Split Linked List into Two Halves

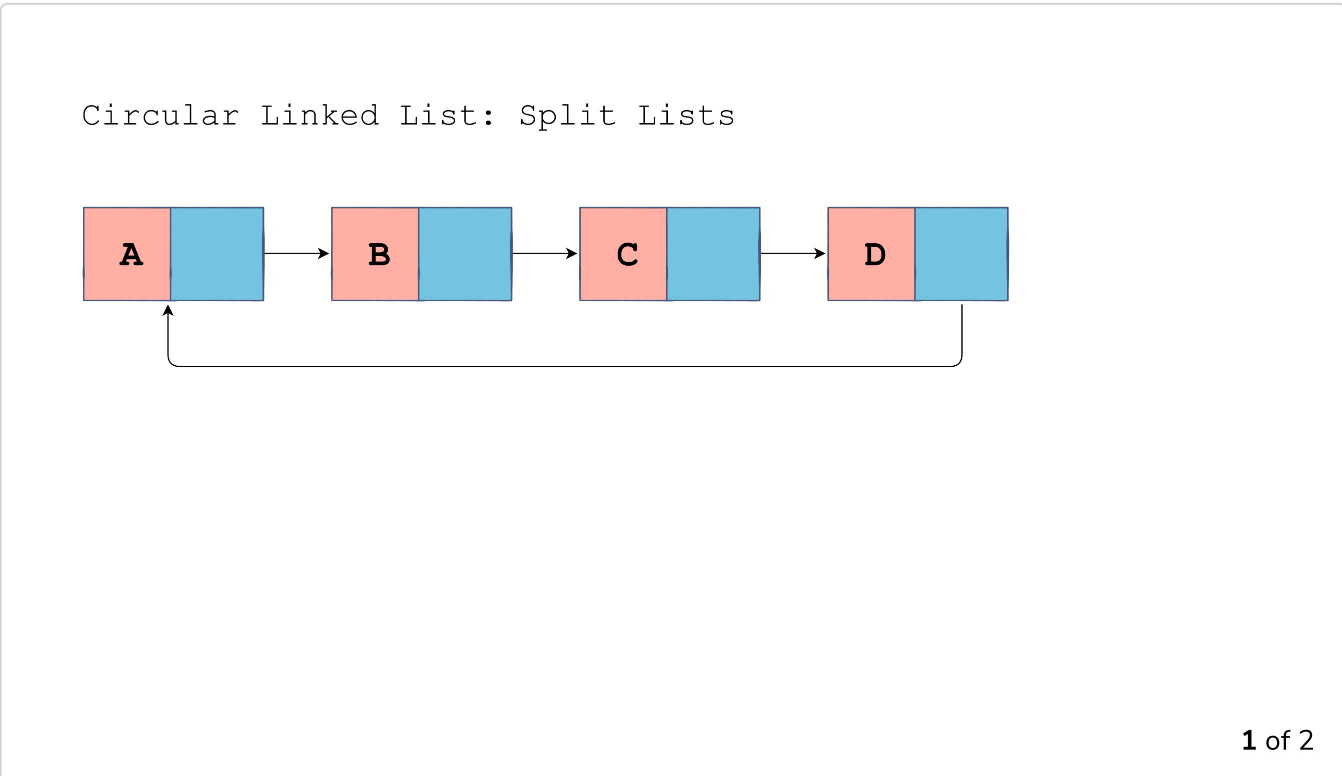
In this lesson, you will learn how to split a circular linked list into two halves in Python.

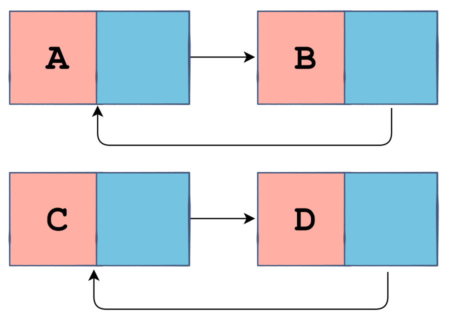
WE'LL COVER THE FOLLOWING

- `__len__()`
- Explanation
- Implementation
- Explanation

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.





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:

```
def __len__(self):
  cur = self.head
  count = 0
  while cur:
    count += 1
    cur = cur.next
    if cur == self.head:
      break
  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

```
def split_list(self):
    size = len(self)

    if size == 0:
        return None
    if size == 1:
        return self.head

    mid = size//2
    count = 0

    prev = None
    cur = self.head

    while cur and count < mid:
        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()
```

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.

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


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

    def prepend(self, data):
        new_node = Node(data)
        cur = self.head
        new_node.next = self.head

        if not self.head:
            new_node.next = new_node
        else:
            while cur.next != self.head:
                cur = cur.next
```

```
            cur.next = new_node
        self.head = new_node


    def append(self, data):
        if not self.head:
            self.head = Node(data)
            self.head.next = self.head
        else:
            new_node = Node(data)
            cur = self.head
            while cur.next != self.head:
                cur = cur.next
            cur.next = new_node
            new_node.next = self.head

    def print_list(self):
        cur = self.head

        while cur:
            print(cur.data)
            cur = cur.next
            if cur == self.head:
                break

    def __len__(self):
        cur = self.head
        count = 0
        while cur:
            count += 1
            cur = cur.next
            if cur == self.head:
                break
        return count

    def split_list(self):
        size = len(self)

        if size == 0:
            return None
        if size == 1:
            return self.head

        mid = size//2
        count = 0

        prev = None
        cur = self.head

        while cur and count < mid:
            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()
cllist.append("A")
cllist.append("B")
cllist.append("C")
cllist.append("D")
cllist.append("E")
cllist.append("F")

cllist.split_list()
```

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