605.202: Introduction to Data Structures

Thomas Muamar

Project 1: Palindromes Analysis

Due Date: July 6, 2021

Dated Turned In: July 5, 2021

Project 1: Plaindromes Analysis

Data Structures choice/justification for data structures:

My program contains a linked list implementation of a queue and that queue is then implemented into a stack. In a stack we have the insertions and deletion happening at the same end. While the queue inserts at the rear end and deletes at the front area. So I decided in order to implement the queue into a stack I would have to create two queues in order to put the new elements at the front of the queue. So when we dequeue the item it pops the newly inserted item and produces that last in first out characteristic for that stack. The reason I choose a linked list implementation of a queue is just due to the fact there is no size limitations because a text file for the palindromes could contain any size and having an array would not be as efficient as a linked list for a set of random sized inputs. Since there was no size limitation on pushing items into the queue I did not add a over flow error check just because there wasn't a set size limit for the given list it will keep pushing items until the end of each input line and then read if it's a palindrome or not. The reason stacks would work for this specified problem is because when you pop out the characters from the stack into a string you are popping them out in the reverse order allowing for the ability to compare the words.

So, what occurs in my program is that the push operation will push everything from q1 into q2 which will allow for all the elements that were first inserted to be at the front end of the queue. Then after that I just swap the names of q1 and q2 by setting them equal to each other. Then when I want to pop the characters from the queue it will then dequeue those items in the reverse order from the front end of that queue. Even though this makes the push method a lot more costly for program it was the only way I could think of that could make sure the stack would follow the LIFO characteristic.

In the main program the way of reading the files was done by a buffered reader. The user is asked for the input the file and if the user miss types the input file user is then prompted to enter again and if user does not want to, they can just type exit to leave the program. This allows for the user the use their own input file for what characters they want to check for palindromes. Once user inputs file it will then read the file and ask for an output text file name where answers are stored to weather the info inside the text are palindromes or not. So each time they run the program they can have different output files for there inputs they want to test. I choose to remove all spaces and special characters because it would allow for the best results to weather a statement is a palindrome or not. The reason I say it allows for the best results is because the punctations with in the sentences would end up giving the wrong output of weather that answer was a palindrome. Once those are removed the program goes through a for loop that pushes the first line of the text into a stack. Once each of those words are pushed into a stack one by one then the stack goes through a while loop until it is emptied through the method stack.pop() into the string called reverse. This allowed for me to compare the reverse word to the initial words pushed on to the stack and allowing to see if the results are a palindrome.

What I Learned:

When doing this program I learned that planning ahead of starting doing the program allowed me to easily implement the program a lot easier. Taking the program step by step by checking if my queue worked with a random main method the enqueues and dequeues stuff. Then doing the same with the stack. The main programing was first done by implementing a library to make sure it worked like it should then implementing my own stack code into it to see if it produced the correct output.

Also learned how to debugged stuff a lot more efficiently to allow me to see the main problem with the program and see were the code might have just took a wrong turn.

Learned how to implement a queue into a stack at first I had tried just using one queue to implement the code but couldn't figure out how to make it work correctly and figured it would be easier using to queues to move the characters.

What I might do differently next time:

Something I might do differently next time would be to figure out a way to implement the stack using a single queue instead of having this costly push operation of moving items from one queue to the other. If I could figure out a way to keep the elements at the rear of the queue with out having to push them into another queue that would probably make it more efficient than it is right now.

Another thing I would do differently is just use a normal stack just due to the fact that it would remain O(1) constant instead of having such a costly push in my stack in order to implement the queue with a stack.

Issues of Efficiency:

Implementing a stack by itself would probably be better just because it would more efficient than having a costly push process with the implementation of a queue for the stack. The reason the push is so costly is because I am pushing characters from one queue into to the second one and then having it swap names. When implementing a stack with using two queues the push operation would probably have a time complexity of O(n) in comparison to the constant with just a normal stack of O(1).

Recursive approach vs Iterative approach:

In a recursive approach instead of having the for loop in the main algorithm I would take a base case of having if the words.length is less than 0 then push each word to the stack. then it would subtract words.length by one each time it pushes the stack. Then it will recall itself until it reaches 0 words in for those string of words. The reason a recursive call might not be the best thing for this is because it would most likely produce a worst time complexity than a iterative solution. The reason for that is because we have a set input file that could have any length of words and that would take a lot longer for a recursive solution to do than iterative. Recursion uses up a lot of memory resources unlike iterative process allowing us to use the same variable over and over again. Eventhough recursive solution might be easily understandable the iterative solution provides a faster run time. The reason the recursive solution has such a slow run time is because it deals with the recursive call stack frame.