STACKS

ALGORITHMS & DATA STRUCTURES – I

COMP 221

Stack as an ADT

A **stack** is **an *ordered* collection of data items in which *access is possible only at one end* (called the *top* of the stack).**

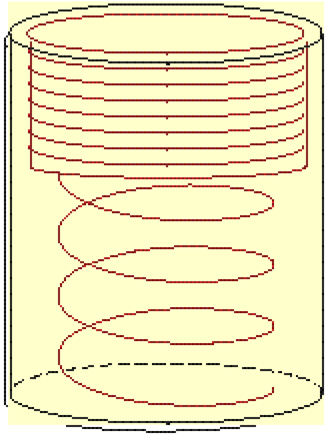
Basic operations:

1. **Construct a stack (usually empty)**
2. Check if stack **is empty**
3. **Push: Add** an element **at the top** of the stack
4. **Top: Retrieve** the **top element** of the stack
5. **Pop: Remove** the **top element** of the stack

Terminology is from spring-loaded stack of plates in a cafeteria:

Adding a plate pushes those below it down in the stack

Removing a plate pops those below it up one position.



Stack Operations:

Construction:	Initializes an empty stack.
Empty operation:	Determines if stack contains any values
Push operation:	Modifies a stack by adding a value to top of stack
Top operation:	Retrieves the value at the top of the stack
Pop operation:	Modifies a stack by removing the top value of the stack

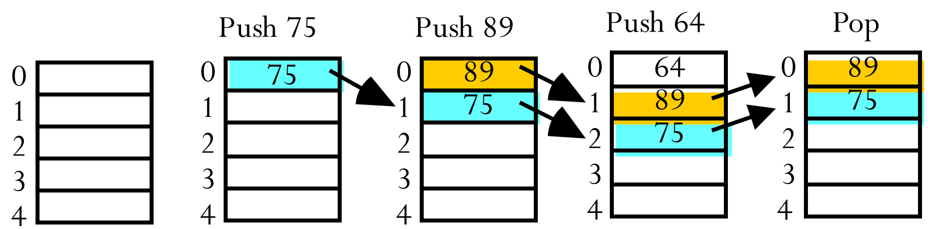
To help with debugging, add early on:

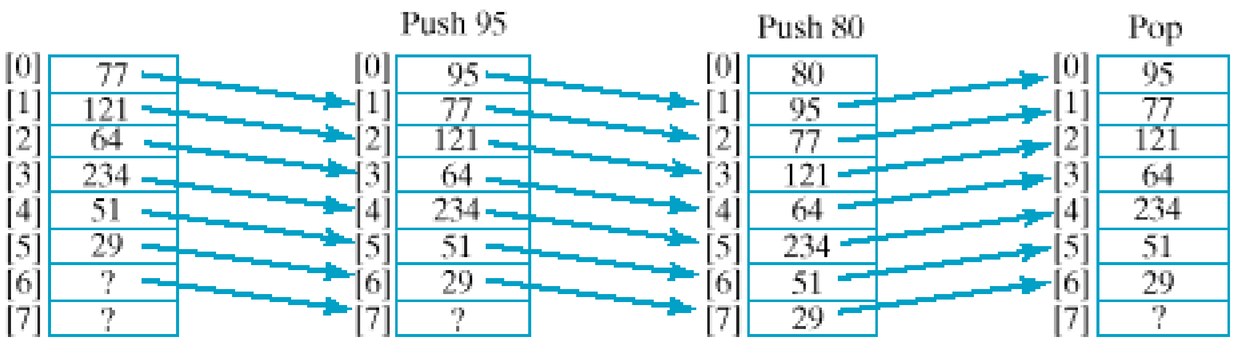
Output:	Displays all the elements stored in the stack.

2. Implementing a `Stack` Class

Define data members: consider storage structure(s)

Attempt #1: Use an array with the top of the stack at position 0.
e.g., Push 75, Push 89, Push 64, Pop

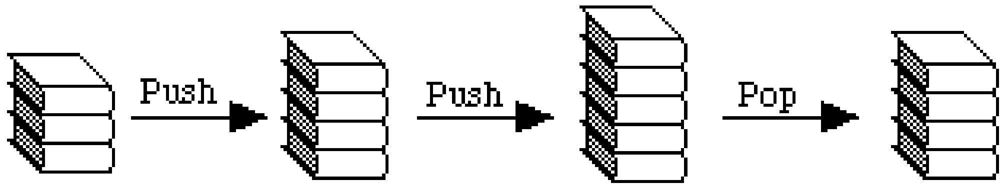




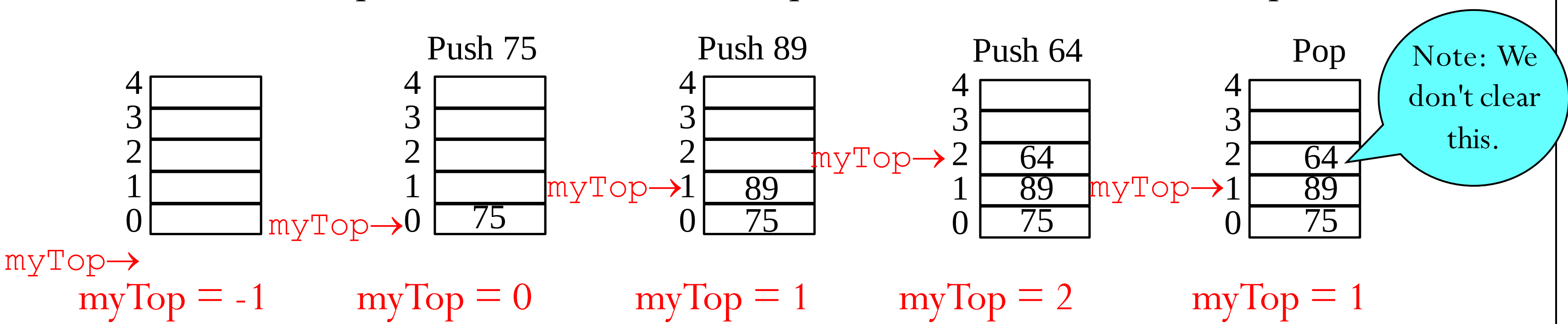
+ features: This **models the operation of the stack of plates**.
– features: **Not efficient to shift the array elements up and down in the array.**

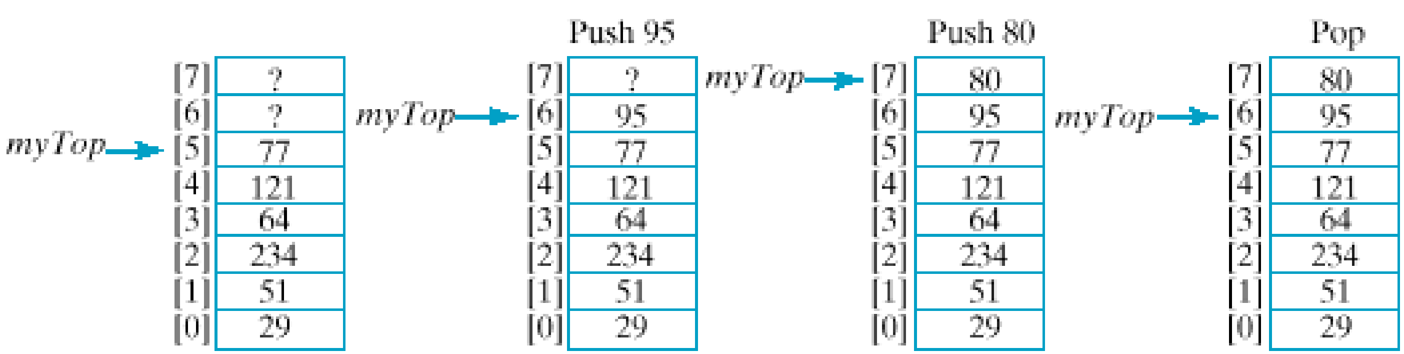
Implementing `Stack` Class – Refined

Instead of modeling a stack of plates, model a stack of <u>books</u> (or a discard pile in a card game.)



Keep the bottom of stack at position 0. Maintain a "pointer" **`myTop`** to the top of the stack.





Note: **No moving of array elements.**

`Stack`'s Data Members

Provide:

- **An array** data member to hold the **stack elements**.
- **An integer** data member to indicate the **top of the stack**.

Problems: We need an array declaration of the form

```
ArrayElementType myArray[ARRAYCAPACITY];
```

— *What type should be used?*

Solution (for now): Use the **`typedef`** mechanism:

```
typedef int StackElement;
// put this before the class declaration
```

— *What about the capacity?*

```
const int STACK_CAPACITY = 128;
// put this before the class declaration
```

Now we can declare the array data member in the private section:

```
StackElement myArray[STACK_CAPACITY];
```

A Simple Array Based Implementation

- We initialize **top** to **-1** (which means stack is empty initially).
- size: No of elements in stack: top + 1. (-1 + 1 = 0 elements initially)
- isEmpty: if top < 0 then true otherwise false.
- To push object:
  - If size is N (full stack) throw Exception
  - Otherwise increment top and store new object at S[top]

A Simple Array Based Implementation

- To pop:
  - If isEmpty() is true then print Stack is Empty
  - Otherwise store S[top] in a local variable, assign null to S[top], decrement top and return local variable having the previous top.

Implementation of Stack Operations

size():

return top+1

isempty()

if(top < 0)

return 1;

else

return 0;

isfull()

if (top == size-1)

return 1;

else

return 0;

push(int):

if isfull() **then**

print overflow

else

top = top + 1;

s[top] = [insert item];

pop():

if isempty() **then**

print stack is empty

else

cout<<s[top]

top--;

Application of Stacks: RPN

For most common arithmetic operations the operator symbol is placed between its two operands. For example,

A + B, C - D, E*F, G/H

This is called ***Infix Notation***

Polish Notation, named after the Polish mathematician Jan Lukasiewich, refers to the notation in which the operator symbol is placed before its two operands. For example,

+AB, -CD, *EF, /GH

Frequently known as **Prefix Notation**

Reverse Polish Notation refers to the analogous notation in which the operator symbol is placed after its two operands. For example,

AB+, CD-, EF*, GH/

Frequently known as **Postfix Notation**

Stack Applications

- Postponement: Evaluating arithmetic expressions.

- Prefix: + a b
- Infix: a + b (what we use in grammar school)
- Postfix: a b +

- In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Examples

Infix	Prefix(PN)	Postfix (RPN)
A + B	+ A B	A B +
A * B + C	+ * A B C	A B * C +
A * (B + C)	* A + B C	A B C + *
A – B – C – D	- - - A B C D	A B – C – D –

Infix to Postfix Conversion

- Rules:
  - Operands immediately go directly to output
  - Operators are pushed into the stack (including parenthesis)
    - Check to see if stack top operator is less than current operator
    - If the top operator is less than, push the current operator onto stack
    - If the top operator is greater than the current, pop top operator and push onto stack, push current operator onto stack
    - Priority 2: * /
    - Priority 1: + -
    - Priority 0: (

If we encounter a right parenthesis, pop from stack until we get matching left parenthesis. Do not output parenthesis.

Infix to Postfix Example

A + B * C − D / E

Infix	**Stack(bot->top)**	**Postfix**
a) A + B * C − D / E		
b) + B * C − D / E		A
c) B * C − D / E	+	A
d) * C − D / E	+	A B
e) C − D / E	+ *	A B
f) − D / E	+ *	A B C
g) D / E	+ −	A B C *
h) / E	+ −	A B C * D
i) E	+ − /	A B C * D
j)	+ − /	A B C * D E
k)		A B C * D E / − +

Infix to Postfix Example

A * B - (C + D) + E

	Infix	**Stack(bot->top)**	**Postfix**
a)	A * B - (C - D) + E	empty	empty
b)	* B - (C + D) + E	empty	A
c)	B - (C + D) + E	*	A
d)	- (C + D) + E	*	A B
e)	- (C + D) + E	empty	A B *
f)	(C + D) + E	-	A B *
g)	C + D) + E	- (	A B *
h)	+ D) + E	- (	A B * C
i)	D) + E	- (+	A B * C
j)	) + E	- (+	A B * C D
k)	+ E	-	A B * C D +
l)	+ E	empty	A B * C D + -
m)	E	+	A B * C D + -
n)		+	A B * C D + - E
o)		empty	A B * C D + - E +

Converting Infix to Postfix Expression

Suppose the following arithmetic expression Q written in Infix notation:

Q: A + (B * C – (D / E ^ F) * G) * H

We transform Infix expression Q to its equivalent expression P in Postfix by using stack to hold operator and left parenthesis.

First we push "(" onto STACK and then we add ")" to the end of Q.

Symbol Scanned	STACK	Expression P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) -	(+ (-	A B C *
(8) (	(+ (- (	A B C *
(9) D	(+ (- (	A B C * D
(10) /	(+ (- (/	A B C * D

Converting Infix to Postfix Expression

Symbol Scanned	STACK	Expression P
(11) E	(+ (- (/	A B C * D E
(12) ^	(+ (- (/ ^	A B C * D E
(13) F	(+ (- (/ ^	A B C * D E F
(14))	(+ (-	A B C * D E F ^ /
(15) *	(+ (- *	A B C * D E F ^ /
(16) G	(+ (- *	A B C * D E F ^ / G
(17))	(+	A B C * D E F ^ / G * -
(18) *	(+ *	A B C * D E F ^ / G * -
(19) H	(+ *	A B C * D E F ^ / G * - H
(20))		A B C * D E F ^ / G * - H * +

Evaluation of Postfix Expression: RPN

Underlining Technique

1. Scan the expression from left to right to find an operator.
2. Locate the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example 1: **2 3 4 + 5 6 - - ***

2 <u>3 4 +</u> 5 6 - - *

2 7 5 6 - - *

2 7 <u>5 6 -</u> - *

2 7 -1 - *

2 <u>7 -1 -</u> *

2 8 *

<u>2 8 *</u>

16

Evaluation of Postfix Expression: RPN

Example 2: Suppose the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, -

We evaluate P by adding a sentinel right parenthesis at the end of P to obtain

5,	**6,**	**2,**	**+,**	***,**	**12,**	**4,**	**/,**	**-,**	**)**
(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**	**(8)**	**(9)**	**(10)**

Symbol Scanned		**STACK**
(1)	**5**	**5**
(2)	**6**	**5, 6**
(3)	**2**	**5, 6, 2**
(4)	**+**	**5, 8**
(5)	*****	**40**
(6)	**12**	**40, 12**
(7)	**4**	**40, 12, 4**
(8)	**/**	**40, 3**
(9)	**-**	**37**
(10)	**)**	

THANK YOU

?