CSC-223
Assignment 1
Work in a Pair

A Doubly Linked List Implementation

One of the most important linear data structures is the doubly linked list; it tests your skills as a programmer but also provides a solid foundation for many other data structures (e.g., stack, queue, etc.). This first assignment will ask you to implement the core functionality of a doubly linked list; our next assignment will add to it.

What You Need to Do: Continuing from the Last Assignment

Copy your project from the last assignment into a new folder for this assignment. This creates a backup that protects your work and allows you to experiment freely without risk.

You will work with your partner using GitHub; see below for more details.

Git and Github Collaboration: Getting Started with Your Partner

Repository Setup. Begin your collaboration by designating one partner as the owner of the repository. This person will take the lead in creating the foundation for your project on GitHub. The repository owner should create a new GitHub repository with a descriptive README explaining the purpose of the project, appropriate `.gitignore` settings tailored for C# development, and suitable license files. For C# projects, your `.gitignore` should exclude build outputs in the `bin/` and `obj/` directories, IDE-specific files like those in the `.vs/` directory, and NuGet package directories (NuGet contains compiled source code). However, you should ensure all source code (`.cs` files), project files (`.csproj`), solution files (`.sln`), and configuration files like `appsettings.json` are included in the repository. Feel free to use an LLM to generate such a file; make sure to have the LLM focus on C# and the type of information in this paragraph.

Once the repository is established, the owner should invite their partner as a collaborator through the GitHub repository settings. Navigate to `Settings > Collaborators and teams`, and add your partner using their GitHub username or email address. Your partner will receive a notification and should accept the invitation to gain access to the repository.

Collaboration Workflow. Establishing an effective branching strategy is crucial for smooth collaboration. Create a main branch to hold stable, working code that serves as the source of truth for your project. For individual tasks, use feature branches with descriptive names that indicate the functionality being developed. You might also consider implementing a development branch that serves as an integration point before merging changes into the main branch.

Work distribution can be managed through GitHub Issues, which allow you to track tasks and assign them to specific team members. Use labels to categorize issues based on their nature, such as bugs, enhancements, or documentation needs. Organization helps maintain clarity about who is responsible for what and the status of each task.

The code contribution process begins when you clone the repository to your local machine. Create a branch for your assigned task, make your changes, and commit regularly with descriptive messages that explain what you have modified and why. Push your branch to GitHub when ready and create a Pull Request (PR) for your partner to review. During the review process, address any feedback through additional commits until your partner approves the changes, at which point you can merge the PR into the main branch.

Communication & Project Management. Intra-group code reviews form a critical part of your collaborative process. Use GitHub's review features to provide specific, constructive feedback on any and all code. Comment on particular lines that need attention and only approve Pull Requests when you are fully satisfied with the changes.

Track your overall progress using GitHub Projects or Milestones. Update issue statuses as work progresses and close them when completed to maintain an accurate picture of your project's status. This visibility helps both partners understand what's been accomplished and what remains to be done.

Documentation should be maintained throughout the project lifecycle. Keep setup instructions, coding standards, and project structure information up-to-date in your repository. For more extensive documentation, consider using GitHub's Wiki feature, which provides a dedicated space for comprehensive project information. Such documentation can be beneficial for an outsider who wants to investigate or use your project; think about it as creating an artifact that can be referenced when seeking an internship or a job.

Best Practices. Throughout your collaboration, commit your changes regularly with clear, descriptive messages that help your partner understand your intentions. Pull frequently from the remote repository to stay updated with your partner's contributions and avoid conflicts. Communicate openly about any challenges you encounter so that you can work together to find solutions.

Test your code thoroughly before submitting Pull Requests to ensure you are contributing high-quality work. Document important decisions and discussions within your repository or Wiki so that you maintain a shared understanding of the project's direction. By embracing these collaborative practices, you will establish an efficient workflow that leverages GitHub's features effectively and sets your project up for success.

What You Need to Do: Adding to the Project Structure

As shown in Figure 1, create a `src/Utilities/Containers` folder as well as `src/Utilities/Containers.Tests`. All code for this assignment will be in `src/Utilities/containers/DLL.cs` as well as the corresponding test file.

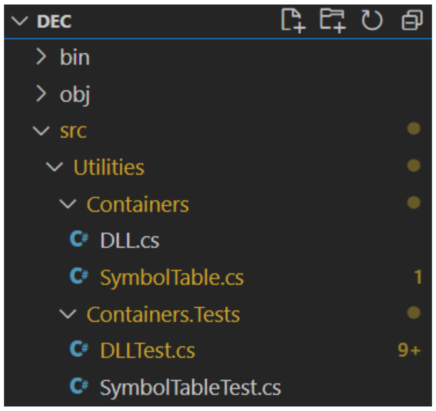

Figure 1: The desired project structure including DLL and its corresponding tests.

What You Need to Do: `DLL` Class

For this assignment, you are to implement a doubly linked list class called `DLL` along with support functionality as described in

Table **1**. Note that your implementation of `DLL` must use head / tail sentinel nodes. You will also need to implement a `DNode` class to facilitate the `DoublyLinkedList` class. `DNode` should implement only a constructor and define attributes for left / right pointers and storing a value.

Recall that the class we are implementing is a data structure that could store data of any type; therefore, you should be implementing a generic structure:

```
public class DLL<T> : IEnumerable<T>, IList<T>
```

Observe that the `DLL<T>` class implements both `IEnumerable<T>` and `IList<T>` because it transforms the specialized linked list into a standard .NET collection that works naturally with C#'s ecosystem. These interfaces enable key language features like foreach loops and indexer access (e.g., `list[0]`), provides LINQ compatibility, and allows the class to hide its complex implementation details behind a familiar abstraction that C# developers expect. Without `IEnumerable<T>`, the linked list would not support iteration patterns, and without `IList<T>`, it would not provide the convenient indexed access that makes arrays and lists so useful. With both interfaces implemented, the linked list becomes a first-class citizen in the .NET ecosystem rather than remaining a specialized data structure that requires custom access patterns.

Table 1: Methods for the `DLL` class implemented with head / tail sentinel nodes.

Return Type	Method / Property	Description	Exceptions Thrown
`None`	`DLL()`	Constructor that properly initializes an instance of this class with sentinel nodes.	
`None`	`Insert(DNode node, T item)`	Private helper method that inserts a new node with the specified item before the given node.	
`None`	`Remove(DNode node)`	Private helper method that removes the specified node from the list.	
`DNode`	`GetNode(int index)`	Private helper method that returns the node at the specified index.	`ArgumentOutOfRangeException`: Thrown when the index is negative or greater than or equal to the size of the list.
`bool`	`Contains(T item)`	Determines whether the list contains the specified `item`.	
`int`	`Size()`	Returns the number of items in the list.	
`String`	`ToString()`	Returns a string representation of the list for debugging.	
`bool`	`Remove(T item)`	Deletes the first occurrence of `item` in the list; returns whether deletion occurred or not.	
`T`	`Front()`	Returns the element stored at the first *valid* node in the list.	`InvalidOperationException`: Thrown when attempting to access

			the first or last element of an empty list.
`T`	`Back()`	Returns the element stored at the last *valid* node in the list.	`InvalidOperationException`: Thrown when attempting to access the first or last element of an empty list.
`void`	`PushFront(T item)`	Adds the element to the beginning of the list.	
`void`	`PushBack(T item)`	Adds the element to the end of the list.	
`T`	`PopFront()`	Removes and returns the element stored at the first valid `DNode` in the list.	`InvalidOperationException`: Thrown when attempting to remove and return an element from an empty list.
`T`	`PopBack()`	Removes and returns the element stored at the last valid `DNode` in the list.	`InvalidOperationException`: Thrown when attempting to remove and return an element from an empty list.
`None`	`clear()`	Removes all nodes in the list (except the sentinel nodes).	
`bool`	`IsEmpty()`	Returns if this list contains any values (or not).	
`IList<T>`			
`int`	`Count { get; }`	A property that returns the current number of elements in the linked list.	
`bool`	`IsReadOnly { get; }`	This property returns false in this implementation, indicating that the linked list allows adding, removing, and modifying elements.	
`void`	`Add(T item)`	Appends an `item` to the end of the list.	
`void`	`Insert(int index, T item)`	Places a new item at a specified index position, 'shifting' subsequent elements down the list.	`ArgumentOutOfRangeException`: Thrown when the index is negative or greater than the count of elements in the list (note that inserting at Count is allowed, as it's equivalent to adding at the end).
`int`	`IndexOf(T item)`	Returns the index of the first occurrence of the specified item, or `-1` if not found.	
`T`	`this[int index] { get; set; }`	Indexer property that gets or sets the element at the specified index.	

`void`	`RemoveAt(int index)`	Deleted the element stored at the specified index.	`ArgumentOutOfRangeException`: Thrown when the index is negative or greater than or equal to the count of elements in the list.
`void`	`CopyTo(T[] array, int arrayIndex)`	Transfers elements from the doubly-linked list directly into a target array starting at a specified index, providing a way to efficiently populate arrays with the list's contents.	`ArgumentNullException`: Thrown when the target array is null. `ArgumentOutOfRangeException`: Thrown when the array index is negative. `ArgumentException`: Thrown when there is not enough space in the array starting at the specified index to accommodate all elements from the list.
		`IEnumerable<T>`	
`IEnumerator<T>`	`GetEnumerator()`	This method returns a typed iterator that yields each data element in the linked list in sequence.	
`IEnumerator`	`IEnumerable.GetEnumerator()`		

Some C# Implementation Specifics: `EqualityComparer`

In this assignment, you will need to compare two objects in methods such as `IndexOf`; to do so we use an `Equals` method:

`bool Equals (T x, T y)`

This method takes two parameters of type T and returns a boolean value indicating whether they are equal according to the default equality comparison logic for that type. The method signature abstracts away the complexities of determining equality for different types while maintaining type safety. When using this method in your code, you do not need to worry about null reference exceptions or type-specific comparison details, as the default comparer handles these concerns automatically.

To use the `Equals` method, we actually need to use the expression path `EqualityComparer<T>.Default.Equals` to access the C# `Equals` method. It automatically uses the most appropriate equality comparison for the given type, whether that means calling an overridden `Equals` method, using `IEquatable<T>`, or falling back to `Object.Equals`. This built-in approach saves you from writing custom comparison logic while ensuring type-safe comparisons. Sample usage is shown below that is unrelated to our doubly linked list.

```
public bool Contains<T>(T[] array, T item)
{
    foreach (T element in array)
    {
        if (EqualityComparer<T>.Default.Equals(element, item)) return true;
    }
    return false;
}
```

Some C# Implementation Specifics: `ToString` and `StringBuilder`

Just like `__str__` in Python, the `ToString` method in C# provides a customized, string representation of an object that can be used for display, debugging, or text output purposes. When implemented for a data structure like a doubly linked list, it creates a meaningful text visualization of the contents of the structure. Rather than using the default representation (which typically returns the type name), a well-implemented `ToString` method makes the data more human-readable and helps understand what is stored in the data structure at a glance; such information is often made available by the debugger when a `ToString` method is implemented.

To implement the `ToString` method, we often use a `StringBuilder` object as an accumulator for string operations, collecting and modifying text (without creating new string objects with each change). This approach offers significant performance benefits over traditional string concatenation, especially when dealing with numerous modifications or larger data structures, as it eliminates the overhead of repeatedly allocating and deallocating memory for new string objects. The final string is only created once when the `ToString` method is called on the `StringBuilder` instance, making it particularly well-suited for scenarios where text is built incrementally through multiple operations. For `ToString` methods of data structures, we typically:

1. Create a new `StringBuilder` instance
2. Iterate through the linked list nodes and appends the value of each node to the `StringBuilder`
3. Last, we convert (and return) the final `StringBuilder` to a `string` using `StringBuilder`'s own `ToString` method.

What You Need to Do: Testing

`DoublyLinkedList` should be a standalone data structure implementation that needs to be rock solid; failure to test well will only cause subsequent problems for you. All tests should be xUnit tests and adhere to the norms of xunit including using `[Fact]` and `[Theory]`.

New to CSC-223! LLMs can be helpful tools for generating testing code for your assignments. When using an LLM to create tests, include prompts like "write thorough tests for all methods", "make sure to test with duplicate data as well" to ensure comprehensive coverage, and "use `[Theory]` and `InlineData` to avoid redundancy in testing code". ***However, be warned that generated code is not always perfect—your professor has encountered several cases where tests were written incorrectly for the corresponding code.*** Always review each test carefully to verify it is testing what you intend and functioning properly with your implementation. It is also fair to ask the LLM to see if any cases have been missed in the tests that it generates. In fact, it is advised that you ask an LLM to 'check its work'; do not accept a first answer.

What You Need to Do: Commenting

Comment code consistent with the standards stated in the first assignment: when needed, use an LLM to generate comments then refine as needed.

Submitting: Source Code

You will demonstrate your source code to the instructor, in person. Be ready to demonstrate successful execution of all unit tests and describe all tests and code you were required to implement.