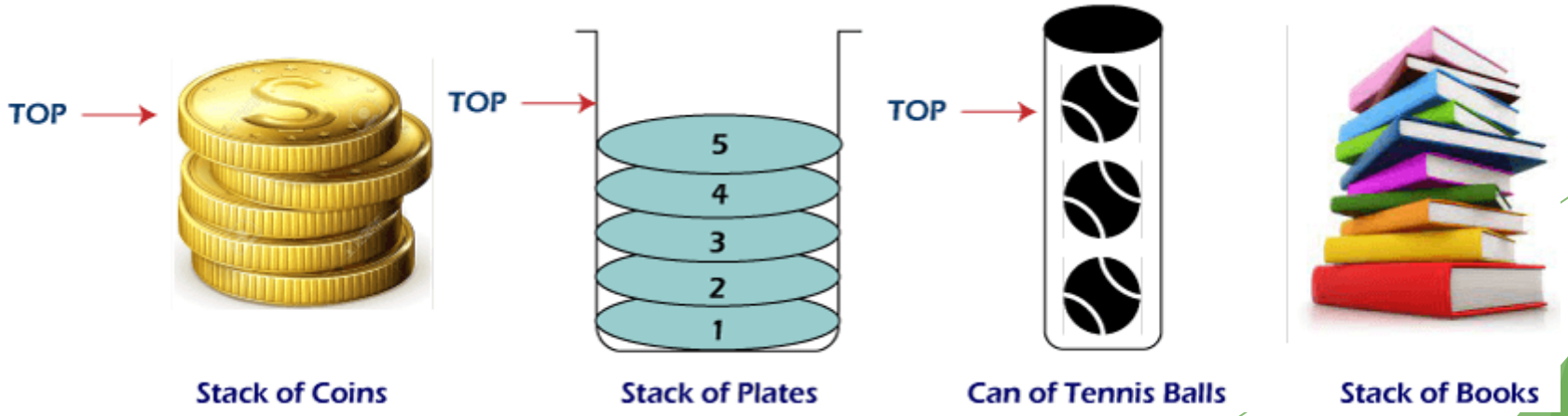
Applications of Stacks & Queues

By Asst Prof Christopher Uz
Course: DSAA
Pillai College of Engineering

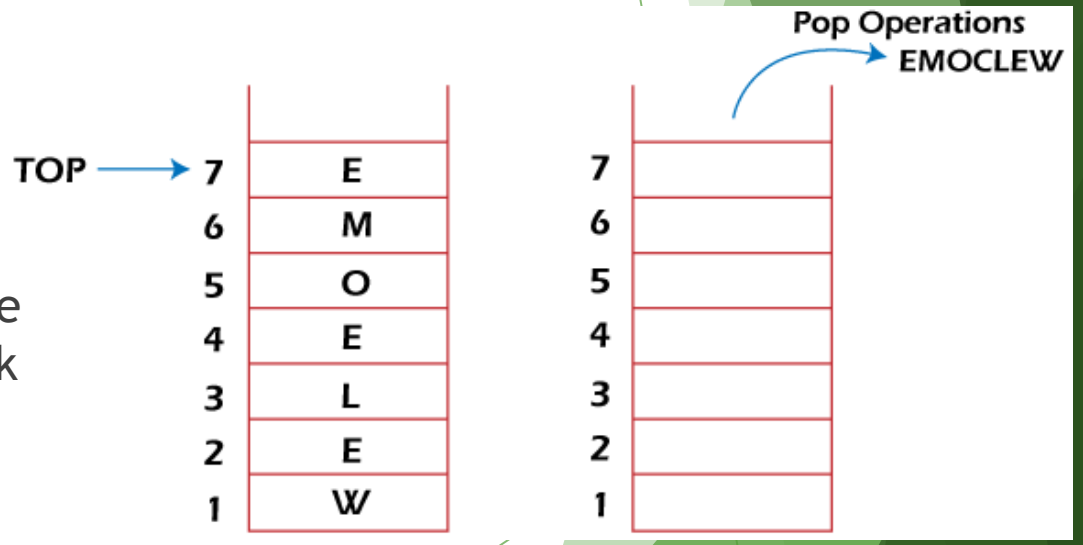
Applications of Stacks

- Reversal of a String
- Delimiter Checking (Well form-ness of Parenthesis)
- Stack Recursion
- Evaluation of Arithmetic Expressions



Reverse a String

- A Stack can be used to reverse the characters of a string
- This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one
- Because of the last in first out property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order



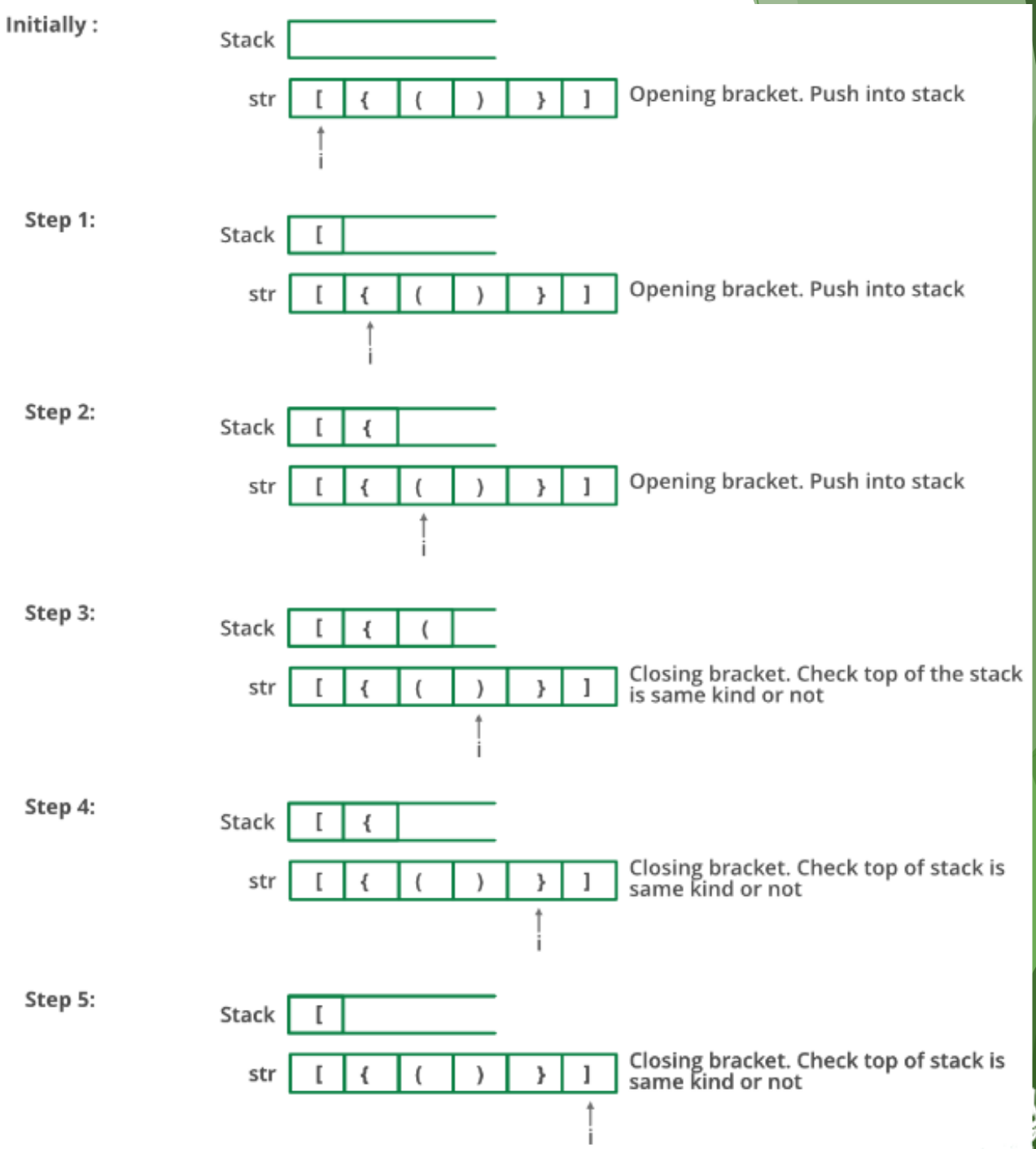
Delimiter Checking (Parenthesis Checking)

- The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically
- When the compiler translates a source program to a machine language one of the main problem encountered while translating is the unmatched delimiters
- We make use of different types of delimiters include the parenthesis checking (,), curly braces {,}, and square brackets [,], and common delimiters /* and */
- Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis
- Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier

Delimiter Checking Algorithm

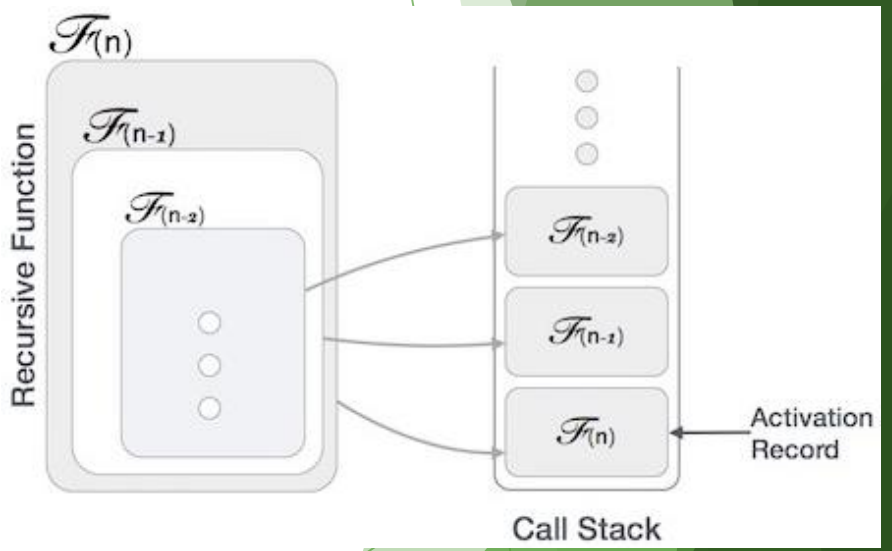
- Declare a character stack S
- Now traverse the expression string exp
- If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack
- If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket, then fine else brackets are not balanced.
- After complete traversal, if there is some starting bracket left in stack then “not balanced”

Delimiter Checking Sample



Stack Recursion

- Many programming languages implement recursion by means of stacks
- Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee
- This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function
- Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on
- For this purpose, an activation record (or stack frame) is created for the caller function
- This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function



Evaluation of Arithmetic Expressions

- A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators
- In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis“
- The precedence rules for the five basic arithmetic operators are:

Operators	Associativity	Precedence
^ Exponentiation	Right to left	Highest followed by *Multiplication and /Division
* Multiplication, / Division	Left to right	Highest followed by +Addition and –Subtraction
+ Addition, - Subtraction	Left to right	Lowest

Notations for Arithmetic Expression

- Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions
- **Infix Notation:**
- The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement
- All these expressions are in infix notation because the operator comes between the operands
- Example: A + B, (C - D) etc.
- **Prefix Notation:**
- The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as Polish Notation
- All these expressions are in prefix notation because the operator comes before the operands
- Example: + A B, -CD etc.

Notations for Arithmetic Expression

- **Postfix Notation:**
- The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish Notation
- All these expressions are in postfix notation because the operator comes after the operands
- Example: AB +, CD+, etc.
- **Conversion of Arithmetic Expression into various Notations:**

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

Conversion of an Infix Expression into a Prefix Expression

- **(A + B) * C**
- (+AB)*C
- *+ABC

- **(A-B) * (C+D)**
- [-AB] * [+CD]
- *-AB+CD

- **(A + B) / (C + D) - (D * E)**
- [+AB] / [+CD] - [*DE]
- [/+AB+CD] - [*DE]
- -/+AB+CD*DE

- The precedence of these operators can be given as follows:
- Higher priority ^,*, /, %
- Lower priority +, -

- *NOTE:* The order of evaluation of these operators can be changed by making use of parentheses

Algorithm to convert an Infix notation to Postfix notation

- Print(add to postfix expression) the operand as they arrive
- If the stack is empty or contains a left parenthesis{‘(’} on top, **push** the incoming operator on to the stack
- If the incoming symbol is ‘(’, **push** it on to the stack
- If the incoming symbol is ‘)’, **pop** the stack and print the operators until the left parenthesis is found
- If the incoming symbol has higher precedence than the top of the stack, **push** it on the stack
- If the incoming symbol has lower precedence than the top of the stack, **pop** and print the top of the stack. Then test the incoming operator against the new top of the stack
- If the incoming operator has the same precedence with the top of the stack then use the *associativity rules*. If the associativity is from left to right, then **pop** and print the top of the stack then **push** the incoming operator. If the associativity is from right to left, then **push** the incoming operator
- At the end of the expression, **pop** and print all the operators of the stack

Input Expression	Stack	Postfix Expression
K		K
+	+	K
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N * -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N * - O
P	+ (^	K L + M N * - O P
)	+	K L + M N * - O P ^
*	+ *	K L + M N * - O P ^
W	+ *	K L + M N * - O P ^ W
/	+ /	K L + M N * - O P ^ W *
U	+ /	K L + M N * - O P ^ W * U
/	+ /	K L + M N * - O P ^ W * U /
V	+ /	K L + M N * - O P ^ W * U / V
*	+ *	K L + M N * - O P ^ W * U / V /
T	+ *	K L + M N * - O P ^ W * U / V / T
+	+	K L + M N * - O P ^ W * U / V / T * +
Q	+	K L + M N * - O P ^ W * U / V / T * + Q
		K L + M N * - O P ^ W * U / V / T * + Q +

Convert the infix expression into postfix expression

K + L - M*N + (O^P) * W/U/V * T + Q

Evaluation of a Postfix Expression (Algorithm)

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Evaluation of a Postfix Expression (Sample)

- Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluate the expression
- The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

Algorithm to convert an infix expression into prefix expression

- Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses
- Step 2: Obtain the postfix expression of the infix expression obtained in Step 1
- Step 3: Reverse the postfix expression to get the prefix expression

Conversion of an Infix Expression into a Prefix Expression (Sample)

- Given an infix expression **(A - B / C) * (A / K - L)**. Find the prefix expression.
- *Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses*
- (L – K / A) * (C / B – A)
- *Step 2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1*
- The expression is: (L – K / A) * (C / B – A)
- Therefore, [L – (K A /)] * [(C B /) – A]
- = [LKA/–] * [CB/A–]
- = L K A / – C B / A – *
- *Step 3: Reverse the postfix expression to get the prefix expression*
- Therefore, the prefix expression is *** - A / B C - / A K L**

Evaluation of a Prefix Expression (Algorithm)

```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
        (a) Scan the prefix expression
            from right, one character at a
            time.
        (b) If the scanned character is an
            operand, push it on the
            operand stack.
        (c) If the scanned character is an
            operator, then
              (i) Pop two values from the
                  operand stack
             (ii) Apply the operator on
                  the popped operands
            (iii) Push the result on the
                  operand stack
Step 3: END
```

Evaluation of a Prefix Expression (Sample)

- Consider the prefix expression
  + – 2 7 * 8 / 4 12. Evaluate this expression

- Applying the algorithm from the previous slide to evaluate this expression

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29

Application of Queues

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU
- Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for e.g. pipes, file IO, sockets
- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc
- Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list
- Queues are used in operating systems for handling interrupts

Any Questions?