605.202: Introduction to Data Structures

E. A. Calderon

Project 1 Analysis

Due Date: July 6, 2021

Dated Turned In: July 6, 2021

PROJECT 1 ANALYSIS

Lab1 challenge was completed with given parameters seen in (Cost, 2021). There are four main components: user input/output through the file system, algorithm to distinguish whether a line is a palindrome, use of the two-queue based stack Class to hold characters in a LIFO scheme and the use of queue Class within the two-queue based stack Class. The Queue class allows for multiple reuse which is made evident when used in the Queue based Stack Class.

As instructed, the stack was implemented as a queue base. Two queues were implemented in order to mimic the LIFO (q1/q2) stack scheme. The ADT of this can be found along with the submittal. Total of three stacks within main were created to LIFO of the two ends of given text. This was done in order to compare both ends. The Java programming language itself provides a Stack class (java.util.Stack) that was not used for this lab.

The user input and output would take longer than the actual application would because the numbers have to be converted into output text (Door, 2011). The ASCII table was utilized in order to understand the inputs received from the conversion over to letters.

What I Learned

The way we are implementing the queue-based stack is highly inefficient and is not something that would be done in the future, though a better understanding of both the tools (Stack/Queue) was gained. Queue & Stacks have the same time and space complexity but with the given set up, doubles the work performed (Drowell, 2021).

There is a lot of startups overhead in the Java application due to library call outs. It is believed that setting up the structure by making own Queue/Stack Class would be better in certain circumstances/applications. In the case of the lab, one would think the work would be less than the Queue based Stack Class created.

Running through this lab and going through the first five modules of the course have started to open one's eyes of the problem's programmers run into, in terms of work performed, time, space, etc… When starting, one used the character count (which utilized methods, less iterative loops). This saved much of the iterative multiplier of work. Then Stack class with Queue LinkedList LIFO vs. Queue Class FIFO which made the program work more but less with a FIFO parameter to run against. Finally the Queue Based Stack vs. Queue Based Stack. As One went through the iterations one saw how they were causing much more work by the computer.

What I Might do differently Next Time

As seen below in appendix B the programming first makes upper case to lower case. For Java this seemed to work out in case a symbol gets inputted into the system (since it's gotten rid of later in the code). The call out seems to ignore if it is a symbol, don't know if this is done with other programs. Should move after the separation is done in if statement. Capitals could have been included but it might make the algorithm error when evaluating both ends. There is more work being done since the algorithm is evaluating more than it should be rather than just moving it to later.

If given the ability to use the library functions, it would be used for ease of creating more coding and a class structure as seen in appendix A. One thing to note is how the peek/pop removed the math the

programmer was doing in order to compare the characters. As mentioned, another difference would be to compare FIFO to LIFO instead of three created stacks with one initially emptying out to the other.

It should also be pointed out that the use of a recursive algorithm would also assist with the function of the program. See efficiency discussion below. Where it would be essential to add the recursive algorithm would be (1) when breaking out the words (whether even or odd inputs) and/or (2) when reading in file.

1. Breaking out whether even or odd seems like a good point to use recursive. The programmer controls this and it cannot get as complex since it is dividing into two. Simple and it does not seem like the recursive nature will dwindle down to a substantial number of copies.
2. The same could be said for the break up of the texts. Here there would be a catch if the .txt file is empty. There is a concern that the file has a substantial number of lines, yet it will not be as substantial as the number of characters that will need to be broken down.

Justification for Design Decisions

There are two classes in this problem (though as mentioned could be three for ease of reading). The main entry point for the application (Lab1) is used to outline solution algorithm. There is logic within the main in order to properly utilize the class. Where ever items were reusable the programmer did so and either placed in a method or its own class. The main works as follows:

1. Three created stacks
2. Read in txt file
    a. Read in character by character
    b. Convert ASCII values to actual lettering
    c. Take in only letters or numbers
3. Empty dummy stack to another end user stack
4. Use algorithm to push/pop/size/empty/clear
5. Compare the given information against itself for palindrome verification
6. Write out results
7. Perform 1-6 if last line not read

The QStack Class holds both the Queue Based Stack function to work as the LIFO scheme normally seen in stack implementation. The Class feeds from the QueueSelf class in order for the methods to work.

The QueueSelf Class was solely created for the QStack Class to be Queue based.

Issues of Efficiency

As mentioned before the way instructed to structure this project is very inefficient. Where we would use the stack for LIFO for a varied amount of work, we are utilizing the queue-based stack for a work double the amount needed. Since queue is being implemented shifting and renaming with two queue structures needs to be performed. The class being used is using pointers in order to navigate through the queues. It would be interesting to see how long it takes a regular stack structure (whether with libraries or class) compared to the queue-based stack structures, to push all items and pop out all items.

Currently there are a lot of iterative loops being utilized (mostly while loops) that are creating a substantial amount of work when processing if the line is/is not a palindrome. One loop has a complexity of O(n) and when doing a loop within a loop causes O(n^2). Adding more loops will clearly cause more work (I have a total of 2 nested while loops so O(n^3) See Appendix C). Switching to recursive algorithm, depending on the application, might reduce the amount of work the program is performing. Recursion can have different complexities depending on how you go and solve the problem. Things might get way more complex with multiple recursion calls or a bit easier on how you approach the solution of the search.

A way to improve this algorithm's runtime is to come up with a way to get rid of the first while loop of going through the text file until it is empty. As mentioned, before the program was run counting chars in a string. The iteration utilized methods instead of while loops. This would drastically reduce the work performed. Have a queue and stack class on their own right and measure LIFO vs FIFO.

WORKS CITED

Cost, R. (2021, 07 4). *605.202 Data Structures and Algorithms*. Retrieved from Lab1: Use of Stacks: https://blackboard.jhu.edu/bbcswebdav/pid-9909878-dt-content-rid-104617249_2/courses/EN.605.202.81.SU21/Palindromes%26Stacks%281%29.pdf

Door, W. T. (2011). *Project 0 Analysis.* Washington D.C.: John Hopkins.

Drowell, E. (2021, 7 5). *Big-O Cheat Sheet*. Retrieved from Know Thy Complexities!: https://www.bigocheatsheet.com/

APPENDIX

Appendix A

```
else if(Stack.size()%2 != 0)
{
    System.out.println("Odd");
    if(Stack.get((Stack.size()/2)-1) == Stack.get((Stack.size()/2)+1))
    {
        compare = Stack.size();
        while(Stack.size() > compare/2)
        {
            System.out.print(Stack.get() + " ");
            System.out.println(Stack.peek());
            if(Stack.get() != Stack.peek())
            {
                System.out.println("Not a palindrome");
                output.write("\nNot a palindrome \n");
                break;
            }
            else if(Stack.get(j) == Stack.peek())
            {
                Stack.pop();
                Stack.popQ();
                if(Stack.size() == compare/2)
                {
                    System.out.println("Is a palindrome");
                    output.write("\nIs a palindrome \n");
                    break;
                }
            }
            j++;
```

Appendix B

```
//reads each character
while ((c = input.read()) != -1 && c != '\n')
{ // read and process one character

    str = (char) c;
    output.write(str);                      //show user given
    str = Character.toLowerCase(str);       //puts in lower case

    if((str >= 'a' && str <= 'z') ||        //input only letters & #'s
       (str >= '0' && str <= '9'))
    {
        Stack.push(str);                    //pushes stack LIFO
        Stack.pushQ(str);                   //pushes Queue FIFO
        System.out.print(Stack.peek());

    }
```

Appendix C

```
while(input.ready() != false)                    //stops after last
{


    }
    while(Stack2.isEmpty() == false)
    {
        Stack3.push(Stack2.pop());
    }
    //runs as long as "stack" is not empty
    while (Stack.isEmpty() == false)
    {
        System.out.println();
        System.out.println("Size: " + Stack.size());
        output.write("\nSize  (letters/numbers): " + Stack.size());

        //in case they decide to be clever and put one letter
        if(Stack.size() == 1)
        {
            System.out.println("Not a palindrome");
            output.write("\nNot a palindrome \n");
        }
        //seperates even inputs from odds
        else if(Stack.size()%2 == 0)
        {
            System.out.println("Even");
            compare = Stack.size();

            while(Stack.size() > compare/2)
            {
                System.out.print(Stack3.peek() + " ");
                System.out.println(Stack.peek());
                if(Stack3.peek() != Stack.peek())
                {
                    System.out.println("Not a palindrome");
                    output.write("\nNot a palindrome \n");
                    break;
                }
                else if(Stack3.peek() == Stack.peek())
                {
```

O(N^3)