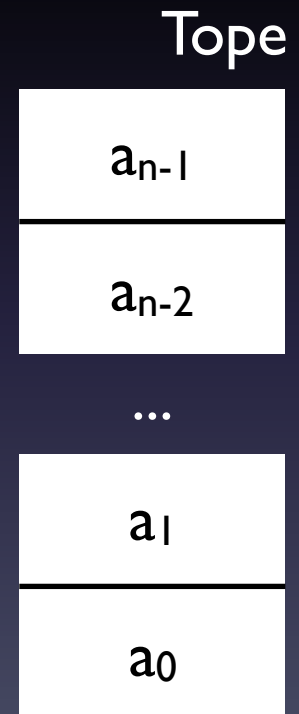


# ESTRUCTURAS DE DATOS LINEALES

# PILAS

# Pilas

- Las estructuras de datos lineales se caracterizan porque consisten en una secuencia de elementos,  $a_0, a_1, \dots, a_n$ , 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
  - ▶ Vacía: 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

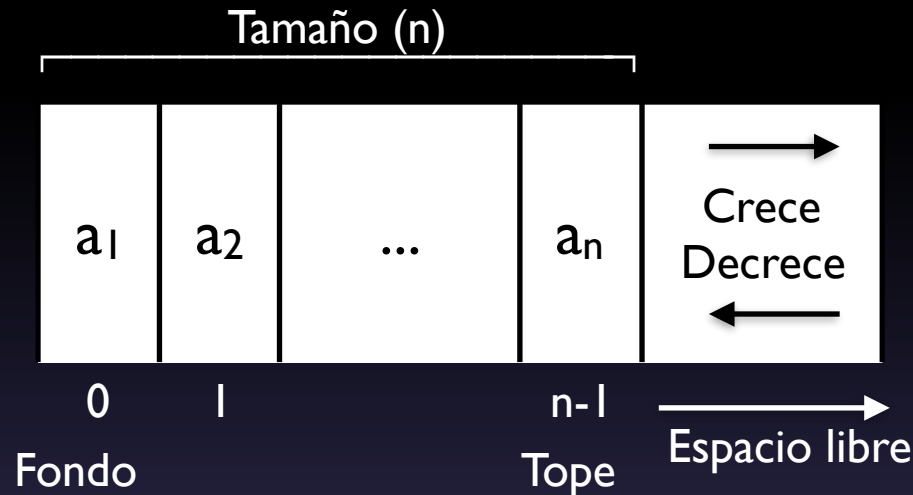
```
#include <iostream>
#include "Pila.hpp"
using namespace std;

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::vacía() 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::vacía() const{
    return(nelem==0);
}

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

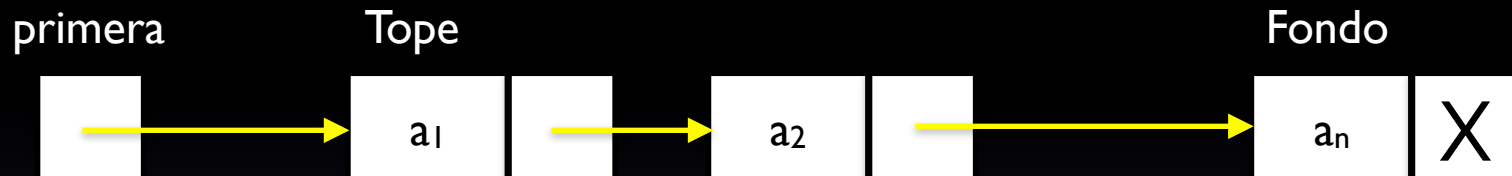
void Pila::quitar(){
    assert(nelem>0);
    nelem--;
    if(nelem<reservados/4)
        resize(reservados/2);
}

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

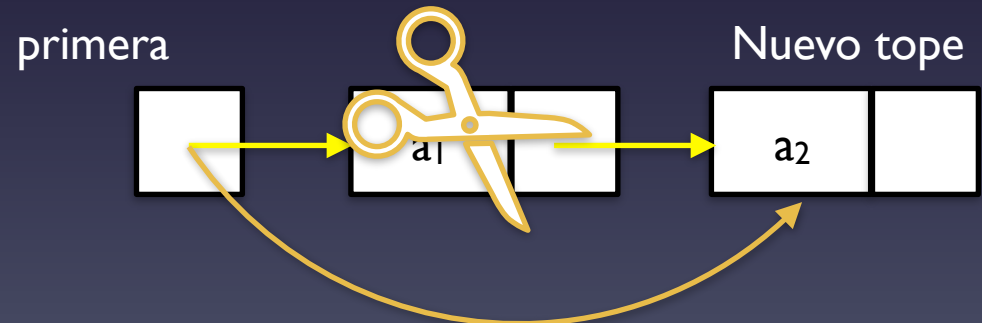
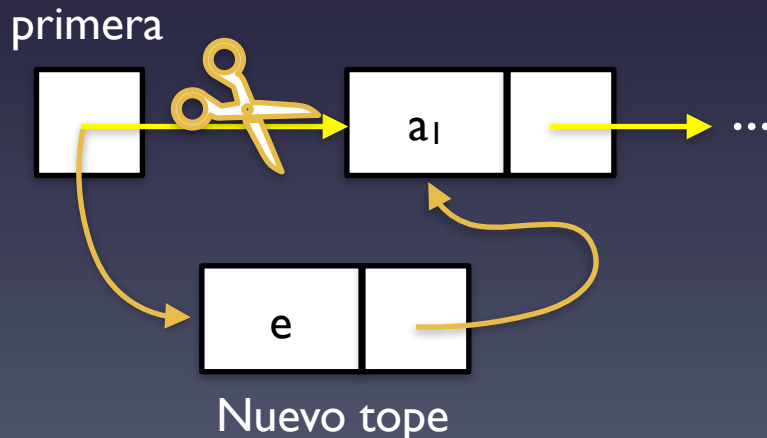
- Esta implementación es mucho más eficiente en cuanto a consumo de memoria
- Ejercicio propuesto: desarrollar el resto de métodos

# 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.primer==0)
        primera = 0;
    else{
        primera = new CeldaPila;
        primera->elemento = p.primer->elemento;
        CeldaPila *orig = p.primer, *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(primer!=0){
        aux = primera;
        primera = primera->sig;
        delete aux;
    }
    primera = 0;
}
```