Data Structures & Algorithms in Python

Lecture 02 – Linked List

Dr. Owen Noel Newton Fernando

College of Engineering

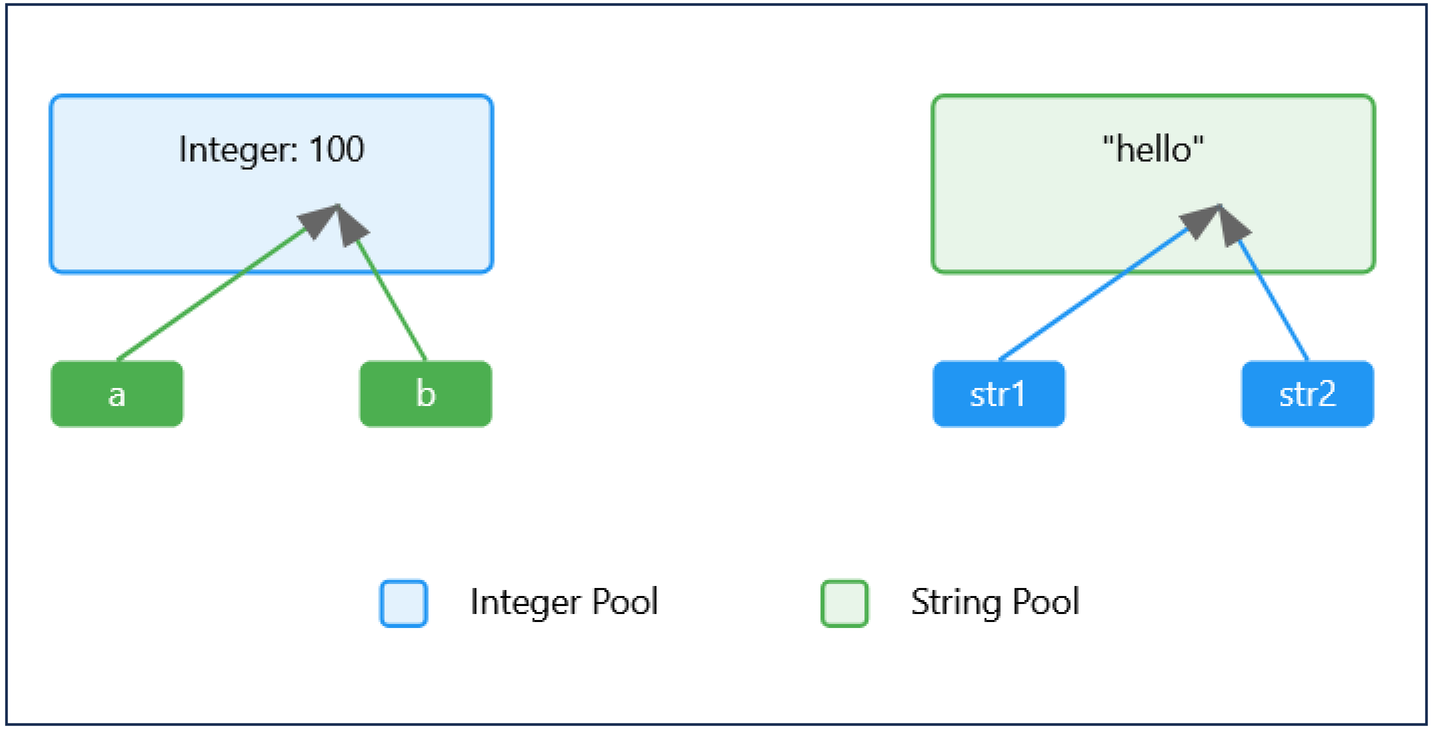
School of Computer Science and Engineering

Memory Pooling

- **Reduces Overhead**: Pre-allocating memory reduces the need for frequent memory allocation and deallocation.
- **Minimizes Fragmentation**: Reuses memory blocks efficiently, reducing gaps in memory.
- **Improves Performance**: Faster memory allocation due to pre-allocation and reuse.
- **Optimized for Small Objects**: Especially beneficial for small integers, strings, and frequently used types.

Memory Pooling: Example

```python
# Memory Pooling Examples
# Integers (small numbers)
a = 100
b = 100
print("Same integers:")
print(a is b)  # True

# Short strings
str1 = "hello"
str2 = "hello"
print("\nShort strings:")
print(str1 is str2)  # True
```

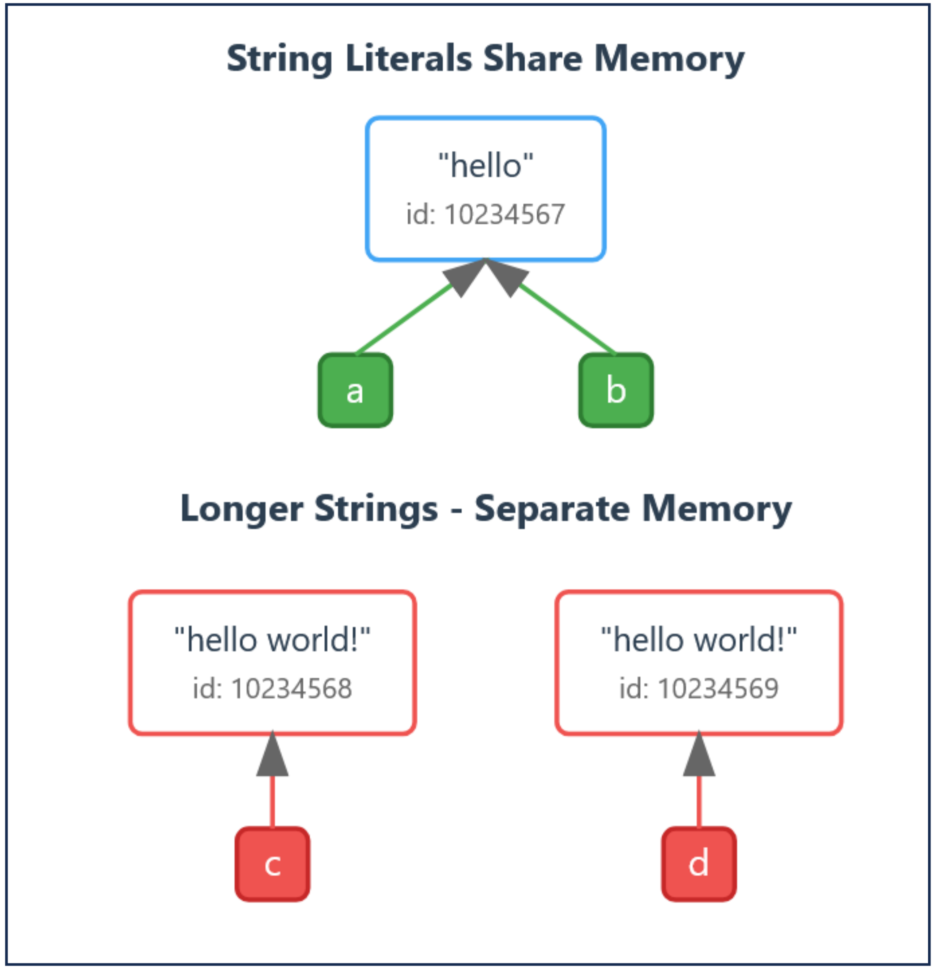


Memory Interning

- **Optimization Technique**: Python uses memory interning to enhance performance by reusing objects instead of creating new ones for frequently used immutable values.
- **Immutable Objects**: Strings and small integers are examples of immutable objects that are commonly interned automatically.
- **Memory Sharing**: Interned objects with the same value share the same memory address. This reduces memory usage and prevents unnecessary duplication.
- **Avoids Duplication**: By interning objects, Python avoids creating multiple copies of the same immutable value, which significantly reduces memory overhead.
- **Enhances Comparisons**: Interned objects improve performance during equality and identity comparisons since their references can be directly compared instead of their values.

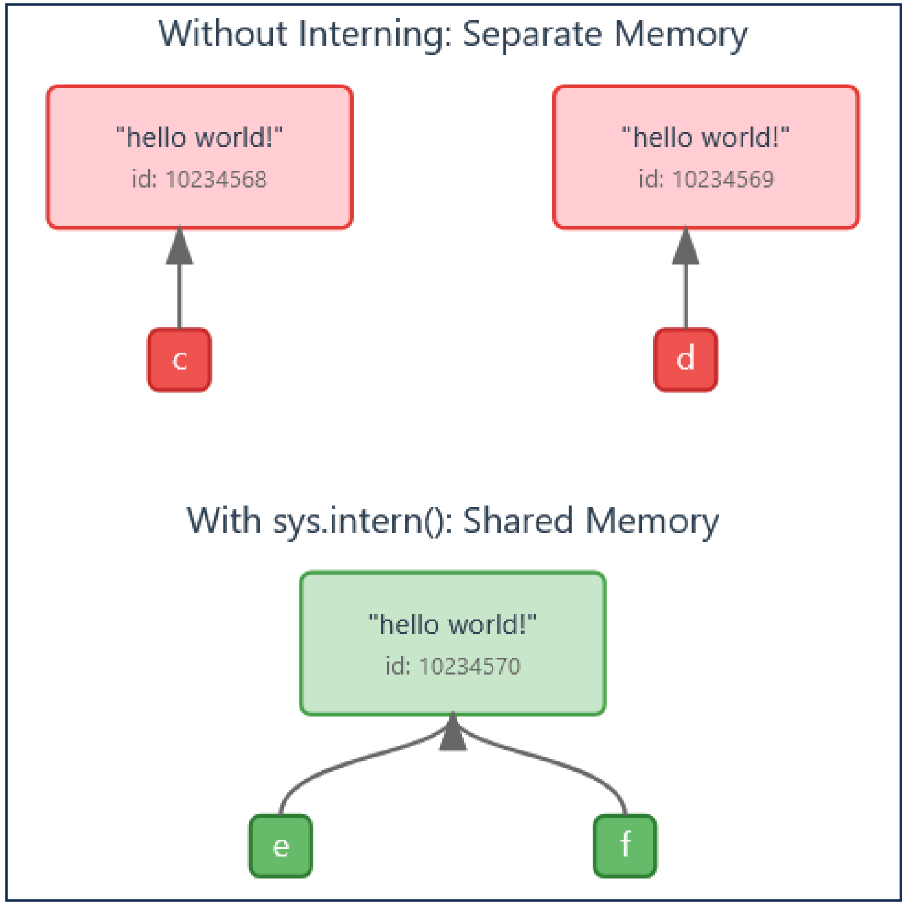
Memory Interning: : Example

```python
# String literals share memory location
a = "hello"
b = "hello"

print(id(a)) # Memory location for 'a' (e.g., 140712834927872)
print(id(b)) # Memory location for 'b' (same as 'a’)
print(a is b) # True - same memory location

# Longer strings - separate memory locations
c = "hello world!"
d = "hello world!"

print(id(c)) # Memory location for 'c' (e.g., 140712834928192)
print(id(d)) # Memory location for 'd' (different from 'c’)
print(c is d) # False - different memory locations
```



Memory Interning: : Manual Interning Example

```python
import sys
# Longer strings - separate memory locations
c = "hello world!"
d = "hello world!"

print(id(c)) # Memory location for 'c' (e.g., 140712834928192)
print(id(d)) # Memory location for 'd' (different from 'c’)
print(c is d) # False - different memory locations

# Manual interning for large strings
e = sys.intern("hello world!")
f = sys.intern("hello world!")
print(id(e))  # Memory location for 'e'
print(id(f))  # Memory location for 'f'
print(e is f)  # True - same memory location due to manual interning
```

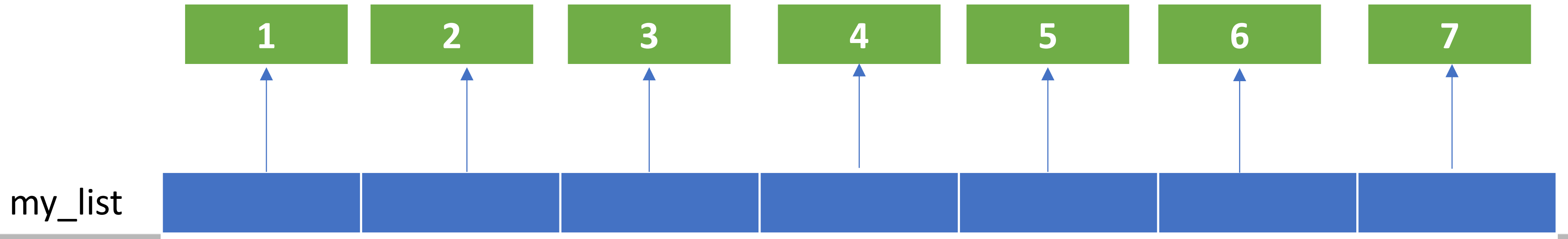


Key Differences: Memory Pooling and Memory Interning

Aspect	Memory Pooling	Memory Interning
Purpose	Improves efficiency by reusing preallocated memory blocks.	Reduces duplication by storing one shared copy of immutable objects.
Scope	Applies to small, frequently created objects (e.g., integers, short strings), by reusing preallocated memory blocks.	Applies specifically to immutable objects (e.g., strings, small integers), by storing one shared copy.
Automatic or Manual	Automatic for small objects (e.g., integers).	Automatic for short strings; manual interning possible for long strings.
Implementation	Reuses preallocated memory blocks	Reuses one shared copy of immutable values.
Example Use Case	Used for small integers and short strings.	Used for frequently repeated strings and small integers.

Referential Arrays

- A referential array typically refers to an array where each element is a reference (or pointer) to another object or data structure, rather than a direct value.
- In Python, everything is an object.
  - Its variables hold references to objects
    - The reference or pointer is the memory address where the data is located



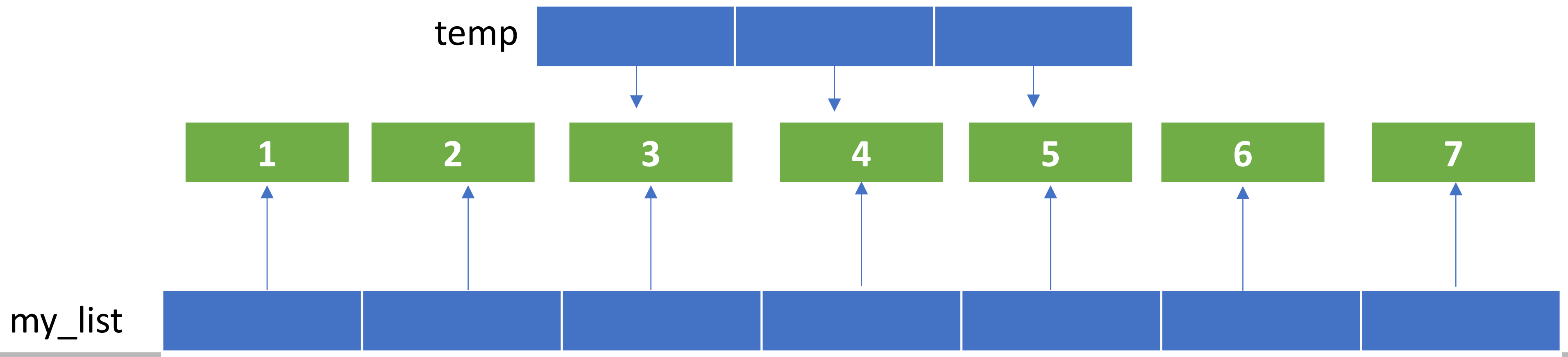
Referential Arrays

- create a list

```
>>my_list = [1, 2, 3, 4, 5, 6, 7]
```

- The fact in python is that each element is stored at different memory location

```
>>temp = my_list[2:5]
```

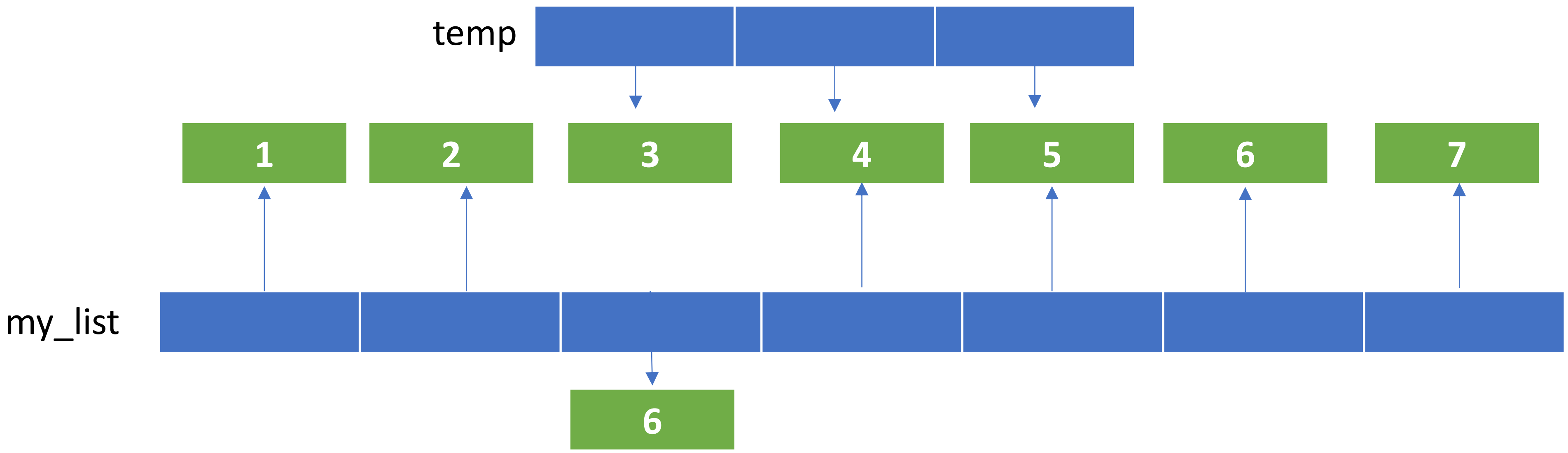


Referential Arrays

- create a list

```
>>my_list[2] = 6
```

- id() return the memory address where the object is stored
- array module can more compactly represent an array
  - import array (https://docs.python.org/3/library/array.html)



Create A Customized Data Type Class

- Limited built-in data types in Python (any programming languages)
  - bool, int, float, complex (3+4j)..
- In practice, the data can be more complex
  - Student Information
    - Name – string
    - Age – int
    - CGPA – float (2 decimal places)
    - Etc.

```
class Student:
    def __init__(self, name, age, cgpa):
        self.name = name
        self.age = age
        self.cgpa = cgpa
```

- Benefits of Custom Data Types
  - **Encapsulation**: Bundle data and methods within a single unit.
  - **Reusability**: Create reusable code components.
  - **Abstraction**: Hide implementation details and expose only necessary operations.

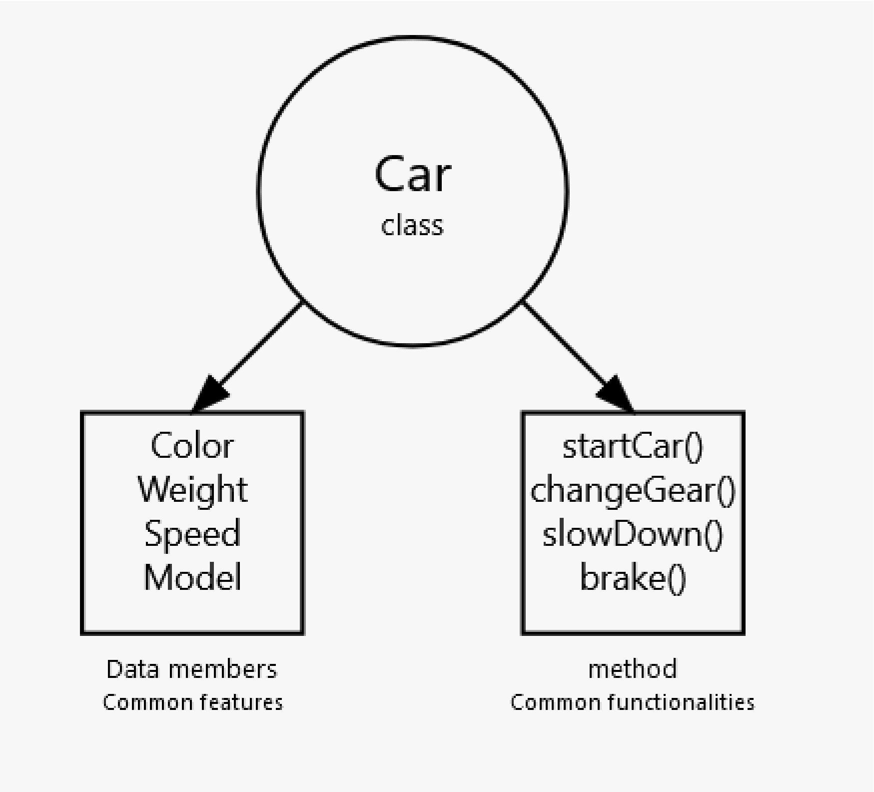
What is OOP?

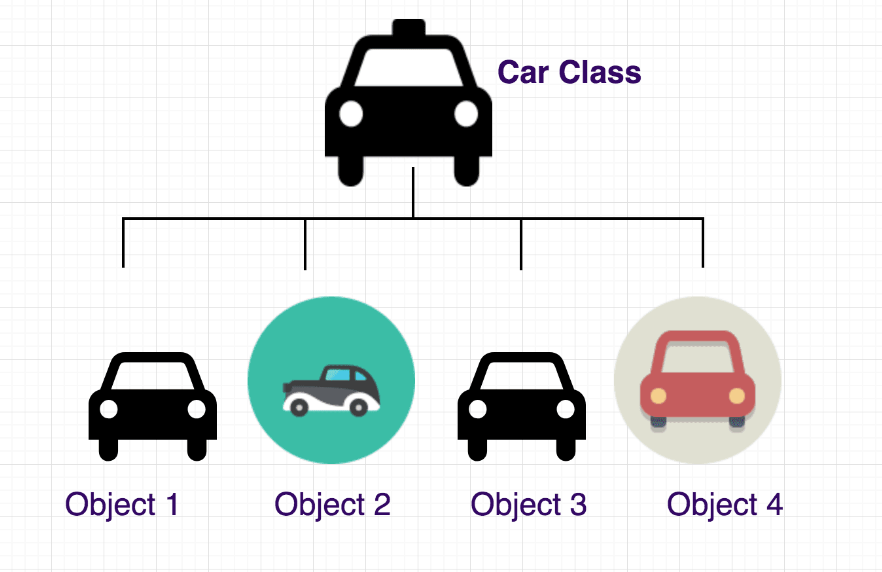
A programming paradigm based on the concept of "**objects**" that encapsulate data and behavior.

OOPs is a way of organizing code that uses **objects** and **classes** to represent real-world entities and their behavior

In OOPs, object has attributes that have specific data and can perform certain actions using methods.

Object-Oriented Programming





Objects

- In Python, **everything**—numbers, strings, lists, dictionaries, etc.—is treated as an **object**.
- Objects represent **data** and provide mechanisms to interact with that data using **methods** and **attributes**.
- **Examples of Objects**
  - **Numbers**
    - 1234: An integer (int) object.
    - 3.14159: A floating-point number (float) object
  - **String**:
    - "Hello": A string (str) object that contains text.
  - **List**:
    - [1, 5, 7, 11, 13]: A list (list) object containing multiple integers.
  - **Dictionary**:
    - {“NTU": “Nanyang Technological University", “SG": “Singapore"}: A dictionary (dict) object mapping keys to values.

Objects

Every Object Has:

1. **A Type:**
    - Defines the **kind of data** the object represents (e.g., int, float, str, list, dict).

2. **An Internal Data Representation:**
    - **Primitive Representation**: Simple objects like int or float store data in a compact, raw form.
    - **Composite Representation**: Complex objects like list or dict combine multiple elements.

3. **Procedures for Interaction:**
    - **Objects expose methods** (functions associated with the object) for interaction.
    - Example: A string object ("Hello") has methods like .upper() or .lower().

An Object is an Instance of a Type

1. When you create an object, it is a **specific realization (instance)** of a broader **type (class)**.
    - Example:
        - 1234 is an **instance of the int type**.
        - "hello" is an **instance of the str type**.

Objects and Classes

Objects:

- Everything in Python is an **object**.
- Objects can be **created**, **manipulated**, and **destroyed** (e.g., using del or garbage collection).

Classes:

- Classes are **blueprints** for creating objects.
- A class defines **attributes (data)** and **methods (functions).**
- Example: A Car class defines attributes (colour, speed) and methods (drive, brake). **Each car is an instance of the class**.

Creating and using your own types with classes

Creating a Class

- A class is a blueprint for creating objects.
- Steps Involved:
  - Define the class name (**Student**).
  - Specify the **attributes** (data) and **methods** (behaviour).

```
class Student:
    institution_name = “NTU"

    def __init__(self, name, age, cgpa):
        self.name = name
        self.age = age
        self.cgpa = cgpa
```

- **`institution_name`** : A class attribute shared by all instances of the class.
- The **`__init__`** method initializes the object with specific attributes.
- The Student class defines attributes like `name, age, and cgpa.`

Creating an instance of the class

Using a Class

Creating instances (objects) from a class and performing operations.

Example:

```
# Creating instances
student1 = Student(“Newton", 20, 3.9)
student2 = Student(“Fernando", 22, 3.7)

# Accessing attributes
print(student1.name)    # Output: Newton
print(student2.cgpa)    # Output: 3.7
```

When you create **student1**:

- `name` = "Newton", `age` = 20, and `cgpa` = 3.9 are passed to the `__init__` method.
- The instance student1 is initialized with `name = "Newton"`, `age = 20`, and `cgpa = 3.9`.
- `institution_name` remains shared from the class.

- student1 and student2 are instances of the Student class.
- Attributes like name, age, and cgpa can be accessed directly.

Accessing Attributes

After you have created student1, student2 objects ：

```
# Creating instances
student1 = Student("Newton", 20, 3.9)
student2 = Student("Fernando", 22, 3.7)

# Accessing instance attributes
print(student1.name)    # Output: Newton
print(student1.age)     # Output: 20
print(student1.cgpa)    # Output: 3.9

print(student2.name)    # Output: Fernando
print(student2.age)     # Output: 22
print(student2.cgpa)    # Output: 3.7

# Accessing the shared class attribute
print(student1.institution_name) # Output: NTU
print(student2.institution_name)   # Output: NTU

# Accessing class attribute directly from the class
print(Student.institution_name)    # Output: NTU
```

Methods in classes

What is a method?

A method is a procedural attribute, like a **function that works only with this class**.

Python always passes the **object** as the first argument. The convention is to use **self** as the name of the first argument of all **methods**.

We will include methods to display the student's name, retrieve their age, and access their CGPA in our Student class.

Methods in classes

```
class Student:
    institution_name = "NTU"  # Class attribute

    def __init__(self, name, age, cgpa):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute
        self.cgpa = cgpa  # Instance attribute

    def get_name(self):       # Getter method for name
        return self.name

    def get_age(self):        # Getter method for age
        return self.age

    def get_cgpa(self):       # Getter method for CGPA
        return self.cgpa
```

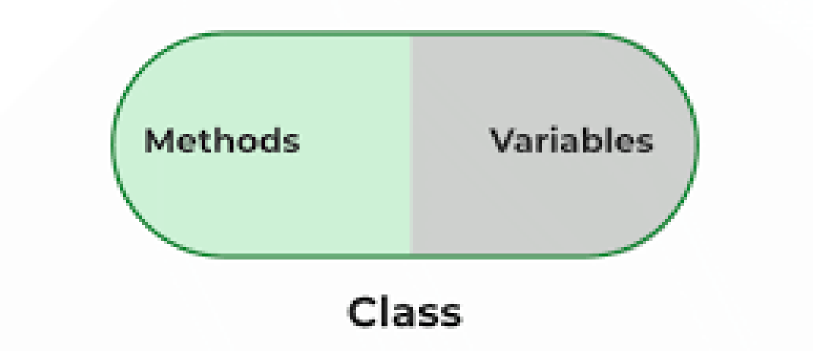
Methods in classes

```
class Student:
    institution_name = "NTU"  # Class attribute

    def __init__(self, name, age, cgpa):
        self.__name = name  # Private instance attribute
        self.__age = age    # Private instance attribute
        self.__cgpa = cgpa  # Private instance attribute

    def get_name(self):       # Getter method for name
        return self.__name

    def get_age(self):        # Getter method for age
        return self.__age

    def get_cgpa(self):       # Getter method for CGPA
        return self.__cgpa
```

Encapsulation

Encapsulation is the bundling of **data (attributes)** and **methods (functions)** within a class, restricting access to some components to control interactions.

Types of Encapsulation:



Public Members: Accessible from anywhere.

Protected Members: Accessible within the class and its subclasses.

Private Members: Accessible only within the class.

OOP Concept : Encapsulation

Encapsulation

Python achieves encapsulation by:

Using a single underscore _attribute to indicate a **protected attribute** (convention, not enforced).

Using a double underscore __attribute to make the attribute **private** (name mangling to prevent direct access)

and

Providing controlled access via methods (getters and setters).

Public Attributes

- self.name, self.age, and self.cgpa are public attributes because they are not prefixed with an underscore (_ or __).
- Public attributes can be accessed, modified, or used from anywhere, both inside and outside the class.

```
class Student:
    institution_name = "NTU"  # Class attribute

    def __init__(self, name, age, cgpa):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute
        self.cgpa = cgpa  # Instance attribute

    def get_name(self):         # Getter method for name
        return self.name

    def get_age(self):          # Getter method for age
        return self.age

    def get_cgpa(self):         # Getter method for CGPA
        return self.cgpa

    def display_details(self):  # Method to display student details
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"CGPA: {self.cgpa}")
        print(f"Institution: {Student.institution_name}")
```

Encapsulation

Python achieves encapsulation by:

Using a single underscore _attribute to indicate a **protected attribute** (convention, not enforced).

Using a double underscore __attribute to make the attribute **private** (name mangling to prevent direct access)

and

Providing controlled access via methods (getters and setters).

Encapsulation: Protected Attribute

- Attributes like _name, _age, and _cgpa are protected, following a convention of a single underscore prefix.
- Protected attributes are not enforced as private by Python but indicate to developers that they should not be accessed directly outside the class or its subclasses.

```
class Student:
    institution_name = "NTU"  # Class attribute

    def __init__(self, name, age, cgpa):
        self._name = name   # Protected instance attribute
        self._age = age     # Protected instance attribute
        self._cgpa = cgpa   # Protected instance attribute

    def get_name(self):        # Getter method for name
        return self._name

    def get_age(self):         # Getter method for age
        return self._age

    def get_cgpa(self):        # Getter method for CGPA
        return self._cgpa

    def display_details(self):  # Method to display student details
        print(f"Name: {self.get_name()}")
        print(f"Age: {self.get_age()}")
        print(f"CGPA: {self.get_cgpa()}")
        print(f"Institution: {Student.institution_name}")
```

Encapsulation

Python achieves encapsulation by:

Using a single underscore _attribute to indicate a **protected attribute** (convention, not enforced).

Using a double underscore __attribute to make the attribute **private** (name mangling to prevent direct access)

and

Providing controlled access via methods (getters and setters).

Encapsulation: Private Attribute

If the attributes are private, they must be accessed using getter methods.

```
class Student:
    institution_name = "NTU"  # Class attribute

    def __init__(self, name, age, cgpa):
        self.__name = name    # Private instance attribute
        self.__age = age      # Private instance attribute
        self.__cgpa = cgpa    # Private instance attribute

    def get_name(self):       # Getter method for name
        return self.__name

    def get_age(self):        # Getter method for age
        return self.__age

    def get_cgpa(self):       # Getter method for CGPA
        return self.__cgpa

    def display_details(self):  # Method to display all student details
        print(f"Name: {self.get_name()}")
        print(f"Age: {self.get_age()}")
        print(f"CGPA: {self.get_cgpa()}")
        print(f"Institution: {Student.institution_name}")
```

Create A Customized Data Type Class

- Limited built-in data types in Python (any programming languages)
  – bool, int, float, complex (3+4j)..
- In practice, the data can be more complex
  – Student Information
    - Name – string
    - Age – int
    - CGPA – float (2 decimal places)
    - Etc.

```
class Student:
    def __init__(self, name, age, cgpa):
        self.name = name
        self.age = age
        self.cgpa = cgpa
```

- Benefits of Custom Data Types
  – **Encapsulation**: Bundle data and methods within a single unit.
  – **Reusability**: Create reusable code components.
  – **Abstraction**: Hide implementation details and expose only necessary operations.

Example: Encapsulation

- Encapsulation **restricts direct access to a class's internal data** by hiding attributes and implementations. Interaction is managed through **public methods** to safeguard data integrity.
- Common Operations in **Encapsulation**
  - **Set**: Use methods to update attributes instead of direct modification.
  - **Print/Display/Get**: Retrieve the value of an attribute.
  - **Insertion**: Add a new object.
  - **Deletion**: Remove an object.
  - **Size**: Return the count of objects.

Create A Customized Data Type Class

```
class Student:
    def __init__(self, name, age, cgpa):
        self.__name = name
        self.__age = age
        self.__cgpa = cgpa

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def get_cgpa(self):
        return self.__cgpa
```

```
class StudentList:
    def __init__(self):
        self.__SL = []

    def insert(self, name, age, cgpa):
        self.__SL.append(Student(name,age,cgpa))

    def delete(self):
        self.__SL.pop()

    def print(self):
        for x in self.__SL:
            print('name: ',x.get_name(),'\t age:
            ',x.get_age(),'\t cgpa: ',x.get_cgpa())

sl = StudentList()
sl.insert(‘Newton1',10,4.0)
sl.insert(‘Newton2',30,4.11)
sl.insert(‘Newton3',20,4.22)
sl.print()
sl.delete()
sl.print()
```

Private Access

Data Protection:

To ensure that critical data is not **accidentally modified by external code**.

Encapsulation:

To hide the internal representation of the data and expose only what is necessary through methods (getters and setters).

Immutability:

To create immutable objects where certain properties cannot be changed after the object is created.

In the example, we are not able to make any change once the Student data is created. No **setter methods** in the class Student and all attributes are private.

Heterogeneous Collection

List: This is the most commonly used data structure in Python.

- To handle heterogeneous collection, we define a new object class (eg. Student)

- `my_list = [1, 'hello', 3.14, True, [1, 2, 3]]`
  - You can do so but it is not recommended
    - **hard to read and manage**

Alternate solutions:

Dictionary: This is a collection of **key-value pairs**. It is unordered and mutable, and it allows for fast lookups, additions, and deletions of elements based on keys.

```
my_dict = {'integer': 1, 'string': 'hello', 'float': 3.14,
'boolean': True, 'list': [1, 2, 3] }
```

Any other solution?

Linked List

Linked List

A **LinkedList** consists of **nodes** where each node has data and a pointer to the next node, ending with None. It's represented as sequential nodes connected by pointers in memory.

Key Operations:

– **Insert**: Add elements at start or **anywhere**

– **Delete**: Remove elements from **any position**

– **Search**: Find elements by **traversing through nodes**

Linked List

A LinkedList consists of nodes where each node has data and a pointer to the next node, ending with None. It's represented as sequential nodes connected by pointers in memory.

Key Operations:

– **Insert**: Add elements at start or anywhere

– **Delete**: Remove elements from any position

– **Search**: Find elements by traversing through nodes

Types

– **Singly Linked List** (one direction).

– **Doubly Linked List** (both directions).

– **Circular Linked List** (last node points to the first)

Linked List

A LinkedList consists of nodes where each node has data and a pointer to the next node, ending with None. It's represented as sequential nodes connected by pointers in memory.

Key Operations:
- **Insert**: Add elements at start or anywhere
- **Delete**: Remove elements from any position
- **Search**: Find elements by traversing through nodes

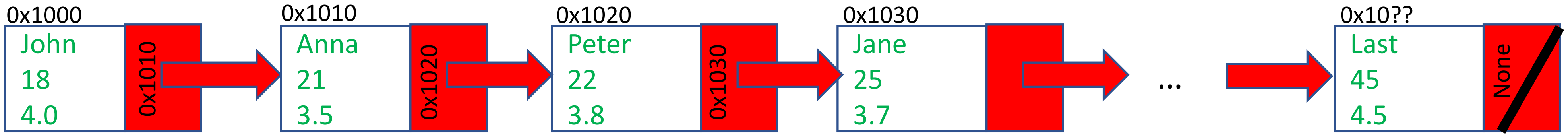
Types
- Singly Linked List (one direction).
- Doubly Linked List (both directions).
- Circular Linked List (last node points to the first)

Benefits:
Dynamic memory allocation, efficient front insertions, flexible size management without wastage

Use Cases
Implementing stacks, queues, music playlists, browser history navigation

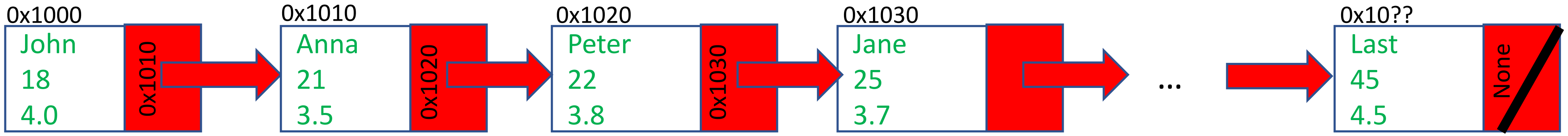
Singly Linked Lists

- A collection of nodes that collectively form a linear sequence.
- Each node stores
  - a reference to an **object** that is an element of the sequence
  - a **reference** to the **next node**



Singly Linked Lists

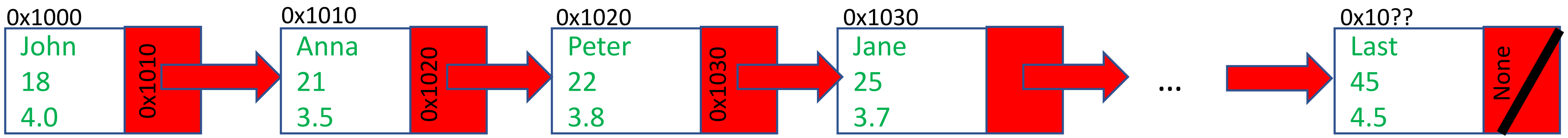
- A collection of nodes that collectively form a linear sequence.
- Each node stores
  - a reference to an object that is an element of the sequence
  - a reference to the next node



- In the Singly Linked List, there is only **ONE link** in each node.
  - Only the **previous** node can allocate the node

Singly Linked Lists

- A collection of nodes that collectively form a linear sequence.
- Each node stores
  - a reference to an object that is an element of the sequence
  - a reference to the next node



- In the Singly Linked List, there is only ONE link in each node.
  - Only the previous node can allocate the node
- Linked lists give more flexibility to manage the data.
- Constructing a linked list is more tedious than the list or dictionary.

Implementation of a linked list in Python

- A node class is required to store each **data** and the **link** to the next node

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

Implementation of a linked list in Python

- A node class is required to store each data and the link to the next node

```
class Node:
  def __init__(self, data, next):
    self.data = data
    self.next = next
```

- `def __init__(self, data, next):` This is the **constructor method** that **initializes** a **new Node object**:
  - `self` refers to the **instance being created** (Refers to current node instance)
  - `data` is the parameter that will store the **node's value**
  - `self.next`: Reference to the **next node** (default None, updated to link nodes).

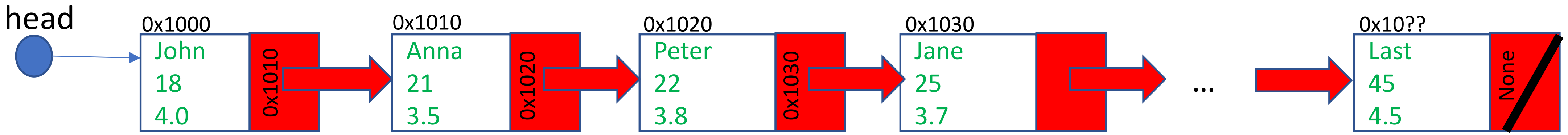
Implementation of a linked list in Python

- A node class is required to store each data and the link to the next node

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

- `def __init__(self, data, next):` This is the constructor method that initializes a new Node object:
  - `self` refers to the instance being created (Refers to current node instance)
  - `data` is the parameter that will store the node's value
  - `self.next`: Reference to the next node (default None, updated to link nodes).
- Node Components:
  - `self.data = data` → Stores the value in node
  - `self.next = next` → Reference to the next node (empty at start)

Implementation of a linked list in Python



- A linked list class is used to create and manage a list of nodes.
- The **head reference** is essential to locate the first node in the list.
- Additional **reference** like a tail node can improve efficiency by pointing to the last node.
- The class supports operations such as insertion, deletion, and traversal to manage the list effectively.

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

Abstract Data Type (ADT)

ADT is a high-level abstraction for a data type, focusing on the operations that can be performed on it while hiding the implementation details.

Specifies:

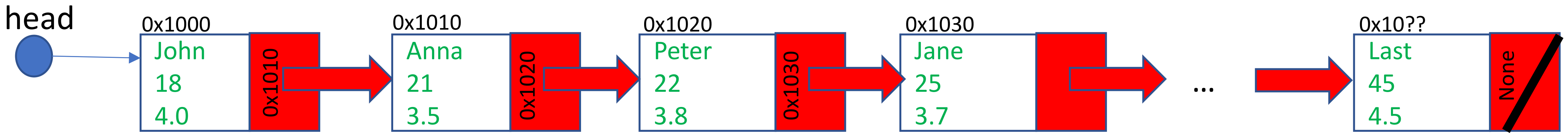
– **Data Type**: The kind of data the ADT can store.

– **Methods**: The operations that can be performed on the data (e.g., insertion, deletion, traversal).

Key Characteristics:

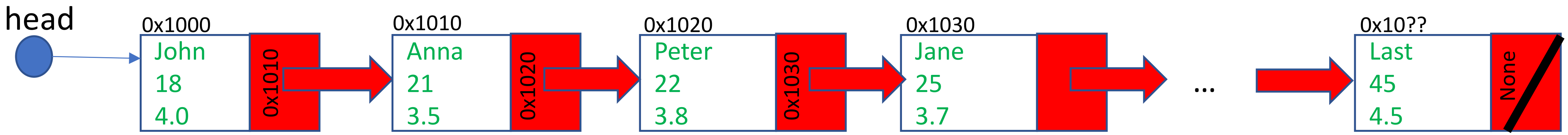
1. Focuses on **behaviour** rather than implementation.
2. **Encapsulates** the data and its operations.
3. **Hides internal workings** and implementation from the user.

Implementation of a linked list in Python



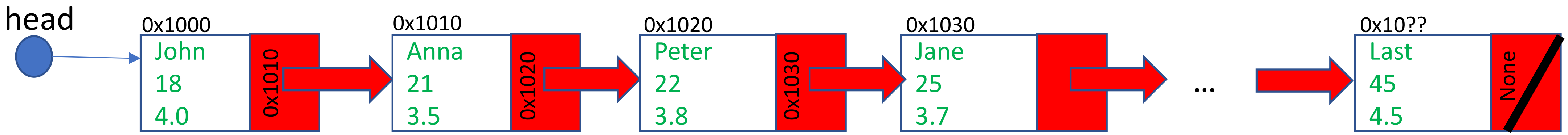
- To design the ADT of a simple Singly Linked List:
  - **Display** each element in the linked list
  - **Search** a node
  - **Add** a new node
  - **Remove** a node
  - **Size** of the linked list

Implementation of a linked list in Python



- **Display** each element in the linked list
  - Given the **head reference** of the linked list
  - Print all items in the linked list
  - From first node to the last node

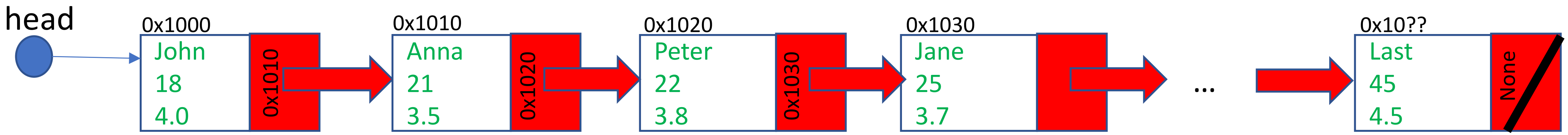
Display each element in the linked list



- Display each element in the linked list

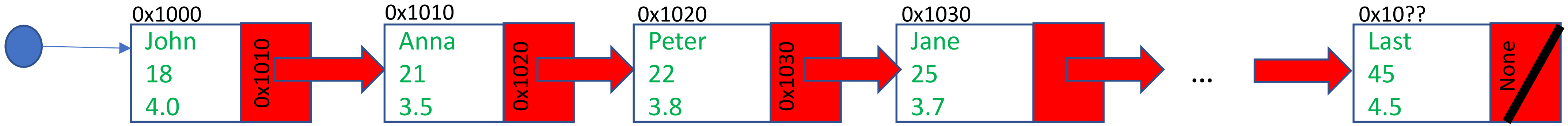
```
def display(self):
    current = self.head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")
```

Search the node at index i



- Looking for the ith node in the list
  - Given the **head reference** of the linked list and **index i**
  - Return the pointer to the **ith** node
  - **None** will be return if index **i** is out of the range or the linked list is empty

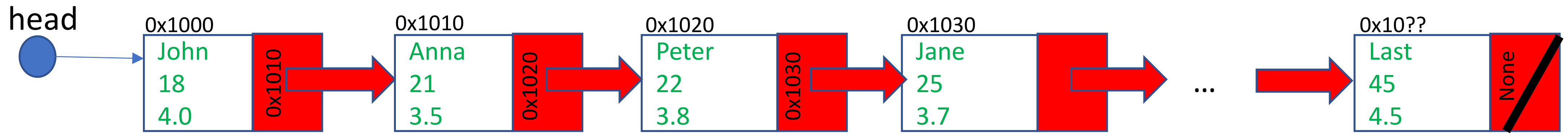
Search the node at index i




```python
def findAt(self, index):
        current = self.head
        if not current:
            return None
        while index>0:
            current = current.next
            if not current:
                return None
            index-=1
        return current
```

- Start from **head** (the reference to the first node).
- If the list is **empty → return None**.
- Traverse node by node, **decrementing index**.
- If you reach the end (None) **before finishing → return None.**
- When **index reaches 0** → return the reference to the current node.

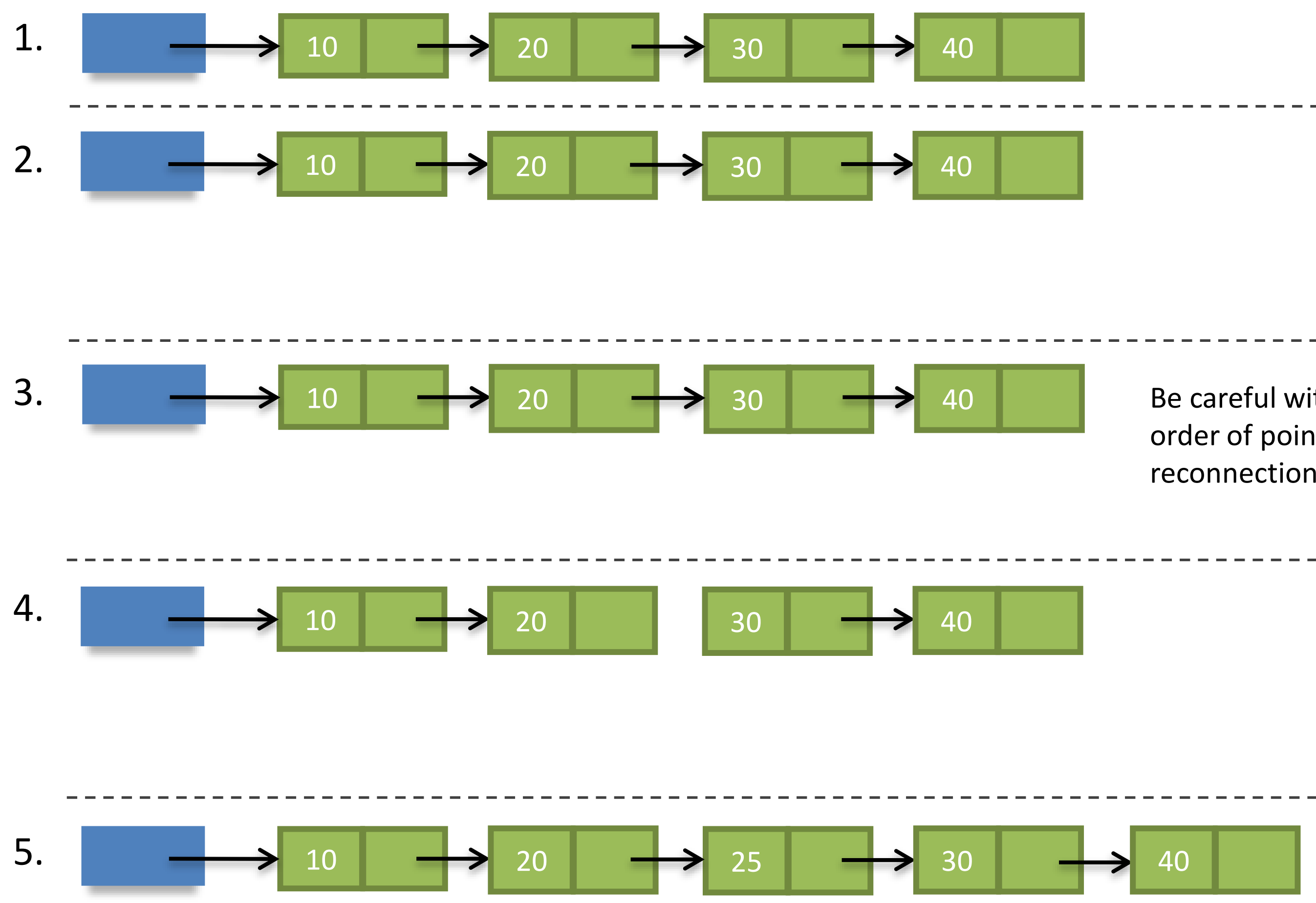
Add a node



- **Add a node in the linked list**
- **Given**
  - the pointer to the **head reference** of a linked list
  - **index i** where the node to be inserted
  - the ***item*** for the node
- *Return **SUCCESS** (1) or **FAILURE** (0)*

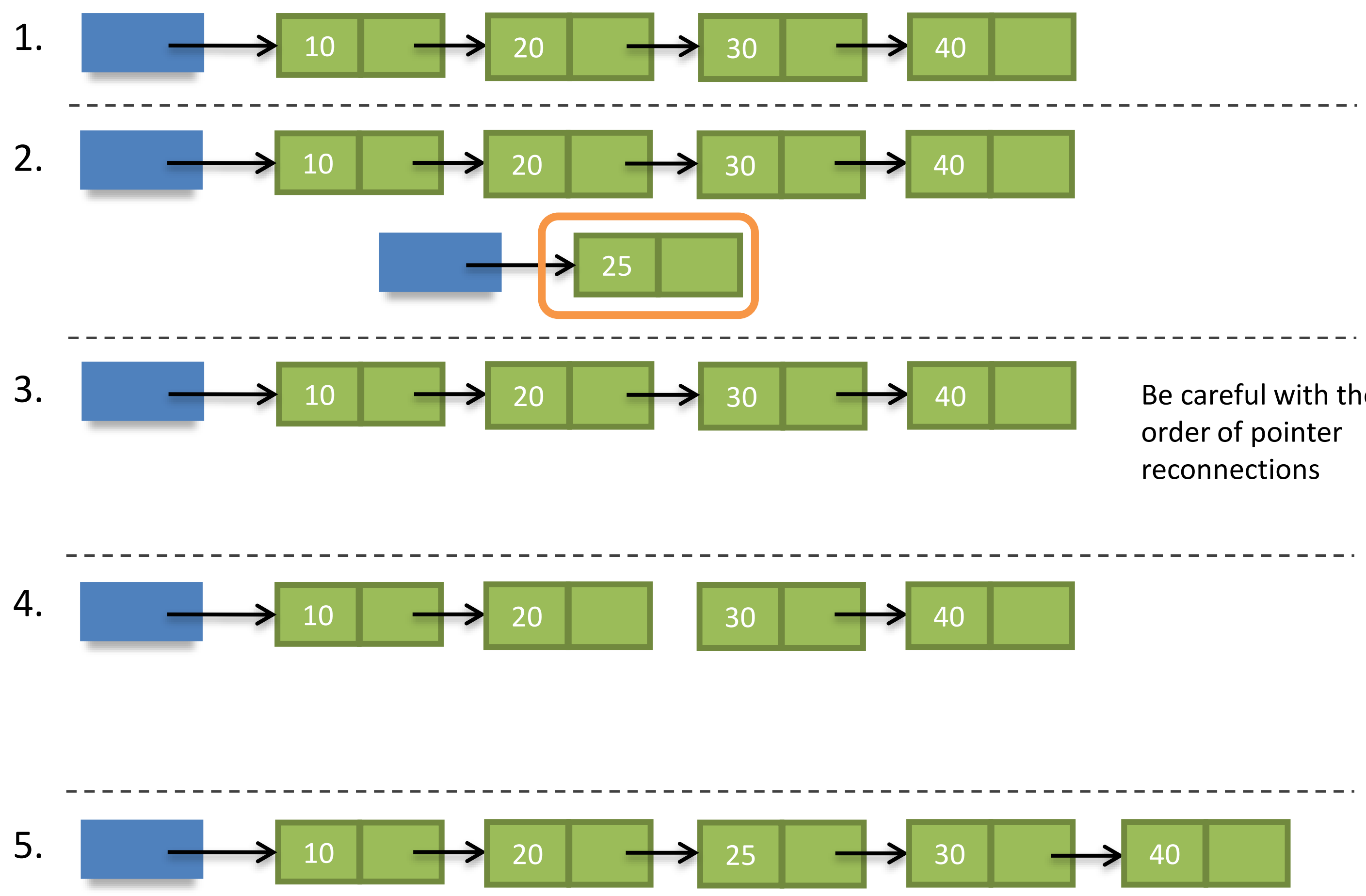
1. Create a node by the given item
2. Insert the node at
   1. **Front**
   2. **Middle**
   3. **Back**

Insert a node in the middle



Be careful with the order of pointer reconnections

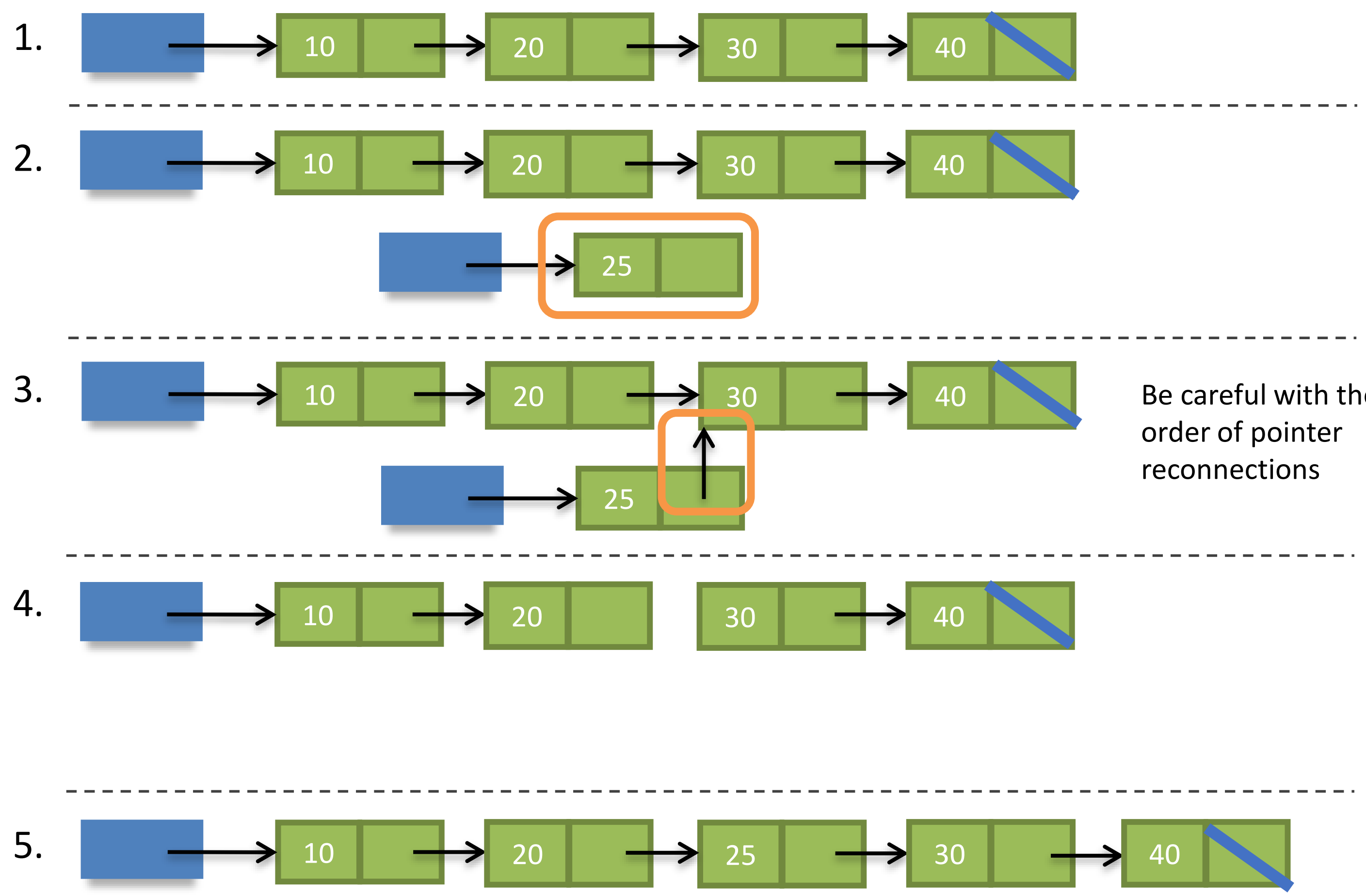
Insert a node in the middle



Be careful with the order of pointer reconnections

```
new_node = Node(data)
```

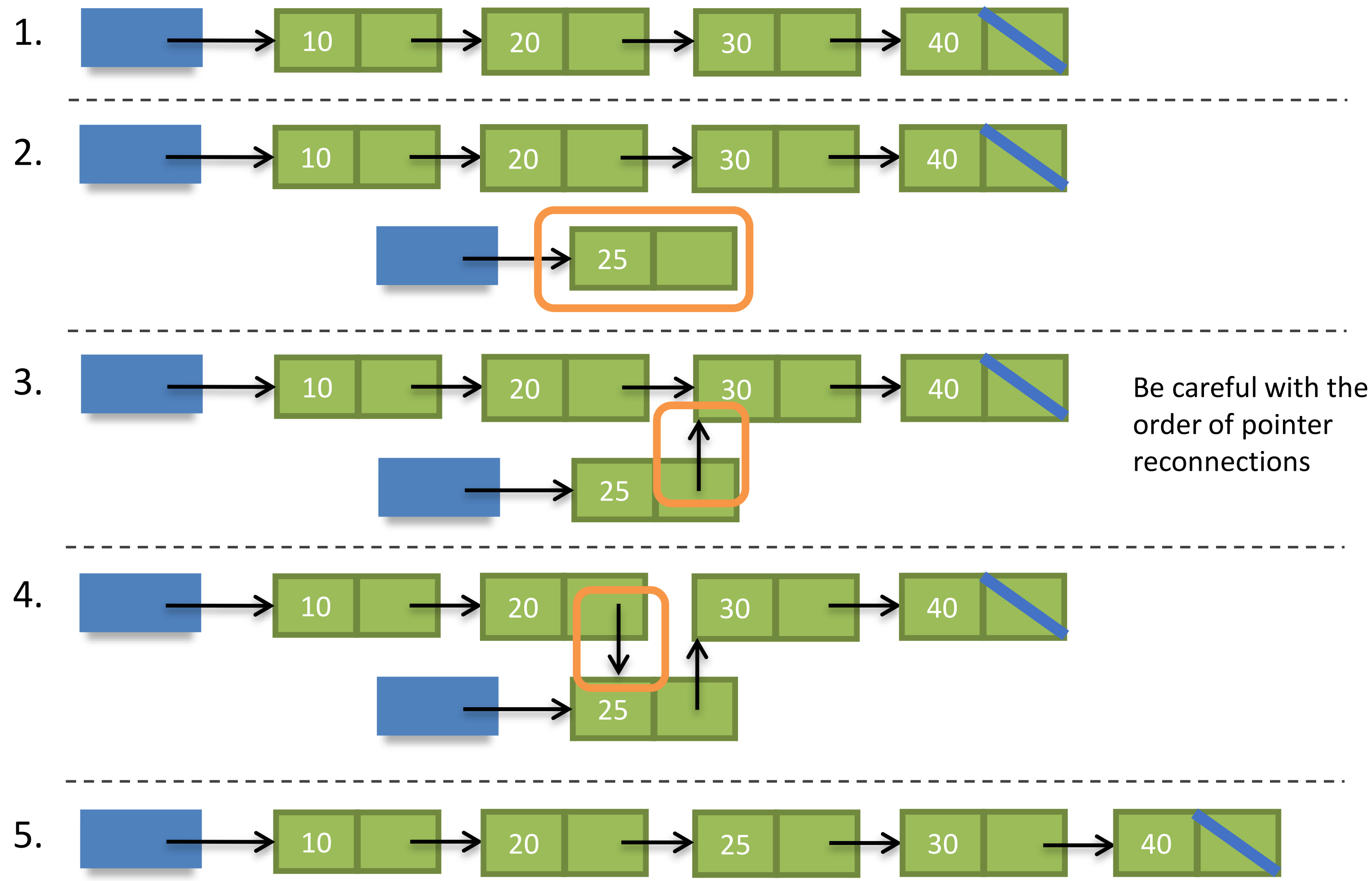
Insert a node in the middle



Be careful with the order of pointer reconnections

```
new_node.next = pre.next
```

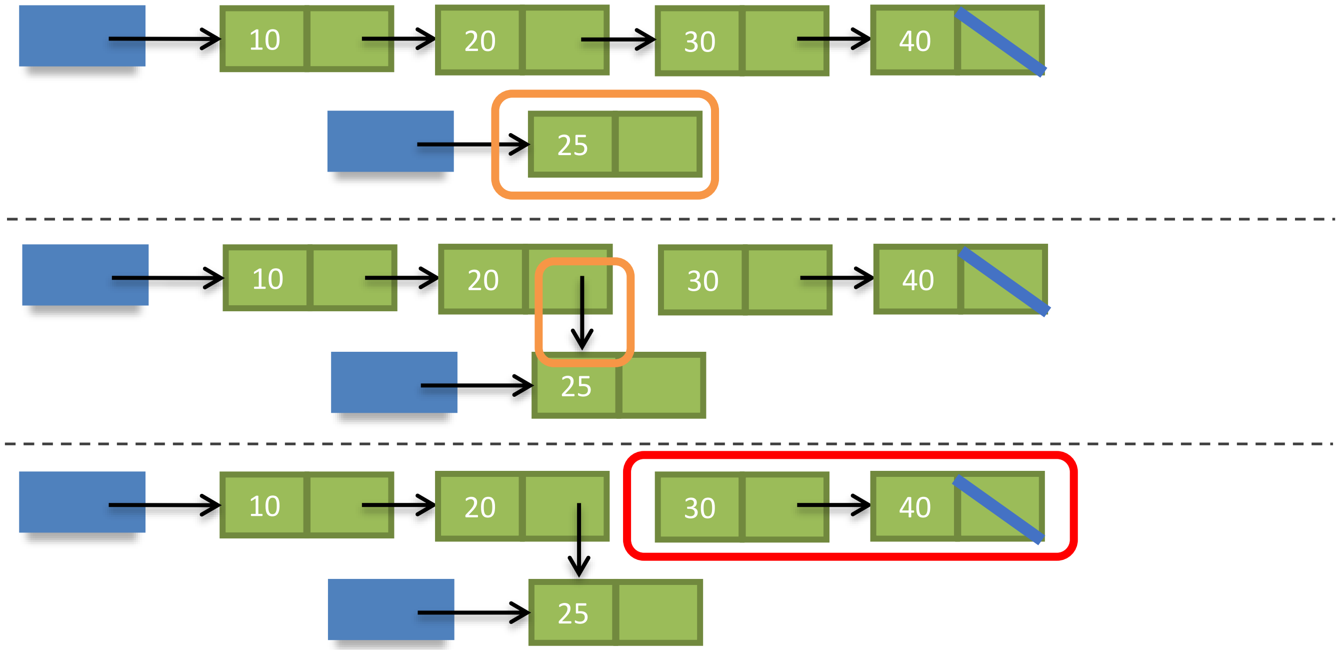
Insert a node in the middle



Be careful with the order of pointer reconnections

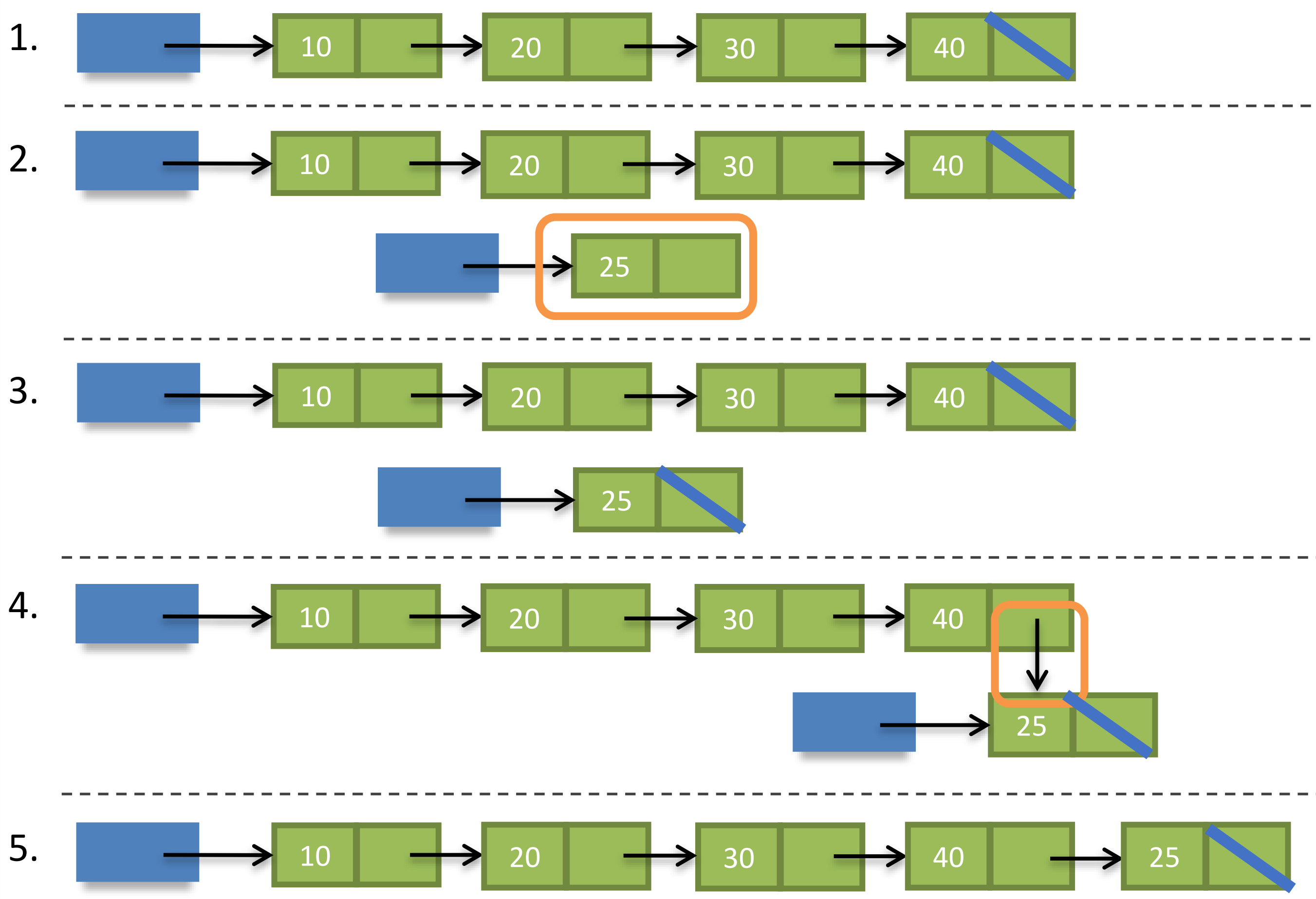
```
pre.next = new_node
```

Insert a node in the middle

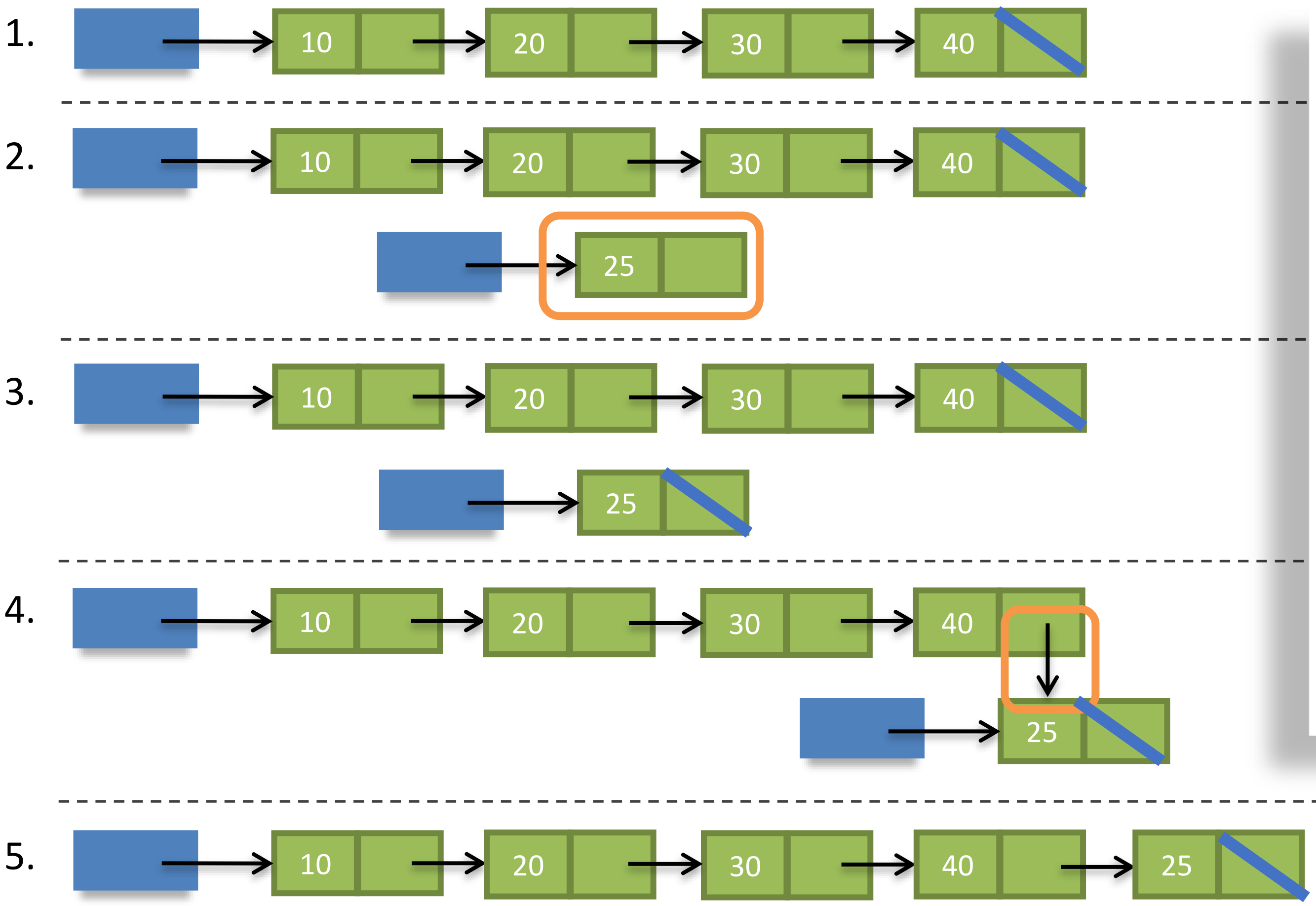



```
pre.next = new_node
```

Insert a node in the back



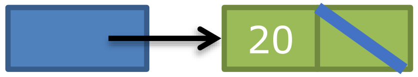
Insert a node in the back



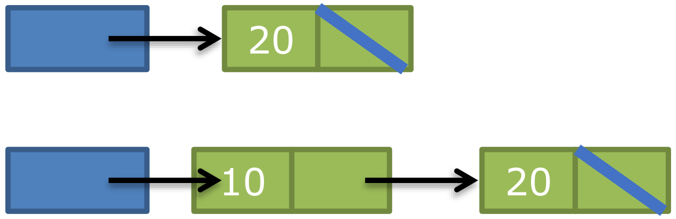

```python
def insert_at_back(self, data):

    new_node = Node(data)

    if not self.head:
        self.head = new_node
        return

    last_node = self.head

    while last_node.next:
        last_node = last_node.next

    last_node.next = new_node
```

Insert a node at the front

- What is common issue to both cases?
  - Empty list
  -  None



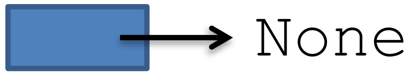
  - Inserting a node at index 0



Need to modify the content of head pointer

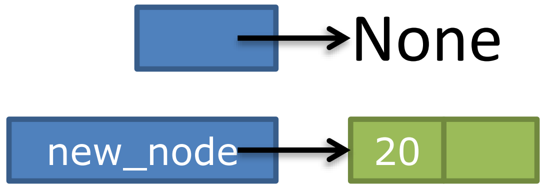
Insert a node at the front

- What is common issue to both cases?
  - Empty list



Insert a node at the front

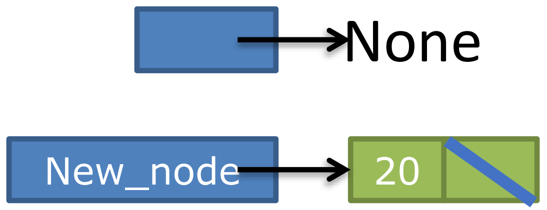
- What is common issue to both cases?
  - Empty list




```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
```

Insert a node at the front

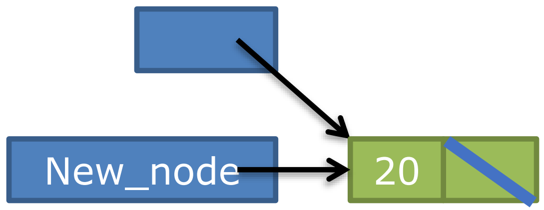
- What is common issue to both cases?
  - Empty list

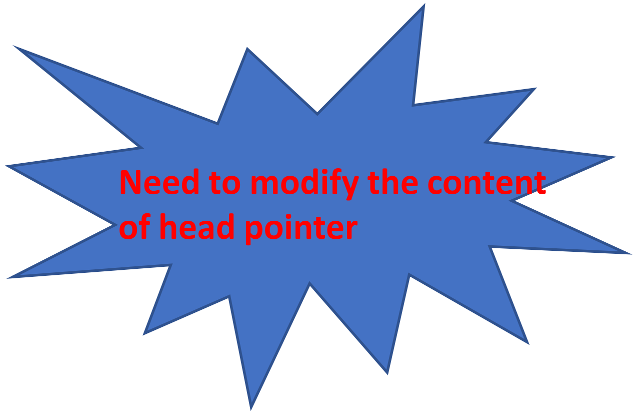



```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
    # Step 2: Point new node to the current head
    new_node.next = self.head
```

Insert a node at the front

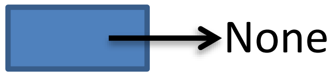
- What is common issue to both cases?
  - Empty list



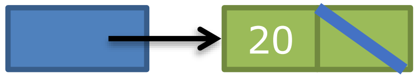



```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
    # Step 2: Point new node to the current head
    new_node.next = self.head
    # Step 3: Update the head to point to the new node
    self.head = new_node
```

Insert a node at the front

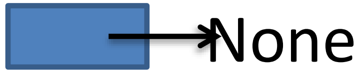
- What is common issue to both cases?
  - Empty list
  - 

  - Inserting a node at index 0

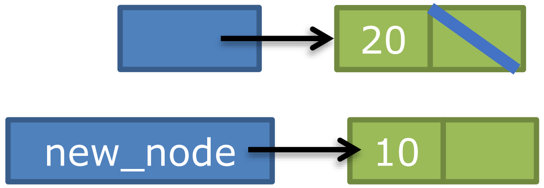


Insert a node at the front

- What is common issue to both cases?
  - Empty list



  - Inserting a node at index 0

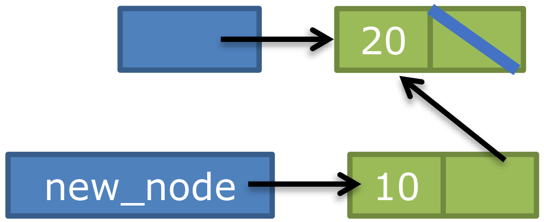



```
def insert_at_front(self, data):
        # Step 1: Create a new node
        new_node = Node(data)
```

Insert a node at the front

- What is common issue to both cases?
  - Empty list

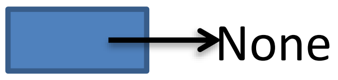
  - Inserting a node at index 0



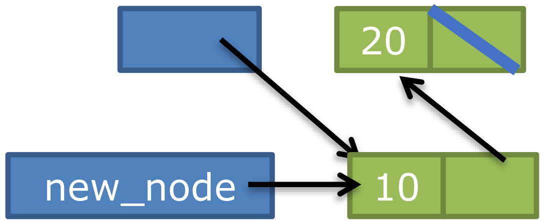

```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
    # Step 2: Point new node to the current head
    new_node.next = self.head
```

Insert a node at the front

- What is common issue to both cases?
  - Empty list



  - Inserting a node at index 0




```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
    # Step 2: Point new node to the current head
    new_node.next = self.head
    # Step 3: Update the head to point to the new node
    self.head = new_node
```

Insert a node at the front

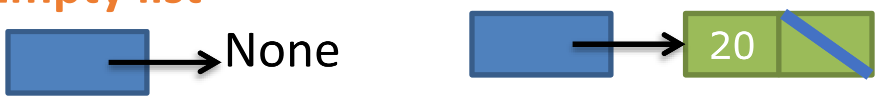
- What is common issue to both cases?
  - Empty list
  - 

  - Inserting a node at index 0

Need to modify the content of head pointer

```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
    # Step 2: Point new node to the current head
    new_node.next = self.head
    # Step 3: Update the head to point to the new node
    self.head = new_node
```

Insert a node at the front

- What is common issue to both cases?
  - Empty list
  - 

  - Inserting a node at index 0

Need to modify the content of head pointer

```
def insert_at_front(self, data):
    # Step 1: Create a new node
    new_node = Node(data)
    # Step 2: Point new node to the current head
    new_node.next = self.head
    # Step 3: Update the head to point to the new node
    self.head = new_node
```

```
def insert_at_back(self, data):

    new_node = Node(data)

    if not self.head:
        self.head = new_node
        return

    last_node = self.head

    while last_node.next:
        last_node = last_node.next

    last_node.next = new_node
```

Append node at given index

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

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_back(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def insert_at_front(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def findAt(self, index):
        current = self.head
        if not current:
            return None
        while index>0:
            current = current.next
            if not current:
                return None
            index-=1
        return current
```

Insert node at given index

```
def insert(self, data, index):
    # Create a new node with the given data
    new_node = Node(data)

    # If list is empty or inserting at head
    if self.head is None or index == 0:
        new_node.next = self.head
        self.head = new_node
        return True

    # Start at the head of the list
    current = self.head
    count = 0
    # Traverse until index-1 position
    # (node before where we want to insert)
    while current and position < index - 1:
        current = current.next
        count += 1

    # If current is None, index was too large
    if not current:
        print("Index out of range")
        return False

    # Insert the new node by updating pointers:
    # 1. New node points to current's next node
    # 2. Current node points to new node
    new_node.next = current.next
    current.next = new_node
    return True
```

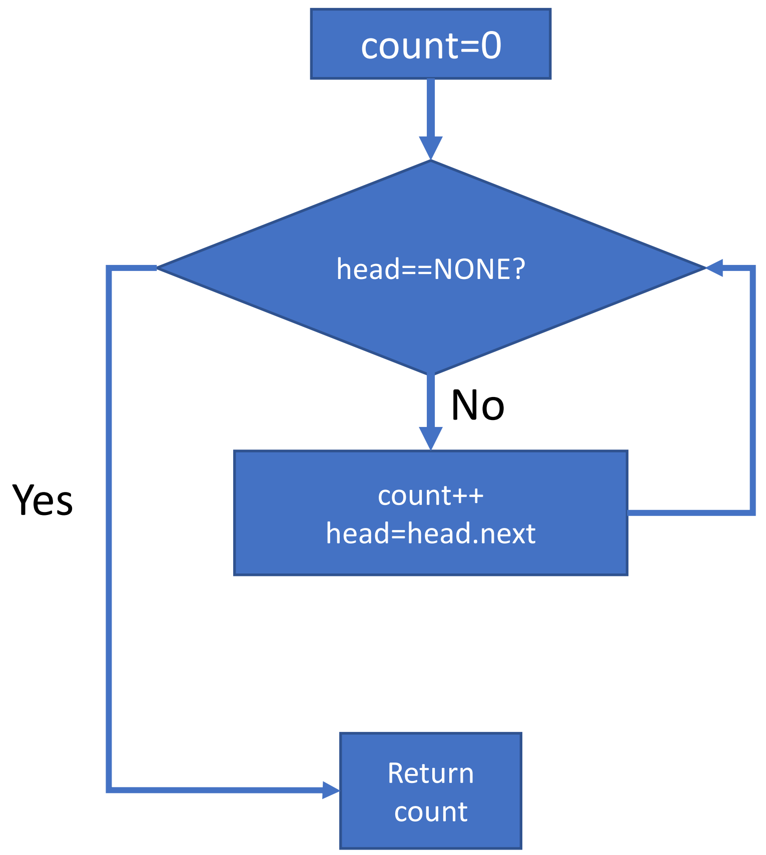
Size of The LinkedList (not smart solution)

Given

- the head reference of the linked list

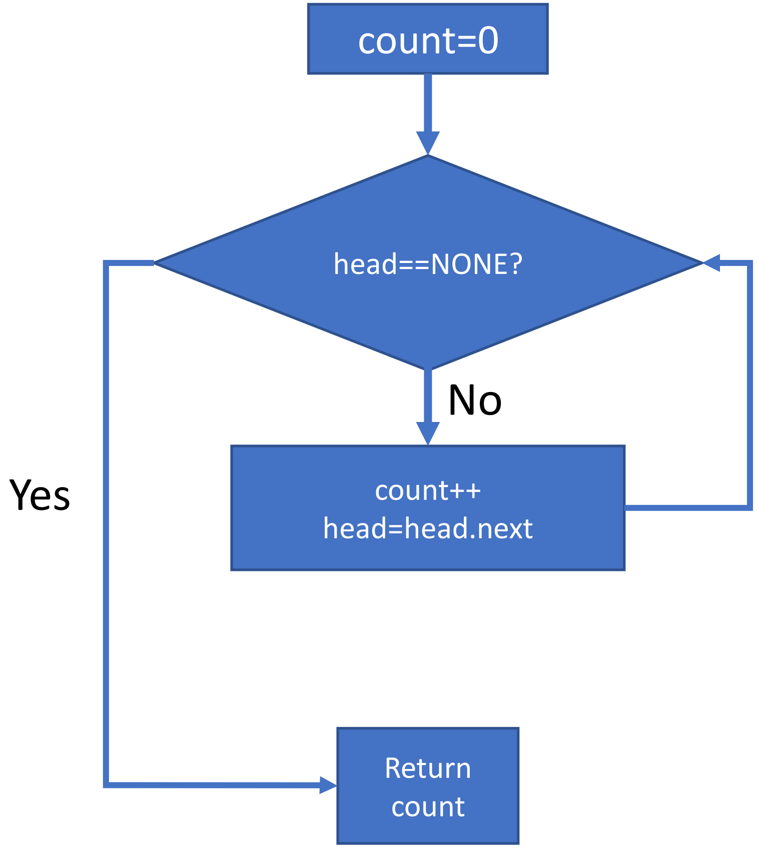
Return the number of nodes in the linked list

1. Declare a counter and initialize it to zero
2. Check the reference whether is None or not
3. Increase the counter
4. Head move to next node
5. Repeat step 2
6. Return the counter



Size of The LinkedList (not smart solution)

```python
def sizeList(head):

    # Initialize counter
    count = 0

    # Start at head
    current = head

    # Traverse the list
    while current is not None:
        count += 1  # Increase counter
        current = current.next  # Move to next node

    return count  # Return final count
```



The Linked List

- Just introduce a new member in the linked list class, *size*
- Initialize *size* as zero
- When you add or remove a node, increase or decrease ***size*** by one accordingly

```
class ListNode:
    def __init__(self, item):
        self.item = item
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0
```

Size of The Linked List (better solution)

- Just introduce a new member in the linked list class, *size*
- Initialize *size* as zero
- When you add or remove a node, increase or decrease *size* by one accordingly

```
class ListNode:
    def __init__(self, item):
        self.item = item
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

def sizeList(ll): # ll = LinkedList() #
   return ll.size
```

findNode

```
def findNode(head, index):
    if head is None or index < 0:
        return None
    cur = head
    while index > 0:
        cur = cur.next
        if cur is None:
            return None
        index -= 1
    return cur
```

```
def findNode2(ll, index):
    # Check if list is empty or index is invalid
    if ll.head is None or index < 0 or index >= ll.size:
        return None

    # Start traversing from head
    cur = ll.head
    while index > 0:
        cur = cur.next
        index -= 1
    return cur
```

insertNode

```python
def insertNode(head, index, item):
    newNode = ListNode(item)
    if head is None:
        return newNode

    if index == 0:
        newNode.next = head
        return newNode

    prev = findNode(head, index - 1)

    if prev is not None:
        newNode.next = prev.next
        prev.next = newNode
    return head
```

```python
def insertNode(head, index, item):
    newNode = ListNode(item)

    # Scenario 1: Inserting into empty list
    if head is None:
        if index == 0:
            head = newNode
            return True
        return False

    # Scenario 2: Inserting at beginning of non-empty list
    if index == 0:
        newNode.next = head
        head = newNode
        return True

    # Scenario 3: Inserting in middle or end of list
    prev = findNode(head, index - 1)
    if prev is not None:
        newNode.next = prev.next
        prev.next = newNode
        return True
    return False
```

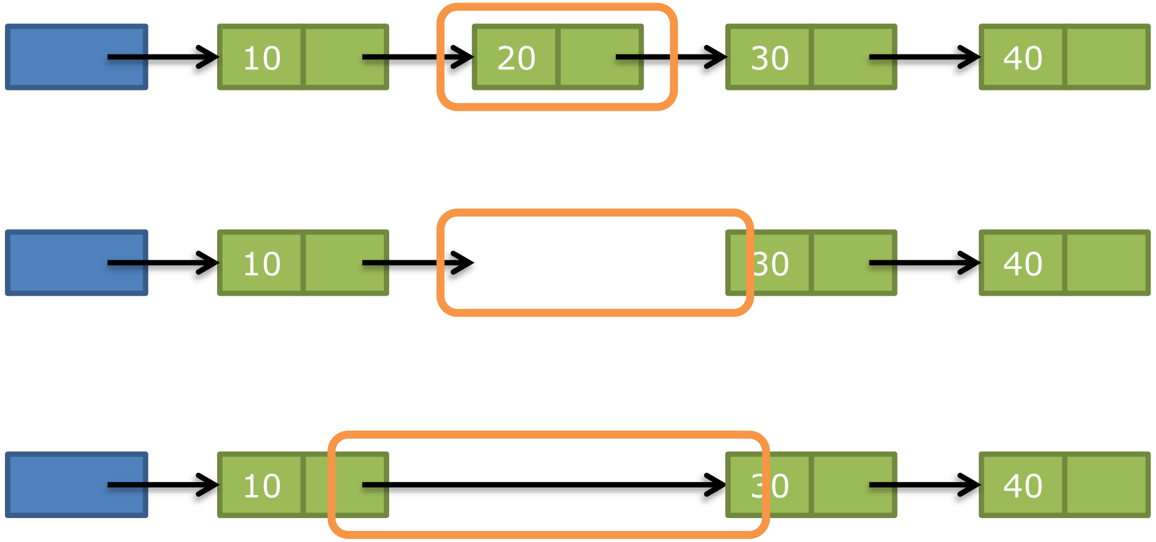
insertNode

```python
def insertNode(head, index, item):
    newNode = ListNode(item)
    if head is None:
        return newNode

    if index == 0:
        newNode.next = head
        return newNode

    prev = findNode(head, index - 1)

    if prev is not None:
        newNode.next = prev.next
        prev.next = newNode
    return head
```

```python
def insertNode2(ll, index, item):
    newNode = ListNode(item)

    # Case 1: Inserting at the beginning
    if index == 0:
        newNode.next = ll.head
        ll.head = newNode
        ll.size += 1
        return True

    # Case 2: Inserting anywhere else
    pre = findNode2(ll, index - 1)

    if pre is not None:
        newNode.next = pre.next
        pre.next = newNode
        ll.size += 1
        return True

    return False
```

Remove a Node (Lab Questions)

- Remove a node at
  1. Front
  2. Middle
  3. Back



Real world Application – Task Scheduling

Task Scheduling :

Linked lists are often used in real-world task scheduling, especially in operating systems and other environments where efficient management of tasks or processes is critical.

Advantages of Linked Lists in Task Scheduling

– Efficient Insertion/Deletion: Linked lists allow for quick insertion and deletion of tasks, which is essential in environments where tasks frequently enter and exit the queue.

– Dynamic Sizing: No pre-allocation of memory is needed, making linked lists ideal for systems with fluctuating task loads.

Summary

- Referential Arrays
- Customized Data Types
- Singly Linked List
- Abstract Data Type (ADT)
  - Display data (Print all data)
  - Search a node
  - Add a node
  - Remove a node
  - Get the size of the linked list