# Assignment 3 Part 2 Report

May 9, 2017

**Program Description** This program contains the implementation of both a templated Stack and Queue Data Structure based from a templated Doubly Linked List Data structure as well as a PostFix Calculator that can output the resultant value of a user input equation, each of which are accompanied by extensive test cases that test program functionality and reliability. The templated LinkedStack and LinkedQueue classes were both implemented by the Linked List Data Structure as well as the node Abstract Data Type in order to visualize the data of the Queue and Stack. The testing cases for the LinkedStack and LinkedQueue are provided below, as well as testing cases for the Postfix Calculator.

**Purpose of the Assignment** The purpose of this assignment was to learn about and implement not only the Linked List Data Structure, but both the Stack and Queue Data Structures as well as implementing a Postfix Calculator that uses both the Stack and Queue Data Structures. Knowledge about generic programming, template types, program control flow, and inheritance were also gained upon implementing the Postfix Calculator and the templated Stack and Queue Data Structures.

**Data Structures Description (Stack and Queue)** The primary data structure utilized in this program was the Doubly Linked List Data Structure which is essentialy a list of sequential nodes that are allocated on the heap that point to either the next or previous node and can be traversed in either direction. Each node consists of a pointer to the next and previous node in the list as well as a data value which can be accessed. Initiallity, an empty Doubly Linked List is created with two sentinel nodes that point to each other called header and trailer that are initialized on the Stack. When nodes are added to the list, the header sentinel node points to the first node in the Linked List that is allocated on the heap, and the trailer sentinel node points to the last node in the Linked List that is allocated on the heap. Overall, the Linked List data structure is implemented with several public member functions used to organize, access, and store data of different types. Nodes can be added and removed from the Linked List, the values can be read from the first and last nodes, the Linked List can be copied, assigned, and cleared, and, the Linked List can be printed via using the respective public member class functions. The templated Stack and Queue Data Structures are based from this Linked List Data Structure and use the Linked List Data Structure as a template along with its public member functions in order to support several operations such as storing and accessing data which are listed in detail below.

**Algorithm Description** The templated Stack and Queue Data Structures are implemented with several public member functions that can used to store and access data.

**LIFO Stack Data Structure Functions (Private and Public members)**

DoublyLinkedList<T> dll - private doublylinkedlist object member.
LinkedStack<T>() - Default constructor, initializes a new stack.
~LinkedStack<T>() - Destructor, clears the stack and frees memory.
bool isEmpty() - checks if the stack is empty.
void push() - pushes a node with a specified type on top of the stack.
T pop() - removes the first node from the stack and returns its data value.
T top() - returns the first node's data from the top of the stack.
ostream& operator<<() - output operator, prints the contents of the stack.

**Queue Data Structure Functions (Private and Public members)**

DoublyLinkedList<T> dll - private doublylinkedlist object member.
LinkedQueue<T>() - default constructor, initializes a new queue.
~LinkedQueue<T>() - Destructor, clears the queue and frees memory.
bool isEmpty() - checks if the queue is empty.
T dequeue() - removes the first node from the stack and returns its data value.
T enqueue() - inserts a node with specified type in the back of the queue.
T first() - returns the first node's data from the top of the stack.
ostream& operator<<() - output operator, prints the contents of the queue.

**PostFix Calculator Description** The PostFix Calculator essentially takes a user input equation string, parses and tokenizes the string, generates the equation's postfix form, and evaluates the postfix equation returning an end value for the user to see. The Calculator supports a number of mathematical operators (+,-,*,/,(,),^), constants, and also supports the usage of variables (A-Z) which must be uppercase letters.

**PostFix Calculator Algorithm Description (Parser and Evaluator)** In order to implement the Postfix Calculator, two classes were designed, Evaluator and Parser, which utilize the Queue and Stack Data Structures to store and access the data used for the PostFix Calculator. Both the Parser and Evaluator Classes have several functions which allow for operations to be performed on the user input infix equation. The overall general program flow of the PostFix Calculator program is quite straightforward, a user inputs a correct infix equation (the equation must be terminated by the '#' character, have balanced parenthesis, and all variables must be uppercase), the infix equation string is passed through to the toPostFix() function to be parsed, the PostFix form is generated and passed through to the Evaluator class's getValue() function which evaluates the PostFix form in order to display the mathematical result of the equation. Below is a list of all functions used for both the Parser and Evaluator Classes.

**Parser Class Functions (Private and Public members)**
string infix- private string member that holds the infix equation
LinkedQueue<double> varQueue- private Queue object member for storing values for PostFix conversion
LinkedQueue<string> opQueue- private Queue object member for storing values for PostFix conversion
LinkedQueue<string> opStack- private Stack object member for storing values for PostFix conversion
bool pCheck() - Checks if user input equation is a proper equation
int getOrder(char c) - tokenizes an operator giving it a priority number
Parser(string s) - Default constructor, initializes a new parser object.
~Parser() - Destructor, frees the memory of the Parser object.
void toPostfix() - Generates the PostFix form from the infix equation.
void printInfix() - prints the infix form of the equation.
void printPostfix() - prints the postfix form of the equation.
string getInfix() - returns the string of the infix equation.

**Evaluator Class Functions (Private and Public members)**
LinkedQueue<double> varQueue- private Queue object used for evaluating the equation
LinkedQueue<string> postQueue- private Queue object member used for evaluating the equation
LinkedQueue<string> valStack- private Stack object member used for evaluating the equation
Evaluator(Parser& par) - Default Constructor, initializes a new evaluator object.
~Evaluator() - Destructor, frees the memory of the Evaluator object.
double getValue() - returns the mathematical result of the postfix equation.
void printValue() - prints the result of postfix evaluation.

**How to Compile and Run**

1. In a Unix terminal, navigate to the respective source directory /**Parser** using the **cd** command.
2. Once in the /**Parser** directory type **make all** to compile the respective program.
3. Run the executable for the PostFix Calculator program by typing **./run-main** for the PostFix Calculator program.

**PostFix Calculator Usage** Using the PostFix Calculator is straightforward. After running the executable, the program will ask you to input an infix equation, this equation must be '#' terminated, have balanced parenthesis, and any variables used in the equation must be uppercase. Once the equation is input by the user, if any variables were used in the equation, the user will be asked for their respective values. After this, the infix equation, postfix equation, and postfix value with be displayed on the screen. The program will then ask if you want to try another equation, type 'Y' for yes or 'N' for no. The program will restart if you choose yes and terminate if you choose no.

**Logical Exceptions** There are several logical exceptions for member functions of the of the LinkedQueue, LinkedStack, Evaluator, and Parser classes, primarily for handling the cases for an empty stack or empty queue, division by zero, incorrect infix equation input, and invalid character input.

**C++ Object Oriented Features** There are four classes, a LinkedQueue and LinkedStack class, a Parser Class, and an Evaluator Class. There are also exceptions and elements of inheritance throughout the program which occur in all the classes when `RuntimeException` is overridden; friend functions and classes are also utilized.

**Tests** The following are output from testing these programs:

**LinkedStack and LinkedQueue:**

The primary testing done on both the LinkedStack and LinkedQueue programs are extended from the instructor's testing cases in order to further test functionality and reliability of these data structures. The testing cases cover the use of the public member functions listed above in the **Algorithm Description** Section for the LinkedStack and LinkedQueue classes. Two instances of each a Stack and Queue are both created and populated with a string type and a floating type (double). The pop(), top(), push(), enqueue(), dequeue(), first(), isEmpty(), and output operator functions are all tested for functionality as shown below.

| | |
|---|---|
| `Part2code.png` | `Part2VSterminal.png` |

**PostFix Calculator:**

Several test cases were created alongside the instructor's test cases on the lecture slides for the PostFix Calculator in order to ensure functionality and reliability. The testing cases were chosen because they cover several different equations, constants, variables, and combinations of operators. Since the program can restart at the user's convenience, several of the exception-triggering cases were also tested, such as invalid equations, invalid inputs, division by zero, etc. The testing cases for the PostFix Calculator program are shown below.

| | |
|---|---|
| `test1.png` | `test2.png` |

**Conclusion** Overall, extensive knowledge about the Stack and Queue Data Structures, the Linked List Data Structure, generic programming principles, templates, and inheritance were obtained by implementing the PostFix Calculator and respective data structures.