Merge Two Sorted Linked Lists

In this lesson, we will learn how to merge two sorted linked lists.

WE'LL COVER THE FOLLOWING

If we are given two already sorted linked lists, how do we make them into one linked list while keeping the final linked list sorted as well? Let's find out in this lesson.

Before getting started, we'll make the following assumption:

Each of the sorted linked lists will contain at least one element.

A related problem is to create a third linked list which is also sorted. In this lesson, the two linked lists given will no longer be available in their original form, and only one linked list which includes both their nodes will remain. Let's get started.

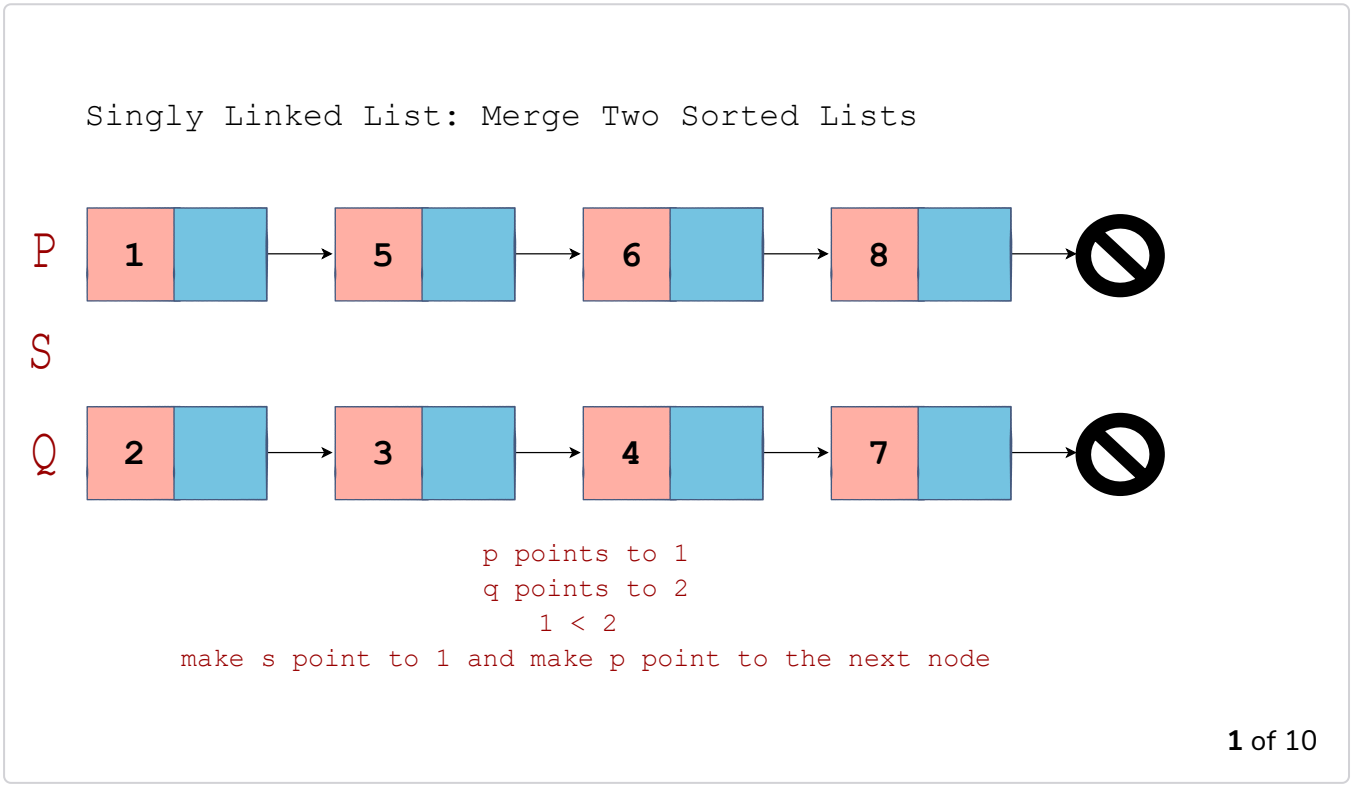
First of all, we'll have a look at the algorithm which we'll use to code and then we'll analyze the implementation in Python.

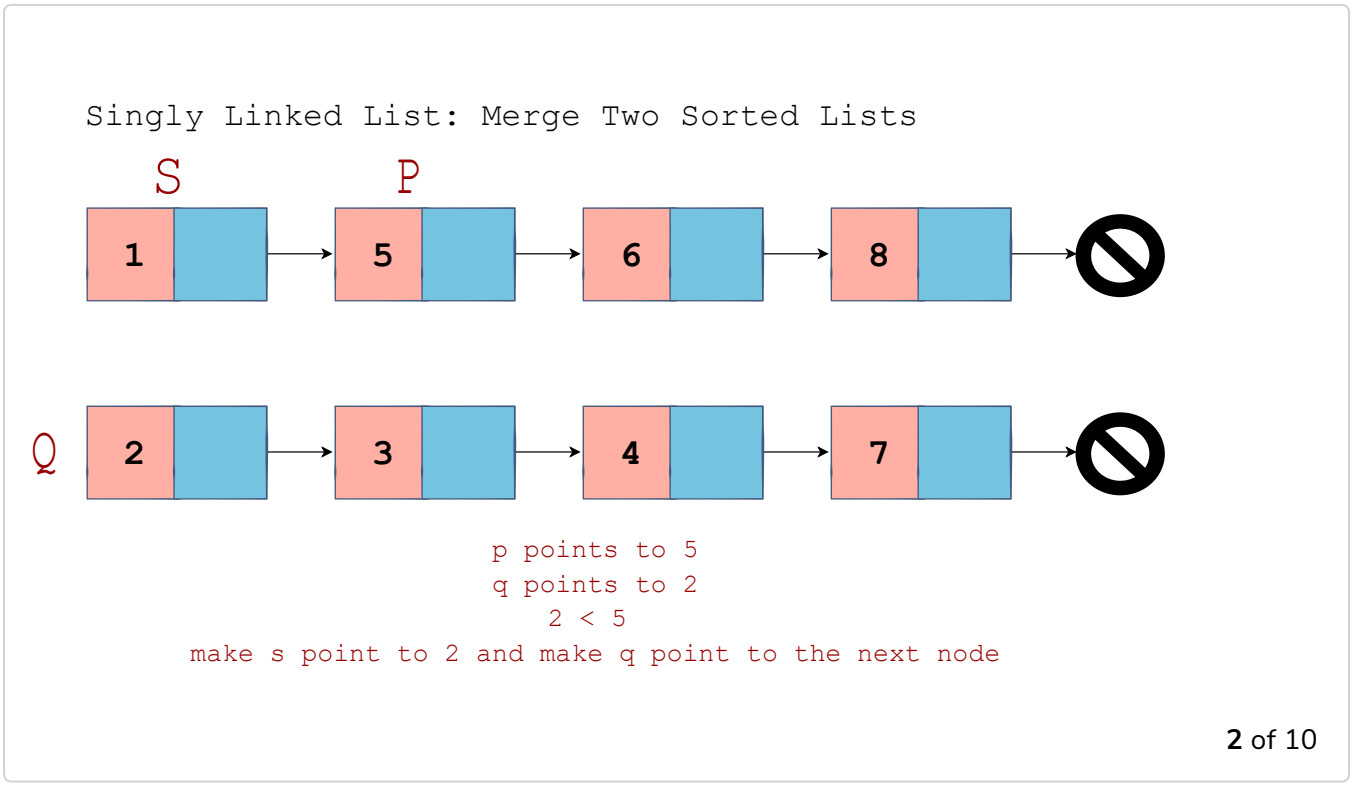
Algorithm

To solve this problem, we'll use two pointers (`p` and `q`) which will each initially point to the head node of each linked list. There will be another pointer, `s`, that will point to the smaller value of data of the nodes that `p` and `q` are pointing to. Once `s` points to the smaller value of the data of nodes that `p` and `q` point to, `p` or `q` will move on to the next node in their respective linked list. If `s` and `p` point to the same node, `p` moves forward; otherwise `q` moves forward. The final merged linked list will be made from the nodes that

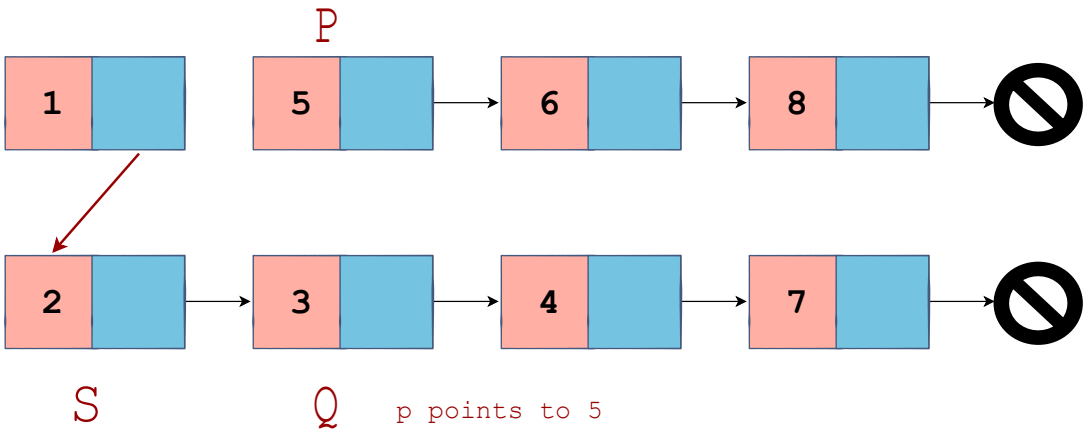
moves forward. The final merged linked list will be made from the nodes that

`s` keeps pointing to. To get a clearer picture, let's look at the illustration below:



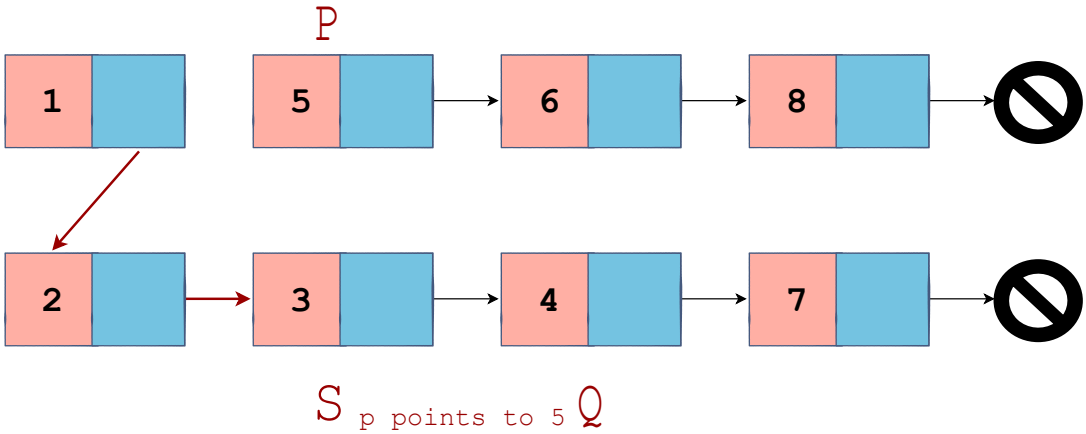


Singly Linked List: Merge Two Sorted Lists



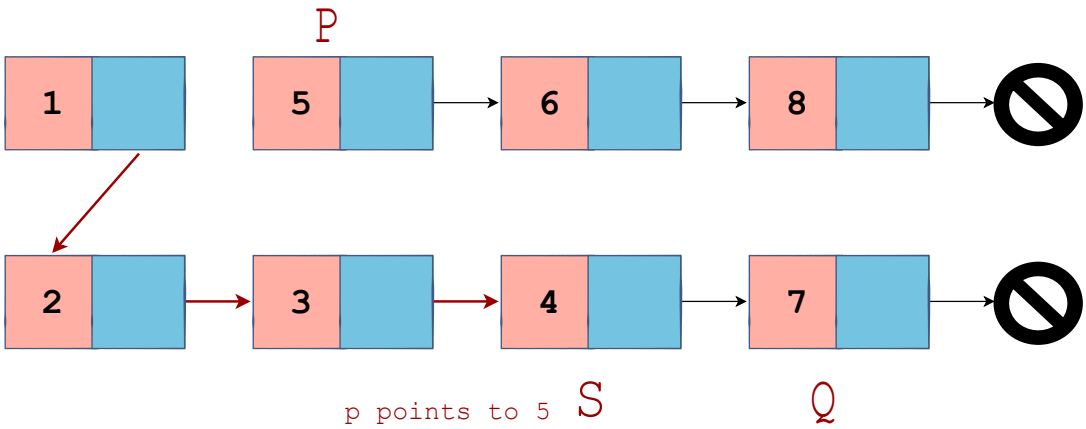
p points to 5
q points to 3
3 < 5
make s point to 3 and make q point to the next node

Singly Linked List: Merge Two Sorted Lists



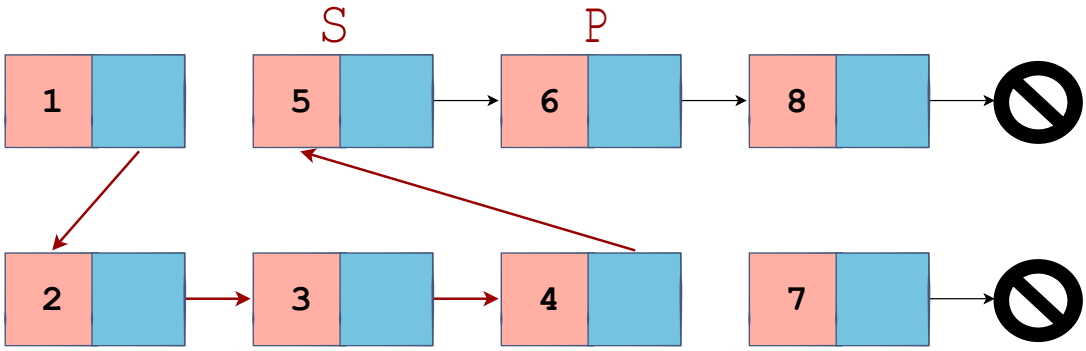
p points to 5
q points to 4
4 < 5
make s point to 4 and make q point to the next node

Singly Linked List: Merge Two Sorted Lists



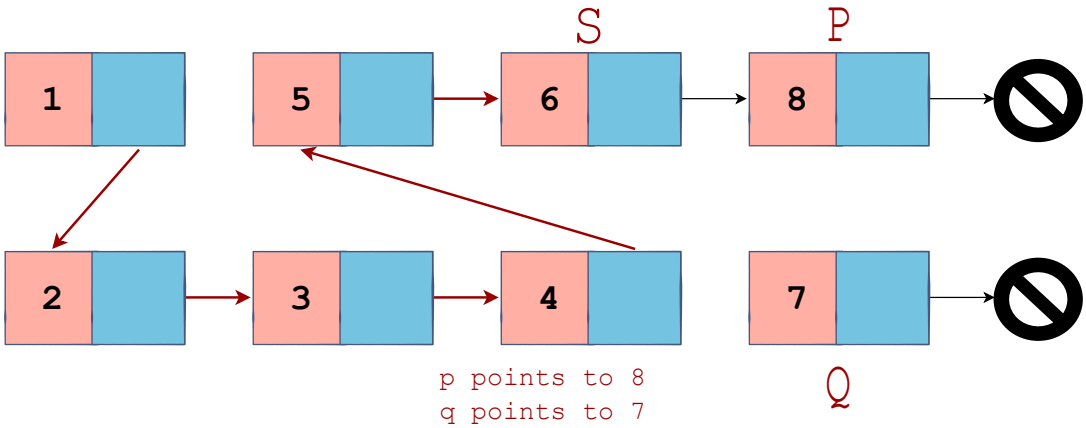
p points to 5
q points to 7
5 < 7
make s point to 5 and make p point to the next node

Singly Linked List: Merge Two Sorted Lists



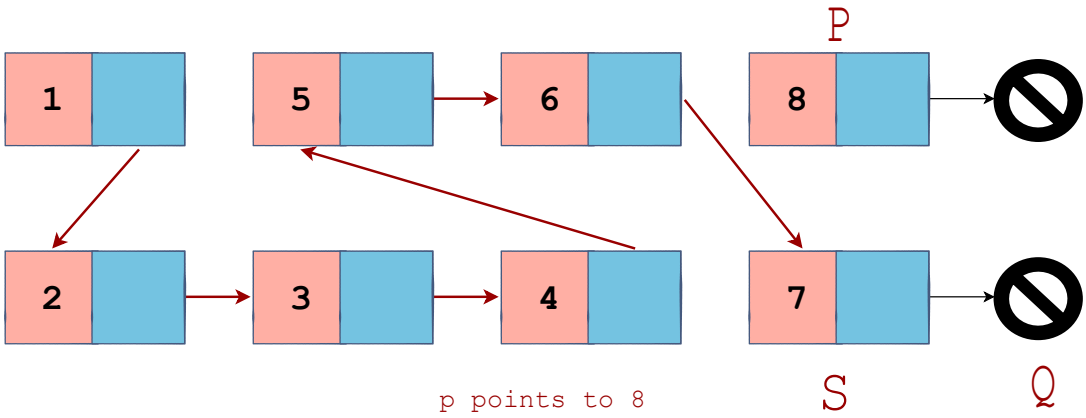
p points to 6
q points to 7
6 < 7
make s point to 6 and make p point to the next node

Singly Linked List: Merge Two Sorted Lists

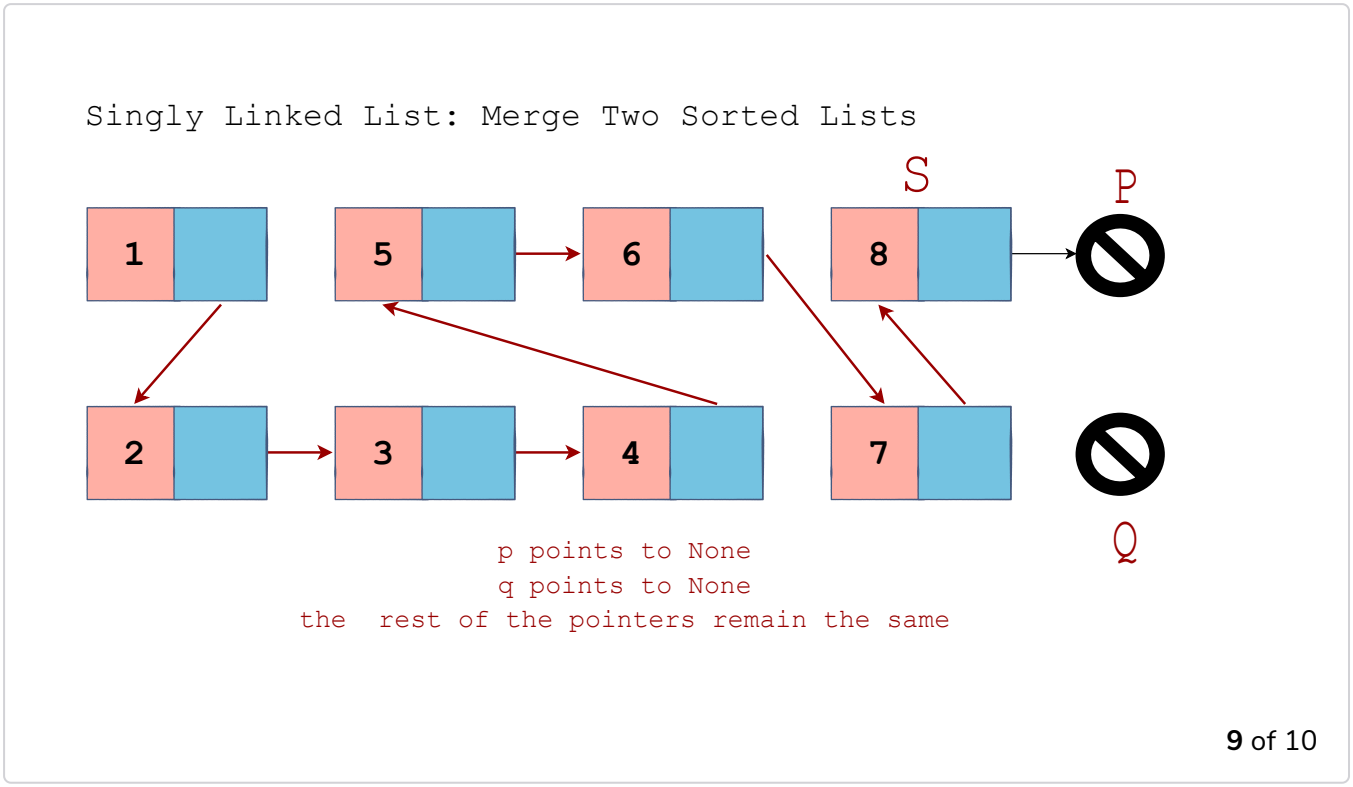


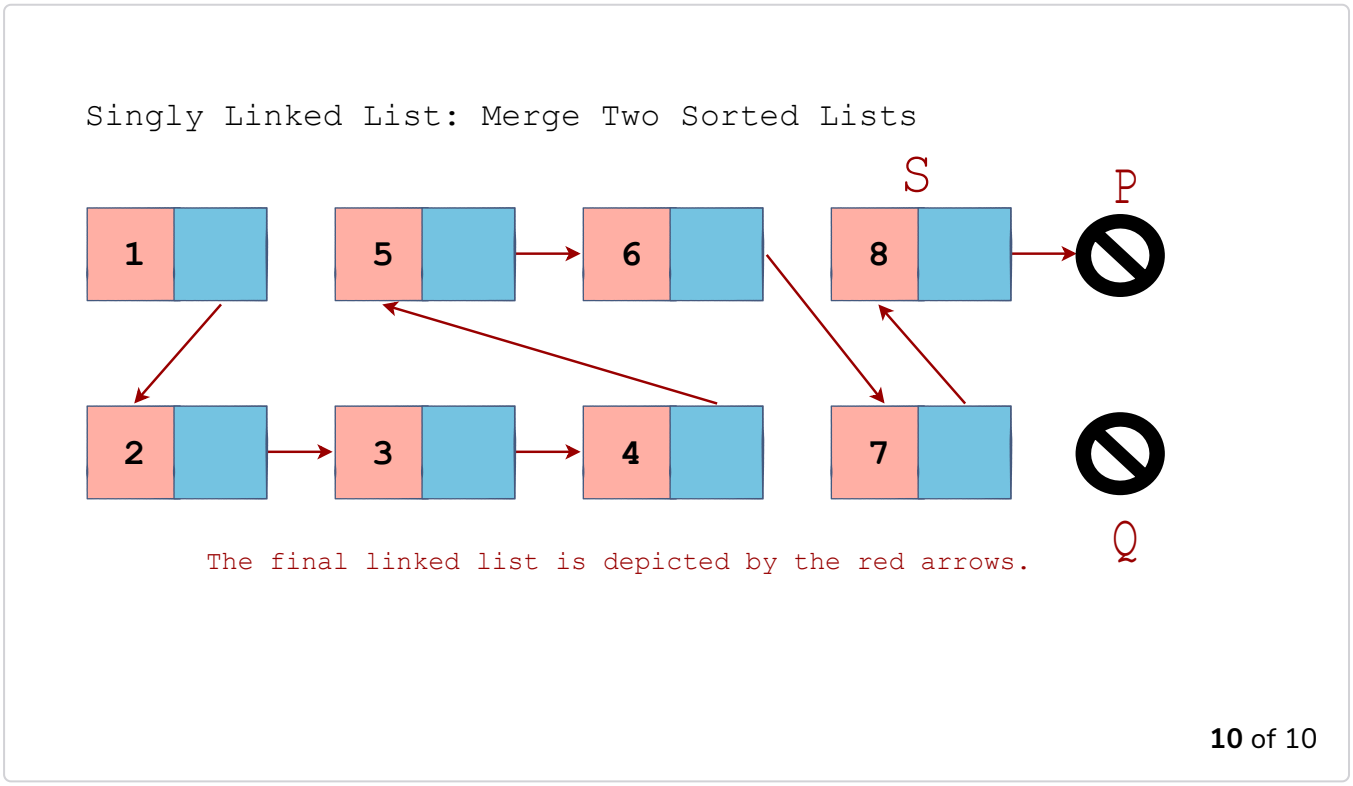
p points to 8
q points to 7
7 < 8
make s point to 7 and make q point to the next node

Singly Linked List: Merge Two Sorted Lists



p points to 8
q points to null
make s point to 8





Hope you understood the algorithm.

Implementation

Now let's look at the code below:

```
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
```

merge_sorted(self, llist)

Explanation

We pass `llist` which is the second linked list that we are going to merge with the linked list on which the class method `merge_sorted` is called. As discussed in the algorithm, `p` and `q` will initially point to the heads of each of the two linked lists (**lines 3-4**). On **line 5**, we declare `s` and assign `None` to it.

On **lines 7-10**, we handle the case if one of the linked lists is empty. If `not p` evaluates to `true`, it means that the first linked list is empty; therefore, we return `q` on **line 8**. Similarly, we check for the second linked list by using `q` which points to the head of the second linked list (`llist`). If it's empty, then we return `p`.

Next, we'll code the main idea of our algorithm. On **line 12**, we will proceed if both `p` and `q` are not `None`. If `p` and `q` are not `None`, then we compare the data of the nodes they are pointing to. If `p.data` is less than `q.data`, then `s`

will point to `p` (**line 14**). On the other hand, if `q.data` is less than `p.data`, `s` will point to `q` (**line 17**). Also, as shown in the algorithm, `p` and `q` will move along if `s` points to the node they were previously pointing to. Therefore, based on whichever condition is true, we update `p` and `q` accordingly by pointing it to `s.next` (**line 15** and **line 18**). Now you can note that the node with the lesser value of `p` and `q` will be the first node of our merged linked list. Therefore, we set it as the `new_head` on **line 19**.

Let's go ahead and break down the code in the following `while` loop (**lines 20-28**):

```
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
```

The while loop will run until either `p` or `q` becomes `None`. Again, we'll execute the corresponding block of code based on the condition in the if-statement. If `p.data` is less than or equal to `q.data`, then we want to point `s` to what `p` is pointing to and move `p` along its respective linked list. Therefore, we save what `p` is pointing to by assigning it to `s.next` (**line 22**). On **line 23**, we update the value of `s` to `p` because `p.data` is less than or equal to `q.data`. Now we make `p` move along by pointing it to the `next` node of `s` (**line 24**).

Lines 26-28 are the mirror image of **lines 22-34** except that they will be executed if `q.data` is less than `p.data`. Also, they'll make `s` point to whatever `q` was pointing to and will make `q` move along its linked list.

Now let's discuss the following portion of the code (**lines 29-33**):

```
  if not p:
      s.next = q
  if not q:
      s.next = p
  return new_head
```

The code will reach this point after the `while` loop terminates which implies

The code will reach this point after the `while` loop terminates which implies either `p` or `q` or both `p` and `q` have become `None`. We check using the

conditions on **line 29** and **line 31** that we have reached the `None` for which of the linked lists. If we have reached the end of `p`, i.e., the condition on **line 29** becomes `true`, we make the `next` of `s` point to `q`. This means that `s` will now point to the remaining `llist`. This will complete the entire chain of our merged linked list. In the same way, we check if `q` has reached the end of the linked list or not and update `s.next` accordingly (**line 32**).

Finally, on **line 33**, we return `new_head` from the method which is the head node of our merged linked list made from `llist` and the linked list on which this class method is called.

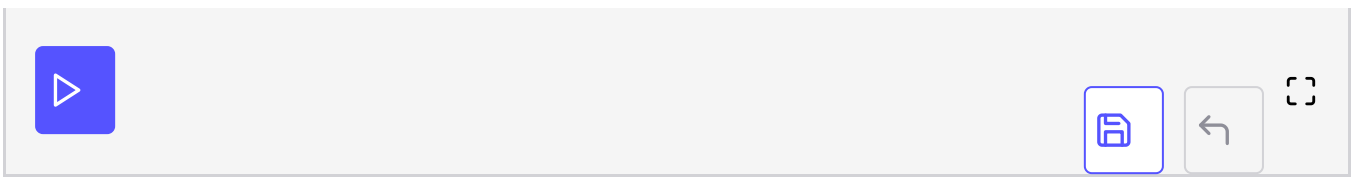
As we have discussed the `merge_sorted` method, we'll verify the method by making it part of the `LinkedList` class:

```
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

llist_1 = LinkedList()
llist_2 = LinkedList()

llist_1.append(1)
llist_1.append(5)
llist_1.append(7)
llist_1.append(9)
llist_1.append(10)

llist_2.append(2)
llist_2.append(3)
llist_2.append(4)
llist_2.append(6)
llist_2.append(8)

llist_1.merge_sorted(llist_2)
llist_1.print_list()
```



You can make your test cases and keep playing with the `merge_sorted` in the coding widget above. I hope you enjoyed this lesson. See you in the next one!