CSCE 156 – Computer Science II

Lab 14.0 - Stacks & Queues

Prior to Lab

1. Review this laboratory handout prior to lab.

Lab Objectives & Topics

Following the lab, you should be able to:

- Understand how stacks and queues operate and how to use them
- Know how to implement stacks and queues using a linked list data structure
- Know how to use stack and queue data structures in an application
- Optionally, you will have exposure to advanced queue usage in a multi-threaded programming environment

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

Clone the starter code for this lab from GitHub using the following url: https://github.com/cbourke/CSCE156-Lab14.

Stacks

Postfix notation (also known as Reverse Polish Notation) is a parenthesis-free way of writing arithmetic expressions where one places the operator symbol after its two operands. For example, the expression `(3 + 2)` would be written `3 2 +`. The result of this value multiplied by 4 could be written `3 2 + 4 *`. As another example, the expression `((3 + 2) * 4) / (5 - 1)` would be written as: `3 2 + 4 * 5 1 - /`. Manually computing this would done as follows

```
     3 2 + 4 * 5 1 - /
=>   5 4 * 5 1 - /
=>   20 5 1 - /
=>   20 4 /
=>   5
```

The advantage of postfix notation is that no precedence rules are necessary as the order of operation is completely unambiguous. Many calculators, mathematical tools, and other systems utilize this notation. Evaluation of postfix expressions can be done using a stack data structure.

Recall that a stack is a LIFO (Last-In First-Out) data structure. Stacks, like many data structures, can be implemented in many different ways. In this lab, our implementation will utilize a linked list data structure. This data structure has been completely implemented for you. Most of the code needed to read-in a postfix expression and evaluate it has also been provided to you. You will complete the application by implementing the stack data structure and use it in the postfix evaluation program.

Instructions

1. Implement the four methods in the `unl.cse.stack.Stack` class (which uses the fully implemented `LinkedList` class).

2. You should test your implementation by creating a small `main()` method and push/pop elements off your stack to see if you get the expected behavior.
3. Once your stack is implemented, modify the `evaluateExpression()` method in the `PostfixEvaluator` class as described in the source code. Run the program, complete your worksheet and have a lab instructor check your work.

Queues

Queue data structures, in contrast to Stacks, are a FIFO (First-In First-Out) data structure. Like stacks, they can easily be implemented using a linked list. Queues have numerous applications; one such application is as a data buffer. In many applications, processing a stream of data is expensive. The data stream may be faster than an application can process it. For this reason, incoming data is temporarily stored in a buffer (a queue). The application then reads from the buffer and processes the data independent of the data stream.

In this activity, you will implement and use a queue that acts as a data buffer for a plain text file. The application reads an entire text file and stores lines into your queue for later processing. The processing in this case is a human user that reads the text file page-by-page (by pressing enter). In addition, the text file is not well-formatted for human readers as it contains very long lines. The application processes these lines and displays them page-by-page with a predefined limit on the number of characters per line so that it is more human-user friendly.

It will be your responsibility to complete the implementation of the `Queue` class and to use it properly in the `FileReader` class.

Instructions

1. Implement the four methods in the `unl.cse.queue.Queue` class (which uses the fully implemented `LinkedList` class)
2. As before, you should test your implementation by creating a small `main()` method and enqueue/dequeue elements from your to see if you get the expected behavior.
3. Once your queue is implemented, modify the `FileReader` class as specified in the source code (the `TODO` tasks) to use your queue to process the text file.

JSON Validation

Recall that JSON (JavaScript Object Notation) is a data exchange format that uses key-value pairs and opening/closing curly brackets to indicate sub-objects. A small example:

```
{
  "employees": [
    {
      "firstName":"John",
      "lastName":"Doe"
    },
    {
      "firstName":"Anna",
      "lastName":"Smith"
    },
    {
      "firstName":"Peter",
      "lastName":"Jones"
    }
  ],
  "deptIds": [1, 5, 4, 21],
  "department": "sales",
  "budget": 120000.00
}
```

JSON validation involves determining if a particular string represents a valid JSON formatted string. While not too complicated, full validation does require several complex rules that need to be checked. For this exercise, you will design and implement a JSON validator that will simply check that the opening and closing curly brackets, `{ }` and square brackets `[ ]` are well-balanced. That is, for every opening bracket there is a corresponding closing bracket of the same type. Moreover, brackets of different types must not overlap. For example, `{ [ } ]` would be invalid.

Instructions

1. Open the `unl.cse.stack.JsonValidator` class. Some basic code has been implemented for you that grabs the contents of a specified file
2. Using an appropriate data structure, implement the `validate()` method (you may change its signature if you wish)
3. Test your program on the 5 JSON data files in the `data/` directory and answer

the questions on your worksheet

4. **Extra**: time permitting; modify your program to also check that double-quote characters (used to denote strings) are well balanced. Note that inside a string square and curly brackets need *not* be balanced (and should be ignored). Moreover, double quotes may appear inside of strings as long as they are escaped: `\"`

Submission

We have included a test suite of unit tests written in JUnit (https://junit.org/junit5/) a popular unit testing framework for Java. Even though the test driver (in the `src/test` source folder) has no main method, you can still run it in Eclipse and get a report on how many of the tests passed, failed or resulted in an unexpected exception. Be sure all of the unit tests pass before submitting your source files through webhandin. You can rerun this test suite in the webgrader to ensure everything works.

Advanced Activity (optional)

Queues are often used in a common software pattern called a *Consumer-Producer* pattern. The pattern follows the idea that many independent producer agents (threads, processes, users, etc.) could be generating data or requests that must be processed or handled by independent consumer agents. This pattern is very useful in improving system performance because it facilitates asynchronous behavior–producer and consumer agents can act independently of each other in a multi-threaded environment while requests are stored in a thread-safe queue. Producers enqueue requests that do not require immediate processing and are free to continue in their execution. Consumers dequeue and process requests as they are able to, enabling asynchronous and independent processing.

Consider the following scenario. A web browser posts a form request to a web application which processes the data, writes to a database, updates a log, and sends an email notification to the user and serves a response page back to the user. If this process were synchronous, each action would have to wait for the previous action to terminate before it could begin even if each of the actions were not dependent on each other. The end user would have to wait for all of the actions to terminate, giving a less than ideal user experience. In a multi-user system, this wait time would only compound.

The advanced activity for this lab will be to familiarize you with the c onsumer-producer pattern using Java. We have mocked up a simple `Producer` class which makes an HTTPS connection to an API provided by Twitter (http://www.twitter.com) which serves a continuous stream of a subsample of twitter posts in a JSON (JavaScript Object Notation). The `Producer` streams these tweets (represented as a JSON string)

and places them in a `BlockingQueue<T>`, a thread-safe queue implementation. When threads dequeue elements from the queue if none are available, the queue "blocks" the thread (places it in a sleep status) until an element is available.

We have also designed a `Consumer` class that grabs elements off the queue and process them by parsing the JSON String and creating an instance of a Tweet class. Subsequently, it outputs this tweet to the standard output.

The `PCDemo` creates one `Producer` instance and several Consumer instances (with ?names? for demonstration purposes) and starts each one in their own thread. This is accomplished by making the `Producer` and `Consumer` classes implement the `Runnable` interface (there are alternatives).

Unfortunately, Twitter decided to change its API and require an API key to connect to its firehose. If you modify and update the code to work with their API (possibly using a library), inform the instructor and share your code!