Chapter 5: Linked Lists

Data Structures

Nodes and links · Singly linked list · Insert · delete · traverse · Reversal · Doubly linked list · Circular list · Sentinel nodes

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/data_structures/slides/Chapter_05_Linked_Lists.html

INTRODUCTION

Learning Outcomes

Node structure

Explain nodes, links, and references.

Core operations

Implement insertion, deletion, search, and traversal.

Kinds of lists

Compare singly, doubly, and circularly linked lists.

Cost analysis

Evaluate the time and space costs of linked-list operations.

BASICS

Kinds of Linked Lists

Singly linked

Each node points only to the next one; the last node points to `null`.



Doubly linked

Each node points to both the previous and the next node, so we can walk either way.



Circularly linked

The last node points back to the first instead of `null`, closing the list into a ring.

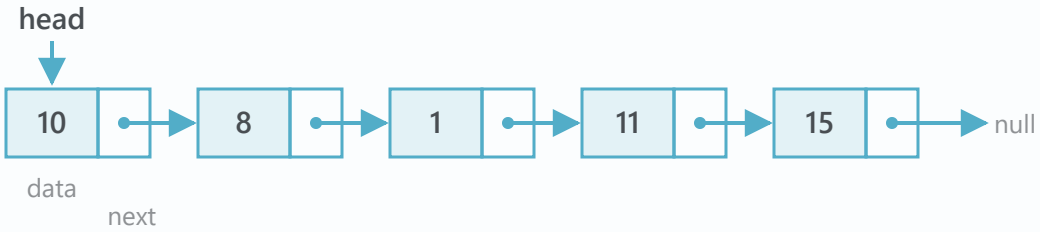


In all three, items live not in adjacent cells, as in an array, but in **nodes** that point to one another.

BASICS

Singly Linked List

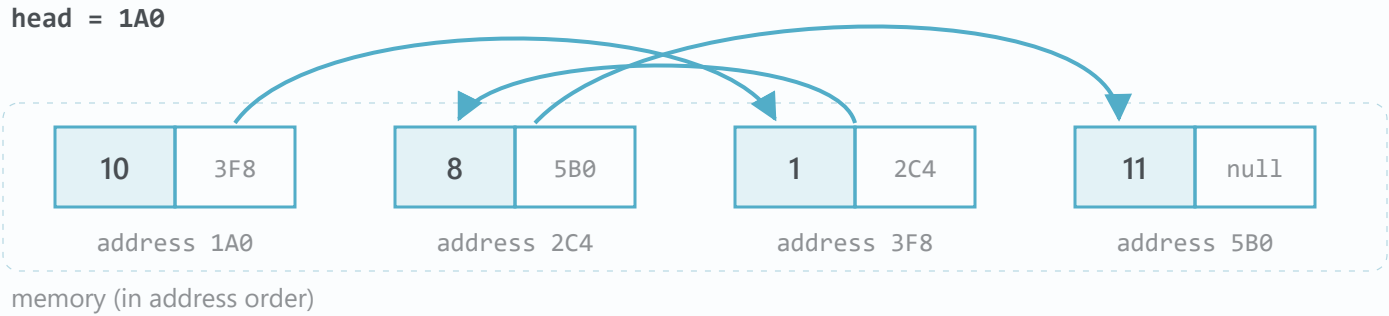
- Items are connected by **one-way** links: each item holds a reference to the one after it.
- The list keeps a special reference to the first item, called the **head** (`head` in code).
- Each node has two parts:
  - **Data**: the value the node carries (a primitive or an object).
  - **Link (pointer)**: a reference to the next node; `null` in the last node.



BASICS

Where Do Nodes Live in Memory?

- Items are **not stored contiguously**; each node is created separately with `new`.
- Still, the list is a **linear** structure: every node except the last has exactly one successor.
- The order comes from the references in the **next** fields, not from memory addresses.



memory (in address order)

The list order is 10 → 1 → 8 → 11, but the nodes sit in memory in a different order; the arrows follow the references.

BASICS

The Node Class

int data

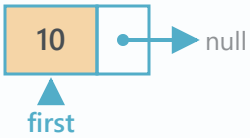
```
// Node class
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

Generic version

```
// Node class
class Node<T> {
    T data;
    Node<T> next;
    Node(T data) {
        this.data = data;
        this.next = null;
    }
}
```

- The class has a field of its own type (`Node next`): a **self-referential** structure.
- The generic version stores any type: `Node<String> d = new Node<>("Ann");`
- A new node's `next` field starts as `null` and changes once the node is linked into a list.

Building a Singly Linked List



`new Node(10)` : creates a node with data = 10 and next = null; `first` points to it.

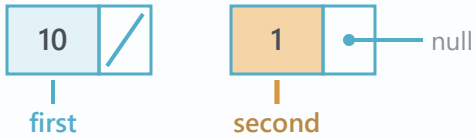
Step 1 / 7

```
▶ Node first = new Node(10);
2 Node second = new Node(1);
3 Node third = new Node(8);
4 Node fourth = new Node(11);
5 first.next = second;
6 second.next = third;
7 third.next = fourth;
```

VARIABLES

`first = node(10)`

Building a Singly Linked List



The second node (1) is created. The nodes are not linked yet.


```
1 Node first = new Node(10);
▶ Node second = new Node(1);
3 Node third = new Node(8);
4 Node fourth = new Node(11);
5 first.next = second;
6 second.next = third;
7 third.next = fourth;
```

VARIABLES

`first = node(10)`

`second = node(1)`

Building a Singly Linked List



The third node (8) is created. Each node can sit anywhere in memory.

Step 3 / 7

```
1 Node first = new Node(10);
2 Node second = new Node(1);
▶ Node third = new Node(8);
4 Node fourth = new Node(11);
5 first.next = second;
6 second.next = third;
7 third.next = fourth;
```

VARIABLES

- `first = node(10)`
- `second = node(1)`
- `third = node(8)`

Building a Singly Linked List



The fourth node (11) is created. We have four separate nodes, each with a `next` field of null (/).

Step 4 / 7

```
1 Node first = new Node(10);
2 Node second = new Node(1);
3 Node third = new Node(8);
▶ Node fourth = new Node(11);
5 first.next = second;
6 second.next = third;
7 third.next = fourth;
```

VARIABLES

- `first = node(10)`
- `second = node(1)`
- `third = node(8)`
- `fourth = node(11)`

Building a Singly Linked List



`first.next = second` : the next field of 10 now points to 1 (orange).

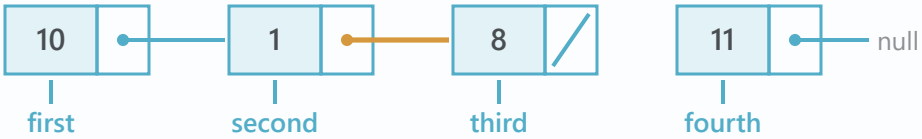
Step 5 / 7

```
1 Node first = new Node(10);
2 Node second = new Node(1);
3 Node third = new Node(8);
4 Node fourth = new Node(11);
▶ first.next = second;
6 second.next = third;
7 third.next = fourth;
```

VARIABLES

- `first = node(10)`
- `second = node(1)`
- `third = node(8)`
- `fourth = node(11)`

Building a Singly Linked List



`second.next = third` : links 1 → 8.

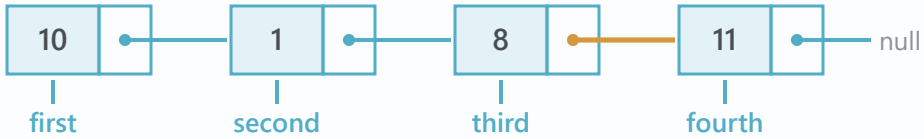
Step 6 / 7

```
1 Node first = new Node(10);
2 Node second = new Node(1);
3 Node third = new Node(8);
4 Node fourth = new Node(11);
5 first.next = second;
▶ second.next = third;
7 third.next = fourth;
```

VARIABLES

- `first = node(10)`
- `second = node(1)`
- `third = node(8)`
- `fourth = node(11)`

Building a Singly Linked List



`third.next = fourth` : the list is ready: 10 → 1 → 8 → 11 → null. `first` alone is enough to reach the whole list.

Step 7 / 7

```
1 Node first = new Node(10);
2 Node second = new Node(1);
3 Node third = new Node(8);
4 Node fourth = new Node(11);
5 first.next = second;
6 second.next = third;
▶ third.next = fourth;
```

VARIABLES

- `first = node(10)`
- `second = node(1)`
- `third = node(8)`
- `fourth = node(11)`

BASICS

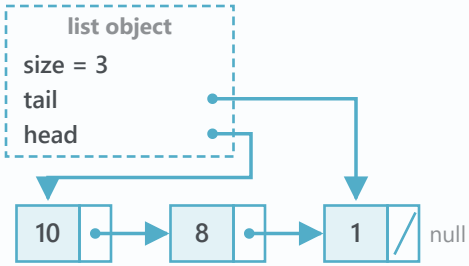
Singly Linked List Operations

Operation	How it works	Cost
Insert at head	New node points to the old head; head becomes the new node	O(1)
Insert at tail	Walk to the last node (no walk with a tail pointer)	O(n) · O(1)
Delete at head	Move head to the next node	O(1)
Delete at tail	Must find the second-to-last node	O(n)
Search · print · length	Walk from head to tail	O(n)
Reverse · merge	One pass over the lists (merge needs a tail pointer)	O(n)

n: number of nodes. Reaching the i-th item is also O(n): unlike an array, we cannot jump to an index.

BASICS

The Singly Linked List ADT and Its Java Class




```java
public class SinglyLinkedList {
    private Node head = null, tail = null;
    private int size = 0;    // number of items
    public int size() { return size; }
    public boolean isEmpty() { return size == 0; }
    public Integer first() {
        return isEmpty() ? null : head.data; }
      // addFirst, addLast, removeFirst …
}
```

Goodrich–Tamassia	In these slides	Purpose	Cost
`size()`, `isEmpty()`	`size()`, `isEmpty()`	the `size` field is updated by every operation	O(1)
`first()`, `last()`	`first()`, `isEmpty() ? null : tail.data`	return `null` on an empty list	O(1)
`addFirst`, `addLast`	`addFirst`, `addLast`	both constant time with a tail pointer; inserting into an empty list also sets `tail`	O(1)
`removeFirst()`	`removeFirst()`	returns the removed node	O(1)

BASICS

Singly Linked Lists: Pros and Cons

Advantages

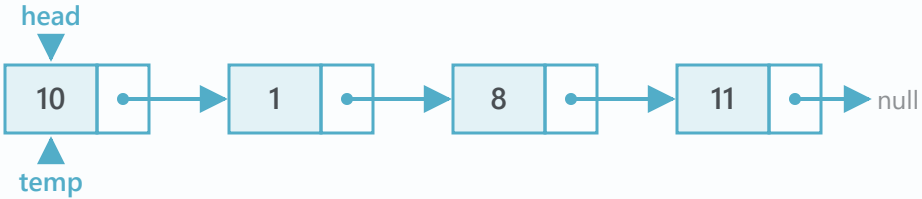
+ Easy to implement.
+ Insert / delete at the head in O(1); inserting in the middle needs no shifting.
+ Dynamic size: no capacity reserved in advance; the item count can change freely.

Disadvantages

– Search is slow: O(n).
– No random access; reaching the i-th item means walking from the head.
– Each node carries an extra reference (more memory).
– Cannot be traversed backward.

Inserting or deleting in the middle is O(1) only **if we already know the position (the previous node)**; finding it still takes an O(n) walk.

Traversing and Printing a List



`temp = head` : the walk starts at the head. `head` never changes; only `temp` moves.

Step 1 / 16

```
1 void printList() {
▶     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

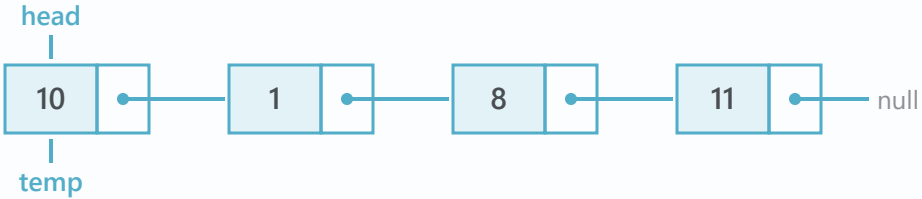
VARIABLES

head = **node(10)**

temp = **node(10)**

OUTPUT

Traversing and Printing a List



Print the heading.

Step 2 / 16

```
1 void printList() {
2     Node temp = head;
▶     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

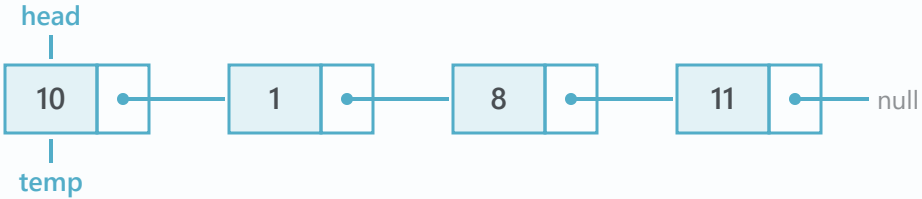
head = **node(10)**

temp = **node(10)**

OUTPUT

```
Linked List:
```

Traversing and Printing a List



`temp != null` (node 10) → enter the loop body.

Step 3 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

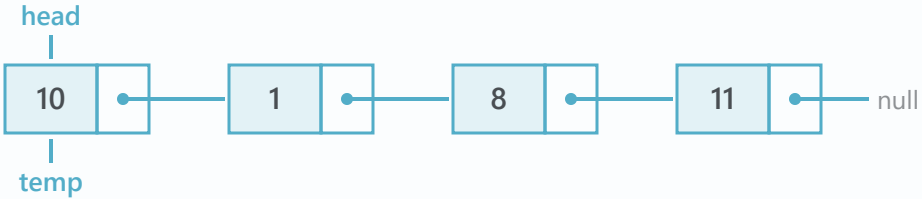
`head = node(10)`

`temp = node(10)`

OUTPUT

```
Linked List:
```

Traversing and Printing a List



Print `temp.data` = 10.

Step 4 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

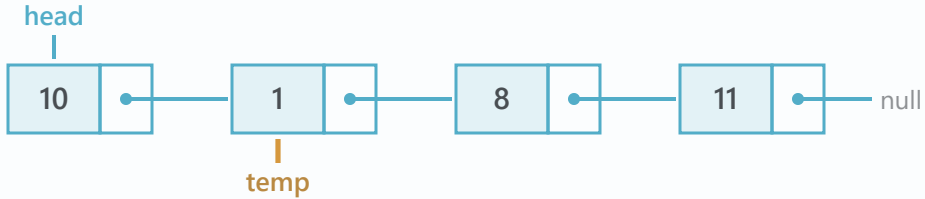
`head = node(10)`

`temp = node(10)`

OUTPUT

```
Linked List: 10 ->
```

Traversing and Printing a List



`temp = temp.next` → temp is now node(1).

Step 5 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
▶         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

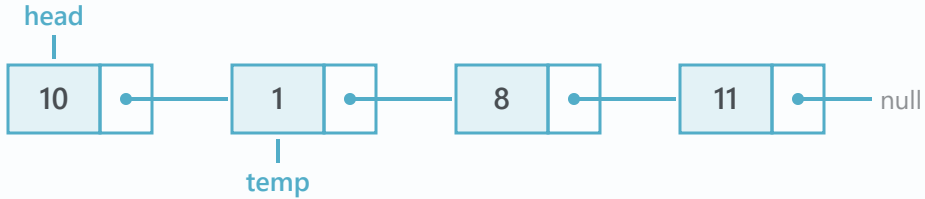
head = **node(10)**

temp = **node(1)**

OUTPUT

```
Linked List: 10 ->
```

Traversing and Printing a List



temp = node(1) ≠ null → continue.


```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

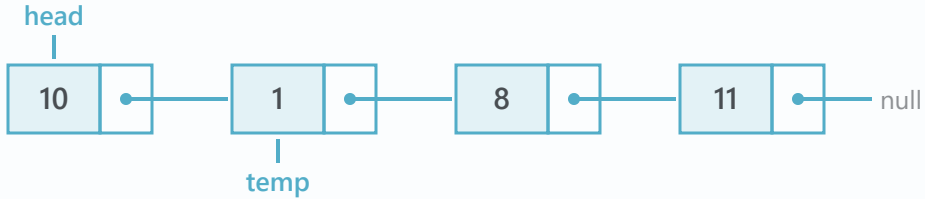
`head = node(10)`

`temp = node(1)`

OUTPUT

```
Linked List: 10 ->
```

Traversing and Printing a List



Print `temp.data` = 1.

Step 7 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

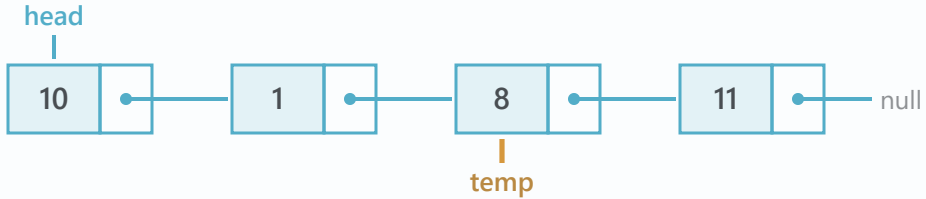
`head = node(10)`

`temp = node(1)`

OUTPUT

```
Linked List: 10 -> 1 ->
```

Traversing and Printing a List



`temp = temp.next` → temp is now node(8).

Step 8 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
▶         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

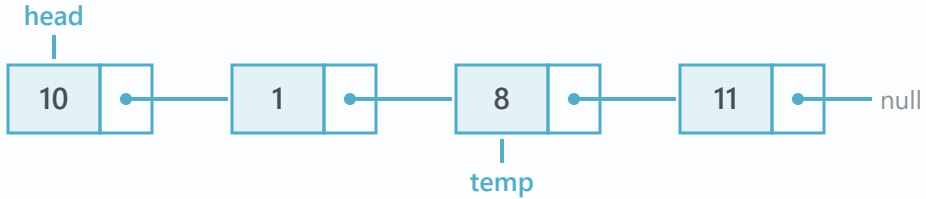
`head = node(10)`

`temp = node(8)`

OUTPUT

```
Linked List: 10 -> 1 ->
```

Traversing and Printing a List



temp = node(8) ≠ null → continue.


```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

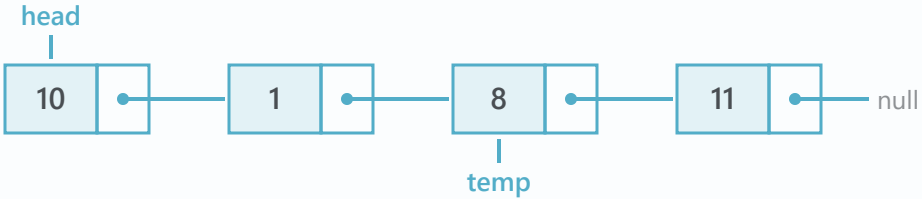
`head = node(10)`

`temp = node(8)`

OUTPUT

```
Linked List: 10 -> 1 ->
```

Traversing and Printing a List



Print `temp.data` = 8.

Step 10 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

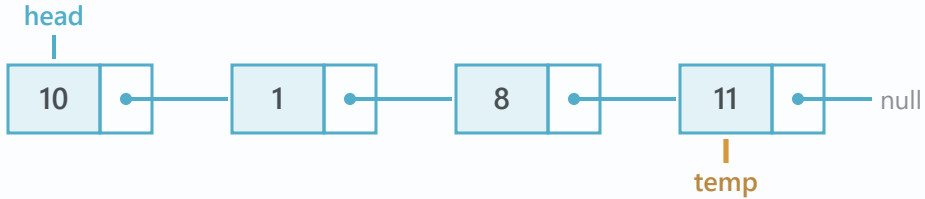
`head = node(10)`

`temp = node(8)`

OUTPUT

```
Linked List: 10 -> 1 -> 8 ->
```

Traversing and Printing a List



`temp = temp.next` → temp is now node(11).

Step 11 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
▶         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

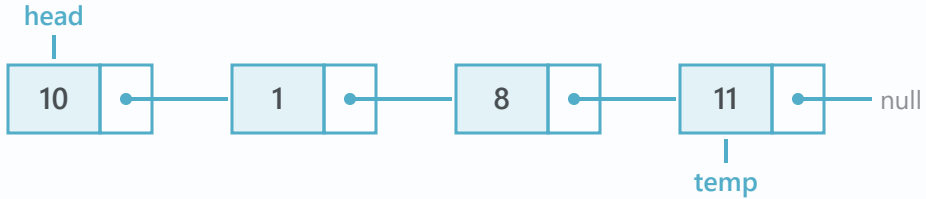
head = **node(10)**

temp = **node(11)**

OUTPUT

```
Linked List: 10 -> 1 -> 8 ->
```

Traversing and Printing a List



temp = node(11) ≠ null → continue.


```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

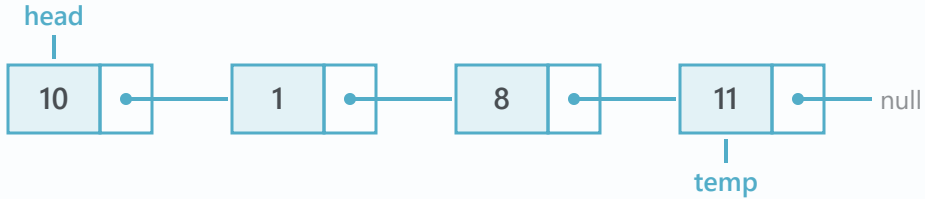
`head = node(10)`

`temp = node(11)`

OUTPUT

```
Linked List: 10 -> 1 -> 8 ->
```

Traversing and Printing a List



Print `temp.data` = 11.

Step 13 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

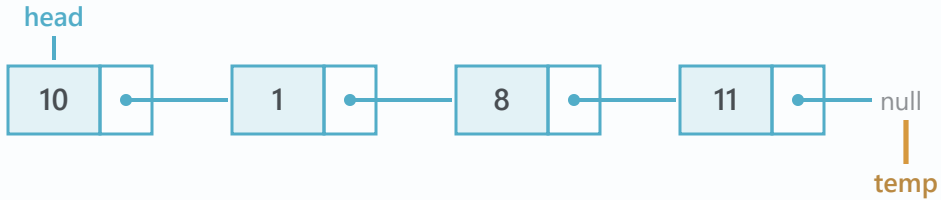
`head = node(10)`

`temp = node(11)`

OUTPUT

```
Linked List: 10 -> 1 -> 8 -> 11
->
```

Traversing and Printing a List



`temp = temp.next` → temp is now **null** (the last node's next field).

Step 14 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
▶         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

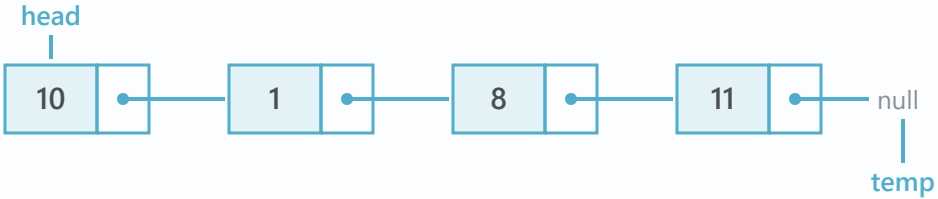
VARIABLES

`head = node(10)` `temp = null`

OUTPUT

```
Linked List: 10 -> 1 -> 8 -> 11
->
```

Traversing and Printing a List



`temp == null` → the condition is false and the loop ends.

Step 15 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
8     System.out.println("null");
9 }
```

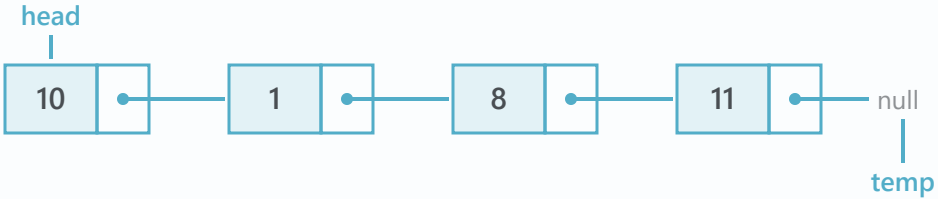
VARIABLES

`head = node(10)` `temp = null`

OUTPUT

```
Linked List: 10 -> 1 -> 8 -> 11
->
```

Traversing and Printing a List



Print `null` at the end. For n nodes the loop runs n times: **O(n)**.

Step 16 / 16

```
1 void printList() {
2     Node temp = head;
3     System.out.print("Linked List: ");
4     while (temp != null) {
5         System.out.print(temp.data + " -> ");
6         temp = temp.next;
7     }
▶     System.out.println("null");
9 }
```

VARIABLES

`head = node(10)` `temp = null`

OUTPUT

```
Linked List: 10 -> 1 -> 8 -> 11
-> null
```

Same Pattern: Length and Search

O(n)

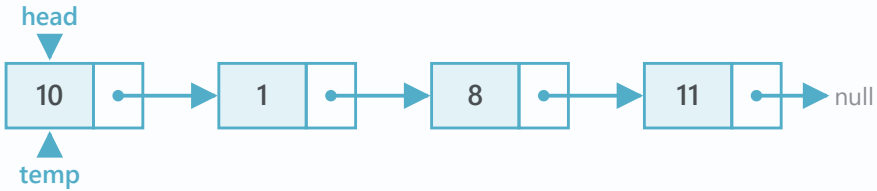
```
int listLength() {
    int length = 0;
    Node temp = head;
    while (temp != null) {
        length++;
        temp = temp.next;
    }
    return length;
}
```

O(n)

```
int search(int target) {
    Node temp = head;
    int position = 0;
    while (temp != null) {
        if (temp.data == target)
            return position;
        position++;
        temp = temp.next;
    }
    return -1;    // not found
}
```

The traversal pattern: start with `temp = head`, do the work while `temp != null`, and advance with `temp = temp.next`. If the length is needed often, keep a `size` field and update it on every insertion and deletion; the length then costs O(1).

Code Trace: `search(8)`



`search(8)` : start at the head with `temp = head` .

Step 1 / 13

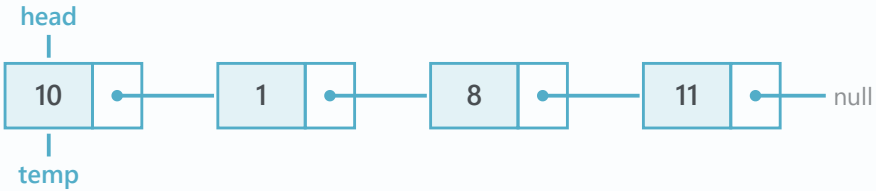
```
1  int search(int target) {
▶      Node temp = head;
3      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(10)`

Code Trace: `search(8)`



`position = 0` : counts temp's place (index) in the list.

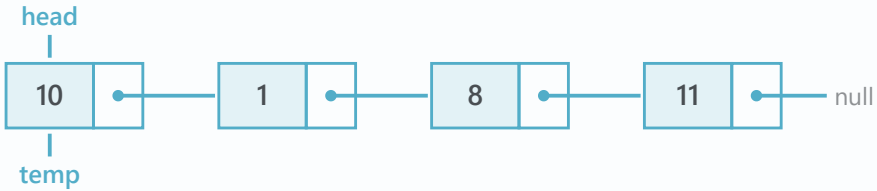
Step 2 / 13

```
1  int search(int target) {
2      Node temp = head;
▶      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

head = **node(10)**	target = **8**
temp = **node(10)**	position = **0**

Code Trace: `search(8)`



temp = node(10) ≠ null → loop.

Step 3 / 13

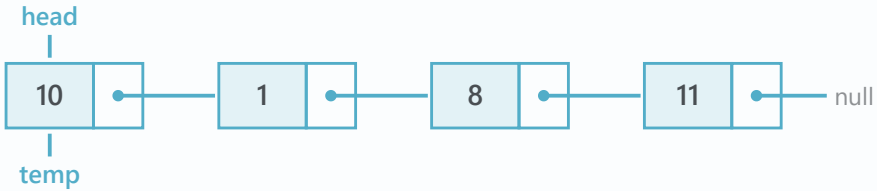
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
▶      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(10)` `position = 0`

Code Trace: `search(8)`



10 == 8? **No** → continue.

Step 4 / 13

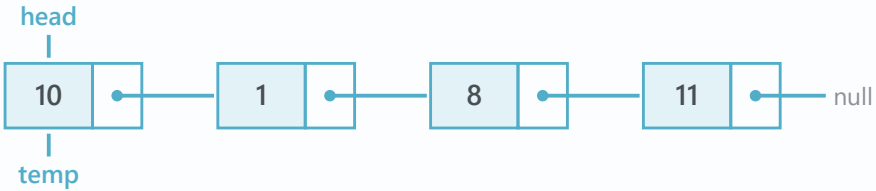
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
▶          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head` = **node(10)** `target` = **8**

`temp` = **node(10)** `position` = **0**

Code Trace: `search(8)`



`position++` → 1.

Step 5 / 13

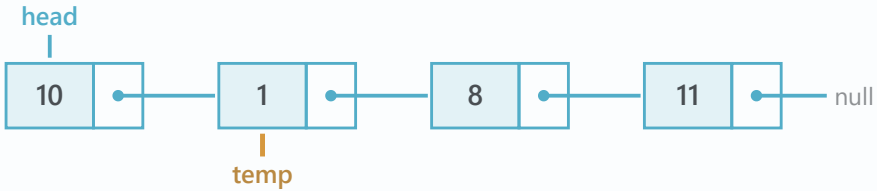
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
6              return position;
▶          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(10)` `position = 1`

Code Trace: `search(8)`



`temp = temp.next` → node(1).

Step 6 / 13

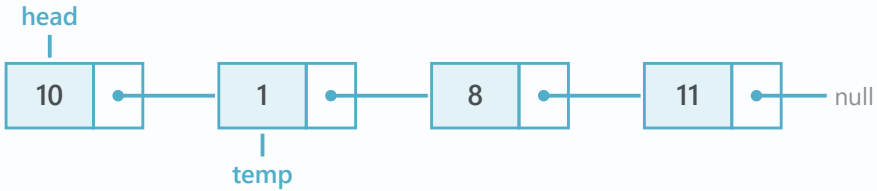
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
▶          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(1)` `position = 1`

Code Trace: `search(8)`



temp = node(1) ≠ null → loop.

Step 7 / 13

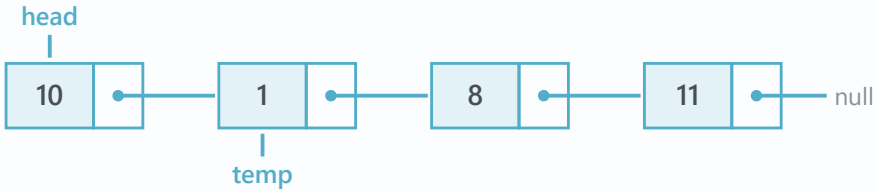
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
▶      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(1)` `position = 1`

Code Trace: `search(8)`



1 == 8? **No** → continue.

Step 8 / 13

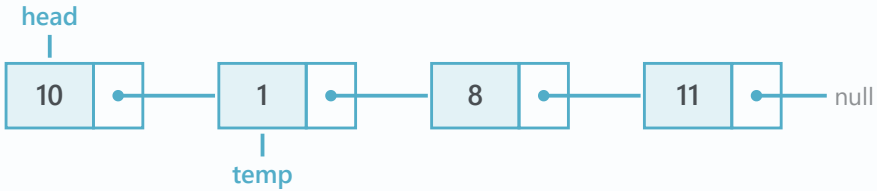
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
▶          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(1)` `position = 1`

Code Trace: `search(8)`



`position++` → 2.

Step 9 / 13

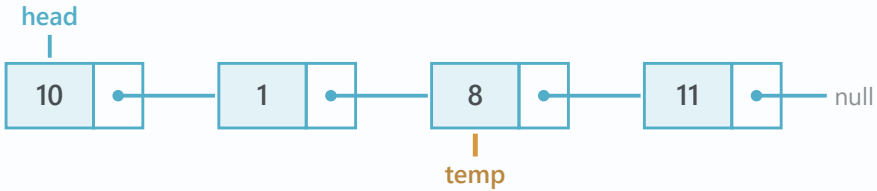
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
6              return position;
▶          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(1)` `position = 2`

Code Trace: `search(8)`



`temp = temp.next` → node(8).

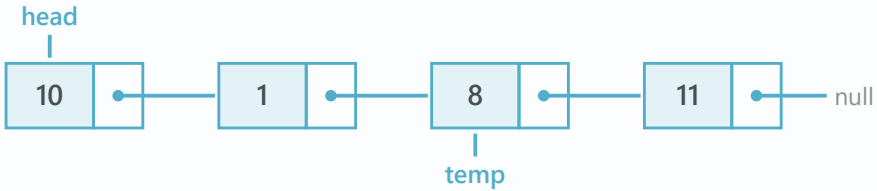
Step 10 / 13

```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
▶          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`
`temp = node(8)` `position = 2`

Code Trace: `search(8)`



temp = node(8) ≠ null → loop.

Step 11 / 13

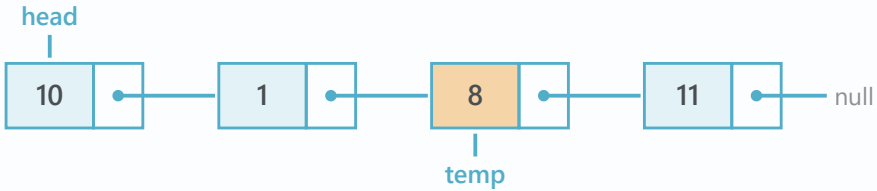
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
▶      while (temp != null) {
5          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(8)` `position = 2`

Code Trace: `search(8)`



8 == 8 → found the target.

Step 12 / 13

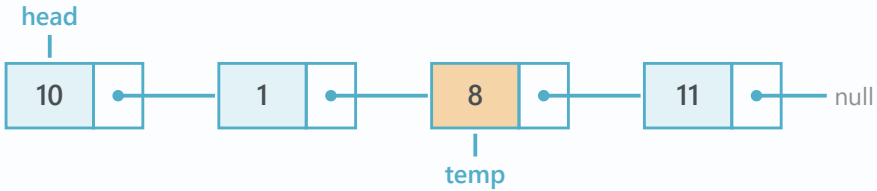
```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
▶          if (temp.data == target)
6              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(8)` `position = 2`

Code Trace: `search(8)`



return 2. We looked at 3 nodes; in the worst case (target absent) we look at all n: **O(n)**, result −1.

Step 13 / 13

```
1  int search(int target) {
2      Node temp = head;
3      int position = 0;
4      while (temp != null) {
5          if (temp.data == target)
▶              return position;
7          position++;
8          temp = temp.next;
9      }
10     return -1;   // not found
11 }
```

VARIABLES

`head = node(10)` `target = 8`

`temp = node(8)` `position = 2`

Inserting at the Head — O(1)



Call `addFirst(11)`. The list is empty: `head = null`.

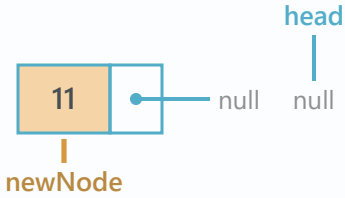
Step 1 / 15

```
▶ void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

`head = null` `tail = null` `size = 0` `data = 11`

Inserting at the Head — O(1)



Create node 11; its next is null.

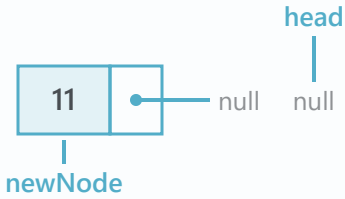
Step 2 / 15

```
1 void addFirst(int data) {
▶     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

head = **null** | tail = **null** | size = **0** | data = **11** | newNode = **node(11)**

Inserting at the Head — O(1)



`newNode.next = head` : head is null, so 11's next stays null.

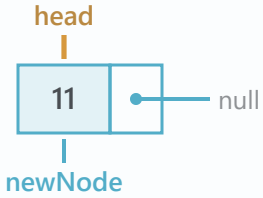
Step 3 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
▶     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

- `head = null`
- `tail = null`
- `size = 0`
- `data = 11`
- `newNode = node(11)`

Inserting at the Head — O(1)



`head = newNode` : head moves to 11: 11 → null.

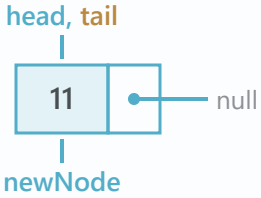
Step 4 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
▶     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

- head = node(11)
- tail = null
- size = 0
- data = 11
- newNode = node(11)

Inserting at the Head — O(1)



`if (tail == null) tail = head` : the list was empty, so 11 is both head and tail. `size++` → 1.

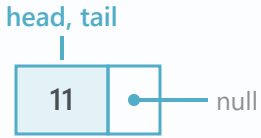
Step 5 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
▶     if (tail == null) tail = head;
▶     size++;
7 }
```

VARIABLES

- `head = node(11)`
- `tail = node(11)`
- `size = 1`
- `data = 11`
- `newNode = node(11)`

Inserting at the Head — O(1)



Call `addFirst(8)`. List: 11 → null.

Step 6 / 15

```
▶ void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

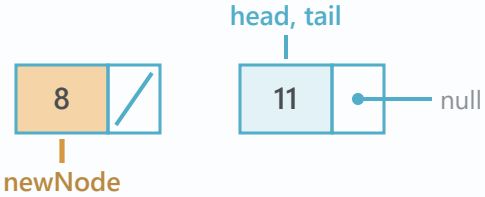
VARIABLES

head = **node(11)**

tail = **node(11)** size = **1**

data = **8**

Inserting at the Head — O(1)



Create node 8; its next is null.

Step 7 / 15

```
1 void addFirst(int data) {
▶     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

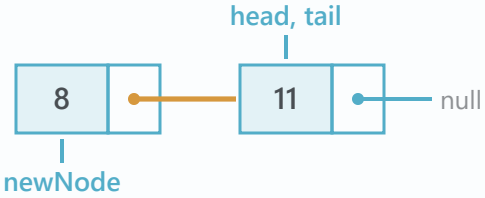
VARIABLES

head = **node(11)**

tail = **node(11)** size = **1**

data = **8** newNode = **node(8)**

Inserting at the Head — O(1)



`newNode.next = head` : 8 points to the old head (11). Changing head first would lose 11.

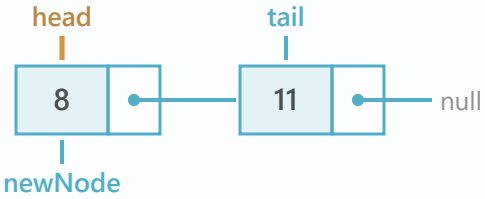
Step 8 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
▶     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

- `head = node(11)`
- `tail = node(11)`
- `size = 1`
- `data = 8`
- `newNode = node(8)`

Inserting at the Head — O(1)



`head = newNode` : head moves to 8: 8 → 11 → null.


```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
▶     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

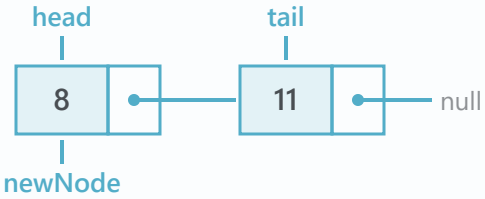
VARIABLES

`head = node(8)`

`tail = node(11)` `size = 1`

`data = 8` `newNode = node(8)`

Inserting at the Head — O(1)



`tail` is not null, so it stays at 11. `size++` → 2.

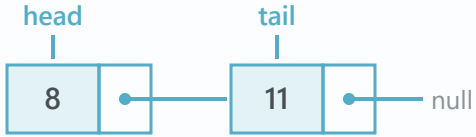
Step 10 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
▶     if (tail == null) tail = head;
▶     size++;
7 }
```

VARIABLES

- head = node(8)
- tail = node(11)
- size = 2
- data = 8
- newNode = node(8)

Inserting at the Head — O(1)



Call `addFirst(1)`. List: 8 → 11 → null.

Step 11 / 15

```
▶ void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

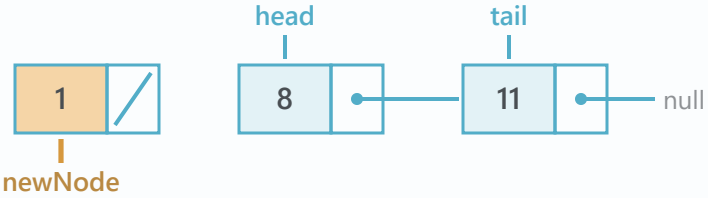
VARIABLES

`head = node(8)`

`tail = node(11)` `size = 2`

`data = 1`

Inserting at the Head — O(1)



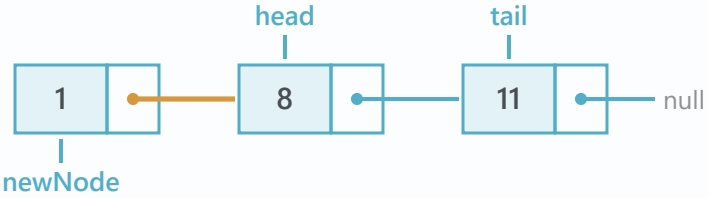
Create node 1; its next is null.


```
1 void addFirst(int data) {
▶     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

- head = node(8)
- tail = node(11)
- size = 2
- data = 1
- newNode = node(1)

Inserting at the Head — O(1)



`newNode.next = head` : 1 points to the old head (8). Changing head first would lose 8.

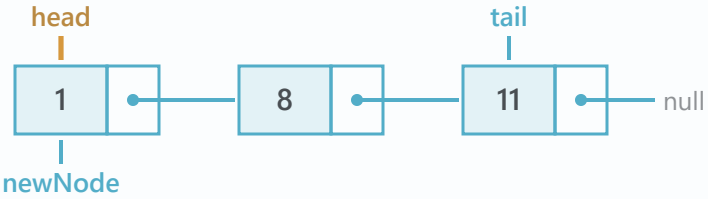
Step 13 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
▶     newNode.next = head;
4     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

VARIABLES

- `head = node(8)`
- `tail = node(11)`
- `size = 2`
- `data = 1`
- `newNode = node(1)`

Inserting at the Head — O(1)



`head = newNode` : head moves to 1: 1 → 8 → 11 → null.

Step 14 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
▶     head = newNode;
5     if (tail == null) tail = head;
6     size++;
7 }
```

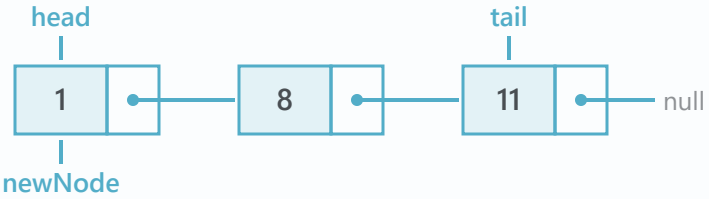
VARIABLES

`head = node(1)`

`tail = node(11)` `size = 2`

`data = 1` `newNode = node(1)`

Inserting at the Head — O(1)



`tail` stays at 11. `size++` → 3. However long the list, a fixed number of assignments: **O(1)**.

Step 15 / 15

```
1 void addFirst(int data) {
2     Node newNode = new Node(data);
3     newNode.next = head;
4     head = newNode;
▶     if (tail == null) tail = head;
▶     size++;
7 }
```

VARIABLES

`head = node(1)`

`tail = node(11)` `size = 3`

`data = 1` `newNode = node(1)`

Inserting at the Tail — O(n)



`addLast(10)` : create the new node.

Step 1 / 16

```
1  void addLast(int data) {
▶      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = null` `data = 10` `newNode = node(10)`

Inserting at the Tail — O(n)



`head == null` → the list is empty.

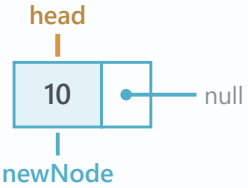
Step 2 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = null` `data = 10` `newNode = node(10)`

Inserting at the Tail — O(n)



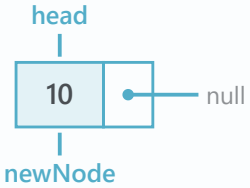
`head = newNode` : a one-node list.


```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
▶          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 10` `newNode = node(10)`

Inserting at the Tail — O(n)



`return` : no walk needed.

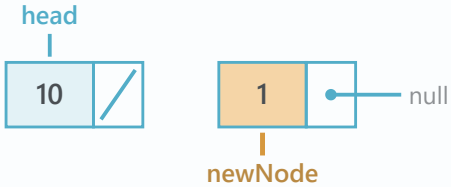
Step 4 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
▶          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 10` `newNode = node(10)`

Inserting at the Tail — O(n)



`addLast(1)` : create the new node.

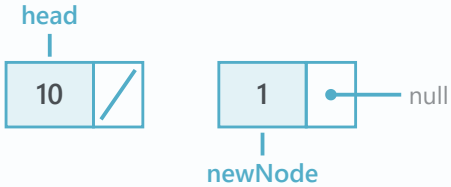
Step 5 / 16

```
1  void addLast(int data) {
▶       Node newNode = new Node(data);
3       if (head == null) {
4           head = newNode;
5           return;
6       }
7       Node temp = head;
8       while (temp.next != null)
9           temp = temp.next;
10      temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 1` `newNode = node(1)`

Inserting at the Tail — O(n)



The list is not empty → walk to the end.

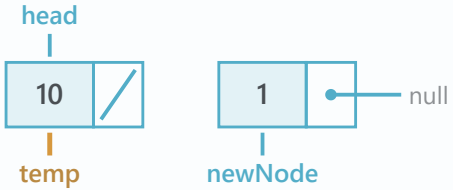
Step 6 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 1` `newNode = node(1)`

Inserting at the Tail — O(n)



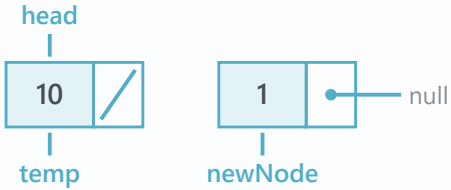
`temp = head` : start at the head to find the last node.


```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
▶      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 1` `newNode = node(1)` `temp = node(10)`

Inserting at the Tail — O(n)



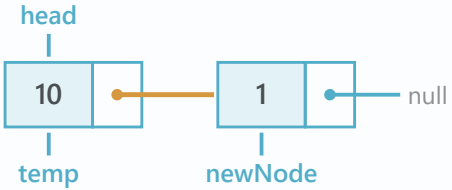
node(10).next == null → temp is at the last node; the loop ends.


```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
▶      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

- head = **node(10)**
- data = **1**
- newNode = **node(1)**
- temp = **node(10)**

Inserting at the Tail — O(n)



`temp.next = newNode` : 10 → 1 → null.

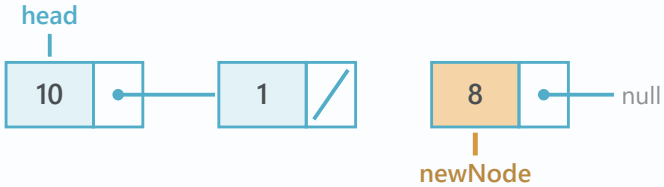
Step 9 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
▶      temp.next = newNode;
11 }
```

VARIABLES

- `head = node(10)`
- `data = 1`
- `newNode = node(1)`
- `temp = node(10)`

Inserting at the Tail — O(n)



`addLast(8)` : create the new node.

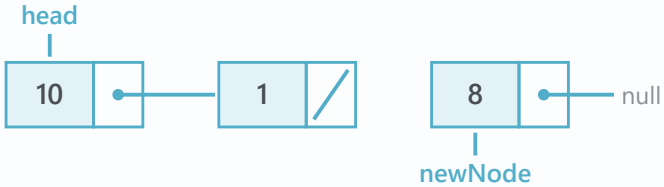
Step 10 / 16

```
1  void addLast(int data) {
▶      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 8` `newNode = node(8)`

Inserting at the Tail — O(n)



The list is not empty → walk to the end.

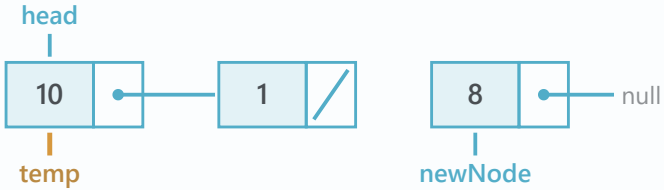
Step 11 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 8` `newNode = node(8)`

Inserting at the Tail — O(n)



`temp = head` : start at the head to find the last node.

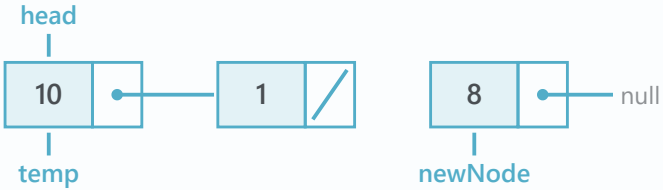
Step 12 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
▶      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 8` `newNode = node(8)` `temp = node(10)`

Inserting at the Tail — O(n)



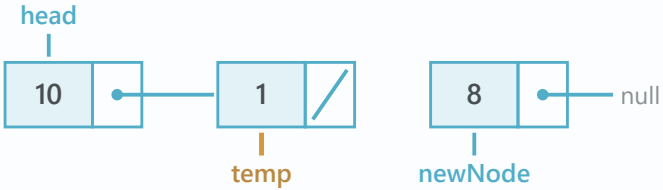
node(10).next is not null → advance.


```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
▶      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

VARIABLES

- head = **node(10)**
- data = **8**
- newNode = **node(8)**
- temp = **node(10)**

Inserting at the Tail — O(n)



`temp = temp.next` → node(1).

Step 14 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
▶          temp = temp.next;
10     temp.next = newNode;
11 }
```

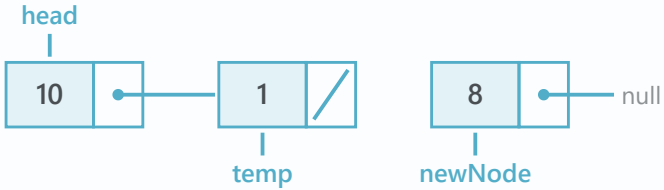
VARIABLES

`head = node(10)` `data = 8`

`newNode = node(8)`

`temp = node(1)`

Inserting at the Tail — O(n)



node(1).next == null → temp is at the last node; the loop ends.

Step 15 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
▶      while (temp.next != null)
9          temp = temp.next;
10     temp.next = newNode;
11 }
```

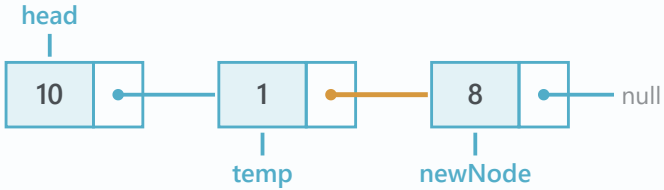
VARIABLES

`head = node(10)` `data = 8`

`newNode = node(8)`

`temp = node(1)`

Inserting at the Tail — O(n)



`temp.next = newNode` : 10 → 1 → 8 → null. Every insertion walks to the end: **O(n)**.

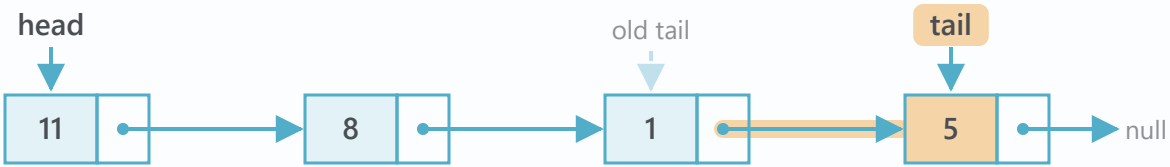
Step 16 / 16

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          return;
6      }
7      Node temp = head;
8      while (temp.next != null)
9          temp = temp.next;
▶      temp.next = newNode;
11 }
```

VARIABLES

`head = node(10)` `data = 8` `newNode = node(8)` `temp = node(1)`

Inserting at the Tail with a Tail Pointer — O(1)




```
class SinglyLinkedList {
    Node head, tail;  int size;
    void addLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
    }
}
```

No walk: O(1)

The last node is known; two assignments suffice.

The price

Every operation must keep `tail` up to date (inserting into an empty list, deleting the last node).

Deleting at the tail is still O(n)

In a singly linked list we cannot step back from `tail`.

Code Trace: `addLast` with `tail`



`addLast(10)` : create the new node.

Step 1 / 14

```
1  void addLast(int data) {
▶       Node newNode = new Node(data);
3       if (head == null) {
4           head = tail = newNode;
5       } else {
6           tail.next = newNode;
7           tail = newNode;
8       }
9       size++;
10 }
```

VARIABLES

`head = null` `tail = null`

`size = 0` `data = 10`

`newNode = node(10)`

Code Trace: `addLast` with `tail`



`head == null` → the list is empty.

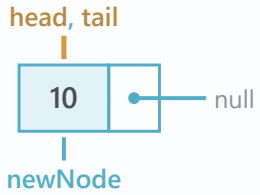
Step 2 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

`head = null` `tail = null` `size = 0` `data = 10` `newNode = node(10)`

Code Trace: `addLast` with `tail`



`head = tail = newNode` : the single node is both head and tail.

Step 3 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
▶          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
9      size++;
10 }
```

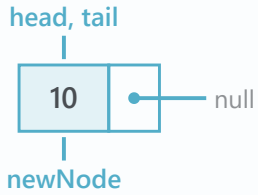
VARIABLES

head = **node(10)**

tail = **node(10)** size = **0**

data = **10** newNode = **node(10)**

Code Trace: `addLast` with `tail`



`size++` → 1.

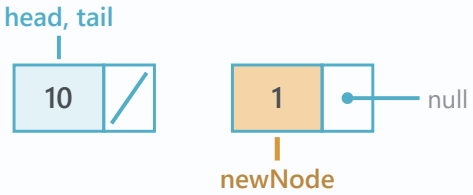
Step 4 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
▶      size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(10)`
- `size = 1`
- `data = 10`
- `newNode = node(10)`

Code Trace: `addLast` with `tail`



`addLast(1)` : create the new node.

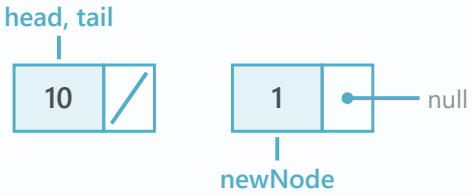
Step 5 / 14

```
1  void addLast(int data) {
▶       Node newNode = new Node(data);
3       if (head == null) {
4           head = tail = newNode;
5       } else {
6           tail.next = newNode;
7           tail = newNode;
8       }
9       size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(10)`
- `size = 1`
- `data = 1`
- `newNode = node(1)`

Code Trace: `addLast` with `tail`



The list is not empty; the last node is already known: node(10).

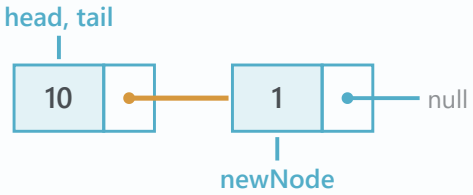
Step 6 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(10)`
- `size = 1`
- `data = 1`
- `newNode = node(1)`

Code Trace: `addLast` with `tail`



`tail.next = newNode` : links 10 → 1. No walk.

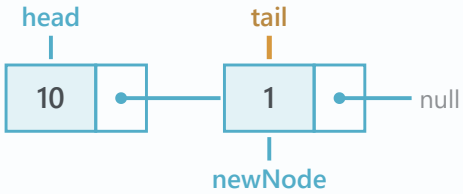
Step 7 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
▶          tail.next = newNode;
7          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

- `head` = **node(10)**
- `tail` = **node(10)**
- `size` = **1**
- `data` = **1**
- `newNode` = **node(1)**

Code Trace: `addLast` with `tail`



`tail = newNode` : tail is now 1.

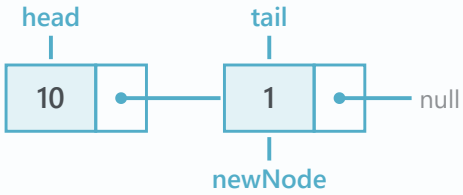
Step 8 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
▶          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(1)`
- `size = 1`
- `data = 1`
- `newNode = node(1)`

Code Trace: `addLast` with `tail`



`size++` → 2.

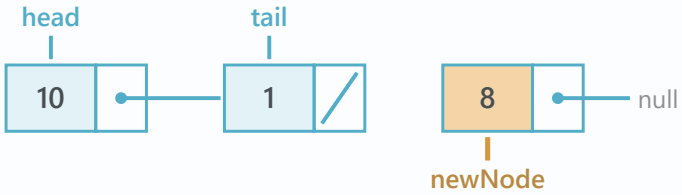
Step 9 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
▶      size++;
10 }
```

VARIABLES

- `head` = **node(10)**
- `tail` = **node(1)**
- `size` = **2**
- `data` = **1**
- `newNode` = **node(1)**

Code Trace: `addLast` with `tail`



`addLast(8)` : create the new node.

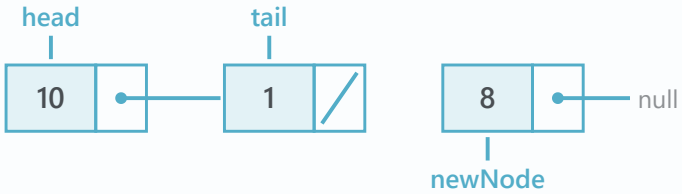
Step 10 / 14

```
1  void addLast(int data) {
▶       Node newNode = new Node(data);
3       if (head == null) {
4           head = tail = newNode;
5       } else {
6           tail.next = newNode;
7           tail = newNode;
8       }
9       size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(1)`
- `size = 2`
- `data = 8`
- `newNode = node(8)`

Code Trace: `addLast` with `tail`



The list is not empty; the last node is already known: node(1).

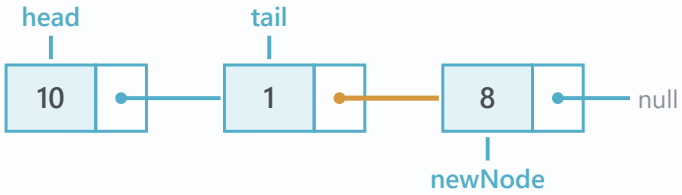
Step 11 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

- `head` = **node(10)**
- `tail` = **node(1)**
- `size` = **2**
- `data` = **8**
- `newNode` = **node(8)**

Code Trace: `addLast` with `tail`



`tail.next = newNode` : links 1 → 8. No walk.

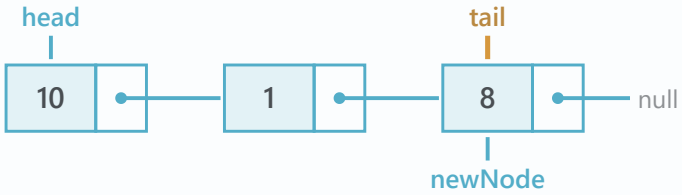
Step 12 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
▶          tail.next = newNode;
7          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

- `head` = **node(10)**
- `tail` = **node(1)**
- `size` = **2**
- `data` = **8**
- `newNode` = **node(8)**

Code Trace: `addLast` with `tail`



`tail = newNode` : tail is now 8.

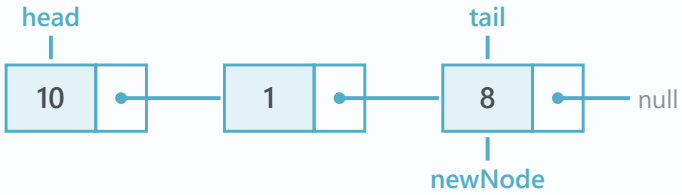
Step 13 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
▶          tail = newNode;
8      }
9      size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(8)`
- `size = 2`
- `data = 8`
- `newNode = node(8)`

Code Trace: `addLast` with `tail`



`size++` → 3. Each call makes a fixed number of assignments: **O(1)**.

Step 14 / 14

```
1  void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = tail = newNode;
5      } else {
6          tail.next = newNode;
7          tail = newNode;
8      }
▶      size++;
10 }
```

VARIABLES

- `head = node(10)`
- `tail = node(8)`
- `size = 3`
- `data = 8`
- `newNode = node(8)`

Inserting at a Given Position

```
void addAt(int data, int index) {   // index starts at 0
    if (index == 0) { addFirst(data); return; }
    Node temp = head;
    for (int i = 0; i < index - 1 && temp != null; i++)
        temp = temp.next;
    if (temp == null) {          // list shorter than index
        System.out.println("Position not found.");
        return;
    }
    Node newNode = new Node(data);
    newNode.next = temp.next;           // (1)
    temp.next = newNode;                // (2)
    if (temp == tail) tail = newNode;   // inserted at the end
    size++;
}
```

Index 0

Same as inserting at the head; handled separately.

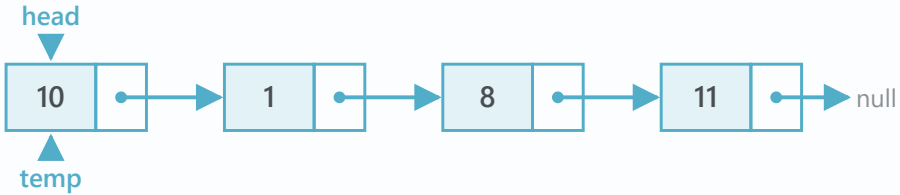
Order matters

First (1), then (2); the reverse cuts off the rest of the list.

Cost

Walking to the position is O(index), O(n) in the worst case; linking is O(1).

Code Trace: `addAt(5, 2)`



`addAt(5, 2)` : index ≠ 0, so this is not an insert at the head. `temp = head` ; we advance index − 1 = 1 step.

Step 1 / 8

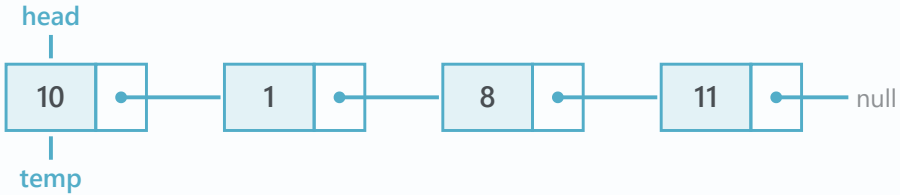
```
▶ Node temp = head;
2 for (int i = 0; i < index - 1 && temp != null; i++)
3     temp = temp.next;
4 if (temp == null) {          // list shorter than index
5     System.out.println("Position not found.");
6     return;
7 }
8 Node newNode = new Node(data);
9 newNode.next = temp.next;    // (1)
10 temp.next = newNode;        // (2)
```

VARIABLES

`head = node(10)` `data = 5`

`index = 2` `temp = node(10)`

Code Trace: `addAt(5, 2)`



i = 0: 0 < 1 and temp != null → loop body.

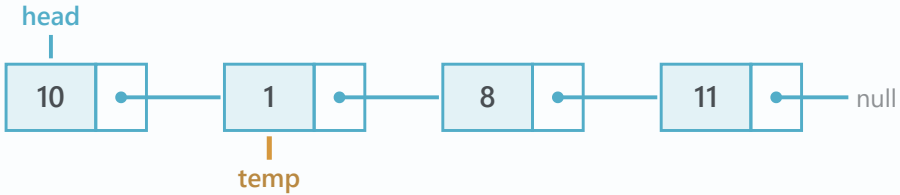
Step 2 / 8

```
1  Node temp = head;
▶  for (int i = 0; i < index - 1 && temp != null; i++)
3      temp = temp.next;
4  if (temp == null) {          // list shorter than index
5      System.out.println("Position not found.");
6      return;
7  }
8  Node newNode = new Node(data);
9  newNode.next = temp.next;    // (1)
10 temp.next = newNode;         // (2)
```

VARIABLES

head = **node(10)** | data = **5** | index = **2** | temp = **node(10)** | i = **0**

Code Trace: `addAt(5, 2)`



`temp = temp.next` → node(1).

Step 3 / 8

```
1  Node temp = head;
2  for (int i = 0; i < index - 1 && temp != null; i++)
▶      temp = temp.next;
4  if (temp == null) {          // list shorter than index
5      System.out.println("Position not found.");
6      return;
7  }
8  Node newNode = new Node(data);
9  newNode.next = temp.next;    // (1)
10 temp.next = newNode;         // (2)
```

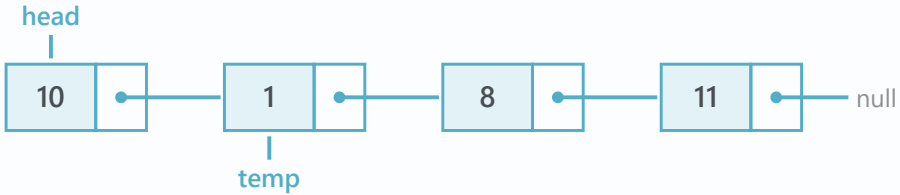
VARIABLES

`head = node(10)` `data = 5`

`index = 2` `temp = node(1)`

`i = 0`

Code Trace: `addAt(5, 2)`



i = 1: 1 < 1 is false → the loop ends. temp is at the node with index 1 (just before the insertion point).

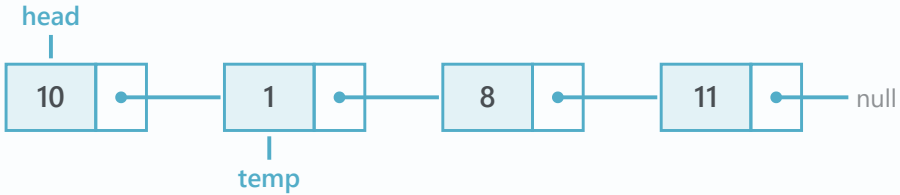
Step 4 / 8

```
1  Node temp = head;
▶ for (int i = 0; i < index - 1 && temp != null; i++)
3      temp = temp.next;
4  if (temp == null) {          // list shorter than index
5      System.out.println("Position not found.");
6      return;
7  }
8  Node newNode = new Node(data);
9  newNode.next = temp.next;    // (1)
10 temp.next = newNode;         // (2)
```

VARIABLES

head = **node(10)** | data = **5** | index = **2** | temp = **node(1)** | i = **1**

Code Trace: `addAt(5, 2)`



temp is not null → the position is valid; no error message.

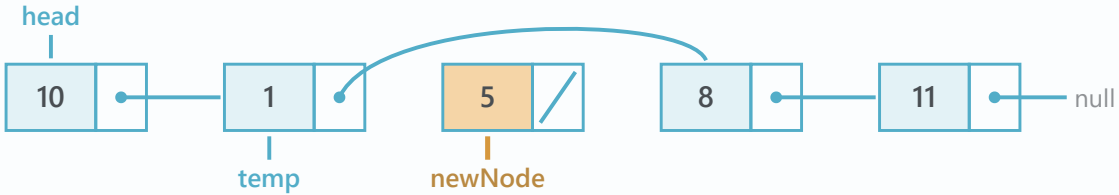
Step 5 / 8

```
1  Node temp = head;
2  for (int i = 0; i < index - 1 && temp != null; i++)
3      temp = temp.next;
▶  if (temp == null) {          // list shorter than index
5      System.out.println("Position not found.");
6      return;
7  }
8  Node newNode = new Node(data);
9  newNode.next = temp.next;   // (1)
10 temp.next = newNode;        // (2)
```

VARIABLES

`head = node(10)` `data = 5` `index = 2` `temp = node(1)`

Code Trace: `addAt(5, 2)`



Create the new node (5). (The drawing shifts 8 and 11 to the right; nothing moves in memory.) 1 still points to 8.

Step 6 / 8

```
1  Node temp = head;
2  for (int i = 0; i < index - 1 && temp != null; i++)
3      temp = temp.next;
4  if (temp == null) {          // list shorter than index
5      System.out.println("Position not found.");
6      return;
7  }
▶  Node newNode = new Node(data);
9  newNode.next = temp.next;    // (1)
10 temp.next = newNode;         // (2)
```

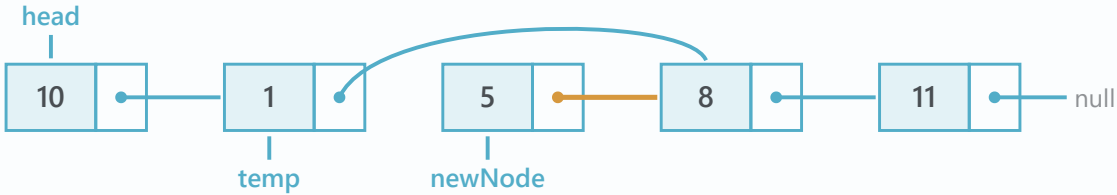
VARIABLES

`head = node(10)` `data = 5`

`index = 2` `temp = node(1)`

`newNode = node(5)`

Code Trace: `addAt(5, 2)`



(1) `newNode.next = temp.next` : 5 points to 8. The list has not changed yet.

Step 7 / 8

```
1  Node temp = head;
2  for (int i = 0; i < index - 1 && temp != null; i++)
3      temp = temp.next;
4  if (temp == null) {          // list shorter than index
5      System.out.println("Position not found.");
6      return;
7  }
8  Node newNode = new Node(data);
▶  newNode.next = temp.next;    // (1)
10 temp.next = newNode;         // (2)
```

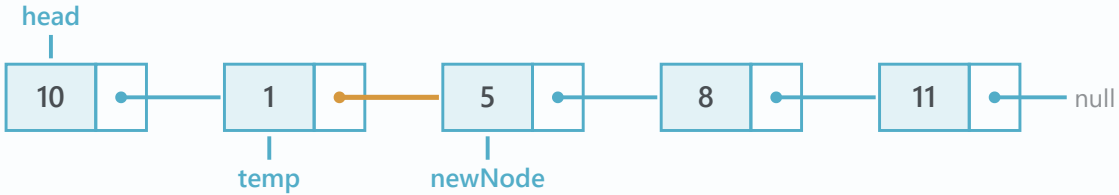
VARIABLES

`head` = **node(10)** `data` = **5**

`index` = **2** `temp` = **node(1)**

`newNode` = **node(5)**

Code Trace: `addAt(5, 2)`



(2) `temp.next = newNode` : 10 → 1 → 5 → 8 → 11. Only **2 links** changed; in the reverse order the path to 8 would be lost.

Step 8 / 8

```
1 Node temp = head;
2 for (int i = 0; i < index - 1 && temp != null; i++)
3     temp = temp.next;
4 if (temp == null) {          // list shorter than index
5     System.out.println("Position not found.");
6     return;
7 }
8 Node newNode = new Node(data);
9 newNode.next = temp.next;   // (1)
▶ temp.next = newNode;        // (2)
```

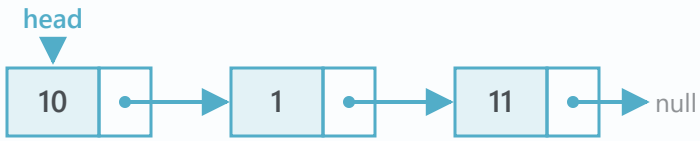
VARIABLES

`head = node(10)` `data = 5`

`index = 2` `temp = node(1)`

`newNode = node(5)`

Deleting at the Head — O(1)



Call 1: `head` is not null (10 → 1 → 11 → null).

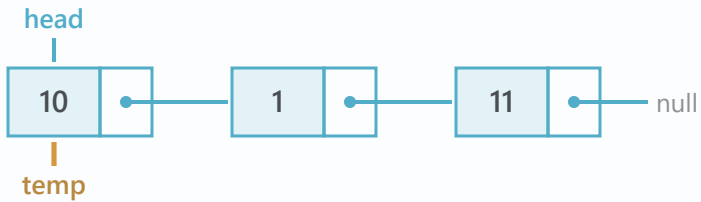
Step 1 / 16

```
1 Node removeFirst() {
▶     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(10)`

Deleting at the Head — O(1)



`temp = head` : save the node being removed (10).

Step 2 / 16

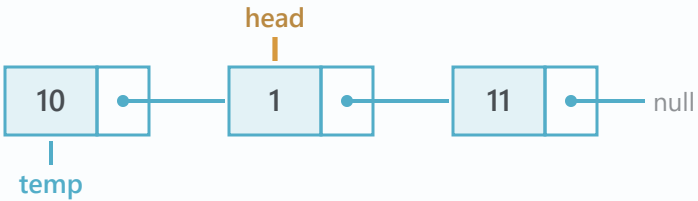
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
▶     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(10)`

`temp = node(10)`

Deleting at the Head — O(1)



`head = head.next` : head is node(1); 10 is now unreachable.

Step 3 / 16

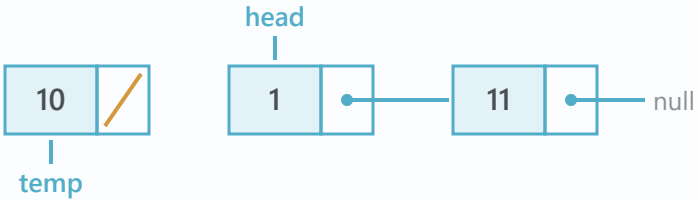
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
▶     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(1)`

`temp = node(10)`

Deleting at the Head — O(1)



`temp.next = null` : 10 is fully detached.

Step 4 / 16

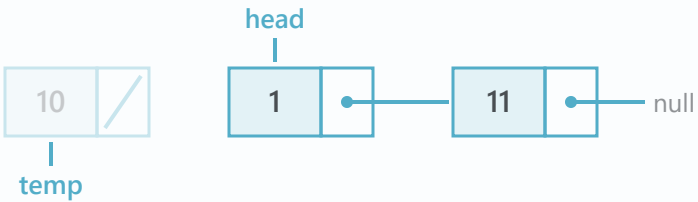
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
▶     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(1)`

`temp = node(10)`

Deleting at the Head — O(1)



`return temp` : 10 goes back to the caller. Once unreferenced, it is reclaimed by the GC. **O(1)**.

Step 5 / 16

```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
▶     return temp;
8 }
```

VARIABLES

- `head = node(1)`
- `temp = node(10)`

Deleting at the Head — O(1)



Call 2: `head` is not null (1 → 11 → null).

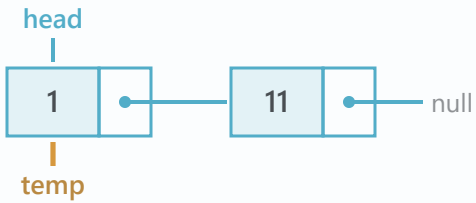
Step 6 / 16

```
1 Node removeFirst() {
▶       if (head == null)
3           return null;
4       Node temp = head;
5       head = head.next;  if (head == null) tail = null;
6       temp.next = null;  size--;
7       return temp;
8   }
```

VARIABLES

`head = node(1)`

Deleting at the Head — O(1)



`temp = head` : save the node being removed (1).

Step 7 / 16

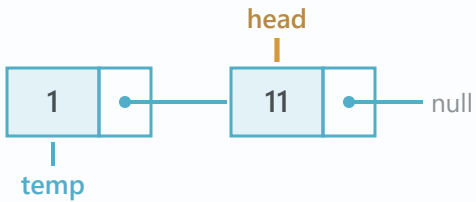
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
▶     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(1)`

`temp = node(1)`

Deleting at the Head — O(1)



`head = head.next` : head is node(11); 1 is now unreachable.

Step 8 / 16

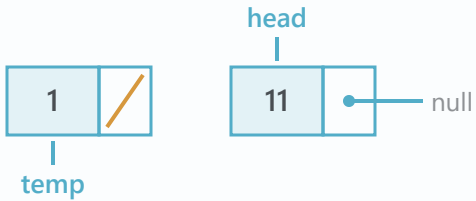
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
▶     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(11)`

`temp = node(1)`

Deleting at the Head — O(1)



`temp.next = null` : 1 is fully detached.

Step 9 / 16

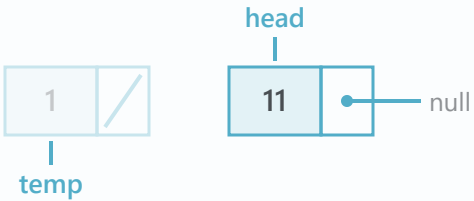
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
▶     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(11)`

`temp = node(1)`

Deleting at the Head — O(1)



`return temp` : 1 goes back to the caller. Once unreferenced, it is reclaimed by the GC. **O(1)**.

Step 10 / 16

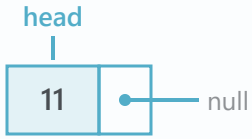
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
▶     return temp;
8 }
```

VARIABLES

`head = node(11)`

`temp = node(1)`

Deleting at the Head — O(1)



Call 3: `head` is not null (11 → null).

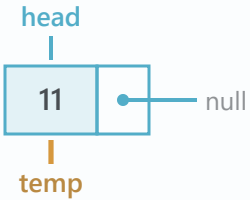
Step 11 / 16

```
1 Node removeFirst() {
▶     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(11)`

Deleting at the Head — O(1)



`temp = head` : save the node being removed (11).

Step 12 / 16

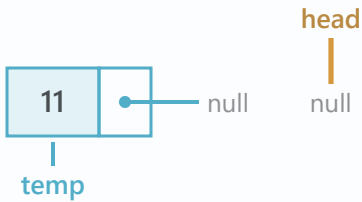
```
1 Node removeFirst() {
2     if (head == null)
3         return null;
▶     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = node(11)`

`temp = node(11)`

Deleting at the Head — O(1)



`head = head.next` : head is **null** and the list is empty; 11 is now unreachable.

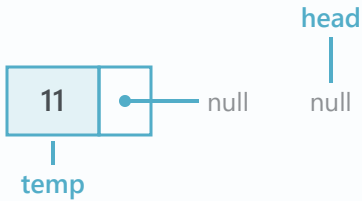
Step 13 / 16

```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
▶     head = head.next;   if (head == null) tail = null;
6     temp.next = null;   size--;
7     return temp;
8 }
```

VARIABLES

`head = null` `temp = node(11)`

Deleting at the Head — O(1)



`temp.next = null` : 11 is fully detached.

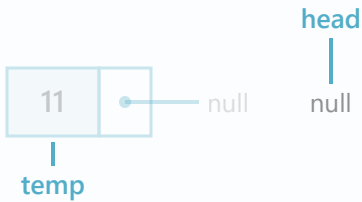
Step 14 / 16

```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
▶     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = null` `temp = node(11)`

Deleting at the Head — O(1)



`return temp` : 11 goes back to the caller. Once unreferenced, it is reclaimed by the GC. **O(1)**.

Step 15 / 16

```
1 Node removeFirst() {
2     if (head == null)
3         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
▶     return temp;
8 }
```

VARIABLES

`head = null` `temp = node(11)`

Deleting at the Head — O(1)



On an empty list, `head == null` → **return null**. Without this check, `head.next` would throw a NullPointerException.

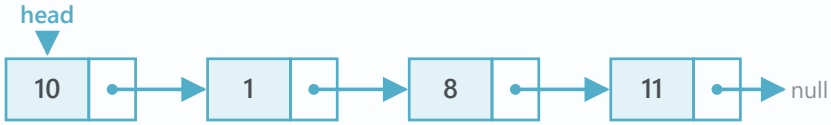
Step 16 / 16

```
1 Node removeFirst() {
▶     if (head == null)
▶         return null;
4     Node temp = head;
5     head = head.next;  if (head == null) tail = null;
6     temp.next = null;  size--;
7     return temp;
8 }
```

VARIABLES

`head = null`

Deleting at the Tail — O(n)



The list is not empty.

Step 1 / 9

```
1  void removeLast() {
▶       if (head == null) return;
3       if (head.next == null) {  // single node
4           head = tail = null;  size--;
5           return;
6       }
7       Node temp = head;
8       while (temp.next.next != null)
9           temp = temp.next;
10      temp.next = null;  tail = temp;  size--;
11  }
```

VARIABLES

head = **node(10)**

Deleting at the Tail — O(n)



`head.next` is not null → there is more than one node.

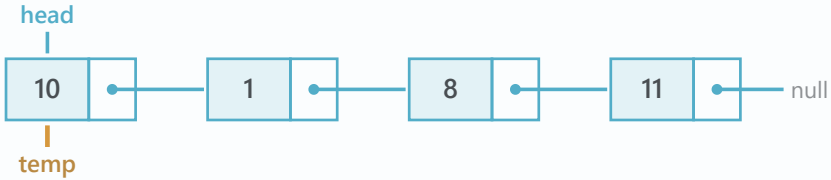
Step 2 / 9

```
1  void removeLast() {
2      if (head == null) return;
▶      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
8      while (temp.next.next != null)
9          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)`

Deleting at the Tail — O(n)



`temp = head`. Goal: find the **second-to-last** node (its link will change).

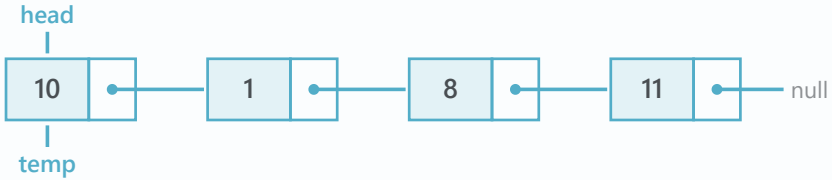
Step 3 / 9

```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
▶      Node temp = head;
8      while (temp.next.next != null)
9          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(10)`

Deleting at the Tail — O(n)



node(10).next.next = 8 ≠ null → advance.

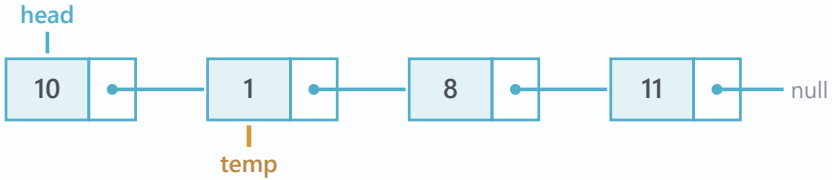
Step 4 / 9

```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
▶      while (temp.next.next != null)
9          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(10)`

Deleting at the Tail — O(n)



`temp = temp.next` → node(1).

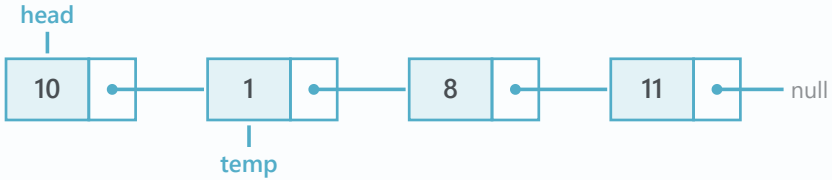
Step 5 / 9

```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
8      while (temp.next.next != null)
▶          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(1)`

Deleting at the Tail — O(n)



node(1).next.next = 11 ≠ null → advance.

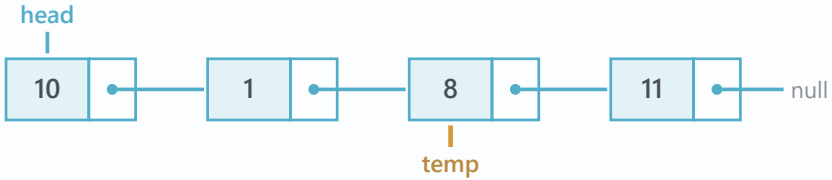
Step 6 / 9

```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
▶      while (temp.next.next != null)
9          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(1)`

Deleting at the Tail — O(n)



`temp = temp.next` → node(8).

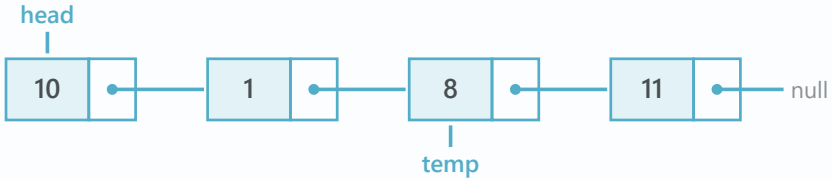
Step 7 / 9

```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
8      while (temp.next.next != null)
▶          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(8)`

Deleting at the Tail — O(n)



node(8).next.next == null → temp is at the second-to-last node.

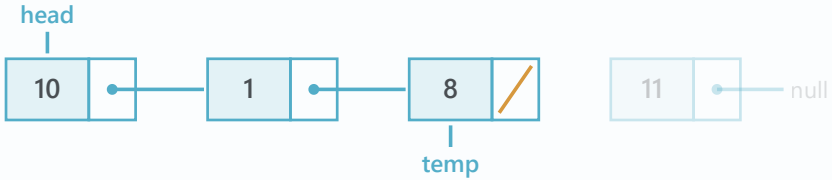
Step 8 / 9

```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
▶      while (temp.next.next != null)
9          temp = temp.next;
10     temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(8)`

Deleting at the Tail — O(n)



`temp.next = null` : 11 leaves the list → 10 → 1 → 8 → null. **O(n)**; a `tail` pointer doesn't help, because we cannot step back.

Step 9 / 9

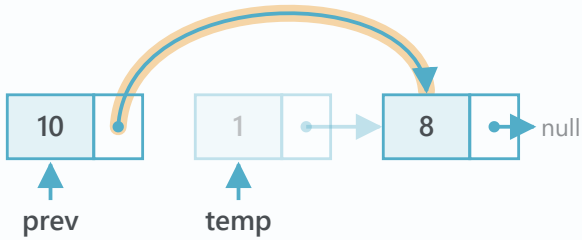
```
1  void removeLast() {
2      if (head == null) return;
3      if (head.next == null) {  // single node
4          head = tail = null;  size--;
5          return;
6      }
7      Node temp = head;
8      while (temp.next.next != null)
9          temp = temp.next;
▶      temp.next = null;  tail = temp;  size--;
11 }
```

VARIABLES

`head = node(10)` `temp = node(8)`

Deleting a Given Item

```
void remove(int key) {
    if (head == null) return;
    if (head.data == key) {
        head = head.next;  size--;
        if (head == null) tail = null;  return;
    }
    Node prev = head;
    Node temp = head.next;
    while (temp != null) {
        if (temp.data == key) {
            prev.next = temp.next;
            if (temp == tail) tail = prev;  size--;
            return;
        }
        prev = temp;
        temp = temp.next;
    }
    System.out.println("Not found.");
}
```

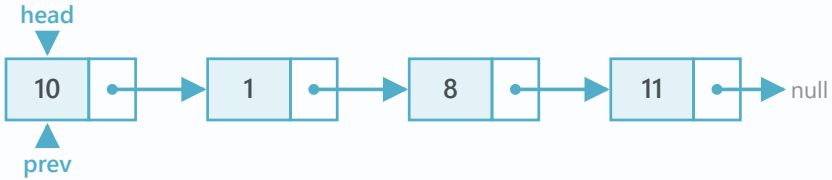


`prev.next = temp.next`

the doomed node is "skipped"

- Two pointers walk together: deleting needs the **previous** node.
- Deleting the head is a separate case.
- Search O(n) + relinking O(1) → **O(n)**.

Code Trace: `remove(8)`



`remove(8)`: the list is not empty and the head 10 ≠ 8 (both `if`s passed). `prev = head`.

Step 1 / 10

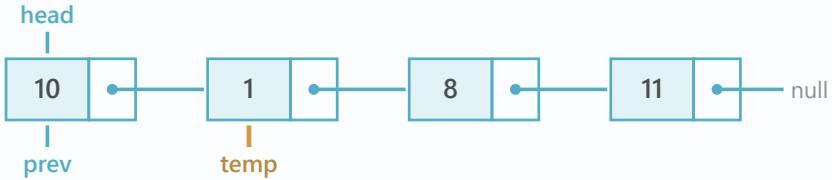
```
▶ Node prev = head;
2 Node temp = head.next;
3 while (temp != null) {
4     if (temp.data == key) {
5         prev.next = temp.next;
6         if (temp == tail) tail = prev;  size--;
7         return;
8     }
9     prev = temp;
10    temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head = node(10)` `key = 8`

`prev = node(10)`

Code Trace: `remove(8)`



`temp = head.next` : the two pointers walk side by side; deleting needs the **previous** node.

Step 2 / 10

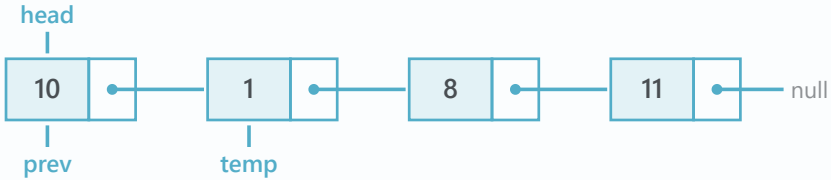
```
1  Node prev = head;
▶  Node temp = head.next;
3  while (temp != null) {
4      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head = node(10)` `key = 8`

`prev = node(10)` `temp = node(1)`

Code Trace: `remove(8)`



temp = node(1) is not null → loop.

Step 3 / 10

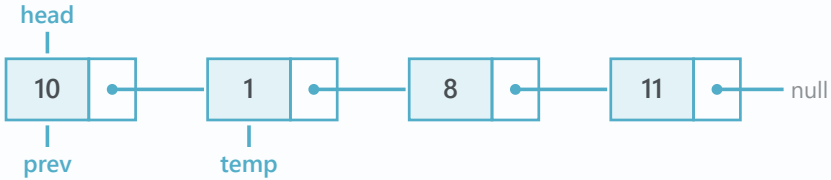
```
1  Node prev = head;
2  Node temp = head.next;
▶  while (temp != null) {
4      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head` = **node(10)** `key` = **8**

`prev` = **node(10)** `temp` = **node(1)**

Code Trace: `remove(8)`



1 == 8? No.

Step 4 / 10

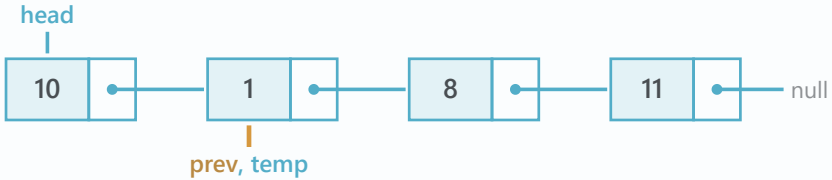
```
1  Node prev = head;
2  Node temp = head.next;
3  while (temp != null) {
▶      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head` = **node(10)** `key` = **8**

`prev` = **node(10)** `temp` = **node(1)**

Code Trace: `remove(8)`



`prev = temp` → node(1).

Step 5 / 10

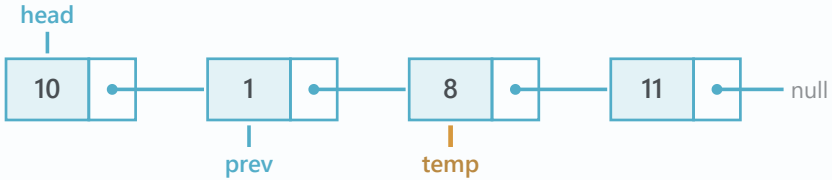
```
1  Node prev = head;
2  Node temp = head.next;
3  while (temp != null) {
4      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
▶      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head = node(10)` `key = 8`

`prev = node(1)` `temp = node(1)`

Code Trace: `remove(8)`



`temp = temp.next` → node(8).

Step 6 / 10

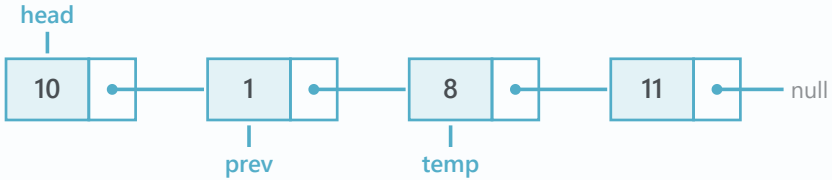
```
1  Node prev = head;
2  Node temp = head.next;
3  while (temp != null) {
4      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
▶      temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head` = **node(10)** `key` = **8**

`prev` = **node(1)** `temp` = **node(8)**

Code Trace: `remove(8)`



temp = node(8) is not null → loop.

Step 7 / 10

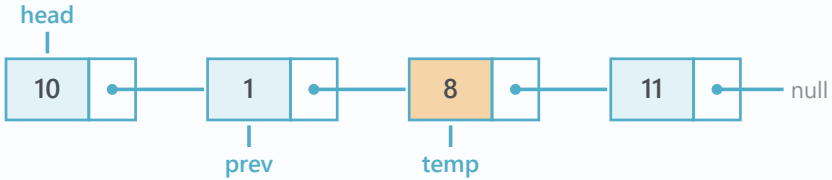
```
1  Node prev = head;
2  Node temp = head.next;
▶  while (temp != null) {
4      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head` = **node(10)** `key` = **8**

`prev` = **node(1)** `temp` = **node(8)**

Code Trace: `remove(8)`



8 == 8 → found the node to delete.

Step 8 / 10

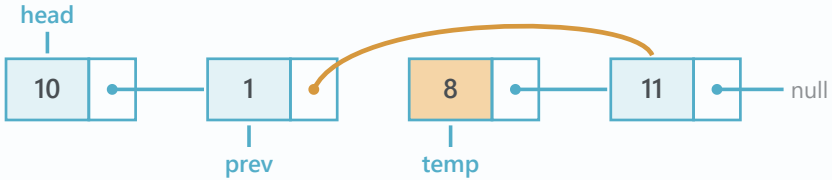
```
1  Node prev = head;
2  Node temp = head.next;
3  while (temp != null) {
▶      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head` = **node(10)** `key` = **8**

`prev` = **node(1)** `temp` = **node(8)**

Code Trace: `remove(8)`



`prev.next = temp.next` : 1 now points to 11; 8 is "skipped".

Step 9 / 10

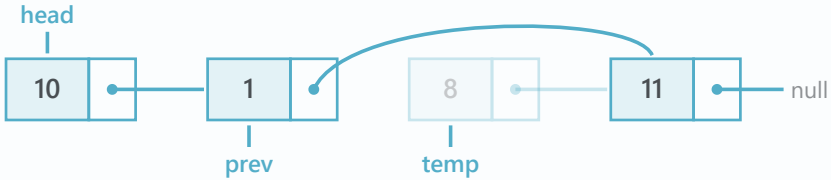
```
1  Node prev = head;
2  Node temp = head.next;
3  while (temp != null) {
4      if (temp.data == key) {
▶          prev.next = temp.next;
▶          if (temp == tail) tail = prev;  size--;
7          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head` = **node(10)** `key` = **8**

`prev` = **node(1)** `temp` = **node(8)**

Code Trace: `remove(8)`



`return`. List: 10 → 1 → 11 → null. Search O(n) + relinking O(1) → **O(n)**.

Step 10 / 10

```
1  Node prev = head;
2  Node temp = head.next;
3  while (temp != null) {
4      if (temp.data == key) {
5          prev.next = temp.next;
6          if (temp == tail) tail = prev;  size--;
▶          return;
8      }
9      prev = temp;
10     temp = temp.next;
11 }
12 System.out.println("Not found.");
```

VARIABLES

`head = node(10)` `key = 8`

`prev = node(1)` `temp = node(8)`

Reversing a List

```
void reverse() {
    tail = head;                    // old head becomes the new t
    Node prev = null;
    Node current = head;
    Node next = null;
    while (current != null) {
        next = current.next;  // 1) save next
        current.next = prev;  // 2) reverse the link
        prev = current;       // 3) advance prev
        current = next;       // 4) advance current
    }
    head = prev;
}
```

Three pointers

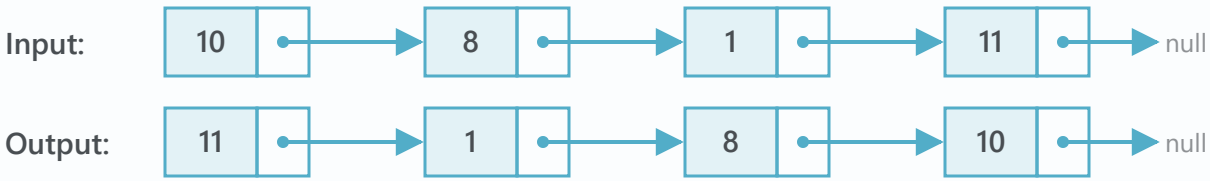
prev, current, next

No new nodes

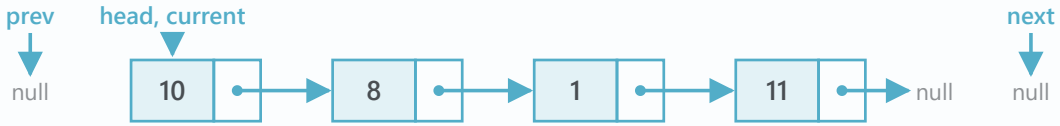
Each round reverses one link.

O(n) time

O(1) extra space



Code Trace: `reverse()`



Start: `prev = null`, `current = head` (10), `next = null`. No new nodes will be created.

Step 1 / 19

```
▶ tail = head;                  // old head becomes the new tail
▶ Node prev = null;
▶ Node current = head;
▶ Node next = null;
5 while (current != null) {
6     next = current.next;  // 1) save next
7     current.next = prev;  // 2) reverse the link
8     prev = current;       // 3) advance prev
9     current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = null`

`current = node(10)` `next = null`

Code Trace: `reverse()`



Round 1: current = node(10) is not null.

Step 2 / 19

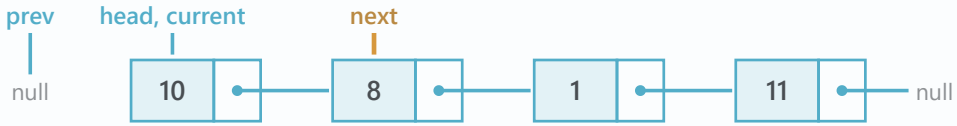
```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
▶  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = null`

`current = node(10)` `next = null`

Code Trace: `reverse()`



1) `next = current.next` = node(8): so the rest of the list survives the cut.

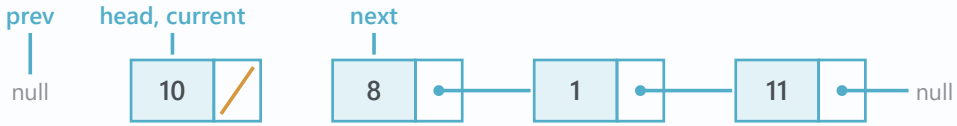
Step 3 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
▶      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = null` `current = node(10)` `next = node(8)`

Code Trace: `reverse()`



2) `current.next = prev` : 10 now points to null (link reversed).

Step 4 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
▶      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = null`

`current = node(10)`

`next = node(8)`

Code Trace: `reverse()`



3–4) `prev = current; current = next` : both pointers move one node forward.

Step 5 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
▶      prev = current;       // 3) advance prev
▶      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(10)` `current = node(8)` `next = node(8)`

Code Trace: `reverse()`



Round 2: current = node(8) is not null.

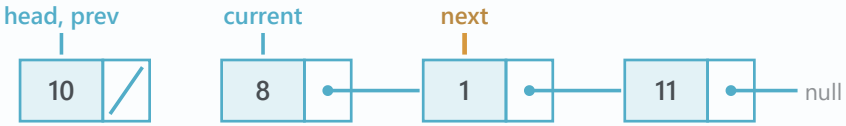
Step 6 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
▶  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(10)` `current = node(8)` `next = node(8)`

Code Trace: `reverse()`



1) `next = current.next` = node(1): so the rest of the list survives the cut.

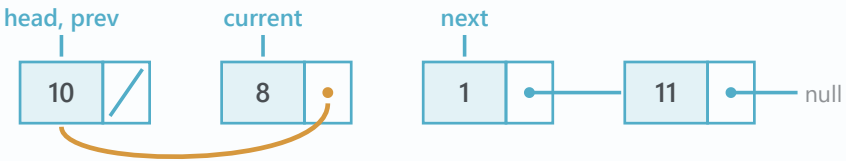
Step 7 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
▶      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(10)` `current = node(8)` `next = node(1)`

Code Trace: `reverse()`



2) `current.next = prev` : 8 now points to node(10) (link reversed).

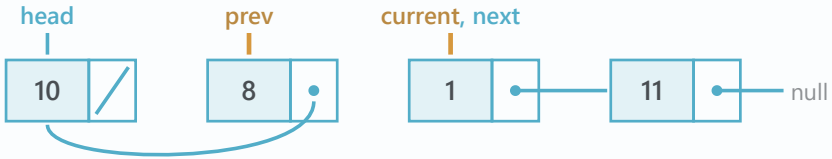
Step 8 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
▶      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(10)` `current = node(8)` `next = node(1)`

Code Trace: `reverse()`



3–4) `prev = current; current = next` : both pointers move one node forward.

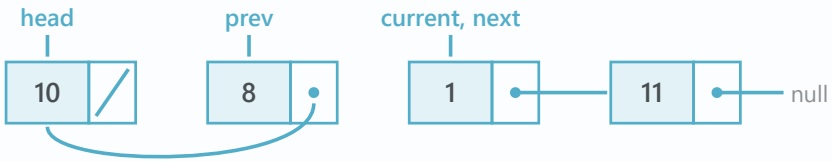
Step 9 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
▶      prev = current;       // 3) advance prev
▶      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(8)` `current = node(1)` `next = node(1)`

Code Trace: `reverse()`



Round 3: current = node(1) is not null.

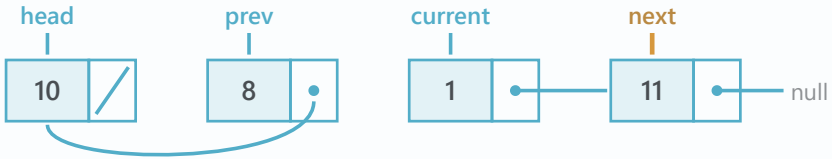
Step 10 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
▶  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(8)` `current = node(1)` `next = node(1)`

Code Trace: `reverse()`



1) `next = current.next` = node(11): so the rest of the list survives the cut.

Step 11 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
▶      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

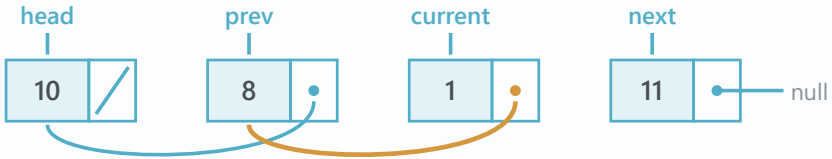
VARIABLES

`head` = **node(10)** `prev` = **node(8)**

`current` = **node(1)**

`next` = **node(11)**

Code Trace: `reverse()`



2) `current.next = prev` : 1 now points to node(8) (link reversed).

Step 12 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
▶      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

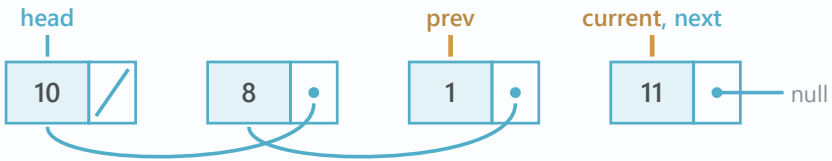
VARIABLES

`head` = **node(10)** `prev` = **node(8)**

`current` = **node(1)**

`next` = **node(11)**

Code Trace: `reverse()`



3–4) `prev = current; current = next` : both pointers move one node forward.

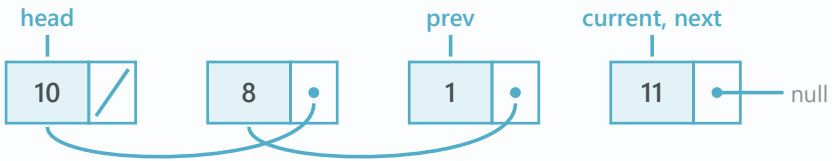
Step 13 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
▶      prev = current;       // 3) advance prev
▶      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(1)` `current = node(11)` `next = node(11)`

Code Trace: `reverse()`



Round 4: current = node(11) is not null.

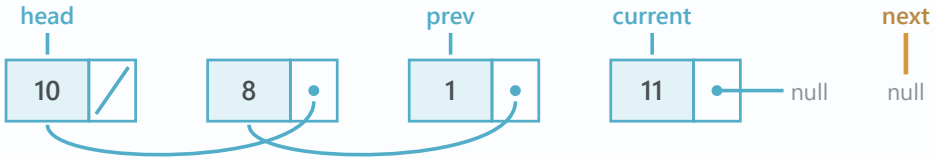
Step 14 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
▶  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

- `head = node(10)`
- `prev = node(1)`
- `current = node(11)`
- `next = node(11)`

Code Trace: `reverse()`



1) `next = current.next` = null: so the rest of the list survives the cut.

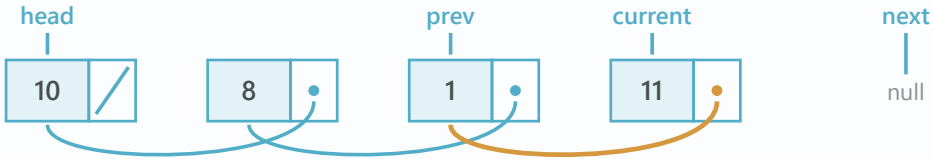
Step 15 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
▶      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(1)` `current = node(11)` `next = null`

Code Trace: `reverse()`



2) `current.next = prev` : 11 now points to node(1) (link reversed).

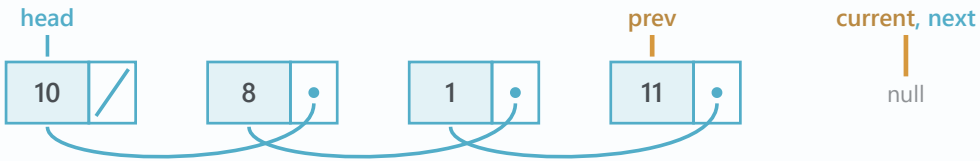
Step 16 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
▶      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(1)` `current = node(11)` `next = null`

Code Trace: `reverse()`



3–4) `prev = current; current = next` : both pointers move one node forward.

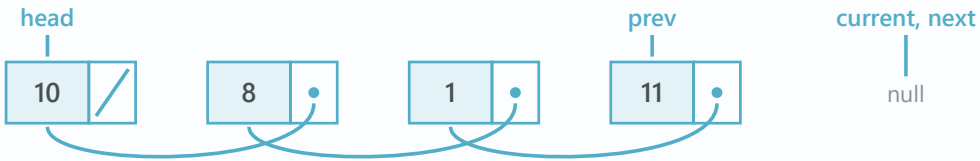
Step 17 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
▶      prev = current;       // 3) advance prev
▶      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(11)` `current = null` `next = null`

Code Trace: `reverse()`



`current == null` → the loop ends; `prev` is at the old last node (11).

Step 18 / 19

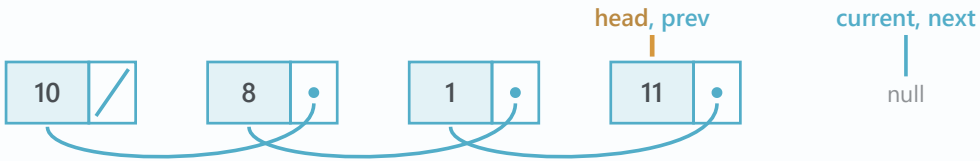
```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
▶  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
11 head = prev;
```

VARIABLES

`head = node(10)` `prev = node(11)`

`current = null` `next = null`

Code Trace: `reverse()`



`head = prev` : the new list is 11 → 1 → 8 → 10 → null. **O(n)** time, **O(1)** extra space.

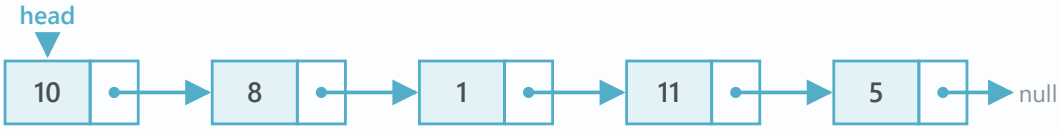
Step 19 / 19

```
1  tail = head;                 // old head becomes the new tail
2  Node prev = null;
3  Node current = head;
4  Node next = null;
5  while (current != null) {
6      next = current.next;  // 1) save next
7      current.next = prev;  // 2) reverse the link
8      prev = current;       // 3) advance prev
9      current = next;       // 4) advance current
10 }
▶  head = prev;
```

VARIABLES

`head = node(11)` `prev = node(11)` `current = null` `next = null`

Finding the Middle: Slow and Fast Pointers



The list is not empty (an empty list throws an exception).

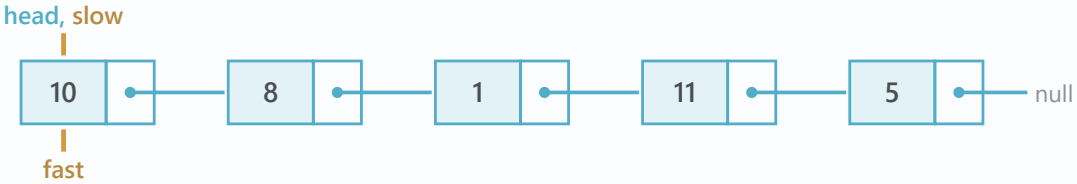
Step 1 / 10

```
1  int findMiddle() {
▶      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
6          slow = slow.next;       // 1 step
7          fast = fast.next.next;  // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

head = **node(10)**

Finding the Middle: Slow and Fast Pointers



Both pointers start at the head. We find the middle without knowing the length.

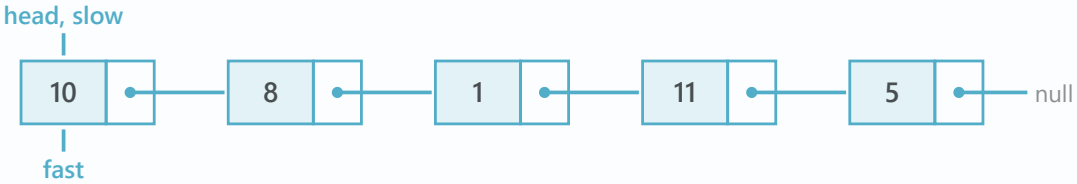
Step 2 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
▶      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
6          slow = slow.next;        // 1 step
7          fast = fast.next.next;   // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

- head = **node(10)**
- slow = **node(10)**
- fast = **node(10)**

Finding the Middle: Slow and Fast Pointers



fast = node(10) and fast.next = node(8) are not null → round 1.

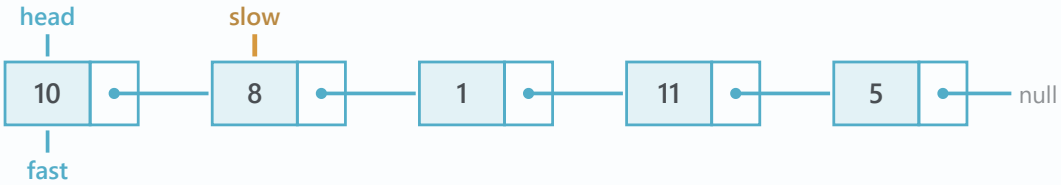
Step 3 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
▶      while (fast != null && fast.next != null) {
6          slow = slow.next;        // 1 step
7          fast = fast.next.next;   // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

- head = **node(10)**
- slow = **node(10)**
- fast = **node(10)**

Finding the Middle: Slow and Fast Pointers



`slow` moves 1 step → node(8).

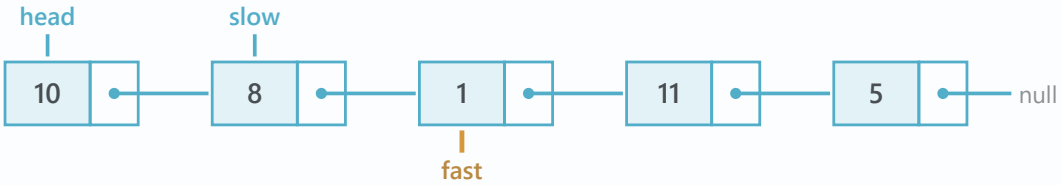
Step 4 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
▶          slow = slow.next;        // 1 step
7          fast = fast.next.next;   // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(8)` `fast = node(10)`

Finding the Middle: Slow and Fast Pointers



`fast` moves 2 steps → node(1).

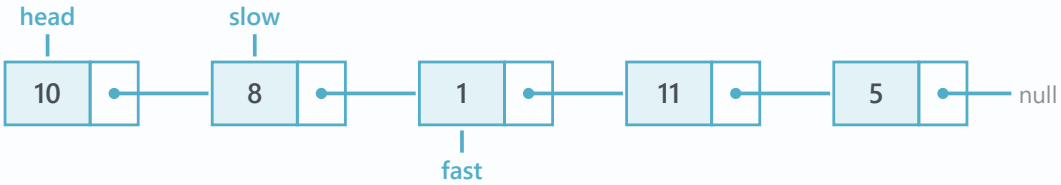
Step 5 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
6          slow = slow.next;       // 1 step
▶          fast = fast.next.next;  // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(8)` `fast = node(1)`

Finding the Middle: Slow and Fast Pointers



fast = node(1) and fast.next = node(11) are not null → round 2.

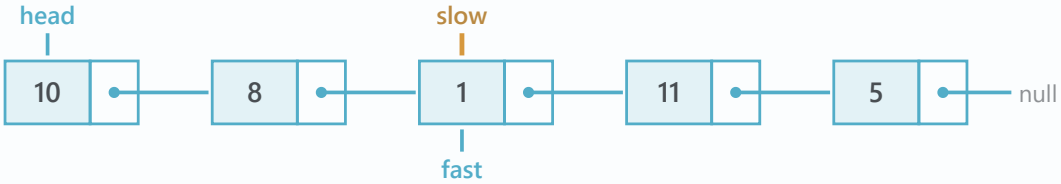
Step 6 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
▶      while (fast != null && fast.next != null) {
6          slow = slow.next;        // 1 step
7          fast = fast.next.next;   // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(8)` `fast = node(1)`

Finding the Middle: Slow and Fast Pointers



`slow` moves 1 step → node(1).

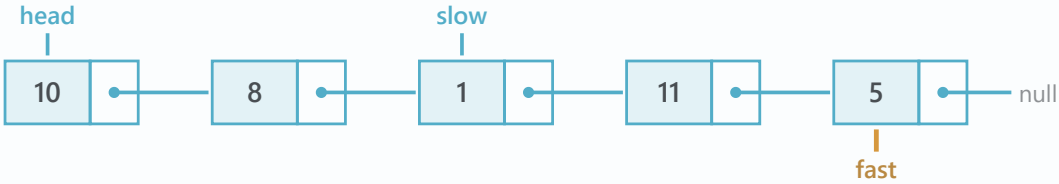
Step 7 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
▶          slow = slow.next;        // 1 step
7          fast = fast.next.next;   // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(1)` `fast = node(1)`

Finding the Middle: Slow and Fast Pointers



`fast` moves 2 steps → node(5).

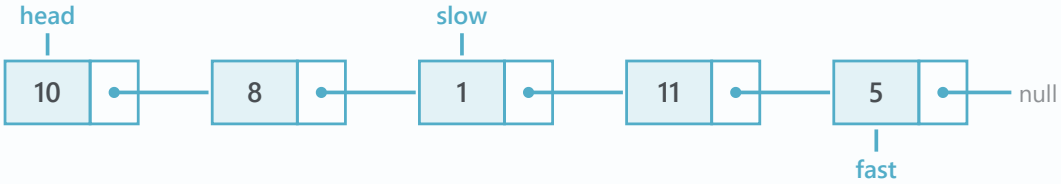
Step 8 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
6          slow = slow.next;       // 1 step
▶          fast = fast.next.next;  // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(1)` `fast = node(5)`

Finding the Middle: Slow and Fast Pointers



fast.next == null → fast has reached the end; the loop ends. In 2 rounds slow took 2 steps and fast took 4.

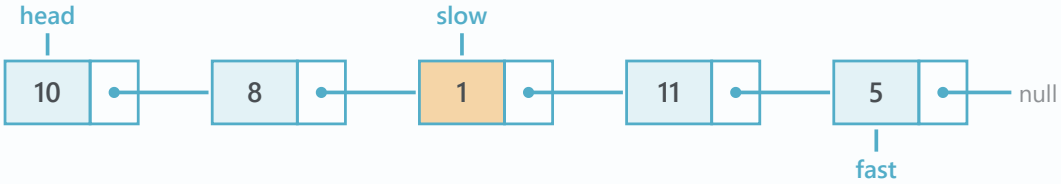
Step 9 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
▶      while (fast != null && fast.next != null) {
6          slow = slow.next;        // 1 step
7          fast = fast.next.next;   // 2 steps
8      }
9      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(1)` `fast = node(5)`

Finding the Middle: Slow and Fast Pointers



return 1: the slow pointer is in the middle. One pass, **O(n)**. For an even length it returns the second middle (10, 8, 1, 11 → 1).

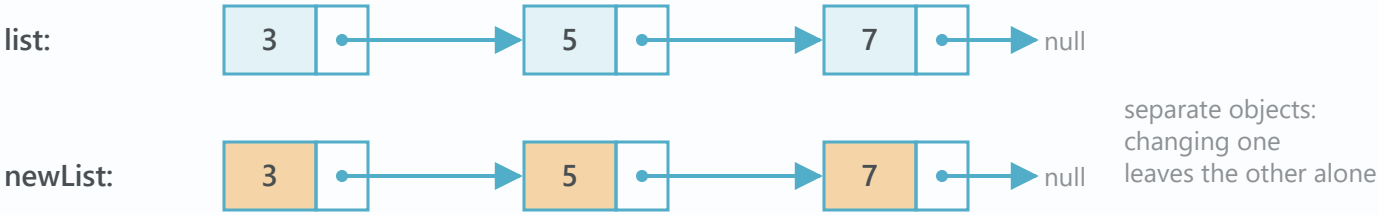
Step 10 / 10

```
1  int findMiddle() {
2      if (head == null)
3          throw new IllegalStateException("List is empty");
4      Node slow = head, fast = head;
5      while (fast != null && fast.next != null) {
6          slow = slow.next;       // 1 step
7          fast = fast.next.next;  // 2 steps
8      }
▶      return slow.data;
10 }
```

VARIABLES

`head = node(10)` `slow = node(1)` `fast = node(5)`

Copying a List




```
SinglyLinkedList copy() {
    SinglyLinkedList newList =
        new SinglyLinkedList();
    Node temp = head;
    while (temp != null) {
        newList.addLast(temp.data);
        temp = temp.next;
    }
    return newList;
}
```

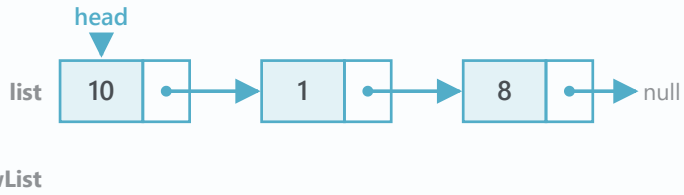
Deep copy

New nodes are created; writing `newList.head = head` would only give a second reference to the same list.

Cost

If `addLast` walks to the end every time, the total is $O(n^2)$; with a tail pointer, $O(n)$.

Code Trace: `copy()`



Create a new, empty list. Writing `newList.head = head` would not be a copy, just a second reference to the same nodes.

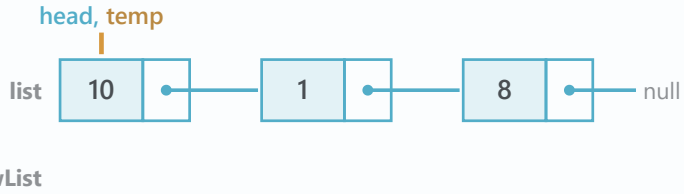
Step 1 / 13

```
1  SinglyLinkedList copy() {
▶      SinglyLinkedList newList =
▶          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = empty`

Code Trace: `copy()`



`temp = head` : walk the original list from the head.

Step 2 / 13

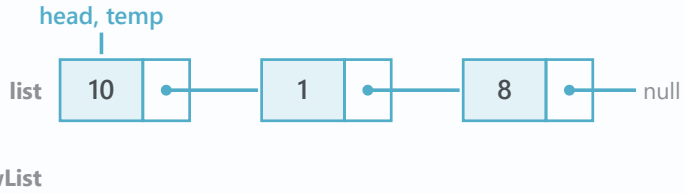
```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
▶      Node temp = head;
5      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = empty`

`temp = node(10)`

Code Trace: `copy()`



temp = node(10) is not null.

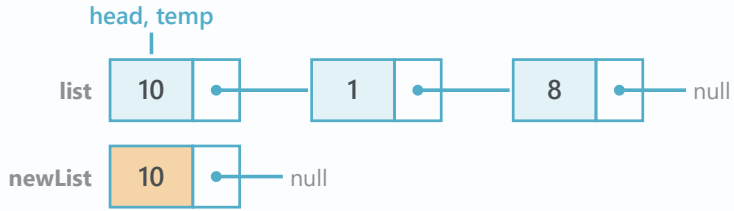
Step 3 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
▶      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = empty` `temp = node(10)`

Code Trace: `copy()`



`newList.addLast(10)` : a **new** node goes on the end of the copy.

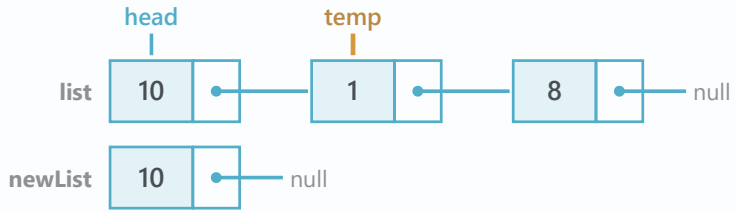
Step 4 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
▶          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = 10` `temp = node(10)`

Code Trace: `copy()`



`temp = temp.next` → node(1).

Step 5 / 13

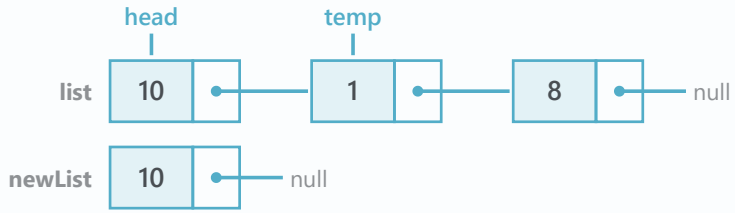
```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
6          newList.addLast(temp.data);
▶          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = 10`

`temp = node(1)`

Code Trace: `copy()`



temp = node(1) is not null.

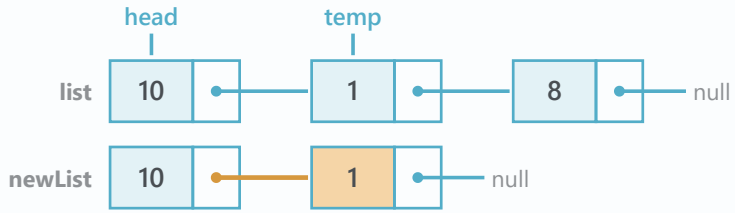
Step 6 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
▶      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = 10` `temp = node(1)`

Code Trace: `copy()`



`newList.addLast(1)` : a **new** node goes on the end of the copy.

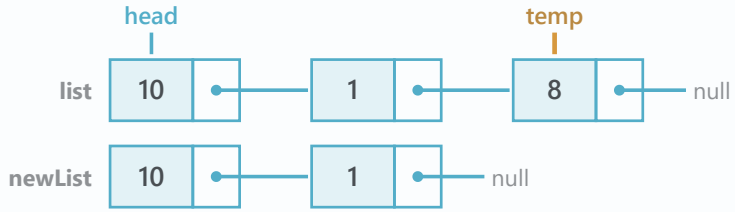
Step 7 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
▶          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = 10→1` `temp = node(1)`

Code Trace: `copy()`



`temp = temp.next` → node(8).

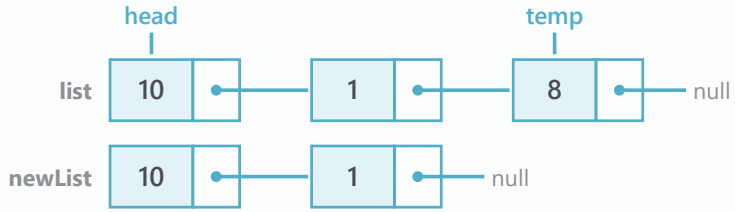
Step 8 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
6          newList.addLast(temp.data);
▶          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = 10→1` `temp = node(8)`

Code Trace: `copy()`



temp = node(8) is not null.

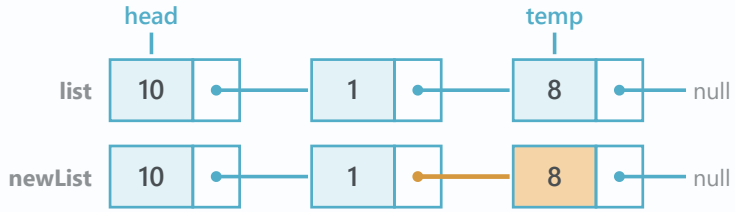
Step 9 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
▶      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)` `newList = 10→1` `temp = node(8)`

Code Trace: `copy()`



`newList.addLast(8)` : a **new** node goes on the end of the copy.

Step 10 / 13

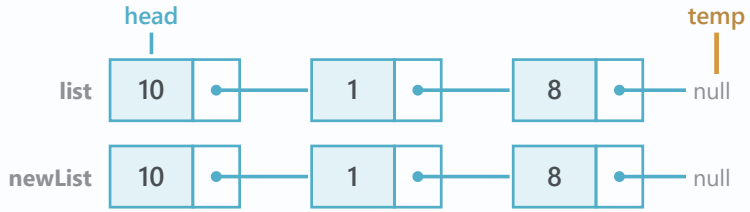
```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
▶          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)`

`newList = 10→1→8` `temp = node(8)`

Code Trace: `copy()`



`temp = temp.next` → null.

Step 11 / 13

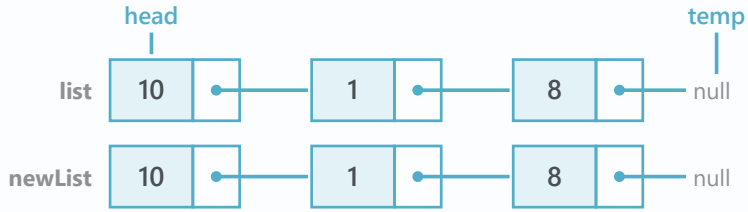
```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
6          newList.addLast(temp.data);
▶          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)`

`newList = 10→1→8` `temp = null`

Code Trace: `copy()`



temp == null → the loop ends.

Step 12 / 13

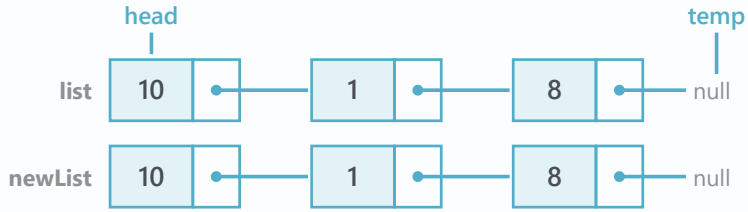
```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
▶      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
9      return newList;
10 }
```

VARIABLES

`head = node(10)`

`newList = 10→1→8` `temp = null`

Code Trace: `copy()`



`return newList` : the same values in different nodes (a deep copy). If addLast is O(n) the total is $O(n^2)$; with a tail pointer, **O(n)**.

Step 13 / 13

```
1  SinglyLinkedList copy() {
2      SinglyLinkedList newList =
3          new SinglyLinkedList();
4      Node temp = head;
5      while (temp != null) {
6          newList.addLast(temp.data);
7          temp = temp.next;
8      }
▶      return newList;
10 }
```

VARIABLES

`head = node(10)`

`newList = 10→1→8` `temp = null`

Removing Duplicates from a List

```
void removeDuplicates() {
    SinglyLinkedList seen =
        new SinglyLinkedList();
    Node prev = null, temp = head;
    while (temp != null) {
        int v = temp.data;
        if (seen.contains(v)) {
            prev.next =
                temp.next;   // skip
            if (temp == tail) tail = prev;  size--;
        } else {
            seen.addFirst(v);
            prev = temp;
        }
        temp = temp.next;
    }
}
```

Idea

Keep the values seen so far in a second list; skip any node that repeats one.

Cost

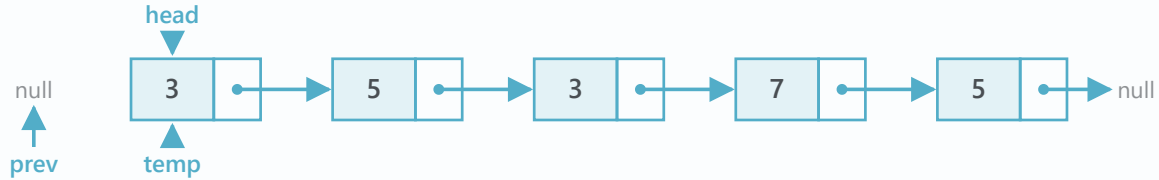
A search in the `seen` list for every node: $O(n^2)$. With a hash set, $O(n)$.

Note

The first node is never a duplicate, so no deletion happens while `prev` is null.

`contains(v)`: `search(v) != -1`. Example: 3 → 5 → 3 → 7 → 5 ⇒ 3 → 5 → 7

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



Start (earlier lines): `seen` is empty, `prev = null`, `temp = head`. temp = node(3) is not null.

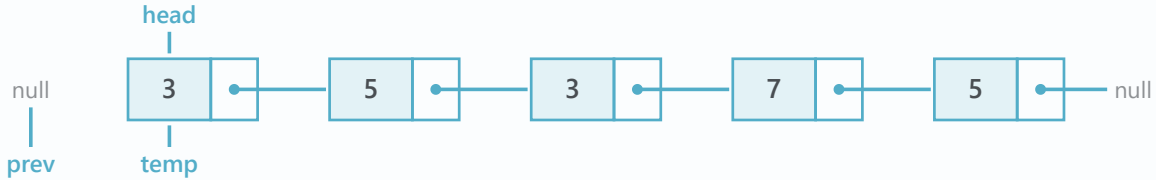
Step 1 / 15

```
▶ while (temp != null) {
2     int v = temp.data;
3     if (seen.contains(v)) {
4         prev.next =
5             temp.next;   // skip
6         if (temp == tail) tail = prev;  size--;
7     } else {
8         seen.addFirst(v);
9         prev = temp;
10    }
11    temp = temp.next;
12 }
```

VARIABLES

`head = node(3)` `seen = empty`
`prev = null` `temp = node(3)`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`v = 3` .

Step 2 / 15

```
1  while (temp != null) {
▶      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

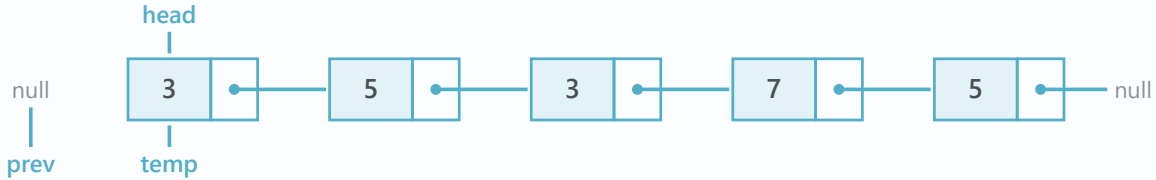
VARIABLES

`head = node(3)` `seen = empty`

`prev = null` `temp = node(3)`

`v = 3`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`seen.contains(3)` → false: 3 is new.

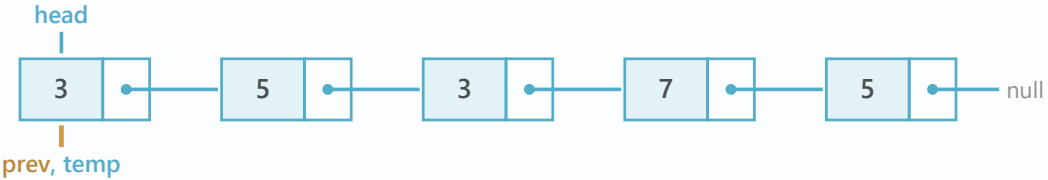
Step 3 / 15

```
1  while (temp != null) {
2      int v = temp.data;
▶      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

- head = **node(3)**
- seen = **empty**
- prev = **null**
- temp = **node(3)**
- v = **3**

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`seen.addFirst(3)` ; `prev = temp` : the node stays; prev moves to it.

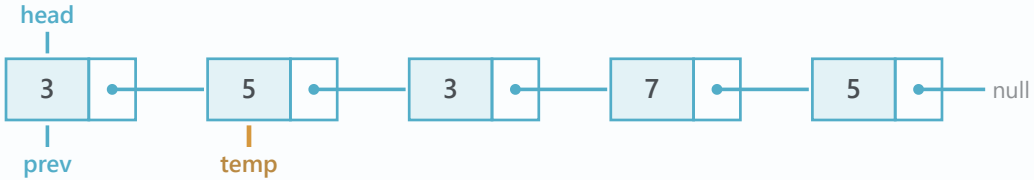
Step 4 / 15

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
▶          seen.addFirst(v);
▶          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

head = **node(3)** | seen = **3** | prev = **node(3)** | temp = **node(3)** | v = **3**

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`temp = temp.next` → node(5).

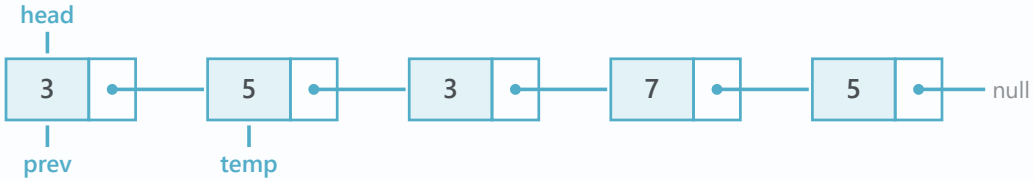
Step 5 / 15

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
▶      temp = temp.next;
12 }
```

VARIABLES

`head = node(3)` `seen = 3`
`prev = node(3)` `temp = node(5)`
`v = 3`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



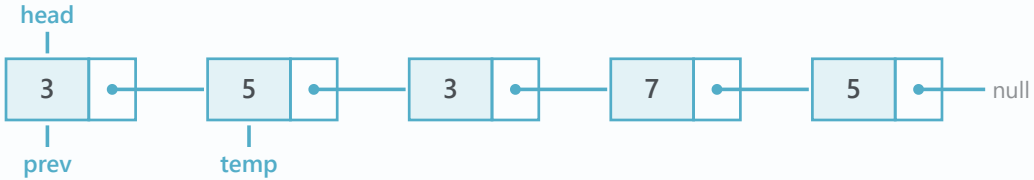
temp = node(5) is not null.


```
▶ while (temp != null) {
2     int v = temp.data;
3     if (seen.contains(v)) {
4         prev.next =
5             temp.next;   // skip
6         if (temp == tail) tail = prev;  size--;
7     } else {
8         seen.addFirst(v);
9         prev = temp;
10    }
11    temp = temp.next;
12 }
```

VARIABLES

head = **node(3)** | seen = **3** | prev = **node(3)** | temp = **node(5)**

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`v = 5` .

Step 7 / 15

```
1  while (temp != null) {
▶      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

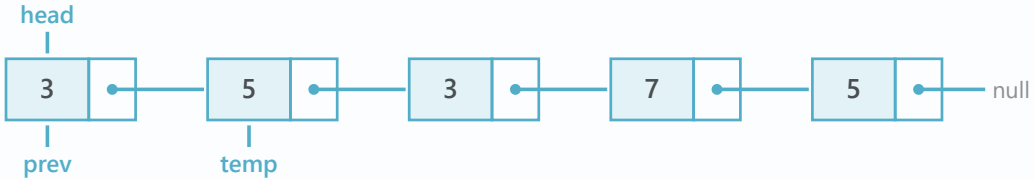
VARIABLES

`head = node(3)` `seen = 3`

`prev = node(3)` `temp = node(5)`

`v = 5`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`seen.contains(5)` → false: 5 is new.

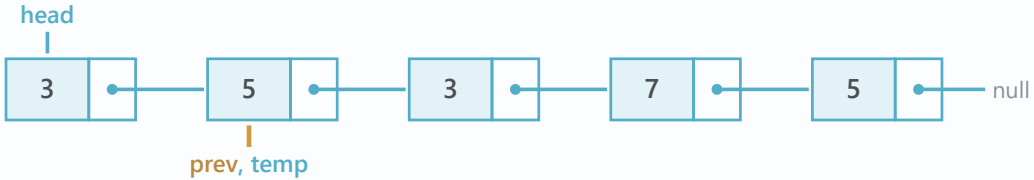
Step 8 / 15

```
1  while (temp != null) {
2      int v = temp.data;
▶      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

`head = node(3)` `seen = 3`
`prev = node(3)` `temp = node(5)`
`v = 5`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`seen.addFirst(5)` ; `prev = temp` : the node stays; prev moves to it.

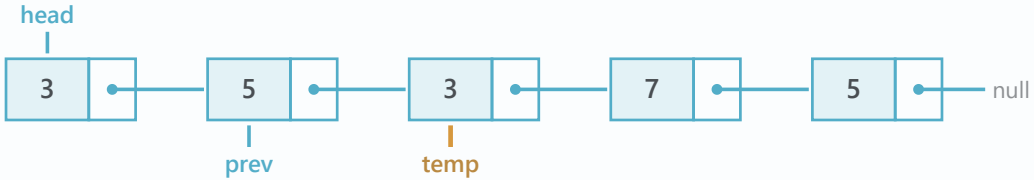
Step 9 / 15

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
▶          seen.addFirst(v);
▶          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

`head = node(3)` `seen = 5→3`
`prev = node(5)` `temp = node(5)`
`v = 5`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`temp = temp.next` → node(3).

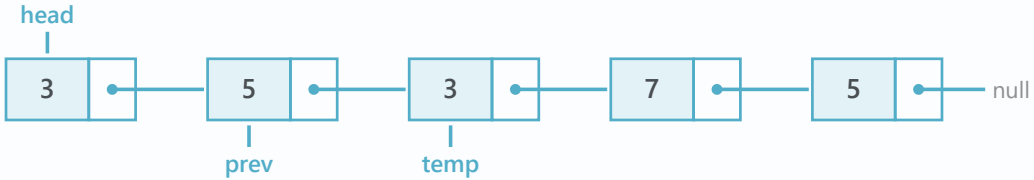
Step 10 / 15

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
▶      temp = temp.next;
12 }
```

VARIABLES

- `head = node(3)`
- `seen = 5→3`
- `prev = node(5)`
- `temp = node(3)`
- `v = 5`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



temp = node(3) is not null.

Step 11 / 15

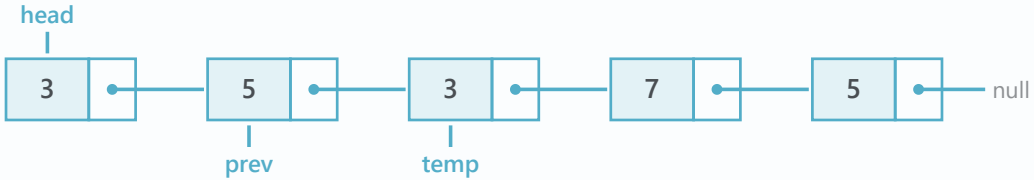
```
▶ while (temp != null) {
2     int v = temp.data;
3     if (seen.contains(v)) {
4         prev.next =
5             temp.next;   // skip
6         if (temp == tail) tail = prev;  size--;
7     } else {
8         seen.addFirst(v);
9         prev = temp;
10    }
11    temp = temp.next;
12 }
```

VARIABLES

`head` = **node(3)** `seen` = **5→3**

`prev` = **node(5)** `temp` = **node(3)**

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`v = 3` .

Step 12 / 15

```
1  while (temp != null) {
▶      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

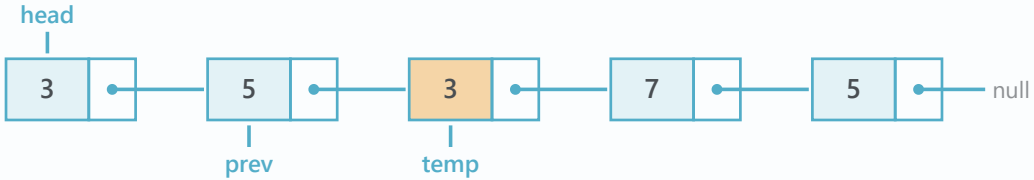
VARIABLES

`head = node(3)` `seen = 5→3`

`prev = node(5)` `temp = node(3)`

`v = 3`

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`seen.contains(3)` → **true**: 3 was seen before; this node is a duplicate.

Step 13 / 15

```
1  while (temp != null) {
2      int v = temp.data;
▶      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

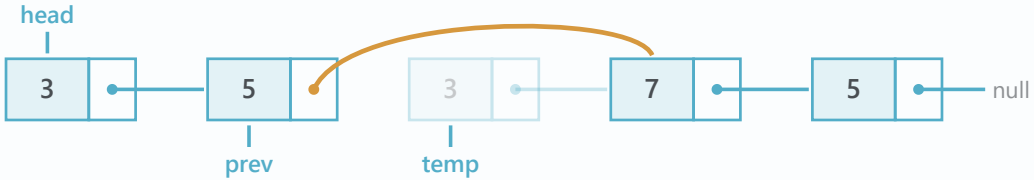
VARIABLES

`head` = **node(3)** `seen` = **5→3**

`prev` = **node(5)** `temp` = **node(3)**

`v` = **3**

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`prev.next = temp.next` : 5 → node(7); the duplicate 3 is skipped. `prev` stays put.

Step 14 / 15

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
▶          prev.next =
▶              temp.next;   // skip
▶          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

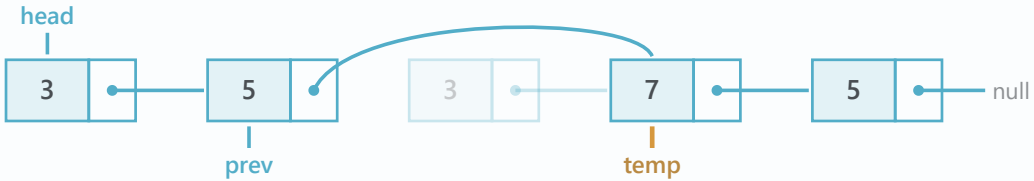
VARIABLES

`head` = **node(3)** `seen` = **5→3**

`prev` = **node(5)** `temp` = **node(3)**

`v` = **3**

Code Trace: Removing Duplicates (3, 5, 3, 7, 5)



`temp = temp.next` → node(7).

Step 15 / 15

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
▶      temp = temp.next;
12 }
```

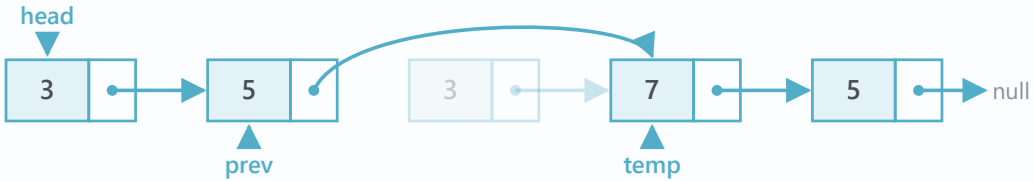
VARIABLES

`head = node(3)` `seen = 5→3`

`prev = node(5)` `temp = node(7)`

`v = 3`

Code Trace: Removing Duplicates (continued)



temp = node(7) is not null.

Step 1 / 11

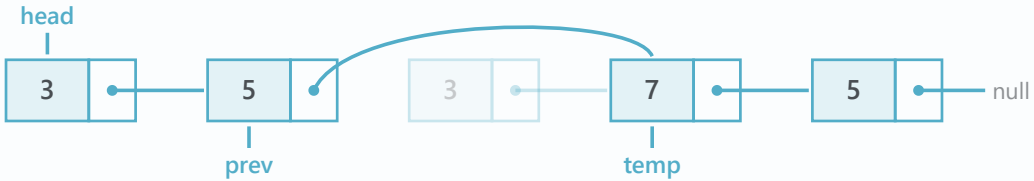
```
▶ while (temp != null) {
2     int v = temp.data;
3     if (seen.contains(v)) {
4         prev.next =
5             temp.next;   // skip
6         if (temp == tail) tail = prev;  size--;
7     } else {
8         seen.addFirst(v);
9         prev = temp;
10    }
11    temp = temp.next;
12 }
```

VARIABLES

head = **node(3)** seen = **5→3**

prev = **node(5)** temp = **node(7)**

Code Trace: Removing Duplicates (continued)



`v = 7` .

Step 2 / 11

```
1  while (temp != null) {
▶      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

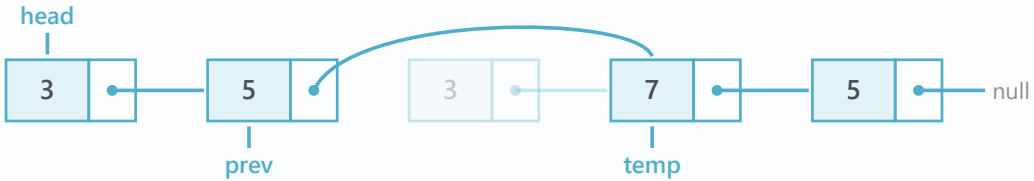
VARIABLES

`head = node(3)` `seen = 5→3`

`prev = node(5)` `temp = node(7)`

`v = 7`

Code Trace: Removing Duplicates (continued)



`seen.contains(7)` → false: 7 is new.

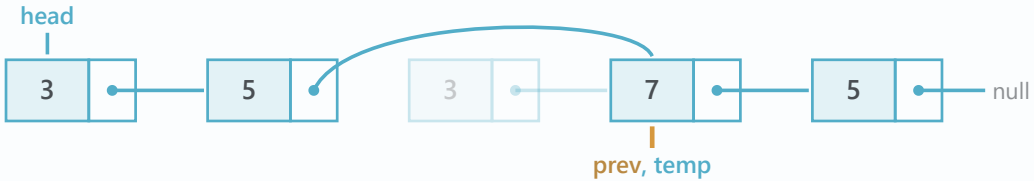
Step 3 / 11

```
1  while (temp != null) {
2      int v = temp.data;
▶      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

- `head` = **node(3)**
- `seen` = **5→3**
- `prev` = **node(5)**
- `temp` = **node(7)**
- `v` = **7**

Code Trace: Removing Duplicates (continued)



`seen.addFirst(7)` ; `prev = temp` : the node stays; prev moves to it.

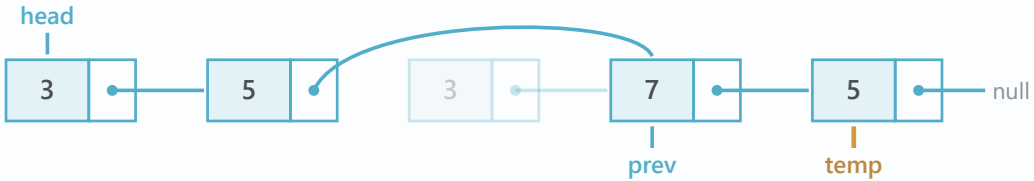
Step 4 / 11

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
▶          seen.addFirst(v);
▶          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

`head = node(3)` `seen = 7→5→3`
`prev = node(7)` `temp = node(7)`
`v = 7`

Code Trace: Removing Duplicates (continued)



`temp = temp.next` → node(5).

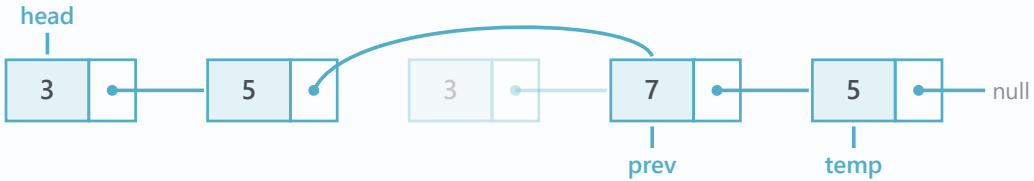
Step 5 / 11

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
▶      temp = temp.next;
12 }
```

VARIABLES

- `head` = **node(3)**
- `seen` = **7→5→3**
- `prev` = **node(7)**
- `temp` = **node(5)**
- `v` = **7**

Code Trace: Removing Duplicates (continued)



temp = node(5) is not null.

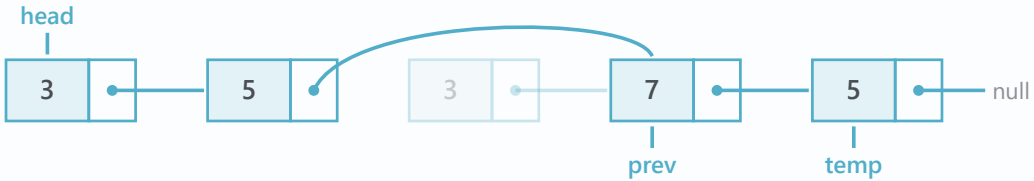
Step 6 / 11

```
▶ while (temp != null) {
2     int v = temp.data;
3     if (seen.contains(v)) {
4         prev.next =
5             temp.next;   // skip
6         if (temp == tail) tail = prev;  size--;
7     } else {
8         seen.addFirst(v);
9         prev = temp;
10    }
11    temp = temp.next;
12 }
```

VARIABLES

- head = **node(3)**
- seen = **7→5→3**
- prev = **node(7)**
- temp = **node(5)**

Code Trace: Removing Duplicates (continued)



`v = 5` .

Step 7 / 11

```
1  while (temp != null) {
▶      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

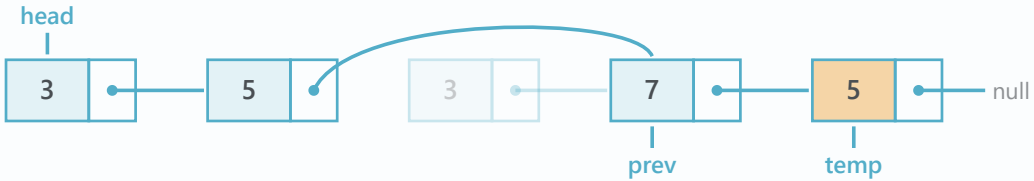
VARIABLES

`head = node(3)` `seen = 7→5→3`

`prev = node(7)` `temp = node(5)`

`v = 5`

Code Trace: Removing Duplicates (continued)



`seen.contains(5)` → **true**: 5 was seen before; this node is a duplicate.

Step 8 / 11

```
1  while (temp != null) {
2      int v = temp.data;
▶      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

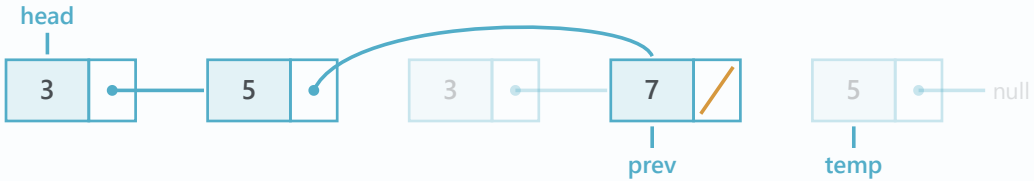
VARIABLES

`head` = **node(3)** `seen` = **7→5→3**

`prev` = **node(7)** `temp` = **node(5)**

`v` = **5**

Code Trace: Removing Duplicates (continued)



`prev.next = temp.next` : 7 → null; the duplicate 5 is skipped. `prev` stays put.

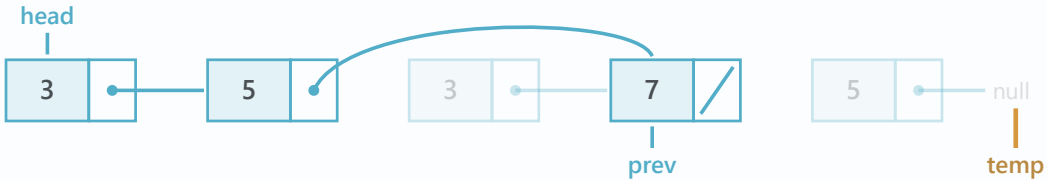
Step 9 / 11

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
▶          prev.next =
▶              temp.next;   // skip
▶          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
11     temp = temp.next;
12 }
```

VARIABLES

- `head` = **node(3)**
- `seen` = **7→5→3**
- `prev` = **node(7)**
- `temp` = **node(5)**
- `v` = **5**

Code Trace: Removing Duplicates (continued)



`temp = temp.next` → null.

Step 10 / 11

```
1  while (temp != null) {
2      int v = temp.data;
3      if (seen.contains(v)) {
4          prev.next =
5              temp.next;   // skip
6          if (temp == tail) tail = prev;  size--;
7      } else {
8          seen.addFirst(v);
9          prev = temp;
10     }
▶      temp = temp.next;
12 }
```

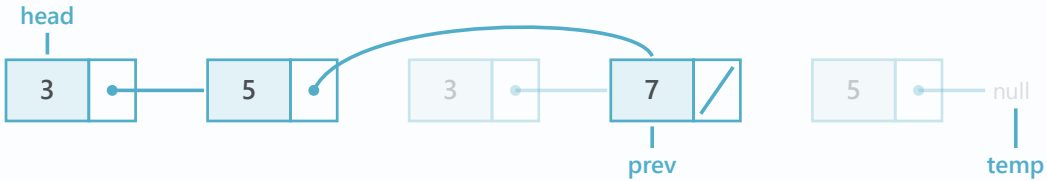
VARIABLES

`head = node(3)` `seen = 7→5→3`

`prev = node(7)` `temp = null`

`v = 5`

Code Trace: Removing Duplicates (continued)



temp == null → the loop ends. Result: 3 → 5 → 7 → null. A search in seen for every node: **O(n²)**; with a hash set, O(n).

Step 11 / 11

```
▶ while (temp != null) {
2     int v = temp.data;
3     if (seen.contains(v)) {
4         prev.next =
5             temp.next;   // skip
6         if (temp == tail) tail = prev;  size--;
7     } else {
8         seen.addFirst(v);
9         prev = temp;
10    }
11    temp = temp.next;
12 }
```

VARIABLES

head = **node(3)**	seen = **7→5→3**
prev = **node(7)**	temp = **null**

Merging Two Sorted Lists

```
SinglyLinkedList merge(
        SinglyLinkedList l1,
        SinglyLinkedList l2) {
    SinglyLinkedList result =
        new SinglyLinkedList();
    Node p1 = l1.head, p2 = l2.head;
    while (p1 != null && p2 != null) {
        if (p1.data <= p2.data) {
            result.addLast(p1.data);
            p1 = p1.next;
        } else {
            result.addLast(p2.data);
            p2 = p2.next;
        }
    }
    // append the leftovers
    while (p1 != null) {
        result.addLast(p1.data);
        p1 = p1.next;
    }
    while (p2 != null) {
        result.addLast(p2.data);
        p2 = p2.next;
    }
    return result;
}
```

$1 \to 4 \to 7$ and $2 \to 3 \to 8$

$\Rightarrow \mathbf{1 \to 2 \to 3 \to 4 \to 7 \to 8}$

If `addLast` is O(1) (tail pointer), the total is **O(n + m)** (n, m: list lengths); if each insertion walks to the end, $O((n + m)^2)$.

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



result

`result` is a new, empty list. `p1` and `p2` start at the heads of the two sorted lists.

Step 1 / 17

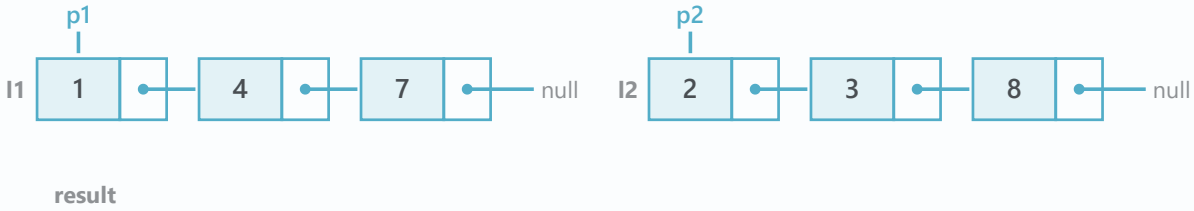
```
▶ Node p1 = l1.head, p2 = l2.head;
2 while (p1 != null && p2 != null) {
3     if (p1.data <= p2.data) {
4         result.addLast(p1.data);
5         p1 = p1.next;
6     } else {
7         result.addLast(p2.data);
8         p2 = p2.next;
9     }
10 }
```

VARIABLES

`p1 = node(1)` `p2 = node(2)`

`result = empty`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Neither list is exhausted → compare.

Step 2 / 17

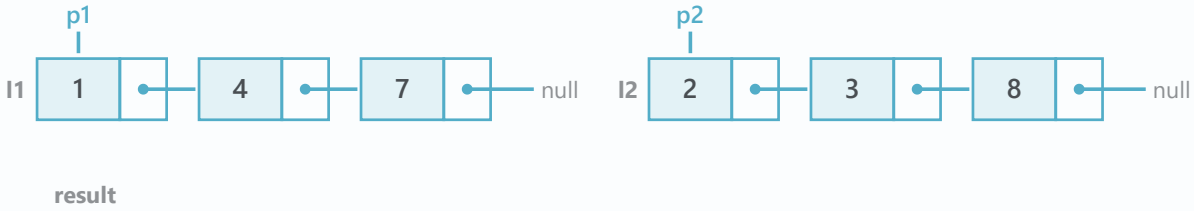
```
1  Node p1 = l1.head, p2 = l2.head;
▶  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(1)` `p2 = node(2)`

`result = empty`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



1 ≤ 2 → **true**: l1 has the smaller value.

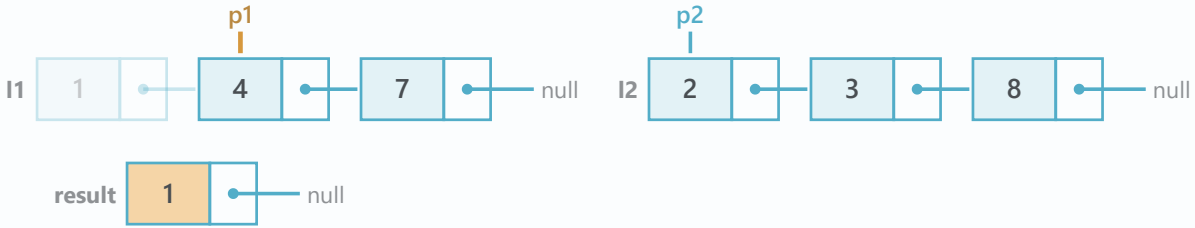
Step 3 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
▶      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(1)` `p2 = node(2)` `result = empty`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Append 1; `p1` advances.

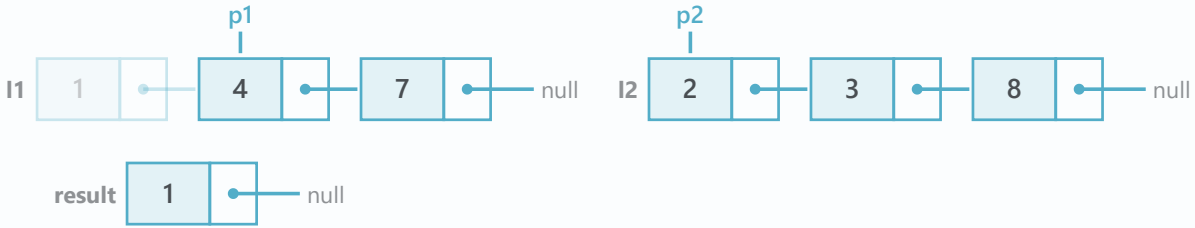
Step 4 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
▶          result.addLast(p1.data);
▶          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(2)` `result = 1`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Neither list is exhausted → compare.

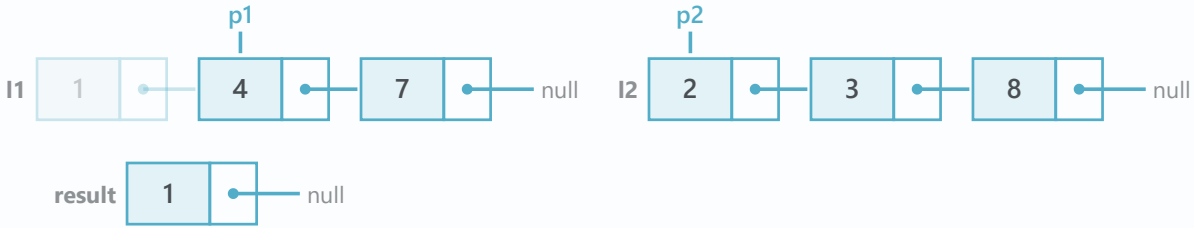
Step 5 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
▶  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(2)` `result = 1`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



4 ≤ 2 → **false**: l2 has the smaller value.

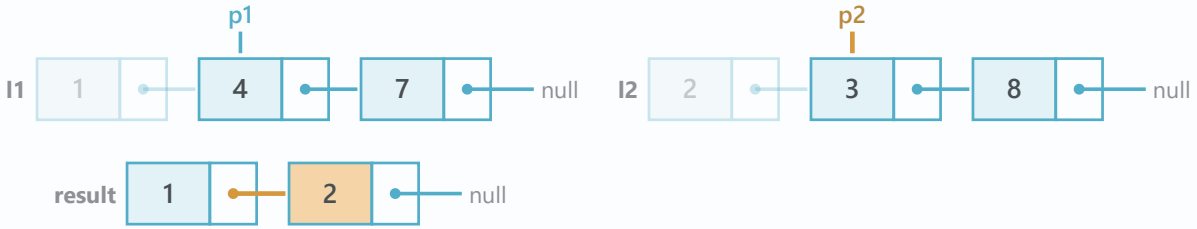
Step 6 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
▶      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(2)` `result = 1`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Append 2; `p2` advances.

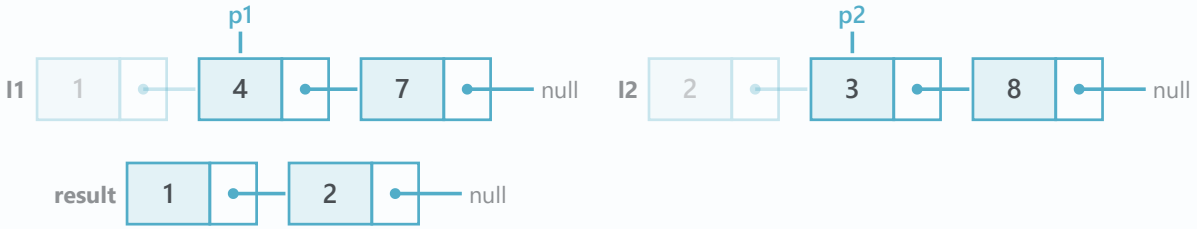
Step 7 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
▶          result.addLast(p2.data);
▶          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(3)` `result = 1→2`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Neither list is exhausted → compare.

Step 8 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
▶  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(3)` `result = 1→2`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



4 ≤ 3 → **false**: l2 has the smaller value.

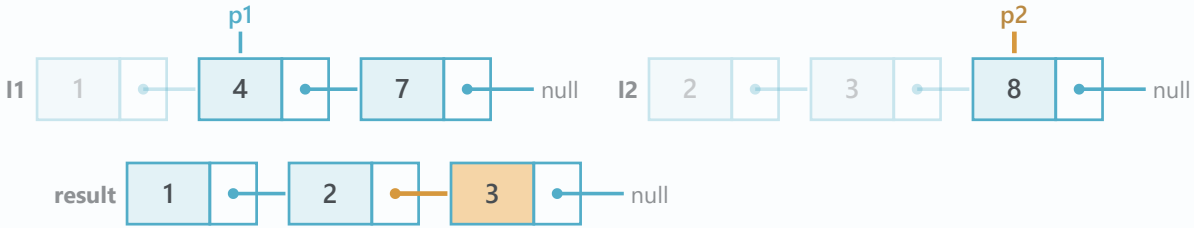
Step 9 / 17

```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
▶      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(3)` `result = 1→2`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Append 3; `p2` advances.

Step 10 / 17

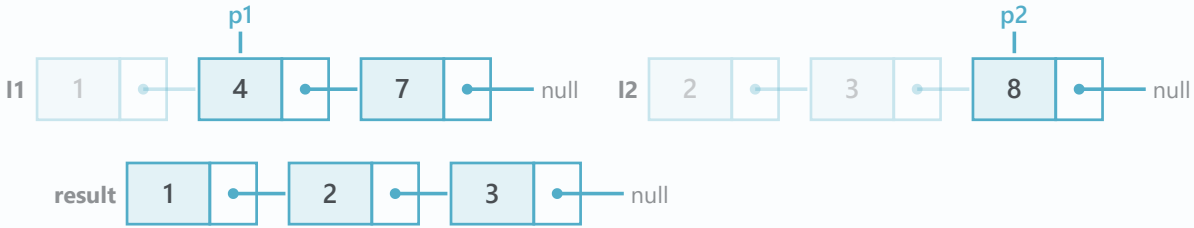
```
1 Node p1 = l1.head, p2 = l2.head;
2 while (p1 != null && p2 != null) {
3     if (p1.data <= p2.data) {
4         result.addLast(p1.data);
5         p1 = p1.next;
6     } else {
▶         result.addLast(p2.data);
▶         p2 = p2.next;
9     }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(8)`

`result = 1→2→3`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Neither list is exhausted → compare.

Step 11 / 17

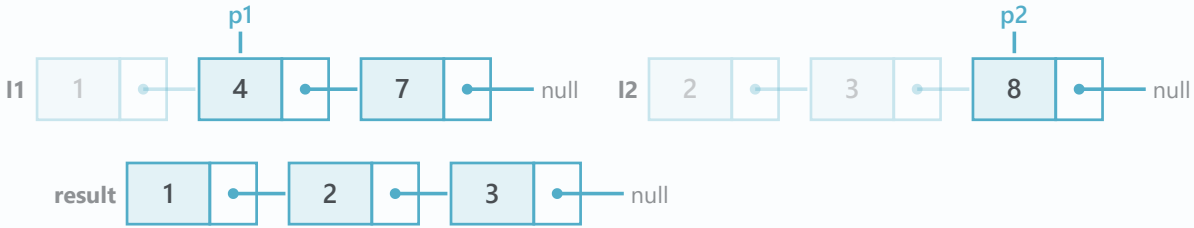
```
1  Node p1 = l1.head, p2 = l2.head;
▶  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(8)`

`result = 1→2→3`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



4 ≤ 8 → **true**: l1 has the smaller value.

Step 12 / 17

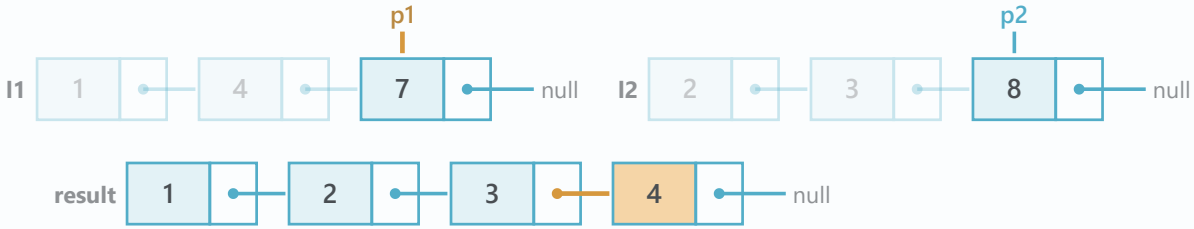
```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
▶      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(4)` `p2 = node(8)`

`result = 1→2→3`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Append 4; `p1` advances.

Step 13 / 17

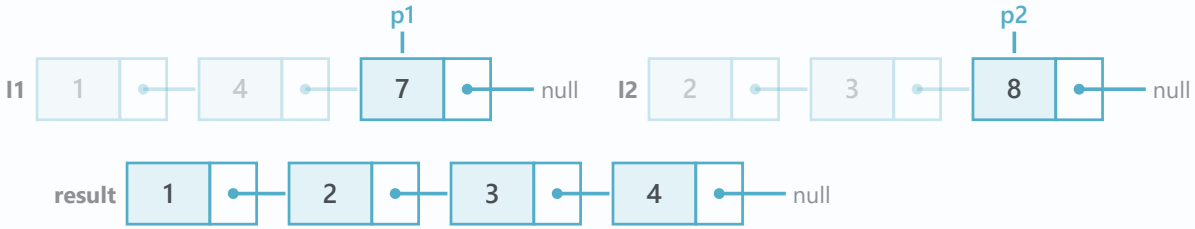
```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
▶          result.addLast(p1.data);
▶          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(7)` `p2 = node(8)`

`result = 1→2→3→4`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Neither list is exhausted → compare.

Step 14 / 17

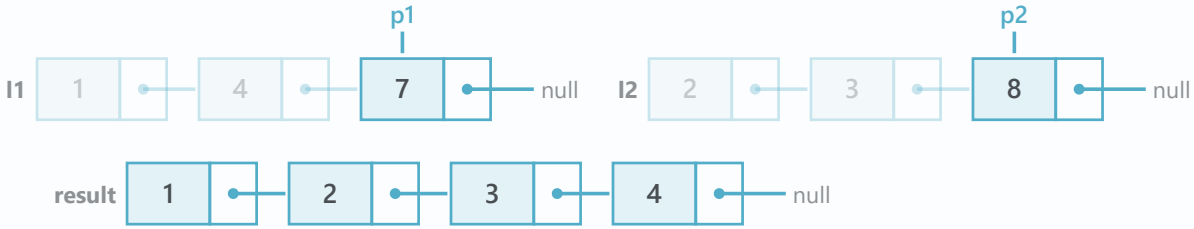
```
1  Node p1 = l1.head, p2 = l2.head;
▶ while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(7)` `p2 = node(8)`

`result = 1→2→3→4`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



7 ≤ 8 → **true**: l1 has the smaller value.

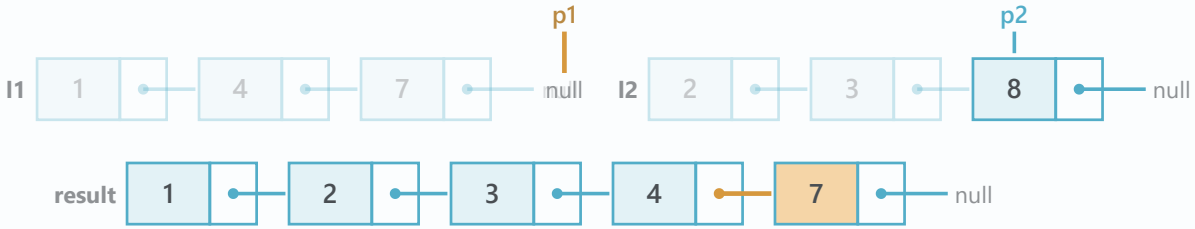

```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
▶      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = node(7)` `p2 = node(8)`

`result = 1→2→3→4`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



Append 7; `p1` advances.

Step 16 / 17

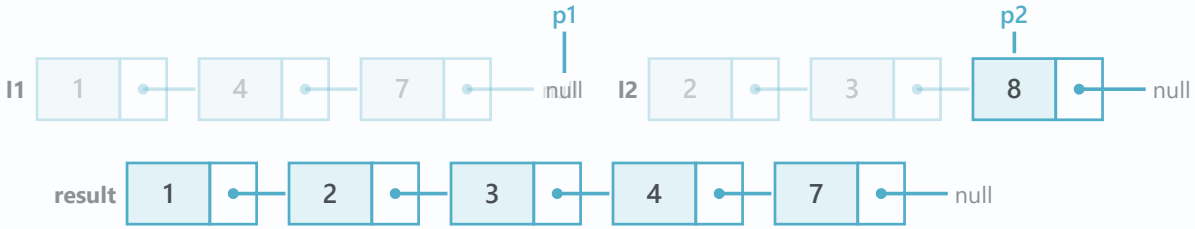
```
1  Node p1 = l1.head, p2 = l2.head;
2  while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
▶          result.addLast(p1.data);
▶          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = null` `p2 = node(8)`

`result = 1→2→3→4→7`

Code Trace: `merge` (1, 4, 7 + 2, 3, 8)



`p1 == null` : l1 is exhausted → the main loop ends. What is left in l2 (8) is already sorted.

Step 17 / 17

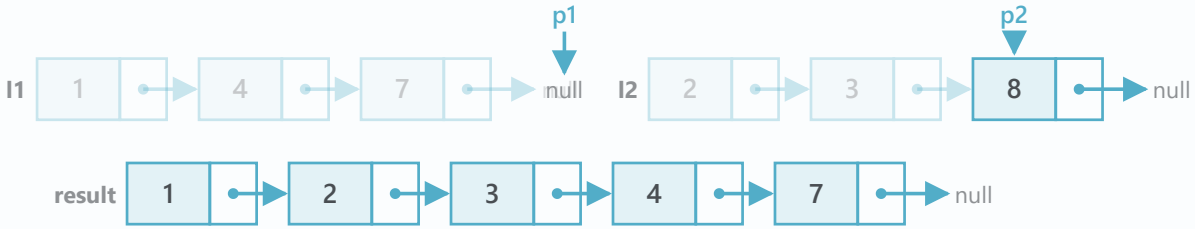
```
1  Node p1 = l1.head, p2 = l2.head;
▶ while (p1 != null && p2 != null) {
3      if (p1.data <= p2.data) {
4          result.addLast(p1.data);
5          p1 = p1.next;
6      } else {
7          result.addLast(p2.data);
8          p2 = p2.next;
9      }
10 }
```

VARIABLES

`p1 = null` `p2 = node(8)`

`result = 1→2→3→4→7`

Code Trace: `merge` (continued)



Leftovers: `p1 == null` → the first loop never runs.

Step 1 / 5

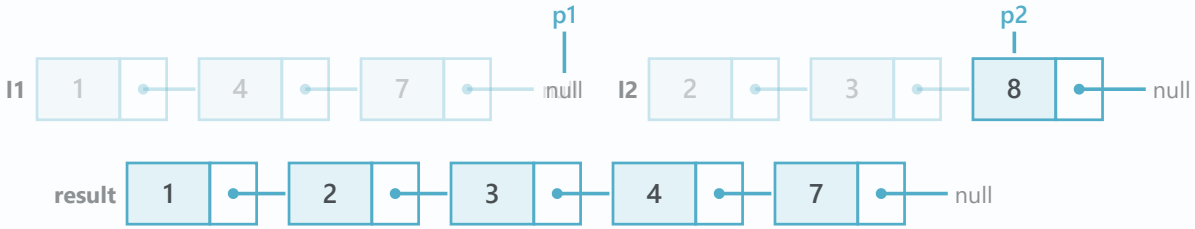
```
1  // append the leftovers
▶  while (p1 != null) {
3      result.addLast(p1.data);
4      p1 = p1.next;
5  }
6  while (p2 != null) {
7      result.addLast(p2.data);
8      p2 = p2.next;
9  }
10 return result;
```

VARIABLES

`p1 = null` `p2 = node(8)`

`result = 1→2→3→4→7`

Code Trace: `merge` (continued)



p2 = node(8) is not null.

Step 2 / 5

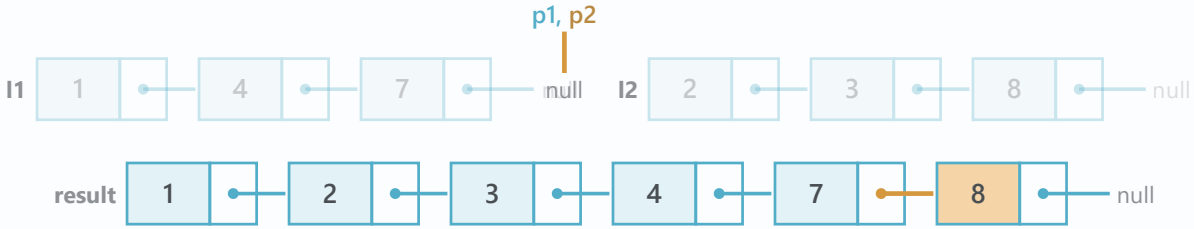
```
1  // append the leftovers
2  while (p1 != null) {
3      result.addLast(p1.data);
4      p1 = p1.next;
5  }
▶  while (p2 != null) {
7      result.addLast(p2.data);
8      p2 = p2.next;
9  }
10 return result;
```

VARIABLES

`p1 = null` `p2 = node(8)`

`result = 1→2→3→4→7`

Code Trace: `merge` (continued)



Append 8; `p2` advances.

Step 3 / 5

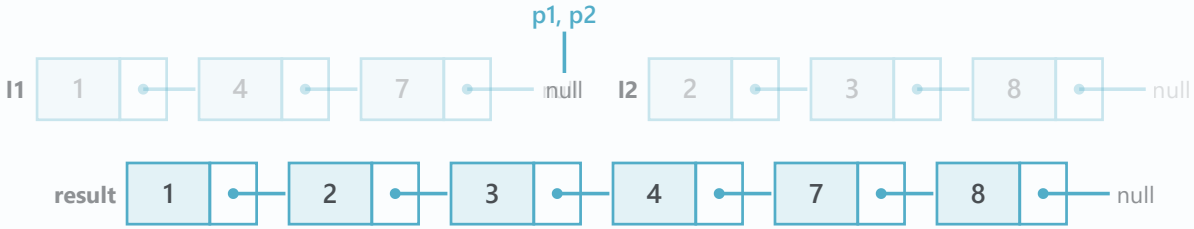
```
1 // append the leftovers
2 while (p1 != null) {
3     result.addLast(p1.data);
4     p1 = p1.next;
5 }
6 while (p2 != null) {
▶     result.addLast(p2.data);
▶     p2 = p2.next;
9 }
10 return result;
```

VARIABLES

`p1 = null` `p2 = null`

`result = 1→2→3→4→7→8`

Code Trace: `merge` (continued)



p2 == null → done.

Step 4 / 5

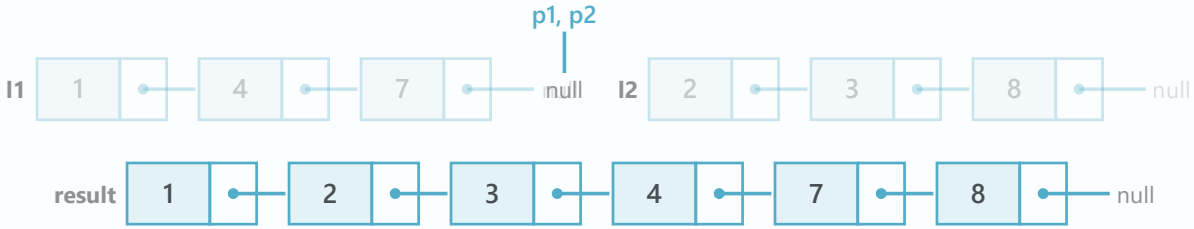
```
1  // append the leftovers
2  while (p1 != null) {
3      result.addLast(p1.data);
4      p1 = p1.next;
5  }
▶  while (p2 != null) {
7      result.addLast(p2.data);
8      p2 = p2.next;
9  }
10 return result;
```

VARIABLES

`p1 = null` `p2 = null`

`result = 1→2→3→4→7→8`

Code Trace: `merge` (continued)



`return result` : 1 → 2 → 3 → 4 → 7 → 8. If addLast is O(1) (tail pointer), the total is **O(n + m)**.

Step 5 / 5

```
1 // append the leftovers
2 while (p1 != null) {
3     result.addLast(p1.data);
4     p1 = p1.next;
5 }
6 while (p2 != null) {
7     result.addLast(p2.data);
8     p2 = p2.next;
9 }
▶ return result;
```

VARIABLES

`p1 = null` `p2 = null`

`result = 1→2→3→4→7→8`

Sorting a Singly Linked List

```
void sort() {
    Node current = head;
    while (current != null) {
        Node other = current.next;
        while (other != null) {
            if (current.data > other.data) {
                int temp = current.data;
                current.data = other.data;
                other.data = temp;
            }
            other = other.next;
        }
        current = current.next;
    }
}
```

Swap values

The nodes stay in place; only their `data` fields change.

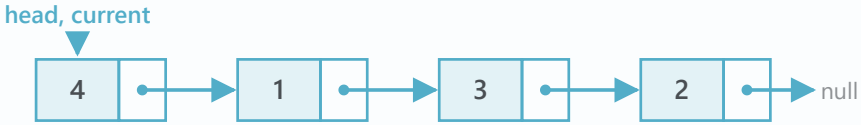
$O(n^2)$

Each node is compared with every node after it: $n(n-1)/2$.

Better

Merge sort runs in O(n log n) on a linked list and needs no extra array.

Code Trace: `sort()` (4 → 1 → 3 → 2)



List 4 → 1 → 3 → 2. `current` starts at the head; the nodes stay in place and only the `data` fields change.

Step 1 / 15

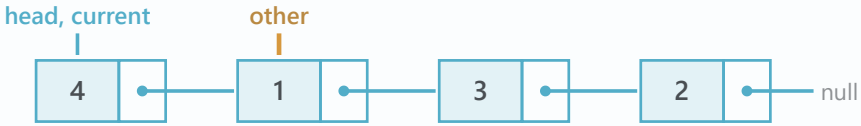
```
1  void sort() {
▶      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

`head = node(4)`

`current = node(4)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



`other = current.next` → node(1). current (4) is compared with each later node.

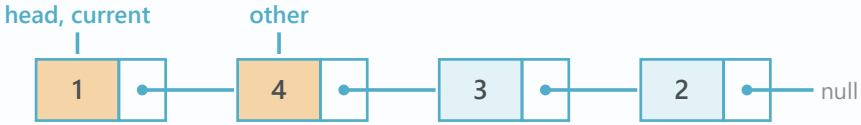
Step 2 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
▶          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

- `head = node(4)`
- `current = node(4)`
- `other = node(1)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



4 > 1 → swap the values (`temp` = 4). List: 1 → 4 → 3 → 2.

Step 3 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
▶              if (current.data > other.data) {
▶                  int temp = current.data;
▶                  current.data = other.data;
▶                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

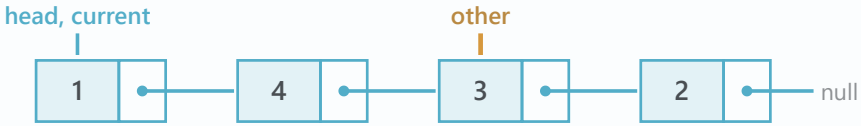
VARIABLES

`head` = **node(1)**

`current` = **node(1)**

`other` = **node(4)** `temp` = **4**

Code Trace: `sort()` (4 → 1 → 3 → 2)



1 > 3? No → no swap; `other` advances.

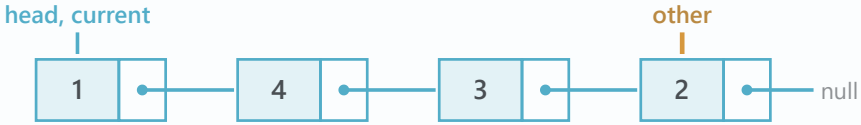
Step 4 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
▶              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

- `head = node(1)`
- `current = node(1)`
- `other = node(3)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



1 > 2? No → no swap; `other` advances.

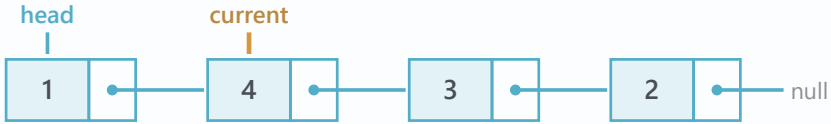
Step 5 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
▶              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

- `head = node(1)`
- `current = node(1)`
- `other = node(2)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



The inner loop is done: the smallest remaining value is now in current's node. `current` advances → node(4).

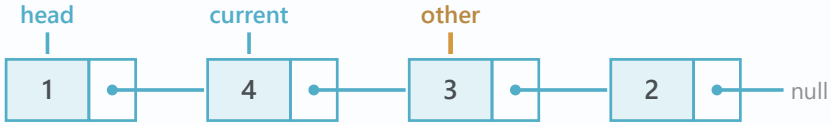

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
▶          current = current.next;
14     }
15 }
```

VARIABLES

`head = node(1)`

`current = node(4)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



`other = current.next` → node(3). current (4) is compared with each later node.

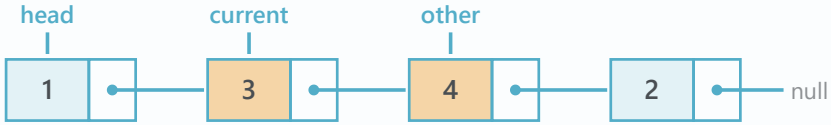
Step 7 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
▶          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

- `head = node(1)`
- `current = node(4)`
- `other = node(3)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



4 > 3 → swap the values (`temp` = 4). List: 1 → 3 → 4 → 2.

Step 8 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
▶              if (current.data > other.data) {
▶                  int temp = current.data;
▶                  current.data = other.data;
▶                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

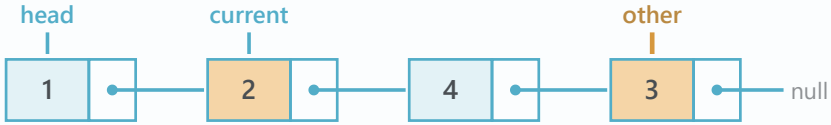
VARIABLES

`head = node(1)`

`current = node(3)`

`other = node(4)` `temp = 4`

Code Trace: `sort()` (4 → 1 → 3 → 2)



3 > 2 → swap the values (`temp` = 3). List: 1 → 2 → 4 → 3.

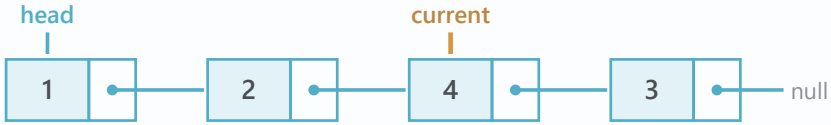
Step 9 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
▶              if (current.data > other.data) {
▶                  int temp = current.data;
▶                  current.data = other.data;
▶                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

- `head` = **node(1)**
- `current` = **node(2)**
- `other` = **node(3)**
- `temp` = **3**

Code Trace: `sort()` (4 → 1 → 3 → 2)



The inner loop is done: the smallest remaining value is now in current's node. `current` advances → node(4).

Step 10 / 15

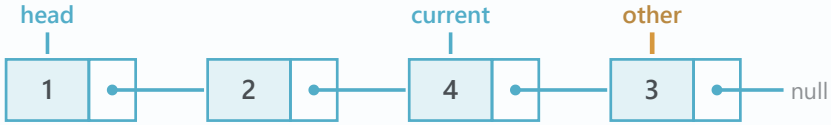
```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
▶          current = current.next;
14     }
15 }
```

VARIABLES

`head = node(1)`

`current = node(4)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



`other = current.next` → node(3). current (4) is compared with each later node.

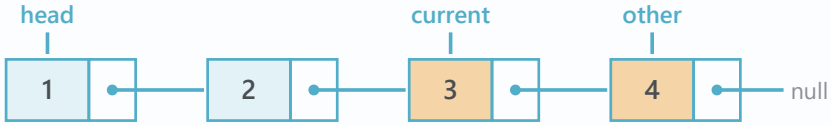
Step 11 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
▶          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

- `head = node(1)`
- `current = node(4)`
- `other = node(3)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



4 > 3 → swap the values (`temp` = 4). List: 1 → 2 → 3 → 4.

Step 12 / 15

```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
▶              if (current.data > other.data) {
▶                  int temp = current.data;
▶                  current.data = other.data;
▶                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

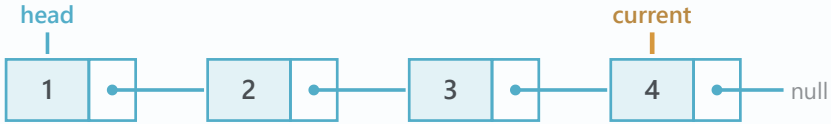
VARIABLES

`head` = **node(1)**

`current` = **node(3)**

`other` = **node(4)** `temp` = **4**

Code Trace: `sort()` (4 → 1 → 3 → 2)



The inner loop is done: the smallest remaining value is now in current's node. `current` advances → node(4).

Step 13 / 15

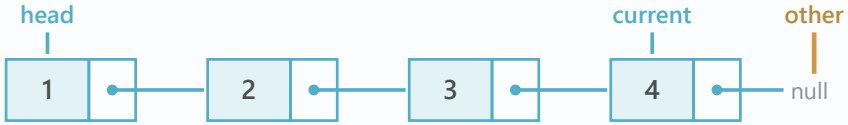
```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
▶          current = current.next;
14     }
15 }
```

VARIABLES

`head = node(1)`

`current = node(4)`

Code Trace: `sort()` (4 → 1 → 3 → 2)



`other = current.next` = null: there is nothing left to compare the last node with.

Step 14 / 15

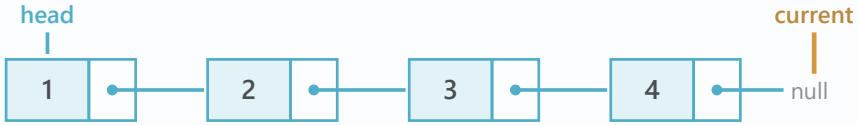
```
1  void sort() {
2      Node current = head;
3      while (current != null) {
▶          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
13         current = current.next;
14     }
15 }
```

VARIABLES

`head = node(1)`

`current = node(4)` `other = null`

Code Trace: `sort()` (4 → 1 → 3 → 2)



`current` = null → the outer loop ends. Result: 1 → 2 → 3 → 4; 6 comparisons = n(n−1)/2, 4 swaps.

Step 15 / 15

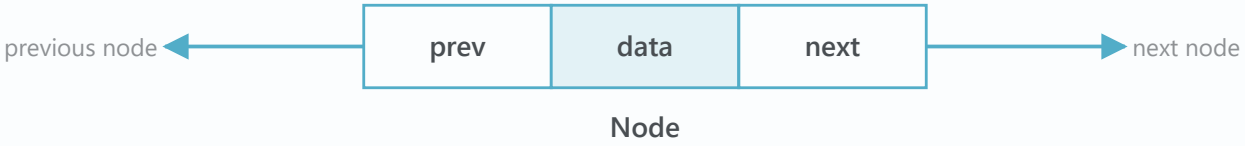
```
1  void sort() {
2      Node current = head;
3      while (current != null) {
4          Node other = current.next;
5          while (other != null) {
6              if (current.data > other.data) {
7                  int temp = current.data;
8                  current.data = other.data;
9                  other.data = temp;
10             }
11             other = other.next;
12         }
▶          current = current.next;
14     }
15 }
```

VARIABLES

`head = node(1)` `current = null`

Doubly Linked List

- A structure whose nodes are linked in **both directions**.
- Each node holds two pointers: one to the **previous** node and one to the **next**.
- The list can be walked either way; usually both a head (`head`) and a tail (`tail`) are kept.



Node

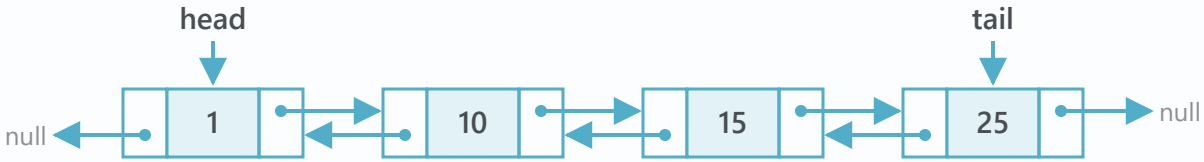
Gain

Backward traversal; deleting a known node or the last node in O(1).

Price

A second reference per node; every insertion and deletion must update both directions.

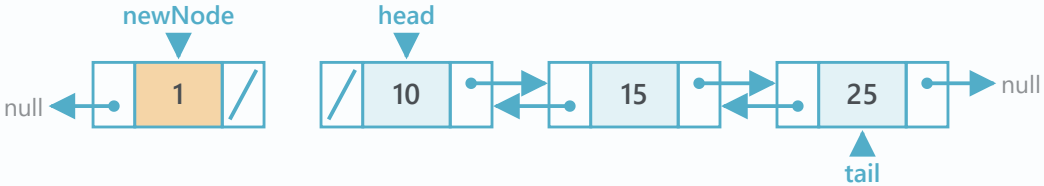
Doubly Linked List: Node and Structure




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

- Items can be reached going forward or backward.
- Inserting a node means updating pointers **in both directions**.
- Removing a node means updating the pointers of both the previous and the next node.
- A search can walk forward or backward.
- The list keeps the fields `Node head, tail;`.

Inserting at the Head of a Doubly Linked List — O(1)



`addFirst(1)` : the new node's `prev` and `next` fields are null (/).

Step 1 / 5

```
1  public void addFirst(int data) {
▶      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.next = head;
8          head.prev = newNode;
9          head = newNode;
10     }
11 }
```

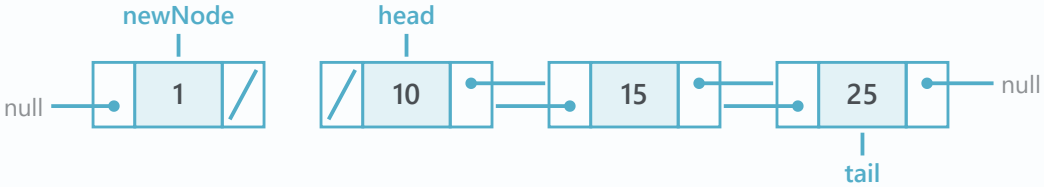
VARIABLES

`head = node(10)`

`tail = node(25)` `data = 1`

`newNode = node(1)`

Inserting at the Head of a Doubly Linked List — O(1)



`head` is not null → `else` branch.

Step 2 / 5

```
1  public void addFirst(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.next = head;
8          head.prev = newNode;
9          head = newNode;
10     }
11 }
```

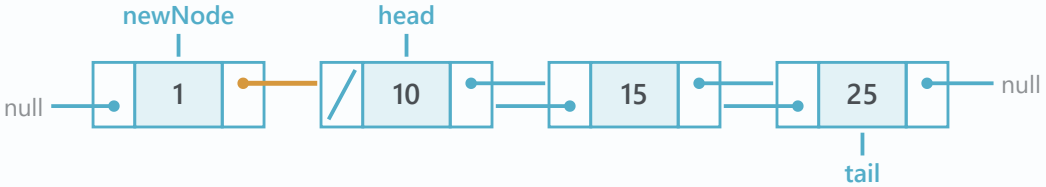
VARIABLES

`head = node(10)`

`tail = node(25)` `data = 1`

`newNode = node(1)`

Inserting at the Head of a Doubly Linked List — O(1)



`newNode.next = head` : forward link 1 → 10.

Step 3 / 5

```
1  public void addFirst(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
▶          newNode.next = head;
8          head.prev = newNode;
9          head = newNode;
10     }
11 }
```

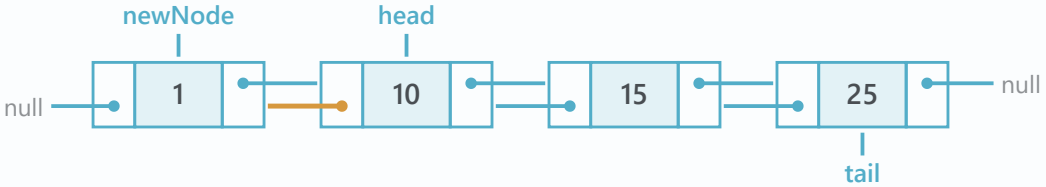
VARIABLES

`head = node(10)`

`tail = node(25)` `data = 1`

`newNode = node(1)`

Inserting at the Head of a Doubly Linked List — O(1)



`head.prev = newNode` : backward link 10 → 1 (10's null is gone).

Step 4 / 5

```
1  public void addFirst(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.next = head;
▶          head.prev = newNode;
9          head = newNode;
10     }
11 }
```

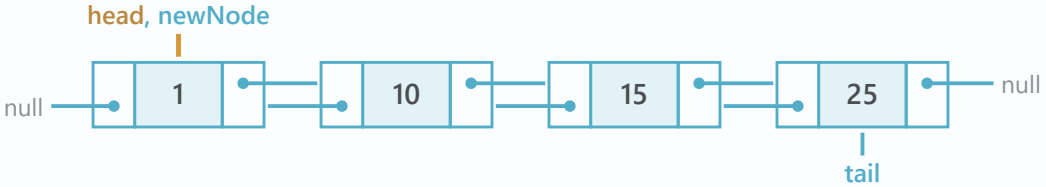
VARIABLES

`head = node(10)`

`tail = node(25)` `data = 1`

`newNode = node(1)`

Inserting at the Head of a Doubly Linked List — O(1)



`head = newNode` : 1 ⇄ 10 ⇄ 15 ⇄ 25. **O(1)**. Had the list been empty, head and tail would both become the new node.

Step 5 / 5

```
1  public void addFirst(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.next = head;
8          head.prev = newNode;
▶          head = newNode;
10     }
11 }
```

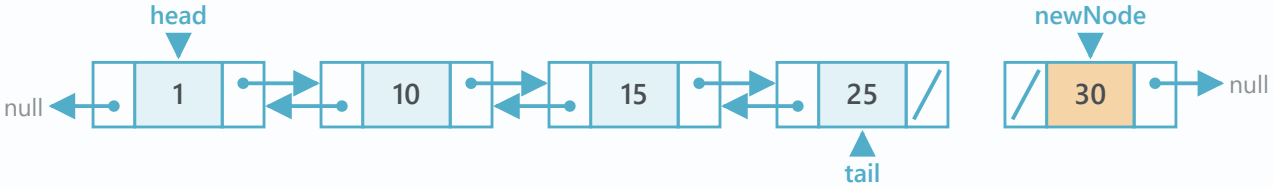
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 1`

`newNode = node(1)`

Inserting at the Tail of a Doubly Linked List — O(1)



`addLast(30)` : create the new node.

Step 1 / 5

```
1  public void addLast(int data) {
▶       Node newNode = new Node(data);
3       if (head == null) {
4           head = newNode;
5           tail = newNode;
6       } else {
7           newNode.prev = tail;
8           tail.next = newNode;
9           tail = newNode;
10      }
11 }
```

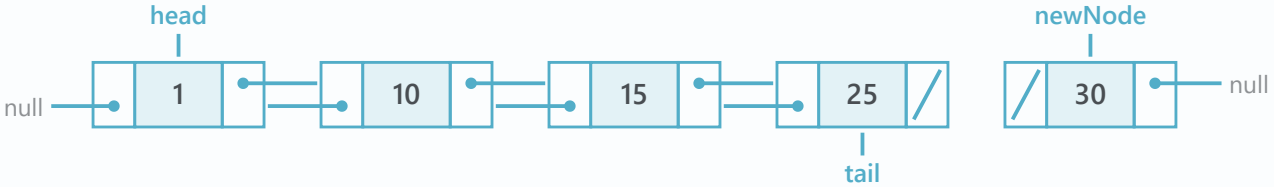
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 30`

`newNode = node(30)`

Inserting at the Tail of a Doubly Linked List — O(1)



The list is not empty → `else`. `tail` gives the last node directly.

Step 2 / 5

```
1  public void addLast(int data) {
2      Node newNode = new Node(data);
▶      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.prev = tail;
8          tail.next = newNode;
9          tail = newNode;
10     }
11 }
```

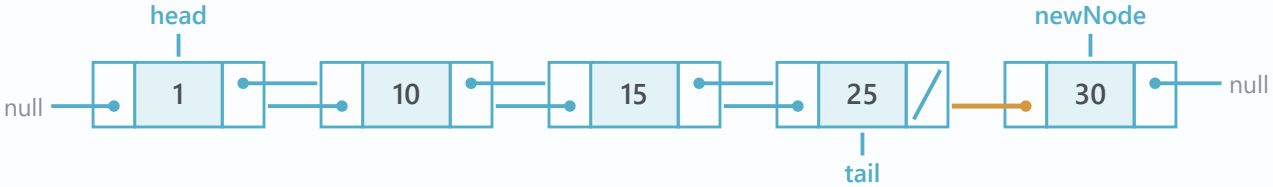
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 30`

`newNode = node(30)`

Inserting at the Tail of a Doubly Linked List — O(1)



`newNode.prev = tail` : backward link 30 → 25.

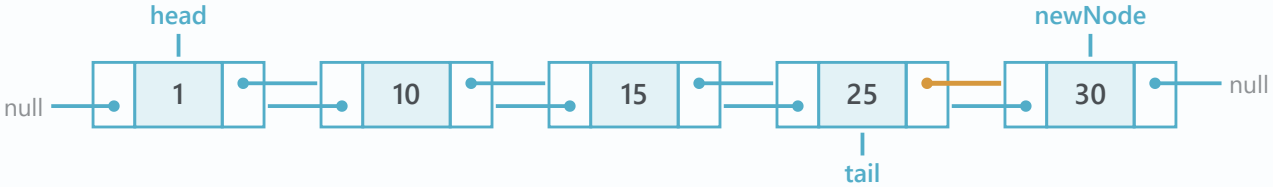
Step 3 / 5

```
1  public void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
▶          newNode.prev = tail;
8          tail.next = newNode;
9          tail = newNode;
10     }
11 }
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 30`
- `newNode = node(30)`

Inserting at the Tail of a Doubly Linked List — O(1)



`tail.next = newNode` : forward link 25 → 30.

Step 4 / 5

```
1  public void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.prev = tail;
▶          tail.next = newNode;
9          tail = newNode;
10     }
11 }
```

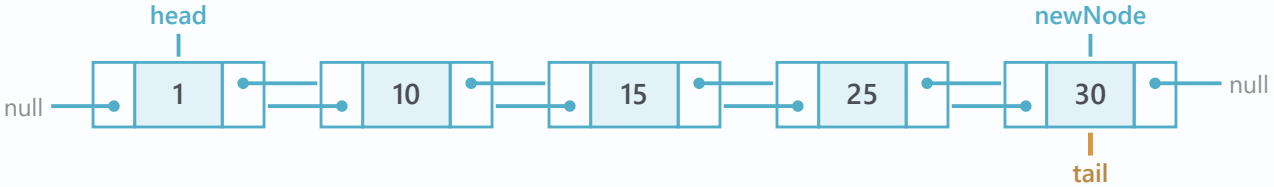
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 30`

`newNode = node(30)`

Inserting at the Tail of a Doubly Linked List — O(1)



`tail = newNode` : 1 ⇄ 10 ⇄ 15 ⇄ 25 ⇄ 30. No walk: **O(1)**.

Step 5 / 5

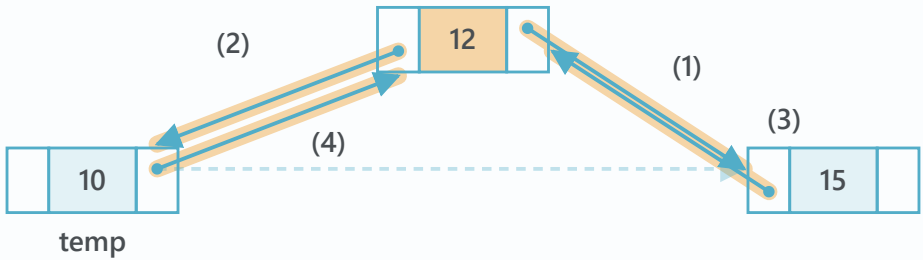
```
1  public void addLast(int data) {
2      Node newNode = new Node(data);
3      if (head == null) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          newNode.prev = tail;
8          tail.next = newNode;
▶          tail = newNode;
10     }
11 }
```

VARIABLES

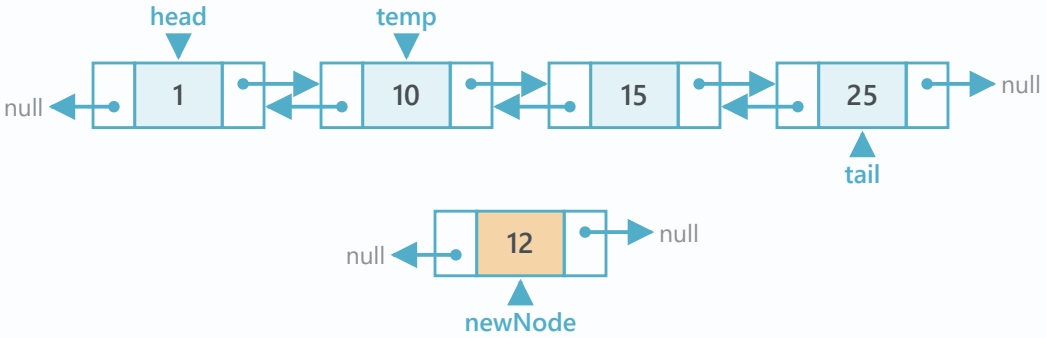
- `head = node(1)`
- `tail = node(30)` `data = 30`
- `newNode = node(30)`

Inserting at a Position in a Doubly Linked List

```
void addAt(int data, int index) {   // index starts at 0
    if (index == 0) { addFirst(data); return; }
    Node temp = head;
    for (int i = 0; i < index - 1 && temp != null; i++)
        temp = temp.next;
    if (temp == null) { System.out.println("Invalid position."); return; }
    if (temp == tail) { addLast(data); return; }   // updates tail too
    Node newNode = new Node(data);
    newNode.next = temp.next;     // (1)
    newNode.prev = temp;          // (2)
    temp.next.prev = newNode;     // (3)
    temp.next = newNode;          // (4)
}
```



Code Trace: `addAt(12, 2)` — the linking part



`addAt(12, 2)`: as in the singly linked version, temp has walked to the node at index − 1 = 1 (10). Create the new node.

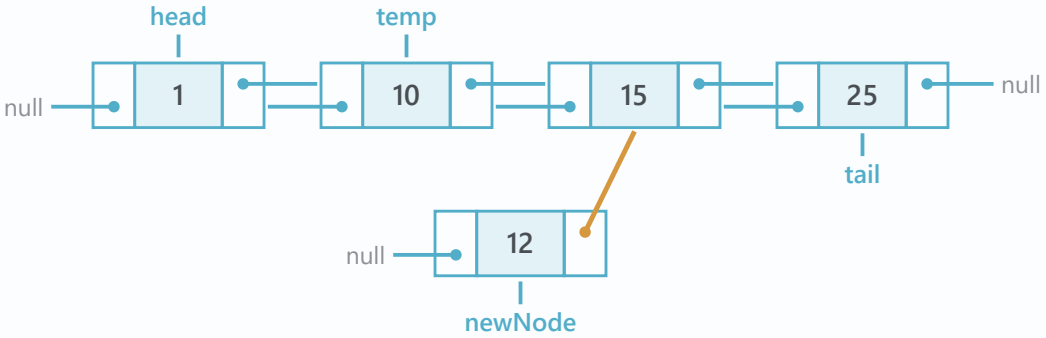
Step 1 / 6

```
▶ Node newNode = new Node(data);
2 newNode.next = temp.next;      // (1)
3 newNode.prev = temp;           // (2)
4 temp.next.prev = newNode;      // (3)
5 temp.next = newNode;           // (4)
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 12`
- `index = 2`
- `temp = node(10)`
- `newNode = node(12)`

Code Trace: `addAt(12, 2)` — the linking part



(1) `newNode.next = temp.next` : 12 → 15. The list has not changed yet.

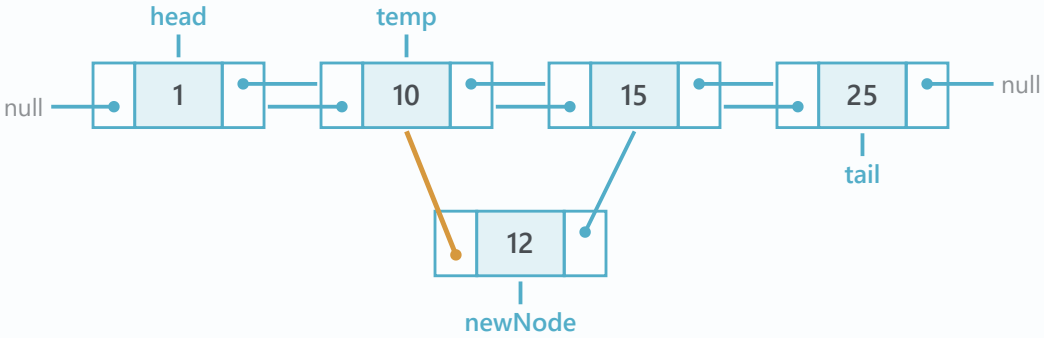
Step 2 / 6

```
1 Node newNode = new Node(data);
▶ newNode.next = temp.next;      // (1)
3 newNode.prev = temp;           // (2)
4 temp.next.prev = newNode;      // (3)
5 temp.next = newNode;           // (4)
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 12`
- `index = 2`
- `temp = node(10)`
- `newNode = node(12)`

Code Trace: `addAt(12, 2)` — the linking part



(2) `newNode.prev = temp` : backward link 12 → 10.

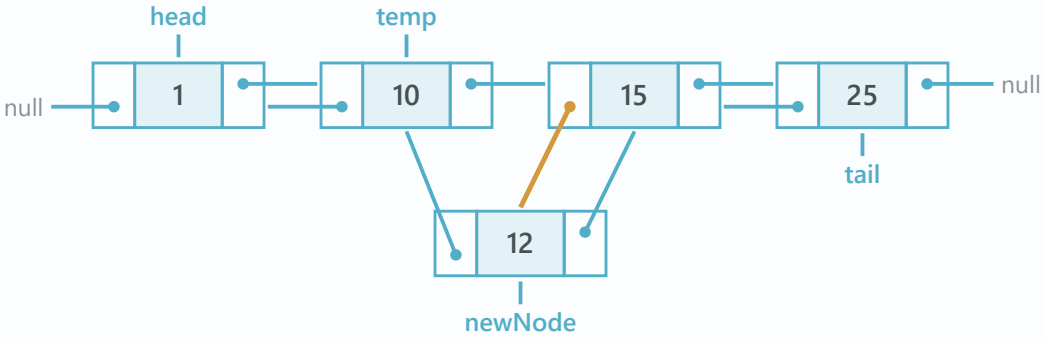
Step 3 / 6

```
1 Node newNode = new Node(data);
2 newNode.next = temp.next;      // (1)
▶ newNode.prev = temp;           // (2)
4 temp.next.prev = newNode;      // (3)
5 temp.next = newNode;           // (4)
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 12`
- `index = 2`
- `temp = node(10)`
- `newNode = node(12)`

Code Trace: `addAt(12, 2)` — the linking part



(3) `temp.next.prev = newNode` : 15's backward link now points to 12. This must come before (4), while temp.next is still 15.

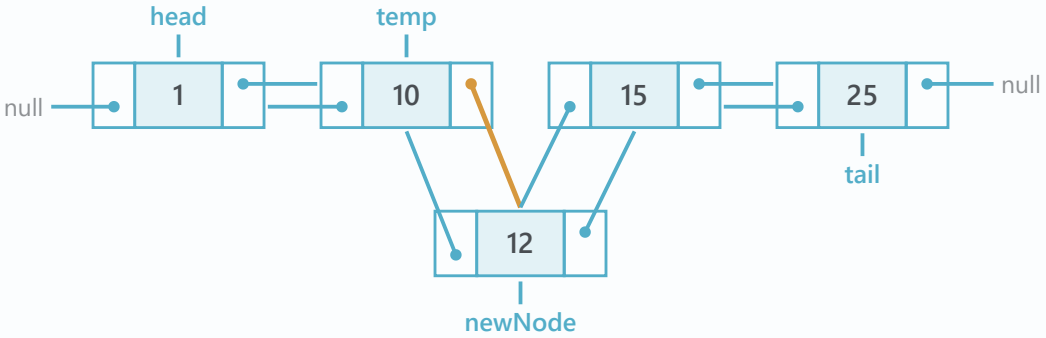
Step 4 / 6

```
1 Node newNode = new Node(data);
2 newNode.next = temp.next;     // (1)
3 newNode.prev = temp;          // (2)
▶ temp.next.prev = newNode;     // (3)
5 temp.next = newNode;          // (4)
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 12`
- `index = 2`
- `temp = node(10)`
- `newNode = node(12)`

Code Trace: `addAt(12, 2)` — the linking part



(4) `temp.next = newNode` : 10 → 12. Four links changed; swapping (3) and (4) would leave 15 unreachable.

Step 5 / 6

```
1 Node newNode = new Node(data);
2 newNode.next = temp.next;      // (1)
3 newNode.prev = temp;           // (2)
4 temp.next.prev = newNode;      // (3)
▶ temp.next = newNode;           // (4)
```

VARIABLES

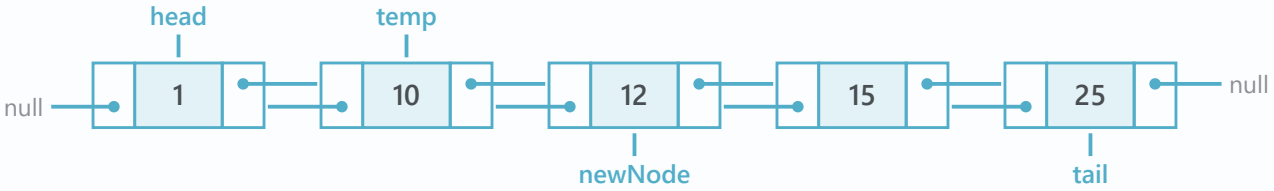
`head = node(1)`

`tail = node(25)` `data = 12`

`index = 2` `temp = node(10)`

`newNode = node(12)`

Code Trace: `addAt(12, 2)` — the linking part



Result (drawing tidied): 1 ⇄ 10 ⇄ 12 ⇄ 15 ⇄ 25. Linking is O(1); walking to the position is O(n).

Step 6 / 6

```
1 Node newNode = new Node(data);
2 newNode.next = temp.next;      // (1)
3 newNode.prev = temp;           // (2)
4 temp.next.prev = newNode;      // (3)
▶ temp.next = newNode;           // (4)
```

VARIABLES

`head = node(1)`

`tail = node(25)` `data = 12`

`index = 2` `temp = node(10)`

`newNode = node(12)`

Deleting at the Head and Tail of a Doubly Linked List

O(1)

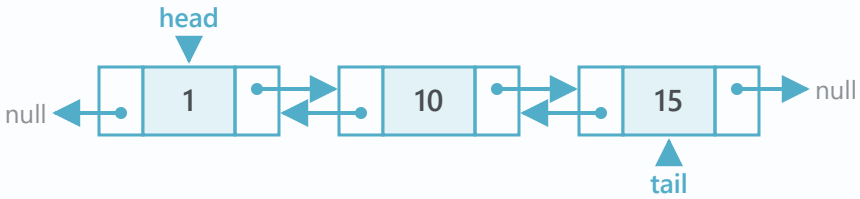
```
void removeFirst() {
    if (head == null) return;  // empty
    if (head == tail) {  // single node
        head = tail = null;
    } else {
        head = head.next;
        head.prev = null;
    }
}
```

O(1)

```
void removeLast() {
    if (tail == null) return;  // empty
    if (head == tail) {  // single node
        head = tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }
}
```

To delete at the tail we reach the previous node in one step with `tail.prev`; **no walk is needed**. In a singly linked list the same operation is O(n). When deleting the only node, set both `head` and `tail` to null.

Code Trace: Doubly Linked `removeFirst()`



List: 1 ⇄ 10 ⇄ 15. Not empty.

Step 1 / 4

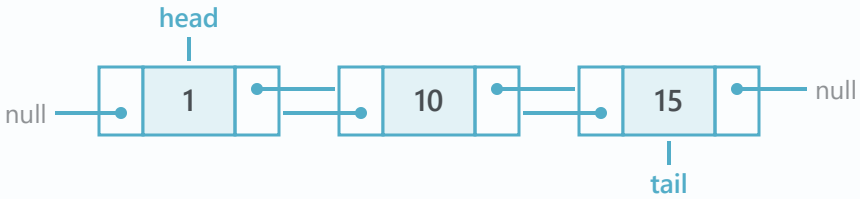
```
1 void removeFirst() {
▶       if (head == null) return;  // empty
3       if (head == tail) {  // single node
4           head = tail = null;
5       } else {
6           head = head.next;
7           head.prev = null;
8       }
9   }
```

VARIABLES

`head` = **node(1)**

`tail` = **node(15)**

Code Trace: Doubly Linked `removeFirst()`



`head != tail` → several nodes, `else` branch.

Step 2 / 4

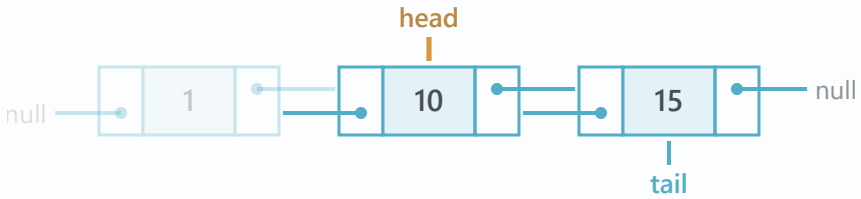
```
1 void removeFirst() {
2     if (head == null) return;  // empty
▶     if (head == tail) {  // single node
4         head = tail = null;
5     } else {
6         head = head.next;
7         head.prev = null;
8     }
9 }
```

VARIABLES

`head = node(1)`

`tail = node(15)`

Code Trace: Doubly Linked `removeFirst()`



`head = head.next` : head moves to 10; 1 is now unreachable.

Step 3 / 4

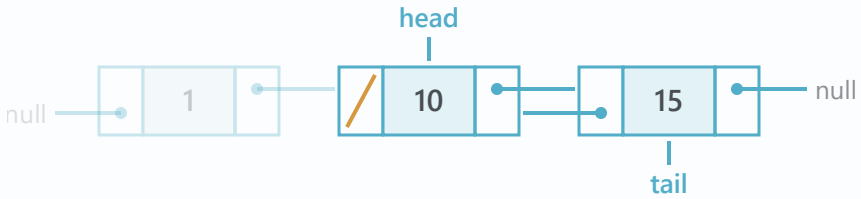
```
1 void removeFirst() {
2     if (head == null) return;  // empty
3     if (head == tail) {  // single node
4         head = tail = null;
5     } else {
▶         head = head.next;
7         head.prev = null;
8     }
9 }
```

VARIABLES

`head = node(10)`

`tail = node(15)`

Code Trace: Doubly Linked `removeFirst()`



`head.prev = null` : clear 10's backward link too; otherwise it would still point to the removed node. **O(1)**.

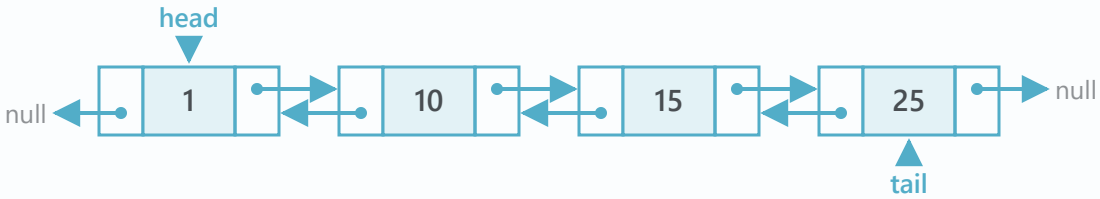
Step 4 / 4

```
1 void removeFirst() {
2     if (head == null) return;  // empty
3     if (head == tail) {  // single node
4         head = tail = null;
5     } else {
6         head = head.next;
          head.prev = null;
8     }
9 }
```

VARIABLES

- `head = node(10)`
- `tail = node(15)`

Deleting at the Tail, Step by Step



List: 1 ⇄ 10 ⇄ 15 ⇄ 25; `tail` points to 25.

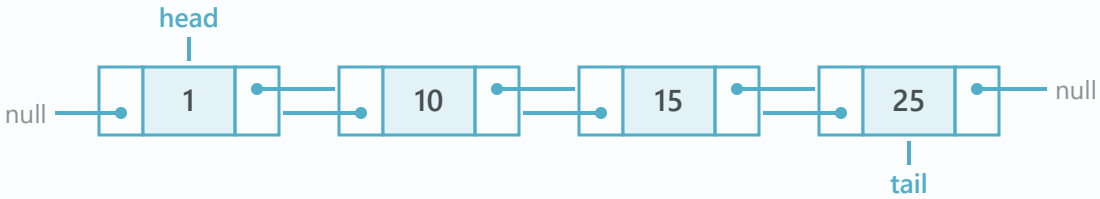
Step 1 / 4

```
1 void removeLast() {
▶       if (tail == null) return;
3       if (head == tail) {
4           head = tail = null;
5       } else {
6           tail = tail.prev;
7           tail.next = null;
8       }
9   }
```

VARIABLES

- head = **node(1)**
- tail = **node(25)**

Deleting at the Tail, Step by Step



`head != tail` → `else` branch.

Step 2 / 4

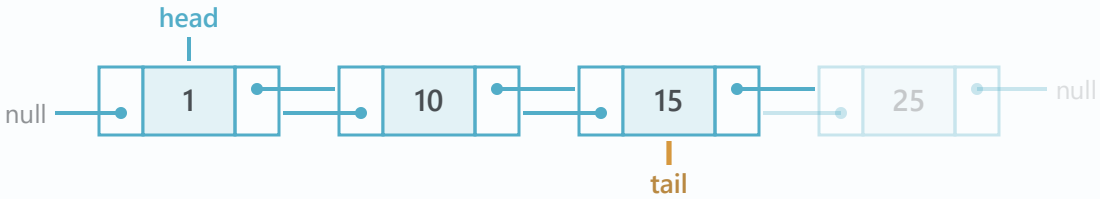
```
1 void removeLast() {
2     if (tail == null) return;
▶     if (head == tail) {
4         head = tail = null;
5     } else {
6         tail = tail.prev;
7         tail.next = null;
8     }
9 }
```

VARIABLES

`head` = **node(1)**

`tail` = **node(25)**

Deleting at the Tail, Step by Step



`tail = tail.prev` : we reach the second-to-last node **in one step** (a singly linked list would need a walk).

Step 3 / 4

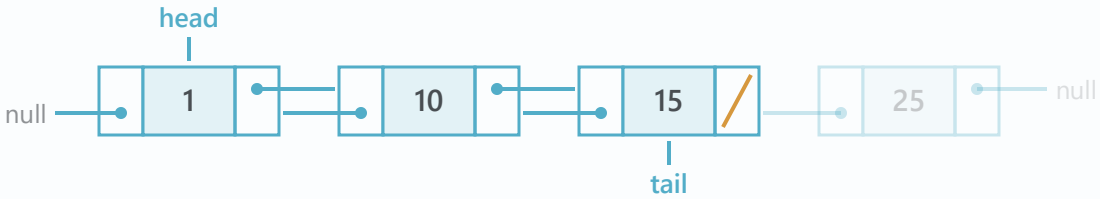
```
1 void removeLast() {
2     if (tail == null) return;
3     if (head == tail) {
4         head = tail = null;
5     } else {
          tail = tail.prev;
7         tail.next = null;
8     }
9 }
```

VARIABLES

`head = node(1)`

`tail = node(15)`

Deleting at the Tail, Step by Step



`tail.next = null` : 25 leaves the list. Result: 1 ⇄ 10 ⇄ 15. No walk: **O(1)**.

Step 4 / 4

```
1 void removeLast() {
2     if (tail == null) return;
3     if (head == tail) {
4         head = tail = null;
5     } else {
6         tail = tail.prev;
▶         tail.next = null;
8     }
9 }
```

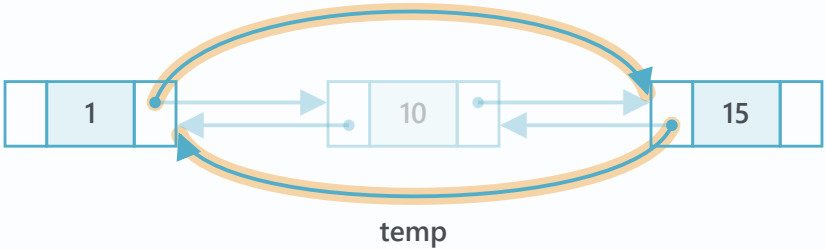
VARIABLES

`head = node(1)`

`tail = node(15)`

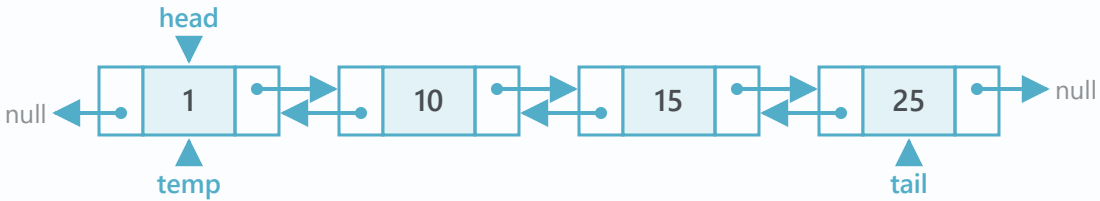
Deleting a Given Item from a Doubly Linked List

```
void remove(int data) {
    Node temp = head;
    while (temp != null && temp.data != data)
        temp = temp.next;
    if (temp == null) return;  // not found
    if (temp.prev != null)
        temp.prev.next = temp.next;
    else head = temp.next;   // removing the head
    if (temp.next != null)
        temp.next.prev = temp.prev;
    else tail = temp.prev;    // removing the tail
}
```



The removed node's **neighbors are linked to each other**. If the head or the tail is removed, update `head` / `tail`. Search O(n), deletion O(1).

Code Trace: Doubly Linked `remove(15)`



`remove(15)` : `temp = head` ; first we search for the node to delete.

Step 1 / 11

```
▶ Node temp = head;
2 while (temp != null && temp.data != data)
3     temp = temp.next;
4 if (temp == null) return;  // not found
5 if (temp.prev != null)
6     temp.prev.next = temp.next;
7 else head = temp.next;   // removing the head
8 if (temp.next != null)
9     temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

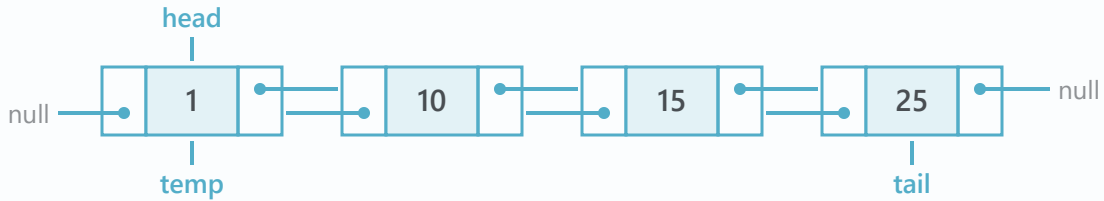
VARIABLES

`head` = **node(1)**

`tail` = **node(25)** `data` = **15**

`temp` = **node(1)**

Code Trace: Doubly Linked `remove(15)`



node(1) is not null and 1 ≠ 15 → advance.

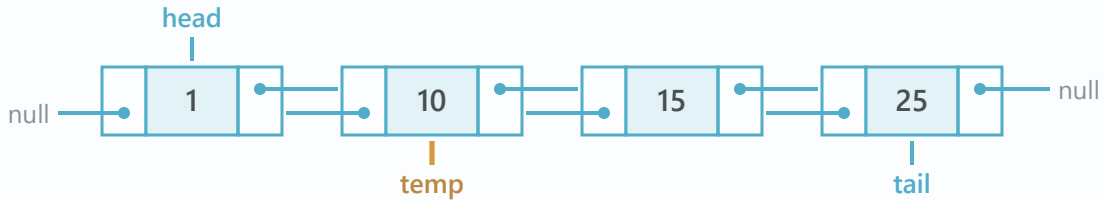
Step 2 / 11

```
1  Node temp = head;
▶  while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

VARIABLES

- `head` = **node(1)**
- `tail` = **node(25)**
- `data` = **15**
- `temp` = **node(1)**

Code Trace: Doubly Linked `remove(15)`



`temp = temp.next` → node(10).

Step 3 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
▶      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

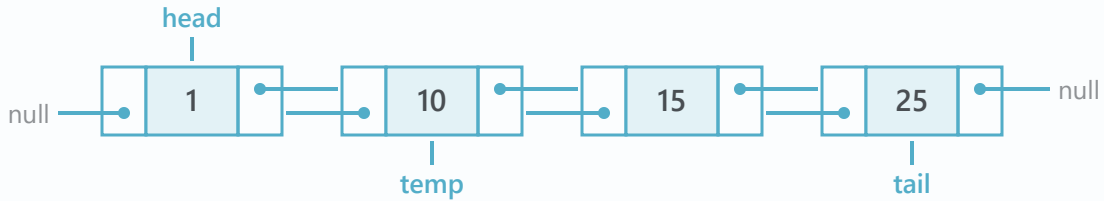
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 15`

`temp = node(10)`

Code Trace: Doubly Linked `remove(15)`



node(10) is not null and 10 ≠ 15 → advance.

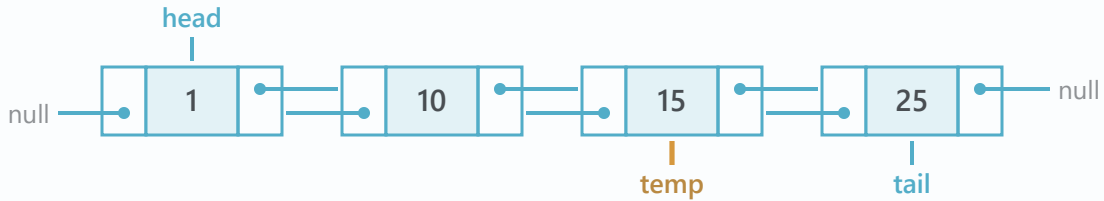
Step 4 / 11

```
1  Node temp = head;
▶ while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

VARIABLES

- `head` = **node(1)**
- `tail` = **node(25)**
- `data` = **15**
- `temp` = **node(10)**

Code Trace: Doubly Linked `remove(15)`



`temp = temp.next` → node(15).

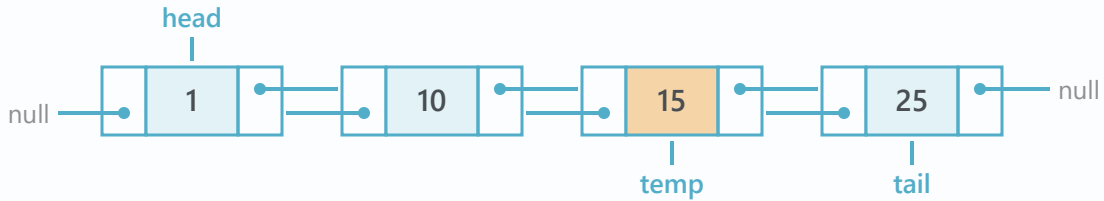
Step 5 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
▶      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 15`
- `temp = node(15)`

Code Trace: Doubly Linked `remove(15)`



temp.data == 15 → the loop ends: found it.

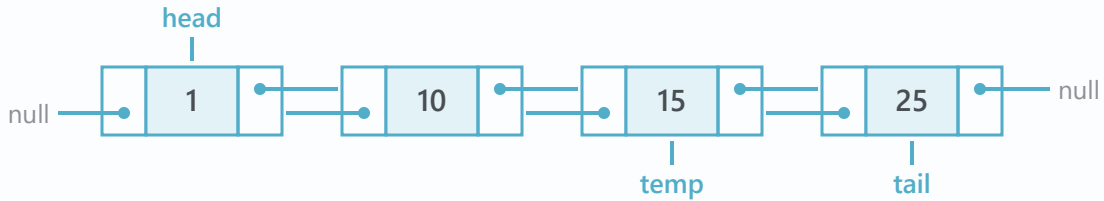
Step 6 / 11

```
1  Node temp = head;
▶ while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `data = 15`
- `temp = node(15)`

Code Trace: Doubly Linked `remove(15)`



temp is not null.

Step 7 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
3      temp = temp.next;
▶  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;    // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;     // removing the tail
```

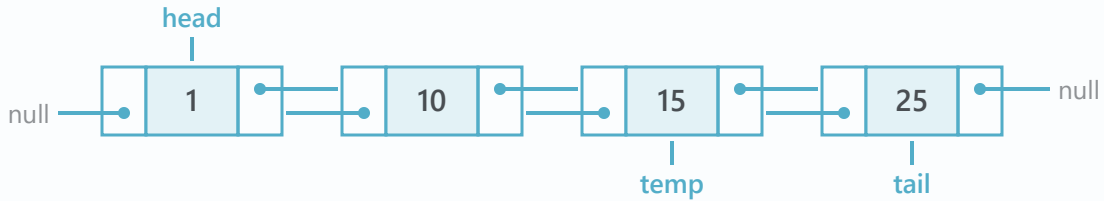
VARIABLES

`head` = **node(1)**

`tail` = **node(25)** `data` = **15**

`temp` = **node(15)**

Code Trace: Doubly Linked `remove(15)`



`temp.prev` (10) is not null → we are not removing the head.

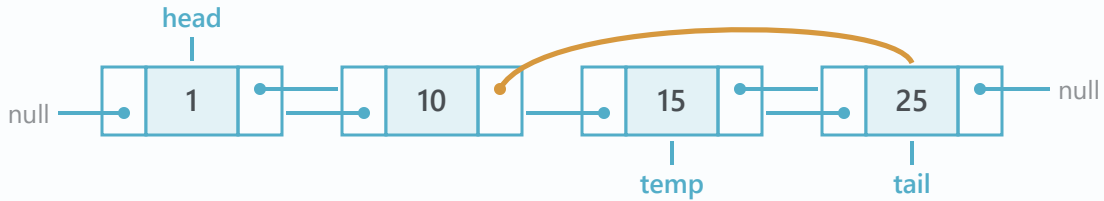
Step 8 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
▶  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

VARIABLES

- `head` = **node(1)**
- `tail` = **node(25)**
- `data` = **15**
- `temp` = **node(15)**

Code Trace: Doubly Linked `remove(15)`



`temp.prev.next = temp.next` : forward link 10 → 25; 15 is skipped.

Step 9 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
▶      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

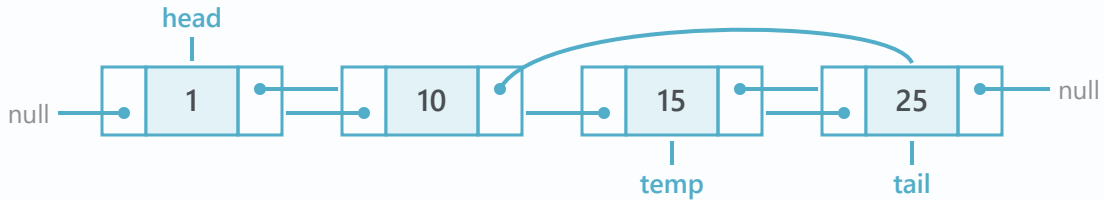
VARIABLES

`head` = **node(1)**

`tail` = **node(25)** `data` = **15**

`temp` = **node(15)**

Code Trace: Doubly Linked `remove(15)`



`temp.next` (25) is not null → we are not removing the tail.

Step 10 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
▶  if (temp.next != null)
9      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

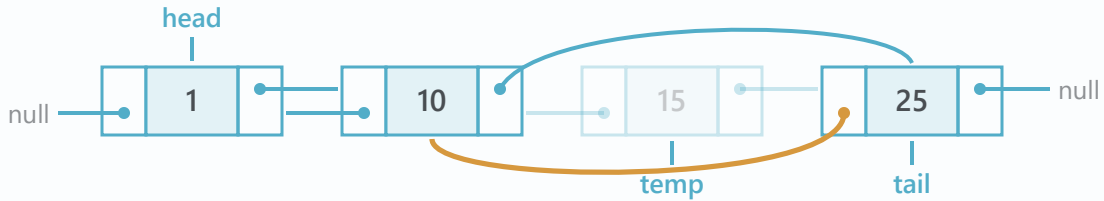
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 15`

`temp = node(15)`

Code Trace: Doubly Linked `remove(15)`



`temp.next.prev = temp.prev` : backward link 25 → 10. 1 ⇄ 10 ⇄ 25. Search O(n), relinking O(1).

Step 11 / 11

```
1  Node temp = head;
2  while (temp != null && temp.data != data)
3      temp = temp.next;
4  if (temp == null) return;  // not found
5  if (temp.prev != null)
6      temp.prev.next = temp.next;
7  else head = temp.next;   // removing the head
8  if (temp.next != null)
▶      temp.next.prev = temp.prev;
10 else tail = temp.prev;    // removing the tail
```

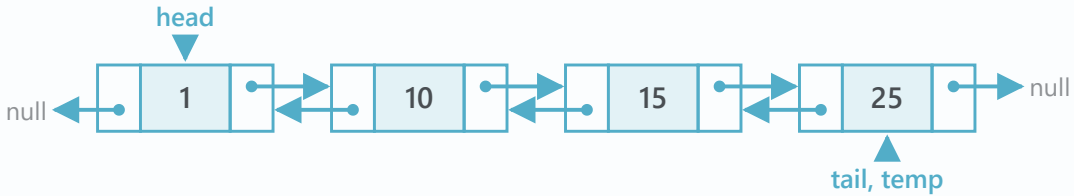
VARIABLES

`head = node(1)`

`tail = node(25)` `data = 15`

`temp = node(15)`

Printing a List Backward



`temp = tail` : the last node is known directly; no walk to the end.

Step 1 / 16

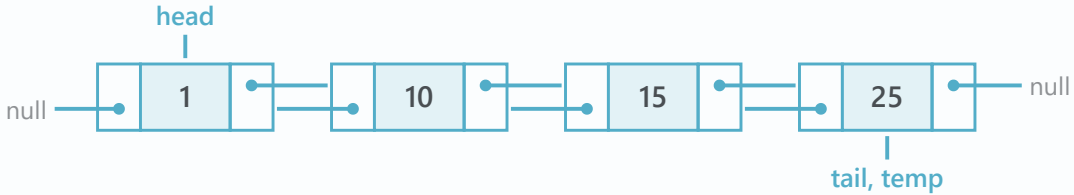
```
1 void printBackward() {
▶       Node temp = tail;
3       System.out.print("Backward: ");
4       while (temp != null) {
5           System.out.print(temp.data + " <-> ");
6           temp = temp.prev;
7       }
8       System.out.println("null");
9   }
```

VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(25)`

OUTPUT

Printing a List Backward



Print the heading.

Step 2 / 16

```
1 void printBackward() {
2     Node temp = tail;
▶     System.out.print("Backward: ");
4     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

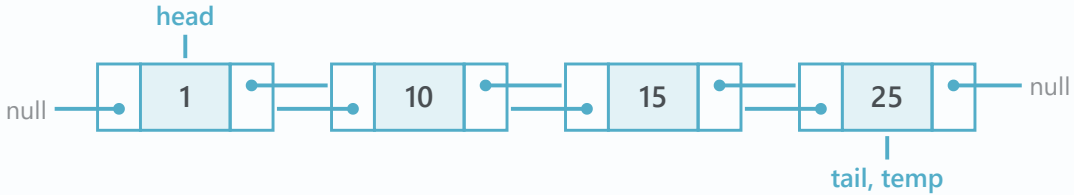
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(25)`

OUTPUT

```
Backward:
```

Printing a List Backward



temp = node(25) is not null.


```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

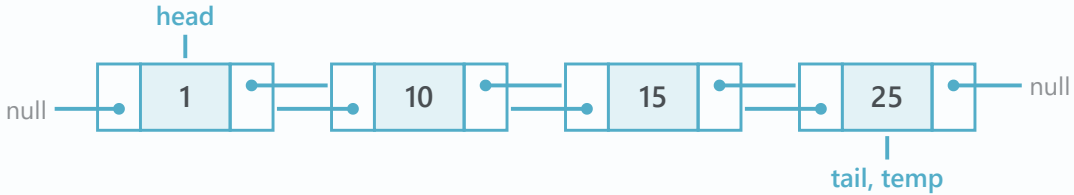
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(25)`

OUTPUT

```
Backward:
```

Printing a List Backward



Print 25.

Step 4 / 16

```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

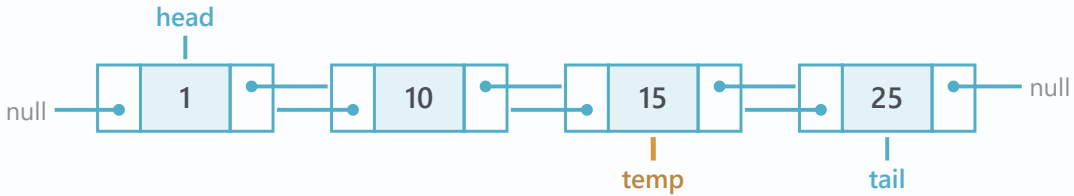
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(25)`

OUTPUT

```
Backward: 25 <->
```

Printing a List Backward



`temp = temp.prev` → node(15).


```
1  void printBackward() {
2      Node temp = tail;
3      System.out.print("Backward: ");
4      while (temp != null) {
5          System.out.print(temp.data + " <-> ");
▶          temp = temp.prev;
7      }
8      System.out.println("null");
9  }
```

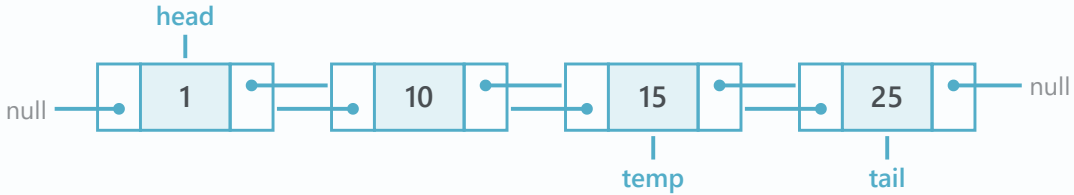
VARIABLES

- `head` = **node(1)**
- `tail` = **node(25)**
- `temp` = **node(15)**

OUTPUT

```
Backward: 25 <->
```

Printing a List Backward



temp = node(15) is not null.


```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

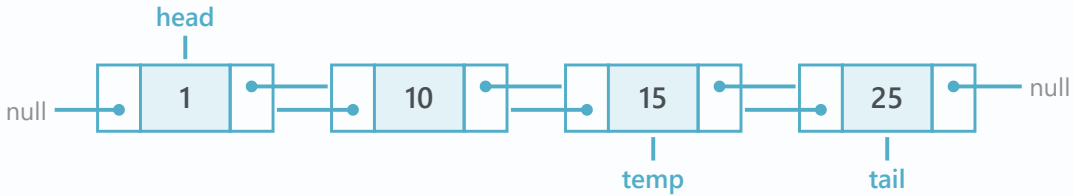
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(15)`

OUTPUT

```
Backward: 25 <->
```

Printing a List Backward



Print 15.

Step 7 / 16

```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

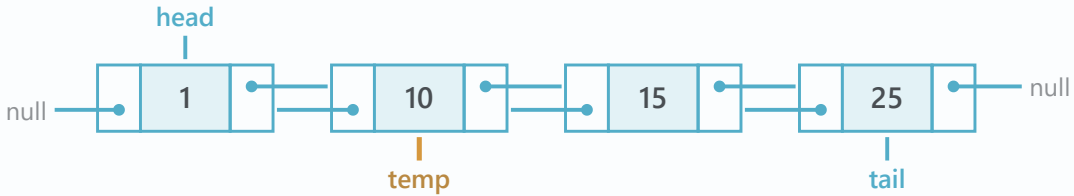
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(15)`

OUTPUT

```
Backward: 25 <-> 15 <->
```

Printing a List Backward



`temp = temp.prev` → node(10).


```
1  void printBackward() {
2      Node temp = tail;
3      System.out.print("Backward: ");
4      while (temp != null) {
5          System.out.print(temp.data + " <-> ");
▶          temp = temp.prev;
7      }
8      System.out.println("null");
9  }
```

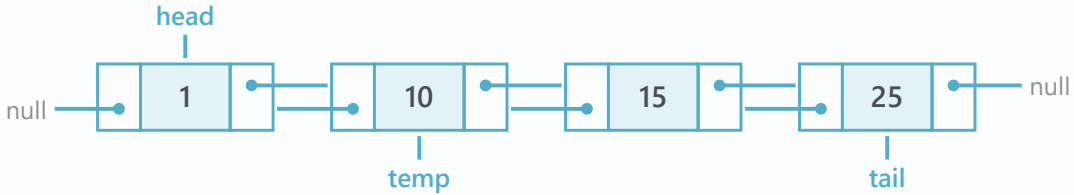
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(10)`

OUTPUT

```
Backward: 25 <-> 15 <->
```

Printing a List Backward



temp = node(10) is not null.


```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

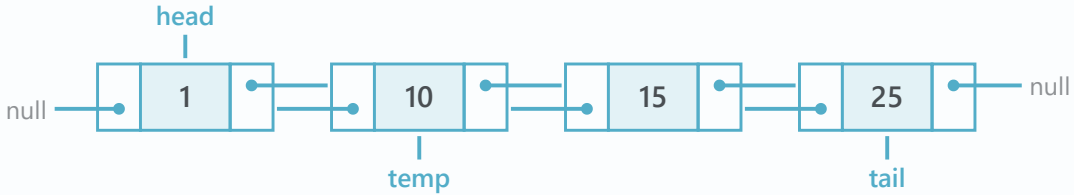
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(10)`

OUTPUT

```
Backward: 25 <-> 15 <->
```

Printing a List Backward



Print 10.

Step 10 / 16

```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

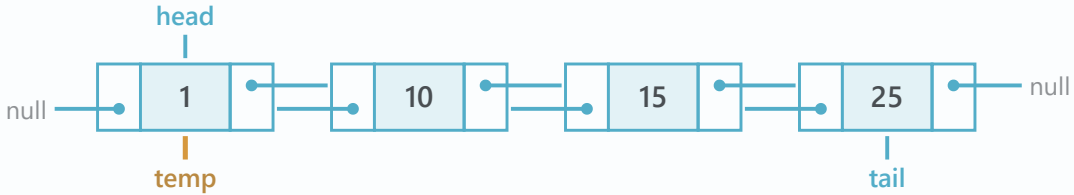
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(10)`

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <->
```

Printing a List Backward



`temp = temp.prev` → node(1).


```
1  void printBackward() {
2      Node temp = tail;
3      System.out.print("Backward: ");
4      while (temp != null) {
5          System.out.print(temp.data + " <-> ");
▶          temp = temp.prev;
7      }
8      System.out.println("null");
9  }
```

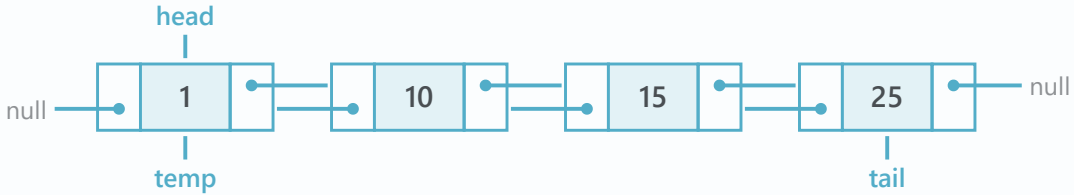
VARIABLES

- `head` = **node(1)**
- `tail` = **node(25)**
- `temp` = **node(1)**

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <->
```

Printing a List Backward



temp = node(1) is not null.


```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

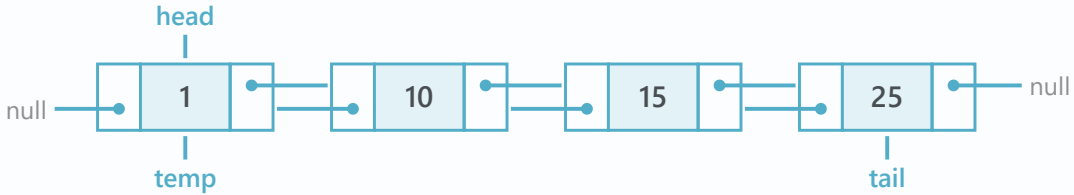
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(1)`

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <->
```

Printing a List Backward



Print 1.

Step 13 / 16

```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
4     while (temp != null) {
▶         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

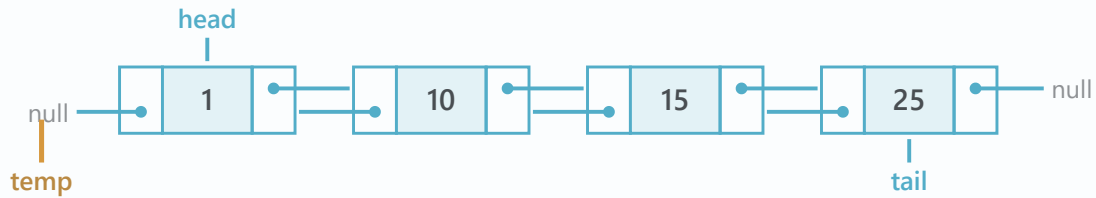
VARIABLES

- `head = node(1)`
- `tail = node(25)`
- `temp = node(1)`

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <-> 1 <->
```

Printing a List Backward



`temp = temp.prev` → **null** (1's prev field).


```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
4     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
▶         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

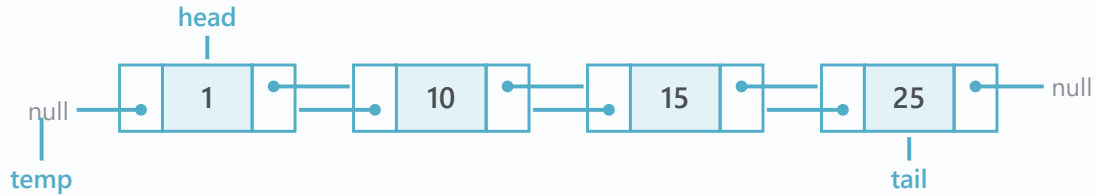
`head = node(1)`

`tail = node(25)` `temp = null`

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <-> 1 <->
```

Printing a List Backward



temp == null → the loop ends.


```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
▶     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
8     System.out.println("null");
9 }
```

VARIABLES

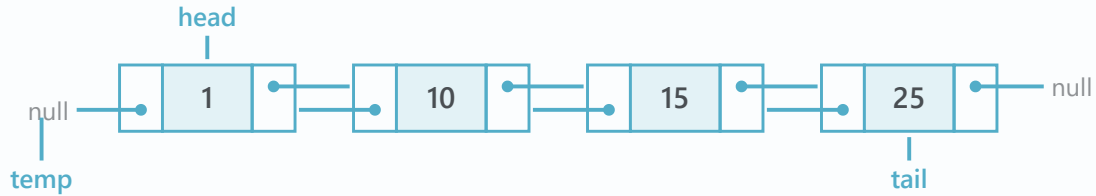
`head = node(1)`

`tail = node(25)` `temp = null`

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <-> 1 <->
```

Printing a List Backward



Print "null". Forward printing is the same: start at `head` and follow `next`. Both are O(n).

Step 16 / 16

```
1 void printBackward() {
2     Node temp = tail;
3     System.out.print("Backward: ");
4     while (temp != null) {
5         System.out.print(temp.data + " <-> ");
6         temp = temp.prev;
7     }
▶     System.out.println("null");
9 }
```

VARIABLES

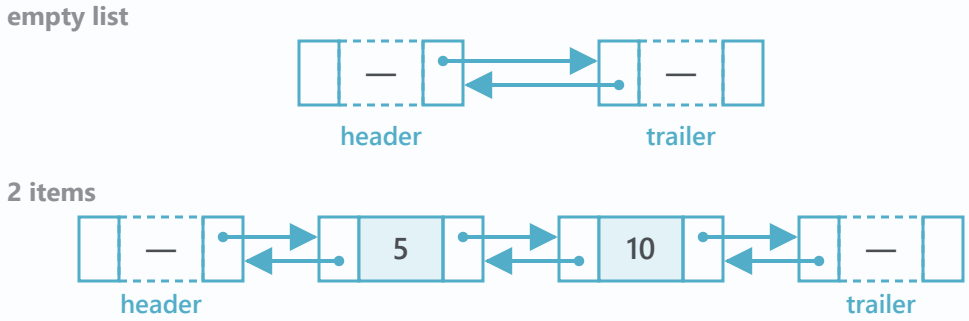
`head = node(1)`

`tail = node(25)` `temp = null`

OUTPUT

```
Backward: 25 <-> 15 <-> 10 <-> 1
<-> null
```

Sentinel Nodes: Header and Trailer



Idea

Put two fixed nodes that carry no data at the ends of the list (Goodrich–Tamassia: *header* / *trailer*).

No special cases

Every real node **always** has a predecessor and a successor: no `null` checks, no `head` / `tail` updates.

Price

Two extra nodes; traversal starts at `header.next` and stops at `trailer`.

Code Trace: Inserting and Deleting with Sentinels



Empty list: just the two sentinels, pointing to each other. `size` = 0. Only the sentinels' outer ends are `null`.

Step 1 / 16

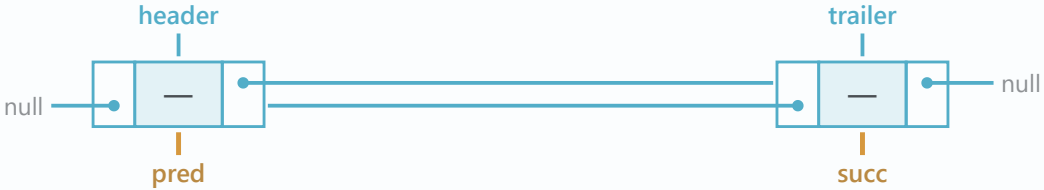
```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
▶  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

`header = node(–)`

`trailer = node(–)` `size = 0`

Code Trace: Inserting and Deleting with Sentinels



`addFirst(10)` : `addBetween(10, header, header.next)`. In an empty list, `header.next` is the trailer.

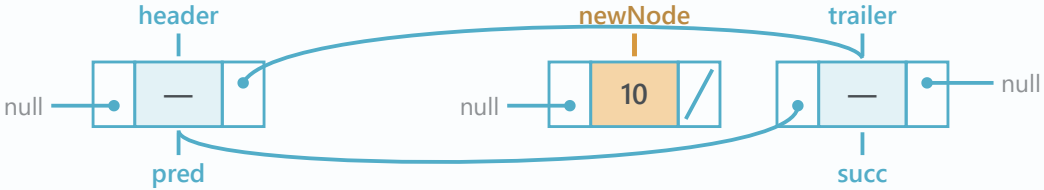
Step 2 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
▶      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 0`
- `v = 10`
- `pred = node(–)`
- `succ = node(–)`

Code Trace: Inserting and Deleting with Sentinels



`new Node(10)` : a new node with both links null.

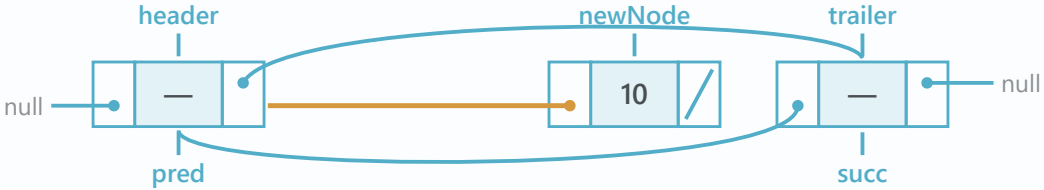
Step 3 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
▶      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 0`
- `v = 10`
- `pred = node(–)`
- `succ = node(–)`
- `newNode = node(10)`

Code Trace: Inserting and Deleting with Sentinels



`newNode.prev = pred` → node(—).

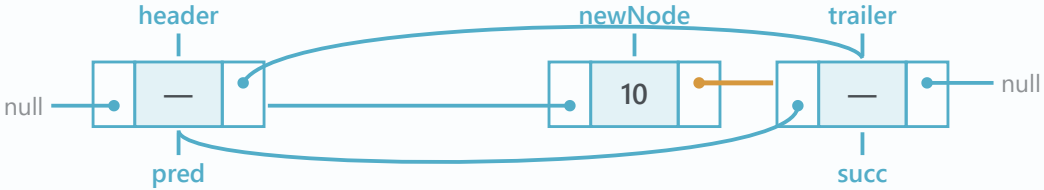
Step 4 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
▶      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(—)`
- `trailer = node(—)` `size = 0`
- `v = 10` `pred = node(—)`
- `succ = node(—)`
- `newNode = node(10)`

Code Trace: Inserting and Deleting with Sentinels



`newNode.next = succ` → node(—). The new node is ready; the list doesn't see it yet.

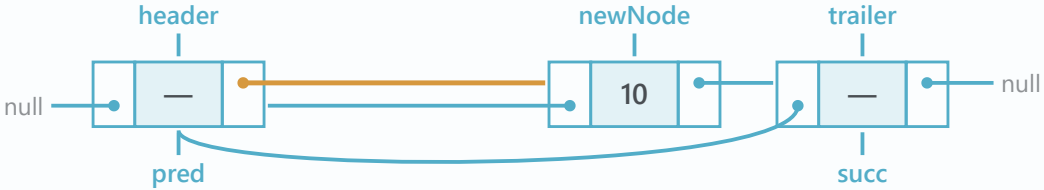
Step 5 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
▶      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(—)`
- `trailer = node(—)` `size = 0`
- `v = 10` `pred = node(—)`
- `succ = node(—)`
- `newNode = node(10)`

Code Trace: Inserting and Deleting with Sentinels



`pred.next = newNode` : forward — → 10.

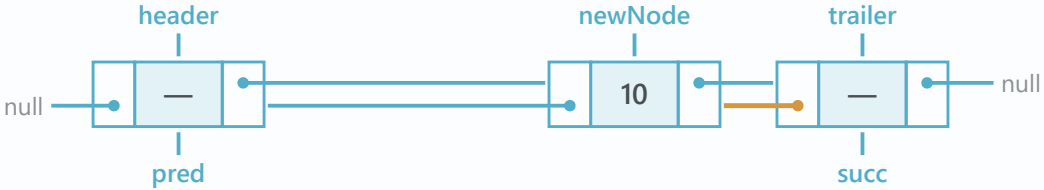
Step 6 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
▶      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)` `size = 0`
- `v = 10` `pred = node(–)`
- `succ = node(–)`
- `newNode = node(10)`

Code Trace: Inserting and Deleting with Sentinels



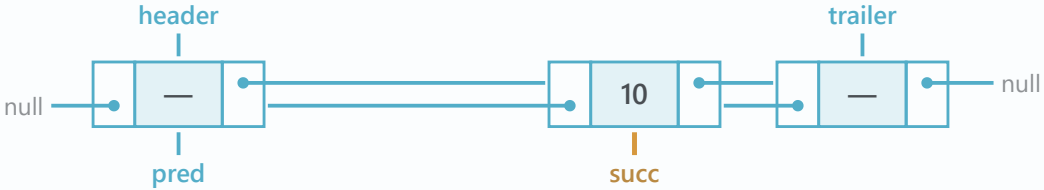
`succ.prev = newNode` : now linked backward too; `size` = 1. **No if**: pred and succ always exist.


```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
▶      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 1`
- `v = 10`
- `pred = node(–)`
- `succ = node(–)`
- `newNode = node(10)`

Code Trace: Inserting and Deleting with Sentinels



`addFirst(5)` : `addBetween(5, header, header.next)` — 5 goes between the header and 10. Same code, no special case.

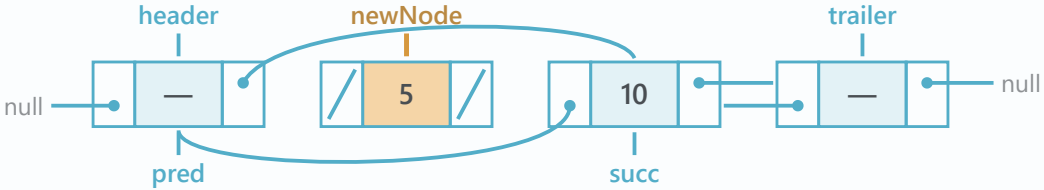
Step 8 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
▶      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 1`
- `v = 5`
- `pred = node(–)`
- `succ = node(10)`

Code Trace: Inserting and Deleting with Sentinels



`new Node(5)` : a new node with both links null.

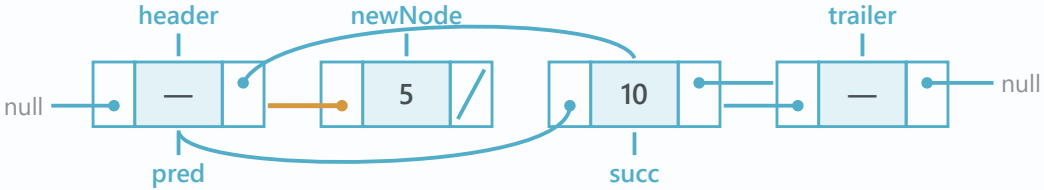
Step 9 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
▶      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 1`
- `v = 5`
- `pred = node(–)`
- `succ = node(10)`
- `newNode = node(5)`

Code Trace: Inserting and Deleting with Sentinels



`newNode.prev = pred` → node(—).

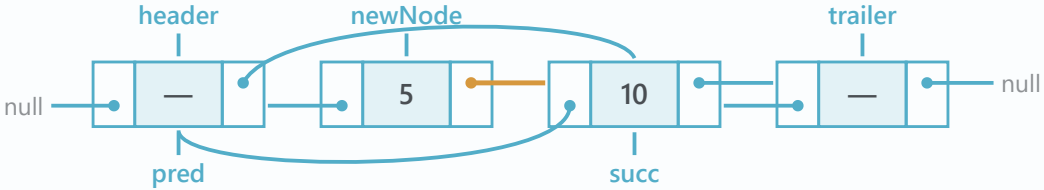
Step 10 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
▶      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 1`
- `v = 5`
- `pred = node(–)`
- `succ = node(10)`
- `newNode = node(5)`

Code Trace: Inserting and Deleting with Sentinels



`newNode.next = succ` → node(10). The new node is ready; the list doesn't see it yet.

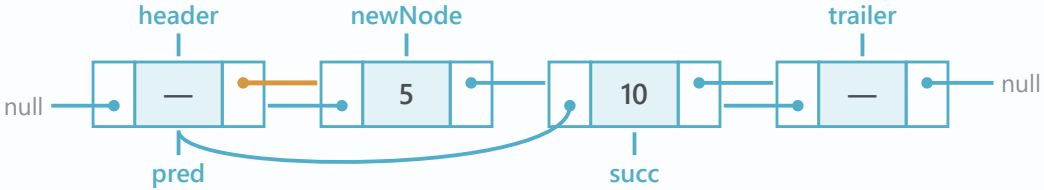
Step 11 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
▶      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- header = node(–)
- trailer = node(–)
- size = 1
- v = 5
- pred = node(–)
- succ = node(10)
- newNode = node(5)

Code Trace: Inserting and Deleting with Sentinels



`pred.next = newNode` : forward — → 5.

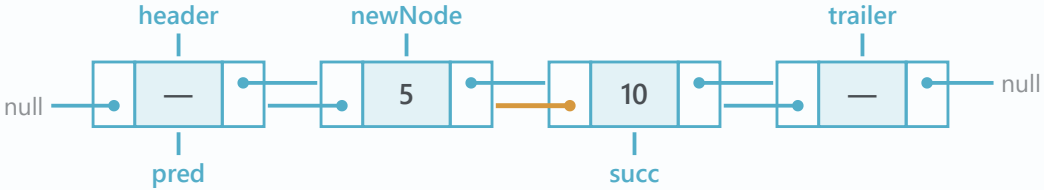
Step 12 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
▶      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)` `size = 1`
- `v = 5` `pred = node(–)`
- `succ = node(10)`
- `newNode = node(5)`

Code Trace: Inserting and Deleting with Sentinels



`succ.prev = newNode` : now linked backward too; `size` = 2. **No if**: pred and succ always exist.

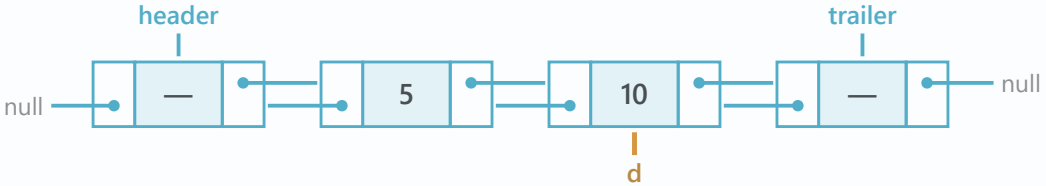
Step 13 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
▶      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

VARIABLES

- `header = node(–)`
- `trailer = node(–)`
- `size = 2`
- `v = 5`
- `pred = node(–)`
- `succ = node(10)`
- `newNode = node(5)`

Code Trace: Inserting and Deleting with Sentinels



`remove(d)`, d = node(10) (the last real node). Thanks to the sentinels, both neighbors are guaranteed to exist.

Step 14 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
▶  void remove(Node d) {
11     d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

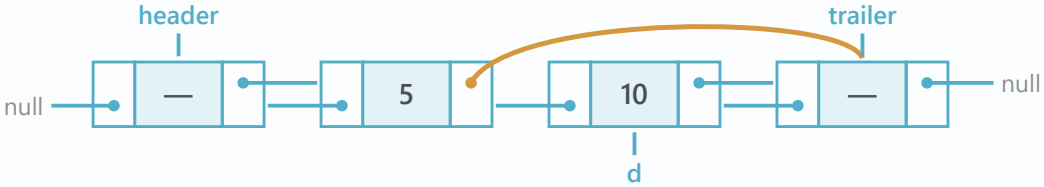
VARIABLES

header = **node(–)**

trailer = **node(–)** size = **2**

d = **node(10)**

Code Trace: Inserting and Deleting with Sentinels



`d.prev.next = d.next` : 5 now points straight to the trailer.

Step 15 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
▶      d.prev.next = d.next;
12     d.next.prev = d.prev;
13     size--; }
```

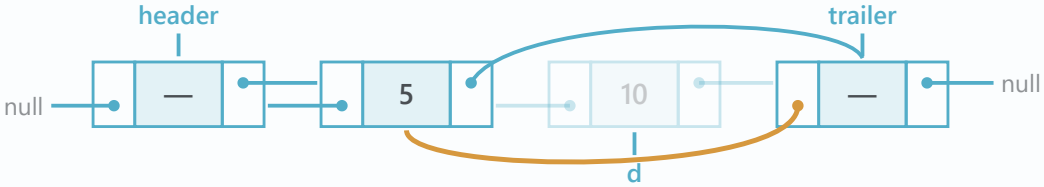
VARIABLES

`header = node(–)`

`trailer = node(–)` `size = 2`

`d = node(10)`

Code Trace: Inserting and Deleting with Sentinels



`d.next.prev = d.prev` : 10 is skipped in both directions; `size` = 1. Even when the last node goes, there is no `tail` update → every operation is **O(1)**.

Step 16 / 16

```
1  void addBetween(int v, Node pred, Node succ) {
2      Node newNode = new Node(v);
3      newNode.prev = pred;
4      newNode.next = succ;
5      pred.next = newNode;
6      succ.prev = newNode;  size++;
7  }
8  void addFirst(int v) {
9      addBetween(v, header, header.next); }
10 void remove(Node d) {
11     d.prev.next = d.next;
▶      d.next.prev = d.prev;
▶      size--; }
```

VARIABLES

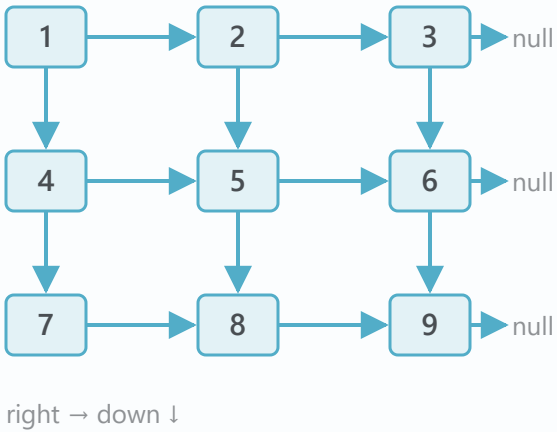
`header = node(–)`

`trailer = node(–)` `size = 1`

`d = node(10)`

Multi-Way Links: Turning a Matrix into a Linked Structure

Each node holds two links: `class Node { int data; Node right, down; }`
(constructor: `Node(int data)`)




```
void display(Node start) {
    Node down = start;
    while (down != null) {        // rows
        Node right = down;
        while (right != null) {     // columns
            System.out.print(right.data + " ");
            right = right.right;
        }
        System.out.println();
        down = down.down;
    }
}
```

Output, row by row: 1 2 3 / 4 5 6 / 7 8 9. Each node is visited once: O(rows · columns).

Building the Linked Structure from a Matrix

```
Node build(int[][] m) {
    int r = m.length, c = m[0].length;
    Node[][] d = new Node[r][c];
    for (int i = r - 1; i >= 0; i--)
        for (int j = c - 1; j >= 0; j--) {
            Node n = new Node(m[i][j]);
            if (j + 1 < c) n.right = d[i][j+1];
            if (i + 1 < r) n.down = d[i+1][j];
            d[i][j] = n;
        }
    return d[0][0];
}
```

Start at the bottom right

The right and lower neighbors already exist; each cell gets **one** node.

O(rows · columns)

Each cell is processed once.

Why the `d` table? A recursive version that calls `build(i, j+1)` and `build(i+1, j)` directly rebuilds the same cell over and over (an exponential number of nodes; the right and down paths don't share a common node).

Code Trace: `build(m)` — a 2×3 matrix



`d` : a 2×3 array of **node references**, all null. Nodes will be created starting from the bottom right.

Step 1 / 8

```
1  Node build(int[][] m) {
▶      int r = m.length, c = m[0].length;
▶      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
6              Node n = new Node(m[i][j]);
7              if (j + 1 < c) n.right = d[i][j+1];
8              if (i + 1 < r) n.down = d[i+1][j];
9              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

VARIABLES

`r = 2` `c = 3`

`d = [[null, null, null], [null, null, null]]`

Code Trace: `build(m)` — a 2×3 matrix



(i=1, j=2): `new Node(6)`; no `right` (last column), no `down` (last row). Neighbors already exist; no cell is built twice.

Step 2 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
▶              Node n = new Node(m[i][j]);
▶              if (j + 1 < c) n.right = d[i][j+1];
▶              if (i + 1 < r) n.down = d[i+1][j];
▶              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

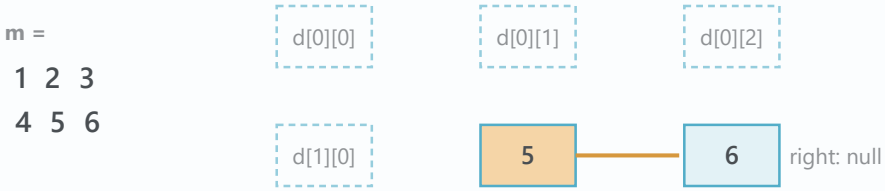
VARIABLES

`r = 2` `c = 3` `i = 1` `j = 2`

`n = node(6)`

`d = [[null, null, null], [null, null, node(6)]]`

Code Trace: `build(m)` — a 2×3 matrix



(i=1, j=1): `new Node(5)` ; `right` → 6, no `down` (last row). Neighbors already exist; no cell is built twice.

Step 3 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
▶              Node n = new Node(m[i][j]);
▶              if (j + 1 < c) n.right = d[i][j+1];
▶              if (i + 1 < r) n.down = d[i+1][j];
▶              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

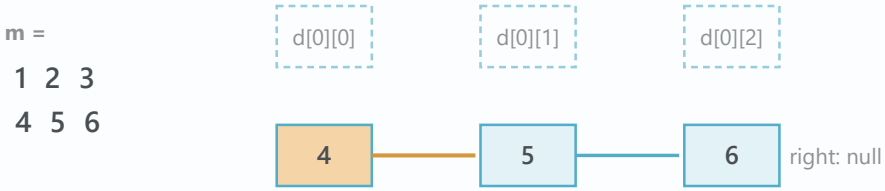
VARIABLES

`r = 2` `c = 3` `i = 1` `j = 1`

`n = node(5)`

`d = [[null, null, null], [null, node(5), node(6)]]`

Code Trace: `build(m)` — a 2×3 matrix



(i=1, j=0): `new Node(4)` ; `right` → 5, no `down` (last row). Neighbors already exist; no cell is built twice.

Step 4 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
▶              Node n = new Node(m[i][j]);
▶              if (j + 1 < c) n.right = d[i][j+1];
▶              if (i + 1 < r) n.down = d[i+1][j];
▶              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

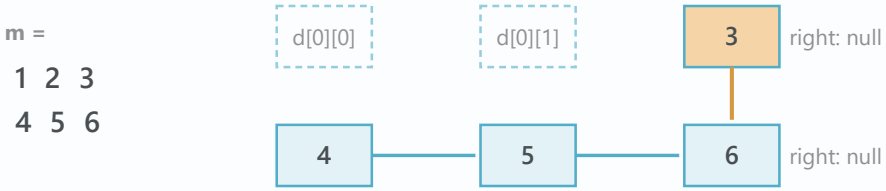
VARIABLES

`r = 2` `c = 3` `i = 1` `j = 0`

`n = node(4)`

`d = [[null, null, null], [node(4), node(5), node(6)]]`

Code Trace: `build(m)` — a 2×3 matrix



(i=0, j=2): `new Node(3)`; no `right` (last column), `down` → 6. Neighbors already exist; no cell is built twice.

Step 5 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
▶              Node n = new Node(m[i][j]);
▶              if (j + 1 < c) n.right = d[i][j+1];
▶              if (i + 1 < r) n.down = d[i+1][j];
▶              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

VARIABLES

`r = 2` `c = 3` `i = 0` `j = 2`

`n = node(3)`

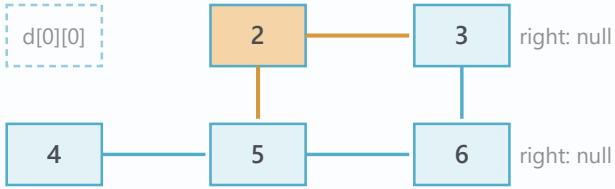
`d = [[null, null, node(3)], [node(4), node(5), node(6)]]`

Code Trace: `build(m)` — a 2×3 matrix

m =

1 2 3

4 5 6



(i=0, j=1): `new Node(2)`; `right` → 3, `down` → 5. Neighbors already exist; no cell is built twice.

Step 6 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
▶              Node n = new Node(m[i][j]);
▶              if (j + 1 < c) n.right = d[i][j+1];
▶              if (i + 1 < r) n.down = d[i+1][j];
▶              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

VARIABLES

`r = 2` `c = 3` `i = 0` `j = 1`

`n = node(2)`

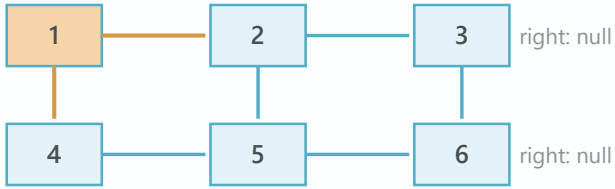
`d = [[null, node(2), node(3)], [node(4), node(5), node(6)]]`

Code Trace: `build(m)` — a 2×3 matrix

m =

1 2 3

4 5 6



(i=0, j=0): `new Node(1)`; `right` → 2, `down` → 4. Neighbors already exist; no cell is built twice.

Step 7 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
▶              Node n = new Node(m[i][j]);
▶              if (j + 1 < c) n.right = d[i][j+1];
▶              if (i + 1 < r) n.down = d[i+1][j];
▶              d[i][j] = n;
10         }
11     return d[0][0];
12 }
```

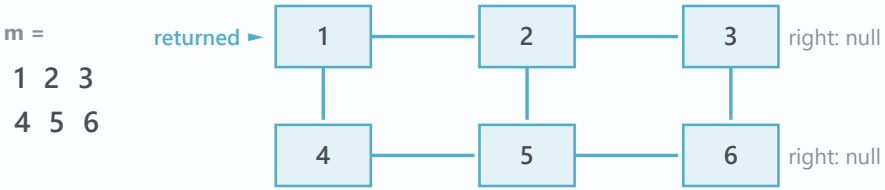
VARIABLES

`r = 2` `c = 3` `i = 0` `j = 0`

`n = node(1)`

`d = [[node(1), node(2), node(3)], [node(4), node(5), node(6)]]`

Code Trace: `build(m)` — a 2×3 matrix



Return `d[0][0]` (1): the whole structure is reachable from the top-left corner through `right` / `down`. 6 cells, 6 nodes → O(r·c).

Step 8 / 8

```
1  Node build(int[][] m) {
2      int r = m.length, c = m[0].length;
3      Node[][] d = new Node[r][c];
4      for (int i = r - 1; i >= 0; i--)
5          for (int j = c - 1; j >= 0; j--) {
6              Node n = new Node(m[i][j]);
7              if (j + 1 < c) n.right = d[i][j+1];
8              if (i + 1 < r) n.down = d[i+1][j];
9              d[i][j] = n;
10         }
▶      return d[0][0];
12 }
```

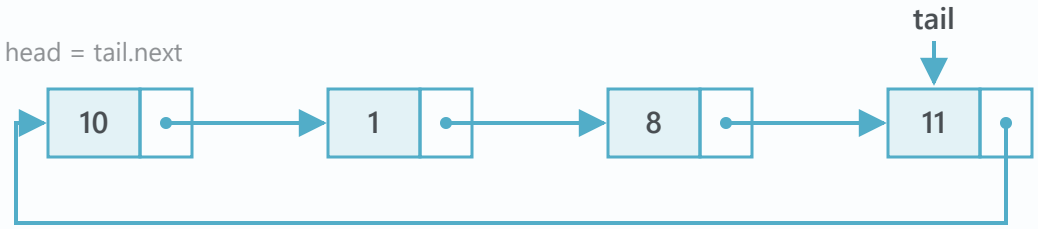
VARIABLES

`r = 2` `c = 3`

`d = [[node(1), node(2), node(3)], [node(4), node(5), node(6)]]`

Circularly Linked List

- The last node points to the **first node** instead of `null`, so the list forms a cycle (a ring).
- Each node still points to the node after it; the list has no `null` links.
- Keeping `tail` alone is enough: the head is `tail.next`.
- Uses: jobs that take turns (round-robin scheduling), turn order in multiplayer games, a looping playlist.



The Circular List Class

```
public class CircularLinkedList {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node tail;  // head = tail.next
    private int size;
}
```

Why only tail?

Head = `tail.next`, so both ends are reachable in O(1).

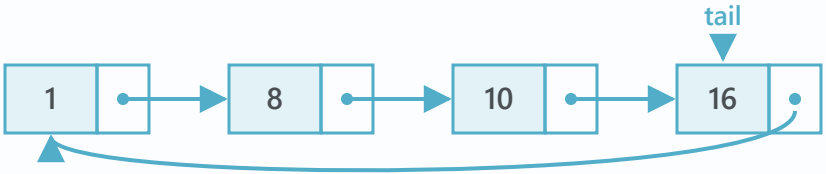
Keeping head as well

Every operation would have to update two pointers; unnecessary.

Special cases

Empty: `tail == null`. One node: `tail.next == tail`.

Printing a Circular List



The list is not empty. Only `tail` is kept; the last node points to the head (the bottom link).

Step 1 / 13

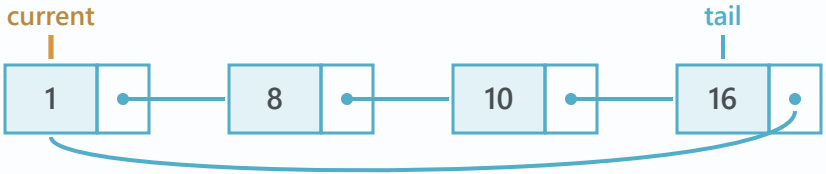
```
1 void print() {
▶     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
5         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

`tail = node(16)`

OUTPUT

Printing a Circular List



The head is `tail.next` : `current` = 1.


```
1  void print() {
2      if (tail == null) return;
▶      Node current = tail.next;
4      while (current != tail) {
5          System.out.print(current.data + " ");
6          current = current.next;
7      }
8      System.out.print(current.data + " ");
9  }
```

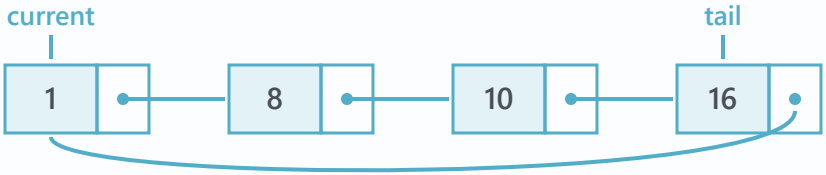
VARIABLES

`tail = node(16)`

`current = node(1)`

OUTPUT

Printing a Circular List



current (1) != tail (16) → loop.

Step 3 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
▶     while (current != tail) {
5         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

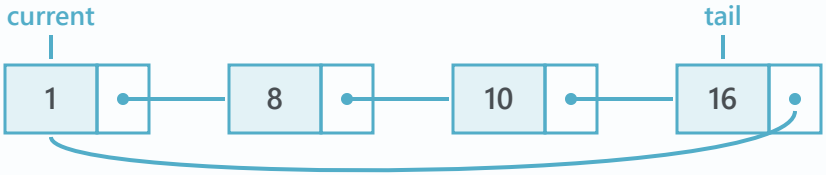
VARIABLES

`tail = node(16)`

`current = node(1)`

OUTPUT

Printing a Circular List



Print 1.

Step 4 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
▶         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

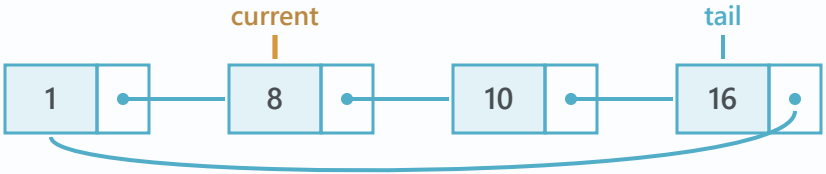
`tail = node(16)`

`current = node(1)`

OUTPUT

```
1
```

Printing a Circular List



`current = current.next` → node(8).

Step 5 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
5         System.out.print(current.data + " ");
▶         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

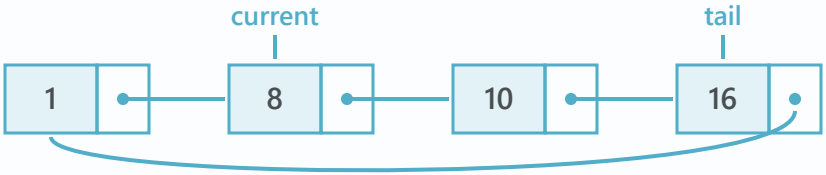
`tail = node(16)`

`current = node(8)`

OUTPUT

```
1
```

Printing a Circular List



current (8) != tail (16) → loop.

Step 6 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
▶     while (current != tail) {
5         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

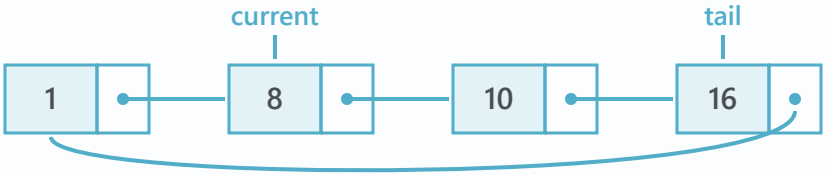
`tail = node(16)`

`current = node(8)`

OUTPUT

```
1
```

Printing a Circular List



Print 8.

Step 7 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
▶         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

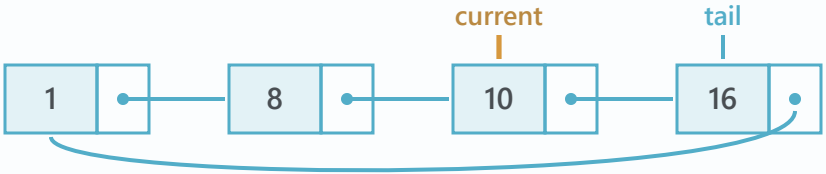
`tail = node(16)`

`current = node(8)`

OUTPUT

```
1 8
```

Printing a Circular List



`current = current.next` → node(10).

Step 8 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
5         System.out.print(current.data + " ");
▶         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

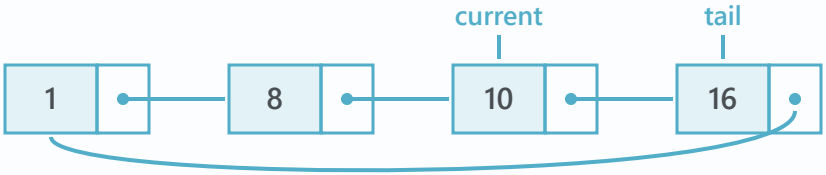
`tail = node(16)`

`current = node(10)`

OUTPUT

```
1 8
```

Printing a Circular List



current (10) != tail (16) → loop.

Step 9 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
▶     while (current != tail) {
5         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

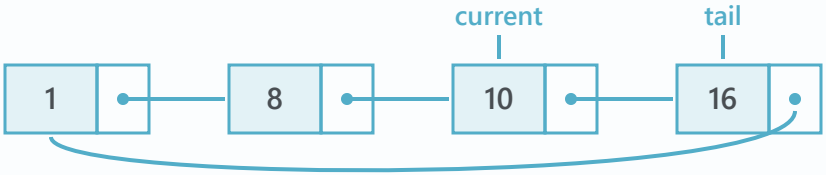
`tail = node(16)`

`current = node(10)`

OUTPUT

```
1 8
```

Printing a Circular List



Print 10.

Step 10 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
▶         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

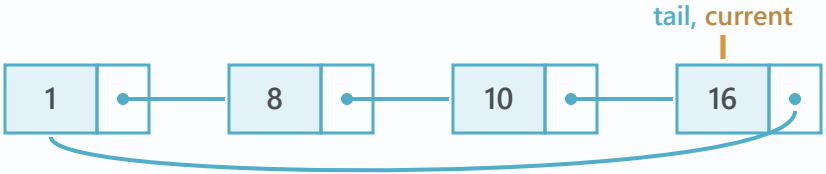
`tail = node(16)`

`current = node(10)`

OUTPUT

```
1 8 10
```

Printing a Circular List



`current = current.next` → node(16).

Step 11 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
5         System.out.print(current.data + " ");
▶         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

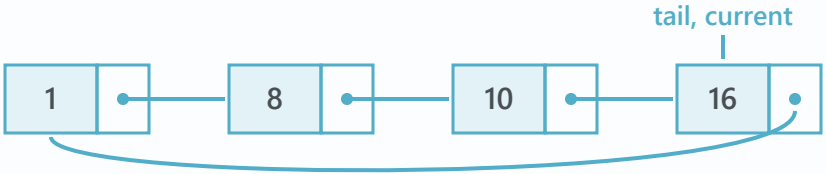
`tail = node(16)`

`current = node(16)`

OUTPUT

```
1 8 10
```

Printing a Circular List



`current == tail` → the loop ends (we never reach null; the condition is `!= tail`).

Step 12 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
▶     while (current != tail) {
5         System.out.print(current.data + " ");
6         current = current.next;
7     }
8     System.out.print(current.data + " ");
9 }
```

VARIABLES

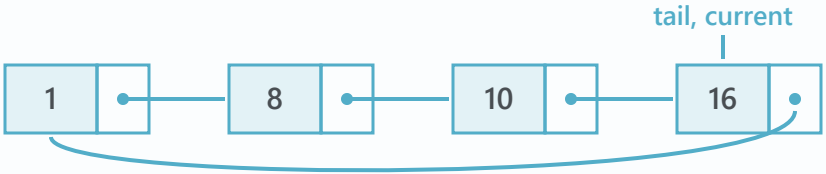
`tail = node(16)`

`current = node(16)`

OUTPUT

```
1 8 10
```

Printing a Circular List



The last node is printed separately after the loop. Each node once: **O(n)**.

Step 13 / 13

```
1 void print() {
2     if (tail == null) return;
3     Node current = tail.next;
4     while (current != tail) {
5         System.out.print(current.data + " ");
6         current = current.next;
7     }
▶     System.out.print(current.data + " ");
9 }
```

VARIABLES

`tail = node(16)`

`current = node(16)`

OUTPUT

```
1 8 10 16
```

Detecting a Cycle in General: Floyd's Algorithm

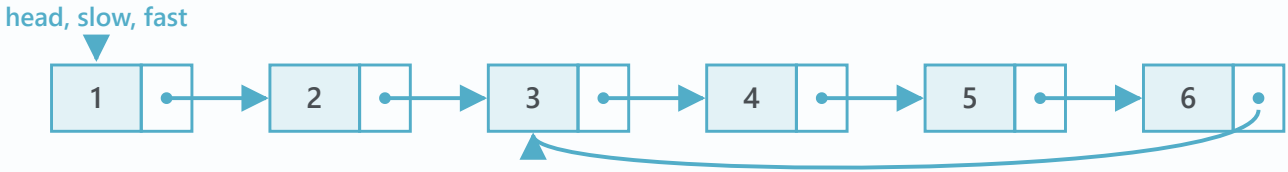
```
boolean hasCycle(Node head) {
    Node slow = head, fast = head;
    while (fast != null &&
           fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
```

- Each round, the slow pointer moves one node and the fast pointer two.
- If there is a cycle, the two pointers eventually meet inside it.
- If there is no cycle, the fast pointer reaches `null`.
- The cycle need not lead back to the `head` node.

Time	**Extra space**
O(n)	O(1)

Following the nodes until we get back to `head` is not enough: the cycle may loop back to a node in the middle of the list.

Code Trace: `hasCycle` (Floyd)



The list is 1 → … → 6, and 6 points back to 3 (the cycle does not return to the head). Both pointers start at the head.

Step 1 / 17

```
1  boolean hasCycle(Node head) {
▶      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

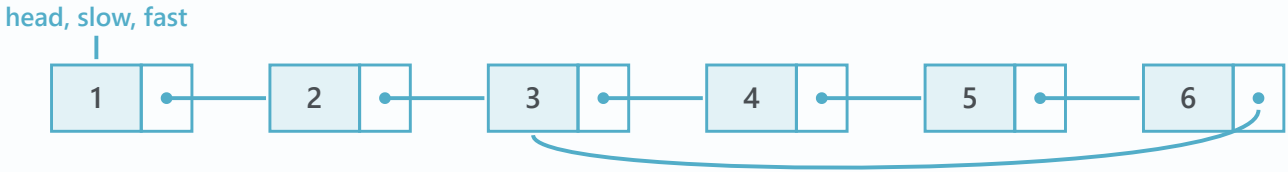
VARIABLES

`head` = **node(1)**

`slow` = **node(1)**

`fast` = **node(1)**

Code Trace: `hasCycle` (Floyd)



Round 1: fast (1) and fast.next (2) are not null.

Step 2 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
▶      while (fast != null &&
▶             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

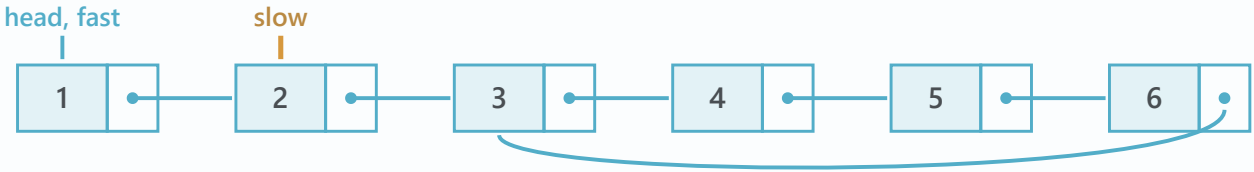
VARIABLES

`head` = **node(1)**

`slow` = **node(1)**

`fast` = **node(1)**

Code Trace: `hasCycle` (Floyd)



`slow` moves 1 step → node(2).

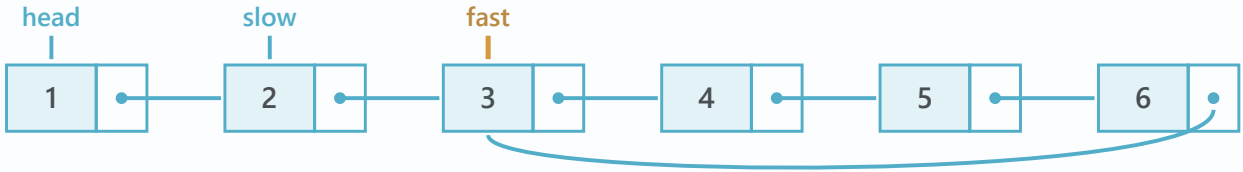
Step 3 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
▶          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head = node(1)` `slow = node(2)` `fast = node(1)`

Code Trace: `hasCycle` (Floyd)



`fast` moves 2 steps → node(3).

Step 4 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
▶          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

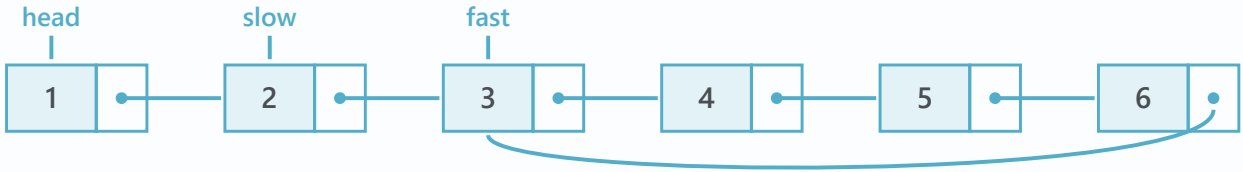
VARIABLES

`head` = **node(1)**

`slow` = **node(2)**

`fast` = **node(3)**

Code Trace: `hasCycle` (Floyd)



slow (2) ≠ fast (3) → continue.

Step 5 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
▶          if (slow == fast) return true;
8      }
9      return false;
10 }
```

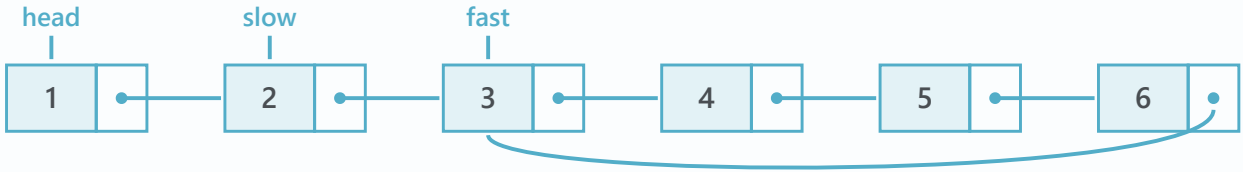
VARIABLES

`head` = **node(1)**

`slow` = **node(2)**

`fast` = **node(3)**

Code Trace: `hasCycle` (Floyd)



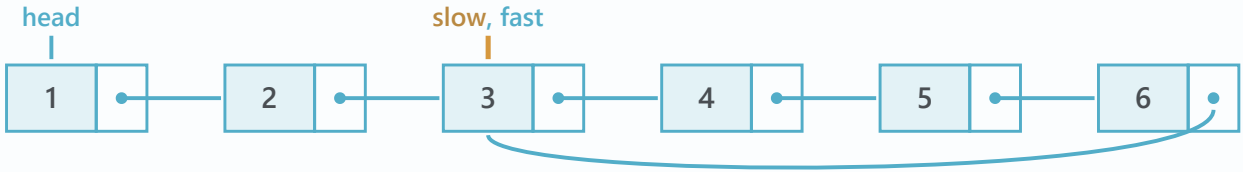
Round 2: fast (3) and fast.next (4) are not null.


```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
▶      while (fast != null &&
▶             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head = node(1)` `slow = node(2)` `fast = node(3)`

Code Trace: `hasCycle` (Floyd)



`slow` moves 1 step → node(3).

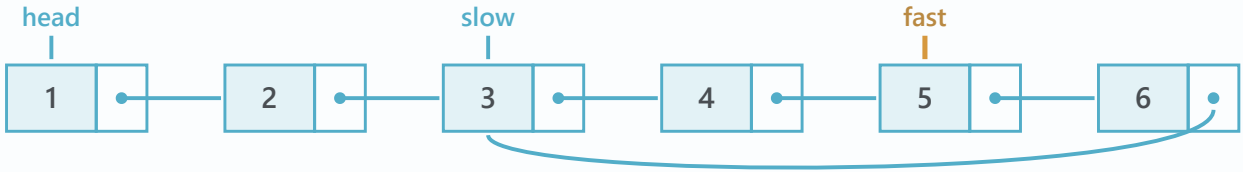
Step 7 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
▶          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head = node(1)` `slow = node(3)` `fast = node(3)`

Code Trace: `hasCycle` (Floyd)



`fast` moves 2 steps → node(5).

Step 8 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
▶          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

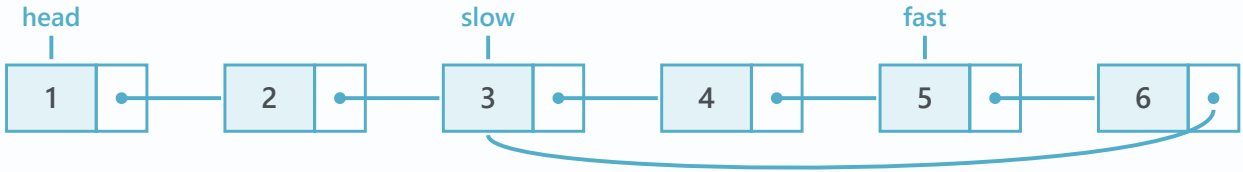
VARIABLES

`head` = **node(1)**

`slow` = **node(3)**

`fast` = **node(5)**

Code Trace: `hasCycle` (Floyd)



slow (3) ≠ fast (5) → continue.

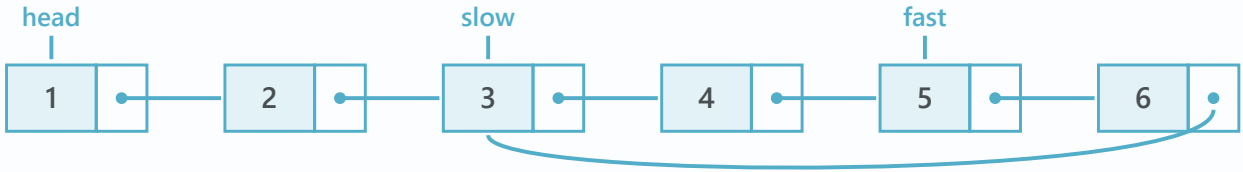
Step 9 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
▶          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head` = **node(1)** `slow` = **node(3)** `fast` = **node(5)**

Code Trace: `hasCycle` (Floyd)



Round 3: fast (5) and fast.next (6) are not null.

Step 10 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
▶      while (fast != null &&
▶             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

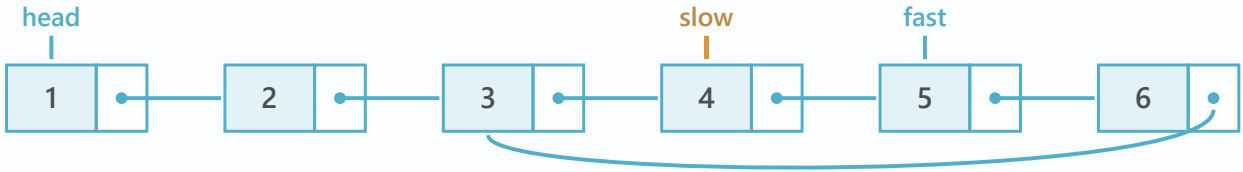
VARIABLES

`head` = **node(1)**

`slow` = **node(3)**

`fast` = **node(5)**

Code Trace: `hasCycle` (Floyd)



`slow` moves 1 step → node(4).

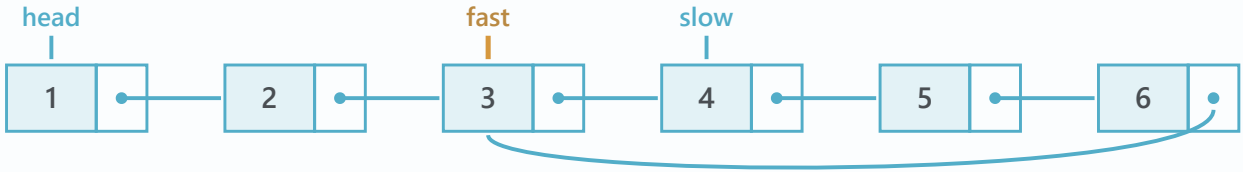
Step 11 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
▶          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head = node(1)` `slow = node(4)` `fast = node(5)`

Code Trace: `hasCycle` (Floyd)



`fast` moves 2 steps → node(3).

Step 12 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
▶          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

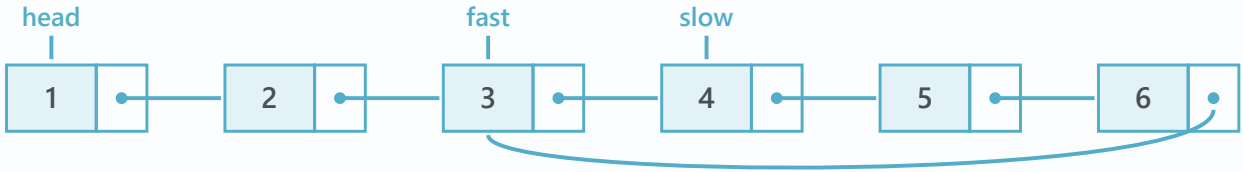
VARIABLES

`head` = **node(1)**

`slow` = **node(4)**

`fast` = **node(3)**

Code Trace: `hasCycle` (Floyd)



slow (4) ≠ fast (3) → continue. fast has entered the cycle and is closing in on slow from behind.

Step 13 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
▶          if (slow == fast) return true;
8      }
9      return false;
10 }
```

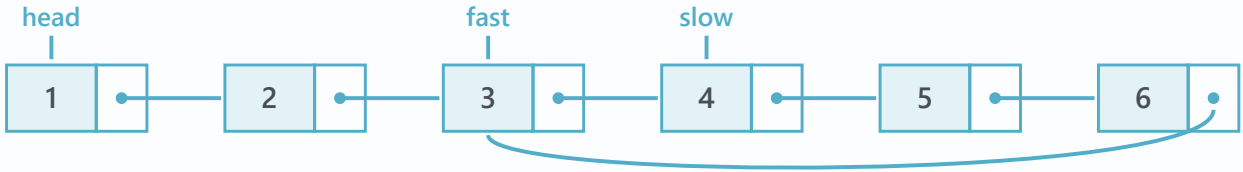
VARIABLES

`head` = **node(1)**

`slow` = **node(4)**

`fast` = **node(3)**

Code Trace: `hasCycle` (Floyd)



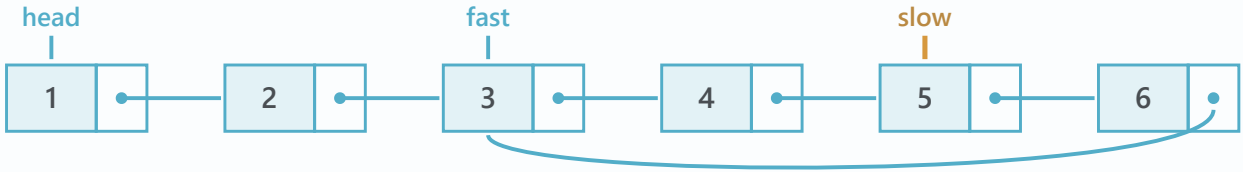
Round 4: fast (3) and fast.next (4) are not null.


```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
▶      while (fast != null &&
▶             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head = node(1)` `slow = node(4)` `fast = node(3)`

Code Trace: `hasCycle` (Floyd)



`slow` moves 1 step → node(5).

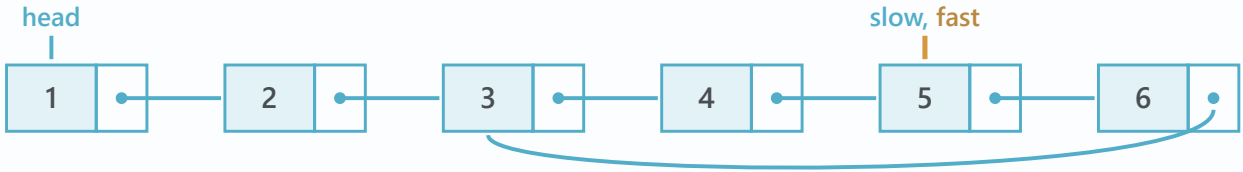
Step 15 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
▶          slow = slow.next;
6          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

- head = node(1)
- slow = node(5)
- fast = node(3)

Code Trace: `hasCycle` (Floyd)



`fast` moves 2 steps → node(5).

Step 16 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
▶          fast = fast.next.next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

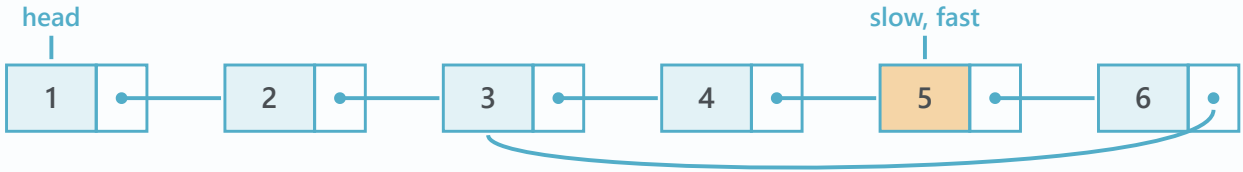
VARIABLES

`head` = **node(1)**

`slow` = **node(5)**

`fast` = **node(5)**

Code Trace: `hasCycle` (Floyd)



slow == fast (node 5) → **return true**. Inside the cycle, fast gains one node on slow each round; O(n) time, O(1) space.

Step 17 / 17

```
1  boolean hasCycle(Node head) {
2      Node slow = head, fast = head;
3      while (fast != null &&
4             fast.next != null) {
5          slow = slow.next;
6          fast = fast.next.next;
▶          if (slow == fast) return true;
8      }
9      return false;
10 }
```

VARIABLES

`head` = **node(1)**

`slow` = **node(5)**

`fast` = **node(5)**

Inserting at the Tail of a Circular List — O(1)



`addLast(1)` : create the new node.

Step 1 / 17

```
▶ Node temp = new Node(value);
2 if (tail == null) {
3     tail = temp;
4     tail.next = tail;
5 } else {
6     temp.next = tail.next;
7     tail.next = temp;
8     tail = temp;
9 }
10 size++;
```

VARIABLES

`tail = null` `size = 0`

`value = 1` `temp = node(1)`

Inserting at the Tail of a Circular List — O(1)



`tail == null` → the list is empty.

Step 2 / 17

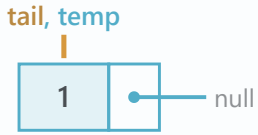
```
1  Node temp = new Node(value);
▶  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = null` `size = 0`

`value = 1` `temp = node(1)`

Inserting at the Tail of a Circular List — O(1)



`tail = temp` .

Step 3 / 17

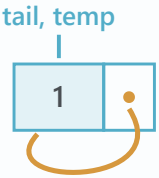
```
1  Node temp = new Node(value);
2  if (tail == null) {
▶      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(1)` `size = 0`

`value = 1` `temp = node(1)`

Inserting at the Tail of a Circular List — O(1)



`tail.next = tail` : the single node points to itself; the ring is closed at once.

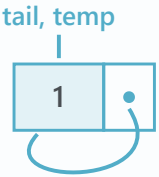

```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
▶      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(1)` `size = 0`

`value = 1` `temp = node(1)`

Inserting at the Tail of a Circular List — O(1)



`size++` .

Step 5 / 17

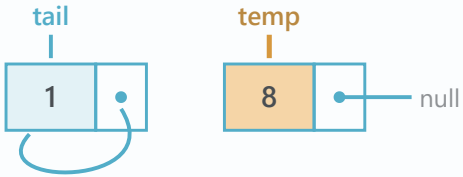
```
1 Node temp = new Node(value);
2 if (tail == null) {
3     tail = temp;
4     tail.next = tail;
5 } else {
6     temp.next = tail.next;
7     tail.next = temp;
8     tail = temp;
9 }
▶ size++;
```

VARIABLES

`tail = node(1)` `size = 1`

`value = 1` `temp = node(1)`

Inserting at the Tail of a Circular List — O(1)



`addLast(8)` : create the new node.

Step 6 / 17

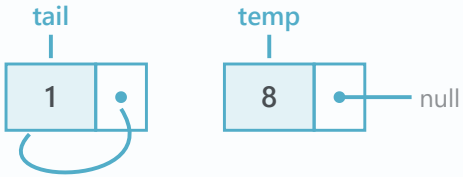
```
▶ Node temp = new Node(value);
2 if (tail == null) {
3     tail = temp;
4     tail.next = tail;
5 } else {
6     temp.next = tail.next;
7     tail.next = temp;
8     tail = temp;
9 }
10 size++;
```

VARIABLES

`tail = node(1)` `size = 1`

`value = 8` `temp = node(8)`

Inserting at the Tail of a Circular List — O(1)



The list is not empty → `else`.

Step 7 / 17

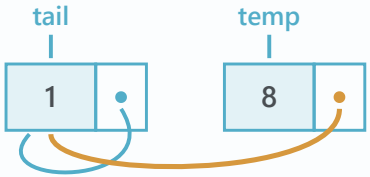
```
1  Node temp = new Node(value);
▶  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(1)` `size = 1`

`value = 8` `temp = node(8)`

Inserting at the Tail of a Circular List — O(1)



`temp.next = tail.next` : the new node points to the head (1).

Step 8 / 17

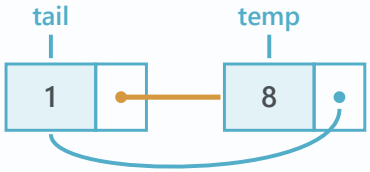
```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
▶      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(1)` `size = 1`

`value = 8` `temp = node(8)`

Inserting at the Tail of a Circular List — O(1)



`tail.next = temp` : the old tail (1) points to the new node.

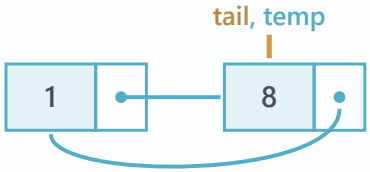

```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
▶      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(1)` `size = 1`

`value = 8` `temp = node(8)`

Inserting at the Tail of a Circular List — O(1)



`tail = temp` : tail is now 8.

Step 10 / 17

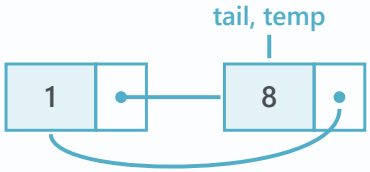
```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
▶      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(8)` `size = 1`

`value = 8` `temp = node(8)`

Inserting at the Tail of a Circular List — O(1)



`size++` .

Step 11 / 17

```
1 Node temp = new Node(value);
2 if (tail == null) {
3     tail = temp;
4     tail.next = tail;
5 } else {
6     temp.next = tail.next;
7     tail.next = temp;
8     tail = temp;
9 }
▶ size++;
```

VARIABLES

`tail = node(8)` `size = 2`

`value = 8` `temp = node(8)`

Inserting at the Tail of a Circular List — O(1)



`addLast(10)` : create the new node.

Step 12 / 17

```
▶ Node temp = new Node(value);
2 if (tail == null) {
3     tail = temp;
4     tail.next = tail;
5 } else {
6     temp.next = tail.next;
7     tail.next = temp;
8     tail = temp;
9 }
10 size++;
```

VARIABLES

`tail = node(8)` `size = 2`

`value = 10` `temp = node(10)`

Inserting at the Tail of a Circular List — O(1)



The list is not empty → `else`.

Step 13 / 17

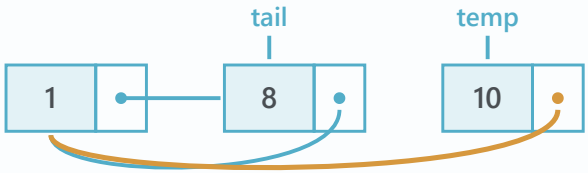
```
1  Node temp = new Node(value);
▶  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(8)` `size = 2`

`value = 10` `temp = node(10)`

Inserting at the Tail of a Circular List — O(1)



`temp.next = tail.next` : the new node points to the head (1).

Step 14 / 17

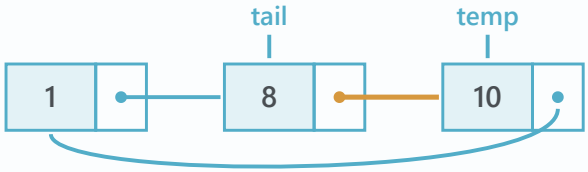
```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
▶      temp.next = tail.next;
7      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(8)` `size = 2`

`value = 10` `temp = node(10)`

Inserting at the Tail of a Circular List — O(1)



`tail.next = temp` : the old tail (8) points to the new node.

Step 15 / 17

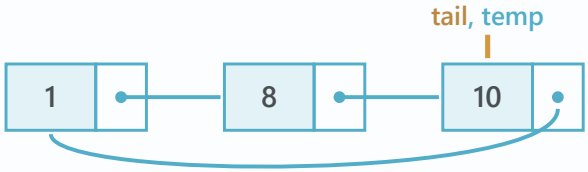
```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
▶      tail.next = temp;
8      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(8)` `size = 2`

`value = 10` `temp = node(10)`

Inserting at the Tail of a Circular List — O(1)



`tail = temp` : tail is now 10.

Step 16 / 17

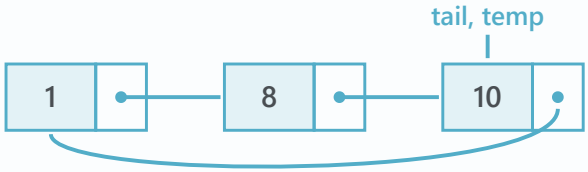
```
1  Node temp = new Node(value);
2  if (tail == null) {
3      tail = temp;
4      tail.next = tail;
5  } else {
6      temp.next = tail.next;
7      tail.next = temp;
▶      tail = temp;
9  }
10 size++;
```

VARIABLES

`tail = node(10)` `size = 2`

`value = 10` `temp = node(10)`

Inserting at the Tail of a Circular List — O(1)



`size++`. Result: 1 → 8 → 10 → (1). No walk: **O(1)**.

Step 17 / 17

```
1 Node temp = new Node(value);
2 if (tail == null) {
3     tail = temp;
4     tail.next = tail;
5 } else {
6     temp.next = tail.next;
7     tail.next = temp;
8     tail = temp;
9 }
▶ size++;
```

VARIABLES

`tail = node(10)` `size = 3`

`value = 10` `temp = node(10)`

Inserting and Deleting at the Head of a Circular List

O(1)

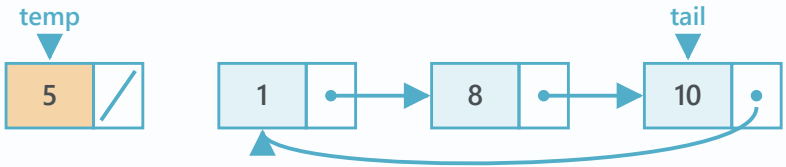
```
void addFirst(int value) {
    Node temp = new Node(value);
    if (tail == null) {
        tail = temp;
        tail.next = tail;
    } else {
        temp.next = tail.next;
        tail.next = temp;
    }   // tail unchanged
    size++;
}
```

O(1)

```
void removeFirst() {
    if (tail == null) return;
    Node head = tail.next;
    if (head == tail)   // single node
        tail = null;
    else
        tail.next = head.next;
    size--;
}
```

Inserting at the head is inserting at the tail minus the line `tail = temp`: the new node enters the ring at the same spot; only which node counts as the "tail" changes.

Code Trace: Circular `addFirst(5)`



`addFirst(5)` : a new node. Ring: 1 → 8 → 10 → (1), tail = 10.

Step 1 / 5

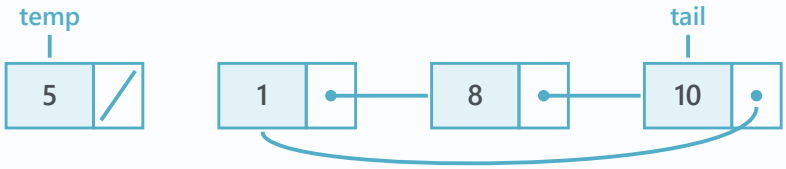
```
1  void addFirst(int value) {
▶      Node temp = new Node(value);
3      if (tail == null) {
4          tail = temp;
5          tail.next = tail;
6      } else {
7          temp.next = tail.next;
8          tail.next = temp;
9      }   // tail unchanged
10     size++;
11 }
```

VARIABLES

`tail = node(10)` `size = 3`

`value = 5` `temp = node(5)`

Code Trace: Circular `addFirst(5)`



The list is not empty → `else`.

Step 2 / 5

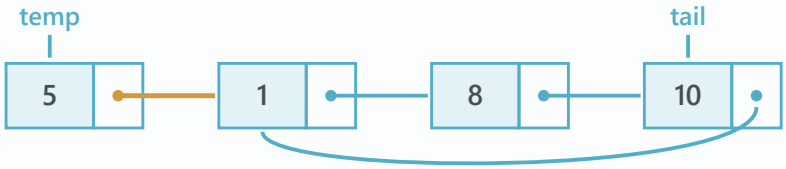
```
1  void addFirst(int value) {
2      Node temp = new Node(value);
▶      if (tail == null) {
4          tail = temp;
5          tail.next = tail;
6      } else {
7          temp.next = tail.next;
8          tail.next = temp;
9      }   // tail unchanged
10     size++;
11 }
```

VARIABLES

`tail = node(10)` `size = 3`

`value = 5` `temp = node(5)`

Code Trace: Circular `addFirst(5)`



`temp.next = tail.next` : 5 points to the old head (1).

Step 3 / 5

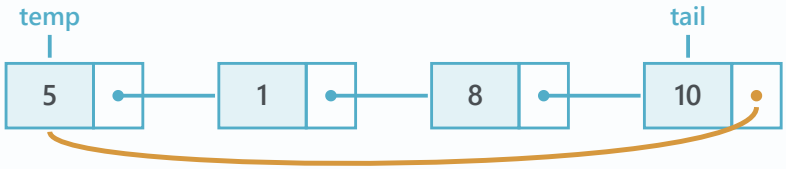
```
1  void addFirst(int value) {
2      Node temp = new Node(value);
3      if (tail == null) {
4          tail = temp;
5          tail.next = tail;
6      } else {
▶          temp.next = tail.next;
8          tail.next = temp;
9      }   // tail unchanged
10     size++;
11 }
```

VARIABLES

`tail` = **node(10)** `size` = **3**

`value` = **5** `temp` = **node(5)**

Code Trace: Circular `addFirst(5)`



`tail.next = temp` : 10 now points to 5 → 5 is the new head. `tail` doesn't change.

Step 4 / 5

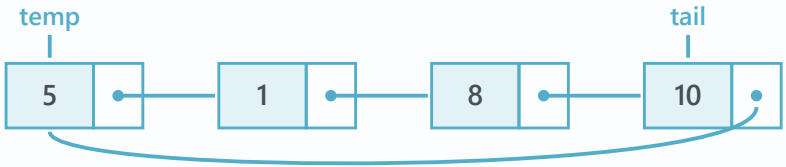
```
1  void addFirst(int value) {
2      Node temp = new Node(value);
3      if (tail == null) {
4          tail = temp;
5          tail.next = tail;
6      } else {
7          temp.next = tail.next;
▶          tail.next = temp;
9      }   // tail unchanged
10     size++;
11 }
```

VARIABLES

`tail = node(10)` `size = 3`

`value = 5` `temp = node(5)`

Code Trace: Circular `addFirst(5)`



`size++`. Ring: 5 → 1 → 8 → 10 → (5). **O(1)**; the only difference from inserting at the tail is the missing `tail = temp`.

Step 5 / 5

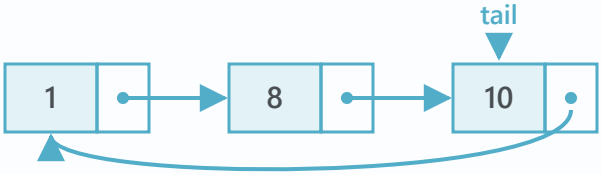
```
1  void addFirst(int value) {
2      Node temp = new Node(value);
3      if (tail == null) {
4          tail = temp;
5          tail.next = tail;
6      } else {
7          temp.next = tail.next;
8          tail.next = temp;
9      }   // tail unchanged
▶      size++;
11 }
```

VARIABLES

`tail = node(10)` `size = 4`

`value = 5` `temp = node(5)`

Code Trace: Circular `removeFirst()`



Ring: 1 → 8 → 10 → (1), tail = 10. Not empty.

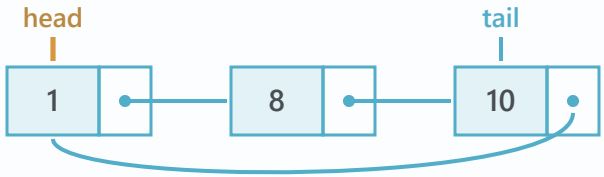
Step 1 / 5

```
1 void removeFirst() {
▶       if (tail == null) return;
3       Node head = tail.next;
4       if (head == tail)   // single node
5           tail = null;
6       else
7           tail.next = head.next;
8       size--;
9   }
```

VARIABLES

`tail = node(10)` `size = 3`

Code Trace: Circular `removeFirst()`



`head = tail.next` : the head node to delete (1).

Step 2 / 5

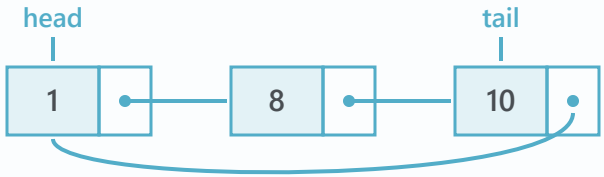
```
1 void removeFirst() {
2     if (tail == null) return;
▶     Node head = tail.next;
4     if (head == tail)   // single node
5         tail = null;
6     else
7         tail.next = head.next;
8     size--;
9 }
```

VARIABLES

`tail = node(10)` `size = 3`

`head = node(1)`

Code Trace: Circular `removeFirst()`



`head != tail` → more than one node.

Step 3 / 5

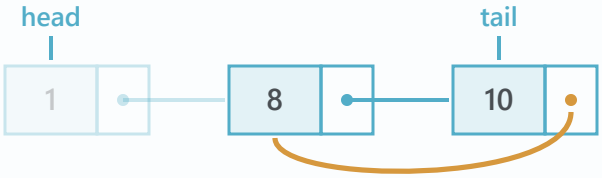
```
1 void removeFirst() {
2     if (tail == null) return;
3     Node head = tail.next;
▶     if (head == tail)   // single node
5         tail = null;
6     else
7         tail.next = head.next;
8     size--;
9 }
```

VARIABLES

`tail = node(10)` `size = 3`

`head = node(1)`

Code Trace: Circular `removeFirst()`



`tail.next = head.next` : 10 now points to 8; 1 has left the ring.

Step 4 / 5

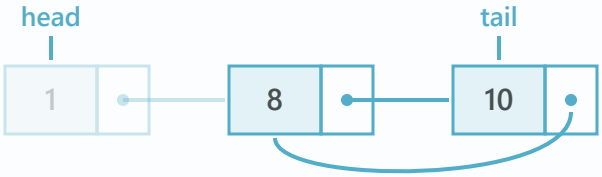
```
1 void removeFirst() {
2     if (tail == null) return;
3     Node head = tail.next;
4     if (head == tail)   // single node
5         tail = null;
6     else
▶         tail.next = head.next;
8     size--;
9 }
```

VARIABLES

`tail = node(10)` `size = 3`

`head = node(1)`

Code Trace: Circular `removeFirst()`



`size--`. Ring: 8 → 10 → (8). **O(1)**.

Step 5 / 5

```
1 void removeFirst() {
2     if (tail == null) return;
3     Node head = tail.next;
4     if (head == tail)   // single node
5         tail = null;
6     else
7         tail.next = head.next;
▶     size--;
9 }
```

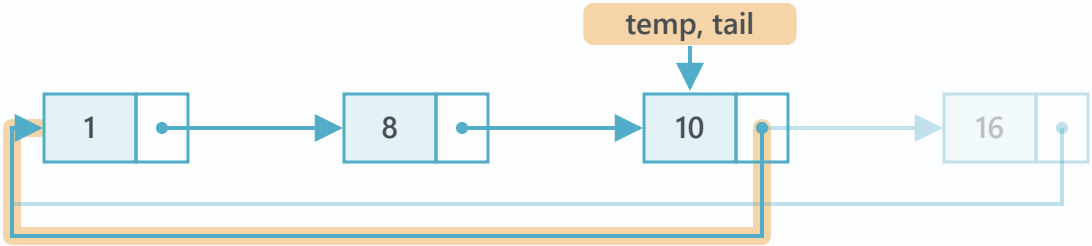
VARIABLES

`tail = node(10)` `size = 2`

`head = node(1)`

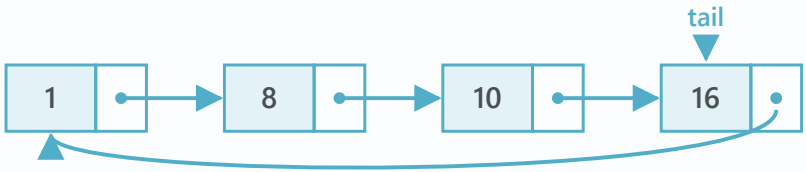
Deleting at the Tail of a Circular List — O(n)

```
void removeLast() {
    if (tail == null) return;
    if (tail.next == tail) tail = null;   // single node
    else {
        Node temp = tail.next;
        while (temp.next != tail)     // second to last
            temp = temp.next;
        temp.next = tail.next;     // link to the head
        tail = temp;
    }
    size--;
}
```



Finding the second-to-last node means walking around the ring: O(n). In a doubly linked circular list, `tail.prev` makes it O(1).

Code Trace: Circular `removeLast()`



`removeLast()` : ring 1 → 8 → 10 → 16 → (1), tail = 16. Not empty.

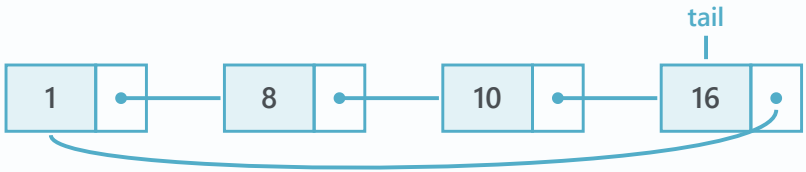
Step 1 / 11

```
▶ if (tail == null) return;
2 if (tail.next == tail) tail = null;   // single node
3 else {
4     Node temp = tail.next;
5     while (temp.next != tail)  // second to last
6         temp = temp.next;
7     temp.next = tail.next;  // link to the head
8     tail = temp;
9 }
10 size--;
```

VARIABLES

`tail = node(16)` `size = 4`

Code Trace: Circular `removeLast()`



`tail.next != tail` → not a single node.

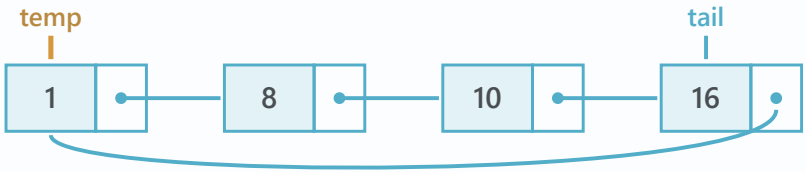
Step 2 / 11

```
1  if (tail == null) return;
▶  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
5      while (temp.next != tail)  // second to last
6          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail = node(16)` `size = 4`

Code Trace: Circular `removeLast()`



`temp = tail.next` : start at the head (1).

Step 3 / 11

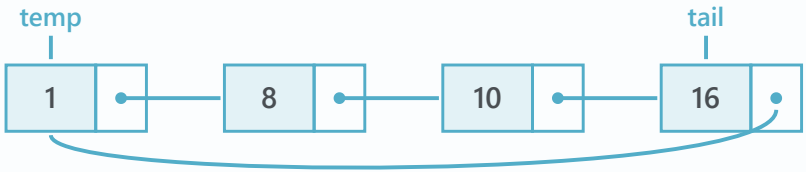
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
▶      Node temp = tail.next;
5      while (temp.next != tail)  // second to last
6          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail = node(16)` `size = 4`

`temp = node(1)`

Code Trace: Circular `removeLast()`



node(1).next = 8 ≠ tail → advance.

Step 4 / 11

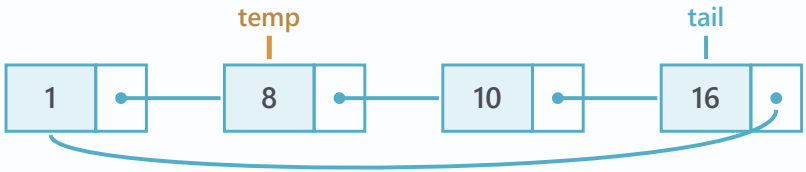
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
▶      while (temp.next != tail)  // second to last
6          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail` = **node(16)** `size` = **4**

`temp` = **node(1)**

Code Trace: Circular `removeLast()`



`temp = temp.next` → node(8).

Step 5 / 11

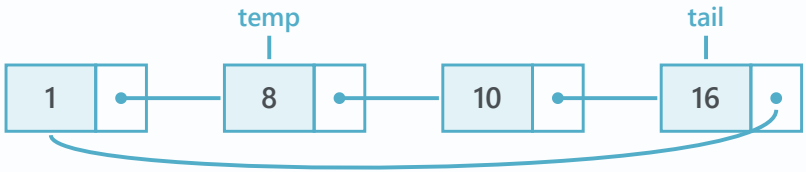
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
5      while (temp.next != tail)  // second to last
▶          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail = node(16)` `size = 4`

`temp = node(8)`

Code Trace: Circular `removeLast()`



node(8).next = 10 ≠ tail → advance.

Step 6 / 11

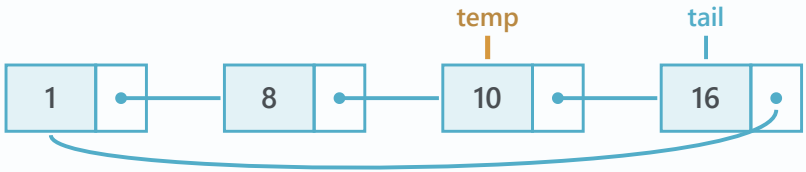
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
▶      while (temp.next != tail)  // second to last
6          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail` = **node(16)** `size` = **4**

`temp` = **node(8)**

Code Trace: Circular `removeLast()`



`temp = temp.next` → node(10).

Step 7 / 11

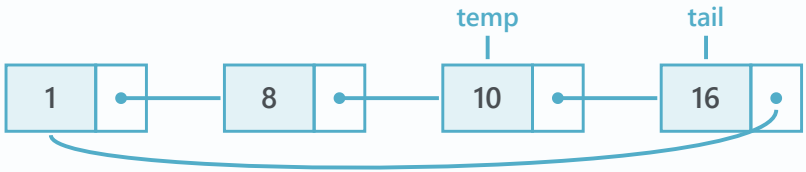
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
5      while (temp.next != tail)  // second to last
▶          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail = node(16)` `size = 4`

`temp = node(10)`

Code Trace: Circular `removeLast()`



node(10).next == tail → temp is at the second-to-last node.

Step 8 / 11

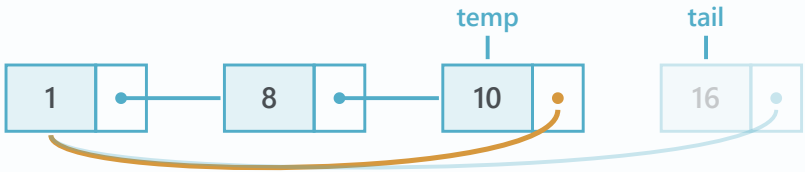
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
▶      while (temp.next != tail)  // second to last
6          temp = temp.next;
7      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail = node(16)` `size = 4`

`temp = node(10)`

Code Trace: Circular `removeLast()`



`temp.next = tail.next` : 10 points to the head (1); 16 has left the ring.

Step 9 / 11

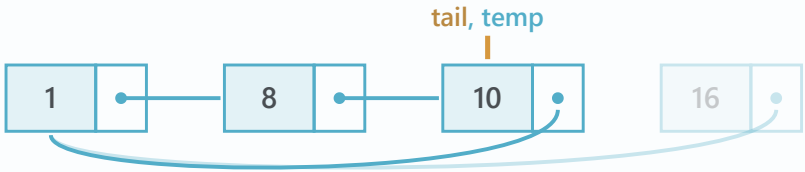
```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
5      while (temp.next != tail)  // second to last
6          temp = temp.next;
▶      temp.next = tail.next;  // link to the head
8      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail` = **node(16)** `size` = **4**

`temp` = **node(10)**

Code Trace: Circular `removeLast()`



`tail = temp` : tail is now 10.

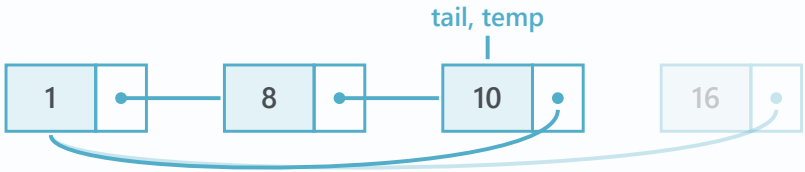
Step 10 / 11

```
1  if (tail == null) return;
2  if (tail.next == tail) tail = null;   // single node
3  else {
4      Node temp = tail.next;
5      while (temp.next != tail)  // second to last
6          temp = temp.next;
7      temp.next = tail.next;  // link to the head
▶      tail = temp;
9  }
10 size--;
```

VARIABLES

`tail = node(10)` `size = 4` `temp = node(10)`

Code Trace: Circular `removeLast()`



`size--`. Ring: 1 → 8 → 10 → (1). We walked to find the previous node: **O(n)**.

Step 11 / 11

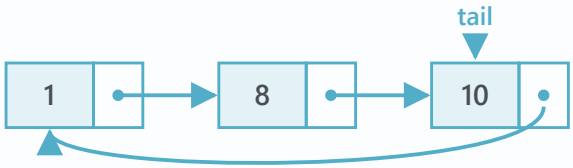
```
1 if (tail == null) return;
2 if (tail.next == tail) tail = null;   // single node
3 else {
4     Node temp = tail.next;
5     while (temp.next != tail)  // second to last
6         temp = temp.next;
7     temp.next = tail.next;  // link to the head
8     tail = temp;
9 }
▶ size--;
```

VARIABLES

`tail = node(10)` `size = 3`

`temp = node(10)`

Code Trace: Round-Robin with `rotate()`



Ring 1 → 8 → 10 → (1), `tail` = 10, head = `tail.next` = 1. Three jobs take turns on the processor.

Step 1 / 9

```
1  void rotate() {              // Goodrich: rotate()
2      if (tail != null)
3          tail = tail.next;    // old head becomes the tail
4  }
5  // round-robin: serve the head each round
▶  for (int round = 1; round <= 4; round++) {
7      System.out.print(tail.next.data + " ");
8      list.rotate();
9  }
```

VARIABLES

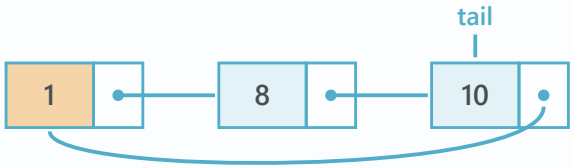
`tail = node(10)`

CALL STACK

`main()`

OUTPUT

Code Trace: Round-Robin with `rotate()`



Round 1: the job at the head, **1**, is served (printed).

Step 2 / 9

```
1 void rotate() {              // Goodrich: rotate()
2     if (tail != null)
3         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
▶     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(10)` `round = 1`

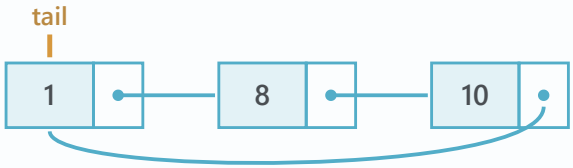
CALL STACK

`main()`

OUTPUT

```
1
```

Code Trace: Round-Robin with `rotate()`



`rotate()` : `tail = tail.next` → tail = 1. No link changes, one pointer moves: **O(1)**. New head: 8.

Step 3 / 9

```
1 void rotate() {              // Goodrich: rotate()
▶     if (tail != null)
▶         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
7     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(1)` `round = 1`

CALL STACK

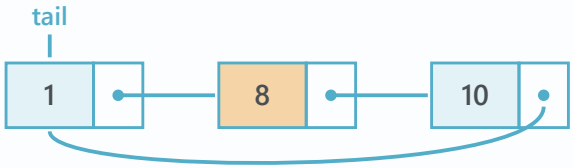
`rotate()`

`main()`

OUTPUT

```
1
```

Code Trace: Round-Robin with `rotate()`



Round 2: the job at the head, **8**, is served (printed).

Step 4 / 9

```
1 void rotate() {              // Goodrich: rotate()
2     if (tail != null)
3         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
▶     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(1)` `round = 2`

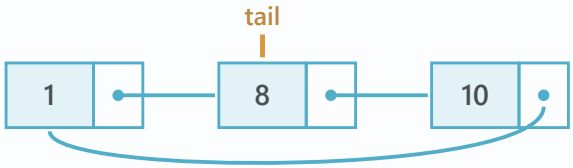
CALL STACK

`main()`

OUTPUT

```
1 8
```

Code Trace: Round-Robin with `rotate()`



`rotate()` : `tail = tail.next` → tail = 8. No link changes, one pointer moves: **O(1)**. New head: 10.

Step 5 / 9

```
1 void rotate() {              // Goodrich: rotate()
▶     if (tail != null)
▶         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
7     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(8)` `round = 2`

CALL STACK

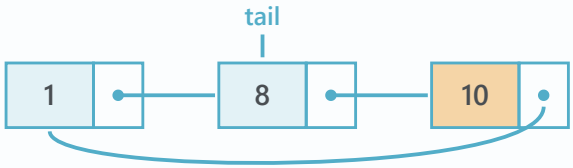
`rotate()`

`main()`

OUTPUT

```
1 8
```

Code Trace: Round-Robin with `rotate()`



Round 3: the job at the head, **10**, is served (printed).

Step 6 / 9

```
1 void rotate() {              // Goodrich: rotate()
2     if (tail != null)
3         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
▶     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(8)` `round = 3`

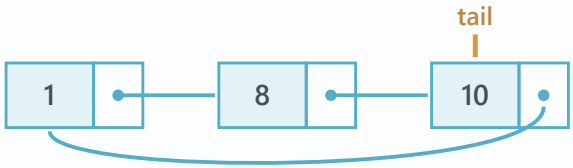
CALL STACK

`main()`

OUTPUT

```
1 8 10
```

Code Trace: Round-Robin with `rotate()`



`rotate()` : `tail = tail.next` → tail = 10. No link changes, one pointer moves: **O(1)**. New head: 1.

Step 7 / 9

```
1  void rotate() {              // Goodrich: rotate()
▶      if (tail != null)
▶          tail = tail.next;    // old head becomes the tail
4  }
5  // round-robin: serve the head each round
6  for (int round = 1; round <= 4; round++) {
7      System.out.print(tail.next.data + " ");
8      list.rotate();
9  }
```

VARIABLES

`tail = node(10)` `round = 3`

CALL STACK

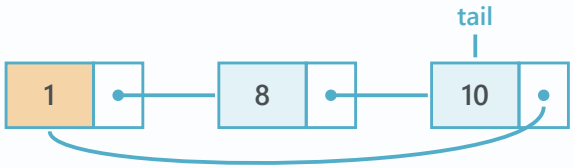
`rotate()`

`main()`

OUTPUT

```
1 8 10
```

Code Trace: Round-Robin with `rotate()`



Round 4: the job at the head, **1**, is served (printed).

Step 8 / 9

```
1 void rotate() {              // Goodrich: rotate()
2     if (tail != null)
3         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
▶     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(10)` `round = 4`

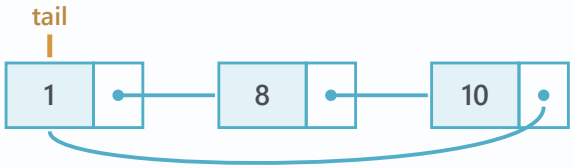
CALL STACK

`main()`

OUTPUT

```
1 8 10 1
```

Code Trace: Round-Robin with `rotate()`



`rotate()` : `tail = tail.next` → tail = 1. No link changes, one pointer moves: **O(1)**. Output **1 8 10 1**: the jobs take fair turns.

Step 9 / 9

```
1 void rotate() {              // Goodrich: rotate()
▶     if (tail != null)
▶         tail = tail.next;    // old head becomes the tail
4 }
5 // round-robin: serve the head each round
6 for (int round = 1; round <= 4; round++) {
7     System.out.print(tail.next.data + " ");
8     list.rotate();
9 }
```

VARIABLES

`tail = node(1)` `round = 4`

CALL STACK

`rotate()`

`main()`

OUTPUT

```
1 8 10 1
```

SUMMARY

Array or Linked List?



Criterion	Array	Linked list
Access the i-th item	O(1): compute the address	O(n): walk from the head
Insert / delete at the front	O(n): shifting	O(1)
Insert after a known position	O(n)	O(1): two links
Size	fixed; growing costs an O(n) copy	dynamic; one `new` per insertion
Overhead / cache	unused capacity; cache-friendly	a reference per node; scattered access

Frequent random access and a fixed size → array. Frequent insertion/deletion at the ends and an unpredictable size → linked list (stack, queue).

SUMMARY

Common Mistakes with Linked Lists

No empty-list check

`head.data` while `head` is null → `NullPointerException`.

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

Linking in the wrong order

If `head` changes first, the old list is lost.

```
newNode.next = head;
head = newNode;
```

Advancing `head` during a traversal

After printing, the list is unreachable.

```
Node temp = head;
```

Forgetting `tail`

After the only node is deleted, `tail` still points to it.

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

One direction in a doubly linked list

If `prev` isn't updated, backward traversal breaks.

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

`!= null` in a circular list

A ring has no null → infinite loop.

```
while (p != tail) …
```

SUMMARY

Operation Costs Across Linked Lists

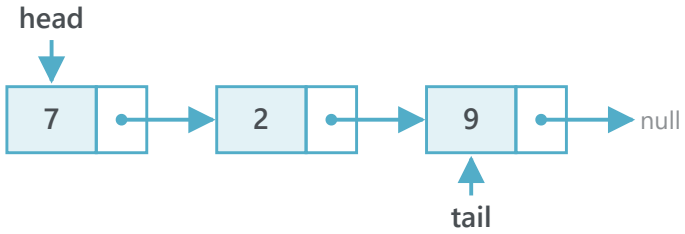
Operation	Singly (head only)	Singly (head + tail)	Doubly (head + tail)	Circular (tail only)
Insert at head	O(1)	O(1)	O(1)	O(1)
Insert at tail	O(n)	O(1)	O(1)	O(1)
Delete at head	O(1)	O(1)	O(1)	O(1)
Delete at tail	O(n)	O(n)	O(1)	O(n)
Search / i-th item	O(n)	O(n)	O(n)	O(n)
Delete a known node	O(n)*	O(n)*	O(1)	O(n)*

* Finding the previous node takes a walk. Inserting *after* a known node is O(1) in every kind. In an array, by contrast, reaching the i-th item is O(1) and inserting at the front O(n).

SUMMARY

Try It: Singly Linked List

addFirst | addLast | removeFirst | removeLast | reset | ☑ tail pointer



Change the list with the buttons; the status line shows each cost.

For more, the simulator: Linked List Simulator (insert, delete, and search step by step). The stack and queue simulators can also be backed by a linked list: Stack · Queue

SUMMARY

Sample Problems

1. What does it take to make inserting at the tail of a singly linked list O(1)? Which operation is still O(n)?
2. What happens if the two lines of `addFirst` are swapped (`head = newNode;` first)?
3. How do you find the middle of a list in one pass, without computing its length?
4. How can you tell whether a singly linked list has a cycle (the last node linked back to an earlier one)?
5. What does it cost to delete the `tail` node of a doubly linked list? Why is it different for a singly linked list?
6. Why does a circular list keep `tail` rather than `head` ?

▶ **Show answers**

SUMMARY

Exercises

1. In the list 3 → 5 → 7, what is the list after `head.next = head.next.next;`? What happens to 5?
2. What does `int mystery(Node d) { return d == null ? 0 : d.data + mystery(d.next); }` return for 2 → 4 → 6?
3. How do you find the k-th item from the end of a singly linked list in one pass, without knowing its length?
4. In a doubly linked list with sentinels, how many link assignments does inserting into an empty list take? Why is no `if` needed?
5. Circular list 1 → 8 → 10 (tail = 10). After two calls to `rotate()`, which item is at the head?
6. How many comparisons and swaps does `sort()` make on the list 3 → 1 → 2?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Lafore, *Data Structures and Algorithms in Java*, 2nd ed., 2002.
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.

The End

Chapter 5: Linked Lists

Next deck: Chapter 6: The List ADT and Java Collections · Study questions · Simulators: Linked List Simulator, Doubly Linked List Simulator, Circular Linked List Simulator · Glossary