Introduction and Insertion

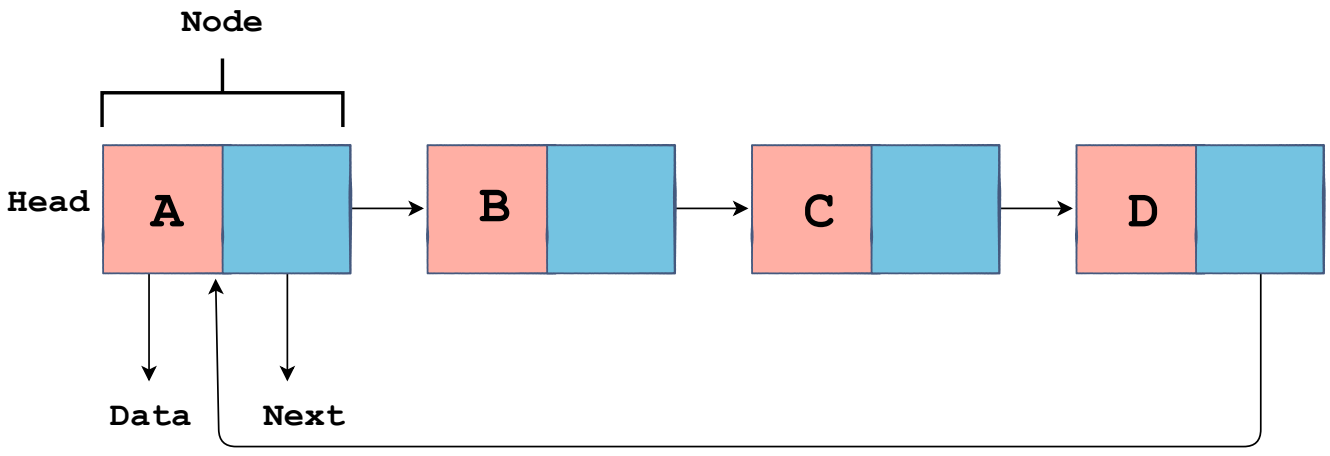
In this lesson, you will be introduced to circular linked lists and you will learn how to insert elements into a circular linked list.

We'll cover the following

Introduction

First of all, let's talk about what a circular linked list is. It is very similar to a singly linked list except for the fact that the next of the tail node is the head node instead of null.

Below is an illustration to help you visualize a circular linked list:



Circular Linked List

You can see from the illustration above that the circular linked list contains the same kind of nodes as a singly linked list. As the name *circular* suggests, the tail node points to the head of the linked list instead of pointing to null which makes the linked list circular. Now let's implement it in Python:

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


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

    def prepend(self, data):
      pass

    def append(self, data):
      pass

    def print_list(self):
      pass
```

class Node and class CircularLinkedList

The *Node* class is the same as before. Also, the constructor of *CircularLinkedList* class is identical to the constructor of the *LinkedList* class. However, the methods `prepend`, `append`, and `print_list` will be different this time.

append

Appending to a circular linked list implies inserting the new node after the node that was previously pointing to the head of the linked list.

Let's have a look at the code below for the `append` method:

```
1  def append(self, data):
2    if not self.head:
3        self.head = Node(data)
4        self.head.next = self.head
5    else:
6        new_node = Node(data)
7        cur = self.head
8        while cur.next != self.head:
9            cur = cur.next
10       cur.next = new_node
11       new_node.next = self.head
```

append(self, data)

One case is to append to an empty linked list. In this case, we make the new node the head of the linked list, and its next node is itself. So, we'll check if the linked list is empty. If `self.head` is `None`, then the condition on **line 2** will evaluate to `true` and we will initialize `self.head` to a new *Node* based on the input parameter `data`. As this is the `append` method for a circular linked list, we make the next of the head point to itself on **line 4**.

Now let's look at the case where the linked list is not empty. In this case, we iterate the linked list to get to the last element whose next is the head node and insert the new element after that last node. On **line 6**, we initialize `new_node` by calling the constructor of the *Node* class and passing `data` to it. Next, we have to find an appropriate position in the linked list to insert `new_node`. For this, we set `cur` to `self.head` on **line 7** and set up a `while` loop on **line 8** which executes until the next node of the `cur` does not equal `self.head`. `cur` is updated to `cur.next` in the body of the `while` loop on **line 9** to help us traverse the linked list. After the `while` loop terminates, `cur` will be the node that points to the head node and we will insert `new_node` after `cur`. Hence, we set `cur.next` equal to `new_node` on **line 10**.

As we have to complete our circular chain, we set `new_node.next` to point to `self.head` on **line 11**. This completes our implementation for the `append` method.

print_list

To verify the `append` method, we will need to print the circular linked list. Let's see how we would do this. Check out the code below:

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

print_list(self)

To print out the linked list, we set `cur` to `self.head` on **line 2** to get a starting point. Then we set up a `while` loop on **line 4** which will run until `cur` becomes `None`. In the `while` loop, we simply print `cur.data` on **line 5** and update `cur` to `cur.next` on **line 6** to proceed to the next node. Note that in a circular linked list, no node points to `None`. Instead, the last node points to the head of the linked list. To break the `while` loop, we check if we have reached the head of the linked list using an if-condition on **line** 7. If we have, we break out from the loop with the help of the `break` statement on **line 8**.

As you can see, the `print_list` method was reasonably straightforward. Let's move on to the `prepend` method.

prepend

To prepend an element to the linked list implies to make it the head of the linked list. Let's see how we can code this functionality for circular linked lists in Python:

```
1  def prepend(self, data):
2    new_node = Node(data)
3    cur = self.head
4    new_node.next = self.head
5
6    if not self.head:
7        new_node.next = new_node
8    else:
9        while cur.next != self.head:
10           cur = cur.next
11       cur.next = new_node
12   self.head = new_node
```

prepend(self, data)

On **line 2**, `new_node` is initialized to a new object of the *Node* class based on `data`. This new node will eventually become the head of the linked list while the current head has to be pushed ahead in the linked list. Therefore, we set `new_node.next` to `self.head` on **line 4** so that the next of the new node will be the previous head of the linked list. `cur` is initialized to `self.head` on **line 2** to help us with the rest of the code.

Next, as with the `append` method, we check if the linked list in which we are about to insert is an empty linked list or not. If `self.head` is `None`, we simply set `new_node.next` to `new_node` on **line 7** to give the circular effect. Otherwise, if `self.head` is not `None`, we proceed to the `else` part of the if-condition. In the `else` part, we set up a `while` loop on **line 9** in which we update `cur` to `cur.next` on **line 10**. This `while` loop executes until `cur.next` is equal to `self.head`. This essentially means that the `while` loop will run until `cur` is the last node or the node before the head node, after which we will set the `cur.next` to `new_node` on **line 11** to implement the circular chain of our linked list.

Finally, we set `self.head` to `new_node` on **line 12** and it becomes the head of the linked list.

As we have individually covered all three methods we discussed at the beginning of the lesson, it is now time to test them. Check out the full code in the code widget below:

```
18          else:
19              while cur.next != self.head:
20                  cur = cur.next
21              cur.next = new_node
22          self.head = new_node
23
24      def append(self, data):
25          if not self.head:
26              self.head = Node(data)
27              self.head.next = self.head
28          else:
29              new_node = Node(data)
30              cur = self.head
31              while cur.next != self.head:
32                  cur = cur.next
33              cur.next = new_node
34              new_node.next = self.head
35
36      def print_list(self):
37          cur = self.head
38
39          while cur:
40              print(cur.data)
41              cur = cur.next
42              if cur == self.head:
43                  break
44
45
46  cllist = CircularLinkedList()
47  cllist.append("C")
48  cllist.append("D")
49  cllist.prepend("B")
50  cllist.prepend("A")
51  cllist.print_list()
```

Output 0.49s

```
A
B
C
D
```

In this lesson, you have been introduced to the concept of a circular linked list and its basic implementation. I hope you were able to easily identify the changes made from the implementation of a singly linked list. In further lessons, we'll go over problems and challenges related to circular linked lists. Stay tuned!

Back

Quiz

Next

Remove Node

Mark as Completed

Report an Issue

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

Remove Node

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

We'll cover the following

In this lesson, we investigate how to remove nodes in a circular linked list and code the method in Python.

There is an assumption that we will make before diving into the implementation:

- The occurrences of nodes will be unique, i.e., there will be no duplicate nodes in the circular linked list that we'll test on.

This is because the code that we will write will only be responsible for removing the first occurrence of the key provided to be deleted.

Implementation

Now let's go ahead and jump to the implementation of `remove` in Python:

```
def remove(self, key):
  if self.head:
    if self.head.data == key:
      cur = self.head
      while cur.next != self.head:
        cur = cur.next
      if self.head == self.head.next:
        self.head = None
```

```
 8           self.head = None
 9         else:
10           cur.next = self.head.next
11           self.head = self.head.next
12       else:
13         cur = self.head
14         prev = None
15         while cur.next != self.head:
16           prev = cur
17           cur = cur.next
18           if cur.data == key:
19             prev.next = cur.next
20             cur = cur.next
```

Explanation

The code is divided into parts based on whether or not we are deleting the head node.

If the condition on **line 2** is `False`, it implies that we are dealing with an empty list and we just return from the method. Otherwise, the execution jumps to **line 3**.

If we are deleting the head node, then the condition on **line 3** will be true, and the execution will jump to **line 4** where we set `cur` equal to `self.head`. To delete the head node, we have to update the node that points to the head node and the node that the head node points to. As a result, we set up a `while` loop on **line 5** which will run until `cur.next` points to `self.head`. We keep updating `cur` to `cur.next` on **line 6**. After the `while` loop terminates, `cur` will be the last node in the linked list which will point to the head node.

At this point, we also need to consider if `self.head` is the only element in the circular linked list. If it is the only element, then `self.head` is pointing to itself. So, we check on **line 7** if that's the case, then `self.head` is set to `None` on **line 8**. On the other hand, if it's not the only element and there is another

element to replace the head node, we set `cur.next` equal to `self.head.next` on **line 10**. We have updated the node which was previously pointing to the head node. In the next line, we'll update `self.head` to `self.head.next` which removes the previous head from the linked list and updates the head of the linked list.

Now we'll focus on the else part on **line 12** which refers to the case where we are not deleting the head node. To traverse the linked list and to keep track of the current and previous nodes, we initialize `cur` to `self.head` and `prev` to `None` (**lines 13-14**). On **lines 16-17**, we update `prev` to `cur` and `cur` to `cur.next` to keep track of the nodes in the `while` loop which will traverse the entire linked list once. Next, we have to check for the node to be deleted which we check using the condition on **line 18**. If we find the node to be deleted, we set the next of the previous node (`prev.next`) to the next of the current node (`cur.next`) on **line 19** and then set `cur` to `cur.next` on **line 20** to move along in the linked list.

Below is the entire implementation of the circular linked list that we have covered so far. Feel free to play around with it!

```
 1  class Node:
 2      def __init__(self, data):
 3          self.data = data
 4          self.next = None
 5
 6
 7  class CircularLinkedList:
 8      def __init__(self):
 9          self.head = None
10
11      def prepend(self, data):
12          new_node = Node(data)
13          cur = self.head
14          new_node.next = self.head
15
16          if not self.head:
17              new_node.next = new_node
18          else:
19              while cur.next != self.head:
```

```
20                 cur = cur.next
21             cur.next = new_node
22         self.head = new_node
23
24     def append(self, data):
25         if not self.head:
26             self.head = Node(data)
27             self.head.next = self.head
28         else:
```

CircularLinkedList

In the next lesson, we will look at another problem regarding circular linked lists. See you there!

Back

Introduction and Insertion

Next

Split Linked List into Two Halves

Mark as Completed

Report an Issue

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

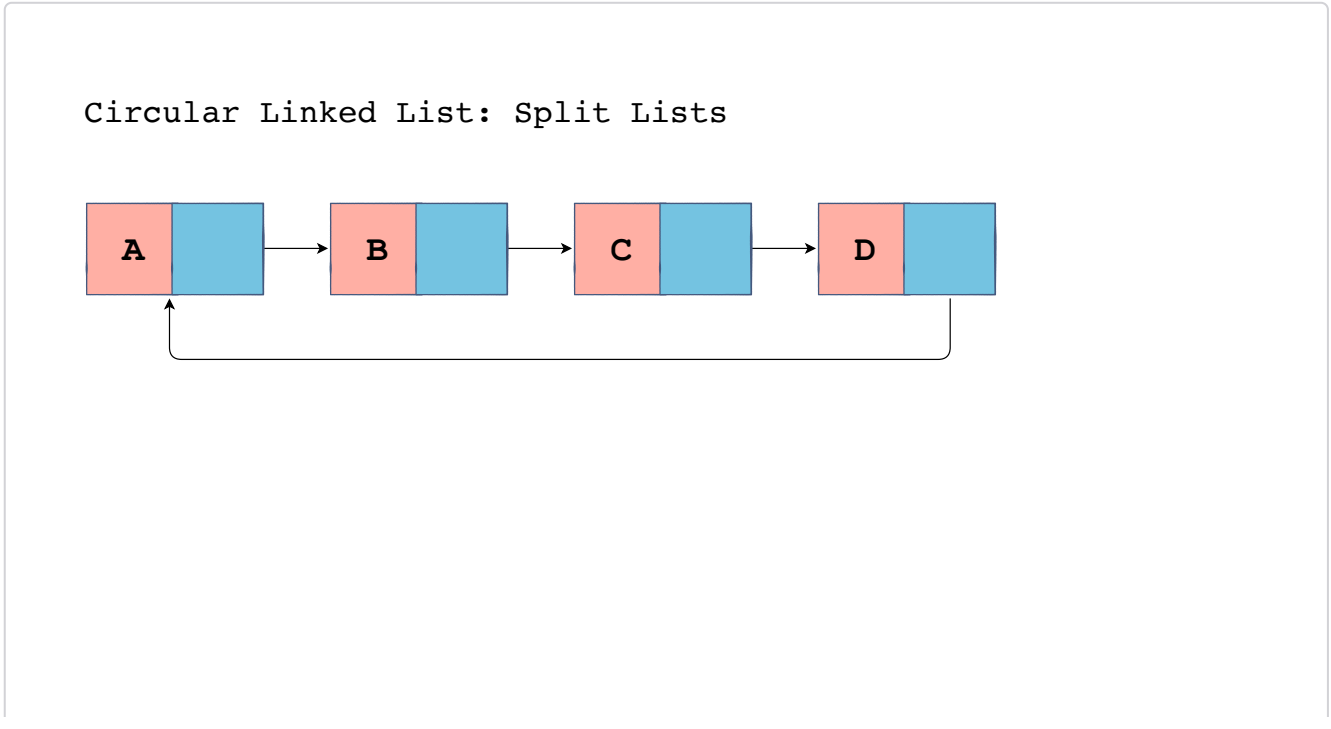
Split Linked List into Two Halves

In this lesson, you will learn how to split a circular linked list into two halves in Python.

We'll cover the following

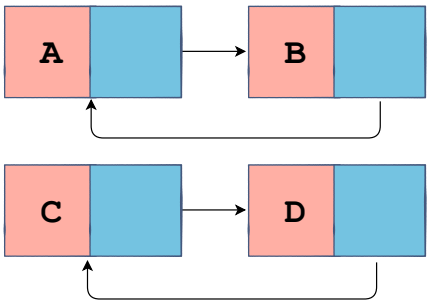
In this lesson, we investigate how to split one circular linked list into two separate circular linked lists and then code the solution in Python.

First of all, let's clarify what we mean by splitting a circular linked list by taking a look at the illustration below.



1 of 2

Circular Linked List: Split Lists



2 of 2

To approach this problem, we'll find the length of the circular linked list and calculate the midpoint. Once that is done, we'll split the linked list around the midpoint. One half will be made by trimming the original linked list while the rest of the elements will be pushed into a new circular linked list.

`__len__()`

Let's see how we calculate the length of a circular linked list in Python:

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

__len__(self)

Explanation

The `__len__` method has been defined with underscores before and after the `len` keyword so that it overrides the `len` method to operate on a circular linked list.

Calculating the length of the circular linked list is very straightforward. We declare `cur` equal to `self.head` on **line 2** to give a start for traversing the circular linked list. `count` is set to `0` initially on **line 3**. Next, we traverse the circular linked list using a `while` loop on **line 4** by updating `cur` to `cur.next` on **line 6**. On **line 5**, we increment `count` to keep track of the number of nodes in a circular linked list. If `cur` becomes equal to `self.head`, we break out of the loop (**lines 7-8**). Finally, we `return` count on **line 9**.

As you see, the length method was as simple as that. Now let's go over the `split_list` method.

Implementation

```
1  def split_list(self):
2      size = len(self)
3
4      if size == 0:
5          return None
6      if size == 1:
7          return self.head
8
9      mid = size//2
10     count = 0
11
12     prev = None
13     cur = self.head
14
```

```
15      while cur and count < mid:
16          count += 1
17          prev = cur
18          cur = cur.next
19      prev.next = self.head
20
21      split_cllist = CircularLinkedList()
22      while cur.next != self.head:
23          split_cllist.append(cur.data)
24          cur = cur.next
25      split_cllist.append(cur.data)
26
27      self.print_list()
28      print("\n")
```

split_list(self)

Explanation

Once we calculate the midpoint using the `len` method, we'll traverse the linked list until we reach the midpoint and then reorient the pointers to split the linked list. On **line 2**, we call our `len` method that we just implemented to calculate the length of the circular linked list object on which the method `split_list` is called and assign it to the variable `size`.

Next, we have if-conditions to handle two edge cases on **lines 4-7**. If `size` turns out to be `0`, we return `None`, while if `size` is `1`, we return `self.head` which is going to be the only node in the linked list. These two cases imply that no splitting can take place.

On **line 9**, we calculate the midpoint (`mid`) by dividing the length by `2` and flooring the answer using the `//` operator.

Now we are going to analyze the following code (**lines 10-19**):

```
    count = 0

    prev = None
    cur = self.head

    while cur and count < mid:
        count += 1
        prev = cur
        cur = cur.next
    prev.next = self.head
```

`count` is initialized to `0` on **line 10**. On **lines 12-13**, we declare two pointers `prev` and `cur` which are initially set to `None` and `self.head`, respectively. These variables will help us keep track of the previous and current nodes as we traverse the circular linked list. Using the `while` loop, we traverse through the linked list until `count` becomes equal to `mid` or `cur` becomes `None`. After `prev` becomes equal to `cur`, `cur` becomes `cur.next` in the `while` loop (**lines 17-18**). Also, we increment `count` on **line 16** so that we only traverse up to the midpoint. When `count` becomes equal to or greater than `mid`, we reach the midpoint from where we have to split. To complete the splitting for the first linked list, we set `prev.next` (next of the last node in the first linked list) to `self.head` on **line 2** to make the first list linked circular. At this point, we are done with our first linked list, and the first node of the second linked list is held in variable `cur`. Now let's go ahead and have a look at the part concerning the second linked list (**lines 21-25**):

```
    split_cllist = CircularLinkedList()
    while cur.next != self.head:
        split_cllist.append(cur.data)
        cur = cur.next
    split_cllist.append(cur.data)
```

We initialize `split_cllist` on **line 21** to an empty circular linked list. Then we traverse the original linked list using a `while` loop until we reach the very last node of the original linked list which points to the head node. In every iteration, we append `cur.data` to our newly created linked list `split_cclist` on **line 23** and update `cur` to `cur.next` on **line 24** to go the next node. Finally when we reach the end of the original linked list as `cur.next` equals `self.head`, we terminate the `while` loop and append the data of `cur` to `split_cclist` on **line 25**. The `append` method of the `CircularLinkedList` already handles all the insertions for us. Finally, we have completed the other half of splitting the initial linked list.

On **lines 27-29**, we print both the linked lists for you to see the split versions of our original linked list.

I hope this implementation was easy to understand.

Below is the entire implementation with a test case of even-length linked lists. You can further verify the implementation by testing on odd-length linked lists.

```
            count += 1
            prev = cur
            cur = cur.next
        prev.next = self.head

        split_cllist = CircularLinkedList()
        while cur.next != self.head:
            split_cllist.append(cur.data)
            cur = cur.next
        split_cllist.append(cur.data)

        self.print_list()
        print("\n")
        split_cllist.print_list()


# A -> B -> C -> D -> ...
# A -> B -> ... and C -> D -> ...

cllist = CircularLinkedList()
```

```
89  cllist = CircularLinkedList()
90  cllist.append("A")
91  cllist.append("B")
92  cllist.append("C")
93  cllist.append("D")
94  cllist.append("E")
95  cllist.append("F")
96
97  cllist.split_list()
```

In the next lesson, we'll have a look at the Josephus Problem. See you there!

Back

Remove Node

Next

Josephus Problem

Mark as Completed

Report an Issue

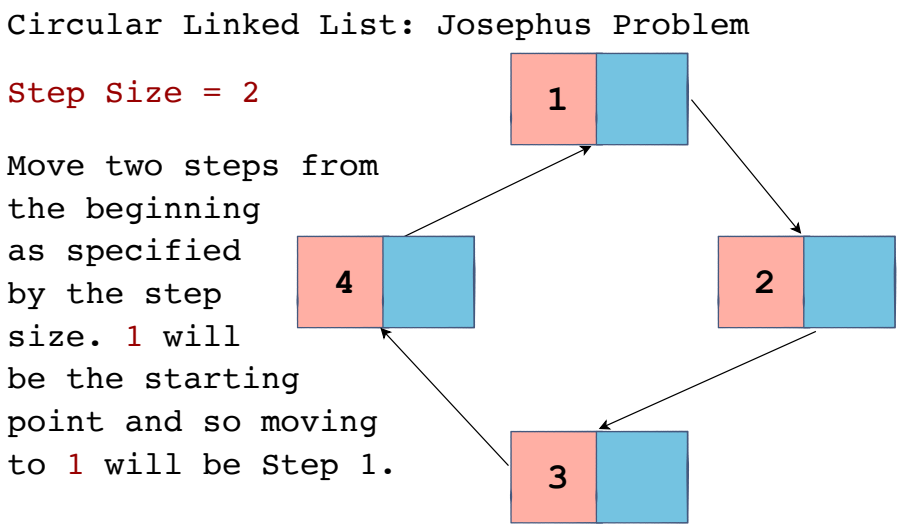
Ask a Question (https://discuss.educative.io/tag/split-linked-list-into-two-halves__circular-linked-lists__data-structures-and-algorithms-in-python)

Josephus Problem

In this lesson, we will learn how to solve the Josephus Problem using a circular linked list in Python.

We'll cover the following

In this lesson, we investigate how to solve the “Josephus Problem” using the circular linked list data structure. Let’s find out what the “Josephus Problem” is through an example in the illustration below:

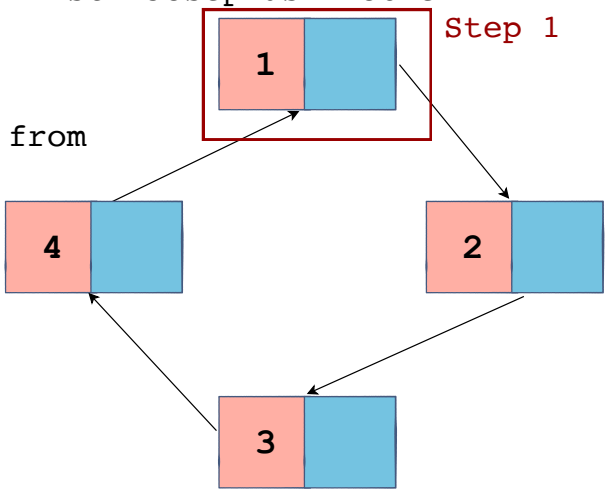


1 of 10

Circular Linked List: Josephus Problem

Step Size = 2

Move two steps from the beginning as specified by the step size

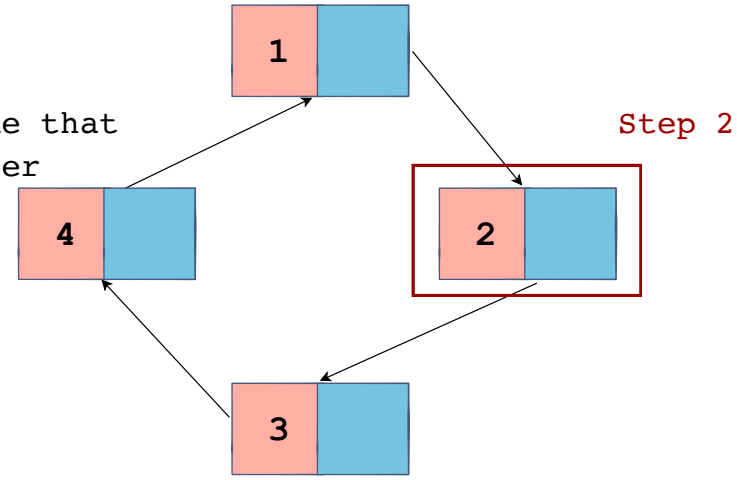


2 of 10

Circular Linked List: Josephus Problem

Step Size = 2

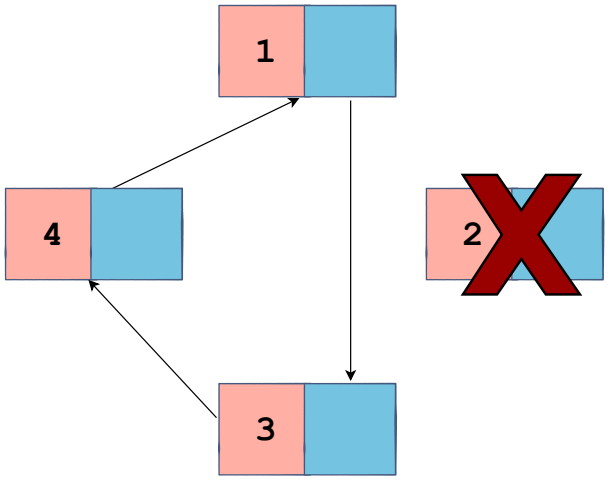
Remove the node that you are on after you have completed the steps



3 of 10

Circular Linked List: Josephus Problem

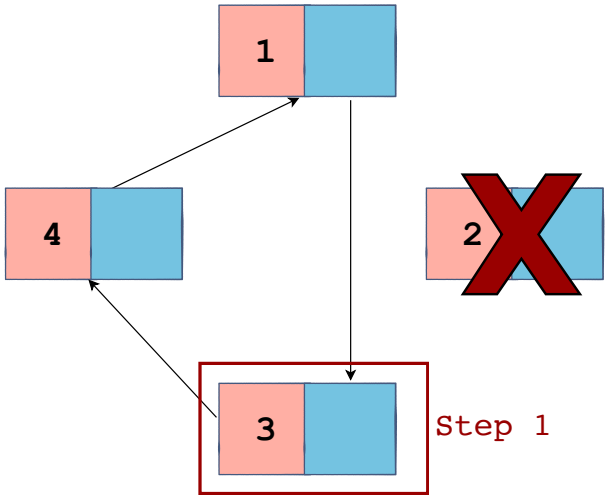
Step Size = 2



4 of 10

Circular Linked List: Josephus Problem

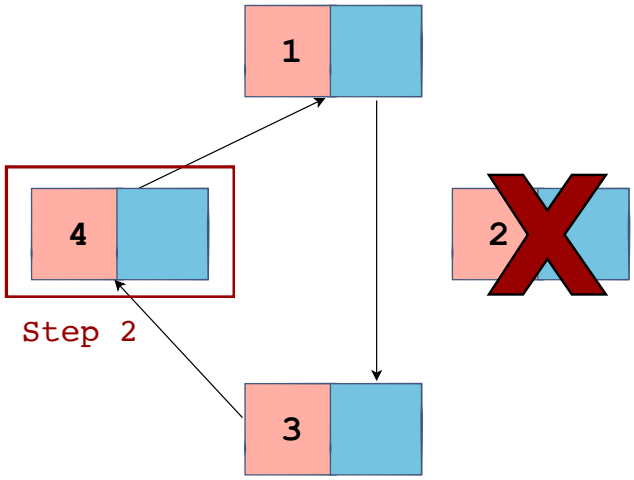
Step Size = 2



5 of 10

Circular Linked List: Josephus Problem

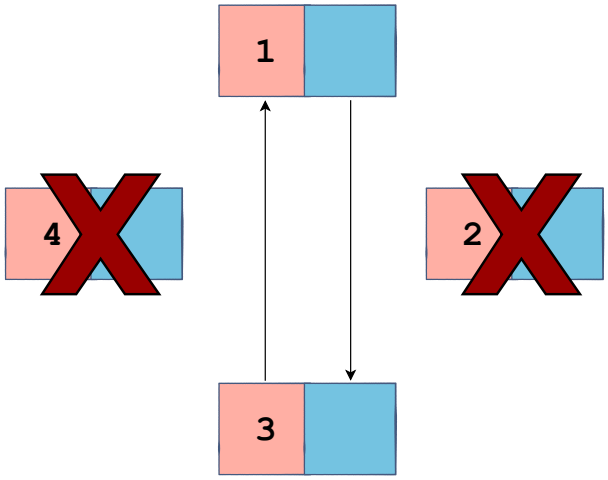
Step Size = 2



6 of 10

Circular Linked List: Josephus Problem

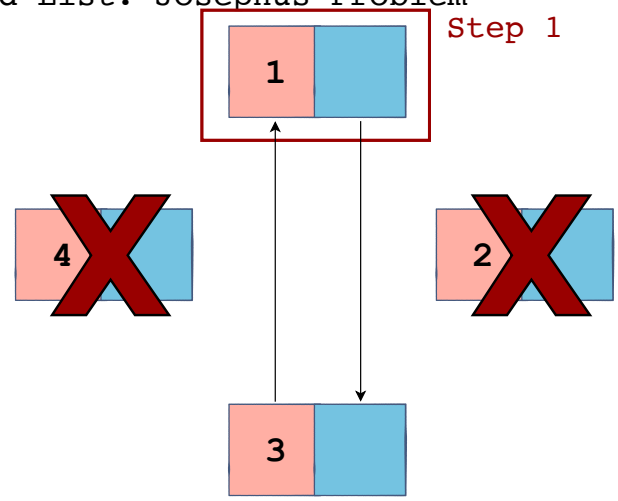
Step Size = 2



7 of 10

Circular Linked List: Josephus Problem

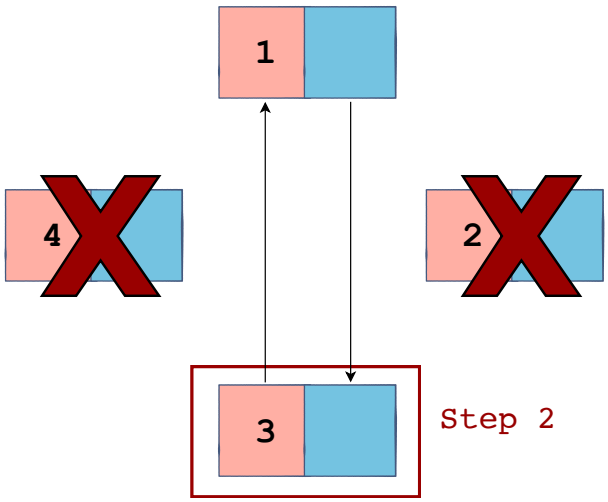
Step Size = 2



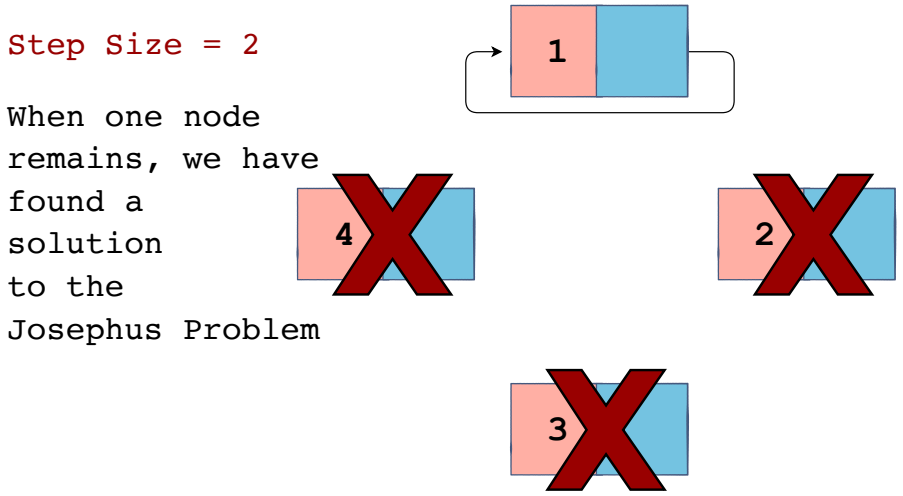
8 of 10

Circular Linked List: Josephus Problem

Step Size = 2



9 of 10



10 of 10

After having a look at the illustration, you'll hopefully understand the Josephus Problem. For this lesson, we have to count out the nodes from the linked list one by one according to the step size until one node remains. To solve this problem, we will tweak the `remove` method from one of the previous lessons so that we can remove nodes by passing the node itself instead of a key. To avoid confusion, we'll use the code from `remove` and paste it in a new method called `remove_node` with some minor modifications.

The modifications are as follows:

```
if self.head.data == key:
```

changes to

```
if self.head == node:
```

and

```
if cur.data == key:
```

changes to

```
if cur == node:
```

As you can see, instead of comparing the data of the node for a match, we compare the entire node itself.

You can check out the method below where the changed code is highlighted:

```
def remove_node(self, node):
  if self.head == node:
    cur = self.head
    while cur.next != self.head:
      cur = cur.next
    if self.head == self.head.next:
      self.head = None
    else:
      cur.next = self.head.next
      self.head = self.head.next
  else:
    cur = self.head
    prev = None
    while cur.next != self.head:
      prev = cur
      cur = cur.next
      if cur == node:
        prev.next = cur.next
        cur = cur.next
```

remove_node(self, node)

Implementation

Now as we're done with the `remove_node` method, let's go ahead and look at the solution for "Josephus Problem":

```
1  def josephus_circle(self, step):
2    cur = self.head
3
4    while len(self) > 1:
5      count = 1
6      while count != step:
7        cur = cur.next
8        count += 1
9      print("KILL:" + str(cur.data))
10     self.remove_node(cur)
11     cur = cur.next
```

josephus_circle(self, step)

Explanation

`step` is passed as one of the arguments to the method `josephus_cirle`. On **line 2**, we initialize `cur` to `self.head` and set up a `while` loop on **line 4** that will keep running until the length of the linked list becomes `1`. We set `count` to `1` at the beginning of the iteration on **line 5**. Next, we have another nested `while` loop on **line 6** which will run until `count` is not equal to `step`. In this nested `while` loop, we move from node to node by updating `cur` to `cur.next` on **line** 7, and in each iteration, we increment the `count` by `1`. As soon as `count` becomes equal to `step`, the `while` loop breaks, and the execution jumps to **line 9**. On **line 9**, we print the node that we land on, so you can visualize the nodes that we will remove. In the next line, we remove the node (`cur`) as the `while` loop ended with that being the current node. On **line 11**, we update `cur` to `cur.next` to repeat the entire process until we are left with one node which will break the `while` loop on **line 4**.

Yes, the solution is as simple as that. You can play around with the entire code in the widget below, which contains all the code that we have implemented for this chapter so far.

```
121                    prev = cur
122                    cur = cur.next
```

```
                    if cur == node:
                        prev.next = cur.next
                        cur = cur.next

    def josephus_circle(self, step):
        cur = self.head

        while len(self) > 1:
            count = 1
            while count != step:
                cur = cur.next
                count += 1
            print("KILL:" + str(cur.data))
            self.remove_node(cur)
            cur = cur.next


cllist = CircularLinkedList()
cllist.append(1)
cllist.append(2)
cllist.append(3)
cllist.append(4)


cllist.josephus_circle(2)
cllist.print_list()
```

By now, you will hopefully be familiar with the circular linked lists and challenges related to it. We have an exercise prepared for you in the next lesson!

Back

Split Linked List into Two Halves

Next

Exercise: Is Circular Linked List

Mark as Completed

Ask a Question

Report an

Issue (https://discuss.educative.io/tag/josephus-problem__circular-linked-lists__data-structures-and-algorithms-in-python)

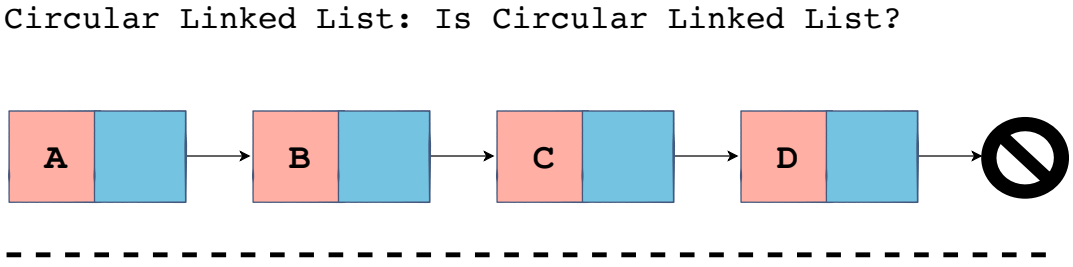
Exercise: Is Circular Linked List

Challenge yourself with an exercise in which you'll have to determine whether a given linked list is circular or not.

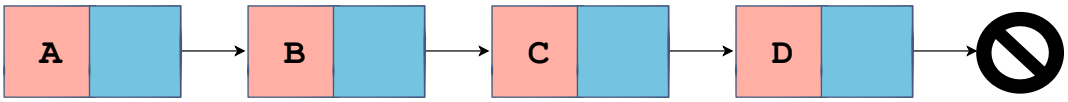
We'll cover the following

Problem

In this exercise, you are required to determine whether a given linked list is a circular linked list or not. Your solution should return `True` or `False` to indicate if the linked list that was passed in to the method is circular or not.



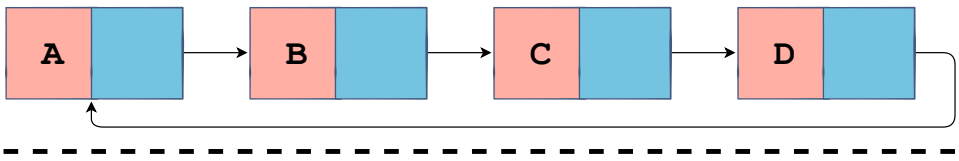
Circular Linked List: Is Circular Linked List?



False

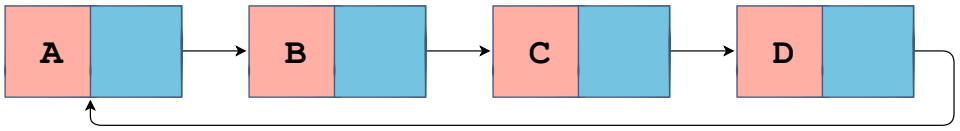
2 of 4

Circular Linked List: Is Circular Linked List?



3 of 4

Circular Linked List: Is Circular Linked List?



True

4 of 4

Coding Time!

In the code below, the `is_circular_linked_list` is a class method of the `CircularLinkedList` class. You cannot see the rest of the code as it is hidden. As `is_circular_linked_list` 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 required to return either `True` or `False` based on whether `input_list` is circular or not. In the test cases, the ellipses (`...`) are used to depict circular linked lists which means that they represent that the next node in the linked list is the head node.

Good luck!

```
    def is_circular_linked_list(self, input_list):
        cur = input_list.head
        is_cllist = False
```

```
4          while cur:
5              cur = cur.next
6              if not cur:
7                  break
8              if cur.next == input_list.head:
9                  is_cllist = True
10                 break
11         return is_cllist
```

Show Results | Show Console

0.18s

5 of 5 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	is_circular_linked_list(1->2->5->13->NU	False	False	Succeeded
✓	is_circular_linked_list(3->7->4->18->11	True	True	Succeeded
✓	is_circular_linked_list(25->...)	True	True	Succeeded
✓	is_circular_linked_list(1000->999->...	True	True	Succeeded
✓	is_circular_linked_list(7->NULL)	False	False	Succeeded

Back

Next

Josephus Problem

Solution Review: Is Circular Linked List

Completed

Report an Issue

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

Solution Review: Is Circular Linked List

This lesson contains the solution review for the challenge of determining whether a given linked list is circular or not.

We'll cover the following

In this lesson, we investigate how to determine whether a given linked list is either a singly linked list or a circular linked list.

Implementation

Have a look at the coding solution below:

```
1  def is_circular_linked_list(self, input_list):
2    cur = input_list.head
3    while cur.next:
4      cur = cur.next
5      if cur.next == input_list.head:
6        return True
7    return False
```

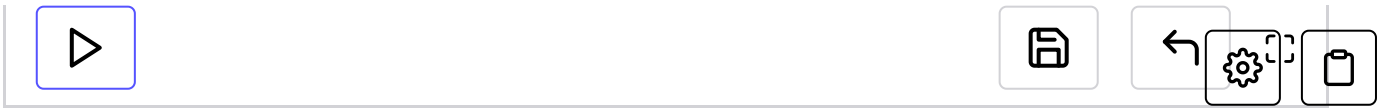
Explanation

Let's discuss the class method `is_circular_linked_list`. On **line 2**, we initialize `cur` to `input_list.head` so that we are able to traverse `input_list` that has been passed to `is_circular_linked_list`. On **line 3**

we set up a `while` loop which runs until `cur.next` becomes `None`. If `cur.next` becomes `None`, it implies that `input_list` must not be a circular linked list and therefore, if the `while` loop terminates, we return `False` on **line** 7. On the other hand, we update `cur` to `cur.next` in the `while` loop on **line 4**, and if we reach a node while traversing such that the next node is the head node, then this confirms that `input_list` is indeed a `circular_linked_list`. Hence, if the condition on **line 5** turns out to be `True`, we return `True` from the method on **line 6**.

As you can see, the `is_circular_linked_list` method was pretty straightforward. Below we test it on a singly linked list and on a circular linked list:

```
157                 cur = cur.next
158                 count += 1
159             print("REMOVED: " + str(cur.data))
160             self.remove_node(cur)
161             cur = cur.next
162 
163     def is_circular_linked_list(self, input_list):
164         cur = input_list.head
165         while cur.next:
166             cur = cur.next
167             if cur.next == input_list.head:
168                 return True
169         return False
170 
171 cllist = CircularLinkedList()
172 cllist.append(1)
173 cllist.append(2)
174 cllist.append(3)
175 cllist.append(4)
176 
177 llist = LinkedList()
178 llist.append(1)
179 llist.append(2)
180 llist.append(3)
181 llist.append(4)
182 
183 print(cllist.is_circular_linked_list(cllist))
184 print(cllist.is_circular_linked_list(llist))
```



That was all we had about circular linked lists in this course. In the next chapter, we'll explore another type of linked list: Doubly Linked Lists.

Hope you are enjoying the lessons and challenges!

Back

Next

Exercise: Is Circular Linked List

Quiz

Mark as Completed

Report an Issue

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