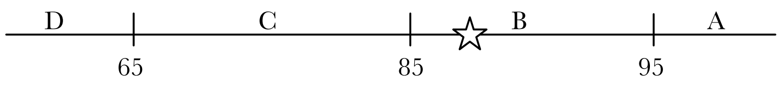
Linear Structures

1. Exam Data
   a. Mean: 87/100. Max = 101/100
   b. Interpreting your grade:



2. Most common misunderstandings:
   a. It is legal to **cast** any expression to any types in Java. A runtime error will occur if the actual objects do not have the right type.
   b. The **lower bound** is the shortest amount of time (fastest) an algorithm could possibly run. O(1) is a lower bound on everything, but it is not a very tight bound (that's like saying the algorithm is infinitely fast, regardless of the size of the input).
   c. The **best and worst** case bounds occur based on the actual data. Whether you are in the best or worst case must be independent of the bound variables (e.g., *n*). Bounds must hold for some arbitrarily large *n*. The best case cannot be "when $n = 1$"; the point is that for some large *n*, what input data of that size would create good luck and let the algorithm terminate as soon as possible.
   d. For an **inductive proof** you must:
      i. Explain why the base case should have some value
      ii. Demonstrate that the base case does have that value
      iii. Assume that the theorem holds for n-1
      iv. Explain why the incremental change from n to n-1 should have some value
      v. Demonstrate that the incremental change does in fact have that value
   e. Any iterative method can trivially be rewritten as recursive by rearranging the parts of the FOR loop into the arguments to a recursive call
3. Be lazy

4. Stack (all methods O(1))

```
public interface Stack<E> {
    public void push(E v);          // addLast
    public E pop();                 // removeLast
    public boolean isEmpty();
    public int size();
}
```

5. Tail recursion

6. Example of a non-tail recursive method made iterative:

Recursive

```
public static void quickSort(int[ ] data) {



   quickSort(data, 0, data.length - 1);
}


/** Sorts elements from L to R, inclusive */
private static void quickSort(int[ ] data, int L, int R) {

   if (L < R) {
      int p = partition(data, L, R);


      quickSort(data, L, p - 1);
      quickSort(data, p + 1, R);
   }
}
```

Iterative with Explicit Stack

```
public static void quickSort2(int[ ] data) {
   Stack<Bounds> stack =
            new ArrayStack<Bounds>();

   stack.push(new Bounds(0, data.length - 1));

   while (! stack.isEmpty()) {
      // Get the next piece of work to do
      Bounds bounds = stack.pop();
      int L = bounds.L, R = bounds.R;

      if (L < R) {
         int p = partition(data, L, R);

         // Push the two sub-arrays onto the stack
         stack.push(new Bounds(L, p - 1));
         stack.push(new Bounds(p + 1, R));
      }
   }
}
```

7. Queue (all O(1))

```
public interface Queue<E> {
      public void pushLast(E v);          // addLast
      public E popFirst();                // removeFirst
      public int size();
      public boolean isEmpty();
}
```

8. Dequeue (all O(1))

```
public interface Dequeue<E> {
      public void pushFirst(E v);         // addFirst
      public void pushLast(E v);          // addLast
      public E popFirst();                // removeFirst
      public E popLast();                 // removeLast
      public int size();
      public boolean isEmpty();
}
```

How would you implement this using an array?