Programming 4 (ADEV-3001)
Assignment 1 – Doubly Linked Lists

Assignment Instructions

You will construct 3 classes: Employee, Node<T> and LinkedList<T>.

You must create the classes with properties and methods described in this document, as well as satisfy all the unit tests.

Each project is a console application. NUnit tests have been supplied in LEARN and they must be added to your project before you begin.

IMPORTANT NOTE:

YOU ARE NOT ALLOWED TO MODIFY THE TEST CODE.
(Unless your instructor tells you to or you ask permission)

Note: You can comment out tests to satisfy syntax until you are ready to code the solutions.

Naming conventions

Please use appropriate Microsoft naming conventions for C#. For guidance:
https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

Employee Class – Milestone 1 (Mark: 15%)

- Properties: EmployeeId, FirstName, LastName
- Must contain an EmployeeId when creating a new Employee.
- First and Last names cannot be set after initialized in the constructor.

Public Method Name	**Description**
Employee(int employeeId)	Constructor, initializes only the EmployeeId
Employee(employeeId, firstName, lastName)	Constructor, initializes all properties
int CompareTo(Employee other)	Compare with other employees based on the EmployeeID. Note: You must implement IComparable<Employee>.
override string ToString()	Format: '*id fname lname*' Example: '6782342 John Doe' Example 2: '77853224 null null' (If first and last name are null, print the word "null" in their place)

Node<T> Class – Milestone 1 (Mark: 20%)

- Properties: Element, Next, Previous

Public Method Name	**Description**
Node()	Constructor, initializes fields to default values
Node(T element)	Constructor, initializes only the element property
Node(T element, Node<T> previousNode, Node<T> nextNode)	Constructor, initializes all properties

Programming 4 (ADEV-3001)
Assignment 1 – Doubly Linked Lists

LinkedList<T> : where T : IComparable<T> – Milestones 1, 2, & 3

- Must be able to use the object's '**CompareTo**' method to compare objects.
- Properties:
    - Head – Points to the first node in the list (or null if there are no nodes).
    - Tail – Points to the last node in the list (or null if there are no nodes).
    - Size – Count of the number of nodes in the list, zero when the list is empty.

Public Method Name	**Milestone**	**Marks**	**Big-O**
LinkedList() // constructor	1	5	O(1)
void Clear()	1	5	O(1)
bool isEmpty()	1	5	O(1)
T GetFirst()	1	5	O(1)
T GetLast()	1	5	O(1)
T SetFirst(T element)	1	5	O(1)
T SetLast(T element)	1	5	O(1)
void AddFirst(T element)	1	15	O(1)
void AddLast(T element)	1	15	O(1)
T RemoveFirst()	2	10	O(1)
T RemoveLast()	2	10	O(1)
T Get(int position)	2	20	O(n)
T Remove(int position)	2	20	O(n)
T Set(T element, int position)	2	10	O(n)
void AddAfter(T element, int position)	2	15	O(n)
void AddBefore(T element, int position)	2	15	O(n)
T Get(T element)	3	20	O(n)
void AddAfter(T element, T oldElement)	3	10	O(n)
void AddBefore(T element, T oldElement)	3	10	O(n)
T Remove(T element)	3	10	O(n)
T Set(T element, T oldElement)	3	10	O(n)
void Insert(T element)	3	20	O(n)
void SortAscending()	3	20	??? *

* SortAscending() Big O is dependent on your algorithm. Try to keep it as small as you can! When you finish this method, can you figure out what your big o is?

When coding each method remember:

1. You should not be looping more than you need to!
2. The only time you should be looping twice through the entire list is during the Sort Ascending.
3. Be mindful of the Big(O) of a method and ensure your code is doing that.
4. When in doubt, satisfy the unit tests!

The following pages describe each LinkedList method above in greater detail.

LinkedList Class Method Descriptions

For each method, think about how these 3 class properties are affected:

1. int Size
2. Node<T> Head
3. Node<T> Tail

LinkedList() - Constructor

Initializes all class variables to their default values.

1. **Size**: Sets to 0.
2. **Head**: Sets to null.
3. **Tail**: Sets to null.

void Clear()

Resets the LinkedList to the same condition it was in when it was first instantiated by the constructor.

> *Theory Note*: values that cannot be reached in memory will be collected by the garbage collector.

1. **Size**: Reset to 0.
2. **Head**: Points to nothing.
3. **Tail**: Points to nothing.
4. **Returns**: Nothing

bool IsEmpty()

Returns: TRUE if the list is empty, false if the list contains 1 or more Nodes.
Note: **Size**/**Head**/**Tail** do not change. There are no exceptions in this function.

T GetFirst()

Not a true “getter”, this is a short cut method to return the head’s element.
Returns: The element value contained by the Head Node.

T GetLast()

Not a true “getter”, this is a short cut method to return the tail’s element.
Returns: The element value contained by the Tail Node.

T SetFirst(T element)

Takes in an element and assigns it to the Head's element, returns the old element.

1. **Size**: No changes.
2. **Head**: Only the head node's element will be replaced.
3. **Tail**: No changes.
4. **Exceptions**: Throws an error if the list is already empty when attempting to set the head.
5. **Returns**: The old element stored in the head, before it was replaced.
   - ➔ Example: if the head contained the number 13,
     Then SetFirst(5) would return the element value 13.
   - ➔ Why? Think of this value as an "undo".
     The calling program has access to the old value and therefore can put it back!

T SetLast(T element)

The same as SetFirst(), but replaces the Tail's element and **returns** the old tail's element.

void AddFirst(T element)

Adds a Node to the front of the LinkedList by creating a new Node and assigning it the value of the incoming element parameter.

1. **Size**: Increases by 1 for each new node (with element) added to the list.
2. **Head**:
   - ➔ Becomes the newly created node and points its next pointer to the old head.
   - ➔ The old head needs to point its previous pointer to the new head.
3. **Tail**:
   - ➔ Does not change if there's already at least 1 node in the list.
4. **Exceptions**:
   - ➔ The very first node added to a list makes the Head and Tail point to the new node!
5. **Returns**: Nothing.

void AddLast(T element)

Same as AddFirst(), but adds a new Node to the end of a LinkedList.

Note, the logic is the same, but the direction of the pointers will be reversed from AddFirst().

T RemoveFirst()

Removes the first Node in the list. Return the removed node's element.

1. **Size**: Decreases by 1 for each node (with element) removed from the list.
2. **Head**:
   - ➔ Moves to the next Node in the list.
   - ➔ The new head's previous pointer needs to be removed.
3. **Tail**:
   - ➔ Does not change if there's more than 2 nodes in the list.
4. **Exceptions**:
   - ➔ The last node removed from the list makes the Head and Tail point to nothing!
   - ➔ Throw an error if the list is already empty when attempting to remove!
5. **Returns**: The old element stored in the head, before it was removed.

T RemoveLast()

Removes the last Node in the list. Return the removed node's element.

T Get(int position)

Returns the value of the element contained in the Node at the position specified in the parameter.

NOTE: The head is at position 1.

1. **Size**: Does not change.
2. **Head**: Does not change.
3. **Tail**: Does not change.
4. **Exceptions**:
   - ➔ Throw an error if the position provided is not valid: zero, negative or above the size of the list.
5. **Returns**: The element in the Node found at the position.

Suggested refactored method to use here:
Node<T> GetNodeByPosition(int position)

T Remove(int position)

Removes the Node at the position specified. Returns the old element's value of the removed Node.

The Nodes to either side of the removed Node will need to point to one another.

NOTE: The head is position 1.

1. **Size**: Decreases by 1 when a node (containing an element) is removed from the LinkedList.
2. **Head**: May need to be changed, if the position to remove is 1.
3. **Tail**: May need to be changed, if the position to remove is the size of the list.
4. **Exceptions**:
   - ➔ Throw an error if the position provided is not valid: zero, negative or above the size of the list.
   - ➔ The head and tail should point to nothing if the last node is removed
5. **Returns**: The element in the Node found at the position specified.

T Set(T element, int position)

Finds the Node at the position specified and replaces the element on that node with a new element. Returns the old element's value.

NOTE: The head is position 1.

1. **Size**: Does not change.
2. **Head**: Does not change.
3. **Tail**: Does not change.
4. **Exceptions**:
   - ➔ Throw an error if the position provided is not valid: zero, negative or above the size of the list.
5. **Returns**: The old element stored in the Node found at the position, before it was replaced.

void AddAfter(T element, int position)

Finds the Node at the position specified and adds a new node after it.

NOTE: The head is position 1.

1. **Size**: Increases by 1 when a node (containing an element) is added to the LinkedList.
2. **Head**: Will never change.
3. **Tail**: May need to be changed, if the position to add after is the size of the list.
4. **Exceptions**:
   - ➔ Throw an error if the position provided is not valid: zero, negative or above the size of the list.
5. **Returns**: Nothing.

void AddBefore(T element, int position)

The same as AddAfter(), but adds before the node at the specified position.

1. **Size**: Increases by 1 when a node (containing an element) is added to the LinkedList.
2. **Head**: May need to be changed, if the position to add before is 1.
3. **Tail**: Will never change.
4. **Exceptions**:
   - ➔ Throw an error if the position provided is not valid: zero, negative or above the size of the list.

Returns: Nothing.

T Get(T element)

Same as Get(position), except this finds the first node found with the specified oldElement value and returns that node's value.

Think of this method as a "does this element exist" method.

Suggested refactored method to use here:
Node<T> GetNodeByElement(T element)

T Remove(T element)

Same as Remove(position), except this finds the first node found with the specified oldElement value and returns that node's value.

T Set(T element, T oldElement)

Finds the first node found with the specified oldElement value and replaces the element on that node with a new element.

Returns: The original value of the found Node's element.

void AddAfter(T element, T oldElement)

The same as AddAfter(), but adds after the first node found with the specified oldElement value.

NOTE: We start from the head and move to the tail when searching for oldElement's value.

void AddBefore(T element, T oldElement)

The same as AddBefore(), but adds before the first node found with the specified oldElement value.

NOTE: We start from the head and move to the tail when searching for oldElement's value.

void Insert(T element)

Add the element into the linked list in natural ascending order. Note that all elements used by LinkedList ***MUST*** implement the comparable **interface**, so please use the **CompareTo()** method within Insert().

> Example Insert:
> Current list is 1 5 8 15
> Adding **7** would add at: 1 5 **7** 8 15
>
> Important note, if the list is NOT already in ascending order, this will not change the algorithm.

1. **Size**: Increases by 1 when a node (containing an element) is added to the LinkedList.
2. **Head**: May change if the element is the smallest in the list.
3. **Tail**: May change if the element is the largest in the list.
4. **Returns**: Nothing.

void SortAscending()

Sort the elements into ascending order.

You can use any algorithms you want to achieve this.

Example of an easy sorting algorithm is a bubble sort.

Refactoring Suggestions and Hints

1. You cannot use Get(position) and Get(element) as your refactored methods, because they will only return a T element Instead: create a private method to return a Node<T> such as: GetNodeByPosition or GetNodeByElement.

2. Are you returning **true** or returning **false**?

```
if (x > 4) {
  return true;
}
else {
  return false;
}
```

```
// DO THIS INSTEAD!!
return x > 4;
```

3. Are you LOOPING within a LOOP? (BIG O of N squared)
   Stop. This is VERY bad. DO NOT DO THIS (unless you must for sorting).

4. By Milestone 3 you may want to consider private methods for generically removing and adding. Example: A Universal RemoveNode() or a UniversalAddNode() or even a Universal SetNode().

5. Run your validation (exceptions) from a single method, instead of on all methods.

6. Walk through your code! Set a breakpoint at the beginning of each method and step through line by line. You will often find repetition and mistakes easily this way. If you do not know how to do this, please ask your instructor or an EA for help.

Other Methods

Extra methods can be coded, however the documentation for that method must contain a justification of why this method is necessary. In this class, you will often make extra private methods to refactor your code.

Deliverable Hand In

Create an archive of the project folder (right-click -> Send to Compressed folder).

Submit the zip file directly to the appropriate LEARN DropBox.

Marking Guidelines

If a milestone is not handed in before the due date and/or the student has not met with the instructor for a code review, a mark of 0 will be given for that milestone. You must meet in person (or online meet up) with your instructor BEFORE the milestone deadline.

IMPORTANT NOTE: You must upload your assignment to the DropBox even if you have met with the instructor. It's a good backup place for your work anyway!

If there are extenuating circumstances preventing you from meeting with your instructor before the due dates, you must communicate this as far ahead of time as possible. The instructor has final say on what can be submitted late.

The grade noted beside each method will be deducted if 1 or more tests related to that method are failing.

Another marking criterion is optimization. Because this is subjective, it will be determined on a case-by-case basis during the time the student sits with the instructor. These suggestions will often relate to the following:

- **Refactoring**: Should be able to reuse duplicate code through a private method
- **Simplification of code**: if it is deemed that the student has written too much code, where less would have sufficed
- **Overuse of Global Variables**
- **Naming conventions**: variables or functions

When an optimization suggestion is made, the student must fix this within an arranged timeline based on the complexity and to be determined by the instructor.

Each optimization suggestion will be default weighted to 25%, unless the instructor says otherwise.