CSE 220
Data Structures

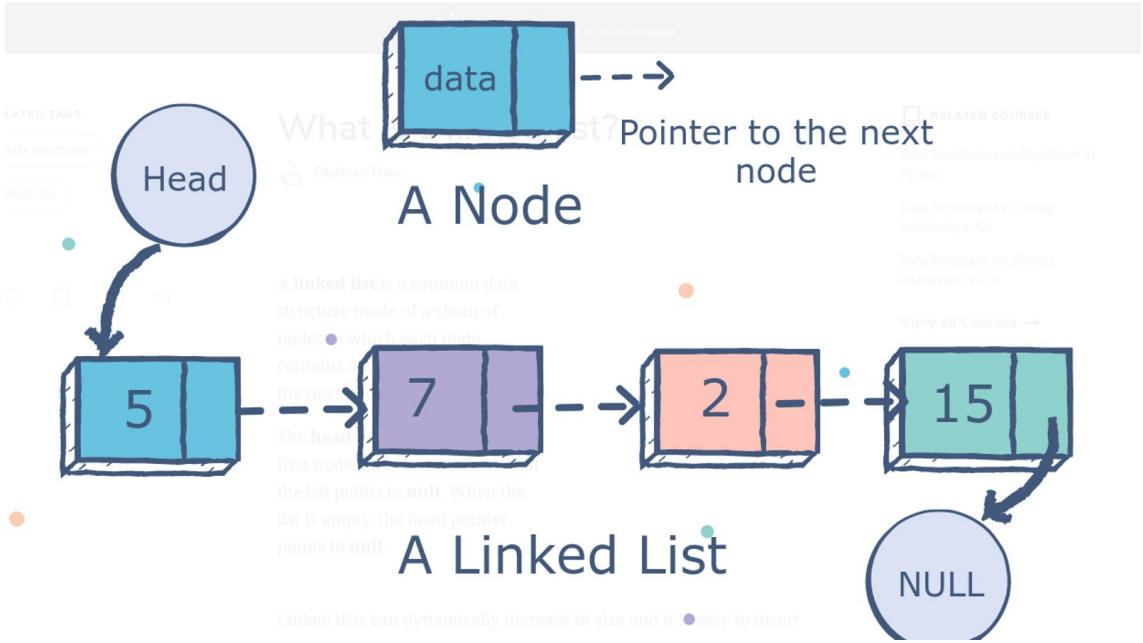
Lecture 03:
Singly Linked Lists – Node Removal and Rotation

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**

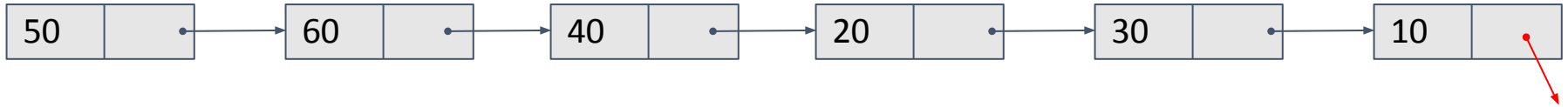
- This version is also known as “Singly Linked Lists (SLL)” as each node only contains the reference to the next node.



Review: Arrays vs. Linked Lists

- Arrays:
  - Fixed size (static).
    - Insertion/deletion is expensive (requires shifting elements).
    - Random access is fast ⇒ 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

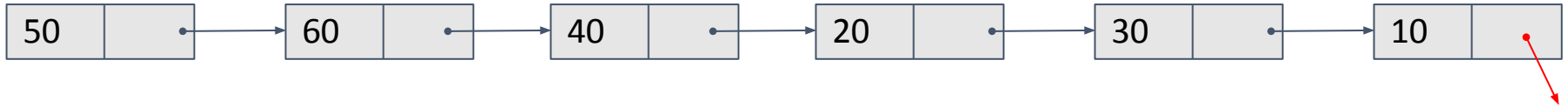


Find the index of 30

```java
// 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
}
```

$O(n)$

Review: Get Node at Index



Get the node at index 3

```java
// 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
}
```

$O(n)$

Exercise: Linked List Construction

- Construct the linked list from the given table

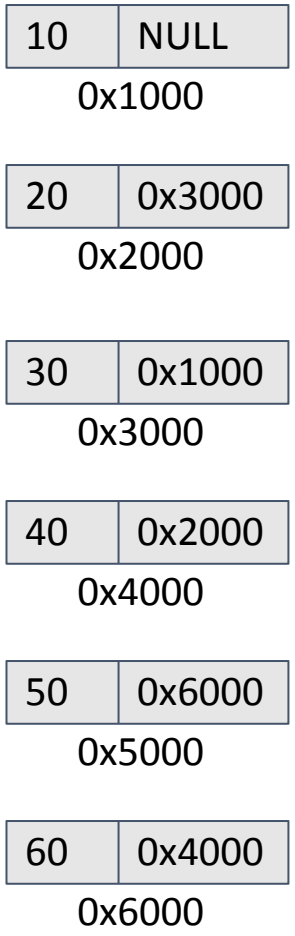
Memory Address	Data	Next Node Address
`0x1000`	10	`NULL`
`0x2000`	20	`0x3000`
`0x3000`	30	`0x1000`
`0x4000`	40	`0x2000`
`0x5000`	50	`0x6000`
`0x6000`	60	`0x4000`

Exercise: Linked List Construction

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

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

Exercise: Linked List Construction



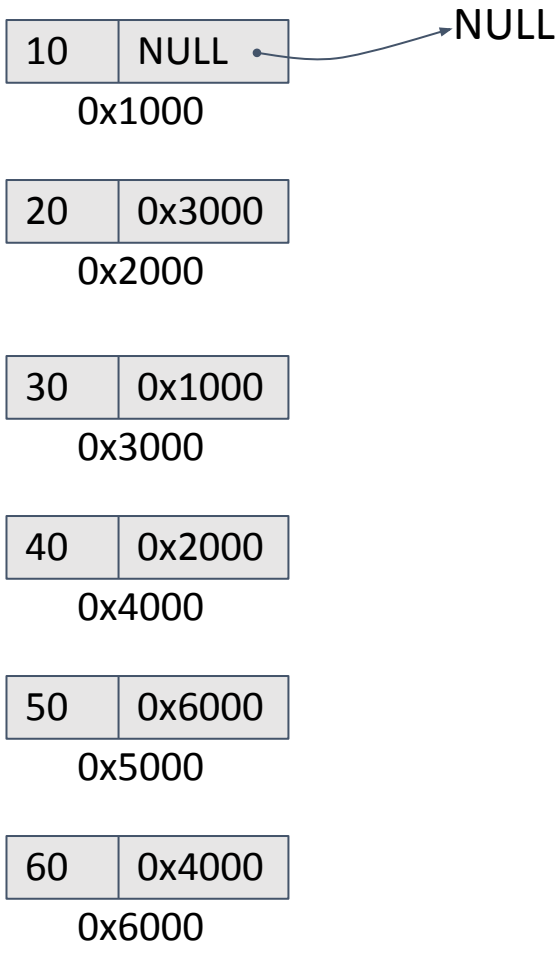
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction

- Step 2: Identify links between nodes and connect

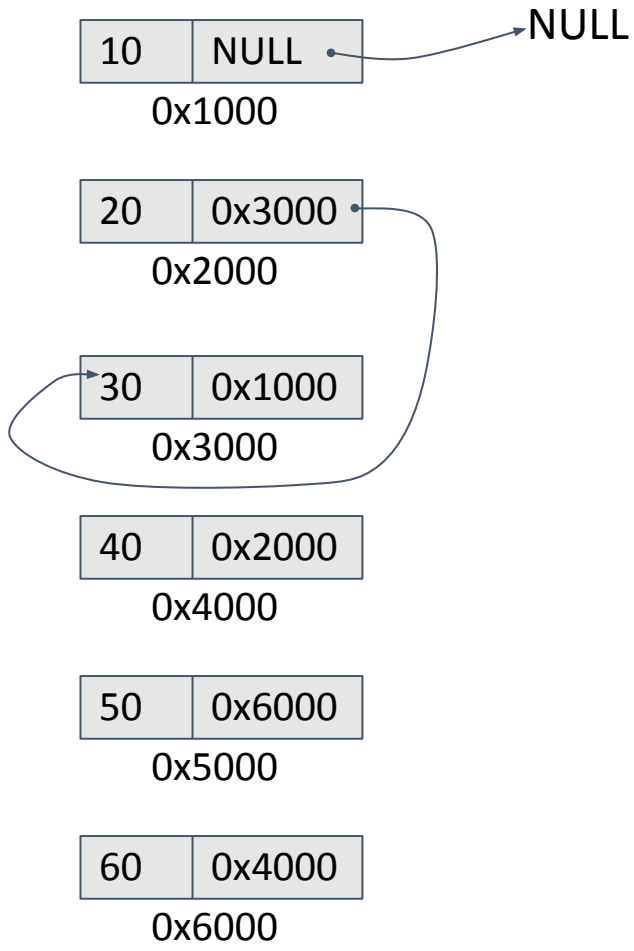
Memory Address	Data	Next Node Address
`0x1000`	10	`NULL`
`0x2000`	20	`0x3000`
`0x3000`	30	`0x1000`
`0x4000`	40	`0x2000`
`0x5000`	50	`0x6000`
`0x6000`	60	`0x4000`

Exercise: Linked List Construction



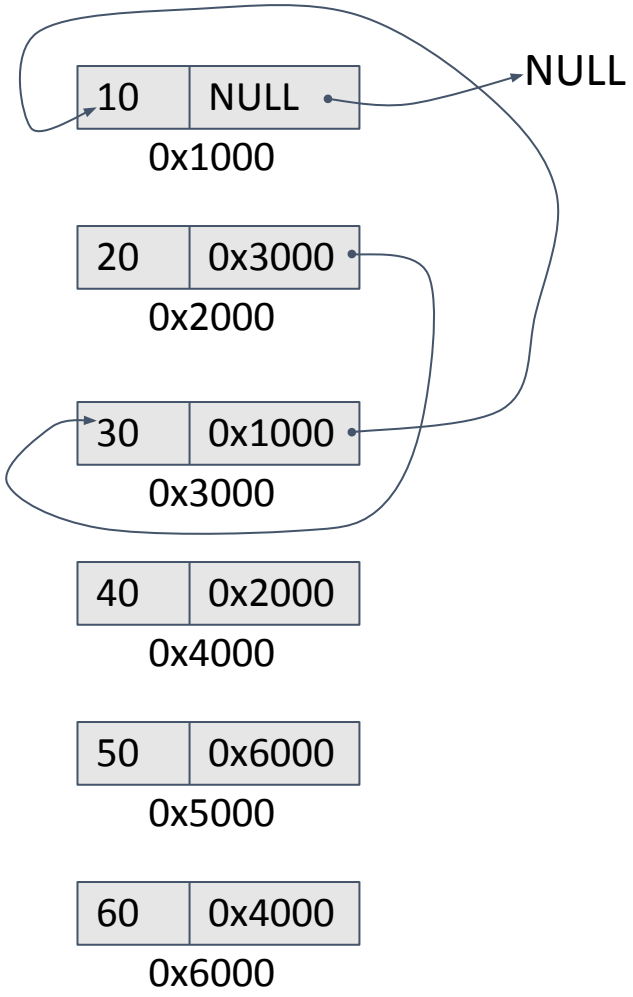
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction



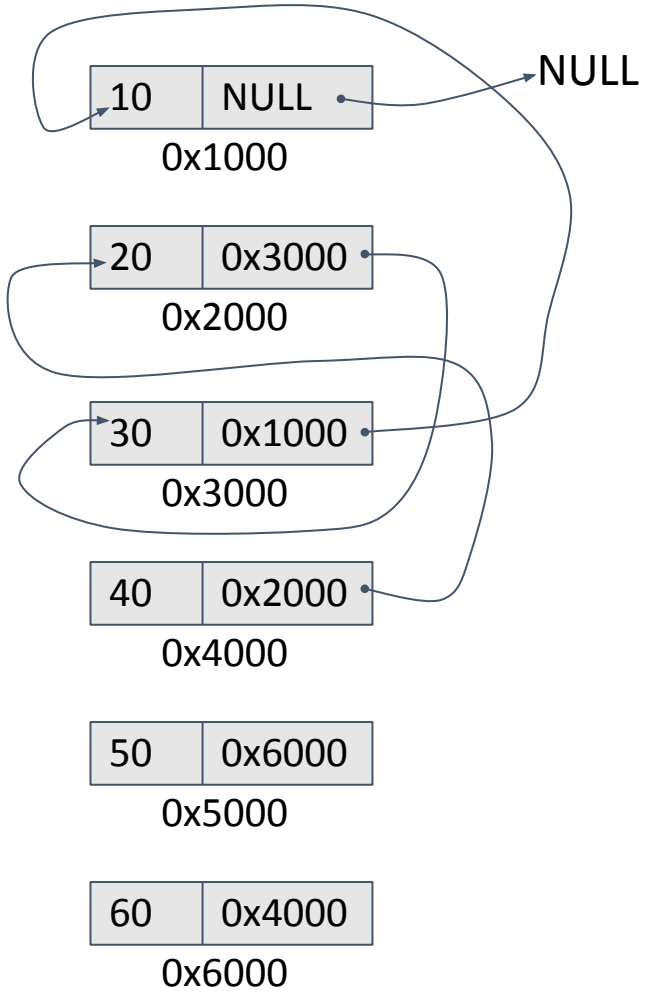
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction



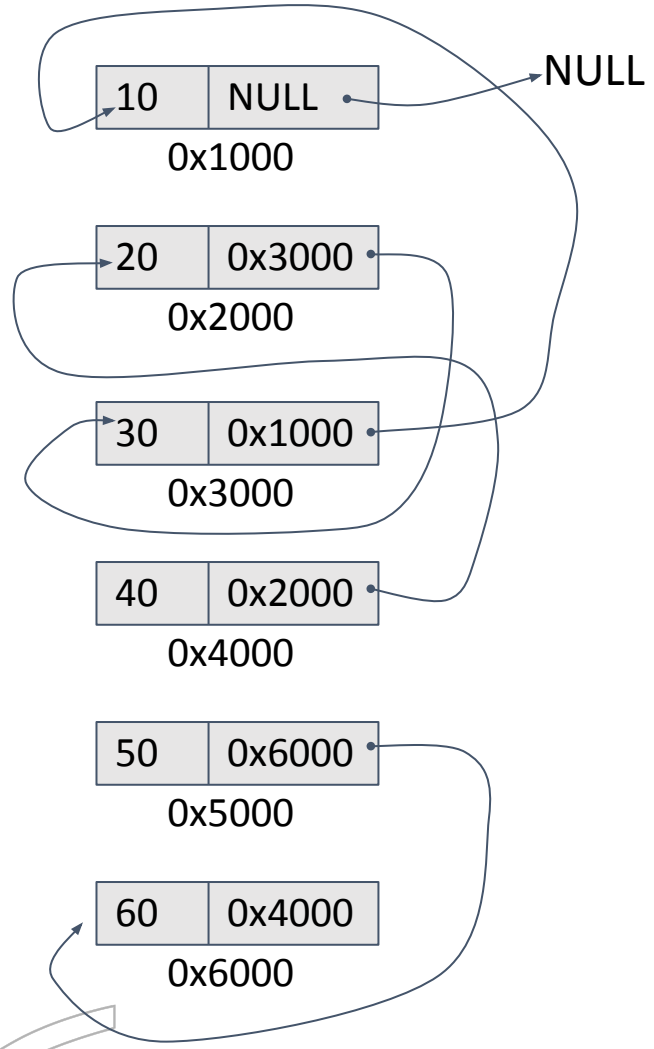
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction



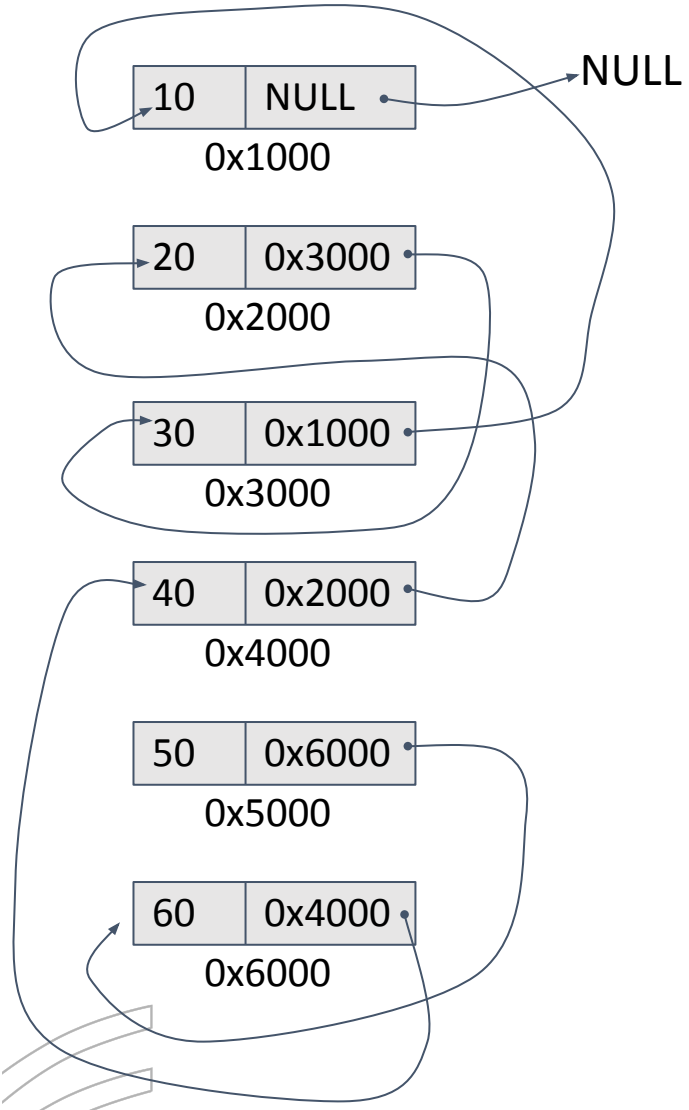
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction



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

Exercise: Linked List Construction



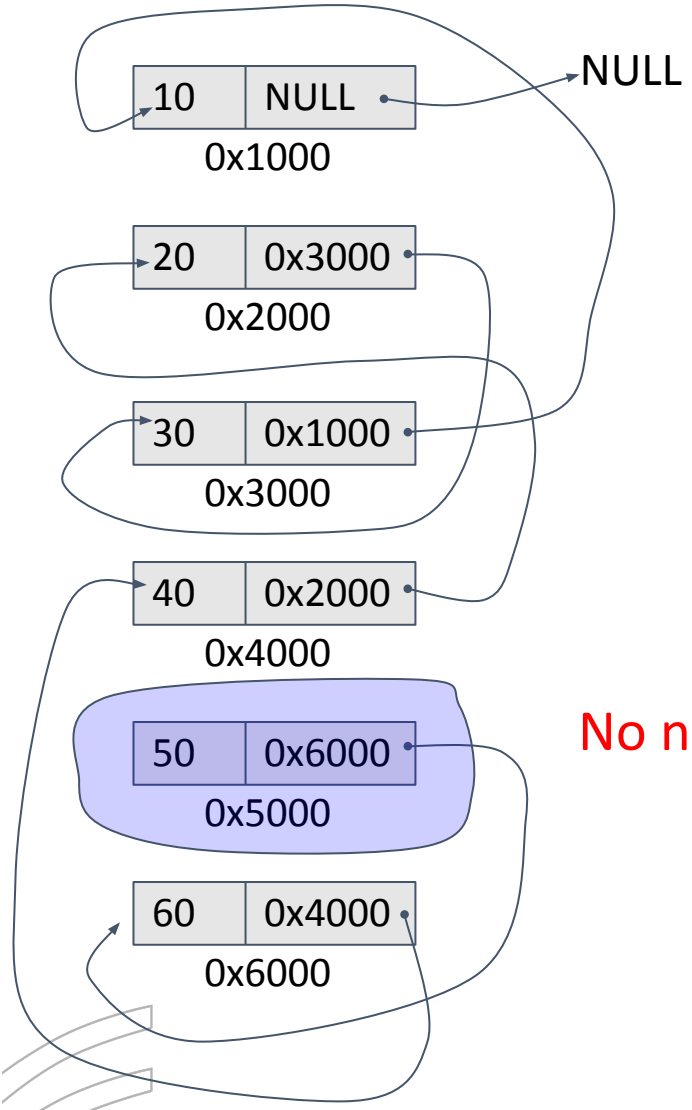
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction

- Step 3: Identify the head

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

Exercise: Linked List Construction



No node points to the head

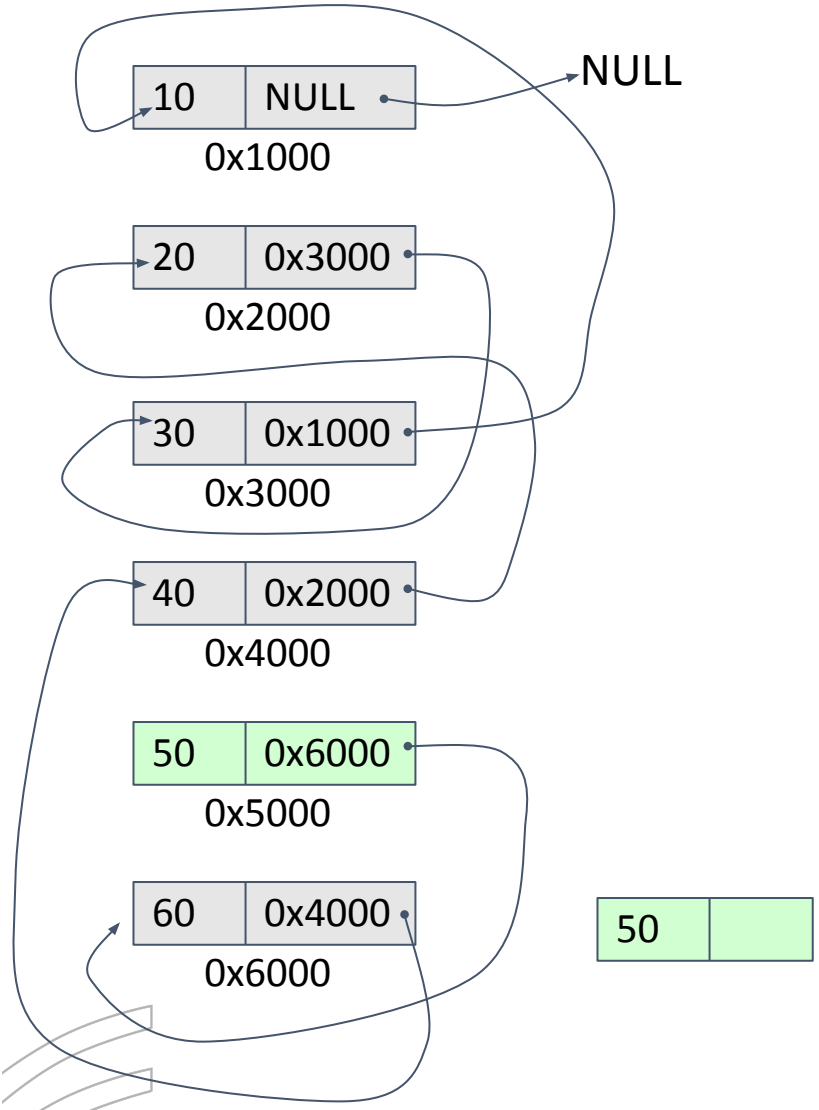
Memory Address	Data	Next Node Address
0x1000	10	NULL
0x2000	20	0x3000
0x3000	30	0x1000
0x4000	40	0x2000
0x5000	50	0x6000
0x6000	60	0x4000

Exercise: Linked List Construction

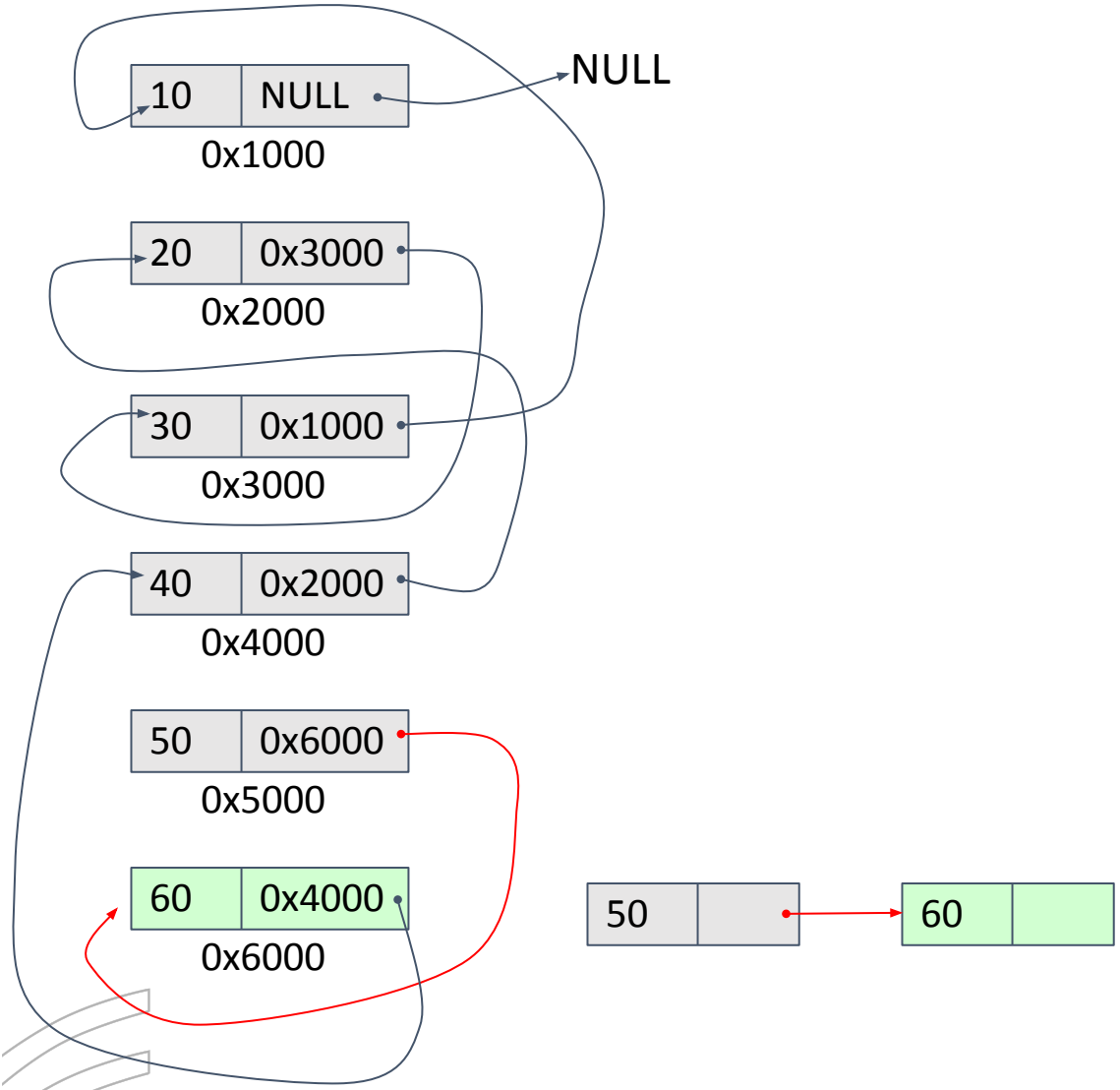
- Step 4: Traverse the list link by link from head and flatten it

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

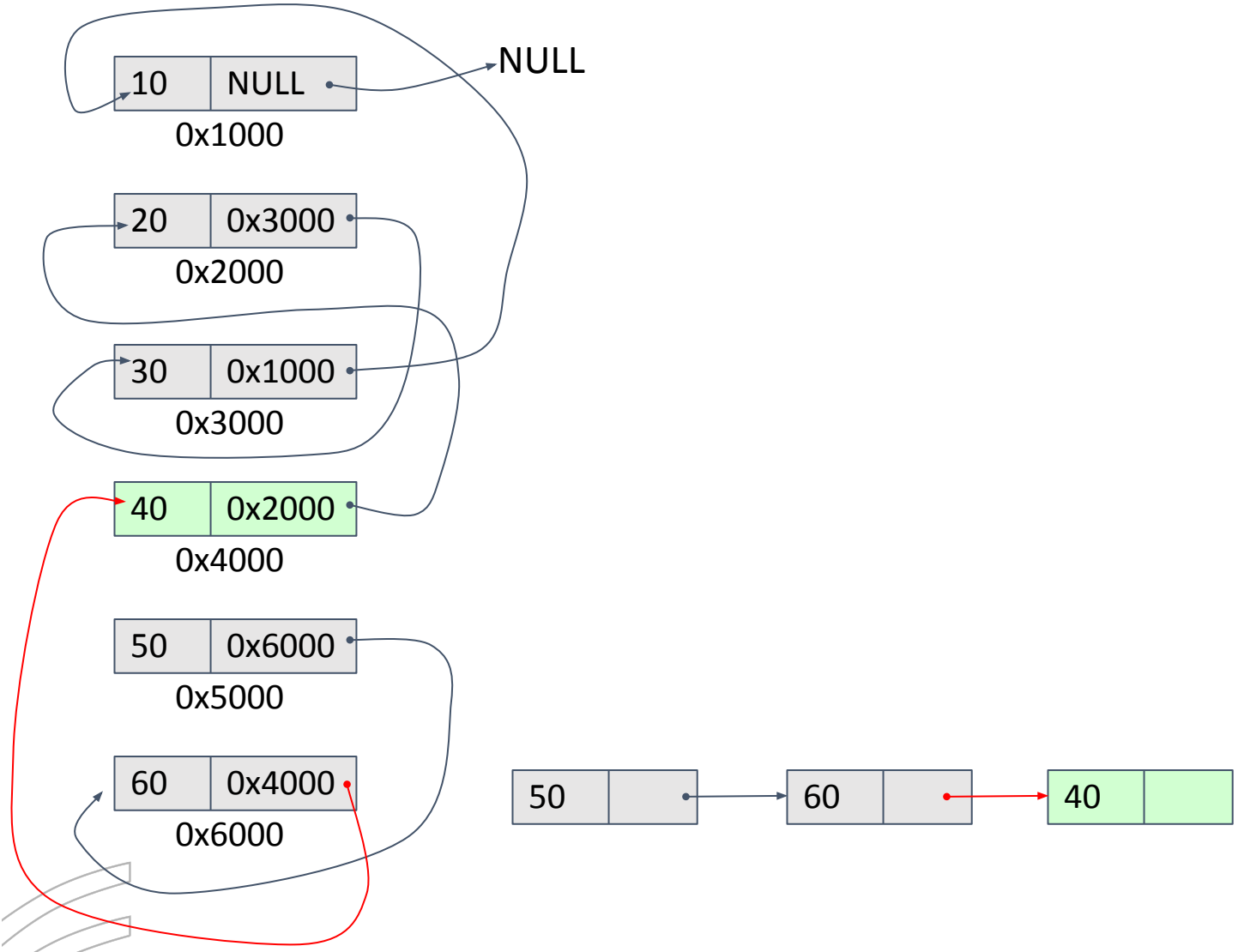
Exercise: Linked List Construction



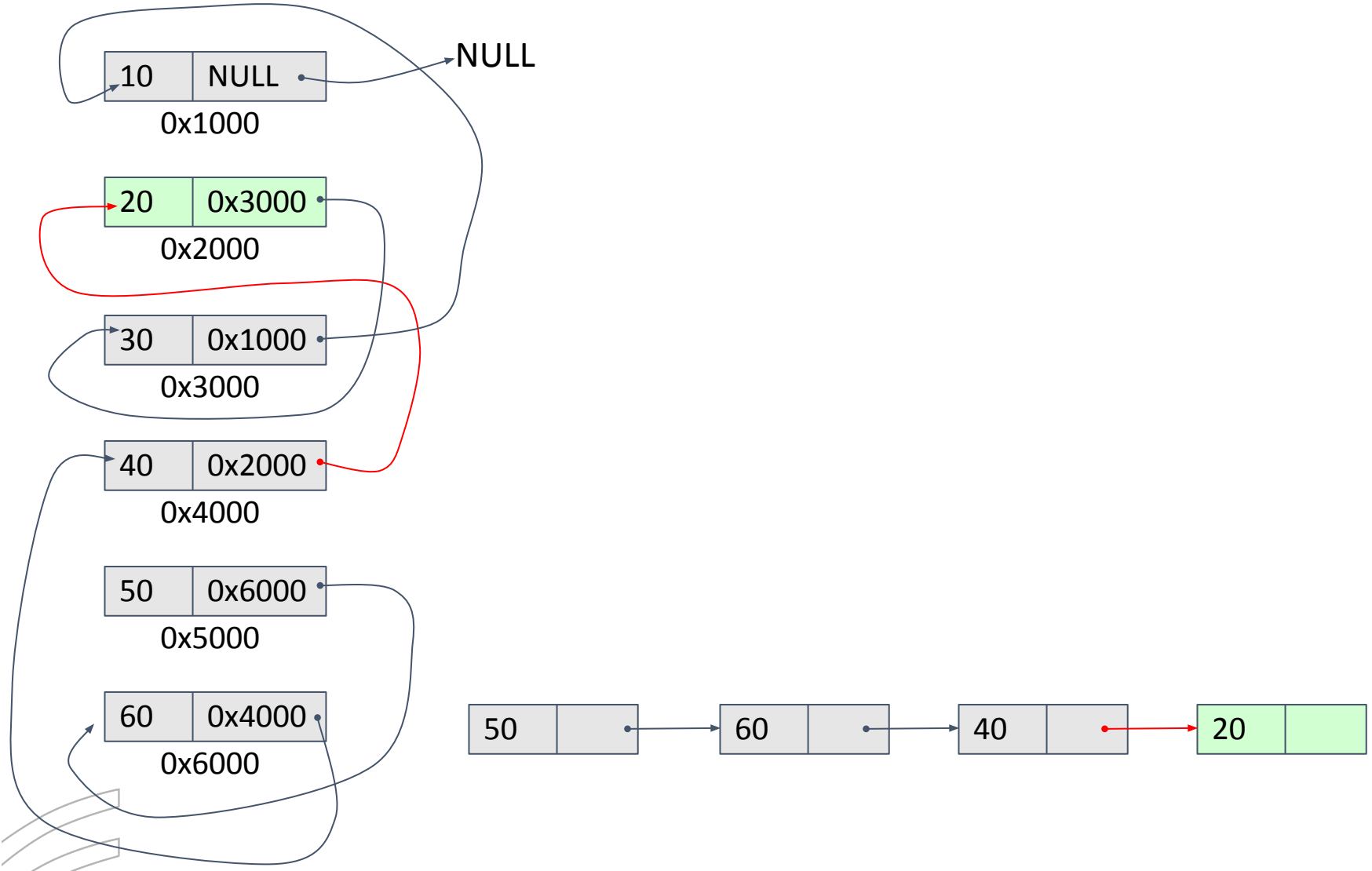
Exercise: Linked List Construction



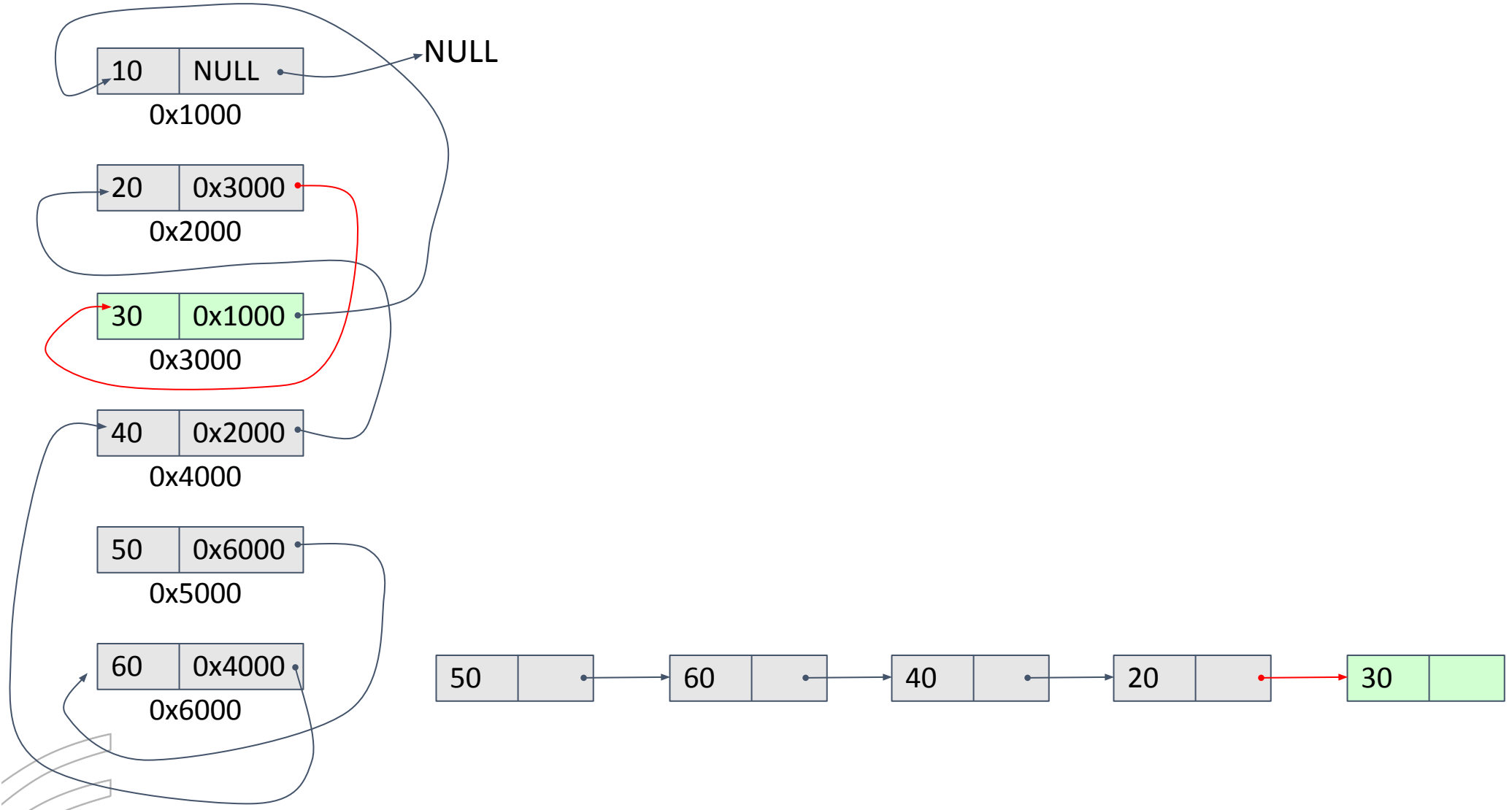
Exercise: Linked List Construction



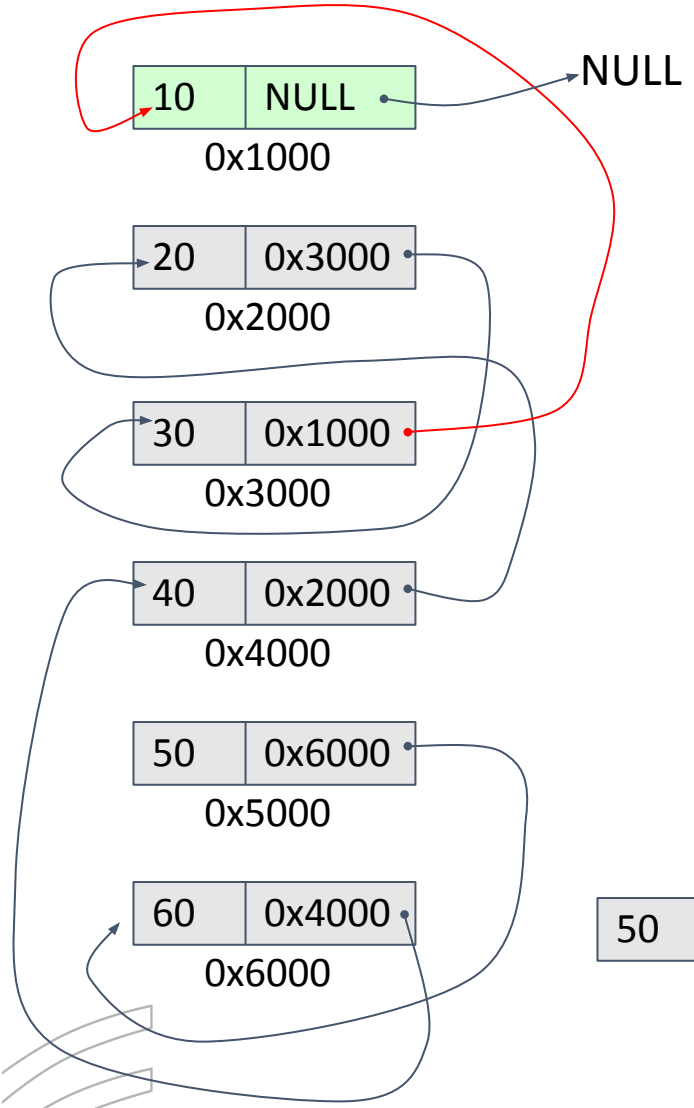
Exercise: Linked List Construction



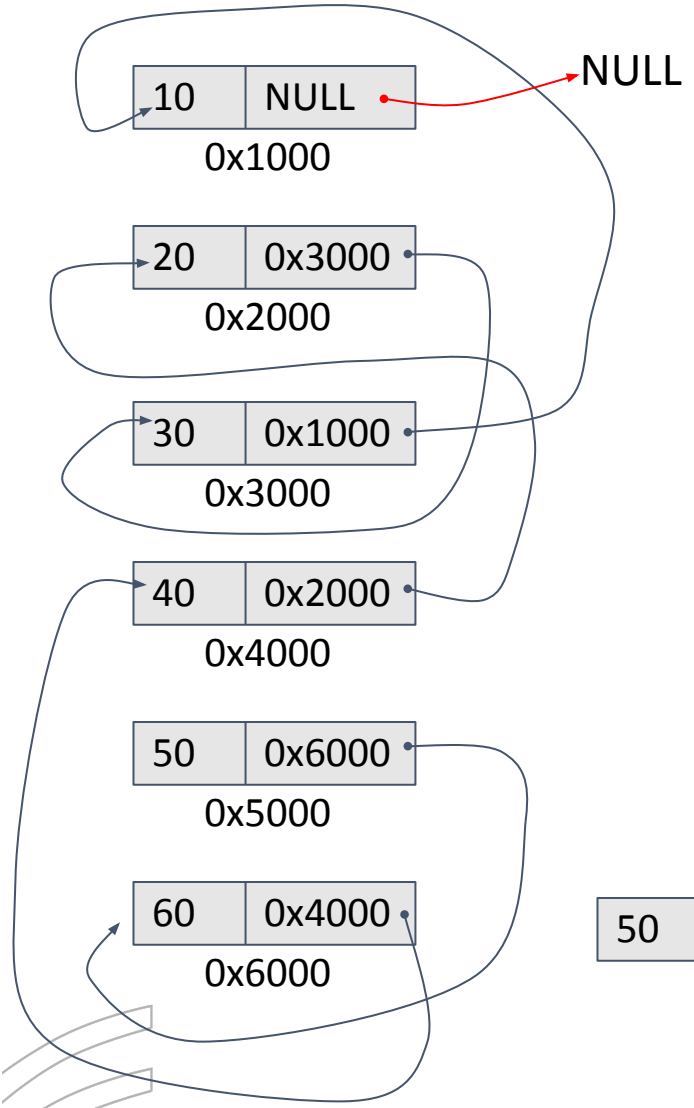
Exercise: Linked List Construction

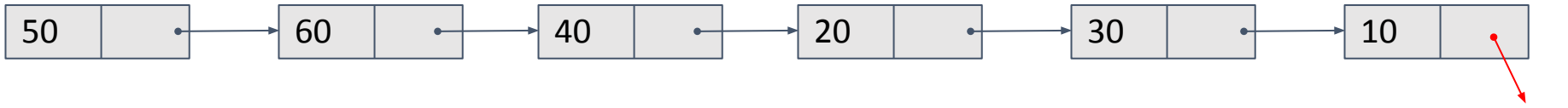


Exercise: Linked List Construction

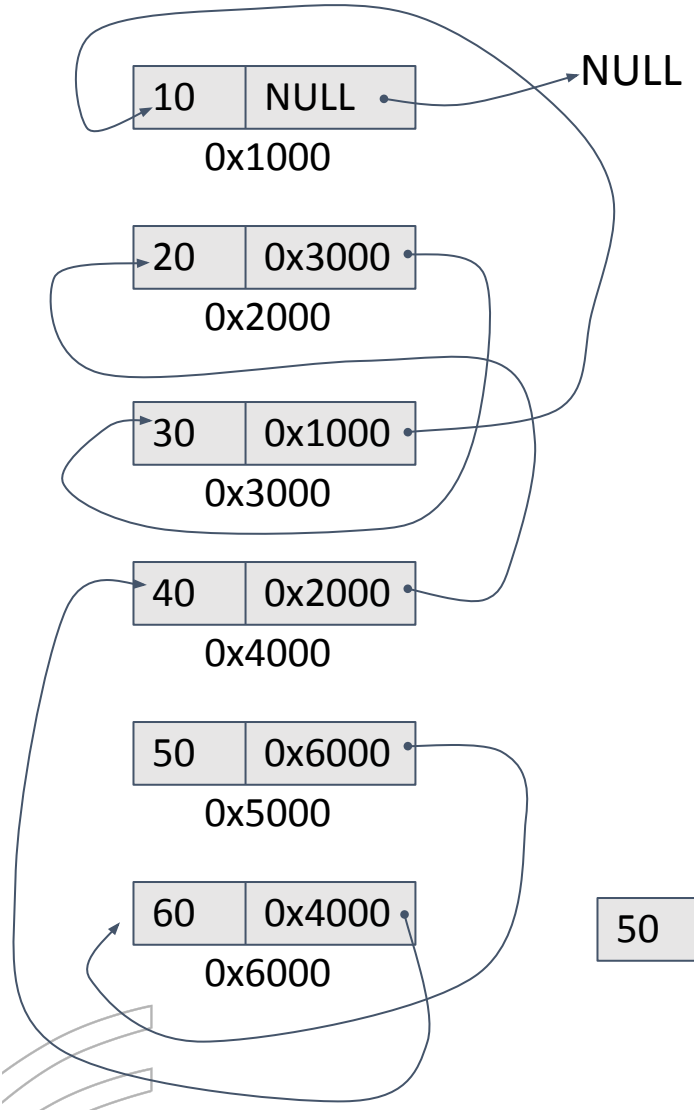


Exercise: Linked List Construction



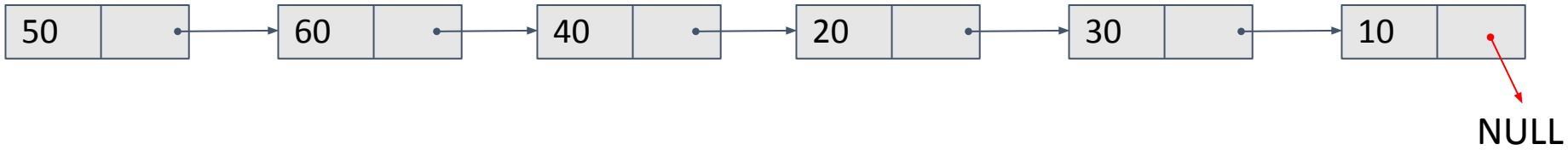


Exercise: Linked List Construction



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

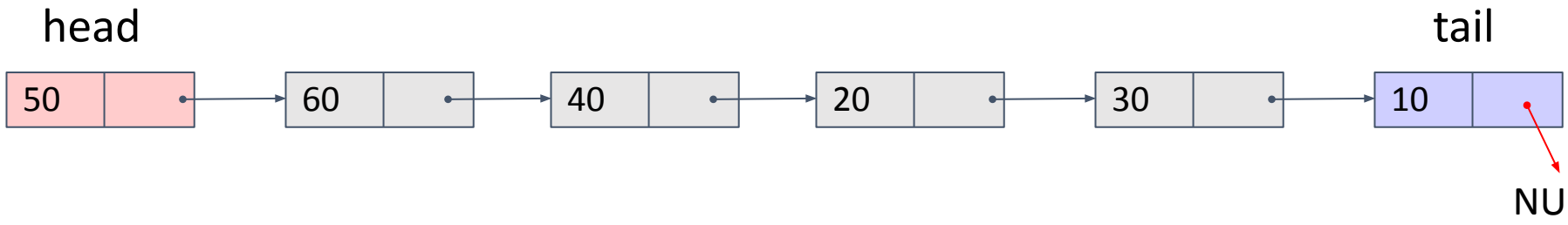
Computers don't use the neat arrows and boxes we draw—this is how linked lists actually look under the hood.



Element Removal (Front)

- Remove the front element (head)

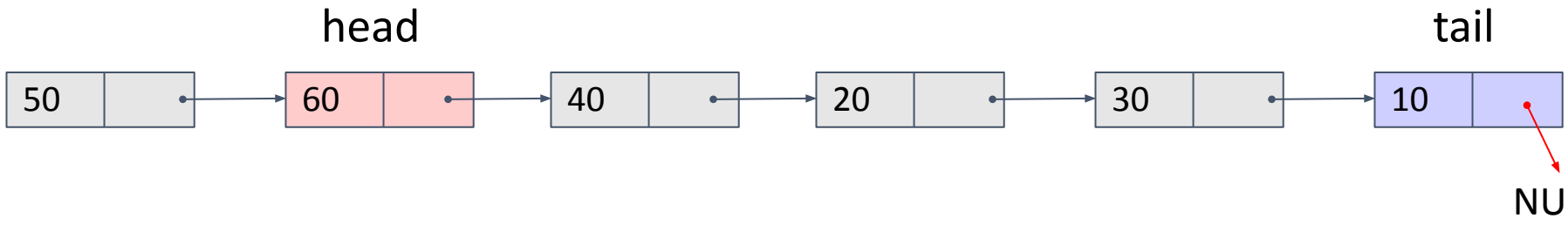
```
// Remove from the beginning
```



Element Removal (Front)

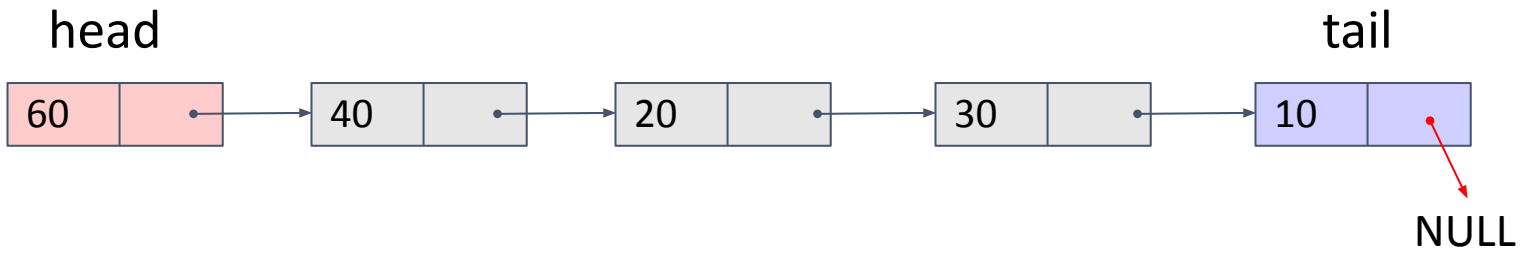
- Remove the front element (head)

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



Element Removal (Anywhere)

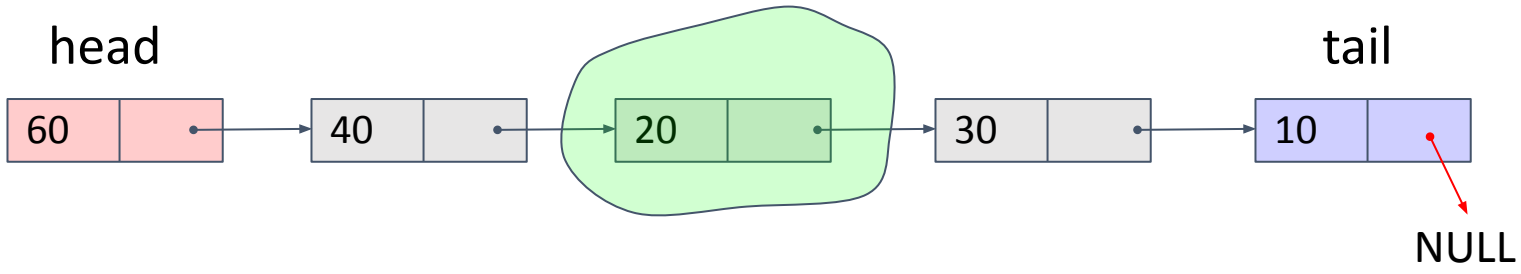
- Remove the element (30) at index 3



Element Removal (Anywhere)

- Remove the element (30) at index 3

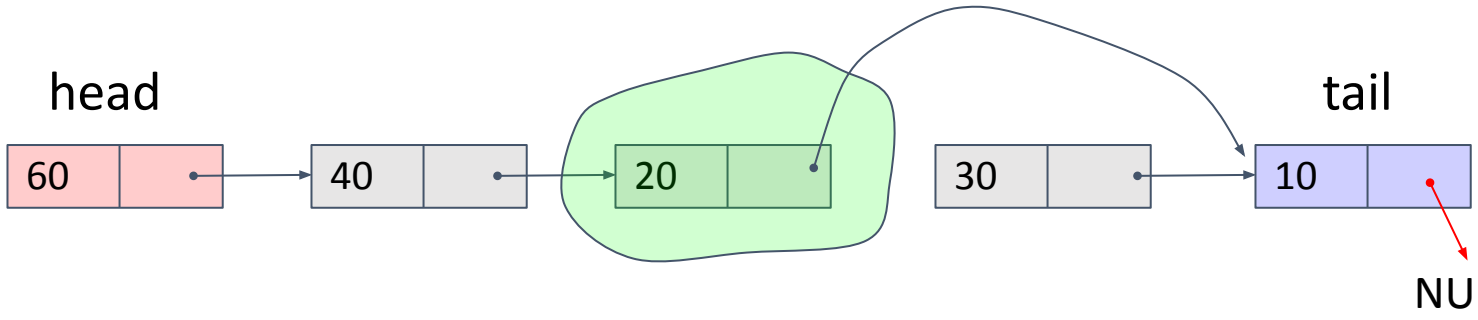
```
Node prev = getNode( index: index - 1);
```



Element Removal (Anywhere)

- Remove the element (30) at index 3

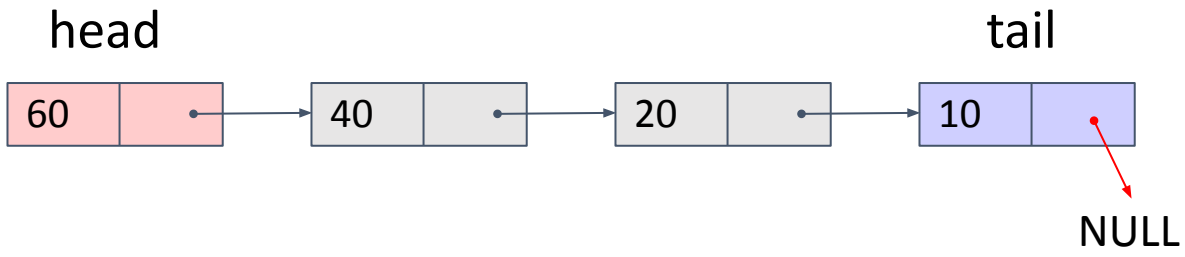
```
Node prev = getNode( index: index - 1);

    prev.next = prev.next.next;
```



Element Removal (Anywhere)

- Remove the element (30) at index 3

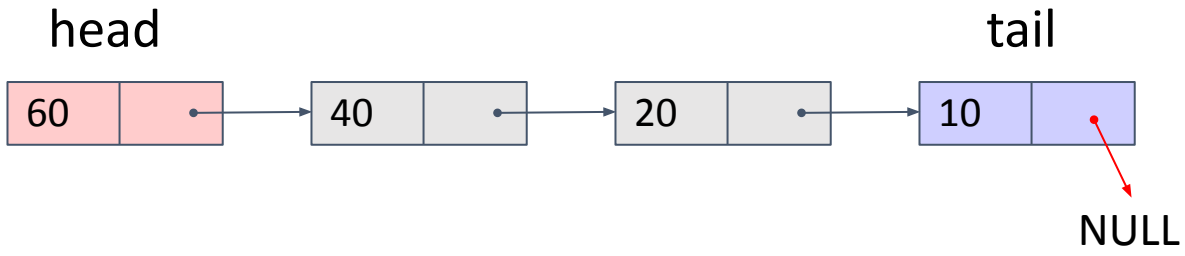
```
Node prev = getNode( index: index - 1);

    prev.next = prev.next.next;
```



Element Removal (Anywhere)

- Remove the element (30) at index 3

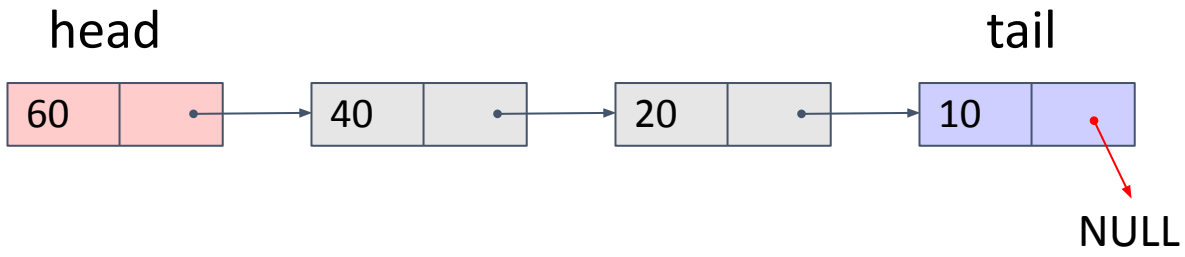
```
Node prev = getNode( index: index - 1);
if (prev != null && prev.next != null) { / Cannot remove before head or after tail
    prev.next = prev.next.next;
}
```



Element Removal (Anywhere)

- Can we optimize the tail removal?

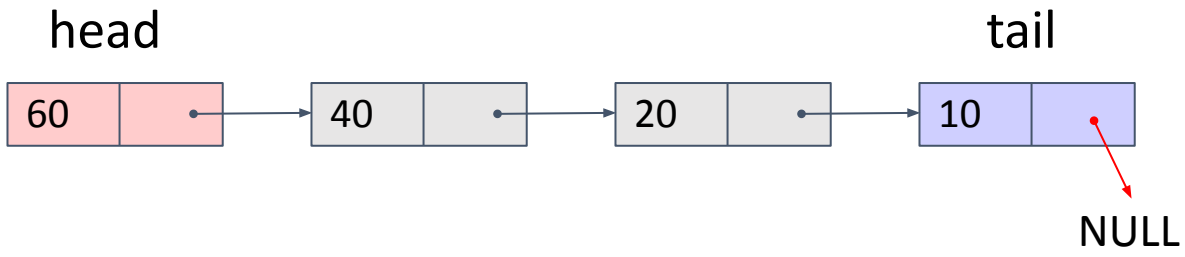
```
Node prev = getNode( index: index - 1);
if (prev != null && prev.next != null) { / Cannot remove before head or after tail
    prev.next = prev.next.next;
}
```



Element Removal (Anywhere)

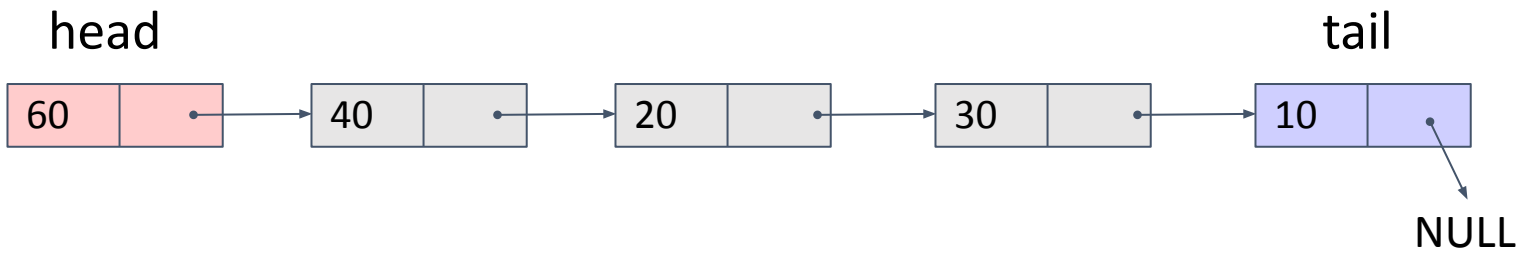
- Can we optimize the tail removal?
  - No, we must be able to access the node before the last node in order to remove the last node.

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



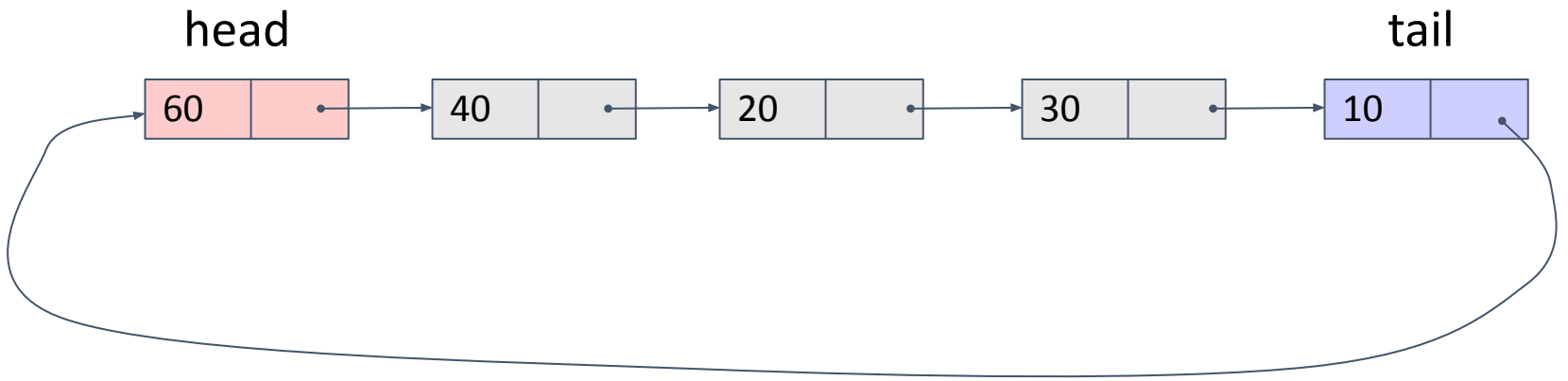
List Rotation (Left)

Rotate the list left by 1 element



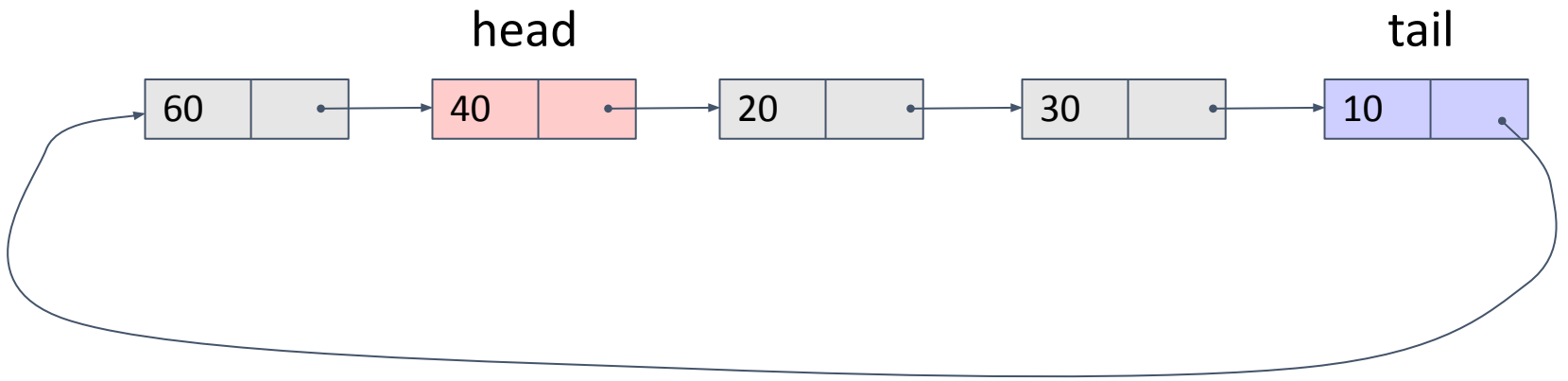
List Rotation (Left)

Rotate the list left by 1 element



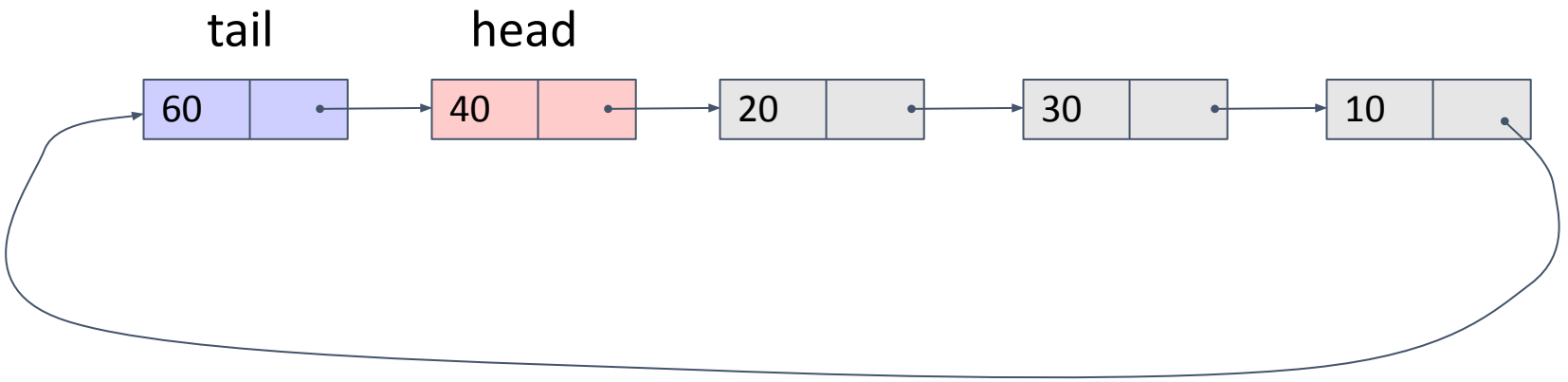
List Rotation (Left)

Rotate the list left by 1 element



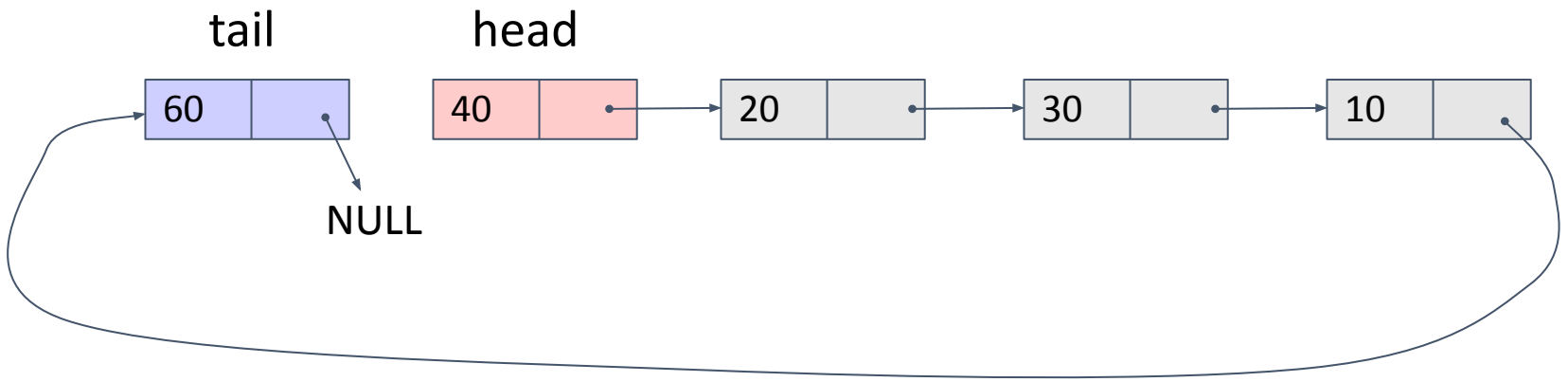
List Rotation (Left)

Rotate the list left by 1 element



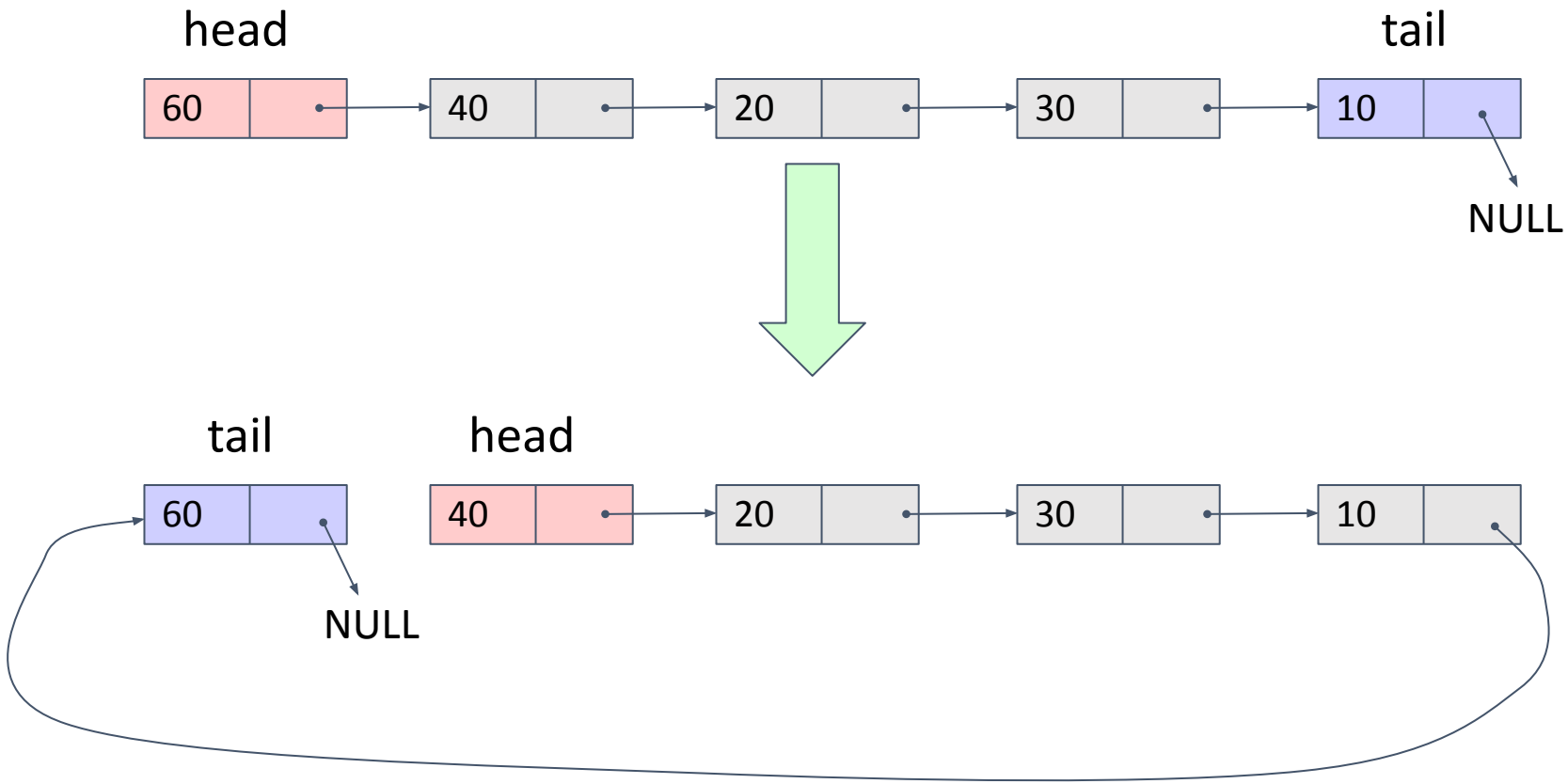
List Rotation (Left)

Rotate the list left by 1 element



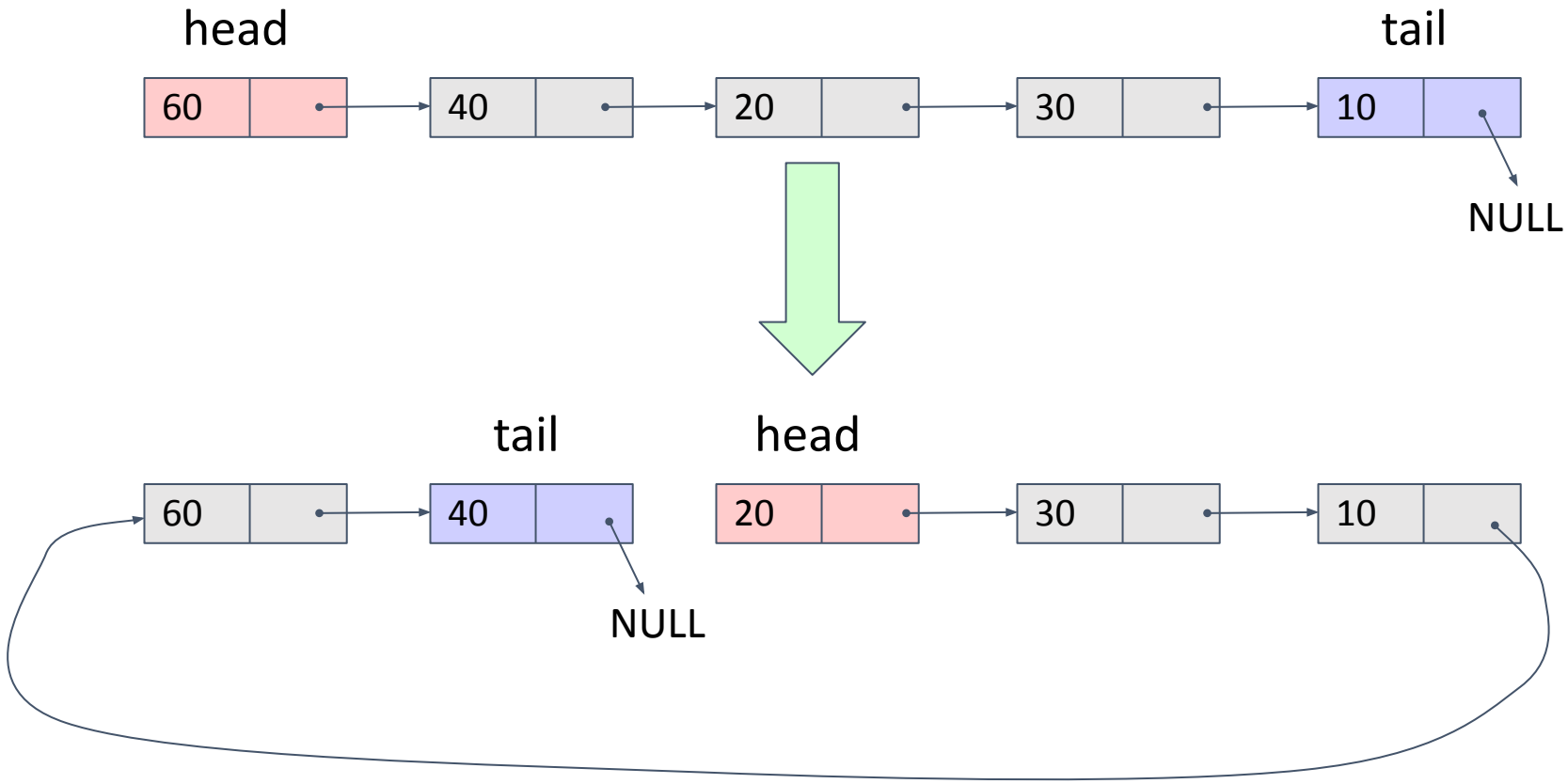
List Rotation (Left)

Rotate the list left by 1 element



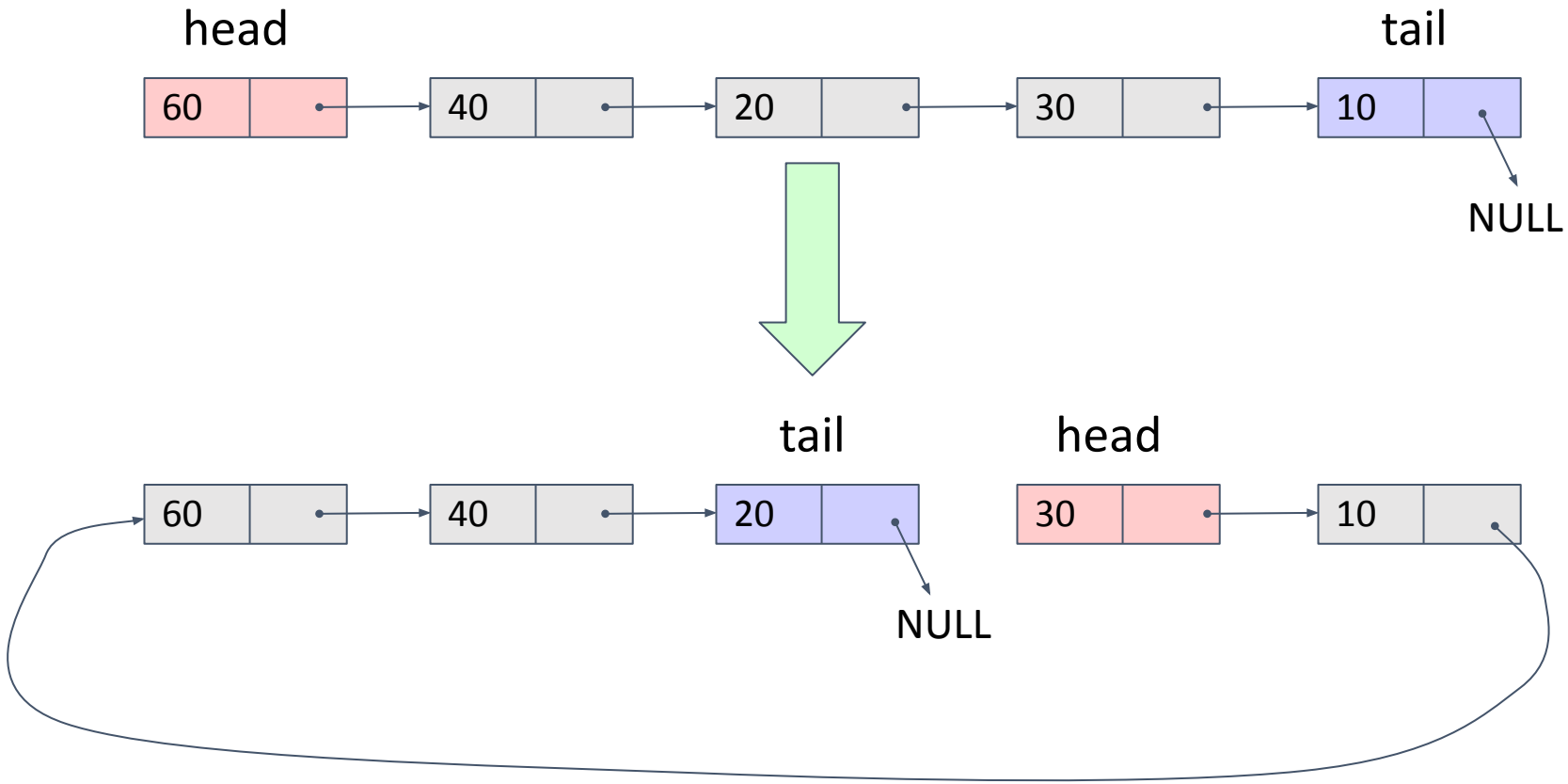
List Rotation (Left)

Rotate the list left by 2 elements



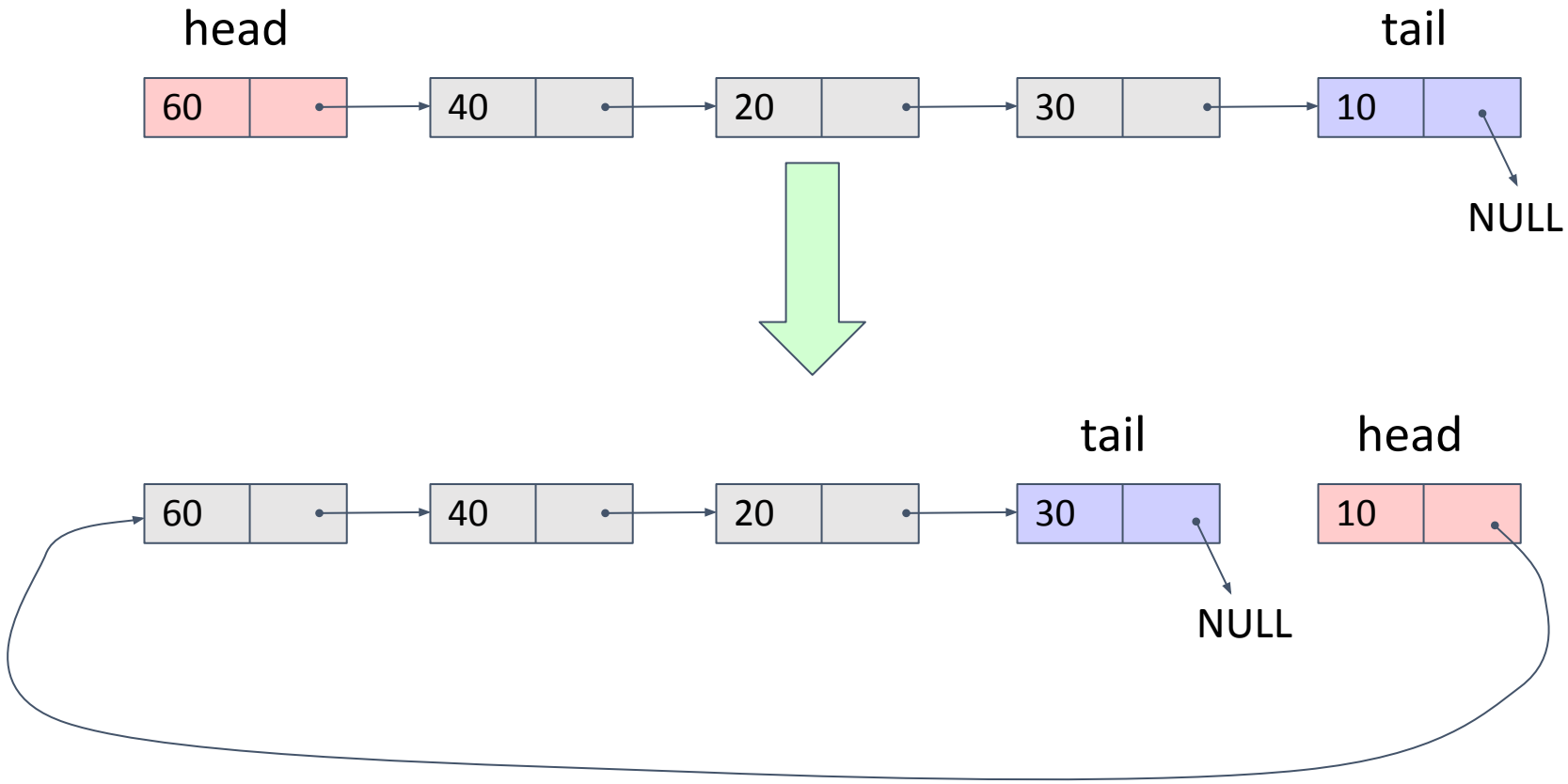
List Rotation (Left)

Rotate the list left by 3 elements



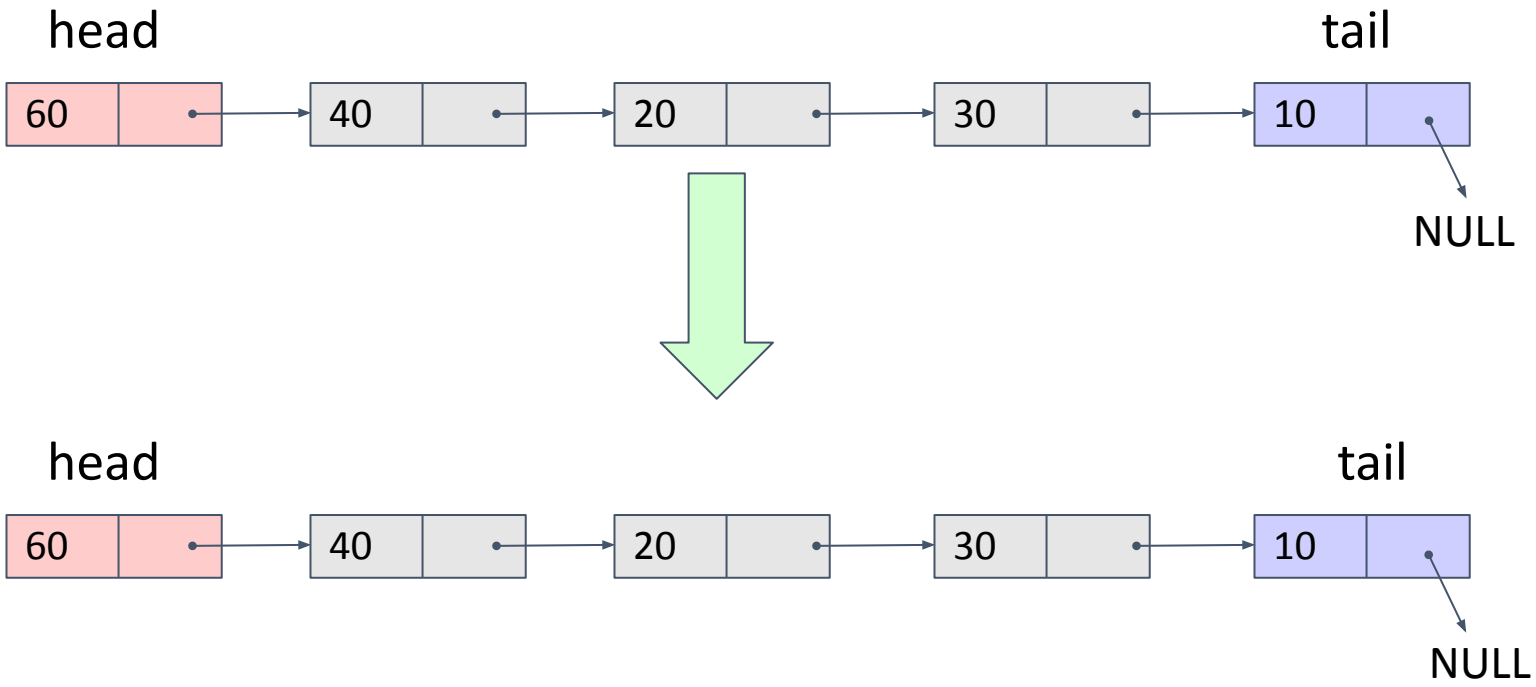
List Rotation (Left)

Rotate the list left by 4 elements



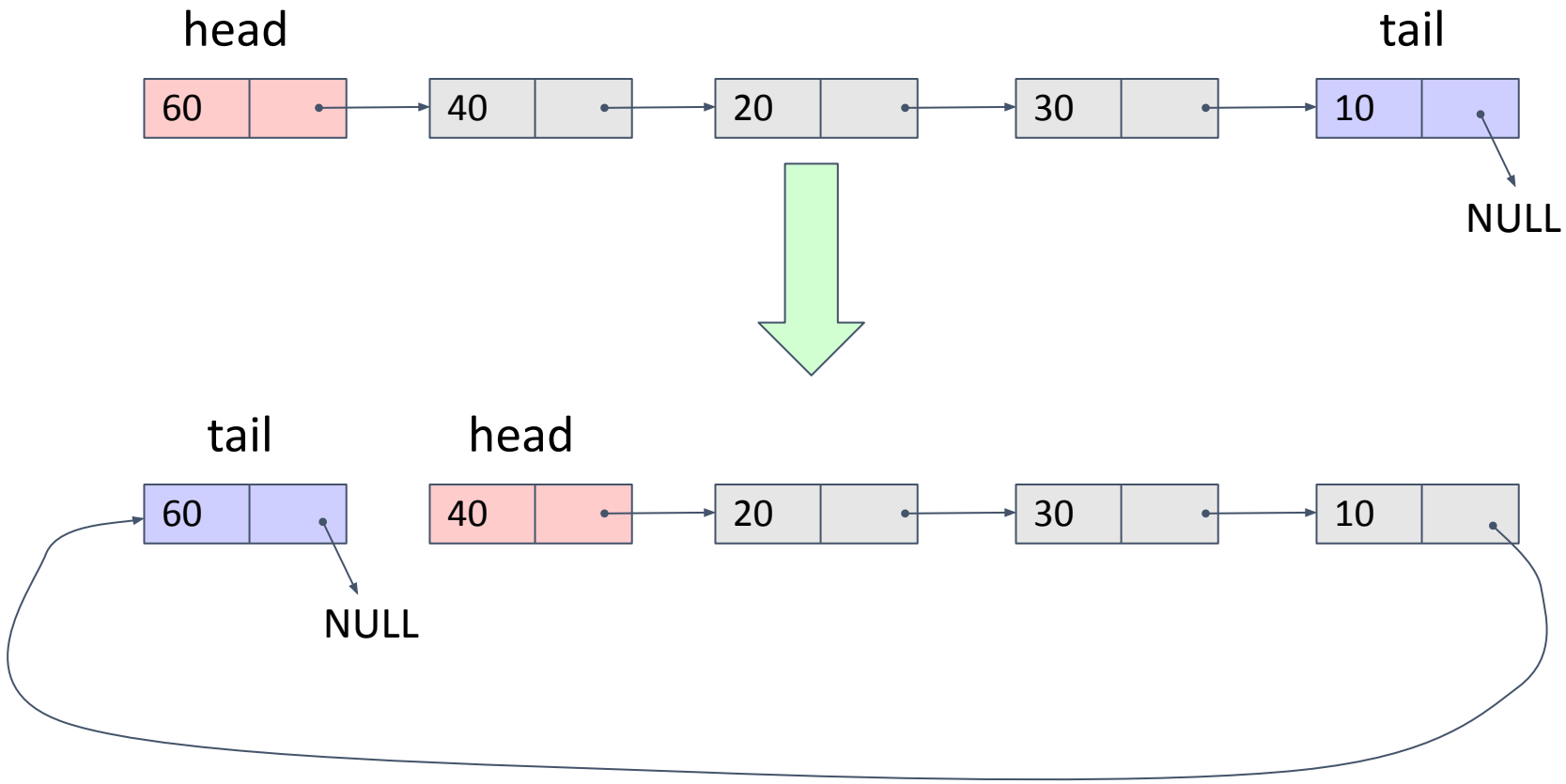
List Rotation (Left)

Rotate the list left by 5 elements



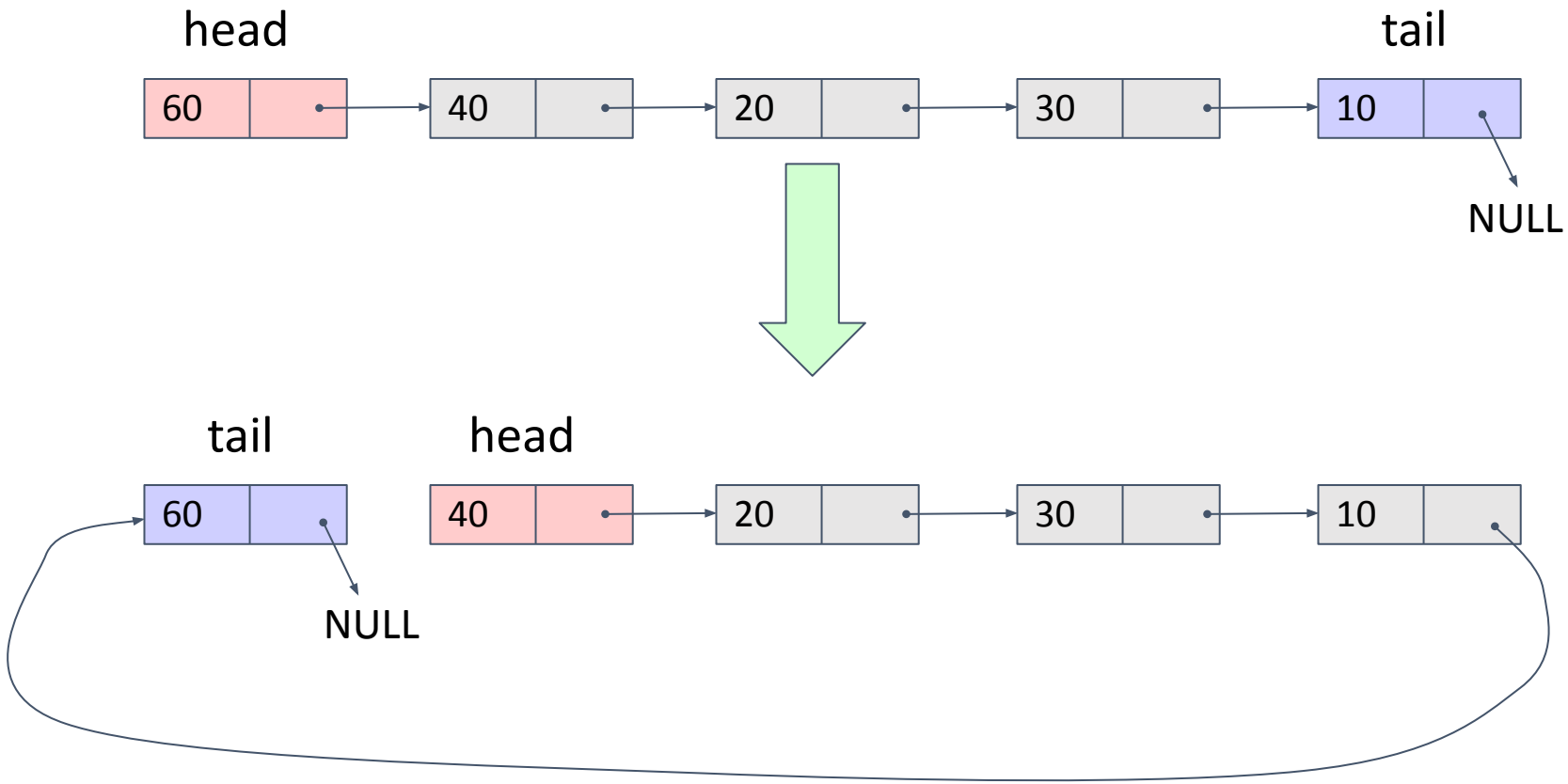
List Rotation (Left)

Rotate the list left by 6 elements



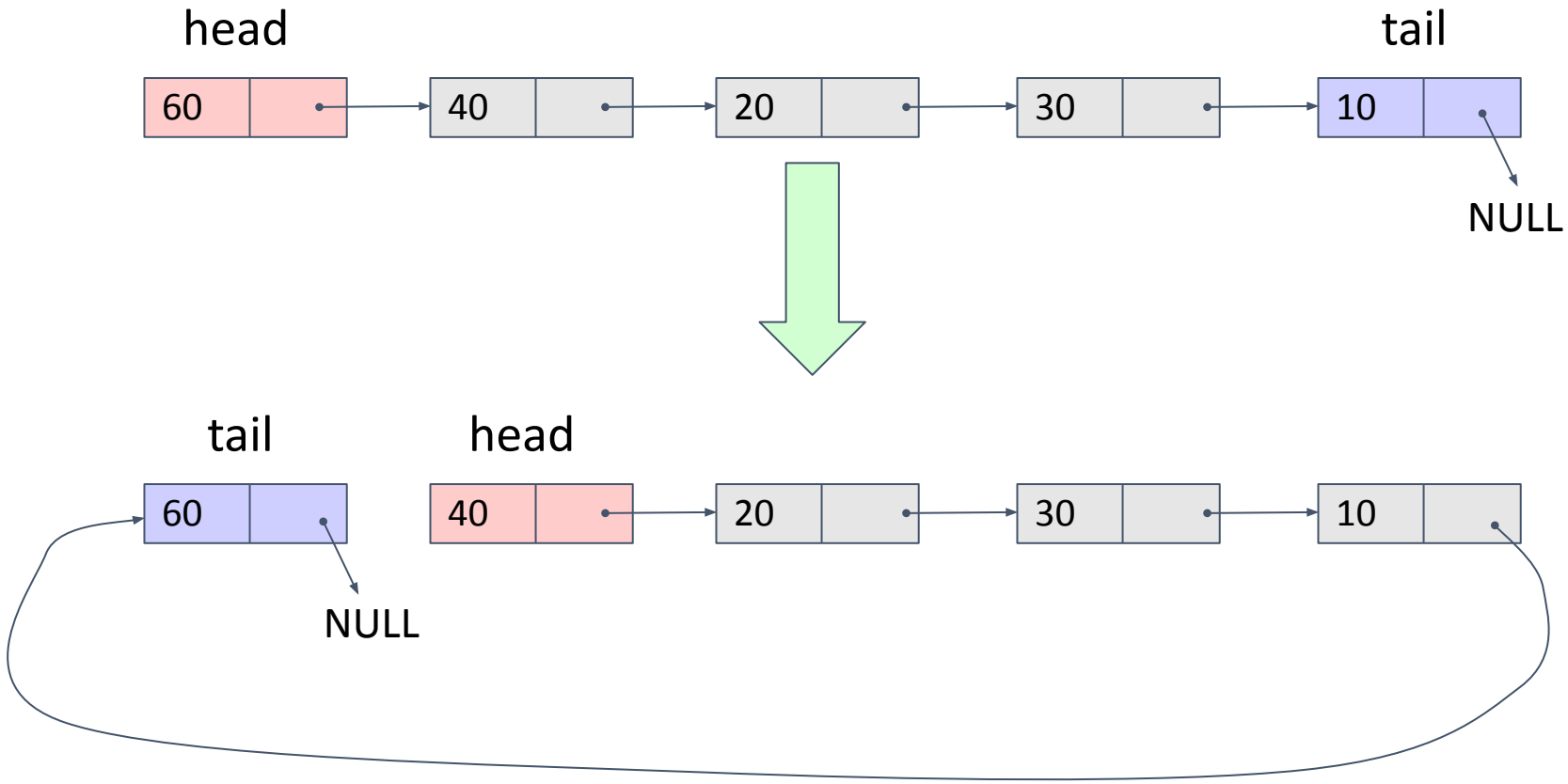
List Rotation (Left)

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



List Rotation (Left)

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



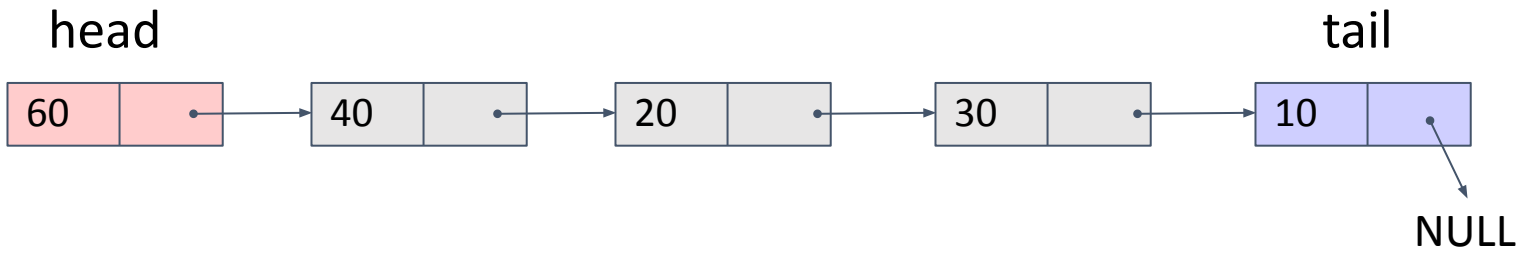
List Rotation (Left)

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

List Rotation (Left) Code Walkthrough

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



}
```

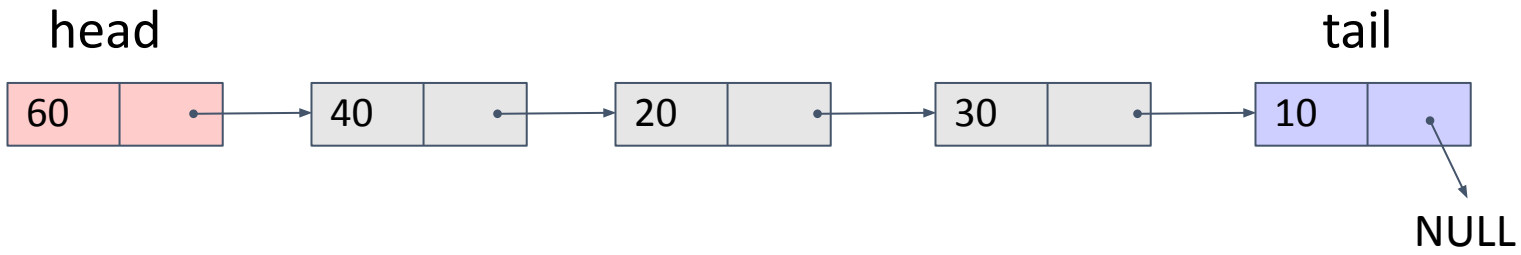
Rotate the list left
by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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;



}
```

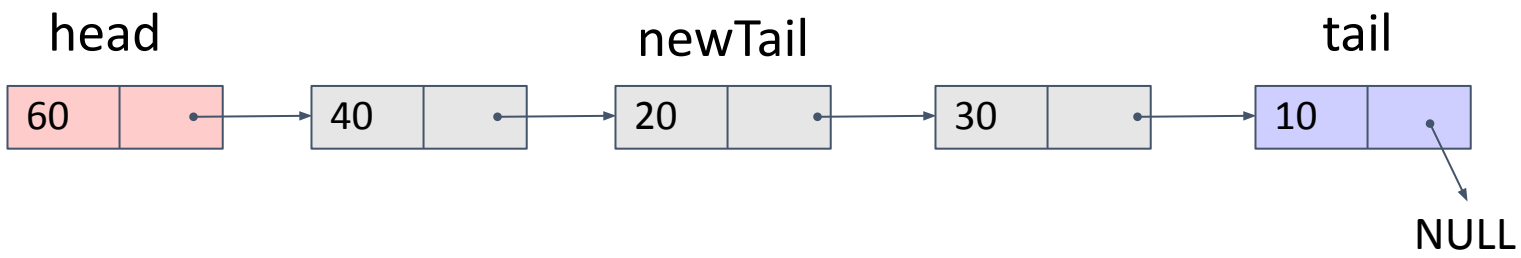
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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);

}
```

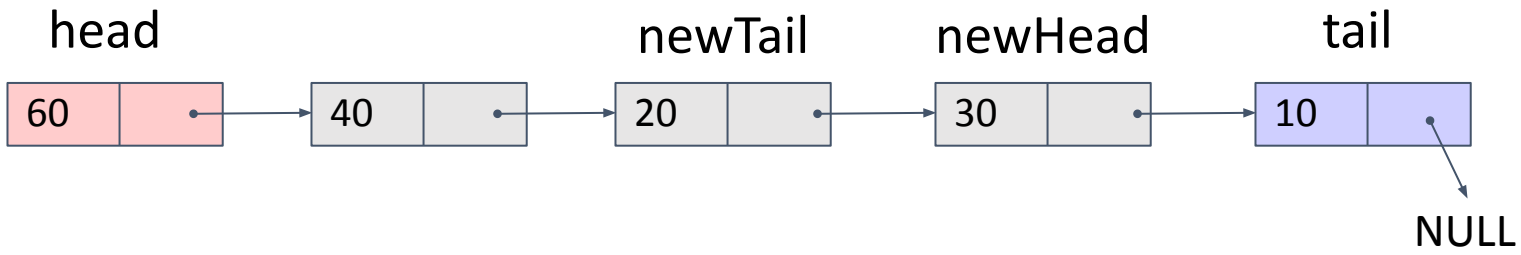
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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




}
```

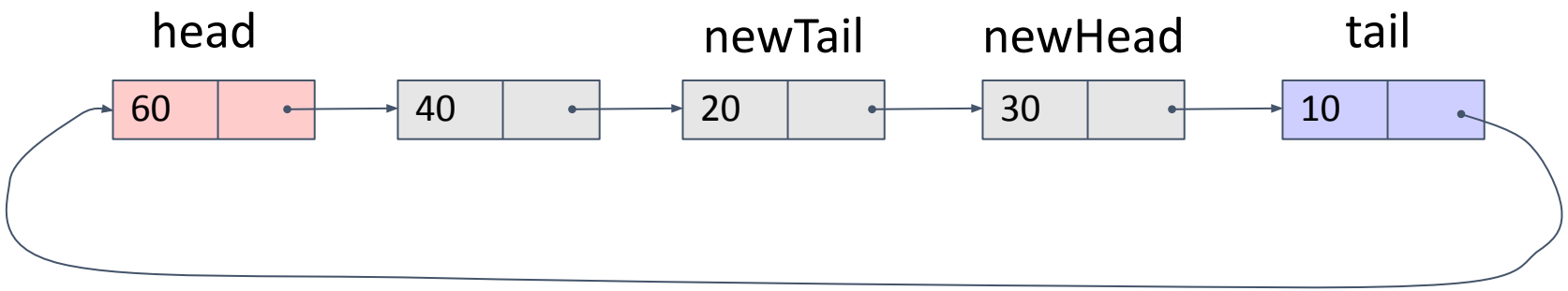
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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


}
```

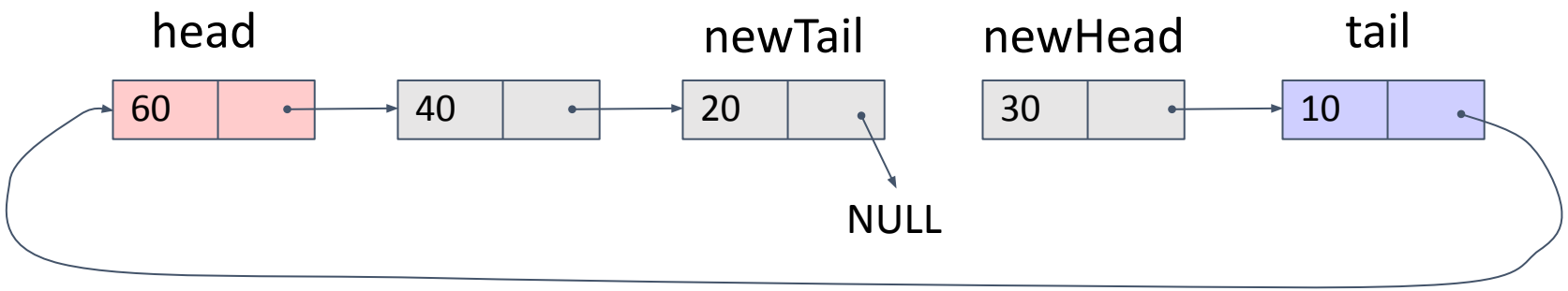
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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


}
```

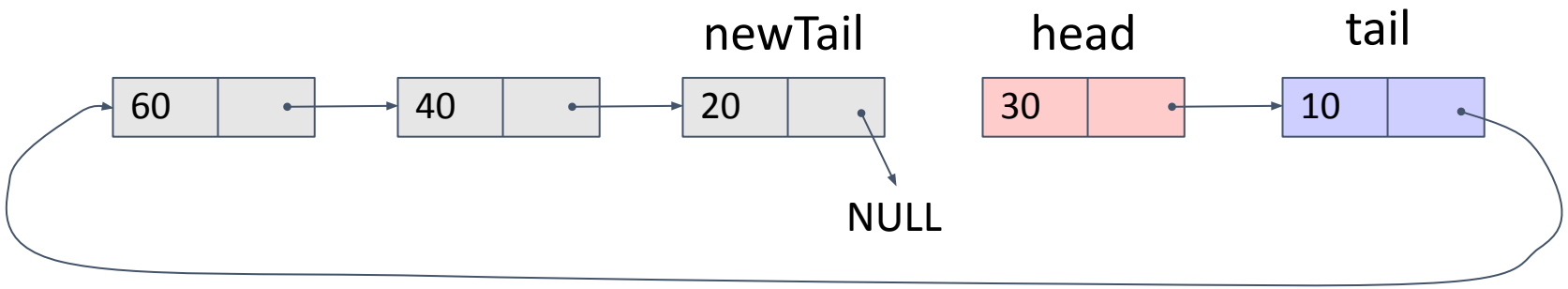
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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

}
```

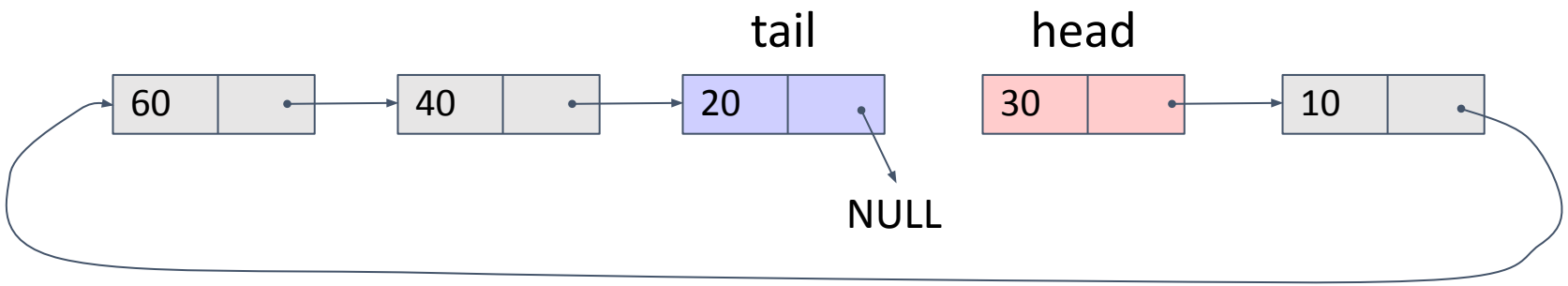
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

```java
// 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
}
```

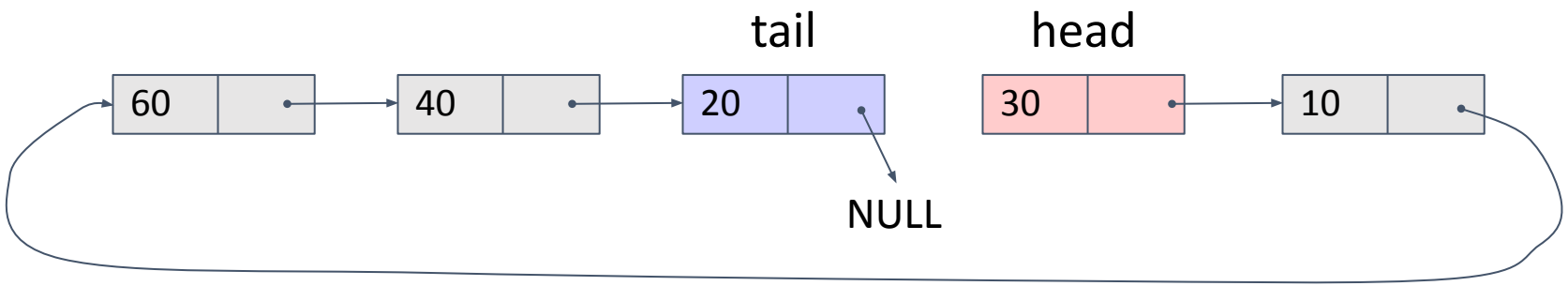
Rotate the list left by 3 elements



List Rotation (Left) Code Walkthrough

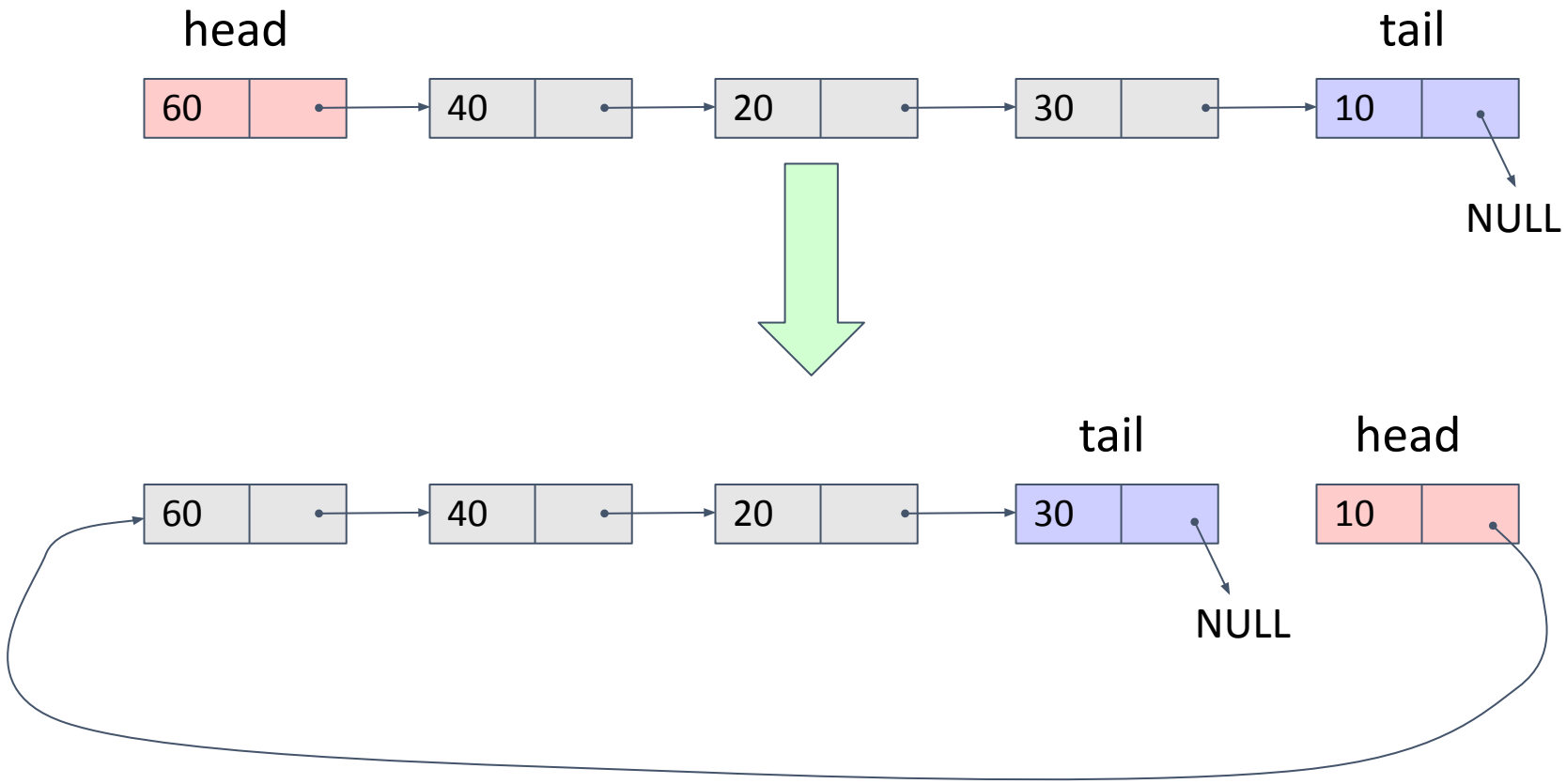
```java
// 13. Rotating the list left
public void rotateLeft(int k) {
    if (head == null || k <= 0) return; // Boundary checks
    int size = count();
    k = k % size; // Handle rotations greater than size
    if (k == 0) return;

    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
}
```

Rotate the list left by 3 elements



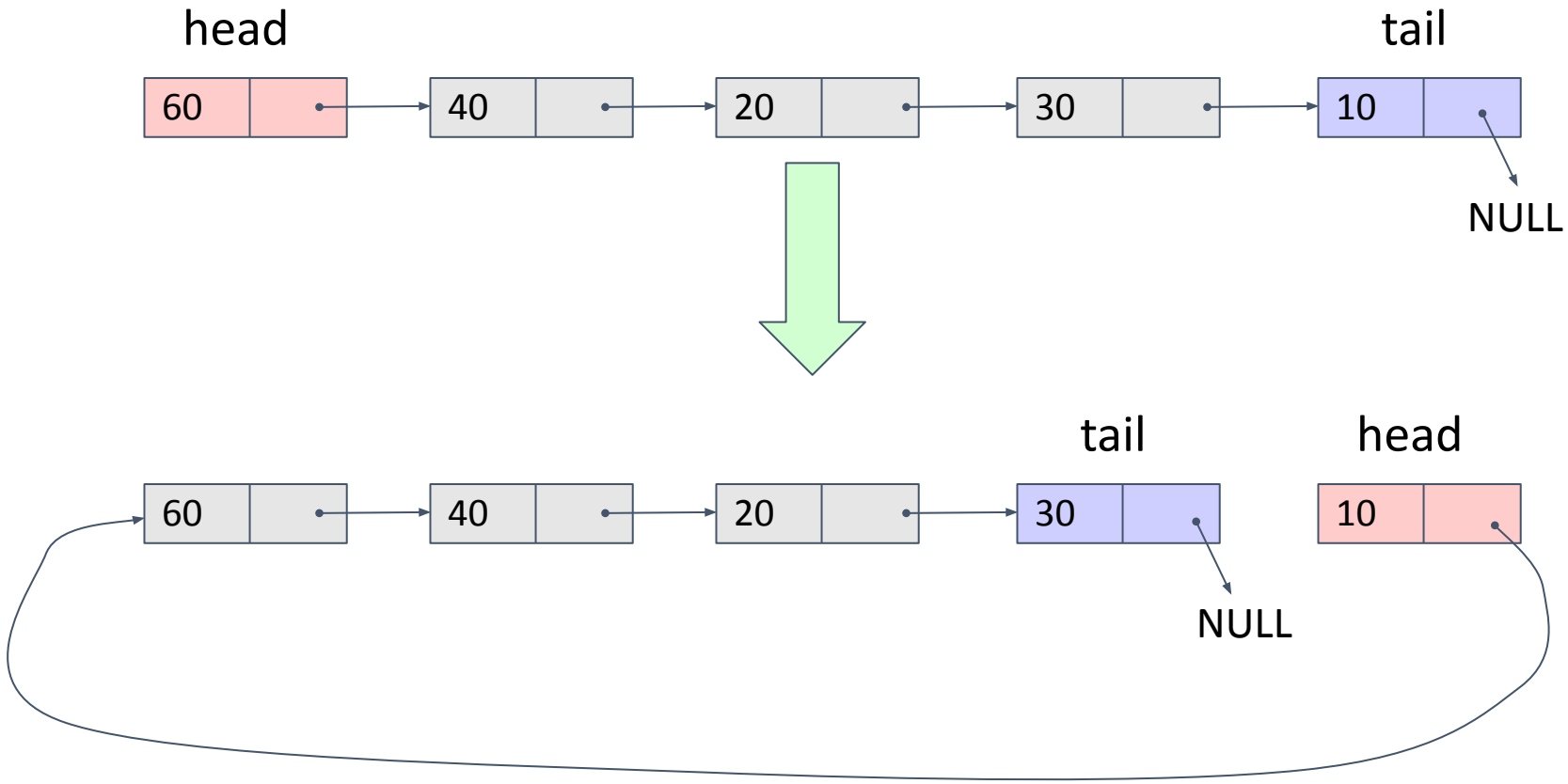
List Rotation (Right)

Rotate **left** by 4 elements



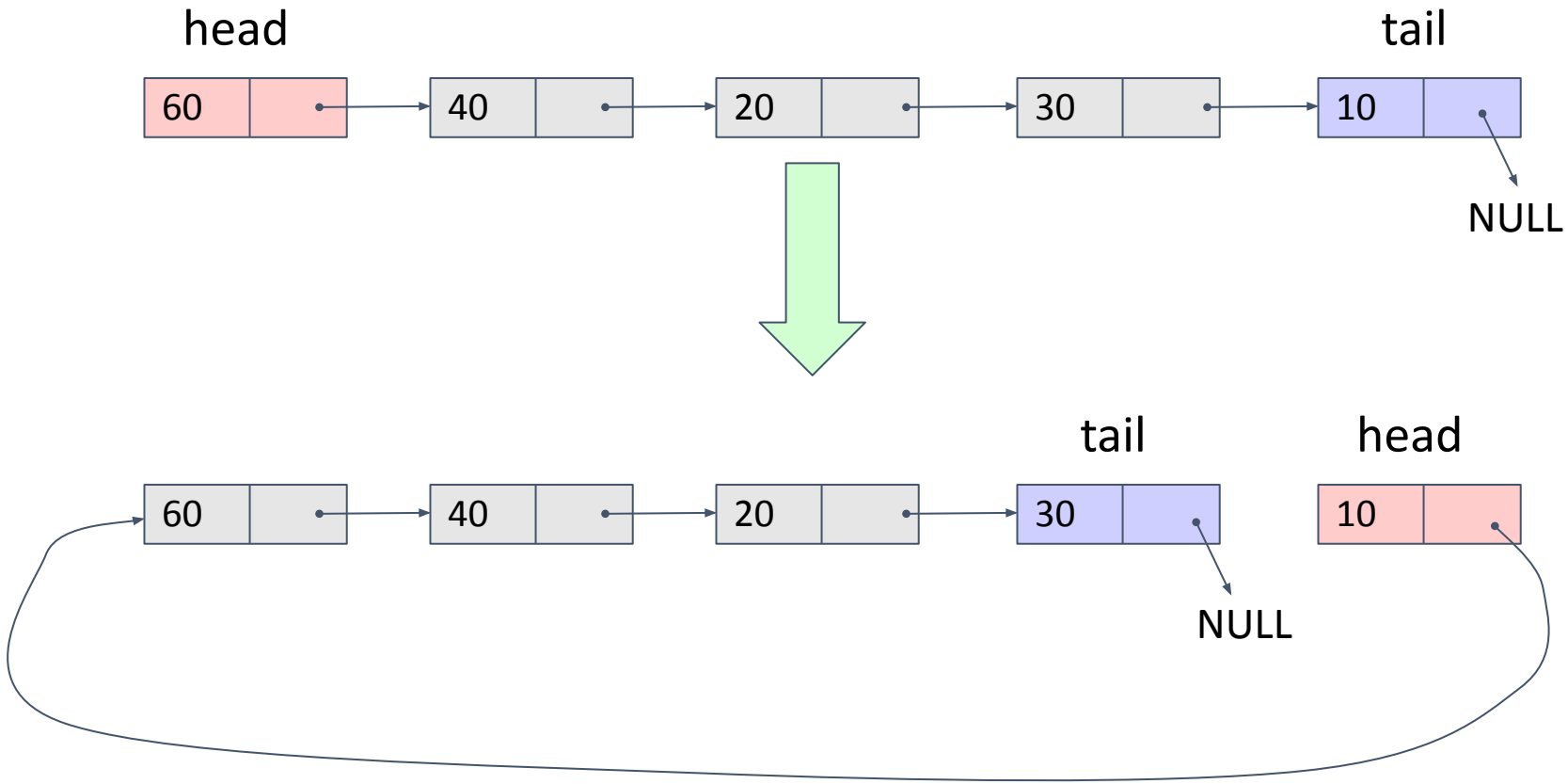
List Rotation (Right)

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



List Rotation (Right)

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



List Rotation (Right)

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

```java
// 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);
}
```

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 ⇒ O(1), Linked lists ⇒ 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