

Estructuras de datos

Clase teórica 6



Contenido

- Colas
- Pilas

Material elaborado por: Julián Moreno

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

¿Qué es eso de colas y pilas?

Expectativa

Realidad

“Trabajemos con pilas”



“Trabajemos con colas”



Colas y pilas

Las colas y las pilas son estructuras de datos ampliamente utilizadas en computación. Por ejemplo la gestión del procesador se hace, en términos generales, mediante una cola, mientras que la gestión de la memoria se hace mediante una pila.

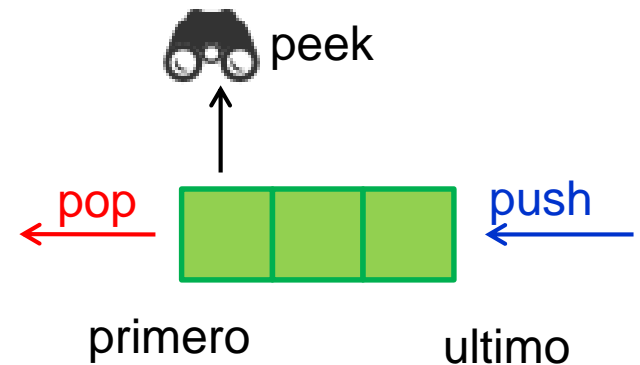
Dependiendo de la forma como se implementen tendrán más o menos métodos, pero sus tres principales son:

- **push**: Inserta un nuevo elemento
- **pop**: Extrae el elemento próximo a salir
- **peek**: “Espía” el elemento próximo a salir

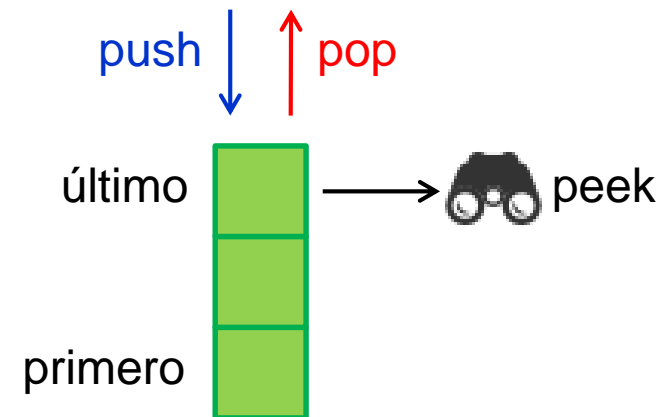
En ambos casos el método push inserta el nuevo elemento después del último que se haya insertado.

Colas y pilas

En la cola, al extraer se obtiene el elemento que se haya insertado hace más tiempo. Por esta razón también se conocen como estructuras de datos FIFO (First In First Out: primero en entrar, primero en salir).

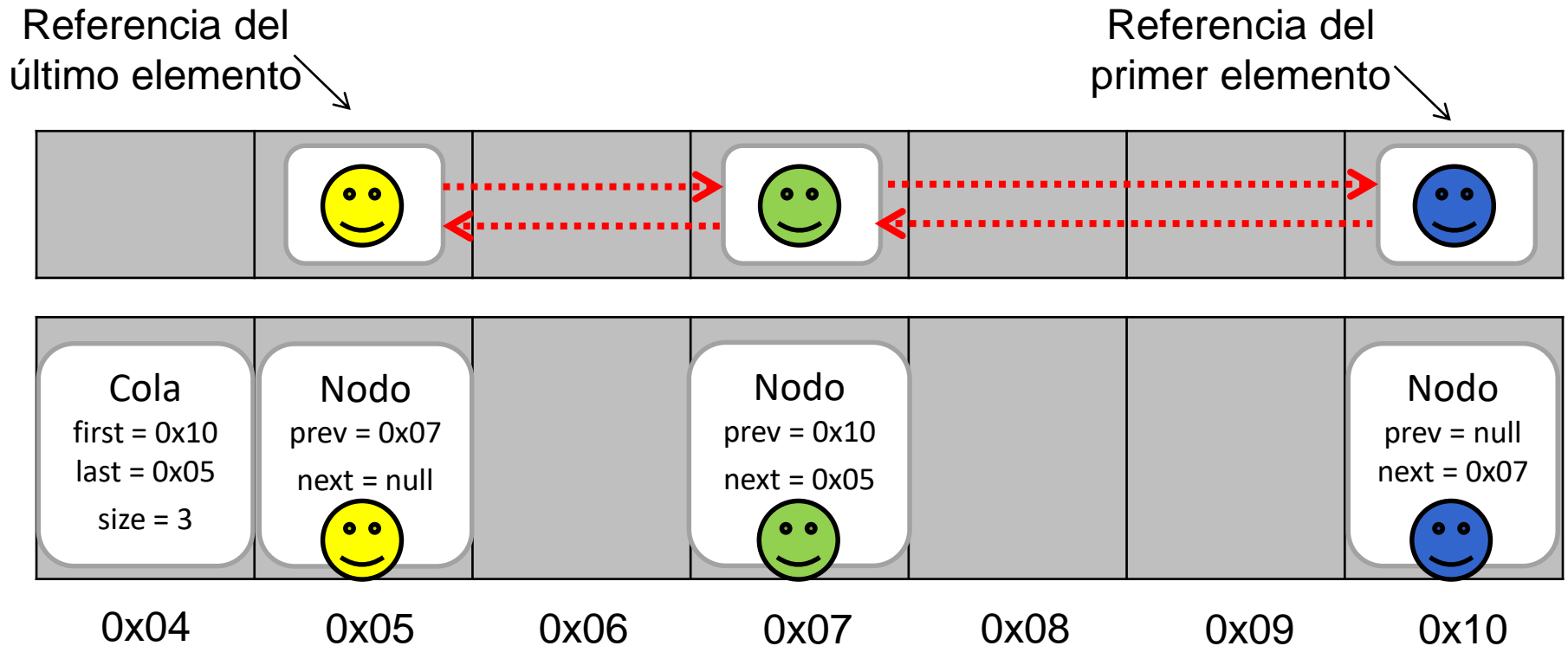


En la pila, al extraer se obtiene el último elemento que acaba de insertarse. Por esta razón también se conocen como estructuras de datos LIFO (Last In First Out: último en entrar, primero en salir).



Implementación de una cola

Las operaciones que discutimos previamente (push, pop, peek) determinan qué debe hacer la estructura, más no el cómo. De lo que hemos visto, una cola se podría implementar tanto mediante un arreglo dinámico, como con una lista enlazada, sin embargo, en procura de la eficiencia, lo común es la segunda alternativa con doble enlace y con dos referencias, una para el primero, y otra para el último elemento.



Inserción en una cola

Comencemos analizando la operación de inserción. Si nos referimos al *push*, dicha inserción (*push*) se realiza al final de la lista lo cual, como ya vimos, tiene una eficiencia $O(1)$:

```
// e es el elemento a ingresar
p = new nodo
p.elm = e
if (cola.size > 0){
    (cola.last).next = p
    p.prev = cola.last
}
else
    cola.first = p
cola.last = p
cola.size++
```

Borrado en una cola

Analicemos ahora la operación de borrado. Si nos referimos al *pop*, dicho borrado + devolución se realiza al inicio de la lista lo cual, como ya vimos, tiene una eficiencia $O(1)$:

```
if (cola.size > 0){
    p = (cola.first).elm
    cola.first = (cola.first).next
    if (cola.size == 1)
        cola.last = NULL
    cola.size--
    return p
}
else
    return ERROR
```

Indexado en una cola

Si lo analizamos, hacer un peek corresponde a hacer un indexado sobre el índice 0 de la lista lo cual, como ya vimos, tiene una eficiencia $O(1)$:

```
if (cola.size > 0){  
    p = (cola.first).elm  
    return p  
}  
else  
    return ERROR
```


Búsqueda en una cola

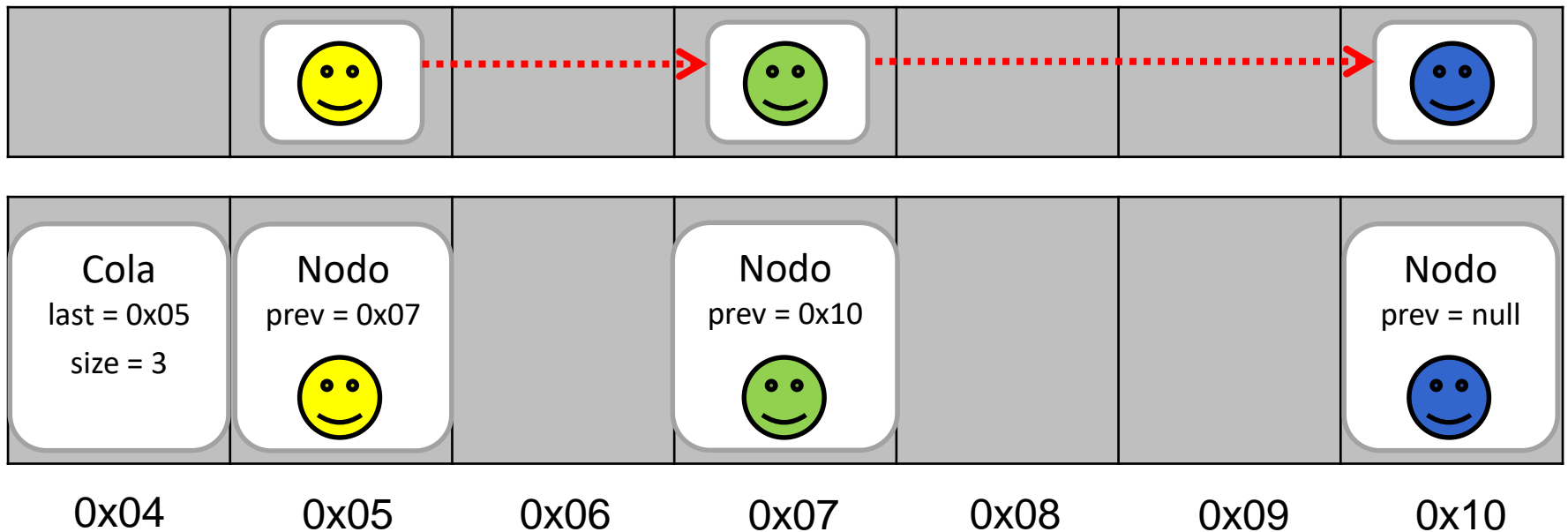
La búsqueda en una cola no suele ser una operación común. Sin embargo, en caso de ser necesaria, esta operación resulta ser casi idéntica que en una lista enlazada (como es doblemente enlazada se puede hacer tanto desde el primero hacia el último como desde el último hacia el primero) y por tanto tiene la misma eficiencia $O(n)$:

```
// e es el elemento buscado
p = cola.first
i = 0
while (i < cola.size && p.elm != e) {
    p = p.next
    i++
}
if (p.elm == e)
    return i
else
    return -1
```

Implementación de una pila

Una pila se puede implementar casi igual que una cola. Sin embargo, considerando que tanto la inserción (*push*) como el borrado (*pop*) se realizan por el mismo extremo de la lista, solo necesitamos una referencia al elemento de ese extremo y un enlace simple entre los nodos:

Referencia del
último elemento ↘



Inserción en una pila

La inserción (*push*) se realiza al final de la lista lo cual, al igual que en la cola, tiene una eficiencia $O(1)$, aunque tener un solo enlace lo hace de hecho es más simple:

```
// e es el elemento a ingresar
p = new nodo
p.elm = e
if (pila.size > 0)
    p.prev = pila.last
pila.last = p
pila.size++
```

Borrado en una pila

Igual que en la cola, esta operación (refiriéndonos al *pop* = borrado + devolución) se puede hacer en $O(1)$:

```
if (pila.size > 0){  
    p = (pila.last).elm  
    pila.last = (pila.last).prev  
    pila.size--  
    return p  
}  
else  
    return ERROR
```

Indexado en una pila

Al igual que en las colas, el indexado que nos interesa es el peek, el cual de nuevo tiene una eficiencia $O(1)$:

```
if (pila.size > 0) {  
    p = (pila.last).elm  
    return p  
}  
else  
    return ERROR
```

Búsqueda en una pila

Resulta similar al de una cola solo que la búsqueda lineal solo se puede realizar en un sentido: desde el último hacia el primero, pero al fin y al cabo con la misma eficiencia $O(n)$:

```
// e es el elemento buscado
p = pila.last
i = 0
while (i < pila.size && p.elm != e) {
    p = p.prev
    i++
}
if (p.elm == e)
    return i
else
    return -1
```

Ejercicio

Para poner a prueba parte de lo aprendido, hagamos una prueba de escritorio al siguiente pseudocódigo, ¿qué mostraría?

```
cola c
c.push(10)
c.push(25)
c.push(18)
print(c.pop())
c.push(34)
print(c.peek())
c.push(27)
print(c.pop())
print(c.peek())
print(c.pop())
```

Ejercicio

Hagamos algo parecido pero ahora con una pila, ¿qué mostraría?

```
pila p
p.push(10)
p.push(25)
p.push(18)
print(p.pop())
p.push(34)
print(p.peek())
p.push(27)
print(p.pop())
print(p.peek())
print(p.pop())
```


Ejercicio

Finalmente, miremos este pseudocódigo que emplea ambas estructuras, ¿qué mostraría?

```
cola c
pila p
c.push(50)
c.push(70)
p.push(60)
p.push(40)
print(c.peak())
print(p.pop())
p.push(c.pop())
c.push(p.peak()-10)
print(p.peak())
print(c.pop())
```

Problema típico: Torres de Hanoi

En una antigua ciudad en la India, los monjes en un templo tienen que mover una pila de 64 discos sagrados de un lugar a otro.

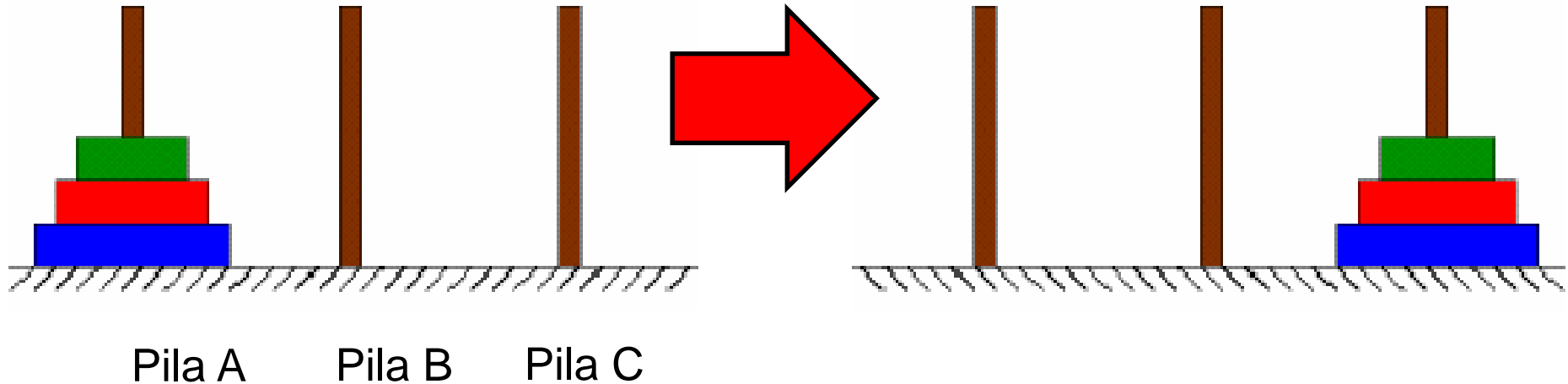
Los discos son frágiles, sólo pueden ser cargados de uno en uno, y un disco no debe nunca ser colocado arriba de otro más pequeño.

Además, solamente hay un lugar en el templo (aparte del lugar original y el lugar destino) suficientemente sagrado para poner una pila de discos allí y servir así de apoyo en el traslado de discos desde el origen hasta el destino.

La leyenda dice además, que antes de que los monjes realicen el último movimiento para completar la torre en su nuevo lugar, el templo se reducirá a cenizas y el mundo se acabará.

Quizás esta leyenda tenga razón debido a la enorme cantidad de movimientos necesarios para cambiar de lugar los 64 discos: como mínimo $2^{64}-1 \approx 1,84 \times 10^{19}$ movimientos, que se traduce en aproximadamente 585 mil millones de años si un movimiento durara un segundo (se estima que el universo tiene unos 14 mil millones de años de antigüedad)

Problema típico: Torres de Hanoi



```
C.push(A.pop())  
B.push(A.pop())  
B.push(C.pop())  
C.push(A.pop())  
A.push(B.pop())  
C.push(B.pop())  
C.push(A.pop())
```

Problema típico: Torres de Hanoi

```
import java.util.*;
public class Main{

    public static void main(String[] args){
        Scanner entrada = new Scanner(System.in);
        int n = entrada.nextInt();
        Hanoi(n, 'A', 'B', 'C');
    }

    public static void Hanoi(int n, char origen, char auxiliar, char destino){
        if (n == 1){
            System.out.println(destino + ".push(" + origen + ".pop())");
        }
        else{
            Hanoi(n - 1, origen, destino, auxiliar);
            System.out.println(destino + ".push(" + origen + ".pop())");
            Hanoi(n - 1, auxiliar, origen, destino);
        }
    }
}
```

Cuya eficiencia es $O(2^n)$

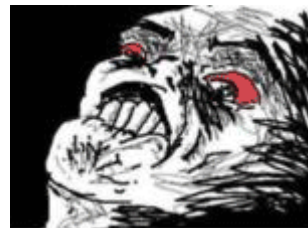


Tabla resumen

Recapitulando la clase de hoy tenemos que:

Estructura	Inserción	Indexación	Búsqueda	Borrado
Cola	$O(1)$ refiriéndonos al push	$O(1)$ refiriéndonos al peek	$O(n)$	$O(1)$ refiriéndonos al pop
Pila	$O(1)$ refiriéndonos al push	$O(1)$ refiriéndonos al peek	$O(n)$	$O(1)$ refiriéndonos al pop