Problem solving:

My first thought when thinking about how to solve the Palindrome problem would be to use two array stacks to compare against each other. The user would enter in a String of a potential palindrome, the program would then take the string and possibly using the ChaAt() would separate the string into single characters. These characters would then be stored into ArrayStack1 in the order they normally appear and the characters would then be stored in ArrayStack2 in reverse order.

I would then create a compare method that would implement a loop that would peek at the top of both Arraystacks, knowing that ArrayStack1 was instantiated with the string going forward and Arraystack2 was instantiated with the string in reverse order. If both entries at the top of the stack matched then those entries would be popped from the stack and the loop would repeat, if they didn’t match the loop would break, with a statement saying the word was not a palindrome. Otherwise, this process would continue until all items in the stack had been compared and it was in fact a palindrome, or the loop broke because a mismatch had been found.

As you’ll see in my drawing attached to the end of this document, I originally thought about using a switch statement to evaluate the comparison between the ArrayStacks, but quickly abandoned that idea due to the unknown (n) amount of cases that this problem creates.

Conclusion:

I was able to successfully complete the program using the techniques and methods I described above, but when looking over the Big O of the program, especially when trying to have the program process large Strings, I found that it was not efficient enough and I had a lot of unnecessary code. First, I was able to eliminate one Array Stack completely since I was already creating a Char[] to hold the individual characters, I could simply compare that array to one ArrayStack. Second, I was using two For loops in my compare() to complete the comparison, one that checked to see if the arrays matched and another to check to see if they didn’t. I was able to reduce that to one For loop that simply checked to see if they matched, because I could assume otherwise that they didn’t match. Last, since I was no longer using two ArrayStacks for comparison, I was able to eliminate the code I used to fill a stack in reverse order. Since a stack is already filled by adding one item on top of the next that was a completely unnecessary step as the ArrayStack would already be filled in the opposite order compared to how the Char[] was instantiated.

Computational complexity:

The basic operation for my code is comparison due the use of the for loop to make comparisons and instantiate arrays. The .length() determines how many times the comparison takes place since I use that to determine how many times my for loop will execute. Since .length() is unknown as it’s based on the user’s input .length() is essentially = n, and n represents how many characters are in the string that the user inputs.

The computational complexity of my code in terms of Big O is O(n). It is O(n) because even though I use the comparison operation 3 times total in my program they are all separate from each other in different methods, so they are operating independently from each other which does not add to the computational complexity of my code. But since I am using comparisons with the for loop to an unknown degree n, that makes my Big O O(n).

A stack data structure differs from a bag data structure due to order. In a bag there is no order, assignment or direct placement of an object, an object is just simply thrown into a bag, just like how a bag operates in our “real” world. On the other hand, order is very important to a stack data structure. Objects are added in a specific order, not in the sense that they have a direct number assignment, but in the sense that where you place an object in a stack is exactly where you will find it. The first object is located on the bottom of the stack, then the next object on top of it, so on and so on. And if you want to retrieve an object from a stack you have to pop the objects off the top of the stack one at a time until you get to it. In comparison an object is easier to retrieve from a bag, but you still don’t know where exactly that object is at, or what objects are in front or behind it. Where with a stack if you know what order, you placed the objects into the stack you know exactly where it is and using pop() and peek() you can find out what objects were on top or underneath it in the stack.

That’s what made using a stack so attractive of a solution to solving the palindrome problem. As I mentioned above since a stack is already instantiated from the bottom up, when entering the string into the ArrayStack it’s already in the reverse order of how the String would normally appear. So all I had to do was complete a comparison using a for loop where the String starting with its first character was then compared to the top of the stack using the peek(), the top of the stack would then be pop() off and the comparison would continue using the second character in the String and the next character at the top of the stack, again using the peek(). The comparisons continue until all characters in both array types had been evaluated. If every comparison was a match, then the String is considered to be a palindrome, but if there is just one or more non-matches then the String is declared to not be a palindrome.

Diagram

Description automatically generated