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

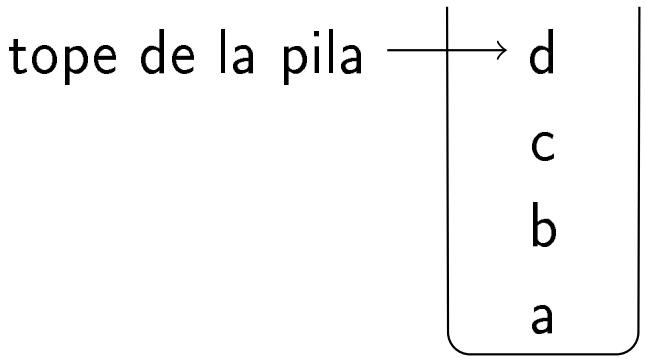
UCA
Universidad de Cádiz

Versión 1.0

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.

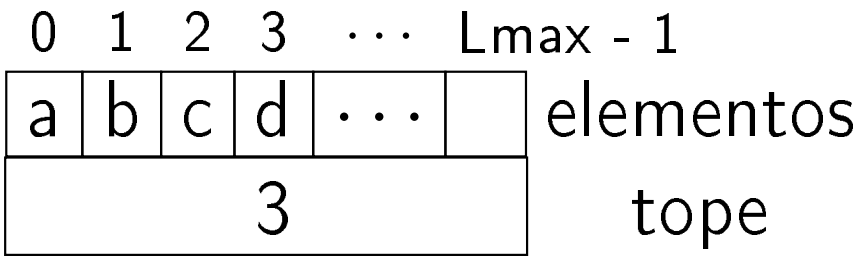
Postcondiciones: Elimina el elemento del tope de la pila y el siguiente se convierte en el nuevo tope.

`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 _tElemento_
2  #define _tElemento_
3     typedef int tElemento; // por ejemplo
4  #endif // _tElemento_
5  #ifndef PILA_VEC0_H
6  #define PILA_VEC0_H
7     const int LMAX = 100; // Longitud máxima de una pila
8     class Pila {
9     public:
10       Pila();
11       bool vacia() const;
12       bool llena() const; // Requerida por la implementación
13       const tElemento& tope() const;
14       void pop();
15       void push(const tElemento& x);
16    private:
17       tElemento elementos[LMAX]; // vector de elementos
18       int tope_; // posición del tope
19    };
20 #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 - 2);
15 }
```

Implementación vectorial estática (pilavec0.cpp)

```
17 const 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(tope_ < LMAX - 1);
32    ++tope_;
33    elementos[tope_] = x;
34 }
```

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>
6  class Pila {
7  public:
8     explicit Pila(unsigned TamaMax); // ctor., requiere ctor.
          tElemento()
9     Pila(const Pila& p); // ctor. de copia
10    Pila& operator =(const Pila& p); // asignación entre pilas
11    bool vacia() const;
12    bool llena() const; // Requerida por la implementación
13    const tElemento& tope() const;
14    void pop();
15    void push(const tElemento& x);
16    ~Pila(); // destructor
```

Implementación genérica vectorial pseudoestática

```
17 private:
18     tElemento *elementos; // vector de elementos
19     int Lmax; // tamaño del vector
20     int tope_; // posición del tope
21 };
```

Implementación genérica vectorial pseudoestática

```
23 template <typename tElemento>
24 inline Pila<tElemento>::Pila(unsigned TamaMax) :
25    elementos(new tElemento[TamaMax]),
26    Lmax(TamaMax),
27    tope_(-1)
28 {}

30 template <typename tElemento>
31 Pila<tElemento>::Pila(const Pila<tElemento>& p) :
32    elementos(new tElemento[p.Lmax]),
33    Lmax(p.Lmax),
34    tope_(p.tope_)
35 {
36    for (int i = 0; i <= tope_; i++) // copiar el vector
37       elementos[i] = p.elementos[i];
38 }
```

Implementación genérica vectorial pseudoestática

```
40 template <typename tElemento>
41 Pila<tElemento>& Pila<tElemento>::operator =(const Pila<
       tElemento>& p)
42 {
43    if (this != &p) { // evitar autoasignación
44       // Destruir el vector y crear uno nuevo si es necesario
45       if (Lmax != p.Lmax) {
46          delete[] elementos;
47          Lmax = p.Lmax;
48          elementos = new tElemento[Lmax];
49       }
50       // Copiar el vector
51       tope_ = p.tope_;
52       for (int i = 0; i <= tope_; i++)
53          elementos[i] = p.elementos[i];
54    }
55    return *this;
56 }
```

Implementación genérica vectorial pseudoestática

```
58 template <typename tElemento>
59 inline bool Pila<tElemento>::vacia() const
60 {
61    return (tope_ == -1);
62 }

64 template <typename tElemento>
65 inline bool Pila<tElemento>::llena() const
66 {
67    return (tope_ > Lmax - 2);
68 }

70 template <typename tElemento>
71 inline const tElemento& Pila<tElemento>::tope() const
72 {
73    assert(!vacia());
74    return elementos[tope_];
75 }
```

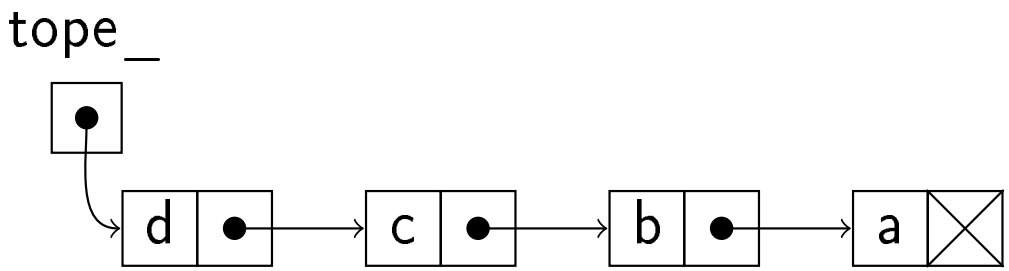
Implementación genérica vectorial pseudoestática

```
77 template <typename tElemento>
78 inline void Pila<tElemento>::pop()
79 {
80    assert(!vacia());
81    --tope_;
82 }

84 template <typename tElemento>
85 inline void Pila<tElemento>::push(const tElemento& x)
86 {
87    assert(!llena());
88    ++tope_;
89    elementos[tope_] = x;
90 }

92 template <typename tElemento>
93 inline Pila<tElemento>::~Pila()
94 {
95    delete[] elementos;
96 }
```

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 = 0): 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_(0) {}

31 template <typename T>
32 Pila<T>::Pila(const Pila<T>& p) : tope_(0)
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* q = tope_;
63    tope_ = q->sig;
64    delete q;
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* q;
78    while (tope_) {
79       q = tope_->sig;
80       delete tope_;
81       tope_ = q;
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* q = tope_; // recorre la pila destino
93        nodo* r = p.tope_->sig; // recorre la pila origen
94        while (r) {
95           q->sig = new nodo(r->elto);
96           q = q->sig;
97           r = r->sig;
98        }
99     }
100 }

102 #endif // PILA_ENLA_H
```