T-201-GSKI, GAGNASKIPAN

DOUBLY LINKED LISTS

Assignment grading. For full marks your solution needs to be accepted by Mooshak. Points may be deducted for

- not implementing the solution according to the specifications (including worst-case time complexity),
- redundant or repeated code.

If your solution is not accepted by Mooshak, it will receive a maximum grade of 7 for that part.

StringList

In this part you will implement an abstract data type *list*, for strings, using a doubly linked list. The data structure should be implemented as the class `StringList`. The interface and implementation of the `StringNode` struct, which is used to construct the doubly linked list, is given in *StringNode.h*. The interface of the class `StringList` is given in *StringList.h*. Your task is to implement this interface in *StringList.cpp*.

A detailed description of the ADT list is given in chapter 4.1 in the main textbook (Shaffer).

The interface of the `StringList` class is as follows.

```
class StringList {
    public:
        StringList();
        ~StringList();

        // Clear contents from the list, to make it empty.
        // Worst-case time complexity: Linear
        void clear();

        // Insert an element at the current location.
        // item: The element to be inserted
        // Worst-case time complexity: Constant
        void insert(const string& item);

        // Append an element at the end of the list.
        // item: The element to be appended.
        // Worst-case time complexity: Constant
```

Last updated January 31, 2015.

```
    void append(const string& item);

    // Remove and return the current element.
    // Return: the element that was removed.
    // Worst-case time complexity: Constant
    // Throws InvalidPositionException if current position is
    // behind the last element
    string remove();

    // Set the current position to the start of the list
    // Worst-case time complexity: Constant
    void move_to_start();

    // Set the current position to the end of the list
    // Worst-case time complexity: Constant
    void move_to_end();

    // Move the current position one step left. No change
    // if already at beginning.
    // Worst-case time complexity: Constant
    void prev();

    // Move the current position one step right. No change
    // if already at end.
    // Worst-case time complexity: Constant
    void next();

    // Return: The number of elements in the list.
    // Worst-case time complexity: Constant
    int length() const;

    // Return: The position of the current element.
    // Worst-case time complexity: Constant
    int curr_pos() const;

    // Set current position.
    // pos: The position to make current.
    // Worst-case time complexity: Linear
    // Throws InvalidPositionException if 'pos' is not a valid position
    void move_to_pos(int pos);

    // Return: The current element.
    // Worst-case time complexity: Constant
    // Throws InvalidPositionException if current position is
    // behind the last element
    const string& get_value() const;
```

```
        // Outputs the elements of 'lis' to the stream 'outs', separated
        // by a single space.
        friend ostream& operator <<(ostream& outs, const StringList& lis);
};
```

Current position. Note that, for a list of n elements, there are $n + 1$ current positions, since the current position is either between two elements of the list or at the front or the end of the list. If you use a `StringNode` pointer to keep track of the current node (as is recommended), you can run into some trouble when you only have n elements to point to, but you need to be able to keep track of $n + 1$ positions.

One way of dealing with this is using `NULL` to denote the first position (i.e. position 0). You then need to be careful in your implementation to handle this case correctly.

Another way is to use a sentinel node (as is done in the textbook). A sentinel node (sometimes called a dummy node) is a special node that is added to the list, but does not hold any value of the list. Using a sentinel node can help reduce the number of special cases you need to consider when implementing your list.

More notes about the implementation.

- No private members are specified in `StringList.h`, you must decide which private member variables and functions you want to use.
- The function `clear()` should release all the memory that has been allocated for the list.
- You are encouraged to implement a copy constructor and `operator=` for `StringList`, but it is not required. Neither will be tested on Mooshak.
- The operator `<<` should output the elements of the list, separated by a single space.
- Make sure your functions handle empty lists.
- Please comment your code.

Submitting. To submit this problem to Mooshak you must create a zip file containing *StringNode.h*, *StringList.h* and *StringList.cpp*. You can include other files, such as *main.cpp*, but that is not necessary.