Abby Gerstner
CS 303 Assignment 3 Readme File

How to use CS 303 Assignment 3 Stack:

1. Open ExpressionManager.h and main.cpp files.
2. Run the program.
3. Type the infix expression you want to be converted into postfix expression.
4. Press enter and the postfix expression will be displayed in the console.

Example output:

```
Enter an infix expression: 7*6+(3-2)
Postfix expression: 76*32-+
```

Assumptions:

- Valid Input Format:
  - The user is expected to input infix expressions that consist of numbers and operators (+, -, *, /, %), as well as parentheses ((), {}, []).
  - The code assumes that the input expression is a valid arithmetic infix expression.
- Balanced Parentheses:
  - The code assumes that balanced parentheses are crucial for the validity of the expression.
  - Parentheses must appear in matching pairs, and the opening and closing symbols must match correctly.
- Alphanumeric Operand Characters:
  - The code assumes that operands are alphanumeric characters.
  - Any other characters, apart from numbers, operators, and parentheses, are considered invalid.
- Operator Precedence:
  - The code assumes predefined operator precedence: + and - have lower precedence than *, /, and %.
  - Operators with higher precedence must be evaluated before those with lower precedence.
- Spaces and Tabs:
  - Spaces and tabs in the input expression are ignored.
  - The code assumes that these whitespace characters do not affect the correctness of the expression.
- Valid Operator Set:
  - The set of valid operators is assumed to be restricted to +, -, *, /, and %.
- Error Handling:
  - The code assumes that, upon encountering an invalid character in the input, it should print an error message and terminate with a non-zero return code.
- Input Validation Loop:

    - The input validation loop assumes that it is sufficient to check each character of the infix expression for validity.
- Memory Management:
    - The code assumes that memory management for stack operations is handled correctly by the standard library stack.
- Return Codes:
    - The code returns 0 upon successful execution and 1 if an error is encountered, such as unbalanced parentheses or invalid characters.

How to use CS 303 Assignment 3 Queue:

1) Open Queue.h and Queue.cpp files.
2) Run the program.
3) The output will show 6 lines of code:
    a) First line: check if the queue is empty before enqueueing
    b) Second line: check if the queue is empty after enqueueing
    c) Third line: get the front element without removing it
    d) Fourth line: get the front element after dequeuing
    e) Fifth line: print dequeued elements
    f) Sixth line: get the total number of elements in the queue
4) To change what items are enqueued/dequeued, change values within the main function.

Example of output:

```
Queue is empty.
Queue is not empty.
Front element: 10
Front element: 30
Dequeued elements: 10 and 20
Number of elements in the queue: 3
```

Assumptions:
Assumptions List for the Given Queue Code:

- Integer Data Type:
    - The queue is designed to store elements of type `int`.
    - The code assumes that elements added to the queue and returned from the queue are of integer type.
- Memory Allocation:
    - The code assumes that sufficient memory is available for dynamic memory allocation using `new` to create nodes.
- Error Handling:
    - The code assumes that an exception of type `std::runtime_error` will be thrown if an attempt is made to dequeue or access the front element of an empty queue.

- Queue Initialization:
    - Upon creating a new `Queue` object, the front and rear pointers are initialized to `nullptr`, and the count of elements is set to 0.
- Destructor Behavior:
    - The destructor is assumed to be responsible for deallocating memory for all nodes in the queue.
    - The destructor iteratively dequeues elements until the queue is empty.
- Enqueue Operation:
    - The `enqueue` operation adds a new element to the rear of the queue.
    - If the queue is empty before enqueuing, both the front and rear pointers are set to the new node.
- Dequeue Operation:
    - The `dequeue` operation removes and returns the front element of the queue.
    - If the queue becomes empty after dequeuing, both front and rear pointers are set to `nullptr`.
- Front Operation:
    - The `front` operation returns the front element of the queue without removing it.
    - Assumes the queue is not empty before invoking this operation.
- Empty Queue Check:
    - The `isEmpty` operation checks if the queue is empty.
    - Returns `true` if the queue has no elements; otherwise, returns `false`.
- Size Operation:
    - The `size` operation returns the total number of elements present in the queue.
- User Interaction:
    - The `main` function demonstrates the usage of the `Queue` class with various operations.
    - The code assumes an interactive user interface, printing messages to the console to indicate the state of the queue.
- Positive Integer Elements:
    - The example usage in `main` assumes that only positive integers are enqueued, and dequeued elements are printed.
- No Overflow Handling:
    - The code assumes that there is no overflow handling for the queue size.
- Sequential Enqueue and Dequeue:
    - The `main` function sequentially enqueues and dequeues elements from the queue for demonstration purposes.
- C++ Standard Library:
    - The code assumes compatibility with the C++ standard library, including the use of exceptions and standard input/output facilities.