educative

Append and Prepend

In this lesson, you will learn how to implement doubly linked lists and insert elements in it using Python.

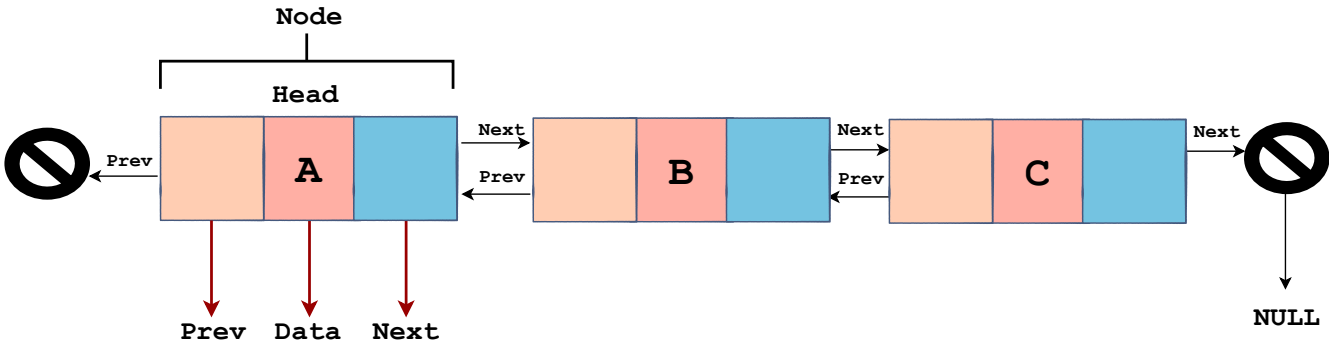
We'll cover the following

- append
- print_list
- prepend

In this lesson, we introduce the doubly linked list data structure and code up an implementation of this data structure in Python.

After that, we look at how to append (add elements to the back of the doubly linked list) and prepend (add elements to the front of the doubly linked list).

Finally, we will write a print method to verify that the append and prepend methods work as expected.



Doubly Linked List

As you can see, the doubly linked list is very similar to the singly linked list except for the difference of the previous pointer. In a doubly linked list, a node is made up of the following components:

- Data
- Next (Points to the next node)
- Prev (Points to the previous node)

As illustrated, the `prev` of the head node points to NULL while the `next` of the tail node also points to NULL.

Let's go ahead and implement the Doubly Linked List in Python:

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


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

    def append(self, data):
        pass

    def prepend(self, data):
        pass

    def print_list(self):
        pass
```

class Node and class DoublyLinkedList

We have two classes in the code above:

1. `Node`

2. `DoublyLinkedList`

The key difference in the `Node` class of the last two chapters and of this chapter is that we have another component, `prev`, present in the `Node` class for this chapter. `prev` will keep track of the previous node in the linked list.

We also define `DoublyLinkedList` class and initialize its head to `None`. In this lesson, we will complete the implementation of the three class methods that we have specified in the code above.

append

Let's go over the implementation of `append`:

```
1  def append(self, data):
2    if self.head is None:
3      new_node = Node(data)
4      self.head = new_node
5    else:
6      new_node = Node(data)
7      cur = self.head
8      while cur.next:
9        cur = cur.next
10     cur.next = new_node
11     new_node.prev = cur
```

append(self, data)

We have handled two cases of appending elements in the code above:

1. The doubly linked list is empty.
2. The doubly linked list is not empty.

If the doubly linked list is empty, i.e., `self.head` is `None`, then the execution jumps to **line 3** and we initialize `new_node` based on `data` inputted to the method. As the `new_node` will be the only node in the linked list, we will

make it the head node on **line 4**.

Now let's move on to the second case. On **line 6**, we initialize `new_node` by calling the constructor of the `Node` class and passing `data` to it. Next, we need an appropriate position to locate the newly created node. Therefore, we initialize `cur` on **line 7** to the head node to traverse the doubly linked list and reach the last node in it. This is done by the `while` loop in which `cur` updates to `cur.next` and the loop runs until we reach the last node where `cur.next` will be equal to `None` (**lines 8-9**). Once the `while` loop terminates, we point the next of `cur` to `new_node` on **line 10**. As now we have appended `new_node` to the linked list, we also need to care about the previous component of `new_node` which should now point to `cur`. This step is implemented on **line 11**.

print_list

As we need some way to test the append method, we'll go on to implement `print_list`:

```
1  def print_list(self):
2    cur = self.head
3    while cur:
4      print(cur.data)
5      cur = cur.next
```

print_list(self)

The `print_list` method is pretty straightforward. `cur` initially points to `self.head` (**line 2**) and then keeps printing its data (**line 4**) and updating itself to `cur.next` (**line 5**) to traverse the entire linked list. This is it!

prepend

It's time for us to implement the `prepend` method of `DoublyLinkedList` class. Check out the code below:

```
1  def prepend(self, data):
2    if self.head is None:
3      new_node = Node(data)
4      self.head = new_node
5    else:
6      new_node = Node(data)
7      self.head.prev = new_node
8      new_node.next = self.head
9      self.head = new_node
```

prepend(self, data)

The code on **lines 2-4** is pretty much the same as the code in the `append` method. Let's jump to **line 6** which will execute if we don't have an empty linked list. On **line 6**, we create `new_node` using `data` passed into the method. Now we need to position it at the start of the linked list appropriately. As the newly created node will precede the current head node and take over the head position, the previous of the current head node should point to `new_node`. This is done on **line 7**. The next of the `new_node` should then also point to the current head node so, on **line 8**, we set `new_node.next` to `self.head`. Finally, on **line 9**, we update `self.head` to `new_node`. This perfects our implementation to prepend an element to a doubly linked list.

Below we have the entire implementation of the `DoublyLinkedList` class with all the methods completely implemented along with a sample test case. Check it out!

```
22              new_node.prev = cur
23
24      def prepend(self, data):
25          if self.head is None:
26              new_node = Node(data)
27              self.head = new_node
```

```
28          else:
29              new_node = Node(data)
30              self.head.prev = new_node
31              new_node.next = self.head
32              self.head = new_node
33
34      def print_list(self):
35          cur = self.head
36          while cur:
37              print(cur.data)
38              cur = cur.next
39
40
41  dllist = DoublyLinkedList()
42  dllist.prepend(0)
43  dllist.append(1)
44  dllist.append(2)
45  dllist.append(3)
46  dllist.append(4)
47  dllist.prepend(5)
48
49  dllist.print_list()
```

In the next lesson, we'll go over other methods of inserting elements in a doubly linked list. Stay tuned!

Back

Quiz

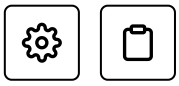
Next

Add Node Before/After

Mark as Completed

Report an Issue

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



Add Node Before/After

In this lesson, you will learn how to add a node before or after a specified node.

We'll cover the following

In this lesson, we consider how to add nodes either before or after a specified node in a doubly-linked list. Once we cover the concept of how to perform this action, we follow through with a Python implementation.

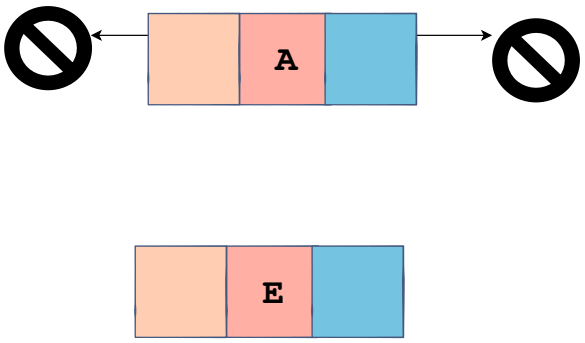
The general approach to solve this problem would be to traverse the doubly linked list and look for the key that we have to insert the new node after or before. Once we get to the node whose value matches the input key, we update the pointers to make room for the new node and insert the new node according to the position specified.

Add Node After

We'll break up this scenario into two cases:

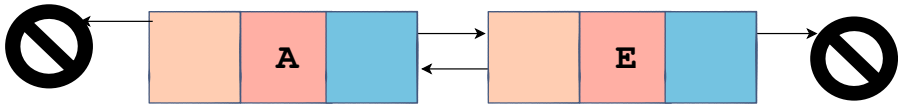
1. The node that we have to insert after the new node is the last node in the doubly linked list so that the next of that node is NULL.

Doubly Linked List: Add Node After



1 of 2

Doubly Linked List: Add Node After



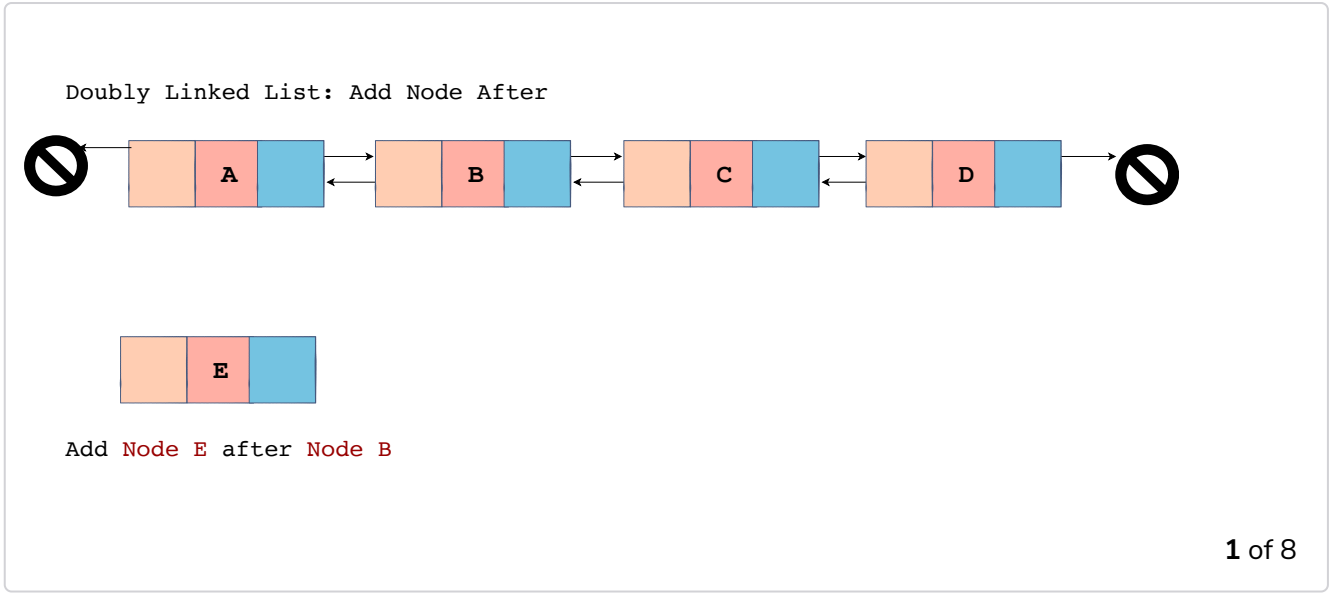
`append` method is called to insert Node E

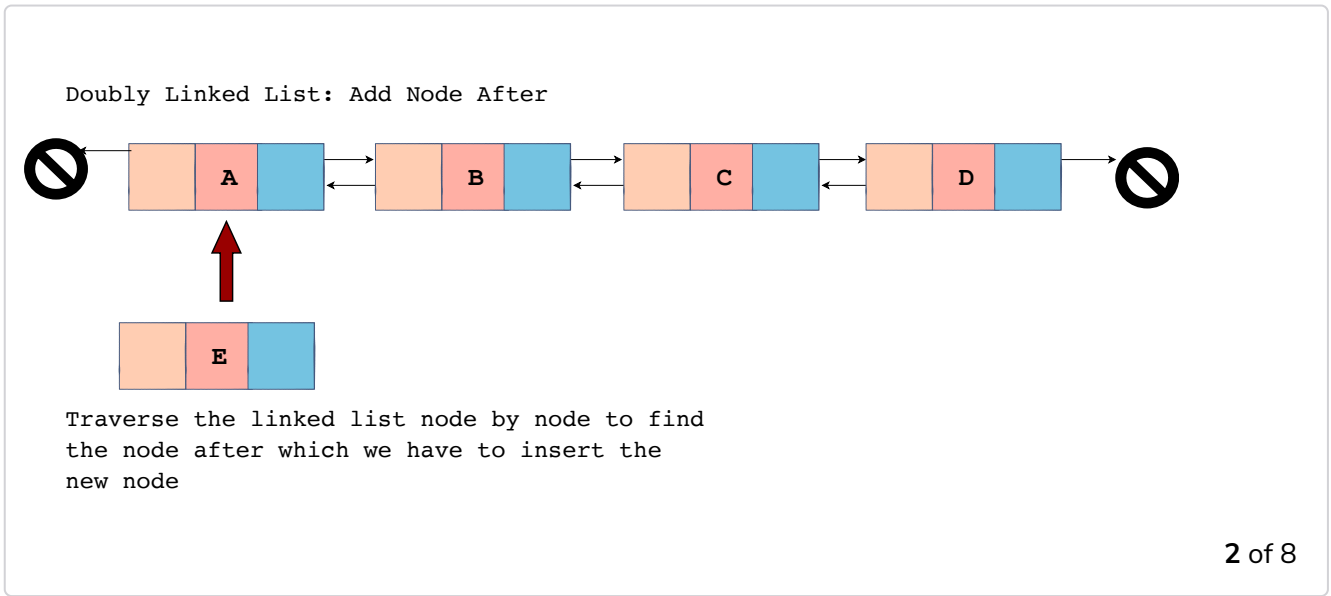
2 of 2

In the above case, we'll make a call to the `append` method that we implemented in the last lesson. `append` will handle everything for us and perfectly works in this scenario as we have to insert the new node after the last node, which is what `append` method does.

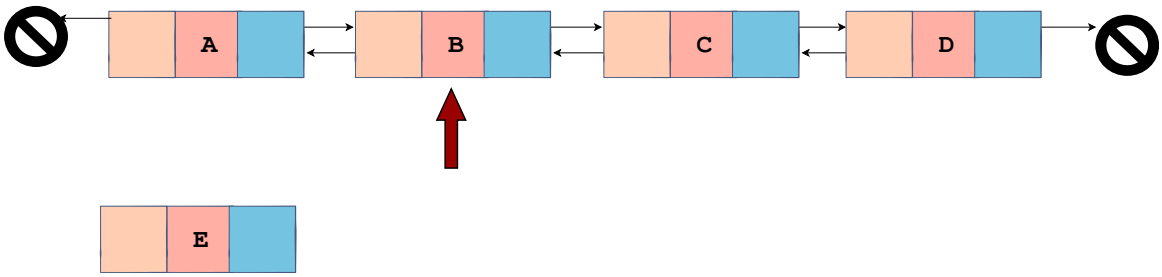
2. The node that we have to insert after the new node is not the last node in the doubly linked list.

Let’s look at the illustration below to check how we will handle the second case:





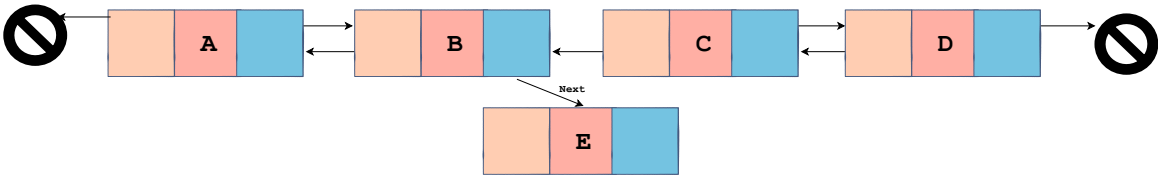
Doubly Linked List: Add Node After



Node found!

3 of 8

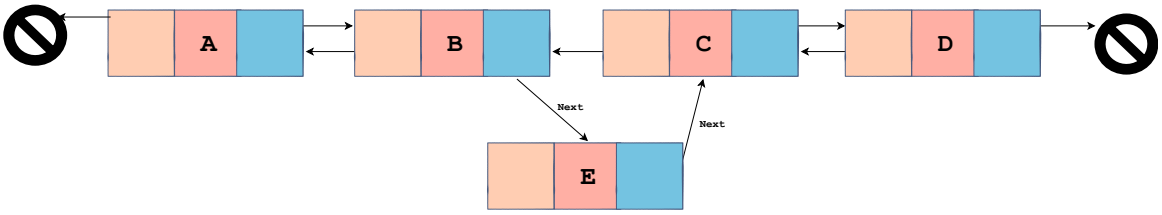
Doubly Linked List: Add Node After



Set next of Node B to Node E

4 of 8

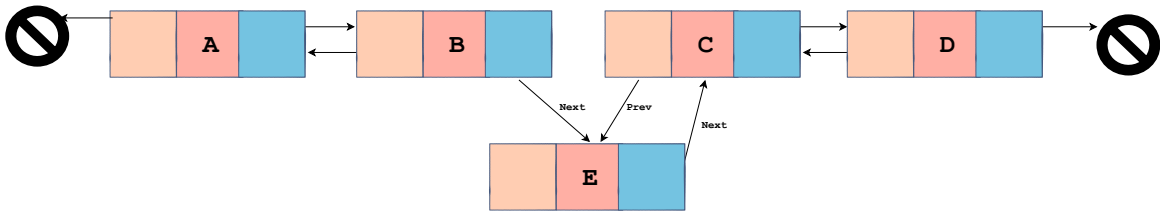
Doubly Linked List: Add Node After



Set next of Node E to Node C

5 of 8

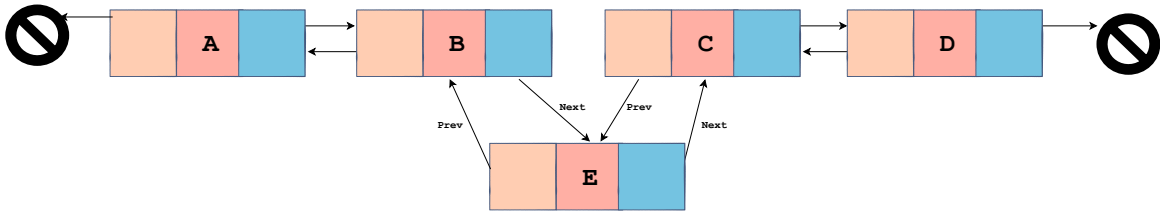
Doubly Linked List: Add Node After



Set prev of Node C to Node E

6 of 8

Doubly Linked List: Add Node After



Set prev of Node E to Node B

7 of 8

Doubly Linked List: Add Node After

8 of 8

Hope the slides above were useful for you to understand the algorithm that we are about to code in Python. Let's go ahead and check out the code below:

```
1  def add_after_node(self, key, data):
2    cur = self.head
3    while cur:
4      if cur.next is None and cur.data == key:
5        self.append(data)
6        return
7      elif cur.data == key:
8        new_node = Node(data)
9        nxt = cur.next
10       cur.next = new_node
11       new_node.next = nxt
12       new_node.prev = cur
13       nxt.prev = new_node
14       return
15     cur = cur.next
```

add_after_node(self, key, data)

`key` passed into `add_after_node` is the key of the node after which we have to insert the new node in the linked list while `data` is for the data component of the new node to be inserted.

On **line 2**, we set `cur` to `self.head` and proceed to the `while` loop on **line 3** which runs until `cur` is `None`. As we divided our algorithm for two cases, we have divided our code into conditions to cater to these two cases in the `while` loop. First, the conditions on **line 4** check if `key` matches the data of the last node of the linked list where `cur.next` will be `None`. If it does, then we call the `append` method to append the new node to the linked list and return on **line 6**.

On the other hand, the condition on **line 7** will handle the second case. If we find the node after which we have to insert the new node, and it is not the last node in the linked list, then execution jumps to **line 8**. `new_node` is created based on the `data` argument passed to the method. Next, we store

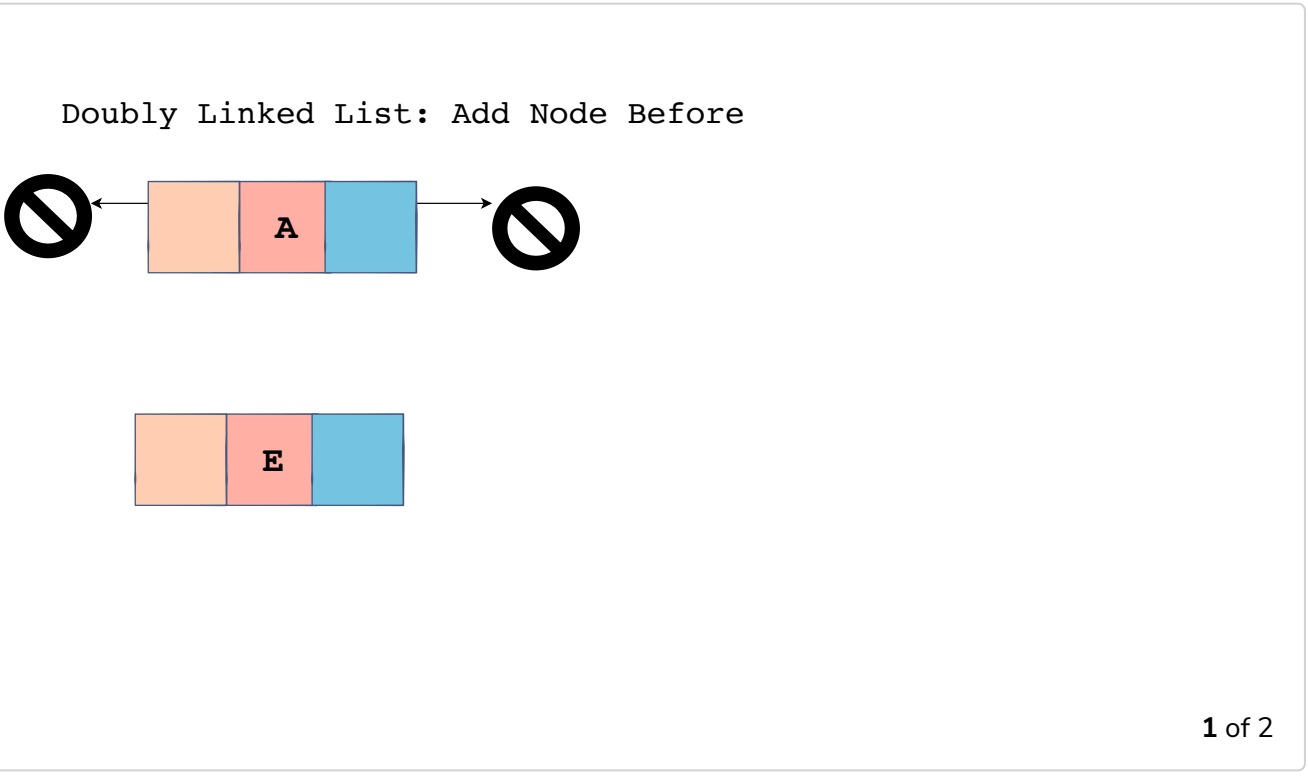
the next node of `cur` in a temporary variable `nxt` on **line 9**. As we have to insert `new_node` after `cur`, `cur.next` is set to `new_node` on **line 10**. On **line 11**, `new_node.next` is set to `nxt` which was previously the next node of `cur`. At this point, we have set all the next pointers, now let's update the previous pointers as we are dealing with a doubly linked list.

As we have inserted `new_node` after `cur`, the `prev` of `new_node` points to `cur` on **line 12**. `nxt` is the next node to the `new_node` so we update the `prev` of `nxt` to `new_node` on **line 13**. After this step, we return from the method on **line 14**. `cur` updates itself to `cur.next` outside the if conditions to help us traverse the linked list (**line 15**).

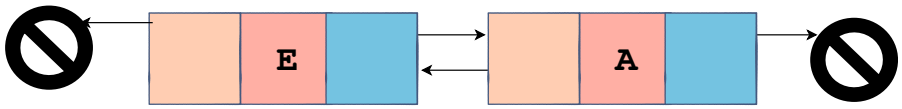
Add Node Before

Now let's discuss how to add nodes before a specified node. We will also divide this problem into two scenarios:

1. We have to insert a new node before the head node.



Doubly Linked List: Add Node Before

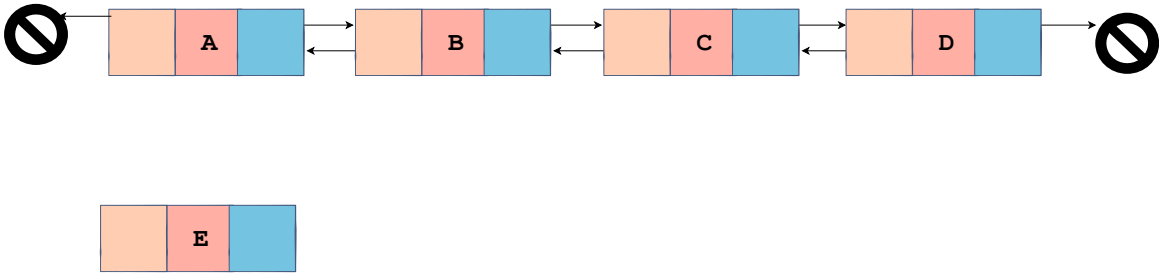


prepend method is called to insert Node E

2 of 2

2. We have to insert a new node before a node that is NOT the head node.

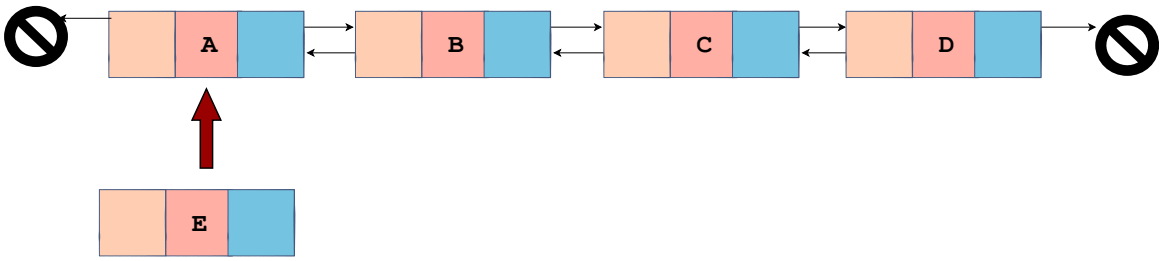
Doubly Linked List: Add Node Before



Add Node E before Node B

1 of 8

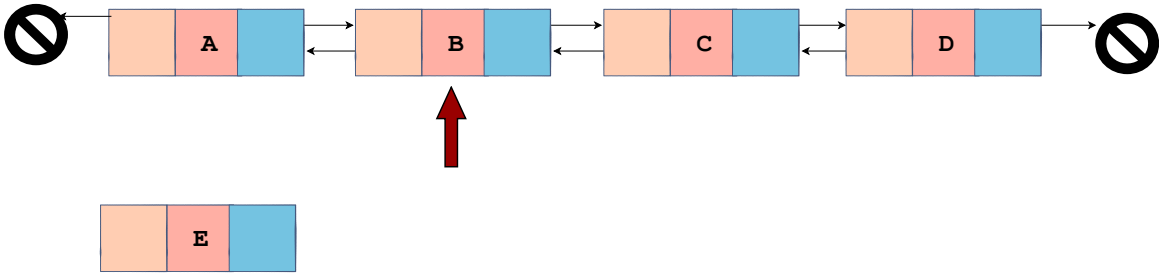
Doubly Linked List: Add Node Before



Traverse the linked list node by node to find the node before which we have to insert the new node

2 of 8

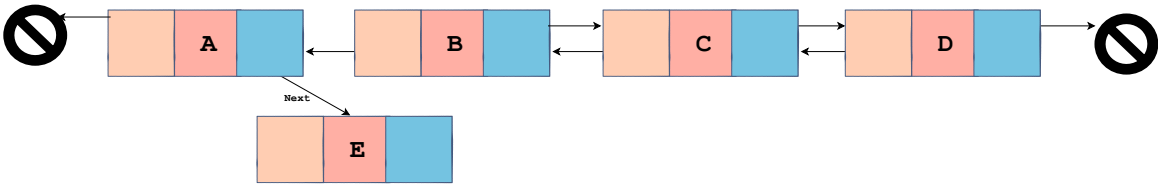
Doubly Linked List: Add Node Before



Node found!

3 of 8

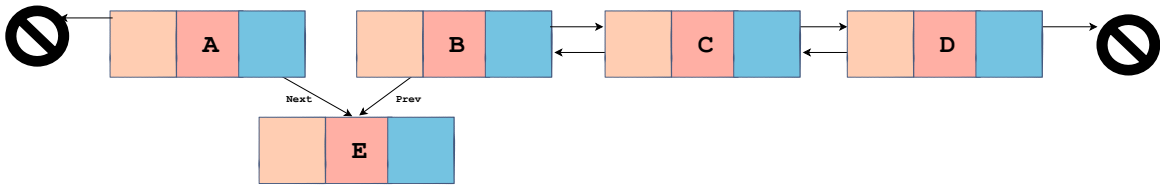
Doubly Linked List: Add Node Before



Set next of Node A to Node E

4 of 8

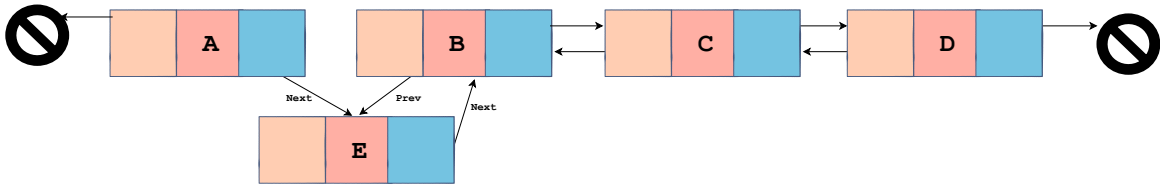
Doubly Linked List: Add Node Before



Set prev of Node B to Node E

5 of 8

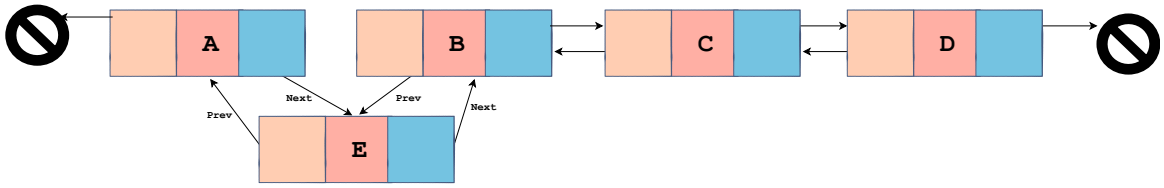
Doubly Linked List: Add Node Before



Set next of Node E to Node B

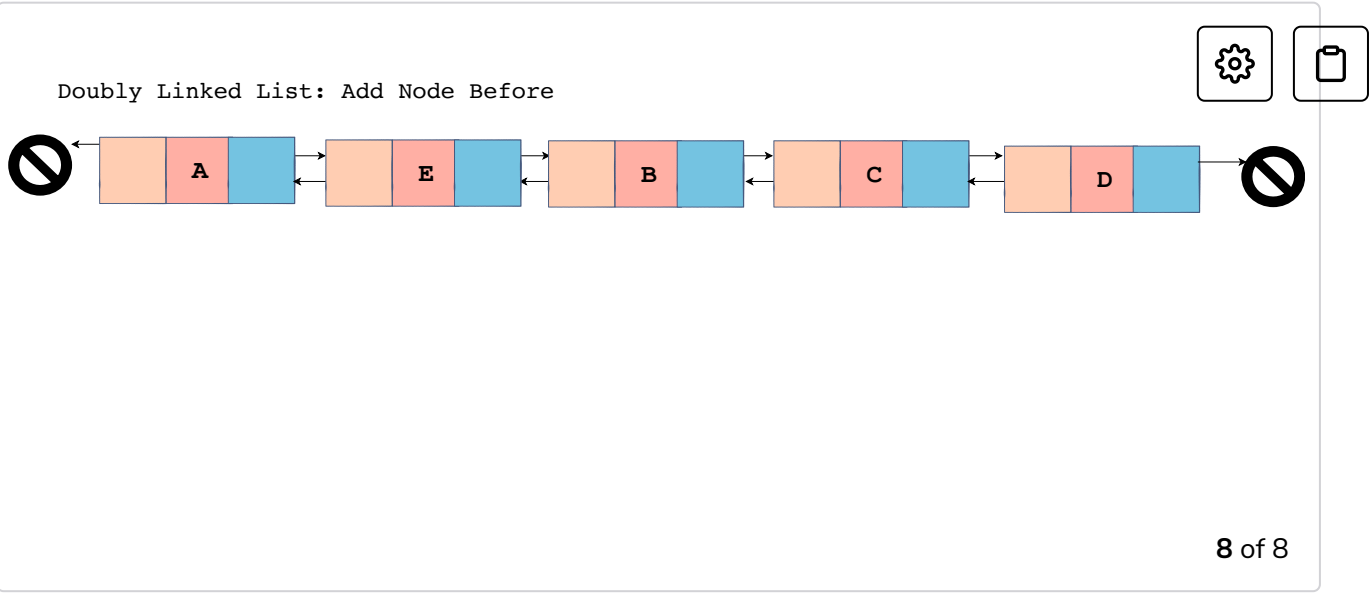
6 of 8

Doubly Linked List: Add Node Before



Set prev of Node E to Node A

7 of 8



Now let's jump to the programming part in Python:

```
def add_before_node(self, key, data):
  cur = self.head
  while cur:
    if cur.prev is None and cur.data == key:
      self.prepend(data)
      return
    elif cur.data == key:
      new_node = Node(data)
      prev = cur.prev
      prev.next = new_node
      cur.prev = new_node
      new_node.next = cur
      new_node.prev = prev
      return
    cur = cur.next
```

add_before_node(self, key, data)

The same arguments (`self` , `key` , `data`) have been passed to `add_before_node` method as were given to `add_after_node` method. `key` is the key of the node before which we will insert the new node that we will

create using `data`.

`cur` is initialized to `self.head` on **line 2**. Next, we have a `while` loop on **line 3** in which `cur` updates to `cur.next` (**line 15**) until `cur` becomes `None`. Now, we check whether or not the previous node of `cur` is `None` and whether if we want to insert our new node before `cur`. If true, then we call the `prepend` method that we implemented in the last lesson which will do the necessary steps to insert the new node before the head node, after which we return (**lines 5-6**). However, if we don't want to insert before the head node and find the node that we want to insert before (which is not the head node), the execution jumps to **line 8**. The first thing we do is create the `new_node` by passing in `data` to the constructor from the `Node` class. On **line 9**, we store the previous node of the current node in a temporary variable `prev`. Now we need to update all the pointers to make the insertion successful. So, on **line 10**, we update the next node of `prev` to be `new_node` and as we have to insert `new_node` before `cur`, we set `cur.prev` to `new_node` on **line 11**. Let's also update the previous and the next pointers of `new_node`. As `new_node` is before `cur`, the `next` of `new_node` points to `cur` (**line 12**) and as `prev` is the previous node to `new_node` now, `new_node.prev` points to `prev` on **line 13**. On **line 14**, we return from the method after we have successfully inserted the `data`.

That was all regarding the insertion of elements in a doubly linked list. We have a test case for you in the code widget below so that you can play around will all the insertion implementations:

```
60            new_node = Node(data)
61            prev = cur.prev
62            prev.next = new_node
63            cur.prev = new_node
64            new_node.next = cur
65            new_node.prev = prev
66            return
67          cur = cur.next
68
69      def print_list(self):
70          cur = self.head
```

```
70      cur = self.head
71      while cur:
72        print(cur.data)
73        cur = cur.next
74
75
76  dllist = DoublyLinkedList()
77
78  dllist.prepend(0)
79  dllist.append(1)
80  dllist.append(2)
81  dllist.append(3)
82  dllist.append(4)
83  dllist.prepend(5)
84  dllist.add_after_node(3,6)
85  dllist.add_before_node(4,9)
86
87  dllist.print_list()
```

Output 0.19s

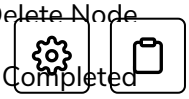
```
5
0
1
2
3
6
9
4
```

So far, we have covered many different methods to insert elements in a doubly linked list. Let's learn how to remove elements from a doubly linked list in the next lesson.

Back Next

Append and Prepend

Delete Node



Mark as Completed

Report an Issue

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

educative

Delete Node

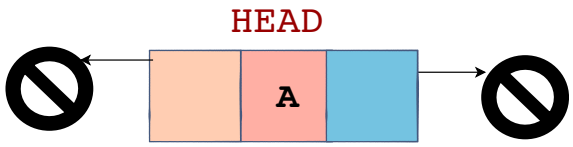
In this lesson, you will learn how to remove a node from a doubly linked list.

We'll cover the following

In this lesson, we consider how to delete, or, remove nodes from a doubly linked list. Once we cover the concept of how to perform this action, we follow through with a Python implementation.

We will analyze the entire implementation step by step in four parts. The following are the different cases that we can encounter while deleting a node from a doubly linked list.

Case 1: Deleting the only node present



Case 1 is where we want to delete the only node present in the linked list. As it is the single node in the linked list, then it is the head node as well. The `prev` and `next` pointer of such a node point to `None` which makes it a special case. Let's look at the implementation for this case below.

Implementation

```
def delete(self, key):
        cur = self.head
        while cur:
            if cur.data == key and cur == self.hea
                # Case 1:
                if not cur.next:
                    cur = None
                    self.head = None
                    return
            cur = cur.next
```

delete(self, key)

Explanation

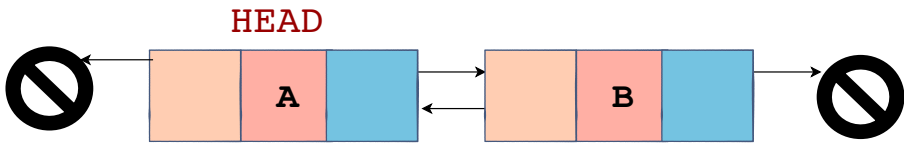
The method takes in `key` which is the key of the node to be deleted. On **line 2**, we set `cur` to `self.head` and proceed to the `while` loop on **line 3** which will run until `cur` is `None`. The execution will jump to **line 6** if the conditions on **line 4** evaluate to `True` i.e., `cur.data` is equal to `key` and the current node is also the head node. Now at this point, we have met most of the conditions for *Case 1*, and we have to do a final check. Therefore, on **line 6**, we check if the next node of `cur` is `None` or not. This will confirm if it's the only node in the linked list. If it is the only node in the linked list, then `cur` has met the criteria for the type of node mentioned in *Case 1*. As a result, we set `cur` and `self.head` to `None` **(lines 7-8)** and return from the method after successfully deleting the specified node and making the linked list empty.

On **line 10**, we update `cur` to `cur.next` to traverse the linked list using the `while` loop.

Case 2: Deleting Head node

Now let's have a look at another case:

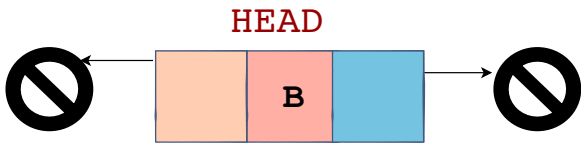
Doubly Linked List: Delete Node -- Case 2



Delete Node A

1 of 2

Doubly Linked List: Delete Node -- Case 2



Delete Node A

2 of 2

Case 2 refers to deleting the head node as in *case 1*, but now the node to be deleted is not the only node in the linked list. The head node points to another node which should replace the head node after the deletion. Let's see how we handle the second case in Python.

Implementation

```
1  def delete(self, key):
2    cur = self.head
3    while cur:
4      if cur.data == key and cur == self.head:
5        # Case 1:
6        if not cur.next:
7          cur = None
8          self.head = None
9          return
10
11       # Case 2:
12       else:
13         nxt = cur.next
14         cur.next = None
15         nxt.prev = None
16         cur = None
17         self.head = nxt
18         return
19     cur = cur.next
```

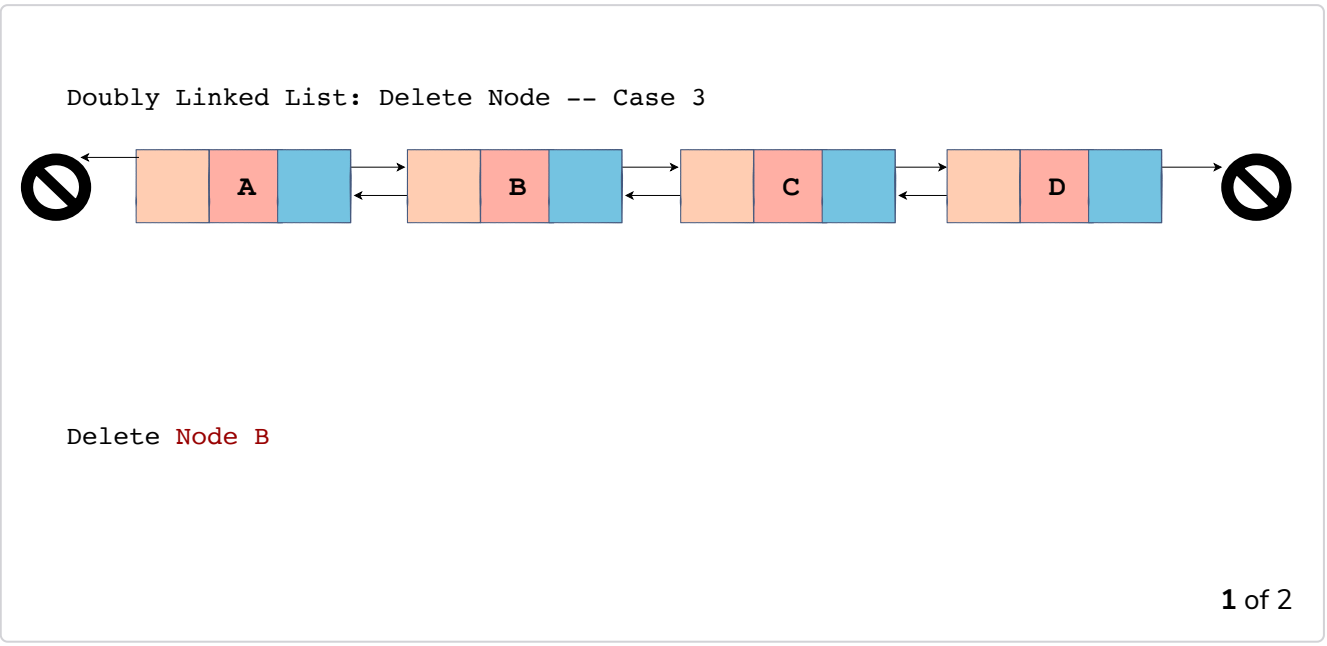
delete(self, key)

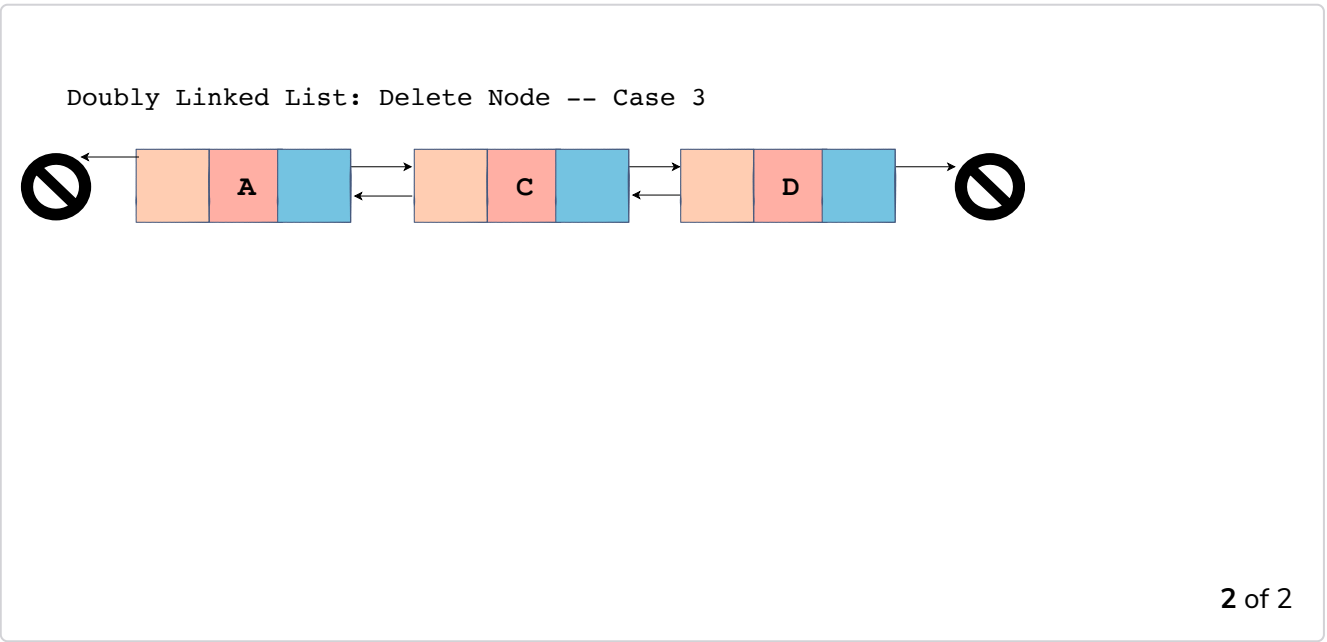
Explanation

Just like in *Case 1*, the condition on **line 4** checks for the case when the node to be deleted is the head node. As explained before, the condition on **line 6** checks if `cur` is the only node in the linked list which is a criterion for *Case 1*. If `cur.next` is not `None`, it implies that `cur` is not the only node in the linked list and the execution jumps to the `else` part (**line 12**). On **line 13**, we save the next node of `cur` in a variable named `nxt`. Now `cur` needs to be deleted, so we set `cur.next` to `None` on **line 14**. After the deletion, `nxt` will be the new head as we are deleting the current head. Therefore, we set `nxt.prev` to `None` (**line 15**) instead of `cur`, which is set to `None` in the next line. Finally, we make `nxt` the head node by setting `self.head` to `nxt` on

line 17 and return from the method in the very next line. In the code above, we have successfully deleted the head node and made the next node of the deleted head node the new head node.

Case 3: Deleting node other than head where `cur.next` is not `None`





In this case, we will code for a node that is not the head node or the last node but is located somewhere in between the two nodes in the linked list.

Implementation

Have a look at the code below:

```
    if cur.data == key and cur == self.head:
      # Case 1:
      if not cur.next:
        cur = None
        self.head = None
        return

      # Case 2:
      else:
        nxt = cur.next
        cur.next = None
        nxt.prev = None
        cur = None
        self.head = nxt
        return

    elif cur.data == key:
      # Case 3:
      if cur.next:
          nxt = cur.next
          prev = cur.prev
          prev.next = nxt
          nxt.prev = prev
          cur.next = None
          cur.prev = None
          cur = None
          return
    cur = cur.next
```

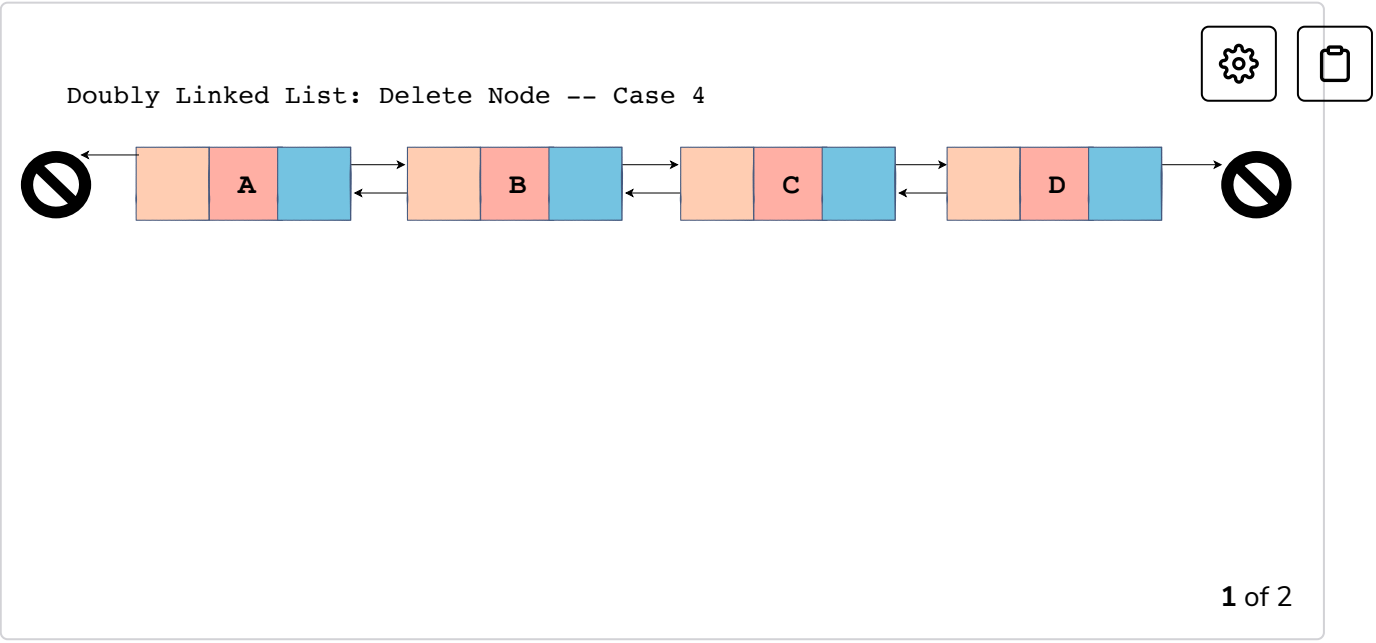
delete(self, key)

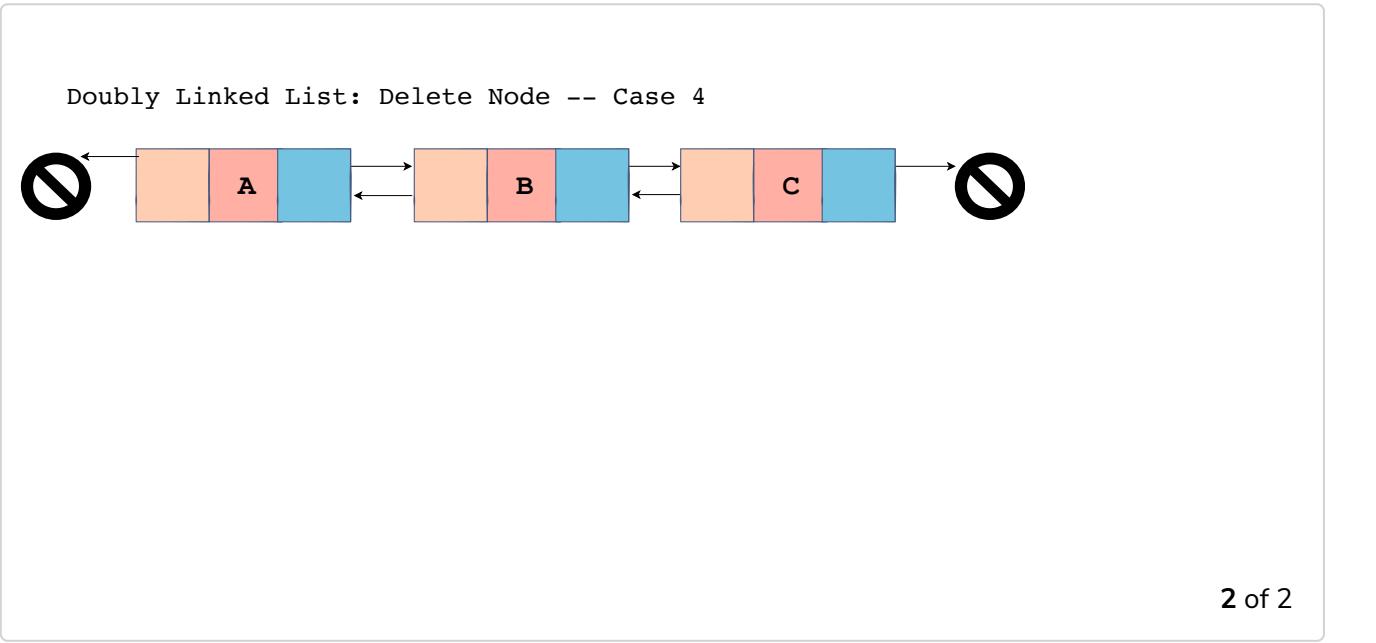
Explanation

For the explanation, we'll only focus on the code between **line 20** and **line 30** inclusive. The condition on **line 20** will evaluate to `True` if `cur.data` in any iteration of the `while` loop becomes equal to the `key` of the node to be deleted. If `cur` is the head node, execution jumps to **line 6** but if it's not the head node, execution jumps to **line 22**. Here, we check whether `cur` is the last node in the linked list or not. If it is not, then `cur` perfectly matches the description for *Case 3*. From **line 23** onwards comes the deletion part where we remove `cur` and make the previous node of `cur` point to the next node of `cur` and vice versa. We save the next and the previous nodes of `cur` in the variables `nxt` and `prev` (**lines 23-24**). To remove `cur` in-between `prev` and `next`, we set `prev.next` to `nxt` (**line 25**) which was previously set to `cur`, and `nxt.prev` updates to `prev` from `cur` on **line 26**. On **lines 27-29**, we set `cur`, `cur.next`, and `cur.prev` to `None` to remove `cur` completely and then return from the method on **line 30**.

Case 4: Deleting node other than head where `cur.next` is `None`

Finally, we have come to the last case where we are deleting a node that is not the head node and where the next node points to `None`. This is essentially the last node in the doubly linked list.





Implementation

Let’s look at the Python implementation for the case illustrated above:

```
12        else:
13          nxt = cur.next
14          cur.next = None
15          nxt.prev = None
16          cur = None
17          self.head = nxt
```

```
18          return
19
20      elif cur.data == key:
21        # Case 3:
22        if cur.next:
23            nxt = cur.next
24            prev = cur.prev
25            prev.next = nxt
26            nxt.prev = prev
27            cur.next = None
28            cur.prev = None
29            cur = None
30            return
31
32          # Case 4:
33        else:
34            prev = cur.prev
35            prev.next = None
36            cur.prev = None
37            cur = None
38            return
39      cur = cur.next
```

delete(self, key)

Explanation

This case is pretty straightforward, and we jump to its code on lines **33-38** if the following conditions are met in any iteration of the `while` loop:

- `cur.data` is equal to `key`
- `cur` is not equal to `self.head`
- `cur.next` is not `None`

In this case, we need to care about the previous node of `cur` which will be the new last node and will now point to `None` instead of `cur`. Therefore, we set `prev` equal to `cur.prev` on **line 34** and then set its `next` to `None` on **line 35**. In the next lines (**lines 36-37**), we just set `cur.prev` and `cur` equal to `None` to remove them out of the linked list and return from the method on **line 38**.

That was all about deleting a node in the doubly linked list. I hope you were able to understand the implementation and explanation for each case!

In the code widget below, we test the `delete` method with a sample test case. Go ahead and play around with the implementation!

```
                    nxt.prev = prev
                    cur.next = None
                    cur.prev = None
                    cur = None
                    return

                # Case 4:
                else:
                    prev = cur.prev
                    prev.next = None
                    cur.prev = None
                    cur = None
                    return
            cur = cur.next


dllist = DoublyLinkedList()
dllist.append(1)
dllist.append(2)
dllist.append(3)
dllist.append(4)

dllist.delete(1)
dllist.delete(6)
dllist.delete(4)

dllist.delete(3)
dllist.print_list()
```

Output 0.17s

```
2
```

In the next lesson, we'll have a look at how to reverse a doubly linked list.

Back

Add Node Before/After

Next

Reverse

Mark as Completed

Reverse

In this lesson, you will learn how to reverse a doubly linked list.

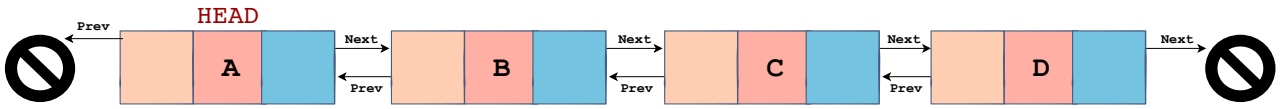
We'll cover the following

In this lesson, we consider how to reverse the nodes in a doubly linked list. Once we cover the concept of how to perform this action, we follow through with a Python implementation.

Algorithm

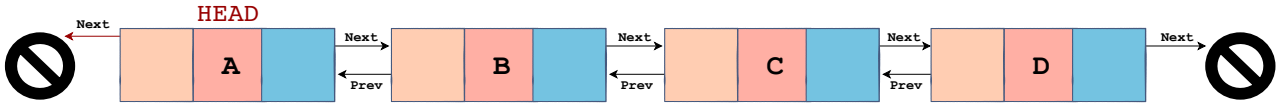
To reverse a doubly linked list, we need to switch the next and the previous pointers of every node. Also, we need to switch the last node with the head node of the linked list. Check out the illustration below for more clarity:

Doubly Linked List: Reverse



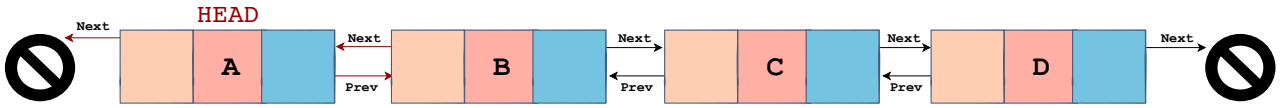
1 of 6

Doubly Linked List: Reverse



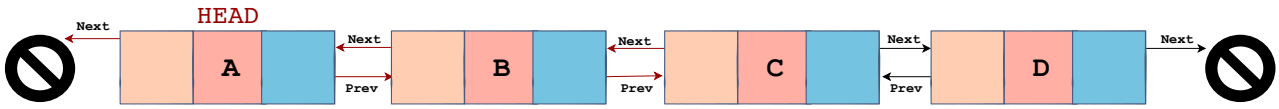
2 of 6

Doubly Linked List: Reverse



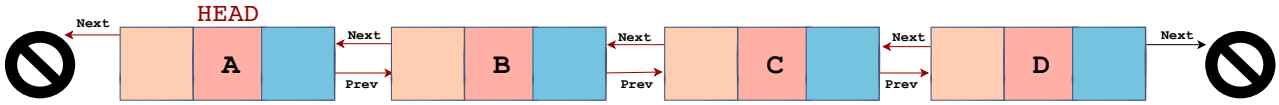
3 of 6

Doubly Linked List: Reverse



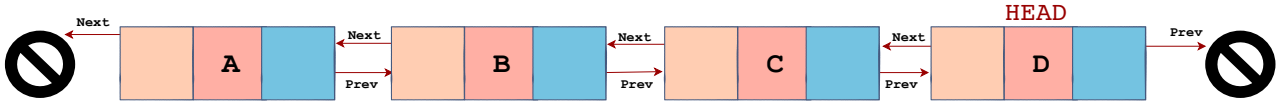
4 of 6

Doubly Linked List: Reverse



5 of 6

Doubly Linked List: Reverse



6 of 6

As we traverse the linked list, we swap the previous pointer with the next pointer and eventually, we make the last node of the original linked list the head node of the reversed linked list.

Implementation

Now let's go ahead and see how this algorithm is implemented in Python:

```
1  def reverse(self):
2    tmp = None
3    cur = self.head
4    while cur:
5      tmp = cur.prev
6      cur.prev = cur.next
7      cur.next = tmp
8      cur = cur.prev
9    if tmp:
10     self.head = tmp.prev
```

reverse(self)

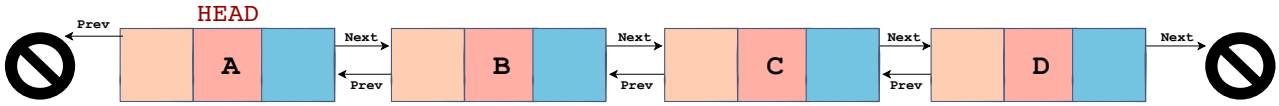
Explanation

On **lines 2-3**, `tmp` and `cur` is set to `None` and `self.head`, respectively. In the `while` loop on **line 4**, which terminates when `cur` equals `None`, we swap the next and the previous pointers of `cur`. To swap, we first save the value of `cur.prev` in a temporary variable, `tmp`, on **line 5**. Next, we update `cur.prev` to `cur.next` on **line 6** while `cur.next` is set to `tmp` on **line 7**. `tmp` has the value of `cur.prev` stored before the update on **line 6**. This is pretty much the standard way of swapping the values of two variables in programming.

Now as the swap has taken place, instead of updating `cur` to `cur.next`, we update `cur` to `cur.prev` to iterate to the next node in the original linked list (**line 8**). When the `while` loop terminates, we are almost done with the reversal except for setting the new head of the reverse linked list. Hence, on **lines 9-10**, we check if `tmp` is not `None` and if it is not, we set `self.head` to `tmp.prev` where `tmp.prev` is the last node in the linked list.

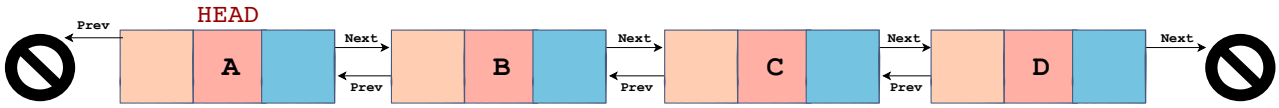
Now the above code is tough to visualize, so we have some illustrations for you to go through and visualize what's happening.

Doubly Linked List: Reverse



1 of 27

Doubly Linked List: Reverse

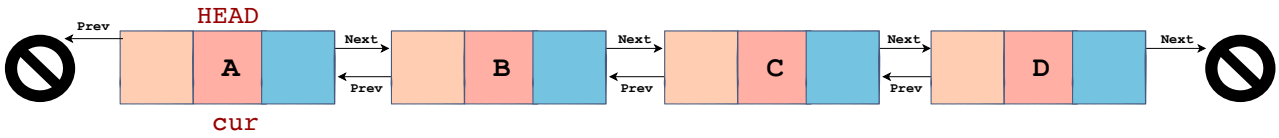



```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

`tmp = None`

2 of 27

Doubly Linked List: Reverse

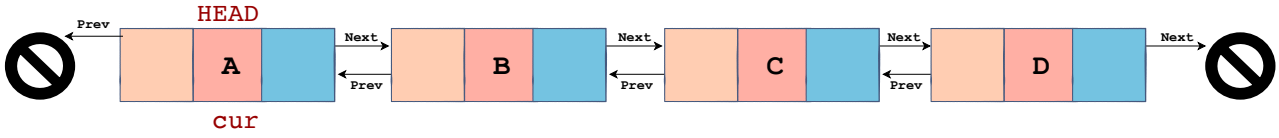


tmp = None

```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

3 of 27

Doubly Linked List: Reverse

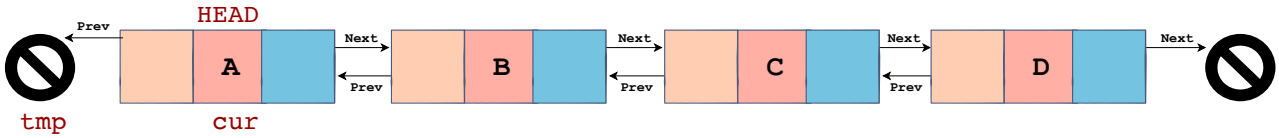


tmp = None

```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

4 of 27

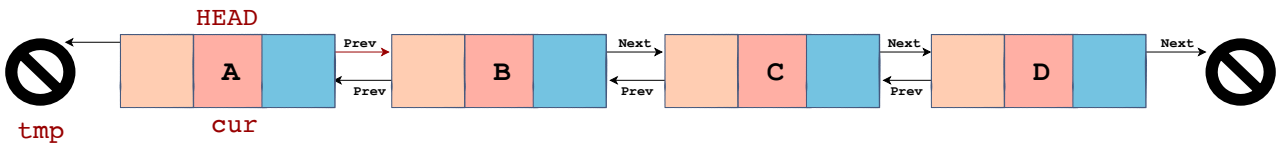
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

5 of 27

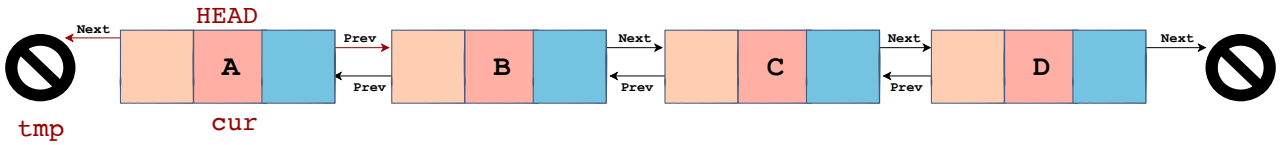
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

6 of 27

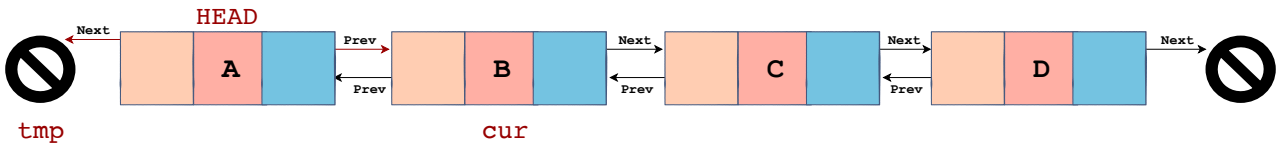
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

7 of 27

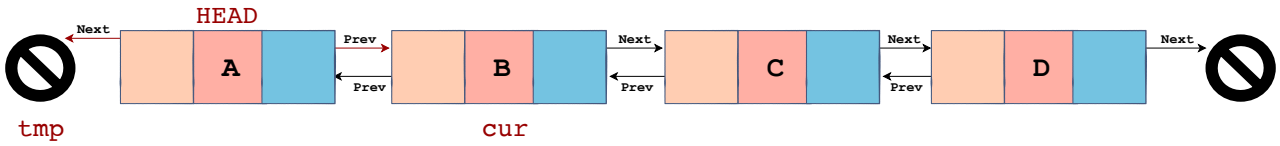
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

8 of 27

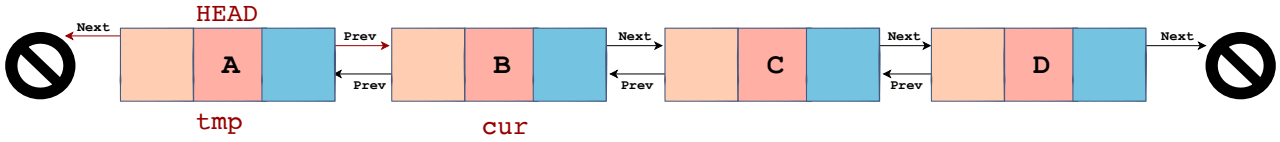
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

9 of 27

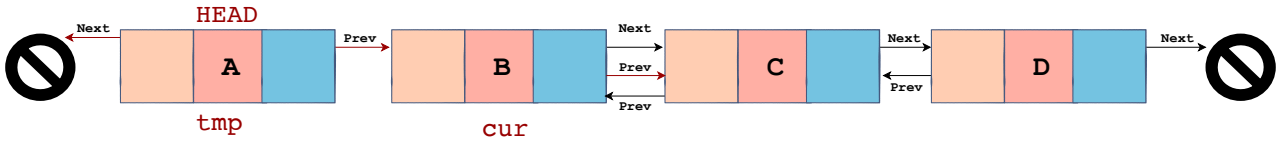
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

10 of 27

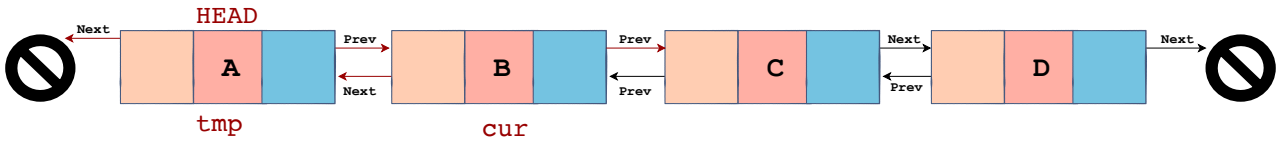
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

11 of 27

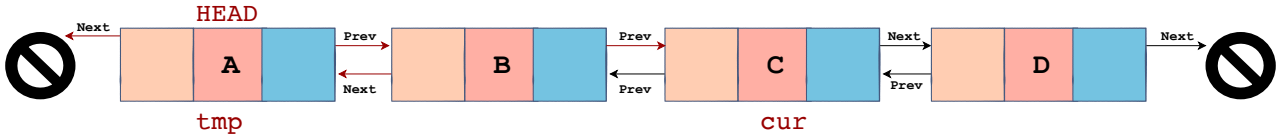
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

12 of 27

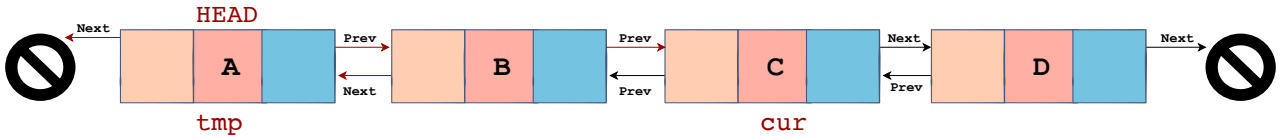
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

13 of 27

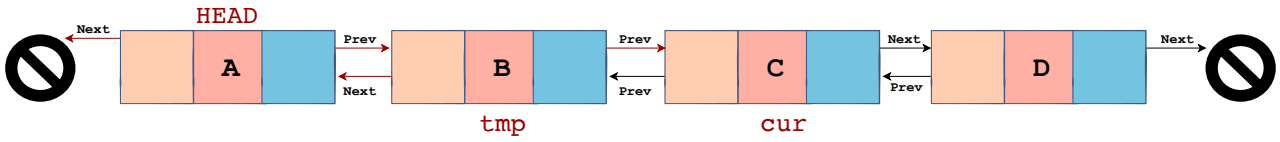
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

14 of 27

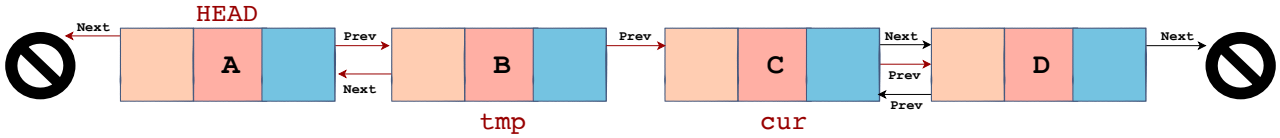
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

15 of 27

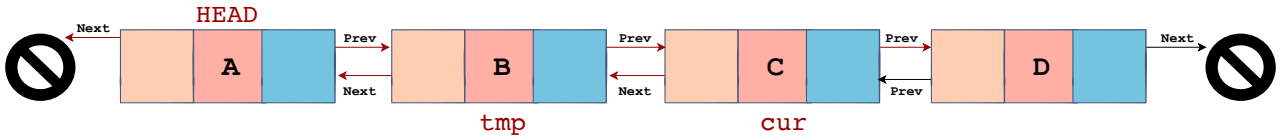
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

16 of 27

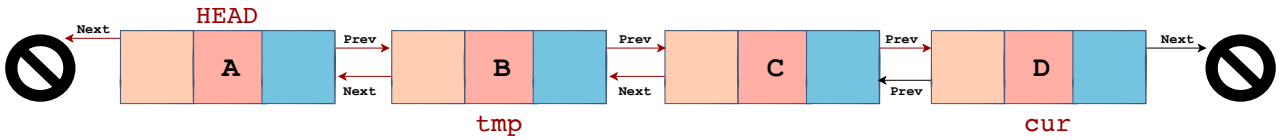
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

17 of 27

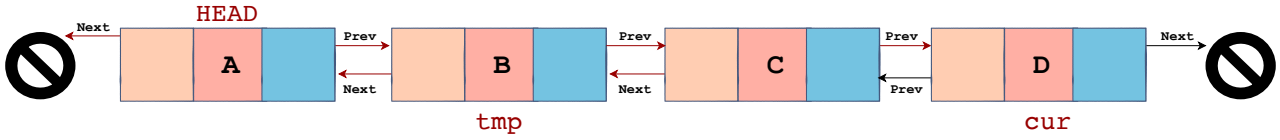
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

18 of 27

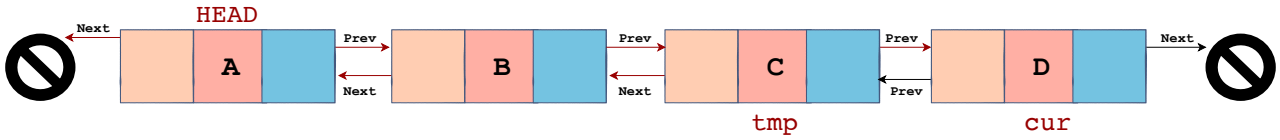
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

19 of 27

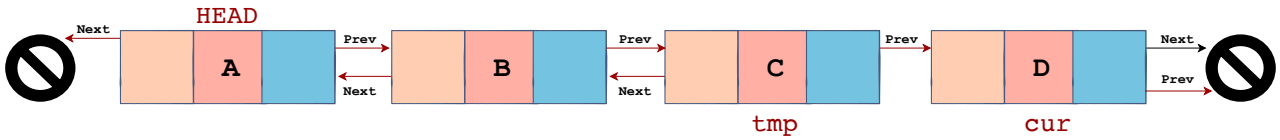
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

20 of 27

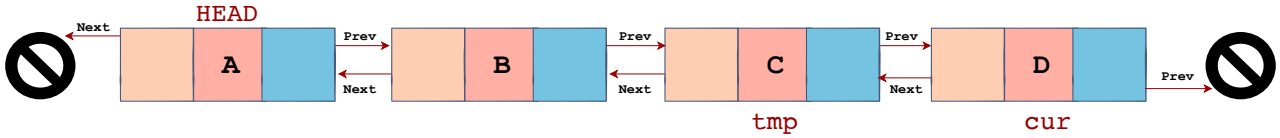
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

21 of 27

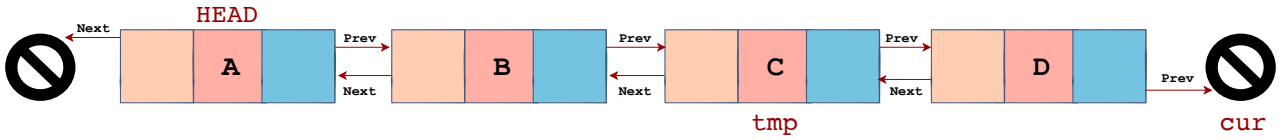
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

22 of 27

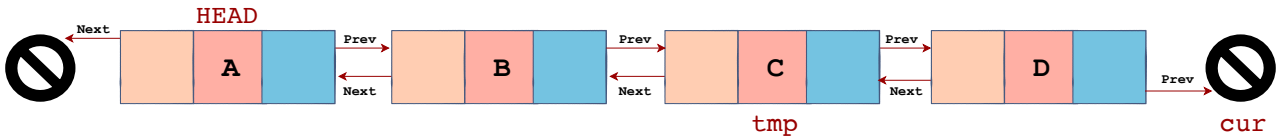
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

23 of 27

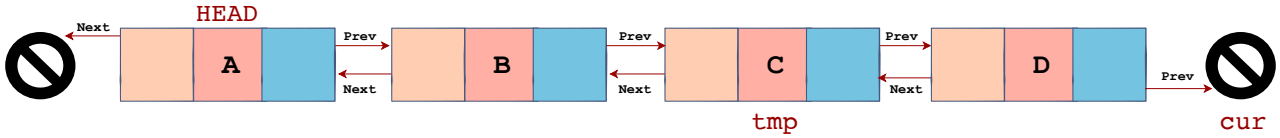
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

24 of 27

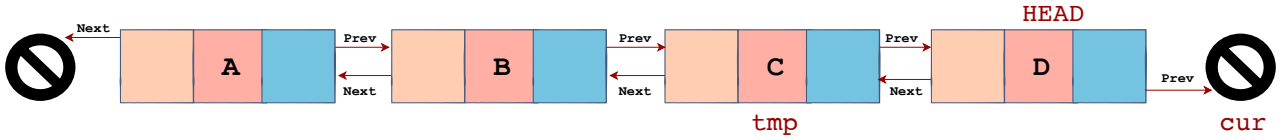
Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

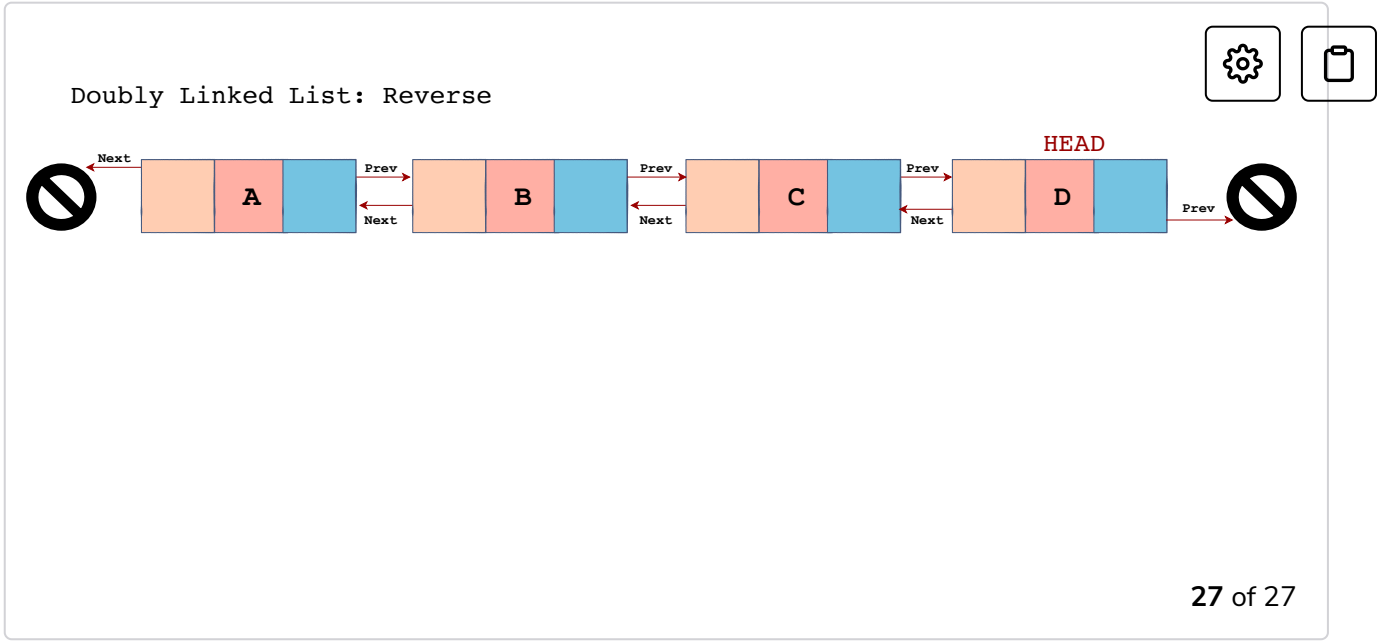
25 of 27

Doubly Linked List: Reverse




```
tmp = None
cur = self.head
while cur:
  tmp = cur.prev
  cur.prev = cur.next
  cur.next = tmp
  cur = cur.prev
if tmp:
  self.head = tmp.prev
```

26 of 27



In the code widget below, we reverse a doubly-linked list. Go ahead and modify the test cases with your own, so you get hands-on practice with our implementation.

```
102
103             # Case 4:
104           else:
105               prev = cur.prev
106               prev.next = None
107               cur.prev = None
108               cur = None
109               return
110         cur = cur.next
111
112     def reverse(self):
113       tmp = None
114       cur = self.head
115       while cur:
116         tmp = cur.prev
117         cur.prev = cur.next
118         cur.next = tmp
119         cur = cur.prev
120       if tmp:
121         self.head = tmp.prev
122
123  dllist·=·DoublyLinkedList()
```

```
124  dllist.append(1)
125  dllist.append(2)
126  dllist.append(3)
127  dllist.append(4)
128  dllist.reverse()
129  dllist.print_list()
```

Output 0.14s

```
4
3
2
1
```

I hope everything has been clear up until now. Now it's time for some challenges! See you in the next lesson.

Back

Delete Node

Next

Exercise: Remove Duplicates

Mark as Completed

Exercise: Remove Duplicates

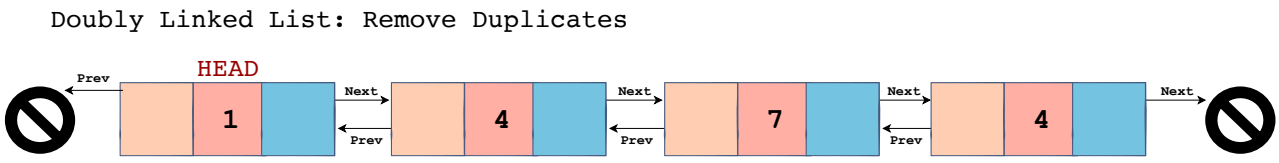
Challenge yourself with an exercise in which you'll have to remove duplicates from a doubly linked list.

We'll cover the following

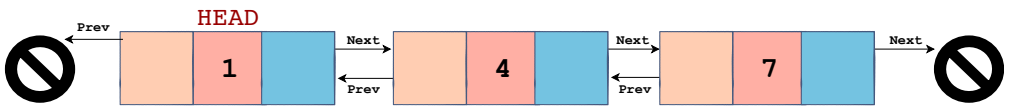
- Problem
- Coding Time!

Problem

In this exercise, you are required to remove the duplicates from a doubly linked list.



1 of 2



2 of 2

In the exercise widget, the `delete_node` method has been given to you. It is a slight modification of the `delete` method we covered in one of the previous lessons. You have to figure out the modification yourself. The hint is to recall the `remove_duplicates` lesson from one of the previous chapters.

Coding Time!

In the code below, the `remove_duplicates` is a class method of the `DoublyLinkedList` class. You cannot see the rest of the code as it is hidden. As `remove_duplicates` 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. Just remove the duplicates from the linked list using the `delete_node` method.

Good luck!

```
1  ··def·remove_duplicates(self):
2  ····pass
3
```

```
  def delete_node(self, node):
    cur = self.head
    while cur:
      if cur == node and cur == self.head:
        # Case 1:
        if not cur.next:
          cur = None
          self.head = None
          return

        # Case 2:
        else:
          nxt = cur.next
          cur.next = None
          nxt.prev = None
          cur = None
          self.head = nxt
          return

      elif cur == node:
        # Case 3:
        if cur.next:
          nxt = cur.next
          prev = cur.prev
          prev.next = nxt
```

Back

Reverse

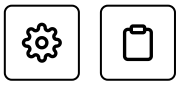
Next

Solution Review: Remove Duplicates

Mark as Completed

Report an Issue

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



Solution Review: Remove Duplicates

This lesson contains the solution review for the challenge of removing duplicates from a doubly linked list.

We'll cover the following

In this lesson, we consider how to remove duplicates from a doubly linked list.

Implementation

Check out the code below:

```
def remove_duplicates(self):
  cur = self.head
  seen = dict()
  while cur:
    if cur.data not in seen:
      seen[cur.data] = 1
      cur = cur.next
    else:
      nxt = cur.next
      self.delete_node(cur)
      cur = nxt
```

remove_duplicates(self)

Explanation

In the method `remove_duplicates`, we will keep track of the duplicates using a Python dictionary which we declare on **line 3**. `cur` is set to `self.head` on **line 2** to help us traverse the linked list using the `while` loop on **line 4**.

In the `while` loop, we have to keep track of the number of times we encounter a data element. Therefore, if `cur.data` is not present in `seen`, we set the value of the key `cur.data` to `1` on **line 6** to indicate that we have encountered it once while traversing the linked list. In the next line, we update `cur` to `cur.next` to iterate to the next node.

On the other hand, if `cur.data` is present in `seen`, we jump to the `else` portion on **line 8**. This implies that `cur` in the current iteration has already been encountered in the previous iterations and is present in `seen`. On **line 9**, we save the next node of `cur` in `nxt` to keep with the traversal after we remove the duplicate node. We call the class method `self.delete_node(cur)` to delete the duplicate node on **line 10** and then set `cur` to `nxt` for the next iteration on **line 11**.

Now let's discuss the `delete_node` method. You can see the modifications through the highlighted lines in the code snippet below. Instead of matching `key` with `cur.data`, we are comparing the entire `node` passed into the method with `cur`.

```
12        else:
13          nxt = cur.next
14          cur.next = None
15          nxt.prev = None
16          cur = None
17          self.head = nxt
18          return
19
20      elif cur == node:
21        # Case 3:
22        if cur.next:
```

```
22        if cur.next:
23          nxt = cur.next
24          prev = cur.prev
25          prev.next = nxt
26          nxt.prev = prev
27          cur.next = None
28          cur.prev = None
29          cur = None
30          return
31
32        # Case 4:
33        else:
34          prev = cur.prev
35          prev.next = None
36          cur.prev = None
37          cur = None
38          return
39      cur = cur.next
```

delete_node(self, node)

In the code widget below, we have the entire implementation of `DoublyLinkedList` that we have learned so far in this chapter. Go ahead and explore it yourself!

```
163    def remove_duplicates(self):
164        cur = self.head
165        seen = dict()
166        while cur:
167            if cur.data not in seen:
168                seen[cur.data] = 1
169                cur = cur.next
170            else:
171                nxt = cur.next
172                self.delete_node(cur)
173                cur = nxt
174
175
176  dllist = DoublyLinkedList()
177  dllist.append(8)
178  dllist.append(4)
179  dllist.append(4)
180  dllist.append(6)
181  dllist.append(4)
182  dllist.append(8)
```

```
183  dllist.append(4)
184  dllist.append(10)
185  dllist.append(12)
186  dllist.append(12)
187
188
189  dllist.remove_duplicates()
190  dllist.print_list()
```

```
Output                                                    0.19s

  8
  4
  6
  10
  12
```

Hope you had fun with this lesson! Now brace yourself for another challenge in the next lesson. All the best!

Back

Exercise: Remove Duplicates

Next

Exercise: Pairs with Sums

Mark as Completed

Report an Issue

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

Exercise: Pairs with Sums

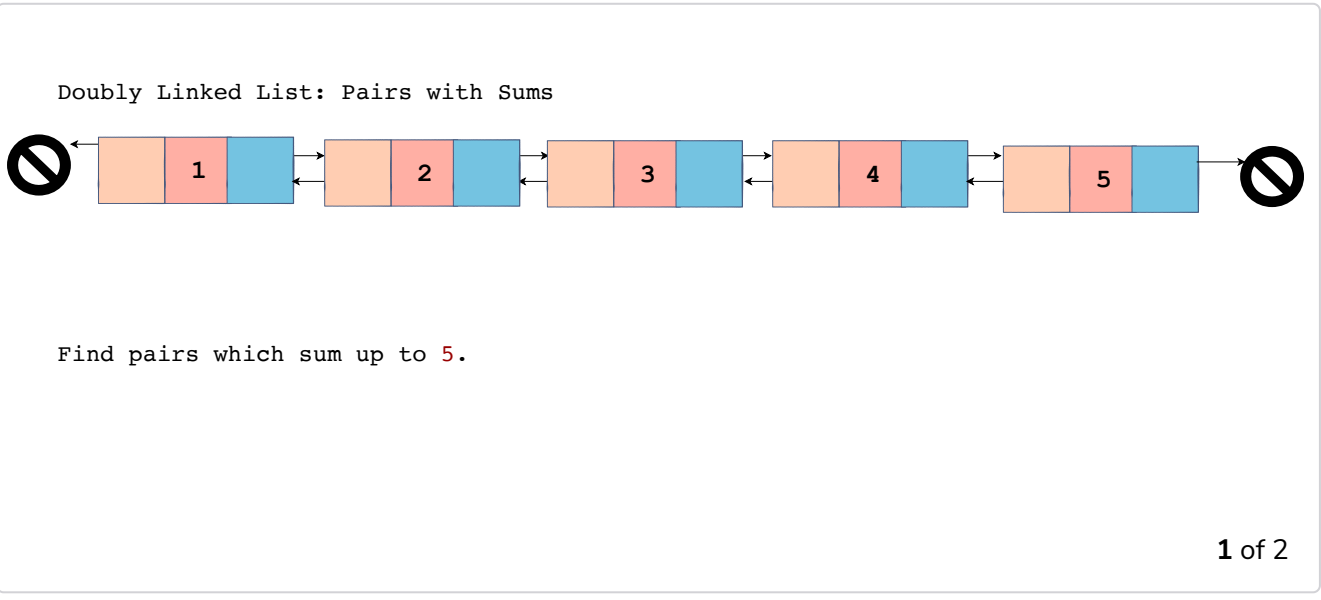
Challenge yourself with an exercise in which you'll have to find pairs from a doubly linked list which sum to some number.

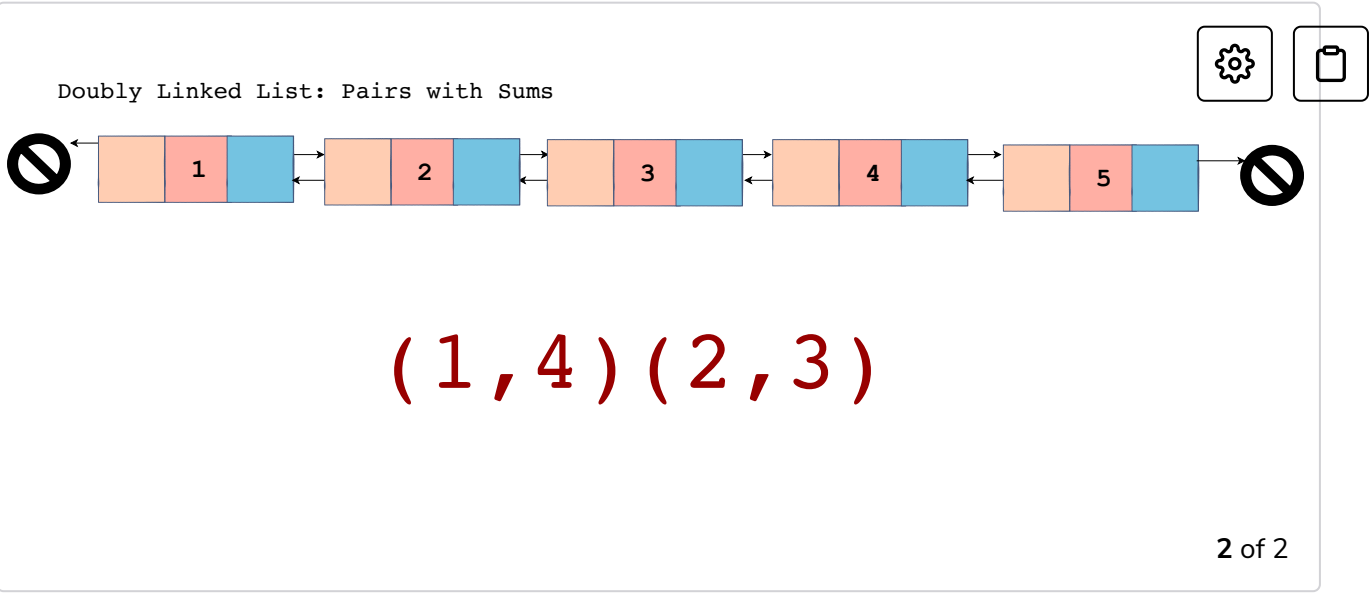
We'll cover the following

Problem

In this exercise, you are required to find pairs from a doubly linked list which sum to a specified number.

An example is illustrated for you below.





Coding Time!

In the code below, `pairs_with_sum` is a class method of the `DoublyLinkedList` class. You cannot see the rest of the code as it is hidden. As `pairs_with_sum` 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 function prototype.

The pairs from `pairs_with_sum` have to follow this condition; for a pair `(a,b)`, `a` should come before `b` in the doubly linked list while it is traversed from the head node.

For this exercise, you are required to return a **Python list** containing the pairs as string types in the following format:

```
(a,b)
```

For example, the pairs for the linked list shown in the illustration are as follows:

```
["(1,4)","(2,3)"]
```

Failure to return the answers in the required format will fail the test cases.

Good luck!

```
1    def pairs_with_sum(self, sum_val):
2        cur = self.head
3        pairs = []
4        while cur:
5            tmp = cur.next
6            while tmp:
7                if cur.data + tmp.data == sum_val:
8                    pairs.append("(" + str(cur.data)
9                    break
10               tmp = tmp.next
11           cur = cur.next
12       return pairs
```

Show Results | Show Console

0.14s

5 of 5 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	pairs_with_sum(1->2->3->4->5,sum_val=5)	['(1,4)', '(2,3)']	['(1,4)', '(2,3)']	Succeeded
✓	pairs_with_sum(5->4->3->2->1,sum_val=5)	['(4,1)', '(3,2)']	['(4,1)', '(3,2)']	Succeeded
✓	pairs_with_sum(1->2->3->4->5,sum_val=0)	[]	[]	Succeeded
✓	pairs_with_sum(2->4->6->8->10,sum_val=14	['(4,10)', '(6,8)']	['(4,10)', '(6,8)']	Succeeded

Result	Input	Expected Output	Actual Output	Reason
✓	`pairs_with_sum(,sum_val=0)`	`[]`	`[]`	Succeeded

Back

Next

Solution Review: Remove Duplicates

Solution Review: Pairs with Sums

Mark as Completed

Report an Issue

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

educative

Solution Review: Pairs with Sums

This lesson contains the solution review for the challenge of finding pairs in a doubly linked list which sum up to the given number.

We'll cover the following

Here is an overview of our solution to finding a pair of nodes in a doubly linked list that sums up to a specific number.

Implementation

Here is the coding solution in Python for the previous challenge:

```
def pairs_with_sum(self, sum_val):
  pairs = list()
  p = self.head
  q = None
  while p:
    q = p.next
    while q:
      if p.data + q.data == sum_val:
          pairs.append("(" + str(p.data) + "," + s
      q = q.next
    p = p.next
  return pairs
```

Explanation

On **line 2**, `pairs` is initialized to an empty Python list. In the next lines (**lines 3-4**), `p` and `q` are set equal to `self.head` and `None` respectively. We will use both these pointers (`p` and `q`) to make pairs out of the doubly linked list by using two `while` loops afterward.

The outer `while` loop on **line 5** will run until `p` becomes equal to `None` while the inner loop on **line 7** will run for every iteration of the outer loop until `q` becomes `None`.

On **line 6**, we set `q` equal to `p.next` as we have to start pairing nodes from the next node of the current node as we would already have checked the pairing with all previous nodes. Then we'll check if the sum of `p.data` and `q.data` equals `sum_value` or not on **line 8**. If it is, then we append `p.data` and `q.data` to the list we declared at the beginning of the `pairs_with_sum` method. If it does not, we move on to the next node of `q` by updating `q` to `q.next` on **line 10**. In each iteration of the inner loop, we pair the data of `p` with all the data of the nodes after `p` using `q` and then check each of these pairs to see if they sum up to `sum_value`. This process is repeated for every node in the linked list as `p` updates to `p.next` on **line 11** in the outer `while` loop. After the outer loop terminates, `pairs` is returned from the method on **line 12**.

You can play around with all the methods that we have implemented for the `DoublyLinkedList` class in the code widget provided below.

```
169                 cur = cur.next
170             else:
171                 nxt = cur.next
172                 self.delete_node(cur)
173                 cur = nxt
174 
175     def pairs_with_sum(self, sum_val):
176       pairs = list()
177       p = self.head
178       q = None
179       while p:
180         q = p.next
181         while q:
```

```
181         while q:
182           if p.data + q.data == sum_val:
183               pairs.append("(" + str(p.data) + "," -
184           q = q.next
185         p = p.next
186       return pairs
187
188
189   dllist = DoublyLinkedList()
190   dllist.append(1)
191   dllist.append(2)
192   dllist.append(3)
193   dllist.append(4)
194   dllist.append(5)
195
196   print(dllist.pairs_with_sum(5))
```

Output 0.16s

```
['(1,4)', '(2,3)']
```

Now this lesson marks an end to the content on linked lists. Get ready to solve problems using another data structure in the next chapter!

Back

Next

Exercise: Pairs with Sums

Quiz

Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-pairs-with-sums__doubly-linked-lists__data-structures-and-algorithms-in-python)