DATA STRUCTURE PRACTICAL ASSIGNMENT 07

Suryakant Upadhyay

20220802043

LAB-07

BATCH-A1

SEM-II

1) Write a python code to perform following operation on circular linked list.

a)Creation() b)Traversal()

a) Creation()

In [17]:
```
class Node:
    def __init_(self, data):
        self.data = data
        self.next = None
class CircularLinkedList:
    def __init__(self):
        self.last = None

    def addToEmpty(self, data):
        if (self.last != None):
            return self.last

        temp = Node(data)
        self.last = temp

        self.last.next = self.last
        return self.last

    def addBegin(self, data):
        if (self.last == None):
            return self.addToEmpty(data)

        temp = Node(data)
        temp.next = self.last.next
        self.last.next = temp
        return self.last

    def addEnd(self, data):
        if (self.last == None):
            return self.addToEmpty(data)

        temp = Node(data)
        temp.next = self.last.next
        self.last.next = temp
        self.last = temp
        return self.last

    def addAfter(self, data, item):
        if (self.last == None):
            return None

        temp = Node(data)
        p = self.last.next
        while p:
            if (p.data == item):
                temp.next = p.next
                p.next = temp

                if (p == self.last):
                    self.last = temp
                    return self.last
                else:
                    return self.last
            p = p.next
            if (p == self.last.next):
                break

        print(item, " not present in the list.")

    def traverse(self):
        if (self.last == None):
            print("List is empty")
            return

        temp = self.last.next
        while temp:
            print(temp.data, end=" ")
            temp = temp.next
            if temp == self.last.next:
                break
# create a circular linked list
cllist = CircularLinkedList()

# add nodes to the list
cllist.addToEmpty(1)
cllist.addBegin(2)
cllist.addEnd(3)
cllist.addAfter(4, 3)

# traverse the list
cllist.traverse()

```

```
2 1 3 4
```

b) Traversal()

In [19]:
```
# Node class to represent a single node in the circular linked list
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

# Circular linked list class
class CircularLinkedList:
    def __init__(self):
        self.head = None

    # Function to traverse and print the circular linked list
    def traversal(self):
        if self.head is None:
            print("Circular linked list is empty!")
        else:
            current = self.head
            print("Nodes of the Circular Linked List: ")
            # Traverse the linked list until we reach the head node again
            while True:
                print(current.data, end=' ')
                current = current.next
                if current == self.head:
                    break
            print()

# Example usage
if __name   == '__main__':
    cll = CircularLinkedList()
    cll.head = Node(1)
    second = Node(2)
    third = Node(3)
    fourth = Node(4)

    # Linking the nodes in a circular manner
    cll.head.next = second
    second.next = third
    third.next = fourth
    fourth.next = cll.head

    # Traversing the circular linked list
    cll.traversal()

```

```
Nodes of the Circular Linked List:
1 2 3 4
```

2) Write a python code to implement stack to perform following operation - push,pop,peek(static and dynamic implementation).

a) Static Implementation.

In [22]:
```python
class Stack:
    def __init_(self, max_size):
        self.max_size = max_size
        self.stack = [None] * max_size
        self.top = -1

    def push(self, data):
        if self.top == self.max_size - 1:
            print("Stack Overflow!")
        else:
            self.top += 1
            self.stack[self.top] = data

    def pop(self):
        if self.top == -1:
            print("Stack Underflow!")
            return None
        else:
            data = self.stack[self.top]
            self.top -= 1
            return data

    def peek(self):
        if self.top == -1:
            print("Stack is empty!")
            return None
        else:
            return self.stack[self.top]


# Example usage
if__name__ == '__main__':
    s = Stack(5)
    s.push(1)
    s.push(2)
    s.push(3)
    print(s.peek())    # Output: 3
    s.pop()
    s.pop()
    s.pop()
    print(s.pop())    # Output: Stack Underflow!

```

```
3
Stack Underflow!
None
```

b) Dynamic Implementation.

In [24]:
```python
class Stack:
    def __init__(self):
        self.stack = []
        self.top = -1

    def push(self, data):
        self.stack.append(data)
        self.top += 1

    def pop(self):
        if self.top == -1:
            print("Stack Underflow!")
            return None
        else:
            data = self.stack.pop()
            self.top -= 1
            return data

    def peek(self):
        if self.top == -1:
            print("Stack is empty!")
            return None
        else:
            return self.stack[self.top]


# Example usage
if__name__ == '__main__':
    s = Stack()
    s.push(1)
    s.push(2)
    s.push(3)
    print(s.peek())   # Output: 3
    s.pop()
    s.pop()
    s.pop()
    print(s.pop())    # Output: Stack Underflow!

```

```
3
Stack Underflow!
None
```

3) Write a python code to implement queue to perform following operation - enqueue,dequeue,peek(static and dynamic implementation).

a) Static Implementation.

In [26]:
```python
class Queue:
    def__init_(self, max_size):
        self.max_size = max_size
        self.queue = [None] * max_size
        self.front = -1
        self.rear = -1

    def enqueue(self, data):
        if self.rear == self.max_size - 1:
            print("Queue Overflow!")
        elif self.front == -1 and self.rear == -1:
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear += 1
            self.queue[self.rear] = data

    def dequeue(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue Underflow!")
            return None
        else:
            data = self.queue[self.front]
            self.front += 1
            return data

    def peek(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue is empty!")
            return None
        else:
            return self.queue[self.front]


# Example usage
if__name__ == '__main__':
    q = Queue(5)
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print(q.peek())    # Output: 1
    q.dequeue()
    q.dequeue()
    q.dequeue()
    print(q.dequeue())    # Output: Queue Underflow!

```

```
1
Queue Underflow!
None
```

b) Dynamic Implementation.

In [27]:
```
class Queue:
    def __init__(self):
        self.queue = []
        self.front = -1
        self.rear = -1

    def enqueue(self, data):
        self.queue.append(data)
        if self.front == -1 and self.rear == -1:
            self.front = 0
            self.rear = 0
        else:
            self.rear += 1

    def dequeue(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue Underflow!")
            return None
        else:
            data = self.queue[self.front]
            self.front += 1
            return data

    def peek(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue is empty!")
            return None
        else:
            return self.queue[self.front]


# Example usage
if __name__ == '__main__':
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print(q.peek())   # Output: 1
    q.dequeue()
    q.dequeue()
    q.dequeue()
    print(q.dequeue())    # Output: Queue Underflow!

```

```
1
Queue Underflow!
None
```