CSc 120
Introduction to Computer Programming II

Lists (arrays) vs. Linked Lists: Complexity

performance puzzler

Example: list insert vs append

insert: adds an element into the middle of a list

```
>>> list0 = [1,2,3,4]
>>> list0
[1, 2, 3, 4]
>>> list0.insert(2, 'aaa')
>>> list0
[1, 2, 'aaa', 3, 4]
>>> list0.insert(3, 'bbb')
>>> list0
[1, 2, 'aaa', 'bbb', 3, 4]
>>>
```

append: adds an element at the end of a list

```
>>> list0 = [1,2,3,4]
>>> list0
[1, 2, 3, 4]
>>> list0.append('aaa')
>>> list0
[1, 2, 3, 4, 'aaa']
>>> list0.append('bbb')
>>> list0
[1, 2, 3, 4, 'aaa', 'bbb']
>>>
```

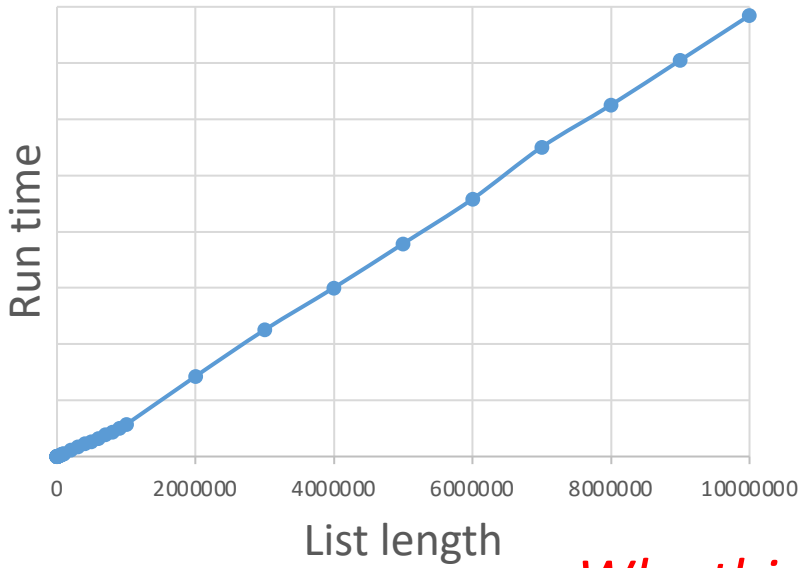
Example: list insert vs append

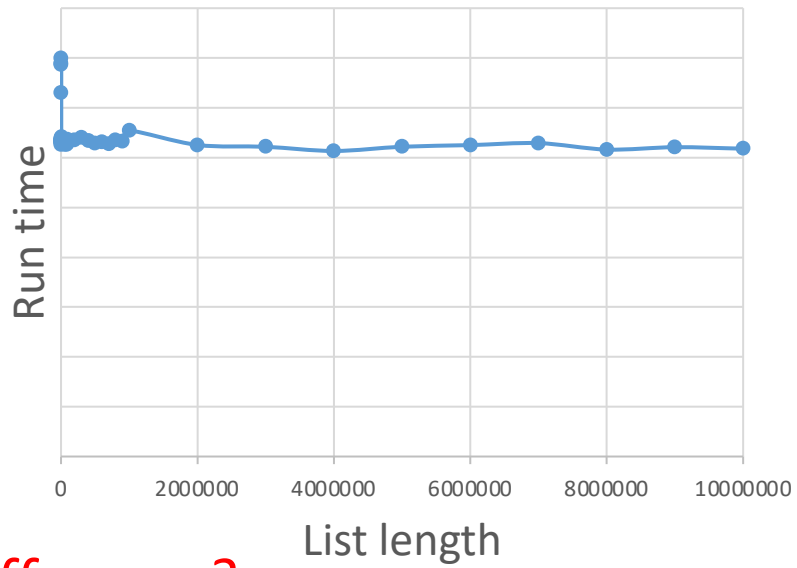
insert: adds an element into the middle of a list

```
list0 = mklist(n)        # length of list0 == n
list0.insert(n//2, 0)    # insert at midpoint
```

append: adds an element at the end of a list

```
list0 = mklist(n)
list0.append(0)  # add at the end
```



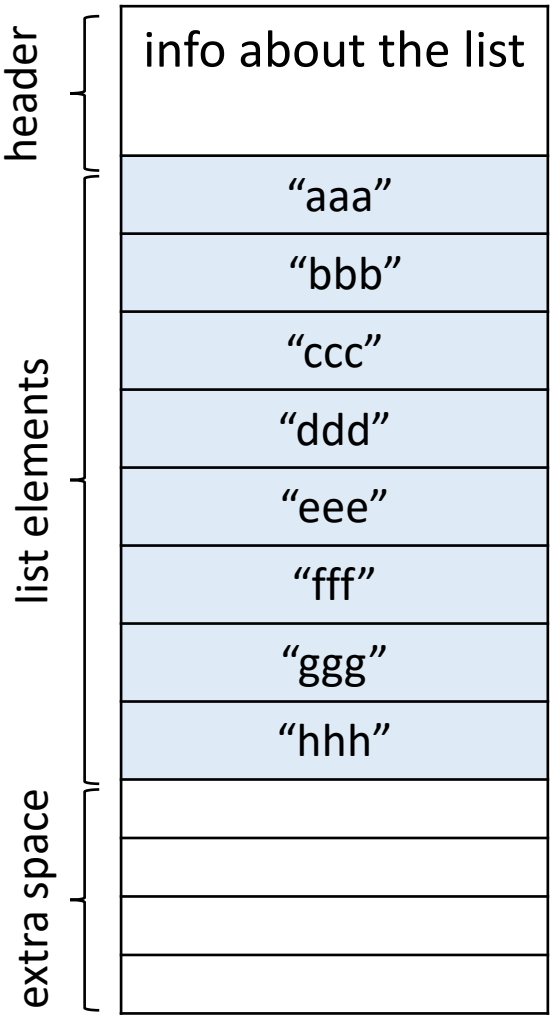


Why this difference?

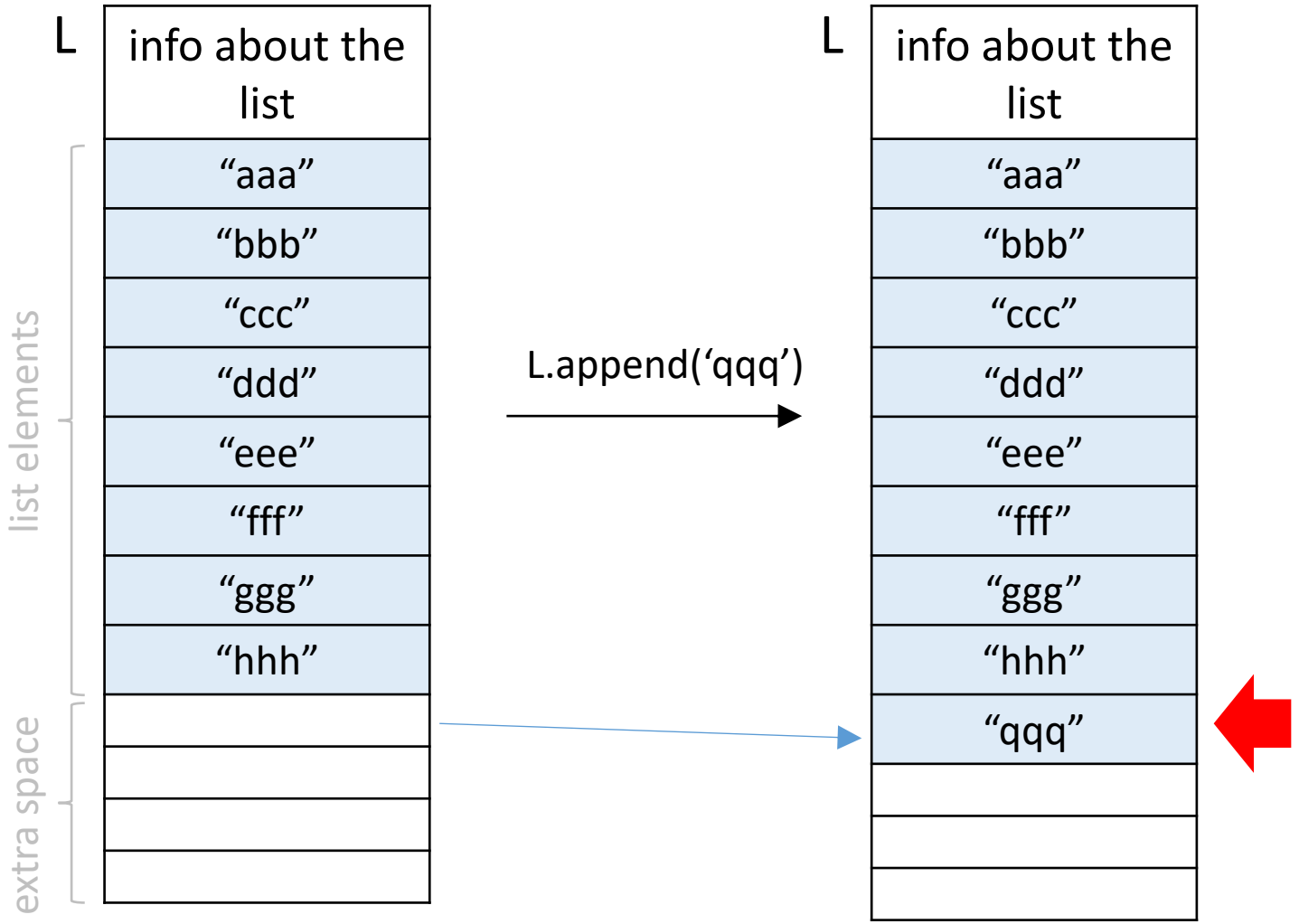
insert vs. append

List (array) organization in Python

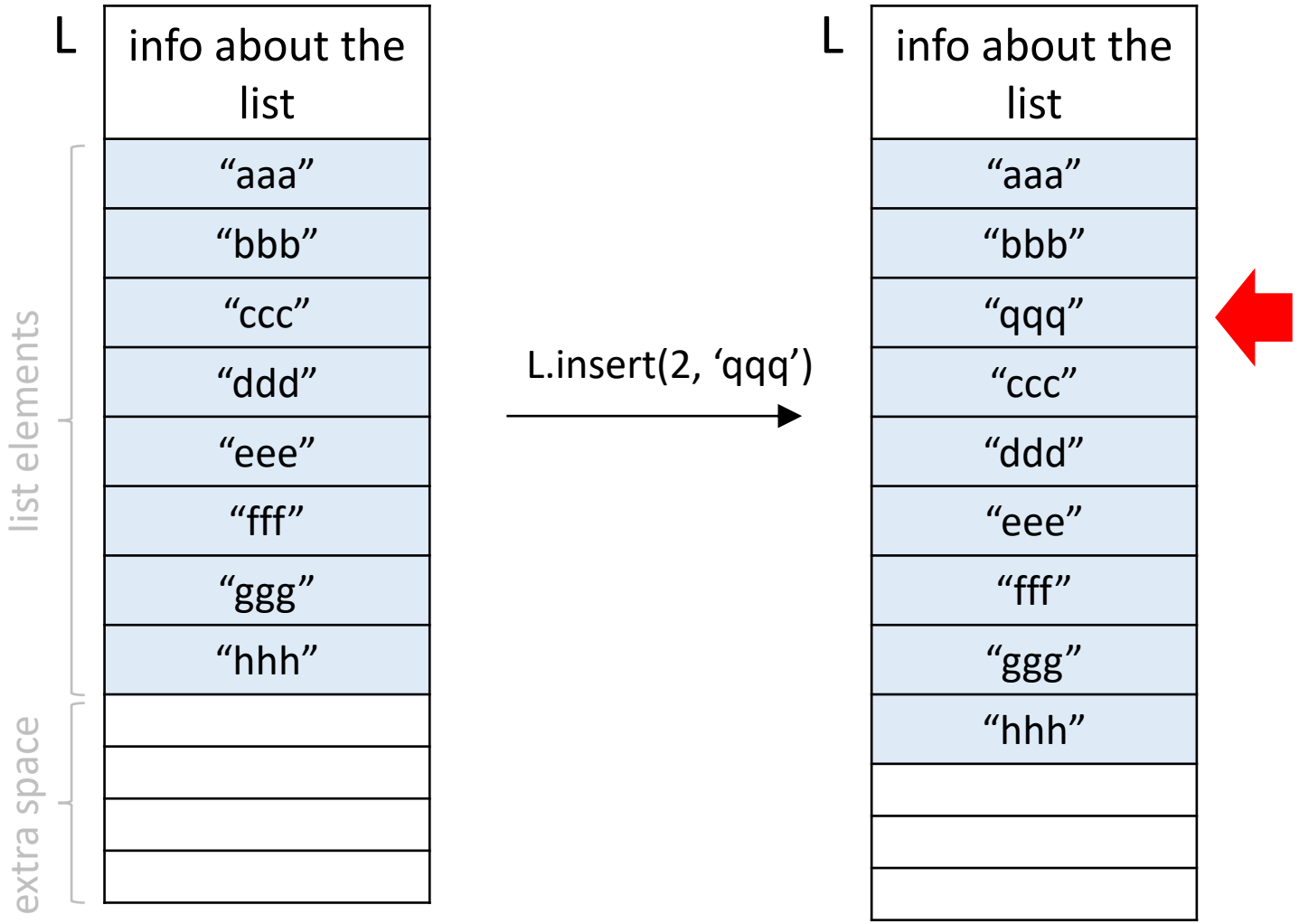
- (References to) the list elements are kept in a contiguous sequence of memory words
  - there is a little extra space at the end to give it some room to grow
- The following operations are O(1):
  - len()
    - read off length info from the header
  - accessing the i^{th} element of the list
    - compute its address using the value of i
    - access memory location at that address



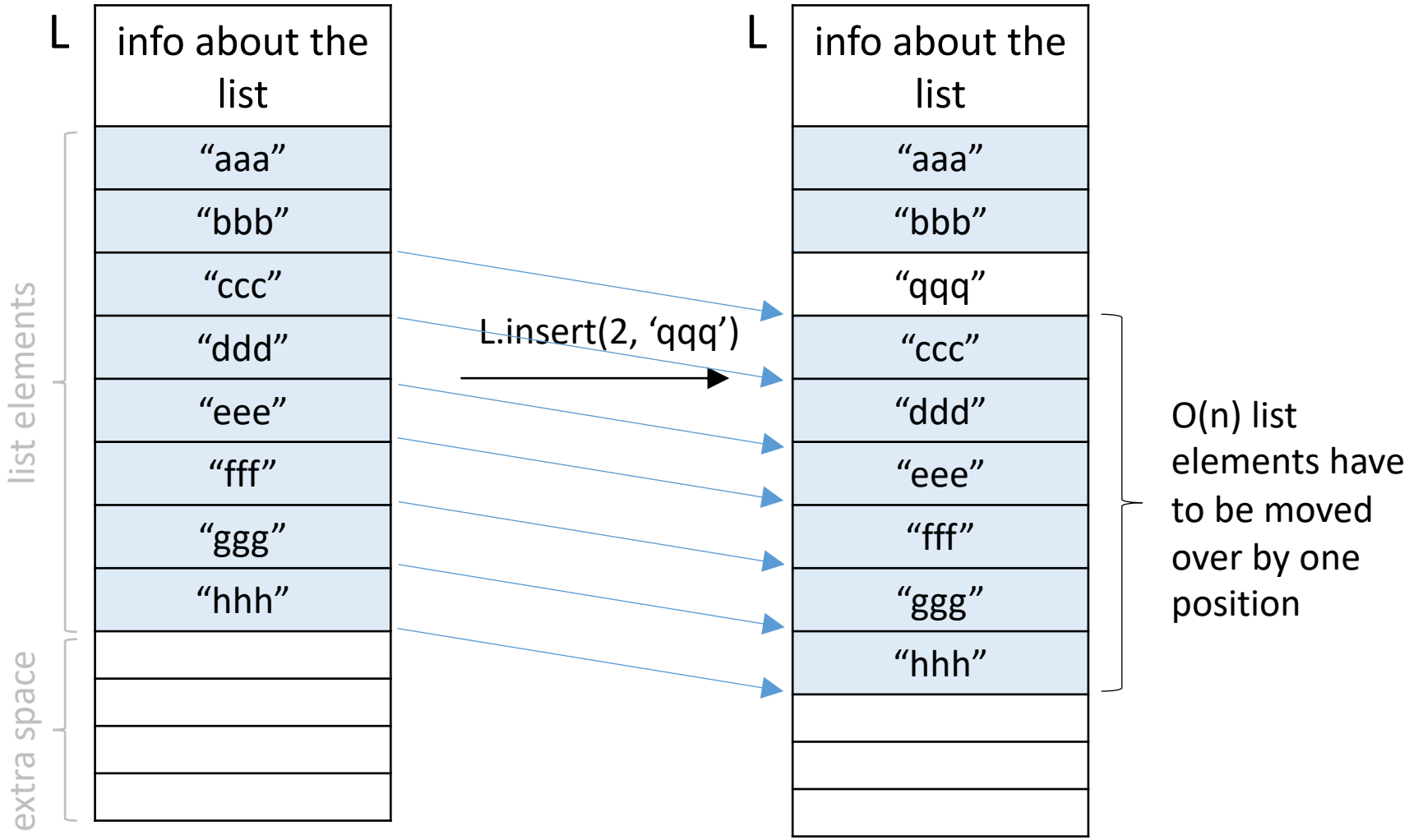
Appending to a list O(1)



Inserting into a list



Inserting into a list O(n)



Python lists: complexity summary

Operation	Complexity
len	O(1)
access an element's value	O(1)
append	O(1)
insert, delete	O(n)

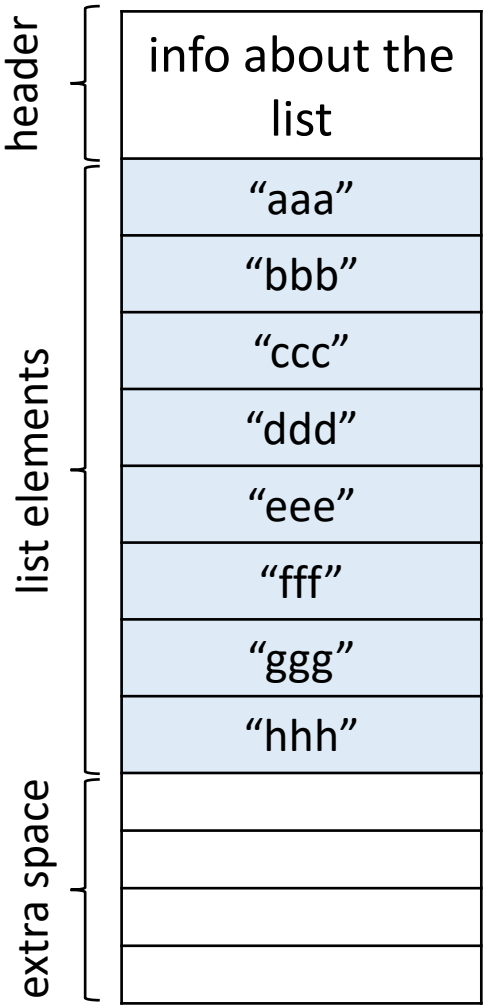
Q: Can we do insert in O(1) time?

(The complexity of other operations may change)

Exercise-ICA 31 Prob. 1&2

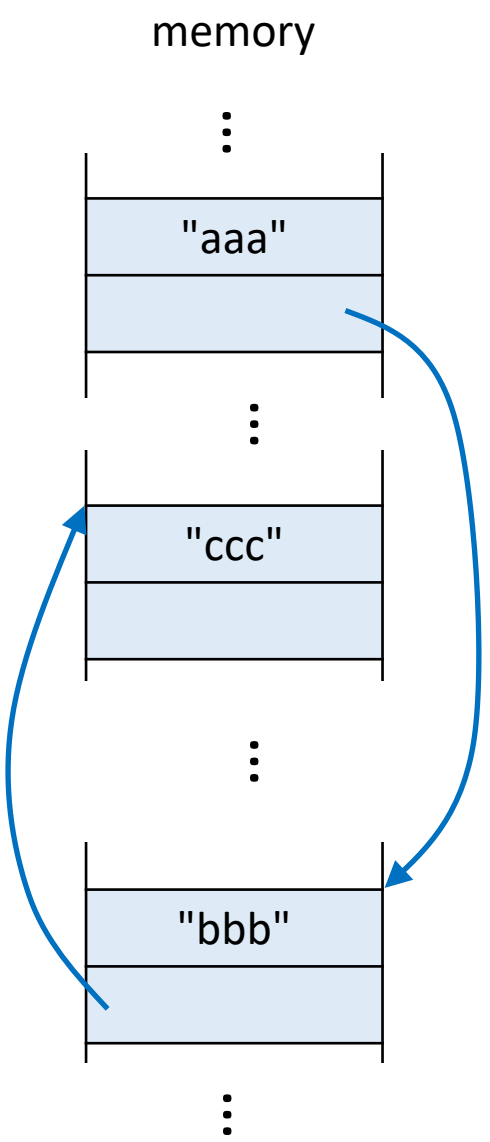
Python lists: reprise

- Key feature: L[i] and L[i+1] are adjacent in memory
- This makes accessing L[i] very efficient
  - O(1)
- Insertion and concatenation require moving O(n) elements
  - O(n)



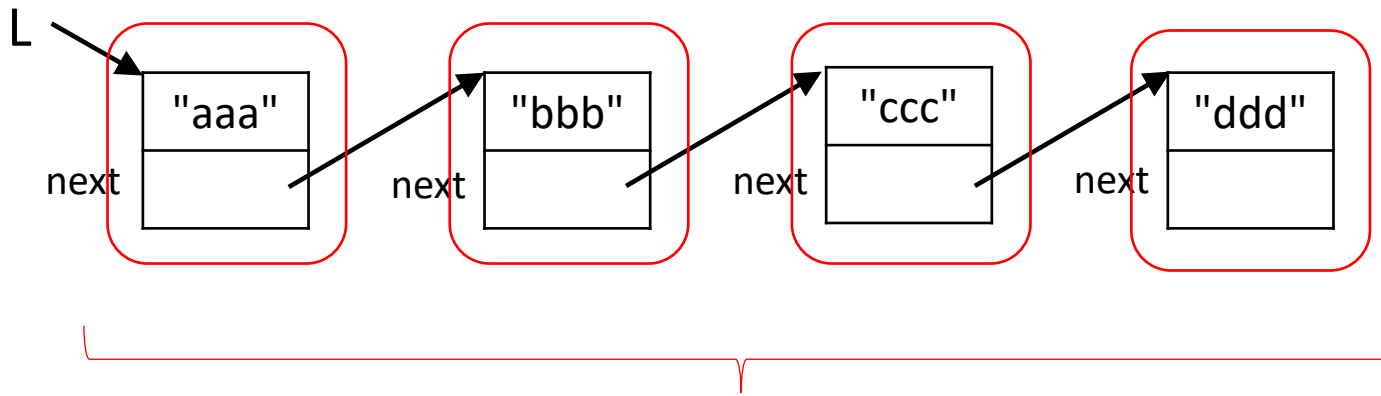
Linked lists: reprise

- To get O(1) insertion and concatenation, we cannot afford to move O(n) 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

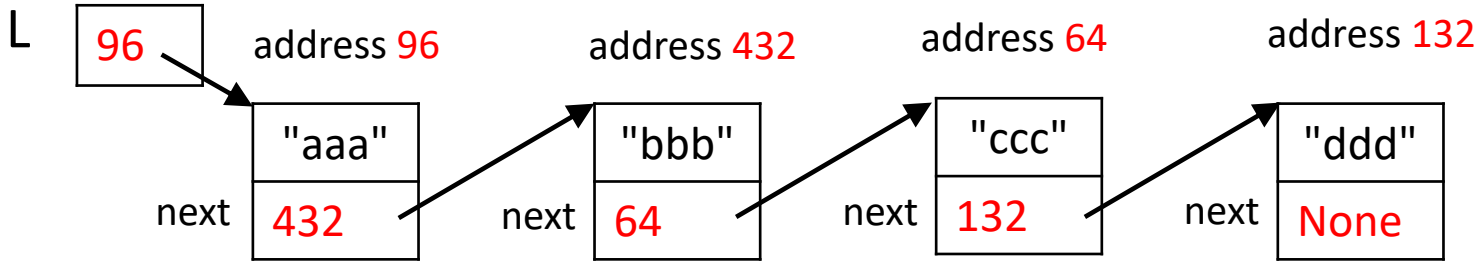
With each element of the list, keep a reference to the next list element



Linked lists

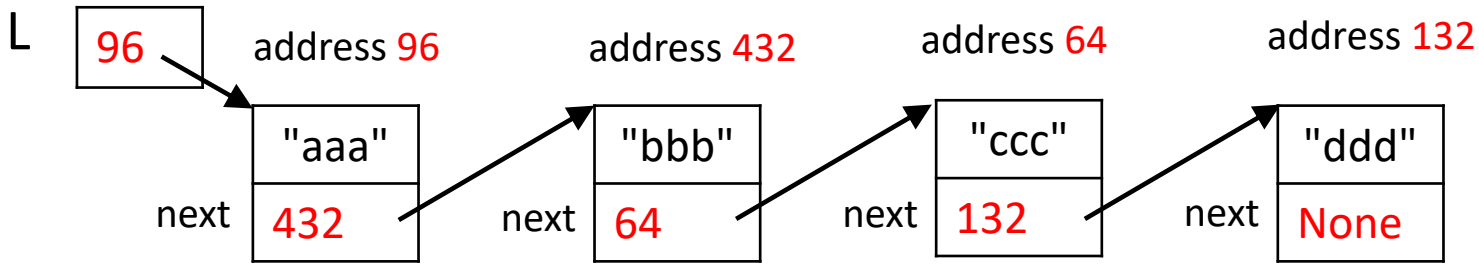
References are addresses in memory.

Here is the diagram with explicit addresses (simplified).



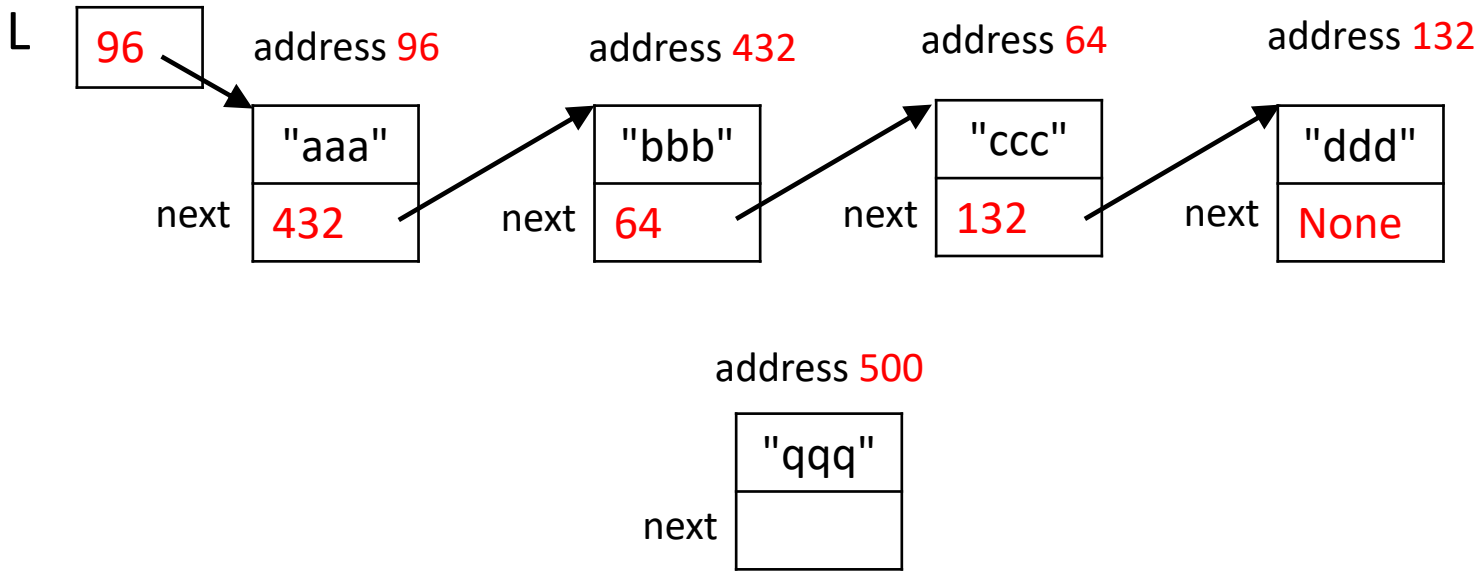
Insertion

Consider inserting a new node into the linked list



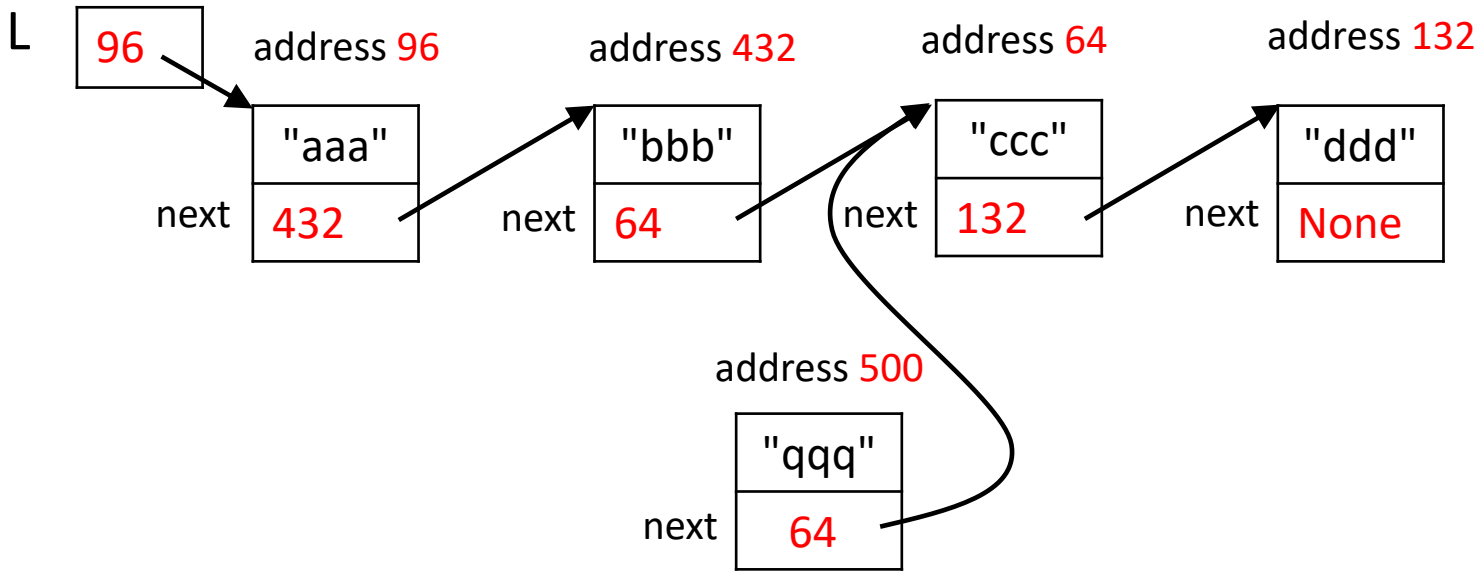
Insertion

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



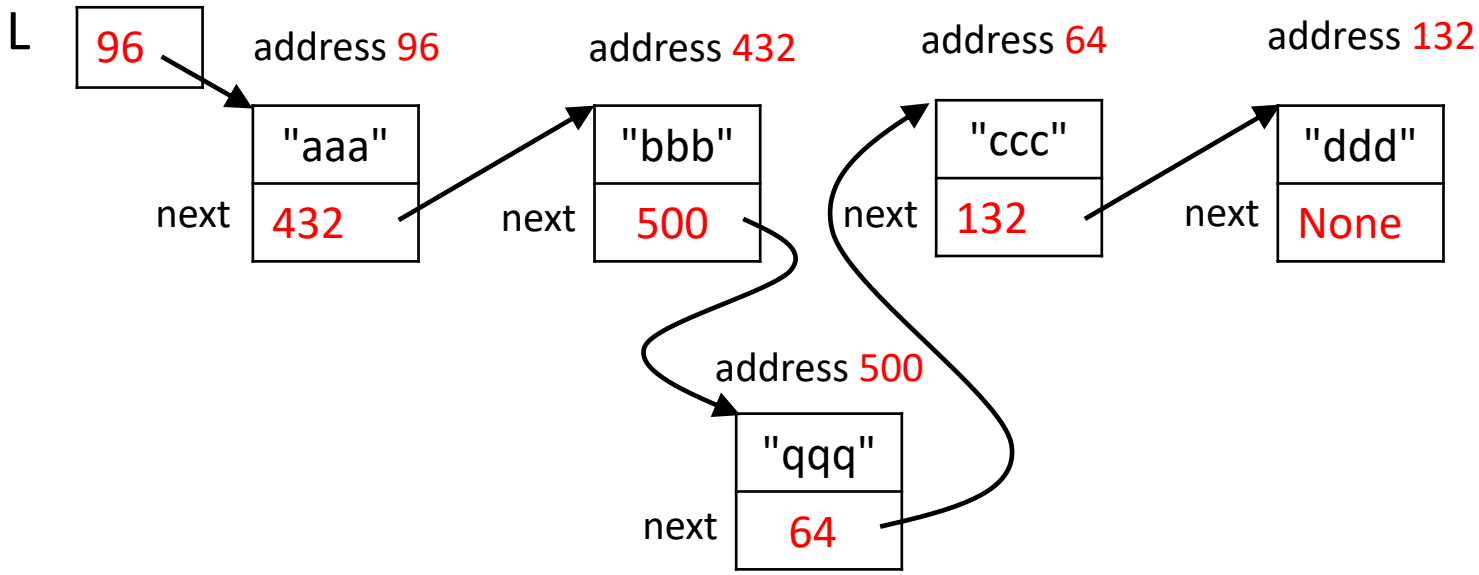
Insertion

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



Insertion

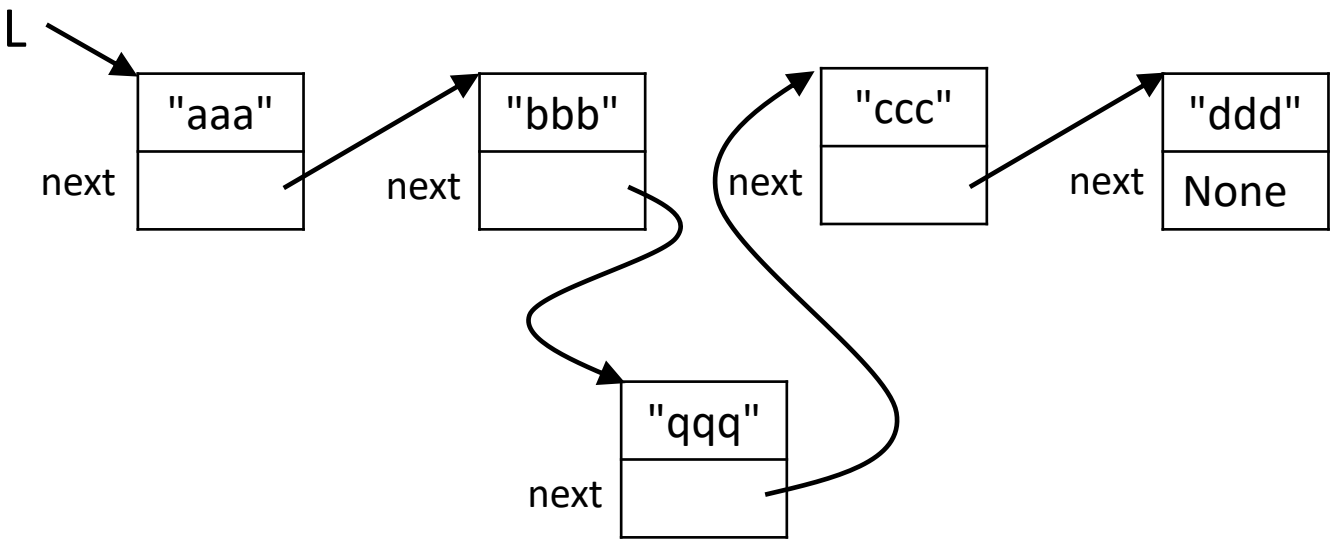
We want to add a new node between "bbb" and "ccc". What do we change?



Insertion

O(1)*

Set the next references appropriately. What is the complexity of insertion?

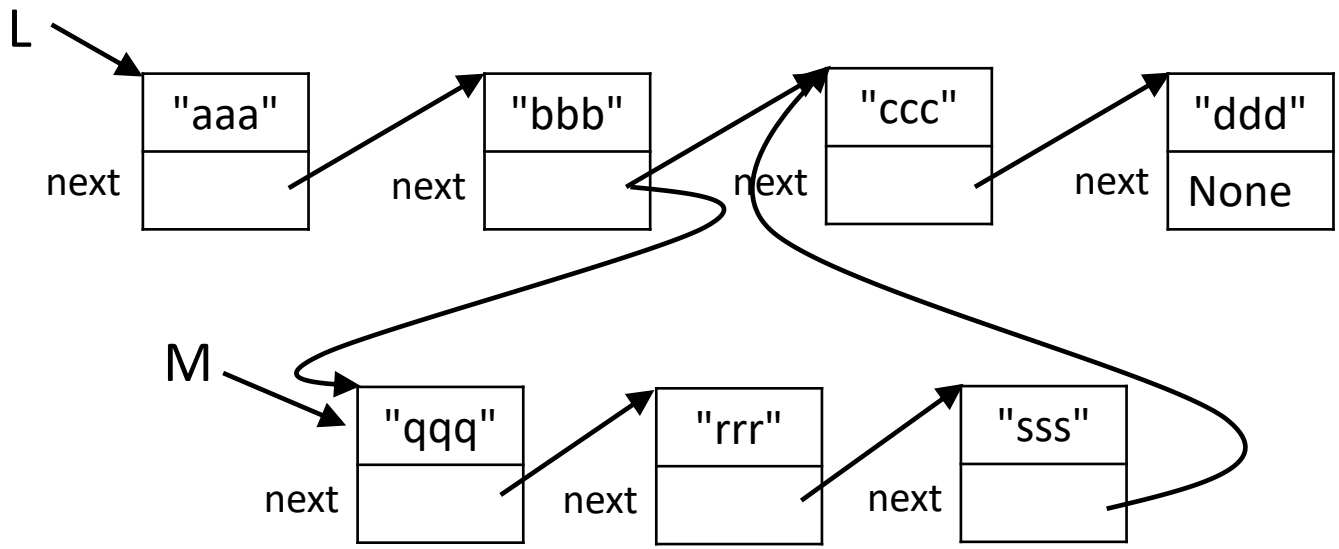


*assuming we have a reference to the node of insertion

Insertion

O(1)

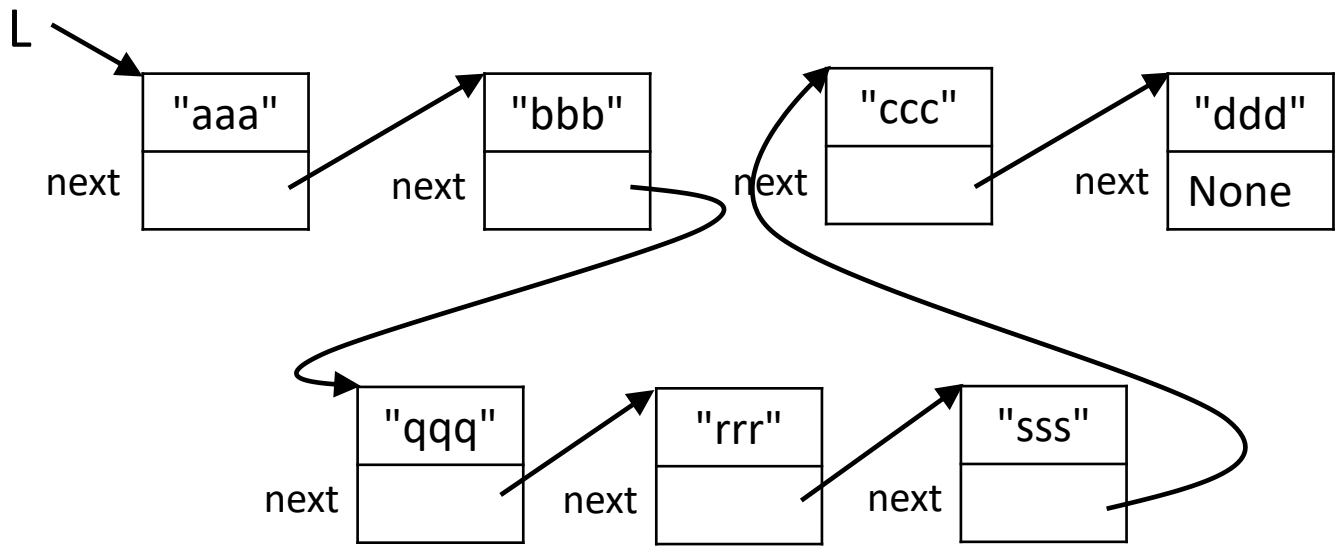
To insert an element (which can be a linked list) into a linked list: set next references appropriately



Insertion

O(1)

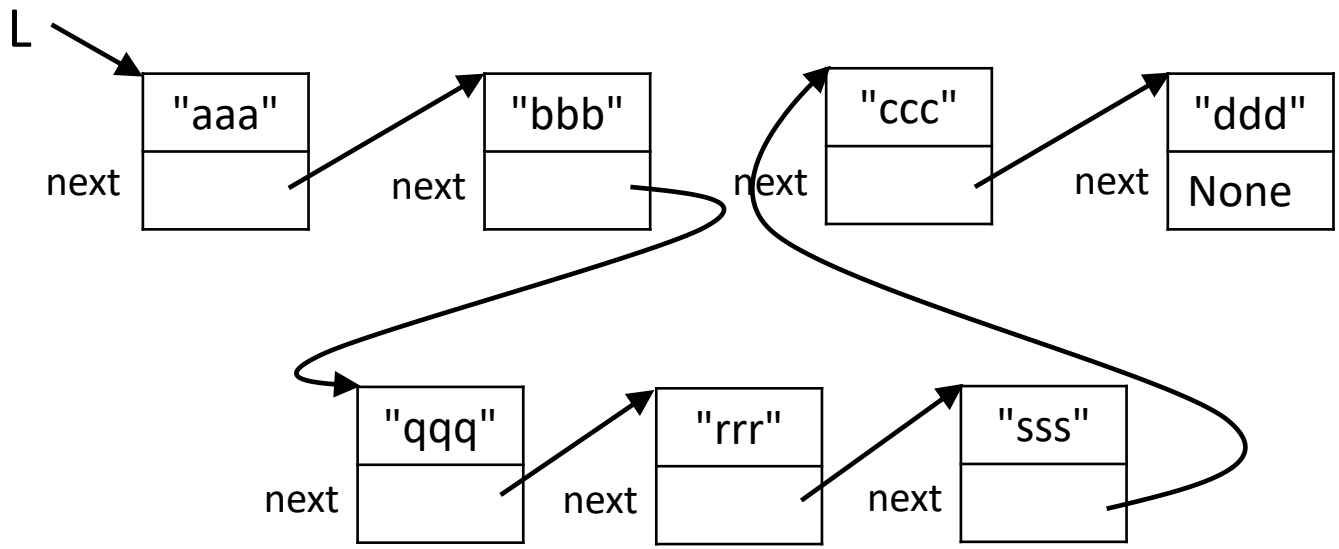
To insert an element into a linked list: set next references appropriately



Insertion

O(1)*

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

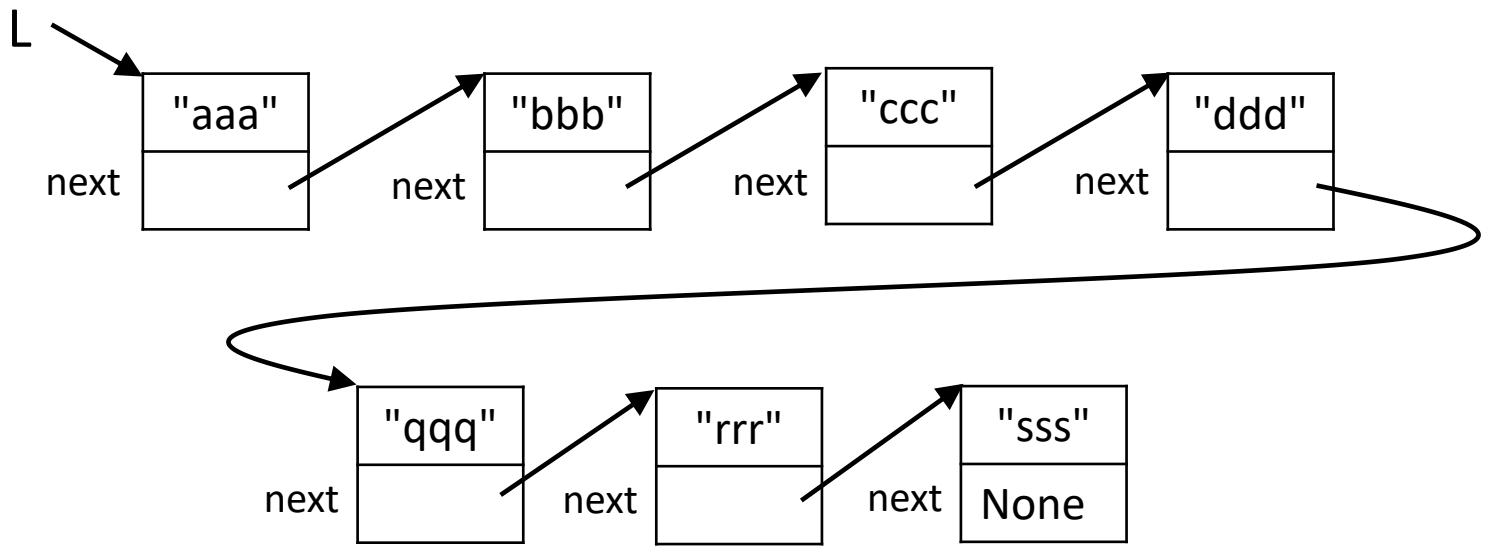


*assuming we have a reference to the node of insertion

Concatenation

O(1)*

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

addition
at the head of the list

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
```

O(1)

Visiting 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
```

O(n)

adding to the end (tail) of the list

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

O(n)

O(1)

Adding to the end

```
class LinkedList:

    def  add_to_end(self, new):
        if self._head == None:        # the list is empty
            self._head = new          # the list now has one node
        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
```

O(n)

finding the n^{th} element

Finding the n^{th} element

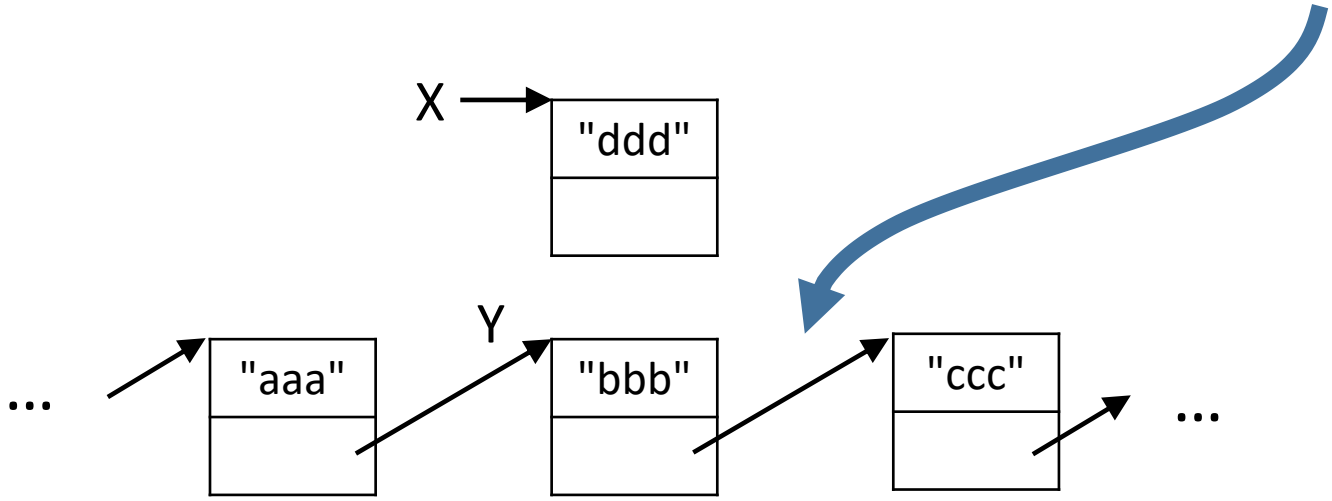
```
class LinkedList:

    # 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
```

O(n)

insertion

Inserting a node

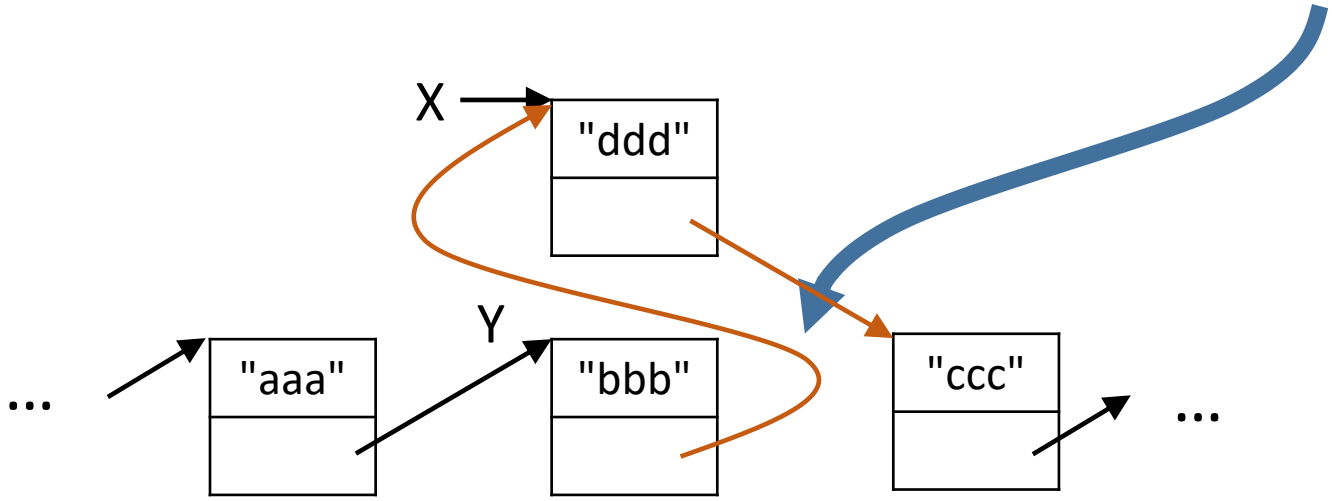
Suppose we want to insert a node X into a list here:



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

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
```

O(n)

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

O(1)

* do something sensible if the list has fewer than n-1 nodes

Inserting a node

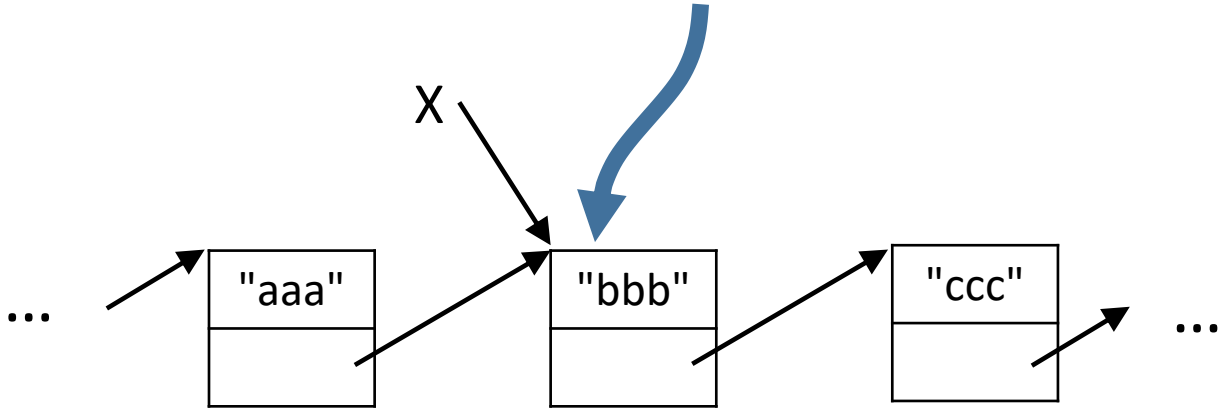
```
class LinkedList:

    # 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
```

O(n)

deletion

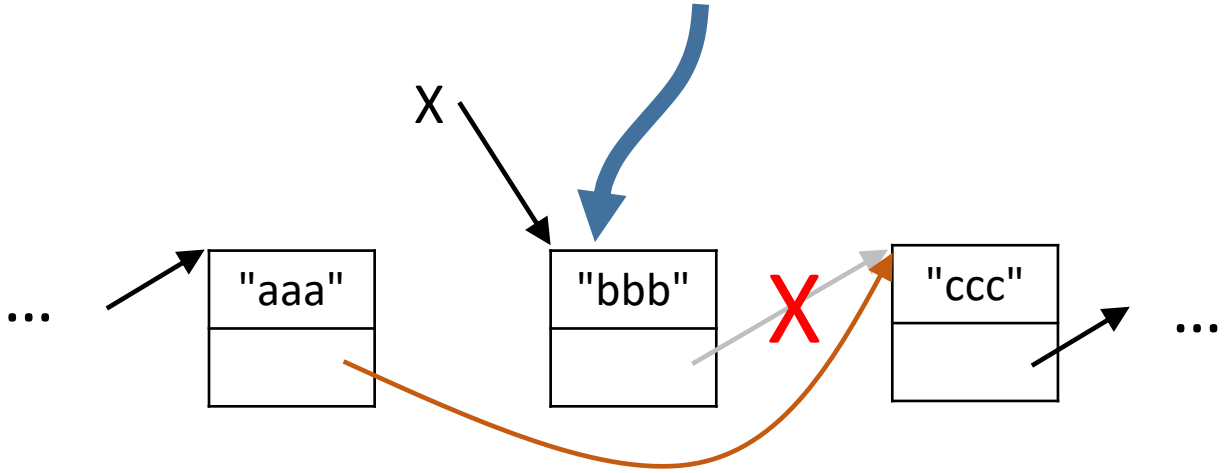
Deleting a node

Suppose we want to delete this node:



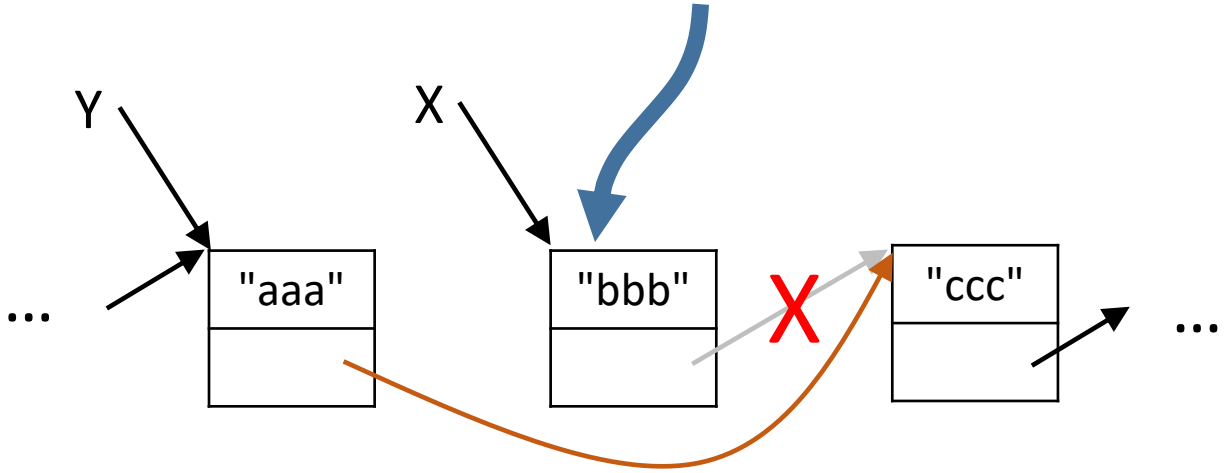
Deleting a node

Suppose we want to delete this node:



Deleting a node

Suppose we want to delete this node:



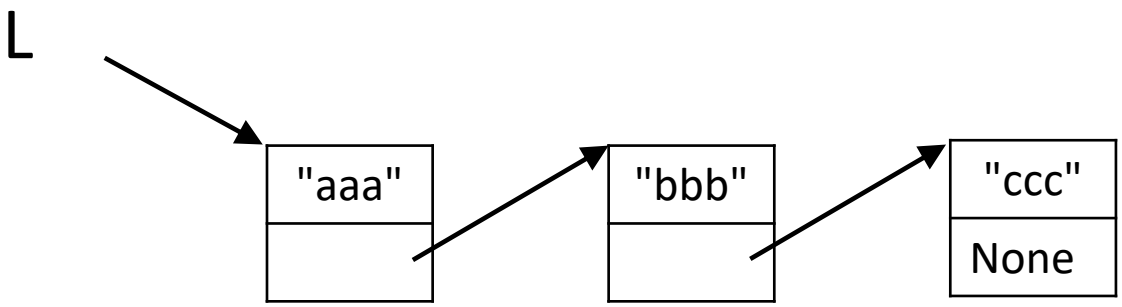
1. find the node Y just before X (i.e., Y._next == X) } O(n)
2. Y._next = X._next } O(1)
3. X._next = None

Deleting a node

```
class LinkedList:
    # delete a node X
    def delete(self, X):
        if self._head == X:                O(1)
            self._head = X._next
        else:
            Y = self._head
            while Y._next != X:            O(n)
                Y = Y._next
            Y._next = X._next
        X.next = None
```

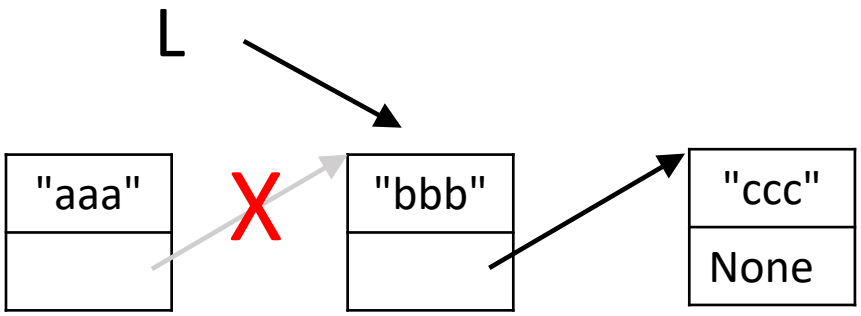
Remove from the front

Removing from the front is simpler:



Removing a node from the front

Removing from the front is simpler:



O(1)

concatenation

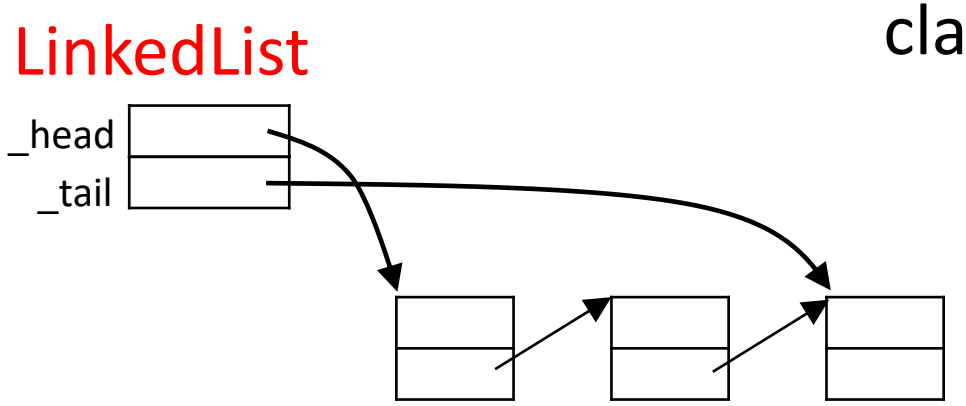
Exercise-ICA 31 Prob. 3

```
class LinkedList:
    # concatenate list2 at the end of the list
    def concat(self, list2):
```

maintaining a tail reference

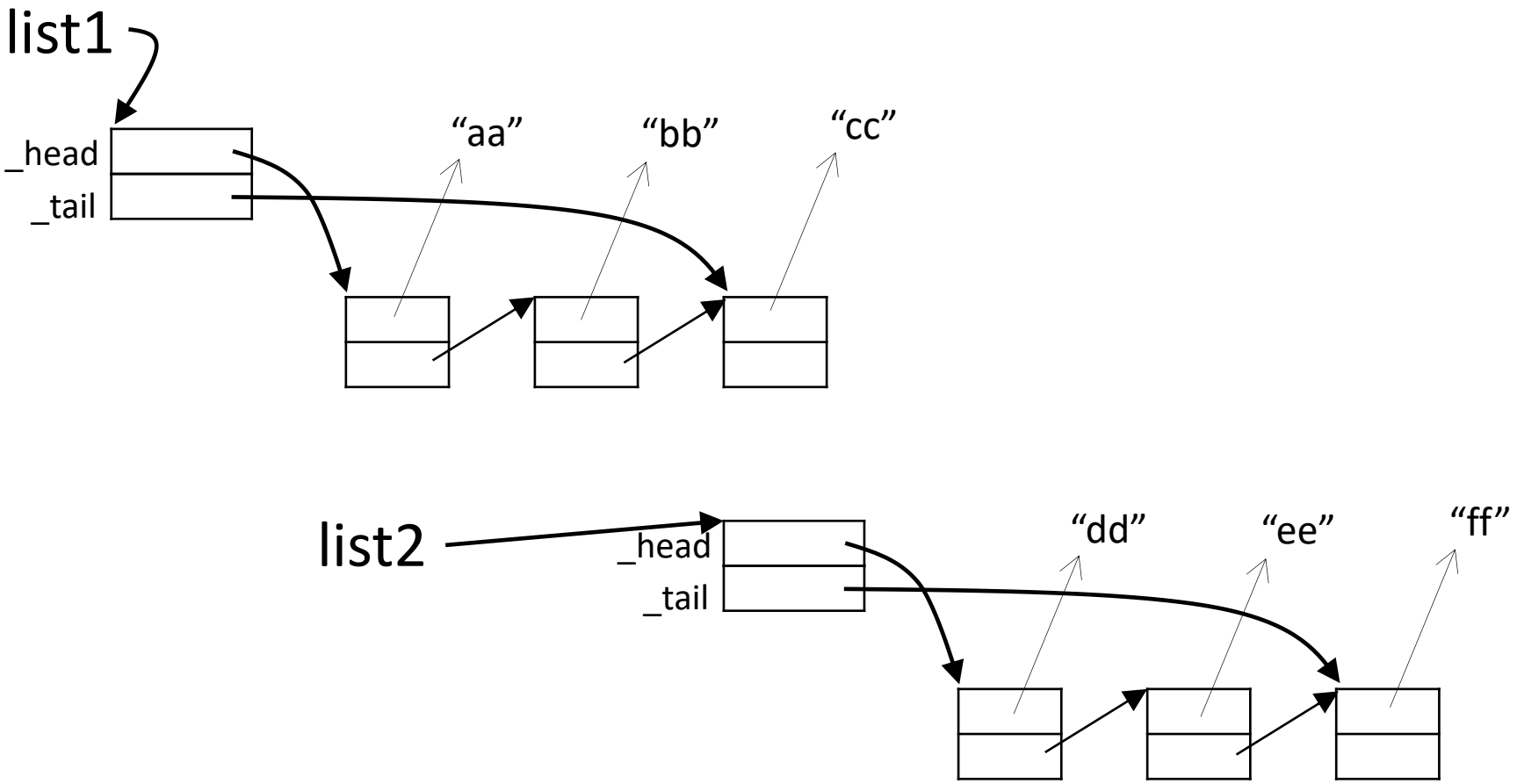
Maintaining a tail reference

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

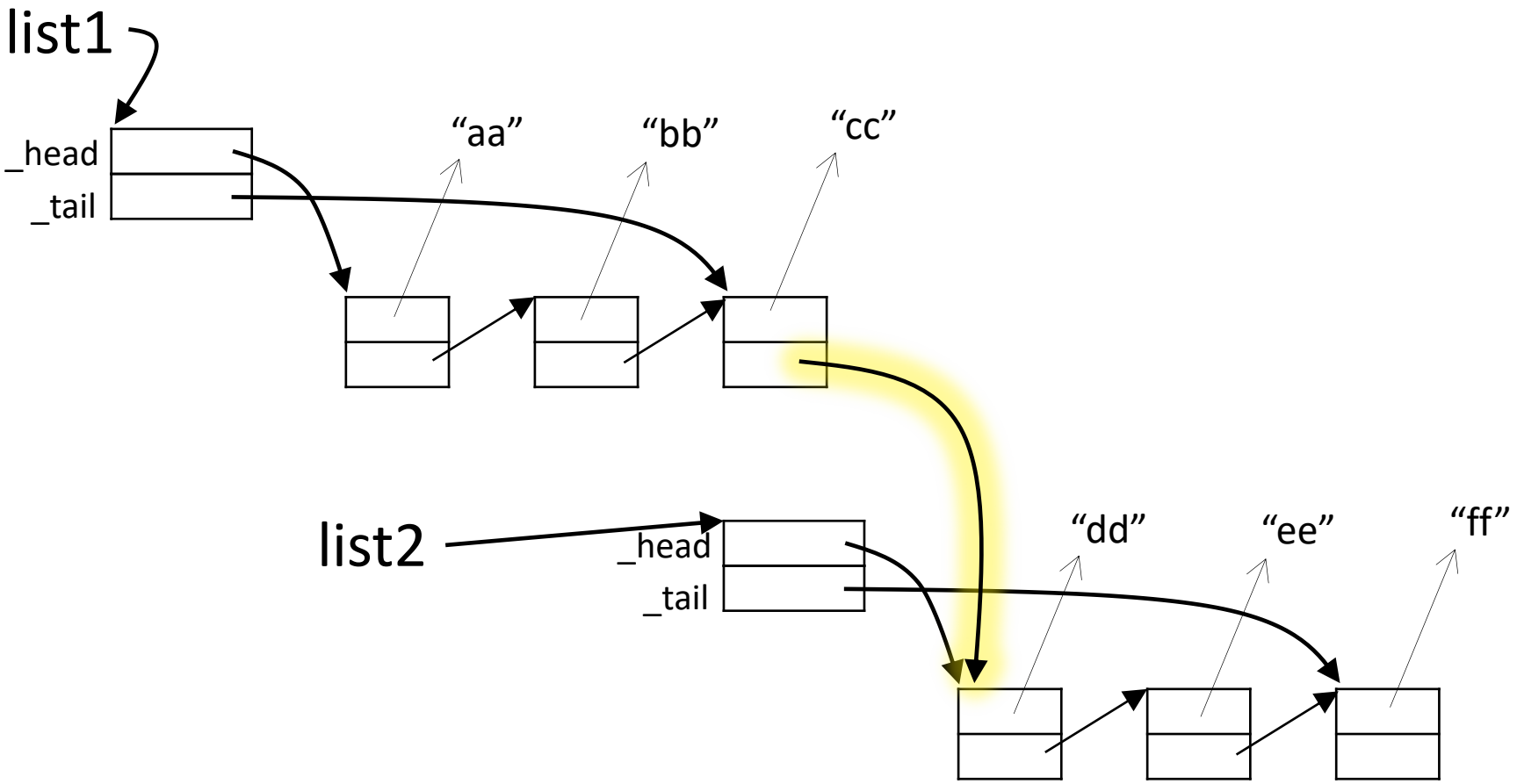



```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
```

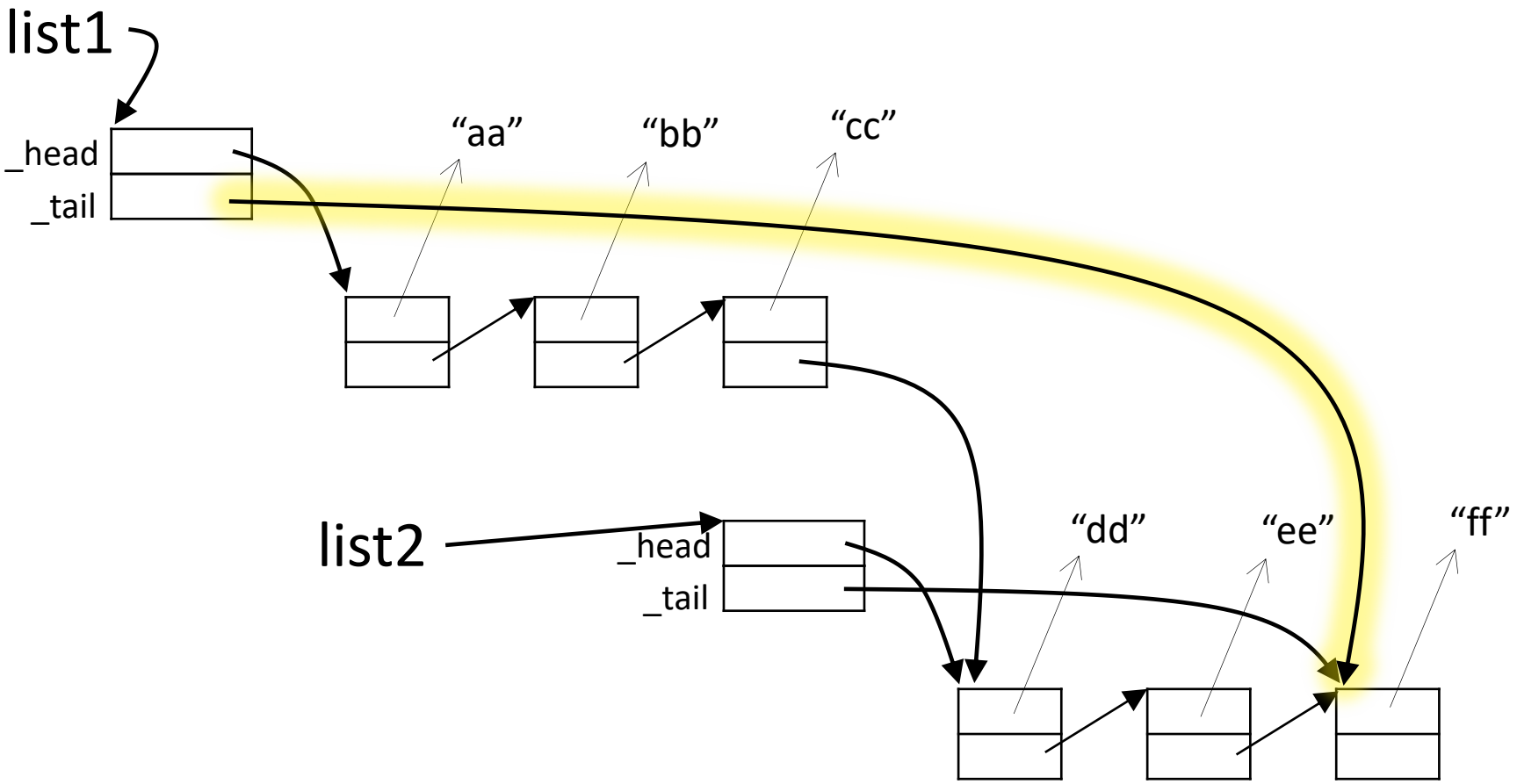
Tail references and concatenation



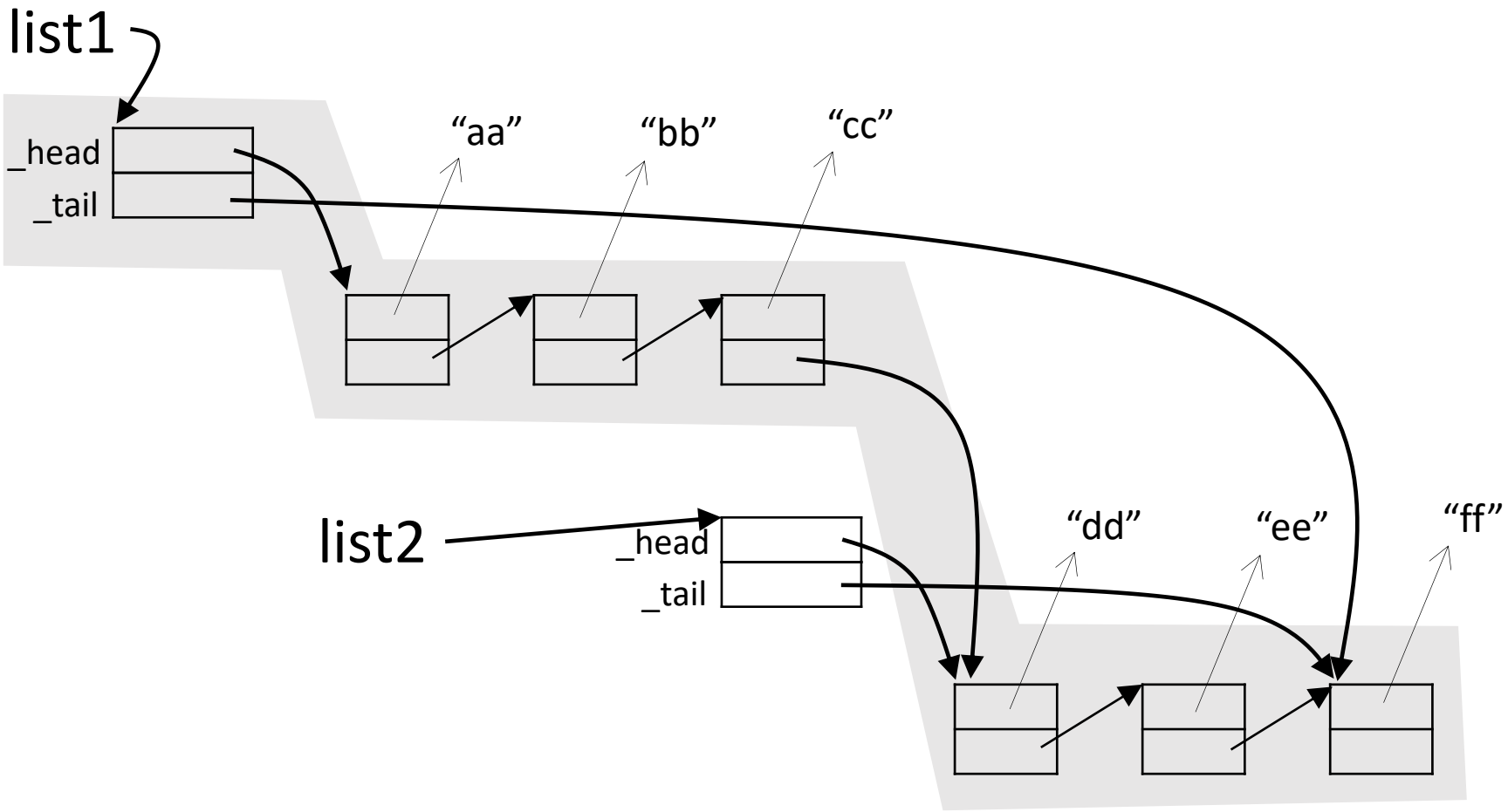
Tail references and concatenation



Tail references and concatenation



Tail references and concatenation



Maintaining a tail reference

- Concatenation and append become O(1):

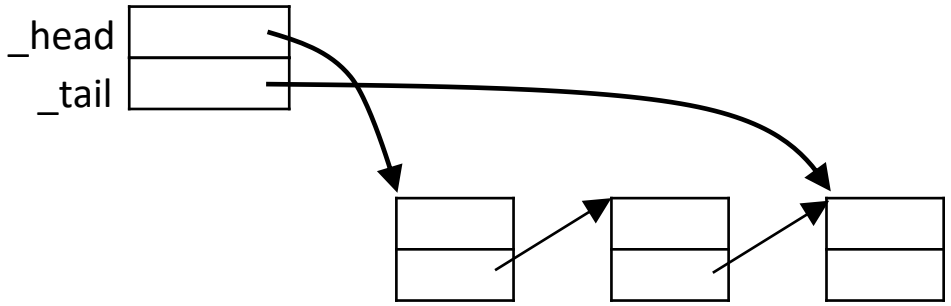
```
def concat(self, list2):
    if self._head == None:
        self._head = list2._head
        self._tail = list2._tail
    else:
        self._tail._next = list2._head
        self._tail = list2._tail
```

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

Exercise-ICA-32, p.1-3

Given the following LinkedList definition:

LinkedList




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

Write the append(self, new) method for the class.

Linked lists: summary

Operation	Without tail reference	With tail reference
add to front of list	O(1)	
append to end of list	O(n)	O(1)
find nth element	O(n)	
insert	O(1) if prev. node is available O(n) otherwise	
delete	O(1) if prev. node is available O(n) otherwise	
concatenate	O(n)	O(1)

Some common growth-rate curves