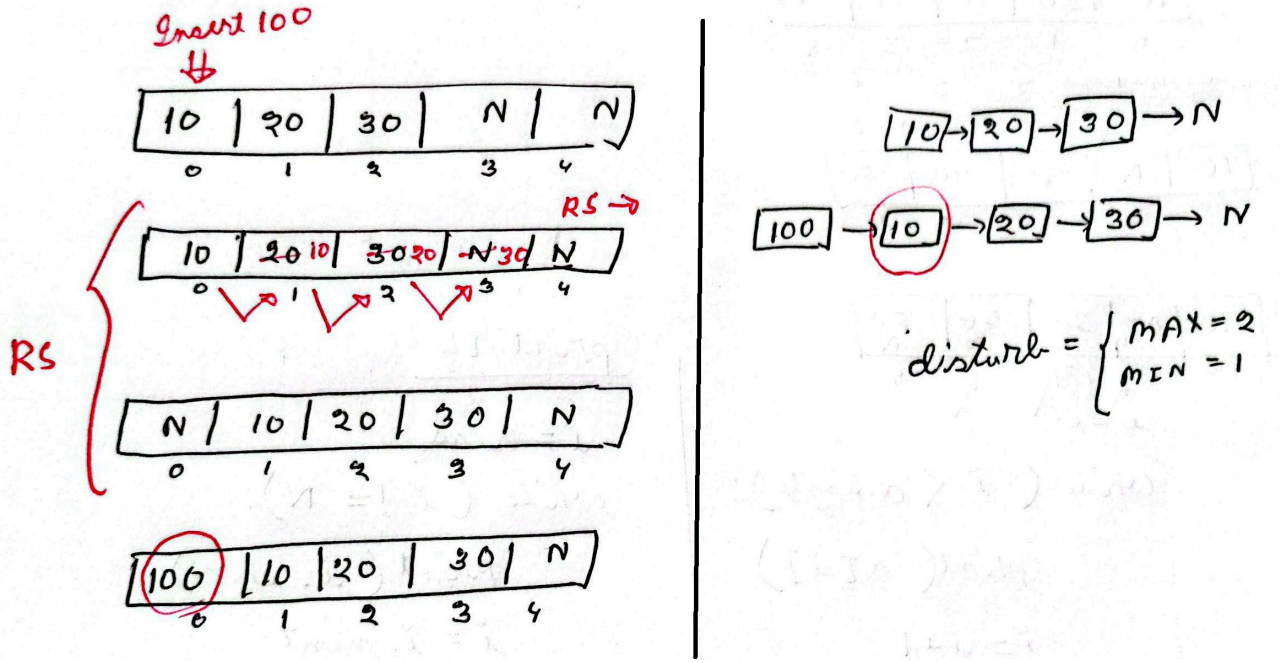
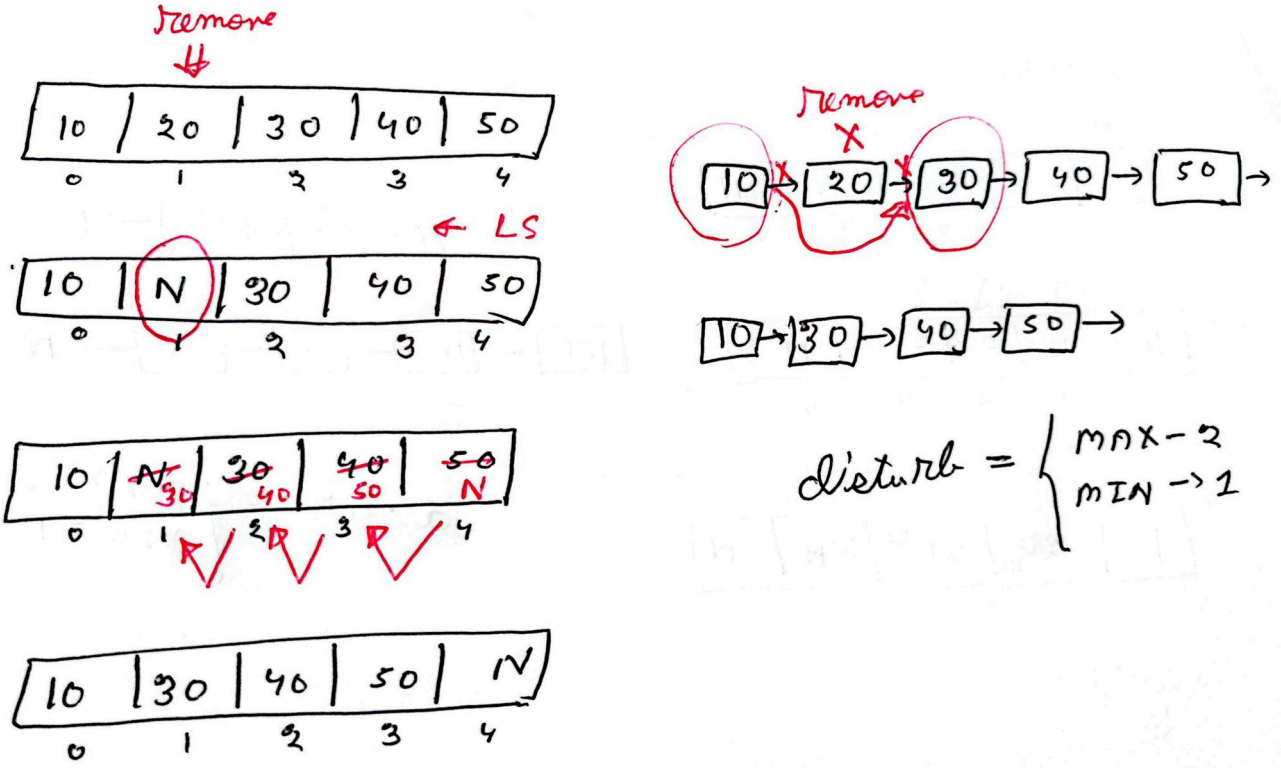
## Problems with array

- **Fixed capacity:** Once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one.
- **Shifting elements in insert:** Since we do not allow gaps in the array, inserting a new element requires that we shift all the subsequent elements right to create a hole where the new element is placed. In the worst case, we "disturb" every slot in the array when we insert it at the beginning of the array! [Do right shift]

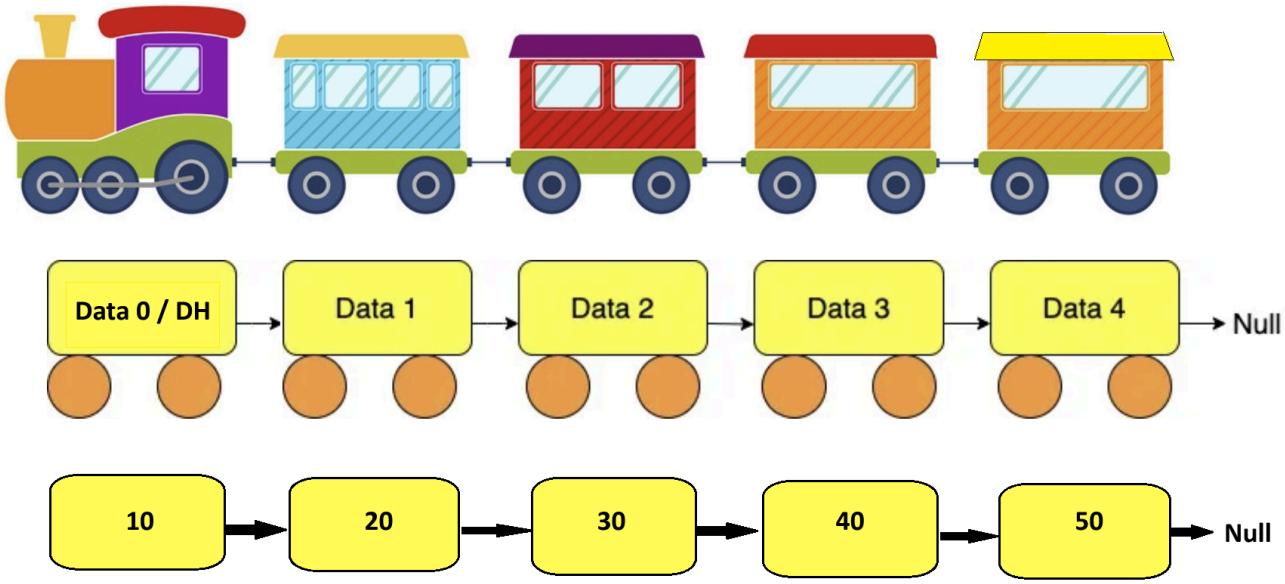



- **Shifting elements in removal:** Removing an element may also necessitate shifting others to fill the gap left behind. If the ordering of the elements does not matter, we can avoid shifting by replacing the removed element with the element in the last slot (and then treating the last slot as empty). [Do left shift]



## Solution: Linked List

- A linked list is a **sequence container** that supports **sequential access only** and requires additional space for **at least n references for the links**. Think of a **train** as the entire linked list.




## Explanation with a Train analogy

Think of a **train** as the entire linked list

### Head:

The engine of the train represents the head of the linked list. This head node is the **starting point** of the list, where the linked list begins. - Just like the **engine pulls the entire train forward**, the head node is the entry point that allows us to access and traverse the entire linked list.

- **Non-dummy headed:** The head (engine) contains data/passengers/elements.


- **Dummy headed:** The head (engine) is empty or a placeholder, with data starting from the next compartment/node. Here, DH means Dummy headed.


## Compartments:

Each compartment in the train as a **node** in the list. Each compartment (node) in the train holds two things:

- **Data:** Passengers, seats, or other items in the compartment (this is analogous to the data stored in the node).
- **Link:** Each compartment has a connector or link to the next compartment, keeping the train together in sequence. This links can have
  - only next compartment information or
  - next and previous compartment information.

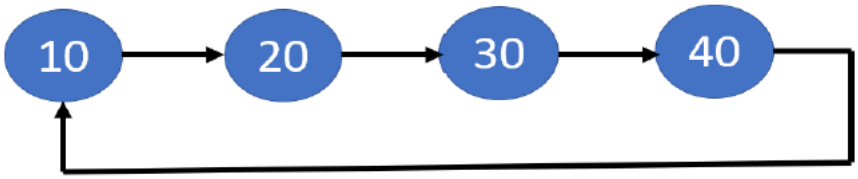
- If each compartment only keeps **next compartment** information, then its called "**Singly**" linked list

- If each compartment keeps **next and previous compartment** information, then its called "**Doubly**" linked list !


Last compartment

- **Linear:** If the last compartment is the end and it points to null/None in its "next".


- "**Circular:**" If the last compartment connects back to the first compartment




## Benefits of Linked List

- Allow the insertion of new nodes anywhere in the list or to remove of an existing node from the list without having to disturb the rest of the list
  - The only nodes affected are the ones **adjacent to the node being inserted or removed**.
- Since we can extend the list one node at a time, we can also **resize a list until we run out of resources**.

## Problems of Linked List

- No random access--Without index, we cannot acces and update a value
- Requires extra memory space to store the links
  - For singly, only next
  - For doubly, both next and prev (double the space of singly)

## Creating the node class

In order to create a linked list, we just need to create objects of the Node class and then connect one node to another using that next variable. We will only store the first Node-type object and call it head. Whenever we want to add a new node-type object, we just need to go to the last node and add the new node after that.

```
class Node:
    def __init__(self, elem, next=None): # constructor in Python, default value for next = None. No link mandatory
        self.elem = elem # data
        self.next = next # link--1---Node type
```

```
public class Node {
    int elem;
    Node next;

    public Node(int elem) {
        this.elem = elem;
        this.next = null;
    }
}
```

## Manually creating a linked list

```
# Creating the nodes
n1 = Node(10)
n2 = Node(20)
n3 = Node(30)
n4 = Node(40)

# Keeping the location of the 1st node in variable called Head
## head = n1 # This line could have been written here too, or in line 16; both are fine

# Linking the nodes
n1.next = n2 # linking 2nd node at the end of 1st node
n2.next = n3 # linking 3rd node at the end of 2nd node
n3.next = n4 # linking 4th node at the end of 3rd node
# 4th node will have null/None at end represting end of the list. None for Python, Null for Java

head = n1 # Keeping the location of the 1st node in variable called Head
```

## Important methods for linkedList (LL)

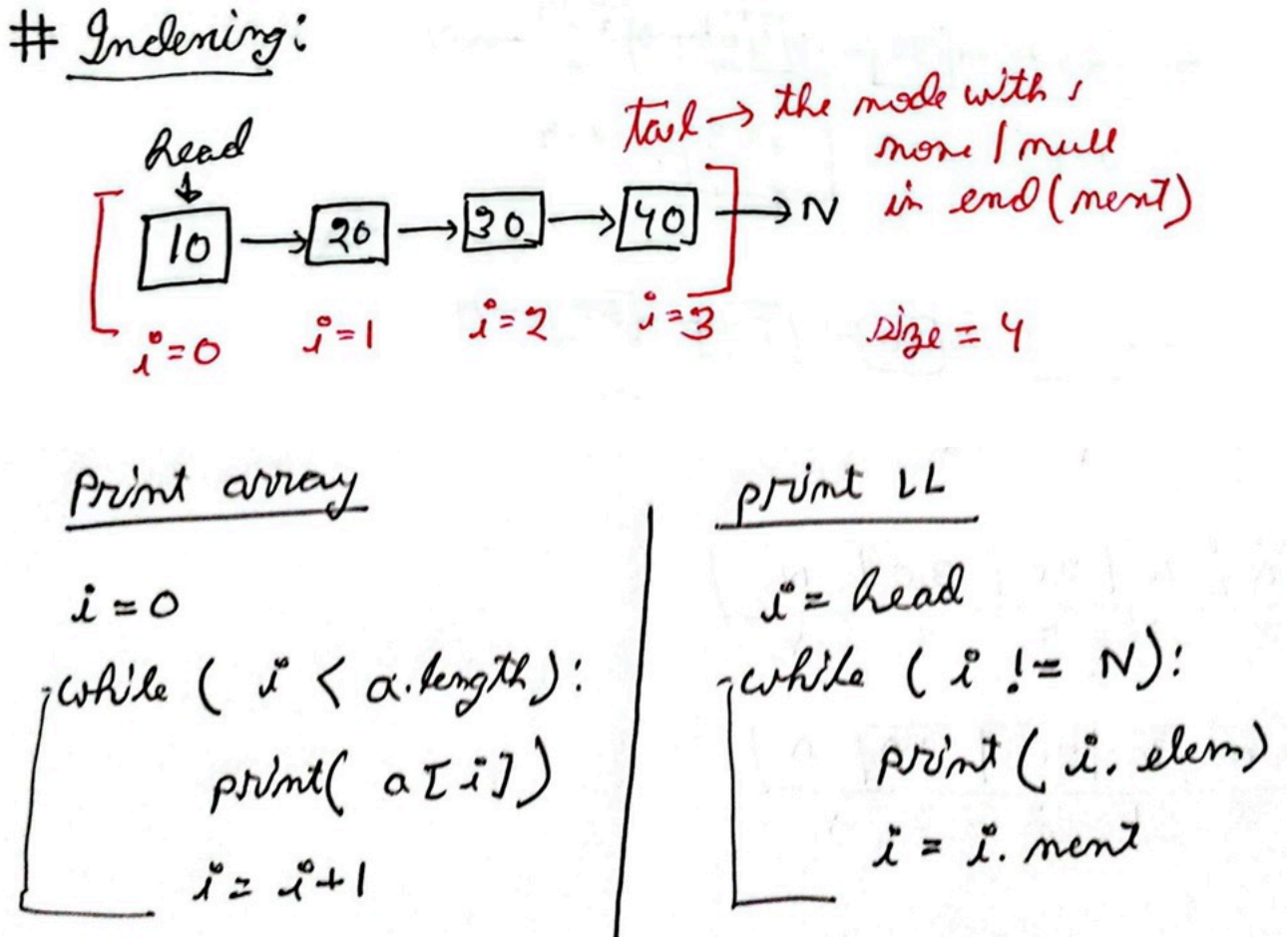
(Shabib said to cover these functions/topics)

1. printLL: for iterating and printing the elements of the LL
2. createList(arr): created a LL using an array- usually this code is used in the constructor of LL class
3. countNode/size(): get the total number of elements in a LL
4. appendLL: Insert at the end of the linked list
5. prependLL: Insert at the beginning of the linked list
6. indexOf(): Searching an element in the list, return the index (Often referred as Search).
7. contains(): Searching an element in the list, returns True/False
8. getElem(index) : gets the value from that index
9. getNode(index)/nodeAt(): gets the Node from that index, Often referred as nodeAt() in few materials/content.
10. setNode(index,new_elem): sets/updates the value from that index Searching an element in the list: IndexOf
11. insertAt(): Inserting an element into a list
12. removeAt(): Removing an element from a list
13. copyList()
14. Reversing a list (Reversal_OutOf_Place, Reversal_In_Place)
15. rotateLeft(): Rotating a list left by 1 position
16. rotateRight(): Rotating a list right by 1 position
17. rotateLeftMul(): Rotating a list left by k position
18. rotateRightMul(): Rotating a list right by k positions

## 1. Iteration/ Print linked list

Iteration is one of the most important aspects of every data structure. We will use an iterative approach to traverse a linked list. The important thing is that to do this traversal we are using a variable (temp in this case) to store the location of node and so we are updating this temp for every iteration.

- Time Complexity O(n)
- Space Complexity O(1)
  - the function requires a single pointer (current) to traverse the linked list and
  - an integer variable (count) to keep track of the number of nodes.
  - Both of these take up a **constant amount of memory**.



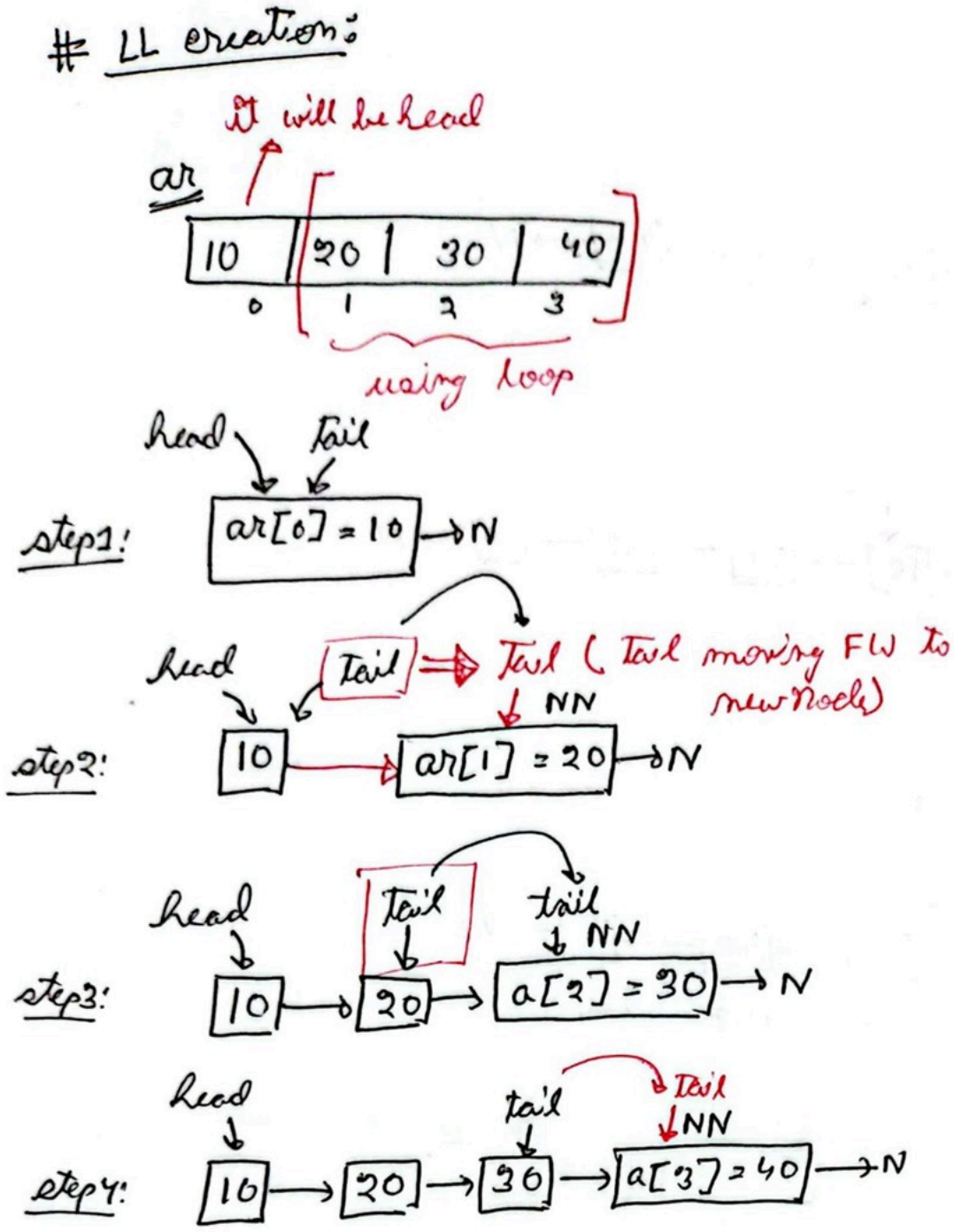

```
def printLL(head):
    temp = head # i=0
    while temp!=None: # while i< len:
        print(temp.elem, end=' -> ') # print(a[i])
        temp = temp.next # i = i+1, i = i.next
    print() # since using end, for going to the next line after end usage

printLL(head)
```

```
10 -> 20 -> 30 -> 40 ->
```

```
// Method to print the linked list
public void printLL(Node head) {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.elem + " -> ");
        temp = temp.next;
    }
    System.out.println();
}
```

## 2. Array to Linked List : creating a LL from an array

In the example code below, we are taking an array and converting it to a Linked List. To keep things easier, we are using the tail variable so that we do not need to go to the end of the list every time we want to add a new item.




```
import numpy as np

def createList(arr):
    if len(arr)==0:  # Check if the array is empty
        return None  # Return None for an empty list

    head = Node(arr[0]) # step1

    tail = head# step1
    for i in range(1, len(arr)):
        newNode = Node(arr[i]) #NN
        tail.next = newNode
        tail = newNode # tail = tail.next--Both are fine
```

```
    return head # returning the head of the newly created LL


arr = np.array([10, 20, 30, 40])
head2 = createList(arr)

# printing the newly created linked list
printLL(head2)
```

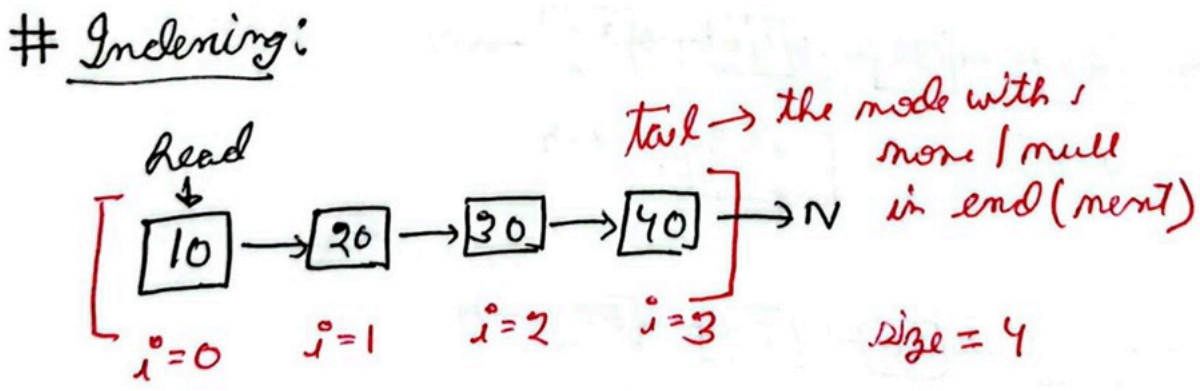
```
10 -> 20 -> 30 -> 40 ->
```

```
// Method to create a linked list from an array
public void createList(int[] arr) {
    if (arr.length == 0) {  // Check if the array is empty
        return null;  // Return null for an empty list
    }
    else{
        head = new Node(arr[0]); // create the head node
        Node tail = head; // start with the head as the tail
        for (int i = 1; i < arr.length; i++) {
            Node newNode = new Node(arr[i]); // create a new node
            tail.next = newNode; // attach the new node to the tail
            tail = newNode; // move the tail to the new node
        }
    }
}
```

## 3. countNode/size(): get the total number of elements in a LL

In order to count the number of nodes present in a linked list, we just need to count how many times the iteration process is running during a traversal.

- Time Complexity O(n)
- Space Complexity O(1)
    - the function requires a single pointer (current) to traverse the linked list and
    - an integer variable (count) to keep track of the number of nodes.
    - Both of these take up a **constant amount of memory**.



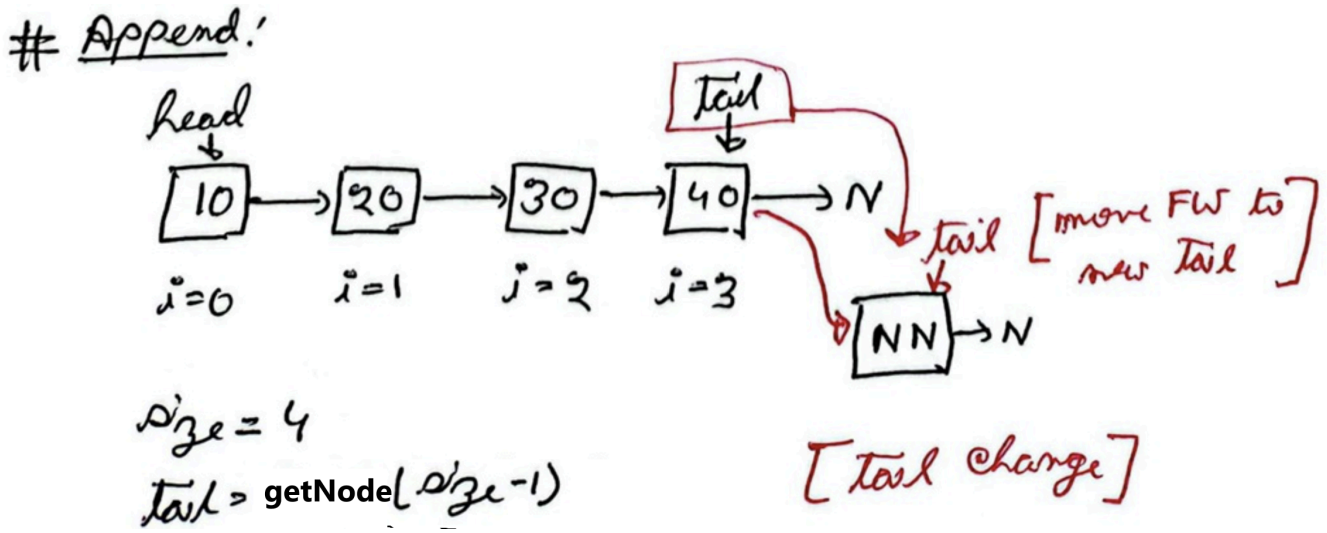

```
def countNode(head): # size
    if head == None:  # If the list is empty, return 0
        return 0

    current_node = head # i = 0 # here, current node is a temporary variable
    count = 0
    while current_node != None: # while i<len:
```

```
            current_node = current_node.next # i = i + 1
            count += 1
        return count

    arr = np.array([10, 20, 30, 40, 50])
    head3 = createList(arr)

    print("Linked List:")
    printLL(head3)

    # Count nodes in the linked list
    node_count = countNode(head3)
    print(f"Number of nodes in the linked list: {node_count}")

    print('-'*50)
    arr = np.array([10, 20])
    head3 = createList(arr)

    print("Linked List:")
    printLL(head3)

    # Count nodes in the linked list
    node_count = countNode(head3)
    print(f"Number of nodes in the linked list: {node_count}")


    print('-'*50)
    arr = np.array([])
    head3 = createList(arr)

    print("Linked List:")
    printLL(head3)

    # Count nodes in the linked list
    node_count = countNode(head3)
    print(f"Number of nodes in the linked list: {node_count}")
```

```
Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
Number of nodes in the linked list: 5
--------------------------------------------------
Linked List:
10 -> 20 ->
Number of nodes in the linked list: 2
--------------------------------------------------
Linked List:

Number of nodes in the linked list: 0
```

```
public int get_size(Node head) {
    if (head == null) {  // If the list is empty, return 0
        return 0;
    }

    Node currentNode = head;  // Temporary variable to traverse the list
    int count = 0;

    while (currentNode != null) {  // While the current node is not null
        currentNode = currentNode.next;  // Move to the next node
        count++;  // Increment the count
    }

    return count;  // Return the total number of nodes
}
```

## 4. appendLL: Insert at the end of the linked list




```
def get_tail(head):
    temp = head
    while temp.next!=None: # iterating till the next == None/Null. For singly only tail has None in next
        temp = temp.next
    return temp

arr2 = np.array([100, 200, 300, 400])
head22 = createList(arr2)
printLL(head22)
tail2 = get_tail(head22)
print("Element of tail:", tail2.elem)
```

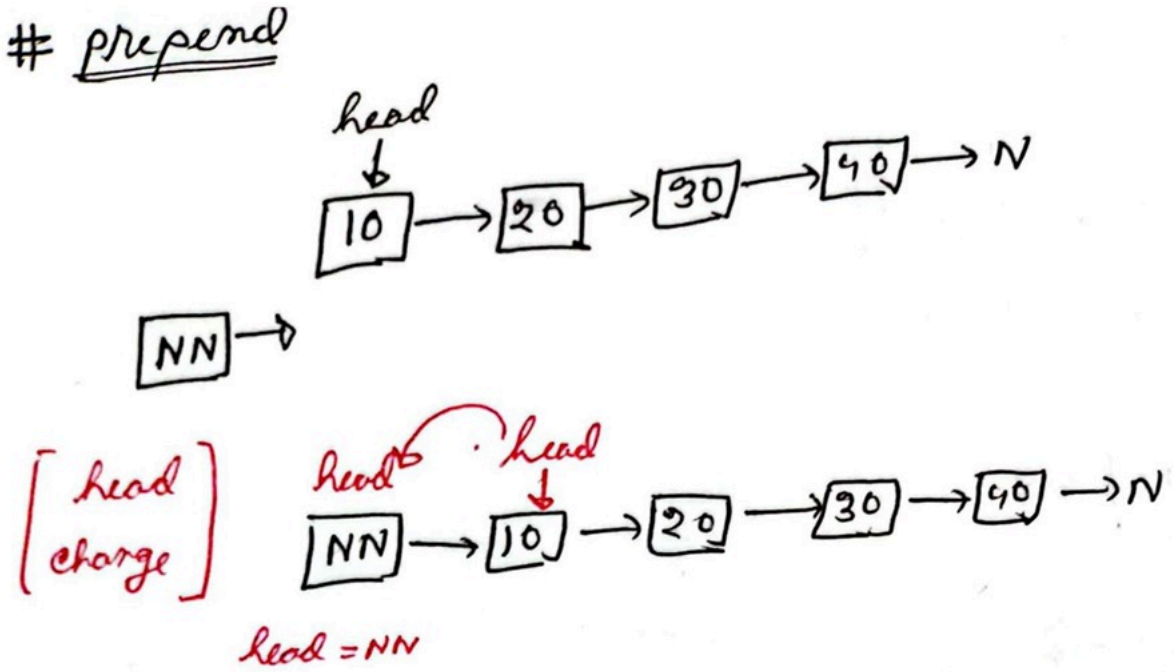
```
100 -> 200 -> 300 -> 400 ->
Element of tail: 400
```

```
# Insert at the end of the linked list
def appendLL(head, data):
    newNode = Node(data)

    if head == None:
        head = newNode
        return

    tail = get_tail(head3)
    print("Element of tail:", tail.elem)
    tail.next = newNode # linking the new node at the end of tail

    #tail = newNode # The change will be to the local variable and wont be saved.
    #So we need to return the updated tail/new tail
    #--In this case, since tail value can be read by iterating from head--We dont need to return but
    # But if we want, we could return the updated tail value

arr = np.array([10, 20, 30, 40])
head3 = createList(arr)

# Append new elements and update the tail each time
print('-'*50)
appendLL(head3, 100)
print("After appending 100:")
printLL(head3)

print('-'*50)
appendLL(head3, -80)
print("After appending -80:")
printLL(head3)
```

```
--------------------------------------------------
Element of tail: 40
After appending 100:
10 -> 20 -> 30 -> 40 -> 100 ->
--------------------------------------------------
Element of tail: 100
After appending -80:
10 -> 20 -> 30 -> 40 -> 100 -> -80 ->
```

```
public void appendLL(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        return;
    }
    Node tail = getTail();
    tail.next = newNode;
}
```

## 5. prependLL: Insert at the beginning of the linked list




```
def prependLL(head, elem):
    # Insert at the beginning of the linked list
    newNode = Node(elem)
    newNode.next = head
    #head = newNode # It will only update the local variable--not the original head3.
    #So, we need to return the updated head
    return newNode  # Return the new head of the linked list

arr = np.array([10, 20, 30, 40])
head3 = createList(arr)

print("Before prepend:")
printLL(head3)

# Prepend new elements
head3 = prependLL(head3, 5)
print("After prepending 5:")
printLL(head3)

head3 = prependLL(head3, -10)
print("After prepending -10:")
printLL(head3)
```

```
Before prepend:
10 -> 20 -> 30 -> 40 ->
```

```
After prepending 5:
5 -> 10 -> 20 -> 30 -> 40 ->
After prepending -10:
-10 -> 5 -> 10 -> 20 -> 30 -> 40 ->
```

```
public void prependLL(int elem) {
    Node newNode = new Node(elem);
    newNode.next = head;
    head = newNode;
}
```

## Searches for an element in the list

Searching for an element in a list can be done by sequentially searching through the list.

- Time Complexity O(n)
- Space Complexity O(1)

There are two typical variants: return the index of the given element (indexOf()), or return true if the element exists (contains()).

### 6.IndexOf(): Searches for an element in the list, return the index

[ ] ↳ 3 cells hidden

## 7. contains(): Searches for an element in the list, returns True/False

- Time Complexity O(n)
- Space Complexity O(1)

[ ] ↳ 3 cells hidden

## Variations of same code

- **getElem(index, data)** : gets the **value** from that index
- **getNode(index, data)**: gets the **Node** from that index
- **setNode(index, data, new_elem)**: sets/updates the value from that index

### 8. getElem(): Retrieving an element (value) from an index

- Time Complexity O(n)
- Space Complexity O(1)

[ ] ↳ 3 cells hidden

## 9. getNode: Getting the node of a particular index

- Often referred as nodeAt() in few materials/content.
- Time Complexity O(n)
- Space Complexity O(1)

[ ] ↳ 3 cells hidden

## 10. setNode():

By tuning the get function, we can also update the element of a linked list using an index.

- Time Complexity O(n)
- Space Complexity O(1)

[ ] ↳ 5 cells hidden

## 11. insertAt(): Inserting an element into a list

- Time Complexity O(n)
- Space Complexity O(1)

There are three places in a list where we can insert a new element:

- In the beginning ("head" changes)
- Except the head
  - in the middle
  - at the end

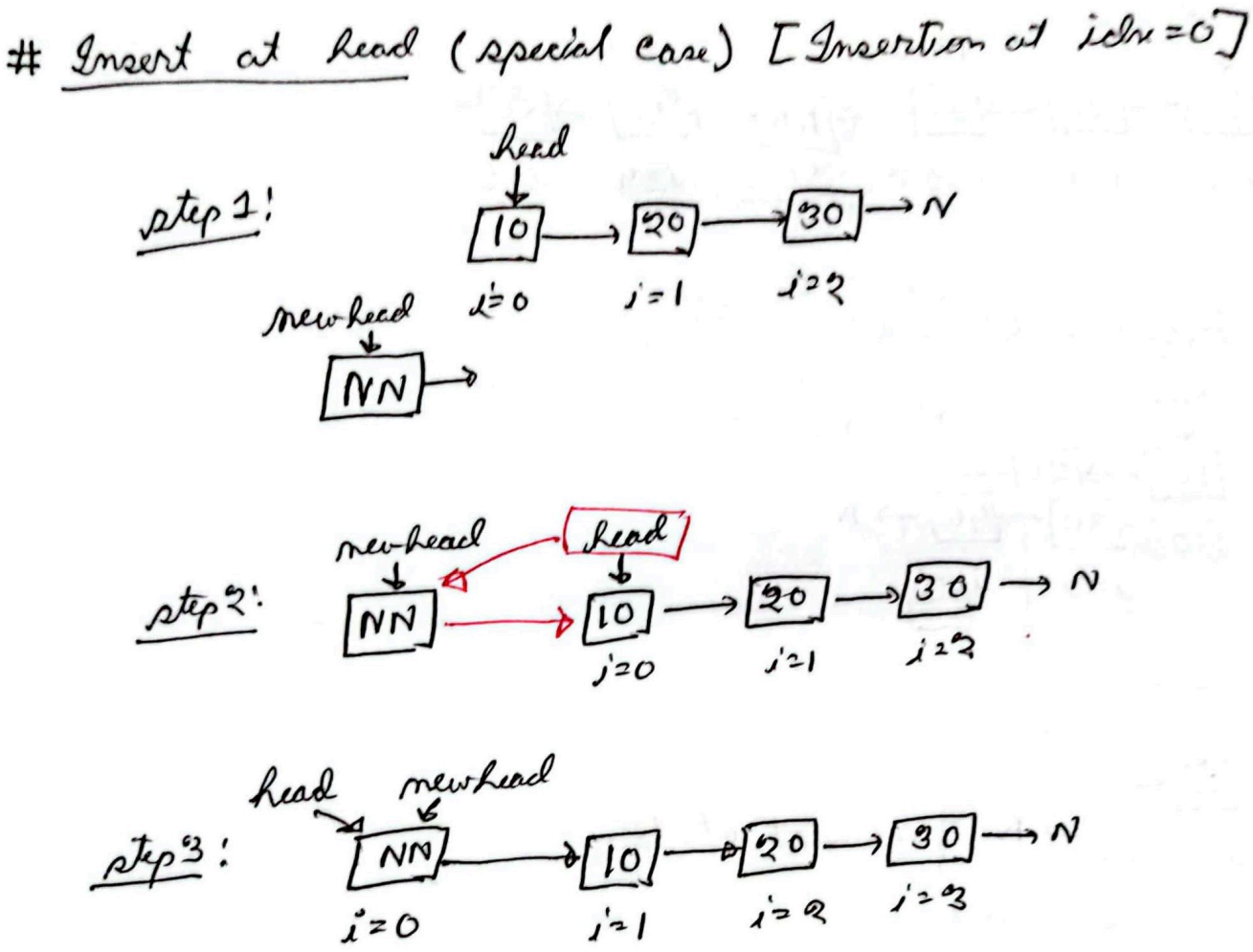
Important terms:

- **predecessor**: The node before the selected node
- **successor**: The node after the selected node

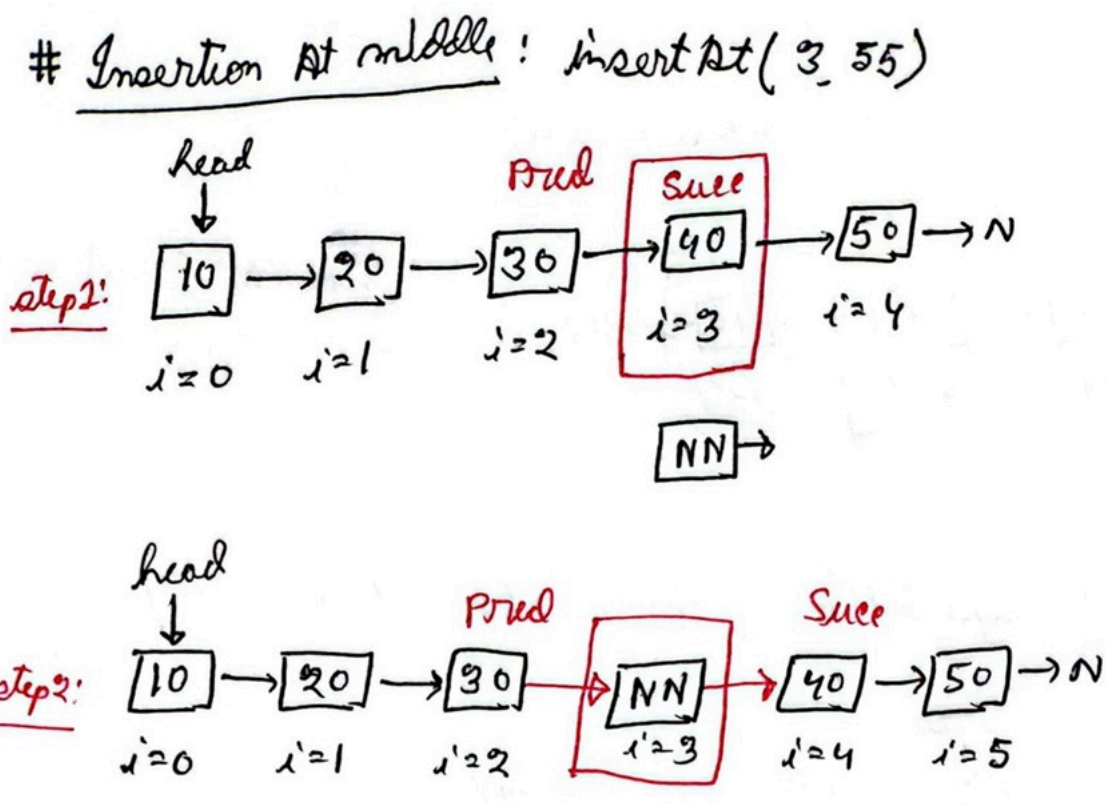
**To insert a new node in the list, you need the reference to the predecessor to link in the new node.**

- There is one "special" case however — inserting a new node at the beginning of the list because the head node does not have a predecessor. Inserting, in the beginning, has an important side effect — it changes the **head reference**!
- Inserting in the middle or at the end is the same — we first find the predecessor node and link in the new node with the given element.
- To find the specific node, we can take the help of the getNode() function mentioned above.

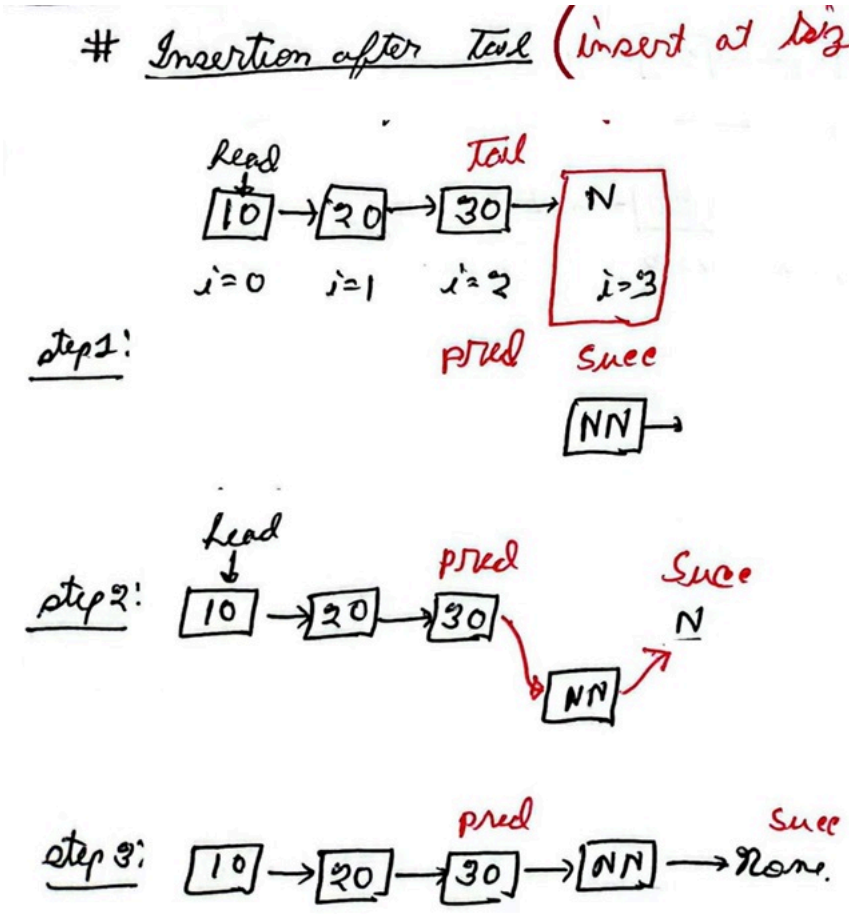
### 1) Illustration of insertion of element at index 0 [Special case]



2) Illustration of insertion of element at index 3 (Middle) [generic case]




3) Illustration of insertion of element at index size (after tail, size) [generic] insert (3, 50)



```
def insertAt(head, idx, elem):
    size = countNode(head)

    if idx<0 or idx>size: # Invalid index
        print("Insertion Failed, Not valid index")
        return None
    elif idx == 0:  # Insertion at head---Prepend
        newNode = Node(elem)
        newNode.next = head  # Link the head
        head = newNode  # Update the head
        return head
    else: # insertion in middle or end
        newNode = Node(elem)

        predecessor = getNode(head, idx-1)
        #print("predecessor", predecessor )
        successor = predecessor.next # successor = getNode(head, idx)--Both are fine

        newNode.next = successor  # Link the successor
        predecessor.next = newNode  # Link the predecessor
        return head


import numpy as np
              # 0   1   2   3   4
arr = np.array([10, 20, 30, 40, 50])
head3 = createList(arr)

print("Original Linked List:")
printLL(head3)
print('-'*50)

# Insert elements at various positions
head3 = insertAt(head3, 0, 5)  # Insert at the head
print("After inserting 5 at index 0 (head):")
printLL(head3)
print('-'*50)

head3 = insertAt(head3, 3, 25)  # Insert in the middle
print("After inserting 25 at index 3 (middle):")
printLL(head3)
print('-'*50)

head3 = insertAt(head3, 7, 60)  # Insert at the end
print("After inserting 60 at index 7 (after tail, size):")
printLL(head3)

# Attempt to insert at an invalid index
head3 = insertAt(head3, 10, 100)
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
--------------------------------------------------
After inserting 5 at index 0 (head):
5 -> 10 -> 20 -> 30 -> 40 -> 50 ->
--------------------------------------------------
After inserting 25 at index 3 (middle):
5 -> 10 -> 20 -> 25 -> 30 -> 40 -> 50 ->
--------------------------------------------------
After inserting 60 at index 7 (after tail, size):
5 -> 10 -> 20 -> 25 -> 30 -> 40 -> 50 -> 60 ->
Insertion Failed, Not valid index
```

```
public void insertAt(int idx, int elem) {
    int size = get_size();

    if (idx < 0 || idx > size) { // Invalid index
        System.out.println("Insertion Failed, Not valid index");
        return;
    } else if (idx == 0) {  // Insertion at head
        Node newNode = new Node(elem);
        newNode.next = head;
        head = newNode;
    } else { // Insertion in middle or end
        Node newNode = new Node(elem);
        Node predecessor = nodeAt(idx - 1);
        Node successor = predecessor.next;

        newNode.next = successor;
        predecessor.next = newNode;
    }
}
```

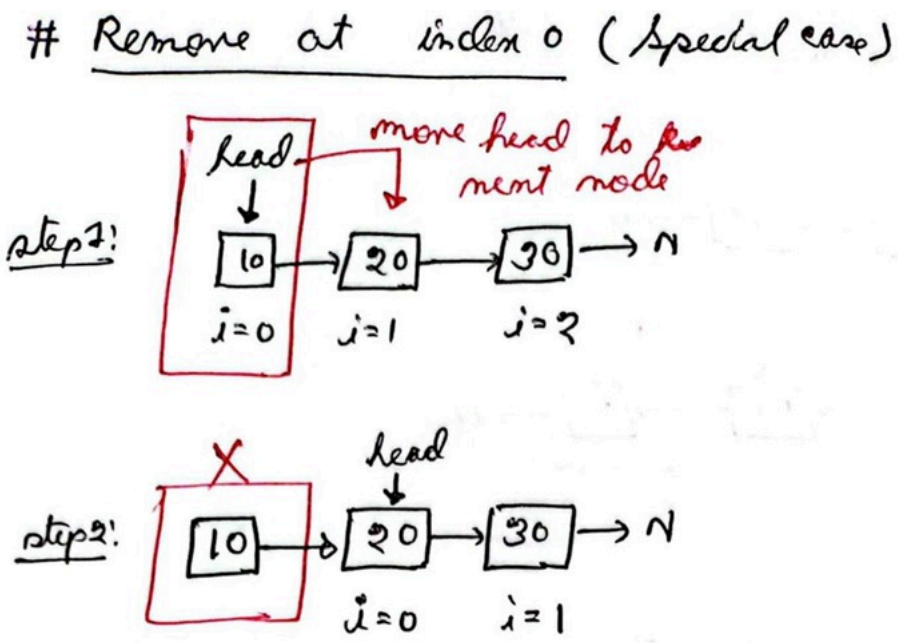
## 12. removeAt(): Removing an element from a list

- Time Complexity O(n)
- Space Complexity O(1)

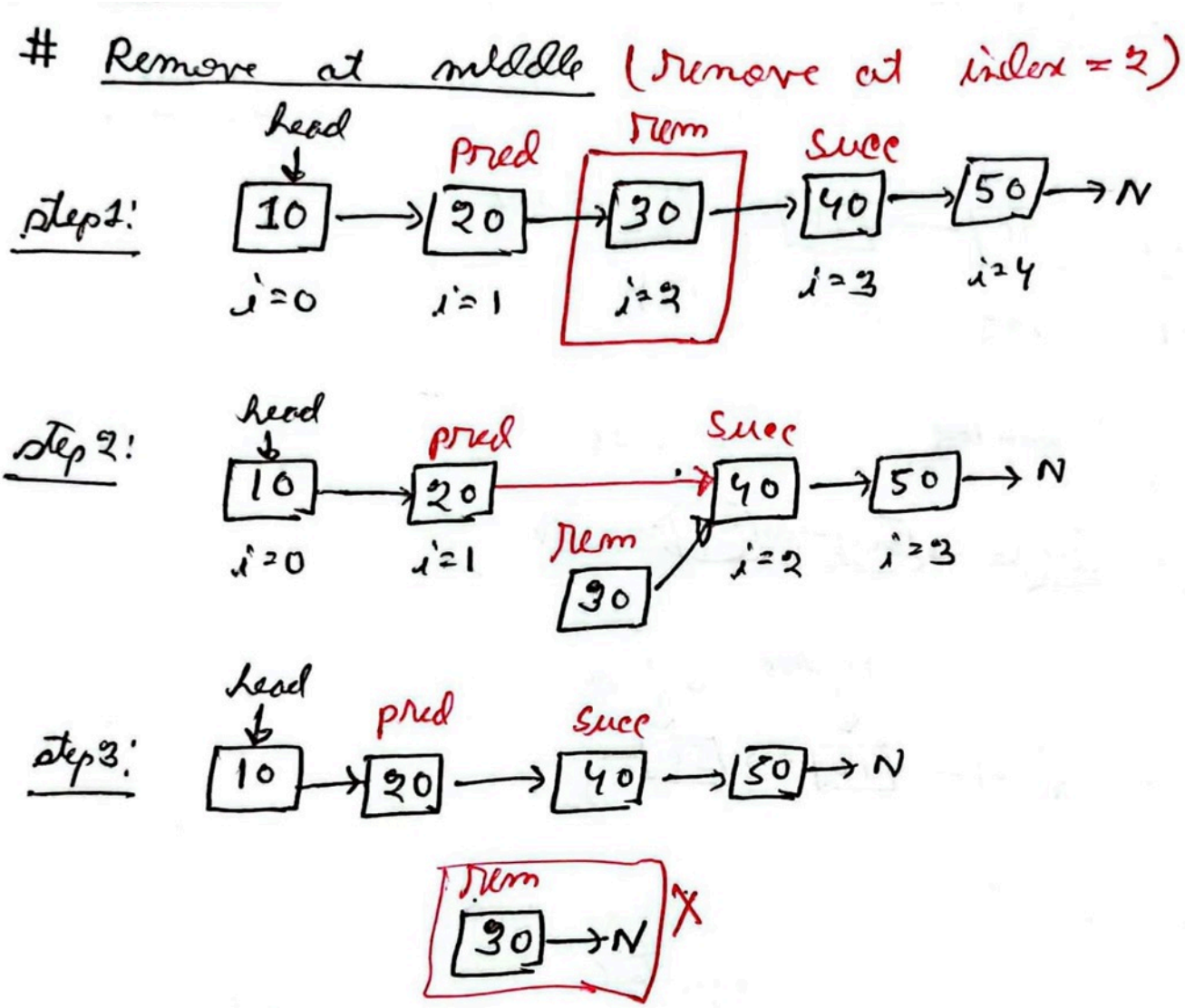
Removing an element from the list is done by removing the node that contains the element.

- Just like inserting a new node in a list, **removing requires that you have the reference to the predecessor node.**
- And just like in insertion, removing the 1st node in the list is a **"special" case** — it does not have a predecessor, and removing it has the side-effect of changing the head. You can try to make this more efficient too!!!

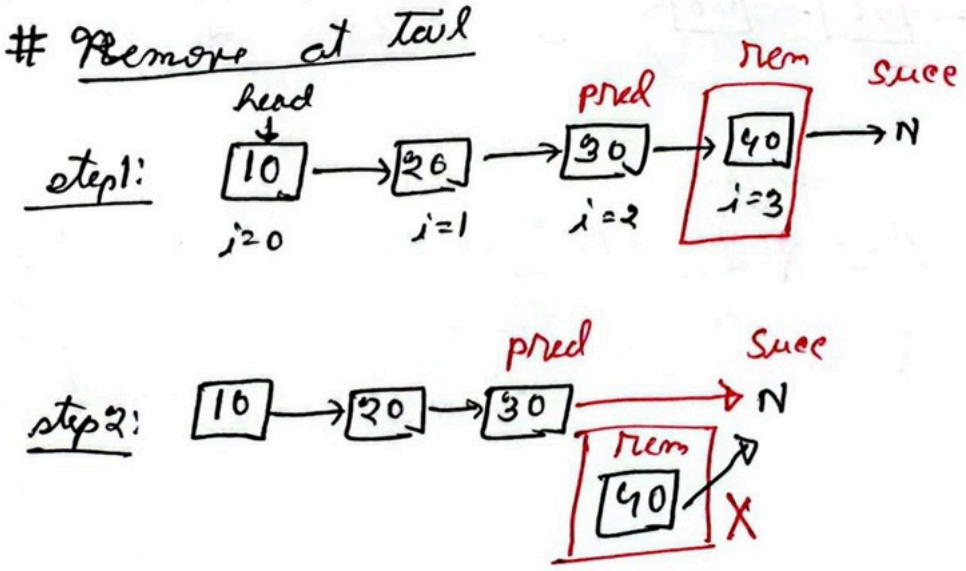
### 1) Illustration of removal of element at index 0 (head) [Special case]




### 2) Illustration of removal of element at index 2 (middle) [generic case]




3) Illustration of removal of element at index 3 (tail) [generic]



```
def removeAt(head, idx):
    size = countNode(head)
    if idx<0 or idx>size: # Invalid index
        print("deletion Failed, Not valid index")
    elif idx == 0:  # deletion at head
        head = head.next  # Update the head to the next element in the list
    else: # # Deletion in the middle or at the end
        predecessor = getNode(head, idx-1)
        remNode = predecessor.next # remNode = getNode(head, idx) #---Both fine
        successor = remNode.next #successor = getNode(head, idx+1), successor = predecessor.next.next#---Both fine

        predecessor.next = successor  # skipping the rem element and making new link

        # Helping the Garbage collector--optional
        remNode.next = None
        remNode.elem = None

    return head

# Example usage

               # 0   1   2   3   4
arr = np.array([10, 20, 30, 40, 50])
head3 = createList(arr)

print("Original Linked List:")
printLL(head3)

# Remove elements at various positions
head3 = removeAt(head3, 0)  # Remove head
print("After removing element at index 0 (head): remove 10")
printLL(head3)

head3 = removeAt(head3, 2)  # Remove element at index 2
print("After removing element at index 2 (middle): remove 40")
printLL(head3)

head3 = removeAt(head3, 2)  # Remove element at index 2
print("After removing element at index 2 (tail), remove: 50:")
printLL(head3)

### Attempt to remove at an invalid index--
#Remove the comment at line 3 to see the error
head3 = removeAt(head3, 3)  # # Invalid index--Currently there are only 3 elements. So indexing 0 to 2
print("After removing element at index 3:")
printLL(head3)
```

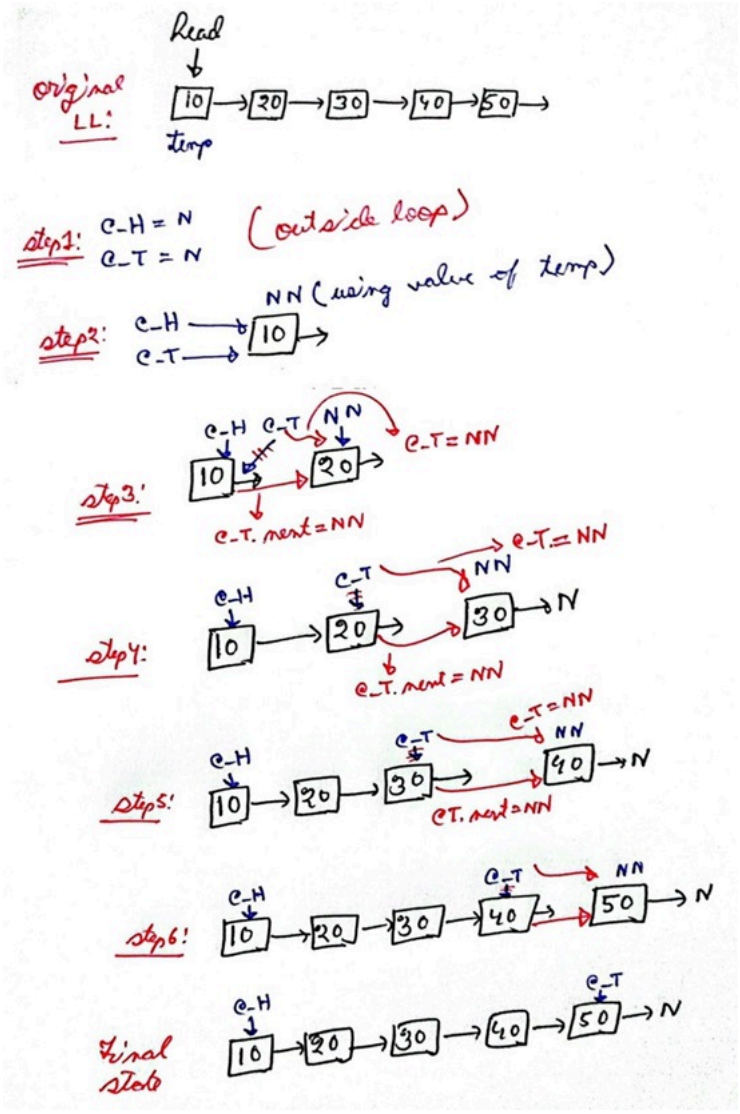
```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
After removing element at index 0 (head): remove 10
20 -> 30 -> 40 -> 50 ->
After removing element at index 2 (middle): remove 40
20 -> 30 -> 50 ->
After removing element at index 2 (tail), remove: 50:
20 -> 30 ->
deletion Failed, Not valid index
After removing element at index 3:
20 -> 30 ->
```

```
public void removeAt(int idx) {
    int size = get_size();
    if (idx < 0 || idx >= size) {
        System.out.println("Invalid Index");
        return;
    }

    // If removing the head node
    if (idx == 0) {
        head = head.next;
        return;
    }

    // Find the predecessor (node at idx-1)
    Node predecessor = nodeAt(idx - 1);
    Node remNode = predecessor.next;
    Node successor = remNode.next;

    // Link predecessor to successor, effectively removing remNode
    predecessor.next = successor;

    // Optional: Help garbage collector by clearing remNode's next
    remNode.next = null;
}
```

## 13. Copying a list:-- Copy by each element/value

Copying the elements of a source list to a destination list is simply a matter of iterating over the elements of the source list, and inserting these elements at the end of the destination list.

- Time Complexity O(n)
- Space Complexity O(n) [a new linked list with n nodes is created, requiring additional memory proportional to n.]




```
def copyList(head):
    if head == None:  # If the list is empty, return None
        return None

    copy_head = None # will return it
    copy_tail = None ## append

    temp = head # will be used for iterating the LL with original elements, i =0
    while temp!= None: # Loop until the end of the original list

        newNode = Node(temp.elem)
        if copy_head == None:  #If this is the first node being added--Only 1 Node--so head and tail are same
            copy_head = newNode
            copy_tail = newNode # copy_tail = copy_head #----Both fine
        else: # # Otherwise, append to the tail of the copied list
            copy_tail.next = newNode
            copy_tail = newNode # Forwaring tail in the new list

        temp = temp.next # Move to the next node in the original list, i = i +1
    return copy_head

arr = np.array([10, 20, 30, 40, 50])
head_original = createList(arr)

print("Original Linked List:")
printLL(head_original)

# Copy the linked list
head_copied = copyList(head_original)

print("Copied Linked List:")
```

```
printLL(head_copied)

# Print memory locations of the heads
print(f"Memory location of original head: {id(head_original)}")
print(f"Memory location of copied head: {id(head_copied)}")
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
Copied Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
Memory location of original head: 134965876142928
Memory location of copied head: 134965876143888
```

```
public Node copyList() {
    if (head == null) {  // If the list is empty, return null
        return null;
    }

    Node copyHead = null;
    Node copyTail = null;
    Node temp = head;

    // Traverse through the original list and create nodes in the copied list
    while (temp != null) {
        Node newNode = new Node(temp.elem);  // Create a new node with the same value
        if (copyHead == null) {  // For the first node, initialize both head and tail
            copyHead = newNode;
            copyTail = newNode;
        } else {  // Append new nodes to the end of the copied list
            copyTail.next = newNode;
            copyTail = copyTail.next;  // Move the tail to the newly added node
        }
        temp = temp.next;  // Move to the next node in the original list
    }

    return copyHead;  // Return the head of the copied list
}
```

## 14. Reversing a list:

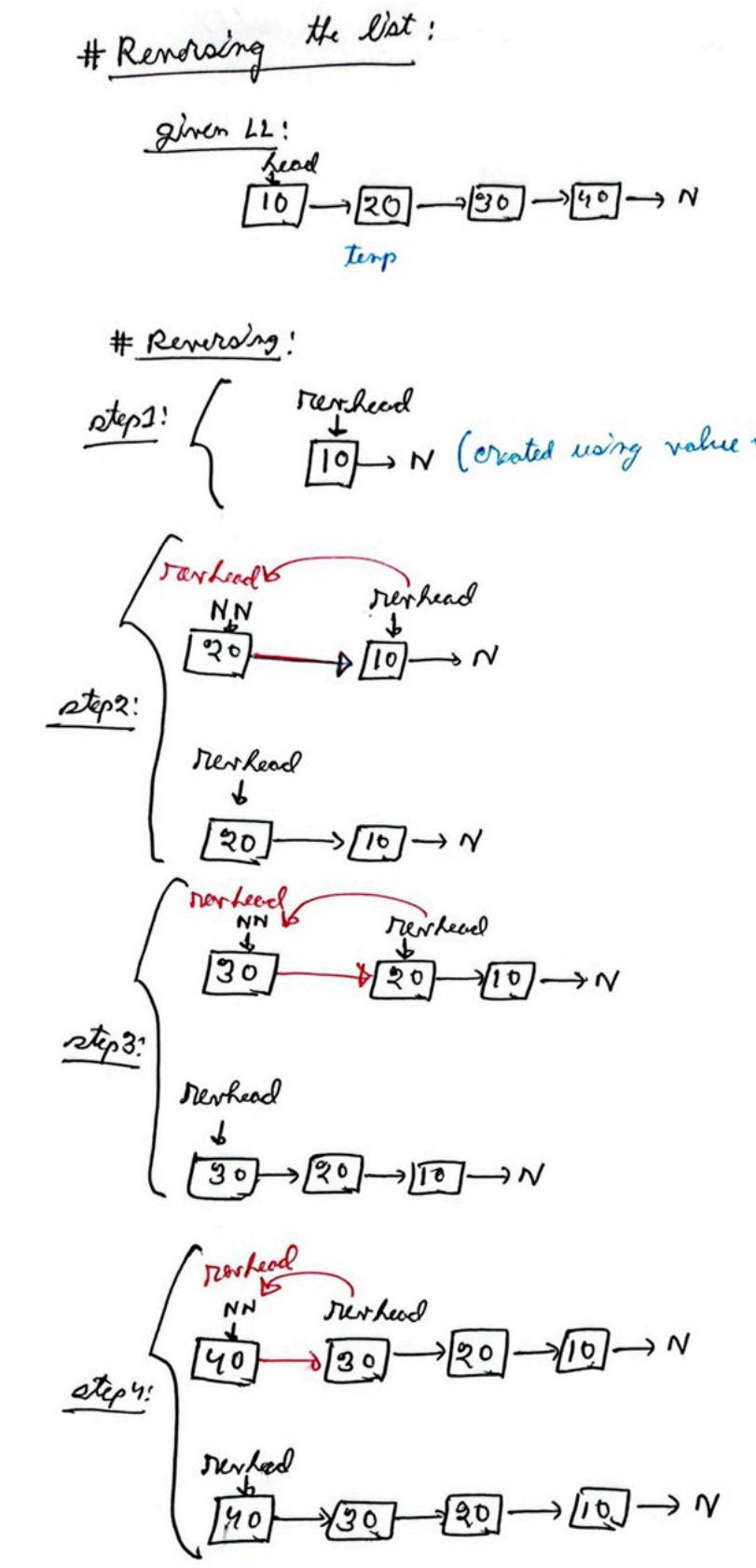
- reversalOutOfplace--new LL created
- reversalInplace-- no new LL, connection change

### reversalOutOfplace

Since a linked list does not support random access, it is difficult to reverse a list in place without changing the head reference.

We'll create a new list with its own head reference, and copy the elements in the reverse order. This method does not modify the original list, so we can call it an out-of-place method.

- Time Complexity O(n)
- Space Complexity O(n)




```
import numpy as np
def reverse_out_of_place(head):
    if head == None:  # If the list is empty, return None
        return None

    rev_head = Node(head.elem) # Create the new head for the reversed list # i=0

    temp = head.next# i=1
```

```
    while temp != None: # Loop until the end of the original list

        #---prepend code
        newNode = Node(temp.elem) # Create a new node with the current value
        newNode.next = rev_head # link the revHead after the new element [Pre-pend]
        rev_head = newNode  # Update new_head to point to the newly created node
        #-------------------

        temp = temp.next # Move to the next node in the original list # i = i + 1
    return rev_head

arr = np.array([10, 20, 30, 40, 50])
head_original = createList(arr)

print("Original Linked List:")
printLL(head_original)

# Reverse the linked list out of place
head_reversed = reverse_out_of_place(head_original)

print("Reversed Linked List (Out of Place):")
printLL(head_reversed)

# Verify that the original list remains unchanged
print("Original Linked List After Reversing:")
printLL(head_original)
```

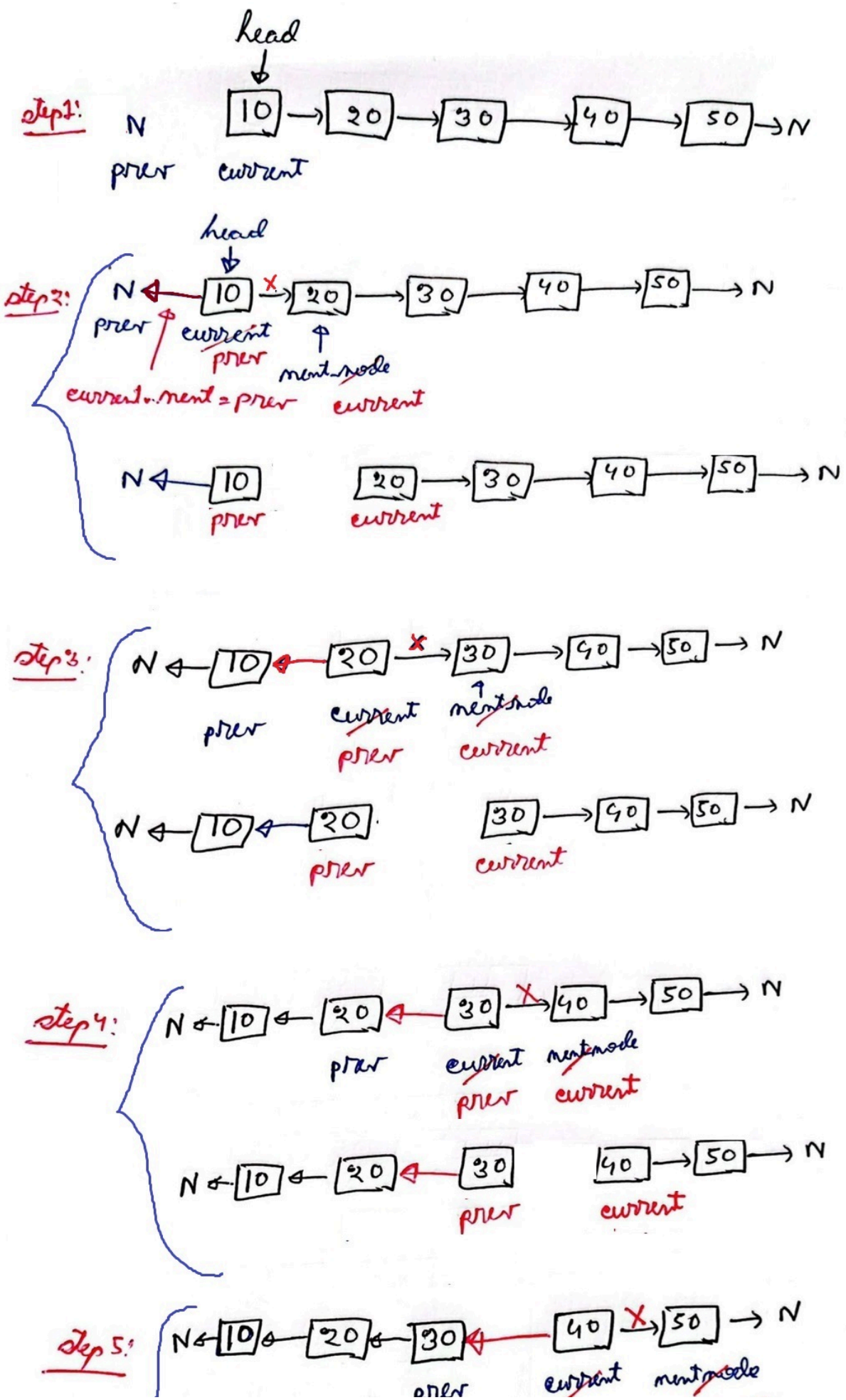
```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
Reversed Linked List (Out of Place):
50 -> 40 -> 30 -> 20 -> 10 ->
Original Linked List After Reversing:
10 -> 20 -> 30 -> 40 -> 50 ->
```

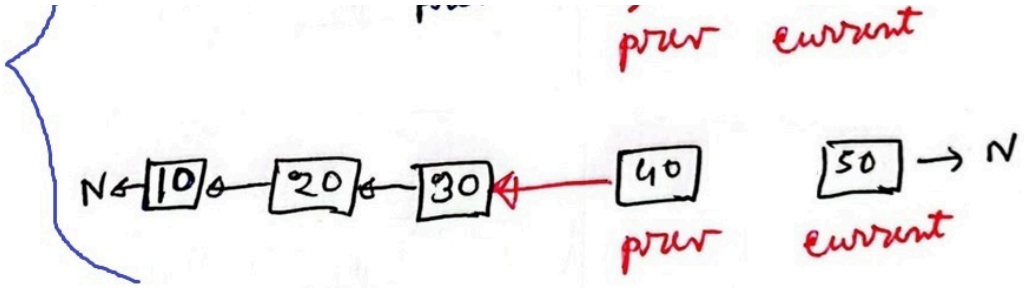
```
public Node reverseOutOfPlace() {
    if (head == null) {  // If the list is empty, return null
        return null;
    }

    Node revHead = new Node(head.elem);  // Create the new head for the reversed list
    Node temp = head.next;

    // Traverse the original list and add each element to the front of the reversed list
    while (temp != null) {
        Node newNode = new Node(temp.elem);  // Create a new node with the current element
        newNode.next = revHead;  // Link the new node to the front of the reversed list
        revHead = newNode;  // Update the head of the reversed list
        temp = temp.next;  // Move to the next node in the original list
    }

    return revHead;  // Return the head of the reversed list
}
```

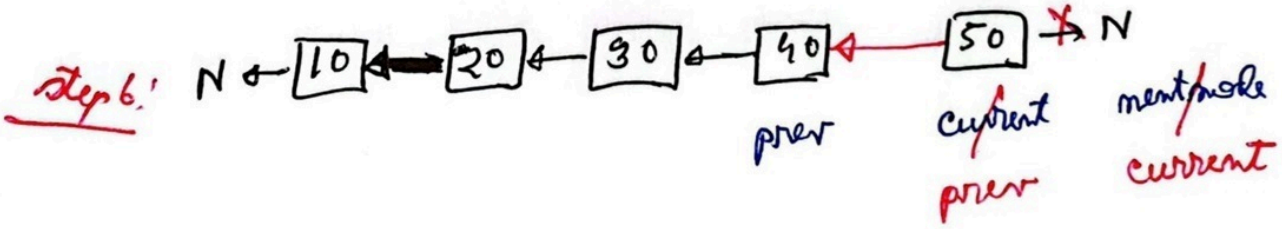
## reversalInplace

Re-order the links instead of copying the nodes! That would of course change the original list and would have a new head reference (which is the reference to the tail node in the original list).

- Time Complexity O(n)
- Space Complexity O(1)


head
step1:
N
prev
current
10
20
30
40
50
N
head
step2:
N
10
X
20
30
40
50
N
prev
current
prev
next_node
current
current.next = prev
current
N
10
prev
20
30
40
50
N
current
step3:
N
10
20
X
30
40
50
N
prev
current
prev
next node
current
N
10
20
prev
30
40
50
N
current
step4:
N
10
20
30
X
40
50
N
prev
current
prev
next node
current
N
10
20
30
prev
40
50
N
current
step 5:
N
10
20
30
40
X
50
N
prev
current
next node




```
def reverseInPlace(head):
    # STEP01
    prev = None # for changing the direction of link
    current = head #---to run the loop

    while current!= None:
        next_node = current.next  # Store next node
        current.next = prev  # Reverse the link

        prev = current  # Move prev forward
        current = next_node  # Move current forward---differnt

    return prev  # New head of the reversed list

# Driver Code
arr = np.array([10, 20, 30, 40, 50])
head_original = createList(arr)

print("Original Linked List:")
printLL(head_original)

# Reverse the linked list in place
head_reversed = reverseInPlace(head_original)

print("Reversed Linked List (In Place):")
printLL(head_reversed)
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
Reversed Linked List (In Place):
50 -> 40 -> 30 -> 20 -> 10 ->
```

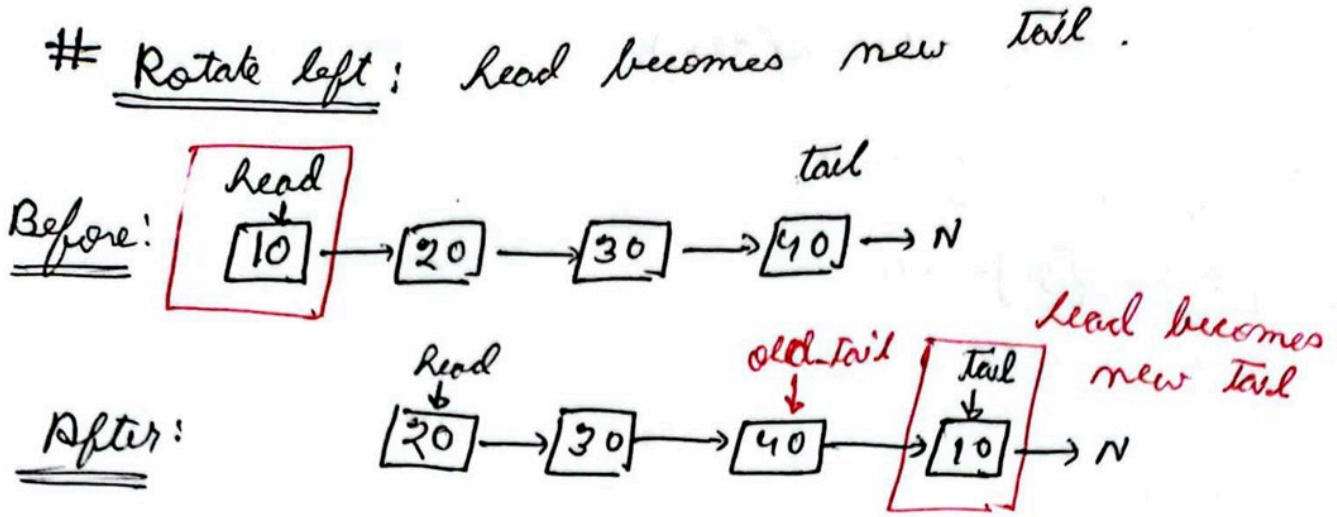
```
// Method to reverse the linked list in place
public Node reverseInPlace() {
    Node prev = null;  // Previous node starts as null
    Node current = head;  // Current node starts as the head of the list

    while (current != null) {  // Traverse until the end of the list
        Node nextNode = current.next;  // Store the next node
        current.next = prev;  // Reverse the current node's link
        prev = current;  // Move prev forward
        current = nextNode;  // Move current forward
    }

    return prev;  // New head of the reversed list
}
```

## Important to remember

- **rotateLeft:** Head becomes the new tail
- **rotateRight:** Tail becomes the new head

## 15. rotateLeft(): Rotating a list left by 1 position [Optional]

Rotating a list left is much simpler than rotating an array — the 2nd node becomes the new head, and the 1st becomes the new tail node. We don't have to actually move the elements, it's just a matter of rearranging a few links.




```
def rotateLeft1(head):
    # Using getNode() and countNode()
    if head==None or head.next ==None:  # Check for empty or single-node list
        return head

    old_head = head # keeping a copy--it will be new tail
    head = head.next  # Assign the new head (2nd element is the new head now)

    size = countNode(head)  # Count nodes starting from the new head
    tail = getNode(head, size - 1)  # Last node in the rotated list # can be done with getTail()
    tail.next = old_head  # Putting original head at the end of the tail

    old_head.next = None  # Original head is the new tail now. Because only tail has None

    return head

arr = np.array([10, 20, 30, 40, 50])
head3 = createList(arr)
```

```
print("Original Linked List:")
printLL(head3)

# Rotate the linked list to the left
head3 = rotateLeft1(head3)
print("After rotating left:")
printLL(head3)

# Rotate again to see the effect
head3 = rotateLeft1(head3)
print("After rotating left again:")
printLL(head3)
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
After rotating left:
20 -> 30 -> 40 -> 50 -> 10 ->
After rotating left again:
30 -> 40 -> 50 -> 10 -> 20 ->
```

```
// Method to rotate the linked list left
public void rotateLeft() {
    if (head == null || head.next == null) {  // Check for empty or single-node list
        return;
    }

    Node oldHead = head; // Keep a copy of the original head to be the new tail
    head = head.next;    // Move head to the next node (new head)

    Node tail = getNode(get_size() - 1); // Find the last node in the list
    tail.next = oldHead;    // Link the last node to the original head
    oldHead.next = null;    // Make the old head the new tail by setting next to null
}
```

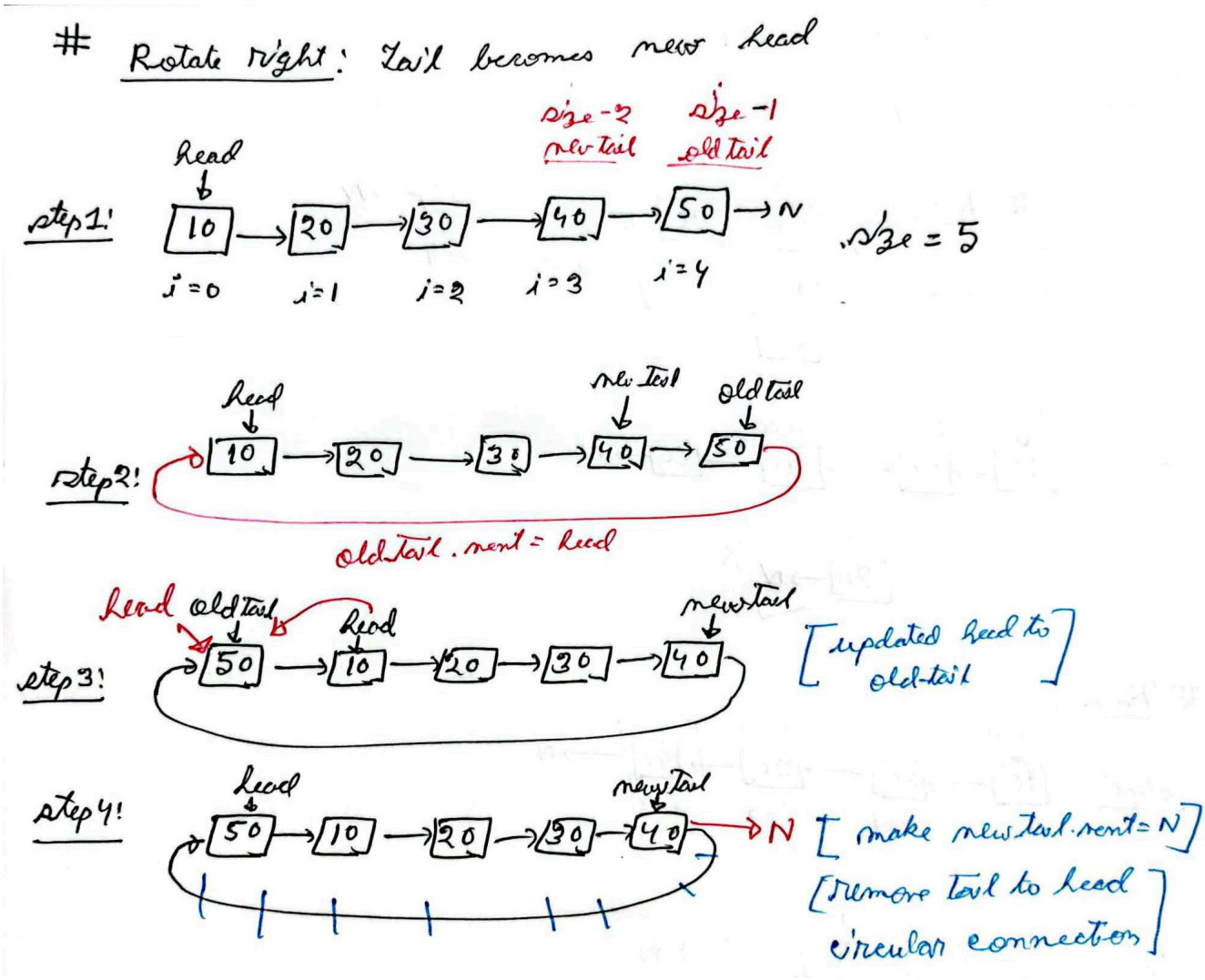
```
def rotateLeft2(head):
    # Using while()
    if head==None or head.next ==None:  # Check for empty or single-node list
        return head

    old_head = head # keeping a copy--it will be new tail
    head = head.next  # Assign the new head (2nd element is the new head now)

    tail = head
    while tail.next!= None: # getting the old tail
        tail = tail.next

    tail.next = old_head  # Putting original head at the end of the tail
    old_head.next = None  # Original head is the new tail now. Because only tail has None

    return head

arr = np.array([10, 20, 30, 40, 50])
head3 = createList(arr)

print("Original Linked List:")
printLL(head3)

# Rotate the linked list to the left
head3 = rotateLeft2(head3)
print("After rotating left:")
printLL(head3)

# Rotate again to see the effect
head3 = rotateLeft2(head3)
print("After rotating left again:")
printLL(head3)
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
After rotating left:
```

```
20 -> 30 -> 40 -> 50 -> 10 ->
After rotating left again:
30 -> 40 -> 50 -> 10 -> 20 ->
```

## 16. rotateRight(): Rotating a list right by 1 position [Optional]

Rotating a list right is almost the same as rotating left — the current last node becomes the new head and the current second last node becomes the last node.




```
def rotateRight1(head):
    # Using getNode() and countNode()
    if head==None or head.next ==None:  # Check for empty or single-node list
        return head

    size = countNode(head)  # Count nodes in the list
    new_tail = getNode(head, size - 2)  # 2nd last node of the list becomes the new tail
    old_tail = new_tail.next  # old_tail =getNode(head, size - 1)#--Both fine# Last node (size-1), which will be the new head

    old_tail.next = head  # Link the old tail to the head (putting the old tail before the head)
    head = old_tail  # Making the old tail the new head

    new_tail.next = None  # 2nd last node becomes the new tail. Only tail has None at end

    return head

arr = np.array([10, 20, 30, 40, 50])
head5 = createList(arr)

print("Original Linked List:")
printLL(head5)
```

```
# Rotate the linked list to the right
head5 = rotateRight1(head5)
print("After rotating right:")
printLL(head5)

# Rotate again to see the effect
head5 = rotateRight1(head5)
print("After rotating right again:")
printLL(head5)
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
After rotating right:
50 -> 10 -> 20 -> 30 -> 40 ->
After rotating right again:
40 -> 50 -> 10 -> 20 -> 30 ->
```

```
// Method to rotate the linked list right
public void rotateRight() {
    if (head == null || head.next == null) {  // Check for empty or single-node list
        return;
    }

    int size = get_size();
    Node newTail = getNode(size - 2); // Find the second-to-last node, which becomes the new tail
    Node oldTail = newTail.next;     // The last node, which will be the new head

    oldTail.next = head;     // Link the last node to the original head
    newTail.next = null;     // Set the new tail's next to null
    head = oldTail;          // Update head to the old tail (new head)
}
```

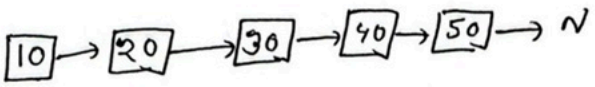
```
def rotateRight2(head):
    # Using while()
    if head==None or head.next ==None:  # Check for empty or single-node list
        return head

    new_tail = head # i=0
    old_tail = head.next # j=1
    while old_tail.next!= None: # getting the 2nd last node
        new_tail = new_tail.next # i = i+1
        old_tail = old_tail.next # j = j+1

    old_tail.next = head  # Link the old tail to the head (putting the old tail before the head)
    new_tail.next = None  # 2nd last node becomes the new tail. Only tail has None at end
    head = old_tail  # Making the old tail the new head

    return head

arr = np.array([10, 20, 30, 40, 50])
head6 = createList(arr)

print("Original Linked List:")
printLL(head6)

# Rotate the linked list to the right
head6 = rotateRight2(head6)
print("After rotating right:")
printLL(head6)

# Rotate again to see the effect
head6 = rotateRight2(head6)
print("After rotating right again:")
printLL(head6)
```

```
Original Linked List:
10 -> 20 -> 30 -> 40 -> 50 ->
After rotating right:
50 -> 10 -> 20 -> 30 -> 40 ->
After rotating right again:
40 -> 50 -> 10 -> 20 -> 30 ->
```

## 17. rotateLeftMul(): Rotate a list left by k positions

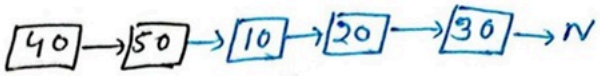
### First check, if rotation is needed or not.

- size = 5
  - rotate= 5
  - rotate = rotate%size = 5%5 = 0 [No rotation needed]
- size = 5
  - rotate= 15
  - rotate = rotate%size = 15%5 = 0 [No rotation needed]
- size = 5
  - rotate= 2
  - rotate = rotate%size = 2%5 = 2 [rotation needed 2 times]
- size = 5
  - rotate= 1
  - rotate = rotate%size = 1%5 = 1 [rotation needed 1 time]




Original list

for K=3




After Left rotation

step 1: Check if rotation is needed

Check Both
- ① if LL exists or K >= 1 → if yes, rotate
- ② check if K is multiple of size
  - → If yes, no rotation

step 2: Find newTail, newHead

↳ traverse (K-1) times, starting from head

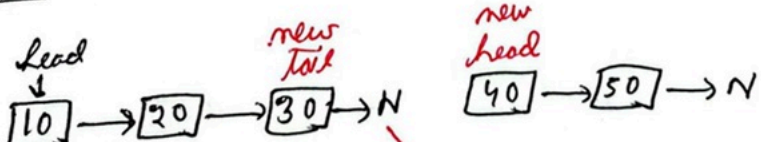

K = 3

traverse K-1 = 3-1 = 2 Times

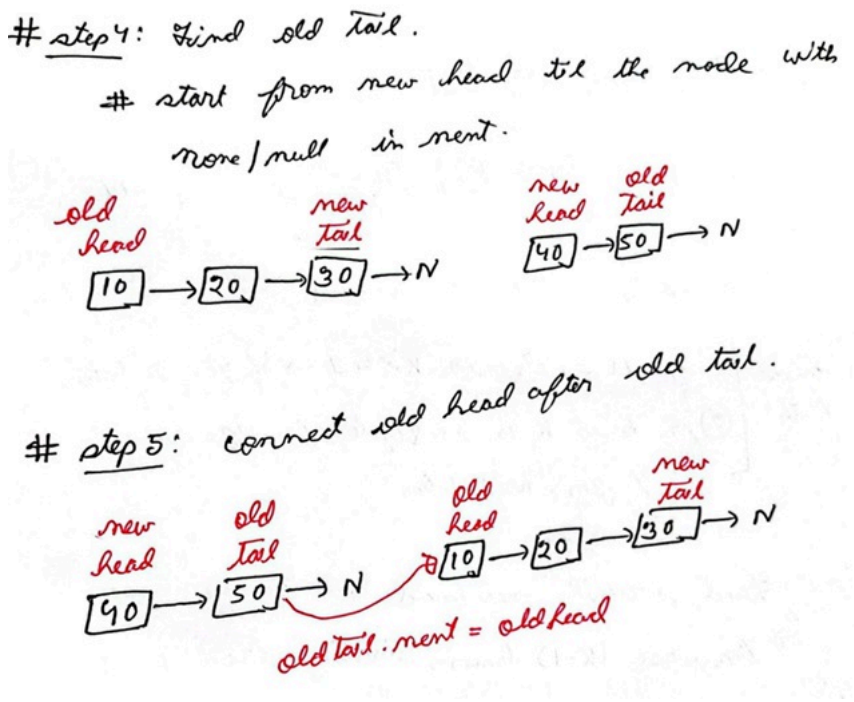
the node after (K-1) Traversal is newTail.

the node after newTail is newHead

step 3: break connection between newTail and newHead.




newTail.next = N




```
import numpy as np
def rotateLeftMul(head, k=1):
    """Rotates the linked list left by k positions.
    If k is not given, it will rotate once by default."""

    #-------STEP1: check if rotation needed---------------------------------
    # If the linked list is empty or k is non-positive, return the original list
    if not head or k <= 0:
        return head

    # Get the total number of nodes in the linked list
    size = countNode(head)

    # Optimize k by taking modulo to avoid unnecessary rotations
    k = k % size
    if k == 0:  # If k is a multiple of size, no rotation is needed
        return head
   #-----------------------------------------------------------------

    #-------STEP2: loop traverse for (k-1),, to get newtail----

    # Traverse to the (k-1)th node (the node just before the new_head--new_tail)
    current = head  # current will be newtail
    for _ in range(k - 1):
        current = current.next

    # The new head will be the next node after (k-1)th node
    new_head = current.next
   #-----------------------------------------------------------------

    # #-------STEP3:Break the link between (k-1)th node (new_tail) and new_head
    current.next = None # putting none after newtail


    ##-------STEP4: Finding the old_tail,,, getTail()
    # Find the tail of the new list (traverse till the last node)
    old_tail = new_head
    while old_tail.next != None:
        old_tail = old_tail.next
    #-----------------------------------------------------------------

    # -------STEP5:Connect the old_tail to the old head, making it a circular connection
    old_tail.next = head

    # Return the new head of the rotated list
    return new_head
```

```
# Create an array representing elements of the linked list
arr = np.array([10, 20, 30, 40, 50])

# Convert the array into a linked list
head22 = createList(arr)

# Print the original linked list
printLL(head22)

# Rotate the list to the left by 2 positions and print the result
print("Rotation left by 2 positions")
rotated_head = rotateLeftMul(head22, 2)
printLL(rotated_head)

# Rotate the newly obtained list to the left by 1 more position and print the result
print("Rotation left by 1 position")
rotated_head = rotateLeftMul(rotated_head, 1)
printLL(rotated_head)
```

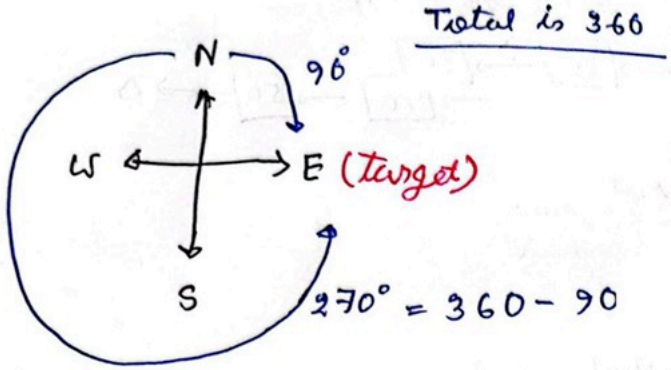
```
10 -> 20 -> 30 -> 40 -> 50 ->
Rotation left by 2 positions
30 -> 40 -> 50 -> 10 -> 20 ->
Rotation left by 1 position
40 -> 50 -> 10 -> 20 -> 30 ->
```

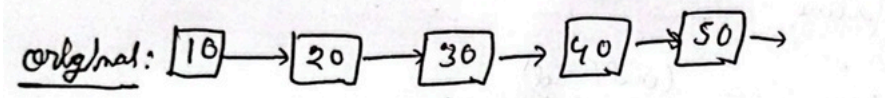
```
public void rotateLeftMul(int k) {
    if (head == null || k <= 0){return;}

    int Size = get_size();
    k = k % Size; // Avoid unnecessary rotations
    if (k == 0) return;

    // Traverse to (k-1)th node
    Node current = head;
    for (int i = 0; i < k - 1; i++) {
        current = current.next;
    }

    // Update new head and break the link
    Node newHead = current.next;
    current.next = null;

    // Find the tail of the new list
    Node tail = newHead;
    while (tail.next != null) {
        tail = tail.next;
    }

    // Connect the old tail to the old head
    tail.next = head;
    head = newHead;
}
```

## 18. rotateRightMul(): Rotating a list right by k positions

- Rotating right by k = Rotate left by (size - k)

# Rotate Right Mod

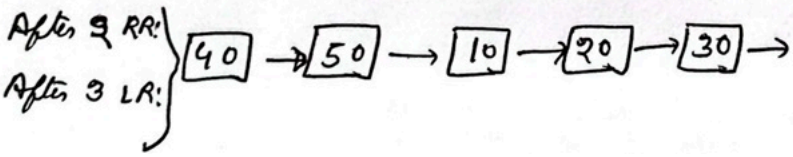



RR = 2 , LR = Size − RR

= 5 − 2

= 3



```
def rotateRightMul(head, k=1):
    """Rotates the linked list right by k positions.
    If k is not given, it will rotate once by default."""

    # If the linked list is empty or k is non-positive, return the original list
    if not head or k <= 0:
        return head

    # Get the total number of nodes in the linked list
    size = countNode(head)

    # Optimize k by taking modulo to avoid unnecessary rotations
    k = k % size
    if k == 0:  # If k is a multiple of size, no rotation is needed
        return head

    # Rotate left by (size - k), equivalent to rotating right by k
    return rotateLeftMul(head, size - k)

arr = np.array([10, 20, 30, 40, 50])
head22 = createList(arr)


printLL(head22)

# Rotate the list to the left by 2 positions
print("Rotation right by 2 position")
rotated_head = rotateRightMul(head22, 2)
printLL(rotated_head)

# Rotate the list to the left by 1 positions
print("Rotation right by 1 position")
rotated_head = rotateRightMul(rotated_head, 1)
printLL(rotated_head)
```

```
10 -> 20 -> 30 -> 40 -> 50 ->
Rotation right by 2 position
40 -> 50 -> 10 -> 20 -> 30 ->
Rotation right by 1 position
30 -> 40 -> 50 -> 10 -> 20 ->
```

```
public void rotateRightMul(int k) {
    if (head == null || k <= 0) return;

    int size = get_size();
    k = k % size; // Avoid unnecessary rotations
    if (k == 0) return;

    // Rotate left by (size - k), equivalent to rotating right by k
    rotateLeftMul(size - k);
}
```

## Linked List Class - optional

Let's try to combine all the codes above and try to create a Linked List Class

```
# Defining the Node class
class Node:
    def __init__(self, elem, next=None):
        self.elem = elem
        self.next = next
#_____________________________________________________________________
#This linked list class will remember the head node--its stored in self
class LinkedList:
    def __init__(self, arr): # Constructor
        self.head = None

    def printLL(self):
        """Prints the linked list in a readable format."""
```

```
        temp = self.head
        while temp!=None:
            print(temp.elem, end=' -> ')
            temp = temp.next
        print()  # Move to the next line after printing

    def createList(self, arr):
        """Creates a linked list from an array."""
        if len(arr)==0:  # Check if the array is empty
            return None  # Return None for an empty list

        self.head = Node(arr[0])
        tail = self.head
        for i in range(1, len(arr)):
            newNode = Node(arr[i])
            tail.next = newNode
            tail = newNode

    def countNode(self):
        """Counts the number of nodes in the linked list."""
        current_node = self.head
        count = 0
        while current_node!=None:
            count += 1
            current_node = current_node.next
        return count

    def get_tail(self):
        """Returns the last node (tail) of the linked list."""
        temp = self.head
        while temp.next!=None:  # Iterating till the next is None
            temp = temp.next
        return temp

    def appendLL(self, data):
        """Appends a new node with the given data to the end of the linked list."""
        newNode = Node(data)
        if self.head == None:  # If the list is empty, newNode becomes the head
            self.head = newNode
            return
        tail = self.get_tail()  # Get the current tail
        tail.next = newNode  # Link the new node at the end of the tail

    def prependLL(self, elem):
        """Prepends a new node with the given element to the beginning of the linked list."""
        newNode = Node(elem)
        newNode.next = self.head
        self.head = newNode  # Update the head to the new node

    def getElem(self, index):
        """Returns the element at the specified index."""
        current_node = self.head
        count = 0
        while current_node!=None:
            if count == index:
                return current_node.elem
            current_node = current_node.next
            count += 1
        return None  # Index is out of bounds

    def getNode(self, index):
        """Returns the node at the specified index."""
        current_node = self.head
        count = 0
        while current_node!=None:
            if count == index:
                return current_node
            current_node = current_node.next
            count += 1
        return None  # Index is out of bounds

    def setNode1(self, head, index, new_elem):
        """Sets the element at the specified index to new_elem."""
        current_node = self.head
        count = 0
        while current_node!=None:
            if count == index:
```

```
                current_node.elem = new_elem
                print("Successfully updated")
                return
            current_node = current_node.next
            count += 1
        print("Invalid Index")  # If the index is invalid

    def indexOf(self, target):
        """Returns the index of the given element, or -1 if not found."""
        current_node = self.head
        index = 0
        while current_node!=None:
            if current_node.elem == target:
                return index
            current_node = current_node.next
            index += 1
        return -1  # Element not found

    def contains(self, target):
        """Returns True if the linked list contains the target element, False otherwise."""
        current_node = self.head
        while current_node!=None:
            if current_node.elem == target:
                return True
            current_node = current_node.next
        return False  # Element not found

    def insertAt(self, idx, elem):
        """Inserts a new element at the specified index."""
        size = self.countNode()

        if idx < 0 or idx > size:  # Invalid index
            print("Insertion Failed, Not valid index")
            return
        elif idx == 0:  # Insertion at head
            self.prependLL(elem)
        else:  # Insertion in middle or end
            newNode = Node(elem)
            predecessor = self.getNode(idx - 1)
            successor = predecessor.next
            newNode.next = successor  # Link the successor
            predecessor.next = newNode  # Link the predecessor

    def removeAt(self, idx):
        """Removes the node at the specified index."""
        size = self.countNode()
        if idx < 0 or idx >= size:  # Invalid index
            print("Deletion Failed, Not valid index")
            return
        elif idx == 0:  # Deletion at head
            self.head = self.head.next  # Update the head
        else:  # Deletion in the middle or at the end
            predecessor = self.getNode(idx - 1)
            remNode = predecessor.next
            successor = remNode.next

            predecessor.next = successor  # Skip the removed element
            remNode.next = None  # Help garbage collection

    def rotateLeft(self):
        """Rotates the linked list to the left by one position."""
        if self.head == None or self.head.next == None:  # Check for empty or single-node lis
            return self.head

        old_head = self.head  # Keeping a copy of the old head
        self.head = self.head.next  # Assign the new head (2nd element is the new head now)

        tail = self.get_tail()  # Last node in the rotated list
        tail.next = old_head  # Putting original head at the end
        old_head.next = None  # Original head is the new tail now

    def rotateRight(self):
        """Rotates the linked list to the right by one position."""
        if self.head == None or self.head.next == None:  # Check for empty or single-node lis
            return self.head

        size = self.countNode()  # Count nodes in the list
```

```
        new_tail = self.getNode(size - 2)  # 2nd last node of the list becomes the new tail
        old_tail = new_tail.next  # Last node (size-1), which will be the new head

        old_tail.next = self.head  # Link the old tail to the head
        new_tail.next = None  # 2nd last node becomes the new tail
        self.head = old_tail  # Making the old tail the new head

    def rotateLeftMul(self, k=1):
        if not self.head or k <= 0:
            return
        size = self.countNode()
        k = k % size
        if k == 0:
            return

        current = self.head
        for _ in range(k - 1):
            current = current.next

        new_head = current.next
        current.next = None
        tail = new_head
        while tail.next:
            tail = tail.next
        tail.next = self.head
        self.head = new_head

    def rotateRightMul(self, k=1):
        if not self.head or k <= 0:
            return
        size = self.countNode()
        k = k % size
        if k == 0:
            return

        self.rotateLeftMul(size - k)

    def copyList(self):
        """Creates a copy of the linked list and returns the new head."""
        if self.head == None:  # If the list is empty, return None
            return None
```