Mark Guerin

Anna Koufakou

COP 3530 – 10413

25 February 2021

Lists Lab

1.
```
public boolean appendList(DynamicList otherList) {
    // If the list to be appended is not empty
    if(!otherList.isEmpty()) {
        // Create a node for the original list
        DynamicNode np = null;
        // Find the end of the original list
        for(np = head; np.getNext() != null; np = np.getNext());
        // Create a node for the list to be added
        DynamicNode ot = null;
        // Loop through the list to be added
        for(ot = otherList.head; ot.getNext() != null; ot = ot.getNext())
{
            // Add the currently pointed to value in the list to be added
to the end of the original list
            np.setNext(ot);
            // Move the original list pointer to the end of the list
            np = np.getNext();
        }
        return true;
    }
    // If the list to be appended is empty, return false
    return false;
 }
```

I created lists named firstList with values [1, 6, 5], and secondList with values [3, 8, 2].

To append and print the updated list, I used the following code inside my main method after creating the lists:

```
firstList.appendList(secondList);
firstList.printList();
```

The console output the following:

```
1
6
5
3
8
2
```

2. The method only appears to work properly for cases B, C, and D. It works for case A if there is more than one item in the list and the value being searched for is not the last value in the list. It appears that the method will insert the value into the list anyway if the value being searched for is the last value in the list. This is because of the line

```
while(nd.getNext(!= null) {
```

which, if looking at the last item in the list, returns false and moves to insert the new node into the list with the parameter value regardless of what value was the last in the list. In other words, this searchInsert method does not check the last item in the list. I have added some comments to the code to explain how it runs:

```
public DynamicNode searchInsert(Object x) {
        // If the list is not empty
        if(!this.isEmpty())   {
            // Set the pointer to the first item in the list
            DynamicNode nd = head;
            // While there is another node ahead (This is bad! The last
item will never be checked!)
            while (nd.getNext() != null) {
                // If the object in the current node is the one for
which we are searching
                if (nd.getInfo().equals(x)) {
                    // Return the node in which it is
                    return nd;
                }
                // Set the pointer to the next node in the list
                nd = nd.getNext();
            }

            // Create a new node containing the object for which we are
searching and set it as the next item in the list
            nd.setNext(new DynamicNode(x, null));
            // Set the pointer to the next node in the list
            nd = nd.getNext();
            // Return the node in which it is
            return nd;
        }

        // If the list is empty, create a new node containing the
object for which we are searching
        DynamicNode nd2 = new DynamicNode(x, null);
        // Set the first item in the list as this new node
        head = nd2;
        // Return the node
        return nd2;
    }
```

3. a) For a list with four nodes, the mystery method returns null and does nothing to the list because there is an even number of nodes in the list. For a list with five nodes, the mystery method deletes the item that was in the middle of the list and returns the item that was deleted because there is an odd number of nodes in the list.

   b) The current node is used to determine the item just before the middle of the list, and to make its next pointer refer to the item two nodes ahead of it, effectively deleting the item in the middle from the list. The temp1 pointer is used to determine the item in the middle of the list, and to assist in the deletion of that item. The temp2 pointer is used to determine the end of the list at the same time as the middle of the list is determined. temp2 is incremented twice per loop, while temp1 is only incremented once per loop, allowing the method to determine if there is an even or odd number of items in the list. If it is even, temp2 reaches the end of the list before it is checked inside the loop. If it is odd, it reaches the end of the list at the very end of a loop iteration, and is checked before the next iteration, preventing the method from returning null.