```
#################################################################################
##########################################################

# کلاس استک

class Stack:
    def __init__(self, limit=10):
        self.stack = []
        self.limit = limit

    def peek(self):
        if len(self.stack) != 0:
            return -1
        else:
            return self.stack[-1]

    def push(self, data):
        if len(self.stack) > self.limit:
            return -1
        else:
            self.stack.append(data)

    def pop(self):

        if len(self.stack) >= 0:
            return self.stack.pop()
        else:
            return -1

    def find(self, data):
        if len(self.stack) <= 0:
            return -1
        for i in range(1, len(self.stack)):
            if self.stack[i] == data:
                return i
        return -1
    def replace(self, data, new_data):
        if len(self.stack) <= 0:
            return -1
        else:
            if data in self.stack:
                self.stack[self.stack.index(data)] = new_data
            else:
                return -1

    def show(self):
        for i in self.stack:
            print(i)

    def is_empty(self):
        return len(self.stack) == 0

    def __len__(self):
        return len(self.stack)
```

```
###############################################################################
########################################################

# تبدیل دسیمال به باینری

def d2b(number):
    my_stack = Stack()
    while number !=0:
        my_stack.push(number % 2)
        number //= 2
    while not(my_stack.is_empty()):
        print(my_stack.pop(), end='')

###############################################################################
########################################################

# کلاس صف

class Queue:
    def __init__(self, limit=100):
        self.Q = [None] * limit
        self.limit = limit
        self.front = -1
        self.rear = -1

    def enqueue(self, data):
        if self.rear >= self.limit-1:
            return -1
        elif self.front == -1:
            self.front= 0
            self.rear = 0
            self.Q[0] = data
        else:
            self.rear += 1
            self.Q[self.rear] = data

    def dequeue(self):
        if self.front == -1:
            return -1
        elif self.rear < self.front:
            return -1
        else:
            self.front += 1

    def show(self):
        for i in range(self.front, self.rear+1):
            print(self.Q[i])

    def replace(self, data, place):
        if self.place > self.front and self.place < self.rear:
            self.Q[place] = data

###############################################################################
########################################################
```

```
# کلاس صف حلقوی

class Cqueue:
    def __init__(self, limit=10) -> None:
        self.Q = [None] * limit
        self.limit = limit
        self.rear = -1
        self.front = -1

    def insert_queue(self, data):
        if (self.rear+1) % self.limit == self.front:
            print('full')
            return -1

        elif self.front == -1:
            self.front += 1
            self.rear += 1
            self.Q[0] = data

        else:
            self.rear  = (self.rear+1) % self.limit
            self.Q[self.rear] = data

    def delete_queue(self):
        if self.front == -1:
            # print("empty")
            return None

        elif self.front == self.rear:
            a = self.front
            self.front= -1
            self.rear = -1
            return self.Q[a]

        else:
            b = self.front
            self.front = (self.front+1) % self.limit
            return self.Q[b]

    def display(self):
        if self.rear == -1 and self.front == -1:
            print("empty")
            return -1
        elif self.front <= self.rear:
            for i in range(self.front, self.rear+1):
                print(self.Q[i])
        else:
            for i in range(self.front, self.limit):
                print(self.Q[i])

            for i in range(0, self.rear + 1):
                print(self.Q[i])
```

```
    def replace(self, data, new_data):
        flag = False
        if self.rear == -1 and self.front == -1:
            print("empty")
        else:
            if self.front< self.rear:
                for i in range(self.front, self.rear + 1):
                    if self.Q[i] == data:
                        self.Q[i] =  new_data
                        flag = True
            elif self.front == self.rear:
                if self.Q[self.front] == data:
                    self.Q[self.front] = new_data
                    flag = True
            elif self.front > self.rear:
                for i in range(self.front, self.limit):
                    if self.Q[i] == data:
                        self.Q[i] = new_data
                        flag = True
                for i in range(0, self.rear+1):
                    if self.Q[i] == data:
                        self.Q[i] =  new_data
                        flag = True
        if not(flag):
            print("Your data not found")

##########################################################################################
################################################################

# برعکس کردن استک  بوسیله صف

class Stack:
    def __init__(self, limit=10):
        self.stack = []
        self.limit = limit

    def peek(self):
        if len(self.stack) != 0:
            return -1
        else:
            return self.stack[-1]

    def push(self, data):
        if len(self.stack) > self.limit:
            return -1
        else:
            self.stack.append(data)

    def pop(self):

        if len(self.stack) >= 0:
            return self.stack.pop()
        else:
            return -1
```

```python
    def find(self, data):
        if len(self.stack) <= 0:
            return -1
        for i in range(1, len(self.stack)):
            if self.stack[i] == data:
                return i
        return -1
    def replace(self, data, new_data):
        if len(self.stack) <= 0:
            return -1
        else:
            if data in self.stack:
                self.stack[self.stack.index(data)] = new_data
            else:
                return -1

    def show(self):
        for i in self.stack[::-1]:
            print(i)

    def is_empty(self):
        return len(self.stack) == 0

    def __len__(self):
        return len(self.stack)

class Cqueue:
    def __init__(self, limit=10) -> None:
        self.Q = [None] * limit
        self.limit = limit
        self.rear = -1
        self.front = -1

    def insert_queue(self, data):
        if (self.rear+1) % self.limit == self.front:
            print('full')
            return -1

        elif self.front == -1:
            self.front += 1
            self.rear += 1
            self.Q[0] = data

        else:
            self.rear  = (self.rear+1) % self.limit
            self.Q[self.rear] = data

    def delete_queue(self):
        if self.front == -1:
            # print("empty")
            return None

        elif self.front == self.rear:
            a = self.front
            self.front= -1
```

```
            self.rear = -1
            return self.Q[a]
        else:
            b = self.front
            self.front = (self.front+1) % self.limit
            return self.Q[b]

    def display(self):
        if self.rear == -1 and self.front == -1:
            print("empty")
            return -1
        elif self.front <= self.rear:
            for i in range(self.front, self.rear+1):
                print(self.Q[i])
        else:
            for i in range(self.front, self.limit):
                print(self.Q[i])

            for i in range(0, self.rear + 1):
                print(self.Q[i])

    def replace(self, data, new_data):
        flag = False
        if self.rear == -1 and self.front == -1:
            print("empty")
        else:
            if self.front< self.rear:
                for i in range(self.front, self.rear + 1):
                    if self.Q[i] == data:
                        self.Q[i] =  new_data
                        flag = True
            elif self.front == self.rear:
                if self.Q[self.front] == data:
                    self.Q[self.front] = new_data
                    flag = True
            elif self.front > self.rear:
                for i in range(self.front, self.limit):
                    if self.Q[i] == data:
                        self.Q[i] = new_data
                        flag = True
                for i in range(0, self.rear+1):
                    if self.Q[i] == data:
                        self.Q[i] =  new_data
                        flag = True
        if not(flag):
            print("Your data not found")

def reverse_data(stack: Stack):
    Q = Cqueue()
    n_stack = Stack()
    while not (stack.is_empty()):
        Q.insert_queue(stack.pop())

    while Q.front != -1:
        n_stack.push(Q.delete_queue())
```

```
    n_stack.show()

##########################################################################################
##############################################################

# برعکس کردن صف بوسیله استک

class Stack:
    def __init__(self, limit=10):
        self.stack = []
        self.limit = limit

    def peek(self):
        if len(self.stack) != 0:
            return -1
        else:
            return self.stack[-1]

    def push(self, data):
        if len(self.stack) > self.limit:
            return -1
        else:
            self.stack.append(data)

    def pop(self):

        if len(self.stack) >= 0:
            return self.stack.pop()
        else:
            return -1

    def find(self, data):
        if len(self.stack) <= 0:
            return -1
        for i in range(1, len(self.stack)):
            if self.stack[i] == data:
                return i
        return -1
    def replace(self, data, new_data):
        if len(self.stack) <= 0:
            return -1
        else:
            if data in self.stack:
                self.stack[self.stack.index(data)] = new_data
            else:
                return -1

    def show(self):
        for i in self.stack:
            print(i)

    def is_empty(self):
        return len(self.stack) == 0
```

```python
    def __len__(self):
        return len(self.stack)

class Cqueue:
    def __init__(self, limit=10) -> None:
        self.Q = [None] * limit
        self.limit = limit
        self.rear = -1
        self.front = -1

    def insert_queue(self, data):
        if (self.rear+1) % self.limit == self.front:
            print('full')
            return -1

        elif self.front == -1:
            self.front += 1
            self.rear += 1
            self.Q[0] = data

        else:
            self.rear  = (self.rear+1) % self.limit
            self.Q[self.rear] = data

    def delete_queue(self):
        if self.front == -1:
            # print("empty")
            return None

        elif self.front == self.rear:
            a = self.front
            self.front= -1
            self.rear = -1
            return self.Q[a]

        else:
            b = self.front
            self.front = (self.front+1) % self.limit
            return self.Q[b]

    def display(self):
        if self.rear == -1 and self.front == -1:
            print("empty")
            return -1
        elif self.front <= self.rear:
            for i in range(self.front, self.rear+1):
                print(self.Q[i])
        else:
            for i in range(self.front, self.limit):
                print(self.Q[i])

            for i in range(0, self.rear + 1):
                print(self.Q[i])

    def replace(self, data, new_data):
```

```
        flag = False
        if self.rear == -1 and self.front == -1:
            print("empty")
        else:
            if self.front< self.rear:
                for i in range(self.front, self.rear + 1):
                    if self.Q[i] == data:
                        self.Q[i] =  new_data
                        flag = True
            elif self.front == self.rear:
                if self.Q[self.front] == data:
                    self.Q[self.front] = new_data
                    flag = True
            elif self.front > self.rear:
                for i in range(self.front, self.limit):
                    if self.Q[i] == data:
                        self.Q[i] = new_data
                        flag = True
                for i in range(0, self.rear+1):
                    if self.Q[i] == data:
                        self.Q[i] =  new_data
                        flag = True
        if not(flag):
            print("Your data not found")

def reverse_data(Q:Cqueue):
    new_Q = Cqueue()
    stack = Stack()

    while Q.front != -1:
        stack.push(Q.delete_queue())

    while not(stack.is_empty()):
        new_Q.insert_queue(stack.pop())
    new_Q.display()

##########################################################################################
##############################################################

# محسبه فاکتوریل #

      توابع بازگشتی
def calc_fac(n):
    pass
    if n == 1:
        return 1
    else:
        return n * calc_fac(n-1)


def fibo(n):
    if n < 2:
        return n

    else:
```

```
        return fibo(n-1) + fibo(n-2)


def sum_num(a, b):
    if b == 0:
        return a
    else:
        return sum_num(a, b-1) + 1


def multi_num(a, b):
    if b == 0:
        return 0
    else:
        return multi_num(a, b-1) + a


def divi(a, b):
    if a < b:
        return 0
    else:
        return divi(a-b, b) + 1


# Hannoi Tower
def Hannoi(n, A, B, C):
    if n == (1):
        print(A, 'to', C)
    else:
        Hannoi(n-1, A, C, B)
        print(A, 'to', C)
        Hannoi(n-1, B, A, C)

##########################################################################################
################################################################

# لیست پیوندی

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

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

    def insert_first(self, x):
        a = Node(x)
        a.next = self.head
        self.head = a

    def insert_last(self, x):
        a = Node(x)
        if self.head == None:
```

```
            self.head = a
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = a

    def insert_after(self, x, data):  # insert after x
        if self.head is None:
            return
        elif self.head.next is None:
            if self.head.data == x:
                a = Node(data)
                self.head.next = a
            else:
                return None
        temp = self.head
        a = Node(data)
        while temp.data != x:
            if temp.next is None:
                print("not found")
                return
            temp = temp.next
        a.next = temp.next
        temp.next = a

    def delete_first(self):
        if self.head is None:
            print("empty")
            return
        self.head = self.head.next

    def delete_last(self):
        if self.head is None:
            print("empty")
            return
        elif self.head.next is None:
            self.head = None
            return

        temp = self.head
        while temp.next.next != None:
            temp = temp.next
        temp.next = None

    def delete_after(self, x):
        if self.head is None:  # check if empty
            print("empty")
            return
        elif not (self.head.next):  # check if only one object
            return
        temp = self.head
        while temp.data != x:
            if temp.next is None:  # check if x is not available
                return
```

```
            temp = temp.next
        if temp.next:  # check if x is the last object
            temp.next = temp.next.next

    def delete(self, x):
        if self.head is None:  # check if empty
            print("empty")
            return

        elif self.head.data == x:
            self.head = self.head.next
            print("done")
            return None

        elif self.head.next is None:
            return

        temp = self.head
        while temp.next.data != x:
            if temp.next.next is None:
                return
            temp = temp.next
        temp.next = temp.next.next

# add replace method to class
    def replace(self, old_data, new_data):
        if self.head is None:
            print("empty")
            return

        elif self.head.data == old_data:
            self.head.data = new_data

        temp = self.head
        while temp.data != old_data:
            if temp.next is None:
                print("your data not found!")
                return
            temp = temp.next
        temp.data = new_data

##########################################################################################
###############################################################

# لیست پیوندی حلقوی

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

class CLinkedList:
    def __init__(self):
        self.head = None
```

```
def insert_first(self, data):
    a = Node(data)
    if self.head is None:
        self.head = a
        a.next = self.head
        return

    a.next = self.head
    temp = self.head
    while temp.next != self.head:
        temp = temp.next
    temp.next = a
    self.head = a

def insert_last(self, data):
    a = Node(data)

    if self.head is None:
        self.head = a
        a.next = self.head
        return

    temp = self.head
    while temp.next != self.head:
        temp = temp.next
    temp.next = a
    a.next = self.head

def insert_after(self, x, data):
    if self.head is None:
        print("empty")
        return

    temp = self.head
    while temp.data != x:
        if temp.next == self.head:
            print("your data not found")
            return
        temp = temp.next

    a = Node(data)
    a.next = temp.next
    temp.next = a

def delete_first(self):
    if self.head is None:
        print("empty")
        return None

    elif self.head.next == self.head:
        del (self.head)
        self.head = None
        return

    temp = self.head
```

```
        while temp.next != self.head:
            temp = temp.next

        temp.next = temp.next.next
        del(self.head)
        self.head = temp.next

    def delete_last(self):
        if self.head is None:
            print("empty")
            return None

        elif self.head.next == self.head:
            del(self.head)
            self.head = None
            return None

        temp = self.head
        while temp.next.next != self.head:
            temp = temp.next
        del(temp.next)
        temp.next = self.head

    def delete_after(self, x):
        if self.head is None:
            print("empty")
            return None

        temp = self.head
        while temp.data != x:
            if temp.next == self.head:
                print("your data not found")
                return
            temp = temp.next
        t = temp.next
        temp.next = t.next
        del(t)

    def delete(self, x):
        if self.head is None:
            print("empty")
            return None

        elif self.head.next == self.head:
            if self.head.data == x:
                del(self.head)
                self.head = None
                return
            else:
                print("your data not found")
                return

        temp = self.head
        while temp.next.data != x:
            if temp.next == self.head:
```

```
                print("your data not found")
                return None
            temp = temp.next

        t = temp.next
        temp.next = t.next
        del(t)

#  write replace method
    def replace(self, old_data, new_data):
        if self.head is None:
            print("empty")
            return None
        temp = self.head
        while temp.data != old_data:
            if temp.next == self.head:
                print("your data not found")
                return None
            temp = temp.next
        temp.data = new_data

###########################################################################################
#################################################################

# لیست پیوندی دو طرفه

class dNode:
    def __init__(self, data):
        self.back = None
        self.data = data
        self.next = None

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

    def insert_first(self, data):
        a = dNode(data)
        if self.head is None:
            self.head = a
            return

        a.next = self.head
        a.next.back = a
        self.head = a

    def insert_last(self, data):
        a = dNode(data)
        if self.head is None:
            self.head = a
            return

        temp = self.head
        while temp.next != None:
            temp = temp.next
```

```
        temp.next = a
        a.back = temp

    def insert_after(self, x, data):
        if self.head is None:
            print("empty")
            return None

        temp = self.head
        while temp.data != x:
            if temp.next is None:
                print("your data not found")
                return None
            temp = temp.next

        a = dNode(data)
        if temp.next:
            a.next = temp.next
            temp.next = a
            a.back = temp
            a.next.back = a
        else:
            temp.next = a
            a.back = temp

    def delete_first(self):
        if self.head is None:
            print("empty")
            return None

        elif self.head.next is None:
            del(self.head)
            self.head = None
            return None

        temp = self.head
        self.head = self.head.next
        self.head.back = None
        del(temp)

    def delete_last(self):
        if self.head is None:
            print("empty")
            return None

        temp = self.head
        while temp.next != None:
            temp = temp.next

        temp.back.next = None
        del(temp)

    def delete_after(self, x):
        if self.head is None:
            print("empty")
```

```
        return None

    elif self.head.next is None:
        return

    temp = self.head
    while temp.data != x:
        if temp.next is None:
            print("your data not found")
            return None
        temp = temp.next

    temp.next = temp.next.next
    a = temp.next.back
    temp.next.back = temp
    del(a)

def delete(self, x):
    if self.head is None:
        print("empty")
        return None

    elif self.head.data == x:
        a = self.head
        self.head = a.next
        self.head.back = None
        del(a)
        return None

    temp = self.head
    while temp.data != x:
        if temp.next is None:
            print("your data not found")
            return None
        temp = temp.next

    if temp.next is None:
        temp.back.next = None
        del(temp)
    else:
        temp.back.next = temp.next
        temp.next.back = temp.back
        del(temp)
def show(self):
    if self.head is None:
        print("empty")
        return None
    elif self.head.next == self.head:
        print(self.head.data)
        return
    else:
        temp = self.head
        while temp.next != self.head:
            print(temp.data)
            temp = temp.next
```

```
###########################################################################################
################################################################
```

```
# لیست پیوندی دو طرفه حلقوی

class dNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.back = None

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

    def insert_first(self, data):
        a = dNode(data)
        if self.head is None:
            self.head = a
            self.head.next = self.head
            self.head.back = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            a.next = temp.next
            temp.next.back = a
            temp.next = a
            a.back = temp
            self.head = a

    def insert_last(self, data):
        a = dNode(data)
        if self.head is None:
            self.head = a
            self.head.next = self.head
            self.head.back = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            a.next = self.head
            self.head.back = a
            temp.next = a
            a.back = temp

    def insert_after(self, x, data):
        if self.head is None:
            print("empty")
            return None
        else:
            temp = self.head
            while temp.data != x:
                if temp.next == self.head:
```

```
            print("your data not found")
            return None
        else:
            temp = temp.next
    a = dNode(data)
    a.next = temp.next
    temp.next.back = a
    temp.next = a
    a.back = temp

def delete_first(self):
    if self.head is None:
        print("empty")
        return None
    elif self.head.next == self.head:
        del(self.head)
        self.head = None
        return None
    else:
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = temp.next.next
        temp.next.back = temp
        del(self.head)
        self.head = temp.next


def delete_last(self):
    if self.head is None:
        print("empty")
        return None
    elif self.head.next == self.head:
        del(self.head)
        self.head = None
        return
    else:
        temp = self.head

        while temp.next.next != self.head:
            temp = temp.next
        del(temp.next)
        temp.next = self.head
        self.head.back = temp


def delete_after(self, x):
    if self.head is None:
        print("empty")
        return None
    temp = self.head
    while temp.data != x:
        if temp.next == self.head:
            print("your data not found")
            return
```

```
            temp = temp.next
        t = temp.next
        temp.next = t.next
        t.next.back = temp
        del(t)

    def delete(self, x):
        if self.head is None:
            print("empty")
            return None

        elif self.head.next == self.head:
            if self.head.data == x:
                del(self.head)
                self.head = None
                return
        else:
            temp = self.head
            while temp.next.data != x:
                if temp.next == self.head:
                    print("your data not found")
                    return None
                temp = temp.next
            t = temp.next
            temp.next = t.next
            t.next.back = temp
            del(t)

    def show(self):
        if self.head is None:
            print("empty")
            return None
        else:
            temp = self.head
            while temp.next != self.head:
                print(temp.daat)
                temp = temp.next

############################################################################################
##################################################################

# کلاس درخت باینری

class BinaryNode:
    def __init__(self, data):
        self.data = data
        self.Lchild = None
        self.Rchild = None

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert_left(self, data):
        a = BinaryNode(data)
```

```
        if self.root is None:
            self.root = a
        else:
            temp = self.root
            while temp.Lchild != None:
                temp = temp.Lchild
            temp.Lchild = a

    def insert_right(self, data):
        a = BinaryNode(data)
        if self.root is None:
            self.root = a
        else:
            temp = self.root
            while temp.Rchild != None:
                temp = temp.Rchild
            temp.Rchild = a

    def preorder(self):
        self.ppreorder(self.root)

    def ppreorder(self, node):
        if node:
            print(node.data, end=" ")
            self.ppreorder(node.Lchild)
            self.ppreorder(node.Rchild)

    def inorder(self):
        self.pinorder(self.root)

    def pinorder(self, node):
        if node:
            self.pinorder(node.Lchild)
            print(node.data, end=" ")
            self.pinorder(node.Rchild)

    def postorder(self):
        self.ppostorder(self.root)

    def ppostorder(self, node):
        if node:
            self.ppostorder(node.Lchild)
            self.ppostorder(node.Rchild)
            print(node.data, end=" ")

    def level_order(self):
        if self.root is None:
            return
        l = list()
        l.append(self.root)
        while l:
            t = l.pop(0)
            print(t.data, end=" ")
            if t.Lchild:
                l.append(t.Lchild)
```

```
        if t.Rchild:
            l.append(t.Rchild)

def insert_after_l(self, x, data):
    self.pinsert_after_l(self.root, x, data)

def pinsert_after_l(self, node, x, data):
    if node:
        if node.data == x:
            temp = node.Lchild
            node.Lchild = BinaryNode(data)
            node.Lchild.Lchild = temp
        self.pinsert_after_l(node.Lchild, x, data)
        self.pinsert_after_l(node.Rchild, x, data)

def insert_after_r(self, x, data):
    self.pinsert_after_r(self.root, x, data)

def pinsert_after_r(self, node, x, data):
    if node:
        if node.data == x:
            temp = node.Rchild
            node.Rchild = BinaryNode(data)
            node.Rchild.Rchild = temp
        self.pinsert_after_r(node.Lchild, x, data)
        self.pinsert_after_r(node.Rchild, x, data)

def delete_left(self):
    if self.root is None:
        print('empty')
        return None
    else:
        temp = self.root
        while temp.Lchild.Lchild != None:
            temp = temp.Lchild
        del(temp.Lchild)
        temp.Lchild = None

def delete_right(self):
    if self.root is None:
        print("empty")
        return None
    else:
        temp = self.root
        while temp.Rchild.Rchild != None:
            temp = temp.Rchild
        del(temp.Rchild)
        temp.Rchild = None

def delete_x(self, x):
    if self.root is None:
        print('empty')
        return None
    else:
        self.pdelete(self.root, x)
```

```
def pdelete(self, node, x):
    if node != None:
        if node.Lchild:
            if node.Lchild.data == x:
                del(node.Lchild)
                node.Lchild = None
                return None
            self.pdelete(node.Lchild, x)
            self.pdelete(node.Rchild, x)
        if node.Rchild:
            if node.Rchild.data == x:
                del(node.Rchild)
                node.Rchild = None
                return None
            self.pdelete(node.Lchild, x)
            self.pdelete(node.Rchild, x)
        if node.data == x:
            node = None
            return

def delete_left_x(self, x):
    if self.root is None:
        print('empty')
        return
    else:
        self.pdelete_left_x(self.root, x)

def pdelete_left_x(self, node, x):
    if node != None:
        if node.Lchild:
            if node.Lchild.data == x:
                if node.Lchild.Lchild != None:
                    del(node.Lchild.Lchild)
                    node.Lchild.Lchild = None
                    return
            self.pdelete_left_x(node.Lchild, x)
            self.pdelete_left_x(node.Rchild, x)
        if node.Rchild:
            if node.Rchild.data == x:
                if node.Rchild.Lchild != None:
                    del(node.Rchild.Lchild)
                    node.Rchild.Lchild = None
                    return
            self.pdelete_left_x(node.Lchild, x)
            self.pdelete_left_x(node.Rchild, x)
        if node.data == x:
            if node.Lchild != None:
                node.Lchild = None
                return

def delete_right_x(self, x):
    if self.root is None:
        print("empty")
    else:
```

```
            return self.pdelete_right_x(self.root, x)

    def pdelete_right_x(self, node, x):
        if node != None:
            if node.Lchild:
                if node.Lchild.data == x:
                    if node.Lchild.Rchild != None:
                        del(node.Lchild.Rchild)
                        node.Lchild.Rchild = None
                        return
                self.pdelete_right_x(node.Lchild, x)
                self.pdelete_right_x(node.Rchild, x)
            if node.Rchild:
                if node.Rchild.data == x:
                    if node.Rchild.Rchild != None:
                        del(node.Rchild.Rchild)
                        node.Rchild.Rchild = None
                        return
                self.pdelete_right_x(node.Lchild, x)
                self.pdelete_right_x(node.Rchild, x)
            if node.data == x:
                if node.Rchild != None:
                    node.Rchild = None
                    return

####################################################################################
############################################################

# کلاس جستجوی درخت باینری

class BinaryNode:
    def __init__(self, data):
        self.data = data
        self.Lchild = None
        self.Rchild = None

class BST:
    def __init__(self):
        self.root = None
        self.l = list()

    def insert(self, data):
        self.l.append(data)
        if self.root is None:
            self.root = BinaryNode(data)
        else:
            return self.recins(self.root, data)

    def recins(self, root, data):
        if data > root.data:
            if root.Rchild is None:
                root.Rchild  = BinaryNode(data)
            else:
                self.recins(root.Rchild, data)
        else:
```

```
            if root.Lchild is None:
                root.Lchild = BinaryNode(data)
            else:
                self.recins(root.Lchild, data)

    def show_list(self):
        print(self.l)
        return(self.l)

    def search(self, data):
        return self.inorder(self.root, data)

    def inorder(self, root, data):
        if root is None:
            return -1
        elif root.data == data:
            return root
        else:
            if data > root.data:
                return self.inorder(root.Rchild)
            return self.inorder(root.Lchild)

    def display(self):
        return self.postorder(self, self.root)

    def postorder(self, root):
        if root is None:
            return None
        else:
            self.postorder(root.Lchild)
            self.postorder(root.Rchild)
            print(root.data, end=' ')

####################################################################################################
######################################################################

# Max Heap کلاس

class MaxHeap:
    def __init__(self):
        self.heap = []  # برای ذخیره عناصر

    def parent(self, i):
        return (i - 1) // 2

    def left(self, i):
        return 2 * i + 1

    def right(self, i):
        return 2 * i + 2

    def heapify_down_up(self, index):
        """Restore the heap property by moving the element at `index` upwards."""
        parent = self.parent(index)
        while index > 0 and self.heap[index] > self.heap[parent]:
```

```
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            index = parent
            parent = self.parent(index)

    def heapify_up_down(self, index):
        """Restore the heap property by moving the element at `index` downwards."""
        n = len(self.heap)
        largest = index
        left = self.left(index)
        right = self.right(index)
        if left < n and self.heap[left] > self.heap[largest]:
            largest = left
        if right < n and self.heap[right] > self.heap[largest]:
            largest = right
        if largest != index:
            self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]
            self.heapify_up_down(largest)

    def insert(self, key):
        self.heap.append(key)
        self.heapify_down_up(len(self.heap) - 1)

    def delete_root(self):
        if len(self.heap) != 0:
            root = self.heap[0]
            self.heap[0] = self.heap[-1]
            self.heap.pop()
            if self.heap:
                self.heapify_up_down(0)
            return root

    def delete(self, x):
        if x in self.heap:
            index = self.heap.index(x)
            removed_element = self.heap[index]
            self.heap[index] = self.heap[-1]
            self.heap.pop()

            if index < len(self.heap):
                self.heapify_up_down(index)
                self.heapify_down_up(index)

            return removed_element

    def display(self):
        print(self.heap)

# مرتب سازی
def heap_sort(elements):
    heap = MaxHeap()

    for element in elements:
        heap.insert(element)

    sorted_list = []
```

```
    while heap.heap:
        sorted_list.append(heap.delete_root())

    return sorted_list[::-1]

####################################################################################################
########################################################################

# Min Heap کلاس

class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, key):
        self.heap.append(key)
        self.heapify_up(len(self.heap) - 1)

    def heapify_up(self, index):
        parent = (index - 1) // 2
        while index > 0 and self.heap[index] < self.heap[parent]:
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            index = parent
            parent = (index - 1) // 2

# تبدیل کردن
def maxheap_to_minheap(max_heap):
    min_heap = MinHeap()
    for element in max_heap.heap:
        min_heap.insert(element)
    return min_heap.heap

def make_max_heap(list):
    result = MaxHeap()
    for i in list:
        result.insert(i)
    return result.heap

####################################################################################################
########################################################################

# کلاس گراف

class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_matrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        if 0 <= u < self.vertices and 0 <= v < self.vertices:
            self.adj_matrix[u][v] = 1
            self.adj_matrix[v][u] = 1
        else:
            print("Vertex index out of bounds")
            return None
```

```
    def bfs(self, start):
        if (0 <= start < self.vertices):
            visited = [False] * self.vertices
            queue = Cqueue()
            queue.insert_queue(start)
            bfs_order = []
            visited[start] = True
            while queue.front!=-1:
                current = queue.delete_queue()
                bfs_order.append(current)
                for neighbor in range(self.vertices):
                    if self.adj_matrix[current][neighbor] == 1 and not visited[neighbor]:
                        visited[neighbor] = True
                        queue.insert_queue(neighbor)
            print(f"BFS starting from vertex {start}: ", end='')
            return bfs_order

    def dfs(self, start):
        if (0 <= start < self.vertices):
            visited = [False] * self.vertices
            stack = Stack()
            stack.push(start)
            dfs_order = []
            while not (stack.is_empty()):
                current = stack.pop()
                if not visited[current]:
                    visited[current] = True
                    dfs_order.append(current)
                    # Push neighbors to the stack in reverse order to maintain order
                    for neighbor in range(self.vertices - 1, -1, -1):
                        if self.adj_matrix[current][neighbor] == 1 and not visited[neighbor]:
                            stack.push(neighbor)
            print(f"DFS starting from vertex {start}: ", end='')
            return dfs_order

    def display(self):
        print("Adjacency matrix: ")
        for row in self.adj_matrix:
            print(" ".join(map(str, row)))

#########################################################################################################################################################

# کلاس گراف جهت دار

class DGraph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_matrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        if 0 <= u < self.vertices and 0 <= v < self.vertices:
            self.adj_matrix[u][v] = 1
        else:
```

```
            raise ValueError("Vertex index out of bounds")

    def remove_edge(self, u, v):
        if 0 <= u < self.vertices and 0 <= v < self.vertices:
            self.adj_matrix[u][v] = 0
        else:
            raise ValueError("Vertex index out of bounds")

    def bfs(self, start):
        if not (0 <= start < self.vertices):
            raise ValueError("Start vertex index out of bounds")
        visited = [False] * self.vertices
        queue = [start]
        bfs_order = []
        visited[start] = True
        while queue:
            current = queue.pop(0)
            bfs_order.append(current)
            for neighbor in range(self.vertices):
                if self.adj_matrix[current][neighbor] != 0 and not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)
        return bfs_order

    def dfs(self, start):
        if not (0 <= start < self.vertices):
            raise ValueError("Start vertex index out of bounds")
        visited = [False] * self.vertices
        stack = [start]
        dfs_order = []

        while stack:
            current = stack.pop()
            if not visited[current]:
                visited[current] = True
                dfs_order.append(current)
                # Push neighbors to the stack in reverse order to maintain order
                for neighbor in range(self.vertices - 1, -1, -1):
                    if self.adj_matrix[current][neighbor] != 0 and not visited[neighbor]:
                        stack.append(neighbor)
        return dfs_order

    def display(self):
        for row in self.adj_matrix:
            print(" ".join(map(str, row)))

##########################################################################################
################################################################

# کلاس گراف وزن دار

class WGraph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0] * vertices for _ in range(vertices)]
```

```python
    def add_edge(self, u, v, weight):
        self.graph[u][v] = weight
        self.graph[v][u] = weight  # For undirected graph

    def print_graph(self):
        print("Adjacency matrix: ")
        for row in self.graph:
            for column in row:
                print(f"{column:^3}", end=' ')
            print()

#########################################################################################
###############################################################

# کلاس گراف جهت دار و وزن دار

class WDGraph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adjacency_matrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, s, t, weight):
        if 0 <= s < self.vertices and 0 <= t < self.vertices:
            self.adjacency_matrix[s][t] = weight
        else:
            print("Vertex index out of bounds.")
            return None

    def remove_edge(self, s, t):
        if 0 <= s < self.vertices and 0 <= t < self.vertices:
            self.adjacency_matrix[s][t] = 0
        else:
            print("Vertex index out of bounds.")
            return None

    def get_weight(self, s, t):
        if 0 <= s < self.vertices and 0 <= t < self.vertices:
            return self.adjacency_matrix[s][t]
        else:
            print("Vertex index out of bounds.")
            return None

    def display_graph(self):
        print("Adjacency matrix: ")
        for row in self.adjacency_matrix:
            for column in row:
                print(f"{column:^3}", end=' ')
            print()

#########################################################################################
###############################################################

# کلاس گراف جهت دار با لیست های مجاورت حلقوی
```

```
class Node:
    def __init__(self, vertex):
        self.vertex = vertex  # Destination vertex of the edge
        self.next = None  # Pointer to the next node

class Vertex:
    def __init__(self, vertex):
        self.vertex = vertex  # The vertex id
        self.adj_list = None  # Circular linked list of outgoing edges

class DirectedGraph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.vertices = [Vertex(i) for i in range(num_vertices)]  # Create vertices

    def add_edge(self, from_vertex, to_vertex):
        if 0 <= from_vertex < self.num_vertices and 0 <= to_vertex < self.num_vertices:
            new_node = Node(to_vertex)

            # Get the adjacency list of the from_vertex
            from_vertex_node = self.vertices[from_vertex]

            # If adjacency list is empty, set the node to point to itself (circular)
            if not from_vertex_node.adj_list:
                from_vertex_node.adj_list = new_node
                new_node.next = new_node  # Circular link to itself
            else:
                # Otherwise, add the new node to the circular linked list
                current = from_vertex_node.adj_list
                while current.next != from_vertex_node.adj_list:
                    current = current.next
                current.next = new_node
                new_node.next = from_vertex_node.adj_list
        else:
            print("Vertex index out of bounds.")

    def remove_edge(self, from_vertex, to_vertex):
        """Remove the directed edge from `from_vertex` to `to_vertex`."""
        if 0 <= from_vertex < self.num_vertices and 0 <= to_vertex < self.num_vertices:
            from_vertex_node = self.vertices[from_vertex]

            if not from_vertex_node.adj_list:
                print(f"No edges from vertex {from_vertex}.")
                return
            current = from_vertex_node.adj_list
            previous = None
            while True:
                if current.vertex == to_vertex:
                    if previous:
                        previous.next = current.next
                    else:
                        # Special case: removing the first node
                        if current.next == from_vertex_node.adj_list:
                            from_vertex_node.adj_list = None  # No more edges
                        else:
```

```
                    # Find the last node to update the circular link
                    temp = from_vertex_node.adj_list
                    while temp.next != from_vertex_node.adj_list:
                        temp = temp.next
                    temp.next = current.next
                    from_vertex_node.adj_list = current.next
                print(f"Edge from {from_vertex} to {to_vertex} removed.")
                return
            previous = current
            current = current.next
            if current == from_vertex_node.adj_list:
                break
        print(f"Edge from {from_vertex} to {to_vertex} not found.")
    else:
        print("Vertex index out of bounds.")

  def display_graph(self):
    for vertex in self.vertices:
      print(f"Vertex {vertex.vertex}:", end=" ")
      if vertex.adj_list:
        current = vertex.adj_list
        while True:
          print(f"{current.vertex}", end=" -> ")
          current = current.next
          if current == vertex.adj_list:
            break
        print("Back to start")
      else:
        print("No edges.")

##########################################################################################
################################################################

# کلاس گراف جهت دار و وزن دار با لیست های پیوندی حلقوی

class Node:
  def __init__(self, vertex, weight=0):
    self.vertex = vertex  # Destination vertex
    self.weight = weight  # Edge weight
    self.next = None

class Vertex:
  def __init__(self, vertex):
    self.vertex = vertex  # The vertex id
    self.adj_list = None  # Circular linked list of edges

class WeightedDirectedGraph:
  def __init__(self, num_vertices):
    self.num_vertices = num_vertices
    self.vertices = [Vertex(i) for i in range(num_vertices)]

  def add_edge(self, from_vertex, to_vertex, weight):
    if 0 <= from_vertex < self.num_vertices and 0 <= to_vertex < self.num_vertices:
      new_node = Node(to_vertex, weight)
```

```
            # Add the new node to the adjacency list of the from_vertex
            from_vertex_node = self.vertices[from_vertex]

            # If adjacency list is empty, set the node to point to itself (circular)
            if not from_vertex_node.adj_list:
                from_vertex_node.adj_list = new_node
                new_node.next = new_node  # Circular link to itself
            else:
                # Otherwise, add the node to the circular list at the end
                current = from_vertex_node.adj_list
                while current.next != from_vertex_node.adj_list:
                    current = current.next
                current.next = new_node
                new_node.next = from_vertex_node.adj_list
        else:
            print("Vertex index out of bounds.")

    def remove_edge(self, from_vertex, to_vertex):
        if 0 <= from_vertex < self.num_vertices and 0 <= to_vertex < self.num_vertices:
            from_vertex_node = self.vertices[from_vertex]

            if not from_vertex_node.adj_list:
                print(f"No edges from vertex {from_vertex}.")
                return
            current = from_vertex_node.adj_list
            previous = None

            # Traverse the circular linked list to find the edge
            while True:
                if current.vertex == to_vertex:
                    if previous:
                        previous.next = current.next
                    else:
                        # Special case: removing the first node
                        if current.next == from_vertex_node.adj_list:
                            from_vertex_node.adj_list = None  # No more edges
                        else:
                            # Find the last node to update the circular link
                            temp = from_vertex_node.adj_list
                            while temp.next != from_vertex_node.adj_list:
                                temp = temp.next
                            temp.next = current.next
                            from_vertex_node.adj_list = current.next
                    print(f"Edge from {from_vertex} to {to_vertex} removed.")
                    return
                previous = current
                current = current.next
                if current == from_vertex_node.adj_list:
                    break
            print(f"Edge from {from_vertex} to {to_vertex} not found.")
        else:
            print("Vertex index out of bounds.")

    def get_weight(self, from_vertex, to_vertex):
        if 0 <= from_vertex < self.num_vertices and 0 <= to_vertex < self.num_vertices:
```

```
            from_vertex_node = self.vertices[from_vertex]
            current = from_vertex_node.adj_list

            # Traverse the circular linked list to find the weight
            while current:
                if current.vertex == to_vertex:
                    return current.weight
                current = current.next
                if current == from_vertex_node.adj_list:
                    break
            print(f"Edge from {from_vertex} to {to_vertex} not found.")
            return None
        else:
            print("Vertex index out of bounds.")
            return None

    def display_graph(self):
        # Display each vertex and its adjacency list
        for vertex in self.vertices:
            print(f"Vertex {vertex.vertex}:", end=" ")
            if vertex.adj_list:
                current = vertex.adj_list
                while True:
                    print(f"({current.vertex}, {current.weight})", end=" -> ")
                    current = current.next
                    if current == vertex.adj_list:
                        break
                print("Back to start")
            else:
                print("No edges.")

##########################################################################################
################################################################
```