1. Stacking:
    a. push(8)
        i. [8]
    b. push(2)
        i. [8,2]
    c. pop()
        i. [8]
    d. push(pop()*2)
        i. [16]
    e. push(10)
        i. [16,10]
    f. push(pop()/2)
        i. [16,5]
2. Queueing:
    a. push(4)
        i. [4]
    b. push(pop()+4)
        i. [8]
    c. push(8)
        i. [8,8]
    d. push(pop()/2)
        i. [8,4]
    e. pop()
        i. [8]
    f. pop()
        i. []
3. Find in deque:

```
public static int findInDeque(Deque<Integer> q, int x) {
int left = 0; // front pointer to search from the back
int right = q.size() - 1; // back pointer to search from the back
Integer[] arr = q.toArray(new Integer[0]); // Convert deque to array for indexing
while (left <= right) { // continues as long left pointer is less than the right pointer
if (arr[left].equals(x)) { return left; }  // Found value from the front
if (arr[right].equals(x)) { return right; } // Found value from the back
left++; // move front pointer forward
```

right--; } // move back pointer backward

return -1; } // Element not found

Logic: I created two pointers to be able to search from the front and the back to reduce searching time. Then, I used an array to for indexing purposes for the while loop and comparisons of the value in the index to target value. The left pointer will continue to increment forward while the right pointer will increment backward until the element is found. If the element is not found, the loop will break when the left is greater than the right pointer.

4. Balanced Brackets code
5. Decode String code:
6. Infix to Postfix code:
7. Algorithm Analysis:
    a. Problem 4:
        i. Time complexity: The time complexity is O(n) where n = s.length() because the operations in the body of the loop are constant time. The O(n) comes from the loop because it iterates through the elements n times.
        ii. Space complexity: The space complexity is O(n) where n is s.length(). The space complexity is O(n) since char c is initialized every time the loop runs.
    b. Problem 5:
        i. Time complexity: The time complexity is O(n + m) where n is s.length() and m is the length of the final output string. The outer loop is O(n) since it iterates through each element. The final string is O(m) since it can grow larger than O(n) depending on the repetitions. The expanded string grows by a total of m characters which means constructing the final string will take O(m) times.
        ii. Space complexity: The space complexity is O(n + m) where n is the size of the input string and m is the size of the expanded decoded string. O(n) represents the input string since elements are stored in the stacks each time the loop iterates through the characters. O(m) is the amount of elements that are added to the final string using Stringbuilder which is stored in currentString. These operations affect storage which is why it is O(n+m).
    c. Problem 6:
        i. Time complexity: The time complexity is O(n) since the main loop iterates through all the characters in the string, where n is expression.length(). This is because each character is processed once

and every operator is pushed and popped at most once, which results in the algorithm running in linear time.

ii. Space complexity: The space complexity is O(n) where n is expression.length(). The stack is used to store elements which can hold up to n elements. The Stringbuilder variable stores up to n elements in the worst-case scenario. O(n) space is required to store the result.