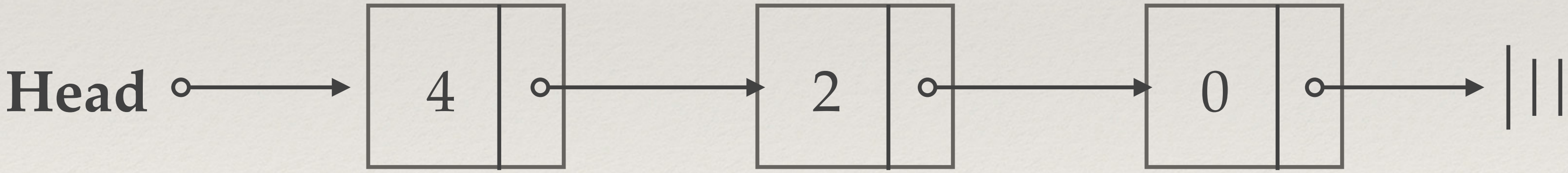
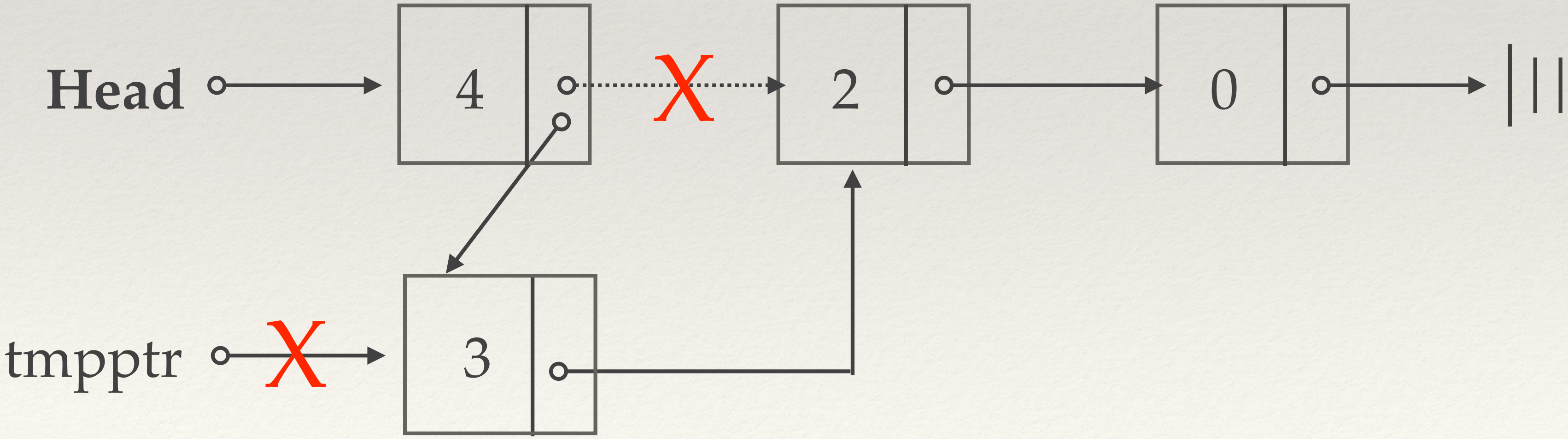
Sorted Lists

- A list is considered to be ordered if they are placed in the list in order based on some part of each element.



Adding to a Sorted List

- Find the two list values that bound the new value.
- Change the pointers to insert the new element.
- You have not changed any of the contents of the elements in the list - just the pointers.

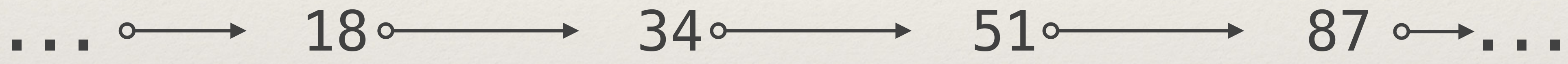


```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Finding where to insert

increasing values

- Step through the list, looking for a value gt the value-to-be-inserted



```
Node * find_gt ( Node * head, int value ){
   Node * node = head;
   while (node != NULL && node->num < value) {
        node = node->next;
   }
   return node;
}
```

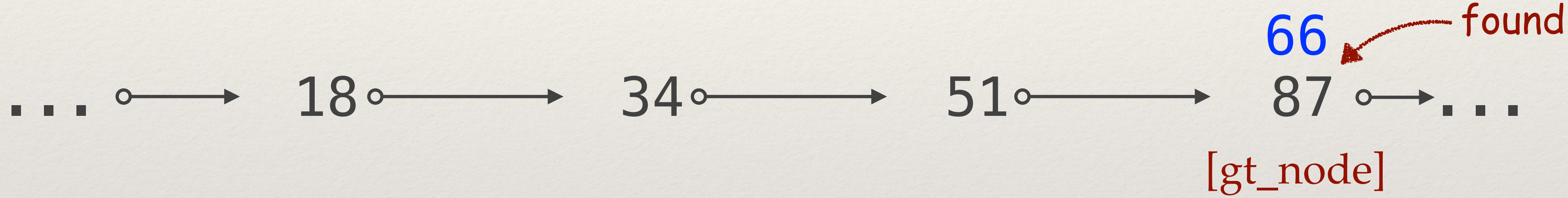
gt = “greater than” (>)

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Finding where to insert

increasing values

- Step through the list, looking for a value gt the value-to-be-inserted



```
Node * find_gt ( Node * head, int value ){
   Node * node = head;
   while (node != NULL && node->num < value) {
        node = node->next;
   }
   return node;
}
```

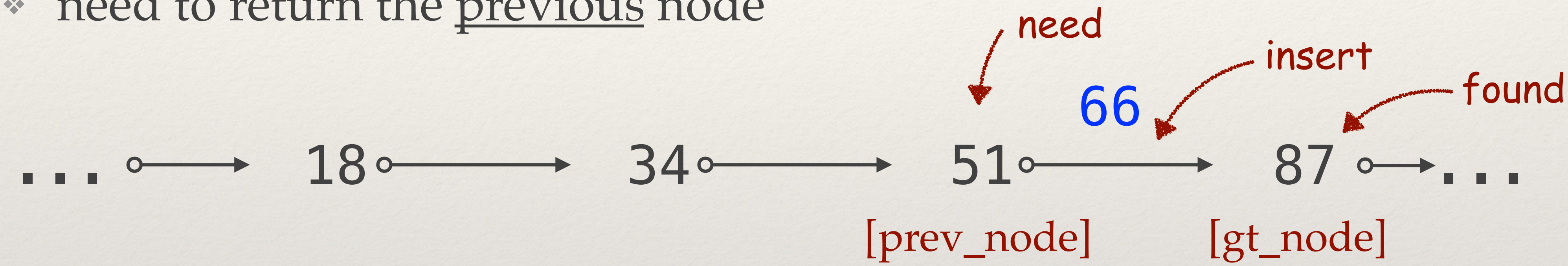
gt = "greater than" (>)

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Finding where to insert

increasing values

- Step through the list, looking for a value gt the value-to-be-inserted
- need to return the previous node



```
Node * find_prev_gt ( Node * head, int value ){
   Node * node = head, * prev = NULL;
   while (node != NULL && node->num < value) {
         prev = node;
         node = node->next;
   }
   return prev;
}
```

gt = "greater than" (>)

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Finding where to insert

increasing values

- Step through the list, looking for a value gt the value-to-be-inserted
- need to return the <u>previous</u> node
- if value is less than the first node, NULL is returned

will need to
change the header

```
Node * find_prev_gt ( Node * head, int value ){
    Node * node = head, * prev = NULL;
    while (node != NULL && node->num < value) {
           prev = node;
           node = node->next;
    }
    return prev;
}
```

gt = "greater than" (>)

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

The code

Adding to Sorted List

increasing values

- Step through the list, looking for a value greater than the value to be inserted
- if value is less than, i.e. should be before, the first node, return NULL

```
void add_sorted ( Node ** head, int value ){
    Node * next_node = NULL;
    Node * insert_node = find_prev_gt(*head, value);
   if (insert_node == NULL) {
        add_front(head, value);
    } else {
        next_node = insert_node->next;
        if (next_node == NULL || next_node->num != value)
            insert_value(insert_node, value);
    }
}
```

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Using add_sorted()

```
Node * head = NULL;

add_sorted(&head, 7);
print_list(head);                  →  < 7 >

add_sorted(&head, 2);
print_list(head);                  →  < 2, 7 >

add_sorted(&head, 10);
print_list(head);                  →  < 2, 7, 10 >

add_sorted(&head, 9);
print_list(head);                  →  < 2, 7, 9, 10 >
```

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

The alternate approach

Adding to Sorted List

increasing values

- Step through the list, looking for a value greater than the value to be inserted
- if value is less than, i.e. should be before, the first node, return NULL

```
Node * add_sorted ( Node * head, int value ){
    Node * next_node = NULL;
    Node * insert_node = find_prev_gt(*head, value);
   if (insert_node == NULL) {
        head = add_front(head, value);
    } else {
        next_node = insert_node->next;
        if (next_node == NULL || next_node->num != value)
            insert_value(insert_node, value);
    }
    return head;
}
```

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

The alternate approach

Using add_sorted()

```
Node * head = NULL;

head = add_sorted(head, 7);
print_list(head);

head = add_sorted(head, 2);
print_list(head);

head = add_sorted(head, 10);
print_list(head);

head = add_sorted(head, 9);
print_list(head);
```

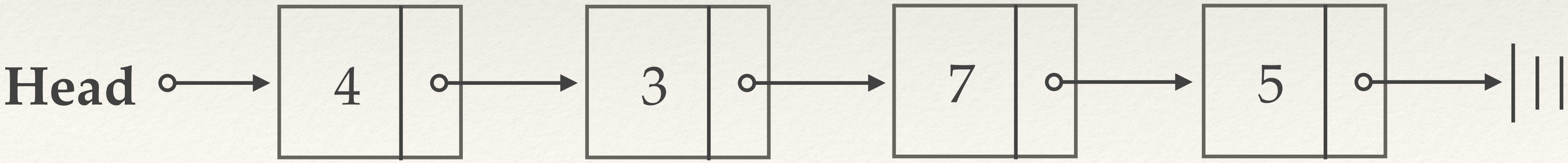
```
< 7 >
< 2, 7 >
< 2, 7, 10 >
< 2, 7, 9, 10 >
```

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

- First we will write a function to remove the **next node**
- Need to return the node being removed (so can be freed if needed)

```
Node * remove_after ( Node * node ){
    Node * remove = node->next;
    node->next = remove->next;
    remove->next = NULL;
    return remove;
}
```



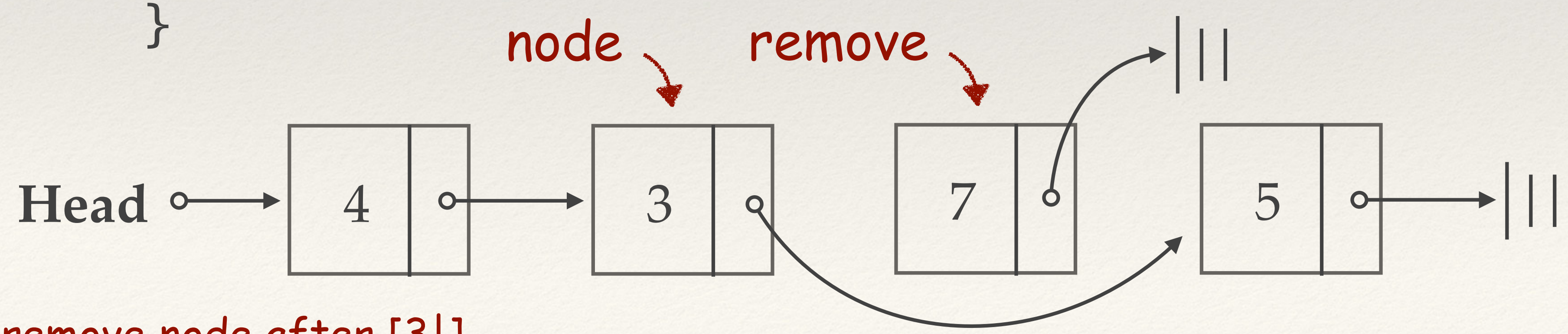
remove node after [3|]

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

- First we will write a function to remove the **next node**
- Need to return the node being removed (so can be freed if needed)

```
Node * remove_after ( Node * node ){
    Node * remove = node->next;
    node->next = remove->next;
    remove->next = NULL;
    return remove;
}
```



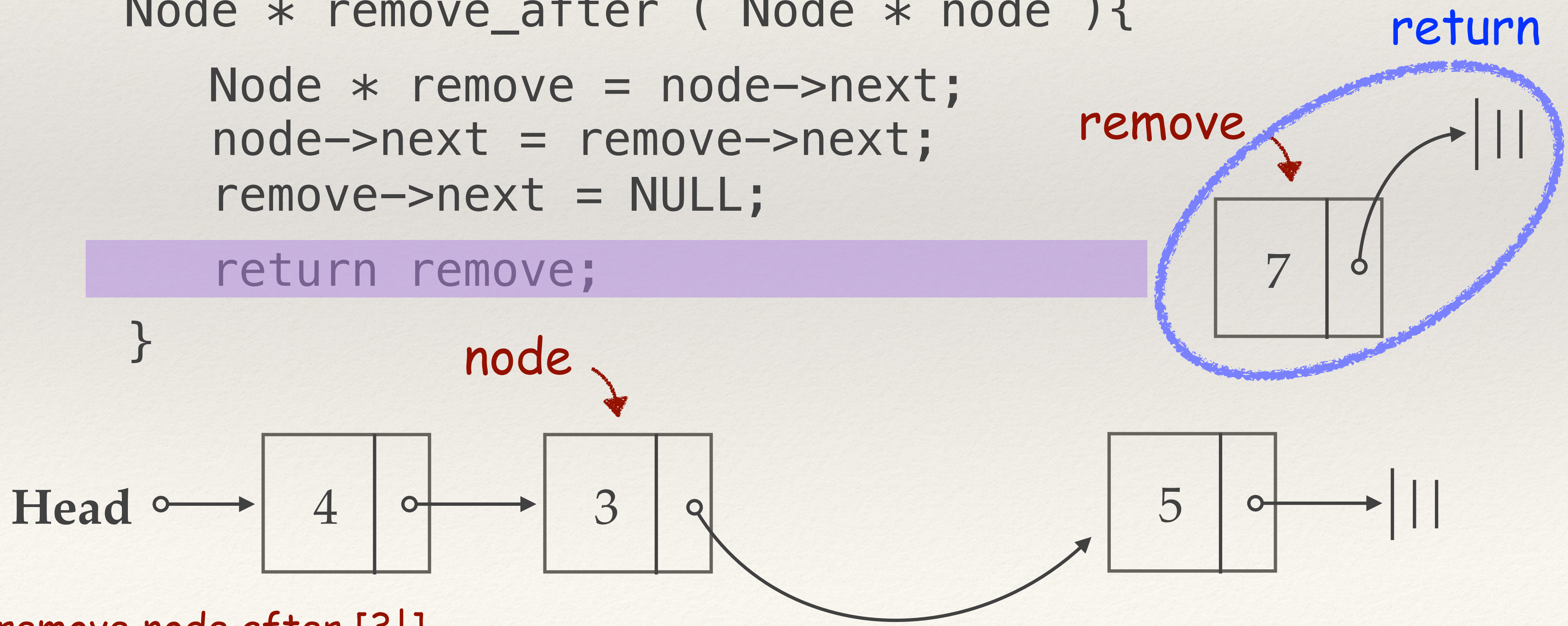
remove node after [3|]

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

- First we will write a function to remove the **next node**
- Need to return the node being removed (so can be freed if needed)

```
Node * remove_after ( Node * node ){
    Node * remove = node->next;
    node->next = remove->next;
    remove->next = NULL;
    return remove;
}
```



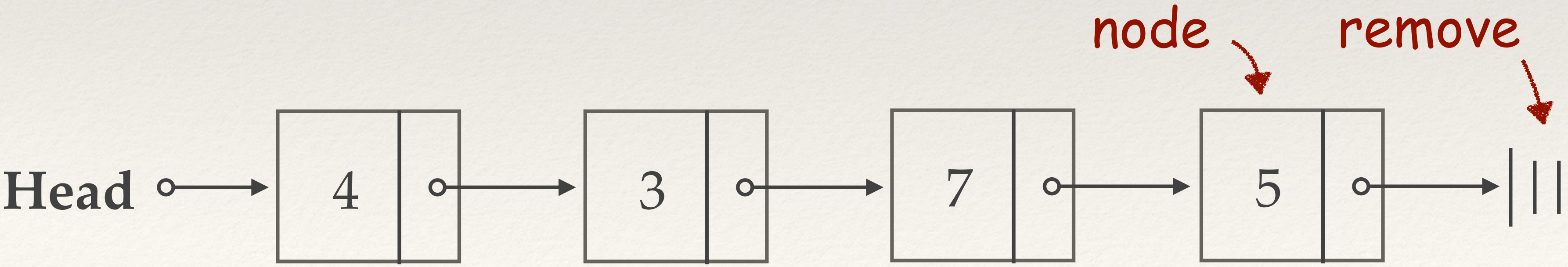
remove node after [3|]

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

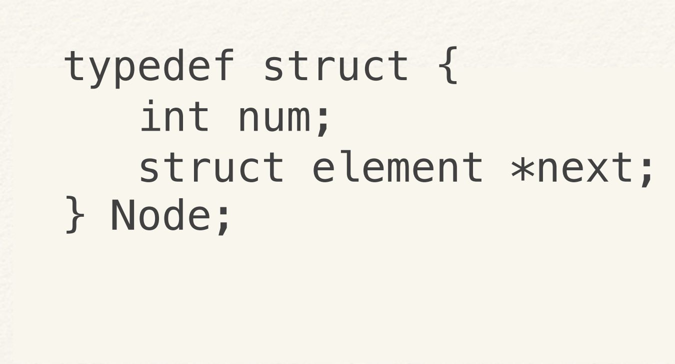
Removing a Node

- In these cases, nothing to remove, so return NULL

```
Node * remove_after ( Node * node ){
    Node * remove = node->next;
    node->next = remove->next;
    remove->next = NULL;
    return remove;
}
```



remove node after [5|]



```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

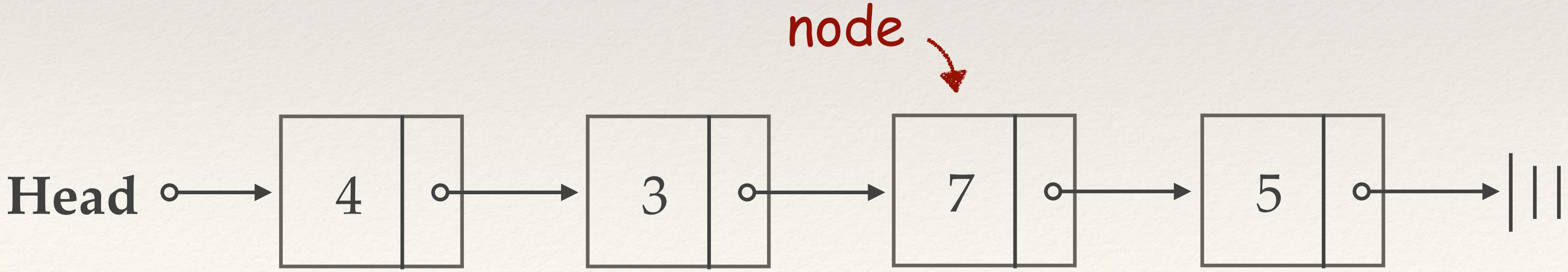
- In these cases, nothing to remove, so return NULL

```
Node * remove_after ( Node * node ){
   Node * remove = (node == NULL) ? NULL : node->next;
   if (remove != NULL) {
      node->next = remove->next;
      remove->next = NULL;
   }
   return remove;
}
```

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

- What if we want to remove the node itself, not the one after?
- Need to find the node before first



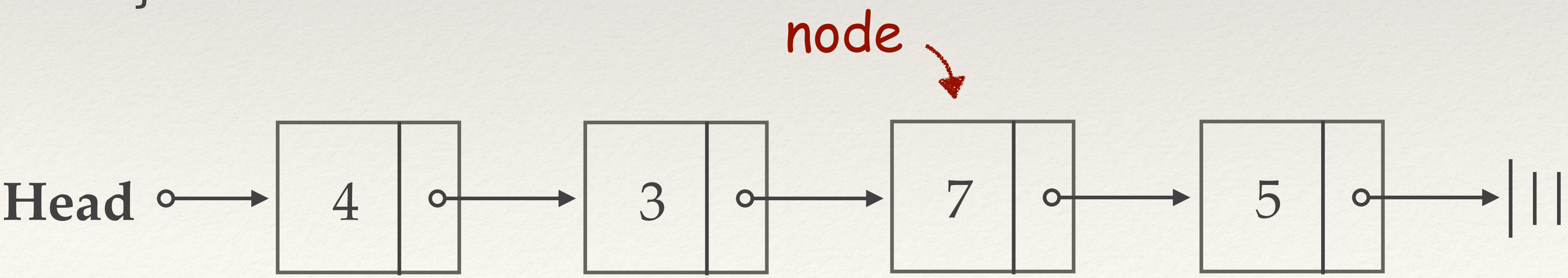
remove node [7|]

```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

- What if we want to remove the node itself, not the one after?
- Need to find the node before first

```
Node * find_before ( Node * head, Node * node){
   Node * prev = head;
   while (prev != NULL && prev->next != node) {
        prev = prev->next;
   }
   return prev;
}
```



remove node [7|]

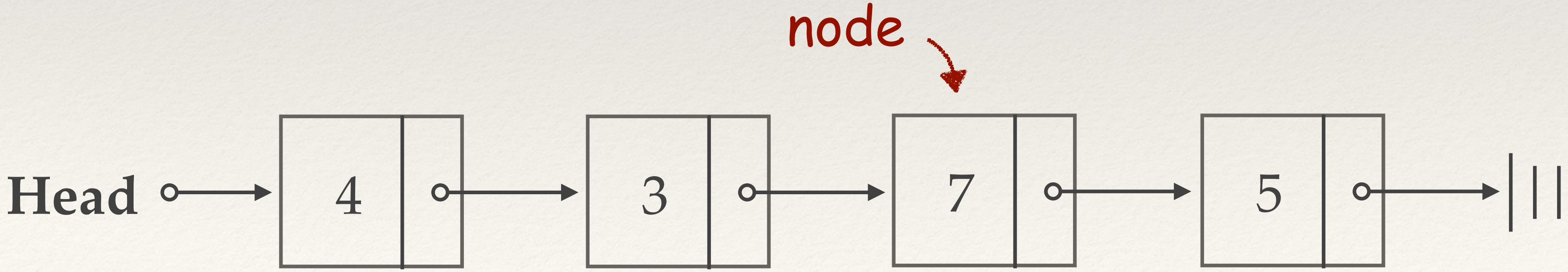
```
typedef struct {
   int num;
   struct element *next;
} Node;
```

Removing a Node

- What if we want to remove the node itself, not the one after?
- The code:

```
Node * remove_node ( Node * head, Node * node){
    Node * prev = find_prev(head, node);
    return remove_after(prev);
}
```

Will return the node if successful
NULL if not in list



remove node [7|]