NANYANG TECHNOLOGICAL UNIVERSITY

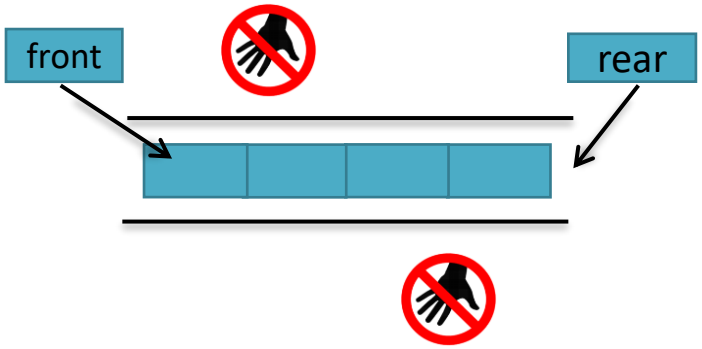
Data Structures & Algorithms in Python: Queues

Dr. Owen Noel Newton Fernando

College of Computing and Data Science

Queues

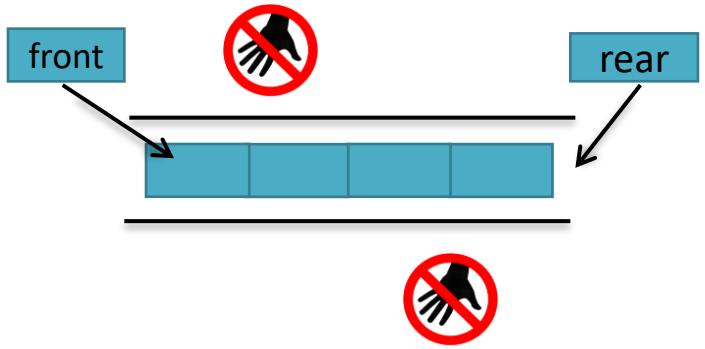
- Elements are added only at the rear and removed from the front
- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- Can be implemented by list or linked list



Implementing a Queue using Linked List

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

```
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0
```



Due to algorithmic efficiency, a rear pointer is introduced to optimize enqueue operations, allowing new elements to be added efficiently without requiring traversal.

Core Queue operations

- `getFront()`: Inspect the item at the front of the queue without removing it
- `enqueue()`: Add an item at the end of the queue
- `dequeue()`: Remove an item from the top of the queue
- `isEmpty()`: Check if the queue has no more items remaining
- `getSize()`: Returns the current size of the stack

getFront()

- Inspect the item at the front of the queue without removing it

```
def getFront(self):
        if self.isEmpty():
            raise IndexError("Peek from empty queue")
        return self.front.data
```

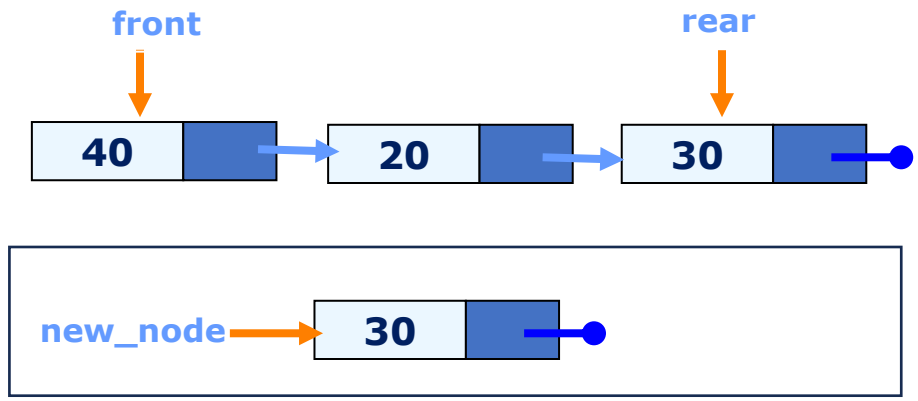
enqueue()

- Insert a new node at the rear of queue
- If we do not have the rear to track the last node of a queue, then we always need to use a loop to retrieve the last node from the first node
  - It is very inefficient
  - It is not the purpose to have a queue structure

```
def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```

enqueue()

```
def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```



Step 1: **Create a new node with the data.**

Step 2: If the queue is empty, set the front to the new node.

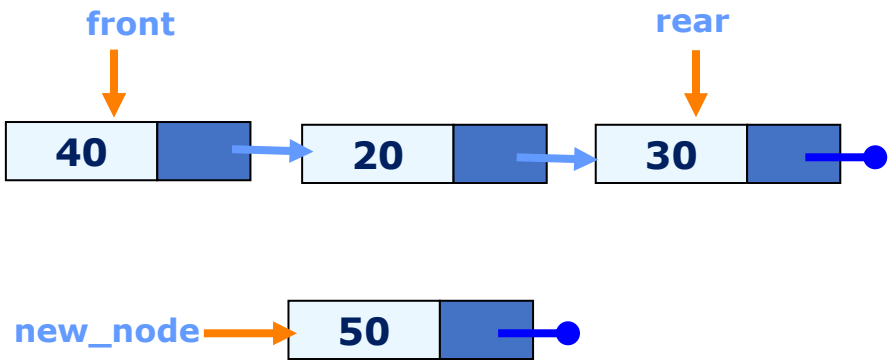
Step 3: Otherwise, link the current rear node to the new node.

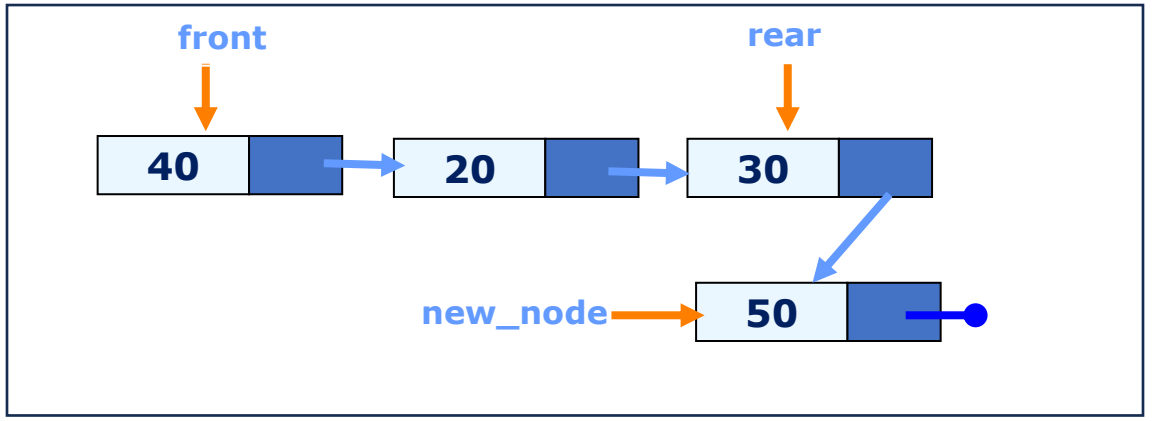
Step 4: Make the new node the rear.

Step 5: Increase the size count

enqueue()

```
def enqueue(self, data):
    new_node = Node(data)
    if self.isEmpty():
        self.front = new_node
    else:
        self.rear.next = new_node
    self.rear = new_node
    self.size += 1
```





Step 1: Create a new node with the data.

Step 2: If the queue is empty, set the front to the new node.

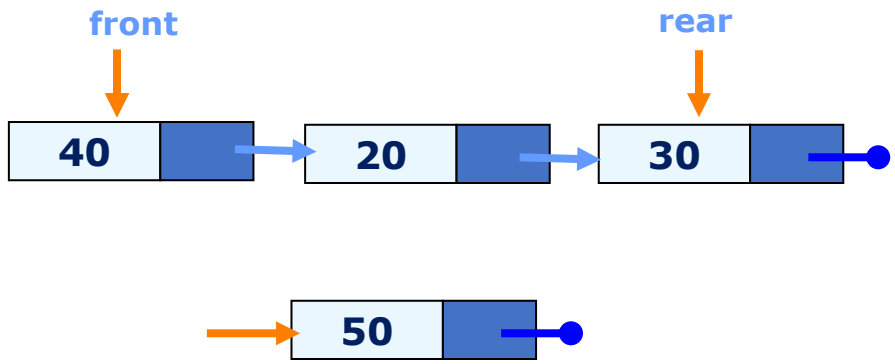
Step 3: **Otherwise, link the current rear node to the new node.**

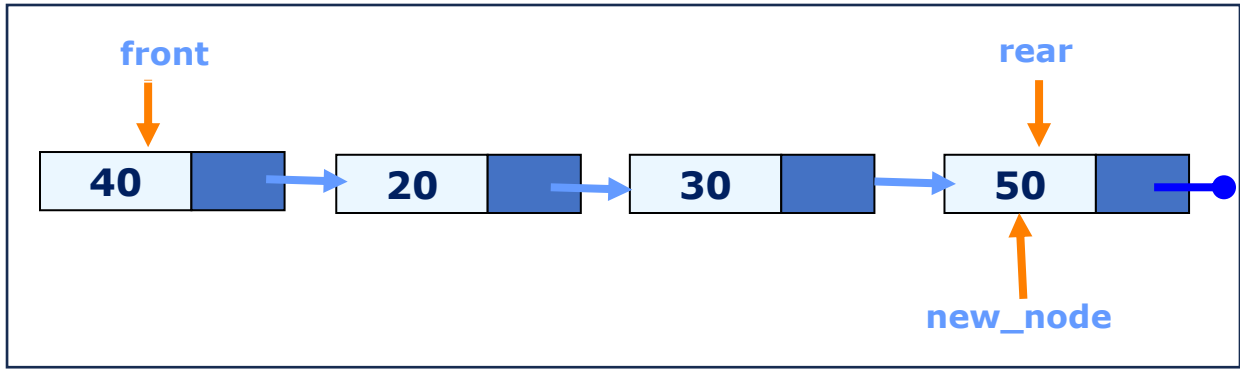
Step 4: Make the new node the rear.

Step 5: Increase the size count

enqueue()

```
def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```





Step 1: Create a new node with the data.

Step 2: If the queue is empty, set the front to the new node.

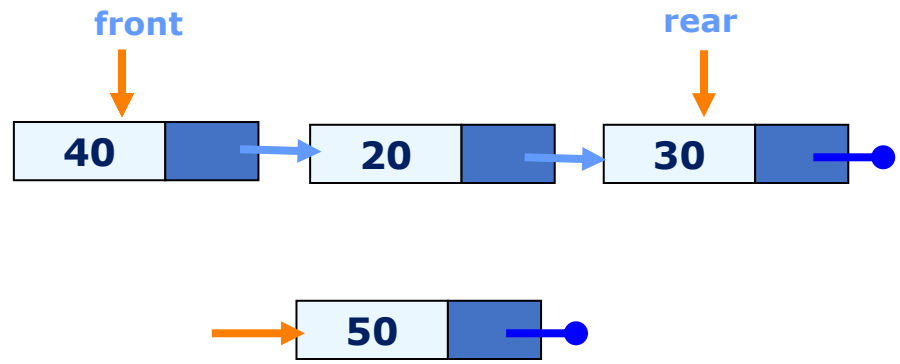
Step 3: Otherwise, link the current rear node to the new node.

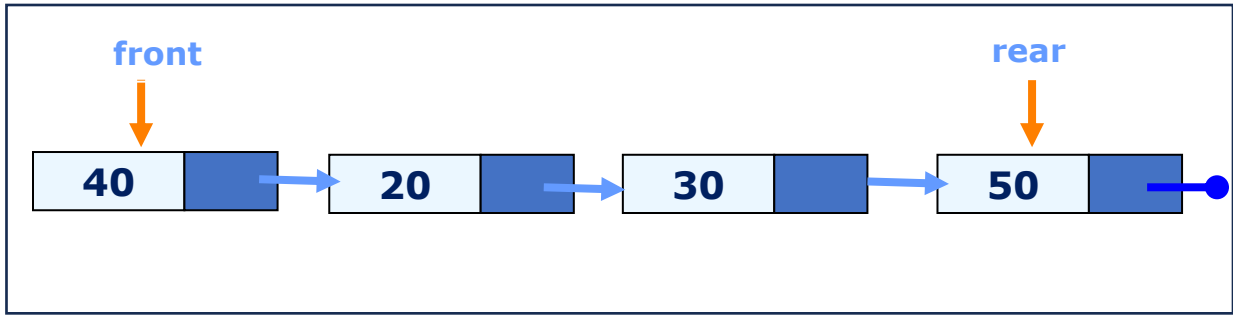
Step 4: **Make the new node the rear.**

Step 5: Increase the size count

enqueue()

```
def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```





Step 1: Create a new node with the data.

Step 2: If the queue is empty, set the front to the new node.

Step 3: Otherwise, link the current rear node to the new node.

Step 4: **Make the new node the rear.**

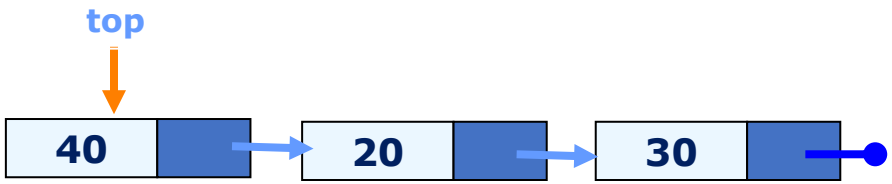
Step 5: Increase the size count

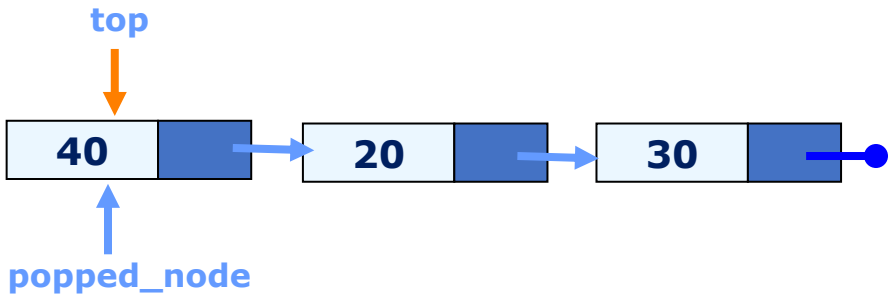
Note that new_node is a local variable inside enqueue, so once the method completes, the variable itself goes out of scope.

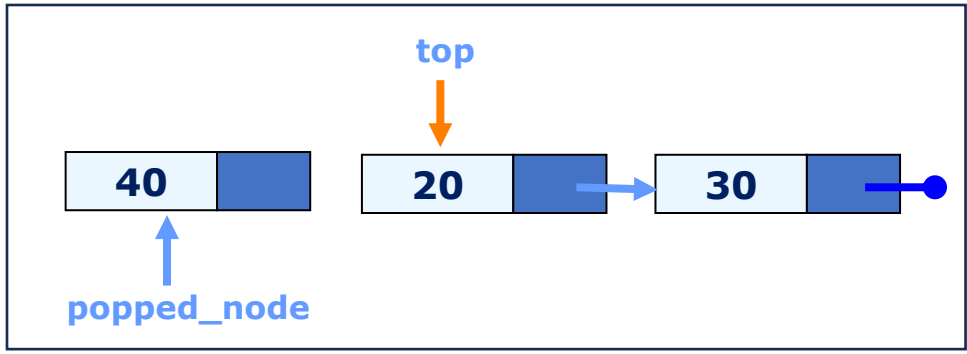
dequeue()

- Remove a new node from the Queue

```
def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        dequeued_node = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self.size -= 1
        return dequeued_node.data
```







isEmpty()

- Check whether the queue is empty

```
def isEmpty(self):
        return self.front is None
```

getSize()

- Returns the current size of the queue

```
def getSize(self):
        return self.size
```

Working Example- 01

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

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0

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

    def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```

```
    def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        dequeued_node = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self.size -= 1
        return dequeued_node.data

    def getFront(self):
        if self.isEmpty():
            raise IndexError("Peek from empty queue")
        return self.front.data

    def getSize(self):
        return self.size
```

Working Example- 01 (cont.)

```
40 if __name__ == "__main__":
41
42     queue = Queue()
43
44     queue.enqueue(10)
45     queue.enqueue(20)
46     queue.enqueue(30)
47
48     print("Front element before dequeue:", queue.getFront())
49
50     print("Size before dequeue:", queue.getSize())
51
52     print("Dequeued element:", queue.dequeue())
53
54     print("Front element after dequeue:", queue.getFront())
55
56     print("Size after dequeue:", queue.getSize())
```

Implementing a Queue class using Linked List

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

```
class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None
        self.tail = None
```

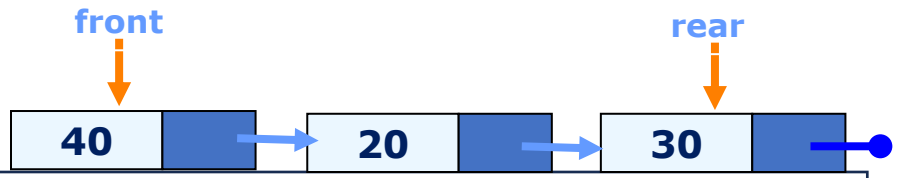
```
class Queue:
    def __init__(self):
        self.ll = LinkedList()
```

Core Queue operations

- `getFront()`: Inspect the item at the front of the queue without removing it
- `enqueue()`: Add an item at the end of the queue
- `dequeue()`: Remove an item from the top of the queue
- `isEmpty()`: Check if the queue has no more items remaining
- `getSize()`: Returns the current size of the stack

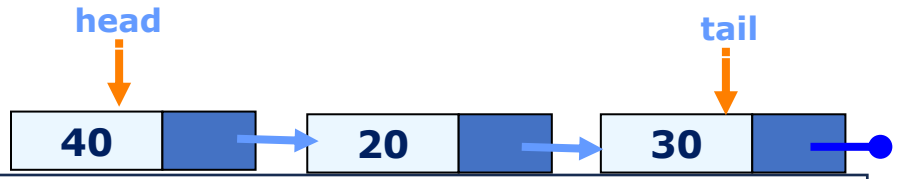
Core Queue operations: getFront():

Previous version



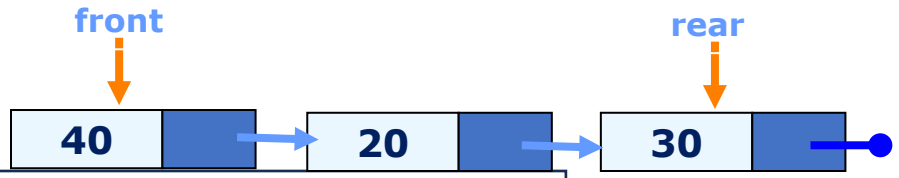

```
    def getFront(self):
            if self.isEmpty():
                raise IndexError("Peek from empty queue")
            return self.front.data
```

New Version



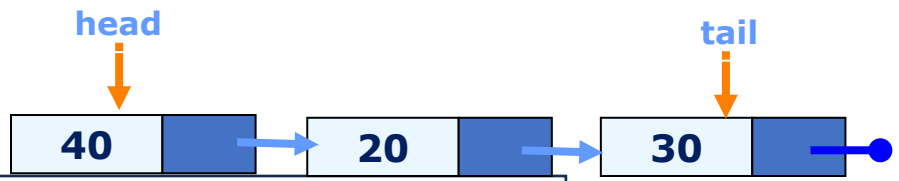

```
def getFront(self):
    if self.isEmpty():
        raise IndexError("Peek from empty stack")
    return self.ll.head.data
```

Core stack operations: enqueue()



Previous version

```
def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```

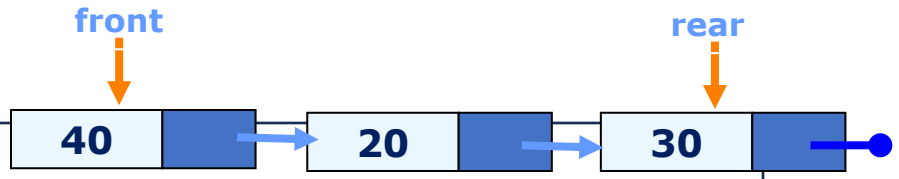


New version

```
def enqueue(self, data):
        self.ll.insert_node(self.ll.size, data)
```

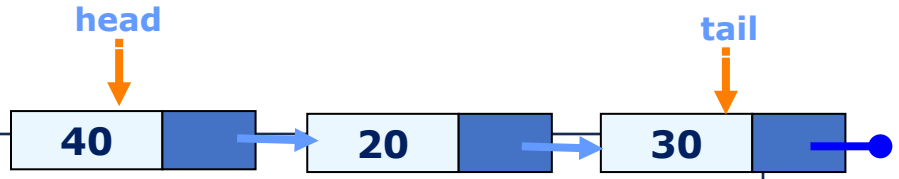
Core stack operations: dequeue()

Previous version




```
def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        dequeued_node = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self.size -= 1
        return dequeued_node.data
```

New version

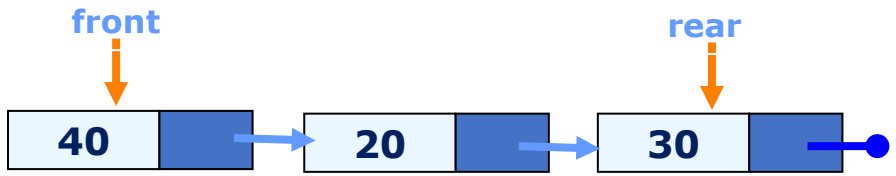



```
def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        data = self.ll.head.data
        self.ll.remove_node(0)
        return data
```

Core stack operations: isEmpty()

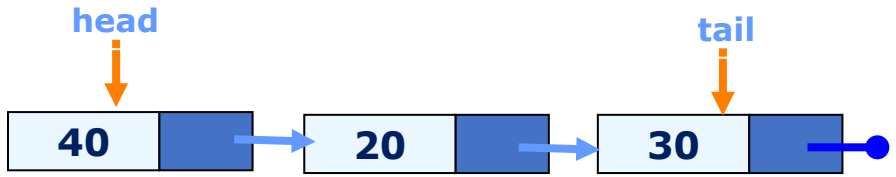
Previous version

```
def isEmpty(self):
    return self.top is None
```



New Version

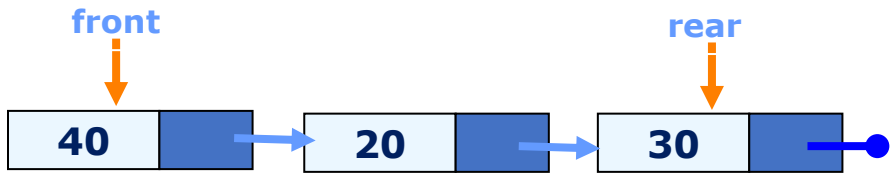
```
def isEmpty(self):
    return self.ll.size == 0
```



Core stack operations: getSize()

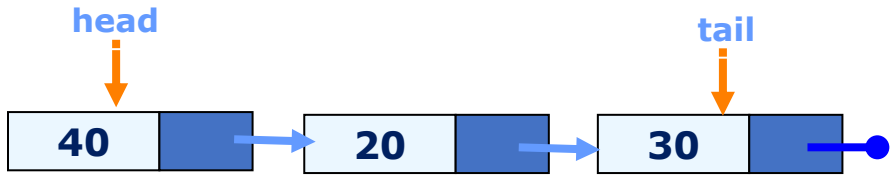
Previous version

```
def getSize(self):
        return self.size
```



New Version

```
def getSize(self):
        return self.ll.size
```



Working Example – 02

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

class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None
        self.tail = None

    def find_node(self, index):
        if index < 0 or index >= self.size:
            return None
        temp = self.head
        for _ in range(index):
            temp = temp.next
        return temp

    def remove_node(self, index):
        if index < 0 or index >= self.size:
            return -1
        if index == 0:
            self.head = self.head.next
            if self.size == 1:
                self.tail = None
        else:
            prev = self.find_node(index - 1)
            prev.next = prev.next.next
            if index == self.size - 1:
                self.tail = prev
        self.size -= 1
        return 0
```

```
    def insert_node(self, index, value):
        if index < 0 or index > self.size:
            return -1
        new_node = ListNode(value)
        if index == 0:
            new_node.next = self.head
            self.head = new_node
            if self.size == 0:
                self.tail = new_node
        elif index == self.size:
            self.tail.next = new_node
            self.tail = new_node
        else:
            prev = self.find_node(index - 1)
            new_node.next = prev.next
            prev.next = new_node
        self.size += 1
        return 0
```

Working Example – 02 (cont.)

```
52  class Queue:
53      def __init__(self):
54          self.ll = LinkedList()
55
56      def enqueue(self, data):
57          self.ll.insert_node(self.ll.size, data)
58
59      def dequeue(self):
60          if self.isEmpty():
61              raise IndexError(“Dequeue from empty queue")
62          data = self.ll.head.data
63          self.ll.remove_node(0)
64          return data
65
66      def getFront(self):
67          if self.isEmpty():
68              raise IndexError("Peek from empty queue")
69          return self.ll.head.data
70
71      def getSize(self):
72          return self.ll.size
73
74      def isEmpty(self):
75          return self.ll.size == 0
```

Working Example – 02 (cont.)

```
73 if __name__ == "__main__":
74
75     queue = Queue()
76
77     queue.enqueue(10)
78     queue.enqueue(20)
79     queue.enqueue(30)
80
81
82     print("Front element before dequeue:", queue.getFront())
83
84     print("Size before dequeue:", queue.getSize())
85
86     print("Dequeued element:", queue.dequeue())
87
88     print("Front element after dequeue:", queue.getFront())
89
90     print("Size after dequeue:", queue.getSize())
```

Stacks and Queues: Working Example – 01

Given a Queue data structure, write a function to reverse its elements using a Stack. The original Queue structure should be modified in-place. You can only use standard Queue operations (enqueue, dequeue) and Stack operations (push, pop).

Stacks and Queues: Working Example – 01

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class Stack:
7     def __init__(self):
8         self.top = None
9         self.size = 0
10
11    def peek(self):
12        if self.is_empty():
13            raise IndexError("Peek: Empty stack")
14        return self.top.data
15
16    def push(self, data):
17        new_node = Node(data)
18        new_node.next = self.top
19        self.top = new_node
20        self.size += 1
```

```
21    def pop(self):
22        if self.is_empty():
23            raise IndexError("Pop: Empty stack")
24        popped_node = self.top
25        self.top = self.top.next
26        self.size -= 1
27        return popped_node.data
28
29    def is_empty(self):
30        return self.top is None
31
32    def get_size(self):
33        return self.size
```

Stacks and Queues: Working Example – 01 (cont.)

```
1 class Queue:
2     def __init__(self):
3         self.front = None
4         self.rear = None
5         self.size = 0
6
7     def isEmpty(self):
8         return self.front is None
9
10    def enqueue(self, data):
11        new_node = Node(data)
12        if self.isEmpty():
13            self.front = new_node
14        else:
15            self.rear.next = new_node
16        self.rear = new_node
17        self.size += 1
18
19
20
21
22
```

```
23    def dequeue(self):
24        if self.isEmpty():
25            raise IndexError("Dequeue from empty queue")
26        dequeued_node = self.front
27        self.front = self.front.next
28        if self.front is None:
29            self.rear = None
30        self.size -= 1
31        return dequeued_node.data
32
33    def getFront(self):
34        if self.isEmpty():
35            raise IndexError("Peek from empty queue")
36        return self.front.data
37
38    def getSize(self):
39        return self.size
40
```

Stacks and Queues: Working Example – 01

```
1       def getSize(self):
2             return self.size
3
4           def printQueue(self):
5            if self.isEmpty():
6                print("Queue is empty")
7                return
8            current = self.front
9            while current:
10               print(current.data, end=" ")
11               current = current.next
12           print()  # New line
13
14 def reverse_queue(queue):
15    stack = Stack()
16
17    # Push all elements from queue to stack
18    while not queue.isEmpty():
19        stack.push(queue.dequeue())
20
21    # Pop from stack and enqueue back to queue
22    while not stack.is_empty():
23        queue.enqueue(stack.pop())
24
25    return queue
```

```
23 if __name__ == "__main__":
24    queue = Queue()
25
26    # Enqueue three values
27    queue.enqueue(10)
28    queue.enqueue(20)
29    queue.enqueue(30)
30
31    # Print original queue
32    print("Original Queue:", end=" ")
33    queue.printQueue()    # Print: 10 20 30
34
35    # Reverse the queue
36    queue = reverse_queue(queue)
37
38    # Print reversed queue
39    print("Reversed Queue:", end=" ")
40    queue.printQueue()    # Print: 30 20 10
```

Stacks and Queues: Working Example – 02

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

class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None
        self.tail = None

    def find_node(self, index):
        if index < 0 or index >= self.size:
            return None
        temp = self.head
        for _ in range(index):
            temp = temp.next
        return temp

    def remove_node(self, index):
        if index < 0 or index >= self.size:
            return -1
        if index == 0:
            self.head = self.head.next
            if self.size == 1:
                self.tail = None
        else:
            prev = self.find_node(index - 1)
            prev.next = prev.next.next
            if index == self.size - 1:
                self.tail = prev
        self.size -= 1
        return 0
```

```
    def insert_node(self, index, value):
        if index < 0 or index > self.size:
            return -1
        new_node = ListNode(value)
        if index == 0:
            new_node.next = self.head
            self.head = new_node
            if self.size == 0:
                self.tail = new_node
        elif index == self.size:
            self.tail.next = new_node
            self.tail = new_node
        else:
            prev = self.find_node(index - 1)
            new_node.next = prev.next
            prev.next = new_node
        self.size += 1
        return 0
```

Stacks and Queues: Working Example – 02 (cont.)

```
52 class Stack:
53     def __init__(self):
54         self.ll = LinkedList()
55
56     def push(self, data):
57         self.ll.insert_node(0, data)
58
59     def pop(self):
60         if self.isEmpty():
61             raise IndexError("Empty Stack")
62         data = self.ll.head.data
63         self.ll.remove_node(0)
64         return data
65
66     def peek(self):
67         if self.isEmpty():
68             raise IndexError("Empty Stack")
69         return self.ll.head.data
70
71     def isEmpty(self):
72         return self.ll.size == 0
73
74     def get_size(self):
75         return self.ll.size
```

```
76 class Queue:
77     def __init__(self):
78         self.ll = LinkedList()
79
80     def enqueue(self, data):
81         self.ll.insert_node(self.ll.size, data)
82
83     def dequeue(self):
84         if self.isEmpty():
84             raise IndexError("Dequeue from empty queue")
86         data = self.ll.head.data
87         self.ll.remove_node(0)
88         return data
89
90     def getFront(self):
91         if self.isEmpty():
92             raise IndexError("Peek from empty queue")
93         return self.ll.head.data
94
95     def getSize(self):
96         return self.ll.size
97
98     def isEmpty(self):
99         return self.ll.size == 0
```

Stacks and Queues: Working Example – 02 (cont.)

```
100     def printQueue(self):
101             if self.isEmpty():
102                 print("Queue is empty")
103                 return
104             current = self.ll.head
105             while current:
106                 print(current.data, end=" ")
107                 current = current.next
108             print()
109
110 def reverse_queue(queue):
111     stack = Stack()
112
113     # Push all elements from queue to stack
114     while not queue.isEmpty():
115         stack.push(queue.dequeue())
116
117     # Pop from stack and enqueue back to queue
118     while not stack.is_empty():
119         queue.enqueue(stack.pop())
120
121     return queue
```

```
122 if __name__ == "__main__":
123     queue = Queue()
124
125     # Enqueue three values
126     queue.enqueue(10)
127     queue.enqueue(20)
128     queue.enqueue(30)
129
130     # Print original queue
131     print("Original Queue:", end=" ")
132     queue.printQueue()     # Print: 10 20 30
133
134     # Reverse the queue
135     queue = reverse_queue(queue)
136
137     # Print reversed queue
138     print("Reversed Queue:", end=" ")
139     queue.printQueue()     # Print: 30 20 10
```

Queues in computer science

- **Operating systems:**
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- **Programming:**
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- **Real world examples:**
  - people on an escalator or waiting in a line
  - cars at a gas station

Uses of Queues

Queues are versatile data structures with various use cases across different domains. Here are some common scenarios where queues are particularly useful:

1.Task Scheduling:

- **Operating Systems**: Queues manage processes in CPU scheduling and task management, where processes are executed in the order they arrive (FIFO).
- **Print Spooling**: Print jobs are managed in a queue, allowing them to be processed in the order they were submitted.

2.Breadth-First Search (BFS):

- In graph traversal algorithms like BFS, queues are used to explore nodes level by level. Nodes are added to the queue as they are discovered and removed as they are processed.

Uses of Queues

3. **Data Buffering**:
   - **Streaming Data**: Queues can buffer data streams (e.g., video/audio streaming), allowing data to be processed as it arrives.
   - **Network Data Packets**: Queues help manage incoming network packets, ensuring they are processed in the order they are received.

4.Event Management:

- In GUI applications, events (like user interactions) can be managed using queues, allowing them to be handled in the order they occur.

Customer Service Call Center

Imagine a customer service call center for a large company. When customers call in, they are placed in a queue to wait for the next available representative. This system perfectly illustrates the First In, First Out (FIFO) principle of queues.

Customer Service Call Center (cont.)

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

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0

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

    def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1

     def getSize(self):
            return self.size
```

```
    def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        dequeued_node = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self.size -= 1
        return dequeued_node.data

    def getFront(self):
        if self.isEmpty():
            raise IndexError("Peek from empty queue")
        return self.front.data

    def displayQueue(self):
        current = self.front
        while current:
            print(current.data)
            current = current.next
```

Customer Service Call Center (cont.)

```
46 if __name__ == "__main__":
47
48     queue = Queue()
49
50     queue.enqueue("Customer 1")
51     queue.enqueue("Customer 2")
52     queue.enqueue("Customer 3")
53     print("\nInitial queue:")
54     queue.displayQueue()
55
56     queue.dequeue()
57     print("\nAfter dequeue:")
58     queue.displayQueue()
59
60     queue.enqueue("Customer 4")
61     print("\nAfter adding Customer 4:")
62     queue.displayQueue()
63
64     print("\nFront of queue:", queue.getFront())
```