Software Testing Homework 1: Debugging

1. Max binary heap

Issue 1 - The program should heapify any given list of positive integers but the resulting tree (and list) does not meet the max binary heap structure.
Input: [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0]
Expected output: [11, 10, 8, 7, 9, 1, 5, 2, 3, 4, 6, 1, 0]
Actual output: [11, 9, 5, 3, 9, 7, 1, 6, 3, 4, 2, 1, 0]

Bug 1

The first bug is on the second and third line of `leftChildIndex` method:

```
private int leftChildIndex(int index) {
    if (valueList.size() > (2 * index) - 1) {
        return (2 * (index) - 1);
    } else return -1;
}
```

If we have a binary heap structure, the index of the left child of an element can be found by $(2 \cdot \texttt{index}) + 1$ where `index` refers to the parent element's position. Therefore, the correct method would be:

```
private int leftChildIndex(int index) {
    if (valueList.size() > (2 * index) + 1) {
        return (2 * (index) + 1);
    } else return -1;
}
```

Bug 2

Another bug is on the third line in the following section of the `BubbleDown` method:

```
if (currentElem < leftChild) {
    if (leftChild > rightChild) {
        valueList.set(index, rightChild);
        valueList.set(leftChildIndex, currentElem);
        BubbleDown(leftChildIndex);
    } else {
        valueList.set(index, rightChild);
        valueList.set(rightChildIndex, currentElem);
    }
}
```

The third and fourth line should swap `currentElem` and `leftChild`, but instead change `currentElem` to `rightChild` and `leftChild` to `currentElem`. For this reason, the if-block of the method should be:

```
if (leftChild > rightChild) {
    valueList.set(index, leftChild);
    valueList.set(leftChildIndex, currentElem);
    BubbleDown(leftChildIndex);
}
```

Bug 3

Third bug is also in the `BubbleDown` method, but in the else-block of the above code shown regarding the second bug:

```
else {
    valueList.set(index, rightChild);
    valueList.set(rightChildIndex, currentElem);
}
```

Even though the swap is done correctly, this part of the method is missing a `BubbleDown` method call, since it's possible that we might need to bring an element down more than one level (currently does exactly one swap). For this reason, the correct code would be:

```
else {
    valueList.set(index, rightChild);
    valueList.set(rightChildIndex, currentElem);
    BubbleDown(rightChildIndex);
}
```

Issue 2 - When adding elements to a heap, the element should automatically go to the correct location in the heap. However, after adding elements, the resulting list (tree) does not match

the expected max bin heap structure, there are different elements than what should be there (some more, some missing).
Input: [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0], then add integers 13 and 4
Expected output: [13, 10, 11, 7, 9, 1, 8, 2, 3, 4, 6, 1, 0, 5, 4]
Actual output: [11, 13, 5, 13, 9, 7, 13, 6, 3, 4, 2, 1, 0, 13, 4]

Bug 4

There is a bug in the `BubbleUp` method:

```
if (currentElem > parent) {
    valueList.set(parent, index);
    valueList.set(parentIndex, currentElem);
    BubbleUp(parentIndex);
    }
```

What happens there is that the `set` method should take `index` and `parent` as arguments in that order, but in the present case, the order is switched so the method inserts the `index` at the position that corresponds to the value of the `parent`. For this reason, the correct order should be as outlined below:

```
if (currentElem > parent) {
    valueList.set(index, parent);
    valueList.set(parentIndex, currentElem);
    BubbleUp(parentIndex);
}
```

Bug 5

There is also a bug in the `elemParentIndex` method:

```
private int elemParentIndex(int index) {
    if (index > 0) {
        return index / 2;
    } else return -1;
}
```

The method can return an incorrect value, since an element's parent index should be calculated $\lfloor (\texttt{index} - 1)/2 \rfloor$. Therefore, the correct version would be (flooring is not required since Java already rounds towards zero):

```
private int elemParentIndex(int index) {
    if (index > 0) {
        return (index - 1) / 2;
    } else return -1;
}
```

Issue 3 - When removing a given element (the first occurrence of that element) the element

should be gone, the rest of the heap in a correctly sorted order, and the list itself one element shorter. The element is gone, but the heap is not in the correct order and the size of the list is not smaller.

Input: [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0], add elements 13 and 4, then remove element 10 and try to remove non-existing element 99.

Expected output: [13, 9, 11, 7, 6, 1, 8, 2, 3, 4, 4, 1, 0, 5]

Actual output: [11, 13, 5, 13, 9, 7, 13, 6, 3, 4, 2, 1, 0, 13, 4]

Bug 6

The last bug is in the following section of the `removeElem` method:

```
valueList.remove(valueList.size() - 1);
valueList.set(indexOfRemoveElem, lastElem);
int parentOfRemoved = elemParent(indexOfRemoveElem);
if (lastElem < parentOfRemoved) {
    BubbleUp(indexOfRemoveElem);
} else if (lastElem > parentOfRemoved) {
    BubbleDown(indexOfRemoveElem);
}
```

Restructuring of the heap is done incorrectly in the above method. The method states that if `lastElem` is smaller than `parentOfRemoved`, `BubbleUp` is be called on it and similarly, if `lastElem` is larger, `BubbleDown` is called. For the restructuring to work correctly, the logic should be implemented in reverse; calling `BubbleDown` when the element is smaller since it might also be smaller than its children and calling `BubbleUp` when it's larger. Thus, the corrected version of this part would be:

```
valueList.remove(valueList.size() - 1);
valueList.set(indexOfRemoveElem, lastElem);
int parentOfRemoved = elemParent(indexOfRemoveElem);
if (lastElem < parentOfRemoved) {
    BubbleDown(indexOfRemoveElem);
} else if (lastElem > parentOfRemoved) {
    BubbleUp(indexOfRemoveElem);
}
```

2. Genetic algorithm