4 Linked Lists

For COMSC 132

Sam Bowne

Sep 12, 2024

Topics

- Arrays
- Introducing linked lists
- Doubly linked lists
- Circular lists
- Practical applications of linked lists

Arrays

Array

- A collection of data items of the same type
- Stored in contiguous memory locations
- Position of an element is **base address** plus **offset**
- Static size declared at time of creation

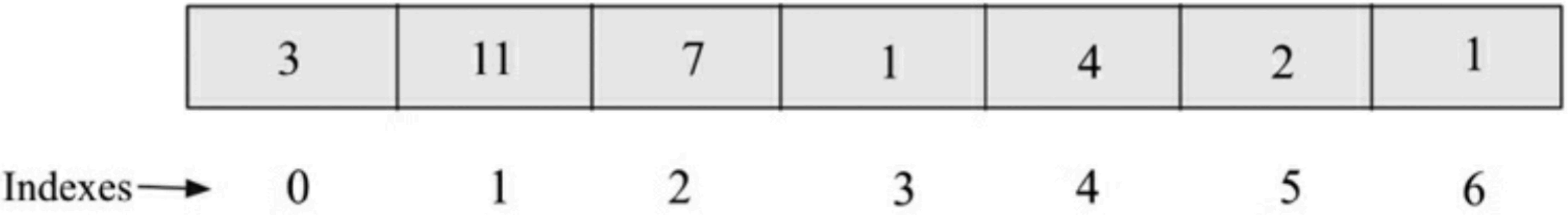


Figure 4.1: Representation of a one-dimensional array

Linked lists

- A collection of data items of the same type
- Stored sequentially
- Connected through pointers
- Stored in different memory locations

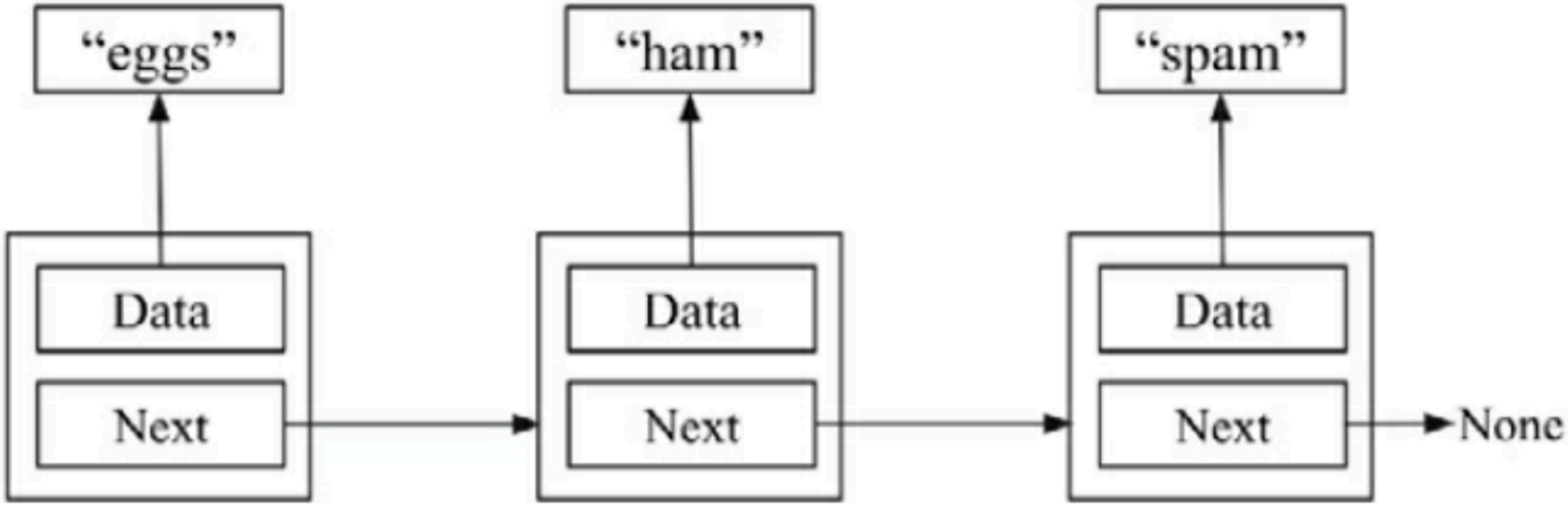


Figure 4.4: A sample linked list of three nodes

Array speed

- Very fast to **store**, **traverse**, or **access** data
  - O(***1***)
- Allows random access
- Slow for **insert** or **delete** operations — O(***n***)
- Poor performance if the array is too large to store in memory

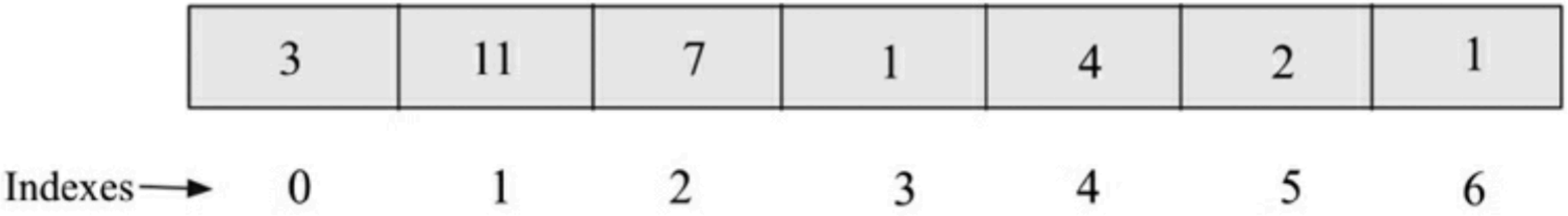


Figure 4.1: Representation of a one-dimensional array

Linked list speed

- **Insert** and **delete** are fast — O(***1***)
- Slow to **store**, **traverse**, or **access** data
  - O(***n***)
- Length of the list can increase or decrease during program execution

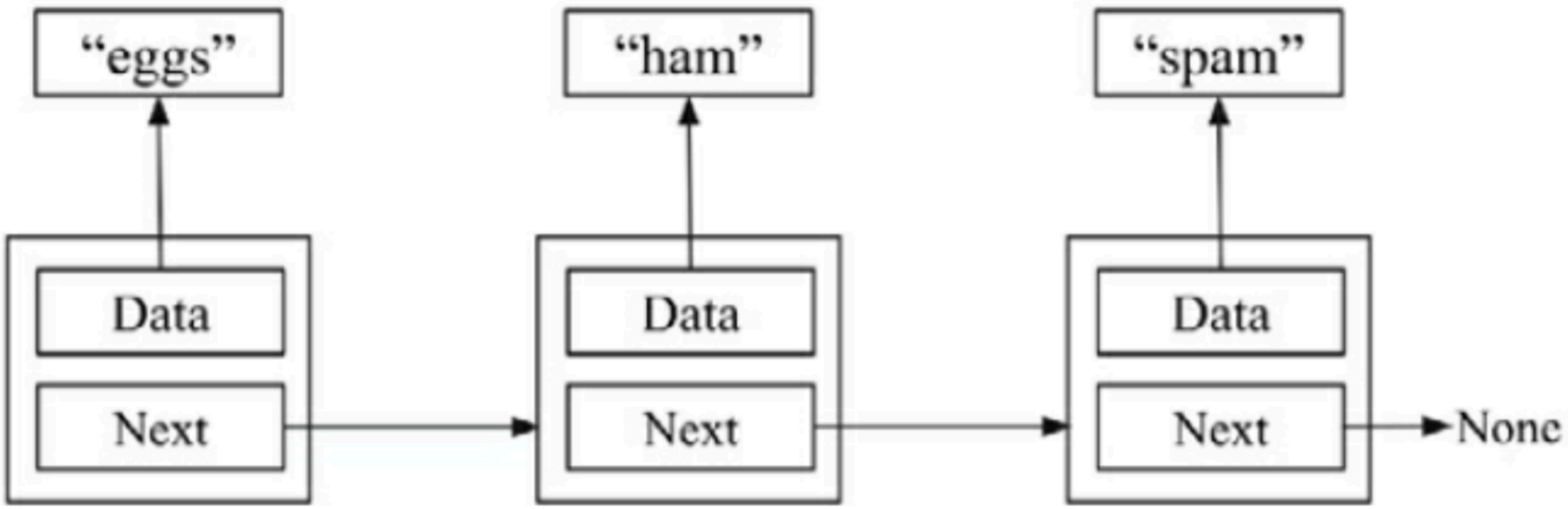


Figure 4.4: A sample linked list of three nodes

Introducing linked lists

Linked Lists

- Each data item is called a **node**
- Each **node** stores **data** and a **pointer**
- The last node points to None

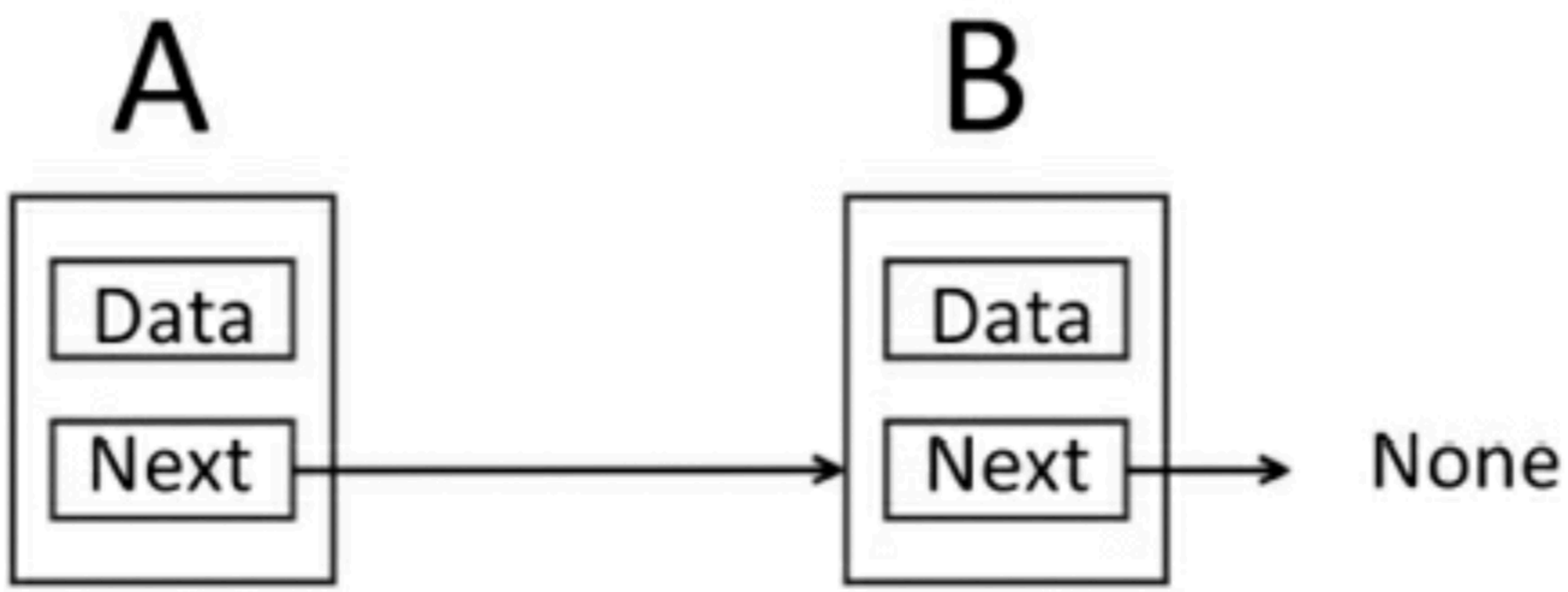
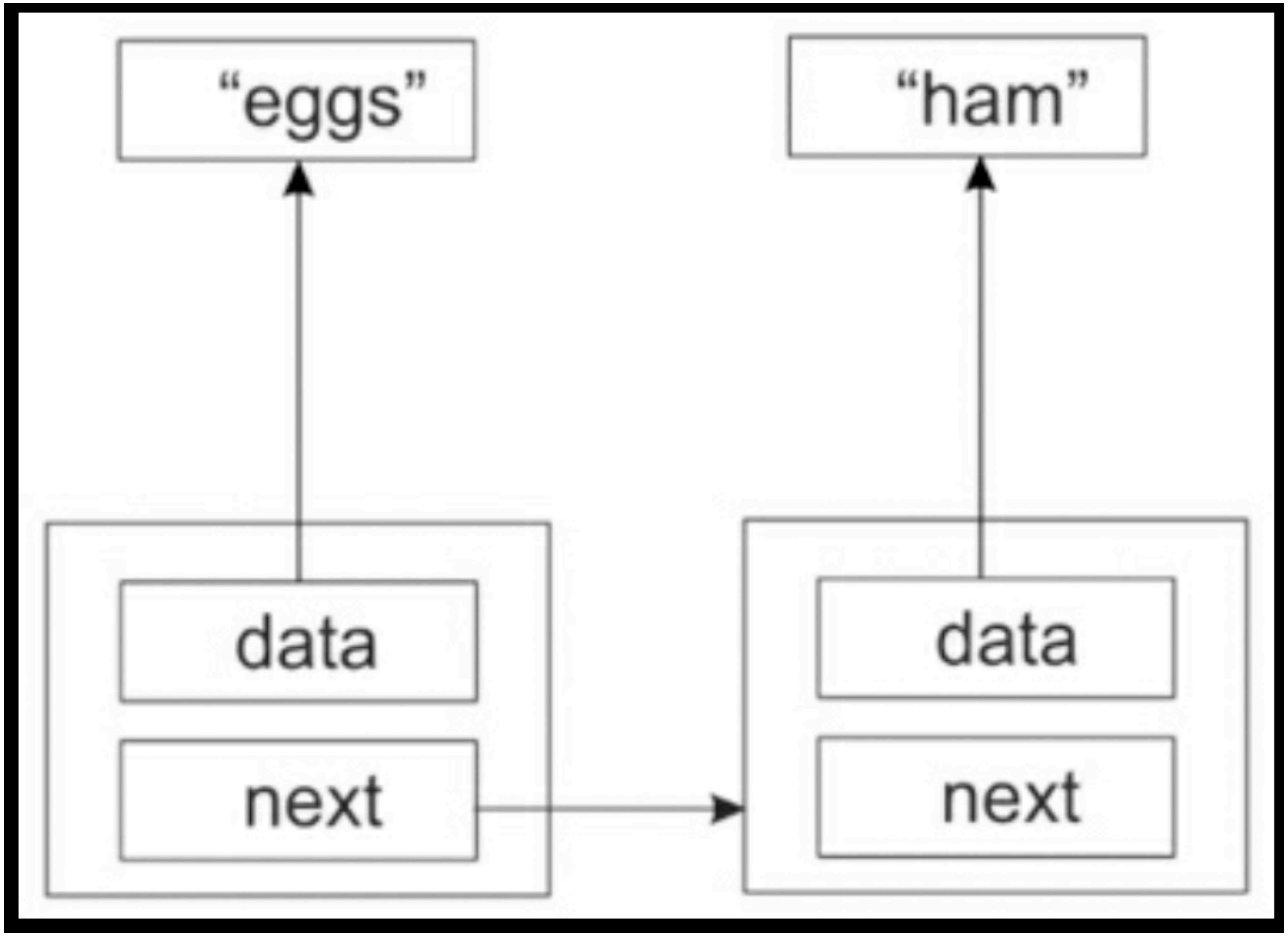


Figure 4.2: A linked list with two nodes

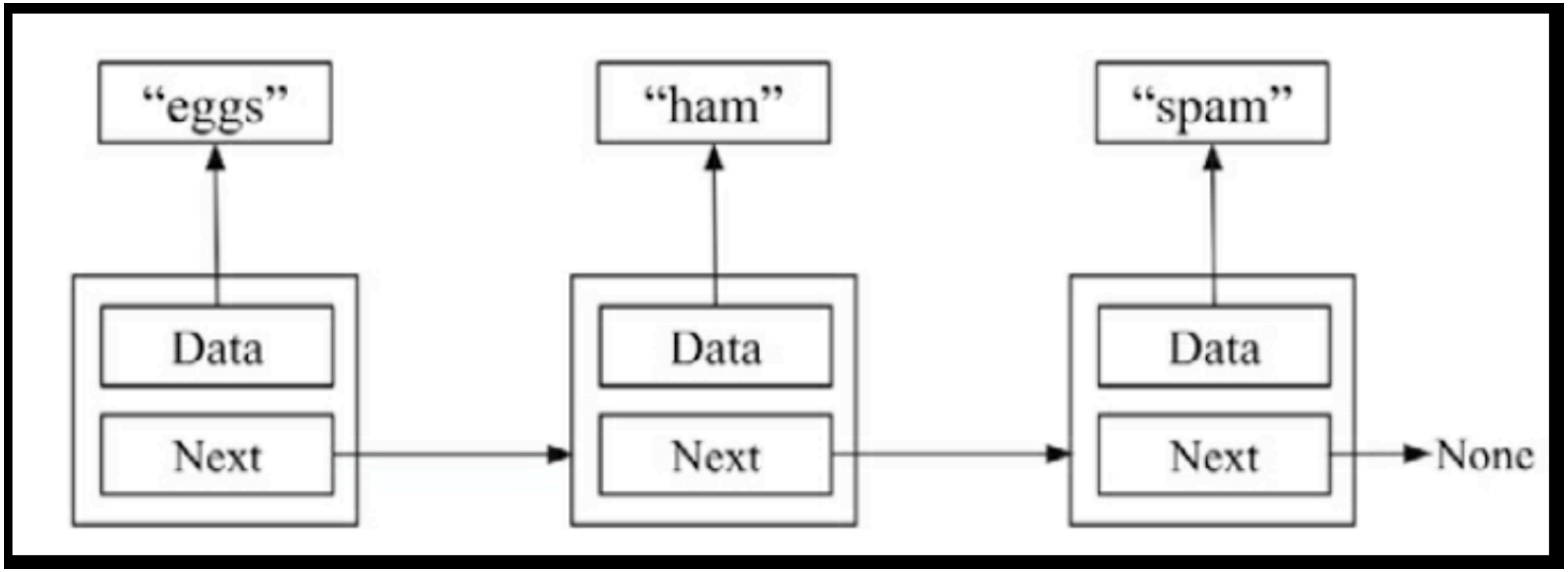
Nodes and pointers

- The nodes may contain pointers as data

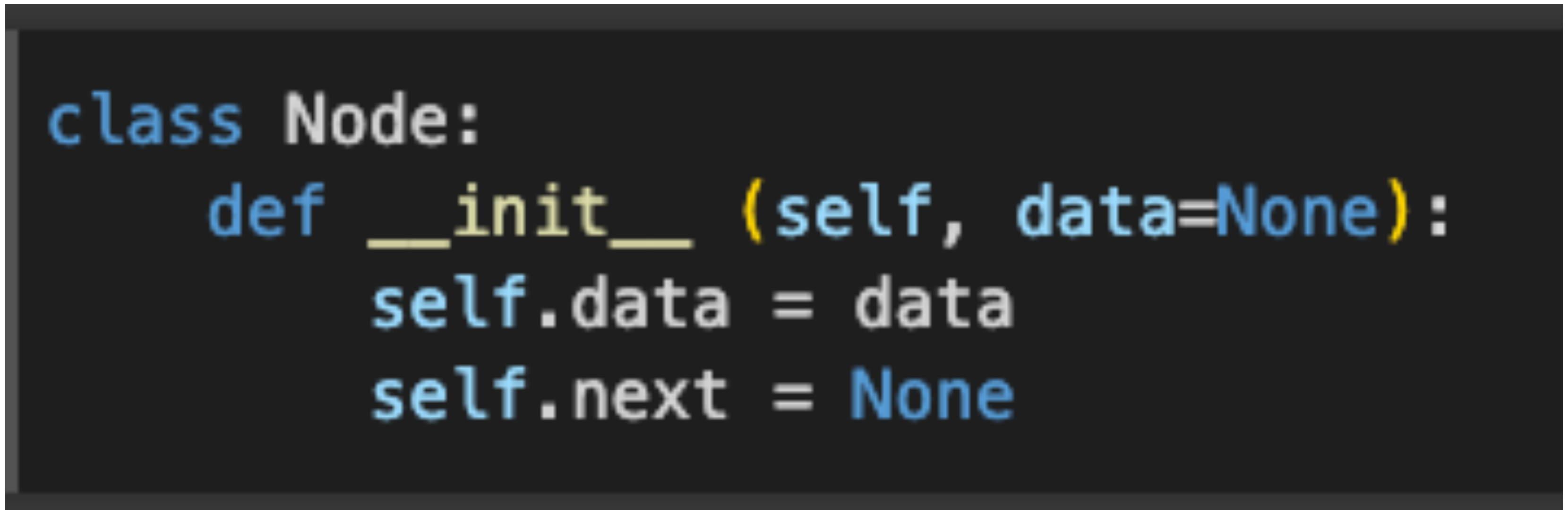


Three nodes

- Last node points to **None**
- Indicating the end of the list

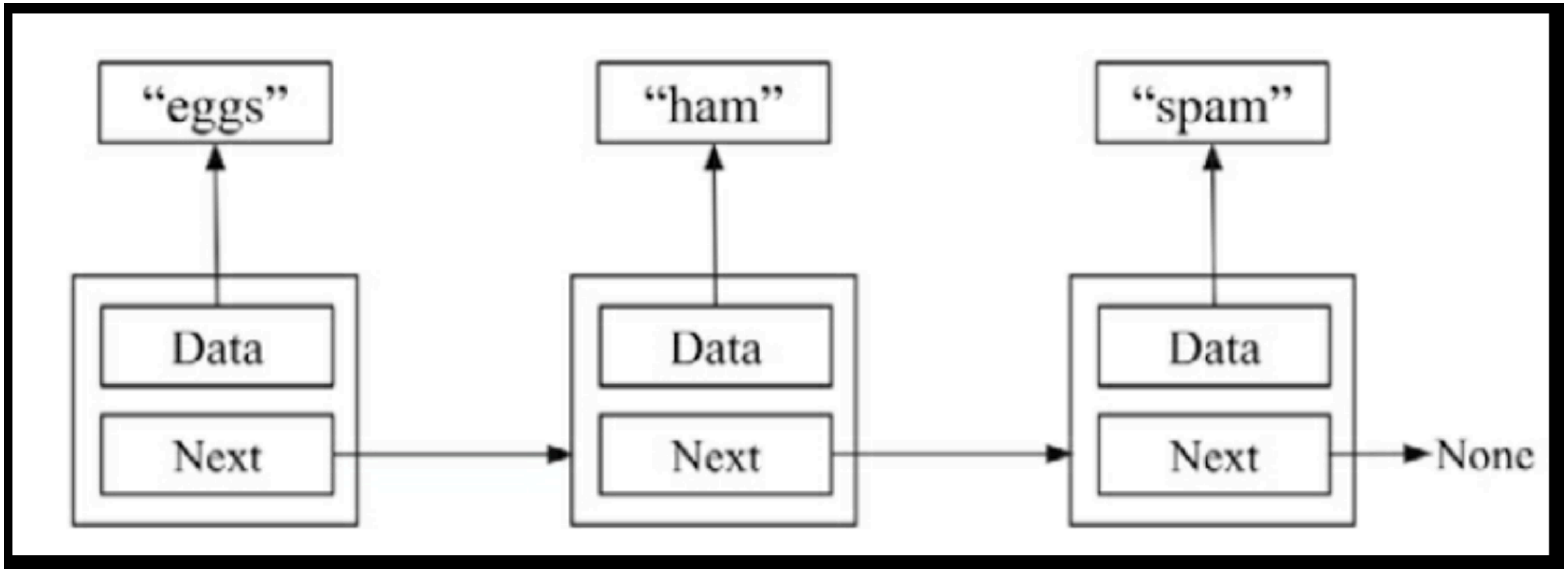


Implementation of a node



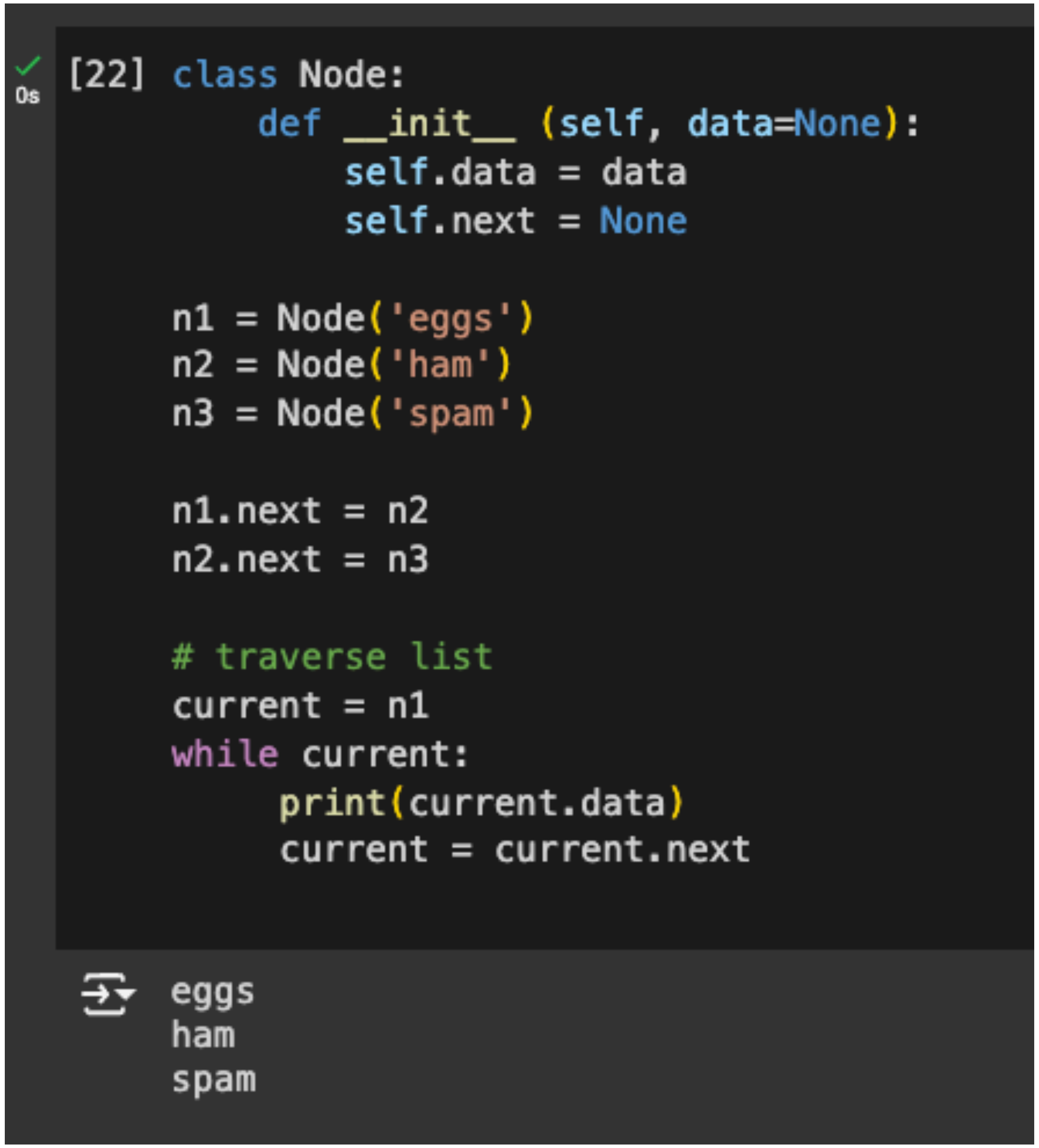
```python
class Node:
    def __init__ (self, data=None):
        self.data = data
        self.next = None
```

Singly linked list



Creating and traversing a list

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

n1 = Node('eggs')
n2 = Node('ham')
n3 = Node('spam')

n1.next = n2
n2.next = n3

# traverse list
current = n1
while current:
     print(current.data)
     current = current.next
```

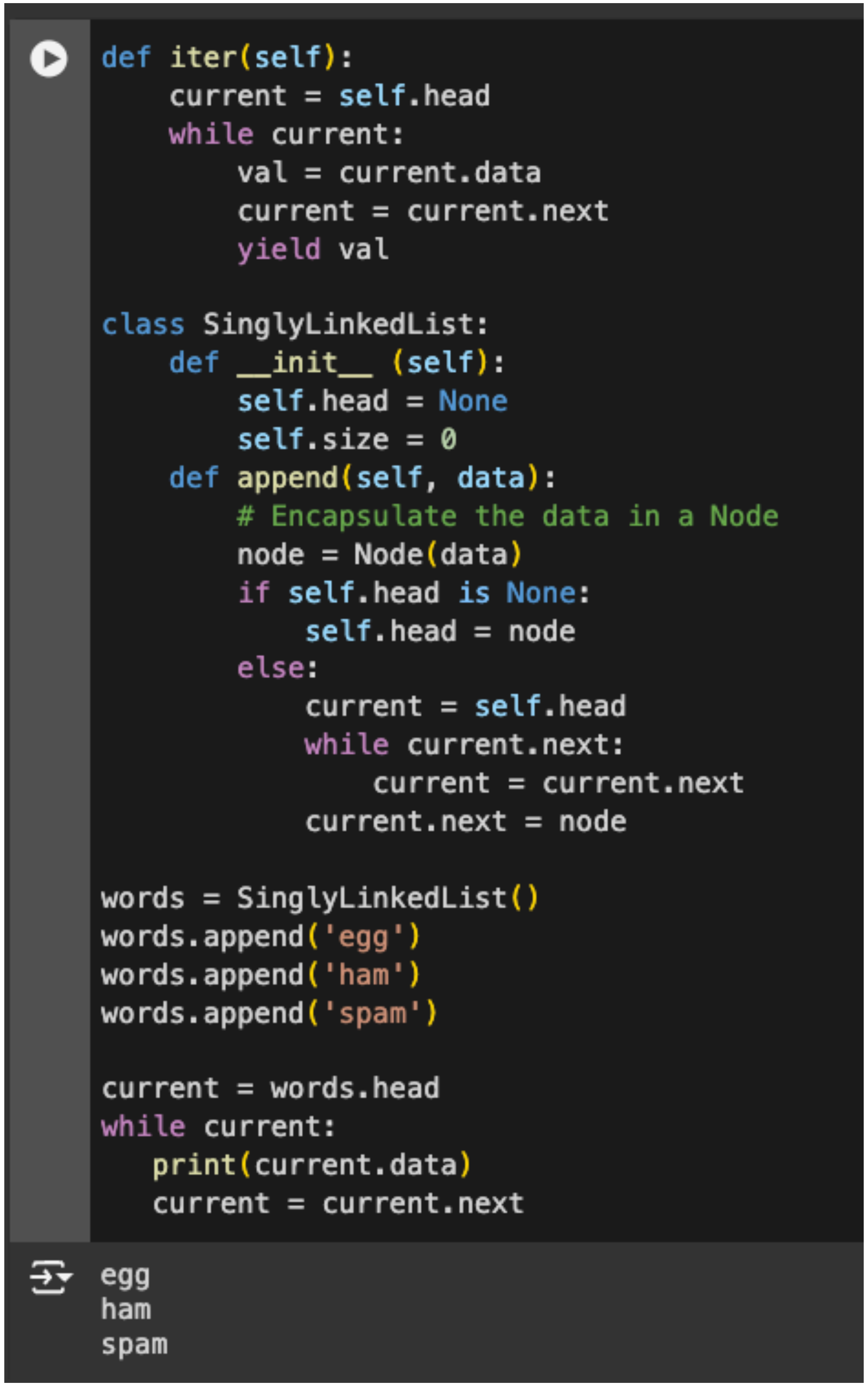


```
[22] class Node:
         def __init__ (self, data=None):
             self.data = data
             self.next = None

     n1 = Node('eggs')
     n2 = Node('ham')
     n3 = Node('spam')

     n1.next = n2
     n2.next = n3

     # traverse list
     current = n1
     while current:
          print(current.data)
          current = current.next
```

```
eggs
ham
spam
```

Improved list creation and traversal

- Encapsulates the Node object
  - End-user does not use it directly
- Generator method uses **yield** instead of **return**
- append traverses the whole list to find the end



```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val

class SinglyLinkedList:
    def __init__ (self):
        self.head = None
        self.size = 0
    def append(self, data):
        # Encapsulate the data in a Node
        node = Node(data)
        if self.head is None:
            self.head = node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = node

words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

current = words.head
while current:
   print(current.data)
   current = current.next
```

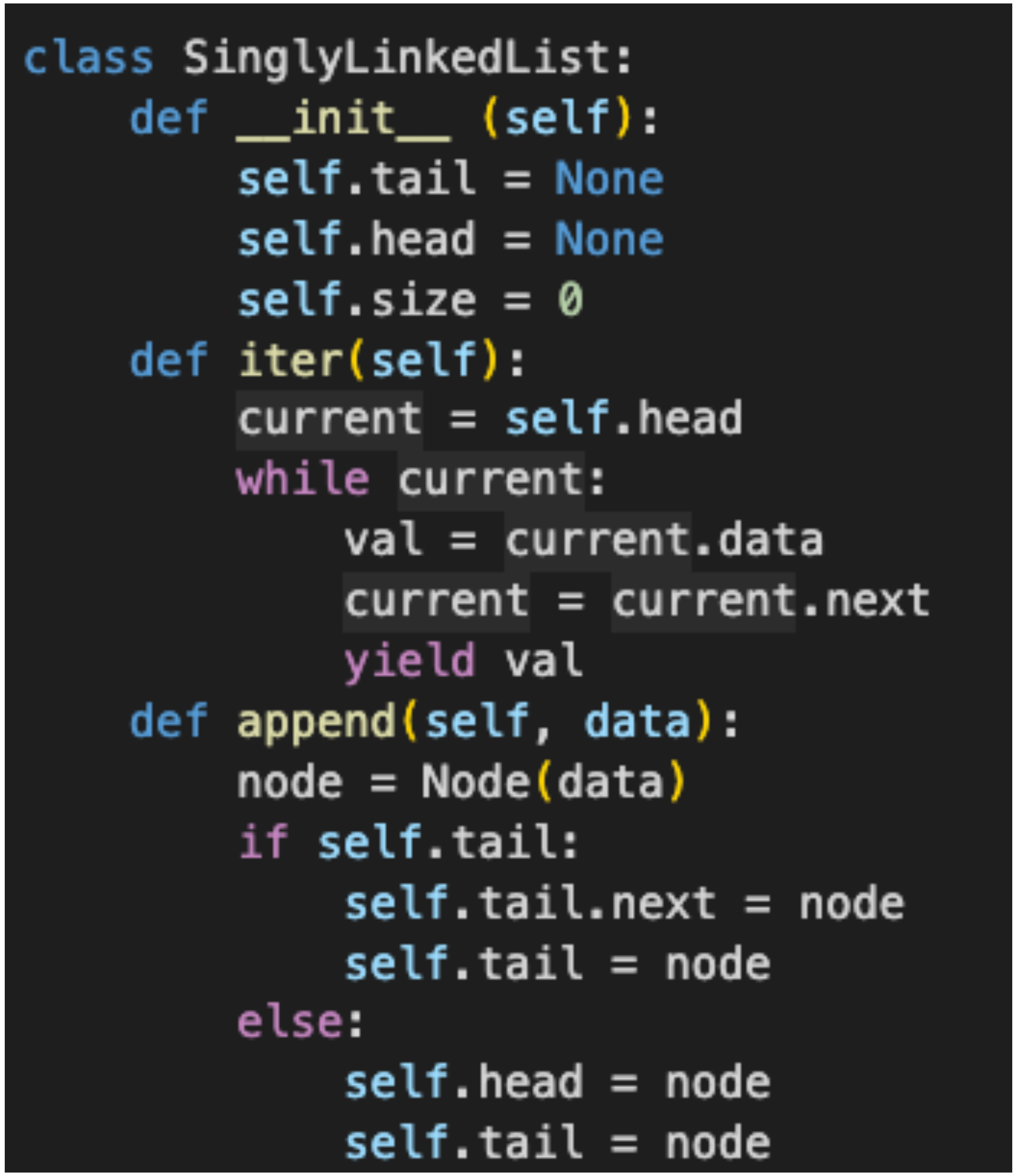
```
egg
ham
spam
```

Code

```python
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val

class SinglyLinkedList:
    def __init__ (self):
        self.head = None
        self.size = 0
    def append(self, data):
        # Encapsulate the data in a Node
        node = Node(data)
        if self.head is None:
            self.head = node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = node

words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

current = words.head
while current:
   print(current.data)
   current = current.next
```

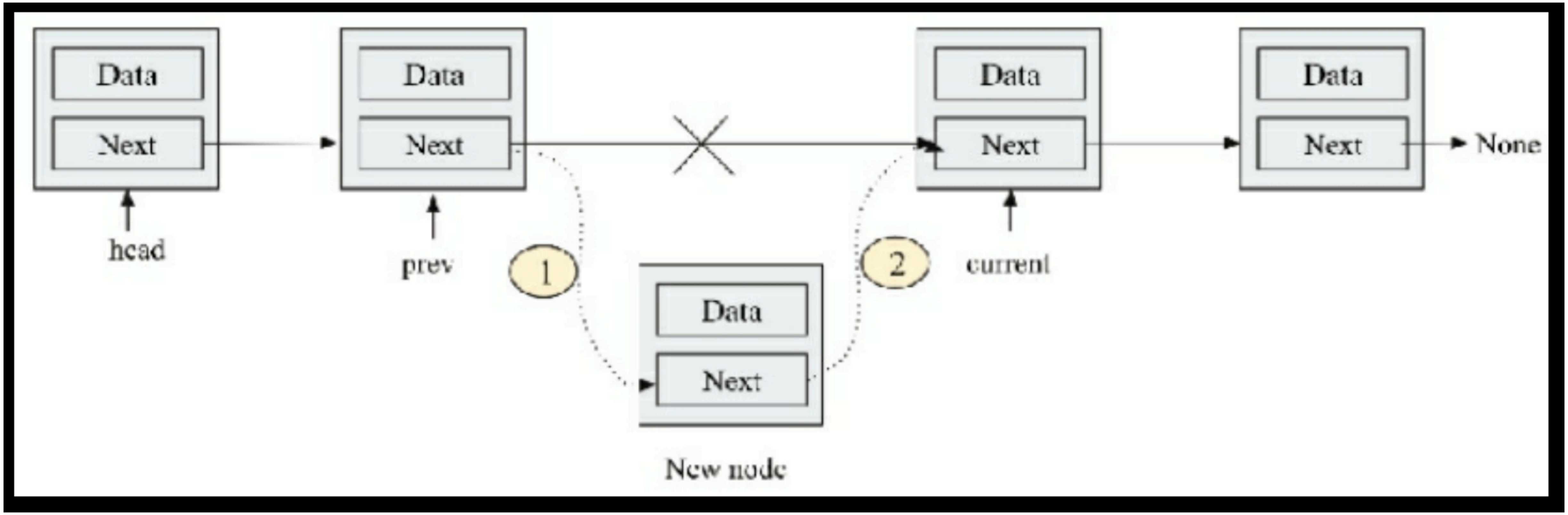
List with head and tail pointers

- Append is more efficient



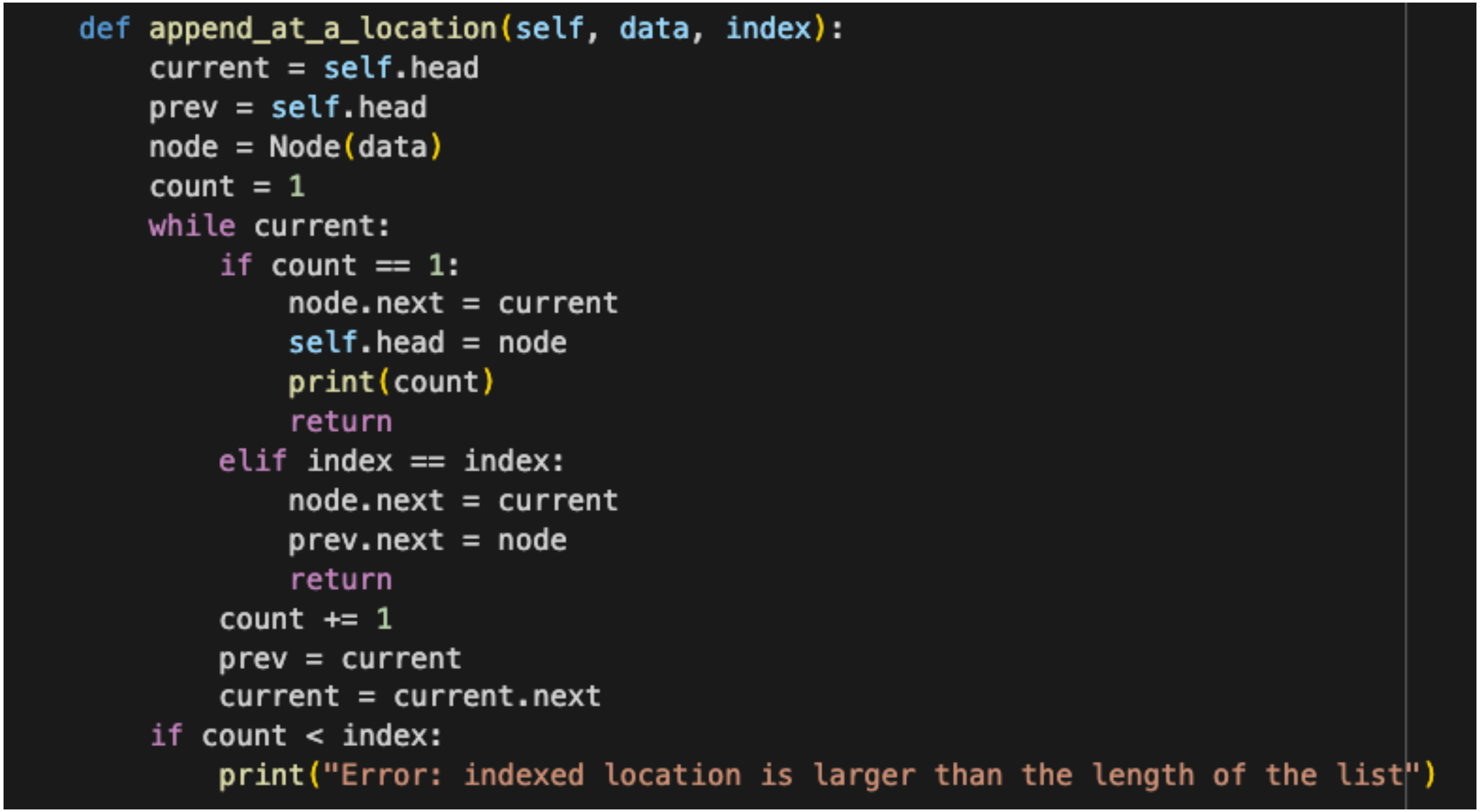
```
class SinglyLinkedList:
    def __init__ (self):
        self.tail = None
        self.head = None
        self.size = 0
    def iter(self):
        current = self.head
        while current:
            val = current.data
            current = current.next
            yield val
    def append(self, data):
        node = Node(data)
        if self.tail:
            self.tail.next = node
            self.tail = node
        else:
            self.head = node
            self.tail = node
```

Inserting a node



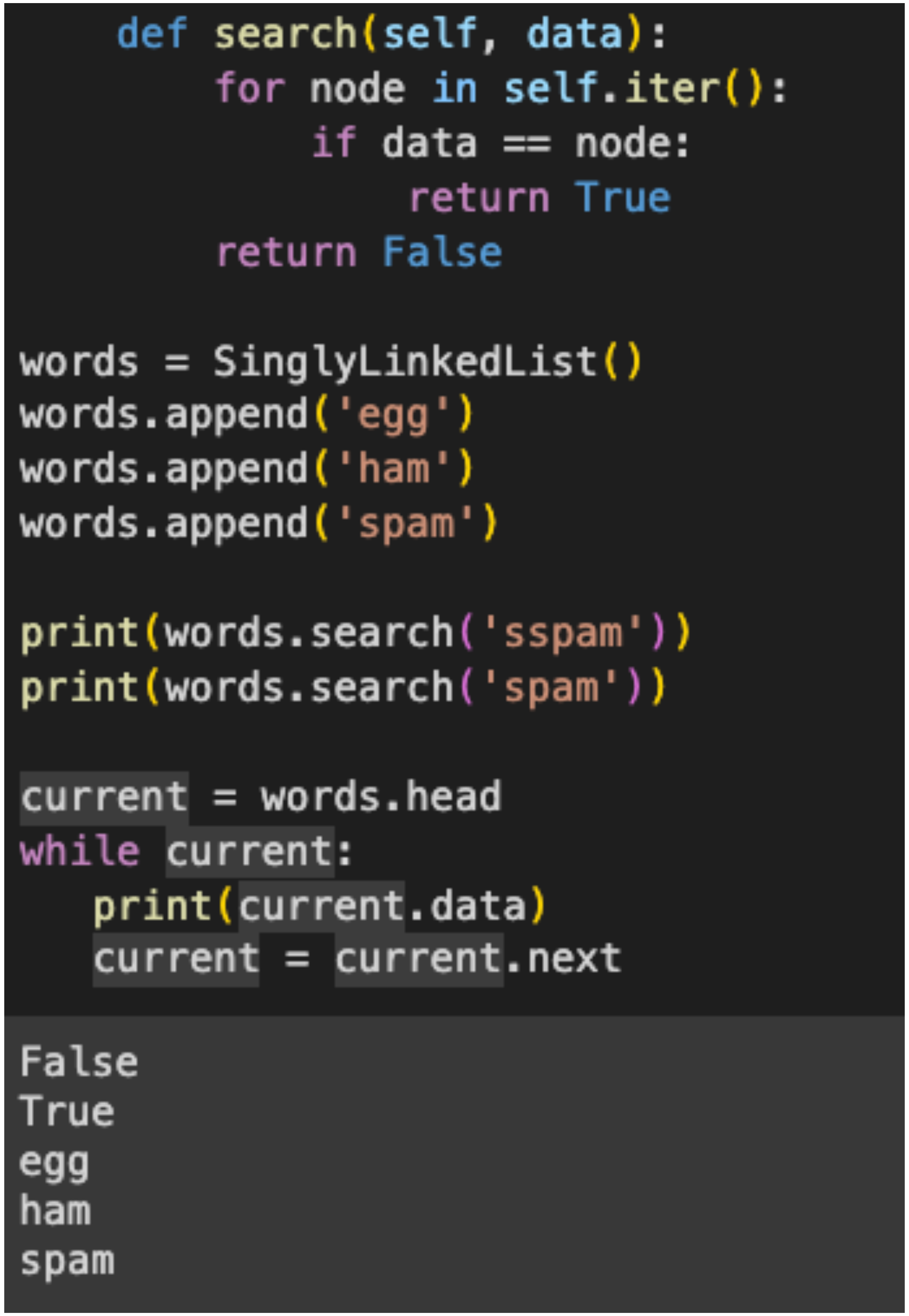
- Must update two links
- Complexity is O(***n***) if there is no link to the tail, but O(***1***) if there is, because the new node goes at the tail

Inserting a node



```
    def append_at_a_location(self, data, index):
        current = self.head
        prev = self.head
        node = Node(data)
        count = 1
        while current:
            if count == 1:
                node.next = current
                self.head = node
                print(count)
                return
            elif index == index:
                node.next = current
                prev.next = node
                return
            count += 1
            prev = current
            current = current.next
        if count < index:
            print("Error: indexed location is larger than the length of the list")
```

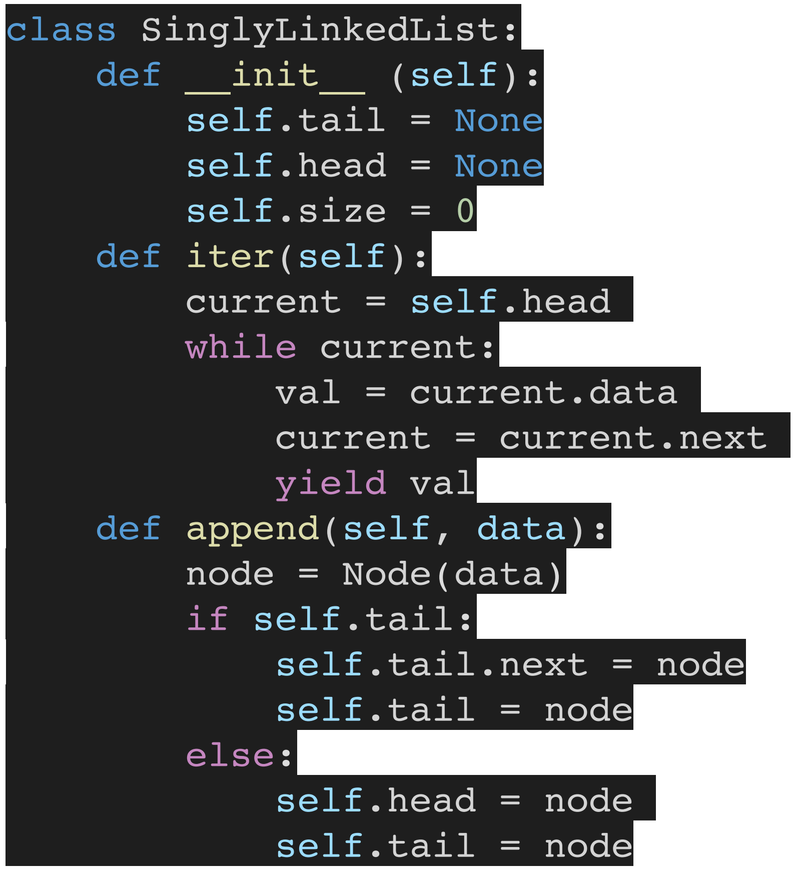
Querying a list



```
    def search(self, data):
        for node in self.iter():
            if data == node:
                return True
        return False

words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

print(words.search('sspam'))
print(words.search('spam'))

current = words.head
while current:
   print(current.data)
   current = current.next
```

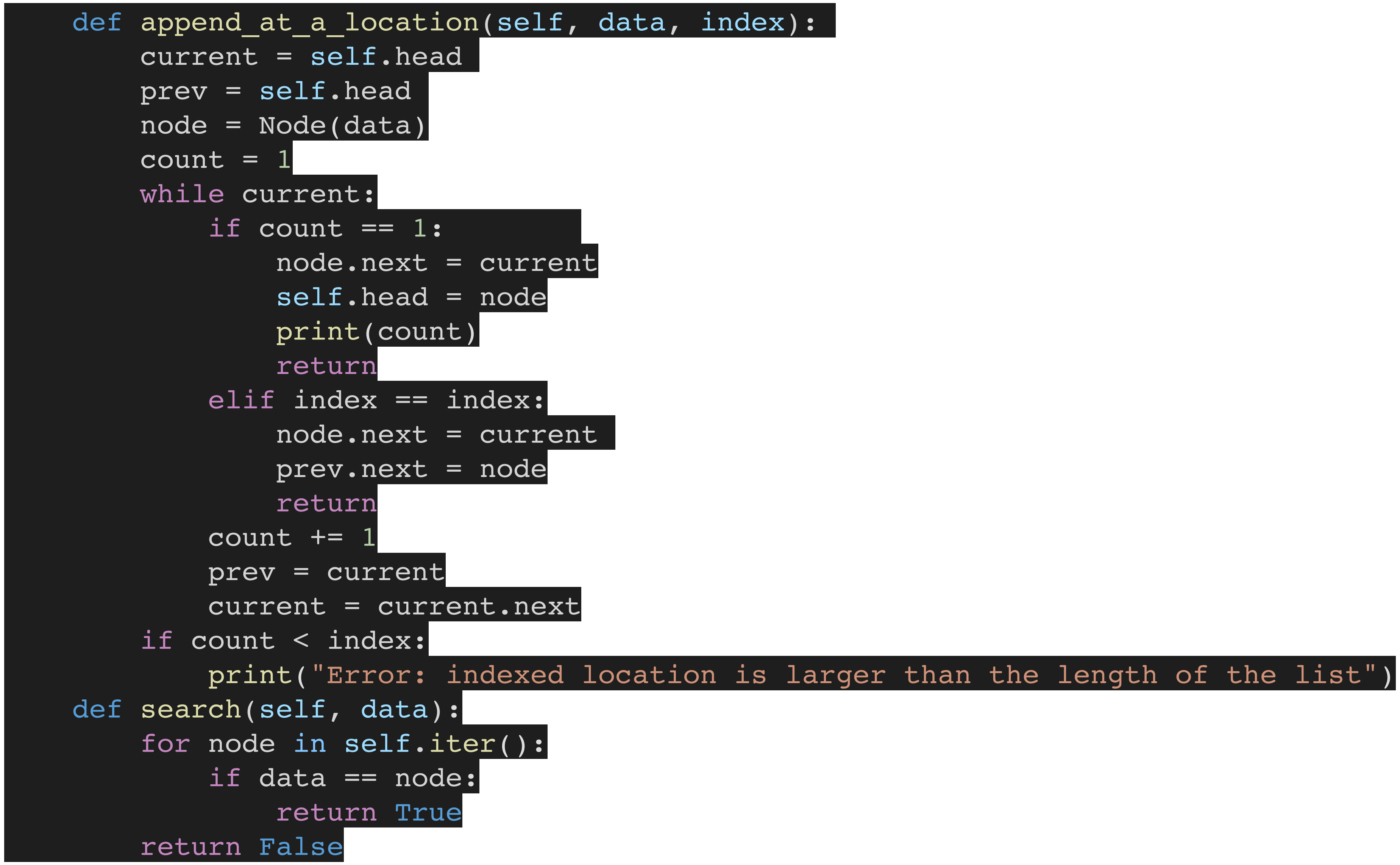
```
False
True
egg
ham
spam
```

Code



```
class SinglyLinkedList:
    def __init__ (self):
        self.tail = None
        self.head = None
        self.size = 0
    def iter(self):
        current = self.head
        while current:
            val = current.data
            current = current.next
            yield val
    def append(self, data):
        node = Node(data)
        if self.tail:
            self.tail.next = node
            self.tail = node
        else:
            self.head = node
            self.tail = node
```

Code



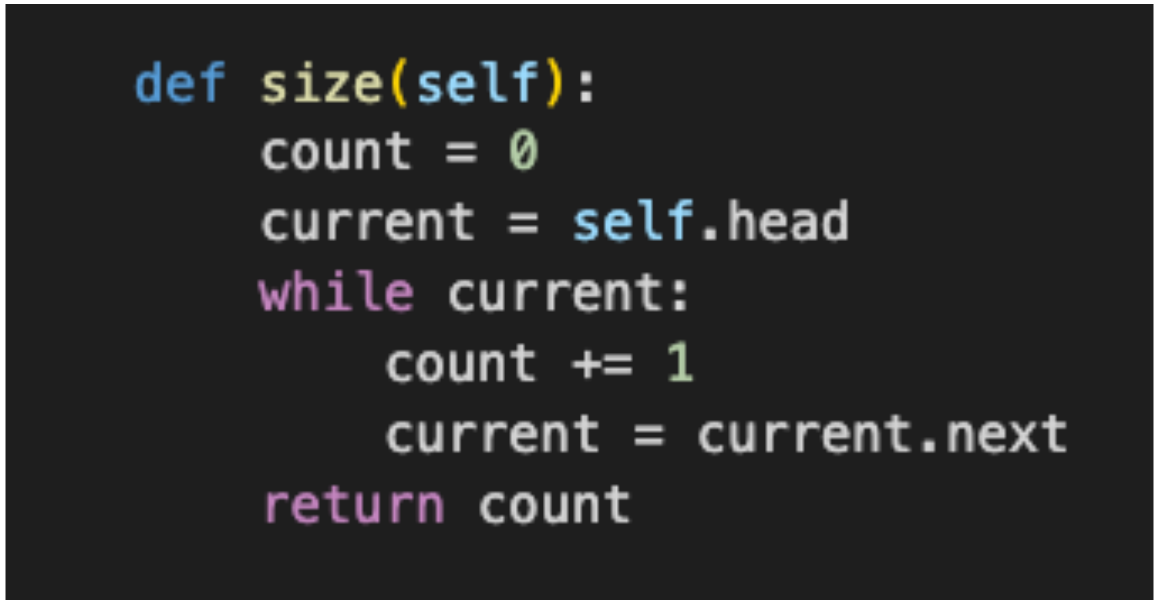
```
    def append_at_a_location(self, data, index):
        current = self.head
        prev = self.head
        node = Node(data)
        count = 1
        while current:
            if count == 1:
                node.next = current
                self.head = node
                print(count)
                return
            elif index == index:
                node.next = current
                prev.next = node
                return
            count += 1
            prev = current
            current = current.next
        if count < index:
            print("Error: indexed location is larger than the length of the list")
    def search(self, data):
        for node in self.iter():
            if data == node:
                return True
        return False
```

Code

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

print(words.search('sspam'))
print(words.search('spam'))

current = words.head
while current:
   print(current.data)
   current = current.next
```

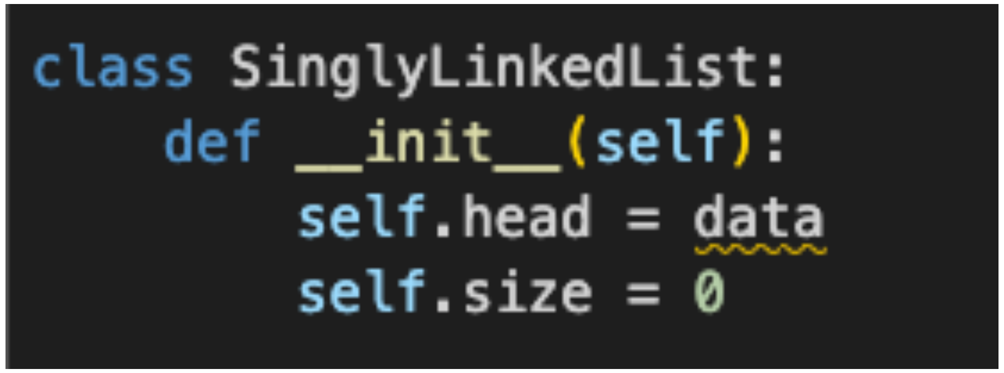
Getting the size of a list

- One way: traverse the list
    - O(***n***)



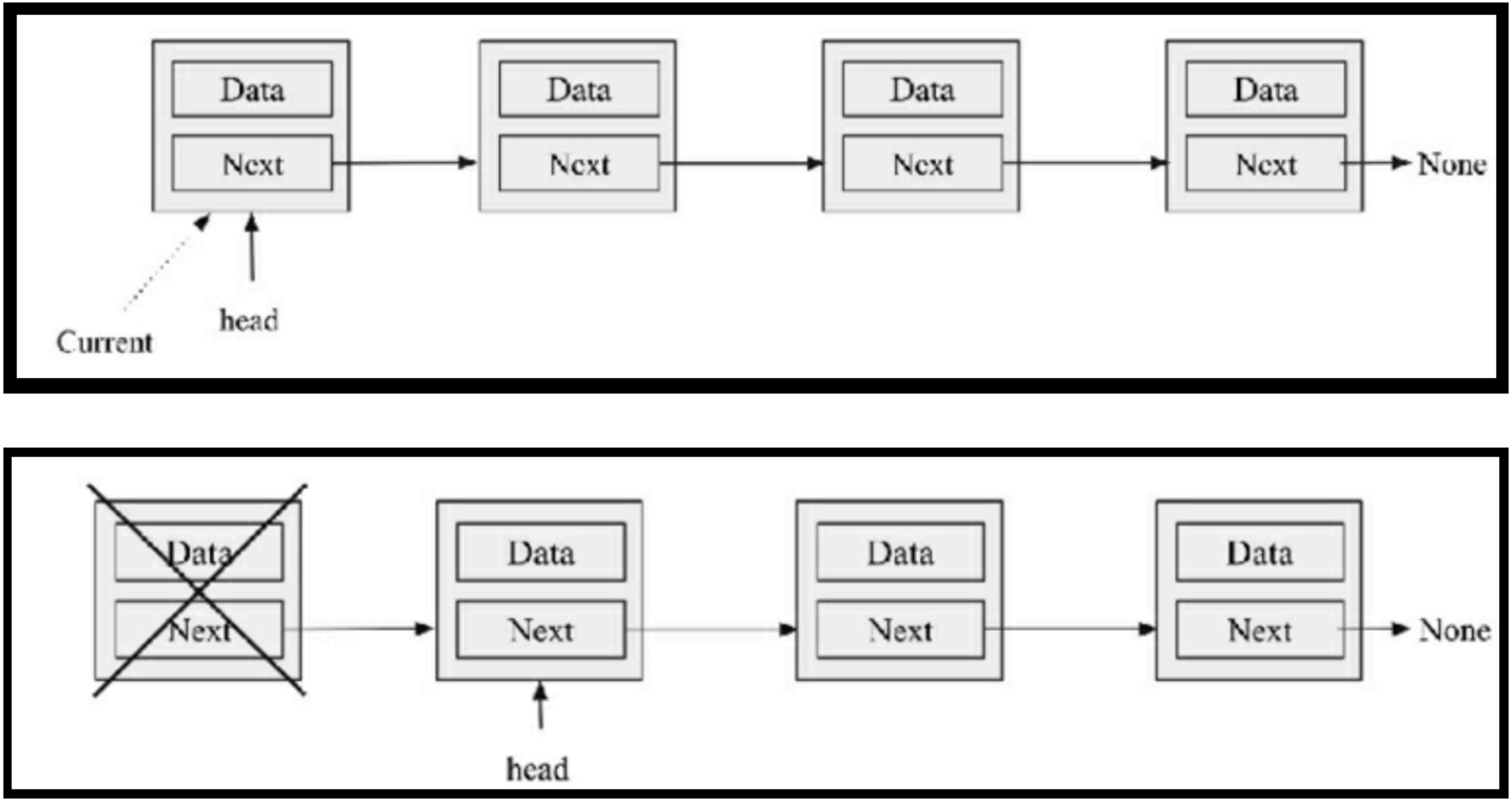
```
def size(self):
    count = 0
    current = self.head
    while current:
        count += 1
        current = current.next
    return count
```

- Or add a size attribute to the SinglyLinkedList class
    - O(***1***)

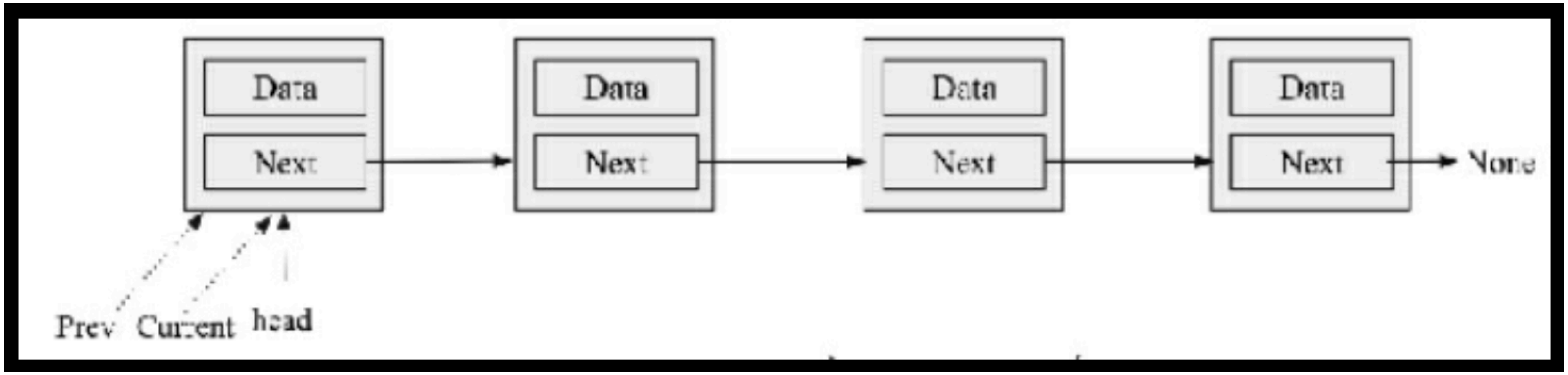


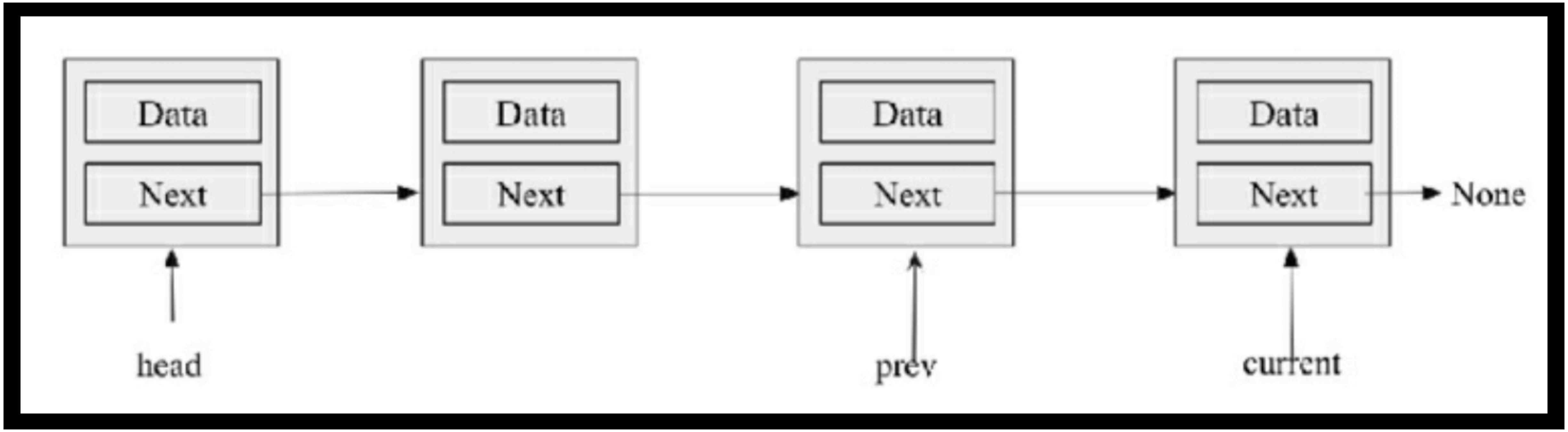
```
class SinglyLinkedList:
    def __init__(self):
        self.head = data
        self.size = 0
```

Deleting first node

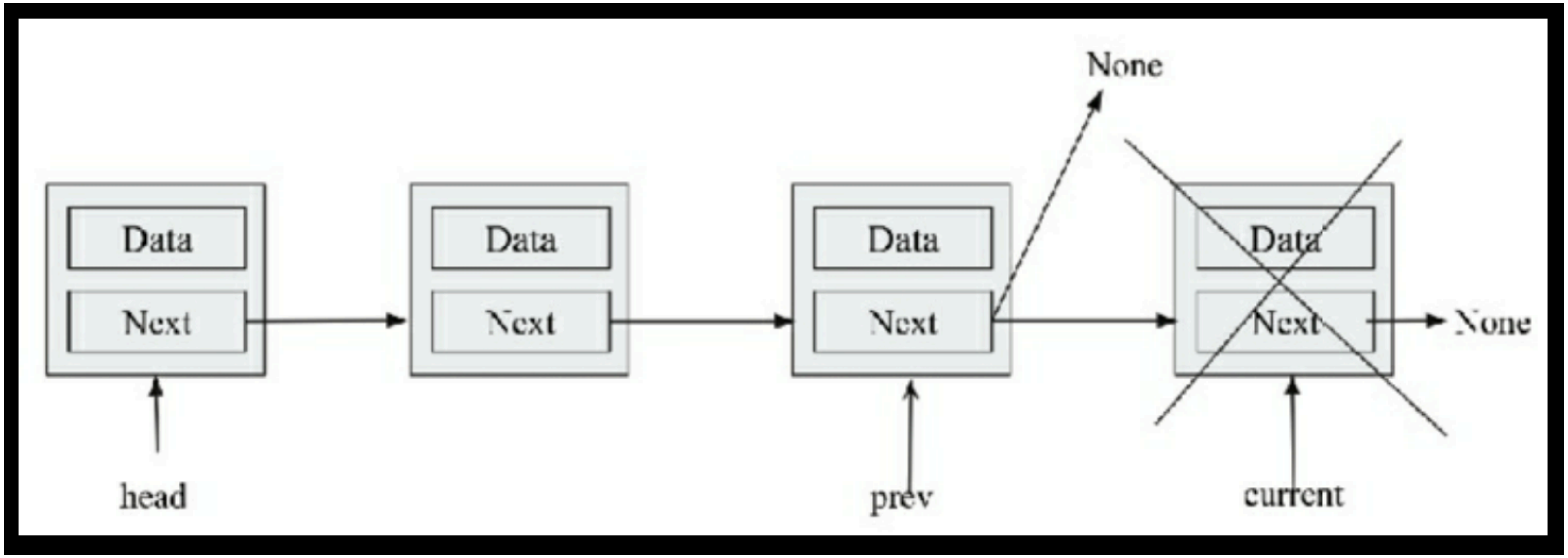


Deleting last node

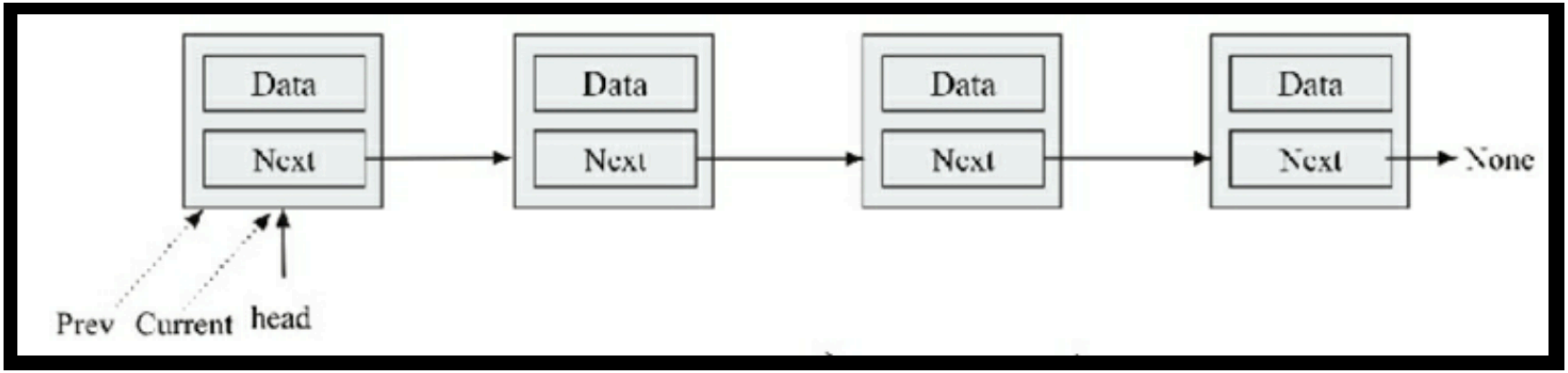


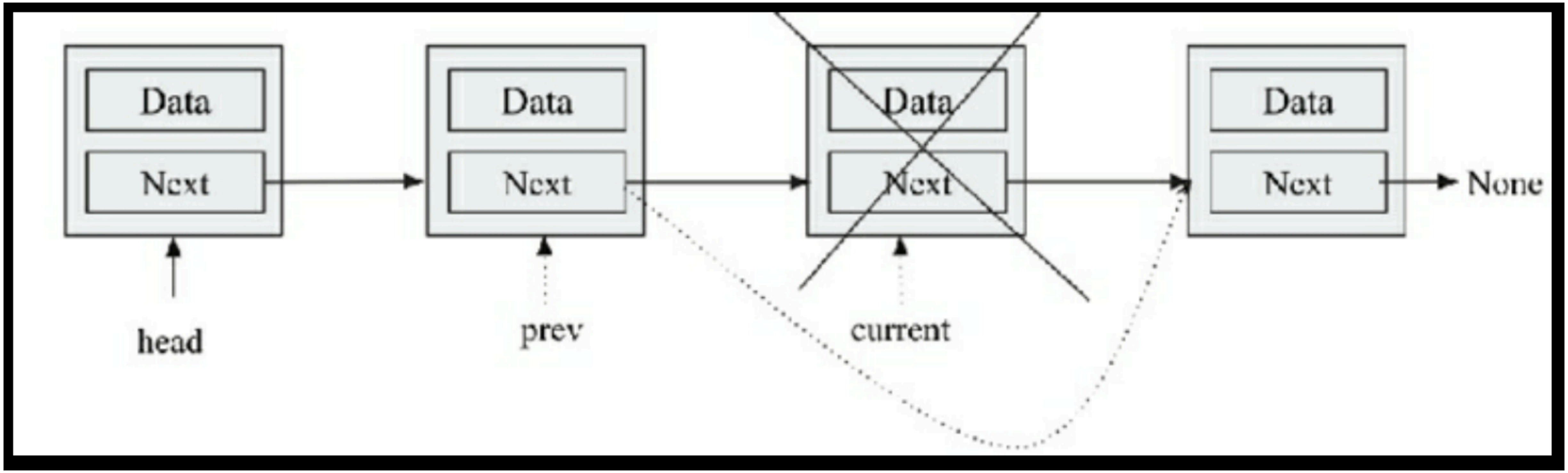


Deleting last node

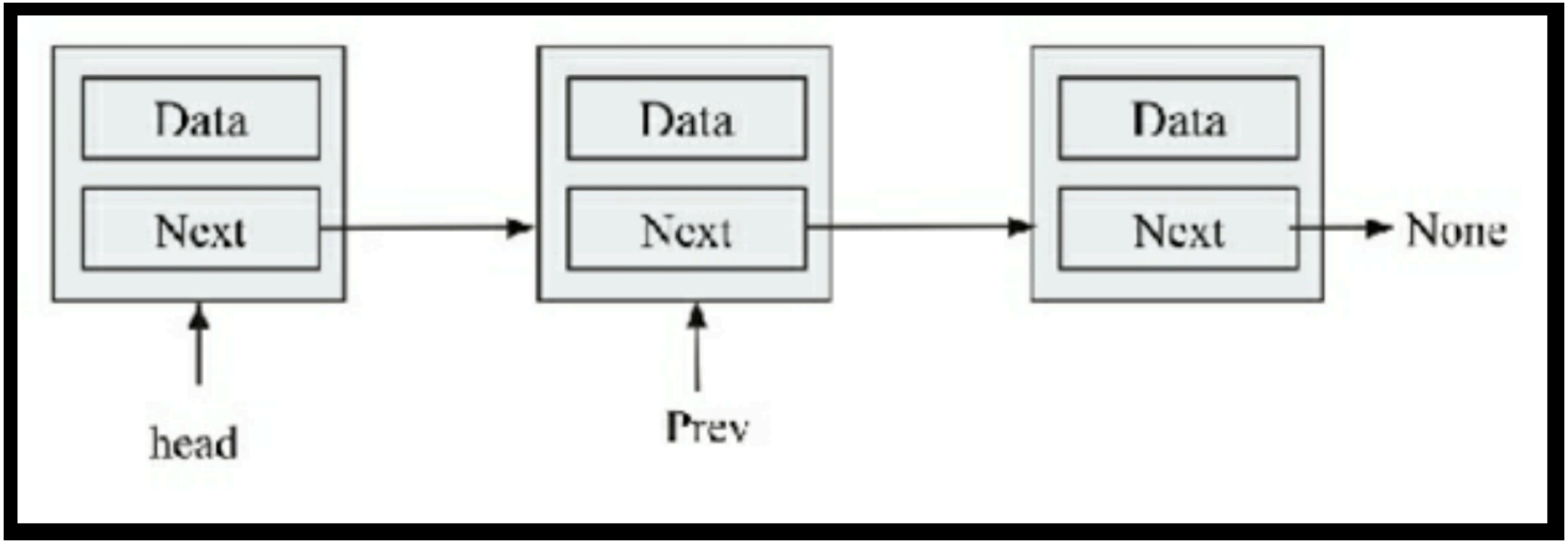


Deleting intermediate node





Deleting intermediate node

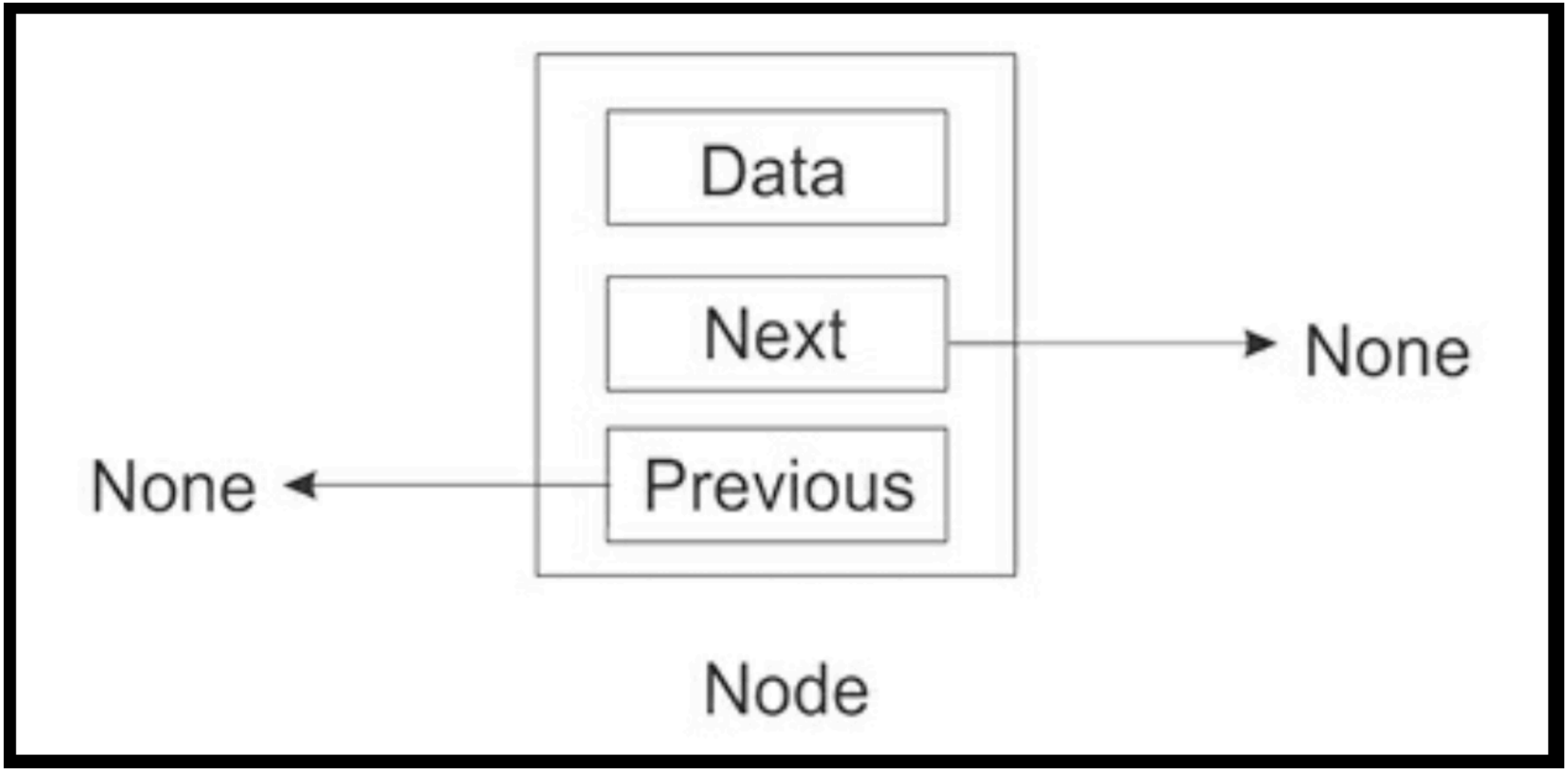


Clearing a list

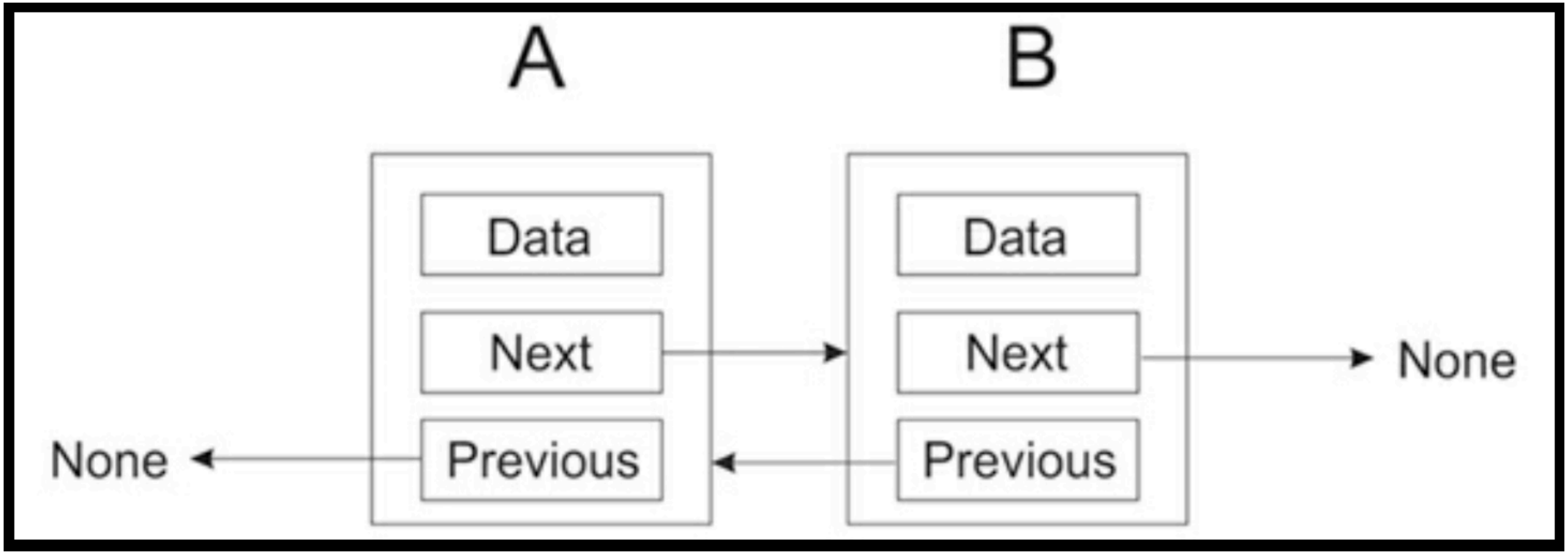
- Simply assign **None** to the tail and head pointers

Doubly linked lists

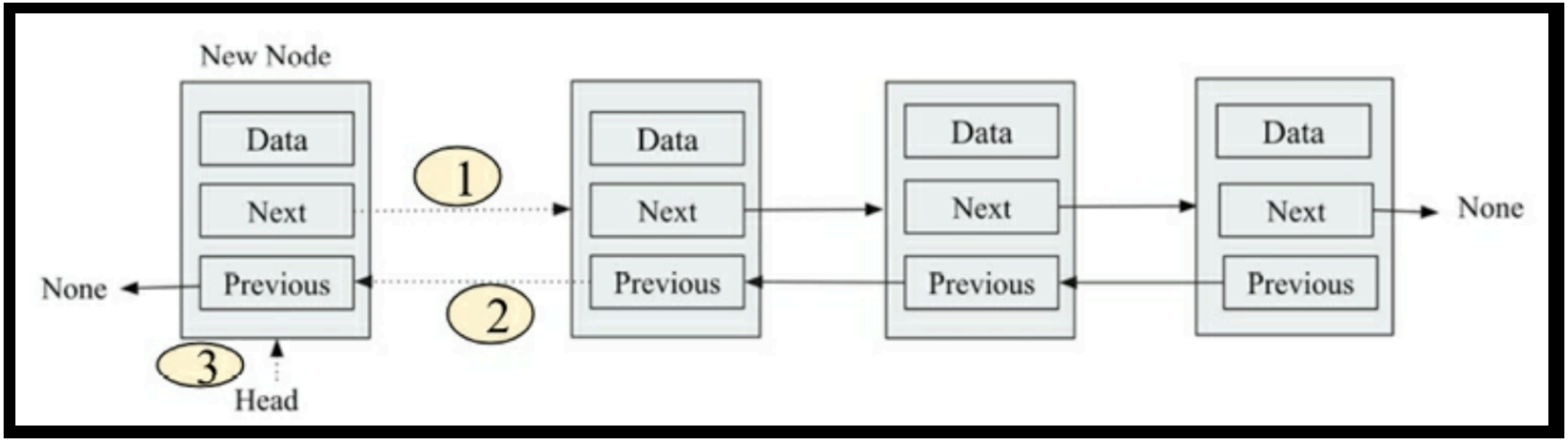
Doubly linked list with a single node



Doubly linked list with two nodes

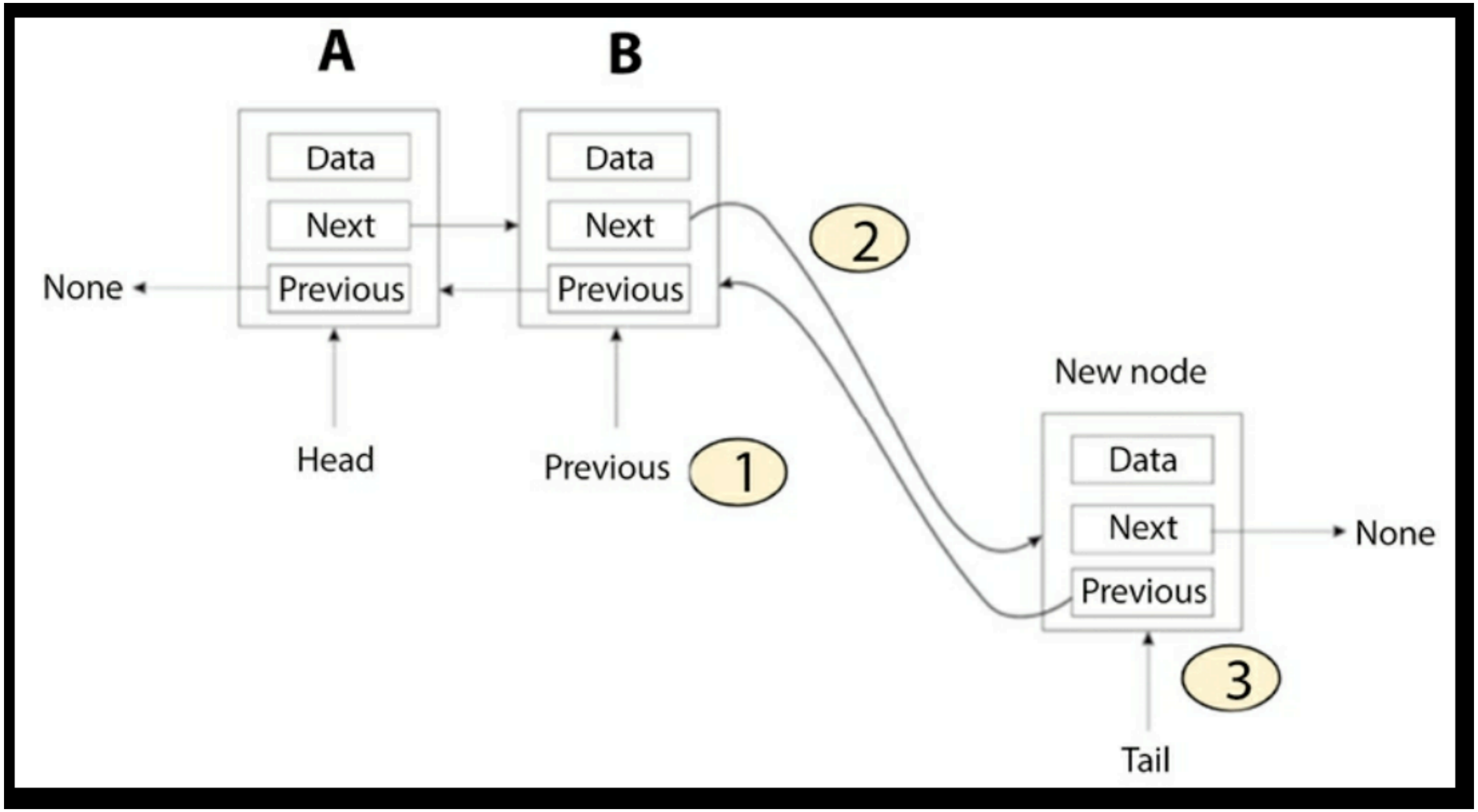


Inserting a node at the beginning



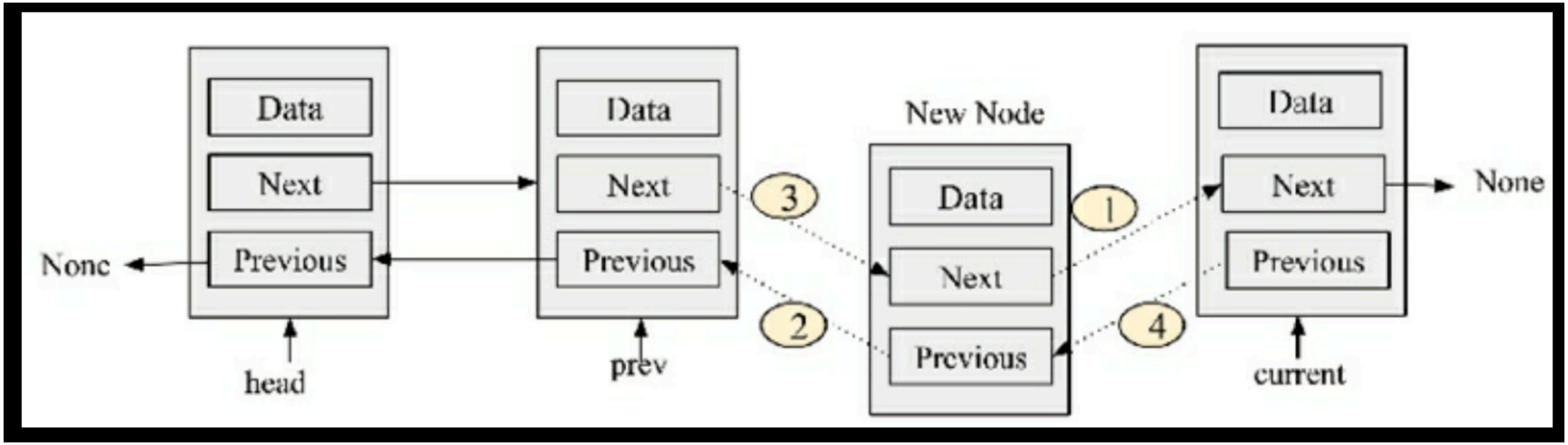
- Firstly, the next pointer of a new node should point to the head node of the existing list
- The prev pointer of the head node of the existing list should point to the new node
- Finally, mark the new node as the head node in the list

Inserting a node at the end

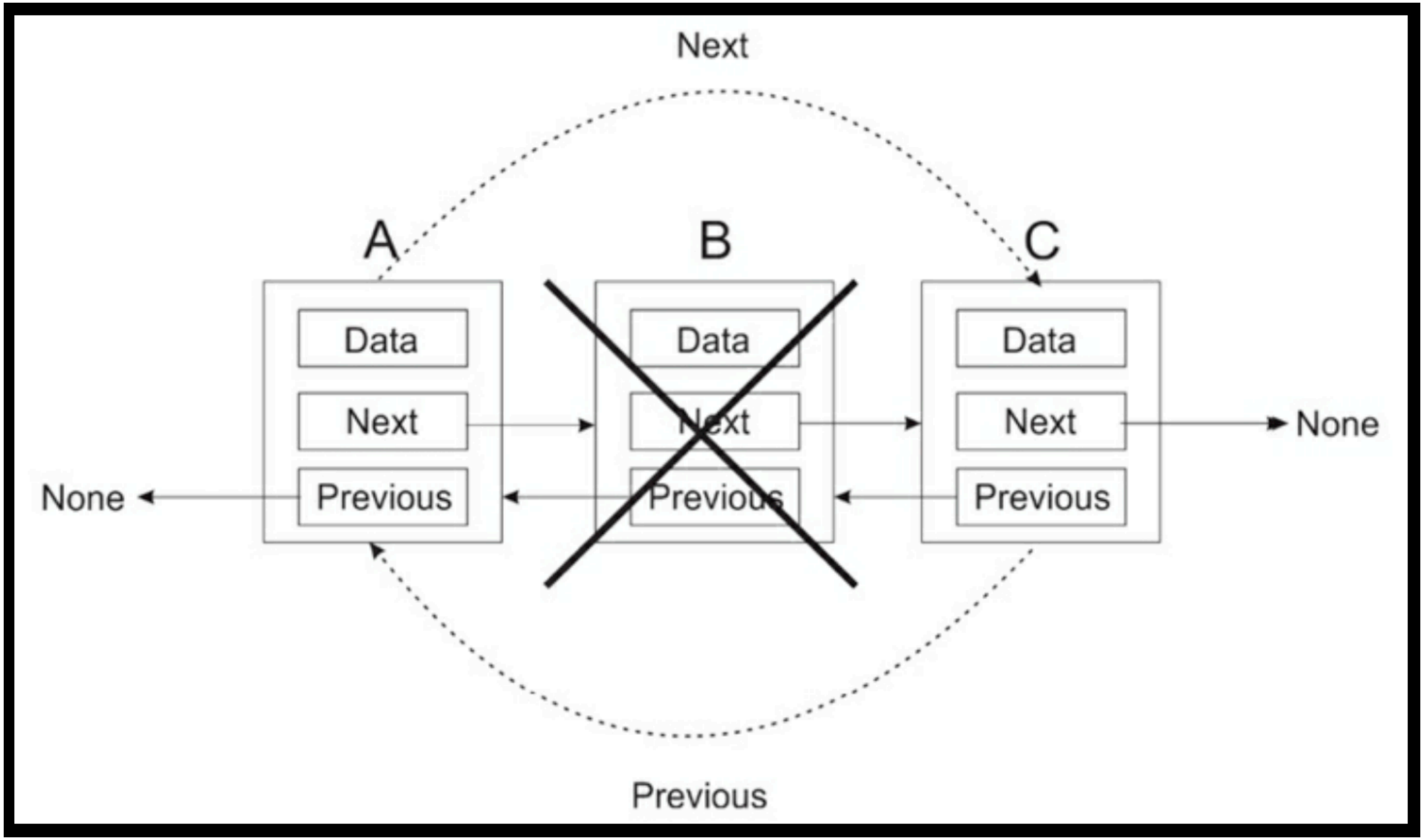


- Make the prev pointer of the new node point to the previous tail node
- Make the previous tail node point to the new node
- Finally, update the tail pointer so that the tail pointer now points to the new node

Inserting a node in the middle

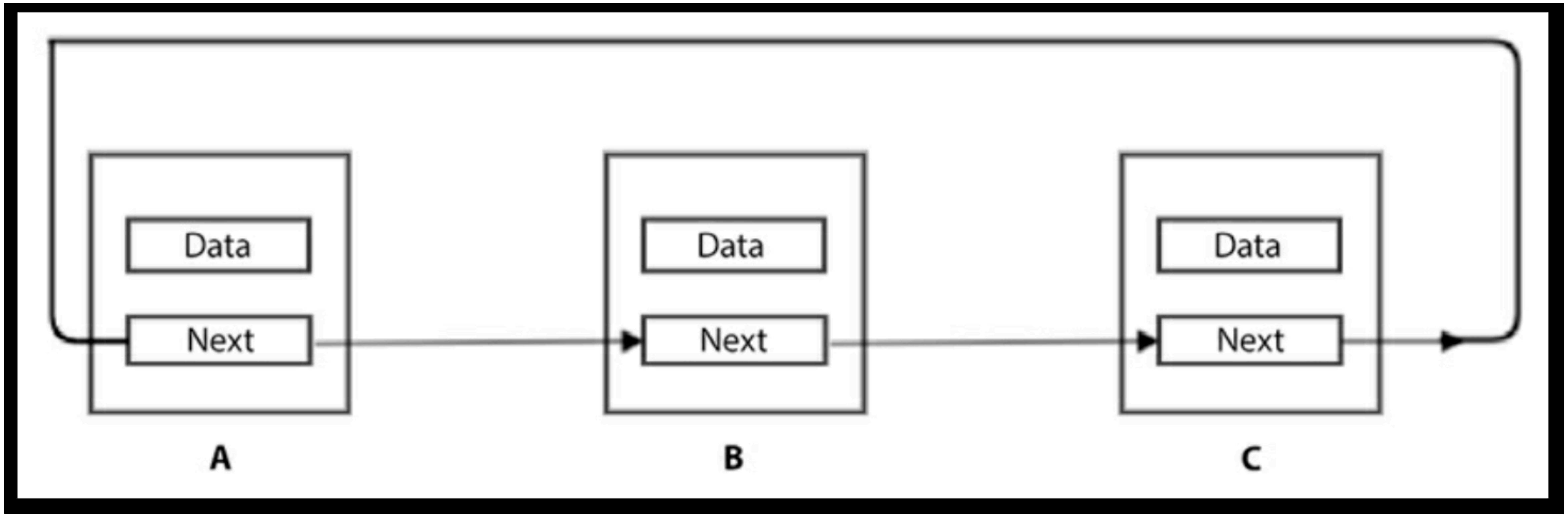


Deleting a node in the middle

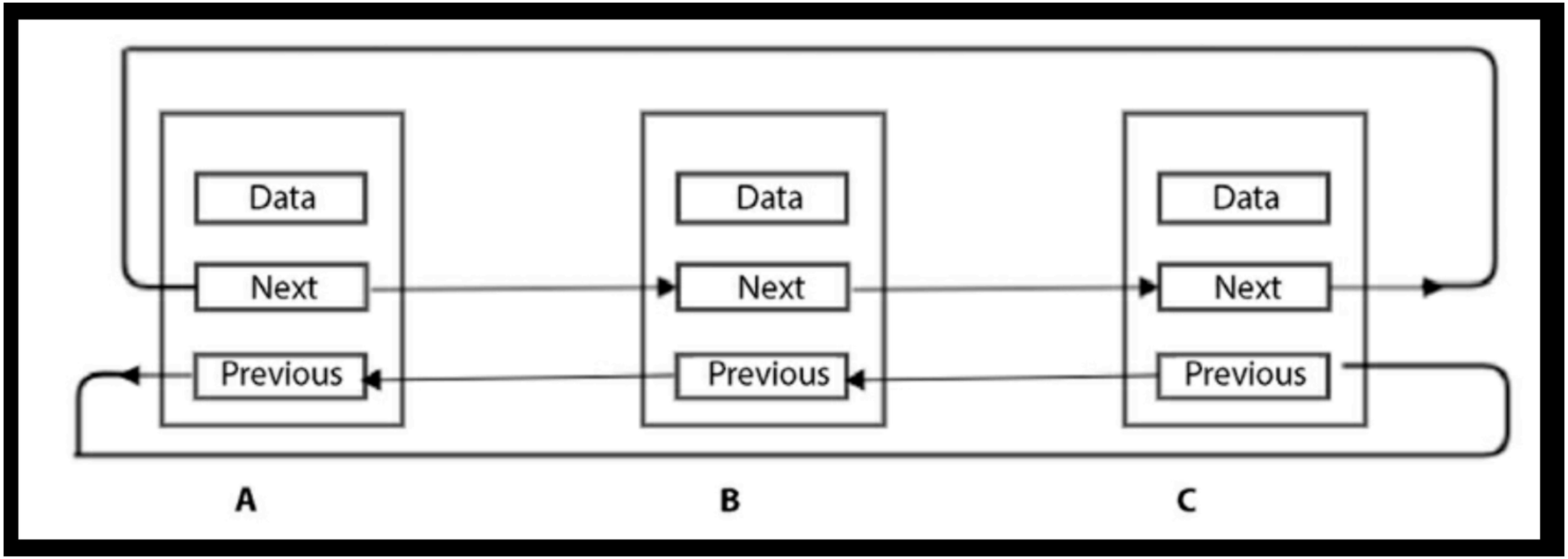


Circular lists

Circular list, singly-linked



Circular list, doubly-linked



Practical applications of linked lists

Applications

- Singly linked list
  - Represent a sparse matrix or a polynomial
  - Dynamically allocated memory (heap)
- Doubly linked list
  - Thread scheduler to maintain list of processes running
  - Most Recently Used (MRU) and Least Recently Used (LRU) caches in the OS
  - Undo and Redo functionality

Applications

- Circular linked list
  - Round-robin scheduling
  - Implement Undo or Redo in Word, or Back in a browser
  - Fibonacci heap
  - Multiplayer games swap between players in a loop

Kahoot!

Ch 4