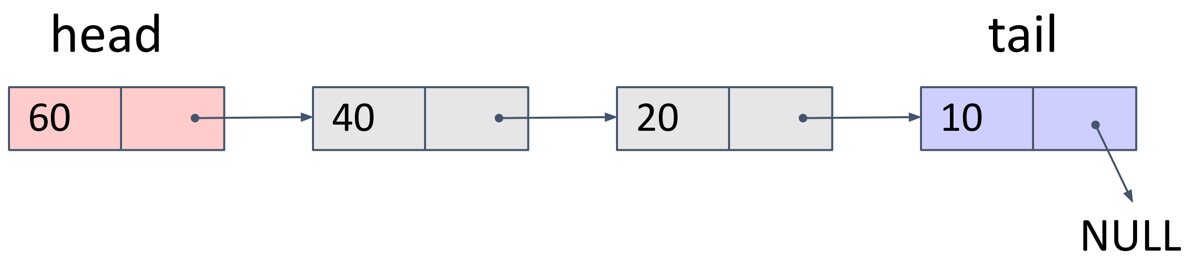
CSE 220
Data Structures

Lecture 04:
Doubly Linked Lists

Anwarul Bashir Shuaib [AWBS]
Lecturer
Department of Computer Science and Engineering
BRAC University

Review: Singly Linked Lists

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

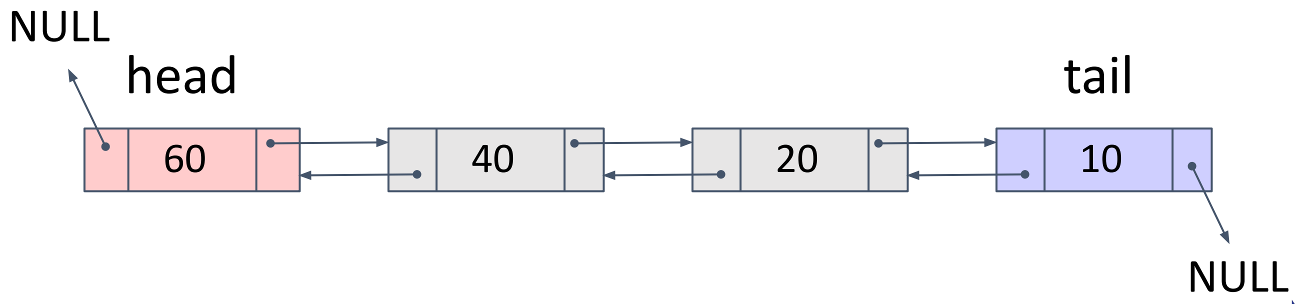


- This version is also known as "Singly Linked Lists (SLL)" as each node only contains the reference to the **next node**.

Doubly Linked Lists

- Each node contains **three** things:
  - Data
  - A reference to the **next** node
  - A reference to the **previous** node

- This version is also known as "Doubly Linked Lists (DLL)" as each node contains the reference to both the **next node and the previous node**.



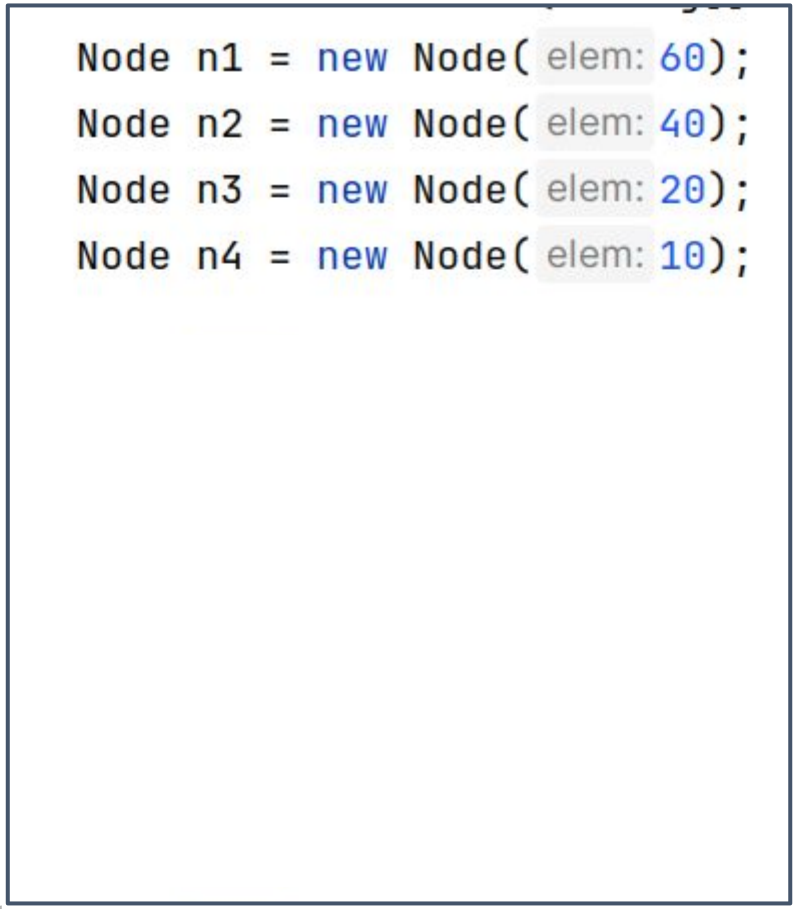
Node in DLLs

Doubly Linked List

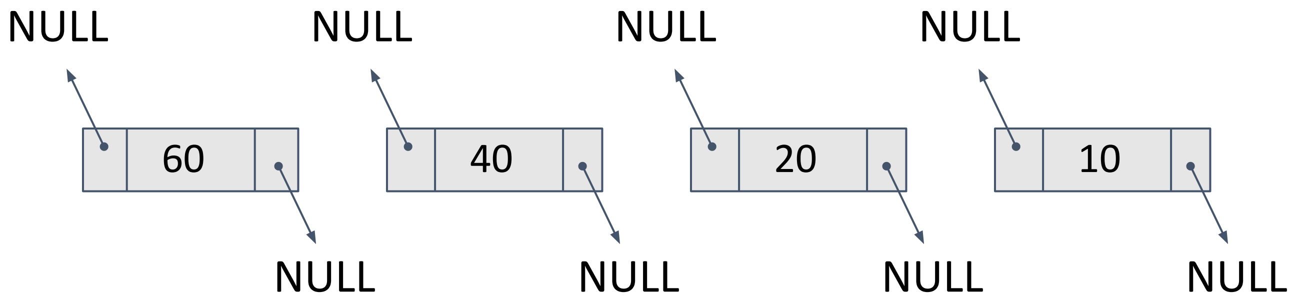
```
private static class Node {
    int elem;
    Node next;
    Node prev;

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

```
private Node head;
private Node tail;
```

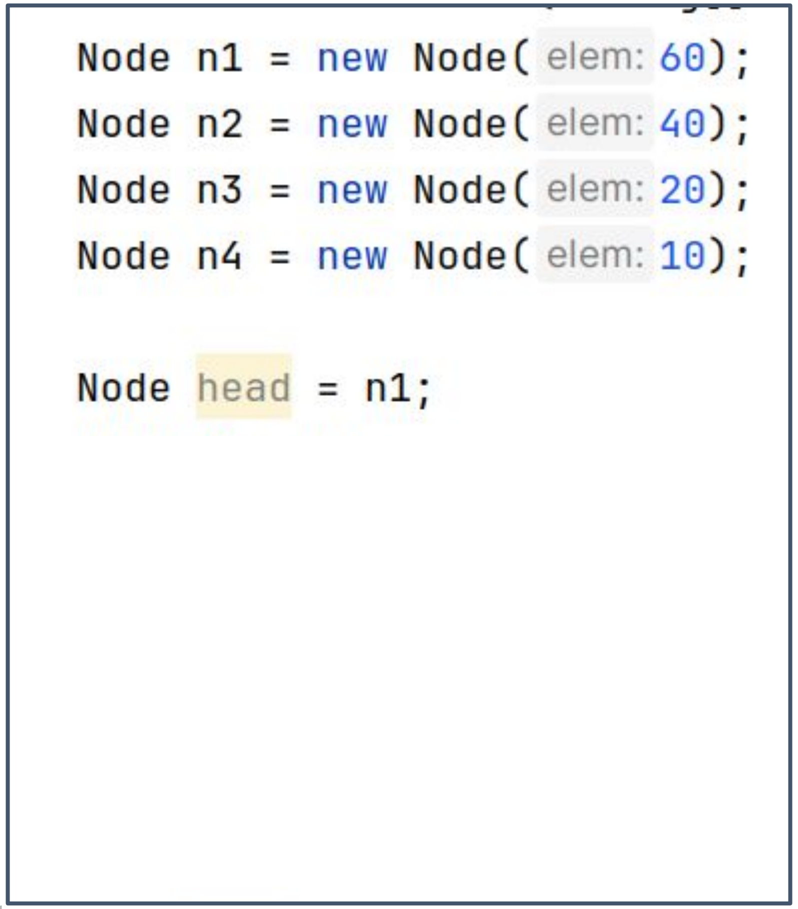
DLL Creation



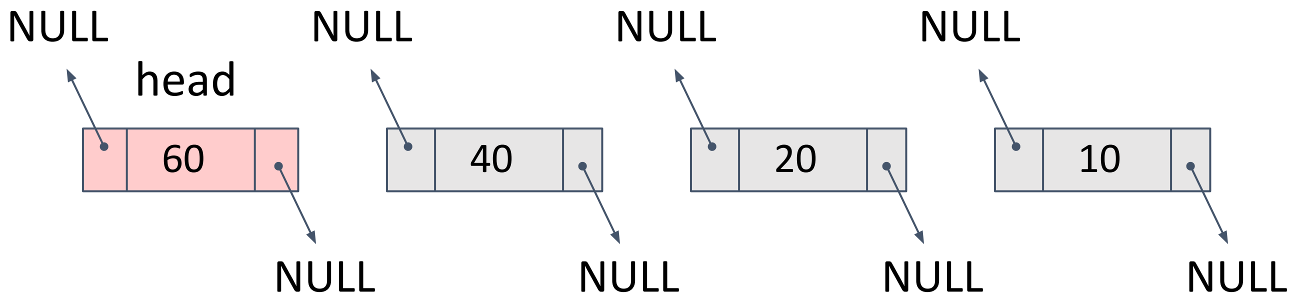
```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);
```



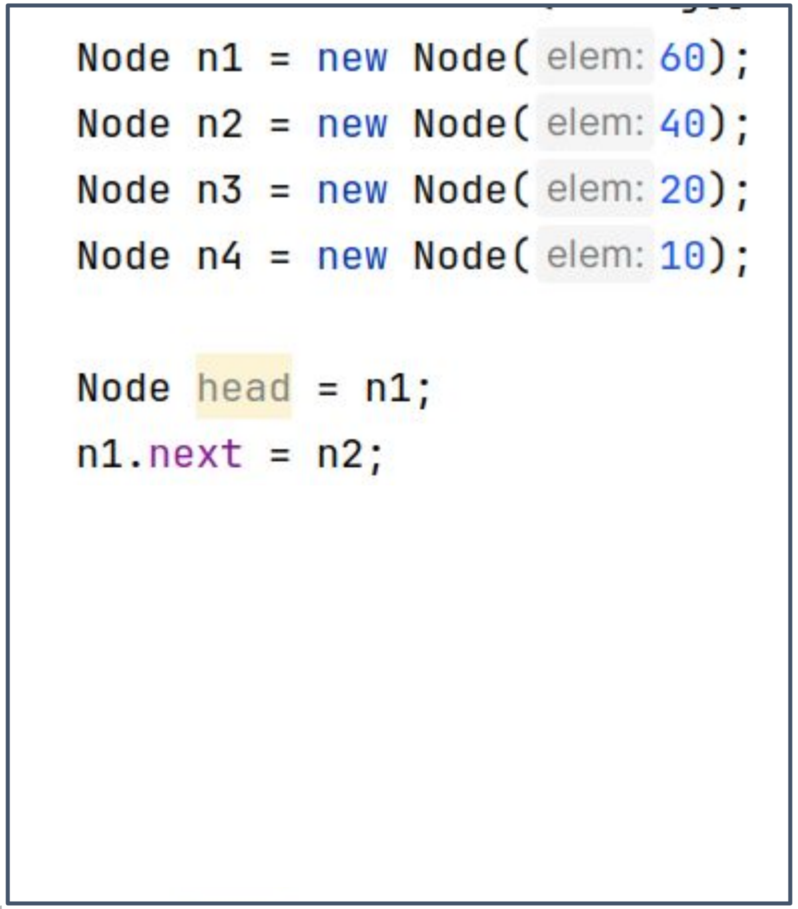
DLL Creation



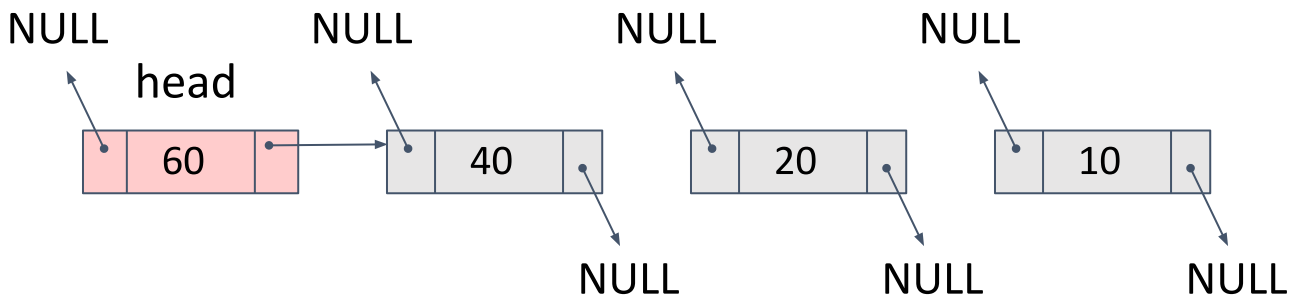
```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);

Node head = n1;
```



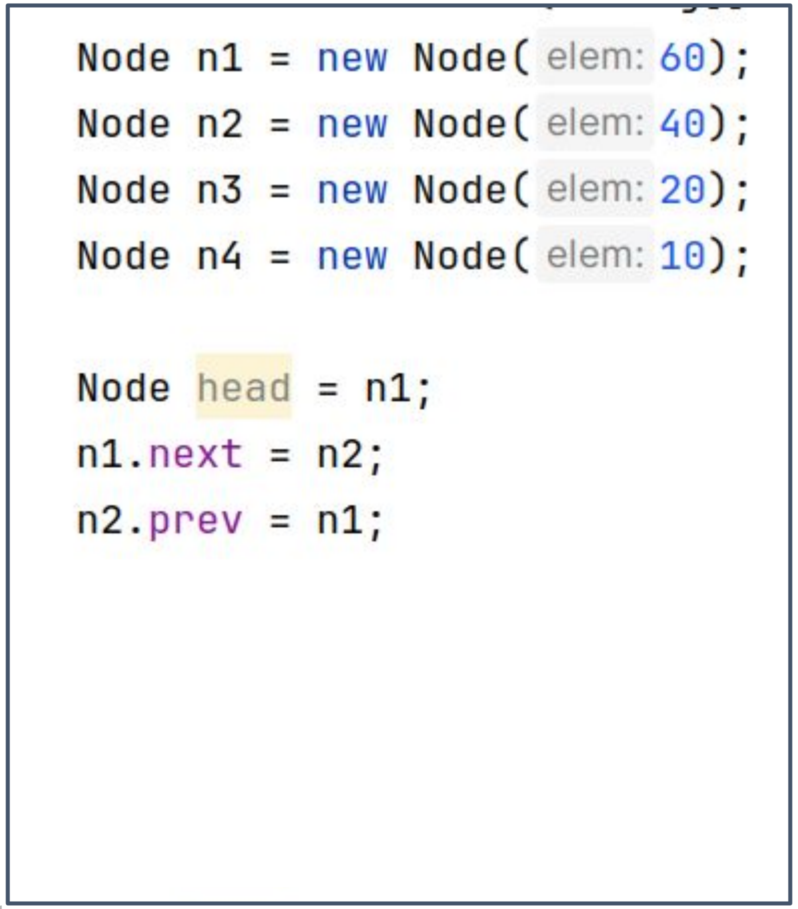
DLL Creation



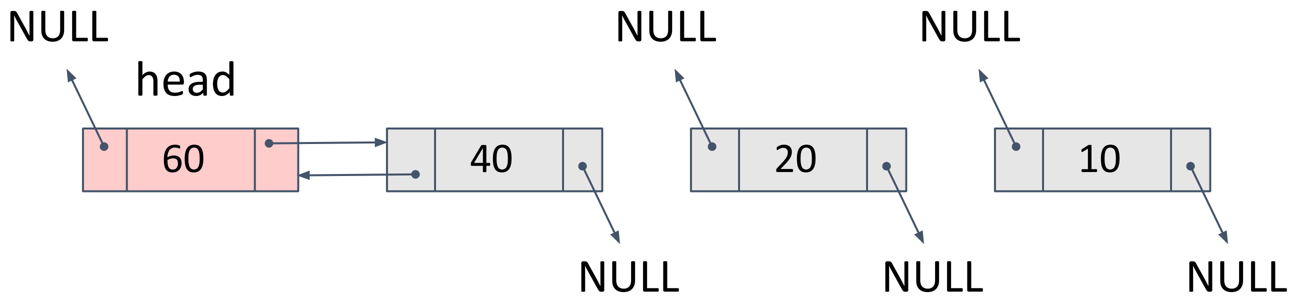
```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);

Node head = n1;
n1.next = n2;
```



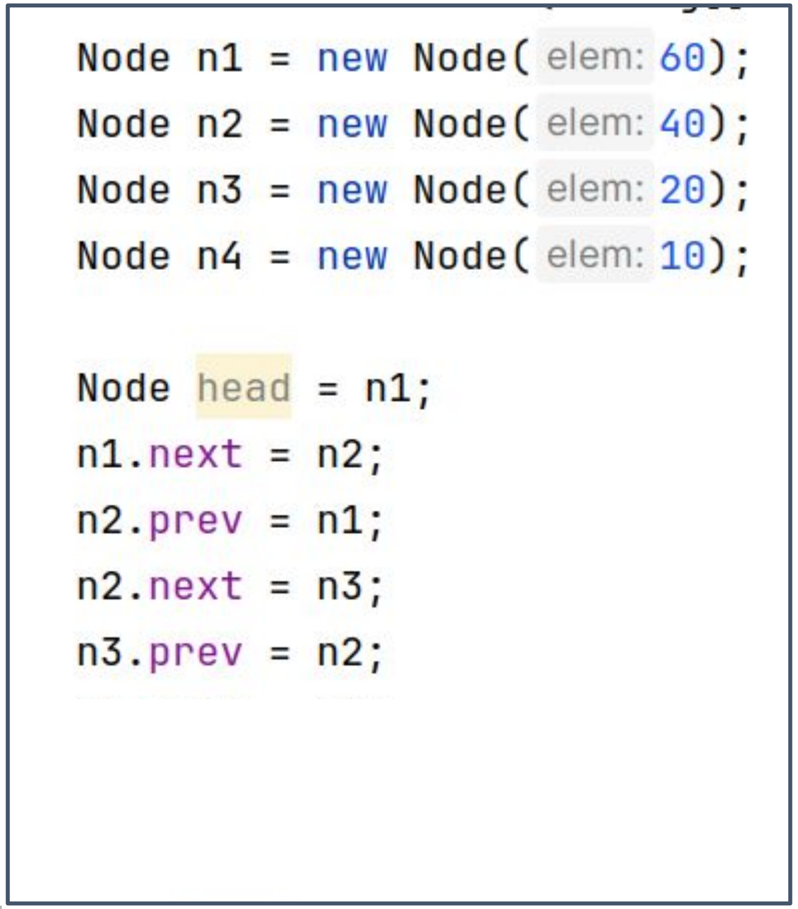
DLL Creation



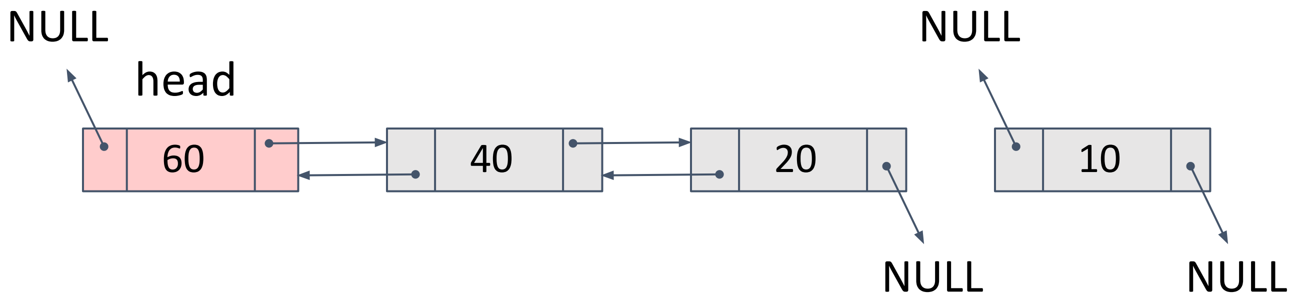
```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);

Node head = n1;
n1.next = n2;
n2.prev = n1;
```



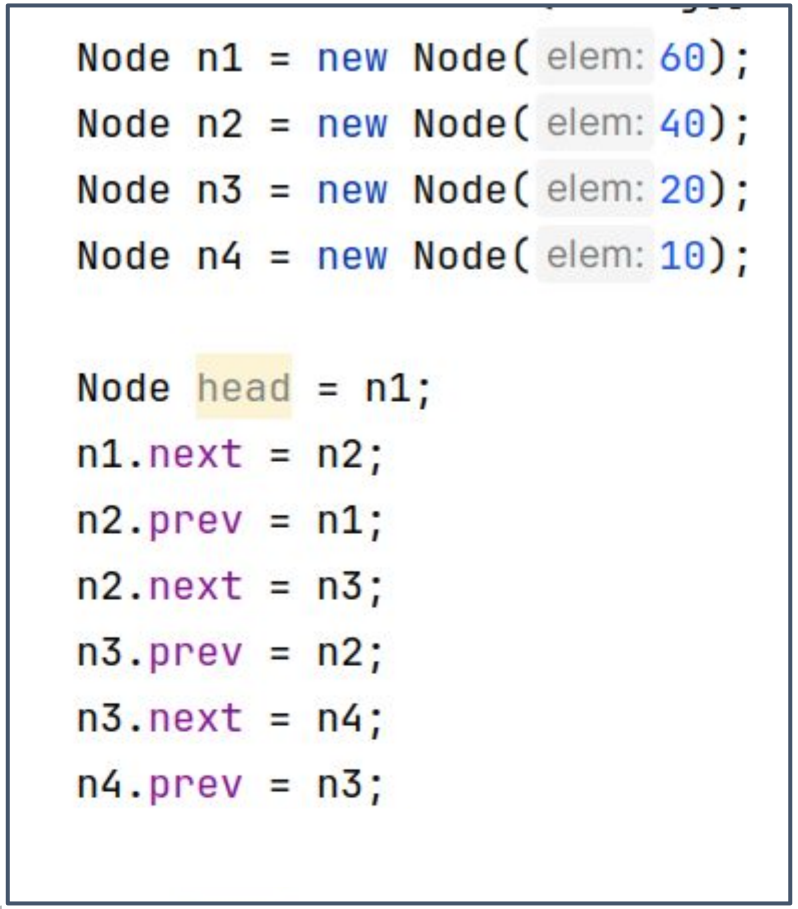
DLL Creation



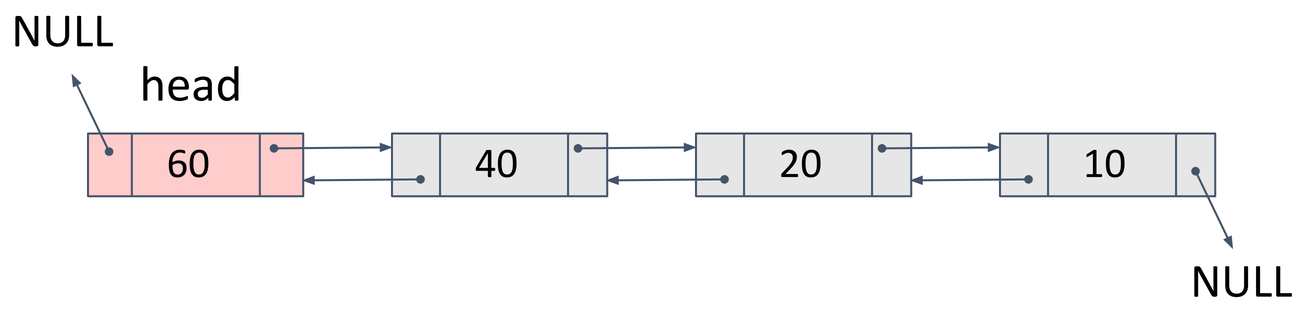
```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);

Node head = n1;
n1.next = n2;
n2.prev = n1;
n2.next = n3;
n3.prev = n2;
```



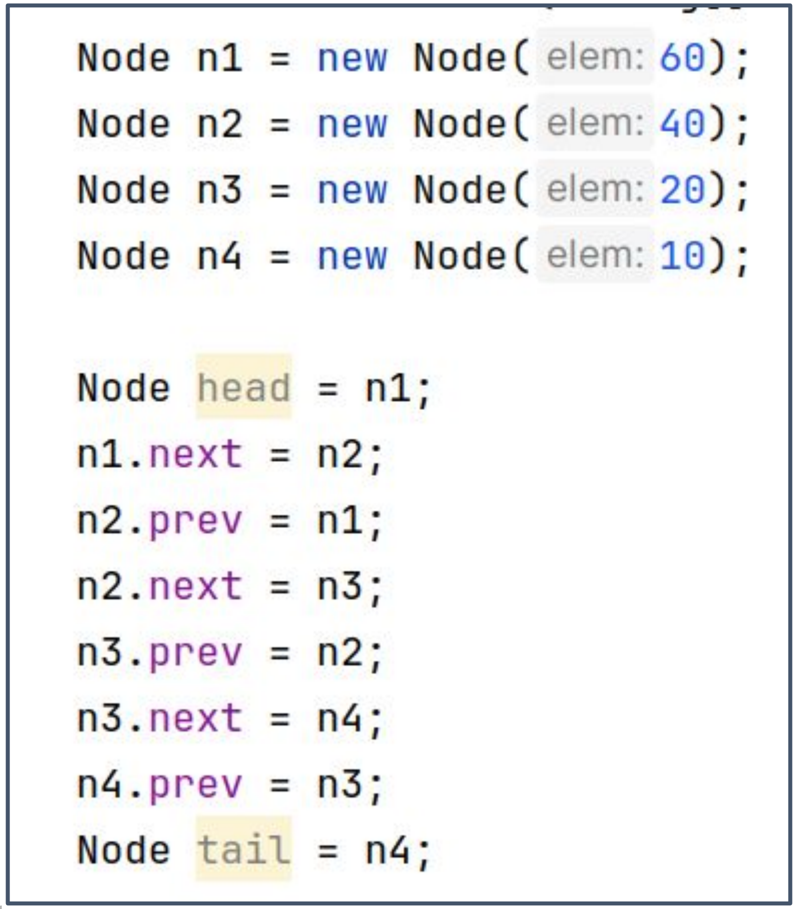
DLL Creation



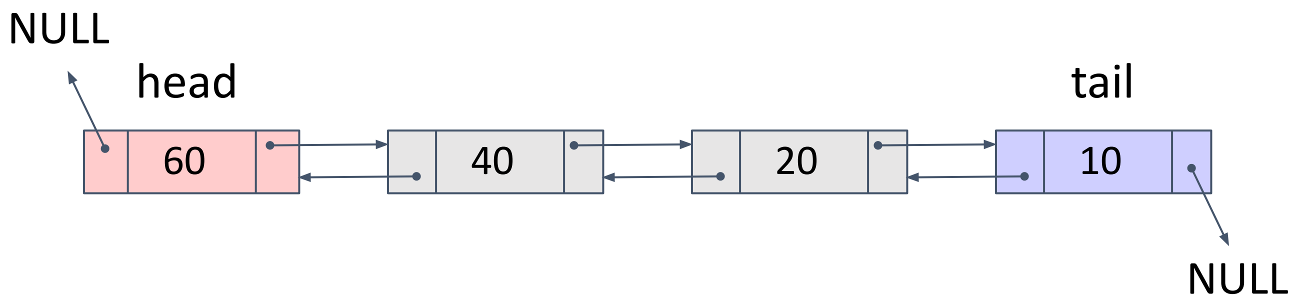
```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);

Node head = n1;
n1.next = n2;
n2.prev = n1;
n2.next = n3;
n3.prev = n2;
n3.next = n4;
n4.prev = n3;
```



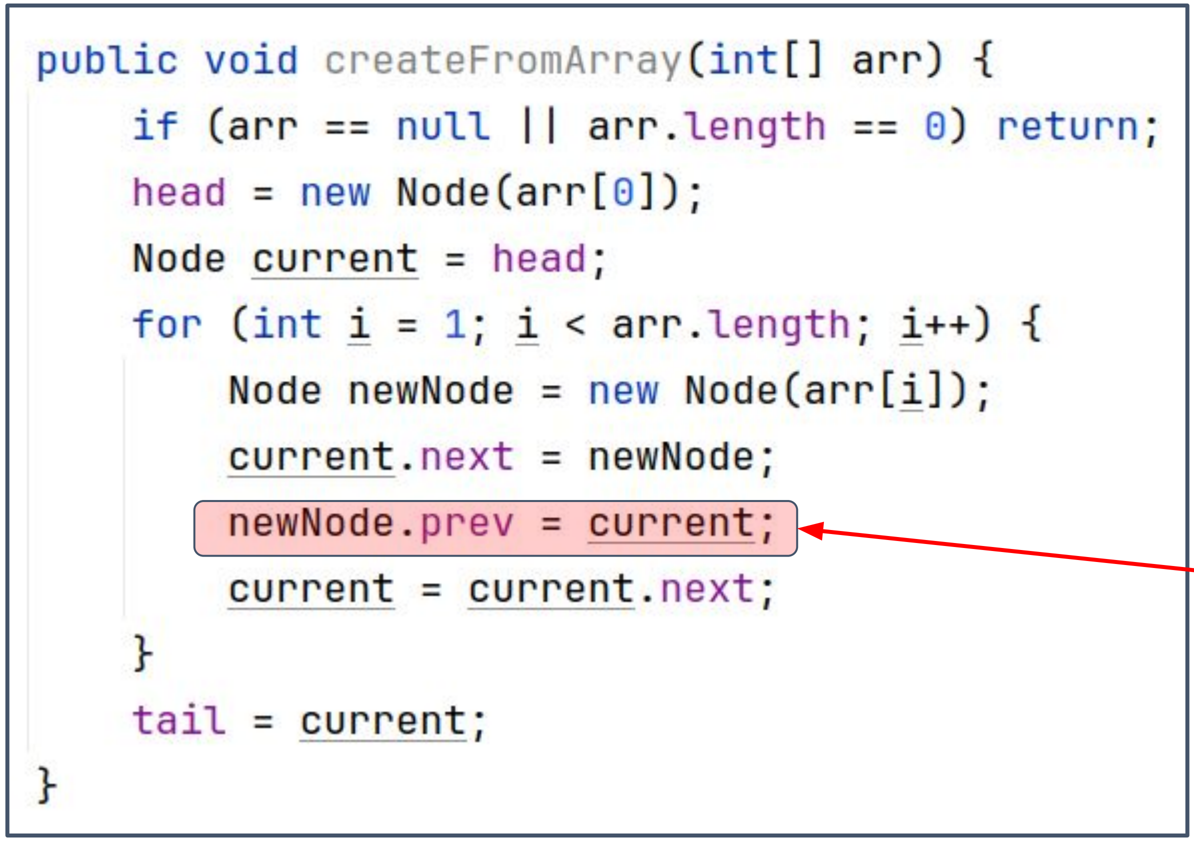
DLL Creation



```
Node n1 = new Node( elem: 60);
Node n2 = new Node( elem: 40);
Node n3 = new Node( elem: 20);
Node n4 = new Node( elem: 10);

Node head = n1;
n1.next = n2;
n2.prev = n1;
n2.next = n3;
n3.prev = n2;
n3.next = n4;
n4.prev = n3;
Node tail = n4;
```



DLL Creation from Array



```java
public void createFromArray(int[] arr) {
    if (arr == null || arr.length == 0) return;
    head = new Node(arr[0]);
    Node current = head;
    for (int i = 1; i < arr.length; i++) {
        Node newNode = new Node(arr[i]);
        current.next = newNode;
        newNode.prev = current;
        current = current.next;
    }
    tail = current;
}
```

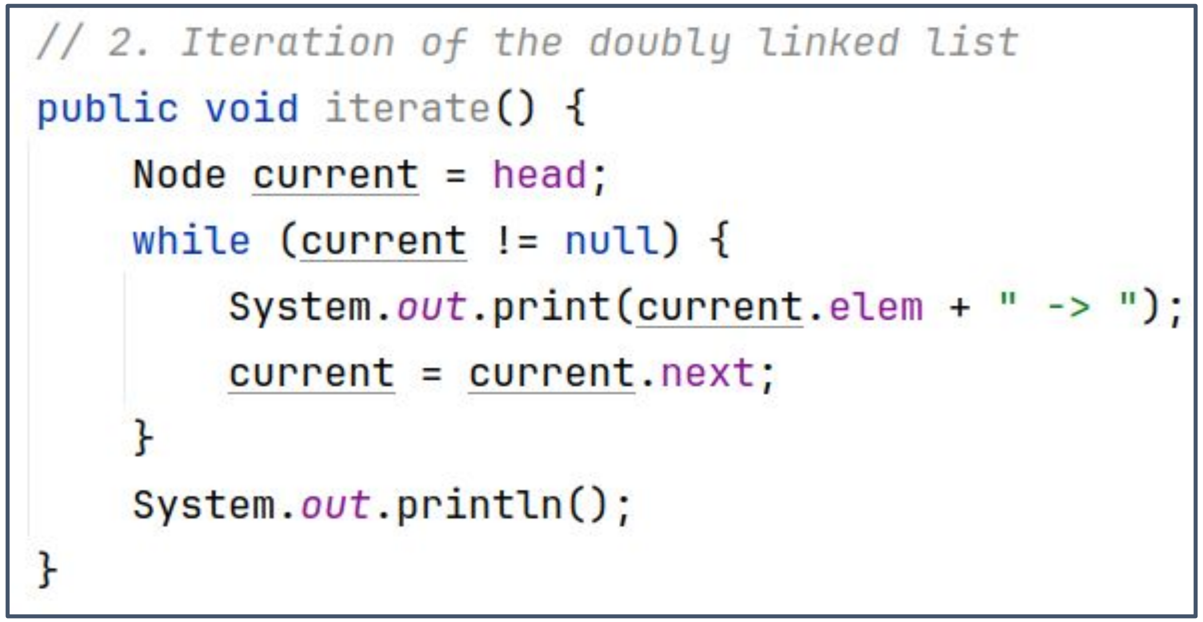
arr = [60, 40, 20, 10]

$O(n)$

Only this part is absent in SLL

Sequential Traversal of DLL

Same as SLL

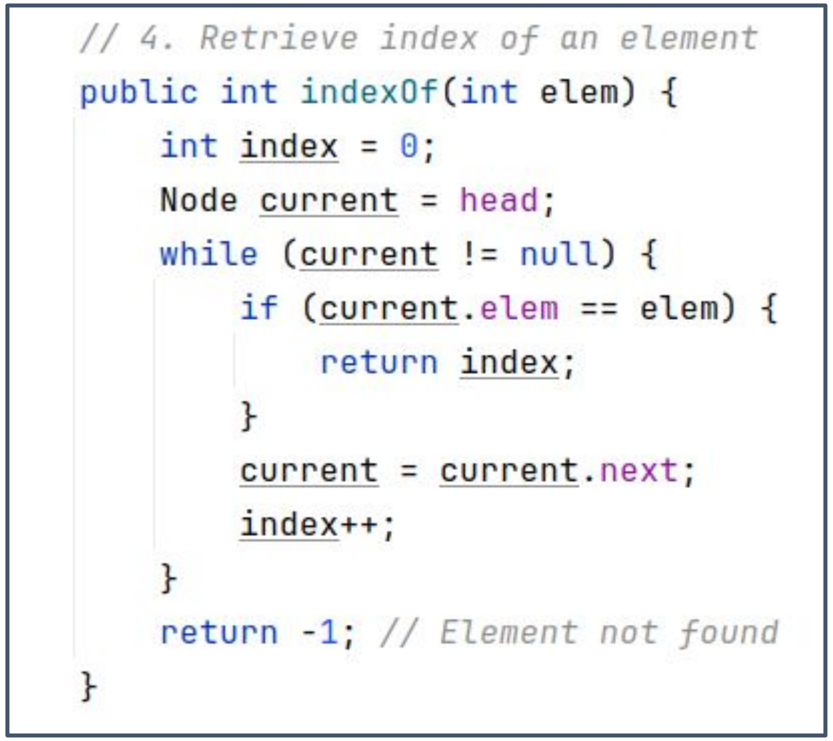


```
// 2. Iteration of the doubly linked list
public void iterate() {
    Node current = head;
    while (current != null) {
        System.out.print(current.elem + " -> ");
        current = current.next;
    }
    System.out.println();
}
```

$O(n)$

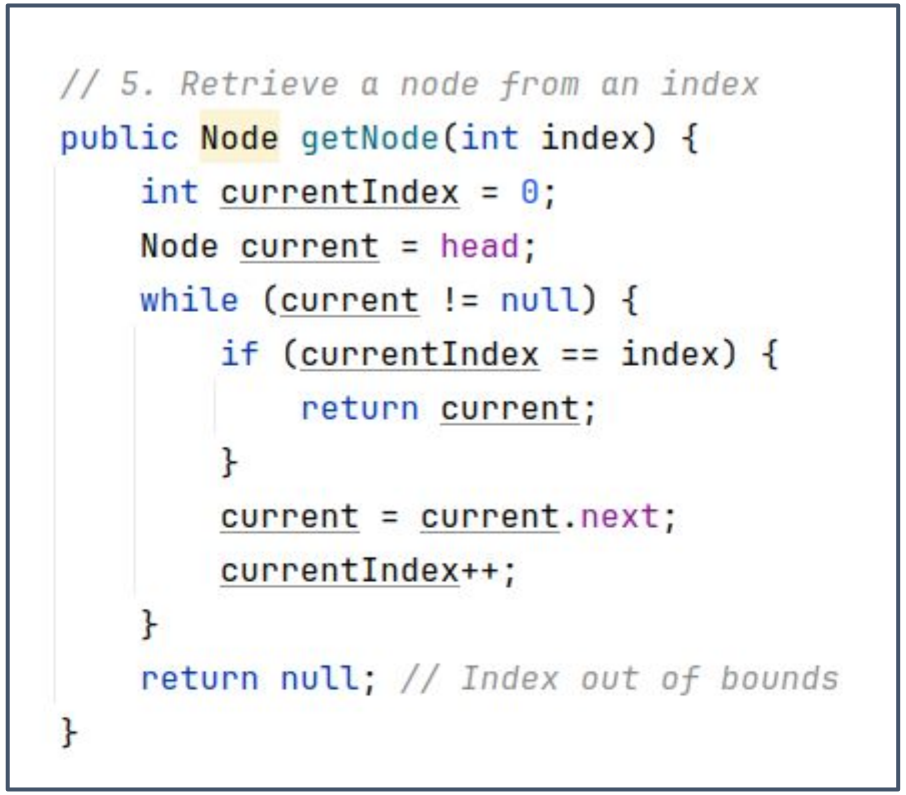
Element Access

Same as SLL



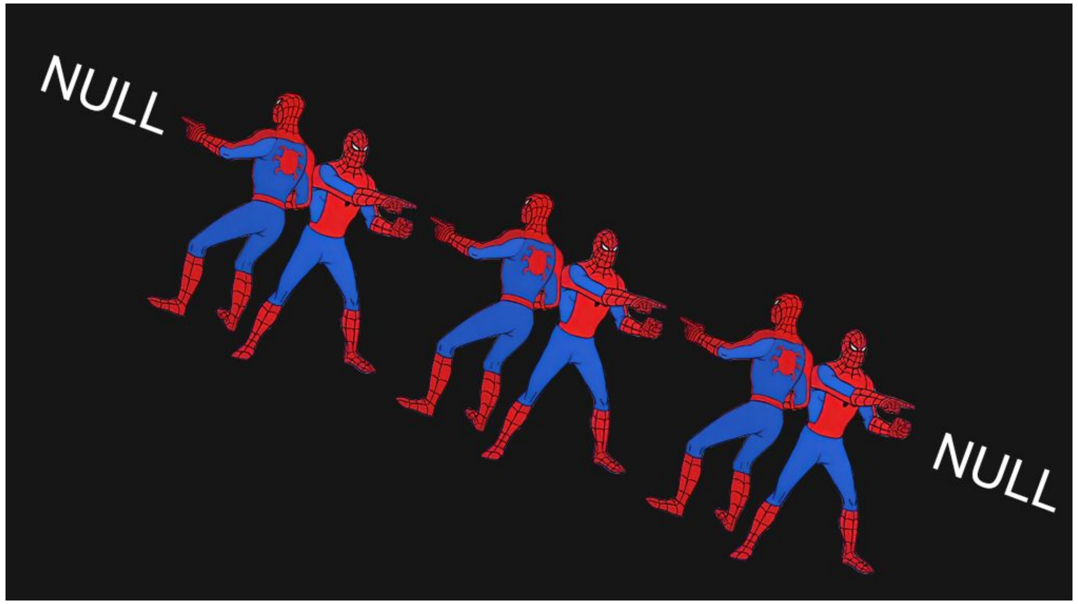
```
// 4. Retrieve index of an element
public int indexOf(int elem) {
    int index = 0;
    Node current = head;
    while (current != null) {
        if (current.elem == elem) {
            return index;
        }
        current = current.next;
        index++;
    }
    return -1; // Element not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

$O(n)$



```
// 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
}
```

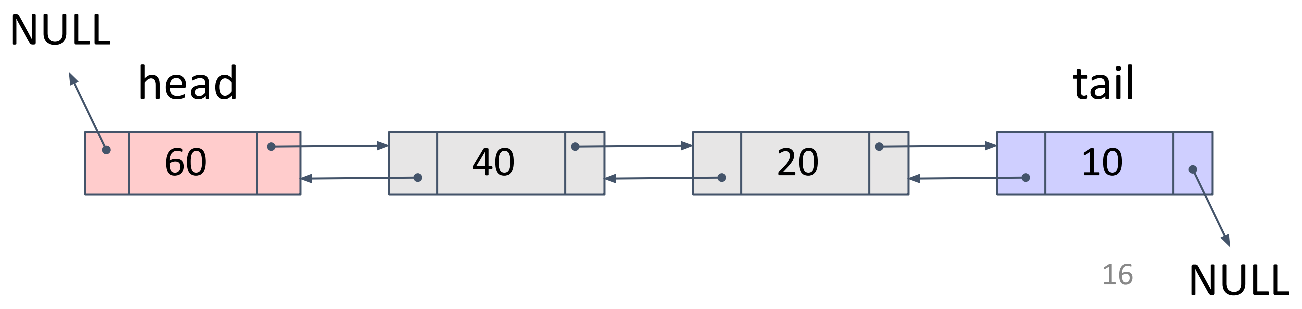
$O(n)$


NULL
NULL

Element Insertion (At the beginning)

Prepend 80 to the list

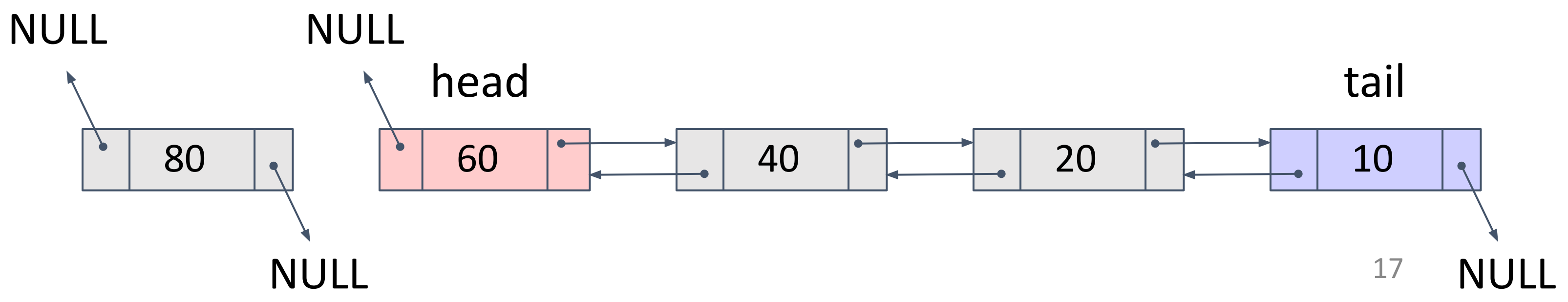
```
    // Insert at the beginning
```



Element Insertion (At the beginning)

Prepend 80 to the list

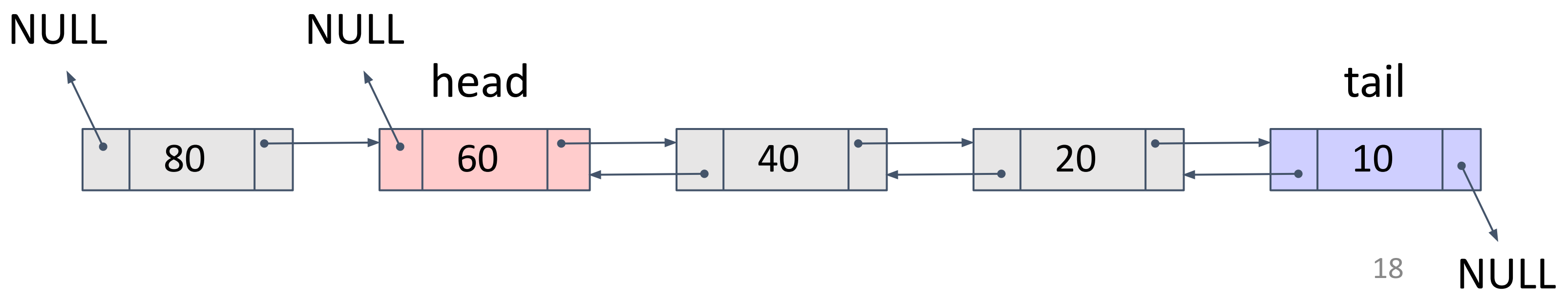
```
    // Insert at the beginning
    Node newNode = new Node( elem: 80);
```



Element Insertion (At the beginning)

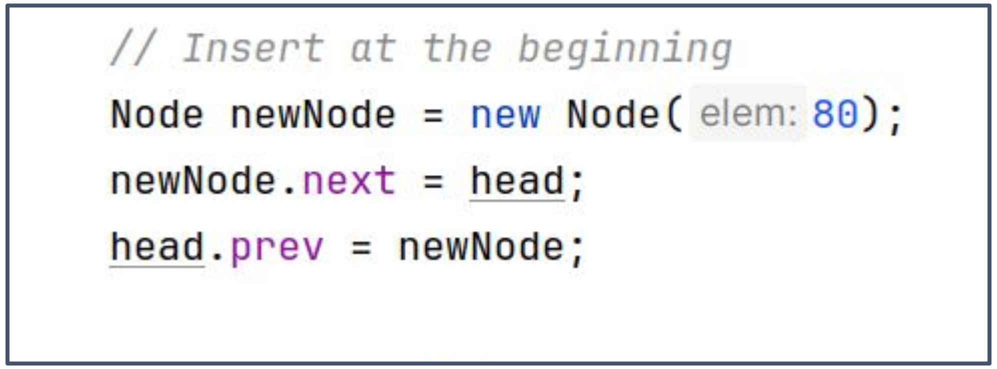
Prepend 80 to the list

```
// Insert at the beginning
Node newNode = new Node( elem: 80);
newNode.next = head;
```

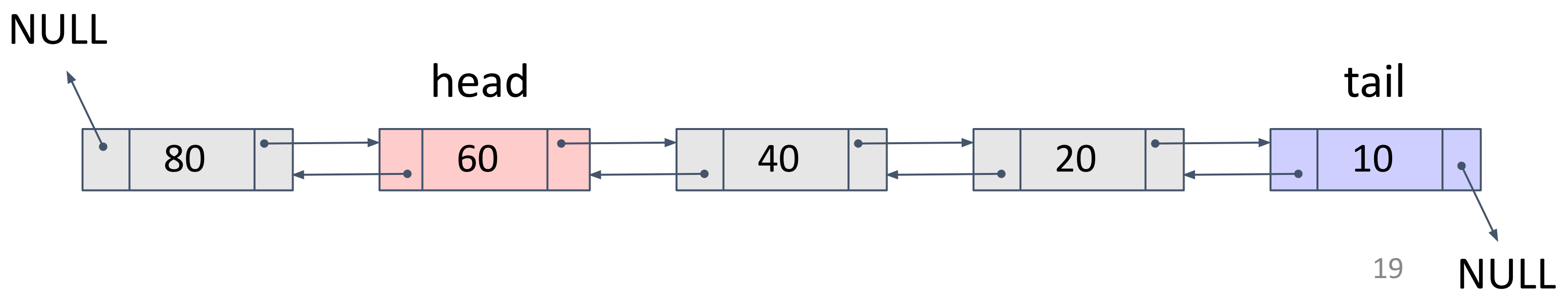


Element Insertion (At the beginning)

Prepend 80 to the list

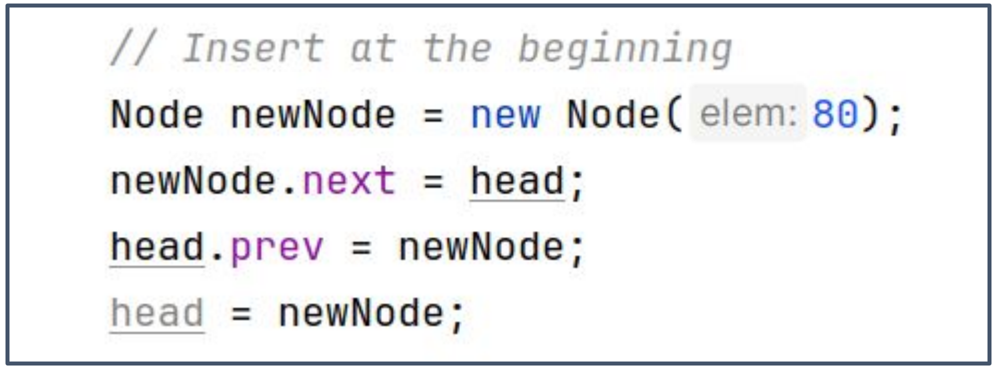


```
// Insert at the beginning
Node newNode = new Node( elem: 80);
newNode.next = head;
head.prev = newNode;
```



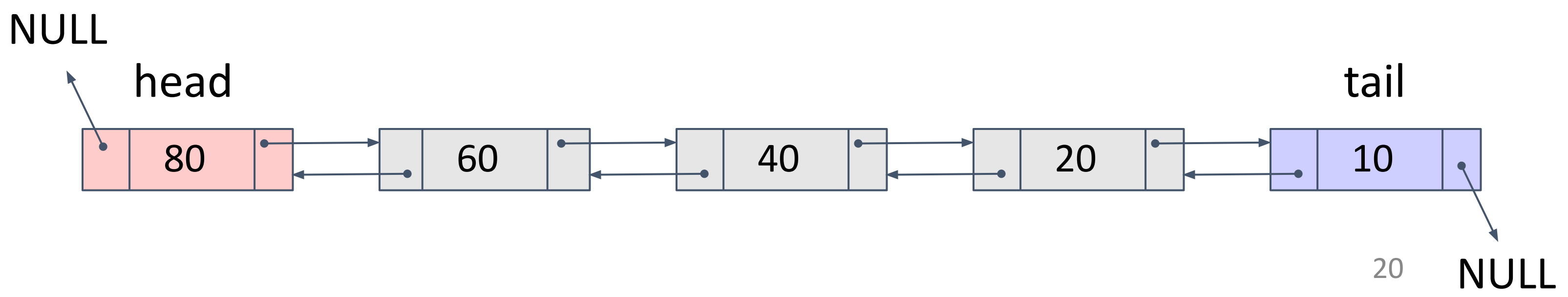
Element Insertion (At the beginning)

Prepend 80 to the list



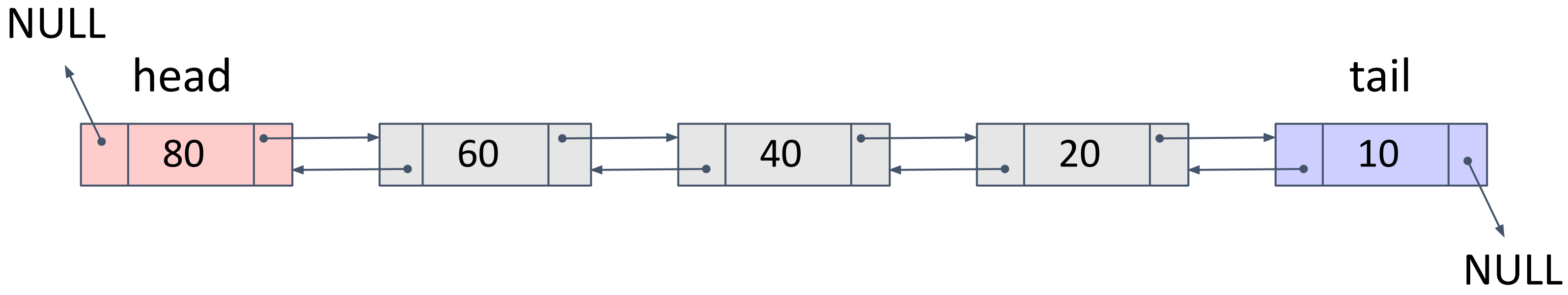
```
// Insert at the beginning
Node newNode = new Node( elem: 80);
newNode.next = head;
head.prev = newNode;
head = newNode;
```

$O(1)$



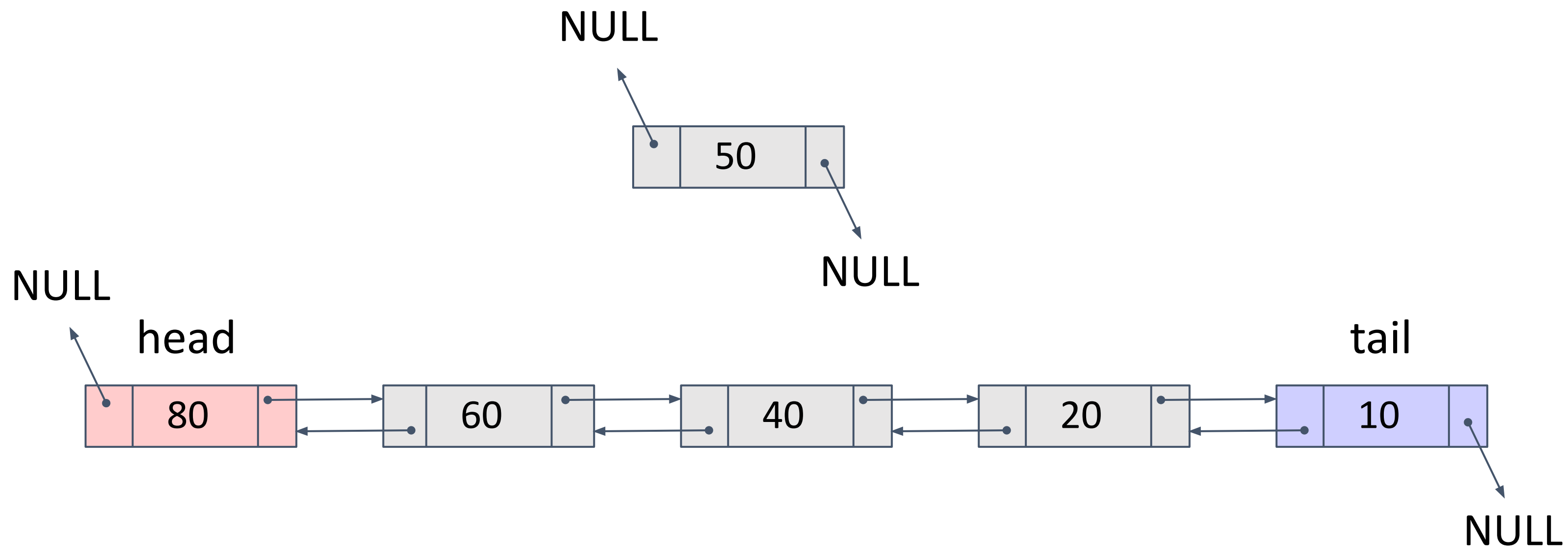
Element Insertion (Middle)

Insert 50 at index 2



Element Insertion (Middle)

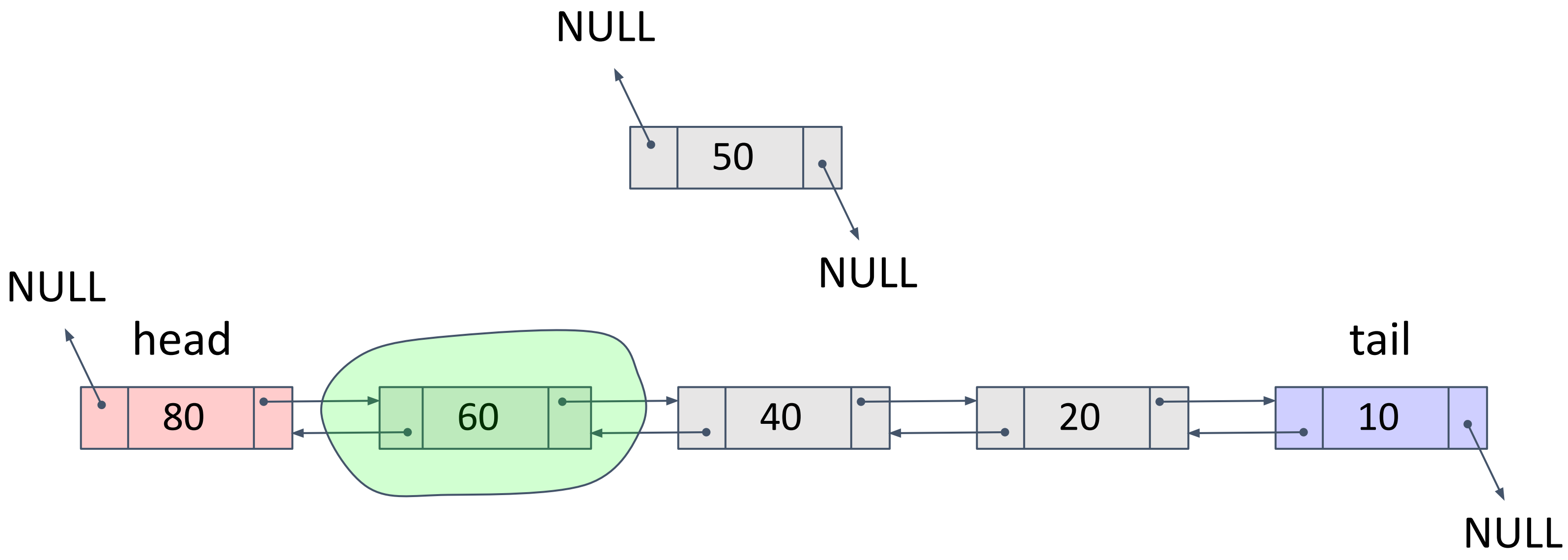
Insert 50 at index 2



```
Node newNode = new Node(elem);
```

Element Insertion (Middle)

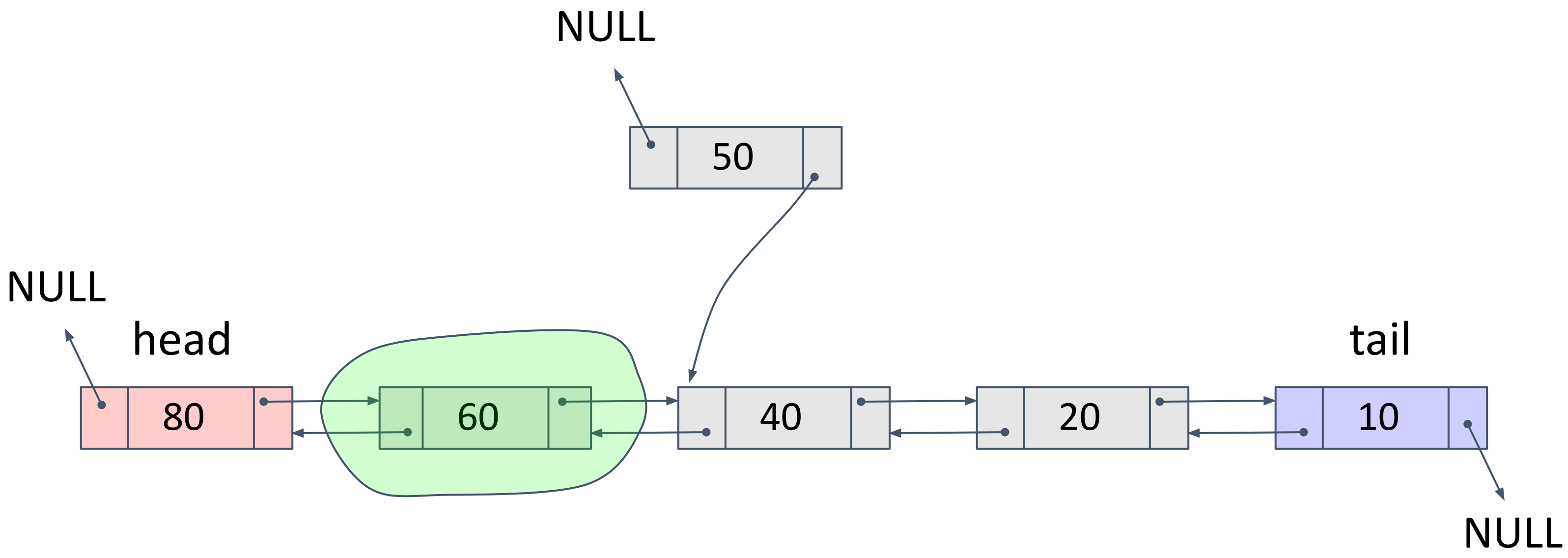
Insert 50 at index 2



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

Element Insertion (Middle)

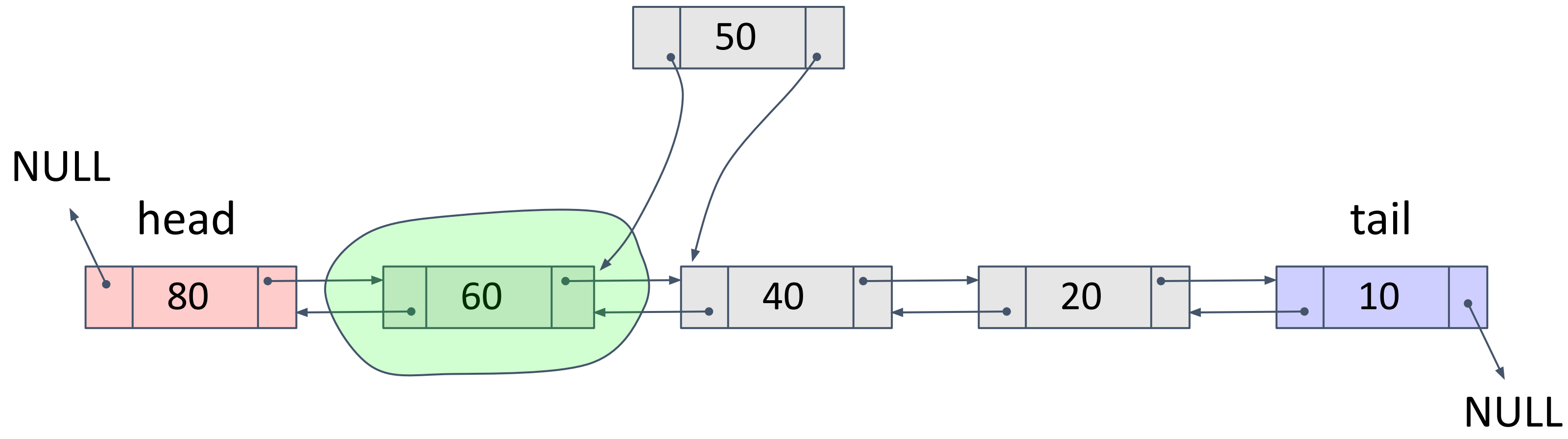
Insert 50 at index 2



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

Element Insertion (Middle)

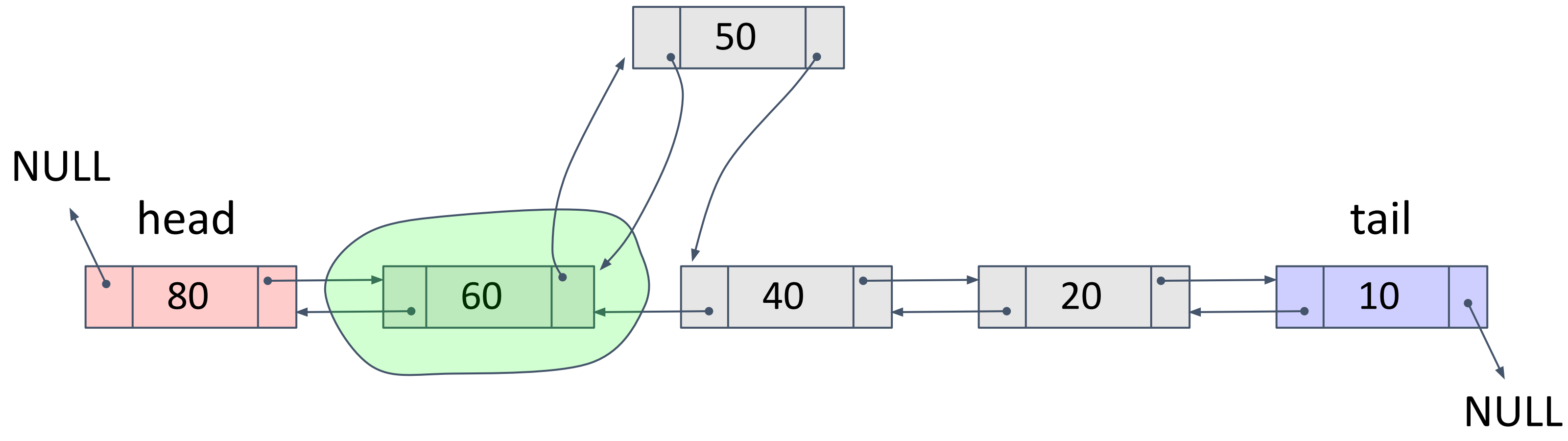
Insert 50 at index 2



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

Element Insertion (Middle)

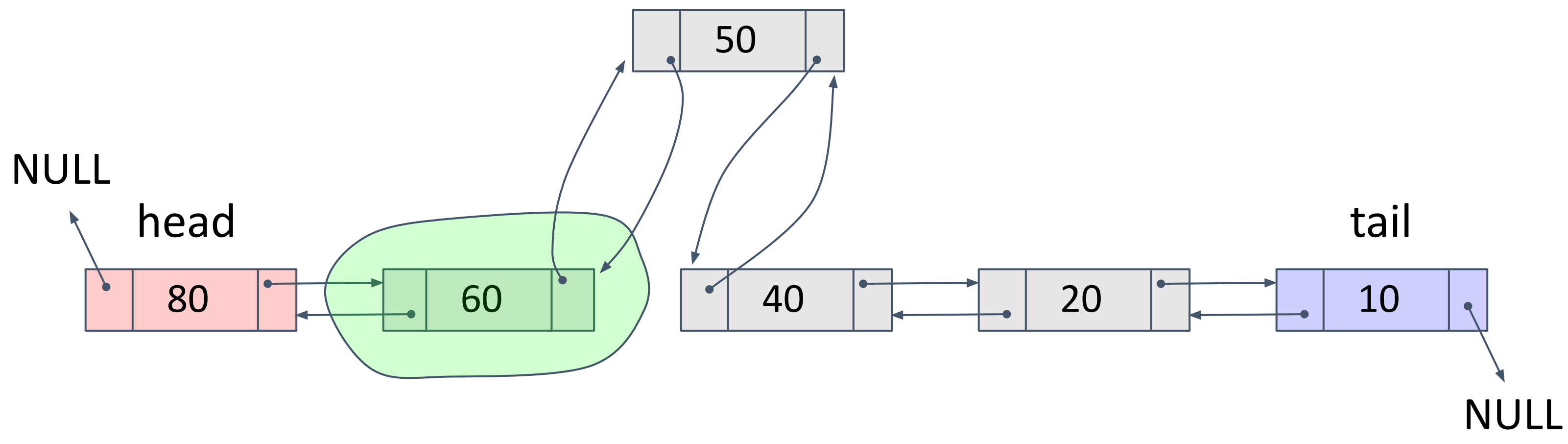
Insert 50 at index 2



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

Element Insertion (Middle)

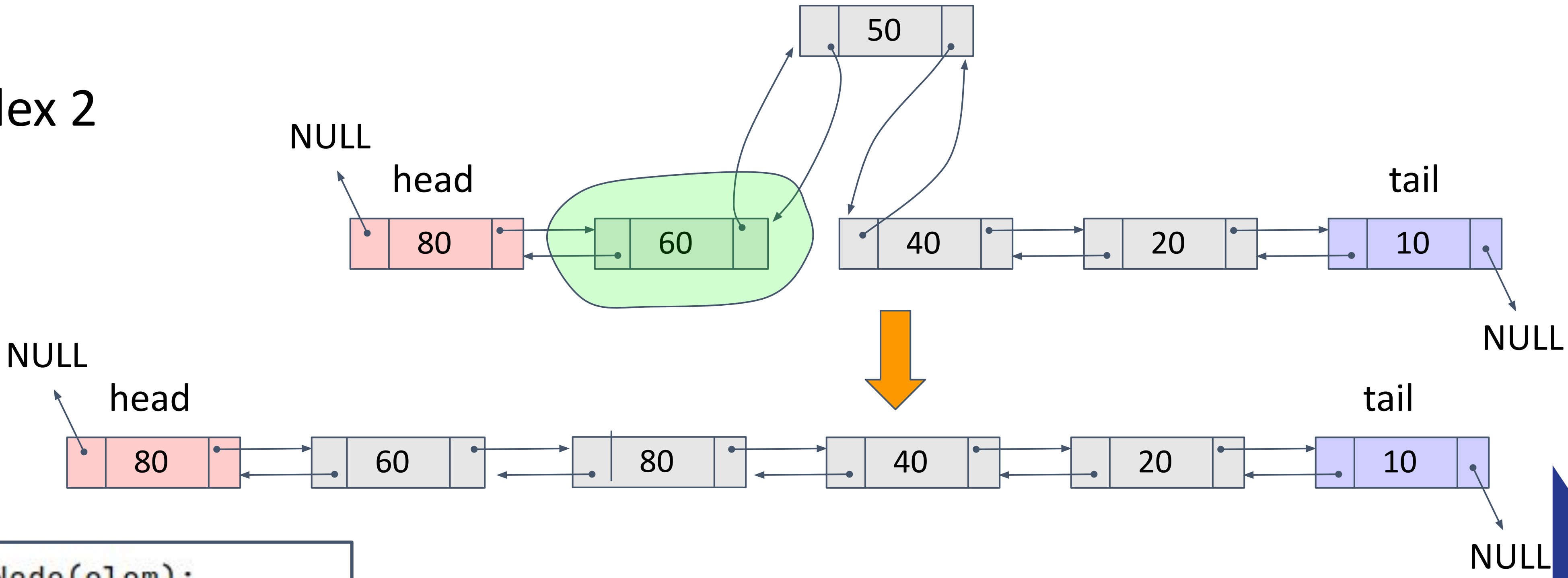
Insert 50 at index 2



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

Element Insertion (Middle)

Insert 50 at index 2



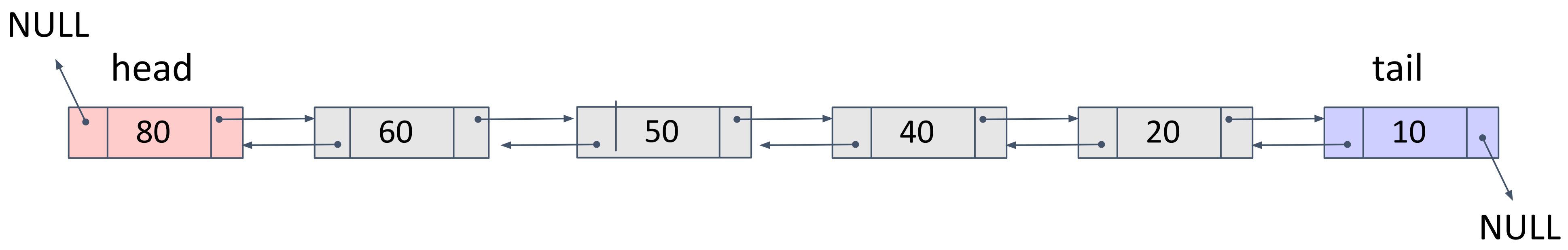
```java
Node newNode = new Node(elem);
Node prev = getNode( index: index - 1);
newNode.next = prev.next;
newNode.prev = prev;
prev.next = newNode;
newNode.next.prev = newNode;
```

$O(n)$

General Rule

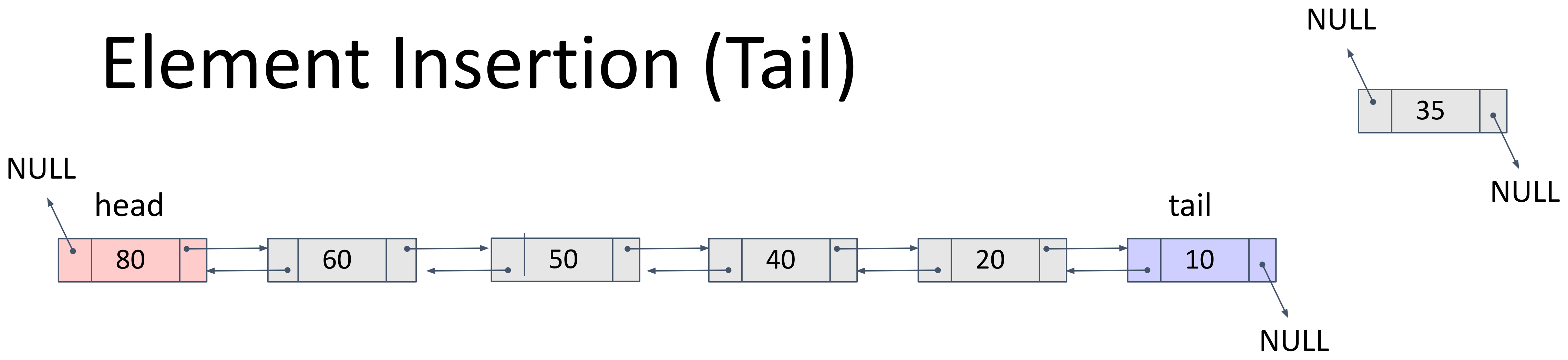
- Better to connect the new node with the linked list first
- Less risk of breaking the list or losing references to other nodes
- Think: what would happen if `prev.next = newNode` was executed first?

Element Insertion (Tail)



Insert 35 at the end of the list

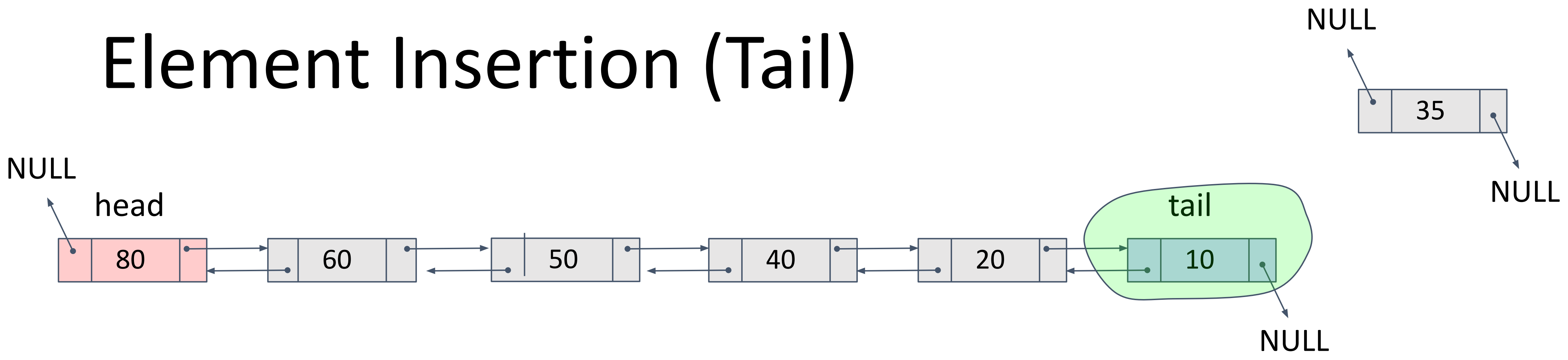
Element Insertion (Tail)



Insert 35 at the end of the list

```
Node newNode = new Node(elem);
```

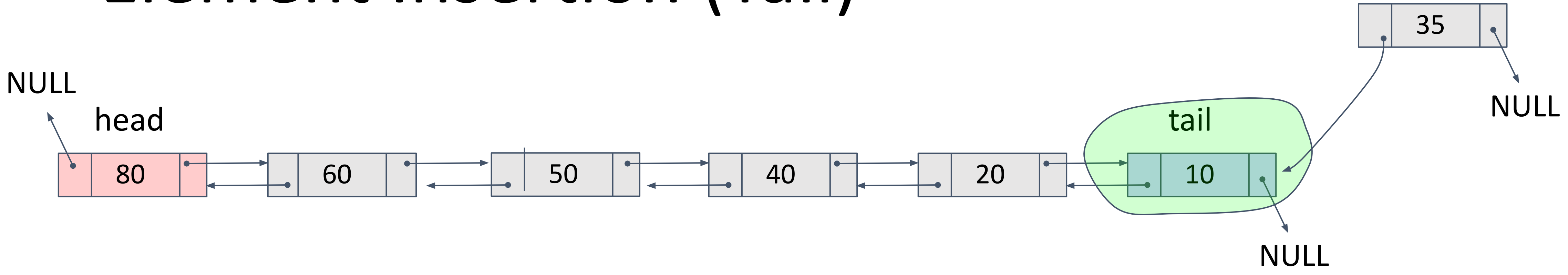
Element Insertion (Tail)



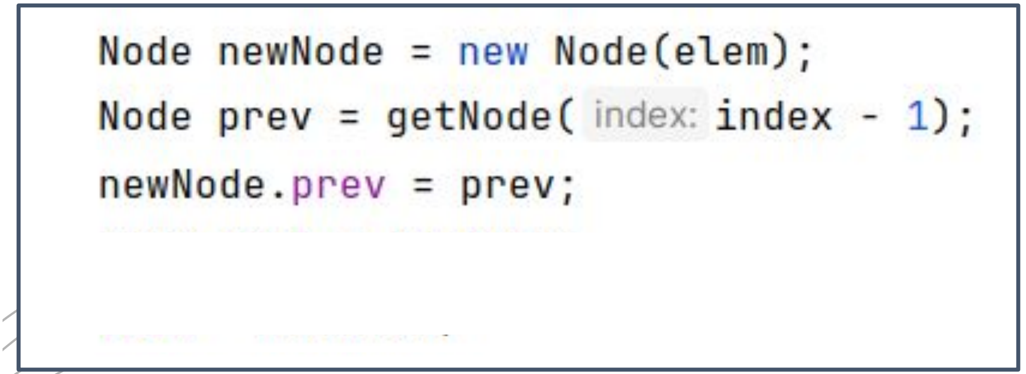
Insert 35 at the end of the list

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

Element Insertion (Tail)

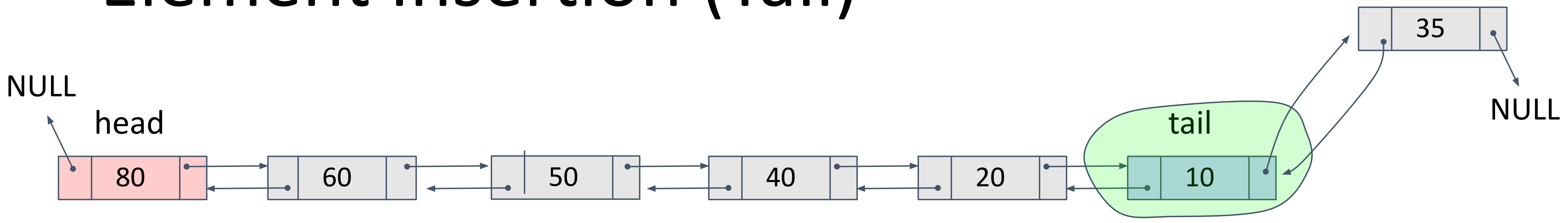


Insert 35 at the end of the list

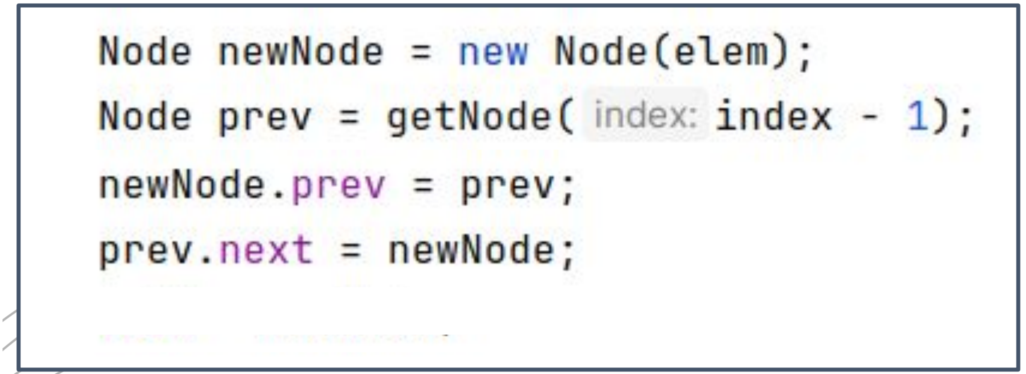


```
Node newNode = new Node(elem);
Node prev = getNode( index: index - 1);
newNode.prev = prev;
```

Element Insertion (Tail)

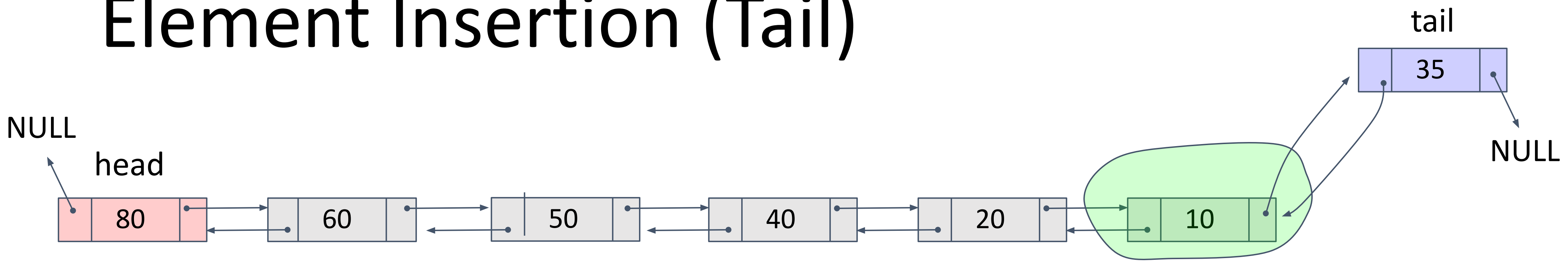


Insert 35 at the end of the list

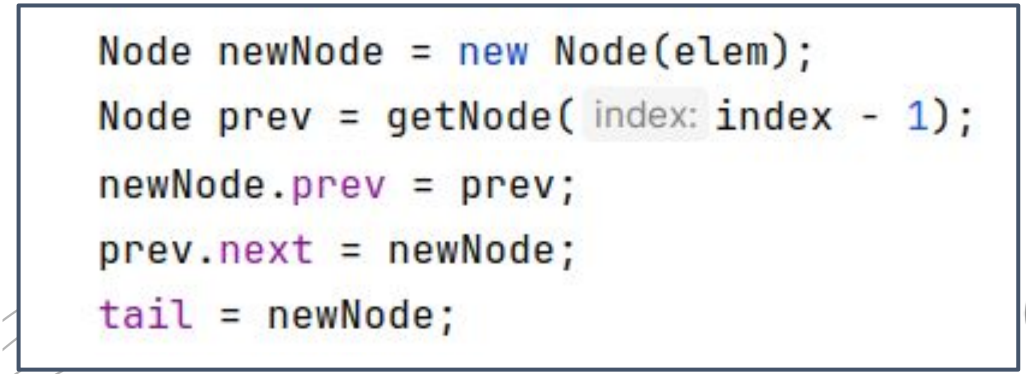


```
Node newNode = new Node(elem);
Node prev = getNode( index: index - 1);
newNode.prev = prev;
prev.next = newNode;
```

Element Insertion (Tail)



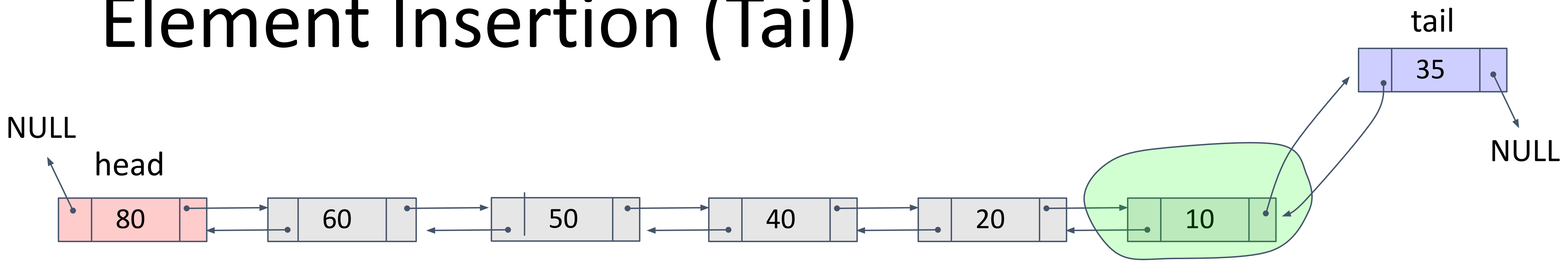
Insert 35 at the end of the list



```
Node newNode = new Node(elem);
Node prev = getNode( index: index - 1);
newNode.prev = prev;
prev.next = newNode;
tail = newNode;
```

$O(n)$

Element Insertion (Tail)

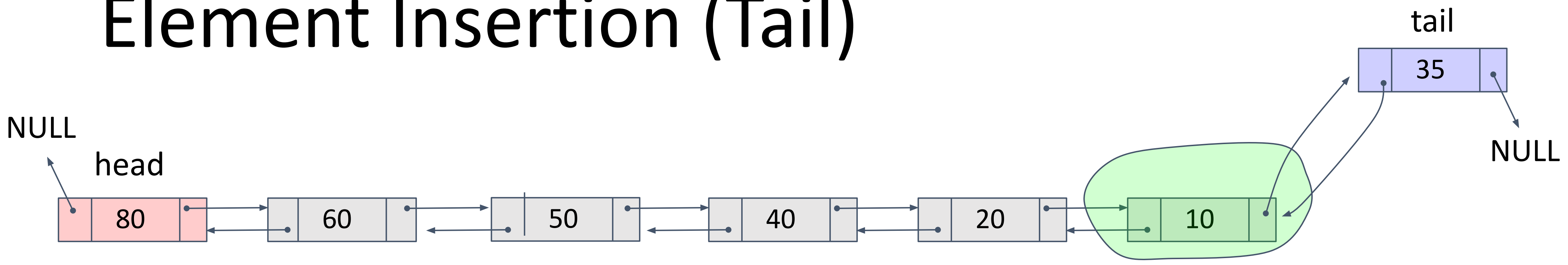


Can we optimize tail insertions?

```
Node newNode = new Node(elem);
Node prev = getNode( index: index - 1);
newNode.prev = prev;
prev.next = newNode;
tail = newNode;
```

$O(n)$

Element Insertion (Tail)

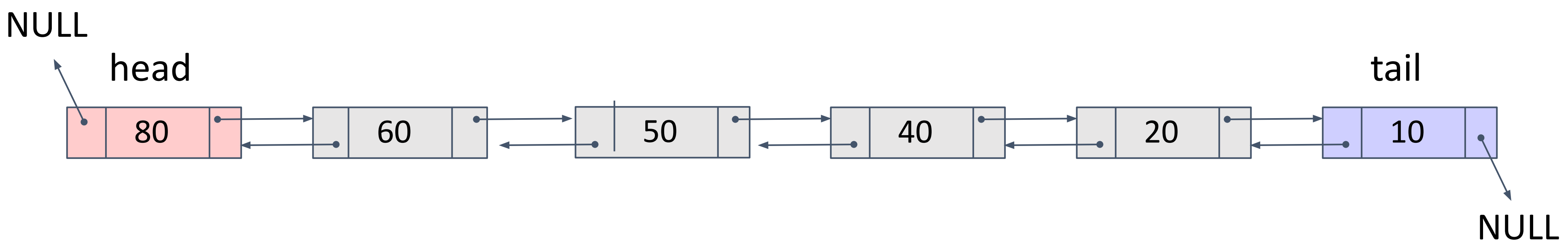


Can we optimize tail insertions?

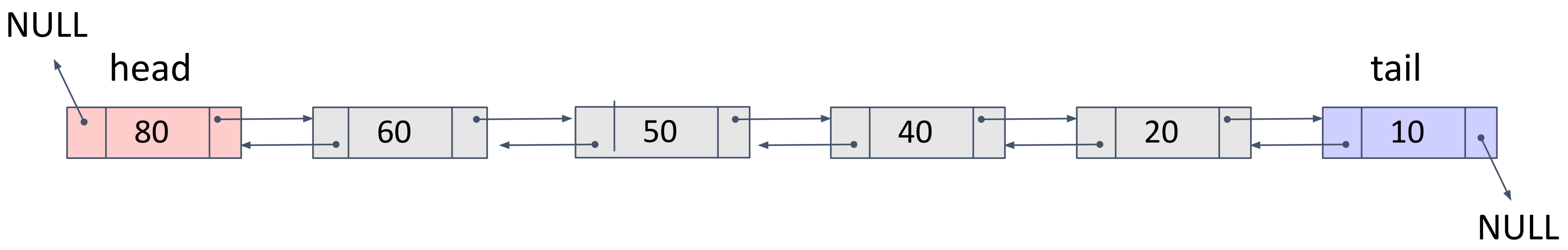
```
Node newNode = new Node(elem);
Node prev = tail;
newNode.prev = prev;
prev.next = newNode;
tail = newNode;
```

$O(1)$

Node Removal (Head)

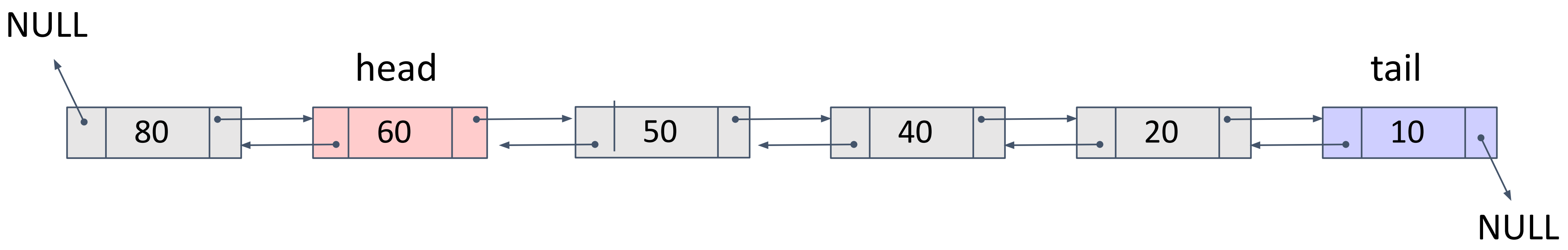


Node Removal (Head)



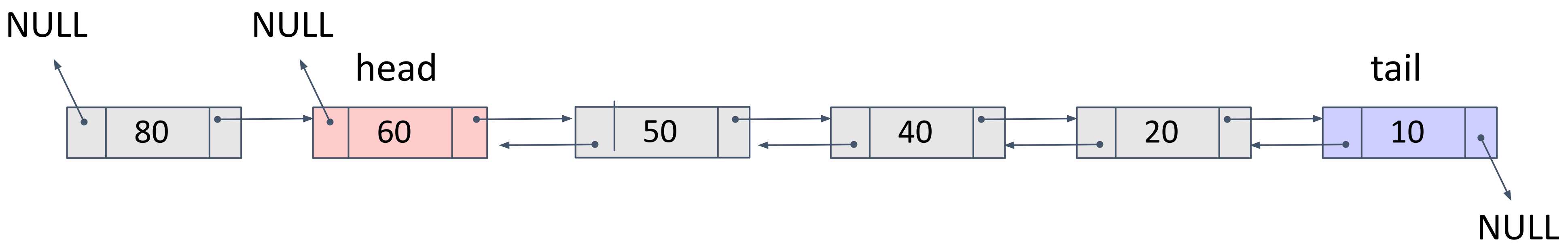
```
if (head == null) return;
```

Node Removal (Head)



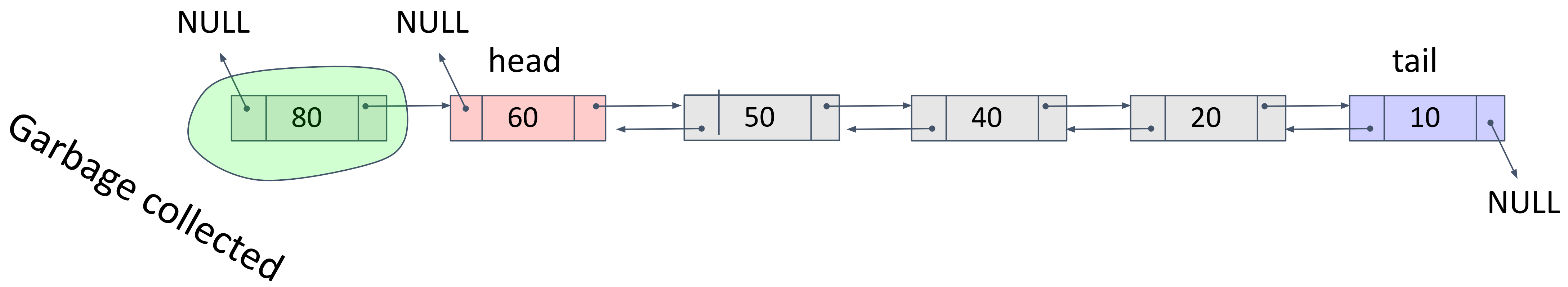
```
if (head == null) return;
head = head.next;
```

Node Removal (Head)



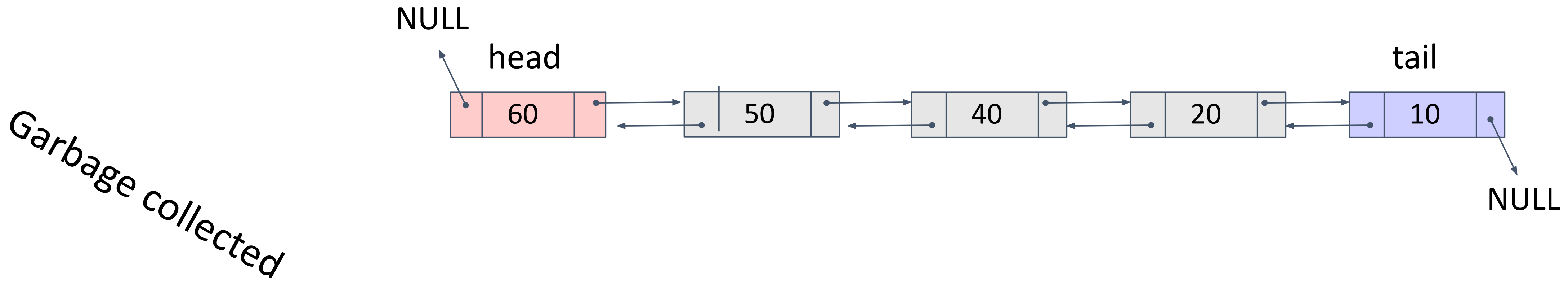
```
if (head == null) return;
head = head.next;




head.prev = null;
```

Node Removal (Head)



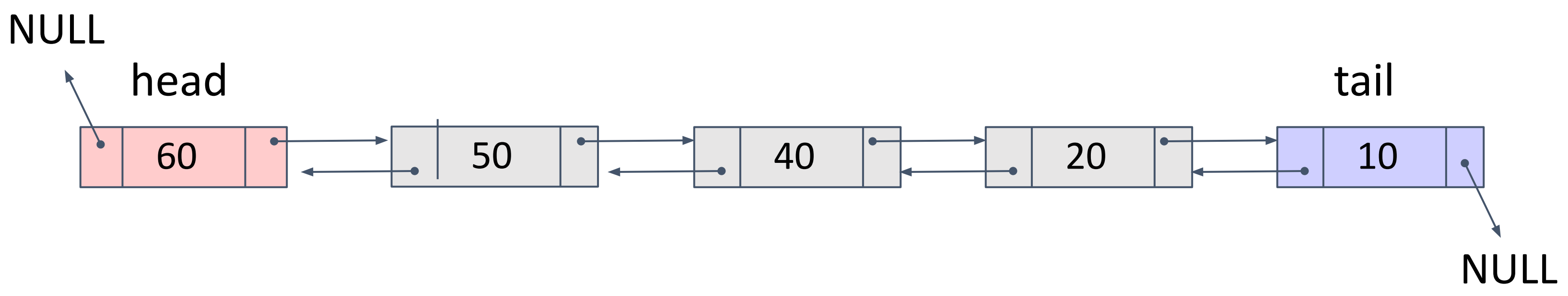
```
if (head == null) return;
head = head.next;




head.prev = null;
```

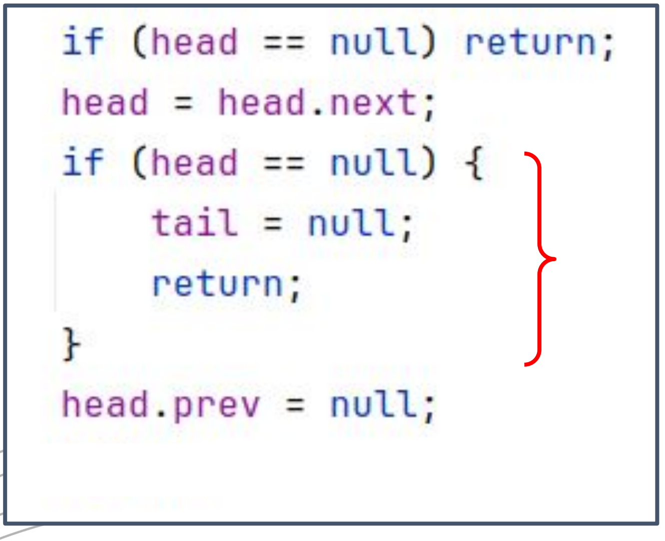
Node Removal (Head)



```
if (head == null) return;
head = head.next;




head.prev = null;
```

Node Removal (Head)



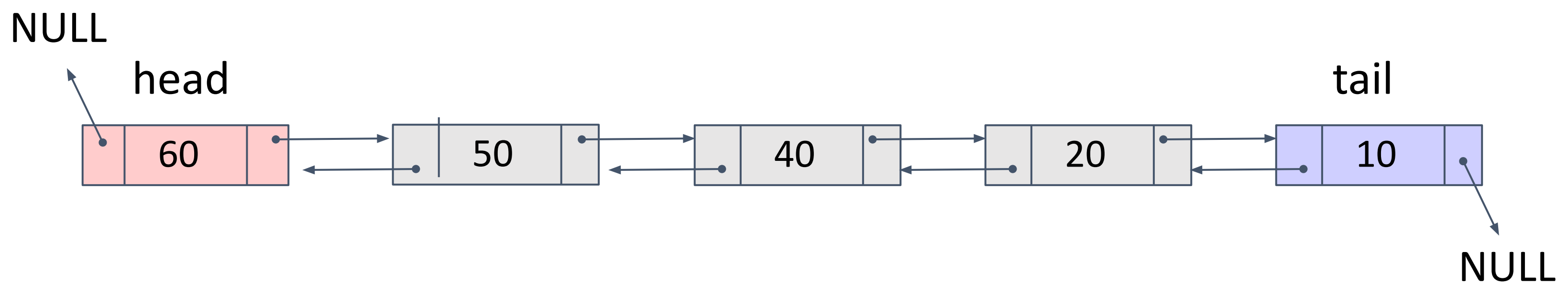


```
if (head == null) return;
head = head.next;
if (head == null) {
    tail = null;
    return;
}
head.prev = null;
```

Special case when list becomes empty

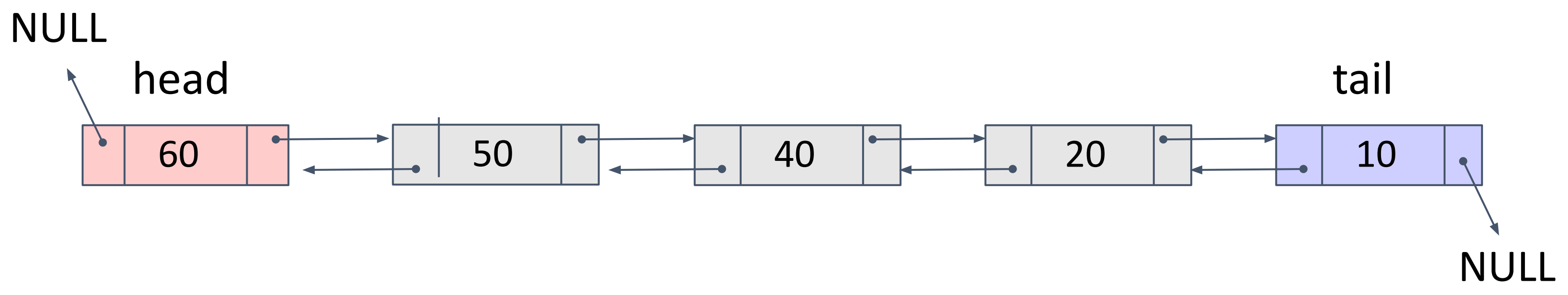
$O(1)$

Node Removal (Tail)



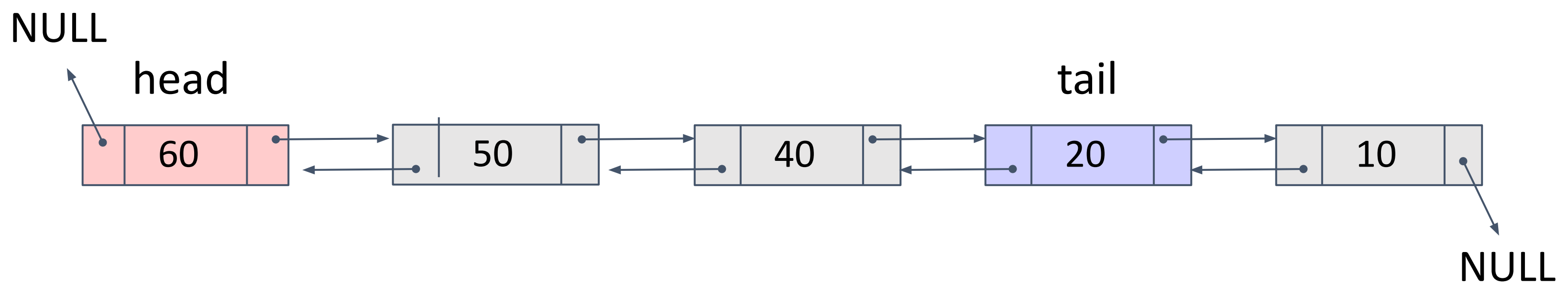
```
Node node = getNode(index);
```

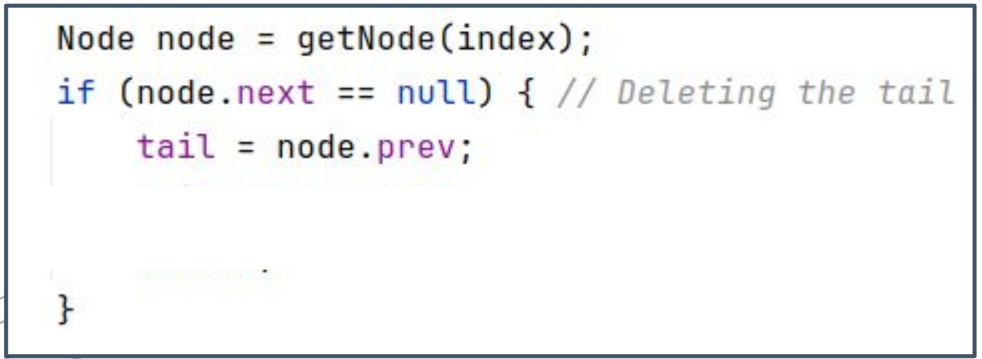
Node Removal (Tail)



```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail



}
```

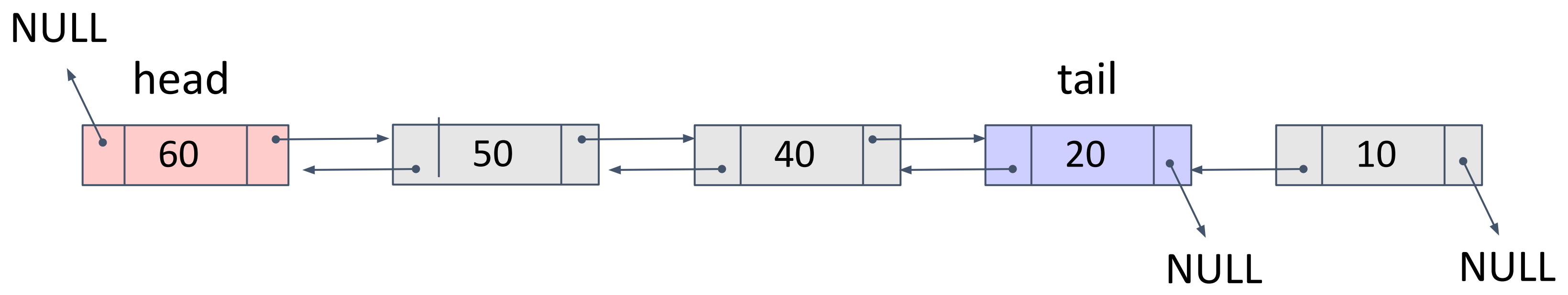
Node Removal (Tail)

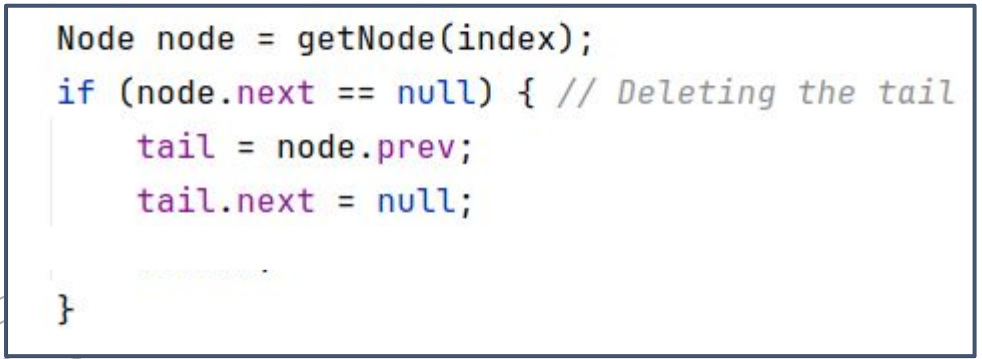




```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail
    tail = node.prev;


}
```

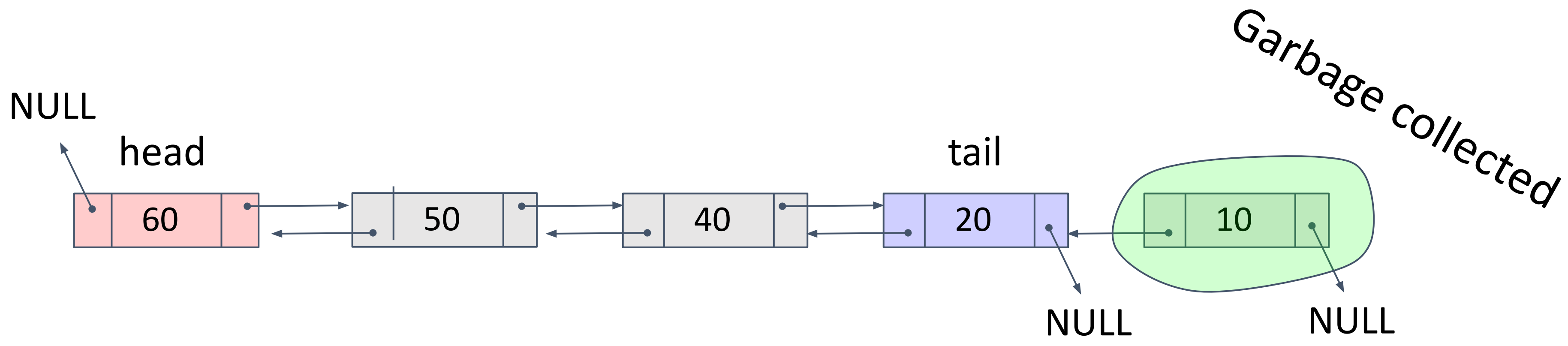
Node Removal (Tail)

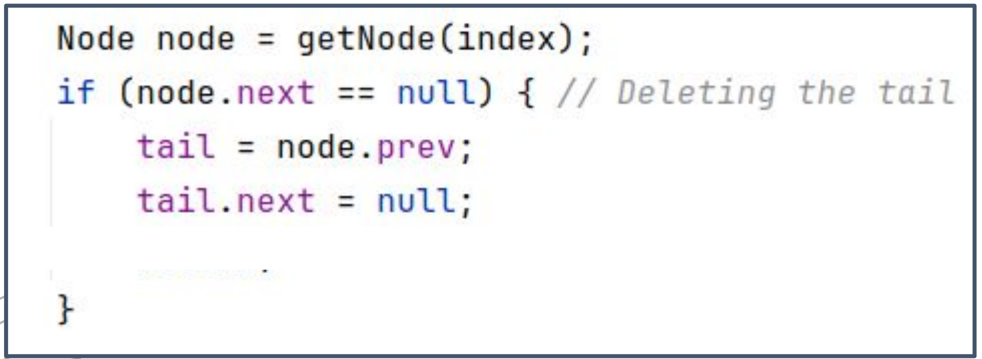




```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail
    tail = node.prev;
    tail.next = null;

}
```

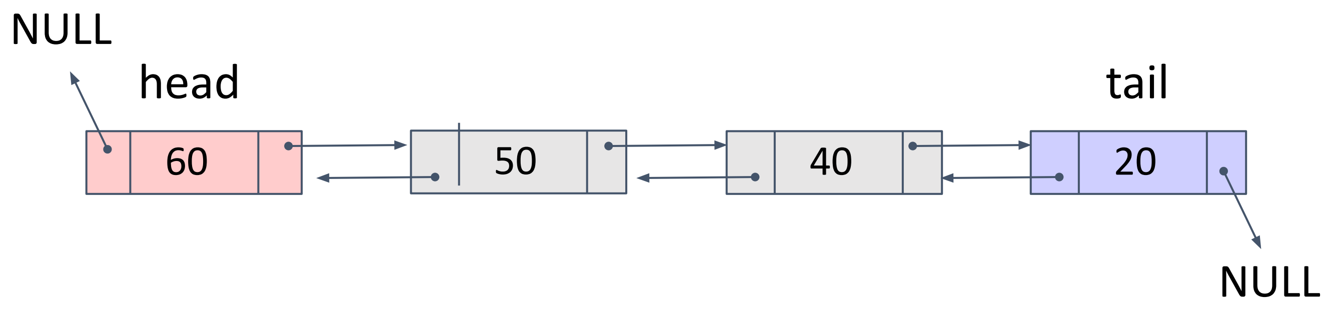
Node Removal (Tail)



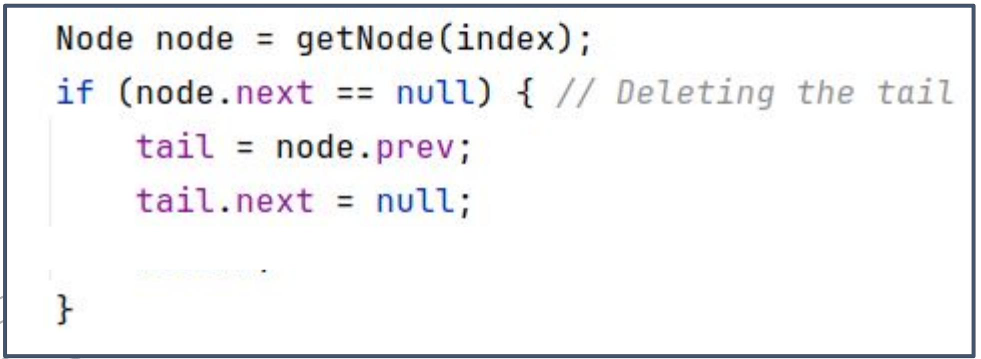


```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail
    tail = node.prev;
    tail.next = null;

}
```

Node Removal (Tail)

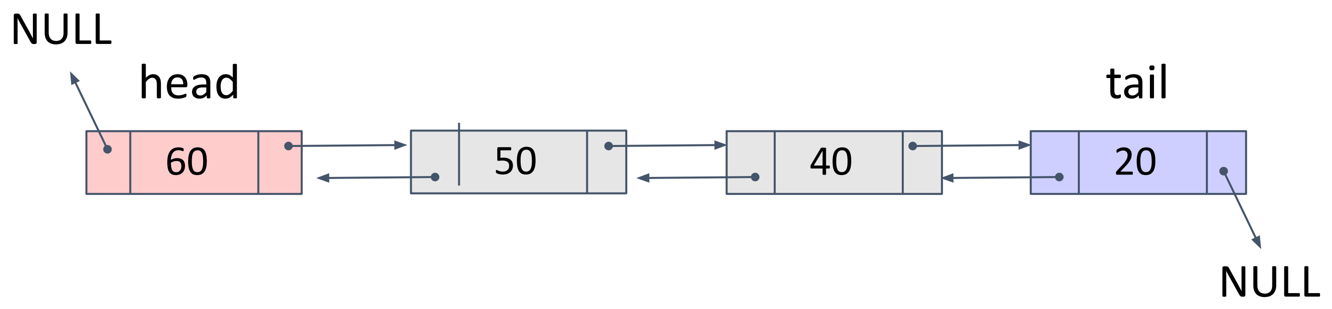


Garbage collected

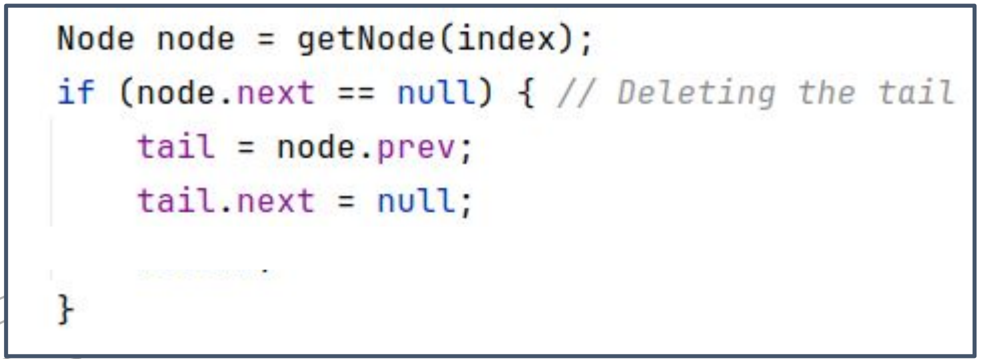


```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail
    tail = node.prev;
    tail.next = null;

}
```

Node Removal (Tail)

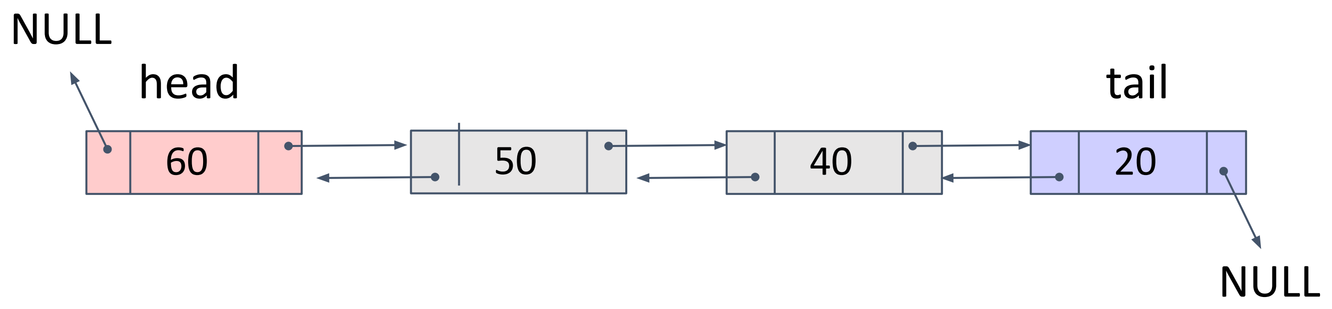


Can we optimize tail deletions?

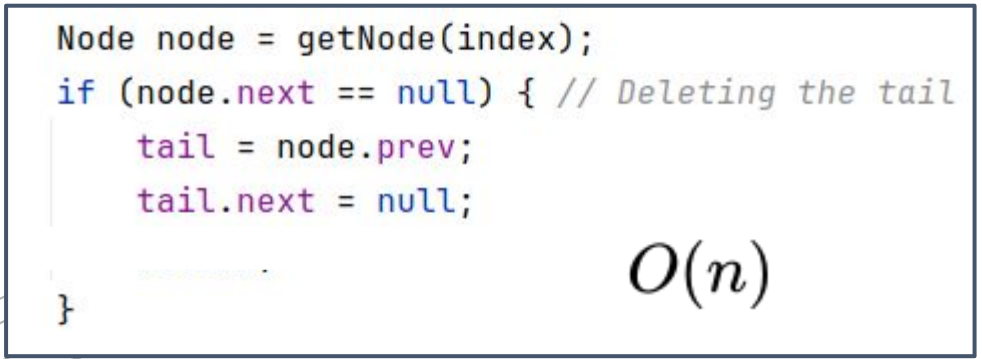


```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail
    tail = node.prev;
    tail.next = null;

}
```

Node Removal (Tail)

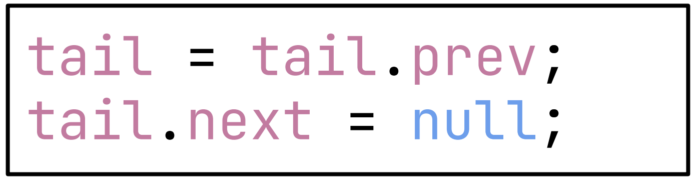


Can we optimize tail deletions?



```
Node node = getNode(index);
if (node.next == null) { // Deleting the tail
    tail = node.prev;
    tail.next = null;
}
```

$O(n)$

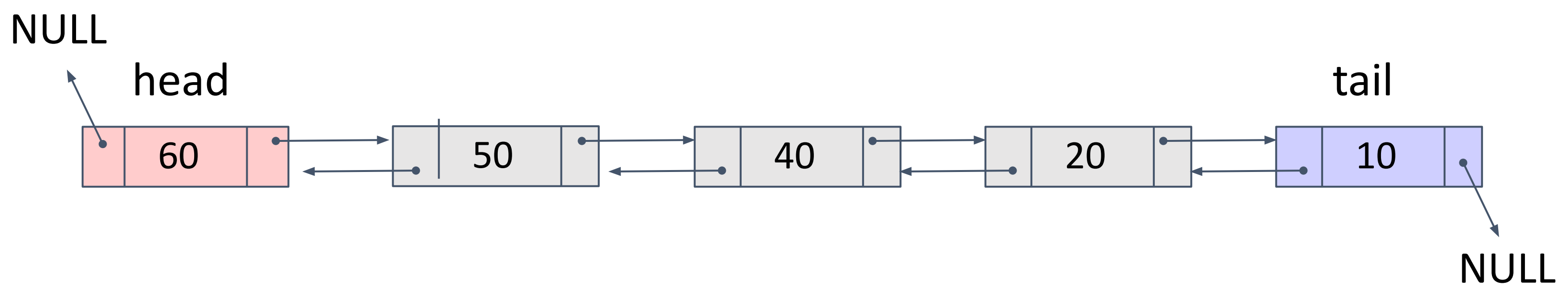


```
tail = tail.prev;
tail.next = null;
```

$O(1)$

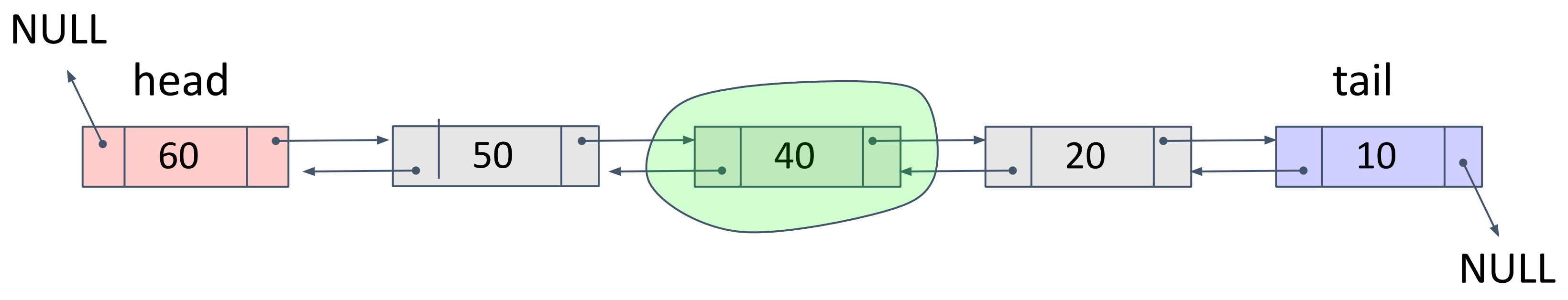
Node Removal (Middle)

Delete the node at index 2



Node Removal (Middle)

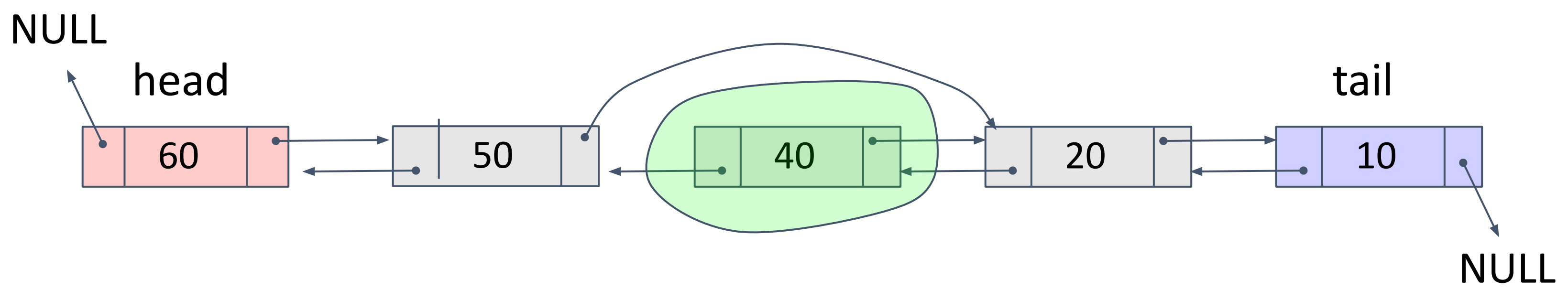
Delete the node at index 2



```
Node node = getNode(index);
```

Node Removal (Middle)

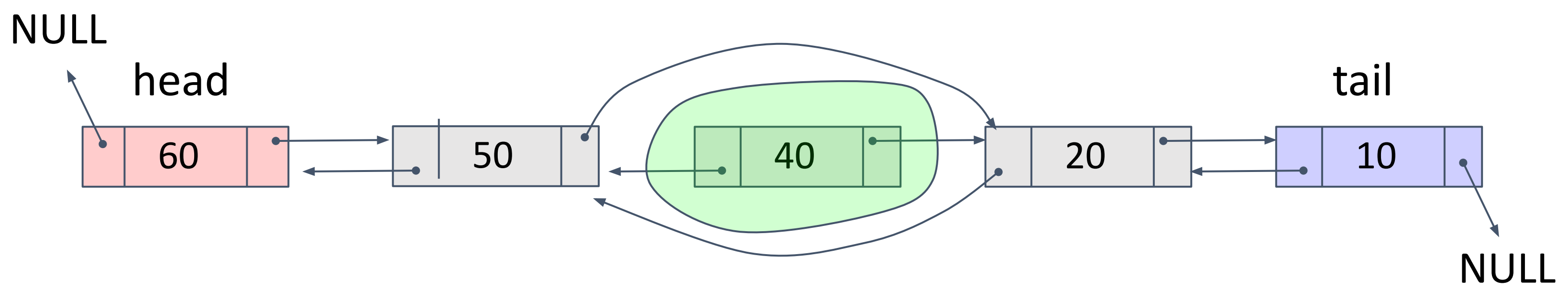
Delete the node at index 2



```
Node node = getNode(index);
node.prev.next = node.next;
```

Node Removal (Middle)

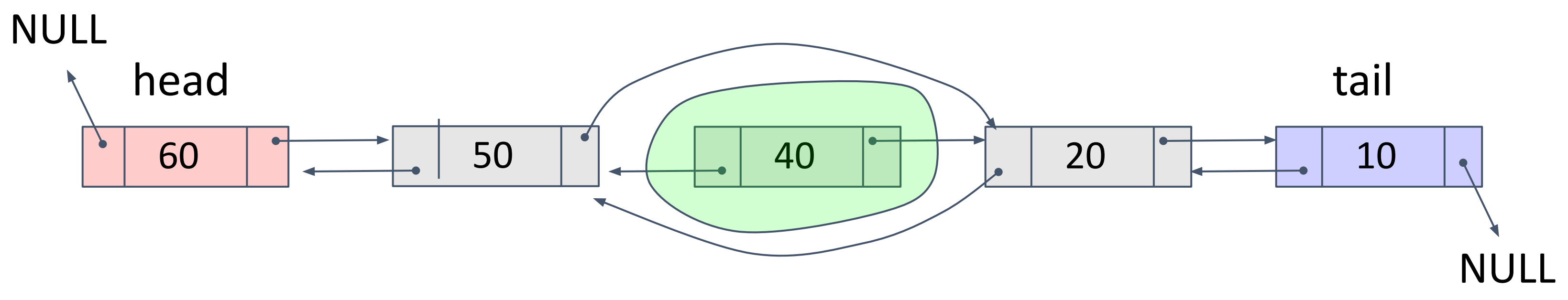
Delete the node at index 2



```
Node node = getNode(index);
node.prev.next = node.next;
node.next.prev = node.prev;
```

Node Removal (Middle)

Delete the node at index 2

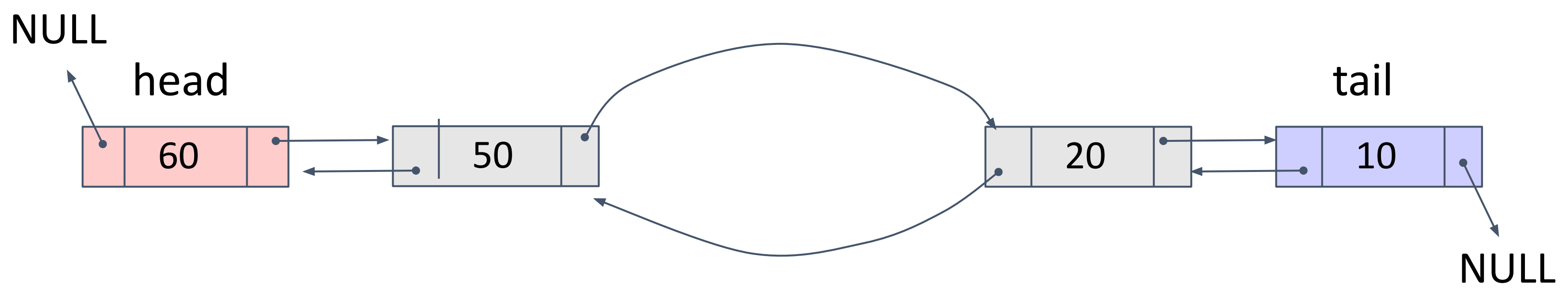


Garbage collected

```
Node node = getNode(index);
node.prev.next = node.next;
node.next.prev = node.prev;
```

Node Removal (Middle)

Delete the node at index 2



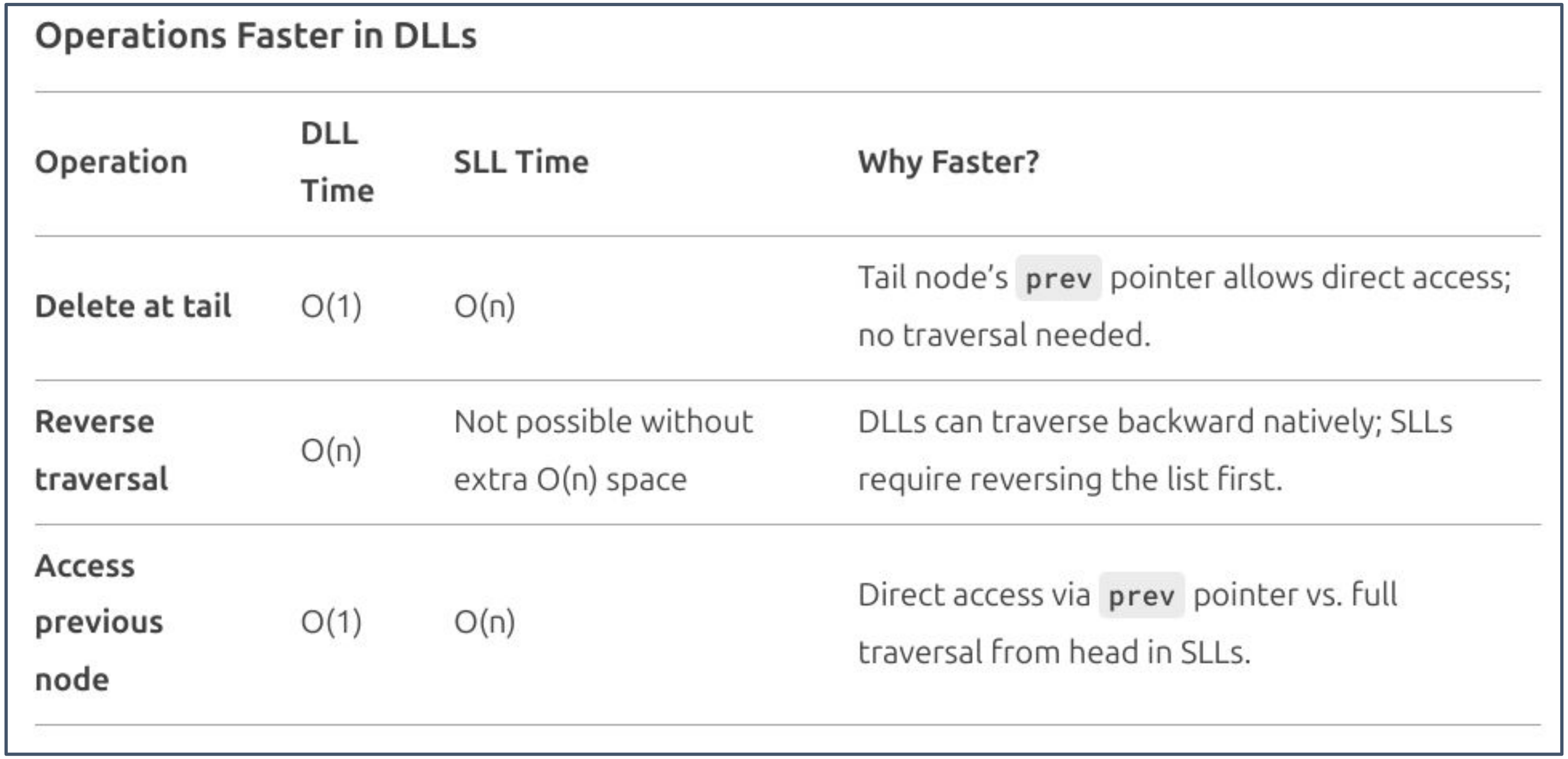
```
Node node = getNode(index);
node.prev.next = node.next;
node.next.prev = node.prev;
```

$O(n)$

Note

- Implementations may differ, but the general idea is same.

Operations Faster in DLLs



Operations Faster in DLLs

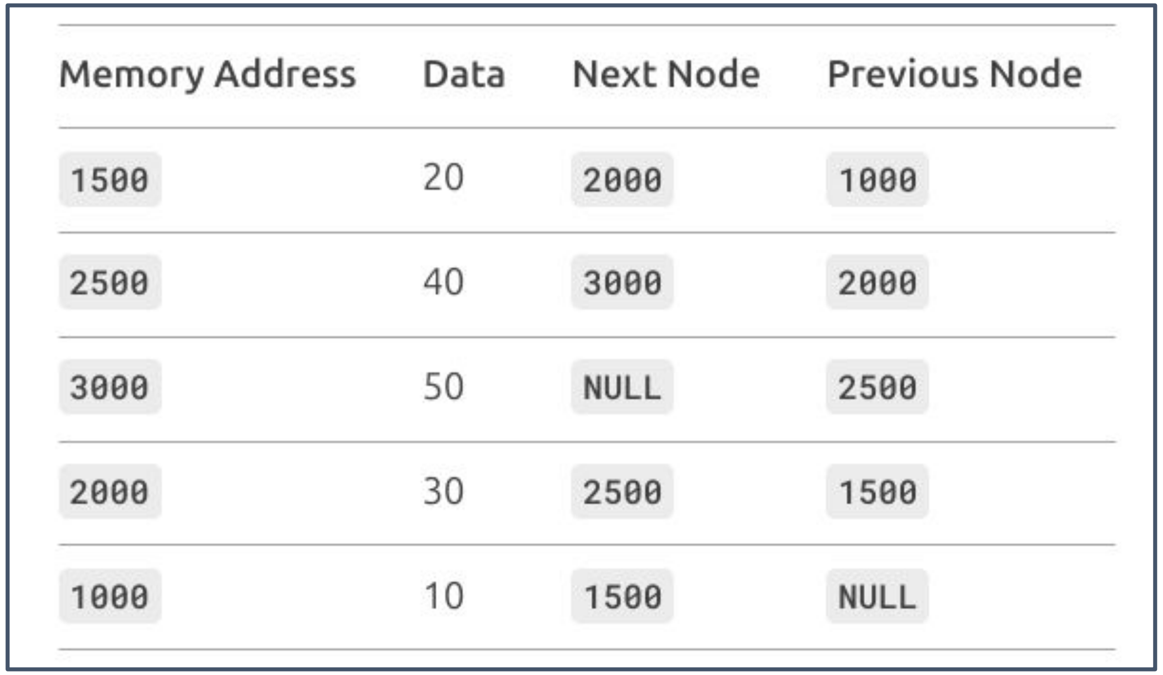
Operation	DLL Time	SLL Time	Why Faster?
Delete at tail	O(1)	O(n)	Tail node's `prev` pointer allows direct access; no traversal needed.
Reverse traversal	O(n)	Not possible without extra O(n) space	DLLs can traverse backward natively; SLLs require reversing the list first.
Access previous node	O(1)	O(n)	Direct access via `prev` pointer vs. full traversal from head in SLLs.

Advantages over SLL

- **Bidirectional Traversal**: Can traverse both forward and backward.
- **Easier Implementation of Complex Data Structures**: Useful in undo/redo operations, navigation systems (e.g., Browsers, File Explorers).
- **More Flexibility in Insertion/Deletion**: Can insert/delete from both ends efficiently.

Exercise: Linked List Construction

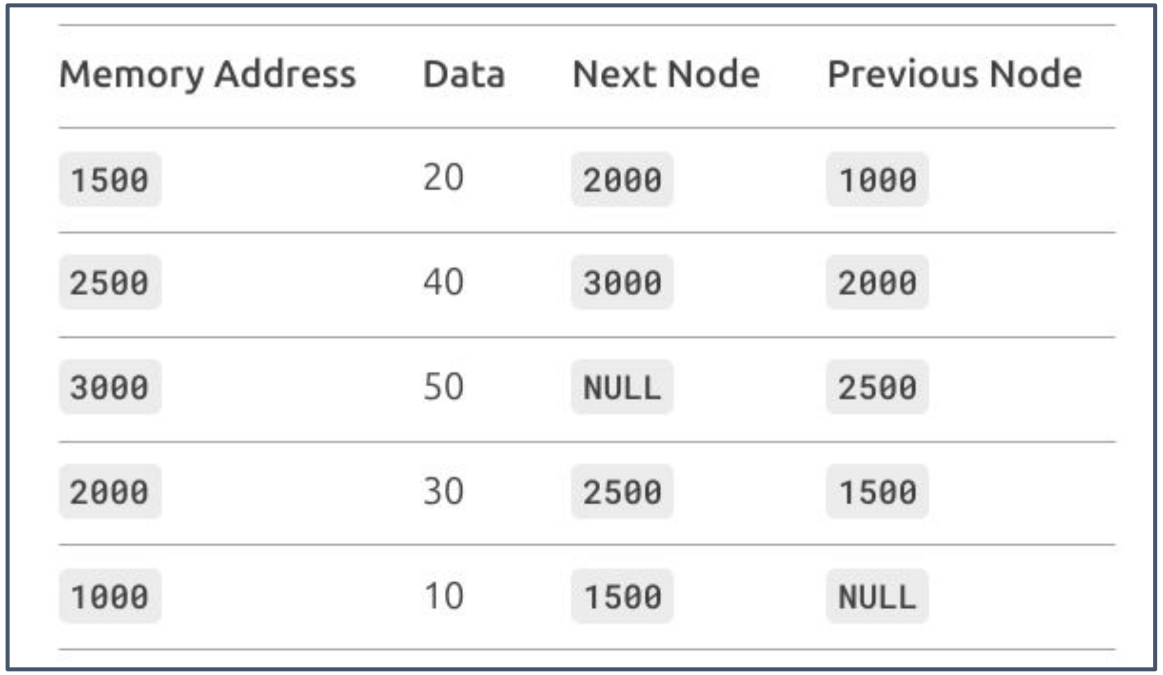
- Construct the linked list from the given table



Memory Address	Data	Next Node	Previous Node
1500	20	2000	1000
2500	40	3000	2000
3000	50	NULL	2500
2000	30	2500	1500
1000	10	1500	NULL

Exercise: Doubly Linked List Construction

- Construct the linked list from the given table



Memory Address	Data	Next Node	Previous Node
1500	20	2000	1000
2500	40	3000	2000
3000	50	NULL	2500
2000	30	2500	1500
1000	10	1500	NULL

← 10 <-> 20 <-> 30 <-> 40 <-> 50 →