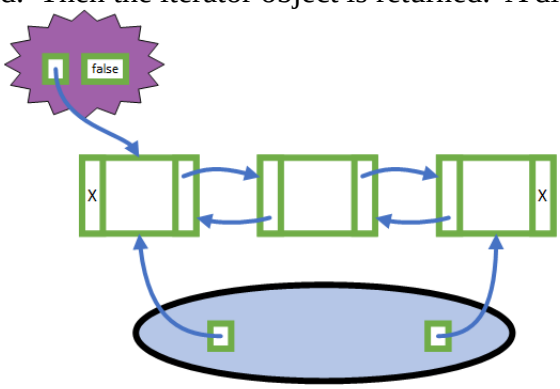
Homework - Linked List Iterators

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

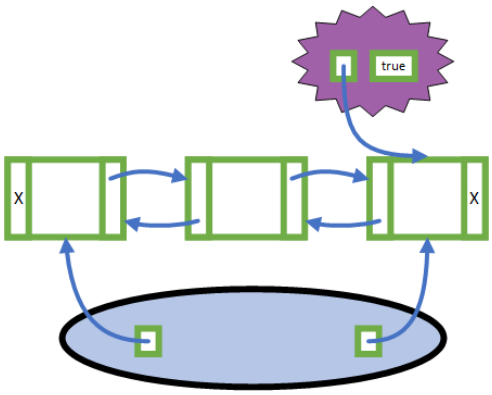
Note: that this assignment does not immediately compile. You must implement the begin() and operator*() to get the first tests to pass. Then you should comment out tests one-by-one and implement the missing parts for those.

Instructions: Create and implement the following methods:

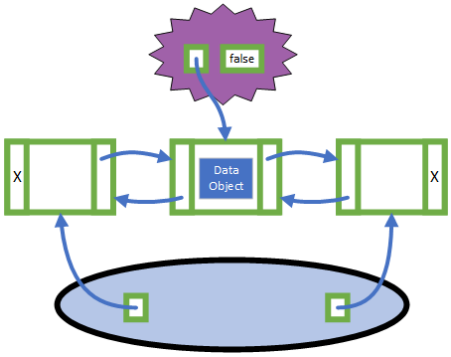
▪ **The DoublyLinkedList begin() method:** This should create a temporary iterator object. Then set the iterator object's `pastTheEnd` to false (unless the list is empty, then set it to true), and the iterator object's pointer should be assigned to the list's head. Then the iterator object is returned.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



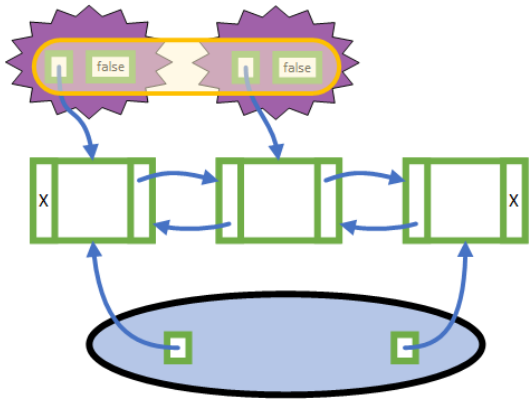
▪ **The DoublyLinkedList end() method:** This should create a temporary iterator object. Then set the iterator object's `pastTheEnd` to true, and the pointer should be assigned to the list's tail. Then the object is returned. Note that the end iterator acts as though it's really "past the end". See `operator--` 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, this one is just checking for full equality.

▪ **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 (e.g. auto copyOfItself = *this;). Now use similar incrementing logic from the operator++ prefix operator. For the return object, return the copy object.

▪ **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. Hint: pastTheEnd should never be false if the list is empty.

▪ 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. You can only use the operators ++ (prefix), ++ (postfix), -- (prefix), ==, !=, and *. Note that it is easy to solve this for a container of an odd number of items. An ugly solution is to somehow count up the number of nodes, then start over and loop half of that count. A better way is given below.

Suppose a container contains these items: A B C D E. An iterator returned by begin() points to item A, and an iterator returned by end() points to the past-the-end virtual state just past item E. The first step should be to decrement that iterator at past-the-end so it is on item E. Now swap the values of the two iterators. The container should now be E B C D A. Verify if the iterators are the same (operator== or operator!=). They are not the same. Move the two iterators inwards so they point to items B and D. Swap data. The container should now be E D C B A. Move the two iterators inwards, so they both point to item C. Verify if the iterators are the same (operator== or operator!=). They are the same, so the algorithm is done.

When the list contains an even number of items, reversing gets a bit tricker.

▪ The **lambda expression** to search for palindrome strings. The unit test relies on `std::copy_if` to copy elements that match a criteria into another container. In this case, the criteria must be supplied by you as a lambda expression. The output container is already given for you, it is called `output`. The `std::copy_if` has four arguments, three of which are already supplied (the first two parameters are iterators to the input and the third parameter is to the output container). You must supply the fourth parameter, a lambda expression.

The lambda expression has syntax of []{}. The lambda expression captures nothing, so you only need an empty `[]`. Only one parameter is needed within the (), one string parameter that is both const and by reference. For the code body within the {}, your logic must check if a string is a palindrome and return an appropriate boolean. The code body must return true if the string is a palindrome, and false if not. Note that the strings have spaces, punctuation, and a mix of upper- and lower-case letters. You should make a copy of the string, remove all non-letter characters, turn all upper case into lower case, then perform a check if the resulting string is a palindrome.

General tips for success: Look at the source code to get an exact idea of every detail of the assignment. Watch the lecture videos. For many university courses, one of the requirements is effectively to ask questions if you get stuck.