Linked Lists

Thomas Schwarz SJ

Linked Lists

- Effective container data structure
  - Container ADT:
    - Insert, Update, Read, Delete records
      - Records can be anything
        - e.g. strings
        - e.g. BLOBs (binary large objects)

Linked Lists

- Linked Lists consists of Nodes
  - Nodes are connected by forward **pointers**
  - And have **pointers** to the record contents
- Pointers: Addresses of objects in memory
  - This is how Python finds objects
    - E.g. if you define a class without __str__ and __repr__ dunder, this is how an instance will be printed
      - It prints the id of the object
        - CPython: Guaranteed to be the memory location
        - Unless ...

Linked Lists

```
>>> class X:
    def __init__(self):
        self.x = 0
```

```
>>> x=X()
>>> print(x)
<__main__.X object at 0x7facf6f1cee0>
>>>
```

Linked Lists

- Implementing a node
  - Need fields for
    - next node (by default Null)
    - object stored in linked list

Linked Lists

```
class Node:
    def __init__(self, my_record):
        self.next_node = None
        self.record = my_record
```

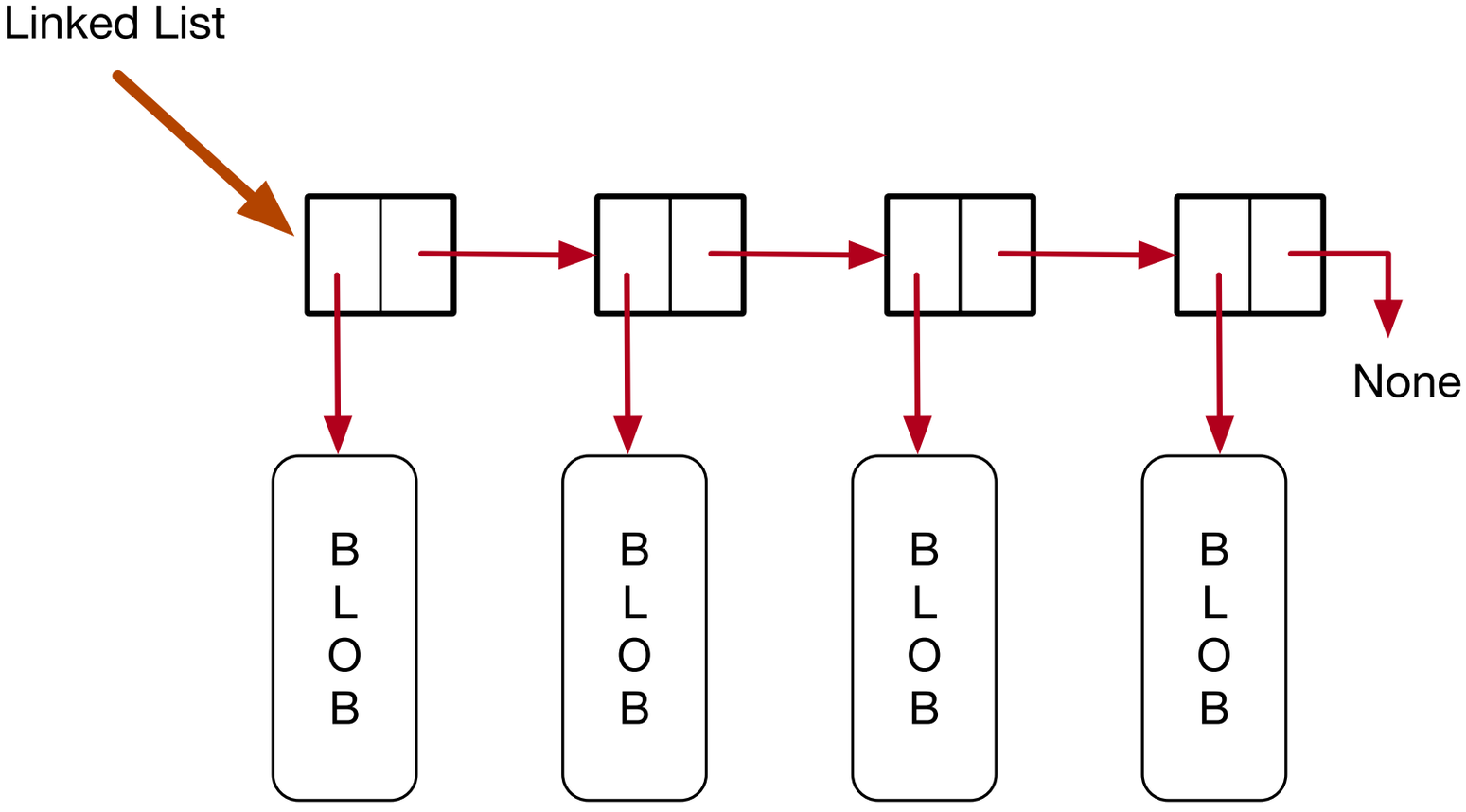
Linked Lists

```
class Node:
        def __repr__(self):
         string = "Class Node "
         string += str(id(self))
         string += ", next is " + str(id(self.next_node))
         string += ", record is " + str(self.record)
         return string
```

- We use the id-trick in order to find the memory address of the nodes in CPython

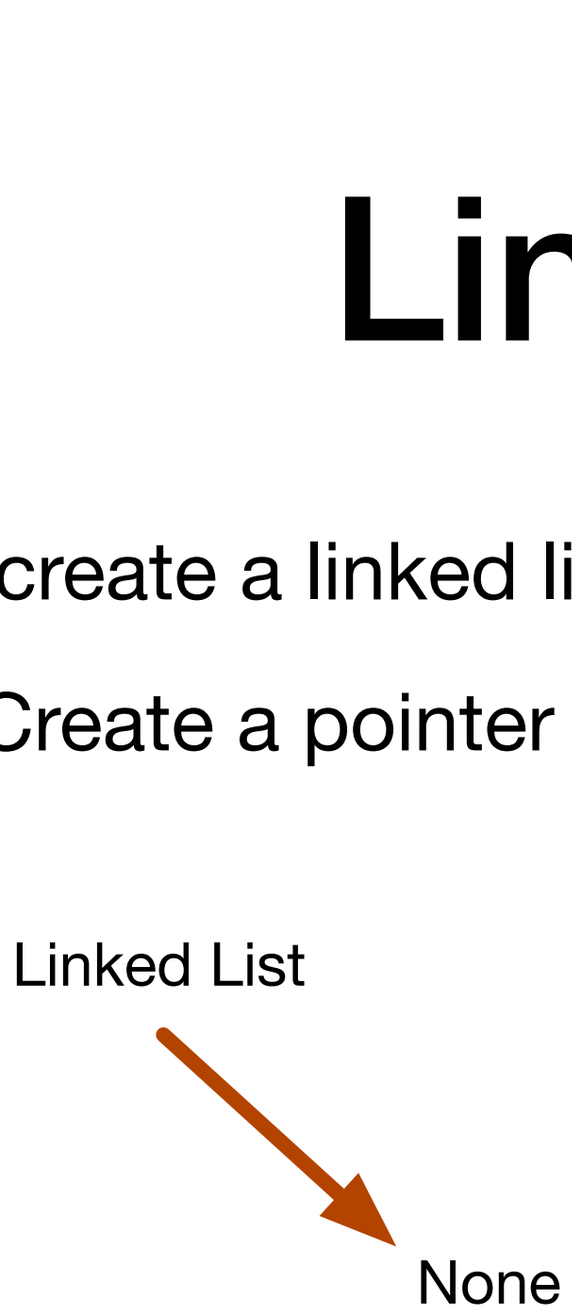
Linked Lists

- Next task is to link the nodes
- A linked list is given by its initial node, the *head*



Linked Lists

- To create a linked list:
  - Create a pointer to null

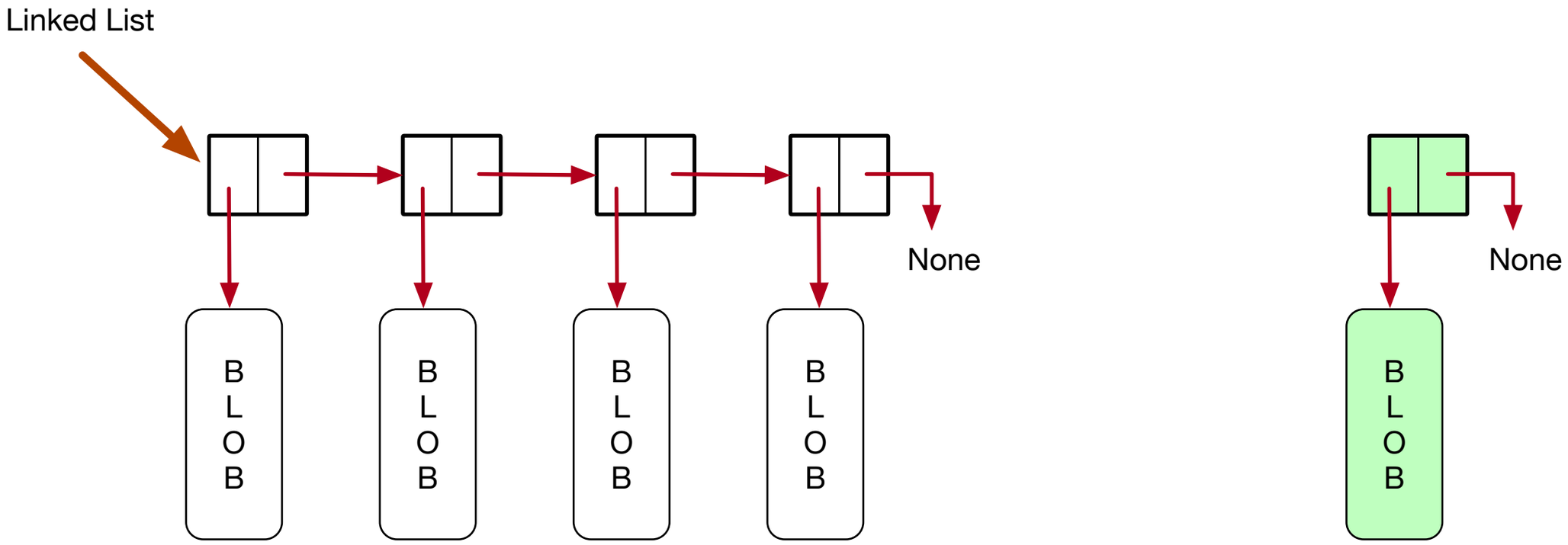



```
class Linked_List:
    def __init__(self):
        self.head = None
```

Linked Lists

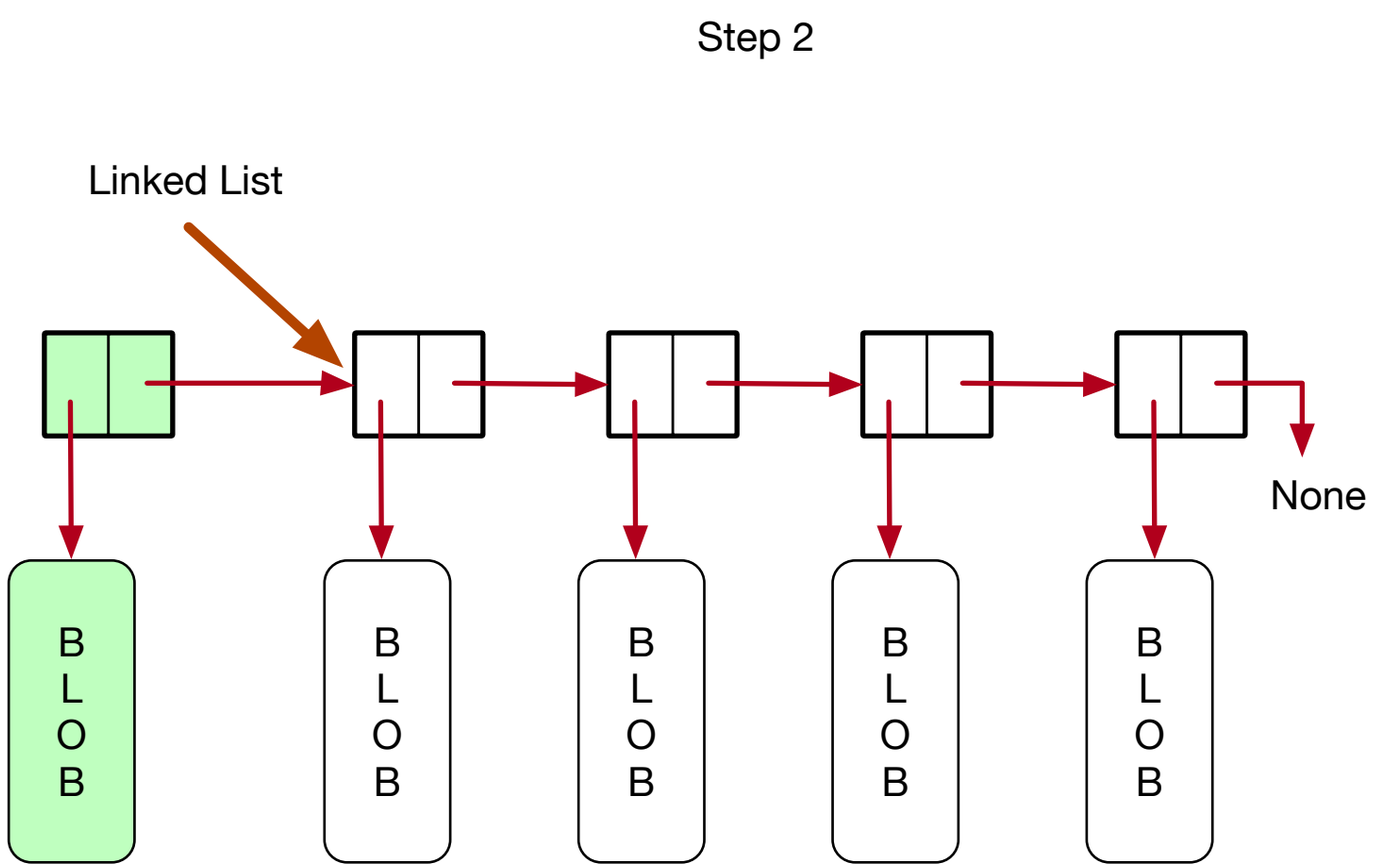
- We insert at the front of a linked list
  - Step 1: We create a node that points to the record that we want to insert

Step 1



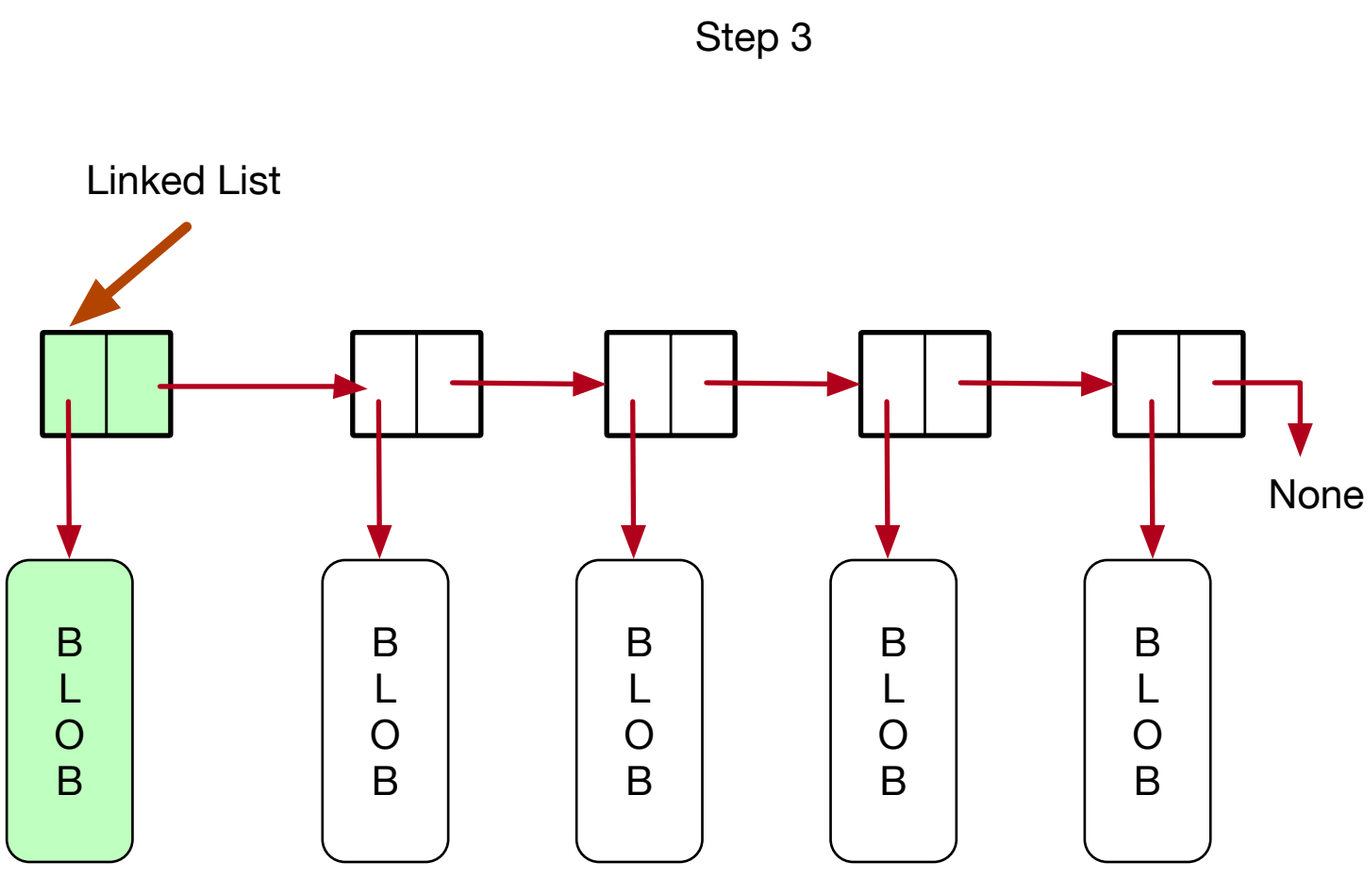
Linked Lists

- Step 2: We have the new node point to the same node as the linked list does



Linked Lists

- Step 3: We reset the head of the linked list

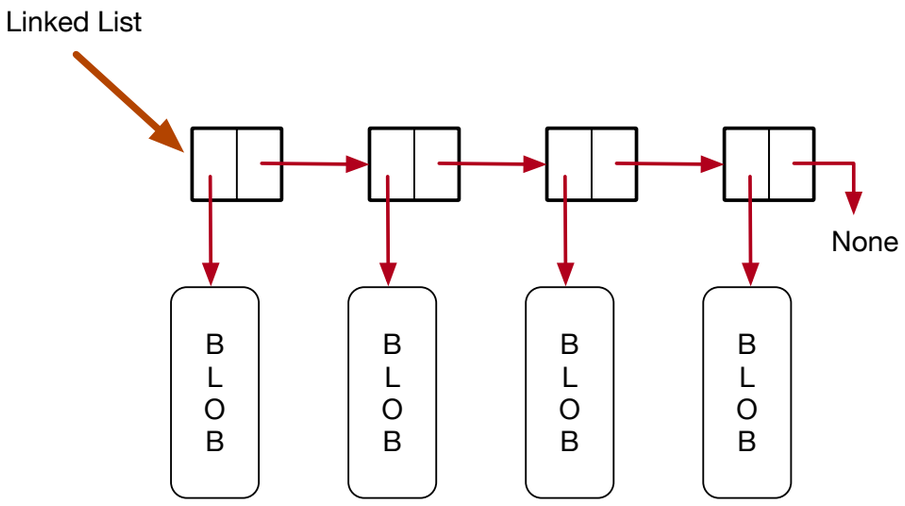


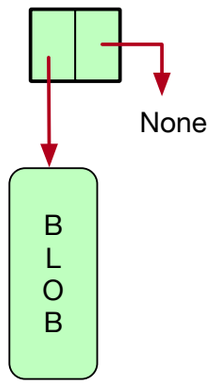
Linked List

- Python Code

```
class Linked_List:
    ...
    def insert(self, record):
        new_node = Node(record)
        new_node.next_node = self.head
        self.head = new_node
```

Step 1



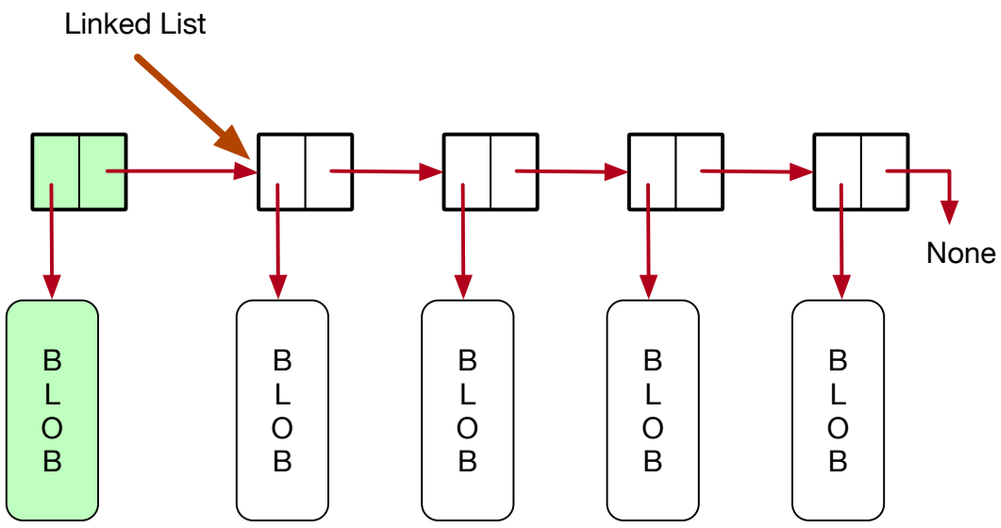


Linked List

- Python Code

```
class Linked_List:
    ...
    def insert(self, record):
        new_node = Node(record)
        new_node.next_node = self.head
        self.head = new_node
```

Step 2

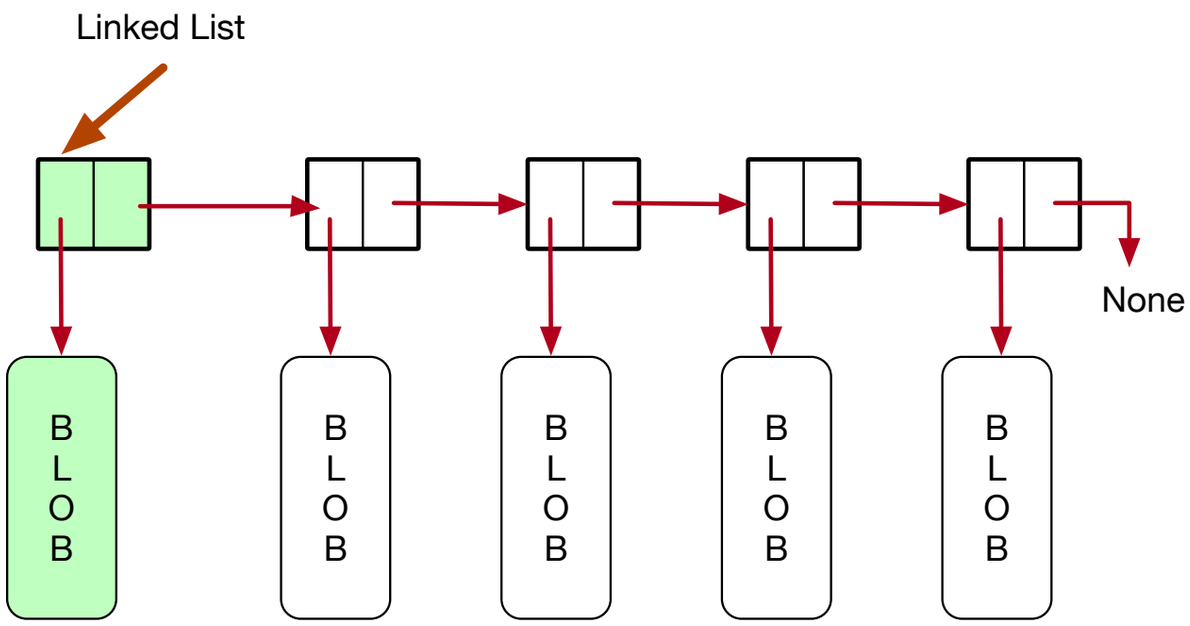


Linked List

- Python Code

```
class Linked_List:
    ...
    def insert(self, record):
        new_node = Node(record)
        new_node.next_node = self.head
        self.head = new_node
```

Step 3

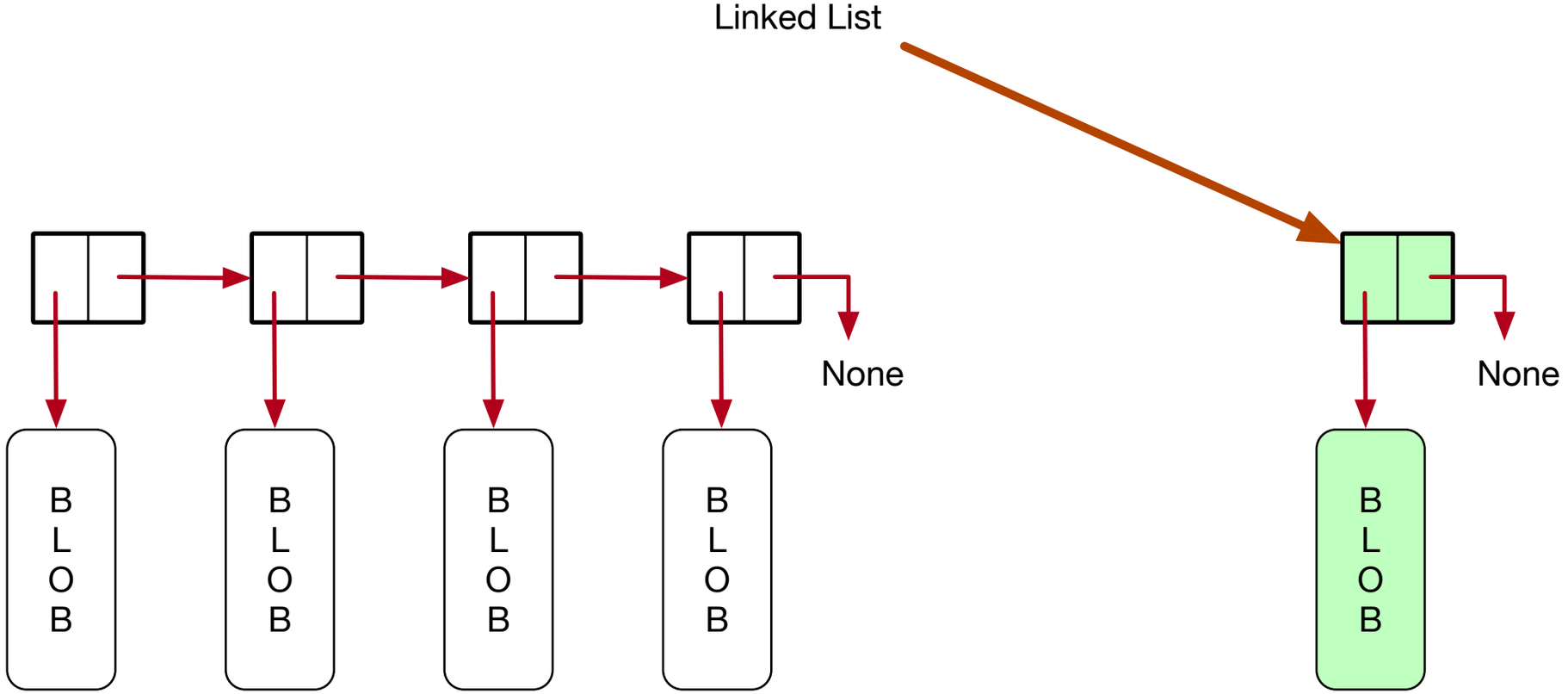


Linked List

- Why do we execute Step 2 before Step 3
  - Many programming languages support threads
    - Independent streams of instructions accessing the same memory structure
    - Not really well implemented in Python
- If we set the head to the new node, there is a moment where the linked list only contains the new node

Linked List

- Doing Step 3 before Step 2 leads to a temporary bad state

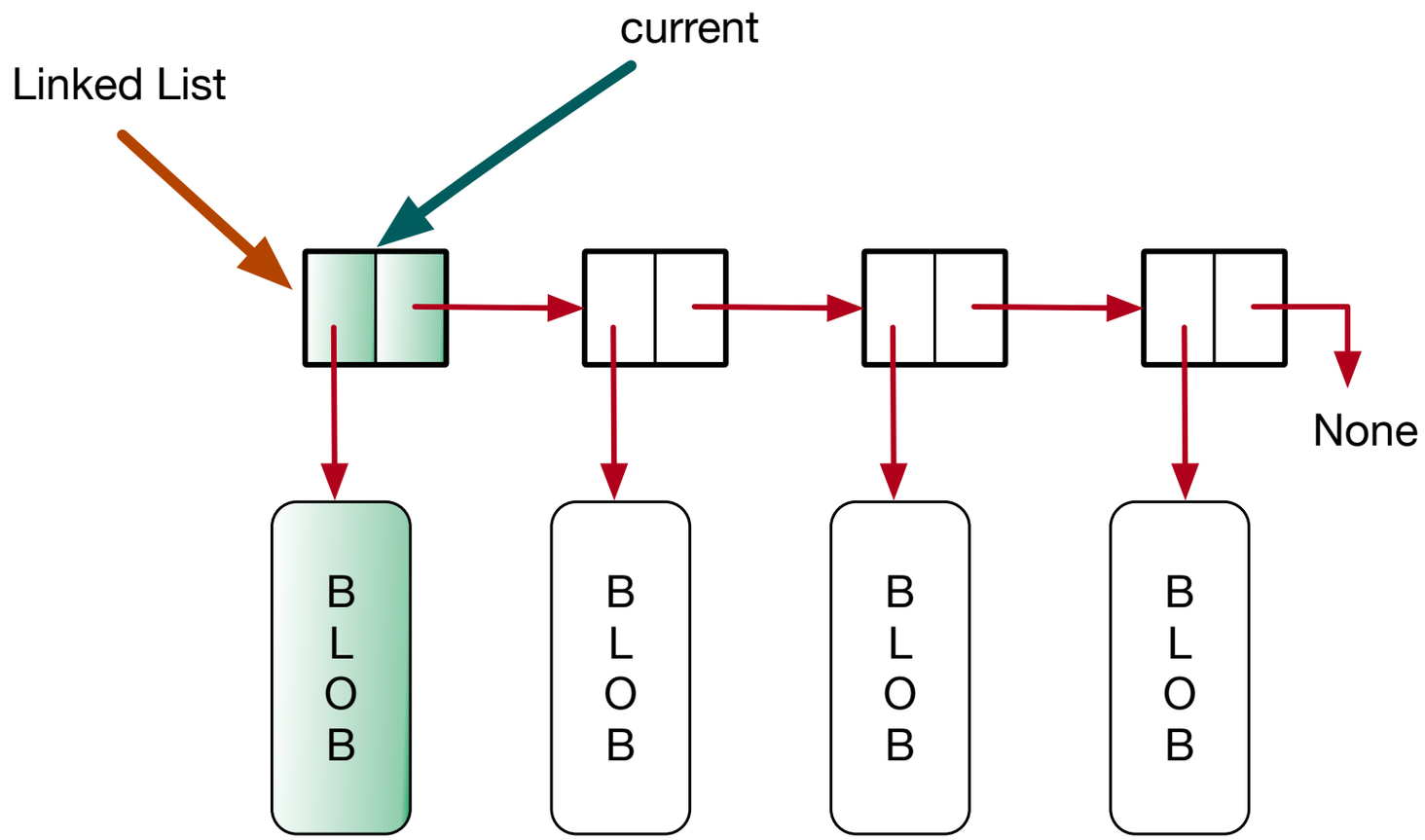


Linked List

- Printing out the contents of a linked list
  - Idea: Create a node pointer current
  - Let current walk through the list of nodes
    - At each step, print out the contents of the record

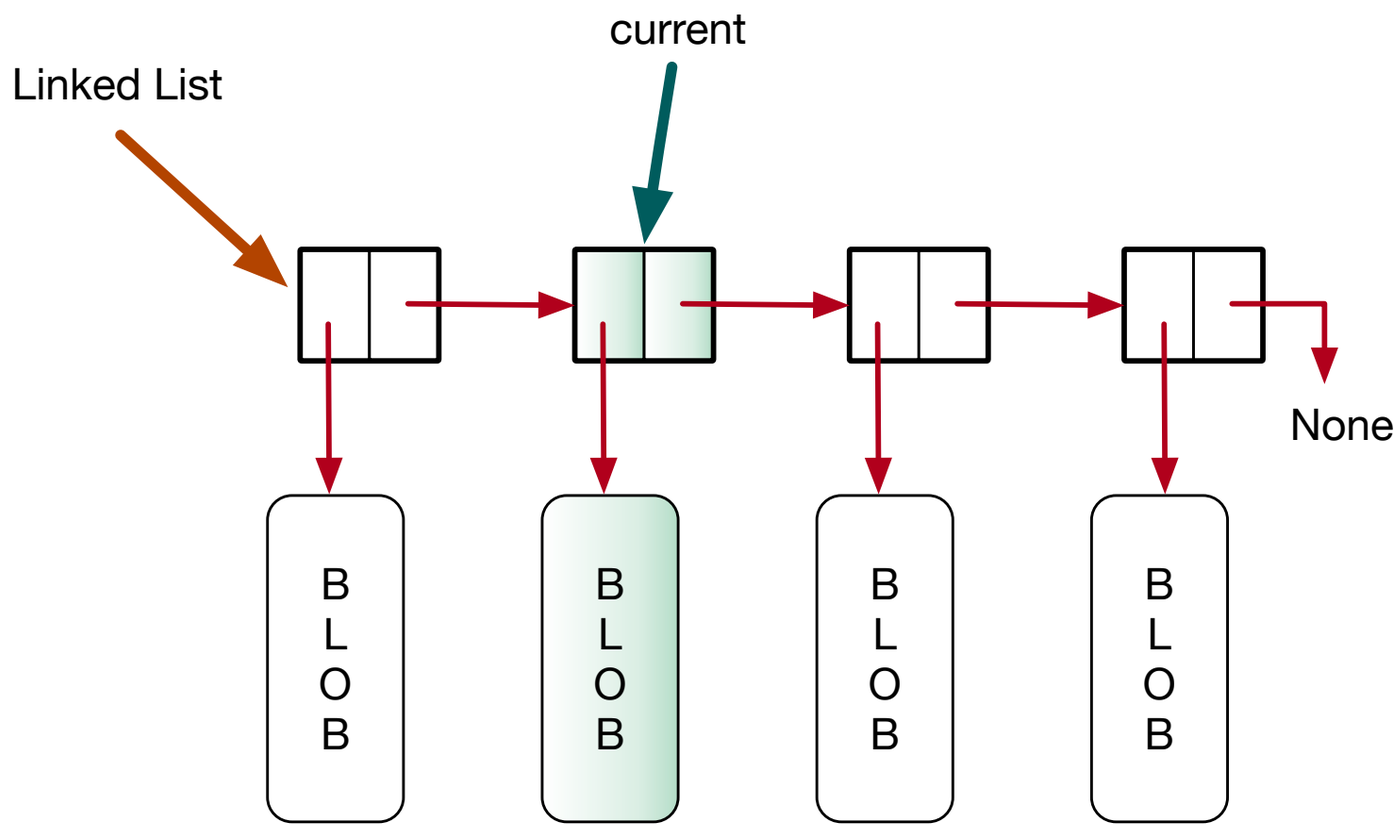
Linked List

- Set current_node to the head



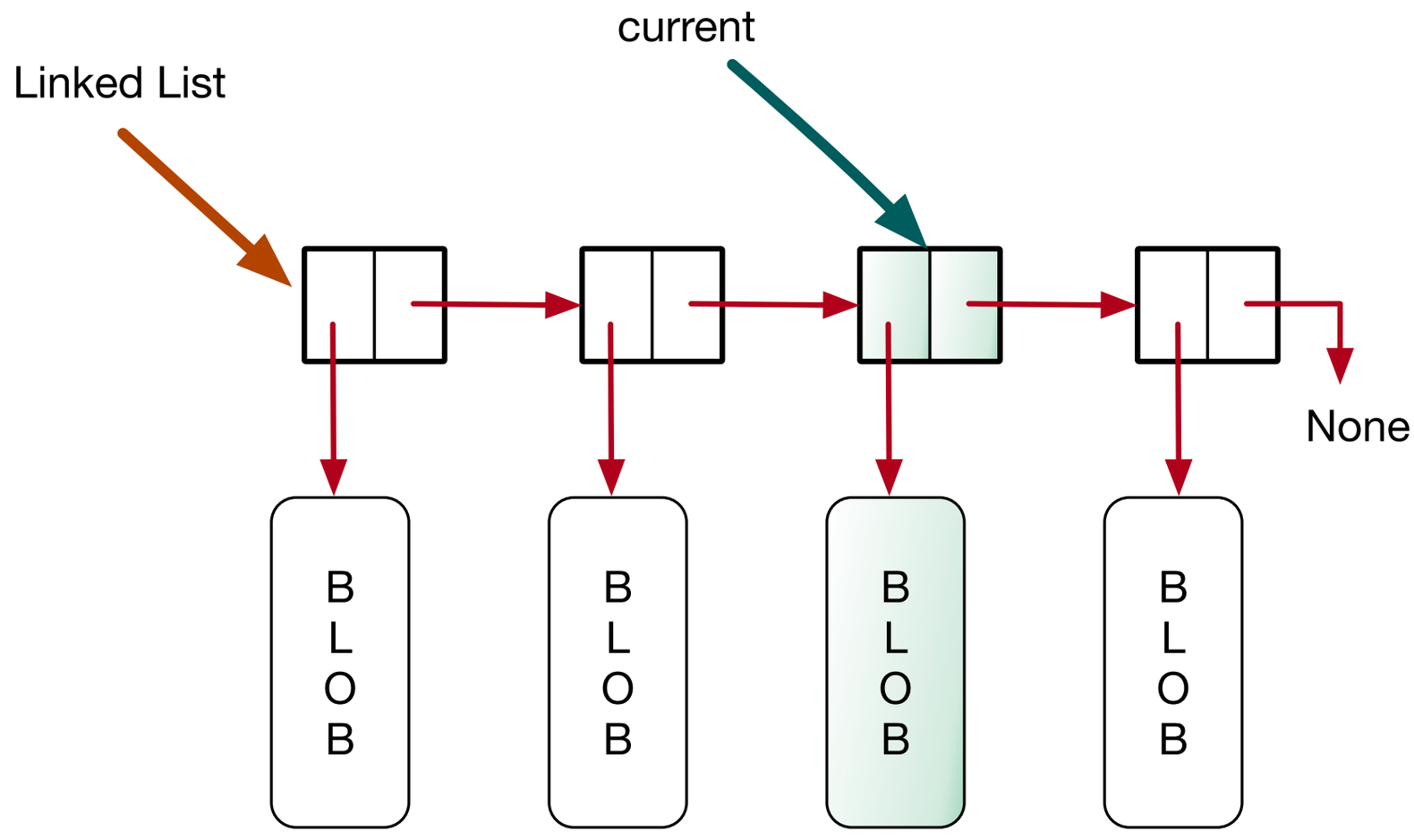
Linked List

- Set `current_node` to `current_node.next_node`



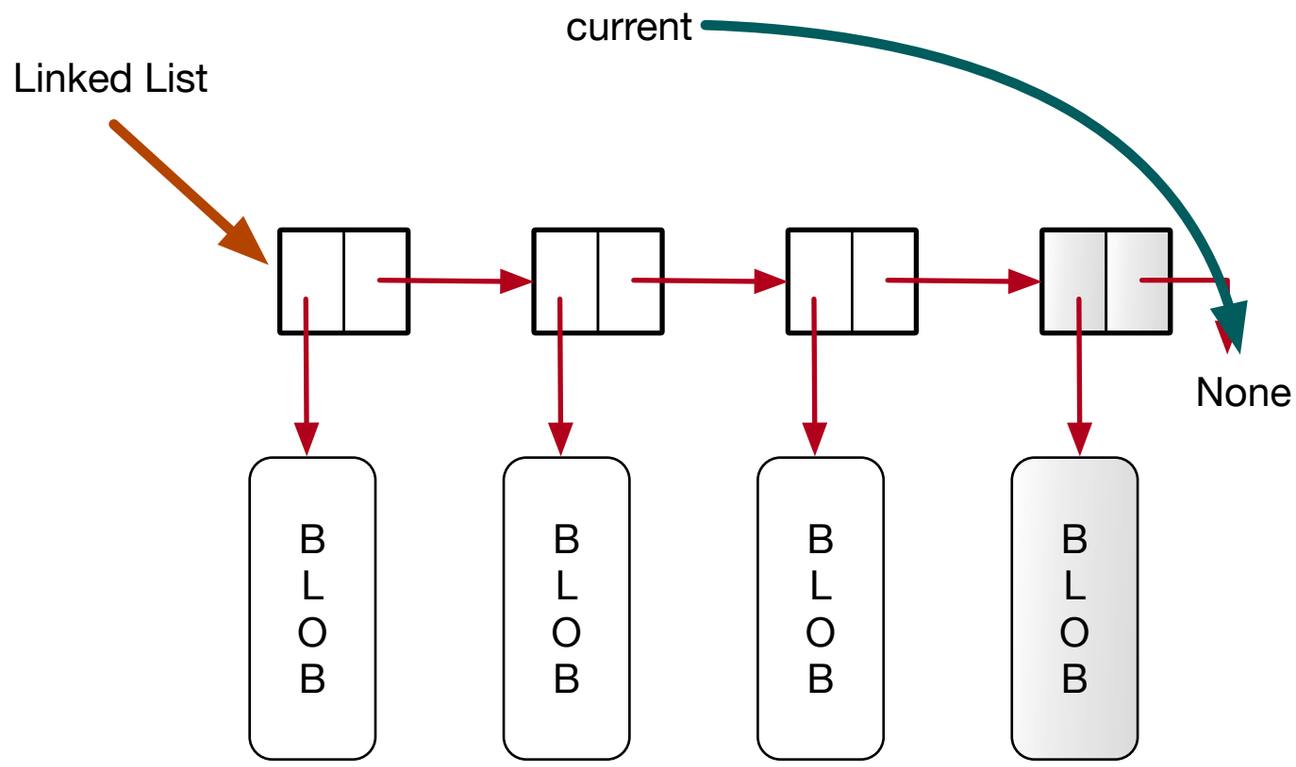
Linked List

- And so on



Linked List

- Until we hit the end, i.e. until `current_node` becomes `None`



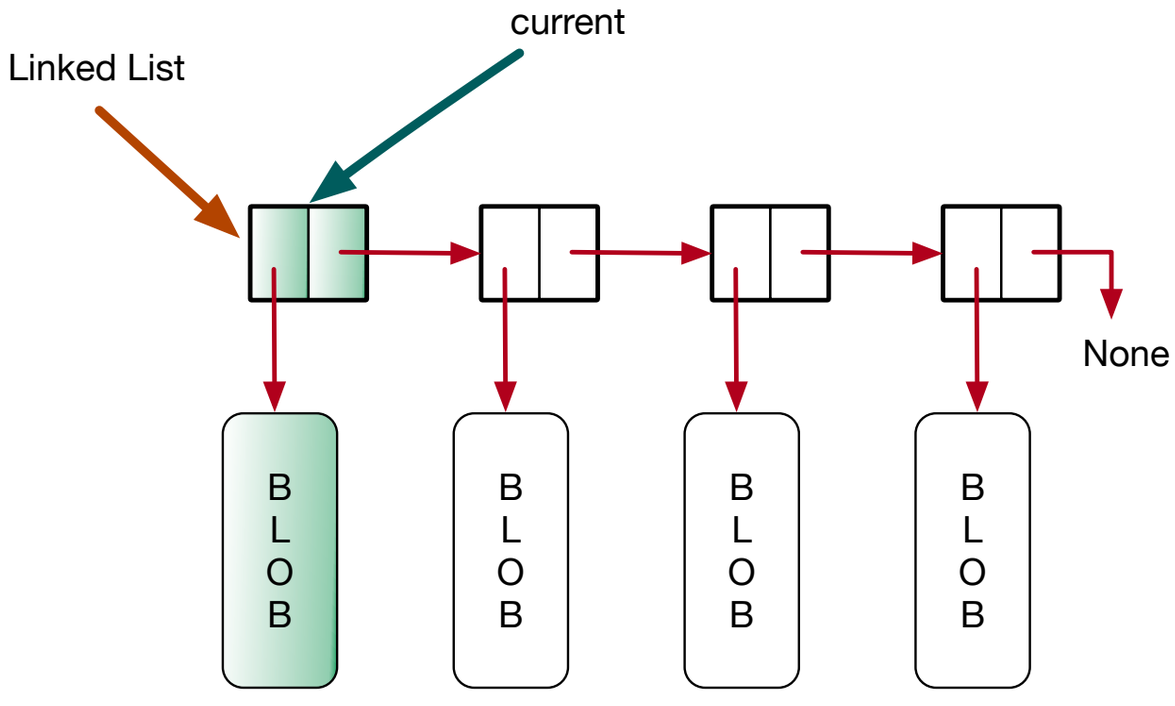
Linked List

- Python code:
  - Use a `current_node` pointer
  - Adjust `current_node` pointer to `current_node.next_node`
  - Repeat until `current_node` becomes `None`

```
def __str__(self):
        output = '['
        current_node = self.head
        while current_node:
            output += str(current_node.record) + ', '
            current_node = current_node.next_node
        return output + ']'
```

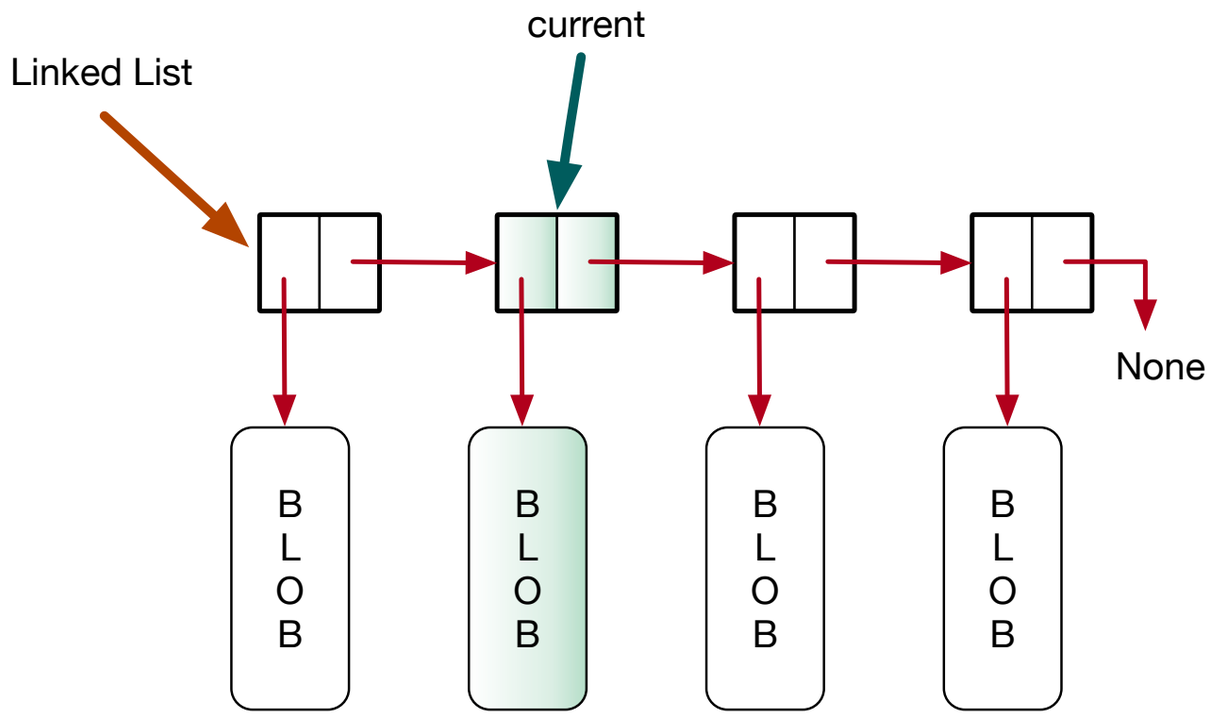
Linked List

```
def __str__(self):
        output = '['
        current_node = self.head
        while current_node:
            output += str(current_node.record) + ', '
            current_node = current_node.next_node
        return output + ']'
```



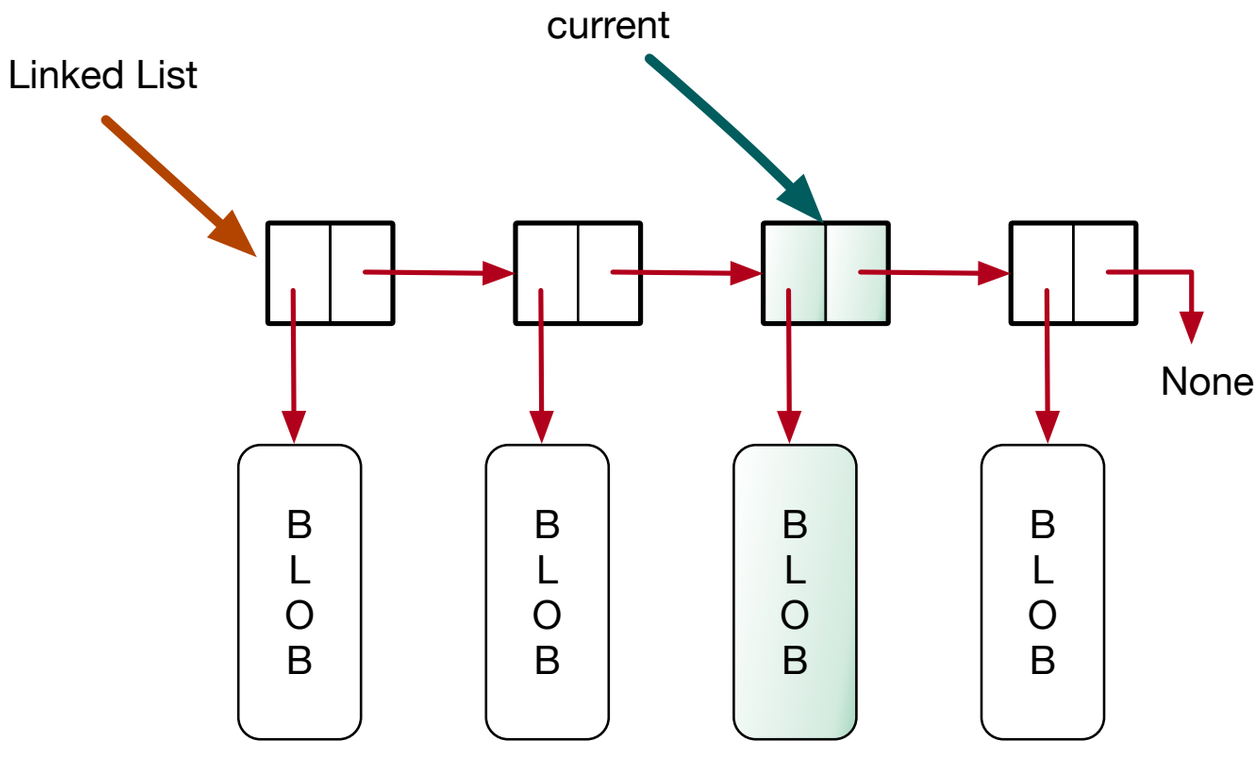
Linked List

```
def __str__(self):
        output = '['
        current_node = self.head
        while current_node:
            output += str(current_node.record) + ', '
            current_node = current_node.next_node
        return output + ']'
```



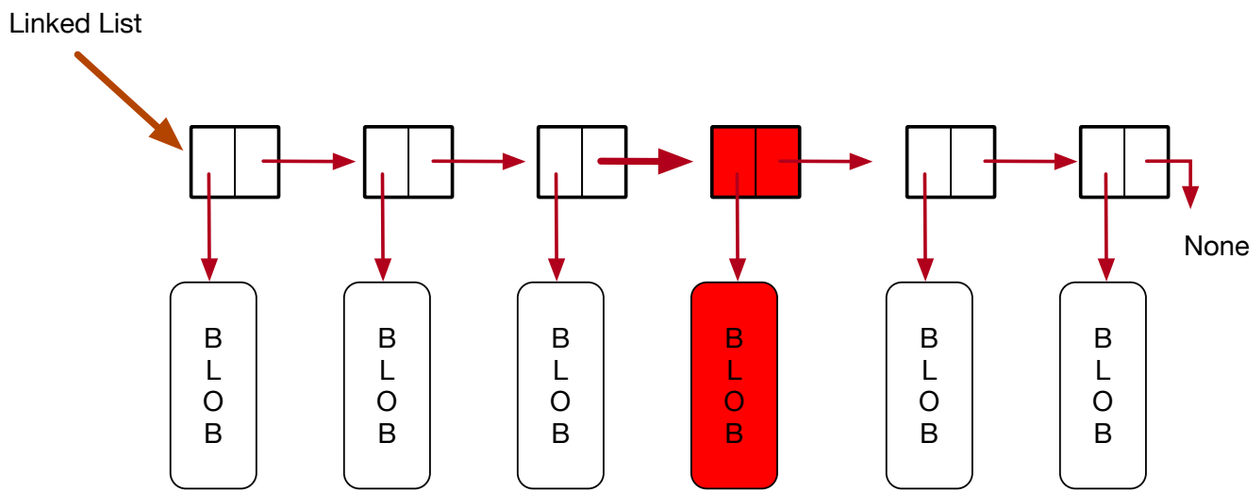
Linked List

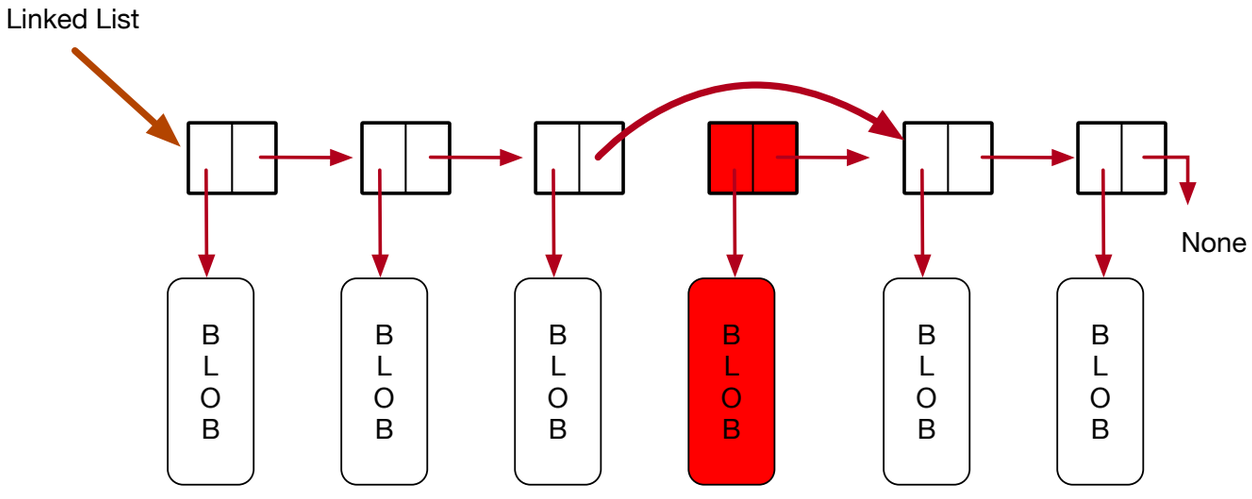
```
def __str__(self):
        output = '['
        current_node = self.head
        while current_node:
            output += str(current_node.record) + ', '
            current_node = current_node.next_node
        return output + ']'
```



Linked List

- Deleting from a linked list
  - We need to reset the next link of the node to the right of the one to be deleted

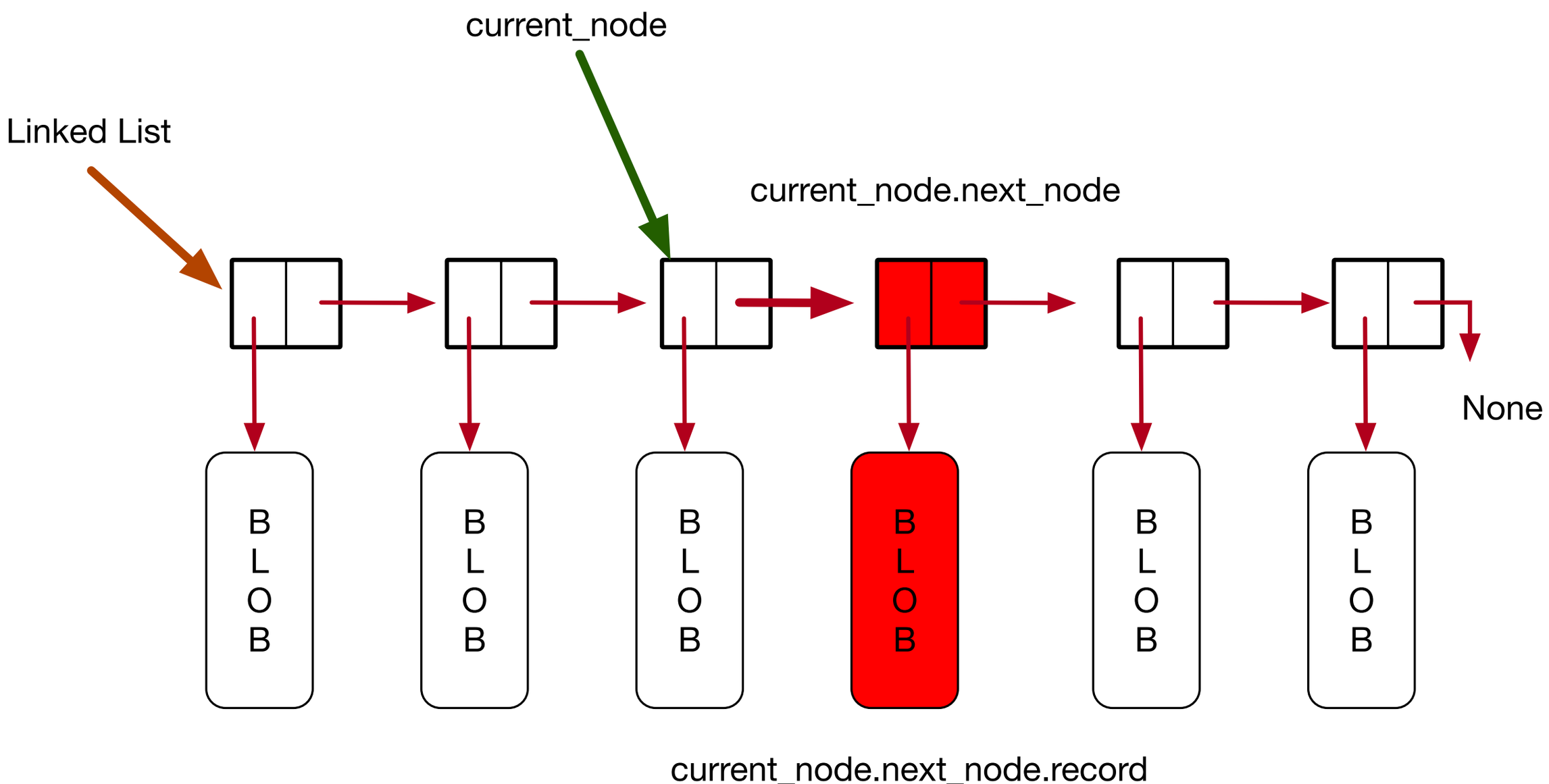




Linked List

- To do so, we need to look ahead
  - We use a current_node pointer as usual
  - We advance, until current_node.next_node.record has the record to be deleted.

Linked List


current_node
Linked List
current_node.next_node
None
B
L
O
B
B
L
O
B
B
L
O
B
B
L
O
B
B
L
O
B
B
L
O
B
current_node.next_node.record

Linked List

- There are a number of special cases
  - The linked list is empty
  - The record to be deleted is in the first node
  - The record to be deleted does not exist
  - The record to be deleted is in the last node
    - This is actually not a problem
- You need to check your algorithm with all these cases

Linked List

- Another problem:
  - Should we return a value
    - One possibility:
      - Return True if we remove a record
      - Return False if we do not, because there was no such record

Linked List

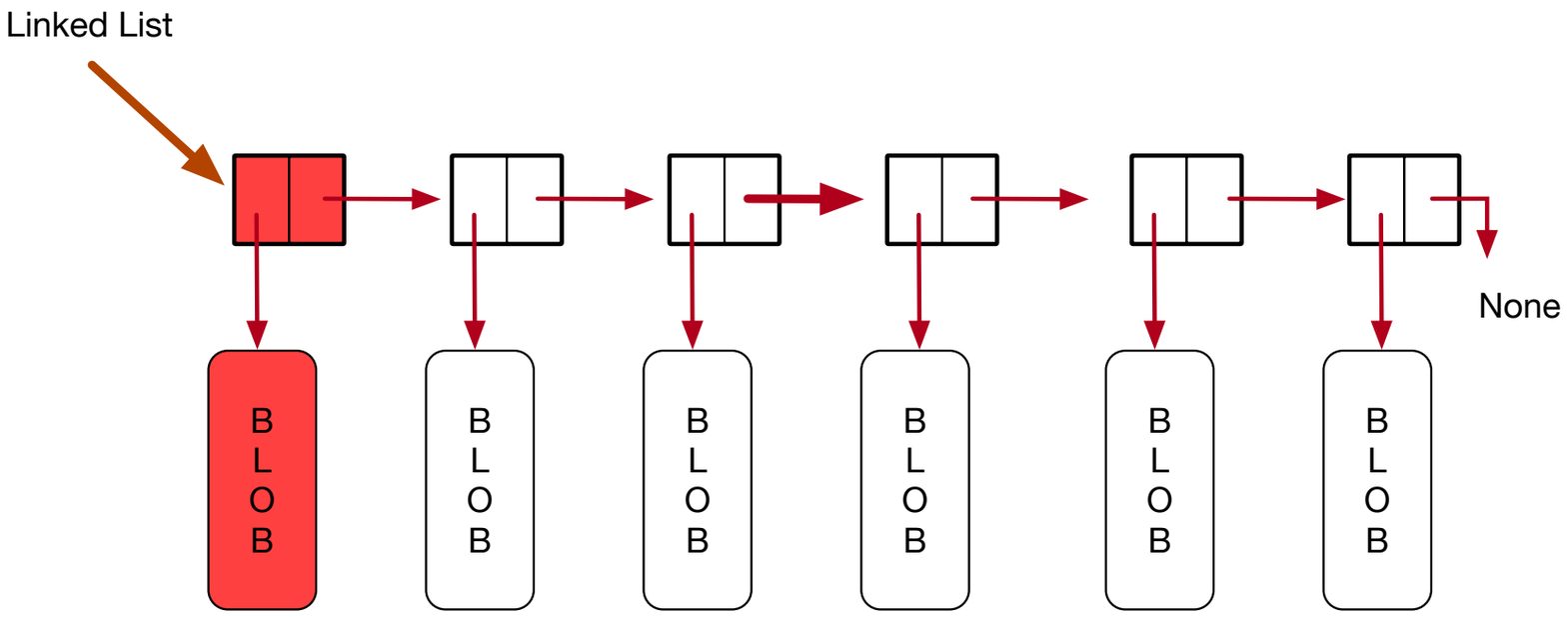
- If the linked list is empty, return False

```
def remove(self, record):
        if not self.head:
            return False
```

Linked List

- `current_node` points to the node before the one to be deleted
- This is impossible if the record is in the first node

```
if self.head.record == record:
    self.head = self.head.next_node
    return True
```



Linked List

- If the record is not there, we just run out of nodes
  - We need to be careful: we cannot check for a record in a node that is not there

```
current_node = self.head
        while current_node.next_node and
current_node.next_node.record != record:
            current_node = current_node.next_node
```

- This is an important trick:
  - We first check that current_node.next_node exists
  - And then current_node.next_node.record

Linked List

- We run out of nodes when `next_node` is `None`
- But otherwise, we have identified the deletion point and now can delete

```
if not current_node.next_node:
     return False
else:
     current_node.next_node =
                    current_node.next_node.next_node
```

Linked List

```
def remove(self, record):
        if not self.head:
            return False
        if self.head.record == record:
            self.head = self.head.next_node
            return True
    current_node = self.head
    while current_node.next_node and current_node.next_node.record !=
                        record:
        current_node = current_node.next_node
    if not current_node.next_node:
        return False
    else:
        current_node.next_node = current_node.next_node.next_node
```

Linked List as Stack

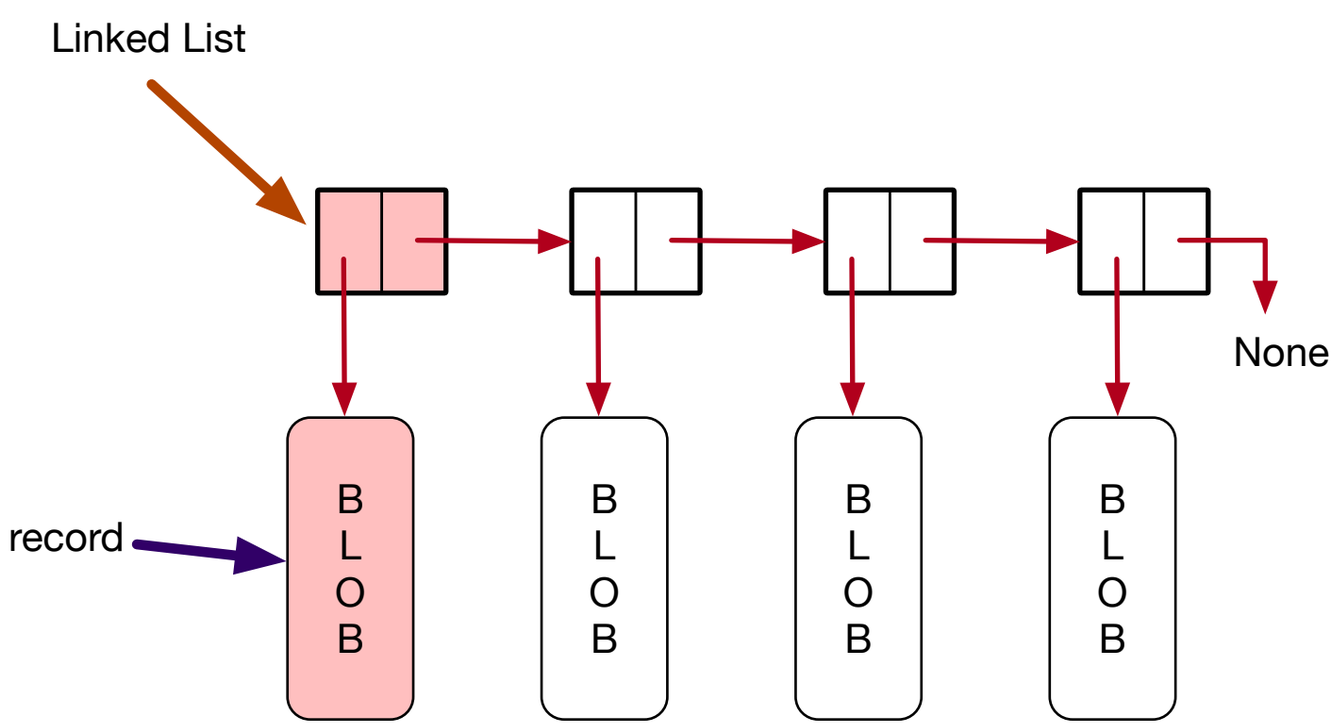
- We insert at the head of the linked list
- Why don't we pop at the head of the linked list?
- This implements a stack as a linked list

Linked List as Stack

- Idea:
  - Remove leading node

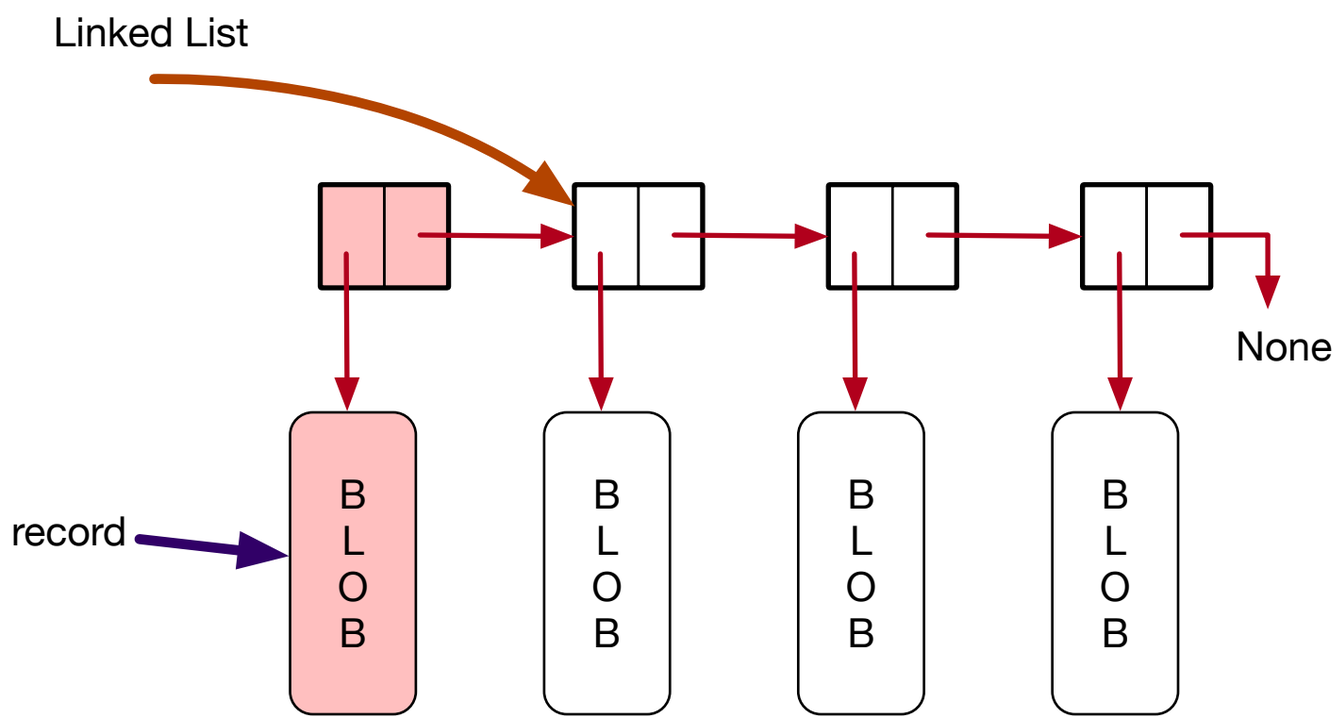
Linked List as Stack

- Step 1:
  - Safe guard the record
    - In `self.head.record`



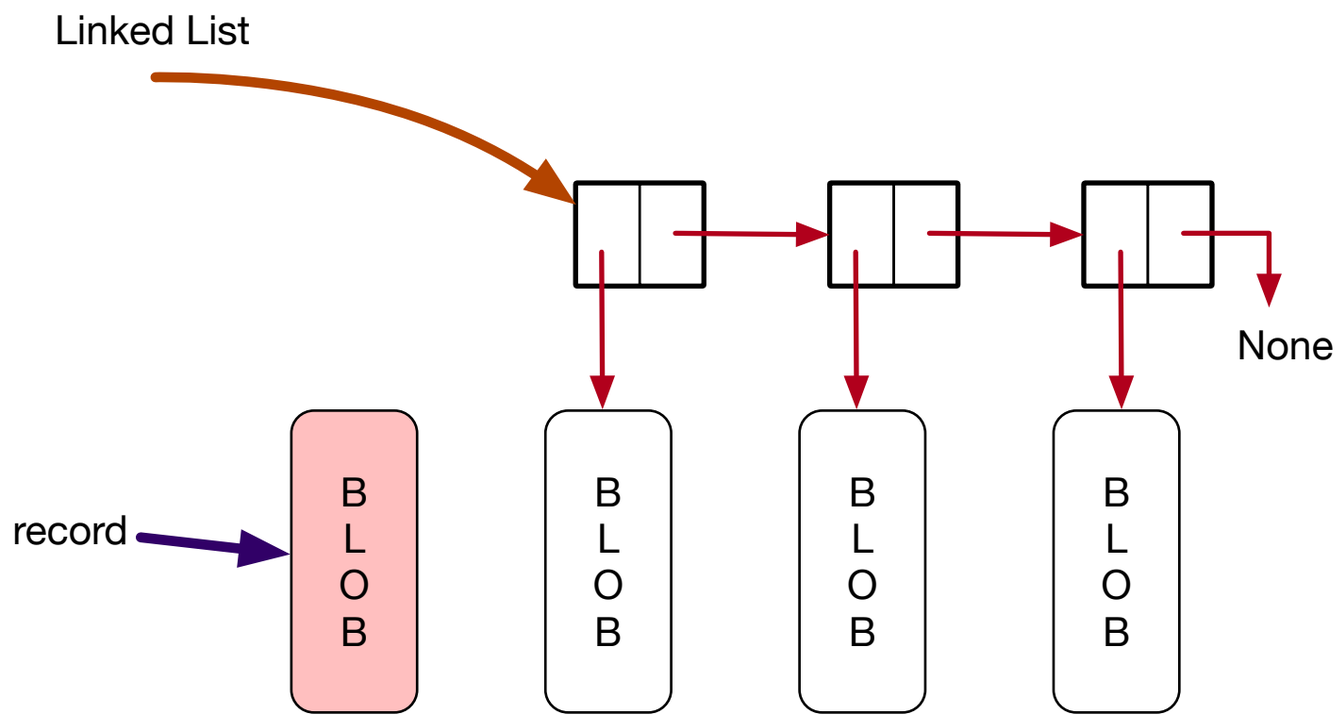
Linked List as Stack

- Step 2:
  - Reset self.head to the next node



Linked List as Stack

- Automatic:
  - There are no links to the previous head
    - The Garbage collection can remove the node



Linked List as Stack

- Python code:
  - Add error handling if the linked list is empty

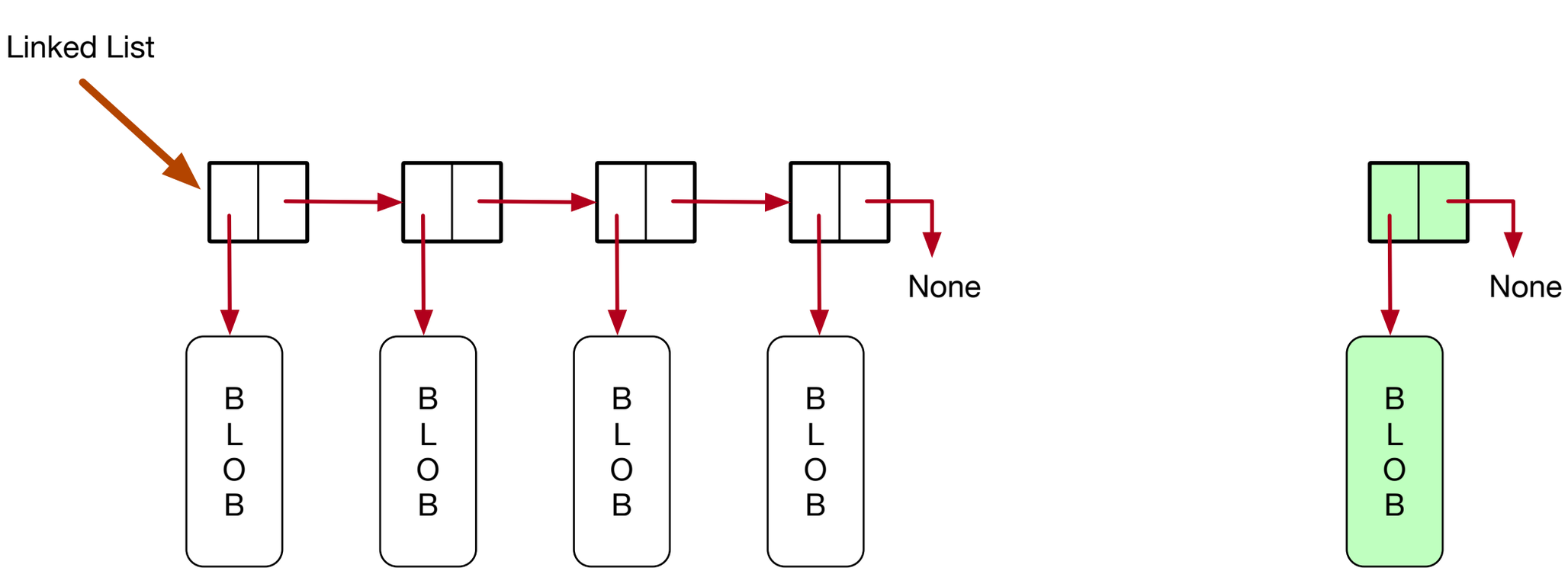
```
def pop(self):
        if not self.head:
            raise ValueError('Popping from empty stack')
        record = self.head.record
        self.head = self.head.next_node
        return record
```

Linked List as Queue

- Now we need to either pop at the tail or push at the tail
  - Push at the tail
    - Idea: Have a `current` node pointer walk through the list
    - Check if the `current.next_node` is `None`
    - If it is: This is the last node and we insert the new node there

Linked List as Queue

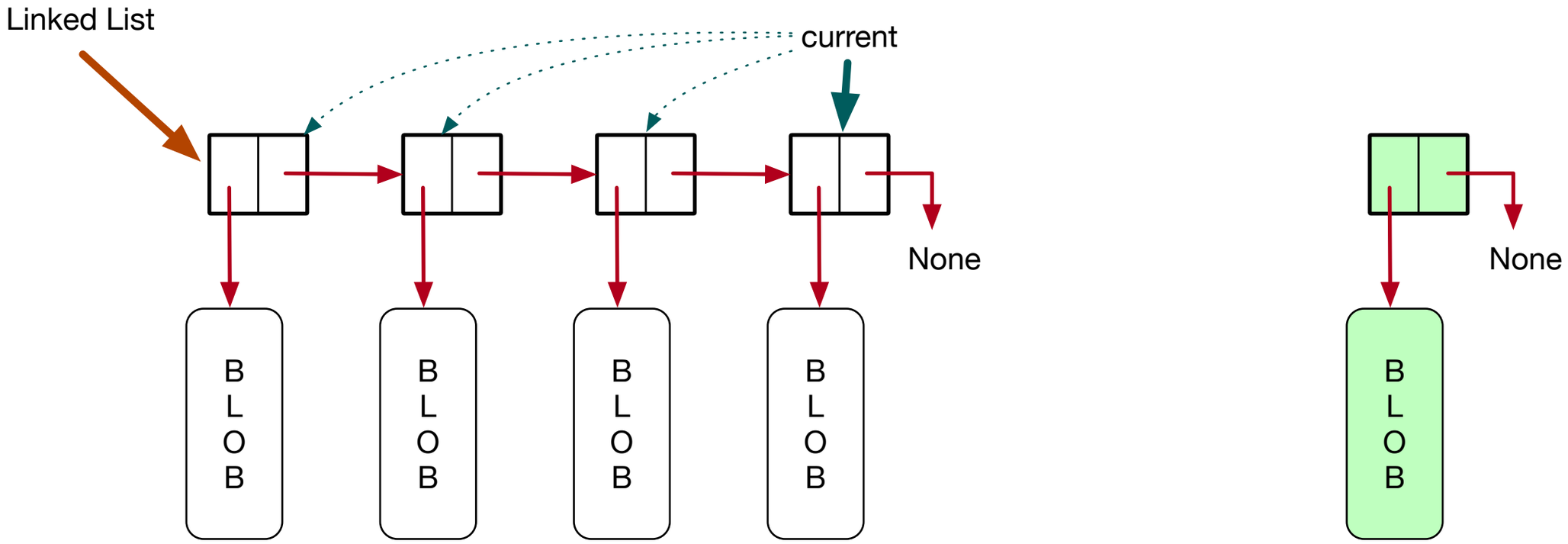
- Create a new node



Linked List as Queue

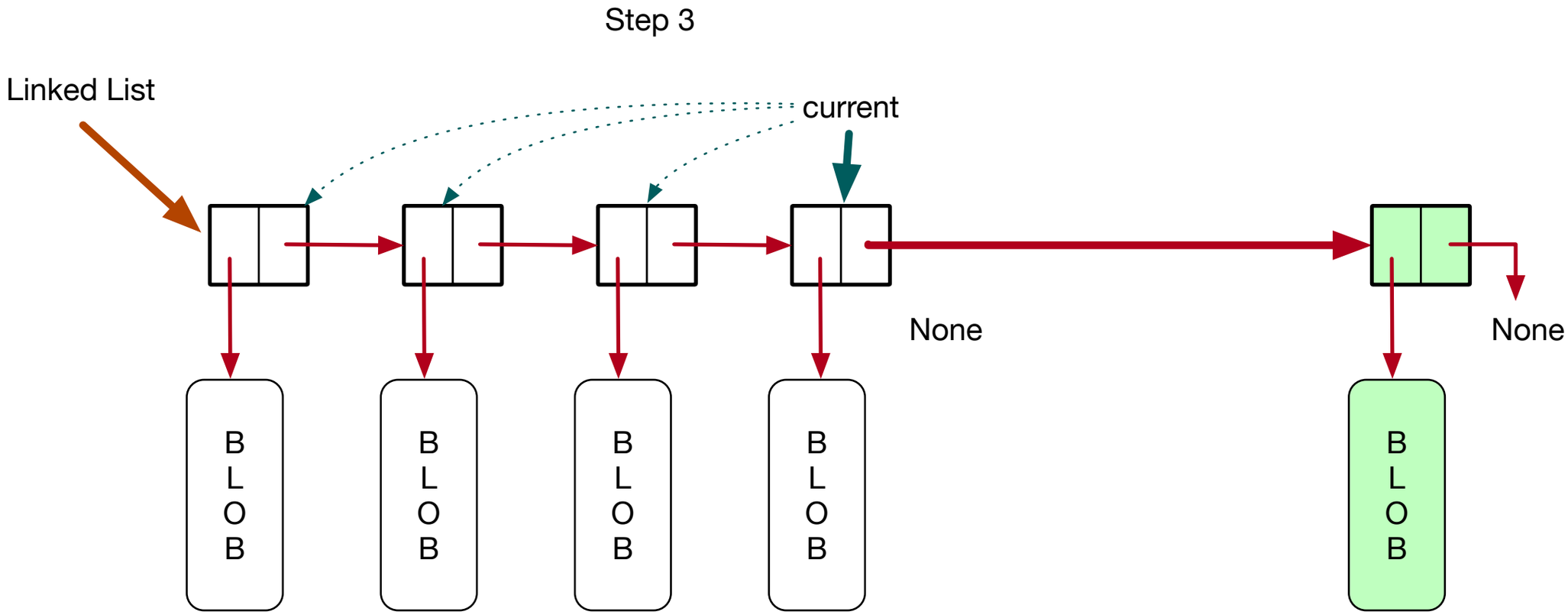
- Adjust current to find the last node in the linked list

Step 2



Linked List as Queue

- Reset the next_node in the current node



Linked List as Queue

- Create new node

```
def push(self, record):
        new_node = Node(record)
        if not self.head:
            self.head = new_node
            return
        current_node = self.head
        while current_node.next_node:
            current_node = current_node.next_node
        current_node.next_node = new_node
```

Linked List as Queue

- Special case:
  - The linked list is empty

```
def push(self, record):
        new_node = Node(record)
        if not self.head:
            self.head = new_node
            return
        current_node = self.head
        while current_node.next_node:
            current_node = current_node.next_node
        current_node.next_node = new_node
```

Linked List as Queue

- Set `current_node` to the beginning of queue
  - Then move until `current_node.next_node` is None

```
def push(self, record):
    new_node = Node(record)
    if not self.head:
        self.head = new_node
        return
    current_node = self.head
    while current_node.next_node:
        current_node = current_node.next_node
    current_node.next_node = new_node
```

Linked List as Queue

- Now, current_node is pointing to the last node
  - So we can connect it to the new node

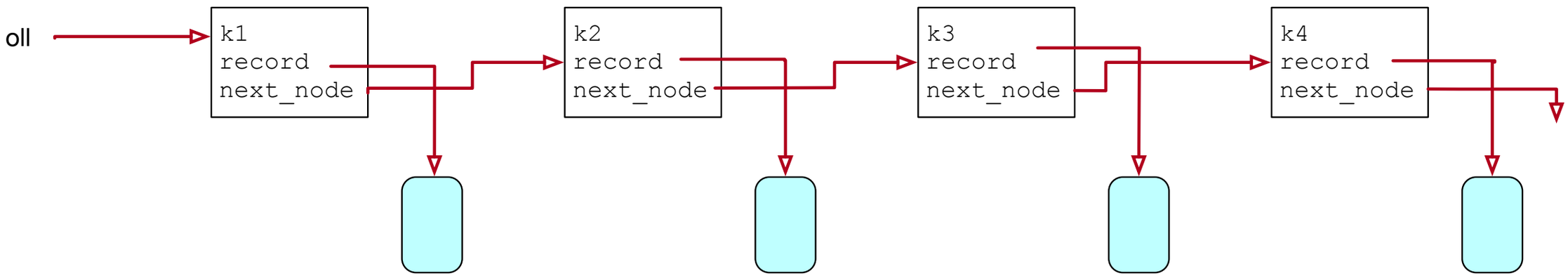
```
def push(self, record):
    new_node = Node(record)
    if not self.head:
        self.head = new_node
        return
    current_node = self.head
    while current_node.next_node:
        current_node = current_node.next_node
    current_node.next_node = new_node
```

Ordered Linked List

- A different type of records:
  - Key and rest of record
  - Example: customer data is organized by giving the customer a unique identifier
    - We look up customer data by using the unique ID
- Ordered Linked List:
  - Linked list where records are ordered by a key

Ordered Linked List

- Have a node structure that has both key and record



  - `k1 < k2 < k3 < k4`
- Advantage: Avoid records that duplicate the key

Ordered Linked List

- Modify the current node structure to have a key

```
class Node:
    def __init__(self, key,  my_record):
        self.key = key
        self.next_node = None
        self.record = my_record
def __repr__(self):
        string = "Class Node "
        string += str(id(self))
        if self.next_node:
            string += ", next is " + str(id(self.next_node))
        else:
            string += ", end node "
        string += ", record is " + str(self.record)
        string += ", key is " + str(self.key)
        return string
```

Ordered Linked List

- We implement an Ordered Linked List as a class
  - Initially empty

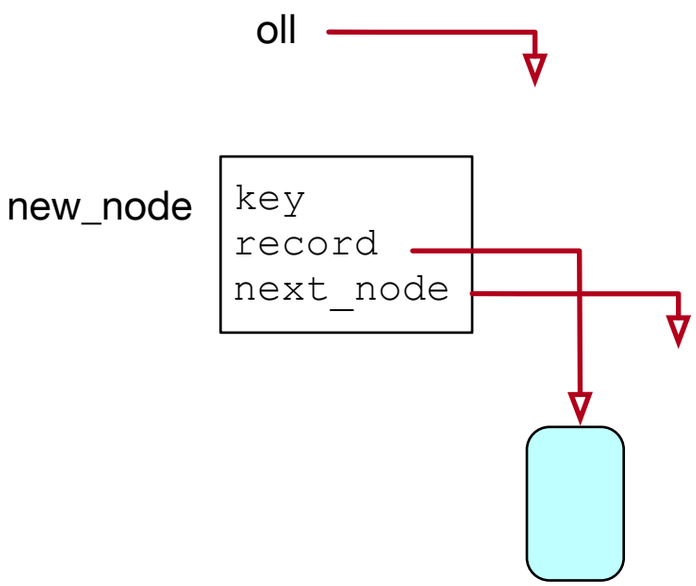
```
class OLL:
    def __init__(self):
        self.head = None
```

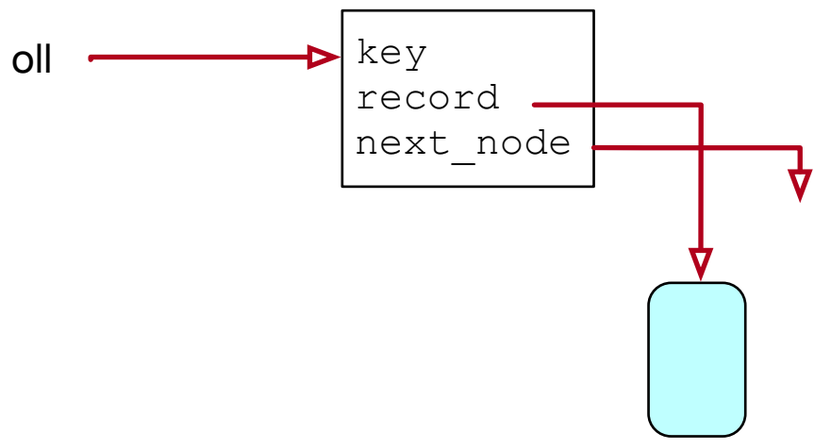
Ordered Linked List

- The linked list assures that the keys increase as we move from the head to the tail
  - This is done by careful structuring of the insert function

Ordered Linked List

- First exceptional case:
  - The list is empty
    - Then just make the new_node the head





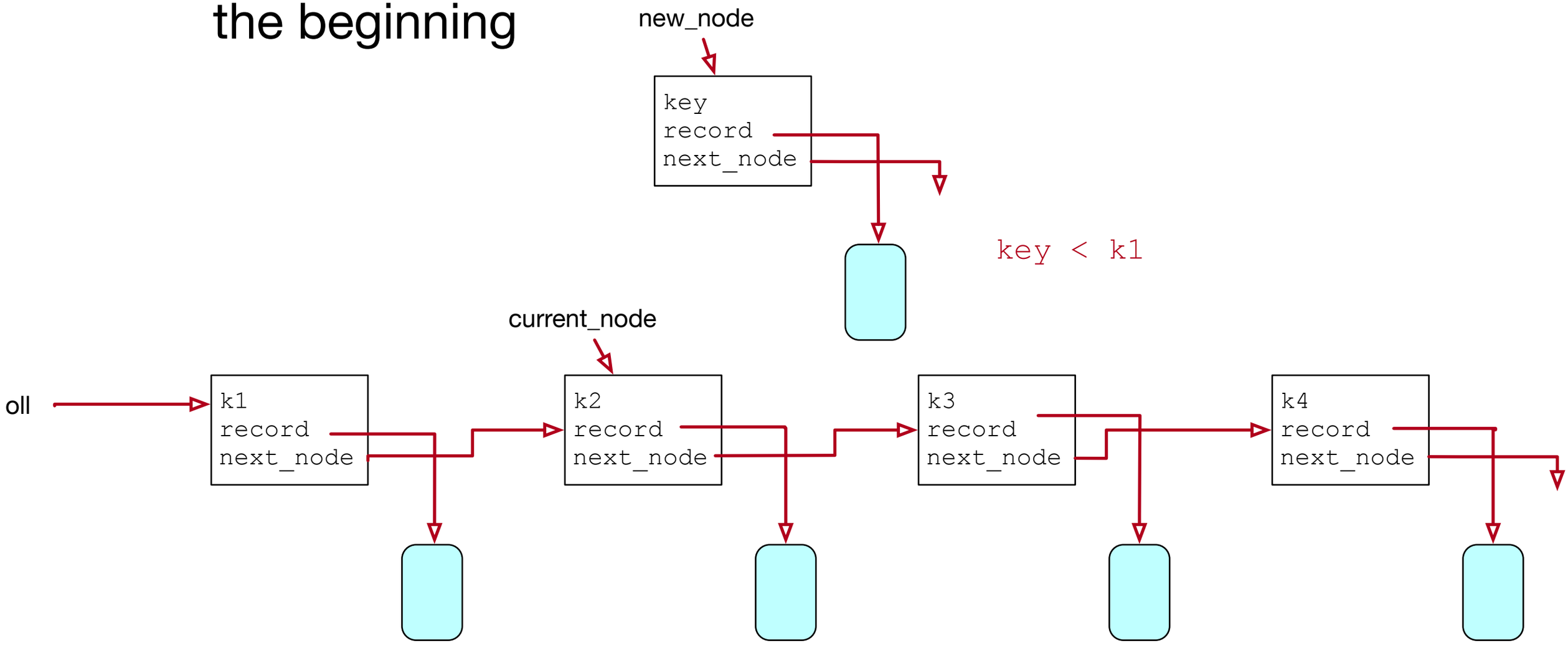
Ordered Linked List

- Python Code
  - The list is empty if self.head is None

```
def insert(self, key, record):
        new_node = Node(key, record)
        if not self.head:
            self.head = new_node
            return
```

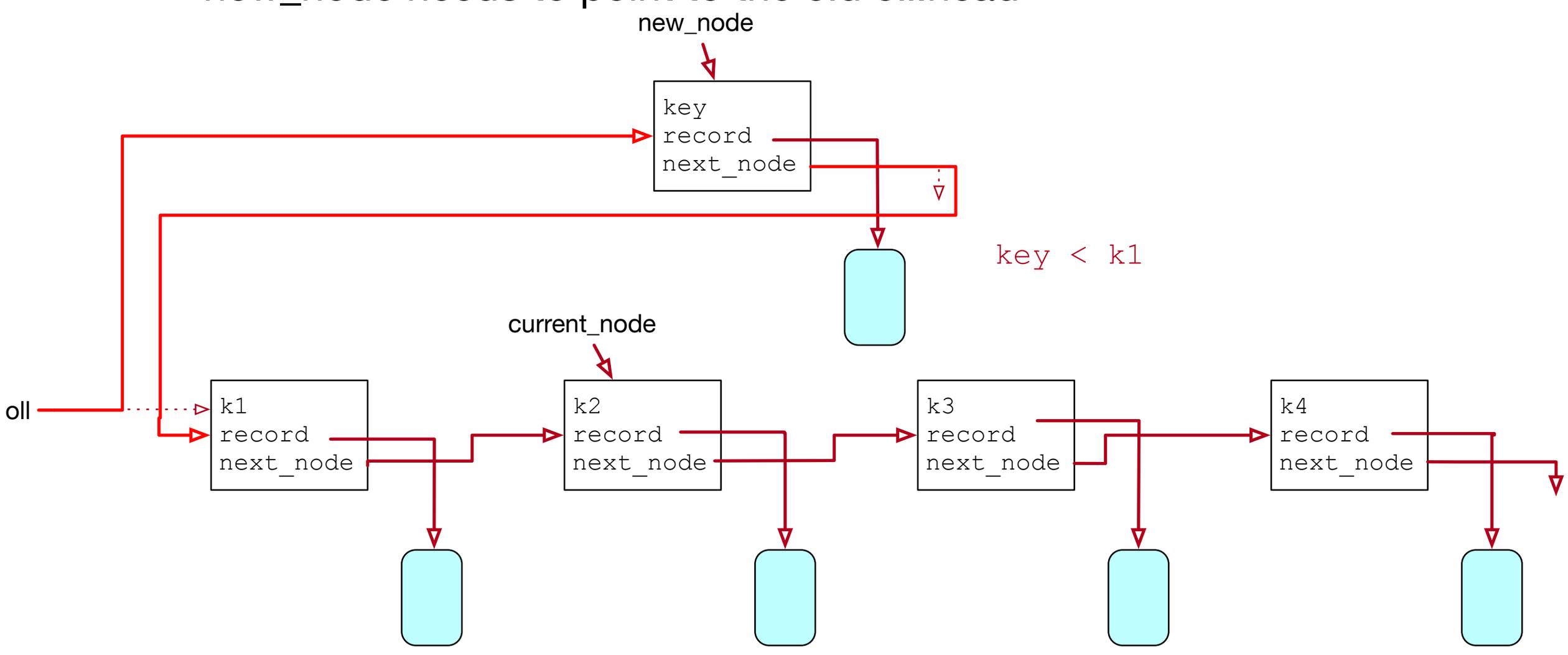
Ordered Linked List

- Second special case:
  - The new key is the smallest and we need to insert at the beginning



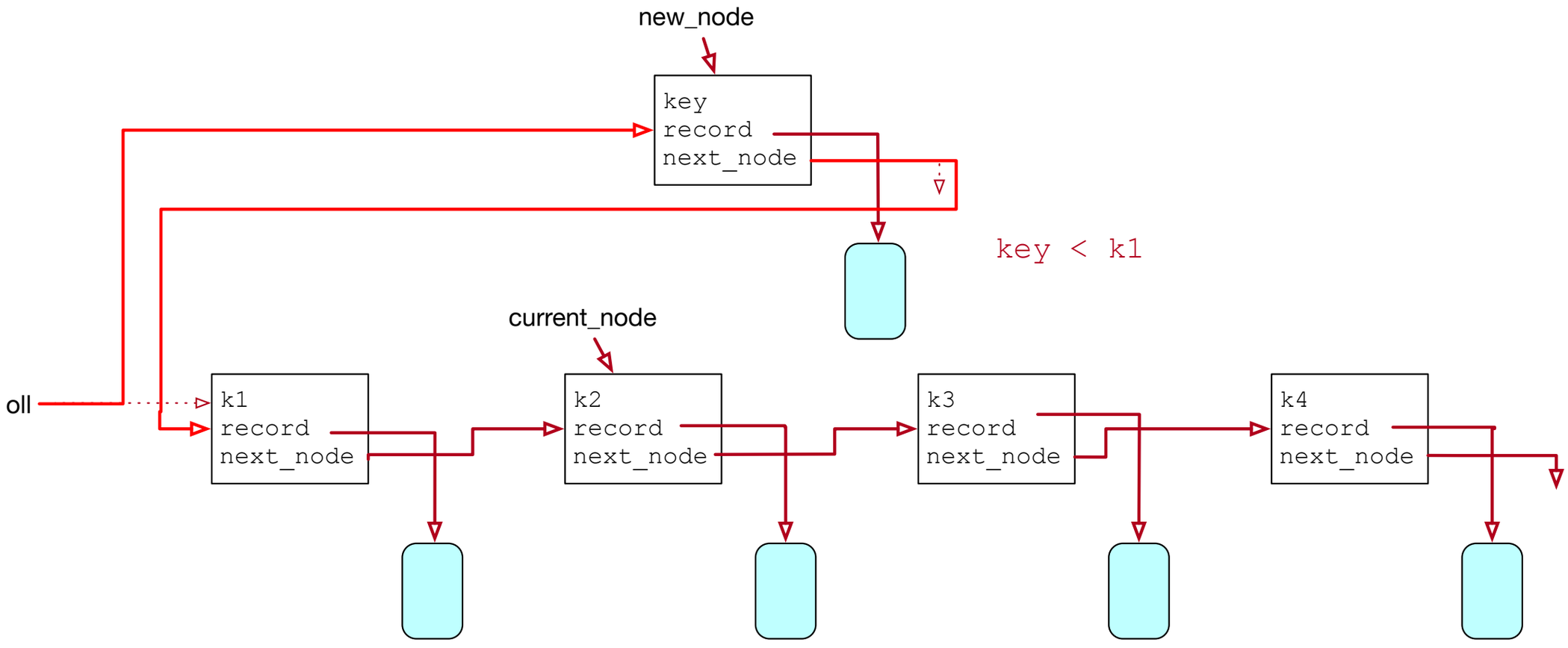
Ordered Linked List

- Need to change two pointers
  - oll needs to point to the new node
  - new_node needs to point to the old oll.head



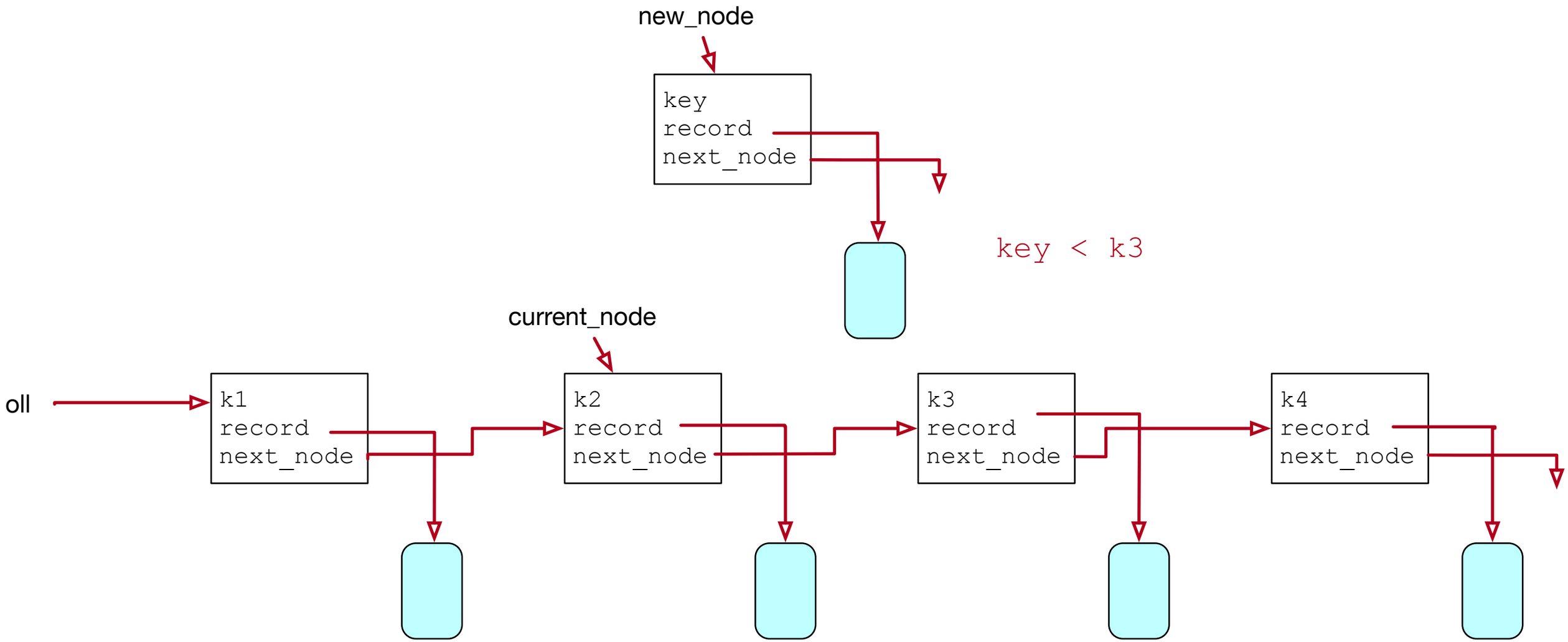
Ordered Linked List

```
elif key < self.head.key:
        new_node.next_node = self.head
        self.head = new_node
        return
```



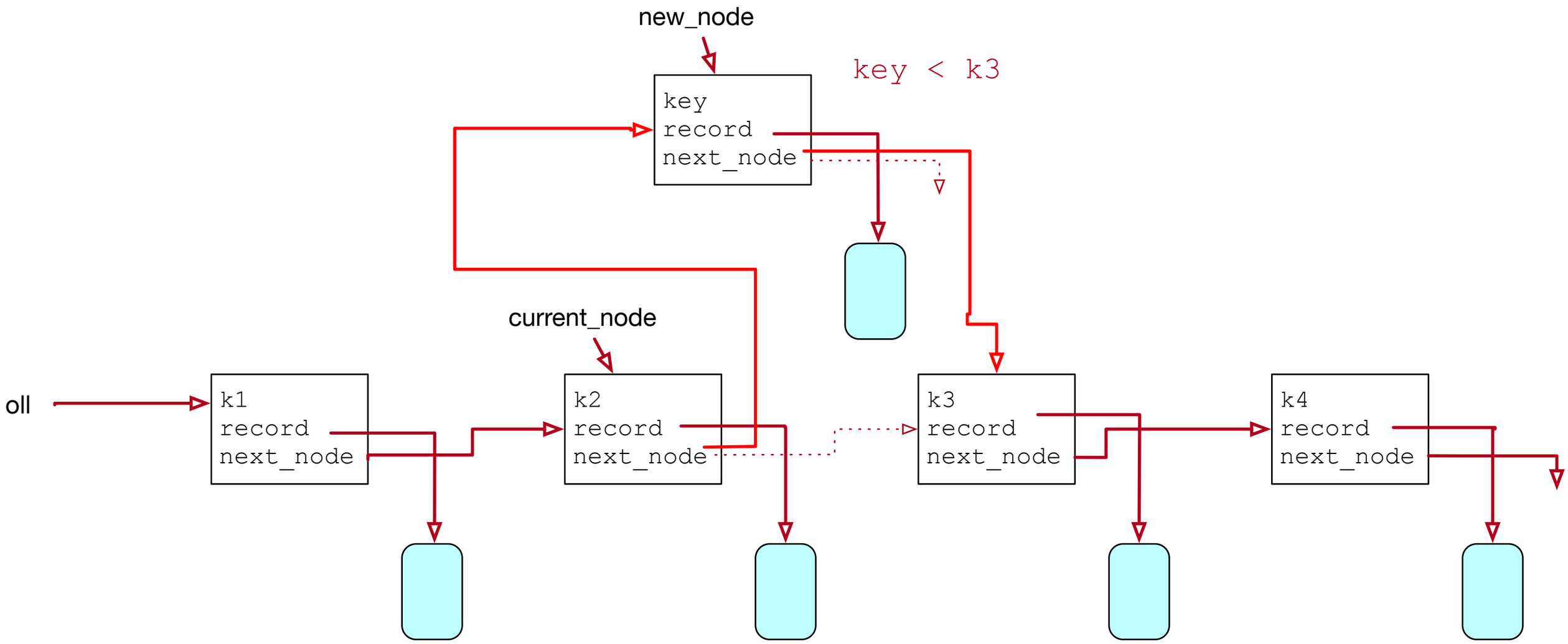
Ordered Linked List

- Normal case: We insert in the middle



Ordered Linked List

- Normal case: We insert in the middle
  - Need to change two pointers

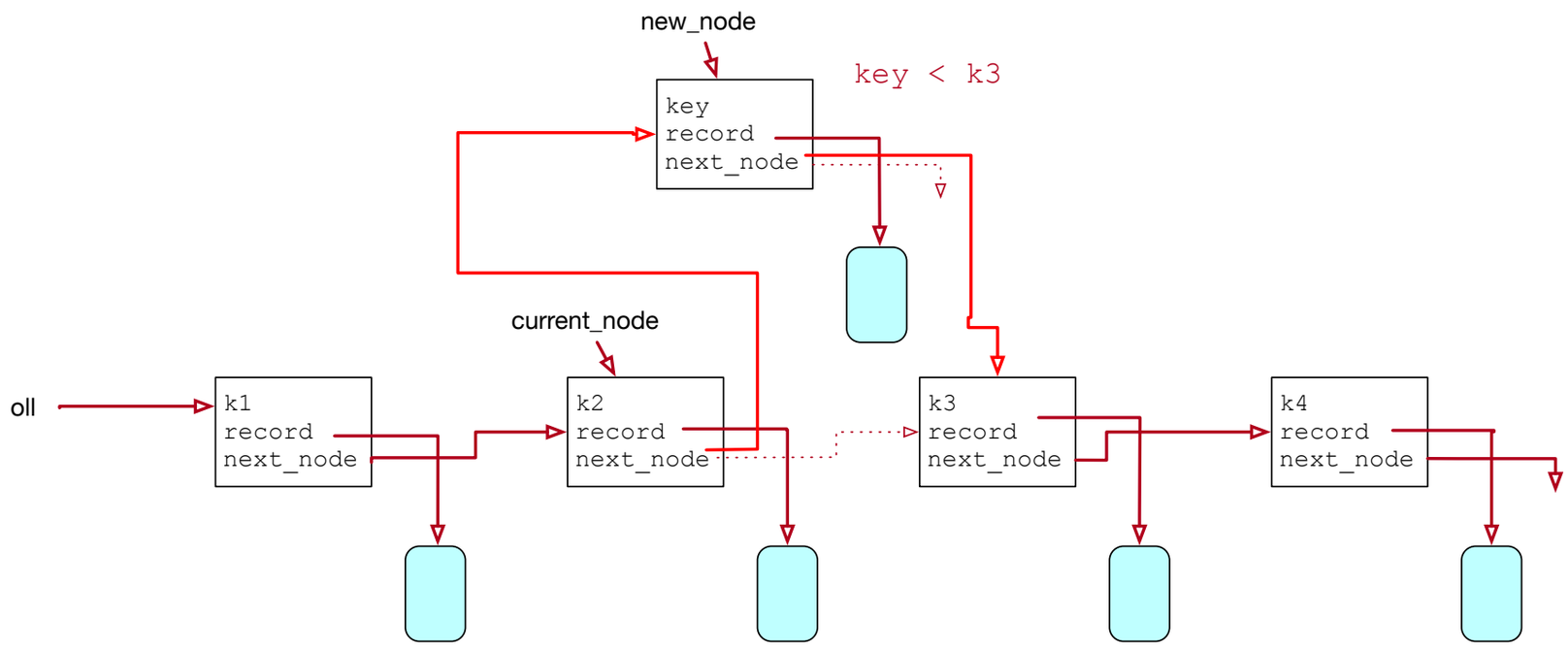


Ordered Linked List

- Trick:
  - Do not loose access, therefore set the pointer in new_node first

Ordered Linked List

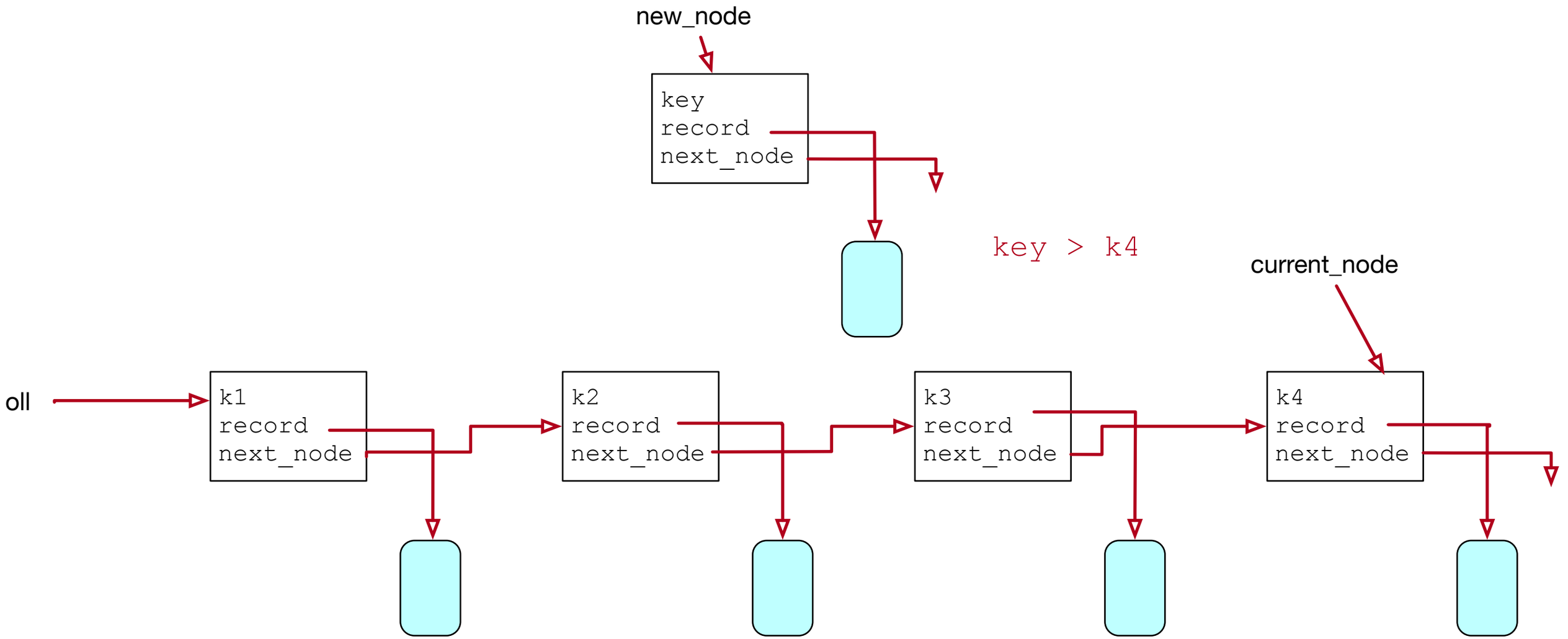
```
else:
      current_node = self.head
      while current_node:
          if current_node.next_node and key < current_node.next_node.key:
              new_node.next_node = current_node.next_node
              current_node.next_node = new_node
              return
current_node = current_node.next_node
```



Ordered Linked List

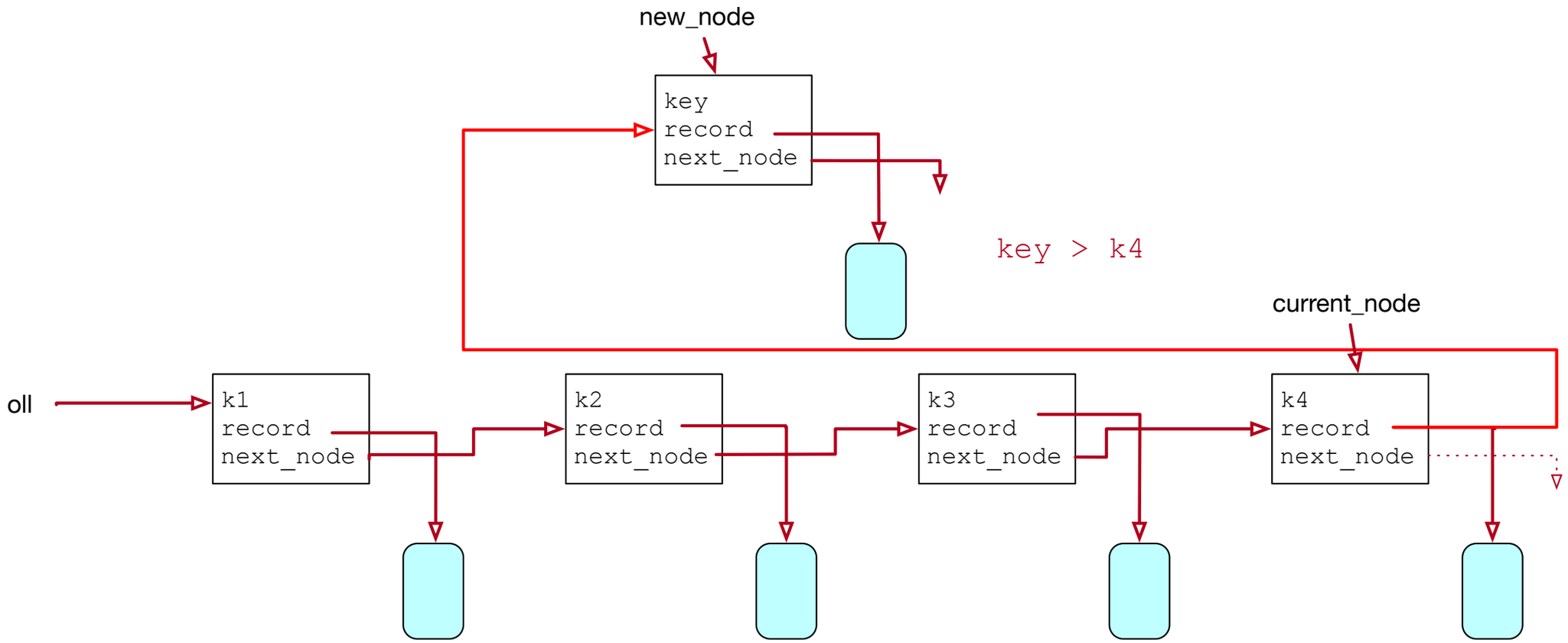
- Final case: The new key is the largest and we need to insert at the end
  - This can be achieved with the previous code
  - But if `current_node` points to the last node
  - Then `current_node.next_node` is `None`

Ordered Linked List



Ordered Linked List

- Resetting new_node.next_node to current_node.next does not cause any harm



Ordered Linked List

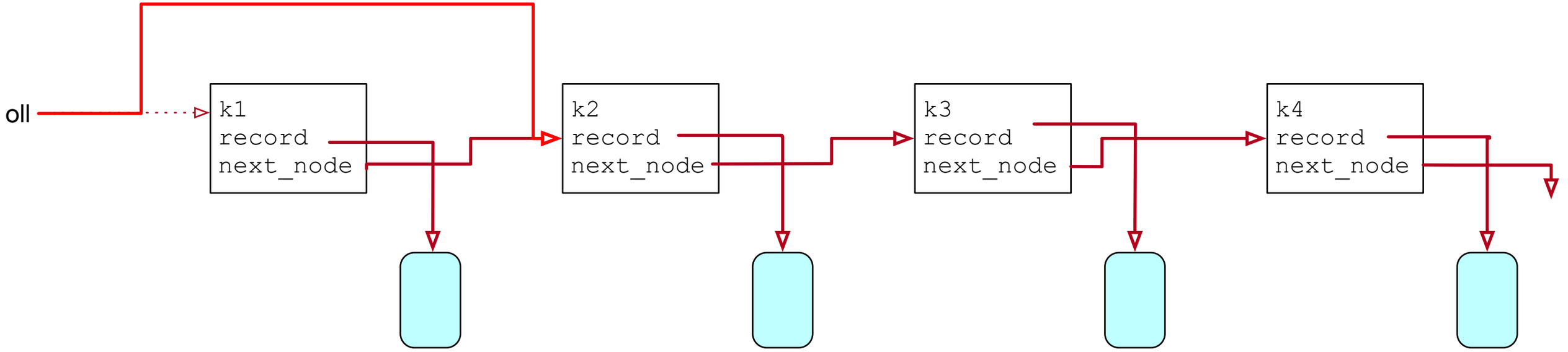
- We need to add one clause to the if statement:
  - `if current_node.next == None`

Ordered Linked List

```
else:
    current_node = self.head
    while current_node:
        if (not current_node.next_node or
        current_node.next_node and key <
        current_node.next_node.key):
        new_node.next_node = current_node.next_node
        current_node.next_node = new_node
        return
current_node = current_node.next_node
```

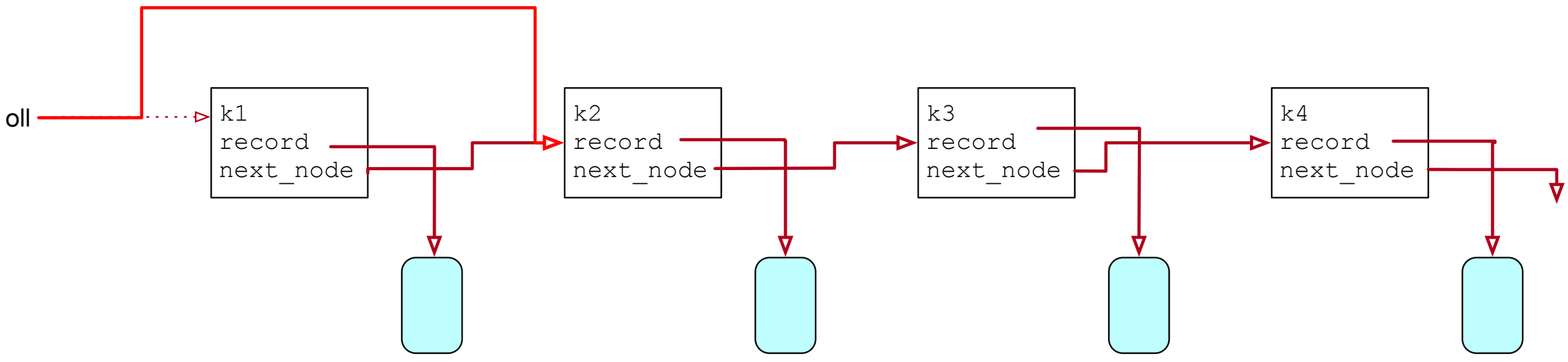
Ordered Linked List

- How do we delete a record by key
  - Case 1: Deleting first record



Ordered Linked List

```
def delete(self, key):
        if not self.head:
            return False
        elif self.head.key == key:
            self.head = self.head.next_node
            return True
```

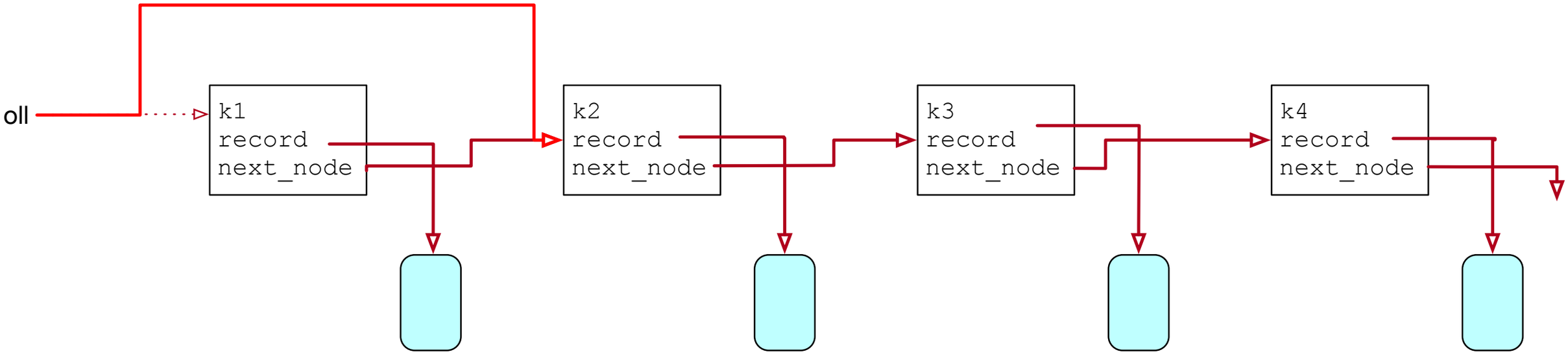


Ordered Linked List

- Deletion by key:
  - Find the node before the node with the key
  - Reset the next-node link in that node
- Special cases:
  - the list is empty: return False
  - the node to be deleted is the first:
    - Reset the list.head to the next node and return True

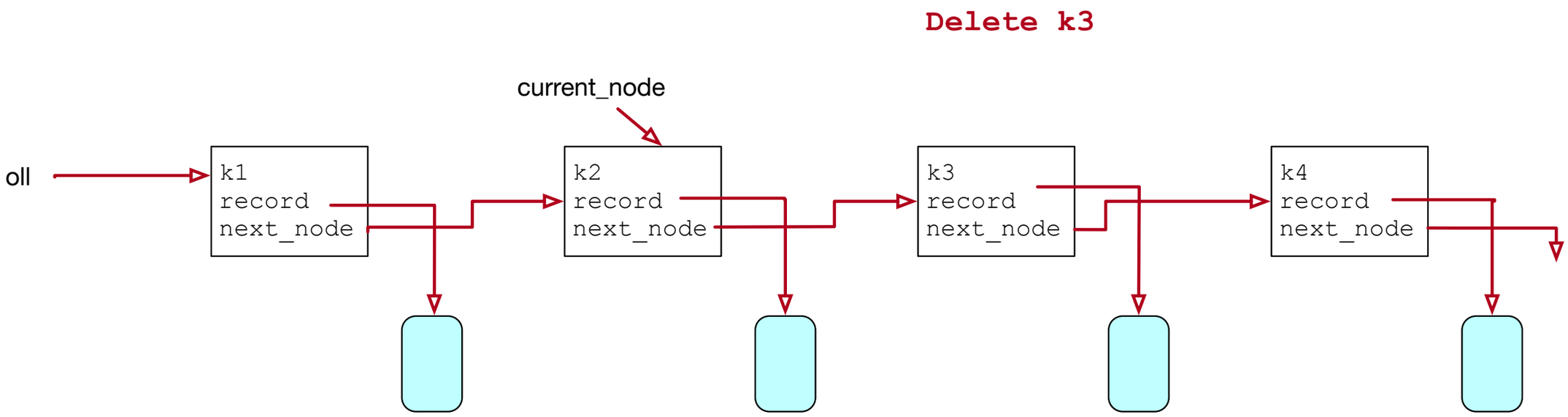
Ordered Linked List

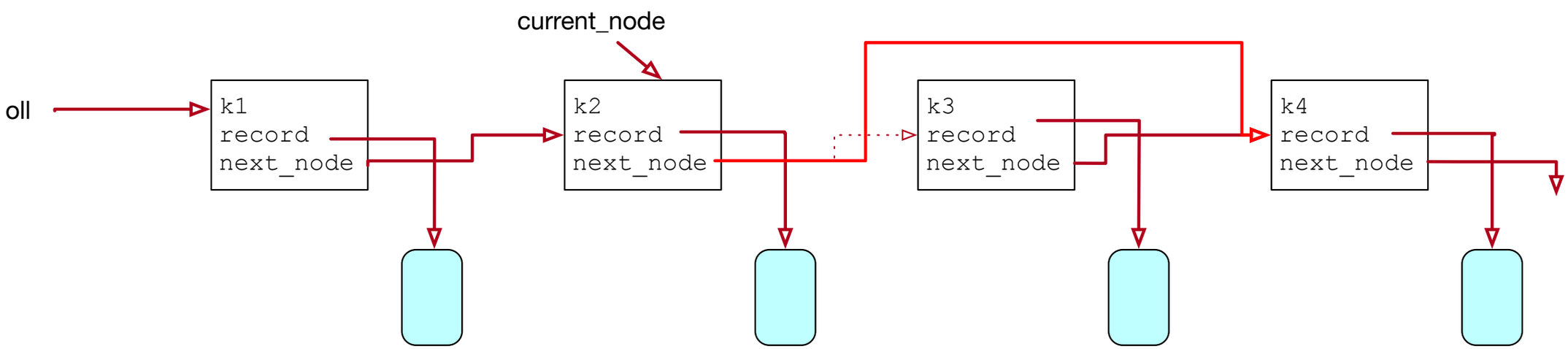
- Delete first node:



Ordered Linked List

- Common case: Delete in the middle or at the tail





Ordered Linked List

- If we fall of the end of the list:
  - key was not present: return False

Ordered Linked List

- Homework

```
def delete(self, key):
        if not self.head:
            return False
        elif self.head.key == key:
            ***
            return True
    else:
        current_node = self.head
        while current_node:
            if current_node.next_node and key == current_node.next_node.key:
                ****
            elif not current_node.next_node:
                return False
            current_node = current_node.next_node
        return False
```