ESTRUCTURAS DE DATOS LINEALES

LISTAS

Listas

- Una **lista** es una estructura de datos lineal que contiene una secuencia de elementos, diseñada para realizar inserciones, borrados y accesos en cualquier posición
- La representaremos como $<a_1, a_2, \ldots, a_n>$
- Operaciones básicas:
  - Set: modifica el elemento de una posición
  - Get: devuelve el elemento de una posición
  - Borrar: elimina el elemento de una posición
  - Insertar: inserta un elemento en una posición
  - Num_elementos: devuelve el número de elementos de la lista
- En una lista con n elementos consideraremos n+1 posiciones, incluyendo *la siguiente a la última*, que llamaremos **fin de la lista**

Listas. Primera aproximación

Esquema de la interfaz

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef char Tbase;

class Lista{
private:
  ...              //La implementación que se elija
public:
  Lista();
  Lista(const Lista& l);
  ~Lista();
  Lista& operator=(const Lista& l);

  Tbase get(int pos) const;
  void set(int pos, Tbase e);
  void insertar(int pos, Tbase e);
  void borrar(int pos);
 int num_elementos() const;
};

#endif // __LISTA_H__
```

Listas. Posibles implementaciones

- **Vectores**. A priori sencilla: las posiciones que se pasan a los métodos son enteras y se traducen directamente en índices del vector. Inserciones y borrados ineficientes (orden lineal)

- **Celdas enlazadas**. Parece más eficiente: inserciones y borrados no desplazan elementos. Los métodos set, get, insertar y borrar tienen orden lineal. El problema son las posiciones enteras

- Conclusión: la implementación de las posiciones debe variar en función de la implementación de la lista

Listas. Posiciones

- Vamos a crear una abstracción de las posiciones, encapsulando el concepto de posición en una clase.
- Crearemos una clase **Posicion**. Un objeto de la clase representa una posición en la lista.
  - En el caso del vector, se implementa como un entero
  - En el caso de las celdas enlazadas, será un puntero
- Observaciones:
  - Para una lista de tamaño n, habrá n+1 posiciones posibles
  - El movimiento entre posiciones se hace una a una
  - La comparación entre posiciones se limita a igualdad y desigualdad (no existe el concepto de anterior o posterior)

Listas. Clases Posicion y Lista

Esquema de la interfaz

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef char Tbase;

class Posicion{
private:
  ...              //La implementación que se elija

public:
  Posicion();
  Posicion(const Posicion& p);
  ~Posicion();
  Posicion& operator=(const Posicion& p);
  Posicion& operator++();
  Posicion& operator++(int);
  Posicion& operator--();
  Posicion& operator--(int);
  bool operator==(const Posicion& p);
  bool operator!=(const Posicion& p);
};
```

Listas. Clases Posicion y Lista

```
class Lista{
private:
  ...                 //La implementación que se elija

public:
  Lista();
  Lista(const Lista& l);
  ~Lista();
  Lista& operator=(const Lista& l);

  Tbase get(Posicion p) const;
  void set(Posicion p, Tbase e);
  Posicion insertar(Posicion p, Tbase e);
  Posicion borrar(Posicion p);
  Posicion begin() const;
  Posicion end() const;
};
#endif // __LISTA_H__
```

Esquema de la interfaz

num_elementos no es fundamental

Se modifican

Necesitamos saber dónde empieza y acaba la lista

begin() devuelve la posición del primer elemento
end() devuelve la posición posterior al último elemento (permite añadir al final)
En una lista vacía, begin() coincide con end()

Listas

Uso de una lista

```
#include <iostream>
#include "Lista.hpp"
using namespace std;
int main() {
  char dato;
  Lista l;
  cout << "Escriba una frase" << endl;
  while((dato=cin.get())!='\n')
    l.insertar(l.end(), dato);
  cout << "La frase introducida es:" << endl;
  escribir(l);
  cout << "La frase en minúsculas:" << endl;
  escribir_minuscula(l);
  if(localizar(l,' ')==l.end())
    cout << "La frase no tiene espacios" << endl;
  else{
    cout << "La frase sin espacios:" << endl;
    Lista aux(l);
    borrar_caracter(aux, ' ');
    escribir(aux);
  }
  cout << "La frase al revés: " << endl;
  escribir(al_reves(l));
  cout << (palindromo(l)? "Es ":"No es ") << "un palíndromo" << endl;
  return 0;
}
```

Listas

```
bool vacia(Lista& l){
  return(l.begin()==l.end());
}

int numero_elementos(const Lista& l){
  int n=0;
  for(Posicion p=l.begin(); p!=l.end(); ++p)
    n++;
  return n;
}

void todo_minuscula(Lista& l){
  for(Posicion p=l.begin(); p!=l.end(); ++p)
    l.set(p, tolower(l.get(p)));
}

void escribir(const Lista& l){
  for(Posicion p=l.begin(); p!=l.end(); ++p)
    cout << l.get(p);
  cout << endl;
}

void escribir_minuscula(Lista l){
  todo_minuscula(l);
  escribir(l);
}
```

Uso de una lista

Listas

```
void borrar_caracter(Lista&l, char c){
  Posicion p = l.begin();
  while(p != l.end())
    if(l.get(p) == c)
      p = l.borrar(p);
    else
      ++p;
}

Lista al_reves(const Lista& l){
  Lista aux;
  for(Posicion p=l.begin(); p!=l.end(); ++p)
    aux.insertar(aux.begin(), l.get(p));
  return aux;
}

Posicion localizar(const Lista& l, char c){
  for(Posicion p=l.begin(); p!=l.end(); ++p)
    if(l.get(p)==c)
      return(p);
  return l.end();
}
```

Uso de una lista

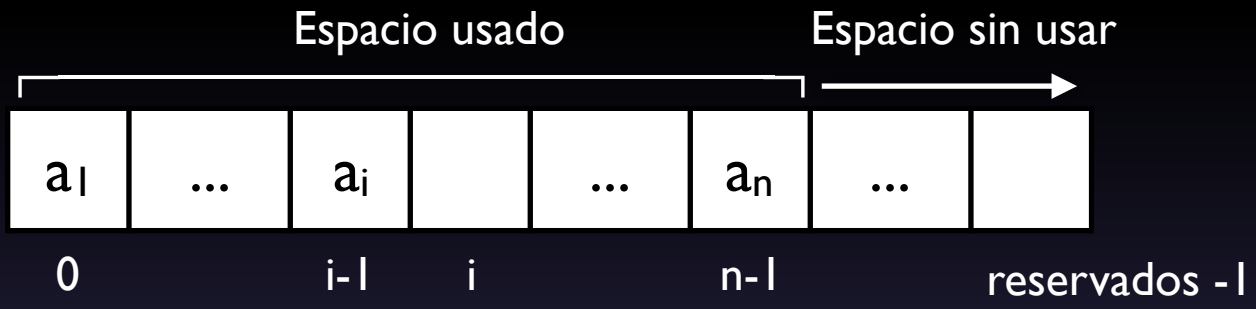
Listas

```
bool palindromo(const Lista& l){
  Lista aux(l);
  int n = numero_elementos(l);
  if(n<2)
    return true;
  borrar_caracter(aux, ' ');
  todo_minuscula(aux);

  Posicion p1, p2;
  p1 = aux.begin();
  p2 = aux.end();
  --p2;
  for(int i=0; i<n/2; i++){
    if(aux.get(p1) != aux.get(p2))
      return false;
    ++p1;
    --p2;
  }
  return true;
}
```

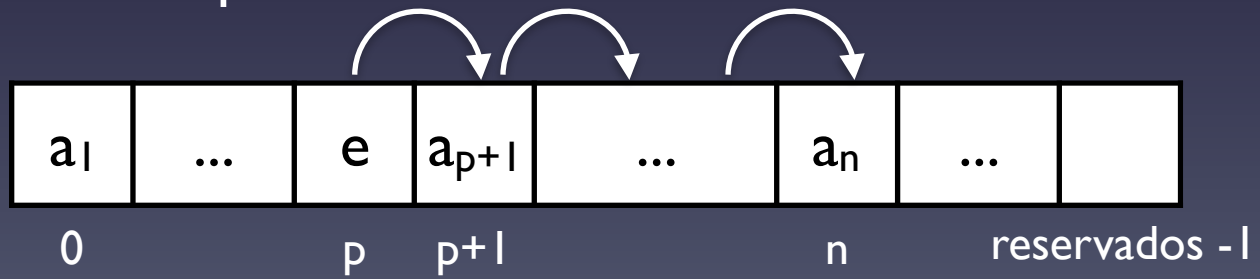
Uso de una lista

Listas. Implementación con vectores

- Almacenamos la secuencia de valores en un vector. Las posiciones son enteros



- La posición begin() corresponde al 0
- La posición end() corresponde a n (después del último)
- Las inserciones suponen desplazar elementos a la derecha y los borrados, a la izquierda



Lista.h

```
#ifndef __LISTA_H__
#define __LISTA_H__

#include <stdio.h>

typedef char Tbase;

class Lista;

class Posicion{
private:
  int i;
public:
  Posicion();
//  Posicion(const Posicion& p);
//  ~Posicion();
//  Posicion& operator=(const Posicion& p);
  Posicion& operator++();
  Posicion operator++(int);
  Posicion& operator--();
  Posicion operator--(int);
  bool operator==(const Posicion& p);
  bool operator!=(const Posicion& p);
  friend class Lista;
};
```

Lista.h

```
class Lista{
private:
  Tbase* datos;
  int nelementos;
  int reservados;

public:
  Lista();
  Lista(const Lista& l);
  ~Lista();
  Lista& operator=(const Lista& l);

  void set(Posicion p, Tbase e);
  Tbase get(Posicion p) const;
  Posicion insertar(Posicion p, Tbase e);
  Posicion borrar(Posicion p);
  Posicion begin() const;
  Posicion end() const;

private:
  void resize(int n);
  void copiar(const Lista& l);
};

#endif // __LISTA_H__
```

Lista.cpp

```
#include <cassert>
#include "Lista.hpp"

using namespace std;

//Clase Posicion

Posicion::Posicion(){
  i = 0;
}

Posicion& Posicion::operator++(){
  ++i;
  return *this;
}

Posicion Posicion::operator++(int){
  Posicion aux;
  aux.i = i++;
  return aux;
}
```

```
Posicion& Posicion::operator--(){
  --i;
  return  *this;
}

Posicion Posicion::operator--(int){
  Posicion aux;
  aux.i = i--;
  return aux;
}

bool Posicion::operator==(const
Posicion& p){
  return i==p.i;
}

bool Posicion::operator!=(const
Posicion& p){
  return i!=p.i;
}
```

Lista.cpp

```
Lista::Lista(){
  nelementos = 0;
  reservados = 1;
  datos = new Tbase[1];
}

Lista::Lista(const Lista& l){
  copiar(l);
}

Lista::~Lista(){
  delete[] datos;
}

Lista& Lista::operator=(const Lista &l){
  delete[] datos;
  copiar(l);
  return *this;
}

void Lista::copiar(const Lista& l){
  nelementos = l.nelementos;
  reservados = l.reservados;
  datos = new Tbase[reservados];
  for(int i=0; i<nelementos; i++)
    datos[i] = l.datos[i];
}
```

Lista.cpp

```
Posicion Lista::insertar(Posicion p, Tbase e){
  if(nelementos == reservados)
    resize(reservados*2);
  for(int j=nelementos; j>p.i; j--)
    datos[j] = datos[j-1];
  datos[p.i] = e;
  nelementos++;
  return p;
}

Posicion Lista::borrar(Posicion p){
  assert(p!=end());
  for(int j=p.i;j<nelementos-1; j++)
    datos[j] = datos[j+1];
  nelementos--;
  if(nelementos<reservados/4)
    resize(reservados/2);
  return p;
}

void Lista::set(Posicion p, Tbase e){
  assert(p.i>=0 && p.i<nelementos);
  datos[p.i] = e;
}
```

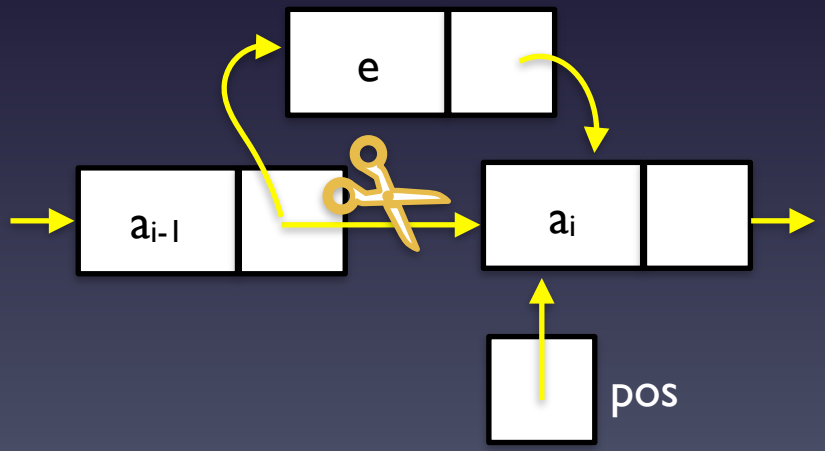
Lista.cpp

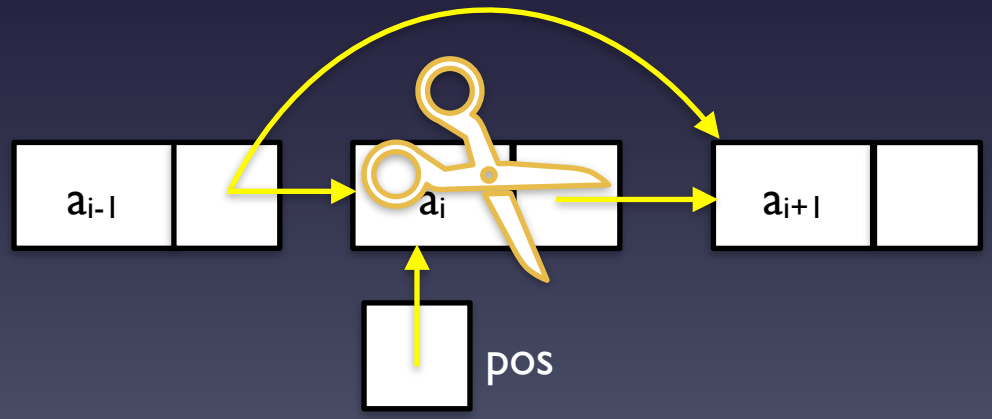
```
Tbase Lista::get(Posicion p) const{
  assert(p.i>=0 && p.i<nelementos);
  return datos[p.i];
}

Posicion Lista::begin() const{
  Posicion p;
  p.i = 0;
  return p;
}

Posicion Lista::end() const{
  Posicion p;
  p.i = nelementos;
  return p;
}

void Lista::resize(int n){
  assert(n>0 && n>nelementos);
  Tbase* aux = new Tbase[n];
  for(int i=0; i<nelementos; i++)
    aux[i] = datos[i];
  delete[] datos;
  datos = aux;
  reservados = n;
}
```

Listas. Celdas enlazadas

Almacenamos la secuencia de valores en celdas enlazadas

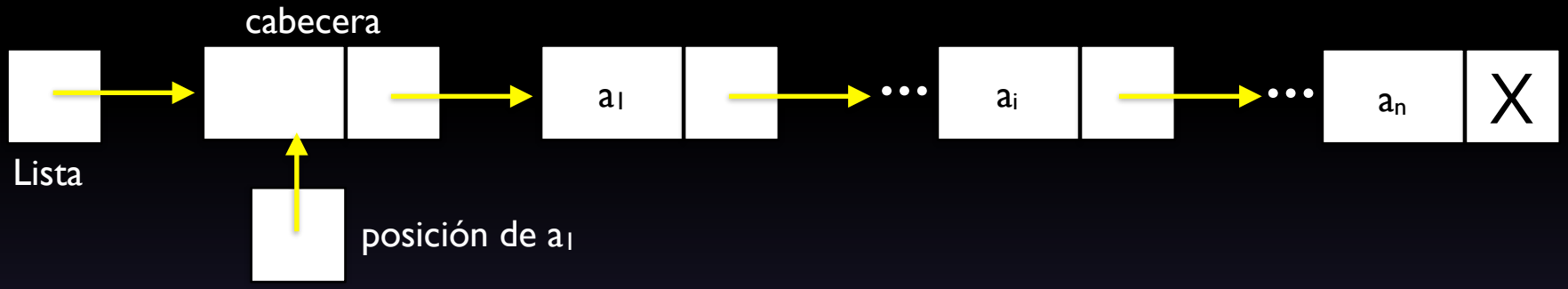
- Una lista es un puntero a la primera celda (si no está vacía)
- Una posición son dos punteros. El segundo (no se muestra) es necesario para algunos operadores
- Inserciones/borrados en la primera posición son casos especiales





Listas. Celdas enlazadas con cabecera

Almacenamos la secuencia de valores en celdas enlazadas



- Una lista es un puntero a la cabecera (si está vacía, sólo tiene una celda)
- Una posición son dos punteros. El segundo (no se muestra) es necesario para algunos operadores
- La posición de un elemento es un puntero a la celda anterior
- Inserciones/borrados la primera posición NO son casos especiales

Lista.h

```
#ifndef __LISTA_H__
#define __LISTA_H__
typedef char Tbase;

struct CeldaLista{
  Tbase elemento;
  CeldaLista* siguiente;
};
class Lista;

class Posicion{
private:
  CeldaLista* puntero;
  CeldaLista* primera;
public:
  Posicion();
  //Posicion(const Posicion& p);
  //~Posicion();
  //Posicion& operator=(const Posicion& p);
  Posicion& operator++();
  Posicion operator++(int);
  Posicion& operator--();
  Posicion operator--(int);
  bool operator==(const Posicion& p);
  bool operator!=(const Posicion& p);
  friend class Lista;
};
```

Lista.h

```
class Lista{
private:
  CeldaLista* cabecera;
  CeldaLista* ultima;
public:
  Lista();
  Lista(const Lista& l);
  ~Lista();
  Lista& operator=(const Lista& l);

  Tbase get(Posicion p) const;
  void set(Posicion p, Tbase e);
  Posicion insertar(Posicion p, Tbase e);
  Posicion borrar(Posicion p);
  Posicion begin() const;
  Posicion end() const;
};

#endif // __LISTA_H__
```

Lista.cpp

```
#include <cassert>
#include "Lista.hpp"

//Clase Posicion

Posicion::Posicion(){
  primera = puntero = 0;
}

Posicion& Posicion::operator++(){
  puntero = puntero->siguiente;
  return *this;
}

Posicion Posicion::operator++(int){
  Posicion p(*this);
  ++(*this);
  return p;
}

Posicion& Posicion::operator--(){
  assert(puntero!=primera);
  CeldaLista* aux = primera;
  while(aux->siguiente!=puntero){
    aux = aux->siguiente;
  }
  puntero = aux;
  return *this;
}

Posicion Posicion::operator--(int){
  Posicion p(*this);
  --(*this);
  return p;
}

bool Posicion::operator==(const Posicion & p){
  return(puntero==p.puntero);
}

bool Posicion::operator!=(const Posicion &p){
  return(puntero!=p.puntero);
}
```

Lista.cpp

```
//Clase Lista

Lista::Lista(){
  ultima = cabecera = new CeldaLista;
  cabecera->siguiente = 0;
}

Lista::Lista(const Lista& l){
  ultima = cabecera = new CeldaLista;
  CeldaLista* orig = l.cabecera;
  while(orig->siguiente!=0){
    ultima->siguiente = new CeldaLista;
    ultima = ultima->siguiente;
    orig = orig->siguiente;
    ultima->elemento = orig->elemento;
  }
  ultima->siguiente = 0;
}

Lista::~Lista(){
  CeldaLista* aux;
  while(cabecera!=0){
    aux = cabecera;
    cabecera = cabecera->siguiente;
    delete aux;
  }
}
```

Lista.cpp

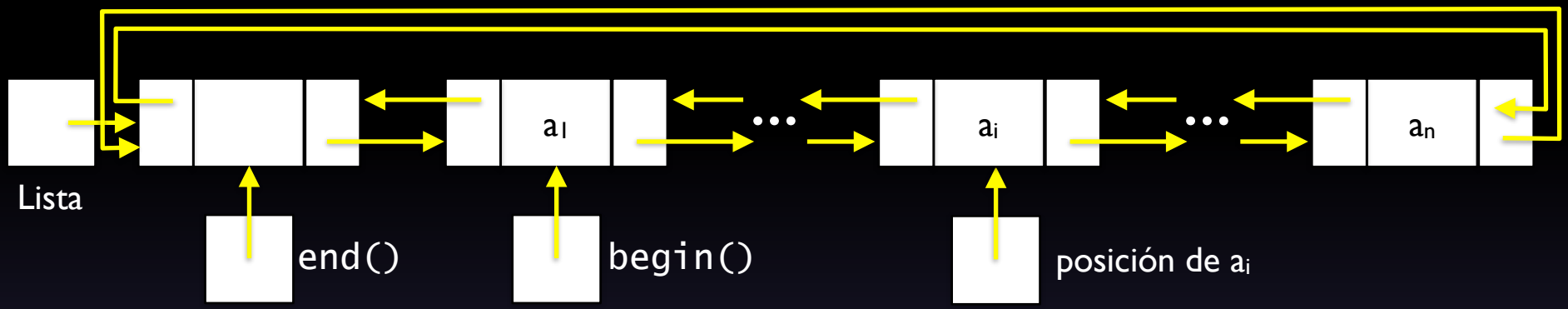
```
Lista& Lista::operator=(const Lista& l){
  Lista aux(l);
  intercambiar(cabecera, aux.cabecera);
  intercambiar(ultima, aux.ultima);
  return *this;
}

void Lista::set(Posicion p, Tbase e){
  p.puntero->siguiente->elemento = e;
}

Tbase Lista::get(Posicion p) const{
  return p.puntero->siguiente->elemento;
}

Posicion Lista::insertar(Posicion p, Tbase e){
  CeldaLista* nueva = new CeldaLista;
  nueva->siguiente = p.puntero->siguiente;
  p.puntero->siguiente = nueva;
  nueva->elemento = e;
  if(p.puntero == ultima)
    ultima = nueva;
  return p;
}
```

Lista.cpp

```
Posicion Lista::borrar(Posicion p){
  assert(p!=end());
  CeldaLista* aux = p.puntero->siguiente;
  p.puntero->siguiente = aux->siguiente;
  if(aux==ultima)
    ultima = p.puntero;
  delete aux;
  return p;
}

Posicion Lista::begin()const{
  Posicion p;
  p.puntero = p.primera = cabecera;
  return p;
}

Posicion Lista::end() const{
  Posicion p;
  p.puntero = ultima;
  p.primera = cabecera;
  return p;
}

void Lista::intercambiar(CeldaLista *p, CeldaLista *q){
  CeldaLista *aux;
  aux = p;
  p = q;
  q = aux;
}
```

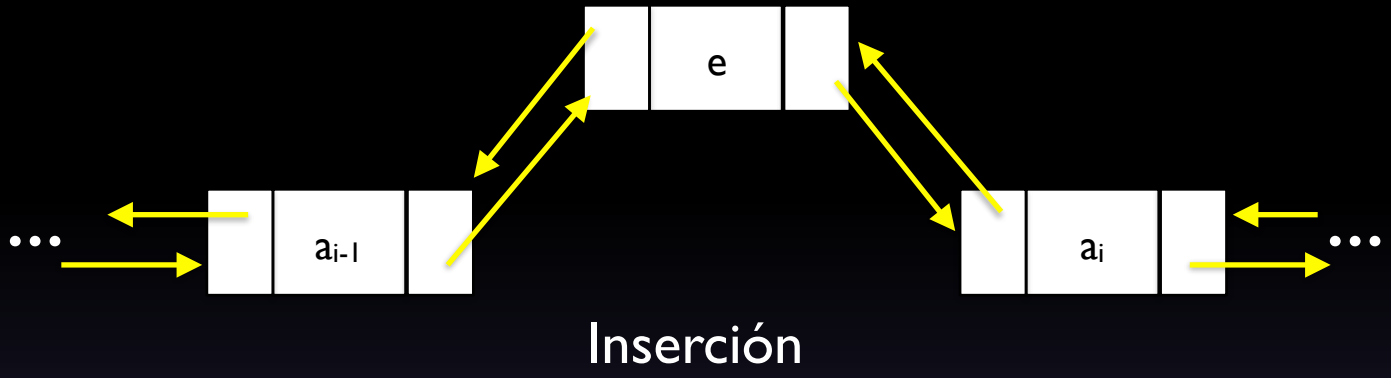
Listas. Celdas doblemente enlazadas circulares

Almacenamos la secuencia de valores en celdas doblemente enlazadas



- Una lista es un puntero a la cabecera (si está vacía, sólo tiene una celda)
- Una posición es un único puntero a la celda
- Inserciones/borrados son independientes de la posición

Listas. Celdas doblemente enlazadas circulares



Inserción

Borrado

Lista.h

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef char Tbase;
struct CeldaLista{
  Tbase elemento;
  CeldaLista* anterior;
  CeldaLista* siguiente;
};

class Lista;

class Posicion{
private:
  CeldaLista* puntero;
public:
  Posicion();
  //Posicion(const Posicion& p);
  //~Posicion();
  //Posicion& operator=(const Posicion& p);
  Posicion& operator++();
  Posicion operator++(int);
  Posicion& operator--();
  Posicion operator--(int);
  bool operator==(const Posicion& p);
  bool operator!=(const Posicion& p);
  friend class Lista;
};
```

Lista.h

```
class Lista{
private:
  CeldaLista* cab;
public:
  Lista();
  Lista(const Lista& l);
  ~Lista();
  Lista& operator=(const Lista& l);

  void set(Posicion p, Tbase e);
  Tbase get(Posicion p)const;
  Posicion insertar(Posicion p, Tbase e);
  Posicion borrar(Posicion p);
  Posicion begin() const;
  Posicion end() const;
};
#endif //__LISTA_H__
```

Lista.cpp

```
#include <cassert>
#include "Lista.hpp"

//Clase Posicion

Posicion::Posicion(){
  puntero = 0;
}

Posicion& Posicion::operator++(){
  puntero = puntero->siguiente;
  return *this;
}

Posicion Posicion::operator++(int){
  Posicion p(*this);
  ++(*this);
  return p;
}

bool Posicion::operator==(const Posicion& p){
  return (puntero==p.puntero);
}

bool Posicion::operator!=(const Posicion& p){
  return (puntero!=p.puntero);
}
```

```
Posicion& Posicion::operator--(){
  puntero = puntero->anterior;
  return *this;
}

Posicion Posicion::operator--(int){
  Posicion p(*this);
  --(*this);
  return p;
}
```

Lista.cpp

```
Lista::Lista(){
  cabecera = new CeldaLista;
  cabecera->siguiente = cabecera;
  cabecera->anterior = cabecera;
}

Lista::Lista(const Lista& l){
  cabecera = new CeldaLista;
  cabecera->siguiente = cabecera;
  cabecera->anterior = cabecera;

  CeldaLista* p = l.cabecera->siguiente;
  while(p!=l.cabecera){
    CeldaLista* q;
    q = new CeldaLista;
    q->elemento = p->elemento;
    q->anterior = cabecera->anterior;
    cabecera->anterior->siguiente = q;
    cabecera->anterior = q;
    q->siguiente = cabecera;
    p = p->siguiente;
  }
}

Lista::~Lista(){
  while(begin()!=end())
    borrar(begin());
  delete cabecera;
}
```

Lista.cpp

```
Lista& Lista::operator=(const Lista &l){
  Lista aux(l);
  CeldaLista* p;
  p = this->cabecera;
  this->cabecera = aux.cabecera;
  aux.cabecera = p;
  return *this;
}

void Lista::set(Posicion p, Tbase e){
  p.puntero->elemento = e;
}

Tbase Lista::get(Posicion p)const{
  return p.puntero->elemento;
}

Posicion Lista::insertar(Posicion p, Tbase e){
  CeldaLista* q = new CeldaLista;
  q->anterior = p.puntero->anterior;
  q->siguiente = p.puntero;
  p.puntero->anterior = q;
  q->anterior->siguiente = q;
  q->elemento = e;
  p.puntero = q;
  return p;
}
```

Lista.cpp

```
Posicion Lista::borrar(Posicion p){
  assert(p!=end());
  CeldaLista* q = p.puntero;
  q->anterior->siguiente = q->siguiente;
  q->siguiente->anterior = q->anterior;
  p.puntero = q->siguiente;
  delete q;
  return p;
}

Posicion Lista::begin() const{
  Posicion p;
  p.puntero = cabecera->siguiente;
  return p;
}

Posicion Lista::end() const{
  Posicion p;
  p.puntero = cabecera;
  return p;
}
```