DS2030 Data Structures and Algorithms for Data Science
Week 2 - Practice Problem

August 19th, 2025

Problem Statement: Implement the CircularQueue

Implement a circular queue `CircularQueue` with a fixed size of 5, supporting initialization, enqueue, dequeue, fullness and emptiness checks, and displaying elements.

Method Descriptions

- `CircularQueue()`: Initializes an empty circular queue of size 5 with front and rear pointers at -1.
- `boolean isFull()`: Checks if the queue is full based on front and rear positions.
- `boolean isEmpty()`: Checks if the queue is empty (front is -1).
- `enQueue(element)`: Adds an element to the rear, printing "Inserted" or "Queue is full".
- `deQueue()`: Removes and returns the front element, or -1 if empty, printing "Queue is empty".
- `Display()`: Prints front, rear, and all elements from front to rear, or "Empty Queue".

Starter Code and Sample Output

```
class CircularQueue():
    SIZE = 5

    def __init__(self):
        # Initialize queue with fixed size
        # Set front and rear pointers to -1 to indicate empty queue
        pass

    def isFull(self):
        # Check if queue is full
        # Condition: (front == 0 and rear == SIZE-1) or (rear + 1) % SIZE == front
        pass

    def isEmpty(self):
        # Check if queue is empty
        # Condition: front == -1
        pass

    def enQueue(self, element):
        # If full     print "Queue is full" and return
        # If empty     set front = 0
        # Move rear in circular manner: (rear + 1) % SIZE
        # Insert element at rear
        # Print "Inserted <element>"
        pass

    def deQueue(self):
        # If empty     print "Queue is empty" and return -1
        # Store element at front
        # If only one element was present     reset front and rear to -1
        # Else     move front in circular manner: (front + 1) % SIZE
        # Return removed element
        pass
    def Display(self):
        # If empty      print ”Empty Queue”
        # Print front and rear values
        # Print all elements from front to rear in correct order (circularly)
        pass

    def test():
        myQ = CircularQueue()    # Create queue

        myQ.deQueue()            # Remove from empty queue

        myQ.enQueue(1)           # Insert 1
        myQ.enQueue(2)           # Insert 2
        myQ.enQueue(3)           # Insert 3
        myQ.enQueue(4)           # Insert 4
        myQ.enQueue(5)           # Insert 5
        myQ.enQueue(6)           # Full queue

        myQ.Display()            # Show all elements

        x = myQ.deQueue()        # Remove 1
        x = myQ.deQueue()        # Remove 2

        myQ.Display()            # Show remaining elements

    test()  # Run test
```

Sample Output

```
Queue is empty
Inserted 1
Inserted 2
Inserted 3
Inserted 4
Inserted 5
Queue is full
front 0
rear 4
Elements
1
2
3
4
5
front 2
rear 4
Elements
3
4
5
```

Problem Statement: Balanced Parentheses Checker using Stack (Linked List Implementation)

Given a string containing parentheses `()`, square brackets `[]`, and curly braces {}, write a program using a `Stack` (implemented via a singly linked list) to determine whether the parentheses are balanced.

Requirements

- Implement the stack using a singly linked list.
- Only push opening brackets onto the stack.
- On encountering a closing bracket, check if it matches the top of the stack.
- If all brackets match correctly and the stack is empty at the end, the expression is balanced.

Starter Code

```python
class Node:
    def __init__(self, data):
        # Store data in node
        # Initialize next pointer to None
        pass

class Stack:
    def __init__(self):
        # Initialize top pointer to None
        pass

    def push(self, item):
        # Create a new node
        # Link it to the current top
        # Update top pointer
        pass

    def pop(self):
        # Check if stack is empty
        # Remove the top node and return its data
        pass

    def peek(self):
        # Return top element without removing it
        pass

    def isEmpty(self):
        # Return True if top is None
        pass


    def isBalanced(expression):
        # Create an empty stack
        # Define a mapping of closing to opening brackets
        # Traverse the expression character by character
            # If it's an opening bracket, push it onto the stack
            # If it's a closing bracket:
                # Check if stack is empty
                # Pop and compare with corresponding opening bracket
        # After traversal, return True if stack is empty, else False
        pass

    def test():
        # Test cases for checking balanced parentheses
        expressions = ["(a+b)", "[a*(b+c)]", "{[()]}", "([)]", "((()"]
        for exp in expressions:
            # Print expression and whether it is balanced
            pass

test()
```

Sample Output

```
(a+b) : Balanced
[a*(b+c)] : Balanced
{[()]} : Balanced
([)] : Not Balanced
((()) : Not Balanced
```