Additional operations on Linked Lists

Pseudocode: Find the nth node in a linked list (assume it's organized like the goodrich Code): https://github.com/rysharprules/Data-Structures-and-Algorithms-in-Java-6th-Edition/blob/master/src/dsa6/chapter_03/SinglyLinkedList.java

```
findNode(n) {
   if (size() >= n || n < 0) return null;
   Node node = head;
   for(i = 0; i < n; i ++) {
      node = node.next
   }
   return node;
}

boolean insertAfter(n, data) {
  Node node = findNode(n);
  if (node != null) {
    Node newNode = new Node(data, node.next);
    node.next = newNode;
    return true // success!
  }
  return false; // Couldn't do it!
}
```

"cloning" and "deep copying".

Sometimes the entire list or object needs to be copied including everything that it references. Otherwise you'll end up with different lists that "share" data. And then when one list is changed, the other list is altered as well.

List Variations

- Circular Linked List
    - The last element links to the first
    - Draw it out
    - We only have a pointer to a tail element.

    - Applications: turns, process switching (why). Good if the items you need to flip through need to be added/deleted midstream - otherwise an array is equally good or better
    - Goodrich Code: https://github.com/rysharprules/Data-Structures-and-Algorithms-in-Java-6th-Edition/blob/master/src/dsa6/chapter_03/CircularlyLinkedList.java

- Doubly Linked List
    - Each List Element has 2 links: next and previous
    - Draw it out
    - Book functions: Insert Between and having blank nodes at the beginning and end
    - Goodrich code: https://github.com/rysharprules/Data-Structures-and-Algorithms-in-Java-6th-Edition/blob/master/src/dsa6/chapter_03/DoublyLinkedList.java
    - This is actually how Java's LinkedList is implemented.
- Combining: Circular Doubly Linked List

Well Founded Data (ensuring that link following terminates) - Optional

- What if we require all list nodes to have final "next" and final "element"
- Cannot create cycles
    - Concept that's helpful to your instructor called "Well Foundedness"
- If 2 lists share the same tail, you can reuse the data without concern if the data is immutable (final, can't be changed)

Iterators and Recursion on Linked Lists

- Well-Foundedness: It means that definitions can't be circular.
    - In data structures, it means that when you start link hopping, eventually you'll reach a null pointer.
    - Related to the DAG concept we'll get to in the second half of the semester
- Recursive functions work when you are getting "closer and closer" to the base case
- Any singularly linked list that's not circular is well founded.
- So, we can write algorithms to read the list and output a new list
    - One option to do this is to get the iterator (linkedList.iterator)
    - An iterator works with the data structure you have, but it keeps it's place, and you can call hasNext(), and next(), which then advanced the iterator
    - So, sum the linked list, or find the number of odd pieces of data.

Example: Iterator that prints out a linked list:

```
printList(LinkedList l) {
  Iterator it = l.iterator()
   while (it.hasNext()) {
     System.out.println(it.next());
   }
}
```

A few more notes on Linked Lists that apply to all recursive data structures

- Mutable version vs. Immutable
- Immutable is final all the way down
- Immutable Recursive data structures MUST be well-founded (why?)
    - Can’t add, remove: what use is it?
- No need to clone it
- We can give you access to the nodes now that you can’t mess with it
    - Recursive algorithm on nodes, very helpful! Double every node
    - Example: Double every node

Recursive function that builds a list with every piece of data doubled:

```
constructDoubleList(Node n):
  if (n == null) return null;
  Node nextDoubleList = constructDoubleList(n.next)
  return new Node(n.data * 2, nextDoubleList);
```