CMPT 225 Assignment 2 - Stack, Queue and Expandable Array

Please, read the entire assignment first before starting it!

Please, read the Submission part of this assignment to determine what constitutes Part 1 and Part 2 of this assignment!

This assignment can be done in pairs -> form groups of two on CourSys (see "About Groups" at the end of this document).

Question 0

First, please, download Assn2-Files.zip and unzip it in your **sfuhome/cmpt-225/A2** folder. It contains your base code for Q1, Q3 and Q4.

Question 1 - Linked list-Based Stack ADT Class

The overall goal of this question is to implement a Stack as an ADT class using a linked-list data structure (CDT).

Careful: This Stack must have its top at the back of the linked list (as opposed to the front). This is different than the linked list-based Stack implementation we saw in our lectures where the elements were pushed at the front (head) of a SHSL list and popped from the front of this SHSL list.

We are not suggesting that locating the top of a Stack at the back (as opposed to the front) of its underlying data structure (i.e. linked list) is a particularly good way of designing your Stack ADT class since this design renders the operations of the Stack rather inefficient.

The point of the exercise is to give you an opportunity to completely design and implement a data collection ADT class using a linked list as its underlying data structure (CDT).

What You Need to Do

You are to design and implement a Stack ADT class with its underlying data structure such that it satisfies the above requirements. This data structure must be a singly headed singly linked list (SHSL list).

To get you started, you will find a partial **Stack.h** file in the zip file you have downloaded. You will see that the only thing this file contains is a private **StackNode** class with public data members. This class can be used to create a linked list. It is yet another way of creating a Node non-ADT class. Unlike the Node non-ADT class we presented in class, this StackNode class is embedded into another class.

Students who submit a linked list implementation where the top of the Stack is at the front of the linked list will receive 0 for this question.

Save your implementation in files called **Stack.h** and **Stack.cpp**, and then submit them to CourSys.

Code that doesn't build and execute on our ***target machine*** will receive 0 for this question.

Question 2 - Time Efficiency Analysis

Referring to the Stack ADT class you have implemented in Question 1, analyze the total time required to push n elements to the Stack and express it using the Big O notation.

Next, analyze the total time required to pop those n elements from the Stack and express it using the Big O notation. Note that you are asked to analyse **pop( )** not **popAll( )**.

A detailed analysis is expected, i.e., if you present a final answer only, you will be unhappy with your grade. In other words, "show your work".

Type your analysis and save it in a file called **Analysis.pdf** and submit this file to CourSys.

Question 3 - 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 question is to code this algorithm in C++.

Requirements

- Your program will evaluate a single well-formed arithmetic expression from standard input, and display the result on standard output.
- You must implement the algorithm given above. The point of this question is for you to use two Stack objects to solve a problem. If you decide to solve this problem by making a system call, or by writing your own parser, etc, and not implementing the algorithm above, you will receive 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). Also, when marking, the marker will not be using ill-formed test data.
- Write the solution in the provided file **Eval.cpp** (you can replace and/or use the code found in this file) and submit it 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 and **Scanner.cpp** to figure out what the Scanner does with these Token objects.
- You must use the provided Stack class for both your number Stack and operator Stack. It is implemented as a template class. This is why everything about this Stack class is in one file: the **Stack.h** file. We shall soon be talking about templates in our labs. Please read the documentation in order to know how **pop()** behaves before using it.
- There are a few sample expressions provided to you to "torture test" your code in the **Samples** folder. You must create some test cases of your own as well.
- In order to use the files containing these sample expressions, you can use input redirection:

  ```
  $./infixeval < Samples/expn.1
  10
  ```

  The above command will execute the executable **infixeval** and will supply the sample expression found in file **expn.1** (stored in the Samples folder) as input to infixeval. You will not have to type the infix expression (i.e., the input test data) at the command line yourself. The executable **infixeval** will then evaluate the infix expression contained in **expn.1** and, if correctly implemented, will output the result **10** on the computer monitor screen.

Code that doesn't build and execute on our ***target machine*** will receive 0 for this question.

Question 4 - Expanding a Queue (Circular Array)

Included in **Assn2_Files.zip** is the array-based implementation of a Queue ADT class (circular array).

If you build and run the Queue test driver as it is, the Queue won't be big enough to handle all the elements from the test data: some elements will be lost due to an overrun. You could rebuild the code for a different value of INITIAL_CAPACITY, but that won't work in every case. If you need something that can grow to an appropriate capacity, an automatically allocated array won't do. The strategy is to dynamically allocate memory for the Queue such that its capacity may grow/shrink as needed.

1. Open **Queue.h** and change the automatically allocated array declaration to a pointer to an array.
2. Next, update the appropriate method in **Queue.cpp** such that it dynamically creates the array.
3. Classes that use dynamic memory also need a destructor, a copy constructor as well as an overloaded assignment = operator. You must implement these three methods.
4. 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 release 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 enqueue() operations that they don't become an issue. (For more detail, please read the document called **Expandable Array** posted under Lecture 3 on our course web site.) Implement this strategy by re-writing **enqueue(newElement)**. Feel free to add a private helper method to the Queue class, if you need one.
5. 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 minimum capacity cannot drop below the value of INITIAL_CAPACITY. Implement this strategy by re-writing **dequeue()**.
6. You may also notice that there are a few other problems with the provided implementation of the methods of this Queue. For example, elementCount may be incorrectly set. As you are making the above changes, ensure that you fix this problem. There are other problems that we shall not yet be able to fix as they will require the use of exception handling. For example, we will not be able to notify the client code when the preconditions of **dequeue()** and of **peek()** have not been satisfied. To do so, we will need to make use of exception handling which we will cover soon in one of our labs. Therefore, we shall have to wait for our next assignments in order to fix such problems.
7. Submit **Queue.h** and **Queue.cpp** to CourSys.

Code that doesn't build and execute on our ***target machine*** will receive 0 for this question.

And that's it!

Marking Scheme

For all problems in this Assignment 2, your solutions will be judged by how appropriately you solved the problem. This includes code correctness and efficiency (except for Question 1 which is asking for an inefficient implementation) as well as good programming style (GPS). Good programming style includes proper documentation: descriptions of your methods, class invariants, preconditions (if there are any), postconditions (if there are any), and time efficiency expressed using the Big O notation.

Remember our *target machine*?

You can use any C++ IDE you wish to do your assignments in this course as long as the code you submit compiles and executes on our ***target machine***, i.e., the CSIL workstations running the Ubuntu Linux platform (O.S.).

Why? Because your assignments will be marked on the ***target machine*** (Ubuntu Linux, C++ and g++ versions running on the CSIL workstations).

Also, you must use the provided makefile when compiling your code, if there is one provided. Otherwise, you are free to create your own.

Submission

Assignment 2 Part 1 is due Friday June 7 at 23:59:59 on CourSys. For Part 1, you need to submit your solution to Question 1, i.e., **Stack.h** and **Stack.cpp**, and your solution to Question 2, i.e., **Analysis.pdf**.

Assignment 2 Part 2 is due Friday June 14 at 23:59:59 on CourSys. For Part 2, you need to submit your solution to Question 3, i.e., **Eval.cpp**, and your solution to Question 4, i.e., **Queue.h** and **Queue.cpp**.

Late assignments will receive a grade of 0.

About Groups

Since this assignment can be done in pairs, you are required to form groups of two on CourSys in order to submit your work and receive a mark for it.

You can do this assignment on your own if you wish, but you still must form a group on CourSys: a group of one.

Only one group member is required to do the submission for the group.
