Practical Task 5.1

(Pass Task)

Submission deadline: Monday, April 14
Discussion deadline: Friday, May 9

Instructions

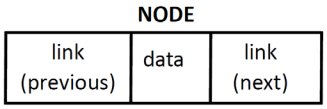
The objective of this task is to implement the Doubly Linked List, a generic data structure capable to maintain an arbitrary number of data elements and support various standard operations including reading, writing, and deleting data elements. Linked lists offer a number of advantages regarding the time complexity and practical application. For example, while an array-based data structure (e.g., a simple list collection) requires a contiguous memory location to store data, a linked list may record and distribute data elements across the memory. This is achievable by encapsulation of data into a special node structure and connecting nodes into a sequence via memory references (also known as links). Because of this, a (singly) linked list is not restricted in size and new nodes can be added increasing the size of the list to almost any extent. Furthermore, it is allowed to use the first free memory slot 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 $O(1)$ runtime, which is the best possible time complexity. Remember that these operations generally run in a linear $O(\mathrm{n})$ time for 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 node as it enables traversing the sequence of nodes in both directions, i.e., from the start to the end and vice versa. Hence, 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 the previous node. Because a singly linked list needs just a single link to the next node, traversing of nodes is only possible in one direction.

Because some operations that you will need to implement in this task are similar to those in the generic LinkedList<T> class of the Microsoft .NET Framework, we recommend you start your work with exploring this class.

The following steps indicate what you need to do.

1. Download the two C# source code files attached to this task. These files serve as a template for the program that you need to develop. Create a new C# project and import the files. Your newly built project should compile and work without errors, although it will initially fail most of provided test cases.
   - **DoublyLinkedList.cs** file contains the partially completed DoublyLinkedList<T> class. Your task is to add the missing methods of the class to obtain a fully functional data structure.
   - **Tester.cs** file is where you will find the Main method to be used as the starting point of your program. It also contains a series of tests that will verify if your DoublyLinkedList class works correctly. When running your program for the first time, these tests will fail. Once you have completed the task, all these tests will report success. See Appendix A for an example of what the program will output when working correctly.
2. Find and explore the nested Node<K> class contained in the DoublyLinkedList<T>. This generic class represents a node of a doubly linked list and is its building block. Because the order of data elements is not defined by their physical positions in the memory, the doubly linked list is a sequence of nodes. Therefore, each node is a data record that holds a payload and two auxiliary pointers referring to the preceding and the succeeding node in the ordered sequence of nodes. The two pointers enable navigation back and forth between two adjacent nodes.



Keep in mind that the Node<K> class implementing the required node structure is ready for your use. It provides the following functionality:

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

  Constructor. Initializes a new instance of the Node<K> class, containing the specified **value** and referring to the **previous** and the **next** node that, respectively, must precede and follow the new node in the sequence of nodes.

- **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.

- **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.

- **string ToString( )**

  Returns a string that represents the current Node<K>.

Note that the Node<K> class implements the INode<K> interface, which is also provided 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. Even though it must be protected, the user still needs access to the data stored in the node. Hence, the Node<K> implements the interface that permits reading and writing the data, while other attributes of the Node<K> class remain hidden. Check the INode<K> to see that Value is the only property expected by the interface.

3. Explore the given DoublyLinkedList<T> class and study its implementation, especially the following methods that make an example:

   - **DoublyLinkedList( )**

     Constructor. Initializes a new instance of the DoublyLinkedList<T> class.

   - **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 contained in the DoublyLinkedList<T>.

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

     Returns the node cast into the INode<T> that succeeds the specified **node** in the DoublyLinkedList<T>. If the given **node** is null, the method throws the ArgumentNullException. If the **node** is not in the current DoublyLinkedList<T>, the method throws the InvalidOperationException.

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

     Adds a new node containing the specified **value** at the end of the DoublyLinkedList<T> and returns the new node cast into the INode<T>.

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

     Searches for the first occurrence of the specified **value** in the DoublyLinkedList<T>. When found, the method returns the node cast into INode<T>, and null otherwise.

- **string ToString( )**

  Returns a string that represents the current DoublyLinkedList<T>.

Importantly, the DoublyLinkedList<T> class already has a number of private properties and methods. Specifically, it uses two auxiliary nodes: the **Head** and the **Tail**. Both nodes are introduced in order to significantly simplify the implementation of the class and make insertion functionality reduced to the following method:

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

In fact, the Head and the Tail are invisible to the user. They are controlled by the DoublyLinkedList<T> class. When the data structure has no contents, thus when the First and the Last properties are both set to null, the Head refers to the Tail, and vice versa. Therefore, the first added node 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. 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 important fact when you design and program other functions of the DoublyLinkedList<T> in this task.

The given template of the DoublyLinkedList<T> class will help you with programming of its remaining methods as they should be similar to the existing methods in terms of logic and implementation aspects.

4. Now, you must complete the DoublyLinkedList<T> and provide the following functionality:

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

     Returns the node cast into the INode<T> that precedes the specified **node** in the DoublyLinkedList<T>. If the given **node** is null, the method throws the ArgumentNullException. If the **node** 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> and returns the new node cast into the INode<T>.

   - **INode<T> AddBefore(INode<T> node, 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 cast into the INode<T>. If the given **node** is null, the method throws the ArgumentNullException. If the given **node** does not exist in the DoublyLinkedList<T>, the method throws the InvalidOperationException.

   - **INode<T> AddAfter(INode<T> node, 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 cast into the INode<T>. If the **node** is null, the method throws the ArgumentNullException. If the **node** 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, the method throws the ArgumentNullException. If the **node** 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 the InvalidOperationException.

   - **void RemoveLast()**

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

5. As you progress through the implementation, 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 code. 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 is checked for other potential mistakes.
6. In this task, you are not allowed to use any library functions (e.g., the methods provided by the LINQ library) that can completely replace the operations that you need to implement.

Further Notes

- Explore Chapter 3.4 and Section 7.3.3 of the SIT221 course book "Data structures and algorithms in Java" (2014) by Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser (2014). You may access the book on-line from the reading list application in CloudDeakin available in Content → Reading List → Course Book: Data structures and algorithms in Java.
- As a complementary material, refer to Chapter 2 of SIT221 Workbook available in CloudDeakin available in Content → Learning Resources → SIT221 Workbook to learn more about the singly and doubly linked lists.

Submission Instructions and Marking Process

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

- Make sure your programs implement the required functionality. They must compile, have no runtime errors, and pass all test cases of the task. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your programs thoroughly before submission. Think about potential errors where your programs might fail.
- **Submit** the expected code files as a solution to the task via OnTrack submission system.
- Once your solution is accepted by your tutor, you will be invited to **continue its discussion and answer relevant theoretical questions through a face-to-face interview**. Specifically, you will need to meet with the tutor to demonstrate and discuss the solution in one of the dedicated practical sessions (run online via MS Teams for online students and on-campus for students who selected to join classes at Burwood\Geelong). Please, come prepared so that the class time is used efficiently and fairly for all students in it. Be on time with respect to the specified discussion deadline.
  You will also need to **answer all additional questions** that your tutor may ask you. Questions will cover the lecture notes; so, attending (or watching) the lectures should help you with this **compulsory** discussion part. You should start the discussion as soon as possible as if your answers are wrong, you may have to pass another round, still before the deadline. Use available attempts properly.

Note that we will not accept 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 throughout 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 grading your achievements 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 the quality of your solutions.

Appendix A: Expected Printout

The following provides an example of the output generated from the testing module (Tester.cs) once you have correctly implemented all methods of the DoublyLinkedList<T> class.

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