Problem #1: Use the LinkedList class in the Java API to write a program that does the following. Add (i.e., “insert”) the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 in that order. Then swap the elements at the third and fourth positions.

Hint: In the Java API, LinkedList has a funny symbol <E> after it. This just means that we are allowed to create a list out of any Object we like. All we have to do is replace the E with the kind of Object we desire. For example, if I want to create a list of strings, I would say

```java
LinkedList<String> myList = new LinkedList<String>();
```

Now, anytime I am using “myList” and I see an E in the LinkedList API, I’ll know that it is really a String. Similarly, if I say

```java
LinkedList<Integer> myOtherList = new LinkedList<Integer>();
```

then any time I am using “myOtherList” and see an E in the LinkedList API, then it actually will be an Integer. So if a method in the API returns E, I’ll know that the method really returns an Integer. Cool! This tailors the LinkedList so that it is a collection of whatever particular stuff you need!

But watch out. The <E> will only take Objects (“complex types”), and not basic types like int, boolean, char, etc. In other words, if I want a list of integers, I cannot use “LinkedList<int>”. Instead, to add an int to the list, I will first convert it to an object by saying “Integer i = new Integer(12);” Now the variable “i” is an object of type Integer. For example,

```java
LinkedList<Integer> myOtherList = new LinkedList<Integer>();
Integer i = new Integer(12);
// And now you read the API to figure out how to add “i” to the list.
```

Problem #2: Use the LinkedList class in the Java API to write a program that does the following. Add the numbers 12, 4, 67, 9, and -2 in that order. Now add 10000 additional random numbers at the end of the list. Now remove the second element in the list. Now print the first six elements.
Problem #3: Write your own code for the array implementation of the Stack ADT. Assume that you only need the following operations: push, pop, and top.

         Hint: The operations are methods in Java. See the handouts I gave you for general code guidance. Although they do not solve this problem, the handouts show nice code outlines. You are permitted to copy any of this code if it helps you. Also, be sure to test your code! i.e., write a driver class that tries each of the methods over and over to make sure they work. Watch those “corner cases” – for example, what happens if you try to pop from an empty stack? Or what if you push more elements than the size of your array?

Problem #4: Give the big-O run times of your push and pop methods for the Array Stack. Briefly explain what parts of your code create these growth rates (e.g., “This is, like, way slow dude ’cause my gnarly sixteen nested while loops take all the time and make it O(N^16).”