# Practical Task 5.1

(Pass Task)

Submission deadline: 10:00am Monday, August 26
Discussion deadline: 10:00am Saturday, September 14

## General Instructions

The objective of this task is to study implementation of a *Doubly Linked List*, a generic data structure capable to maintain an arbitrary number of data elements and support various standard operations to read, write, and delete data. Compared to other popular data structures, linked list like data structures offer a number of advantages with respect to time complexity and practical application. For example, where an array-based data structure, such as a simple list (or a vector), requires a contiguous memory location to store data, a linked list may record new data elements anywhere in the memory. This is achievable by encapsulation of a payload (the user's data record) into a node, then connecting nodes into a sequence via memory references (also known as links). Because of this, a linked list is not restricted in size and new nodes can be added increasing the size of the list to any extent. Furthermore, it is allowed to use the first free and available memory location with only a single overhead step of storing the address of memory location in the previous node of a linked list. This makes insertion and removal operations in a linked list of a constant $O(1)$ time; that is, as fast as possible. Remember that these operations generally run in a linear $O(\mathrm{n})$ time in an array since memory locations are consecutive and fixed.

A doubly linked list outperforms a singly linked list achieving better runtime for deletion of a given data node as it enables traversing the sequence of nodes in both directions, i.e. from starting to end and as well as from end to starting. For a given a node, it is always possible to reach the previous node; this is what a singly linked list does not permit. However, these benefits come at the cost of extra memory consumption since one additional variable is required to implement a link to previous node. In the case of a simpler singly linked list, just one link is used to refer to the next node. However, traversing is then possible in one direction only, from the head of a linked list to its end.

1. To start, follow the link below and explore the functionality of the LinkedList<T> generic class available within the Microsoft .NET Framework.

   https://msdn.microsoft.com/en-au/library/he2s3bh7(v=vs.110).aspx.

   Because some operations that you are asked to develop in this task are similar to those in the LinkedList<T>, you may refer to the existing description of the class to get more insights about how your own code should work.

2. Explore the source code attached to this task. Create a new Microsoft Visual Studio project and import the DoublyLinkedList.cs file. This file contains a template of the DoublyLinkedList<T> class. The objective of the task is to develop the missing functionality of the class to obtain a fully-functional data structure. Subsequently, import the Tester.cs file to the project to enable the prepared Main method important for the purpose of debugging and testing the expected program class and its interfaces for runtime and logical errors.

3. Find the nested Node<K> class presented inside the DoublyLinkedList<T> and learn its structure. This is a generic class whose purpose is to represent a node of a doubly linked list. Think about it as an atomic data structure itself that serves the DoublyLinkedList<T> as a building block. In fact, a doubly linked list is a linear collection of data elements, whose order is not given by their physical positions in memory, for example like in arrays. Instead, each element points to the next (and the previous) one. It is a data structure consisting of a set of nodes which together represent a sequence. Generally, a node of a doubly linked list consists of a *data record* that holds a payload and *two auxiliary pointers* referring to the

preceding and succeeding nodes in the ordered sequence of nodes constituting the linked list. The two pointers allow to navigate back and forth between two adjacent nodes.

Note that the Node<K> class is ready for you to use. It provides the following functionality:

- **Node(K value, Node<K> previous, Node<K> next)**

  Initializes a new instance of the Node<K> class, containing the specified *value* and referring to *previous* and *next* arguments as nodes before and after the new node, respectively, in the sequence of the associated doubly linked list.

- **K Value**

  Property. Gets or sets the value (payload) of type K contained in the node.

- **Node<K> Next**

  Property. Gets a reference to the next node in the DoublyLinkedList<T>, or null if the current node is the last element of the DoublyLinkedList<T>.

- **Node<K> Previous**

  Property. Gets a reference to the previous node in the DoublyLinkedList<T>, or null if the current node is the first element of the DoublyLinkedList<T>.

- **string ToString()**

  Returns a string that represents the current Node<K>. ToString() is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display.

You may have already noticed that the Node<K> implements the INode<K> interface, which is available in the attached INode.cs file. The reason for the use of the interface is that the Node<K> is a data structure internal to the DoublyLinkedList<T> class, thus an instance of the Node<K> must not be exposed to the user. It must be hidden to protect an instance of the DoublyLinkedList<T> from potential corruption caused by the user's activities. However, because a user needs access to the data that the user owns and stores inside an instance of the DoublyLinkedList<T>, the Node<K> implements the interfaces that permits to read and set (write) the data. Check the INode<K> and see that the only property it implies is *Value* of generic type K.

4. Proceed with the given template of the DoublyLinkedList<T> class and explore the methods that it has implemented for you for the purpose of example, in particular:

   - **DoublyLinkedList()**

     Initializes a new instance of the DoublyLinkedList<T> class that is empty.

   - **First**

     Property. Gets the first node of the DoublyLinkedList<T>. If the DoublyLinkedList<T> is empty, the First property returns null.

   - **Last**

     Property. Gets the last node of the DoublyLinkedList<T>. If the DoublyLinkedList<T> is empty, the Last property returns null.

   - **Count**

     Property. Gets the number of nodes actually contained in the DoublyLinkedList<T>.

   - **INode<T> After(INode<T> node)**

     Returns the node casted to the INode<T> that succeeds the specified node in the DoublyLinkedList<T>. If the node given as parameter is null, it throws the ArgumentNullException. If the parameter is not in the current DoublyLinkedList<T>, the method throws the InvalidOperationException.

   - **public INode<T> AddLast(T value)**

     Adds a new node containing the specified value at the end of the DoublyLinkedList<T>. Returns the new node casted to the INode<T> with the recorded value.

- **INode<T> Find(T value)**

  Finds the first occurrence in the DoublyLinkedList<T> that contains the specified value. The method returns the node casted to INode<T>, if found; otherwise, null. The DoublyLinkedList<T> is searched forward starting at *First* and ending at *Last*.

- **string ToString()**

  Returns a string that represents the current DoublyLinkedList<T>. ToString() is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display.

As part of the prepared DoublyLinkedList<T> class, you can also observe a number of private properties and methods. An important aspect of the DoublyLinkedList<T> is the use of two auxiliary nodes: the *Head* and the *Tail*. The both are introduced in order to significantly simplify the implementation of the class and make insertion functionality reduced just to a single method designated here as

Node<T> AddBetween(T value, Node<T> previous, Node<T> next)

In fact, the Head and the Tail are invisible to a user of the data structure and are always maintained in it, even when the DoublyLinkedList<T> is formally empty. When there is no element in it, the Head refers to the Tail, and vice versa. Note that in this case the First and the Last properties are set to null. The first added node therefore is to be placed in between the Head and the Tail so that the former points to the new node as the *Next* node, while the latter points to it as the *Previous* node. Hence, from the perspective of the internal structure of the DoublyLinkedList<T>, the First element is the next to the Head, and similarly, the Last element is previous to the Tail. Remember about this crucial fact when you design and code other functions of the DoublyLinkedList<T> in this task.

The given template of the DoublyLinkedList <T> class should help you with development of its remaining methods. Therefore, explore the existing code as other methods are to be similar in terms of logic and implementation.

5. You must complete the DoublyLinkedList<T> and provide the following functionality to the user:

   - **INode<T> Before(INode<T> node)**

     Returns the node, casted to the INode<T>, which precedes the specified node in the DoublyLinkedList<T>. If the node given as parameter is null, the method throws the ArgumentNullException. If the parameter is not in the current DoublyLinkedList<T>, the method throws the InvalidOperationException.

   - **INode<T> AddFirst(T value)**

     Adds a new node containing the specified value at the start of the DoublyLinkedList<T>. Returns the new node casted to the INode<T> containing the value.

   - **INode<T> AddBefore(INode<T> before, T value)**

     Adds a new node before the specified node of the DoublyLinkedList<T> and records the given value as its payload. It returns the newly created node casted to the INode<T>. If the node specified as an argument is null, the method throws the ArgumentNullException. If the node specified as argument does not exist in the DoublyLinkedList<T>, the method throws the InvalidOperationException.

   - **INode<T> AddAfter(INode<T> after, T value)**

     Adds a new node after the specified node of the DoublyLinkedList<T> and records the given value as its payload. It returns the newly created node casted to the INode<T>. If the node specified as argument is null, the method throws the ArgumentNullException. If the node specified as argument does not exist in the DoublyLinkedList<T>, the method throws the InvalidOperationException.

   - **void Clear()**

     Removes all nodes from the DoublyLinkedList<T>. Count is set to zero. For each of the nodes, links to the previous and the next nodes must be nullified.

   - **void Remove(INode<T> node)**

     Removes the specified node from the DoublyLinkedList<T>. If node is null, it throws the ArgumentNullException. If the node specified as argument does not exist in the DoublyLinkedList<T>, the method throws the InvalidOperationException.

- **void RemoveFirst()**

  Removes the node at the start of the DoublyLinkedList<T>. If the DoublyLinkedList<T> is empty, it throws InvalidOperationException.

- **void RemoveLast()**

  Removes the node at the end of the DoublyLinkedList<T>. If the DoublyLinkedList<T> is empty, it throws InvalidOperationException.

Note that you are free in writing your code that is private to the DoublyLinkedList<T> unless you respect all the requirements in terms of functionality and signatures of the specified methods.

6. As you progress with the implementation of the DoublyLinkedList <T> class, you should start using the Tester class to thoroughly test the DoublyLinkedList<T> aiming on the coverage of all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the vector class. The given version of the testing class covers only some basic cases. Therefore, you should extend it with extra cases to make sure that your doubly linked list class is checked against other potential mistakes.

## Further Notes

- Learn the material of chapters 3.4 and especially that of section 7.3.3 of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser. You may access the book on-line for free from the reading list application in CloudDeakin available in Resources → Additional Course Resources → Resources on Algorithms and Data Structures → Course Book: Data structures and algorithms in Java. As a complementary material, to learn more about a singly linked and doubly linked lists, you may refer to Chapter 2 of SIT221 Workbook available in CloudDeakin in Resources → Additional Course Resources → Resources on Algorithms and Data Structures → SIT221 Workbook.
- If you still struggle with such OOP concepts as Generics and their application, you may wish to read Chapter 11 of SIT232 Workbook available in Resources → Additional Course Resources → Resources on Object-Oriented Programming. You may also have to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you need excellent understanding of these topics to progress well through the practical tasks of the unit. Make sure that you are proficient with them as they form a basis to design and develop programming modules in this and all the subsequent tasks. You may find other important topics required to complete the task, like exceptions handling, in other chapters of the workbook.
- We will test your code in Microsoft Visual Studio 2017. Find the instructions to install the community version of Microsoft Visual Studio 2017 available on the SIT221 unit web-page in CloudDeakin at Resources → Additional Course Resources → Software → Visual Studio Community 2017. You are free to use another IDE if you prefer that, e.g. Visual Studio Code. But we recommend you to take a chance to learn this environment.

## Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your program implements all the required functionality, is compliable, and has no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit your program code as an answer to the task via OnTrack submission system.
- Meet with your marking tutor to demonstrate and discuss your program in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.

- Answer all additional (theoretical) questions that your tutor can ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this compulsory interview part. Please, come prepared so that the class time is used efficiently and fairly for all the students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not check your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work through the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.

## Expected Printout

This section displays the printout produced by the attached Tester class, specifically by its *Main* method. It is based on our solution. The printout is provided here to help with testing your code for potential logical errors. It demonstrates the correct logic rather than an expected printout in terms of text and alignment.

```
Test A: Create a new list by calling 'DoublyLinkedList<int> vector = new DoublyLinkedList<int>( );'
 :: SUCCESS: list's state []


Test B: Add a sequence of numbers 2, 6, 8, 5, 1, 8, 5, 3, 5 with list.AddLast( )
 :: SUCCESS: list's state [{XXX-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-8},{1-(8)-5},{8-(5)-3},{5-(3)-5},{3-(5)-XXX}]


Test C: Remove sequentially 4 last numbers with list.RemoveLast( )
 :: SUCCESS: list's state [{XXX-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]


Test D: Add a sequence of numbers 10, 20, 30, 40, 50 with list.AddFirst( )
 :: SUCCESS: list's state [{XXX-(50)-40},{50-(40)-30},{40-(30)-20},{30-(20)-10},{20-(10)-2},{10-(2)-6},{2-(6)-8},{6-(8)-5},{8-
(5)-1},{5-(1)-XXX}]


Test E: Remove sequentially 3 last numbers with list.RemoveFirst( )
 :: SUCCESS: list's state [{XXX-(20)-10},{20-(10)-2},{10-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]


Test F: Run a sequence of operations:
list.Find(40);
 :: SUCCESS: list's state [{XXX-(20)-10},{20-(10)-2},{10-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]
list.Find(0);
 :: SUCCESS: list's state [{XXX-(20)-10},{20-(10)-2},{10-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]
list.Find(2);
 :: SUCCESS: list's state [{XXX-(20)-10},{20-(10)-2},{10-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]


Test G: Run a sequence of operations:
Add 100 before the node with 2 with list.AddBefore(2,100)
 :: SUCCESS: list's state [{XXX-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-6},{2-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]
```

```
Add 200 after the node with 2 with list.AddAfter(2,200)
 :: SUCCESS: list's state [{XXX-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-6},{200-(6)-8},{6-(8)-5},{8-(5)-
1},{5-(1)-XXX}]
Add 300 before node list.First with list.AddBefore(list.First,300)
 :: SUCCESS: list's state [{XXX-(300)-20},{300-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-6},{200-(6)-
8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]
Add 400 after node list.First with list.AddAfter(list.First,400)
 :: SUCCESS: list's state [{XXX-(300)-400},{300-(400)-20},{400-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-
6},{200-(6)-8},{6-(8)-5},{8-(5)-1},{5-(1)-XXX}]
Add 500 before node list.First with list.AddBefore(list.Last,500)
 :: SUCCESS: list's state [{XXX-(300)-400},{300-(400)-20},{400-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-
6},{200-(6)-8},{6-(8)-5},{8-(5)-500},{5-(500)-1},{500-(1)-XXX}]
Add 600 after node list.First with list.AddAfter(list.Last,600)
 :: SUCCESS: list's state [{XXX-(300)-400},{300-(400)-20},{400-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-
6},{200-(6)-8},{6-(8)-5},{8-(5)-500},{5-(500)-1},{500-(1)-600},{1-(600)-XXX}]


Test H: Run a sequence of operations:
Remove the node list.First with list.Remove(list.First)
 :: SUCCESS: list's state [{XXX-(400)-20},{400-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-6},{200-(6)-
8},{6-(8)-5},{8-(5)-500},{5-(500)-1},{500-(1)-600},{1-(600)-XXX}]
Remove the node list.Last with list.Remove(list.Last)
 :: SUCCESS: list's state [{XXX-(400)-20},{400-(20)-10},{20-(10)-100},{10-(100)-2},{100-(2)-200},{2-(200)-6},{200-(6)-
8},{6-(8)-5},{8-(5)-500},{5-(500)-1},{500-(1)-XXX}]
Remove the node list.Before, which is before the node containing element 2, with list.Remove(list.Before(...))
 :: SUCCESS: list's state [{XXX-(400)-20},{400-(20)-10},{20-(10)-2},{10-(2)-200},{2-(200)-6},{200-(6)-8},{6-(8)-5},{8-(5)-
500},{5-(500)-1},{500-(1)-XXX}]
Remove the node containing element 2 with list.Remove(...)
 :: SUCCESS: list's state [{XXX-(400)-20},{400-(20)-10},{20-(10)-200},{10-(200)-6},{200-(6)-8},{6-(8)-5},{8-(5)-500},{5-
(500)-1},{500-(1)-XXX}]


Test I: Remove the node containing element 2, which has been recently deleted, with list.Remove(...)
 :: SUCCESS: list's state [{XXX-(400)-20},{400-(20)-10},{20-(10)-200},{10-(200)-6},{200-(6)-8},{6-(8)-5},{8-(5)-500},{5-
(500)-1},{500-(1)-XXX}]


Test J: Clear the content of the vector via calling vector.Clear();
 :: SUCCESS: list's state []


Test K: Remove last element for the empty list with list.RemoveLast()
 :: SUCCESS: list's state []



 ------------------- SUMMARY -------------------
Tests passed: ABCDEFGHIJK
```