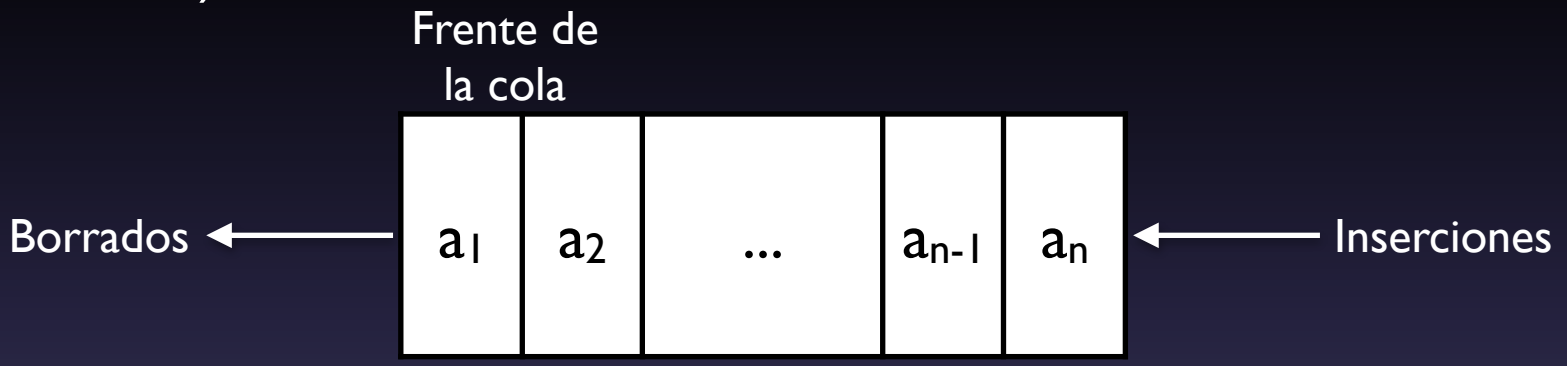
ESTRUCTURAS DE DATOS LINEALES

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__
```

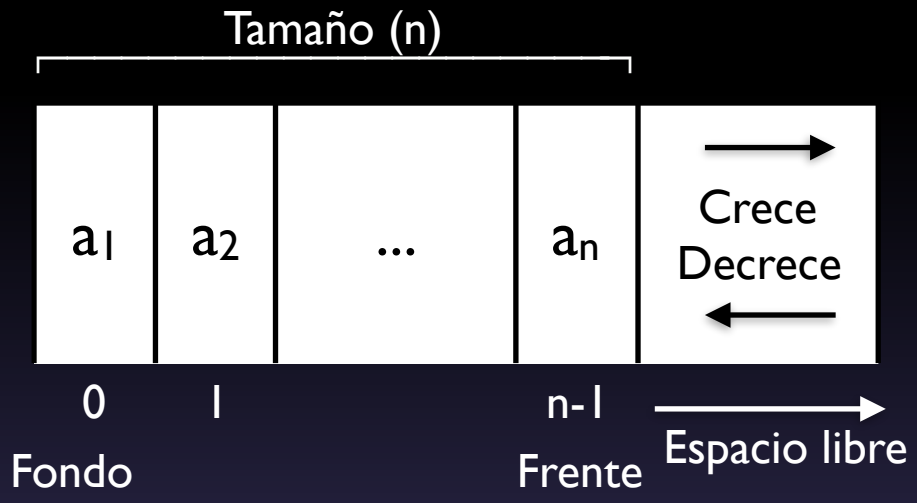
Colas

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

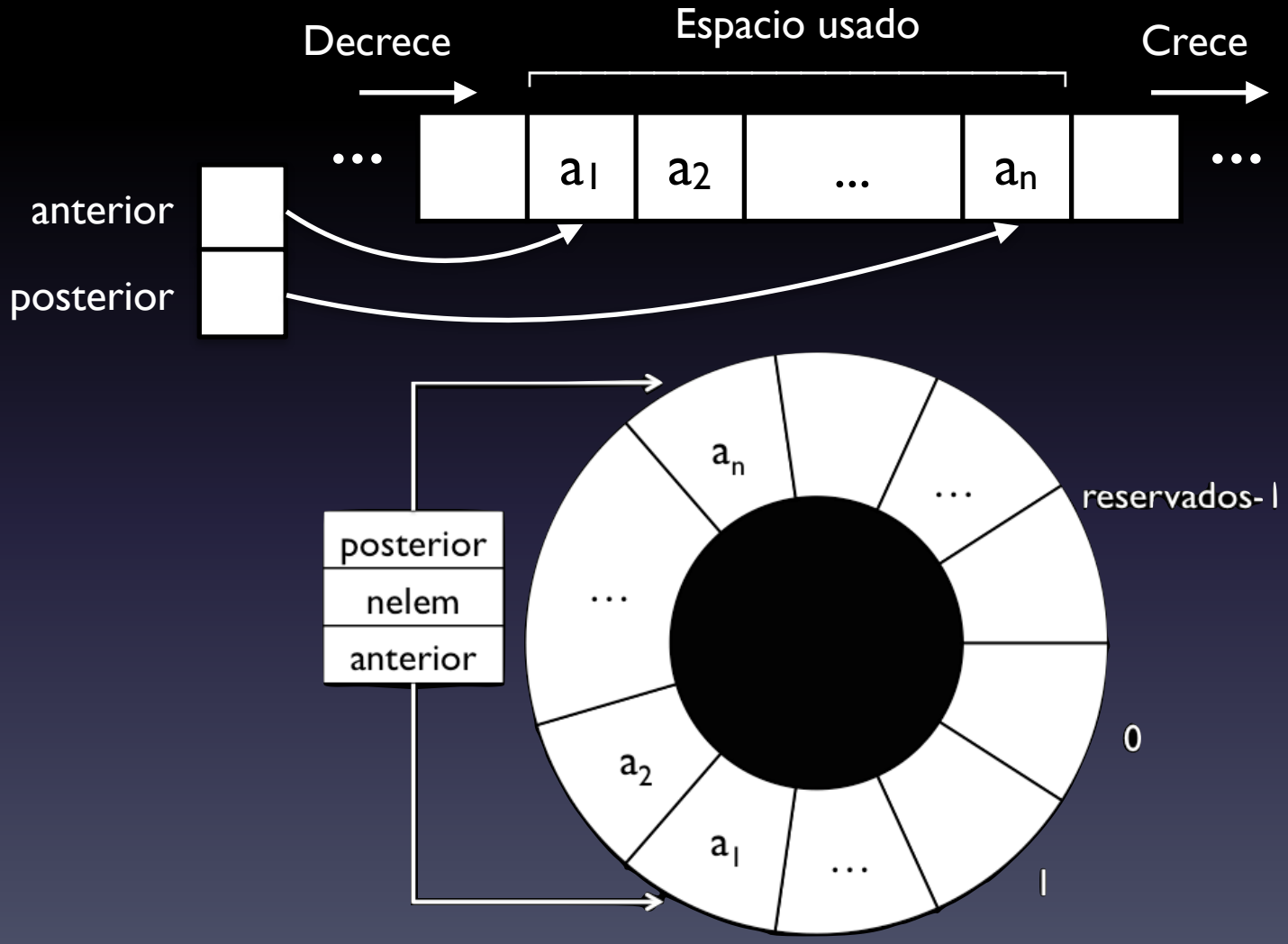
Almacenamos la secuencia de valores en un vector



- El fondo de la cola 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();
}
```

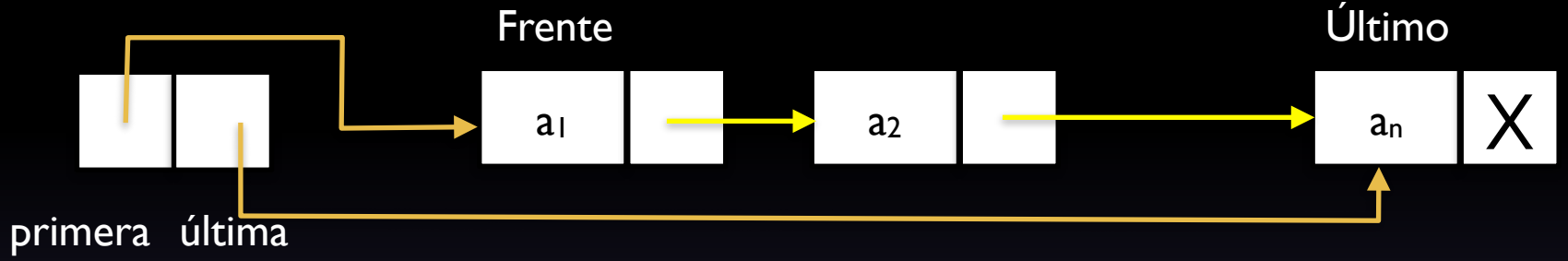
```
void Cola::poner(const Tbase valor){
  if(nelem==reservados)
    resize(2*reservados);
  datos[posterior] = valor;
  posterior=(posterior+1)%reservados;
  nelem++;
}

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

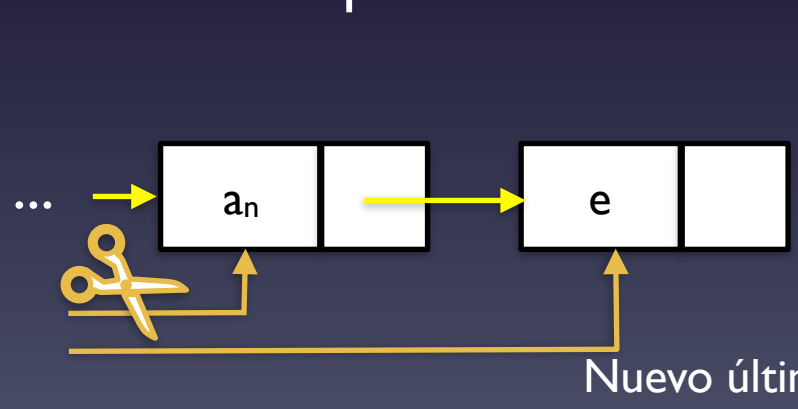
```
void Cola::copiar(const Cola &c){
  reservados = c.reservados;
  datos = new Tbase[reservados];
  for (int i= anterior; i!=posterior; i= (i+1)%reservados)
    datos[i] = c.datos[reservados];
  anterior = c.anterior;
  posterior = c.posterior;
  nelem = c.nelem;
}

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;
}
```

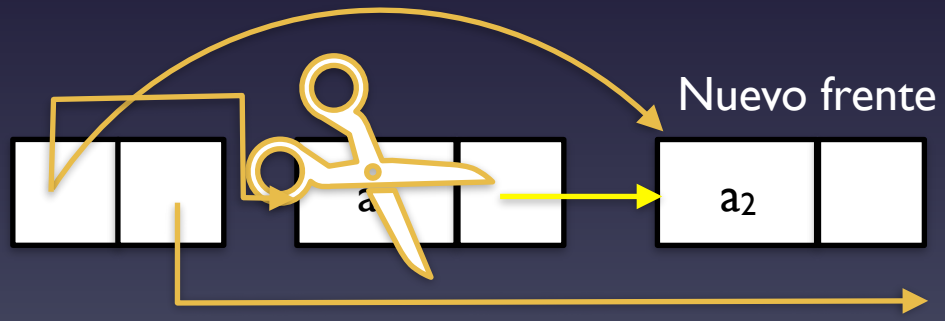
Colas. Implementación con listas

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;
}
```