# Implementing a Linked List in Java using Class

**Pre-requisite:** [Linked List Data Structure](https://www.geeksforgeeks.org/data-structures/linked-list/)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at the contiguous location, the elements are linked using pointers as shown below.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2013/03/Linkedlist.png)

In Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

filter\_none

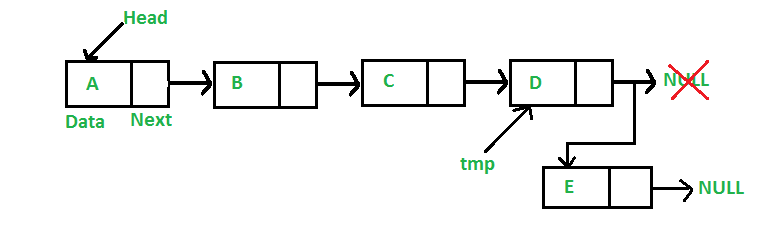
brightness\_4

|  |
| --- |
| class LinkedList {      Node head; // head of list        /\* Linked list Node\*/      class Node {          int data;          Node next;            // Constructor to create a new node          // Next is by default initialized          // as null          Node(int d) { data = d; }      }  } |

### Creation and Insertion

In this article, insertion in the list is done at the end, that is the new node is added after the last node of the given Linked List. For example, if the given Linked List is 5->10->15->20->25 and 30 is to be inserted, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head pointer of it, it is required to traverse the list till the last node and then change the next of last node to new node.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2013/03/Linkedlist_insert_last.png)

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| import java.io.\*;    // Java program to implement  // a Singly Linked List  public class LinkedList {        Node head; // head of list        // Linked list Node.      // This inner class is made static      // so that main() can access it      static class Node {            int data;          Node next;            // Constructor          Node(int d)          {              data = d;              next = null;          }      }        // Method to insert a new node      public static LinkedList insert(LinkedList list, int data)      {          // Create a new node with given data          Node new\_node = new Node(data);          new\_node.next = null;            // If the Linked List is empty,          // then make the new node as head          if (list.head == null) {              list.head = new\_node;          }          else {              // Else traverse till the last node              // and insert the new\_node there              Node last = list.head;              while (last.next != null) {                  last = last.next;              }                // Insert the new\_node at last node              last.next = new\_node;          }            // Return the list by head          return list;      }        // Method to print the LinkedList.      public static void printList(LinkedList list)      {          Node currNode = list.head;            System.out.print("LinkedList: ");            // Traverse through the LinkedList          while (currNode != null) {              // Print the data at current node              System.out.print(currNode.data + " ");                // Go to next node              currNode = currNode.next;          }      }        // Driver code      public static void main(String[] args)      {          /\* Start with the empty list. \*/          LinkedList list = new LinkedList();            //          // \*\*\*\*\*\*INSERTION\*\*\*\*\*\*          //            // Insert the values          list = insert(list, 1);          list = insert(list, 2);          list = insert(list, 3);          list = insert(list, 4);          list = insert(list, 5);          list = insert(list, 6);          list = insert(list, 7);          list = insert(list, 8);            // Print the LinkedList          printList(list);      }  } |

**Output:**

LinkedList: 1 2 3 4 5 6 7 8

### Traversal

For traversal, below is a general purpose function printList() that prints any given list by traversing the list from head node to the last.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| import java.io.\*;    // Java program to implement  // a Singly Linked List  public class LinkedList {        Node head; // head of list        // Linked list Node.      // This inner class is made static      // so that main() can access it      static class Node {            int data;          Node next;            // Constructor          Node(int d)          {              data = d;              next = null;          }      }        // Method to insert a new node      public static LinkedList insert(LinkedList list, int data)      {          // Create a new node with given data          Node new\_node = new Node(data);          new\_node.next = null;            // If the Linked List is empty,          // then make the new node as head          if (list.head == null) {              list.head = new\_node;          }          else {              // Else traverse till the last node              // and insert the new\_node there              Node last = list.head;              while (last.next != null) {                  last = last.next;              }                // Insert the new\_node at last node              last.next = new\_node;          }            // Return the list by head          return list;      }        // Method to print the LinkedList.      public static void printList(LinkedList list)      {          Node currNode = list.head;            System.out.print("LinkedList: ");            // Traverse through the LinkedList          while (currNode != null) {              // Print the data at current node              System.out.print(currNode.data + " ");                // Go to next node              currNode = currNode.next;          }      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*MAIN METHOD\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // method to create a Singly linked list with n nodes      public static void main(String[] args)      {          /\* Start with the empty list. \*/          LinkedList list = new LinkedList();            //          // \*\*\*\*\*\*INSERTION\*\*\*\*\*\*          //            // Insert the values          list = insert(list, 1);          list = insert(list, 2);          list = insert(list, 3);          list = insert(list, 4);          list = insert(list, 5);          list = insert(list, 6);          list = insert(list, 7);          list = insert(list, 8);            // Print the LinkedList          printList(list);      }  } |

**Output:**

LinkedList: 1 2 3 4 5 6 7 8

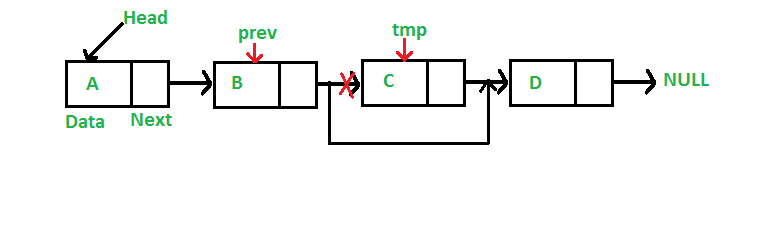
### Deletion By KEY

The deletion process can be understood as follows:

**To be done:**  
Given a ‘key’, delete the first occurrence of this key in linked list.

**How to do it:**  
To delete a node from linked list, do following steps.

1. Search the key for its first occurrence in the list
2. Now, Any of the 3 conditions can be there:
   * **Case 1: The key is found at head**
     1. In this case, Change the head of the node to the next node of current head.
     2. Free the memory of replaced head node.
   * **Case 2: The key is found at in the middle or last, except at head**
     1. In this case, Find previous node of the node to be deleted.
     2. Change the next of previous node to the next node of current node.
     3. Free the memory of replaced node.
   * **Case 3: The key is not found in the list**
     1. In this case, No operation needs to be done.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/05/Linkedlist_deletion.png)

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| import java.io.\*;    // Java program to implement  // a Singly Linked List  public class LinkedList {        Node head; // head of list        // Linked list Node.      // This inner class is made static      // so that main() can access it      static class Node {            int data;          Node next;            // Constructor          Node(int d)          {              data = d;              next = null;          }      }        // Method to insert a new node      public static LinkedList insert(LinkedList list, int data)      {          // Create a new node with given data          Node new\_node = new Node(data);          new\_node.next = null;            // If the Linked List is empty,          // then make the new node as head          if (list.head == null) {              list.head = new\_node;          }          else {              // Else traverse till the last node              // and insert the new\_node there              Node last = list.head;              while (last.next != null) {                  last = last.next;              }                // Insert the new\_node at last node              last.next = new\_node;          }            // Return the list by head          return list;      }        // Method to print the LinkedList.      public static void printList(LinkedList list)      {          Node currNode = list.head;            System.out.print("LinkedList: ");            // Traverse through the LinkedList          while (currNode != null) {              // Print the data at current node              System.out.print(currNode.data + " ");                // Go to next node              currNode = currNode.next;          }            System.out.println();      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*DELETION BY KEY\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // Method to delete a node in the LinkedList by KEY      public static LinkedList deleteByKey(LinkedList list, int key)      {          // Store head node          Node currNode = list.head, prev = null;            //          // CASE 1:          // If head node itself holds the key to be deleted            if (currNode != null && currNode.data == key) {              list.head = currNode.next; // Changed head                // Display the message              System.out.println(key + " found and deleted");                // Return the updated List              return list;          }            //          // CASE 2:          // If the key is somewhere other than at head          //            // Search for the key to be deleted,          // keep track of the previous node          // as it is needed to change currNode.next          while (currNode != null && currNode.data != key) {              // If currNode does not hold key              // continue to next node              prev = currNode;              currNode = currNode.next;          }            // If the key was present, it should be at currNode          // Therefore the currNode shall not be null          if (currNode != null) {              // Since the key is at currNode              // Unlink currNode from linked list              prev.next = currNode.next;                // Display the message              System.out.println(key + " found and deleted");          }            //          // CASE 3: The key is not present          //            // If key was not present in linked list          // currNode should be null          if (currNode == null) {              // Display the message              System.out.println(key + " not found");          }            // return the List          return list;      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*MAIN METHOD\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // method to create a Singly linked list with n nodes      public static void main(String[] args)      {          /\* Start with the empty list. \*/          LinkedList list = new LinkedList();            //          // \*\*\*\*\*\*INSERTION\*\*\*\*\*\*          //            // Insert the values          list = insert(list, 1);          list = insert(list, 2);          list = insert(list, 3);          list = insert(list, 4);          list = insert(list, 5);          list = insert(list, 6);          list = insert(list, 7);          list = insert(list, 8);            // Print the LinkedList          printList(list);            //          // \*\*\*\*\*\*DELETION BY KEY\*\*\*\*\*\*          //            // Delete node with value 1          // In this case the key is \*\*\*at head\*\*\*          deleteByKey(list, 1);            // Print the LinkedList          printList(list);            // Delete node with value 4          // In this case the key is present \*\*\*in the middle\*\*\*          deleteByKey(list, 4);            // Print the LinkedList          printList(list);            // Delete node with value 10          // In this case the key is \*\*\*not present\*\*\*          deleteByKey(list, 10);            // Print the LinkedList          printList(list);      }  } |

**Output:**

LinkedList: 1 2 3 4 5 6 7 8

1 found and deleted

LinkedList: 2 3 4 5 6 7 8

4 found and deleted

LinkedList: 2 3 5 6 7 8

10 not found

LinkedList: 2 3 5 6 7 8

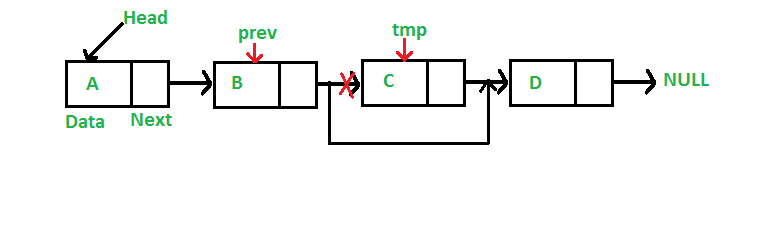
### Deletion At Position

This deletion process can be understood as follows:

**To be done:**  
Given a **‘position’**, delete the node at this position from the linked list.

**How to do it:**  
The steps to do it are as follows:

1. Traverse the list by counting the index of the nodes
2. For each index, match the index to be same as position
3. Now, Any of the 3 conditions can be there:
   * **Case 1: The position is 0, i.e. the head is to be deleted**
     1. In this case, Change the head of the node to the next node of current head.
     2. Free the memory of replaced head node.
   * **Case 2: The position is greater than 0 but less than the size of the list, i.e. in the middle or last, except at head**
     1. In this case, Find previous node of the node to be deleted.
     2. Change the next of previous node to the next node of current node.
     3. Free the memory of replaced node.
   * **Case 3: The position is greater than the size of the list, i.e. position not found in the list**
     1. In this case, No operation needs to be done.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/05/Linkedlist_deletion.png)

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| import java.io.\*;    // Java program to implement  // a Singly Linked List  public class LinkedList {        Node head; // head of list        // Linked list Node.      // This inner class is made static      // so that main() can access it      static class Node {            int data;          Node next;            // Constructor          Node(int d)          {              data = d;              next = null;          }      }        // Method to insert a new node      public static LinkedList insert(LinkedList list, int data)      {          // Create a new node with given data          Node new\_node = new Node(data);          new\_node.next = null;            // If the Linked List is empty,          // then make the new node as head          if (list.head == null) {              list.head = new\_node;          }          else {              // Else traverse till the last node              // and insert the new\_node there              Node last = list.head;              while (last.next != null) {                  last = last.next;              }                // Insert the new\_node at last node              last.next = new\_node;          }            // Return the list by head          return list;      }        // Method to print the LinkedList.      public static void printList(LinkedList list)      {          Node currNode = list.head;            System.out.print("LinkedList: ");            // Traverse through the LinkedList          while (currNode != null) {              // Print the data at current node              System.out.print(currNode.data + " ");                // Go to next node              currNode = currNode.next;          }            System.out.println();      }        // Method to delete a node in the LinkedList by POSITION      public static LinkedList deleteAtPosition(LinkedList list, int index)      {          // Store head node          Node currNode = list.head, prev = null;            //          // CASE 1:          // If index is 0, then head node itself is to be deleted            if (index == 0 && currNode != null) {              list.head = currNode.next; // Changed head                // Display the message              System.out.println(index + " position element deleted");                // Return the updated List              return list;          }            //          // CASE 2:          // If the index is greater than 0 but less than the size of LinkedList          //          // The counter          int counter = 0;            // Count for the index to be deleted,          // keep track of the previous node          // as it is needed to change currNode.next          while (currNode != null) {                if (counter == index) {                  // Since the currNode is the required position                  // Unlink currNode from linked list                  prev.next = currNode.next;                    // Display the message                  System.out.println(index + " position element deleted");                  break;              }              else {                  // If current position is not the index                  // continue to next node                  prev = currNode;                  currNode = currNode.next;                  counter++;              }          }            // If the position element was found, it should be at currNode          // Therefore the currNode shall not be null          //          // CASE 3: The index is greater than the size of the LinkedList          //          // In this case, the currNode should be null          if (currNode == null) {              // Display the message              System.out.println(index + " position element not found");          }            // return the List          return list;      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*MAIN METHOD\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // method to create a Singly linked list with n nodes      public static void main(String[] args)      {          /\* Start with the empty list. \*/          LinkedList list = new LinkedList();            //          // \*\*\*\*\*\*INSERTION\*\*\*\*\*\*          //            // Insert the values          list = insert(list, 1);          list = insert(list, 2);          list = insert(list, 3);          list = insert(list, 4);          list = insert(list, 5);          list = insert(list, 6);          list = insert(list, 7);          list = insert(list, 8);            // Print the LinkedList          printList(list);            //          // \*\*\*\*\*\*DELETION AT POSITION\*\*\*\*\*\*          //            // Delete node at position 0          // In this case the key is \*\*\*at head\*\*\*          deleteAtPosition(list, 0);            // Print the LinkedList          printList(list);            // Delete node at position 2          // In this case the key is present \*\*\*in the middle\*\*\*          deleteAtPosition(list, 2);            // Print the LinkedList          printList(list);            // Delete node at position 10          // In this case the key is \*\*\*not present\*\*\*          deleteAtPosition(list, 10);            // Print the LinkedList          printList(list);      }  } |

**Output:**

LinkedList: 1 2 3 4 5 6 7 8

0 position element deleted

LinkedList: 2 3 4 5 6 7 8

2 position element deleted

LinkedList: 2 3 5 6 7 8

10 position element not found

LinkedList: 2 3 5 6 7 8

### All Operations

Below is the complete program that applies each operations together:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| import java.io.\*;    // Java program to implement  // a Singly Linked List  public class LinkedList {        Node head; // head of list        // Linked list Node.      // This inner class is made static      // so that main() can access it      static class Node {            int data;          Node next;            // Constructor          Node(int d)          {              data = d;              next = null;          }      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*INSERTION\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // Method to insert a new node      public static LinkedList insert(LinkedList list, int data)      {          // Create a new node with given data          Node new\_node = new Node(data);          new\_node.next = null;            // If the Linked List is empty,          // then make the new node as head          if (list.head == null) {              list.head = new\_node;          }          else {              // Else traverse till the last node              // and insert the new\_node there              Node last = list.head;              while (last.next != null) {                  last = last.next;              }                // Insert the new\_node at last node              last.next = new\_node;          }            // Return the list by head          return list;      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*TRAVERSAL\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // Method to print the LinkedList.      public static void printList(LinkedList list)      {          Node currNode = list.head;            System.out.print("\nLinkedList: ");            // Traverse through the LinkedList          while (currNode != null) {              // Print the data at current node              System.out.print(currNode.data + " ");                // Go to next node              currNode = currNode.next;          }          System.out.println("\n");      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*DELETION BY KEY\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // Method to delete a node in the LinkedList by KEY      public static LinkedList deleteByKey(LinkedList list, int key)      {          // Store head node          Node currNode = list.head, prev = null;            //          // CASE 1:          // If head node itself holds the key to be deleted            if (currNode != null && currNode.data == key) {              list.head = currNode.next; // Changed head                // Display the message              System.out.println(key + " found and deleted");                // Return the updated List              return list;          }            //          // CASE 2:          // If the key is somewhere other than at head          //            // Search for the key to be deleted,          // keep track of the previous node          // as it is needed to change currNode.next          while (currNode != null && currNode.data != key) {              // If currNode does not hold key              // continue to next node              prev = currNode;              currNode = currNode.next;          }            // If the key was present, it should be at currNode          // Therefore the currNode shall not be null          if (currNode != null) {              // Since the key is at currNode              // Unlink currNode from linked list              prev.next = currNode.next;                // Display the message              System.out.println(key + " found and deleted");          }            //          // CASE 3: The key is not present          //            // If key was not present in linked list          // currNode should be null          if (currNode == null) {              // Display the message              System.out.println(key + " not found");          }            // return the List          return list;      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*DELETION AT A POSITION\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // Method to delete a node in the LinkedList by POSITION      public static LinkedList deleteAtPosition(LinkedList list, int index)      {          // Store head node          Node currNode = list.head, prev = null;            //          // CASE 1:          // If index is 0, then head node itself is to be deleted            if (index == 0 && currNode != null) {              list.head = currNode.next; // Changed head                // Display the message              System.out.println(index + " position element deleted");                // Return the updated List              return list;          }            //          // CASE 2:          // If the index is greater than 0 but less than the size of LinkedList          //          // The counter          int counter = 0;            // Count for the index to be deleted,          // keep track of the previous node          // as it is needed to change currNode.next          while (currNode != null) {                if (counter == index) {                  // Since the currNode is the required position                  // Unlink currNode from linked list                  prev.next = currNode.next;                    // Display the message                  System.out.println(index + " position element deleted");                  break;              }              else {                  // If current position is not the index                  // continue to next node                  prev = currNode;                  currNode = currNode.next;                  counter++;              }          }            // If the position element was found, it should be at currNode          // Therefore the currNode shall not be null          //          // CASE 3: The index is greater than the size of the LinkedList          //          // In this case, the currNode should be null          if (currNode == null) {              // Display the message              System.out.println(index + " position element not found");          }            // return the List          return list;      }        // \*\*\*\*\*\*\*\*\*\*\*\*\*\*MAIN METHOD\*\*\*\*\*\*\*\*\*\*\*\*\*\*        // method to create a Singly linked list with n nodes      public static void main(String[] args)      {          /\* Start with the empty list. \*/          LinkedList list = new LinkedList();            //          // \*\*\*\*\*\*INSERTION\*\*\*\*\*\*          //            // Insert the values          list = insert(list, 1);          list = insert(list, 2);          list = insert(list, 3);          list = insert(list, 4);          list = insert(list, 5);          list = insert(list, 6);          list = insert(list, 7);          list = insert(list, 8);            // Print the LinkedList          printList(list);            //          // \*\*\*\*\*\*DELETION BY KEY\*\*\*\*\*\*          //            // Delete node with value 1          // In this case the key is \*\*\*at head\*\*\*          deleteByKey(list, 1);            // Print the LinkedList          printList(list);            // Delete node with value 4          // In this case the key is present \*\*\*in the middle\*\*\*          deleteByKey(list, 4);            // Print the LinkedList          printList(list);            // Delete node with value 10          // In this case the key is \*\*\*not present\*\*\*          deleteByKey(list, 10);            // Print the LinkedList          printList(list);            //          // \*\*\*\*\*\*DELETION AT POSITION\*\*\*\*\*\*          //            // Delete node at position 0          // In this case the key is \*\*\*at head\*\*\*          deleteAtPosition(list, 0);            // Print the LinkedList          printList(list);            // Delete node at position 2          // In this case the key is present \*\*\*in the middle\*\*\*          deleteAtPosition(list, 2);            // Print the LinkedList          printList(list);            // Delete node at position 10          // In this case the key is \*\*\*not present\*\*\*          deleteAtPosition(list, 10);            // Print the LinkedList          printList(list);      }  } |

**Output:**

LinkedList: 1 2 3 4 5 6 7 8

1 found and deleted

LinkedList: 2 3 4 5 6 7 8

4 found and deleted

LinkedList: 2 3 5 6 7 8

10 not found

LinkedList: 2 3 5 6 7 8

0 position element deleted

LinkedList: 3 5 6 7 8

2 position element deleted

LinkedList: 3 5 7 8

10 position element not found

LinkedList: 3 5 7 8

GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer [Placement 100](https://practice.geeksforgeeks.org/courses/Placement-100/?ref=gfg_post) for details

**An In-Depth Look at java.util.LinkedList**

## Difference Between LinkedList and java.util.LinkedList

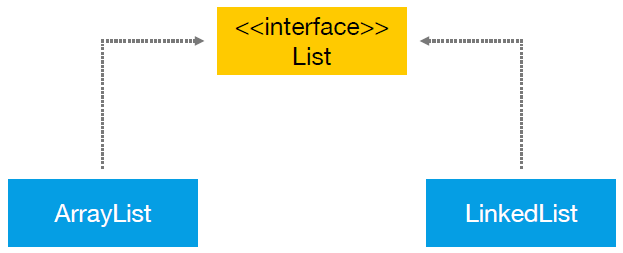
So what is the difference between the LinkedList data structure and the class java.util.LinkedList?

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.

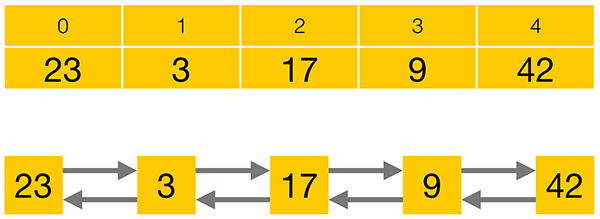
So in this article, I will focus on one specific Linked List implementation, the java.util.LinkedList class. Among other interfaces, LinkedList implements the java.util.List interface. You can have duplicate elements in a List and you can go from element to element in the same order as the elements were 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. [Michael Rasmussen](https://dzone.com/users/1239605/Maldivia.html)has done a[JMH benchmark showing this nicely](https://gist.github.com/anonymous/1ff7112bd9c187a9438a).

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) {…}

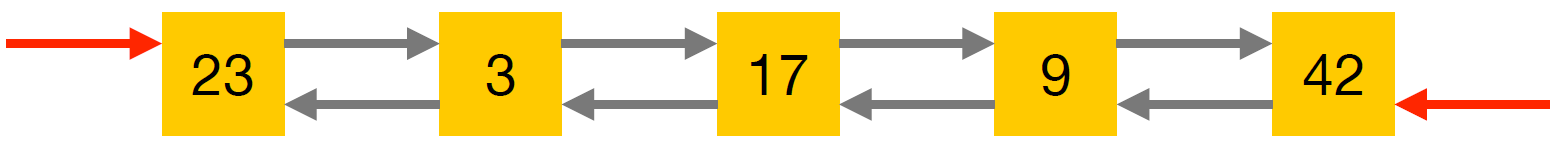
public E remove(int index) {…}

[…]

}

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.

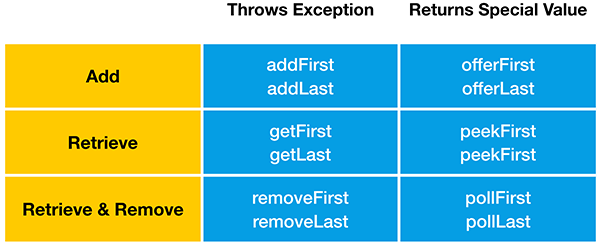
A Queue allows adding elements to the end of the Queue.

“**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 null. 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 null.

## **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 null 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 null 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.