Estrategias de programación y estructuras de datos

Grado en Ingeniería Informática

Grado en Tecnologías de la Información

Departamento de Lenguajes y Sistemas informáticos

6

Tipos abstractos de datos lineales

Colas

Javier Vélez Reyes *jvelez@lsi.uned.es*

José Ignacio Mayorga Toledano *nmayorga@lsi.uned.es*

Departamento de Lenguajes Y Sistemas Informáticos

UNED

UNED

ETS de Ingeniería Informática

Índice

- Introducción
  - Tipos abstractos de datos lineales
  - El tipo abstracto de datos Cola
- Interfaz del tipo abstracto de datos Cola
- Implementaciones del tipo abstracto de datos Cola
  - Implementación dinámica
  - Implementación estática
  - Implementación con Lista
- Algoritmos sobre Colas
  - Recorrido de los elementos de una Cola
  - Búsqueda de un elemento sobre una Cola
- Problemas y ejercicios
- Bibliografía

Tipos abstractos de datos lineales. Colas

Objetivos generales

Objetivos

- Conocer la interfaz del tipo abstracto de datos Cola
- Aprender a implementar el TAD Cola mediante la interfaz QueueIF
  - En versión estática (con limitación explícita de capacidad máxima)
  - En versión dinámica (sin limitación explícita de capacidad máxima)
  - Utilizando una lista
- Conocer los principales problemas algorítmicos sobre colas
  - Recorrido de los elementos de una cola
  - Búsqueda de un elemento sobre una cola
- Practicar el diseño de funciones recursivas sobre cola

Tipos abstractos de datos lineales

Los tipos abstractos de datos lineales representan abstracciones en las que los datos son organizados de forma secuencial. A continuación se muestran los diferentes TADs que caen dentro de esta categoría y que se diferencian, esencialmente, en la forma en que se ofrecen acceso a sus datos

Tipos abstractos de datos lineales

I. Listas

El tipo abstracto de datos Lista representa una estructuras de datos donde los elementos se disponen de forma secuencial y las operaciones de inserción y extracción se aplican por el principio, que puede moverse recursivamente haciendo referencia a otras sublistas

◀ **Tema 4**

II. Pilas

El tipo abstracto de dato Pila representa una estructuras de datos donde los elementos se disponen de forma secuencial y las operaciones de inserción y extracción se aplican por el principio según la política de acceso último en entrar, primero en salir

◀ **Tema 5**

III. Colas

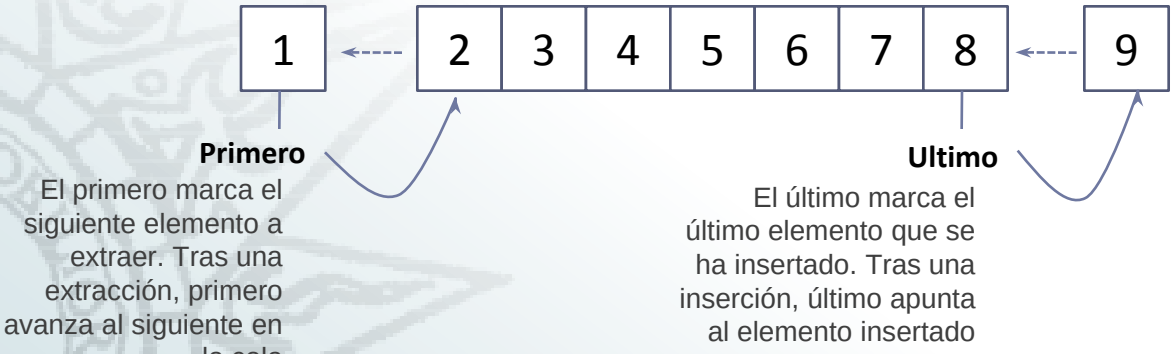
El tipo abstracto de dato Cola representa una estructuras de datos donde los elementos se disponen de forma secuencial y las operaciones de inserción y extracción se aplican por el final y por el principio respectivamente de acuerdo a la política de primero en entrar, primero en salir

◀ **Tema 6**

El tipo abstracto de datos Cola

Las colas son abstracciones de datos que organizan una colección de elementos de manera secuencial, donde existen dos puntos de interés llamados comienzo y final. Por el comienzo se realizan las operaciones de extracción mientras que por el final se realizan las inserciones. Según esto, la política de acceso prescribe que el primer elemento en entrar en una cola es el primero en salir de ella (First In First Out, FIFO).

Una cola es un tipo abstractos de datos que organiza una colección de elementos de forma secuencial. Las operaciones de extracción se realizan por el principio y las de inserción por el final de acuerdo a la política de acceso FIFO que prescribe que el primero en entrar es el primero en salir



La interfaz del tipo abstracto de datos Cola QueueIF <T>

El interfaz de Cola que utilizaremos a lo largo de este curso, QueueIF está compuesto por las operaciones que describimos a continuación

```
public interface QueueIF <T>
{
```

primero

Devuelve el elemento que se encuentra en la cabeza de la cola sin extraerlo de la misma. Esta es una operación consultora

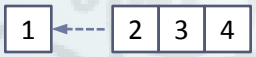
```
    /**
     * Devuelve la cabeza de la cola.
     * @return la cabeza de la cola.
     */
    public T getFirst ();
```

Insertar

La operación add inserta un elemento detrás del último elemento de la cola de manera que este pasa a ser el nuevo último

```
    /**
     * Inserta un nuevo elemento a la cola.
     * @param element El elemento a añadir.
     * @return la cola incluyendo el elemento.
     */
    public QueueIF<T> add (T element);
```

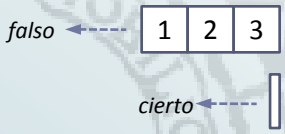
borrar



La operación remove extrae el elemento que se encuentra en la cabeza de la cola dejando al siguiente a éste como nueva cabeza

```
    /**
     * Borra la cabeza de la cola.
     * la cola excluyendo la cabeza.
     */
    public QueueIF<T> remove ();
```

Es vacía

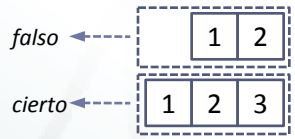


Este predicado indica si la cola no contiene ningún elemento. Semánticamente equivale a preguntar si el número de elementos es 0 (ver después)

```
    /**
     * Devuelve cierto si la cola esta vacia.
     * @return cierto si la cola esta vacia.
     */
    public boolean isEmpty ();
```

La interfaz del tipo abstracto de datos Cola QueueIF <T>

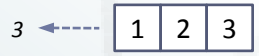
El interfaz de Cola que utilizaremos a lo largo de este curso, QueueIF está compuesto por las operaciones que describimos a continuación



Está llena

En implementaciones con capacidad máxima limitada tiene sentido preguntar si una cola ha alcanzado el número máximo de elementos que puede albergar

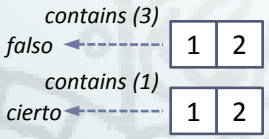
```
/**
 * Devuelve cierto si la cola esta llena.
 * @return cierto si la cola esta llena.
 */
public boolean isFull();
```



Longitud

La longitud de una cola devuelve el número de elementos que ésta contiene. Si la cola esta vacía devuelve 0. Si esta llena devuelve lsu capacidad máxima

```
/**
 * Devuelve el numero de elementos de la cola.
 * @return el numero de elementos de la cola.
 */
public int getLength ();
```



Búsqueda

La operación contains busca un elemento en la cola e indica si está contenido dentro de la misma

```
/**
 * Devuelve cierto si la cola contiene el elemento.
 * @param element El elemento buscado.
 * @return cierto si la cola contiene el elemento.
 */
public boolean contains (T element);
```

Recorrido

El recorrido de una cola devuelve un iterador que permite visitar todos los elementos de la misma según aparecen desde el primero hasta el último

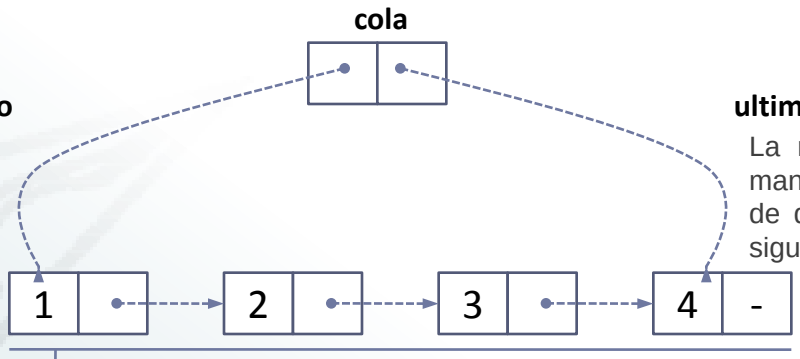
```
/**
 * Devuelve un iterador para la cola.
 * @return un iterador para la cola.
 */
public IteratorIF<T> getIterator ();
}
```

Implementación del tipo abstracto de datos Cola QueueIF <T>

Existen varias estrategias para implementar colas que responden al interfaz QueueIF anterior. En general, éstas se dividen en implementaciones dinámicas, que articulan colas de capacidad indefinida e implementaciones estáticas, con una capacidad máxima limitada y establecida como parámetro. A continuación presentamos varias implementaciones

Implementación dinámica basada en primero y último

Según esta estrategia de implementación, una cola está formada por una cadena de nodos enlazados con referencias al siguiente y dos referencias que apuntan al primer y último elemento de la cola. La desventaja de esta implementación es que hay que separar la abstracción de la cadena de nodos



Tipos abstractos de datos lineales. Colas

Implementación del tipo abstracto de datos Cola

Implementación dinámica basada en primero y siguiente

Una cola está formada por una cadena de nodos enlazados con referencias al siguiente y dos referencias que apuntan al primer y último elemento de la cola. La desventaja de esta implementación es que hay que separar la abstracción de la cadena de nodos

```
public class QueueDynamic <T> implements QueueIF <T> *
{
    private Node<T> first;
    private Node<T> last;

    public QueueDynamic () {
        first = null;
        last  = null;
    }

    public QueueDynamic (QueueIF<T> queue) {
        this ();
        if (queue != null)
            for (int i = 0; i < queue.getLength (); i++) {
                T element = queue.getFirst ();
                add (element);
                queue.remove ();
                queue.add (element);
            }
    }

    public QueueDynamic (ListIF<T> list)
    {
        this ();
        if (list != null) {
            ListIF<T> l = list;
            while (!l.isEmpty ()){
                add (l.getFirst ());
                l = l.getTail ();
            }
        }
    }
```

```
    @Override
    public T getFirst () {
        return first.getElement ();
    }

    @Override
    public QueueDynamic<T> add (T element) {
        if (element != null) {
            if (isEmpty ()) {
                Node<T> aNode = new Node<T> (element);
                first = aNode;
                last  = aNode;
            } else {
                Node<T> aNode = new Node<T> (element);
                last.setNext (aNode);
                last = aNode;
            }
        }
        return this;
    }

    @Override
    public QueueDynamic<T> remove () {
        if (getLength () == 1) {
            first = null;
            last  = null;
        }
        if (getLength () > 1) {
            first = first.getNext ();
        }
        return this;
    }
```

* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Implementación dinámica basada en primero y siguiente

Una cola está formada por una cadena de nodos enlazados con referencias al siguiente y dos referencias que apuntan al primer y último elemento de la cola. La desventaja de esta implementación es que hay que separar la abstracción de la cadena de nodos

```
@Override
public boolean isEmpty () {
    return first == null &&
           last == null;
}

@Override
public boolean isFull()  {
    return false;
}

@Override
public int getLength () {
    int length = 0;
    Node<T> node = first;
    while (node != null) {
        length ++;
        node = node.getNext ();
    }
    return length;
}

@Override
public boolean contains (T element){
    boolean found = false;
    Node<T> node = first;
    while (!found && node != null) {
        found = node.getElement ().equals (element);
        node  = node.getNext ();
    }
    return found;
}
```

```
@Override
public IteratorIF<T> getIterator ()
{
    QueueIF<T> handler = new QueueDynamic<T> (this);
    return new QueueIterator<T> (handler);
}

@Override
public int hashCode ()
{
    return 31 * ((first == null) ? 0 : first.hashCode ()) +
                ((last  == null) ? 0 : last.hashCode ());
}

@SuppressWarnings("unchecked")
@Override
public boolean equals (Object o)
{
    if (o == this) return true;
    if (o == null) return false;

    if (o.getClass () != this.getClass ()) return false;
    else {
        QueueDynamic<T> q = (QueueDynamic<T>) o;
        return q.first.equals (first) &&
               q.last.equals (last);
    }
}
```

* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Implementación dinámica basada en primero y siguiente

Una cola está formada por una cadena de nodos enlazados con referencias al siguiente y dos referencias que apuntan al primer y último elemento de la cola. La desventaja de esta implementación es que hay que separar la abstracción de la cadena de nodos

```
    @Override
    public String toString ()
    {
        StringBuffer buff = new StringBuffer ();
        buff.append ("QueueDynamic - [");

        IteratorIF<T> queueIt = getIterator ();
        while (queueIt.hasNext ()) {
            T element = queueIt.getNext ();
            buff.append (element);
            if (queueIt.hasNext ())
                buff.append (", ");
        }

        buff.append ("]");
        return buff.toString ();
    }
}

class Node <T>
{
    private T element;
    private Node<T> next;

    public Node ()
    {
        this.element = null;
        this.next    = null;
    }

    public Node (T element) {
        this ();
        this.element = element;
    }

    public Node (T element, Node<T> next) {
        this ();
        this.element = element;
        this.next    = next;
    }

    public T getElement ()
    {
        return element;
    }

    public void setElement (T element)
    {
        this.element = element;
    }

    public Node<T> getNext ()
    {
        return next;
    }

    public void setNext (Node<T> next)
    {
        this.next = next;
    }
}
```

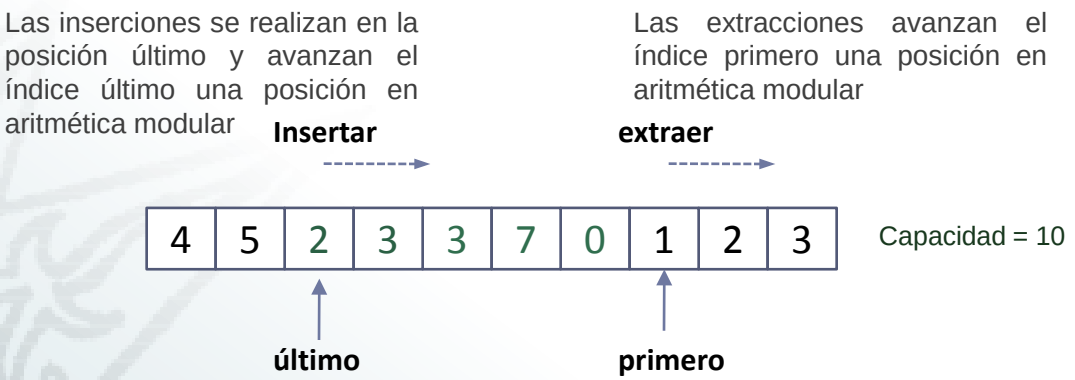
* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Implementación del tipo abstracto de datos Cola QueueIF <T>

Existen varias estrategias para implementar colas que responden al interfaz QueueIF anterior. En general, éstas se dividen en implementaciones dinámicas, que articulan colas de capacidad indefinida e implementaciones estáticas, con una capacidad máxima limitada y establecida como parámetro. A continuación presentamos varias implementaciones

Implementación estática basada en array circular

Según esta estrategia de implementación, una cola se almacena en un array de capacidad limitada accedido circularmente mediante dos índices en aritmética modular que apuntan al primer y ultimo elemento



Implementación estática basada en array circular

Según esta estrategia de implementación, una cola se almacena en un array de capacidad limitada accedido circularmente mediante dos índices en aritmética modular que apuntan al primer y ultimo elemento

```
public class QueueStatic <T> implements QueueIF <T> *
{
    private Object[] elements;
    private int capacity;
    private int first;
    private int last;

    public QueueStatic (int capacity)
    {
        this.elements = new Object [capacity + 1];
        this.capacity = capacity + 1;
        this.first    = 0;
        this.last     = 0;
    }

    public QueueStatic (QueueIF<T> queue, int capacity)
    {
        this (capacity);
        if (queue != null)
            for (int i = 0; i < queue.getLength (); i++) {
                T element = queue.getFirst ();
                add (element);
                queue.remove ();
                queue.add (element);
            }
    }
```

```
public QueueStatic (ListIF<T> list, int capacity)
{
    this (capacity);
    if (list != null) {
        ListIF<T> l = list;
        while (!l.isEmpty ()){
            add (l.getFirst ());
            l = l.getTail ();
        }
    }
}

@SuppressWarnings("unchecked")
@Override
public T getFirst ()
{
    if (isEmpty ()) return null;
    return (T) elements [first];
}

@Override
public QueueStatic<T> add (T element)
{
    if (element != null) {
        if (!isFull ()) {
            elements [last] = element;
            last = next (last);
        }
    }
    return this;
}
```

* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Implementación estática basada en array circular

Según esta estrategia de implementación, una cola se almacena en un array de capacidad limitada accedido circularmente mediante dos índices en aritmética modular que apuntan al primer y ultimo elemento

```
@Override
public QueueStatic<T> remove ()  {
    if (! isEmpty ()) {
        first = next (first);
    }
    return this;
}

@Override
public boolean isEmpty () {
    return first == last;
}

@Override
public boolean isFull() {
    return next(last) == first;
}

@Override
public int getLength () {
    if (first <= last) return last - first;
    else return capacity - (first - last);
}

private int next (int index)
{
    return (index + 1) % capacity;
}
```

```
    @Override
    public boolean contains (T element)
    {
        boolean found = false;
        int index = first;
        while (!found && Math.abs (last - index) > 0) {
            found = elements [index].equals (element);
            index = next (index);
        }
        return found;
    }

    @Override
    public IteratorIF<T> getIterator ()
    {
        QueueIF<T> handler = new QueueStatic<T> (this, capacity);
        return new QueueIterator<T> (handler);
    }

    @Override
    public int hashCode () {...}

    @SuppressWarnings("unchecked")
    @Override
    public boolean equals (Object o) {...}

    @Override
    public String toString () {...}
}
```

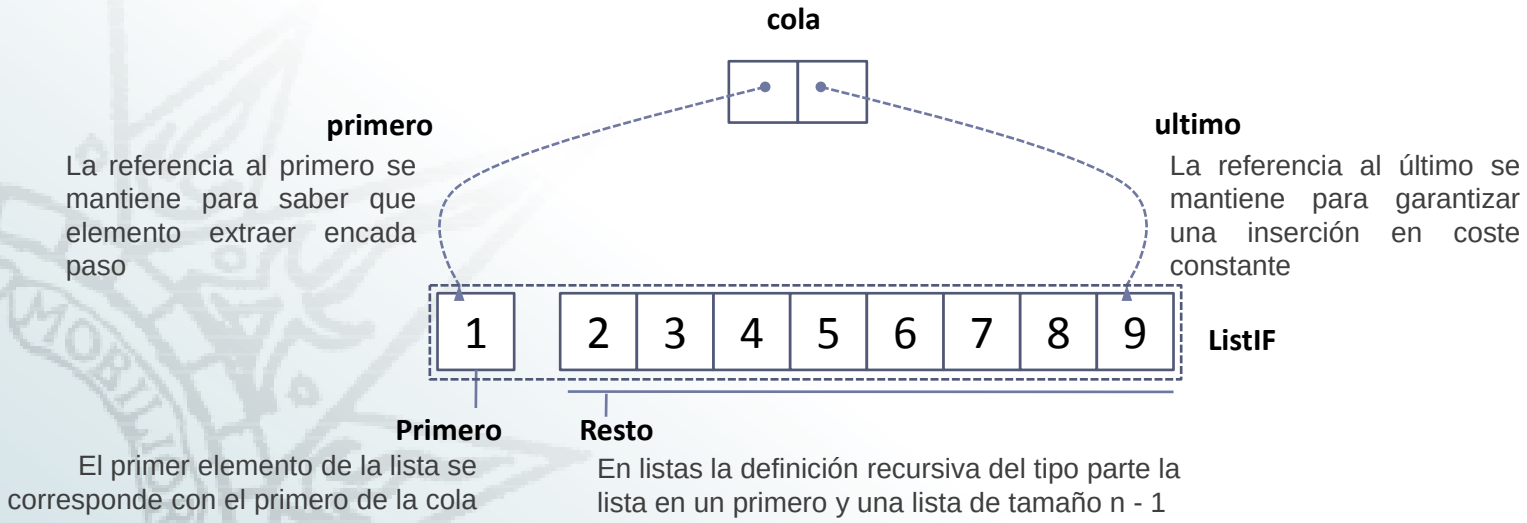
* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Implementación del tipo abstracto de datos Cola QueueIF <T>

Existen varias estrategias para implementar colas que responden al interfaz QueueIF anterior. En general, éstas se dividen en implementaciones dinámicas, que articulan colas de capacidad indefinida e implementaciones estáticas, con una capacidad máxima limitada y establecida como parámetro. A continuación presentamos varias implementaciones

Implementación basada en lista

Dado que la implementación dinámica de colas que hemos visto es bastante similar a la de listas, es posible implementar una cola a partir de una lista. El carácter estático o dinámico de la implementación dependerá del tipo de implementación de lista utilizada



Implementación estática basada en lista

Dado que la implementación dinámica de colas que hemos visto es bastante similar a la de listas, es posible implementar una cola a partir de una lista. El carácter estático o dinámico de la implementación dependerá del tipo de implementación de lista utilizada

```
public class QueueList <T> implements QueueIF <T>
{
    private ListIF<T> first;
    private ListIF<T> last;

    public QueueList ()
    {
        first = new ListDynamic<T> ();
        last  = first;
    }

    public QueueList (QueueIF<T> queue)
    {
        this ();
        if (queue != null)
            for (int i = 0; i < queue.getLength (); i++) {
                T element = queue.getFirst ();
                add (element);
                queue.remove ();
                queue.add (element);
            }
    }

    public QueueList (ListIF<T> list)
    {
        first = new ListDynamic<T> (list);
        ListIF<T> l = list;
        while (!l.isEmpty ()) l = l.getTail ();
        last = l;
    }
```

```
    @Override
    public T getFirst ()
    {
        return first.getFirst ();
    }

    @Override
    public QueueList<T> add (T element)
    {
        last.insert (element);
        last = last.getTail ();
        return this;
    }

    @Override
    public QueueList<T> remove ()
    {
        first = first.getTail ();
        return this;
    }

    @Override
    public boolean isEmpty ()
    {
        return first.isEmpty ();
    }
```

* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Implementación estática basada en lista

Dado que la implementación dinámica de colas que hemos visto es bastante similar a la de listas, es posible implementar una cola a partir de una lista. El carácter estático o dinámico de la implementación dependerá del tipo de implementación de lista utilizada

```
@Override
public boolean isFull()
{
    return false;
}

@Override
public int getLength ()
{
    return first.getLength ();
}

@Override
public boolean contains (T element)
{
    return first.contains (element);
}

@Override
public IteratorIF<T> getIterator ()
{
    return first.getIterator ();
}

@Override
public int hashCode ()
{
  return 31 * ((first == null) ? 0 : first.hashCode ())
            + ((last  == null) ? 0 : last.hashCode ());
}
```

```
@SuppressWarnings("unchecked")
@Override
public boolean equals (Object o)
{
    if (o == this) return true;
    if (o == null) return false;

    if (o.getClass () != this.getClass ()) return false;
    else {
        QueueList<T> q = (QueueList<T>) o;
        return q.first.equals (first) &&
               q.last.equals (last);
    }
}

@Override
public String toString ()  {
    StringBuffer buff = new StringBuffer ();
    buff.append ("QueueList - [");

    IteratorIF<T> queueIt = getIterator ();
    while (queueIt.hasNext ()) {
        T element = queueIt.getNext ();
        buff.append (element);
        if (queueIt.hasNext ())
            buff.append (", ");
    }

    buff.append ("]");
    return buff.toString ();
}
}
```

* Los comentarios de las cabeceras se han omitido por cuestiones de espacio

Algoritmos sobre colas

A diferencia de lo que ocurría en el caso de las listas donde estudiamos que la ordenación era uno de los problemas recurrentes, en el caso de las colas se considera que tal tipo de manipulaciones no son de interés ya que irían en contra de la esencia organizativa del tipo – mantener a los elementos dispuestos en el orden de llegada de acuerdo a una política FIFO. Por ello, en esta sección nos centraremos en discutir el problema del recorrido de los elementos de una cola y la búsqueda de un elemento dentro de una cola.

Algoritmos sobre colas

I. Recorrido

El recorrido de los elementos de una cola devuelve un iterador que permite acceder secuencialmente a los mismos según aparecen almacenados en la estructura, desde el primero hasta el último de la cola. El recorrido en sentido contrario es fácilmente implementable y se deja como ejercicio

II. Búsqueda

La búsqueda de un elemento sobre los elementos de una cola se realiza de forma secuencial dada la definición del tipo lo que imprime un coste lineal al algoritmo. El problema fundamental en este tipo de algoritmos es que dicha búsqueda es destructiva y requiere o utilizar un iterador (con copia) o reconstruir la estructura tras el algoritmo

Independencia de la implementación

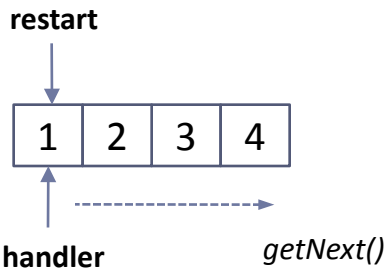
Recorrido de los elementos de una cola

Para articular el recorrido secuencial de los elementos de una cola debe implementarse el interfaz IteratorIF. Las implementaciones de QueueIF anteriores construyen una copia de la cola como manejador para garantizar que la iteración no modifique el estado del tipo

```
class QueueIterator<T> implements IteratorIF<T>
{
    private QueueIF<T> handler;
    private QueueIF<T> restart;

    /**
     * Constructor para QueueIterator.
     * @param handler el manejador de colas.
     */
    public QueueIterator (QueueIF<T> handler)
    {
        this.handler = handler;
        this.restart = new QueueDynamic<T> (handler);
    }

    /**
     * Devuelve el siguiente elemento de la iteracion.
     * @return el siguiente elemento de la iteracion.
     */
    @Override
    public T getNext ()
    {
        T element = handler.getFirst ();
        handler.remove ();
        return element;
    }
```

Iteración con reset

Se mantienen 2 manejadores constantemente, uno para articular el recorrido de la cola. El otro siempre apunta a la cabeza de la cola y se utiliza para implementar la operación de reset del iterador



Avance

La operación de avance devuelve el elemento en cima de la pila y actualiza el manejador de recorrido extrayendo la cima. Para invocar esta operación con seguridad hay que asegurarse de que el manejador no apunta a la cola vacía…

Recorrido de los elementos de una cola

Para articular el recorrido secuencial de los elementos de una cola debe implementarse el interfaz IteratorIF. Las implementaciones de QueueIF anteriores construyen una copia de la cola como manejador para garantizar que la iteración no modifique el estado del tipo

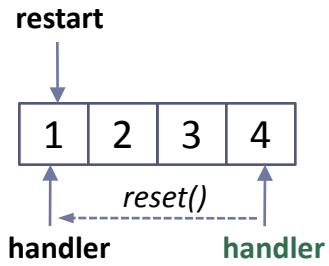
```
/**
     * Devuelve cierto si existen mas elementos a iterar.
     * @return cierto si existen mas elementos a iterar.
     */
    @Override
    public boolean hasNext ()
    {
        return !handler.isEmpty ();
    }

    /**
     * Restablece el iterador para volver iterar.
     */
    @Override
    public void reset ()
    {
        handler = new QueueDynamic<T> (restart);
    }
}
```

Hay siguiente

Para saber si el iterador contiene más elementos y por tanto puede volver a invocarse la operación de avance o no, hay que preguntar si el manejador ha alcanzado el final de la cola

Reset

Una vez que un iterador ha llegado a su fin sólo se puede construir otro iterador, o resetear éste a su valor inicial. En este caso sin embargo, la operación de reset requiere una copia con lo que el coste es idéntico al de construir un nuevo iterador.



Búsqueda de un elemento sobre una cola

La búsqueda de un dato sobre los elementos contenidos en una cola es un problema de recorrido secuencial que termina cuando se encuentra una cabeza que corresponde con el dato buscado. Dado que el tipo no tiene una definición recursiva, el problema de búsqueda no se presta a este tipo de resolución algorítmica. A continuación presentamos dos versiones de la búsqueda iterativa con centinela que aprovechan la estructura interna de la implementación del tipo

Búsqueda en QueueDynamic

Es esta implementación se realiza una búsqueda con centinela sobre la estructura de nodos enlazados que implementar el tipo cola

```
@Override
public boolean contains (T element){
    boolean found = false;
    Node<T> node = first;
    while (!found && node != null) {
        found = node.getElement ().equals (element);
        node  = node.getNext ();
    }
    return found;
}
```

Búsqueda en QueueStatic

En la implementación estática los datos son almacenados sobre una estructura vectorial que debe ser recorrida circularmente desde el primero hasta el último elemento, si el centinela no lo impide

```
@Override
public boolean contains (T element)
{
    boolean found = false;
    int index = first;
    while (!found && Math.abs (last - index) > 0) {
        found = elements [index].equals (element);
        index = next (index);
    }
    return found;
}
```

Ejercicios

Dado el carácter destructivo de las operaciones sobre colas – para avanzar en una cola es necesario extraer su cabeza con invocando a remove – la realización de muchos de sus algoritmos requiere de un proceso de protección consistente en hacer una copia de la misma. Ignorando el problema de destrucción resuelva los siguientes problemas

I. Longitud de una cola

Diseñe una función que calcule el número de elementos que contiene una cola

II. Pasar a lista

Devolver una lista con los elementos de una cola según aparecen en ésta desde el primero al último

III. Subcola

Diseñe una función que dada una cola determine si es subcola de otra dada de mayor tamaño

IV. Extraer pares

Función que extrae los números pares de una cola de enteros. Repita el ejercicio para extraer los de posición par en la cola

V. Insertar en orden

Diseñe una función que inserte en una cola de enteros los elementos de forma ordenada crecientemente

VI. Concatenar dos colas

Diseñe una función recursiva que devuelve la cola resultante de concatenar dos colas

VII. Cola inversa

Diseñe una función recursiva que devuelva una cola con los elementos colocados de forma inversa a otra

VIII. Borrado de un elemento

Diseñe una función que devuelva una cola que elimine la primera aparición de un elemento

IX. Borrar todos

Diseñe una función que devuelva una cola donde se hayan eliminado todas las apariciones de un elemento

X. Cola prefijo

Diseñe una función que devuelva una cola con los elementos que preceden a un elemento dado

XI. Cola sufijo

Diseñe una función que devuelva la cola sufijo que sucede a un elemento dado

XII. Cola mayores

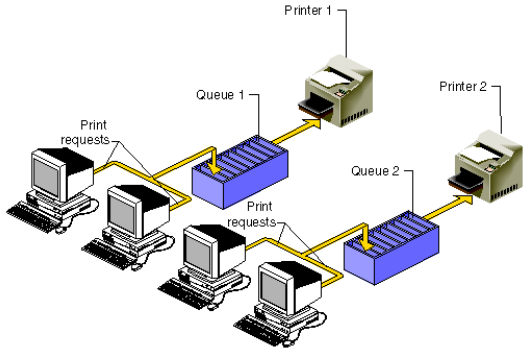
Diseñe una función recursiva que devuelva una cola con todos los elementos mayores a uno dado

Problemas

Las colas son un tipo abstracto de datos que se utiliza recurrentemente en los problemas de cierta complejidad de programación. A continuación planteamos algunos ejemplos de problemas que se resuelven con el uso de colas.

Competición de procesos por un recurso

Diferentes procesos de un sistema operativo compiten en ejecución por adquirir el uso de una impresora que esta a su vez conectada a varios equipos dispuestos en una red de área local. La cola de impresión es una abstracción donde se encolan los procesos a la espera de ser atendidos por el servidor de impresión



Competición de procesos por un recurso

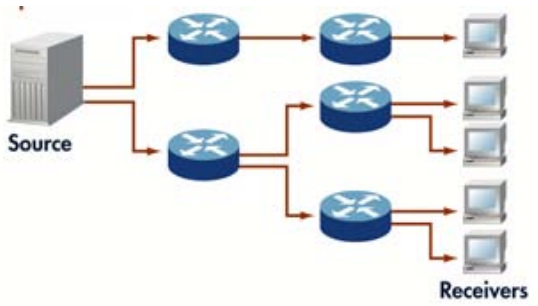
En escenarios concurrentes los procesos productores deben comunicarse con los consumidores de información. Dado que el ritmo de trabajo de unos y otros no es el mismo se utilizan colas de almacenamiento de mensajes como mecanismo de amortiguación

Problemas

Las colas son un tipo abstracto de datos que se utiliza recurrentemente en los problemas de cierta complejidad de programación. A continuación planteamos algunos ejemplos de problemas que se resuelven con el uso de colas.

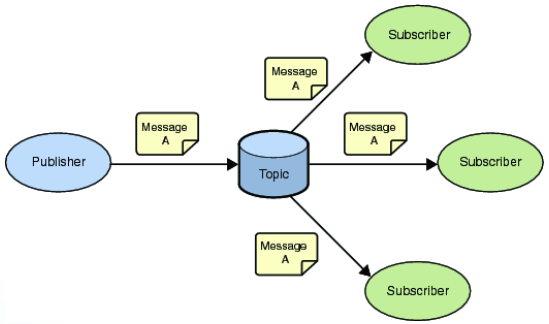
Enrutamiento de paquetes IP

En las redes de computadoras los dispositivos de interconexión utilizan colas para almacenar la información entrante y entregarla a sus destinatarios al ritmo que ellos requieran. Asimismo se utiliza esta técnica para realizan propagación en broadcast o multicast



Computación distribuida

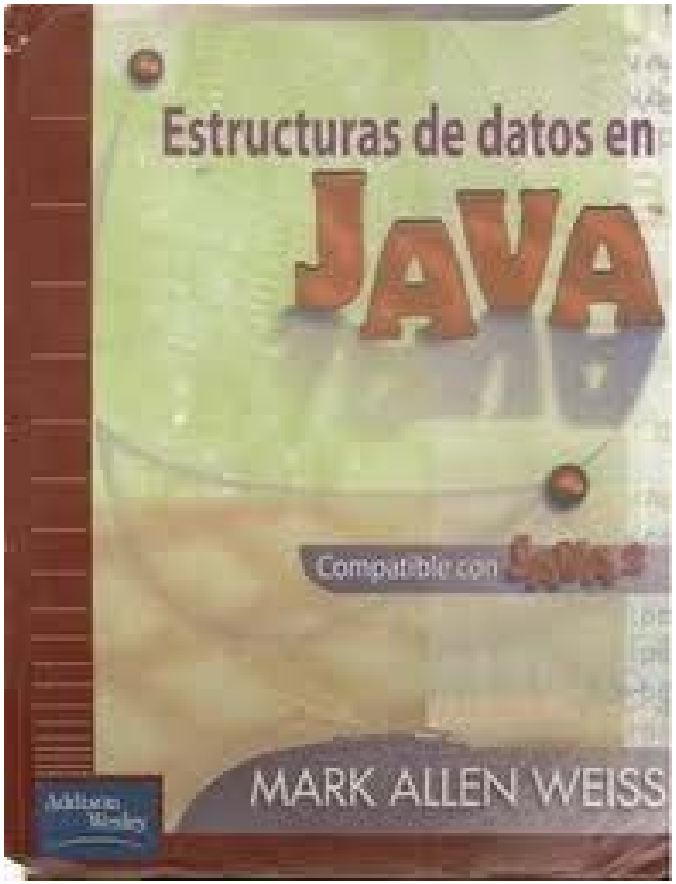
En computación distribuida se utilizan sistemas de mensajería encargados de mover la información. En comunicación punto a punto las colas se utilizan para persistir los mensajes y garantizar la entrega. En comunicación de eventos varios procesos se registran como escuchadores de una cola donde otros se publican eventos de un determinado tópico



Bibliografía

Bibliografía básica

Estructuras de datos en java. Weiss, Mark Allen. Pearson Addison – Wesley. ISBN 9788478290352



Bibliografía

Bibliografía complementaria

Estructura de datos y algoritmos en java. A. Drozdek. Thomsom. ISBN: 9706866116. 2007

Estructuras de datos con Java. J. Lewis, J. Chase. Pearson – Addison Wesley. ISBN 13: 9788420541914