Announcements

- Added comments to your program 3 scores
  - Please put author documentation
  - Number one reason for missing points was poor variable names
- Program 4: Due 11/9 at 11:59 PM
  - Will be released later tonight
  - It will be **very difficult** so start early
  - Extra credit: early turn in before 11/2 at 11:59 will be given an extra 15% (of the score they earned, including the 10% that I grade) towards the program 4 grade. If you submit the assignment after 11:59 on 11/2 then you are no longer eligible for this extra credit.
- Midterm 2 is on 11/6
  - Next week I will release information about it

Quiz Time ;)

Linked Lists (cont.)

- The **insertAt** operation for a singly-linked list inserts the new node at a provided index. (java tends to call this operation add(int index, E element))
- The element at the desired index and subsequent indices get “shifted” over
- Note: this is different from zybooks which talks about “InsertAfter” which inserts after a specified node, the two operation are very similar.
- There are now three scenarios we have to worry about when doing this operation: empty list, insert at lists tail node, insert in the middle of the list

Linked Lists (cont.)

- Insert as the list's first node: if the list's head pointer is null, and the index to insert at is zero, the list's head and tail should then point to (i.e reference) the new node
- Insert at the tail node: If the list's head pointer is not null (i.e the list is not empty) AND the index is the same as the Tail pointers current index, then we must set the Tail pointers previous nodes next to reference the new node, and the new node's next pointer to reference the tail.
    - To do this assume we have access to the previous node, let us call it prevNode.

Linked Lists (cont.)

- Insert in the middle of the list: if the list's head pointer is not null, set the new node's "next" pointer to reference the previous indexes nodes "next" reference, THEN set the "next" pointer at the previous indexes node to reference the new node.
    - To do this let us assume we have access to the previous node and call it: prevNode.

Let's look at an example...

Linked Lists (cont.)

- The RemoveAt operation removes a node at a specific index location (this is similar to java's "remove(int index)" method.
- Note this is different from zybooks: RemoveAfter, which refers to removing a node after a specified node.
- There are now four scenarios we have to worry about when doing this operation: empty list, remove at the lists head node, remove at the lists tail node, remove in the middle of the list

Linked Lists (cont.)

- Remove from an empty list: If the head and tail nodes point to null then the list is empty, therefore there is nothing to remove
- Remove at the lists head node: If the index to remove is zero, then the **head** node is the desired node to remove. You do this by setting the head reference to the head's next pointer.
    - What happens to the old head node in memory? Garbage collection because now nothing references it.

Let's look at an example...

Linked Lists (cont.)

- Remove at the lists tail node: if the index to remove is equal to the length of the list minus 1, then the desired node to remove is the **tail node**. To do this let us assume we have a reference to the node preceding the tail node, called **prevNode**.
  - Set the tail node reference to be prevNode, then set prevNode's "next" pointer to reference Null.
  - What happens to the old tail node? Garbage collection because nothing now references it

Let's look at an example...

Linked Lists (cont.)

- Remove in the middle of the list: if the index to remove is a valid location in the middle of the list, then we need access to two different nodes: the node to remove, and the node preceding the node to remove.
  - Assume the node to remove is called **nodeToRemove**, and the preceding node is called **prevNode**.
  - Set prevNode's "next" pointer to reference the nodeToRemove's "next" pointer, then set nodeToRemove to reference Null

Let's look at an example...

Collections

A brief aside: Collections.

How do we know what operations (methods) a data structure should include?

Java's answer: the Collections Framework

- A collection represents a group of objects, known as elements
- The Collections Framework defines the starting point for data structures that will be working on a collection (Which is just about every data structure).

Let's take a look at Java's Collections Framework...

Collections (cont.)

The **List** interface defined within the Java Collections Framework, represents a collection of ordered elements, i.e. a sequence, the base set of operations (methods) a List data structure should have.

A **Linked List** is one of several data structures that implements the List interface.

Let’s look at the List interface and the Linked List class in Java’s documentation...

We will comeback to Java’s Collections Framework soon (there are other useful data structures that it contains other than a linked list), but for let us continue with Linked Lists

Linked Lists (cont.)

In the: **InsertAt** and **RemoveAt** operations, a key proponent that made our lives easy was having access to the **previous node**.

But accessing such a node is not free let us look at what it takes to traverse a linked list and see the cost for finding a node.

But first let us introduce the concept of an **Algorithm**...

Algorithms

An algorithm describes a sequence of steps to solve a computational problem or perform a calculation.

A computational problem specifies:

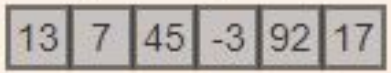
- An input
- A question about the input
- A desired output

Algorithms (cont.)

Example:

Computational problem:

Input: Array of integers



Question: What is the maximum value in the input array?

Output: Maximum integer in input array

92

Algorithm:

```
FindMax(inputArray) {
   max = inputArray[0]

   for (i = 1; i < inputArray.size; ++i) {
      if (inputArray[i] > max) {
         max = inputArray[i]
      }
   }

   return max
}
```

Linked Lists (cont.)

Linked List traversal algorithm: visits all nodes in the list once and performs an operation on each node.

- A common traversal operation prints all list nodes.
- The computational problem:
  - Input: Linked List
  - Question: Can all of the nodes in a linked list be visited?
  - Desired Output: Print all of the nodes in the linked list.

Linked Lists (cont.)

- The algorithm starts by assigning a node reference called: **curNode**, to the list's head node.
- Then loop through all of the elements in the linked list by setting the curNode reference to the curNode's "next" pointer, printing the elements as the linked list is traversed.
- The stopping condition for this algorithm is when curNode eventually becomes null, in which case the end of the list has been reached.

Linked Lists (cont.)

Example (pseudo code):

```
curNode = Head;

while ( curNode != null ) {
        print ( curNode.data );
        curNode = curNode.next;
}
```

Let’s look at an example on the board...

Linked Lists (cont.)

Now that we have a basic understanding of a linked list let us try to make one on the board together...