Análisis de Algoritmos y Estructuras de Datos

Tema 5: Tipo Abstracto de Datos Pila

Mª Teresa García Horcajadas José Fidel Argudo Argudo
Antonio García Domínguez Francisco Palomo Lozano

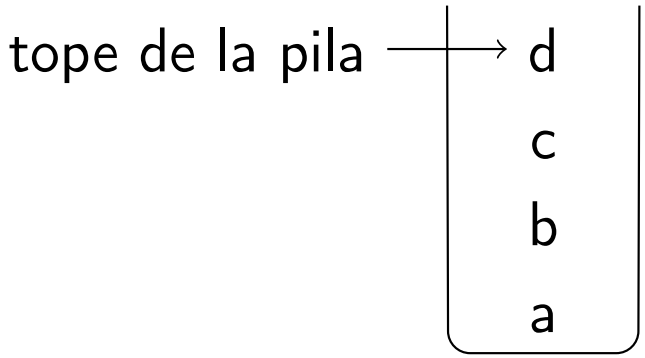
UCA
Universidad de Cádiz

Versión 2.0

Índice

Definición de Pila

- Una pila es una secuencia de elementos en la que todas las operaciones se realizan por un extremo de la misma. Dicho extremo recibe el nombre de tope, cima, cabeza...
- En una pila el último elemento añadido es el primero en salir de ella, por lo que también se les conoce como estructuras LIFO: *Last Input First Output*



Especificación del TAD *Pila*

Definición:

Una pila es una secuencia de elementos de un tipo determinado, en la cual se pueden añadir y eliminar elementos sólo por uno de sus extremos llamado tope o cima.

Operaciones:

Pila()

Postcondiciones: Crea una pila vacía.

bool vacia() const

Postcondiciones: Devuelve `true` si la pila está vacía.

const tElemento& tope() const

Precondiciones: La pila no está vacía.

Postcondiciones: Devuelve el elemento del tope de la pila.

Especificación del TAD *Pila*

`void pop()`

Precondiciones: La pila no está vacía.

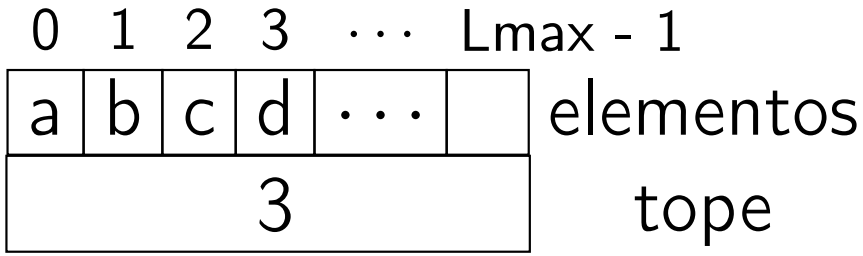
Postcondiciones: Elimina el elemento del tope de la pila y el siguiente se convierte en el nuevo tope.

`void push(const tElemento& x)`

Postcondiciones: Inserta el elemento x en el tope de la pila y el antiguo tope pasa a ser el siguiente.

Implementación vectorial estática

Tamaño de la pila definido por el diseñador del TAD mediante una constante.



Implementación vectorial estática (pilavec0.h)

```
1  #ifndef PILA_VEC0_H
2  #define PILA_VEC0_H

4  class Pila {
5  public:
6     typedef int tElemento; // por ejemplo
7     Pila();
8     bool vacia() const;
9     bool llena() const; // Requerida por la implementación
10    const tElemento& tope() const;
11    void pop();
12    void push(const tElemento& x);
13 private:
14    static const int Lmax = 100; // Longitud máxima de una pila
15    tElemento elementos[Lmax]; // vector de elementos
16    int tope_; // posición del tope
17 };

19 #endif // PILA_VEC0_H
```

Implementación vectorial estática (pilavec0.cpp)

```
1  #include <cassert>
2  #include "pilavec0.h"

4  Pila::Pila() : tope_(-1)
5  {}

7  bool Pila::vacia() const
8  {
9     return (tope_ == -1);
10 }

12 bool Pila::llena() const
13 {
14    return (tope_ == Lmax - 1);
15 }
```

Implementación vectorial estática (pilavec0.cpp)

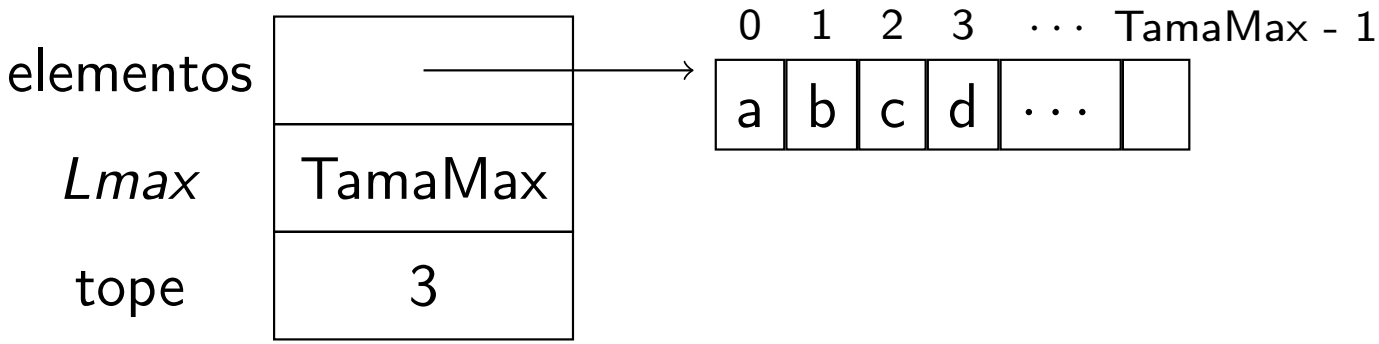
```
17 const Pila::tElemento& Pila::tope() const
18 {
19    assert(!vacia());
20    return elementos[tope_];
21 }

23 void Pila::pop()
24 {
25    assert(!vacia());
26    --tope_;
27 }

29 void Pila::push(const tElemento& x)
30 {
31    assert(!llena());
32    ++tope_;
33    elementos[tope_] = x;
34 }
```

Implementación vectorial pseudoestática

Tamaño de la pila definido en su creación por el usuario del TAD.



Implementación vectorial pseudoestática

Copia y destrucción de objetos en C++

Aspectos a considerar

1. Clase: Módulo que encapsula datos (atributos) y operaciones (métodos). Extiende el concepto de estructura de C.
2. En C++ `struct` y `class` son palabras reservadas sinónimas.
   - `struct`: Por defecto, miembros públicos (retrocompatibilidad con C)
   - `class`: Por defecto, miembros privados
3. En C están permitidas la copia y asignación entre estructuras del mismo tipo.
4. En C++, por preservar la compatibilidad, también se permite la copia y asignación de estructuras/clases del mismo tipo.

Implementación vectorial pseudoestática

Copia y destrucción de objetos en C++

Constructor de copia

Toda clase definida en C++ tiene un método llamado constructor de copia, que por defecto copia uno a uno los atributos. Si este comportamiento no es válido para una clase en particular, se debe redefinir. El constructor de copia se invoca cuando:

- se pasan parámetros por valor
- una función devuelve el resultado por valor
- se inicializa un objeto a partir de otro

Implementación vectorial pseudoestática

Copia y destrucción de objetos en C++

Operador de asignación

Toda clase definida en C++ tiene sobrecargado el operador = como miembro de la clase, que por defecto asigna uno a uno los atributos. Si esta asignación no es correcta para una clase en particular, se debe redefinir.

Destructor

Toda clase definida en C++ tiene un método llamado destructor que se invoca cuando un objeto termina su vida o, automáticamente, cuando se abandona el ámbito donde se ha definido. Por defecto, el destructor no hace nada especial, simplemente destruye uno por uno los atributos, por lo que puede ser necesario redefinirlo.

Copia y destrucción de objetos en C++

```
1  #include <iostream>
2  #include "pilavec0.h" // Implementación vectorial estática
3  using namespace std;

5  void imprimir(Pila P) // Parámetro por valor/copia
6  {
7     while (!P.vacia()) {
8        cout << '␣' << P.tope();
9        P.pop();
10    } // Destrucción implícita de P
11 }

13 Pila fun(Pila P) // Parámetro y resultado por valor/copia
14 {
15    Pila R(P); // Inicialización por copia
16    cout << "En␣fun()" << endl;
17    cout << "P:"; imprimir(P); cout << endl;
18    cout << "R:"; imprimir(R); cout << endl;
19    return R; // Devolución por valor/copia
20    // Destrucción implícita de R y P
21 }
```

Copia y destrucción de objetos en C++

```
23 int main()
24 {
25    Pila P, Q;
26    for (int i = 0; i < 10; ++i)
27       P.push(i);
28    Q = fun(P); // Asignación
29    cout << "En␣main()" << endl;
30    cout << "P:"; imprimir(P); cout << endl;
31    cout << "Q:"; imprimir(Q); cout << endl;
32    // Destrucción ímplicita de Q y P
33 }
```

Salida del programa

```
En fun()
P: 9 8 7 6 5 4 3 2 1 0
R: 9 8 7 6 5 4 3 2 1 0
En main()
P: 9 8 7 6 5 4 3 2 1 0
Q: 9 8 7 6 5 4 3 2 1 0
```

Implementación vectorial pseudoestática

Copia y destrucción de objetos en C++

Estructuras en memoria dinámica

Supongamos una clase que tiene atributos que son punteros a estructuras en memoria dinámica:

1. La copia y la asignación atributo a atributo crean «alias» de las zonas de memoria dinámica (se copian los punteros, pero no las posiciones de memoria apuntadas). Cambiar este comportamiento requiere definir el constructor de copia y el operador de asignación para la clase en cuestión.
2. El destructor por defecto elimina, uno por uno, los atributos de un objeto, incluidos los de tipo puntero, pero no libera la memoria dinámica a la que estos apuntan. Para liberarla es necesario definir el destructor de la clase.

Implementación vectorial pseudoestática

Copia y destrucción de objetos en C++

Implementación de constructor de copia, asignación y destructor

El comportamiento de un TAD ante las operaciones de copia, asignación y destrucción debe ser análogo al de los tipos del lenguaje de programación. Si detectamos que el comportamiento por defecto de una cualquiera de estas tres operaciones no es válido, generalmente, tampoco será válido el de las otras dos y deberemos implementar las tres.

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Operador new

- Para reservar memoria dinámica se utiliza una expresión como

  ```
  new tipo
  ```

  que devuelve la dirección de un bloque de memoria del tamaño requerido para el `tipo`.
- Si la reserva falla, se lanza una excepción estándar `bad_alloc`.
- Si se trata de un tipo del lenguaje, se puede escribir un valor de inicialización entre paréntesis.

  ```
  new tipo(inicializador)
  ```
- Si el tipo es una clase, la memoria reservada se inicializa con el constructor de la clase. La lista de parámetros, si es necesaria, se escribe a continuación entre paréntesis.

  ```
  new clase(param1, param2,...)
  ```

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Operador new [ ]

- Una expresión como

  ```
  new tipo[n]
  ```

  reserva memoria dinámica para un vector de *n* elementos del `tipo` y devuelve su dirección.
- El valor *n* debe ser una expresión de tipo `unsigned`.
- Si el tipo es una clase, cada posición del vector se inicializa con el constructor predeterminado de la clase (el que se puede invocar sin parámetros).
- No hay posibilidad de utilizar otro constructor y si la clase no dispone del predeterminado, el uso del operador `new []` provocará un error de compilación.

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Operador delete

- La expresión

  ```
  delete p
  ```

  libera la memoria a la que apunta p, que debe haber sido reservada con `new`.
- Si p apunta a un objeto, previamente se llama al destructor.
- Si p es un puntero nulo, el operador `delete` no hace nada.

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Operador delete [ ]

- La expresión

  ```
  delete[] p
  ```

  libera la memoria ocupada por el vector al que apunta p, que deberá haber sido reservada con el operador `new[]`.
- Si el tipo del vector es una clase, primero se llama al destructor de la clase con cada objeto del vector.

Implementación vectorial pseudoestática (pilavec1.h)

```
1  #ifndef PILA_VEC1_H
2  #define PILA_VEC1_H
3  class Pila {
4  public:
5     typedef int tElemento; // por ejemplo
6     explicit Pila(unsigned TamaMax); // constructor
7     Pila(const Pila& P); // ctor. de copia
8     Pila& operator =(const Pila& P); // asignación entre pilas
9     bool vacia() const;
10    bool llena() const; // Requerida por la implementación
11    const tElemento& tope() const;
12    void pop();
13    void push(const tElemento& x);
14    ~Pila(); // destructor
15 private:
16    tElemento *elementos; // vector de elementos
17    int Lmax; // tamaño del vector
18    int tope_; // posición del tope
19 };
20 #endif // PILA_VEC1_H
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
1  #include <cassert>
2  #include "pilavec1.h"

4  // Constructor
5  Pila::Pila(unsigned TamaMax) :
6     elementos(new tElemento[TamaMax]),
7     Lmax(TamaMax),
8     tope_(-1)
9  {}

11 // Constructor de copia
12 Pila::Pila(const Pila& P) :
13    elementos(new tElemento[P.Lmax]),
14    Lmax(P.Lmax),
15    tope_(P.tope_)
16 {
17    for (int i = 0; i <= tope_; i++) // copiar el vector
18       elementos[i] = P.elementos[i];
19 }
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
21 // Asignación entre pilas
22 Pila& Pila::operator =(const Pila& P)
23 {
24    if (this != &P) { // evitar autoasignación
25       // Destruir el vector y crear uno nuevo si es necesario
26       if (Lmax != P.Lmax) {
27          delete [] elementos;
28          Lmax = P.Lmax;
29          elementos = new tElemento[Lmax];
30       }
31       // Copiar el vector
32       tope_ = P.tope_;
33       for (int i = 0; i <= tope_; i++)
34          elementos[i] = P.elementos[i];
35    }
36    return *this;
37 }
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
39 bool Pila::vacia() const
40 {
41    return (tope_ == -1);
42 }

44 bool Pila::llena() const
45 {
46    return (tope_ == Lmax - 1);
47 }

49 const Pila::tElemento& Pila::tope() const
50 {
51    assert(!vacia());
52    return elementos[tope_];
53 }
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
55 void Pila::pop()
56 {
57    assert(!vacia());
58    --tope_;
59 }

61 void Pila::push(const tElemento& x)
62 {
63    assert(!llena());
64    ++tope_;
65    elementos[tope_] = x;
66 }

68 // Destructor
69 Pila::~Pila()
70 {
71    delete[] elementos;
72 }
```

Implementación genérica vectorial pseudoestática

Plantillas (templates)

- En C++ una plantilla es una definición genérica de una familia de clases (o funciones), que difieren en detalles (como algunos tipos de datos usados) de los cuales no depende el concepto representado. A partir de la plantilla el compilador puede generar una clase (o función) específica.
- Mediante una plantilla de clase realizaremos una implementación genérica de un TAD y después en los programas usaremos las clases específicas que el compilador generará automáticamente.

Implementación genérica vectorial pseudoestática

Definición de plantillas

- Una clase (o función) se generaliza definiendo una plantilla con parámetros formales que pueden ser tipos o valores.

```
1 template <typename T1, typename T2,..., tipo1 param1,...>
2 class C {
3     // declaraciones/definiciones de miembros
4     // en los que se usan los tipos T1, T2,...
5     // y valores constantes como param1
6 };
```

- Al definir una plantilla se presuponen propiedades de los parámetros formales que se convierten en requisitos que deben satisfacer los parámetros reales, de lo contrario se producen errores de compilación. Por ejemplo, que el tipo `T1` tenga constructor predeterminado, que sus valores se puedan comparar con los operadores relacionales (==, <, >,...), etc.

Implementación genérica vectorial pseudoestática

Ejemplo

```
1  template <typename tElemento> class Pila {
2  public:
3     explicit Pila(unsigned TamaMax); // requiere ctor. tElemento()
4     void push(const tElemento& x);
5     // ... declaraciones del resto de miembros
6  };

8  // Las funciones miembro de una plantilla de clase se definen
9  // como plantillas de funciones.
10 template <typename tElemento>
11 Pila<tElemento>::Pila(unsigned TamaMax) {
12    // ...
13 }
14 template <typename tElemento>
15 Pila<tElemento>::push(const tElemento& x) {
16    // ...
17 }
```

Implementación genérica vectorial pseudoestática

Instanciación de plantillas

Las clases (o funciones) específicas las genera automáticamente el compilador cuando especializamos la plantilla al proporcionar los parámetros reales.

```
#include "pila.h" // definición de la plantilla Pila<T>

Pila<char> P1(20); // pila de caracteres, de longitud 20
Pila<double> P2(150); // pila de double, de longitud 150
Pila<string> P3(100); // pila de string, de longitud 100
Pila<Pila<int>> P4(5); // Error, Pila<int> no dispone
                       // de ctor. predeterminado
```

Implementación genérica vectorial pseudoestática

Organización del código fuente

- El código de una clase habitualmente se separa en dos partes:
  1. Una cabecera (fichero `.h`) con sólo las declaraciones de todos los miembros de la clase (métodos y atributos).
  2. La definición de los métodos de la clase en un fichero `.cpp`
- Por razones técnicas e históricas los compiladores de C++ no ofrecen un buen mecanismo de generación automática de especializaciones de plantillas mediante la compilación separada de las definiciones y sus usos.

Organización del código fuente

Una plantilla de clase que implementa un TAD genérico la definiremos completamente en un fichero de cabecera (`.h`), que incluiremos en cada unidad de compilación en la que se utilice.

Implementación genérica vectorial pseudoestática

```
1  #ifndef PILA_VEC_H
2  #define PILA_VEC_H
3  #include <cassert>

5  template <typename tElemento> class Pila {
6  public:
7     explicit Pila(unsigned TamaMax); // ctor., requiere ctor. tElemento()
8     Pila(const Pila& P); // ctor. de copia
9     Pila& operator =(const Pila& P); // asignación entre pilas
10    bool vacia() const;
11    bool llena() const; // Requerida por la implementación
12    const tElemento& tope() const;
13    void pop();
14    void push(const tElemento& x);
15    ~Pila(); // destructor
16 private:
17    tElemento *elementos; // vector de elementos
18    int Lmax; // tamaño del vector
19    int tope_; // posición del tope
20 };
```

Implementación genérica vectorial pseudoestática

```
22 template <typename tElemento>
23 inline Pila<tElemento>::Pila(unsigned TamaMax) :
24    elementos(new tElemento[TamaMax]),
25    Lmax(TamaMax),
26    tope_(-1)
27 {}

29 template <typename tElemento>
30 Pila<tElemento>::Pila(const Pila<tElemento>& P) :
31    elementos(new tElemento[P.Lmax]),
32    Lmax(P.Lmax),
33    tope_(P.tope_)
34 {
35    for (int i = 0; i <= tope_; i++) // copiar el vector
36       elementos[i] = P.elementos[i];
37 }
```

Implementación genérica vectorial pseudoestática

```
39 template <typename tElemento>
40 Pila<tElemento>& Pila<tElemento>::operator =(const Pila<tElemento>&
41 {
42    if (this != &P) { // evitar autoasignación
43       // Destruir el vector y crear uno nuevo si es necesario
44       if (Lmax != P.Lmax) {
45          delete[] elementos;
46          Lmax = P.Lmax;
47          elementos = new tElemento[Lmax];
48       }
49       // Copiar el vector
50       tope_ = P.tope_;
51       for (int i = 0; i <= tope_; i++)
52          elementos[i] = P.elementos[i];
53    }
54    return *this;
55 }
```

Implementación genérica vectorial pseudoestática

```
57 template <typename tElemento>
58 inline bool Pila<tElemento>::vacia() const
59 {
60    return (tope_ == -1);
61 }

63 template <typename tElemento>
64 inline bool Pila<tElemento>::llena() const
65 {
66    return (tope_ == Lmax - 1);
67 }

69 template <typename tElemento>
70 inline const tElemento& Pila<tElemento>::tope() const
71 {
72    assert(!vacia());
73    return elementos[tope_];
74 }
```

Implementación genérica vectorial pseudoestática

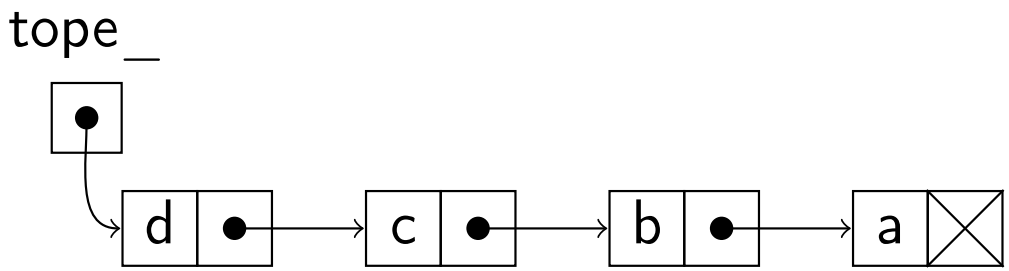
```
76 template <typename tElemento>
77 inline void Pila<tElemento>::pop()
78 {
79    assert(!vacia());
80    --tope_;
81 }

83 template <typename tElemento>
84 inline void Pila<tElemento>::push(const tElemento& x)
85 {
86    assert(!llena());
87    ++tope_;
88    elementos[tope_] = x;
89 }

91 template <typename tElemento>
92 inline Pila<tElemento>::~Pila()
93 { delete[] elementos; }

95 #endif // PILA_VEC_H
```

Implementación genérica mediante celdas enlazadas



Implementación genérica mediante celdas enlazadas

```
1  #ifndef PILA_ENLA_H
2  #define PILA_ENLA_H
3  #include <cassert>

5  template <typename T>
6  class Pila {
7  public:
8     Pila(); // constructor
9     Pila(const Pila<T>& P); // ctor. de copia
10    Pila<T>& operator =(const Pila<T>& P); // asignación
11    bool vacia() const;
12    const T& tope() const;
13    void pop();
14    void push(const T& x);
15    ~Pila(); // destructor
```

Implementación genérica mediante celdas enlazadas

```
16 private:
17    struct nodo {
18       T elto;
19       nodo* sig;
20       nodo(const T& e, nodo* p = nullptr): elto(e), sig(p) {}
21    };

23    nodo* tope_;

25    void copiar(const Pila<T>& P);
26 };
```

Implementación genérica mediante celdas enlazadas

```
28 template <typename T>
29 inline Pila<T>::Pila() : tope_(nullptr) {}

31 template <typename T>
32 Pila<T>::Pila(const Pila<T>& P) : tope_(nullptr)
33 {
34    copiar(P);
35 }

37 template <typename T>
38 Pila<T>& Pila<T>::operator =(const Pila<T>& P)
39 {
40    if (this != &P) { // evitar autoasignación
41       this->~Pila(); // vaciar la pila actual
42       copiar(P);
43    }
44    return *this;
45 }
```

Implementación genérica mediante celdas enlazadas

```
47 template <typename T>
48 inline bool Pila<T>::vacia() const
49 { return (!tope_); }

51 template <typename T>
52 inline const T& Pila<T>::tope() const
53 {
54    assert(!vacia());
55    return tope_->elto;
56 }

58 template <typename T>
59 inline void Pila<T>::pop()
60 {
61    assert(!vacia());
62    nodo* p = tope_;
63    tope_ = p->sig;
64    delete p;
65 }
```

Implementación genérica mediante celdas enlazadas

```
67 template <typename T>
68 inline void Pila<T>::push(const T& x)
69 {
70    tope_ = new nodo(x, tope_);
71 }

73 // Destructor: vacía la pila
74 template <typename T>
75 Pila<T>::~Pila()
76 {
77    nodo* p;
78    while (tope_) {
79       p = tope_->sig;
80       delete tope_;
81       tope_ = p;
82    }
83 }
```

Implementación genérica mediante celdas enlazadas

```
85  // Método privado
86  template <typename T>
87  void Pila<T>::copiar(const Pila<T>& P)
88  {
89     if (!P.vacia()) {
90        tope_ = new nodo(P.tope()); // copiar el primer elto
91        // Copiar el resto de elementos hasta el fondo de la pila.
92        nodo* p = tope_; // recorre la pila destino
93        nodo* q = P.tope_->sig; // 2º nodo, recorre la pila origen
94        while (q) {
95           p->sig = new nodo(q->elto);
96           p = p->sig;
97           q = q->sig;
98        }
99     }
100 }

102 #endif // PILA_ENLA_H
```