Estructuras de datos: Pilas, Colas, Listas

Algoritmos

Facultad de Informática
Universidad de A Coruña

UNIVERSIDADE DA CORUÑA

Table of Contents

Table of Contents

Pilas

- Acceso limitado al último elemento insertado
- Operaciones básicas: `apilar`, `desapilar` y `cima`.
  - `desapilar` o `cima` en una pila vacía es un error en el TDA pila.
  - Quedarse sin espacio al `apilar` es un error de implementación.
- Cada operación debería tardar una cantidad constante de tiempo en ejecutarse.
  - Con independencia del número de elementos apiladas.

Implementación a base de vectores (i)

```
tipo Pila = registro
 Cima_de_pila : 0..Tamaño_máximo_de_pila
 Vector_de_pila : vector [1..Tamaño_máximo_de_pila]
                          de Tipo_de_elemento
fin registro

procedimiento Crear Pila ( P )
 P.Cima_de_pila := 0
fin procedimiento

función Pila Vacía ( P ) : test
  devolver P.Cima_de_pila = 0
fin función
```

Implementación a base de vectores (ii)

```
procedimiento Apilar ( x, P )
  si P.Cima_de_pila = Tamaño_máximo_de_pila entonces
    error Pila llena
  sino
    P.Cima_de_pila := P.Cima_de_pila + 1;
    P.Vector_de_pila[P.Cima_de_pila] := x
fin procedimiento
función Cima ( P ) : Tipo_de_elemento
  si Pila Vacía (P) entonces error Pila vacía
  sino devolver P.Vector_de_pila[P.Cima de Pila]
fin función
procedimiento Desapilar ( P )
  si Pila Vacía (P) entonces error Pila vacía
  sino P.Cima_de_pila := P.Cima_de_pila - 1
fin procedimiento
```

Código JAVA: Interfaz Pila

```
// OPERACIONES PÚBLICAS
//  void apilar(x) ->Inserta x
//  void desapilar() ->Elimina el último elemento insertado
//  Object cima() ->Devuelve el último elemento insertado
//  boolean esVacia() ->Devuelve true si vacía, sino false
//  void vaciar() ->Elimina todos los elementos
// ERRORES: cima o desapilar sobre la pila vacía
public interface IPila {
    void apilar(Object x);
    void desapilar() throws Exception;
    Object cima() throws Exception;
    boolean esVacia();
    void vaciar();
}
```

Código JAVA: Clase PilaVec (i)

```
import java.util.*;
public class PilaVec implements IPila {
    private Vector p;
    private int cimaDePila;
    static final int CAPACIDAD_POR_DEFECTO = 10;
    public PilaVec() {
        p = new Vector(CAPACIDAD_POR_DEFECTO);
        cimaDePila = -1;
    }
    public boolean esVacia() {
        return (cimaDePila == -1);
    }
    public void vaciar() {
        cimaDePila = -1;
    }
```

Código JAVA: Clase `PilaVec` (ii)

```
    public void apilar(Object x) {
        if (++cimaDePila == p.size()) p.add(x);
        else p.set(cimaDePila, x);
    }
    public void desapilar() throws Exception {
        if (esVacia()) throw new Exception("pila vacía");
        cimaDePila--;
    }
    public Object cima() throws Exception {
        if (esVacia()) throw new Exception("pila vacía");
        return p.get(cimaDePila);
    }
}
```

Table of Contents

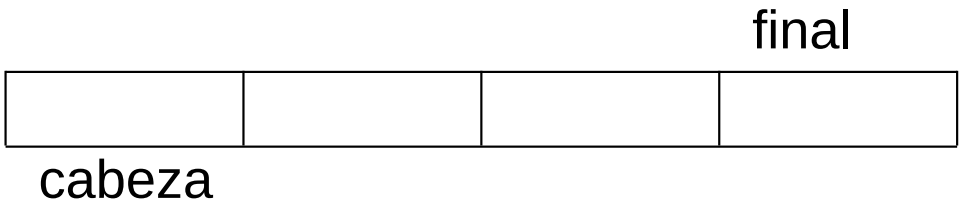
Colas

- Operaciones básicas: `insertar`, `quitarPrimero` y `primero`.
- Cada rutina debería ejecutarse en tiempo constante.

Implementación circular a base de vectores (i)

- La implementación circular devuelve cabeza y fin al principo del vector cuando rebasan la última posición.

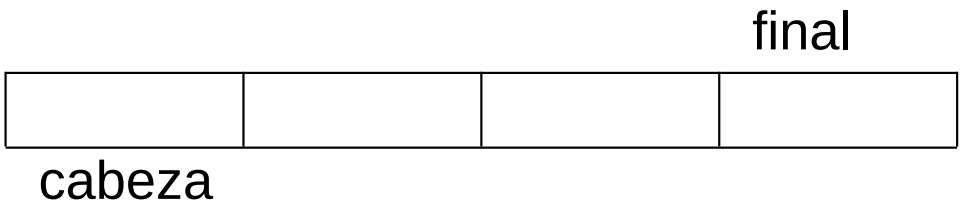
1) Crear_Cola (C)



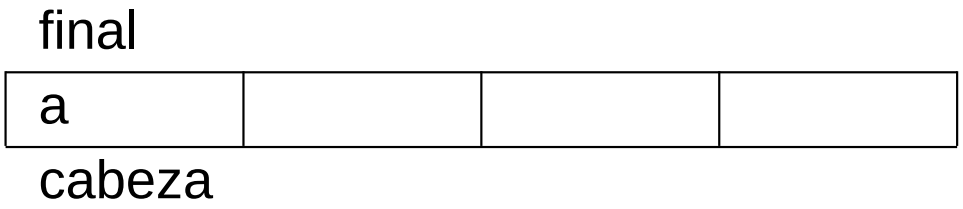
Implementación circular a base de vectores (i)

- La implementación circular devuelve cabeza y fin al principo del vector cuando rebasan la última posición.

1) Crear_Cola (C)



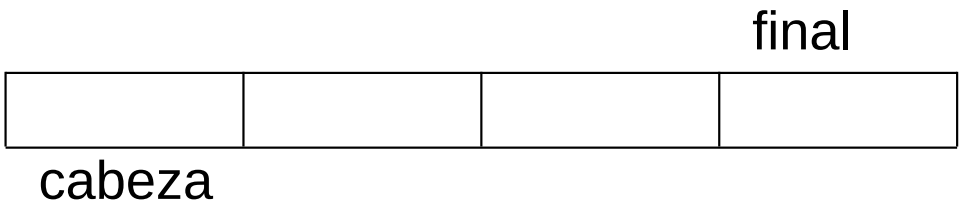
2) Insertar_en_Cola (a,C)



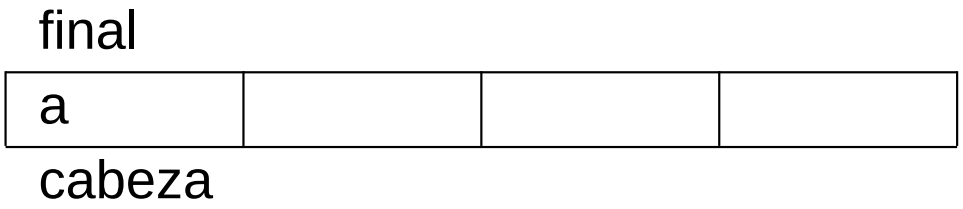
Implementación circular a base de vectores (i)

- La implementación circular devuelve cabeza y fin al principo del vector cuando rebasan la última posición.

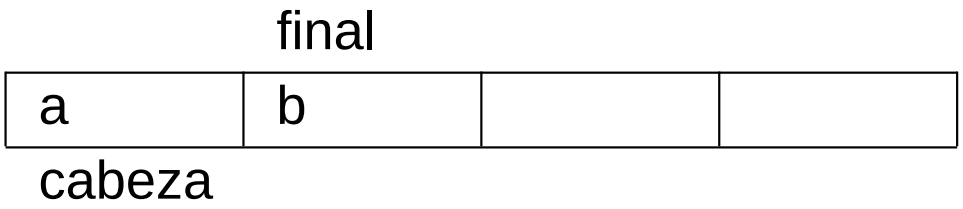
1) Crear_Cola (C)



2) Insertar_en_Cola (a,C)

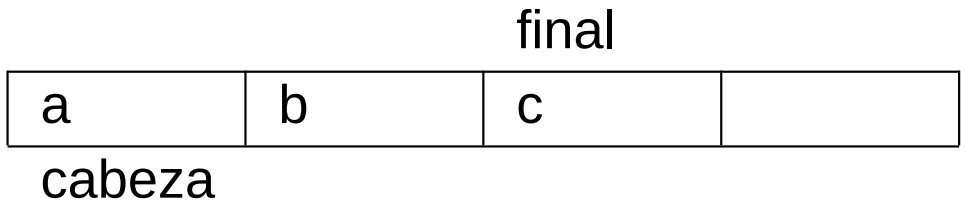


3) Insertar_en_Cola (b,C)



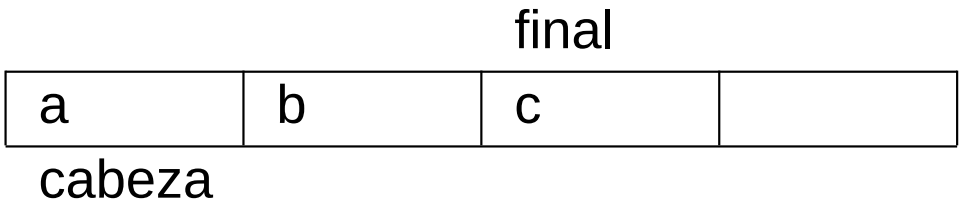
Implementación circular a base de vectores (i)

4) Insertar_en_Cola (c,C)

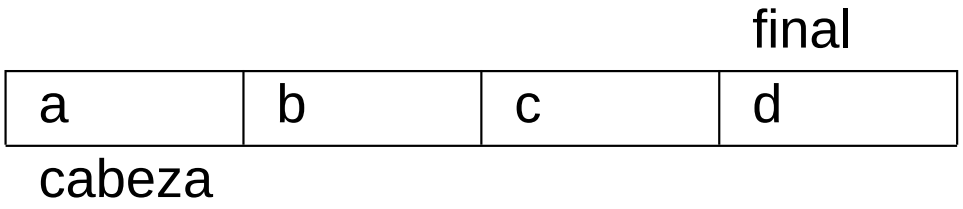


Implementación circular a base de vectores (i)

4) Insertar_en_Cola (c,C)

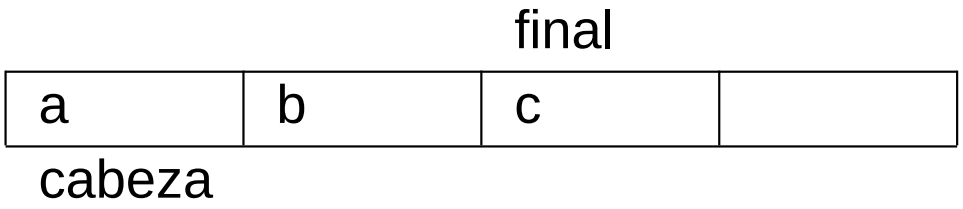


5) Insertar_en_Cola (d,C)

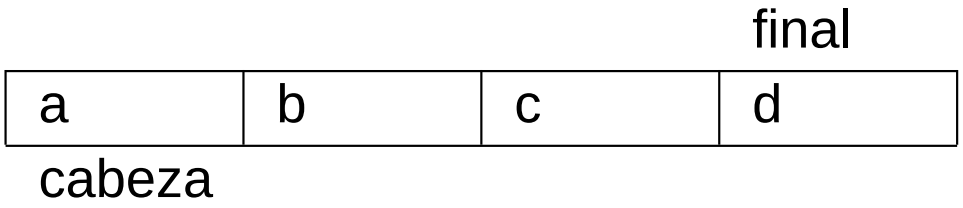


Implementación circular a base de vectores (i)

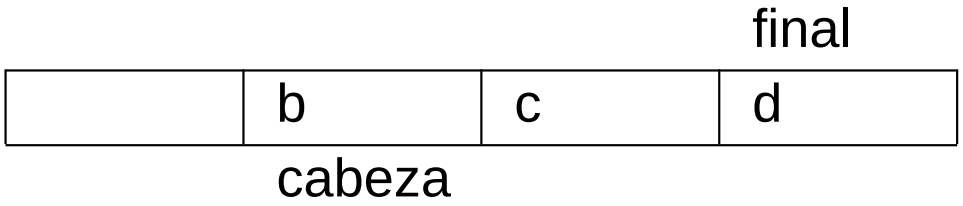
4) Insertar_en_Cola (c,C)



5) Insertar_en_Cola (d,C)

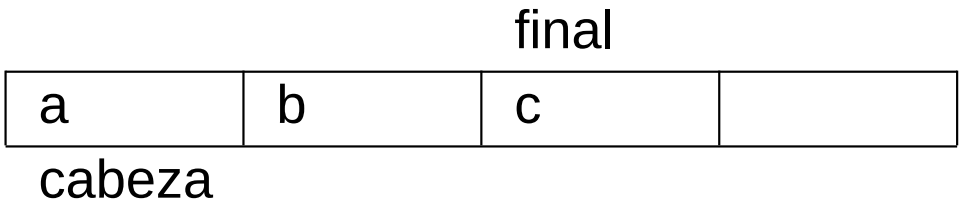


6) Quitar_Primero (C)

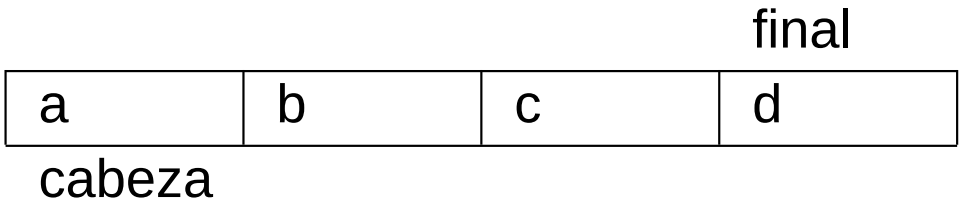


Implementación circular a base de vectores (i)

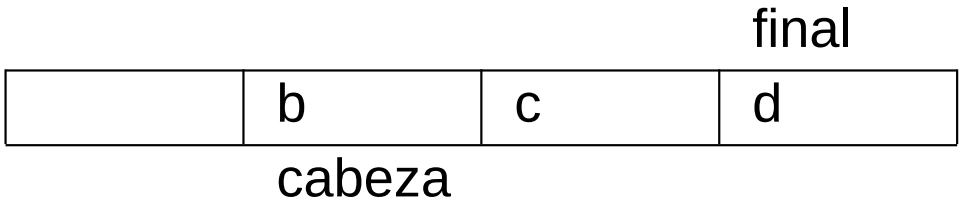
4) Insertar_en_Cola (c,C)



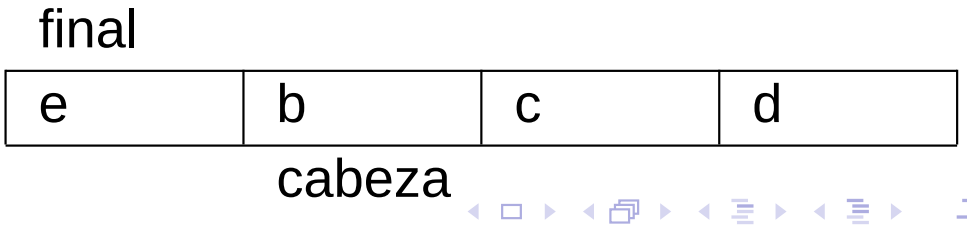
5) Insertar_en_Cola (d,C)



6) Quitar_Primero (C)



7) Insertar_en_Cola (e,C)



Pseudocódigo (i)

```
tipo Cola = registro
  Cabeza_de_cola, Final_de_cola: 1..Tamaño_máximo_de_cola
  Tamaño_de_cola : 0..Tamaño_máximo_de_cola
  Vector_de_cola : vector [1..Tamaño_máximo_de_cola]
                          de Tipo_de_elemento
fin registro
procedimiento Crear_Cola ( C )
  C.Tamaño_de_cola := 0;
  C.Cabeza_de_cola := 1;
  C.Final_de_cola := Tamaño_máximo_de_cola
fin procedimiento
función Cola_Vacía ( C ) : test
  devolver C.Tamaño_de_cola = 0
fin función
```

Pseudocódigo (ii)

```
procedimiento incrementar ( x ) (* privado *)
  si x = Tamaño_máximo_de_cola entonces x := 1
  sino x := x + 1
fin procedimiento

procedimiento Insertar_en_Cola ( x, C )
  si C.Tamaño_de_Cola = Tamaño_máximo_de_cola entonces
    error Cola llena
  sino
    C.Tamaño_de_cola := C.Tamaño_de_cola + 1;
    incrementar(C.Final_de_cola);
    C.Vector_de_cola[C.Final_de_cola] := x;
fin procedimiento
```

Pseudocódigo (iii)

```
función Quitar_Primero ( C ) : Tipo_de_elemento
  si Cola_Vacía ( C ) entonces
    error Cola vacía
  sino
    C.Tamaño_de_cola := C.Tamaño_de_cola - 1;
    x := C.Vector_de_cola[C.Cabeza_de_cola];
    incrementar(C.Cabeza_de_cola);
    devolver x
fin función
función Primero ( C ) : Tipo_de_elemento
  si Cola_Vacía ( C ) entonces
    error Cola vacía
  sino
    devolver Vector_de_cola[C.Cabeza_de_cola]
fin función
```

Código JAVA: Interfaz Cola

```
// OPERACIONES PÚBLICAS
//  void insertar(x) -> Inserta x
//  Object primero() -> Devuelve el primer elemento
//  Object quitarPrimero() -> Devuelve y elimina el primer e
//  boolean esVacia() -> Devuelve true si vacía, si no false
//  void vaciar() -> Elimina todos los elementos
// ERRORES: primer y quitarPrimero sobre una cola vacía
public interface ICola {
    void insertar(Object x);
    Object primero() throws Exception;
    Object quitarPrimero() throws Exception;
    boolean esVacia();
    void vaciar();
}
```

Código JAVA: Clase ColasVec (i)

```
public class ColaVec implements ICola {
    private Object [] vector;
    private int tamanoActual;
    private int cabezaDeCola;
    private int finalDeCola;
    static final int CAPACIDAD_POR_DEFECTO = 10;
    public ColaVec() {
        vector = new Object[CAPACIDAD_POR_DEFECTO];
        vaciar();
    }
    public void vaciar() {
        tamanoActual = 0;
        cabezaDeCola = 0;
        finalDeCola = -1;
    }
```

Código JAVA: Clase ColaVec (ii)

```
public boolean esVacia() {
    return (tamanoActual == 0);
}

public Object primero() throws Exception {
    if (esVacia()) throw new Exception("cola vacía");
    return vector[cabezaDeCola];
}

private int incrementar(int x) {
    if (++x == vector.length)
        x = 0;
    return x;
}
```

Código JAVA: Clase ColaVec (iii)

```
public Object quitarPrimero() throws Exception {
    if (esVacia())
        throw new Exception("cola vacía");
    tamanoActual--;
    Object valorDevuelto = vector[cabezaDeCola];
    cabezaDeCola = incrementar(cabezaDeCola);
    return valorDevuelto;
}
public void insertar(Object x) {
    if (tamanoActual == vector.length)
        duplicarCola();
    finalDeCola = incrementar(finalDeCola);
    vector[finalDeCola] = x;
    tamanoActual++;
}
```

Código JAVA: Clase ColaVec (iv)

```
    private void duplicarCola() {
        Object [] nuevoVector =
            new Object[2*vector.length];
        for (int i=0; i<tamanoActual; i++,
                cabezaDeCola = incrementar(cabezaDeCola))
            nuevoVector[i] = vector[cabezaDeCola];
        vector = nuevoVector;
        cabezaDeCola = 0;
        finalDeCola = tamanoActual - 1;
    }
}
```

Amortización de la duplicación del vector

- Cuando el vector no se duplica, toda operación se realiza en tiempo constante.
  - La complejidad de una inserción con duplicación es $O(N)$.
- La duplicación de un vector de N elementos está precedida, al menos, por $N/2$ inserciones que no duplican el vector.
- Repartiendo el coste $O(N)$ de la duplicación entre las inserciones precedentes,
  - el coste de `insertar` aumenta sólo en una constante.

Table of Contents

Listas

- Operaciones básicas:
  - `Visualizar` su contenido.
  - `Buscar` la posición de la primera ocurrencia de un elemento.
  - `Insertar` y `Eliminar` un elemento en alguna posición.
  - `Buscar_k_esimo`, que devuelve el elemento de la posición indicada

Implementación de listas a base de vectores

- Tiene que declararse el tamaño de la lista.
  - Exige sobrevaloración.
  - Consume mucho espacio.
- Complejidad computacional de las operaciones:
  - `Buscar_k_esimo`, tiempo constante
  - `Visualizar` y `Buscar`, tiempo lineal.
  - `Insertar` y `Eliminar` son costosas.
    - Insertar o eliminar un elemento exige, en promedio, desplazar la mitad de los valores, $O(n)$.
    - La construcción de una lista o la eliminación de todos sus elementos podría exigir un tiempo cuadrático.

Implementación de listas a base de apuntadores

- Cada nodo apunta al siguiente; el último no apunta a nada.
- La lista es un puntero al primer nodo (y al último).
- Complejidad computacional de las operaciones:
  - `Visualizar` y `Buscar`, tiempo lineal.
  - `Buscar_k_esimo`, tiempo lineal.
  - `Eliminar` realiza un cambio de apuntadores y una orden `dispose`, $O(1)$.
    - Usa `Buscar_anterior` cuyo tiempo de ejecución es lineal.
  - `Insertar` tras una posición `p` require una llamada a `new` y dos maniobras con apuntadores, $O(1)$.
    - Buscar la posición `p` podría llevar tiempo lineal.
  - Un nodo cabecera facilita la inserción y la eliminación al comienzo de la lista.

Implementación de listas doblemente enlazadas

- Cada nodo apunta al siguiente y al anterior.
- Duplica el uso de la memoria necesaria para los punteros.
- Duplica el coste de manejo de punteros al insertar y eliminar.
- La eliminación se simplifica.
  - No es necesario buscar el elemento anterior.

Implementación con un nodo cabecera (i)

```
tipo PNodo = puntero a Nodo
     Lista = PNodo
     Posición = PNodo
     Nodo = registro
       Elemento : Tipo_de_elemento
       Siguiente : PNodo
     fin registro
procedimiento Crear Lista ( L )
  nuevo ( tmp );
  si tmp = nil  entonces error Memoria agotada
  sino
     tmp^.Elemento := { nodo cabecera };
     tmp^.Siguiente := nil;
     L := tmp
fin procedimiento
```

Implementación con un nodo cabecera (ii)

```
función Lista Vacía ( L ) : test
  devolver Lˆ.Siguiente = nil
fin función

función Buscar ( x, L ) : posición de la 1ª ocurrencia
                          o nil
  p := Lˆ.Siguiente;
  mientras p <> nil y pˆ.Elemento <> x hacer
    p := pˆ.Siguiente;
  devolver p
fin función

función Último Elemento ( p ) : test { privada }
  devolver pˆ.Siguiente = nil
fin función
```

Implementación con un nodo cabecera (iii)

```
función Buscar Anterior ( x, L ) : posición anterior a x
                                   o a nil { privada }
  p := L;
  mientras pˆ.Siguiente <> nil y
                    pˆ.Siguienteˆ.Elemento <> x hacer
    p := pˆ.Siguiente;
  devolver p
fin función
procedimiento Eliminar ( x, L )
  p := Buscar Anterior ( x, L );
  si Último Elemento ( p ) entonces error No encontrado
  sino tmp := pˆ.Siguiente;
       pˆ.Siguiente := tmpˆ.Siguiente;
       liberar ( tmp )
fin procedimiento
```

Implementación con un nodo cabecera (iv)

```
procedimiento Insertar ( x, L, p )
  nuevo ( tmp );   { Inserta después de la posición p }
  si tmp = nil  entonces
    error Memoria agotada
  sino
    tmp^.Elemento := x;
    tmp^.Siguiente := p^.Siguiente:
    p^.Siguiente := tmp
fin procedimiento
```