

Rotate

In this lesson, we will learn how to rotate a linked list.

WE'LL COVER THE FOLLOWING ^

- Algorithm
- Implementation
- Explanation

In this lesson, we investigate how to rotate the nodes of a singly linked list around a specified pivot element. This implies shifting or rotating everything that follows the pivot node to the front of the linked list.

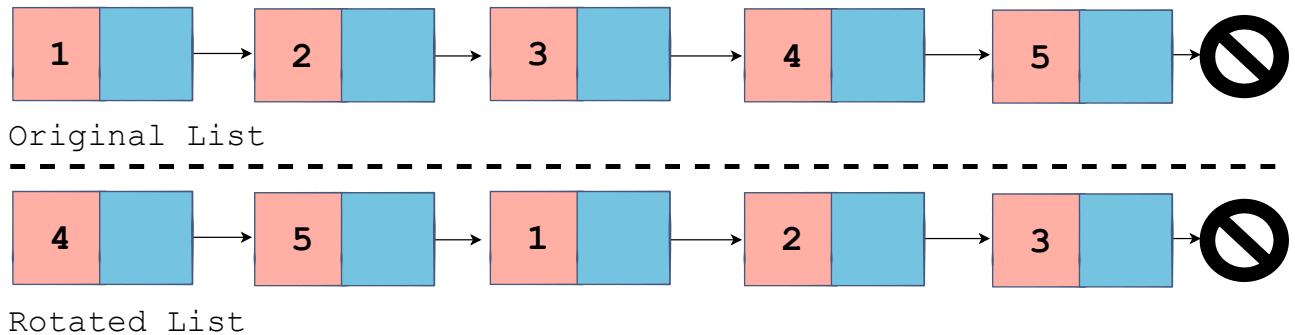
The illustration below will help you understand how to rotate the nodes of a singly linked list.

Singly Linked List: Rotate



Let's rotate the linked list around pivot = 3.

Singly Linked List: Rotate



The linked list is rotated by pivot = 3.

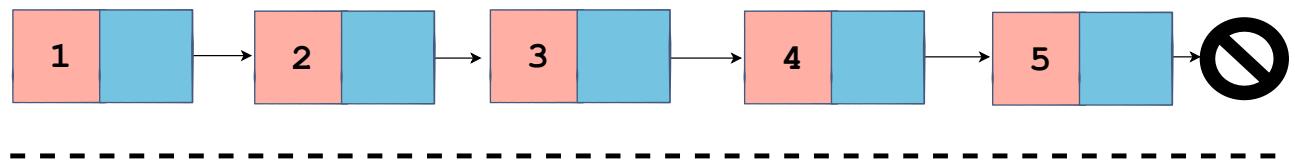
2 of 2



Algorithm

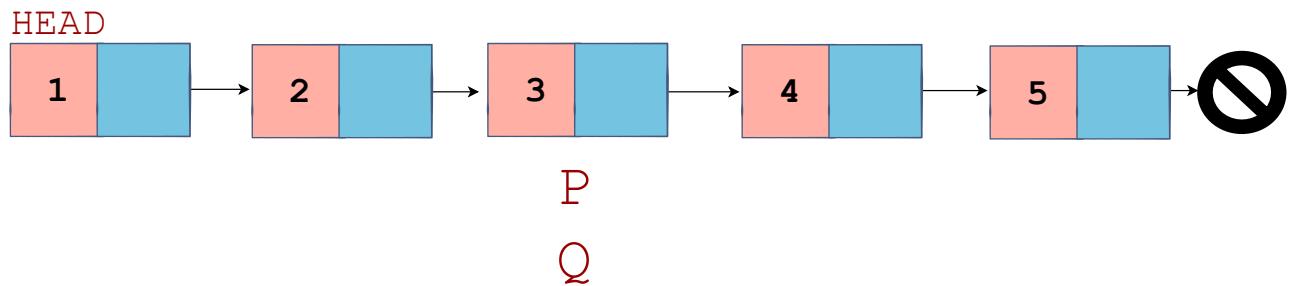
The algorithm to solve this problem has been illustrated below:

Singly Linked List: Rotate



1 of 6

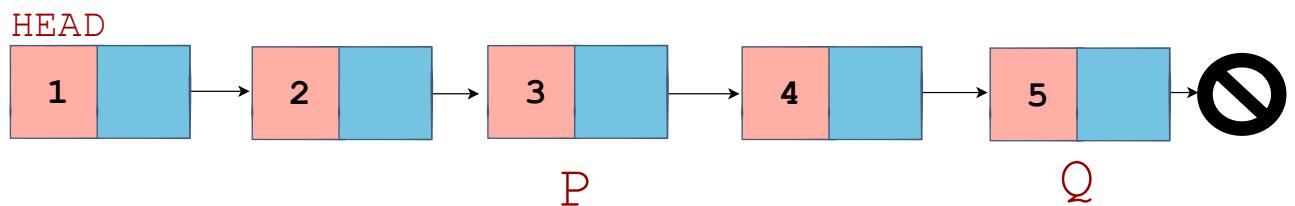
Singly Linked List: Rotate



pivot = 3

2 of 6

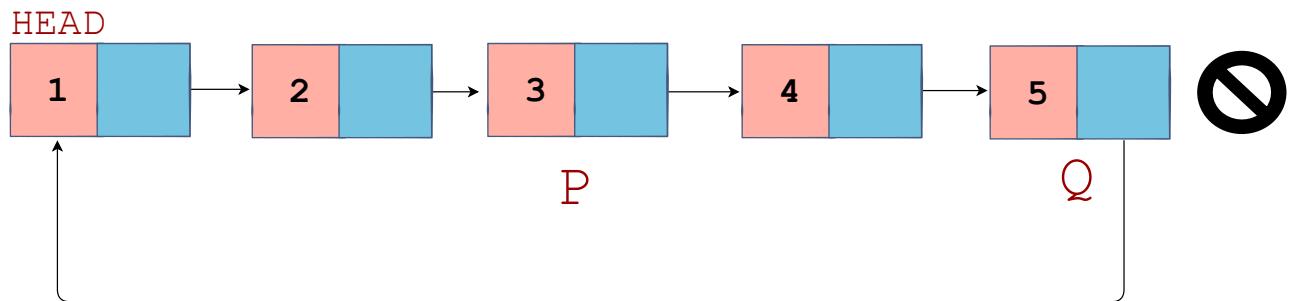
Singly Linked List: Rotate



pivot = 3

3 of 6

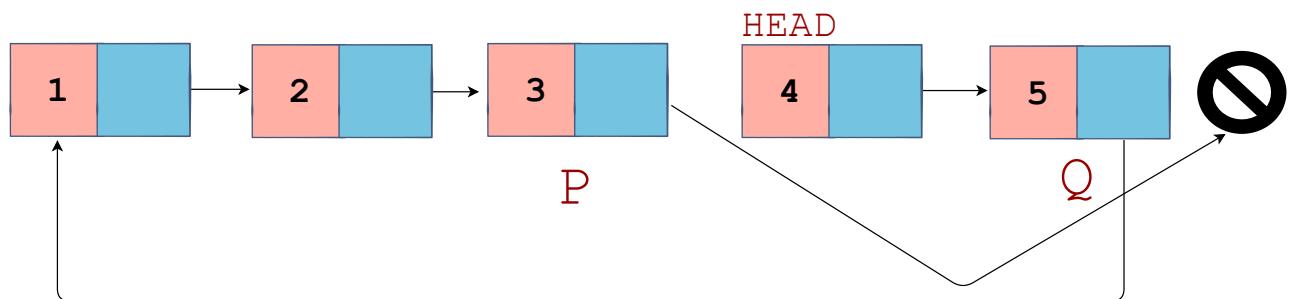
Singly Linked List: Rotate



pivot = 3

4 of 6

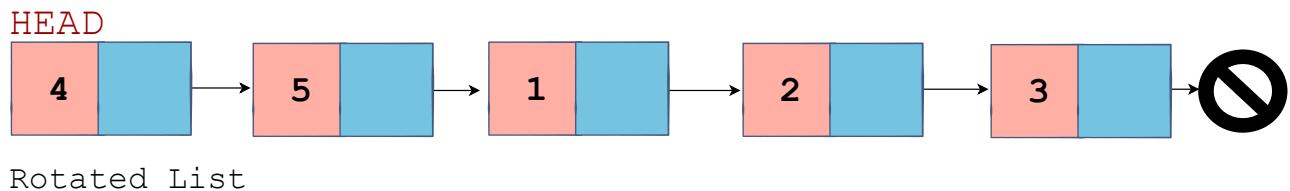
Singly Linked List: Rotate



pivot = 3

5 of 6

Singly Linked List: Rotate



6 of 6



As you can see from the illustrations above, we make use of two pointers `p` and `q`. `p` points to the pivot node while `q` points to the end of the linked list. Once the pointers are rightly positioned, we update the last element, and instead of making it point to `None`, we make it point to the head of the linked list. After this step, we achieve a circular linked list. Now we have to fix the end of the linked list. Therefore, we update the head of the linked list, which will be the next element after the pivot node, as the pivot node has to be the last node. Finally, we set `p.next` to `None` which breaks up the circular linked list and makes `p` (pivot node) the last element of our rotated linked list.

Implementation

Now that you are familiar with the algorithm, let's start with the implementation of the algorithm.

```
def rotate(self, k):
    if self.head and self.head.next:
        p = self.head
        q = self.head
        prev = None
        count = 0

        while p and count < k:
```



```
prev = p
p = p.next
q = q.next

    count += 1
p = prev
while q:
    prev = q
    q = q.next
q = prev

q.next = self.head
self.head = p.next
p.next = None
```

rotate(self, k)

Explanation

The `rotate` method takes in `self` and `k` as input parameters. We want to rotate our linked list around the `k` th element. We only rotate a linked list if it's not empty or contains more than one element as there is no point in rotating a single element. Therefore, on **line 2**, it is ensured that the execution proceeds to **line 3** if both `self.head` and `self.head.next` are not `None`. Otherwise, we return from the method.

Three variables `p`, `q`, and `prev` are initialized on **lines 3-5**. `p` and `q` are initialized to `self.head` while `prev` is initialized to `None`. We also declare another variable `count` on **line 6** and initialize it to `0`. `count` will help us in making `p` and `q` point to the nodes specified in the algorithm. Therefore, in the `while` loop starting from **line 8**, we move `p` and `q` along the linked list by updating them to their next nodes (**lines 10-11**). `count` is incremented by `1` in each iteration of the loop until it becomes greater than or equal to `k` in which case we'll break out of the `while` loop (**line 12**). In each iteration, `prev` is also being updated as it is set to `p` before `p` updates itself to its next node (**line 9**). Additionally, we keep a check if `p` does not equal to `None`, in which case we'll also break out of the `while` loop.

After breaking out of the `while` loop, we set `prev` equal to `p` on **line 13** to position our first pointer correctly. For the second pointer (`q`), we have to make it point to the last element in the linked list. As a result, we set up another `while` loop on **lines 14-16** where we move `q` along with the linked list until it becomes `None`. We also keep track of the previous node (`prev`) and make `q` point to the last element in the linked list on **line 17**.

Now that we have positioned the two pointers according to our algorithm, we'll execute the other simple steps. First, the last element (`q`) has to point to the first element of the linked list (`head`). So we make `q.next` set to `self.head` on **line 19**. Second, we now need to update the `head` as shown in the slides. On **line 20**, we update `self.head` to the next element of `p`. Lastly, as our pivot node (`p`) will become the end of the linked list, it should point to `None`. Therefore, we update `p.next` to `None` on **line 21**. That completes our implementation for the `rotate` method.

Below is the code widget which contains all the code we have written so far, along with the `rotate` method. Verify and test the code below:

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

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

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next

    def append(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def prepend(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node

    def insert_after_node(self, prev_node, data):

        if not prev_node:
            print("Previous node does not exist.")
            return

        new_node = Node(data)
```

```
new_node.next = prev_node.next
prev_node.next = new_node

def delete_node(self, key):

    cur_node = self.head

    if cur_node and cur_node.data == key:
        self.head = cur_node.next
        cur_node = None
        return

    prev = None
    while cur_node and cur_node.data != key:
        prev = cur_node
        cur_node = cur_node.next

    if cur_node is None:
        return

    prev.next = cur_node.next
    cur_node = None

def delete_node_at_pos(self, pos):
    if self.head:
        cur_node = self.head

        if pos == 0:
            self.head = cur_node.next
            cur_node = None
            return

        prev = None
        count = 1
        while cur_node and count != pos:
            prev = cur_node
            cur_node = cur_node.next
            count += 1

        if cur_node is None:
            return

        prev.next = cur_node.next
        cur_node = None

def len_iterative(self):

    count = 0
    cur_node = self.head

    while cur_node:
        count += 1
        cur_node = cur_node.next
    return count

def len_recursive(self, node):
    if node is None:
        return 0
    return 1 + self.len_recursive(node.next)

def swap_nodes(self, key_1, key_2):
```

```

if key_1 == key_2:
    return

prev_1 = None
curr_1 = self.head
while curr_1 and curr_1.data != key_1:
    prev_1 = curr_1
    curr_1 = curr_1.next

prev_2 = None
curr_2 = self.head
while curr_2 and curr_2.data != key_2:
    prev_2 = curr_2
    curr_2 = curr_2.next

if not curr_1 or not curr_2:
    return

if prev_1:
    prev_1.next = curr_2
else:
    self.head = curr_2

if prev_2:
    prev_2.next = curr_1
else:
    self.head = curr_1

curr_1.next, curr_2.next = curr_2.next, curr_1.next

def print_helper(self, node, name):
    if node is None:
        print(name + ": None")
    else:
        print(name + ":" + node.data)

def reverse_iterative(self):

    prev = None
    cur = self.head
    while cur:
        nxt = cur.next
        cur.next = prev

        self.print_helper(prev, "PREV")
        self.print_helper(cur, "CUR")
        self.print_helper(nxt, "NXT")
        print("\n")

        prev = cur
        cur = nxt
    self.head = prev

def reverse_recursive(self):

    def _reverse_recursive(cur, prev):
        if not cur:
            return prev

        nxt = cur.next
        cur.next = prev
        prev = cur

```

```

        cur = nxt
        return _reverse_recursive(cur, prev)

    self.head = _reverse_recursive(cur=self.head, prev=None)

def merge_sorted(self, llist):

    p = self.head
    q = llist.head
    s = None

    if not p:
        return q
    if not q:
        return p

    if p and q:
        if p.data <= q.data:
            s = p
            p = s.next
        else:
            s = q
            q = s.next
        new_head = s
        while p and q:
            if p.data <= q.data:
                s.next = p
                s = p
                p = s.next
            else:
                s.next = q
                s = q
                q = s.next
        if not p:
            s.next = q
        if not q:
            s.next = p
        return new_head

def remove_duplicates(self):

    cur = self.head
    prev = None

    dup_values = dict()

    while cur:
        if cur.data in dup_values:
            # Remove node:
            prev.next = cur.next
            cur = None
        else:
            # Have not encountered element before.
            dup_values[cur.data] = 1
            prev = cur
        cur = prev.next

def print_nth_from_last(self, n, method):
    if method == 1:
        #Method 1:
        total_len = self.len_iterative()
        cur = self.head

```

```

while cur:
    if total_len == n:
        #print(cur.data)
        return cur.data
    total_len -= 1
    cur = cur.next
if cur is None:
    return

elif method == 2:
    # Method 2:
    p = self.head
    q = self.head

    count = 0
    while q:
        count += 1
        if(count>=n):
            break
        q = q.next

    if not q:
        print(str(n) + " is greater than the number of nodes in list.")
        return

    while p and q.next:
        p = p.next
        q = q.next
    return p.data

def rotate(self, k):
    if self.head and self.head.next:
        p = self.head
        q = self.head
        prev = None
        count = 0

        while p and count < k:
            prev = p
            p = p.next
            q = q.next
            count += 1
        p = prev
        while q:
            prev = q
            q = q.next
        q = prev

        q.next = self.head
        self.head = p.next
        p.next = None

llist = LinkedList()
llist.append(1)
llist.append(2)
llist.append(3)
llist.append(4)
llist.append(5)
llist.append(6)

```

```
llist.rotate(4)  
llist.print_list()
```



In this lesson, we looked at how to rotate a linked list about a pivot node. I hope you were able to understand the algorithm and the code. I'll see you in the next lesson with another exciting challenge. Stay tuned!