ESTRUCTURAS DE DATOS LINEALES

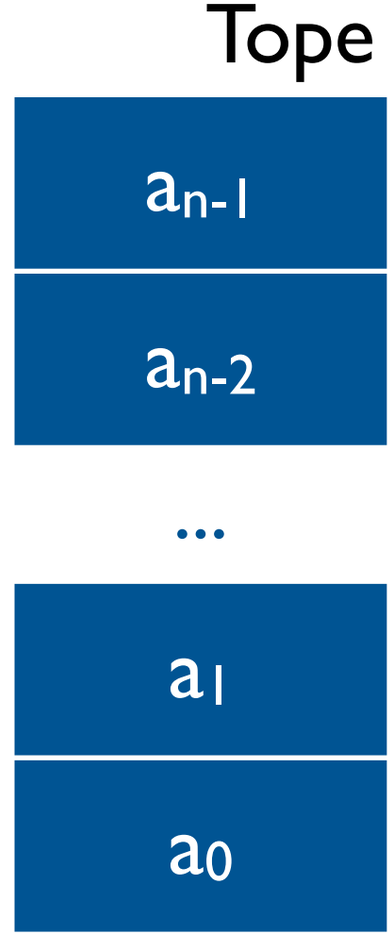
PILAS

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

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

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

#endif /* Pila_hpp */
```

→

```
Tbase & tope();
const Tbase & tope() const;
```

Podríamos sobrecargar quitar() y poner() en los operadores -- y +=

Ahora bien, ¿tiene sentido que devuelvan la pila? ¿Podemos sobrecargarlos como métodos void?

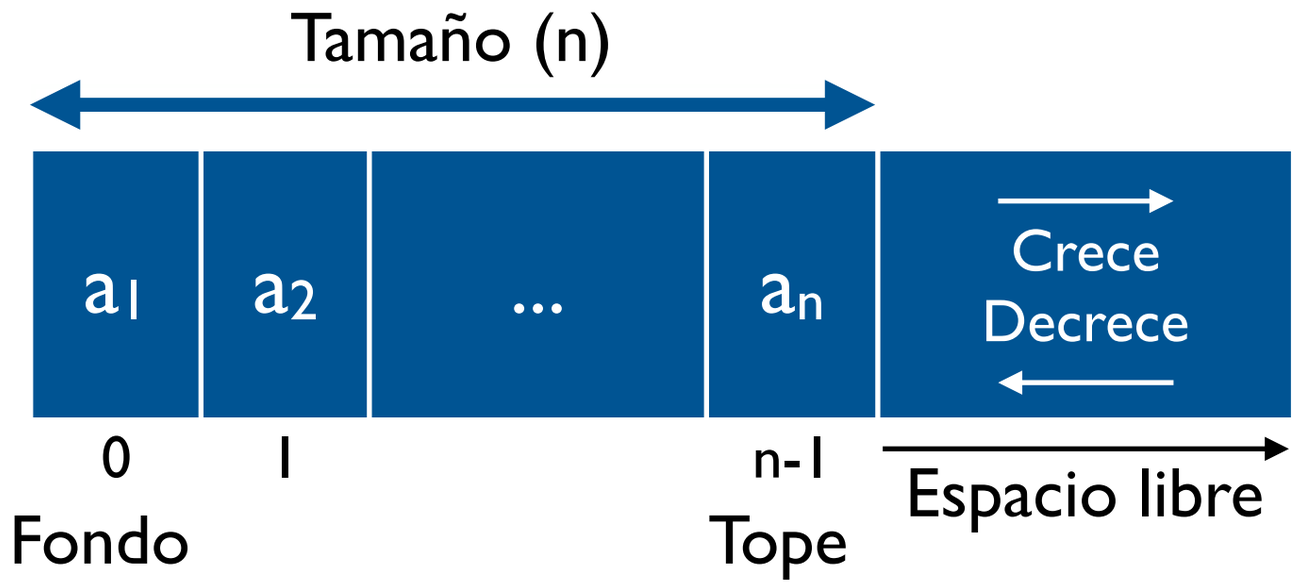
Pilas

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.hpp

```
#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.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(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

Ejercicios propuestos:

- Desarrollar el resto de métodos
- Sobrecargar quitar() y poner() en los operadores -- y +=

Pila.hpp (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(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 */
```

¡Ojo! Característica específica de una implementación vectorial

Pila.cpp (Vectores dinámicos)

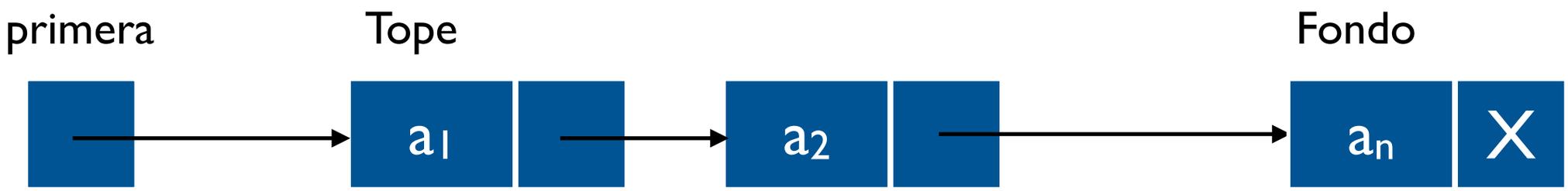
```
#include <cassert>
#include "Pila.hpp"

//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;
}

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

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

- Esta implementación es mucho más eficiente en cuanto a consumo de memoria
- Ejercicios propuestos:
  - Desarrollar el resto de métodos
  - Desarrollar una clase Pila genérica con templates

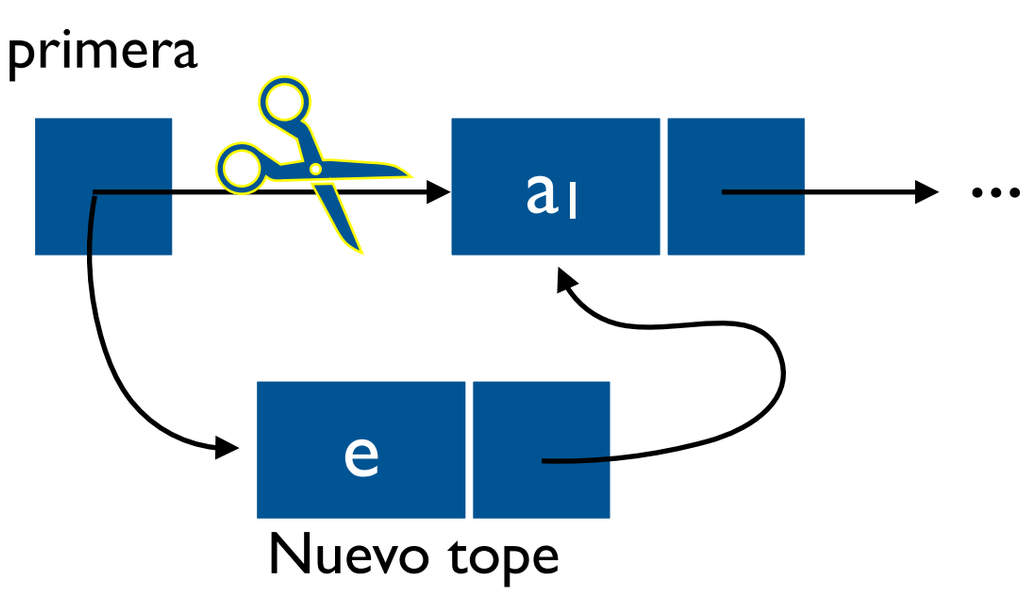
```
const Tbase & Pila::tope() const{
  assert(nelem>0);
  return datos[nelem-1];
}
```

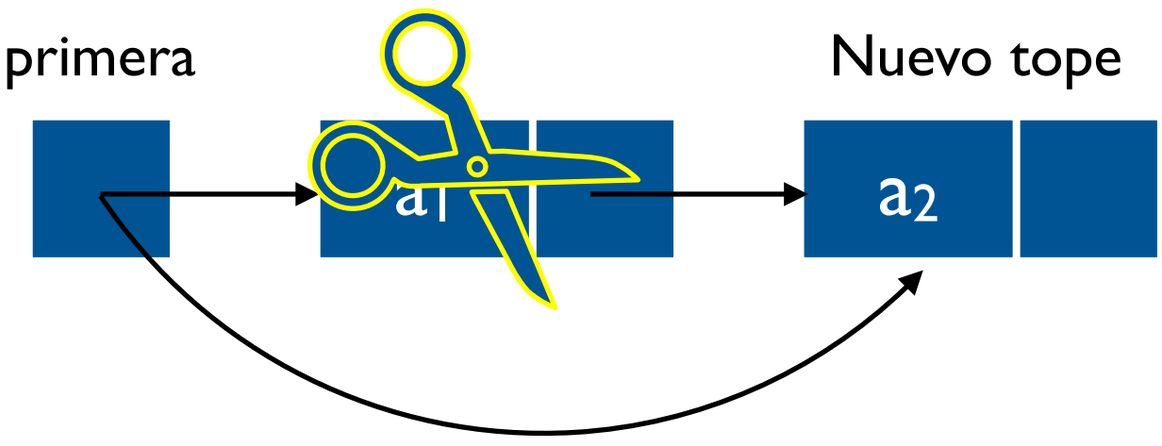
Pilas. Implementación con celdas enlazadas

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;
}
```

- Esta implementación tiene un consumo de memoria directamente proporcional al número de elementos.
- Coste adicional: los punteros empleados para conectar celdas
- Ejercicio propuesto:
  - Desarrollar una clase Pila genérica con templates

TDA stack (STL)

Uso de una pila
STL

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

int main(){
  stack<char> p, q;
  char dato;

  cout << "Escriba una frase" << endl;
  while((dato=cin.get())!='\n')
    p.push(dato);

  cout << "La escribimos del revés" << endl;
  while(!p.empty()){
    cout << p.top();
    q.push(p.top());
    p.pop();
  }

  cout << endl << "La frase original era" << endl;
  while(!q.empty()){
    cout << q.top();
    q.pop();
  }
  cout << endl;
  return 0;
}
```