Unidades IV y V - Pilas y Colas

- Pilas que "Recuerdan"
- Priorizar con Colas

- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

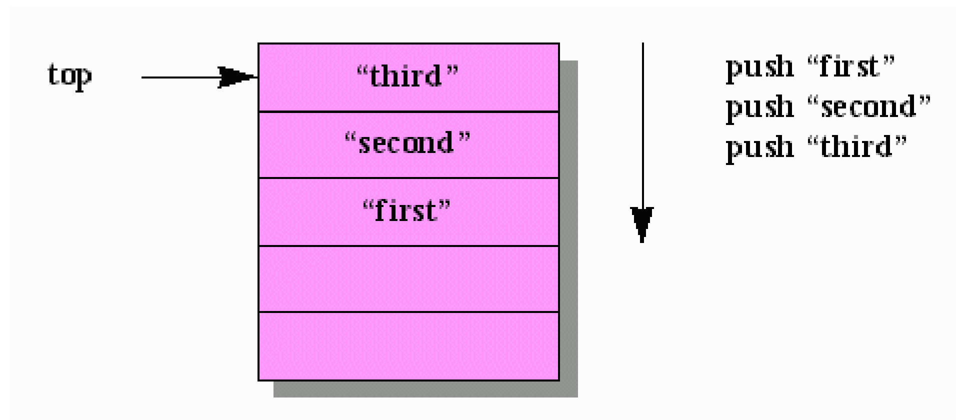
Pilas y Colas

Los desarrolladores utilizan los arrays y las variantes de listas enlazadas para construir una gran variedad de estructuras de datos complejas. Esta página explora dos de esas estructuras: las Pilas, las Colas. Cuando presentemos los algoritmos lo haremos únicamente en código Java por motivos de brevedad.

Pilas que "Recuerdan"

La *Pila* es una estructura de datos donde las inserciones y recuperaciones/borrados de datos se hacen en uno de los finales, que es conocido como el *top* de la pila. Como el último elemento insertado es el primero en recuperarse/borrarse, los desarrolladores se refieren a estas pilas como pilas LIFO (*last-in, first-out*).

Los datos se *push* (insertan) dentro y se *pop* (recuperan/borran) de la parte superior de la pila. La siguiente figura ilustra una pila con tres `String` cada uno insertado en la parte superior de la pila:



Como muestra la figura anterior, las pilas se construyen en memoria. Por cada dato insertado, el ítem superior anterior y todos los datos inferiores se mueven hacia abajo. Cuando llega el momento de sacar un ítem de la pila, se recupera y se borra de la pila el ítem superior (que en la figura anterior se revela como `"third"`).

Las pilas son muy útiles en varios escenarios de programación. Dos de los más comunes son:

Pilas que contienen direcciones de retorno:

- Cuando el código llama a un método, la dirección de la primera instrucción que sigue a la llamada se inserta en la parte superior de la pila de llamadas de métodos del thread actual.

Cuando el método llamado ejecuta la instrucción `return`, se saca la dirección de la parte superior de la pila y la ejecución continúa en esa dirección. Si un método llama a otro método, el comportamiento LIFO de la pila asegura que la instrucción `return` del segundo método transfiere la ejecución al primer método, y la del primer método transfiere la ejecución al código que sigue al código que llamó al primer método. Como resultado una pila "recuerda" las direcciones de retorno de los métodos llamados.

Pilas que contienen todos los parámetros del método llamado y las variables locales:

- Cuando se llama a un método, la JVM reserva memoria cerca de la dirección de retorno y almacena todos los parámetros del método llamado y las variables locales de ese método. Si el método es un método de ejemplar, uno de los parámetros que almacena en la pila es la referencia `this` del objeto actual.

Es muy común implementar una pila utilizando un array uni-dimensional o una lista de enlace simple. En el escenario del array uni-dimensional, una variable entera, típicamente llamada `top`, contiene el índice de la parte superior de la pila. De forma similar, una variable de referencia, también nombrada normalmente como `top`, referencia el nodo superior del escenario de la lista de enlace simple.

He modelado mis implementaciones de pilas después de encontrar la arquitectura del API Collections de Java. Mis implementaciones constan de una interfase `Stack` para una máxima flexibilidad, las clases de implementación `ArrayStack` y `LinkedListStack`, y una clase de soporte `FullStackException`. Para facilitar su distribución, he empaquetado estas clases en un paquete llamado `com.javajeff.cds`, donde `cds` viene de *estructura de datos complejas*. El siguiente listado presenta la interfase `Stack`:

```
// Stack.java

package com.javajeff.cds;

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

Sus cuatro métodos determinan si la pila está vacía, recuperan el elemento superior sin borrarlo de la pila, sitúan un elemento en la parte superior de la pila y el último recuerda/borra el elemento superior. Aparte de un constructor específico de la implementación, su programa únicamente necesita llamar a estos métodos.

El siguiente listado presenta una implementación de un `Stack` basado en un array uni-dimensional:

```
// ArrayStack.java

package com.javajeff.cds;

public class ArrayStack implements Stack  {
      private int top = -1;
      private Object [] stack;

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

      public boolean isEmpty () {
             return top == -1;
      }

      public Object peek () {
             if (top < 0)
                    throw new java.util.EmptyStackException ();
             return stack [top];
      }

      public void push (Object o) {
```

```
		if (top == stack.length - 1)
			throw new FullStackException ();
		stack [++top] = o;
	}

	public Object pop () {
		if (top < 0)
			throw new java.util.EmptyStackException ();
		return stack [top--];
	}
}
```

ArrayStack revela una pila como una combinación de un índice entero privado top y variables de referencia de un array uni-dimensional stack. top identifica el elemento superior de la pila y lo inicializa a -1 para indica que la pila está vacía. Cuando se crea un objeto ArrayStack llama a public ArrayStack(int maxElements) con un valor entero que representa el número máximo de elementos. Cualquier intento de sacar un elemento de una pila vacía mediante pop() resulta en el lanzamiento de una java.util.EmptyStackException. De forma similar, cualquier intento de poner más elementos de maxElements dentro de la pila utilizando push(Object o) lanzará una FullStackException, cuyo código aparece en el siguiente listado:

```
// FullStackException.java

package com.javajeff.cds;

public class FullStackException extends RuntimeException {
}
```

Por simetría con EmptyStackException, FullStackException extiende RuntimeException. Como resultado no se necesita añadir FullStackException a la cláusula throws del método.

El siguiente listado presenta una implementación de Stack utilizando una lista de enlace simple:

```
// LinkedListStack.java

package com.javajeff.cds;

public class LinkedListStack implements Stack  {
	private static class Node {
		Object o;
		Node next;
	}

	private Node top = null;

	public boolean isEmpty () {
		return top == null;
	}

	public Object peek () {
		if (top == null)
			throw new java.util.EmptyStackException ();
		return top.o;
	}

	public void push (Object o) {
		Node temp = new Node ();
		temp.o = o;
		temp.next = top;
		top = temp;
	}

	public Object pop () {
		if (top == null)
			throw new java.util.EmptyStackException ();
		Object o = top.o;
		top = top.next;
		return o;
```

```
    }
}
```

LinkedListStack revela una pila como una combinación de una clase anidada privada de alto nivel llamada Node y una variable de referencia privada top que se inicializa a null para indicar una pila vacía. Al contrario que su contrapartida del array uni-dimensional, LinkedListStack no necesita un constructor ya que se expande dinámicamente cuando se ponen los ítems en la pila. Así, void push(Object o) no necesita lanzar una FullStackException. Sin embargo, Object pop() si debe chequear si la pila está vacía, lo que podría resultar en el lanzamiento de una EmptyStackException.

Ahora que ya hemos visto la interfase y las tres clases que generan mis implementaciones de las pilas, juguemos un poco. El siguiente listado muestra casi todo el soporte de pilas de mi paquete com.javajeff.cds:

```
// StackDemo.java

import com.javajeff.cds.*;

class StackDemo {
    public static void main (String [] args)  {
        System.out.println ("ArrayStack Demo");
        System.out.println ("---------------");
        stackDemo (new ArrayStack (5));
        System.out.println ("LinkedListStack Demo");
        System.out.println ("--------------------");
        stackDemo (new LinkedListStack ());
    }

    static void stackDemo (Stack s) {
        System.out.println ("Pushing \"Hello\"");
        s.push ("Hello");

        System.out.println ("Pushing \"World\"");
        s.push ("World");

        System.out.println ("Pushing StackDemo object");
        s.push (new StackDemo ());

        System.out.println ("Pushing Character object");
        s.push (new Character ('C'));

        System.out.println ("Pushing Thread object");
        s.push (new Thread ("A"));

        try {
            System.out.println ("Pushing \"One last item\"");
            s.push ("One last item");
        }
        catch (FullStackException e)  {
            System.out.println ("One push too many");
        }

        System.out.println ();

        while (!s.isEmpty ())
            System.out.println (s.pop ());
        try {
            s.pop ();
        }
        catch (java.util.EmptyStackException e) {
            System.out.println ("One pop too many");
        }
        System.out.println ();
    }
}
```

Cuando se ejecuta StackDemo, produce la siguiente salida:

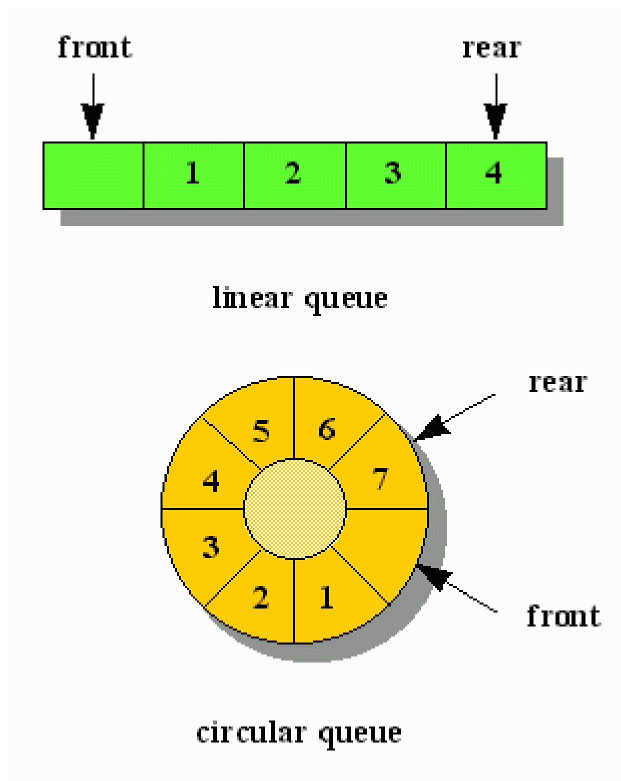
```
ArrayStack Demo
```

```
----------------
Pushing "Hello"
Pushing "World"
Pushing StackDemo object
Pushing Character object
Pushing Thread object
Pushing "One last item"
One push too many

Thread[A,5,main]
C
StackDemo@7182c1
World
Hello
One pop too many

LinkedListStack Demo
--------------------
Pushing "Hello"
Pushing "World"
Pushing StackDemo object
Pushing Character object
Pushing Thread object
Pushing "One last item"

One last item
Thread[A,5,main]
C
StackDemo@cac268
World
Hello
One pop too many
```

Priorizar con Colas

La `Cola` es una estructura de datos donde la inserción de ítem se hace en un final (el fin de la cola) y la recuperación/borrado de elementos se hace en el otro final (el inicio de la cola). Como el primer elemento insertado es el primero en ser recuperado, los desarrolladores se refieren a estas colas como estructuras FIFO (*first-in, first-out*).

Normalmente los desarrolladores trabajan con dos tipos de colas: lineal y circular. En ambas colas, la inserción de datos se realiza en el fin de la cola, se mueven hacia adelante y se recuperan/borran del inicio de la cola. La siguiente figura ilustra las colas lineal y circular:



La cola lineal de la figura anterior almacena cuatro enteros, con el entero 1 en primer lugar. Esa cola está llena y no puede almacenar más datos adicionales porque `rear` identifica la parte final de la cola. La razón de la posición vacía, que identifica `front`, implica el comportamiento lineal de la cola. Inicialmente, `front` y `rear` identifican la posición más a la izquierda, lo que indica que la cola está vacía. Para almacenar el entero 1, `rear` avanza una posición hacia la derecha y almacena 1 en esa posición. Para recuperar/borrar el entero 1, `front` avanza una posición hacia la derecha.

Nota:

Para señalar que la cola lineal está vacía, no necesita gastar una posición, aunque esta aproximación algunas veces es muy conveniente. En su lugar asigne el mismo valor que indique una posición no existente a `front` y a `rear`. Por ejemplo, asumiendo una implementación basada en un array unidimensional, `front` y `rear` podrían contener -1. El índice 0 indica entonces la posición más a la izquierda, y los datos se insertarán empezando en este índice.

Cuando `rear` identifique la posición más a la derecha, la cola lineal podría no estar llena porque `front` podría haber avanzado al menos una posición para recuperar/borrar un dato. En este escenario, considere mover todos los ítems de datos hacia la izquierda y ajuste la posición de `front` y `rear` de la forma apropiada para crear más espacio. Sin embargo, demasiado movimiento de datos puede afectar al rendimiento, por eso debe pensar cuidadosamente en los costes de rendimiento si necesita crear más espacio.

La cola circular de la figura anterior tiene siete datos enteros, con el entero 1 primero. Esta cola está llena y no puede almacenar más datos hasta que `front` avance una posición en sentido horario (para recuperar el entero 1) y `rear` avance una posición en la misma dirección (para identificar la posición que contendrá el nuevo entero). Al igual que con la cola lineal, la razón de la posición vacía, que identifica `front`, implica el comportamiento circular de la cola. Inicialmente, `front` y `rear` identifican la misma posición, lo que indica una cola vacía. Entonces `rear` avanza una posición por cada nueva inserción. De forma similar, `front` avanza una posición por cada recuperación/borrado.

Las colas son muy útiles en varios escenarios de programación, entre los que se encuentran:

Temporización de Threads:

- Una JVM o un sistema operativo subyacente podrían establecer varias colas para coincidir con diferentes prioridades de los threads. La información del thread se bloquea porque todos los threads con una prioridad dada se almacenan en una cola asociada.

Trabajos de impresión:

- Como una impresora normalmente es más lenta que un ordenador, un sistema operativo maneja los trabajos de impresión en un subsistema de impresión, que inserta esos trabajos de impresión en una cola. El primer trabajo en esa cola se imprime primero, y así sucesivamente.

Los desarrolladores normalmente utilizan una array uni-dimensional para implementar una cola. Sin embargo, si tienen que co-existir múltiple colas o las inserciones en las colas deben ocurrir en posiciones distintas a la última por motivos de prioridades, los desarrolladores suelen cambiar a la lista doblemente enlazada. Con un array uni-dimensional dos variables enteras (normalmente llamadas `front` y `rear`) contienen los índices del primer y último elemento de la cola, respectivamente. Mis implementaciones de colas lineales y circulares usan un array uni-dimensional y empiezan con la interfase `Queue` que puede ver en el siguiente listado:

```
// Queue.java

package com.javajeff.cds;

public interface Queue {
      void insert (Object o);
      boolean isEmpty ();
      boolean isFull ();
      Object remove ();
}
```

`Queue` declara cuatro métodos para almacenar un datos, determinar si la cola está vacía, determinar si la cola está llena y recuperar/borrar un dato de la cola. Llame a estos métodos (y a un constructor) para trabajar con cualquier implementación de `Queue`.

El siguiente listado presenta una a implementación de `Queue` de una cola lineal basada en un array uni-dimensional:

```
// ArrayLinearQueue.java

package com.javajeff.cds;

public class ArrayLinearQueue implements Queue {
      private int front = -1, rear = -1;
      private Object [] queue;

      public ArrayLinearQueue (int maxElements) {
             queue = new Object [maxElements];
      }

      public void insert (Object o) {
             if (rear == queue.length - 1)
                     throw new FullQueueException ();
             queue [++rear] = o;
      }

      public boolean isEmpty () {
             return front == rear;
      }

      public boolean isFull () {
             return rear == queue.length - 1;
      }

      public Object remove () {
             if (front == rear)
                     throw new EmptyQueueException ();
             return queue [++front];
      }
```

```
}
```

`ArrayLinearQueue` revela que una cola es una combinación de variables privadas `front`, `rear`, y `queue`. `front` y `rear` se inicializan a -1 para indicar una cola vacía. Igual que el constructor de `ArrayStack` llama a `public ArrayLinearQueue(int maxElements)` con un valor entero que especifique el número máximo de elementos durante la construcción de un objeto `ArrayLinearQueue`.

El método `insert(Object o)` de `ArrayLinearQueue` lanza una `FullQueueException` cuando `rear` identifica el elemento final del array uni-dimensional. El código de `FullQueueException` aparece en el siguiente listado:

```
// FullQueueException.java

package com.javajeff.cds;

public class FullQueueException extends RuntimeException {
}
```

El método `remove()` de `ArrayLinearQueue` lanza una `EmptyQueueException` cuando los objetos `front` y `rear` son iguales. El siguiente listado presenta el código de esta clase:

```
// EmptyQueueException.java

package com.javajeff.cds;

public class EmptyQueueException extends RuntimeException {
}
```

El siguiente listado presenta una implementación de `Queue` para una cola circular basada en un array uni-dimensional:

```
// ArrayCircularQueue.java

package com.javajeff.cds;

public class ArrayCircularQueue implements Queue {
     private int front = 0, rear = 0;
     private Object [] queue;

     public ArrayCircularQueue (int maxElements) {
            queue = new Object [maxElements];
     }

     public void insert (Object o) {
            int temp = rear;
            rear = (rear + 1) % queue.length;
            if (front == rear) {
                    rear = temp;
                    throw new FullQueueException ();
            }
            queue [rear] = o;
     }

     public boolean isEmpty () {
            return front == rear;
     }

     public boolean isFull () {
            return ((rear + 1) % queue.length) == front;
     }

     public Object remove () {
            if (front == rear)
                    throw new EmptyQueueException ();
            front = (front + 1) % queue.length;
            return queue [front];
     }
}
```

ArrayCircularQueue revela una implementación, en términos de variables privadas y un constructor, muy similar a ArrayLinearQueue. El método insert(Object o) es interesante porque guarda el valor actual de rear antes de hacer que esa variable apunte a la siguiente posición. Si la cola circular está llena, rear restaura su valor original antes de lanzar una FullQueueException. La restauración de rear es necesaria porque front es igual a rear (en ese punto), y una subsecuente llamada a remove() resulta en la lanzamiento de una EmptyQueueException (incluso aunque la cola circular no esté vacía).

Después de estudiar el código de la interfase y de varias clases que lo implementan basándose en arrays uni-dimensionales, consideremos en el siguiente listado una aplicación que demuestra las colas lineales y circulares:

```
// QueueDemo.java

import com.javajeff.cds.*;

class QueueDemo {
     public static void main (String [] args) {
            System.out.println ("ArrayLinearQueue Demo");
            System.out.println ("---------------------");
            queueDemo (new ArrayLinearQueue (5));
            System.out.println ("ArrayCircularQueue Demo");
            System.out.println ("---------------------");
            queueDemo (new ArrayCircularQueue (6)); // Need one more slot because
                                                    // of empty slot in circular
                                                    // implementation
     }

     static void queueDemo (Queue q) {
            System.out.println ("Is empty = " + q.isEmpty ());
            System.out.println ("Is full = " + q.isFull ());

            System.out.println ("Inserting \"This\"");
            q.insert ("This");

            System.out.println ("Inserting \"is\"");
            q.insert ("is");

            System.out.println ("Inserting \"a\"");
            q.insert ("a");

            System.out.println ("Inserting \"sentence\"");
            q.insert ("sentence");

            System.out.println ("Inserting \".\"");
            q.insert (".");

            try  {
                    System.out.println ("Inserting \"One last item\"");
                    q.insert ("One last item");
            }
            catch (FullQueueException e) {
                    System.out.println ("One insert too many");
                    System.out.println ("Is empty = " + q.isEmpty ());
                    System.out.println ("Is full = " + q.isFull ());
            }

            System.out.println ();

            while (!q.isEmpty ())
               System.out.println (q.remove () + " [Is empty = " + q.isEmpty () +
                   ", Is full = " + q.isFull () + "]");

            try {
                    q.remove ();
            }
            catch (EmptyQueueException e) {
                    System.out.println ("One remove too many");
            }
            System.out.println ();
     }
```

```
}
```

Cuando se ejecuta `QueueDemo`, se produce la siguiente salida:

```
ArrayLinearQueue Demo
---------------------
Is empty = true
Is full = false
Inserting "This"
Inserting "is"
Inserting "a"
Inserting "sentence"
Inserting "."
Inserting "One last item"
One insert too many
Is empty = false
Is full = true

This [Is empty = false, Is full = true]
is [Is empty = false, Is full = true]
a [Is empty = false, Is full = true]
sentence [Is empty = false, Is full = true]
. [Is empty = true, Is full = true]
One remove too many

ArrayCircularQueue Demo
---------------------
Is empty = true
Is full = false
Inserting "This"
Inserting "is"
Inserting "a"
Inserting "sentence"
Inserting "."
Inserting "One last item"
One insert too many
Is empty = false
Is full = true

This [Is empty = false, Is full = false]
is [Is empty = false, Is full = false]
a [Is empty = false, Is full = false]
sentence [Is empty = false, Is full = false]
. [Is empty = true, Is full = false]
One remove too many
```