

CSE 220

Data Structures

Lecture 02: Linked List Primer

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



Review: Time and Space Complexity

- Big-O is like a "*speedometer*" for your code.
 - It tells you **how fast your code runs** or **how much memory it uses** as the **input grows**.
 - **It's Not Exact:** Big-O doesn't count every single step. Instead, it gives you a general idea of **how your code scales**.
- Example:
 - If your code takes $3n^2 + 2n + 1$ steps, Big-O simplifies it to $O(n^2)$
 - Why? Because as n gets really big, the n^2 part dominates the others



Exercise

```
public void test1(int[] arr) { no usages
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

Exercise

TC: $O(n)$

SC: $O(1)$

```
public void test1(int[] arr) { no usages
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```



Exercise

```
public void test1_a(int[] arr) { no usages
    // Make a copy of the array
    int[] copy = Arrays.copyOf(arr, arr.length);
    // print first 10 elemements
    for (int i = 0; i < 10; i++) {
        System.out.println(copy[i]);
    }
}
```

Exercise

TC: $O(n)$

SC: $O(n)$

```
public void test1_a(int[] arr) { no usages
    // Make a copy of the array
    int[] copy = Arrays.copyOf(arr, arr.length);
    // print first 10 elemements
    for (int i = 0; i < 10; i++) {
        System.out.println(copy[i]);
    }
}
```

Exercise

```
public void test1_b(int rows, int cols) {  
    int size = rows * cols;  
    int[][] matrix = new int[rows][cols];  
    for (int i = 0; i < size; i++) {  
        int row = i / cols;  
        int col = i % cols;  
        matrix[row][col] = i;  
    }  
}
```

Exercise

TC: $O(n^2)$

SC: $O(n^2)$

```
public void test1_b(int rows, int cols) {  
    int size = rows * cols;  
    int[][] matrix = new int[rows][cols];  
    for (int i = 0; i < size; i++) {  
        int row = i / cols;  
        int col = i % cols;  
        matrix[row][col] = i;  
    }  
}
```


Exercise

```
public void test2(int n) { no usages
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < n; k++) {
                System.out.println("Some operation...");
            }
        }
    }
}
```

Exercise

TC: $O(n^2)$

SC: $O(1)$

```
public void test2(int n) { no usages
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < n; k++) {
                System.out.println("Some operation...");
            }
        }
    }
}
```



Exercise

```
public void test3(int[] sortedArr, int[] valuesToFind) { no usages
    for (int i : valuesToFind) {
        // binary search
        int idx = Arrays.binarySearch(sortedArr, i);
        if (idx >= 0) {
            System.out.println("Value found at index: " + idx);
        } else {
            System.out.println("Value not found");
        }
    }
}
```

Exercise

TC: $O(n \log n)$
SC: $O(1)$

```
public void test3(int[] sortedArr, int[] valuesToFind) { no usages
    for (int i : valuesToFind) {
        // binary search
        int idx = Arrays.binarySearch(sortedArr, i);
        if (idx >= 0) {
            System.out.println("Value found at index: " + idx);
        } else {
            System.out.println("Value not found");
        }
    }
}
```

Exercise

```
public ArrayList<ArrayList<Integer>> findPowerSets(int[] arr) { 1 usage
    int n = arr.length;
    int total = 1 << n;
    ArrayList<ArrayList<Integer>> powerSet = new ArrayList<>();
    for (int i = 0; i < total; i++) {
        ArrayList<Integer> subset = new ArrayList<>();
        for (int j = 0; j < n; j++) {
            if ((i & (1 << j)) > 0) {
                subset.add(arr[j]);
            }
        }
        powerSet.add(subset);
    }
    return powerSet;
}
```

Exercise

TC: $O(n \cdot 2^n)$

SC: $O(n \cdot 2^n)$

```
public ArrayList<ArrayList<Integer>> findPowerSets(int[] arr) { 1 usage
    int n = arr.length;
    int total = 1 << n;
    ArrayList<ArrayList<Integer>> powerSet = new ArrayList<>();
    for (int i = 0; i < total; i++) {
        ArrayList<Integer> subset = new ArrayList<>();
        for (int j = 0; j < n; j++) {
            if ((i & (1 << j)) > 0) {
                subset.add(arr[j]);
            }
        }
        powerSet.add(subset);
    }
    return powerSet;
}
```

Goal

- Design data structures that lets us perform the following tasks in an efficient way:
 - **Insertion** of a new element
 - **Deletion** of an existing element
 - **Checking** if an element exists
 - **Modification** of an existing element
 - Finding the **minimum** / **maximum** element
- There is no “one size fits all” solution!



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).



Interactive Visualization

<https://visualgo.net/en/list>

The screenshot shows the Visualgo.net website interface for a linked list. The top navigation bar includes "VISUALGO.NET", a language dropdown set to "en", and a path "/list". The main menu has tabs for "LINKED LIST", "STACK", "QUEUE", "DLL", and "DEQUE". The current view is "Exploration Mode" with a "LOGIN" button.

The central visualization shows a linked list with five nodes containing values 49, 78, 16, 34, and 10. The first node (49) is labeled "head/0" and the last node (10) is labeled "tail/5". The next pointer of the last node points to a green node with the value -29.

At the bottom left, there is a control panel with buttons for "Create(A)", "Search", "Insert", and "Remove". Below these are input fields for specifying the index and value for operations. The "Remove" section is active, showing "i = 4" and "v = 22" with a "Go" button.

On the right side, a panel titled "Remove i = N-1 (Tail)" provides instructions and code for the operation:

```

Delete tmp (the previous tail) then update the tail pointer to pre (the current tail). The whole process is O(N) just to find the pre pointer.

if empty, do nothing
Vertex pre = head
tmp = head.next
while (tmp.next != null)
    pre = pre.next; tmp = tmp.next;
pre.next = null
delete tmp, tail = pre
    
```

At the bottom of the interface, there is a progress bar and a "1x" speed indicator, along with navigation controls (back, forward, etc.). The footer includes links for "About", "Team", "Terms of use", and "Privacy Policy".

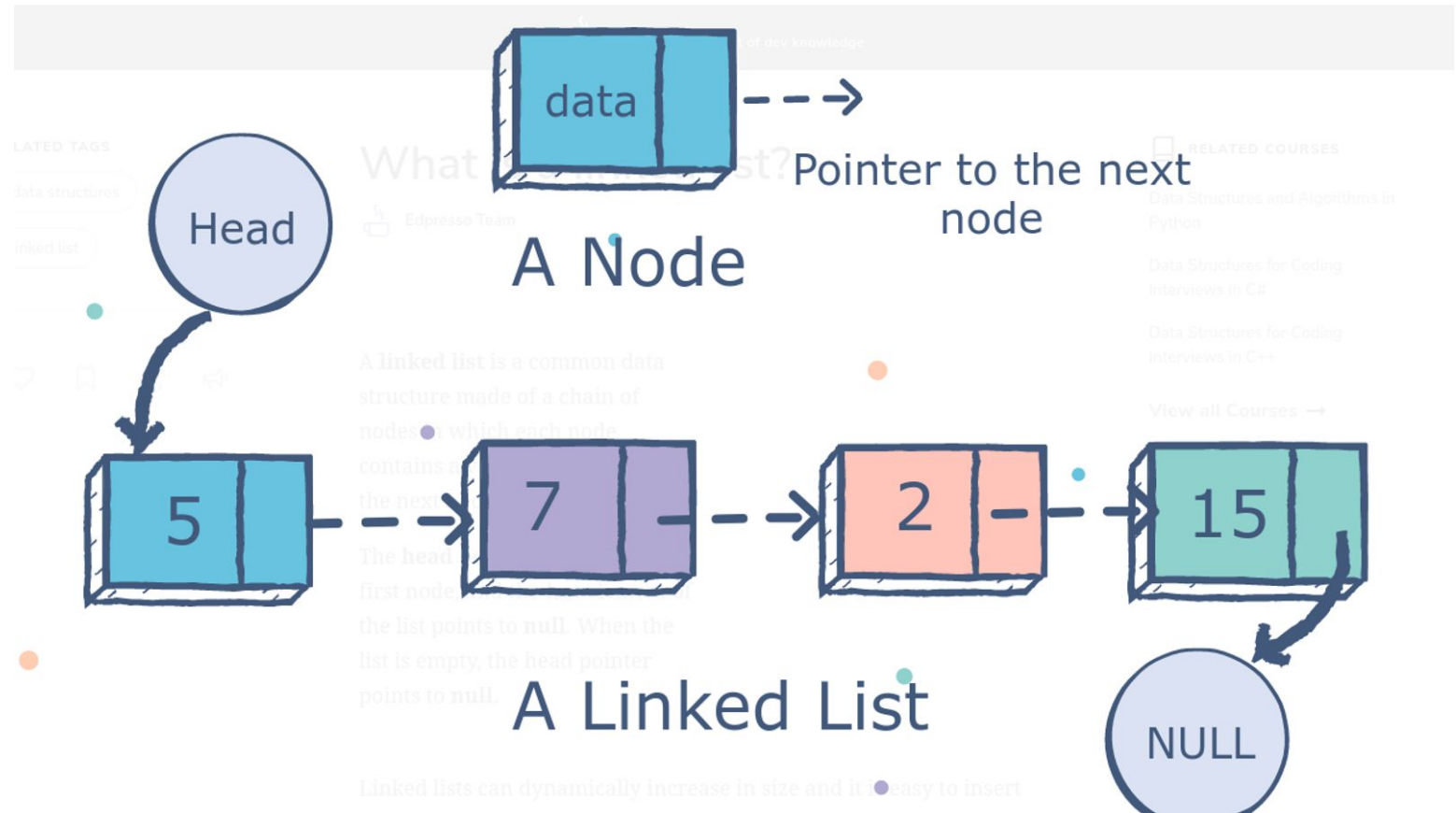
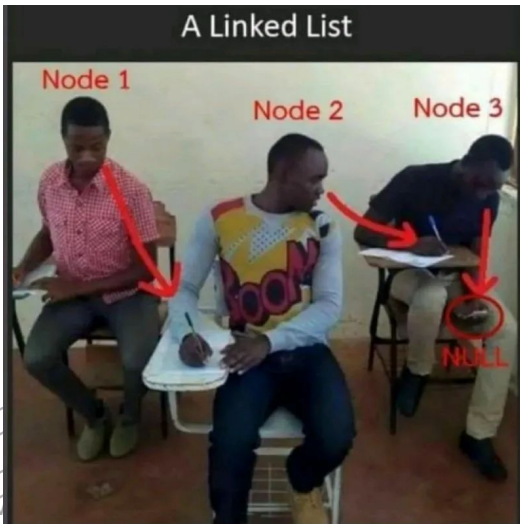


Linked List Overview

A node is the building block for linked lists.

Contains two key things:

1. Data
2. Reference to next node

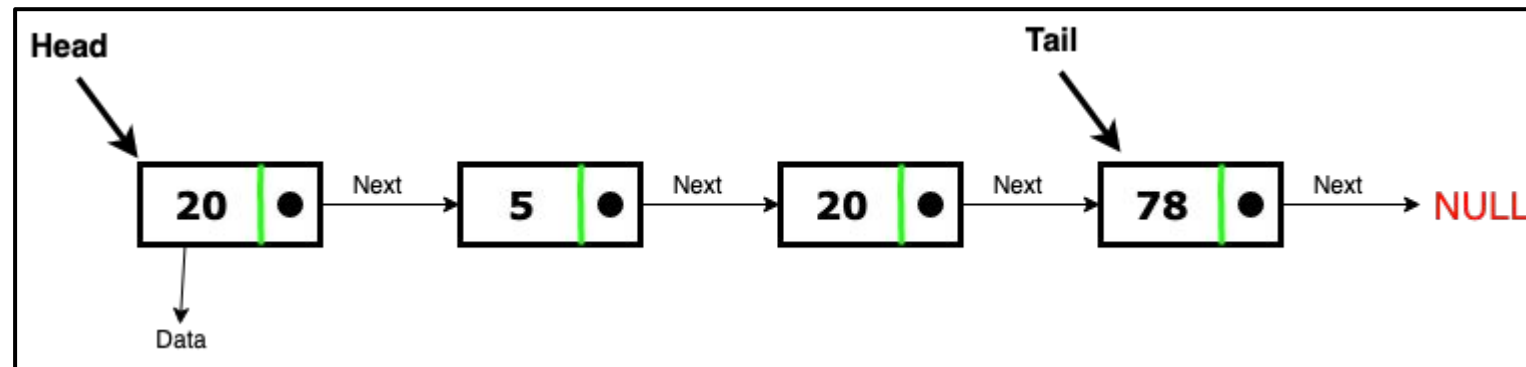


<https://dev.to/tastaslim/an-introduction-to-linked-list-1bmp>

Linked List Overview

The first element is often called “**head**”

The last element is often called “**tail**”



Node

Linked List

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

```
private Node head;
```



Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

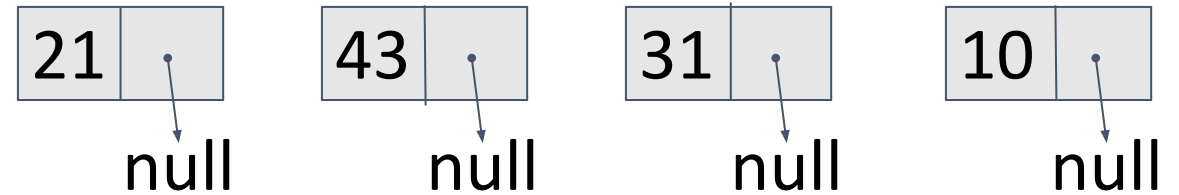
```
Node n1 = new Node( elem: 21);  
Node n2 = new Node( elem: 43);  
Node n3 = new Node( elem: 31);  
Node n4 = new Node( elem: 10);
```



Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

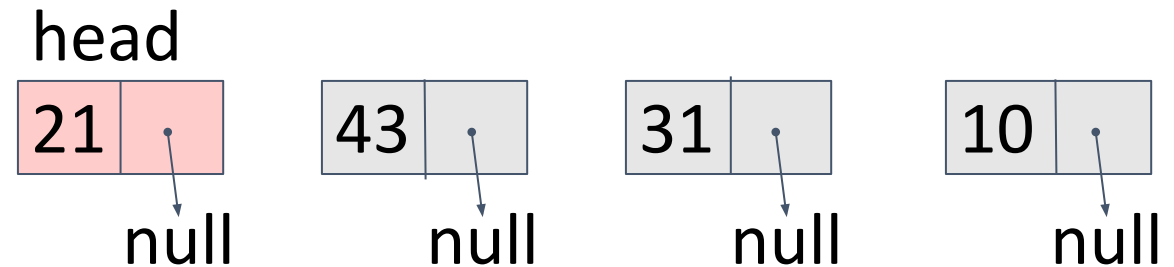
```
Node n1 = new Node( elem: 21);  
Node n2 = new Node( elem: 43);  
Node n3 = new Node( elem: 31);  
Node n4 = new Node( elem: 10);
```



Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

```
Node n1 = new Node( elem: 21);  
Node n2 = new Node( elem: 43);  
Node n3 = new Node( elem: 31);  
Node n4 = new Node( elem: 10);  
  
Node head = n1;
```

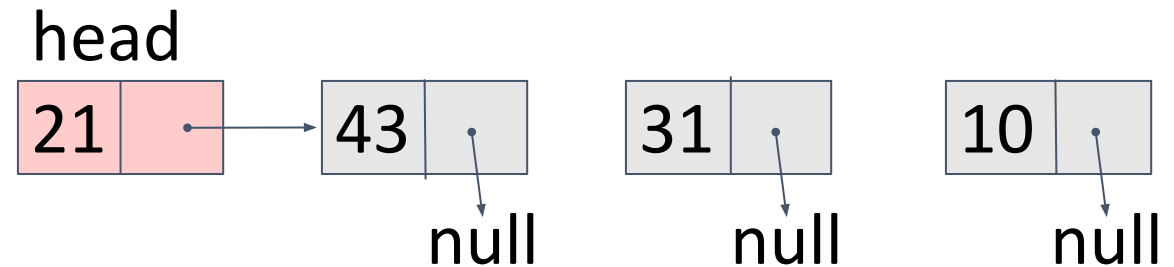


Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

```
Node n1 = new Node( elem: 21);
Node n2 = new Node( elem: 43);
Node n3 = new Node( elem: 31);
Node n4 = new Node( elem: 10);
```

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

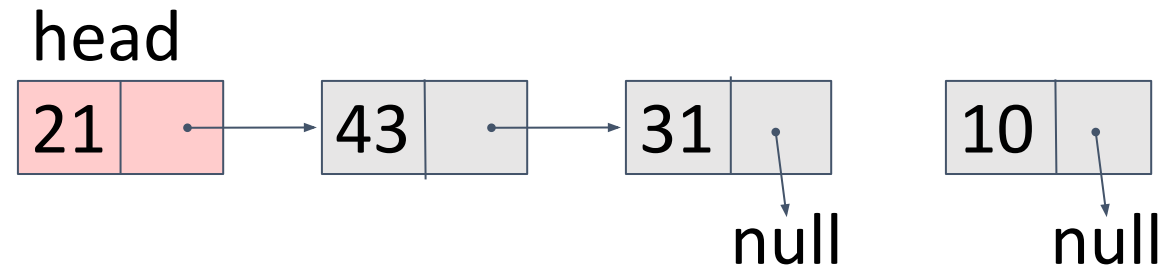


Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

```
Node n1 = new Node( elem: 21);
Node n2 = new Node( elem: 43);
Node n3 = new Node( elem: 31);
Node n4 = new Node( elem: 10);
```

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

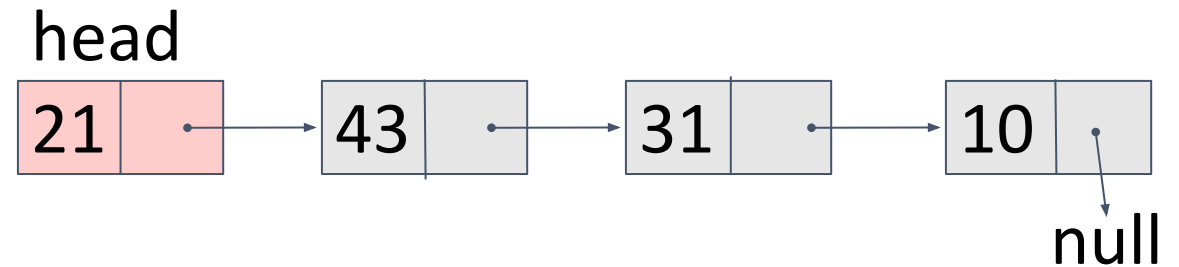


Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

```
Node n1 = new Node( elem: 21);
Node n2 = new Node( elem: 43);
Node n3 = new Node( elem: 31);
Node n4 = new Node( elem: 10);
```

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

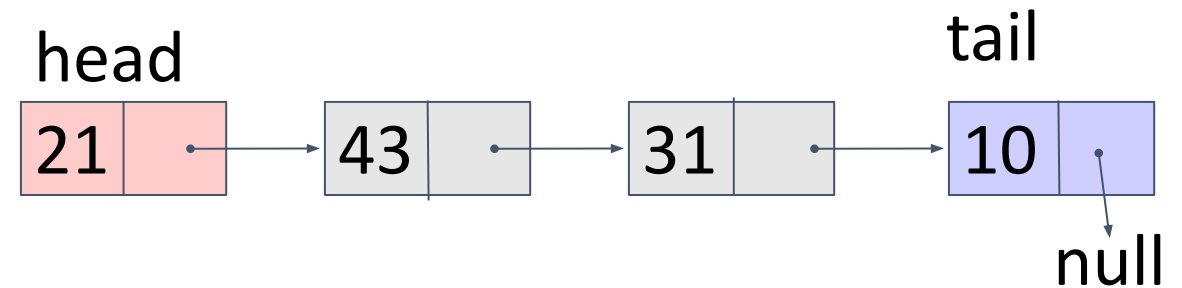


Linked List Creation

- Create a linked list from these elements: 21, 43, 31, 10

```
Node n1 = new Node( elem: 21);  
Node n2 = new Node( elem: 43);  
Node n3 = new Node( elem: 31);  
Node n4 = new Node( elem: 10);
```

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



Linked List Creation

- Create a linked list from an array

arr = [21, 43, 31, 10]


```
// 1. Create a Linked List from an array  
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++) {  
        current.next = new Node(arr[i]);  
        current = current.next;  
    }  
}
```

$O(n)$

Linked List Creation

- Create a linked list from an array

arr = [21, 43, 31, 10]



cur = head



cur = cur.next

```
// 1. Create a Linked List from an array
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++) {
        current.next = new Node(arr[i]);
        current = current.next;
    }
}
```

$O(n)$

Sequential Traversal

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

$O(n)$

Output: 21->43->31->10->

Think: Why did we need a “current” node?

Element Access

```
// 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
}
```

$O(n)$

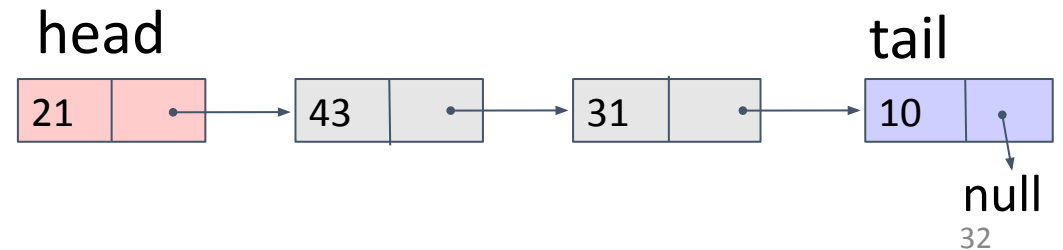
```
// 5. Retrieve a node from an index
public Node getNode(int index) {
    int currentIndex = 0;
    Node current = head;
    while (current != null) {
        if (currentIndex == index) {
            return current;
        }
        current = current.next;
        currentIndex++;
    }
    return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

$O(n)$

Element Insertion (At the beginning)

- Prepend 99 to the list

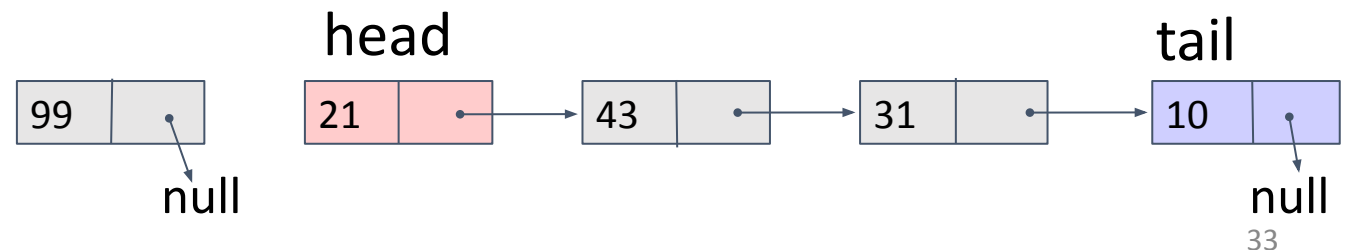
```
// insert at beginning
```



Element Insertion (At the beginning)

- Prepend 99 to the list

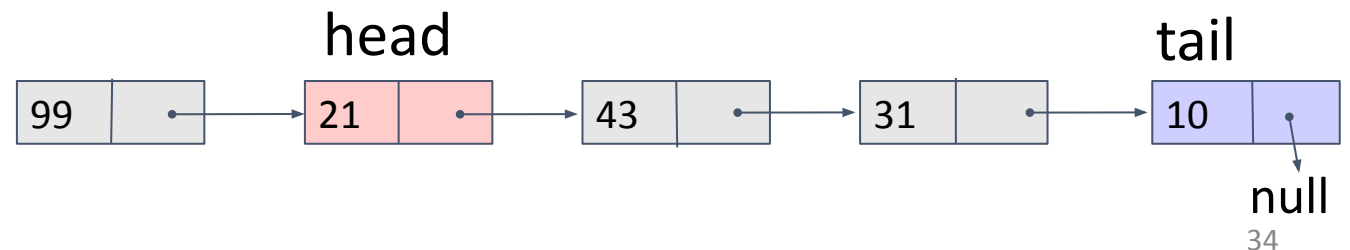
```
// insert at beginning  
Node newNode = new Node( elem: 99);
```



Element Insertion (At the beginning)

- Prepend 99 to the list

```
// insert at beginning  
Node newNode = new Node( elem: 99);  
newNode.next = head;
```

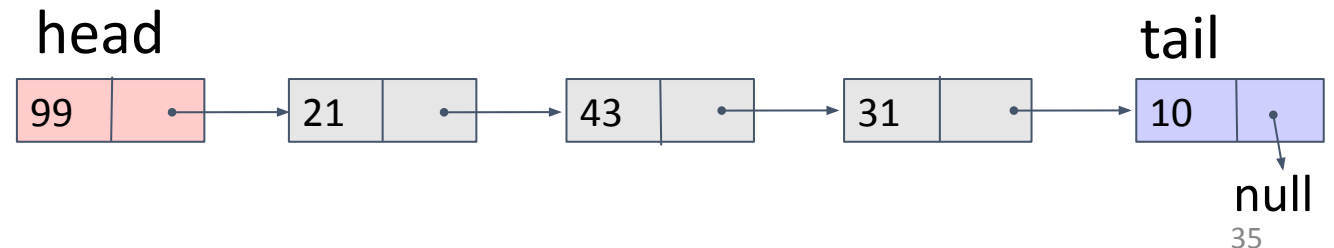


Element Insertion (At the beginning)

- Prepend 99 to the list

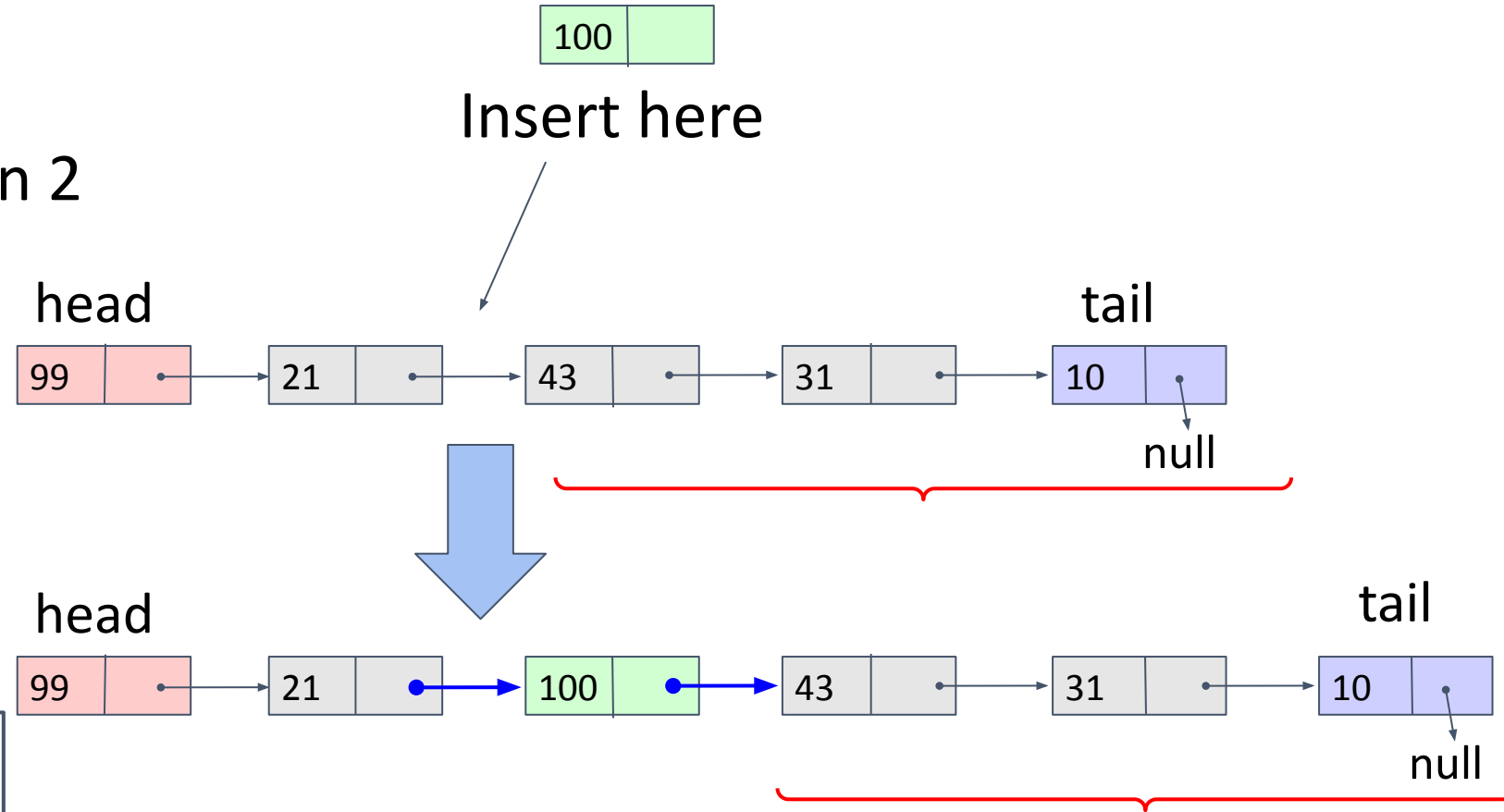
```
// insert at beginning  
Node newNode = new Node( elem: 99);  
newNode.next = head;  
head = newNode;
```

$O(1)$



Element Insertion (any position)

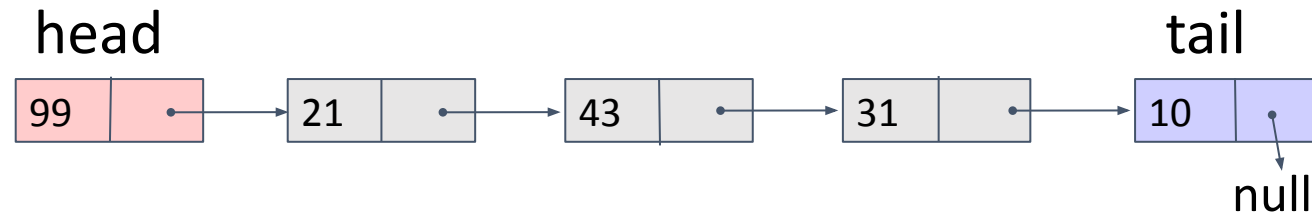
- Insert 100 at position 2



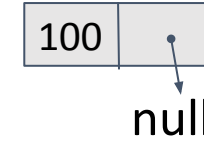
```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);
if (prev != null) { // Insert anywhere
    newNode.next = prev.next;
    prev.next = newNode;
}
```

Element Insertion (any position)

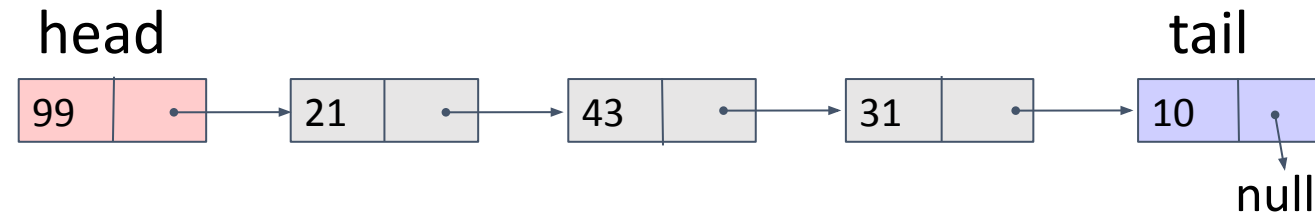
- Insert 100 at position 2



Element Insertion (any position)



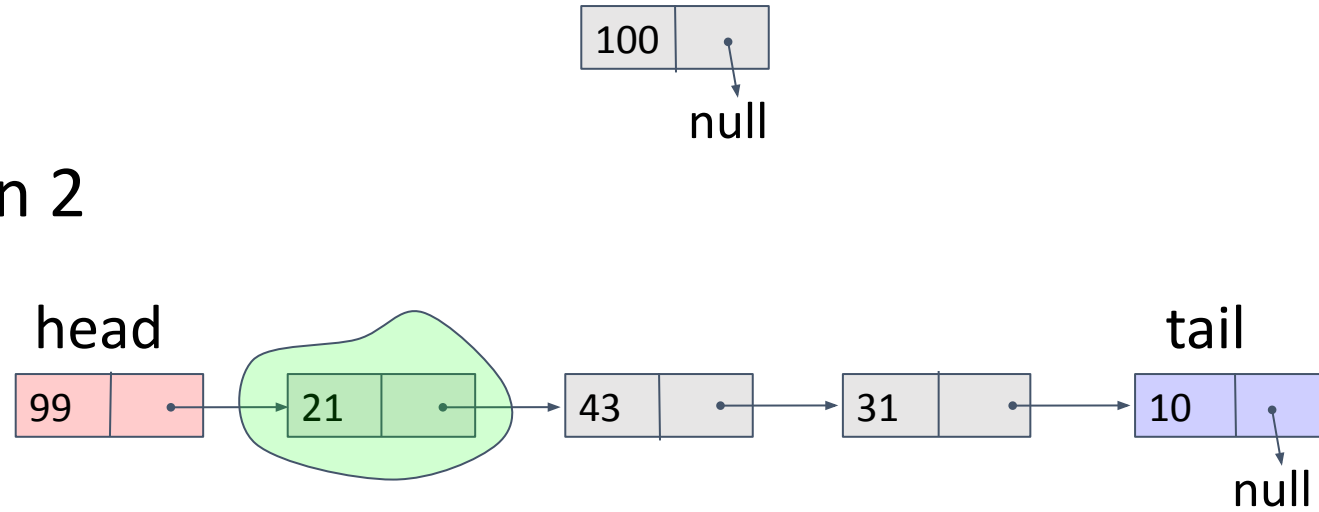
- Insert 100 at position 2



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

Element Insertion (any position)

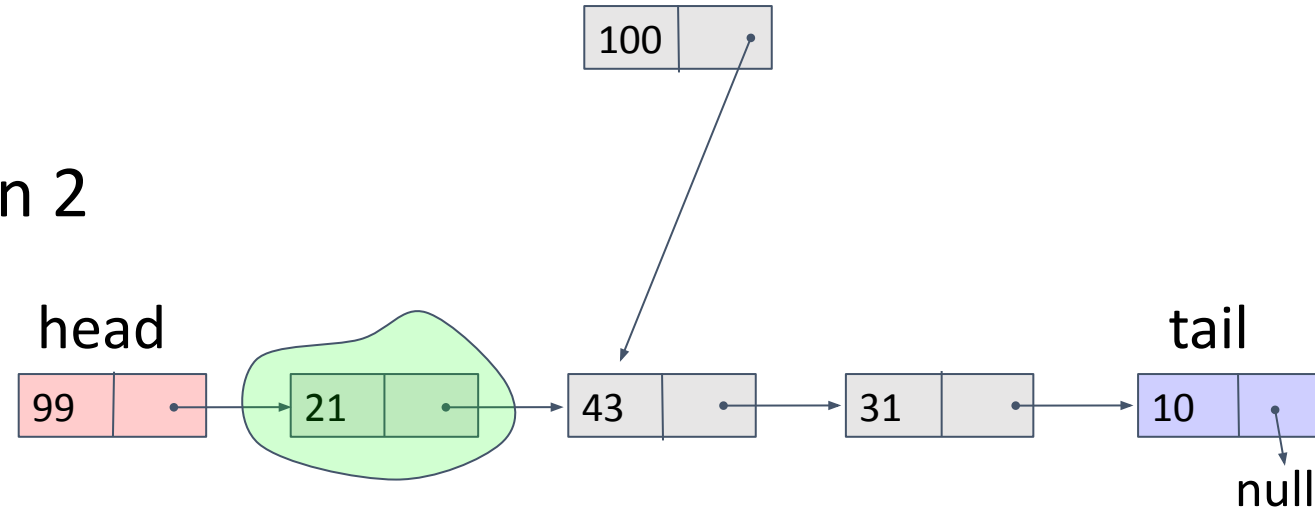
- Insert 100 at position 2



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

Element Insertion (any position)

- Insert 100 at position 2

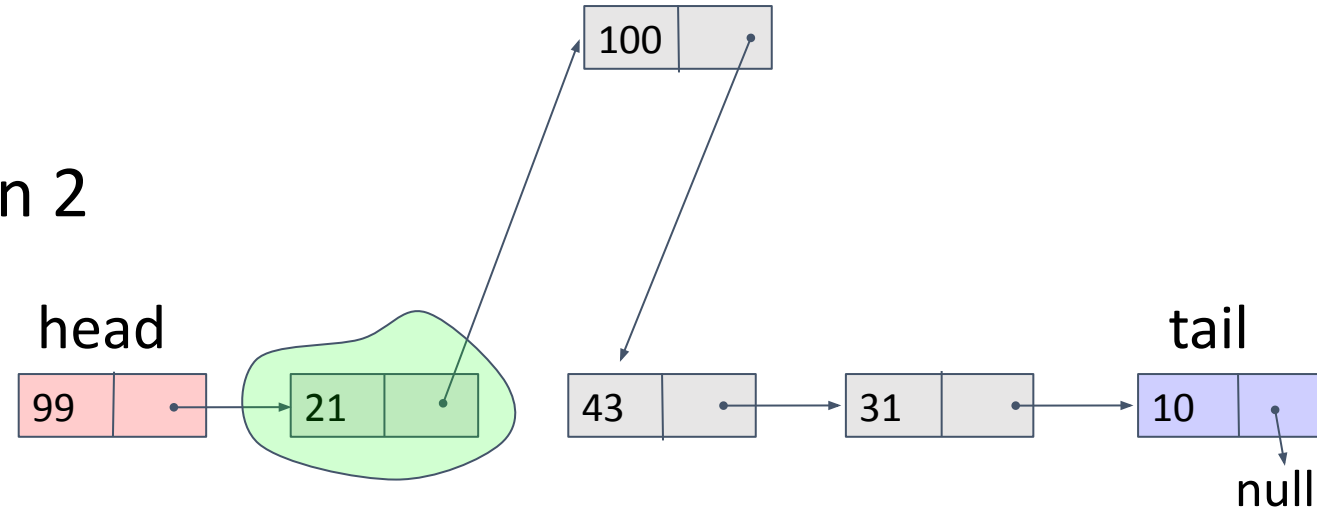


```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);

newNode.next = prev.next;
```


Element Insertion (any position)

- Insert 100 at position 2

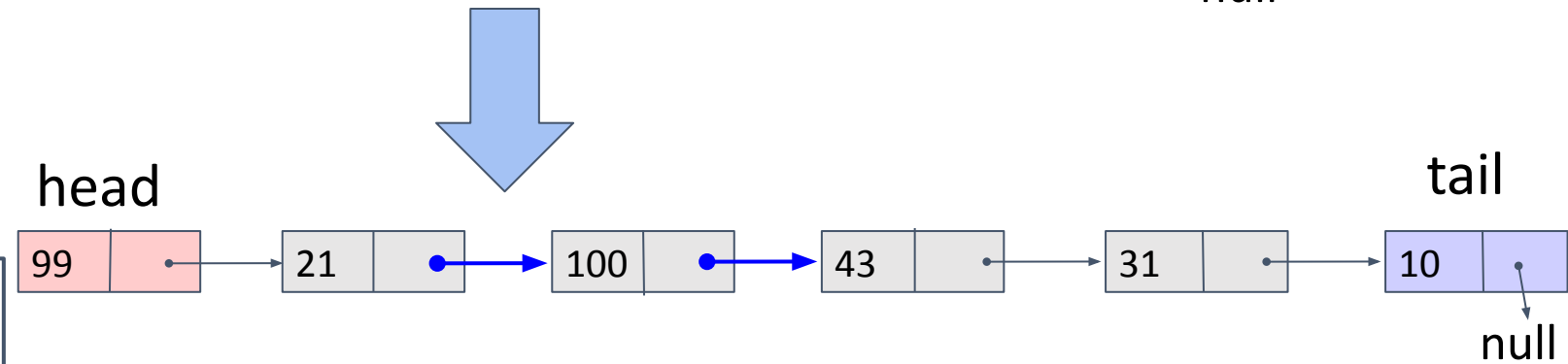
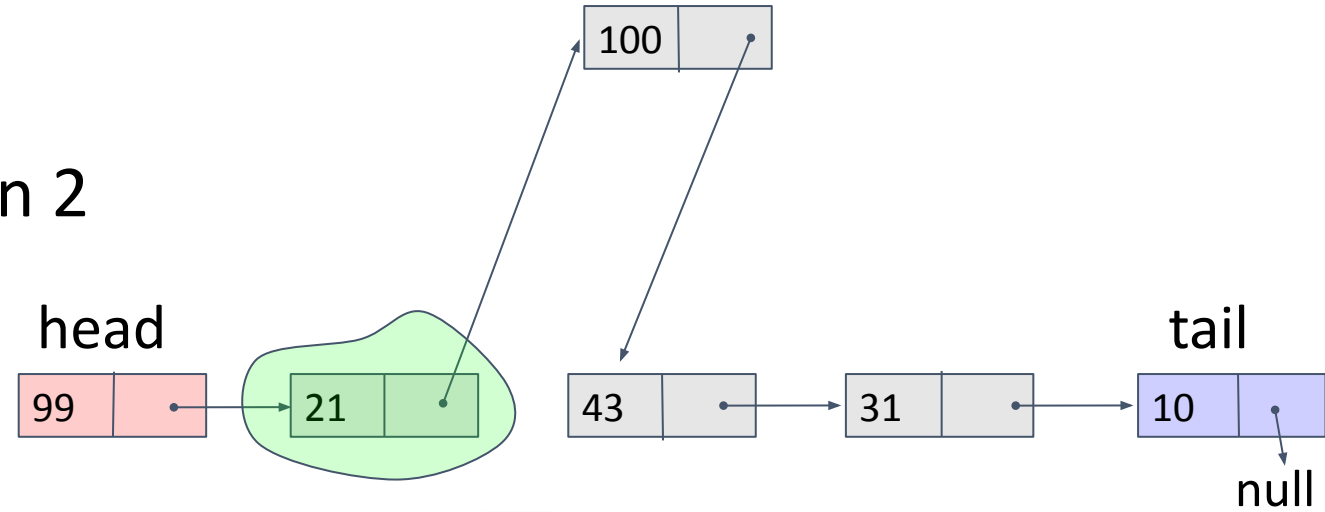


```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);

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

Element Insertion (any position)

- Insert 100 at position 2

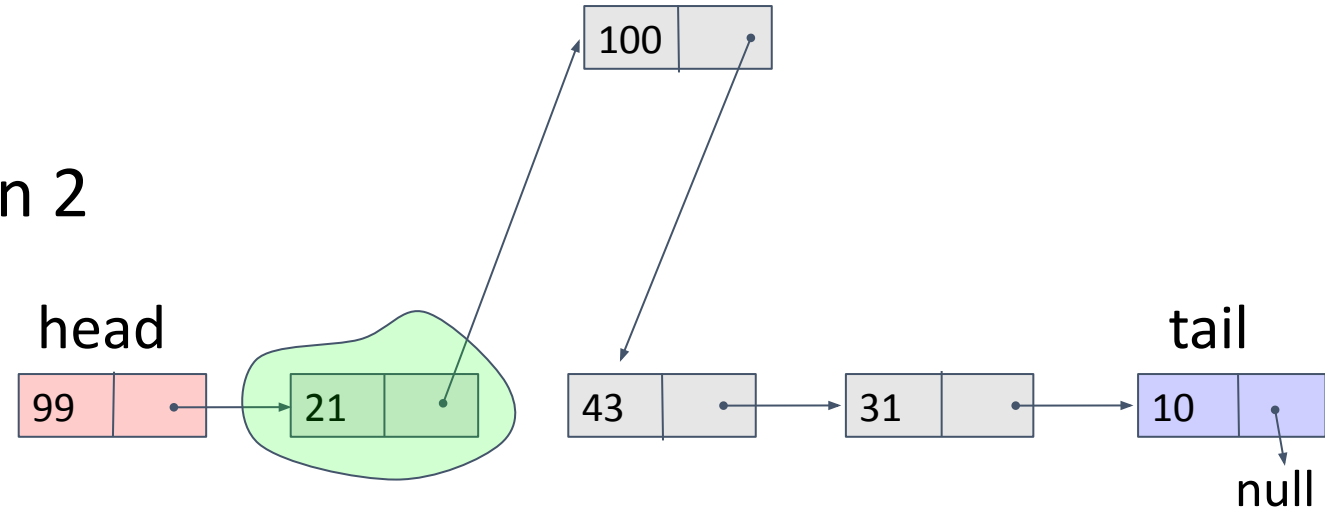


```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);

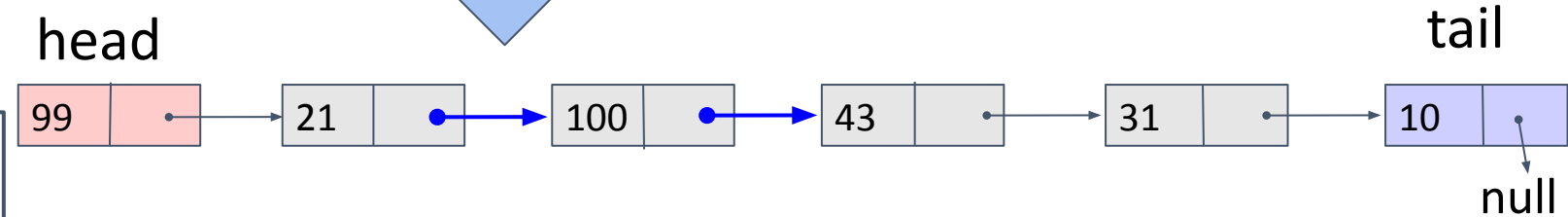
newNode.next = prev.next;
prev.next = newNode;
```

Element Insertion (any position)

- Insert 100 at position 2



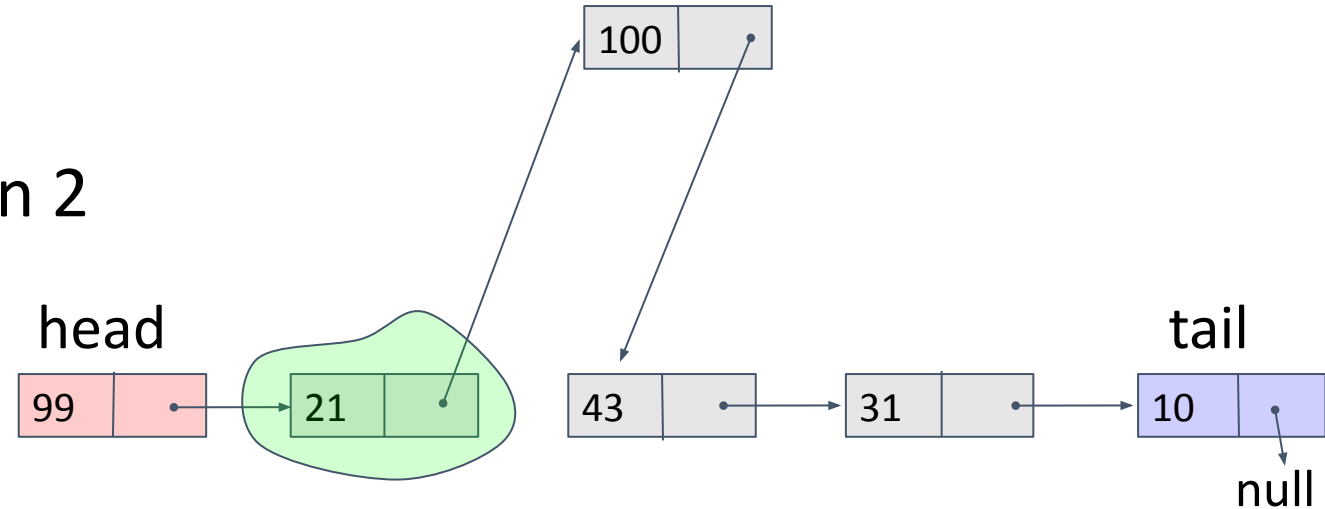
$O(n)$



```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);
if (prev != null) { // Cannot insert before head
    newNode.next = prev.next;
    prev.next = newNode;
}
```

Element Insertion (any position)

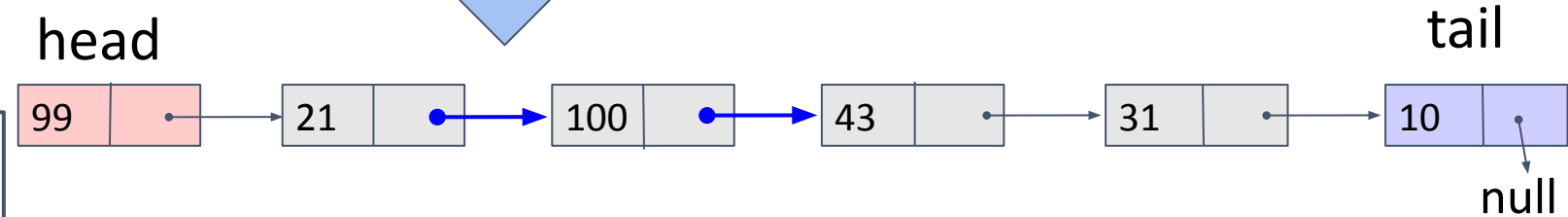
- Insert 100 at position 2



$O(n)$

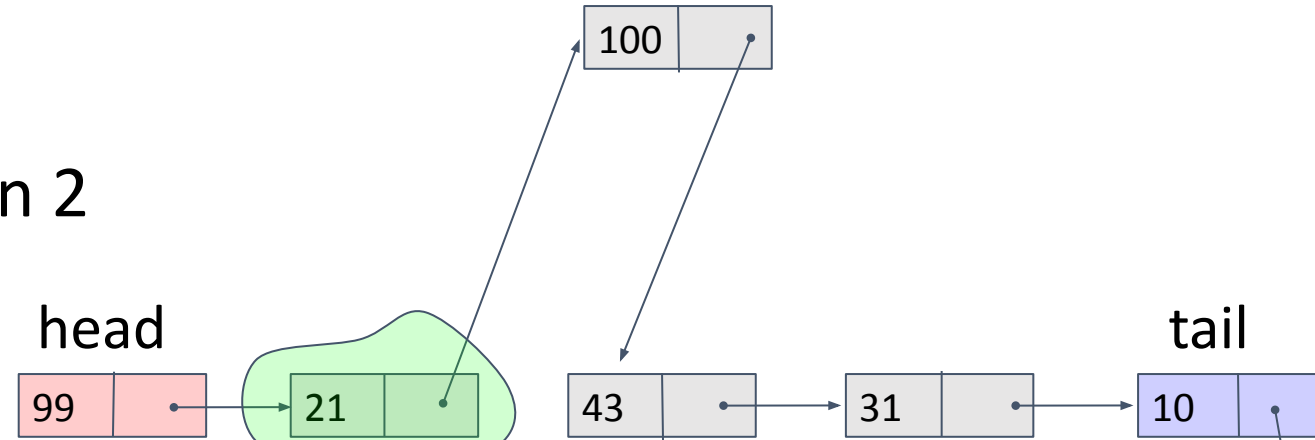
```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);
if (prev != null) { // Cannot insert before head
    newNode.next = prev.next;
    prev.next = newNode;
}
```

Ordering is important!



Element Insertion (any position)

- Insert 100 at position 2



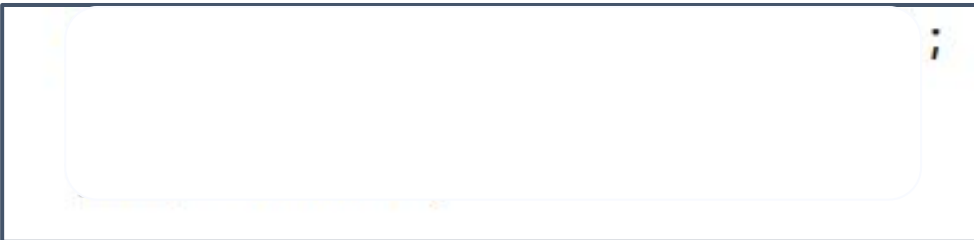
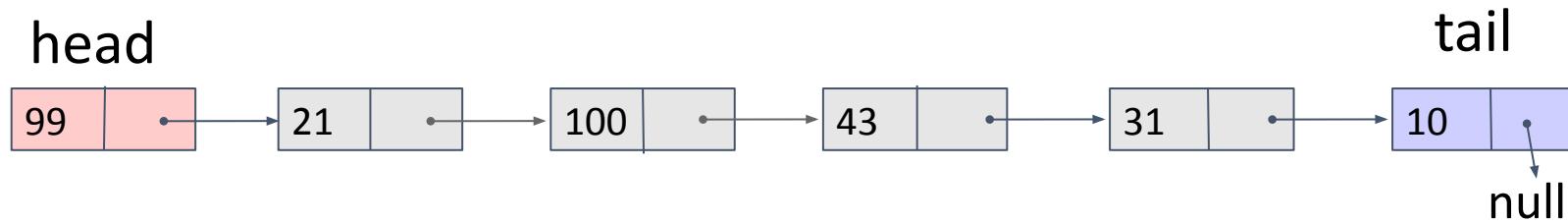
$O(n)$

```
Node newNode = new Node(elem);
Node prev = getNode(index: index - 1);
if (prev != null) { // Cannot insert before head
    newNode.next = prev.next;
    prev.next = newNode;
}
```

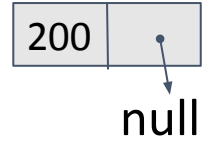


Optimizing Tail Insertion

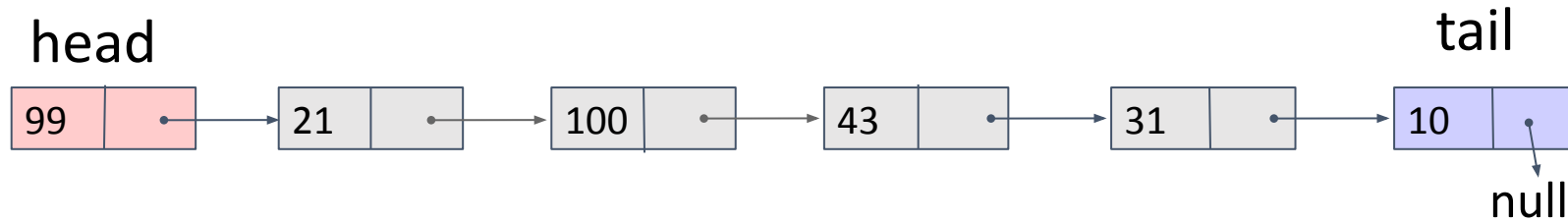
- Append 200 (Insert at the end)



Optimizing Tail Insertion



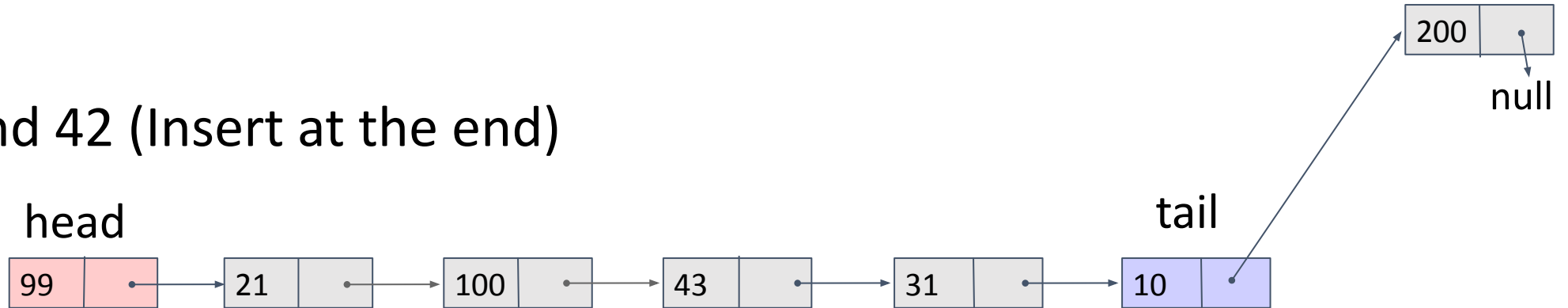
- Append 200 (Insert at the end)



```
Node newNode = new Node( elem: 200);
```

Optimizing Tail Insertion

- Append 42 (Insert at the end)

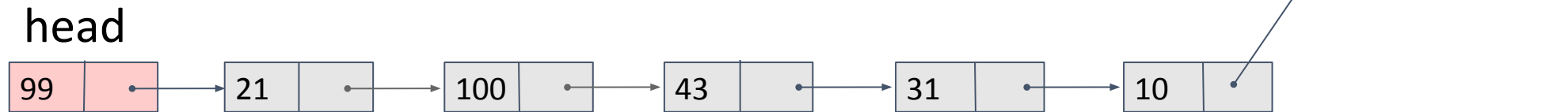


```
Node newNode = new Node( elem: 200);
tail.next = newNode;
```



Optimizing Tail Insertion

- Append 42 (Insert at the end)



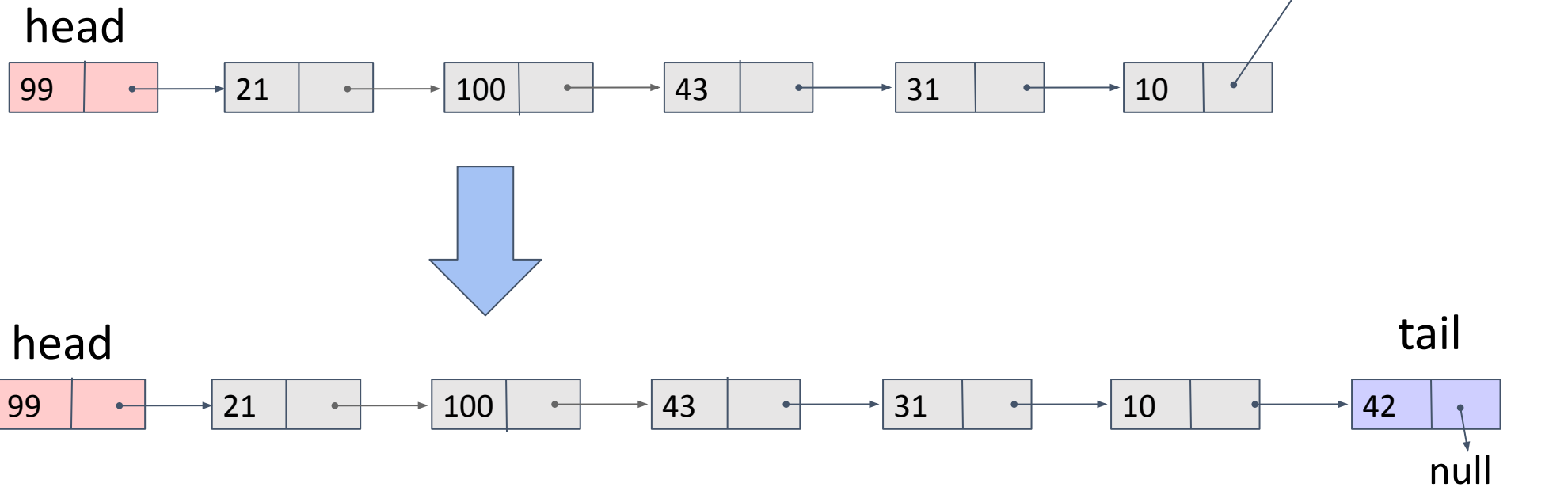
```

Node newNode = new Node( elem: 200);
tail.next = newNode;
tail = newNode;
  
```



Optimizing Tail Insertion

- Append 42 (Insert at the end)



```
Node newNode = new Node( elem: 200);
tail.next = newNode;
tail = newNode;
```

$O(1)$



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

- Remove head
- Remove tail
- Remove from anywhere
- Update element value at index n

