CS35 Data Structures and Algorithms
Practice Quiz 2, Fall 2015

1. Perform `partition` on the array below. You only need to perform `partition` once, not the whole `quickSort` algorithm. Show (i) the resulting array and (ii) what is returned by the function. Assume the right most element is your pivot. I have provided empty arrays below so that you can show your work - you do not necessarily need to use all of them.

   (NOTE: you should be equally comfortable with merge and mergeSort).

8	10	15	4	13	6	7

   (i) **Final array**:

   (ii) **Return**:

2. For your reference, I have provided part of declarations for the templated `LinkedListNode` and `LinkedList` classes from lecture:

```
// linkedList.h:
#include "list.h"

template <typename T>
class LinkedListNode {
  private:
    T value;
    LinkedListNode<T>* next;

  public:
    LinkedListNode(T value);
    T getValue();
    LinkedListNode<T>* getNext();
    void setNext(LinkedListNode<T>* n);
};
// continued ------->
```

```
template <typename T>
class LinkedList : public List<T> {
  private:
    LinkedListNode<T>* head;
    LinkedListNode<T>* tail;
    int size;

  public:
    LinkedList();
    ~LinkedList();
    void insertAtHead(T value);
    T removeHead();
    //Etc. All methods are available
    LinkedList<T>* reverse();
};
#include "linkedList-inl.h"
```

(a) Implement the `LinkedList` method `reverse`. This method should return a pointer to a LinkedList that contains the current object's elements in reverse order. For example, calling reverse on a list of elements `[1,2,3]` should return a pointer to a list of elements `[3,2,1]`. Calling `reverse` on a list of elements `[apple, blueberry, pecan, key lime]` should return a pointer to a list of elements `[key lime, pecan, blueberry, apple]`. The current list should not be modified, and your method should be efficient. Use correct C++ syntax (i.e., as it would appear in the `linkedList-inl.h`).

```
template <typename T>
LinkedList<T>* LinkedList<T>::reverse() {




}
```

(b) What is the running time of your implementation?

3. Consider a variation of merge sort that uses lists. That is, rather than sorting arrays, we receive as input a set of items stored in a list. The algorithm proceeds as normal – recursively dividing the elements in half for sorting and then merging.

   Below, implement `merge`. You receive two lists `l1, l2` that are already sorted with the smallest values at the front. The sorted combination of the two queues needs to be stored in `result`. You can use **pseudocode**, but your solution must obey the `List` interface from lecture and lab and you should not need to utilize any other data structures.

   HINT: it's actually a pretty simple solution, so don't panic! Think about how merge works and replace the steps with equivalent List operations. Use pseudocode and don't worry about memory management.

```
merge(l1, l2, result):
  Input: three lists; l1 and l2 are sorted.  result is empty.
  Return: none; result should contain the sorted combination
          of l1 and l2 in ascending order.
```

Consider the following program that uses an `ArrayStack` which implements the standard `Stack` interface backed by an `ArrayList`.

```
#include "arraystack.h"
#include<iostream>
using namespace std;

int main(){
  ArrayStack<int>* stack = new ArrayStack;

  stack.push(10);
  stack.push(15);
  stack.push(8);
  stack.push(17);

  cout << stack.getTop() << ", ";
  cout << stack.pop() << ", ";

  stack.push(-1);

  // draw the memory diagram here.

  while(! stack.isEmpty()){
    cout << stack.pop() << ", ";
  }
}
```

4. For the program above, write down the expected output, and the memory diagram at the point indicated.