VNUHCM - University of Science
fit@hcmus

Fundamentals of Programming for Artificial Intelligence

Session 07
Singly Linked List

Instructors:
Dr. Lê Thanh Tùng
Dr. Nguyễn Tiến Huy

Content

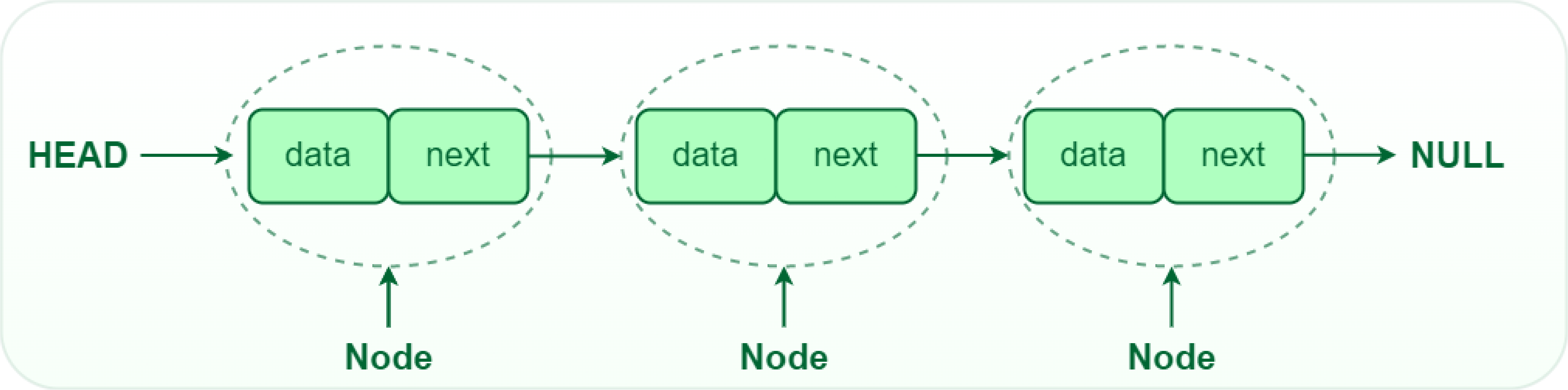
1. Linked List

- A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the **data** and the **address** of the next node

- Unlike Arrays, Linked List elements are not stored at a contiguous location

Advantages of Linked List

- Dynamic Data structure: based on the operation insertion or deletion
- Ease of Insertion/Deletion
- Efficient Memory Utilization: avoids the wastage of memory
- Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc



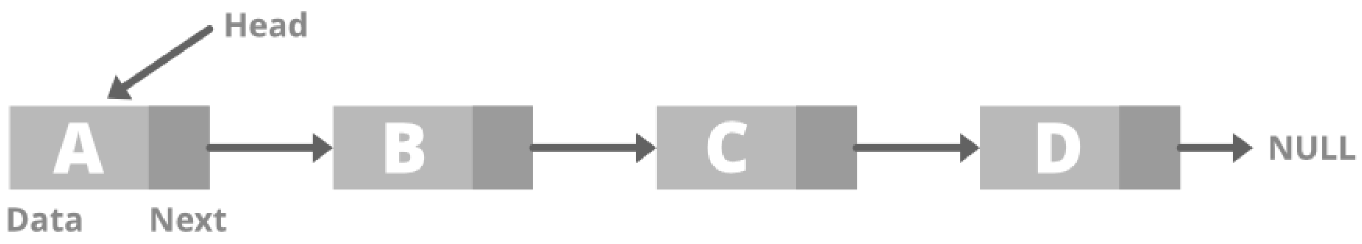
Linked List

- So, how do linked lists differ than arrays
  - An array is direct access; we supply an element number and can go directly to that element (through pointer arithmetic) – via index
  - With a linked list, we must either start at the head or the tail pointer and **sequentially traverse** to the desired position in the list

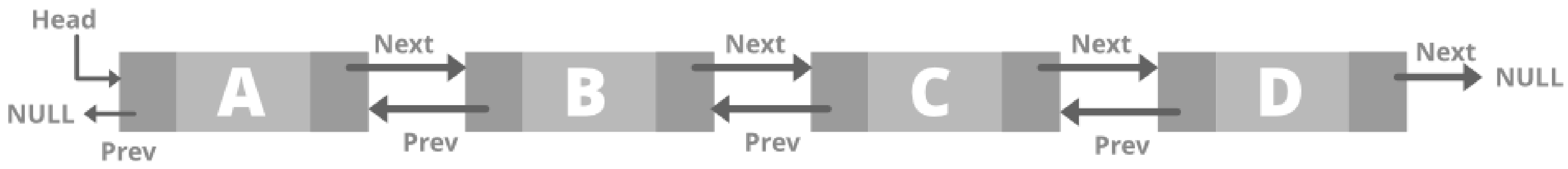
- So, how about the fundamental operations:
  - Declare …
  - Input and Output …
  - Traversal …
  - Find an element from …
  - Add an element into …
  - Delete an element from …
  - Sort the … in ascending/descending

Types of Linked List

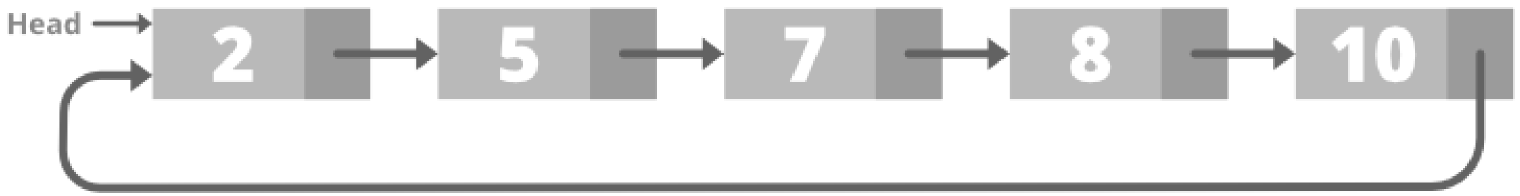
- Singly Linked List:



- Doubly Linked List:



- Circular Linked List



JSON Format

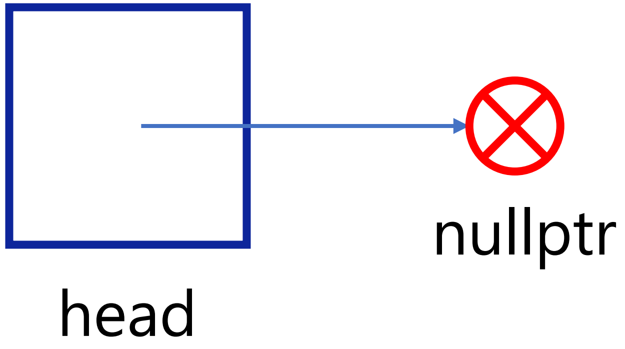
- JSON (JavaScript Object Notation) is a popular data format used for representing structured data
- JSON format is quite similar dictionary in Python

```
{
    "emp_details": [
        {
            "emp_name": "Shubham",
            "email": "ksingh.shubh@gmail.com",
            "job_profile": "intern"
        },
        {
            "emp_name": "Gaurav",
            "email": "gaurav.singh@gmail.com",
            "job_profile": "developer"
        },
        {
            "emp_name": "Nikhil",
            "email": "nikhil@geeksforgeeks.org'
            "job_profile": "Full Time"
        }
    ]
}
```

2. Singly Linked List

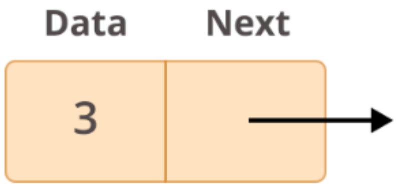
Singly Linked List

- A linear linked list starts out as empty
  - An empty list is represented by a null pointer
  - We commonly call this the head pointer



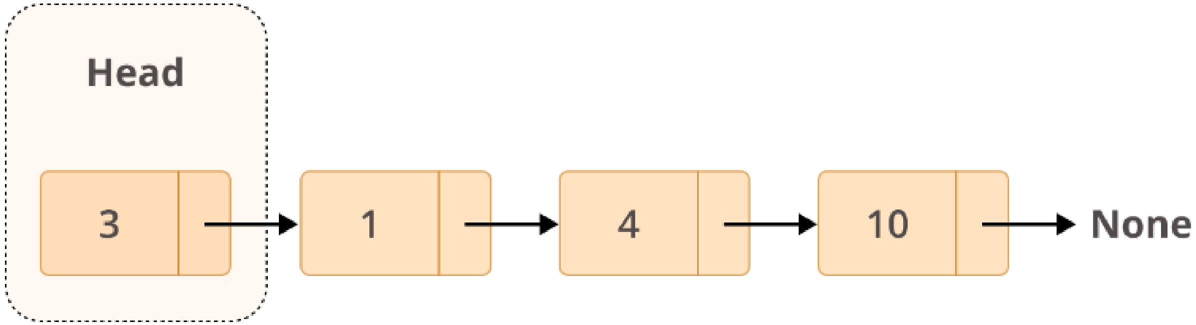
Node in Linked List

- A node in a linked list contains one or more members that represent data
- Each node also contains (at least) a link to another node



Node

- A linked list is a collection of nodes
- The first node is called the head



Linked List

Create a Linked List

- Create the node to store the data

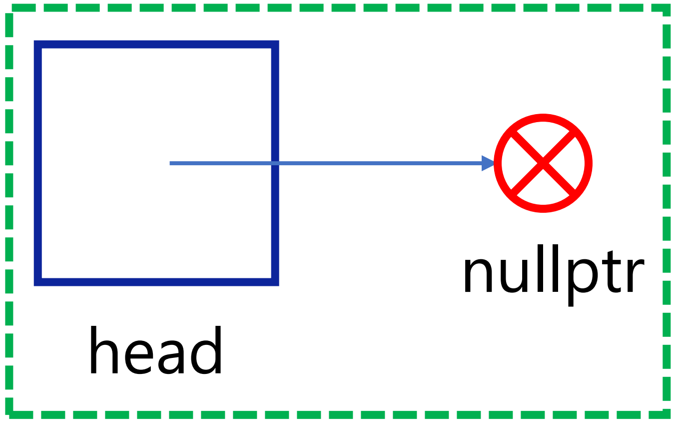
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

- Create a class LinkedList to represent your linked list

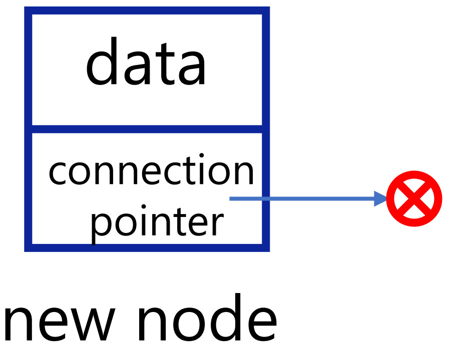
```python
class LinkedList:
    def __init__(self):
        self.head = None
```

Singly Linked List

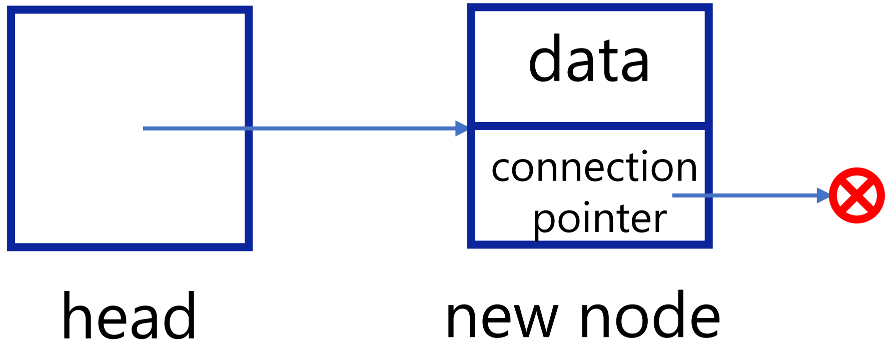
- As we add the first data item, the list gets one node added to it with the following procedure:
  - Create a node containing data and connection
  - Add node into the current list



current list



1. Create node

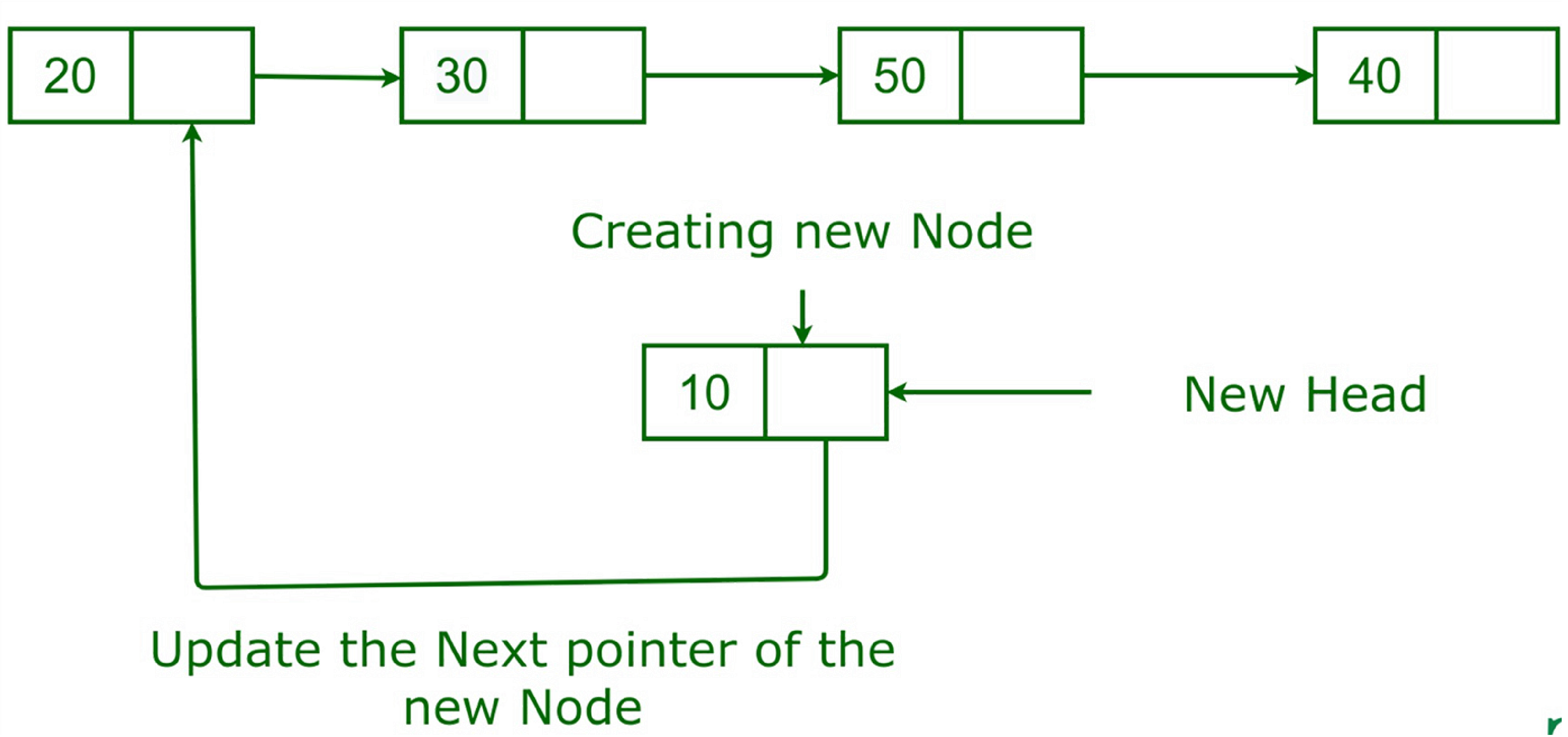


2. Add node into list

- To add another data item we must first decide in what order
  - does it get added at the **beginning**
  - does it get inserted in **sorted order**
  - does it get added at the **end**
  - does it get added at the **position pos**

Add Node to Head

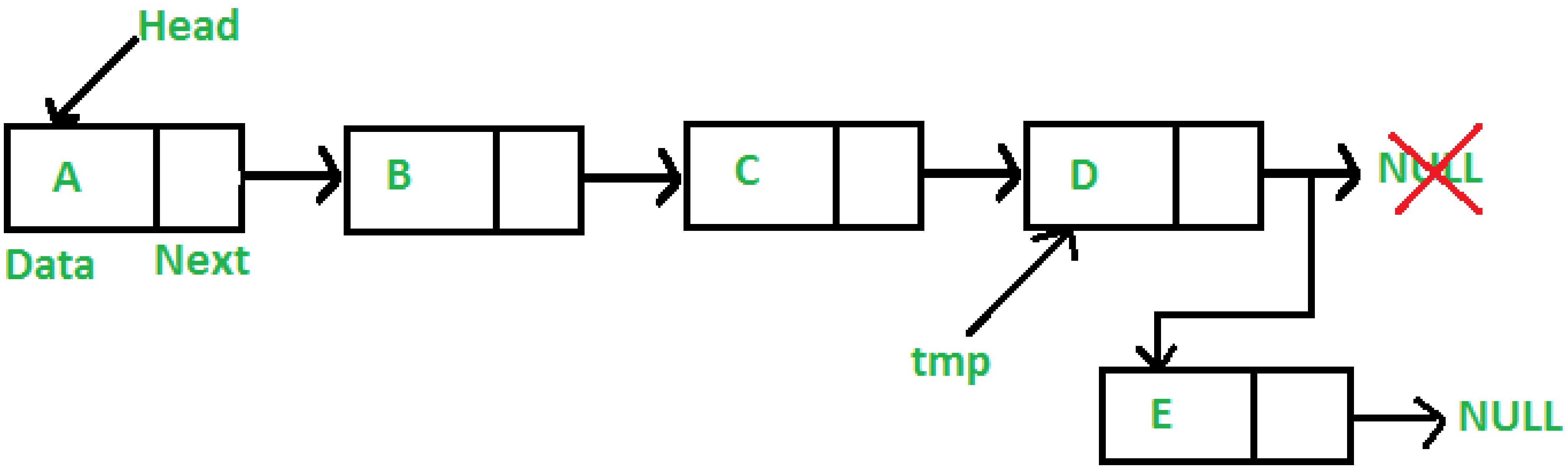
- Assume that we will control a linked list by only pointer **head**



Add Node to Head

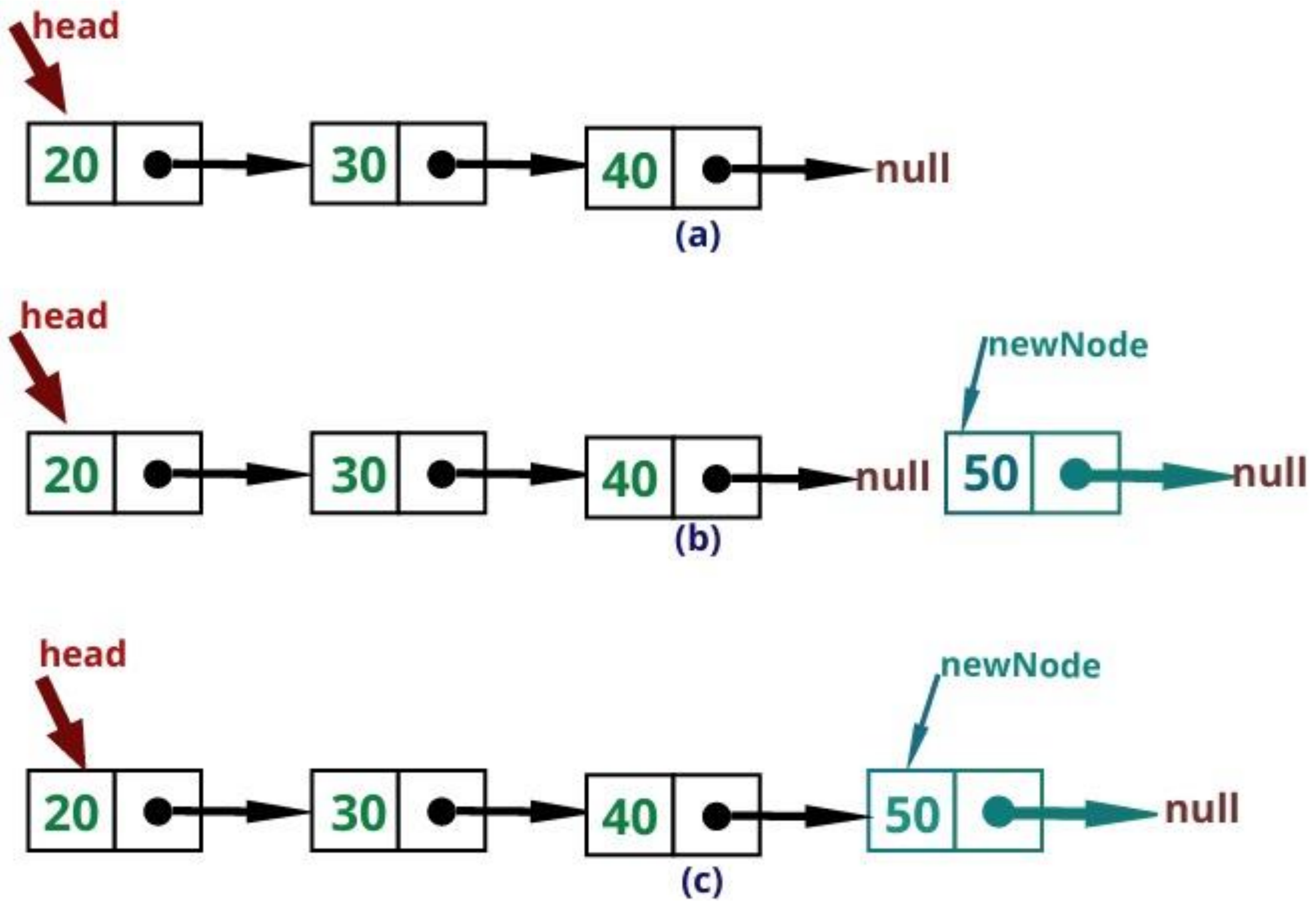
```
def addHead(self, data):
    newNode = Node(data)
    if self.head is None:
        self.head = newNode
    else:
        newNode.next = self.head
        self.head = newNode
```

Add Tail



Add Node to Tail

- Assume that we will control a linked list by only pointer **pHead**



Sequential Traversal

- Move to the next node via the **next** pointer:

```
// move to the next node
pCur = pCur.next;
```

- Display all the members in the linked list

Traverse a Linked List

- Print all value of data in a linked list with pointer head

```python
def traverse(self):
    if self.head is None:
        print("Empty List")
        return
    else:
        curNode = self.head
        while (curNode is not None):
            print(curNode.data, end = "->")
            curNode = curNode.next
```

- **Note: Must not change the value of head node**

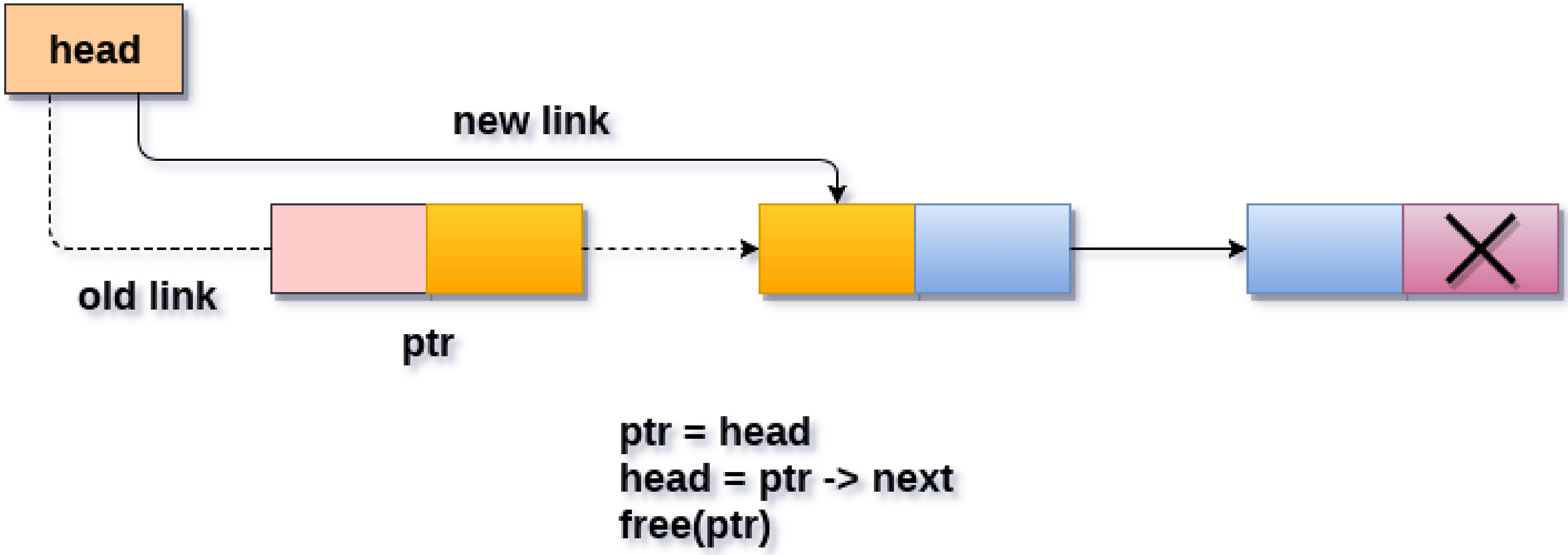
Add Tail

- So, to add a node in the end of linked list, we should go to the last node of the list
  - Step 1: Traverse into the end of the list
  - Step 2: Change the **`next`** of ending node into the new node

Add Node to Tail

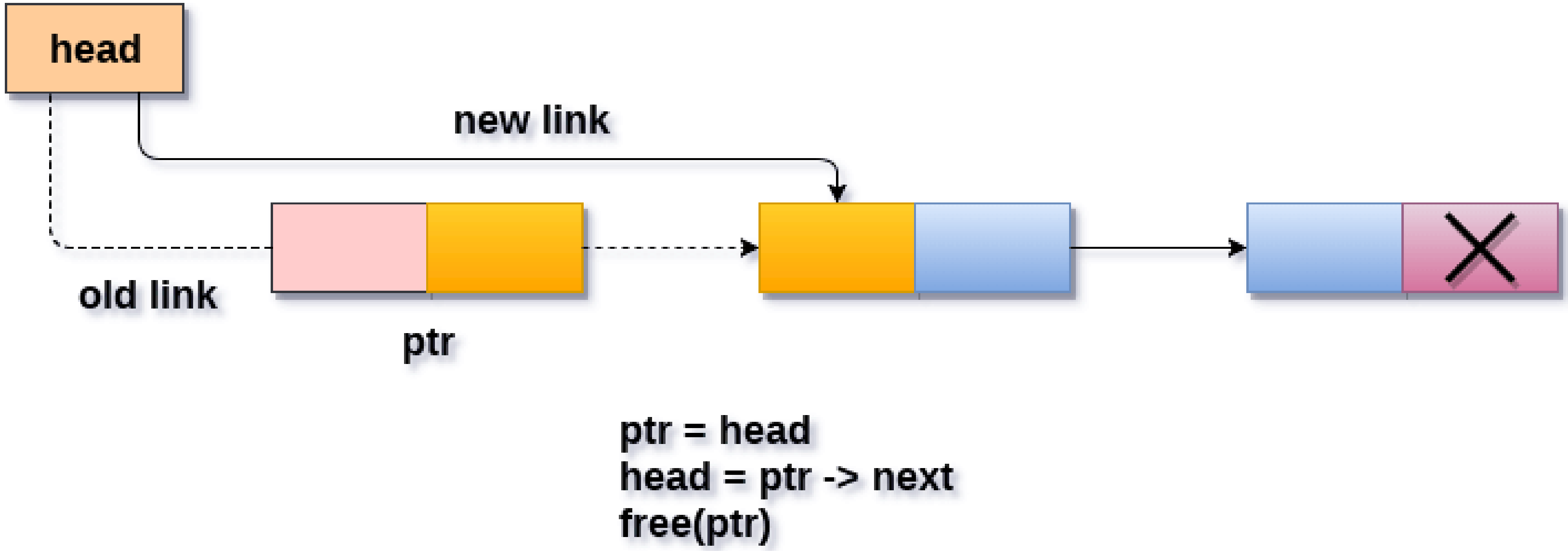

```python
def addTail(self, data):
    newNode = Node(data)
    if self.head is None:
        self.head = newNode
    else:
        curNode = self.head
        while curNode.next is not None:
            curNode = curNode.next
        curNode.next = newNode
```

Remove First Node



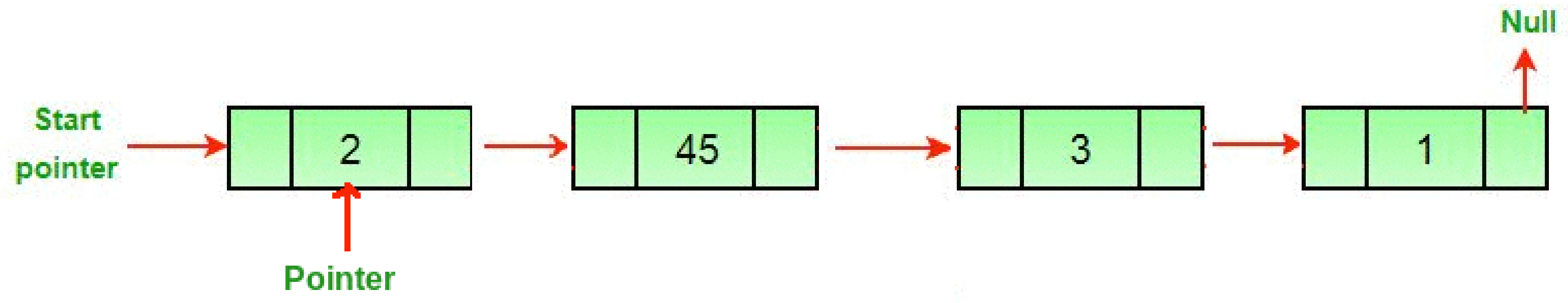
Deleting a node from the beginning

Remove First Node




```python
def removeHead(self):
    if self.head is not None:
        self.head = self.head.next
```

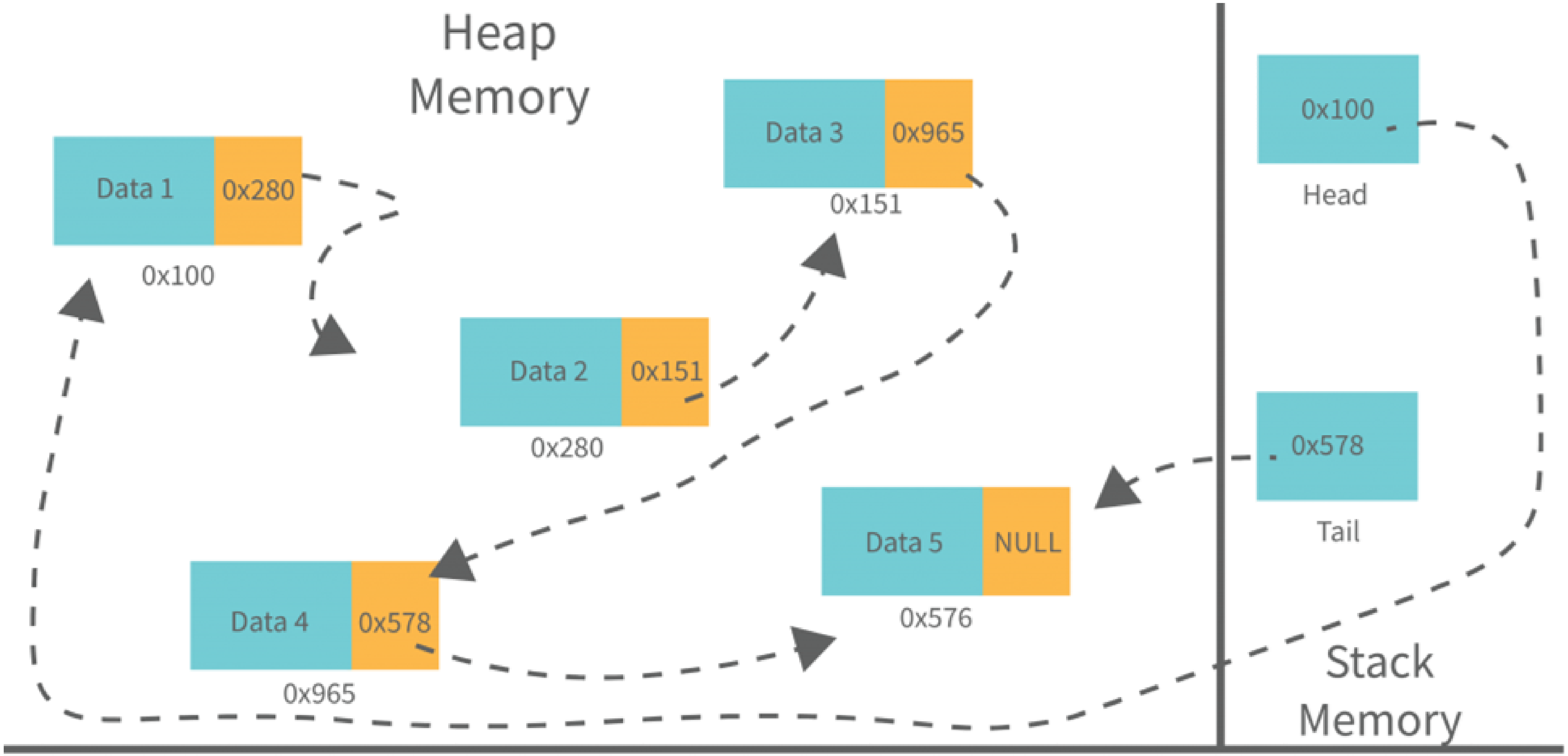
Remove Last Node



Remove Last Node

fit@hcmus

```
def removeTail(self):
    if self.head is not None:
        # only 1 Node
        if self.head.next is None:
            self.head = None
        else:
            curNode = self.head
            while curNode.next.next is not None:
                curNode = curNode.next
            curNode.next = None
```



Main Memory Representation

Linked List Vs Array

Array	Linked List
Stored in contiguous location	Not stored in contiguous location
Fixed in size	Dynamic in Size
Memory is allocated at compile time	Memory is allocated at run time
Less memory than linked list for each element	More memory, cuz it stores data and connection (address to next node)
Random Access via index	Access via sequential traversal
Insertion and deletion ops takes time	Insertion and deletion ops is faster

Exercise

- Write a function to get the length of singly linked list

Exercise

- Write a function to get the length of singly linked list

```
def getLength(self):
    size = 0
    curNode = self.head
    while curNode is not None:
        curNode = curNode.next
        size += 1
    return size
```

Exercise

- Write a function to get the length of singly linked list

```
def __len__(self):
    size = 0
    curNode = self.head
    while curNode is not None:
        curNode = curNode.next
        size += 1
    return size
```

Exercise

- Write a function to add the value into the position pos of linked list

Example:

- Input: insert 10 into position 2

$$3 \rightarrow 5 \rightarrow 8 \rightarrow 4$$

- Out:

$$3 \rightarrow 5 \rightarrow 10 \rightarrow 8 \rightarrow 4$$

0 1 2 3 4

Solution:

- Traverse the Linked list up to position -1 nodes.
- Once all the position -1 nodes are traversed, create new Node
- Point the next pointer of the new node to the next of current node.
- Point the next pointer of current node to the new node.

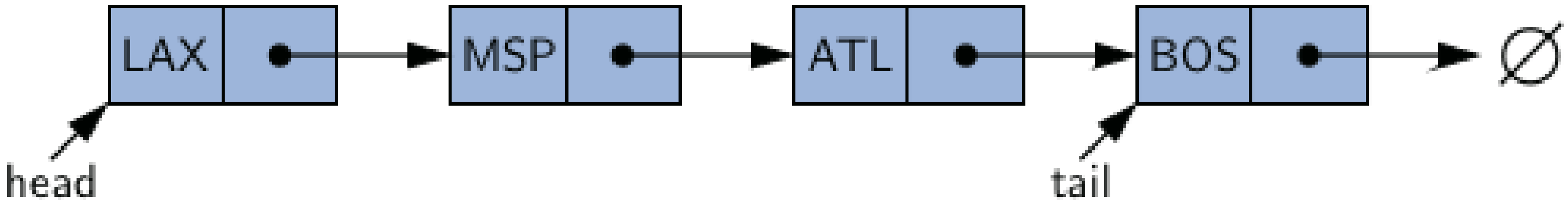
Insert Node with Position


```python
def insertAtPos(self, data, pos):
    if pos == 0: self.addHead(data)
    elif pos >= self.getLength(): self.addTail(data)
    else:
        curNode = self.head
        while pos != 1:
            curNode = curNode.next
            pos = pos - 1
        newNode = Node(data)
        newNode.next = curNode.next
        curNode.next = newNode
```

3. Singly Linked List with Tail Pointer

Linked List

- Control a linked list with pointer to head and tail



- Now, let's write again the function to add a node at the end of list

```python
def addTail(self, data):
    newNode = Node(data)
    if self.head is None:
        self.head = newNode
        self.tail = newNode
    else:
        self.tail.next = newNode
        self.tail = newNode
```

Linked List

Tail Pointer	Without Tail Pointer
Insertion at the End: directly	Insertion at the End: sequentially traverse
Memory Usage: Slightly higher due to the additional pointer	More memory-efficient
Use Cases: Better suited for scenarios such as queue implementations	Use Cases: Suitable for applications where end-insertions are rare or where the list is not expected to grow significantly

Linked List with Tail

- Rewrite the function:
  - Remove Last Node
  - Add Node before the last node

Linked List with Tail

```python
def remove_last_node(self):
    if self.head is None:   return "List is empty"
    if self.head == self.tail:
        self.head = None
        self.tail = None
    else:
        current = self.head
        while current.next != self.tail:
            current = current.next
        current.next = None
        self.tail = current
```

Linked List with Tail


```python
def add_node_before_last(self, data):
    if self.head is None or self.head == self.tail:
        return "Not enough nodes to perform the operation"
    newNode = Node(data)
    current = self.head
    # Traverse until the second last node
    while current.next != self.tail:
        current = current.next
    # Insert the new node before the last node
    newNode.next = self.tail
    current.next = newNode
```

Tail Pointer	Without Tail Pointer
Insertion at the End: directly	Insertion at the End: sequentially traverse
Memory Usage: Slightly higher due to the additional pointer	More memory-efficient
Use Cases: Better suited for scenarios such as queue implementations	Use Cases: Suitable for applications where end-insertions are rare or where the list is not expected to grow significantly

THANK YOU

for YOUR ATTENTION