COMP132 – Computer Science II
Fall 2015

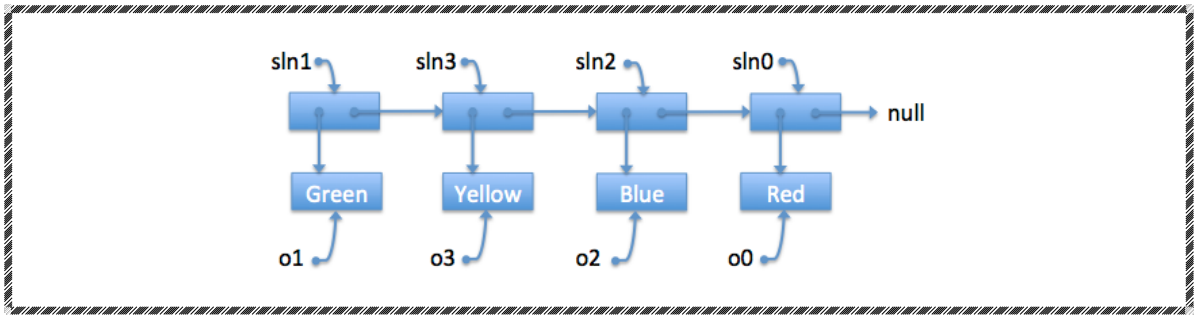
Homework #11
Abstract Data Types
Solutions

1. Consider the following lines of code which create several `SinglyLinkedNode` objects (see `comp132.examples.adt.LinkedListExamples` for the definition of `SinglyLinkedNode`):

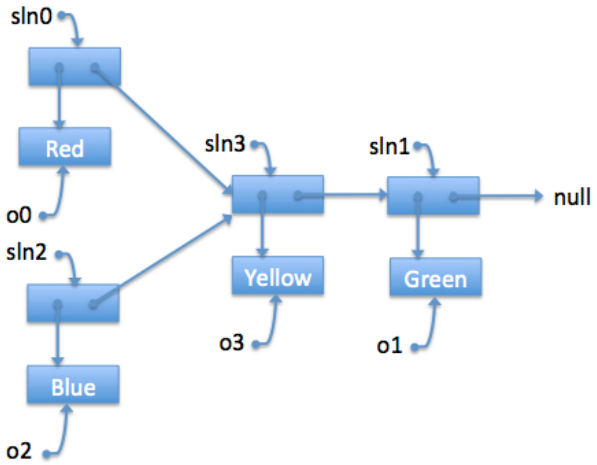
```
String o0 = "Red";
String o1 = "Green";
String o2 = "Blue";
String o3 = "Yellow";

SinglyLinkedNode sln0 = new SinglyLinkedNode(o0);
SinglyLinkedNode sln1 = new SinglyLinkedNode(o1);
SinglyLinkedNode sln2 = new SinglyLinkedNode(o2);
SinglyLinkedNode sln3 = new SinglyLinkedNode(o3);
```

Draw the linked list that would be produced by the following snippets of code:

a.
```
sln1.next = sln3;
sln2.next = sln0;
sln3.next = sln2;
```



b.
```
sln0.next = sln3;
sln2.next = sln3;
sln3.next = sln1;
```



2. Give the implementation of each of the following methods as they would appear if you were to add them to the `LinkedListExamples` class in the `comp132.examples.adt` package. Implement each method by directly manipulating the list (i.e. don't call the `getNode`, `insertNode` or `removeNode` methods that we wrote in class). You can test your implementations by building lists in `main` and then calling your methods, possibly followed by `printList` to ensure that the list has changed appropriately.

a. `countNodesAfter(SinglyLinkedNode node)`: return the number of nodes that come after `node` in the list. (+1 bonus point for doing it recursively).

```
public int countNodesAfter(SinglyLinkedNode node) {
  int count = 0;
  SinglyLinkedNode cur = node;
  while(cur.next != null) {
    cur = cur.next;
    count++;
  }
  return count;
}
```

b. `swap(SinglyLinkedNode pred1, SinglyLinkedNode pred2)`: Swap the node following `pred1` with the node following `pred2`. Move the actual nodes by manipulating their `next` references, do not simply swap the elements contained in the nodes. You may assume that there is at least one node between `pred1` and `pred2`. You may also assume that there are at least two nodes following `pred2`.

```
public void swap(SinglyLinkedNode pred1, SinglyLinkedNode pred2) {
  SinglyLinkedNode sw1 = pred1.next;    // sw1 is node to swap.
  SinglyLinkedNode sw2 = pred2.next;    // sw2 is node to swap.

  SinglyLinkedNode sw2n = sw1.next;     // will be after sw2.
  SinglyLinkedNode sw1n = sw2.next;     // will be after sw1.

  pred1.next = sw2;   // move sw2 into sw1's spot
  sw2.next = sw2n;

  pred2.next = sw1;   // move sw1 into sw2's spot
  sw1.next = sw1n;
}
```

NOTE: Drawing a picture can help a lot with understanding this type of question!

c. `jumpByN(SinglyLinkedNode pred, int n)`: Move the node following `pred` down the list by jumping it over n successive nodes. Move the actual nodes by manipulating their `next` references, do not simply move the elements contained in the nodes. For example, if `pred` refers to the node at index 2 in the list and n = 5, then the node at index 3 in the list will jump over 5 other nodes (those at indices 4,5,6,7 and 8) and will appear following the node at index 8. You may assume that n is greater than 0 and there are at least `n+1` nodes following `pred`.

```
public void jumpByN(SinglyLinkedNode pred, int n) {
  // Remove the node to be shifted from the list.
  // tmp will refer to the node being shifted.
  SinglyLinkedNode tmp = pred.next;
  pred.next = tmp.next;

  // Go down the list to the new predecessor of tmp.
  // newPred will refer to the new predecessor for tmp.
  SinglyLinkedNode newPred = pred;
  for (int i=1; i<=n; i++) {
    newPred = newPred.next;
  }

  // Insert tmp following newPred.
  tmp.next = newPred.next;
  newPred.next = tmp;
}
```

NOTE: Drawing a picture can help a lot with understanding this type of question!

3. For each of the following applications indicate if an array-based list or a linked list will result in a faster program. Briefly explain your answer incorporating the Big-O bounds for the operations.

a. An application that models customers waiting in a line: Customer objects are inserted at the back of the line and removed from the front of the line when it is their turn. There is no fixed limit on the length of the line.

This application would be best implemented using a **Linked List**. Inserting at the front and removing from the end of a linked list are both O(1) operations. Inserting or removing at the front of an Array List is a O(n) operation and thus would be significantly slower for this application.

b. An electronic messaging system: The system maintains a list of all of the user accounts on the system. The list is maintained in alphabetical order by user name. Arriving messages are addressed by user name. When a message arrives the account for the recipient user is found and the message is added to the account.

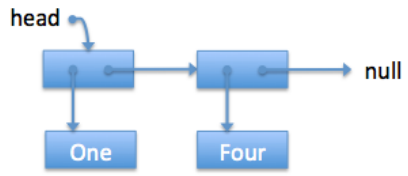
This application would be best implemented using an **Array List**. Because an array list provides O(1) access to any element, we can use binary search to find the recipient's account in O(lg n) time. With a Linked List, the recipient's account could only be found using a linear search in O(n) time. Thus, an Array List would be significantly faster than a Linked List in this application.

4. Consider the following snippet of code that creates a `java.util.LinkedList`:

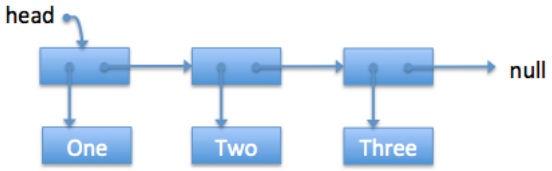
```
LinkedList<String> list = new LinkedList<String>();
list.add("One");
list.add("Two");
list.add("Three");
list.add("Four");
```

Show the contents of the linked list after each of the following snippets that use an `Iterator` is executed. Assume that the full list is recreated before each snippet is executed.

a.
```
Iterator<String> it = list.iterator();
it.next();
it.next();
it.remove();
it.next();
it.remove();
```



b.
```
Iterator<String> it = list.iterator();
while(it.hasNext()) {
   it.next();
}
it.remove();
```



5. Consider the following list:

"A" → "B" → "C" → "D" → "E" → "F" → "G"

Assuming the variable `list` contains a reference to the above list, and that the variable `it` contains a reference to an `Iterator` located at the beginning of `list`, give a sequence of calls to `it.next()` and `it.remove()` that would transform list into the one shown below.

"A" → "C" → "D → "G"

```
Iterator<String> it = list.iterator();
it.next();    // Pass A
it.next();    // Pass B
it.remove();  // Remove B
it.next();    // Pass C
it.next();    // Pass D
it.next();    // Pass E
it.remove();  // Remove E
it.next();    // Pass F
it.remove();  // Remove F
```

6. Write a method named `removeEveryOtherElement` that accepts a `List<String>` as a parameter. This method should remove every other element from the list (i.e. the elements at index 1, 3, 5, … should be removed). The list may have any number of elements, including 0 or 1. For full credit your method must use an `Iterator` to traverse the list and to remove the elements. You may want to write and test this method in the `IteratorExamples` class provided in the `comp132.examples.adt` package of the `132SampleCode` project.

```
public static void removeEveryOtherElement(List<String> list) {
  Iterator<String> it = list.iterator();
  while(it.hasNext()) {
    it.next(); // Skip element 0, 2, 4,…
    if (it.hasNext()) {
      it.next();   // Pass element 1, 3, 5,…
      it.remove(); // Remove element 1, 3, 5,…
    }
  }
}
```