CMPSCI 187: Programming With Data Structures

Lecture #13: A Linked Stack Implementation
David Mix Barrington
3 October 2012

A Linked Stack Implementation

- Generic Linked Lists: `LLNode`
- Data Fields and Constructors for `LinkedStack`
- Transformers for LinkedStack
- Observers for LinkedStack
- Comparing the Implementations: Running Time
- Comparing the Implementations: Memory Usage

Generic Linked Lists: `LLNode`

- For our `LinkedStringLog` class, we first defined a class `LLStringNode` whose objects were the nodes of the linked list. Each `LLStringNode` object had an `info` field containing a `String` and a link field "containing" an `LLStringNode`. (Of course this is really a **pointer** to the next node.)

- To make a generic class `LinkedStack<T>`, we will need a similar class `LLNode<T>`, whose objects will contain a `T` object and point to another `LLNode<T>`. We don't have the business about casting in our constructor because we are not trying to build an array out of generic `T` objects.

```
public class LLNode<T> {
   private LLNode link;
   private T info;
   public LLNode(T info) {
      this.info = info;}
   // two getters, two setters
}
```

Data Fields and Constructors for `LinkedStack`

- Now the code for `LinkedStack<T>` is very simple. Our only data field is the pointer to the top of the stack -- it says it is just a node but it implicitly includes the entire linked list since we can reach that by traversing.

- Note that constructors for generic classes do *not* include the "`<T>`" in their name, but just the name of the class.

- The data fields of `LLNode<T>` were private, because we know that the only methods that should ever access them are the getters and setters. Here, a future class that might extend `LinkedStack<T>` should be able to get at `top`.

```
public class LinkedStack<T> implements
                              UnboundedStackInterface<T> {
   protected LLNode top;
   public LinkedStack( ) {
      top = null;}
```

Transformers for `LinkedStack`

- As will `ArrayStack`, we can change the contents of the stack in two ways, with the `push` or `pop` method. Again remember that DJW have separated the transformer pop from the observer top, unlike the `java.util.stack` class.

- The `push` method calls a generic constructor, and has to indicate the value of the type variable. It never throws an exception as the stack can't get full.

- The `pop` method just bypasses the top element if it exists.

```
public void push (T element) {
   LLNode<T> newNode = new LLNode<T>(element);
   newNode.setLink(top);
   top = newNode;}

public void pop( ) {
   if (!isEmpty( ))
      top = top.getLink( );
   else throw new StackUnderflowException("pop from empty");}
```

Observers for LinkedStack

- You might think that the `top` method might be as simple as "`return top`", but there are two complications. The object top is a node, and we need to return its contents rather than just itself. Also, we must check for an empty stack. The `isEmpty` method *is* as simple as checking if `top == null`.

- DJW don't include `size` as a standard stack observer. How would we compute and return the number of elements in the stack?

```
public T top( ) {
   if (!isEmpty( ))
      return top.getInfo( );
   else throw new StackUnderflowException("top of empty stack");}

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

Comparing the Implementations: Running Time

- Will any of our methods take longer for a stack with many elements than for a stack with few elements? In other words, if N is the number of elements in the stack, what is the asymptotic running time of each of our methods as a function of N? Does it make a difference which implementation we use?

- In fact the `push`, `pop`, `top`, and `isEmpty` methods take O(1) time in each case. In `ArrayStack`, pushing and popping each involve moving one element of the array and changing topIndex, while in `LinkedStack`, we only need a few pointer operations around the top node.

- The constructor for `LinkedStack` takes O(1) time, while that for `ArrayStack` takes time proportional to the *capacity* of the stack.

- It's more interesting to compare the times for implementations of the size method. As is, `ArrayStack` takes O(1) and `LinkedStack` takes O(N). But we could make `LinkedStack` O(1) by adding a size field and always updating it.

Comparing the Implementations: Memory Usage

- If a stack has N elements in it, all of some class T, then it must use the memory for N objects of class T, whatever it is. Depending on the implementation, we may be using more memory. Both lists and arrays have a pointer to each object, at least.

- An array uses a number of pointers equal to its capacity (length of array) rather than to the size (number of elements). This could be significant extra memory use if the array is much bigger. Linked lists only allocate enough memory for the nodes that exist, but note that there may be some overhead to making an object out of the pointer and the T element. Still, the space use for N stack elements is certainly O(N).

- Array operations are often faster than list operations for reasons we aren't well equipped to analyze. Not all memory is equal, and there is a greater chance that list elements are stored in memory that takes longer to access.

Searching With a Stack

- In Project #2 we try to find a solution to a Sudoku puzzle by **backtrack search**, making provisional guesses and withdrawing them if they turn out to lead to invalid boards. We use a stack to keep track of the moves we have made and not yet unmade -- this works because we only ever unmake the latest move that we made.

- This is an example of **depth-first search** (see XKCD #761). We'll look at this in much more detail later, but for now consider the problem of trying to find a path from where we are to some goal in a **graph**, a set of points with **edges** allowing us to go from one point to another. In a DFS, we consider one option at a time out of each point, "untaking" that option if and when we find that it cannot lead to the goal. Again, we can use a stack to record the options we are currently considering, because we only unmake the latest among the choices we have made.

- In Project #2 we are searching a graph that we cannot draw, only explore -- the tree of choice sequences that have so far led to a valid board.