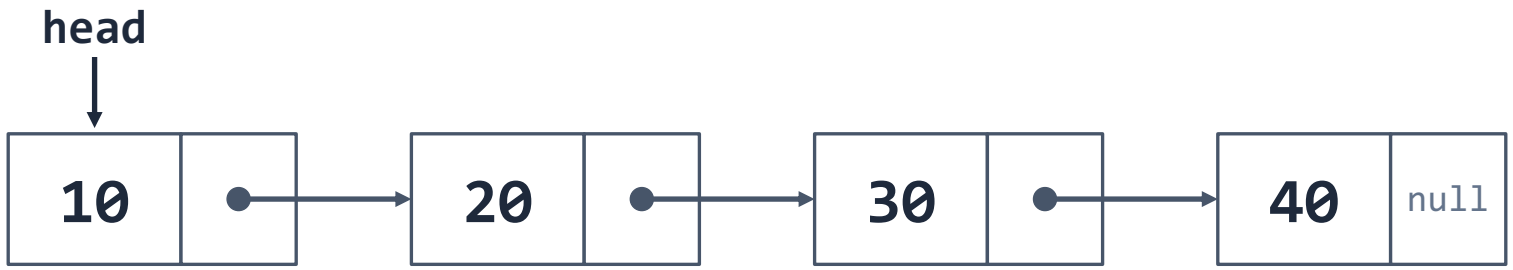
Linked Lists

Nodes, the LinkedList class, and walking through a list

Data Structures · Linked Lists, 1 of 2

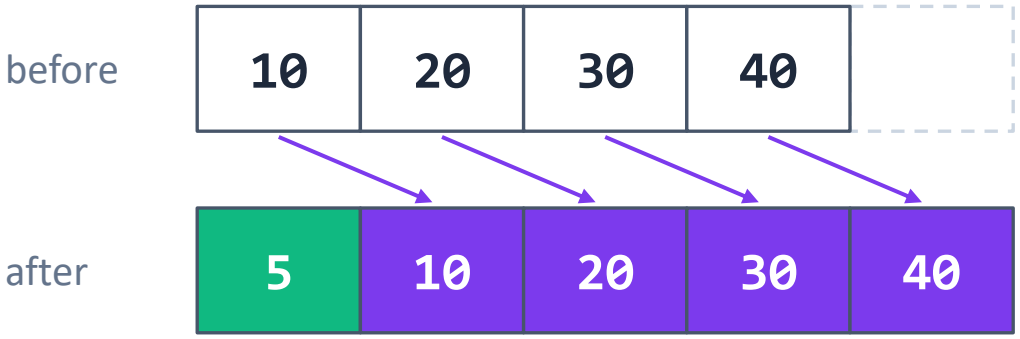
Dr. Muhammad Alsharif

Partially based on slides by Dr. Jonathan Cazalas and Dr. Abdullah Alharbi



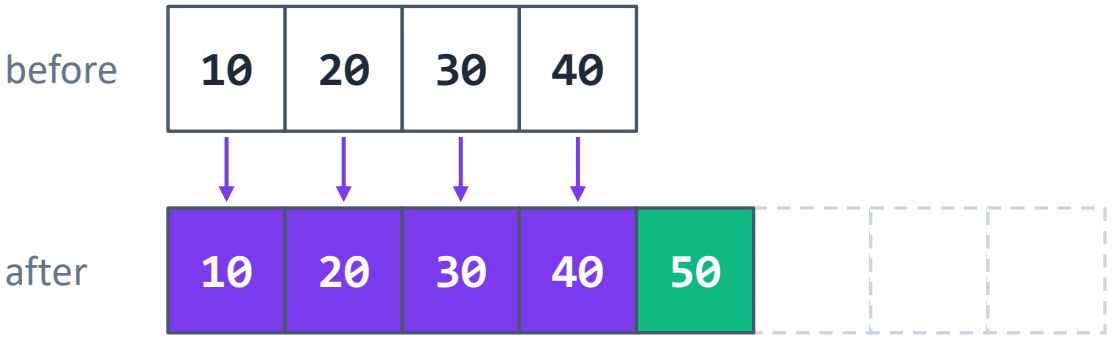
ArrayList can grow, but it moves elements

`list.add(0, 5)`



Every element moves one place: O(n).

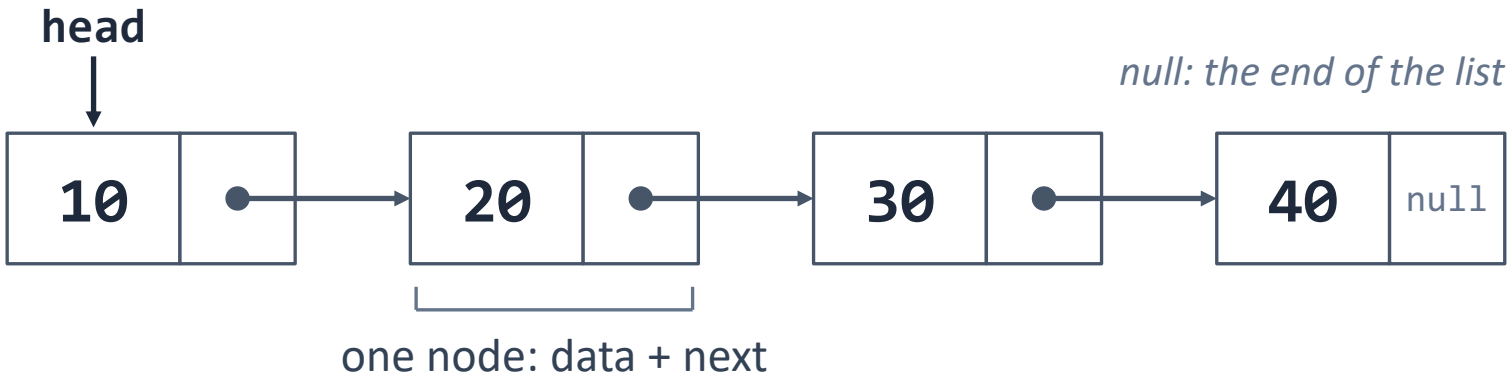
`list.add(50)` when the array is full



Every element is copied into a bigger array. This copy is O(n), but it is rare: the new array has room for many more adds. Averaged over many adds, `add(value)` is still O(1).

- A linked list **never shifts or copies** its elements.
- To add a value, it creates one new node and changes a couple of references.

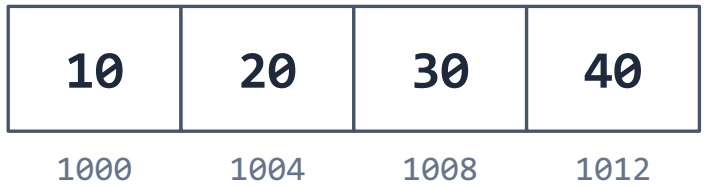
A linked list is a chain of nodes



- Each **node** holds a value (the **data**) and a reference to the **next** node.
- A reference called **head** points to the first node. The last node's `next` is `null`.
- The only way into the list is through `head`: from there, follow the links one node at a time.

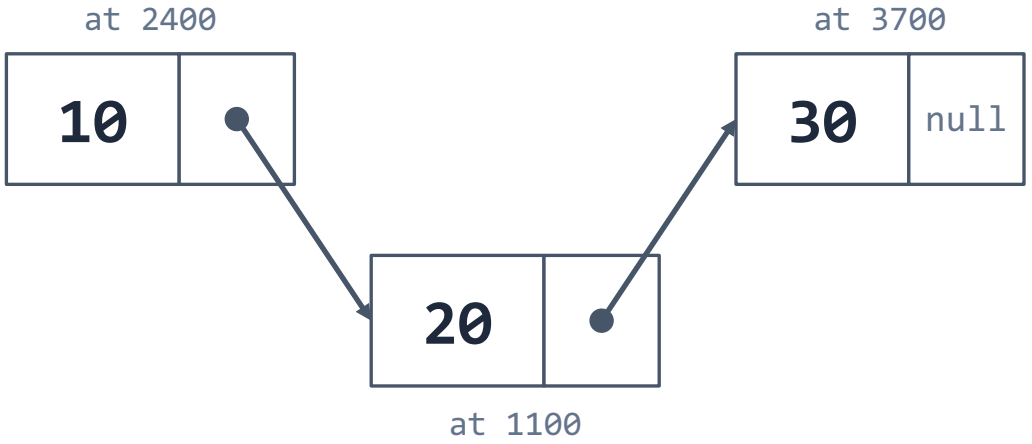
In memory: one block vs scattered nodes

Array



The elements sit side by side: they are **contiguous**. The address of `a[i]` is computed directly, so reading any element is O(1). This is called **direct access**.

Linked list



Each node is placed wherever free memory was found. Only the node before it knows where it is, so reaching the node at index i means walking i steps from `head`: O(n).

Nodes are created one at a time as values are added, and released as they are removed. That is why a linked list is called a **dynamic** structure: it never has unused slots.

Part 1

Nodes

The building block of a linked list

A node

```
public class Node {
    int data;    // the value
    Node next;   // the next node, or null

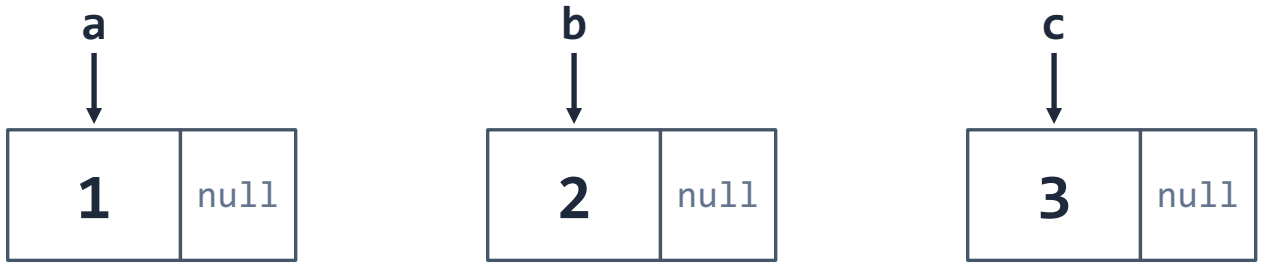
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```



- `next` has type `Node`: it refers to another node.
- `null` means there is no next node.
- The data can be of any type, even an object with several fields. Here it is an `int`.

Creating and linking nodes


```
Node a = new Node(1);
Node b = new Node(2);
Node c = new Node(3);
a.next = b;
b.next = c;
```

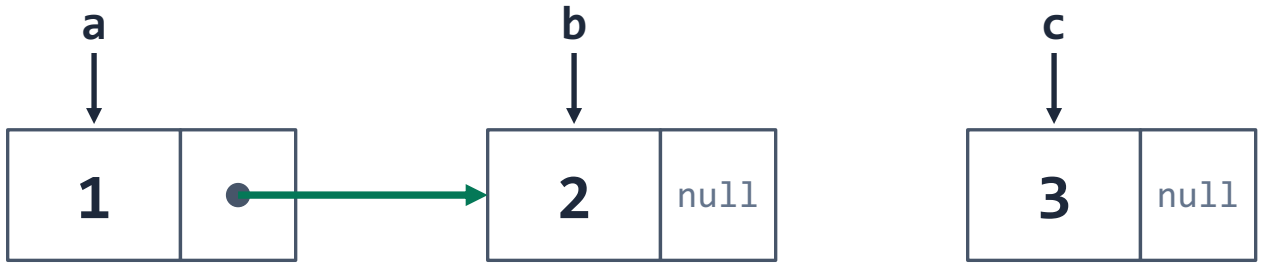


Each `new Node(...)` creates a node whose `next` is `null`.

The variables a, b and c refer to three separate nodes. Nothing is linked yet.

Creating and linking nodes


```
Node a = new Node(1);
Node b = new Node(2);
Node c = new Node(3);
a.next = b;
b.next = c;
```

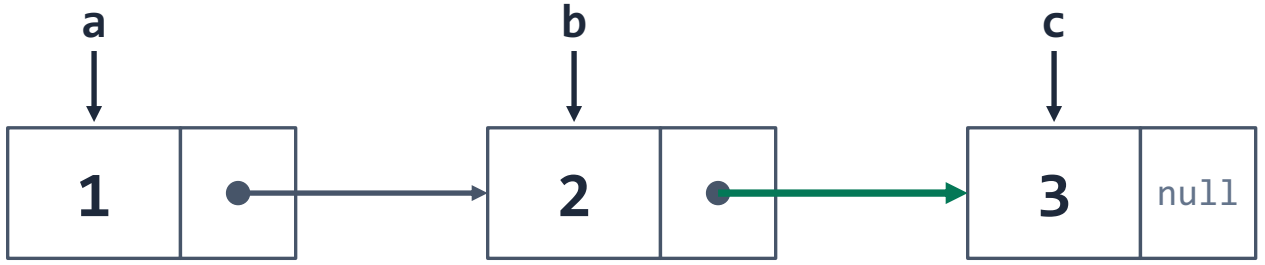


`a.next = b;` makes the `next` of node 1 refer to node 2.

Node 1 is now linked to node 2.

Creating and linking nodes

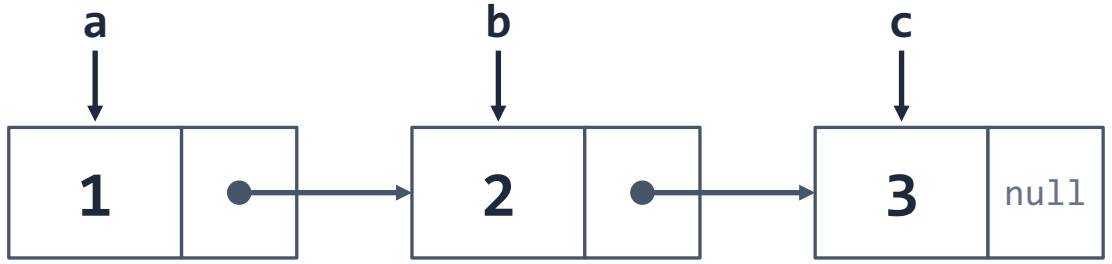

```
Node a = new Node(1);
Node b = new Node(2);
Node c = new Node(3);
a.next = b;
b.next = c;
```



`b.next = c;` links node 2 to node 3.

The nodes now form a chain 1 → 2 → 3. The `next` of node 3 is still `null`: it is the last node.

Following the links




```
System.out.println(a.data);             // 1
System.out.println(a.next.data);        // 2
System.out.println(a.next.next.data);   // 3
System.out.println(b.next.data);        // 3
System.out.println(c.next.data);        // error
```

- Each `.next` moves one node forward.
- Each node knows only the next node. There is no link back to the previous one.
- `c.next` is `null`, so `c.next.data` throws a **NullPointerException**.

Part 2

The LinkedList class

head, size, and walking through the list

Moving `Node` inside `LinkedList`

Part 1: a separate class

```
public class Node {
    int data;
    Node next;
    Node(int data) { ... }
}
```

Now: inside the list, and private

```
public class LinkedList {
    private class Node {
        int data;
        Node next;
        Node(int data) { ... }
    }
    private Node head;
    ...
}
```

- A class can be declared inside another class.
- `private` means only `LinkedList` can use `Node`, just like a private field.
- The methods of `LinkedList` can read and change a node's `data` and `next` directly.

Why: users of the list call methods such as `add` and `get`. They never see nodes; only the list's own methods do.

The LinkedList class

```
public class LinkedList {
    private class Node {   // inner class
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

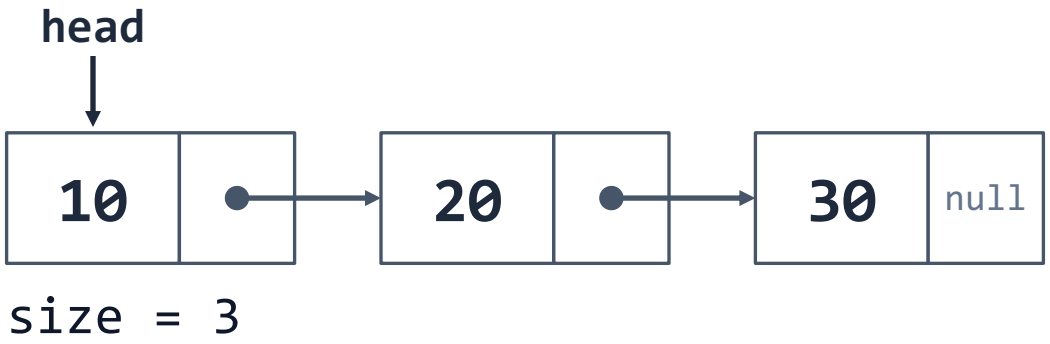
    private Node head;   // the first node
    private int size;    // the number of nodes

    public LinkedList() {
        head = null;
        size = 0;
    }
}
```

An empty list



A list with three nodes



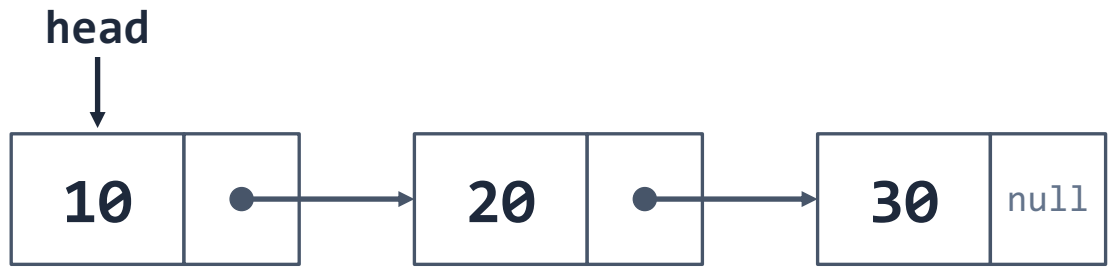
- The list is empty exactly when `head == null`.
- `Node` is private: only the list uses it.

Walking through the list

```
public void printAll() {
    Node current = head;           // start at the first node
    while (current != null) {      // stop after the last node
        System.out.print(current.data + " ");
        current = current.next;    // move to the next node
    }
    System.out.println();
}
```

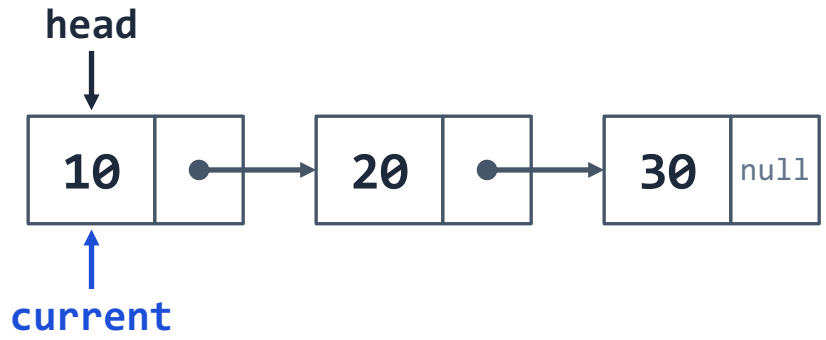
Output

```
10 20 30
```



printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```

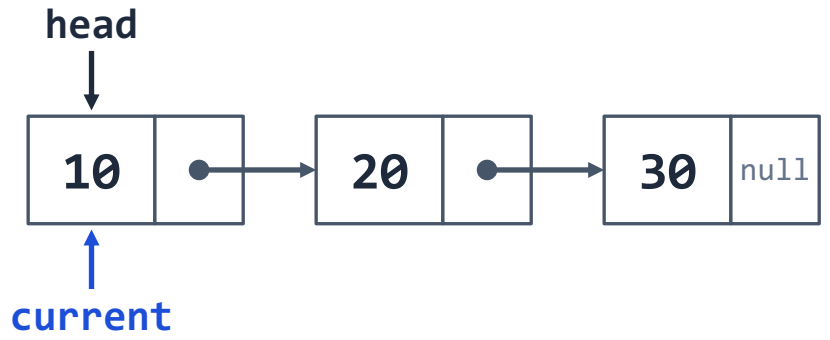


Output

`current` starts at the first node: the same node `head` refers to.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```

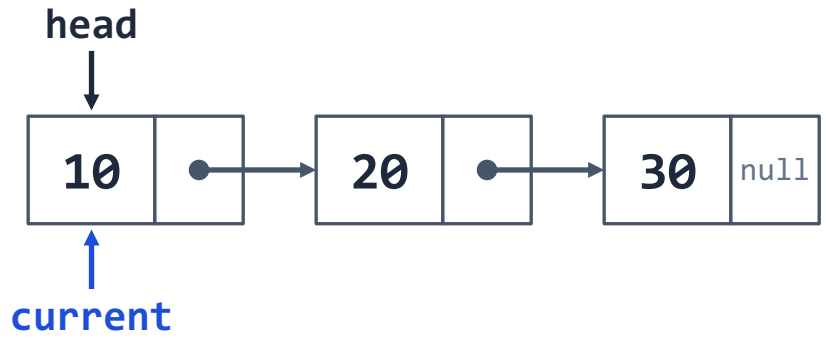


Output

Check: `current != null` is true, so the loop body runs.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



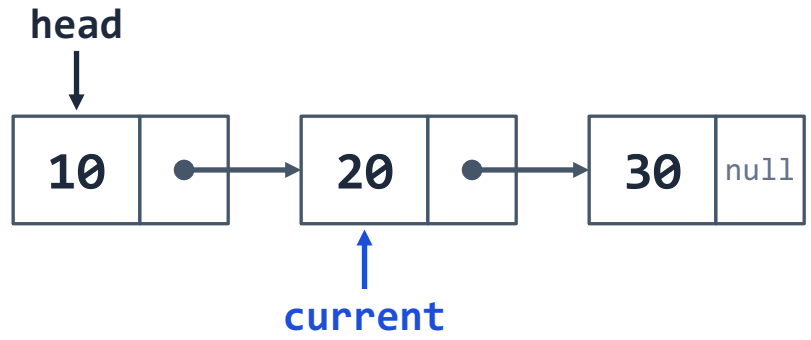
Output

```
10
```

It prints `current.data`, which is 10.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



Output

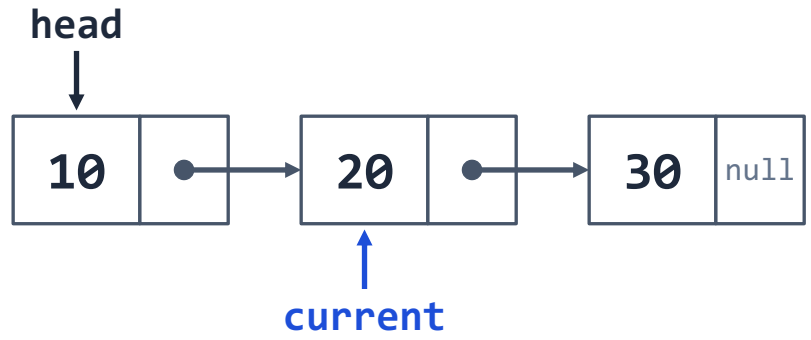
```
10
```

`current = current.next;` moves `current` to the next node, 20.

This compiles because `next` has type `Node`, the same type as `current`.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



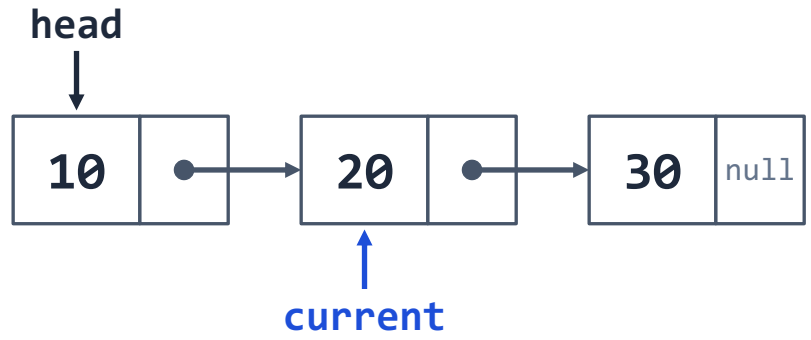
Output

```
10
```

Back to the check: `current` is not `null`, so the loop body runs again.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



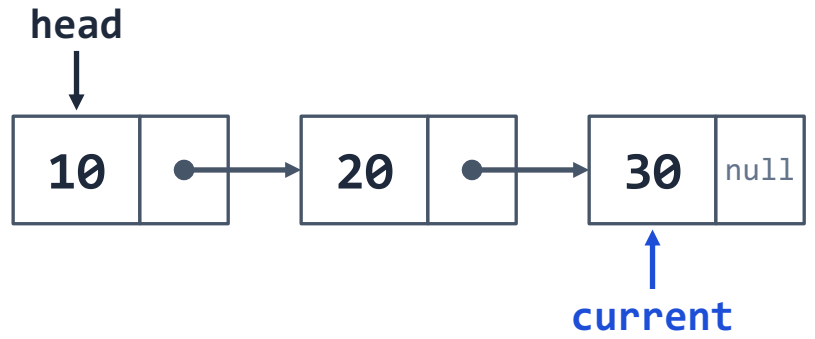
Output

```
10 20
```

It prints 20.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



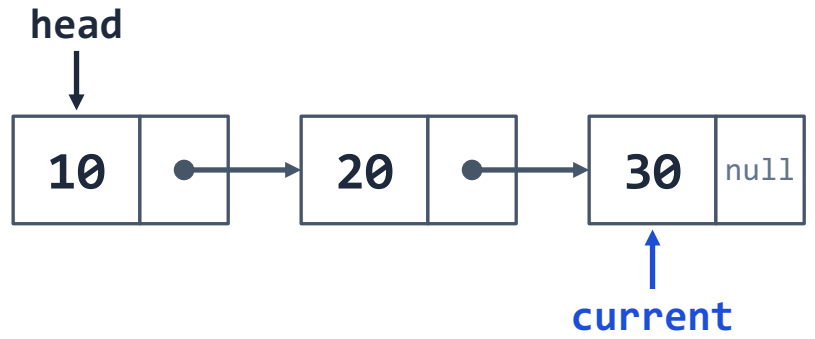
Output

```
10 20
```

`current` moves to the next node, 30.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



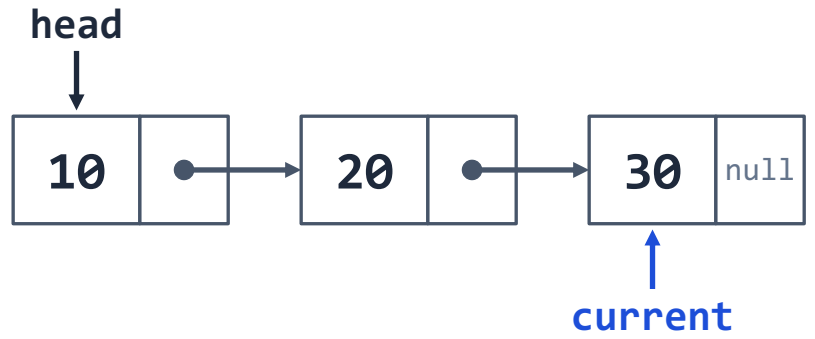
Output

```
10 20
```

Check: `current` is not `null`, so the loop body runs again.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



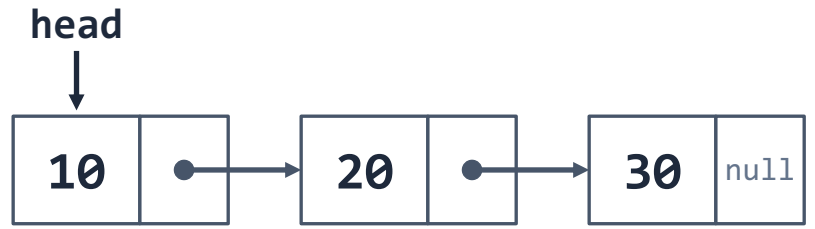
Output

```
10 20 30
```

It prints 30.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



current = null

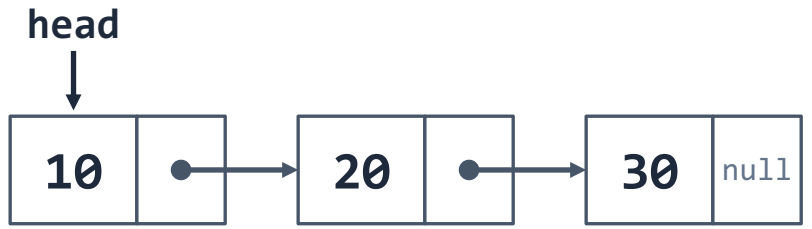
Output

```
10 20 30
```

The `next` of the last node is `null`, so `current` becomes `null`.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



current = null

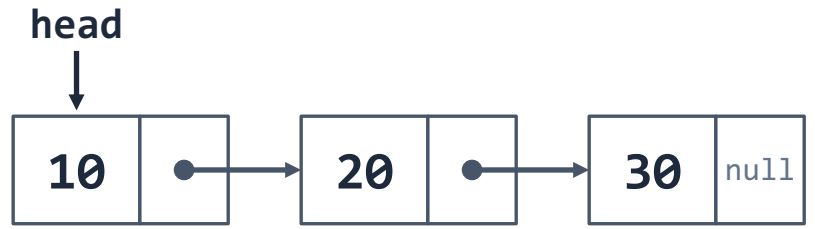
Output

```
10 20 30
```

Check: `current != null` is now false, so the loop stops.

printAll()


```
public void printAll() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```



current = null

Output

```
10 20 30
```

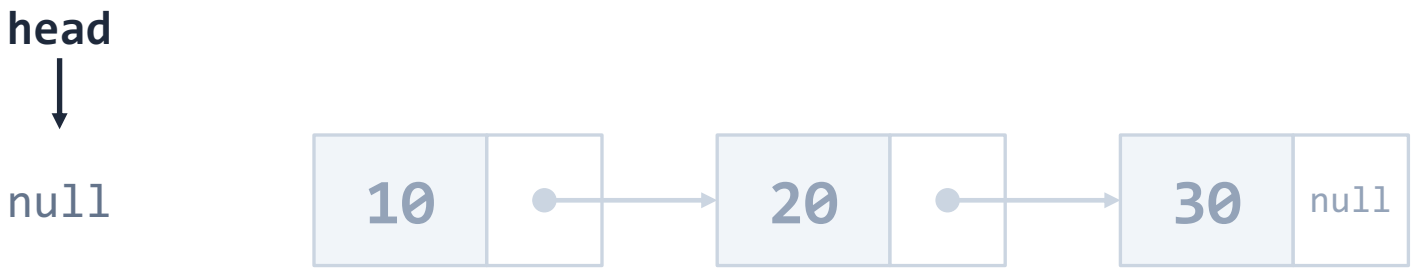
`println()` ends the line.

`head` never moved, so the list is unchanged.
Visiting every node once is O(n).

Never move head

```
// wrong: head itself is used to walk
while (head != null) {
    System.out.print(head.data + " ");
    head = head.next;
}
```

After the loop

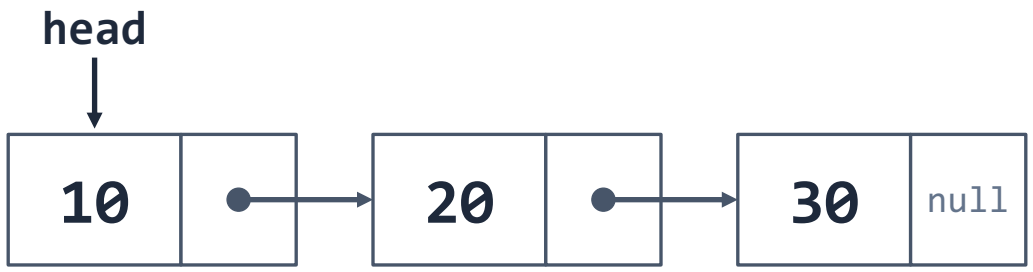


- It prints 10 20 30, but now `head` is `null`, and nothing leads back to the nodes: **the list is lost**.
- Always walk with a separate reference, `current`.

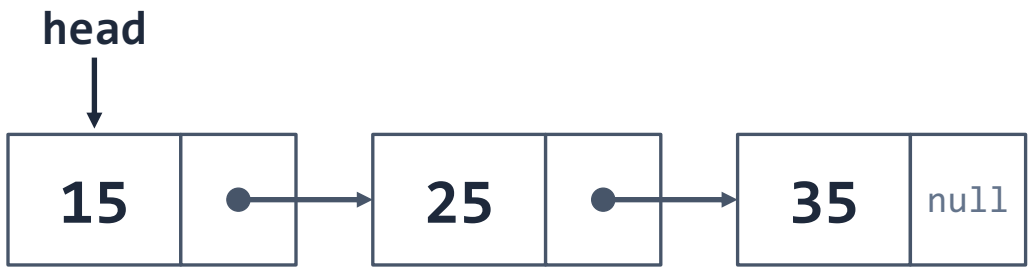
Changing values while walking

```
public void addToAll(int amount) {
    Node current = head;
    while (current != null) {
        current.data += amount;   // change this node
        current = current.next;
    }
}
```

Before



After addToAll(5)



The methods we will build next

Group	Method	What it does
Reading	`get(index)`	returns the value at index
	`indexOf(value)`	returns the index of value, or -1 if it is not there
	`size()`	returns the number of values
	`isEmpty()`	returns true if the list has no nodes
Adding	`addFirst(value)`	adds value at the front
	`addLast(value)`	adds value at the end
	`add(index, value)`	adds value at position index
Removing	`removeFirst()`	removes and returns the first value
	`remove(index)`	removes and returns the value at index
	`removeValue(value)`	removes the first node holding value

Indexes start at 0, as in arrays and ArrayList. An invalid index throws an `IndexOutOfBoundsException`.

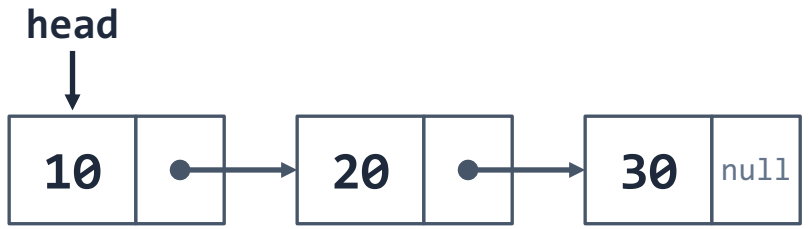
Part 3

Reading values

get and indexOf: walking to a position or a value

get(2)


```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

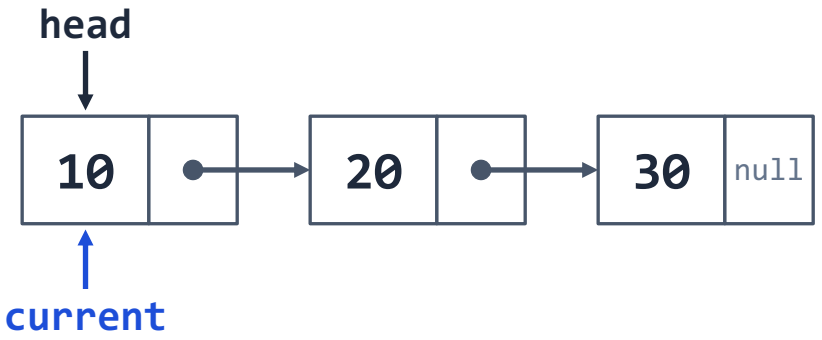


Goal: return the value at index 2 of the list 10 20 30.

Index 2 is valid (0 ≤ 2 < size 3), so no exception is thrown.

get(2)

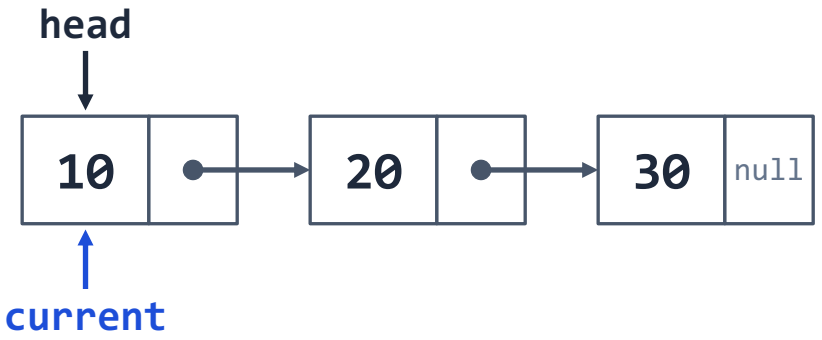

```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```



`current` starts at the first node, index 0.

get(2)

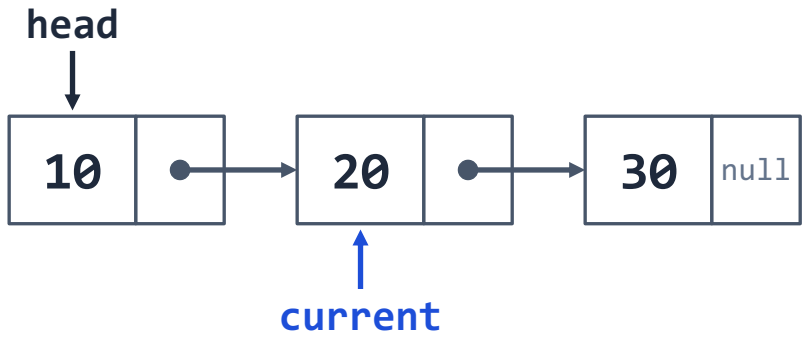

```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```



The loop starts with `i = 0`. Check: 0 < 2 is true, so the body runs.

get(2)

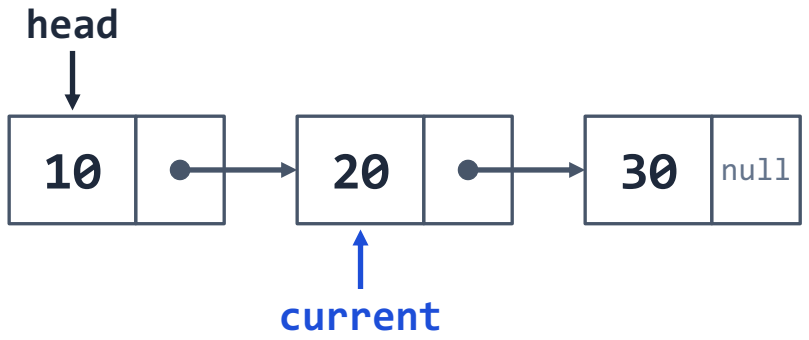

```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```



`current` moves to index 1.

get(2)

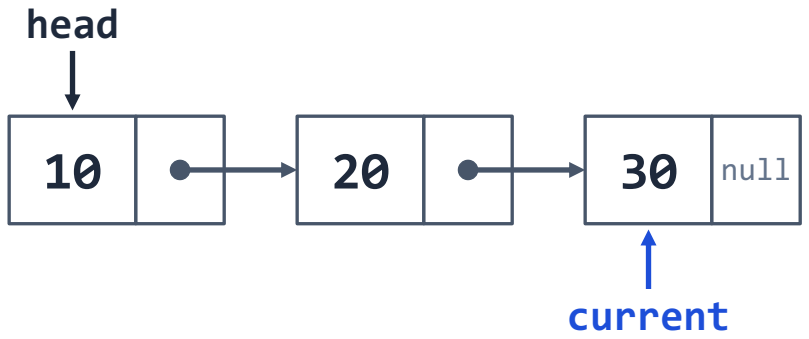

```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```



`i++` makes `i = 1`. Check: 1 < 2 is true, so the body runs again.

get(2)

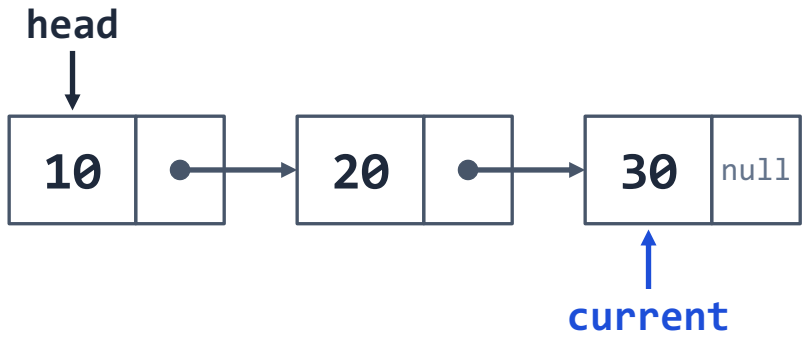

```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```



`current` moves to index 2.

get(2)

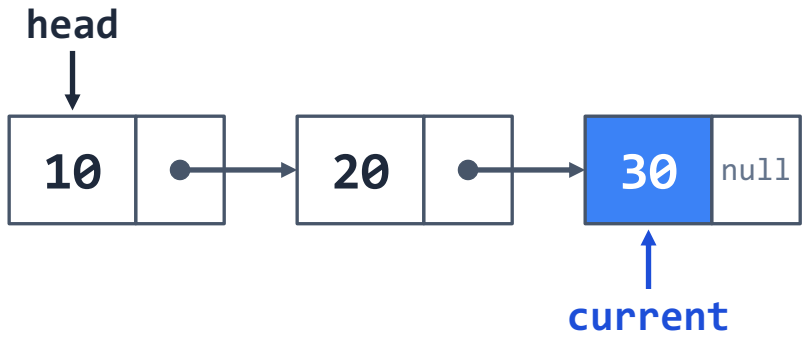

```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```



`i++` makes `i = 2`. Check: 2 < 2 is false, so the loop stops.

get(2)


```
public int get(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    Node current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

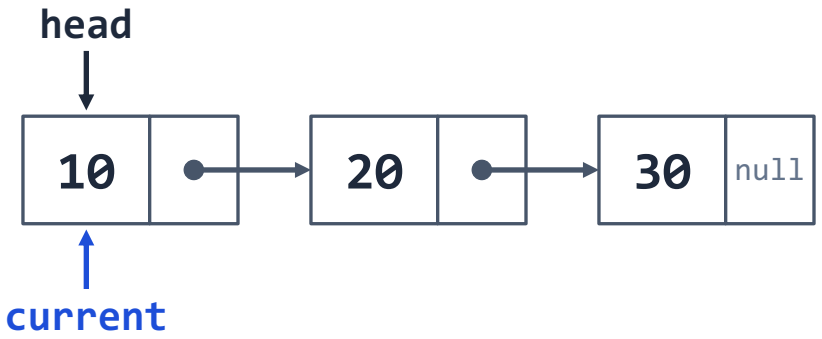


The method returns `current.data`, which is 30.

The loop runs `index` times, so `get` is **O(n)**. An array or ArrayList reads `a[i]` directly in O(1).

indexOf(30)


```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```

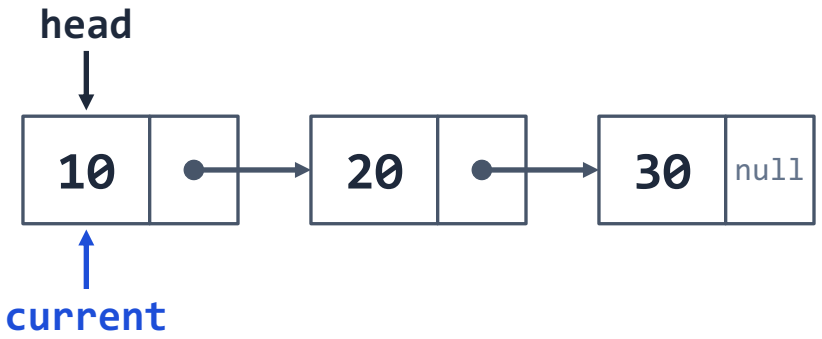


Goal: find the index of 30 in the list 10 20 30.

`current` starts at the first node.

indexOf(30)

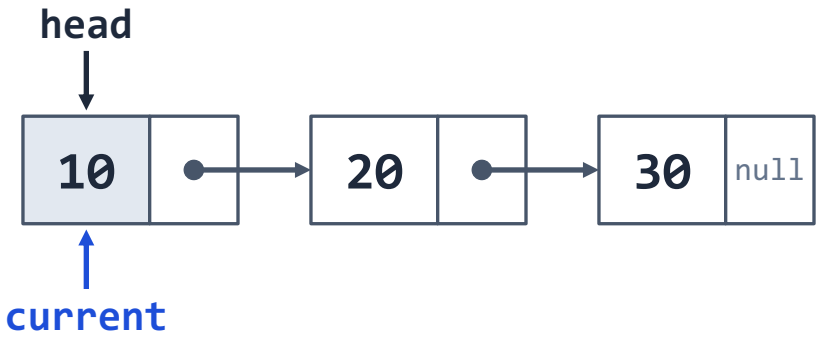

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



The loop starts with `i = 0`. Check: 0 < 3 is true, so the body runs.

indexOf(30)

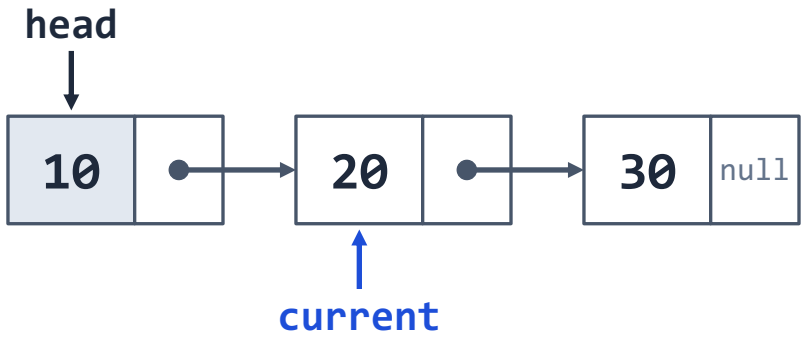

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`current.data` is 10, not 30.

indexOf(30)

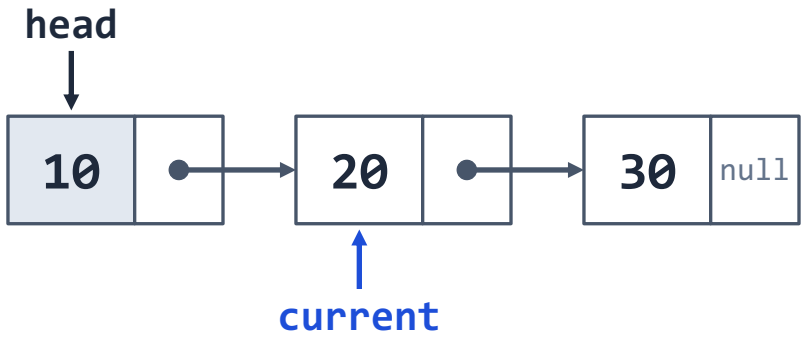

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`current` moves to the next node.

indexOf(30)

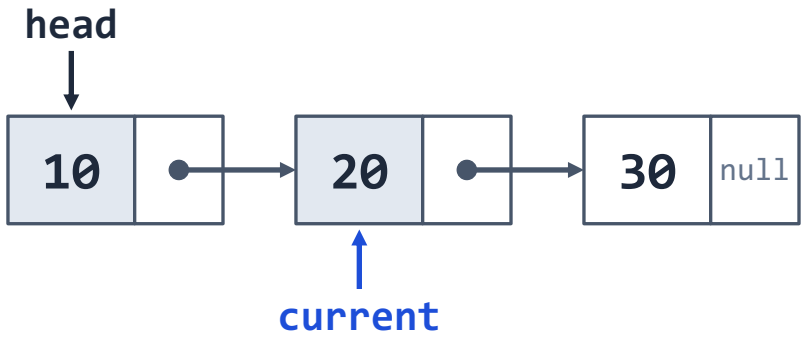

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`i++` makes `i = 1`. Check: 1 < 3 is true, so the body runs again.

indexOf(30)

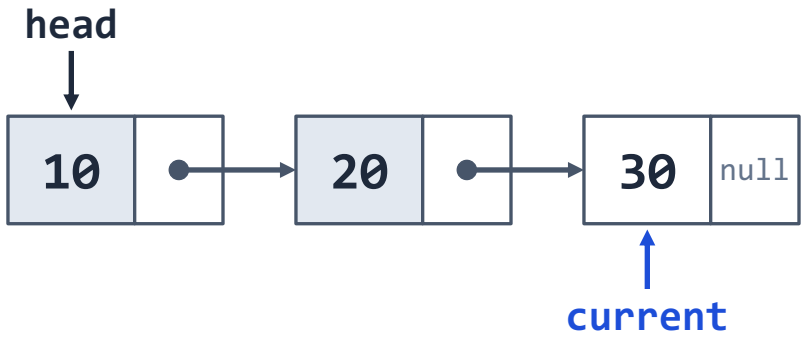

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`current.data` is 20, not 30.

indexOf(30)

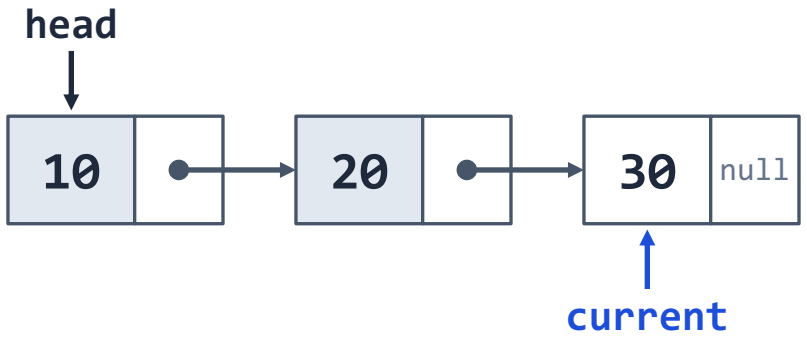

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`current` moves to the next node.

indexOf(30)

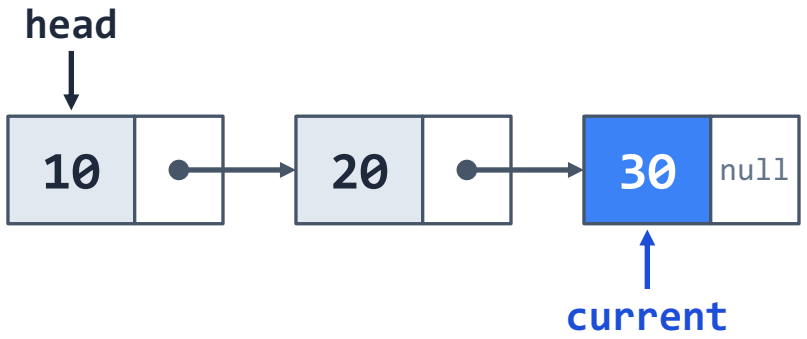

```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`i++` makes `i = 2`. Check: 2 < 3 is true, so the body runs again.

indexOf(30)


```
public int indexOf(int value) {
    Node current = head;
    for (int i = 0; i < size; i++) {
        if (current.data == value) {
            return i;
        }
        current = current.next;
    }
    return -1;
}
```



`current.data` is 30: found. The method returns 2.

If 30 were not in the list, the loop would stop after `size` checks and the method would return -1. Cost: **O(n)**.

Summary

- A linked list is a chain of **nodes**. Each node holds a value and a reference to the `next` node; the last node's `next` is `null`.
- `head` refers to the first node. The list is empty when `head == null`.
- Nodes are scattered in memory, so there is no direct access: reaching index i means walking i steps from `head`.
- `Node` is a private class inside `LinkedList`: only the list's methods use nodes.
- Walk with `current`, never with `head`. `while (current != null)` visits every node.
- `get` and `indexOf` walk the list: O(n).

Next: adding and removing nodes (Linked Lists, 2 of 2).