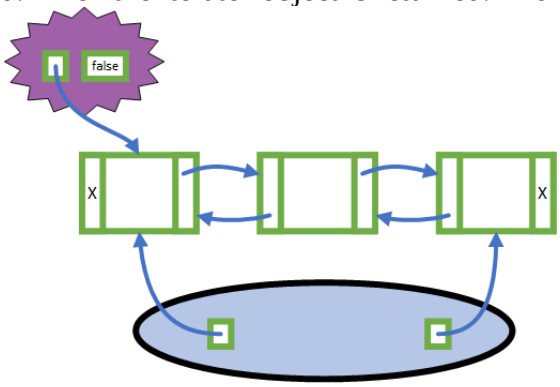
Homework - Linked List Iterators

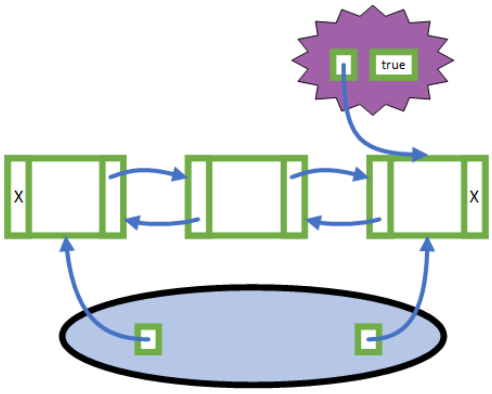
Goal: To understand how to work with iterators of a linked list. To also learn how iterators are used, and why iterators are useful. To be exposed to using algorithms and lambda expressions alongside iterators

Create and implement the following methods:

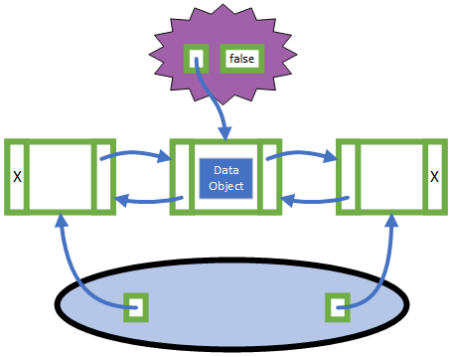
The DoublyLinkedList begin() method: This should create a temporary iterator object. Then set the iterator object's `pastTheEnd` to false (unless the list is empty, then set it to true), and the iterator object's pointer should be assigned to the list's head. Then the iterator object is returned. A diagram is given below:



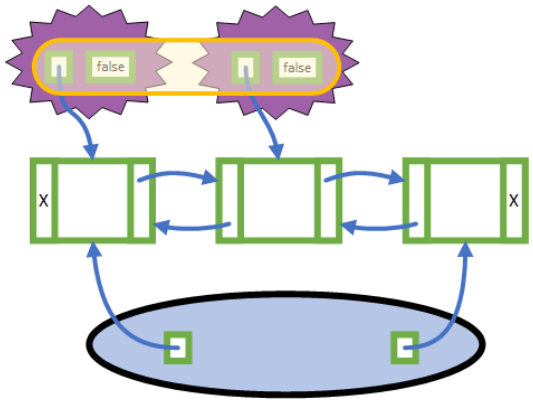
The DoublyLinkedList end() method: This should create a temporary iterator object. Then set the iterator object's `pastTheEnd` to true, and the pointer should be assigned to the list's tail. Then the object is returned. Note that the end iterator acts as though it's really "past the end". See `operator--` to get a better understanding what this means. A diagram is given below:



overloaded * operator: This just returns the data that is in the node that iterator object is pointing to. A one-liner. Make sure the return type is a `T&` and not a `T`. A diagram of the data object you need to return is given below.



overloaded != operator: This checks to see if both iterator objects have the same `pastTheEnd` and the same pointer values. If both objects have the same values, then return false. Otherwise just return true. Note that you are comparing the "this" object with another Iterator object passed in as an argument into `operator!=`, which requires that operator!= has a single Iterator parameter (const and by-reference is a good idea here). A diagram is given below:



overloaded == operator: Similar to the prior overloaded operator, just returns the opposite bool.

overloaded ++ prefix operator: The return type needs to be `Iterator<T>`. The iterator should be moved forward one node, if possible. If moving it forward would mean it would become empty/nullptr, don't move it forward, and instead set the boolean `pastTheEnd` to true. If `pastTheEnd` is already true, you don't need to do anything. You need a return type, you need to return a copy of the iterator, and you can do that with return *this.
overloaded ++ postfix operator: The return type needs to be `Iterator<T>`. You can overload the postfix operator by having a single (int) in the parameter section. Very similar to the prefix operator with one change. You first have to make a copy of itself. Increment the object pointed to by this. Return the copy you previously made.
overloaded -- prefix operator: The return type needs to be `Iterator<T>`. Somewhat like to the `operator++` prefix, but going backwards. You need to return a copy of the iterator, and you can do that with return *this. Note that while `operator++` uses `pastTheEnd` to prevent the iterator from doing undesirable things on the right side of the collection, `operator--` has no equivalent mechanism for the left side. The behavior of going off the left side of the collection is undefined, or in other words, don't worry about supporting it.

Note that I added one additional test on this assignment, `testIterationTricky()`. Those test the ability to handle a one node and a zero node linked list. Make sure you use the data member `pastTheEnd` appropriately.

The **assignmentReverse() method** also has you test working with iterators and not with the collection itself. Note that the parameters `begin` and `end` are iterator objects. You must use only these two iterators to find a way to reverse the containers. These iterators are not node pointers, so you cannot request a node's forward pointer, for example. You can only use the operators ++ (prefix), ++ (postfix), -- (prefix), ==, !=, and *. Note that it's easy to solve this for a container of an odd amount of items, but an even amount of items is trickier. An ugly solution is to somehow count up the amount of nodes, then start over and loop half of that count. A better way is to detect when the two iters have hit the middle or are going to cross over each other.

You must also complete the two lambda expressions for the **`processCollection()` function**. This method passes in two iterators and two lambda expressions. The purpose of processCollection is to 1) loop through all elements, and if an element is "true" via the first lambda, then 2) run the second lambda. The first lambda needs to pass in a Student object by reference. Then if the obj's money in the bank is less than 100, return true, otherwise, return false. The second lambda needs to pass in Student object by reference then sets the money in the bank to 100. Note that both use the basic lambda syntax: []{}. You don't need to capture anything into the [].

You do need a parameter in the () for a collection's data item (not the entire collection). The {} is where you place your code.

As always, look at the source code to get an exact idea of every detail of the assignment. Watch the lecture videos. And as always, it is effectively a requirement of the course to ask questions if you get stuck.