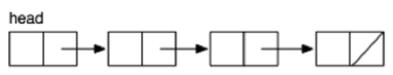
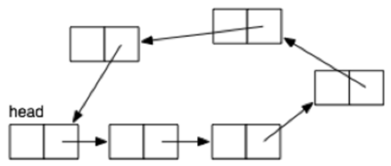
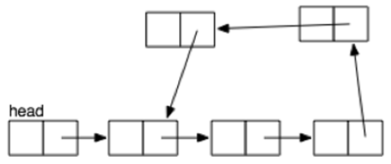
170025298 W08-Lists Tutor: Michael Weir **11/4/2018**

Outline:
1. Basic – Iterative List Manipulator
2. Extension – Recursive List Manipulation

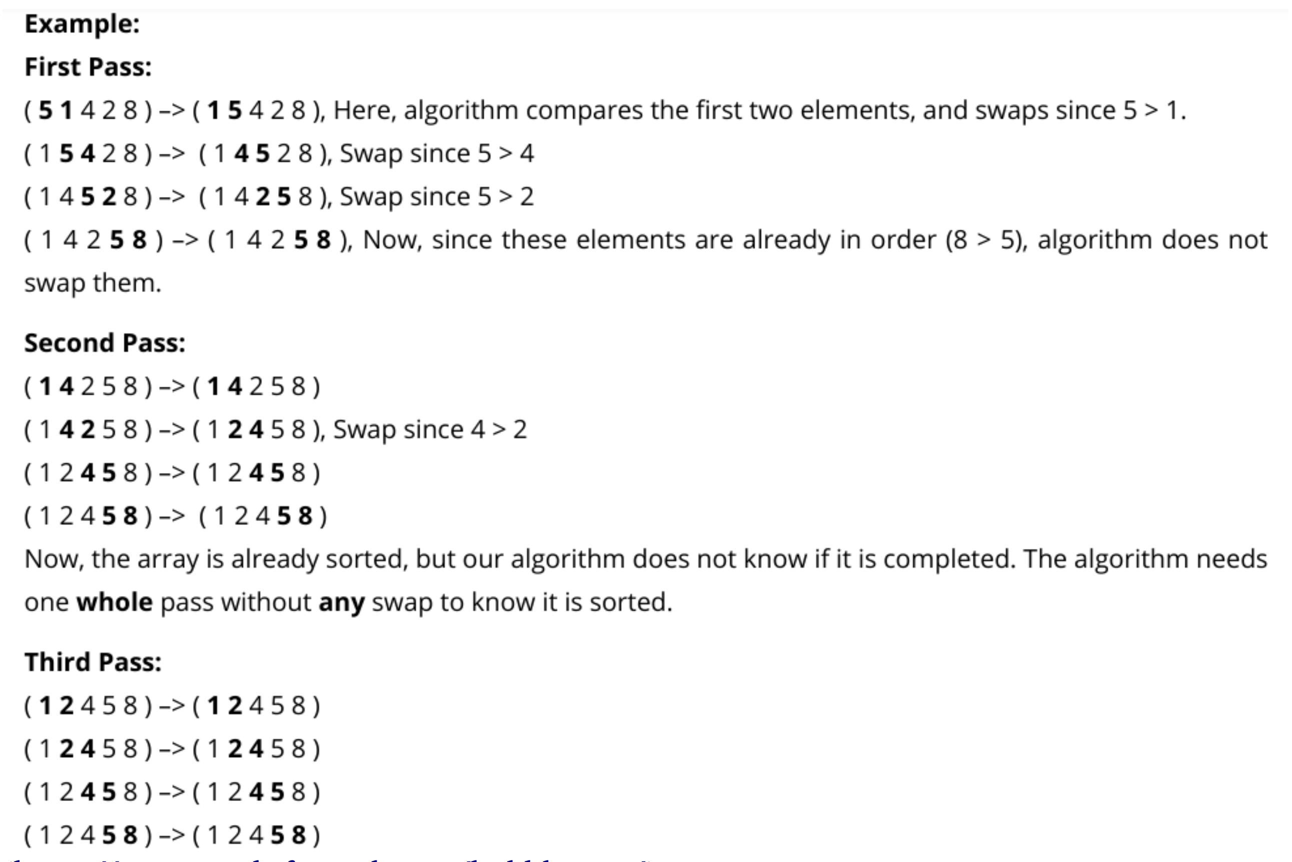
Basic: Iterative List Manipulator

1. Design:
1.0.1 structure of ListNode:
ListNode → {object: element, ListNode: next → {…… → {object: element, null}}}
1.0.2 Deal with null (ListNode)head:
When the program find the head == null, the program will immediately return the value with 0/null/(Blank)"
(constant)1.1 size: As structure shows above (1.0), when counting the size of the list node, the program should read across the list node until it reaches to null. The size of listnode will always add 1 when a successful read happens.
(Linear) 1.2 contains: As structure shows still, when finding whether the element is contained in ListNode, the root will read whole ListNode head and when the root.next.element is equal to the object in in-line variable, boolean contains will be set as true.
1.2.improve contain method might be more efficient when implement method with direct "return true;" or "return false".
(Constant) 1.3 count: With structure shows in 1.0, count method is designed similar to size method, however the size will only increase by 1 when root.element = (in line variable)element.
(Linear) 1.4 convert to string: This method will print all element in the (ListNode)head which separated with comma. So root will read across the (ListNode)head and add each root.element with comma at the front and return such string at the end.
(Constant) 1.5 get from front / back: In this method, the input value n might larger than size(n) (1.1), so the program will throw InvalidIndexException when input value n is larger than the size of the head.
In get from front method, the reading will start from first value to nth value which will throw the invalid index exception when the next list node of root is null as the current node is less than input value n; get from back method follows similarly.
(Linear) 1.6 deepequals: Deep equal method will judge the element of two ListNodes in in-line variable are in the same position and the element are the same. The method will judge the null value first and then compare each element at the same time in both nodes. Once the element are not equal, the method will return false, else the program will return true.
(Linear) 1.7 deepcopy: Deep copy will copy only with the element in each listnodes and make a new ListNode finally. This method is related to append method(1.8) which can add the new list node to the original node. The program append (ListNode)deepcopy with new node with only element inside and return the deepcopy at the end.

(Constant) 1.8 append: Append method will add the input ListNode head2 to the last element of head 1 and return a new ListNode head1 at the end. Before this process the method will check three conditions: head1 and head2 are null or one of the head is null, for different conditions the method will return null/ head1/ head2.
(Linear) 1.9 contains duplicates: duplicates will happens when an element occurred more than once, which is a great place to use count(1.3) before to count the apperance of each element. Once an element occured more than twice the method will return true.
(Linear) 1.10 flatten: flatten requires types of element inside are ListNodes. Otherwise it will return error message but not being thrown. This method is also related to append(1.8) method which will append ListMethod with combined list which will be set as the first element of the root. The program will read through the ListNode and combine every ListNode in root.next and will return the combined list in the end.
(Quadratic) 1.11 is circular/conatins cycle:

		
isCircular(): false containsCycles(): false	isCircular(): true containsCycles(): true	isCircular(): false containsCycles(): true

To jutisfy the design I copied the image from the requirement.
A circular ListNode is like a clock, when minute is equal to hour, two arrows must be in the 12 o'clock.
So the design of circular is when the arrow is not in 12 o'clock but minute == hour, it fits to the third condition as image shows; when minute == hour in 12 o'clock, the condition is as second condition. The first image will happens when the method finds fast == null which shows that the ListNodes are not circular.
(Linear) 1.12 sort: As lecture metions bubble sort, I uses bubble sort to sort ListNodes. The ListNode.element will be changed with ListNode.next.element when using comparator shows the result > 0 and it will return a sorted linked list at the end. The following is the theorem which I searched from the internet about bubble sort:


Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

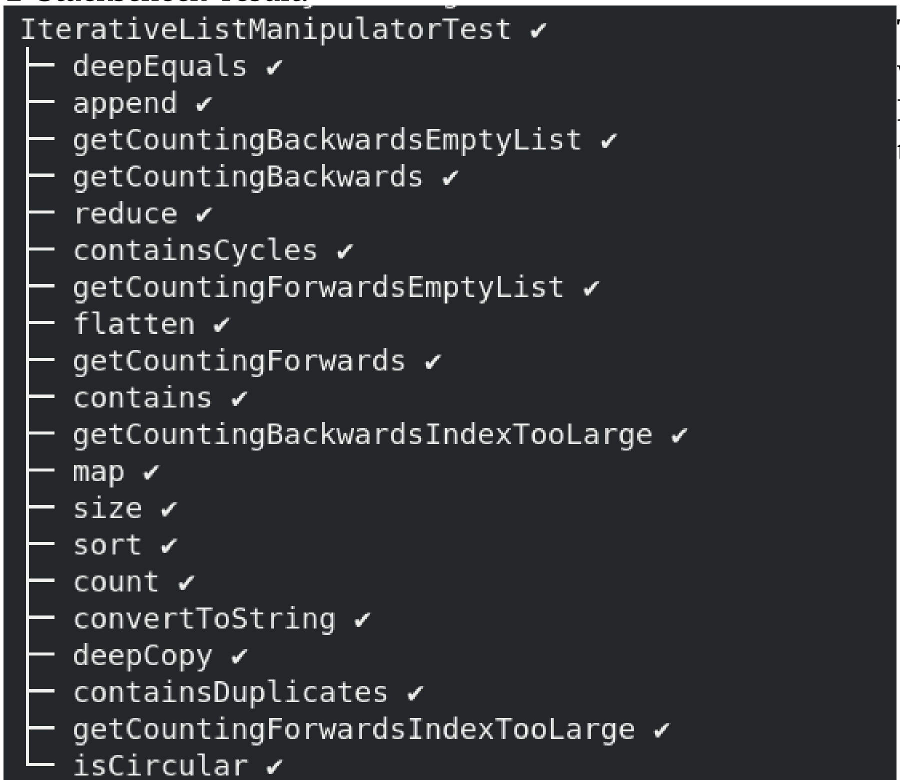
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

(https://www.geeksforgeeks.org/bubble-sort/)

(Linear) 1.13 map: Map method will use transformation method to transform each element in ListNodes and add them to the new List, and the design follows such theorem which append(1.8) each transformed ListNode with new ListNode to get new ListNodes.

(Linear) 1.14 reduce: Reduce method will use operator method to operate initial object and each nodes in (ListNodes)head. After operate all elements and initial objects the method will return reduce result as an object.

2 Stackscheck result:

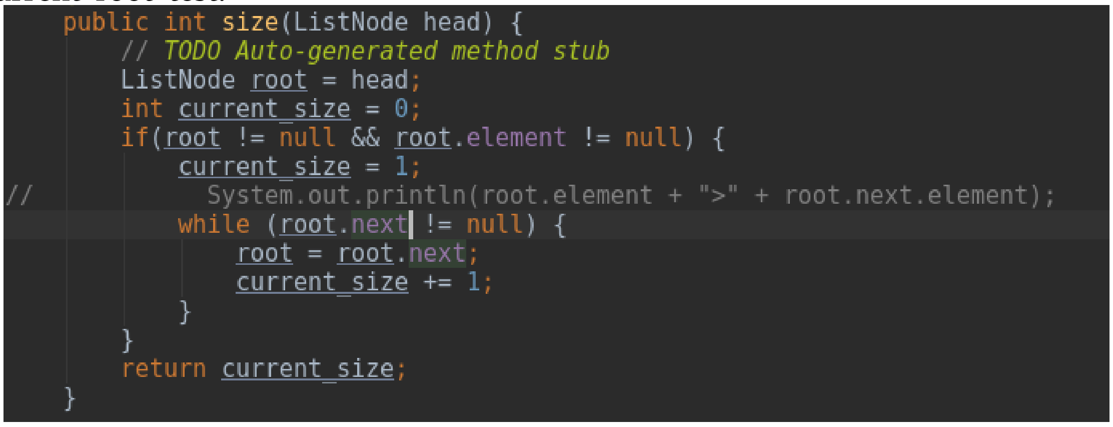


```
IterativeListManipulatorTest ✔
├─ deepEquals ✔
├─ append ✔
├─ getCountingBackwardsEmptyList ✔
├─ getCountingBackwards ✔
├─ reduce ✔
├─ containsCycles ✔
├─ getCountingForwardsEmptyList ✔
├─ flatten ✔
├─ getCountingForwards ✔
├─ contains ✔
├─ getCountingBackwardsIndexTooLarge ✔
├─ map ✔
├─ size ✔
├─ sort ✔
├─ count ✔
├─ convertToString ✔
├─ deepCopy ✔
├─ containsDuplicates ✔
├─ getCountingForwardsIndexTooLarge ✔
└─ isCircular ✔
```

The stacscheck passed with all IterativelistManipulator test

3 Testing:

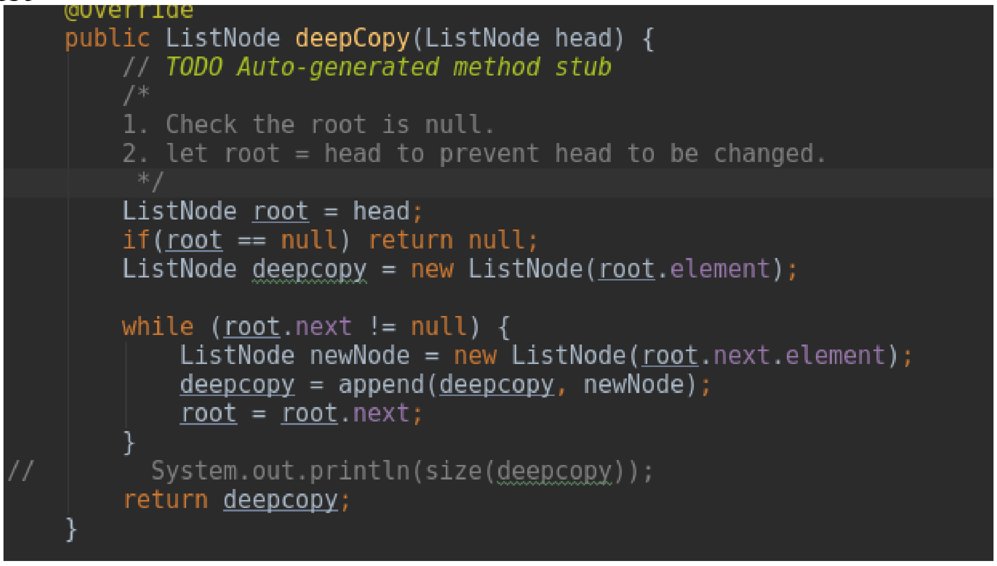
3.1 Current root test:



```java
    public int size(ListNode head) {
        // TODO Auto-generated method stub
        ListNode root = head;
        int current_size = 0;
        if(root != null && root.element != null) {
            current_size = 1;
//            System.out.println(root.element + ">" + root.next.element);
            while (root.next != null) {
                root = root.next;
                current_size += 1;
            }
        }
        return current_size;
    }
```

Current root test test the current list node that the program is reading in order to prevent the condition that the next listnode is null by wrong programming logic.

3.2 size test

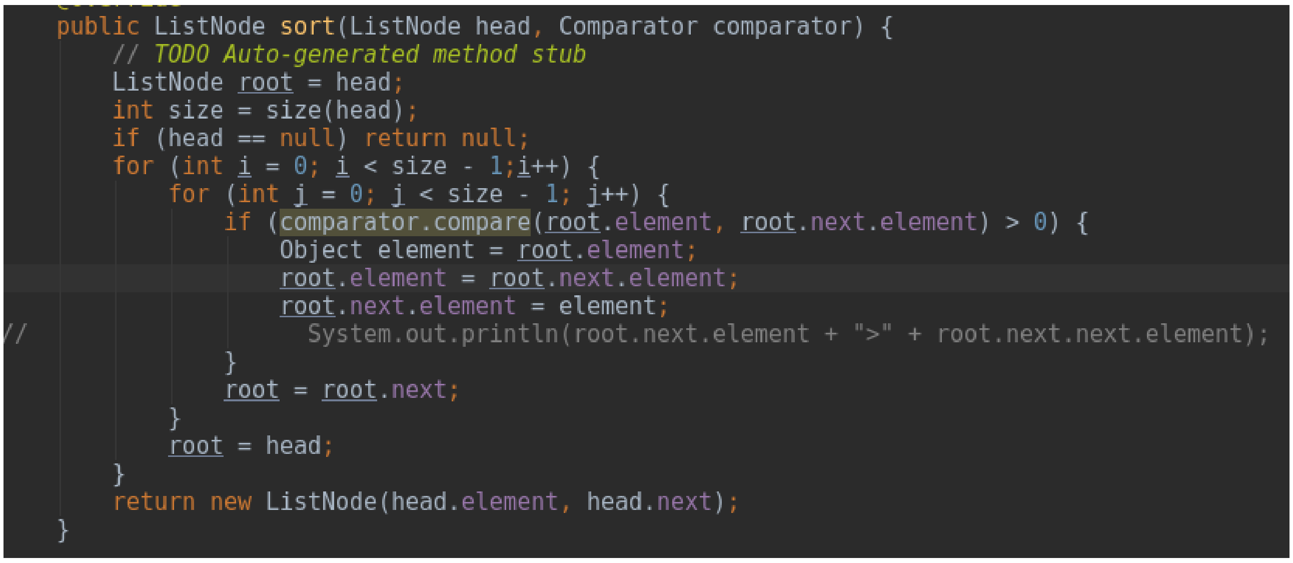


```java
    @Override
    public ListNode deepCopy(ListNode head) {
        // TODO Auto-generated method stub
        /*
        1. Check the root is null.
        2. let root = head to prevent head to be changed.
         */
        ListNode root = head;
        if(root == null) return null;
        ListNode deepcopy = new ListNode(root.element);

        while (root.next != null) {
            ListNode newNode = new ListNode(root.next.element);
            deepcopy = append(deepcopy, newNode);
            root = root.next;
        }
//        System.out.println(size(deepcopy));
        return deepcopy;
    }
```

Example above is the test about size of deep copy which aims at briefly test the size of deepcopy listnode is equal to the size of head, which shows both listnodes have equal size.

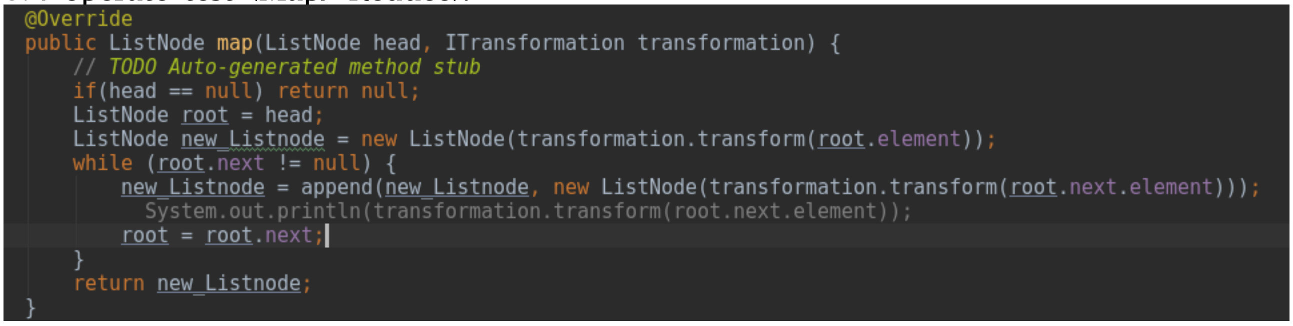
3.3 sorting test:



```
    public ListNode sort(ListNode head, Comparator comparator) {
        // TODO Auto-generated method stub
        ListNode root = head;
        int size = size(head);
        if (head == null) return null;
        for (int i = 0; i < size - 1;i++) {
            for (int j = 0; j < size - 1; j++) {
                if (comparator.compare(root.element, root.next.element) > 0) {
                    Object element = root.element;
                    root.element = root.next.element;
                    root.next.element = element;
//                    System.out.println(root.next.element + ">" + root.next.next.element);
                }
                root = root.next;
            }
            root = head;
        }
        return new ListNode(head.element, head.next);
    }
```

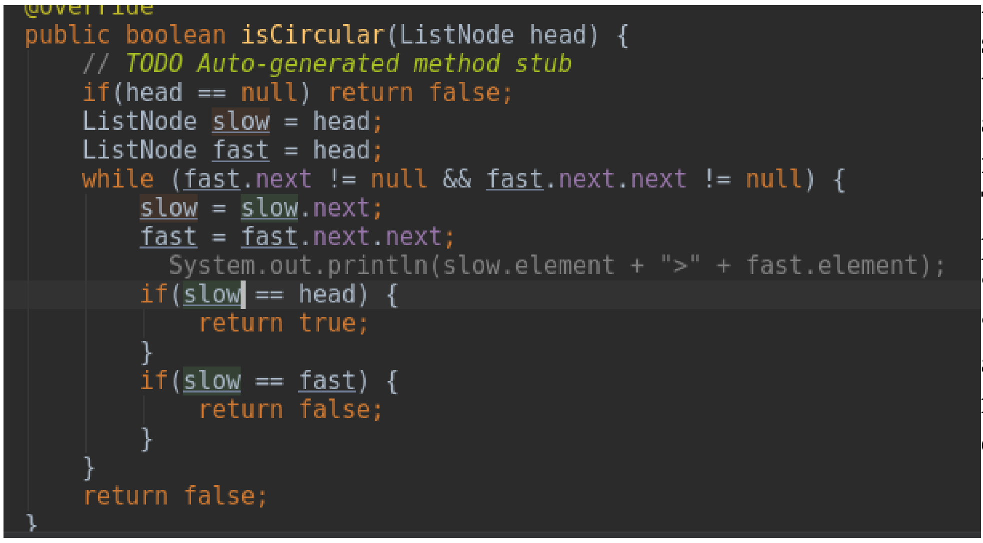
The example prints the next element and the next two element which shows the process of sorting, in order to prove (1.12) theorem.

3.4 operate test (Map/ Reduce):



```
@Override
public ListNode map(ListNode head, ITransformation transformation) {
    // TODO Auto-generated method stub
    if(head == null) return null;
    ListNode root = head;
    ListNode new_Listnode = new ListNode(transformation.transform(root.element));
    while (root.next != null) {
        new_Listnode = append(new_Listnode, new ListNode(transformation.transform(root.next.element)));
        System.out.println(transformation.transform(root.next.element));
        root = root.next;
    }
    return new_Listnode;
}
```

The example is in the map method which prints the result after transforming method in order to return the correct value.
The similar theorem follows by reduce method.
3.5 clock test:



```
@Override
public boolean isCircular(ListNode head) {
    // TODO Auto-generated method stub
    if(head == null) return false;
    ListNode slow = head;
    ListNode fast = head;
    while (fast.next != null && fast.next.next != null) {
        slow = slow.next;
        fast = fast.next.next;
          System.out.println(slow.element + ">" + fast.element);
        if(slow == head) {
            return true;
        }
        if(slow == fast) {
            return false;
        }
    }
    return false;
}
```

Example below shows the clock theorem which aiming at solve infinity circular. The test simply print the result of "hour" and "minute" (See 1.11) and to check the return value is correct.

Extensions: Recursive List Manipulator

4. Design:
4.1 Basic: Recursive method calls back to itself once and once again.
(Constant) 4.2 size: this method is designed as logic follows:
when the current node is not null the program will call back and add 1.
when the current node is null and the program will return the current value.
(Linear) 4.3 contains: This is similar to size() which just read through the ListNode and when element is found from the listnode the program will return true.
(Linear) 4.4 count: This method will only add 1 when the current node is equal to the element that from in-line variable.
(Linear) 4.5 convert to string: This method will read through the program and return call back with plus the element cast to String
(Linear) 4.6 deep equals: The method will stop when both list node is equal to null which will return true; when one of the list node goes to the end of the list node or the element in bot nodes are not equal, the program will return false.
(Linear) 4.7 deep copy: This method is designed with returning a new list node which calls back in list.next variable, when the current node goes to an end, this function will return the full list node.
(Linear) 4.8 contains duplicates: This method will use counts method to judge duplicates that while counts returns more than 2 the method will return true.
(Constant) 4.9 append: Append will return values in three conditions:
4.9.1 head1 == null && head2 == null: In this condition, both head1 and head2 are null or reach to the end, so it will return a full list of append.
4.9.2 head1 != null: This will read the head1 until the head1 == null
4.9.3 head2 != null: This will read head2 until the head2 == null
(Constant) 4.10 map: This method uses transformation to transform the given object in head node, so the method is designed similar to deep copy that it will return new listnode with element and call back until reach to the end.
(Linear) 4.11 reduce: Reduce is not a complete method once operator returns not an integer, however it does works fine by call back reduce when the head is not null and return operate(initial, 0) when the head == null.

P.S: At for time matter, the rest of the recursive list manipulator haven't been finished.