COIS-2020H: Assignment 3

# Part 1: Making a doubly linked list

In Lab 5, you created a singly linked list. To finish off with linked lists, you're going to build on that lab to create a **doubly linked list.** It should support the following operations:

1. Adding to the front (head) of the list.
   1. **void AddFront(T item)**
   2. **void AddFront(Node<T> node)**
2. Adding to the end (tail) of the list.
   1. **void AddBack(T item)**
   2. **void AddBack(Node<T> node)**
3. Inserting after a given node within the list.
   1. **void InsertAfter(Node<T> node, T item)**
   2. **void InsertAfter(Node<T> node, Node<T> newNode)**
4. Inserting before a given node within the list.
   1. **void InsertBefore(Node<T> node, T item)**
   2. **void InsertBefore(Node<T> node, Node<T> newNode)**
5. Removing an item from the list.
   1. **void Remove(Node<T> node)** – Removes a node directly from the list.
   2. **void Remove(T item)** – Searches the list front-to-back for the first element equal to **item** and removes its node. There is a small C#-ism about getting this to work: note on that later.
6. Splitting the list.
   1. **LinkedList<T> SplitAfter(Node<T> node)** – Creates a new list whose Head is the node after the provided **node**. The given **node** stays in this list.
7. Merging two lists together.
   1. **void AppendAll(LinkedList<T> otherList)** – Removes all the nodes from **otherList** and adds them to the end of the current list.
8. Finding an element within the list.
   1. **Node<T>? Find(T item)** – Searches the list front-to-back and returns the first node that contains a matching item (or null if not found). Same note from Remove(T item) applies here (see below).

With the exception of Find and the T item overload of Remove, **you should be able to do all of these in constant-time** (without any loops) just by fiddling with the Head, Tail, Next, and Prev pointers. All of these methods should be **public.**

## Notes on the design/handling edge-cases

Start by taking a second to skim the provided LinkedList.cs file. There are a few notable differences between the "design" of the provided LinkedList<T> class when compared to the SinglyLinkedList<T> from lab.

In lab, the Node class was nested and private. It was only used within the internals of the list as an *implementation detail;* a means to an end. Here, the Node<T> class is its own public class. The reason for this change is to allow the list to *hand out references* to its nodes to someone using the list. The user of the list can then *give the references back* to the list so it doesn't need to traverse itself every time. The user can hold onto their Node<T> references for as long as they want.

This design is much better at taking advantage of the strengths of a linked list, but it has a drawback. The primary challenge with this approach is that **you have to be *very* careful with your nodes:**

* Some of your methods expect to be given a node as a parameter: it's possible for them to be given one that **belongs to a different list.**
* It's possible to create a Node<T> that doesn't belong to ***any* list**.
* It's possible to remove a node from a list ***while* you're foreach-ing** over it, which might cut your loop short (e.g., if you remove the "current" node, the IEnumerable method in the background might lose its Next pointer).

The first one can be a very easy mistake to make when dealing with multiple lists.

Just to be clear on the expectations: **you do not need to handle these edge-cases.** You are allowed to assume that any **node** parameters belong to the current list, and you are allowed to assume that any **newNode** parameters are not already part of the current or another list. **However:** **it may be wise to do some of these checks anyways.** It's much easier to write one *if-throw* statement ahead of time than it is to debug a list breaking mysteriously (especially a list that's made out of pointers).

If you choose to do so, there are several ways to track/check for these problems. Feel free to add as many fields/properties/methods as you need to (or want to) to both the node and list classes. **Just note** that adding depending on what extra things you decide to add, some of your methods *might* not stay , since you *might* need a loop to keep them updated. **You won't lose any marks for that,** but it's worth being aware of.

There are a number of ways to handle these edge-cases, too (e.g., maybe some are fixable within your if-statement), but I would suggest simply throwing an [ArgumentException](https://learn.microsoft.com/en-us/dotnet/api/system.argumentexception?view=net-6.0) or [InvalidOperationException](https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-invalidoperationexception) (make sure to pass a message string to the constructors).

## Couple quick hints

* Make your life easier by **re-using methods you've already written.** For example, you really only need to write one AddFront method: adding a plain T item to a list is as simple as creating a new Node<T> and calling your other version of AddFront.
  + As a specific example, doing Find first would make Remove(T item) a lot easier… 👀
* **Be careful with your pointers!** The hardest part of this exercise is managing all the edge-cases. For example, when splitting/merging lists, you have to worry about two Heads *and* two Tails on top of the Nexts and Prevs. That's six+ references to handle!
  + As a specific example, consider what happens if you call InsertBefore with the current Head of the list.
* *This* part of *this* assignment is perhaps where doing some doodles ahead of time on a piece of scratch paper will be the most valuable.

## Getting Find/Remove to work

Since a generic type, T, is a placeholder for "literally any type," you can't really call any methods on an object of type T. For the ArrayList<T> class, we got around this by using a [*generic type constraint*](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint)*[[1]](#footnote-1):*

|  |
| --- |
| public class ArrayList<T> where T : IComparable<T>, IEquatable<T> |

The *where* clause makes it so that only things that are comparable and equatable can be put into an ArrayList<T>, which allows us to call .CompareTo() and .Equals() on our list elements. However, this constraint also means that we can't, for example, have a list of lists: an ArrayList<ArrayList<int>> is simply not possible because ArrayLists themselves are not IComparable and IEquatable.

The LinkedList<T> class you'll be making **does not** have these type constraints. That means you can put whatever you want into your list—including another LinkedList. But it *also* means that **you can no longer call any item1.Equals(item2) methods** when you're trying to Find() something in your list.

**The little bit of C# that you need to know to make your Find()/Remove() methods work is:**

|  |
| --- |
| // Instead calling T.Equals(T) directly:  if (item1.Equals(item2)) { /\* ... \*/ }  // Ask C# for some magical object that knows how to compare two T's:  EqualityComparer<T> comparer = EqualityComparer<T>.Default;  if (comparer.Equals(item1, item2)) { /\* ... \*/ } |

As a side note, you can do the same for IComparable by using Comparer<T> instead of EqualityComparer<T>:

|  |
| --- |
| // This:  if (item1.CompareTo(item2) < 0) { /\* ... \*/ }  // Turns into this:  var comparer = Comparer<T>.Default;  if (comparer.Compare(item1, item2) < 0) { /\* ... \*/ } |

Both of these "magical" objects are from the System.Collections.Generic namespace.

That should be all you need to know to get started… Good luck! 🙂

(Part 2 starts on next page)

# Part 2: Making a queue

In Lab 7, you created a circular buffer. Now, you're going to expand on that basic concept to implement a higher level data structure: **a Queue<T>.** A queue supports two or three main operations:

1. **void Enqueue(T item)** – Adds an item to the end of the queue. When the queue is full, it should **double** its current capacity by resizing its internal buffer. You should write a private/protected **Grow()** method to handle this.
2. **T Dequeue()** – Removes an item from the start of the queue. Attempting to dequeue from an empty queue **should throw an InvalidOperationException** (including a message string like the one in the lab).
3. **T Peek()** – Returns a reference to the first thing in the queue without actually dequeuing it. If there is no first item, this should throw the same exception.

In addition to these operations, your queue should:

1. Have a public bool getter for **IsEmpty.**
2. Have a public int getter for **Count.**
3. Have a public int getter for **Capacity.**
4. Implement the **IEnumerable<T>** interface.

In the lab, you used a count property to help differentiate between an empty/full circular buffer. This time, you're going to have to use a different approach: **as a challenge/learning exercise, you should make these getters without using an internal count property.** That is, **your IsEmpty, Count, and Capacity getters should all compute their return values** using the existing buffer, start, and end properties.

## Some notes/hints

* A queue's *capacity* refers to "the number of elements it can hold before it needs to expand its internal buffer," which is **not necessarily the same** as the buffer's length.
* When you remove an element from the queue, don't forget to put a default in its place.
* When implementing Grow(), you **can** use the built-in Array.Copy/Array.Resize methods, if they help.
* As always, make sure to draw lots of pictures! It may also be helpful to start by writing out your algorithms as pseudocode and walking through a couple examples with different initial conditions.

Once again, good luck! I believe in you! 😄

# Part 3: The Battle of the Wizards and Goblins continues

Just like last time, you're going to be presented with a word problem describing how a small visualization/animation/game involving your Wizards and Goblins. This time, the goblins are attacking the wizards at their stronghold!

The A3-Program project in the assignment starter code has the following files/folders:

* **Game.cs** – Contains the logic that runs on startup and on each frame of the animation. **This is where you'll write most (if not all) of your code for Part 3.**
* **Entities folder** – Contains the CombatEntity, Wizard, Goblin, and Spell classes.
* **Renderer folder** – Contains the code that actually draws all the sprites onto the screen. **Aside from fixing the namespace at the top of the file, you do not need to touch anything in this folder.** Of course, you totally *can* if you want to customize the look/animation of the game… just don't be surprised if things break! 😛
* **Program.cs** – Simply serves as an entrypoint that starts the Renderer. There isn't really much to change here.

If you run the starter project, you should see something like this:

A screenshot of a video game

Description automatically generated

The empty boxes on the right-hand side will show entity information and Console output. The region on the left is the actual play-field, where the wizards and goblins will duke it out.

## Game overview

Your Wizards stand at the bottom of the screen in a line, shooting spells up at the Goblins, *Space Invaders* style. If you look at the CastleDefender class in **Game.cs,** you'll see several properties waiting for you:

* **LinkedList<Wizard> WizardSquad** – Your main "front line" of wizards. These wizards take turns firing spells up at the goblins roaming in their courtyard.
* **uint nextSpellTime** – A timer for when the wizards are allowed to cast a spell next. Each time they fire, the next one has to wait a few frames before they can shoot.
* **Node<Wizard>? ActiveWizard** – The wizard who will cast a spell next. Each time they cast a spell, some of their Energy is depleted.
* **Queue<Wizard> RecoveryQueue** – Once wizards run out of energy, they fall out of the front line and get into a queue in the back of the castle where they can eat some food to rejuvenate.
* **LinkedList<Goblin> GoblinSquad** – This is your main attacking force of goblins. They roam around the courtyard trying to wreak havok (though not very successfully).
* **LinkedList<Goblin> BackupGoblins** – Once half of your initial GoblinSquad has been wiped out, more emerge from the forest as reinforcements!
* **Vector2 goblinDirection** – The direction that the goblin squad is currently walking in. They bounce off the walls of the courtyard like a DVD screensaver.
* **LinkedList<Spell> Spells** – A list of currently-in-the-air spell projectiles.

More details to follow on what exactly should happen on each frame.

**On start-up,** your game should:

* Create **eight** wizards and add them to the main WizardSquad.
* Create **eight** goblins and add them to the main GoblinSquad.
* Create **six** backup goblins and add them to the BackupGoblins list.
* Pick a random direction for your goblin squad to walk in (create a random Vector2 with a length of 1.0).
* Mark the first wizard as active by getting its node from the linked list.
* Set the nextSpellTime so the first wizard doesn't fire immediately after spawning. Something like **15 ± 5** (using RNG) seems to be a fairly balanced number.

## The Update method

Since this assignment's visualization is much closer to an actual "game" than Assignment 2's was, there's quite a bit more little movie pieces to this Update method. So, this time, you're going to be given the entire process in a step-by-step process: **your job is just to figure out how to convert the following into code.** Thankfully, you can do each of them in their own method and tackle them one at a time (\*cough,\* you're gonna want to split them up into methods!)

On every frame(AKA, in your Update method), your game/animation/visualization should do as follows.

1. **Iterate through the list of Spells and update each one.**
   1. **Move them up the screen.** Note that "up" on the screen is the negative Y-direction. Spells have a static property for speed that you can use to determine how fast they moves.
   2. **If the spell flies off the top of the screen,** it should be removed from the list of spells. You can call the **CastleGameRenderer.IsOffScreen()** method to check if the spell has flown off the top yet.
2. **Iterate & update the list of Goblins.**
   1. **Move the goblins.**
      * The first goblin (the one at Head) should **move according to goblinDirection.**
      * All of the other goblins **follow the one in front of them.**
   2. **Loop over all the spells and check for any collisions.** If any spell collides with a goblin, **both of them** should be removed from their respective lists. When any goblin in the squad gets hit, the lead goblin should pick a new goblinDirection.
3. **Update the Wizards.**
   1. **If the nextSpellTime timestamp has passed,** it's time for the currently active wizard to **cast a spell.**
      * Casting a spell involves creating a new Spell object at the wizard's current position with the wizard's SpellType, and then decrementing the wizard's Energy by their SpellLevel (kind of like the "cost" of the spell).
      * Once a wizard casts their spell, the next wizard in the chain becomes the "active" wizard, and they wait for their turn (be sure to actually re-set your nextSpellTimer).
   2. **If a wizard runs out of energy,** they should fall-out of the front lines and get into the recovery queue to replenish their Energy.
   3. The first wizard in the recovery queue (if applicable), should take a bite out of the food in front of them every 5 or so frames, **replenishing their Energy** by one point.
   4. Once the first wizard in the recovery queue is finished eating (their energy hits its max), they should excuse themselves from the queue and rejoin the frontlines immediately **before the currently active wizard.**
4. **Check if the Goblins need reinforcements yet.**
   1. Remember, the six extra goblins from the backup list show up when around half of the original group is remaining. **Use your AppendAll() method** to simply merge the backup goblins directly onto the main group.
5. **Check if the Goblins have been defeated yet.**
   1. **If all the goblins are gone,** you can call the Pause() method (inherited from the Visualization base class) to stop the simulation. Or you can keep it running, if you want. But you should write a message to the console at the very least.

## Provided functions

This assignment has a few extra helper methods to simplify some of the tedious or math-heavy parts. Of particular note are:

* **CastleGameRenderer.IsOffScreen()** takes an entity or position as a parameter and tells if you if it is beyond the bounds of the regular game area.
* **CastleGameRenderer.CheckWallCollision()** should be used to stop your leader goblin from meandering off the side of the playfield into the forest. It will automatically handle fiddling with the vectors to make the goblins bounce off the walls.
* The **Move()** and **MoveTowards()** methods have been updated, there are now additional overloads that accept actual positions (not just entities). Move() can now be given a direction + speed instead of just a + .
* The CombatEntity class has a **Colliding()** method that can be used to check if two entities are touching one another. There is both a static- and instance-method version.
* Last time, there was a LogMessage() function you could use to get your messages to appear within the game. This time, all you have to do is Console.WriteLine—they will be redirected to the game console automatically.

## Other notes

* As much as you implemented IEnumerable for your data structures, you probably won't be using them much: since you need access to Node objects, you'll be doing a lot of classic while-loop traversals.
* The warning in Part 1 about removing nodes while iterating *mostly* applies to foreach loops, since having direct access to your nodes in a while loop gives can help avoid the issue. You still need to be *very* mindful, though: since your Remove() method sets the node's .Next and .Prev pointers to null (or, it's supposed to), think twice about the order of operations before you call it.

# Part 4: Theory questions

Just like last time, nothing too formal here. One or two short paragraphs for each one; we're looking for intelligent discussion.

## Question 1: LinkedList edge-cases

In Part 1), you were told that you did not have to handle any of the funky edge-cases that come with making the Node<T> class into a public class, but that you could if you wanted to. It was also mentioned that handling some edge-cases may require missing out on that sweet, sweet time-complexity.

**If you did implement any of these fixes,** pick any two of the three bullet points from earlier and **describe and explain** some "fix" for them. That is, how might your data structure detect (and potentially recover from) these user errors? Which extra fields, methods, etc., does your data structure need to have? Does it affect the time complexity? How about space complexity?

**If you did implement one of the "fixes,"** simply go over how it changed or did not change the time complexity of your implementation.

## Question 2: Cycles in a linked list

There are solutions to this out there on the web already, but it's still a good thought-exercise: **describe, in words, an algorithm to detect a *cycle* in a linked list.** You can either come up with your own or find an existing solution online. If you do go with an existing one, provide a bit of background information/history on it, too.

As an example, if you had a linked list that was *supposed* to look like this:

A ⇄ B ⇄ C ⇄ D ⇄ E ⇄ F ⇄ G

but E had its .Next pointer set to B by mistake. Then, going backwards would give you G, F, E, D, C, B, A—as expected—but going forwards would give you A, B, C, D, E, B, C, D, E, B, C, D, E, B, … you get the idea.

1. <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint> [↑](#footnote-ref-1)