Data Structures in Python
Chapter 3

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

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

Agenda

- Linked List
  - Introduction
  - The Node class
  - The Linked List ADT
  - Comparing Implementations

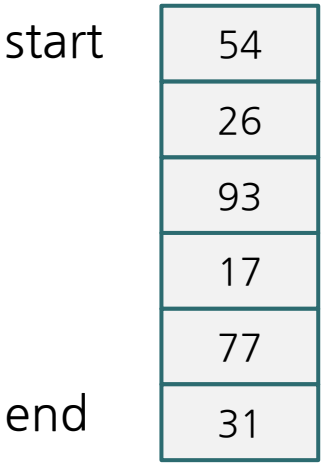
Review

- The `list` in Python is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations.
  - We may use Python `list` to implement both Stack and Queue.

- A Python `list` stores each element in **contiguous memory** if possible.
  - It is an array-based sequence.
  - This makes it possible to access any element in O(1) time.
  - However, insertion or deletion elements at the beginning of the list takes O(n).

Linked List

- An **array** provides the more **centralized** representation, with one large chunk of memory capable of accommodating references to many elements.

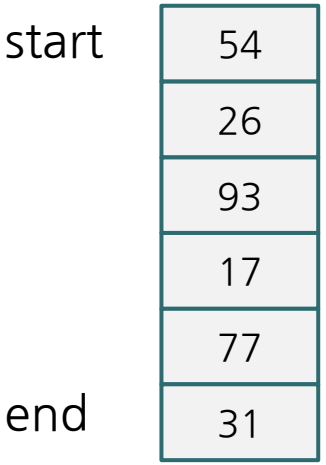


An array-based collection

A Linked List

Linked List

- An **array** provides the more **centralized** representation, with one large chunk of memory capable of accommodating references to many elements.
- A **linked list** relies on a more **distributed** representation in which a lightweight object, known as a **node**, is allocated for each element.
  - Each node maintains a **reference** to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.

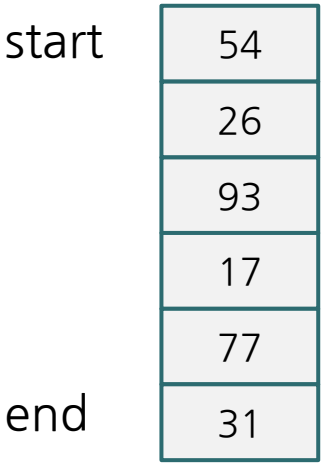


An array-based collection

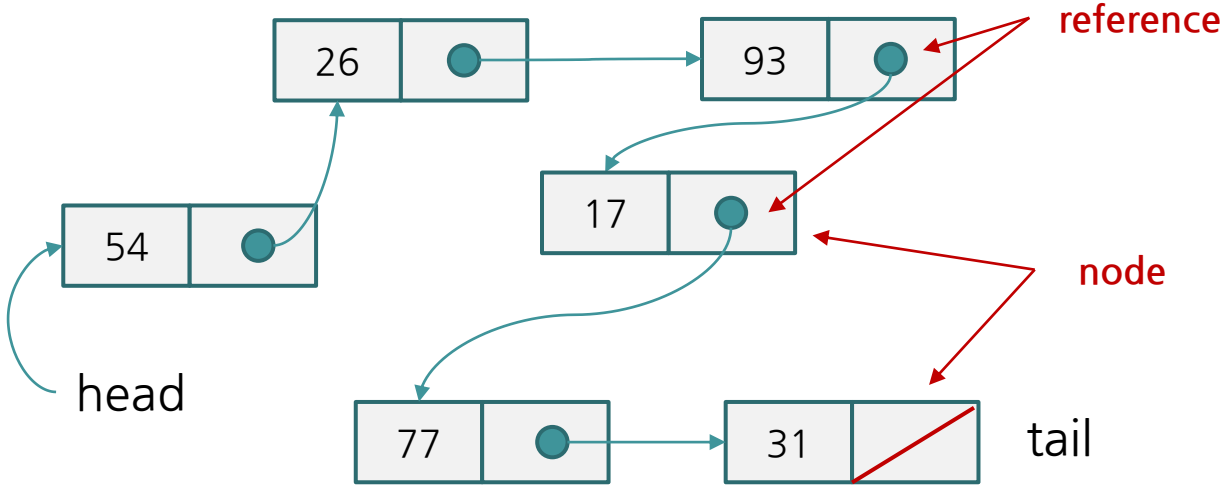
A Linked List

Linked List

- An **array** provides the more **centralized** representation, with one large chunk of memory capable of accommodating references to many elements.
- A **linked list** relies on a more **distributed** representation in which a lightweight object, known as a **node**, is allocated for each element.



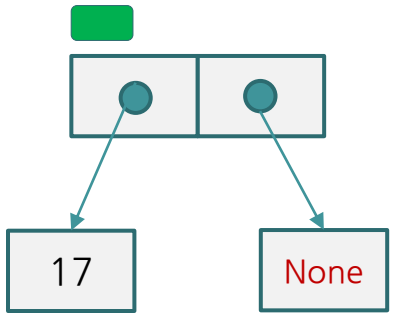
An array-based collection



A Linked List

A Node

- A **node** is the basic building block of a linked list.
- It contains the **data** as well as a **link** to the **next node** in the list.
- The node's element references an arbitrary object that is an element of the sequence (17 in this example), which the next references the subsequent node the linked list or None.



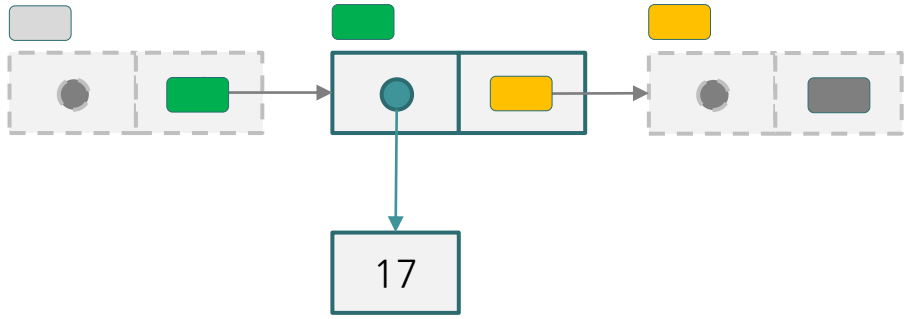
a node in memory

a compact representation of a node

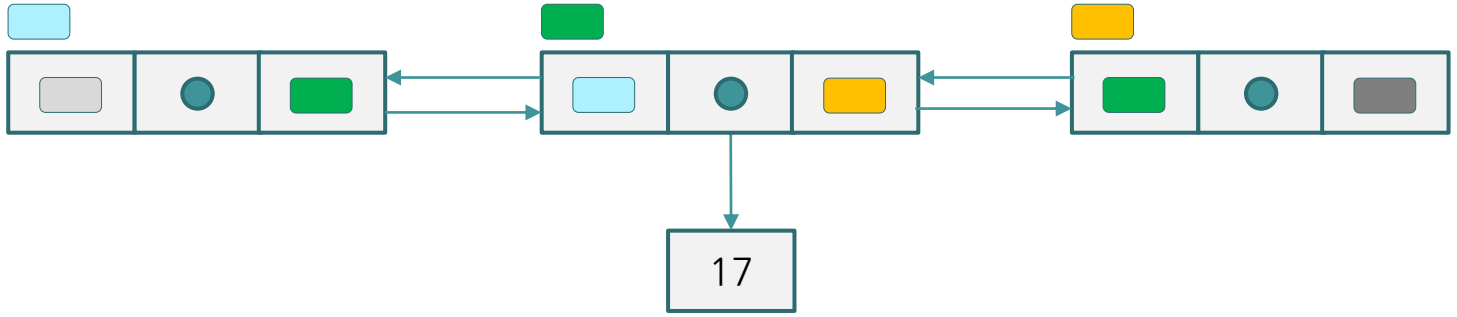
Singly Linked Lists vs Doubly Linked List

- An example of a node instance that forms part of a linked list.
- Each node maintains a reference to its element and **one or more references** to neighboring nodes in order to collectively represent the linear order of the sequence.

A Singly Linked List



A Doubly Linked List

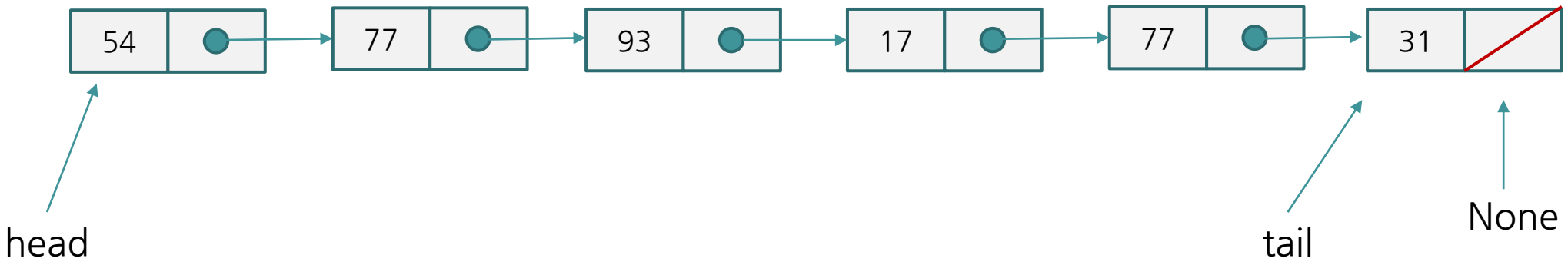


Terminology

- **head and tail:**
  - The first and last node of a linked list are known as the **`head`** and **`tail`** of the list, respectively.
- **traverse**
  - By starting at the head and moving from one node to another by following each node's next reference, we can reach the tail of the list.
  - We can identify the tail as the node having None as its next reference. This process is commonly known as **traversing** the linked list.

Terminology

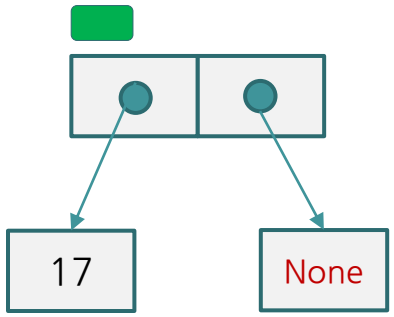
- An example of a singly linked list whose elements are number.
  - The list instance maintains a member named **`head`** that identifies the first node of the list, and another member named **`tail`** that identifies the last node of the list.
  - The **None** object is denoted as a slash.



For a compact illustration of a singly linked list, with elements embedded in the nodes.

The Node class

- A **node** is the basic building block of a linked list.
- It contains the **data** as well as a **link** to the **next node** in the list.
- The node's element references an arbitrary object that is an element of the sequence (17 in this example), which the next references the subsequent node the linked list or None.



a node in memory

a compact representation of a node

The Node class

- A **node** may be defined as shown below:

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

node = Node(17)
```

an implementation of a node

The Node class

- A **Node** class may be defined as shown below:

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

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self, data):
        self._data = data

    def set_next(self, next):
        self._next = next

if __name__ == "__main__":
    node = Node(17)
```

getting the next node linked; it may be None.

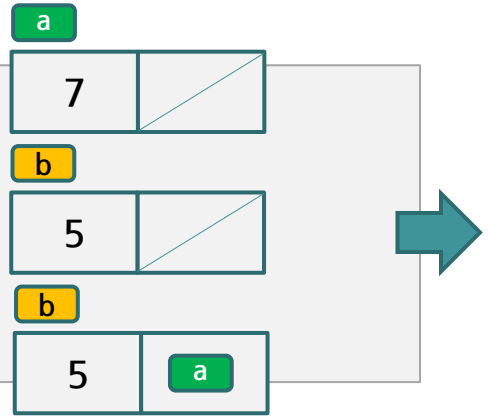
setting the data of the current node

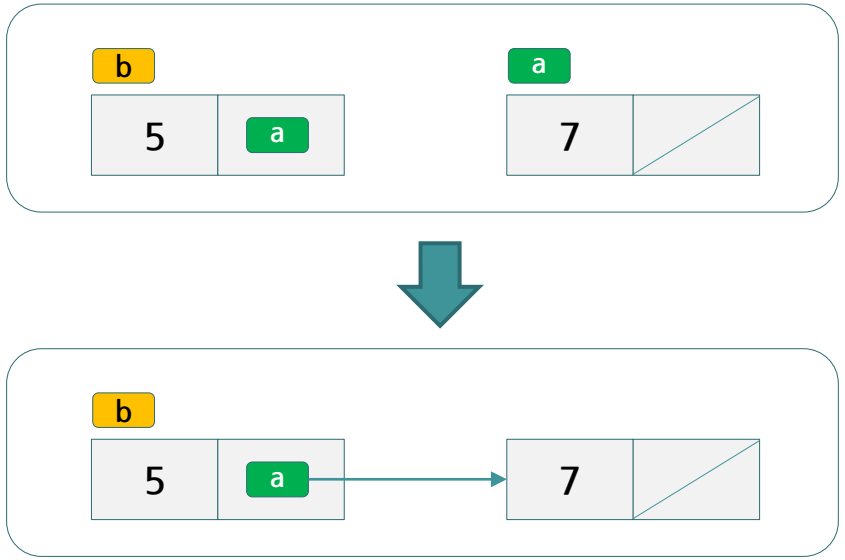
setting the next of the current node
namely, linking the next node

The Node class - Chain of nodes

- Chain of nodes:

```
a = Node(7)

b = Node(5)

b.set_next(a)
```





The node reference **'a'** is stored in **b.next**;
Now, we just keep the node reference of **'b'** which is called the head of the linked list.

The Node class - Chain of nodes

- Change the data of two nodes to 55 and 77 in the linked list, respectively. The head of the list, b is given.

- Step 1:

```
b.set_data(55)
```

- Step 2:

```
x = b.get_next()
x.set_data(77)
```

using a temporary name, x


```
b.get_next().set_data(77)
```

without using a temporary name

Exercise 1

- Step 1: Draw a linked list diagram. Which one is the first node of the list?

```
def print_chain(node):
    while not node == None:
        print(node.get_data(), end = " ")
        node = node.get_next()

a = Node(15)
b = Node(26)
c = Node(37)
d = Node(48)

b.set_next(a)
c.set_next(d)
d.set_next(b)
```

(1) Create nodes.

(2) Link nodes.

(3) Linked list

(1) Create nodes.

(2) Link nodes.

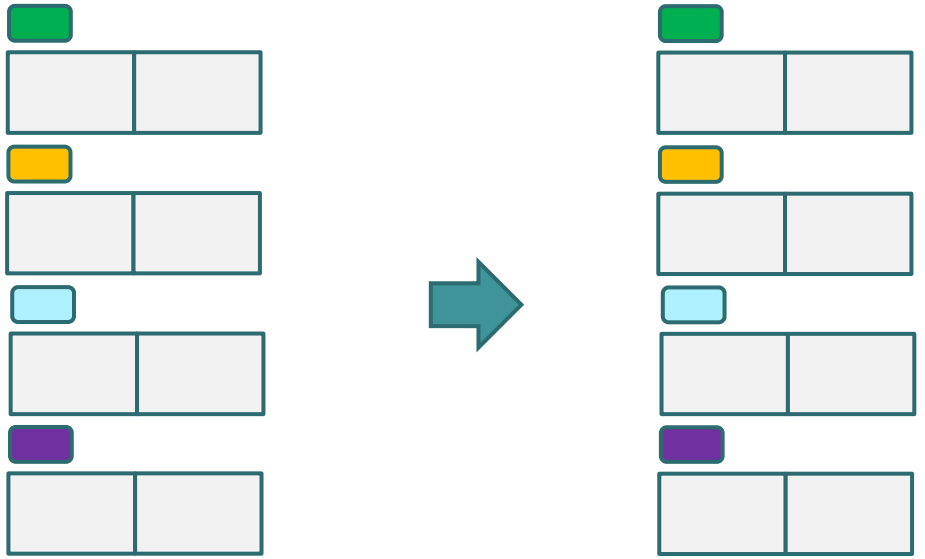
Exercise 1

- Step 1: Draw a linked list diagram. Which one is the first node of the list?

```
def print_chain(node):
    while not node == None:
        print(node.get_data(), end = " ")
        node = node.get_next()

a = Node(15)
b = Node(26)
c = Node(37)
d = Node(48)

b.set_next(a)
c.set_next(d)
d.set_next(b)
```

(1) Create nodes.

(2) Link nodes.



(1) Create nodes.

(2) Link nodes.

(3) Linked list

Exercise 1

- Step 2: What is the output of the following program?

```
def print_chain(node):
    while not node == None:
        print(node.get_data(), end = " ")
        node = node.get_next()

a = Node(15)
b = Node(26)
c = Node(37)
d = Node(48)

b.set_next(a)
c.set_next(d)
d.set_next(b)
```

```
print_chain(a)
print()
print_chain(b)
print()
print_chain(c)
```

(3) Linked list

Exercise 1 Observation

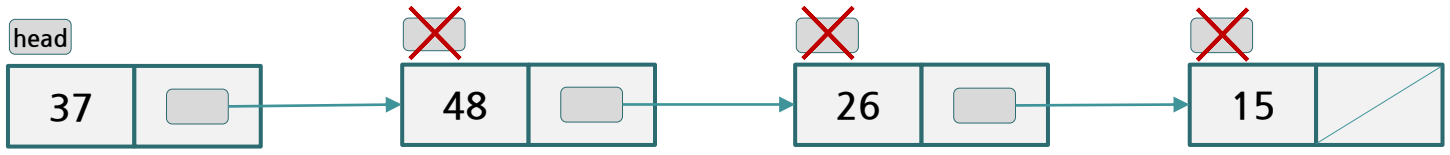
- Step 2: What is the output of the following program?

```
def print_chain(node):
    while not node == None:
        print(node.get_data(), end = " ")
        node = node.get_next()

a = Node(15)
b = Node(26)
c = Node(37)
d = Node(48)

b.set_next(a)
c.set_next(d)
d.set_next(b)
```

```
print_chain(a)
print()
print_chain(b)
print()
print_chain(c)
```

- Notice that only one reference is passed to the function, and others are unknown in the function.
- You may `traverse` **the whole list** if the first node reference or the `head` is known.

(3) Linked list



Linked List ADT

- `LinkedList()`
  - Creates a new list that is empty and returns an empty list.
- `is_empty()`
  - Tests to see whether the list is empty and **returns** a Boolean value.
- `size()` and `__len__()`
  - Returns the number of nodes in the list.
- `__str__()`
  - Returns contents of the list in human readable format.
- `push(data), push_back(data)`
  - Pushes a new node with data to the list.
- `pop_front(), pop(data)`
  - Removes the node with data from the list.
- `find(data)`
  - Finds for the data in the list and **returns** a Boolean value.

Summary

- Reference variables can be used to implement the data structure known as **a linked list.**
- Each reference, "next", in a linked list is a reference to the next node in the list.
- Any element in a list can be accessed, however, you must traverse a linked list to access a particular node using the **head** node available.