Week 8

LIFO Data Structure

- Last In First Out (Stack) Data structures - Example: Reversing a word, evaluating an expression, message box etc. (to be used to explain concept of LIFO).
- The Stack implementation - push, pop, display.
- Stack Applications- Balanced Delimiters, Evaluating Postfix Expressions.

A stack is used to store data such that the last item inserted is the first item removed. It is used to implement a **last-in first-out (LIFO)** type protocol.

Definition of Stack:The stack is a linear data structure in which new items are added, or existing items are removed from the same end, commonly referred to as the **top of the stack**. The opposite end is known as the **base.**

Consider the example in Figure, which illustrates new values being added to the top of the stack and one value being removed from the top.

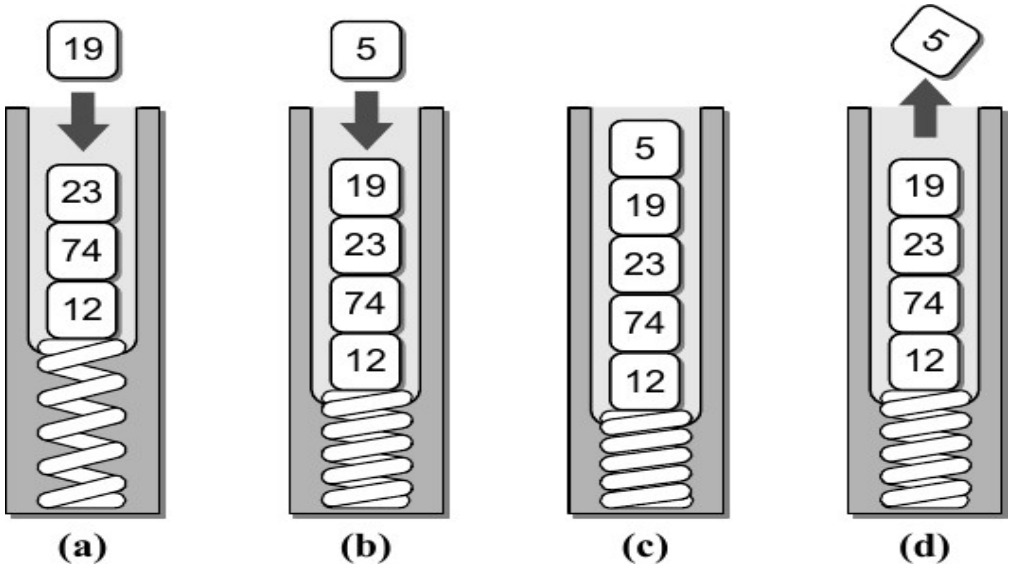

Figure : Abstract view of a stack: (a) pushing value 19; (b) pushing value 5; (c) resulting stackafter 19 and 5 are added; and (d) popping top value.

Applications of Stack:

a. Reverse a String(A stack can be used to check whether the given string is palindrome):

- 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 (LIFO)** property of the Stack, the first character of the String 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.

b. Evaluation of Arithmetic Expressions: A stack is a very effective data structure for evaluating arithmetic expressions

c. Recursion: A stack is used for implementing recursion

Define Stack ADT

A stack is a data structure that stores a linear collection of items with access limited to a last-in first-out order. Adding and removing items is restricted to one end known as the top of the stack. An empty stack is a stack that doesn't contain any items. The following are the operations which can be defined and implemented for stack ADT

- Stack(): Creates a new empty stack.
- isEmpty(): Returns True if the stack is empty.
- length (): Returns the number of items in the stack.
- pop(): Removes and returns the top item of the stack, if the stack is not empty. Itemscannot be popped from an empty stack. The next item on the stack becomes the new top item.
- peek(): Returns a reference to the item on top of a non-empty stack without removing it.Peeking, which cannot be done on an empty stack, does not modify the stack contents.

- push(item): Adds the given item to the top of the stack.

Stack implementation

Let us consider implementation of a stack in Python using List Data structure .Let us start by creating a Stack class.

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

Push operation

The push operation is used to add an element to the top of the stack. Here is an implementation:

```
def push(self, data):
        self.items.append(data)
```

Pop operation

Pop method will remove the top element from the stack. We will make the stack return None ifthere are no more elements:

```
def pop(self):
        return self.items.pop()
```

Peek

Peek method will return the top of the stack without removing it from the stack. If there is a topelement, return its data, otherwise return None.

```
def peek(self):
        return self.items[len(self.items) – 1]
```

Display

Display method will display the all the elements of stack.

```
def display(self):
        for self.i in self.items:
                print(self.i)
```

Stack Empty condition:

IsEmpty() method will return True if the Stack is empty

```
def IsEmpty(self):
    return self.items==[ ]
```

Balanced Delimiter

- A number of applications use delimiters to group strings of text or simple data into sub-parts by marking the beginning and end of the group.
- Some common examples include mathematical expressions, programming languages, and the HTML markup language used by web browsers.
- Parentheses can be used in mathematical expressions to group or override the order of precedence for various operations.
- The delimiters must be used in pairs of corresponding types: {}, [], and (). They must also be positioned such that an opening delimiter within an outer pair must be closed within the same outer pair. For example, the following expression would be valid.

{A + (B * C) - (D / [E + F])}

- For example, the following expression would be invalid since the pair of braces [] begin inside the pair of parentheses () but end outside.

(A + [B * C)] - {D / E}

- The Stack ADT is a perfect structure for implementing an algorithm which will be used to check balanced parantheses.
- We can push each opening parantheses onto the stack. When a closing parantheses is encountered, we pop the opening parantheses from the stack and compare it to the closing one.
- For properly paired delimiters, the two should match. Thus, if the top ofthe stack contains a left bracket [, then the next closing delimiter should be a right bracket]. If the two delimiters match, we know they are properly paired and can continue processing the source code. But if they do not match, then we know the delimiters are not correct and we can stop processing .

Tables shows the steps performed by an algorithm and the contents of the stack after eachdelimiter is encountered in our samples.

Sample : {A + (B * C) - (D / [E + F])}

Operation	Stack	Current Scan Line
Push {	{	{
Push (	{(	{A+(
Pop (& Match)	{	{A+(B*C)-
Push (	{(	{A+(B*C)-(
Push[	{([	{A+(B*C)-(D/[

Pop[& Match]	{(	{A+(B*C)-D/[E+F]
Pop (& Match)	{	{A+(B*C)-D/[E+F])
Pop { & Match	Empty	

Sample : (A + [B * C)] - {D / E}

Operation	Stack	Current Scan Line
Push (	(	(
Push [	([	{A+[
Pop (& Match)	([- Error	{A+[B*C)

Evaluating Postfix Expressions.

Three different notations can be used to represent a mathematical expression.

1) **Infix notation**: This is the most common traditional algebraic notation where the operator is specified between the operands Ex: A+B.
2) **Prefix notation:** In this, an operator is placed before the operands EX: +AB (An operator precedes the operands)
3) **Postfix notation**: In this, an operator is placed after the operands EX: AB+ (An operator follows the operands)

Infix expressions can be easily converted by hand to postfix notation. The expression A + B - C would be written as AB+C- in postfix form.

To help in this conversion we can use a simple algorithm:

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 / +

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 or token at a time. For each token, we perform the following steps:

1. If the current character/token is an operand, push its value onto the stack.
2. If the current token is an operator:
    (a) Pop the top two operands off the stack.
    (b) Perform the operation.

(c) Push the result of this operation back onto the stack.

The final result of the expression will be the last value on the stack.

To illustrate the use of this algorithm, let's evaluate the postfix expression A B C + * D / from our earlier example. Assume the existence of an empty stack and the following variable assignments have been made:

A = 8 **C = 3**
B = 2 **D = 4**

The complete sequence of algorithm steps and the contents of the stack after each operation are illustrated in Table.

Token	Alg. Step	Stack	Description
ABC+*D/	1	8	push value of A
A**B**C+*D/	1	8 2	push value of B
AB**C**+*D/	1	8 2 3	push value of C
ABC**+***D/	2(a)	8	pop top two values: y = 3, x = 2
	2(b)	8	compute z = x + y or z = 2 + 3
	2(c)	8 5	push result (5) of the addition
ABC+*****D/	2(a)		pop top two values: y = 5, x = 8
	2(b)		compute z = x * y or z = 8 * 5
	2(c)		push result (40) of the multiplication
ABC+*D/	1	40 4	push value of D
ABC+*D**/**	2(a)		pop top two values: y = 4, x = 40
	2(b)		compute z = x / y or z = 40 / 4
	2(c)	10	push result (10) of division

*Table : The stack contents and sequence of algorithm steps required to evaluate the valid postfix expression A B C + * D.*

Consider the following invalid expression in which there are more operands than available operators: A B * C D +. In this case, there are too few operands on the stack when we encounter the addition operator, as illustrated in Table . If we attempt to perform two pops from the stack, an assertion error will be thrown since the stack will be empty on the second pop.

Token	Alg. Step	Stack	Description
AB*CD+	1	8	push value of A
A**B**CD+	1	8 2	push value of B
AB*****CD+	2(a)		pop top two values: y = 2, x = 8
	2(b)		compute z = x * y or z = 8 * 2
	2(c)	16	push result (16) of the multiplication
AB*CD+	1	16 3	push value of C
AB*C**D**+	1	16 3 4	push value of D
AB*CD**+**	2(a)	16	pop top two values: y = 4, x = 3
	2(b)	16	compute z = x + y or z = 3 + 4
	2(c)	16 7	push result (7) of the addition
Error	***xxxxx***	***xxxxx***	***Too many values left on stack.***

*Table : The sequence of algorithm steps when evaluating the invalid postfix expression A B * C D +.*

PROGRAMS

Implement Stack Data Structure. (stack.py)

stack.py

```
class stack:
        def __init__(self):
                self.items = []
        def isEmpty(self):
                return self.items == []
        def push(self, item):
                self.items.append(item)
        def pop(self):
                return self.items.pop()
        def peek(self):
                return self.items[len(self.items) - 1]
        def size(self):
                return len(self.items)
        def display(self):
                return  (self.items)
```

stack_impl.py

```
import stack
s=stack.Stack()
print(s.isEmpty())
print("push operations")
s.push(11)
s.push(12)
s.push(13)
print("size:",s.size())
print(s.display())
print("peek",s.peek())
print("pop  operations")
print(s.pop())
print(s.pop())
print(s.display())
print("size:",s.size())
```

Implement bracket matching using stack.

Stack_Bracket.py

```
import stack
def check_brackets(statement):
    s = stack.stack()
    for token in statement:
        if token in "{[(":
            s.push(token)
        elif token in "}])":
            if s.isEmpty():
                return False
            else:
                left = s.pop()
                if (token == "}" and left != "{") or \
                    (token == "]" and left != "[") or \
                    (token == ")" and left != "("):
                    return False
    return s.isEmpty()
```

```
# Driver code
sl = ( "{(foo)(bar)}[hello](((this)is)a)test", "{(foo)(bar)}[hello]
(((this)is)atest", "{(foo)(bar)}[hello](((this)is)a)test))"  )

for s in sl:
    m = check_brackets(s)
    print("{}: {}".format(s, m))
```

OUTPUT:

```
{(foo)(bar)}[hello](((this)is)a)test: True
{(foo)(bar)}[hello](((this)is)atest: False
{(foo)(bar)}[hello](((this)is)a)test)): False
```