

COEN 70L - Formal Specification and Advanced Data Structures

Implement the **sequence** class. A sequence class is similar to a bag—both contain a bunch of items, but unlike a bag, the items in a sequence are arranged in an order. In contrast to the bag class, the member functions of a sequence will allow a program to **step through the sequence one item at a time**. Member functions also permit a program to control precisely where items are inserted and removed within the sequence. Our sequence is a class that depends on an underlying **value_type**, and the class also provides a **size_type**.

Three member functions work together to enforce the **in-order retrieval rule**:

```
void start();
value_type current( ) const;
void advance( );
```

- After activating `start`, the `current` function returns the first item
- Each time we call `advance`, the `current` function changes so that it returns the next item in the sequence

The documentation of the class is as follows:

(Note: This documentation requires extension while you work on this project.)

```
// FILE: sequence1.h
// CLASS PROVIDED: sequence (part of the namespace scu_coen70)
//
// TYPEDEFS and MEMBER CONSTANTS for the sequence class:
// typedef ____ value_type
// sequence::value_type is the data type of the items in the
// sequence. It may be any of the C++ built-in types (int, char,
// etc.), or a class with a default constructor, an assignment
// operator, and a copy constructor.
//
// typedef ____ size_type
// sequence::size_type is the data type of any variable that keeps
// track of how many items are in a sequence.
//
// static const size_type CAPACITY = _____
// sequence::CAPACITY is the maximum number of items that a sequence
// can hold.
//
// CONSTRUCTOR for the sequence class:
// sequence( )
// Postcondition: The sequence has been initialized as an empty sequence.
//
// MODIFICATION MEMBER FUNCTIONS for the sequence class:
// void start( )
// Postcondition: The first item on the sequence becomes the current
// item (but if the sequence is empty, then there is no current item).
//
// void advance( )
// Precondition: is_item returns true.
// Postcondition: If the current item was already the last item in the
// sequence, then there is no longer any current item. Otherwise, the
// new current item is the item immediately after the original current
```

```

// item.
//
// void insert(const value_type& entry)
// Precondition: size( ) < CAPACITY.
// Postcondition: A new copy of entry has been inserted in the
// sequence before the current item. If there was no current item,
// then the new entry has been inserted at the front of the sequence.
// In either case, the newly inserted item is now the current item of
// the sequence.
//
// void attach(const value_type& entry)
// Precondition: size( ) < CAPACITY.
// Postcondition: A new copy of entry has been inserted in the
// sequence after the current item. If there was no current item, then
// the new entry has been attached to the end of the sequence. In
// either case, the newly inserted item is now the current item of the
// sequence.
//
// void remove_current( )
// Precondition: is_item returns true.
// Postcondition: The current item has been removed from the sequence,
// and the item after this (if there is one) is now the new current item.
//
// CONSTANT MEMBER FUNCTIONS for the sequence class:
// size_type size( ) const
// Postcondition: The return value is the number of items in the
// sequence.
//
// bool is_item( ) const
// Postcondition: A true return value indicates that there is a valid
// "current" item that may be retrieved by activating the current
// member function (listed below). A false return value indicates that
// there is no valid current item.
//
// value_type current( ) const
// Precondition: is_item( ) returns true.
// Postcondition: The item returned is the current item in the
// sequence.
//
// VALUE SEMANTICS for the sequence class:
// Assignments and the copy constructor may be used with sequence
// objects.

```

Provide some additional useful member functions, such as:

1. **insert_front**: insert a new value at the front of the sequence. This new item should now be the current item.
2. **remove_front** : remove the value at the front of the sequence. The new front item should now be the current item.
3. **attach_back** : insert a new value at the back of the sequence. This new item should now be the current item.
4. **end** : The last item in the sequence should now be the current item.
5. **operator+** and **operator+=** : These operators should have the precondition that the sum of the sizes of the two sequences being added is smaller than the **CAPACITY** of a sequence.

Now let's make a **sorted sequence**. Suppose that you implement a sequence where the `value_type` has a comparison operator `<` to determine when one item is “less than” another item. For example, integers, double numbers, and characters all have such a comparison operator (and classes that you implement yourself may also be given such a comparison). Rewrite the sequence class using a new class name, `sorted_sequence`. In a sorted sequence, the `insert` function always inserts a new item so that all the items stay in order from smallest to largest. There is no `attach` function. All the other functions are the same as the original sequence class.

The names of the files that you submit should be `sequence.h`, `sequence.cpp`, `sorted_sequence.h`, and `sorted_sequence.cpp`. When you upload your solution to Camino, please make sure that your sequence is set up to hold the `value_type` of `double` and has a capacity of 50.