Notes to Accompany Array Stack Implementation

The handout you have includes three files:

1) stack.h
2) stack.c
3) revword.c

As you might imagine, stack.h and stack.c go together. The first provides a list of the functions that pertain to using a stack.

Note that I call my struct stackDT instead of stackADT as the text has done. The reason for this is that in my implementation of a stack, the storage of the data is NOT hidden. Ideally, one could look at the .h listing of the functions, not KNOW how the data was stored, and still use all these functions. However, this information can be ascertained by looking at stack.h

Notice however, that all of these functions can be used without the user directly manipulating the data. All manipulations occur through the functions outlined in stack.h. This is the essence of an abstract data type. It is one that the user does NOT need to know the specific manner in which data is stored in order to use it effectively. Knowing how to use the structure comes from understanding the functions that are provided to manipulate it. All one has to understand is WHAT the functions do, not HOW they accomplish the feat.

stack.h

This file contains the struct used to store a stack, a constant definition, and the function headers of all the functions that will operate on a stack. Each of these very closely mirrors the functions discussed in the text. My implementation stores characters as can be seen by the Push function.

Each function always takes in a stack by reference so that changes made to the stack in the function are reflected in the calling function.

Just using the information in stack.h, it is possible to write a new program that USES a stack without knowing exactly how the functions pertaining to the stack were written.

revword.c

This simple application reads in a word from the user, and then uses a stack to aid it print out the word read in, in reverse order. The characters of the input string are pushed onto a stack created in the main function, one-by-one starting from the beginning of the string. Then, after these are all pushed on the stack, each item gets popped off the stack, one-by-one and printed out as they are popped. In order to do these tasks, the following functions were necessary:

NewStack, Push, Pop, StackIsEmpty

stack.c

The actual implementation of a stack using an array isn't too difficult. Here are the components we need to store:

1) The array storing the elements.
2) An index to the top of the stack (actually the element AFTER the top of the stack.) We'll assume the bottom of the stack is index 0, and go from there.

When we create a new stack, we simply need to get space for the object and then set top to 0, to indicate that the stack is empty.

Quick Question: Why didn't we do this?

```
struct stackDT s;
s.top = 0;
return &s;
```

To push an element, we must simply copy it into the next location for the top of the stack and then adjust top.

To pop, we must save the value at the top of the stack, decrement top, and then return the saved element.

The way I have implemented this stack, top is already the depth of the stack.

Checking to see whether the stack is full or empty is reasonably easy since top keeps track of the number of elements currently in a stack.

How to Use Dynamically Allocated Arrays for Stacks

In the implementation of a stack we discussed in class last time, we were forced to not allow more than 20 elements onto the stack. However, it's mostly likely that there would have been more memory available in the computer on which the program was running. Furthermore, we already KNOW how to dynamically allocate memory for an array.

Thus, rather that saying that our stack is full when our array is, we can simply dynamically "grow" our array!

Let's consider how we can do this:

Given an array values that is already full and stores length number of elements, when we want to add newval to the end of the array, we can do the following:

1) Dynamically create a new array temp, that has more space that values.
2) Copy each element in values into temp, one by one.
3) Add the new element that didn't fit in values into a remaining open slot in temp.
4) Deallocate the memory for values.
5) Assign the pointer values to temp.

Here is a code segment that takes care of this task:

```
int *temp = (int *)malloc((length+1)*sizeof(int));
for (i=0; i<length; i++)
   temp[i] = values[i];
temp[i] = newval;
delete [] values;
values = temp;
```

This will work, but do you notice any potential weaknesses here?

What would happen if we had to add one more element into this array?

How much time would it take for each addition in terms of the total number of elements currently in the array?

Clearly, if we just extend or "grow" the array by one element each time, we have LOTS of work in front of us for each addition into the array. Instead, here's a better strategy:

Whenever you expand the array, double its size.

It turns out that this rule leads to great performance in the long run. Instead of adding an element taking O(n) time where n is the total number of elements, adding an element takes O(1) on average.

In order not to waste huge amounts of memory, it is also advisable to "shrink" this type of array by half when the array is less than one quarter full. Once again, it can be shown that this procedure leads to efficient average case running times.

These ideas can easily be incorporated into a stack class so that the size of the stack is not so limiting. Even so, a StackIsFull function is advisable since sometimes a malloc call may return NULL.

Linked List Implementation of a Stack

We can essentially use a standard linked list to simulate a stack, where a push is simply designated as inserting into the front of the linked list, and a pop would be deleting the front node in a linked list.

There are multiple ways to achieve this:

1) Create just one struct for the stack which essentially acts similar to the struct defined for use with linked lists.

2) Use an existing linked list data structure, and simply borrow an instance of that, creating a new struct that has a pointer to a linked list as a component.

In my example, I chose the former method. A good exercise for you would be to see if you can adapt my code to work the other way, using the linked list code we have already gone over.

Some issues to think about when writing this code:

1) Making sure changes get reflected in the original stack struct. (This is done using **.) The other way would be to return a pointer.

2) Checking to see if memory was allocated properly for a new node.

Most of the rest of the code (even though it looks long), is reasonably straight-forward.

Exercise to use the Stack Data Structure Presented

Plan the outline to a program that will read in a postfix expression and evaluate it. For this exercise, assume that the postfix expression read in is valid.

If you come up with a good outline, start filling in some of the implementation details.

For your outline, assume you have a function that reads in each "token" as a string, and automatically converts it to either an integer or an operator (which is a character), and you can simply proceed based upon this assumption.