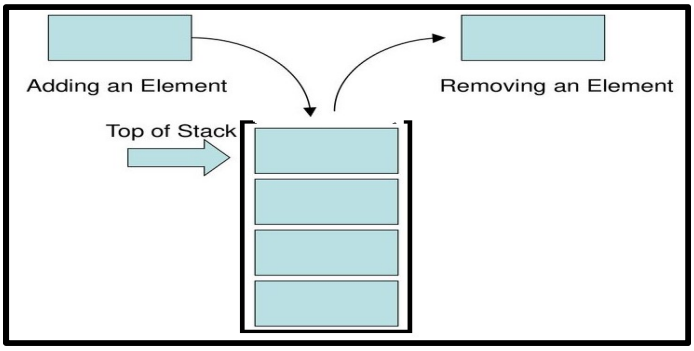
Week 8	**Stack (Last In First Out) Data Structures – Example: Reversing a word, evaluating an expression, message box etc.** **The Stack Implementation – Push, Pop, display.** **Stack Applications – Balanced Delimiters, Evaluating Postfix Expressions.**

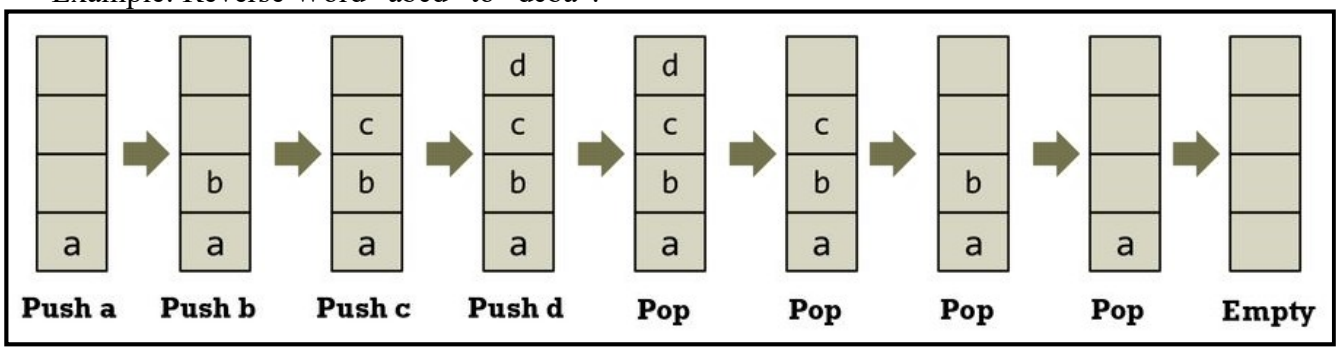
Stack

- **A Stack is a linear data structure, used to store group of data items such that the last item inserted is the first item removed.**
- **Stack data structure follows LIFO (Last In First Out)** principle for insertion-deletion operation.
- In Stack, new data items are added, or existing data items are deleted from the same end, known as **top** of the Stack.
- **Abstract View of Stack:**



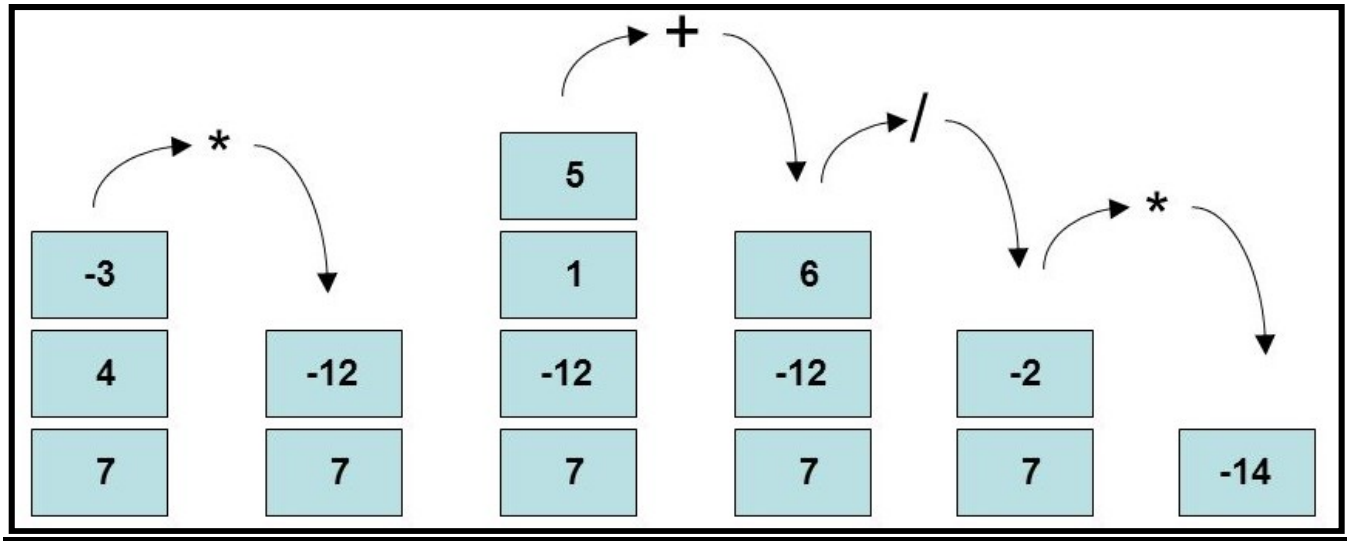
Example #1: Reversing a Word

- To reverse a given word or string, the characters of the word or string are inserted/pushed onto the stack, one by one from left to right.
- Once all characters are inserted / pushed onto the stack, they are removed/popped back, one by one.
- Since the character last inserted/pushed in comes out first, subsequent remove/pop operations produces the reverse of the given word or string.
- Example: Reverse Word “abcd” to “dcba”.



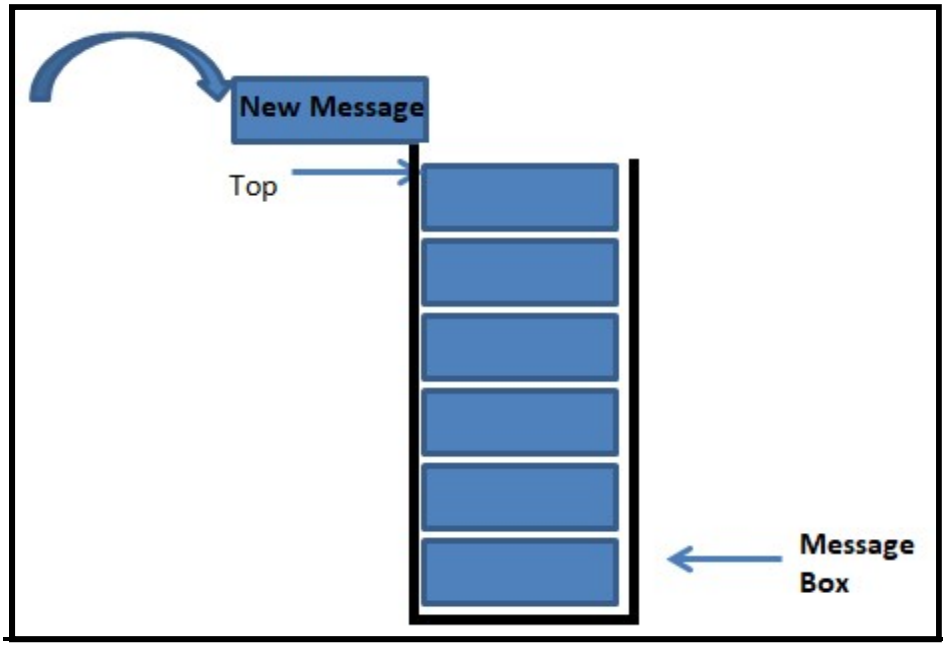
Example #2: Evaluating an Expression

- Mathematical expressions are rather easy for humans to evaluate. But the task is more difficult in a computer program.
- To simplify the evaluation of a mathematical expression, we need to convert it to another form i.e postfix form.
- Postfix Expressions can be evaluated using stack.
- **Example: Postfix Expression, 7 4 -3 * 1 5 + / ***



Example #3: Message Box

- Message box in messenger applications work like Stack data structure.
- New Message are always added or inserted at the top of the message box.



Stack ADT

A stack is a data structure that stores a linear collection of data items. Adding and removing items is done from one end known as the top of the stack. An empty stack is a stack containing no items.

- **Stack( )**: Creates a new empty stack.
- **isEmpty( )**: Returns a boolean value indicating whether the stack is empty or not.
- **length( )**: Returns the number of items in the stack.
- **push(item)**: Adds the given item to the top of the stack.
- **pop()**: Removes and returns the top item of the stack, if the stack is not empty. Items cannot be popped from an empty stack. The next item on the stack becomes the new top item.
- **peek()**: Returns data item from top of a non-empty stack without removing it.

Stack Implementation

```
class Stack:
    def __init__(self):
        self.items = list()

    def push(self,value):
        self.items.append(value)

    def pop(self):
        if self.isEmpty() == True:
            print("Stack is empty, cannot remove")
        else:
            return self.items.pop()

    def peek(self):
        if self.isEmpty() == True:
            print("Stack is empty")
        else:
            return self.items[-1]

    def  isEmpty(self):
        if len(self.items) == 0:
            return True
        else:
            return False

    def  length(self):
        return len(self.items)
```

Stack Applications

Balanced Delimiters

- A number of applications use delimiters.Some common examples include mathematical expressions, programming languages, and the HTMLmarkup language used by web browsers.
- There is a strict rule that delimiters must be paired and balanced.
- The delimiters must be used in pairs of corresponding types: {}, [], and ().

Balanced Delimiters can be checked using Stack as follows:

1. As the input is scanned, we can push each opening delimiter onto the stack.
2. When a closing delimiter is found, we pop the opening delimiter from the stack and compare it to the closing delimiter.
3. For properly paired delimiters, the two should match. Thus, if the top of the stack contains a left bracket [, then the next closing delimiter should be a right bracket].
4. If the two delimiters match, it indicates that current delimiter is properly paired and can continue processing the remaining input.
5. But if they do not match, it means that delimiters are not balanced / matched and we can stop processing the input.

Example #1: input : ({ [ ] })

Input	Current Character	Stack Operation	Stack
({ [ ] })	(	Push (to the Stack	(
({ [ ] })	{	Push { to the Stack	({
({ [ ] })	[	Push [to the Stack	({ [
({ [ ] })	]	Pop i.e [Compare it with current character i.e] Yes, it matches	({
({ [ ] })	}	Pop i.e { Compare it with current character i.e } Yes, it matches	(
({ [ ] })	)	Pop i.e (Compare it with current character i.e) Yes, it matches	---------
({ [ ] })	--------	Since the stack is empty now, Brackets in given input is matched.	

Example #2: input: a=int(input( ))

Input	Current Character	Stack Operation	Stack
a=int(input())	a	No Operation. Just ignore.	Empty
a=int(input())	=	No Operation. Just ignore.	Empty
a=int(input())	i	No Operation. Just ignore.	Empty
a=int(input())	n	No Operation. Just ignore.	Empty
a=int(input())	t	No Operation. Just ignore.	Empty
a=int(input())	(	Push (to the Stack	(
a=int(input())	i	No Operation. Just ignore.	(
a=int(input())	n	No Operation. Just ignore.	(
a=int(input())	p	No Operation. Just ignore.	(
a=int(input())	u	No Operation. Just ignore.	(
a=int(input())	t	No Operation. Just ignore.	(
a=int(input())	(	Push (to the Stack	((
a=int(input())	)	Pop i.e (Compare it with current character i.e (Yes, it matches	(
a=int(input())	)	Pop i.e (Compare it with current character i.e (Yes, it matches	Empty
a=int(input())	--------	Since the stack is empty now, Brackets in given input is matched.	

Example #3: input : ({)

Input	Current Character	Stack Operation	Stack
({ [ )	(	Push (to the Stack	(
({)	{	Push { to the Stack	({
({)	)	Pop i.e { Compare it with current character i.e) No, it does not match. Stop processing the input.	(
Brackets in given input is not matched.			

Evaluating Postfix Expressions

- We work with mathematical expressions on a regular basis and they are rathereasy for humans to evaluate. But the task is more difficult in a computer program.
- To simplify the evaluation of a mathematical expression, we need an alternative representation for the expression.
- Three different notations can be used to represent a mathematical expression.
  - **Infix notation - the operator is specified between the operands A+B.**
  - **Prefix notation - the operator immediately precedes the two operands +AB,**
  - **Postfix notation - theoperator follows the two operands AB+.**

Converting from Infix to Postfix

1. Place parentheses around every group of operators in the correct order of evaluation. There should be one set of parentheses for every operator in the infix expression.

   ((A * B) + (C / D))
2. For each set of parentheses, move the operator from the middle to the end preceding the corresponding closing parenthesis.

   ((A B *) (C D /) +)
3. Remove all of the parentheses, resulting in the equivalent postfix expression.

   A B * C D / +

Evaluation of Postfix Expressions

Evaluating a postfix expression requires the use of a stack to store the operands or variables at the beginning of the expression until they are needed. Assume we are given a valid postfix expression stored in a string consisting of operators and single-letter variables. We can evaluate the expression by scanning the string, one character at a time. For each character or item, we perform the following steps:

1. If the current item is an operand, push its value onto the stack.
2. If the current item is an operator:
   a. Pop the top two operands off the stack.
   b. Perform the operation. (Note the top value is the right operand while the next to the top value is the left operand.)
   c. Push the result of this operation back onto the stack.

3. At the end of the expression evaluation, If the stack contains only one value i.e result, then it indicates that the given infix/postfix expression is valid.
4. At the end of the expression evaluation, If the stack contains too many values, it indicates that the given infix/postfix expression is invalid (we need to display appropriate error message in applications to indicate the same).

Example #1: Evaluate the postfix expression: A B C + * D/.

Given, A = 8 B = 2 C = 3 D = 4

Expression	Current Character	Stack Operation	Stack
A B C + * D /	A	Push the value of A to the Stack Push (8)	**8**
A B C + * D /	B	Push the value of B to the Stack Push (2)	**8 2**
A B C + * D /	C	Push the value of C to the Stack Push (3)	**8 2 3**
A B C + * D /	+	Pop two items, right = 3 left = 2 perform 2 + 3 = 5 Push 5 to the stack , Push(5)	**8 5**
A B C + * D /	*	Pop two items, right = 5 left = 8 perform 8 * 5 = 40 Push 40 to the stack , Push(40)	**40**
A B C + * D /	D	Push the value of D to the Stack Push (4)	**40 4**
A B C + * D /	/	Pop two items, right = 4 left = 40 perform 40 / 4 = 10 Push 10 to the stack , Push(10)	**10**

At the end, Here Only one value (result) left on stack, it indicates that given expression is valid and 10 is the result of the expression.

Example #2: Evaluate the expression: A * B / C.

Given, A = 5 B =4 C = 3

1. **Convert given expression to its postfix form.**
   a. **((A*B)/C)**
   b. **((AB*)C/)**
   c. **AB*C/**

Equivalent Postfix Expression: AB*C/

2. **Evaluate Postfix Expression.**

Postfix Expression: AB*C/

Given, A = 5 B =4 C = 10

Expression	Current Character	Stack Operation	Stack
A B * C /	A	Push the value of A to the Stack Push (5)	5
A B * C /	B	Push the value of B to the Stack Push (4)	5 4
A B * C /	*	Pop two items, right = 4 left = 5 perform 5 * 4 = 20 Push 20 to the stack , Push(20)	20
A B * C /	C	Push the value of C to the Stack Push (10)	20 10
A B * C /	/	Pop two items, right = 10 left = 20 perform 20 / 10 = 2 Push 2 to the stack , Push(2)	2

At the end, Here Only one value (result) left on stack, it indicates that given expression is valid and 2 is the result of the expression.

Example #3: Evaluate the postfix expression: A B * C D +.

Given, A = 8 B = 2 C = 3 D = 4

Expression	Current Character	Stack Operation	Stack
A B * C D +	A	Push the value of A to the Stack Push (8)	8
A B * C D +	B	Push the value of B to the Stack Push (2)	8 2
A B * C D +	*	Pop two items, right = 2 left = 8 perform 8 * 2 = 16 Push 16 to the stack , Push(16)	16
A B * C D +	C	Push the value of C to the Stack Push (3)	16 3
A B * C D +	D	Push the value of C to the Stack Push (4)	16 3 4
A B * C D +	+	Pop two items, right = 4 left = 3 perform 3 + 4 = 7 Push 7 to the stack , Push(7)	16 7

At the end, Here Too many values left on stack, it indicates that given expression is invalid.

Program #1: Python Program to implement Stack.

```
class Stack:
def __init__(self):
self.items = list()

def push(self,value):
self.items.append(value)

def pop(self):
      if self.isEmpty() == True:
print("Stack is empty, cannot remove")
      else:
         return self.items.pop()

def peek(self):
      if self.isEmpty() == True:
print("Stack is empty")
      else:
         return self.items[-1]

def display(self):
print("Stack items are")
      for i in self.items:
         print(i)

defisEmpty(self):
      if len(self.items) == 0:
         return True
      else:
         return False

def length(self):
      return len(self.items)

S = Stack()

S.push(10)
S.push(20)
```

```
S.push(30)
S.display()
print("Top of the Stack:",S.peek())
print("Length: ", S.length())
S.pop()
S.display()
print("Top of the Stack:",S.peek())
S.pop()
S.display()
print("Top of the Stack:",S.peek())
print("Empty?: ",S.isEmpty())
S.pop()
print("Empty?: ",S.isEmpty())
print("Length: ", S.length())
```

Output #1:

Stack items are

10

20

30

Top of the Stack: 30

Length: 3

Stack items are

10

20

Top of the Stack: 20

Stack items are

10

Top of the Stack: 10

Empty?: False

Empty?: True

Length: 0

Program #2: Python Program to implement Bracket Matching using Stack

```
class Stack:
def __init__(self):
self.items = list()

def push(self,value):
self.items.append(value)

def pop(self):
      if self.isEmpty() == True:
print("Stack is empty, cannot remove")
      else:
         return self.items.pop()


defisEmpty(self):
      if len(self.items) == 0:
         return True
      else:
         return False

defisBalanced(S,text):
   for char in text:
     if char in ["[","{","("]:
S.push(char)
elif char in ["]","}",")"]:
        temp = S.pop()
        if temp == "(":
           if char != ")":
              return False
        if temp == "{":
           if char != "}":
              return False
        if temp == "[":
           if char != "]":
              return False
     else:
        continue
   if S.isEmpty() == True:
      return True
```

```
S = Stack()
text = input("Enter Text\n")

res=isBalanced(S,text)
if res==True:
print("Valid and Balnaced Expression")
else:
   print("Invalid expression")
```

Output #1:

Enter Text

a=int(input("Enter number"))

Valid and Balnaced Expression

Output #2:

Enter Text

a=int(input("Enter number")

Invalid Expression

Activity #8

1. **Develop Python Program to reverse a given word using Stack.**

2. **A letter means push and an asterisk means pop in the following sequence. Give the sequence of values returned by the pop operations when this sequence of operations is performed on an initially empty Stack.**

   E A S * Y * Q U E * * * S T * * * I O * N * * *

3. **Hand execute the following code segment and show the contents of the resulting stack.**

```
values = Stack()
for iin range( 16 ) :
    if i % 3 == 0 :
        values.push( i )
```

4. **Hand execute the following code segment and show the contents of the resulting stack.**

```
values = Stack()
for iin range( 16 ) :
    if i % 3 == 0 :
        values.push( i )
    elifi % 4 == 0 :
        values.pop()
```

5. **Translate each of the following infix expressions into postfix notation and evaluate it using Stack. Given, A= 5, B=4, C=1 D=5 E=3.**

   (a) (A * B) / C
   (b) A - (B * C) + D / E
   (c) (X - Y) + (W * Z) / V
   (d) V * W * X + Y - Z
   (e) A / B * C - D + E

6. **Translate each of the following infix expressions to prefix notation.**

   (a) (A * B) / C
   (b) A - (B * C) + D / E
   (c) (X - Y) + (W * Z) / V
   (d) V * W * X + Y - Z
   (e) A / B * C - D + E

7. **Evaluate the following postfix expressions using stack.**
   Given, A= 5, B=4, C=1 D=5 E=3.

   (a) A B C - D * +
   (b) A B + C D - / E +
   (c) A B C D E * + / +
   (d) X Y Z + A B - * -
   (e) A B + C - D E * +

8. **For the following statements / Expressions, Verify whether delimiters are paired and balanced or not.**

   (a) print(sum(10,20)
   (b) [ ] ({ })
   (c) (A - C) + (C * D) / E
   (d) [{ } ( ) { } [ ]
   (e) ([{ } ( )])
   (f) <a>Good<a<