Stack

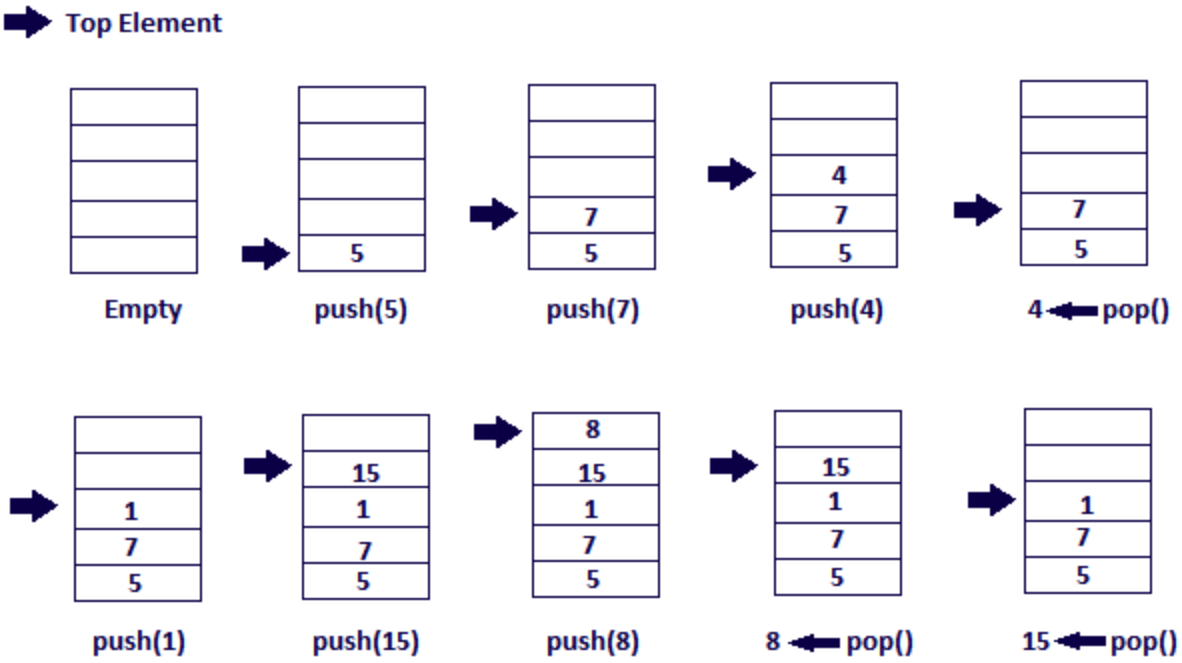
Stacks are the containers where items are retrieved according to the order of insertion, independent of content. Stack is a linear data structure which can store any abstract data type and maintains *Last-In First-Out* (LIFO) order.Which means the element inserted in the very last will be retrieved very first.

Stacks naturally model piles of objects, such as dinner plates. After a new plate is washed, it is pushed on the top of the stack. When someone is hungry, a clean plate is popped off the top. A stack is an appropriate data structure for this task since the plates don't care which one is used next. Thus one important application of stacks is whenever order doesn't matter, because stacks are particularly simple containers to implement.

The abstract operations on a stack are :

1. **Push(S,x) -** Insert item x at the top of stack S
2. **Pop(S) -** Remove the top item from the stack S
3. **Top(S) -**Return the top element of the stack S
4. **Initialize(S) -** Create an empty stack S
5. **Full(S) -** Test whether stack S can accept more pushes
6. **Empty(S) -** Test whether stack S can accept more pops

These basic operations can be understood by the following diagram :



The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed). A stack pointer is the register which holds the value of the stack. The stack pointer always points to the top value of the stack.

There are limitations with the Stack data type. Only small number of operations can be performed on it.The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest.

Applications of Stack : Stacks are used in many applications extensively. Some of these are :

- Infix to postfix conversion
- Memory management
- Expression parsing evaluation
- DFS traversal of a graph

- Tower of Hanoi problem
- Queue Implementation
- Quick Sorting
- Back Tracking

Postfix conversion of a fully paranthesized infix expression is explained here.

Infix to Postfix conversion

Given is a fully paranthesized Infix expression : ((a+b)*(c/((a-d)^(b*c)))).

The Postfix representation for this expression is: ab+cad-bc*^/* .

The procedure of the conversion is to scan the infix string, considering each character one by one. The postfix expression is stored in an Output string. If the character is a variable (here a,b,c ...etc.), then simply add it to the Output string. Otherwise if an operator is read, then we perform a *priority* check and then push it into the stack. The priorities of the operators are (,),^,*,/,%,+,- in the non - increasing order. If the stack top contains an operator of priority higher than or equal to the priority of the input operator, just read, then we pop it and add it to the Output string. We keep on performing the priority check until the top of stack either contains an operator of lower priority or if it does not contain an operator. Finally, consider the case when a paranthesis is read. If the input character is left paranthesis, then simply push it into the stack; else if input character is a right paranthesis then we keep poping and adding top element of the stack until it is a left paranthesis or the stack is empty. If top element is a left paranthesis then pop it, discard the input character and move to the next character. The complete procedure for postfix conversion of the expression ((a+b)*(c/((a-d)^(b*c)))) is explained below:

Input Character	Operation to be performed	Stack(bottom to top)	Output String
(	Push into the stack	(	Null
(	Push into the stack	(,(	Null
a	Add to the output string	(,(	a
+	Push into the stack	(,(,+	a
b	Add to the output string	(,(,+	ab
)	Pop until popped element is '('	(	ab+
*	Push into the stack	(,*	ab+
(	Push into the stack	(,*,(	ab+
c	Add to the output string	(,*,(	ab+c
/	Push into the stack	(,*,(,/	ab+c
(	Push into the stack	(,*,(,/,(	ab+c
(	Push into the stack	(,*,(,/,(,(	ab+c
a	Add to the output string	(,*,(,/,(,(	ab+ca
-	Push into the stack	(,*,(,/,(,(,-	ab+ca
d	Add to the output string	(,*,(,/,(,(,-	ab+cad
)	Pop until popped element is '('	(,*,(,/,(	ab+cad-
^	Push into the stack	(,*,(,/,(,^	ab+cad-
(	Push into the stack	(,*,(,/,(,^,(	ab+cad-
b	Add to the output string	(,*,(,/,(,^,(	ab+cad-b

*	Push into the stack	(,*,(,/,(,^,(,*	ab+cad-b
c	Add to the output string	(,*,(,/,(,^,(,*	ab+cad-bc
)	Pop until popped element is '('	(,*,(,/,(,^	ab+cad-bc*
)	Pop until popped element is '('	(,*,(,/	ab+cad-bc*^
)	Pop until popped element is '('	(,*	ab+cad-bc*^/
)	Pop until popped element is '('	Empty	ab+cad-bc*^/*

Hence, the postfix expression for the given infix expression is : ab+cad-bc*^/*. The procedure explained here is performed on a fully paranthesized Infix expression. Similar method can be applied for the postfix conversion of partially paranthesized expressions and expressions without paranthesis.

Source:

http://www.learnalgorithms.in/#