Project #3 - Linked Sequence Data Structure

Learning Objectives

- Apply object-oriented programming concepts in C++
- Design and implement a data structure for a specified abstract data type
- Use pointers and appropriate memory management methods in C++

Overview

Many languages include “enhanced” array classes that include features of both arrays and linked-lists. The Java ArrayList class and the Vector class from the C++ standard template library are examples. For this project, you will constructe a Sequence class that supports random access like an array, but also allows dynamic insertion and removal of new elements.

The Sequence Class

Your Sequence Class should be implemented as a doubly-linked list. Here are a few example of how your Sequence class will be used:

```
Sequence mySequence(5);  // create a Sequence of length 5 (indexes 0 through 4)
mySequence[0] = 1;
mySequence[1] = 2;
mySequence[4] = 3;
```

After executing this code block, your sequence would appear as follows:

Index	0	1	2	3	4
Content	1	2	*?*	*?*	3

Note: Sequence locations with a content of ? can contain any value.

The push_back() member function grows the Sequence by adding values to the end. The call `mySequence.push_back(20)` produces:

Index	0	1	2	3	4	5
Content	1	2	*?*	*?*	3	20

We can also grow the Sequence using the insert() member function. The first argument to insert is the index (position) at which to insert the new data. The second argument is the data to insert. The data is inserted at the designated position, and the remaining items in the sequence are shifted to the right. Starting with the previous Sequence, `mySequence.insert(1, 30)` would produce:

Index	0	1	2	3	4	5	6
Content	1	30	2	*?*	*?*	3	20

We can reduce the size of the Sequence using pop_back(), which removes the last element of the Sequence, or erase(). The call **`mySequence.erase(3,2)`** removes 2 items starting at position 3, producing:

Index	0	1	2	3	4
Content	1	30	2	3	20

Required Member Functions

You will be provided a starting template for your sequence, implemented in the files **`Sequence.h`** and **`Sequence.cpp`**. You must implement each of the functions defined in these files. A full description of each function is given in the header file (**`Sequence.h`**).

Error Handling

Your solution should throw an exception if a user attempts to access an item outside of the bounds of the sequence in any member function. For example, each of the following calls after the Sequence of length three is created should throw an exception:

```
Sequence mySequence(3);          // mySequence has elements 0 through 2
mySequence[3] = 100;             // Error: There is no element 3
cout << mySequence[-1] << endl;  // Another error
mySequence.erase(2,5);           // Error: Tries to erase non-existent elements
```

You can throw a C++ exception using the following syntax:

```
#include <exception>

Sequence::value_type& Sequence::operator[]( index_type position )
{
  if ( index position is invalid ) {
    throw exception();
  }
  ...
}
```

The Test Harness

For this project (only), you will be provided with the entire test harness that I will use to test your code. The number of points allocated for passing each test is given in the grading rubric at the end of this assignment.

Documentation and Style

CS 3100 is the capstone programming course of your academic career. You should employ professional best practices in your coding and documentation. These practices include:

- **Descriptive variable names** – Avoid variable names like `p` and `temp`. Use names that describe the purpose of the variable. Use either underscores or camelCase for multiple-word names. Examples of appropriate variable names include: `currentNode`, `toBeDeleted`, and `studentList`. *Exception: It is generally acceptable to use single-letter variables, such as `i`, as loop indices in `for` loops.*
- **Function header comments** – At a minimum should include the name of the function, its purpose, a list of input parameters, the return value, and any side effects.
- **Whitespace and inline comments** – Code should be broken into blocks of ~5 to 20 lines of code, separated by whitespace, and annotated with inline comments. Any particularly hard to understand lines of code should have an explanatory inline comment

Example of appropriate code style

```
// Function:  addGrade -- add a grade to a Student object
// Inputs:    theGrade -- grade to be added (char)
// Returns:   boolean -- true if successful, false if
//                       unable to add grade
// Side effects:  The new grade is added to the linked
//                list of grades for the Student object

bool Student::addGrade(char theGrade) {

// Create a new linked list node with the
// grade to be added
GradeNode *newGrade, *currentGrade;
newGrade = new GradeNode();
newGrade->grade = theGrade;

try {
        // If the list is empty, point head at
        // the new grade node
        if (!head) {
                head = newGrade;
        }

        // Otherwise find the last node in the list,
        // and point its next pointer at the new
        // grade node.
```

```
                else {
                        currentGrade = head;
                        while (currentGrade->next) {
                                currentGrade = currentGrade->next;
                        }
                        currentGrade->next = newGrade;
                }
        }

        // If unsuccessful for any reason, return false
        catch(exception &e){
                return false;
        }

        // If successful return true
        return true;
}
```

Turn in and Grading

- Your Sequence class should be implemented as a separate .h and .cpp file.
- If you define other classes, (such as SequenceNode) you may use a separate class, or a class defined entirely within Class Sequence.
- Zip your entire project directory and turn it in via pilot. Make sure that all your files are included.

Code that will not compile will receive a grade of zero.

This Project is worth 50 points, distributed as follows:

Test	**Points**
Can correctly create a Sequence, modify items with the [] operator, and print the contents of the sequence with the << operator	5
Can create multiple, independent Sequence objects and print them	3
`push_back()()` correctly adds to the end of an existing Sequence	3
`push_back()` can add elements to an empty Sequence	3
`pop_back()` correctly removes the last element of a Sequence	3
`pop_back()` on an empty Sequence throws an exception	1
`insert()` correctly adds an element in the correct position of an existing Sequence	3
`insert()` throws an exception when called with an invalid index	1
`front()` correctly returns the first element of a Sequence	2
`front()` throws an exception when called on an empty Sequence	1

`back()` correctly returns the last element of a Sequence	2
`back()` throws an exception when called on an empty Sequence	1
`empty()` returns 1 if the Sequence is empty, 0 otherwise	1
`size()` correctly returns the size of a Sequence	1
`clear()` correctly removes all elements of a Sequence	2
`erase()` correctly removes specified elements of a Sequence	3
`erase()` throws an exception when called with invalid parameters	1
The assignment operator (=) correctly produces an independent copy of a Sequence	3
The copy constructor correctly produces an independent copy of a Sequence	3
Code has no memory leaks	3
Your code is well organized, clearly written, and well-documented	5