Unit 3: Stacks and Queues

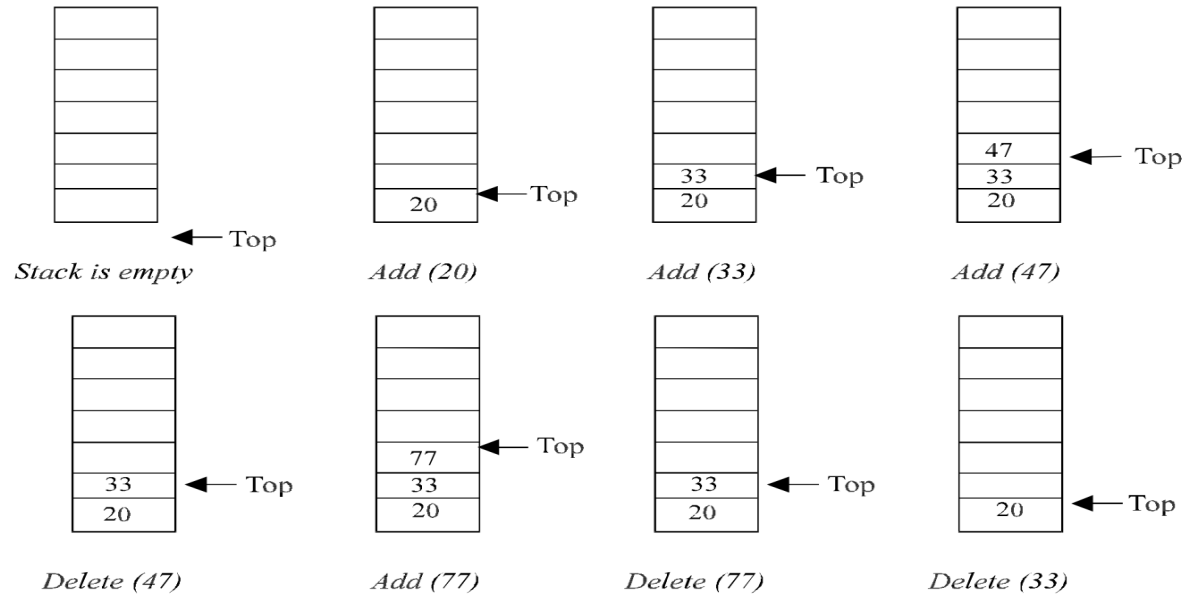
Stacks

- A stack is a linear data structure that follows LIFO (Last in First Out) or FILO (First in Last Out) order to perform operations.
- Storing and retrieving operations can be performed only at one of its ends called top or tos (top of stack).
- Such a stack resembles piles of trays in a cafeteria. New trays are put on the top of the pile and top tray is the first tray removed from the stack. For this reason, a stack is called a LIFO structure: last in /first out.

The Stack ADT

A stack of element of type T is a finite sequence of elements of T together with the operation:

1. **clear ()** : Clear the stack.
2. **isEmpty()** : Check to see if the stack is empty.
3. **Push (el)** : Put the element **el** on the top of stack.
4. **Pop()** : Take the topmost element from the stack.
5. **topEl()** : Return the topmost stack without removing it.



Stack can be implemented in two ways:

1. **Array Implementation of stack**
2. **Linked List Implementation of stack**

1. Array implementation of stack

A. Push Operation:

- For performing push operation, check to see whether the stack is full or not.
- If we try to push an element onto a full stack, the condition is called stack overflow.
- The stack is full when **top = size-1**

Algorithm that pushes an ITEM into a stack:

Step 1: Is Stack full? If TOP = Size-1, then PRINT "Stack Overflow" and return.
Step 2: Set TOP = TOP+1
Step 3: Set STACK [TOP] = ITEM.
Step 4: Return

Method to push an item onto a stack:

```
void push (int stack [], int top, int size, int item)
{
        if (top == size - 1)
                System.out.println ("Stack Overflow");
        else
        {
                top = top + 1;
                stack[top] = item;
        }
}
```

B. Pop Operation:

- For performing pop operation, check to see whether the stack is empty or not.
- If we try to pop an element from an empty stack, the condition is called stack underflow.
- The stack is empty when **top < 0** or **top = -1**

Algorithm to pop an ITEM from a stack:

Step 1: If TOP < 0, then PRINT "Stack Underflow" and return.
Step 2: Set ITEM = STACK[TOP].
Step 3: Set TOP = TOP-1.
Step 4: Return

Method to pop an item from a stack:

```
void pop (int stack [], int top, int item)
{
        if (top == - 1)        // if (top < 0)
                System.out.println("Stack Underflow");
        else
        {
                item = stack[top];
                top = top - 1;
        }
}
```

3. Linked List Implementation of Stack

A linked list of nodes could be used to implement a stack. Linked list representation of stack is nothing but a liked list of nodes where each node is element of the stack. To point to the top most node we have a pointer **top** that points to node recently added. Every new node is added to the beginning of the liked list and top point to it. The main advantage of this representation is that size of stack is not fixed so there is least chance of stack overflow. The node class is:

```
class Node
{
        int data; Node next;
        public Node ()
        {
                next = null;
        }
        public Node (int d)
        {
                data = d;
                next = null;
        }
}
```

Push operation:

Step 1: Create a newNode with given value

Step 2: Check to see if stack is empty

a) If it is empty, then set newNode.next = NULL

b) else set newNode.next = top

Step 3: Set top = newNode

Function:

```
void push (int el)
{
        Node newNode = new Node(el);
        if (top == null)
        {
                newNode.next = null;
                top = newNode;
        }
        else
        {
                newNode.next = top;
                top = newNode;
        }
}
```

POP Operation:

Step 1: Check to see if stack is empty (i.e., top == NULL)

a) If it is empty

print "Stack is empty"

return

b) else

define a node temp

set temp = top

Step 2: Set top = top.next

Step 3: Set temp.next = null

Function:

```
void pop()
{
        Node temp;
        if (top == NULL)
        {
                System.out.println("Stack is empty");
                return;
        }
        else
        {
                temp = top;
                top = top.next;
                temp.next = null;
        }
}
```

- Generally, the stack is very useful in situations when data have to be stored and then retrieved in reverse order.

Examples of Stack Application:

1. Matching delimiters in a program
2. Adding very large numbers

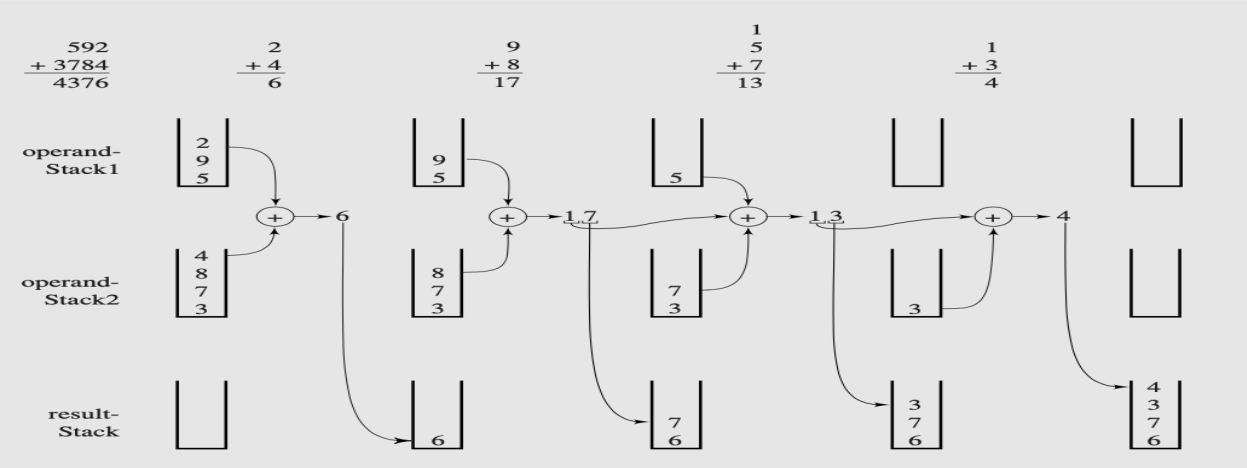
Algorithm for Matching delimiters in a program:

```
delimiterMaching(file)
    read character ch from file;
    while not end of file
        if ch is '(' ,'[', or '{'
            push(ch);
        else if ch is '/'
            read the next character;
             if this character is '*'
                  skip all characters until "*/" is found and report an error if the
                   end of file is reached before "*/" is encountered;
            else ch = the character read in;
                   continue; // go to the beginning of the loop;
        else if ch is ')', ']', or '}'
              if ch and popped off delimiter do not match
                  failure;
        // else ignore other characters;
        read next character ch from file;
    if stack is empty
        success;
    else
        failure;
```

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
(	(	w + y));
(	w	+y));
(	+	y));
(	y	));
(	)	);
empty	)	;
empty	;	

Algorithm for Adding very large numbers:

```
addingLargeNumbers()
   read the numerals of the first number and store the numbers corresponding to
   them on one stack;
   read the numerals of the second number and store the numbers corresponding
   to them on another stack;
result = 0;
while  at least one stack is not empty
      pop a number from each nonempty stack and add them to  result;
      push the unit part on the result stack;
      store carry in  result;
   push carry on the result stack if it is not zero;
   pop numbers from the result stack and display them;
```



Infix, Prefix and Postfix Expressions:

Infix expression:

- Infix expression is a mathematical notation in which operator is written in between the operands.
- Example: A+B, where A and B are operands and + is an operator.

Prefix expression:

- Prefix expression is a mathematical notation in which the operator preceds the two operands.
- Example: +AB, where A and B are operands and + is an operator.

Postfix expression:

- Infix expression is a mathematical notation in which the operator is written after the operands.
- Example: AB+, where A and B are operands and + is an operator.

Note: Prefix and Postfix are parenthesis free expressions.

- Postfix notation is most suitable for a computer to calculate any expression.
- Postfix notation is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore, it is necessary to study the postfix notation.
- Any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

Advantages of using Postfix Notation:

- Using infix notation, one cannot tell the order in which operators should be applied.
- Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first.
- But in postfix expression operands appear before the operator, so there is no need for operator precedence and other rules.

Notation Conversions

- Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result.
- For example: A + B * C = 4 + 3 * 7 = 7 * 7 = 49
- The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like.
- The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus, expression A + B * C can be interpreted as A + (B * C).
- Using this alternative method, we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

CONVERTING INFIX TO POSTFIX EXPRESSION:

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Converting infix expression, A + B * C to postfix form is:

A+B*C	Infix Form
= A + (B * C)	Parenthesized expression
= A + (B C *)	Convert the multiplication
= A (B C *) +	Convert the addition
= ABC*+	Postfix Form

Algorithm to convert Infix to Postfix Notation:

1. Start
2. Scan one character at a time of an infix expression from left to right.
3. opstack = the empty stack
4. Repeat till there is data in infix expression
4.1 if scanned character is '('
push it to opstack
4.2 if scanned character is operand
push it to postfix string
4.3 if scanned character is operator
while (opstack != -1 && precedence(opstack[otop]) > precedence (scan character))
pop and push it into postfix string
otherwise
push scanned character into opstack
4.4 if scanned character is ')'
pop and push into postfix string until '(' is not found and ignore both symbols
5. pop and push it into postfix string until opstack is not empty
6. Stop

Example 1: A+B*C

Scan character	Postfix String	opstack
A	**A**	
+	**A**	**+**
B	**AB**	**+**
*****	**AB**	**+ ***
C	**ABC**	**+ ***
	ABC*	**+**
	ABC*+	

Example 2: (A+B)*C

Scan character	Postfix String	opstack
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB+	
*	AB+	*
C	AB+C	*
	AB+C*	

Example 3: ((A-(B+C)) *D) $(E+F)

Scan character	Postfix String	opstack
(		(
(		((
A	A	((
-	A	((-
(	A	((- (
B	AB	((- (
+	AB	((- (+
C	ABC	((- (+
)	ABC +	((-
)	ABC + -	(
*	ABC + -	(*
D	ABC + - D	(*
)	ABC + - D *	
$	ABC + - D *	$
(	ABC + - D *	$ (
E	ABC + - D * E	$ (
+	ABC + - D * E	$ (+
F	ABC + - D * EF	$ (+
)	ABC + - D * EF +	$
	ABC + - D * EF + $	

Convert following Infix expressions into Postfix expressions

1. **P+Q–(R*S/T+U)-V*W** = **PQ+RS*T/U+V-W***
2. **A+(B/C-(D*E^F) +G) *H**
3. **A+ [(B + C) +(D+E) *F]/G**
4. **(A+B) *C/D+E^A/B**
5. **((A+B)-C*D/E) *(H-I) *F+G**
6. **((H*((((A+((B+C) *D)) *F) *G) *E)) +J)**

Converting Infix expressions to Prefix expressions:

- The precedence rule for converting an infix expression into prefix expression is identical to that of infix to postfix.
- Only change is that the operator is placed before the operands.

Convert the given infix expressions into prefix expressions:

(i) A+B-C
= +AB-C
= -+ABC

(ii) A$B*C-D+E/F/(G+H)
= A$B*C-D+E/F/(+GH)
= $AB*C-D+E/F/(+GH)
= *$ABC-D+E/F/(+GH)
= *$ABC-D+ (/EF)/(+GH)
= *$ABC-D+ (//EF+GH)
= -*$ABCD+ (//EF+GH)
= +-*$ABCD//EF+GH

Convert following Infix expressions into Prefix expressions

1. P+Q–(R*S/T+U)-V*W
2. A+(B/C-(D*E^F) +G) *H
3. A+ [(B + C) +(D+E) *F]/G
4. (A+B) *C/D+E^A/B
5. ((A+B)-C*D/E) *(H-I) *F+G

Evaluating the Postfix Expression

Procedures to evaluate postfix expression:

- Each time we read an operand we push it onto a stack.
- When we reach an operator, its operands will be the top two elements on the stack.
- We then pop these two elements from stack and perform the indicated operation on them.
- Then push the result on the stack so that it will be available for use an operand of the next operator.

Algorithm to evaluate postfix expression:

1. Scan one character at a time from left to right of the given postfix expression

1.1 if scanned character is an operand

read its corresponding value and push it into vstack;

1.2 if scanned character is an operator

pop and assigned to op2;

pop and assigned to op1;

compute the result according to given operator and push result into vstack;

2. pop and display which is required value of the given postfix expression;

3. Stop

Examples:

(i) 345*+

= 3 20+ [multiply 4 and 5]

= 23 [add 3 and 20]

Tracing:

character	op2	op1	result	vstack
3				3
4				3 4
5				3 4 5
*	5	4	20	3 20
+	20	3		23

(ii) 623+-382/+*2^3+

= 6 5 -382/+*2^3+

= 1 382/+*2^3+

= 1 3 4+*2^3+

= 1 7 *2^3+

= 7 2^3+

= 49 3+

= 52

623+-382/+*2^3+

character	op2	op1	result	vstack
6				6
2				2
3				3
+	3	2	5	6 5
-	5	6	1	1
3				1 3
8				1 3 8
2				1 3 8 2
/	2	8	4	1 3 4
+	4	3	7	1 7
*	7	1	7	7
2				7 2
^	2	7	49	49
3				49 3
+	3	49	52	52

(iii) 123+*321-+*

(iv) 48-2569/+1-*-3-

Evaluating the Prefix Expression

(i) +5*32

= +5 6 // *32 = 3*2

= 11 // +56 = 11

(ii) /+53-42

= / 8 -42

= /8 2

= 4

Transform each of the following prefix expressions to infix:

(i) +-**AB**C
= + A-**BC**
= + **AB**-C
= A+B-C

(ii) ++A-*^**BC**D/+EF*GHI
= ++A-*B^**CD**/+EF*GHI
= ++A-***BC**^D/+EF*GHI
= ++A-B*C^D/+**EF***GHI
= ++A-B*C^D/E+F***GH**I
= ++A-B*C^D/E+**FG***HI
= ++A-B*C^D/**EF**+G*HI
= ++A-B*C^**DE**/F+ G*HI
= ++A-B***CD**^E/F+ G*HI
= ++A-**BC***D^E/F+ G*HI
= ++**AB**-C*D^E/F+ G*HI
= +A+B-C*D^E/F+ G***HI**
= +A+B-C*D^E/F+ **GH***I
= +A+B-C*D^E/**FG**+H*I
= +A+B-C*D^**EF**/G+H*I
= +A+B-C***DE**^F/G+H*I
= +A+B-**CD***E^F/G+H*I
= +A+BC-D*E^F/G+H*I
= +**AB**+C-D*E^F/G+H*I
= A+B+C-D*E^F/G+H*I

(iv) +-^ABC*D**EFG

Transform each of the following postfix expressions to infix:

(i) **AB+**C-
= A+**BC-**
= A+B-C
(ii) A**BC+**-
= AB**+**C-
= A+**BC-**
= A+B-C

(iii) AB-C+DEF-+^

(iv) ABCDE-+^*EF*-