**SSN College of Engineering**

Department of Information Technology

UIT2201 — Programming and Data Structures

2022 – 2023

**Exercise — 09**

U. Pranaav | IT-B | 3122225002093

1)

I. AIM:

To design and implement stack and queue data structure as wrapper around the Python List and using stack and queue to check whether a given number is a palindrome.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides two classes that implement stacks and

queues in python and use the two data structures to find whether

a given number is a palindrome or not. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 31 2023

Revised on Wed May 31 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

*class* QueueWrapper:

    '''

    The QueueWrapper class is a wrapper class that provides a simple implementation

    of a queue data structure.

    Attributes:

        queue (list): A list representing the queue elements.

        top (int): The index representing the top element of the queue.

    Methods:

        enqueue(ele): Adds an element to the end of the queue.

        dequeue(): Removes and returns the element from the front of the queue.

        is\_empty(): Checks if the queue is empty.

        top\_ele(): Returns the top element of the queue without removing it.

        peek(): Returns the element from the front of the queue without removing it.

        \_\_str\_\_(): Returns a string representation of the queue.

        \_\_len\_\_(): Returns the number of elements in the queue.

    '''

*def* \_\_init\_\_(*self*):

        '''

        Initializes an empty QueueWrapper object.

        '''

*self*.queue = []

*self*.top = 0

*def* enqueue(*self*, *ele*):

        '''

        Adds an element to the end of the queue.

        Args:

            ele: The element to be added to the queue.

        '''

*self*.queue.append(*ele*)

*self*.top += 1

*def* dequeue(*self*):

        '''

        Removes and returns the element from the front of the queue.

        Returns:

            The element removed from the front of the queue.

        Raises:

            Exception: If the queue is empty.

        '''

        if *self*.is\_empty():

            raise Exception("Empty Queue")

        else:

            pop\_ele = *self*.queue.pop(0)

*self*.top -= 1

            return pop\_ele

*def* is\_empty(*self*):

        '''

        Checks if the queue is empty.

        Returns:

            True if the queue is empty, False otherwise.

        '''

        return len(*self*.queue) == 0

*def* top\_ele(*self*):

        '''

        Returns the top element of the queue without removing it.

        Returns:

            The top element of the queue.

        '''

        return *self*.queue[*self*.top - 1]

*def* peek(*self*):

        '''

        Returns the element from the front of the queue without removing it.

        Returns:

            The element from the front of the queue.

        '''

        return *self*.queue[0]

*def* \_\_str\_\_(*self*):

        '''

        Returns a string representation of the queue.

        Returns:

            A string representation of the queue.

        '''

        return str(*self*.queue)

*def* \_\_len\_\_(*self*):

        '''

        Returns the number of elements in the queue.

        Returns:

            The number of elements in the queue.

        '''

        return len(*self*.queue)

*class* StackWrapper:

    '''

    The StackWrapper class is a wrapper class that provides a simple

    implementation of a stack data structure.

    Attributes:

        stack (list): A list representing the stack elements.

        top (int): The index representing the top element of the stack.

    Methods:

        push(ele): Adds an element to the top of the stack.

        pop(): Removes and returns the element from the top of the stack.

        is\_empty(): Checks if the stack is empty.

        top\_ele(): Returns the top element of the stack without removing it.

        \_\_str\_\_(): Returns a string representation of the stack.

        \_\_len\_\_(): Returns the number of elements in the stack.

    '''

*def* \_\_init\_\_(*self*):

        '''

        Initializes an empty StackWrapper object.

        '''

*self*.stack = []

*self*.top = 0

*def* push(*self*, *ele*):

        '''

        Adds an element to the top of the stack.

        Args:

            ele: The element to be added to the stack.

        '''

*self*.stack.append(*ele*)

*self*.top += 1

*def* pop(*self*):

        '''

        Removes and returns the element from the top of the stack.

        Returns:

            The element removed from the top of the stack.

        Raises:

            Exception: If the stack is empty (underflow).

        '''

        if *self*.is\_empty():

            raise Exception("Stack Underflow")

        else:

            pop\_ele = *self*.stack.pop()

*self*.top -= 1

            return pop\_ele

*def* is\_empty(*self*):

        '''

        Checks if the stack is empty.

        Returns:

            True if the stack is empty, False otherwise.

        '''

        return len(*self*.stack) == 0

*def* top\_ele(*self*):

        '''

        Returns the top element of the stack without removing it.

        Returns:

            The top element of the stack.

        '''

        return *self*.stack[*self*.top - 1]

*def* \_\_str\_\_(*self*):

        '''

        Returns a string representation of the stack.

        Returns:

            A string representation of the stack.

        '''

        return str(*self*.stack)

*def* \_\_len\_\_(*self*):

        '''

        Returns the number of elements in the stack.

        Returns:

            The number of elements in the stack.

        '''

        return len(*self*.stack)

*def* digits(*data*):

    digits\_list = []

    while *data*:

        digits\_list.append(*data*%10)

*data* //= 10

    return digits\_list

*def* palindrome(*n*):

    digits\_list = digits(*n*)

    stack = StackWrapper()

    queue = QueueWrapper()

    for val in digits\_list:

        stack.push(val)

        queue.enqueue(val)

    while len(stack) != 0:

        if stack.pop() != queue.dequeue():

            return False

    return True

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    for n in range(10,1000):

        if palindrome(n):

            print(*f*"{n} is a palindrome")

        else:

            continue

III. OUTPUT:

11 is a palindrome

22 is a palindrome

33 is a palindrome

44 is a palindrome

55 is a palindrome

66 is a palindrome

77 is a palindrome

88 is a palindrome

99 is a palindrome

101 is a palindrome

111 is a palindrome

121 is a palindrome

131 is a palindrome

141 is a palindrome

151 is a palindrome

161 is a palindrome

171 is a palindrome

181 is a palindrome

191 is a palindrome

202 is a palindrome

212 is a palindrome

222 is a palindrome

232 is a palindrome

242 is a palindrome

252 is a palindrome

262 is a palindrome

272 is a palindrome

282 is a palindrome

292 is a palindrome

303 is a palindrome

313 is a palindrome

323 is a palindrome

333 is a palindrome

343 is a palindrome

353 is a palindrome

363 is a palindrome

373 is a palindrome

383 is a palindrome

393 is a palindrome

404 is a palindrome

414 is a palindrome

424 is a palindrome

434 is a palindrome

444 is a palindrome

454 is a palindrome

464 is a palindrome

474 is a palindrome

484 is a palindrome

494 is a palindrome

505 is a palindrome

515 is a palindrome

525 is a palindrome

535 is a palindrome

545 is a palindrome

555 is a palindrome

565 is a palindrome

575 is a palindrome

585 is a palindrome

595 is a palindrome

606 is a palindrome

616 is a palindrome

626 is a palindrome

636 is a palindrome

646 is a palindrome

656 is a palindrome

666 is a palindrome

676 is a palindrome

686 is a palindrome

696 is a palindrome

707 is a palindrome

717 is a palindrome

727 is a palindrome

737 is a palindrome

747 is a palindrome

757 is a palindrome

767 is a palindrome

777 is a palindrome

787 is a palindrome

797 is a palindrome

808 is a palindrome

818 is a palindrome

828 is a palindrome

838 is a palindrome

848 is a palindrome

858 is a palindrome

868 is a palindrome

878 is a palindrome

888 is a palindrome

898 is a palindrome

909 is a palindrome

919 is a palindrome

929 is a palindrome

939 is a palindrome

949 is a palindrome

959 is a palindrome

969 is a palindrome

979 is a palindrome

989 is a palindrome

999 is a palindrome

2)

I. AIM:

To design and implement data structure to maintain two stacks in a single array. All the basic stack operations include an argument to select one of the stacks. For example, ‘push(0, item)’ pushes item into the first stack, while ‘push(1, item)’ pushes item into the second stack.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class called dualstack that implements a

data structure to maintain two stacks in a single array. This

is a part of the exercises given under the course UIT2201

(Programming and Data Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 31 2023

Revised on Wed May 31 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

import ctypes

*class* dualstack:

    '''

    The dualstack class represents a data structure that combines two

    stacks in a single array.

    Attributes:

        top1 (int): The index representing the top element of the first stack.

        top2 (int): The index representing the top element of the second stack.

        array (ctypes array): The array used to store the elements of the stacks.

        n (int): The size of the array.

    Methods:

        makearray(size): Creates a new array of the specified size.

        push(ind, ele): Pushes an element onto the specified stack.

        pop(ind): Pops and returns the top element from the specified stack.

        \_\_setitem\_\_(ind, ele): Sets the element at the specified index in the array.

        \_\_str\_\_(): Returns a string representation of the dualstack.

        isfull(): Checks if the dualstack is full.

    '''

*def* \_\_init\_\_(*self*, *size*):

        '''

        Initializes a dualstack object with the specified size.

        Args:

            size (int): The size of the dualstack.

        '''

*self*.top1 = 0

*self*.top2 = *size* - 1

*self*.array = *self*.makearray(*size*)

*self*.n = *size*

*def* makearray(*self*, *size*):

        '''

        Creates a new array of the specified size.

        Args:

            size (int): The size of the new array.

        Returns:

            A new array of the specified size.

        '''

        B = (*size* \* ctypes.py\_object)()

        return B

*def* push(*self*, *ind*, *ele*):

        '''

        Pushes an element onto the specified stack.

        Args:

            ind (int): The index of the stack to push the element onto.

            ele: The element to be pushed.

        Raises:

            Exception: If the stack is full (overflow).

        '''

        if *ind* == 0:

            if not *self*.isfull():

*self*.array[*self*.top1] = *ele*

*self*.top1 += 1

            else:

                raise Exception("Stack overflow")

        else:

            if not *self*.isfull():

*self*.array[*self*.top2] = *ele*

*self*.top2 -= 1

            else:

                raise Exception("Stack overflow")

*def* pop(*self*, *ind*):

        '''

        Pops and returns the top element from the specified stack.

        Args:

            ind (int): The index of the stack to pop the element from.

        Returns:

            The element removed from the top of the stack.

        Raises:

            Exception: If the stack is empty (underflow).

        '''

        if *ind* == 0:

            try:

                if *self*.top1 >= 0:

                    pop\_ele = *self*.array[*self*.top1]

*self*.array[*self*.top1] = ctypes.py\_object()

*self*.top1 -= 1

                    return pop\_ele

                else:

                    raise Exception("Stack underflow")

            except:

*self*.top1 -= 1

                if *self*.top1 >= 0:

                    pass

                else:

                    raise Exception("Stack underflow")

                return None

        else:

            try:

                if *self*.top2 < *self*.n:

                    pop\_ele = *self*.array[*self*.top2]

*self*.array[*self*.top2] = ctypes.py\_object()

*self*.top2 += 1

                    return pop\_ele

                else:

                    raise Exception("Stack underflow")

            except:

*self*.top2 += 1

                if *self*.top2 < *self*.n:

                    pass

                else:

                    raise Exception("Stack underflow")

                return None

*def* \_\_setitem\_\_(*self*, *ind*, *ele*):

        '''

        Sets the element at the specified index in the array to the specified value.

        Args:

            ind (int): The index of the element to set.

            ele: The new value for the element at the specified index.

        Raises:

            IndexError: If the index is out of range.

        '''

        if *ind* <= *self*.n:

*self*.n += 1

*self*.array[*ind*] = *ele*

        else:

            raise IndexError("Index out of range")

*def* \_\_str\_\_(*self*):

        '''

        Returns a string representation of the dualstack.

        Returns:

            A string representation of the dualstack.

        '''

        out = '<'

        for i in range(*self*.n):

            try:

                if i != (*self*.n - 1):

                    out += str(*self*.array[i])

                    out += ','

                else:

                    out += str(*self*.array[i])

            except:

                out += '-,'

        out = out.rstrip(',')

        return out + '>'

*def* isfull(*self*):

        '''

        Checks if the dualstack is full.

        Returns:

            True if the dualstack is full, False otherwise.

        '''

        if *self*.top1 > *self*.top2:

            return True

        return False

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    dual\_stack = dualstack(10)

    for i in range(1,6):

        dual\_stack.push(0,i)

        print(dual\_stack)

    print(dual\_stack)

    print()

    for j in range(100,105):

        dual\_stack.push(1,j)

        print(dual\_stack)

    print(dual\_stack)

    print()

    #now checking if it is full

    print(*f*"Is the stack full : {dual\_stack.isfull()}")

    print()

    for i in range(5):

        dual\_stack.pop(0)

        print(dual\_stack)

    print(dual\_stack)

    print()

    for j in range(5):

        dual\_stack.pop(1)

        print(dual\_stack)

    print(dual\_stack)

    print()

    #now removing everything

    dual\_stack.pop(0)

    dual\_stack.pop(1)

    print(dual\_stack)

    print()

III. OUTPUT:

<1,-,-,-,-,-,-,-,-,->

<1,2,-,-,-,-,-,-,-,->

<1,2,3,-,-,-,-,-,-,->

<1,2,3,4,-,-,-,-,-,->

<1,2,3,4,5,-,-,-,-,->

<1,2,3,4,5,-,-,-,-,->

<1,2,3,4,5,-,-,-,-,100>

<1,2,3,4,5,-,-,-,101,100>

<1,2,3,4,5,-,-,102,101,100>

<1,2,3,4,5,-,103,102,101,100>

<1,2,3,4,5,104,103,102,101,100>

<1,2,3,4,5,104,103,102,101,100>

Is the stack full : True

<1,2,3,4,5,-,103,102,101,100>

<1,2,3,4,-,-,103,102,101,100>

<1,2,3,-,-,-,103,102,101,100>

<1,2,-,-,-,-,103,102,101,100>

<1,-,-,-,-,-,103,102,101,100>

<1,-,-,-,-,-,103,102,101,100>

<1,-,-,-,-,-,103,102,101,100>

<1,-,-,-,-,-,103,102,101,100>

<1,-,-,-,-,-,-,102,101,100>

<1,-,-,-,-,-,-,-,101,100>

<1,-,-,-,-,-,-,-,-,100>

<1,-,-,-,-,-,-,-,-,100>

<-,-,-,-,-,-,-,-,-,->

PART – B

3)

I. AIM:

To make a food delivery system accepts a maximum of M orders. Orders are served in first come first basis. Orders once placed cannot be cancelled.

II. CODE:

import sys

sys.path.append('D:\college files\DSA')

from QueueWrapper import QueueWrapper

import ctypes

# -\*- coding: utf-8 -\*-

'''

This module provides a class for implementation of the queue data

structure in python using wrapper method. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 31 2023

Revised on Wed May 31 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

*class* CircularQueue:

    '''

    The CircularQueue class is a wrapper class that provides a simple implementation

    of a circular queue data structure.

    Attributes:

        queue (list): A list representing the queue elements.

        rear (int): The index representing the rear element of the queue.

        front (int): The index representing the front element of the queue.

        cap (int): The maximum capacity of the queue.

    Methods:

        enqueue(ele): Adds an element to the end of the queue.

        dequeue(): Removes and returns the element from the front of the queue.

        is\_full(): Checks if the queue is full.

        is\_empty(): Checks if the queue is empty.

        rear\_element(): Returns the rear element of the queue without removing it.

        peek(): Returns the element from the front of the queue without removing it.

        \_\_contains\_\_(ele): Checks if the given element is present in the queue.

        \_\_getitem\_\_(ind): Gets the element at the specified index from the queue.

        \_\_setitem\_\_(ind, ele): Sets the element at the specified index in the queue to the specified value.

        \_\_str\_\_(): Returns a string representation of the queue.

        \_\_len\_\_(): Returns the number of elements in the queue.

    '''

*def* \_\_init\_\_(*self*, *cap*):

        '''

        Initializes an empty CircularQueue object.

        Parameters:

            cap (int): The maximum capacity of the queue.

        '''

*self*.queue = *self*.make\_queue(*cap*)

*self*.rear = *self*.front = 0

*self*.cap = *cap*

*def* make\_queue(*self*, *size*):

        '''

        Creates a new queue of the specified size.

        Parameters:

            size (int): The size of the new queue.

        Returns:

            A new queue of the specified size.

        '''

        B = (*size* \* ctypes.py\_object)()

        return B

*def* enqueue(*self*, *ele*):

        '''

        Adds an element to the end of the queue.

        Parameters:

            ele: The element to be added to the queue.

        '''

        if *self*.is\_full():

            raise Exception("Queue is full")

        else:

*self*.queue[*self*.rear] = *ele*

*self*.rear = *self*.next(*self*.rear)

*def* is\_full(*self*):

        '''

        Checks if the queue is full.

        Returns:

            True if the queue is full, False otherwise.

        '''

        return (*self*.rear + 1) % *self*.cap == *self*.front

*def* next(*self*, *val*):

        '''

        Returns the next index in the circular queue.

        Parameters:

            val (int): The current index.

        Returns:

            The next index in the circular queue.

        '''

*val* = (*val* + 1) % *self*.cap

        return *val*

*def* dequeue(*self*):

        '''

        Removes and returns the element from the front of the queue.

        Returns:

            The element removed from the front of the queue.

        Raises:

            Exception: If the queue is empty.

        '''

        if *self*.is\_empty():

            raise Exception("Empty Queue")

        else:

            pop\_ele = *self*.queue[*self*.front]

*self*.queue[*self*.front] = ctypes.py\_object()

*self*.front = *self*.next(*self*.front)

            return pop\_ele

*def* is\_empty(*self*):

        '''

        Checks if the queue is empty.

        Returns:

            True if the queue is empty, False otherwise.

        '''

        return *self*.rear == *self*.front

*def* rear\_element(*self*):

        '''

        Returns the rear element of the queue without removing it.

        Returns:

            The rear element of the queue.

        '''

        return *self*.queue[*self*.rear - 1]

*def* peek(*self*):

        '''

        Returns the element from the front of the queue without

        removing it.

        Returns:

            The element from the front of the queue.

        '''

        return *self*.queue[*self*.front]

*def* \_\_contains\_\_(*self*, *ele*):

        '''

        Check if the given element is present in the queue.

        Parameters:

            ele: The element to check for presence in the queue.

        Returns:

            bool: True if the element is found in the queue, False

            otherwise.

        '''

        try:

            for i in *self*.queue:

                if *ele* == i:

                    return True

        except:

            return False

        return False

*def* \_\_getitem\_\_(*self*, *ind*):

        '''

        Gets the element at the specified index from the queue.

        Parameters:

            ind (int): The index of the element to get.

        Returns:

            The element at the specified index from the queue.

        '''

        return *self*.queue[*ind*]

*def* \_\_setitem\_\_(*self*, *ind*, *ele*):

        '''

        Sets the element at the specified index in the queue to the specified value.

        Parameters:

            ind (int): The index of the element to set.

            ele: The new value for the element at the specified index.

        '''

        if *ind* <= *self*.n:

*self*.n += 1

*self*.queue[*ind*] = *ele*

        else:

            raise IndexError("Index out of range")

*def* \_\_str\_\_(*self*):

        '''

        Converts the queue to a string.

        Returns:

            A string representation of the queue.

        '''

        out = '<'

        for i in range(*self*.front, *self*.cap):

            try:

                if i != (*self*.cap - 1):

                    out += str(*self*.queue[i])

                    out += ','

                else:

                    out += str(*self*.queue[i])

            except:

                continue

        return out + '>'

*def* \_\_len\_\_(*self*):

        '''

        Returns the number of elements in the queue.

        Returns:

            The number of elements in the queue.

        '''

        return len(*self*.queue)

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    Order\_circular\_queue = CircularQueue(7)

    print("Checking whether queue is empty : ", Order\_circular\_queue.is\_empty())

    print()

    print("Now adding orders to queue while taking M to be 6")

    print()

    # Add orders to the queue

    Order\_circular\_queue.enqueue("Pizza")

    Order\_circular\_queue.enqueue("Burger")

    Order\_circular\_queue.enqueue("Dosa")

    Order\_circular\_queue.enqueue("Idly")

    Order\_circular\_queue.enqueue("Pasta")

    Order\_circular\_queue.enqueue("Lasagne") # Queue is full, cannot accept more orders

    print("Checking whether queue is full : ", Order\_circular\_queue.is\_full())

    print()

    # Serve orders from the queue

    print("Dequeueing orders : ")

    print()

    print(Order\_circular\_queue.dequeue())  # Pizza served

    print(Order\_circular\_queue.dequeue())  # Burger served

    print(Order\_circular\_queue.dequeue())  # Dosa served

    print(Order\_circular\_queue.dequeue())  # Idly served

    print(Order\_circular\_queue.dequeue())  # Pasta served

    print(Order\_circular\_queue.dequeue())  # Lasagne served

III. OUTPUT:

Checking whether queue is empty : True

Now adding orders to queue while taking M to be 6

Checking whether queue is full : True

Dequeueing orders :

Pizza

Burger

Dosa

Idly

Pasta

Lasagne

4)

I. AIM:

To design and implement data structure for a queue like abstraction, referred to as PQueue, that internally maintains two arrays — one for a high priority queue and the other for the low priority queue. Enqueue operation will mention the priority of the ‘item’ to be added to the PQueue — for example, ‘enqueue(0, item)’ will add the ‘item’ to the high priority queue, while ‘enqueue(1, item)’ will add it to the low priority queue. ‘dequeue()’ operation will dequeue from the high priority queue. If the high priority queue is empty, then first item from the low priority queue will be dequeued.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class called PQueue that implements a

data structure to maintain two queues in a single data structure

that has one high priority queue and another low priority queue.

This is a part of the exercises given under the course UIT2201

(Programming and Data Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 31 2023

Revised on Wed May 31 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

import sys

sys.path.append('D:\college files\DSA')

from QueueWrapper import QueueWrapper

*class* PQueue:

    '''

    The PQueue class represents a priority queue implemented using two

    separate queues: one for high priority elements and another for low

    priority elements.

    Attributes:

        highprio (QueueWrapper): A QueueWrapper instance representing

        the queue for high priority elements.

        lowprio (QueueWrapper): A QueueWrapper instance representing

        the queue for low priority elements.

        top1 (int): The index representing the top element of the high

        priority queue.

        bottom1 (int): The index representing the bottom element of the

        high priority queue.

        top2 (int): The index representing the top element of the low

        priority queue.

        bottom2 (int): The index representing the bottom element of the

        low priority queue.

    Methods:

        enqueue(val, ele): Adds an element to the end of the queue based on

        its priority level.

        dequeue(): Removes and returns the element from the front of the queue.

        is\_empty(val): Checks if the queue is empty based on its priority level.

        peek(val): Returns the element from the front of the queue without

        removing it based on its priority level.

        \_\_str\_\_(): Returns a string representation of the queue.

    '''

*def* \_\_init\_\_(*self*):

        '''

        Initializes a Priority Queue.

        Creates two QueueWrapper instances, one for high priority elements

        and another for low priority elements. Initializes the top and bottom

        indices for both queues.

        '''

*self*.highprio = QueueWrapper()

*self*.lowprio = QueueWrapper()

*self*.top1 = 0

*self*.bottom1 = 0

*self*.top2 = 0

*self*.bottom2 = 0

*def* enqueue(*self*, *val*, *ele*):

        '''

        Adds an element to the end of the queue.

        Args:

            ele: The element to be added to the queue.

        '''

        if *val* == 0:

*self*.highprio.enqueue(*ele*)

*self*.top1 += 1

        else:

*self*.lowprio.enqueue(*ele*)

*self*.top2 += 1

*def* dequeue(*self*):

        '''

        Removes and returns the element from the front of the queue.

        Returns:

            The element removed from the front of the queue.

        Raises:

            Exception: If the queue is empty.

        '''

        if *self*.is\_empty(0):

            if *self*.is\_empty(1):

                raise Exception("Empty Queue")

            else:

                pop\_ele = *self*.lowprio.dequeue()

*self*.bottom2 += 1

                return pop\_ele

        else:

            pop\_ele = *self*.highprio.dequeue()

*self*.bottom1 -= 1

            return pop\_ele

*def* is\_empty(*self*,*val*):

        '''

        Checks if the queue is empty.

        Returns:

            True if the queue is empty, False otherwise.

        '''

        if *val* == 0:

            return len(*self*.highprio) == 0

        else:

            return len(*self*.lowprio) == 0

*def* peek(*self*,*val*):

        '''

        Returns the element from the front of the queue without removing it.

        Returns:

            The element from the front of the queue.

        '''

        if *val* == 0:

            return *self*.highprio[0]

        else:

            return *self*.lowprio[0]

*def* \_\_str\_\_(*self*):

        '''

        Returns a string representation of the queue.

        Returns:

            A string representation of the queue.

        '''

        a = str(*self*.highprio)

        b = str(*self*.lowprio)

        return "High priority is:" + a + "\nLow priority is:" + b

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    priority\_queue = PQueue()

    for i in range(1,6):

        priority\_queue.enqueue(0,i)

    print(priority\_queue)

    print()

    for j in range(100,105):

        priority\_queue.enqueue(1,j)

    print(priority\_queue)

    print()

    print("Now dequeueing high priority elements:")

    print()

    for i in range(5):

        priority\_queue.dequeue()

        print(priority\_queue)

    print(priority\_queue)

    print()

    print("Now dequeueing the remaining elements:")

    print()

    for i in range(5):

        priority\_queue.dequeue()

        print(priority\_queue)

III. OUTPUT:

High priority is:[1, 2, 3, 4, 5]

Low priority is:[]

High priority is:[1, 2, 3, 4, 5]

Low priority is:[100, 101, 102, 103, 104]

Now dequeueing high priority elements:

High priority is:[2, 3, 4, 5]

Low priority is:[100, 101, 102, 103, 104]

High priority is:[3, 4, 5]

Low priority is:[100, 101, 102, 103, 104]

High priority is:[4, 5]

Low priority is:[100, 101, 102, 103, 104]

High priority is:[5]

Low priority is:[100, 101, 102, 103, 104]

High priority is:[]

Low priority is:[100, 101, 102, 103, 104]

High priority is:[]

Low priority is:[100, 101, 102, 103, 104]

Now dequeueing the remaining elements:

High priority is:[]

Low priority is:[101, 102, 103, 104]

High priority is:[]

Low priority is:[102, 103, 104]

High priority is:[]

Low priority is:[103, 104]

High priority is:[]

Low priority is:[104]

High priority is:[]

Low priority is:[]