Homework 1 – Doubly Linked List

Due Tuesday, January 28th by 11:59PM

You’ve created the basics of a singly linked list data structure in class. For this homework, your job is to create and work with a Doubly Linked List. As discussed in class, this is a Linked Lists whose nodes contain references to both the previous node in the list (if available) and the next node in the list (if available).

Feel free to start with the work you've done for the Linked List practice exercises, altering the data structure to be doubly linked and adding the methods outlined below.

Feel free to include any helper methods you may need.

# Task Overview

This is a brief overview of the tasks you must complete for this assignment. Specifics are given in the corresponding sections later in this document. You should ***read the entire document*** before starting.

* Create a generic CustomLinkedNode<T> class that represents one node in a doubly linked list.
* Create a generic CustomLinkedList<T> class representing a doubly linked list.
* Implement the required methods outlined below, plus any others you think you’d need.
* Test your CustomLinkedList methods by implementing the specified user interaction in Main().

# CustomLinkedNode<T> Details

Before implementing the linked list itself, you’ll need a class which represents a generic node in the list. Your custom node class must be able to hold one piece of data, plus references to a previous and next node. Include the fields, properties and constructors necessary for this.

# CustomLinkedLink<T> Details

This class represents a doubly linked list data structure, holding references to the head and tail nodes.

## Fields

The class should have only three fields:

* head – the node at the beginning of the list
* tail – the node at the end of the list
* count – the number of nodes currently in the list

When there is nothing in the list, *count* is zero and both *head* and *tail* are null.

## Properties

Implement the following properties (and only the following properties) for your linked list. Remember that it's dangerous to return references to individual nodes, as that permits code outside of the list to alter those objects.

**int Count**

Create a get property for the count of the list

**Indexer**

Create a get and set indexer property for the list. These should allow a user to get or set data at a specified index in the list. If an invalid index is specified, throw an appropriate exception. (Remember that an indexer is a special kind of property. It is NOT a property simply *named* "indexer").

Note: You may already have a method called Get() or GetData() that does much of what this get indexer property will do. You are *replacing* that method with this property.

## Methods

Implement each of the methods described below. You probably have some of these started from practice exercises; you'll need to update each to account for the now doubly-linked nature of your list.

**void Add(T item)**

Create a node to hold the specified item, then add that node to the end of the list, updating the head and tail references as necessary. Remember to update the count property, too.

**void Insert(T item, int index)**

Create a node to hold the specified item, then insert that node into the list at the specified index. For example: if the specified index is 1, this new node should be the *second* node in the list (just beyond the head node), since index 0 would be the head node.

If the specified index is invalid (less than zero or greater than the count of the list), throw an appropriate exception with a meaningful error message. If the index is valid, place the node at the correct location in the list and ensure you're connecting it properly to the nodes that should come before and after it. Be sure you're handling the following cases (some may overlap depending on your implementation):

* Index is zero.
* Index is just after the last node in the list.
* Index is somewhere in the middle of the list.
* The list is currently empty.

If you're having trouble visualizing how to "hook up" this new node to the other nodes around it, I suggest drawing a basic linked list out on paper or a whiteboard. Each "box" is a node object, and each "arrow" is a reference you'll need to handle in your code.

**T RemoveAt(int index)**

This method removes the node at the specified index from the list and returns that node's data (not the node itself). Depending on the index, you'll be altering the Next and/or Previous references of surrounding nodes. You may also need to update the head and/or tail node as necessary. Be sure your code handles the following cases (some may overlap depending on your implementation):

* Invalid index: Throw an appropriate exception
* The index is zero.
* The index is the end of the list.
* The index is somewhere in the middle of the list.
* The node you're removing is the very last item remaining in the list.

Again, if you're having trouble visualizing what you need to do for each case, draw it out.

**void Clear()**

Clears the list, resetting it to an empty state. This can be as easy as updating the three fields of the class to their original values.

**void PrintReversed()**

This method should utilize the “previous” field of each node to print out all of the data in reverse order. This will help to test if all of your “arrows” point to the correct “boxes”.

# Main Method

Your main method should have a loop that will get user input. Check for special commands typed by the user and perform the corresponding action. If the input is not a command, add it to the list instead.

|  |  |
| --- | --- |
| **Command** | **Action** |
| q or quit | End the loop |
| print | Print everything in the list – Use the *get* indexer in a loop for this |
| count | Print the number of items in the list |
| clear | Clear the entire list |
| remove | *Randomly* remove one element from the list. To aid in debugging, print out the data and the randomly generated index. |
| reverse | Print the list out in reverse order |
| scramble | Remove a single *random* element from the list and insert it back into the list at another *random* index. Print out the randomly generated indices and data first. |

# Sample Output

**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

Moving a random element to a new, random position

- Removing goodbye from index 0

- Inserting into index 1

**Type something:** print

The following items are in the list:

goodbye

hello

What’s up?

**Type something:** reverse

The following items are in the list (in reverse order):

What’s up?

hello

goodbye

**Type something:** remove

Removed hello from index 1

**Type something:** print

The following items are in the list:

goodbye

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!