Stacks

- Introduction: List and Array representations,
- Operations on stack (traversal, push and pop)
- Arithmetic expressions: polish notation, evaluation and transformation of expressions.

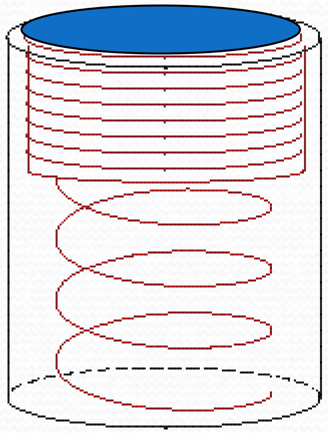
Introduction to Stacks

- Consider a card game with a discard pile
  - Discards always <u>placed</u> on the <u>top</u> of the pile
  - Players may <u>retrieve</u> a card only from the top

What other examples can you think of that are modeled by a stack?

- We seek a way to represent and manipulate this in a computer program
- This is a <u>stack</u>

- A stack is a last-in-first-out (LIFO) data structure
- Adding an itemReferred to as <u>pushing</u> it onto the stack
- Removing an item
  - Referred to as <u>popping</u> it from the stack
- A stack is a list of elements in which an element may be inserted or deleted only at one end, called TOP of the stack.
- Also called LIFO, piles, push-down lists.
- 2 basic operations:
  - Push
  - Pop



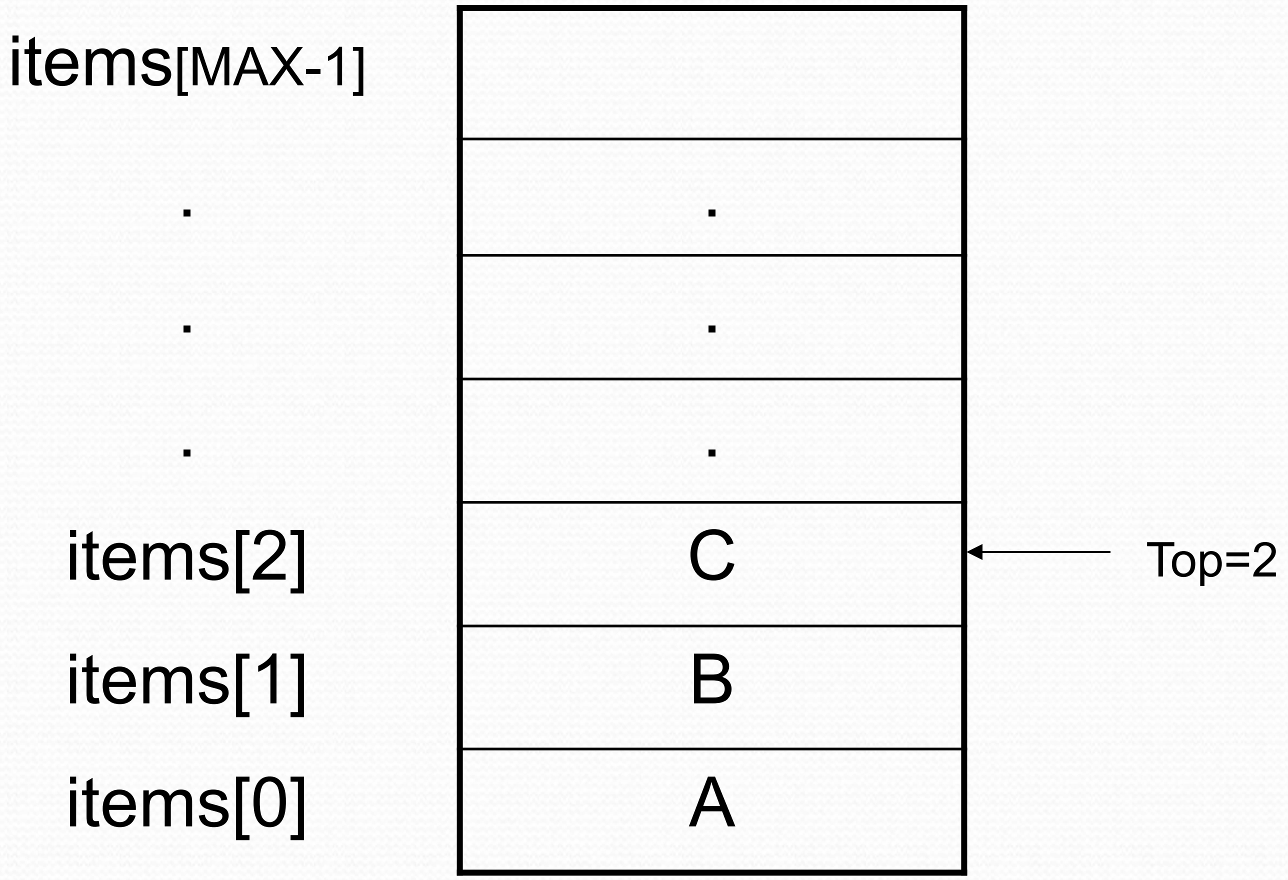
Stack

- Definition:
  - An ordered collection of data items
  - Can be accessed at only one end (the top)
- Operations:
  - construct a stack (usually empty)
  - check if it is empty
  - Push: add an element to the top
  - Top: retrieve the top element
  - Pop: remove the top element

Applications of stack:

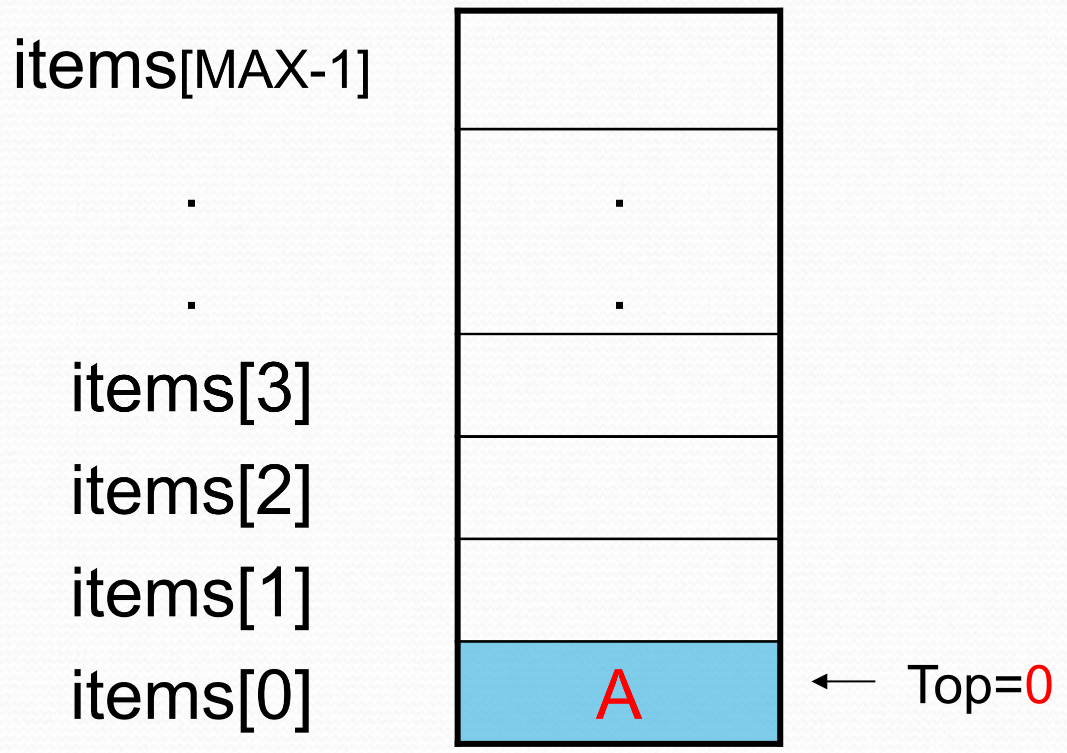
- Recursion: Quick Sort,Tower Of Hanoi
- Postponed decision : Stacks are frequently used to indicate the order of processing of data when certain steps of the processing must be postponed until other conditions are fulfilled. **Polish Notations and Reverse Polish Notations**

Stack



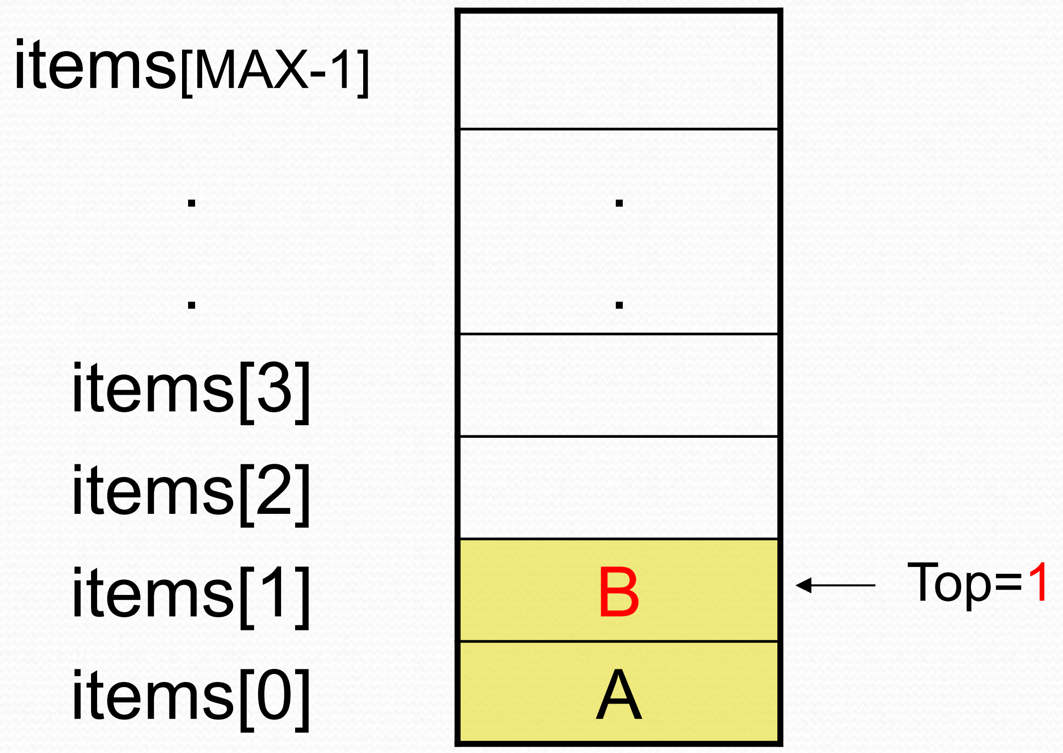
Insert an item A

- **A new item (*A*) is *inserted* at the *Top* of the stack**



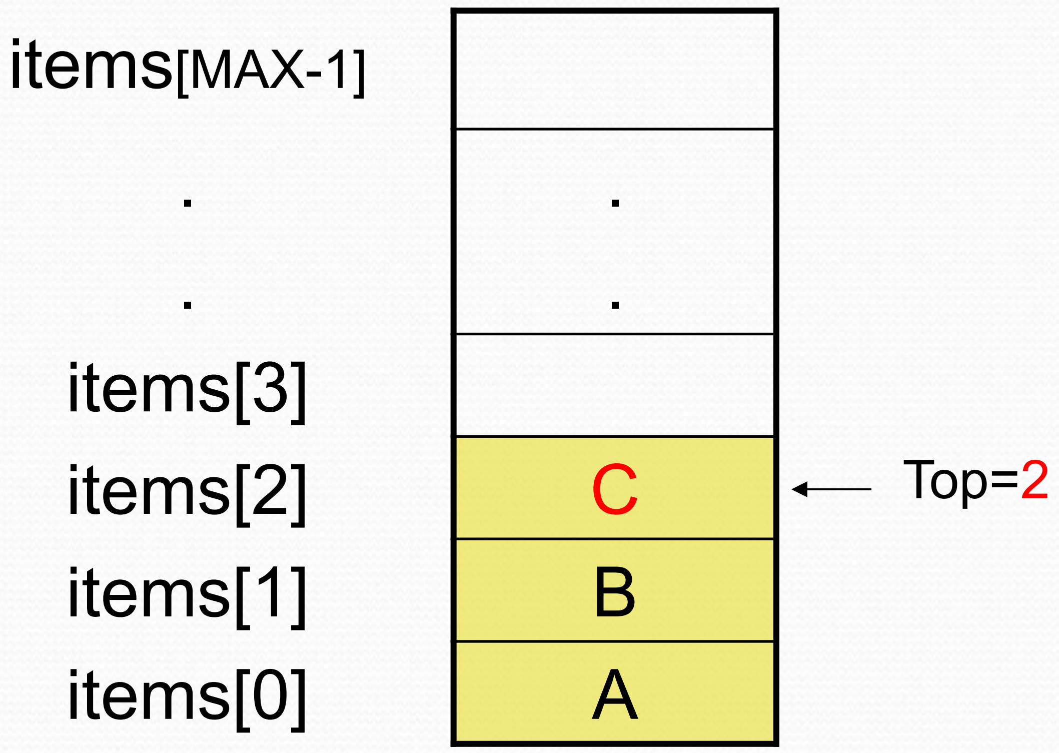
Insert an item B

- **A new item (*B*) is *inserted* at the *Top* of the stack**



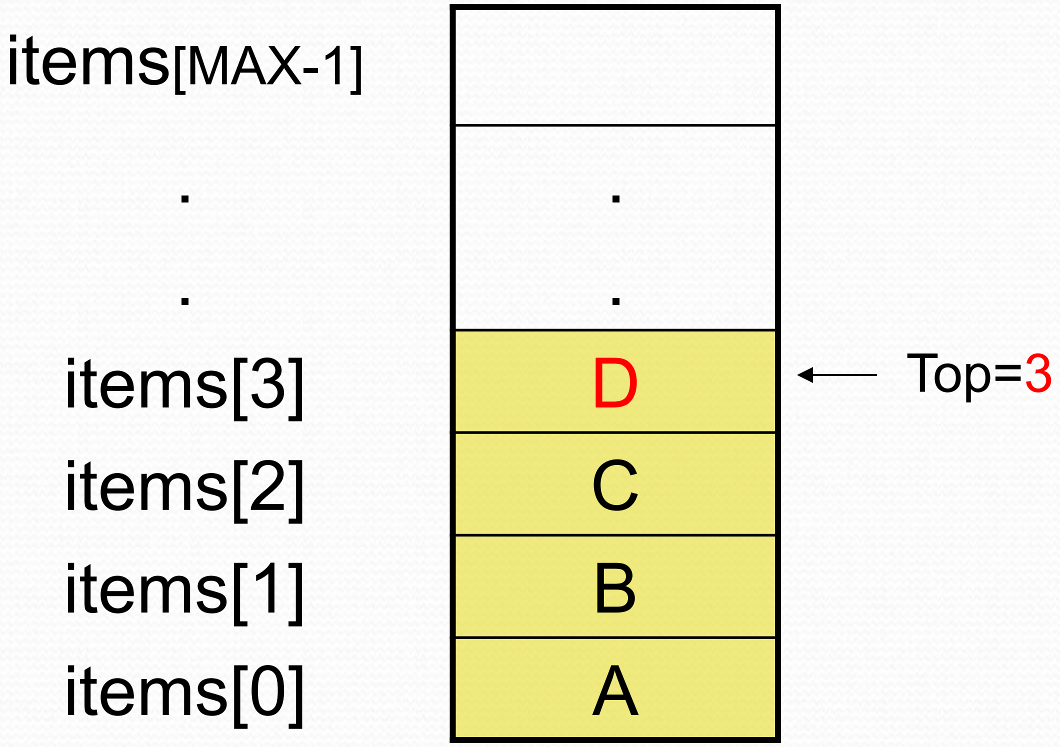
Insert an item C

- **A new item (*C*) is *inserted* at the *Top* of the stack**



Insert an item D

- **A new item (*D*) is *inserted* at the *Top* of the stack**



Array Representation of Stacks

- We represent a stack in memory using a one-way list or a linear array.
- Array representation:

✓ Array : STACK.

✓ A pointer variable TOP.

✓ A variable MAXSTK.

✓ The condition, TOP = NULL, indicates stack is empty.

✓ The condition, TOP = MAXSTK, indicates the stack is full.

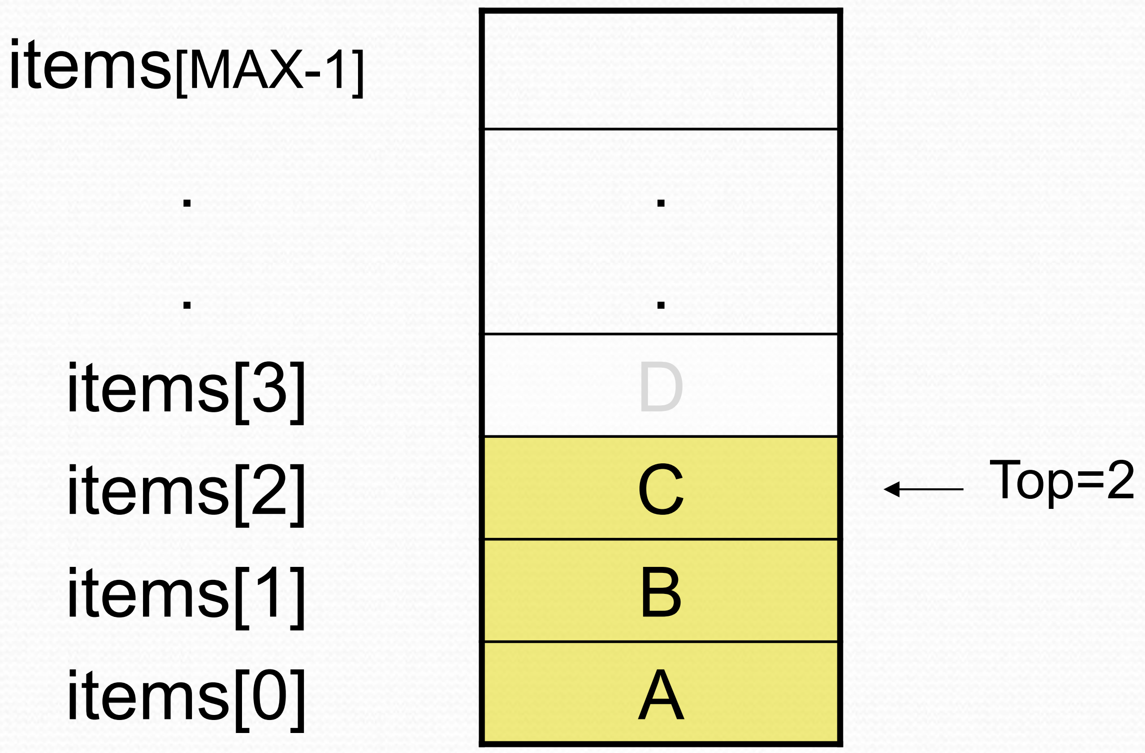
Insert Operation(Array)

PUSH(STACK, N, TOP, ITEM):to insert element in stack array STACK having N elements where TOP represent index of last element in the array.

```
1. If TOP=N,then:
       write :OVERFLOW, and Exit.
2. Set TOP := TOP + 1.
3. Set STACK[TOP]:= ITEM.
4. Exit.
```

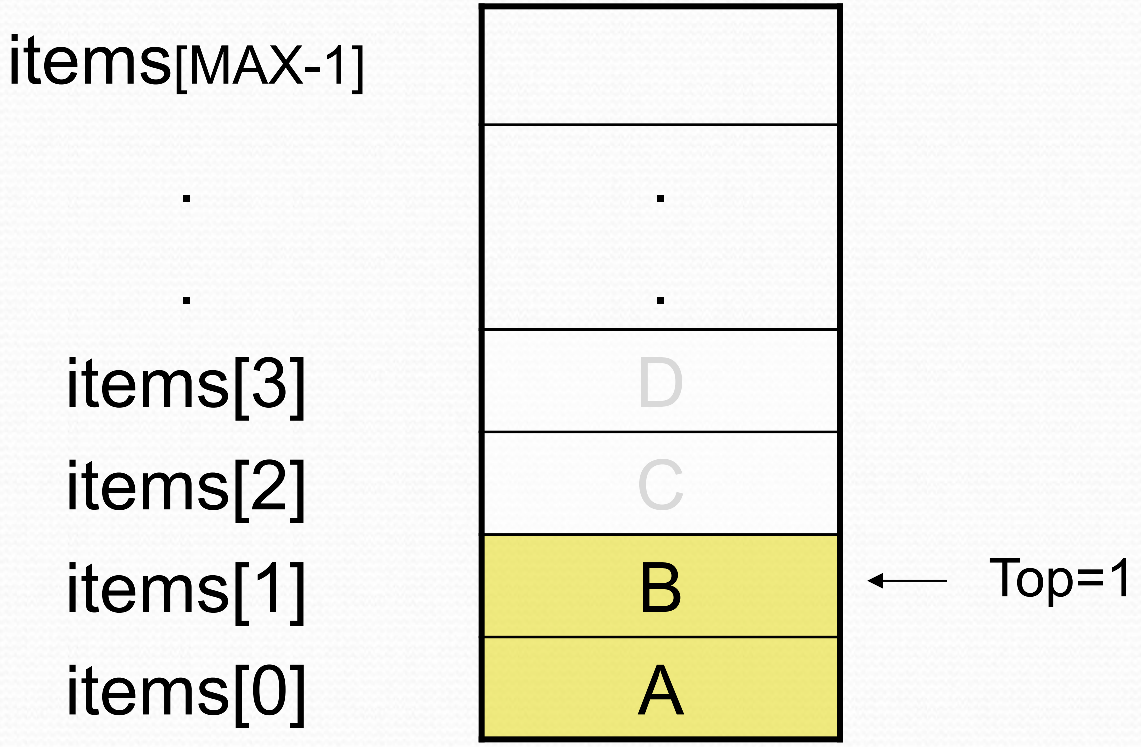
Delete D

- **an item (*D*) is deleted from the *Top* of the stack**



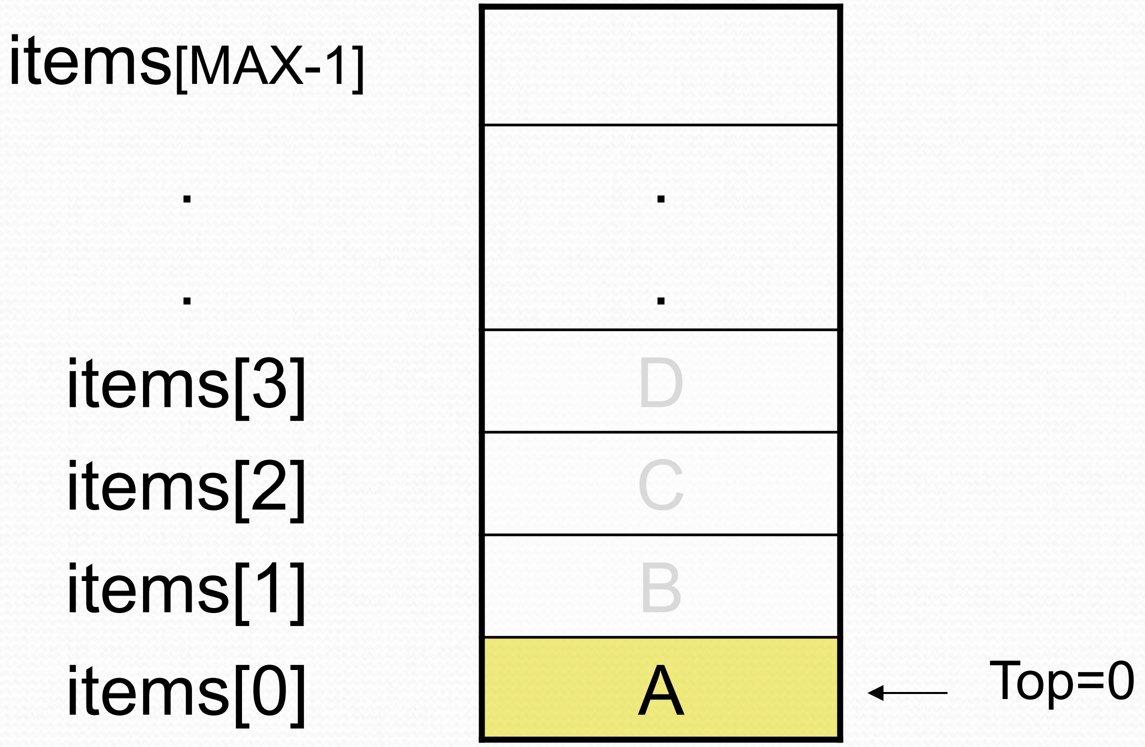
Delete C

- **an item (*C*) is deleted from the *Top* of the stack.**



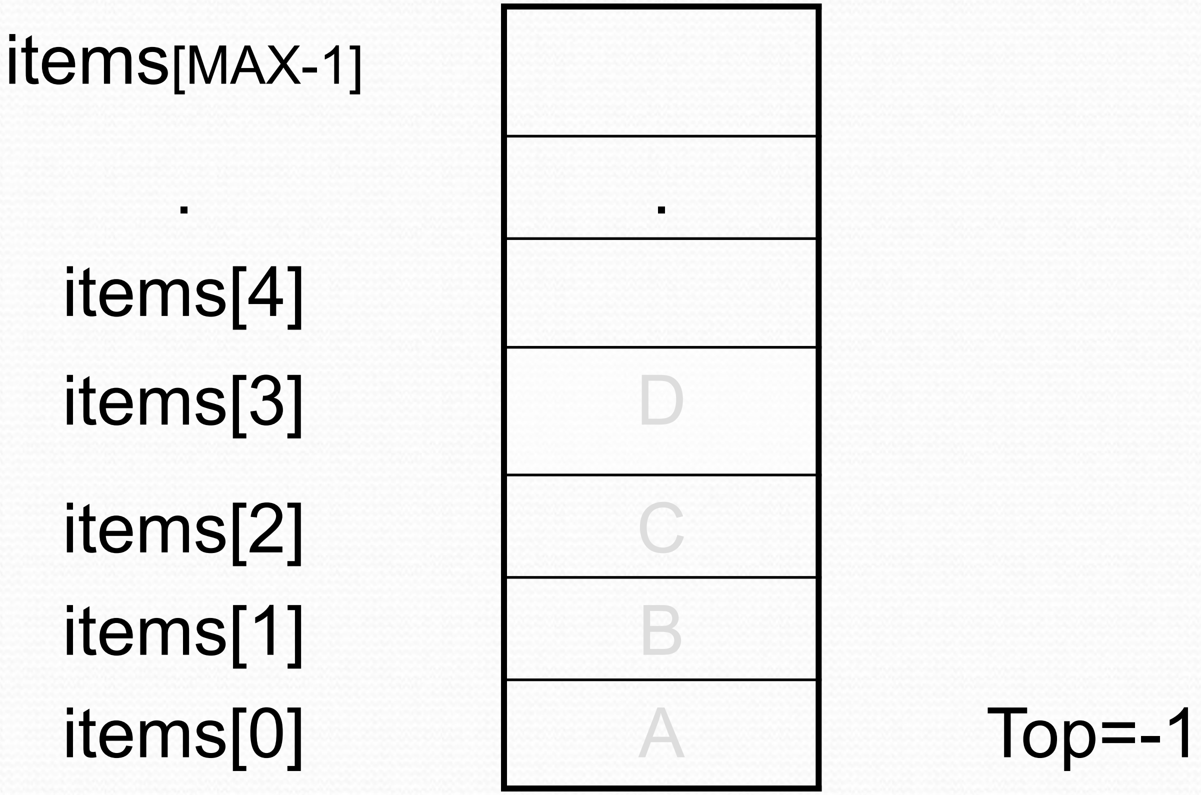
Delete B

- **an item (*B*) is deleted from the *Top* of the stack**



Delete A

- **an item (*A*) is deleted from the *Top* of the stack.**



Delete Operation(Array)

```
POP(STACK, N, TOP, ITEM)
1.  If TOP = NULL ,then :
        write: UNDERFLOW, and Return.
2.  Set ITEM := STACK [TOP].
3.  Set TOP := TOP - 1.
4.  Return.
```

Linked List Representation of Stacks

- We also call it linked stack, implemented using a singly linked list.
- The INFO fields of the nodes hold the elements of the stack and the LINK fields hold the pointers to the neighboring elements in the stack.
- The START pointer acts as the TOP of the stack.
- The node containing NULL pointer denotes the BOTTOM of the stack.
- PUSH and POP is done from START only.
- No need of MAXSTK.

Insert Operation(LL)

```
PUSH(INFO, LINK, TOP, AVAIL, ITEM)
1.  If AVAIL = NULL, then :
        write :OVERFLOW, and Exit.
2.  Set NEW := AVAIL and AVAIL := LINK[AVAIL]
3.  Set INFO[NEW]:= ITEM
4.  Set LINK[NEW] := TOP
5.  Set TOP := NEW
6.  Exit
```

Delete Operation(LL)

```
POP(INFO, LINK, TOP, AVAIL, ITEM)
1.  If TOP = NULL, then :
        write UNDERFLOW, and Exit.
2.  Set Set ITEM :=INFO[TOP]
3.  Set TEMP:==TOPa nd TOP :=LINK[TOP]
4.  Set LINK[TEMP] := AVAIL  and Set AVAIL := TEMP
5.  Exit
```

Example

- Consider following stack of characters, where MAXSTK=8.
- STACK= A, C, D, G, K,_, _, _. Demonstrate following operations in stack:
    - POP(STACK, ITEM).
    - POP(STACK, ITEM).
    - PUSH(STACK, L).
    - PUSH(STACK, P).
- Is there any chance that C would get deleted before D?

Minimizing Overflow

- Programmer does not have any direct control over underflow, because it depends totally upon algorithm and input data.
- But overflow depends on the memory space reserved for each stack.
- The space reserved also influences the number of times the overflow occurs.
- The choice of space reserved involves time space tradeoff.

- Allocating large number of space to a stack will reduce the number of times the overflow occurs, but it could result in wastage of memory space if complete memory is seldom used.
- On the other hand, reserving a small amount of memory may result in increased number of overflows, and would increase the time to resolve the overflow.
- One solution is to use one single array whenever we need 2 arrays. But it would need some modification in the PUSH and POP algorithms.

Arithmetic Expressions: Polish Notations

- This section gives an algorithm which finds the value of an arithmetic expression using reverse Polish(postfix) notation.
- It is another application of STACK.
- Assumptions:
- Precedence
  - Exponent
  - Multiplication and division
  - Addition and subtraction
- All expressions are evaluated from left to right.

- Eg: 2^3 +5*2^2-12/6

- In normal arithmetic operations:
  - A+B.
  - This is called INFIX notation.
  - In this (A+B)*C is different from A+(B*C).
  - Hence the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Postfix Notation(RPN)

- Polish notation, also known as prefix notation.
- It is a symbolic logic invented by **Polish** mathematician **Jan Lukasiewicz** in the 1920's.
- Most compilers convert an expression in ***infix*** notation to ***postfix***
  - the operators are written <u>after</u> the operands
- So a * b + c becomes a b * c +
- Advantage:
  - expressions can be written without parentheses

Postfix and Prefix Examples

INFIX	POSTFIX	PREFIX
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
A - B - C - D	A B-C-D-	---A B C D

Prefix : Operators come <u>before</u> the operands

- How computer evaluates an arithmetic expression????
  - In 2 steps:
    1. It converts the expression into postfix notation.
    2. Then it evaluates the postfix expression.

    All this is done using stacks.

Evaluating RPN Expressions

"By hand" (Underlining technique):

1. Scan the expression from left to right to find an operator.

2. Locate ("underline") the last two preceding operands
   and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:

$$2\ 3\ 4\ +\ 5\ 6\ -\ -\ *$$

$$\rightarrow\ 2\ \underline{3\ 4\ +}\ 5\ 6\ -\ -\ *$$

$$\rightarrow\ 2\ 7\ 5\ 6\ -\ -\ *$$

$$\rightarrow\ 2\ 7\ \underline{5\ 6\ -}\ -\ *$$

$$\rightarrow\ 2\ 7\ -1\ -\ *$$

$$\rightarrow\ 2\ \underline{7\ -1\ -}\ *\ \rightarrow\ 2\ 8\ *\ \rightarrow\ \underline{2\ 8\ *}\ \rightarrow\ 16$$

Evaluating RPN Expressions

- P is an arithmetic expression in Postfix Notation.

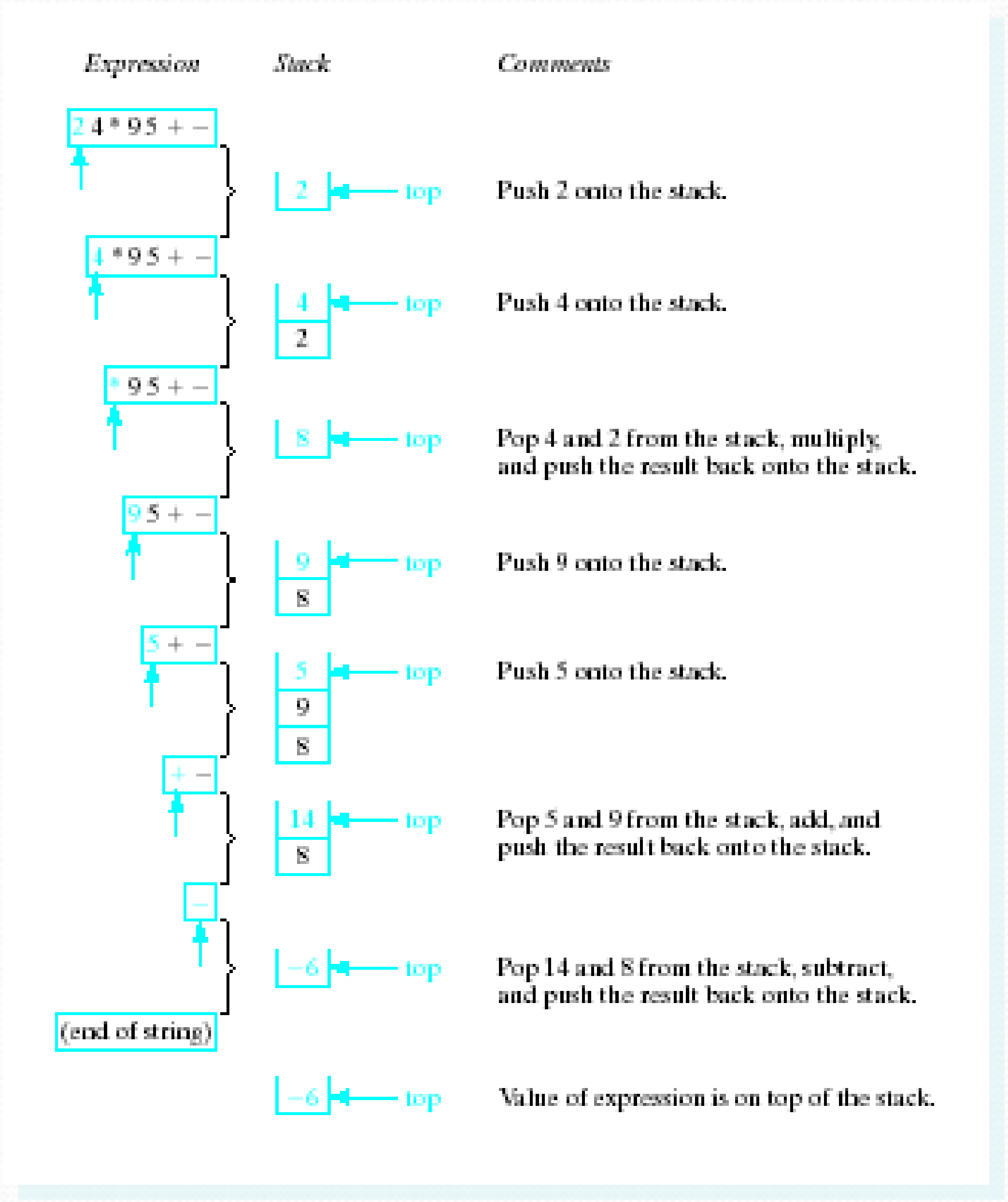
```
1.  Add a right parenthesis ")" at the end of P.

2.  Scan P from left to right and Repeat Step 3 and 4 for each element of P
    until the sentinel ")" is encountered.

3.          If an operand is encountered, put it on STACK.
4.          If an operator @ is encountered, then:
             (a) Remove the two top elements of STACK, where A
                 is the top element and B is the next to top element.
             (b) Evaluate B @ A.
             (c) Place the result of (b) back on STACK.
            [End of if structure.]
    [End of step 2 Loop.]
5.  Set VALUE equal to the top element on STACK.
6.  Exit.
```

Evaluating RPN Expressions

- Note the changing status of the stack



Expression	*Stack*	*Comments*
2 4 * 9 5 + −	2 ← top	Push 2 onto the stack.
4 * 9 5 + −	4 ← top 2	Push 4 onto the stack.
* 9 5 + −	8 ← top	Pop 4 and 2 from the stack, multiply, and push the result back onto the stack.
9 5 + −	9 ← top 8	Push 9 onto the stack.
5 + −	5 ← top 9 8	Push 5 onto the stack.
+ −	14 ← top 8	Pop 5 and 9 from the stack, add, and push the result back onto the stack.
−	−6 ← top	Pop 14 and 8 from the stack, subtract, and push the result back onto the stack.
(end of string)	−6 ← top	Value of expression is on top of the stack.

Transforming Infix into Postfix

By hand: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.

2. Replace each right parenthesis by the corresponding operator.

3. Erase all left parentheses.

Examples:

```
A * B + C   →  ((A * B) + C)      A * (B + C)  →  (A * (B + C) )
            →  ((A B * C +                      →  (A (B C + *
            →  A B * C +                        →  A B C + *
```

Stack Algorithm

```
POLISH (Q, P)
1.  PUSH "(" on to STACK and add ")" to the end of Q.
2.  Scan Q from left to right and Repeat steps 3 to 6 for each element of Q
    until the STACK is empty:
3.      If an operand is encountered, add it to P.
4.      If a left parenthesis is encountered, push it onto STACK.
5.      If an operator is encountered, then:
            (a) Repeatedly POP from STACK and add to P each operator
    (On the TOP of STACK) which has the same precedence as or higher
    precedence than @.
            (b) Add @ to STACK.
    [End of If structure.]
6.  If a right parenthesis is encountered, then:
        (a) Repeatedly POP from STACK and add to P each operator (On the
    TOP of STACK.) until a left parenthesis is encountered.
        (b) Remove the left parenthesis. [Don't add the left parenthesis to P.]
    [End of If Structure.]
   [End of step 2 Loop.]
7. Exit.
```

Transforming Infix into prefix

By hand: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.

2. Replace each left parenthesis by the corresponding operator.

3. Erase all right parentheses.

Examples:

```
A * B + C   → ((A * B) + C)      A * (B + C)   → (A * (B + C))
            → + * A B) C)                      → * A + B C ))
            → + * A B C                        → * A + B C
```

Thank You