Problem Set 5

Stack, Array List, Queue

---

1. Suppose that the Stack class consisted only of the three methods push, pop, and isEmpty:

```
public class Stack<T> {
  ...
  public Stack() { ... }
  public void push(T item) { ... }
  public T pop() throws NoSuchElementException { ... }
  public boolean isEmpty() { ... }
}
```

   Implement the following "client" method (i.e. *not* in the Stack class, but in the program that uses a stack):

```
public static <T> int size(Stack<T> S) {
   // COMPLETE THIS METHOD
}
```

   to return the number of items in a given stack S.

   Derive the worst case big *O* running time of the algorithm you used in your implementation. What are the dominant algorithmic operations you are counting towards the running time?

---

2. A postfix expression is an arithmetic expression in which the operator comes *after* the values (operands) on which it is applied. Here are some examples of expressions in their regular (infix) form, and their postfix equivalents:

Infix	Postfix
2	2
2 + 3	2 3 +
2 * (3 + 4)	2 3 4 + *
2 * (3 - 4) / 5	2 3 4 - * 5 /

   Note that the postfix form does not ever need parentheses.

   Implement a method to evaluate a postfix expression. The expression is a string which contains either single-digit numbers (0--9), or the operators +, -, *, and /, and nothing else. There is exactly one space between every two characters. The string has no leading spaces and no trailing spaces. You may assume that the input expression is not empty, and is correctly formatted as above.

   You may find the following Stack class to be useful - assume the constuctor and methods are already implemented.

```
public class Stack<T> {
  public Stack() { ... }
  public push(T item) { ... }
  public T pop() throws NoSuchElementException { ... }
  public T peek() throws NoSuchElementException { ... }
  public boolean isEmpty() { ... }
  public void clear(T item) { ... }
```

```
    public int size (T item) { ... }
  }

}
```

You may use the Character.digit(char,10) method to convert a character to the integer value it represents. For example, Character('2',10) returns the integer 2. (The parameter 10 stands for the "radix" or base of the decimal number system.)

You may write helper methods (with full implementation) as necessary. You may not call any method that you have not implemented yourself.

```
public static float postfixEvaluate(String expr) {
  /** COMPLETE THIS METHOD **/
}
```

---

3. This question compares the space usage for two versions of a stack, one using a linked list in which each node holds a reference to an object and a pointer to the next node, and the other using the Java ArrayList (array cells holding references to objects). Suppose the stack holds 1000 objects at its peak usage. How many bytes of space are used (a) by the linked list implementation, and (b) by the ArrayList implementation, at peak usage? Use the following data:
   - A reference/pointer to an object uses 4 bytes of space.
   - The ArrayList starts with an initial capacity of 10, and doubles each time it is resized.
   - The linked list implementation keeps a "front" reference/pointer to the first node
   - Both implementations keep an integer "size" field (4 bytes)
   - The ArrayList implementation keeps an integer capacity field (4 bytes)

---

4. **WORK OUT THE SOLUTION TO THIS PROBLEM ON PAPER, AND TURN IT IN AT RECITATION**

   Consider a smart array that automatically expands on demand. (Like the java.util.ArrayList.) It starts with some given initial capacity of 100, and whenever it expands, it **adds 50 to the current capacity**. So, for example, at the 101st add, it expands to a capacity of 150.

   How many total units of work would be needed to add 1000 items to this smart array? Assume it takes one unit of work to write an item into an array location, and one unit of work to allocate a new array.

---

5. Suppose you set up a smart array with an initial capacity of 5, with a *DOUBLING* of capacity every time there is a resize. What would be the **average** number of units of work per add, in the course of performing **100** adds? Assume the same work units as the previous exercise.

---

6. You are given the following Queue class:

```
public class Queue<T> {
  public Queue() { ... }
  public void enqueue(T item) { ... }
  public T dequeue() throws NoSuchElementException { ... }
  public boolean isEmpty() { ... }
  public int size() { ... }
}
```

Complete the following *client* method (*not* a Queue class method) to implement the peek feature, using only the methods defined in the Queue class:

```
// returns the item at the front of the given queue, without
// removing it from the queue
```

```
public static <T> T peek(Queue<T> q)
throws NoSuchElementException {
  /** COMPLETE THIS METHOD **/
}
```

Derive the worst case big *O* running time of the algorithm that drives your algorithm. What are the dominant algorithmic operations you are counting towards the running time?

---

7. * Suppose there is a long line of people at a check-out counter in a store. A new counter is opened, and people in the even positions (second, fourth, sixth, etc.) in the original line are directed to the new line. If a check-out counter line is modeled using a Queue class, we can implement this "even split" operation in this class.

   Assume that a Queue class is implemented using a CLL, with a rear field that refers to the last node in the queue CLL, and that the Queue class already contains the following constructors and methods:

```
public class Queue<T> {
  public Queue() { rear = null; }
  public void enqueue(T obj) { ... }
  public T dequeue() throws NoSuchElementException { ... }
  public boolean isEmpty() { ... }
  public int size() { ... }
}
```

   Implement an additional method in this class that would perform the even split:

```
// extract the even position items from this queue into
// the result queue, and delete them from this queue
public Queue<T> evenSplit() {
  /** COMPLETE THIS METHOD **/
}
```

   Derive the worst case big *O* running time of the algorithm that drives your algorithm. What are the dominant algorithmic operations you are counting towards the running time?