# DSA FINAL PROJECT

**Ma'am Farzeen Ashfaq**

**Laveezah Noor**

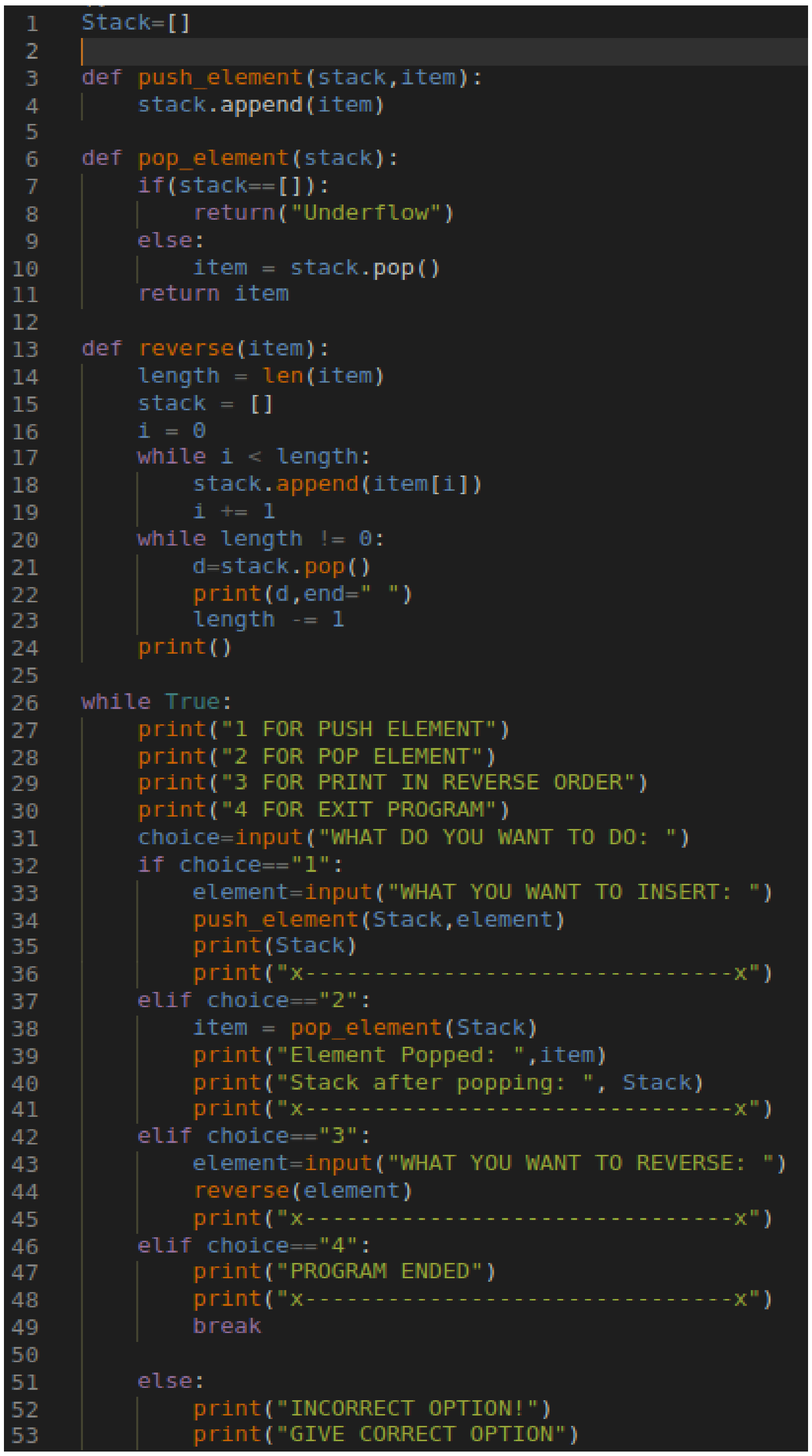
**BSSE-3rd Semester -A**

**EB-20103048**

**Q1. Write a program to perform following operations on stack.**

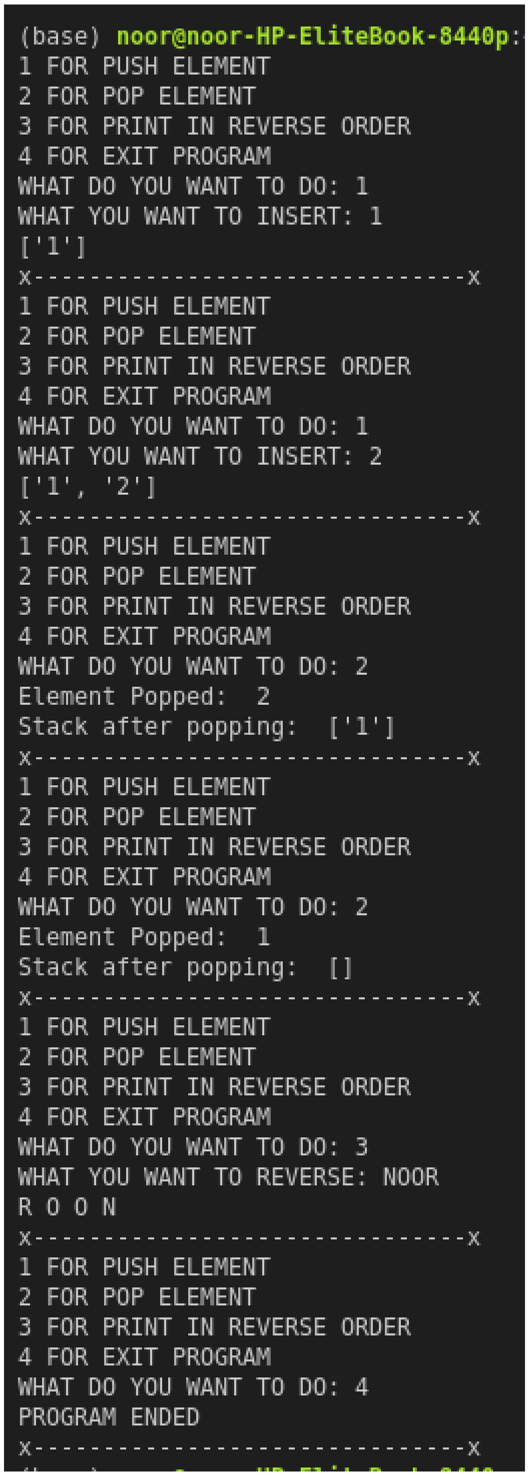
**a) Create functions for push and pop operations of stack. Insert your name and print it in reverse order.**

Code



```python
Stack=[]

def push_element(stack,item):
    stack.append(item)

def pop_element(stack):
    if(stack==[]):
        return("Underflow")
    else:
        item = stack.pop()
    return item

def reverse(item):
    length = len(item)
    stack = []
    i = 0
    while i < length:
        stack.append(item[i])
        i += 1
    while length != 0:
        d=stack.pop()
        print(d,end=" ")
        length -= 1
    print()

while True:
    print("1 FOR PUSH ELEMENT")
    print("2 FOR POP ELEMENT")
    print("3 FOR PRINT IN REVERSE ORDER")
    print("4 FOR EXIT PROGRAM")
    choice=input("WHAT DO YOU WANT TO DO: ")
    if choice=="1":
        element=input("WHAT YOU WANT TO INSERT: ")
        push_element(Stack,element)
        print(Stack)
        print("x--------------------------------x")
    elif choice=="2":
        item = pop_element(Stack)
        print("Element Popped: ",item)
        print("Stack after popping: ", Stack)
        print("x--------------------------------x")
    elif choice=="3":
        element=input("WHAT YOU WANT TO REVERSE: ")
        reverse(element)
        print("x--------------------------------x")
    elif choice=="4":
        print("PROGRAM ENDED")
        print("x--------------------------------x")
        break

    else:
        print("INCORRECT OPTION!")
        print("GIVE CORRECT OPTION")
```

Output



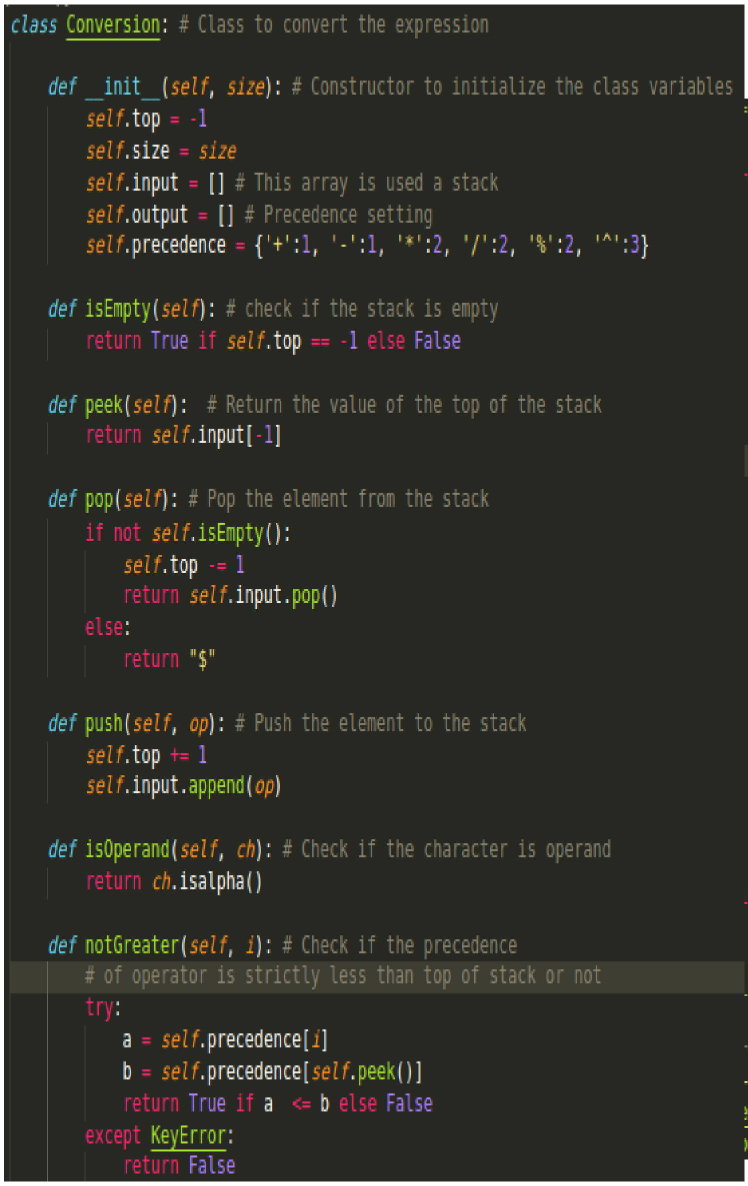
```
(base) noor@noor-HP-EliteBook-8440p:
1 FOR PUSH ELEMENT
2 FOR POP ELEMENT
3 FOR PRINT IN REVERSE ORDER
4 FOR EXIT PROGRAM
WHAT DO YOU WANT TO DO: 1
WHAT YOU WANT TO INSERT: 1
['1']
x--------------------------------x
1 FOR PUSH ELEMENT
2 FOR POP ELEMENT
3 FOR PRINT IN REVERSE ORDER
4 FOR EXIT PROGRAM
WHAT DO YOU WANT TO DO: 1
WHAT YOU WANT TO INSERT: 2
['1', '2']
x--------------------------------x
1 FOR PUSH ELEMENT
2 FOR POP ELEMENT
3 FOR PRINT IN REVERSE ORDER
4 FOR EXIT PROGRAM
WHAT DO YOU WANT TO DO: 2
Element Popped:  2
Stack after popping:  ['1']
x--------------------------------x
1 FOR PUSH ELEMENT
2 FOR POP ELEMENT
3 FOR PRINT IN REVERSE ORDER
4 FOR EXIT PROGRAM
WHAT DO YOU WANT TO DO: 2
Element Popped:  1
Stack after popping:  []
x--------------------------------x
1 FOR PUSH ELEMENT
2 FOR POP ELEMENT
3 FOR PRINT IN REVERSE ORDER
4 FOR EXIT PROGRAM
WHAT DO YOU WANT TO DO: 3
WHAT YOU WANT TO REVERSE: NOOR
R O O N
x--------------------------------x
1 FOR PUSH ELEMENT
2 FOR POP ELEMENT
3 FOR PRINT IN REVERSE ORDER
4 FOR EXIT PROGRAM
WHAT DO YOU WANT TO DO: 4
PROGRAM ENDED
x--------------------------------x
```

## Method

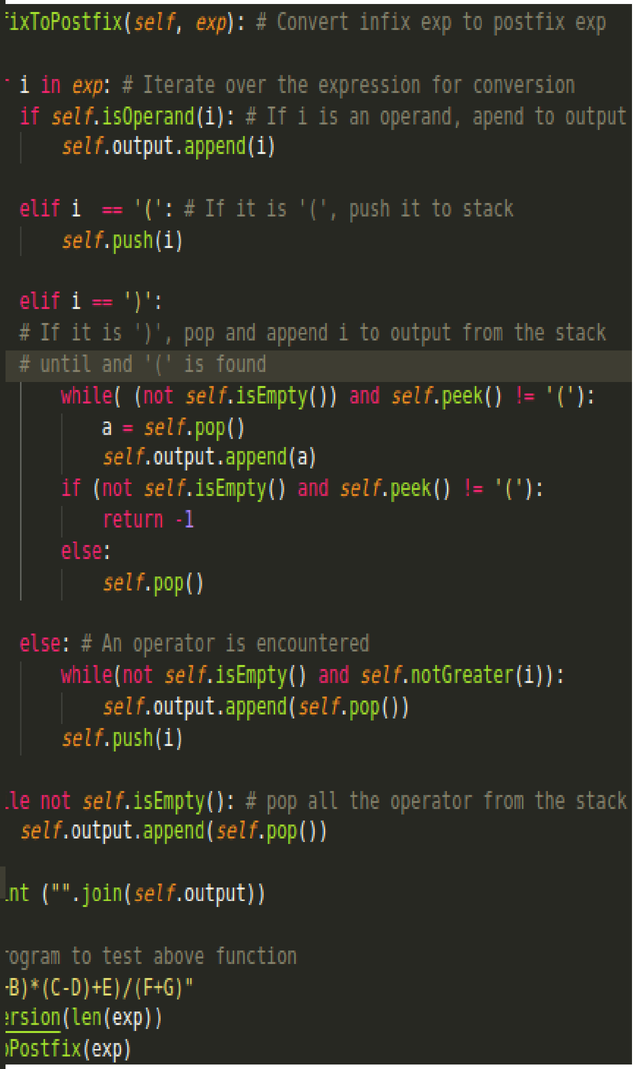
- Program asks user to choose from the given four choices.
- If user chooses option 1, then it will push the given element by the user in the stack and print the resultant stack.
- If user chooses option 2, then it will pop the element from the stack and print the resultant stack. Since,  stack works on the principle Last In First Out (LIFO), so it will delete the last element of stack.
- If user chooses option 3, then it will print the given item by the user in reverse order with help of LIFO principle of stack.
- If user chooses option 4, then it will end the program.

**b) Write a function to convert an infix expression to post fix expression. Pass a one dimensional character array P to the function as input (infix exp) and return character array Q (post fix exp). Test your program on your own input expression.**

Code

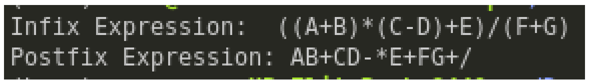


```
class Conversion: # Class to convert the expression

    def __init__(self, size): # Constructor to initialize the class variables
        self.top = -1
        self.size = size
        self.input = [] # This array is used a stack
        self.output = [] # Precedence setting
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '%':2, '^':3}

    def isEmpty(self): # check if the stack is empty
        return True if self.top == -1 else False

    def peek(self):  # Return the value of the top of the stack
        return self.input[-1]

    def pop(self): # Pop the element from the stack
        if not self.isEmpty():
            self.top -= 1
            return self.input.pop()
        else:
            return "$"

    def push(self, op): # Push the element to the stack
        self.top += 1
        self.input.append(op)

    def isOperand(self, ch): # Check if the character is operand
        return ch.isalpha()

    def notGreater(self, i): # Check if the precedence
        # of operator is strictly less than top of stack or not
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a  <= b else False
        except KeyError:
            return False
```



```
ixToPostfix(self, exp): # Convert infix exp to postfix exp

 i in exp: # Iterate over the expression for conversion
 if self.isOperand(i): # If i is an operand, apend to output
     self.output.append(i)

 elif i  == '(': # If it is '(', push it to stack
     self.push(i)

 elif i == ')':
 # If it is ')', pop and append i to output from the stack
 # until and '(' is found
     while( (not self.isEmpty()) and self.peek() != '('):
         a = self.pop()
         self.output.append(a)
     if (not self.isEmpty() and self.peek() != '('):
         return -1
     else:
         self.pop()

 else: # An operator is encountered
     while(not self.isEmpty() and self.notGreater(i)):
         self.output.append(self.pop())
     self.push(i)

le not self.isEmpty(): # pop all the operator from the stack
 self.output.append(self.pop())

nt ("".join(self.output))

ogram to test above function
B)*(C-D)+E)/(F+G)"
rsion(len(exp))
Postfix(exp)
```

Output



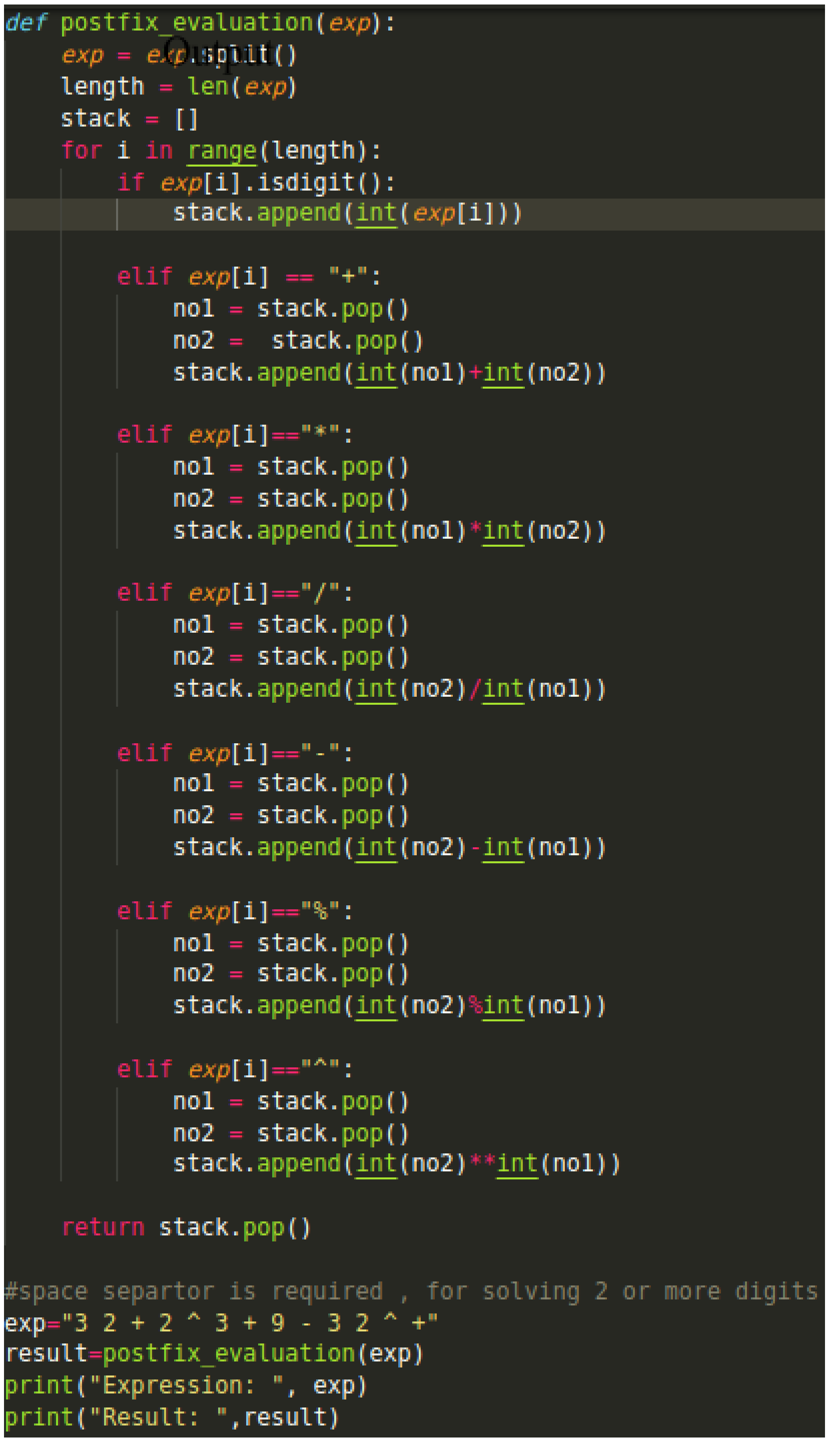
```
Infix Expression:  ((A+B)*(C-D)+E)/(F+G)
Postfix Expression: AB+CD-*E+FG+/
```

## Method

- We created a class to convert the expression, in which we initialized an empty stack as input and precedence levels of the operators and an empty stack as output.
- Then we created some functions that are
    - IsEmpty function checks whether the input stack is empty or not.
    - Peek function returns the top value of the input stack.
    - Push function appends the given item to the input stack.
    - Pop function deletes the last item of the input stack if input stack is not empty else it will return $.
    - IsOperand function checks if the given character is operand.
    - NotGreater function check if the precedence level of the operator is strictly less than the top of the stack or not.
- The main function **InfixtoPostfix** function
    - iterates all over the expression to do the conversion and checks if the character is operand or open bracket or close bracket or operator.
    - If the character is operand, then it will append it in the output stack.
    - If the character is open bracket, then it will push it in input stack.
    - If the character is close bracket, while input stack is not empty and top item of input stack is not open bracket, then pop the items from input stack and append them in output stack, else pop the items from input stack.
    - Else if operator is encountered, while input stack is not empty and precedence of the top of the input stack than the given operator then pop and append the items of input stack to the output stack.
    - In the end, pop all the operators of the input stack and append them into output stack. In the last, print all items of output stack.

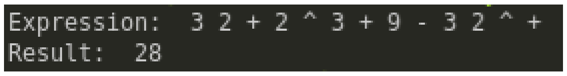
**c) Write a function for the evaluation of a given Post-fix expression. For testing pass the Post-fix expression Q of part b and supply your own set of values.**

Code



```python
def postfix_evaluation(exp):
    exp = exp.split()
    length = len(exp)
    stack = []
    for i in range(length):
        if exp[i].isdigit():
            stack.append(int(exp[i]))

        elif exp[i] == "+":
            no1 = stack.pop()
            no2 =  stack.pop()
            stack.append(int(no1)+int(no2))

        elif exp[i]=="*":
            no1 = stack.pop()
            no2 = stack.pop()
            stack.append(int(no1)*int(no2))

        elif exp[i]=="/":
            no1 = stack.pop()
            no2 = stack.pop()
            stack.append(int(no2)/int(no1))

        elif exp[i]=="-":
            no1 = stack.pop()
            no2 = stack.pop()
            stack.append(int(no2)-int(no1))

        elif exp[i]=="%":
            no1 = stack.pop()
            no2 = stack.pop()
            stack.append(int(no2)%int(no1))

        elif exp[i]=="^":
            no1 = stack.pop()
            no2 = stack.pop()
            stack.append(int(no2)**int(no1))

    return stack.pop()

#space separtor is required , for solving 2 or more digits
exp="3 2 + 2 ^ 3 + 9 - 3 2 ^ +"
result=postfix_evaluation(exp)
print("Expression: ", exp)
print("Result: ",result)
```

Output



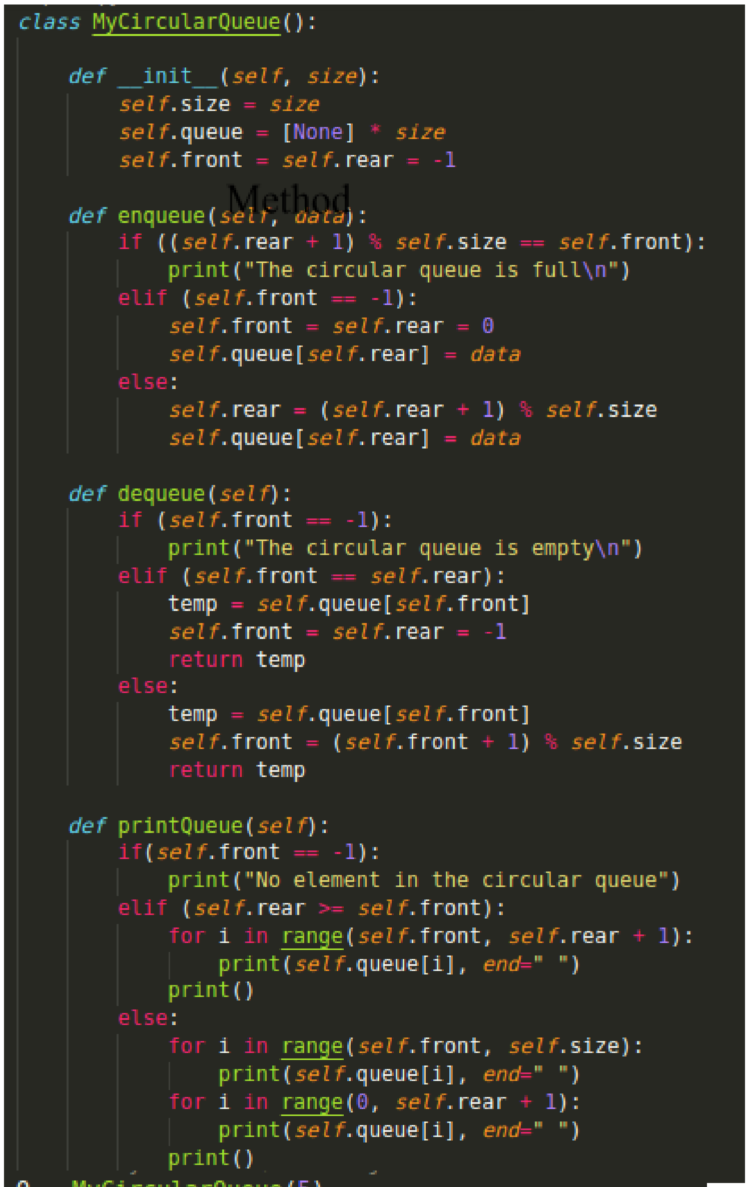
```
Expression:  3 2 + 2 ^ 3 + 9 - 3 2 ^ +
Result:  28
```

Method

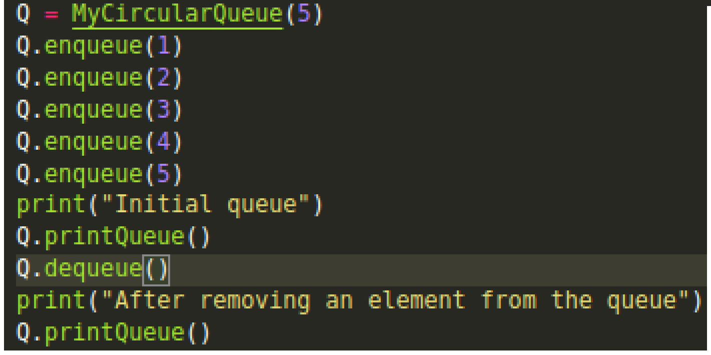
- We created a function **postfix_evaluation** to evaluate the given postfix expression and return the result of it.
- It takes the given expression as input and iterate all over the expression, if the character is digit, it will be appended to the stack.
- Otherwise, if character is any operator, then it will pop the two digits from the stack and operate them according to the operator and then append the result in the stack. This process will continue till the last character of the expression.
- And in the end it will return the result.

**Q2. Write a program using functions for implementation of circular Queue.**

Code

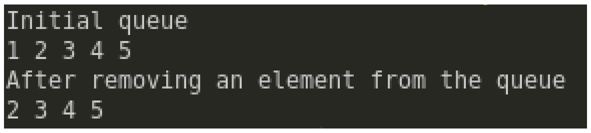


```python
class MyCircularQueue():

    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = self.rear = -1

    def enqueue(self, data):
        if ((self.rear + 1) % self.size == self.front):
            print("The circular queue is full\n")
        elif (self.front == -1):
            self.front = self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data

    def dequeue(self):
        if (self.front == -1):
            print("The circular queue is empty\n")
        elif (self.front == self.rear):
            temp = self.queue[self.front]
            self.front = self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp

    def printQueue(self):
        if(self.front == -1):
            print("No element in the circular queue")
        elif (self.rear >= self.front):
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end=" ")
            print()
        else:
            for i in range(self.front, self.size):
                print(self.queue[i], end=" ")
            for i in range(0, self.rear + 1):
                print(self.queue[i], end=" ")
            print()
```



```python
Q = MyCircularQueue(5)
Q.enqueue(1)
Q.enqueue(2)
Q.enqueue(3)
Q.enqueue(4)
Q.enqueue(5)
print("Initial queue")
Q.printQueue()
Q.dequeue()
print("After removing an element from the queue")
Q.printQueue()
```

Output



```
Initial queue
1 2 3 4 5
After removing an element from the queue
2 3 4 5
```

Method

- We created a class to declare the Circular Queue, in which we initialized the size of the queue given by the user and according to that we created a queue where the front and rear end is -1.
- It has three main functions of circular queue that are

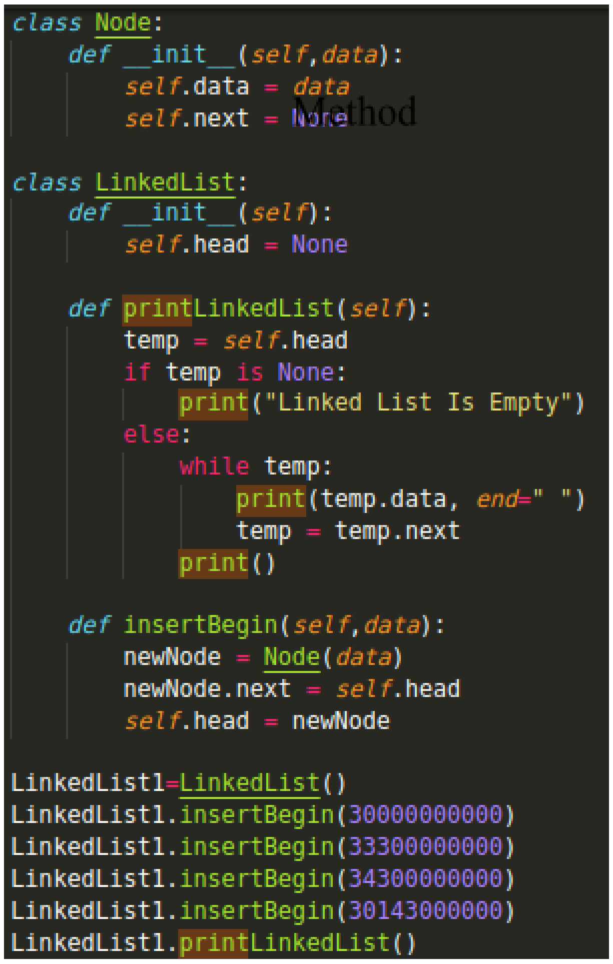
Enqueue, Dequeue and PrintQueue.

- Enqueue function inserts the given item into the circular queue at the rear end if the queue is not full.
- Dequeue function deletes the item of the circular queue at the front end.
- PrintQueue function prints the all the items of the circular queue.

**Q 3. Create a Phone Directory of your own friend's and relative's contacts using Linked List. Your program must allow performance of following operations:**

a) INSERTION AT THE BEGINNING.

Code



```python
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def printLinkedList(self):
        temp = self.head
        if temp is None:
            print("Linked List Is Empty")
        else:
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()

    def insertBegin(self,data):
        newNode = Node(data)
        newNode.next = self.head
        self.head = newNode

LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
```

Method

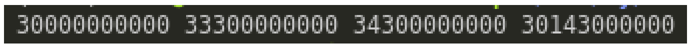
• We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node.

• We also created a class **LinkedList** to initialize linked list where head of the linked list is none.

• For insertion at the beginning of the linked list, we created a function **insertBegin**, which will insert the new node as the head of the linked list and previous head of the linked list as the next node to the head node.

• Function printLinkedList prints all the nodes of the linked list if linked list is not empty.

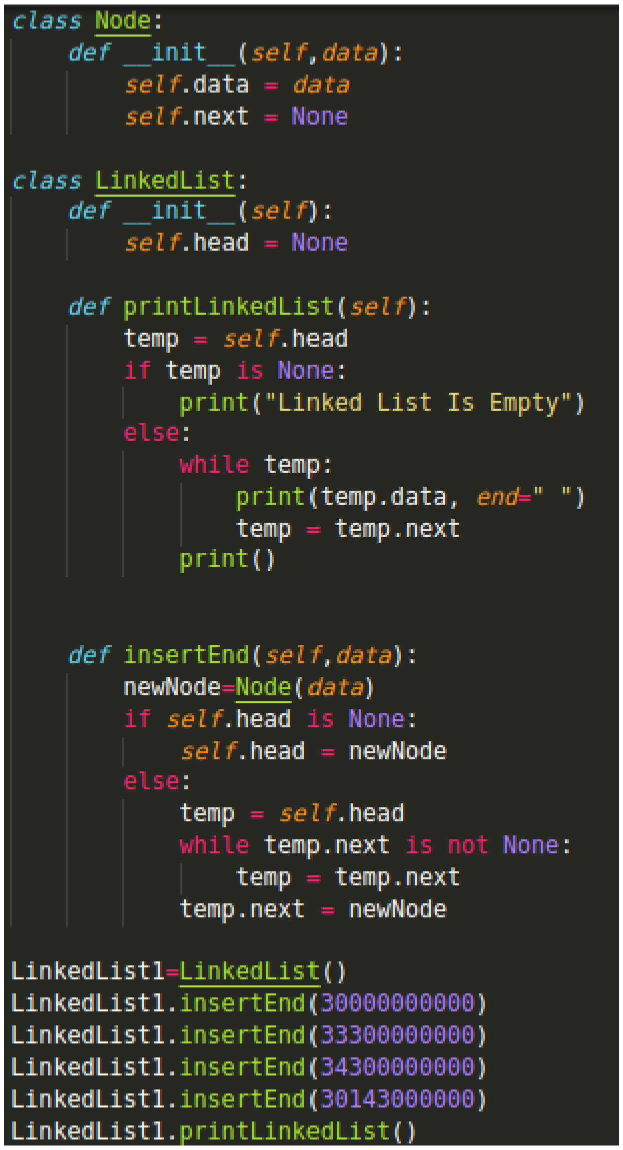
Output



```
30000000000 33300000000 34300000000 30143000000
```

b) INSERTION AT THE END.

Code

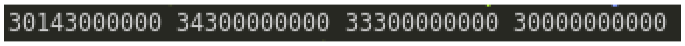


```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def printLinkedList(self):
        temp = self.head
        if temp is None:
            print("Linked List Is Empty")
        else:
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()


    def insertEnd(self,data):
        newNode=Node(data)
        if self.head is None:
            self.head = newNode
        else:
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = newNode

LinkedList1=LinkedList()
LinkedList1.insertEnd(30000000000)
LinkedList1.insertEnd(33300000000)
LinkedList1.insertEnd(34300000000)
LinkedList1.insertEnd(30143000000)
LinkedList1.printLinkedList()
```

Method

- We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node.
- We also created a class **LinkedList** to initialize linked list where head of the linked list is none.
- For insertion at the end of the linked list, we created a function **insertEnd**, which will insert the new node as the head of the linked list if the linked list is empty, else it will iterate all over the linked list and insert the new node at the last of linked list.
- Function printLinkedList prints all the nodes of the linked list if linked list is not empty.

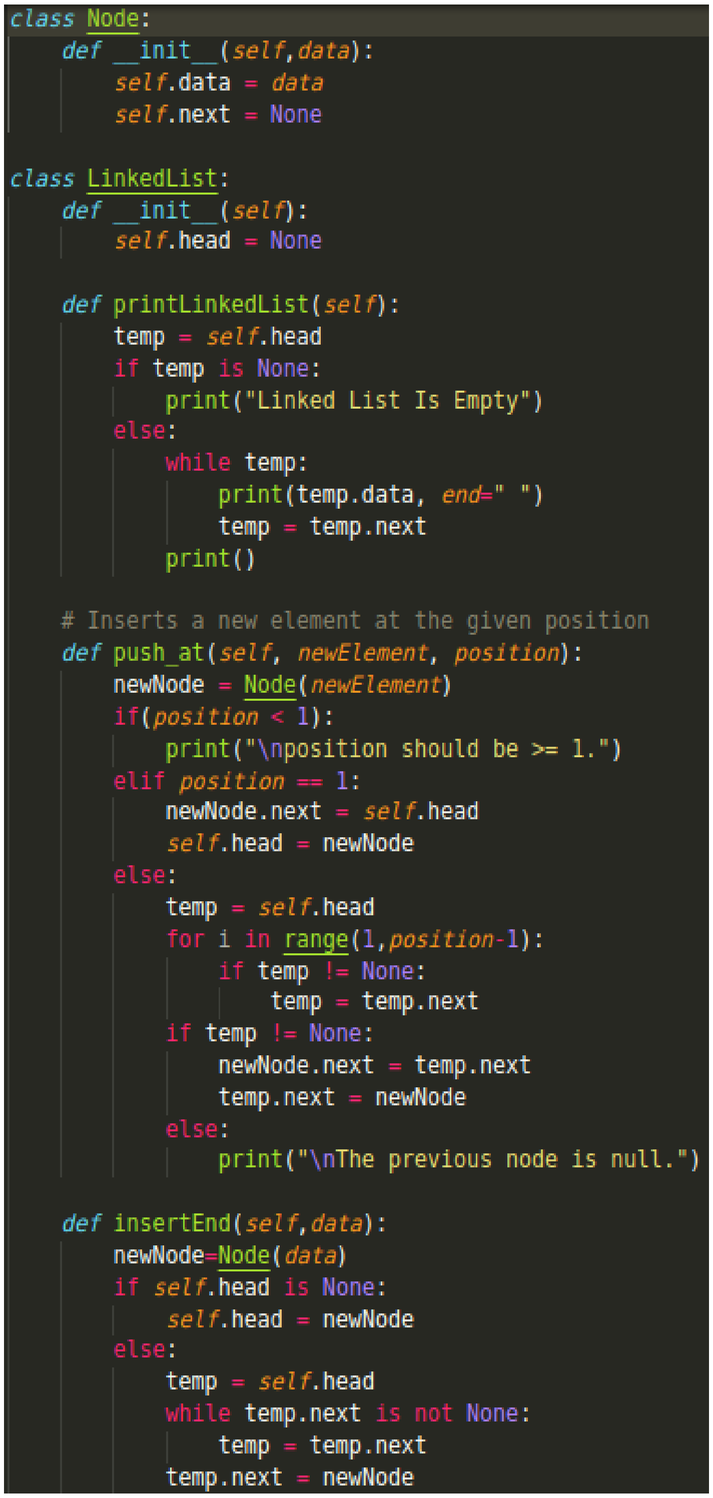
Output



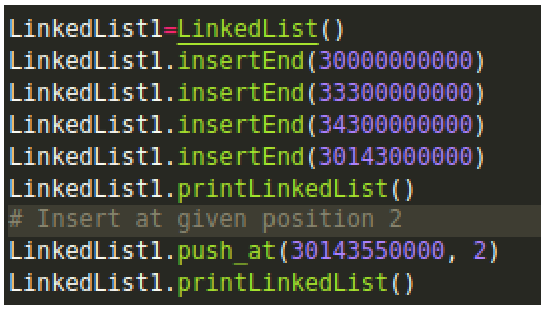
```
30143000000 34300000000 33300000000 30000000000
```

c) INSERTION AT THE GIVEN POSITION.

## Code

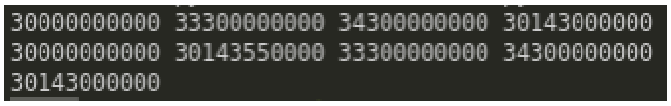


```python
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def printLinkedList(self):
        temp = self.head
        if temp is None:
            print("Linked List Is Empty")
        else:
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()

    # Inserts a new element at the given position
    def push_at(self, newElement, position):
        newNode = Node(newElement)
        if(position < 1):
            print("\nposition should be >= 1.")
        elif position == 1:
            newNode.next = self.head
            self.head = newNode
        else:
            temp = self.head
            for i in range(1,position-1):
                if temp != None:
                    temp = temp.next
            if temp != None:
                newNode.next = temp.next
                temp.next = newNode
            else:
                print("\nThe previous node is null.")

    def insertEnd(self,data):
        newNode=Node(data)
        if self.head is None:
            self.head = newNode
        else:
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = newNode
```



```python
LinkedList1=LinkedList()
LinkedList1.insertEnd(30000000000)
LinkedList1.insertEnd(33300000000)
LinkedList1.insertEnd(34300000000)
LinkedList1.insertEnd(30143000000)
LinkedList1.printLinkedList()
# Insert at given position 2
LinkedList1.push_at(30143550000, 2)
LinkedList1.printLinkedList()
```

## Output



```
30000000000 33300000000 34300000000 30143000000
30000000000 30143550000 33300000000 34300000000
30143000000
```

## Method

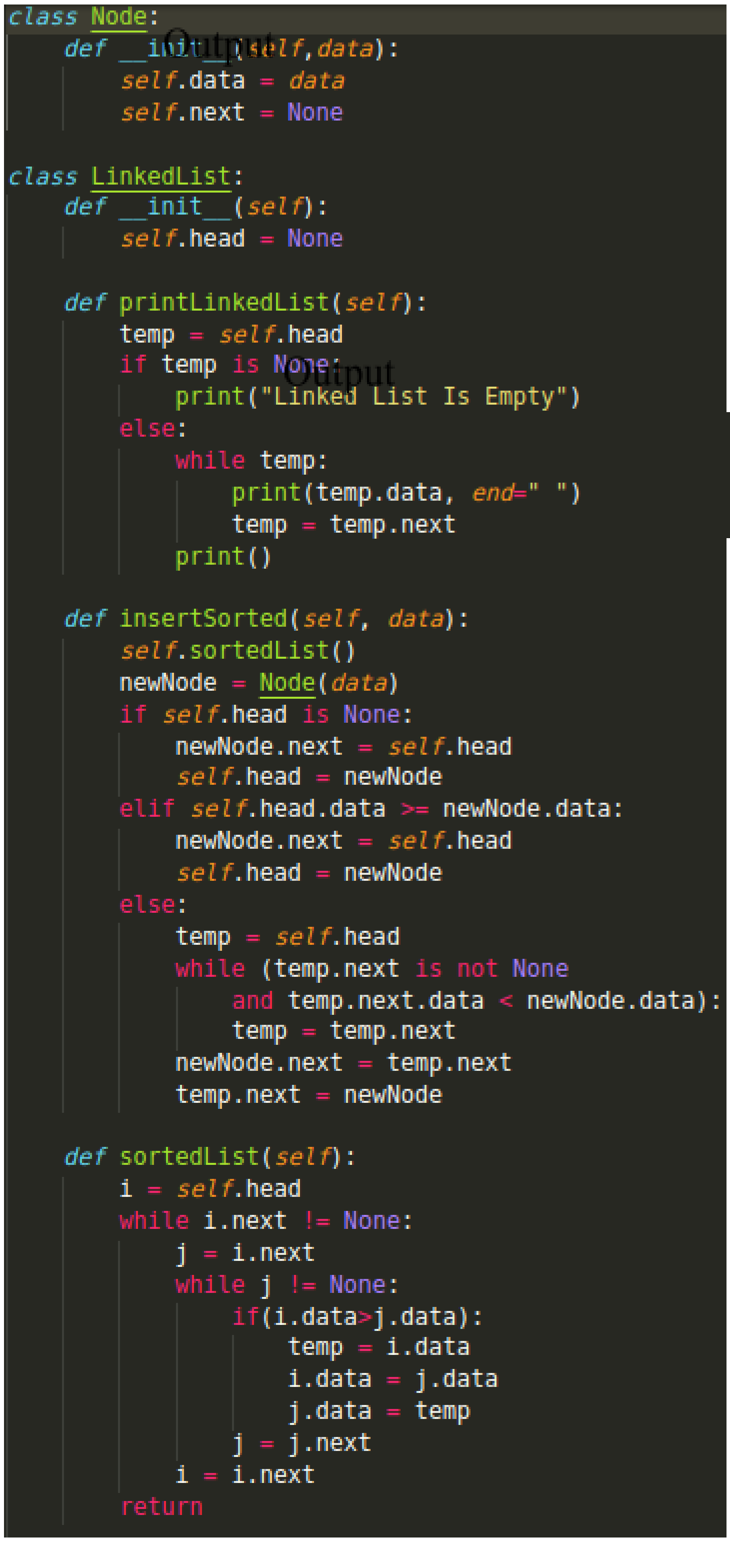
• We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.

• For insertion at the given position of the linked list, we created a function **push_at**, which will insert the new node as the head of the linked list if the position is 1, else it will iterate all over the linked list and insert the new node at the given position of the linked list where the previous node of the position is not null, while position is not less than 1.

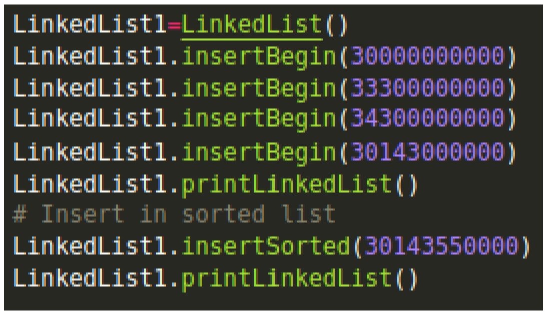
• Other functions are same as above.

d) INSERTION IN THE SORTED LIST.

## Code

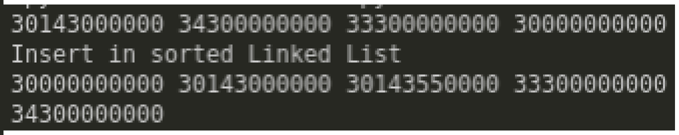


```python
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def printLinkedList(self):
        temp = self.head
        if temp is None:
            print("Linked List Is Empty")
        else:
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()

    def insertSorted(self, data):
        self.sortedList()
        newNode = Node(data)
        if self.head is None:
            newNode.next = self.head
            self.head = newNode
        elif self.head.data >= newNode.data:
            newNode.next = self.head
            self.head = newNode
        else:
            temp = self.head
            while (temp.next is not None
                and temp.next.data < newNode.data):
                temp = temp.next
            newNode.next = temp.next
            temp.next = newNode

    def sortedList(self):
        i = self.head
        while i.next != None:
            j = i.next
            while j != None:
                if(i.data>j.data):
                    temp = i.data
                    i.data = j.data
                    j.data = temp
                j = j.next
            i = i.next
        return
```



```python
LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
# Insert in sorted list
LinkedList1.insertSorted(30143550000)
LinkedList1.printLinkedList()
```

## Output



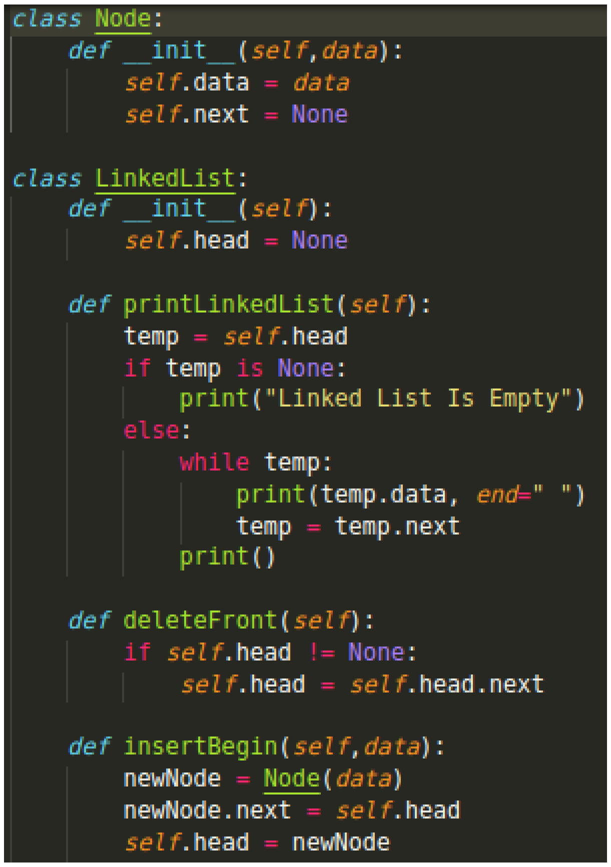
```
30143000000 34300000000 33300000000 30000000000
Insert in sorted Linked List
30000000000 30143000000 30143550000 33300000000
34300000000
```

## Method

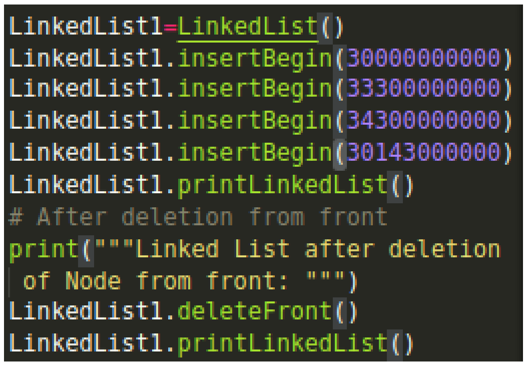
- We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.
- For insertion in the sorted linked list, we created a function **sortedlist**, which sorts the linked list according to the data of the node in the ascending order.
- We also created a **insertSorted** function, which calls the function sortedlist, then it inserts the the new node according to its data in the linked list.
- Other functions are same as above.

e) DELETION FROM THE FIRST NODE.

Code

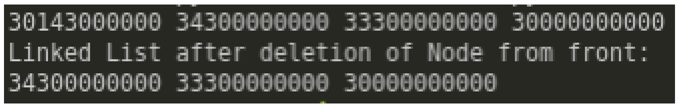


```python
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def printLinkedList(self):
        temp = self.head
        if temp is None:
            print("Linked List Is Empty")
        else:
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()

    def deleteFront(self):
        if self.head != None:
            self.head = self.head.next

    def insertBegin(self,data):
        newNode = Node(data)
        newNode.next = self.head
        self.head = newNode
```



```python
LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
# After deletion from front
print("""Linked List after deletion
 of Node from front: """)
LinkedList1.deleteFront()
LinkedList1.printLinkedList()
```

Output



```
30143000000 34300000000 33300000000 30000000000
Linked List after deletion of Node from front:
34300000000 33300000000 30000000000
```

Method

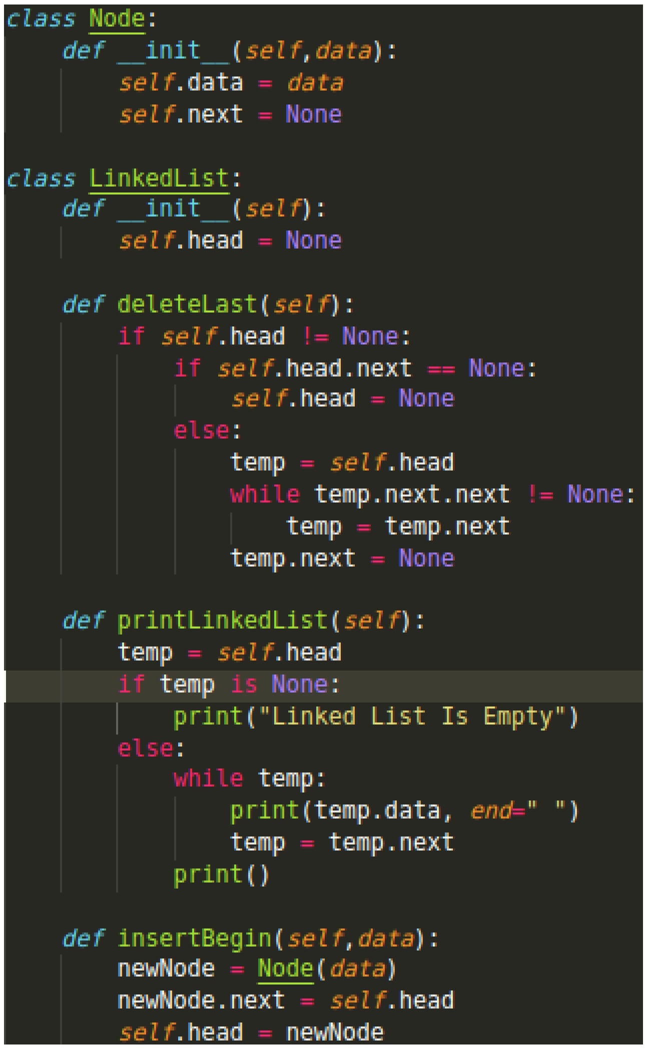
• We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.

• For deletion from the first node in the linked list, we created a function **deleteFront**, which deletes the first node and replace it with the next node.

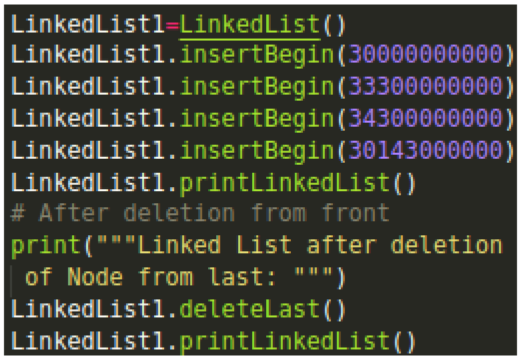
• Other functions are same as above.

e) DELETION FROM THE LAST NODE.

Code

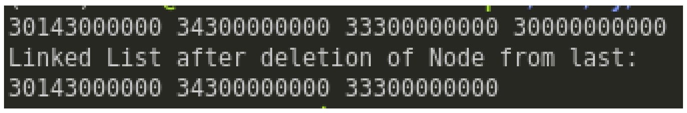


```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def deleteLast(self):
        if self.head != None:
            if self.head.next == None:
                self.head = None
            else:
                temp = self.head
                while temp.next.next != None:
                    temp = temp.next
                temp.next = None

    def printLinkedList(self):
        temp = self.head
        if temp is None:
            print("Linked List Is Empty")
        else:
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()

    def insertBegin(self,data):
        newNode = Node(data)
        newNode.next = self.head
        self.head = newNode
```



```
LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
# After deletion from front
print("""Linked List after deletion
 of Node from last: """)
LinkedList1.deleteLast()
LinkedList1.printLinkedList()
```

Output



```
30143000000 34300000000 33300000000 30000000000
Linked List after deletion of Node from last:
30143000000 34300000000 33300000000
```

Method

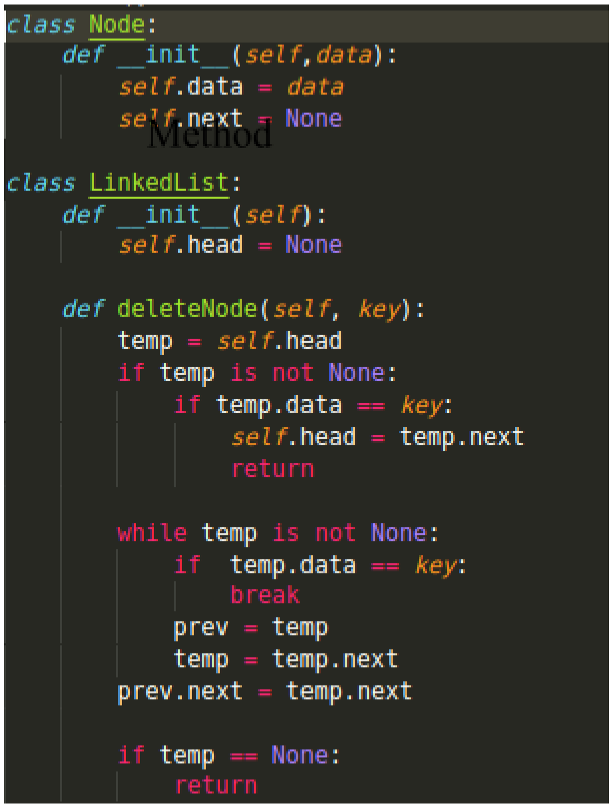
• We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.

• For deletion from the last node in the linked list, we created a function **deleteLast**, which iterates all over the linked list till the last node and deletes it if the linked list is not empty.

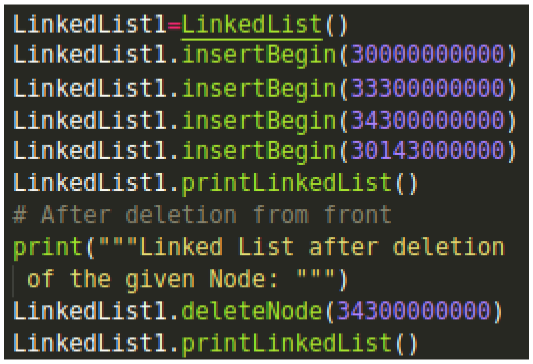
• Other functions are same as above.

f) DELETION OF THE GIVEN NODE.

## Code



```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def deleteNode(self, key):
        temp = self.head
        if temp is not None:
            if temp.data == key:
                self.head = temp.next
                return

        while temp is not None:
            if  temp.data == key:
                break
            prev = temp
            temp = temp.next
        prev.next = temp.next

        if temp == None:
            return
```



```
LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
# After deletion from front
print("""Linked List after deletion
 of the given Node: """)
LinkedList1.deleteNode(34300000000)
LinkedList1.printLinkedList()
```

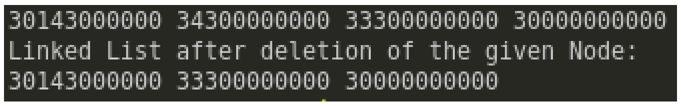
## Method

• We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.

• For deletion of the given node in the linked list, we created a function **deleteNode**, which iterates all over the linked list till the last node and deletes the given node and replaces it with previous node if the linked list is not empty and given node is present in the linked list.

• Other functions are same as above.

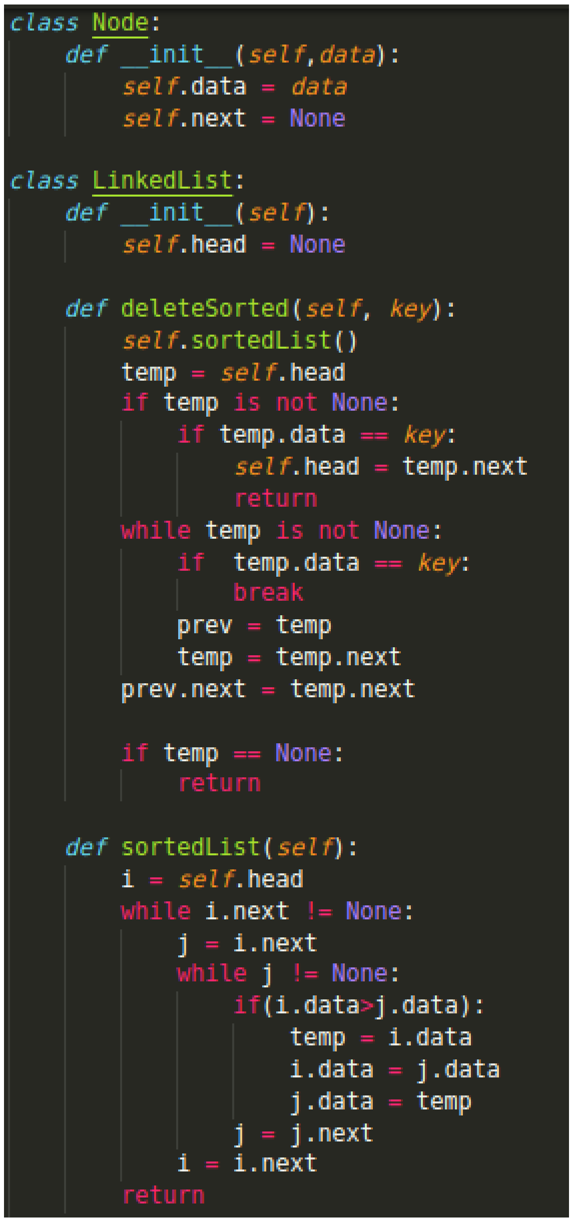
## Output



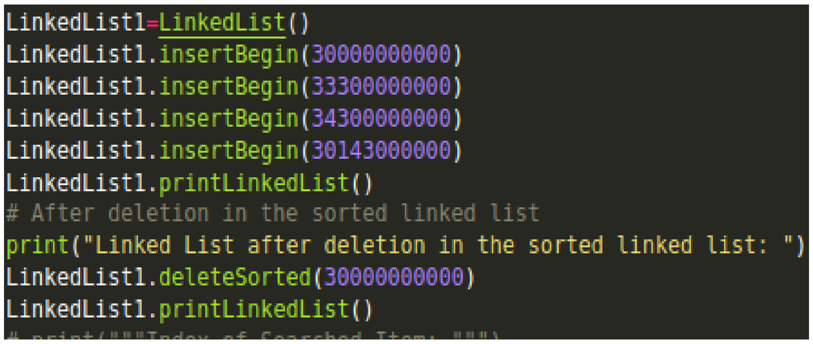
```
30143000000 34300000000 33300000000 30000000000
Linked List after deletion of the given Node:
30143000000 33300000000 30000000000
```

g) DELETION IN THE SORTED LINKED LIST.

## Code

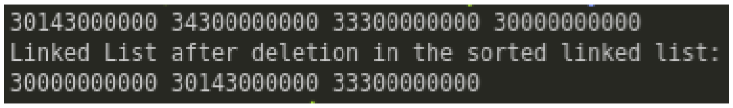


```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def deleteSorted(self, key):
        self.sortedList()
        temp = self.head
        if temp is not None:
            if temp.data == key:
                self.head = temp.next
                return
        while temp is not None:
            if  temp.data == key:
                break
            prev = temp
            temp = temp.next
        prev.next = temp.next

        if temp == None:
            return

    def sortedList(self):
        i = self.head
        while i.next != None:
            j = i.next
            while j != None:
                if(i.data>j.data):
                    temp = i.data
                    i.data = j.data
                    j.data = temp
                j = j.next
            i = i.next
        return
```



```
LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
# After deletion in the sorted linked list
print("Linked List after deletion in the sorted linked list: ")
LinkedList1.deleteSorted(30000000000)
LinkedList1.printLinkedList()
```

## Output



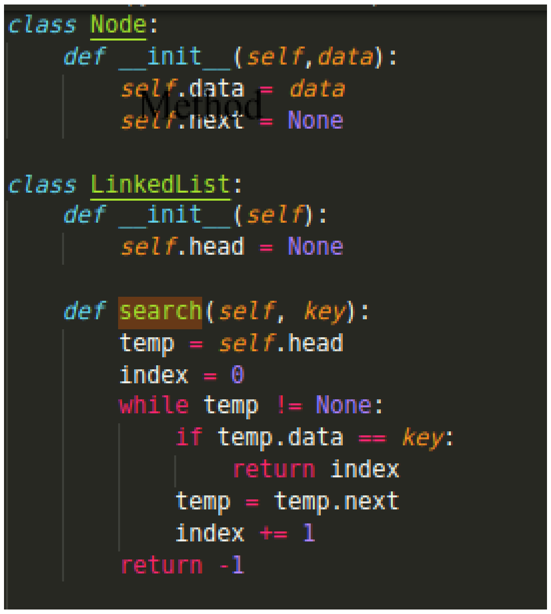
```
30143000000 34300000000 33300000000 30000000000
Linked List after deletion in the sorted linked list:
30000000000 30143000000 33300000000
```

## Method

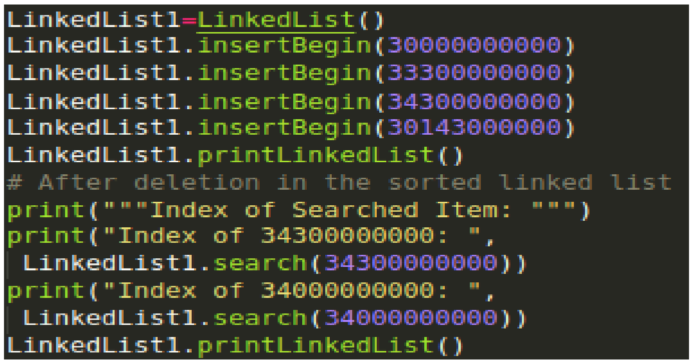
- We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.
- For deletion in the sorted linked list, we created a function **deleteSorted**, which calls the sort function which sorts the linked list and then delete the given item of the linked list.
- Other functions are same as above.

## h) SEARCHING

### Code

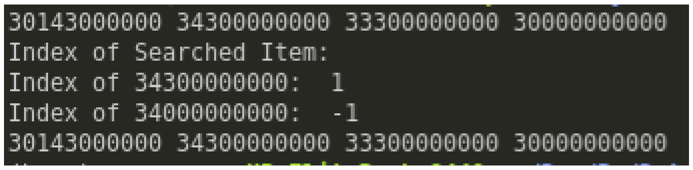


```python
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def search(self, key):
        temp = self.head
        index = 0
        while temp != None:
            if temp.data == key:
                return index
            temp = temp.next
            index += 1
        return -1
```



```python
LinkedList1=LinkedList()
LinkedList1.insertBegin(30000000000)
LinkedList1.insertBegin(33300000000)
LinkedList1.insertBegin(34300000000)
LinkedList1.insertBegin(30143000000)
LinkedList1.printLinkedList()
# After deletion in the sorted linked list
print("""Index of Searched Item: """)
print("Index of 34300000000: ",
 LinkedList1.search(34300000000))
print("Index of 34000000000: ",
 LinkedList1.search(34000000000))
LinkedList1.printLinkedList()
```

### Output



```
30143000000 34300000000 33300000000 30000000000
Index of Searched Item:
Index of 34300000000:  1
Index of 34000000000:  -1
30143000000 34300000000 33300000000 30000000000
```

### Method

• We created a class Node to declare the Node, in which we initialized the data of the node and the reference of the next node and also created a class **LinkedList** to initialize linked list where head of the linked list is none like above.

• For searching of the given node in the linked list, we created a function **search**, which iterates all over the linked list till the given node is found and return its index if the given node is present in the linked list or else it will return -1.

• Other functions are same as above.