Anand Patel

1).Explain briefly how you implemented the DoubleStack. Explain how your implementation guarantees an efficient usage of the allocated memory.

Ans) I created the template of DoubleStack and created it in such a way that it uses as much space as the length of the line requires in the code. For the Doublestack problem it uses the first number given in the code and it makes doublestack as long as that, if it tries to put anything more into it then the size given then it will throw runtimeexception and end program or if it tries to take anything out of empty stack then it will throw emptystackexception.

For Infix to prefix and evalprefix it takes the length of the line and then create DoubleStack of that length. So it doesn't use anything more or anything less than needed.

2. 2.Describe the algorithm you implemented for the conversion from infix to prefix and justify its correctness.

Ans) So I read the whole line in the program and then use string tokenizer to remove all the spaces in the program. Then I reverse the string in the code and after doing that I use the length of the string to create the DoubleStack in the program. After that it checks for operands and operator and then puts the operands and the closing parenthesis in the operator stack. Everytime a open parenthesis comes it pop till the closing parenthesis (excluding closing) or if the operator's precedence is lower than one in the stack then it will pop everything till you can put that precedence. After everything is done reverse the output to print. It will give you prefix of infix.

The program is memory efficient, doesn't take more space and reads in every character correctly checks it thoroughly and outputs correct answer.

3. 3.Explain the algorithm you implemented for the evaluation of prefix expressions.

Ans) So I read the whole line then use string tokenizer to remove all the spaces in the program . Then I reverse the string in the code. I used the length of the string to initialize doublestack. After that everytime it sees a operand it will put it to stack and then when It will see a operator, it will pop two things from operand stack, do that operation and then push the answer to the operand stack again. Do it till you are left with one thing in the operand stack and that will be your final answer. I will just pop that and it will give the final answer.

4. 4.Explain how you implemented MyQueue and justify the constant time complexity of your "lookup" function.

Ans) I implemented the MyQueue structure from the site and wrote code in it creating a regular MyQueue (Not Dynamic) because I couldn't figure it out. I also added a few function to the MyQueue like resize which will resize the array if the space is not enough in the array. Then I created the iterator to iterate through the stack. Then I searched if "-", "*" or "?", if I found I performed their operations.
My constant time could have been faster if I had used hashtable then the look up function would have given the O(1) constant time. Since I didn't know how to implement the hashtable in queue, I just used array and I know it probably doesn't give O(1) constant time but it's still a fast lookup function.