Chapter 02, 2.7 – Intro to Linked Lists, Review Problem(s): ***38, 39***

Chapter 02, 2.8 – List Based Stack, Review Problem(s): ***43, 45, 46, 48***

Chapter 02, 2.9 – Postfix Evaluator, Review Problem(s): ***50, 51***

Chapter 02, 2.10 – Stack Variations

The textbook presents a linked-list-based implementations of a stack.

There are differences between the array-based and the linked-list implementations:

- Memory use
- Manner of access
- Efficiency of various operations
- Language support

In a linked-list implementation, as we said previously, the methods which need to be created to implement a stack are the same:

- isEmpty
- isFull
- push
- pop
- top

But first, let's consider a linked-list. A node in a linked list is an object that holds:

1. Data
2. A link to a node

This link to a node, like itself, makes the node a ***self-referential class***, which is a class that contains instance variable(s) that can hold references to object(s) of the same class.

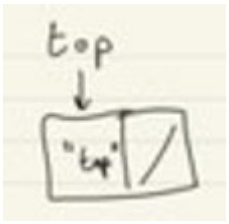
Pages 48-51 of Chapter 02 PPT.

Show the definition for the `LLNode` class in Chapter 02 PPT

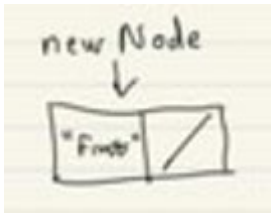
Let's build a linked-list using the `LLNode` class.

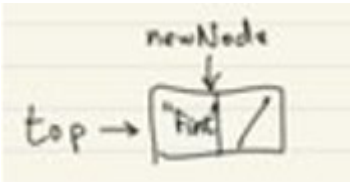
Create a new JavaProject, Download the files from Moodle for the Linked List, and import these files into your Java Project (***we first did this for Lab 06, the detailed instructions are there, please be sure to import into your /src subfolder***).

Note that these files are from the textbook code, but they have been "flattened," since they are all in the same directory, so all Java commands referring to packages and importing have been removed.



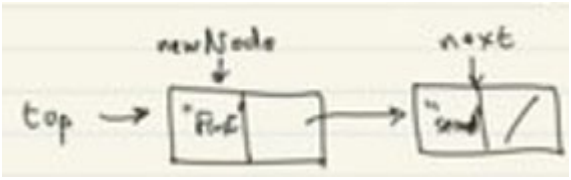

```
// the pointer to the start of the list, top, is null initially
LLNode<String> top = new LLNode<String>("top");

// declare and instantiate the first LLNode in the list
LLNode<String> newNode = new LLNode<String>("first");

// assign top to be this newNode
// note that the old top will be garbage
top = newNode;
```






```
// declare and instantiate the second LLNode in the list
LLNode<String> next = new LLNode<String>("second");
```


```
// make top, the first node, point to this
next LLNode
top.setLink(next);
```




```
// traverse the linked list "manually"
System.out.println("manual traversal: "+top.getInfo() + " " +
top.getLink().getInfo());

// after automating below, come back up here and add another LLNode to
the list, after next
LLNode<String> next2 = new LLNode<String>("third");
// set the new link
top.getLink().setLink(next2);

// automate traversing the linked list
// create a temporary node, called currNode to do this
LLNode<String> currNode = top;
while (currNode != null)
{
      System.out.println("auto traversal: "+currNode.getInfo());
      currNode = currNode.getLink();
}
```

Continue with Chapter 02 PPT, tracing a traversal and inserting into a linked list, pg. 52-61.

Next, implementing the Stack ADT with a linked list, Chapter 02 PPT, pg. 62-75.

The following discussion, of using a Stack to do evaluation of a Postfix expression, Chapter 02 PPT, pg. 62-75.

Infix to Postfix (not in textbook):

This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ')'. We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

1. Create an empty stack and an empty postfix output string/stream
2. Scan the infix input string/stream left to right
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. The order of popping is the order in the output.
5. If the current input token is '(', push it onto the stack
6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the output string.

This algorithm doesn't handle errors in the input, although careful analysis of parenthesis or lack of parenthesis could point to such error determination.