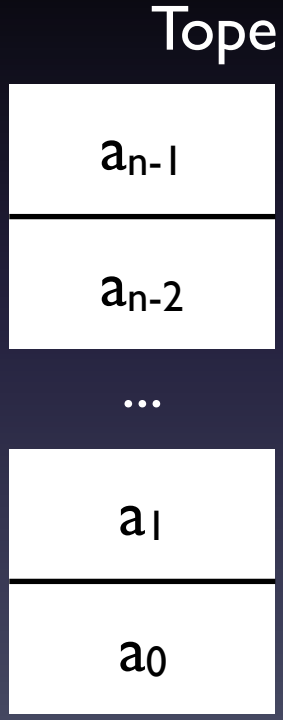
ESTRUCTURAS DE DATOS LINEALES

PILAS

Pilas

- Las estructuras de datos lineales se caracterizan porque consisten en una secuencia de elementos, a_0, a_1,..., a_n, dispuestos a lo largo de una dimensión
- Las pilas son un tipo de ED lineales que se caracterizan por su comportamiento LIFO (*Last In, First Out*): todas las inserciones y borrados se realizan en un extremo de la pila que llamaremos **tope**
- **Operaciones básicas:**
  - Tope: devuelve el elemento del tope
  - Poner: añade un elemento encima del tope
  - Quitar: quita el elemento del tope
  - Vacia: indica si la pila está vacía



Pilas

Esquema de la interfaz

```
#ifndef __PILA_H__
#define __PILA_H__

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

public:
  Pila();
  Pila(const Pila & p);
  ~Pila();
  Pila & operator=(const Pila &p);

  bool vacia() const;
  void poner(Tbase c);
  void quitar();
  Tbase tope() const;
};
#endif /* Pila_hpp */
```

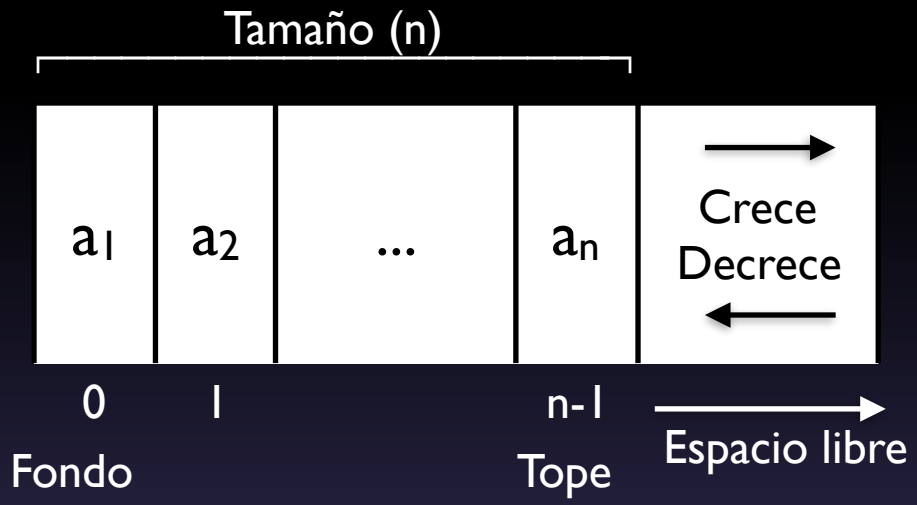
Pilas

Uso de una pila

```
#include <iostream>
#include "Pila.hpp"
using namespace std;

int main() {
  Pila p, q;
  char dato;

  cout << "Escriba una frase" << endl;
  while((dato=cin.get())!='\n')
    p.poner(dato);
  cout << "La escribimos del revés" << endl;
  while(!p.vacia()){
    cout << p.tope();
    q.poner(p.tope());
    p.quitar();
  }
  cout << endl << "La frase original era" << endl;
  while(!q.vacia()){
    cout << q.tope();
    q.quitar();
  }
  cout << endl;
  return 0;
}
```

Pilas. Implementación con vectores

Almacenamos la secuencia de valores en un vector



- El fondo de la pila está en la posición 0
- El número de elementos varía. Debemos almacenarlo
- Si insertamos elementos, el vector puede agotarse (tiene una capacidad limitada). Podemos resolverlo con memoria dinámica

Pila.h

```
#ifndef __PILA_H__
#define __PILA_H__

typedef char Tbase;
const int TAM =500;

class Pila{
private:
  Tbase datos[TAM];
  int nelem;

public:
  Pila();
  Pila(const Pila & p);
  ~Pila();
  Pila & operator=(const Pila &p);

  bool vacia() const;
  void poner(Tbase c);
  void quitar();
  Tbase tope() const;
private:
  void copiar(const Pila &p); //auxiliar
};

#endif /* Pila_hpp */
```

Pila.cpp

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

//No se incluyen constructores, destructor ni operador de asignación

bool Pila::vacia() const{
  return(nelem==0);
}

void Pila::poner(Tbase c){
  assert(nelem<TAM);
  datos[nelem] = c;
  nelem++;
}

void Pila::quitar(){
  assert(nelem>0);
  nelem--;
}

Tbase Pila::tope() const{
  assert(nelem>0);
  return datos[nelem-1];
}
```

- Ventaja: implementación muy sencilla
- Desventaja: limitaciones de la memoria estática. Se desperdicia memoria y puede desbordarse el espacio reservado
- Ejercicio propuesto: desarrollar el resto de métodos

Pila.h (vectores dinámicos)

```
#ifndef __PILA_H__
#define __PILA_H__

typedef char Tbase;
const int TAM =10;

class Pila{
private:
  Tbase *datos;
  int reservados;
  int nelem;
public:
  Pila();
  Pila(const Pila & p);
  ~Pila();
  Pila & operator=(const Pila &p);
  bool vacia() const;
  void poner(Tbase c);
  void quitar();
  Tbase tope() const;
private:
  void resize(int n);
};

#endif /* Pila_hpp */
```

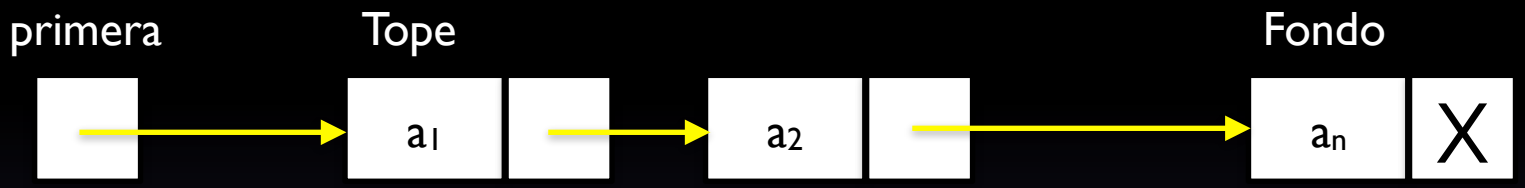
Pila.cpp (vectores dinámicos)

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

//No se incluyen constructores, destructor, resize ni operador =

bool Pila::vacia() const{
  return(nelem==0);
}

void Pila::poner(Tbase c){
  if (nelem==reservados)
    resize(2*reservados);
  datos[nelem] = c;
  nelem++;
}

void Pila::quitar(){
  assert(nelem>0);
  nelem--;
  if(nelem<reservados/4)
    resize(reservados/2);
}

Tbase Pila::tope() const{
  assert(nelem>0);
  return datos[nelem-1];
}
```

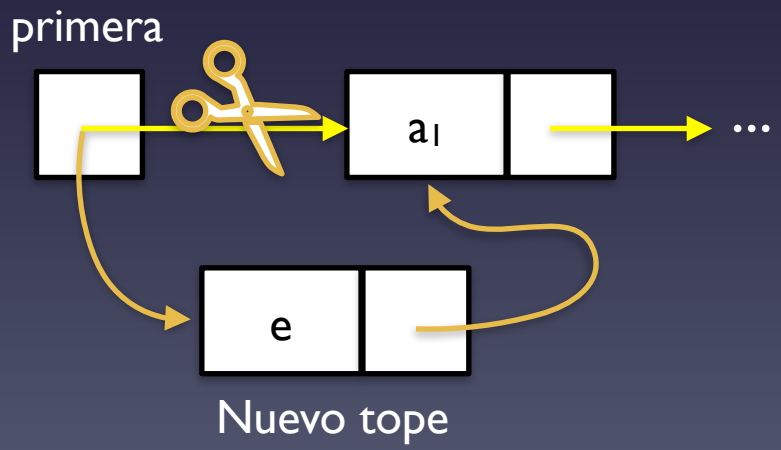
- Esta implementación es mucho más eficiente en cuanto a consumo de memoria
- Ejercicio propuesto: desarrollar el resto de métodos

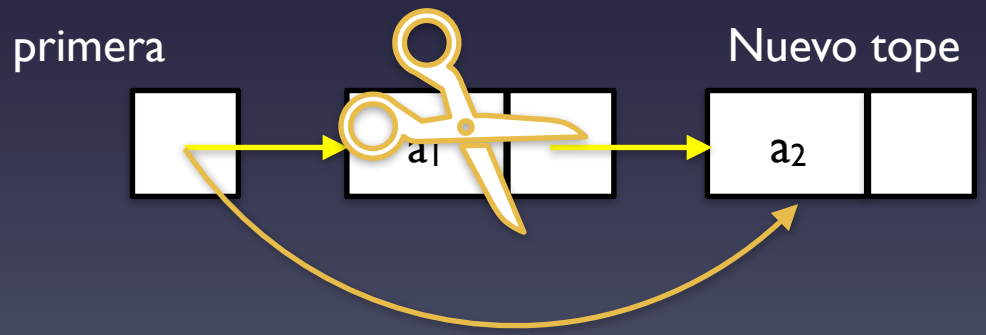
Pilas. Implementación con listas

Almacenamos la secuencia de valores en celdas enlazadas



- Una pila vacía tiene un puntero (primera) nulo
- El tope de la pila está en la primera celda (muy eficiente)
- La inserción y borrado de elementos se hacen sobre la primera celda





Pila.h

```
#ifndef __PILA_H__
#define __PILA_H__

typedef char Tbase;

struct CeldaPila{
  Tbase elemento;
  CeldaPila * sig;
};

class Pila{
private:
  CeldaPila * primera;
public:
  Pila();
  Pila(const Pila& p);
  ~Pila();
  Pila& operator=(const Pila& p);
  bool vacia() const;
  void poner(Tbase c);
  void quitar();
  Tbase tope() const;
private:
  void copiar(const Pila& p);
  void liberar();
};

#endif // Pila_hpp
```

Pila.cpp

```
#include "Pila.hpp"

Pila::Pila(){
  primera = 0;
}

Pila::Pila(const Pila& p){
  copiar(p);
}

Pila::~Pila(){
  liberar();
}

Pila& Pila::operator=(const Pila &p){
  if(this!=&p)
    liberar();
  copiar(p);
  return *this;
}
```

```
void Pila::poner(Tbase c){
  CeldaPila *aux=new CeldaPila;
  aux->elemento = c;
  aux->sig = primera;
  primera = aux;
}

void Pila::quitar(){
  CeldaPila *aux = primera;
  primera = primera->sig;
  delete aux;
}

Tbase Pila::tope() const{
  return primera->elemento;
}

bool Pila::vacia() const{
  return (primera==0);
}
```

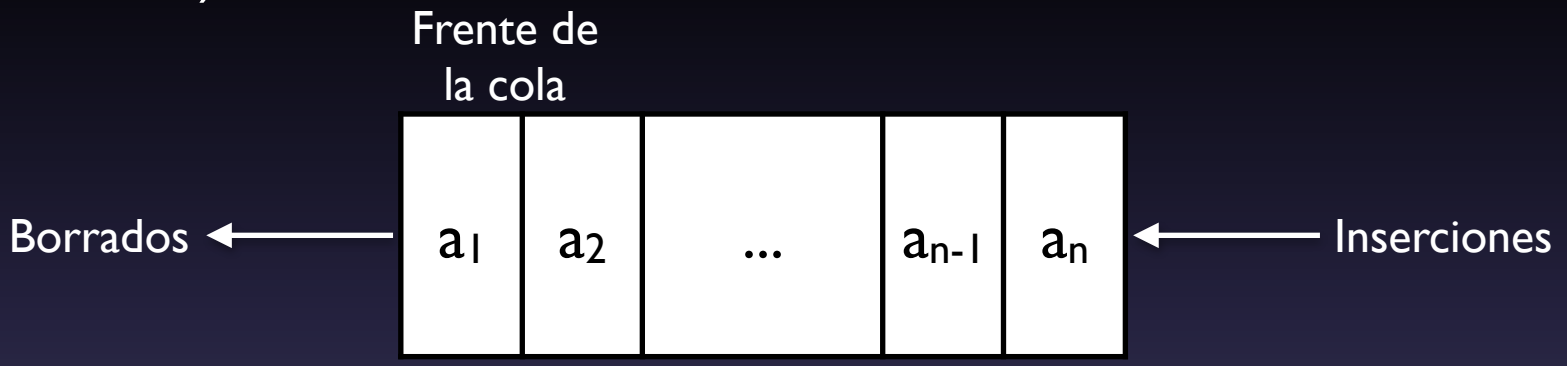
Pila.cpp

```
void Pila::copiar(const Pila &p){
  if (p.primera==0)
    primera = 0;
  else{
    primera = new CeldaPila;
    primera->elemento = p.primera->elemento;
    CeldaPila *orig = p.primera, *dest=primera;
    while(orig->sig!=0){
      dest->sig = new CeldaPila;
      orig = orig->sig;
      dest = dest->sig;
      dest->elemento = orig->elemento;
    }
    dest->sig = 0;
  }
}

void Pila::liberar(){
  CeldaPila* aux;
  while(primera!=0){
    aux = primera;
    primera = primera->sig;
    delete aux;
  }
  primera = 0;
}
```

COLAS

Colas

- Una cola es una estructura de datos lineal en la que los elementos se insertan y borran por extremos opuestos
- Se caracterizan por su comportamiento FIFO (*First In, First Out*)



- **Operaciones básicas:**
  - Frente: devuelve el elemento del frente
  - Poner: añade un elemento al final de la cola
  - Quitar: elimina el elemento del frente
  - Vacia: indica si la cola está vacía

Colas

Esquema de la interfaz

```
#ifndef __COLA_H__
#define __COLA_H__

typedef char Tbase;

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

public:
  Cola();
  Cola(const Cola& c);
  ~Cola();
  Cola& operator=(const Cola& c);

  bool vacia() const;
  void poner(const Tbase valor);
  void quitar();
  Tbase frente() const;
};

#endif // __COLA_H__
```

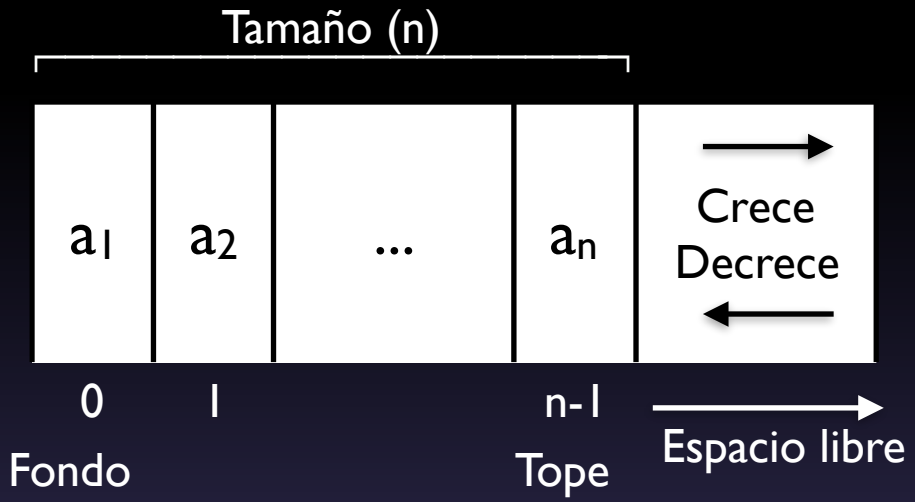
Colas

Uso de una cola

```
#include <iostream>
#include "Pila.hpp"
#include "Cola.hpp"
using namespace std;

int main() {
  Pila p;
  Cola c;
  char dato;
  cout << "Escriba una frase" <<endl;
  while((dato=cin.get()) !='\n')
    if (dato != ' '){
      p.poner(dato);
      c.poner(dato);
    }
  bool palindromo = true;
  while(!p.vacia() && palindromo){
    if(c.frente() != p.tope())
      palindromo = false;
    p.quitar();
    c.quitar();
  }
  cout <<"La frase "<<(palindromo?"es":"no es") <<" un palíndromo" <<endl;

  return 0;
}
```

Colas. Implementación con vectores

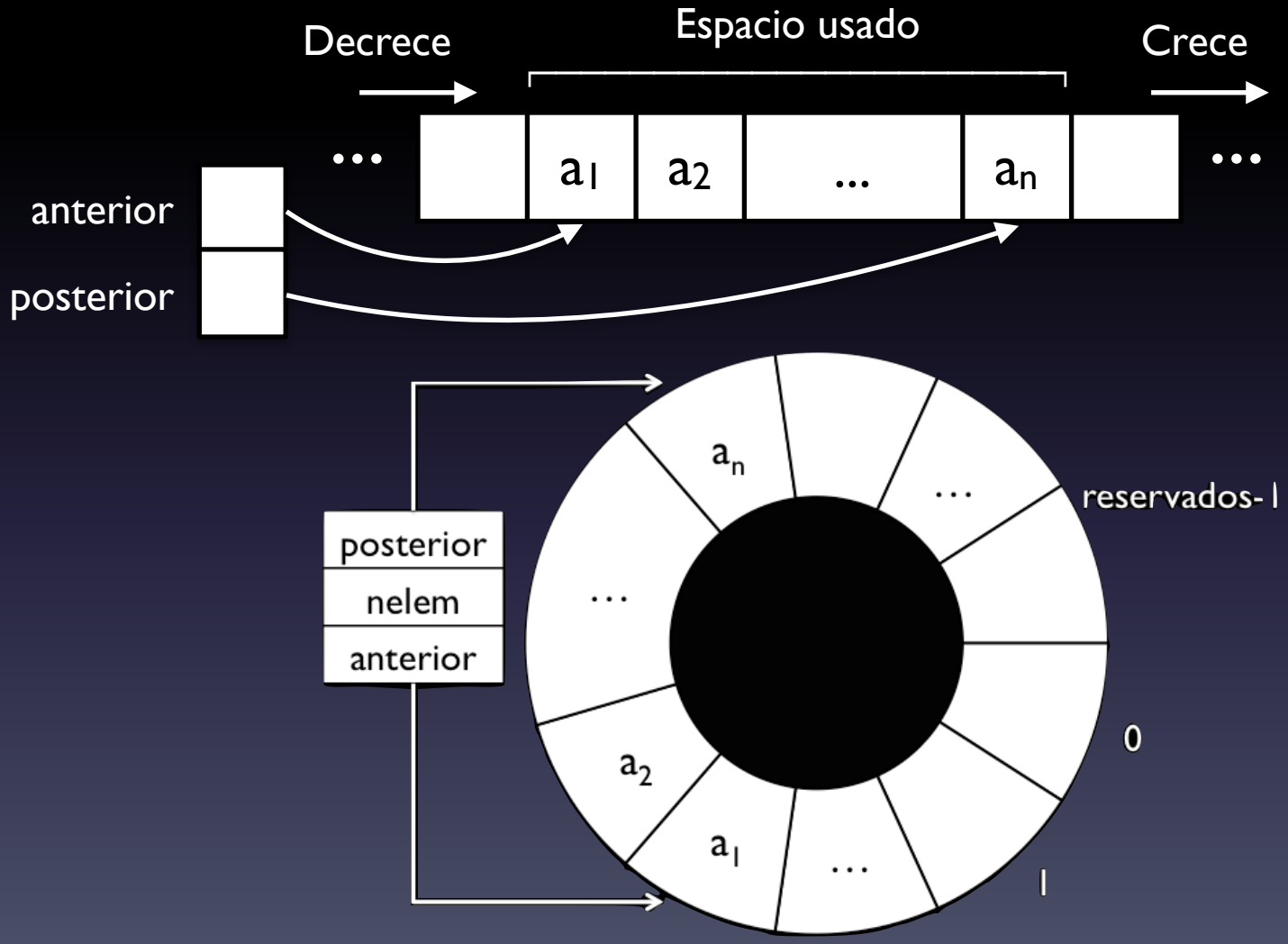
Almacenamos la secuencia de valores en un vector



- El fondo de la pila está en la posición 0
- El número de elementos varía. Debemos almacenarlo
- Si insertamos elementos, el vector puede agotarse (tiene una capacidad limitada). Podemos resolverlo con memoria dinámica

Colas. Implementación con vectores circulares

- Almacenamos la secuencia de valores en un vector



Cola.h

```
#ifndef __COLA_H__
#define __COLA_H__

typedef char Tbase;

class Cola{
private:
  Tbase * datos;
  int reservados;
  int nelem;
  int anterior, posterior;
public:
  Cola();
  Cola(const Cola& c);
  ~Cola();
  Cola& operator=(const Cola& c);
  bool vacia() const;
  void poner(const Tbase valor);
  void quitar();
  Tbase frente() const;
private:
  void resize(const int n);
  void copiar(const Cola& c);
  void liberar();
};

#endif // __COLA_H__
```

Cola.cpp

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

Cola::Cola(){
  datos = new Tbase[1];
  reservados = 1;
  anterior = posterior = 0;
  nelem = 0;
}

Cola::Cola(const Cola& c){
  copiar(c);
}

Cola& Cola::operator=(const Cola& c){
  if(this!=&c){
    liberar();
    copiar(c);
  }
  return(*this);
}

Cola::~Cola(){
  liberar();
}
```

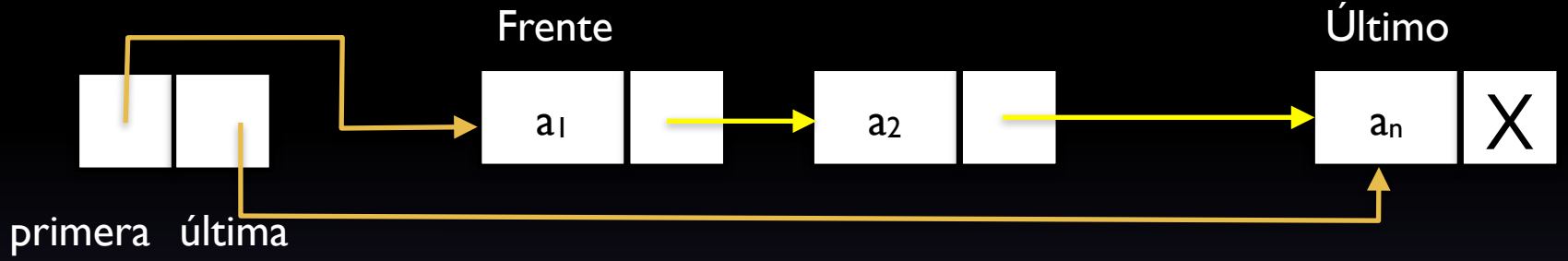
```
void Cola::poner(const Tbase valor){
  if(nelem==reservados)
    resize(2*reservados);
  datos[posterior] = valor;
  posterior=(posterior+1)%reservados;
  nelem++;
}

void Cola::quitar(){
  assert(nelem!=0);
  anterior = (anterior+1)%reservados;
  nelem--;
  if (nelem< reservados/4)
    resize(reservados/2);
}

Tbase Cola::frente() const{
  assert(nelem!=0);
  return datos[anterior];
}

bool Cola::vacia() const{
  return (nelem == 0);
}
```

Cola.cpp

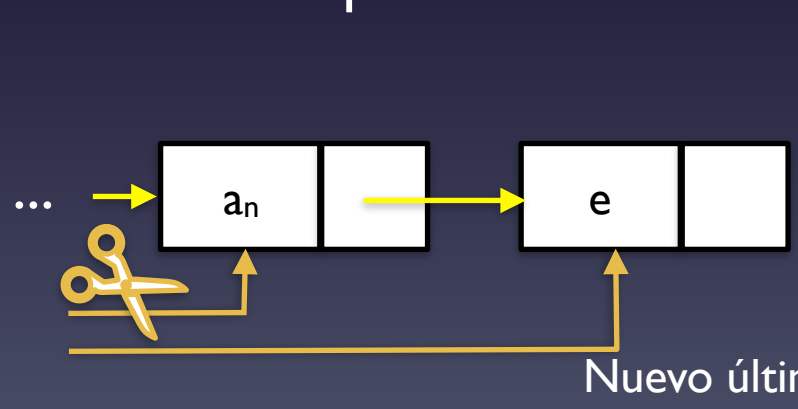
```
void Cola::copiar(const Cola &c){
  reservados = c.reservados;
  datos = new Tbase[reservados];
  for (int i= anterior; i!=posterior; i= (i+1)%reservados)
    datos[i] = c.datos[reservados];
  anterior = c.anterior;
  posterior = c.posterior;
  nelem = c.nelem;
}

void Cola::liberar(){
  delete[] datos;
  anterior = posterior = nelem = reservados = 0;
}

void Cola::resize(const int n){
  assert(n>0 && n>nelem);
  Tbase* aux = new Tbase[n];
  for(int i=0; i<nelem; i++)
    aux[i] = datos[(anterior+i)%reservados];
  anterior = 0;
  posterior = nelem;
  delete[] datos;
  datos = aux;
  reservados = n;
}
```

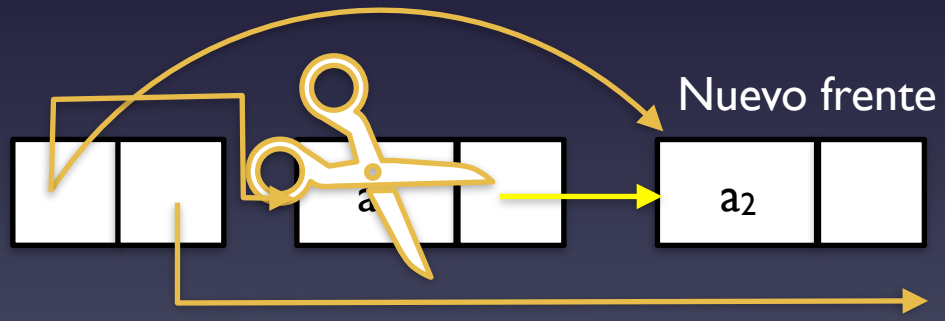
Colas. Implementación con listas

Almacenamos la secuencia de valores en celdas enlazadas



- Una cola vacía tiene dos punteros nulos
- El frente de la cola está en la primera celda (muy eficiente)
- En la inserción se añade una nueva celda al final y en el borrado se elimina la primera celda





Cola.h

```
#ifndef __COLA_H__
#define __COLA_H__

typedef char Tbase;

struct CeldaCola{
  Tbase elemento;
  CeldaCola* sig;
};

class Cola{
private:
  CeldaCola *primera, *ultima;
public:
  Cola();
  Cola(const Cola& c);
  ~Cola();
  Cola& operator=(const Cola& c);
  bool vacia() const;
  void poner(Tbase c);
  void quitar();
  Tbase frente() const;
private:
  void copiar(const Cola& c);
  void liberar();
};

#endif // __COLA_H__
```

Cola.cpp

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

Cola::Cola(){
  primera = ultima = 0;
}

Cola::Cola(const Cola& c){
  copiar(c);
}

Cola::~Cola(){
  liberar();
}

Cola& Cola::operator=(const Cola &c){
  if(this!=&c){
    liberar();
    copiar(c);
  }
  return *this;
}

bool Cola::vacia() const{
  return (primera == 0);
}
```

```
void Cola::poner(Tbase c){
  CeldaCola* aux = new CeldaCola;
  aux->elemento = c;
  aux->sig = 0;
  if (primera==0)
    primera = ultima = aux;
  else{
    ultima->sig = aux;
    ultima = aux;
  }
}

void Cola::quitar(){
  assert(primera!=0);
  CeldaCola* aux = primera;
  primera = primera->sig;
  delete aux;
  if (primera==0)
    ultima = 0;
}

Tbase Cola::frente() const{
  assert(primera!=0);
  return primera->elemento;
}
```

Cola.cpp

```
void Cola::copiar(const Cola& c){
  if (c.primera == 0)
    primera = ultima = 0;
  else{
    primera = new CeldaCola;
    primera->elemento = c.primera->elemento;
    ultima = primera;
    CeldaCola* orig = c.primera;
    while(orig->sig != 0){
      orig = orig->sig;
      ultima->sig = new CeldaCola;
      ultima = ultima->sig;
      ultima->elemento = orig->elemento;
    }
    ultima->sig = 0;
  }
}

void Cola::liberar(){
  CeldaCola* aux;
  while(primera!=0){
    aux = primera;
    primera = primera->sig;
    delete aux;
  }
  ultima = 0;
}
```

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

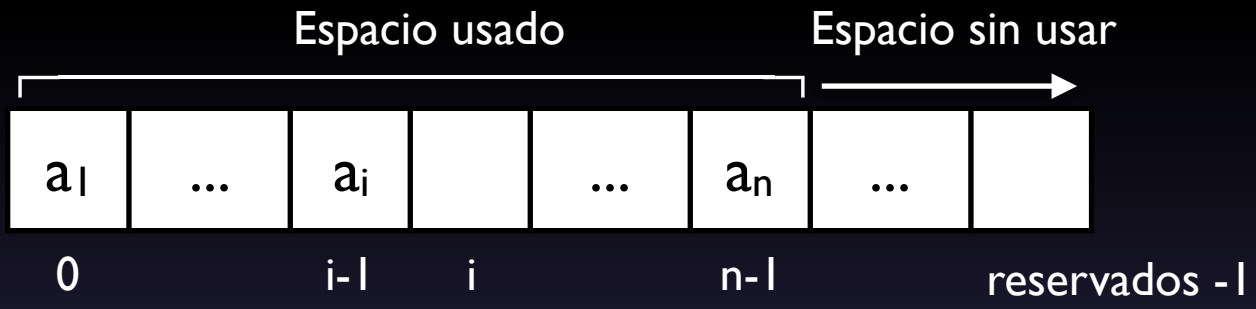
Listas

```
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;
}
```

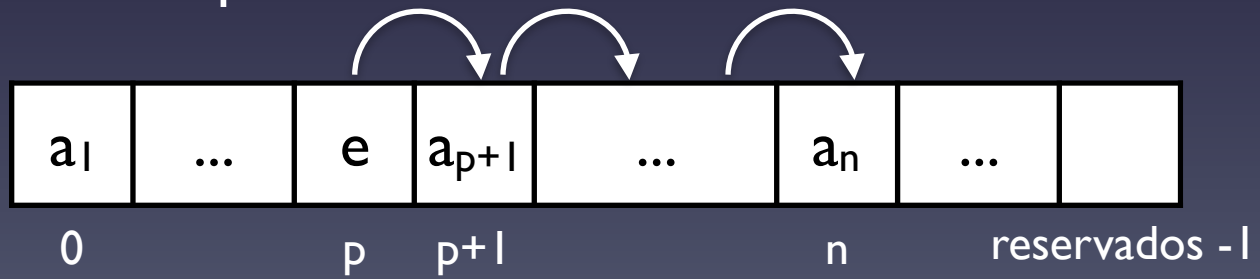
Uso de una lista

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;
}
```

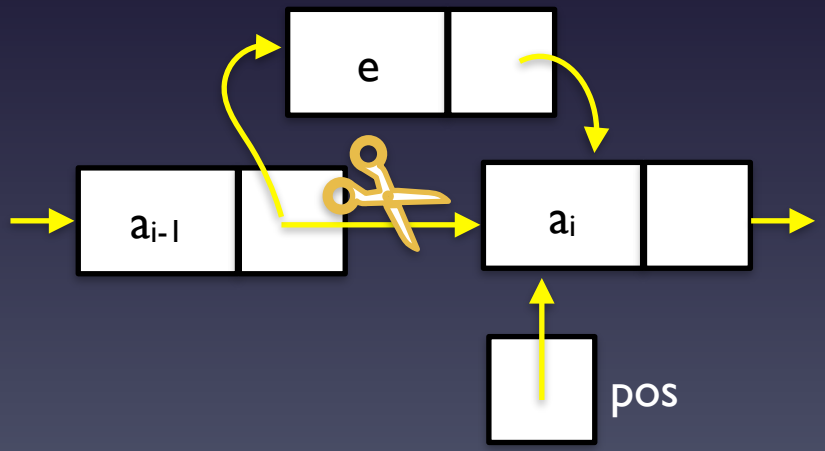
Lista.cpp

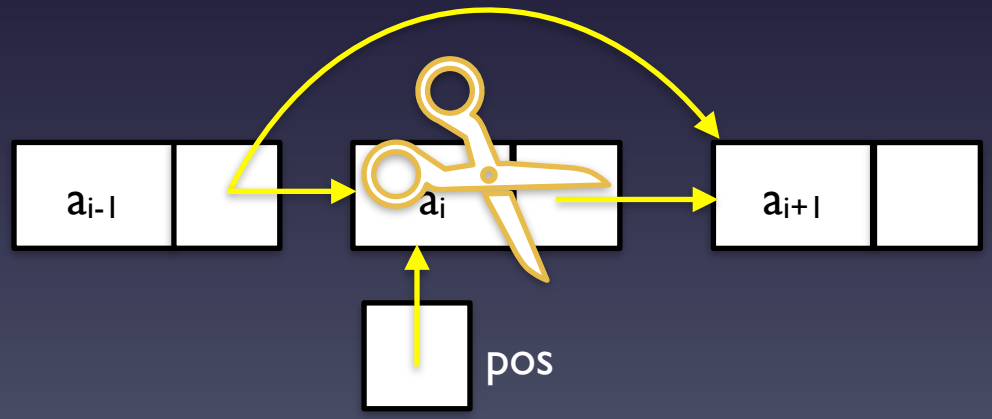
```
Tbase Lista::get(Posicion p) const{
  assert(p.i>=0 && p.i<nelementos);
  return datos[p.i];
}

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

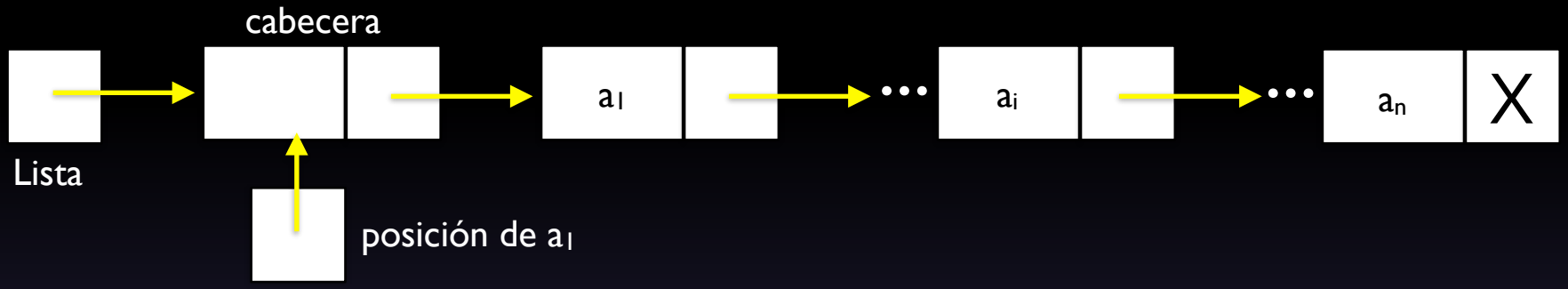
- Una lista es un puntero a la primera celda (si no está vacía)
- Una posición son dos punteros. El segundo (no se muestra) es necesario para algunos operadores
- Inserciones/borrados en la primera posición son casos especiales





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

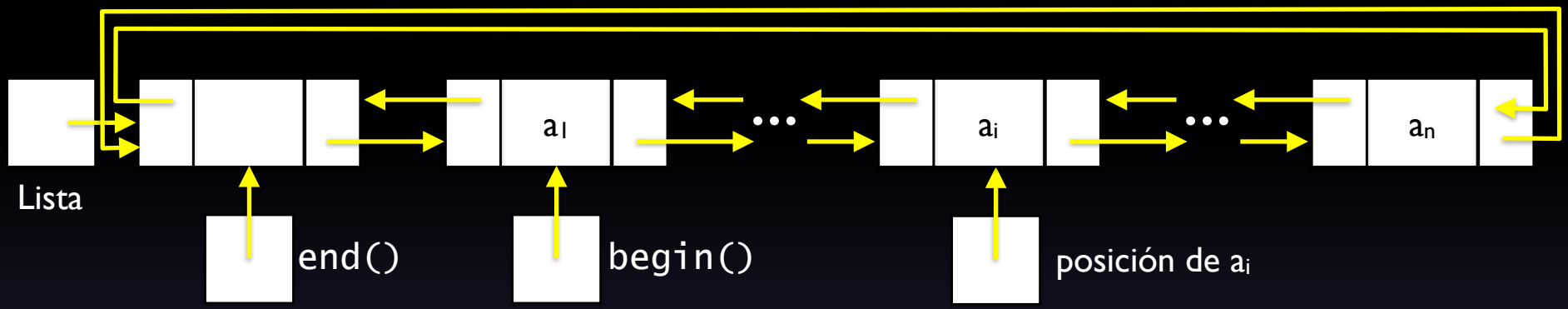
```
Lista& Lista::operator=(const Lista& l){
  Lista aux(l);
  intercambiar(cabecera, aux.cabecera);
  intercambiar(ultima, aux.ultima);
  return *this;
}

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;
}
```

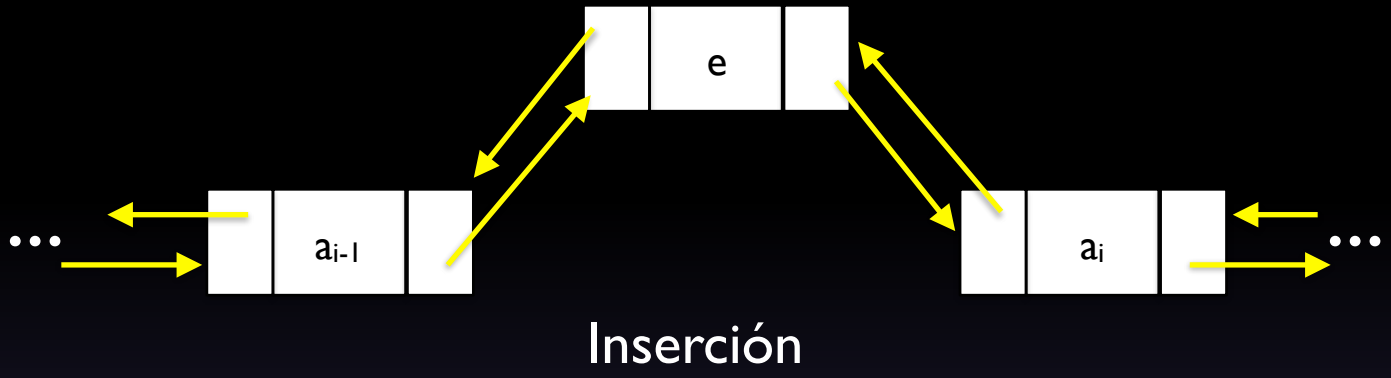
Listas. Celdas doblemente enlazadas circulares

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;
}
```

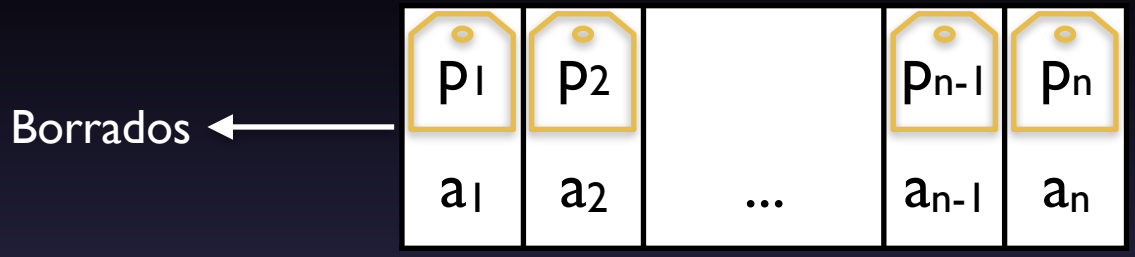
Lista.cpp

```
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;
}
```

COLAS CON PRIORIDAD

Colas con prioridad

- Una cola con prioridad es una estructura de datos lineal diseñada para realizar accesos y borrados en uno de sus extremos(frente). Las inserciones se realizan en cualquier posición, de acuerdo a un valor de prioridad



- **Operaciones básicas:**
  - Frente: devuelve el elemento del frente
  - Prioridad_Frente: devuelve la prioridad asociada al elemento del frente
  - Poner: añade un elemento con una prioridad asociada
  - Quitar: elimina el elemento del frente
  - Vacia: indica si la cola está vacía

Colas con prioridad

Esquema de la interfaz

```
#ifndef __COLA_PRI__
#define __COLA_PRI__

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

public:
  ColaPri();
  ColaPri(const ColaPri& c);
  ~ColaPri();
  ColaPri& operator=(const ColaPri& c);

  bool vacia() const;
  Tbase frente() const;
  Tprio prioridad_frente() const;
  void poner(Tbase e, Tprio prio);
  void quitar();
};

#endif /* ColaPri_hpp */
```

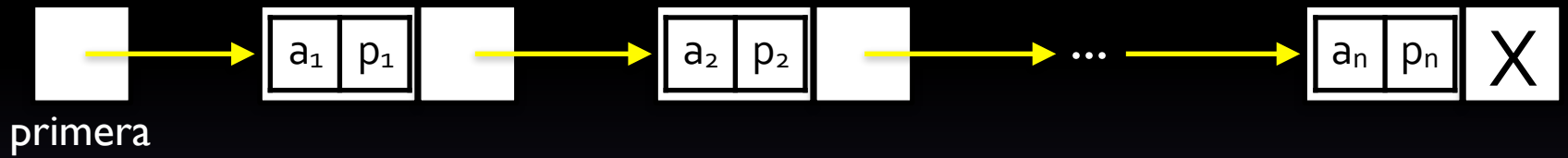
Colas con prioridad

Uso de una cola

```
#include <iostream>
#include <string>
#include "ColaPri.hpp"
using namespace std;

int main(){
  ColaPri c;
  int nota;
  string dni;
  cout << "Escriba una nota: ";
  cin >> nota;
  while(nota >=0 && nota <=10){
    cout << "Escriba un dni: ";
    cin >> dni;
    c.poner(dni, nota);
    cout << "Escriba una nota: ";
    cin >> nota;
  }
  cout << "DNIs ordenados por nota:" << endl;
  while(!c.vacia()){
    cout << "DNI: " << c.frente() << " Nota: "
         << c.prioridad_frente() << endl;
    c.quitar();
  }
  return 0;
}
```

Colas con prioridad. Celdas enlazadas

Almacenamos la secuencia de parejas en celdas enlazadas



- Una cola contiene un puntero nulo
- El frente de la cola está en la primera celda (muy eficiente)
- Si borramos el frente, eliminamos la primera celda
- En la inserción tenemos que buscar la posición según su prioridad

ColaPri.h

```
#ifndef __COLA_PRI__
#define __COLA_PRI__

#include <string>
using namespace std;
typedef int Tprio;
typedef string Tbase;

struct Pareja{
  Tprio prioridad;
  Tbase elemento;
};

struct CeldaColaPri{
  Pareja dato;
  CeldaColaPri* sig;
};
```

```
class ColaPri{

private:
  CeldaColaPri* primera;

public:
  ColaPri();
  ColaPri(const ColaPri& c);
  ~ColaPri();
  ColaPri& operator=(const ColaPri& c);

  bool vacia() const;
  Tbase frente() const;
  Tprio prioridad_frente() const;
  void poner(Tbase e, Tprio prio);
  void quitar();
};

#endif /* ColaPri_hpp */
```

ColaPri.cpp

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

ColaPri::ColaPri(): primera(0){}

ColaPri::ColaPri(const ColaPri& c){
  if(c.primera==0)
    primera = 0;
  else{
    primera = new CeldaColaPri;
    primera->dato = c.primera->dato;
    CeldaColaPri* src = c.primera;
    CeldaColaPri* dest = primera;
    while(src->sig!=0){
      dest->sig = new CeldaColaPri;
      src = src->sig;
      dest = dest->sig;
      dest->dato = src->dato;
    }
    dest->sig = 0;
  }
}

ColaPri::~ColaPri(){
  CeldaColaPri* aux;
  while(primera != 0){
    aux = primera;
    primera = primera->sig;
    delete aux;
  }
}
```

ColaPri.cpp

```
ColaPri& ColaPri::operator=(const ColaPri &c){
  ColaPri colatemp(c);
  CeldaColaPri* aux = this->primera;
  this->primera = colatemp.primera;
  colatemp.primera = aux;
  return *this;
}

bool ColaPri::vacia() const{
  return (primera==0);
}

Tbase ColaPri::frente()const{
  assert(primera!=0);
  return (primera->dato.elemento);
}

Tprio ColaPri::prioridad_frente() const{
  assert(primera!=0);
  return(primera->dato.prioridad);
}

void ColaPri::quitar(){
  assert(primera!=0);
  CeldaColaPri* aux = primera;
  primera = primera->sig;
  delete aux;
}
```

ColaPri.cpp

```
void ColaPri::poner(Tbase e, Tprio prio){
  CeldaColaPri* aux = new CeldaColaPri;
  aux->dato.elemento = e;
  aux->dato.prioridad = prio;
  aux->sig = 0;
  if (primera==0)
    primera = aux;
  else if(primera->dato.prioridad<prio){
    aux->sig = primera;
    primera = aux;
  }
  else{ //caso general
    CeldaColaPri* p = primera;
    while(p->sig!=0){
      if(p->sig->dato.prioridad<prio){
        aux->sig = p->sig;
        p->sig = aux;
        return;
      }
      else p = p->sig;
    }
    p->sig = aux;
  }
}
```