COMP 2310
Data Structures and Algorithms

Lecture 4

Stacks


How can you remove the 4th stone ?

Linear Data Structures: Stacks

- Linear data structure: Data elements are in sequential order →One follows the other physically or logically.

  Examples: Arrays, stacks, lists, queues.

- A stack is a very common abstract data type that insertions and deletions can be made only at one end.
  - This end is called the top of stack. Storing and retrieving data is performed only at the top position.

Stack Operations

- Consider a physical analogy: A stack (Pile) of books. We can do the following set of things with this stack:
  - Check to see if we have space for adding more books.
  - Add a New book: Place a book on the top.
  - Remove one book off the top.
  - Remove all books: Empty the stack.

A stack of books

Abstract Data Type: `Stack`

- A finite number of objects
  - Having the same data type
- A set of operations
- Note that ADT Concerns only on the concept or model, it does not concern implementation details
- When we define implementation details it becomes a data structure.

Stack Operations

We have stack functions that perform the following operations:

create() : Create an empty stack

push(x) : Put x on top of the stack.

pop() : Remove top element of the stack

clear() : Clear the stack.

isEmpty() : Check to see if the stack is empty.

isFull() : Check to see if the stack is full.

Stack Sructure

- Since the contents of a stack data structure are stored in linear form → Linear data structure

- Arrays are also linear structures and any element in an array can be accesed directly.

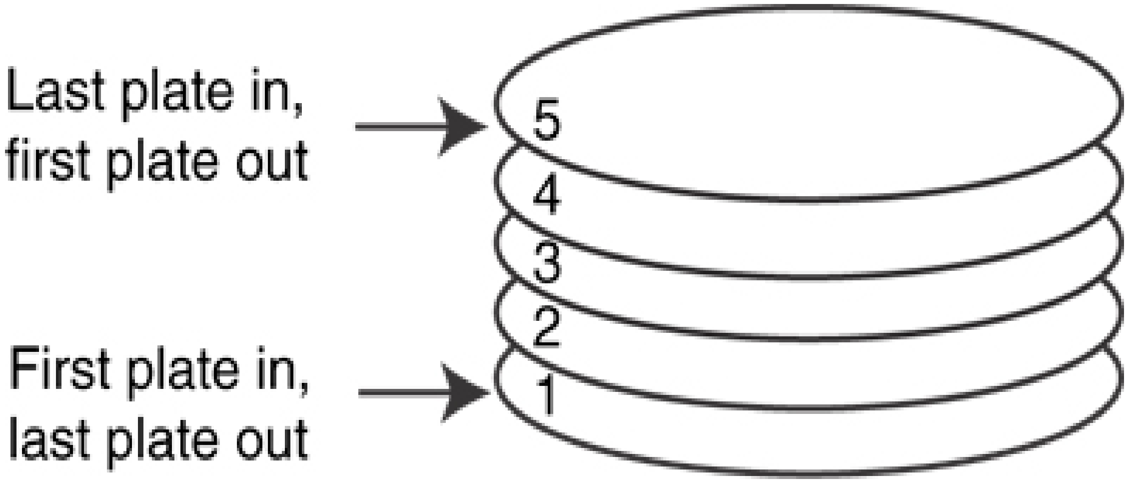
- But in stacks we can only access the top element.

- The elements of a stack move according to LIFO principle:

  Last **I**n **F**ist **O**ut (Or FILO)

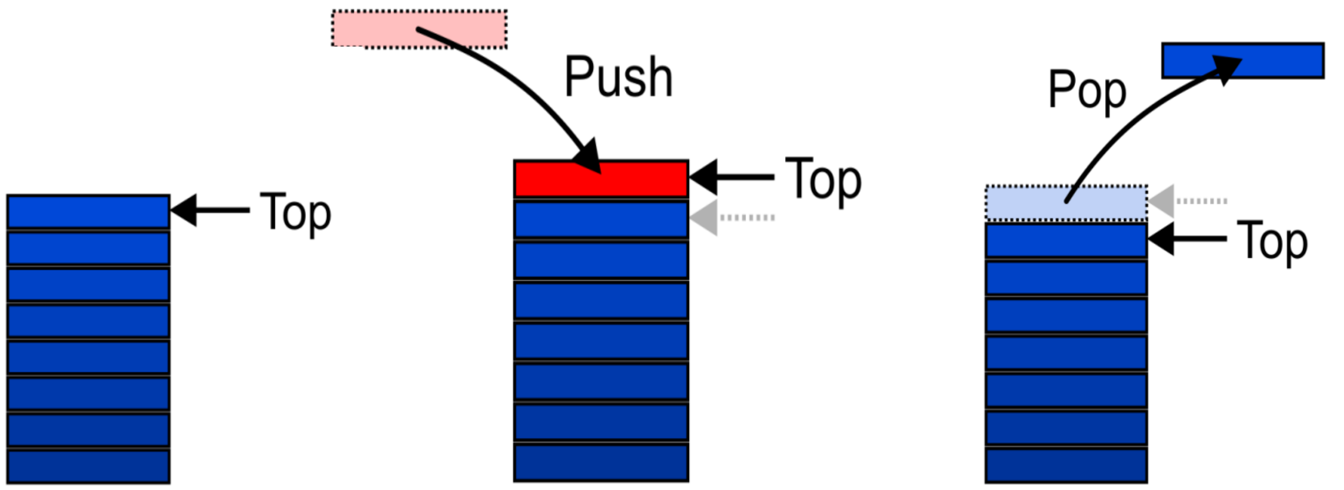
A LIFO Structure

A stack of plates :



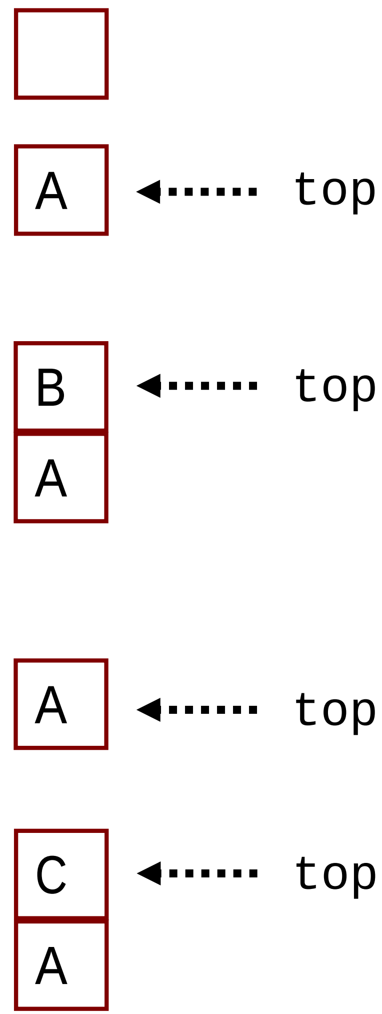
Stack Operations: LIFO

Graphically, we may view these operations as follows:



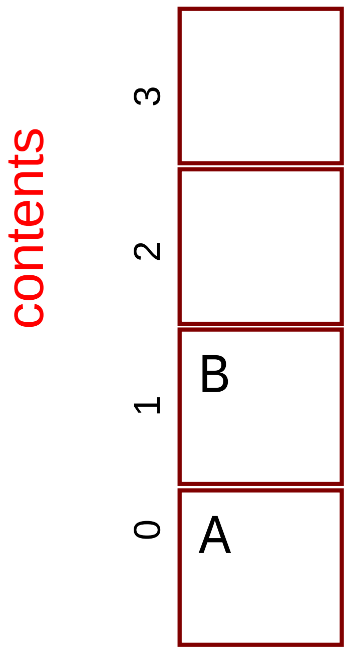
Stack Operations: Illustration

```
stack empty

stack.push(A)

stack.push(B)

letter=stack.pop()

stack.push(C)
```



Implementation of stack structures

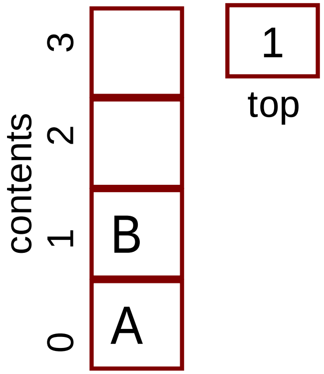
An array can be used to implement stack structure:



- Consider an array of characters called `contents` and size of 4.
- Adding first A and then B into the stack will be represented in an array as in the figure.

→`contents[0]=A, contents[1]=B,`…

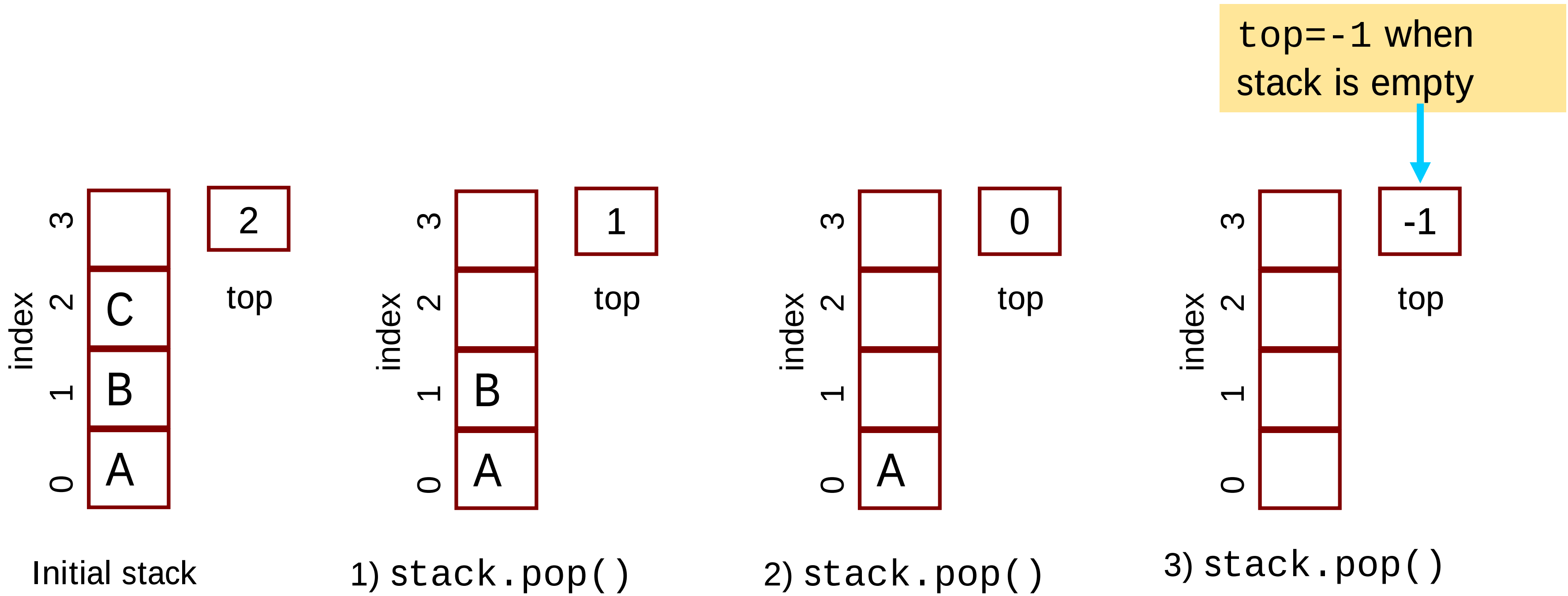
Array implementation of stacks



- We need to keep track of the *top* of the stack since not all of the array holds stack elements (Part of it is empty)
- We use an integer, `top`, which will hold the array index of the element at the top of the stack.

Array implementation of stacks: Pop

Results of a sequence of pop operations

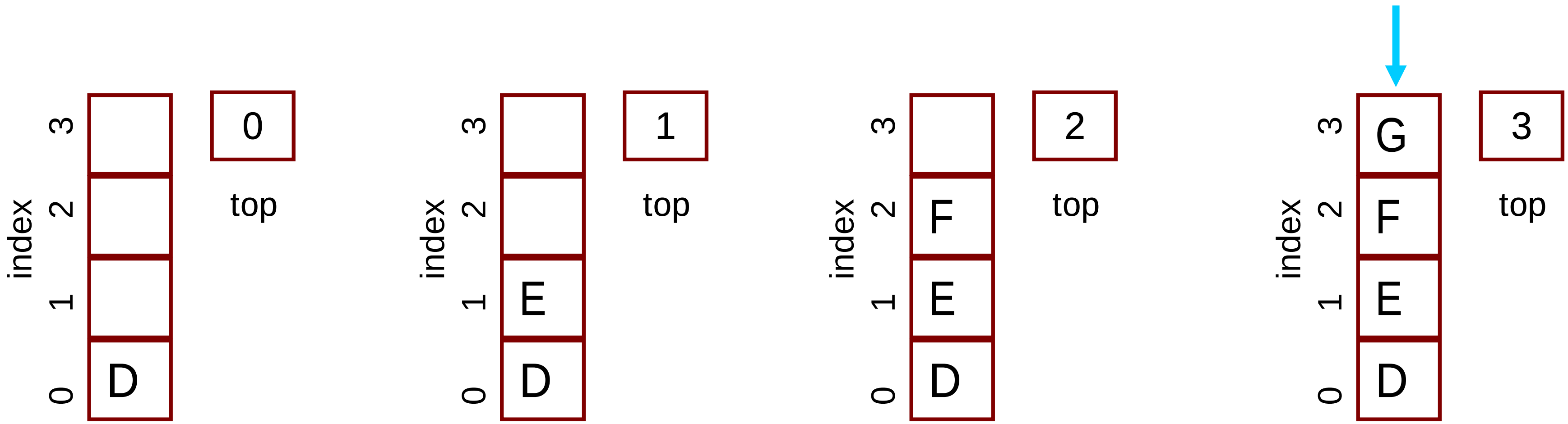


Array implementation of stacks: Push

Suppose stack capacity is 4.
Continue with the following push operations

1) `stack.push(D)`
2) `stack.push(E)`
3) `stack.push(F)`
3) `stack.push(G)`

`top=3` when stack is full



`stack.push(D)` `stack.push(E)` `stack.push(F)` `stack.push(G)`

Array implementation of stacks: Disadvantage

- If we try to push the letter H the stack will not accept it since it is full.
- A major disadvantage of using static arrays!
- We need to be able to decide the maximum size of the stack at run time.
- We must use a pointer to a dynamically-allocated array instead of a regular array.

Array implementation of Stacks: IsFull

- Suppose that we implement a stack of at most n elements with an array *S.*

IsFull ()

if $top \geq n$-1

return TRUE

else

return FALSE

Array implementation: IsEmpty

```
// Is the stack empty?
IsEmpty()
  if top = -1
     return TRUE
  else
     return FALSE
```

Array implementation: Push

//The new value x is pushed (added) to the top of S

Push(x)

if IsFull()

error "Stack is full"

else

$top \leftarrow top+1$

$S[top] \leftarrow x$

Array implementation: Pop

//The top element in S is removed and sent to call point

Pop()

if IsEmpty()

error "Stack is empty" //Exit

else

$top \leftarrow top$-1

Complexities of stack operations: Assume n elements

Filling the stack : O(n)

Emptying the stack : O(n)

Stack push and pop : O(1)

Stack Applications

- There are many computer related application areas of stacks.

- In this sections we present the following problems that can be solved by using stacks.
  - Reversing a word or text.
  - Balancing symbols in an expression.
  - Infix to postfix conversion.
  - Evaluating postfix expressions.
  - ……………………….

Reversing a word

We want to reverse a given **word** using a stack.

EX : STACK → KCATS

Solution Method:

- First the characters are extracted one by one from the input string
- Next they are pushed onto the stack.
- Then they're popped off the stack and displayed.

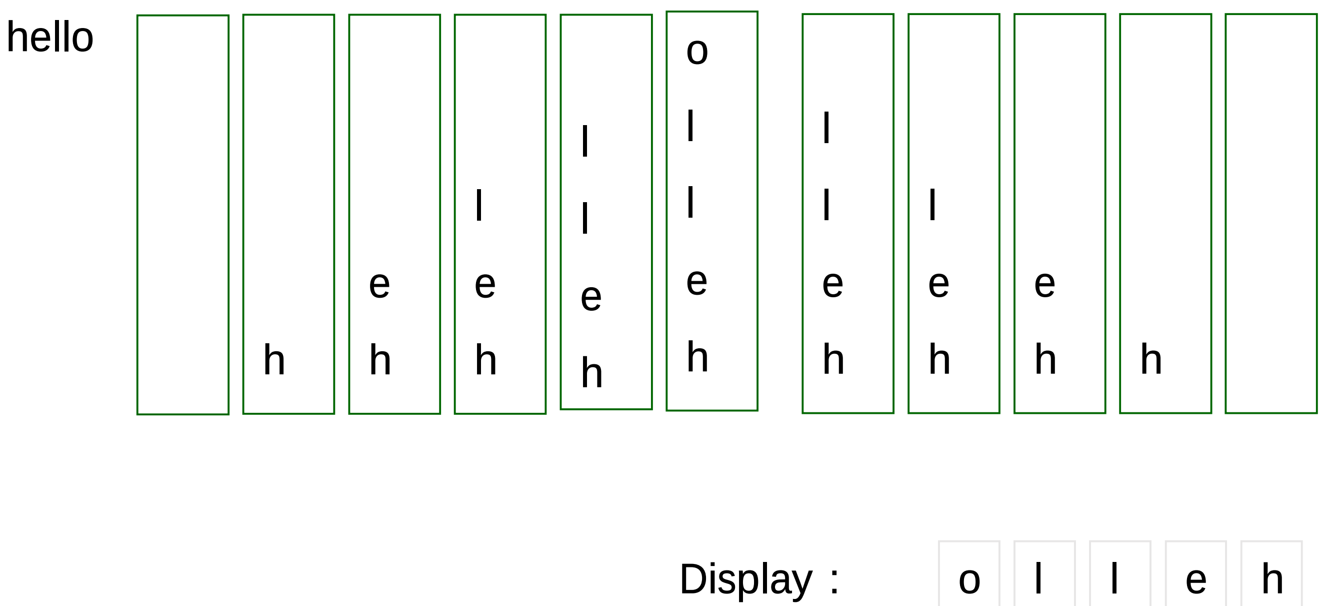
Pseudocode for Reversing a Word

```
// Uses two functions: char_at_ pos_i  and display
S ← createStack( )
ReverseWord(W)
  for (i ← 0 to lengthOfW-1) do
      ch ← char_at_ pos_i
      Push(ch )
  for (j ← lengthOfW-1 to 0) do
       display S[j]
       Pop()
```

Reversing a word

Example: Trace of the algorithm that checks for reversing the word “hello”

Extract the characters and push them to the stack. Then pop stack contents



1. Push h
2. Push e
3. Push l
4. Push l
5. Push o
6. Pop o
7. Pop l
8. Pop l
9. Pop e
10. Pop h

Balancing Symbols in Expressions

- Searching for syntax errors in a computer program is a common problem.
- Every left bracket must be paired with its right counterpart.
- For example, the sequence ((( )) is wrong whereas the sequence ((( ))) is correct.
- For simplicity, we check for balancing of parantheses and ignore any other character in the algorithm fragment.
- A simple solution to this problem: Use stacks.

How to Check Balance?

- ([]({()}[()])) is balanced; ([]({()}[())]) is not.
- Simple counting is not enough to check balance
- You can do it with a stack: going from left to right

  - If you see a (, [, or {, push it on the stack
  - If you see a),], or }, pop the stack and check whether you got the corresponding (, [, or {
  - When you reach the end, check the stack state:

    Stack is empty → Balanced

    Stack is not empty → Not balanced

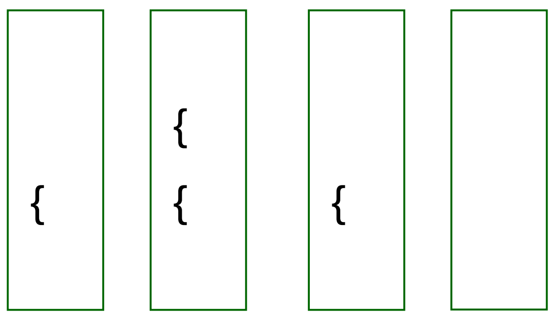
Example:
Input: exp = “[()]{}{[()]()}”
 Output: Balanced
Input: exp = “[( ])”
 Output: Not Balanced

Balancing Symbols in Expressions: Algorithm

- Steps of the algorithm:
  - Start with an empty stack
  - Read all characters until the end of the expression
    - Push the character onto the stack if it is an opening symbol.
    - If it is a closing symbol, pop the stack. At any point, if the stack becomes empty while there is a closing symbol report an error.
    - If the symbol just popped does not match with a closing symbol then report an error.
  - When end-of-expression is reached, if the stack is not empty, report an error.

Balancing Symbols in Expressions

Trace of the algorithm that checks for the balanced braces in the expression: {a { b } c}



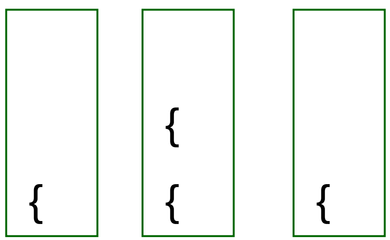
1. Push {
2. Push {
3. Pop
4. Pop

Stack is empty, braces are balanced.

Balancing symbols

Trace of the algorithm that checks for the balanced braces:

{a { b c}



1. Push {
2. Push {
3. Pop

Stack is not empty, braces are not balanced.

Example: Pseudocode for Balancing Symbols

```
//Algorithm checks balance for only braces : { , }. More symbols can be added
S ← createStack( )  // str is the expression string
CheckBalance (str)
  balanced ←TRUE
  i ← 0
  while (balanced  and  i< lengthOfstr) do
          ch ← str(i)
          i ← i+1
          if  ch ="{"
                Push({ )
          else if  ch ="}" do
                if IsEmpty() = FALSE and S(top)== "{"  //No problem
                      Pop()
                else
                      balanced ←FALSE  // Not balanced
  If balanced and IsEmpty()
     return TRUE   // String is balanced
  else
     return FALSE   // String is not balanced
```

Evaluating Expressions: Different Notations

- Is it possible to simplify expression evaluation by removing parantheses?
- Consider the expression: 5*(8+9)
- This form of expression is called the infix form
- Programming language compilers use another form for evaluating expressions: The postfix form.

  (Another variation is called prefix form)
- Expressions in postfix form are easier for a computer to evaluate than expressions in infix form.

  →No need for parantheses !
- What is postfix form and how can we convert an expression from infix to postfix form?

Infix and Postfix Forms

5 + 8*9 Expression is in Infix form: Result=77

Consider this ordering:

5 8 9 * + Expression in postfix form: Result=77

In general we have :

Infix form: Operand, operator, operand→ a / b , a+b , …

Postfix form: Operand, operand, operator→ a b / , ab+, …

Notice that the order of operands remain the same!

For evaluating infix we need to know precedence of the operators.

Infix and Postfix Forms

Evaluating Postfix for 5 + 8*9 :

An operator in a postfix expression applies to two operands that immediately preceed it:

5 8 9 * + → 5 72 + → 77

Infix to postfix conversion rules:

Precedence of the operators:

1. ( ) *highest*
2. * /
3. + - *lowest*

No two operators with the same precedence can be placed one after another in the stack

An operator of low precedence cannot be pushed onto an operator of high precedence

Infix to postfix conversion:

Infix expression ⟹ Postfix expression

If parantheses exist in infix expression:

Start from the innermost parenthesis and move outward by handling parantheses one by one.

Apply precedence rules to operators.

a+(b+c)	⟹	abc++
(a+b)+c	⟹	ab+c+
a-b*c	⟹	abc*-
(a/b)*(c/d)	⟹	ab/cd/*
a/(b+c*d-e)	⟹	abcd*+e-/
a-b*c+d/e	⟹	abc*-de/+

Postfix Expressions: Examples

Infix	Postfix	Value
5 + 4	5 4 +	9
5 + 4 * 2	5 4 2 * +	13
(5 + 4) * 2	5 4 + 2 *	18
6* ((5+ (2+3) *8)+3)	6523+8*+3+*	288

Converting Infix to Postfix Form: Algorithm

Steps of the conversion algorithm:

- Start with an empty stack
- Place an operand onto the output when it is read.
  (Finally, Output will contain the postfix form.)
- Push an operator (except the right paranthesis) onto the stack when it is read. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
- Pop the stack when a right paranthesis is encountered writing all symbols onto the output until a matching left paranthesis is seen and pop and discard this paranthesis.
- If one of the symbols **+,*, - , /,(** is seen, pop entries from the stack until an entry of lower priority is found.
- If end of the input is reached pop the stack until it is empty, writing symbols onto the output (Discard the pair of parentheses).

Infix to postfix conversion: Example

Assume the operators are : +, *, ( )

Convert the following infix expression into postfix form.

E=a+b*c+(d*e+f)*g

Infix to postfix conversion: Pseudocode

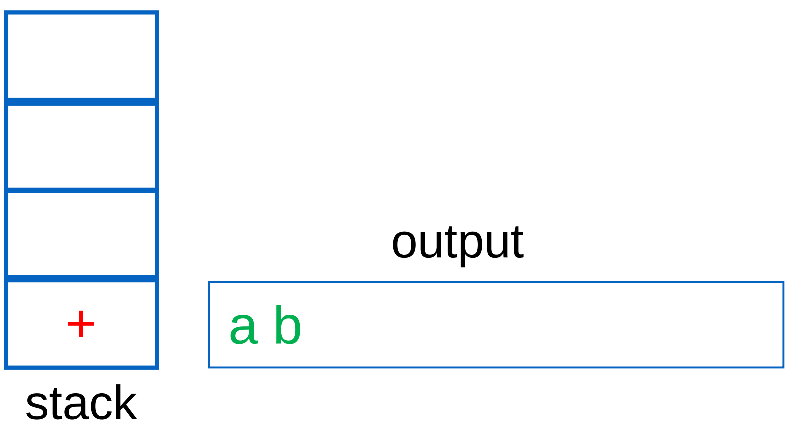
```
s ←createStack
Postfix(expr)     //expr : Given expression in infix form
 while(more symbols)
   x←next symbol   // Extract next symbol from expr
   if(x == operand )
      output x   //Append to output string
   else
     while(precedence(x) <= precedence(top(s)))
        output (pop())   //Except brackets
      push(x) //Push x to stack s
 while(! isEmpty())
   output(pop())   //Remaining stack contents,except (
//Output will be the postfix expression now.
```

Infix to postfix conversion: Example

E= a+b*c+(d*e+f)*g

Conversion order :

Output a
Push +
Output b
What to do with * ?



Infix to postfix conversion

a+b*c+(d*e+f)*g

since the symbol + (top entry) has lower precedence than *, * is pushed onto stack. Next, c is output

*
+

stack

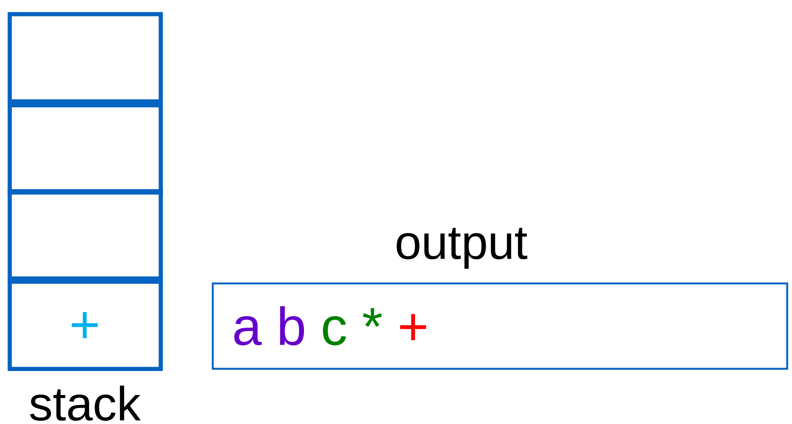
output

a b c

Evaluating postfix expressions

E= a+b*c+(d*e+f)*g

What to do with + ? Since + has lower precedence than *, * is popped from stack.
The previous + , which has not lower but equal priority, + is also popped. * and + are put on the output.
Then next (second) + is pushed



Infix to postfix conversion

a+b*c+(d*e+f)*g

Next symbol (has the highest precedence, so it is pushed onto the stack.
Next, d is output.

stack
(
+

output

a b c * + d

Infix to postfix conversion

a+b*c+(d*e+f)*g

Next symbol *. Don't pop (, it stays in the stack until the closing bracket is read.
Push * , place e on the output.

stack
*
(
+

output

a b c * + d e

Infix to postfix conversion

a+b*c+(d*e+f)*g

Next, + is read, * is popped and output (It has higher priority than +)
Then + is pushed and f is output.

stack
+
(
+

output

a b c * + d e * f

Infix to postfix conversion

a+b*c+(d*e+f)*g

) is read, stack popped back until the opening (.
output +

+

stack

output

a b c * + d e * f +

Infix to postfix conversion

a+b*c+(d*e+f)*g

* is read, pushed onto the stack and g is output.

*
+

stack

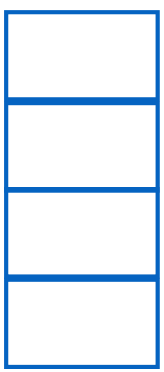
output

a b c * + d e * f +g

Infix to postfix conversion

a+b*c+(d*e+f)*g

End of the input is reached.
Pop and output remaining symbols * and + from the stack.
Output string is now in postfix form.



stack

output

a b c * + d e * f +g * +

Notice that the postfix form does not include parentheses.

Another Example

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

How do Compilers Evaluate Expressions?

- A compiler which translates a program, has to generate machine instructions to evaluate expressions.
- Stacks are used by compilers to evaluate expressions and generate machine code.
- Compiler first converts each infix expression into postfix form.
- The reason is that, postfix expressions are parenthesis-free and precedence rules are not needed for evaluation.
- The internal evaluation of expressions is performed on the postfix form in just one pass.
- The complexity is linear time, n is the expression length.

Evaluating Postfix Expressions

Numeric postfix expressions can be evaluated by using stacks as follows:

- When a number is seen it is pushed onto the stack
- When an operator is seen it is applied to two preceeding numbers in such a way that these numbers are popped and the corresponding result is pushed on to the stack.

Evaluating Postfix Using Stacks: Algorithm

Informal algorithm:

- Read all the symbols in the given Postfix Expression one by one from left to right
- If next symbol is operand, then push it on to the Stack.
- If next symbol is operator, then perform TWO pop operations and store the two popped operands in two different variables: operand1 and operand2.
- Perform the operation using operand1 and operand2 and push result back on to the Stack.
- Finally perform a pop operation and display the popped value as the final result.

Algorithm for Evaluating a Postfix Expression

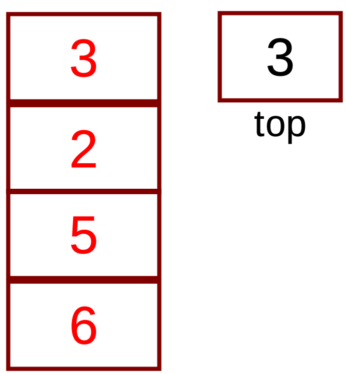
```
S ←createStack
EvalPostfix(Expression)
 symb ←next symbol  // Next symbol is extracted from postfix expression
 WHILE (more symb)
  {
     If symb is an operand
           then push (symb)
     else    //symbol is an operator
     {
           Opnd1=pop();
           Opnd2=pop();
           value = result of applying symb to Opnd1 & Opnd2
           Push(value)
      }
  }
Result = pop ()   //The last value in the stack is the result of expression
```

Evaluating Postfix Expressions:Example

Evaluating a numeric expression using stacks:

Infix: expression = 6*((5+(2+3)*8)+3), Convert to postfix

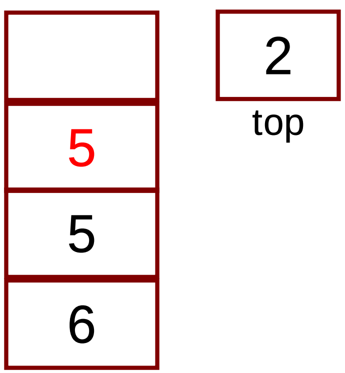
Postfix : expression= 6523+8*+3+*



numbers before + are pushed onto the stack

Evaluating postfix expressions

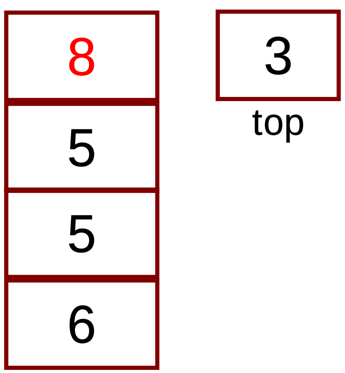
Expression: 6523+8*+3+*



Next is operator +. 3 and 2 are popped and their sum 5 is pushed onto the stack (The order: 2+3)

Evaluating postfix expressions

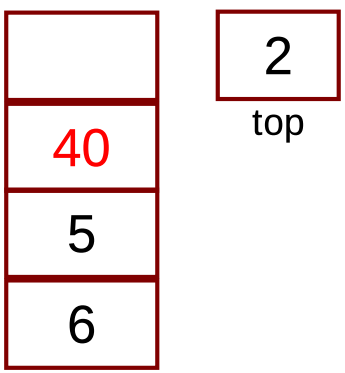
Expression: 6523+8*+3+*



Next operand 8 is pushed onto the stack.
Next operatr is *

Evaluating postfix expressions

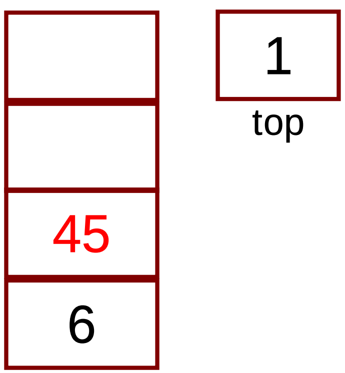
Expression: 6523+8*+3+*



8 and 5 are popped, * is applied and their product 40 is pushed

Evaluating postfix expressions

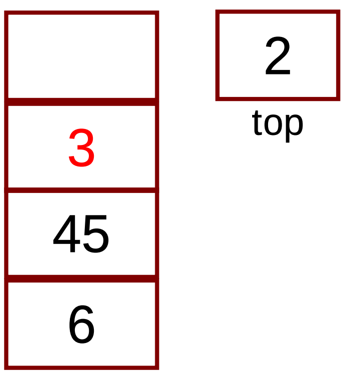
Expression: 6523+8*+3+*



40 and 5 popped, + is applied and their sum 45 is pushed

Evaluating postfix expressions

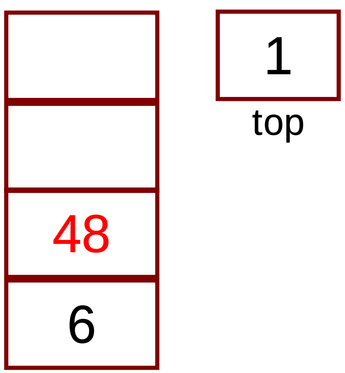
Expression: 6523+8*+3+*



3 is pushed

Evaluating postfix expressions

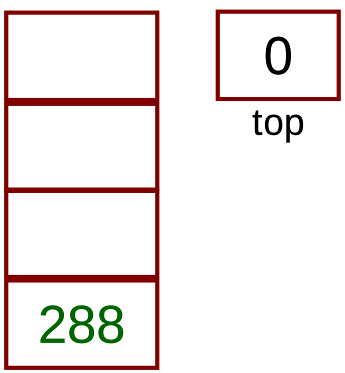
Expression: 6523+8*+3+*



3 and 45 popped, + is applied and their sum is pushed

Evaluating postfix expressions

Expression: 6523+8*+3+*



48 and 6 are popped, * is applied and their product is pushed. Stack now contains one element. This is the result.

Evaluating postfix expressions: Complexity

- Running time for evaluating postfix expression is $O(n)$.
- Because we are running a single stack whose size is n.
- n is determined by the number of operands and operators in the expression.