Stack

Stack Implementation using python

```python
class Node:

    def __init__(self, data=None, next=None):

        self.data = data

        self.next = next


class LinkedList:

    def __init__(self):

        self.head = None

    def insert_at_begining(self,data):

        node = Node(data,self.head)

        self.head = node

    def print(self):

        if self.head is None:

            print("Linked list is empty")

            return

        itr = self.head

        llstr = ''

        while itr:

            llstr = llstr+str(itr.data) + '-->'

            itr = itr.next

        print(llstr)

    def insert_at_end(self,data):

        if self.head is None:

            node = Node(data,None)

            self.head = node
            return
        itr = self.head
        while itr.next:
            itr = itr.next
        itr.next = Node(data, None)
    def insert_multiple_val(self,data_list):
        self.head = None
        for data in data_list:
            self.insert_at_end(data)
    def get_length(self):
        count = 0
        itr = self.head
        while itr:
            count+=1;
            itr = itr.next
        return count
    def remove_at(self,index):
        if index < 0 or index >= self.get_length():
            raise Exception("invalid index !")

        if index == 0:
            self.head = self.head.next
            return

        count = 0
        itr = self.head
        while itr:
            if count == index - 1:
                itr.next = itr.next.next
```

```python
                break
            itr = itr.next
            count+=1
    def insert_at(self, index_at, data):
        if index_at<0 or index_at>self.get_length():
            raise Exception("Invalid index")
        if index_at == 0:
            self.insert_at_begining(data)
            return
        count = 0
        itr = self.head
        while itr:
            if count == index_at - 1:
                node = Node(data,itr.next)
                itr.next = node
            itr = itr.next
            count += 1


if __name__ == '__main__':
    ll = LinkedList()

    ll.insert_at_begining(5)
    ll.insert_at_begining(4)
    ll.insert_at_begining(3)
    ll.insert_at_end(7)
    ll.insert_at_end(8)

    # ll.insert_multiple_val(['sumit','snehasis','dipali'])
    # ll.print()
    # print(ll.get_length())
    # ll.remove_at(2)
    # ll.print()
    # ll.insert_at(0,"shoyeb")
    # ll.insert_at(2,"pritam")
    ll.print()
```

Output:

```
3-->4-->5-->7-->8-->
```

Postfix to infix

```
class stack:
    def __init__(self, Maxsize):
        self.top = -1
        self.MS = Maxsize
        self.arr = [None]*Maxsize

    def isEmpty(self):
        return self.top == -1

    def isFull(self):
        return self.top == self.MS-1

    def push(self, ele):
        if self.isFull():
            print("Full")
```

```python
        else:
            self.top += 1
            self.arr[self.top] = ele

    def pop(self):
        if self.isEmpty():
            print("empty")
        else:
            ele = self.arr[self.top]
            self.arr[self.top] = None
            self.top -= 1
            return ele

    def peek(self):
        return (self.arr[self.top])



s = stack(10)

operators = ['(', ')', '+', '-', '*', '/', '%', '^']
exp = "ABC*+"
for i in exp:
    if i.isalpha():
        s.push(i)
    elif i in operators:
        A = s.pop()
        B = s.pop()
        EXP = "("+B+i+A+")"
        print(EXP)

        s.push(str(EXP))
```

Output:

(B*C)

(A+(B*C))

Postfix to Prefix

```
class stack:

    def __init__(self, Maxsize):

        self.top = -1

        self.MS = Maxsize

        self.arr = [None]*Maxsize


    def isEmpty(self):

        return self.top == -1


    def isFull(self):

        return self.top == self.MS-1


    def push(self, ele):

        if self.isFull():

            print("Full")

        else:

            self.top += 1
```

```python
            self.arr[self.top] = ele

    def pop(self):
        if self.isEmpty():
            print("empty")
        else:
            ele = self.arr[self.top]
            self.arr[self.top] = None
            self.top -= 1
            return ele

    def peek(self):
        return (self.arr[self.top])

    def __str__(self):
        data = []
        for i in range(self.top+1):
            data.append(self.arr[i])
        return str(data)
        '''if self.isEmpty():
                                    print("empty")
                        else:
                                    for i in range(self.top+1):
                                                print(self.arr[i])'''


s = stack(10)


operators = ['(', ')', '+', '-', '*', '/', '%', '^']
```

```python
exp = "-+P*QCD"
exp_r = exp[::-1]
print(exp_r)
for i in exp_r:
    if i.isalpha():
        s.push(i)
    elif i in operators:
        A = s.pop()
        B = s.pop()
        EXP = "("+B+A+i+")"

        print(EXP)
        s.push(str(EXP))
```

Output:

```
DCQ*P+-
(CQ*)
((CQ*)P+)
(D((CQ*)P+)-)
```

Prefix to infix

```python
operators = ['(', ')', '+', '-', '*', '/', '%', '^']
exp = "-+P*QCD"
P = 2
```

```
Q = 3

C = 1

D = 9

exp_r = exp[::-1]

print(exp_r)

for i in exp_r:

    if i.isalpha():

        s.push(i)

    elif i in operators:

        A = s.pop()

        B = s.pop()

        EXP = "("+A+i+B+")"


        print(EXP)

        s.push(str(EXP))
```

Output:

```
DCQ*P+-

(Q*C)

(P+(Q*C))

((P+(Q*C))-D)
```

<u>Prefix to postfix</u>

```
operators = ['(', ')', '+', '-', '*', '/', '%', '^']

exp = "-+P*QCD"
```

```python
exp_r = exp[::-1]

print(exp_r)

for i in exp_r:

    if i.isalpha():

        s.push(i)

    elif i in operators:

        A = s.pop()

        B = s.pop()

        EXP = "("+B+i+A+")"

        print(EXP)

        s.push(str(EXP))

postfix = ""

s = stack(50)

infix = EXP

operators = ['(', ')', '+', '-', '*', '/', '%', '^']

preced = {'^': 3, '*': 2, '/': 2, '%': 2, '+': 1, '-': 1}

for ch in infix:

    if ch.isalpha():

        postfix += ch

    elif ch in operators:

        if ch == '(':

            s.push(ch)

        elif s.peek() == None or s.peek() == '(':

            s.push(ch)

        elif ch == ')':

            while s.peek() != '(':

                postfix += s.pop()

            s.pop()

        elif preced[ch] > preced[s.peek()]:
            s.push(ch)
        elif preced[ch] < preced[s.peek()]:
            while s.peek() != '(' and preced[ch] < preced[s.peek()]:
                postfix += s.pop()
            if s.peek() != '(' and preced[ch] == preced[s.peek()]:
                postfix += s.pop()
            s.push(ch)
        elif preced[ch] == preced[s.peek()] and ch == '^':
            s.push(ch)
        elif preced[ch] == preced[s.peek()] and ch != '^':
            while s.peek() != None and preced[ch] == preced[s.peek()]:
                postfix += s.pop()
            s.push(ch)
    print("{:>3}   {:<25} {}".format(ch, str(s), postfix))
while s.peek() != None:
    postfix += s.pop()
print(postfix)
```

Output:

```
DCQ*P+-
(C*Q)
((C*Q)+P)
(D-((C*Q)+P))
  (   ['(']
  D   ['(']                 D
  -   ['(', '-']            D
  (   ['(', '-', '(']       D
  (   ['(', '-', '(', '(']      D
  C   ['(', '-', '(', '(']      DC
  *   ['(', '-', '(', '(', '*'] DC
  Q   ['(', '-', '(', '(', '*'] DCQ
  )   ['(', '-', '(']       DCQ*
  +   ['(', '-', '(', '+']      DCQ*
  P   ['(', '-', '(', '+']      DCQ*P
  )   ['(', '-']            DCQ*P+
  )   []                    DCQ*P+-
DCQ*P+-
```