**Stacks**

The operations on the stack give it a last in, first out behavior. Two of the several applications of a stack that we consider are evaluating algebraic expressions and searching for a path between two points. Finally, we discuss the important relationship between stacks and recursion.

**1. The Abstract Data Type Stack**  
 Developing an ADT during the Design of a Solution

Read & Correct  
When you type a line of text on a keyboard, you are likely to make mistakes. If you use the backspace key to correct these mistakes, each backspace erases the previous character entered. Consecutive backspaces are applied in sequence and so erase several characters. For instance, if you type the line  
abcc<-ddde<-<-<-ef<-fg  
where <- represents the backspace character, the corrected input would be   
abcdefg  
  
How can a program read the original line and get the correct input? In designing a solution to this problem, you eventually must decide how to store the input line. In accordance with the ADT approach, you should postpone this decision until you have a better idea of what operations you will need to perform on the data.  
A first attempt at a solution leads to the following pseudocode:  
  
while(not end of line){  
 Read a new character ch  
 if(ch is not a ‘<-‘)  
 Add ch to the ADT  
 else  
 Remove from the ADT the item added most recently  
}  
  
This solution calls two of the operations that the ADT include:  
 Add a new item to the ADT  
 Remove from the ADT the item that was added most recently

Note, the potential trouble could be if you type backspace in the beginning. Here, you will try to remove something which does not exist. Thus:  
  
while(not end of line){  
 Read a new character ch  
 if(ch is not a ‘<-‘)  
 Add ch to the ADT  
 else if (the ADT is not empty)  
 Remove from the ADT the item added most recently  
 else Ignore ‘<-‘  
}

From this pseudocode you can identify a third operation required by the ADT:   
 Determine whether the ADT is empty.

Now suppose that you want to display the line. What you should have done was to retrieve from the ADT the item was added most recently. A retrieval operation means to *look at*, but leave *unchanged*.  
Thus the fourth operation is required by the ADT:  
Retrieve from the ADT the item that was added most recently.

The following operations define the ADT stack:  
1. Create an empty stack  
2. Determine whether a stack is empty  
3. Add a new item to the stack  
4. Remove from the stack the item that was added most recently  
5. Remove all the items from the stack  
6. Retrieve from the stack the item that was added most recently

A stack has the property that the last item placed on the stack will be the first item removed. This property is commonly referred to as last-in, first-out, or simply LIFO. The LIFO property of stacks seems inherently unfair.  
A stack of dishes in a cafeteria makes a very good analogy of the abstract data type stack.

|  |
| --- |
| Stack |
| top |
| items |
| createStack() |
| popAll() |
| isEmpty() |
| push() |
| pop()  peek() |

To capture formally the intuitive notion that the last item inserted into stack is the first item to be removed, you could write an axiom such as:  
(stack.push(item).pop() = stack

**2. Simple Applications of the ADT Stack**

Checking for Balanced Braces  
Java uses curly braces, “{“ & “}” , to delimit groups of statements. If you treat a Java program as a string of characters, you can use a stack to verify that a program contains balanced braces. For example:  
abc{defg{ijk}{l{mn}op}qr  
are balanced, while the braces in the string  
abc{def}}{ghij{kl}m  
are not balanced.   
One way to perform this task is to push each “{“ encountered onto a stack and pop one off each time you encounter “}”:

while(not at the end of the string){  
 if(the next character is a ‘{‘)  
 stack.push(“{“)  
 else if (the character is a “}”s)  
 stack.pop()  
}

This pseudocode keeps track of braces, but missing the check whether the braces are balanced. Thus needs to be added isEmpty check before pop operation.   
The figure below shows the stacks that result when this algorithm is applied to several simple examples:  
