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

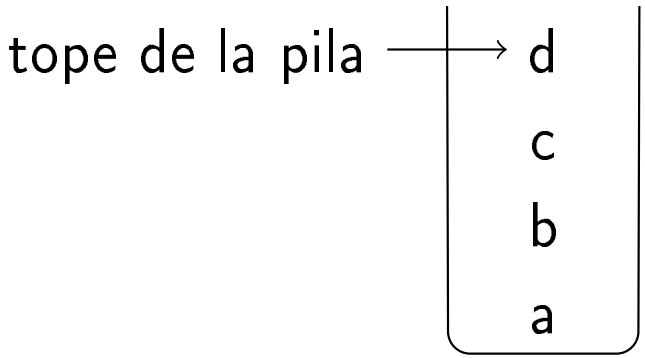
UCA
Universidad de Cádiz

Versión 3.0

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...
- El último elemento añadido a una pila es el primero en salir de ella, por lo que también se conoce como estructura LIFO: *Last Input First Output*



Especificación del TAD *Pila*

Definición:

Una pila es una secuencia de elementos de un mismo tipo T, 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.

size_t tama() const

Postcondiciones: Devuelve el número de elementos que contiene la pila.

const T& 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, si existe, se convierte en el nuevo tope.

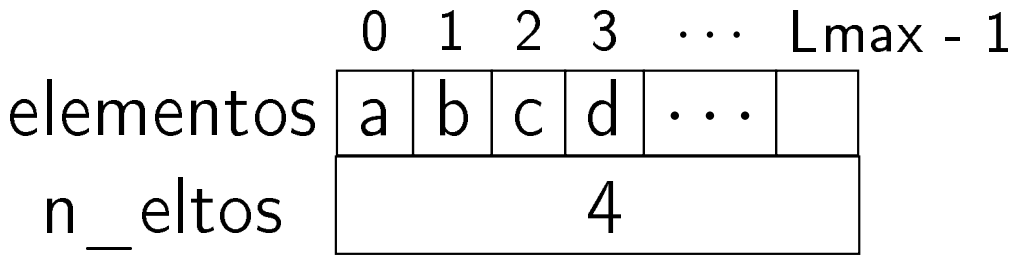
`void push(const T& x)`

Postcondiciones: Inserta el elemento x en el tope de la pila y, si no estaba vacía, el antiguo tope pasa a ser el siguiente.

Implementación vectorial estática

Implementación vectorial estática

Capacidad de la pila definida por el diseñador del TAD mediante una constante.



Implementación vectorial estática (pilavec0.h)

```
1  #ifndef PILA_VEC0_H
2  #define PILA_VEC0_H
3  #include <cstddef>    // size_t

5  class Pila {
6  public:
7     typedef int T; // Por ejemplo
8     Pila();
9     bool vacia() const;
10    size_t tama() const;
11    size_t tamaMax() const; // Requerida por la implementación
12    const T& tope() const;
13    void pop();
14    void push(const T& x);
15 private:
16    static const size_t Lmax = 100; // Longitud máx. de una pila
17    T elementos[Lmax];              // Vector de elementos
18    size_t n_eltos;                 // Tamaño de la pila
19 };

21 #endif // PILA_VEC0_H
```

Implementación vectorial estática (pilavec0.cpp)

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

4  Pila::Pila() : n_eltos(0)
5  {}

7  bool Pila::vacia() const
8  {
9     return (n_eltos == 0);
10 }

12 size_t Pila::tama() const
13 {
14    return n_eltos;
15 }

17 size_t Pila::tamaMax() const
18 {
19    return Lmax;
20 }
```

Implementación vectorial estática (pilavec0.cpp)

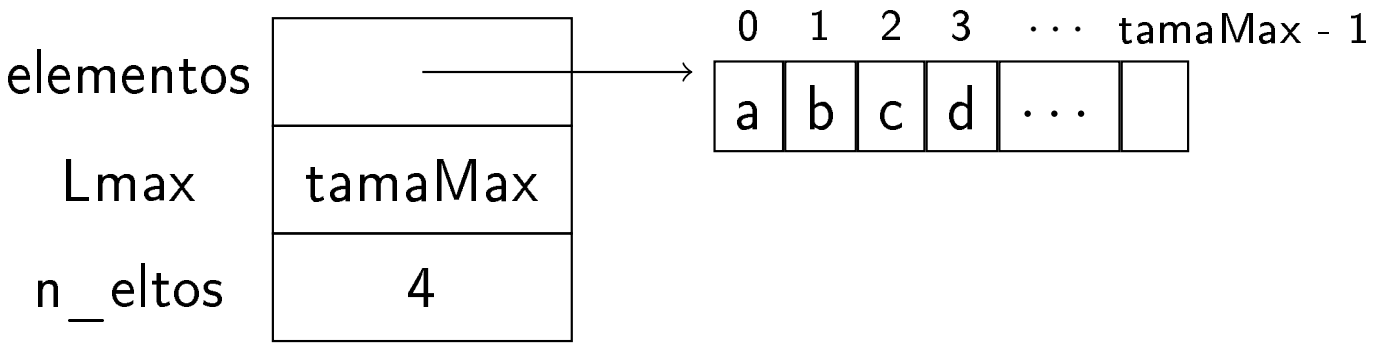
```
22 const Pila::T& Pila::tope() const
23 {
24    assert(!vacia());
25    return elementos[n_eltos - 1];
26 }

28 void Pila::pop()
29 {
30    assert(!vacia());
31    --n_eltos;
32 }

34 void Pila::push(const T& x)
35 {
36    assert(n_eltos < Lmax);
37    elementos[n_eltos] = x;
38    ++n_eltos;
39 }
```

Implementación vectorial pseudoestática

Capacidad de la pila definida 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 variables del mismo tipo de estructura.
4. En C++, por preservar la compatibilidad, también se permite la copia y asignación de variables/objetos del mismo tipo de estructura/clase.

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 conocido como destructor, que se invoca automáticamente cuando un objeto termina su vida, p.e., cuando se abandona el ámbito donde se ha definido un objeto local. Por defecto el destructor no hace nada especial, por lo que se debe redefinir si son necesarias algunas tareas previas a la destrucción en sí, como liberar recursos usados por el objeto (memoria dinámica, ficheros, puertos, etc.)

Copia y destrucción de objetos en C++

```
1  #include <iostream>
2  #include "pilavec0.h" // Ctor. de copia, asignación y dtor. por defecto
3  using namespace std;

5  void imprimir(Pila P)    // Parámetro por valor/copia
6  {
7     while (P.tama() > 0) {
8        cout << '␣' << P.tope();
9        P.pop();
10    }
11 }  // Destrucción de P

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 de R y P
```

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
```

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. Por defecto, la destrucción de un objeto 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, 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 o llaves.

```
new tipo(inicializador)
```

- Si `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 o llaves.

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

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Ejemplos

```
1  class X {
2    int x;
3  public:
4    X(int i) : x{i} {}
5  };

7  int* pi = new int;                  // *pi no inicializado
8  unsigned* pu = new unsigned{};      // *pu == 0
9  double* pd = new double(1.);        // *pd == 1.0
10 racional* pr1 = new racional;       // *pr1 == 0/1
11 racional* pr2 = new racional(6, 14); // *pr2 == 3/7
12 // X* px1 = new X;                  // Error, falta ctor. X::X()
13 // X* px1 = new X();                // Error, falta ctor. X::X()
14 X* px1 = new X{0};                  // px1->x == 0
15 X* px2 = new X(4);                  // px2->x == 4
```

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Operador new [ ]

- La reserva de memoria dinámica para un vector de *n* elementos de un `tipo` y obtención de su dirección se realiza con la expresión

```
new tipo[n]
```

- El valor *n* debe ser una expresión de tipo `unsigned`.
- Opcionalmente, se puede añadir una lista de inicializadores de los elementos del vector, encerrada entre llaves.

```
new tipo[n]{ini_1, ini_2, ...,ini_n}
```

- Si `tipo` es una clase y no se usa lista de inicializadores, cada posición del vector se inicializa con el constructor predeterminado de la clase (el que no tiene parámetros).
- No es posible utilizar otro constructor y si la clase no dispone del predeterminado, el operador `new[]` provocará un error de compilación.

Implementación vectorial pseudoestática

Operadores de memoria dinámica en C++

Ejemplos

```
1  class X {
2    int x;
3  public:
4    X(int i) : x{i} {}
5  };

7  int* vi = new int[5];                  // 5 int no inicializados
8  unsigned* vu = new unsigned[3]{};      // 3 unsigned int a 0
9  char* vc = new char[6]{'a', 'b'};      // 2 char 'a' y 'b' y 4 char a 0
10 double* vd = new double[4]{1., 2.5, 3.8, 4.0}; // 4 double inic.
11 // X* vx = new X[10];                  // Error, falta ctor. X::X()
12 X* vx = new X[3]{2, 3, 4};            // 3 X inic. con X::X(int)
13 racional* vr1 = new racional[4];      // vr1 == {0/1, 0/1, 0/1, 0/1}
14 racional* vr2 = new racional[4]{1, {6, 3}, {12, 30}};
       // vr2 == {1/1, 2/1, 2/5, 0/1}
```

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 a su 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.
- Si p es un puntero nulo, el operador `delete[]` no hace nada.

Implementación vectorial pseudoestática (pilavec1.h)

```
1  #ifndef PILA_VEC1_H
2  #define PILA_VEC1_H
3  #include <cstddef>    // size_t

5  class Pila {
6  public:
7     typedef int T; // Por ejemplo
8     explicit Pila(size_t tamaMax);
9     bool vacia() const;
10    size_t tama() const;
11    size_t tamaMax() const; // Requerida por la implementación
12    const T& tope() const;
13    void pop();
14    void push(const T& x);
15    Pila(const Pila& P);                  // Ctor. de copia
16    Pila& operator =(const Pila& P); // Asignación entre pilas
17    ~Pila();                              // Destructor
18 private:
19    T* elementos;      // Vector de elementos
20    size_t Lmax,       // Tamaño del vector
21           n_eltos;    // Tamaño de la pila
22 };
23 #endif // PILA_VEC1_H
```

Implementación vectorial pseudoestática (pilavec1.cpp)

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

4  Pila::Pila(size_t tamaMax) :
5     elementos(new T[tamaMax]),
6     Lmax(tamaMax),
7     n_eltos(0)
8  {}

10 bool Pila::vacia() const
11 {
12    return (n_eltos == 0);
13 }

15 size_t Pila::tama() const
16 {
17    return n_eltos;
18 }
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
20 size_t Pila::tamaMax() const
21 {
22    return Lmax;
23 }

25 const Pila::T& Pila::tope() const
26 {
27    assert(!vacia());
28    return elementos[n_eltos - 1];
29 }

31 void Pila::pop()
32 {
33    assert(!vacia());
34    --n_eltos;
35 }
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
37 void Pila::push(const T& x)
38 {
39    assert(n_eltos < Lmax);
40    elementos[n_eltos] = x;
41    ++n_eltos;
42 }

44 // Constructor de copia
45 Pila::Pila(const Pila& P) :
46    elementos(new T[P.Lmax]),
47    Lmax(P.Lmax),
48    n_eltos(P.n_eltos)
49 {
50    for (size_t i = 0; i < n_eltos; ++i)    // Copiar elementos
51       elementos[i] = P.elementos[i];
52 }
```

Implementación vectorial pseudoestática (pilavec1.cpp)

```
54 // Asignación entre pilas
55 Pila& Pila::operator =(const Pila& P)
56 {
57    if (this != &P) {    // Evitar autoasignación
58       // Destruir el vector y crear uno nuevo si es necesario
59       if (Lmax != P.Lmax) {
60          delete[] elementos;
61          Lmax = P.Lmax;
62          elementos = new T[Lmax];
63       }
64       n_eltos = P.n_eltos;
65       for (size_t i = 0; i < n_eltos; ++i)   // Copiar elementos
66          elementos[i] = P.elementos[i];
67    }
68    return *this;
69 }

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

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 una plantilla el compilador puede generar una clase (o función) específica, llamada especialización de la plantilla.
- Mediante una plantilla de clase realizaremos una implementación genérica de un TAD y después en los programas usaremos las clases específicas (especializaciones) que el compilador generará automáticamente a partir de ella.

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 genéricos T1, T2,...
5     // y datos como param1 de tipos específicos como tipo1
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: generalización de la clase `Pila`

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

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

Implementación genérica vectorial pseudoestática

Instanciación de plantillas

- Las especializaciones las genera automáticamente el compilador cuando una plantilla es instanciada, es decir, al proporcionar los parámetros reales que sustituirán a los formales. Por tanto, una plantilla debe estar disponible para el compilador en el punto donde sea instanciada.

```
#include "pila.h"          // Definición de la plantilla de clase Pila<T>
                           // y de las plantillas de sus métodos.

Pila<char> P1(20);         // Pila de caracteres, de capacidad 20
                           // Genera la clase Pila<char> y
                           // el ctor. Pila<char>::Pila<char>
Pila<double> P2(150);      // Pila de double, de capacidad 150
Pila<string> P3(100);      // Pila de string, de capacidad 100
Pila<Pila<int>> P4(5);     // Pila de Pila<int>. Error: Pila<int>
                           // no dispone de ctor. predeterminado.
P1.push('A');              // Genera el método Pila<char>::push<char>,
// parámetro de plantilla de función deducido de parámetro de función.
```

Implementación genérica vectorial pseudoestática

Instanciación de plantillas (cont.)

- El compilador no traduce el código fuente de una plantilla, simplemente revisa su sintaxis.
- El código fuente de una especialización (generado por el compilador al instanciar la plantilla correspondiente) sí es traducido a código objeto.
- Ya que el compilador únicamente genera código objeto cuando una plantilla es instanciada, los métodos de una especialización de plantilla de clase no utilizados no serán compilados. Esto reduce el tiempo de compilación y el tamaño del ejecutable.

Implementación genérica vectorial pseudoestática

Organización del código fuente de una clase

Puesto que la interfaz de una clase es independiente de la implementación concreta de sus métodos, aplicando el principio de ocultación de información, el código se separa en dos partes:

1. Una cabecera (fichero `.h` o `.hpp`) que normalmente sólo contiene las declaraciones de los miembros de la clase (métodos y atributos).
2. La definición/implementación de los métodos de la clase en un fichero `.cpp`

Organización del código fuente de una plantilla de clase

Debido al modo en que el compilador realiza la instanciación, una plantilla de clase que implemente un TAD genérico la definiremos, junto a las plantillas de sus métodos, en una cabecera (`.h`) que incluiremos en cada unidad de compilación donde la utilicemos.

Implementación genérica vectorial pseudoestática (pilavec.h)

```
1  #ifndef PILA_VEC_H
2  #define PILA_VEC_H
3  #include <cstddef>    // size_t
4  #include <cassert>

6  template <typename T> class Pila {
7  public:
8     explicit Pila(size_t tamaMax); // Ctor. requiere ctor. T()
9     bool vacia() const;
10    size_t tama() const;
11    size_t tamaMax() const; // Requerida por la implementación
12    const T& tope() const;
13    void pop();
14    void push(const T& x);
15    Pila(const Pila& P);              // Ctor. de copia, req. ctor. T()
16    Pila& operator =(const Pila& P); // Asig. entre pilas, req. ctor. T()
17    ~Pila();                          // Destructor
18 private:
19    T* elementos;      // Vector de elementos
20    size_t Lmax,       // Tamaño del vector
21           n_eltos;    // Tamaño de la pila
22 };
```

Implementación genérica vectorial pseudoestática (pilavec.h)

```
24 template <typename T>
25 inline Pila<T>::Pila(size_t tamaMax) :
26    elementos(new T[tamaMax]),
27    Lmax(tamaMax),
28    n_eltos(0)
29 {}

31 template <typename T>
32 inline bool Pila<T>::vacia() const
33 {
34    return (n_eltos == 0);
35 }

37 template <typename T>
38 inline size_t Pila<T>::tama() const
39 {
40    return n_eltos;
41 }
```

Implementación genérica vectorial pseudoestática (pilavec.h)

```
43 template <typename T>
44 inline size_t Pila<T>::tamaMax() const
45 {
46    return Lmax;
47 }

49 template <typename T>
50 inline const T& Pila<T>::tope() const
51 {
52    assert(!vacia());
53    return elementos[n_eltos - 1];
54 }

56 template <typename T>
57 inline void Pila<T>::pop()
58 {
59    assert(!vacia());
60    --n_eltos;
61 }
```

Implementación genérica vectorial pseudoestática (pilavec.h)

```
63 template <typename T>
64 inline void Pila<T>::push(const T& x)
65 {
66    assert(n_eltos < Lmax);
67    elementos[n_eltos] = x;
68    ++n_eltos;
69 }

71 // Constructor de copia
72 template <typename T>
73 Pila<T>::Pila(const Pila& P) :
74    elementos(new T[P.Lmax]),
75    Lmax(P.Lmax),
76    n_eltos(P.n_eltos)
77 {
78    for (size_t i = 0; i < n_eltos; ++i)    // Copiar elementos
79       elementos[i] = P.elementos[i];
80 }
```

Implementación genérica vectorial pseudoestática (pilavec.h)

```
82 // Asignación entre pilas
83 template <typename T>
84 Pila<T>& Pila<T>::operator =(const Pila& P)
85 {
86    if (this != &P) {    // Evitar autoasignación
87       // Destruir el vector y crear uno nuevo si es necesario
88       if (Lmax != P.Lmax) {
89          delete[] elementos;
90          Lmax = P.Lmax;
91          elementos = new T[Lmax];
92       }
93       n_eltos = P.n_eltos;
94       for (size_t i = 0; i < n_eltos; ++i)   // Copiar elementos
95          elementos[i] = P.elementos[i];
96    }
97    return *this;
98 }
```

Implementación genérica vectorial pseudoestática (pilavec.h)

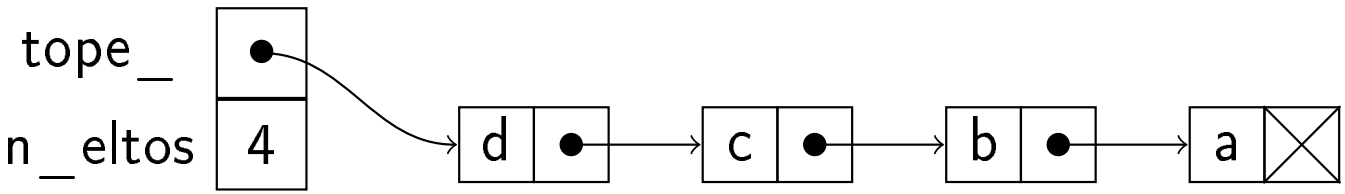
```
100 // Destructor
101 template <typename T>
102 inline Pila<T>::~Pila()
103 {
104    delete[] elementos;
105 }

107 #endif // PILA_VEC_H
```

Implementación genérica mediante celdas enlazadas

Implementación mediante celdas enlazadas

Estructura de datos dinámica de tamaño ajustado en tiempo de ejecución al contenido de la pila. La capacidad no está limitada a priori.



Implementación genérica mediante celdas enlazadas (pilanela.h)

```
1  #ifndef PILA_ENLA_H
2  #define PILA_ENLA_H
3  #include <cstddef>    // size_t
4  #include <cassert>

6  template <typename T> class Pila {
7  public:
8     Pila();
9     bool vacia() const;
10    size_t tama() const;
11    const T& tope() const;
12    void pop();
13    void push(const T& x);
14    Pila(const Pila& P);                // Ctor. de copia
15    Pila& operator =(const Pila& P); // Asignación entre pilas
16    ~Pila();                            // Destructor
```

Implementación genérica mediante celdas enlazadas (pilanela.h)

```
17 private:
18     struct nodo {
19         T elto;
20         nodo* sig;
21         nodo(const T& e, nodo* p = nullptr) : elto(e), sig(p) {}
22     };

24     nodo* tope_;
25     size_t n_eltos;

27     void copiar(const Pila& P);
28 };
```

Implementación genérica mediante celdas enlazadas (pilanela.h)

```
30 template <typename T>
31 inline Pila<T>::Pila() :
32    tope_(nullptr),
33    n_eltos(0)
34 {}

36 template <typename T>
37 inline bool Pila<T>::vacia() const
38 {
39    return (n_eltos == 0);
40 }

42 template <typename T>
43 inline size_t Pila<T>::tama() const
44 {
45    return n_eltos;
46 }

48 template <typename T>
49 inline const T& Pila<T>::tope() const
50 {
51    assert(!vacia());
52    return tope_->elto;
53 }
```

Implementación genérica mediante celdas enlazadas (pilanela.h)

```
55 template <typename T>
56 inline void Pila<T>::pop()
57 {
58    assert(!vacia());
59    nodo* p = tope_;
60    tope_ = p->sig;
61    delete p;
62    --n_eltos;
63 }

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

Implementación genérica mediante celdas enlazadas (pilanela.h)

```
72 // Constructor de copia
73 template <typename T>
74 inline Pila<T>::Pila(const Pila& P) :
75    tope_(nullptr),
76    n_eltos(0)
77 {
78    copiar(P);
79 }

81 // Asignación entre pilas
82 template <typename T>
83 Pila<T>& Pila<T>::operator =(const Pila& P)
84 {
85    if (this != &P) {    // Evitar autoasignación
86       this->~Pila();    // Vaciar la pila actual
87       copiar(P);
88    }
89    return *this;
90 }
```

Implementación genérica mediante celdas enlazadas (pilanela.h)

```
92  // Destructor: vacía la pila
93  template <typename T>
94  Pila<T>::~Pila()
95  {
96     nodo* p;
97     while (tope_) {
98        p = tope_->sig;
99        delete tope_;
100       tope_ = p;
101    }
102    n_eltos = 0;
103 }
```

Implementación genérica mediante celdas enlazadas (pilanela.h)

```
105 // Método privado
106 template <typename T>
107 void Pila<T>::copiar(const Pila& P)
108 // Pre: *this está vacía.
109 // Post: *this es copia de P.
110 {
111    if (!P.vacia()) {
112       tope_ = new nodo(P.tope()); // Copiar el primer elemento.
113       // Copiar el resto de elementos hasta el fondo de la pila.
114       nodo* p = tope_;          // p recorre la pila destino (*this).
115       nodo* q = P.tope_->sig;   // q recorre la pila origen (P) desde 2º nodo.
116       while (q) {
117          p->sig = new nodo(q->elto);
118          p = p->sig;
119          q = q->sig;
120       }
121       n_eltos = P.n_eltos;
122    }
123 }

125 #endif // PILA_ENLA_H
```

Implementación vectorial vs. celdas enlazadas

- Ambas implementaciones son igual de eficientes en tiempo. Las operaciones *pop()* y *push()*, así como *vacia()*, *tama()* y *tope()*, son elementales, $\Theta(1)$.
- Al usar la implementación vectorial hay que predecir la capacidad de la pila, lo cual conlleva un problema de desbordamiento si la capacidad estimada es escasa o desaprovechamiento de memoria si es excesiva.
- *Problema de desbordamiento*: si la pila está llena, es imposible apilar nuevos elementos; a no ser que asumamos el coste en tiempo de copiar el contenido en una pila de mayor capacidad.
- La implementación mediante celdas enlazadas no presenta el problema de desbordamiento, pero a cambio requiere espacio extra para los enlaces, un puntero adicional por cada elemento, lo cual puede suponer un coste excesivo, especialmente si el tamaño de los elementos es pequeño.