Implementing Stacks With Linked Lists

- Overview: The LinkedStack Class from L&C
- The Fields and Constructors
- The push Method
- The pop Method
- The Other Methods
- L&C’s Version of the Maze Search
Overview: The **LinkedStack** Class From L&C

- Now that we know how linear linked structures work, we can see that they are well suited to implement a stack.

- We only need to change pointers on elements near the top to push or pop, giving us $O(1)$ time for these operations in the worst case. (The ArrayStack needed $O(n)$ time to resize on some pushes, though these operations only took $O(1)$ time per push on average.)

- We'll use the `LinearNode` class we defined last lecture, where a node points to the next node and points to its contents, from some class $T$.

- The ArrayStack already kept track of its size through the field `top`, but we will need a field to keep the size of our stack explicitly.
The Fields and Constructors

- We have two fields, the more important of which is the pointer that starts the linear structure. We also keep track of the number of elements in the stack.

- Note that constructors for the generic class do not have the "<T>" in their name when they are declared, even if they depend on T (as this one doesn’t). I got this wrong earlier. But at any given time there can be only one type version of a generic around, because two classes can’t have the same name in the same scope.

```java
public class LinkedStack<T> implements StackADT<T>{
    private int count;
    private LinearNode<T> top;

    public LinkedStack( ) {
        count = 0;
        top = null;
    }
```
The push Method

- The basic idea is simple -- we create a new node with the contents provided, then link it into the top of the stack and update the size.

- Although the constructor had no "<T>" when we defined it, it has one now when we are calling it -- the compiler needs to know that we are creating something that can fit into a variable of the type LinearNode<T>.

- This is of course an O(1) time operation -- we have no idea how big the stack might be when we do this.

```
public void push (T element) {
    LinearNode<T> temp = new LinearNode<T> (element);
    temp.setNext(top);
    top = temp;
    count ++;
}
```
The **pop** Method

- To pop, all we need to do is save a pointer to the top element, reset the top pointer to bypass that element, and update the size counter.

- But if the stack happens to be empty we need to throw the exception, and include a `throws` clause to let this be handled in the calling method if desired.

```java
public T pop ( ) throws EmptyCollectionException {
    if (isEmpty( ))
        throw new EmptyCollectionException(“Stack”);
    T result = top.getElement( );
    top = top.getNext( );
    count--; 
    return result;}
```
The Other Methods

- The rest of the five basic StackADT methods are simple and also O(1) time.

- The `toString()` method would naturally take O(n) time, as we want to print something for each element of the stack -- exactly what would be a style decision but it would include `getElement().toString()` for each node in turn. Clearly we would want to use only peeks, not pushes or pops.

```java
public T peek () throws EmptyCollectionException {
    if (isEmpty( ))
        throw new EmptyCollectionException("Stack");
    return top.getElement( );}

public boolean isEmpty ( ) {return (top == null);}

public int size ( ) {return count;}
```
L&C’s Version of the Maze Search

• In Section 4.5 L&C create a Maze class where the entries of the two-dimensional array are Integer objects, holding numbers that are code for our “open” and “seen” boolean fields.

• They put Position objects on the stack, where a Position is an (x, y) pair.

• But they only search for paths from the top left to the bottom right.

• Note “StackADT<Position> = new LinkedStack<Position>( )” -- they keep the stack in a StackADT<Position> variable so that the rest of the code doesn’t care which implementation is used. We could replace LinkedStack<Position> with ArrayStack<Position> in this one line and the rest of the code would work perfectly well with the new implementation. Our use of Stack< SCEll> committed us to arrays.