educative

Introduction

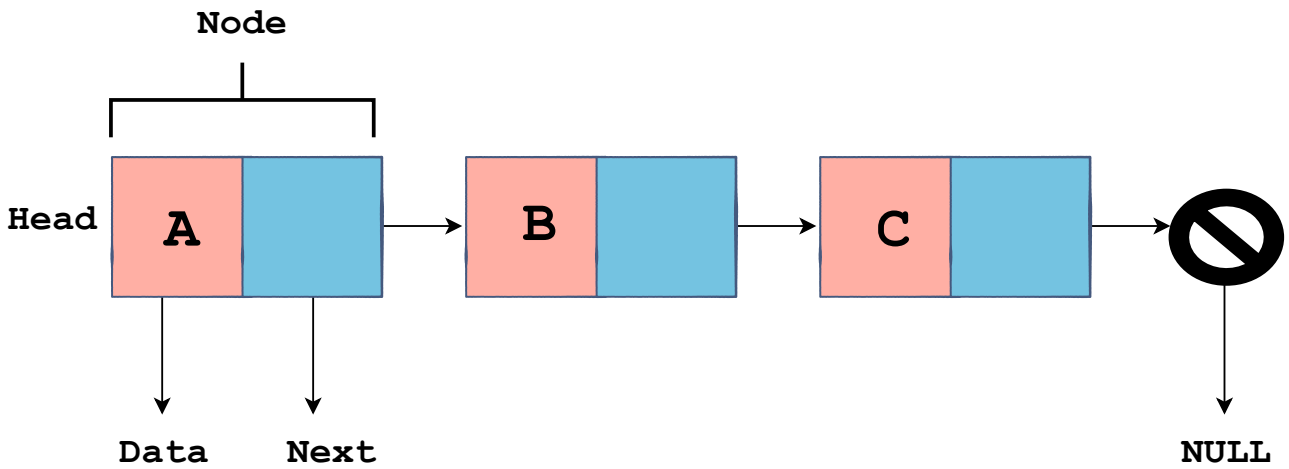
This lesson introduces you to the singly linked list and implements its structure in Python.

We'll cover the following

For the sake of this course, we will go over the following different types of linked lists and implement them in Python:

1. Singly Linked Lists
2. Doubly Linked Lists
3. Circular Linked List

Below is a simple depiction of a singly linked list:



Singly Linked List

Structure

Every linked list consists of nodes, as shown in the illustration above. Every node has two components:

1. Data
2. Next

The *data* component allows a node in the linked list to store an element of data that can be of type string, character, number, or any other type of object. In the illustration above, the data elements are `A`, `B`, and `C` which are of character type.

The *next* component in every node is a pointer that points from one node to another.

The start of the linked list is referred to as the **head**. `head` is a pointer that points to the beginning of the linked list, so if we want to traverse the linked list to obtain or access an element of the linked list, we'll start from `head` and move along.

The last component of a singly linked list is a notion of null. This null idea terminates the linked list. In Python, we call this `None`. The last node in a singly linked list points to a null object, and that tells you that it's the end of the linked list.

Arrays vs. Linked Lists

	Arrays	Linked Lists

	Arrays	Linked Lists
Insertion/Deletion at the beginning of the array or linked list given a value	$O(n)$	$O(1)$
Access Element	$O(1)$	$O(n)$
Contiguous Memory	Yes	No

Insertion/Deletion

The insertion/deletion operation is in $O(n)$ operations for insertion/deletion of value at the beginning of the array. Now think if we are given an array and a value to insert at the beginning of an array. For insertion, we have to shift all the elements in the array to the right. The shifting makes room for the element at the beginning of the array. Due to the shifting of the elements, the time complexity is $O(n)$. The same is the case with the deletion of an element from the beginning of an array. Again, we shift all the elements back by one index. Therefore, this operation also has a cost of $O(n)$ time complexity.

Inserting a node at the head of a linked list given the head node is a constant-time operation as we need to change the orientation of a few pointers. If we are given the exact pointer after which we have to insert another node, it will be a constant-time operation.

Accessing Elements

Accessing any element given an index in arrays is better than accessing nth elements in linked lists. It is a constant time operation to access elements in arrays. If given an array and an index, it can immediately give

you the element at which the entry is stored. This is because arrays are contiguous.

Accessing an nth element in a linked list is an $O(n)$ operation given that you have access to the head node of the linked list. If we want to access an element, we need to start from the head pointer and traverse the entire linked list before we can get to it.

Contiguous Memory

Arrays are contiguous in memory which allows the access time to be constant, whereas, in linked lists, you do not have the luxury of contiguous memory.

You should probably keep in mind the table above when it comes to trade-offs between arrays and linked lists. If you're given a problem, you might want to consider whether or not an array is the preferred data structure. For that assessment, you will need to be aware of the pros and cons of the two data structures.

Implementation

Now let's go ahead and create our classes in Python:

- `Node` class
- `LinkedList` class

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

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

class Node and class LinkedList

We create a class called `Node`. In the constructor of this class, we give the argument of `self` and `data` on **line 2**. Every node is going to consist of `data` and `next`. We define `self.data` equal to `data` that is passed into the constructor of the object of class `Node` (**line 3**). We set `self.next` equal to `None` on **line 4**. This is something that we'll set again as we make use of the node, but for now, we just set it to `None`. That's pretty much all we need for the `Node` class right now.

On **line 6**, we define a `LinkedList` class, and in the constructor, we again pass `self`. On **line 8**, we define the `head` pointer, which will point to the first node in the linked list. Initially, we just set it equal to `None`.

Now that we have defined the classes in our implementation, we'll learn how to insert elements into a linked list in the next lesson.

Back

Quiz

Next

Insertion

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/introduction__singly-linked-lists__data-structures-and-algorithms-in-python)

Insertion

In this lesson, we'll learn how to insert elements in a linked list at different places.

We'll cover the following

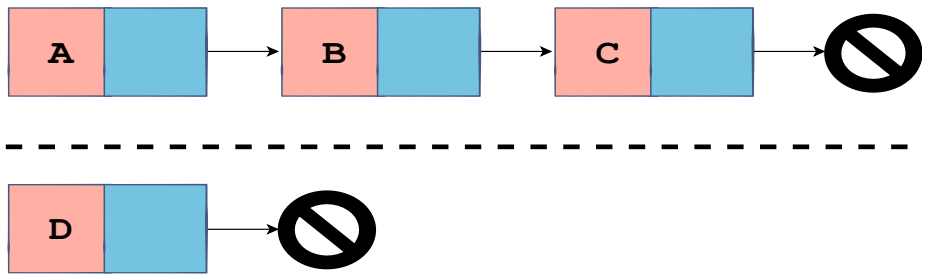
In the previous lesson, we defined our `Node` and `LinkedList` classes. In this lesson, we'll implement the class methods to insert elements in a linked list:

1. `append`
2. `prepend`
3. `insert_after_node`

Append

The `append` method will insert an element at the end of the linked list. Below is an illustration which depicts the append functionality:

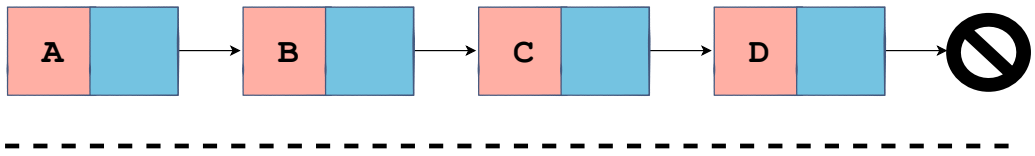
Singly Linked List: Append



Append "D" to the linked list

1 of 2

Singly Linked List: Append



2 of 2

Now let's move on to writing some code.

```
1  class Node:
2    def __init__(self, data):
3      self.data = data
4      self.next = None
5
6  class LinkedList:
7    def __init__(self):
```

```
8       self.head = None
9
10    def append(self, data):
11      new_node = Node(data)
```

class Node and class LinkedList

We define a `new_node` using our `Node` class on **line 11**. It consists of the `data` and the `next` field. We pass in `data` to the `append` method, and the data field in `new_node` has the entry of `data` that we passed to the `append` method.

Empty Linked List Case

For the `append` method, we also need to cater for the case if the linked list is empty.

```
1  class Node:
2    def __init__(self, data):
3      self.data = data
4      self.next = None
5
6  class LinkedList:
7    def __init__(self):
8      self.head = None
9
10   def append(self, data):
11     new_node = Node(data)
12     if self.head is None:
13       self.head = new_node
14       return
```

class Node and class LinkedList

In the above code, we check if the linked list is empty by checking the head of the linked list. If the `self.head` is `None` on **line 12**, it implies that it's an empty linked list and there's nothing there. The head pointer doesn't point to anything at all, and therefore there is no node in the

linked list. If there is no node in the linked list, we set the head pointer to the `new_node` that we created on **line 13**. In the next line, we simply `return`. The case of an empty linked list is relatively easy to handle.

Non-empty Linked List Case

Let's see what we can do if the linked list is not empty. We have `new_node` that we create, and we want to append it to the linked list. We can start from the head pointer and then move through each of the nodes in the linked list until we get to the end, i.e. `None`. Once we arrive at the location that we want to insert the `new_node` at, we insert as shown below:

```
1  class Node:
2    def __init__(self, data):
3      self.data = data
4      self.next = None
5
6  class LinkedList:
7    def __init__(self):
8      self.head = None
9
10   def append(self, data):
11     new_node = Node(data)
12     if self.head is None:
13       self.head = new_node
14       return
15     last_node = self.head
16     while last_node.next:
17       last_node = last_node.next
18     last_node.next = new_node
```

class Node and class LinkedList

On **line 15**, we define `last_node` which is initially equal to the head. This implies we're at the start of the linked list. We have named the variable we defined on **line 15** `last_node` because that's what it will eventually point to. It will start at the beginning of the linked list and move through the linked list as long as the `last_node.next` doesn't point to `None`. We keep moving from node to node on **line 17** until we get to the `last_node`

where `last_node.next` will point to `None` and will terminate the while loop on **line 16**. After the `while` loop concludes, `last_node` points to the last node. On **line 18**, we input our `new_node` into the linked list by setting the `next` of `last_node` to `new_node` which has its own `next` pointing to `None`.

Now we want some way to verify our `append` method where we can print out the nodes of the linked list. For this purpose, let's create a method called `print_list()`.

print_list()

`print_list` is a class method, so it will take `self` as an argument and print out the entries of a linked list. We will start from the head pointer and print out the data component of the node and then move to the next node. We'll keep a check on the next node to make sure it is not `None`. If it's not, we move to the next node. This way, we keep printing out data until we've hit the null terminating component of the linked list. Let's implement this in Python!

```
 6  class LinkedList:
 7    def __init__(self):
 8      self.head = None
 9
10    def print_list(self):
11      cur_node = self.head
12      while cur_node:
13        print(cur_node.data)
14        cur_node = cur_node.next
15
16    def append(self, data):
17      new_node = Node(data)
18      if self.head is None:
19        self.head = new_node
20        return
21      last_node = self.head
22      while last_node.next:
23        last_node = last_node.next
24      last_node.next = new_node
25
26  llist = LinkedList()
27  llist.append("A")
```

```
llist.append("A")
llist.append("B")
llist.append("C")
llist.append("D")


llist.print_list()
```

class Node and class LinkedList

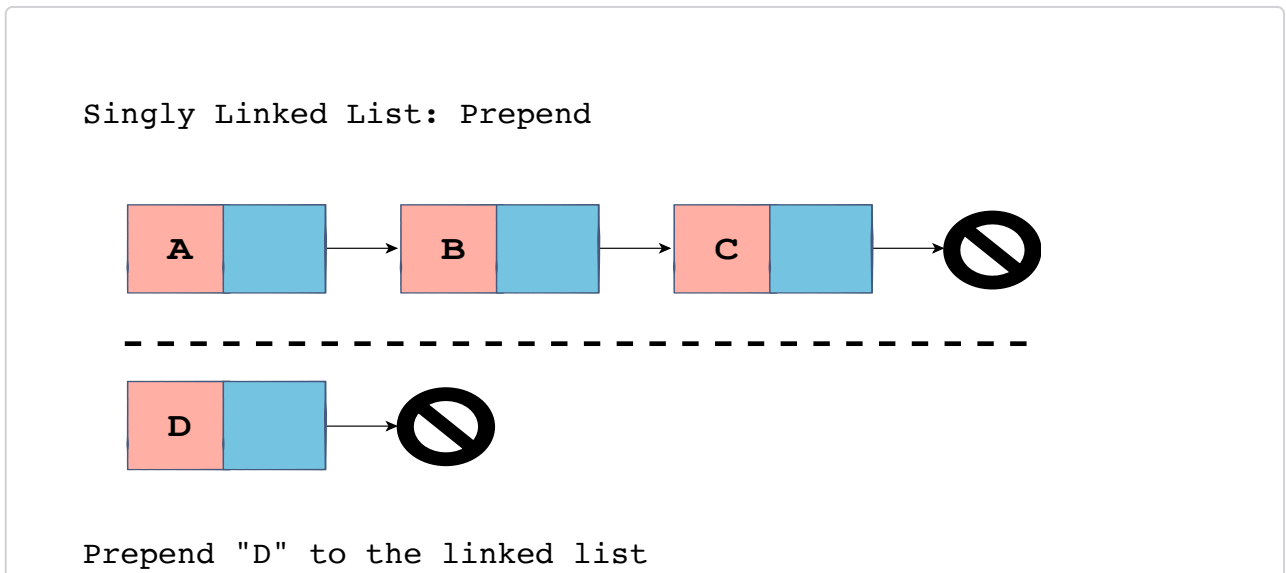
There you go! We initialize `cur_node` equal to the head of the linked list. Then we use a `while` loop which keeps running and printing the data if `cur_node` is not equal to `None`.

In the code above, we append four elements to the linked list. You can see this for yourself in the output.

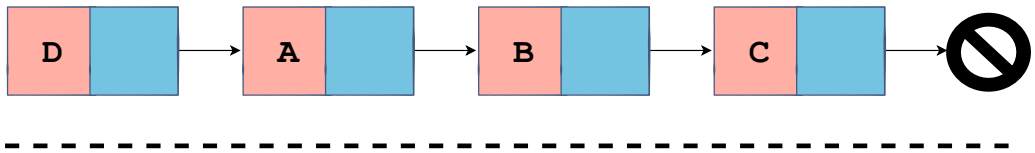
Now we'll move on to another method of inserting elements in a linked list.

Prepend

The `prepend` method will insert an element at the beginning of the linked list, as shown in the illustration below:



1 of 2



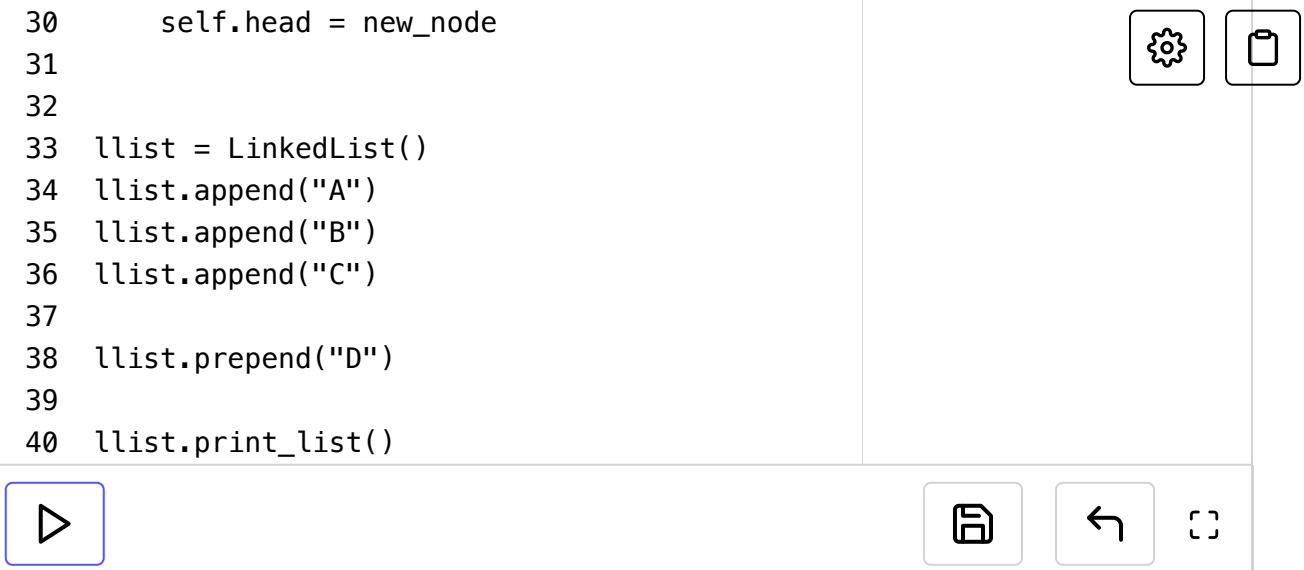
2 of 2

We create a new node based on the data that is passed in, which in the above case is “D”. Now we want the next of this node to point to the current head of the linked list and replace the head of the linked list.

Let’s go ahead and write this code after which we’ll walk it through step by step.

```
13        print(cur_node.data)
14        cur_node = cur_node.next
15
16    def append(self, data):
17      new_node = Node(data)
18      if self.head is None:
19        self.head = new_node
20        return
21      last_node = self.head
22      while last_node.next:
23        last_node = last_node.next
24      last_node.next = new_node
25
26    def prepend(self, data):
27      new_node = Node(data)
28
29      new_node.next = self.head
```

```
30      self.head = new_node
31
32
33  llist = LinkedList()
34  llist.append("A")
35  llist.append("B")
36  llist.append("C")
37
38  llist.prepend("D")
39
40  llist.print_list()
```



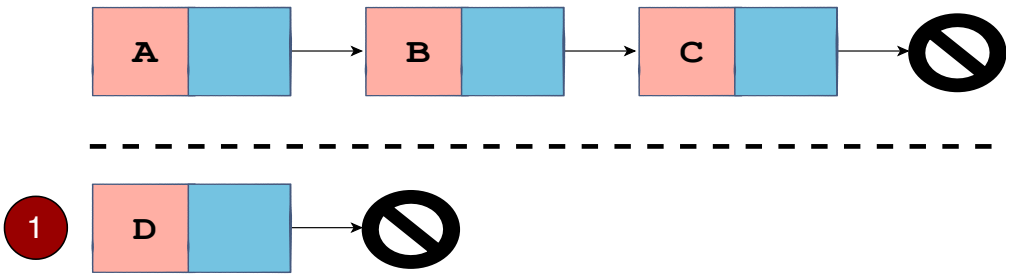
class Node and class LinkedList

We create a method called `prepend`. This also takes `self` and `data` since we need to tell it what to prepend to the linked list. We create a node based on the data passed into the method. Next, on **line 29**, we point the next of the `new_node` to the current head of the linked list, and then we set the `head` of the linked list equal to `new_node` on **line 30**. We have now prepended `D` to `llist` in the code above which previously only contained `A`, `B`, and `C`. You can play around and verify the `prepend` method for yourself!

Insert After Node

The last insertion method that we want to consider in this lesson is inserting an element after a given node. In the example illustrated below, we have a linked list that contains `A`, `B`, and `C` elements. Now we want to insert `D`, which is a new node, after node `B`.

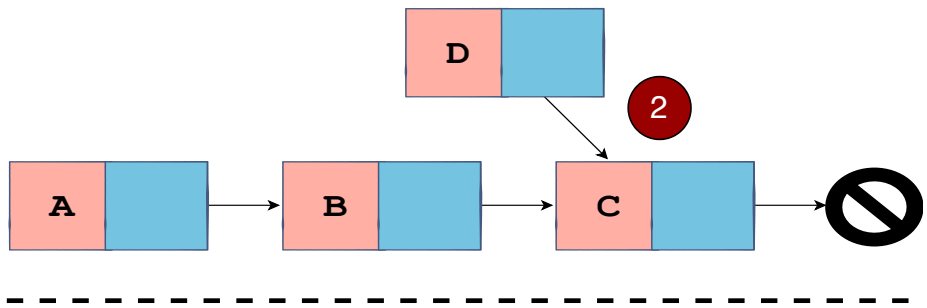
Singly Linked List: Insert After Node



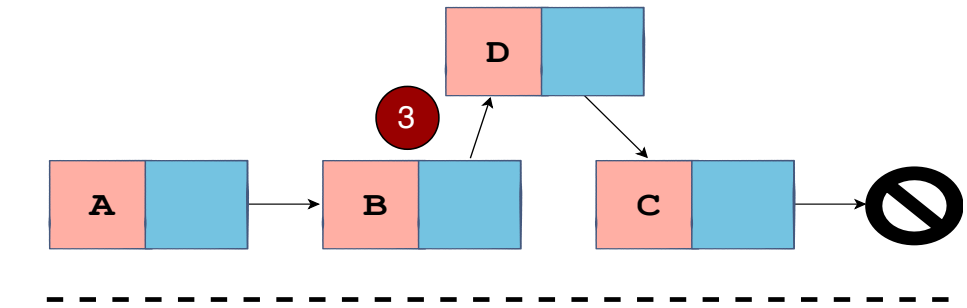
Insert "D" After Node B

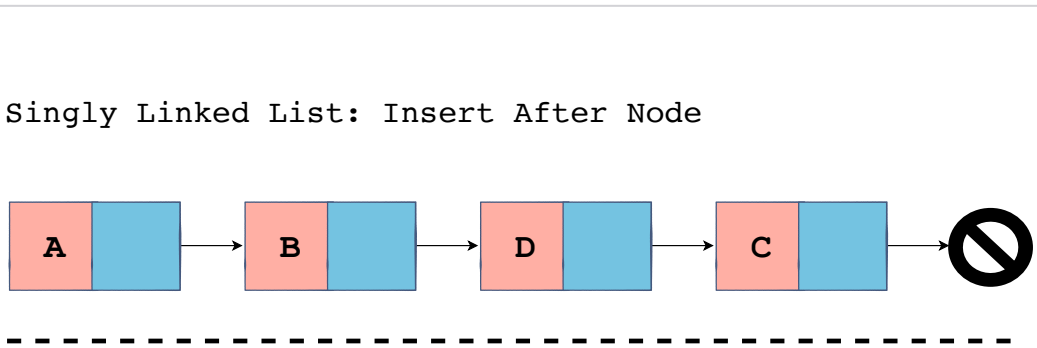
1 of 4

Singly Linked List: Insert After Node



2 of 4





Let's break down the steps required for us to do the operation of inserting `D` after `B`.

First of all, we will create a new node based on the data `D`. That is *step 1* as depicted in the slides. Next, we need to check if the node to be inserted after is in the linked list or not. If it's not in the linked list, we'll return; otherwise, we set the next pointer of the new node (`D`) to point to what

the next pointer of the node `B` is pointing to, i.e. Node `C`. You can refer to *step 2* in the slides above to make this clearer for yourself. Next, to implement *step 3*, we can change the next pointer of the node `B` to point to the new node `D`.

Now let's go ahead and code it!

```
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)

```

class Node and class LinkedList

In the code above, we create a new method called `insert_after_node` on **line 32**. It takes `self` since it is a class method. It also takes `prev_node` which is the previous node after which we have to insert the new node and `data` which we'll use to make the `new_node`.

As mentioned before, we first want to check if the `prev_node` is `None` or not. If `prev_node` is `None` or does not exist, then we print the following on **line 34**:

```
Previous node does not exist.
```

and return on **line 35**.

If `prev_node` is not `None`, then we create a new node on **line 36**. Now you need to refer to the illustration for the *Insert After Node* method. As shown in the illustration on *step 2*, on **line 38**, we point the `next` of the `new_node` to the next node of the node after which the insertion has to take place.

To execute the *third step* according to the illustration, we set the `prev_node.next` to the `new_node` on **line 39** so that the `new_node` now comes after the `prev_node`.

In the code above, we insert `D` after `B` and print out the linked list to verify our method. As you have seen, it works!

This is as much as we'll cover about insertion in a linked list, so in the next lessons, we'll continue to build on this code to run other methods in the `LinkedList` class. I hope this lesson was helpful. See you in the next lesson!

Back

Introduction

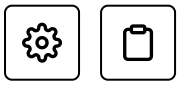
Next

Deletion by Value

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/insertion__singly-linked-lists__data-structures-and-algorithms-in-python)

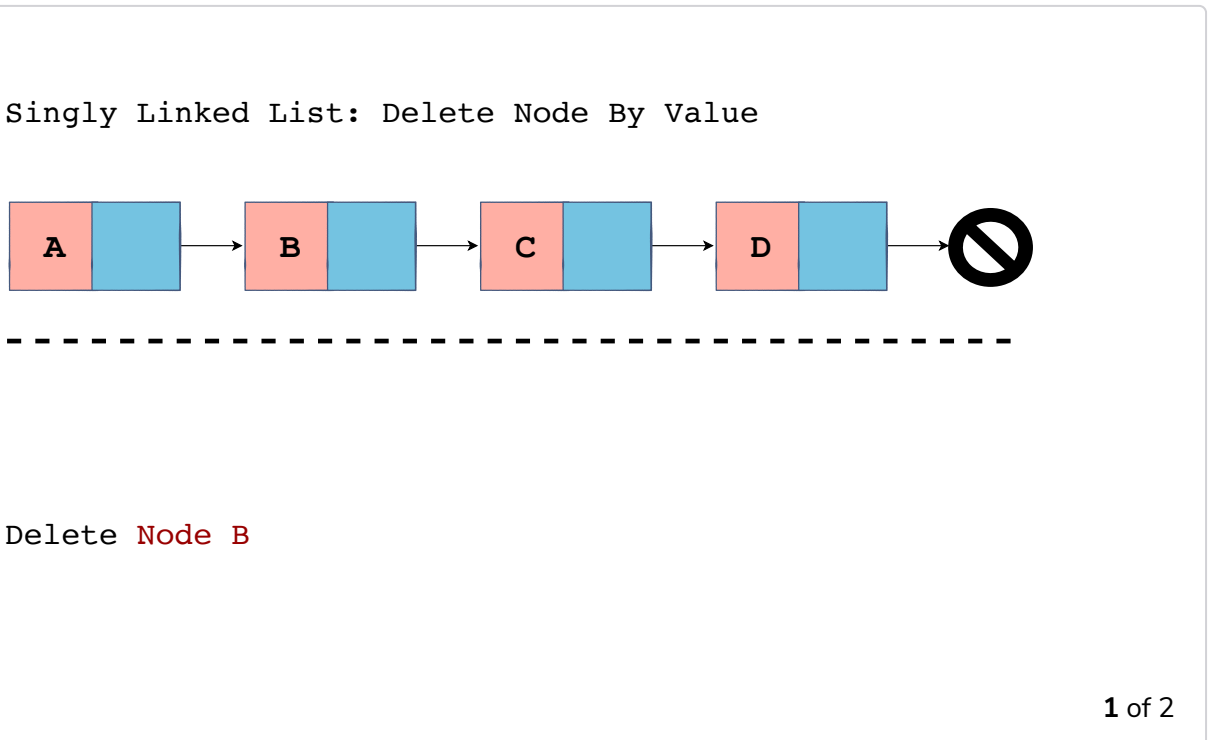


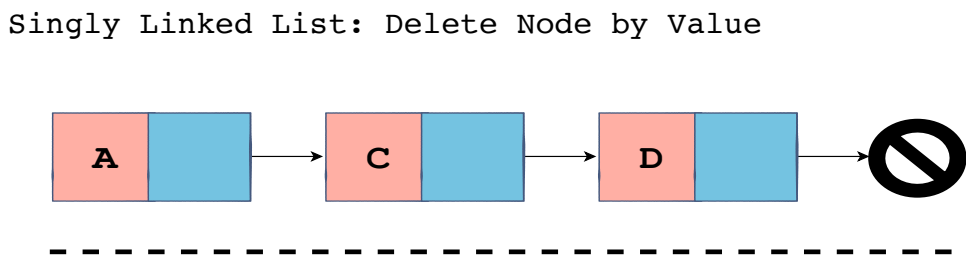
Deletion by Value

In this lesson, we will learn how to delete a node based on a value from a linked list.

We'll cover the following

In this lesson, we will investigate singly-linked lists by focusing on how one might delete a node in the linked list. In summary, to delete a node, we'll first find the node to be deleted by traversing the linked list. Then, we'll delete that node and update the rest of the pointers. That's it!





2 of 2

Algorithm

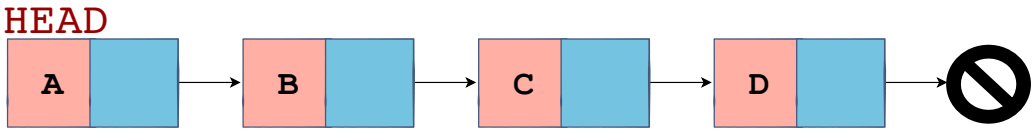
To solve this problem, we need to handle two cases:

1. Node to be deleted is head.
2. Node to be deleted is not head.

Case of Deleting Head

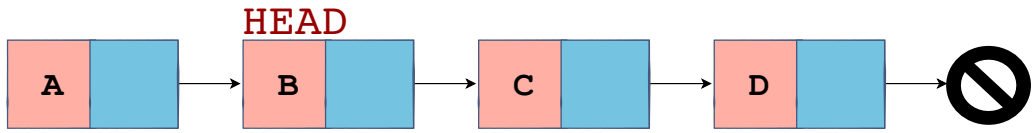
Let’s look at the illustration below to get a fair idea of the steps that we are going to follow while writing the code.

Singly Linked List: Delete Head Node



1 of 4

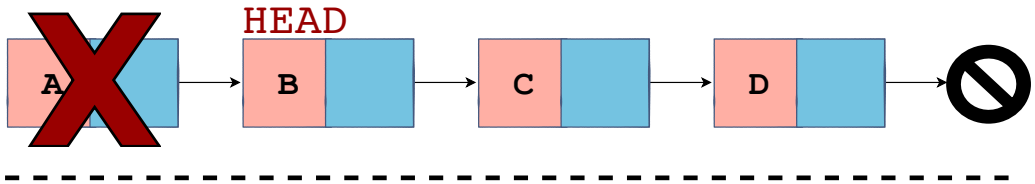
Singly Linked List: Delete Head Node



Update head to the next node of the previous head

2 of 4

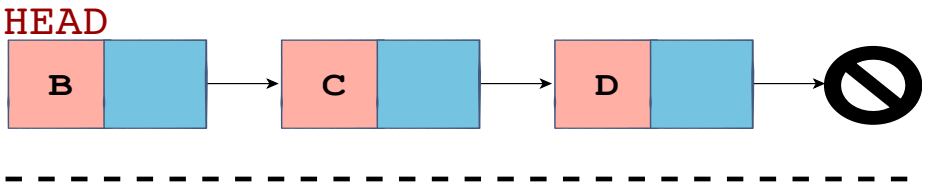
Singly Linked List: Delete Head Node



Delete the previous head node

3 of 4

Singly Linked List: Delete Node



4 of 4

Now let's go ahead and implement the case illustrated above in Python.

```
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
```

```
8       return
```

delete_node(self, key)

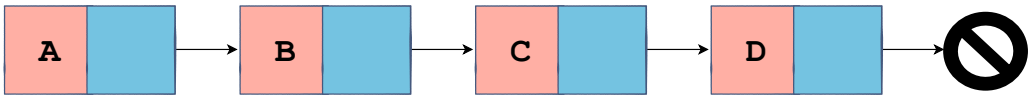
The class method `delete_node` takes `key` as an input parameter. On **line 3**, we'll declare `cur_node` as `self.head` to have a starting point to traverse the linked list. To handle the case of deleting the head, we'll check if `cur_node` is not `None` and if the data in `cur_node` is equal to `key` on **line 5**. Note that `cur_node` is pointing to the head of the linked list at this point. If `key` matches `cur_node.data`, we'll update the head of the linked list (`self.head`) to `cur_node.next`, i.e., the next node of the previous head node (**line 6**). Once we have updated the head of the linked list, we'll set the node to be deleted (`cur_node`) to `None` (**line** 7) and return from the method.

Now that we have written the code for deleting the head node let's move on to the other case of deletion, deleting a node from a linked list which is not the head node.

Case of Deleting Node Other Than the Head

In this case, we'll traverse the linked list to find the node that we are supposed to delete. As we move along, we also need to keep track of the previous node of the node to be removed. Below is a slide for you to visualize the process.

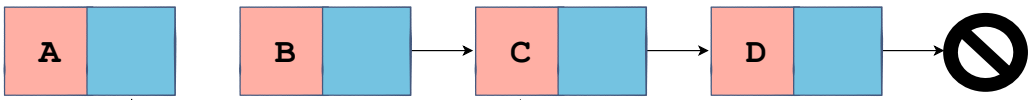
Singly Linked List: Delete Node By Value



Delete Node B

1 of 4

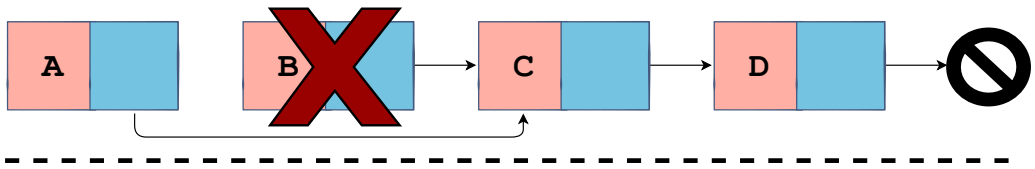
Singly Linked List: Delete Node By Value



Previous node of Node B will point to the next node of Node B

2 of 4

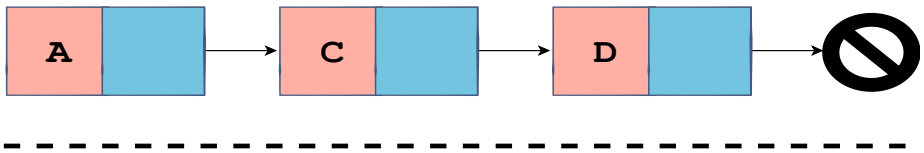
Singly Linked List: Delete Node By Value



Delete Node B

3 of 4

Singly Linked List: Delete Node By Value



4 of 4

Now, let's turn to the code part. Check it out below:

```
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
```

```
 8      return
 9
10    prev = None
11    while cur_node and cur_node.data != key:
12      prev = cur_node
13      cur_node = cur_node.next
14
15    if cur_node is None:
16      return
17
18    prev.next = cur_node.next
19    cur_node = None
```

delete_node(self, key)

We have already discussed the code present in **lines 2-8**. Now we'll concern ourselves with the code starting from **line 10** where `prev` is declared and set to `None`. This will keep track of the previous node of the node to be deleted. On the next line (**line 11**), we have a `while` loop which will run until `cur_node` becomes `None`. This implies that `key` doesn't exist in our linked list. Another occurrence that would stop the loop would be the `cur_node.data` equaling `key` which refers to the case where we have found the node to be deleted. In the `while` loop, we set `prev` to `cur_node` on **line 12** to keep track of the previous node while `cur_node` is updated to the next node in the next line.

When the `while` loop terminates, we check using an if-condition on **line 15** whether or not it's because of `cur_node` being `None`, which will imply that `key` did not match any of the data of the current node. If it's true, then we return from the method as there is no node to delete. However, if `cur_node` is not `None` and its data matches with the `key`, we proceed to **line 18**. As we kept track of the previous node of the node to be deleted (`prev`), we now want to point it to the next node of the node to be deleted instead of pointing it to the node that we want to delete (`cur_node`). Therefore, we set `prev.next` to `cur_node.next`. In this way, we have removed the link of the `cur.node` that was previously in between. Finally, we set the node to be deleted (`cur_node`) equal to `None` on **line 19**.

Deleting the node based on a value from a linked list was as simple as that! The `delete_node` has been made a class method in the code widget below. Go ahead and play around with it by making more test cases!

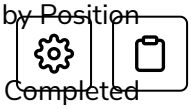
```
41          new_node = Node(data)
42
43          new_node.next = prev_node.next
44          prev_node.next = new_node
45
46      def delete_node(self, key):
47
48          cur_node = self.head
49
50          if cur_node and cur_node.data == key:
51              self.head = cur_node.next
52              cur_node = None
53              return
54
55          prev = None
56          while cur_node and cur_node.data != key
57              prev = cur_node
58              cur_node = cur_node.next
59
60          if cur_node is None:
61              return
62
63          prev.next = cur_node.next
64          cur_node = None
65
66
67  llist = LinkedList()
68  llist.append("A")
```

delete_node(self, key)

In the next lesson, we'll consider another case of deleting a node from a linked list, i.e., we will delete a node based on position rather than value. Let's find out more in the next lesson!

Insertion

Deletion by Position



Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/deletion-by-value__singly-linked-lists__data-structures-and-algorithms-in-python)

Deletion by Position

In this lesson, you will learn how to delete a node at a given position in a linked list.

We'll cover the following

We will solve this problem in a very similar way as we have done in the last lesson.

Cases to Consider

Again, we'll consider two cases while writing our code:

1. Node to be deleted is at position 0
2. Node to be deleted is not at position 0

The overall logic will stay the same as in the previous lesson except that we'll change the code a bit to cater to position rather than a key.

Implementation

Without any further ado, let's jump to the implementation:

```
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
```

```
 8
 9      prev = None
10      count = 0
11      while cur_node and count != pos:
12          prev = cur_node
13          cur_node = cur_node.next
14          count += 1
15
16      if cur_node is None:
17          return
18
19      prev.next = cur_node.next
20      cur_node = None
```

delete_node_at_pos(self, pos)

Explanation

The `delete_node_at_pos` takes in `pos` as one of the input parameters.

First of all, we check on **line 2** if the linked list is an empty list or not. We only proceed to **line 3** if `self.head` is not `None`.

As we discussed before, we need to handle the case where `pos` will equal `0`. If `pos` equals `0`, it essentially means that we want to delete the head node.

On **line 3**, `cur_node` is initialized to the head node. Next, we check if `pos` is `0` or not. If it is, we update the head node to the next node of `cur_node`, set `cur_node` to `None`, and return from the method (**lines 5-7**). On the other hand, if we are deleting a node at a position other than the head node, we will proceed to **line 9**. We declare `prev` and set it to `None` and on **line 10**, we initialize `count` to `0`. Now we traverse the linked list by updating `prev` and `cur_node` (**lines 12-13**) and increment `count` by `1` on **line 14**. The `while` loop will terminate if `cur_node` becomes `None` or `count` becomes equal to `pos` which will imply that `cur_node` will be the node that we want to delete.

The code on **lines 16-20** is precisely the same as in the `delete_node` class method.

I hope everything's been clear up until now. You can practice this method more by playing around with it in the coding widget below:

```
64          cur_node = None
65
66      def delete_node_at_pos(self, pos):
67          if self.head:
68              cur_node = self.head
69
70              if pos == 0:
71                  self.head = cur_node.next
72                  cur_node = None
73                  return
74
75              prev = None
76              count = 0
77              while cur_node and count != pos:
78                  prev = cur_node
79                  cur_node = cur_node.next
80                  count += 1
81
82              if cur_node is None:
83                  return
84
85              prev.next = cur_node.next
86              cur_node = None
87
88
89  llist = LinkedList()
90  llist.append("A")
```

class Node and class LinkedList

That was all regarding deleting nodes from a singly linked list. In the next lesson, we'll learn how to calculate the length of a linked list.

Deletion by Value

Length

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/deletion-by-position__singly-linked-lists__data-structures-and-algorithms-in-python)

Length

In this lesson, you will learn how to calculate the length of a linked list.

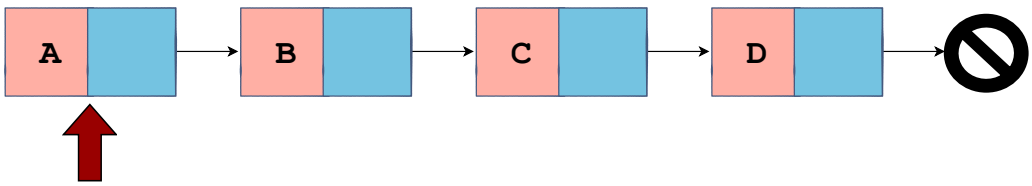
We'll cover the following

In this lesson, we'll calculate the length or the number of nodes in a given linked list. We'll be doing this in both an iterative and recursive manner.

Algorithm

Let's look at a linked list and recall how we managed to print out the elements of a linked list. We iterate through every element of the linked list. We start from the head node and while we don't reach `None`, we print the data field of the node that we point to and increment the while loop by setting the current node equal to the next node.

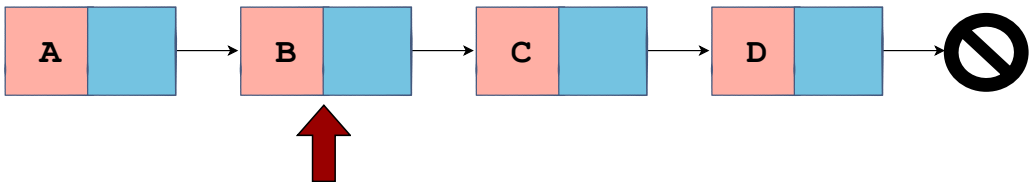
Singly Linked List : Print Method



A

1 of 5

Singly Linked List : Print Method

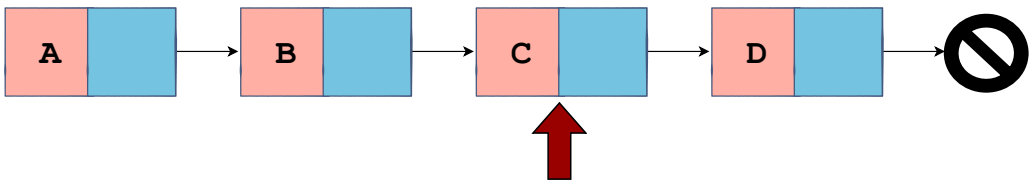


A

B

2 of 5

Singly Linked List : Print Method



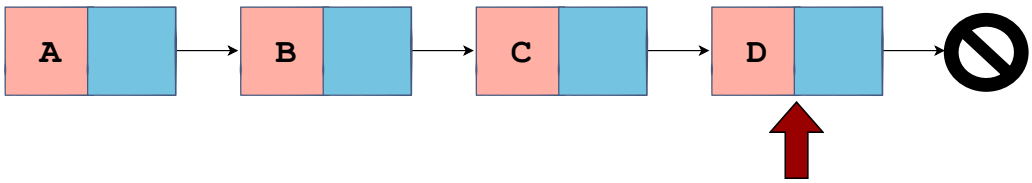
A

B

C

3 of 5

Singly Linked List : Print Method



A

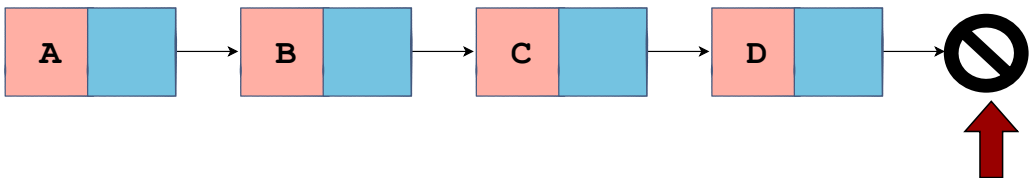
B

C

D

4 of 5

Singly Linked List : Print Method



A

B

C

D Loop terminated.

5 of 5

Iterative Implementation

The above algorithm is going to help us construct an iterative method to calculate the length of a linked list. Let's go ahead and create a method `len_iterative` and step through it.

```
1  def len_iterative(self):
2      count = 0
3      cur_node = self.head
4      while cur_node:
5          count += 1
6          cur_node = cur_node.next
7      return count
```

len_iterative(self)

`len_iterative` takes `self` since it's a class method. As we start from the beginning of the linked list, we set `cur_node` equal to the head of the linked list on **line 3**. Then we go through each of the nodes until we hit `None`, which will terminate the `while` loop on **line 4**. We keep a count of how many nodes by setting a `count` variable equal to zero at the

beginning of the method on **line 2**. `count` will keep track of the number of nodes we've encountered as long as the `cur_node` is not `None` by incrementing itself on **line 5**.

Let's go ahead and verify this code:

```
79              cur_node = cur_node.next
80              count += 1
81
82          if cur_node is None:
83              return
84
85          prev.next = cur_node.next
86          cur_node = None
87
88      def len_iterative(self):
89
90          count = 0
91          cur_node = self.head
92
93          while cur_node:
94              count += 1
95              cur_node = cur_node.next
96          return count
97
98
99  llist = LinkedList()
100 llist.append("A")
101 llist.append("B")
102 llist.append("C")
103 llist.append("D")
104
105
106 print(llist.len_iterative())
```

```
Output                                                    0.22s

 4
```

class Node and class LinkedList

In the code above, we have a linked list object `llist` and we insert four entries into the linked list (**lines 100-103**).

The statement on **line 106** `print(llist.len_iterative())` gives an output of 4 which proves that our implementation is correct.

Recursive Implementation

Let's move on to the recursive implementation of calculating the length of a linked list:

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

len_recursive(self, node)

In the implementation of `len_recursive`, we pass in a `node` to the method. Now if we want to calculate the length of the whole linked list, we have to pass the start of the linked list as the `node` on **line 1**. On **line 4**, we have a recursive call to `self.len_recursive` where we pass `node.next` to it.

Now, whenever we have a recursive function, we need a base case. For the `len_recursive` method, the base case is whether or not we've encountered the end of the linked list. If we reach the end of the linked list, meaning the `node` is `None`, we return zero on **line 3**. Otherwise, if the `node` is not `None`, we call `len_recursive` on **line 4** and pass in the next node. Also on **line 4**, we return `1` plus what we're going to return from `self.len_recursive(node.next)`.

Now we'll call this method in a way similar to the iterative method, but we're going to pass the node that corresponds to the head of the linked list to this method.

```
88      def len_iterative(self):
89
```

```
        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)


llist = LinkedList()
print("The length of an empty linked list is:")
print(llist.len_recursive(llist.head))
llist.append("A")
llist.append("B")
llist.append("C")
llist.append("D")

print("The length of the linked list calculated
print(llist.len_recursive(llist.head))
print("The length of the linked list calculated
print(llist.len_iterative())
```

class Node and class LinkedList

As you can see from the code above, we get output equal to `4` from the `len_recursive` method. If the linked list is empty, this method returns zero.

In conclusion, it doesn't matter if we calculate the length of a linked list

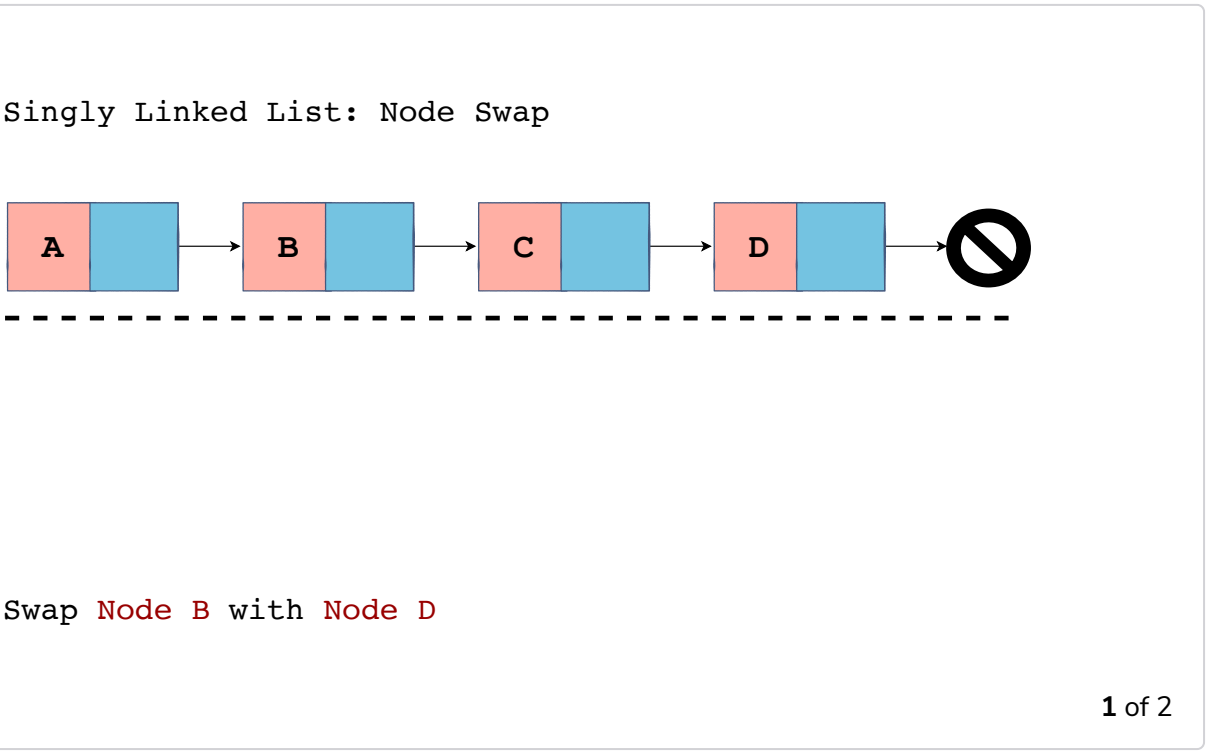
Node Swap

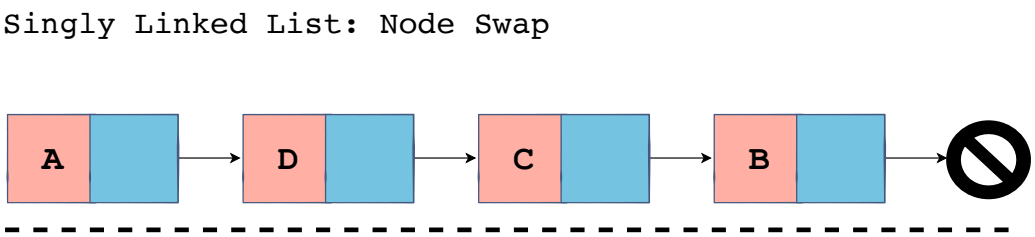
This lesson will teach you how to swap two nodes in a linked list.

We'll cover the following

- Algorithm
- Implementation
- Explanation

In this lesson, we will continue with our linked list implementation and focus on how to swap two different nodes in a linked list. We will give different keys corresponding to the data elements in the nodes. Now we want to swap the two nodes that contain those two keys.





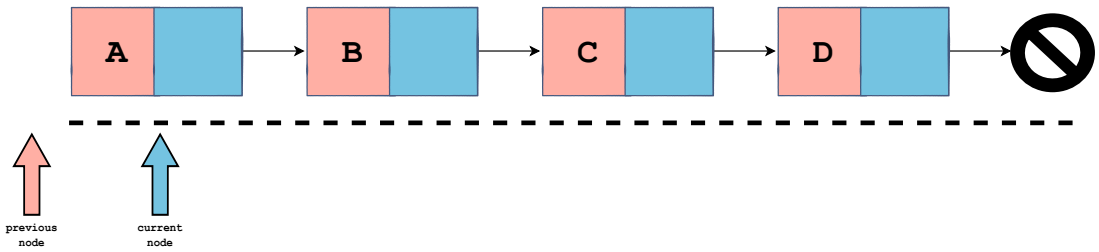
2 of 2

Let’s go ahead and break down how we might go about solving this problem. One way to solve this is by iterating the linked list and keeping track of certain pieces of information that are going to be helpful.

Algorithm

We can start from the first node, i.e., the head node of the linked list and keep track of both the previous and the current node.

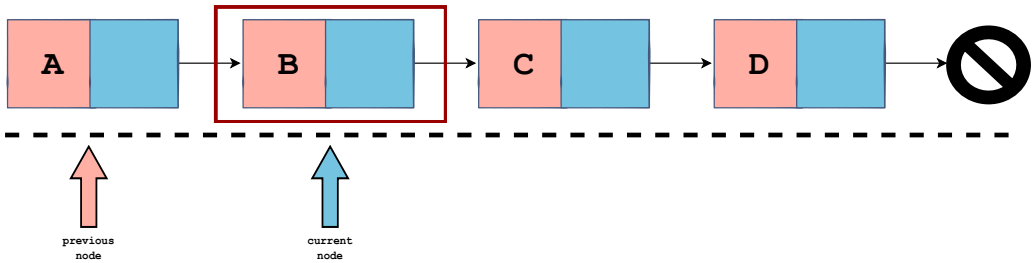
Singly Linked List: Node Swap



Swap Node B and Node C

1 of 7

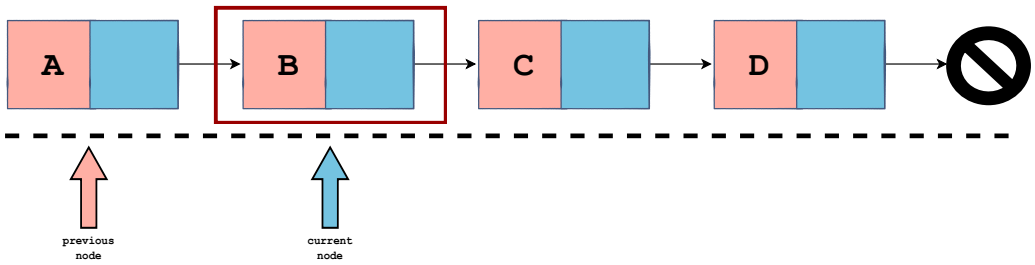
Singly Linked List: Node Swap



One Node Found!

2 of 7

Singly Linked List: Node Swap

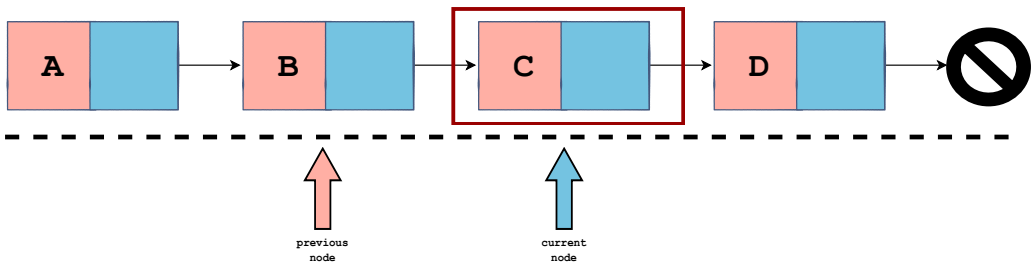


Record the current and the previous node once you find one of the nodes to be swapped while traversing

Look for the other node!

3 of 7

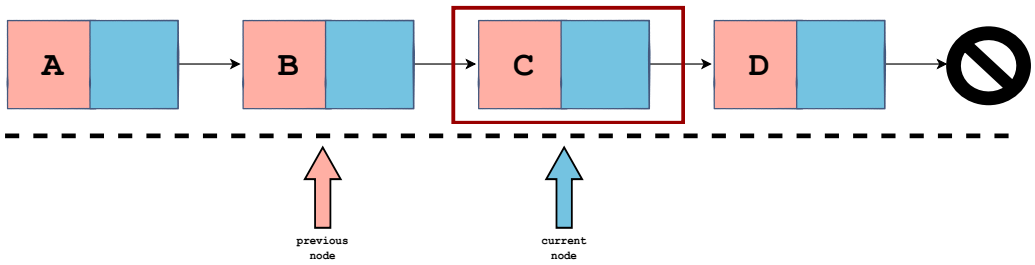
Singly Linked List: Node Swap



Other Node Found!

4 of 7

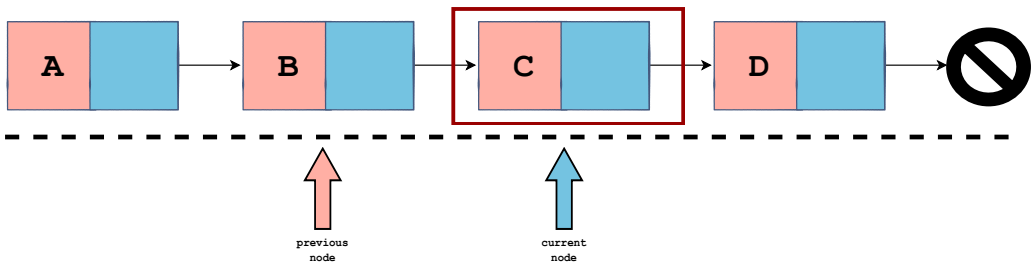
Singly Linked List: Node Swap



Record the current and the previous node once you find one of the nodes to be swapped while traversing

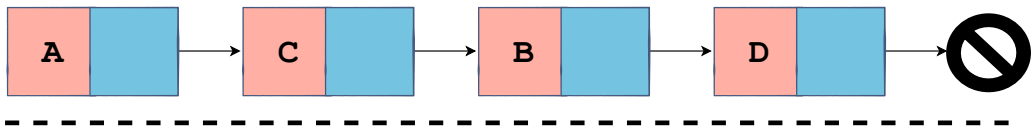
5 of 7

Singly Linked List: Node Swap



Now, let's swap the nodes found!

6 of 7



In the above illustration, we first set the current node to the head of the linked list and the previous node to nothing because there's no previous node to the current node. Next, we proceed through the linked list looking at the data elements and checking if the data element of the node that we're on matches one of the two keys. If we find the match, we record that information and repeat the same process for the second key that we're looking for. This is the general way we will keep track of the information.

There are two cases that we'll have to cater for:

1. `Node 1` and `Node 2` are not head nodes.
2. Either `Node 1` or `Node 2` is a head node.

Implementation

Now let's go ahead and write up some code that will allow us to loop through this linked list and keep track of both the current and previous node for the keys given to the method.

```
1  def swap_nodes(self, key_1, key_2):
2
```

```
3    if key_1 == key_2:
4      return
5
6    prev_1 = None
7    curr_1 = self.head
8    while curr_1 and curr_1.data != key_1:
9      prev_1 = curr_1
10     curr_1 = curr_1.next
11
12   prev_2 = None
13   curr_2 = self.head
14   while curr_2 and curr_2.data != key_2:
15     prev_2 = curr_2
16     curr_2 = curr_2.next
17
18   if not curr_1 or not curr_2:
19     return
20
21   if prev_1:
22     prev_1.next = curr_2
23   else:
24     self.head = curr_2
25
26   if prev_2:
27       prev_2.next = curr_1
28   else:
```

swap_nodes(self, key_1, key_2)

Explanation

We create a method, `swap_nodes` , in the code above, which takes `key_1` and `key_2` as input parameters. First of all, we check if `key_1` and `key_2` are the same element **(line 3)**. If they are, we return from the method on **line 4**. On **line 6** and **line 7**, we declare `prev_1` and `curr_1` to `None` and `self.head` respectively. We loop through the linked list using the `while` loop on **line 8** which runs while `curr_1` is not at the end of the linked list or it is not equal to the `key_1` that we seek. In the `while` loop, we keep updating the `prev_1` node equal to the `curr_1` and the `curr_1` to the next node in the linked list.

In the same way, we try to find if `key_2` exists in the linked list or not. We set `prev_2` equal to `None` while we set `curr_2` equal to the head of the linked list. Then again while the `curr_2` is not `None` and the `curr_2.data` is not equal to `key_2`, we update the `prev_2` and `curr_2` nodes.

On **lines 18-19**, we check to make sure that the elements we found, i.e., `curr_1` and `curr_2` actually exist or not. If either of the conditions, `not curr_1` or `not curr_2`, are not true (`curr_1` or `curr_2` is `None`) then one of them doesn't exist in the linked list. If neither key exists in the linked list or if only one of the keys exists in the linked list, we can't swap, so we `return`.

Recall the two cases that we specified while discussing the algorithm. Let's consider the case where either `curr_1` or `curr_2` is the head node. If both of the current nodes have a previous node, it implies that neither is a head node. So, we will check if the previous nodes of the current nodes exist or not. If they don't exist and are `None`, then the node without the previous node is the head node.

On **line 21**, we check if `prev_1` exists or not. If it exists, we set the `next` of `prev_1` to `curr_2` to swap it. Previously, `prev1.next` was pointing to `curr_1` but on **line 22**, we set it to point to `curr_2`. On the other hand, if `prev1` does not exist, it implies that `curr_1` is the head node and we set `self.head` to its new value, i.e., `curr_2` on **line 24**.

We repeat the same steps as above on **lines 26-29** for `prev_2` and `curr_2` and update the relevant positions with `curr_1`.

Now that we have handled the previous nodes that will point to the different nodes, we'll swap the `next` of `curr_1` with the `next` of the `curr_2` and vice versa. On **line 31**, we code this swap using the Python shorthand.

I hope you understand the above explanation. However, if the code is too hard to follow, then you can always print `curr_1`, `prev_1`, `curr_2`, or `prev_2` after the corresponding `while` loop, so it's easy for you to follow. Now let's go ahead and verify our code in the coding widget below!

```
123
124         if prev_1:
125             prev_1.next = curr_2
126         else:
127             self.head = curr_2
128
129         if prev_2:
130             prev_2.next = curr_1
131         else:
132             self.head = curr_1
133
134         curr_1.next, curr_2.next = curr_2.next,
135
136
137 llist = LinkedList()
138 llist.append("A")
139 llist.append("B")
140 llist.append("C")
141 llist.append("D")
142
143 print("Original List")
144 llist.print_list()
145
146
147 llist.swap_nodes("B", "C")
148 print("Swapping nodes B and C that are not head
149 llist.print_list()
150
```

class Node and class LinkedList

That's pretty much it for this lesson. `swap_nodes` is a tricky method to write because there are some edge cases that are not super obvious.

I hope this lesson was helpful for you and I'll see you in the next one.

Back

Length

Next

Reverse

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/node-swap__singly-linked-lists__data-structures-and-algorithms-in-python)

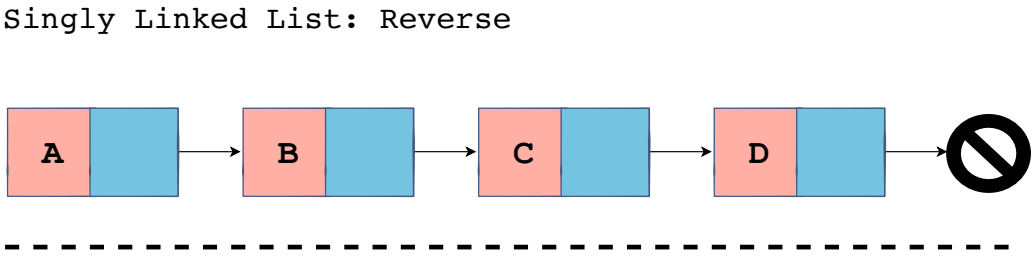
Reverse

In this lesson, you will learn how to reverse a linked list in both an iterative and recursive manner.

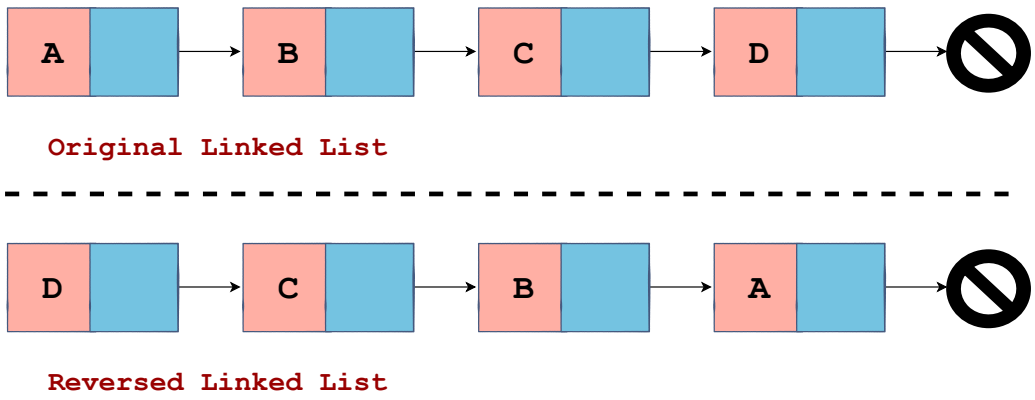
We'll cover the following

In this lesson, we will look at how we can reverse a singly linked list in an iterative way and a recursive way.

Let's first be clear about what we mean by reversing a linked list. Have a look at the illustration below to get a clear idea.



1 of 2

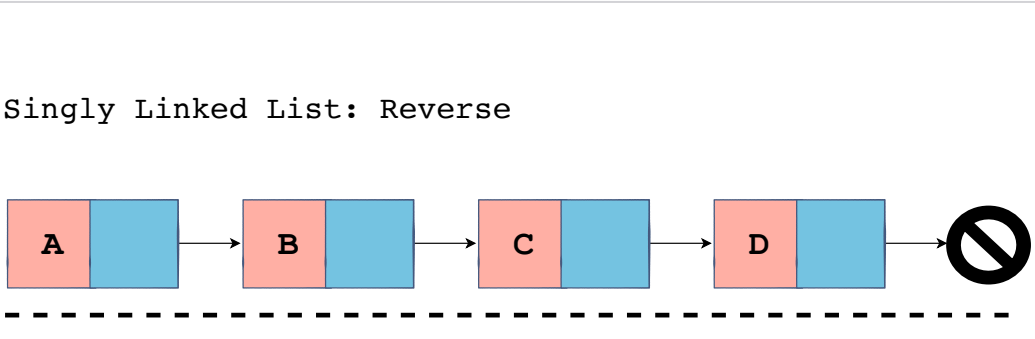


2 of 2

Let’s discuss the algorithm to reverse the linked list as depicted in the illustration above.

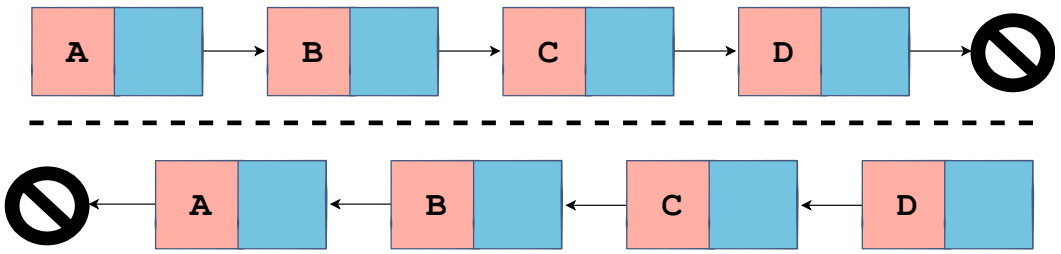
Algorithm

Before jumping to the code, let’s figure out the algorithm.



1 of 3

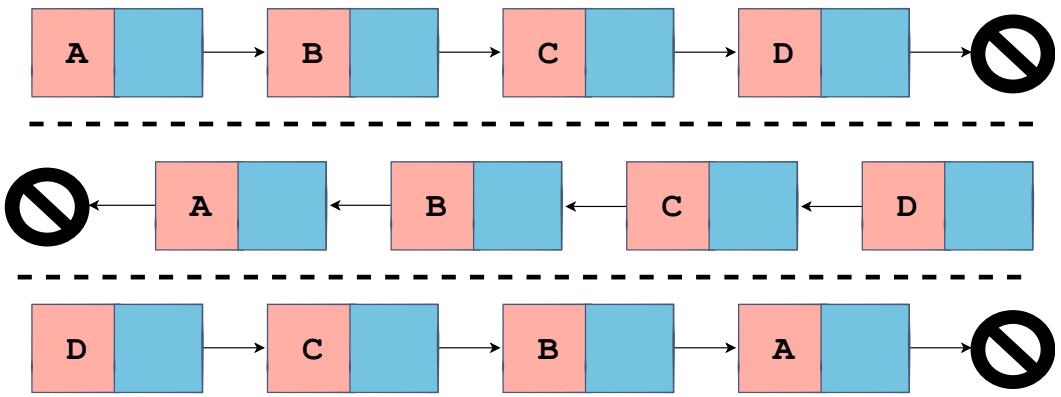
Singly Linked List: Reverse



Arrows are flipped to reverse the pointers

2 of 3

Singly Linked List: Reverse



3 of 3

Iterative Implementation

If we look at the reversal above, the key idea is that we're reversing the orientation of the arrows. For example, **node A** is initially pointing to **node B** but after we flip them, **node B** points to **node A**. The same is the

case for other nodes. We can take this key observation to solve our problem and implement the method `reverse_iterative`.

```
1  def reverse_iterative(self):
2    prev = None
3    cur = self.head
4    while cur:
5      nxt = cur.next
6      cur.next = prev
7      prev = cur
8      cur = nxt
9    self.head = prev
```

reverse_iterative(self)

The crux of the solution will be to iterate through the linked list using the *previous and current node strategy* in the *Node swap* lesson.

On **line 2** and **line 3**, we set the initial values of `prev` and `cur` which are `None` and `self.head` respectively.

Next, we set the `while` loop on **line 4** which will run until `cur` is not equal to `None`. On **line** 7 and **line 8**, we update the `prev` to `cur` and `cur` to `nxt`, i.e., `cur.next`, which help us to iterate through the linked list while keeping track of the previous and current nodes. `nxt` is used as a temporary variable to store the value of `cur.next` on **line 5** because it gets modified on **line 6**. `cur.next = prev` is the statement which does the actual work. This is because the flipping of the arrows takes place through this statement. Instead of pointing to the next node, we point the next of the current node to the previous node.

Now we just need to take care of one more thing. On **line 9**, after iterating through the linked list, `prev` is the last node in the linked list. We set `self.head` equal to the last node in the linked list. This completes our code to reverse a linked list.

Let's verify our method by making it part of the linked list implementation! You can also print out the current node and the previous node after each line in the `while` loop for a better understanding.

```
101          return 1 + self.len_recursive(node.next)
102
103      def print_helper(self, node, name):
104          if node is None:
105              print(name + ": None")
106          else:
107              print(name + ":" + node.data)
108
109      def reverse_iterative(self):
110
111          prev = None
112          cur = self.head
113          while cur:
114              nxt = cur.next
115              cur.next = prev
116              prev = cur
117              cur = nxt
118          self.head = prev
119
120  llist = LinkedList()
121  llist.append("A")
122  llist.append("B")
123  llist.append("C")
124  llist.append("D")
125
126  llist.reverse_iterative()
127
128  llist.print_list()
```

class Node and class LinkedList

Recursive Implementation

Now that we have implemented the `reverse_iterative` method, we'll turn our attention towards the recursive implementation which is going to be similar to the iterative implementation.

```
1  def reverse_recursive(self):
```

```
1  def reverse_recursive(self):
2
3    def _reverse_recursive(cur, prev):
4      if not cur:
5        return prev
6
7      nxt = cur.next
8      cur.next = prev
9      prev = cur
10     cur = nxt
11     return _reverse_recursive(cur, prev)
12
13   self.head = _reverse_recursive(cur=self.head,
```

reverse_recursive(self)

The crux of any recursive solution is as follows:

- We implement the base case.
- We agree to solve the simplest problem, which in this case is to reverse just one pair of nodes.
- We defer the remaining problem to a recursive call, which is the reversal of the rest of the linked list.

Now, let's discuss the code. In `reverse_recursive` method, we define another helper method called `_reverse_recursive` with input parameters `cur` and `prev`. On **line 4**, we write the base case for the recursive method:

```
if not cur:
  return prev
```

The base case is when we'll reach the end of the linked list and `cur` is `None`, meaning `not cur` will evaluate to true. Then we'll return `prev` from the method.

The next steps on **lines 7-10** are pretty much the same as in the iterative implementation.

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

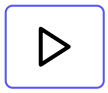
However, on **line 11**, we make a recursive call to `_reverse_recursive` method and pass `cur` and `prev` to it. This will reverse the pointers for the other nodes.

On **line 13**, we actually make a call to the helper method `_reverse_recursive` and pass `cur` as `self.head` and `prev` as `None`. Note that our base case returns `prev` which will be the last node in the linked list. Therefore, we assign `self.head` to the return argument from the `_reverse_recursive` method which will be the last node in the linked list. That concludes our implementation as the last node in the original linked list will be the head node in the reversed linked list.

Let's play around with our complete implementation and verify our method:

```
121
122             prev = cur
123             cur = nxt
124         self.head = prev
125
126     def reverse_recursive(self):
127
128         def _reverse_recursive(cur, prev):
129             if not cur:
130                 return prev
131
132             nxt = cur.next
133             cur.next = prev
134             prev = cur
135             cur = nxt
136             return _reverse_recursive(cur, prev
137
138         self.head = _reverse_recursive(cur=self.
139
140 llist = LinkedList()
141 llist.append("A")
142 llist.append("B")
143 llist.append("C")
```

```
144  llist.append("D")
145
146  llist.reverse_recursive()
147
148  llist.print_list()
```




class Node and class LinkedList

Hope everything's clear up until now. See you in the next lesson where we are going to solve another interesting problem!

← Back

Node Swap

Next →

Merge Two Sorted Linked Lists

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/reverse__singly-linked-lists__data-structures-and-algorithms-in-python)

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.

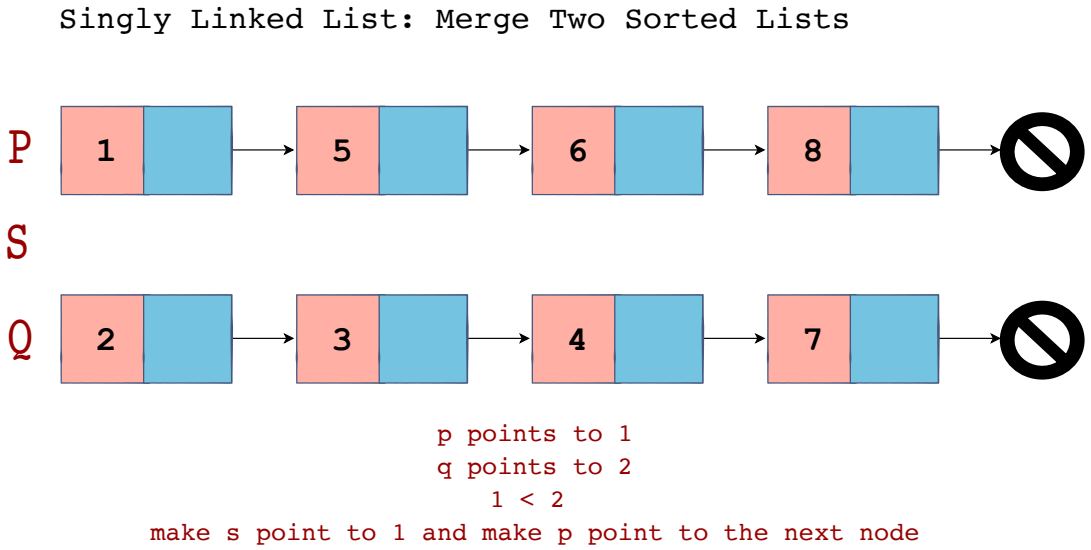
A related problem is to create a third linked list which is also sorted. In this lesson, the two linked lists given will no longer be available in their original form, and only one linked list which includes both their nodes will remain. Let's get started.

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.

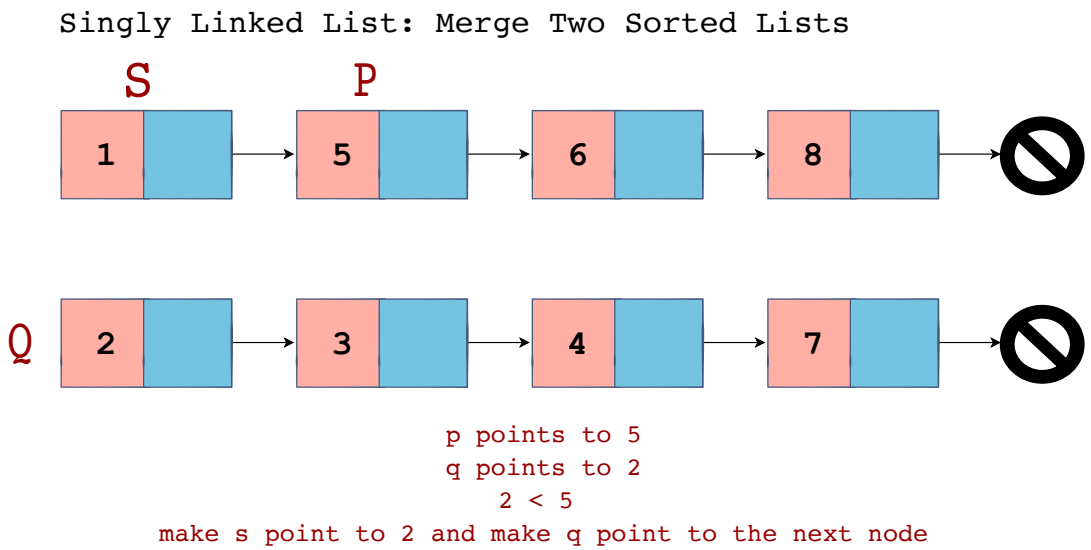
Algorithm

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

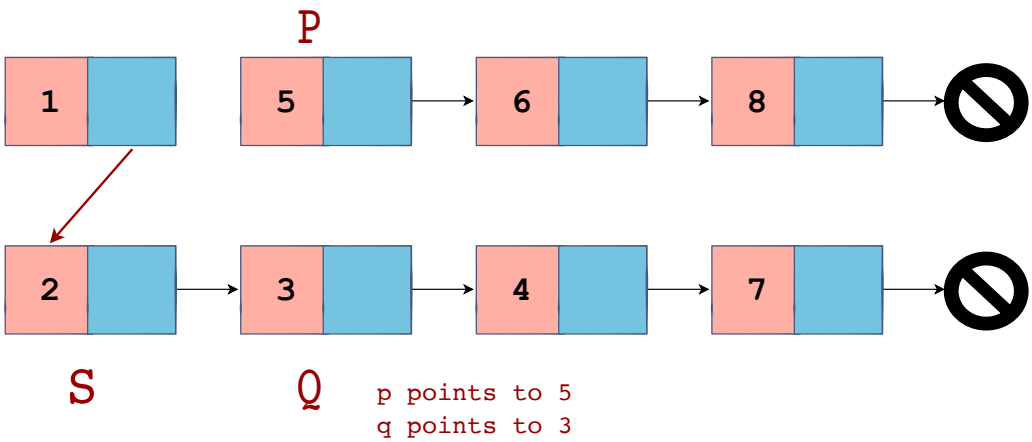


1 of 10



2 of 10

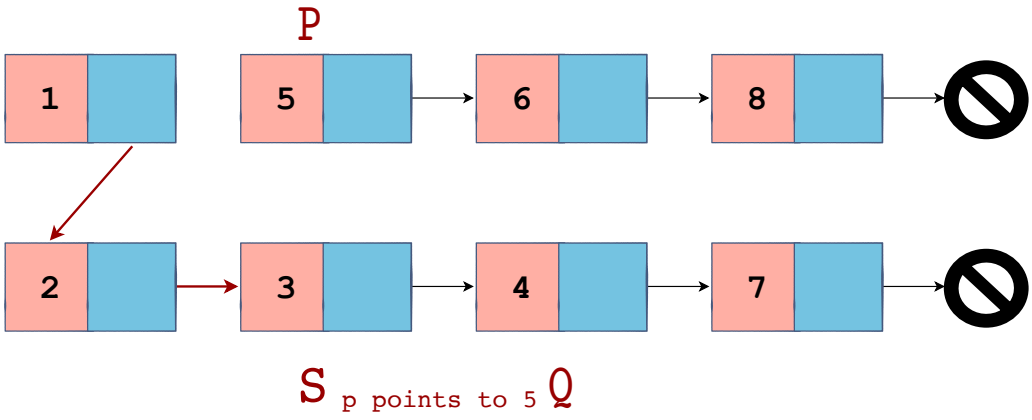
Singly Linked List: Merge Two Sorted Lists



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

3 of 10

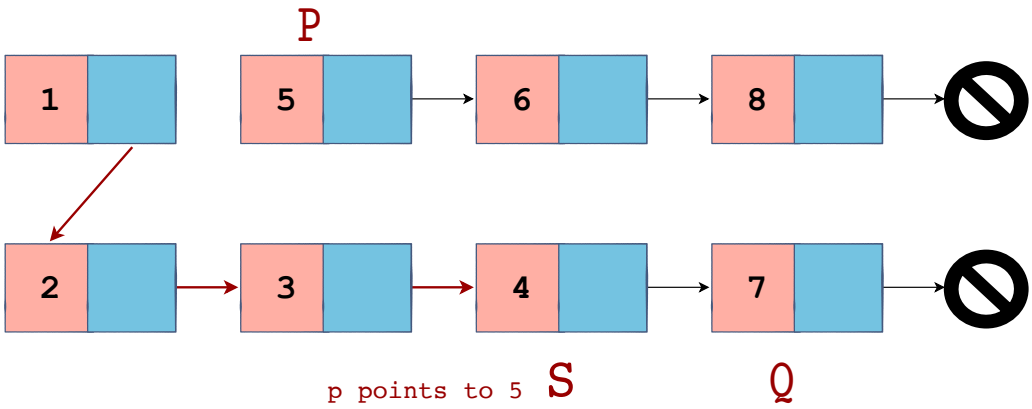
Singly Linked List: Merge Two Sorted Lists



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

4 of 10

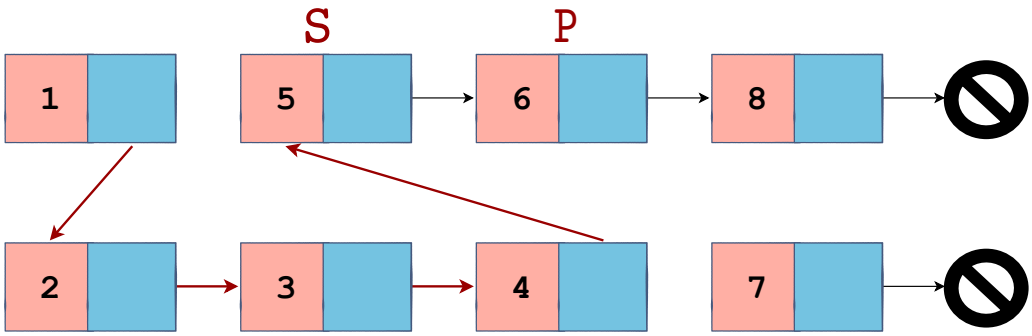
Singly Linked List: Merge Two Sorted Lists



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

5 of 10

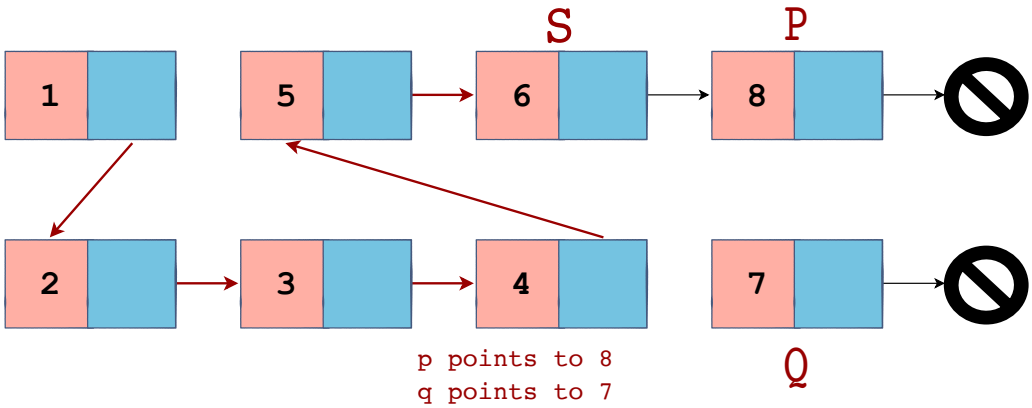
Singly Linked List: Merge Two Sorted Lists



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

6 of 10

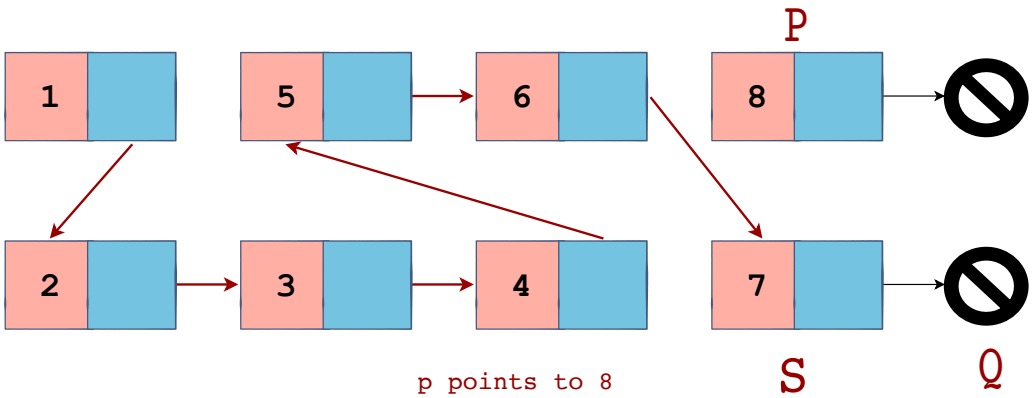
Singly Linked List: Merge Two Sorted Lists



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

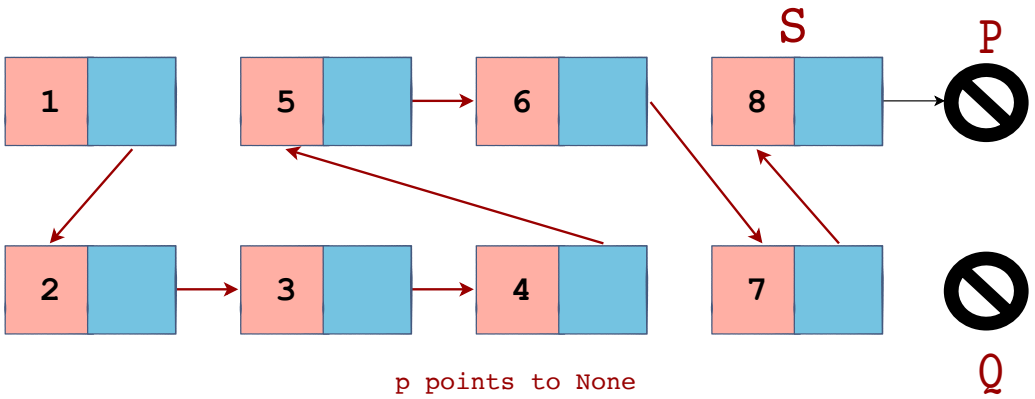
7 of 10

Singly Linked List: Merge Two Sorted Lists

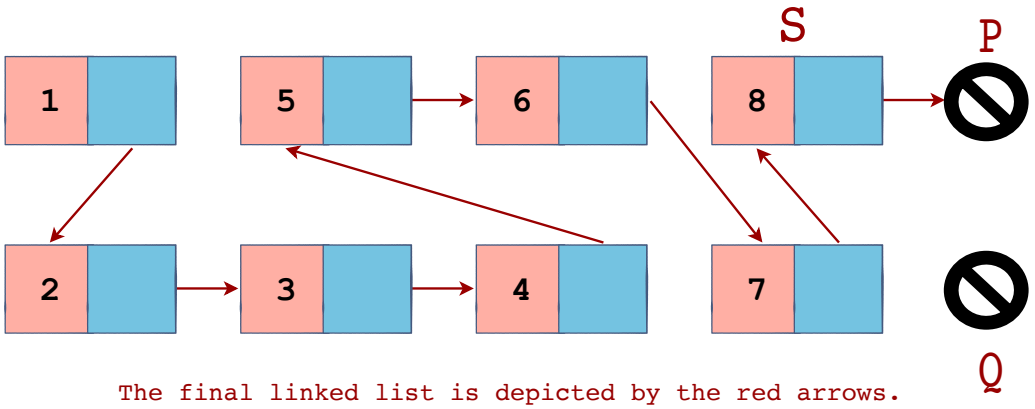


p points to 8
q points to null
make s point to 8

8 of 10



9 of 10



10 of 10

Hope you understood the algorithm.

Implementation

Now let's look at the code below:

```
1  def merge_sorted(self, llist):
```

```
 2
 3    p = self.head
 4    q = llist.head
 5    s = None
 6
 7    if not p:
 8        return q
 9    if not q:
10        return p
11
12    if p and q:
13        if p.data <= q.data:
14            s = p
15            p = s.next
16        else:
17            s = q
18            q = s.next
19        new_head = s
20    while p and q:
21        if p.data <= q.data:
22            s.next = p
23            s = p
24            p = s.next
25        else:
26            s.next = q
27            s = q
28            q = s.next
```

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

self.head = new_head
return self.head
```

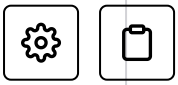
The code will reach this point after the `while` loop terminates which implies either `p` or `q` or both `p` and `q` have become `None`. We check using the conditions on **line 29** and **line 31** that we have reached the `None` for which of the linked lists. If we have reached the end of `p`, i.e., 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**).

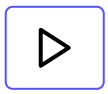
Finally, on **line 35**, we return updated `self.head (new_head)` from the method which is the head node of our merged linked list made from `llist` and the linked list on which this class method is called.

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

```
188                 s = q
189                 q = s.next
190             new_head = s
191         while p and q:
192             if p.data <= q.data:
193                 s.next = p
194                 s = p
195                 p = s.next
196             else:
197                 s.next = q
198                 s = q
199                 q = s.next
200         if not p:
201             s.next = q
```

```
202          if not q:
203              s.next = p
204
205          self.head = new_head
206          return self.head
207
208  llist_1 = LinkedList()
209  llist_2 = LinkedList()
210
211  llist_1.append(1)
212  llist_1.append(5)
213  llist_1.append(7)
214  llist_1.append(9)
215  llist_1.append(10)
```





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!

Back

Reverse

Next

Remove Duplicates

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/merge-two-sorted-linked-lists__singly-linked-lists__data-structures-and-algorithms-in-python)

Remove Duplicates

In this lesson, we will learn how to remove duplicates from a linked list.

We'll cover the following

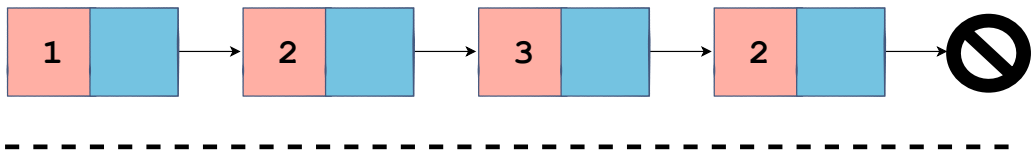
In this lesson, we will use a hash table to remove all duplicate entries from a single linked list. For instance, if our singly linked list looks like this:

```
1 – 6 – 1 – 4 – 2 – 2 – 4
```

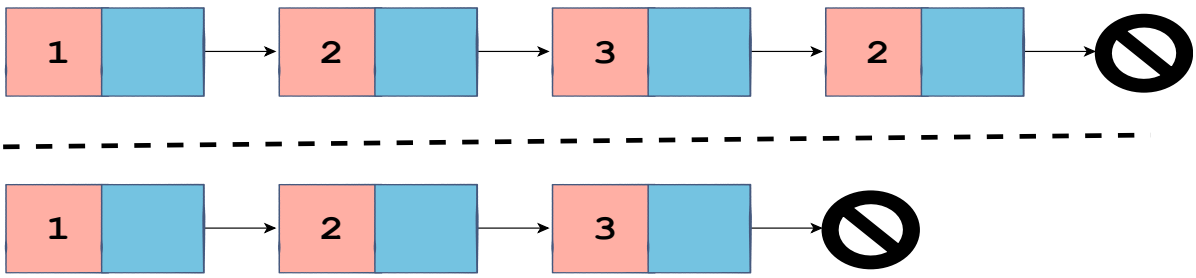
Then the desired resulting singly linked list should take the form:

```
1 – 6 – 4 – 2
```

Below is another example to illustrate the concept of removing duplicates:



1 of 2



2 of 2

Algorithm

The general approach to solve this problem is to loop through the linked list once and keep track of all the data held at each of the nodes. We can use a hash table or Python dictionary to keep track of the data elements that we encounter. For example, if we encounter `6`, we will add that to

the dictionary or hash table and move along. Now if we meet another `6` and we check for it in our dictionary or hash table, then we'll know that we already have a `6` and the current node is a duplicate.

Implementation

Let's go ahead and code a solution using the idea discussed above:

```
1  def remove_duplicates(self):
2    cur = self.head
3    prev = None
4    dup_values = dict()
5
6    while cur:
7      if cur.data in dup_values:
8        # Remove node:
9        prev.next = cur.next
10       cur = None
11     else:
12       # Have not encountered element before.
13       dup_values[cur.data] = 1
14       prev = cur
15     cur = prev.next
```

remove_duplicates(self)

Explanation

In the `remove_duplicates` method, we'll first declare two variables: `cur` and `prev` and assign them the values `self.head` and `None`, respectively. On **line 4**, we declare a Python dictionary and name it `dup_values`. Now we have to iterate through the linked list using the `while` loop on **line 6**. As you can see, the `while` loop will run until we hit the `None`. Next, we check if `cur.data` exists in `dup_values` or not. Let's first consider the case if `cur.data` does not exist in `dup_values` and move to the `else` portion on **line 11**. We add an entry using `cur.data` as a key to the dictionary and assign `1` as a value to it on **line 13**, while on **line 14**, we update the `prev` with the `cur`.

Now let's move to the case where `cur.data` actually exists from before in the `dup_values`. This is the case where we have found a duplicate! Now we need to remove the duplicate entry. Now, instead of pointing to `cur`, we make `prev.next` point to the next of `cur`, i.e., `cur.next` (**line 9**). Additionally, to completely remove the duplicate entry, i.e. `cur`, we set it equal to `None` on **line 10**.

On **line 15**, we set `cur` to `prev.next` to traverse the linked list.

We have made the `remove_duplicates` method part of the implementation of linked lists. Let's play around and check it on more test cases.

```
211         dup_values = dict()
212 
213         while cur:
214             if cur.data in dup_values:
215                 # Remove node:
216                 prev.next = cur.next
217                 cur = None
218             else:
219                 # Have not encountered element b
220                 dup_values[cur.data] = 1
221                 prev = cur
222             cur = prev.next
223 
224 llist = LinkedList()
225 llist.append(1)
226 llist.append(6)
227 llist.append(1)
228 llist.append(4)
229 llist.append(2)
230 llist.append(2)
231 llist.append(4)
232 
233 print("Original Linked List")
234 llist.print_list()
235 print("Linked List After Removing Duplicates")
236 llist.remove_duplicates()
237 llist.print_list()
238 
```

Output 0.295s

```
Original Linked List
1
6
1
4
2
2
4
Linked List After Removing Duplicates
```

I hope you understood the solution provided in this lesson. See you in the next lesson!

Back

Merge Two Sorted Linked Lists

Next

Nth-to-Last Node

Mark as Completed

Report an Issue

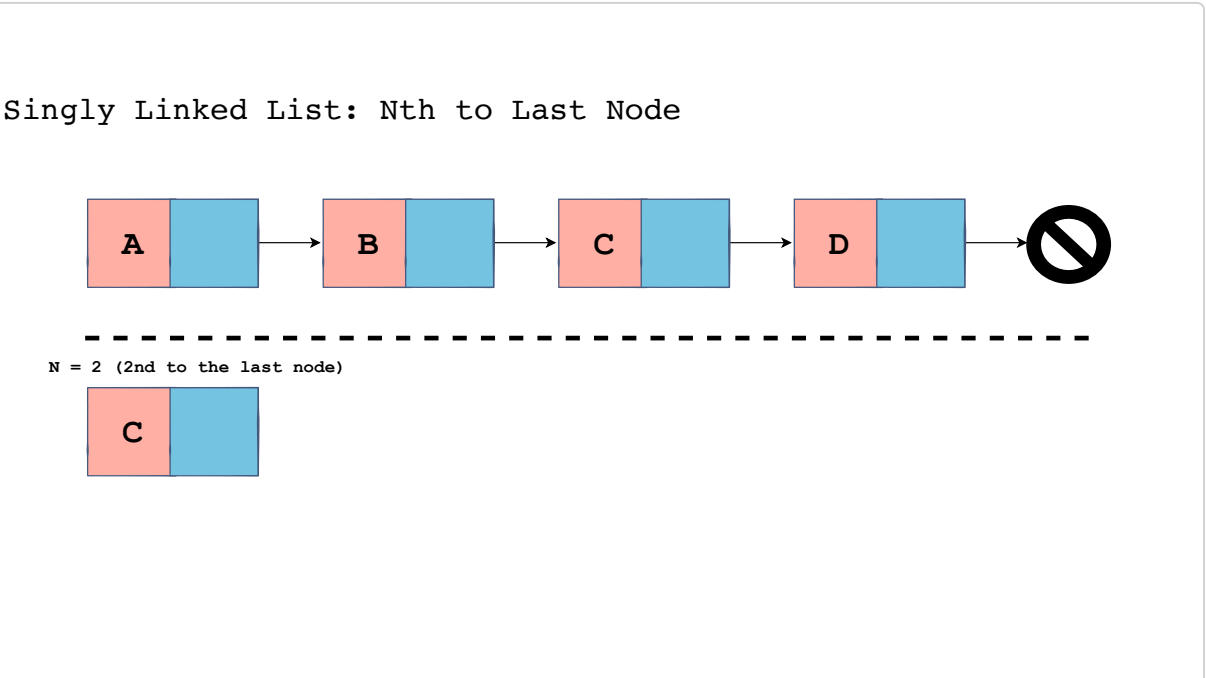
Ask a Question (https://discuss.educative.io/tag/remove-duplicates__singly-linked-lists__data-structures-and-algorithms-in-python)

Nth-to-Last Node

In this lesson, we will learn how to get the Nth-to-Last Node from a given linked list.

We'll cover the following

In this lesson, we are going to find how to get the *Nth-to -Last Node* from a linked list. First of all, we'll clarify what we mean by *Nth-to -Last Node* in the illustration below:



As you can see from the illustration above, if `N` equals `2`, we want to get the second to last node from the linked list.

We will be using two solutions to solve this problem.

Solution 1

We'll break down this solution in two simple steps:

1. Calculate the length of the linked list.
2. Count down from the total length until `n` is reached.

For example, if we have a linked list of length four, then we'll begin from the head node and decrement the calculated length of the linked list by one as we traverse each node in the linked list. We'll only stop on the node when our count becomes equal to `n`.

Implementation

Let's try implementing this solution in Python:

```
def print_nth_from_last(self, n):
  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
```

print_nth_from_last(self, n)

Explanation

The method `print_nth_from_last` only takes in `n` as an input parameter. You are already familiar with how to calculate the length of a linked list from a previous lesson. We'll use the class method `self.len_iterative()` that we have implemented before to calculate the length of the linked list and store it in the variable named `total_len` on **line 2**.

So far, we have completed step 1 of solution 1. Let's move on to the second step. `cur` is initialized to `self.head` on **line 4**. Next, we have a `while` loop on **line 5** which will traverse the entire linked list or stop after traversing n nodes. In the body of the `while` loop, we check if `total_len` equals `n` which is our primary goal for this method. If `total_len` equals `n`, we print the `data` of the current node (`cur`) and return the data of the current node from the method. Otherwise, we decrement `1` from `total_len` on **line 9** and update `cur` to the next node on **line 10** to traverse the linked list.

If, however, we reach the end of the linked list but `total_len` never becomes equal to `n`, then we handle this on **lines 11-12**. In this case, we check if `cur` is `None` and we return from the method.

Solution 2

That was all about Solution 1. Let's proceed with Solution 2, which can be described as follows:

There will be a total of two pointers `p` and `q`:

- `p` will point to the head node.
- `q` will point `n` nodes beyond head node.

Next, we'll move these pointers along with the linked list one node at a time. When `q` will reach `None`, we'll check where `p` is pointing, as that is the node we want.

Implementation

Let's make it clearer by implementing it in Python:

```
1  def print_nth_from_last(self, n):
2      p = self.head
3      q = self.head
4
5      count = 0
6      while q:
7          count += 1
8          if(count>=n):
9              break
10         q = q.next
11
12     if not q:
13         print(str(n) + " is greater than the nur
14         return
15
16     while p and q.next:
17         p = p.next
18         q = q.next
19     return p.data
```

print_nth_from_last(self, n)

Explanation

We initialize `p` and `q` to `self.head` on **line 2** and **line 3** respectively. According to the algorithm, we have to make `q` point to `n` nodes beyond the head. Therefore, we initialize `count` to `0` on **line 5** and using the `while` loop on **line 6**, we keep updating `q` to the next node (**line 10**) until and unless `count`, which is being incremented in each iteration (**line 7**), becomes equal to or greater than `n`. Additionally, the `while` loop will terminate if `q` reaches `None`. To handle this case, we put up a condition on **line 12** which checks if `q` is `None` or not. If `q` is `None`, then this implies that `n` is greater than the length of the linked list and getting the *Nth-to-last* node is not possible. We return from the method on **line 14**.

Now let's jump to the main part of the implementation. In the `while loop` on **line 16**, we keep updating `p` and `q` to the next nodes. This `loop` will terminate when either `p` or `q.next` equals `None`. As a result, we return `p.data` which will be the *Nth-to-last* node according to our algorithm.

Both the solutions have been made part of the `LinkedList` class in the coding widget below. You can call the solution of your choice by passing in `1` or `2` as `method` to `print_nth_from_last(self, n, method)`. Feel free to play around with any of the methods in the `LinkedList` implementation below:

```
237
238         elif method == 2:
239             # Method 2:
240             p = self.head
241             q = self.head
242
243             count = 0
244             while q:
245                 count += 1
246                 if(count>=n):
247                     break
248                 q = q.next
249
250             if not q:
251                 print(str(n) + " is greater than
252                 return
253
254             while p and q.next:
255                 p = p.next
256                 q = q.next
257             return p.data
258
259 llist = LinkedList()
260 llist.append("A")
261 llist.append("B")
262 llist.append("C")
263 llist.append("D")
264
```

Hope you are clear about both the solutions. Let's move on to another problem about singly linked lists in the next lesson. See you there!

Back

Remove Duplicates

Next

Count Occurrences

Mark as Completed

Ask a Ouestion

Count Occurrences

In this lesson, we will learn how to count occurrences of a data element in a linked list.

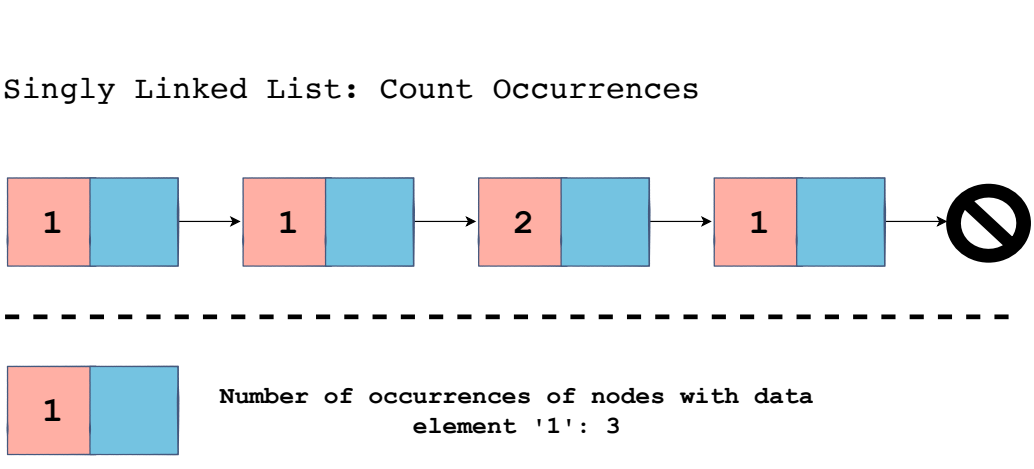
We'll cover the following

In this lesson, we investigate how to count the occurrence of nodes with a specified data element. We will consider how one may solve this problem in both an iterative and recursive manner, and we will code the solution to both of these approaches in Python.

As an example, have a look at the illustration below where we have a linked list with the following elements:

```
1 – 1 – 2 – 1
```

You can see that the number of occurrences of `1` in the linked list is `3`.



Iterative Implementation

Our approach to iteratively solving this problem is straightforward. We'll traverse the linked list and count the occurrences as we go along. Let's go ahead and code it in Python!

```
1  def count_occurences_iterative(self, data):
2    count = 0
3    cur = self.head
4    while cur:
5        if cur.data == data:
6            count += 1
7        cur = cur.next
8    return count
```

count_occurences_iterative(self, data)

Explanation

`count_occurences_iterative` takes in `data` as one of the input parameters. On **line 2** and **line 3**, `count` and `cur` are initialized to `0` and `self.head` respectively. `count` will keep count of the number of

occurrences of `data` while `cur` is used for the linked list traversal. The `while` loop on **line 4** runs until `cur` becomes equal to `None`. `cur` is updated to `cur.next` in each iteration (**line** 7).

On **line 5**, there is a simple check to see if `cur.data` is equal to `data`. If the data of the current node is equal to `data` that we are looking for, `count` is incremented by `1`. This is how the number of occurrences will be counted.

On **line 8**, `count` after being updated in the `while` loop, is returned from the method.

Hope the iterative implementation is clear.

Recursive Implementation

Let's turn to the recursive implementation now.

```
1  def count_occurences_recursive(self, node, data
2    if not node:
3        return 0
4    if node.data == data:
5        return 1 + self.count_occurences_recursive
6    else:
7        return self.count_occurences_recursive(no
```

count_occurences_recursive(self, node, data)

Explanation

The general idea is nearly the same except that it has been implemented recursively. The `count_occurences_recursive` method takes in `self`, `node`, and `data` as input parameters. `self` is passed because it is a class method. `node` will be the current node that we are on while `data` is the data that we have to check the occurrences of.

The base case written on **lines 2-3** is an empty linked list. An empty linked list has no number of occurrences of any value. If we hit a `None` node, we return `0` from the method.

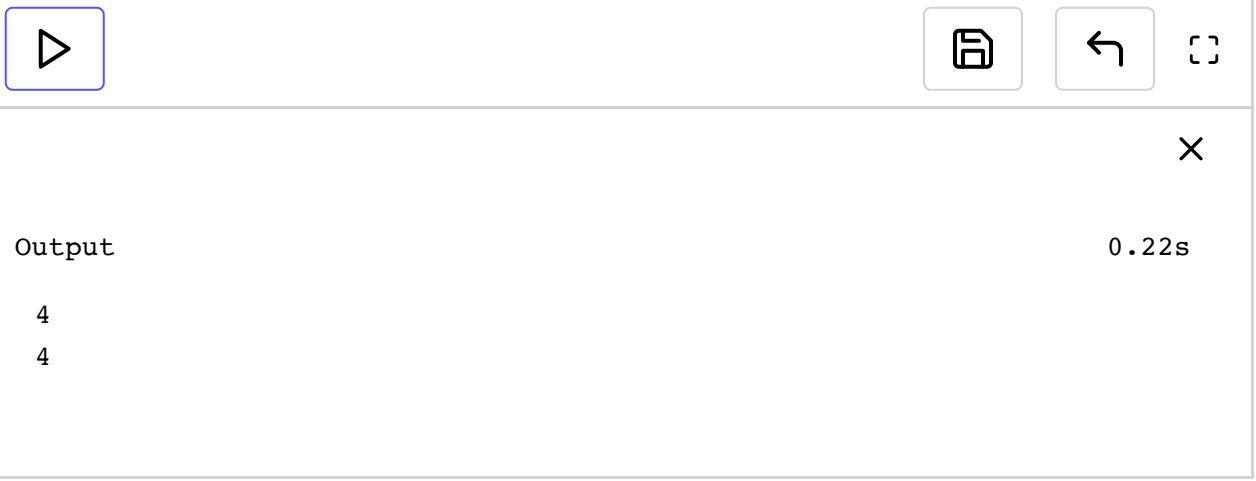
However, if the current node (`node`) is not `None` , then we check if `node.data` is equal to `data` . If it is, then we make a recursive call to `count_occurences_recursive` and pass the next node of the current node (`node.next`) and `data` to it. Also, we add `1` to whatever will be returned from the recursive call and return it from the method. This is because we have to make the occurrence of the current node count as `node.data` match the `data` . However, if `node.data` does not match `data` , we don't add `1` and simply return whatever is returned from `self.count_occurences_recursive(node.next, data)` (**line** 7).

Overall, for a non-empty linked list, we only look at the first node in the linked list. If it has that value, we've found one match. We let the recursive calls count the number of occurrences of the desired value in the remaining linked list, i.e., one that starts at `cur.next` . We either add one to it, or we don't.

Let's test both the implementations in the code widget below. We pass the head node as an argument to the recursive implementation.

```
268     def count_occurences_recursive(self, node, d
269         if not node:
270             return 0
271         if node.data == data:
272             return 1 + self.count_occurences_re
273         else:
274             return self.count_occurences_recurs
275 
276 
277 llist = LinkedList()
278 llist.append(1)
279 llist.append(2)
280 llist.append(3)
281 llist.append(4)
282 llist.append(5)
283 llist.append(6)
284 
285 llist_2 = LinkedList()
```

```
286  llist_2.append(1)
287  llist_2.append(2)
288  llist_2.append(1)
289  llist_2.append(3)
290  llist_2.append(1)
291  llist_2.append(4)
292  llist_2.append(1)
293  print(llist_2.count_occurences_iterative(1))
294  print(llist_2.count_occurences_recursive(llist_
```



```
Output                                                    0.22s

4
4
```

Hope you were able to understand the lesson! See you in the next one.

Back

Nth-to-Last Node

Next

Rotate

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/count-occurrences__singly-linked-lists__data-structures-and-algorithms-in-python)

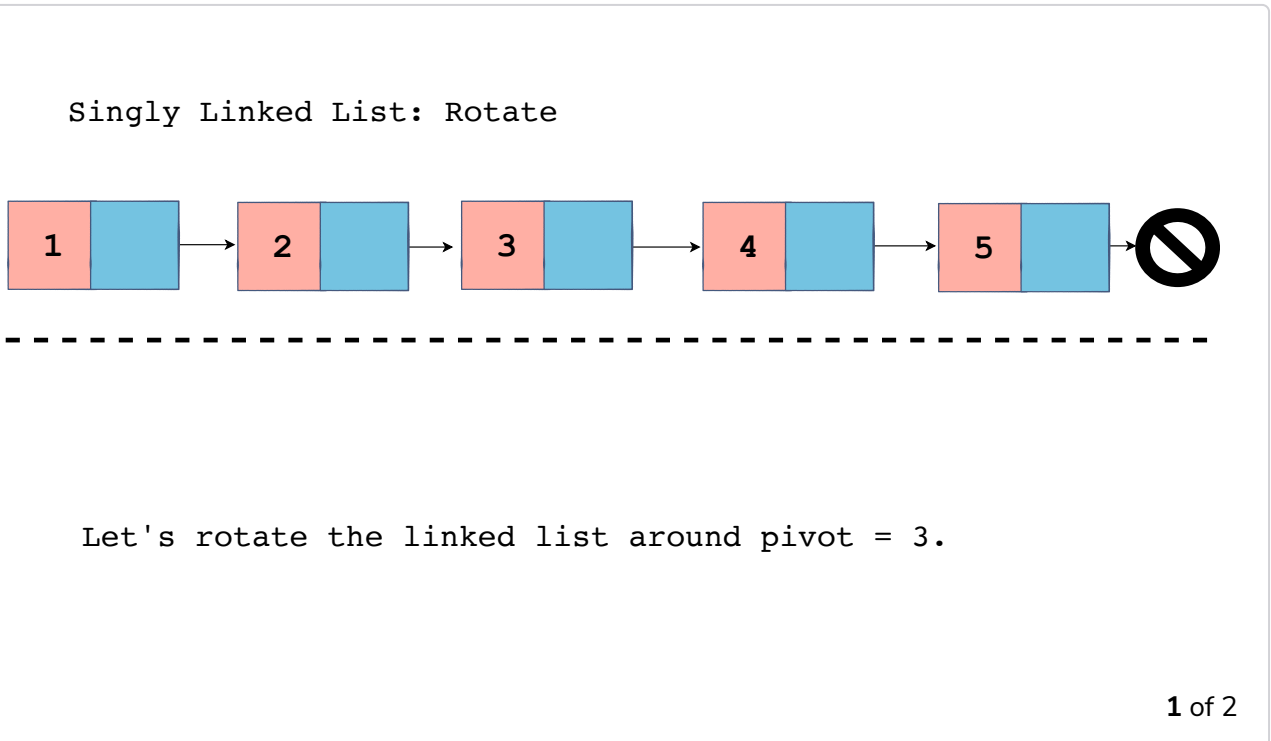
Rotate

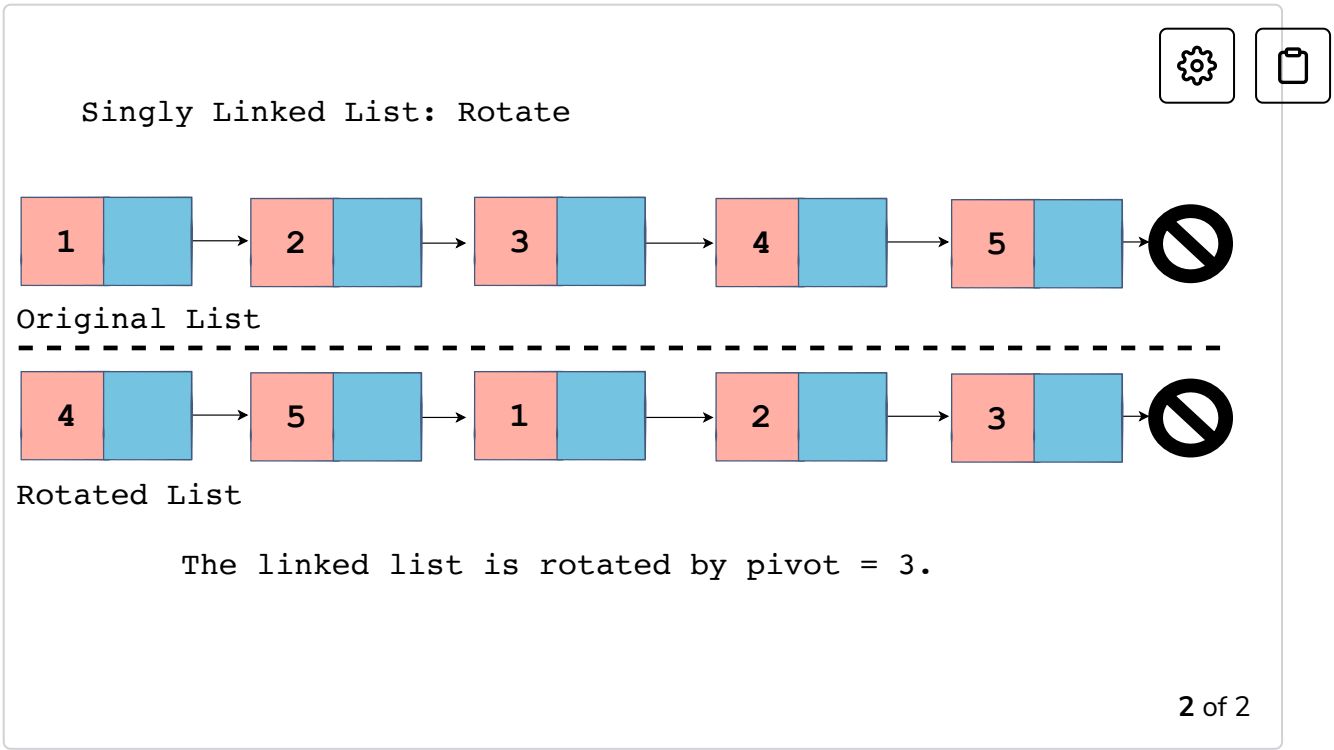
In this lesson, we will learn how to rotate a linked list.

We'll cover the following

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.

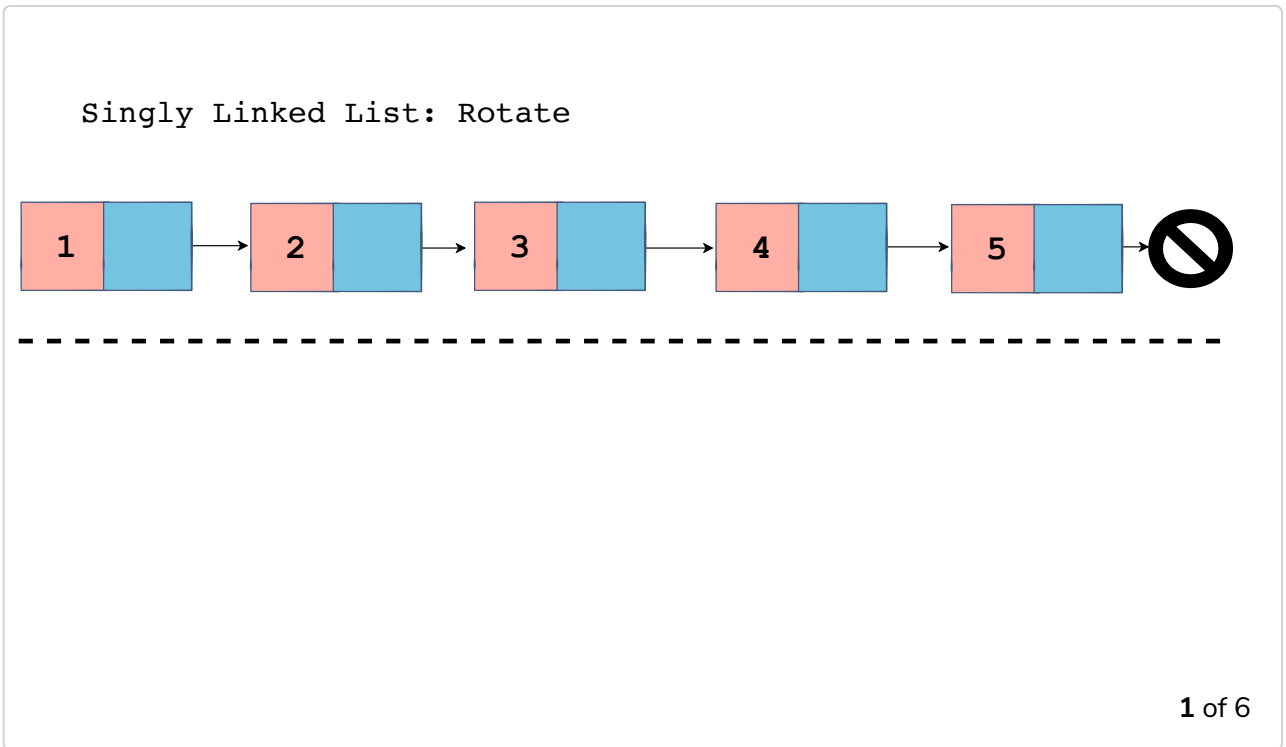
The illustration below will help you understand how to rotate the nodes of a singly linked list.



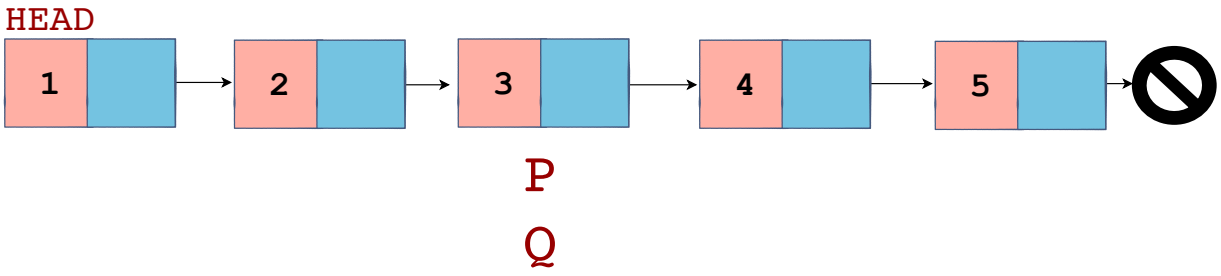


Algorithm

The algorithm to solve this problem has been illustrated below:



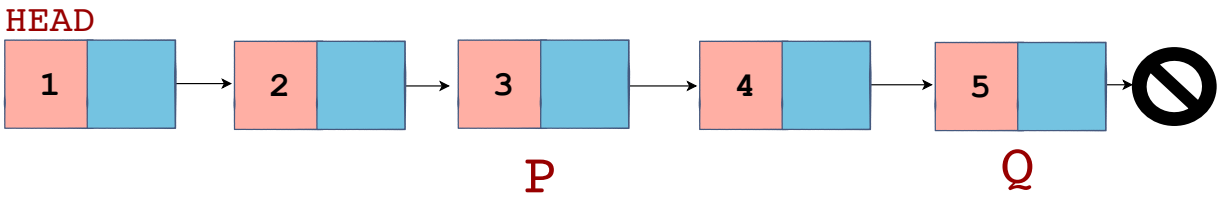
Singly Linked List: Rotate



pivot = 3

2 of 6

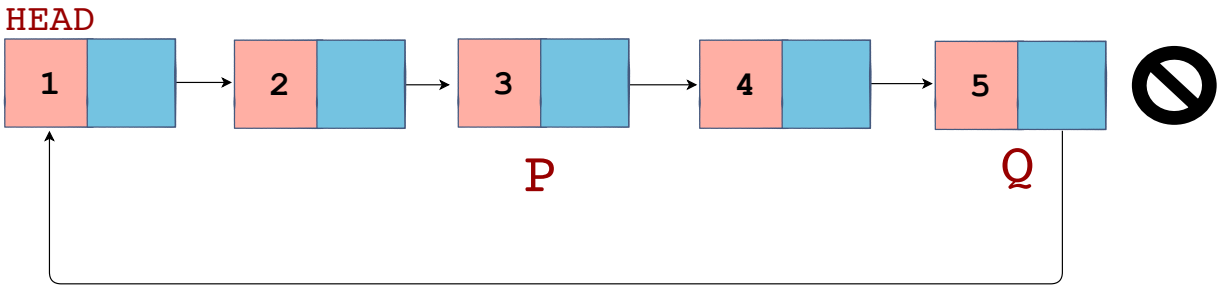
Singly Linked List: Rotate



pivot = 3

3 of 6

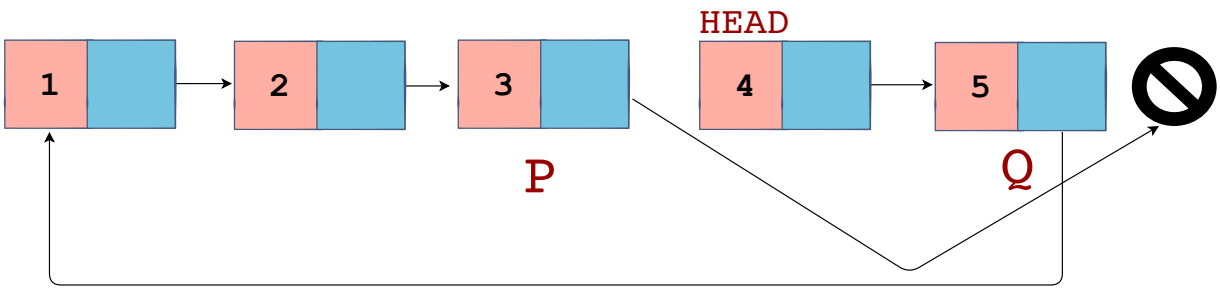
Singly Linked List: Rotate



pivot = 3

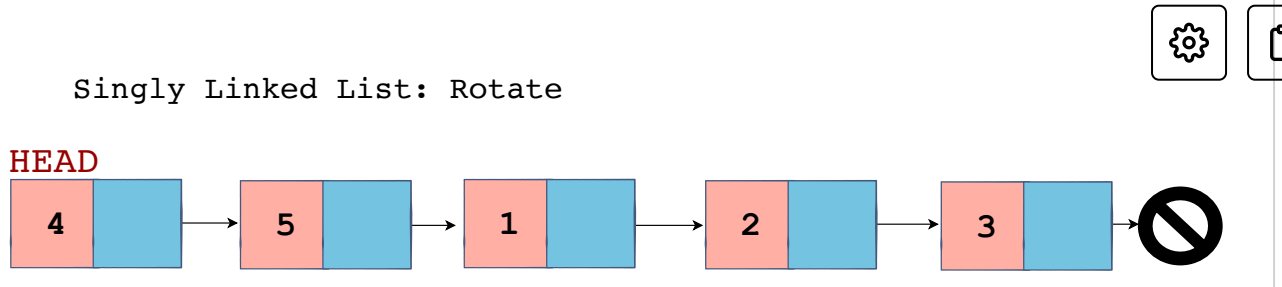
4 of 6

Singly Linked List: Rotate



pivot = 3

5 of 6



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.

```
1  def rotate(self, k):
2    if self.head and self.head.next:
3      p = self.head
4      q = self.head
5      prev = None
```

```
6       count = 0
7
8       while p and count < k:
9           prev = p
10          p = p.next
11          q = q.next
12          count += 1
13      p = prev
14      while q:
15          prev = q
16          q = q.next
17      q = prev
18
19      q.next = self.head
20      self.head = p.next
21      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.

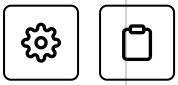
After breaking out of the `while` loop, we set `prev` equal to `p` on **line 13** to position our first pointer correctly. For the second pointer (`q`), we have to make it point to the last element in the linked list. As a result, we set up another `while` loop on **lines 14-16** where we move `q` along 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:

```
266             while p and count < k:
267                 prev = p
268                 p = p.next
269                 q = q.next
270                 count += 1
271             p = prev
272             while q:
273                 prev = q
274                 q = q.next
275             q = prev
276 
277             q.next = self.head
278             self.head = p.next
279             p.next = None
280 
281 
282 
283 llist = LinkedList()
```

```
284  llist.append(1)
285  llist.append(2)
286  llist.append(3)
287  llist.append(4)
288  llist.append(5)
289  llist.append(6)
290
291  llist.rotate(4)
292  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!

Back

Count Occurrences

Next

Is Palindrome

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/rotate__singly-linked-lists__data-structures-and-algorithms-in-python)

Is Palindrome

In this lesson, we'll learn how to determine whether a singly linked list is a palindrome or not.

We'll cover the following

In this lesson, we investigate how to determine if a singly linked list is a palindrome, that is if the data held at each of the nodes in the linked list can be read forward from the head or backward from the tail to generate the same content. We will code the solution to both of these approaches in Python.

First of all, let’s learn what a palindrome is. A palindrome is a word or a sequence that is the same from the front as from the back. For instance, *racecar* and *radar* are palindromes. Examples of non-palindromes are *ABC*, *hello*, and *test*.

For this lesson, we’ll solve the *Is Palindrome* problem in Python by using three solutions.

Now let’s go ahead and check out *Solution 1*:

Solution 1: Using a string

Implementation

```
1  def is_palindrome(self):
2    # Solution 1:
3    s = ""
4    p = self.head
5    while p:
6      s += p.data
7      p = p.next
8    return s == s[::-1]
```

is_palindrome(self)

Explanation

The first solution makes use of a string, `s`, which is initialized to an empty string on **line 3**. Then we initialize `p` to `self.head` on **line 4**. On **line 5**, a `while` loop is set which will run until `p` becomes `None`. In the `while` loop, the data of the current node is concatenated to the string `s` on **line 6** while we keep updating `p` to its next node on **line 7**. Finally, we return a boolean value based on the following condition on **line 8**:

```
s == s[::-1]
```

The condition above checks if `s` is equivalent to the reverse of `s` and returns `True` or `False` accordingly. `s[::-1]` is just a shorthand in Python for reversing a string. As you can see the first method is fairly simple, let's move on to the next solution.

Solution 2: Using a stack

The second method is a little bit different than the first method, but it has a similar spirit. Solution 2 will involve a stack, but, we will not make use of a stack per se. Instead, we'll use a Python list and its `append` and `pop`

methods. These operations are similar to the `push` and `pop` methods of the stack data structure.

In this method, we'll move along the list, but instead of concatenating onto a string, we'll push onto the stack.

Implementation

Let's code this up!

```
1  def is_palindrome(self):
2    # Solution 2:
3    p = self.head
4    s = []
5    while p:
6      s.append(p.data)
7      p = p.next
8    p = self.head
9    while p:
10     data = s.pop()
11     if p.data != data:
12       return False
13     p = p.next
14   return True
```

is_palindrome(self)

Explanation

On **lines 3-4**, we declare `p` and `s` as `self.head` and `[]` respectively. `s` is a Python list, but we'll use it as a stack. On **line 5**, we set up a `while` loop in which we append (push) the data of every node onto `s` (**line 6**) by traversing the linked list with the help of `p` which is updated to its next node in every iteration (**line** 7).

On **line 8**, we assign `p` to `self.head` as we want to traverse the list again using the `while` loop on **line 9**. In this `while` loop, we pop from `s` and save it into the `data` variable. Note that the `pop` method will return us the latest element pushed in. Therefore, we are going to get values starting

from the last item of the linked list. We are traversing the list from the head while `s` gives us values from the end. We make use of this arrangement and check if `p.data` is equal to `data` on **line 11**. If it is, we continue to the next node by updating `p` to `p.next` on **line 13**. On the other hand, if `p.data` is not equal to `data` in any iteration, we'll return `False` from the method (**line 12**) as we've hit a case where the data from the front is not the same as the data when reading from the back. If we traverse the entire linked list and do not encounter any such case, we'll return `True` from the method after the `while` loop on **line 14**.

This was all about the second solution. Let's move on to the final solution!

Solution 3: Using Two Pointers

In this method, we'll have two pointers `p` and `q`. `p` will initially point to the head of the list and `q` to the last node of the list. Next, we'll check the data elements at each of these nodes that are being pointed to by `p` and `q` and see if they are equal to each other. If they are, we'll progress `p` by one, and we'll move `q` by one in reverse until we get to a point where `p` and `q` either meet or essentially can't move any further without crossing.

Implementation

Let's jump to the coding part now.

```
1  def is_palindrome(self):
2    if self.head:
3      p = self.head
4      q = self.head
5      prev = []
6
7      i = 0
8      while q:
9        prev.append(q)
10       q = q.next
11       i += 1
12     q = prev[i-1]
13
14     count = 1
15
```

```
16      while count <= i//2 + 1:
17        if prev[-count].data != p.data:
18          return False
19        p = p.next
20        count += 1
21      return True
22    else:
23      return True
```

is_palindrome(self)

Explanation

On **line 2**, we check if the linked list is empty or not. If it is, execution jumps to **lines 22-23** and `True` is returned from the method. Otherwise, we initialize `p` and `q` by making them point to the head node on **lines 3-4** while another variable `prev` is initialized to an empty list on **line 5**. `prev` will help us in making `q` move to the previous node in each iteration in the `while` loop on **line 16**.

On **line** 7, `i` is initialized to `0`. The `while` loop on **line 8** will help us in making `q` move to the end of the linked list. While doing so, we also append the nodes to `prev` list on **line 9** and update `q` to `q.next` to traverse the linked list on **line 10**. In the same loop, which runs until `q` is `None`, we increment `i` by `1`. `i` helps us in indexing on **line 12** where we set `q` to the last node in the `prev` list where `i-1` is the index of the last node.

Now that both `p` and `q` are rightly positioned, let's jump to the last `while` loop on **line 16**. The condition for the `while` loop makes use of the `count` variable which is initialized to `1` on **line 14**. The `while` loop will run until `count` is less than or equal to `i//2 + 1` (approximately half of the list). In the `while` loop on **line 17**, we check if `prev[-count].data` is not equal to `p.data`. Let's first see what we mean by `prev[-count].data`. It is basically the data of a node present in the `prev` list. The node depends on the index `-count` which, for the first iteration, will be `-1`, i.e., the last

element in the `prev` list. This implies that we are comparing elements from the end of the linked list to the start of the list and checking if they are the same and form a palindrome. If they are not the same, we return `False` from the method on **line 18**. If they are the same, we proceed to the next iteration and update `p` to `p.next` on **line 19**. Additionally, we increment `count` by `1` on **line 20** so that the second iteration compares the second to last element with the second element and so on. If in all the iterations, we do not encounter the case where the compared elements are not the same, and we exit the `while` loop, we return `True` on **line 21** to confirm the linked list as a palindrome.

Below we have the `is_palindrome` method as part of the `LinkedList` class. It takes an integer as an input which specifies the solution we want to use. For instance, `is_palindrome(1)` will execute solution 1 to solve the problem. We verify the methods on two cases `radar` and `abc`. `is_palindrome` should return `True` in the first case while it should return `False` in the second case. Go ahead and use the solution of your choice to play with the methods!

```
326             prev = []
327 
328             i = 0
329             while q:
330                 prev.append(q)
331                 q = q.next
332                 i += 1
333             q = prev[i-1]
334 
335             count = 1
336 
337             while count <= i//2 + 1:
338                 if prev[-count] != p.data:
339                     return False
340                 p = p.next
341                 count += 1
342             return True
343         else:
344             return True
345 
346     def is_palindrome(self,method):
347         if method == 1:
348             return self.is_palindrome_1()
```

```
349         elif method == 2:
350             return self.is_palindrome_2()
351         elif method == 3:
352             return self.is_palindrome_3()
353
```

```
Output                                                    0.25s

 True
 True
 True
 False
 False
 False
```

I hope you were able to comprehend and appreciate the three solutions. Next up we have a challenge for you, so brace yourself!

Back

Rotate

Next

Exercise: Move Tail to Head

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/is-palindrome__singly-linked-lists__data-structures-and-algorithms-in-python)

Exercise: Move Tail to Head

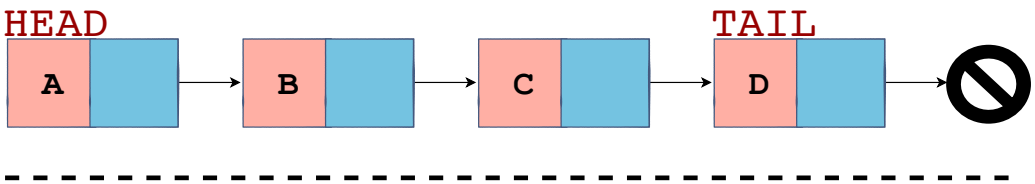
Challenge yourself with an exercise in which you'll move the tail node to the head node in a linked list.

We'll cover the following

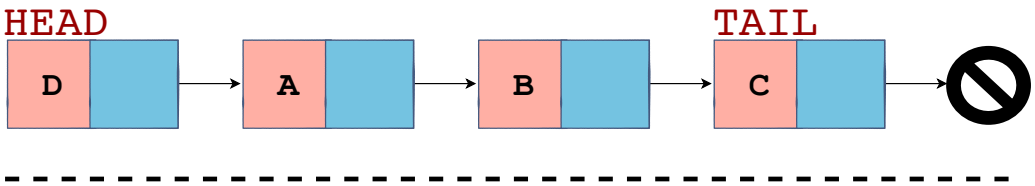
Problem

You are required to solve the `Move Tail to Head` problem in a linked list. In this exercise, you are supposed to move the tail (or last) node in a singly linked list to the front of the linked list so that it becomes the new head of the linked list.

For example, in the illustration below, the tail node (`D`) moves to the start of the linked list and replaces the head node (`A`).



1 of 2



2 of 2

Coding Time!

In the code below, the `move_tail_to_head` is a class method of the `LinkedList` class. You cannot see the rest of the code as it is hidden. As `move_tail_to_head` is a class method, please make sure that you don't

change the indentation of the code provided to you. You are required to write your solution under the method prototype.

For this exercise, you are not required to return anything from the method. Just modify the linked list so that the tail node is moved to the head node, the previous head node shifts ahead, and the tail node is updated to point to `None`. Remove the `pass` statement if you start implementing your solution.

Good luck!

```
1      def move_tail_to_head(self):
2          p = self.head
3          while p.next:
4              prev = p
5              p = p.next
6          p.next = self.head
7          self.head = p
8          prev.next = None
```

Show Results | Show Console

0.2s

3 of 3 Tests Passed

Input	Expected Output	Actual
move_tail_to_head(A,B,C,D)	D,A,B,C	D,A,
move_tail_to_head(3,2,8,5,11)	11,3,2,8,5	11,3,
move_tail_to_head(1,2,3,4,5,6,7,8,9,0)	0,1,2,3,4,5,6,7,8,9	0,1,2,3,4,

Back Next

Is Palindrome

Solution Review: Move Tail to Head

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/exercise-move-tail-to-head__singly-linked-lists__data-structures-and-algorithms-in-python)

Solution Review: Move Tail to Head

This lesson contains the solution review for the challenge of moving the tail node of a linked list to the head.

We'll cover the following

For this problem, we will use two pointers where one will keep track of the last node of the linked list, and the other will point to the second-to-last node of the linked list. Let's have a look at the code below.

Implementation

```
def move_tail_to_head(self):
  if self.head and self.head.next:
    last = self.head
    second_to_last = None
    while last.next:
        second_to_last = last
        last = last.next
    last.next = self.head
    second_to_last.next = None
    self.head = last
```

Explanation

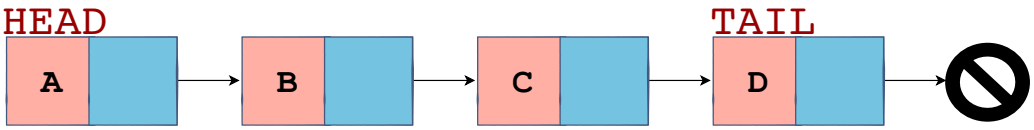
Let's go over a line by line explanation of the solution above.

Line 2 ensures that the code proceeds to **line 3** if there is more than one element in the linked list. This implies `self.head` and `self.head.next` is not `None`. We initialize `last` and `second_to_last` to `self.head` and `None` on **lines 3-4**. Now we have to make them point to what they are supposed to point to, i.e., `last` should point to the last node while `second_to_last` should point to the second to last node in a linked list. Therefore, we traverse the linked list using the `while` loop on **line 5**. The `while` loop will run until `last.next` becomes `None` which implies that `last` is the last node in the linked list. Before updating `last` to `last.next` on **line 7**, we keep updating `second_to_last` to `last` on **line 6** so that in the last iteration, when `last` is the last node in the linked list, `second_to_last` will be the second to last node in the linked list.

Now that the two pointers are rightly positioned, we have to change a few pointers to complete our solution. Therefore, `last.next` which was previously pointing to `None` is pointed to `self.head` on **line 8**. This makes a circular linked list where the last node points to the first element of the linked list. To make the linked list linear, we make `second_to_last.next` point to `None` on **line 9**. In this way, we have updated the tail pointer of the linked list. At this stage, all that is left is to update the head pointer which we do on **line 10**, and set `self.head` equal to `last`. This completes our solution when we have to move the tail node to the head.

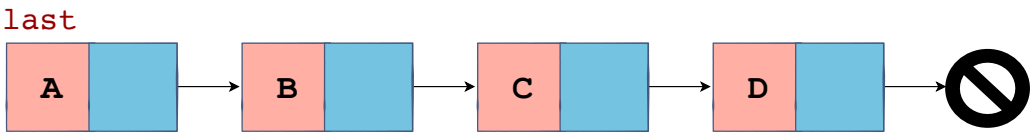
You can visualize the solution above with the help of the slides below:

Singly Linked List: Move Tail to Head



1 of 17

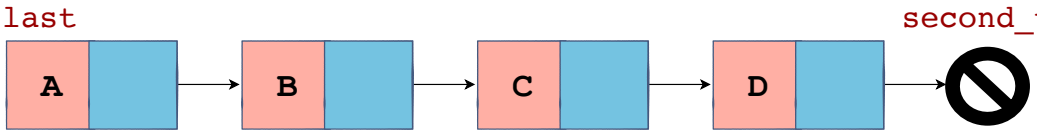
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
    last = self.head
    second_to_last = None
    while last.next:
        second_to_last = last
        last = last.next
    last.next = self.head
    second_to_last.next = None
    self.head = last
```

2 of 17

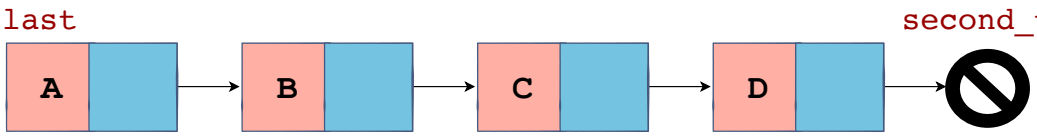
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

3 of 17

Singly Linked List: Move Tail to Head



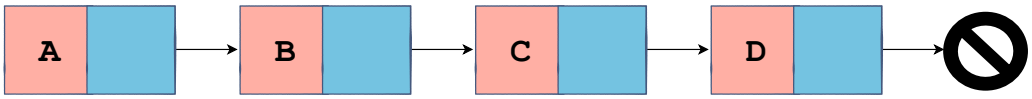

```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

4 of 17

Singly Linked List: Move Tail to Head

second_to_last

last

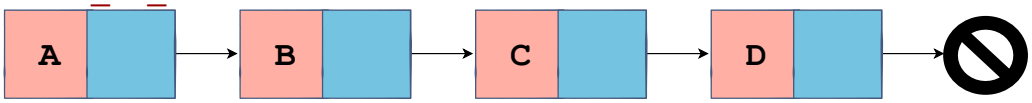



```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

5 of 17

Singly Linked List: Move Tail to Head

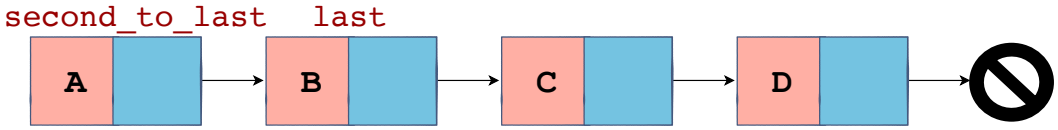
second_to_last last




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

6 of 17

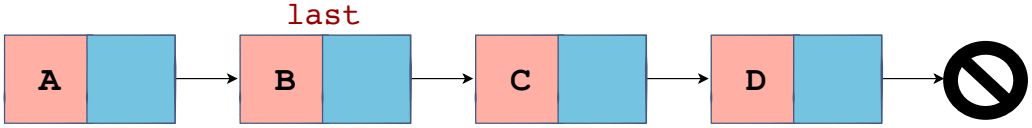
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

7 of 17

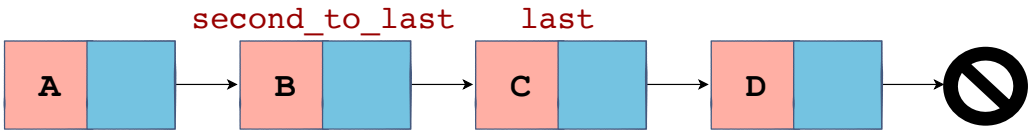
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

8 of 17

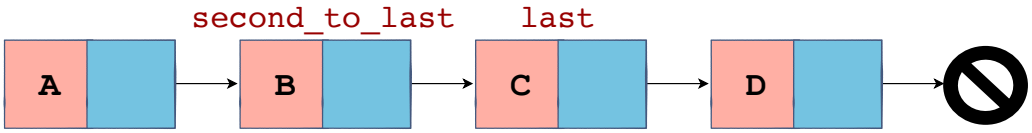
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

9 of 17

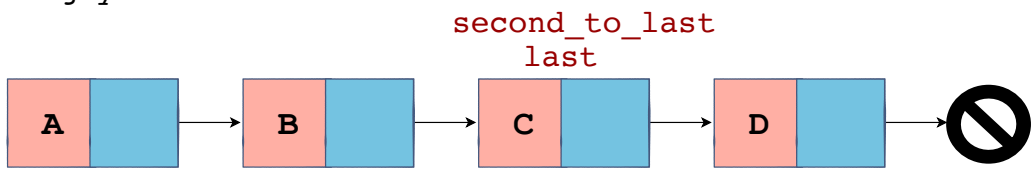
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

10 of 17

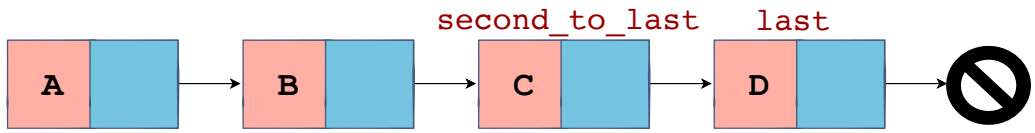
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

11 of 17

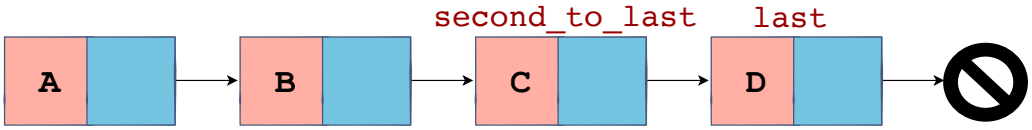
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

12 of 17

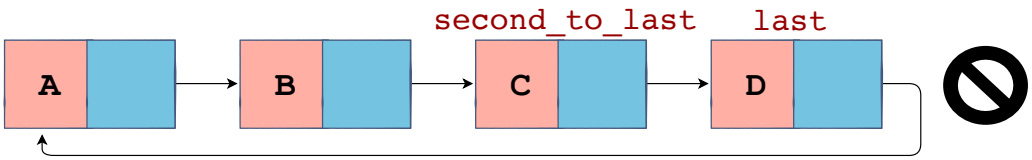
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

13 of 17

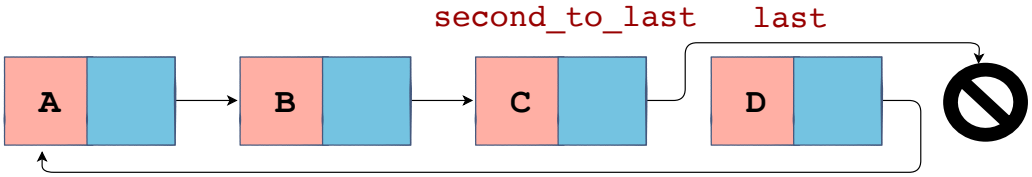
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

14 of 17

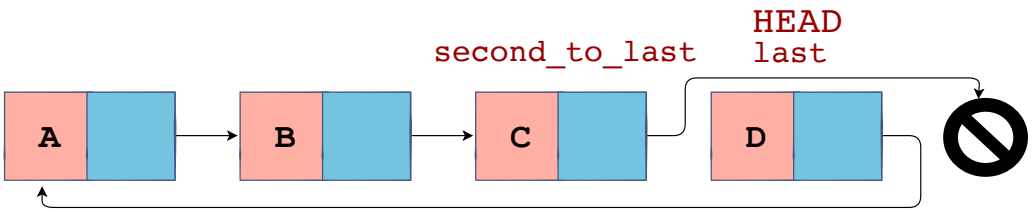
Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

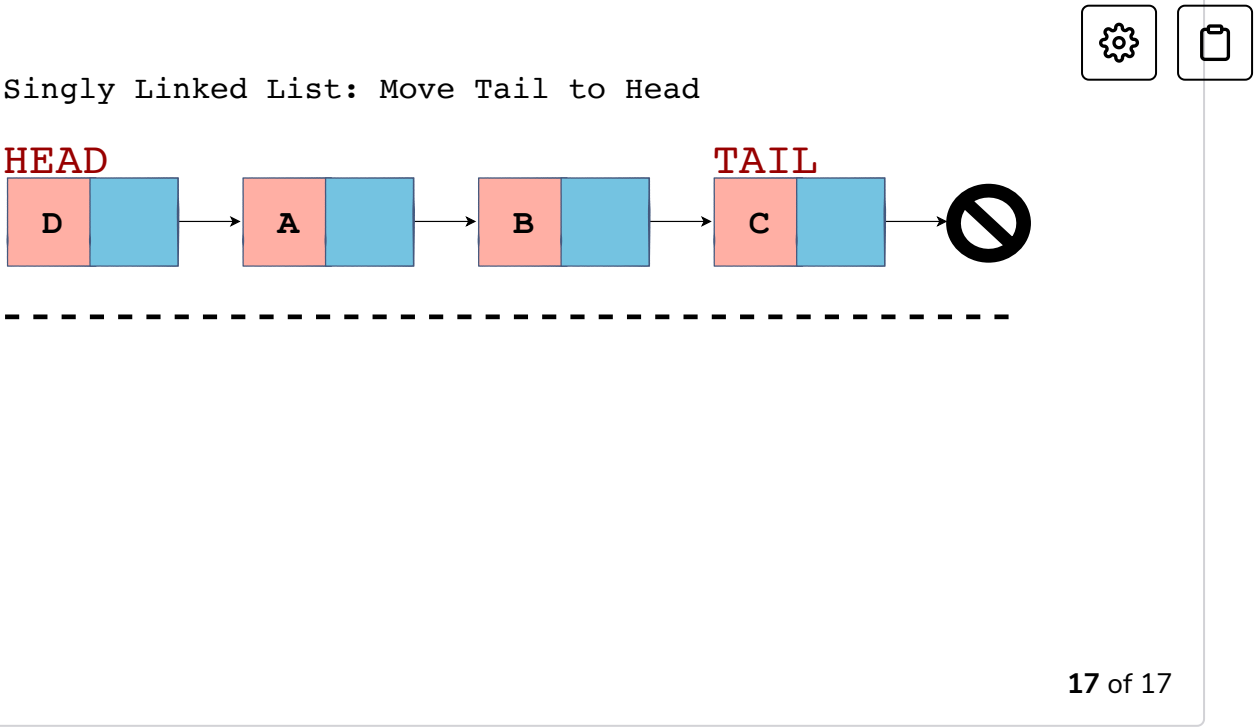
15 of 17

Singly Linked List: Move Tail to Head




```
def move_tail_to_head(self):
  last = self.head
  second_to_last = None
  while last.next:
      second_to_last = last
      last = last.next
  last.next = self.head
  second_to_last.next = None
  self.head = last
```

16 of 17



17 of 17

I hope the solution was clear to you! Below is the entire implementation of the `LinkedList` class that we have coded up until now. You can play around with and verify the method `move_head_to_tail`.

```
349          elif method == 2:
350              return self.is_palindrome_2()
351          elif method == 3:
352              return self.is_palindrome_3()
353
354      def move_tail_to_head(self):
355          if self.head and self.head.next:
356              last = self.head
357              second_to_last = None
358              while last.next:
359                  second_to_last = last
360                  last = last.next
361              last.next = self.head
362              second_to_last.next = None
363              self.head = last
364
365  # A -> B -> C -> D -> Null
366  # D -> A -> B -> C -> Null
367  llist = LinkedList()
368  llist.append("A")
369  llist.append("B")
370  llist.append("C")
```

```
371  llist.append("D")
372
373  llist.print_list()
374  llist.move_tail_to_head()
375  print("\n")
376  llist.print_list()
```

```
Output                                                    0.14s
 A
 B
 C
 D


 D
 A
 B
 C
```

Hope you liked this exercise. We have another challenge waiting for you in the next lesson. Best of Luck!

Back

Exercise: Move Tail to Head

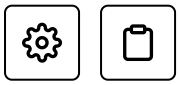
Next

Exercise: Sum Two Linked Lists

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-move-tail-to-head__singly-linked-lists__data-structures-and-algorithms-in-python)



Exercise: Sum Two Linked Lists

Challenge yourself with the following exercise in which you'll sum up two linked lists!

We'll cover the following

Problem

In this exercise, you are required to sum two linked lists and return the sum embedded in another linked list.

The first number that we append to the linked list represents the unit place and will be the least significant digit of a number. The next numbers appended to the linked list will subsequently represent the *tenth*, *hundredth*, *thousandth*, and so on places.

For example,

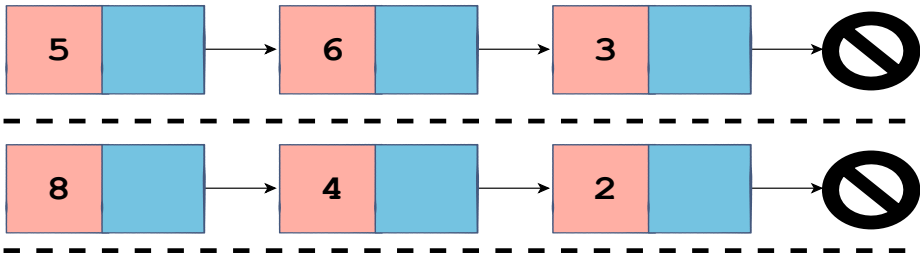
```
llist1 = LinkedList()
llist1.append(5)
llist1.append(6)
llist1.append(3)
```

in the code above, `llist1` represents the number `365`.

Note that you will not be tested on numbers whose sum requires an additional digit.

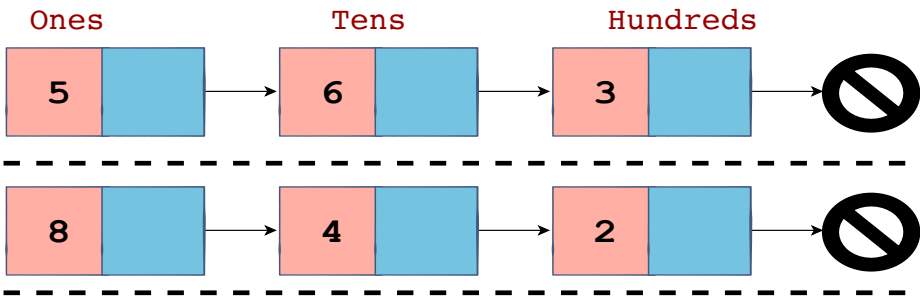
Below is an illustration to make the task clearer to you:

Singly Linked List: Sum Two Lists

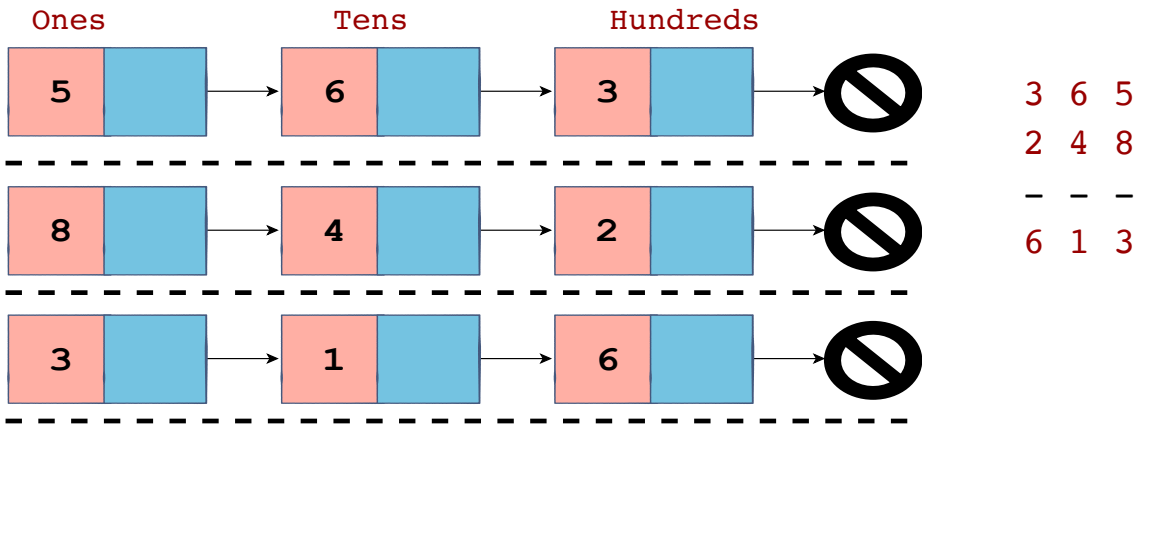


1 of 3

Singly Linked List: Sum Two Lists



2 of 3



3 of 3

As you can see from the illustration above, we have two linked lists from which we calculate the sum, and the third linked list represents that sum.

Coding Time!

In the code below, the `sum_two_lists` is a class method of the `LinkedList` class. You cannot see the rest of the code as it is hidden. As `sum_two_lists` is a class method, please make sure that you don't change the indentation of the code provided to you. You are required to write your solution under the method prototype.

The two linked lists that you have to sum are `llist` and the linked list on which the method `sum_two_lists` is called. You will return the third linked list from the method where the data values in that linked list will represent the sum.

Keep in mind that the linked lists can be of different sizes.

Good luck!

```
1      def sum_two_lists(self, llist):
```

```
        p = self.head
        q = llist.head
        sum = None
        carry = 0
        sum_llist = LinkedList()
        sum_llist.head = None
        if not p:
            return llist
        if not q:
            return self
        while p and q:
            total = p.data + q.data + carry
            rem = total % 10
            carry = total // 10
            new_node = Node(rem)
            if not sum:
                sum_llist.head = new_node
                sum = sum_llist.head
            else:
                sum.next = new_node
                sum = sum.next
            p = p.next
            q = q.next
        while p:
            total = p.data + carry
            rem = total % 10
            carry = total // 10
```

Show Results | Show Console

0.2s

5 of 5 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	(5,6,3).sum_to_lists(8,4,2)	3,1,6	3,1,6	Succeeded
✓	(8,2,0,1).sum_to_lists(2,8)	0,1,1,1	0,1,1,1	Succeeded
✓	(9,5).sum_to_lists(6,3,1)	5,9,1	5,9,1	Succeeded
✓	(2).sum_to_lists(5)	7	7	Succeeded

Result	Input	Expected Output	Actual Output	Reason
✓	().sum_to_lists(4,9)	4,9	4,9	Succeeded

Back

Next

Solution Review: Move Tail to Head

Solution Review: Sum Two Linked Lists

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/exercise-sum-two-linked-lists__singly-linked-lists__data-structures-and-algorithms-in-python)

Solution Review: Sum Two Linked Lists

This lesson contains the solution review for the challenge of summing two linked lists.

We'll cover the following

In this lesson, we investigate how to sum two singly linked lists. Check out the code below, after which, we will do a line by line analysis of it.

Implementation

```
    q = llist.head

    sum_llist = LinkedList()

    carry = 0
    while p or q:
        if not p:
            i = 0
        else:
            i = p.data
        if not q:
            j = 0
        else:
            j = q.data
        s = i + j + carry
        if s >= 10:
            carry = 1
            remainder = s % 10
            sum_llist.append(remainder)
        else:
            carry = 0
            sum_llist.append(s)
        if p:
            p = p.next
        if q:
```

```
28            q = q.next
29    return sum_llist
30
```

Explanation

First of all, we initialize `p` and `q` to point to the heads of each of the two linked lists (**lines 2-3**). On **line 5**, we declare `sum_llist` and initialize it to a linked list. The data values of `sum_llist` will represent the final sum at the end of this method. `carry` is initialized to `0` on **line** 7 and it will help us in evaluating the sum.

With the help of `p` and `q`, we set up a `while` loop which will run until both `p` and `q` equal `None`. On **lines 9-16**, we have handled the cases if either `p` or `q` equal `None`. If `p` or `q` equal `None`, we set `i` or `j` to `0` accordingly. In the other case where `p` and `q` are not equal to `None`, we use the data values of the node they are pointing to and store the data values in variables `i` and `j`. We are using `i` to represent the current digit picked from the first linked list and `j` to represent the current digit picked from the second one.

Once we get the values in `i` and `j`, we evaluate the sum on **line 1**7 by adding `i`, `j`, and `carry` and storing the sum in variable `s`. Note that `carry` will be `0` in the first iteration. After calculating the sum, we check if that sum is greater than or equal to `10` on **line 18**. If `s` is greater than or equal to `10`, we set `carry` to `1` (**line 19**) and calculate the `remainder` using the modulus operator on **line 20**. Now we append `remainder` to the final linked list (`sum_llist`) on **line 20**. On the other hand, if `s` is less than `10`, then there is no carry (`carry = 0`), and we append `s` to `sum_llist`. These steps are almost the same as we perform the arithmetic operation of addition. On **line 25-28**, we update `p` and `q` to their next nodes if they are not already `None`.

`sum_llist` is returned from the end of the method and contains the sum of the two linked lists we had at the start.

The `sum_two_lists` has been made part of the `LinkedList` class. You can run it and verify our solution!

```
365     def sum_two_lists(self, llist):
366         p = self.head
367         q = llist.head
368 
369         sum_llist = LinkedList()
370 
371         carry = 0
372         while p or q:
373             if not p:
374                 i = 0
375             else:
376                 i = p.data
377             if not q:
378                 j = 0
379             else:
380                 j = q.data
381             s = i + j + carry
382             if s >= 10:
383                 carry = 1
384                 remainder = s % 10
385                 sum_llist.append(remainder)
386             else:
387                 carry = 0
388                 sum_llist.append(s)
389             if p:
390                 p = p.next
391             if q:
392                 q = q.next
```

I hope you were able to enjoy and understand this challenge. By now, you'll have a firm grasp on the problems concerning singly linked lists. In the next chapter, we are going to explore another type of linked list: Circular Linked List. See you there!

Back

Next

Exercise: Sum Two Linked Lists

Quiz

Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-sum-two-linked-lists__singly-linked-lists__data-structures-and-algorithms-in-python)