



Phantone

www.wuolah.com/student/Phantone

708

ED-T3-1.pdf

Apuntes Teoría



2º Estructuras de Datos



Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



**El más PRO del lugar
puedes ser Tú.**

¿Quieres eliminar toda la publi
de tus apuntes?

Hazte PRO

4,95€/mes

W

TEMA 3.1: PILAS

Las estructuras de datos lineales se caracterizan por consistir 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 lineal caracterizadas por su comportamiento LIFO: todas las inserciones y borrados se realizan en un extremo de la pila llamado **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.

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() = default;
    Pila & operator=(const Pila &p);

    bool vacia() const;
    void poner(const Tbase & c);
    void quitar();
    Tbase & tope() const; → Tbase & tope();
    const Tbase & tope() const;
};

#endif /* Pila_hpp */
```

Uso de una pila.

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

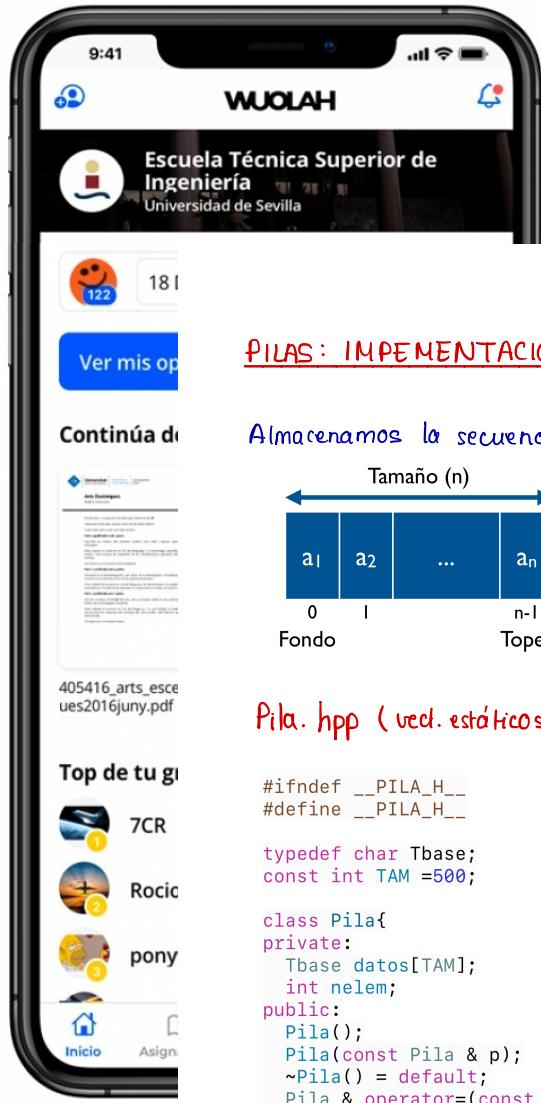
int main() {
    Pila p, q;
    char dato;

    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;
}
```



Descarga la APP de Wuolah.

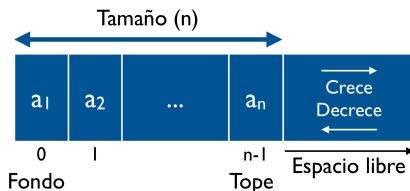
Ya disponible para el móvil y la tablet.



PILAS: IMPLEMENTACIÓN CON VECTORES

Continúa di

Almacenamos la secuencia de valores en un vector.



405416_arts_escuela2016juny.pdf

Top de tu grupo

7CR

Rocic

pony

Inicio

Asign

Pila.hpp (vect. estáticos)

```
#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() = default;
    Pila & operator=(const Pila &p);
    bool vacia() const;
    void poner(const Tbase & c);
    void quitar();
    Tbase & tope();
    const Tbase & tope() const;
private:
    //Método auxiliar privado
    void copiar(const Pila &p);
};

#endif /* Pila_hpp */
```

Pila.h (vect. 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(int tam=TAM);
    Pila(const Pila & p);
    ~Pila();
    Pila & operator=(const Pila &p);
    bool vacia() const;
    void poner(Tbase c);
    void quitar();
    Tbase & tope();
    const Tbase & tope() const;
private: //Métodos auxiliares
    void resize(int n);
    void copiar(const Pila& p);
    void liberar();
    void reservar(int n);
};

#endif /* Pila_hpp */
```

Característica
específica de una
implementación
vectorial

- El fondo de la pila está en la posición 0.
- El nº elementos varía (lo almacenamos)
- Si insertamos, el vector puede agotarse. Resolvemos con memoria dinámica.

Pila.cpp (vect. estáticos)

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

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

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

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

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

const 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.

Pila.cpp (vectores dinámicos)

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

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

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

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

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

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

Esta implementación es mucho más eficiente en cuanto a consumo de memoria.

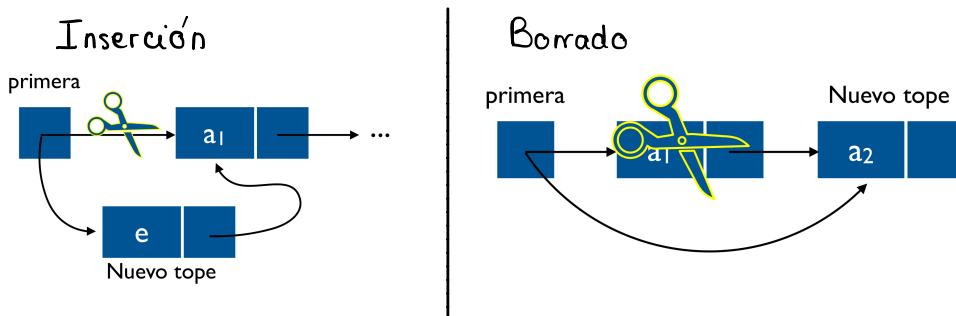
WUOLAH

PILAS: IMPLEMENTACIÓN CON CELDAS ENLAZADAS

Almacenamos una secuencia de valores en celdas enlazadas.



- Una pila tiene un puntero (primera) nulo.
- El tope de la pila está en la primera celda (eficiente).
- Inserción y borrado sobre la primera celda.



Pila.hpp

```
#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);
}

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;
}
```

• Esta implementación tiene un consumo de memoria directamente proporcional al nº de elementos.

• Coste adicional: los punteros empleados para conectar celdas.