The Sequence Class—Design

Here is the complete invariant of our class, stated as three rules:

1. The number of items in the sequence is stored in the member variable **many_nodes**.
2. For an empty sequence, the **head_ptr** and **tail_ptr** data fields will be set to null; for a non-empty sequence, the items are stored in the class Node objects with their sequence order.
3. If there is a **cursor** item, then it references a node that is somewhere in the list; if there is no current item, then **cursor** and **precursor** equals null.

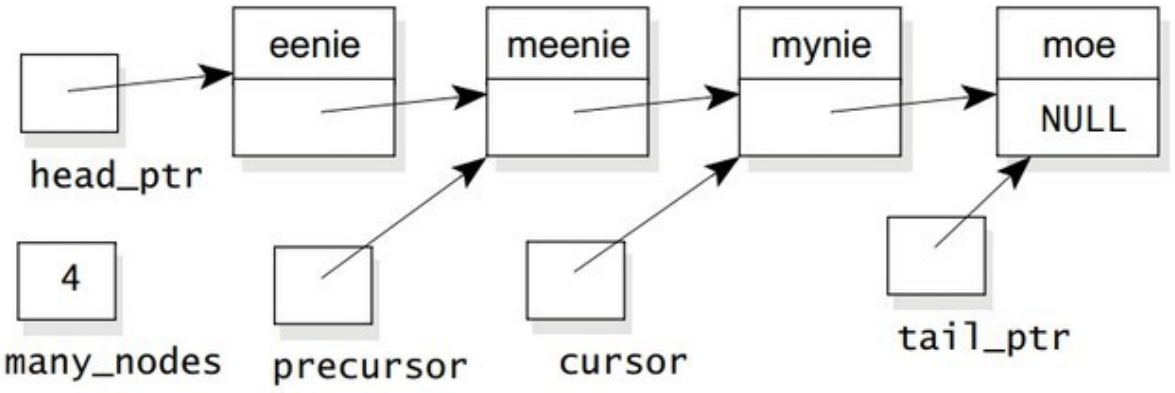
The Sequence Class—Design Suggestions

Using a linked list to implement the sequence class seems natural. We'll keep the items stored in a linked list, in their sequence order. The "current" item on the list can be maintained by a member variable that points to the node that contains the current item. When the start function is activated, we set this "current pointer" to point to the first node of the linked list. When advance is activated, we move the "current pointer" to the next node on the linked list.

With this in mind, we propose five private member variables for the new sequence class. The first variable, **many_nodes**, keeps track of the number of nodes in the list. The other four member variables are node pointers:

- **head_ptr and tail_ptr**—the head and tail pointers of the linked list. If the sequence has no items, then these pointers are both null. The reason for the tail pointer is the attach function. Normally this function adds a new item immediately after the current node. But if there is no current node, then attach places its new item at the tail of the list, so it makes sense to keep a tail pointer around.
- **cursor**—points to the node with the current item (or null if there is no current item).
- **precursor**—pointstothenodebeforecurrentitem(or null ifthereisno current item or if the current item is the first node). Can you figure out why we propose a precursor? The answer is the insert function, which normally adds a new item immediately before the current node. But the linked-list functions have no way of inserting a new node before a speci- fied node. We can only add new nodes after a specified node. Therefore, the insert function will work by adding the new item after the precursor node—which is also just before the cursor node.

For example, suppose that a list contains four strings, with the current item at the third location. The member variables of the object might appear as shown in the following drawing.



Notice that **cursor** and **precursor** data fields are pointers to nodes rather than actual nodes.

Start your implementation by writing the header file and the invariant for the new sequence class. You might even write the invariant in large letters on a sheet of paper and pin it up in front of you as you work. Each of the member functions count on that invariant being true when the function begins. And each function is responsible for ensuring that the invariant is true when the function finishes.

Keep in mind the four rules for a class that uses dynamic memory:

1. Some of your member variables are pointers. In fact, for your sequence class, four member variables are pointers.
2. 2. Member functions allocate and release memory as needed. Don't forget to write documentation indicating which member functions allocate dynamic memory so that experienced programmers can deal with failures.
3. 3. You must override the automatic copy constructor and the automatic assignment operator. Otherwise two different sequences end up with pointers to the same linked list. Some hints on these implementations are given in the following "value semantics" section.
4. 4. The class requires a destructor, which is responsible for returning all dynamic memory to the heap.

The Sequence Class—Value Semantics

The value semantics of your sequence class consists of a copy constructor and an assignment operator. The primary job of both these functions is to make one sequence equal to a new copy of another. The sequence that you are copying is called the "source," and we suggest that you handle the copying in these cases:

1. If the source sequence has no current item, then simply copy the source's linked list with list_copy(). Then set both precursor and cursor to the null pointer.
2. If the current item of the source sequence is its first item, then copy the source's linked list with list_copy(). Then set precursor to null, and set cursor to point to the head node of the newly created linked list.
3. If the current item of the source sequence is after its first item, then copy the source's linked list in two pieces using list_piece. The first piece that you copy goes from the **head_ptr** to the **precursor**; the second piece goes from the **cursor** to the **tail_ptr**. Put these two pieces together by making the link field of the **precursor** node point to the **cursor** node. The reason for copying in two separate pieces is to easily set the new **precursor** and **cursor** references.

After copying the linked list, be sure to set **many_nodes** to equal the number of nodes in the source. Also, beware of the potential pitfalls that accompany the implementation of your assignment operator

To test your implementation, use the **sequenceTest.cpp** test driver provided with the assignment.