As the name implies, the Java class LinkedList is called LinkedList because internally it is based on a **Doubly Linked List**.

**Difference between LinkedList and java.util.LinkedList**

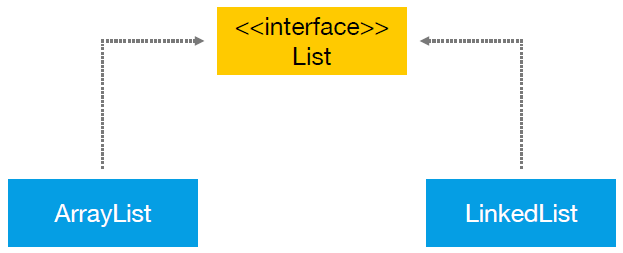
So what is the difference between the LinkedList data structure and the class java.util.LinkedList?

As an analogy, think of the abstract concept of a car and a concrete car.

The [Linked List data structure](https://marcus-biel.com/linkedlist-data-structure/) is an abstract concept, independent of any specific programming language. The LinkedList Java class is a concrete implementation of this abstract concept.

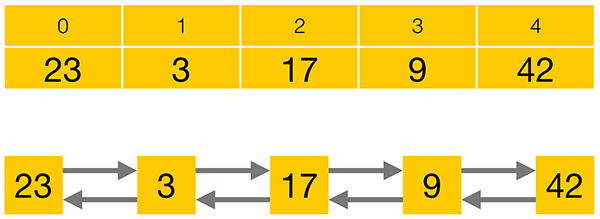
So in this article, I will focus on one specific Linked List implementation, the java.util.LinkedList class. Among other interfaces, LinkedList implements the java.util.List interface. You can have duplicate elements in a List and you can go from element to element in the same order as the elements were inserted.

**Difference between ArrayList and LinkedList**



As you can see, both classes implement the List interface which makes them somewhat similar. So what’s the difference between ArrayList and LinkedList?

First of all, ArrayList is based on an Array data structure, while LinkedList is based on a Doubly Linked List data structure:



Compared to a LinkedList, storing elements in an ArrayList consumes less memory and generally gives faster access times. Adding or removing elements is usually faster for a LinkedList, but as you usually have to iterate to the position at which you want to add or remove an element, the performance loss for iterating to the correct position often prevails over the performance gain in adding or removing an element.  Besides the different data structures of ArrayList and LinkedList, LinkedList also implements the Queue and the Deque interfaces which give it some additional functionality over ArrayList.

In conclusion, there is no overall winner between ArrayList and LinkedList. Your specific requirements will determine which class to use.

**LinkedList Implementation**

Let’s put ArrayList aside for now and have an in-depth look at the LinkedList implementation. Here is a simplified code excerpt from the java.util.LinkedList class:

package java.util;

public class LinkedList implements List,Deque {

private Node first;

private Node last;

public E get(int index) {…}

public boolean add(E e) {…}

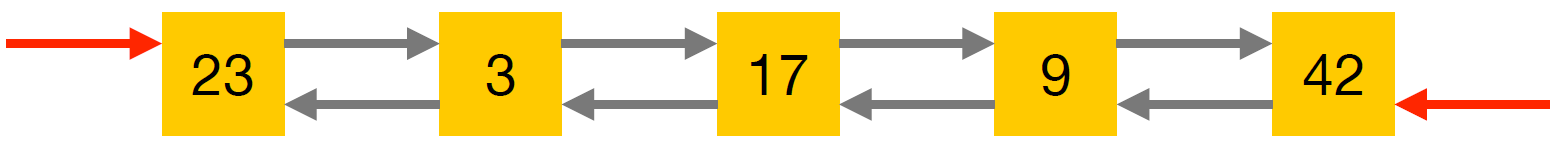
public E remove(int index) {…}

[…]

}

I don’t expect you to fully grasp every detail of the code, I just want to show you that LinkedList is a normal Java class which anyone could have written, given enough time and knowledge. The real source code is available online. After reading this article, I recommend that you take a look at it for yourself. Okay. So, as you can see, LinkedList implements the List, Queue and Deque interfaces, as Deque extends the Queue interface.

Next you can see that the LinkedList class has a reference to the first and the last elements of the list. Finally, you can see that the class has functions like get, add, or remove – to access, insert or delete elements from the list.

So the LinkedList class has a reference to the first and last elements of the list, shown as red arrows in this image below:

Every single element in a Doubly Linked List has a reference to its previous and next elements as well as a reference to an item, simplified as a number within a yellow box on this image above.

public class Node {

private E item;

private Node previous;

private Node next;

public Node(E element, Node previous, Node next) {

this.item = element;

this.next = next;

this.previous = previous;

}

[...]

}

Here you see a code excerpt of a Node. It has private members for the item it holds, and for the previous and next Node in the list. As a user of the Collections class LinkedList, you never directly access the Nodes. Instead, you use the public methods of the LinkedList class that internally operate on the private Node members.

In my tutorial about [ArrayList](http://www.marcus-biel.com/arraylist/), I wrote about the List interface methods already. In this article I want to look at the methods of the Queue and Deque interface as implemented by LinkedList.

**java.util.Queue interface**

From a high-level perspective, the Queue interface consists of three simple operations:

* **add an element** to the end of the Queue
* **retrieve an element** from the front of the Queue, without removing it
* **Retrieve and remove an element** from the front of the Queue.

In the lifetime of a Queue, there are special situations, like trying to remove an element from an empty Queue or trying to add an element to a Queue that has a limited capacity and is currently full.

Depending on your specific implementation, this might be an expected situation and you need a method that returns null or false in this case. Alternatively, this might be an unexpected situation and you need a method that throws an Exception in this case. Therefore, the Queue interface offers each of its operations in two flavours –**one method that will throw an Exception, and one that will return a special value in certain cases**:



Okay, let’s look at this in more detail.

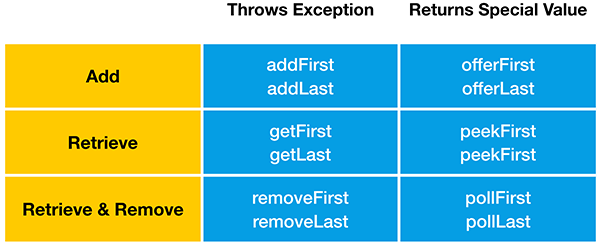
A Queue allows to add elements to the end of the Queue.

“**add**” will throw an Exception when the Queue is full, while “**offer**” will return false in this case. LinkedList, like most Queue implementations, has an unlimited capacity, so it will never be full. **ArrayBlockingQueue**, on the other hand, is a Queue implementation that has a limited capacity.

Next, “**element**” and “**peek**” allow you to retrieve an element from the front of the Queue, without removing it. If the Queue is empty, the element function will throw an Exception, while peek() will return false. Finally, you can retrieve and remove an element from the front of the Queue. If the Queue is empty, remove will throw an Exception, while poll will return false.

**java.util.Deque interface**

Okay, now we will look at some methods of the Deque interface as implemented by LinkedList. Deque is the short form of “Double Ended Queue”, so it is a Queue that can be accessed from either end. Just like a Queue, a Deque allows **adding, retrieving and – retrieving and removing** – an element. But as it can be accessed from either end, the Queue methods we saw before now exist in two variations – one for the first and one for the last element in the Deque:



Again, let’s look at this in more detail.You can add elements to both ends of the Deque. Just like the add method of the Queue interface, **addFirst**and **addLast**will throw an Exception when the Deque is full.

“**offerFirst**” and “**offerLast**” will return false instead of throwing an Exception. Please keep in mind that LinkedList has an unlimited capacity, so it will never be full. LinkedBlockingDeque, on the other hand, is a Deque implementation that may have a limited capacity. Okay, let’s go on.

You can retrieve elements from both ends of the Deque, without removing them.  
“**getFirst**” and “**getLast**” will throw an Exception when the Queue is empty, while “**peekFirst**” and “**peekLast**” will return false in this case. Finally, you can retrieve and remove elements from both ends of the Deque. “**removeFirst**” and “**removeLast**” will throw an Exception when the Queue is empty, while “**pollFirst”**and **“pollLast”**will return false in this case.

**Stack data structure**

Now on to a completely different topic. The Deque interface also supports the methods of the Stack data structure, “**push**” “**peek**” and “**pop**”. Therefore java.util.LinkedList can also be used as Stack.

A Stack is a very simple data structure that can only be accessed from the top. As an analogy, think of a stack of books:

“**push**” adds an element to the top of the Stack. It is equivalent to the “addFirst” method. “**peek**” retrieves but does not remove an element from the top of the Stack. It is equivalent to the “peekFirst” method. “**pop**” retrieves and removes an element from the top of the Stack. It is equivalent to the “removeFirst” method.

The two functions come from two different interfaces that PriorityQueue implements:

* add() comes from Collection.
* offer() comes from Queue.

For a capacity-constrained queue, the difference is that add() always returns true and throws an exception if it can't add the element, whereas offer() is allowed to return false if it can't add the element.

However, this doesn't apply to PriorityQueue; the two functions are synonymous.