ListIterator in Java

The ListIterator works only with the List interface in Java and it is inherited from the Iterator interface. So, it includes all functionalities of the iterator interface.

- It works only with Lists like ArrayList, Vector, LinkedLists.
- It is inherited from the Iterator interface. So, it includes all functionalities of the iterator interface.
- In addition to next(), hashNext() and remove(), it provides the below methods:
    1. **hasPrevious()** - It is used to check if we have previous item for the item pointed by current iterator or not.
    2. **previous()** - It returns the previous element of the list, and moves the iterator one position back.
    3. **add()** - It is used to add an item while iterating through the List.
    4. **set()** - replaces the element returned by either next() or previous() with the specified element
    5. **nextIndex()** returns the index of the element that the next() method will return
    6. **previousIndex()** - returns the index of the element that the previous() method will return

Note: We can use iterators in List to traverse in both forward and backward directions. The listIterator() method gives us an iterator pointing to the first element whereas the listIterator(int index) method gives us an iterator pointing to a specific element. We can pass the size of the list as a parameter to get an iterator pointing to the last element.

Forward Traversal using ListIterator:

Java

```java
import java.util.*;

class Gfg{
    public static void main (String[] args) {

        // Create a List
        List<Integer> list = new ArrayList<Integer>();

        // Add element to List
        list.add(10);
        list.add(20);
        list.add(30);

        // Iterator pointing to position just before
        // first element
        ListIterator<Integer> it = list.listIterator();

        // While there is a next element of the
        // current iterator
        while(it.hasNext())
        {
            // Print the next element and increment
            // iterator by one position
            System.out.println(it.next());
        }
    }
}
```

Output:

```
10
20
30
```

Backward Traversal using ListIterator: This time use the same List as above and create an iterator pointing to the last position that is the position just after the last element in the List.

Java

```java
import java.util.*;

class Gfg{
    public static void main (String[] args) {

        // Create a List
        List<Integer> list = new ArrayList<Integer>();

        // Add element to List
        list.add(10);
        list.add(20);
        list.add(30);

        // Iterator pointing to position just after
        // last element
        ListIterator<Integer> it = list.listIterator(list.size());

        // While there is a previous element of the
        // current iterator
        while(it.hasPrevious())
        {
            // Print the previous element and decrement
            // iterator by one position
            System.out.println(it.previous());
        }
    }
}
```

Output:

```
30
20
10
```

Example of set() method: The set() methods sets or replaces the item last returned by next() or previous() method with a given value.

Java

```
import java.util.*;
class Gfg{
    public static void main (String[] args) {
        // Create a List
        List<Integer> list = new ArrayList<Integer>();
         // Add element to List
        list.add(10);
        list.add(20);
        list.add(30);
         // Iterator pointing to position just after
        // last element
        ListIterator<Integer> it = list.listIterator(list.size());
         // While there is a previous element of the
        // current iterator
        while(it.hasPrevious()) {
            int x = (Integer)it.previous();

            // Replaces last element returned by previous()
            // everytime with double of its value.
            it.set(x*2); }
        // Print list
        System.out.println(list);
    }}
```

Output:

```
[20, 40, 60]
```

add() method: The add() method in List interface is used to add an element just before the current iterator position.

ArrayLists in Java

ArrayList is a part of the collection framework and is present in java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

Advantages of ArrayList over Arrays:

1. **Dynamic Size:** As we stated above already, ArrayList is a dynamic array, that is, we can increase size of ArrayList in runtime. Unlike arrays, we don't need to predefined size or number of elements for an ArrayList. This is helpful in situations where we don't know beforehand the number of elements we are going to store in an Array.
2. **Rich Library:** ArrayList provides us with a lot of built-in functions to performs some complex operations efficiently. For example, we can add an element directly at the middle of ArrayList using the add() function. There is a function to find the first occurrence of an item, there is a function to remove an item and a lot more.
   Note: Since, ArrayList is a part of Collections, it can be created to store data of only non-primitive types.
   Below is a sample program to demonstrate ArrayList:

Note: You can also specify size while creating an ArrayList. By doing this, we can also use an ArrayList as normal Arrays.

Internal Working of ArrayLists

Like dynamic arrays in other languages, ArrayLists in Java also uses normal arrays internally to implement dynamic resizing.

1. It internally uses arrays.
2. If internal array becomes full, do the following:
    - Create a new array of double size.
    - Copy elements from previous array to this newly created array.
    - Free space allocated to old array.

Note: This is a general implementation of dynamic arrays, the actual ArrayList implementation may vary from version to version. In Java 1.8, it is claimed at different sources like StackOverflow and Wikipedia that Java pre-allocates space for 10 items and creates a new space of size 1.5 times instead of double when old space gets full.

The major advantage of ArrayLists is that it has all advantages that come with arrays because it internally uses Arrays. The two major advantages of arrays are:

1. Cache Friendliness.
2. Random access of elements.

The amortized time complexity of adding an element to the end of ArrayList(dynamic arrays) is O(1), however, the worst-case time complexity is still O(N). So, it is wiser to use Arrays over ArrayList if you know the number of elements you are going to store beforehand.

Let us understand the internal implementation of ArrayList more deeply, and see how is the amortized time complexity of adding an element is O(1).

Consider the below example where we create an empty ArrayList first and then inserts 100 items in this ArrayList:

```
    public static void main (String[] args) {

        // Create a ArrayList
        ArrayList<Integer> al = new ArrayList<Integer>();

        for(int i=1; i<=100; i++) {
            al.add(i); }
            // Print the ArrayList
        System.out.println(al);     }
```

Consider that internally ArrayList contains an array of size 10 initially, and creates an array of double size when old array gets full.

For the above program, we need to insert 100 elements.

Since there is space for 10 elements initially, we can easily insert first 10 elements.

For the 11th element, create an array of size 20, copy the last 10 elements to it and proceed further.

- Similarly, for 21st element, create an array of size 40, copy the last 20 elements to it and proceed further.
- For the 41st element, create an array of size 80, copy the last 40 elements to it and proceed further.
- Again for the 81st element, create an array of size 160, copy the last 80 elements to it and proceed further.

The Amortized time complexity is said to be the average time complexity for any operation.

Let's assume there are N items initially and we double the size of the array every time it gets full.

Therefore, the average time to insert (N + 1) items = { θ(1) + θ(1) + θ(1) +.......+ θ(1) + θ(N)} / (N + 1)

Since, the array initially has N capacity, inserting first N items will cost θ(1) time, and for the last item which is (N + 1)th item, it will cost θ(N) as we have to copy first N items to newly created array of double size.

```
So, average time = {θ(N) + θ(N)}/(N + 1)
                 = θ(1)
```

ArrayList Methods

Performing Various Operations on ArrayList

Let's see how to perform some basics operations on the ArrayList.

Adding Elements: In order to add an element to an ArrayList, we can use the add() method. This method is overloaded to perform multiple operations based on different parameters. They are:

- **add(Object):** This method is used to add an element at the end of the ArrayList.
- **add(int index, Object):** This method is used to add an element at a specific index in the ArrayList and moves all element previously present from that index one position ahead.

Checking if an element is present. The ArrayList class provides a method **contains()** which takes an object of type ArrayList element as a parameter and checks if it present in the ArrayList or not. It returns a boolean value true or false based on if the object is present or not.

Removing Elements: In order to remove an element from an ArrayList, we can use the remove() method. This method is overloaded to perform multiple operations based on different parameters. They are:

- **remove(Object):** This method is used to simply remove an object from the ArrayList. If there are multiple such objects, then the first occurrence of the object is removed.
- **remove(int index):** Since an ArrayList is indexed, this method takes an integer value which simply removes the element present at that specific index in the ArrayList. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.

```
    public static void main (String[] args) {
        // Create a ArrayList
        ArrayList<String> al = new ArrayList<String>();

        al.add("geeks");
        al.add("ide");
        al.add("courses");

        System.out.println(al.contains("ide"));
        al.remove(1);
        System.out.println(al.contains("ide"));

        al.remove("courses");
        System.out.println(al.contains("courses"));
```

Output:

```
true
false
false
```

Let's discuss some more methods of ArrayList.

- **get(index)**: The get() method of ArrayList in Java is used to get the element of a specified index within the list. It throws an IndexOutOfBounds exception if the index is not within the range of ArrayList.
- **set(index, Object)**: The set() method of java.util.ArrayList class is used to replace the element at the specified position in this list with the specified element.
- **indexOf(Object):** The indexOf() method of ArrayList returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
- **lastIndexOf(Object):** The lastIndexOf() method of ArrayList returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

Below program illustrates the above 4 methods:

Java

```
import java.util.*;

class Test{
    public static void main (String[] args) {

        // Create a ArrayList
        ArrayList<Integer> al = new ArrayList<Integer>();

        al.add(10);
        al.add(20);
        al.add(10);
        al.add(30);

        System.out.println(al.get(1));
        al.set(1, 40);
        System.out.println(al.get(1));
        System.out.println(al.indexOf(10));
        System.out.println(al.lastIndexOf(10));
        System.out.println(al.indexOf(50));
```

Output:

```
20
40
0
2
-1
```

There are two more important methods available in ArrayList:

- **clear()**: This method is simply used to clear an ArrayList by deleting all of its elements. It's return type is void and it doesn't return anything.
- Below program illustrates the above methods:

```
import java.util.*;

class Test{
    public static void main (String[] args) {

        // Create a ArrayList
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(10);
        al.add(30);

        System.out.println(al.isEmpty());
        al.clear();
        System.out.println(al.isEmpty());
```

Output:

```
false
true
```

Time Complexity Analysis of the above methods:

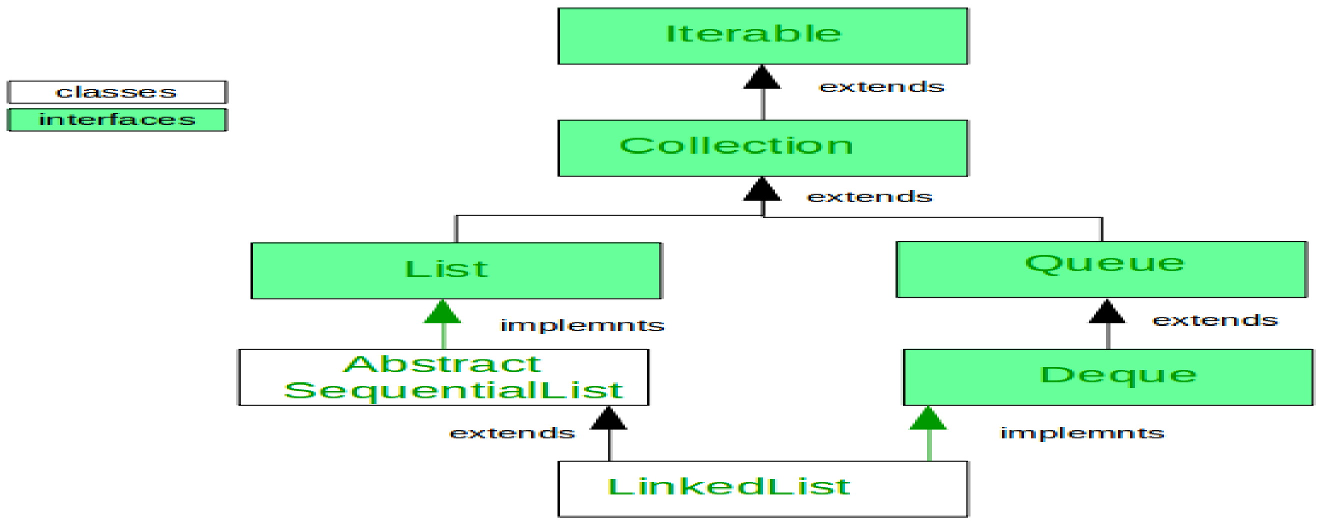
```
add(obj) --------> Amortized O(1)

size()     -------
isEmpty() -------  \____  Worst Case O(1)
get()------------  /
set() -----------

contains()  ----------------
indexOf()   ----------------  \
lastIndexOf()  -------------    ------ Worst Case O(N)
remove() [both versions] ---  /
add(index, obj)  -----------
```

LinkedList in Java

Linked List is a part of the Collection framework present in java. util package. This class is an implementation of the LinkedList data structure which is a linear data structure where every element is stored as a separate object with a data part and address part. The elements are linked using pointers and addresses called nodes. It is rarely used in a production scenario but it has its own advantages. Basically it can be used in two particular cases.

It can be used as an ArrayList when the list type operation is performed for eg, insertion, deletion, remove, etc. It implements all the functions of the List interface just like an ArrayList.

- It can be also be used as an ArrayDeque when queue or dequeue type of operations are performed, thus implementing the Deque interface.



Advantages of LinkedList class in Java.

The LinkedList allows Dynamic Memory allocation, which means that memory allocation is done at the run time by the compiler, unlike the Arrays where the size is needed to be fixed from before.

- LinkedList elements don't need contiguous memory locations.
- Insert and delete operations in the LinkedList are less expensive even in the worst case due to the reference of the previous and next elements which is highly unlikely in the case of ArrayList and ArrayDeque which internally uses a resizable array.

Disadvantages of LinkedList class in Java.

The nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

- ArrayList will have fewer cache misses while traversal than the LinkedList. Therefore it is more cache-friendly.

Example:
Java
```java
// Java code implementing LinkedList
import java.util.LinkedList;

public class GfG {

    public static void main(String args[])
    {
        // Creating object of the
        // class linked list
        LinkedList<Integer> list
        = new LinkedList<Integer>();

        // Adding elements to the linked list
        list.add(10);
        list.add(20);
        list.add(30);

        // Displaying the LinkedList
        System.out.println(list);
    }
}
```
Output:
```
[10, 20, 30]
```
Internal Working of a LinkedList
Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the doubly linked list data structure. The main difference between a normal linked list and a doubly LinkedList is that a doubly linked list contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.
Methods common to both LinkedList and ArrayList

Method	Description	Time Complexity
add(E e)	This method Appends the specified element to the end of this list.	Theta(1)
add(int index, E element)	This method Inserts the specified element at the specified position in this list.	Theta(index)
contains(Object o)	This method returns true if this list contains the specified element.	O(n)
remove(int index)	This method removes the element at the specified position in this list.	Theta(index)
remove(Object o)	This method removes the first occurrence of the specified element from this list, if it is present.	O(n)
get(int index)	This method returns the element at the specified position in this list.	Theta(index)
set(int index, E element)	This method replaces the element at the specified position in this list with the specified element.	Theta(index)
indexOf(Object o)	This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.	O(n)
lastIndexOf(Object o)	This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.	O(n)
isEmpty()	This method is used to check if a list is empty or not.	O(1)

Methods implementing Queue interface

Method	Description
add(E e)	This method Appends the specified element to the end of this list. Throws exception when element cannot be added to the list.
remove()	This method retrieves and removes the head (first element) of this list. Throws exception when the list is empty.
element()	This method retrieves, but does not remove, the head (first element) of this list. Throws exception when the list is empty.
offer(E e)	This method Adds the specified element as the tail (last element) of this list. Returns null when element cannot be added to the list .
poll()	This method retrieves and removes the head (first element) of this list. Returns null when the list is empty.
peek()	This method retrieves, but does not remove, the head (first element) of this list. Returns null when the list is empty.

Methods implementing DeQueue interface

Method	Description	Unsuccesful Response
addFirst(E e)	This method Inserts the specified element at the beginning of this list.	Throws Exception
addLast(E e)	This method Appends the specified element to the end of this list.	Throws Exception
removeFirst()	This method removes and returns the first element from this list.	Throws Exception
removeLast()	This method removes and returns the last element from this list.	Throws Exception
getFirst()	This method returns the first element in this list.	Throws Exception
getLast()	This method returns the last element in this list.	Throws Exception
offerFirst(E e)	This method Inserts the specified element at the front of this list.	Returns Null
offerLast(E e)	This method Inserts the specified element at the end of this list.	Returns Null
pollFirst()	This method retrieves and removes the first element of this list, or returns null if this list is empty.	Returns Null
pollLast()	This method retrieves and removes the last element of this list, or returns null if this list is empty.	Returns Null
peekFirst()	This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.	Returns Null
peekLast()	This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.	Returns Null

ArrayList vs LinkedList in Java

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. However, the limitation of the array is that the size of the array is predefined and fixed. There are multiple ways to solve this problem. In this article, the difference between two classes that are implemented to solve this problem named **ArrayList** and **LinkedList** is discussed.
ArrayList is a part of the **collection framework**. It is present in the **java.util** package and provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. We can dynamically add and remove items. It automatically resizes itself. The following is an example to demonstrate the implementation of the ArrayList.
Example
Java
```java
// Java program to Illustrate Working of an ArrayList

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an ArrayList of Integer type
        ArrayList<Integer> arrli
            = new ArrayList<Integer>();

        // Appending the new elements
        // at the end of the list
        // using add () method via for loops
        for (int i = 1; i <= 5; i++)
            arrli.add(i);

        // Printing the ArrayList
        System.out.println(arrli);

        // Removing an element at index 3
        // from the ArrayList
        // using remove() method
        arrli.remove(3);

        // Printing the ArrayList after
        // removing the element
        System.out.println(arrli);
```
Output
```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
```
LinkedList is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. The following is an example to demonstrate the implementation of the LinkedList.
Note: This class implements the **LinkedList Data Structure**.
Example
Java
```java
    // main driver method
    public static void main(String args[])
    {

        // Creating an object of the
        // class linked list
        LinkedList<String> object
            = new LinkedList<String>();

        // Adding the elements to the object created
        // using add() and addLast() method
        // Custom input elements
        object.add("A");
        object.add("B");
        object.addLast("C");

        // Print the current LinkedList
        System.out.println(object);

        // Removing elements from the List object
        // using remove() and removeFirst() method
        object.remove("B");
        object.removeFirst();

        System.out.println("Linked list after "
                           + "deletion: " + object);
    }
}
```

Output:

```
[A, B, C]
Linked list after deletion: [C]
```

Now after having an adequate understanding of both of them let us do discuss the differences between ArrayList and LinkedList in Java

ArrayList	LinkedList
This class uses a dynamic array to store the elements in it. With the introduction of generics, this class supports the storage of all types of objects.	This class uses a doubly linked list to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.
Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
This class implements a List interface. Therefore, this acts as a list.	This class implements both the List interface and the Deque interface. Therefore, it can act as a list and a deque.
This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.