Intro to Linked Lists:

A linked list is a bunch of node objects that are linked together. Like ArrayList, LinkedList implements the List Interface.

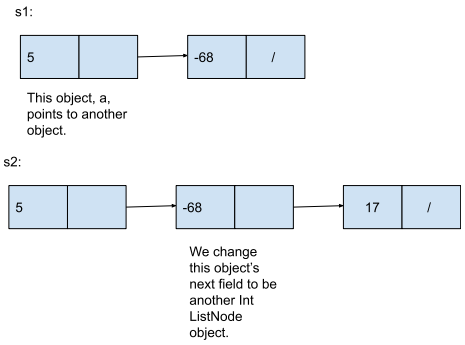
Nodes: contains some value and a reference variable to the next node.

Example of a Node in a Linked List for Integer (we'll make it generic and iterable later):

```
public class IntListNode {
      public int data;
      public IntListNode next;
      public IntListNode(int data, IntListNode next) {
            this.data = data;
            this.next = next;
}
}
```

The instance variables are public, and since no constructor is provided, the default constructor is put in.

```
IntListNode a = new IntListNode(5, new IntListNode(-68, null)); //s1
a.next.next = new IntListNode(17, null); //s2
```



Traverse Linked Lists:

From now on, we'll work with a ListNode class that has setter/getter methods for data and next (before, they were public, which violates the encapsulation principle. Now we will make them private).

```
while(list != null) {
      System.out.println(list.getValue());
      list = list.getNext();
      }
```

The whole list is gone after this code runs!
The node objects are still around, but since we burned through all the Nexts, there is no way to access them.

To preserve the list:

```
ListNode current = list;
while(current != null) {
      System.out.println(current.getValue());
      current = current.getNext();
}
```

Two cases for the boolean condition in the while-loop header:

- The code needs to leave temp on the last node, but not do anything to the last node in the while loop:
  - `temp.getNext() != null`
- The code does not need to leave temp on the last node, and needs to do something to the last node in the while loop:
  - `temp != null`

Basic Operations in Linked Lists:

Adding a node to the start of the list:

```
list = new IntListNode( value_you_need_to_add, list);
Ex:
IntListNode list = new IntListNode(10, new IntListNode(46,
null));
//list looks like this: 10 → 46 /
list = new IntListNode(15, list);
//list is now: 15 → 10 → 46 /
```

Tack on a list (list2) to the end of another list (list1):

```
IntListNode copy_list1 = list1;
while(copy_list1.getNext() != null) {
      copy_list1 = copy_list1.getNext();
```

}//this code stops when the "next" of the copy node is null. We need to set that "next" to list2 !

```
copy_list1.setNext(list2);
```

Removing Every Nth Node

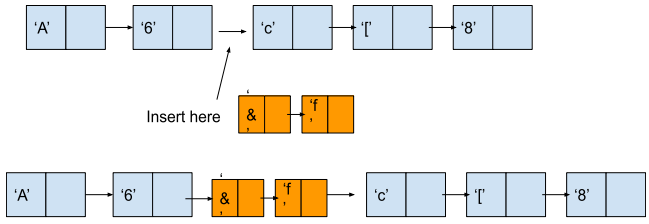
```
public void removeNthNode(int x) {
    int counter = 1;
    ListNode temp = list;
    while(temp != null) {
      if(counter % x == x-1) {
        if(temp.getNext() != null) {
          temp.setNext(temp.getNext().getNext());
```

```
            counter++;
          }
        }
        counter++;
        temp = temp.getNext();
      }
    }
```

Insert a list (list2) into another list (list1) at a certain index N.

Have the last node in list 2 point to the Nth node in list1.

Have the (N-1) node in list1 point to the start of list2.



Example Diagram

In another example, let's say we want to insert list2 into list1 at the 5th index.

The last node in list2 needs to point to the 5th node of list1.

Then, the 4th node of list1 should point to the first node in list2.

```
ListNode temp_1 = list1;
for(int i = 0; i < 4; i++) { //gets us to 4th node.
     temp_1 = temp_1.getNext();
}

ListNode temp_2 = list2;
while(temp_2.getNext() != null) { //last node of list2
     temp_2 = temp_2.getNext();
}
temp_2.setNext(temp_1.getNext());

temp_1.setNext(list2);
```

Application of this: Create a new node with the same value as the second to last node and add this new node at the second to last spot. Once finished, the second to last node will occur twice.

Solution: Create a new node that has the same value as the second to last, and points to the second to last node of the original list. Then, get the third to last node in the list to point to this new node.

Infinite Loops in LinkedLists

If you have nodes that point to each other in circular patterns, the length of your list could be infinite (a while loop would never finish). To be clear, infinite length LinkedLists are *legal* and okay in Java, and they might show up part-way through your solution, but they could cause problems if you don't do it intentionally!

```
Ex1:
ListNode AP = new ListNode("Norwalk", null);
AP.setNext(AP);
```

The AP list is as follows: "Norwalk" → "Norwalk" → "Norwalk" …
There is only one node in the list, but trying to traverse it would run infinitely.

```
Ex2:
ListNode BC = new ListNode(7.6, new ListNode(9.3, null));
BC.getNext().setValue(-9.4);
BC.getNext().setNext(new ListNode(0, BC.getNext()));
```

We set up a LinkedList where the first Node is called BC and it points to a Node with the value 9.3. That node changes its value to -9.4, so now the LinkedList looks like this: 7.6 → -9.4.
The second node now points to another Node, which in turns points to the second node.
7.6 → -9.4 → 0 → -9.4 → 0 → -9.4 ….
So the 2nd and 3rd node repeat forever.

To be clear, there are only **3** nodes in this list, but since they point to each other in a cyclical way, running a while loop through this would run infinitely.

Passing LinkedLists as Parameters

When you pass an object as a parameter to a method, the method gets an alias to that object.

- This means when you are traversing the list, you don't need to create an alias like "temp" or "current", because the Node you are working with is already an alias.

- If you change what node the parameter variable points to, this will not affect the original variable!

Generic and Iterable Linked Lists

To make it easier to work with linked lists, make two classes: A LinkedList class that can do all the basic operations described above with method calls as opposed to copy-pasting the code every time, and a private class within that linked list class to represent the nodes. Making it Iterable means we just need to implement the iterator method. Making an iterator just means making a class with `.next()` , `.hasNext()`, and `.remove()`.

```
//General outline of the code

public class CustomLinkedList implements List<E>, Iterable<E> {
     private int size; //keeps track of # of nodes in list
     private ListNode<E> front; //first node in list.

     //constructors

     //iterable method to satisfy Iterable interface

     //methods to satisfy List interface

     //whatever other methods desired.

     private class ListNode<E> {
          private E data;
          private ListNode<E> next;
          //constructors.
     }
     private class CustomLinkedListIterator  {
          //implement .next(), .hasNext(), .remove()
     }

}
```

ArrayList vs LinkedList: Big-O Comparison

Given a List of Size N:

Operation	LinkedList	ArrayList
Getter/Setter at index (Access)	O(N)	O(1)
Add to Back	O(N)	O(1)
Remove Back	O(N)	O(1)
Add to Front	O(1)	O(N)
Remove Front	O(1)	O(N)
Add at Index	O(N)	O(N)
Remove at Index	O(N)	O(N)

Don't memorize this, because the Time complexity of each of these operations is pretty easy to figure out.

Example: what's the Big-O complexity of removing the first element in a LinkedList?

- `list = list.getNext()`, which takes one step. O(1)

Example 2: compare Big-O complexity of adding to the back of a list if the list is implemented with LinkedList vs ArrayList:

- The backing structure of an ArrayList is an array, so it's O(1) to access any element. Just add the element to the size+1 index of the backing array.
- Meanwhile, for LinkedList you must traverse through the entire list to get to the last node, then set that node next to the element. Going through an N element list takes O(N).

If your code is going to deal with the back of the list more often, ArrayList will be more time-efficient. If the list is going to deal with the front of the list more often, LinkedList will be more efficient.