Listas Enlazadas en Java

AGOSTO 15, 2022 / SIN COMENTARIOS

Las listas enlazadas en Java constituyen la base de las estructuras de datos y sustentan una gran cantidad de algoritmos, en este artículo se explicaran estas listas basadas en memoria estática y memoria dinámica.

Repasa los conceptos fundamentales de Java en este vínculo.

1. Necesidad

Las listas enlazadas surgen de la necesidad de manejar de forma más eficiente la memoria, debido a que los arreglos nativos, si bien proporcionan toda la funcionalidad requerida respecto al manejo de listas tiene un gran inconveniente y es que todas sus posiciones se encuentran contiguas (juntas) en memoria, una a continuación de la otra. Suponga entonces que se crea un arreglo de gran tamaño, afortunadamente en Java cada casilla contendrá una referencia a un objeto, es decir cada casilla tiene un tamaño de 32 o 64 bits dependiendo de la arquitectura de la máquina, sin embargo, si el arreglo es muy grande es posible que no se disponga de un gran bloque de memoria contigua para albergar el arreglo.

El problema de encontrar este espacio de memoria disponible empeora con el tiempo que lleve ejecutándose el programa, puesto que la creación de objetos y su recolección ocasionan huecos en la memoria en un fenómeno conocido como fragmentación de la memoria (muy parecido al que sucede en el disco duro, conocido como fragmentación del disco). La Figura 1. muestra un ejemplo de items almacenados en casillas de un arreglo.

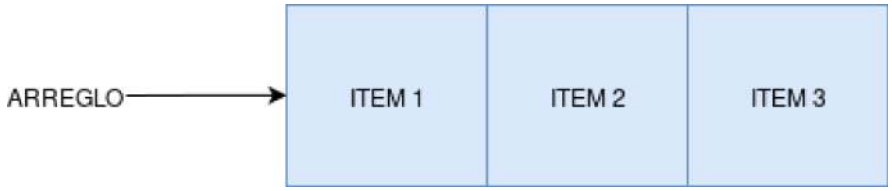

Figura 1. *Arreglo tradicional*

La idea tras la lista enlazada es permitir que los items que se almacenan en las casillas no tengan que estar necesariamente uno detrás del otro, y para lograr esto las listas enlazadas se forman usando el concepto de clase autoreferenciada que se muestra a continuación.

2. Clases autoreferencias

Las listas enlazadas y muchas otras estructuras de datos se forman utilizando clases autoreferenciadas que no es otra cosa que una clase que contiene al menos un atributo cuyo tipo coincide con la misma clase, en otras palabras una clase autoreferenciada tiene una referencia a un objeto del mismo tipo de la clase.

En la Figura 2, se muestra un ejemplo de una clase autoreferenciada denominada `Node` que representa una casilla de una lista enlazada, observe como el nodo tiene un espacio de memoria para referenciar el objeto contenido en la casilla y una referencia al nodo siguiente.

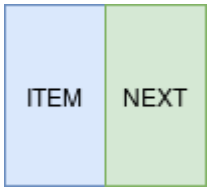

Figura 2. *Nodo lista enlazada*

3. Estructura de una Lista Enlazada

Las listas enlazadas aprovechan el concepto de clases autoreferenciadas para encadenar nodos y permitir armar una lista, observe que únicamente se requiere saber cual es el primer nodo de la lista, puesto que avanzando a través de la referencias `next` de cada nodo se puede llegar a cualquier otro nodo en la lista. El hecho de conocer la primera posición de la lista es algo que nos es familiar, ya que

en el caso de los arreglos el nombre del arreglo realmente es un apuntador al primer elemento de la lista.

Observe que la lista tal como se plantea resuelve el problema de memoria mencionado anteriormente, es decir, al estar los nodos desligados los unos de los otros ya no es necesario para grandes arreglos encontrar un bloque contiguo de memoria, infortunadamente esta solución genera un nuevo inconveniente y es el desplazamiento por la lista, supongamos que se desea acceder a la casilla tercera de la lista, será necesario entonces a partir de la primera casilla iterar para posicionar una referencia que termine apuntando a la tercera casilla, esta iteración toma tiempo y empeora con el tamaño de la lista, situación que no sucede con los arreglos puesto que ubicarse en una posición cualquiera es simplemente una operación aritmética, por lo anterior indexar una posición en la lista enlazada es una operación de tiempo lineal con el tamaño de la lista, en cambio en el arreglo es una operación de tiempo constante.

El programador debe elegir adecuadamente su estructura de datos de tal manera que supla las necesidades del algoritmo, teniendo en cuenta también aspectos de rendimiento.

Figura 3. *Lista enlazada*

4. Implementación de Lista Enlazada

Basados en el gráfico de la Figura 3, podemos realizar la implementación de una lista enlazada, para ello consideraremos tres clases

- Interface: especifica los métodos públicos que implementaremos para las listas.
- Nodo: implementación de la clase autoreferenciada para representar las casillas de la lista enlazada.
- Lista: implementación de todos los métodos y utilidades de la lista enlazada.

El código que se muestra a continuación implementa las tres clases anteriores. Note que la implementación de la interface pude darse con diversas técnicas y formas, no solo con clases autoreferenciadas, de tal manera que para la misma interface se puede tener diferentes implementaciones, la primera que se mostrará es la lista enlazada, y la segunda la lista implementada con arrays.

4.1. Interface

Esta interface representa las operaciones habituales con la lista, observe algo interesante en el caso de los métodos add, los cuales devuelven una lista enlazada, esto a primera vista puede parecer extraño, pero realmente la idea es devolver la misma lista que llamo el método add y de esta forma habilitar el encadenamiento de llamados a métodos de la lista, por ejemplo, se pueden hacer llamados así `list.add(obj1).add(obj2)` y asi sucesivamente.

Código 1. Interface de una lista enlazada

```
package lists;
/**
 * Provee las acciones que se pueden realizar con una lista
 * @author ochoscar
 * @param < T > Tipo genérico de los cuales se almacenaran elementos dentro de la lista
 */
public interface IList< T > {
    /**
     * Método que verifica si la lista esta vacía
     * @return Devuelve true si la lista esta
     * vacía y false en caso contrario
     */
    public boolean isEmpty();
    /**
     * Obtiene el primer item de la lista
     * @return Devuelve la Clase al principio de la lista
     * null en caso que la lista este vacía
     */

    public T getFirst();
    /**
     * Obtiene el ultimo elemento de la lista
     * @return Devuelve la ultima Clase en la lista
     * null si la lista esta vacía
     */
    public T getLast();
    /**
     * Devuelve el i - esimo elemento de la lista
     * @param i Posición del elemento a devolver (comienza en 0)
     * @return Devuelve la Clase en la posición i
     */
    public T get(int i);
    /**
     * Método que establece un item de la lista en una posición especifica
     * @param p Objeto que sera establecido en una posición especifica
     * @param i Posición a establecer el objeto
     */
    public void set(T p, int i);
    /**
     * Método que determina el tamaño de la lista
     * @return Devuelve un entero que indica el tamaño de la lista
     */
    public int size();
    /**
     * Método que inserta al final de la lista
     * @param p Objeto a insertar
     * @return Devuelve la propia lista enlazada
     */
    public IList< T > addLast(T p);
    /**
     * Método que permite agregar un objeto en un posición
     * arbitraria de la lista
     * @param p Objeto que se quiere agregar a la lista
     * @param i posición en la cual se quiere agregar
     * @return Devuelve la propia lista enlazada
     */
    public IList< T > add(T p, int i);
    /**
```

```
    /**
     * Método que elimina un elemento dado un objeto
     * @param p Objeto a eliminar
     */
    public void remove(T p);
    /**
     * Método que remueve un elemento de la lista
     * @param i Posición o índice a eliminar de la lista
     */
    public void remove(int i);
    /**
     * Devuelve si esta o no
     * @param p Objeto a verificar si esta en la lista
     * @return Devuelve true si p esta en la lista false sino
     */
    public boolean contains(T p);
}
```

4.2. Nodo

La clase que establece la raíz de la auto referenciación es esta y permite además almacenar el objeto que esta contenido en esta casilla de la lista. El código proporcionado muestra que esta clase no ofrece ninguna operación particular, solamente almacena los atributos para enlazar los nodos y así conformar la lista.

Código 2. Clase nodo

```
package linkedlist;
/**
 * Clase auto referenciada para construir la estructura
 * tipo lista enlazada que contendrá tanto el item almacenado
 * como la referencia al siguiente elemento de la lista
 * @author ochoscar
 */
public class Node< T > {
    ////////////////////
    // Atributos
    ////////////////////
    /** Referencia al tipo T que ocupa el nodo */
    private T item;
    /** Autoreferencia al siguiente elemento */
    private Node< T > next;
    ////////////////////
    // Métodos
    ////////////////////
    /**
     * Constructor por defecto
     */
    public Node() {
    }
    /**
     * Constructor con parámetros
     * @param pItem Item a almacenar en el nodo
     * @param pNext Siguiente elemento en la lista
     */
    public Node(T pItem, Node< T > pNext) {
        item = pItem;
        next = pNext;
    }
    /**
     * Constructor de copia
     * @param pNode Objeto que servirá para realizar la copia de los
     * atributos del nodo
     */
```

```
     */
    public Node(Node< T > pNode) {
        item = pNode.getItem();
        next = pNode.getNext();
    }
    /**
     * Método que devuelve el nodo en su
     * representación de string
     * @return Devuelve la representación en String
     */
    @Override

    public String toString() {
        return item.toString();
    }
    /**
     * @return the item
     */
    public T getItem() {
        return item;
    }
    /**
     * @param item the item to set
     */
    public void setItem(T item) {
        this.item = item;
    }
    /**
     * @return the next
     */
    public Node< T > getNext() {
        return next;
    }
    /**
     * @param next the next to set
     */
    public void setNext(Node< T > next) {
        this.next = next;
    }
}
```

4.3. Lista Enlazada

La clase lista implementa no solo la interface de lista sino también la interface iterator que permite devolver un objeto Iterador con el cual se recorrerá la lista, este objeto es utilizado automáticamente por java en la instrucción `for - each` que corresponde a la siguiente sintaxis `for(Obj obj : list)`

Código 3. Lista enlazada

```
package lists;
import java.util.Iterator;
/**
 * Clase que encapsula las diferentes acciones de una lista enlazada
 * @author ochoscar

 * @param < T > Tipo genérico que contra los item de la lista
 */
public class LinkedList< T > implements IList< T >, Iterable< T > {
    ////////////////////
    // Atributos
    ////////////////////
    /** Referencia al primer nodo de la lista */
    private Node< T > first;
    /** Tamaño de la lista */
    private int listSize = 0;
```

```java
////////////////////
// Métodos
////////////////////
/**
 * Obtiene el primer item de la lista
 * @return Devuelve la Clase al principio de la lista
 * null en caso que la lista este vacía
 */
@Override
public T getFirst() {
    return first != null ? first.getItem() : null;
}
/**
 * Obtiene el ultimo elemento de la lista
 * @return Devuelve la ultima Clase en la lista
 * null si la lista esta vacía
 */
@Override
public T getLast() {
    return listSize == 0 ? null : getNode(listSize - 1).getItem();
}
/**
 * Devuelve el i - esimo elemento de la lista
 * @param i Posición del elemento a devolver (comienza en 0)
 * @return Devuelve la Clase en la posición i
 */
@Override
public T get(int i) {
    if(i >= 0 && i < listSize) {
        return getNode(i).getItem();
    }
    return null;
}
/**
 * Método que establece un item de la lista en una posición especifica
 * @param p Objeto que sera establecido en una posición especifica
 * @param i Posición a establecer el objeto
 */
@Override
public void set(T p, int i) {
    if(i >= 0 && i < listSize) {
        getNode(i).setItem(p);
    }
}
/**
 * Método que determina el tamaño de la lista
 * @return Devuelve un entero que indica el tamaño de la lista
 */
@Override
public int size() {
    return listSize;
}
/**
 * Método que inserta al final de la lista
 * @param p Objeto a insertar
 * @return Devuelve la propia lista enlazada
 */
@Override
public IList< T > addLast(T p) {
    add(p, listSize);

    return this;
}
/**
 * Método que permite agregar un objeto en un posición
 * arbitraria de la lista
 * @param p Objeto que se quiere agregar a la lista
 * @param i posición en la cual se quiere agregar
 * @return Devuelve la propia lista enlazada
 */
@Override
```

```java
public IList< T > add(T p, int i) {
    if(i >= 0 && i <= listSize) {
        Node< T > newNode = new Node();
        newNode.setItem(p);
        // La lista esta vacía
        if(first == null) {
            first = newNode;
        } else {
            // La lista no esta vacía, y van a inserta
            // en la posición 0
            if(i == 0) {
                newNode.setNext(first);
                first = newNode;
            } else if(i == listSize) {
                // Se quiere insertar al final de la lista
                Node< T > last = getNode(listSize - 1);
                last.setNext(newNode);
            } else {
                Node< T > previous = getNode(i - 1);
                Node< T > current = previous.getNext();
                newNode.setNext(current);
                previous.setNext(newNode);
            }
        }
        listSize++;
    }
    return this;
}
/**
 * Método que elimina un elemento dado un objeto
 * @param p Objeto a eliminar
 */
@Override
public void remove(T p) {
    Node< T > aux = first;
    int i = 0;
    boolean find = false;
    if(contains(p)){
        while(true){
            if(aux.getItem().equals(p)){
                break;
            }else{
                aux = aux.getNext();
                i++;
            }
        }
        remove(i);
    }
}
/**
 * Método que remueve un elemento de la lista
 * @param i Posición o índice a eliminar de la lista
 */
@Override
public void remove(int i) {
    if(i >= 0 && i < listSize) {
        // La posición a eliminar es valida, se procede a eliminar
        if(i == 0) {
            first = first.getNext();
        } else {

            Node< T > previous = getNode(i - 1);
            previous.setNext(previous.getNext().getNext());
        }
        // Disminuye el tamaño de la lista
        listSize--;
    }
}
/**
 * Devuelve si esta o no
 * @param p Objeto a verificar si esta en la lista
```

```
     * @return Devuelve true si p esta en la lista false sino
     */
    @Override
    public boolean contains(T p) {
        Node< T > iterator = first;
        while(iterator != null) {
            if(iterator.getItem().equals(p)) {
                return true;
            }
            iterator = iterator.getNext();
        }
        return false;
    }
    /**
     * Método que verifica si la lista esta vacia
     * @return Devuelve true si la lista esta
     * vacia y false en caso contrario
     */
    @Override
    public boolean isEmpty() {
        return first == null;
    }
    /**
     * Método que devuelve la lista en su
     * representación de string
     * @return Devuelve la representación en String
     */
    @Override
    public String toString() {
        String strToReturn = "[ ";
        Node< T > iterator = first;
        while(iterator != null) {
            strToReturn += iterator.toString() + " ";
            iterator = iterator.getNext();
        }
        return strToReturn + "]";
    }
    /**
     * Implementación de la interface iterable que permite recorrer la lista
     * con ciclos estilo for - each
     * @return Devuelve el iterador para recorrer la lista
     */
    @Override
    public Iterator< T > iterator() {
        // Creación de un objeto anónimo para retornarlo
        return new Iterator< T >() {
            /** Ubica un puntero en el primer elemento de la lista a retornar */
            private Node< T > iteratorNode = first;
            /**
             * Método que indica si existen mas elementos a recorrer o no
             * @return True si existen mas elementos para recorrer y false en caso contrario
             */
            public boolean hasNext() {
                return iteratorNode.getNext() != null;
            }
            /**
             * Devuelve el elemento dond se encuentra parado el iterador y lo avanza
             * @return
             */
            public T next() {

                T item = iteratorNode.getItem();
                iteratorNode = iteratorNode.getNext();
                return item;
            }
            /**
             * Método que le permite al iterador remover un elemento de forma segura
             * En la lista hay que tener cuidado cuando se remueve mientras se itera puesto
             * que la eliminación cambia el tamaño de la lista. Este método
             * no se encuentra soportado en esta version.
             */
```

```
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
    /**
     * Método que devuelve un nodo dada una posición
     * @param pos Posición de la lista para retornar el nodo
     * @return Devuelve el nodo indicado en el parámetro pos
     */
    private Node< T > getNode(int pos) {
        int i = 0;
        Node< T > iterator = first;
        while(iterator != null) {
            if(i == pos) {
                return iterator;
            }
            i++;
            iterator = iterator.getNext();
        }
        return null;
    }
}
```

Las listas enlazadas también se pueden disponer como listas doblemente enlazada, donde cada nodo no solamente mantenga una referencia de su siguiente elemento sino también del elemento anterior. A continuación se muestra la implementación de listas usando nodos doblemente enlazados, note que cada nodo tiene dos referencias una apuntando hacía el elemento siguiente y otra apuntando hacía el elemento anterior.

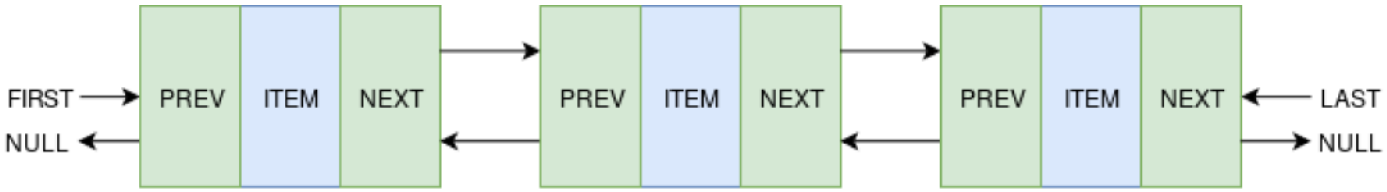

Figura 4. *Lista doblemente enlazada*

Algunas ventajas de esta lista es que ciertas operaciones de desplazamiento resultan más óptimas al no tener que recorrer siempre la lista completa desde el principio para posicionarse en cualquier elemento, considere en este sentido, por ejemplo, recorrer la lista en orden inverso. Otra lista importante es la **Lista Circular** que simplemente es aquella que se forma uniendo el principio de la lista con el final y en este caso es común utilizar un nodo ficticio para determinar un primer elemento de la lista

El programador debe elegir adecuadamente su estructura de datos de acuerdo con la naturaleza de los mismos y de esta forma se le facilitaran las operaciones, concentrarse en la lógica de fondo, realizar códigos menos propensos a errores y finalmente con mayor facilidad de mantenimiento.

5. Lista con arreglos

Como se había mencionado anteriormente la lista con arreglos puede ser una implementación de la misma interface `IList` . La ventaja en este caso es la facilidad para posicionarse en cualquier punto, sin embargo, al estar la lista conformada con arreglos, tiene las desventajas del uso de arreglos que se menciono al principio de esta sección. El arreglo es muy óptimo para indexare, pero cuando se acaba el tamaño es necesario crear otro nuevo con más tamaño y hacer un copiado de los elementos del arreglo original en el nuevo con mayor capacidad, esta operación ralentiza los métodos de la lista. Por esta razón las listas con arreglos deben tener mínimamente dos conceptos incluidos: el tamaño (que comúnmente se refiere al tamaño ocupado) y la capacidad (que es la máxima cantidad de elementos que puede tener el arreglo utilizado en la lista). A continuación, se presenta el código de la lista enlazada usando arreglos.

Código 4. Lista con arreglos

```
package lists;
import java.util.Iterator;
/**
 * Lista que usa arreglos nativos para implementar las operaciones de una lista
 * @author ochoscar
 * @param < T > Tipo genérico que contra los item de la lista
 */
public class ArrayList< T > implements IList< T >, Iterable< T >  {
    ////////////////////
    // Atributos
    ////////////////////
    /** Arreglo nativo de la lista que contiene los elementos y su longitud es
     la capacidad máxima de elementos a almacenar*/
    private T array[];
    /** Tamaño de la lista que refleja la cantidad de casillas del arreglo usadas */
    private int listSize;
    ////////////////////
    // Métodos
    ////////////////////
    /**
     * Constructor por defecto que crea la lista con 10 casillas de capacidad
     */
    public ArrayList() {
        array = (T[]) new Object[10];
        listSize = 0;
    }
    /**
     * Constructor especificando la capacidad inicial
     * @param initCapacity Capacidad inicial
     */
    public ArrayList(int initCapacity) {
        array = (T[]) new Object[initCapacity];
        listSize = 0;
    }
    /**
     * Método que verifica si la lista esta vacía
     * @return Devuelve true si la lista esta
     * vacía y false en caso contrario
     */
    @Override
    public boolean isEmpty() {
        return listSize == 0;
    }
    /**
     * Obtiene el primer item de la lista
     * @return Devuelve la Clase al principio de la lista
     * null en caso que la lista este vacía
```

```
     */
    @Override
    public T getFirst() {
        return listSize > 0 ? array[0] : null;
    }
    /**
     * Obtiene el ultimo elemento de la lista
     * @return Devuelve la ultima Clase en la lista
     * null si la lista esta vacía
     */
    @Override
    public T getLast() {
        return listSize > 0 ? array[listSize - 1] : null;
    }
    /**
     * Devuelve el i - esimo elemento de la lista
     * @param i Posición del elemento a devolver (comienza en 0)
     * @return Devuelve la Clase en la posición i
     */
    @Override
    public T get(int i) {
        return listSize > 0 && i > 0 && i < listSize ? array[i] : null;
    }

    /**
     * Método que establece un item de la lista en una posición especifica
     * @param p Objeto que sera establecido en una posición especifica
     * @param i Posición a establecer el objeto
     */
    @Override
    public void set(T p, int i) {
        if(listSize > 0 && i > 0 && i < listSize) {
            array[i] = p;
        }
    }
    /**
     * Método que determina el tamaño de la lista
     * @return Devuelve un entero que indica el tamaño de la lista
     */
    @Override
    public int size() {
        return listSize;
    }
    /**
     * Método que inserta al final de la lista
     * @param p Objeto a insertar
     * @return Devuelve la propia lista enlazada
     */
    @Override
    public IList< T > addLast(T p) {
        if(listSize == array.length) {
            adjustCapacity(2 * listSize);
        }
        array[listSize] = p;
        listSize++;
        return this;
    }
    /**
     * Método que permite agregar un objeto en un posición
     * arbitraria de la lista
     * @param p Objeto que se quiere agregar a la lista
     * @param i posición en la cual se quiere agregar
     * @return Devuelve la propia lista enlazada
     */
    @Override
    public IList< T > add(T p, int i) {
        if(listSize == array.length) {
            adjustCapacity(2 * listSize);
        }
        for(int j = listSize - 1; j >= i; j--) {
            array[j + 1] = array[j];
```

```
        }
        array[i] = p;
        listSize++;
        return this;
    }
    /**
     * Método que elimina un elemento dado un objeto
     * @param p Objeto a eliminar
     */
    @Override
    public void remove(T p) {
        for(int i = 0; i < listSize; i++) {
            if(array[i].equals(p)) {
                remove(i);
                break;
            }
        }
    }
    /**
     * Método que remueve un elemento de la lista
     * @param i Posición o índice a eliminar de la lista
     */
    @Override

    public void remove(int i) {
        for(int j = i; j < listSize; j++) {
            array[j] = array[j + 1];
        }
        listSize--;
    }
    /**
     * Devuelve si esta o no
     * @param p Objeto a verificar si esta en la lista
     * @return Devuelve true si p esta en la lista false sino
     */
    @Override
    public boolean contains(T p) {
        for(int i = 0; i < listSize; i++) {
            if(array[i].equals(p)) {
                return true;
            }
        }
        return false;
    }
    /**
     * Implementación de la interface iterable que permite recorrer la lista
     * con ciclos estilo for - each
     * @return Devuelve el iterador para recorrer la lista
     */
    @Override
    public Iterator< T > iterator() {
        // Creación de un objeto anónimo para retornarlo
        return new Iterator< T >() {
            /** Contador para referenciar el elemento que actualmente se itera */
            private int i = 0;
            /**
             * Método que indica si existen mas elementos a recorrer o no
             * @return True si existen mas elementos para recorrer y false en caso contrario
             */
            public boolean hasNext() {
                return i < listSize;
            }
            /**
             * Devuelve el elemento donde se encuentra parado el iterador y lo avanza
             * @return
             */
            public T next() {
                i++;
                return array[i - 1];
            }
            /**
```

```
             * Método que le permite al iterador remover un elemento de forma segura
             * En la lista hay que tener cuidado cuando se remueve mientras se itera puesto
             * que la eliminación cambia el tamaño de la lista. Este método
             * no se encuentra soportado en esta version.
             */
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
    /**
     * Método privado utilitario encargado de asegurar una capacidad en el arreglo
     * haciendo una copia de los elementos del arreglo viejo en el nuevo
     * @param newCapacity Nueva capacidad del arreglo
     */
    private void adjustCapacity(int newCapacity) {
        T newArray[] = (T[]) new Object[newCapacity];
        for(int i = 0; i < listSize; i++) {
            newArray[i] = array[i];
            array[i] = null;
        }
        array = newArray;


    }
}
```

La siguiente imagen muestra el diagrama correspondiente a la lista implementada con arreglos.

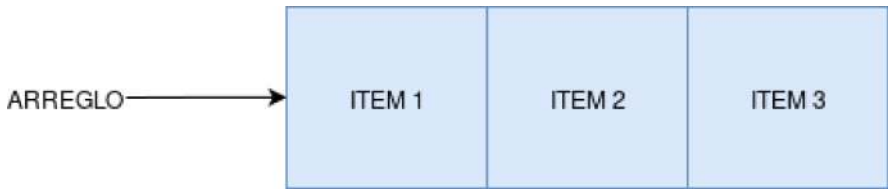

Figura 5. *Lista implementada con arreglos*

Observe que el tamaño del arreglo, es decir, su capacidad puede verse incrementada en la operación add, en la cual se puede duplicar el tamaño del arreglo; una practica común es reducir también el tamaño del arreglo cuando se remueve, y es común disminuir el tamaño del arreglo a un cuarto del mismo si la mitad del arreglo esta libre. Es importante con lo anterior evitar que se hagan secuencias de operaciones que dupliquen – recorren el tamaño del arreglo, es decir add – remove – add – remove y así sucesivamente decrementando sustancialmente el rendimiento del arreglo.