## Homework # 4

# Linked Lists

Due Date: **Sunday at the end of week 11**

---

## Goals

Create and work with a Linked List data structure.

## Overview

Data structures exist in many places inside of games. You've seen (and helped create) a singly-linked list in class. Your job is to now create a doubly-linked list. This is a list whose Nodes know about the Nodes before and after themselves. Read this article for some general information on linked lists, and this article for the doubly linked list algorithms.

In this assignment, you will implement a linked list to hold input from the user. Then you will take user input in a loop, and alter the list based on what the user types. See the last page for sample output.

## Details

1. Download the homework zip file from MyCourses.

2. Create a class called **LinkedList** that implements the provided **IList** interface. The point of this interface is to give you the method signatures of all the required methods.

3. Create stub code for each of the interface methods (if a method requires you to return something, just return null for now). This will remove all the error messages that appeared when you implemented the interface and allow you to compile as you work on the homework.

4. The provided **Node** class represents a single node in a double linked list and includes properties that you should use. Examine the attributes, constructors and properties to see how they work.

5. The **LinkedList** class will hold **String** objects. Each **String** represents a bit of user input.

6. Note: you may not use arrays for this assignment.

## LinkedList Class Details:

1. Create the following attributes
    a. head – The node at the beginning of the list
    b. tail – The node at the end of the list
    c. count – The number of nodes currently in the list

2. Implement each of the methods as described below:

    **a.** `void Add( String data )`
    - Add a new **Node** object (with the specified **data**) to the end of the list.
    - Increment the count and update the head and/or tail when you add the node.

    **b.** `void Insert( String data, int index )`
    - Insert a new **Node** object (with the specified **data**) at the specified index in the list. The index is zero based, of course.

    **Specifics:**

    - If the index is less than or equal to zero, insert the new Node into the list at position zero.
        - Be sure to increment the count and update the head and/or tail if necessary
    - If there is already a **Node** at that index, then
        - Make the new Node
        - This new Node needs to be "hooked up" between the node at the specified index and the one before it.
            - This requires you to set a total of 4 "next" and "previous" attributes. Drawing it out on paper can help you visualize it!
        - Make sure all next/previous attributes maintain the correct chain of doubly-linked Nodes
        - Increment the count and update the head and/or tail if necessary.
    - If the index is greater than or equal to the count of the list, add the new node to the end of the list.
        - You already have a method for adding – use it!
        - Remember that the Add method updates the count for you

c. `String Remove( int index )`

- This method will remove a **Node** from the list and return its data.

**Four possible cases:**

- If an invalid index is given, do not change the list. Just return a null.
- If the index is zero, update the head pointer to refer to the current head's "next node", and set that node's "previous node" to null (if it exists). Decrease the count and return the data value from the old head node.
- If the index is the last node in the list, update the tail pointer to the second to last node in the list. Then update the new tail node's "next node" to be null. Decrease the count and return the data value from the old tail node.
- If the index points to a node in the middle of the list, locate the next node and the previous node of the one you want to remove. Update the previous and next node's pointers to keep the linked list intact (essentially removing all references to the node we're removing). Decrease the count and return the data value from the old node at the given index.

d. `String GetElement( int index )`

This method returns the data value from the node at the index specified. If the index value is invalid, return null.

e. `void Clear()`

This method will clear the list. Update the count attribute, as well as the head and tail.

f. `int Count { get; }`

This property returns a count of the number of nodes in the linked list.

## Test Class:

Your main method should have a loop that will get user input. As the user types things, add each one to the list (as long as it is not a special command). Check for special commands typed by the user and perform the corresponding action rather than adding that command to the list.

| Command | Action |
|---|---|
| q or quit | End the loop |
| print | Print everything in the list |
| count | Print the number of items in the list |
| clear | Clear the entire list |
| remove | Randomly remove one element from the list |
| scramble | Remove a random element from the list and insert it back into the list at a random index |

## Deliverables:

Turn in all of the homework materials (the entire solution) in a single zip file.

## Sample Output:

```
Welcome to the Linked List homework!

Type something: hello
“hello” has been added to the list

Type something: goodbye
“goodbye” has been added to the list

Type something: count
There are currently 2 items in the list

Type something: print
The following items are in the list:
hello
goodbye

Type something: What’s up?
“What’s up?” has been added to the list

Type something: scramble
A random element has been moved to a new position

Type something: print
The following items are in the list:
goodbye
hello
What’s up?

Type something: clear
The list has been cleared

Type something: count
There are currently 0 items in the list

Type something: quit
Thanks for typing!
```