CMPT 225 Assignment 2 (5%)

Submit your solutions by Wednesday, February 6, 2019 3:00pm.

Marking Criteria:

For all problems, your solutions will be judged by how appropriately you solved the problem. That includes not just the correctness and efficiency of your code, but also how your code is presented, i.e., coding style. The best code contains class invariants, assertions, preconditions, postconditions, and descriptions of your algorithms, written in a high-level voice.

Follow the instructions carefully. Changing header files is particularly hazardous because your code may not build when we try to test it. Code that doesn't build never receives a mark much higher than 0.

0. [0 marks] *Care Package Download*

   Download the care package from the course web page and `unzip` it. It contains your base code.

1. [30 marks] *Linked-List Stack*

   The overall goal of this question is to implement a Stack ADT using a linked-list data structure which *pushes the topmost element to the tail of the list.* This will be different than the in-class implementation where the elements were pushed to the head of the list.

   We're not suggesting this is a particularly *good* choice of data structure, that is it will force at least one of the operations to have a $O(n)$ number of steps. The point of the exercise is to give you an opportunity to program an ADT with a linked list.

   What You Need to Write

   You are going to complete the implementation of a Stack ADT with a specific data structure. That data structure is a singly linked list in which pushed items are placed at the *tail* of the linked list. (Similarly, popped items will be removed from the tail of the list.)

   Students who submit a linked list implementation where the top of the stack was at the head of the list will receive 0 for this question.

   Complete your implementation in `Stack.h` and `Stack.cpp`, and then submit them to CourSys.

2. [10 marks] *Running-Time Analysis*

   Referring to the proposed implementation in Question 1, analyze the total running time required to push n items to the Stack. Next, analyze the total running time required to pop those n items from the Stack. A detailed analysis is expected, i.e., if you present a final answer only, you will be unhappy with your grade.

   Type your analysis in `Analysis.txt` and submit to CourSys.

3. [30 marks] *Evaluating Infix Expressions*

Though a stack can be employed to calculate expressions in postfix notation, the reality is most humans don't write expressions that way: we tend to use infix notation. There is an algorithm to calculate infix expressions as well, and it requires two stacks: one for numbers and the other for operators.

Decisions are made based on the next input token `T` (either a number, an operator or EOF) and the top of the operator stack as follows:

```
while T is not EOF or the operator stack is non empty

    if T is a number:
        push T to the number stack; get the next token

    else if T is a left parenthesis:
        push T to the operator stack; get the next token

    else if T is a right parenthesis:
        if the top of the operator stack is a left parenthesis:
            pop it from the operator stack; get the next token:
        else:
            pop the top two numbers and the top operator
            perform the operation
            push the result to the number stack

    else if T is +, - or EOF:
        if the operator stack is nonempty and the top is one of +, -, *, /:
            pop the top two numbers and the top operator
            perform the operation
            push the result to the number stack
        else:
            push T to the operator stack; get the next token

    else if T is * or /:
        if the operator stack is nonempty and the top is one of *, /:
            pop the top two numbers and the top operator
            perform the operation
            push the result to the number stack
        else:
            push T to the operator stack; get the next token
```

Your task for this problem is to code this algorithm in C++.

Specification

- Your program will evaluate a single well-formed arithmetic expression from standard input, and display the result on standard output.
- You will implement the algorithm given above. The point of this problem is for you to use a stack to solve a problem. If you decide to solve this problem by making a system call, or by writing your own parser, etc, you will receive a grade of 0 for this question.
- All input numbers are positive integers. Thus you will do integer arithmetic.
- Your program may assume the input is well-formed, i.e., the behaviour may be indeterminate on incorrect input (bad infix expressions).
- Name your file `eval.cpp` and submit to CourSys.

Some Help

- You are provided a `Scanner` class that produces Tokens for you. Please read `Scanner.h` to see what objects of type `Token` look like.
- You will use the provided `Stack` for both your number stack and operator stack. Please check the documentation for the behaviour of `.pop()` before using it.
- There are a few sample expressions provided to you to torture test your code. You should come up with some of your own test cases as well.

4. [30 marks] *Augmenting a Queue (Circular Array)*

   Included in your care package is the implementation of a Queue ADT by a static array (circular array). If you build and run the test driver as it is, the queue won't be big enough to handle all the elements: some elements will be lost due to an overrun. You could rebuild the code for a different value of `INITIAL_SIZE`, but that won't work in every case. If you need something that can grow to an appropriate size, a static array won't do.

   If you recall the `capacity` attribute from Assignment 1, this is where it will come into play. The strategy will be to augment the Queue so that the capacity may grow as needed.

   (a) Open `Queue.h` and change the static array declaration to a pointer `int * arr`. Next, update the constructor in `Queue.cpp` so it dynamically creates the array.

   (b) Classes that use dynamic memory also need a destructor. You should add it now.

   (c) Resizing the array is a relatively expensive operation. You need to find a larger space, copy the elements from the old array into the new array, and recycle the old array. Overall, this is an $O(n)$ operation, and should occur sparingly. One effective strategy is to double the capacity of the Queue whenever you enqueue into a full array. The expensive resizing operations are amortized across enough operations that they don't become an issue. (For more detail, please read the Appendix on expandable arrays.)

   Implement this strategy by re-writing `.enqueue(x)`.

   (d) To have an array that has too large a capacity compared to the number of elements is also bad. It is a waste of space. One good strategy is to halve the capacity of the Queue whenever the array is less than 1/4 full. However, the min capacity cannot drop below the value of `INITIAL_SIZE`.

   Implement this strategy by re-writing `.dequeue()`.