# Diferencia entre ArrayList y LinkedList en Java: código y rendimiento

## Introducción

Las listas son algunas de las estructuras de datos más utilizadas. En Java, una pregunta común al usar un List la implementación es:

¿Qué implementación utilizo?

¿Debería elegir un ArrayList o un LinkedList? ¿Cuál es la diferencia entre estos dos?

En este artículo, revisaremos estas dos implementaciones, observaremos su funcionamiento interno y discutiremos su desempeño. Saber qué implementación de un List usar en qué situación es una habilidad esencial.

## Descripción general de listas en Java

Las listas son estructuras de datos que se utilizan para el almacenamiento secuencial de elementos. Esto significa que cada elemento de la lista tiene un predecesor y un sucesor (excepto el primero y el último, por supuesto, solo tienen uno de cada uno).

Por tanto, las listas son colecciones ordenadas (a diferencia de los conjuntos) que también permiten duplicados. Son convenientes porque permiten una fácil manipulación de elementos (como inserción o recuperación) y una iteración simple de toda la colección.

Lists a menudo van de la mano con otros mecanismos, como Java Streams, que ofrecen formas simples pero efectivas de iteración, filtrado, mapeo y otras operaciones útiles.

En Java, List es una interfaz bajo el java.util paquete. Dado que es una interfaz, simplemente proporciona una lista de métodos que deben anularse en la clase de implementación real.

ArrayList y LinkedList son dos implementaciones diferentes de estos métodos. sin embargo, el LinkedList también implementa el Queue interfaz.

## Funcionamiento interno de ArrayList y LinkedList

Un ArrayList es una matriz de tamaño variable que crece a medida que se agregan elementos adicionales.

* [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) por otro lado, permite un acceso de lectura aleatorio rápido, para que puedas agarrar cualquier elemento en tiempo constante. Pero agregar o eliminar desde cualquier lugar menos el final requiere desplazar todos los últimos elementos, ya sea para abrir o llenar el espacio. Además, si agrega más elementos que la capacidad de la matriz subyacente, se asigna una nueva matriz (1.5 veces el tamaño), y la matriz anterior se copia a la nueva, por lo que agregar a un ArrayList es O (n) en el peor caso pero constante en promedio.

Un LinkedList es una implementación de lista / cola doblemente enlazada.

* [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) permite inserciones o eliminaciones de tiempo constante utilizando iteradores , pero solo acceso secuencial de elementos. En otras palabras, puede recorrer la lista hacia adelante o hacia atrás, pero encontrar un puesto en la lista lleva tiempo proporcional al tamaño de la lista. Javadoc dice que "las operaciones que indexan en la lista atravesarán la lista desde el principio o el final, lo que esté más cerca" , por lo que esos métodos son O (n) ( n/4 pasos) en promedio, aunque O (1) para index = 0.

Esto significa que ArrayList Contiene internamente una matriz de valores y una variable de contador para conocer el tamaño actual en cualquier punto. Si se agrega un elemento, se aumenta el tamaño. Si se elimina un elemento, se reduce el tamaño.

LinkedList no tiene una matriz, sino una cola de dos extremos de elementos conectados entre sí. El primer elemento apunta al segundo, que apunta al tercero, y así sucesivamente. Dado que esta es una lista doblemente enlazada, cada elemento también apunta a su predecesor. El quinto elemento, por ejemplo, apunta tanto al cuarto como al sexto elemento.

ArrayList contiene una única matriz para el almacenamiento de datos. LinkedList necesita una estructura de datos personalizada. Esta estructura de datos personalizada es una Node. Es una pequeña clase interna que sirve como envoltorio para cada elemento.

Para almacenar el elemento B, no es suficiente almacenar su valor como lo haría con un ArrayList.

También se necesita un puntero al elemento anterior y al siguiente para que la lista vinculada sea transitable. Por tanto, toda la estructura de la lista consta de nodes conectados entre sí. Cada node contiene su elemento y dos punteros: un enlace al node anterior y el enlace al siguiente node. El primer node no tiene un node anterior y el último node no tiene un node siguiente.

Finalmente, en el caso de una lista enlazada, podemos asumir la existencia de dos punteros que monitorean continuamente el primer y último elemento de la lista. El primer puntero, head, apunta al primer elemento y se actualiza cada vez que se inserta un nuevo elemento al principio. El segundo puntero, tail, apunta al último elemento y también se actualiza cada vez que se agrega un nuevo elemento al final.

# Comparación de implementaciones de ArrayList y LinkedList

#### Obteniendo elementos con get ()

## ArrayList.get ()

Si uno desea obtener un elemento de un ArrayList utilizando el get(int index) método, la implementación podría simplemente delegar esta tarea a su matriz interna:

public E get(int index) {

rangeCheck(index);

return elementData(index);

}

Por supuesto, se realiza una verificación adicional en el índice dado (asegurándose de que no sea menor que cero o mayor que el tamaño de la matriz).

Podemos ver que esta operación se realiza en tiempo constante, o O (1). Esto significa que no importa el tamaño de la matriz, cualquier elemento solicitado se devolverá instantáneamente, sin necesidad de recorrer la lista. Esto se debe a que toda la matriz se almacena en un lugar único en la memoria.

La ranura para el segundo elemento está ubicada precisamente después del primero, y la ranura para el n-ésimo elemento está ubicada precisamente antes del n + 1-ésimo. Basándose en esta estructura interna, cualquier elemento se puede recuperar fácilmente por índice.

## LinkedList.get ()

Si uno desea obtener un elemento de un LinkedList, utilizando el get(int index) método – puedes, pero es realmente ineficiente.

Anteriormente mencionamos que una lista vinculada no existe en un solo lugar en la memoria, sino que contiene diferentes nodes conectados entre sí. Para obtener un elemento, es necesario recorrer la lista desde el principio (o el final, lo que esté más cerca) y seguir las conexiones de cada uno de los nodes hasta encontrar el elemento deseado.

La implementación del mismo método se ve así:

public E get(int index) {

checkElementIndex(index);

return node(index).item;

}

private void checkElementIndex(int index) {

if (!isElementIndex(index))

throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

}

private boolean isElementIndex(int index) {

return index >= 0 && index < size;

}

Node<E> node(int index) {

if (index < (size >> 1)) {

Node<E> x = first;

for (int i = 0; i < index; i++)

x = x.next;

return x;

} else {

Node<E> x = last;

for (int i = size - 1; i > index; i--)

x = x.prev;

return x;

}

}

Primero, se realiza una verificación para asegurarse de que el índice no 0 o por encima del tamaño del LinkedList. Entonces la node() El método atraviesa la lista hasta que encuentra el que estamos buscando.

Esto se hace en tiempo O (N), comparado con ArrayListtiempo de O (1).

#### Insertar elementos con add ()

Esencialmente, cualquier tipo de inserción se puede generalizar e implementar utilizando un método común: insertar en un índice determinado.

Si es necesario insertar un elemento al principio, se puede llamar al método con un índice de 0. Si es necesario insertar un elemento al final, el índice corresponderá al tamaño actual de la lista. Si es necesario insertar un elemento en algún lugar intermedio, el usuario debe proporcionar este índice.

##### ArrayList.add ()

Insertar un elemento al final es bastante simple, especialmente para una estructura como un ArrayList. Simplemente extienda la longitud en uno e inserte el elemento al final:

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

Sin embargo, insertar en una posición determinada es un poco más complicado. Debe dividir la matriz en el lugar que desea insertar: copie todo después de ese punto y muévalo hacia la derecha, agregando el nuevo elemento en el índice:

public void add(int index, E element) {

rangeCheckForAdd(index);

ensureCapacityInternal(size + 1); // Increments modCount!!

System.arraycopy(elementData, index, elementData, index + 1, size - index);

elementData[index] = element;

size++;

}

Cuanto más grande sea la parte copiada, más lenta será esta operación. Esto hace que la adición de elementos a un ArrayList una operación relativamente ineficiente. Sin embargo, llegar al punto en el que debe realizarse la inserción es realmente eficaz.

##### LinkedList.add ()

LinkedListLa implementación de nos permite agregar elementos en cualquier índice dado, con bastante facilidad. Solo apunta el head y tail punteros de los elementos anteriores y posteriores al nuevo, respectivamente. Si está insertando al principio o al final de la lista, solo es necesario actualizar un puntero.

Echemos un vistazo a la implementación:

public boolean add(E e) {

linkLast(e);

return true;

}

Public void linkLast(E e) {

final Node<E> l = last;

final Node<E> newNode = new Node<>(l, e, null);

last = newNode;

if (l == null)

first = newNode;

else

l.next = newNode;

size++;

modCount++;

}

Alternativamente, si especificamos un índice, ambos linkLast() y linkBefore() ser llamado:

public void add(int index, E element) {

checkPositionIndex(index);

if (index == size)

linkLast(element);

else

linkBefore(element, node(index));

}

void linkBefore(E e, Node<E> succ) {

// assert succ != null;

final Node<E> pred = succ.prev;

final Node<E> newNode = new Node<>(pred, e, succ);

succ.prev = newNode;

if (pred == null)

first = newNode;

else

pred.next = newNode;

size++;

modCount++;

}

No importa cuán grande sea la lista, solo es necesario cambiar dos punteros. Esto hace que la adición de elementos a un LinkedList una operación altamente eficiente. Sin embargo, llegar a la posición en la que se debe insertar el elemento es ineficaz.

## Encontrar elementos con indexOf ()

Encontrar un elemento de una lista, ya sea un ArrayList o un LinkedList debería ser bastante similar. Esto se debe a que no hay forma de saber a priori dónde se almacena un elemento en particular, a menos que la matriz esté ordenada y distribuida uniformemente.

Una lista simplemente realiza un seguimiento de sus elementos y ofrece formas de manipularlos. Para saber exactamente dónde está cada uno de estos elementos, ambas implementaciones deben pasar por algún tipo de proceso iterativo hasta que se encuentra el elemento.

## ArrayList.indexOf ()

En el ArrayList implementación, esto se hace con un simple for bucle que va desde 0 a size-1 y verificar si el elemento en el índice actual coincide con el valor dado:

public int indexOf(Object o) {

if (o == null) {

for (int i = 0; i < size; i++)

if (elementData[i]==null)

return i;

} else {

for (int i = 0; i < size; i++)

if (o.equals(elementData[i]))

return i;

}

return -1;

}

Esta es literalmente una búsqueda lineal, que no es muy eficiente, pero en realidad es la única forma de buscar un elemento en una colección barajada (si ignoramos los algoritmos metaheurísticos y las aproximaciones).

## LinkedList.indexOf ()

LinkedList hace esto un poco diferente. En lugar de iterar a través de una matriz, tiene que recorrer la lista saltando de un elemento al siguiente con el uso de punteros. En última instancia, el resultado es el mismo: visitando cada elemento, uno por uno, hasta encontrar el buscado:

public int indexOf(Object o) {

int index = 0;

if (o == null) {

for (Node<E> x = first; x != null; x = x.next) {

if (x.item == null)

return index;

index++;

}

} else {

for (Node<E> x = first; x != null; x = x.next) {

if (o.equals(x.item))

return index;

index++;

}

}

return -1;

}

#### Eliminar elementos con remove ()

## ArrayList.remove ()

Muy similar a agregar elementos en un índice dado, eliminarlos requiere un ArrayList para copiar una parte de sí mismo y reinicializar la matriz sin un valor, desplazando la parte copiada hacia la izquierda:

public E remove(int index) {

rangeCheck(index);

modCount++;

E oldValue = elementData(index);

int numMoved = size - index - 1;

if (numMoved > 0)

System.arraycopy(elementData, index+1, elementData, index, numMoved);

elementData[--size] = null; // clear to let GC do its work

return oldValue;

}

Cuanto más grande sea la parte copiada, más lenta será esta operación. Nuevamente, esto hace que la eliminación de elementos de un ArrayList una operación ineficiente. Aunque, algo bueno sobre ArrayLists es que puedes llegar a ese elemento muy fácilmente. elementData(index) devuelve el elemento que desea eliminar en O (1) tiempo.

## LinkedList.remove ()

Eliminar un elemento de un LinkedList funciona desvinculando los punteros anteriores y posteriores del elemento que nos gustaría eliminar. Después de eso, el elemento anterior está vinculado al siguiente en la línea. De esta forma, el elemento antiguo queda “varado” y sin referencias a él, el GC se encarga de ello:

public boolean remove(Object o) {

if (o == null) {

for (Node<E> x = first; x != null; x = x.next) {

if (x.item == null) {

unlink(x);

return true;

}

}

} else {

for (Node<E> x = first; x != null; x = x.next) {

if (o.equals(x.item)) {

unlink(x);

return true;

}

}

}

return false;

}

Esto hace que la operación de eliminar elementos de un LinkedList eficiente, ya que de nuevo, solo es necesario cambiar algunos puntos. Sin embargo, cuanto más larga sea la lista, más tardará en llegar al elemento que debe eliminarse, ya que no podemos acceder a los elementos a través de su índice.

## Comparación de rendimiento

Hasta ahora, hemos discutido cómo ArrayList y LinkedList trabajar bajo el capó. Hemos analizado cada uno de ellos para comprender mejor sus similitudes y, lo que es más importante, las diferencias.

En esta sección, compararemos brevemente las dos implementaciones desde la perspectiva del rendimiento:

## Comparando get ()

Podemos ver que la obtención de elementos de una lista es siempre O (1) para ArrayList.

por LinkedList, buscar el primer o el último elemento es O (1) porque siempre tiene punteros a estos dos. No hay necesidad de lógica transversal adicional. Sin embargo, buscar cualquier otro elemento es O (N) porque no podemos acceder a ellos a través de un índice.

Por lo tanto, en general, si recupera muchos elementos de la lista, una ArrayList se prefiere.

## Comparando insert ()

por ArrayList, la inserción es O (1) solo si se agrega al final. En todos los demás casos (agregando al principio o en el medio), la complejidad es O (N), porque la parte derecha de la matriz debe copiarse y desplazarse.

La complejidad de un LinkedList será O (1) tanto para la inserción al principio como al final. Una vez más, esto se debe a la head y tail punteros que se pueden utilizar para insertar un elemento en cualquiera de estas dos posiciones instantáneamente.

LinkedListLa complejidad de insertar en el medio es O (N), lo mismo que para ArrayList. La operación de inserción es realmente eficiente, pero para llegar a ese punto hay que atravesar todos los elementos anteriores.

Por lo general, la inserción de elementos se realiza por igual entre ArrayList y un LinkedList, a menos que esté trabajando principalmente con el primer y último elemento.

## Comparando remove ()

Las complejidades de la eliminación son prácticamente las mismas que las de la inserción. ArrayLists eliminará elementos en O (1) si están al final – O (N) en todos los demás casos.

LinkedLists tienen complejidad O (1) para eliminar desde el principio o el final, y O (N) en otros casos.

Por lo tanto, la eliminación de elementos es generalmente la misma, a menos que trabaje principalmente con los elementos iniciales y últimos.

## Conclusión

ArrayList y LinkedList son dos implementaciones diferentes del List interfaz. Tienen sus diferencias que es importante comprender para utilizarlas correctamente.

* LinkedList lo implementa con una lista doblemente vinculada.
* ArrayList lo implementa con una matriz de redimensionamiento dinámico.

La implementación que se debe utilizar depende de los casos de uso exactos. Si los elementos se van a buscar con frecuencia, tiene poco sentido usar LinkedList ya que la recuperación es más lenta en comparación con ArrayList. Por otro lado, si se necesitan inserciones de tiempo constante o si el tamaño total se desconoce de antemano, entonces LinkedList se prefiere.

# **¿Cuál es la diferencia entre las siguientes implementaciones?**

ArrayList<String> **nombreArray** = new ArrayList<>();

List<String**> nombreLista** = new ArrayList<>();

**List** es una colección, y una colección puede ser interfaces y clase abstracta que nos permite identificar los objetos independientemente de la implementación. Es decir, son genéricas.

Mientras, un **ArrayList** es contenedor que contiene una implementación de la colección List.

En base a esto, ***List*** es una interfaz genérica que representa una colección ordenada de elementos que pueden repetirse.

Mientras, dos listas de propósito general serían las clases ***LinkedList*** y ***ArrayList*** y de propósito específico, ***Vector*** y ***CopyOnWriteArrayList***.

En tu duda, la ***ArrayList*** es un array que se maneja como una clase y no tiene tamaño fijo. Es eficiente cuando se realizan muchos acceso y en este caso los elementos se añaden al final. Permite que se puedan eliminar elementos intermedios, pero eso provocará un desplazamiento de índices.

En cambio, para una clase ***LinkedList*** tendremos una lista enlazada en la que los elementos se añaden en cualquier parte de la lista muy fácilmente. Aquí, para encontrar un elemento hay que recorrer la lista. Es eficiente cuando el tamaño fluctúa y sobre todo en posiciones centrales.

¿Qué sucede si utilizas ***List*** como lo haces en tu caso?, que podrías cambiar el tipo de ***nombreLista*** a ***LinkedList*** en su definición y todo sería compatible 100% y fácilmente.

No obstante, en el primer caso de ***nombreArray*** tendrías que cambiar en todos los sitios el tipo debido a que estas implementando un ***ArrayList*** y podrías tener problemas si lo cambias a ***LinkedLIst***. Por eso se recomienda el uso de ***List***, como lo haces de ***nombreLista.***