**IFT194 – Lab2 – 3/03/18 – Rick Bird**

The first task was to understand exactly how the custom made linked list worked. It was somewhat confusing at first, but at its base, a linked list is made of nodes, which are subclasses of the linked list. Each node points to its following node with node.next (unlike a doubly linked list which has previous and next). Each linked list has a front which is the first node in the list. When removing or adding a node, it is important to consider which node is being analyzed, if its next is null, or there is only 1 item in the list. The ability to change the front node to front’s next node is also useful in removing from the front.

Next was the first task in adding to the linked list’s functions: returning the length. This was as simple as creating a for loop with a placeholder node as an iterator, n. If n.next != null; n = n.next as the loop iterates through the linked list. After each iteration, the size is increased by 1. Before the loop even occurs for the first time, it checks to see if the front node is null. If It is then the size is 0. Immediately after this check, if the node isn’t null, the size is incremented by 1 because we know that there is at least one item in the list. It then proceeds to the for loop. After that it simply returns the method’s size integer.

toString was a little bit more challenging. Initially I wanted to create an array of the nodes’ values and simply use the build in print array function, but for some reason it would print the first integer correctly, but the rest would be zero.

public String toString()

{

//make an array of integers that is the LENGTH of this list

int[] toStringArray = new int [this.length()];

int iterator = 0;

IntNode temp = front;

while (temp != null)

{

toStringArray[iterator] = temp.val;

temp = temp.next;

}

return Arrays.toString(toStringArray);

}

I then realized that toString could be achieved by doing something similar to the preexisting print function. All I needed to do was initialize and empty string, and use a temporary node (I realized this functioned identically to my for loop idea) to iterate through the list while concatenating each of temp’s current values to the initially empty string.

removeLast was the most complicated part of this assignment, mostly because there are a lot of special cases when dealing with the last element of the linked list. First, we had to consider if the front of the list didn’t have anything, we do not do anything at all. Next was the simple case of if there is only one item in the list, we set that item to null to create an empty list. Next we have to create two separate iterators to sort of “inch through” the list. Parent and child are a constant duo of nodes with parent always pointing to child. If the item after child is null, that means child must be the last element of the list, and must be set to null. Otherwise, the parent shifts over by 1 item in the list by setting itself equal to the child, while the child becomes the parent’s new next, or in other words the child’s next node. This is done within a loop.

Replace was simple after understanding the concept of creating an iterator node that points to an existing node. The iterator node, temp, initially points to the front of the list. We use the size of the list as an end point in the search for integers to replace. If temp’s value ever matches the parameter oldVal, the value being searched for, it is changed to newVal. Afterwards, we set temp to point to the next node in the list.

The parenthesis match taught me more about how a stack functions. At it’s core, parenthesis match iterates through a string that is input by the user. It pushes back the current character to the stack ONLY IF it is a ‘(‘ or else it pops off a ‘(‘ if a matching ‘)’ is found. The complicated part was implementing a way to check for errors if there was a mismatched left or right parenthesis. We would know there are mismatched left parentheses if the stack still had some left parenthesis after doing every push back operation. However for right parenthesis, if the current character is a ‘)’ and the size of the stack is 0, that means that there is no matching ‘(‘ so an error is returned. However, this is not a void method. This is a part of main. So I created a global variable (bad I know) bool called error that gets set to true if there is a mismatched right parenthesis. If it is true, after the for loop that checks for right parenthesis mismatches, if the error is still set to its initial value of false, then a message is printed to show that all parenthesis have matches.

After this lab, I think I understand the concept of collections a little better. I understood arrays already for the most part, but I can now see the practical application of having a list with pointers leading to their next value and how a stack functions. The lab took me about 4 hours to complete, with the only major setback being the removeLast method in the first part as well as having to redo my toString method, although once I figured out the trick of using parts of the print method for this, it became much simpler. The most important thing I learned is that you can create a placeholder object as an iterator. You can set it equal to an existing object, and that means both the original object and the placeholder object point to the same value. If the placeholder is edited, that means the original is also edited. I know there must be practical applications of this beyond simply the linked list, and it is probably used frequently in many built in java classes.