1. When am I Useful Senpai?

Based on the description, choose the data structure which would best suit our purposes. Choose from: A - arrays, B - linkedlists, C - stacks, D - queues (excluding dequeue’s cause they’re too OP).

1. Keeping track of which customer in a line came first.

2. We will expect many inserts and deletes on some dataset, but not too many searches and lookups.

3. We gather a lot of data of a fixed length that will remain relatively unchanged overtime, but we access its contents very frequently.

4. Maintaining a history of the last actions on Word in case I need to undo something.
2 Reverse Me

Assume that we have a `MyIntQueue` class with API:

boolean isEmpty() // returns true if the queue is empty  
void enqueue(int item) // adds item to the back of the queue  
int dequeue() // removes the item at the front of the queue  
int peek() // returns but doesn’t remove the item at the front of the queue  
int size() // returns the size of the queue

We also have a `Stack` API as follows:

boolean isEmpty() // returns true if the stack is empty  
void push(int item) // adds item to the top of the stack  
int pop() // removes the item at the top of the stack  
int peek() // returns but doesn’t remove the item at the top of the stack  
int size() // returns the size of the stack

Fill in the method below that takes in a `MyIntQueue q`, and reverses its elements using a `Stack`.

```java
private static void reverse(MyIntQueue q) {
    Stack s = new Stack();
    while (________________________) {

    }

    while (________________________) {

    }
}
```
3. Pseudo Stack

Implement a stack’s pop and push methods using two Queues. We have the same MyIntQueue API as in the previous question.

```java
class MyIntStack {
    MyIntQueue q1 = new MyIntQueue();
    MyIntQueue q2 = new MyIntQueue();

    public boolean isEmpty() {
        //Implementation not shown
    }
    public int size() {
        //Implementation not shown
    }
    public void push(int item) {
    }

    public int pop() {
    }
}
```
4 A Balancing Act

Given a string str, containing just the characters (), {}, [], implement a method hasValidParens which determines if the string is valid.

The brackets must close in the correct order so "()", "{}", and "[]" are all valid, but "()", "{}", and "[]" are not.

You may refer to the Stack API from problem 2 (but apply for chars) and use the getRightParen method provided below.

```java
private static boolean hasValidParens(String str) {
    Stack s = new Stack();
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (c == '(') s.push(c);
        else if (c == ')') {
            if (s.isEmpty()) return false;
            s.pop();
        } else if (c == '{') s.push(c);
        else if (c == '}') {
            if (s.isEmpty()) return false;
            s.pop();
        } else if (c == '[') s.push(c);
        else if (c == ']') {
            if (s.isEmpty()) return false;
            s.pop();
        } else {
            //not one of the valid parenthesis characters
            throw new IllegalArgumentException();
        }
    }
    return s.isEmpty();
}
```

/**
 * The method getRightParen takes in the left parenthesis
 * and returns the corresponding right parenthesis.
 **/
private static char getRightParen(char leftParen) {
    if (leftParen == '(') return ')';
    else if (leftParen == '{') return '}'
    else if (leftParen == '[') return ']
    else {
        //not one of the valid parenthesis characters
        throw new IllegalArgumentException();
    }
}
```