# **Doubly Linked List**

* In this section, we provide the implementation of a usable LinkedList generic class similar to the one available in the Java API.
* We call our implementation of Java’s LinkedList class MyLinkedList.java to avoid any ambiguity.

## **Class Design**

* In considering the design, we will need to provide three classes:

1. **List Class**
   1. Contains links to the head and to the tail.
   2. Contains the size of the list.
   3. Contains a host of other methods.
2. **Node Class**
   1. A private nested class of MyLinkedList.
   2. Contains the data and links to the previous and next nodes, along with appropriate constructors.
3. **Iterator Class**
   1. A private nested class of MyLinkedList.
   2. Implements the Iterator interface
   3. Abstracts the notion of a position in the list by providing implementations of next(), hasNext(), and remove().

* Recall that the LinkedList class will be implemented as a doubly linked list, and that we will need to maintain references to both ends of the list.
* Making a singly linked list doubly linked gives us the luxury of **adding or removing an element from the end of a list in O(1) instead of O(N-1) time.**
* Therefore, a doubly linked list allows us to maintain constant time cost per operation, so long as the operation occurs at a known position.
* The known position can be either end, or at a position specified by an iterator (however, we do not implement a ListIterator, thus leaving some code for the reader).

## **Sentinel Nodes**

* Because the iterator classes store a reference to the “current node,” and the end marker is a valid position, it makes sense to create an extra node at the end of the list to represent the end marker.
* Further, we can create an extra node at the front of the list, logically representing the beginning marker.
* These extra nodes are sometimes known as **sentinel nodes**
  1. the node at the front is sometimes known as a **header node**
  2. the node at the end is sometimes known as a **tail node**
* The advantage of using these extra nodes is that they greatly simplify the coding by removing a number of special cases.
* For instance, if we do not use a header node, then removing the first node becomes a special case, because we must reset the list’s head pointer to the first node during the remove, and also because the remove algorithm in general needs to access the node prior to the node being removed (and without a header node, the first node does not have a node prior to it).

## **Class Visualization**

* Figure 3.22 shows a doubly linked list with header and tail nodes.

Diagram, schematic

Description automatically generated

* Figure 3.23 shows an empty list.

Diagram

Description automatically generated

## **Node Class**

* Inside the MyLinkedList class is a private static nested Node class.
* The Node class, consists of
  1. the stored item,
  2. link to the previous Node
  3. link to the next Node,
  4. a constructor
* Figure 3.25 shows the Node class.

Text

Description automatically generated

* All the data members are public.
* Recall that normally in a class, the data members should be private.
* However, members in a nested class are visible even in the outer class.
* Since the Node class is private, the visibility of the data members in the Node class is irrelevant.
  + The MyLinkedList methods can see all Node data members
  + Classes outside of MyLinkedList cannot see the Node class at all
* It doesn’t matter what access specifiers you give your static-nested class fields, because the class acts as a static variable of the outer class.
* The MyLinkedList methods can see all Node data members (and vice-versa), and classes outside of MyLinkedList cannot see the Node class at all.

## **MyLinkedList Fields and Methods**

* MyLinkedList contains four main class fields
  1. **head**: a reference to the header node
  2. **tail**: a reference to the tail node
  3. **size**: the size of the list
  4. **modCount**: an iterator variable called modCount to track changes in the list
* modCount represents the number of changes to the linked list since construction.
* Each call to add or remove will update modCount.
* The idea is that when an iterator is created, it will store the modCount of the collection.
* Each call to an iterator method (next or remove) will check the stored modCount in the iterator with the current modCount in the linked list and will throw a ConcurrentModificationException if these two counts don’t match.
* The rest of the MyLinkedList class consists of
  1. constructors,
  2. the implementation of the iterator,
  3. insert and remove methods
  4. other useful methods (any of the methods are one-liners).

**Great Resources:**

Different Version of Doubly Linked Lists: <https://www.cct.lsu.edu/~sidhanti/tutorials/data_structures/page163.html>

Greater Detail:

<https://www.cs.dartmouth.edu/~scot/cs10/lectures/6/6.html>

Why Have Sentinel Nodes?

<https://stackoverflow.com/questions/5384358/how-does-a-sentinel-node-offer-benefits-over-null>

<https://stackoverflow.com/questions/61683371/sentinel-node-in-doubly-linked-list>

Summary of Notes:

<https://hajiamini.cs.grinnell.edu/courses/CSC207/class_notes/CircularList.pdf>