CCDSALG MCO1 - CCDSALG Calculator Term 3 AY 2024 - 2025

Angelyn Kalman Chua
Thaeron Sean Conducto
Kyle Dominique Rubia
2401 Taft Avenue
Manila, Philippines
angelyn_chua@dlsu.edu.ph
thaeron_conducto@dlsu.edu.ph
kyle_rubia@dlsu.edu.ph

ABSTRACT

This document details the architecture and implementation of a calculator application developed in C, leveraging stack and queue data structures for expression evaluation. This project aims to provide a robust understanding of algorithm implementation for infix to postfix conversion and postfix expression evaluation. The stack and queue is assembled as a singly linked list. Queue. follows the 'First In First Out' rule and stores the postfix expression tokens. The stack follows the 'Last In First Out' rule and is used for operator and operand management. The infix-to-postfix conversion utilizes a variation of Dijkstra's Shunting Yard algorithm, handling various operator types and parentheses. Postfix evaluation relies on a stack-based method to process the following operations. Limitations include the assumption of syntactically correct input, restricted operand types (non-negative integers only, 0/1 for logical) and basic error handling for division by zero.

STACK AND QUEUE IMPLEMENTATION

Stack Implementation

Following the Last In First Out (LIFO) principle, the stack was implemented as a singly linked list. Each node in the stack (singlyStackNode) contains an integer value (nData) and a pointer to the next node (pNext). The stack structure itself (singlyLinkedStack) holds a pointer to the top of the stack (pTop). Dynamic memory allocation (malloc and free) is used to manage the nodes.

This data structure supports the following operations:

- initializeLinkedStack()
- emptyLinkedStack()
- pushLinkedStack()
- popLinkedStack()
- peekLinkedStack()
- clearLinkedStack()

This stack is well-suited and primarily used in Infix to Postfix Conversion and Postfix Evaluation as it can grow dynamically as elements are pushed or popped.

Queue Implementation

The queue was also implemented using a singly linked list with head and tail pointers and follows the First In First Out (FIFO) principle. Each node (singlyQueueNode) stores a string (pData) representing a token (e.g., operand or operator) and a pointer to the next node (pNext). The queue structure (singlyLinkedQueue) contains pointers to both the front (pHead) and rear (pTail) of the queue.

Below are the core functions used to manage the queue:

- initializeLinkedQueue()
- isEmptyLinkedQueue()
- createQueueNode()
- enqueueLinkedQueue()
- dequeueLinkedQueue()
- peekLinkedQueue()
- clearLinkedQueue()
- displayLinkedQueue()

The queue is used to store postfix expression tokens during conversion, which are later processed in evaluation. This implementation supports variable-length expressions and multi-digit operands. It avoids fixed-size limits by using dynamic memory, which lets the queue expand as the expression becomes more complex.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

INFIX TO POSTFIX IMPLEMENTATION

The infix-to-postfix conversion used a linked stack to temporarily hold operators and a linked queue to save the resulting postfix expression. The algorithm is a variation of Dijkstra's Shunting Yard algorithm, as described by The Shunting Yard Algorithm (n.d.), updated to handle different types of operators: arithmetic, relational, logical, and grouping symbols like parentheses. Operators were assigned unique negative integer constants to make comparisons and precedence checking easier. These constants are used during the conversion to ensure consistent processing of both single- and double-character operators.

The conversion function begins by scanning the infix string one character at a time. When digits are encountered, they are combined to form multi-digit numbers and are immediately enqueued into the postfix queue.

When an operator is encountered, a helper function checks its precedence and whether it is right-associative. The program then compares this operator with the one at the top of the operator stack. It removes any higher or equal precedence operators from the stack and adds them to the queue before placing the current operator onto the stack.

Parentheses are handled specifically: when an opening parenthesis is encountered, it is pushed onto the stack, and when a closing parenthesis is found, operators are popped and enqueued until the corresponding opening parenthesis is removed.

The helper functions are separated into smaller parts to make the code easier to manage. The main function, convertToPostfix(), puts everything together to convert the infix expression into postfix form. After reading the entire input, any remaining operators in the stack are added to the queue. The final postfix expression is then ready to be evaluated.

TESTING PROCESS & RESULTS

Testing Process

Test cases were carefully designed to cover all the calculator's functions. This included different operators, their order of operations, and special cases. The 50 provided infix expressions ranged from simple math problems to more complex ones that rigorously tests the program implementation at hand. We focused on testing multi-digit numbers, chained operations, and nested parentheses to check how well the program converts infix to postfix. We also included important error conditions, like division by zero and modulo by zero, to test the program's ability to handle errors.

Results

The testing showed that the calculator converted infix expressions to postfix and evaluated them correctly for many valid inputs. This included complex arithmetic, relational, and logical expressions. Error test cases were also added to showcase that the program does not print out the errors. The implementation managed operator precedence and associativity well. For example, 2+3*4 gave 14, and 5$\hat{2}\hat{3}$ resulted in 390625. The program recognized and reported "Division by zero error!" for cases like 10/0 and 8%(4-4). This prevented runtime crashes.

During error testing, I placed invalid/incomplete expressions like single digits and operators. The program would display multiple copies of the output instead of showing no output.

To fix this, we implemented a token count checker (countNodes) before evaluating or printing the postfix expression. This checker checks whether or not the expression has at least 3 tokens, as this is the minimum amount of variables needed to complete a valid binary expression. If the expression is less than 3 tokens, the program will skip the evaluation and printing process.

Stack and queue were updated accordingly to properly clear after each test cases to prevent any leftover data from previous inputs (since this will affect the next ones).

For evaluation.c, the evaluation cases had a lot of redundancy which makes the code harder to debug and read. To fix this, we created helper functions (exponentHelper, operatorHelper, and symbolHelper) that handle specific tasks to further simplify and organize the evaluation of these expressions, making it easier to debug the code for any errors.

LIMITATIONS

Although the program successfully converts infix expressions to postfix and evaluates them correctly in most cases, it has several limitations.

First, it assumes that all infix expressions are syntactically correct. It does not check input expressions for errors like unmatched parentheses, missing operands or operators, or invalid token sequences. If such cases arise, the outcome is unpredictable and may lead to incorrect output or runtime errors.

Second, the evaluation only supports non-negative integer operands. It does not handle negative numbers or floating-point values. For logical expressions, the operands are assumed to be strictly 0 or 1. Any deviation from this (for example, 3 && 2) is not specifically checked or converted to boolean values.

Third, while there is error handling for division and modulo by zero, the program only prints an error message and stops. It does not recover smoothly or skip to the next expression when processing multiple expressions in batch mode.

Finally, implementing the project in C, a low-level language, required manual memory management for all data structures and string handling. This method allowed for deeper control over program behavior, but it also came with risks like memory leaks, pointer errors, and segmentation faults if memory was not allocated or freed correctly. These challenges are common when working in C and highlight the difficulty of managing dynamic structures in a low-level setting.

Despite these limitations, the program successfully shows how to use stacks and queues to solve expression evaluation problems. It also follows modular and dynamic design principles.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3 ed.). MIT Press, Cambridge, MA.

[2] DigitalOcean. n.d.a. Queue in C. https://www.digitalocean.com/community/tutorials/queue-in-c. (n.d.).

[3] DigitalOcean. n.d.b. Stack in C. https://www.digitalocean.com/community/tutorials/stack-in-c. (n.d.).

[4] GeeksforGeeks. n.d. Implement Stack in C. https://www.geeksforgeeks.org/c/implement-stack-in-c/. (n.d.).

[5] Michael T. Goodrich, Roberto Tamassia, and David M. Mount. 2014. *Data Structures and Algorithms in C++* (2 ed.). Wiley, Hoboken, NJ.

[6] Math Center, Emory University. n.d. The Shunting Yard Algorithm. https://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/. (n.d.).

[7] Computer Science. 2021. The Shunting Yard Algorithm Visualization. https://www.youtube.com/watch?v=Z_iOeznK2OA. (2021).

APPENDIX A: RECORD OF CONTRIBUTION

Group Members:

- P1: Angelyn Kalman Chua
- P2: Thaeron Sean Conducto
- P3: Kyle Dominique Rubia

Activity	**P1**	**P2**	**P3**
Stack Implementation	100	0	0
Queue Implementation	0	0	100
Infix to Postfix Conversion	60	30	10
Postfix Evaluation	15	70	15
Testing	33.33	33.33	33.33
Documentation	10	50	40
Raw Total	218.33	183.33	198.33
TOTAL	**36.39%**	**3**0.56%	**33.05%**