3 (text) Find in deque [10 points] We discussed that using a doubly-linked list, you are able to search an element in O(N/2), the same is true for a deque. Given a Deque q and an element y, provide an algorithm that finds the position in the deque in which element x is stored in O(N/2).

Hint: The i-th element from the right it a position i, whereas the i-th position from the left it a position n-i

```
Public static int SearchDeq(Deque q, Element y){

Int CountDown = q.size();
Int CountUp = 0;
Element front = q.popFront();
Element back = q.popBack();

        while( front != y && back != y){

                Front = q.popFront()
                Back = q.popBack()
                CountDown - -  ;
                CountUp++;

        }

        if( CountUp < CountDown ){
                Return CountUp;
        }
Return CountDown;
}
```

7 (text) Algorithm Analysis [10 points] For each of the algorithms you wrote for problems 4-6, explain their time complexity and space complexity using Big-O notation. Explain how you arrived at your answer.

```java
public static String isBalanced(String s) {
   Stack<Character> st = new Stack<>();
   for (int i = 0; i < s.length(); i++) {
       if (s.charAt(i) == '{' || s.charAt(i) == '(' || s.charAt(i) == '['){
           st.push(s.charAt(i));
       } else if ( s.charAt(i) == '}' && st.peek() != '{' ||
                   s.charAt(i) == ']' && st.peek() != '[' ||
                   s.charAt(i) == ')' && st.peek() != '(' ){
          return "NO";
       }
       else {
           st.pop();
       }
   }
   return "YES";
}
```

Time Complexity : O(N)
Space Complexity : O(N)

The time complexity of isBalanced is O(N) where n is the length of the String s. The for loop iterates from i = 0 and increments by one until i = n. During each iteration of the for loop, the algorithm checks whether the character at position i on the string s is a parenthesis, bracket, or brace, and depending on the condition the algorithm will return, pop, or push from the stack. Checking the condition, pop/push, s.charAt(), and returning are all constant operations. This makes it so that the time complexity is O(N) + each constant operation, which we can simplify to O(N).
The Space complexity is also O(N) where N is the length of String s. In the worst case, s is a string with only open parentheses/brackets/braces. In this case, the algorithm pushes the character into the stack until i is equal to the length of s. Given this case, we allocate a new space in memory for each character pushed onto the stack. This is only a trivial case. In the case where s is a balanced String; s will have an n/2 amount of open parenthesis/brackets/braces. In this case, we are allocating n/2 spaces of memory. The space complexity will then be O(N/2) which can be simplified as O(N).

```java
public static String decodeString(String s) {

   Stack<Integer> num = new Stack<>();
   Stack<String> str = new Stack<>();
   int integer = 0;
   StringBuilder res = new StringBuilder();

   for (int i = 0; i < s.length(); i++) {
       if (s.charAt(i) >= '0' && s.charAt(i) <= '9') {
           integer = integer * 10 + (s.charAt(i) - '0');
       } else if (s.charAt(i) == '[') {
           num.push(integer);
           str.push(res.toString());
           res = new StringBuilder();
           integer = 0;
       } else if (s.charAt(i) == ']') {
           StringBuilder buf = new StringBuilder(str.pop());
           int count = 0;
           int j = num.pop();
           while (count < j) {
               buf.append(res);
               count++;
           }
           res = buf;
       } else {
           res.append(s.charAt(i));
       }
   }
   return res.toString();
}
```

Time Complexity : O(N)
Space Complexity : O(N)

The time complexity is O(N) where N is the length of the String s. The for loop in this algorithm initiates int i to 0 and with each iteration i increments by one until it is equal to the length of s. With each iteration of the loop, the algorithm checks whether the character at position i on the string s is an integer or a bracket. Each condition in the loop will either push, pop, perform an operation, and assign values which means they are all constant. The exception is when there is a closing bracket. When a closing bracket is found another loop is run and in each iteration of that loop count starts from 0 and is incremented by one until it is equal to int j. Each iteration also appends a string to buf and is a constant operation. Since s is expected to be a valid input, there is no case where the inner loop will run for every character in s. The expression will be O(N *AM) where A is a constant less than N and represents the number of closing braces. M is the input for the inner loop and it is given the constraint of being in between 1 and 300. Because A is some

constant less than N, it makes AM the slowest-growing term therefore we can remove that term giving us O(N).
The Space Complexity is O(N) where N is the size of the String s. This is because in the case where there is a closing bracket, the algorithm allocates a new place in memory for buf. Although this would not happen on every iteration of the outer loop, it is guaranteed to happen for a valid input. So as the size of the String increases, the amount of time memory allocated for buf would also increase linearly.

```
public static String infixToPostFix(String infix){
    Stack<Character> op = new Stack<>();
    String postFix = "";
    char c;
    int prec;
    for(int i = 0; i < infix.length(); i++){
        c = infix.charAt(i);
        prec = gibVal(c);
      if (c >= 'a' && c <= 'z' || (c >= '0' && c <= '9')) {
          postFix += c;
      } else if (c == ')') {
            while(op.peek() != '('){
                postFix += op.pop();
            }
            op.pop();
      } else if (c == '(') {
          op.push(c);
      } else if (op.empty()) {
          op.push(c);
      } else if ( prec > gibVal(op.peek()) ) {
           op.push(c);
      } else if ( prec <= gibVal(op.peek()) ) {
              while (!op.empty() && gibVal(op.peek()) >= gibVal(c)) {
                  if(op.peek() == '('){
                      break;
                  }
                  postFix += op.pop();
              }
              op.push(c);
      }
    }
   while(!op.empty()){
       postFix += op.pop();
   }
    return postFix;
}

public static int gibVal(char op){

    if(op == '*' || op == '/' ) {
        return 1;
    }
    else if(op == '^'){
        return 2;
    }
    else if(op == '+' || op == '-'){
        return 0;
    }
    return -1;
    }

}
```

Time Complexity : O(N)
Space Complexity : O(N)

The method GibVal has a constant time complexity because no matter the size of the char op, the amount of times the code runs remains the same. It also has a space complexity that is constant because the memory allocated also remains the same based on the input.

The time complexity is O(N) where N is the size of the infix string. Through each character iteration in the infix string, the algorithm checks five conditions and four of them are constant. Two of the conditions have loops but they are not guaranteed to occur through each iteration of the outer loop, therefore the first loop has a time complexity of O(N). We also need to consider the second while loop that occurs after the first loop. This loop pops from the stack until it is empty and the size of the stack will never exceed the size of the infix string. This makes it so that it is a slow-growing term and we only need to worry about the first loop.

The space complexity is O(N) where N is the size of the infix string. In every iteration of the first loop, the algorithm either pushes a value onto the stack or concatenates the string postFix. Pushing a value into the stack requires allocating memory for that value. That means the algorithm creates more space based on the size of the infix. Also, to concatenate a string, a new string is created which means more memory is allocated. At every iteration of the first loop and second loops, the algorithm allocates more memory for the string postfix based on the size of the infix. This means that at every iteration we are allocating a linear amount of space based on the size of the infix.

```
Public static int SearchDeq(Deque q, Element y){

Int CountDown = q.size();
Int CountUp = 0;
Element front = q.popFront();
Element back = q.popBack();

        while( front != y && back != y){

                Front = q.popFront()
                Back = q.popBack()
                CountDown - -  ;
                CountUp++;

        }

        if( CountUp < CountDown ){
                Return CountUp;
        }
Return CountDown;
}
```

Time Complexity : O(N)
Space Complexity: O(1)

Time complexity is O(N) where N is the size of the deque. In the worst case, the element y will be in the middle of the deque. The algorithm pops the front and back of the deque and checks if either of the elements is y. If not, it does it again until element y is found. This is a constant time operation and the loop will run at most n/2 times. This means the Big O is O(N/2) and we can remove the 1/2 because it is a constant factor term.

Space complexity is constant because the size of the deque does not change the amount of memory that is allocated in the method.