Short Answers

P1. (8pts, 1pt each) Q1 on page 284

1. True or False? Explain your answers.
   a. A queue is a "first in, first out" structure.
   b. The element that has been in a queue the longest is at the "rear" of the queue.
   c. If you `enqueue` five elements into an empty queue and then `dequeue` five elements, the queue will be empty again.
   d. If you `enqueue` five elements into an empty queue and then perform the `isEmpty` operation five times, the queue will be empty again.
   e. The `enqueue` operation should be classified as a "transformer."
   f. The `isEmpty` operation should be classified as a "transformer."
   g. The `dequeue` operation should be classified as an "observer."
   h. If we first `enqueue elementA` into an empty queue and then `enqueue elementB`, the `front` of the queue is `elementA`.

a)True
b)False
c)True
d)False
e)True
f)False
g)False
h)True

P2. (8pts, 2pt each) Q16 on page 340

16. An `equals` method is supposed to provide an equivalence relation among the objects of a class. This means that if a, b, and c are non-null objects of the class, then
    i. `a.equals(a)` is true.
    ii. `a.equals(b)` has the same value as `b.equals(a)`.
    iii. If `a.equals(b)` is true and `b.equals(c)` is true, then `a.equals(c)` is true.

    State whether the following definitions of `equals` are valid. If they are not, explain why not.
    a. Two circles are equal if they have the same area.
    b. Two circles are equal if their radii are within 10% of each other.
    c. Two integers are equal if they have the same remainder when divided by a specific integer, for example, when divided by 3.
    d. Two integers are equal if the second integer is a multiple of the first.

a) **False These circles may have the same area but not necessarily the same radius/shape and therefore are different object and will be false through the equals method**
b) **False These areas will be different unless they have the same radius and so they will be different objects**
c) **False This is false since 15/3 has a remainder of 0, and is different than 21/3 which also has a remainder of 0**
d) **False This is false in most cases since 12 is a multiple of 36 but don't equal each other**

P3. (10pts, 2pt each) Q12 on page 413

12. Describe the "special cases" specifically addressed by each of the following methods and how they are handled—for example, adding an element to an empty list is a "special case."
    a. The `ABList` `add` method
    b. The `ABList` `set` method
    c. The `LBList` indexed `add` method
    d. The `LBList` indexed `remove` method
    e. The `LBList` iterator `remove` method

a) **For the ABList *add* method, our special cases consist of when the array is full or when our position is not in range and therefore the element cannot be added to the list. For the case of being full, we must increase the array, and move the items from the index to the length of the original array 1 position to the right in the array to make room for the new element, while incrementing total the total number of items. This is usually done calling a separate method called expandArray.**

When we consider whether the case of whether the given position is permissible to the current array, we are faced with an IndexOutOfBoundsException. This is usually an user error that will result in a specific return message, "No elements in the array" so users know why their input does not work.

b) **Depending on the manner in which we write the *set* method we have special cases where the content may be altered when trying to implement this method. We would therefore be forced to update the original elements otherwise it would compromise the integrity of the set, and compromise any other methods we may decide to call on the set. The best way to deal with this is to use the wrapping approach, which is similar to the java wrapper class, and hold an object of one type within an object of another type to delegate calls to it. ,**
c) **The linked based sorted list *add* method must take care when adding to the front of the list, rear of the list, or to an empty list. If any of these cases occur, i.e. if an index passed such that index < 0 or index > size() it will throw an**

IndexOutOfBoundsException. Otherwise it will add the element to this list at the position index; all current elements at that position or higher have 1 added to their index. We also have a special case when the the input is already within the list. When this occurs we must handle it by ignoring the input and simply going onto the next input. Optional throws are UnsupportedOperationException if not our operation is not supported.

d) **The indexed *remove* method for the LBList is much trickier than the array based methods. This is because when an index is provided as an argument to a method there is only one way to access the associated element. We must walk through the list from the start or the front reference, until we reach the indicated element. This special case will throws IndexOutOfBoundsException if passed an index argument such that index < 0 or >= size() Otherwise, we must remove the element on this list at position index and returns the removed element; all current elements at positions higher than index have 1 subtracted from their position to keep accurate count in the linked based sorted list implementation of the *remove* method.**