Linked List 2

Thomas Schwarz, SJ

Iterators

- The `current_node` programming paradigm is an ***iterator***
  - An iterator has:
    - A method to access the current object
    - A method to move forward
    - And sometimes a method to move backwards
    - Methods to compare two different iterators

Linked List Performance

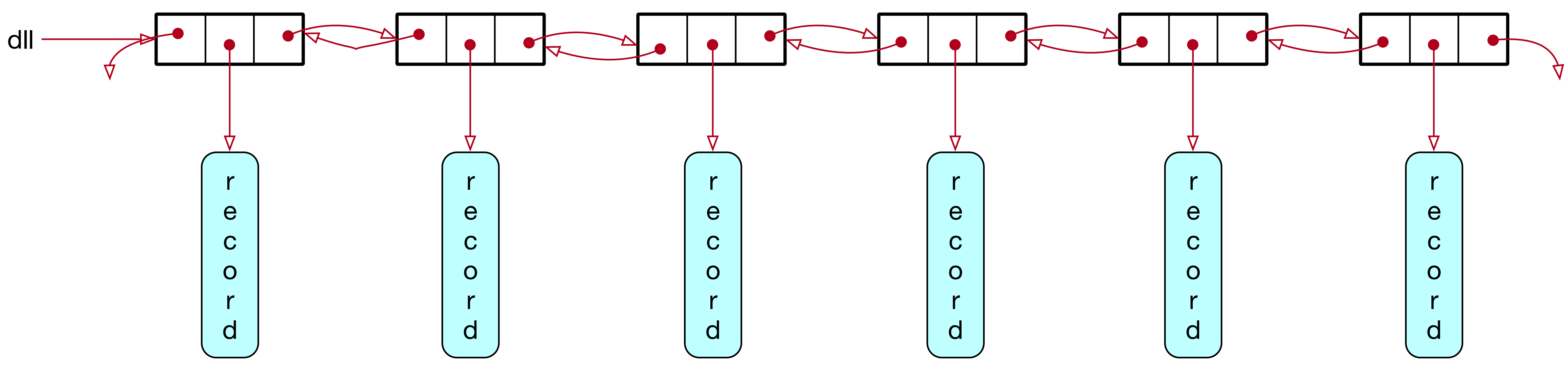
- We can estimate performance of linked -- list operations by looking at the number of nodes accessed
- Assume a list with n nodes
  - Inserting at the head: 1 node
  - Inserting at the tail: n nodes
  - Inserting in the middle: $n/2$ on average
  - Deleting at the head: 1 node
  - Deleting at the tail: n node
  - Deleting in the middle: $n/2$ on average

Linked List Performance

- Implementing a stack:
  - We access one node
- Implementing a queue:
  - Insert at the head, pop at the tail
  - Or: Insert at the tail, pop at the head
    - One is going to use *n* nodes

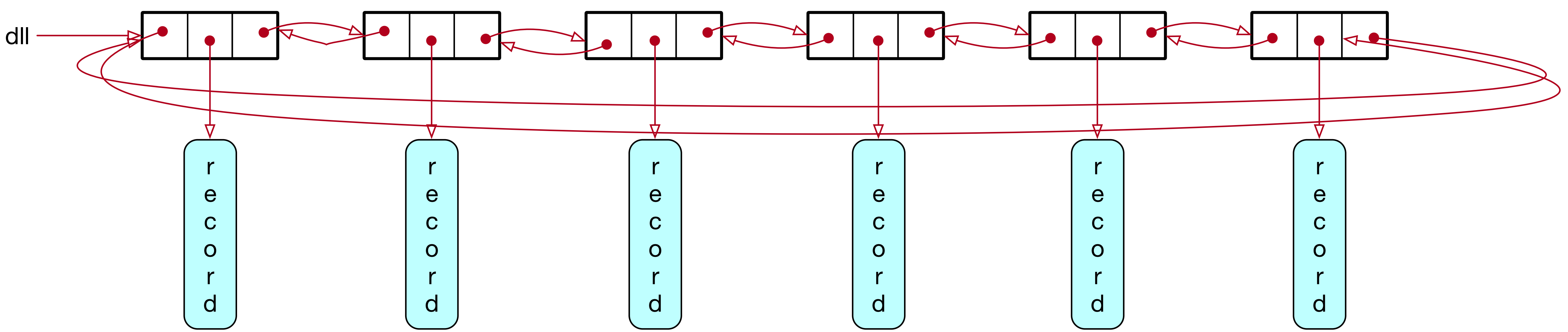
Double Linked List

- To overcome the performance penalty,
  - use a double linked list
    - Each node has now a forward and a backward pointer



Double Linked List

- And then connect head and tail in order to give a **circular *double linked list***



- The backward pointer of head allows easy access to the tail
- But:
    - For an insert or a delete, we now need to set four pointers instead of two

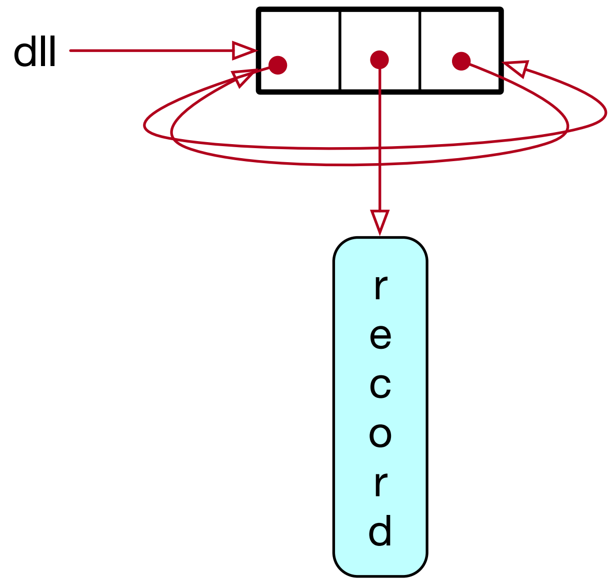
Double Linked List

- Nodes now have a forward and a backward pointer

```
class Node:
    def __init__(self, my_record):
        self.forward = None
        self.back = None
        self.record = my_record
    def __repr__(self):
        string = "Class Node "
        string += str(id(self))
        string += ", forward is " + str(id(self.forward))
        string += ", backward is " + str(id(self.forward))
        string += ", record is " + str(self.record)
        return string
```

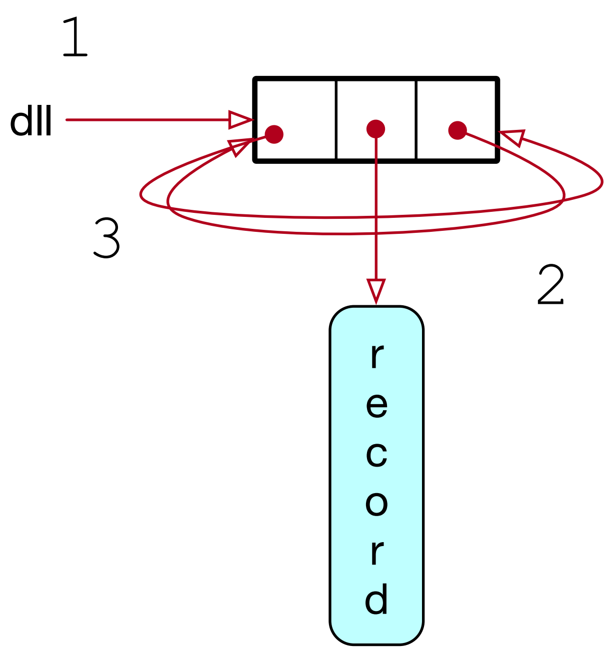
Double Linked List

- Creating a double linked list
  - Initially the list is empty
  - Create a new node
  - Then adjust the head and the two node pointers



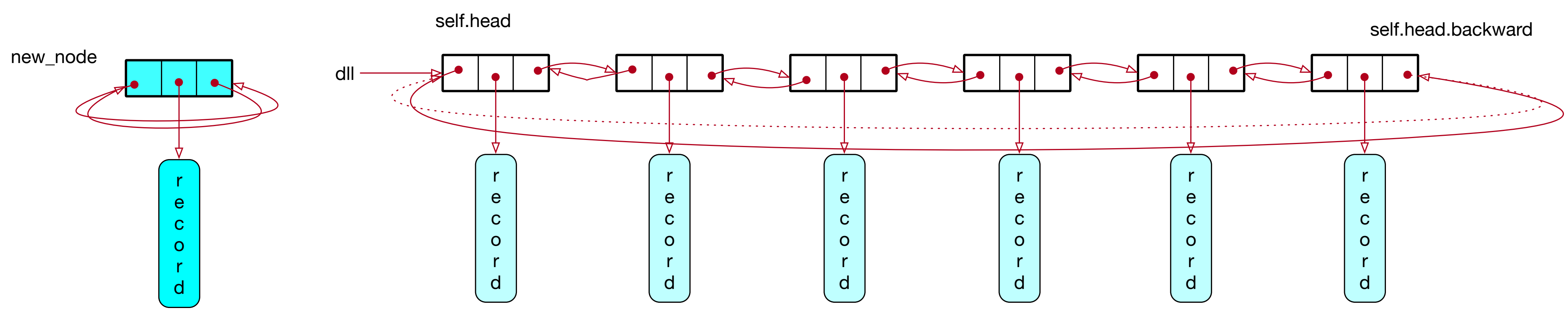
Double Linked List

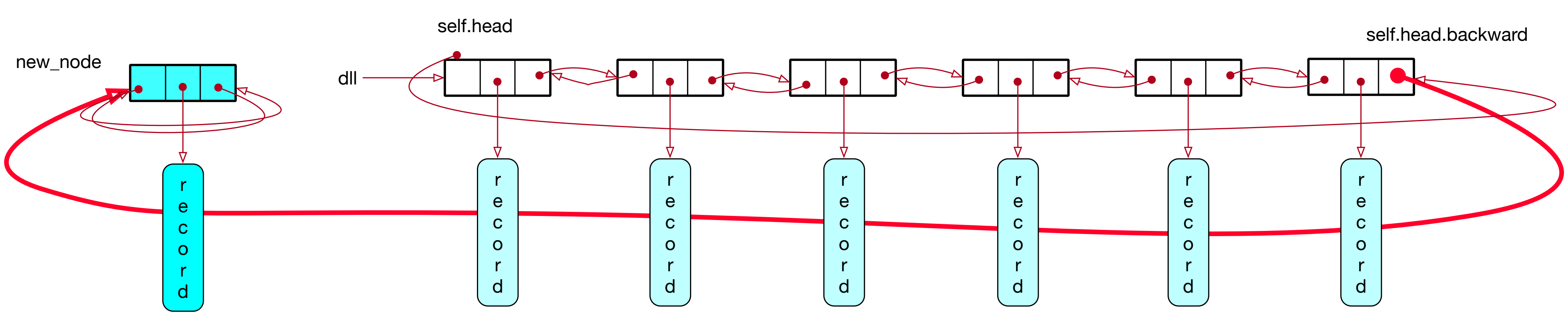
```
if not self.head:
            self.head = new_node
            new_node.forward = new_node
            new_node.back = new_node
```



Double Linked List

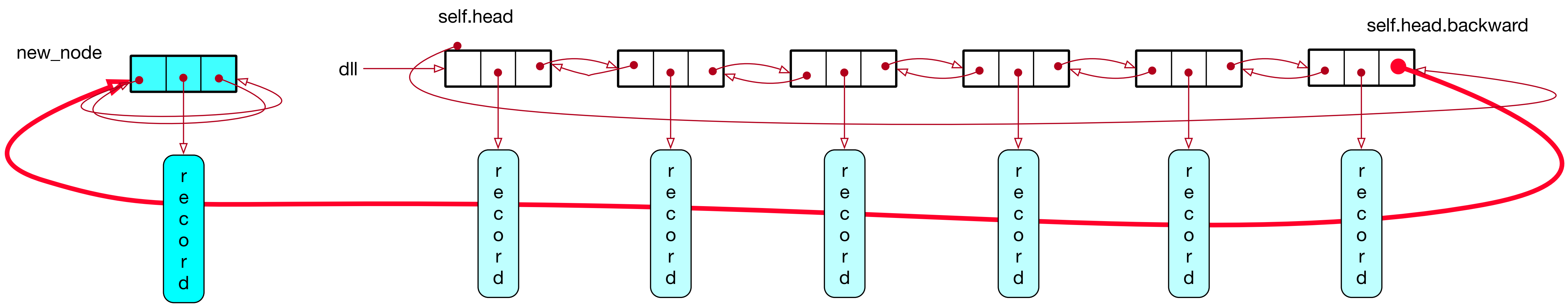
- Inserting at the head:
  - Create new node
  - Then change the forward pointer of the tail to the new node

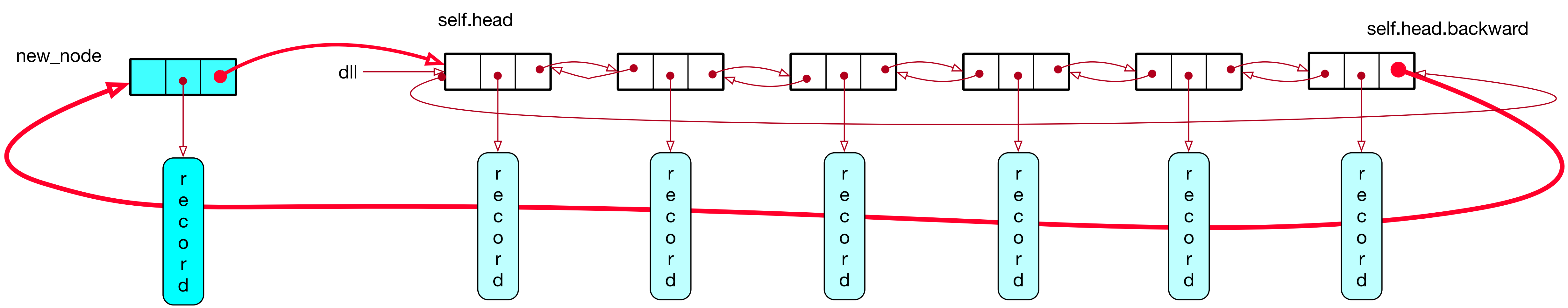




Double Linked List

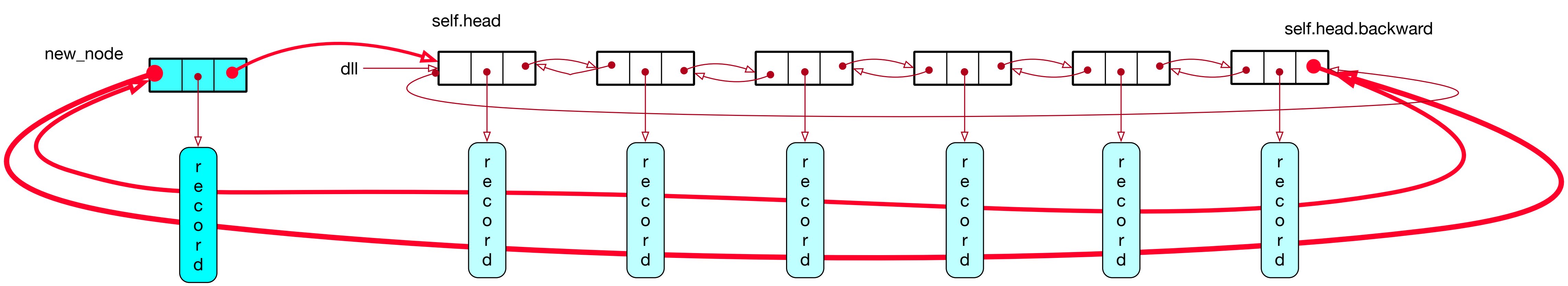
- Set new_node forward to self.head





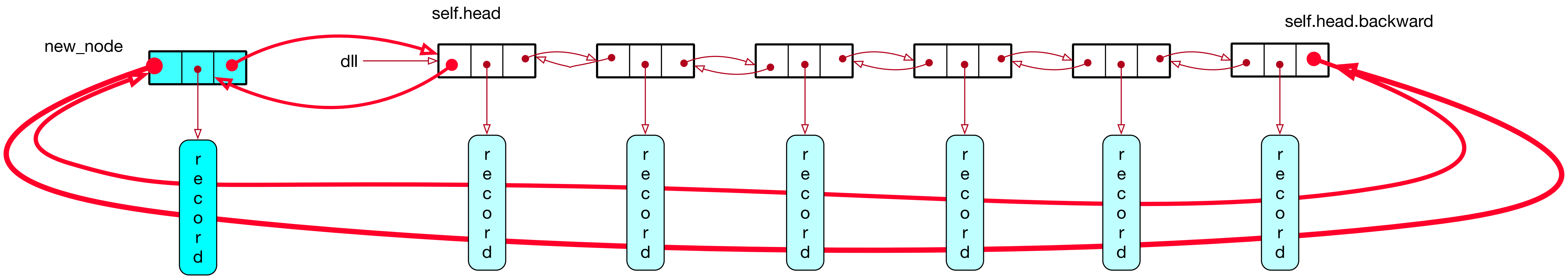
Double Linked List

- Set new_node.backward to the tail



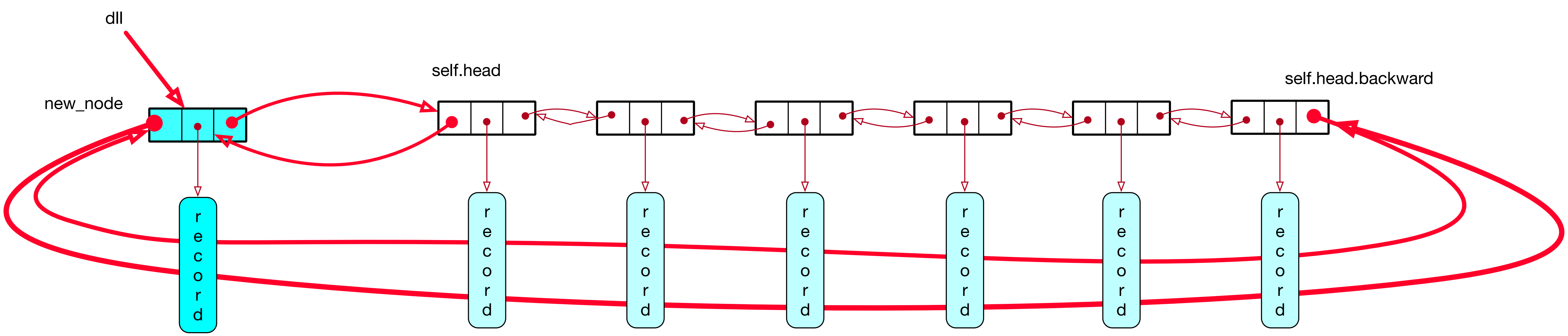
Double Linked List

- Set self.head.backward to new_node



Double Linked List

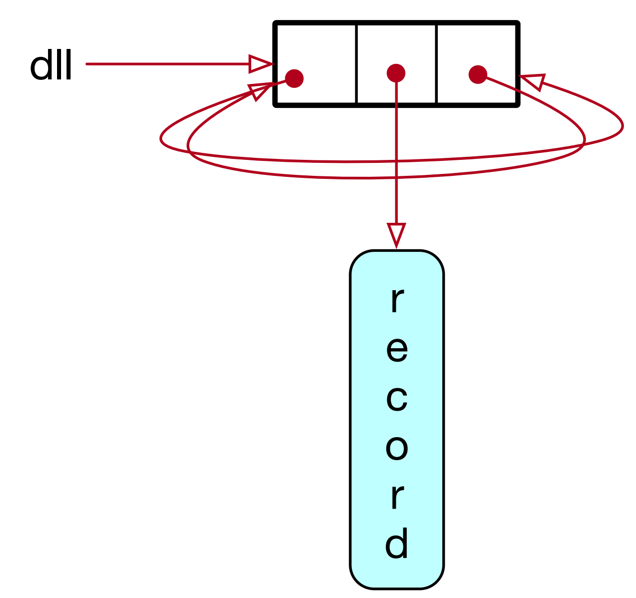
- Finally, set dll.head to the new_node



Double Linked List

- Code:
    - Order is important!
        - Easy part: inserting into an empty list

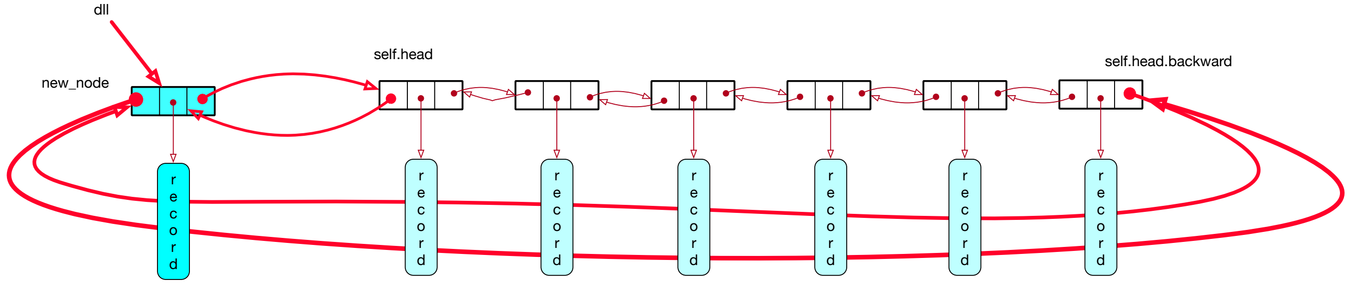
```
def insert_head(self, record):
        new_node = Node(record)
        if not self.head:
            self.head = new_node
            new_node.forward = new_node
            new_node.back = new_node
```



Double Linked List

- Order of operations is important!
  - Otherwise, we loose access or need a temporary variable

```
def insert_head(self, record):
        new_node = Node(record)
        if not self.head:
            ...
        else:
            self.head.back.forward = new_node
            new_node.back = self.head.back
            self.head.back = new_node
            new_node.forward = self.head
            self.head = new_node
```



Double Linked List

- Printing all records from left to right
  - Use the `current_node` paradigm:

```
def print_it_forward(self):
        current = self.head
        while current:
            print(current.record)
            current = current.forward
            if current == self.head:
                return
```

Double Linked List

- Printing all nodes from right to left
    - ```
      def print_it_backward(self):
              current = self.head.back
              while current:
                  print(current.record)
                  current = current.back
                  if current == self.head.back:
                      return
      ```
      ```

Double Linked List

- We can also build an explicit iterator class
  - Iterators
    - provide access to the record
    - allow us to move to the next record
    - allow us to move to the previous record
    - can compare two iterators

Double Linked List

- Implementation

```
class DLL_Iterator:
    def __init__(self, dll):
        self.current_node = dll.head
        self.dll = dll
    def forward(self):
        self.current_node = self.current_node.forward
    def backward(self):
        self.current_node = self.current_node.back
    def get_record(self):
        return self.current_node.record
    def at_tail(self):
        return self.current_node == self.dll.head.back
    def at_head(self):
        return self.current_node == self.dll.head
    def __eq__(self, other):
        return self.dll == other.dll and self.current_node == 
other.current_node
```

Double Linked List

- Example based on iterator

```
it = DLL_Iterator(dll)
while True:
    print(it.get_record())
    it.forward()
    if it.at_head():
        break
```

Double Linked List

- Homework:
    - Add to the iterator class to set an iterator to the tail

Double Linked List

- Maintaining an ordered double linked list
  - Add a field key to the Node class

```
class Node:
   def __init__(self, my_record, key):
       self.forward = None
       self.back = None
       self.record = my_record
       self.key = key
```

Double Linked List

- Only difference:
  - Now need to insert in the middle of a list
  - One special case:
    - Inserting in an empty list
  - Normal case
    - Inserting between two nodes
      - which can be identical

Double Linked List

- Special case:
    - If the list is empty, self.head is None

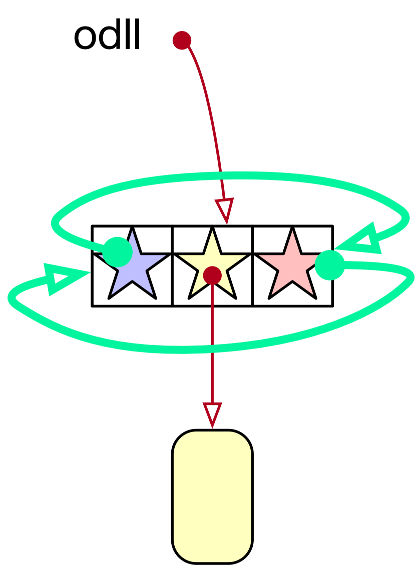
```
class OLL:
    """implements an ordered list of double linked nodes """
    def __init__(self):
        self.head = None
```

Double Linked List

- Special case:
  - If the list is empty, self.head is None
  - Do not forget to set the forward and back pointers

```
class OLL:
    """implements an ordered list of double linked nodes """
    def insert(self, my_record, my_key):
        new_node = Node(my_record, my_key)
        if self.head:
            ......
        else:
            self.head = new_node
            new_node.forward = new_node
            new_node.back = new_node
```

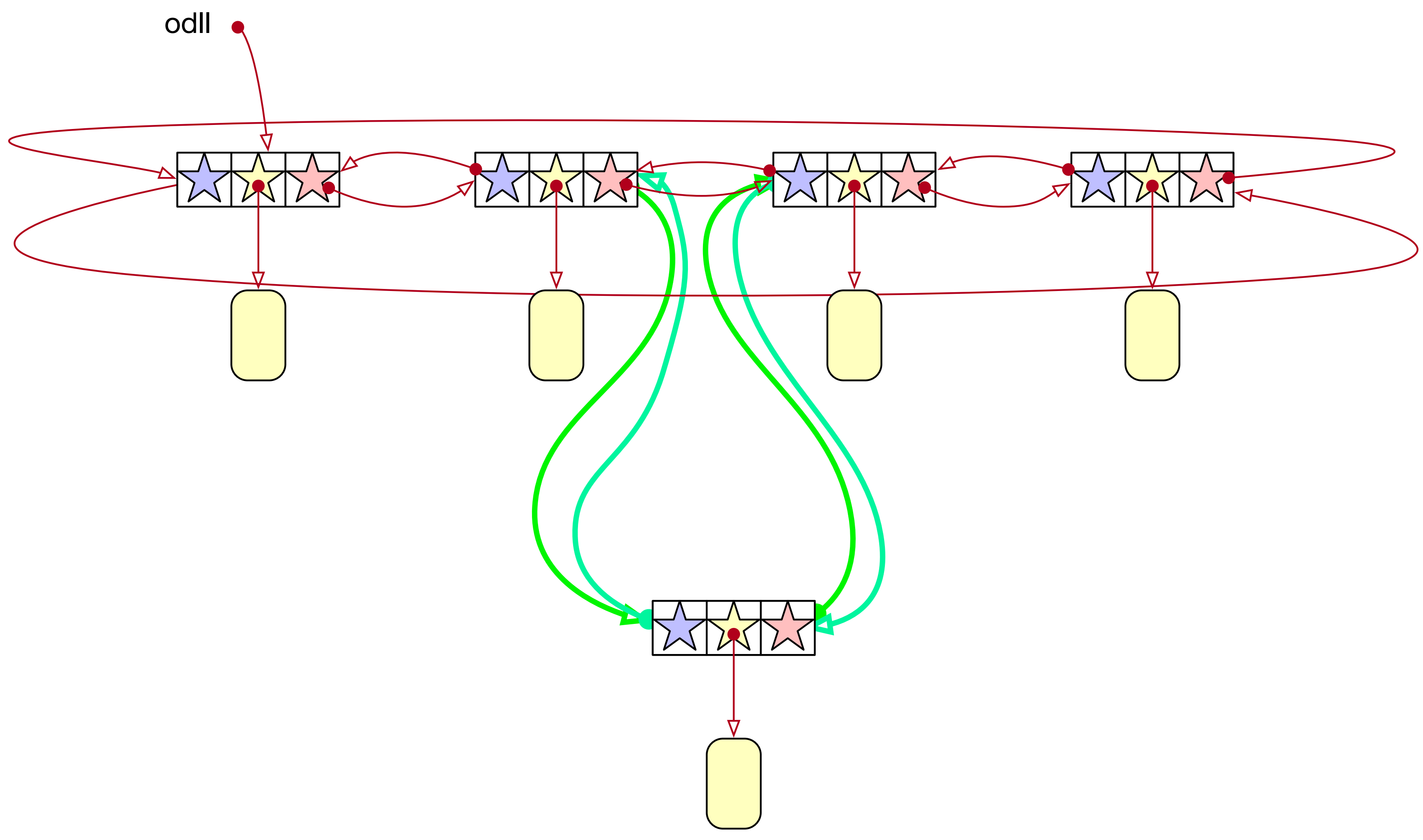
Double Linked List



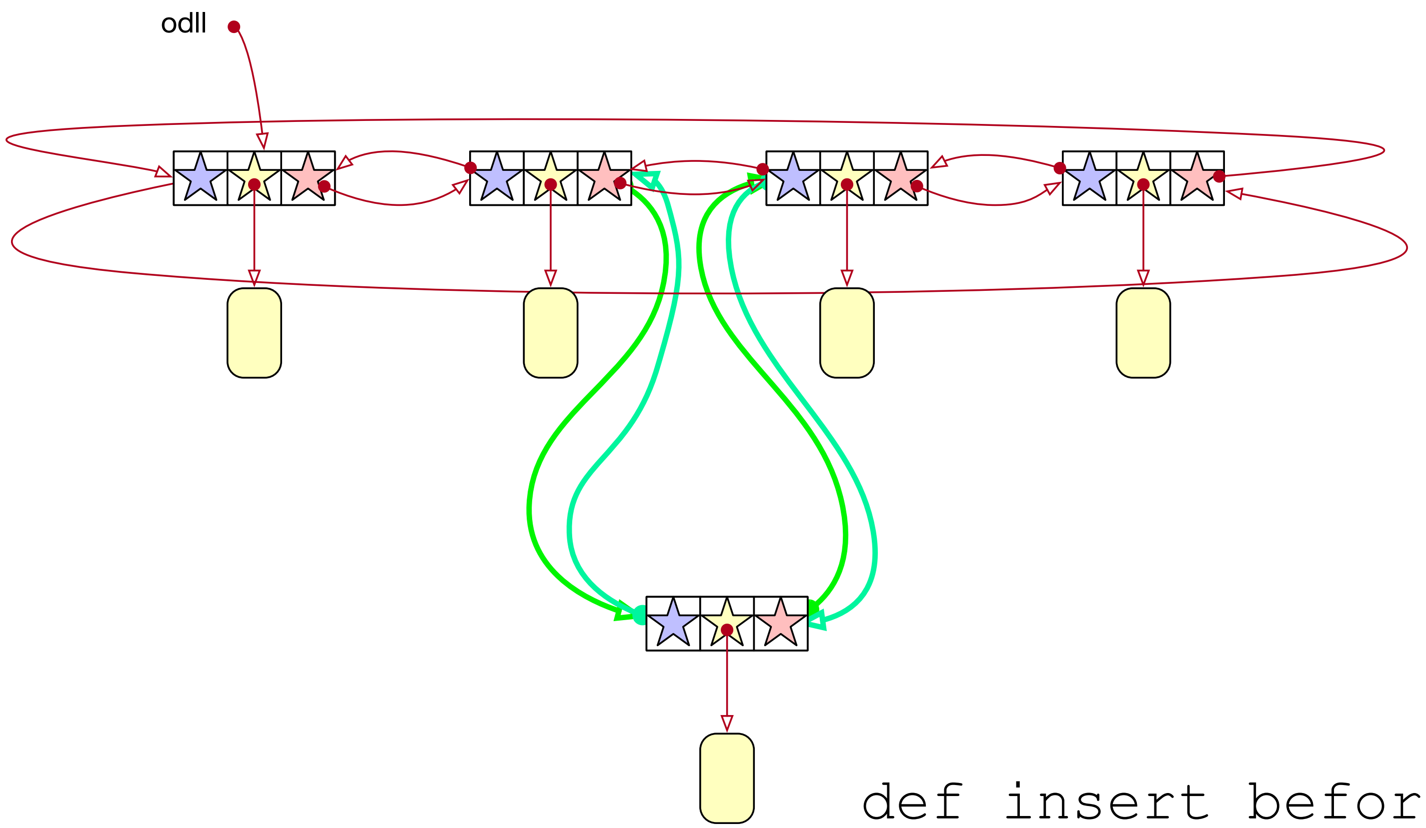

```
else:
    self.head = new_node
    new_node.forward = new_node
    new_node.back = new_node
```

Double Linked List

- Inserting between two nodes
  - Implement as a class (not instance) method
    - Reset four pointers



Double Linked List




```
def insert_before_after(new_node, left, right):
        left.forward = new_node
        new_node.back = left
        new_node.forward = right
        right.back = new_node
```

Double Linked List

- Need to find the insertion point:
  - Slightly tricky, because if the inserted key is larger than the present key, we do not want to circle around
  - Special case: The key to be inserted is smaller, so the new node becomes the head.
  - In which case we insert between the head and head.back
    - Even if they are the same node

Double Linked List

- Special case if we insert at the beginning, because we then need to reset self.head
  - Notice that it says OLL.insert_before_after because this is a class method

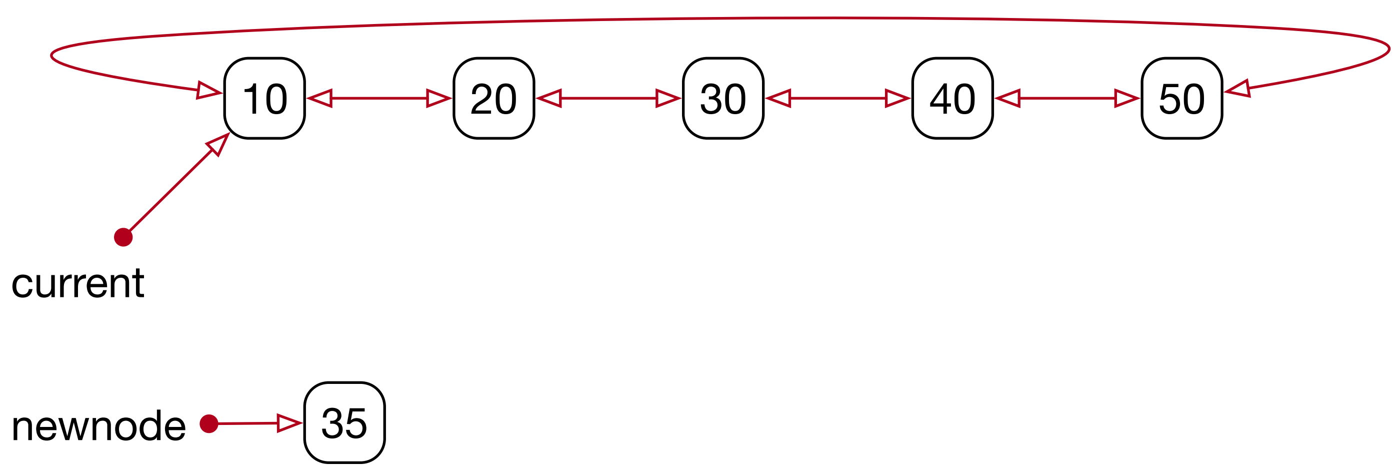
```
if self.head:
            current_node = self.head
            if my_key < self.head.key:
                OLL.insert_before_after(new_node,
                                        self.head,
                                        self.head.forward)
                self.head = new_node
```

Double Linked List

- Otherwise, we need to find the insertion point
    - Start out with current_node = self.head
    - Then move to the right until current_node.forward has a larger key
        - This gives us the insertion point

Double Linked List

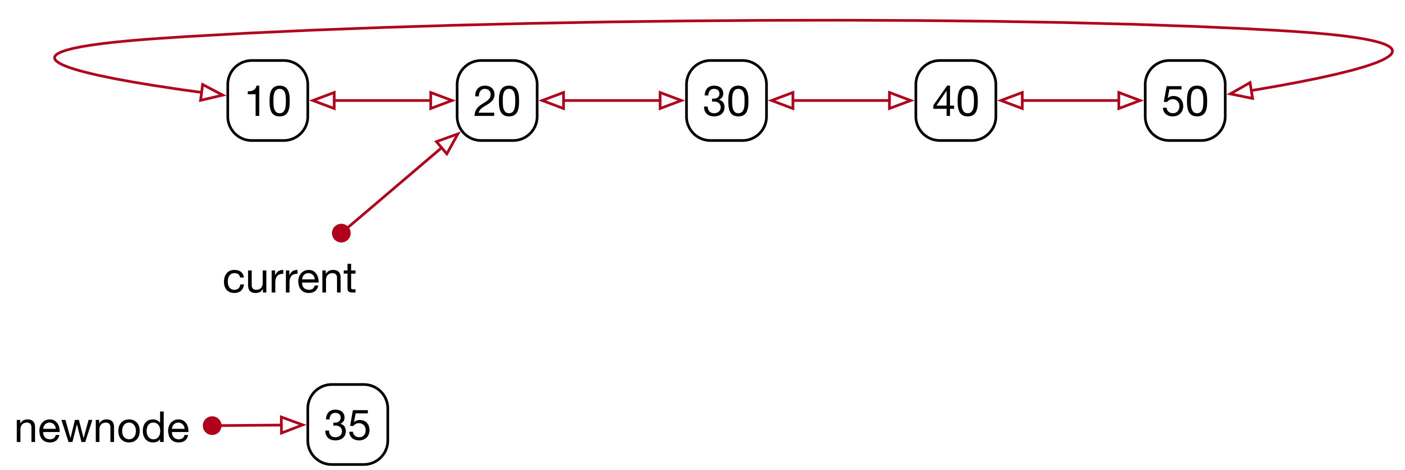
- Finding the insertion point
  - Already excluded that we need to insert before the head



  - key is 35, which is more than current.forward.key
  - Move current to the left

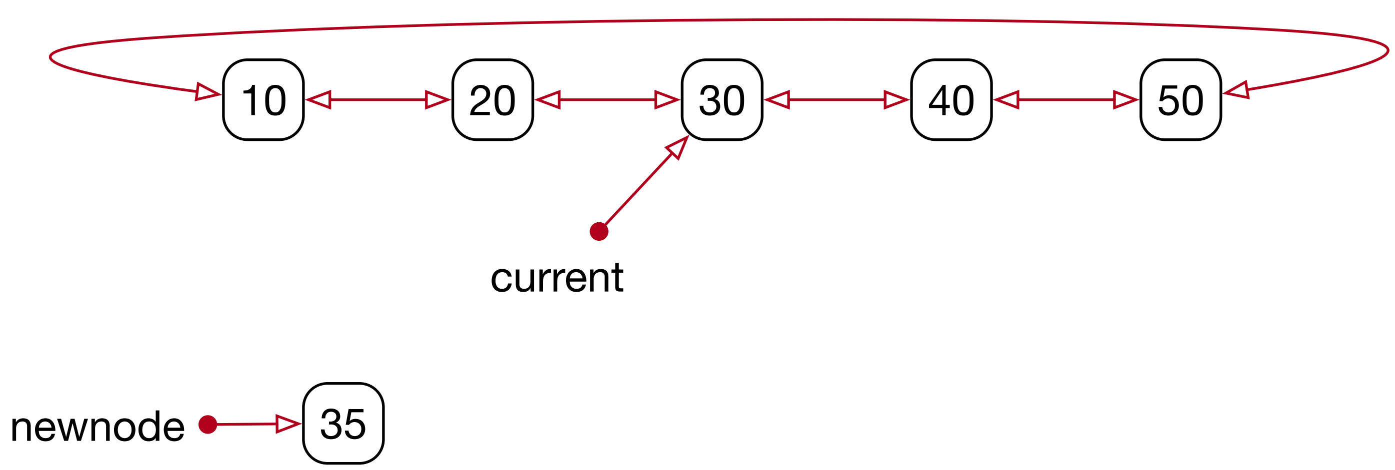
Double Linked List

- Finding the insertion point
  - current.forward.key is 30 is
  - still less than 35
  - move



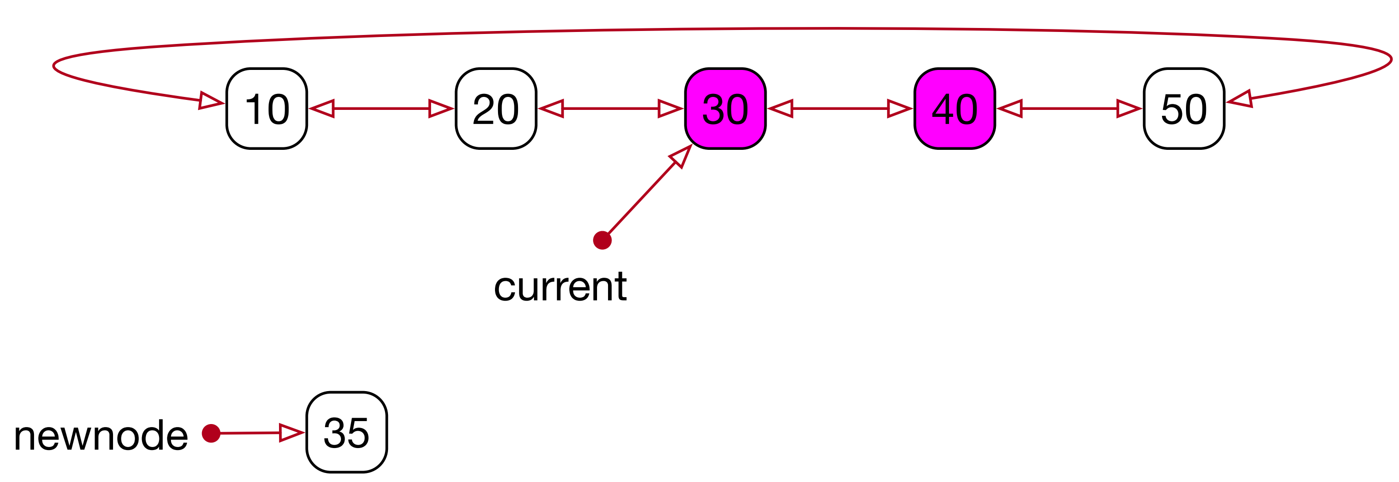
Double Linked List

- Finding the insertion point:
    - But not any longer: current.forward.key is 40



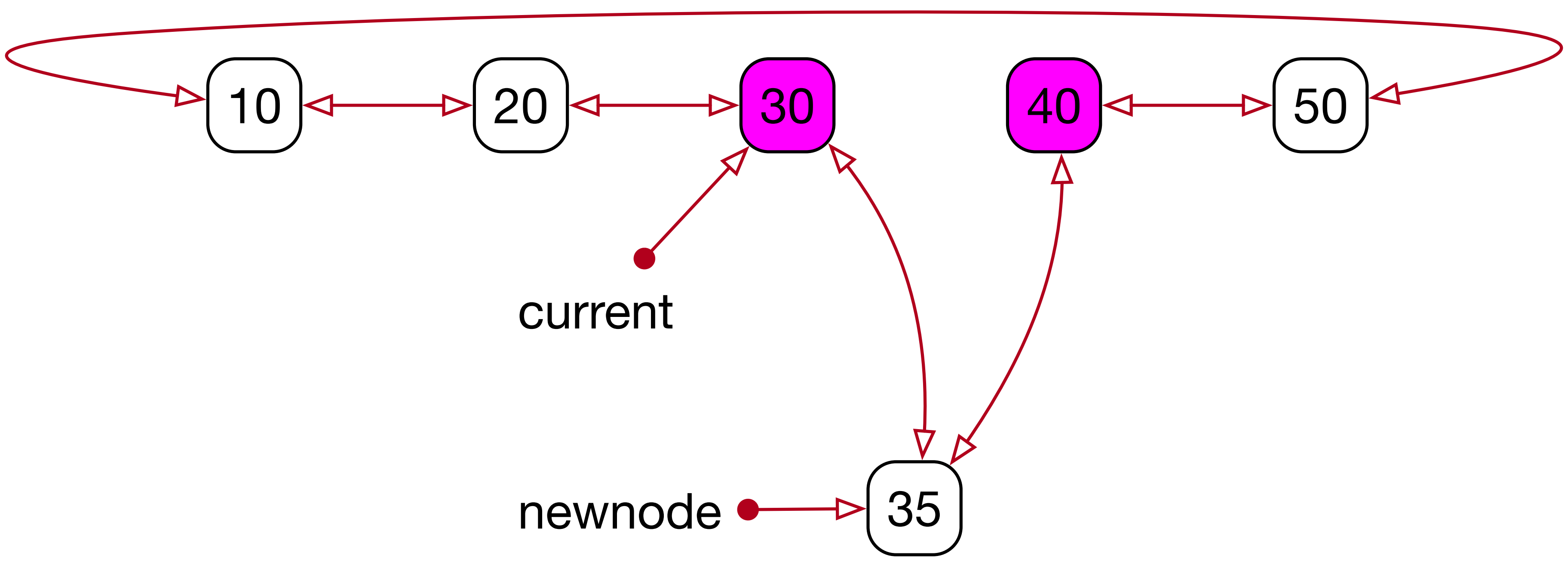
Double Linked List

- Finding the insertion point
  - Now we have found the insertion point
  - Insert between current and current.forward



Double Linked List

- Found the insertion point and inserted



Double Linked List

- Walk through the list
  - Can but not have to use iterators

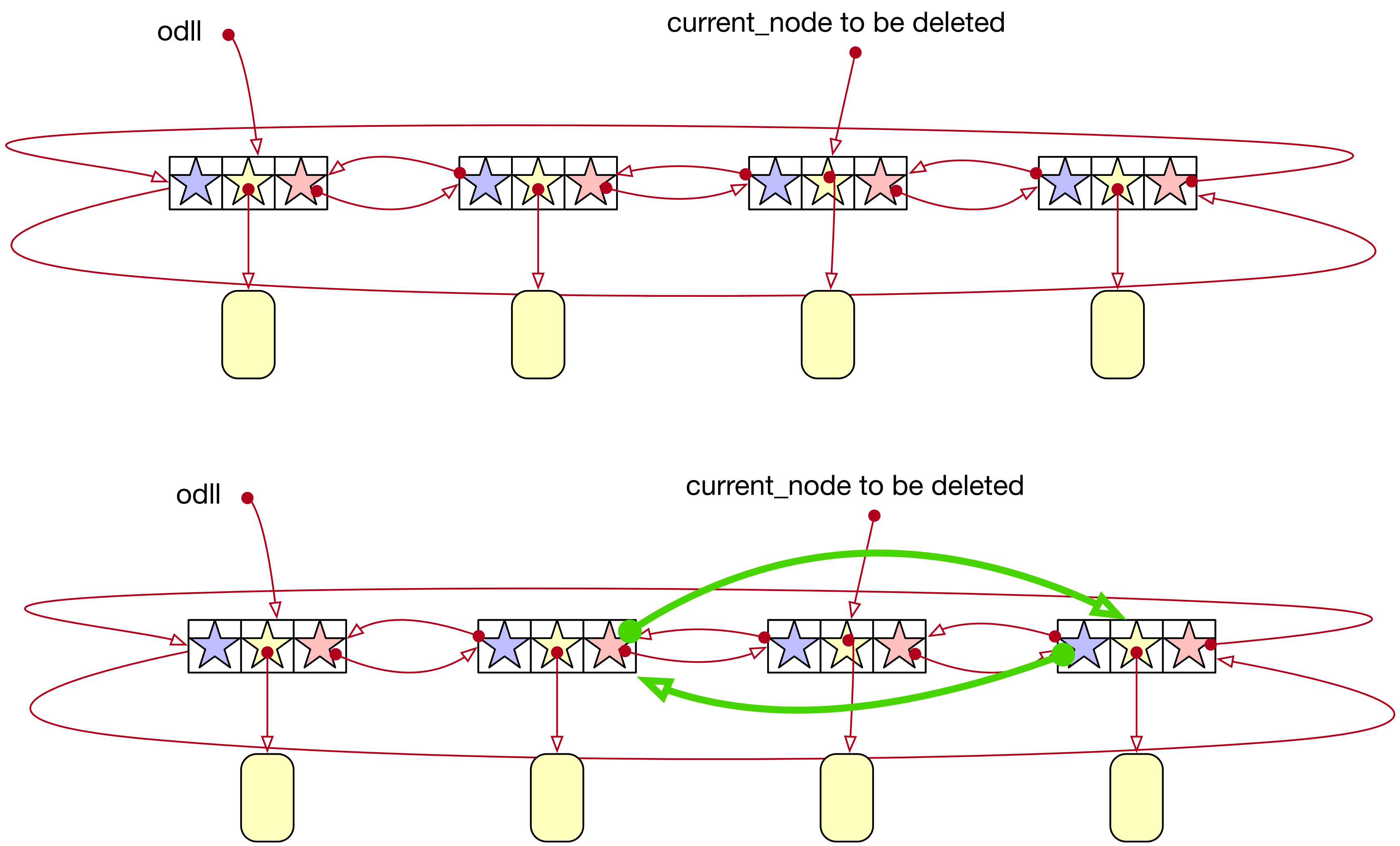
```
def show(self):
        if not self.head:
            print('empty')
            return
        print(self.head.key, self.head.record, sep=': ')
        current_node = self.head.forward
        while current_node != self.head:
            print(current_node.key, current_node.record, sep= ': ')
            current_node = current_node.forward
```

Double Linked List

- Deleting a record
  - Need to find the record first
    - Then delete the node
      - Special case:
        - This is the only node
          - In which case forward and back pointer point to the same node

Double Linked List

- Deleting a node from several nodes:



Double Linked List

- Deleting the only node
    - Just set self.head to None

```
def delete(self, key):
    current_node = self.head
    if current_node.forward == current_node:
        # there is only one node left
        self.head = None
```

Double Linked List

- Otherwise:
  - Use the current_node pointer (effectively an iterator) in order to find the node to be deleted
  - But be careful, because the key might not be there
    - After going to the next node, check that we are not at the beginning

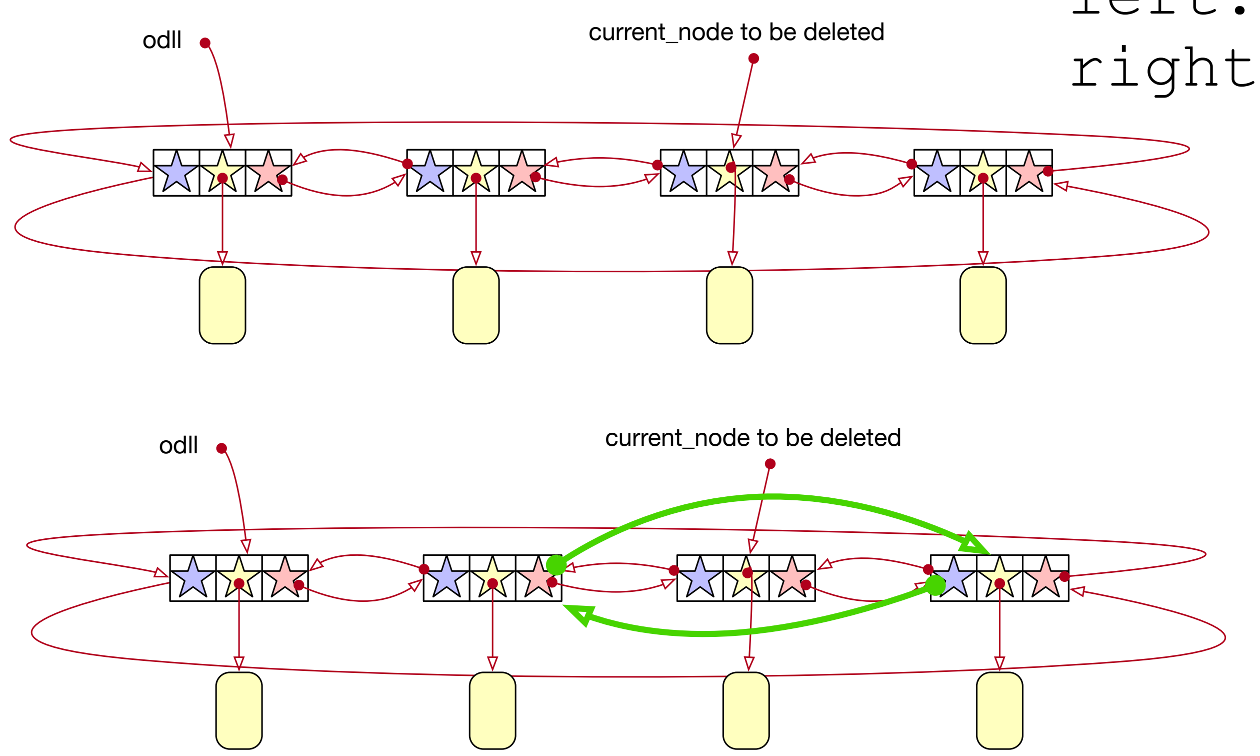
Double Linked List

```
def delete(self, key):
        current_node = self.head
        if current_node.forward == current_node:
            # there is only one node left
            self.head = None
            return
        while True:
            if current_node.key == key:
                self.delete_node(current_node)
                return
            else:
                current_node = current_node.forward
#Check that we have not reached the beginning of the list
                if current_node == self.head:
                    return
```

Double Linked List

- Deletion itself is fairly simple

```
def delete_node(self, current_node):
        if self.head == current_node:
            self.head = current_node.forward
        left = current_node.back
        right = current_node.forward
        left.forward = right
        right.back = left
```



Double Linked List

- But needs to take care of the case where we delete the head of the list
  - Though we can verify that we only need to reset head.

```
def delete_node(self, current_node):
        if self.head == current_node:
            self.head = current_node.forward
        left = current_node.back
        right = current_node.forward
        left.forward = right
        right.back = left
```

Double Linked List

- Performance
  - Storage costs
    - Python is generous with using storage
      - Each object has a number of fields
    - If we implement in a high performance language like C
      - Per record, need a node with three pointers
        - 3*32 or 3*64 bits = 12B, 24B per object

Double Linked List

- Timing measured in number of nodes
  - Double linked list as a Stack
    - insert / delete at head (1 node)
  - Double linked list as a Queue
    - insert at head / delete at tail (1 / 2 nodes)
    - insert at tail / delete at head (2 / 1 nodes)
  - Sometimes Stack and Queue are combined in a single structure: Deque

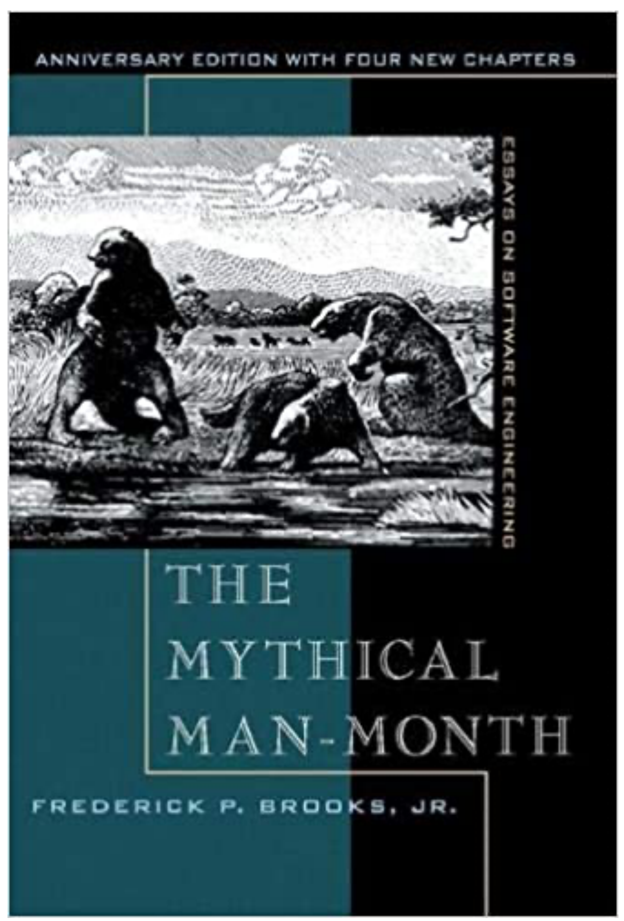
Double Linked List

- Performance
    - Ordered linked list:
        - Finding a record, inserting a record, deleting a record
        - Timing is $n/2$ nodes on average

Software Engineering

- This is highly non-trivial code
  - I know because of the mistakes that I make

Software Engineering

- Two core problems of Software Engineering:
  - How to get people to work on code together successfully
    - 

Software Engineering

- Second problem:
  - How can we guarantee correctness of code
    - Formal methods
      - Not popular because very difficult
    - Testing
      - Difficult because we need to test for all things that are likely to go wrong

Software Engineering

- Test generation:
  - Think about all things that could have an influence
    - E.g. node deletion: location of node with respect to other nodes
      - Lonely node
      - Node at the beginning
      - Node at the end
      - Node in the middle
      - Node first after head
      - ...

Software Engineering

- Test Generation:
  - Write a test for all of these cases

Software Engineering

- Idea of unit tests
  - Divide tasks into modules
    - Implementing a cyclical ordered linked list would be a module
  - Modularization:
    - Makes design easier
    - Allows small groups to generate software
    - Can test already at the unit level

Software Engineering

- Unit tests in Python
  - Can have code that only executes if the module is the one that is called
  - But not if the module is imported

```
if __name__ == '__main__':
    oll = OLL()
    oll.insert('z',1)
    oll.insert('a', 100000)
    oll.insert('d', 2)
    oll.insert('e', 3)
    for _ in range(10):
        x = random.getrandbits(16)
        print(x)
        oll.insert( 3*str(x),x )
    oll.show()
```

Software Engineering

- One of the big problems is **software maintenance**
  - The programmer or someone else will add functionality and / or change the implementation
  - A simple code addition can *break code elsewhere*
  - Therefore:
    - Test the interface in the unit test
    - So that an addition / modification that breaks an interface can be caught

Software Engineering

- Hence:
  - Test your algorithms
    - By making all case distinction
    - And verifying your code with paper and pencil
      - This is formal method (very) light

Software Engineering

- Hence:
  - Test your implementation
    - By making all case distinctions
      - And writing test code for them
        - Even better: print out what the result of your test should be

Software Engineering

- What to do when you detect an error
  - **READ the ERROR MESSAGE**
  - Identify the location of the fault
    - The error happened there **or** before on the execution path
  - Adorn your code with additional print statements
    - To help you locate the statement that causes the error

Software Engineering

- What type of errors should you expect:
  - Typos and similar mechanical errors that are not detected by the UI
  - Violated **assumptions**
    - You deal with that by making your assumptions explicit
      - You do not need to write them down, just acknowldge them

Software Engineering

- Debugging is more of an art than science
    - Experience helps a lot