**Machine Problem 1: Crazy Calculator**   
Lluz, Kimberly  
Mantua, Christine Joy

**I. Abstract**

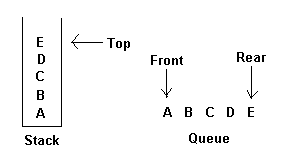
This machine problem aims to enhance understanding on Abstract Data Types by creating a GUI crazy calculator application. It is a simple calculator with simple calculating task such as addition, subtraction, multiplication and division but what makes it crazy is the code implementation used. This highlights the use of the different data structures such as arrays, linked lists, stack and queues. The specifications of Crazy Calculator were chosen to allow us to spend more time on computational thinking while figuring out and communicating well-annotated, structured, and mathematically correct solutions to quantitative problems.

**II. Introduction**  
 Stack is data structure whose functionality is in a form of a LIFO method. LIFO stands for Last In, First Out wherein the last element that has been pushed in the stack is the first element that will be popped out of the stack. Another data structure that will be used in this program is the Queue. Queue is a data structure that has a FIFO method which stands for First In, First Out. Queue’s function is similar to a line where the first element that enters the line is the first one to be taken out of the line.  
 With that simple introduction of stack and queues, we already have the idea when to use one of which data structure, but what if we implement one from the other. In this program we will be using stack as home data structure customized queues and the queues use as home data structure a pseudo array which intends to provide users similar functionality as conventional arrays. And the data structure used in implementing this is a linked list.

The task of this ‘Crazy Calculator’ is to do a simple calculation in the user’s perspective but has a crazy kind of implementation of different data structures inside that simple calculation. Behind this crazy manipulation of data structures, the calculator should be able to calculate numbers efficiently with positive inputs from the user and gives a negative output if the result is a negative number.  
  
  
**III. Backgrounder**  
  
 A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

**Stacks and Queues**

Two of the more common data objects found in computer algorithms are stacks and queues. Both of these objects are special cases of the more general data object, an ordered list.  
A stack is an ordered list in which all insertions and deletions are made at one end, called the top. A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. Given a stack S=(a[1],a[2],.......a[n]) then we say that a1 is the bottommost element and element a[i]) is on top of element a[i-1], 1<i<=n. When viewed as a queue with a[n] as the rear element one says that a[i+1] is behind a[i], 1<i<=n.



The restrictions on a stack imply that if the elements A, B, C, D, and E are added to the stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason, stacks are sometimes referred to as Last in First out (LIFO) lists. The restrictions on queue imply that the first element which is inserted into the queue will be the first one to be removed. Thus A is the first letter to be removed, and queues are known as First in First out (FIFO) lists. Note that the data object queue as defined here need not necessarily correspond to the mathematical concept of queue in which the insert/delete rules may be different.

**Infix to Postfix conversion**

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

**8 Steps Summary**

1. Print operands as they arrive.  
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.  
3. If the incoming symbol is a left parenthesis, push it on the stack.  
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.  
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.  
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.  
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.  
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

**Postfix Evaluation**

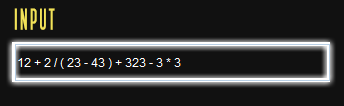
Infix Expression:

Any expression in the standard form like "2\*3-4/5" is an Infix (Inorder) expression.  
  
Postfix Expression:

The Postfix (Postorder) form of the above expression is "23\*45/-".  
  
Postfix Evaluation:

In normal algebra we use the infix notation like a+b\*c. The corresponding postfix notation is abc\*+. The algorithm for the conversion is as follows:  
 -Scan the Postfix string from left to right.  
 -Initialize an empty stack.  
 -If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be at least two operands in the stack.  
 -If the scanned character is an Operator, then we store the top most element of the stack (topStack) in a variable temp. Pop the stack. Now evaluate topStack (Operator)temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.  
 -Repeat this step till all the characters are scanned.  
 -After all characters are scanned, we will have only one element in the stack. Return topStack.

**IV. Report proper**  
 The first step is to get an input from the user. The expression inputted must have spaces in between every operator and operands. For example:

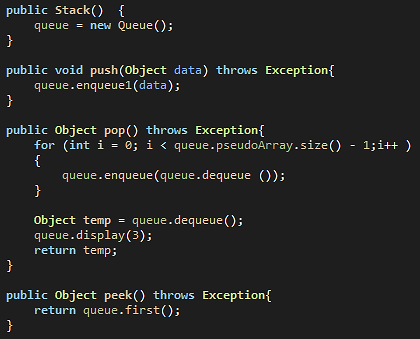


12 + 2 / ( 23 – 43 ) + 323 - 3

After having the user’s input expression to be solved, the actual computation will already take part.

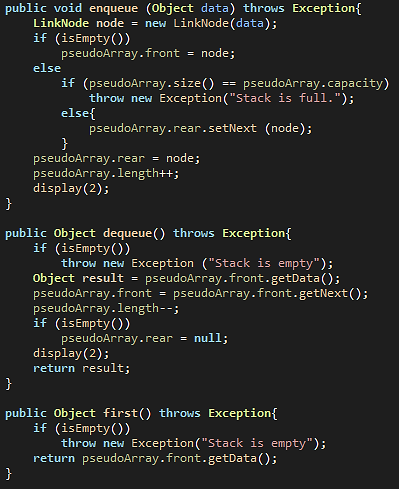
The computation process necessitates the use of a stack to assist in the calculation. The Stack data structure uses customized Queues as home data structure. The Queues use a PseudoArray as home data structure which intends to provide users similar functionality as conventional arrays and also shares the same limitations like having a static size. Finally, the underlying data structure used to implement each PseudoArray is a LinkedList.

The first step of the computation will be the conversion from infix to postfix. Since the input of the user is read as a string we need to identify if that certain character is a number or an operator. If the inputted character is neither of the two, the program will terminate. If the expression is valid, then the computation will proceed and start from converting infix to postfix.  
 We now compare the usual stack implementation of converting infix to postfix and the stack implementation of the crazy calculator in converting. In the normal stack implementation, we push the operators including the open parenthesis, and we will pop the operators with higher precedence than the current operator pushed. On the other hand, the implementation of stack in the crazy calculator will work as a queue. Our stack is implemented with the use of a queue. Below is the code of our stack class where we implement queues in pushing and popping elements.



The thing that happened here is when we pushed an operator in the stack, the stack will enter the element into a queue and when we pop the element in the stack, the implementation of the stack will dequeue the first element of the queue.

There is also our own implementation of our queue. It is implemented with the use of a linked list with a certain capacity of 12 elements. So when the element is being enqueued in the queue, the queue will make a link list.

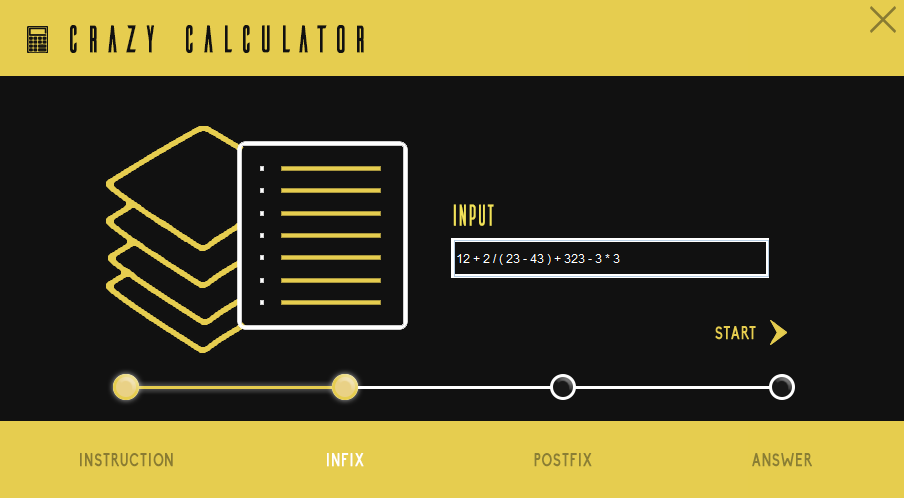


The next step after having our postfix is evaluating the postfix to come up with the answer of the expression. In the process of evaluating the postfix, we need to use another stack to push the operands and if we will encounter an operator, we will pop the two operands that were pushed in the stack and evaluate the two. If we already got the answer, we will push it back to the stack. And the same process until we only get the final answer in our stack.

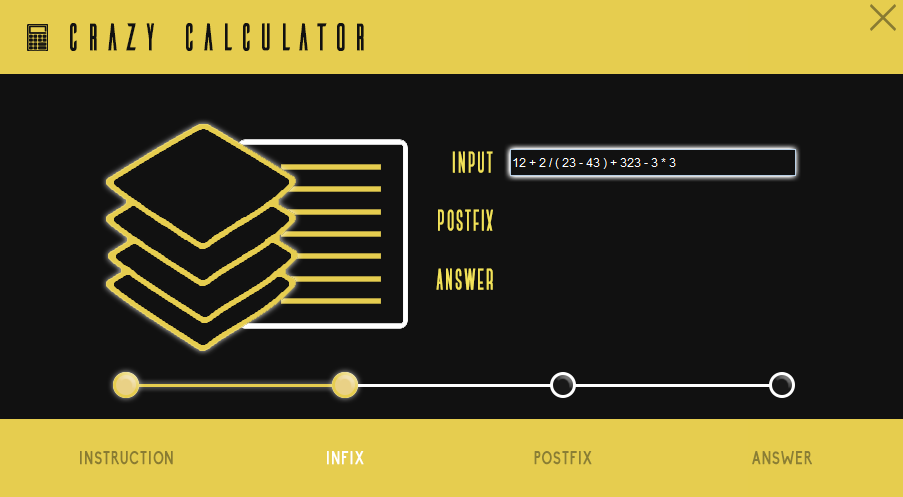
In the normal evaluation of stack, we were simply just popping and pushing the elements into our stack, but in this crazy calculator, when we push an element it will be added into a queue and our queue will be implemented through a linked list and it made this calculator a lot more challenging than the normal calculator.

**HOW THE PROGRAM WORKS**

Our program has an artistic features of graphical user interface. First, our program will ask for an arithmetic expression from the user, if the user inputted the wrong inputs it will warn the user that the input is not acceptable and will tell the user to try again.



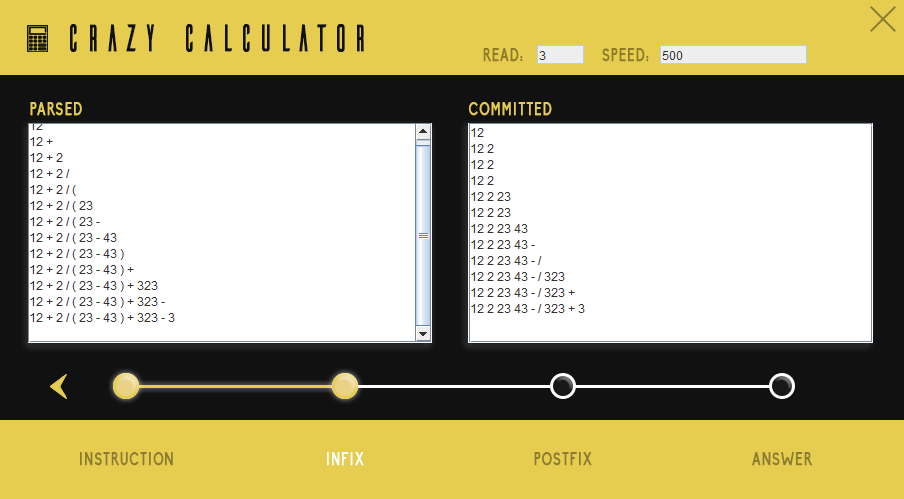
In the next step it will show the user the accepted input and will start parsing.



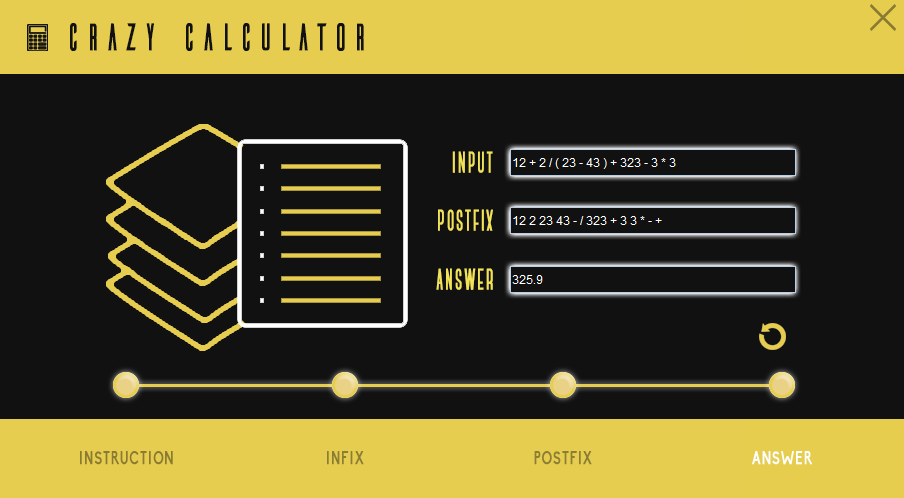
If the user will click the stacked files icon it will show the user the contents of the stack, queue, pseudoArray, and the link list. At the upper right corner of the screen there is a textbox of speed, it is where the program will identify how fast the parsing process will be.



In the other hand, if the user will click the bulleted icon, it will show the user the parsed and committed operators and operands in the process of parsing.



If the process is already done and the program already has its answer, it will show the user this kind of screen where the postfix and answer is shown.



The user can also try another expression in this program.

**V. Result and Discussion**

In this machine problem we experimented the different functionalities of some data structures namely: Arrays, linked list, stack and queues. We experimented the use of each data structures and we manipulate each of them with the implementation of other data structures.

In the normal implementation of the stack, it will be very easy for us to push and pop elements in our stack, but in our program the variations of data structures in a simple push or pop will give us more a complicated algorithm and manipulation. Though having a complex kind of implementation of the data structures, if we have the right algorithm in the manipulation of these it will still give us the correct result of the expression.

**VI. Conclusion**  
  
 In this activity, we conclude that data structures have a very powerful use in the  
field of computer science because in any data structures that we will be using in our code, if we have the right algorithm we will still come up with the right result. We conclude that we can use many data structures in a single program and it gives us the idea that data structures can be used in different ways with respect to their own concepts.

We also conclude, that we can efficiently execute simple tasks or problems no matter how flexible or challenging the implementation of the data structures is.

**VII. References**  
  
[1] Erdem, E. (1996, June). *Data Structures and Algorithms*. Retrieved from https://www.cmpe.boun.edu.tr/~akin/cmpe223/chap2.htm

[2] Rouse, M. (2006, February). *Data structure*. Retrieved from http://searchsqlserver.techtarget.com/definition/data-structure

[3] [Pillai](http://www.qiksearch.com/), P. (2002)*. Postfix evaluation*. Retrieved from http://scriptasylum.com/tutorials/infix\_postfix/algorithms/postfix-evaluation/

[4] Wolf, C. (1999, November). *Infix to postfix conversion algorithm.* Retrieved from http://csis.pace.edu/~wolf/CS122/infix-postfix.htm