Análisis de Algoritmos y Estructuras de Datos

Tema 6: Tipo Abstracto de Datos Cola

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

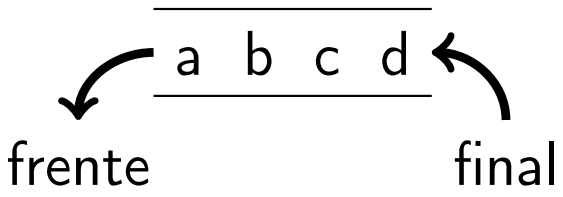
UCA
Universidad de Cádiz

Versión 1.0

Índice

Definición de Cola

- Una cola es una secuencia de elementos en la que las operaciones se realizan por los extremos:
  - Las eliminaciones se realizan por el extremo llamado inicio, frente o principio de la cola.
  - Los nuevos elementos son añadidos por el otro extremo, llamado fondo o final de la cola.
- En una cola el primer elemento añadido es el primero en salir de ella, por lo que también se les conoce como estructuras FIFO: *First Input First Output*.



Especificación del TAD *Cola*

Definición:
Una cola es una secuencia de elementos de un tipo determinado, en la cual se pueden añadir elementos sólo por un extremo, al que llamaremos fin, y eliminar por el otro, al que llamaremos inicio.

Operaciones:

cola()

Postcondiciones: Crea una cola vacía.

bool vacia() const

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

const tElemento& frente() const

Precondiciones: La cola no está vacía.

Postcondiciones: Devuelve el elemento del inicio de la cola.

Especificación del TAD *Cola*

void pop()

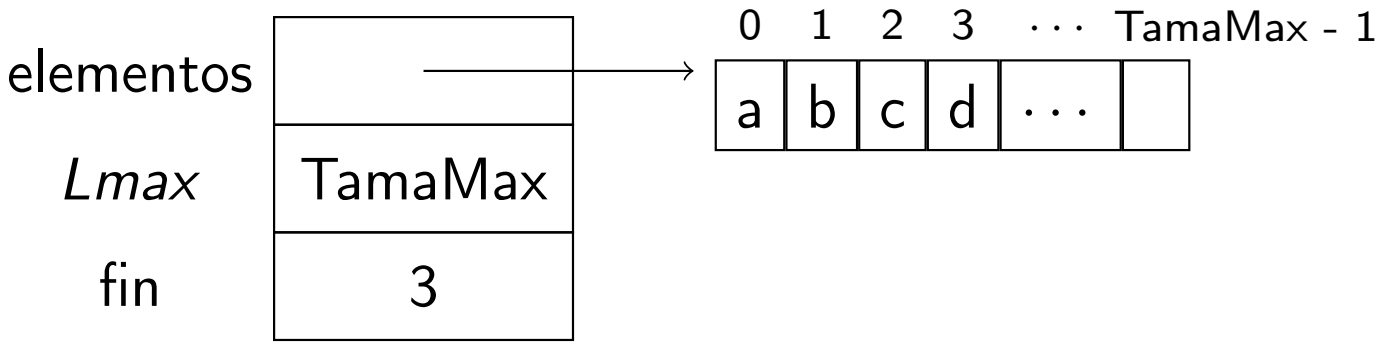
Precondiciones: La cola no está vacía.

Postcondiciones: Elimina el elemento del inicio de la cola y el siguiente se convierte en el nuevo inicio.

void push(const tElemento& x)

Postcondiciones: Inserta el elemento x al final de la cola.

Implementación vectorial pseudoestática



Implementación vectorial pseudoestática

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

5  template <typename T> class Cola {
6  public:
7     explicit Cola(size_t TamaMax); // constructor, requiere ctor. T()
8     Cola(const Cola<T>& C); // ctor. de copia, requiere ctor. T()
9     Cola<T>& operator =(const Cola<T>& C); // asig. colas, req. T()
10    bool vacia() const;
11    bool llena() const; // Requerida por la implementación
12    const T& frente() const;
13    void pop();
14    void push(const T& x);
15    ~Cola(); // destructor
16 private:
17    T *elementos; // vector de elementos
18    int Lmax; // tamaño del vector
19    int fin; // posición del último
20 };
```

Implementación vectorial pseudoestática

```
22 template <typename T>
23 inline Cola<T>::Cola(size_t TamaMax) :
24    elementos(new T[TamaMax]),
25    Lmax(TamaMax),
26    fin(-1)
27 {}

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

Implementación vectorial pseudoestática

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

Implementación vectorial pseudoestática

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

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

69 template <typename T>
70 inline const T& Cola<T>::frente() const
71 {
72    assert(!vacia());
73    return elementos[0];
74 }
```

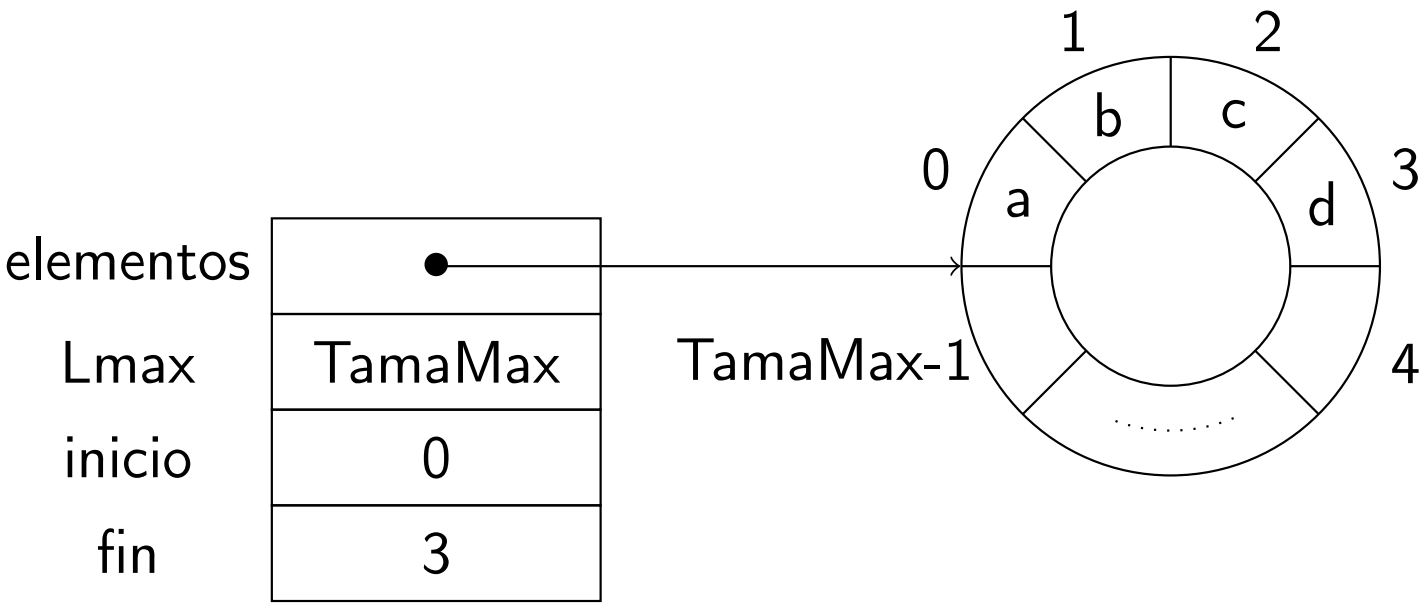
Implementación vectorial pseudoestática

```
76 template <typename T>
77 void Cola<T>::pop()
78 {
79    assert(!vacia());
80    for (int i = 0; i < fin; i++)
81       elementos[i] = elementos[i+1];
82    fin--;
83 }

85 template <typename T>
86 inline void Cola<T>::push(const T& x)
87 {
88    assert(!llena());
89    fin++;
90    elementos[fin] = x;
91 }
```

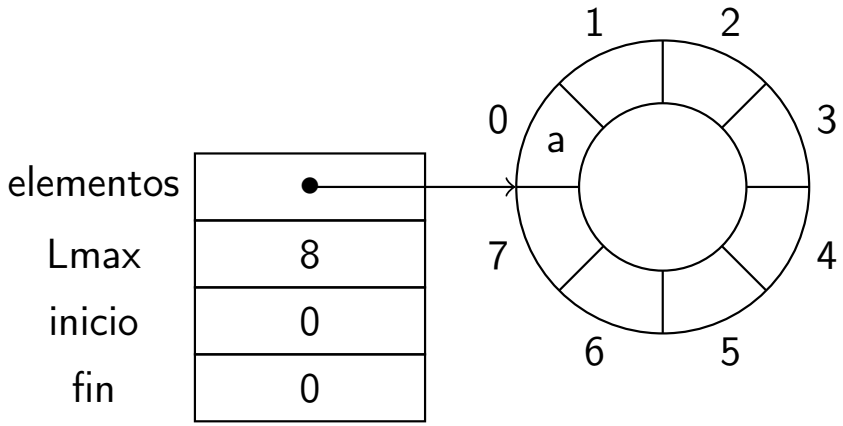
Implementación vectorial pseudoestática

```
93 template <typename T>
94 inline Cola<T>::~Cola()
95 {
96    delete[] elementos;
97 }

99 #endif // COLA_VEC_H
```

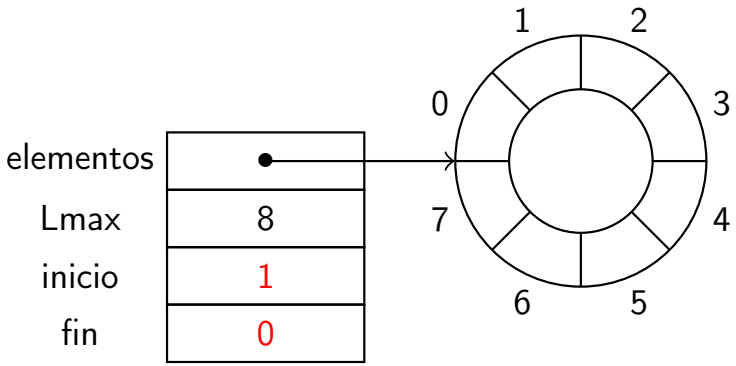
Implementación vectorial circular: esquema general



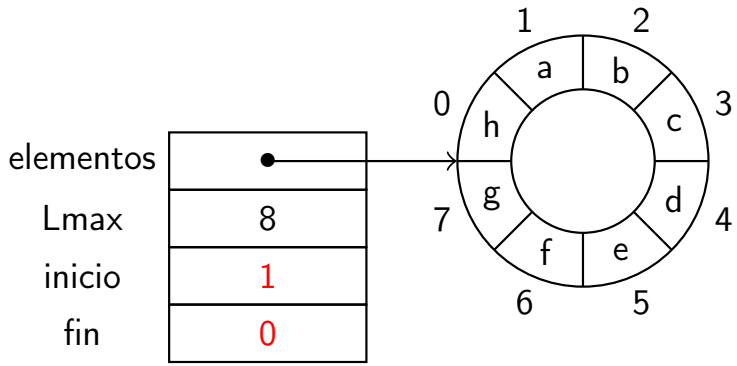
Implementación vectorial circular: casos



(a) Un elemento



(b) Vacía

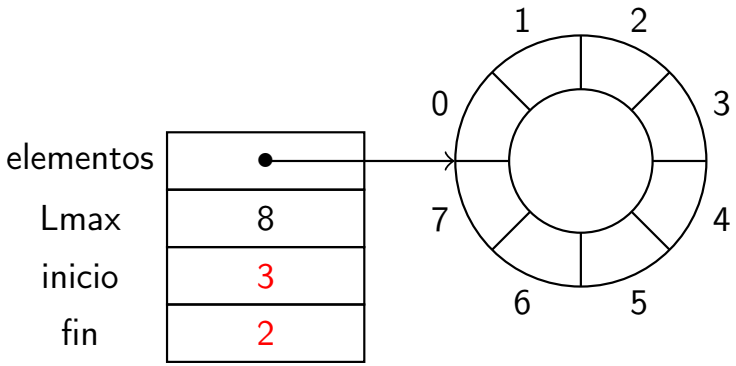


(c) Llena

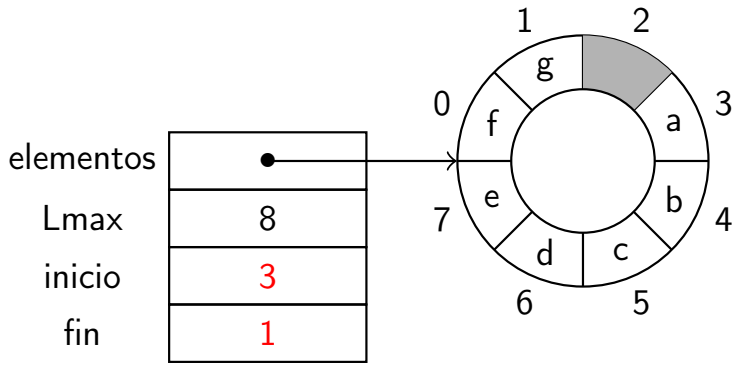
Implementación vectorial circular

Distinción cola vacía/llena

1. Atributo adicional de la clase cola:
   - Indicador de tipo `bool`, vacía/no-vacía o llena/no-llena
   - Contador de elementos
2. Conservar al menos una posición libre en el vector



(d) Vacía



(e) Llena

Implementación vectorial circular: código

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

5  template <typename T> class Cola {
6  public:
7     explicit Cola(size_t TamaMax); // constructor, requiere ctor. T()
8     Cola(const Cola<T>& C); // ctor. de copia, requiere ctor. T()
9     Cola<T>& operator =(const Cola<T>& C); // asig. colas, req. T()
10    bool vacia() const;
11    bool llena() const; // Requerida por la implementación
12    const T& frente() const;
13    void pop();
14    void push(const T& x);
15    ~Cola(); // destructor
16 private:
17    T *elementos; // vector de elementos
18    int Lmax; // tamaño del vector
19    int inicio, fin; // posiciones de los extremos de la cola
20 };
```

Implementación vectorial circular: código

```
22 template <typename T>
23 Cola<T>::Cola(size_t TamaMax) :
24    elementos(new T[TamaMax + 1]), // +1 para detectar cola llena
25    Lmax(TamaMax + 1),
26    inicio(0),
27    fin(TamaMax)
28 {}

30 template <typename T>
31 Cola<T>::Cola(const Cola<T>& C) :
32    elementos(new T[C.Lmax]),
33    Lmax(C.Lmax),
34    inicio(C.inicio),
35    fin(C.fin)
36 {
37    if (!C.vacia()) // Copiar el vector
38       for (int i = inicio; i != (fin + 1) % Lmax;
39            i = (i + 1) % Lmax)
40          elementos[i] = C.elementos[i];
41 }
```

Implementación vectorial circular: código

```
43 template <typename T>
44 Cola<T>& Cola<T>::operator =(const Cola<T>& C)
45 {
46    if (this != &C) { // evitar autoasignación
47       // Destruir el vector y crear uno nuevo si es necesario
48       if (Lmax != C.Lmax) {
49          delete[] elementos;
50          Lmax = C.Lmax;
51          elementos = new T[Lmax];
52       }
53       // Copiar el vector
54       inicio = C.inicio;
55       fin = C.fin;
56       if (!C.vacia())
57          for (int i = inicio; i != (fin + 1) % Lmax;
58               i = (i + 1) % Lmax)
59             elementos[i] = C.elementos[i];
60    }
61    return *this;
62 }
```

Implementación vectorial circular: código

```
64 template <typename T>
65 inline bool Cola<T>::vacia() const
66 {
67    return ((fin + 1) % Lmax == inicio);
68 }

70 template <typename T>
71 inline bool Cola<T>::llena() const
72 {
73    return ((fin + 2) % Lmax == inicio);
74 }

76 template <typename T>
77 inline const T& Cola<T>::frente() const
78 {
79    assert(!vacia());
80    return elementos[inicio];
81 }
```

Implementación vectorial circular: código

```
83 template <typename T>
84 inline void Cola<T>::pop()
85 {
86    assert(!vacia());
87    inicio = (inicio + 1) % Lmax;
88 }

90 template <typename T>
91 inline void Cola<T>::push(const T& x)
92 {
93    assert(!llena());
94    fin = (fin + 1) % Lmax;
95    elementos[fin] = x;
96 }
```

Implementación vectorial circular: código

```
 98 template <typename T>
 99 inline Cola<T>::~Cola()
100 {
101    delete[] elementos;
102 }

104 #endif // COLA_CIR_H
```