# CSE 143 Section Handout #13
# Practice Midterm #5

1. **ArrayList Mystery**. Consider the following method:

```
public static void mystery5(ArrayList<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        int element = list.get(i);
        list.remove(i);
        list.add(0, element + 1);
    }
    System.out.println(list);
}
```

Write the output produced by the method when passed each of the following ArrayLists:

**List** **Output**

**(a)** [10, 20, 30] ____________________

**(b)** [8, 2, 9, 7, 4] ____________________

**(c)** [-1, 3, 28, 17, 9, 33] ____________________

2. **ArrayList Programming**. Write a method filterRange that accepts an ArrayList of integers and two integer values *min* and *max* as parameters and removes all elements whose values are in the range *min* through *max* (inclusive) from the list. For example, if a variable called list stores the values:

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

The call of filterRange(list, 5, 7); should remove all values between 5 and 7, therefore it should change the list to store [4, 9, 2, 3, 1, 8]. If no elements in range *min*-*max* are found in the list, the list's contents are unchanged. If an empty list is passed, the list remains empty. You may assume that the list is not null.

3. **Stack and Queue Programming**. Write a method `removeMin` that accepts a stack of integers as a parameter and removes and returns the smallest value from the stack. For example, if a variable `s` stores:

   ```
   bottom [2, 8, 3, 19, 7, 3, 2, 42, 9, 3, 2, 7, 12, -8, 4] top
   ```

   And we make the following call:

   ```
   int n = removeMin(s);
   ```

   The method removes and returns `-8`, so `n` will store -8 after the call and `s` will store the following values:

   ```
   bottom [2, 8, 3, 19, 7, 3, 2, 42, 9, 3, 2, 7, 12, 4] top
   ```

   If the minimum value appears more than once, all occurrences of it should be removed. For example, given the stack above, if we again call `removeMin(s);`, it would return `2` and would leave the stack as follows:

   ```
   bottom [8, 3, 19, 7, 3, 42, 9, 3, 7, 12, 4] top
   ```

   You may use one queue as auxiliary storage. You may not use any other structures to solve this problem, although you can have as many primitive variables as you like. You may not solve the problem recursively. You may assume that the stack is not empty. For full credit, your solution must run in O(*n*) time.

   You have access to the following two methods and may call them as needed to help you solve the problem:

   ```
   public static void s2q(Stack<Integer> s, Queue<Integer> q) {
       while (!s.isEmpty()) {
           q.add(s.pop());           // Transfers the entire contents
       }                             // of stack s to queue q
   }
   public static void q2s(Queue<Integer> q, Stack<Integer> s) {
       while (!q.isEmpty()) {
           s.push(q.remove());       // Transfers the entire contents
       }                             // of queue q to stack s
   }
   ```

4. **Collections Programming**. Write a method `whoPassed` that determines which students "passed" a course. The method accepts three parameters: A `students` map from students' names (strings) to their overall percentages (integers) in the course; a `grades` map from percentages (integers) to course grades out of 4.0 (real numbers); and a `minGrade` real number representing the minimum grade out of 4.0 required to pass. Your method should return a set containing the names of all students who earned at least the given minimum grade out of 4.0 (inclusive). For example, if your method is passed the following parameters:

   ```
   students = {Marty=76, Dan=81, Alyssa=98, Kim=52, Lisa=87, Whit=43, Jeff=70, Sylvia=92}
   grades   = {76=2.1, 81=2.6, 98=4.0, 52=0.0, 87=3.3, 43=0.0, 70=1.5, 92=3.7}
   minGrade = 2.6
   ```

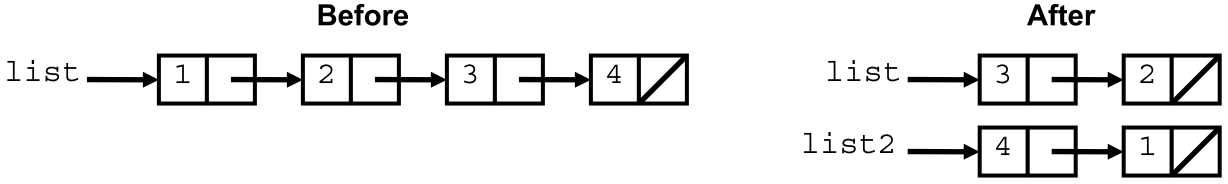
   Then your method should return the set `[Alyssa, Dan, Lisa, Sylvia]`, because Alyssa's percentage of 98 earns her a grade of 4.0 in the course, Dan's 81 earns him a 2.6, Lisa's 87 earns her a 3.3, and Sylvia's 92 earns her a 3.7.

   The names can appear in any order in the set. If no students meet the desired minimum grade, return an empty set. You may assume that no parameter is `null`, that every student's percentage is between 0 and 100 inclusive, that the `grades` map contains a grade entry between 0.0 and 4.0 inclusive for every percentage earned by a student, and that the `minGrade` parameter is between 0.0 and 4.0 inclusive. For full credit your code must run in less than O($n^2$) time where *n* is the number of students and/or percentages.

# CSE 143 Section Handout #13

5. **Linked Nodes**. Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but you are NOT allowed to change any existing node's `data` field value. You also should not create new `ListNode` objects unless necessary to add new values to the chain, but you may create a single `ListNode` variable to refer to any existing node if you like. If a variable does not appear in the "after" picture, it doesn't matter what value it has after the changes are made.

   To help maximize partial credit in case you make mistakes, we suggest that you include optional comments with your code that describe the links you are trying to change, as shown in Section 7's solution code.



   Assume that you are using the `ListNode` class as defined in lecture and section:

```
public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // a link to the next node in the list

    public ListNode() { ... }
    public ListNode(int data) { ... }
    public ListNode(int data, ListNode next) { ... }
}
```

6. **Linked List Programming**. Write a method `hasAlternatingParity` that could be added to the `LinkedIntList` class from lecture that returns whether or not the list of integers has alternating parity (`true` if it does, `false` if not). The *parity* of an integer is 0 for even numbers and 1 for odd numbers. To have alternating parity, a list would have to alternate between even and odd numbers, as in the list:

```
[3, 2, 19, 8, 43, 64, 1, 0, 3]
```

   If a variable called `list` stores the values above, then the call of `list.hasAlternatingParity()` would return `true`. If instead the list stored the following values, the call would return `false` because the list has two even numbers in a row (4 and 12):

```
[2, 13, 4, 1, 0, 9, 2, 7, 4, 12, 3, 2]
```

   By definition, an empty list or a list of one element has alternating parity. You may assume that every element in the list is greater than or equal to 0.

   Assume that we are adding this method to the `LinkedIntList` class as seen in lecture and as shown below. You may not call any other methods of the class to solve this problem and your method cannot change the contents of the list.

```
public class LinkedIntList {
    private ListNode front;

    methods
}
```

7. **`Comparable` Programming**. Suppose you have a pre-existing class `Pokemon` that represents characters in a game. The class has the following data and behavior:

| Field/Constructor/Method | Description |
|---|---|
| `private String name` | name of the Pokemon, such as `"Pikachu"` |
| `private boolean aggressive` | `true` if this Pokemon likes to fight |
| `private int attack` | attack power of the Pokemon |
| `private int defense` | defense power of the Pokemon |
| `public Pokemon(String name, boolean aggressive, int attack, int defense)` | makes a Pokemon with the given attributes |
| `public int getAttack()` | returns the Pokemon's attack power |
| `public int getDefense()` | returns the Pokemon's defense power |
| `public String getName()` | returns the Pokemon's name |
| `public boolean isAggressive()` | returns `true` if the Pokemon likes to fight, and false if he is defensive |

**Make `Pokemon` objects comparable to each other using the `Comparable` interface**. All defensive Pokemon are considered to be "less than" aggressive Pokemon. An defensive Pokemon is considered to be "less than" another if it has a lower defense power. An aggressive Pokemon is considered to be "less than" another if it has a lower sum of attack and defense power. If two Pokemon are both aggressive or defensive and have the same defense or attack+defense power, they are considered to be "equal." Your method should not modify any Pokemon object's state. You may assume the parameter passed is not `null`.

8. **Searching and Sorting**.

**(a)** Suppose we are performing a **binary search** on a sorted array called `numbers` initialized as follows:

```
// index           0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
int[] numbers = {-2,  4,  6,  7, 11, 15, 29, 33, 38, 45, 56, 59, 70, 80, 81, 99};

// search for the value -10
int index = binarySearch(numbers, -10);
```

Write the indexes of the elements that would be examined by the binary search (the `mid` values in our algorithm's code) and write the value that would be returned from the search. Assume that we are using the binary search algorithm shown in lecture and section.

- Indexes examined: ______________________________________________
- Value Returned: ____________________

**(b)** Write the state of the elements of the array below after each of the first 3 passes of the outermost loop of the **selection sort** algorithm.

```
int[] numbers = {45, 78, 89, 34, 23, 12, 67, 56};
selectionSort(numbers);
```

**(c)** Trace the complete execution of the **merge sort** algorithm when called on the array below, similarly to the example trace of merge sort shown in the lecture slides. Show the sub-arrays that are created by the algorithm and show the merging of sub-arrays into larger sorted arrays.

```
int[] numbers = {45, 78, 89, 34, 23, 12, 67, 56};
mergeSort(numbers);
```

# CSE 143 Section Handout #13

9. **Recursive Tracing**. For each call to the following method, indicate what output is produced:

```
public void mystery(int n) {
    if (n % 2 == 1) {
        System.out.print(n);
    } else {
        System.out.print(n + ", ");
        mystery(n / 2);
    }
}
```

| Call | Output |
|---|---|
| `mystery(13);` | |
| `mystery(42);` | |
| `mystery(40);` | |
| `mystery(60);` | |
| `mystery(48);` | |

10. **Recursive Programming**. Write a recursive method `indexOf` that accepts two `String`s as parameters and that returns the starting index of the first occurrence of the second `String` inside the first `String` (or -1 if not found). The table below lists several calls to your method and their expected return values. Notice that case matters, as in the last example that returns -1.

| Call | Value Returned |
|---|---|
| `indexOf("Barack Obama", "Bar")` | 0 |
| `indexOf("Barack Obama", "ck")` | 4 |
| `indexOf("Barack Obama", "a")` | 1 |
| `indexOf("Barack Obama", "McCain")` | -1 |
| `indexOf("Barack Obama", "BAR")` | -1 |

`String`s have an `indexOf` method, but you are not allowed to call it. You are limited to these methods:

| Method | Description |
|---|---|
| `equals(`**other**`)` | returns `true` if the two strings contain the same characters |
| `length()` | returns the number of characters in the string |
| `substring(`**fromIndex**`, `**toIndex**`)`<br>`substring(`**fromIndex**`)` | returns a new string containing the characters from this string from **fromIndex** (inclusive) to **toIndex** (exclusive), or to the end of the string if **toIndex** is omitted |

You are not allowed to construct any structured objects other than `Strings` (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion.