Candace Holcombe-Volke
Lab 1 Analysis

Lab 1 Analysis

In this analysis, I will discuss the implementation of the language checking program using stacks, alternate implementations including improvements to the existing structure and function of the program, as well as a recursive alternative implementation, and finally lessons learned.

I implemented the stacks used in this program as doubly linked lists. I chose this implementation over using arrays for multiple reasons, including that the length of the input text is always unknown, this program does not need to make use of random access, and I wanted to build the foundation of the stacks in such a way that it would easily support insertions or deletions in the least costly way.

The LinkedStack class inherits methods from the Stack interface. While this program did not find the need to implement array-based stacks, this interface could ultimately be inherited by that stack class as well, if tests became required that could more easily use array-based stacks. The program runs from the main method, which calls readLang() passing the command line arguments (output file, any number of input files). Using the LinkedStack class push() method, readLang() pushes each character into a stack one character at a time, until a new line is found. It then reverses the stack using reverse(), makes copies of the stack for each test that will be performed using copy(), and then prints each item in the stack using the printAll() method. After each stack is printed, the readLang() method calls each successive test, which checks whether the language is valid, pushes the determination into a new stack, and sends the validity stack to the printValidityStmt() method to produce human readable results, by simply testing the values of the validity stack, before finally storing the results to an output file. The stacks are comprised of LSNodes and contain storage for data, head location, and pointers to previous and next nodes.

The program uses one method for each test it performs, and each uses stacks a bit differently, though all stacks are linked list-based. testL1 takes the input and simply distributes each character based on its value, into one stack for As, another for Bs, and a third for anything else, which is invalid in Test 1. Once the input stack is empty (tested for by popping and calling the isEmpty() method), the program pops out the top item from the A stack and the B stack. If both stacks become empty at the same time, and the third stack remains empty, this language is determined to be valid.

Test 2 makes use of a similar technique, however additionally uses the peek() function to test the value of the top item in a stack. Test 2 only allows for As to come before Bs, and using the peek() test for A inputs, when Bs have already been pushed to their stack, allows for a determination of an invalid language. Test 3 operates similarly to Test 2, however manages three stacks of valid characters instead of two. Test 4 adds to the requirements and flexibility of Test 2, by allowing As to appear in the input after Bs. This method operates by initially collecting the first n number of As which are followed by n number of Bs and pushing them into the first stack. Then all other input is pushed into the second stack. Once the input has been collected, the top of both stacks are popped, testing for value equivalence, while both stacks are not empty. If the original set of As and Bs becomes empty before the stack of all the other inputs, or the values don't match, the input is determined to be invalid. If the first stack holding the original As and Bs empties first, the first stack is copied, and continues to be compared with the second stack. This allows the program to produce an accurate answer if the original set of As and Bs appears more than once. While this was a bit challenging, utilizing the peek() method in combination with the isEmpty() method allows the program to determine whether an A or B is appearing for the first time in the string, in which it is pushed onto the first stack, or if it is time to push onto the second stack to be verified against the first. The programs final class, Test6, tests whether each character in the first half of the input string, which is separated by a ‘C’ is the opposite (A versus B) of the character in its position in the second half of the string. (I represent this by $A^{-1}\ C\ B^{-1}$.) This class uses push, pop, and peek heavily in determining the validity of each string.

One alternate implementation for Test 4 could have been to take the entire stack and iterate through from the beginning until the first set of As and Bs had been found, which could be completed by a while loop. Once that was determined, the stack could be divided into segments of the same size as the first stack, and then pop all stacks at the same time and compare values. This would require the upfront division of the stack that may ultimately not be needed (if the very first pop results in unequal values, for example), however removes the requirement to copy the initial stack when the secondary stack is larger.

Additionally, my stacks could have monitored their size, which would make the determination for Test 1 quite simple, without requiring the popping of each element. The size comparison calculation would have cost only O(1), where the popping costs O(n) for each stack, though each push and pop action would require additional cost, and a small amount of additional storage space would be required.

While this program is not built recursively, it could make use of recursion to execute the tests required. Some tests show inclinations toward recursive implementations within their own method. One example of that is Test 5, which checks the validity of a string that begins with AB and is followed by (AB)+$(AB)^2 + \cdots + (AB)^n$, and follows the pattern ABAABBAAABBB. This test lends toward a recursive implementation, as each successive set of As and Bs grows by one. When reversed (into the LIFO stack structure), the stopping case is finding the first character in the stack. From there, the program can build a comparison stack to pop and test the values of the original input. This would work well recursively because it does not require knowing how many AB multiplier sets will be present in the string at the outset, and can adjust as needed.

Additionally, Test 4 seems to lend toward a recursive implementation to find the first set of As and Bs, and compare with the original set, which would become the stopping case, or found after reaching the end of the input. The program can then recursively call a function that would build a test stack to compare the input. Alternatively, a circular singly linked stack with a header node could be used to traverse a comparison stack against the input. Maintaining a reference to the top of the comparison stack would not be necessary, because once it starts, all that matters is that the values popped from the input stack match the value of the comparison stack at the node currently being pointed to. Checking the values (moving the header node around the circular stack) can continue as long as needed. One drawback would be the difficulty in knowing whether the final test completes a full iteration through the comparison stack, for this, another marker pointing to the end would be needed.

Unfortunately, I struggled a lot with the error handling of this program, which led to a lot of unexpected results and a lack of modularization of the components. I had multiple challenges handling the stacks using the pop() and peek() methods, due to the previous nodes being null. This caused a great deal of trouble for me due to my initial pop() design, which initiated the return node with the space character. When popping the last item, I returned spaces rather than null characters, which caused discrepancies with many of my algorithms. Improving the reusability of methods would be one improvement I would make to this program as well. Some ways I could do that include developing separate methods that are recycled through the program, such as pushing out invalid characters, cycling through non-empty stacks, checking equivalence of values, finding the first set of As and Bs in an input, and reading in the input characters. My test validation makes use of complicated while/if/if else loops, and I think the complexity could be reduced. In a future iteration of this program, I would likely develop a set of validations that could be used more universally, for example checking that invalid characters were not present, and then individual tests could call the relevant checks in combinations that can test for a wide range of requirements simply. For example, once determining that the base set of an input is AB, testing for the ABAB pattern in Test 4 could be easily adapted to test for an AABB pattern by using an insert method, instead of the way I implemented which continues to complete these tasks by implementing each step all over, rather than calling a single universal function.

In all, I struggled quite a bit with some of the error handling related to using stacks, but I learned a lot about the nature and usefulness of using a stack data structure. I believe my linked list approach worked well for the tasks that this program performs, and is particularly useful because of its ability to accept and manipulate any size input that is required.