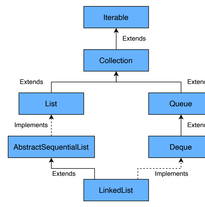
The **LinkedList** class in Java implements the **List** and the **Deque** interface. Some of the salient features of a LinkedList are:

1. The elements are inserted in the order of insertion.
2. It supports duplicate elements.
3. We can add any number of null elements.



Internal implementation of LinkedList

The **LinkedList** class has a static inner class called `Node`. This class contains three fields:

`item` - This contains the value of the current element.

`next` - This contains the pointer to the next element.

`prev` - This contains the pointer to the previous element.

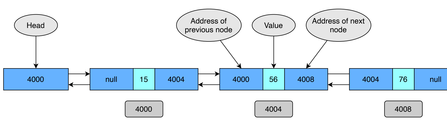
Below is the code for the `Node` class.

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

When an element is added to the **LinkedList**, a new `Node` instance is created. Depending on where the new node is being added, the **prev** and **next** fields are set.

When a node at index `i` is removed, the `next` field of node at index `i-1` is set to the node at index `i+1`. Similarly, the `prev` field of node at index `i+1` is set to node `i-1`.



LinkedList internal

Time complexities for LinkedList operations

Let's see what the time complexities are for different operations in a LinkedList.

Adding an element

The complexity of adding an element in a **LinkedList** is $O(1)$. If we need to search for the position where the Node needs to be inserted, then the complexity is $O(n)$, but the element is usually inserted at the beginning or end, which makes it $O(1)$.

The biggest benefit of **LinkedList**, in comparison to an array, is that in a **LinkedList**, in comparison to an array, is that when new elements are added or removed, the other elements are not rearranged.

Removing an element

Removing an element is also an $O(1)$ operation if we are aware of the position of the element that needs to be removed. If we need to search and remove an element, it is an $O(n)$ operation.

Searching an element

Searching an element is an $O(n)$ operation, as the entire **LinkedList** is iterated to search the element in the worst case.

Creating a LinkedList

There are two ways to create a **LinkedList**:

Using the no-arg constructor

The default constructor does not take any argument and creates a **LinkedList** of size zero. Below is the syntax to create **LinkedList** using the default constructor.

```
List<Integer> list = new LinkedList<Integer>();
```

Using existing Collection

A **LinkedList** can also be created using an existing Collection. The newly created **LinkedList** will contain all the elements in the same order as the original Collection.

```
List<Integer> list = new LinkedList<Integer>(oldList);
```

Inserting an element into a LinkedList

Let's look at some of the methods used to insert an element into LinkedList.

Inserting a single element at the end.

To insert a single element at the end, we can use the `add(E e)` or `addLast(E e)` method. These methods insert the given element at the end of the list and do not return anything.

Inserting a single element at the beginning

We can use the `addFirst(E e)` method to insert an element at the beginning.

Inserting an element at a particular index

We can use the `add(int index, E element)` method to insert an element at a particular index. The index should be greater than zero and less than the size of the **LinkedList**; otherwise, `IndexOutOfBoundsException` is thrown.

Inserting multiple elements from another Collection

If we have a **Collection** and we need to add all its elements to another **LinkedList**, then the `addAll(Collection c)` method can be used. This method will add all the elements at the end of the **LinkedList**.

```
list.addAll(anotherList)
```

Inserting multiple elements from another Collection at a particular index

If we have a Collection and we need to add all its elements to another **LinkedList** at a particular index, then the `addAll(int index, Collection c)` method can be used. This method inserts all of the elements in the specified collection into this list starting at the specified position.

```
list.addAll(3, anotherList)
```

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class LinkedListDemo {

    public static void main(String args[]) {
        LinkedList<Integer> linkedList = new LinkedList<>();

        linkedList.add(1); // Adds 1 to the list.
        linkedList.add(2); // Adds 2 to the end of the list.
        linkedList.addLast(3); // Adds 3 to the end of the list.
        System.out.println(linkedList);

        linkedList.addFirst(10); // Adds 10 to the start of the list.
        System.out.println(linkedList);

        linkedList.add(2, 20); // Adds 20 to second position in the list.
        System.out.println(linkedList);

        List<Integer> list = new ArrayList<>();
        list.add(101);
        list.add(102);
        list.add(103);

        linkedList.addAll(3, list); // Adds the collection of elements at third position in the list.
        System.out.println(linkedList);
    }
```

Run Save Reset