**Introduction:**

"Imagine a (literal) stack of plates. If the stack gets too high, it might topple. There-fore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure SetOfStacks that mimics this. SetOf-Stacks should be composed of several stacks and should create a new stack once the previous one exceeds capacity. SetOfStacks.push() and SetOfStacks.pop() should behave identically to a single stack (that is, pop() should return the same values as it would if there were just a single stack).   
FOLLOW UP  
Implement a function popAt(int index) which performs a pop operation on a specific sub-stack."

**FirstAttempt**:

- We created two versions of a SetOfStacks class to demonstrate what an outer stack could look like.  
-First was an ArrayBoundedSetOfStacks, and the second was a LinkedListSetOfStacks.

-Both iterations worked fine for using the push method, the pop method, and the top method, because the three traditional stack methods only need reference to the top element to function. If we wanted to create a popAt method however, we’d need be able to reference any index in the entirety of the stack.

**Walk Through 1st Demo Program**  
  
**PopAt Dilemma:**

- A good way to find an element at a specific index is to use the modulus operation. To identify which InnerStack contains the desired index we use integer division to divide the index by the how many elements are in each individual stack, and the result gives us the correct index of the InnerStack in the OuterStack. Then we would use the modulus operation to find the index of the element within the InnerStack.   
- However, once we remove our element the number of elements in each inner stack becomes inconsistent. This means we can no longer divide the desired index by how many elements are in each individual stack to find the correct index of the InnerStack in the OuterStack.   
- To call popAt a second time this way would require us to shift all the elements above the one we just removed and condense the inner stacks to be uniform at full capacity. This is seriously inefficient and is why we chose to go in another direction: the double linked list.

**How A DLL Overcomes This:**

-We use an accumulator variable, and a next accumulator variable to keep track of how many elements we have passed, and how many we will have passed if we have gone from one index to the next.   
-When the nextAccumulator is greater than the index we are searching for, we know we are in the correct stack.   
-Because we are using a doubly linked list, we immediately have access to both the top and bottom of the stack. This also allows us to retain O(1) access to the top element while still being able to seach from the bottom during the popAt method.   
**Walk Through 2nd Program**