Introduction and Insertion

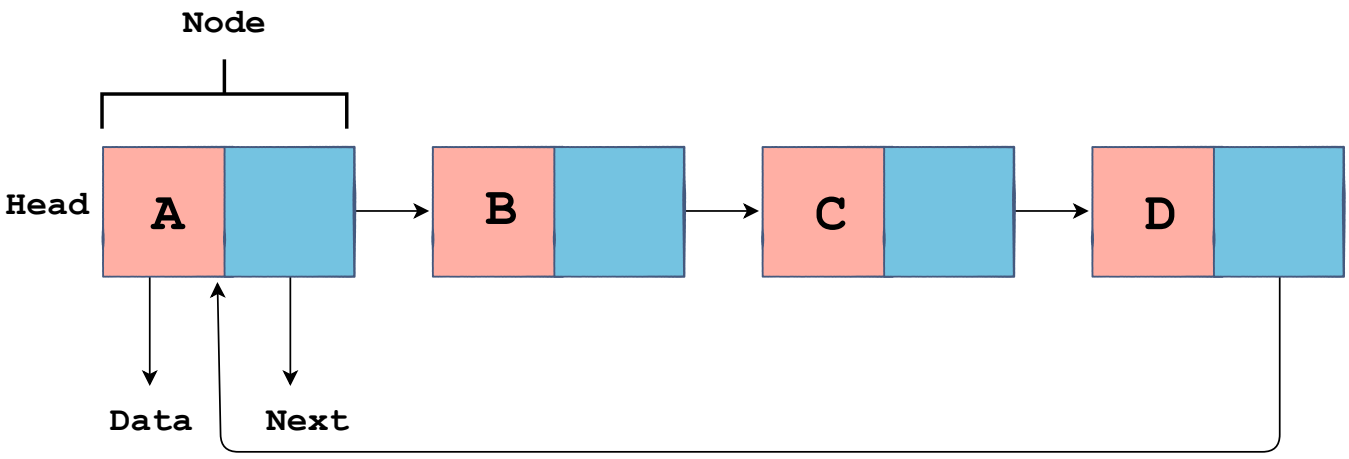
In this lesson, you will be introduced to circular linked lists and you will learn how to insert elements into a circular linked list.

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:

```
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
```

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:

```
def print_list(self):
  cur = self.head

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

```
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
```

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

last node of the node before the head node, after which we will set the

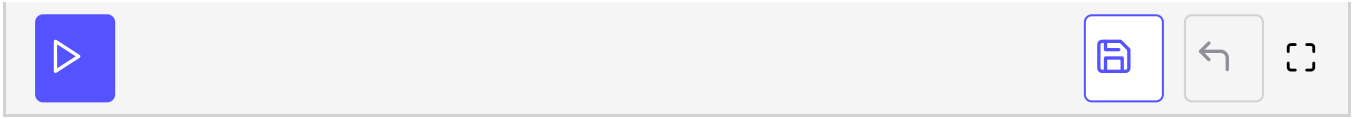
`cur.next` to `new_node` on **line 11** to implement the circular chain of our linked list.

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:

```
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
```

```
cllist = CircularLinkedList()

cllist.append("C")
cllist.append("D")
cllist.prepend("B")
cllist.prepend("A")
cllist.print_list()
```



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!