

# CSE 220 Data Structures

Lecture 03: Singly Linked Lists – Node Removal and Rotation

Anwarul Bashir Shuaib [AWBS]

Lecturer

Department of Computer Science and Engineering

BRAC University

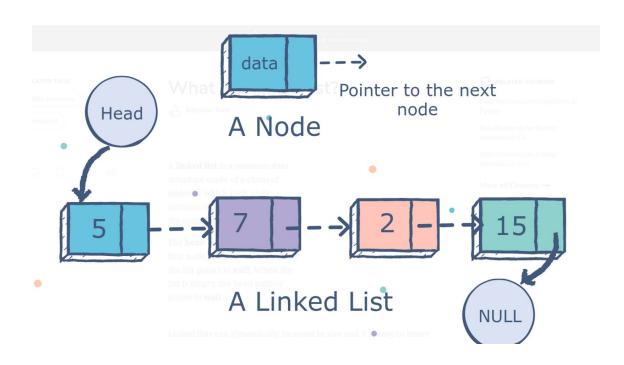




#### Review: Linked Lists

- Collection of nodes
- Each node contains two things:
  - Data
  - A reference to the **next node**

 This version is also known as "Singly Linked Lists (SLL)" as each node only contains the reference to the next node.





## Review: Arrays vs. Linked Lists

#### Arrays:

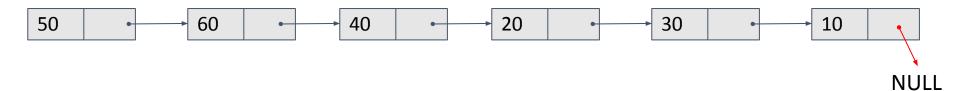
- Fixed size (static).
  - Insertion/deletion is expensive (requires shifting elements).
  - Random access is fast  $\Rightarrow$  O(1).

#### Linked Lists:

- Dynamic size (grows/shrinks as needed).
  - Insertion/deletion is efficient (O(1) at head/tail).
  - Sequential access (no random access, O(n) for traversal).



#### Review: Index of Element



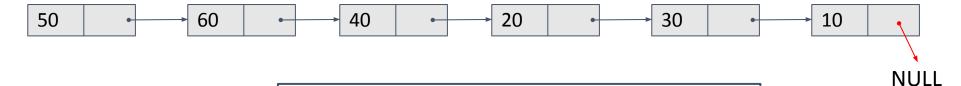
Find the index of 30

```
// 4. Retrieve index of an element
public int indexOf(int elem) {
   int index = 0;
   Node current = head;
   while (current != null) {
        if (current.elem == elem) {
           return index;
        current = current.next;
        index++;
   return -1; // Element not found
```





#### Review: Get Node at Index



Get the node at index 3

```
// 5. Retrieve a node from an index
public Node getNode(int index) {
    int currentIndex = 0;
    Node current = head;
    while (current != null) {
        if (currentIndex == index) {
            return current;
        current = current.next;
        currentIndex++;
    return null; // Index out of bounds
```

Anwarut Bashir Shuaib [AWBS]

 $\overline{O(n)}$ 



Construct the linked list from the given table

Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0×3000
0x3000	30	0×1000
0x4000	40	0x2000
0×5000	50	0x6000
0x6000	60	0×4000



 Step 1: Write down all nodes with their corresponding address.

Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0×3000
0x3000	30	0×1000
0x4000	40	0x2000
0×5000	50	0x6000
0x6000	60	0×4000



10 NULL 0x1000

20 0x3000 0x2000

30 0x1000 0x3000

0x2000 0x4000

50 0x6000 0x5000

60 0x4000 0x6000

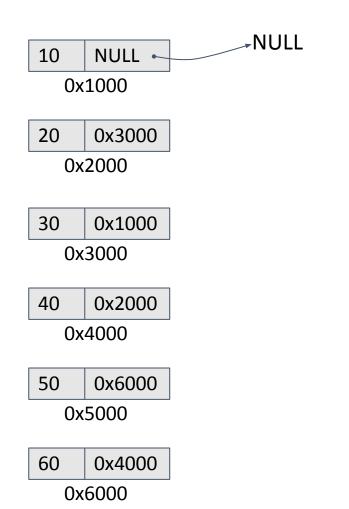
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0×1000
0x4000	40	0x2000
0x5000	50	0x6000
0×6000	60	0×4000



 Step 2: Identify links between nodes and connect

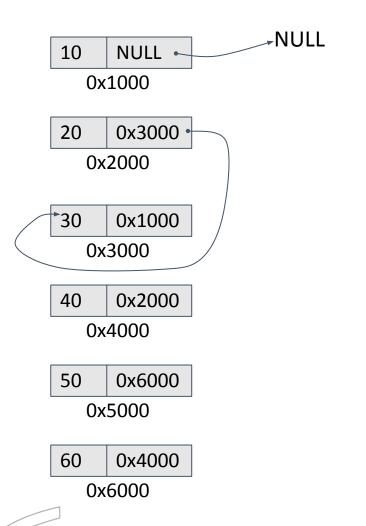
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000





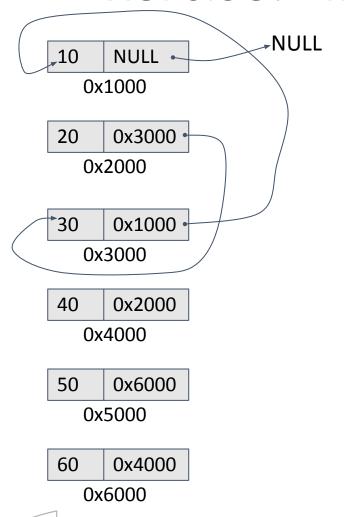
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0×5000	50	0x6000
0x6000	60	0×4000





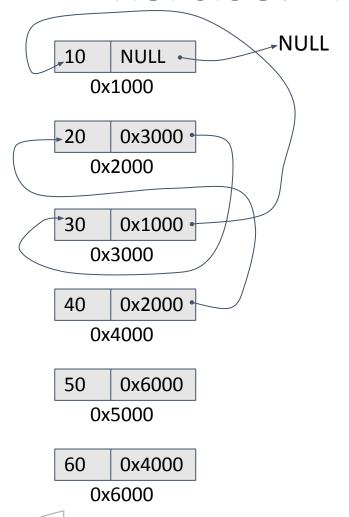
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0×3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0×4000





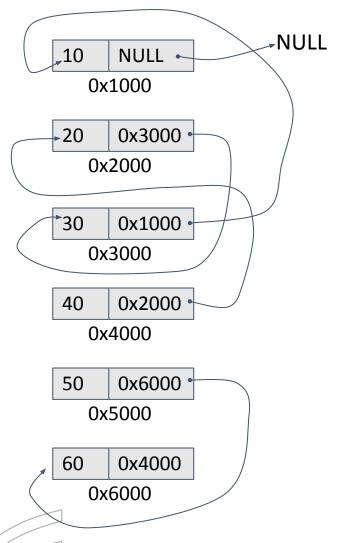
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0×4000





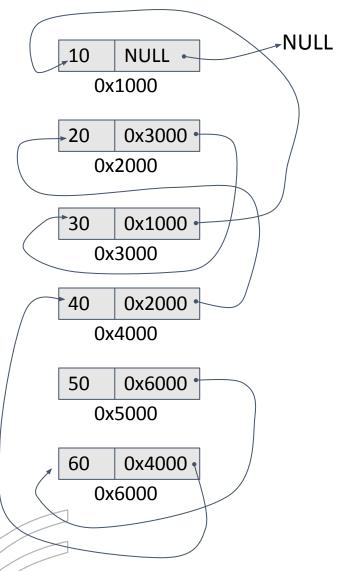
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000





Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0×3000
0x3000	30	0×1000
0x4000	40	0×2000
0x5000	50	0×6000
0x6000	60	0×4000





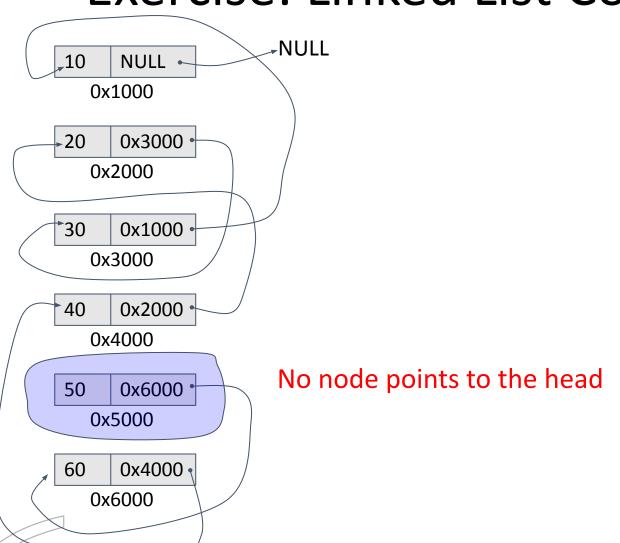
Memory Address	Data	Next Node Address
0×1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0×5000	50	0x6000
0x6000	60	0×4000



• Step 3: Identify the head

Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0×1000
0x4000	40	0x2000
0x5000	50	0x6000
0×6000	60	0×4000





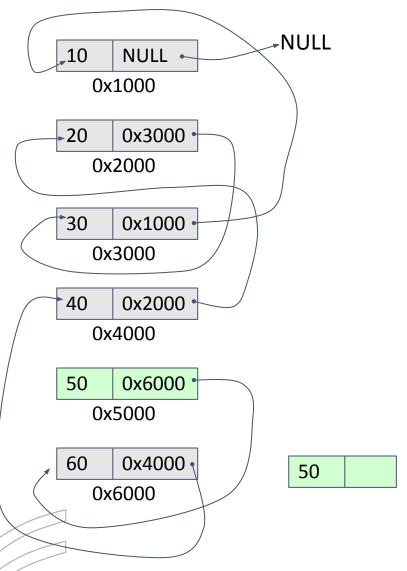
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0×5000	50	0x6000
0×6000	60	0x4000



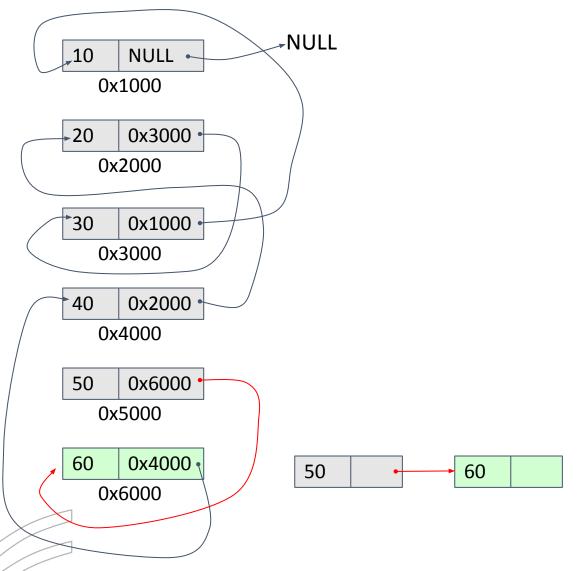
 Step 4: Traverse the list link by link from head and flatten it

Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0×1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0×4000

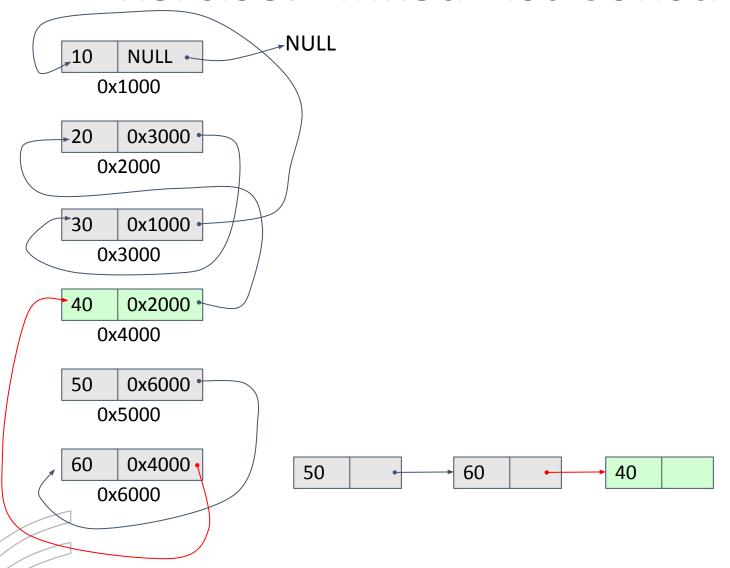




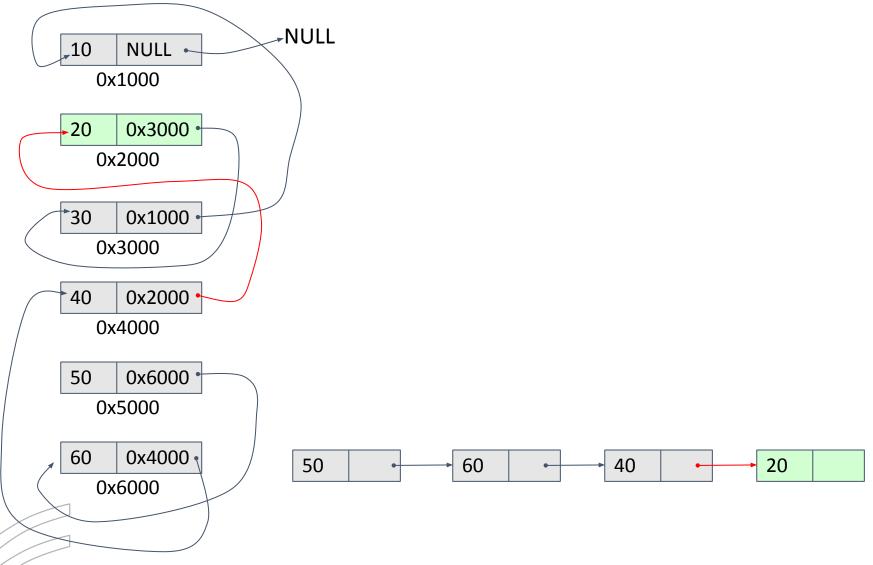




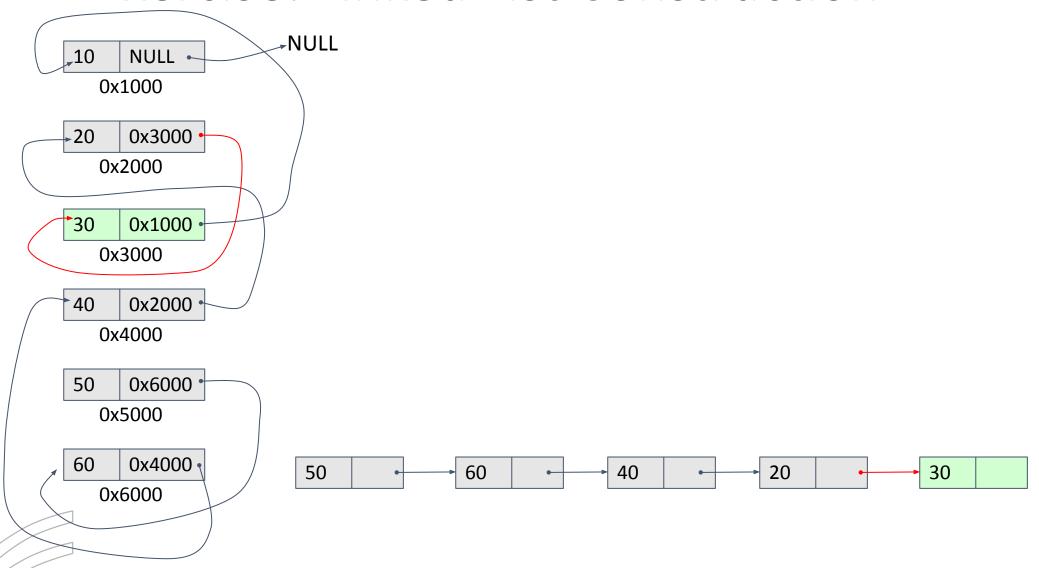




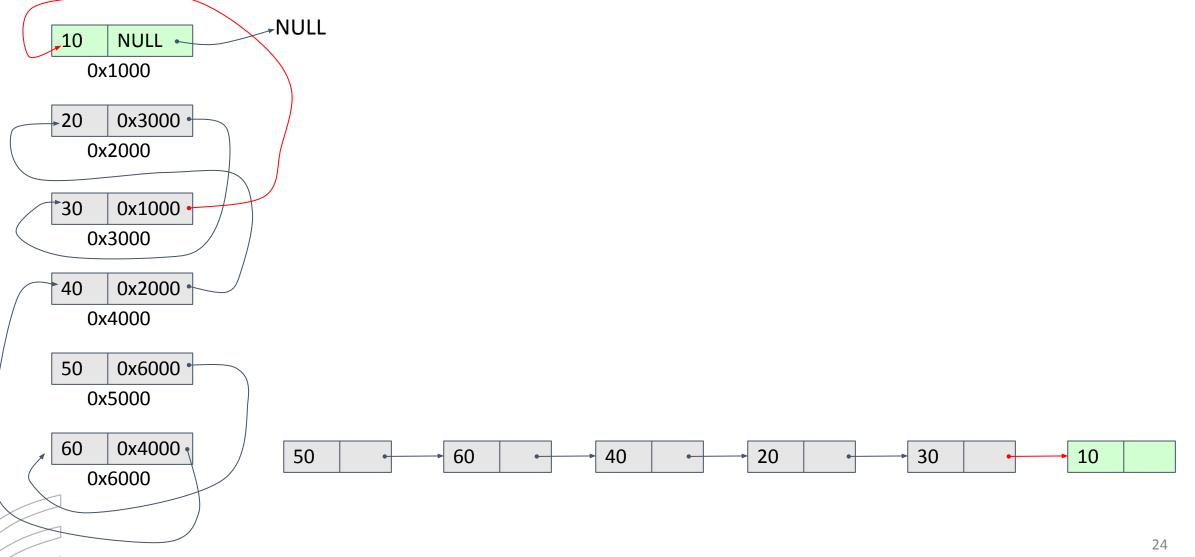




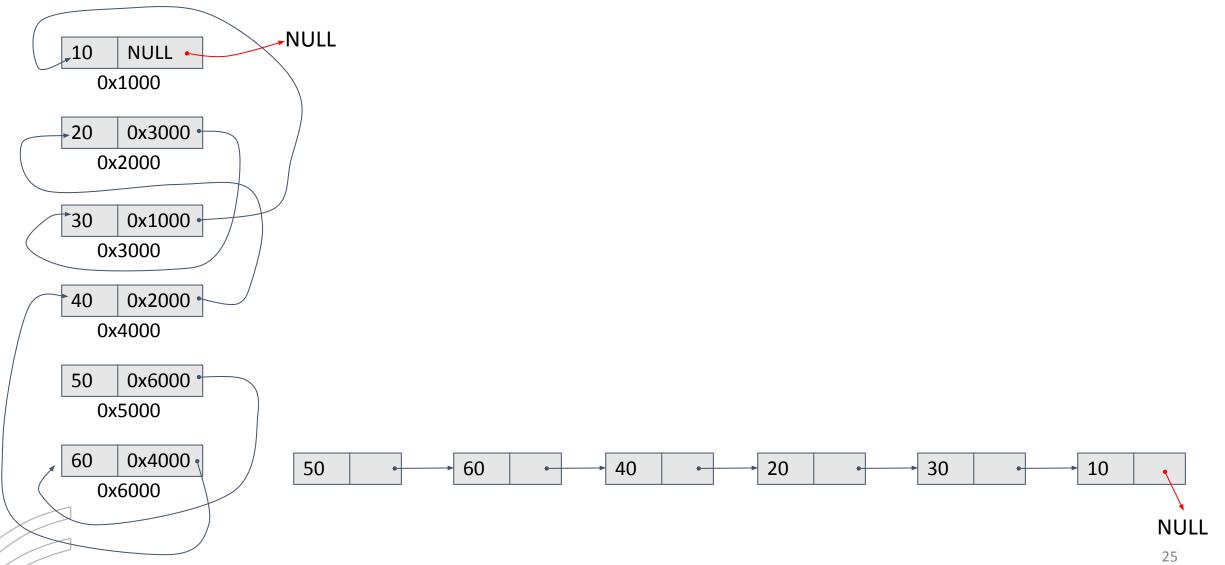




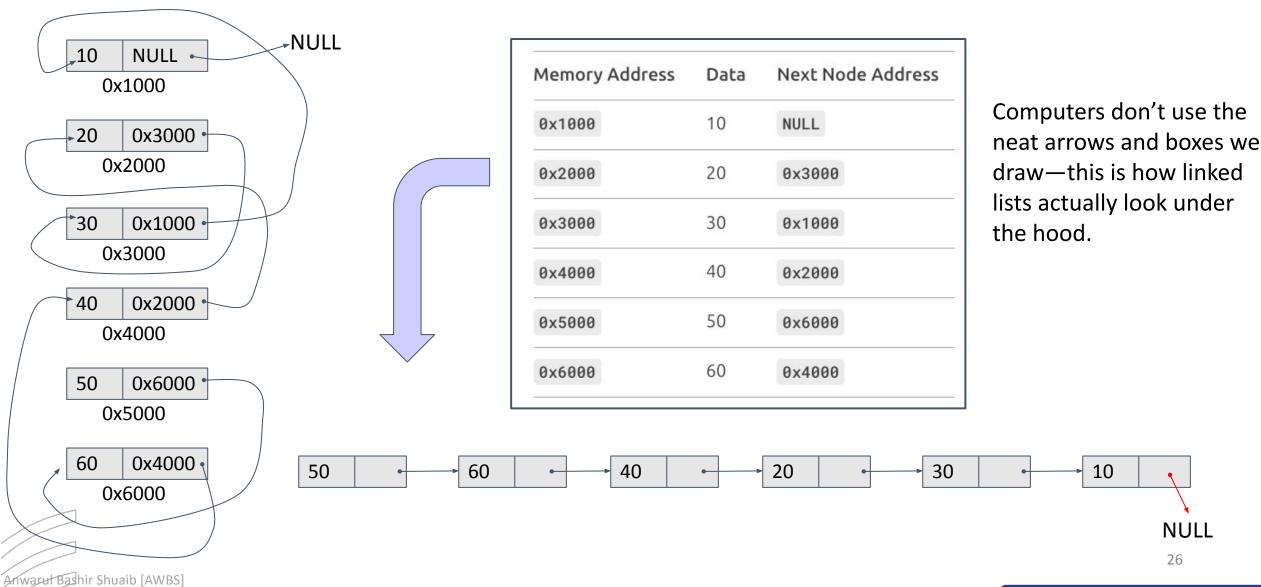










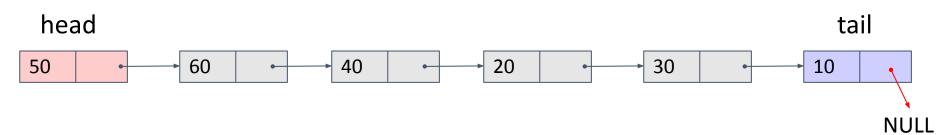




## Element Removal (Front)

Remove the front element (head)

```
// Remove from the beginning
```

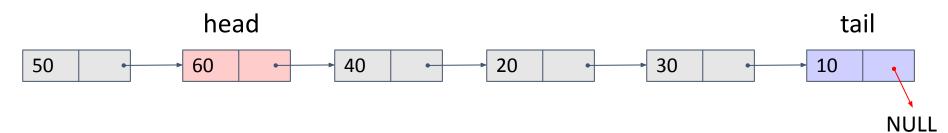




## Element Removal (Front)

Remove the front element (head)

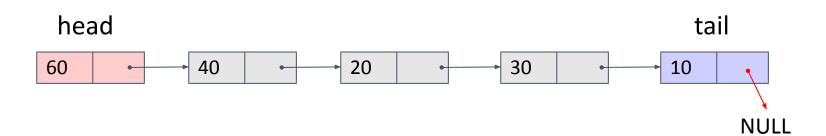
```
// Remove from the beginning
head = head.next;
```





• Remove the element (30) at index 3



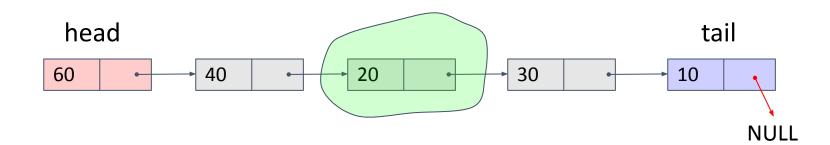






• Remove the element (30) at index 3

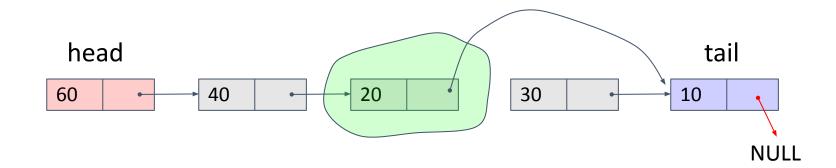
```
Node prev = getNode( index: index - 1);
```





• Remove the element (30) at index 3

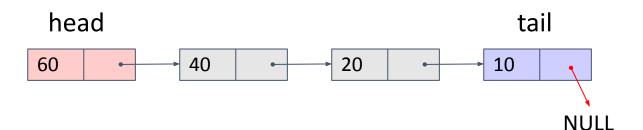
```
Node prev = getNode( index: index - 1);
prev.next = prev.next.next;
```





• Remove the element (30) at index 3

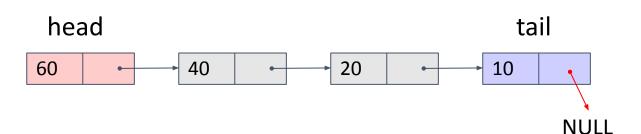
```
Node prev = getNode( index: index - 1);
prev.next = prev.next.next;
```





• Remove the element (30) at index 3

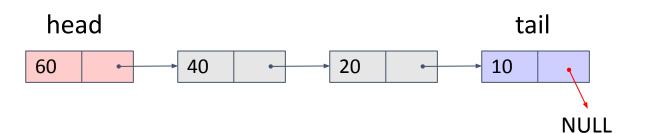
```
Node prev = getNode(index: index - 1);
if (prev != null && prev.next != null) { / Cannot remove before head or after tail
    prev.next = prev.next.next;
}
```





Can we optimize the tail removal?

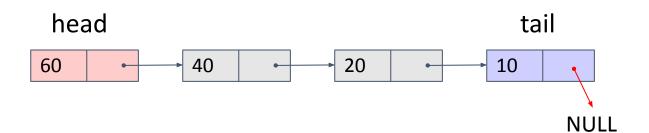
```
Node prev = getNode(index: index - 1);
if (prev != null && prev.next != null) {
    prev.next = prev.next.next;
}
```





- Can we optimize the tail removal?
  - No, we must be able to access the node before the last node in order to remove the last node.

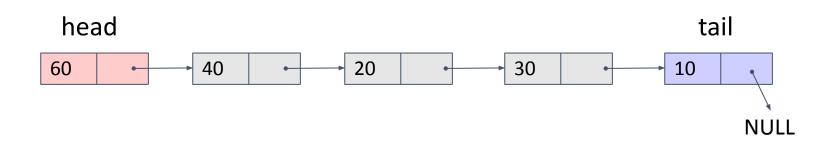
```
Node prev = getNode(index: index - 1);
if (prev != null && prev.next != null) {    / Cannot remove before head or after tail
    prev.next = prev.next.next;
}
```



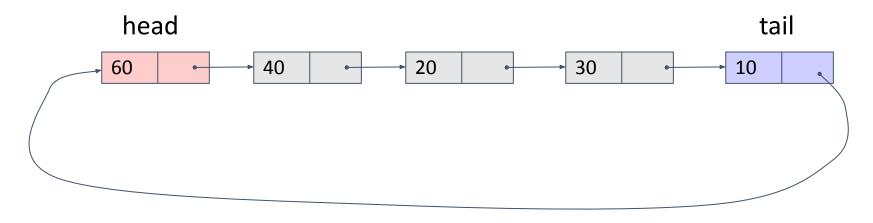


# List Rotation (Left)

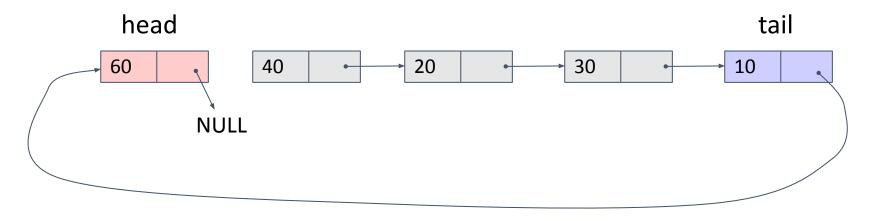
Rotate the list left by 1 element



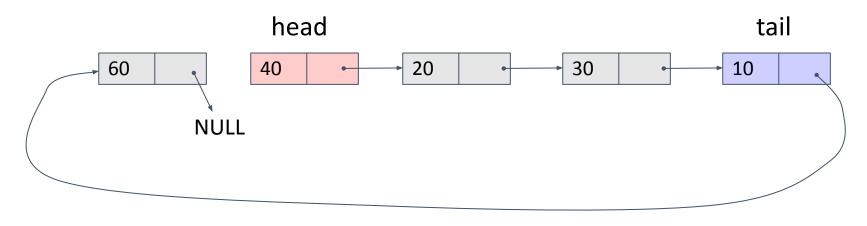




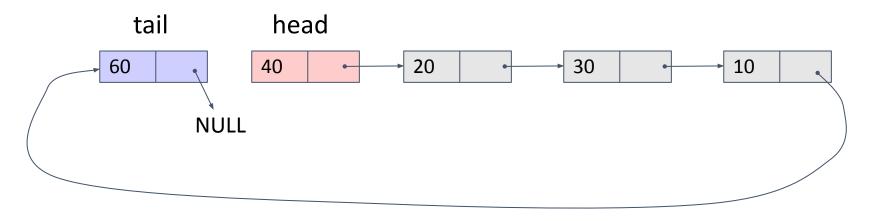






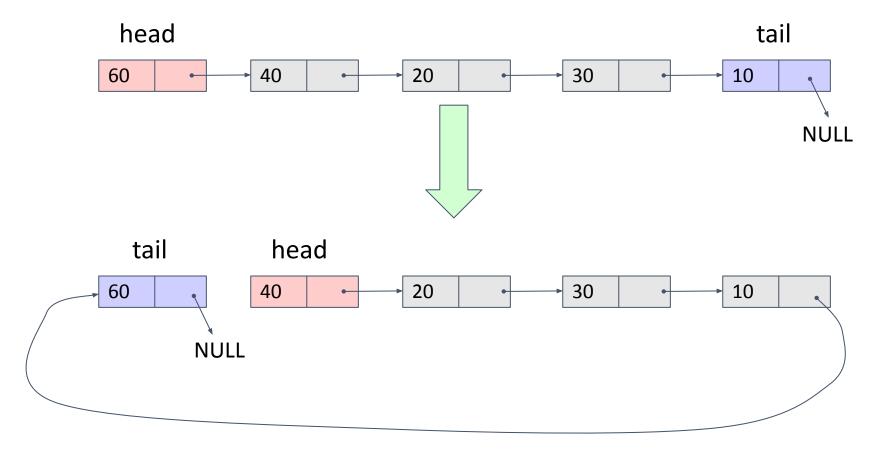






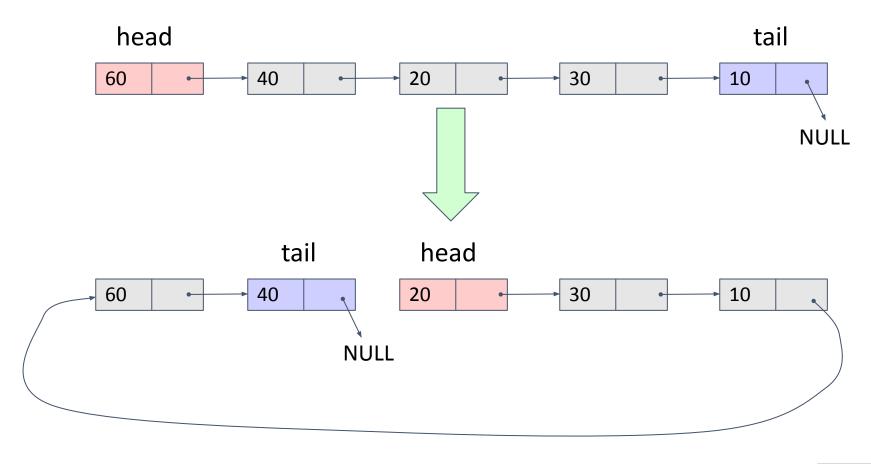


#### Rotate the list left by 1 element

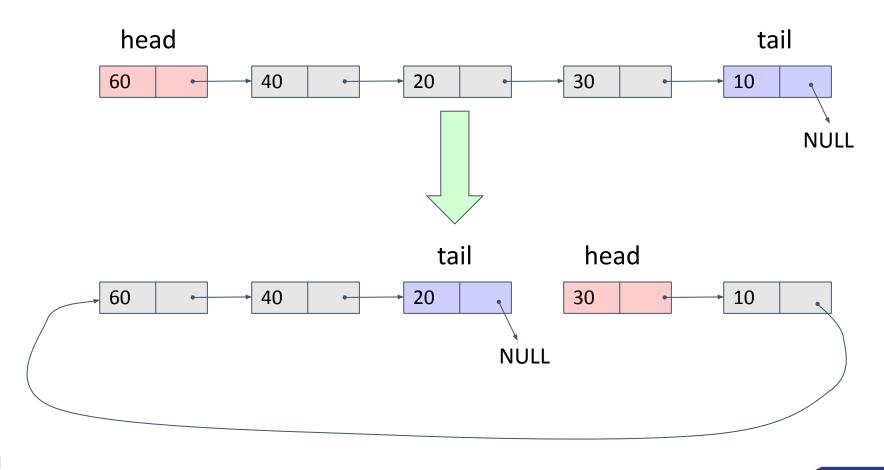


41

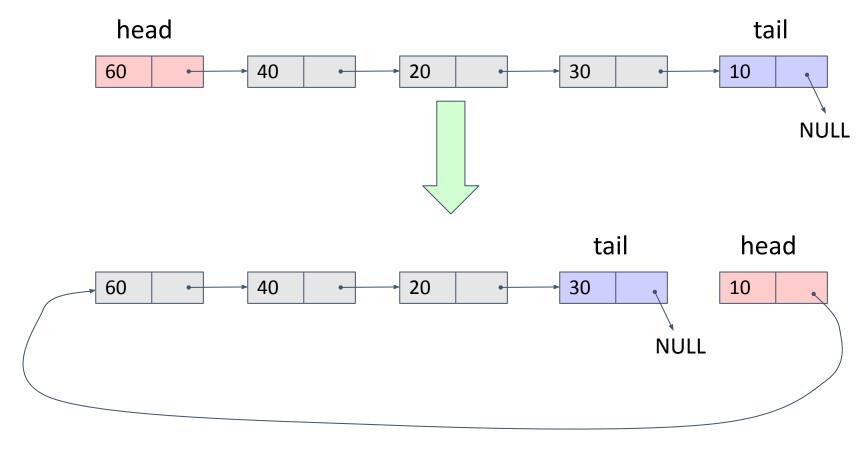






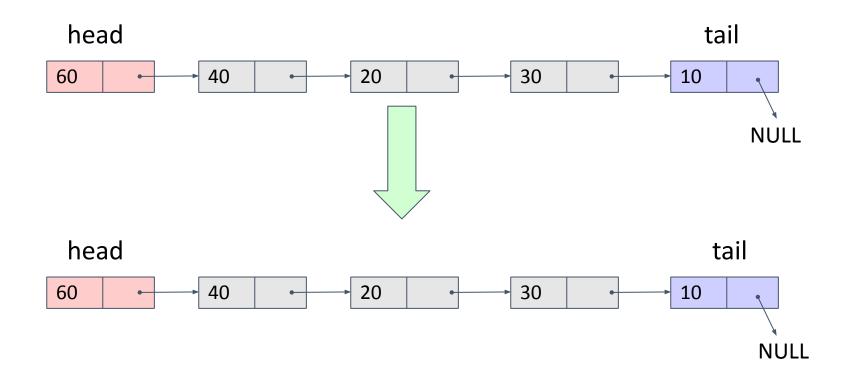






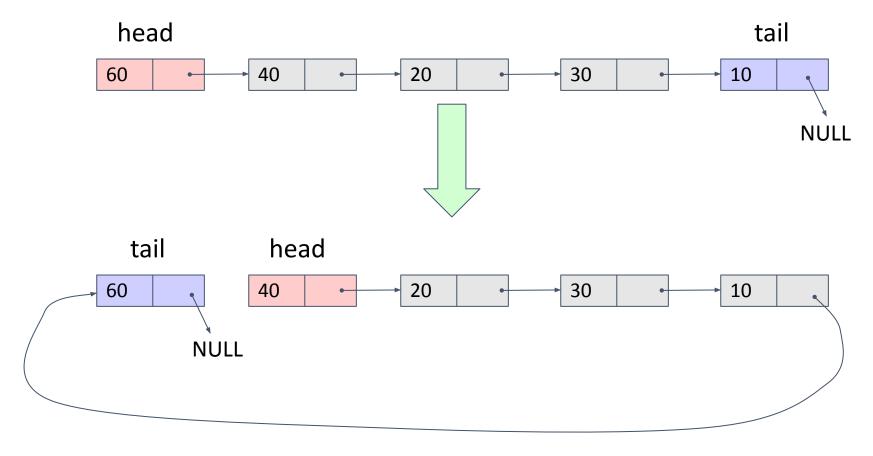


#### Rotate the list left by 5 elements



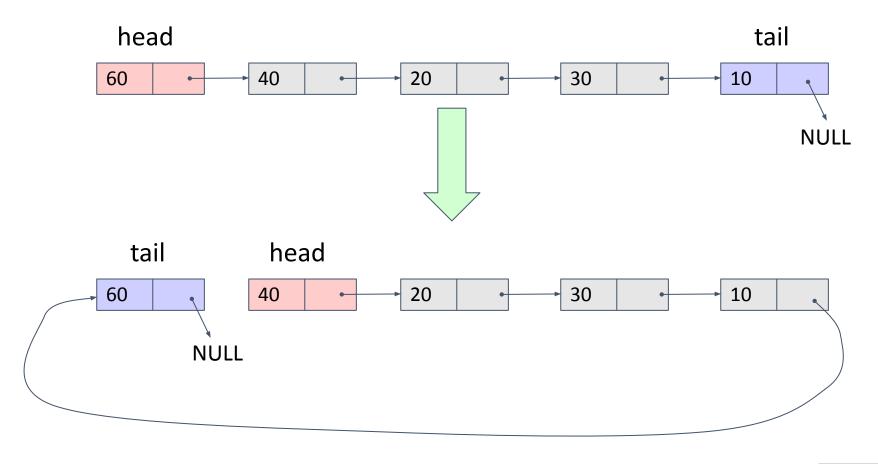
Anwarut Bashir Shuaib [AWBS]





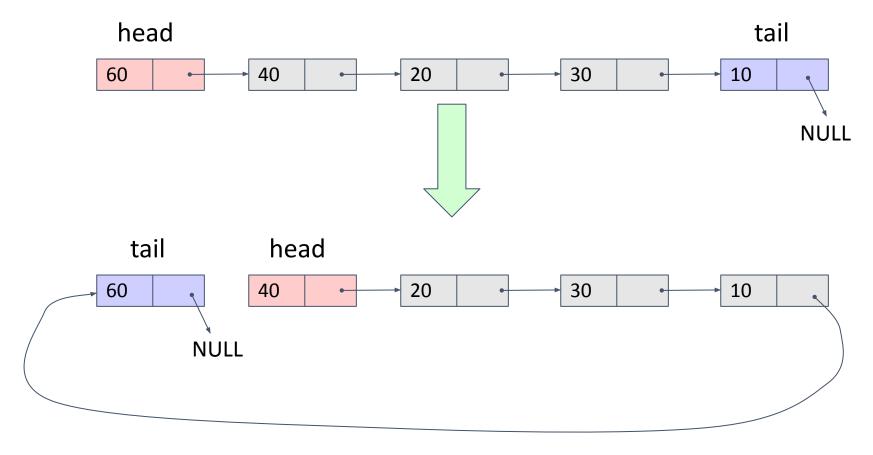


Rotate the list left by 6 elements == Rotating by 1 element





Rotate the list left by 6 elements == Rotating by 6 % 5 element



48



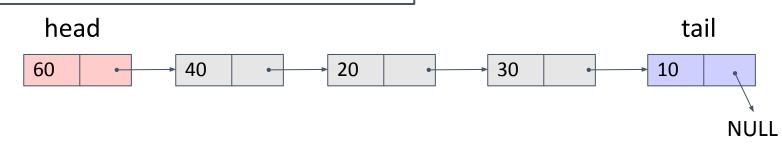
- Observations:
  - Rotation by  $k \Rightarrow$  Rotation by k % size
  - (k-1)th element becomes the new tail
  - kth element becomes the new head

Anwarut Bashir Shuaib [AWBS]



```
// 13. Rotating the list left
public void rotateLeft(int k) {
```

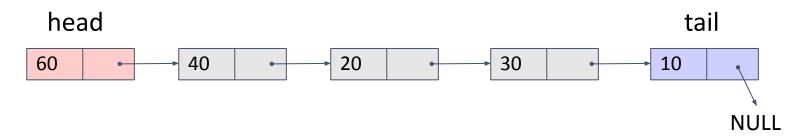
Rotate the list left by 3 elements



50

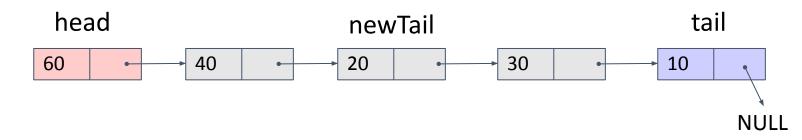


```
// 13. Rotating the list left
public void rotateLeft(int k) {
    int size = count();
    k = k \% size; // Handle rotations greater than size
    if (k == 0) return;
```



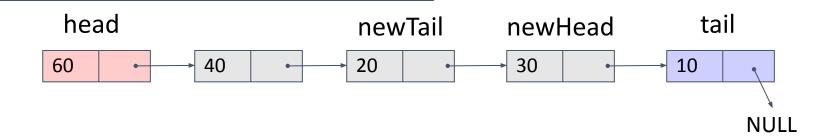


```
// 13. Rotating the list left
public void rotateLeft(int k) {
    int size = count();
    k = k \% size; // Handle rotations greater than size
    if (k == 0) return;
    Node newTail = getNode(index: k - 1);
```



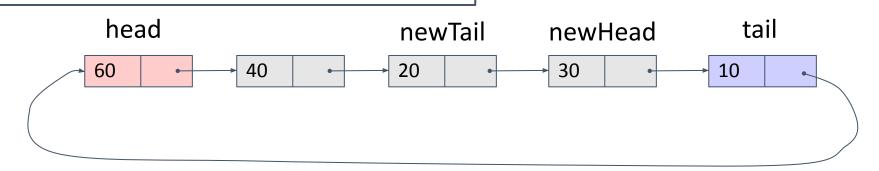


```
// 13. Rotating the list left
public void rotateLeft(int k) {
    int size = count();
    k = k % size; // Handle rotations greater than size
    if (k == 0) return;
    Node newTail = getNode(index: k - 1);
    Node newHead = newTail.next; // kth element is the new head
```



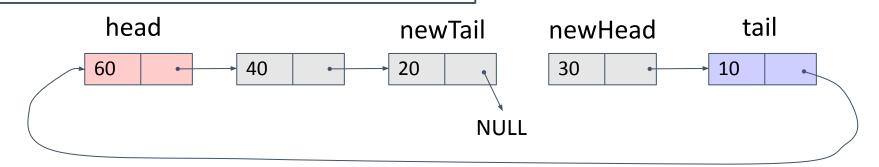


```
// 13. Rotating the list left
public void rotateLeft(int k) {
    int size = count();
    k = k % size; // Handle rotations greater than size
    if (k == 0) return;
    Node newTail = getNode(index: k - 1);
    Node newHead = newTail.next; // kth element is the new head
    // Adjust references
    tail.next = head;
                         // Connect original tail to old head
```





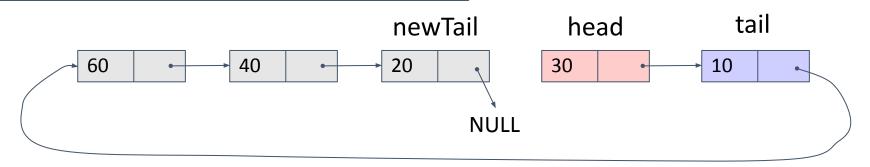
```
// 13. Rotating the list left
public void rotateLeft(int k) {
    int size = count();
    k = k % size; // Handle rotations greater than size
    if (k == 0) return;
   Node newTail = getNode(index: k - 1);
    Node newHead = newTail.next; // kth element is the new head
    // Adjust references
   tail.next = head; // Connect original tail to old head
   newTail.next = null; // Break the chain at new tail
```





```
// 13. Rotating the list left
public void rotateLeft(int k) {
   int size = count();
   k = k % size; // Handle rotations greater than size
   if (k == 0) return;
   Node newTail = getNode(index: k - 1);
   Node newHead = newTail.next; // kth element is the new head
   // Adjust references
   tail.next = head; // Connect original tail to old head
   newTail.next = null; // Break the chain at new tail
   head = newHead;
                         // Update head to new head
```

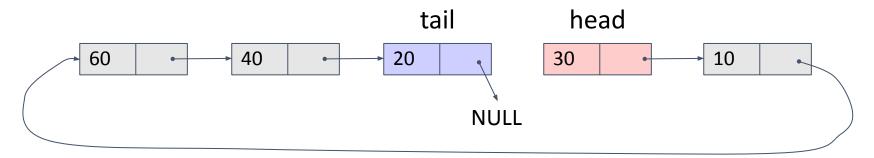
Anwarut Bashir Shuaib [AWBS]





```
// 13. Rotating the list left
public void rotateLeft(int k) {
    int size = count();
    k = k % size; // Handle rotations greater than size
    if (k == 0) return;
   Node newTail = getNode(index: k - 1);
    Node newHead = newTail.next; // kth element is the new head
    // Adjust references
   tail.next = head; // Connect original tail to old head
    newTail.next = null; // Break the chain at new tail
    head = newHead;
                         // Update head to new head
    tail = newTail;
                         // Update tail to new tail
```

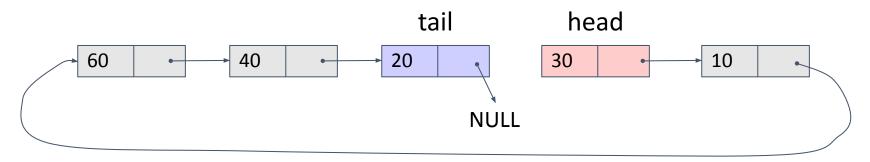
Anwarut Bashir Shuaib [AWBS]





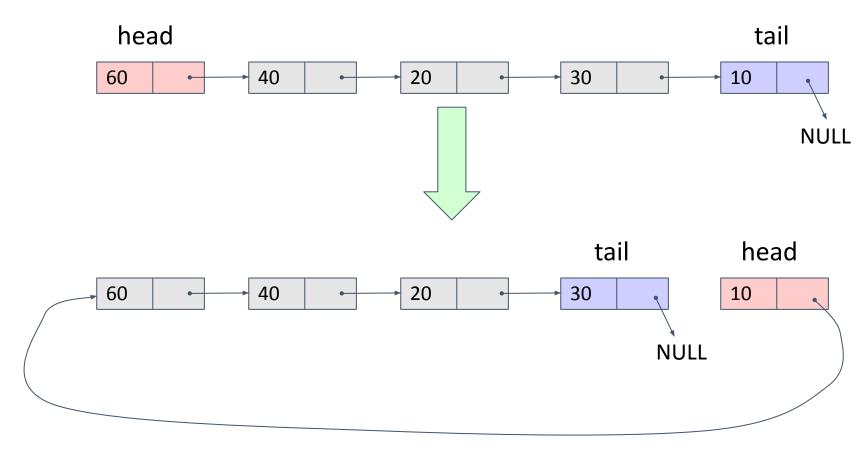
```
// 13. Rotating the list left
public void rotateLeft(int k) {
   if (head == null || k <= 0) return; // Boundary checks
   int size = count();
   k = k % size; // Handle rotations greater than size
   if (k == 0) return;
   Node newTail = getNode(index: k - 1);
   Node newHead = newTail.next; // kth element is the new head
   // Adjust references
   tail.next = head; // Connect original tail to old head
   newTail.next = null; // Break the chain at new tail
   head = newHead;
                        // Update head to new head
   tail = newTail;
                        // Update tail to new tail
```

Anwarut Bashir Shuaib [AWBS]



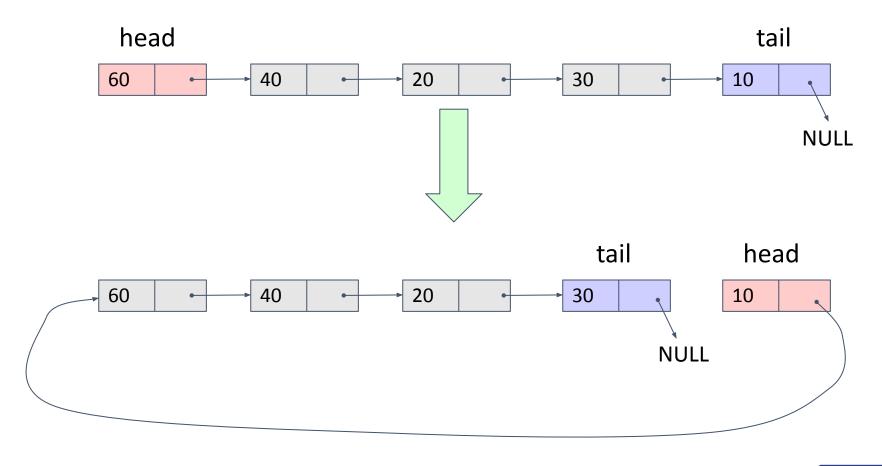


#### Rotate **left** by 4 elements



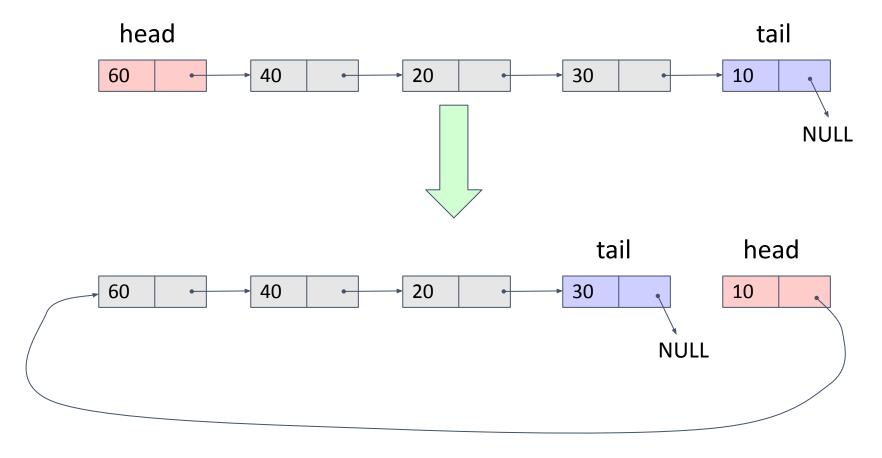


Rotate **left** by 4 elements == Rotate **right** by 1 element





Rotate **left** by 4 elements == Rotate **right** by (5 - 4) element



61



- Observations:
  - Left rotate by k == Right rotate by (size k)
  - Right rotate by k == left rotate by (size k)

```
// 14. Rotating the list right
public void rotateRight(int k) {
   if (head == null || k <= 0) return;
   int size = count();
   k = k % size; // Handle rotations greater than size
   if (k == 0) return;
   rotateLeft(k: size - k);
}</pre>
```

62



#### When to Use Linked Lists?

- Use linked lists:
  - When you need frequent insertions/deletions (e.g., implementing a queue or stack).
  - When the size of the data is unknown or changes frequently.
- Do not use linked lists:
  - Random access is needed (Arrays  $\Rightarrow$  O(1), Linked lists  $\Rightarrow$  O(n)
- Practical usage:
  - In xv6, the freelist refers to a linked list of free memory pages
  - Each free page contains a pointer to the next free page.



#### Exercise

- Practice easy Singly Linked List problems from Leetcode
- https://leetcode.com/problems/merge-two-sorted-lists/description/?envType=problem-list-v2&envId=linked-list
- https://leetcode.com/problems/delete-node-in-a-linked-list/description/?envType=problem-list-v2&envId=linked-list
- https://leetcode.com/problems/remove-duplicates-from-sorted-list/description/?envType=problem-list-v2&envId=linked-list
- https://leetcode.com/problems/linked-list-cycle/description/?envType=problem-list-v2&envId=linked-list
- https://leetcode.com/problems/intersection-of-two-linked-lists/description/?envType=problem-list-v2&envId=linked-list

