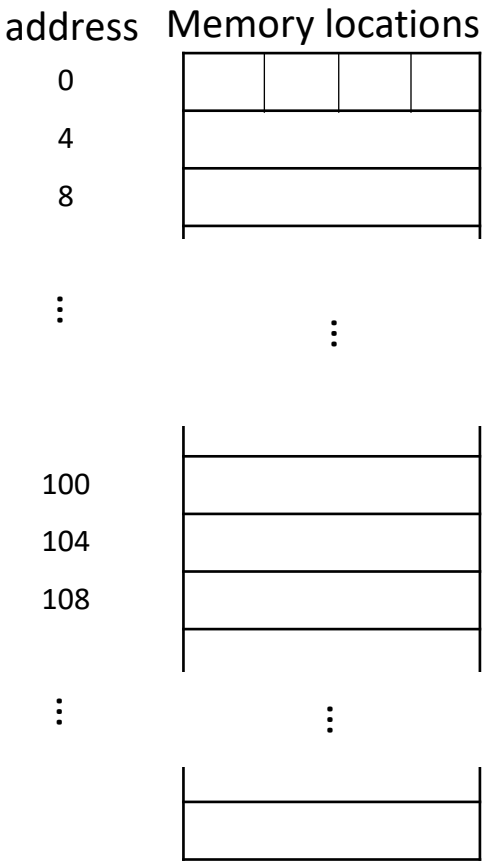
# CSc 120
## Introduction to Computer Programming II

04: Linked Lists

# Recall: Data organization in memory

- Computer memory is organized as a sequence of *locations*
  - each location is identified by its *address* (a number)
  - a location typically consists of 8 bits (a "byte")
  - bytes are often grouped into "words" (32 or 64 bits)

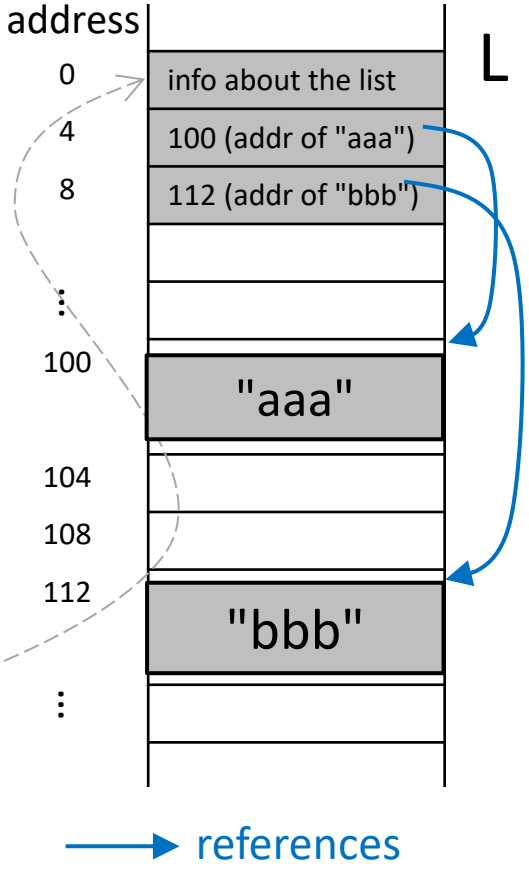
$\Rightarrow$ A location (or word) can only hold a limited amount of data



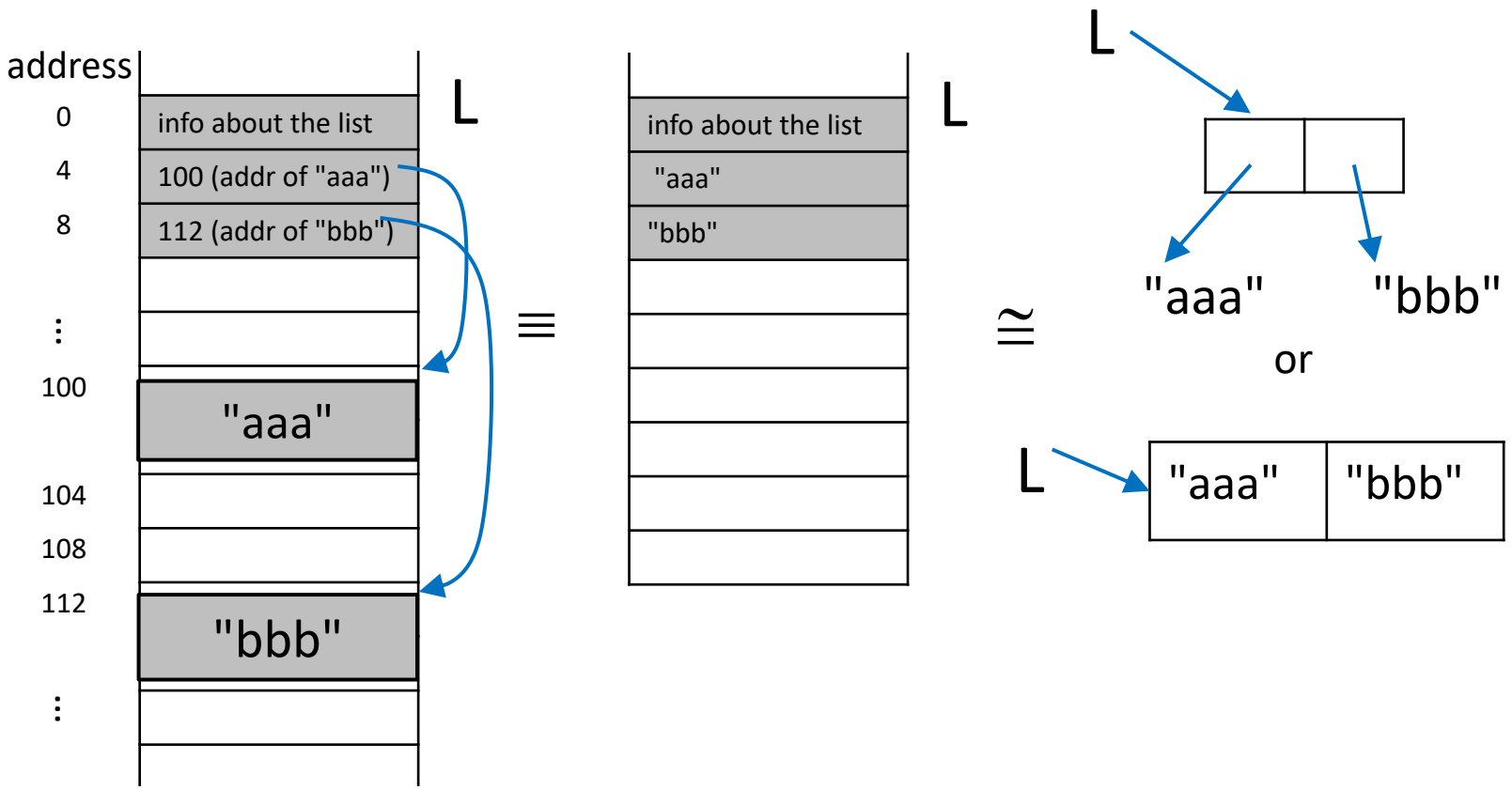
# Data organization in memory

- A memory location can hold only a limited amount of data
- An object typically spans multiple memory locations
- Data are organized as follows:
  - objects are placed where memory is available
  - the object's memory address is used as a *reference* to it

E.g.: for L = ["aaa", "bbb"]



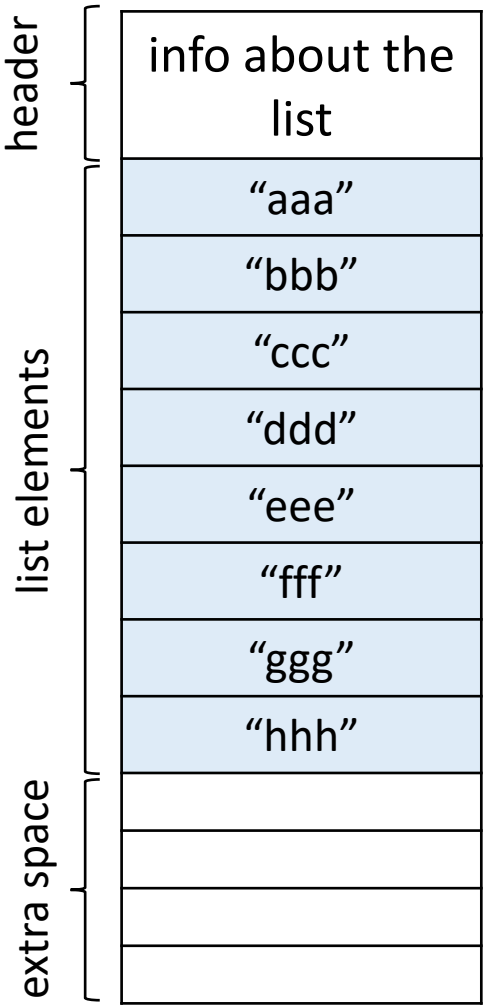
# Data organization in memory




We have been drawing reference diagrams in a way that abstracts away actual address values

# Python lists:

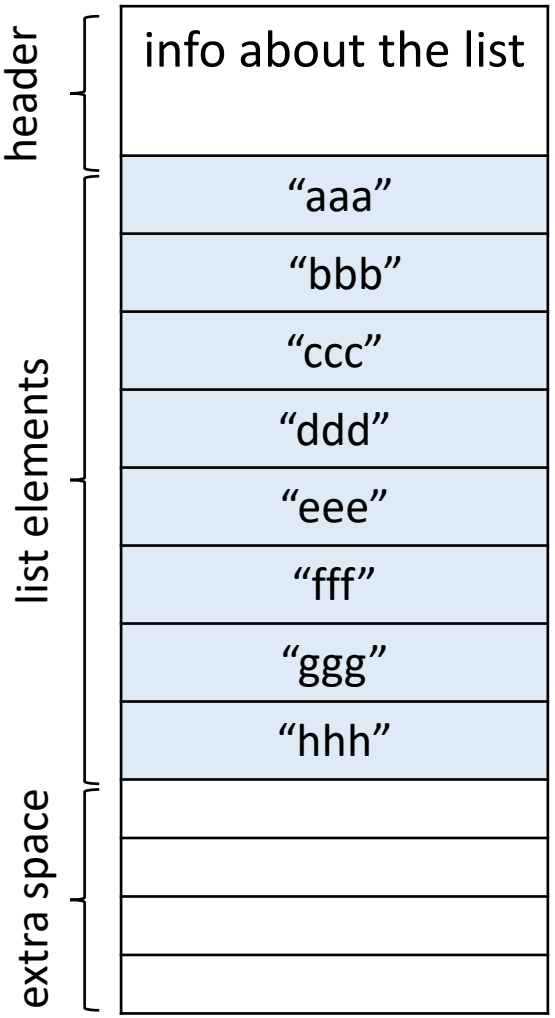
- Key feature: L[i] and L[i+1] are adjacent in memory
- This makes accessing L[i] very efficient
- Insertion and concatenation require moving existing elements
- Insert at the beginning, must move all elements



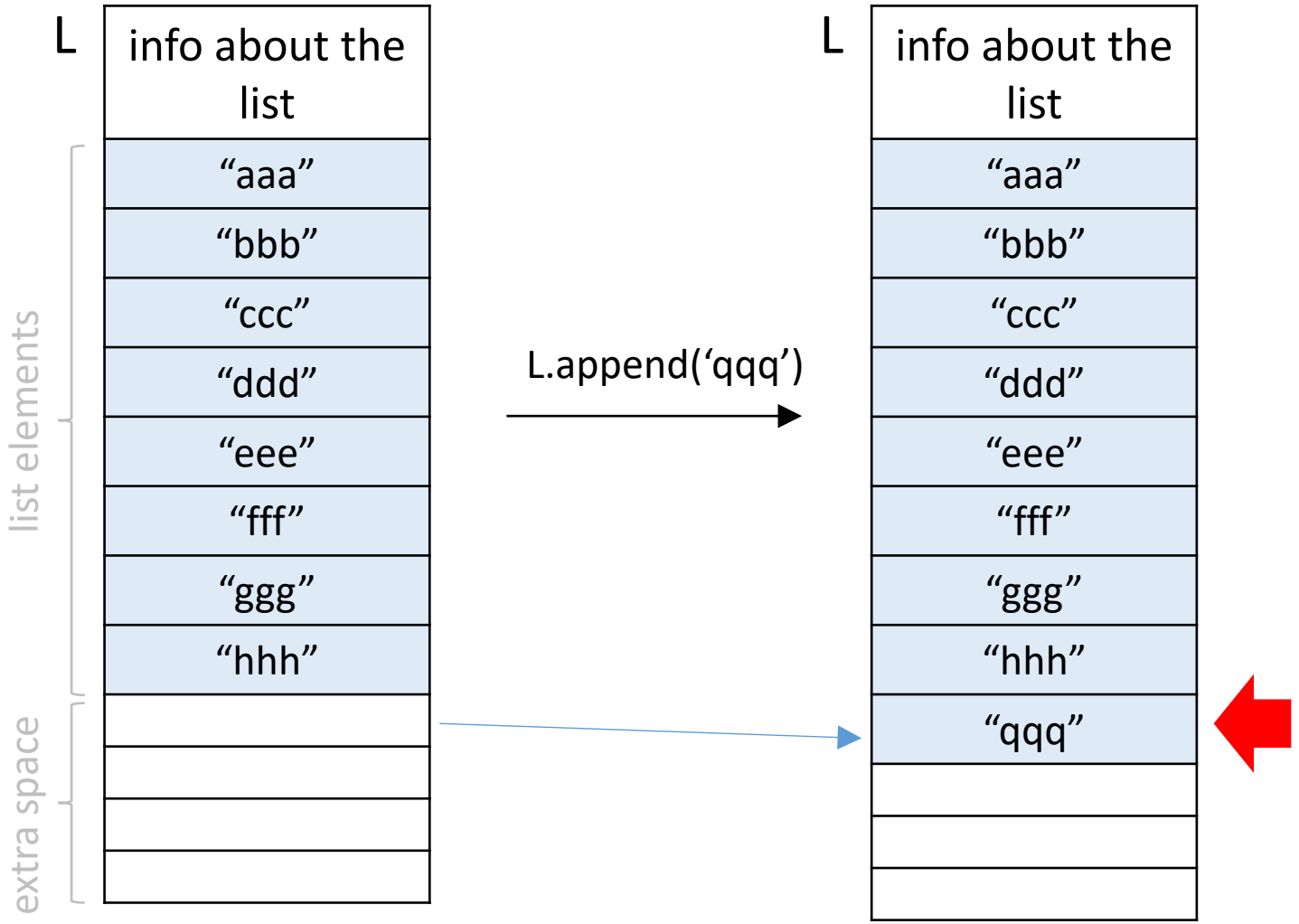
# insert vs. append

# List (array) organization in Python

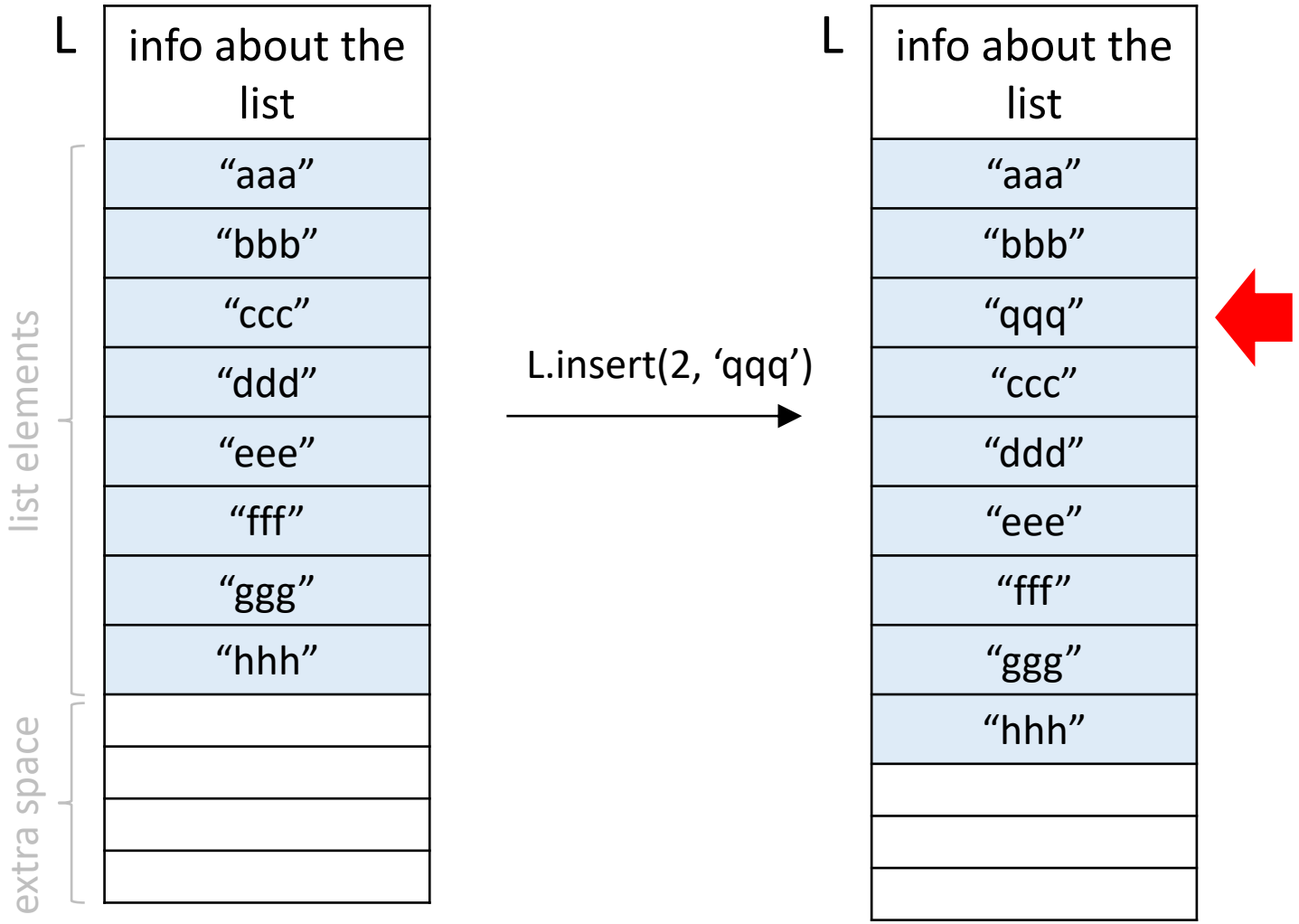
- (References to) the list elements are kept in a contiguous sequence of memory words
  - there is a little extra space at the end to give it some room to grow
- The following operations are fast:
  - len()
    - read off length info from the header
  - accessing the $i^{th}$ element of the list
    - compute its address using the value of i
    - access memory location at that address



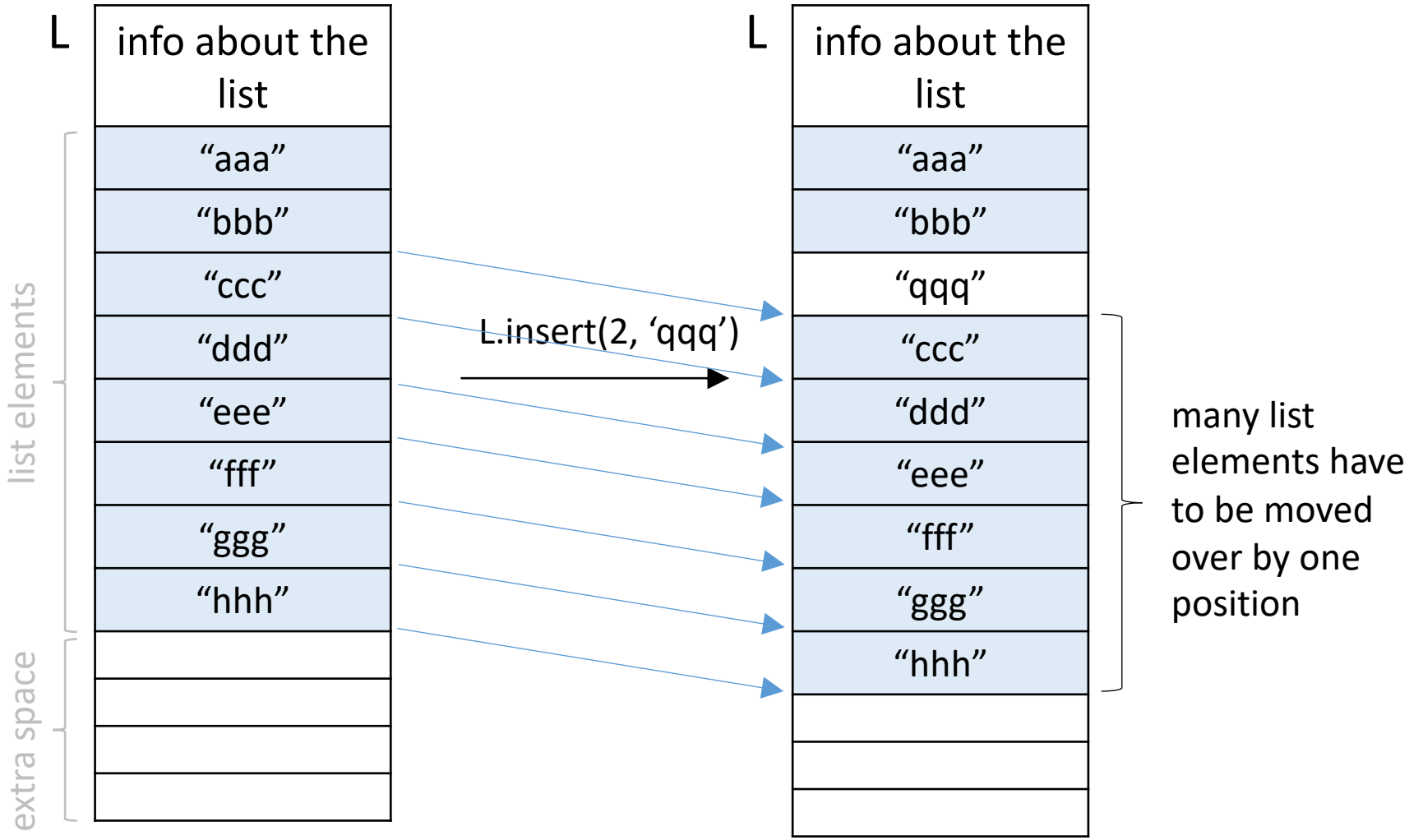
# Appending to a list



# Inserting into a list



# Inserting into a list



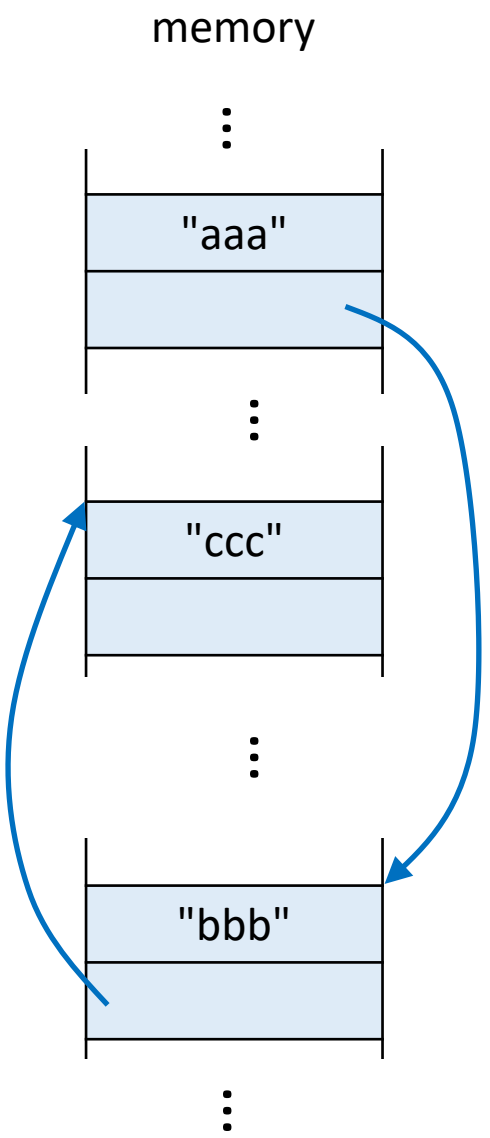
# Python lists: efficiency summary

| Operation | Efficiency |
|---|---|
| len | fast |
| access an element's value | fast |
| append | fast |
| insert, delete | very slow |

Q: Can we create a data structure that is efficient for insertion?

# Linked lists

- To get fast insertion and concatenation, we cannot afford to move later list elements
- We have to relax the requirement that $i^{th}$ element is adjacent to $(i+1)^{st}$ element
  - any element can be anywhere in memory
- Each element has to tell us where to find the next element



# linked lists

# Linked lists

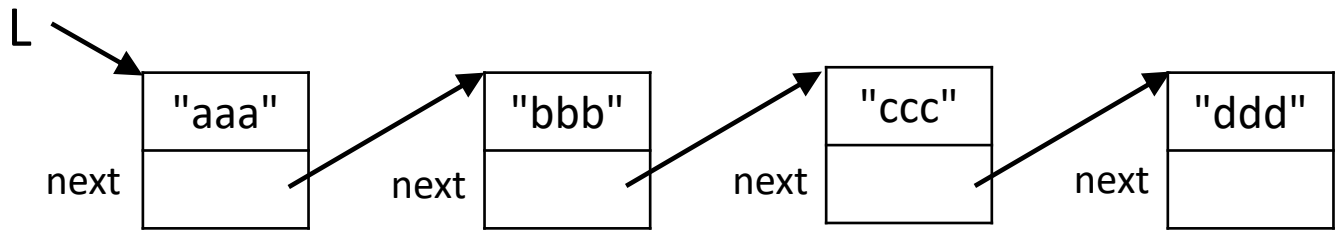
- Linked list:

  A collection of ordered elements where each element has a value and a reference to the next element.

  There is at least one variable that references the beginning of the list.

# Linked lists

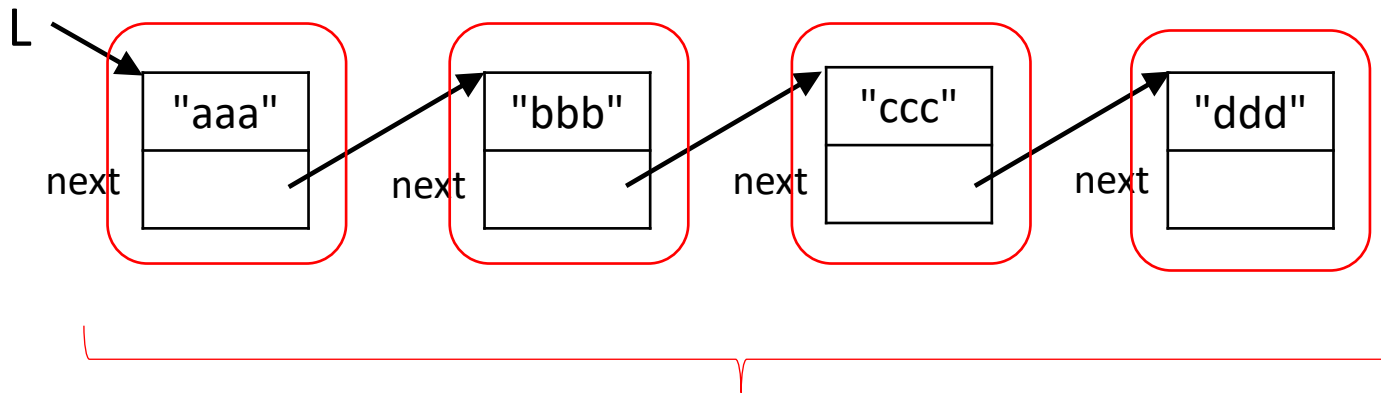
Each element of the list has a reference to the next list element.




This is how we draw a linked list!
Each element has a *value* and a reference to the *next* element.

# Linked lists

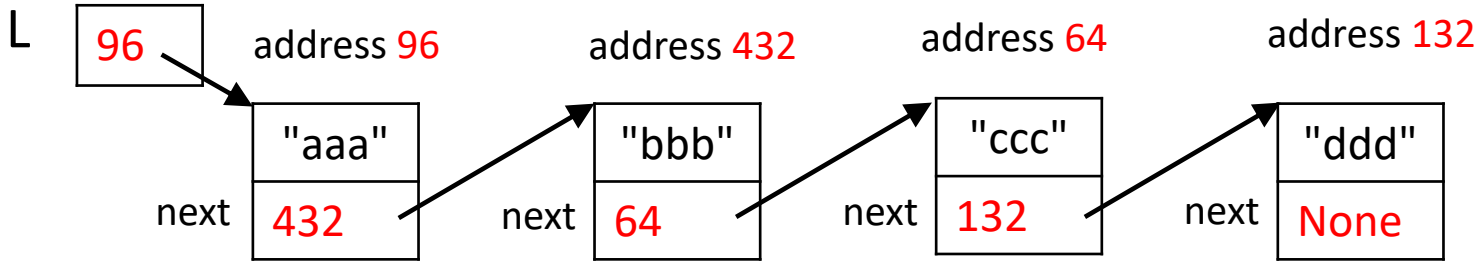
With each element of the list, keep a reference to the next list element



# Linked lists

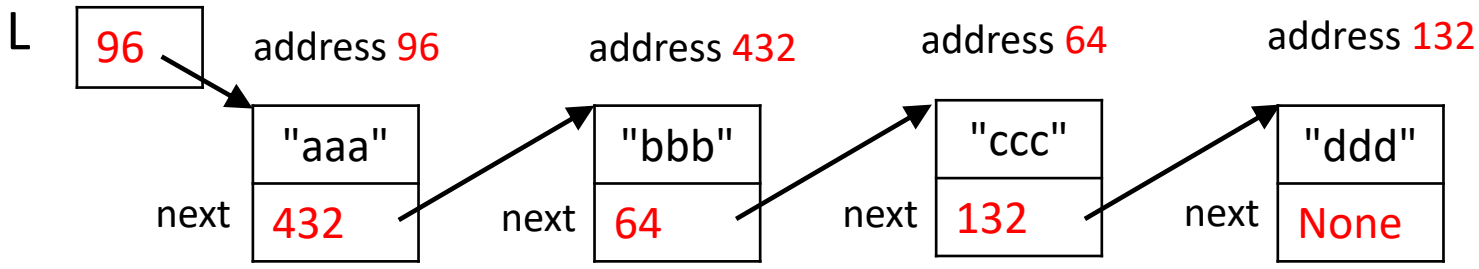
References are addresses in memory.

Here is the diagram with explicit addresses (simplified).



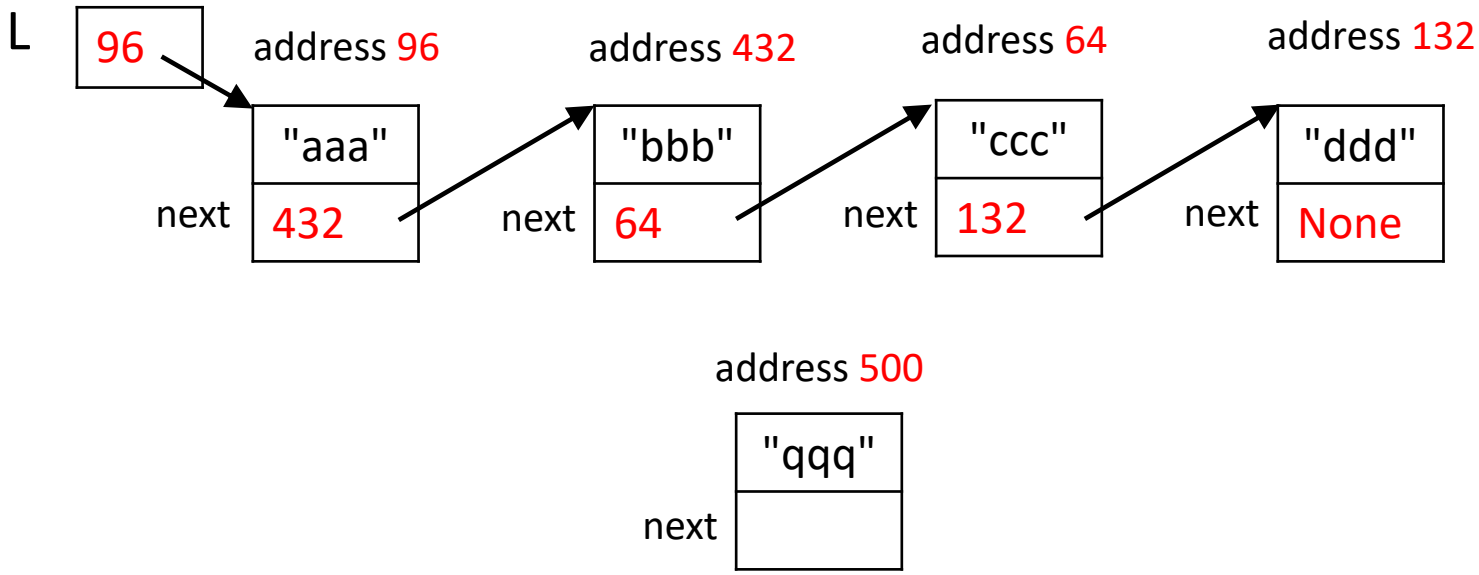
# Insertion

Consider inserting a new node into the linked list



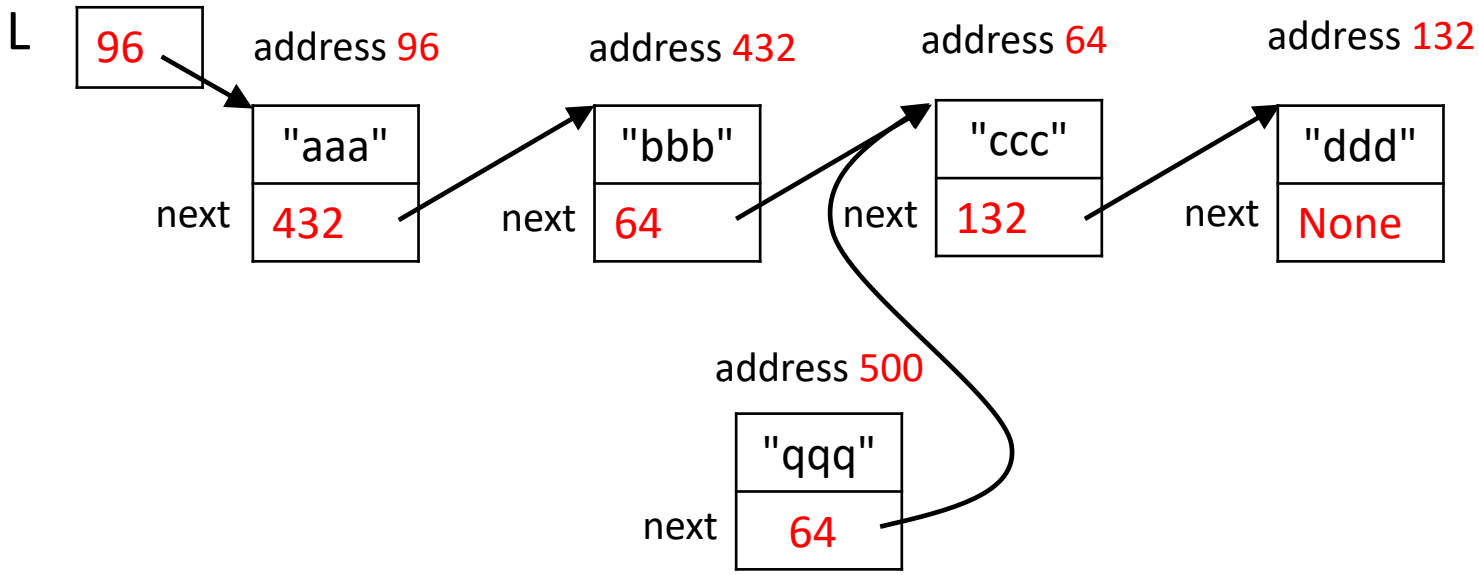
# Insertion

Specifically, add a new node between "bbb" and "ccc". What do we change?



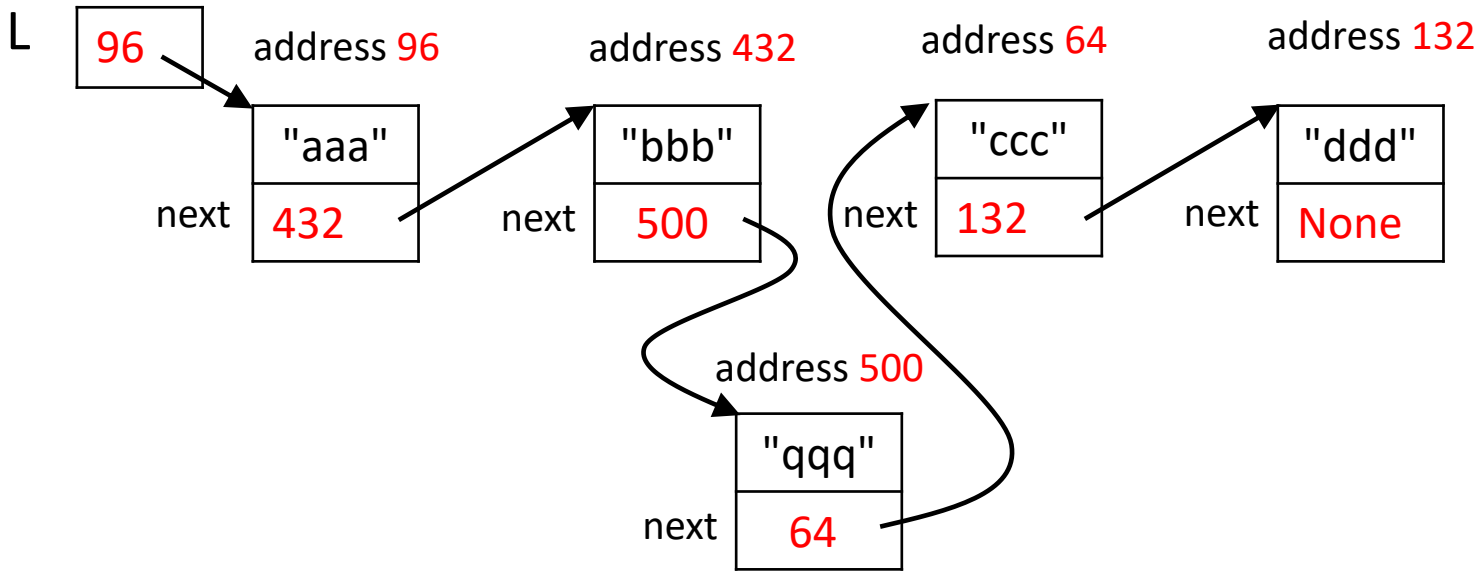
# Insertion

Specifically, add a new node between "bbb" and "ccc". What do we change?



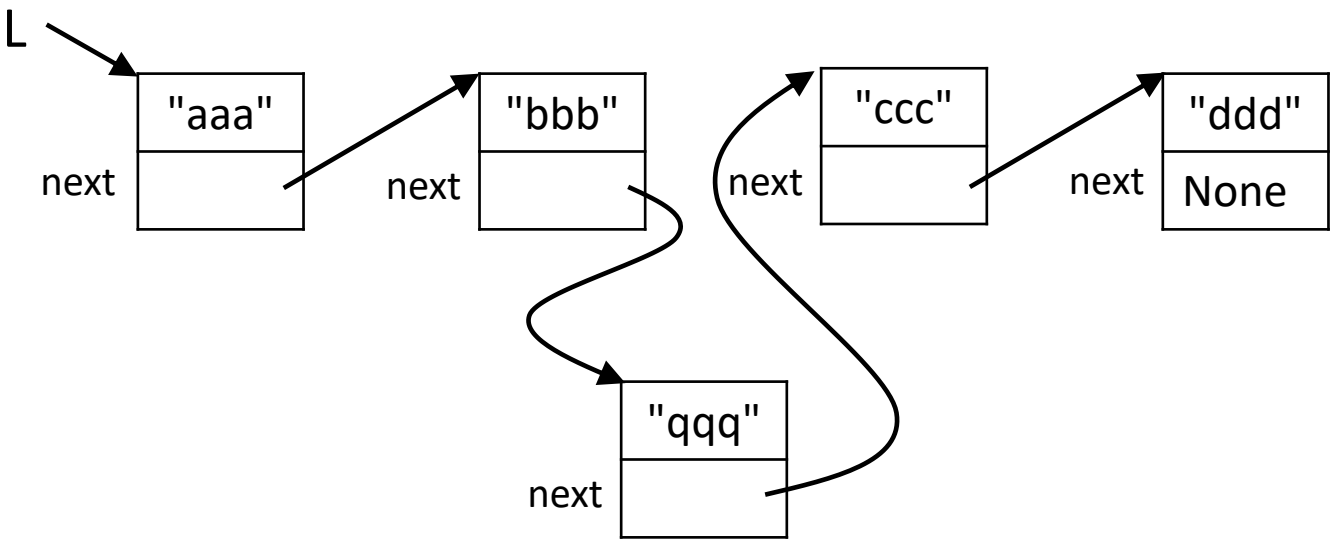
# Insertion

We want to add a new node between "bbb" and "ccc". What do we change?



# Insertion

Set the next references appropriately. Is it faster?*

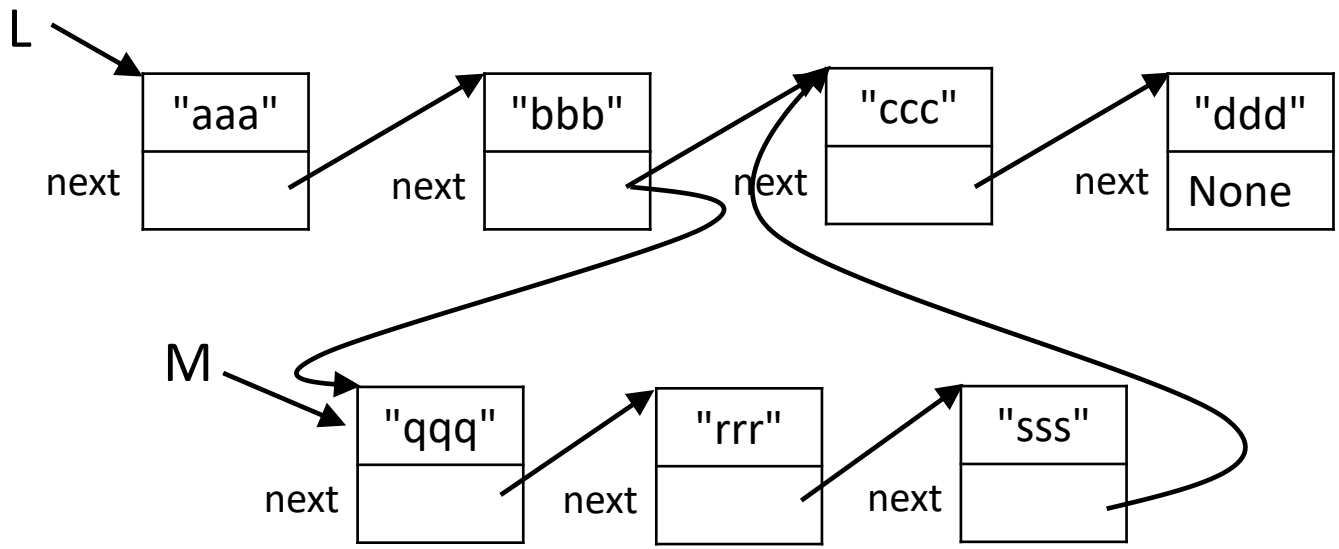



*yes — we didn't have to move anything—just changed two references

# Insertion

fast

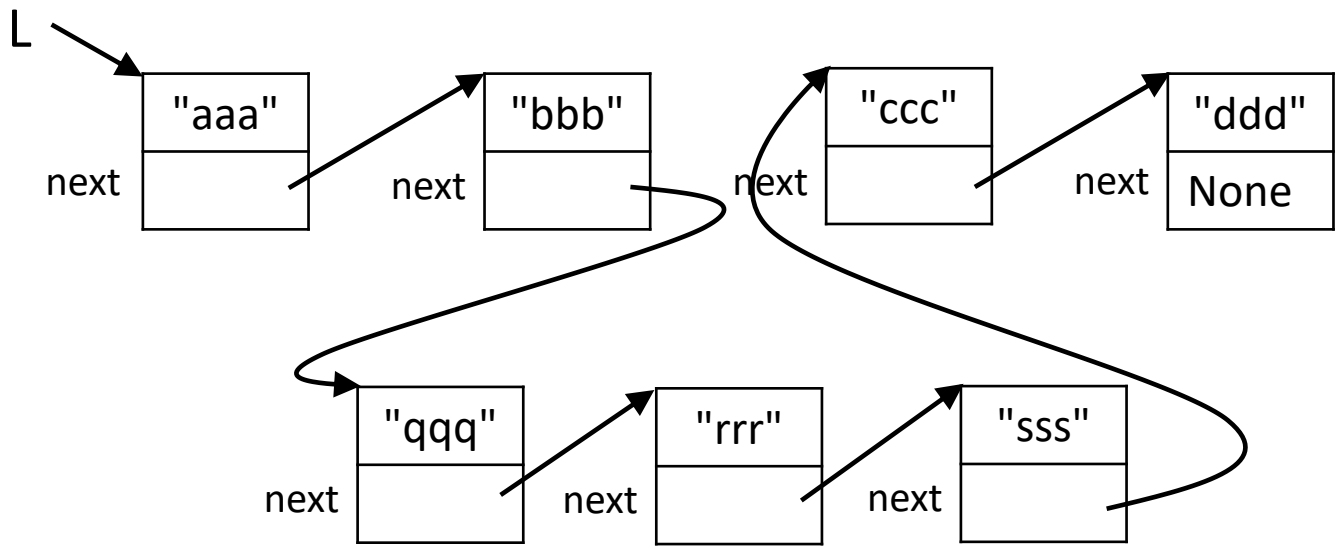
To insert an element (which can be a linked list) into a linked list: set next references appropriately



# Insertion

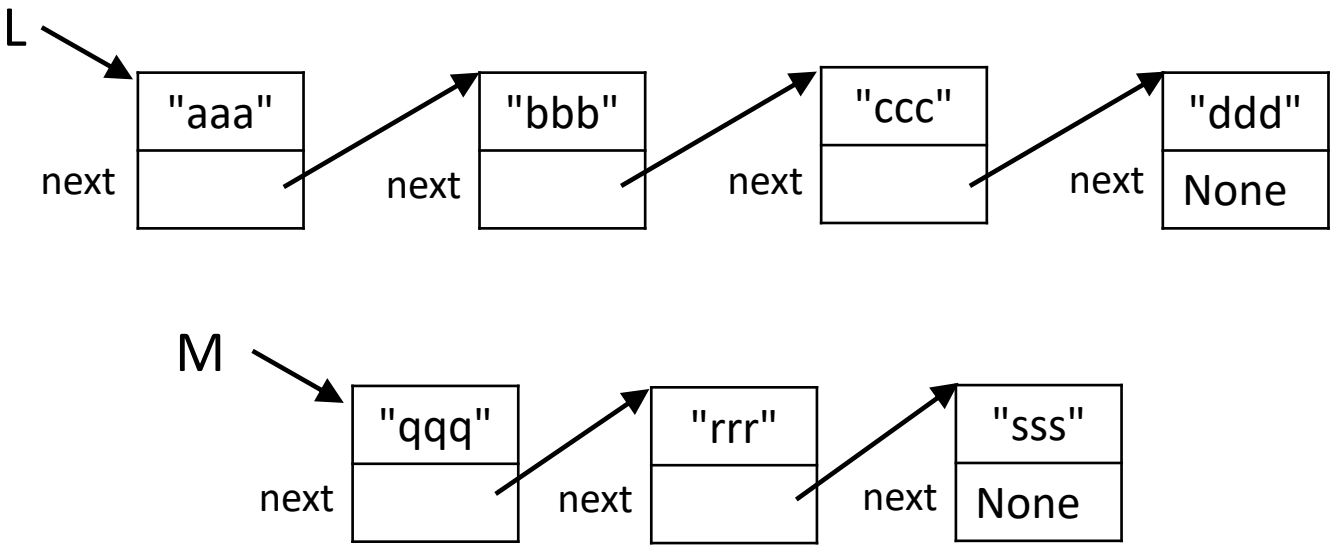
fast

To insert an element into a linked list: set next references appropriately



# Concatenation

To concatenate two linked lists: set next reference of end of first list to refer to beginning of second list

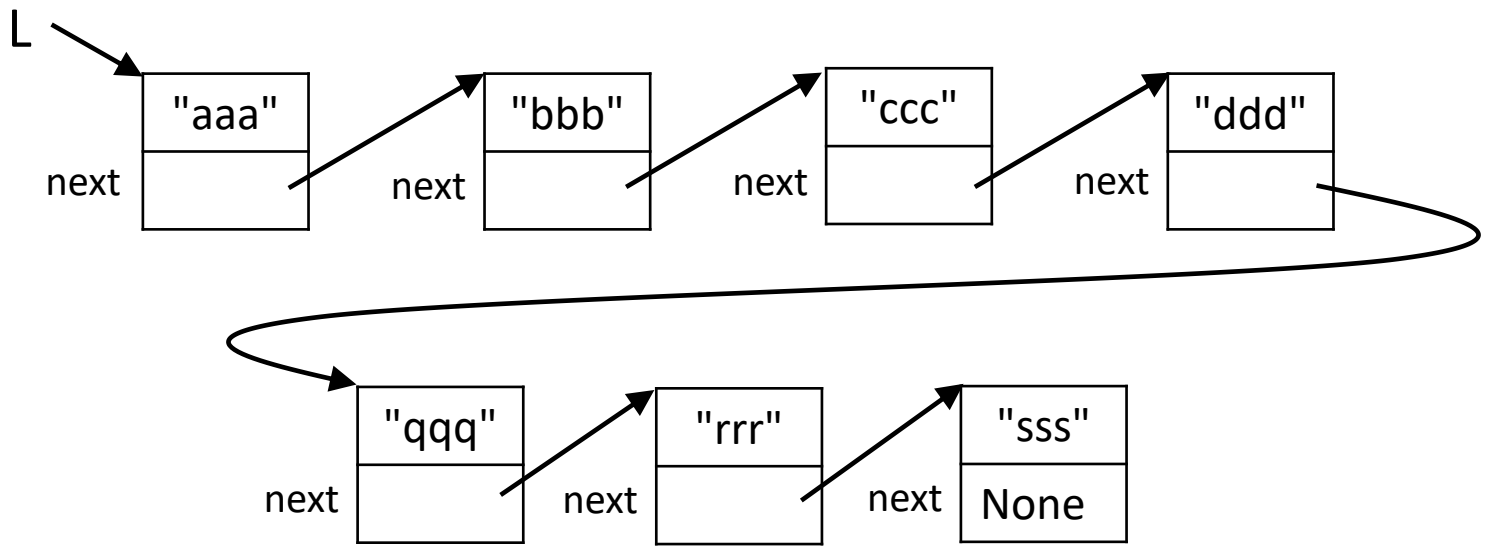



need to get the reference to the end of the first list

# Concatenation

fast*

To concatenate two linked lists: set next reference of end of first list to refer to beginning of second list




* once we have a reference to the end of the first list

# implementation

# Nodes: Implementation

```
class Node:
    def __init__(self, value):
        self._value = value  # reference to the object at that node
        self._next = None  # reference to the next node in the list
```

# Nodes: Implementation

```
class Node:
    def __init__(self, value):
        self._value = value  # reference to the object at that node
        self._next = None  # reference to the next node in the list
```

*Getters*:

```
    def value(self):
        return self._value
```

```
    def next(self):
        return self._next
```

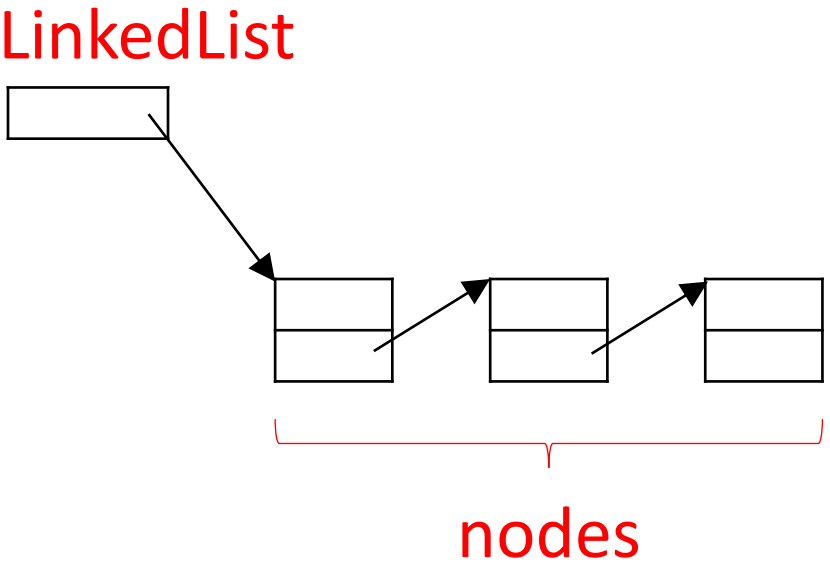
*Setters*:

```
    def set_value(self, value):
        self._value = value
```

```
    def set_next(self, next):
        self._next = next
```

# Linked Lists: Implementation

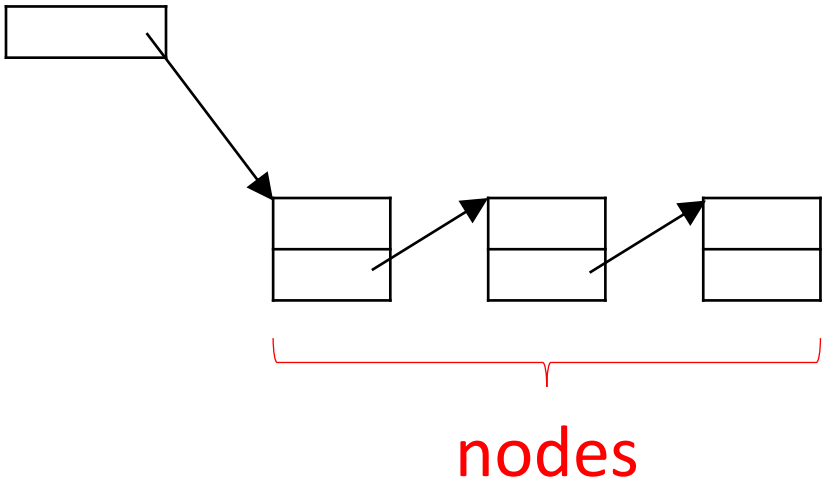
A linked list is just (a reference to) a sequence of nodes



# Linked Lists: Implementation

A linked list is just (a reference to) a sequence of nodes

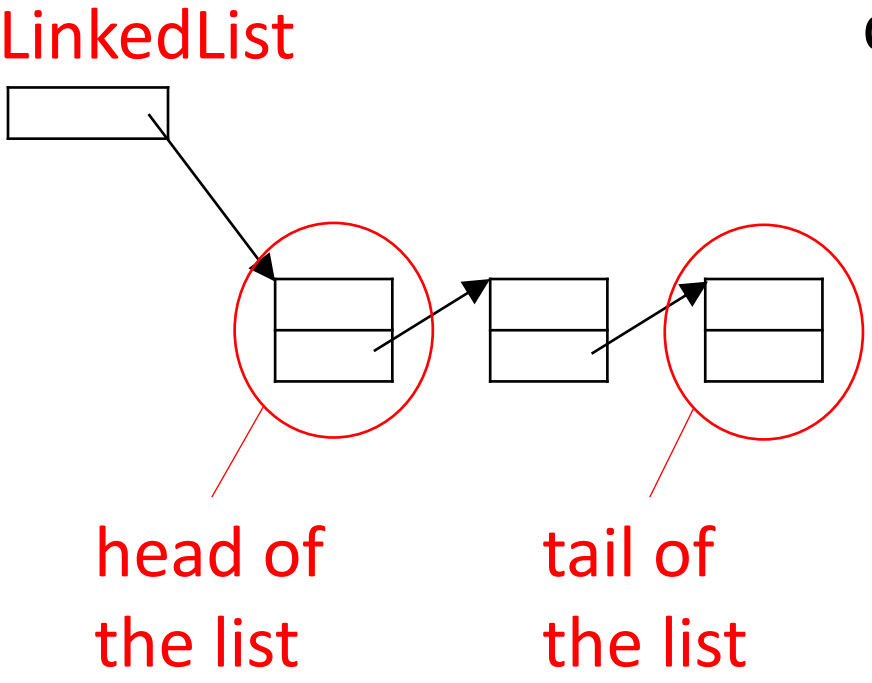
LinkedList




```
class LinkedList:
    def __init__(self):
        self._head = None
```

# Linked Lists: Implementation

A linked list is just (a reference to) a sequence of nodes




```
class LinkedList:
    def __init__(self):
        self._head = None
```

# Linked Lists: Implementation

```
class LinkedList:
    def __init__(self):
        self._head = None
```

# Linked Lists: Implementation

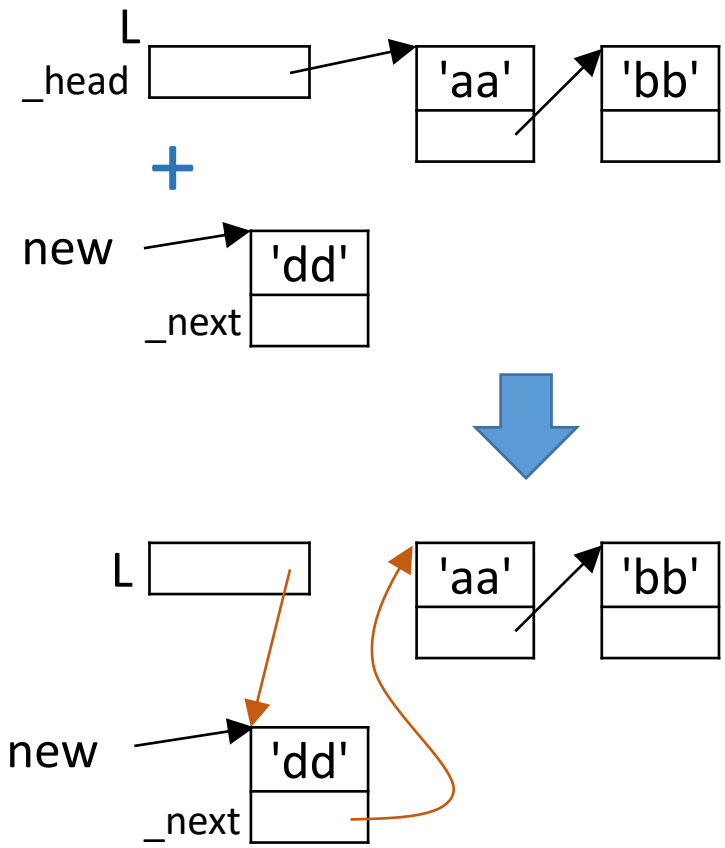
```
class LinkedList:
    def __init__(self):
        self._head = None

    def is_empty(self):
        return self._head == None

    def head(self):
        return self._head
```

# Exercise

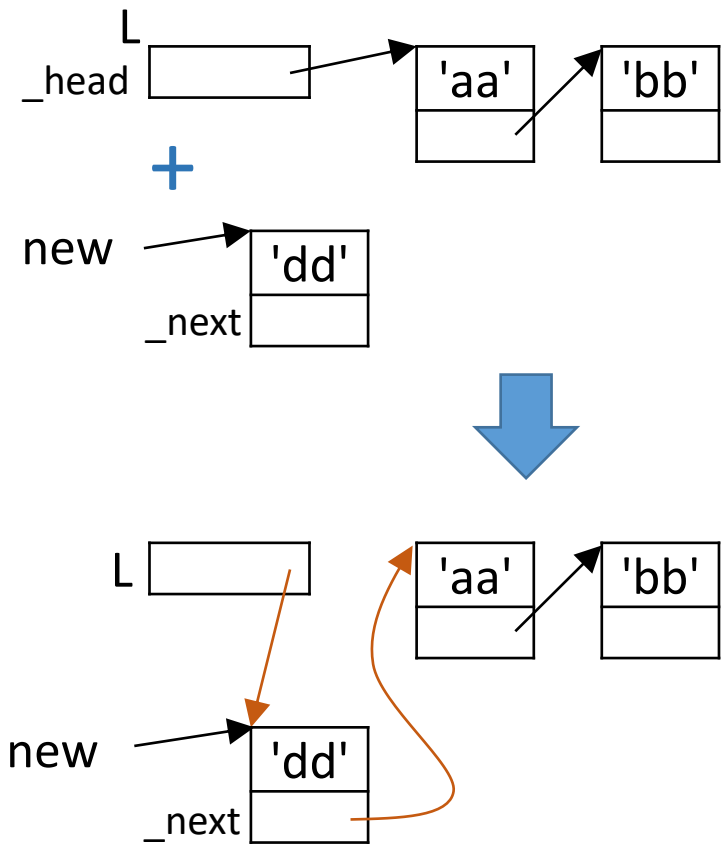
- Do problems 2, 4, and 5 in ICA-12.

# addition
# at the head of the list

# Adding a node at the head

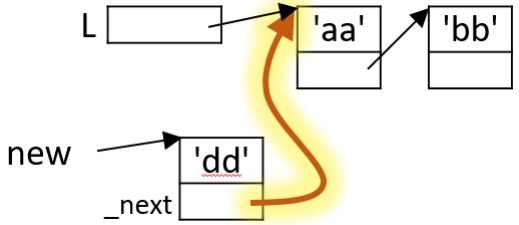


# Adding a node at the head

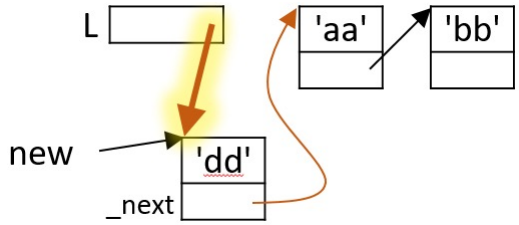



Sequence of operations for an add method:

1. new._next = L._head




2. L._head = new



# Adding a node at the head

```
class LinkedList:
   def __init__(self):
       self._head = None

   # add a node new at the head of the linked list
   def  add(self, new):
        new._next = self._head
        self._head = new
```

# Creating a linked list: Example 1

```
class Node:
    def _ _init_ _(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def _ _init_ _(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
>>> my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```

# Creating a linked list: Example 1

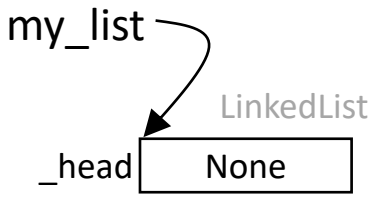
```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
>>> my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```

# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

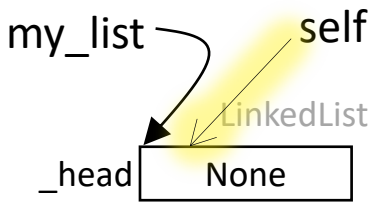
```
>>> my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```

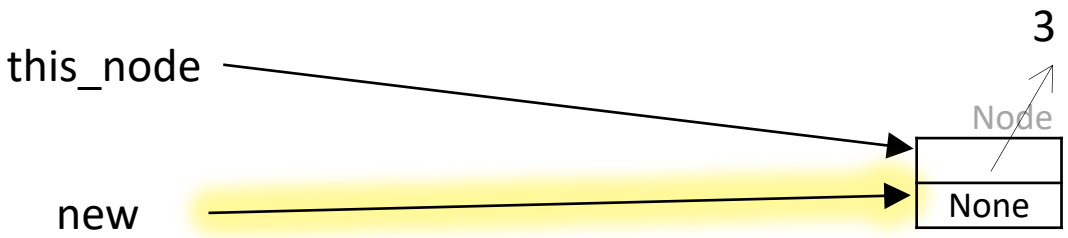



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
>>> my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```

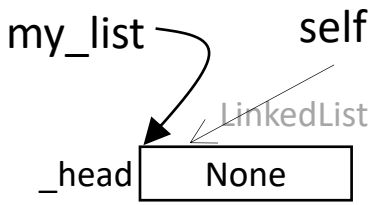


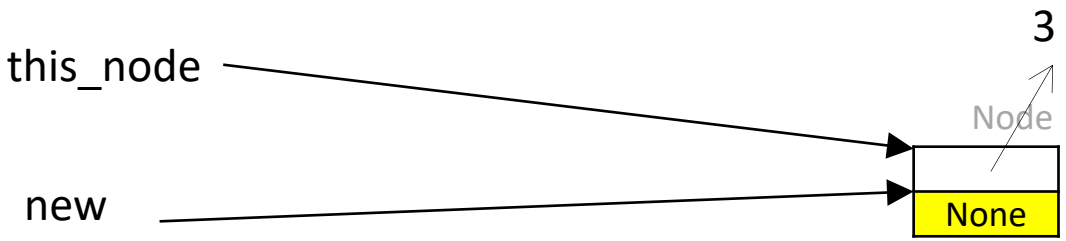



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
>>> my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```

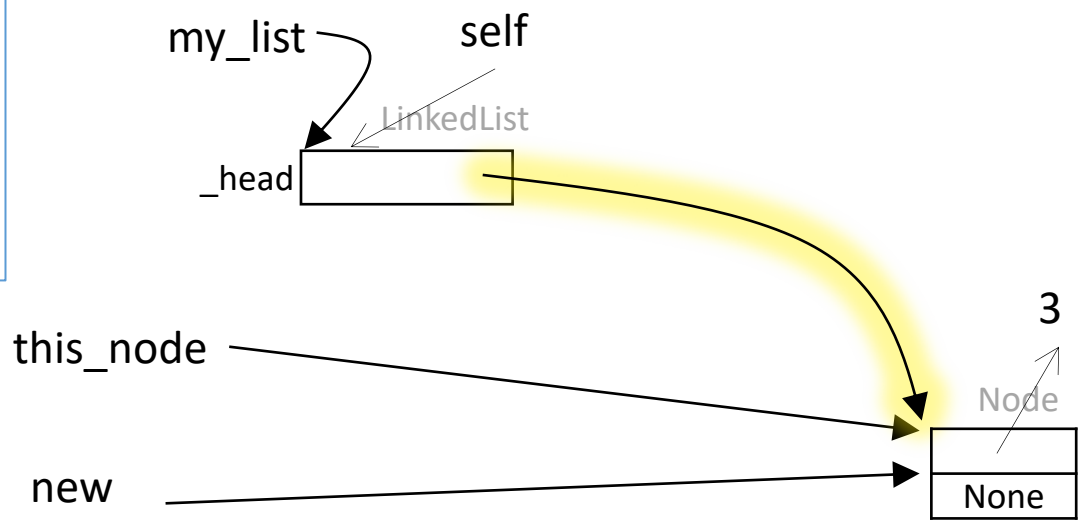


# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

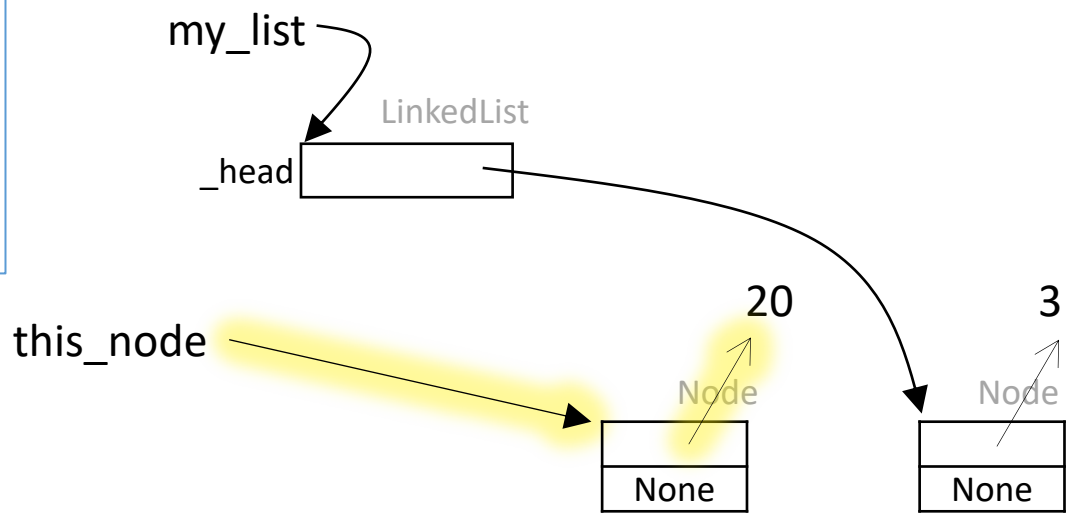
```
>>> my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

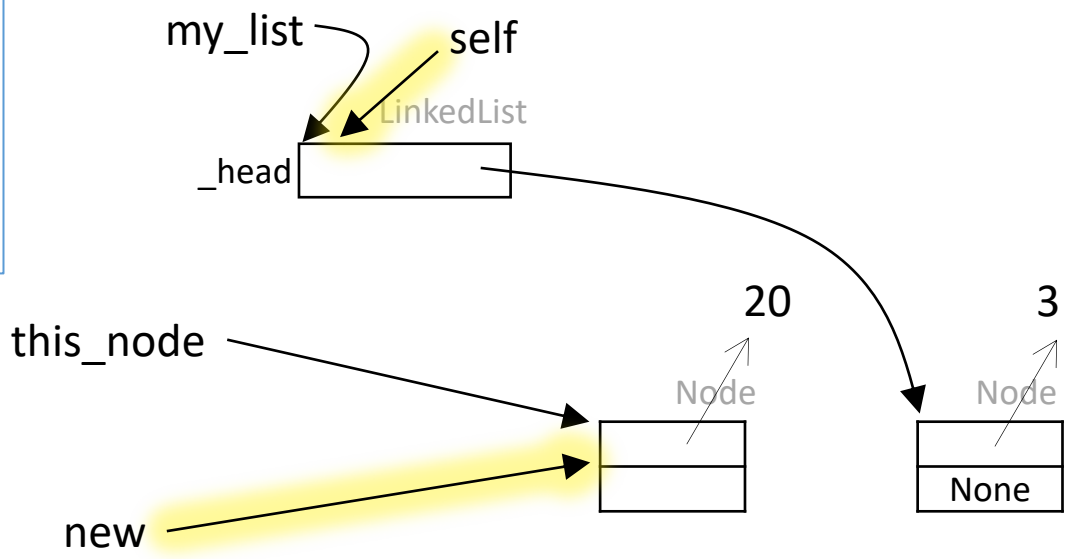
```
>>>my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

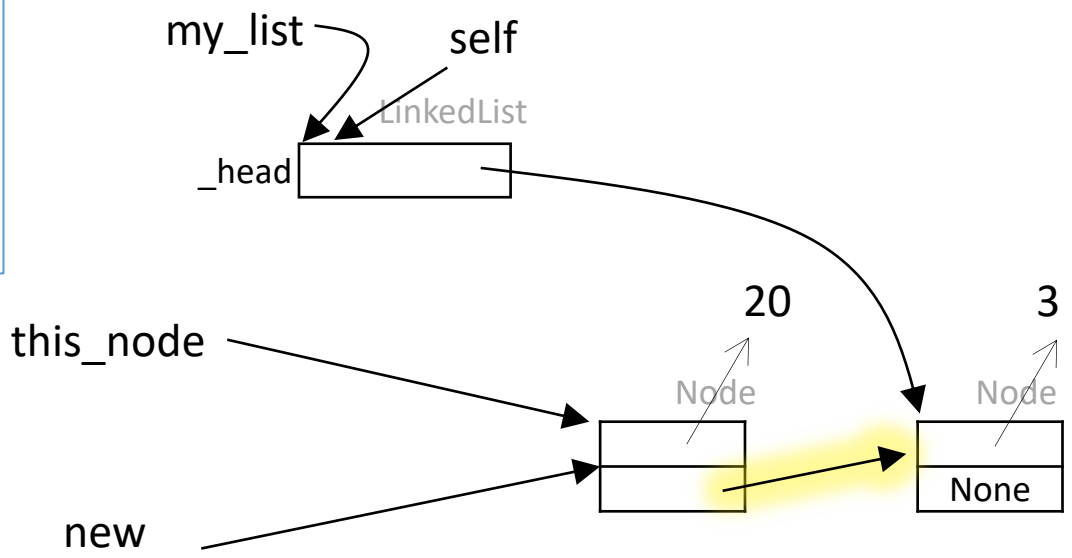
```
>>>my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

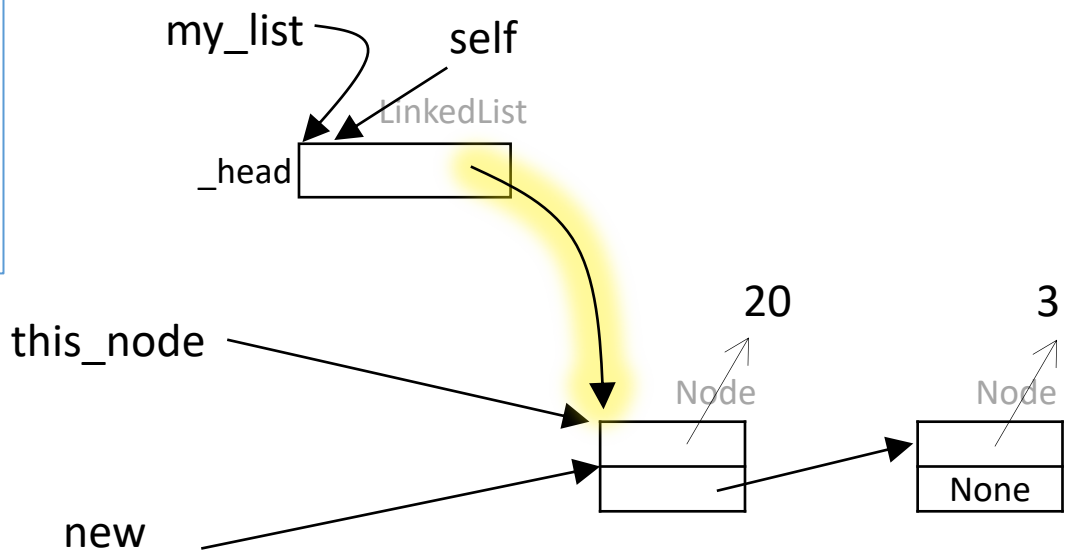
```
>>>my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

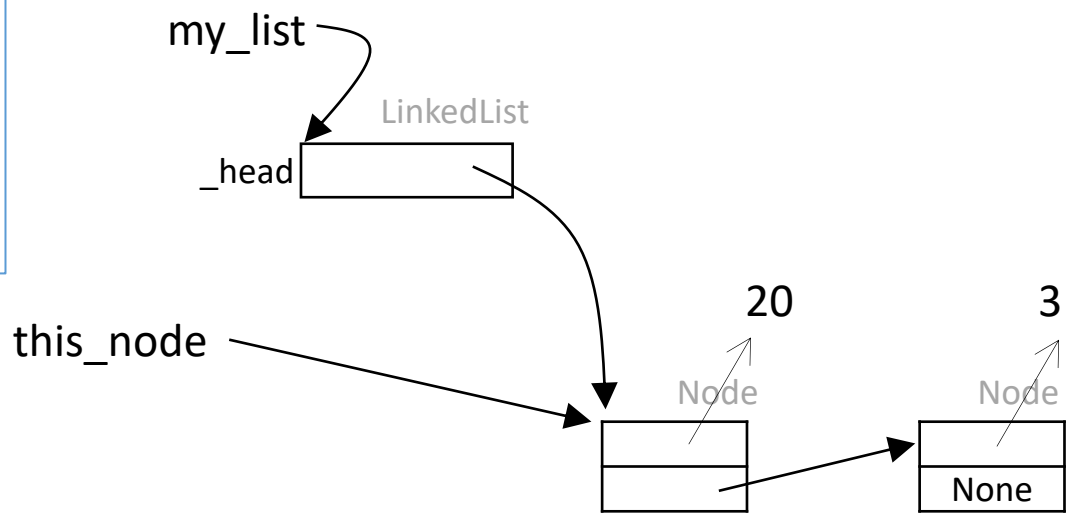
```
>>>my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

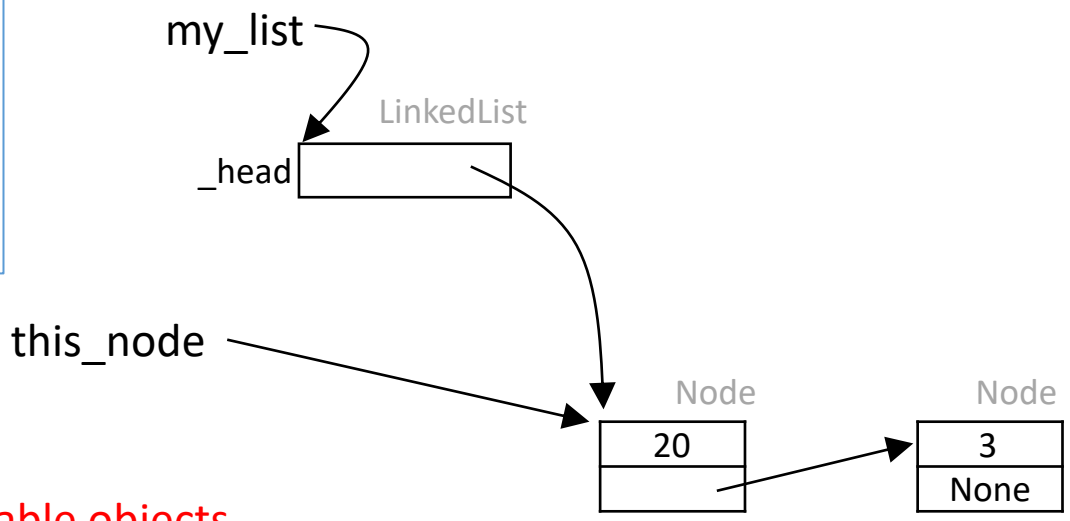
```
>>>my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```



# Creating a linked list: Example 1

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
>>>my_list = LinkedList()
>>> this_node = Node(3)
>>> my_list.add(this_node)
>>> this_node = Node(20)
>>> my_list.add(this_node)
```




Integers are immutable objects.
Can simplify the diagram as shown.

# Exercise-ICA-13

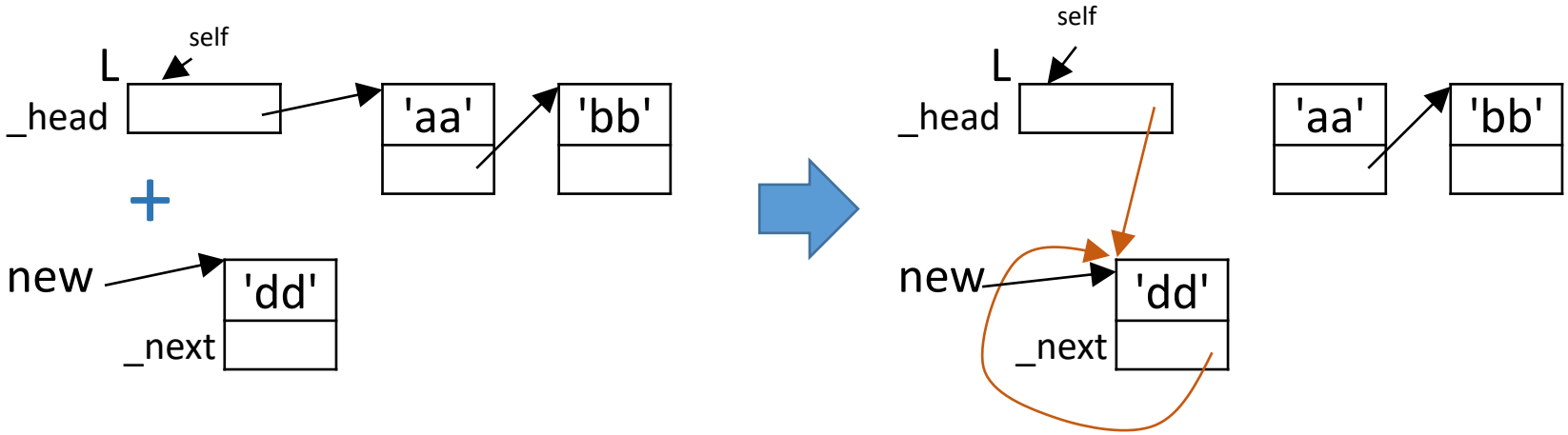
- Do problem 1 in the ICA.

# Adding a node at the head

Changing the order of assignments does not work:

```
def broken_add(self, new):
    self._head = new
    new._next = self._head
```

```
def  add(self, new):
    new._next = self._head
    self._head = new
```



# Creating a linked list: Example 2

```
class Node:
    def _ _init_ _(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def _ _init_ _(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |

# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |

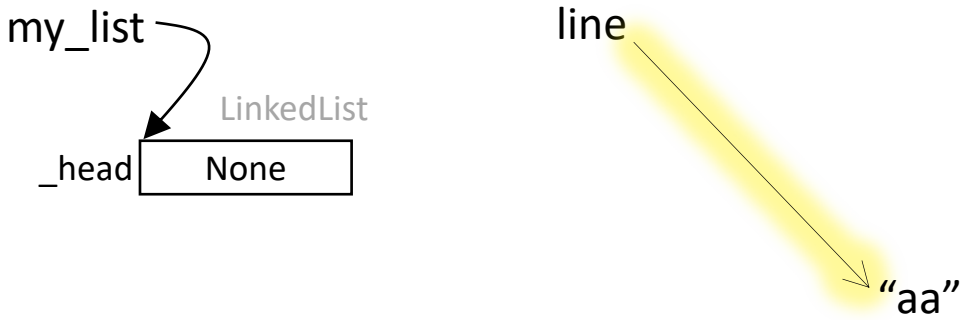
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

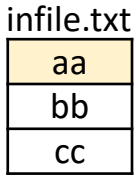
| aa |
|---|
| bb |
| cc |

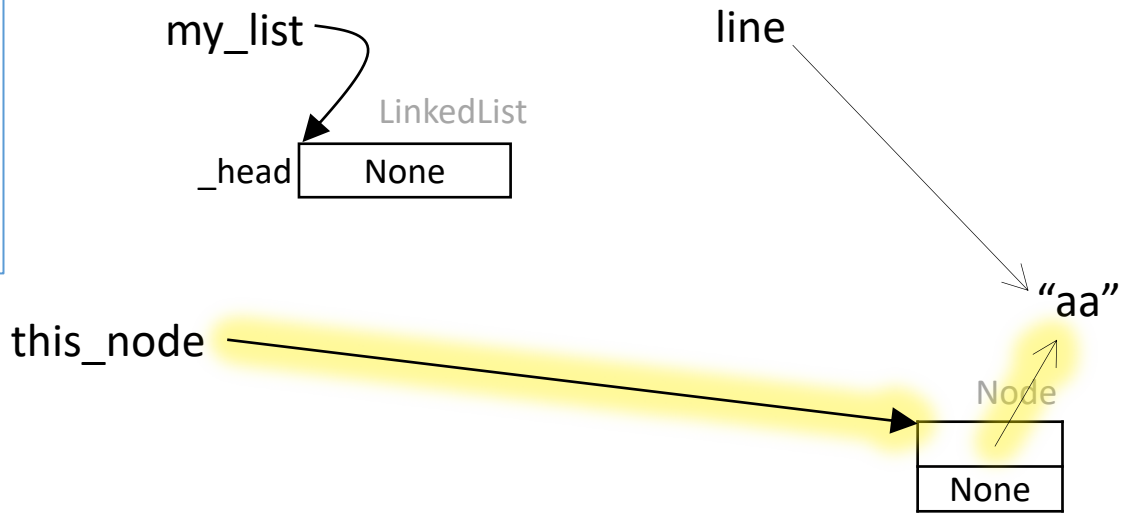


# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

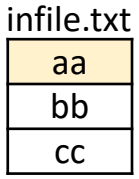


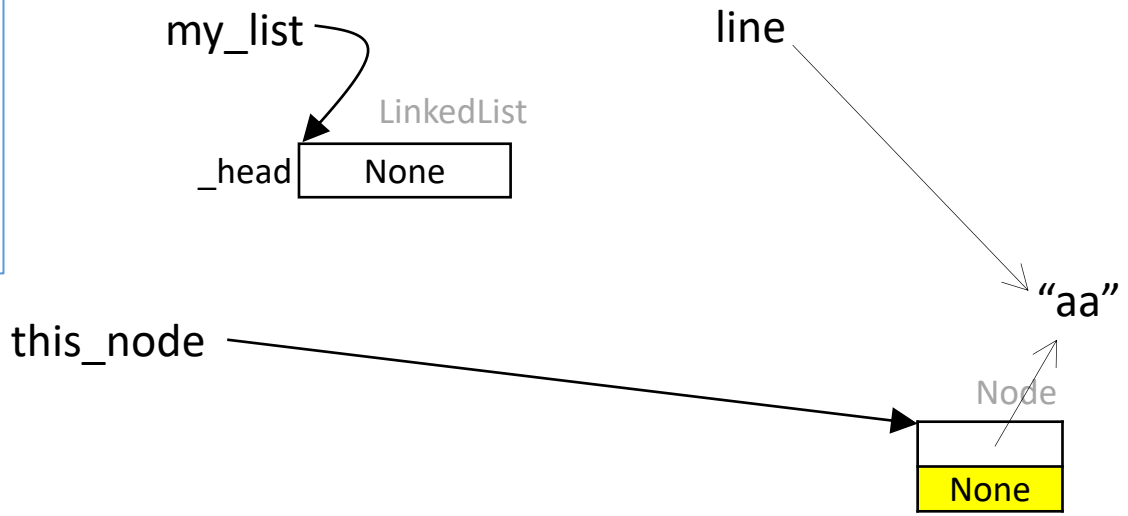



# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

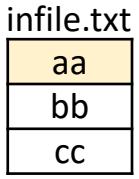


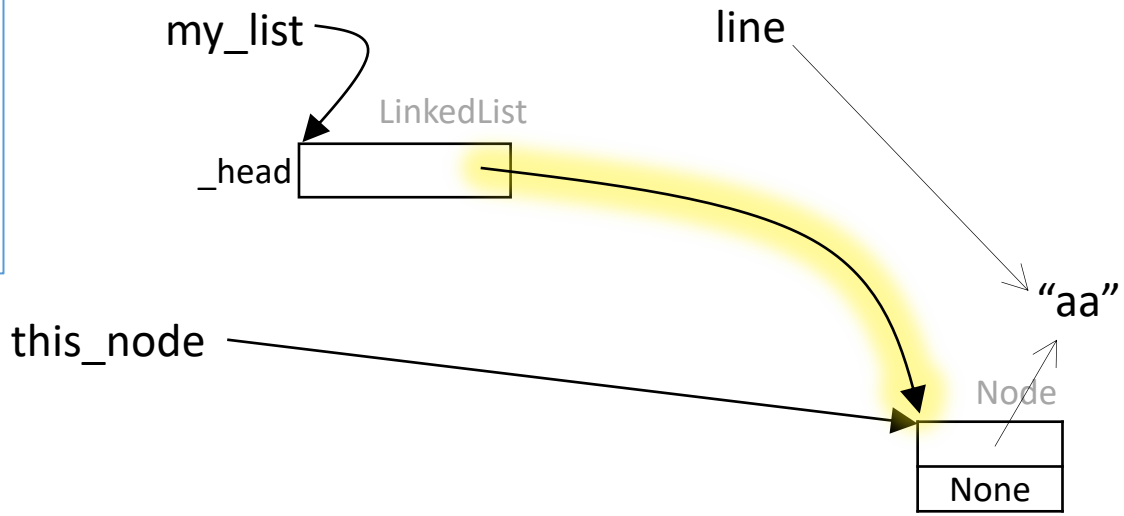



# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

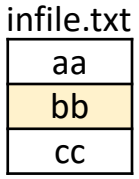


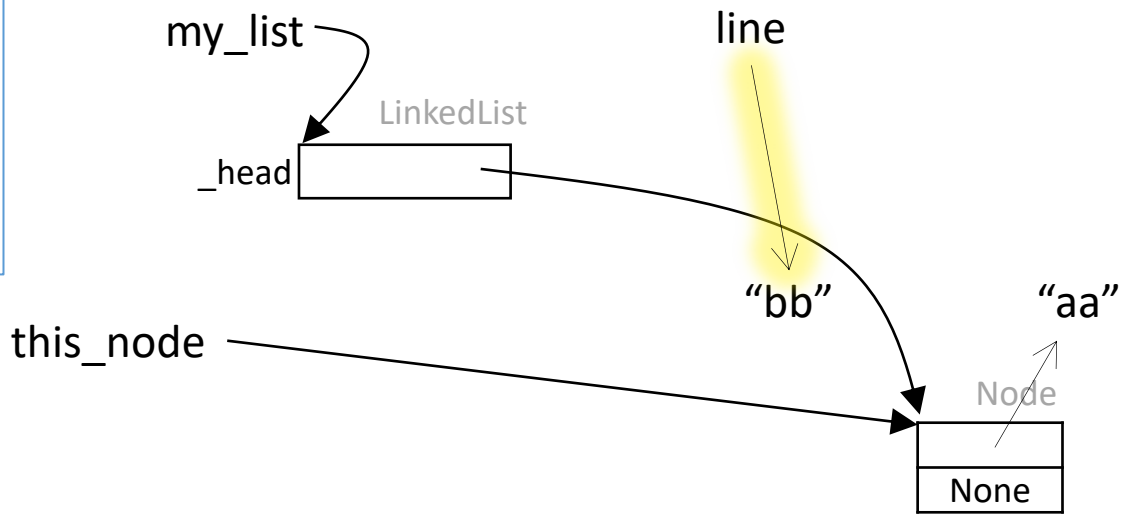



# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

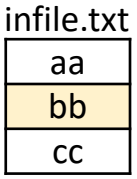


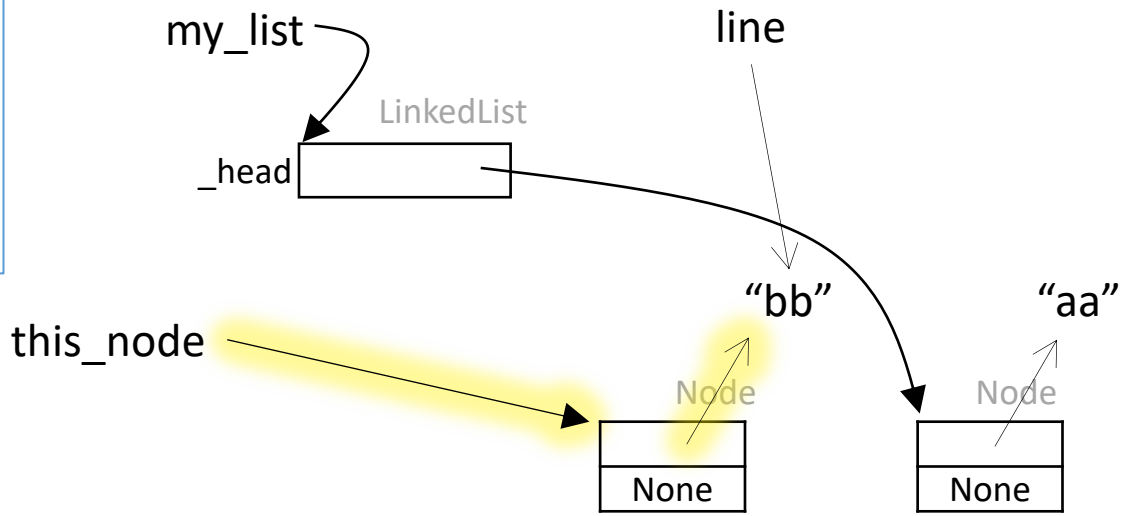



# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```



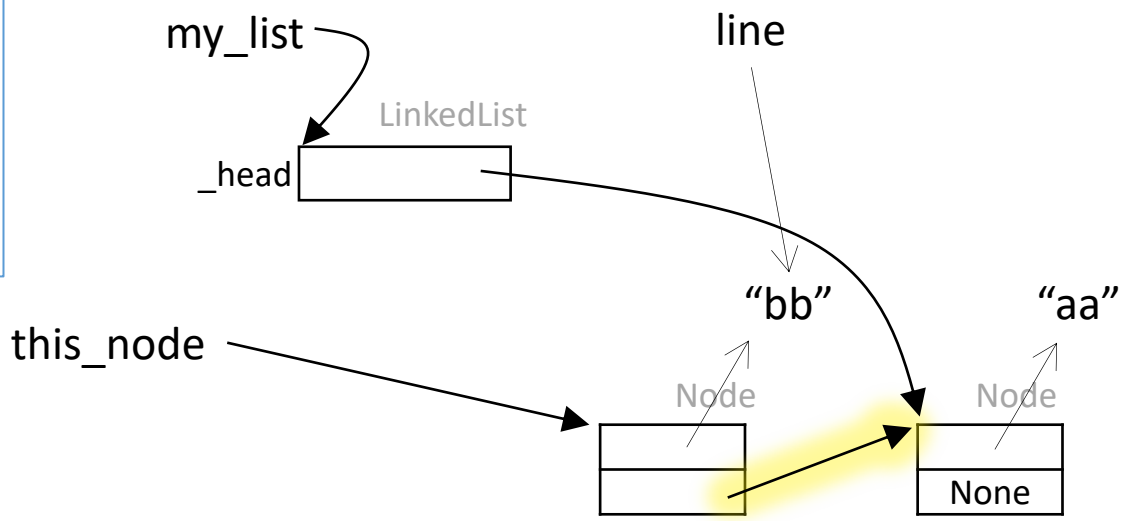
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



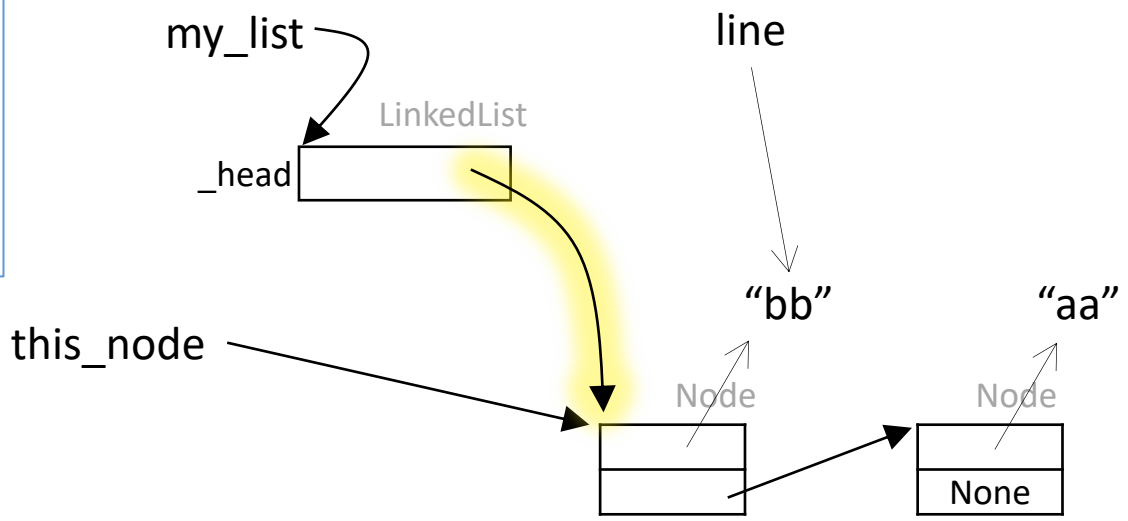
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



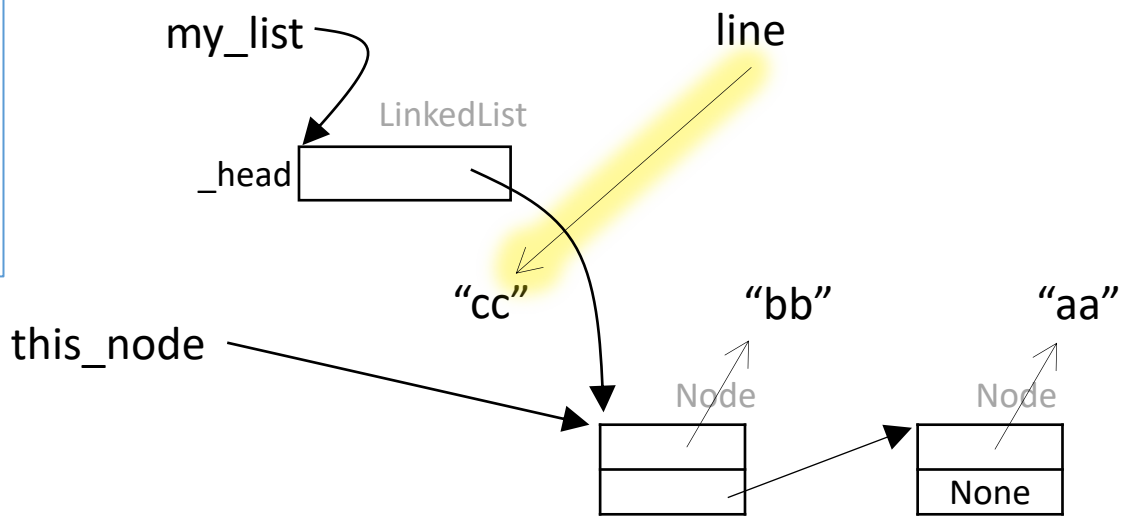
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



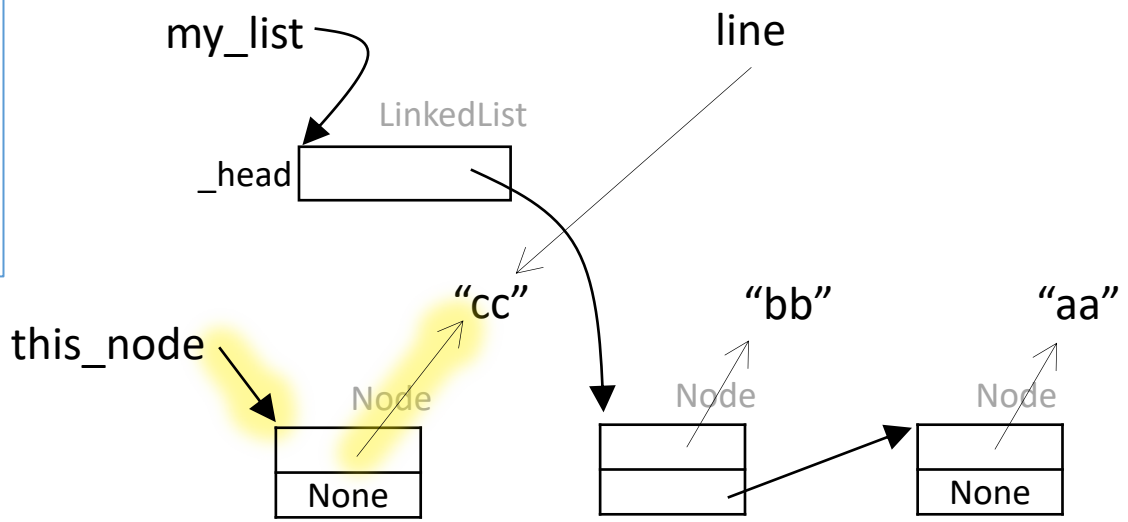
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



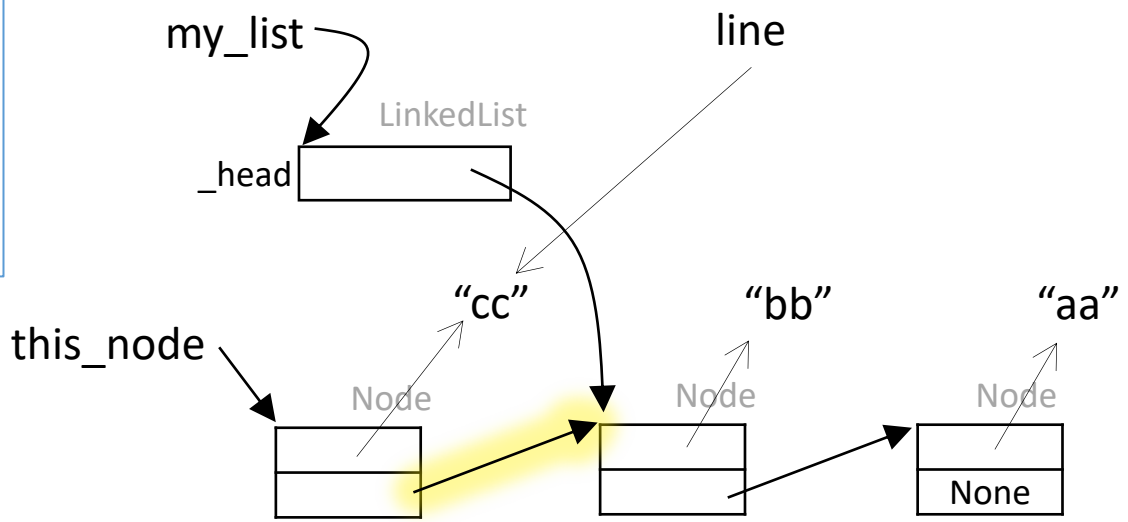
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



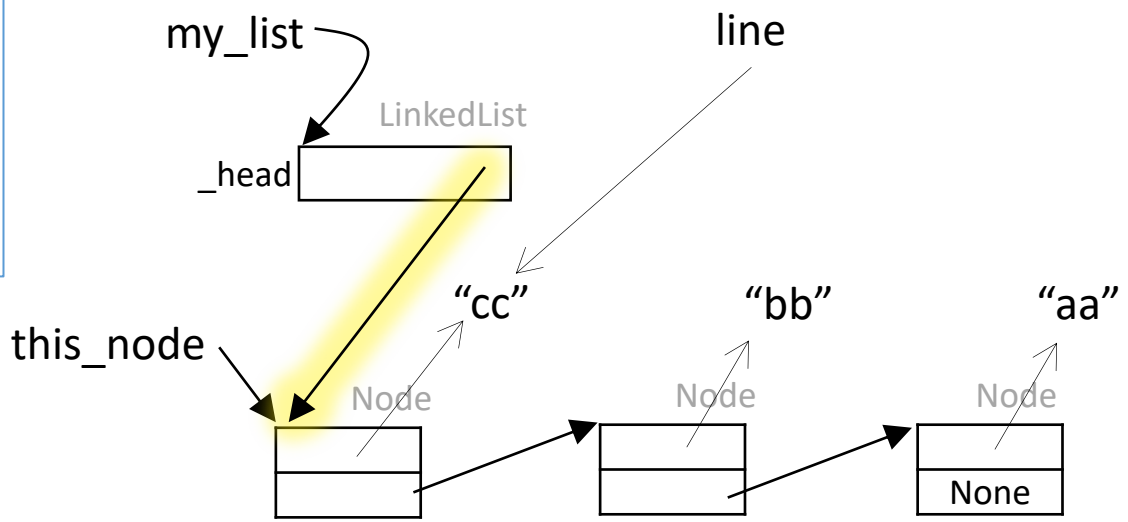
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



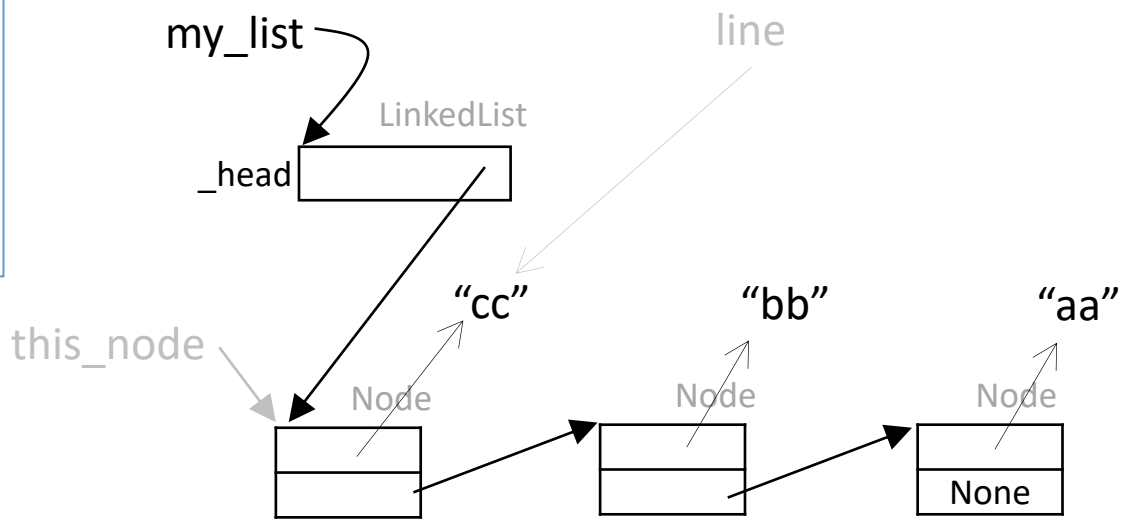
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

| aa |
|---|
| bb |
| cc |



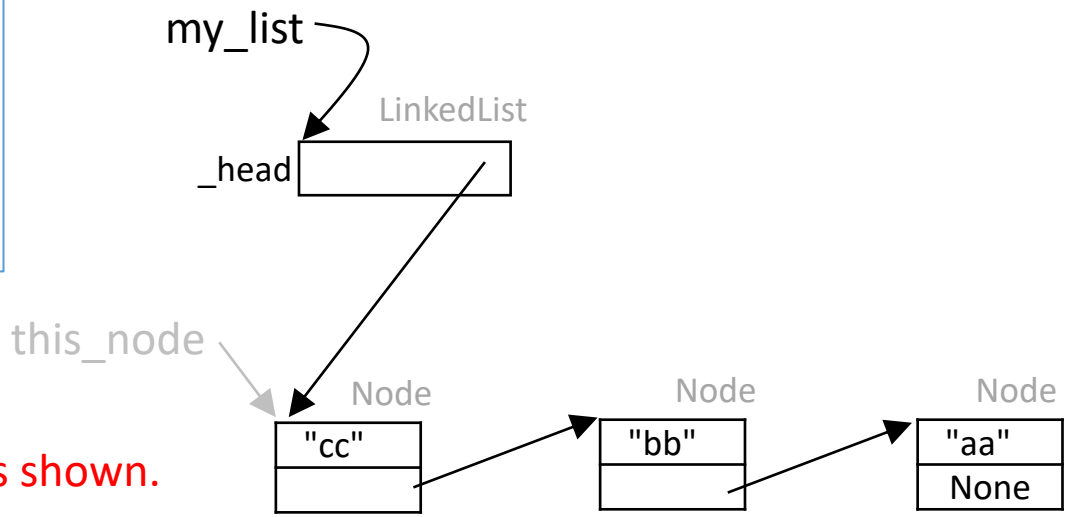
# Creating a linked list: Example 2

```
class Node:
    def __init__(self, value):
        self._value = value
        self._next = None
    . . .
class LinkedList:
    def __init__(self):
        self._head = None

    def add(self, new):
        new._next = self._head
        self._head = new
```

```
infile = open("infile.txt")
my_list = LinkedList()
for line in infile:
    this_node = Node(line)
    my_list.add(this_node)
```

infile.txt

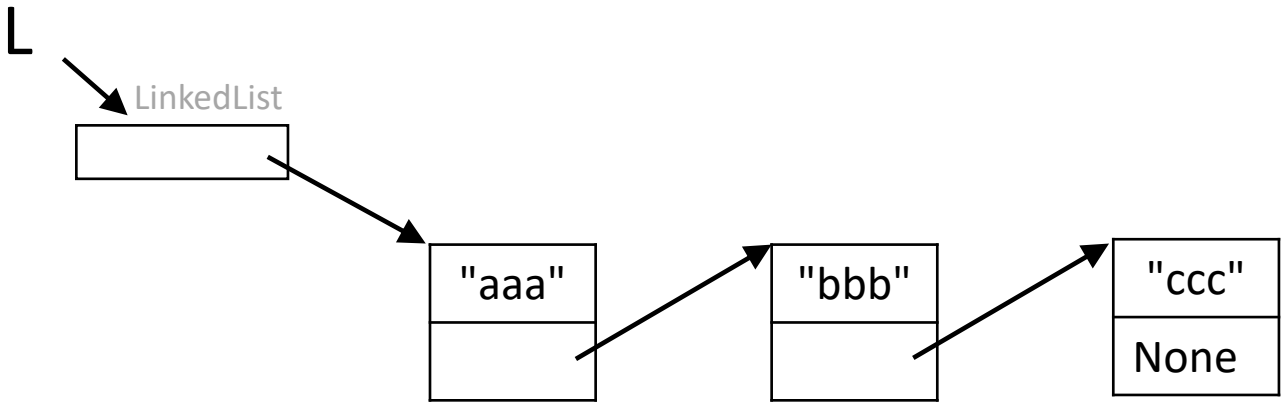
| aa |
|---|
| bb |
| cc |




Can simplify the diagram as shown.

# Visiting all of the nodes

Suppose we want to do something to each node of a list.




How do we loop through the nodes (elements)?

- with a built-in list, we would use a for or while loop

# Consider a Python list

Suppose we want to do something to each node of a list.

In a Python list, use a for or while loop:

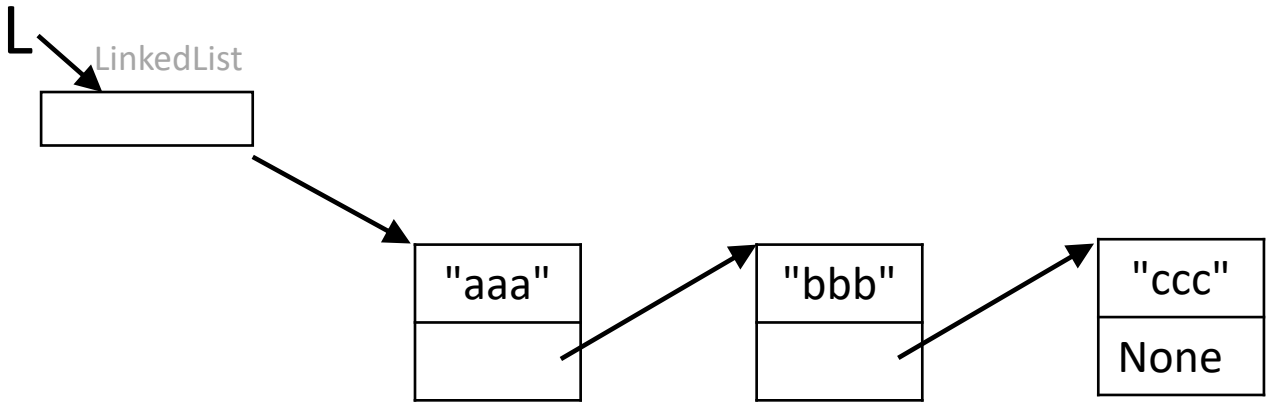
```
alist = [20, 6, 17, 4, 28, 5]
i = 0                          #start at the beginning
while i < len(alist):  # stop when you hit the end
    < do something with alist[i] >
    i = i + 1                  # go to the next element
```

How do we do each of these steps with a linked list?

- start at the beginning
- go to the next element
- stop when you hit the end

# Visiting all of the nodes

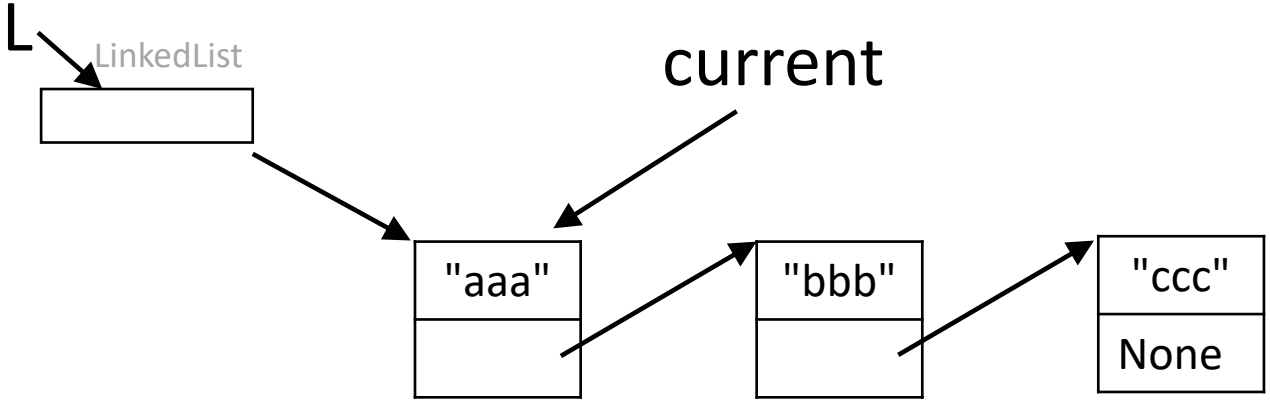
Suppose we want to do something to each node of a list:




Start at the beginning: use the head of the list

# Visiting all of the nodes

Suppose we want to do something to each node of a list:



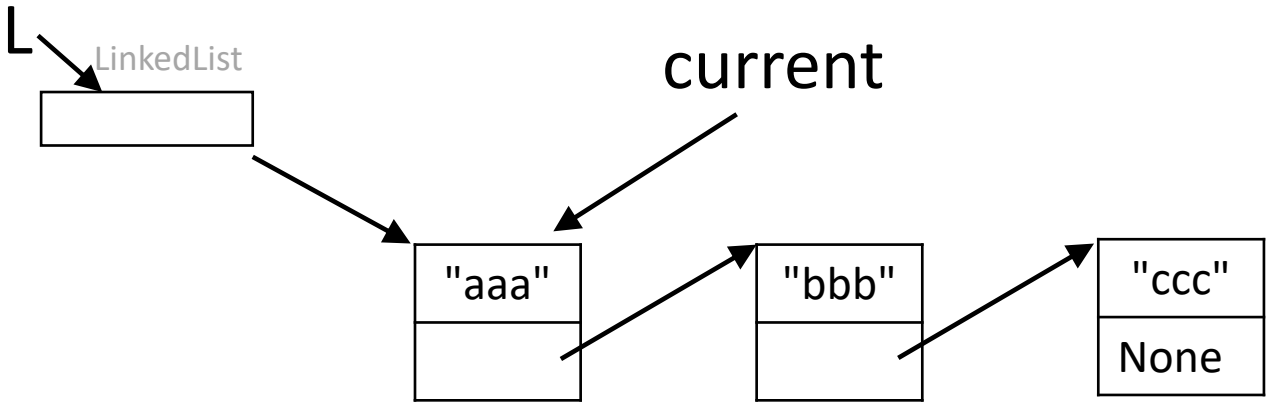

Start at the beginning: use the head of the list

```
current = self._head
```

Use a variable to refer to each element of the list in turn.

# Visiting all of the nodes

Suppose we want to do something to each node of a list:



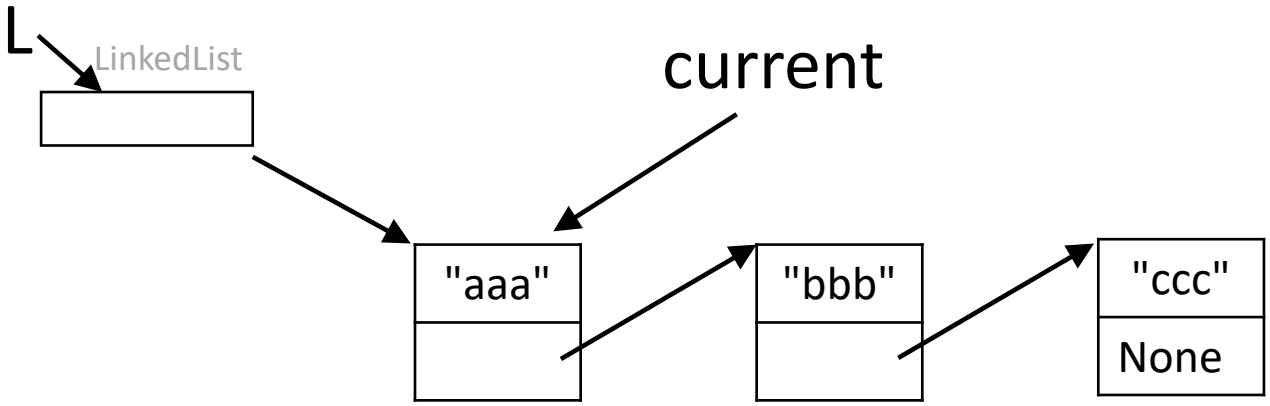

Start at the beginning: use the head of the list

```
current = self._head
```

Go to the next element: use the _next attribute of current

# Visiting all of the nodes

Suppose we want to do something to each node of a list:




Start at the beginning: use the head of the list

```
current = self._head
```

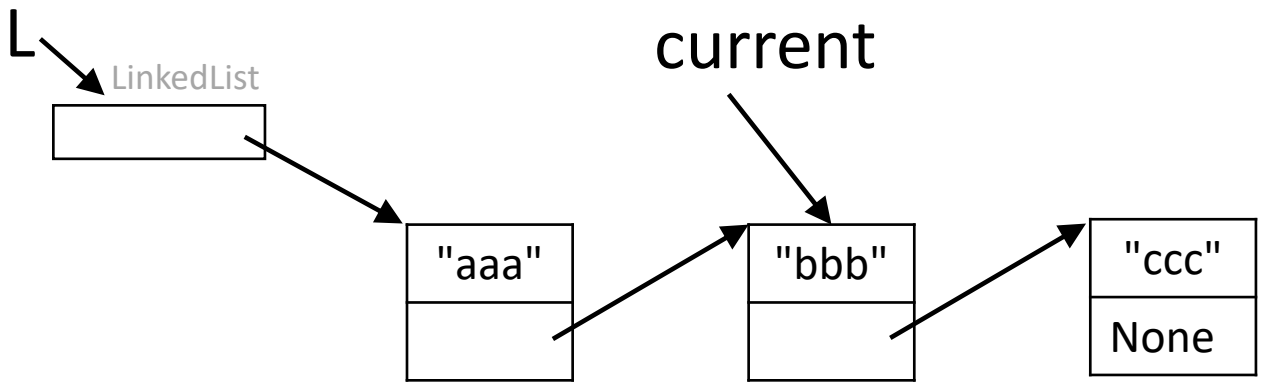
Go to the next element: use the _next attribute of current

```
current = current._next
```

When this is executed, the reference will change.

# Visiting all of the nodes

Suppose we want to do something to each node of a list:




Start at the beginning: use the head of the list

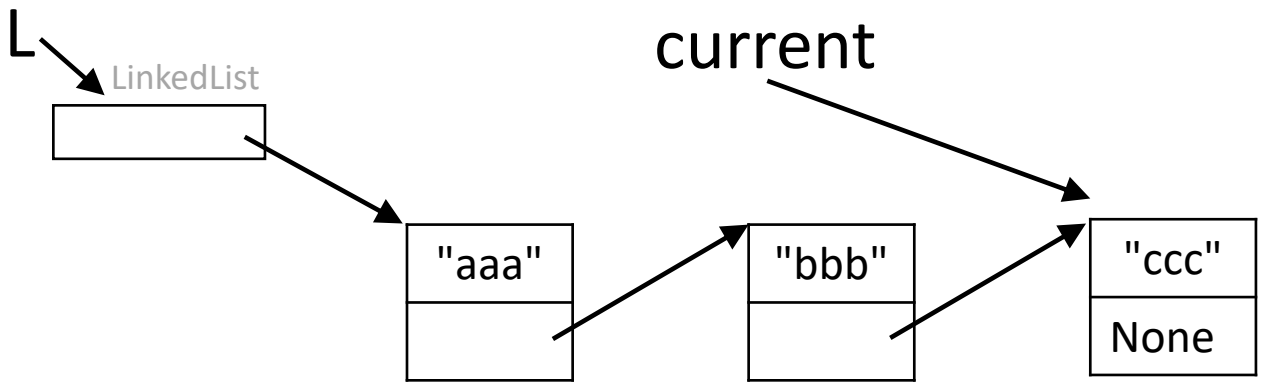
```
current = self._head
```

Go to the next element: use the next attribute

```
current = current._next
```

# Visiting all of the nodes

Suppose we want to do something to each node of a list:




Start at the beginning: use the head of the list

```
current = self._head
```

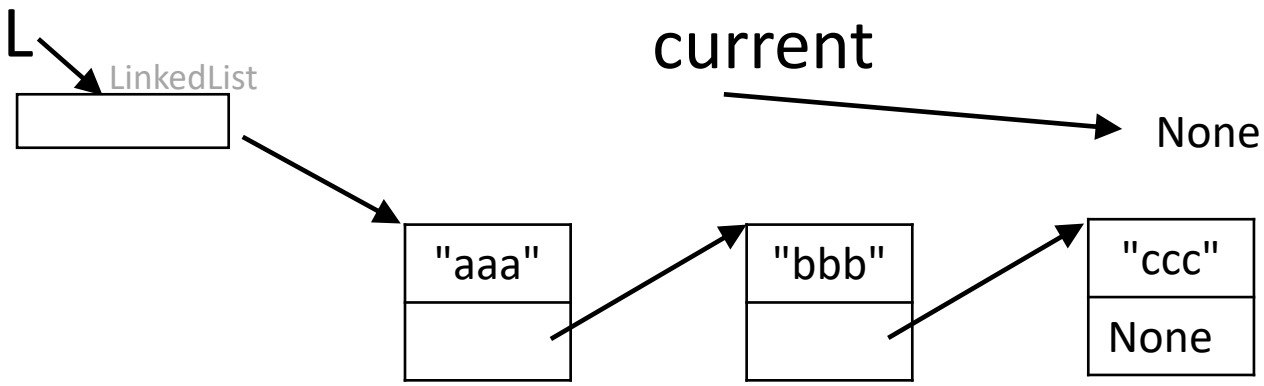
Go to the next element: use the next attribute

```
current = current._next
```

Keep going until we hit the end of the list.

# Visiting all of the nodes

Suppose we want to do something to each node of a list:




Start at the beginning: use the head of the list

```
current = self._head
```

Go to the next element: use the next attribute

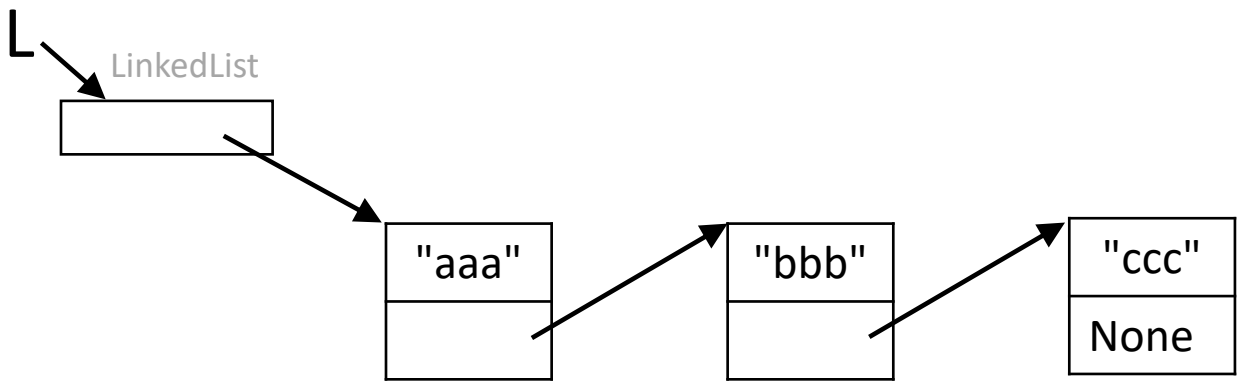
```
current = current._next
```

Stop when you hit None

Stop when current == None

# Visiting all of the nodes

Suppose we want to do something to each node of a list:

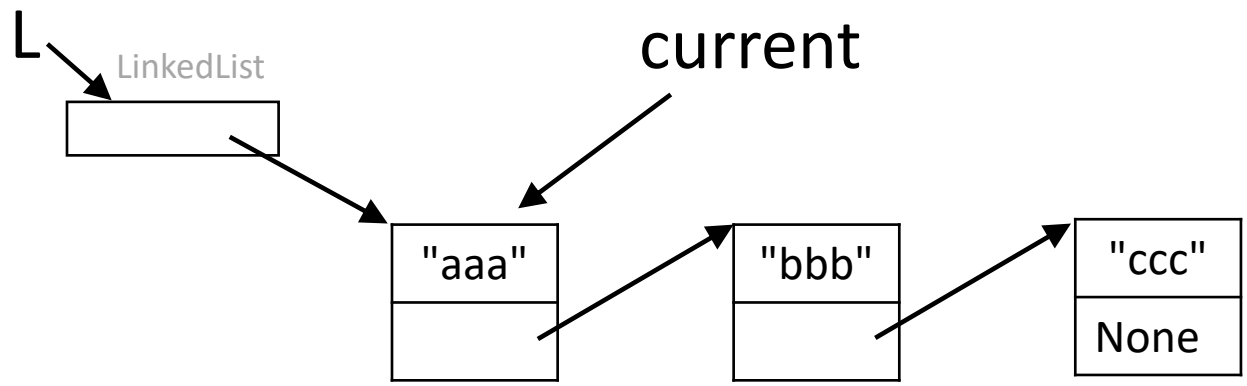



Code to iterate through a linked list:

```
current = self._head
while current != None:
        <do something with current._value>
        current = current._next
```

# Visiting all of the nodes

Suppose we want to do something to each node of a list:

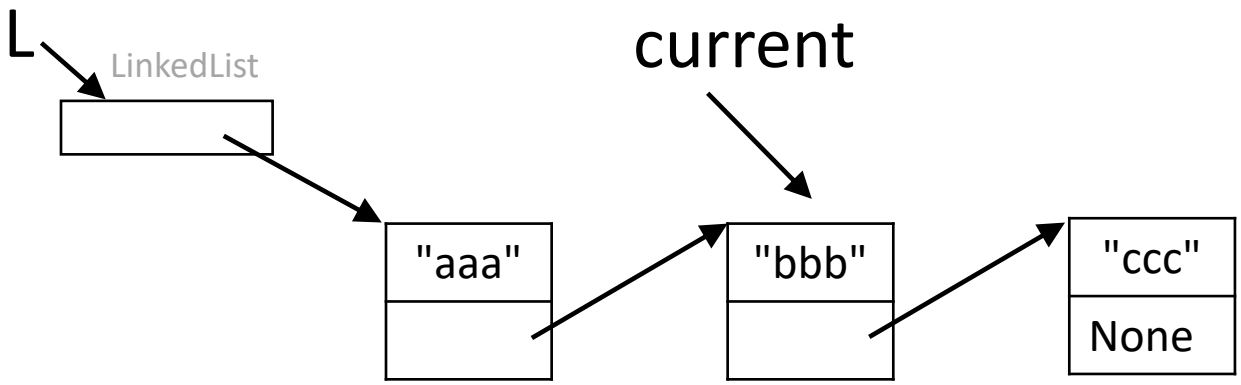



Code to iterate through a linked list:

```
current = self._head
while current != None:
        <do something with current._value>
        current = current._next
```

# Visiting all of the nodes

Suppose we want to do something to each node of a list:

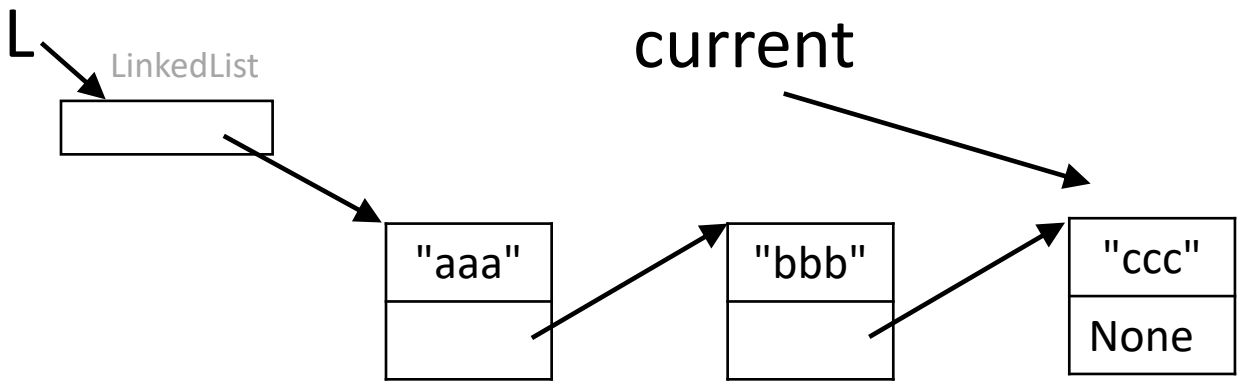



Code to iterate through a linked list:

```
current = self._head
while current != None:
        <do something with current._value>
        current = current._next
```

# Visiting all of the nodes

Suppose we want to do something to each node of a list:

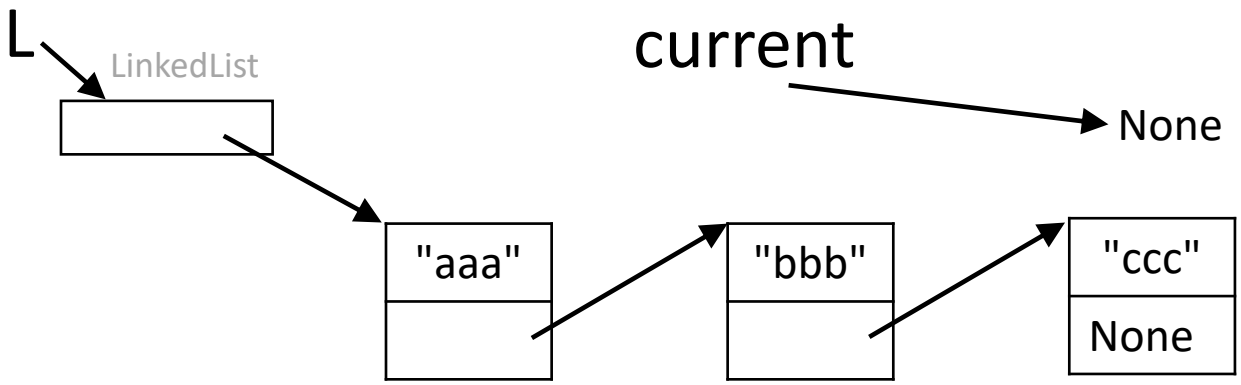



Code to iterate through a linked list:

```
current = self._head
while current != None:
        <do something with current._value>
        current = current._next
```

# Visiting all of the nodes

Suppose we want to do something to each node of a list:




Code to iterate through a linked list:

```
current = self._head
while current != None:
        <do something with current._value>
        current = current._next
```

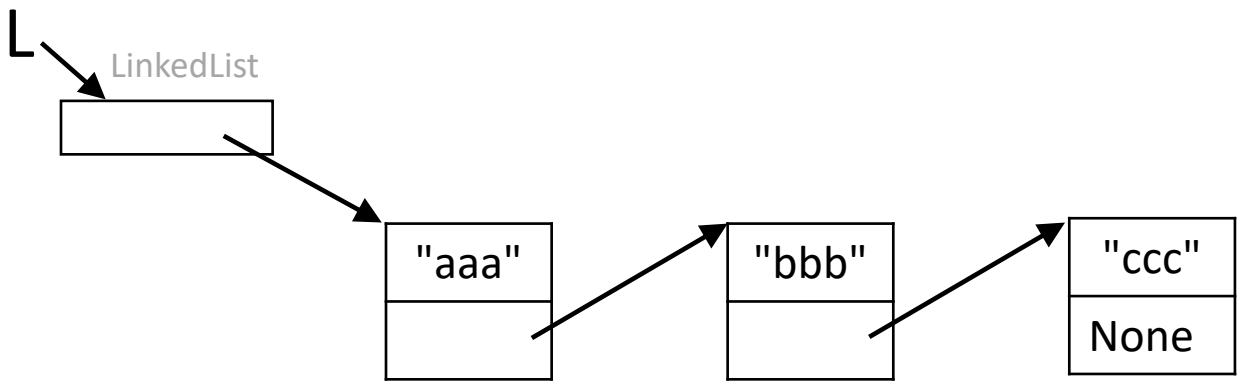
# Example: print each element

```
class LinkedList:
    def __init__(self):
        self._head = None
    ....

    def  print_elements(self):
          current = self._head
          while current != None:
              print(str(current._value))
              current = current._next
```

# Exercise-ICA-13

- Do problem 2 and 3.
- (Do the extra problem 4 if you have time.)

# Iterating through a list



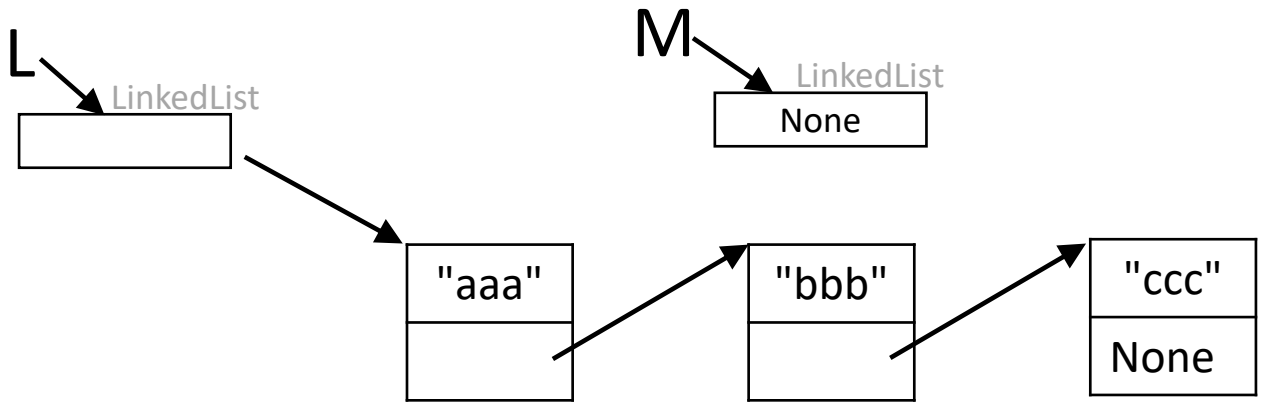

Template for iterating through a linked list:

```
def visit_nodes(self):
    current = self._head
    while current != None:
        <do something with current._value>
        current = current._next
```

Or,
while current is not None:

# Iterating through a list

What if the list is empty? Does the code work?

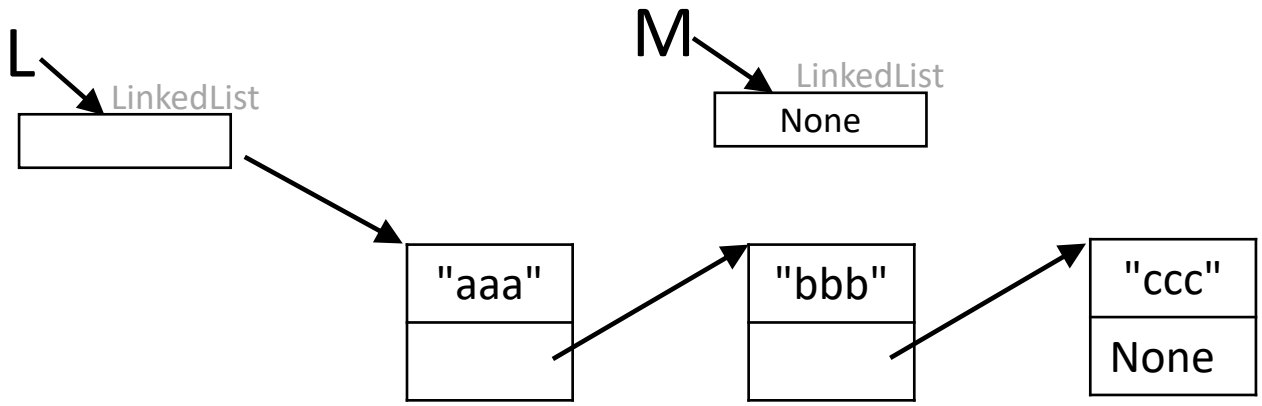



Code to iterate through a linked list:

```
def visit_nodes(self):
    current = self._head
    while current != None:
        <do something with current._value>
         current = current._next
```

# Iterating through a list

What if the list is empty? Does the code work? Yes…




Code to iterate through a linked list:

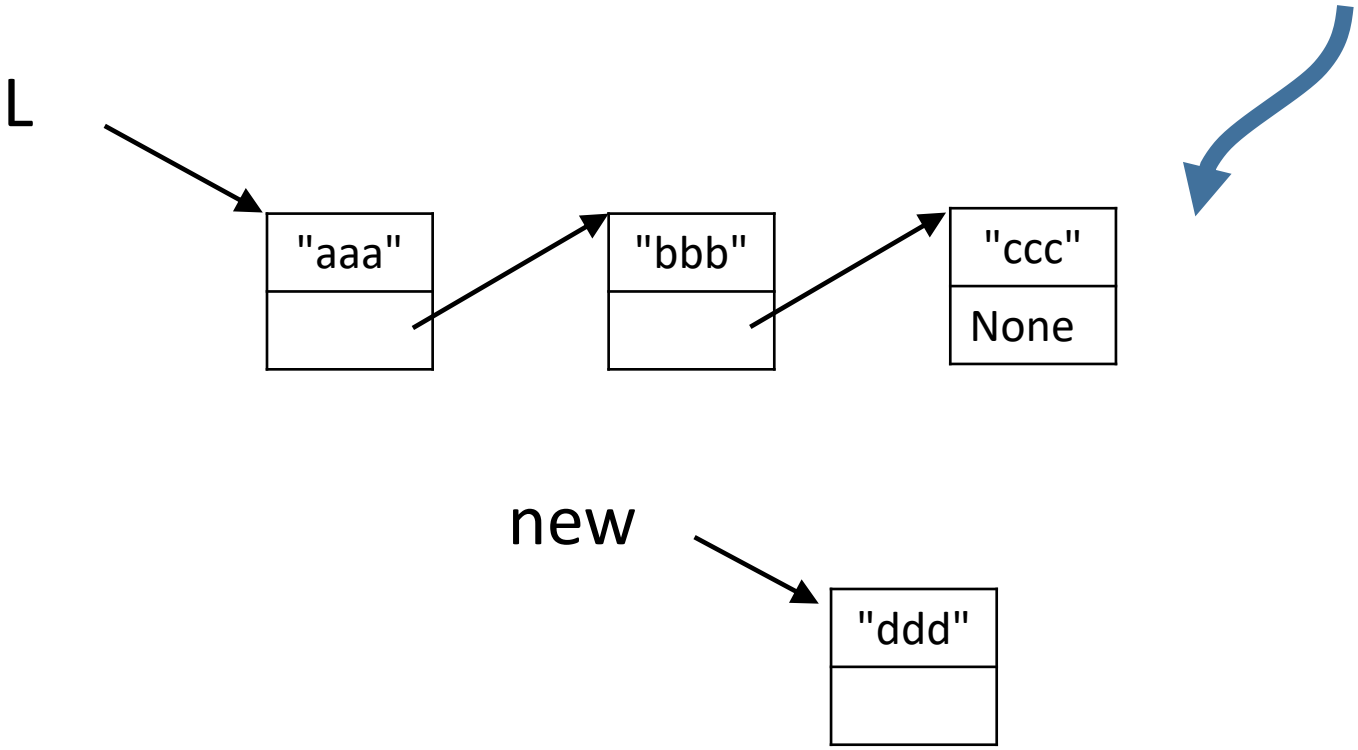
```
if self._head == None:
    <do something special>
    return
current = self._head
while current != None:
        <do something with current._value>
        current = current._next
```

but, might need special handling for an empty list.

# adding to the end (tail) of the list

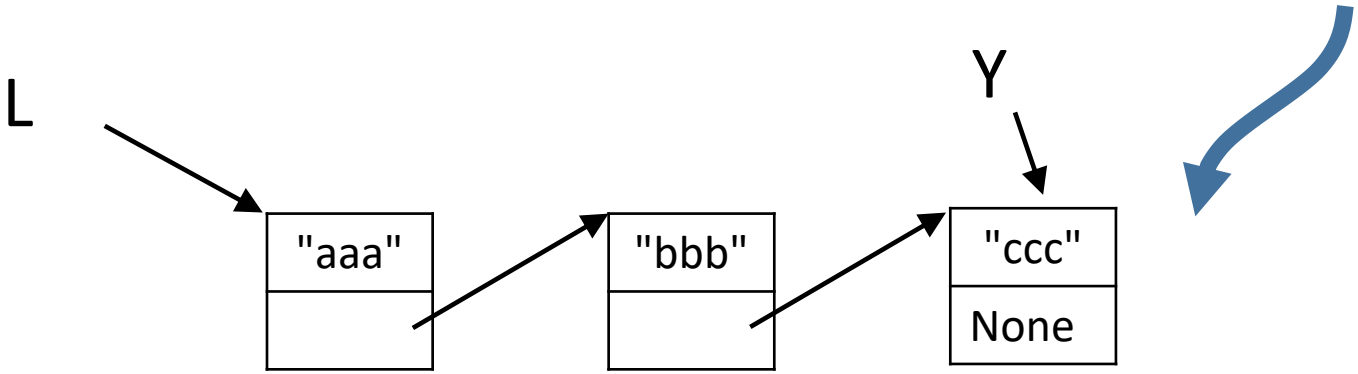
# Adding a node to the tail

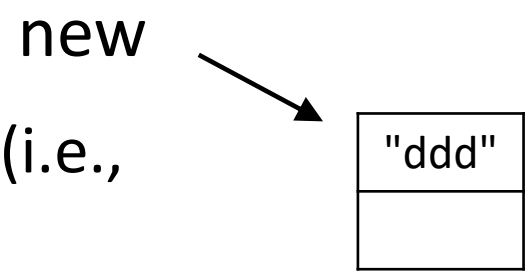
Suppose we want to add a node to the end of a list:



# Adding a node to the tail

Suppose we want to add a node to the end of a list:




To add a node new at the end (i.e., tail) of a list L:

1. find the last element Y of L
2. Y._next = new

# Adding a node to the tail

To add a node new at the end (i.e., tail) of a list L:

1. find the last element Y of L
2. Y._next = new

# Adding a node to the tail

To add a node new at the end (i.e., tail) of a list L:

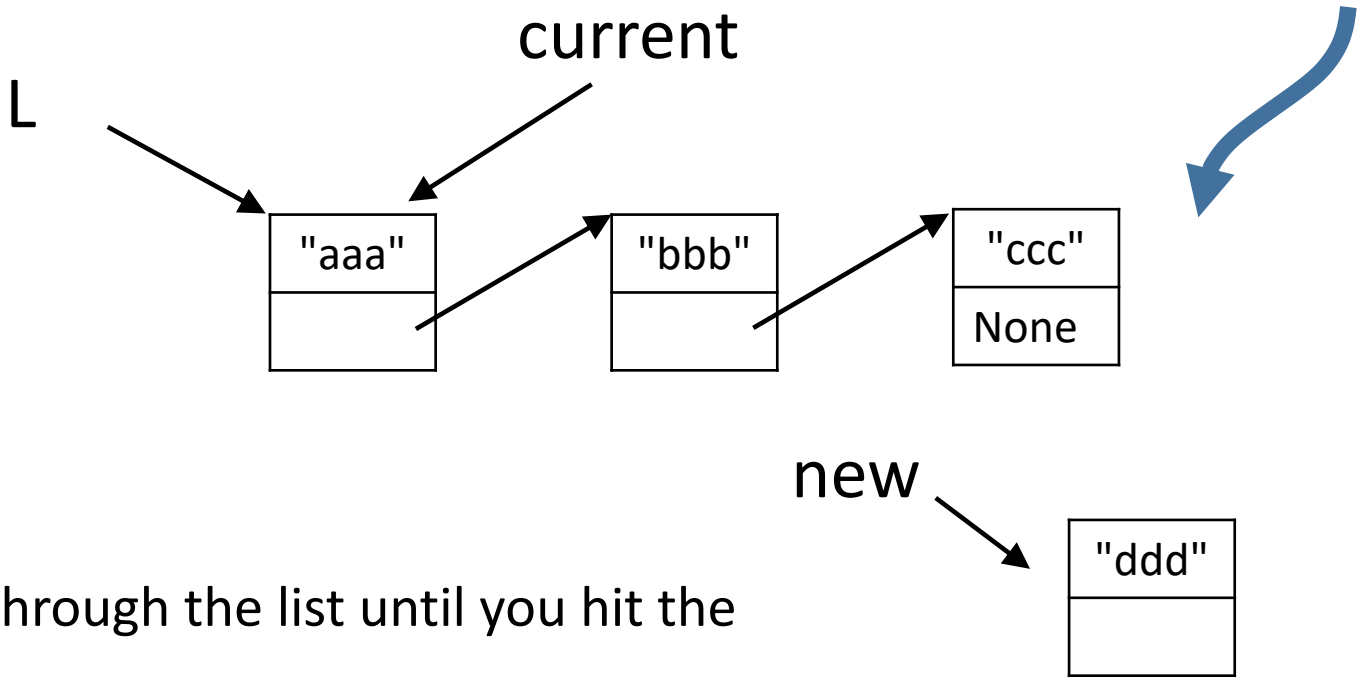
1. find the last element Y of L
2. Y._next = new

Idea:

Use the template code to iterate through a list

# Adding a node to the tail

Suppose we want to add a node to the end of a list:

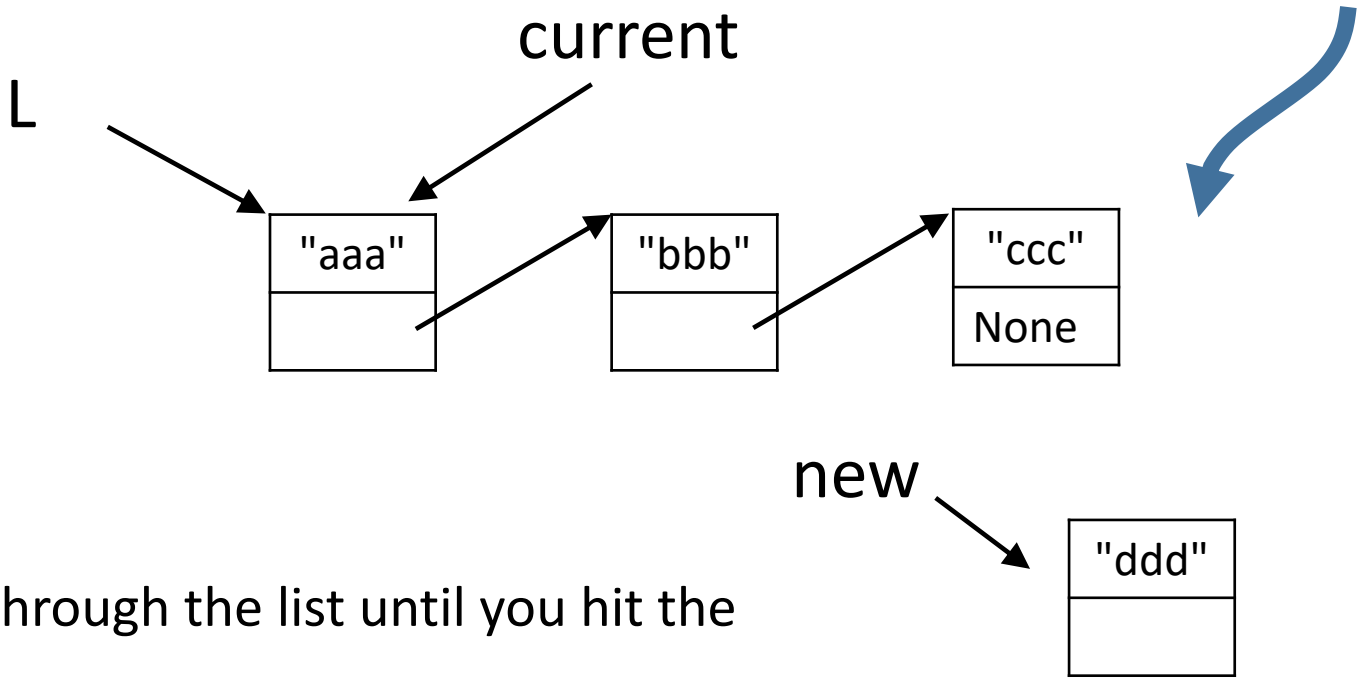



Progress through the list until you hit the end:

```
current = self._head
while current != None:
        current = current._next
```

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



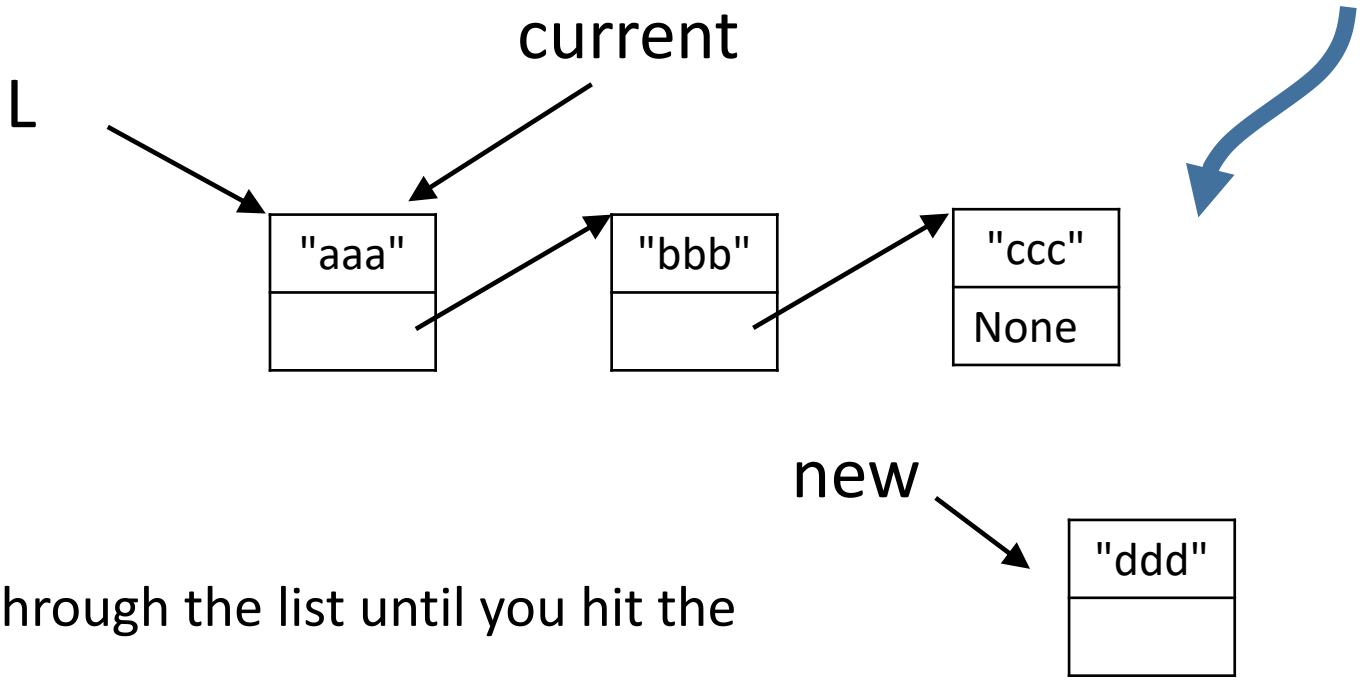

Progress through the list until you hit the end:

```
current = self._head
while current != None:
        current = current._next
```

Any issues with this code?

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



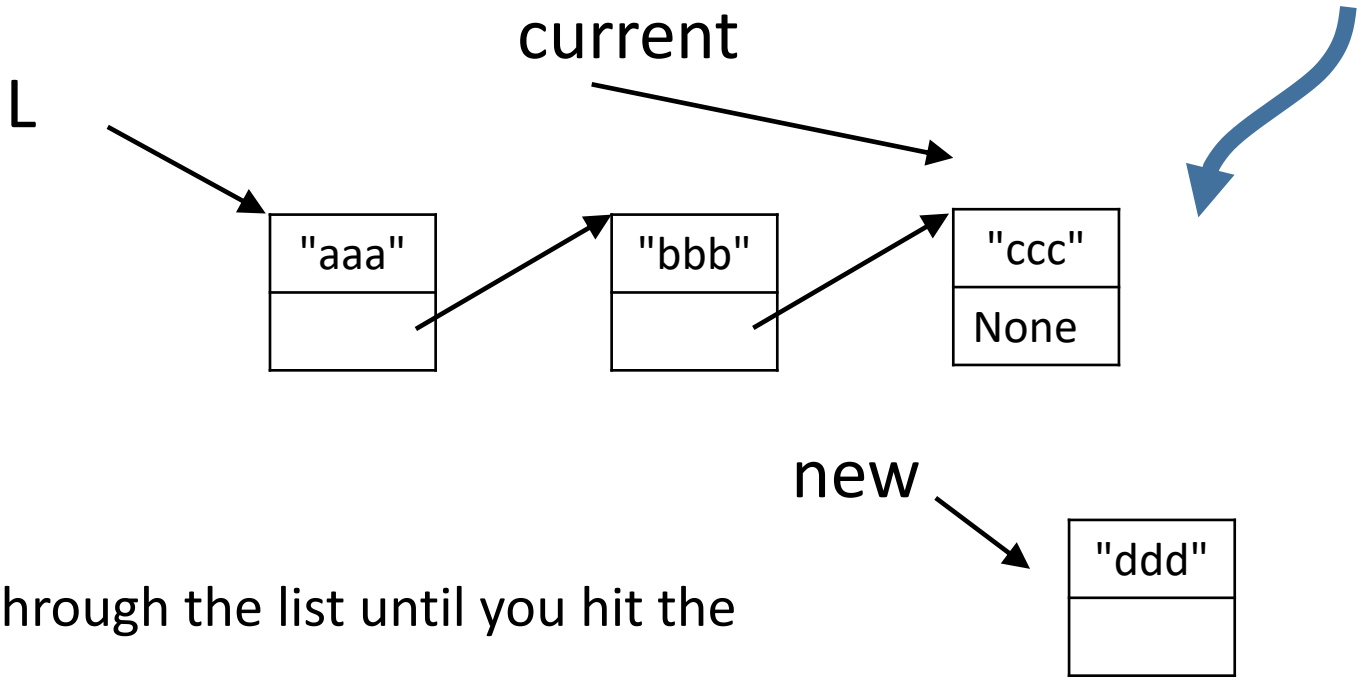

Progress through the list until you hit the end:

```
current = self._head
while current != None:
        current = current._next
```

Issues? Yes! The reference in current is None when we exit the loop.

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



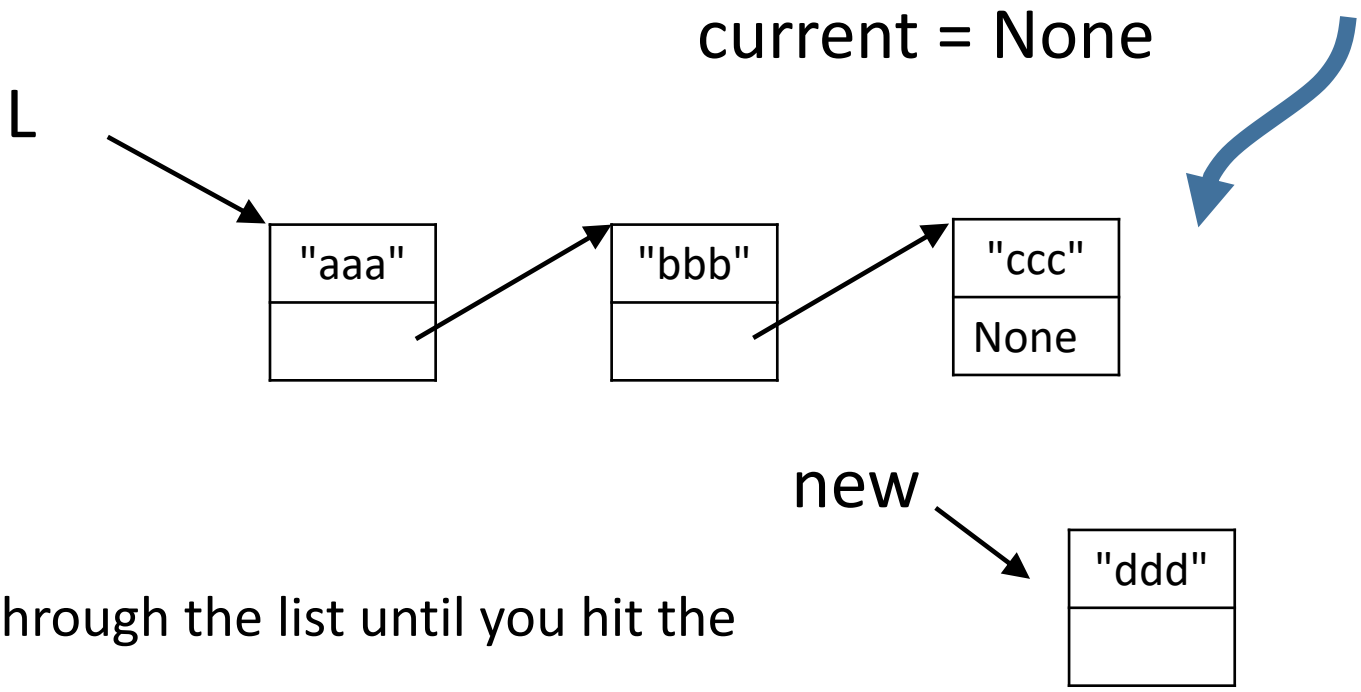

Progress through the list until you hit the end:

```
current = self._head
while current != None:
        current = current._next
```

Issues? Yes! What happens here on the last iteration?

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



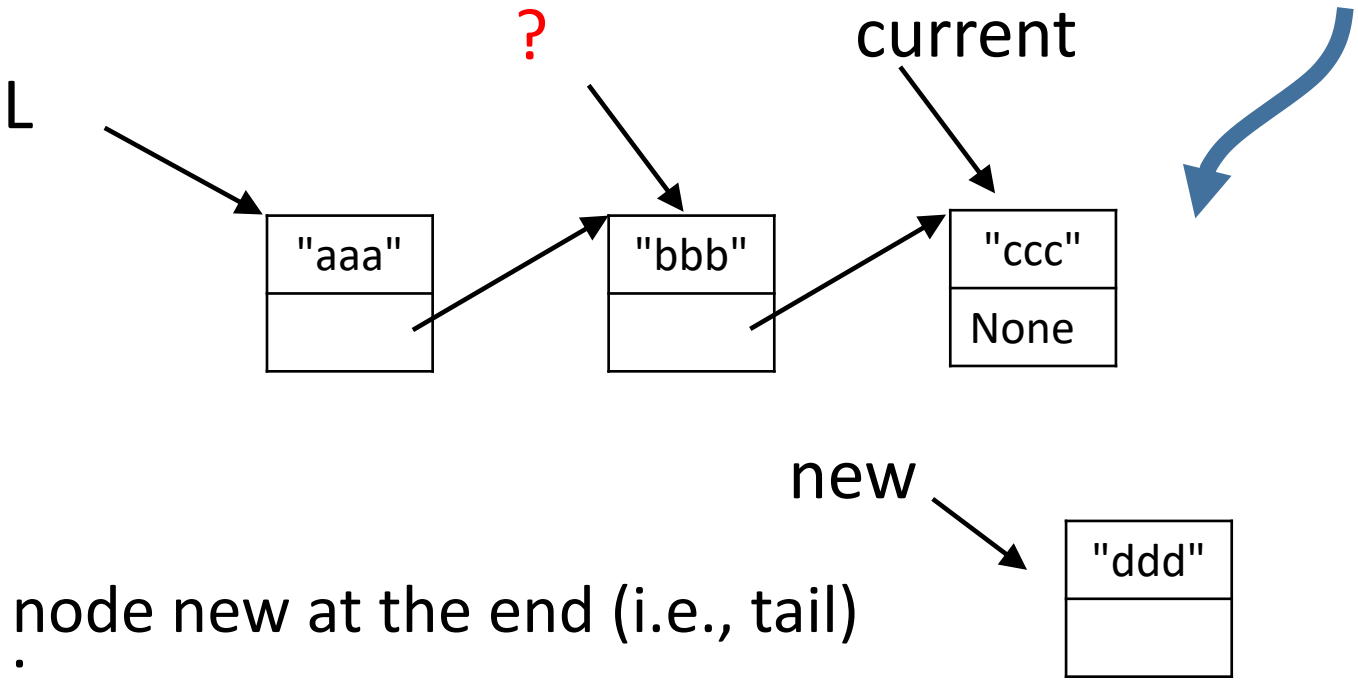

Progress through the list until you hit the end:

```
current = self._head
while current != None:
        current = current._next
```

The reference in current is None when we exit the loop.

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



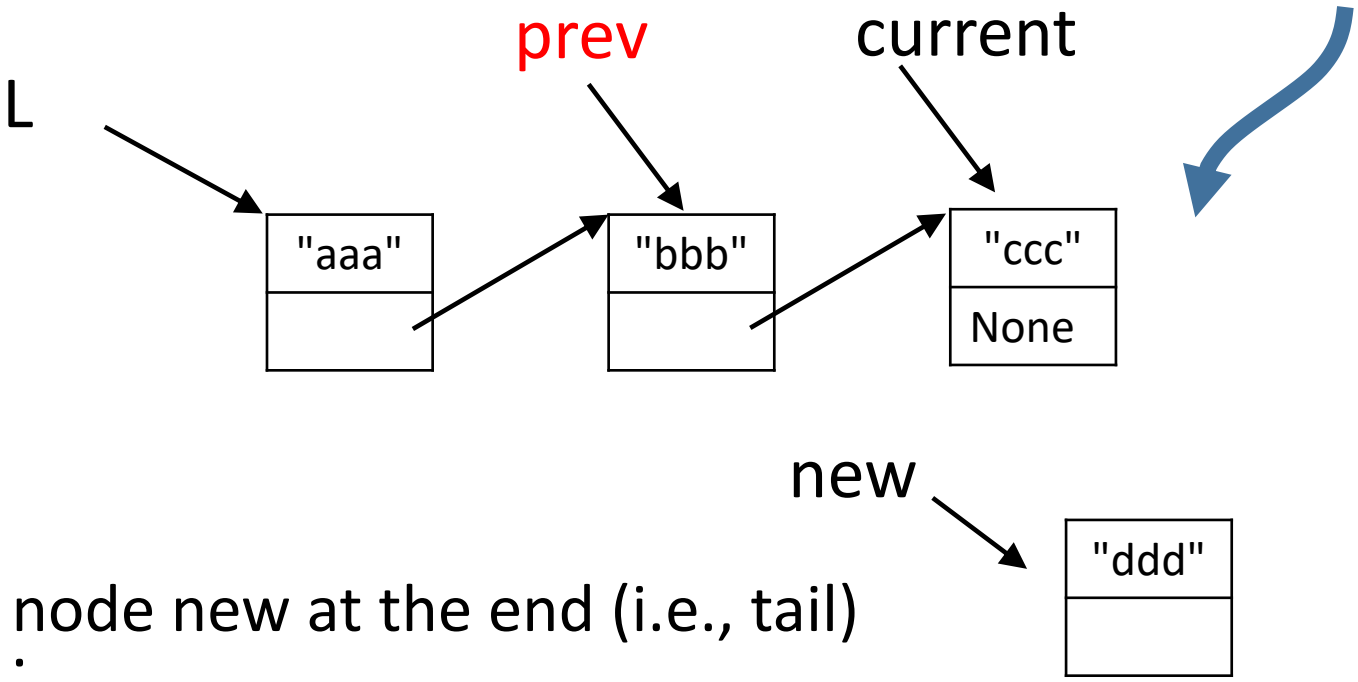

To add a node new at the end (i.e., tail) of a list L:

1. Traverse in a loop

Idea: Keep track of the previous node with another reference

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



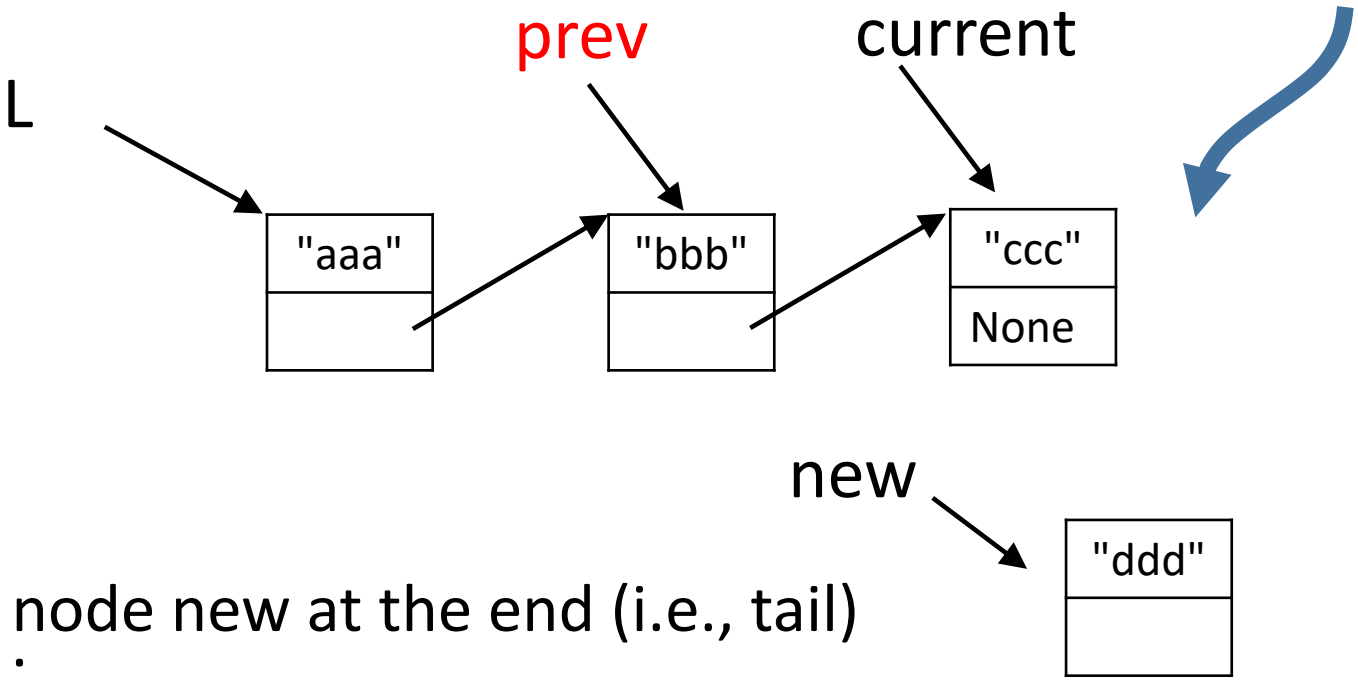

To add a node new at the end (i.e., tail) of a list L:

1. Traverse in a loop

Idea: Keep track of the previous node with another reference.

# Adding a node to the tail

Suppose we want to add a node to the end of a list:



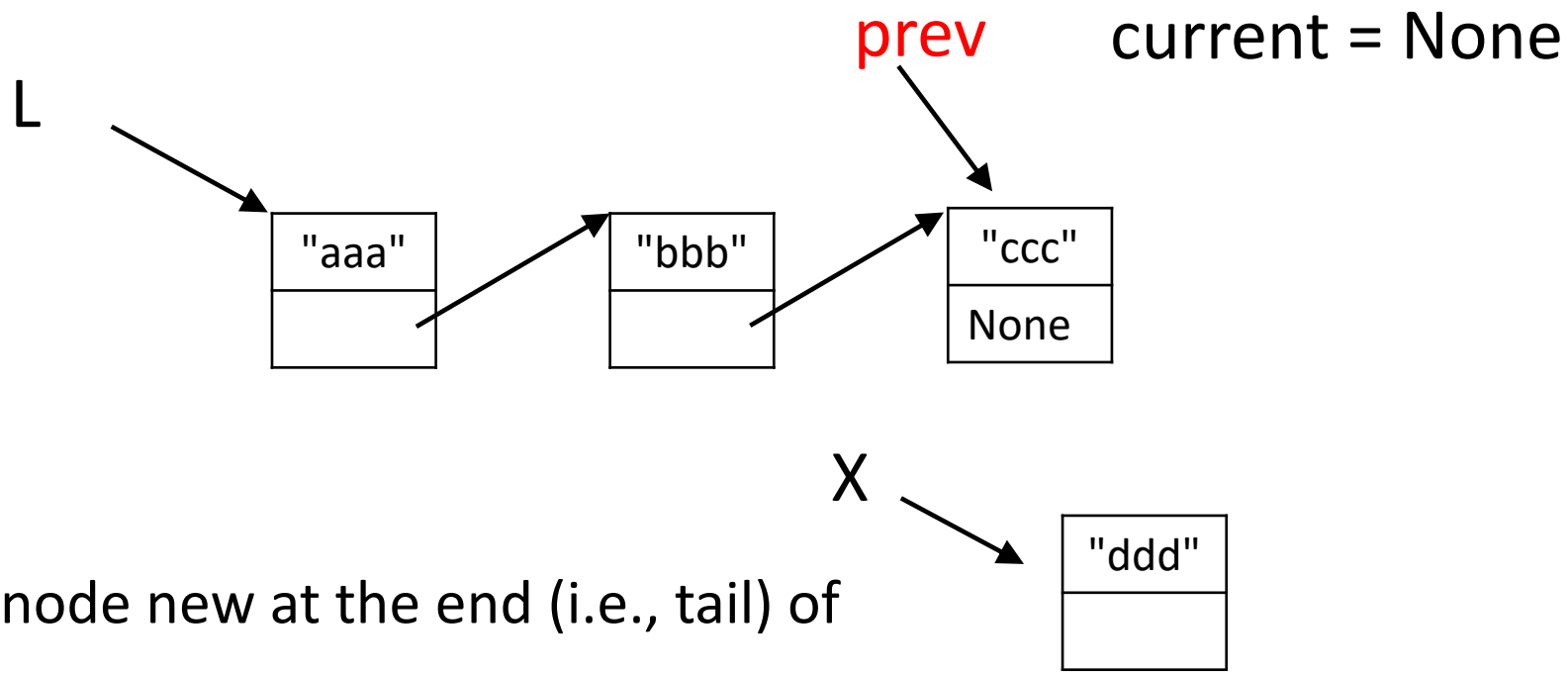

To add a node new at the end (i.e., tail) of a list L:

1. Traverse in a loop

Before advancing current, set prev to current.

# Adding a node to the tail

Suppose we want to add a node to the end of a list:




To add a node new at the end (i.e., tail) of a list L:

1. Traverse in a loop

When current is None, prev is the last node

# Adding to the tail

```
class LinkedList:

    def  add_to_end(self, new):
        current = self._head
        prev = None              # initialize prev
        while current != None:
            prev = current       # keep track of previous node
            current = current._next
        prev._next = new         # add to the end
```

Note: this is a first pass; may need to change it!

# Exercise-ICA-14 prob. 1

Do problem 1 (diagram the process of adding to the tail of a LL).

# Exercise ICA-14 probs. 2-6

- Do the remaining problems (2-6).
- Download the starter code from the class
  - ICA-14-starter.py
  - The code is next to the ICA-14 pdf
- If you leave early, raise your hand and show a TA your solutions before leaving.

# Adding to the tail

```
class LinkedList:

    def  add_to_end(self, new):
        current = self._head
        prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
        prev._next = new      # add to the end
```

Any issues
with this code?

# Adding to the tail

```
class LinkedList:

    def  add_to_end(self, new):
        current = self._head
        prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
        prev._next = new      # add to the end
```

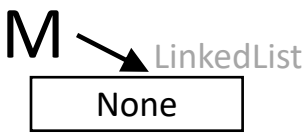
What if the list is empty?

# Adding to the tail

- Make an empty list
- Walk through the code to add a node to M

```
def  add_to_end(self, new):
        current = self._head
        prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
        prev._next = new      # add to the end
```

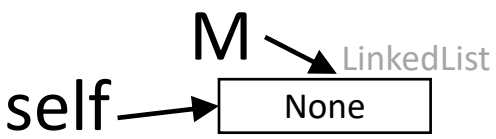
# Adding to the tail




- Make an empty list
- Walk through the code to add a node to M

```
def  add_to_end(self, new):
        current = self._head
        prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
        prev._next = new      # add to the end
```

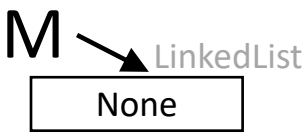
# Adding to the tail




- Make an empty list
- Walk through the code to add a node to M

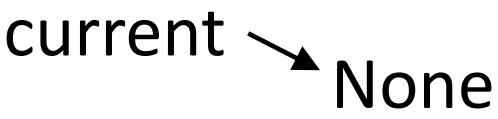
```
def  add_to_end(self, new):
        current = self._head
        prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
        prev._next = new      # add to the end
```

# Adding to the tail

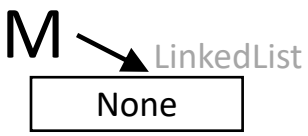



- Make an empty list
- Walk through the code to add a node to M

```
def  add_to_end(self, new):
    current = self._head
    prev = None
    while current != None:
        prev = current      # keep track of previous node
        current = current._next
    prev._next = new        # add to the end
```

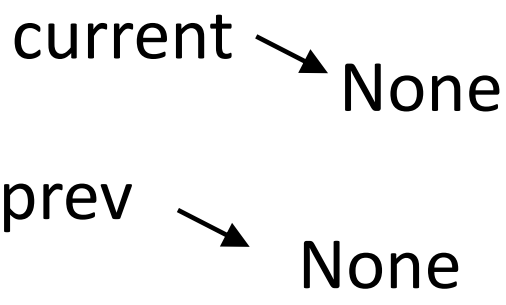


# Adding to the tail

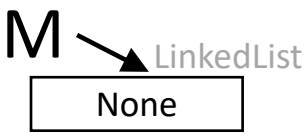



- Make an empty list
- Walk through the code to add a node to M

```
def  add_to_end(self, new):
        current = self._head
→       prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
        prev._next = new        # add to the end
```

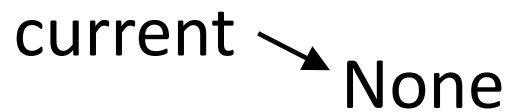


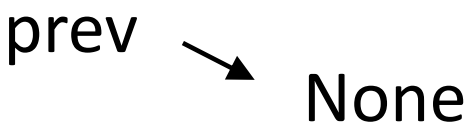
# Adding to the tail




- Make an empty list
- Walk through the code to add a node to M

```
def  add_to_end(self, new):
        current = self._head
        prev = None
        while current != None:
            prev = current      # keep track of previous node
            current = current._next
→       prev._next = new      # add to the end
```




Runtime error on this line above: prev is None!

# Exercise-ICA-15

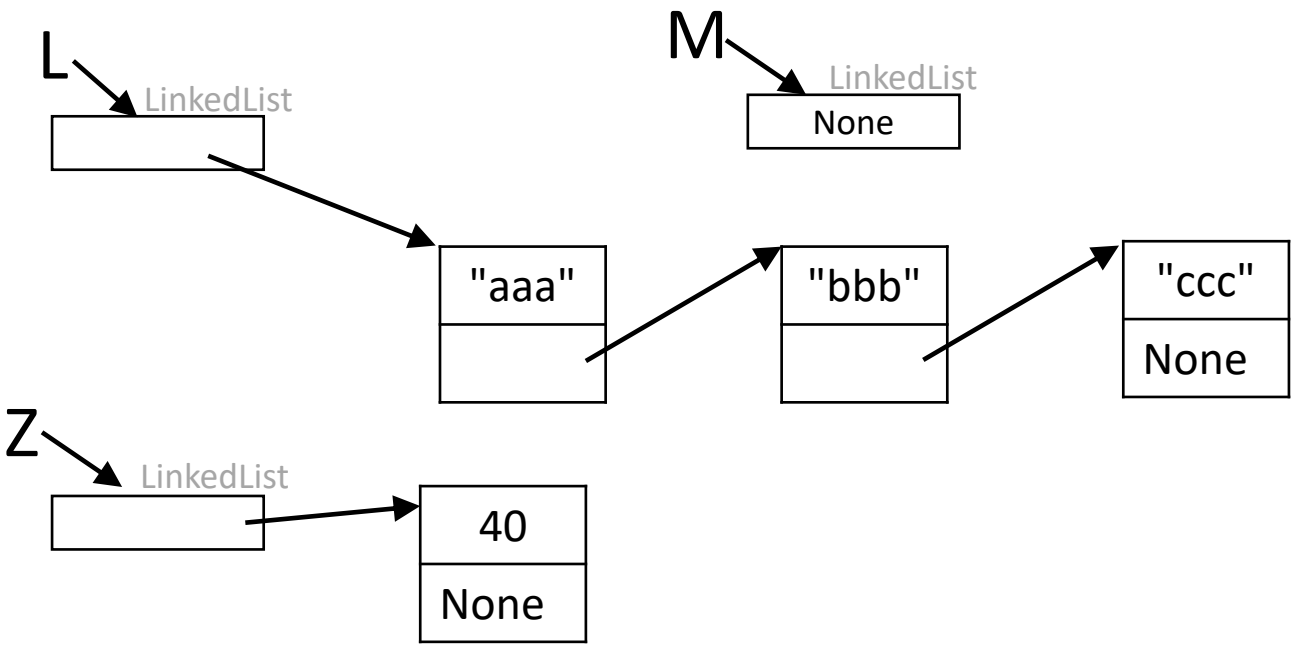
Do problem 1.

(Then we'll continue with lecture.)

# Adding to the tail – corrected

```
class LinkedList:

    def  add_to_end(self, new):
        if self._head == None:        # the list is empty
            self._head = new          # insert new in the front
        else:
            current = self._head
            prev = None
            while current != None:
                prev = current        # keep track of previous node
                current = current._next
            prev._next = new          # add to the end
```

# Cases to consider




Always check the code for all possibilities:

- the list is empty
- the list has one element
- the list has many elements

# Adding to the tail (little brother method)

```
class LinkedList:

    def  add_to_end(self, new):
        if self._head == None:        # the list is empty
            self._head = new          # add new to the front
        else:
            current = self._head
            prev = None               # prev is the little brother
            while current != None:
                prev = current        # always follows current
                current = current._next
            prev._next = new          # add to the end
```

# Adding to the tail (alternative)

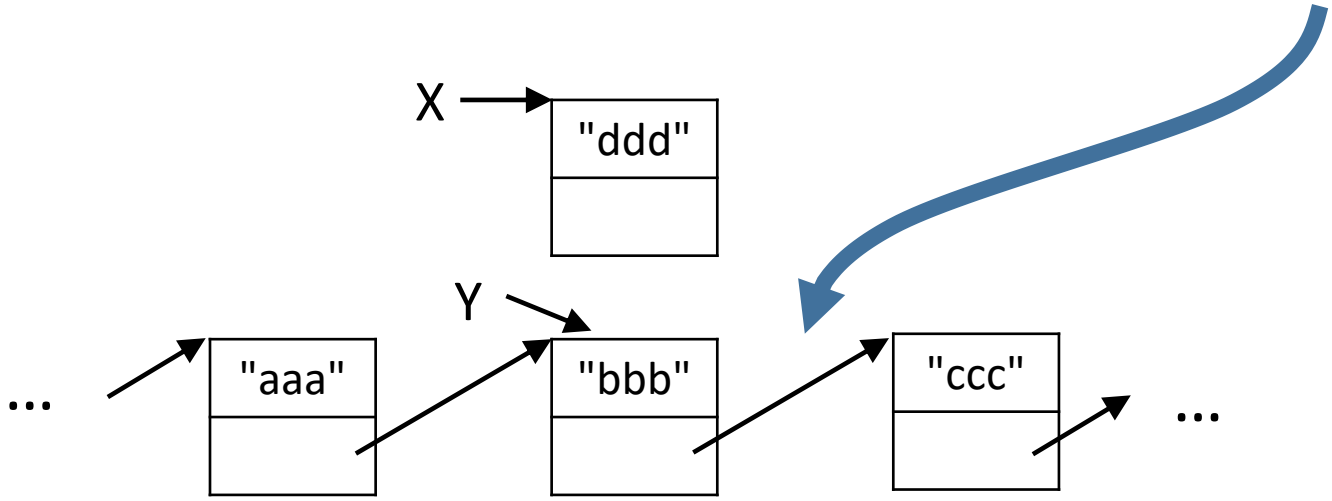
```
class LinkedList:

    def  add_to_end_v2(self, new):
        if self._head == None:        # the list is empty
            self._head = new          # add the new node
        else:
            current = self._head
            while current._next != None:
                current = current._next
            current._next = new       # add to the end
```

Look-ahead method

# insertion

# Inserting a node

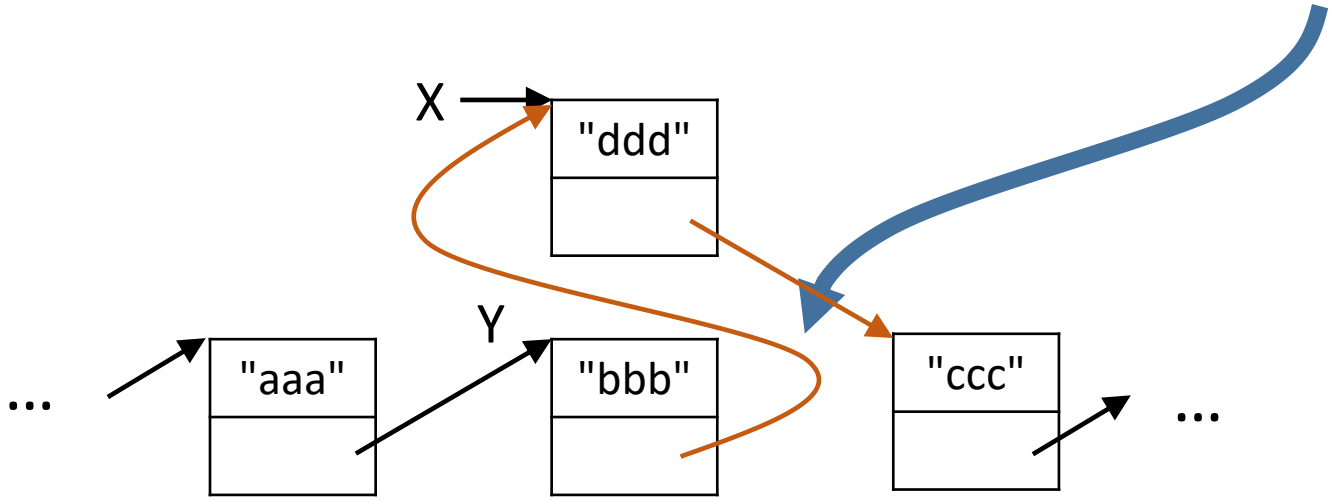
Suppose we want to insert a node X into a list here:




Then we have to adjust the next-node reference on the node Y just before that position

# Inserting a node

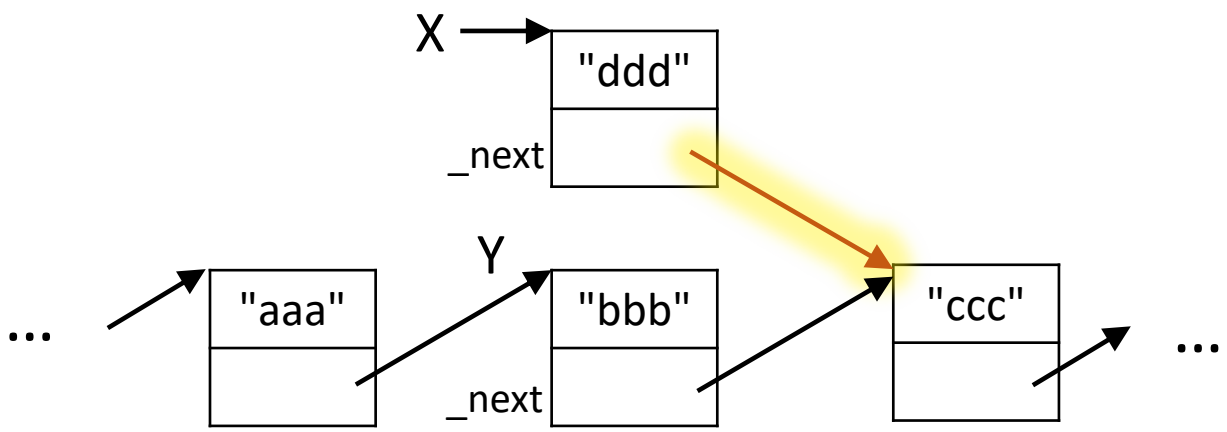
Suppose we want to insert a node X into a list here:




Then we have to adjust the next-node reference on the node Y just before that position

# Inserting a node

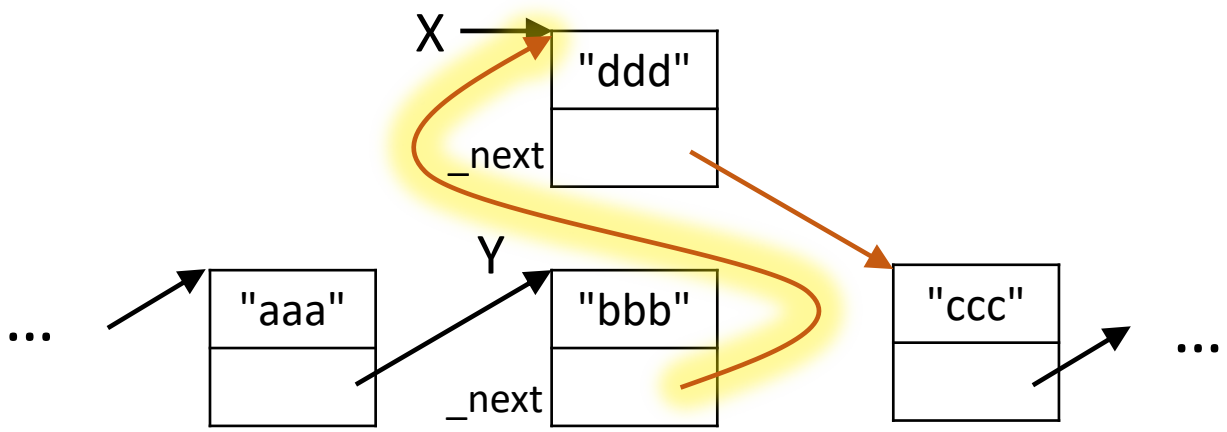
The order of operations is important:




1. X._next = Y._next

# Inserting a node

The order of operations is important:




1. X._next = Y._next
2. Y._next = X

# Inserting a node

Inserting a node X at position *n* in a list L:

1. find the node Y at position *n*−1
   - iterate *n*−1 positions from the head of the list*
2. insert X after Y
   - adjust next-node references as in previous example

```
Y =  L._head
for i in range(n-1):
    Y = Y._next

X._next = Y._next
Y._next = X
```

* would need to know that we have n-1 nodes
better to use a while loop and count

# finding the $n^{th}$ element

# Finding the $n^{th}$ element

```
class LinkedList:

    # return the node at position n of the linked list
    def  get_element(self, n):
        elt = self._head
         while elt != None and n > 0:
            elt = elt._next
            n –= 1
        return elt
```

# Inserting a node

```
class LinkedList:

    # insert a node new at position n
    def  insert(self, new, n):
            if n == 0:
                self.add(new)
            else:
                prev = self.get_element(n−1)
                new._next = prev._next
                prev._next = new
```

# Exercise-ICA-15

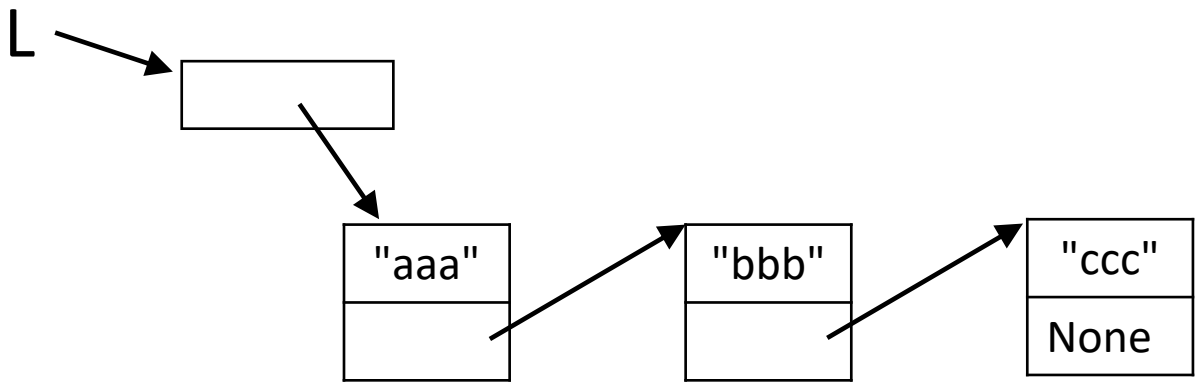
Do problems 2, 3 and 4.

(Then we'll continue with lecture.)

# deletion

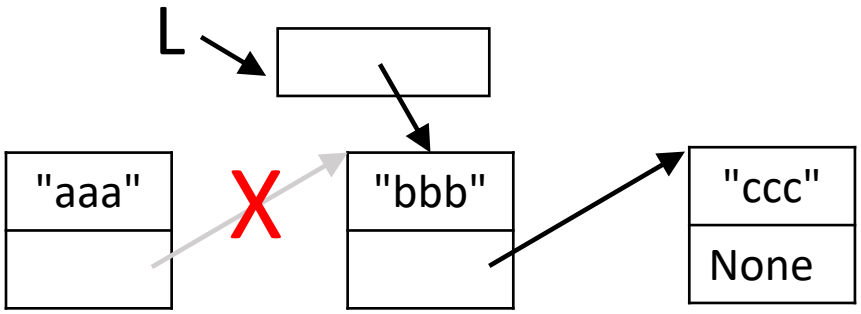
# Remove from the front

Suppose we want to delete the first element



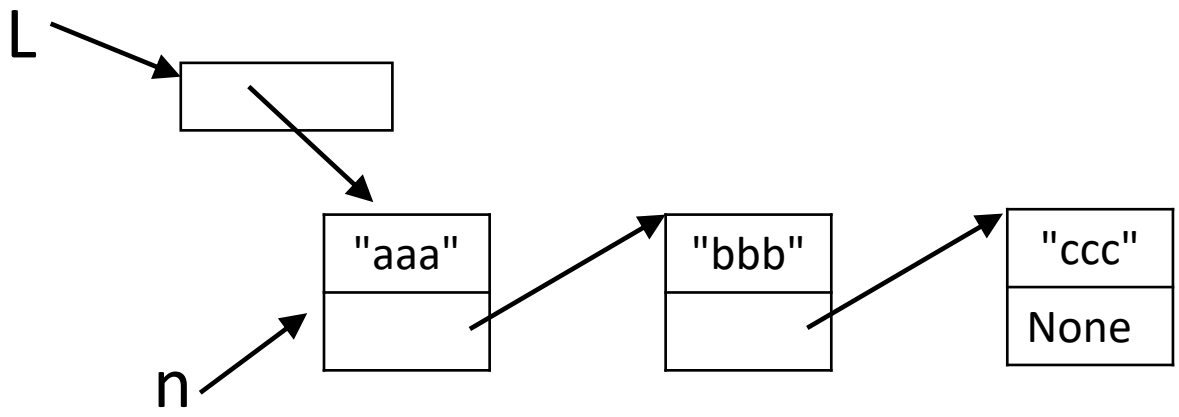
# Removing a node from the front

Suppose we want to delete the first element



# Remove from the front

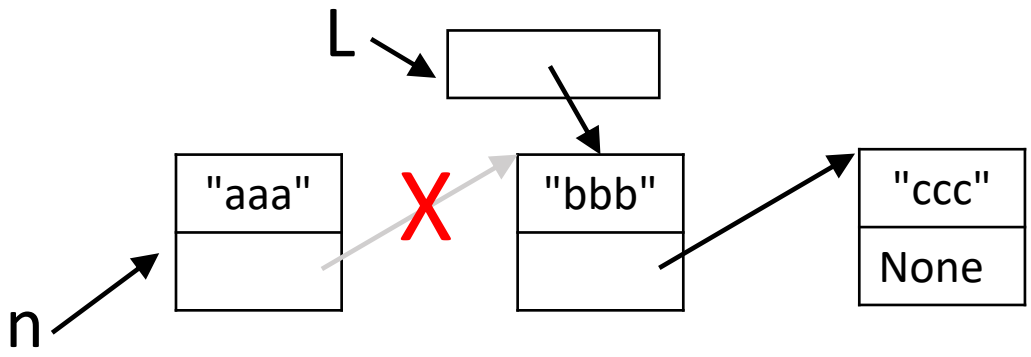
Suppose we want to delete the first element




1. n = L._head
2. L._head = n._next
3. n._next = None

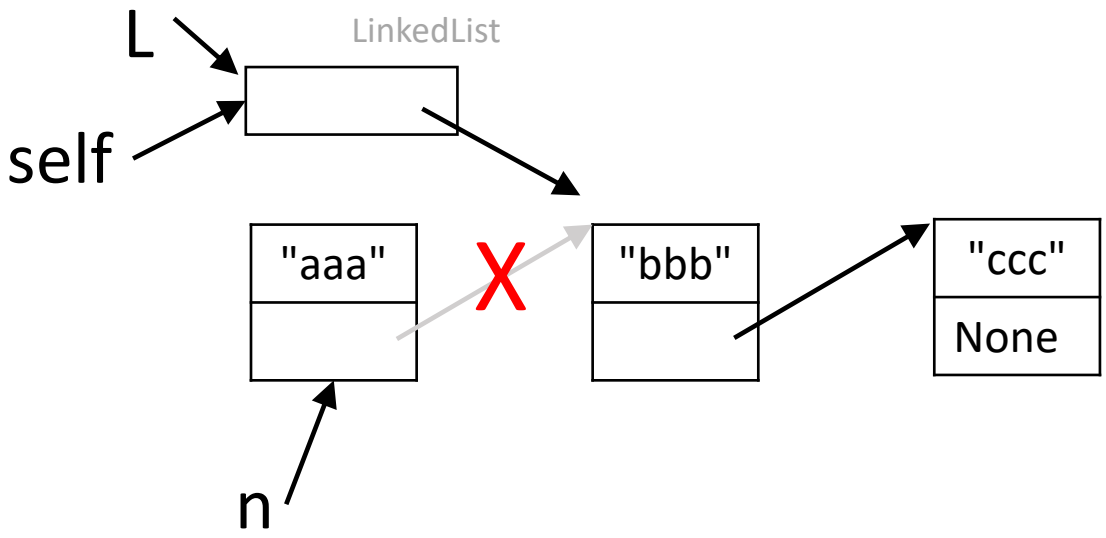
# Removing a node from the front

Suppose we want to delete the first element




1. n = L._head
2. L._head = n._next
3. n._next = None

# Removing a node from the front




In the method, self will refer to the linked list.

# Remove from front

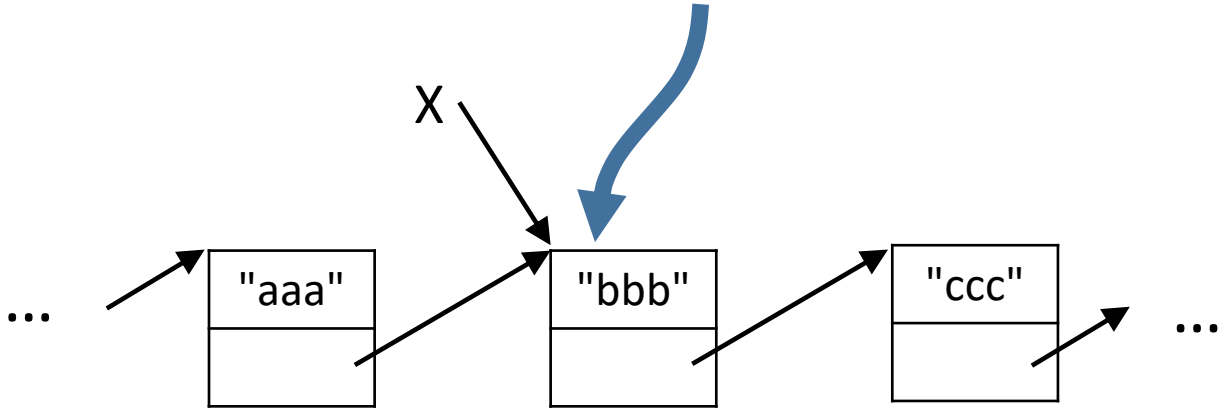
```
class LinkedList:
    def remove_first(self):
        if self._head == None:
            return None
        else:
            n = self._head
            self._head = n._next
            # set the deleted node's next reference to None
            n._next = None
            return n
```

# Remove from front

```
class LinkedList:
    def remove_first(self):
        if self._head == None:      # check for empty list
            return None
        else:
            n = self._head
            self._head = n._next
            # set the deleted node's next reference to None
            n._next = None
            return n
```

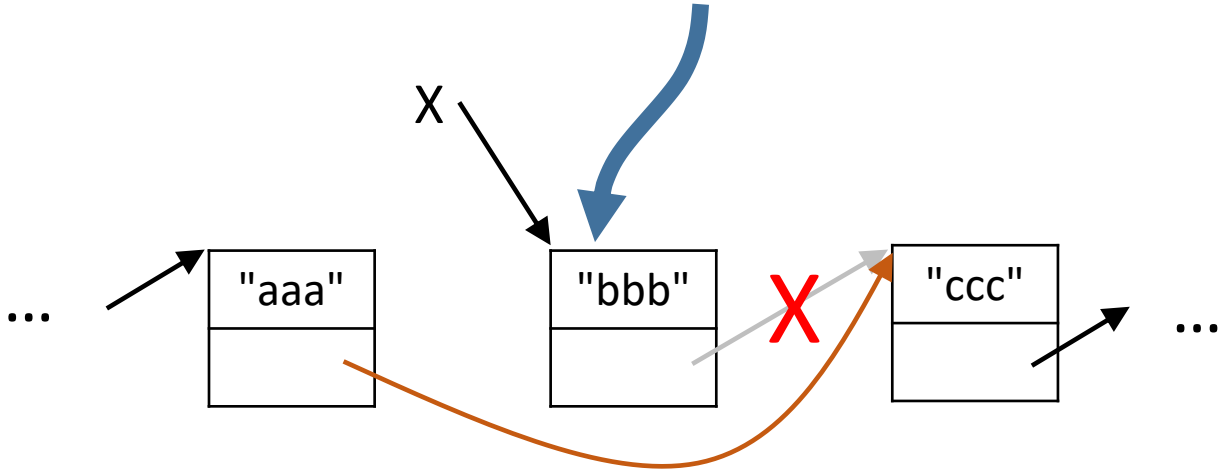
# Deleting a node

Suppose we want to delete this node:



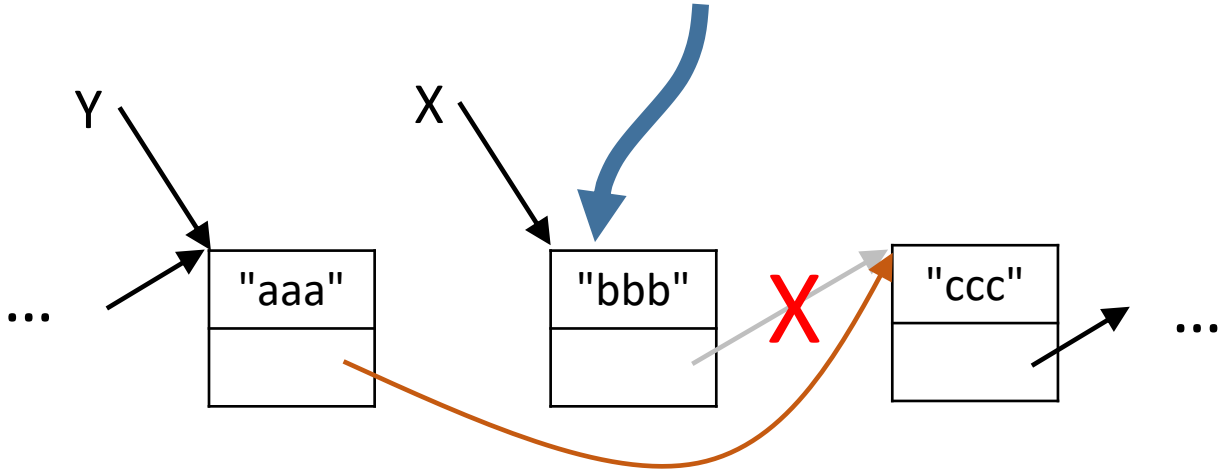
# Deleting a node

Suppose we want to delete this node:



# Deleting a node

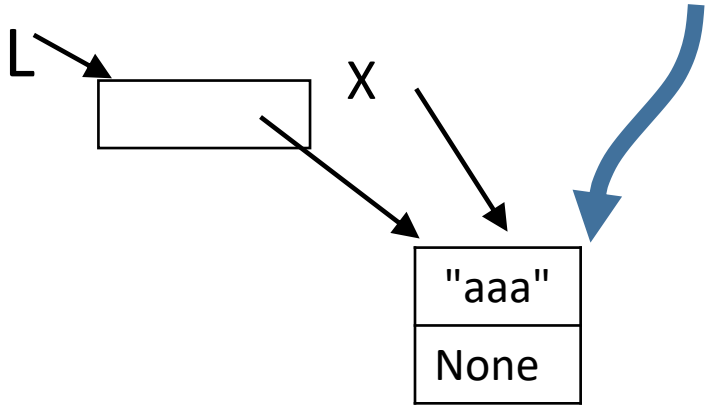
Suppose we want to delete this node:




1. find the node Y just before X (i.e., Y._next == X)
2. Y._next = X._next
3. X._next = None

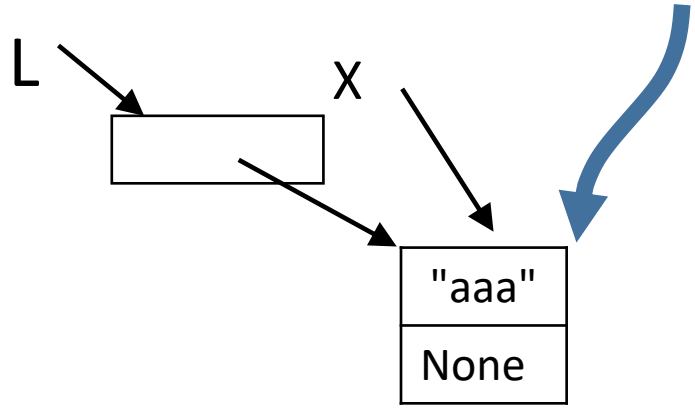
# Deleting a node

What if the list has one element?



# Deleting a node

What if the list has one element?




There is no Y before X.

Must check for that condition first.

# Deleting a node

```
class LinkedList:
    # delete a node X
    def delete(self, X):
        if self._head == X:          # X is the head of the list
            self._head = X._next
        else:
            Y = self._head
            while Y._next != X:      #  look-ahead method
                Y = Y._next
            Y._next = X._next
        X._next = None
```

# concatenation

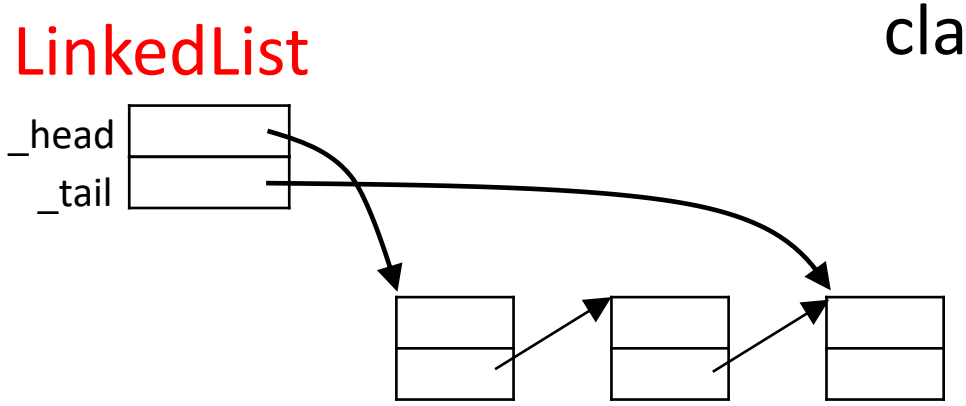
# Concatenating two linked lists

```
class LinkedList:
    # concatenate list2 at the end of the list
    def concat(self, list2):
        if self._head == None:      # list is empty
            self._head = list2._head
        else:
            current = self._head
            while current._next != None:
                current = current._next
            current._next = list2._head
```

# maintaining a tail reference

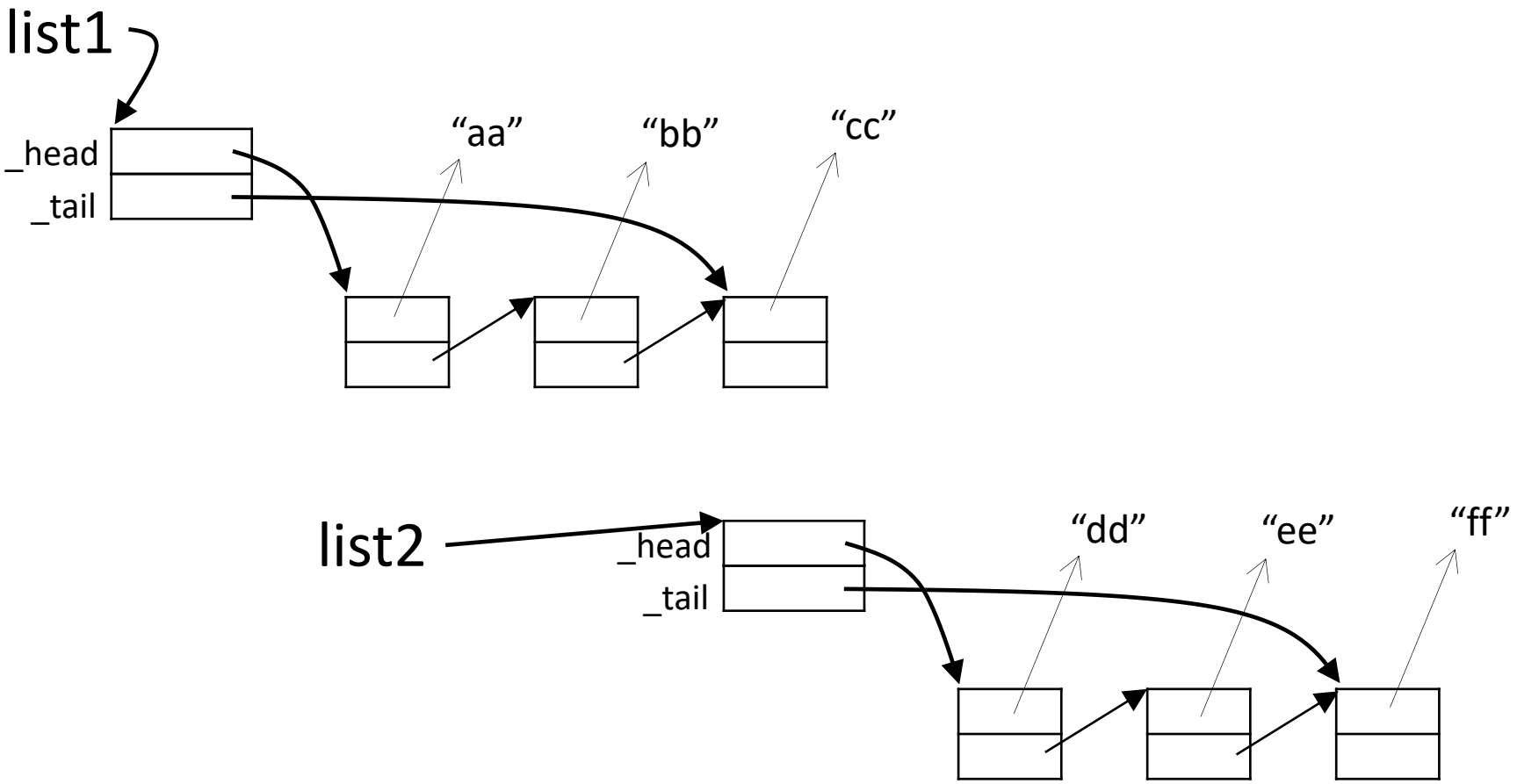
# Maintaining a tail reference

A variation is to also maintain a reference to the tail of the list

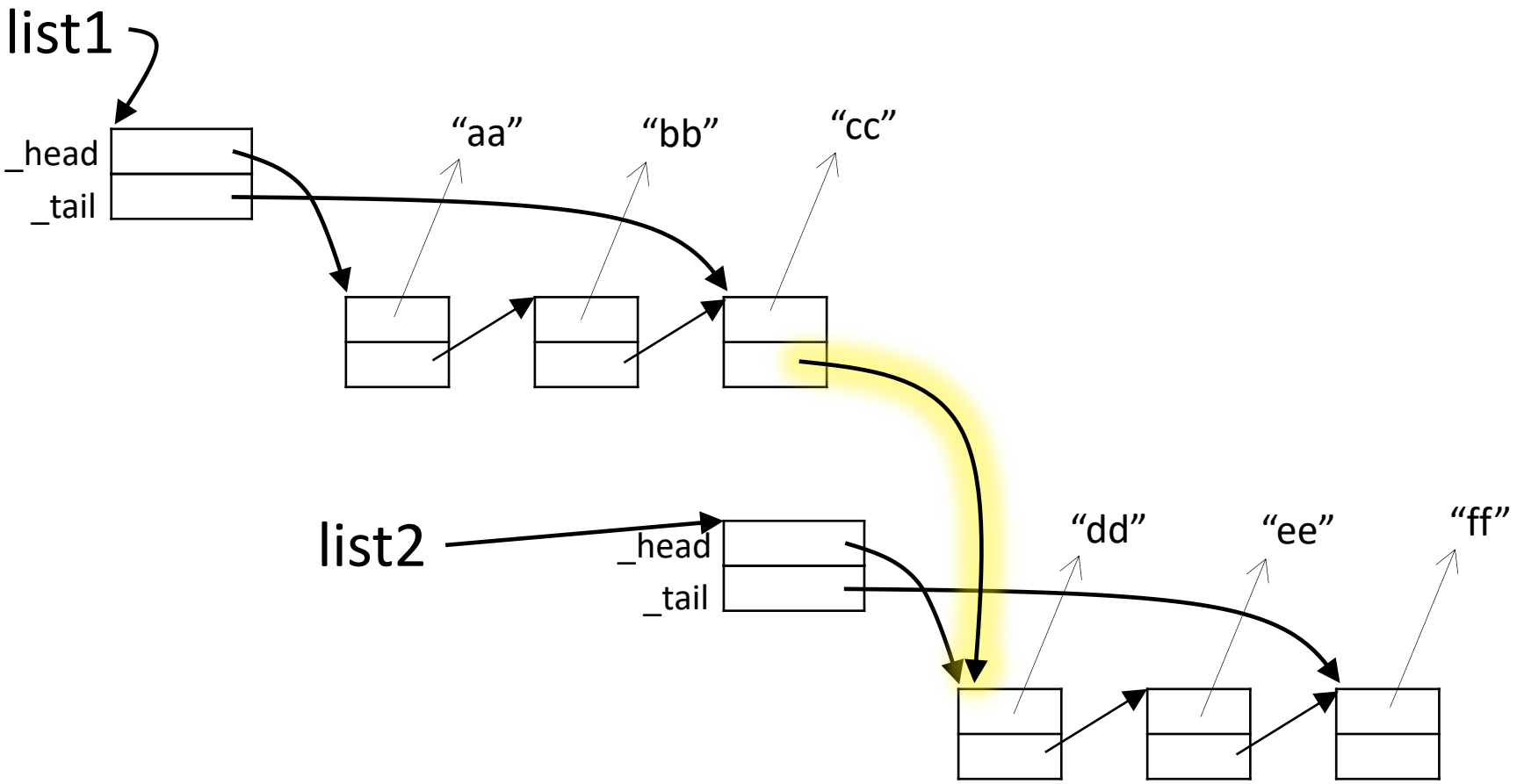



```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
```

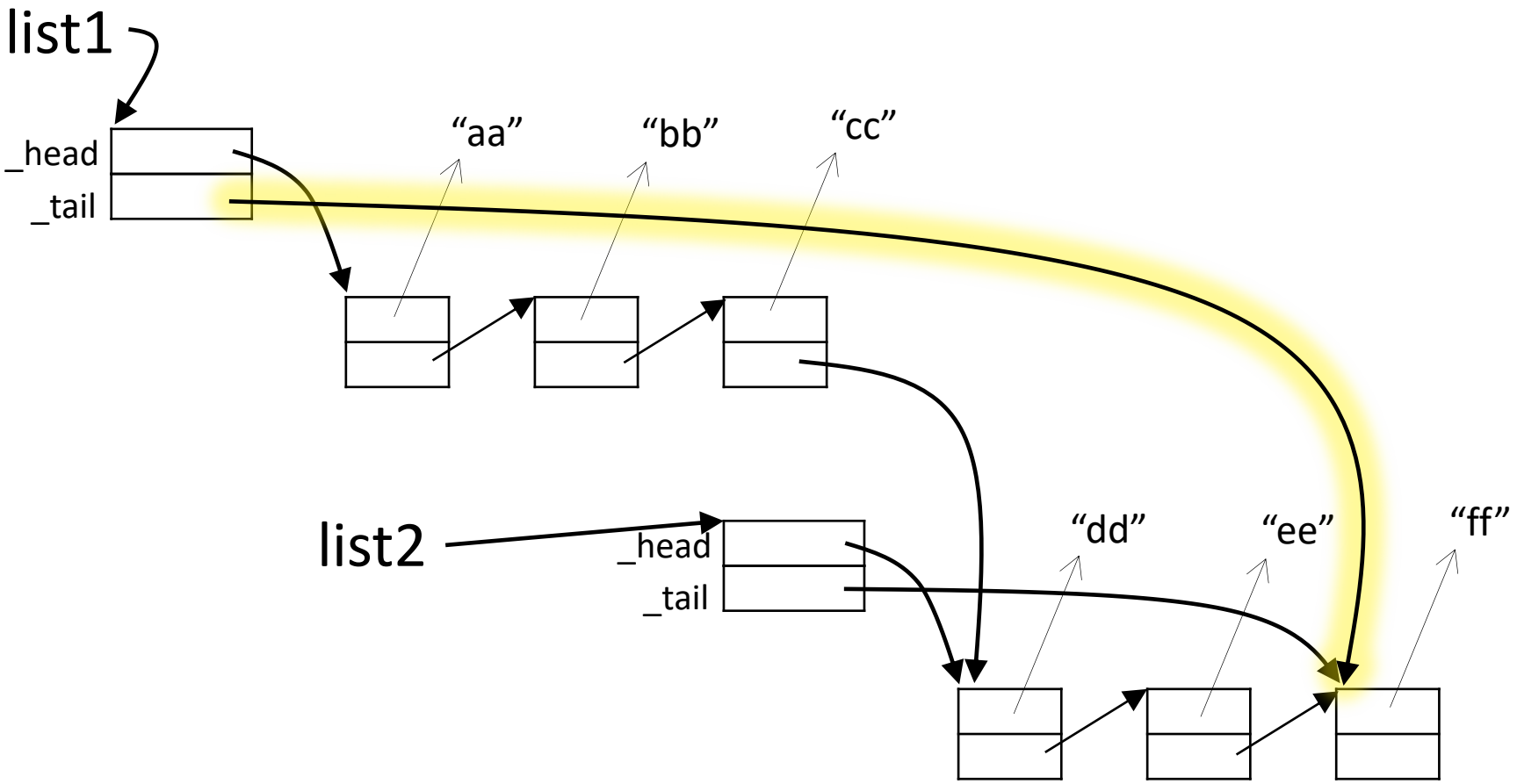
# Tail references and concatenation



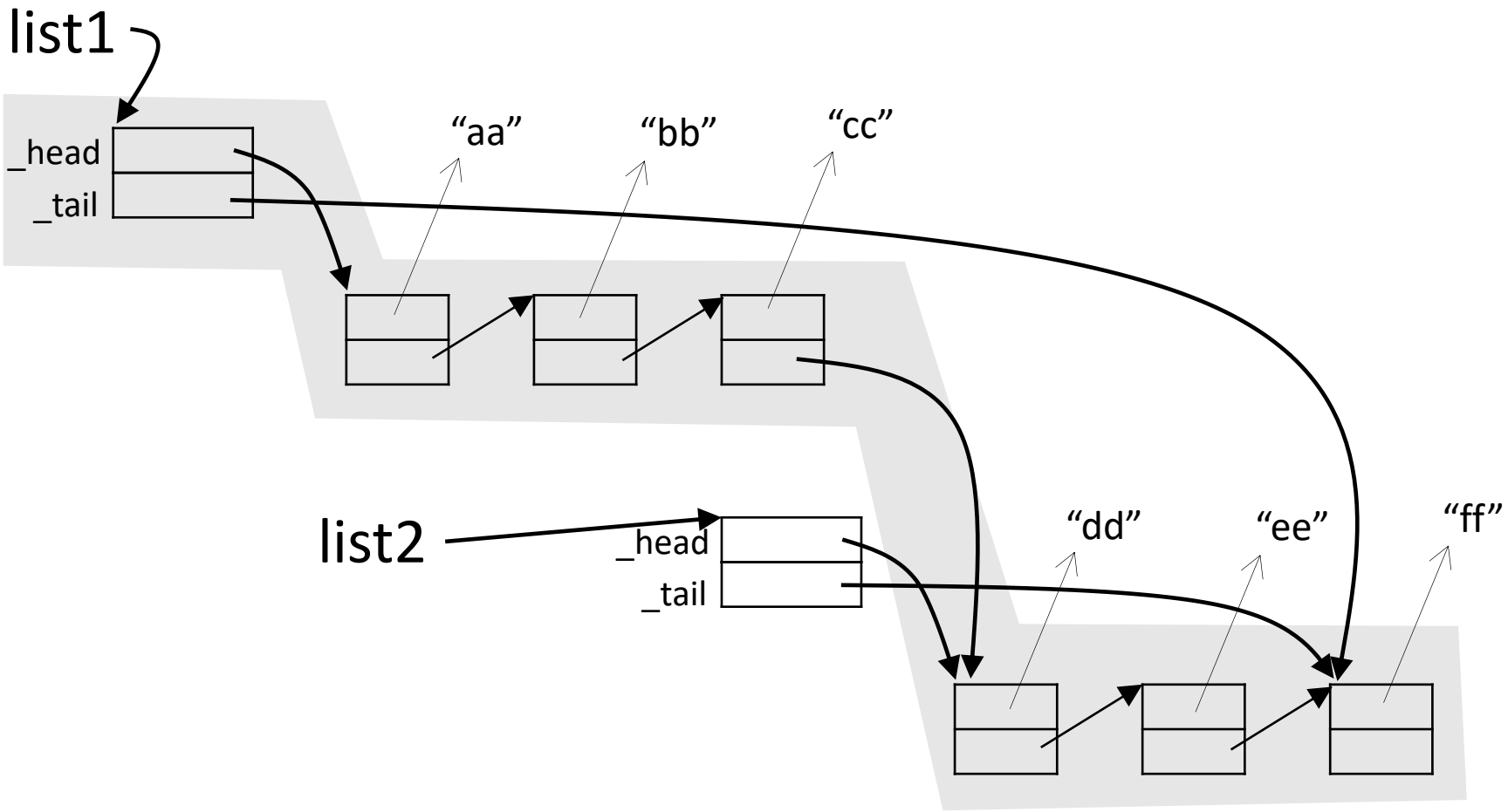
# Tail references and concatenation



# Tail references and concatenation



# Tail references and concatenation



# Maintaining a tail reference

- Concatenation and append become efficient:

```
def concat(self, list2):
    if self._head == None:
        self._head = list2._head
        self._tail = list2._tail
    else:
        self._tail._next = list2._head
        self._tail = list2._tail
```

- All linked list operations must now make sure that the tail reference is kept properly updated

# Exercise-ICA 16

Do problem 1.