ESTRUCTURAS DE DATOS LINEALES

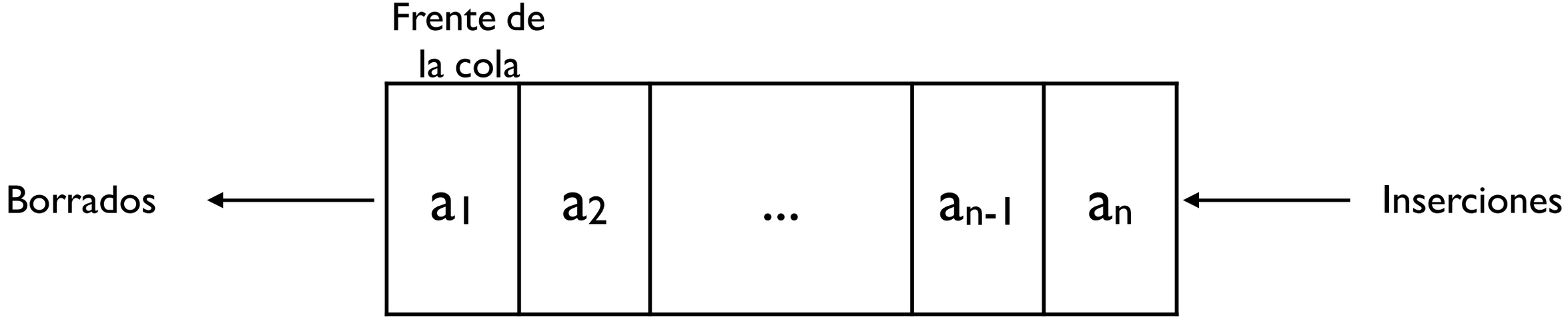
COLAS

Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Colas

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



Colas

1. **Especificación**:

- Contiene una secuencia de datos $\{a_0, a_1, a_n\}$ especialmente diseñada para hacer las inserciones por un extremo y los borrados y consultas por otro.
- El extremo en el que están el primer elemento $\{a_0, a_1, \ldots\}$ se llama ***frente***, y es por el que se hacen las consultas y borrados.
- El extremo en el que están los últimos valores (a_n) se llama ***última*** y es por el que se realizan las inserciones.
- Las colas responden a la política FIFO (First In, First Out).

Colas

2. **Operación:**
   - Frente: consulta o accede al elemento en el frente
   - Vacía: devuelve true si la cola está vacía
   - Quitar (pop): elimina le elemento que está en el frente
   - Poner (push): inserta un nuevo elemento por el final (la posición última)

3. **Implementación**:
   - Dinámica: Basada en celdas enlazadas y dos punteros (para que todas las operaciones sean O(1))
   - Estática: Basada en vectores **(circulares)**

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__
```

```
Tbase & frente();
const Tbase & frente() const;
```

Ejemplos

- Palíndromo

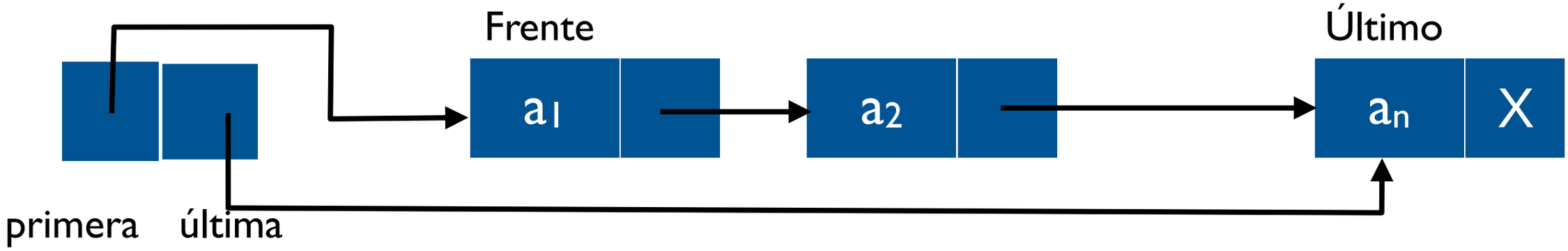
Colas

Uso de una cola

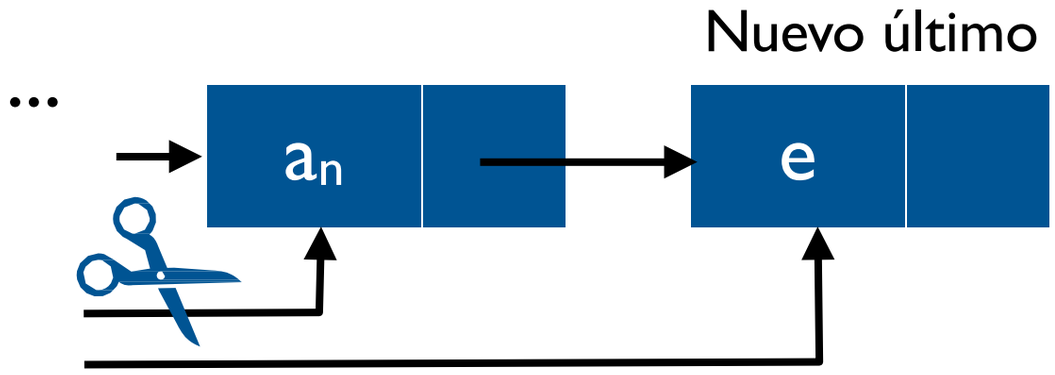
```
#include <iostream>
#include "Pila.hpp"
#include "Cola.hpp"
using namespace std;

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;
}
```

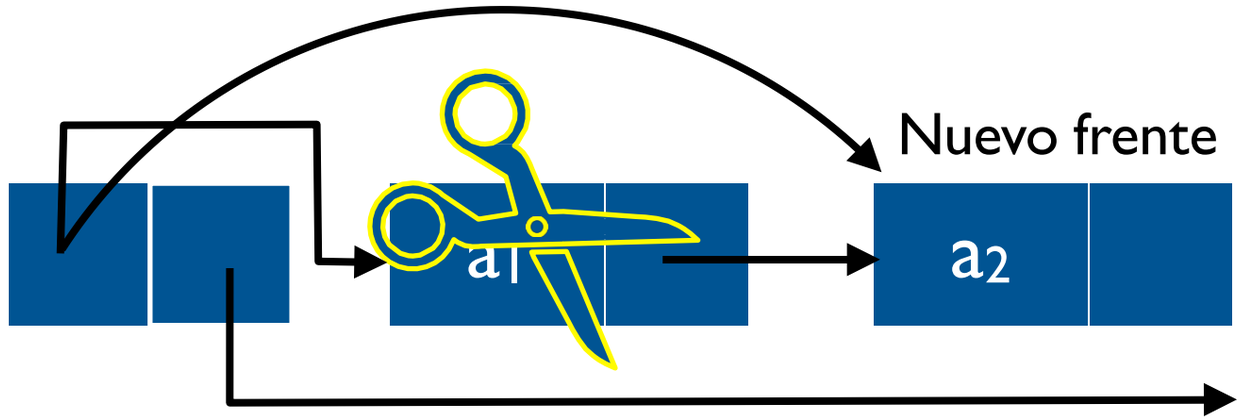
Colas. Implementación con celdas enlazadas

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(const Tbase & c);
  void quitar();
  Tbase frente() const;
private:
  void copiar(const Cola& c);
  void liberar();
};
#endif // __COLA_H__
```

Tbase & frente ();
const Tbase & frente () const;

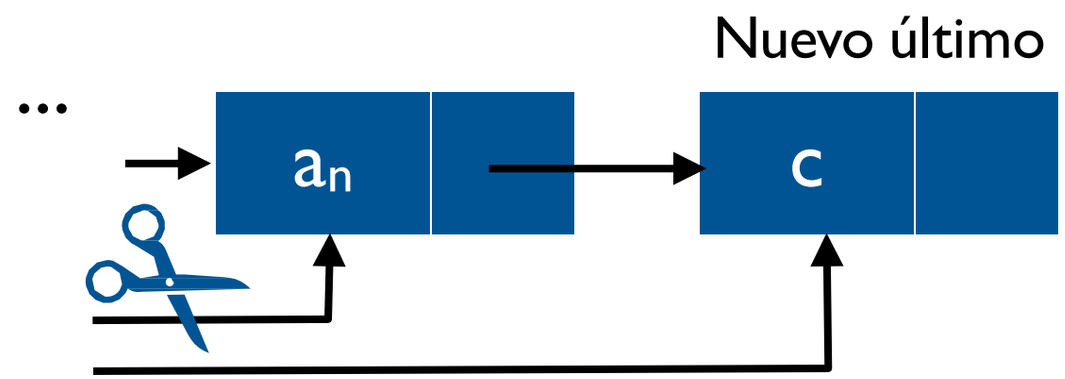
Cola.cpp

```
#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);
}
```

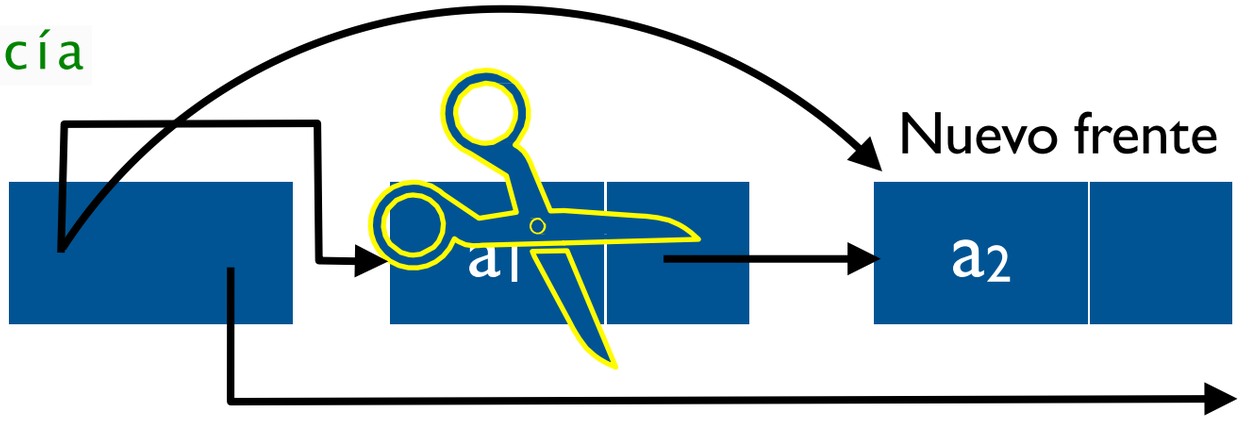
```
Tbase Cola::frente() const{
  //Comprobamos que no está vacía
  assert(primera!=0);
  return primera->elemento;
}
```

Cola.cpp

```
void Cola::poner(const Tbase & c){
   //Creamos una nueva celda
   CeldaCola* nueva = new CeldaCola;
   nueva->elemento = c;
   nueva->sig = 0;
   //Conectamos la celda
   if (primera==0) //Cola vacía
      primera = ultima = nueva;
   else{ //Cola no vacía
      ultima->sig = nueva;
      ultima = nueva;
   }
```




```
void Cola::quitar(){
   //Comprobamos que la cola no está vacía
   assert(primera!=0);
   //Hacemos que primera apunte
   //a la siguiente celda
   CeldaCola* aux = primera;
   primera = primera->sig;
   //Borramos la celda
   delete aux;
   //Si la cola queda vacía, tenemos que ajustar última
   if (primera==0)
      ultima = 0;
}
```

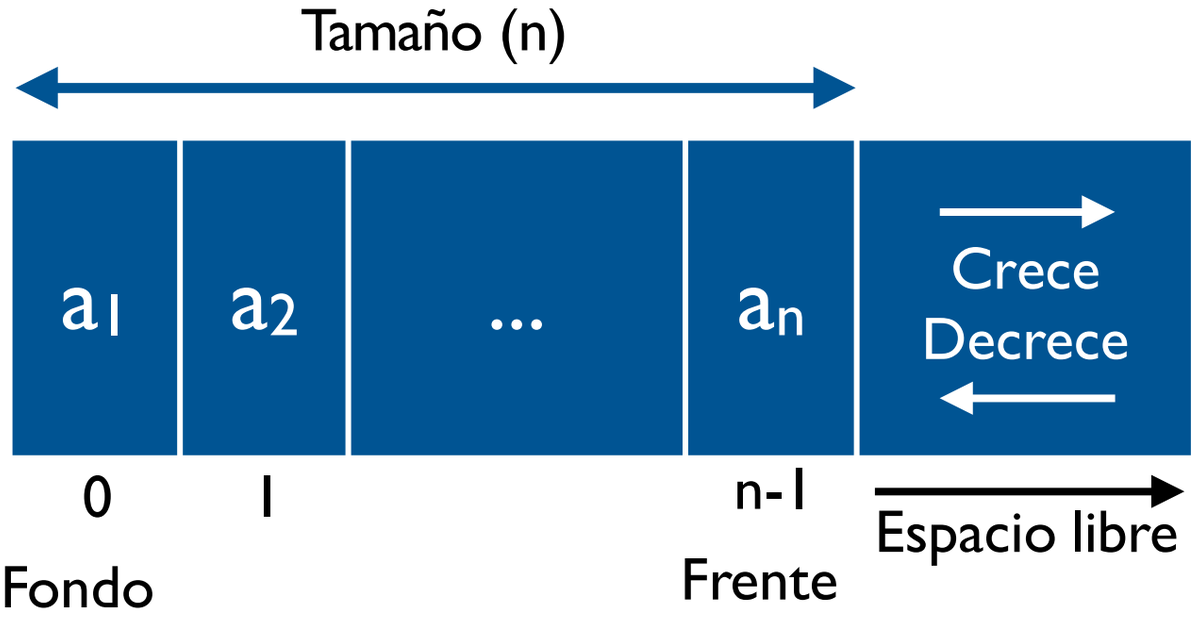


Cola.cpp

```
void Cola::copiar(const Cola& c){
  if (c.primera == 0) //Si la cola está vacía
    primera = ultima = 0;
  else{ //Caso general. No está vacía
    //Creamos la primera celda
    primera = new CeldaCola;
    primera->elemento = c.primera->elemento;
    ultima = primera;
    //Recorremos y copiamos el resto de la cola
    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;
}
```

Colas. Implementación con vectores

Almacenamos la secuencia de valores en un vector



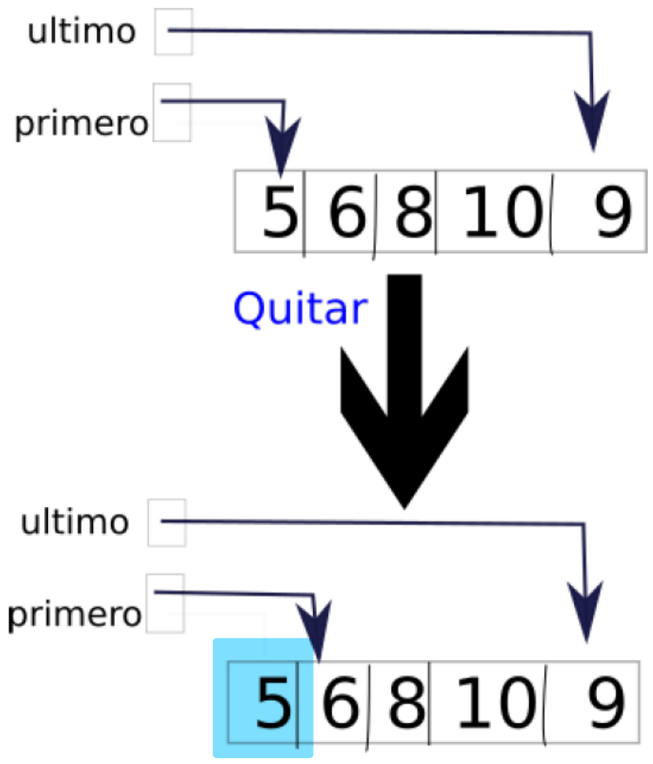
- El fondo de la cola está en la posición 0
- El número de elementos varía. Debemos almacenarlo
- Si insertamos elementos, el vector puede agotarse (tiene una capacidad limitada). Podemos resolverlo con memoria dinámica
- Problema: no se puede garantizar O(1) en inserciones y borrados

Colas. Implementación con vectores

- **Coste de las operaciones:**
  1. Vacía → *return n == 0*, tiene eficiencia O(1)
  2. Frente → *return v[0]*, tiene eficiencia O(1)
  3. Poner → *v[n] = nuevo_elemento*;, si tenemos memoria suficiente, tiene eficiencia O(1), y si tenemos que ampliar el vector, O(n). Si tenemos en cuenta el tiempo amortizado realmente nos costaría en promedio O(1)
  4. Quitar → debemos hacer un *for* para desplazar todos los elementos, por lo que tenemos eficiencia O(n)

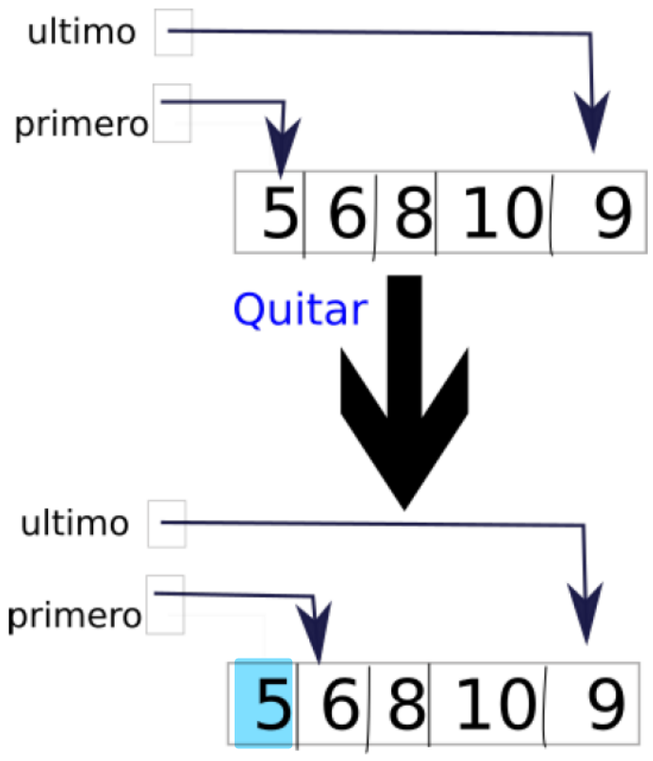
Colas. Implementación con vectores

- **Mejora de eficiencia Quitar:**
  - Debemos pasar de O(n) a O(1) para ello podemos tener dos índices extra: uno que apunte al primer elemento y otro que apunte al último. Así, cuando queramos eliminar un elemento sólo debemos desplazar el primer índice dejando ese elemento como valor basura.



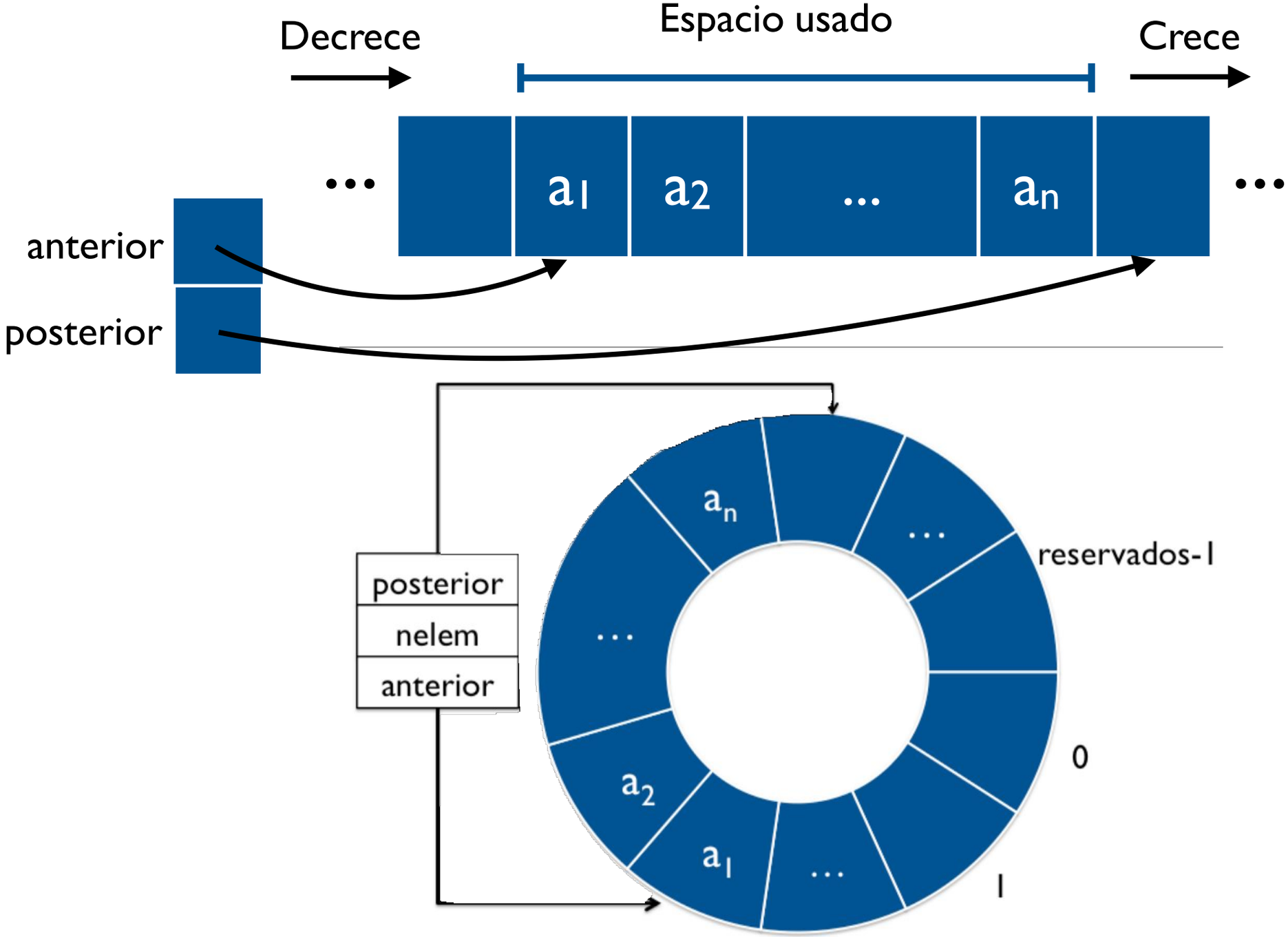
Colas. Implementación con vectores

- **Mejora de eficiencia Quitar:**
  - El problema de esta solución es que puede llegar un momento en el que se colapse la memoria.
  - Para solucionarlo, usamos *vectores circulares:* cuando nos quedamos sin espacio por el final (el índice del primer elemento y el del último se igualan), seguimos rellenando por el principio.



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 reservar(const int n);
  void liberar();
  void copiar(const Cola& c);
  void redimensionar(const int n);
};
#endif // __COLA_H__
```

Tbase & frente ();
const Tbase & frente () const;

Cola.cpp

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

Cola::Cola(){
  reservar(10);
  anterior = posterior = nelem = 0;
}

Cola::Cola(const Cola& c){
  reservar(c.reservados);
  copiar(c);
}

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

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

Cola.cpp

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

uso del operador %

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

uso del operador %

```
Tbase Cola::frente() const{
   assert(!vacia());
   return datos[anterior];
}
```

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

Cola.cpp

```
void Cola::reservar(const int n){
  assert(n>0);
  reservados = n;
  datos = new Tbase[n];
}
void Cola::liberar(){
  delete[] datos;
  datos = 0;
  anterior = posterior = nelem = reservados = 0;
}
void Cola::copiar(const Cola &c){
  for (int i= c.anterior; i!=c.posterior; i= (i+1)%reservados)
    datos[i] = c.datos[i];
  anterior = c.anterior;
  posterior = c.posterior;
  nelem = c.nelem;

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

Ejercicios propuestos:

- Desarrollar una clase Cola genérica con templates
- Sobrecargar += y --

TDA Cola (Queue)

Uso de una cola
STL

```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;

int main() {
  stack<char> p;
  queue<char> c;
  char dato;
  cout << "Escriba una frase" <<endl;
  while((dato=cin.get()) !='\n')
    if (dato != ' '){
      p.push(dato);
      c.push(dato);
    }
  bool palindromo = true;
  while(!p.empty() && palindromo){
    if(c.front() != p.top())
      palindromo = false;
    p.pop();
    c.pop();
  }
  cout << "La frase “ << (palindromo ? "es" : "no es")
       << " un palíndromo" << endl;

  return 0;
}
```

Ejemplos

- Eliminar elementos repetidos consecutivos de una cola

Ej. 1 1 2 2 2 5 5 1 → 1 2 5 1

Ejemplos

- Eliminar elementos repetidos consecutivos de una cola

Ej. 1 1 2 2 2 5 5 1 → 1 2 5 1

```
1  #include "Cola.h"
2
3  void EliminarConsecutivos (Cola<int> &c)
4  {
5        Cola<int> caux;
6
7        while (!c.Vacia())
8        {
9              int d = c.Frente();
10             c.Quitar();
11             caux.Poner(d);
12             while (!c.Vacia() && d==c.Frente())
13                   c.Quitar();
14       }
15
16       while (!caux.Vacia())
17       {
18             int d = caux.Frente();
19             c.Poner(d);
20             caux.Quitar();
21       }
22 }
```

Ejemplos

- Representa el T.D.A. Cola haciendo uso de T.D.A. Pila.

Ejemplos

- Representa el T.D.A. Cola haciendo uso de T.D.A. Pila.
  - Para ello representaremos nuestra cola con dos pilas. Una primera pila sobre la que hago las consultas y borrados P_2 y otra para realizar las inserciones P_1.

1. **Insertar**: 3 2 1 9
   P1: 3 2 1 9 donde 9 es el tope de la pila
   P2:
2. **Consultar frente**: para ello debemos insertar los elementos en P2 para obtener el primer elemento que se insertó en la cola:
   P1:
   P2: 9 1 2 3 en este caso, el frente sería 3
3. **Insertar**: 5 7
   P1: 5 7
   P2: 9 1 2 3
4. **Quitar (el frente)**:
   P1: 5 7
   P2: 9 1 2

Ejemplos

- Representa el T.D.A. Cola haciendo uso de T.D.A. Pila.
  - Podemos quitar elementos de P_2 hasta dejarla vacía, y si quisiésemos seguir quitando, insertaríamos los elementos de P_1 en P_2 y haríamos la operación de quitar el frente.

1. **Insertar**: 3 2 1 9
   P1: 3 2 1 9 donde 9 es el tope de la pila
   P2:
2. **Consultar frente**: para ello debemos insertar los elementos en P2 para obtener el primer elemento que se insertó en la cola:
   P1:
   P2: 9 1 2 3 en este caso, el frente sería 3
3. **Insertar**: 5 7
   P1: 5 7
   P2: 9 1 2 3
4. **Quitar (el frente)**:
   P1: 5 7
   P2: 9 1 2

Ejemplos

- Representa el T.D.A. Cola haciendo uso de T.D.A. Pila.

```
1  #include "Pila.h"
2
3  class Cola
4  {
5  private:
6        Pila<int> p1; //insertar
7        Pila<int> p2; //frente y quitar
8
9  public:
10       int Frente () {
11             if (p2.Vacia()) {
12                   while (!p1.Vacia()) {
13                         int d = p1.Tope();
14                         p2.Poner(d);
15                         p1.Quitar();
16                   }
17             }
18             return p2.Tope();
19       }
20
21       bool Vacia() const {
22             return p1.Vacia() && p2.Vacia();
23       }
```

Ejemplos

- Representa el T.D.A. Cola haciendo uso de T.D.A. Pila.

```
25          void Poner (int d) {
26                p1.Poner(d);
27          }
28
29          void Quitar () {
30                if (p2.Vacia()) {
31                      while (!p1.Vacia()) {
32                            int d = p1.Tope();
33                            p2.Poner(d);
34                            p1.Quitar();
35                      }
36                }
37                p2.Quitar();
38          }
39    };
```

Colas (celdas enlazadas).

Cola.h

```
template <class T>
class Cola{
  private:
    struct Celda {
      T elemento;
      Celda * siguiente;

      Celda() : siguiente(0) {}
      Celda(const T & elem, Celda * sig)
        : elemento(elem), siguiente(sig) {}
    };
    Celda * primera;
    Celda * ultima;
    int num_elem;
  public:
  Cola(): primera(0),ultima(0),num_elem(0)
    {}
  Cola(const Cola<T> & p);
  ~Cola();
  Cola& operator= (const Cola<T>& p);
  bool Vacia() const {return num_elem==0;}
  T& Frente ()
    { assert(primera! =0); return primera->elemento;}
  const T & Frente () const
    { assert(primera! =0); return primera->elemento;}
  void Poner(const T & elem);
  void Quitar();
  int Num_elementos() const { return num_elem; }
};
```

Cola.cpp

```
template <class T>
Cola<T>::~Cola() {
  Celda *aux;
  while (primera! =0) {
    aux= primera;
    primera=primera->siguiente;
    delete aux;
  }
}
template <class T>
void Cola<T>::Poner(const T & elem) {
  Celda *aux=new Celda(elem,0);
  if (primera==0) primera=ultima= aux;
  else {
    ultima->siguiente=aux;
    ultima= aux;
  }
  num_elem++;
}
template <class T>
void Cola<T>::Quitar() {
  assert(primera! =0);
  Celda *aux=primera;
  primera= primera->siguiente;
  delete aux;
  if (primera==0) ultima=0;
  num_elem--;
}
```

Colas (vectores).

Cola.h

```
template <class T>
class Cola{
  private:
    Vector<T> v;
    int num_elem;
    int anterior,posterior;
    void Expandir(int nelem) {
        assert(nelem>v.size());
        Vector<T> aux(nelem);
        for (int i=0;i<num_elem;i++)
          aux[i]=v[(anterior+i) %v.size()];
        anterior=0;
        posterior=(anterior+num_elem);
        v=aux;
      }
    void Contraer(int nelem) {
        assert(nelem<v.size());
        Vector<T> aux(nelem);
        for (int i=0;i<num_elem;i++)
          aux[i]=v[(anterior+i) %v.size()];
        anterior=0;
        posterior=(anterior+num_elem);
        v=aux;
      }
  public:
    Cola(): v(1),num_elem(0),anterior(0),posterior(0) {}
    Cola(const Cola<T> & p):v(p.v),num_elem(p.num_elem),
                            anterior(p.anterior),posterior(p.posterior) {}
    ~Cola() {}
```

Cola.h

```
    Cola& operator= (const Cola<T>& p)
      {
        v=p.v; num_elem=p.num_elem;
        anterior=p.anterior; posterior=p.posterior;
      }
    bool Vacia() const
      {return num_elem==0;}
    T& Frente ()
      { assert(num_elem! =0); return v[anterior];}
    const T & Frente () const
      { assert(num_elem! =0); return v[anterior];}
    void Poner(const T & elem)
      {
        if (num_elem==v.size())
          Expandir(2*v.size());
        v[posterior]=elem;
        posterior=(posterior+1) %v.size();
        num_elem++;
      }
    void Quitar()
      {
        assert(num_elem! =0);
        anterior=(anterior+1) %v.size();
        num_elem--;
        if (num_elem<v.size()/4)
          Contraer(v.size()/2);
      }
    int Num_elementos() const { return num_elem; }
};
```