# LAB NO 11

# STACK ALGORITHM

## AIM:

To understand the concepts of Stack data structure in Python.

## INTRODUCTION:

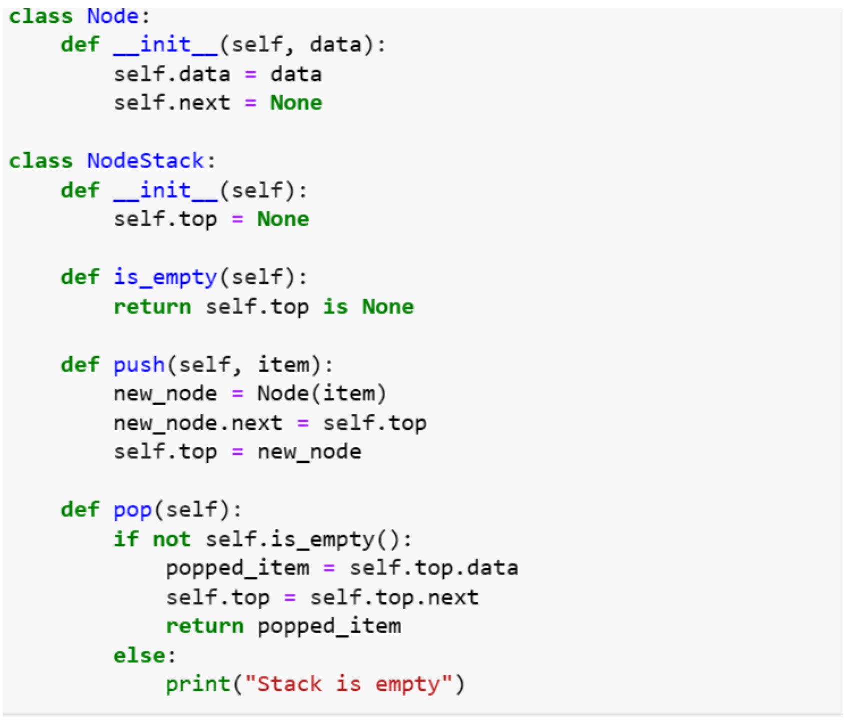
A stack is a fundamental data structure that follows the Last In, First Out (LIFO) principle. This means the last element added to the stack is the first to be removed.

## TYPES OF STACK:

i. List Stack
ii. Deque Stack
iii. Node base Stack.

## LAB REPORT TASK

i. Make a node base implementation for Stack and pass different data types and push and pop different items and display the top item



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

class NodeStack:
    def __init__(self):
        self.top = None

    def is_empty(self):
        return self.top is None

    def push(self, item):
        new_node = Node(item)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if not self.is_empty():
            popped_item = self.top.data
            self.top = self.top.next
            return popped_item
        else:
            print("Stack is empty")
```

## IMPLEMENTATION OF STACKS IN PYTHON:

### 1. List Stack:

```python
class StackList:
    def __init__(self):
        self.items = []

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("Stack is empty")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            print("Stack is empty")

    def size(self):
        return len(self.items)
```

## 2. Creating object of this class:

```python
# Example usage:
stack = StackList()
```

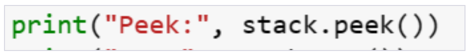
### In this implementation:

- The `__init__` method initializes an empty list to store the stack elements.
- `is_empty()` checks if the stack is empty.
- `push(item)` adds an item to the top of the stack using the `append()` method.

```python
stack.push(1)
stack.push(2)
stack.push(3)
```

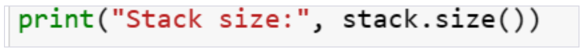
- `pop()` removes and returns the item from the top of the stack using the `pop()` method.
  `print("Pop:", stack.pop())`

- `peek()` returns the item from the top of the stack without removing it.



```
print("Peek:", stack.peek())
```

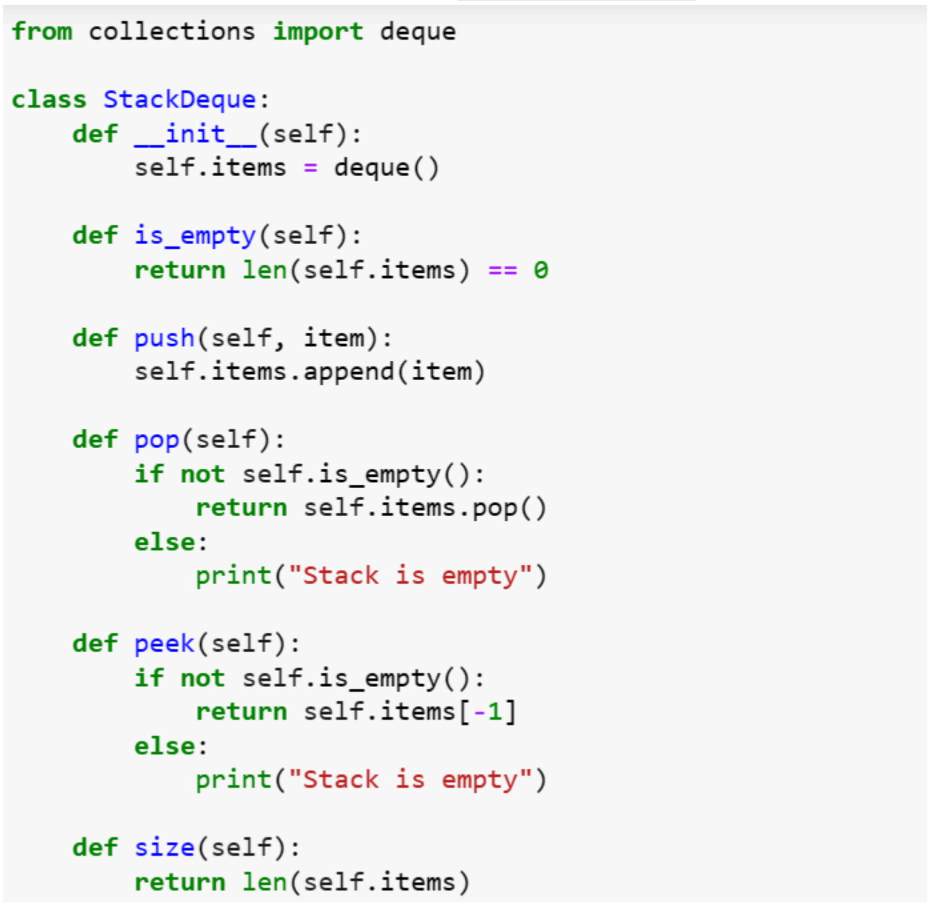
- `size()` returns the number of elements in the stack.



```
print("Stack size:", stack.size())
```

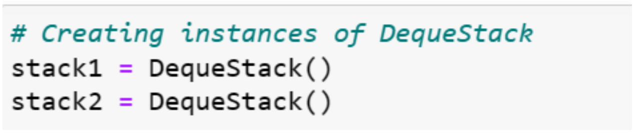
## 1. Deque Implementation:

i. A deque (double-ended queue) is a versatile data structure that allows efficient insertion and deletion operations at both ends. In Python, you can implement a deque using the **collections.deque** class.



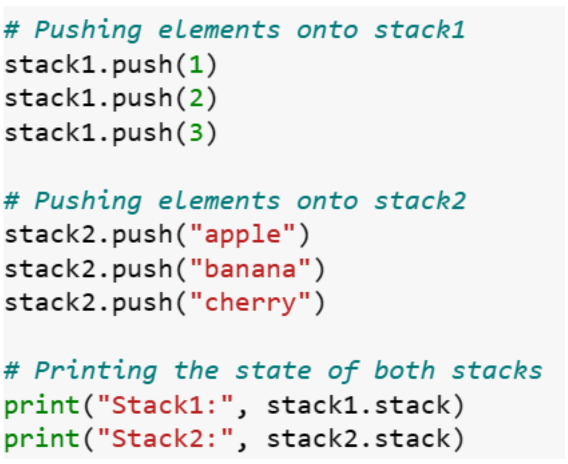
```
from collections import deque

class StackDeque:
    def __init__(self):
        self.items = deque()

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("Stack is empty")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            print("Stack is empty")

    def size(self):
        return len(self.items)
```

### a. Creating Data types in Deque Stack Implementation.



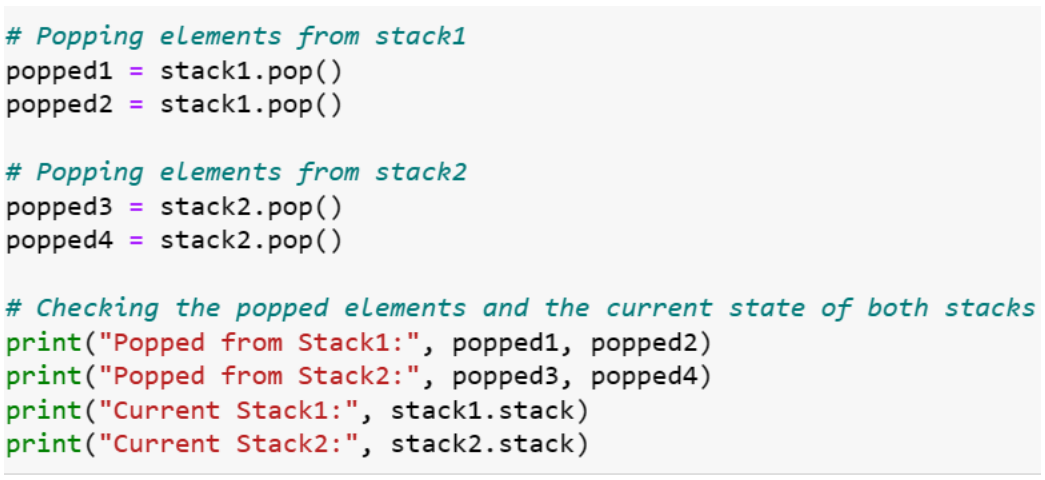
```
# Creating instances of DequeStack
stack1 = DequeStack()
stack2 = DequeStack()
```

### b. Pushing element in Stack:



```python
# Pushing elements onto stack1
stack1.push(1)
stack1.push(2)
stack1.push(3)

# Pushing elements onto stack2
stack2.push("apple")
stack2.push("banana")
stack2.push("cherry")

# Printing the state of both stacks
print("Stack1:", stack1.stack)
print("Stack2:", stack2.stack)
```

### c. Popping out element from Stack.



```python
# Popping elements from stack1
popped1 = stack1.pop()
popped2 = stack1.pop()

# Popping elements from stack2
popped3 = stack2.pop()
popped4 = stack2.pop()

# Checking the popped elements and the current state of both stacks
print("Popped from Stack1:", popped1, popped2)
print("Popped from Stack2:", popped3, popped4)
print("Current Stack1:", stack1.stack)
print("Current Stack2:", stack2.stack)
```

Difference between Deque and List Implementation of Stack.

| **Deque** | **List** |
|---|---|
| Deques are optimized for fast appends and pops from both ends, making them more efficient than lists for implementing stacks. | Lists are a built-in data type in Python and are versatile. |
| Appending and popping elements from both ends of a deque is O(1) time on average. | List operations, such as indexing, are generally O(1). |
| Deques are slightly more memory-intensive than lists. | Appending and popping elements from the end of a list (used as a stack) takes O(1) time on average, but occasionally O(n) time when the list needs to be resized. |

## 2. NODE BASE IMPLEMENTATION:

i. A node-based implementation refers to a data structure or algorithm design that involves the use of nodes, where each node contains data and a reference (or link) to the next node in the sequence. This is commonly seen in linked lists, stacks, and queues.

ii. The **Node** class represents each element in the stack with a **data** attribute and a **next** attribute pointing to the next node.

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

- The **NodeStack** class maintains a reference to the top of the stack (**top**).

```
class NodeStack:
    def __init__(self):
        self.top = None
```

- The **`push()`** method adds a new node to the top of the stack.

```
def push(self, item):
    new_node = Node(item)
    new_node.next = self.top
    self.top = new_node
```

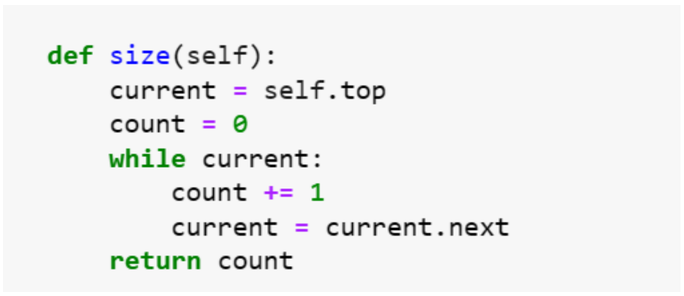
- The **`pop()`** method removes and returns the item from the top of the stack.

```
def pop(self):
    if not self.is_empty():
        popped_item = self.top.data
        self.top = self.top.next
        return popped_item
    else:
        print("Stack is empty")
```

- The **`peek()`** method returns the item from the top of the stack without removing it.

```
def peek(self):
    if not self.is_empty():
        return self.top.data
    else:
        print("Stack is empty")
```

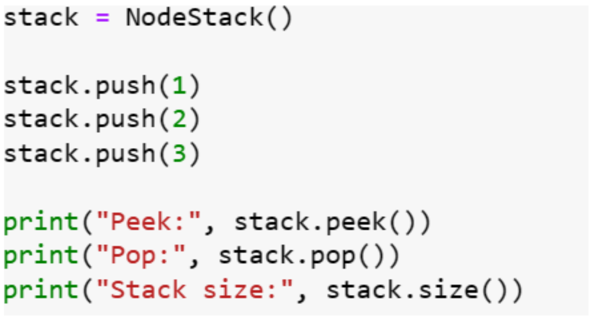
- The **`size()`** method calculates the number of elements in the stack by traversing the linked nodes.



```python
def size(self):
    current = self.top
    count = 0
    while current:
        count += 1
        current = current.next
    return count
```

## 3. NOW CREATING INSTANCES:

Creating instances of Node base Stack and apply push pop and other methods on that instance.



```python
stack = NodeStack()

stack.push(1)
stack.push(2)
stack.push(3)

print("Peek:", stack.peek())
print("Pop:", stack.pop())
print("Stack size:", stack.size())
```

# LAB 12
# QUEUE ALGORITHMS

## AIM:

To understand the concept of queue algorithms in OOP (PYTHON).

## INTRODUCTION:

A queue is a fundamental data structure in computer science that follows the First-In-First-Out (FIFO) principle. In a queue, elements are added at the rear (enqueue) and removed from the front (dequeue). This ensures that the oldest element in the queue is the first to be processed.

## TYPES OF QUEUE:

i. List Queue
ii. Deque Queue
iii. Node base Queue.

### 1. LIST QUEUE:

a. Run this Code:

```python
class Queue:
    def __init__(self):
        self.items = []

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

i. **__init__(self):**This is the constructor method that initializes a new instance of the **Queue** class. It creates an empty list (**self.items**) to store the elements of the queue

ii. **is_empty(self)**: This method returns **True** if the queue is empty. (i.e., it has no elements), and **False** otherwise. It checks the length of the list (**self.items**) to determine if the queue is empty.

**b. Run the Following Code:**

```python
def enqueue(self, item):
        self.items.append(item)
```

iii. **enqueue(self, item)**: This method adds an item to the rear of the queue. It appends the specified item to the end of the list (**self.items**), effectively adding it to the rear of the queue.

**c. Now Run this Code:**

```
def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            print("Queue is empty. Cannot dequeue.")
```

iv. **dequeue(self)**: This method removes and returns the item from the front of the queue. It uses the **pop(0)** method to remove and return the first element in the list. Before attempting to dequeue, it checks if the queue is empty using the **is_empty** method to avoid errors.

**d. Now Run this code:**

```
# Example usage:
my_queue = Queue()

my_queue.enqueue(1)
my_queue.enqueue(2)
my_queue.enqueue(3)

print("Dequeue:", my_queue.dequeue())
print("Dequeue:", my_queue.dequeue())

print("Is the queue empty?", my_queue.is_empty())
```

v. This example creates a **Queue** object, enqueues three elements (1, 2, and 3), prints  the queue, dequeues two elements, and checks if the queue is empty. The output should demonstrate the basic functionality of the queue.

## 2. DEQUE QUEUE:

```
from collections import deque

class DequeQueue:
    def __init__(self):
        self.items = deque()

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

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.popleft()
        else:
            print("Queue is empty. Cannot dequeue.")
```

i. **__init__(self)**: The constructor initializes a new instance of the **DequeQueue** class. It creates an empty deque (**self.items**) to store the elements of the queue.

```
def __init__(self):
        self.items = deque()
```

ii. **is_empty(self)**: This method returns **True** if the queue is empty (i.e., it has no elements), and **False** otherwise. It checks the length of the deque (**self.items**) to determine if the queue is empty.

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

iii. **enqueue(self, item)**: This method adds an item to the rear of the queue. It uses the **append** method of the deque to add the specified item to the end, effectively enqueueing it.

```
def enqueue(self, item):
        self.items.append(item)
```

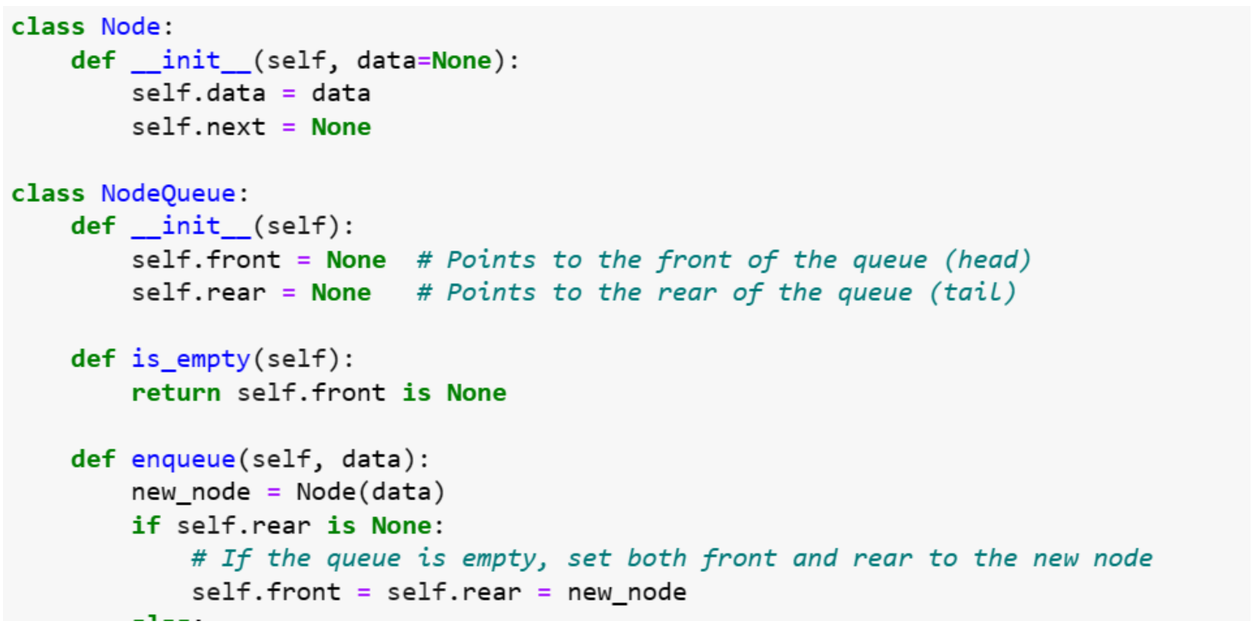
iv. **dequeue(self)**: This method removes and returns the item from the front of the queue. It uses the **popleft** method of the deque to remove and return the leftmost element. Before attempting to dequeue, it checks if the queue is empty using the **is_empty** method to avoid errors.

```
def dequeue(self):
        if not self.is_empty():
            return self.items.popleft()
        else:
            print("Queue is empty. Cannot dequeue.")
```

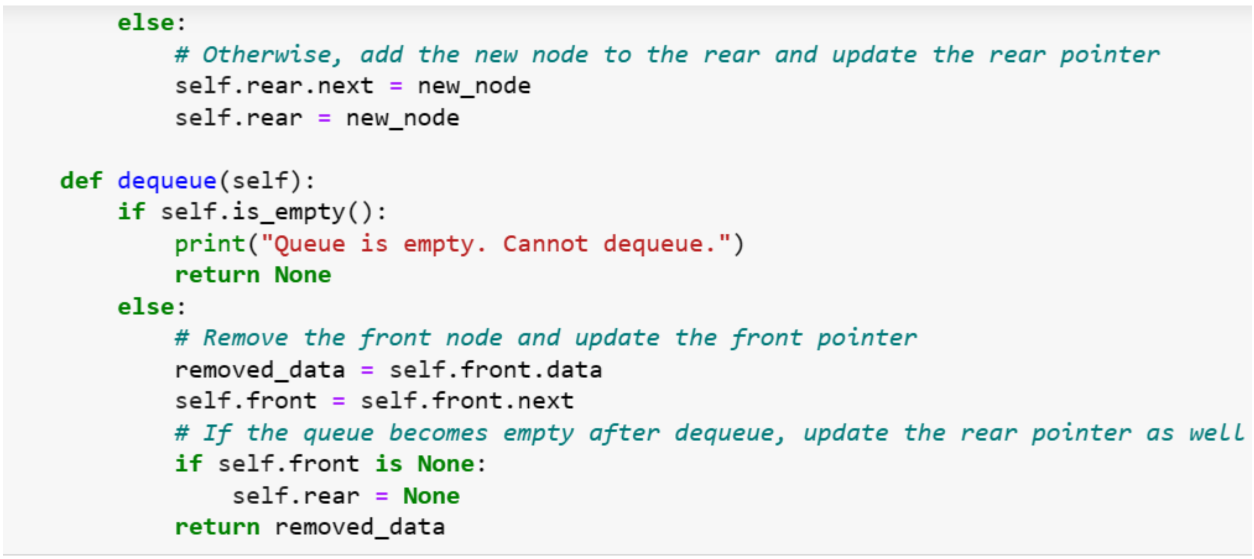
**Now, let's create instances and demonstrate the usage:**

v. This creates a **DequeQueue** object, enqueues three elements, prints the queue, dequeues two elements, and checks if the queue is empty. The output should demonstrate the basic functionality of the deque-based queue.

## 3. NODE BASE QUEUE:

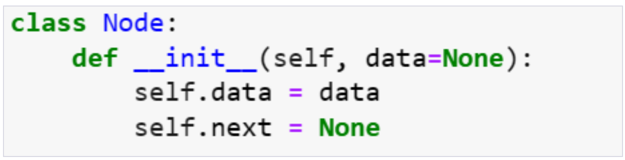


```python
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class NodeQueue:
    def __init__(self):
        self.front = None  # Points to the front of the queue (head)
        self.rear = None   # Points to the rear of the queue (tail)

    def is_empty(self):
        return self.front is None

    def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:
            # If the queue is empty, set both front and rear to the new node
            self.front = self.rear = new_node
```



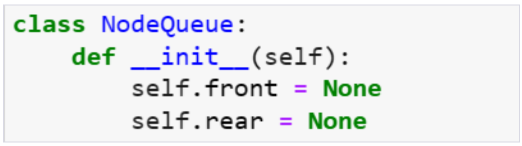
```python
        else:
            # Otherwise, add the new node to the rear and update the rear pointer
            self.rear.next = new_node
            self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
            return None
        else:
            # Remove the front node and update the front pointer
            removed_data = self.front.data
            self.front = self.front.next
            # If the queue becomes empty after dequeue, update the rear pointer as well
            if self.front is None:
                self.rear = None
            return removed_data
```

i. **Node** class: This class represents a node in the queue. Each node contains data and a reference to the next node in the queue.



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

ii. **__init__(self)**: The constructor initializes a new instance of the **NodeQueue** class. It sets both the front and rear pointers to **None** because the queue is initially empty.



```python
class NodeQueue:
    def __init__(self):
        self.front = None
        self.rear = None
```

iii. **is_empty(self)**: This method returns **True** if the queue is empty (i.e., front is **None**), and **False** otherwise.

```python
def is_empty(self):
        return self.front is None
```

iv. **enqueue(self, data)**: This method adds a new node with the given data to the rear of the queue. If the queue is empty, both the front and rear pointers are set to the new node.

```python
def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
```

v. **dequeue(self)**: This method removes and returns the data from the front of the queue. If the queue becomes empty after dequeue, the rear pointer is also updated to **None**.

```python
def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
            return None
        else:
            removed_data = self.front.data
            self.front = self.front.next
            if self.front is None:
                self.rear = None
            return removed_data
```

vi. This  creates a **NodeQueue** object, enqueues three elements, prints the queue, dequeues two elements, and checks if the queue is empty. The output should demonstrate the basic functionality of the node-based queue.

```python
# Example usage:
my_node_queue = NodeQueue()

my_node_queue.enqueue(1)
my_node_queue.enqueue(2)
my_node_queue.enqueue(3)

print("Dequeue:", my_node_queue.dequeue())
print("Dequeue:", my_node_queue.dequeue())

print("Is the queue empty?", my_node_queue.is_empty())
```

THANKS !!

THAT'S FOR TODAY

# Lab 13

## PRIORITY QUEUE

## Aim:

The aim of this lab is to understand the concepts of priority queue and how it works.

## Introduction:

A priority queue is a data structure that stores elements along with associated priorities. The key feature of a priority queue is that the element with the highest (or lowest) priority is always at the front and is the next to be removed.

## 1. Lab work task:

### Importing heapq for priority queue:

**We use the heapq module for its efficient implementation of a binary heap, which is the underlying data structure for our priority queue.**

```
import heapq
```

## 2. Enqueue (Insert):

Adds an element to the priority queue with a specified priority.

When you push a new element onto a non-empty heap, it'll end up in the right spot, maintaining the heap invariant.

Note:

Note that this doesn't necessarily mean that the resulting elements will become sorted.

### IMPLEMENTING PUSH OPERATION:

### CODE:

```
from heapq import heappush
fruits = []
heappush(fruits, "orange")
heappush(fruits, "apple")
heappush(fruits, "banana")
```

RUN THE FOLLOWING CODE:

### CODE:

```
Print(Fruits)
```

Fruit names in the resulting heap in the example above don't follow alphabetical order.

## 3. Dequeue (Remove):

When you pop an element from a heap, you'll always get the first one, while the remaining elements might shuffle a little bit:

In the unordered list, elements are added without any specific order, and the dequeue operation simply removes elements from the front, which may not reflect any priority order.

**RUN THE FOLLOWING CODE:**

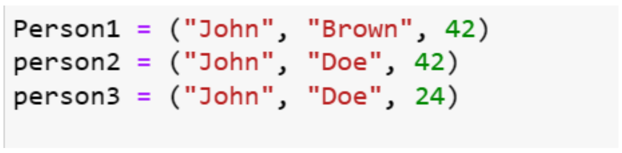
```
from heapq import heappop

heappop(fruits)

fruits
```

Notice how the banana and orange swapped places to ensure the first element continues to be the smallest.

**COMPAIRING STRING OBJECTS:**

a. When you tell Python to compare two string objects by value, it starts looking at their characters pairwise from left to right and checks each pair one by one . Then after processing the string it will give you the output.

CODE:



```
Person1 = ("John", "Brown", 42)
person2 = ("John", "Doe", 42)
person3 = ("John", "Doe", 24)
```

RUN THE CODE:

- `Person1 < person2`
- `person2 < person3`

## 4. BUILDING PRIORITY QUEUE DATA TYPE:

a. To build a priority queue data type, you can use the heapq module in Python, which provides a convenient way to implement a priority queue using a binary heap.

**CODE:**

```python
from collections import deque
from heapq import heappop, heappush

class PriorityQueue:
    def __init__(self):
        self._elements = []

    def enqueue_with_priority(self, priority, value):
        heappush(self._elements, (priority, value))

    def dequeue(self):
        return heappop(self._elements)
```

i. It's a basic priority queue implementation, which defines a heap of elements using a Python list and two methods that manipulate it. The enqueue_with_priority() method takes two arguments, a priority and a corresponding value.

```python
#Assigning priorities value to some strings

CRITICAL = 3
IMPORTANT = 2
NEUTRAL = 1

messages = PriorityQueue()
messages.enqueue_with_priority(IMPORTANT, "Windshield wipers turned on")
messages.enqueue_with_priority(NEUTRAL, "Radio station tuned in")
messages.enqueue_with_priority(CRITICAL, "Brake pedal depressed")
messages.enqueue_with_priority(IMPORTANT, "Hazard lights turned on")
```

**RUN THE FOLLOWING CODES:**

- `messages._elements`
- `messages.dequeue()`

**NOTE:**

You defined three priority levels: critical, important, and neutral. Next, you instantiated a priority queue and used it to enqueue a few messages with those priorities. However, instead of dequeuing the message with the highest priority, you got a tuple corresponding to the message with the lowest priority.

## 5. NEGATIVE PRIORITY:

i. By just making the priority negative you will get With this small change, you'll push critical messages ahead of important and neutral ones.
ii. You will unpack the value from the tuple by accessing its second component, located at index one using the square bracket ( [ ] ) syntax.

## CODE:

```python
class PriorityQueue:
    def __init__(self):
        self._elements = []

    def enqueue_with_priority(self, priority, value):
        heappush(self._elements, (-priority, value))

    def dequeue(self):
        return heappop(self._elements)[1]
```

**NOW RUN THE FOLLOWING CODE:**

#Run the codes one by one and observe outputs.

- `messages.dequeue()`
- `messages.dequeue()`
- `messages.dequeue()`
- `messages.dequeue()`

## NOTE:

a. You get the critical message 1st followed by the two important ones, and then the neutral message.

HANDLING CORNER CASES IN PRIORITY QUEUE:

COMPARISON CASE:

## CODE:

```python
from dataclasses import dataclass

@dataclass
class Message:
    event: str
```

```python
wipers = Message("Windshield wipers turned on")
hazard_lights = Message("Hazard lights turned on")
```

**NOW RUN THE FOLLOWING CODE AND OBSERVE OUTPUT:**

**wipers < hazard_lights**

a. Message objects might be more convenient to work with than plain strings, but unlike strings, they aren't comparable unless you tell Python how to perform the comparison. As you will see, custom class instances don't provide the implementation for the less than (<) operator by default.

## CODE:

```
messages = PriorityQueue()
messages.enqueue_with_priority(CRITICAL, wipers)
messages.enqueue_with_priority(IMPORTANT,hazard_lights)
```

## NOW RUN THE CODE AND OBSERVE OUTPUT:

```
messages.enqueue_with_priority(CRITICAL,message("ABS engaged"))
```

This time the comparison will fails because two of the messages are of equal priority and Python falls back to comparing them by value, which you haven't defined for your custom Message class instances.

You can use the count() iterator from the itertools module to count from zero to infinity in a concise way:

## CODE:

```
from collections import deque
from heapq import heappop, heappush
from itertools import count

class PriorityQueue:
    def __init__(self):
        self._elements = []
        self._counter = count()

    def enqueue_with_priority(self, priority, value):
        element = (-priority, next(self._counter), value)
        heappush(self._elements, element)

    def dequeue(self):
        return heappop(self._elements)[-1]
```

i. The counter gets initialized when you create a new PriorityQueue instance. Whenever you enqueue a value, the counter increments and retains its current state in a tuple pushed onto the heap.

**Refactoring The Code Using A Mixin Class**:

Now we will make the same code using mixin clases __len__ and _iter__.

## CODE:

```python
class IterableMixin:
    def __len__(self):
        return len(self._elements)

    def __iter__(self):
        while len(self) > 0:
            yield self.dequeue()
```

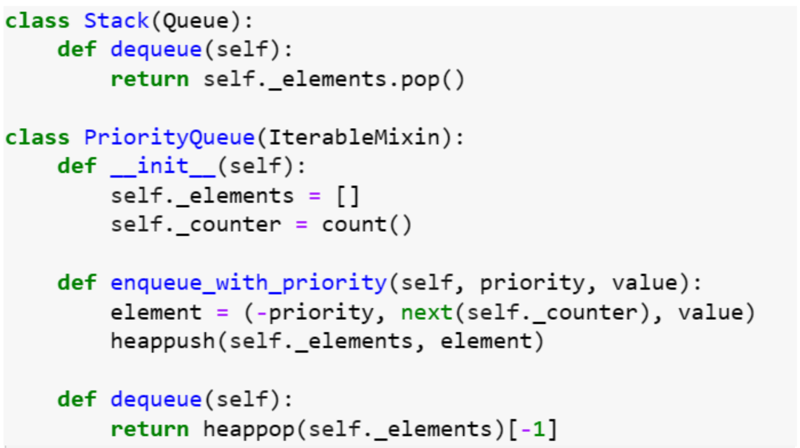
## ENCOURAGING :

With the three queue classes in place, you can finally apply them to solving a real problem.

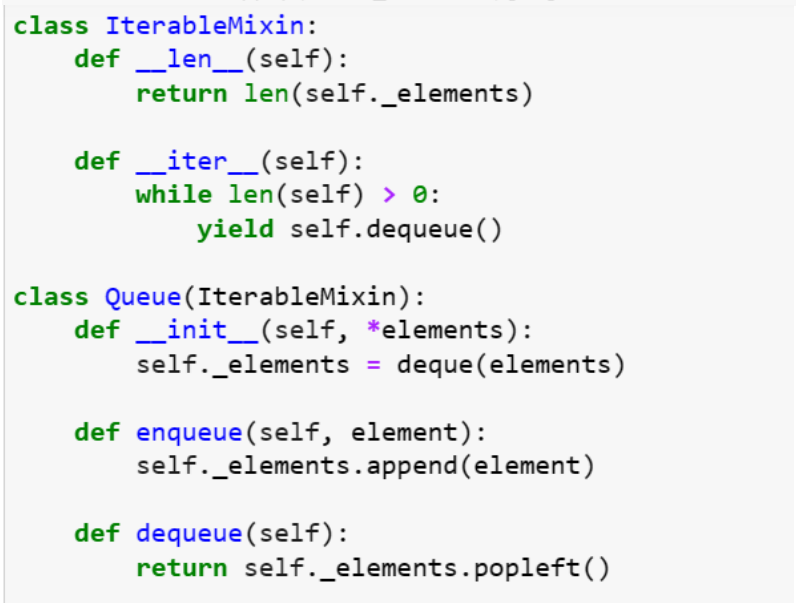
## Note:

- You moved the .__len__() and .__iter__() methods from the Queue class to a separate IterableMixin class and made the former extend that mixin

```python
from collections import deque
from heapq import heappop, heappush
from itertools import count
```



```python
class Stack(Queue):
    def dequeue(self):
        return self._elements.pop()

class PriorityQueue(IterableMixin):
    def __init__(self):
        self._elements = []
        self._counter = count()

    def enqueue_with_priority(self, priority, value):
        element = (-priority, next(self._counter), value)
        heappush(self._elements, element)

    def dequeue(self):
        return heappop(self._elements)[-1]
```



```python
class IterableMixin:
    def __len__(self):
        return len(self._elements)

    def __iter__(self):
        while len(self) > 0:
            yield self.dequeue()

class Queue(IterableMixin):
    def __init__(self, *elements):
        self._elements = deque(elements)

    def enqueue(self, element):
        self._elements.append(element)

    def dequeue(self):
        return self._elements.popleft()
```

THANKS !!

THAT'S FOR TODAY!