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 construct 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 examples of how your Sequence class will be used:

```
Sequence mySequence(5); // create a Sequence of length 5 (indexes 0 through 4)
mySequence[0] = "function";
mySequence[1] = "parameter";
mySequence[4] = "pointer";
```

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

Index	0	1	2	3	4
Content	"function"	"parameter"	???	???	"pointer"

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("const")` produces:

Index	0	1	2	3	4	5
Content	"function"	"parameter"	???	???	"pointer"	"const"

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, "constructor")` would produce:

Index	0	1	2	3	4	5	6
Content	"function"	"constructor"	"parameter"	???	???	"pointer"	"const"

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	"function"	"constructor"	"parameter"	"pointer"	"const"

Class Members

Your Sequence class will implement the given methods, as well as contain certain member variables. The Sequence class should be *declared* in `Sequence.h` and *defined* in `Sequence.cpp`.

Methods

At a *miminum*, you need to implement the following methods (member functions) of the Sequence class. The method signature **must** exactly match those given here.

Note: The method names below have `Sequence::` before them. You should only put the class name before the method name in the source (`.cpp`) file. You will get errors if you put them in the header (`.h`) file.

```
// Creates an empty sequence (numElts == 0) or  a sequence of numElts items
// indexed from 0 ... (numElts - 1).
Sequence::Sequence(size_t sz = 0)

// Creates a (deep) copy of sequence s
Sequence::Sequence(const Sequence& s)

// Destroys all items in the sequence and release the memory
// associated with the sequence
Sequence::~Sequence()

// The current sequence is released and replaced by a (deep) copy of sequence
// s. A reference to the copied sequence is returned (return *this;).
Sequence& Sequence::operator=(const Sequence& s)

// The position satisfies ( position >= 0 && position <= last_index() ).
// The return value is a reference to the item at index position in the
// sequence. Throws an exception if the position is outside the bounds
// of the sequence
std::string& Sequence::operator[](size_t position)

// The value of item is append to the sequence.
void Sequence::push_back(std::string item)

// The item at the end of the sequence is deleted and size of the sequence is
// reduced by one. If sequence was empty, throws an exception
void Sequence::pop_back()

// The position satisfies ( position >= 0 && position <= last_index() ). The
// value of item is inserted at position and the size of sequence is increased
// by one. Throws an exceptionif the position is outside the bounds of the
// sequence
void Sequence::insert(size_t position, std::string item)

// Returns the first element in the sequence. If the sequence is empty, throw an
// exception.
std::string Sequence::front() const

// Return the last element in the sequence. If the sequence is empty, throw an
// exception.
std::string Sequence::back() const

// Return true if the sequence has no elements, otherwise false.
bool Sequence::empty() const

// Return the number of elements in the sequence.
size_t Sequence::size() const

// All items in the sequence are deleted and the memory associated with the
```

```
    // sequence is released, resetting the sequence to an empty state that can have
    // items re-inserted.
    void Sequence::clear()

    // The item at position is removed from the sequence, and the memory
    // is released. If called with an invalid position throws an exception.
    void Sequence::erase(size_t position)

    // The items in the sequence at ( position ... (position + count - 1) ) are
    // deleted and their memory released. If called with invalid position and/or
    // count throws an exception.
    void Sequence::erase(size_t position, size_t count)

    // Outputs all elements (ex: <4, 8, 15, 16, 23, 42>) as a string to the output
    // stream. This is *not* a method of the Sequence class, but instead it is a
    // friend function
    friend ostream& operator<<(ostream& os, const Sequence& s)
```

Data Members

The Sequence class needs at least three data members. You must have both a **`head`** and **`tail`** pointer to the first and last nodes in the linked list. You also need a `size_t` to keep track of how many elements are in the Sequence.

The SequenceNode Class

Because the Sequence class is a **doubly-linked list**, you will need to implement a class for the nodes of the list. Each node **must** store at least a **`std::string`** to store the data for the node. To make the list doubly-linked, each node will have two pointers: one pointing to the next node in the sequence and the other pointing to the previous node in the sequence. For the **`head`** node, the previous node pointer will be `nullptr` and for the **`tail`** pointer, the next pointer will be `nullptr`. Here is an example of a possible **`SequenceNode`** class that you are free to use. You can put this inside **`Sequence.h`**, or you may make a separate header and source file for **`SequenceNode`**.

```
class SequenceNode {
  public: // to make it easier, we can make the data members public so we don't need
          // getters and setters
    SequenceNode* next; // pointer to next Node. If node is the tail, next is
                        // nullptr
    SequenceNode* prev; // pointer to previous Node. If node is the head, prev is
                        // nullptr
    std::string item; // the element being stored in the node

    //default constructor, ensure next and prev are nullptr
    SequenceNode() : next(nullptr), prev(nullptr)
    {}

    /// parameterized constructor, next and prev are set to nullptr and the
    /// node's element is set to the given value
    SequenceNode(std::string item) : next(nullptr), prev(nullptr), item(item)
    {}
  };
```

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();
  }
  ...
}
```

Test Harness

For this project (only) you will be provided with the entire test harness that will be used to test and grade 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.

IMPORTANT NOTES

You should make a habit of committing your code regularly. If your repository shows a few large commits and few others, that will flag your submission for potential academic integrity violation.

The course policy disallows A.I. on ANY assignment. Any A.I. use will result in a zero on the assignment and an academic integrity violation.

Programming assignments for this course are INDIVIDUAL. Submissions are partially or fully copied from another submission will receive a zero and an academic integrity violation

Code that will not compile will receive a grade of zero. Also note, if you include a .cpp file, your code will not compile in our tests.

Turn in and Grading

Your Sequence class should be implemented as a separate header (Sequence.h) and source (Sequence.cpp) file. If you define other classes, (such as SequenceNode) you may use a separate class, or a class defined entirely within class Sequence. The last commit made to your assignment repo that is on the remote branch will be graded.

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
Total	**50**