ESTRUCTURAS DE DATOS LINEALES

TDA Lineales

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

DECSAI

DECSAI

Contenedores Lineales

- Las estructuras de datos lineales se caracterizan porque consisten en una secuencia de elementos del mismo tipo, $a_0, a_1, ..., a_n$, dispuestos a lo largo de una dimensión

- Ejemplo: Vector dinámico o celdas enlazadas

- Los contenedores lineales se comportan de forma idéntica independientemente del tipo de dato que almacenen (candidatos a ser implementados como clases plantillas)

Contenedores Lineales

- **Contenedor**: estructura que almacena datos de un mismo tipo base. Ej: vectores y listas.

- **Lineal:** contiene una secuencia de elementos dispuestos en una dimensión

- **Contenedores lineales:**
  - Pilas
  - Colas
  - Colas con prioridad
  - Listas

  ....
  ....

ESTRUCTURAS DE DATOS LINEALES

PILAS

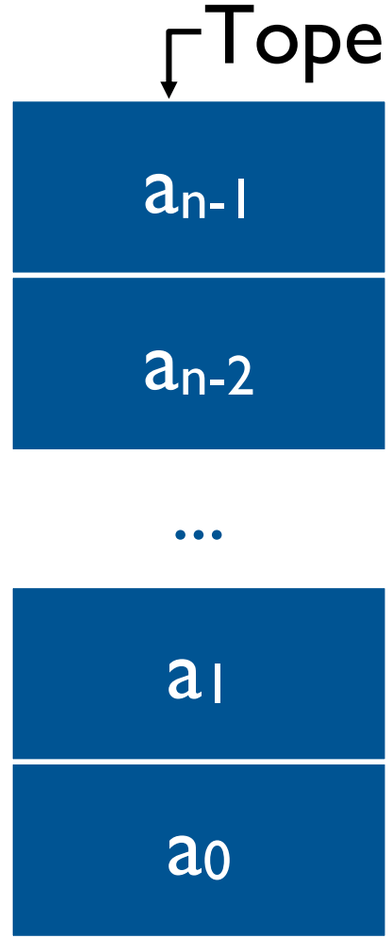
Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

DECSAI

Pilas

- 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**



Pilas

1. **Especificación**:

Contiene una secuencia de elementos $\{a_0, a_1, a_{n-1}\}$ en la que las inserciones, consultas y borrados se realizan sólo por uno de los extremos (Estructura LIFO)

- Las consultas se realizan sobre a_{n-1}
- Los borrados se hacen sobre a_{n-1}
- Las inserciones se hacen sobre a_{n-1}
- No se puede acceder a la pila por otro lado que no sea el tope

Pilas

2. **Operación:**
   - Tope: consulta el elemento del tope
   - Vacía: devuelve true si la pila no tiene ningún elemento
   - Quitar (pop): elimina le elemento en el tope
   - Poner (push): inserta un nuevo elemento en el tope

3. **Implementación**:
   - Basada en vectores estáticos
   - Basada en vectores dinámicos
   - Basada en celdas enlazadas

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?

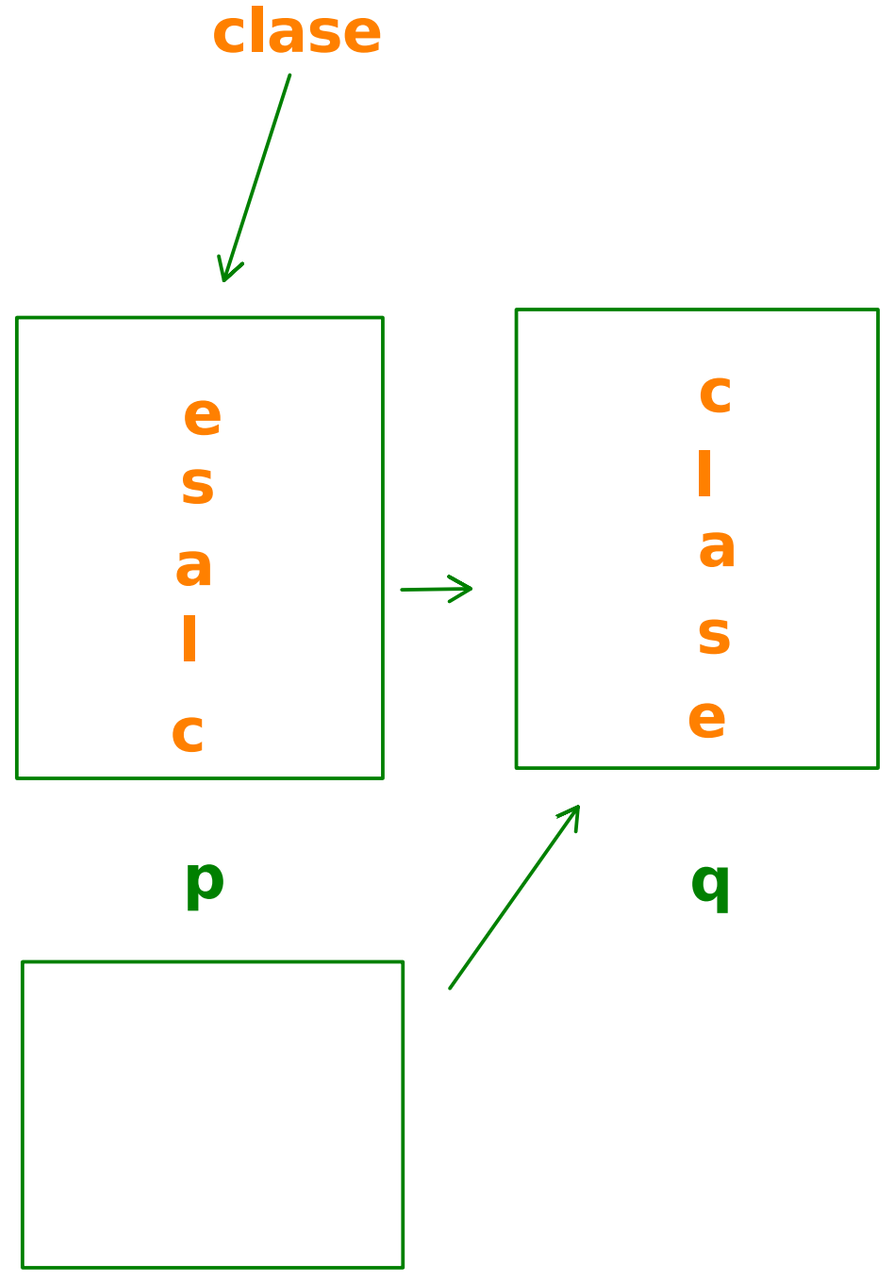
Ejemplo

- Invertir frase

Pilas

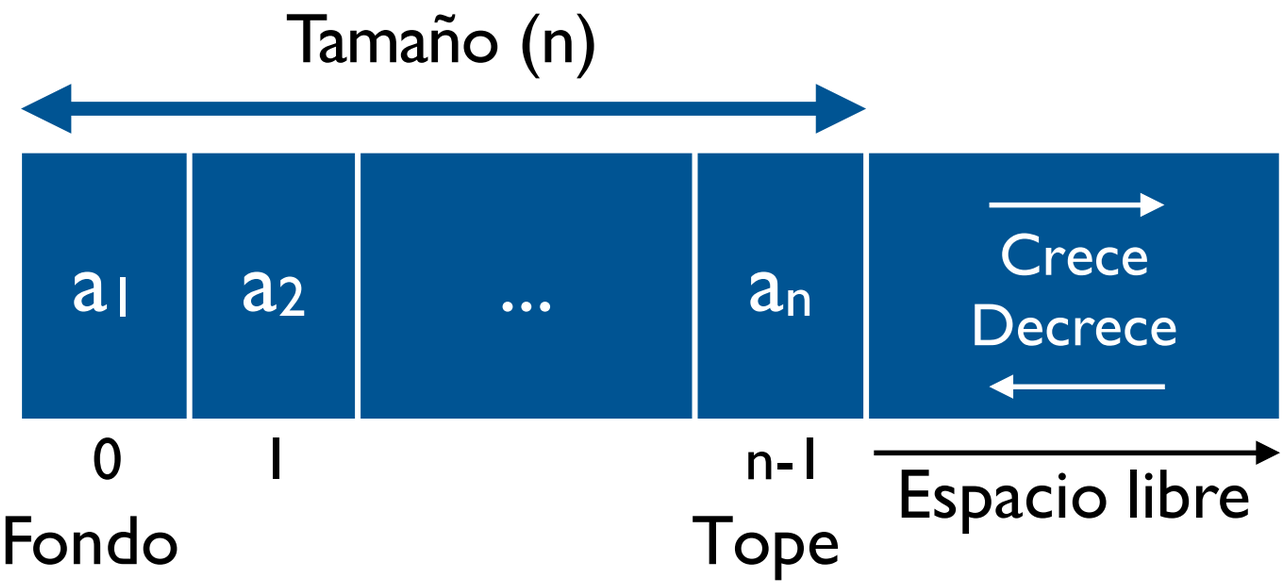
```
#include <iostream>
#include "Pila.hpp"
using namespace std;

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;
}
```

Uso de una pila



Pilas. Implementación con vectores

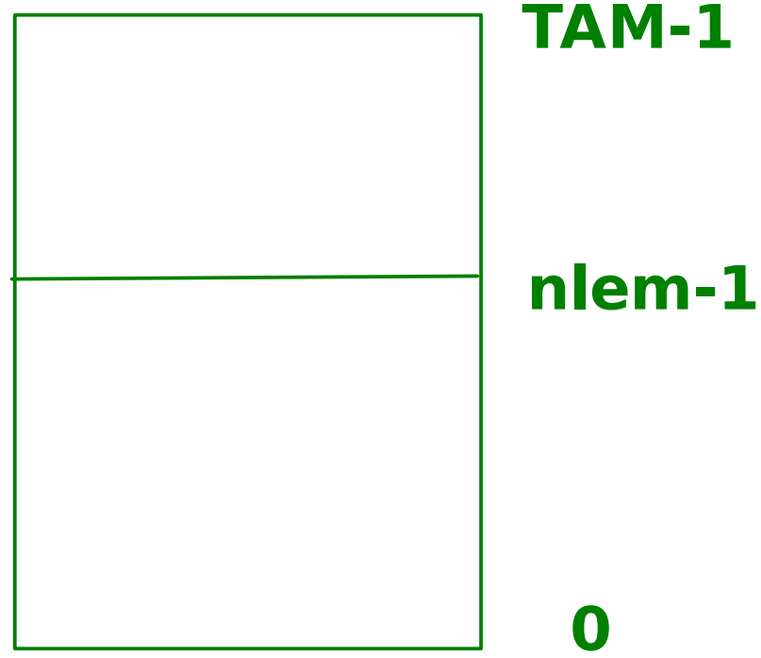
Almacenamos la secuencia de valores en un vector



- 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
- Tope ocupa la posición n-1
- Para borrar, se decrementa n en 1

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];
}
```

//int x=p.tope();
//p.tope()=25;

- 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)

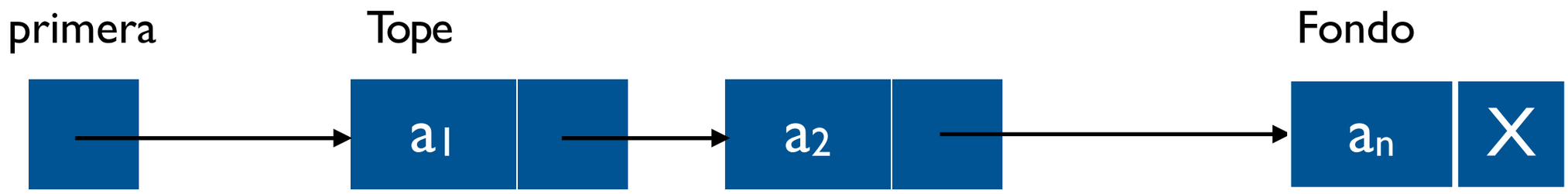
```
#include <cassert>
#include "Pila.hpp"

//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

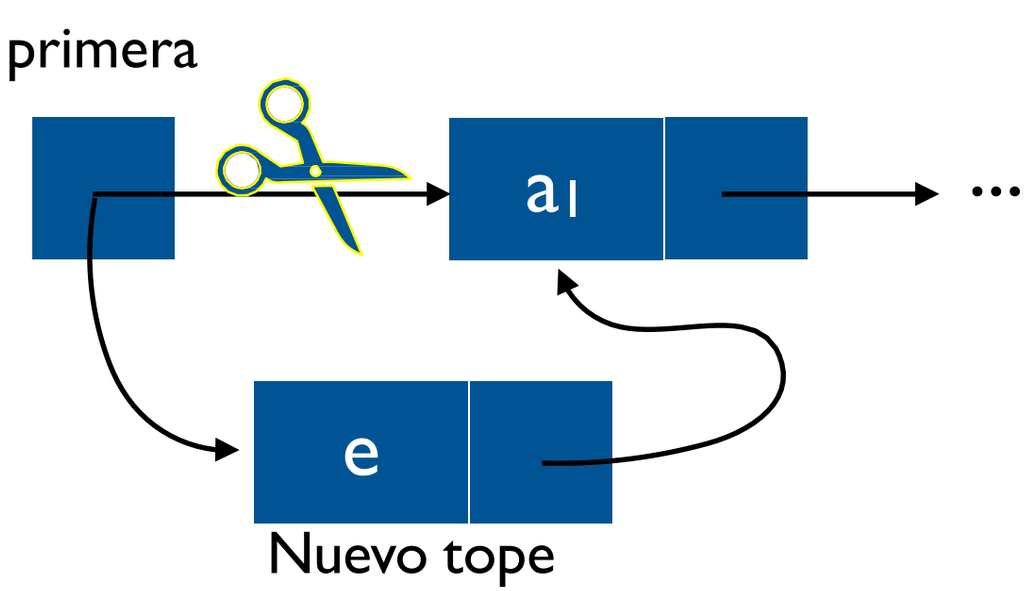
```
const Tbase & Pila::tope() const{
  assert(nelem>0);
  return datos[nelem-1];
}
```

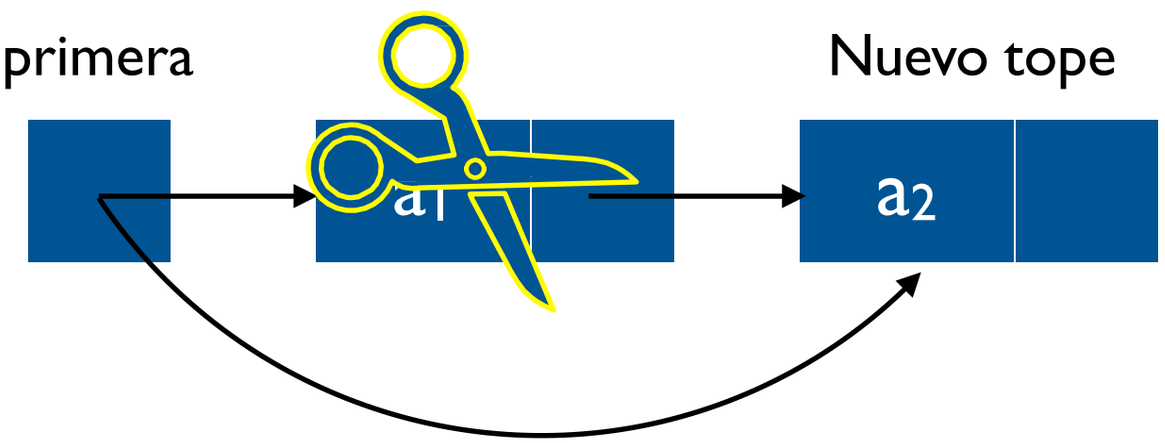
Pilas. Implementación con celdas enlazadas

Almacenamos la secuencia de valores en celdas enlazadas



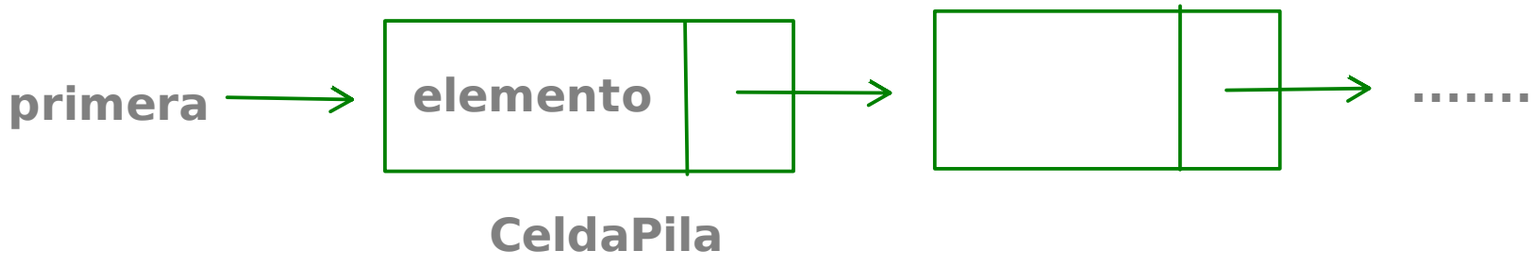
- Una pila vacía tiene un puntero (primera) nulo
- El tope de la pila está en la primera celda (muy eficiente)
- La inserción y borrado de elementos se hacen sobre la primera celda





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

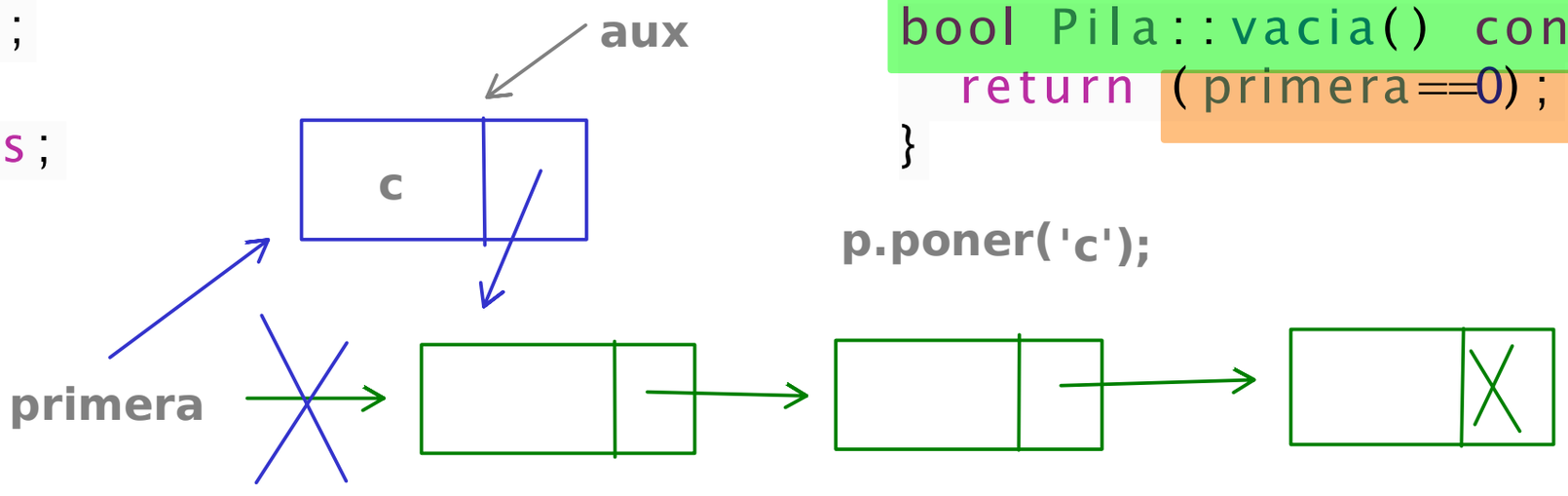
```
#include "Pila.hpp"

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

Eficiencia

Eficiencia

Pila	VD	Celdas
Poner	$O(1)$ [1]	$O(1)$
Quitar	$O(1)$ [1]	$O(1)$
Vacia	$O(1)$	$O(1)$
Tope	$O(1)$	$O(1)$

- Las operaciones Poner y Quitar usando para representar a la Pila un vector dinámico son constantes ya que estamos considerando el tiempo amortizado, lo que supone un promedio de inserciones y borrados sucesivos
- Usando las celdas, en el peor de los casos todas las funciones tienen un tiempo constante.

Ejemplo

- Convertir número decimal a binario

Ejemplo

- Convertir número decimal a binario

```
1  #include "Pila.h"
2  #include <iostream>
3  using namespace std;
4  int main(){
5  Pila<int> mipila;
6  int numero;
7   cout<<"Introduce un numero:";
8   cin>>numero;
9   while (numero>0){
10     int digit=(numero %2);
11     mipila.Poner(digit);
12     numero = numero /2;
13  }
14  while (!mipila.Vacia()){
15     int digit = mipila.Tope();
16     mipila.Quitar();
17     cout<<digit;
18   }
19 }
```

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;
}
```

Pilas (vectores).

Pila.h

```
template <class T>
class Pila{
  private:
    Vector<T> v;
    int num_elem;
  public:
    Pila(): v(1), num_elem(0)
      {}
    Pila(const Pila<T> & p): v(p.v),num_elem(p.num_elem)
      {}
    ~Pila()
      {}
    Pila& operator= (const Pila<T>& p)
    {
      v=p.v;
      num_elem= num_elem;
    }
    bool Vacia() const {return num_elem==0;}
    T& Tope ()
      {
        assert(num_elem! =0);
        return v[num_elem-1];
      }
```

Pila.h

```
    const T & Tope () const
    {
      assert(num_elem! =0);
      return v[num_elem-1];
    }
    void Poner(const T & elem)
    {
      if (num_elem==v.size())
        v.Resize(2*num_elem);
      v[num_elem]= elem;
      num_elem++;
    }
    void Quitar()
    {
      assert(num_elem! =0);
      num_elem--;
      if (num_elem<v.size()/4)
        v.Resize(v.size()/2);
    }
    int Num_elementos() const
      { return num_elem; }
};
```

Pilas (celdas enlazadas).

Pila.h

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

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

Pila.cpp

```
template <class T>
Pila<T>::~Pila()
{
  Celda *aux;
  while (primera! =0) {
    aux= primera;
    primera=primera→siguiente;
    delete aux;
  }
}

template <class T>
void Pila<T>::Poner(const T & elem)
{
  primera= new Celda(elem,primera);
  num_elem++;
}

template <class T>
void Pila<T>::Quitar()
{
  assert(primera! =0);
  Celda *aux=primera;
  primera= primera→siguiente;
  delete aux;
  num_elem--;
}
```