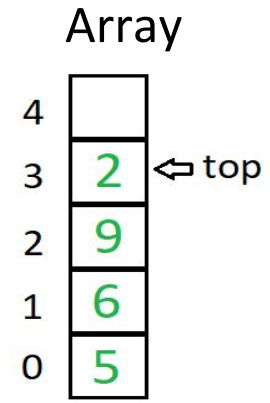
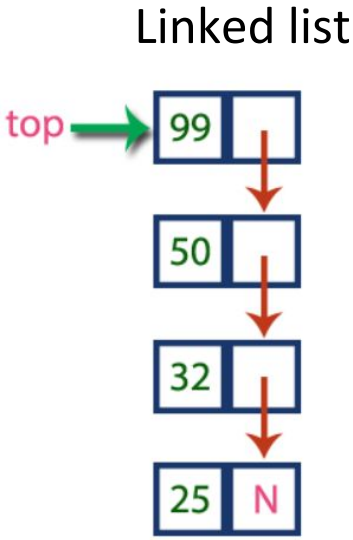
Stack and Queue ADT using Linked list

Stack using Linked list

- Stack is a data structure which follows **LIFO** i.e. Last-In-First-Out method.
- The data/element which is stored last in the stack i.e. the element at **top will be accessed first**.
- And both **insertion & deletion takes place at the top**.
- Stack can be implemented with arrays or linked list.
- The major problem with the stack implemented using an array is, it works only for a fixed number of data values.
- The stack implemented using linked list can work for an unlimited number of values.
- A linked stack may be implemented using the linked list allows appropriate insertions and deletions in **O**(1) time.

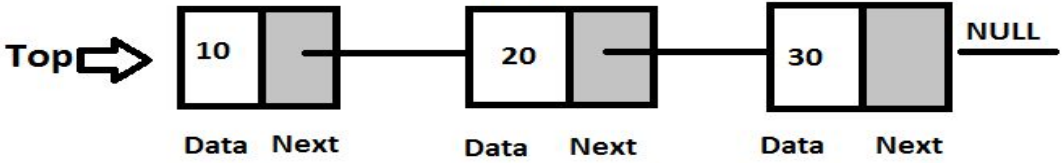


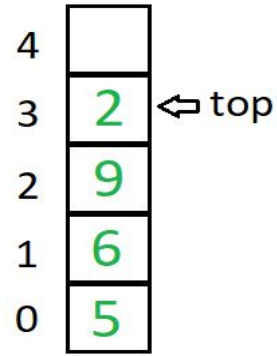


Stack using Linked list

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

- In stack Implementation, a stack contains a top pointer. which is “top” of the stack where pushing and popping items happens at the top of the list.
- With Linked list, the **push** operation can be replaced by the addAtFront() method of linked list and **pop** operation can be replaced by a function which deletes the front node of the linked list.





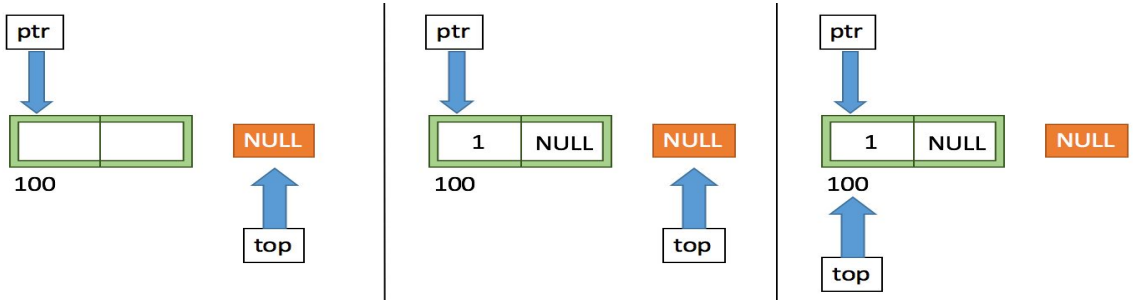
Stack using Linked list

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

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

Push operation

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

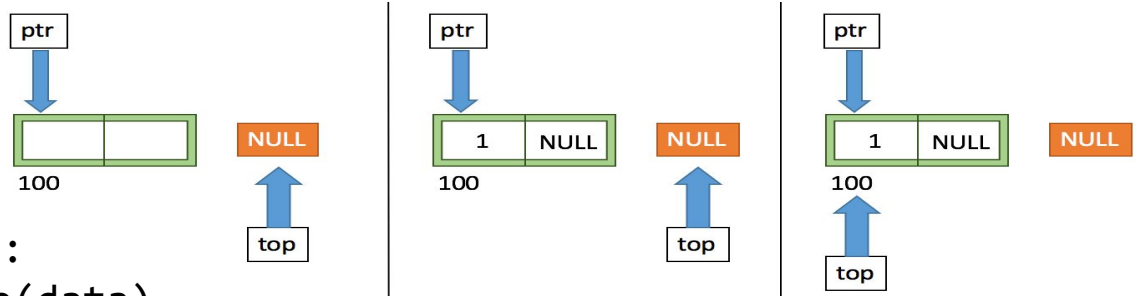
- Create a node first and assign data to it.
- If the list is empty then the item is to be pushed as the start node of the list.
- If there are some nodes in the list already, then we have to add the new element in the beginning of the list



Push operation

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

- Create a node first and assign data to it.
- If the list is empty then the item is to be pushed as the start node of the list.
- If there are some nodes in the list already, then we have to add the new element in the beginning of the list

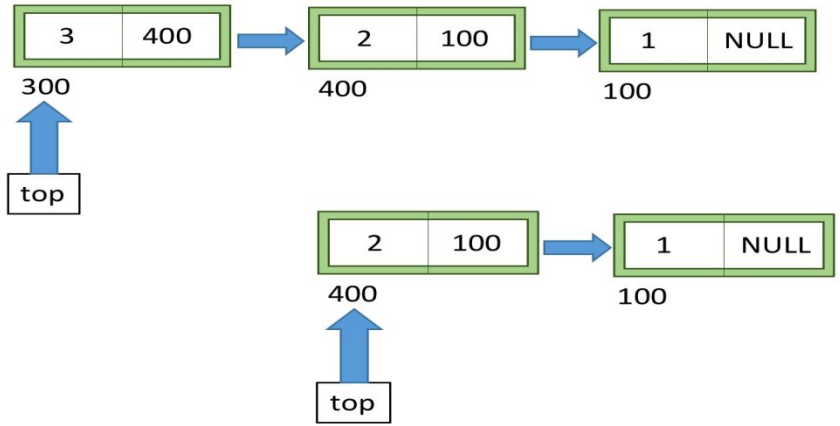



```
def push(self, data):
        if self.head == None:
            self.head = Node(data)
        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode
```

Pop operation

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

- Check for the underflow condition: The stack will be empty if the top pointer of the list points to null.
- In stack, the elements are popped only from one end, therefore, the value stored in the top pointer must be deleted and the node must be freed. The next node of the top node now becomes the top node.

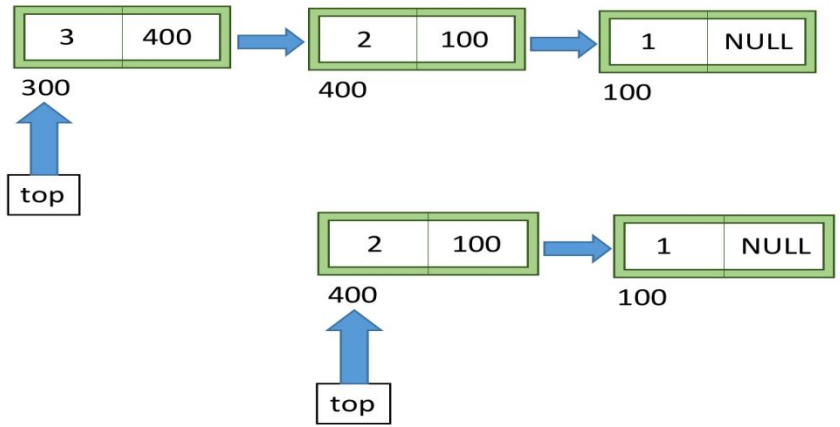


Pop operation

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

- Check for the underflow condition: The stack will be empty if the top pointer of the list points to null.
- In stack, the elements are popped only from one end, therefore, the value stored in the top pointer must be deleted and the node must be freed. The next node of the top node now becomes the top node.

```
def pop(self):
        if self.isempty():
            return None
         else:
            poppednode = self.head
            self.head = self.head.next
            poppednode.next = None
            return poppednode.data
```



Display the nodes (Traversing)

- Displaying all the nodes of a stack needs traversing all the nodes of the stack. For this purpose, we need to follow the following steps.

– **Copy the top pointer into a temporary pointer.**

– **Move the temporary pointer through all the nodes of the stack and print the value field attached to every node.**

```
def display(self):
        iternode = self.head
       if self.isempty():
           print("Stack Underflow")
        else:
            while(iternode != None):
                print(iternode.data, "->", end=" ")
               iternode = iternode.next
           return
```

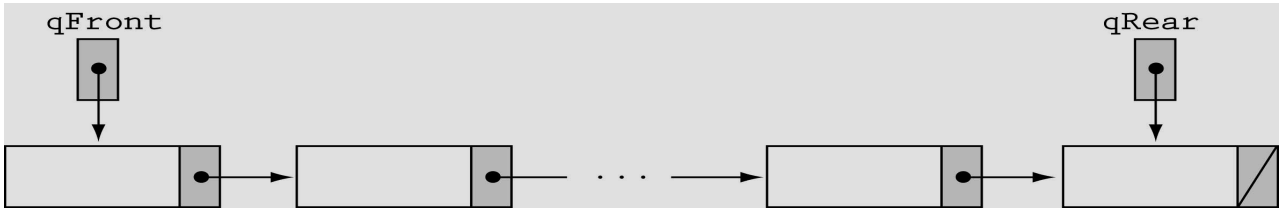
```
class Node:
        def __init__(self,data):
                self.data = data
                self.next = None
```

Queue using linked list

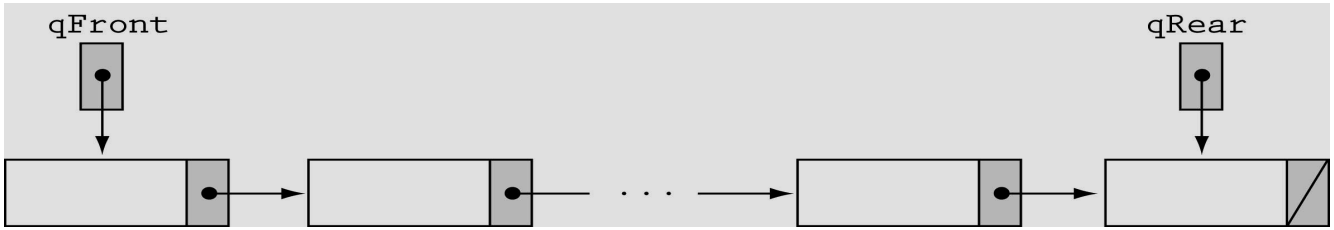
- Queue is a linear data structure which follows **FIFO** i.e. First-In-First-Out method.
- The two ends of a queue are called Front and Rear, where **Insertion always takes place at the Rear and the elements are accessed or removed from the Front**.
- While implementing queues using arrays:
  - We cannot increase the size of array, if we have more elements to insert than the capacity of array.
  - If we create a very large array, we will be wasting a lot of memory space.
- Therefore if we implement Queue using Linked list we can solve these problems, as in Linked list Nodes are created dynamically as an when required.

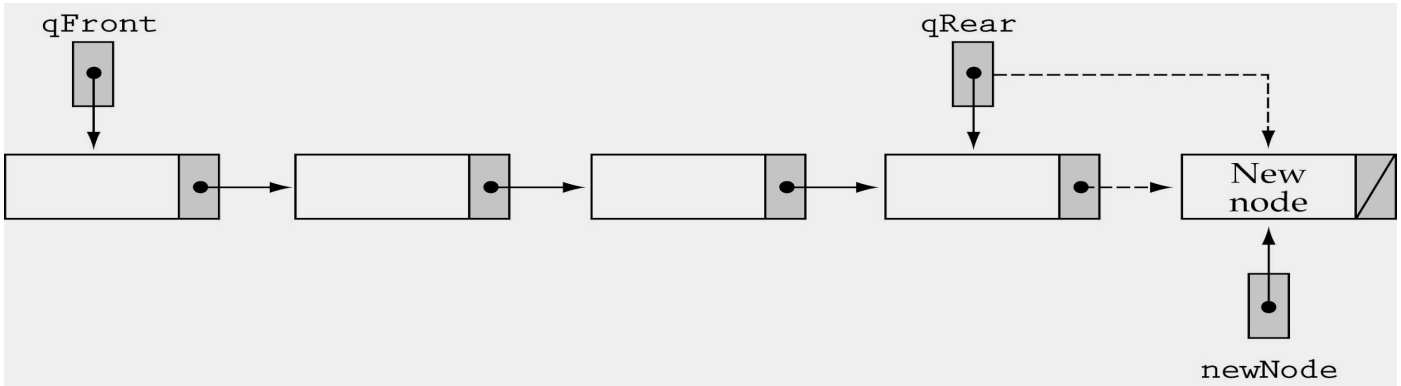
Implementing queues using linked lists

- Allocate memory for each new element dynamically
- Link the queue elements together
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue.



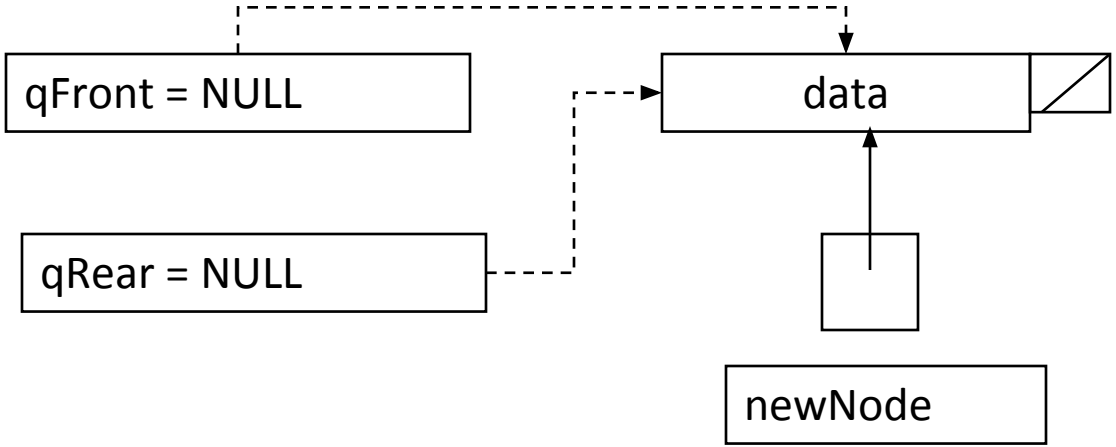
Enqueuing (non-empty queue)





Enqueuing (empty queue)

- We need to make *qFront* point to the new node also



Queue using Linked list

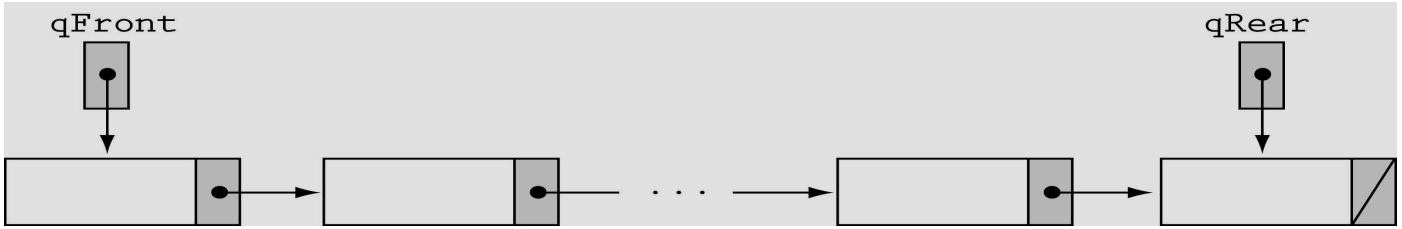
```
class Node:
  def __init__(self,data):
    self.data = data
    self.next = None

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

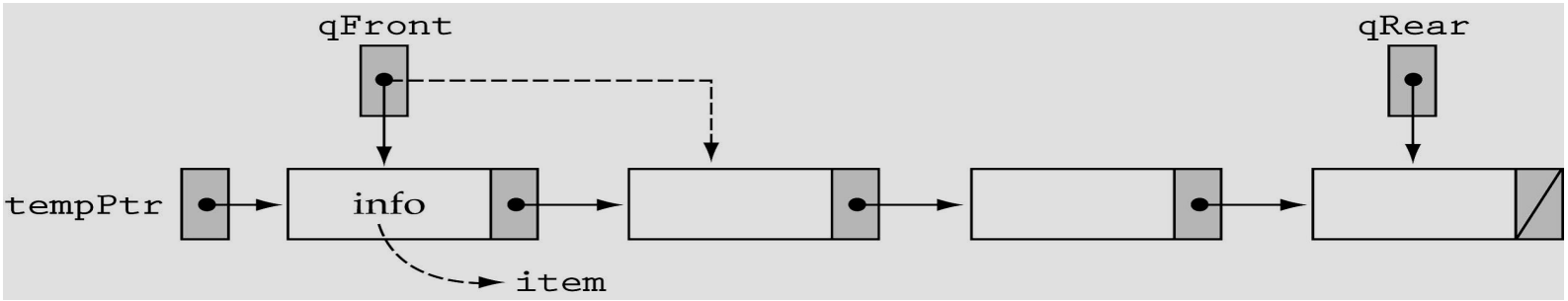
Enqueue operation

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

```
def enqueue ( data)
    newnode = Node(data)
    if  front == None and rear == None:
        front = newnode;
        rear = newnode;
    else
        rear ->next = newnode;
        rear = newnode;
```

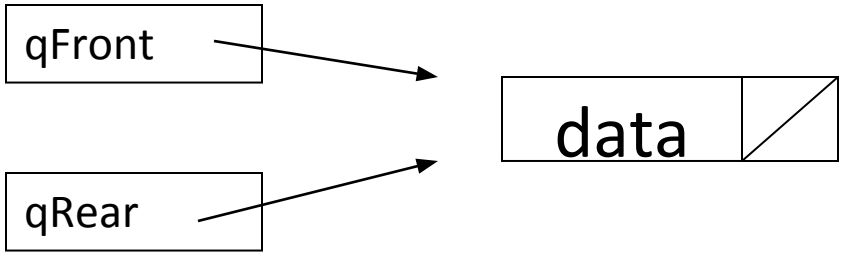
Dequeueing (the queue contains more than one element)





Dequeue
(the queue contains only one element)

- We need to reset *qRear and qFront* to NULL also



After dequeue:

qFront = NULL

qRear = NULL

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
```

Dequeue operation

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

```
def dequeue ()
        if  front == None:
                print("queue is empty")
        elif front ==rear:
                element = front.data
                front = rear = None
        else:
                element = front.data
                front = front.next;
        return element
```

Display the nodes (Traversing)

```
def display ()
    if  front == None:
        print("queue is empty")
        return
    p= front
    while(p !=None):
        print(p.data)
        p = p.next
```

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