circular también permite implementar todas las operaciones en $\mathcal{O}(1)$ con el matiz de las redimensiones en la operación push (igual que en el TAD stack).
- Ambas representaciones se utilizan en la práctica.
- Una representación alternativa consiste en utilizar un *nodo fantasma*: un nodo extra al principio de la lista enlazada que hace de *cabecera* especial y que no guarda ningún elemento.
- Lo vemos a continuación en el TAD dequeue.

Colas dobles - El TAD dequeue

Colas dobles - El TAD dequeue

- Las colas dobles son una generalización de las colas que permiten insertar, acceder y extraer por ambos extremos.
- Operaciones básicas:
  - Constructora: Genera una cola doble vacía.
  - push_back: Añade un nuevo elemento al final.
  - push_front: Añade un nuevo elemento al principio.
  - pop_front: Modificadora parcial que elimina el primer elemento de la cola. Lanza excepción si la cola está vacía.
  - pop_back: Modificadora parcial que elimina el último elemento de la cola. Lanza excepción si la cola está vacía.
  - front : Observadora parcial que devuelve el primer elemento de la cola. Lanza excepción si la cola está vacía.
  - back: Observadora parcial que devuelve el último elemento de la cola. Lanza excepción si la cola está vacía.
  - empty: Observadora que permite averiguar si la cola tiene elementos.
  - size : Observadora que devuelve el número de elementos de la cola.

Colas dobles - El TAD dequeue

- De nuevo se podría optar por utilizar un array circular o bien nodos enlazados.
- Representación usando nodos enlazados:
  - Como en el TAD queue es necesario guardar un puntero tanto al primero como al último.
  - Sin embargo, no es suficiente con tener enlaces simples si queremos tener complejidad $\mathcal{O}(1)$ en la operación pop_back.
  - Los nodos tendrán por tanto doble enlace: al nodo siguiente (atributo sig) y al anterior (atributo ant).
  - Además, usaremos en este caso (por comodidad) una lista con nodo cabecera (fantasma) y *circular* (el último/primer nodo tiene como siguiente/anterior al nodo cabecera).

Colas dobles - El TAD dequeue

```
template <class T>
class deque {
protected:

    // puntero al nodo fantasma
    Nodo* fantasma;

    // número de elementos
    int nelems;

public:

    // constructor: cola doble vacía
    deque() : fantasma(new Nodo()), nelems(0) {
        fantasma->sig = fantasma->ant = fantasma; // circular
    }

    // destructor
    ~deque() {
        libera();
    }
```

Colas dobles - El TAD dequeue

- Haremos uso de estas dos operaciones privadas:

```
protected:
    // insertar un nuevo nodo entre anterior y siguiente
    Nodo* inserta_elem(T const& e, Nodo* anterior, Nodo* siguiente){
        Nodo* nuevo = new Nodo(e, anterior, siguiente);
        anterior->sig = nuevo;
        siguiente->ant = nuevo;
        ++nelems;
        return nuevo;
    }

    // eliminar el nodo n
    void borra_elem(Nodo* n) {
        assert(n != nullptr);
        n->ant->sig = n->sig;
        n->sig->ant = n->ant;
        delete n;
        --nelems;
    }
```

Colas dobles - El TAD dequeue

```
public:
    // añadir un elemento por el principio. O(1)
    void push_front(T const& e) {
        inserta_elem(e, fantasma, fantasma->sig);
    }

    // añadir un elemento por el final. O(1)
    void push_back(T const& e) {
        inserta_elem(e, fantasma->ant, fantasma);
    }

    // consultar el primer elemento de la cola. O(1)
    T const& front() const {
        if (empty())
          throw std::domain_error("la cola vacia no tiene primero");
        return fantasma->sig->elem;
    }

    // consultar el último elemento de la cola. O(1)
    T const& back() const {
        if (empty())
           throw std::domain_error("la cola vacia no tiene ultimo");
        return fantasma->ant->elem;
    }
```

Colas dobles - El TAD dequeue

```
    // eliminar el primer elemento. O(1)
    void pop_front() {
        if (empty())
            throw domain_error("eliminando primero de cola vacia");
        borra_elem(fantasma->sig);
    }

    // elminar el último elemento. O(1)
    void pop_back() {
        if (empty())
            throw domain_error("eliminando ultimo de cola vacia");
        borra_elem(fantasma->ant);
    }

    // consultar si la cola está vacía. O(1)
    bool empty() const {
        return nelems == 0;
    }

    // consultar el tamaño de la cola doble. O(1)
    int size() const { return nelems; }
};
```

Listas - El TAD list

Listas - El TAD list

- Las listas son una generalización de las colas dobles que a parte de permitir la consulta/inserción/eliminación por los dos extremos, también permiten acceder a cualquier punto intermedio tanto para consultar como para eliminar e insertar en él.
- Partiremos por tanto de la interfaz e implementación de las colas dobles mediante herencia:

```
template <class T>
class list : public deque<T> {
    ...
};
```

La operación at

- Para poder acceder a puntos intermedios de la lista una primera idea consistiría en proporcionar la operación `at` (análoga a la operación del mismo nombre en el TAD `vector`) que nos devuelva una referencia al elemento en la posición i-ésima de la lista.

```
/**
Devuelve el elemento i-ésimo de la lista (el primero es el 0 y el
último el size()-1). Lanza excepción si se proporciona un índice
no válido. O(n), con n=nelems
*/
T& at(unsigned int idx) const {
     if (idx >= nelems)
          throw std::out_of_range("Indice no valido");
     Nodo* aux = prim;
     for (int i = 0; i < idx; ++i)
          aux = aux->sig;
     return aux->elem;
}
```

La operación at

- Sin embargo un uso descuidado de esta operación podría llevar a situaciones no deseadas en cuanto a ineficiencia.
- Por ejemplo, consideremos este bucle aparentemente inofensivo para escribir uno a uno todos los elementos de la lista:

```
list<int> l;

...

for (int i = 0; i < l.size(); ++i)
    cout << l.at(i) << endl;
```

- El coste *no* sería lineal sino **cuadrático!**
- Por esta razón algunas implementaciones reales de las listas (como es el caso de la clase `list` de la STL de C++) de hecho no incluyen esta operación.

Iteradores

- La solución estándar para recorrer estructuras de datos y para acceder a (e insertar y eliminar en) puntos intermedios son los *iteradores*.
- Entenderemos un iterador como un objeto de una clase que:
  - Representa un punto intermedio en el recorrido de una colección de datos.
  - Tiene un método que devuelve el elemento por el que va el recorrido $\Rightarrow$ En C++ el operador $*$.
  - Tiene un método para avanzar al siguiente $\Rightarrow$ En C++ el operador ++.
  - Tiene implementada la operación de comparación, de forma que se puede saber si dos iteradores son iguales. Esto permite detectar el final de los recorridos (comparando con un iterador apuntando al final).
- Implementaremos la clase `iterator` (y `const_iterator` para recorridos observadores) dentro de la clase `list`.

Iteradores

Extendemos el TAD list para que proporcione dos operaciones adicionales:

- begin(): devuelve un iterador apuntando al primero del recorrido.
- end(): devuelve un iterador apuntando *fuera* del recorrido (al final), es decir un iterador cuya operación $*$ *falla*.

Así recorreríamos una lista para imprimirla:

```
list<int> l;
...
for (list<int>::iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl;
}
```

Desde *C++11* podemos usar los bucles *range-based for*:

```
list<int> l;
...
for (int e : l) {
    cout << e << endl;
}
```

Iteradores

- Implementaremos una clase Iterador, interna a list y protegida, que será instanciada correspondientemente para los iteradores constante y no-constante en const_iterator e iterator respectivamente.
- Atributos: un puntero al nodo actual en el recorrido (act) y otro al nodo fantasma fan (para saber cuándo estamos fuera del recorrido).

```
protected: // Estamos dentro de class list
    using Nodo = typename deque<T>::Nodo;

    template <class Apuntado>
    class Iterador {
        // puntero al nodo actual del recorrido
        Nodo* act;
        // puntero al nodo fantasma (para saber cuándo estamos fuera)
        Nodo* fan;

    public:

        Iterador() : act(nullptr), fan(nullptr) {}
```

Iteradores

```
// para acceder al elemento apuntado
Apuntado& operator*() const {
  if (act == fan) throw std::out_of_range("fuera de la lista");
  return act->elem;
}

Iterador& operator++() { // ++ prefijo (recomendado)
  if (act == fan) throw std::out_of_range("fuera de la lista");
  act = act->sig;
  return *this;
}

bool operator==(Iterador const& that) const {
  return act == that.act && fan == that.fan;
}

bool operator!=(Iterador const& that) const {
  return !(*this == that);
}
```

Iteradores

```
private:
  friend class list; // para que list pueda construir

  // constructora privada
  Iterador(Nodo* ac, Nodo* fa) : act(ac), fan(fa) {}
};// Fin de la clase interna Iterador

public: // de la clase list
  /* Iterador que permite recorrer la lista pero no modificar. */
  using const_iterator = Iterador<T const>;

  // devuelve un iterador constante al principio de la lista
  const_iterator cbegin() const {
    return const_iterator(this->fantasma->sig, this->fantasma);
  }

  // devuelve un iterador constante al final del recorrido (fuera)
  const_iterator cend() const {
    return const_iterator(this->fantasma, this->fantasma);
  }
```

Iteradores

```
  /* Iterador que permite recorrer la lista y modificar.*/
  using iterator = Iterador<T>;

  // devuelve un iterador al principio de la lista
  iterator begin() {
    return iterator(this->fantasma->sig, this->fantasma);
  }

  // devuelve un iterador al final del recorrido y fuera de este
  iterator end() {
    return iterator(this->fantasma, this->fantasma);
  }
};
```

Observa que con este iterador podemos alterar la lista. Por ej.:

```
for (int& e : l) e++; // Incrementa todos los elementos
```

En este caso es necesario el & para indicar que cada elemento del recorrido se debe capturar en la variable *e* por referencia.

Operación `insert`

- Extendamos el TAD list para permitir insertar elementos en medio de la lista.
- La operación recibirá un iterador situado en el punto de la lista donde se desea insertar un elemento y el elemento a insertar.

```
// Inserta un elemento delante del apuntado por el iterador it
// (it puede estar apuntado detrás del último)
// devuelve un iterador al nuevo elemento
iterator insert(iterator const& it, T const& elem) {
    Nodo* nuevo = this->inserta_elem(elem, it.act->ant, it.act);
    return iterator(nuevo, this->fantasma);
}
```

- Si insertamos un elemento a partir del iterador begin() el nuevo elemento pasa a ser el primero de la lista.
- Si insertamos delante de end(), el elemento insertado será el nuevo último elemento de la lista.
- La operación devuelve un iterador apuntando al nodo del elemento insertado.

Operación `insert`

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(list<int>& lista) {
    list<int>::iterator it = lista.begin();
    while (it != lista.end()) {
        lista.insert(it, *it);
        ++it;
    }
}
```

Operación erase

- También queremos extender el TAD list para permitir borrar elementos internos de la lista.
- La operación recibe un iterador situado en el punto de la lista que se desea borrar.

```
// elimina el elemento apuntado por el iterador (debe haber uno)
// devuelve un iterador al siguiente elemento al borrado
iterator erase(iterator const& it) {
    if (it.act == this->fantasma)
        throw std::out_of_range("fuera de la lista");
    iterator next(it.act->sig, this->fantasma);
    this->borra_elem(it.act);
    return next;
}
```

- 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 para continuar el recorrido.

Operación erase

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

```
void quitaPares(list<int>& l) {
    for (list<int>::iterator it = l.begin(); it != l.end(); ++it) {
        if ((*it)%2 == 0)
            it = l.erase(it); //Ojo! Continua con iterador devuelto
        else
            ++it;
    }
}
```