Capítulo 16

Colas y colas de prioridad

Este capítulo presenta dos TADs: Colas y Colas de Prioridad. En la vida real, una **cola** es una fila de clientes esperando por algún servicio. En la mayoría de los casos, el primer cliente en la fila es el cliente más próximo a ser atendido. Sin embargo, hay algunas excepciones. Por ejemplo, en los aeropuertos, aquellos clientes cuyos vuelos están por partir son a veces atendidos a pesar de encontrarse a la mitad de la fila. También, en los supermercados, un cliente amable puede dejar pasar adelante a otro con sólo unos pocos productos.

La regla que determina quién es atendido a continuación se llama **disciplina de la cola**. La más simple es conocida como **FIFO**, del inglés "first-in-first-out"[1]. La disciplina más general, es la que se conoce como **encolado con prioridad**, en la cual a cada cliente se le asigna una prioridad, y el cliente con la prioridad más alta va primero, sin importar el orden de arribo. La razón por la cual digo que esta es la disciplina más general es que la prioridad puede ser basada en cualquier cosa: a qué hora sale el avión, cuántos productos tiene el cliente, o qué tan importante es el cliente. Desde luego, no todas las disciplinas de cola son "justas", pero la justicia se encuentra en el ojo del espectador.

El TAD Cola y el TAD Cola de Prioridad tienen el mismo conjunto de operaciones y sus interfaces son la misma. La diferencia se encuentra en la semántica de las operaciones: una cola usa una política FIFO, mientras que la Cola de Prioridad (como el nombre sugiere) usa una política de encolado con prioridades.

Como con la mayoría de los TADs, hay muchas formas de implementar colas. Dado que la cola es una colección de elementos, podemos usar cualquiera de los mecanismos básicos para almacenar elementos, incluyendo

1. N.d.T.:"El primero en entrar es el primero en salir".

arreglos y listas. Nuestra elección de cuál usaremos se basará en la eficiencia —cuánto toma efectuar las operaciones que queremos utilizar— y, en parte, en la facilidad de la implementación.

16.1 El TAD Cola

El TAD cola se define por las siguientes operaciones:

constructor: Crear una nueva cola, vacía.

agregar: Agregar un elemento a la cola.

quitar: Quitar un elemento de la cola y devolverlo. El elemento que se devuelve es el primero en haber sido agregado.

estaVacia: Verifica si la cola está vacía.

Esta es una implmentación de una Cola genérica, basada en la clase preincorporada `java.util.LinkedList`:

```
public class Cola {
    private LinkedList lista;

    public Cola() {
        lista = new LinkedList ();
    }

    public boolean estaVacia() {
        return lista.isEmpty();
    }

    public void agregar(Object obj) {
        lista.addLast (obj);
    }

    public Object quitar() {
        return list.removeFirst ();
    }
}
```

Un objeto cola contiene una única variable de instancia, que es la lista que la implementa. Para cada uno de los métodos, todo lo que tenemos que hacer es llamar a un método de la clase `LinkedList`.

16.2 Veneer

Al utilizar una LinkedList para implementar una Cola, podemos aprovechar el código ya existente; el código que escribimos sólo traduce métodos de LinkedList en métodos de Cola. Una implementación así se llama **veneer**[2]. En la vida real, el enchapado (veneer en inglés) es una fina capa de un madera de buena calidad que se utiliza en la manufactura de muebles para esconder madera de peor calidad por debajo. Los científicos de la computación utilizan esta metáfora para describir una pequeña porción de código que oculta los detalles de implementación y proveen una interfaz más simple o más estándar.

El ejemplo de la Cola demuestra una de las cosas interesantes de un veneer, y es que es fácil de implementar, y uno de los peligros, que es el **riesgo de eficiencia**.

Normalmente, cuando llamamos a un método, no nos preocupan los detalles de su implementación. Pero hay un "detalle" que sí querríamos conocer—la eficiencia del método. ¿Cuánto le toma ejecutarse, en función de la cantidad de elementos de la lista?

Para responder esta pregunta, debemos saber más acerca de la implementación. Si asumimos que `LinkedList` está efectivamente implementada como una lista enlazada, entonces la implementación de `removeFirst`[3] probablemente se vea parecido a esto:

```
public Object removeFirst () {
    Object resultado = cabeza;
    cabeza = cabeza.siguiente;
    return resultado.carga;
}
```

Asumimos que `cabeza` referencia al primer nodo de la lista, y que cada nodo contiene una `carga` y una referencia al siguiente nodo de la lista.

No hay ciclos, ni llamadas a funciones aquí, con lo que el tiempo de ejecución de este método, es más o menos el mismo, cada vez. A un método así se lo conoce como un método de **tiempo constante**.

La eficiencia del método `addLast`[4] es muy diferente. He aquí una implementación hipotética:

```
public void addLast (Object obj) {
  // caso especial: lista vacía
  if (cabeza == null) {
```

2. N.d.T.: En inglés significa "enchapado".
3. N.d.T.: En inglés significa `quitarPrimero`.
4. N.d.T.: En inglés significa `agregarAlFinal`.

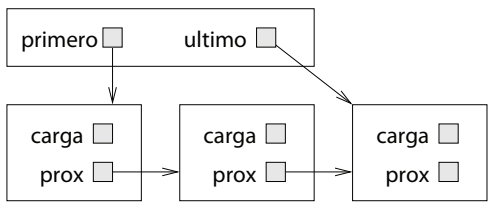
```
      cabeza = new Nodo(obj, null);
      return;
    }
    Nodo ultimo;
    for (ultimo = cabeza; ultimo.siguiente != null;
                                ultimo = ultimo.siguiente) {
      // atravesar la lista para encontrar el último nodo
    }
    ultimo.siguiente = new Nodo(obj, null);
  }
```

El primer condicional controla el caso especial de añadir un nuevo nodo en una lista vacía. En ese caso, nuevamente, el tiempo de ejecución no depende del largo de la lista. Sin embargo, en el caso general, debemos atravesar la lista para encontrar el último elemento de modo que podamos hacer que referencia al nuevo nodo.

Esta iteración toma un tiempo proporcional al largo de la lista. Dado que el tiempo de ejecución es una función lineal del largo de la lista, decimos que este método es de **tiempo lineal**. Comparado con tiempo constante, es muy malo.

16.3 Cola enlazada

Querríamos una implementación del TAD cola que pueda efectuar todas las operaciones en tiempo constante. Una forma de lograr esto es implementando una **cola enlazada**, que es similar a la lista enlazada en el sentido de que se compone de cero o más objetos `Nodo` enlazados. La diferencia es que la cola mantiene una referencia tanto al primero como al último nodo, como se muestra en la figura.



Así es cómo se vería una implementación de una `Cola` enlazada:

```
public class Cola {
    public Nodo primero, ultimo;

    public Cola () {
        primero = null;
        ultimo = null;
    }
    public boolean estaVacia() {
        return primero == null;
    }
}
```

Hasta acá es directa la implementación. En una cola vacía, tanto `primero` como `ultimo` son null. Para verificar si la lista está vacía, sólo tenemos que controlar uno sólo. El método `agregar` es un poco más complicado, porque tenemos que manejar varios casos especiales.

```
    public void agregar(Object obj) {
        Nodo nodo = new Nodo(obj, null);
        if (ultimo != null) {
            ultimo.siguiente = nodo;
        }
        ultimo = nodo;
        if (primero == null) {
            primero = ultimo;
        }
    }
```

La primera condición verifica que `ultimo` referencie algún nodo; si es así, debemos hacer que ese nodo referencie al nodo nuevo. La segunda condición maneja el caso especial de que la lista haya estado inicialmente vacía. En este caso, tanto `primero` como `ultimo` deben referenciar al nuevo nodo. El método `quitar` también controla varios casos especiales.

```
    public Object quitar() {
        Nodo resultado = primero;
        if (primero != null) {
            primero = primero.siguiente;
        }
        if (primero == null) {
            ultimo = null;
        }
        return resultado;
    }
```

La primera condición verifica si había algún nodo en la cola. Si así fuera, debemos hacer que el `primero` referencie al `siguiente`. La segunda condición controla el caso de que la lista ahora está vacía, en cuyo caso debemos hacer que `ultimo` valga también null.

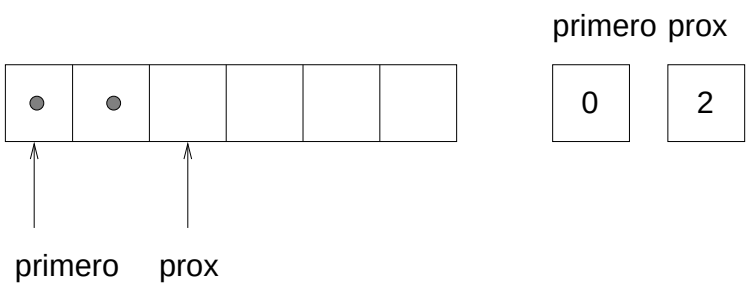
Como ejercicio, dibujá diagramas mostrando estas operaciones tanto en el caso normal, como en los casos especiales, y convencete de que son correctas.

Claramente, esta implementación es más complicada que la veneer, y es más difícil de demostrar que es correcta. La ventaja es que hemos alcanzado el objetivo: `agregar` y `quitar` son operaciones de tiempo constante.

16.4 Buffer circular

Otra implementación común de una cola es un **buffer circular**. La palabra "buffer" es un nombre general para un espacio de almacenamiento temporal, a pesar de que muchas veces se refiere a un arreglo, como ocurre en este caso. Qué quiere decir que sea "circular" debería quedar claro en breve.

La implementación de un buffer circular es similar a la implementación de la pila en la Sección 15.12. Los elementos de la cola se almacenan en un arreglo, y usamos índices para mantener un registro de dónde estamos ubicados dentro del arreglo. En la implementación de la pila, había un único índice que apuntaba al siguiente espacio disponible. En la implementación de cola, hay dos índices: `primero` apunta al espacio del arreglo que contiene el primer cliente en cola y `prox` apunta al siguiente espacio disponible. La siguiente figura muestra una cola con dos elementos (representados por puntos).



Hay dos maneras de pensar las variables `primero` y `ultimo`. Literalmente, son enteros, y sus valores se muestran en las cajas a la derecha. Abstractamente, sin embargo, son índices del arreglo, y por lo tanto, sue-

len dibujarse como flechas apuntando a ubicaciones en el arreglo. La representación de flecha es conveniente, pero deberías recordar que los índices no son referencias; son sólo enteros. Aquí hay una implementación incompleta de una cola sobre arreglo:

```
public class Cola {
    public Object[] arreglo;
    public int primero, prox;

    public Queue () {
        arreglo = new Object[128];
        primero = 0;
        prox = 0;
    }

    public boolean estaVacia () {
        return primero == prox;
    }
```

Las variables de instancia y el constructor salen de manera directa, sin embargo, nuevamente tenemos el problema de elegir un tamaño arbitrario para el arreglo. Más tarde resolveremos el problema, como hicimos con la pila, redimensionandolo si el arreglo se llena.

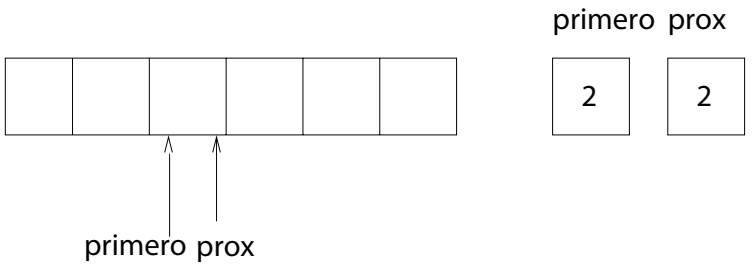
La implementación de `estaVacia` es algo sorprendente. Podrías haber pensado que `primero == 0` hubiera indicado una cola vacía, pero eso ignora el hecho de que la cabeza de la cola no está necesariamente al inicio del arreglo. En cambio, sabemos que la cola está vacía si `primero` es igual a `prox`, en cuyo caso no hay más elementos. Una vez que veamos la implementación de `agregar` y `quitar`, esta condición tendrá más sentido.

```
    public void agregar (Object elemento) {
        arreglo[prox] = elemento;
        prox++;
    }

    public Object quitar () {
        Object resultado = arreglo[primero];
        primero++;
        return resultado;
    }
```

El método `agregar` se parece mucho a `apilar` en la Sección 15.12; pone un nuevo elemento en el próximo espacio disponible y luego incrementa el índice. El método `quitar` es similar. Toma el primer elemento de la cola,

y luego incrementa `primero` de modo que referencie a la nueva cabeza de la cola. La siguiente figura muestra cómo se vería la cola luego de que ambos items se hayan quitado.



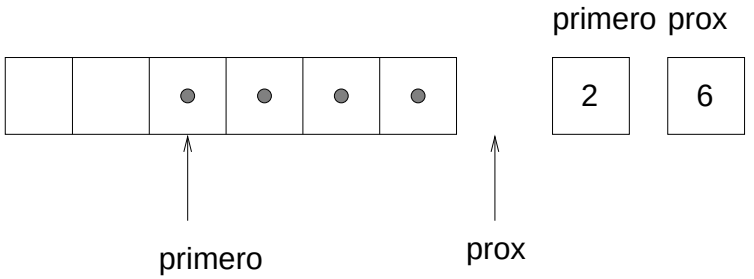
Siempre es cierto que `prox` apunta a un espacio disponible. Si `primero` alcanza a `prox` y apuntan al mismo espacio, entonces `primero` está apuntando a una posición "vacía", y la cola está vacía. Pongo "vacía" entre comillas porque es posible que la ubicación a la que apunta `primero` en realidad contenga un valor (no hacemos nada para asegurarnos de que las posiciones vacías contengan `null`); por otro lado, dado que sabemos que la cola está vacía, nunca vamos a leer esta posición, de modo que podemos pensarla, abstractamente, como si estuviera vacía.

Ejercicio 16.1

Modificar `quitar` para que devuelva `null` si la cola está vacía.

El otro problema de esta implementación es que eventualmente se quedará sin espacio. Cuando agregamos un elemento incrementamos `prox` y cuando quitamos un elemento incrementamos `primero`, pero nunca decrementamos ninguno de los dos. ¿Qué pasa cuando llegamos al final del arreglo?

La siguiente figura muestra la cola después de que agregamos cuatro elementos más:



El arreglo ahora está lleno. No hay "próximo espacio disponible," con lo que `prox` no tiene a dónde apuntar. Una posibilidad es redimensionar el arreglo, como hicimos con la implementación de la pila. Pero en ese caso el arreglo sólo continuaría creciendo independientemente de cuántos elementos haya efectivamente en la cola. Una mejor solución es dar la vuelta hasta el principio del arreglo y reutilizar los espacios disponibles ahí. Esta "vuelta" es la razón por la cual se conoce a esta implementación como buffer circular.

Una forma de reiniciar el índice es agregar un caso especial cuando lo incrementamos:

```
prox++;
if (prox == arreglo.length) prox = 0;
```

Una alternativa más vistosa es utilizar el operador módulo:

```
prox = (prox + 1) % arreglo.length;
```

En cualquier caso, tenemos un último problema por resolver. ¿Cómo sabemos si la cola está *realmente* llena, con la implicación de que no podemos agregar un nuevo elemento? La siguiente figura muestra cómo se vería la cola cuando esté "llena".