Stack ADT and Its Applications

Simple Array Implementation of Lists

- Disadvantages:
  - An estimate of the maximum size of the list is required, even if the array is dynamically allocated. Usually this requires a high overestimate, which wastes considerable space.
  - Insertion and deletion are expensive. For example, inserting at position 0 requires first pushing the entire array down one spot to make room.

Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

Motivation

- **How is undo and redo functionality typically implemented?**
  - Suppose you perform following in word application
  - First you typed a word
  - Secondly, you **bold** it
  - Thirdly, You underlined it
  - Fourth, you changed its color?
  - Fifth you change its font size?
- In which **order** action will redo?
  - The action that we performed in **last** will redo **first**
  - **L**ast **I**n **F**irst **O**ut

Examples

- **Example 1:** Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.
- **Example 2:** Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.

Stack ADT

- A stack is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the top.

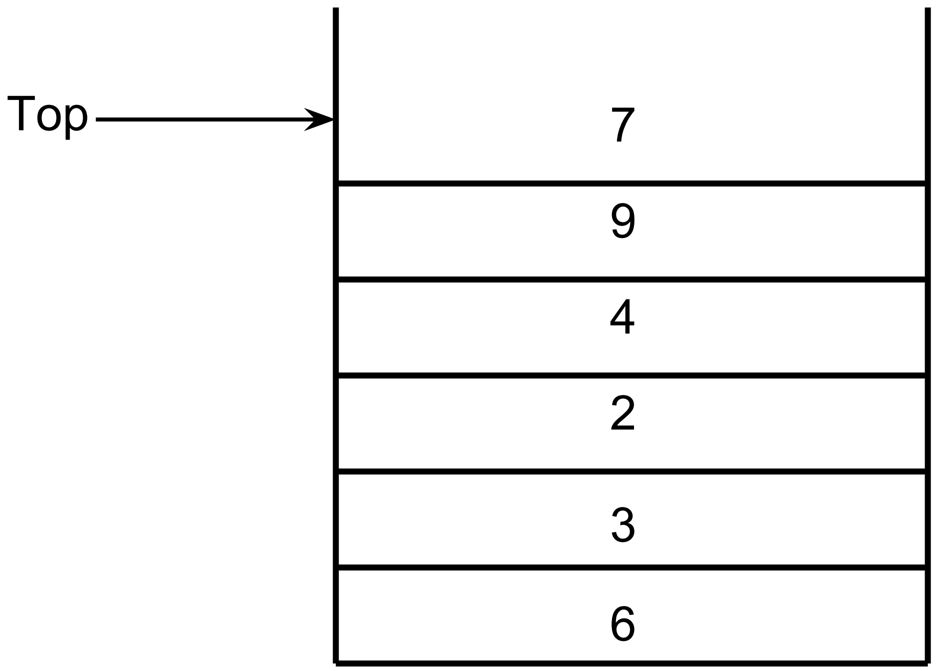
- The fundamental operations on a stack are Push, which is equivalent to an insert, and Pop, which deletes the most recently inserted element.

- The most recently inserted element can be examined prior to performing a Pop by use of the Top routine.

Stack ADT

- A Pop or Top on an empty stack is generally considered an error (stack underflow error) in the stack ADT.
- Running out of space when performing a Push is an implementation error (stack overflow error) but not an ADT error.
- Stacks are sometimes known as LIFO (last in, first out) lists.

Stack ADT



Stack model: only the top element is accessible

Implementation of Stacks

- Since a stack is a list, any list implementation will do.

- We will give two popular implementations. One uses pointers and the other uses an array.

- No matter in which case, if we use good programming principles, the calling routines do not need to know which method is being used.

Array Implementation of Stacks

- A Stack is defined as a pointer to a structure. The structure contains the TopOfStack and Capacity fields. Once the maximum size is known, the stack array can be dynamically allocated.

- Associated with each stack is TopOfStack, which is -1 for an empty stack (this is how an empty stack is initialized).

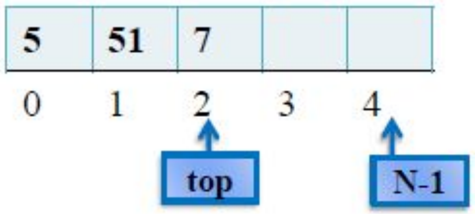
Array Implementation of Stacks

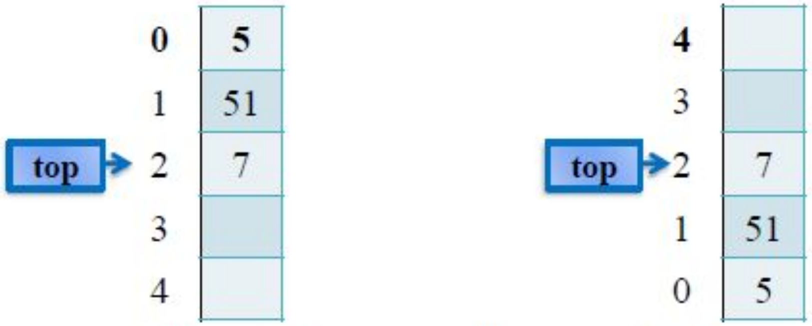
- To push some element X onto the stack, we increment TopOfStack and then set Stack[TopOfStack]=X, where Stack is the array representing the actual stack.

- To pop, we set the return value to Stack[TopOfStack] and then decrement TopOfStack.

Using array

Add and delete at end → O(1)

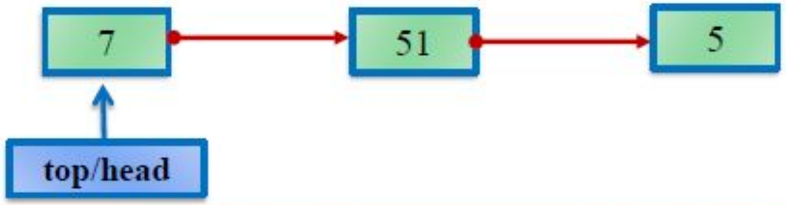




Top to bottom & bottom to top

Using Linked List

Add and delete at start → O(1)



Implementation

- Algorithms:
  - algorithm depends upon individual implementation
- In array
  - a variable top is maintained to denote most recent element's index
    - top =-1 →stack is empty
    - Size = top+1
    - top =N-1 →stack is full
- In Linked list
  - Head node is top
    - Head=null→ stack is empty
- No need of tail

Algorithm (Array based)

Algorithm: Push(Stack, E)

Input: a Stack, a data element E

Output: updated stack with E inserted

Steps:

1. If(Stack is Full)
2. Print "Stack overflow"
3. Else
4. top=top+1
5. Stack[top]= E
6. End If

Algorithm: Pop(Stack)

Input: a stack

Output: updated stack, with top element removed

Steps:

1. Let E = null
2. If(Stack is Empty)
3. Print "Stack underflow"
4. Else
5. E=Stack[top]
6. top=top-1
7. End If
8. return E

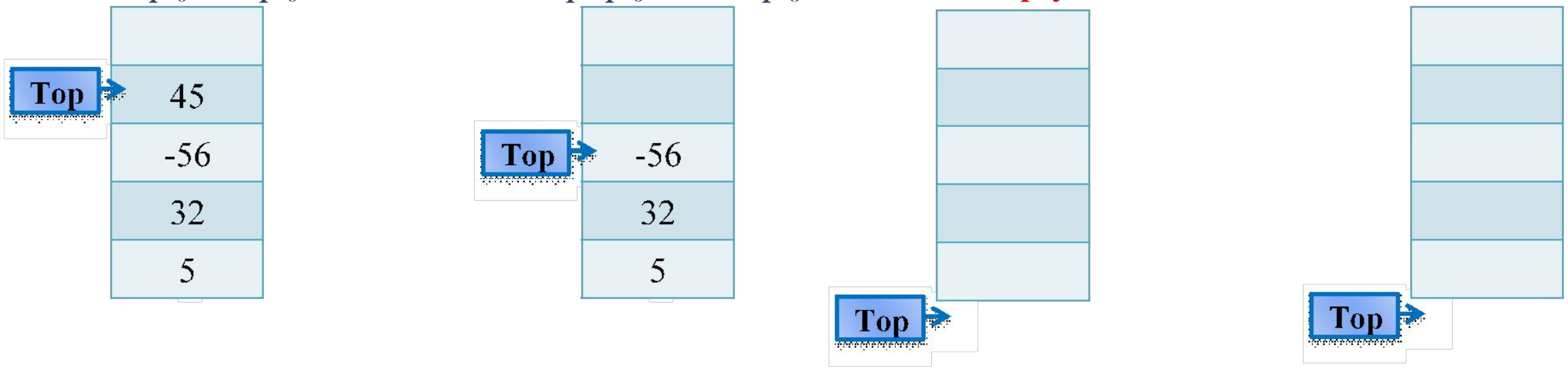
Array Implementation of Stacks

How Stack Works?

- Push (5) Push(32) Push(-56), Push(45), Push(32) Push(11) → **Stack is full**

Array Implementation of Stacks

- Pop() Pop() after 3 more pop() Pop()→**Stack is empty**



Application of Stacks

- Methods calls within a program; functions are called in order and data is stored in activation stack
- Back button on browser store page history
- Undo/redo in editors
- Used by compilers to check program syntax
- Arithmetic Expression Evaluation
- To reverse contents of something like string, array
- Decimal to binary conversion

Infix Notation

- To add A, B, we write

  A+B
- To multiply A, B, we write

  A*B
- The operators ('+' and '*') go in between the operands ('A' and 'B')
- This is *"Infix"* notation.

Prefix Notation

- Instead of saying "A plus B", we could say "add A,B " and write

  + A B
- "Multiply A,B" would be written

  * A B
- This is *Prefix* notation.

Postfix Notation

- Another alternative is to put the operators after the operands as in

A B +

and

A B *

- This is *Postfix* notation.

Infix, Prefix and Postfix

- The terms infix, prefix, and postfix tell us whether the operators go between, before, or after the operands.

Arithmetic Expression Evaluation

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Reverse-Polish Notation

Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

The operator is placed between to operands

One weakness: parentheses are required

$$(3 + 4) \times 5 - 6 = 29$$

$$3 + 4 \times 5 - 6 = 17$$

$$3 + 4 \times (5 - 6) = -1$$

$$(3 + 4) \times (5 - 6) = -7$$

Parentheses

- Evaluate 2+3*5.
- + First:

  (2+3)*5 = 5*5 = 25
- * First:

  2+(3*5) = 2+15 = 17
- Infix notation requires Parentheses.

What about Prefix Notation?

- + 2 * 3 5 =

  = + 2 * 3 5

  = + 2 15 = 17
- * + 2 3 5 =

  = * + 2 3 5

  = * 5 5 = 25
- No parentheses needed!

Postfix Notation

- 2 3 5 * + =

  = 2 3 5 * +

  = 2 15 + = 17
- 2 3 + 5 * =

  = 2 3 + 5 *

  = 5 5 * = 25
- No parentheses needed here either!

Conclusion:

- Infix is the only notation that requires parentheses in order to change the order in which the operations are done.

Fully Parenthesized Expression

- A FPE has exactly one set of Parentheses enclosing each operator and its operands.
- Which is fully parenthesized?

(A + B) * C

((A + B) * C)

((A + B) * (C))

Reverse-Polish Notation

When we place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3 \;\; 4 \;\; + \;\; 5 \;\; \times \;\; 6 \;\; -$$

Parsing reads left-to-right and performs any operation on the last two operands:

$$3 \;\; 4 \;\; + \;\; 5 \;\; \times \;\; 6 \;\; -$$

$$7 \;\; 5 \;\; \times \;\; 6 \;\; -$$

$$35 \;\; 6 \;\; -$$

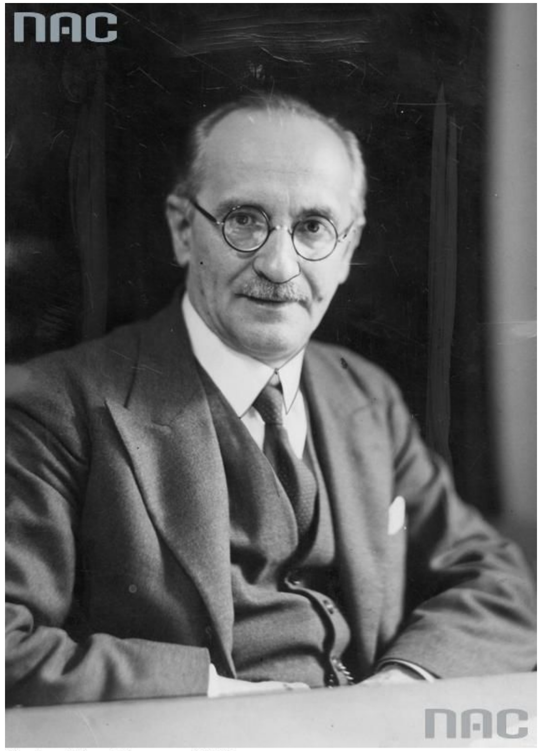
$$29$$

Reverse-Polish Notation

This is called *reverse-Polish* notation after the mathematician Jan Łukasiewicz

- this forms the basis
  of the recursive stack used on all processors

He also made significant contributions to logic and other fields



http://www.audiovis.nac.gov.pl/

http://xkcd.com/645/

Reverse-Polish Notation

Benefits:

- No ambiguity and no brackets are required
- It is the same process used by a computer to perform computations:
  - operands must be loaded into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

((A + B) * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

(+ A B * (C + D))

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B (C + D)

Infix to Prefix Conversion

Move each operator to the left of its operands & remove the parentheses:

* + A B + C D

Order of operands does not change!

Infix to Postfix

(((A + B) * C) - ((D + E) / F))

A B + C * D E + F / -

- Operand order does not change!
- Operators are in order of evaluation!

Computer Algorithm
FPE Infix To Postfix

- Assumptions:

1. Space delimited list of tokens represents a FPE infix expression
2. Operands are single characters.
3. Operators +,-,*,/

FPE Infix To Postfix

- Initialize a Stack for operators, output list
- Split the input into a list of tokens.
- for each token (left to right):

    if it is operand: append to output

    if it is '(': push onto Stack

    if it is ')': pop & append till '('

Infix to Postfix

- Initialize a Stack for operators, output list
- Split the input into a list of tokens.
- for each token (left to right):

    if it is operand: append to output

    if it is '(': push onto Stack

    if it is ')': pop & append till '('

    if it is '+-*/':

     while peek has precedence ≥ it:

        pop & append

     push onto Stack

  pop and append the rest of the Stack.

- Create an empty stack for keeping operators. Create an empty list for output.
- Split the input into a list of tokens.
- Scan the token list from left to right.
  - If the token is an operand, append it to the end of the output list.
  - If the token is a left parenthesis, push it on the opstack.
  - If the token is a right parenthesis, pop the opstack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
  - If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.
- When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list.

Infix to Postfix Algorithm

Input: a infix expression string

Output: a postfix expression string of given input

Steps:

```
Let P is string, and S is character stack
While(input expression has more characters)
  Read the next character
   If character is an operand
      append it to P
   Else If character is an operator
      pop S, until top of the S has an element of lower precedence
          append popped character to P
      Then push the character to S
   Else If character is "("
      Push it to S
   Else If character is  ')'
     pop S until we find the matching '('
       append popped character to P
     pop "("                        // '(' has the lowest precedence when in the stack but has the highest precedence when in the input
 End If
End While
Pop until the stack is empty and append popped character to P
```

Keep in Mind:

1. Relative order of variables is not changed
2. No parenthesis in postfix
3. Operators are arranged according to precedence
4. If same precedence operators, then evaluate left to right

Examples are given in infix-to-postfix slides

FPE Infix to Postfix

(((A + B) * (C - E)) / (F + G))

- stack: <empty>
- output: []

FPE Infix to Postfix

((A + B) * (C - E)) / (F + G))

- stack: (
- output: []

FPE Infix to Postfix

(A + B) * (C - E)) / (F + G))

- stack: ((
- output: []

FPE Infix to Postfix

A + B) * (C - E)) / (F + G))

- stack: (((
- output: []

FPE Infix to Postfix

+ B) * (C - E)) / (F + G))

- stack: (((
- output: [A]

FPE Infix to Postfix

B) * (C - E)) / (F + G))

- stack: (((+
- output: [A]

FPE Infix to Postfix

) * (C - E)) / (F + G))

- stack: (((+
- output: [A B]

FPE Infix to Postfix

* (C - E)) / (F + G))

- stack: ((
- output: [A B +]

FPE Infix to Postfix

(C - E)) / (F + G))

- stack: ((*
- output: [A B +]

FPE Infix to Postfix

C - E)) / (F + G))

- stack: ((* (
- output: [A B +]

FPE Infix to Postfix

- E)) / (F + G))

- stack: ((* (
- output: [A B + C]

FPE Infix to Postfix

E)) / (F + G))

- stack: ((* (-
- output: [A B + C]

FPE Infix to Postfix

)) / (F + G))

- stack: ((* (-
- output: [A B + C E]

FPE Infix to Postfix

) / (F + G))

- stack: ((*
- output: [A B + C E -]

FPE Infix to Postfix

/ (F + G))

- stack: (
- output: [A B + C E - *]

FPE Infix to Postfix

(F + G))

- stack: (/
- output: [A B + C E - *]

FPE Infix to Postfix

F + G))

- stack: (/ (
- output: [A B + C E - *]

FPE Infix to Postfix

+ G))

- stack: (/ (
- output: [A B + C E - * F]

FPE Infix to Postfix

G))

- stack: (/ (+
- output: [A B + C E - * F]

FPE Infix to Postfix

))

- stack: (/ (+
- output: [A B + C E - * F G]

FPE Infix to Postfix

)

- stack: (/
- output: [A B + C E - * F G +]

FPE Infix to Postfix

- stack: <empty>
- output: [A B + C E - * F G + /]

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Evaluation a postfix expression

- Each operator in a postfix string refers to the previous two operands in the string.
- Suppose that each time we read an operand we **push** it into a stack. When we reach an operator, its operands will then be top two elements on the stack
- We can then **pop** these two elements, perform the indicated operation on them, and **push** the result on the stack.
- So that it will be available for use as an operand of the next operator.

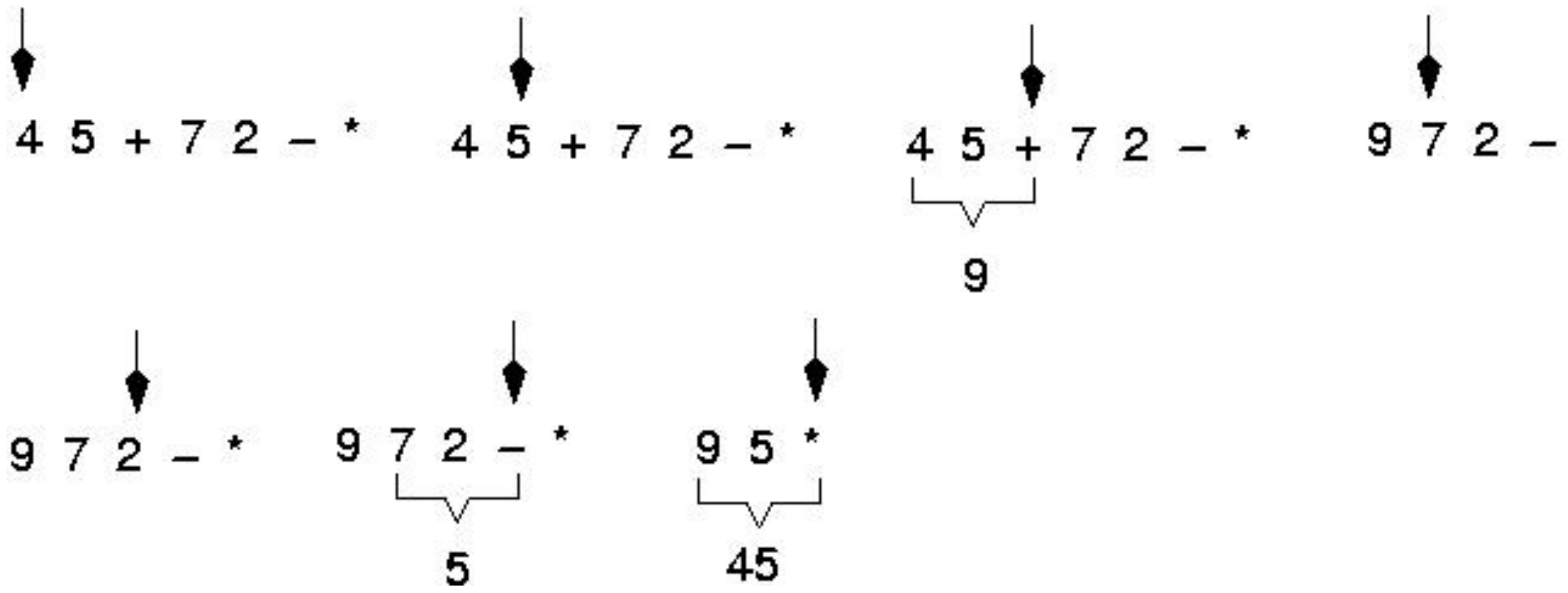
Evaluating Postfix Notation

- Use a stack to evaluate an expression in postfix notation.
- The postfix expression to be evaluated is scanned from left to right.
- Variables or constants are pushed onto the stack.
- When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.

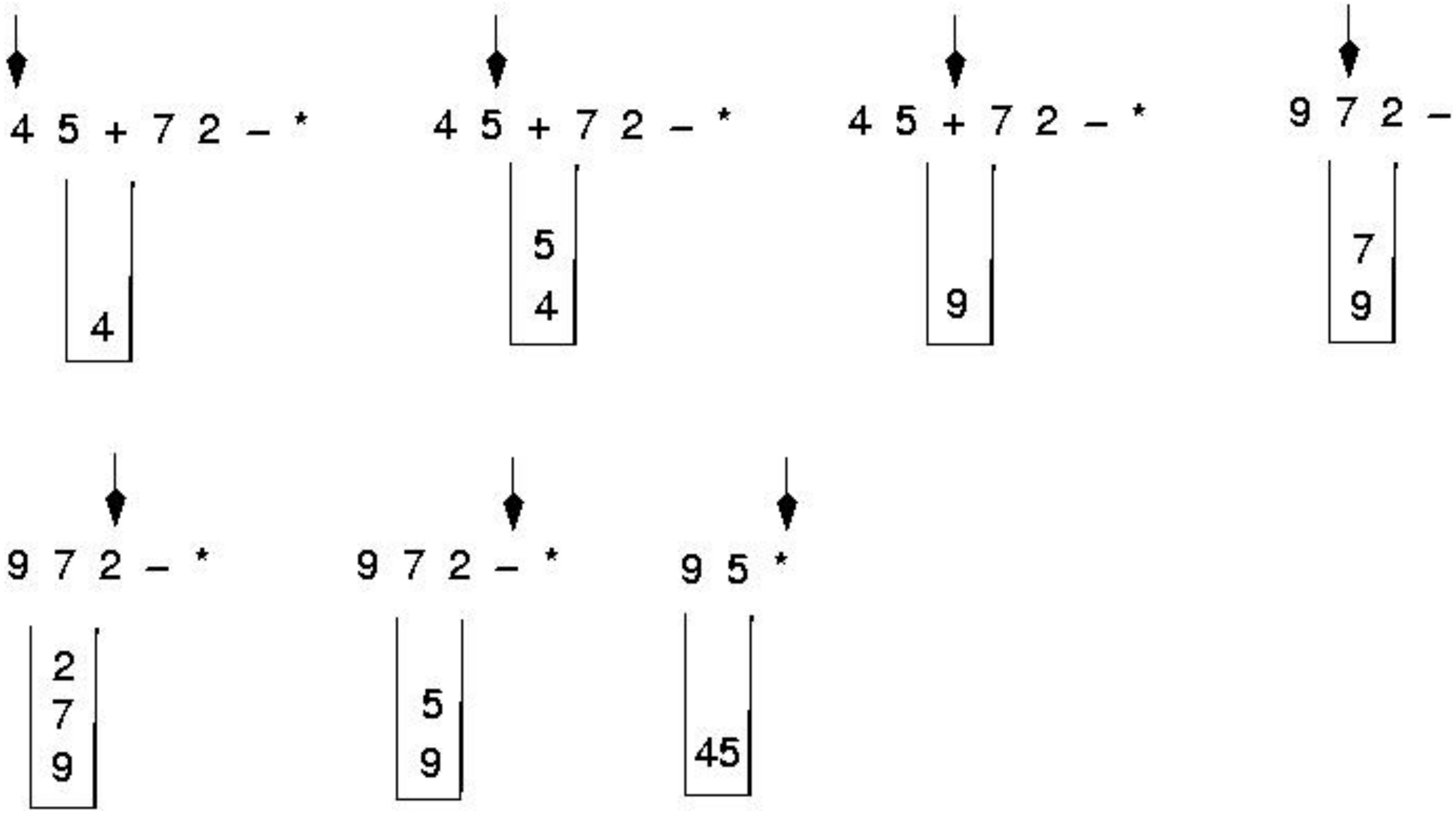
Evaluating a postfix expression

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack,apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Example: postfix expressions (cont.)



Postfix expressions: Algorithm using stacks (cont.)



Algorithm for evaluating a postfix expression (Cond.)

```
WHILE more input items exist
{
  If symb is an operand
    then push (opndstk,symb)
  else   //symbol is an operator
  {
    Opnd1=pop(opndstk);
    Opnd2=pop(opndnstk);
    Value = result of applying symb to opnd1 & opnd2
    Push(opndstk,value);
  }       //End of else
} // end while
Result = pop (opndstk);
```

Question : Evaluate the following expression in postfix :
623+-382/+*2^3+

Final answer is

- 49
- 51
- 52
- 7
- None of these

Parsing Reverse-Polish Notation

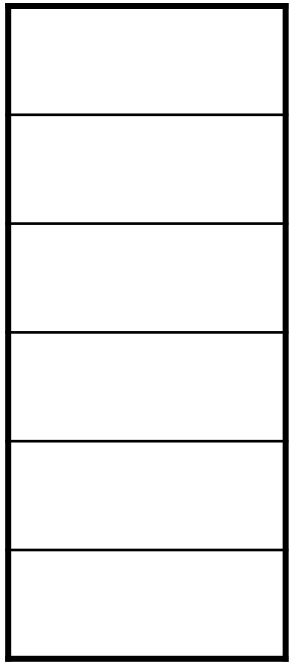
The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

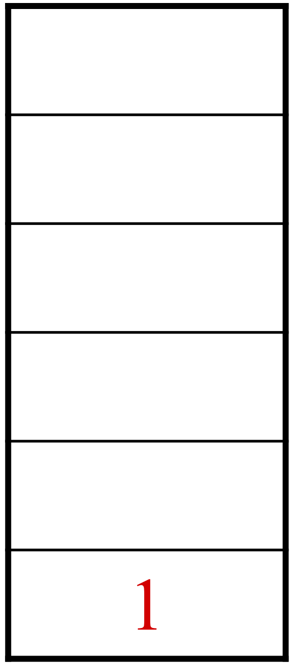
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

Push 1 onto the stack

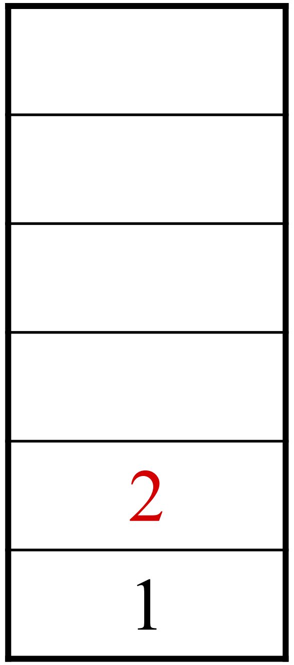
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Push 1 onto the stack

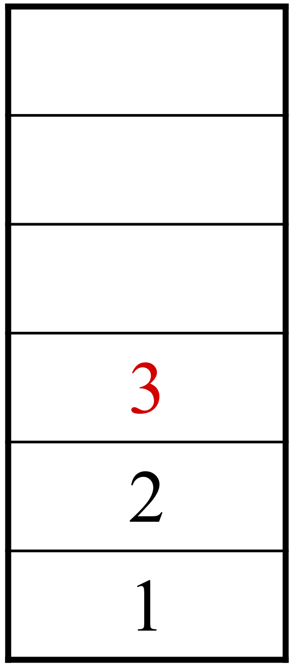
$$1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$$



Reverse-Polish Notation

Push 3 onto the stack

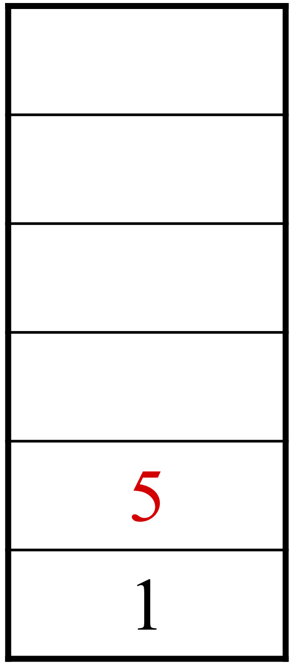
$$1 \quad 2 \quad {\color{red}3} \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Pop 3 and 2 and push $2 + 3 = 5$

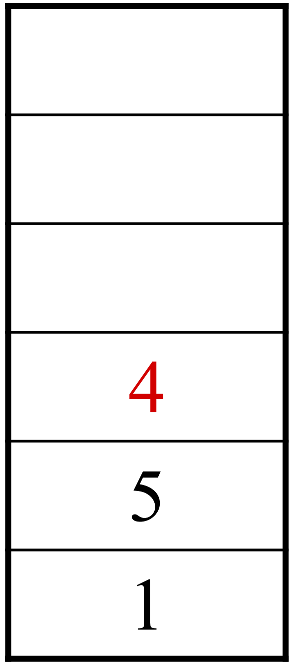
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Push 4 onto the stack

$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$



Reverse-Polish Notation

Push 5 onto the stack

$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; \textcolor{red}{5} \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

5
4
5
1

Reverse-Polish Notation

Push 6 onto the stack

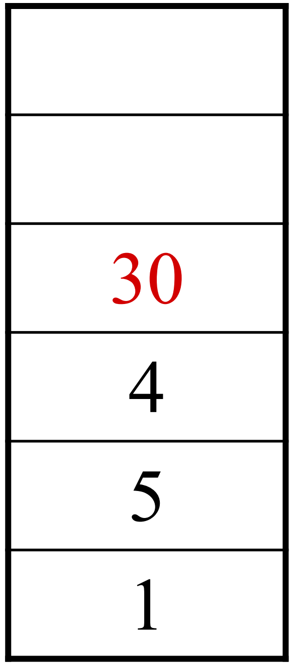
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

6
5
4
5
1

Reverse-Polish Notation

Pop 6 and 5 and push $5 \times 6 = 30$

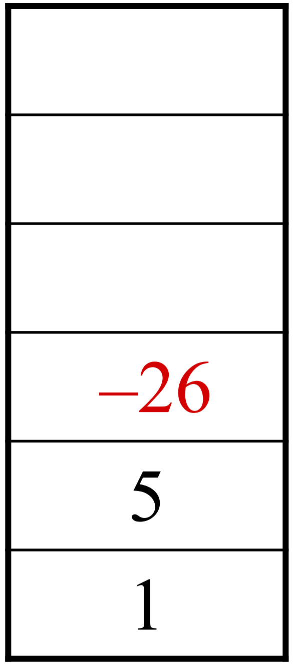
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Reverse-Polish Notation

Pop 30 and 4 and push $4 - 30 = -26$

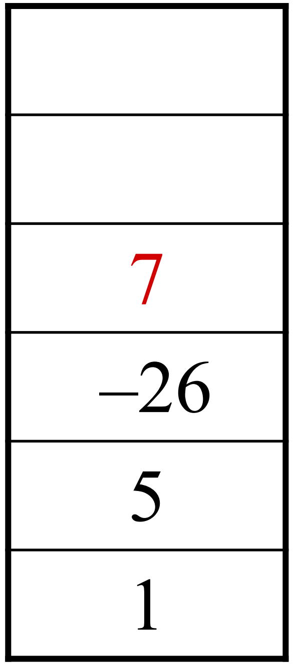
$$1\ \ 2\ \ 3\ \ +\ \ 4\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$



Reverse-Polish Notation

Push 7 onto the stack

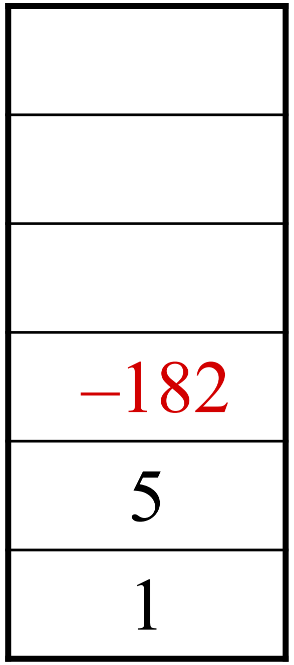
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 7 and -26 and push $-26 \times 7 = -182$

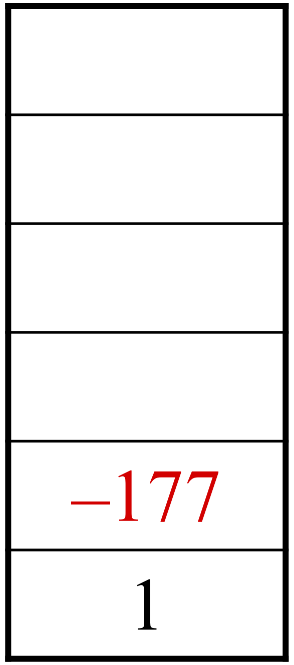
$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$



Reverse-Polish Notation

Pop –182 and 5 and push –182 + 5 = –177

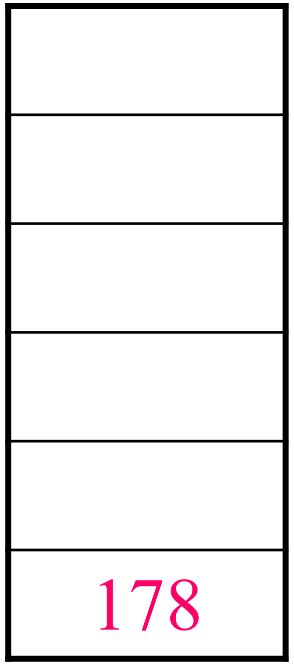
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop -177 and 1 and push $1 - (-177) = 178$

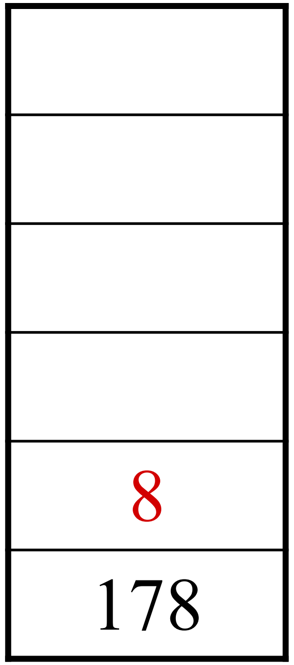
$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$



Reverse-Polish Notation

Push 8 onto the stack

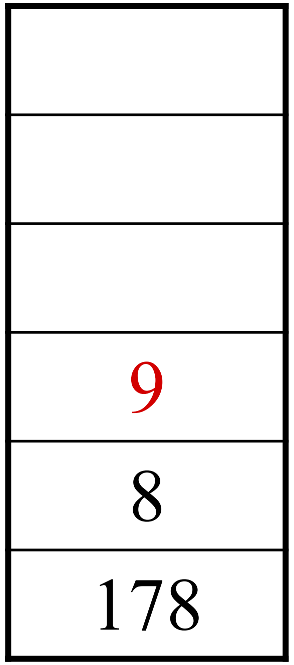
$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$



Reverse-Polish Notation

Push 1 onto the stack

$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$



Reverse-Polish Notation

Pop 9 and 8 and push $8 \times 9 = 72$

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

72
178

Reverse-Polish Notation

Pop 72 and 178 and push $178 + 72 = 250$

$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$