빅데이터 혁신공유대학

파이썬으로 배우는 데이터 구조

한동대학교 전산전자공학부
김영섭 교수

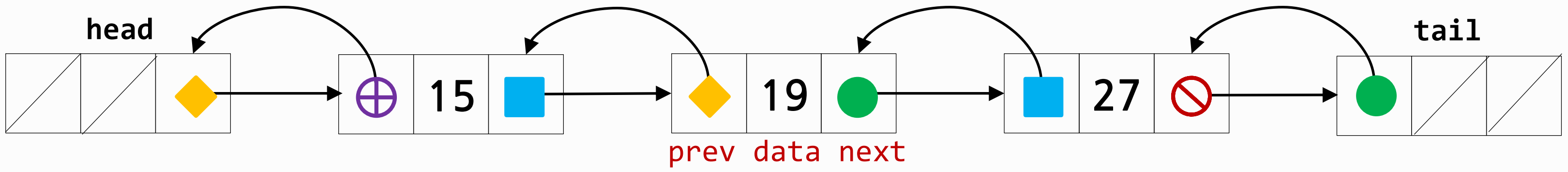
Data Structures in Python
Chapter 3 - 4

- Doubly Linked List - Structures
- **Doubly Linked List - Operations**
- Doubly Linked List - DequeCircular

Agenda

- **`DoublyLinked`** Class ADT
  - Basic Operations:
  - Key Operations:
  - Other Operations

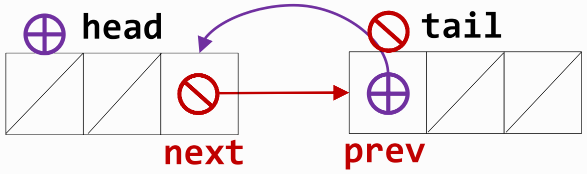
DoublyLinked Class ADT



- Basic Operations:
  - `__init__()`, `__str__()`,
  - `begin()`, `end()`, `is_empty()`, `size()`, `find()`, `clear()`
- Key Operations:
  - `remove()`
  - `insert()`
- Other Operations: (left as coding exercise)
  - `reverse()`
  - `__iter__()`

Basic Operations: begin() and end()

- **begin()** returns 1st node (reference) that the head's **next** points to. It may return the **tail** node. **end()** returns the **tail** node (reference).
- The list must be empty if what **begin()** returns is the same what **end()** returns.




```
def begin(self):
    return self.__head.next

def end(self):
    return self.__tail

def is_empty(self):
    return self.begin() == self.end()
```

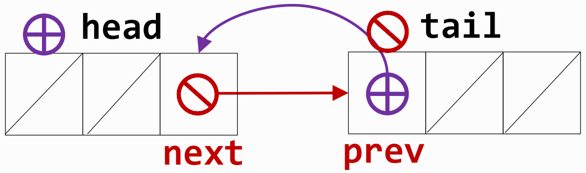
```
alist = DoublyLinked()

print(alist.begin())
print(alist.end())
print(alist.is_empty())
```

```
<__main__.Node object at 0x000001B3D089AB80>
<__main__.Node object at 0x000001B3D089AB80>
True
```

Basic Operations: begin() and end()

- **`begin()`** returns 1st node (reference) that the head's **`next`** points to. It may return the **`tail`** node. **`end()`** returns the **`tail`** node (reference).
  - For easy coding, it is recommended to use **`begin()`** and **`end()`** rather than head and tail. That is a reason we use **`__head`** and **`__tail`**.




```
def begin(self):
    return self.__head.next

def end(self):
    return self.__tail

def is_empty(self):
    return self.begin() == self.end()
```

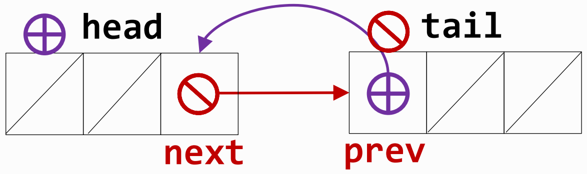
```
alist = DoublyLinked()

print(alist.begin())
print(alist.end())
print(alist.is_empty())
```

```
<__main__.Node object at 0x000001B3D089AB80>
<__main__.Node object at 0x000001B3D089AB80>
True
```

Basic Operations: is_empty()

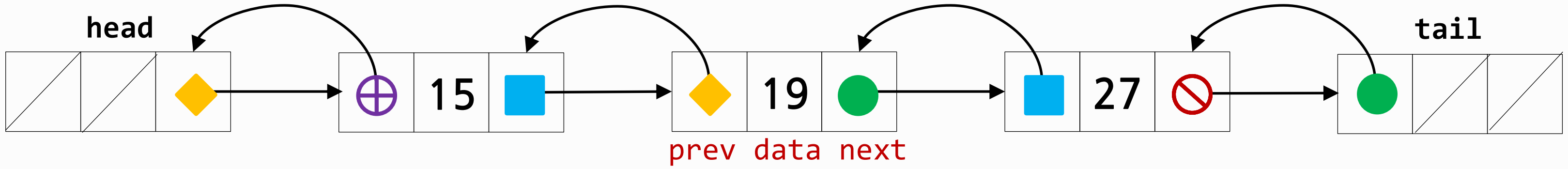
- **`is_empty()`** returns **`True`** if the list is empty, False otherwise.
- The list must be empty if what **`begin()`** returns is the same what **`end()`** returns.




```
def begin(self):
    return self.__head.next

def end(self):
    return self.__tail

def is_empty(self):
    return self.begin() == self.end()
```

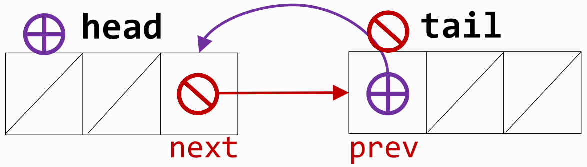


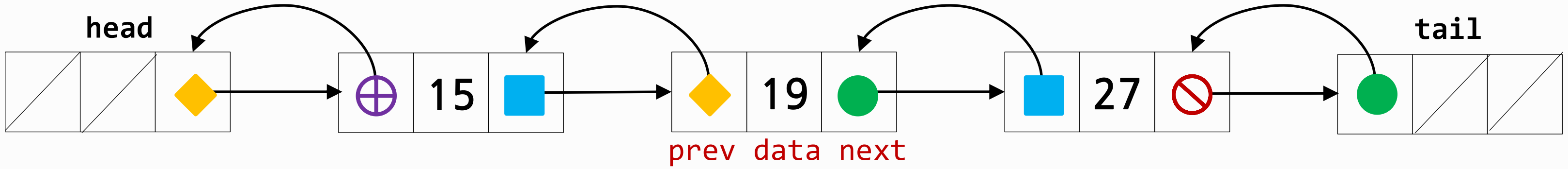
Basic Operations: size()

- **`size()`** returns the number of node in the list.
  - The two sentinel nodes are **not** counted for the size of the list.

```
def size(self):
    count = 0
    curr = self.begin()
    while curr != self.end():
        count = count + 1
        curr = curr.next
    return count
```

```
•  initialize count
•  curr is set to the 1st node
•  loop through the list
•      increment count by 1
•      go for the next
•  return count
```



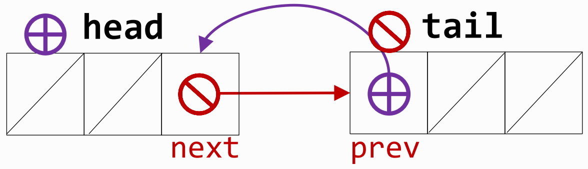


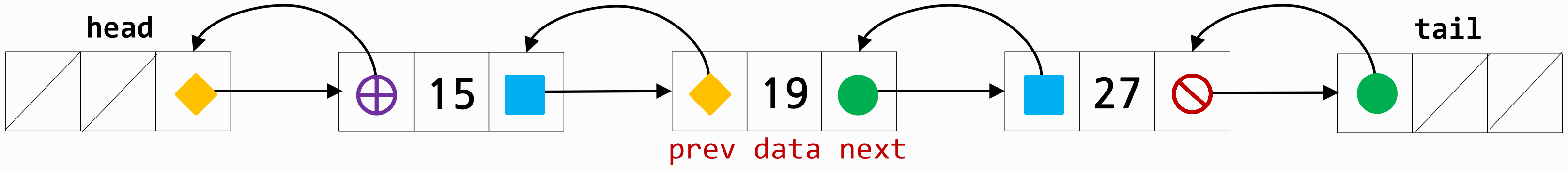
Basic Operations: find(data)

- `find()` returns the node (reference) with the **data**, **None** if not found.
  - One method fits for all cases. No special case is needed.
  - Pay attention that we cannot use the expression such as "**`while curr:`**" since `self.end()` does not return **None** but the **tail** node (reference).

```
def find(self, data):
    curr = self.begin()
    while curr != self.end():
        if curr.data == data:
            return curr
        curr = curr.next
    return None
```

- **curr** is set to the 1st node
- loop through the list
- check for the matching
- return **curr** matched
- go for the next since no match
- return **None** since not found

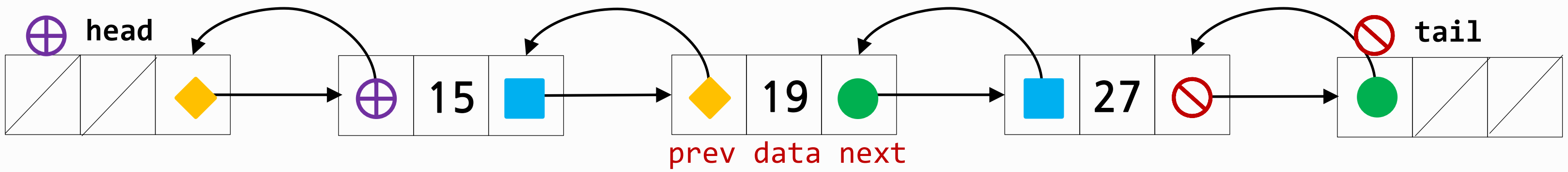




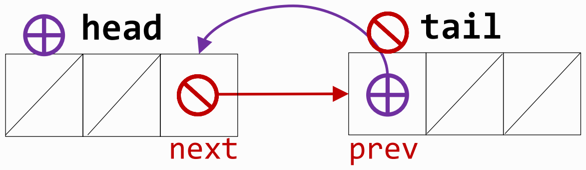
Basic Operations: clear()

- **`clear()`** removes all the nodes in the list and becomes an empty list.
  - The following two statements make no nodes in the list be referenced. Then the Python garbage collector, **`gc.collect()`,** kicks in automatically.
  - To invoke it by yourself, import **`gc`**.

before:



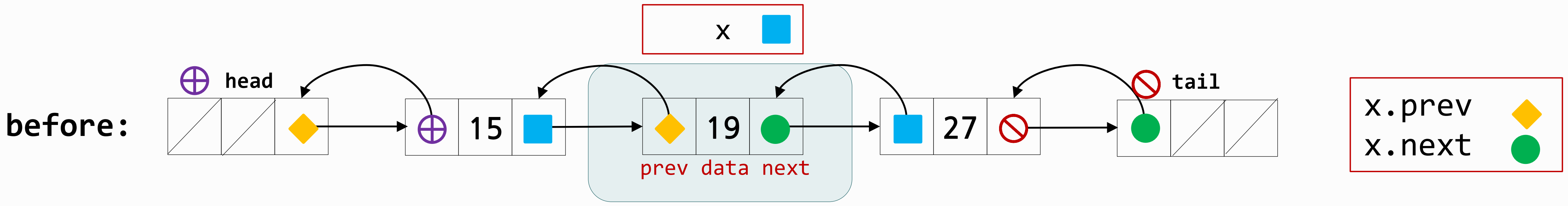
after:




```
def clear(self):
    self.__head.next = self.__tail
    self.__tail.prev = self.__head
    #gc.collect()    # unnecessary
```

Key Operations: remove(x)

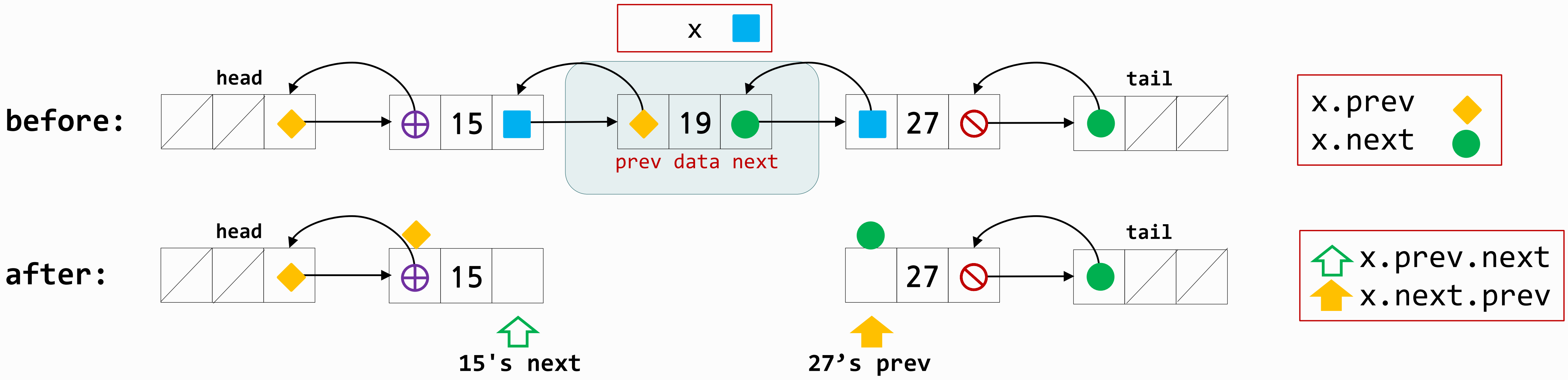
- **`remove()`** removes the node x only if x is a node in the list. If not, return None.



- Concept:
  - Using given the node x, remove by itself, but keep the links alive.
  - The node **`15's next`** must set to the **`node 27,`** the green circle.
    The node **`27's prev`** must set to the **`node 15,`** the orange diamond.
    as shown in the following figure.

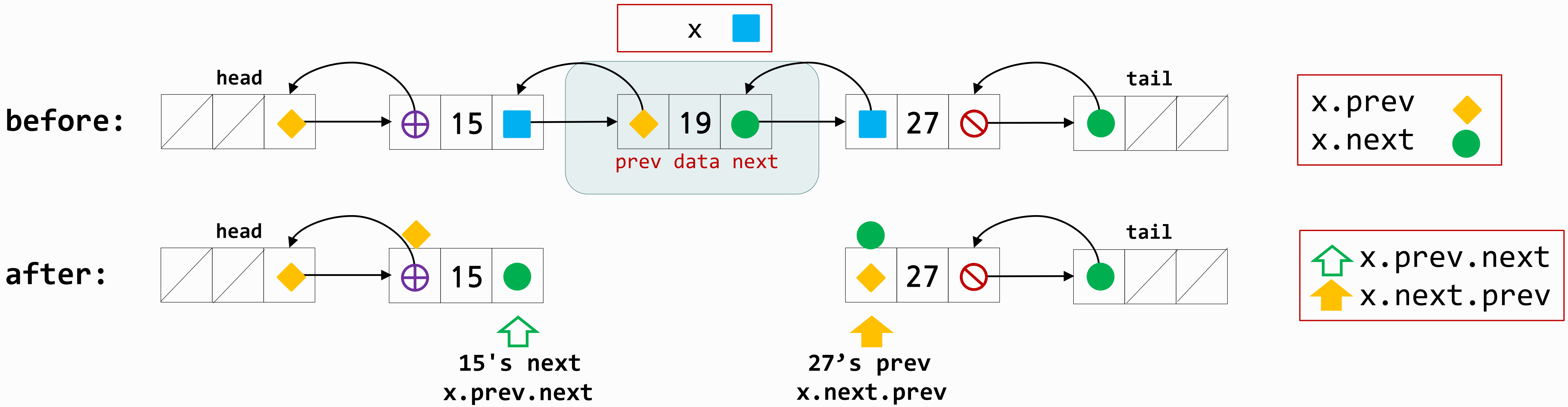
Key Operations: remove(x)

- `remove()` removes the node x only if x is a node in the list. If not, return None.



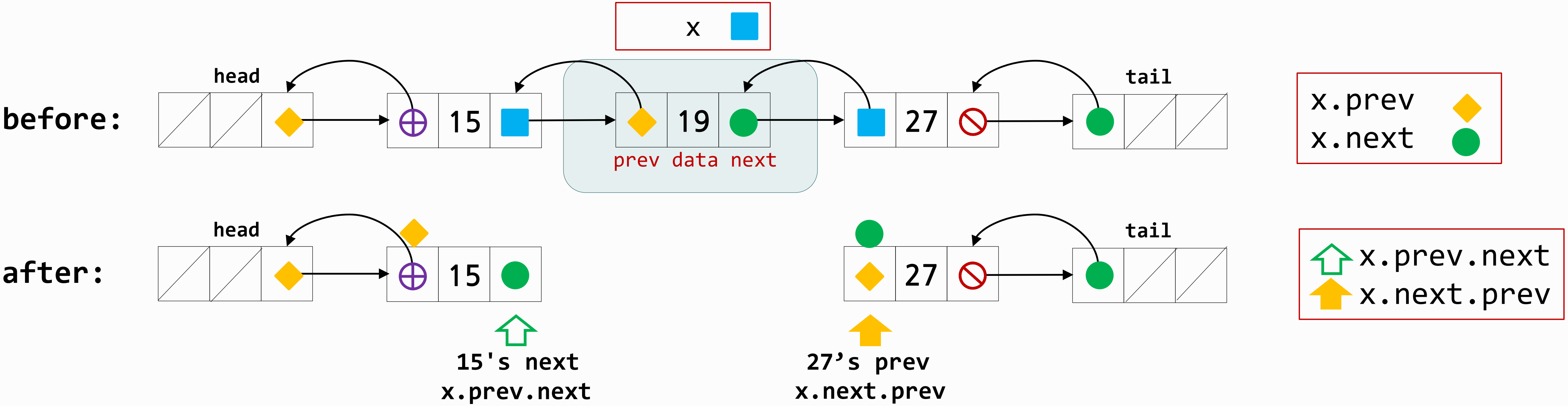
Key Operations: remove(x)

- **`remove()`** removes the node x only if x is a node in the list. If not, return None.



Key Operations: remove(x)

- **`remove()`** removes the node x only if x is a node in the list. If not, return None.

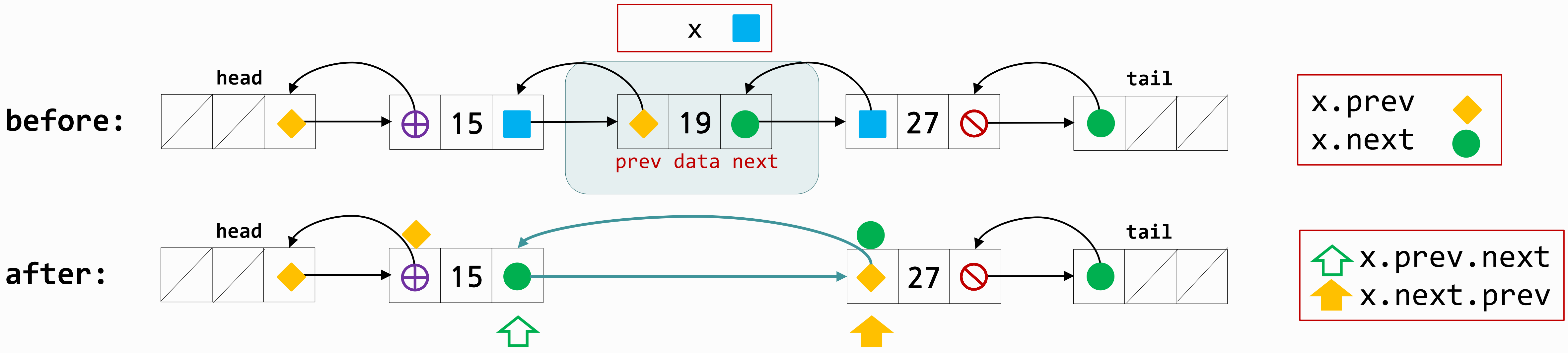


- Since the node **`15's next`** is **`x.prev.next,`** the node **`27's prev`** is **`x.next.prev,`**

```
def remove(self, x):
    if x == None: return None
    x.prev.next = x.next
    x.next.prev = x.prev
```

Key Operations: remove(x)

- **`remove()`** removes the node x only if x is a node in the list. If not, return None.

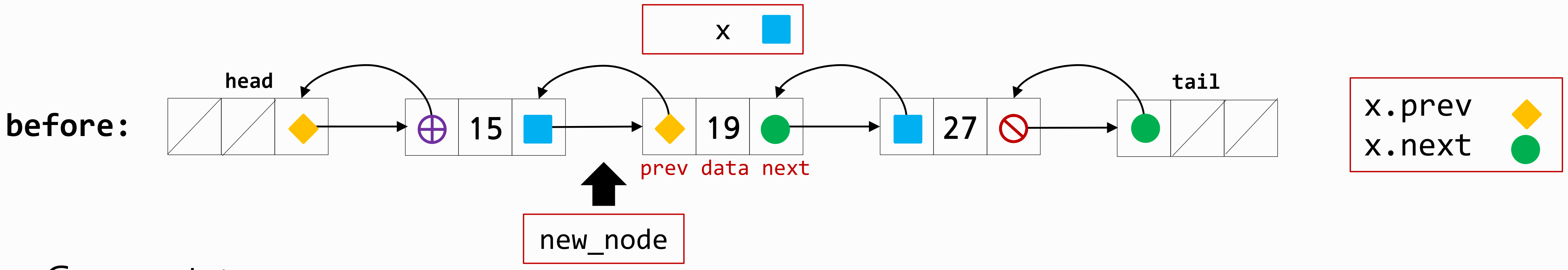


- Since the node **`15's next`** is **`x.prev.next,`** the node **`27's prev`** is **`x.next.prev,`**

```
def remove(self, x):
    if x == None: return None
    x.prev.next = x.next
    x.next.prev = x.prev
```

Key Operations: insert(data, x)

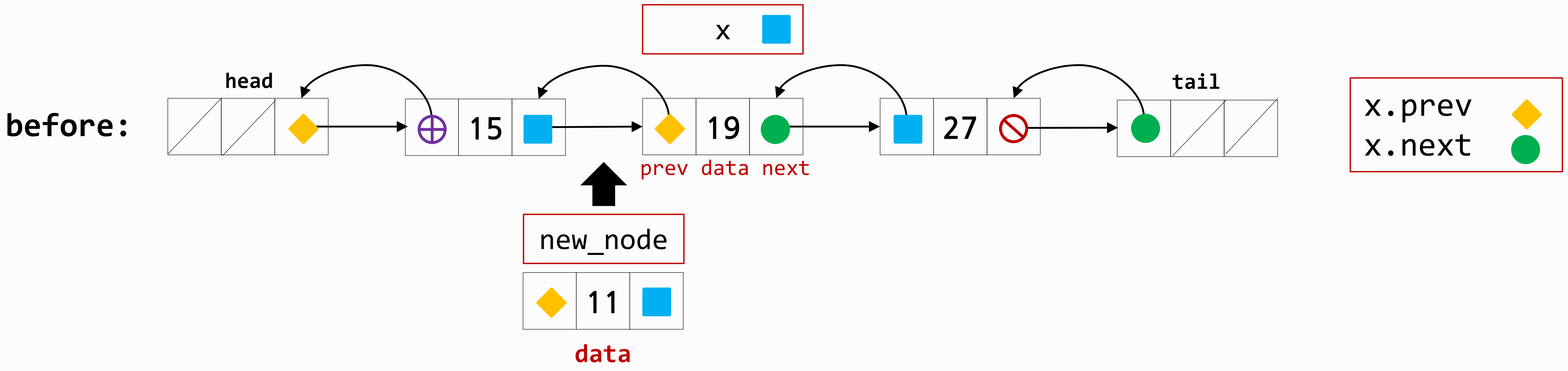
- `insert()` inserts **a new node** with **data** at the position of the node **x** in the list.



- Concepts:
  - The new node goes into between the node 15 and the node 19.
  - The new node pushes the node 19 to the right.
  - The new links must be made between the nodes **15**, the **new node** and the node **19**.

Key Operations: insert(data, x)

- `insert()` inserts **a new node** with **data** at the position of the node **x** in the list.

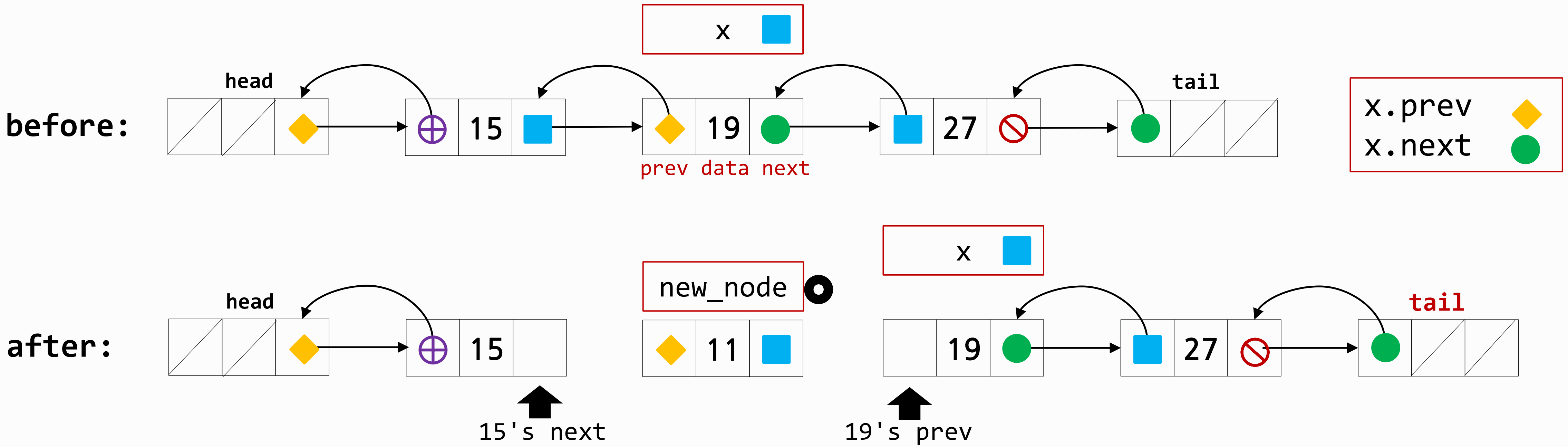


- Instantiate a **new node** between the node15 and 19 with the following settings:
  - (1) **data =** data provided with an argument, 11 for example.
  - (2) **prev** = the node 15
  - (3) **next** = the node 19
  - Then, the new node would be instantiated: **new_node = Node(data, x.prev, x)**

```
def __init__(self, data=None, prev=None, next=None):
```

Key Operations: insert(data, x)

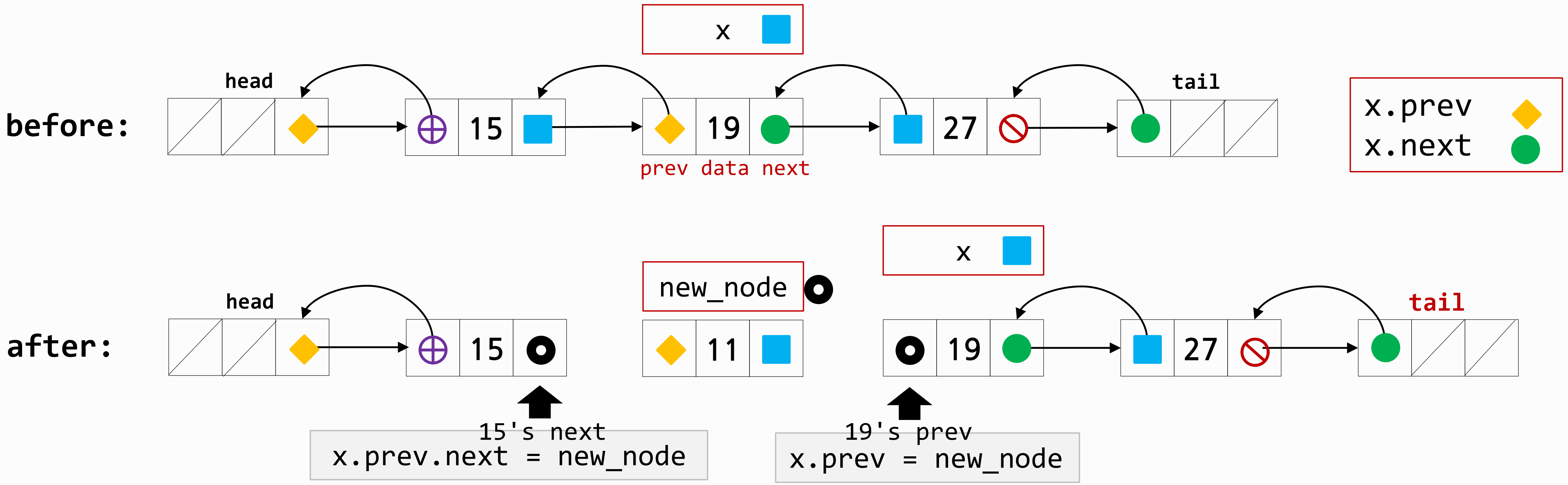
- `insert()` inserts **a new node** with **data** at the position of the node **x** in the list.



  - Now the new_node is linked with the node 15 and the node 19.
  - The nodes 15 and 19, however, must link to the new node.
  - Let us suppose the **new_node** is instantiated, denoting with **a donut shape dot.**
  - **This new node's reference** must go in the node **`15's next`** and the node **`19's prev`**.
  - The node 15's next is **`x.prev.next`**, and the node 19's prev is **`x.prev.`**

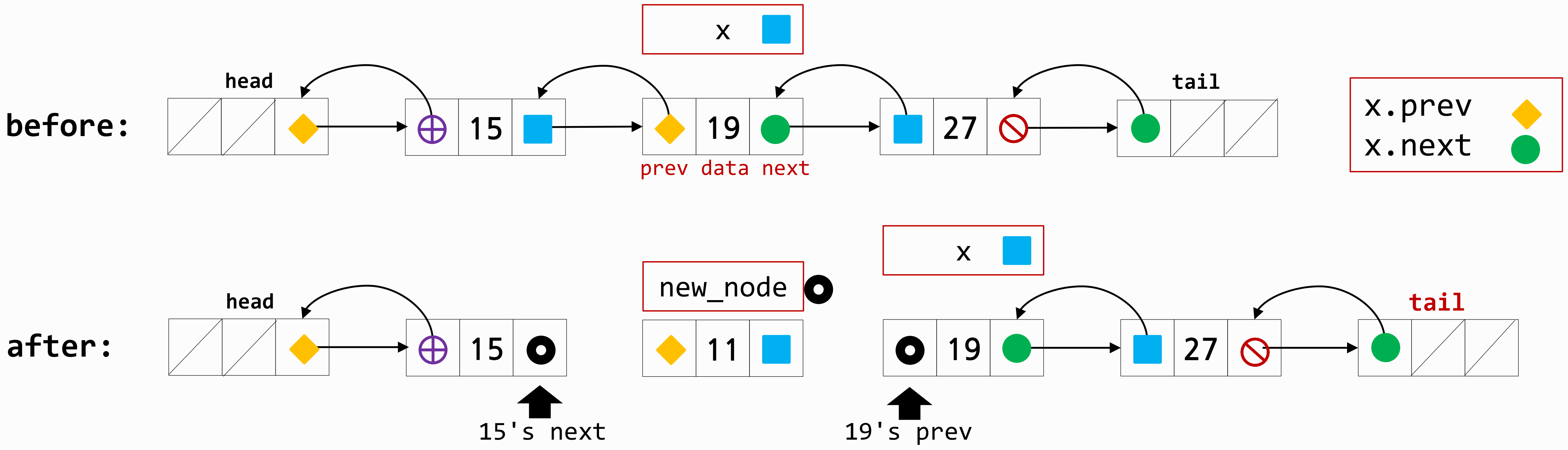
Key Operations: insert(data, x)

- `insert()` inserts **a new node** with **data** at the position of the node **x** in the list.



Key Operations: insert(data, x)

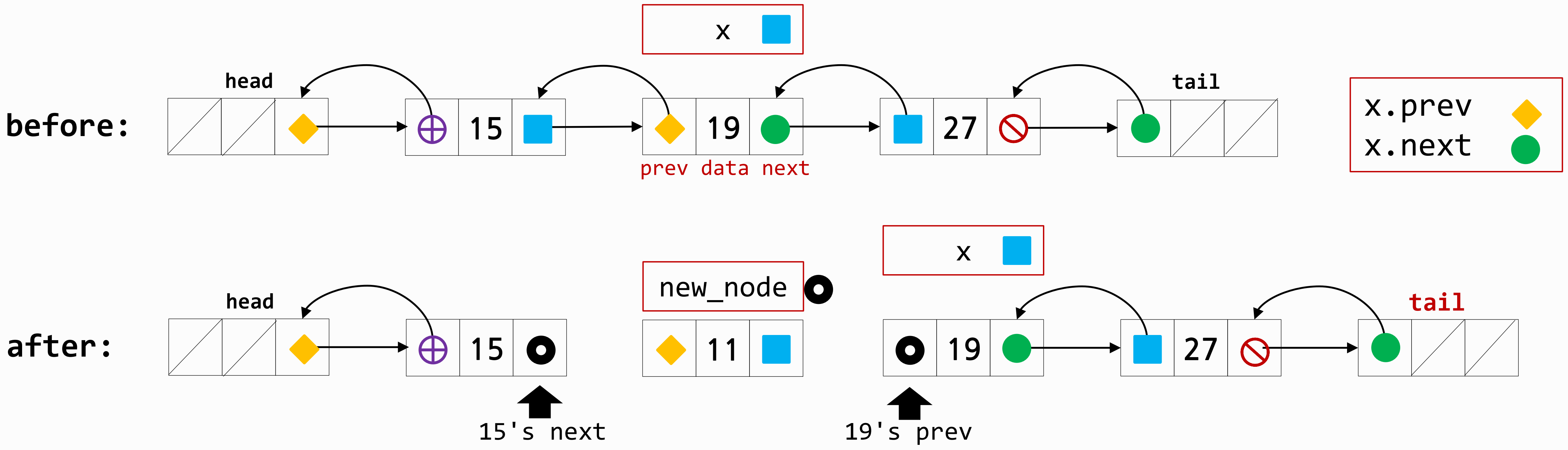
- **`insert()`** inserts **a new node** with **data** at the position of the node **x** in the list.




```
def insert(self, data, x):
    new_node = Node(data, x.prev, x)
    x.prev.next = new_node
    x.prev      = new_node
```

Key Operations: insert(data, x)

- **`insert()`** inserts **a new node** with **data** at the position of the node **x** in the list.




```python
def insert(self, data, x):
    new_node = Node(data, x.prev, x)
    x.prev.next = new_node
    x.prev      = new_node
```

- The node **x** can be any node in the list including the `1st` node and the `tail` node.
- **`begin()`** returns the1st node, and **`end()`** returns the `tail` node, respectively.

Key Operations: remove() and insert()

- With two operations, remove() and insert(), Some methods may be simply coded.
- For example:
  - `pop() - remove the last node`
    - `self.`**`remove`**`(self.end().prev)`
  - `popleft() - remove the first node`
    - `self.`**`remove`**`(self.begin())`
  - `append(data) - insert a node at the end`
    - `self.`**`insert`**`(data, self.end())`
  - `appendleft(data) - insert a node at the front`
    - `self.`**`insert`**`(data, self.begin())`

Summary

- Doubly Linked List Class ADT
  - Two sentinel nodes helps simplifying some operations.
  - Use `begin()` and `end()` method instead of accessing `__head` and `__tail` directly.
  - The time complexity of two key operations such as `remove()` and `insert() is O(1).`

Data Structures in Python
Chapter 3 - 4

- Doubly Linked List - Structures
- **Doubly Linked List - Operations**
- Doubly Linked List - DequeCircular

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University