

CS3310

Second Assignment (a1)

Report

Brandon Rodriguez

10-13-17

# Preface

First off, a working, compiled jar of the assignment can be found in the root directory, called “a1.jar”. To run this, open up a terminal, navigate to the directory, and type “java -jar a1.jar”. The rest should be self-explanatory.

To find the source code of the assignment, navigate from the assignment’s root folder to “root/a1/src/edu/wmich/cs3310/a1”. A majority of the code is located within the “DataStructures” subfolder.

Used libraries are included in the “root/a1/lib/” folder.

JavaDocs can be found at “root/Documents/JavaDocs/index.html”, as well as within the project dist folder.

Runtime output logs can be located at “root/Documents/RuntimeOutput/”. The input for this was read directly in from the “balancedParenCheckInputs.txt” file, which is also saved within “root/Documents/”.

Program was created using JetBrains’s “IntelliJ” IDE.

If not familiar with IntelliJ:

Should the jar file not execute, you should be able to load it into IntelliJ and get the same results. When first opening the project, use “Import Project” through IntelliJ’s splash screen. Then select “Create project from existing sources”.

If IntelliJ doesn’t automatically do it for you, make sure to set the “src” directory as the “Sources Root”, and the “tests” directory as the “Test Sources Root”. You may also need to load in JUnit, which should be provided within the lib folder.

You may also need to “Edit Configurations” and set Main as an application to launch from. But after that, it should work. Just run the program with the green arrow button and it should launch.

# Problem Statement

In an attempt to examine various data structures and efficiency of various algorithms, this assignment was created. Specifically, it focuses on linked lists and comparing the stack implementation to the queue implementation.

The node is of “char” type, and the “main functionality” is to read in a string (either from user input or file read in) and determine if the parentheses within said string are properly balanced.

---

## Program Description

Program is overall rather simple. It starts by prompting user to type 1, 2, or 3. Input 1 prompts user to enter system path for file read in. 2 prompts user to enter string to parse. 3 exits program.

If the file option is selected, then the program reads the file and separates it line by line, sending each individual line to be processed accordingly. If the user enters input, then the input is treated as a single line and sent to be processed.

In either case, the given line is “processed” by first being sent to the stack linked list class, then the queue linked list class. They do the appropriate functions to separate the string character by character, save them within the list, and then iterate through all nodes in the list and check for parentheses. Processing times are recorded and output for the user.

Once it finishes, the program resets and the user is once again prompted to type 1, 2, or 3.

The core functionality of this program is in the implementation of the linked lists. Due to habit from previous projects, I initially went with a doubly-linked linked list. This allows both stacks and queues to have a running time of  $O(1)$ , due to saving the head and tail nodes.

However, after implementing time calculation for the project, I realized it was probably implied that we should be using a singly-linked list instead, so that there would be actual variation in times. As such, I added another method to queues that iterates through the list as if the tail node was not saved. However, this method does still properly update the tail node for list consistency and integrity.

On that note, integrity of all lists was mostly enforced via unit testing. I attempted to test all methods using values of  $n$ ,  $n+1$ , and  $n+2$  list size where  $n$  was 0. By inductive reasoning, the list logic should function for all further  $n+1$  list sizes, where  $n$  includes any realistically valid size. When it seemed

appropriate, I even went up to testing size  $n+3$ , so that I could ensure logic was solid even with a middle node present.

# Algorithm Descriptions

The used algorithms were standard implementation of a doubly-linked linked list. Stack pushed and popped at the head node of the list. Queue queued to tail and dequeued from head. Insert and delete were able to do such at any location in the list, but due to project specifications, were not actually used outside of class unit tests.

It was stated in class that we “do not need to show algorithms for commonly used, standard implementations.” A doubly-linked linked list is a fairly standard data structure and I didn’t do anything special for my implementation. As such, I will only show a fairly generalized psuedo-code here, that will roughly apply to insert/push/enqueue.

---

For any such method, it first creates a new node and saves the appropriate data to said node. The method then checks if the list is empty. If so, both the head and tail nodes are set to the new node, and it’s done. Otherwise, it travels to the appropriate location within the list (which varies, depending on the method). The list then updates the new node’s next/previous pointers, as well as the next/previous pointers of the surrounding nodes, if appropriate.

The only real difference between this and the removal methods is that no newNode is created, and the selected node is returned after removal. Pointer updating and location finding still handles roughly identically.

Psuedo-Code:

```
create newNode
if list is empty,
    first and last pointer equal newNode
end method
else
    go to location in list, which is O(1) time for all but insert
    look at node immediately surrounding location
    set these surrounding nodes to newNode’s next/previous pointers
    if surrounding nodes are not null
        also update next/previous pointers of surrounding nodes
    else
        set first/last node appropriately, as it means node is either at start or end of list
if method was insert, return true to indicate success
```

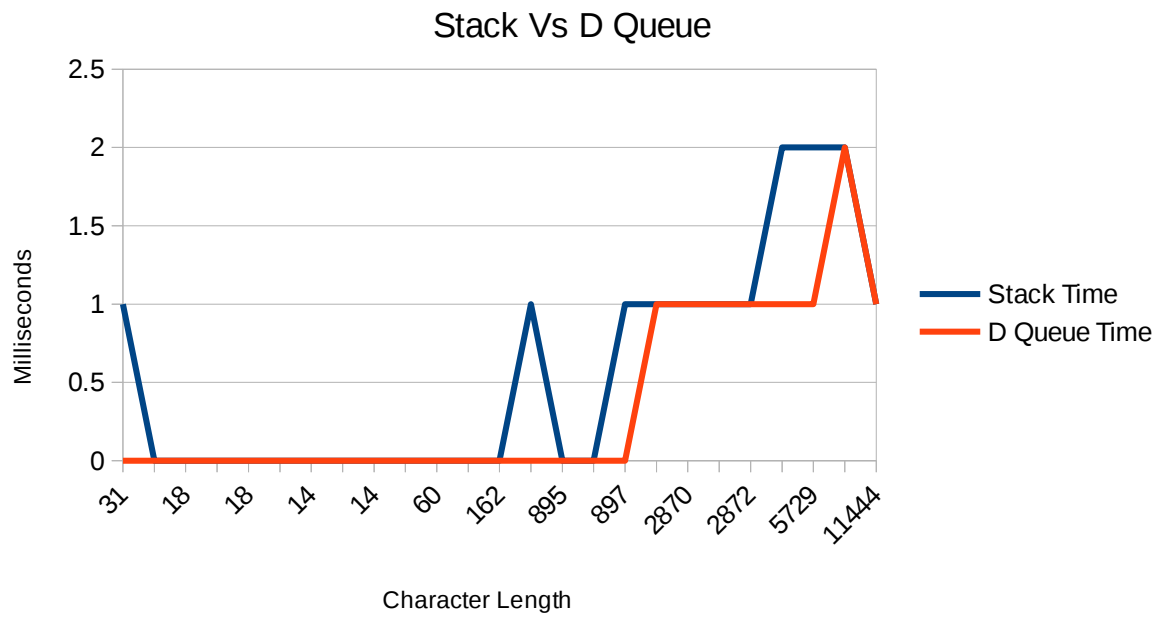
# Tables of Observed Time Complexities

Note: S Queue – Singly-Linked Simulated Queue

D Queue – Doubly-Linked Queue

Processing Time (in milliseconds)

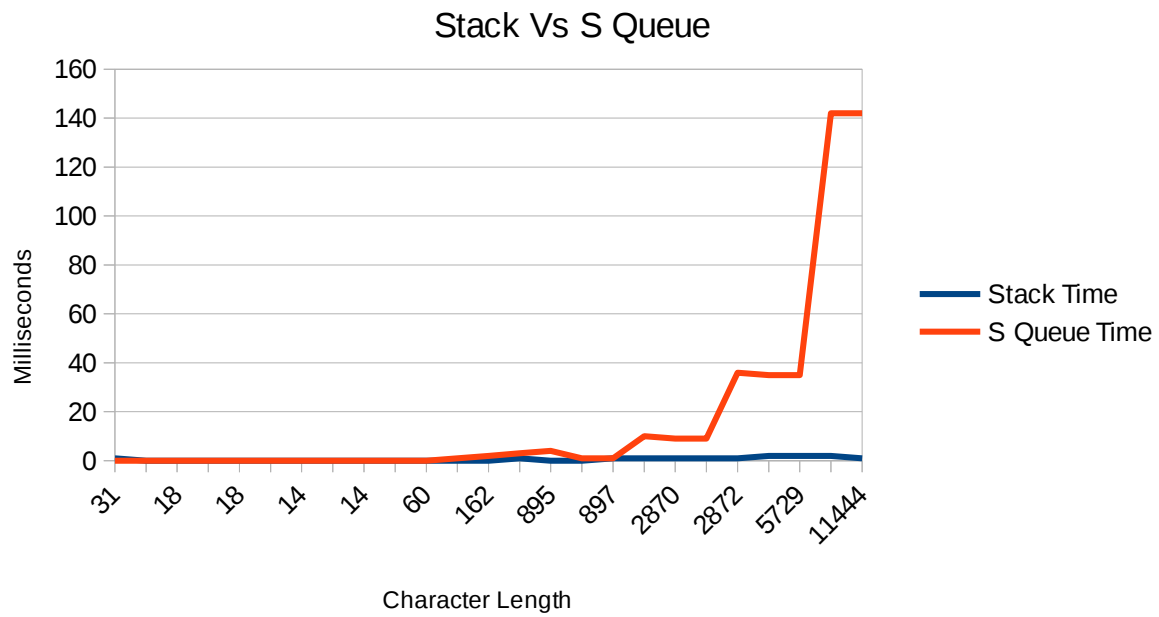
Character Length	Stack Time	D Queue Time	S Queue Time	Total String Time
31	001	000	000	001
18	000	000	000	000
18	000	000	000	000
17	000	000	000	000
18	000	000	000	001
12	000	000	000	000
14	000	000	000	000
14	000	000	000	000
14	000	000	000	000
14	000	000	000	000
60	000	000	000	000
113	000	000	001	001
162	000	000	002	002
215	001	000	003	004
895	000	000	004	004
896	000	000	001	001
897	001	000	001	002
898	001	001	010	012
2870	001	001	009	011
2871	001	001	009	011
2872	001	001	036	039
5728	002	001	035	038
5729	002	001	035	038
11444	002	002	142	146
11444	001	001	142	144



Character Length	Stack Time	D Queue Time
31	001	000
18	000	000
18	000	000
17	000	000
18	000	000
12	000	000
14	000	000
14	000	000
14	000	000
14	000	000
60	000	000
113	000	000
162	000	000
215	001	000
895	000	000
896	000	000
897	001	000
898	001	001
2870	001	001

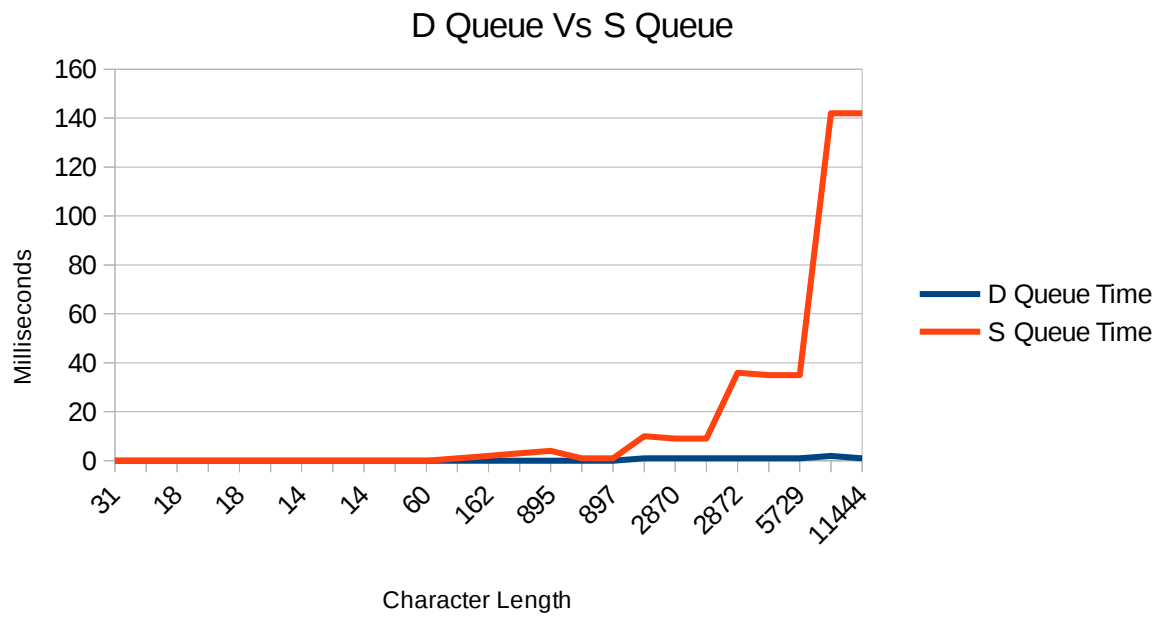
2871	001	001
2872	001	001
5728	002	001
5729	002	001
11444	002	002
11444	001	001





Character Length	Stack Time	S Queue Time
31	001	000
18	000	000
18	000	000
17	000	000
18	000	000
12	000	000
14	000	000
14	000	000
14	000	000
14	000	000
60	000	000
113	000	001
162	000	002
215	001	003
895	000	004
896	000	001
897	001	001
898	001	010
2870	001	009

2871	001	009
2872	001	036
5728	002	035
5729	002	035
11444	002	142
11444	001	142



Character Length	D Queue Time	S Queue Time
31	000	000
18	000	000
18	000	000
17	000	000
18	000	000
12	000	000
14	000	000
14	000	000
14	000	000
14	000	000
60	000	000
113	000	001
162	000	002
215	000	003
895	000	004
896	000	001
897	000	001
898	001	010
2870	001	009

2871	001	009
2872	001	036
5728	001	035
5729	001	035
11444	002	142
11444	001	142

## Concluding Analysis

The data very closely matches what is expected.

---

### Time for Acquiring a Node

A stack's push and pop methods should always have roughly a  $O(1)$  access time, due to only needing the head node.

In a doubly-linked linked list, a queue should always have roughly a  $O(1)$  access time, due to both the head and tail nodes being saved.

In a singly-linked linked list, a queue's dequeue will have roughly a  $O(1)$  access time, due to saving the head node. However, to access the tail node when enqueueing, it needs to iterate through every single node in the list, which significantly increases time to  $O(n)$ .

This means that, when comparing the three with the same action, the singly linked list queue will always have a higher time cost, once the list is more than just a few nodes long.

---

### Time for Processing a Node by Checking Parens

Comparatively, the time to process a node by checking parens is expected to be always trivial to getting the node.

This is because, once a node is acquired, it's essentially just a simple if statement to check what the data value is. The if statement simplifies to time of  $O(1)$  so the time for this can generally be ignored. Acquiring a node will always be greater or equal.

---

## Summary

Due to the above facts, this program is expected to generally run fairly quickly. It still needs to grab each string, add each character to a list, then remove each character to check for parents. So ultimately, it's looking at a minimum of  $O(n)$ , where  $n$  is the number of characters in the string.

However, a singly-linked list queue will also add an additional  $O(n)$  to the time cost, making that by far the most expensive part of the program. Which matches up with the data, as everything else kept at between 0 and 2 milliseconds to process the same data. Meanwhile, the singly-linked list queue was up to 100 times longer to process, when it came to the larger strings.

Ultimately, if the program is only meant to process smaller strings (such as those initially provided in the project description), then it doesn't much matter which version of a linked list is used. However, it looks like, as soon as the strings reach lengths of 10000+ characters, the cost difference starts becoming

noticeable, and would make a rather large impact if it had to handle large volumes of data, such as thousands or millions of records at once.

Even at 5000+ characters, the cost difference arguably starts becoming noticeable, but it still seems small enough that some may consider it insignificant.