# Data Structures in Python Chapter 3

- Linked List
- OOP Inheritance
- ListUnsorted Class
- ListSorted Class
- Iterator
- **Doubly Linked List - Structures**
- Doubly Linked List - Operations
- Doubly Linked List - DequeCircular

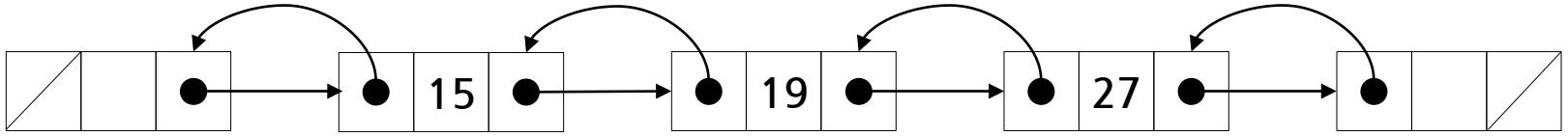
*Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University*

# Agenda

- Doubly Linked List
  - Introduction
    - Data Structure
    - Node Structure
    - Using sentinel nodes and their advantages
  - Node Class ADT
    - `__init__()`, `__str__()`,
      `get_data()`, `set_data()`, `get_prev()`, `set_prev()`, `get_next()`, `set_next()`
    - `debug_headtail()`
  - Node Class as an Inner class

# Doubly Linked List - Definition

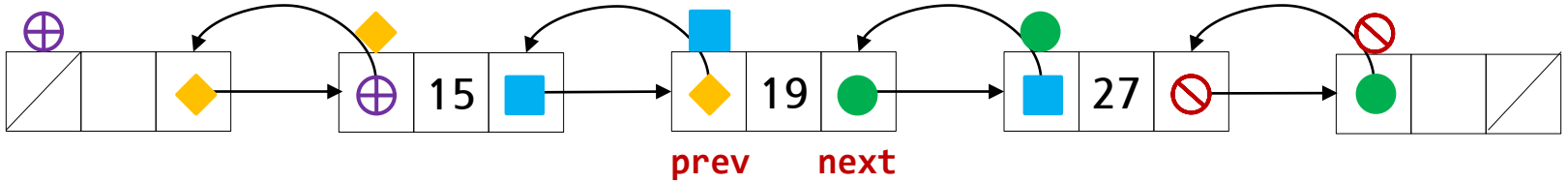
- It is a linked list in which each node keeps an explicit reference to the node **before** it and a reference to the node **after** it.




- It allows a greater variety of O(1)-time update operations, including insertions and deletions at arbitrary positions within the list.

# Doubly Linked List - Definition

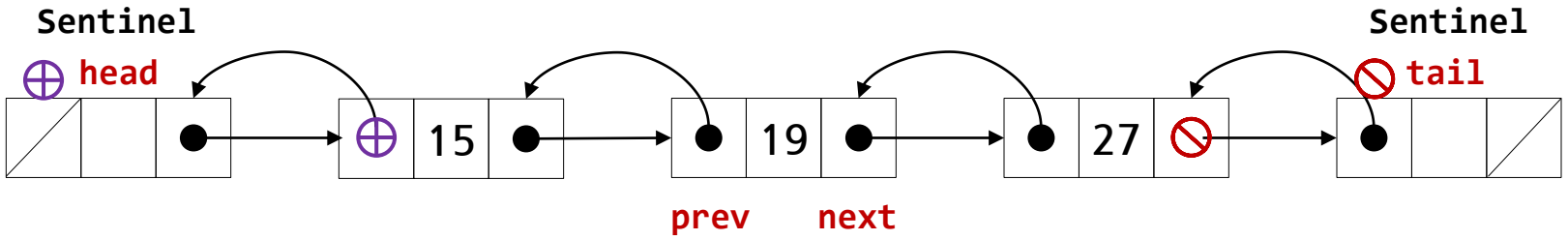
- We continue to use the term "**next**" for the reference to the node that follows another, and we introduce the term "**prev**" for the reference to the node that precedes it.




⊕ ◆ ■ ● ⊘ **Each dot** denotes **a reference of the node object** or an address of the memory segment of the node allocated or a unique id of the node; For example: **0x000001B3D089AB80**

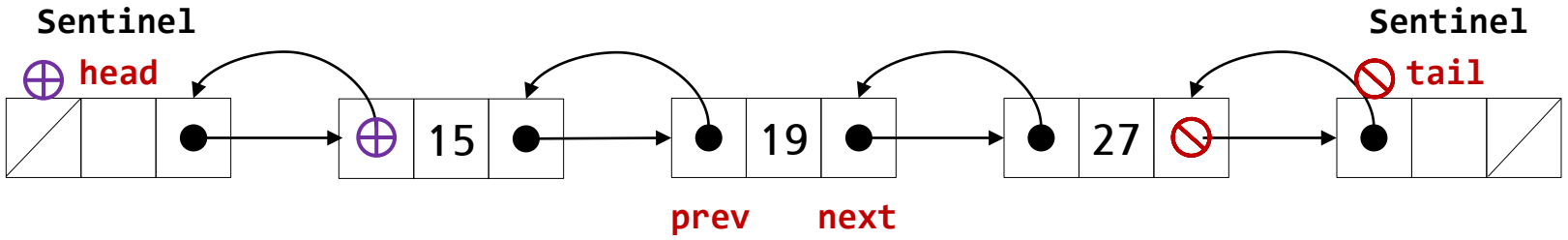
# Doubly Linked List - Sentinels

- In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list:
  - a **head** node at the beginning of the list, and a **tail** node at the end of the list.
  - **These "dummy" nodes** are known as **sentinels** (or guards), and they do not store elements of the primary sequence.



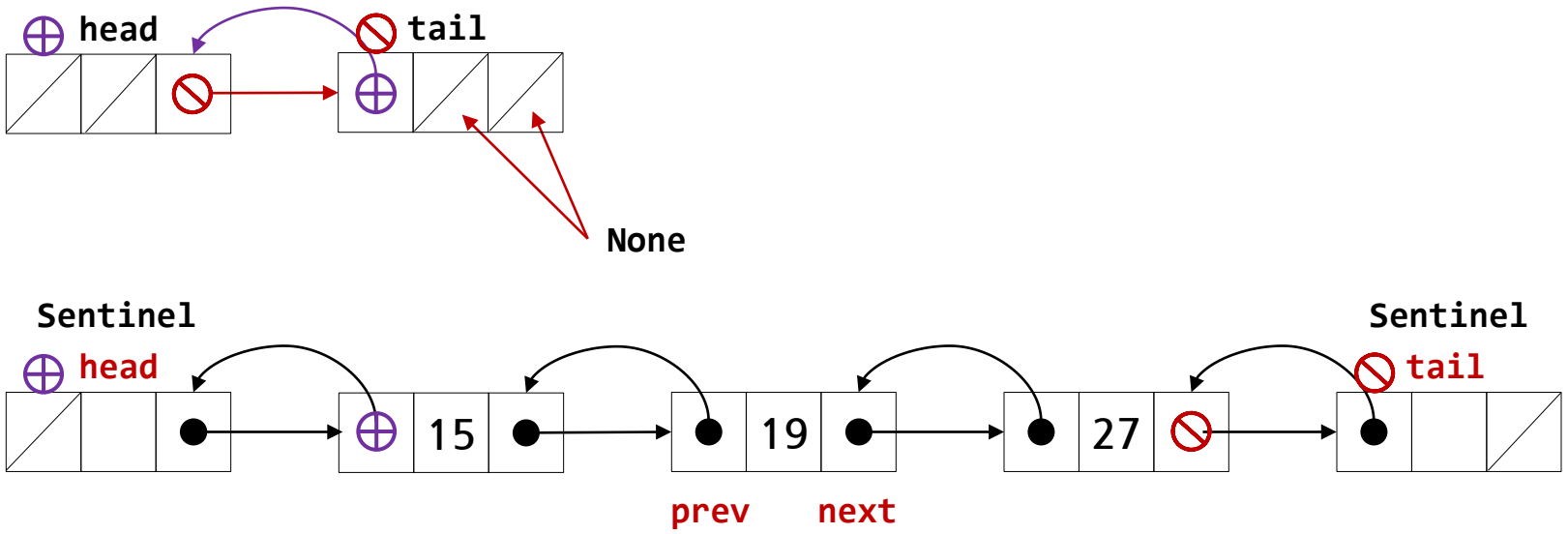
# Doubly Linked List - Sentinels

- Although we could implement a doubly linked list without sentinel nodes, the slight extra space devoted to the sentinels greatly simplifies the logic of our operations like a magic.
  - The head and tail nodes always exist and never change - only the nodes between them change.
  - We can treat either insertions or deletions in a unified manner since a node will always be inserted or deleted between a pair of existing nodes. No special cases necessary.



# Doubly Linked List - Sentinels

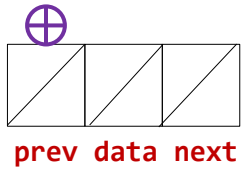
- When using sentinel nodes,
  - An empty list is initialized so that the **next** of the head points to the tail, and the **prev** of the tail points to the head; the remaining fields of the sentinels are set **None**;
  - For a nonempty list, the head's **next** will refer to a node containing the first real element of a sequence, just as the tail's **prev** references the node containing the last element of a sequence.



# Doubly Linked List - Data Structure

- Constructors for `Node` class and `DoublyLinked` class

```
class Node:
    def __init__(self, data = None, prev = None, next = None):
        self.__data = data
        self.__prev = prev
        self.__next = next
```

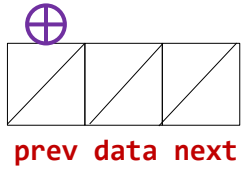


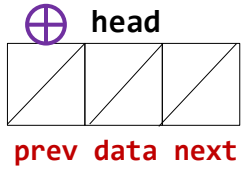
# Doubly Linked List - Data Structure

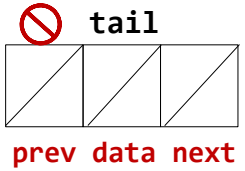
- Constructors for `Node` class and `DoublyLinked` class

```
class Node:
    def __init__(self, data = None, prev = None, next = None):
        self.__data = data
        self.__prev = prev
        self.__next = next
```

```
class DoublyLinked:
    def __init__(self):
        self.__head = Node()
        self.__tail = Node()
```



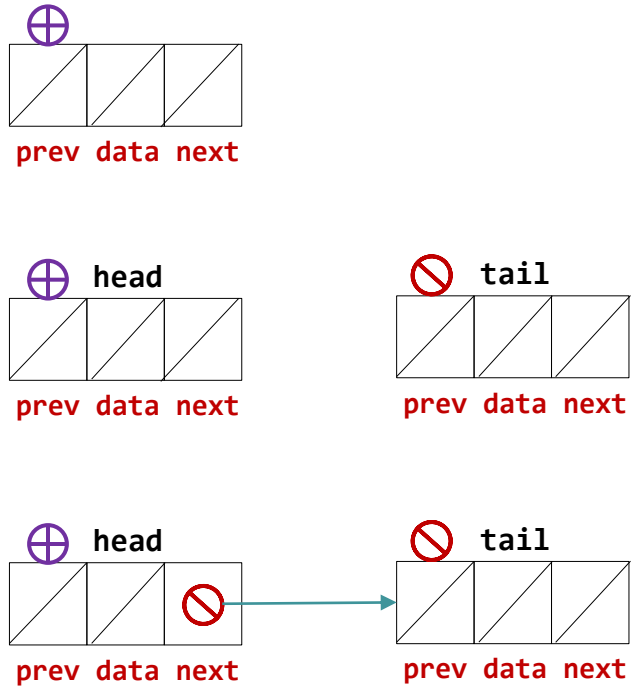



# Doubly Linked List - Data Structure

- Constructors for `Node` class and `DoublyLinked` class

```
class Node:
    def __init__(self, data = None, prev = None, next = None):
        self.__data = data
        self.__prev = prev
        self.__next = next
```

```
class DoublyLinked:
    def __init__(self):
        self.__head = Node()
        self.__tail = Node()
        self.__head.next = self.__tail
```

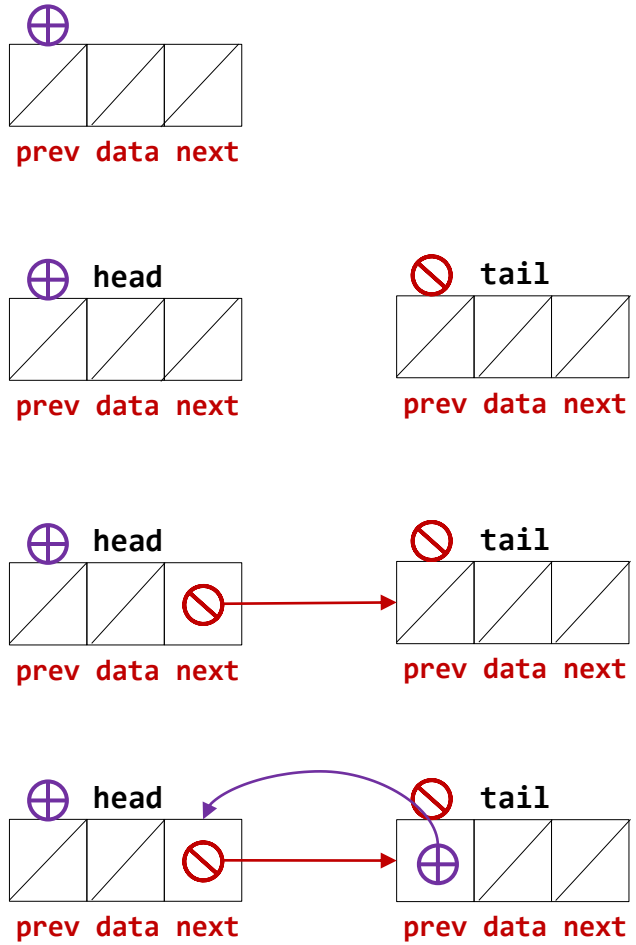


# Doubly Linked List - Data Structure

- Constructors for `Node` class and `DoublyLinked` class

```
class Node:
    def __init__(self, data = None, prev = None, next = None):
        self.__data = data
        self.__prev = prev
        self.__next = next
```

```
class DoublyLinked:
    def __init__(self):
        self.__head = Node()
        self.__tail = Node()
        self.__head.next = self.__tail
        self.__tail.prev = self.__head
```



# Doubly Linked List - Data Structure

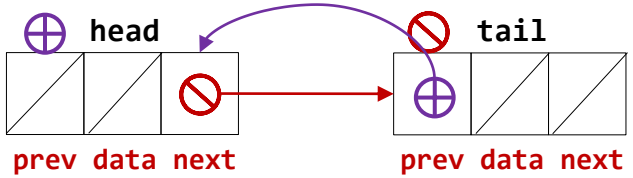
- Let us define the `Node` class as an **inner** class since it is used only `DoublyLinked`.

```
class DoublyLinked:
    class Node:
        def __init__(self, data = None, prev = None, next = None):
            self.__data = data
            self.__prev = prev
            self.__next = next
        ...

    def __init__(self):
        self.__head = self.Node()
        self.__tail = self.Node()
        self.__head.next = self.__tail
        self.__tail.prev = self.__head
    ...
```

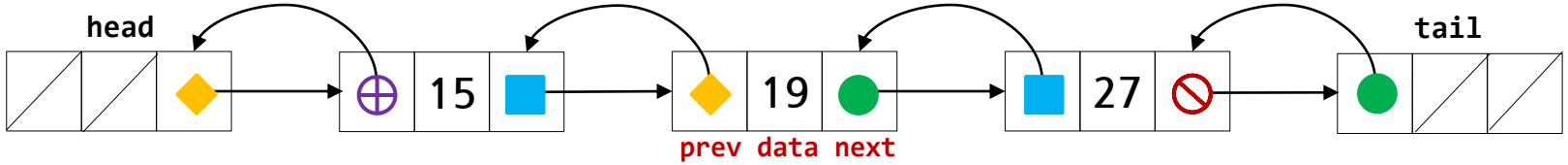
# Node Operations: __str__() in Node

- The color dots are representations of the references of the nodes. If you print them, for example, head and tail would be like these:

```
<__main__.Node object at 0x000001BA19CCA160>●
<__main__.Node object at 0x000001BA19CCACD0>●
```

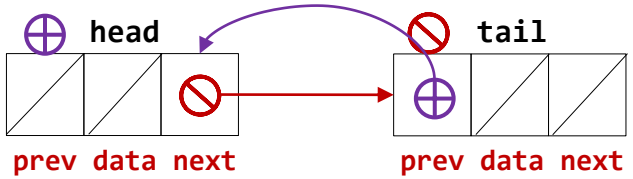



```
class Node:
    def __init__(self, data=None, prev=None, next=None):
        self.__data = data
        self.__prev = prev
        self.__next = next
    ...
```



# Node Operations: __str__() in Node

- Now override `__str__()` such that the node may return a human readable output format or string format only.




```
class Node:
    def __init__(self, data=None, prev=None, next=None):
        self.__data = data
        self.__prev = prev
        self.__next = next
    ...
    def __str__(self):
        return str(self.__data)
```

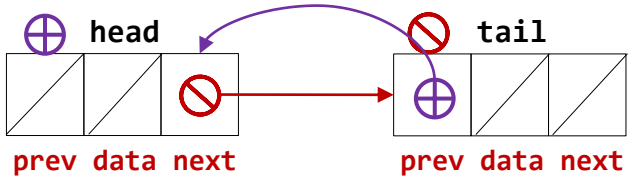
```
alist = DoublyLinked()

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

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

```
None
None
True
```

# Node Operations: debug_headtail() in Node

- For debugging purpose, we may provide a method `debug_headtail()` to produce the references of nodes as shown below




```
class Node:
    ...
    def debug_headtail(self):
        print('     __head:', self.__head)
        print('__head.prev:', self.__head.prev)
        print('__tail.data:', self.__head.data)
        print('__head.next:', self.__head.next)
        print('     __tail:', self.__tail)
        print('__tail.prev:', self.__tail.prev)
        print('__tail.data:', self.__tail.data)
        print('__tail.next:', self.__tail.next)
```

```
alist = DoublyLinked()

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

print(debug_headtail())
```

```
begin: <__main__.Node object at 0x000001BA19CCACD0>
  end: <__main__.Node object at 0x000001BA19CCACD0>
True
     __head: <__main__.Node object at 0x000001BA19CCA160>
__head.prev: None
__tail.data: None
__head.next: <__main__.Node object at 0x000001BA19CCACD0>
     __tail: <__main__.Node object at 0x000001BA19CCACD0>
__tail.prev: <__main__.Node object at 0x000001BA19CCA160>
__tail.data: None
__tail.next: None
```

# Doubly Linked List - Node Class

- Then we may conclude the Node class code as shown below:

```
class DoublyLinked:
    class Node:
        def __init__(self, data = None, prev = None, next = None):
            self.__data = data
            self.__prev = prev
            self.__next = next

        def get_data(self):                 # find() uses
            return self.__data

        def set_data(self, newdata):
            self.__data = newdata

        # this let us access '__data' directly by 'data'
        data = property(get_data, set_data)

        def get_next(self):                 # __str__() uses
            return self.__next

        def set_next(self, newnext):
            self.__next = newnext

        next = property(get_next, set_next)
```

```
        def get_prev(self):                 # insert() uses
            return self.__prev

        def set_prev(self, new_prev):
            self.__prev = new_prev

        # this let us access '__prev' directly by 'prev'
        prev = property(get_prev, set_prev)

        def __str__(self):
            return str(self.__data)

        def debug_headtail(self):
            print('     __head:', self.__head)
            print('__head.prev:', self.__head.prev)
            print('__tail.data:', self.__head.data)
            print('__head.next:', self.__head.next)
            print('     __tail:', self.__tail)
            print('__tail.prev:', self.__tail.prev)
            print('__tail.data:', self.__tail.data)
            print('__tail.next:', self.__tail.next)
```

# Summary

- Doubly Linked List
  - Each node structure has two references which make the list traversal in two ways.
  - Two sentinel nodes in the list helps simplifying the code.
  - Using inner class helps the code maintenance.

# Data Structures in Python Chapter 3

- Linked List
- OOP Inheritance
- ListUnsorted Class
- ListSorted Class
- Iterator
- **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*