Kevin (Feng Yong) Chen
JHU Email: fchen29@jhu.edu
Student ID: FE4A8B

Lab 1: Analysis

Data Structure and Implementation

Description of data structure and justification

Stacks are the primary data structure used in this lab. The classifications of all of the 6 language in my program were done using only stacks. A stack is a reasonable data structure to use for this assignment because of a few reasons. Firstly, it is a relatively simple data structure which means it is easy to implement and very easy to use. The implementation of it takes very little time and does not require much coding. In addition to that, a stack is also quite intuitive to use. Secondly, a stack allows for an ordered collection of data which means the relative position of a character with respect to its neighbors in a string does not change when the string is pushed inside a stack. Given the LIFO nature of stacks, the order of the characters in a string would be reversed in the stack such that the last character of the string is at the top of the stack and the first character is at the very bottom of the stack. Retaining order of the string is crucial in determining what language(s) a string belongs to because each language has its own format with a particular order of A's and B's that its members must follow. For example, language 2 follows the format A^nB^n. If I used a structure that does not retain the order of characters in the string then after the program puts a string in that data structure, the program cannot determine whether the string is in language 2 because doing so requires the program to know the order of the characters.

Explanation of implementation and justification

I have decided to use a linked implementation for my stacks. The primary reason for choosing this implementation is because of its dynamic allocative nature. I do not know the length of the strings I will be given so for optimal use of space, it is best to use a linked implementation and allocate space as characters are being added to the stack. If I used an array implementation, I would have to allocate a maximum amount of space each time I initiate a stack object to avoid stack overflow. This is highly inefficient as it is likely that not all of the space allocated will be used. Hence, I think a linked implementation is a better choice for this lab. The problem with using a linked implementation is that it requires sequential access to its elements. But that is not a problem for stacks since we only need to access the top element and we do not need to search through the elements in the data structure.

I have implemented 6 methods for my stacks. They are: push, pop, isEmpty, peek, print and copy. Other than push, pop and isEmpty, the other three methods didn't have to be implemented in this lab. However, because stack is such a simple data structure, it was easy for me to implement all 6 methods. In fact, there aren't that much codes in each of the methods. Since stack is a simple data structure, it is easier for programmers to add new features to the structure without having to write much code and yet these features might be very useful in helping the coder solve a particular problem. For example, the print method was useful to me when I was debugging my codes in each module.

Efficiency

Since language 1,2,3 and 6 only consist of iterative loops with constant time operations at each iteration, they all have a runtime complexity of O(N). For example, language 1 has a worst case runtime of :

$k+a(N)+ b(N/2)$

- k represents the runtime of the assignment statements prior to the 2 loops
- a(N) represents the first loop with N being the length of the input string and "a" being the constant time operations for each iteration
- b(N/2) represents the first loop with N being the length of the input string and "b" being the constant time operations for each iteration.

The big O for this equation would then be

$O(k+aN+bN/2)=O[k+(N)[(2a+b)/2]]$
Omitting all constants, the equation becomes O(N).
Language 2 and 3 have similar approaches as language 1 and so they all have a runtime complexity of O(N).

Although language 6 is more complex in terms of its codes, in essence it is just a series of loops, one happening after another. Since the runtime complexity for a loop with constant time operations is O(N), language 6 should still have a runtime complexity of O(N) although in reality its processing time should be longer than language 1, 2 and 3 due to more constant time operations in each loop and more loops are involved in the test method.

Since language 4 and 5 mostly consist of iterative loops, their runtime complexities are also O(N) where N is the length of the input string. But in reality the actual processing time of these 2 language should also be longer than language 1, 2 and 3 due to the fact that in the last loop of both test methods, a stack gets re-filled every time it is empty. This adds additional processing time to the code and the test methods for these two languages should be longer than 1, 2 and 3 even though theoretically all 6 languages have the same runtime complexity.

Appropriateness of Approach

A different approach

All 6 of the languages in this program used an iterative approach to evaluate input strings - all of the methods in the language classes used a combination of for loops and while loops. While this approach may be more intuitive for beginners like myself, the strings could certainly be evaluated using a recursive approach.

For a recursive approach, one could create a method that takes in 2 stacks and compare them. Take language 1 for example, the 2 parameters would be stack A which only contains A and stack B which only contains B. The basic algorithm would look something like what is shown below:

```
For(int i=0; i<string.length();i++)
        If(string.charAt(i)== 'A')
                stackA.push(string.charAt(i))
        else if (string.charAt(i)== 'B')
                stackB.push(string.charAt(i))
        else
                return false

Recursion(stack A, stack B)
If (stackA.isEmpty && stackB.isEmpty). // base case #1
        Return true
Else if (!stackA.isEmpty&&stackB.isEmpty) // base case #2
        Return false
Else if(stackA.isEmpty&&!stackB.isEmpty) // base case #3
        Return false
```

```
Else                    // recursive case
    stackA.pop
    stackB.pop
    return recursion(stackA, stackB)
```

As shown above, there are 3 base cases and 1 recursive case. If neither stacks are empty, the recursion will reduce the size of each stack by one and pass on those two stacks to the next recursive call until one or both stacks are eventually empty. Then the base cases can be used to solve the problem. When both stacks are emptied at the same time, that means stack A and stack B are the same size. In other words, there is an equal number of A and B.

The other 5 languages can be evaluated using recursion in a similar fashion. For example, for language 5 (A^nB^m)^p, one could first prepare a stackA with one repetition of the string (i.e. a sequence of As followed by B's that are from the input string) and a stackB containing the entire sequence from the original string. One could then pass down the 2 stacks to the recursive method. Note that in the stacks, the order of characters in the string will be reversed but since the order in both stackA and stackB are reversed, this will not be a problem. The algorithm will look something like below:

```
Recursion (stackA, stackB)
If(stackA.isEmpty&&stackB.isEmpty)
    Return true
Else if(!stackA.isEmpty&&stackB.isEmpty)
    Return false
Else
    if (stackA.isEmpty &&!stackB.isEmpty)
        Refill stackA so that it has exactly the same content as when it first started
    If(stackA.peek==stackB.peek)
        stackA.pop
        stackB.pop
        return recursion(stackA, stackB)
    else if (stackA.peek!=stackB.peek)
        return false
```

As shown above, the recursive solution to language 5 is similar to language 1 except for a few changes.

Efficiency
Based on the pseudocodes above, a method that uses recursion performs constant time operations within the function and then makes a recursive call. The method should make N recursive calls where N is equal to the size of the shortest stack. Thus, the runtime complexity equals O(1)+f(N-1) = O(N).

A comparison between the iterative approach and recursive approach
Although both approaches have a runtime complexity of O(N), I believe in this assignment the iterative approach is better because it is more efficient. Although the runtime of the iterative approach and the recursive approach are theoretically the same, in reality recursion takes up a lot more computational resources as well as memory due to its extensive use of the call stack. If a recursive method is used to evaluate a string that is exceptionally long, stack overflow may occur because the recursion makes too many recursive calls and the call stack runs out of space. This is less likely to happen with an iterative approach. In short, an iterative approach is better because it is more efficient in terms of space use and other computational resources.

One could also argue that recursion allows coders to write shorter and simpler codes, thus making the problem easier. But since this assignment is not the most difficult to do, I believe there is less of a need to make the codes simpler.

Description and Justification of Design

In this program I created a class for each language and within each language class I created a test method that tests if a string belongs to that language. I decided to use this design because I think it is intuitive which makes it easier for anyone to understand my code. The second reason why I chose to do it this way is because it allows me to develop a language, completely debug it and make sure it is error free before I move on to another language. In essence I am breaking down my codes into modules, making it easier to be debugged so that when there is an error, I can pin point where that error originated.

As mentioned earlier, 6 methods were implemented for the stack data structure although technically only 3 were needed in this lab. Although peek, print and copy were methods that are not required in this lab, these methods were very helpful when I was writing codes as they simplified coding and made it easier for me to implement certain functions. For example, the copy method was useful in the test methods of language 5 and 6. Although I technically did not need to use a copy method, using such method allows me to simplify my code and it made it easier for others to read.

Lessons From This Assignment

1) I have learned how to pass on parameters to a program through command line parameters. Although this may be simple to experienced programmers, as a beginner I have never done this before and so I am glad to finally learn what the "args" parameter in the main method is and how to pass on values to "args".
2) Through analyzing the pros and cons of an iterative approach versus a recursive approach, my understand of each approach is enhanced and I am better able to apply each approach to the right kind of problems. For example, through analysis, I discovered that an iterative approach is usually more efficient than a recursive approach. However, the programmer will likely have to write more codes and the codes may be more complex. Recursion, on the other hand, allows the programmer to make shorter and simpler codes but it may be bad for performance since it takes up more resources.
3) Through this assignment, I have learned how to compile a java file into a class file using command line.
4) My understand of the stack data structure has improved dramatically through this assignment. Although methods such as copy and print were mentioned in lectures, we were not taught how to implement them. So writing these methods gave me more insights on how a stack could be used.

Enhancement

Enhancements made in this assignment include:

1) I implemented the print, peek and copy methods of stack
2) I have added an additional language, language 6 to the program which follows the format (A^nB^2n)^3. The fact that each repetition needs to follow the format A^nB^2n and that there has to be exactly 3 repetitions adds more complexity to my codes compared to language 4 (A^nB^m)^p which allows for any combination of n, m and p.
3) I created an EmptyStackException class for better error handling and prevents stack underflow when performing the pop method of the stack.

Room For Improvement

- The output format could be more user friendly. For example, it could include statistical data in the output file that specify which of the 6 languages has the highest passing rate and which has the lowest passing rate.
- Coding could be more efficient - write shorter, simpler codes.