CSE 373 Section Handout #1

`ArrayIntList` class

(stores a list of integer elements using an internal array and size field)

```
public class ArrayIntList {
    private int[] elementData;
    private int size;

    public ArrayIntList() { ... }
    public ArrayIntList(int capacity) { ... }

    public void add(int value) { ... }
    public void add(int index, int value) { ... }
    public boolean contains(int value) { ... }
    public void ensureCapacity(int capacity) { ... }
    public int get(int index) { ... }
    public int indexOf(int value) { ... }
    public boolean isEmpty() { ... }
    public void remove(int index) { ... }
    public void set(int index, int value) { ... }
    public int size() { ... }
    public String toString() { ... }

    // your methods
}
```

Above is an example of the **List ADT** being implemented to store integers only. **Practice your Java skills and being the implementer of a data structure** by adding the methods described below to the `ArrayIntList` class. Unless otherwise noted, assume that you may call any other methods of the class to help you solve the problem.

1. Write a method `maxCount` that returns the number of occurrences of the most frequently occurring value in a sorted list of integers. Because the list will be sorted, all duplicates will be grouped together, which will make it easier to count duplicates. For example, suppose that a variable called `list` stores the following sequence of values:

   `[1, 3, 4, 7, 7, 7, 7, 9, 9, 11, 13, 14, 14, 14, 16, 16, 18, 19, 19, 19]`

   This list has some values that occur just once (1, 3, 4, 11, 13, 18), some values that occur twice (9, 16), some values that occur three times (14, 19) and a single value that occurs four times (7). Therefore, the call of `list.maxCount()` should return `4` to indicate that the most frequently occurring value occurs 4 times. It is possible that there will be a tie for the most frequently occurring value, but that doesn't affect the outcome because you are just returning the count, not the value. For example, if there are no duplicates in the list, then every value will occur exactly once and the maximum would be 1. If the list is empty, your method should return `0`.

2. Write a method `longestSortedSequence` that returns the length of the longest sorted sequence within a list of integers. For example, if a variable called `list` stores the following sequence of values:

   `[1, 3, 5, 2, 9, 7, -3, 0, 42, 308, 17]`

   then the call of `list.longestSortedSequence()` would return the value 4 because it is the length of the longest sorted sequence within this list (the sequence -3, 0, 42, 308). If the list is empty, your method should return 0. Notice that for a non-empty list the method will always return a value of at least 1 because any individual element constitutes a sorted sequence.

CSE 373 Section Handout #1

3. Write a method `runningTotal` that returns a new `ArrayIntList` that contains a running total of the original list. In other words, the i^{th} value in the new list should store the sum of elements 0 through i of the original list. For example, suppose a variable `list` stores the following sequence of values:

   ```
   [2, 3, 5, 4, 7, 15, 20, 7]
   ```

   If the following call is made:

   ```
   ArrayIntList list2 = list.runningTotal();
   ```

   Then the variable `list2` should store the following sequence of values:

   ```
   [2, 5, 10, 14, 21, 36, 56, 63]
   ```

   The original list should not be changed by the call. If the original list is empty, the result should be empty.

4. Write a method `isPairwiseSorted` that returns whether or not a list of integers is pairwise sorted (`true` if it is, `false` otherwise). A list is considered pairwise sorted if each successive pair of numbers is in sorted (non-decreasing) order. For example, suppose a variable called `list` stores the following values:

   ```
   [3, 8, 2, 5, 19, 24, -3, 0, 4, 4, 8, 205, 42]
   ```

   The call of `list.isPairwiseSorted()` should return `true` because the successive pairs of this list are all sorted: (3, 8), (2, 5), (19, 24), (-3, 0), (4, 4), (8, 205). Notice that the extra value 42 at the end had no effect on the result because it is not part of a pair. If the list had instead stored the following:

   ```
   [1, 9, 3, 17, 4, 28, -5, -3, 0, 42, 308, 409, 19, 17, 2, 4]
   ```

   Then the method should return `false` because the pair (19, 17) is not in sorted order. If a list is so short that it has no pairs, then it is considered to be pairwise sorted.

For the following problems, **practice being the client of a data structure**. Assume that you are using the Java.util `Stack` class, `Queue` interface, and `LinkedList` implementations.

5. Write a method `copyStack` that takes a stack of integers as a parameter and returns a copy of the original stack (i.e., a new stack with the same values as the original, stored in the same order as the original). Your method should create the new stack and fill it up with the same values that are stored in the original stack. You will be removing values from the original stack to make the copy, but you have to be sure to put them back into the original stack in the same order before you are done. In other words, when your method is done executing, the original stack must be restored to its original state and you will return the new independent stack that is in the same state. You may use one queue as auxiliary storage.

6. Write a method `equals` that takes as parameters two stacks of integers and returns `true` if the two stacks are equal and that returns `false` otherwise. To be considered equal, the two stacks must store the same sequence of integer values in the same order. Your method will need to manipulate the two stacks, but must return them to their original state before terminating. You may use one stack as auxiliary storage.

CSE 373 Section Handout #1

7. Write a method `rearrange` that takes a queue of integers as a parameter and rearranges the order of the values so that all of the even values appear before the odd values and that otherwise preserves the original order of the list. For example, suppose a queue called `q` stores this sequence of values:

   ```
   front [3, 5, 4, 17, 6, 83, 1, 84, 16, 37] back
   ```

   Then the call of `rearrange(q);` should rearrange the queue to store the following sequence of values:

   ```
   front [4, 6, 84, 16, 3, 5, 17, 83, 1, 37] back
   ```

   Notice that all of the evens appear at the front of the queue followed by the odds and that the order of the evens is the same as in the original list and the order of the odds is the same as in the original list. You may use one stack as auxiliary storage.

8. Write a method `isPalindrome` that takes a queue of integers as a parameter and returns `true` if the numbers in the queue represent a palindrome (and `false` otherwise). A sequence of numbers is considered a palindrome if it is the same in reverse order. For example, suppose a queue called `q` stores these values:

   ```
   front [3, 8, 17, 9, 17, 8, 3] back
   ```

   Then the call of `isPalindrome(q)` should return `true` because this sequence is the same in reverse order. If the queue had instead stored these values:

   ```
   front [3, 8, 17, 9, 4, 17, 8, 3] back
   ```

   The call on `isPalindrome(q)` would instead return `false` because this sequence is not the same in reverse order (the 9 and 4 in the middle don't match). The empty queue should be considered a palindrome. You may not make any assumptions about how many elements are in the queue and your method must restore the queue so that it stores the same sequence of values after the call as it did before. You may use one stack as auxiliary storage.

9. Write a method `shift` that takes a stack of integers and an integer *n* as parameters and that shifts *n* values from the bottom of the stack to the top of the stack. For example, if a variable called `s` stores the following sequence of values:

   ```
   bottom [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] top
   ```

   If we make the call `shift(s, 6);` the method shifts the six values at the bottom of the stack to the top of the stack and leaves the other values in the same order producing:

   ```
   bottom [7, 8, 9, 10, 6, 5, 4, 3, 2, 1] top
   ```

   Notice that the value that was at the bottom of the stack is now at the top, the value that was second from the bottom is now second from the top, the value that was third from the bottom is now third from the top, and so on, and that the four values not involved in the shift are now at the bottom of the stack in their original order. If s had stored these values instead:

   ```
   bottom [7, 23, -7, 0, 22, -8, 4, 5] top
   ```

   If we make the following call: `shift(s, 3);` then s should store these values after the call:

   ```
   bottom [0, 22, -8, 4, 5, -7, 23, 7] top
   ```

   You are to use one queue as auxiliary storage to solve this problem. You may assume that the parameter n is ≥ 0 and not larger than the number of elements in the stack.