1. Given the phone direct entry class below,

```
public class DirectoryEntry {
        private String name; // person's name
        private String number; // person's phone number

        public void setNumber(String phoneNum) { number = phoneNumber; }
        public String getNumber() { return number; }
 }
```

Complete the following methods for a class that has an instance variable *theDirectory* with the type ArrayList<*DirectoryEntry*>.

(1) The method *addOrChangeEntry* adds an entry to *theDirectory* or changes an existing entry. It returns the old number or 'null' if a new entry is added. **[10 points]**

```
// aName: The name of the person being added or changed
// newNumber: The new number to be assigned
 public String addOrChangeEntry(String name, String newNumber)
 {
        … …
}
```

(2) The method *removeEntry* removes an entry from theDirectory. It returns the entry removed, or null if there is no entry for the input *aName*. **[10 points]**

```
// aName: The name of the person to be removed
public DirectoryEntry remove (String aName)
 {
        … …
}
```

2. Phone numbers and PIN codes can be easier to remember when you find words that spell out the number on a standard phone pad. For example, instead of remembering the combination 5282, you can just think of JAVA.

Write a java program with a recursive method that allows users to enter a PIN with maximum 6 digits and outputs all possible strings. For instance, if user enters '2', the output will be 'A', 'B' and 'C'; if users enter '23', the output will be AD, AE, AF, BD, BE, BF, CD, CD, CF. You're not allowed to use a data structure other than array and ArrayList. You may want to verify the user's input and prompt them to try entering again if the format is incorrect, or throw an exception with an informative message after 10 trials. **[5 points for 'main' + 10 points for the recursive method]**

3. Complete a first in first out (FIFO) queue using only two stacks, with the following methods. **[total 20 points]**

```
public class MyQueue<E>
{
        // define attributes required
        … …

        // add 'e' to the back of the queue
        public void enQueue(E e) {  … …} [3 points]

        // remove the element from the front of the queue and return it
        public E deQueue() { … … } [5 points]

        // return the element at the front of the queue
        public E peek() { … … } [5 points]

        // return true if the queue is empty and not otherwise
        public boolean isEmpty() { … … } [2 points]

        // test the methods above, print if necessary [5 points]
        public static void main(String[] str) { … … }
}
```

4. Complete a data structure MyLinkedList below. **[total 15 points]**

```
public class MyLinkedList<E>
{
        private static class MyNode<E> {
                E item;
                MyNode<E> next;

                MyNode(E element) {
                        this.item = element;
                         this.next = null;
                }
         }

        private MyNode head;

        // implement the following method to reverse elements in the list
        // @start, end: the positions for reversing
        // for example, before calling this method, the list looks like
        // { A -> B -> C -> D -> E}. After calling reverse (1, 3), it will become
        // { A -> D -> C -> B -> E}.
        // @return: true if successful reversing, throw an exception otherwise
        public boolean reverse(int start, int end) throws Exception [10 points]
        { … … }

        // test the methods above, print if necessary [5 points]
        public static void main(String[] str) { … … }
}
```