Chapter 2:**Linked List (Continued) Circular Linked List.**

- BITE1523: COMPUTER GAME PROGRAMMING

- HAMZAH ASYRANI SULAIMAN

Recap: Types of data structure

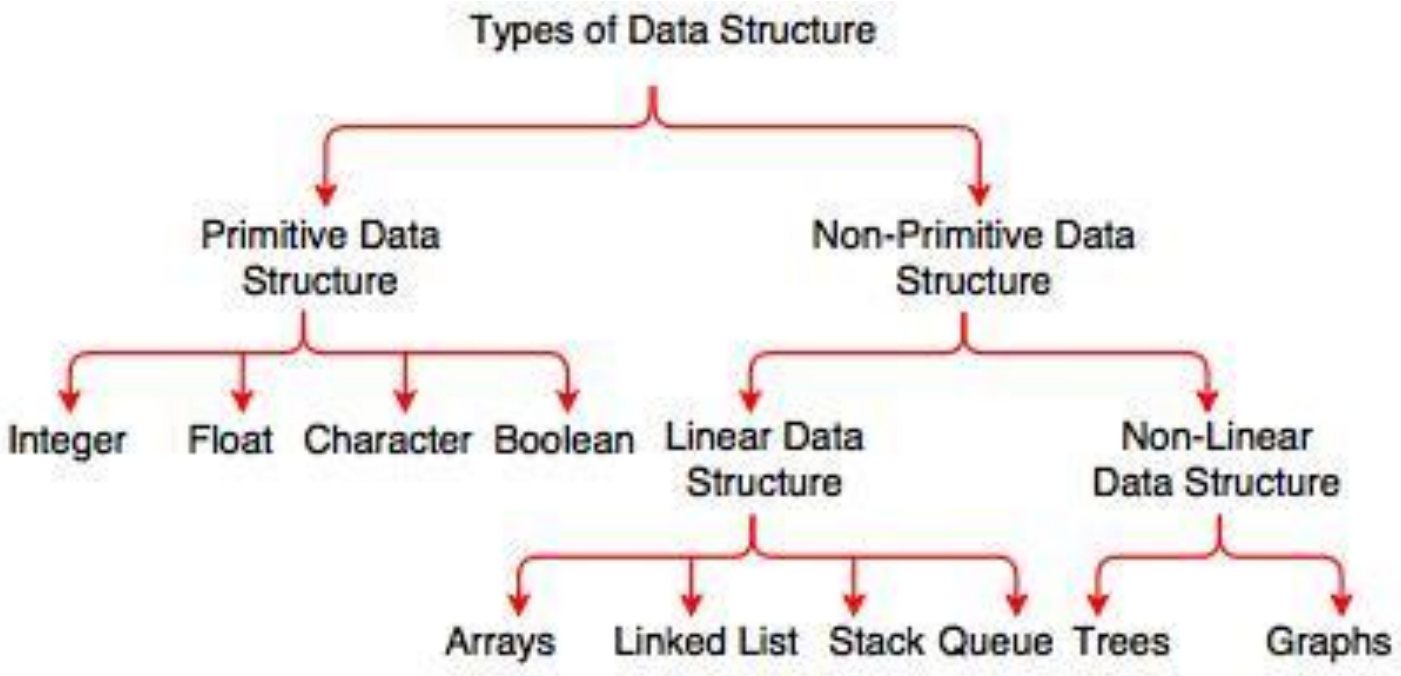


Fig. Types of Data Structure

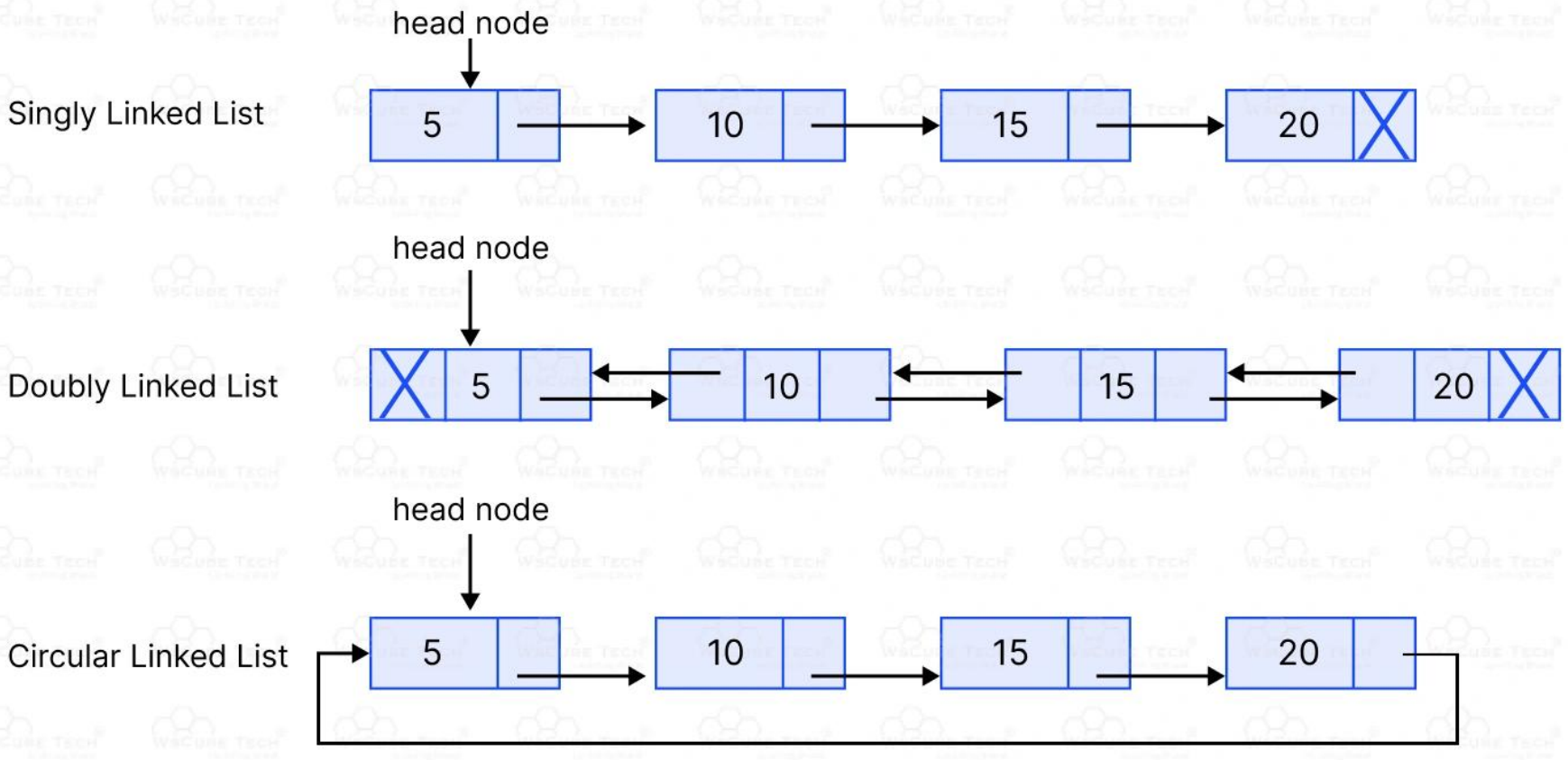
Types of Linked List

Simple/singly Linked List – Item navigation is forward only.

Doubly Linked List – Items can be navigated forward and backward.

Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Types of linked list

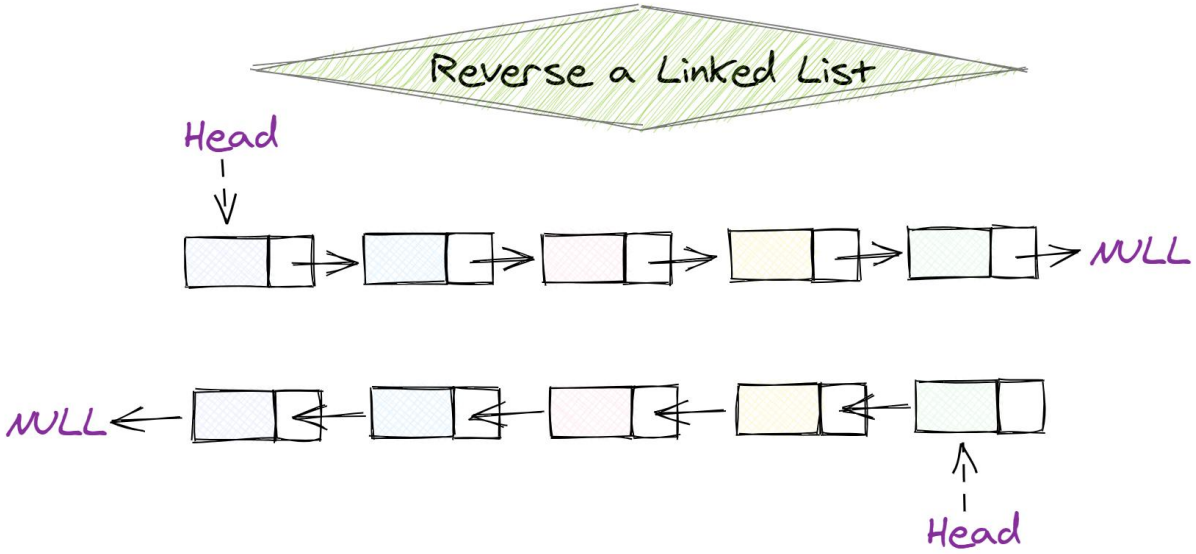


We will use this original code

```
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
};

void print_list(Node* n) {
    cout << "\nPrinting new list..." << endl;
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}
int main() {
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 10;
    head->next = second;

    second->data = 20;
    second->next = third;

    third->data = 30;
    third->next = NULL;

    print_list(head);
}
```

Reverse Linked List (Iterative)

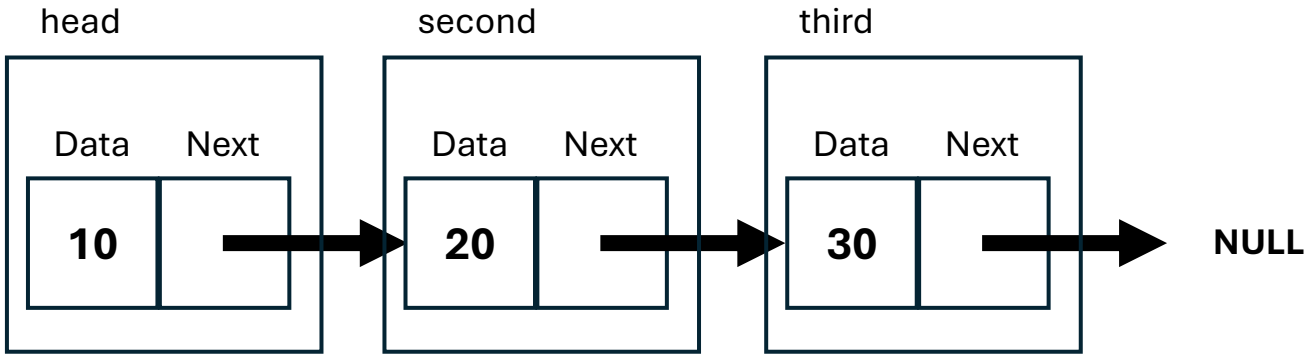
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // Simpan node seterusnya
        current->next = prev;  // Balikkan arah pointer

        // Gerakkan "prev" dan "current" ke depan
        prev = current;
        current = next;
    }

    return prev; // "prev" jadi kepala baru
}
```

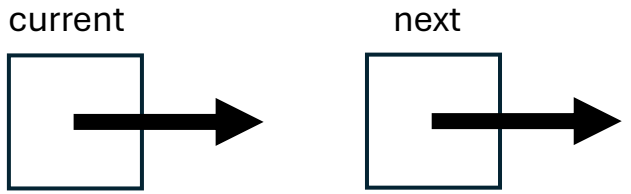


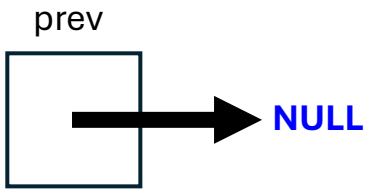
Preparation

- **prev**: Simpan node sebelumnya (mula dengan NULL sebab kepala baru nanti tak ada next).
- **current**: Pegang node semasa (start dari head).
- **next**: Tempat parking sementara untuk node seterusnya, supaya kita tak hilang rantaian asal.

```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;        // 3 Gerak 'prev' ke 'current'
        current = next;        // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



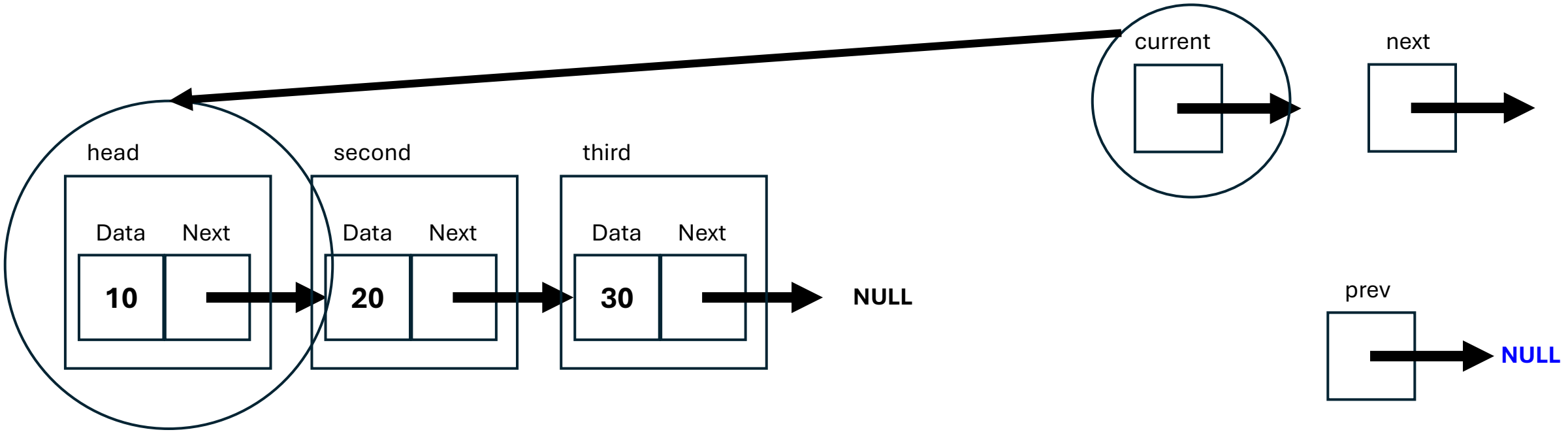




Preparation

- **prev**: Simpan node sebelumnya (mula dengan NULL sebab kepala baru nanti tak ada next).
- **current**: Pegang node semasa (start dari head).
- **next**: Tempat parking sementara untuk node seterusnya, supaya kita tak hilang rantaian asal.

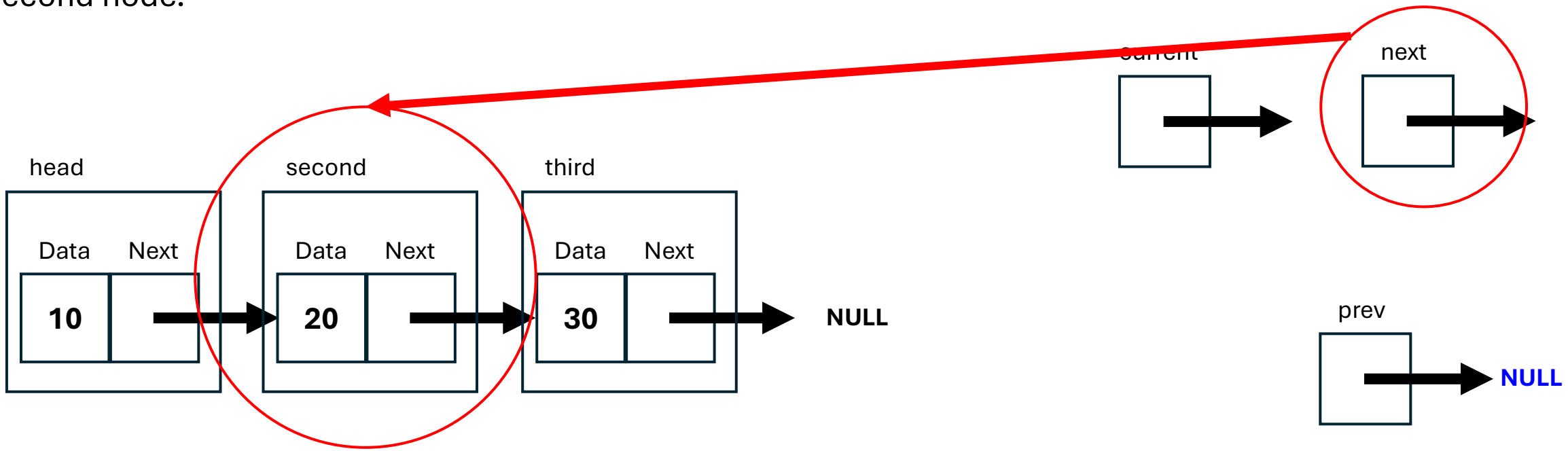
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;        // 3 Gerak 'prev' ke 'current'
        current = next;        // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



1st. LOOP

- In while loop, we will keep check whether **current** node (which is head now) is not **NULL**< otherwise we will in the loop.
- So we will enter the loop and start with appoint next pointer to what has been pointed by **current** (which is head).
- So current (head) next is pointed to second node.

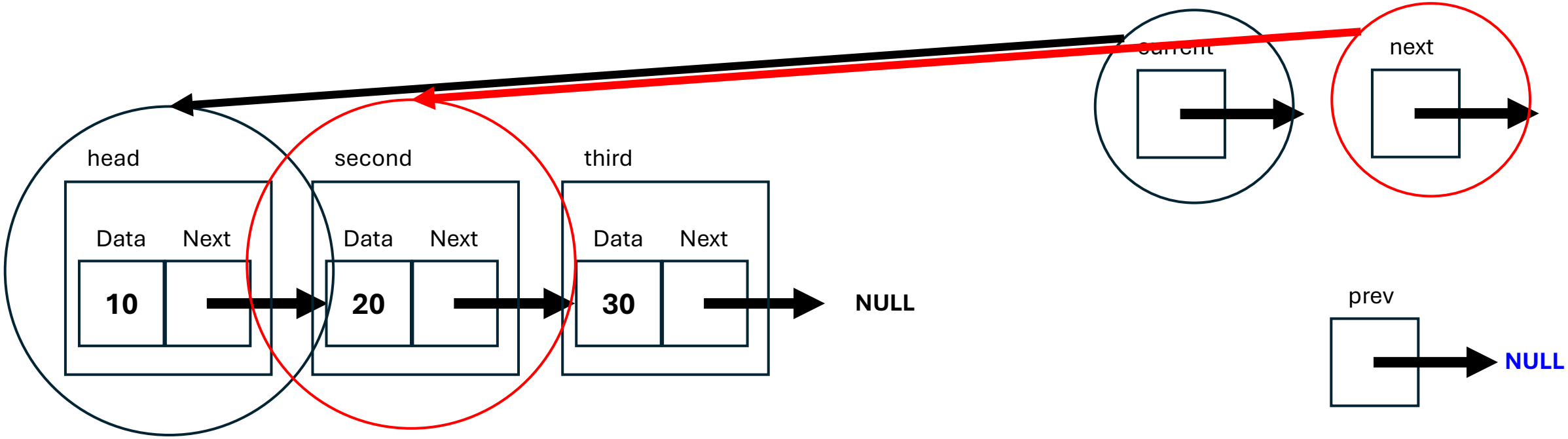
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;   // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



1st. LOOP

- This one is a little bit tricky; we will reverse what the node is pointed

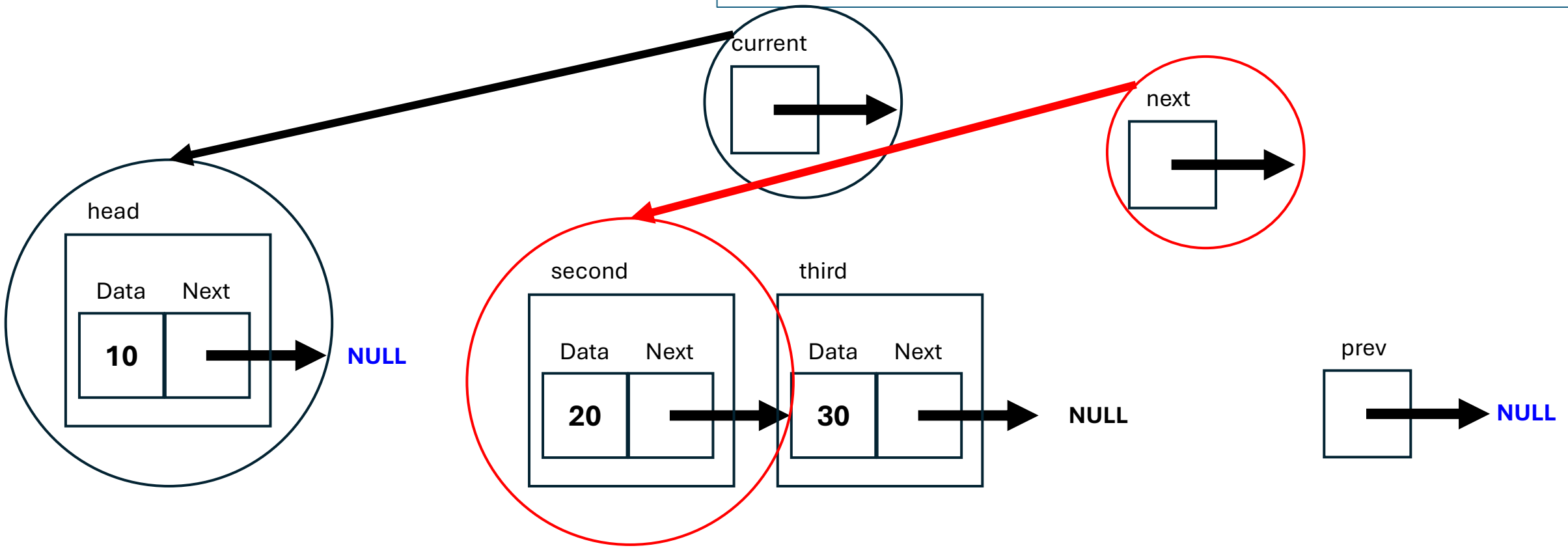
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;        // 3 Gerak 'prev' ke 'current'
        current = next;        // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



1st. LOOP

- This one is a little bit tricky; we will reverse what the node is pointed

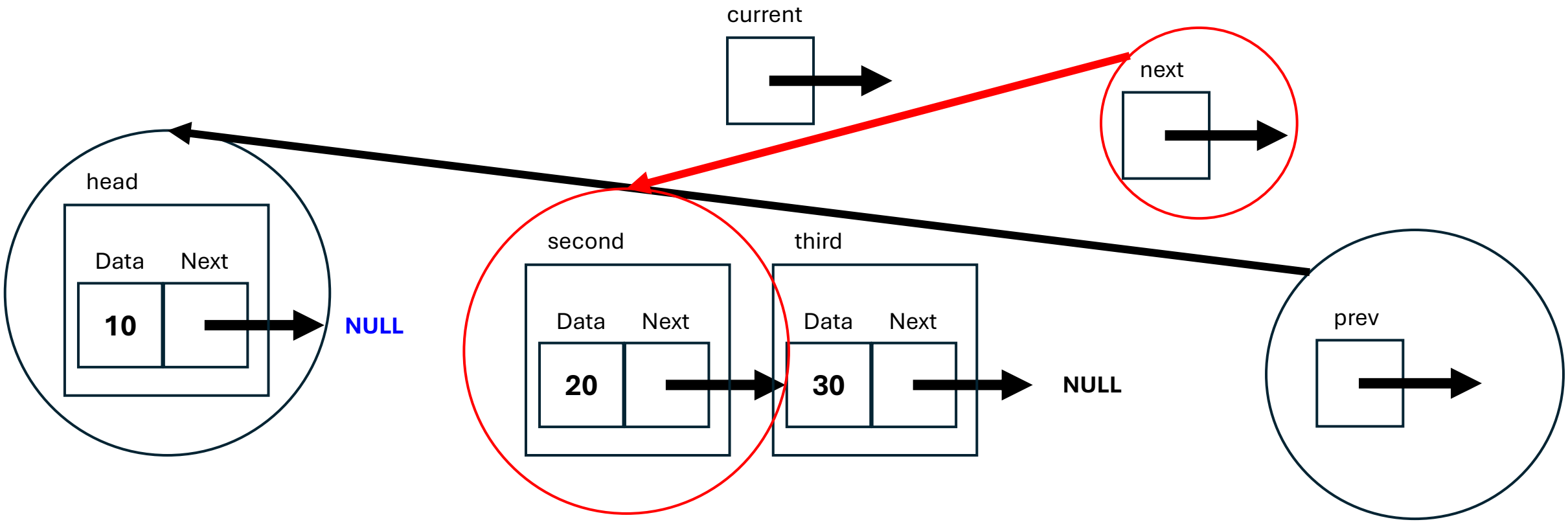
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;        // 3 Gerak 'prev' ke 'current'
        current = next;        // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



1st. LOOP

- This one is a little bit tricky; we will reverse what the node is pointed

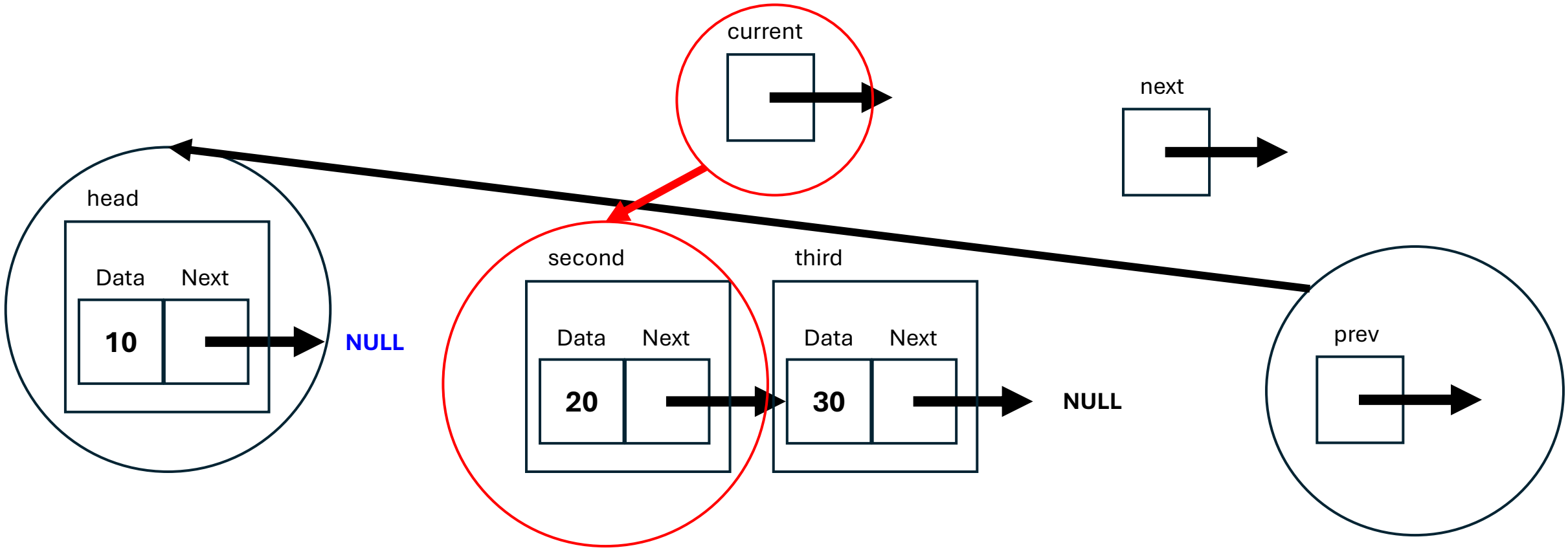
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;          // 3 Gerak 'prev' ke 'current'
        current = next;          // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



1st. LOOP

- This one is a little bit tricky; we will reverse what the node is pointed

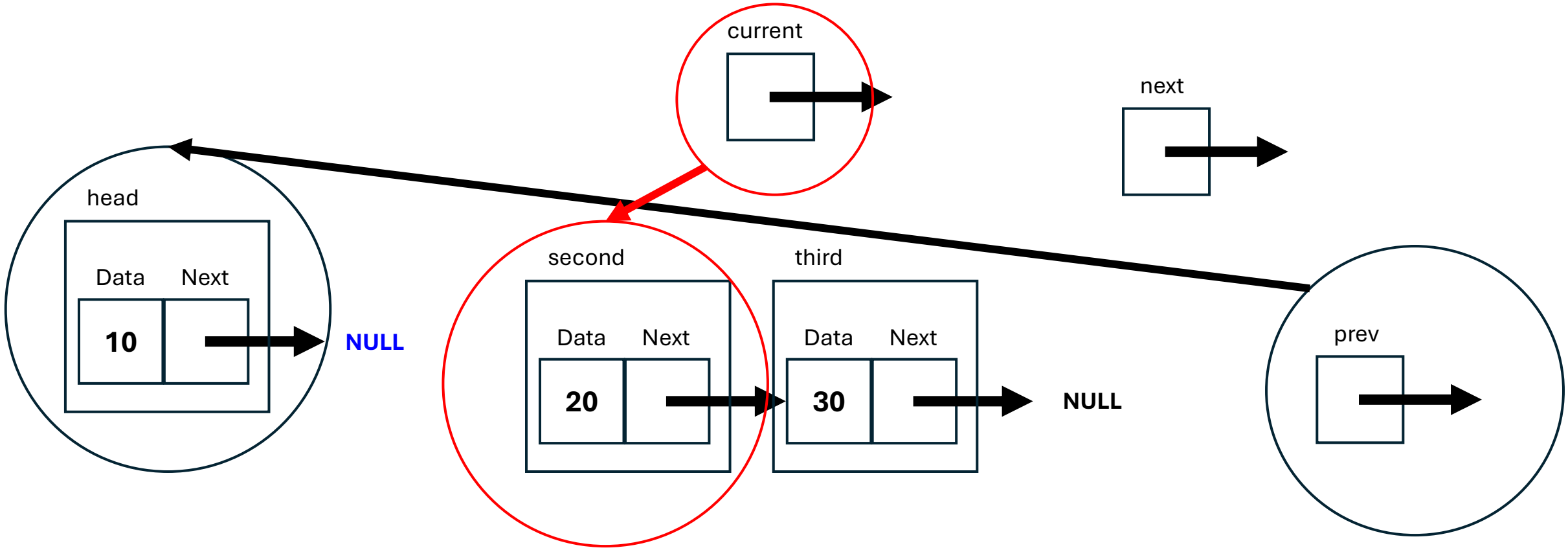
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



1st. LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```

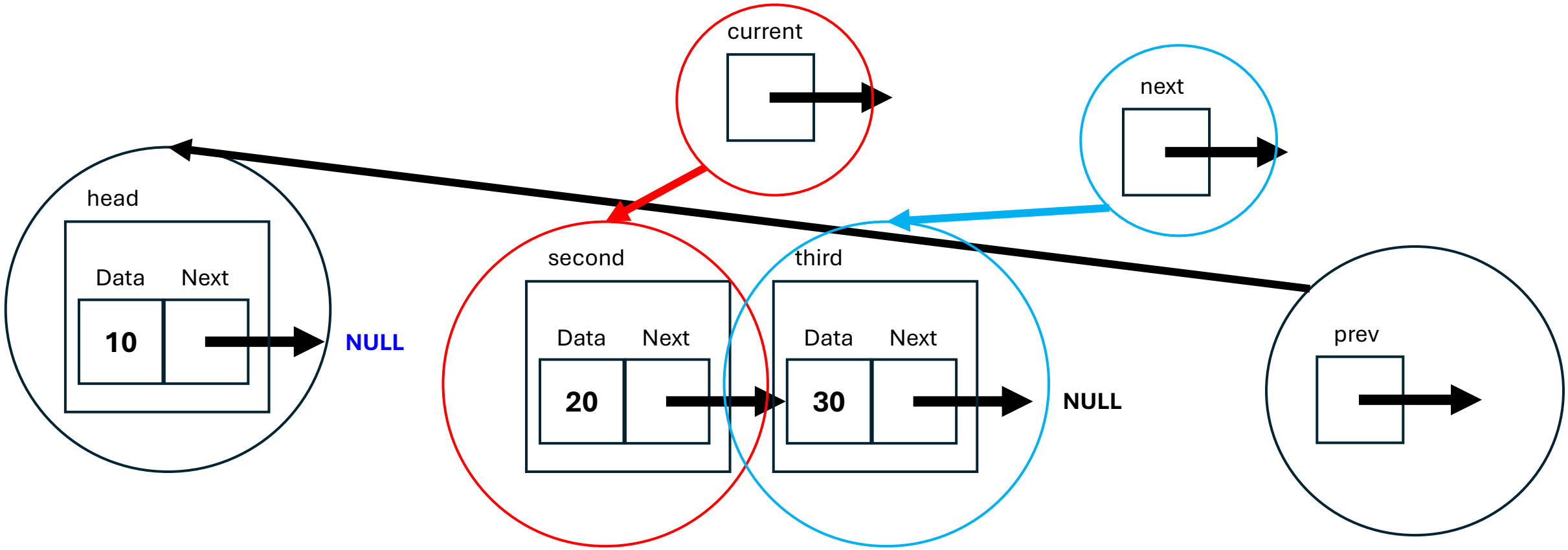


Second loop

2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

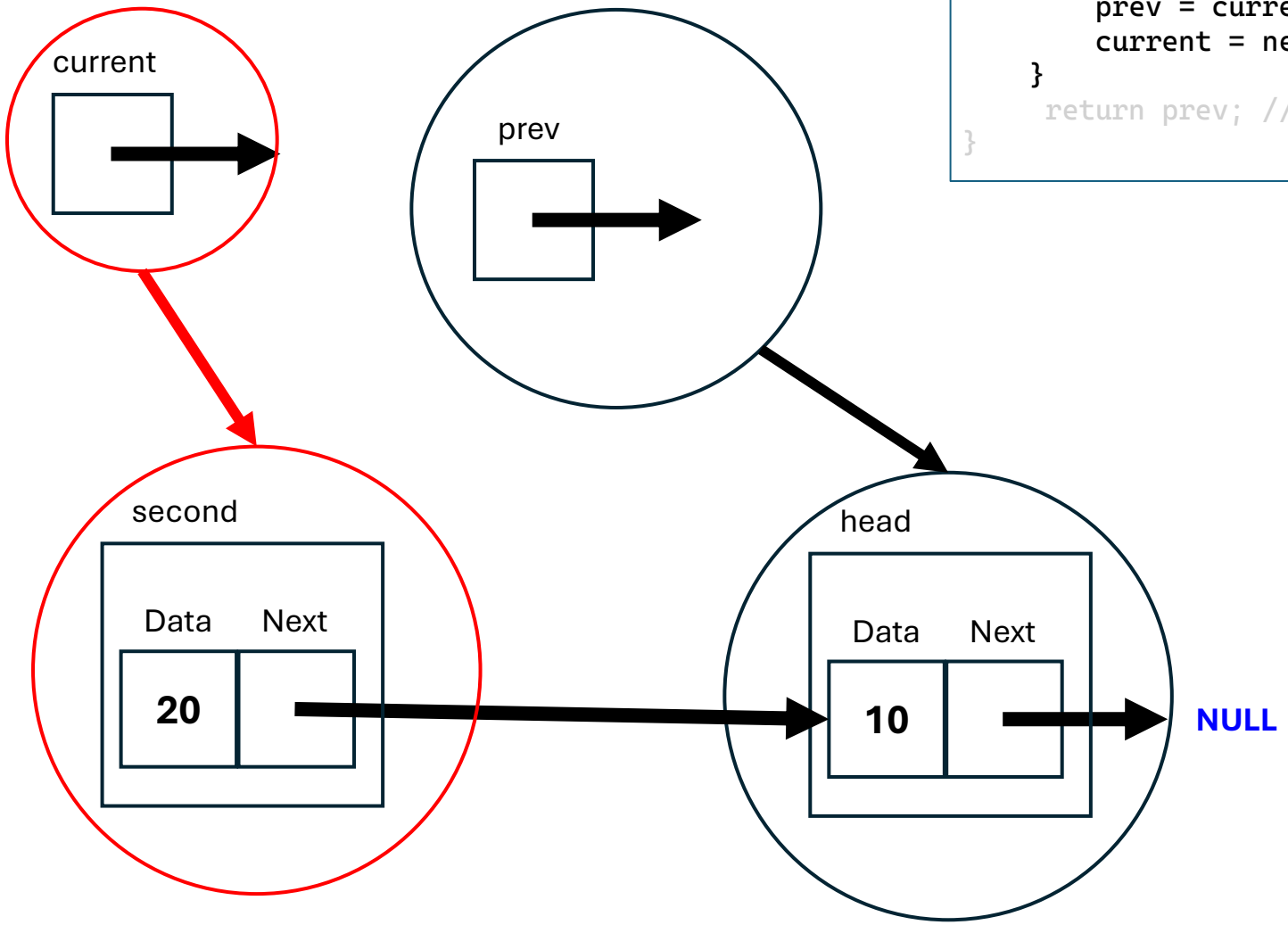
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;          // 3 Gerak 'prev' ke 'current'
        current = next;          // 4 Gerak 'current' ke node seterusnya
    }
     return prev; // 5 'prev' kini jadi kepala baru
}
```

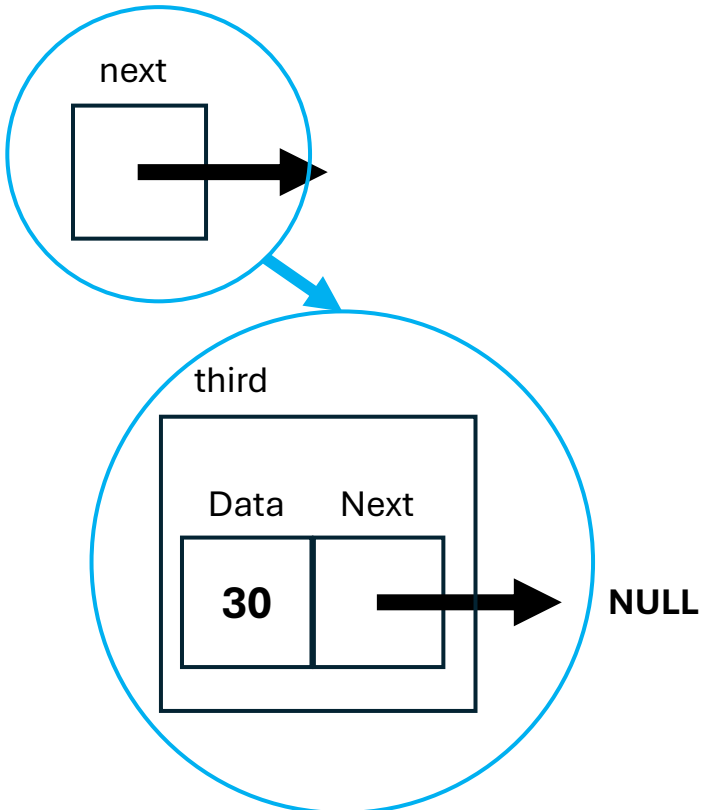


2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```

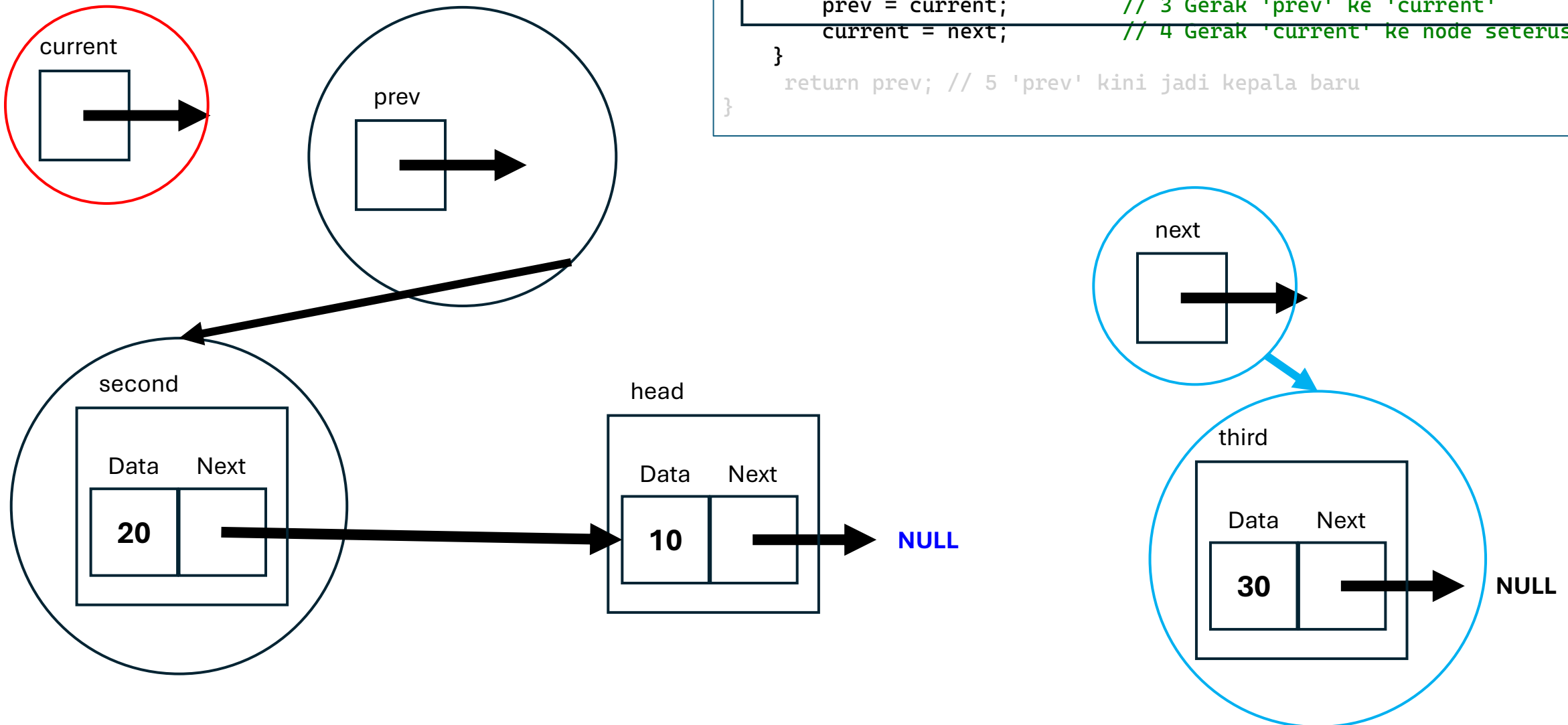




2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

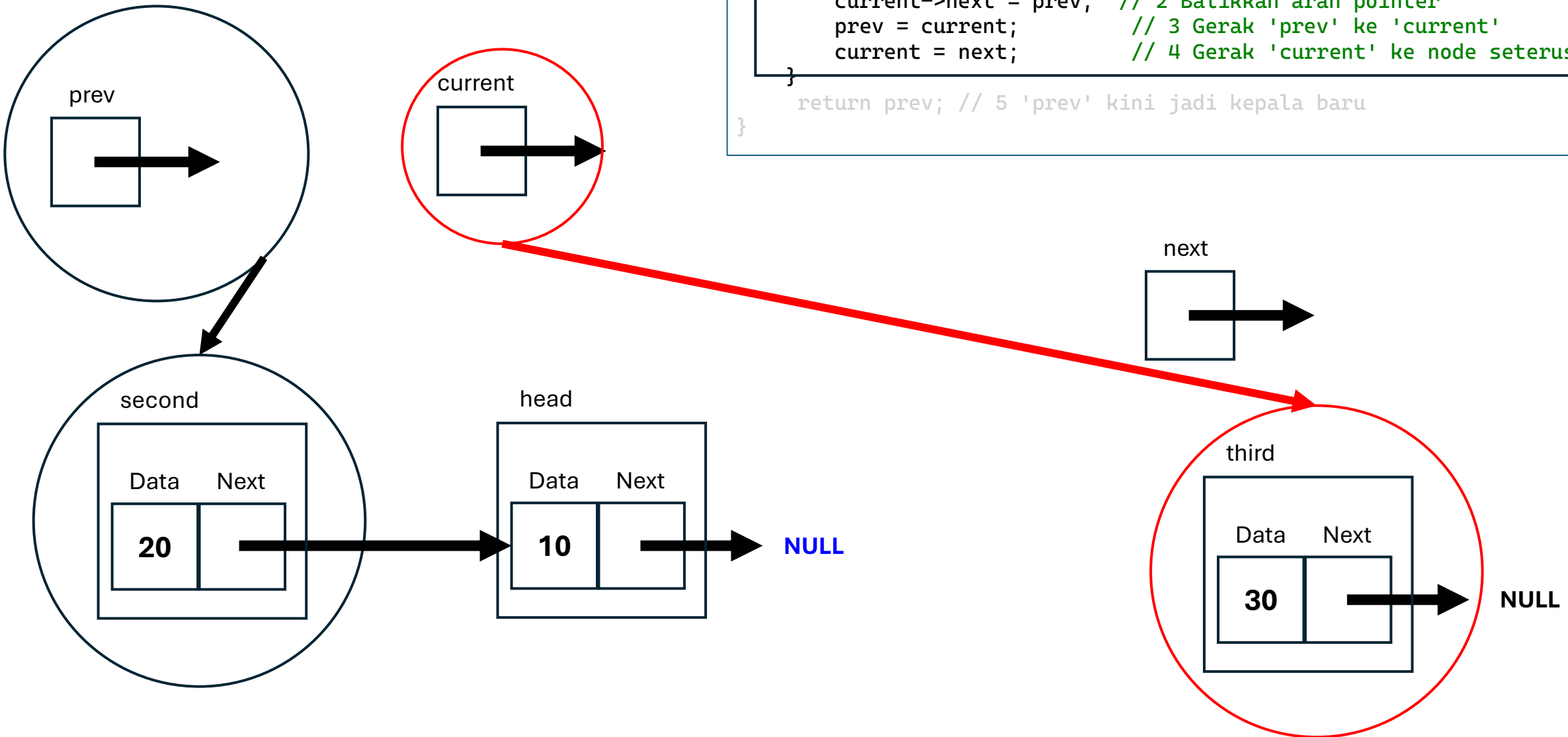
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```

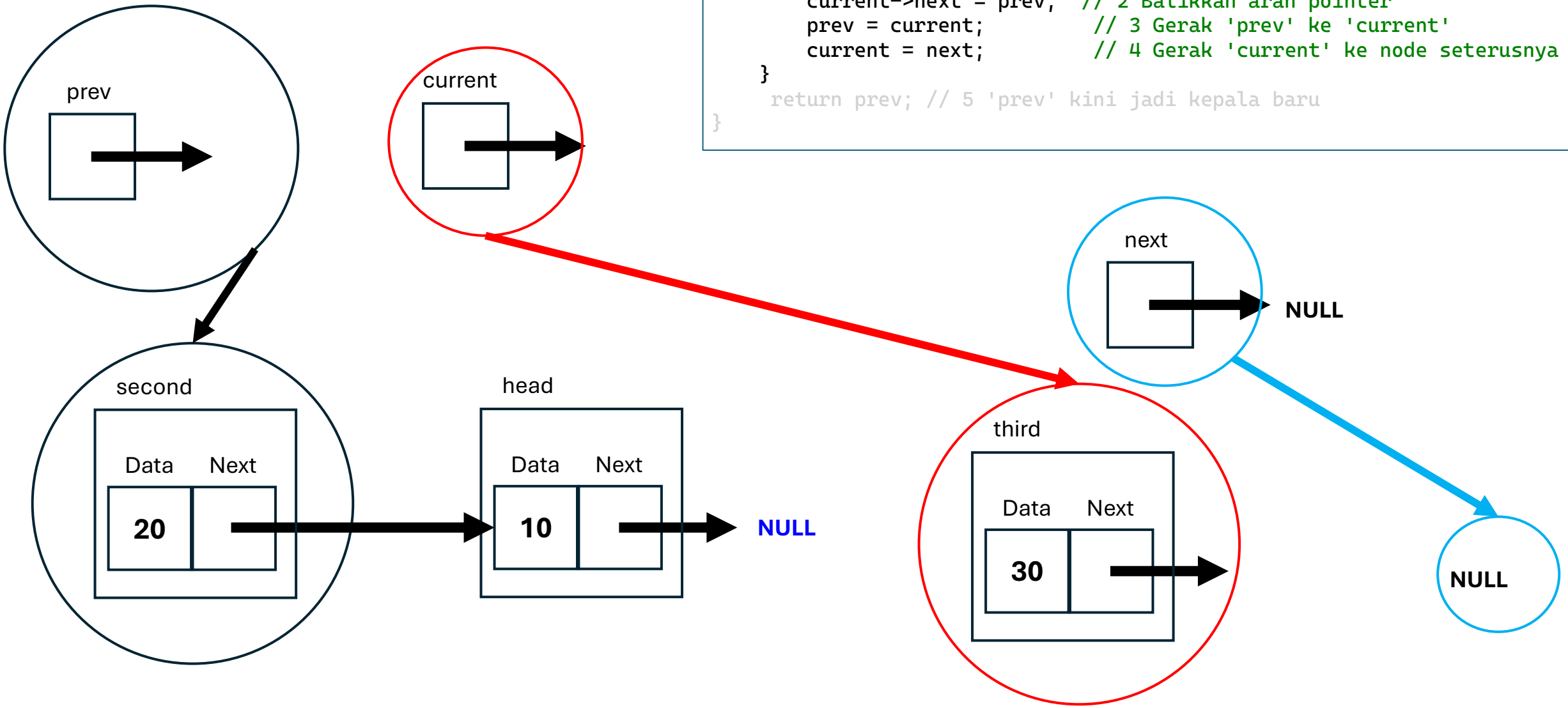


Third loop

2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

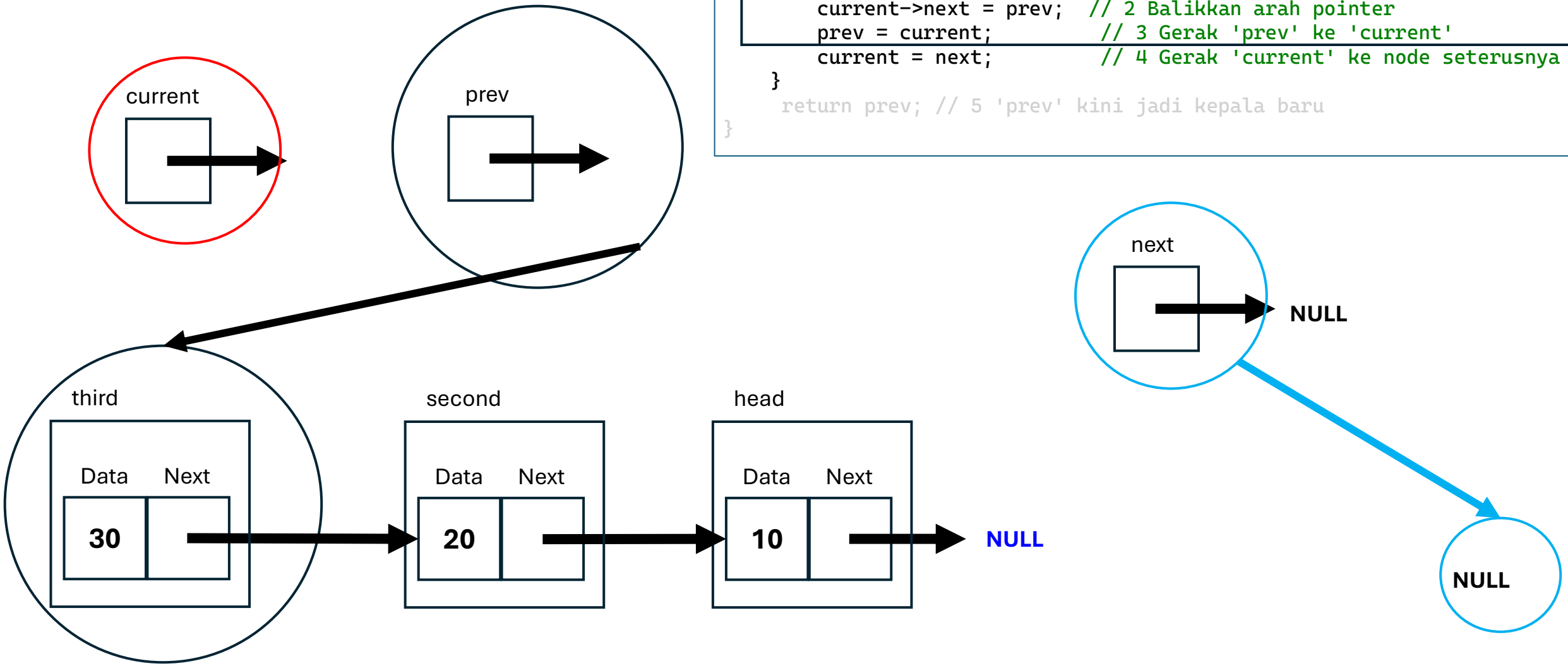
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
     return prev; // 5 'prev' kini jadi kepala baru
}
```



2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

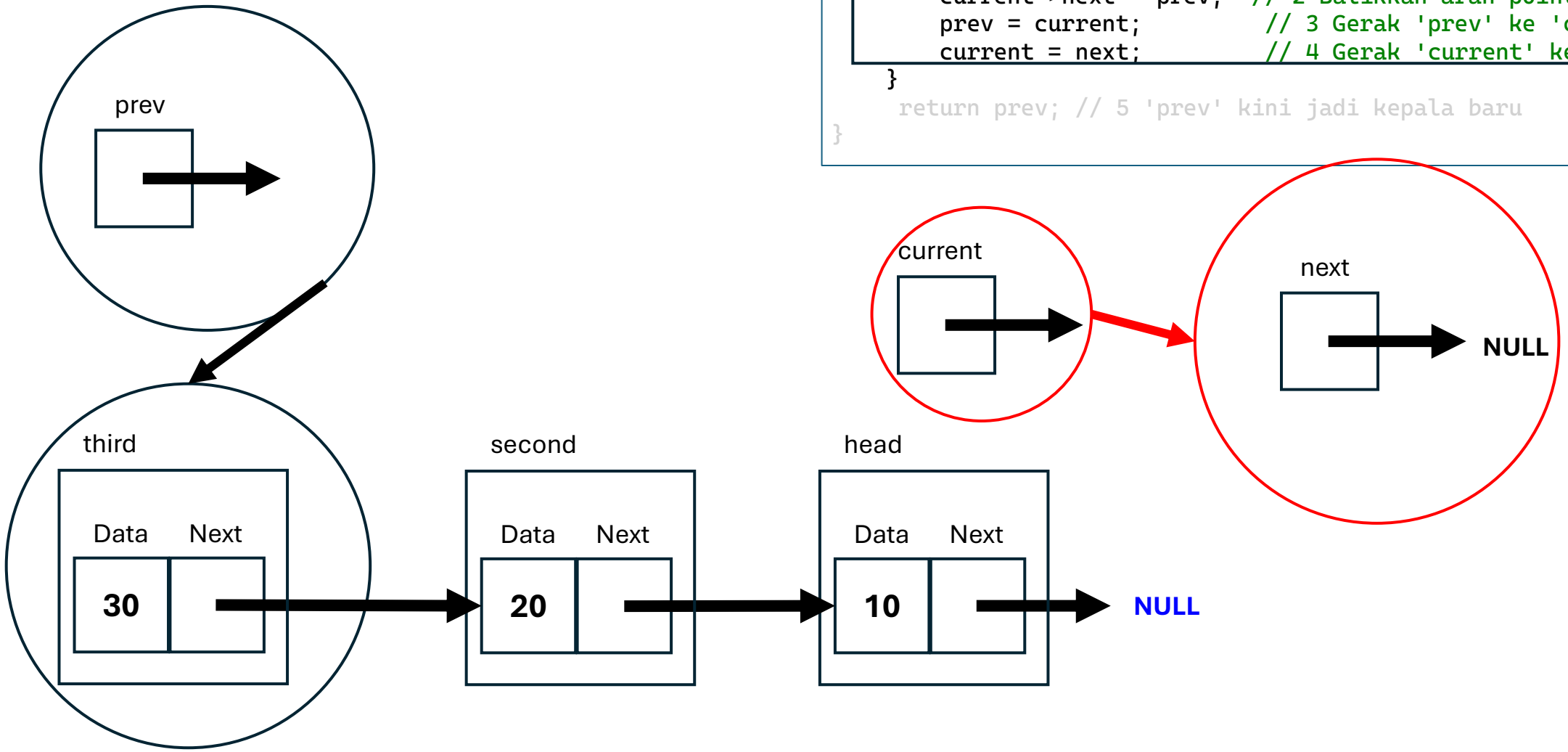
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

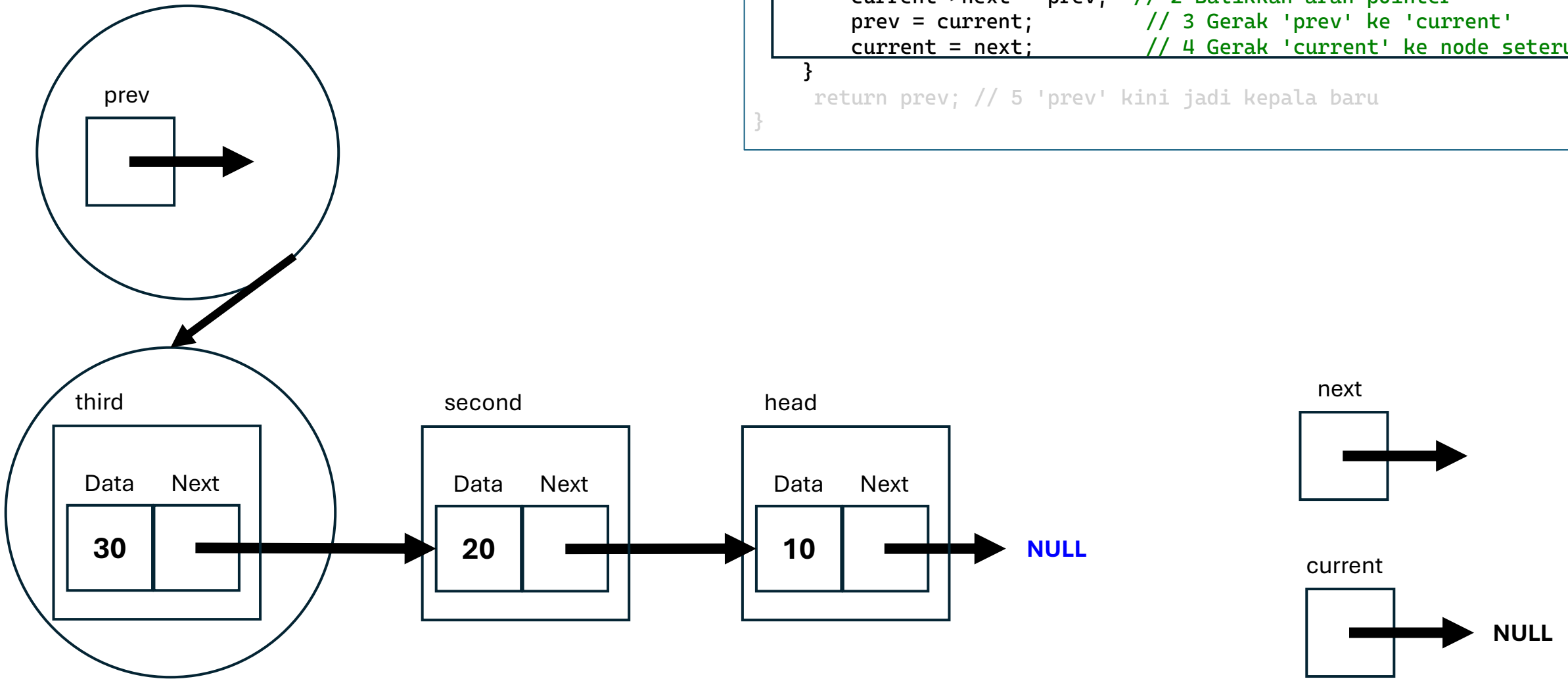
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



2nd LOOP

- **'current' gerak ke node seterusnya (SECOND)** untuk ulang proses yang sama.

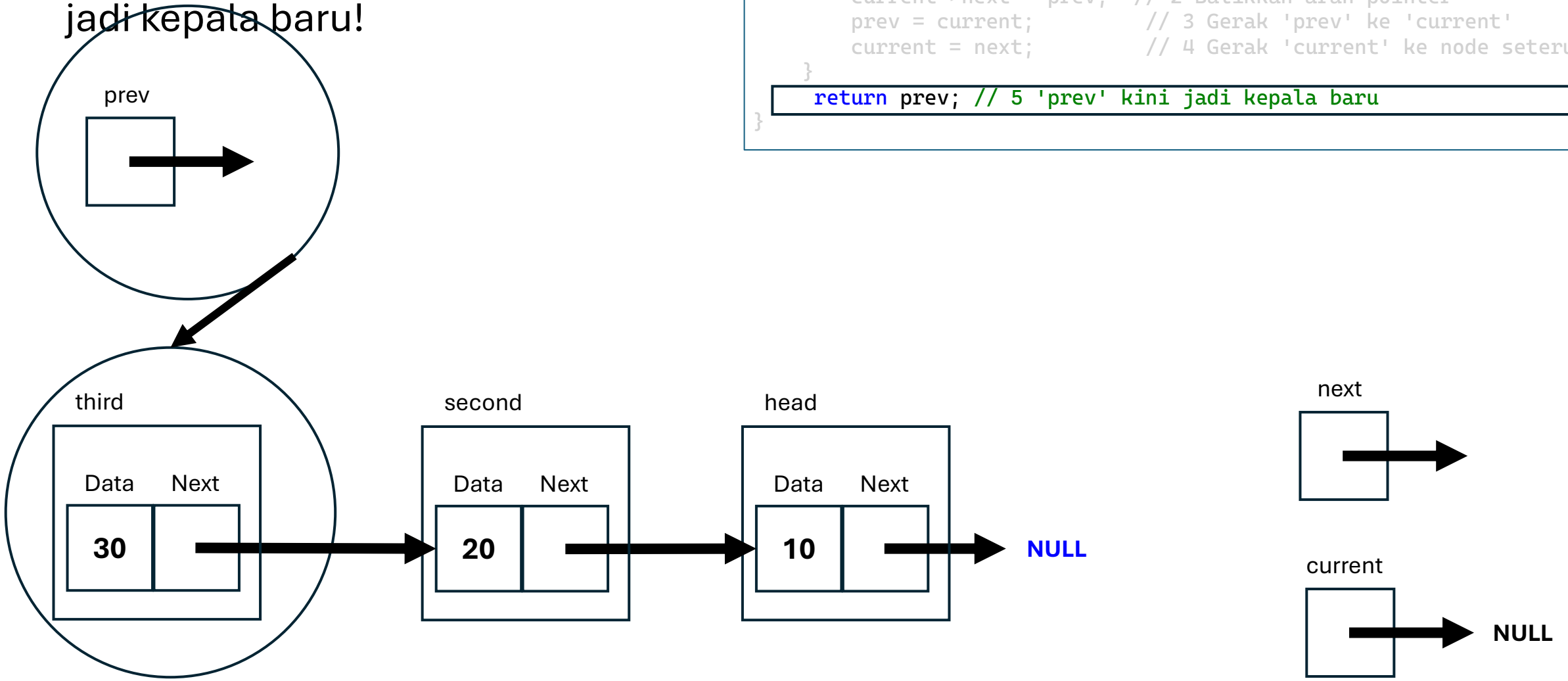
```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



END OF LOOP

- Selepas habis loop, **prev** akan pegang node terakhir asal — ia jadi kepala baru!

```
Node* reverse_iterative(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next; // 1 Simpan node seterusnya
        current->next = prev;  // 2 Balikkan arah pointer
        prev = current;         // 3 Gerak 'prev' ke 'current'
        current = next;         // 4 Gerak 'current' ke node seterusnya
    }
    return prev; // 5 'prev' kini jadi kepala baru
}
```



"Dry Run" (Manual Trace)

Step	Current	Prev	Next	List Baru
Start	head	NULL	second	head -> NULL
Step 1	second	head	third	second-> head -> NULL
Step 2	third	second	NULL	third -> second -> head -> NULL
End	NULL	third	-	DONE ✅

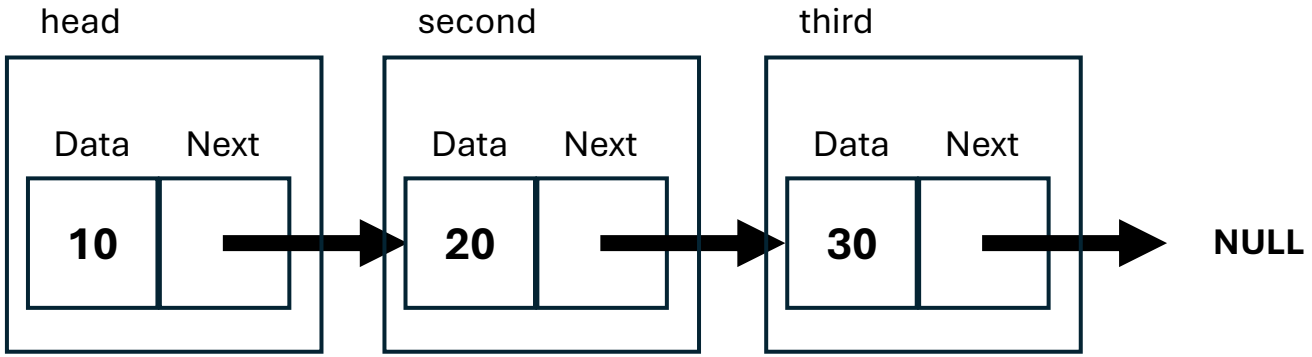
Reverse Linked List (Recursive)

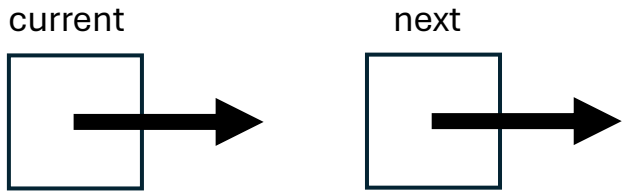
```
Node* reverse_recursive(Node* head) {
    if (head == NULL || head->next == NULL)
        return head;

    Node* rest = reverse_recursive(head->next);
    head->next->next = head;
    head->next = NULL;

    return rest;
}
```

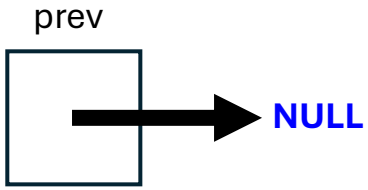
Preparation

- Kalau list kosong (NULL) atau hanya ada satu node, kita terus pulangkan kepala asal — tak perlu reverse apa-apa pun!

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```







Let’s Go

- ➡️ Kita panggil diri sendiri (reverse_recursive) untuk reverse node lepas head sampai ke node terakhir.
- ➡️ Bila recursion stop? — Bila sampai ke node terakhir (base case).

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```

Contoh:

1 -> 2 -> 3 -> NULL

✅ Panggilan rekursif akan jadi:

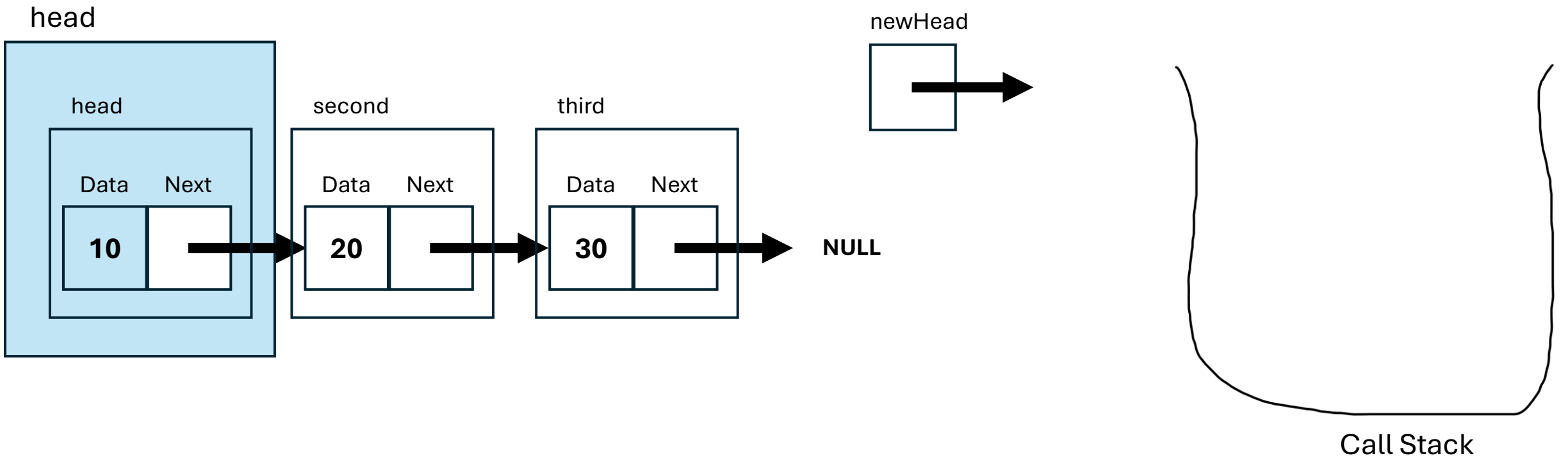
reverse_recursive(1) → reverse_recursive(2) → reverse_recursive(3) → return 3

➡️ **newHead** akan simpan node terakhir (3), yang akan jadi kepala baru.

Preparation

- ➡️ Kita panggil diri sendiri (reverse_recursive) untuk reverse node lepas head sampai ke node terakhir.
- ➡️ Bila recursion stop? — Bila sampai ke node terakhir (base case).

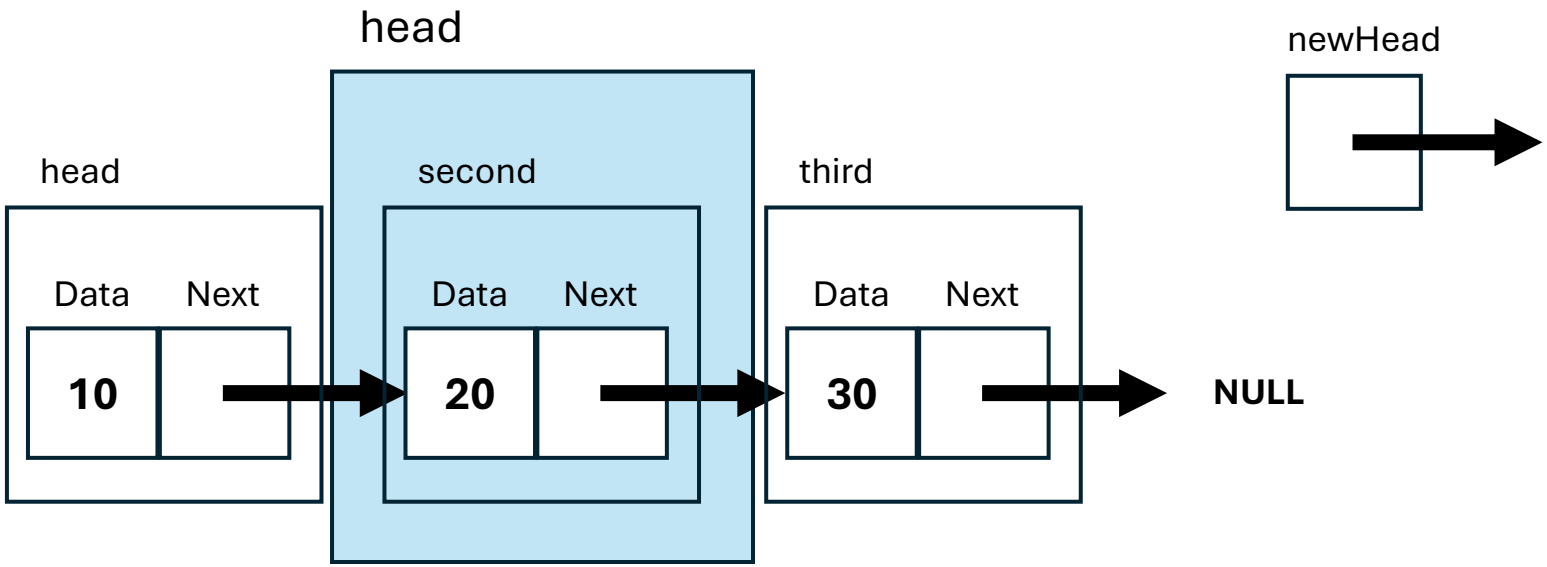
```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```

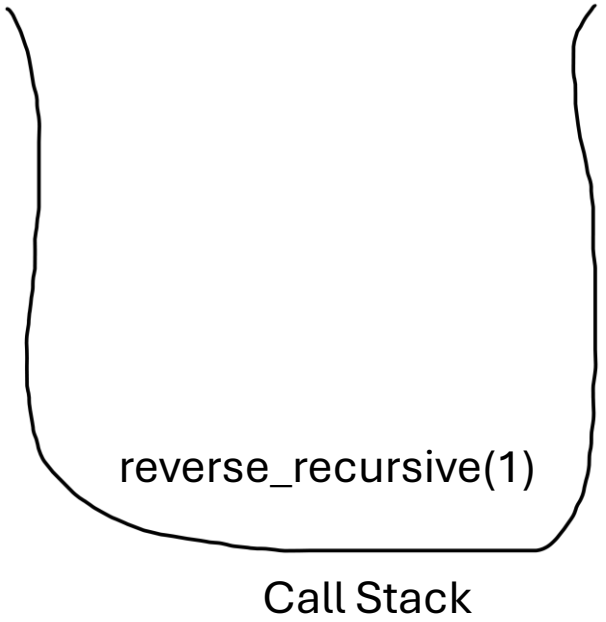


Preparation

- ➡️ Kita panggil diri sendiri (reverse_recursive) untuk reverse node lepas head sampai ke node terakhir.
- ➡️ Bila recursion stop? — Bila sampai ke node terakhir (base case).

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```

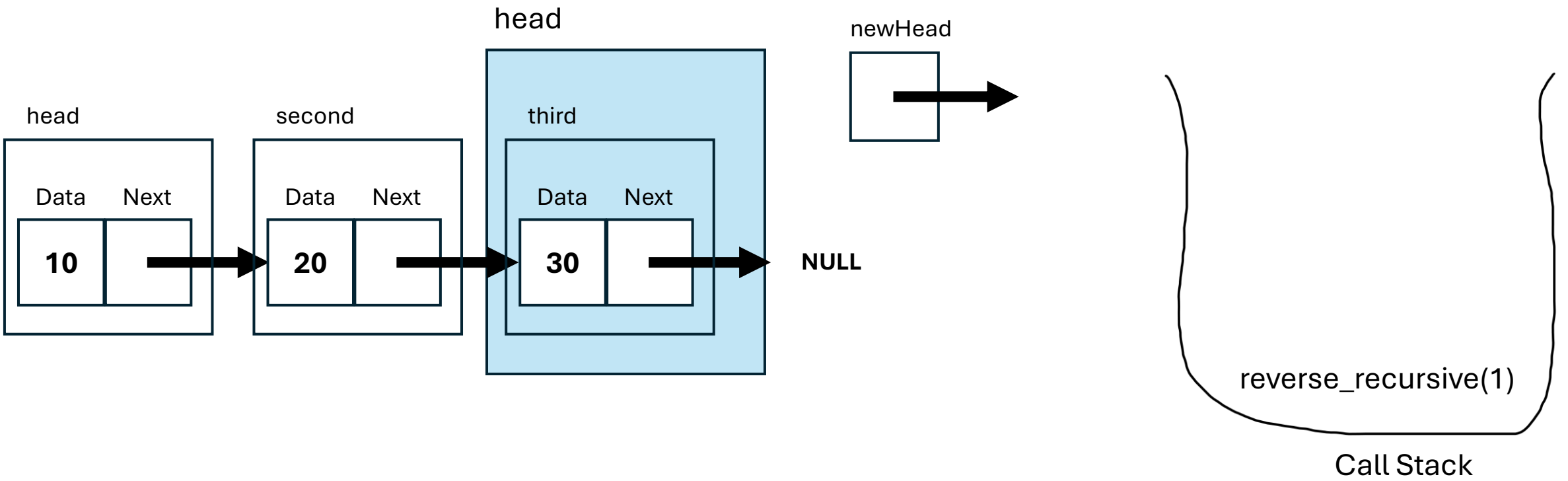




Preparation

- ➡️ Kita panggil diri sendiri (reverse_recursive) untuk reverse node lepas head sampai ke node terakhir.
- ➡️ Bila recursion stop? — Bila sampai ke node terakhir (base case).

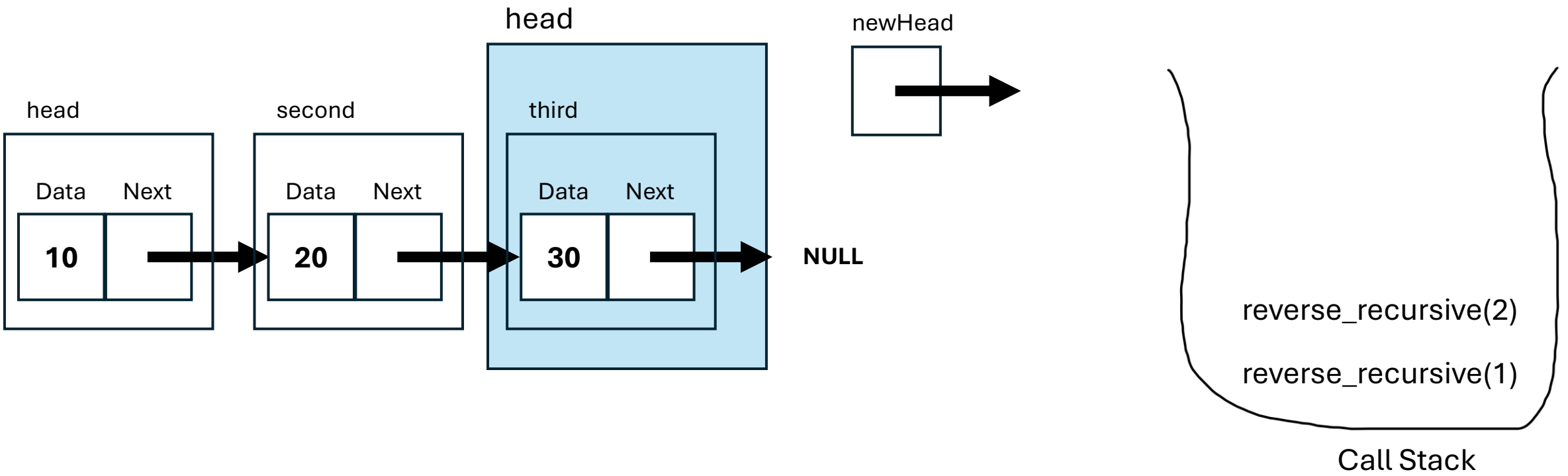
```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Preparation

- ➡️ Kita panggil diri sendiri (reverse_recursive) untuk reverse node lepas head sampai ke node terakhir.
- ➡️ Bila recursion stop? — Bila sampai ke node terakhir (base case).

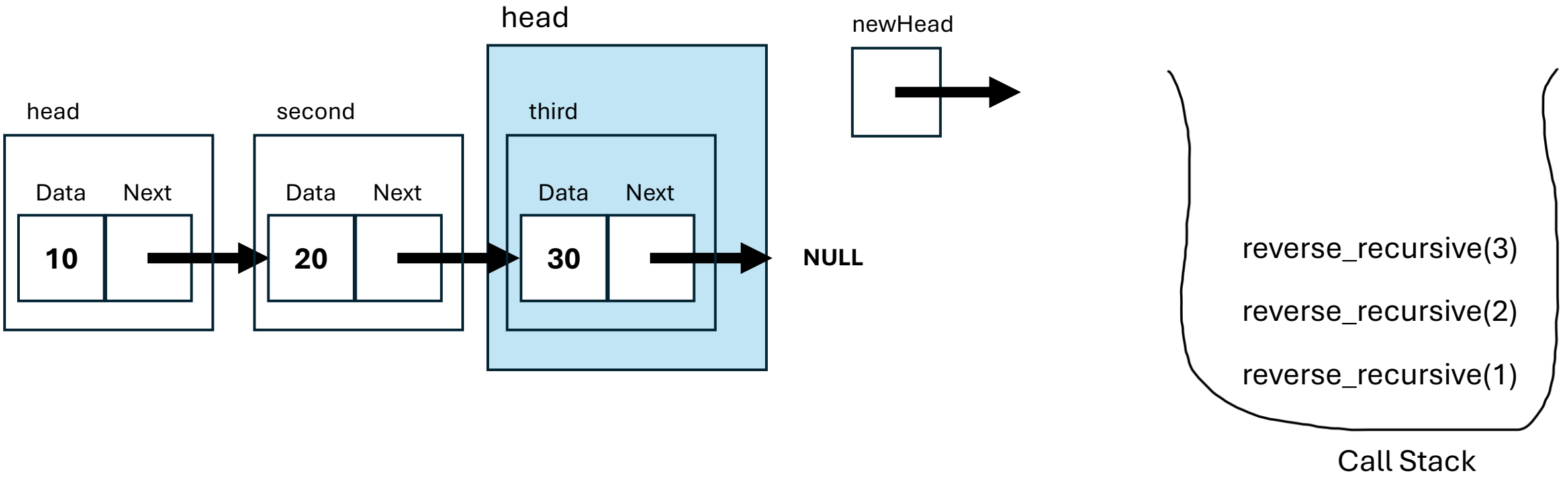
```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Preparation

- ➡️ Kita panggil diri sendiri (reverse_recursive) untuk reverse node lepas head sampai ke node terakhir.
- ➡️ Bila recursion stop? — Bila sampai ke node terakhir (base case).

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Preparation

➡️ Ini bahagian otak twist sikit! Kita balikkan pointer dari node seterusnya.Kalau head = 20, maksudnya:

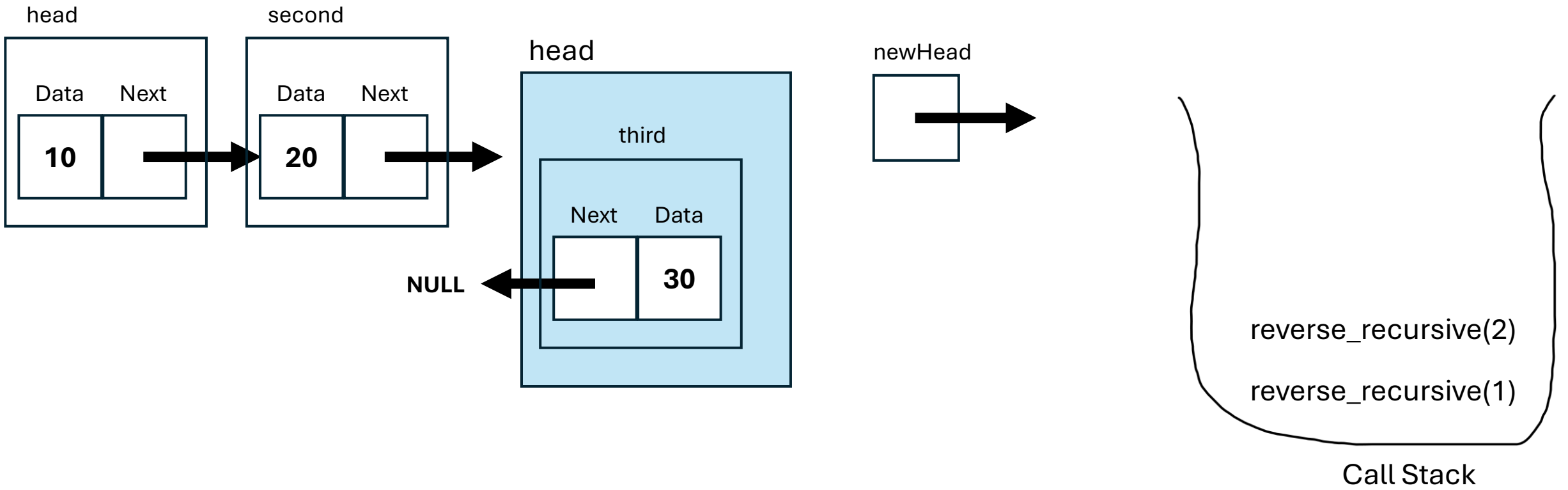
20->next = 30

30->next = 20 // Kita pusing arah pointer!

Jadinya:

10 -> 20 -> 30 → 10 -> 20 <- 30

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Preparation

➡️ Ini bahagian otak twist sikit! Kita balikkan pointer dari node seterusnya.Kalau head = 20, maksudnya:

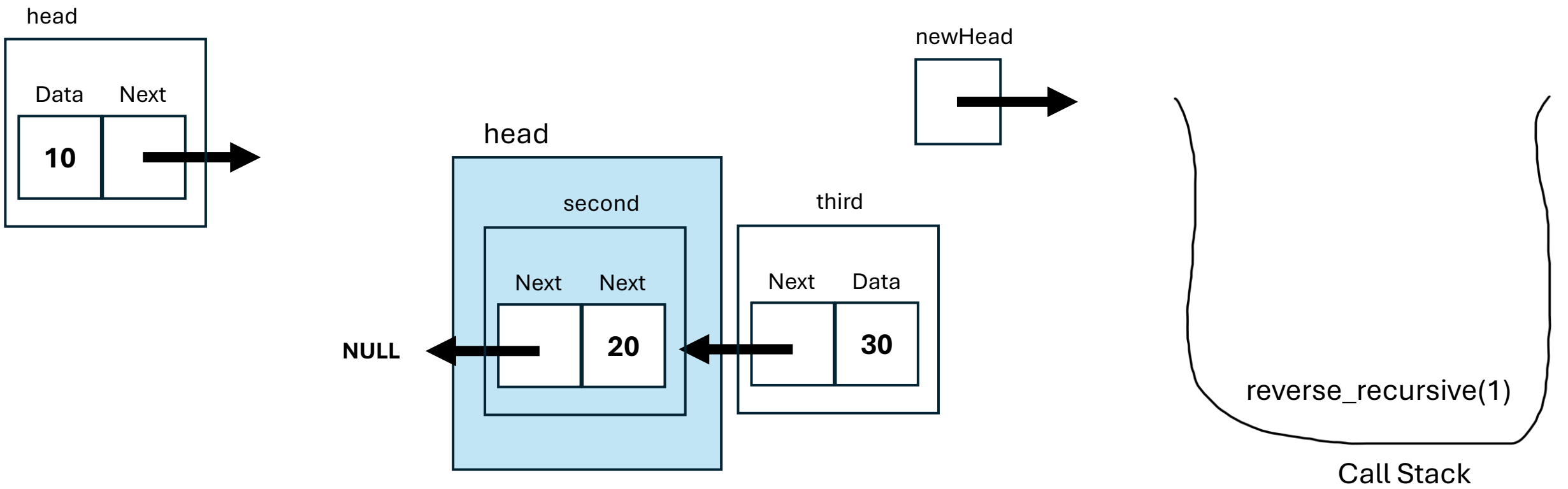
20->next = 30

30->next = 20 // Kita pusing arah pointer!

Jadinya:

10 -> 20 -> 30 → 10 -> 20 <- 30

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Preparation

➡️ Ini bahagian otak twist sikit! Kita balikkan pointer dari node seterusnya.Kalau head = 20, maksudnya:

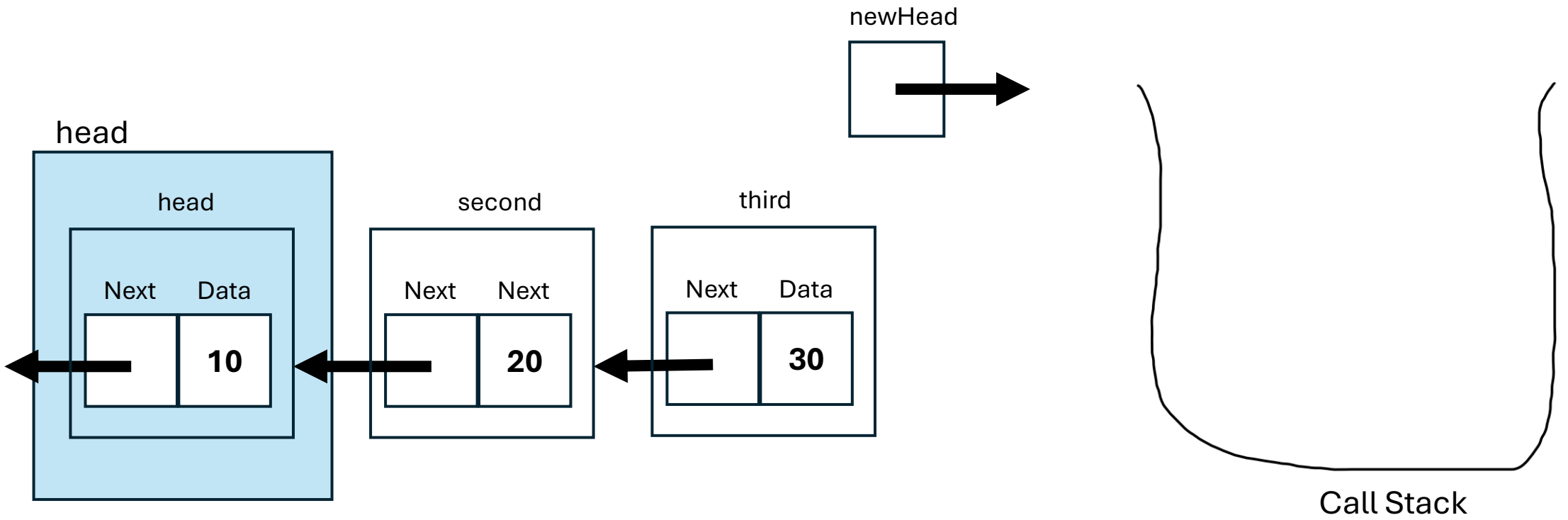
20->next = 30

30->next = 20 // Kita pusing arah pointer!

Jadinya:

10 -> 20 -> 30 → 10 -> 20 <- 30

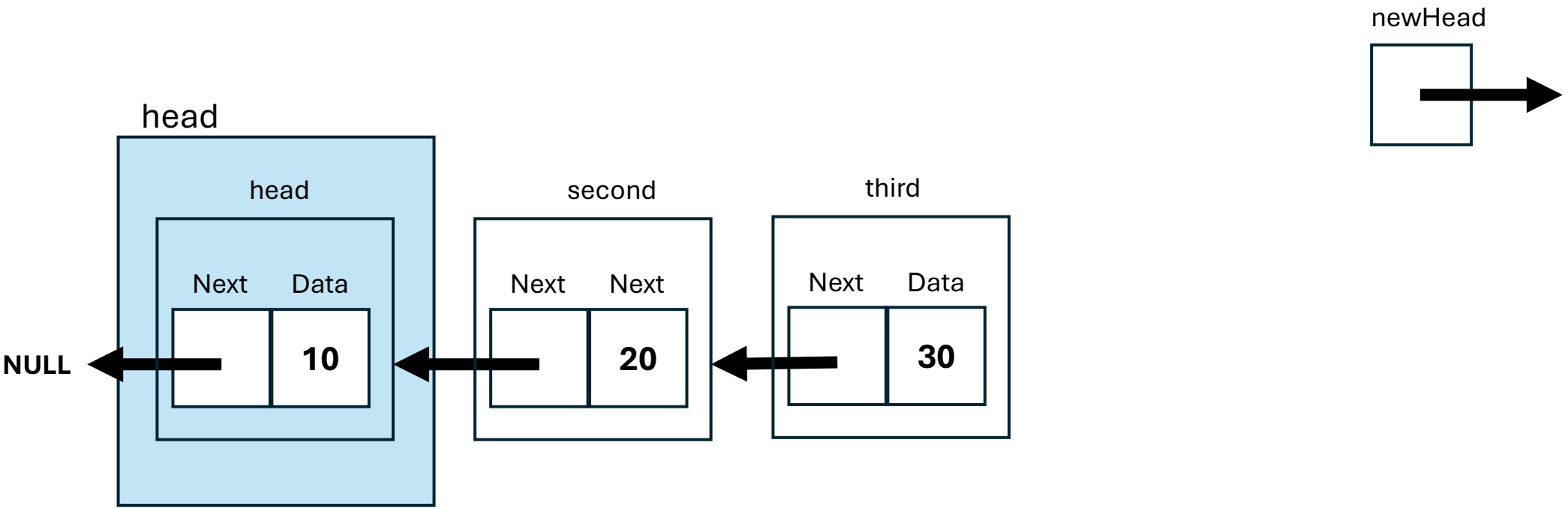
```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Preparation

➡️ Lepas reverse, kita **potong** connection lama supaya node asal tak lagi tunjuk ke node seterusnya yang lama.

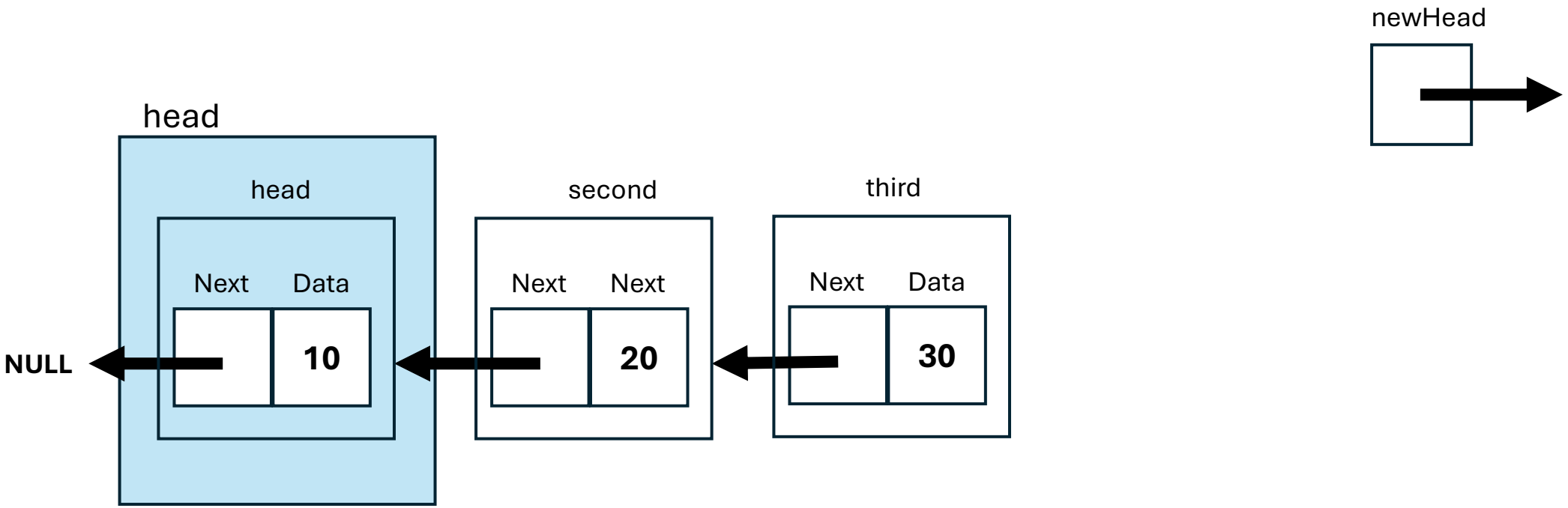
```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



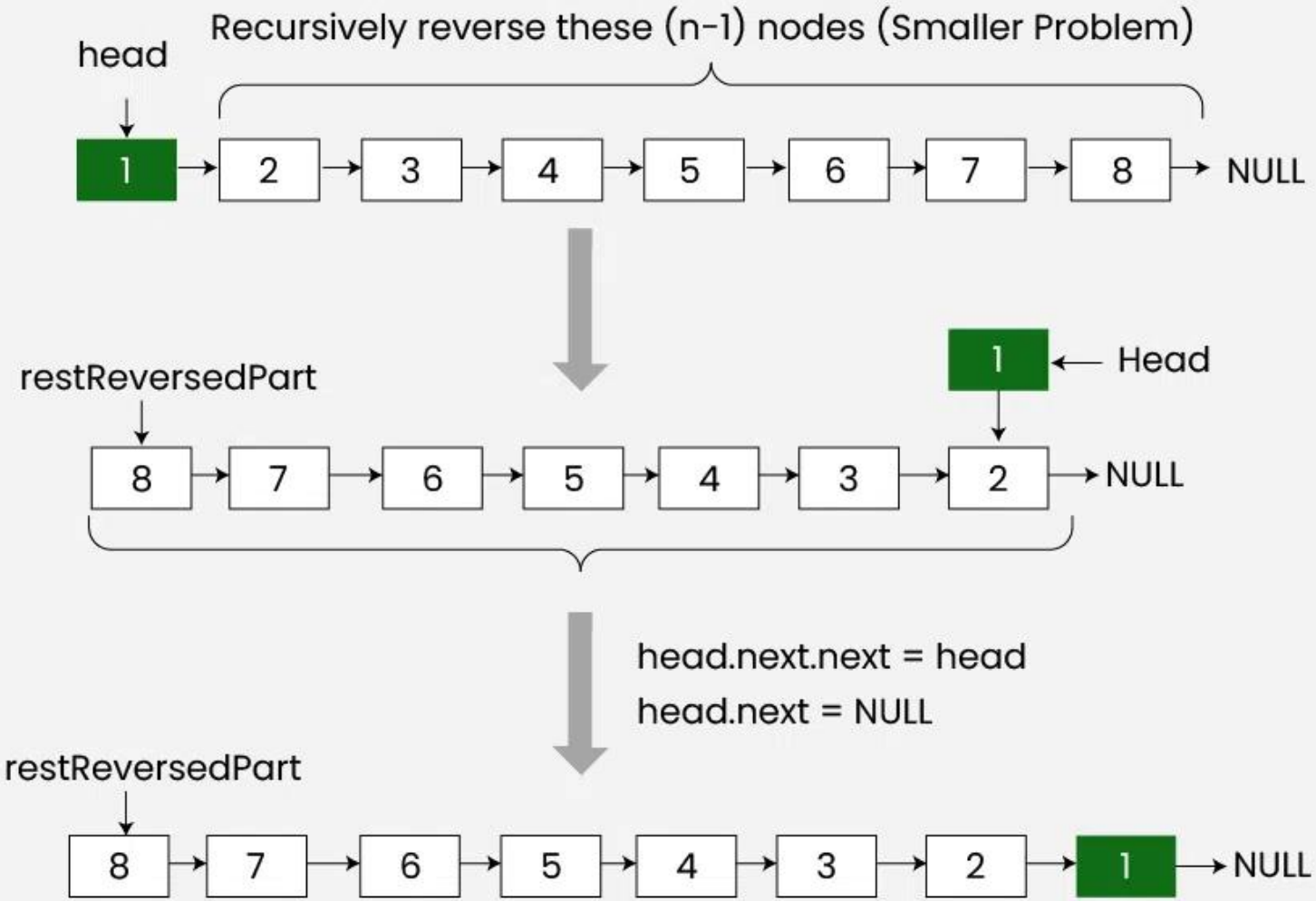
Preparation

Pulangkan kepala baru yang kita dapat dari recursion tadi (node terakhir asal).

```
Node* reverse_recursive(Node* head) {
    // Base case: kalau list kosong atau ada satu node je
    if (head == NULL || head->next == NULL)
        return head;

    // Recursion: reverse semua node lepas node pertama
    Node* newHead = reverse_recursive(head->next);

    // Balikkan arah node semasa
    head->next->next = head;
    head->next = NULL;

    return newHead;  // Pulangkan kepala baru (node terakhir asal)
}
```



Reverse a Linked List Recursively



Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

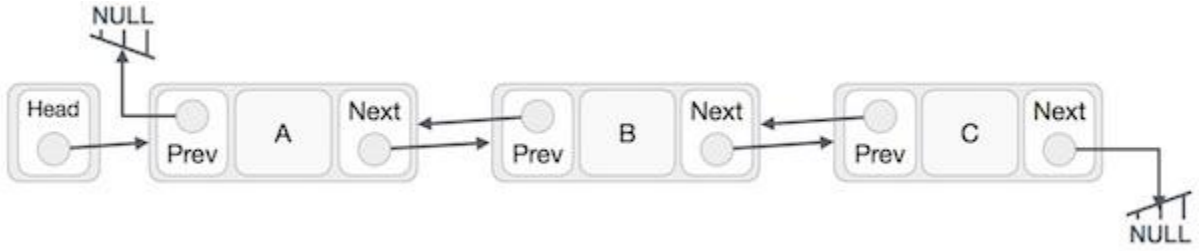
Following are the important terms to understand the concept of doubly linked list.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List

As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.



Doubly Linked List

Basic Operations

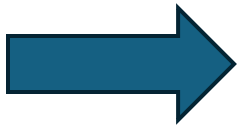
- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.
- **Display backward** – Displays the complete list in a backward manner.

Let’s look upon coding examples

```
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
};

void print_list(Node* n) {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}

int main() {
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = NULL;

    print_list(head);
}
```

First Step

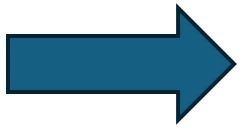
```cpp
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
};
```



```cpp
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node* prev;
};
```

2nd Step

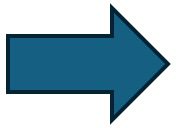
```
int main() {
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = NULL;

    print_list(head);
}
```



```
int main() {
    Node* head = new Node();
    Node* second = new Node();
    Node* third = new Node();

    // Masukkan data ke dalam setiap node
    head->data = 1;
    head->next = second;
    head->prev = nullptr;

    second->data = 2;
    second->next = third;
    second->prev = head;

    third->data = 3;
    third->next = nullptr;
    third->prev = second;

    print_list(head);
}
```

3rd Step

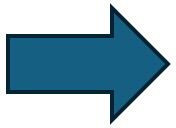
```
void print_list(Node* n) {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}
```



```
// Print dari depan ke belakang
void print_forward(Node* node) {
    while (node != nullptr) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

// Print dari belakang ke depan
void print_backward(Node* node) {
    while (node->next != nullptr) {
        node = node->next;
    }

    while (node != nullptr) {
        cout << node->data << " ";
        node = node->prev;
    }
    cout << endl;
}
```

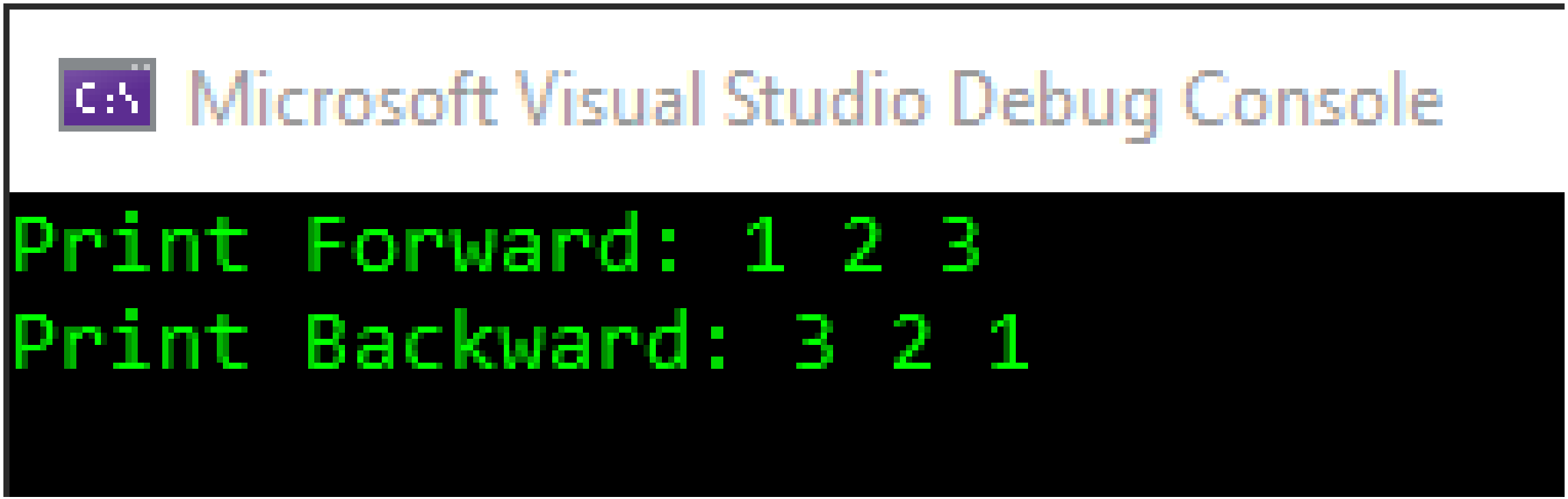
4th Step – inside main function

```
print_list(head);
```



```
cout << "Print Forward: ";
   print_forward(head);

   cout << "Print Backward: ";
   print_backward(head);

   return 0;
```

Output


Microsoft Visual Studio Debug Console
Print Forward: 1 2 3
Print Backward: 3 2 1

How to insert node in the middle of Doubly Linked List?

1st Step – Add following function

insert_middle(second, 99);

```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
    }

    // Buat node baru
    Node* new_node = new Node();
    new_node->data = new_data;

    // Sambungkan node baru ke node seterusnya
    new_node->next = prev_node->next;

    // Sambungkan node lama ke node baru
    prev_node->next = new_node;

    // Sambungkan node baru ke node sebelumnya
    new_node->prev = prev_node;

    // Kalau node baru bukan di hujung, sambung prev node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
}
```

1st Step – Add following function

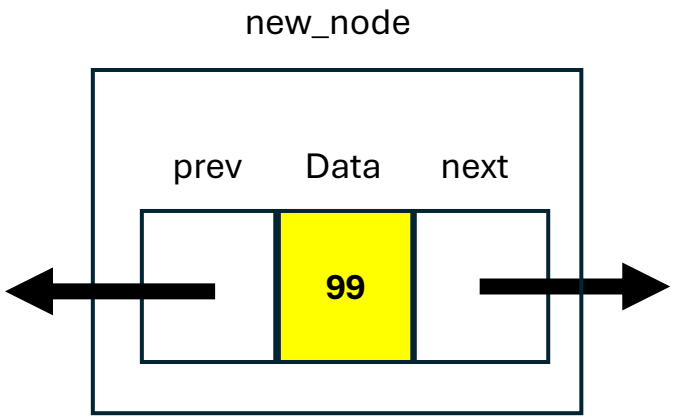
insert_middle(second, 99);

```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
    }

    // Buat node baru

    // Sambungkan node lama ke node baru
    prev_node->next = new_node;

    // Sambungkan node baru ke node sebelumnya
    new_node->prev = prev_node;

    // Kalau node baru bukan di hujung, sambung prev node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
}
```

Kita check dulu sama ada prev_node valid atau tak. Kalau prev_node kosong (nullptr), kita stop terus sebab kita tak nak sambung node baru ke node yang tak wujud.

1st Step – Add following function

insert_middle(second, 99);



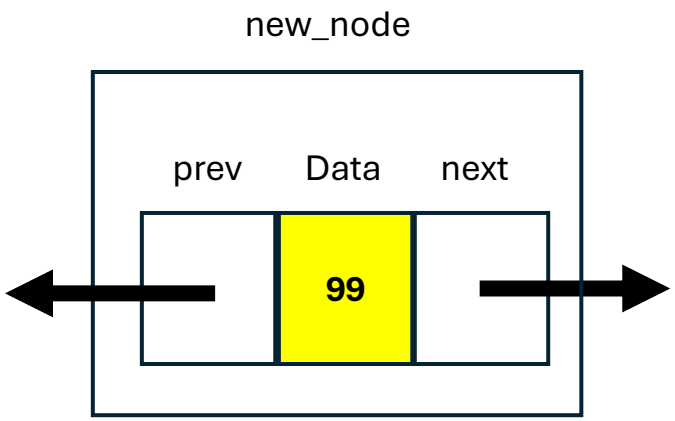

```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
    }

    // Buat node baru
    Node* new_node = new Node();
    new_node->data = new_data;

    // Sambungkan node baru ke node sebelumnya
    new_node->prev = prev_node;

    // Kalau node baru bukan di hujung, sambung prev node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
}
```

Kita cipta node baru dan masukkan data ke dalamnya. Contoh:
Kalau new_data = 99, node baru kita sekarang jadi macam ni:

[99 | next=nullptr | prev=nullptr]

1st Step – Add following function

insert_middle(second, 99);




```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
    }

    // Buat node baru
    Node* new_node = new Node();
    new_node->data = new_data;

    // Sambungkan node baru ke node seterusnya
    new_node->next = prev_node->next;
```

Kita sambungkan node baru ke node selepas prev_node (iaitu node ketiga).
Katakan doubly linked list asal macam ni:

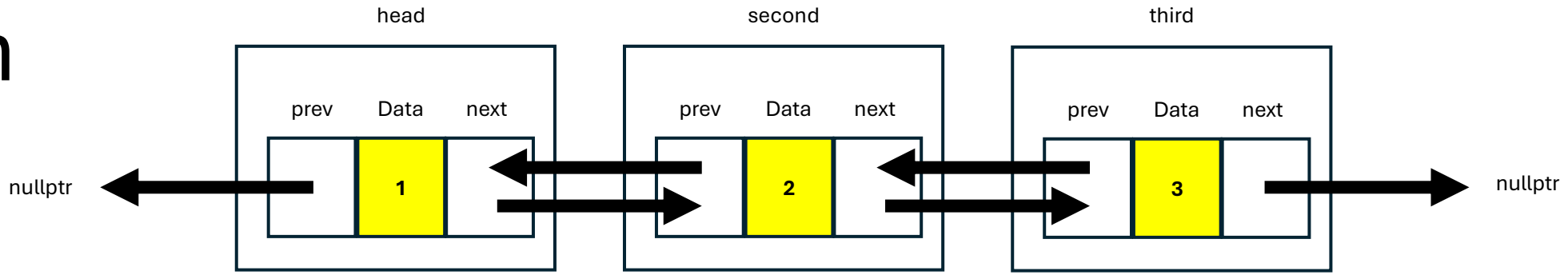
[1] <-> [2] <-> [3]

Sekarang, kita sambungkan new_node->next ke node ketiga:

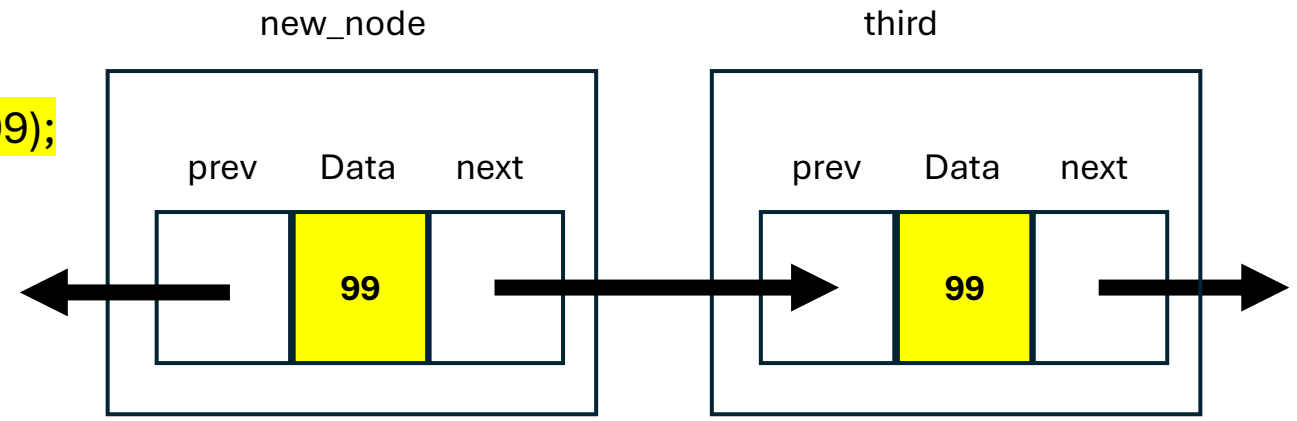
[99] -> [3]

1st Step – Add following function

```
// Sambungkan node baru ke node seterusnya
   new_node->next = prev_node->next;
```



insert_middle(second, 99);



1st Step – Add following function

insert_middle(second, 99);

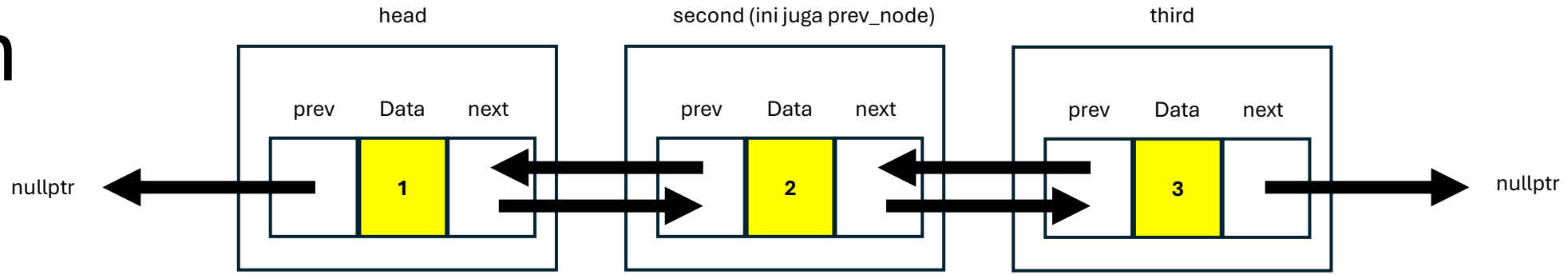
```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
    }

    // Buat node baru
    Node* new_node = new Node();
    new_node->data = new_data;

    // Sambungkan node baru ke node seterusnya
    new_node->next = prev_node->next;

    // Sambungkan node lama ke node baru
    prev_node->next = new_node;

    // Update prev node dari node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
}
```

Kita sambungkan node kedua ke node baru:

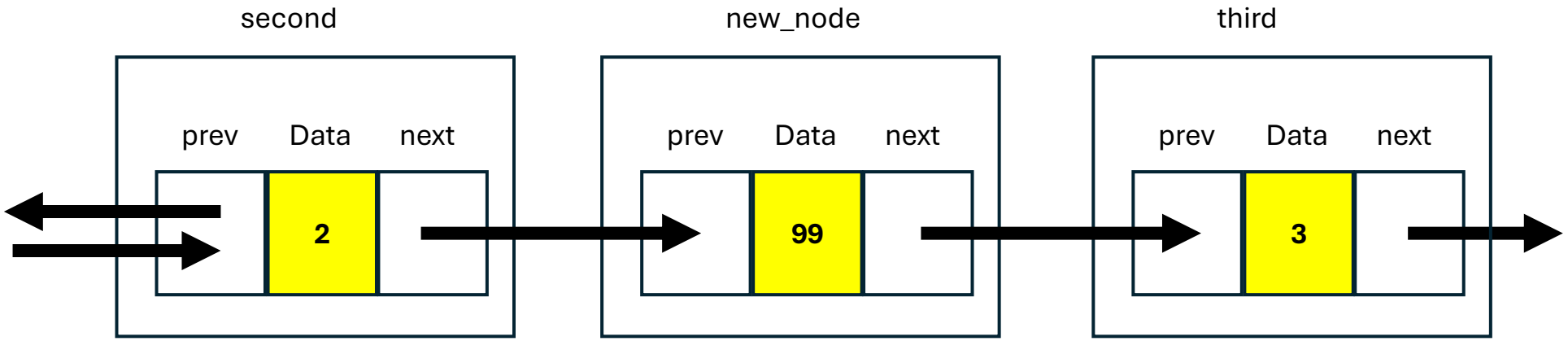
[2] -> [99]

1st Step – Add following function

```
// Sambungkan node lama ke node baru
   prev_node->next = new_node;
```



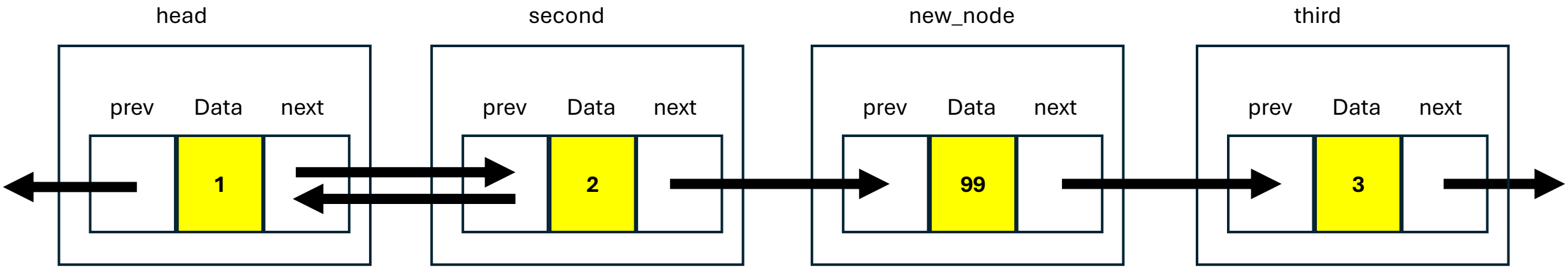
insert_middle(second, 99);



1st Step – Add following function

insert_middle(second, 99);

Sambungan sekarang



1st Step – Add following function

insert_middle(second, 99);

```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
```

Kita sambungkan node baru punya prev ke node sebelumnya (prev_node):

[2]<-[99]

Sekarang doubly linked list jadi:

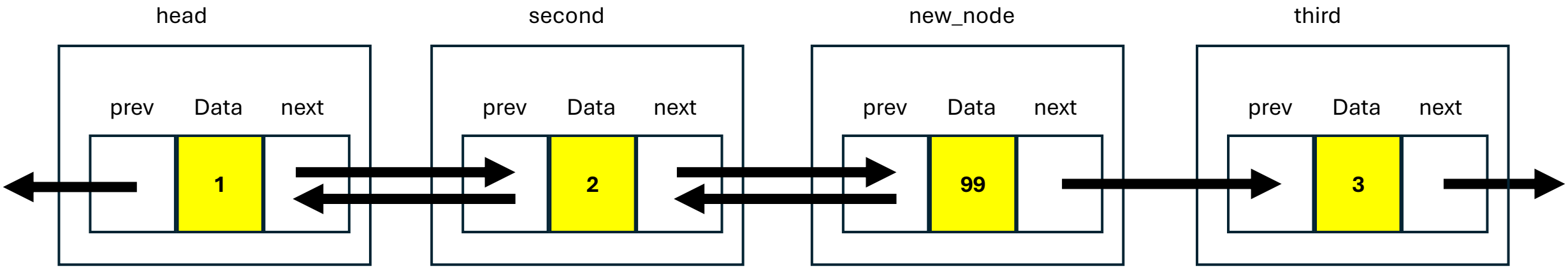
[1]<->[2]<->[99]->[3]

```
    // Sambungkan node baru ke node sebelumnya
    new_node->prev = prev_node;

    // Kalau node baru bukan di hujung, sambung prev node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
}
```

1st Step – Add following function

```
insert_middle(second, 99);
// Sambungkan node baru ke node sebelumnya
    new_node->prev = prev_node;
```

Harus ingat, prev_node adalah ‘second’ node jadi new_node tersebut refer kepada ‘second’ node



1st Step – Add following function

insert_middle(second, 99);

```
// Fungsi tambah node di tengah (lepas node kedua)
void insert_middle(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        cout << "Node sebelumnya tidak boleh kosong!" << endl;
        return;
    }
```

Kita check kalau node baru bukan yang terakhir (maksudnya ada node seterusnya), kita kena pastikan node ketiga sambung balik prev ke node baru:

[99] <-> [3]

Akhirnya, list kita jadi lengkap macam ni:

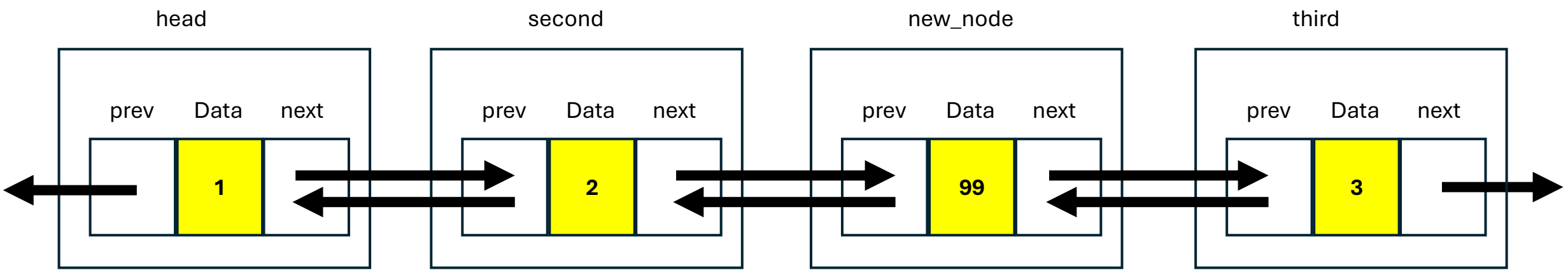
[1] <-> [2] <-> [99] <-> [3]

```
    // Kalau node baru bukan di hujung, sambung prev node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
}
```

1st Step – Add following function

insert_middle(second, 99);

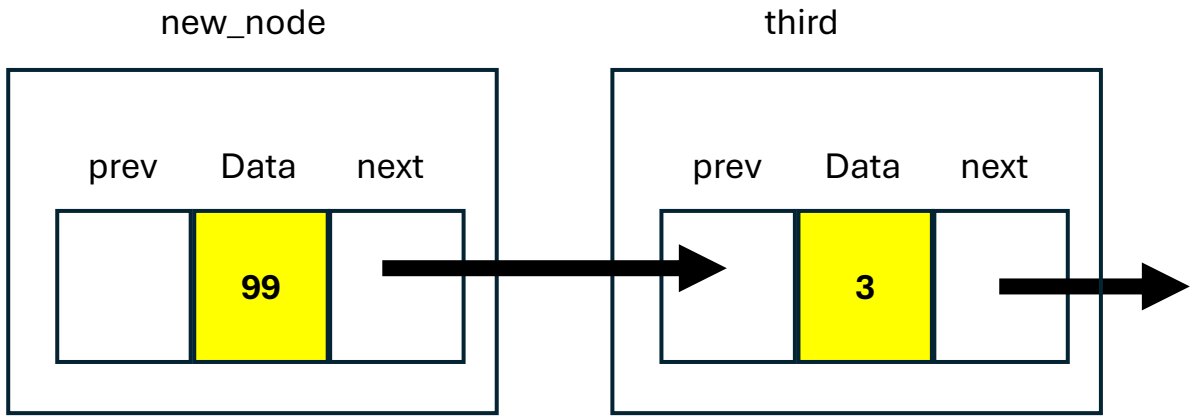
```
// Kalau node baru bukan di hujung, sambung prev node seterusnya
    if (new_node->next != nullptr) {
        new_node->next->prev = new_node;
    }
```



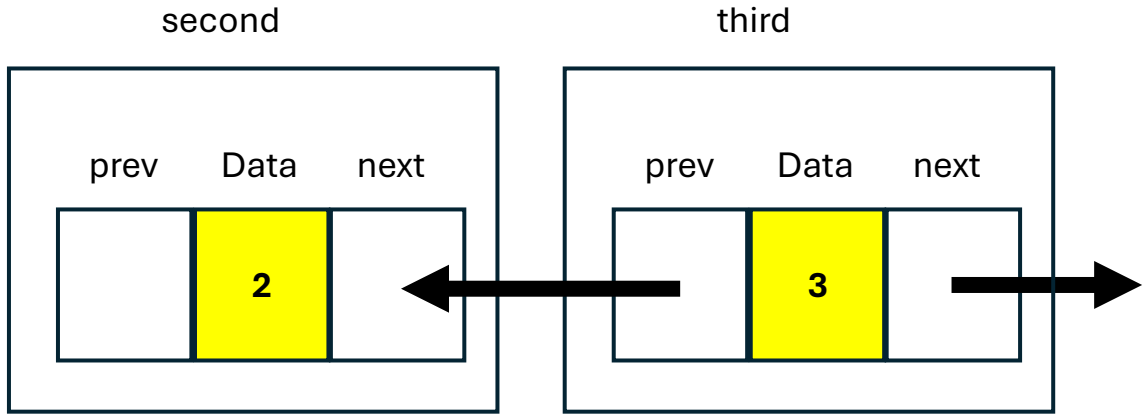
Kena ingat, **third node "prev"** masih ingat merujuk second node tu, jadi kita kene baiki supaya merujuk kepada new_node (yg ada 99 tu). **new_node->next** hanya tunjuk node yg ketiga. Jadi, **new_node->next->prev akan memastikan third node punya "prev" dioverwritekan** agar merujuk kepada new_node.

How?

```
new_node->next->prev = new_node
```

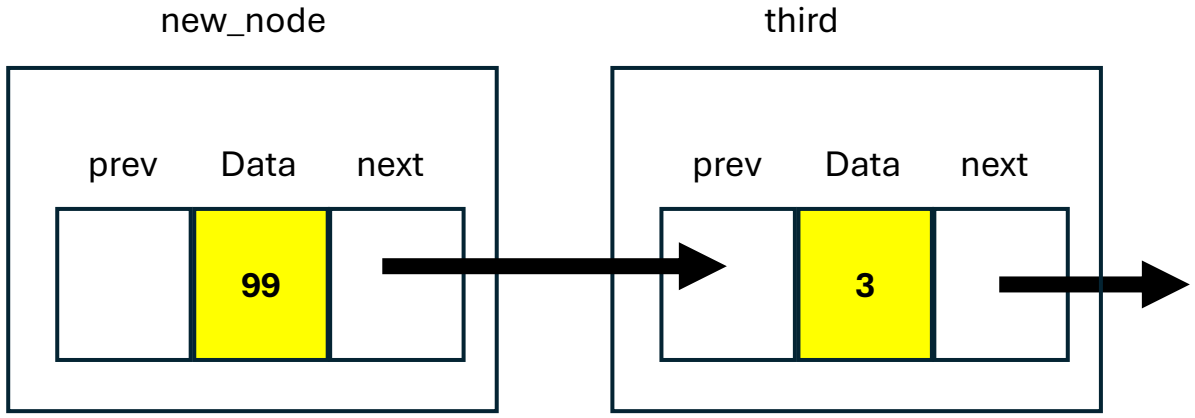


Kena ingat, **third node "prev"** masih merujuk kepada second node

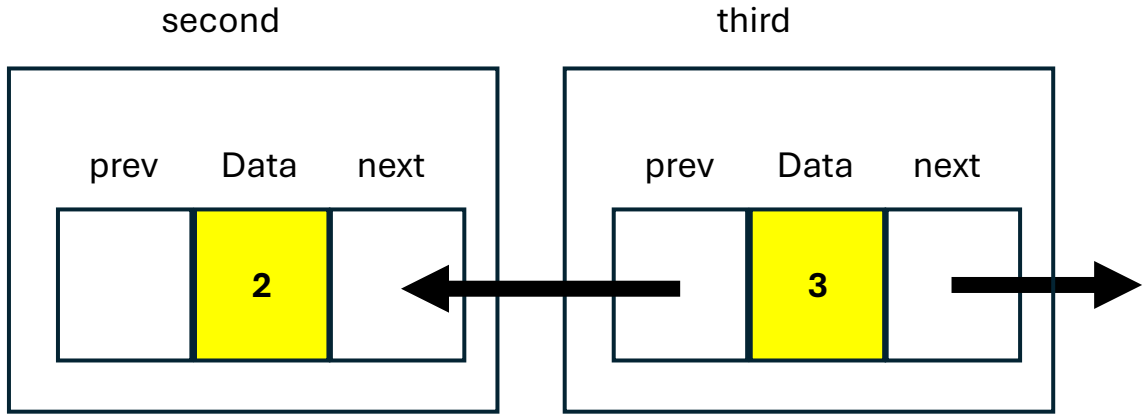


How?

```
new_node->next->prev = new_node
```

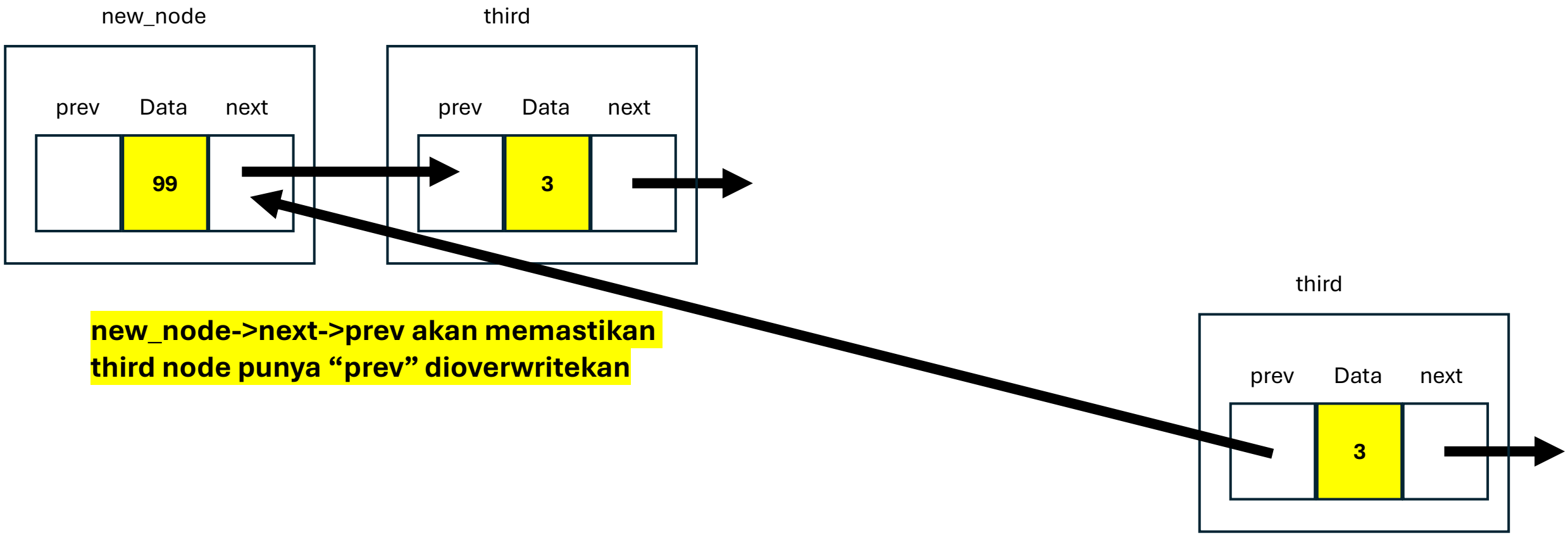


new_node->next hanya tunjuk node yg ketiga



How?

```
new_node->next->prev = new_node
```



new_node->next->prev akan memastikan third node punya “prev” dioverwritekan

Sebab `new_node->next->prev = new_node`

👉 Tanpa line ni:

Node 3 masih "ingat" node sebelumnya adalah 2, jadi sambungan akan rosak kalau kita reverse list.

👉 Dengan line ni:

Node 3 sekarang "ingat" node sebelumnya adalah 99, jadi sambungan betul dua hala.

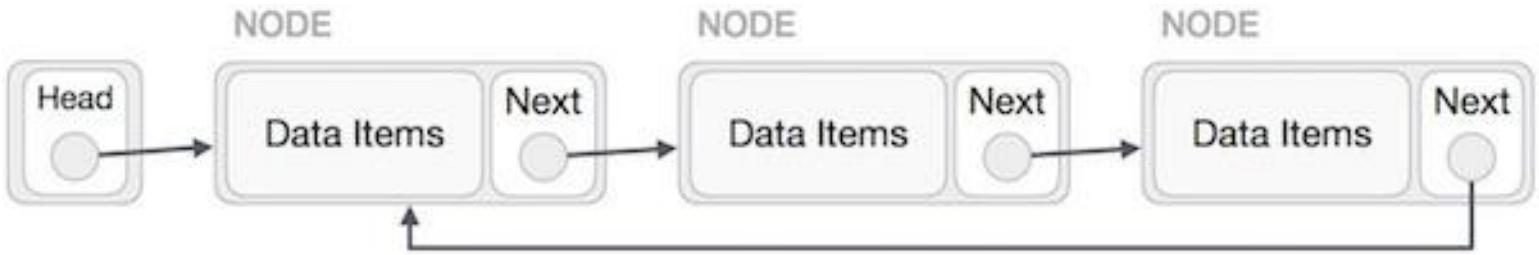
Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element.

Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

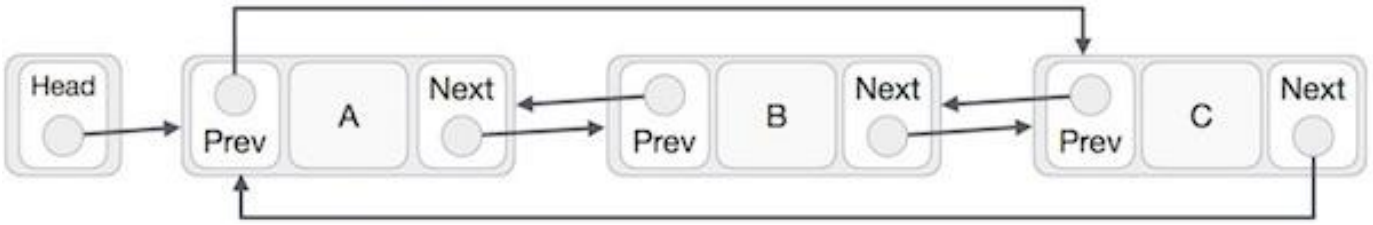
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



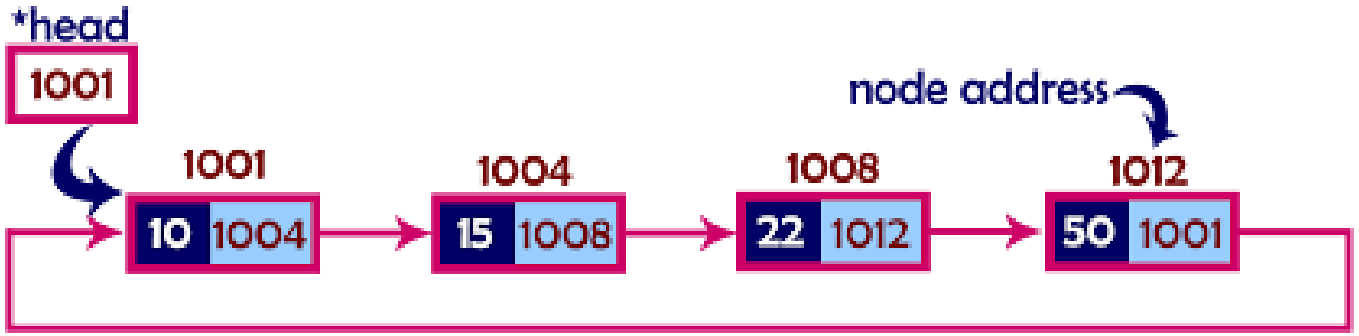
Doubly Linked List as Circular

- In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.
- As per the above illustration, following are the important points to be considered.
  - The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
  - The first link's previous points to the last of the list in case of doubly linked list.



Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.

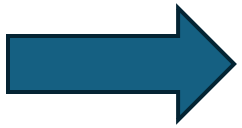


Let’s look upon coding examples - Circular Singly Linked List

```cpp
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
};

void print_list(Node* n) {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}

int main() {
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = NULL;

    print_list(head);
}
```

First Step – node still the same

```cpp
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
};
```



```cpp
#include <cstddef>
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
};
```

2nd Step – print_list

```
void print_list(Node* n) {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}
```

```
void print_list(Node* head) {
    if (head == NULL) return;  // Kalau kosong, keluar awal.

    Node* temp = head;

// Guna do-while supaya pasti cetak sekurang-kurangnya sekali
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head); //Stop bila balik semula ke head
     cout << endl;
}
```

Dalam Singly Linked List biasa, kita loop sampai **NULL**. Tapi dalam Circular Linked List, tak ada **NULL** sebab node terakhir link balik ke head.Jadi, kita tukar while(n != NULL) kepada do-while loop untuk pastikan ia pusing balik ke head.

2nd Step – print_list

```
void print_list(Node* n) {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}
```

```
void print_list(Node* head) {
    if (head == NULL) return;  // Kalau kosong, keluar awal.

    Node* temp = head;

// Guna do-while supaya pasti cetak sekurang-kurangnya sekali
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head); //Stop bila balik semula ke head
     cout << endl;
}
```

✅ Kenapa **do-while** penting?
Kalau kita guna **while** biasa, kalau hanya ada satu node, dia terus skip sebab node pertama dah **head**.
do-while confirm dia cetak node sekurang-kurangnya sekali, baru periksa syarat.

3rd Step – int main function

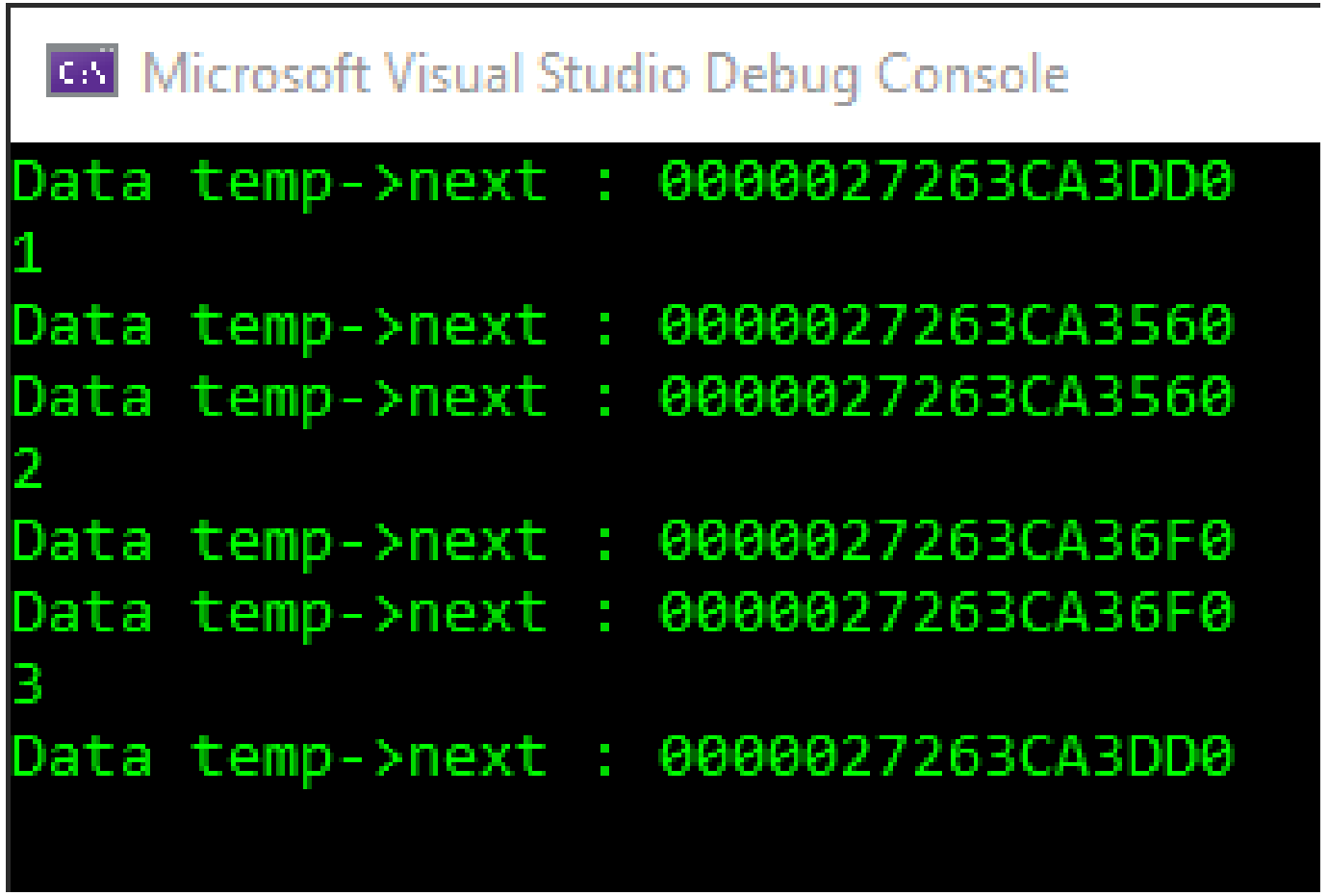
```
Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = NULL;

    print_list(head);
```

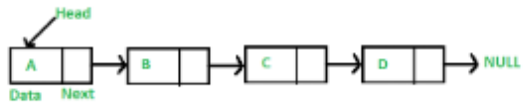
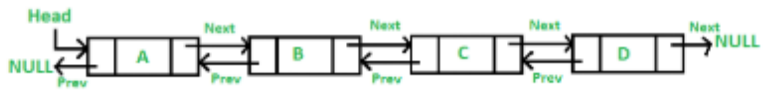
```
// Cipta tiga node
    Node* head = new Node();
    Node* second = new Node();
    Node* third = new Node();

    // Isi data dalam node
    head->data = 1;
    second->data = 2;
    third->data = 3;

    // Sambungkan node
    head->next = second;
    second->next = third;
    third->next = head; // Circular link ke head balik!

    // Cetak hasilnya
    print_list(head);

    return 0;
```

Adjusted Print_list function to display what inside the temp->next

```
void print_list(Node* head) {
    if (head == NULL) return;  // Kalau kosong, keluar awal.

    Node* temp = head;

    // Guna do-while supaya pasti cetak sekurang-kurangnya sekali
    do {
        cout << "Data temp->next : " << temp->next << endl;
        cout << temp->data << " “ <<endl;
        temp = temp->next;
        cout <<"Data temp->next : " << temp->next <<endl;
    } while (temp != head);  // Stop bila balik semula ke head
```

Output


Microsoft Visual Studio Debug Console
Data temp->next : 0000027263CA3DD0
1
Data temp->next : 0000027263CA3560
Data temp->next : 0000027263CA3560
2
Data temp->next : 0000027263CA36F0
Data temp->next : 0000027263CA36F0
3
Data temp->next : 0000027263CA3DD0

Singly linked list (SLL)	Doubly linked list (DLL)
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.
	
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
Supports lesser number of operations in constant time	Supports additional operations like insert before, delete previous, delete current node and delete last in O(1) time. Since it supports delete last, it is used to efficiently implement Deque.
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Complexity of deletion with a given node is O(n), because the previous node needs to be known, and traversal takes O(n)	Complexity of deletion with a given node is O(1) because the previous node can be accessed easily
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.
Singly linked list is relatively less used in practice due to limited number of operations	Doubly linked list is implemented more in libraries due to wider number of operations. For example Java LinkedList implements Doubly Linked List.

Learn some codes

Try to do yourself

- For Doubly Linked List
  - Delete doubly linked list
  - Insert in front or at the end of the list
  - Insert and delete based on user input either by data search or location of the node search
- For Circular Linked List
  - Insert or delete nodes
  - Insert and delete based on user input either by data search or location of the node search
- Any Possible Linked List coding – try and see for yourself.