**EXP 4.A**

class Queue:

def \_init\_(self, size):

self.queue = [None] \* size # Initialize the queue with None values

self.front = -1 # Front pointer

self.rear = -1 # Rear pointer

self.size = size # Max size of the queue

# Check if the queue is full

def isfull(self):

return self.rear == self.size - 1

# Check if the queue is empty

def isempty(self):

return self.front == -1 or self.front > self.rear

# Add an item to the queue

def enqueue(self, data):

if self.isfull():

print("Overflow error: Queue is full.")

return

if self.front == -1: # If the queue is empty, reset front pointer

self.front = 0

self.rear += 1

self.queue[self.rear] = data

print(f"Enqueued {data} to the queue.")

# Remove an item from the queue

def dequeue(self):

if self.isempty():

print("Underflow error: Queue is empty.")

return

data = self.queue[self.front]

self.front += 1

# If all elements are dequeued, reset the queue

if self.front > self.rear:

self.front = -1

self.rear = -1

print(f"Dequeued {data} from the queue.")

return data

# Print the queue

def display(self):

if self.isempty():

print("Queue is empty.")

else:

print("Queue elements:", self.queue[self.front:self.rear + 1])

# Example usage:

queue\_size = 5 # Define the queue size

q = Queue(queue\_size)

# Perform operations

q.enqueue(10)

q.enqueue(20)

q.enqueue(30)

q.display()

q.dequeue()

q.display()

q.enqueue(40)

q.enqueue(50)

q.enqueue(60) # This should show an overflow error

q.display()

q.dequeue()

q.display()

**EXP 4.B**

# Node structure

class Node:

def \_init\_(self, data):

self.data = data

self.next = None

# Queue implementation using linked list

class Queue:

def \_init\_(self):

self.front = None

self.rear = None

# Function to enqueue (add element to the queue)

def enQueue(self, value):

newNode = Node(value)

# If the queue is empty

if self.rear is None:

self.front = self.rear = newNode

print(f"{value} enqueued successfully.")

return

# If the queue is not empty

self.rear.next = newNode

self.rear = newNode

print(f"{value} enqueued successfully.")

# Function to dequeue (remove element from the queue)

def deQueue(self):

# If the queue is empty

if self.front is None:

print("Queue is Empty!!! Deletion is not possible!!!")

return

# If the queue is not empty

temp = self.front

self.front = temp.next

if self.front is None: # If front becomes None, set rear to None as well

self.rear = None

print(f"{temp.data} dequeued successfully.")

del temp

# Function to display the queue

def display(self):

# If the queue is empty

if self.front is None:

print("Queue is Empty!!!")

return

# If the queue is not empty

temp = self.front

print("Queue elements: ", end="")

while temp is not None:

print(f"{temp.data} --->", end=" ")

temp = temp.next

print("NULL")

# Main method to demonstrate the queue operations

if \_name\_ == "\_main\_":

q = Queue()

while True:

print("\nQueue Operations Menu:")

print("1. Enqueue")

print("2. Dequeue")

print("3. Display")

print("4. Exit")

choice = int(input("Enter your choice (1-4): "))

if choice == 1:

value = int(input("Enter value to enqueue: "))

q.enQueue(value)

elif choice == 2:

q.deQueue()

elif choice == 3:

q.display()

elif choice == 4:

print("Exiting the program.")

break

else:

print("Invalid choice! Please try again.")

**EXP 5**

def precedence(op):

if op == '+' or op == '-':

return 1

if op == '\*' or op == '/':

return 2

if op == '^':

return 3

return 0

# Function to check if the character is an operand

def is\_operand(c):

return c.isalpha() or c.isdigit()

# Function to convert infix expression to postfix

def infix\_to\_postfix(expression):

stack = [] # Initialize an empty stack

postfix = '' # To store the resulting postfix expression

# Scan the infix expression from left to right

for char in expression:

# If the character is an operand, add it to the postfix string

if is\_operand(char):

postfix += char

# If the character is an opening bracket, push it to the stack

elif char == '(':

stack.append(char)

# If the character is a closing bracket, pop and add to postfix until opening bracket is found

elif char == ')':

while stack and stack[-1] != '(':

postfix += stack.pop()

stack.pop() # Remove the '(' from the stack

# If the character is an operator

else:

while stack and precedence(stack[-1]) >= precedence(char):

postfix += stack.pop()

stack.append(char)

# Pop all the operators from the stack

while stack:

postfix += stack.pop()

return postfix

# Main program

if \_name\_ == "\_main\_":

# Example input

infix\_expr = input("Enter infix expression: ")

print("Postfix expression:", infix\_to\_postfix(infix\_expr))