Insertion

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

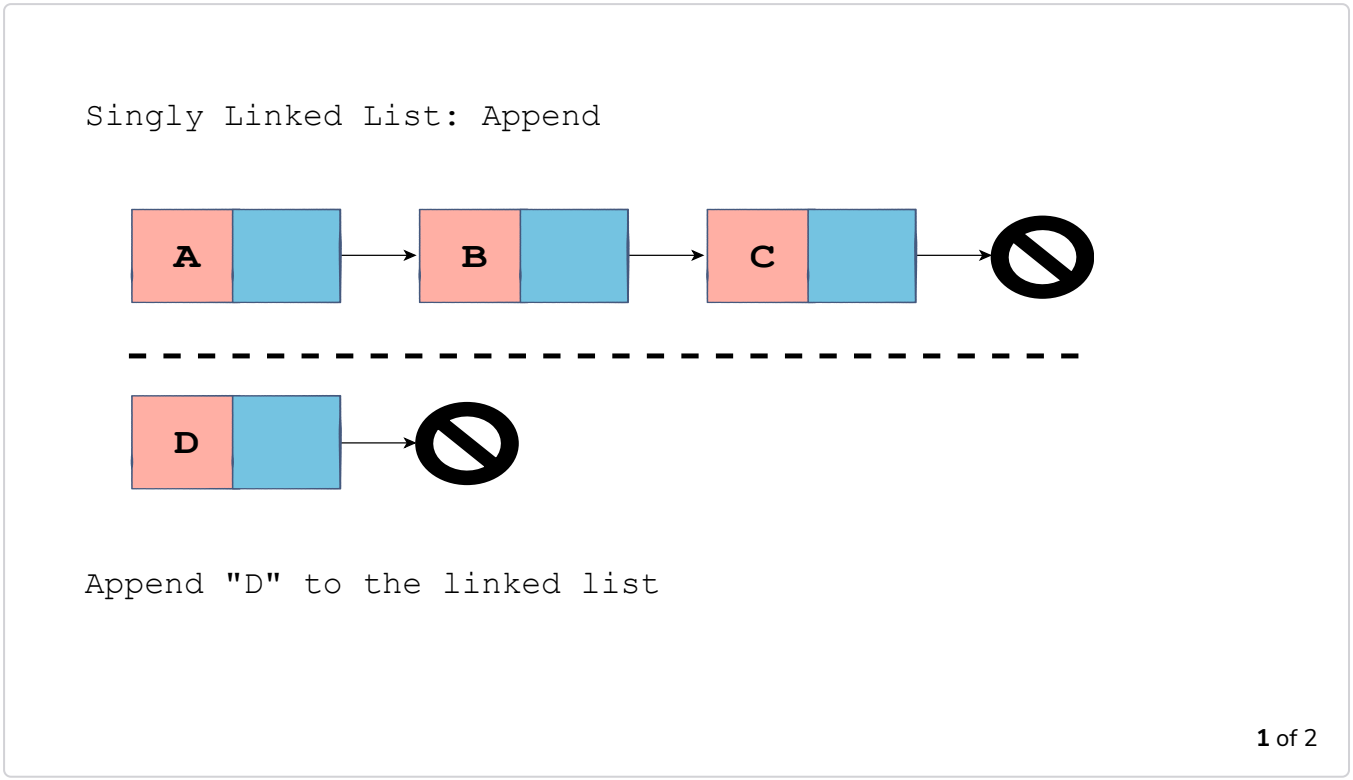
We'll cover the following

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

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

Append

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



Now let's move on to writing some code.

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

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

  def append(self, data):
    new_node = Node(data)
```

class Node and class LinkedList

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

Empty Linked List Case

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

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

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

  def append(self, data):
    new_node = Node(data)
    if self.head is None:
      self.head = new_node
      return
```

class Node and class LinkedList

In the above code, we check if the linked list is empty by checking the head of the linked list. If the `self.head` is `None` on **line 12**, it implies that it's an empty linked list and there's nothing there. The head pointer doesn't point to anything at all, and therefore there is no node in the linked list. If there is no node in the linked list, we set the head pointer to the `new_node` that we created on **line 13**. In the next line, we simply `return`. The case of an empty linked list is relatively easy to handle.

Non-empty Linked List Case

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

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

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

  def append(self, data):
    new_node = Node(data)
    if self.head is None:
      self.head = new_node
      return
    last_node = self.head
    while last_node.next:
      last_node = last_node.next
    last_node.next = new_node
```

class Node and class LinkedList

On **line 15**, we define `last_node` which is initially equal to the head. This implies we're at the start of the linked list. We have named the variable we defined on **line 15** `last_node` because that's what it will eventually point to. It will start at the beginning of the linked list and move through the linked list as long as the `last_node.next` doesn't point to `None`. We keep moving from node to node on **line 17** until we get to the `last_node` where `last_node.next` will point to `None` and will terminate the while loop on **line 16**. After the `while` loop concludes, `last_node` points to the last node. On **line 18**, we input our `new_node` into the linked list by setting the `next` of `last_node` to `new_node` which has its own `next` pointing to `None`.

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

`print_list()`

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

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

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

  def print_list(self):
    cur_node = self.head
    while cur_node:
      print(cur_node.data)
      cur_node = cur_node.next

  def append(self, data):
    new_node = Node(data)
    if self.head is None:
      self.head = new_node
      return
```

```
    last_node = self.head
    while last_node.next:
      last_node = last_node.next

    last_node.next = new_node

llist = LinkedList()
llist.append("A")
llist.append("B")
llist.append("C")
llist.append("D")


llist.print_list()
```

class Node and class LinkedList

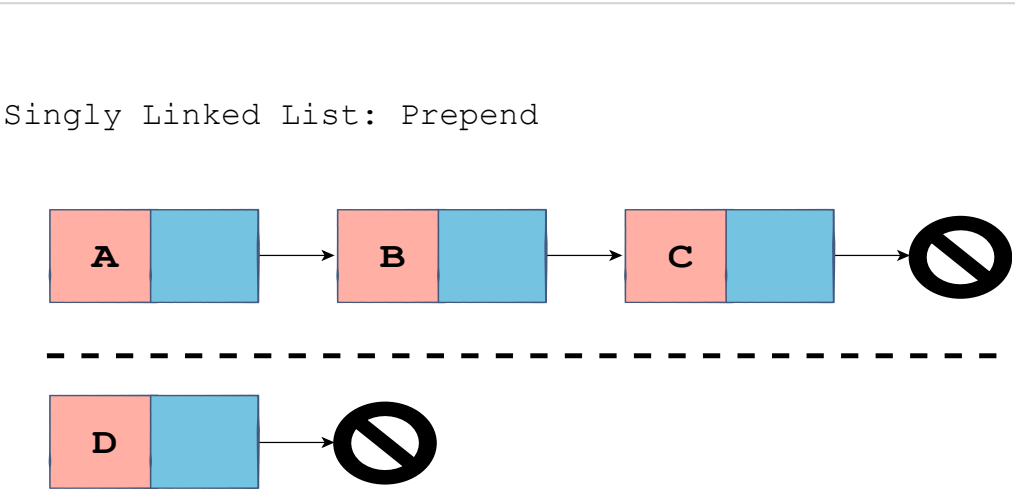
There you go! We initialize `cur_node` equal to the head of the linked list. Then we use a `while` loop which keeps running and printing the data if `cur_node` is not equal to `None`.

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

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

Prepend

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



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

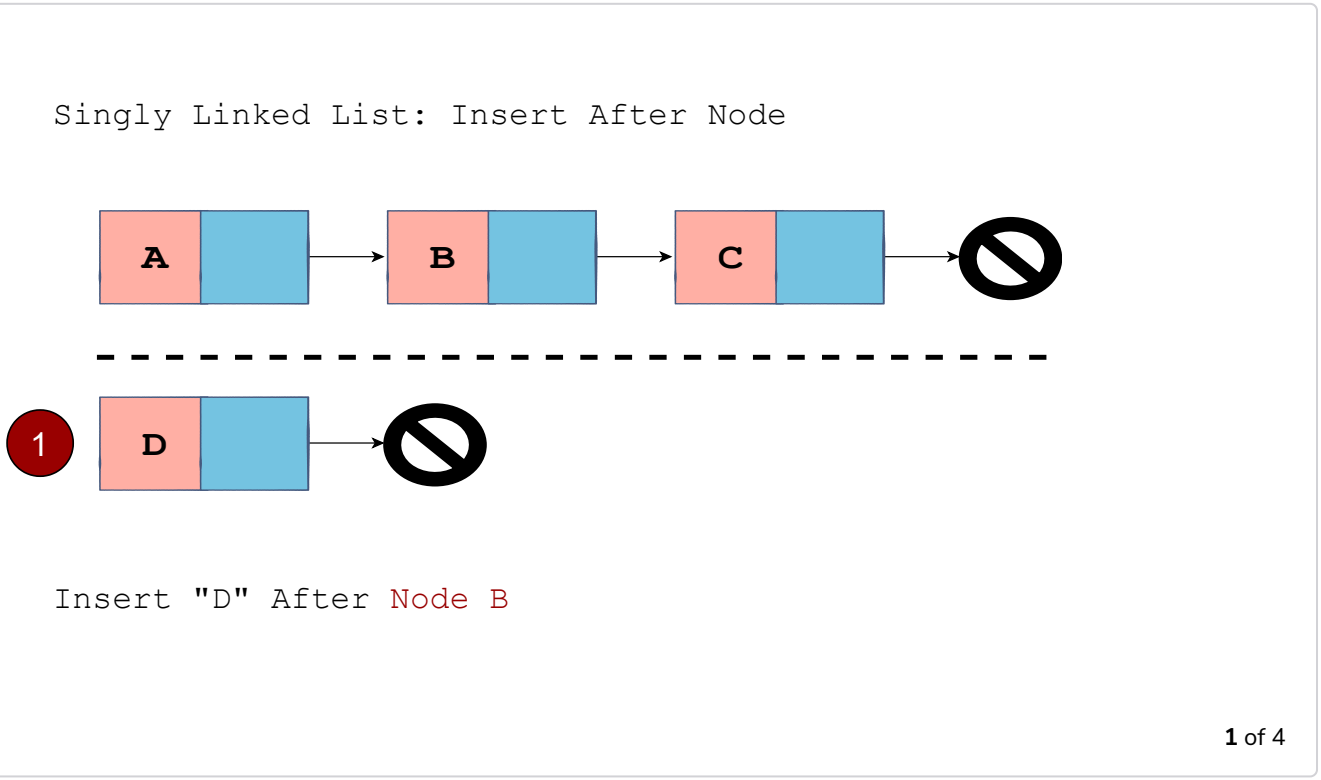
Let’s go ahead and write this code after which we’ll walk it through step by step.

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

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

  def print_list(self):
    cur_node = self.head
    while cur_node:
      print(cur_node.data)
      cur_node = cur_node.next

  def append(self, data):
    new_node = Node(data)
    if self.head is None:
      self.head = new_node
      return
    last_node = self.head
    while last_node.next:
      last_node = last_node.next
    last_node.next = new_node

  def prepend(self, data):
    new_node = Node(data)

    new_node.next = self.head
    self.head = new_node


llist = LinkedList()
llist.append("A")
llist.append("B")
llist.append("C")

llist.prepend("D")

llist.print_list()
```

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

Insert After Node

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



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

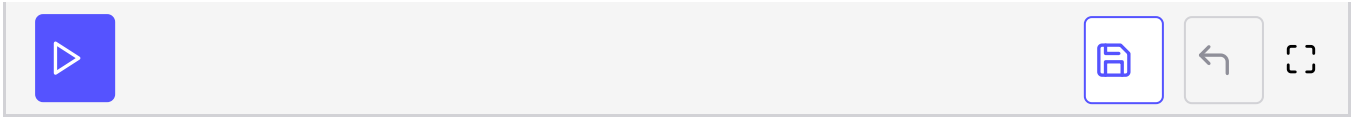
First of all, we will create a new node based on the data `D`. That is *step 1* as depicted in the slides. Next, we need to check if the node to be inserted after is in the linked list or not. If it's not in the linked list, we'll return; otherwise, we

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

Now let's go ahead and code it!

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

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

  def print_list(self):
    cur_node = self.head
    while cur_node:
      print(cur_node.data)
      cur_node = cur_node.next

  def append(self, data):
    new_node = Node(data)
    if self.head is None:
      self.head = new_node
      return
    last_node = self.head
    while last_node.next:
      last_node = last_node.next
    last_node.next = new_node

  def prepend(self, data):
    new_node = Node(data)

    new_node.next = self.head
    self.head = new_node

  def insert_after_node(self, prev_node, data):
    if not prev_node:
      print("Previous node does not exist.")
      return
    new_node = Node(data)

    new_node.next = prev_node.next
    prev_node.next = new_node



llist = LinkedList()
llist.append("A")
llist.append("B")
llist.append("C")


llist.insert_after_node(llist.head.next, "D")
```

```
llist.print_list()
```



class Node and class LinkedList

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

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

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

and return on **line 35**.

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

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

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

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