1 More Practice with Linked Lists

Recall the definition of `SList` from lecture:

```
public class SList {
    private class SNode {
        public int item;
        public SNode next;
        public SNode(int item, SNode next) {
            this.item = item;
            this.next = next;
        }
    }

    private SNode front;

    public void insertFront(int x) {
        front = new SNode(x, front);
    }
}
```

1.1 Insert

Add a method to the `SList` class that inserts a new element at the given position. If the position is past the end of the list, insert the new node at the end of the list. For example, if the SList is 5 –> 6 –> 2, `insert(10, 1)` should result in 5 –> 10 –> 6 –> 2.

```
public void insert(int item, int position) {
    if (front == null || position == 0) {
        insertFront(item);
        return;
    }
    SNode currentNode = front;
    while (position > 1 && currentNode.next != null) {
        position--;
        currentNode = currentNode.next;
    }
    SNode newNode = new SNode(item, currentNode.next);
    currentNode.next = newNode;
}
```

1.2 Reverse

Add another method to the `SList` class that reverses the elements. Do this using the existing `SNodes` (you should not use `new`).

```
public void reverse() {
    // One way to think about this method is that we're going to
    // traverse through the SNodes, and for each SNode, we're going
    // to insert it at the front of the new list. To do this, we'll
    // maintain two pointers: one to the current front node of our
    // newly reversed list (frontOfReversed), and one to the next
    // node in the un-reversed part of the old list (nextNodeToAdd).
    SNode frontOfReversed = null;
    SNode nextNodeToAdd = front;
    while (nextNodeToAdd != null) {
        SNode remainderOfOriginal = nextNodeToAdd.next;
        // Put nextNodeToAdd on the front of the reversed list.
        nextNodeToAdd.next = frontOfReversed;
        // Update the pointers.
        frontOfReversed = nextNodeToAdd;
        nextNodeToAdd = remainderOfOriginal;
    }
    front = frontOfReversed;
}
```

Bonus: If you wrote `reverse()` iteratively, write a second version that uses recursion (you may need a helper method). If you wrote it recursively, write an iterative version.

```
private SNode reverseRecursiveHelper(SNode front) {
    if (front == null || front.next == null) {
        return front;
    } else {
        // Reverse everything except the front node.
        SNode reversed = reverseRecursiveHelper(front.next);
        // Put the front onto the back of the new reversed list.
        // Since everything after front got reversed, front.next is
        // the LAST thing in reversed. Change this last thing's
        // next pointer to point to front, so front is now at the back
        // of reversed.
        front.next.next = front;
        // Since the front is now the last element, its next pointer
        // should be null.
        front.next = null;
        return reversed;
    }
}

public void reverse() {
    front = reverseRecursiveHelper(front);
}
```

2 Arrays

2.1 Insert

Write a method that non-destructively inserts `item` into array `x` at the given position. The method should return the resulting array. For example, if `x = [5, 9, 14, 15]`, `item = 6`, and `position = 2`, then the method should return `[5, 9, 6, 14, 15]`. If `position` is past the end of the array, insert `item` at the end of the array.

```
public static int[] insert(int[] x, int item, int position) {
    int[] newX = new int[x.length + 1];
    position = Math.min(x.length, position);
    for (int i = 0; i < position; i++) {
        newX[i] = x[i];
    }
    // Alternately, if you can remember the syntax for this method:
    // System.arraycopy(x, 0, newX, 0, position);
    newX[position] = item;
    for (int indexInOld = position; indexInOld < x.length; indexInOld++) {
        newX[indexInOld + 1] = x[indexInOld];
    }
    // Alternately:
    // System.arraycopy(x, position, newX, position + 1, x.length -
       position)
    return newX;
}
```

Is it possible to write a version of this method that returns void and changes `x` in place (i.e., destructively)?

No, because arrays have a fixed size, so to add an element, you need to create a new array.

2.2 Bonus: reverse

Write a method that destructively reverses the items in `x`. For example calling `reverse` on an array `[1, 2, 3]` should change the array to be `[3, 2, 1]`.

```
public static void reverse(int[] x) {
    for (int i = 0; i < x.length / 2; i++) {
        int j = x.length - i - 1;
        // Can separate following into "swap" method
        int temp = x[i];
        x[i] = x[j];
        x[j] = temp;
    }
}
```

2.3 Bonus: xify

Write a non-destructive method `xify(int[] x)` that replaces the ith number with x[i] copies of itself. For example, `xify([3, 2, 1])` would return `[3, 3, 3, 2, 2, 1]`.

```
public static int[] xify(int[] x) {
    int total = 0;
    for (int item : x) {
        total += item;
    }
    int[] newX = new int[total];
    int newIndex = 0;
    for (int item : x) {
        for (int counter = 0; counter < item; counter++) {
            newX[newIndex] = item;
            newIndex++;
        }
    }
    return newX;
}
```