

Clase 23

Introducción a las estructuras de datos: pilas y colas

1

Algoritmos

Un *algoritmo* es una definición exacta, aunque no necesariamente formal, de cómo resolver una clase de problema computacional.

Tomemos como ejemplo el problema de clasificar una lista de objetos ordenados.

2

Ordenación por inserción

crear una lista temporal vacía, *tempList*
para cada elemento, x , de la lista *sortList*
para cada elemento a de *tempList*
 si $x \leq a$,
 insertar x en *tempList* antes de a
 continuar con el siguiente x
añadir x a *tempList*
cambiar *tempList* por *sortList*

3

Hipótesis de ordenación

- Los elementos de la lista se pueden comparar y ordenar
- La lista tiene su propio orden, un inicio y un fin
- Se pueden realizar iteraciones sobre la lista (actuar sobre cada uno de los elementos en orden)
- Se pueden insertar nuevos elementos en la lista

4

Abstracciones aplicables a la ordenación

- No es necesario conocer lo que se clasifica, basta con saber que se puede ordenar
- Tampoco hace falta conocer el modo en el que se ha implementado la lista
- En lenguaje Java, estas dos series de asunciones se pueden considerar como dos interfaces, `Comparable` y `List`

5

Estructuras de datos

- La interfaz `List` define una estructura de datos.
- Las estructuras de datos y los algoritmos suelen guardar entre sí una relación simbiótica:
 - Las interfaces `List` sirven para muchos algoritmos
 - Muchos algoritmos se pueden aplicar a diversas estructuras de datos (p. ej., también se pueden ordenar los datos de una matriz)

6

Tipos de datos abstractos y concretos

La “estructura de datos” combina en realidad dos conceptos de orientación a objetos:

- la *interfaz* de la estructura de datos, que define *tipos de datos abstractos (ADT)*, y
- una implementación específica de esta interfaz que da lugar a *tipos de datos concretos (CDT)*.

7

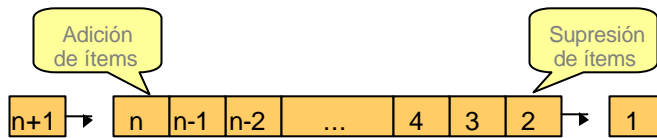
Tipos de datos concretos

- Los algoritmos se definen con tipos de datos abstractos, pero para poder ejecutar un programa se necesitan versiones concretas de esos tipos
- Las diferentes implementaciones (CDTs) de un tipo determinado de datos abstractos tendrán mayor o menor eficacia dependiendo de la aplicación.
- El concepto de eficacia puede variar según la importancia que se dé al tiempo y al espacio

8

Colas

Se entiende por *cola* una estructura de datos en la que se añaden nuevos ítems en un extremo y se suprimen ítems viejos en el opuesto.



9

Operaciones con colas

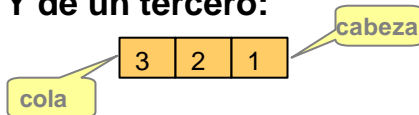
Adición de un ítem:



Adición de otro ítem:



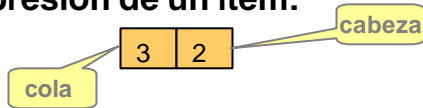
Y de un tercero:



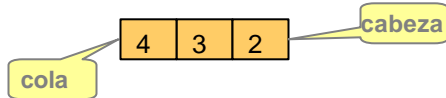
10

Operaciones con colas (2)

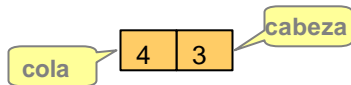
Supresión de un ítem:



Adición de otro ítem:



Supresión de un tercero:



11

Interfaz Queue

```
public interface Queue
{
    public boolean isEmpty();
    public void add( Object o );
    public Object remove() throws
        NoSuchElementException;
    public void clear();
}
```

12

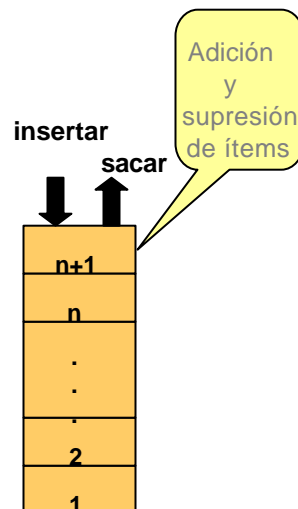
Aplicaciones de las colas

- Las colas se utilizan en muchos algoritmos y en situaciones en las que el rendimiento de dos sistemas que se cruzan datos entre sí es más eficiente cuando no se intercambian indicativos y señales de control (*handshaking*) en cada transferencia.
- También almacenan temporalmente la transferencia de información, lo que permite procesarla en origen y en destino a tasas independientes.
- La cola de eventos en Java es un buen ejemplo.
- Tipos derivados: colas de prioridad y flujos de datos

13

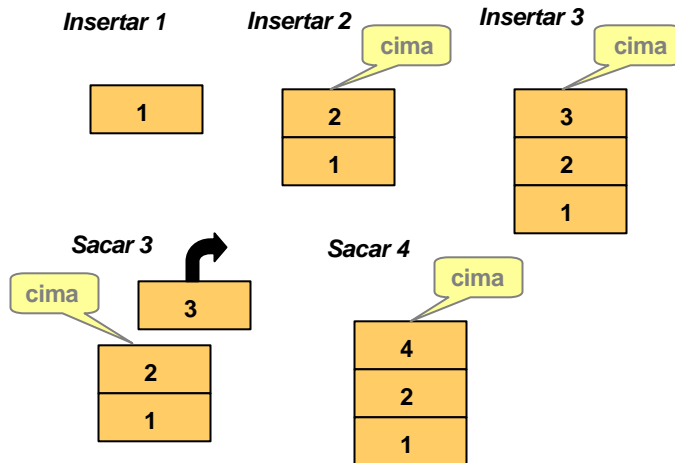
Pilas

- A diferencia de las colas, en las pilas los ítems se añaden y se eliminan en el mismo extremo.
- Las pilas se conocen también como *colas LIFO* (último en entrar, primero en salir), para diferenciarlas de las *colas FIFO* (primero en entrar, primero en salir) al describir el tipo de cola básico.



14

Operaciones con pilas



15

Aplicaciones de las pilas

- Proporcionan un medio ordenado de demorar la realización de las tareas secundarias que aparecen durante la ejecución del programa.
- Suelen ir asociadas a algoritmos recursivos.
- Tipos derivados: pilas de programas, pila del analizador sintáctico (*parser*).

16

Interfaz Stack

```
public interface Stack
{
    public boolean isEmpty();
    public void push( Object o );
    public Object pop() throws
        EmptyStackException;
    public void clear();
}
```

17

Inversión mediante pila del orden de un *array*

```
public class Reverse {
    public static void main(String args[] ) {
        int [] array = { 1, 2, 3, 4, 5 };
        int i;
        Stack stack = new ArrayStack();

        for ( i = 0; i < array.length; i++ )
            stack.push( new Integer( array[ i ] ) );
        i = 0;
        while ( !stack.isEmpty() ) {
            array[ i ] =
                ((Integer) stack.pop()).intValue();
            System.out.println( array[ i++ ] );
        }
    }
}
```

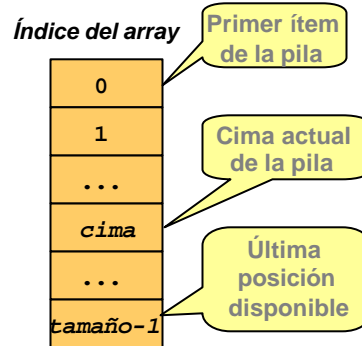
18

Implementación de pilas basada en *arrays*

En esta implementación basada en *arrays*, la posición en el *array* del elemento cima de la pila va bajando a medida que se insertan ítems, y subiendo a medida que se sacan.

El inicio de la pila se hallará siempre en el elemento 0 del *array*.

En cierto sentido, la pila se va formando "cabeza abajo" en el *array*.



19

ArrayStack, 1

```
public class ArrayStack implements Stack {
    final static int DEFAULT_CAPACITY = 16;

    private Object [] stack;
    private int     top   = -1;
    private int     capacity;

    public ArrayStack( int cap ) {
        capacity = cap;
        stack = new Object[ cap ];
    }

    public ArrayStack() {
        this( DEFAULT_CAPACITY );
    }
}
```

20

Array Stack, 2

```
public boolean isEmpty() {
    return ( top < 0 );
}

public void clear() {
    for ( int i = 0; i < top; i++ )
        stack[ i ] = null; // para recogida de basura
    top = -1;
}
```

21

Array Stack, 3

```
public Object pop()
    throws EmptyStackException
{
    if ( isEmpty() )
        throw new EmptyStackException();
    else {
        Object ret = stack[ top ];
        stack[ top-- ] = null; //para recogida de basura
        return ret;
    }
}
```

22

Array Stack, 4

```
public void push(Object o) {
    if ( top == capacity-1 )
        grow();
    stack[ ++top ] = o;
}

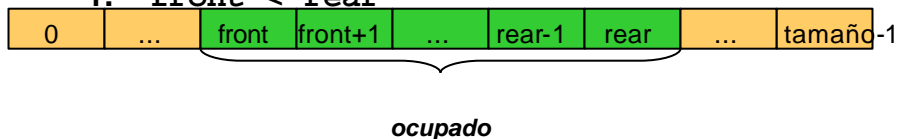
private void grow() {
    Object [] old = stack;
    int oldCapacity = capacity;
    capacity *= 2;
    stack = new Object[ capacity ];
    System.arraycopy( old, 0, stack, 0, oldCapacity );
}
```

23

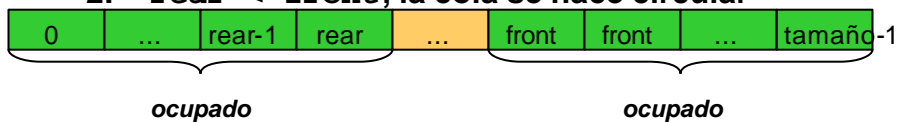
Implementación de colas basada en *arrays*

Dos supuestos:

1. $front < rear$



2. $rear < front$, la cola se hace circular



24

Array Queue, 2

```
public boolean isEmpty() { return ( size == 0 ); }

public void clear() {
    size = 0; front = 0; rear = capacity - 1;
    for ( int i = 0; i < capacity; i++ )
        queue[ i ] = null; // para recogida de basura
}

public void add(Object o) {
    if ( size == capacity ) grow();
    rear = ( rear + 1 ) % capacity;
    queue[ rear ] = o;
    size++;
}
```

27

Array Queue, 3

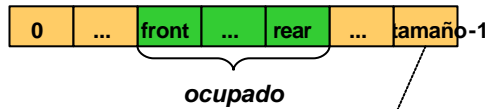
```
public Object remove() throws NoSuchElementException {
    if ( isEmpty() )
        throw new NoSuchElementException();
    else {
        Object ret = queue[ front ];
        queue[ front ] = null; // para recogida de basura
        front = (front + 1) % capacity;
        size--;
        return ret;
    }
}
```

28

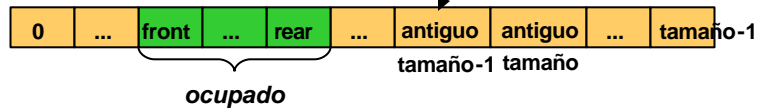
Aumento del tamaño de un *ArrayQueue*

Supuesto básico:

$front < rear$, antes



$front < rear$, después



29

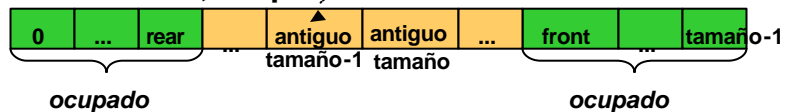
Aumento del tamaño de un *ArrayQueue*

Supuesto más complicado:

$rear < front$, antes



$rear < front$, después



30

Array Queue, 4

```
private void grow() {
    Object[] old= queue;
    int oldCapacity= capacity; capacity *= 2;
    queue= new Object[capacity];

    if ( size == 0 ) return;
    else if ( front <= rear ) {
        System.arraycopy(old, front, queue, front, size );
    } else if ( rear < front ) {
        System.arraycopy(old, 0, queue, 0, rear + 1);
        System.arraycopy(old, front, queue,
            front + oldCapacity, oldCapacity - front );
        front += oldCapacity;
    }
}
```

31