Hyo Lee
Week 2 Homework Submission
01/19/2016

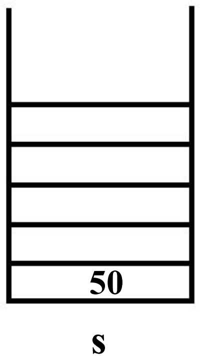
CST 370
Homework (Stacks)

1. Consider two stacks each of size 10. When you pop an element from the first stack you multiply it by 2 and add it to the second stack. When you pop an element from the second stack you multiply it by 3 and add it to the first stack.

Push numbers 1 to 5 to the first stack in order. Push numbers 6 to 10 to the second stack in order. Pop two numbers from the first stack (remember these numbers are going to be added to the second stack). Pop three numbers from the second stack (remember these numbers are going to be added to the first stack).
a) What is the value in the top of the first stack? = **30**
b) What is the value at the top of the second stack? = **9**
c) What is the current size of the first stack? = **6**
d) What is the current size of the second stack? = **4**

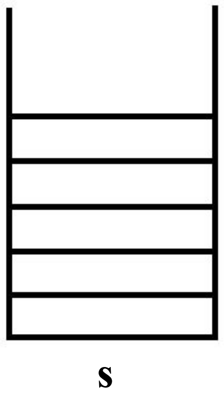
2. Draw the stack after the following code executes

```
Stack s;
s.push(50);
s. push (25);
s.push(60);
s.pop();
s.pop ();
```



3. Draw the stack after the following code executes

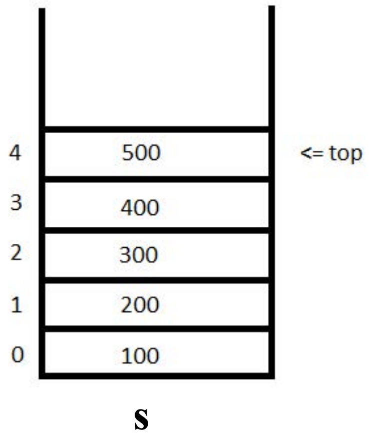
```
Stack s;
s.push (22);
s.push (33);
s.push (44);
while (!s.empty())
        s.pop();
```



Empty stack

4. Draw the stack after the following code executes

```
Stack s;
for (int i = 1; i <= 5; i++)
        s. push (100*i);
```



5. Suppose that some application requires using two stacks whose elements are of the same type. A natural storage structure of such a two-stack data type would consist of two arrays and two top pointers. Explain why this may not be a space- wise efficient implementation.

This implementation may not be efficient space-wise if one, or worse both arrays are allocated with a capacity that is much too large than what is needed to store the data elements. Since two stacks are available with the same type, the two stacks could share a single array with two pointers so that the single array can be used and filled by either stack reducing the possibility that one stack completely fills up its own array while the other stack's array contains wasted empty space that instead could be used by the other stack. Furthermore, while one array is nearly empty wasting space, if the other array fills up, it will need an additional array to compensate for overflow further wasting more space.

6. The last element added to a stack is the **first** one removed. This behavior is known as maintaining the elements in **LIFO (Last In First Out)** order.