PG 2 – LAB 4: LINKED LISTS

CONTENTS

OBJECTIVES

Create generic classes. Learn the differences between a stack and a queue. Write code to implement the stack and queue behavior. Traverse and reverse a stack and a queue.

Topics Covered

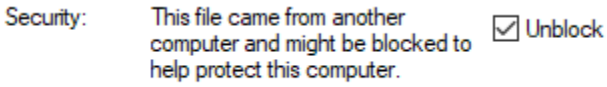
Stack, Queue, Linked List, Generics

PROJECT SETUP

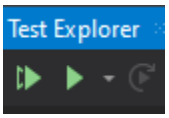
For this lab, you will be given the projects that you need: a **C# class library (.NET Framework)** and a **Unit Test Project**. You do NOT need to create any projects.

You need to **unblock** the zip file after downloading it from FSO.

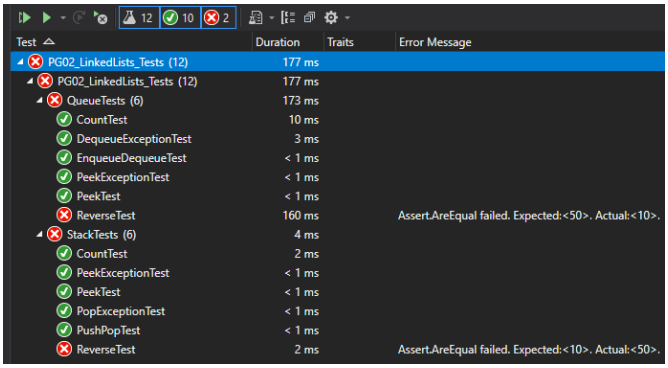
1. Right-click the zip file and select "Properties"
2. check the "Unblock" checkbox and click Ok



3. unzip the file
4. open the solution. In the solution explorer, right-click the solution and select “Rebuild Solution”
5. Open the “Test Explorer”. If you don’t see the tab at the bottom of Visual Studio, you can open it from the menu at “Tests->Test Explorer”.
6. In the Test Explorer, click the “Run All Tests” button



7. NOTE: 2 of the tests fail out of the box. That is expected.



You will create your classes in the class library.

Hint

A good way to approach this lab is to start with the unit tests.

For instance, you create the public PG2Stack class (no methods) in the class library then plug that one into one of the unit tests. (replace Stack with PG2Stack)

That should instantly give you a red-squig in the unit test for the method that it is calling.

You can have visual studio generate the signature of the method for you! 😊

Open the light-bulb for the red-squig and select the option to generate the method.

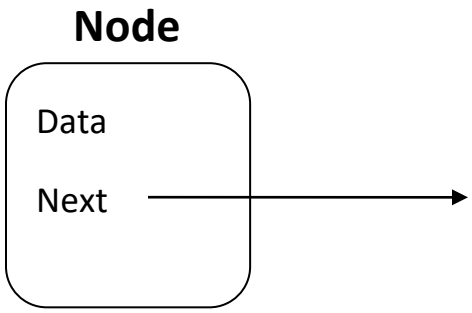
You'll find the method in your class now!

NOTE: the method signatures that Visual Studio generates might not be generic (they might be specific to ints). You'll need to make them generic.

PART 4-1

Node Class

You need a **Node** class to hold the **Data** and link for the **Next** node in the lists. This class can be used for both the Stack and the Queue. The Node class should be able to hold any data type so make the class generic. For this class, it is ok to make the Data and Next **public fields**.



GRADING: 10 POINTS

COMMON MISTAKES:

-2: the type of Next in the Node class should be Node<T>

-2: Data should have T as its type

-4: the Node class was not made generic

PART 4-2

4-2.1 The PG2Stack Class

Create a class that behaves like a Stack data structure.

- Name your stack class **PG2Stack** so you will not have a naming conflict with .NET's Stack.
- Your stack class must be a **generic class** meaning that it should be able to take any type for its data.
- A **Count** property to keep track of how many items in the list. Make sure the **setter is private**.
- A field for the **_head** node to keep track of the first node in the linked list

GRADING: 5 POINTS

COMMON MISTAKES:

-1: do not make the _head public. It exposes the structure of your linked list.

-1: the stack should not have a _tail field. It is not used.

-1: Count should be a property

4-2.2 Push method

Create a **Push** method to **add a new node** to the head of the stack. You will need to create a new node and connect it (set the next of the new node) to the head of the stack. Make sure you **update the Count**.

GRADING: 10 POINTS

COMMON MISTAKES:

-3: Push should be adding the new node to the front of the stack (newNode.Next = _head) which is O(1) performance. Your algorithm is putting it at the end of the stack which is O(N) performance because you're looping over all the nodes.

-1: Push should increment count

4-2.3 Pop method

Create a **Pop** method that **removes a node** from the head of the stack and **returns the data** that was in the node that was removed. If a node is removed from the linked list, make sure you **update the Count**.

If the stack is empty, you must throw an InvalidOperationException. Do not throw a NullReferenceException.

GRADING: 10 POINTS

COMMON MISTAKES:

-3: Pop should be removing the head from the stack which would be O(1) performance. Your algorithm is removing the back of the stack which is O(N) performance because you're looping over all the nodes.

-5: Pop is not removing the head node correctly. It should set head = head.next. But first it should check if head == null and if so throw an exception.

-2: Pop should return T, not Node<T>. So if the head is not null, return Data that is in _head

-1: Pop should decrement count

-2: Pop should return the data in the head node. The return type of Pop should be T.

-2: Pop should throw an exception if the stack is empty

4-2.4 Peek method

Create a **Peek** method **returns the data** in the head node on the stack.

If the stack is empty, you must throw an InvalidOperationException. Do not throw a NullReferenceException.

GRADING: 5 POINTS

COMMON MISTAKES:

-2: Peek should throw an exception if the stack is empty

-1: Peek should return the data in the head node. The return type of Peek should be T.

-2: Peek should not be creating a new node. First it should check if head == null and if so throw an exception. If not, then return the data in the head node.

-2: Peek should return T, not Node<T>. So if the head is not null, return head.Data. The method should not have a parameter passed to it.

-2: Peek should throw an exception if the stack is empty

4-2.5 Reverse method

Create a **Reverse** method to reverse the nodes in the stack. You can solve this iteratively (like with a while loop) or recursively. For example, if your stack has 5,4,3,2,1 in it, after reversing the nodes would be re-ordered to be 1,2,3,4,5.

GRADING: 10 POINTS

COMMON MISTAKES:

-5: Reverse should point the node's next to the previous node for each node in the stack. So you would keep track of the previous node (initially null), the current node (initially the _head) and the next. The next is like a temp variable that holds the current's next, set current's next = previous, set previous = current, set current = next.

-1: the Reverse method should not have a parameter. It should work on the current instance.

-2: Reverse should not use another stack or queue to reverse the nodes. That is an inefficient way.

PART 4-3

4.3-1 The PG2Queue Class

Create a class that behaves like a Queue data structure.

- Name your queue class **PG2Queue** so you will not have a naming conflict with .NET's Queue.
- Your queue class must be a **generic class** meaning that it should be able to take any type for its data.
- A **Count** property to keep track of how many items in the list. Make sure the **setter is private**
- A field for the **_head** node to keep track of the first node in the linked list
- A field for the **_tail** node to keep track of the last node in the linked list

GRADING: 5 POINTS

COMMON MISTAKES:

-1: do not make the _head and _tail public. It exposes the structure of your linked list.

-1: Count should be a property

-1: the setter for Count should be private

4-3.2 Enqueue method

Create an **Enqueue** method to **add a node** to the tail of the queue. You will need to create a new node and make it the tail node of the linked list. **HINT**: you'll need to connect the tail to the new node before making the new node be the tail. Make sure you **update the Count** when a new node is added to the linked list.

When the new node is the first node in the queue, the head must point to the new node as well.

GRADING: 10 POINTS

COMMON MISTAKES:

-2: in Enqueue, you need to put _tail.Next = nodeToAdd in an else block. Otherwise, the first node is pointing to itself when adding the first item.

-5: Enqueue should be putting the new node at the end of the list. If count != 0, then set tail.next = newNode then tail = newNode

-1: Enqueue should increment count

4-3.3 Dequeue method

Create a **Dequeue** method that **removes a node** from the head of the queue and **returns the data** that was in the node that was removed. If a node is removed from the queue, make sure you **update the Count**. If the count becomes 0, you also need to update the tail and set it to null (it means the queue is empty so both head and tail need to be null).

If the queue is empty, you must throw an InvalidOperationException. Do not throw a NullReferenceException.

GRADING: 10 POINTS

COMMON MISTAKES:

-2: Dequeue needs to set the tail to null if the head becomes null after moving the head

-2: Dequeue should return the data in the head node. The return type of Pop should be T.

-1: Dequeue should decrement count

-2: Dequeue should throw an exception if the queue is empty

4-3.4 Peek method

Create a **Peek** method **returns the data** in the head node on the queue without removing it.

If the queue is empty, you must throw an InvalidOperationException. Do not throw a NullReferenceException.

GRADING: 5 POINTS

COMMON MISTAKES:

-2: Peek should throw an exception if the queue is empty

-1: Peek should return the data in the head node. The return type of Peek should be T.

-2: Peek should not be creating a new node. First it should check if head == null and if so throw an exception. If not, then return the data in the head node.

-2: Peek should return T, not Node<T>. So if the head is not null, return head.Data. The method should not have a parameter passed to it.

4-3.5 Reverse method

Create a **Reverse** method to reverse the nodes in the queue. You can solve this iteratively (like with a while loop) or recursively. For example, if your queue has 5,4,3,2,1 in it, after reversing the nodes would be re-ordered to be 1,2,3,4,5.

GRADING: 10 POINTS

COMMON MISTAKES:

-5: Reverse should point the node's next to the previous node for each node in the stack. So you would keep track of the previous node (initially null), the current node (initially the _head) and the next. The next is like a temp variable that holds the current's next, set current's next = previous, set previous = current, set current = next.

-1: the Reverse method needs to update _tail also.

-1: the Reverse method should not have a parameter. It should work on the current instance.

-2: Reverse should not use another stack or queue to reverse the nodes. That is an inefficient way.

PART 4-4

Unit Tests

Make sure that the unit tests pass. The current unit tests that are provided are written for the .NET Stack and Queue.

You must change the unit tests to use your PG2Stack and PG2Queue classes.

Search the unit tests for the `//TODO:` comments and make the changes.

GRADING: 10 POINTS

COMMON MISTAKES:

-5: the unit tests were not modified to work with your classes

-2: you need to create your classes in the class library

-2: the unit tests do not build

-1: the reverse unit tests were not modified to work with your classes

LAB 4: RUBRIC

PART	FEATURE	VALUE
4-1	Node Class	10
4-2.1	The PG2Stack Class	5
4-2.2	Push method	10
4-2.3	Pop method	10
4-2.4	Peek method	5
4-2.5	Reverse method	10
4-3.1	The PG2Queue Class	5
4-3.2	Enqueue method	10
4-3.3	Dequeue method	10
4-3.4	Peek method	5
4-3.5	Reverse method	10
4-4	Unit Tests	10
	TOTAL	100