Introduction to Programming

Week – *9*, Lecture – *1*

Structures in C – Part 2

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Lets revise Structures again…

Structures are a collection of variables of possibly different data types

Lets revise Structures again…

Structures are a collection of variables of possibly different data types

Before creating structure variables, you have to declare its template

Lets revise Structures again…

Structures are a collection of variables of possibly different data types

Before creating structure variables, you have to declare its template

This template provides the exact details of the variables that the structure contains, e.g.

```
struct sample_structure
{
        int i;
        float f;
        char c;
};
```

Lets revise Structures again…

Structures are a collection of variables of possibly different data types

Before creating structure variables, you have to declare its template

This template provides the exact details of the variables that the structure contains, e.g.

```
struct sample_structure
{
        int i;
        float f;
        char c;
};
```

Then, you can create instances of the declared structure as

```
  struct sample_structure s1, s2;
```

Lets revise Structures again…

Structures are a collection of variables of possibly different data types

Before creating structure variables, you have to declare its template

This template provides the exact details of the variables that the structure contains, e.g.

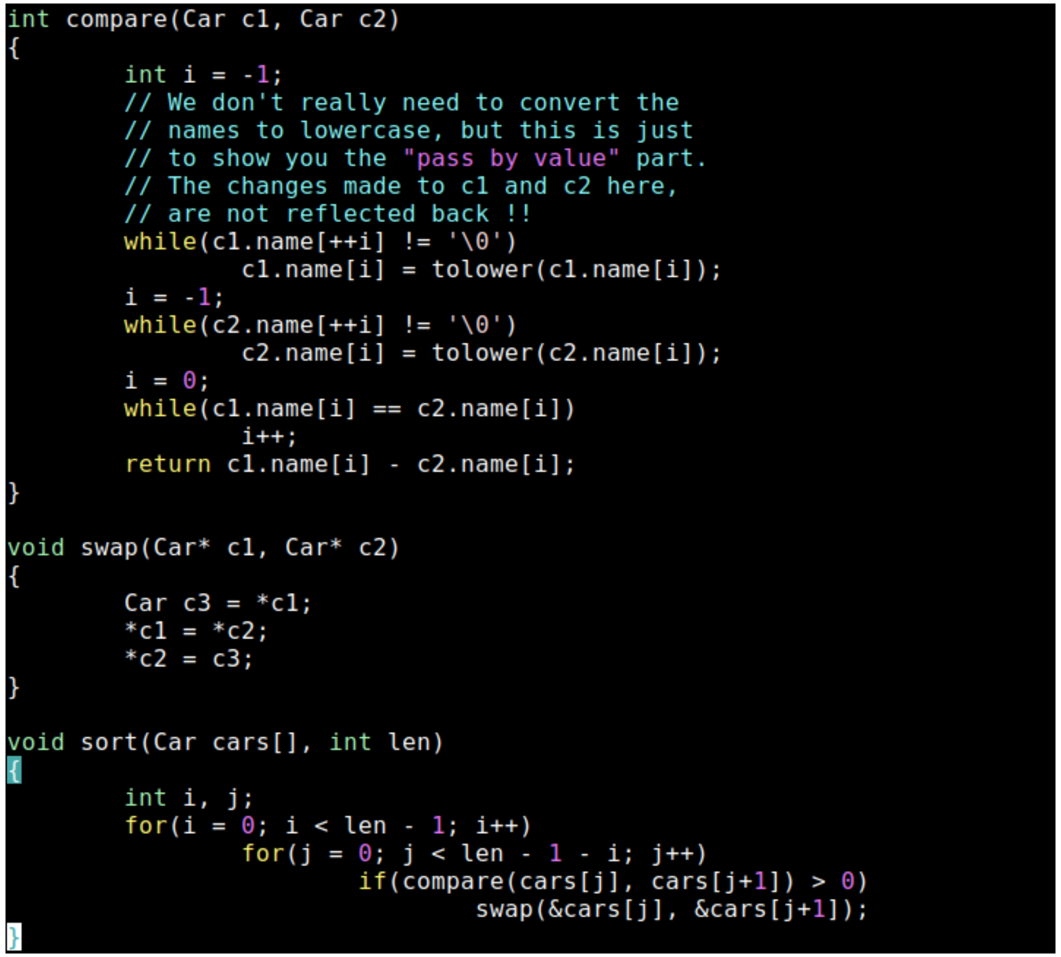
```
struct sample_structure
{
        int i;
        float f;
        char c;
};
```

Then, you can create instances of the declared structure as

```
  struct sample_structure s1, s2;
```

Also, you can use pointers to structures, in the same way, as you use pointers to other variables

Example – Sorting Structure Variables

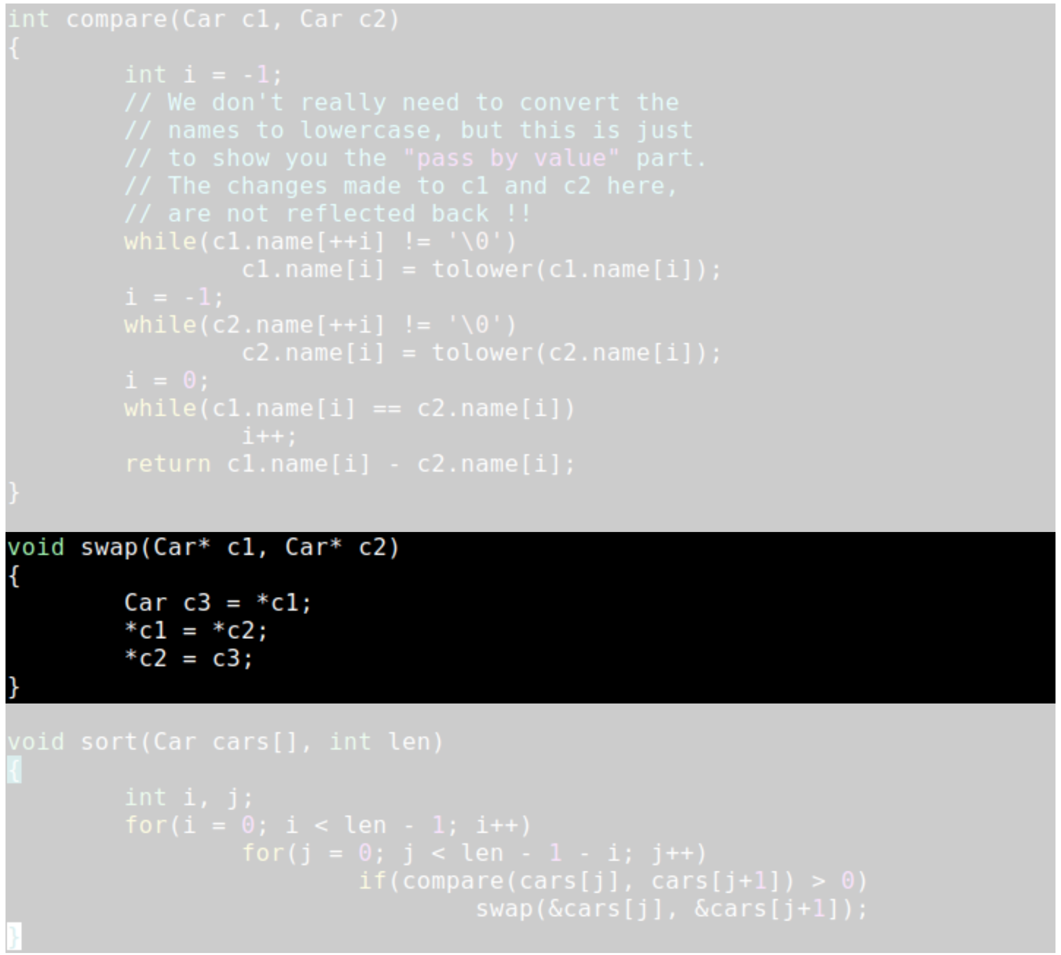


```c
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

Example – Sorting Structure Variables



```c
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

These are some examples of passing structure variables to functions by reference

The pointer to member operator

Remember that if you have a pointer to a structure variable, say `ptr`…

The pointer to member operator

Remember that if you have a pointer to a structure variable, say `ptr`…

- … you can use it like this…

  ```
  (*ptr).i = 6;
  ```

- … to access a particular member of the structure – in this case, the member being `i`

The pointer to member operator

Remember that if you have a pointer to a structure variable, say `ptr`…

- … you can use it like this…

  ```
  (*ptr).i = 6;
  ```

- … to access a particular member of the structure – in this case, the member being `i`

We need the parenthesis, because of some precedence issue

The pointer to member operator

Remember that if you have a pointer to a structure variable, say `ptr`…

- … you can use it like this…

  ```
  (*ptr).i = 6;
  ```

- … to access a particular member of the structure – in this case, the member being `i`

We need the parenthesis, because of some precedence issue

There is alternative to writing the above statement using another operator, which is more readable

- The operator is called the *pointer to member* operator or informally, the “arrow” operator
- Thus, the above statement can also be written as

  ```
  ptr->i = 6;
  ```

The pointer to member operator

Remember that if you have a pointer to a structure variable, say `ptr`…

- … you can use it like this…

  ```
  (*ptr).i = 6;
  ```

- … to access a particular member of the structure – in this case, the member being `i`

We need the parenthesis, because of some precedence issue

There is alternative to writing the above statement using another operator, which is more readable

- The operator is called the *pointer to member* operator or informally, the “arrow” operator
- Thus, the above statement can also be written as

  ```
  ptr->i = 6;
  ```

You may read more about this operator, if you are studying (or planning to study) C++

- In C, it doesn’t serve much purpose, other than a shorthand, and a more readable code

Self-referential Structures

What is a structure template contains a pointer variable?

- It is fine… pointer variables are variables too, just that they store addresses, instead of values

Self-referential Structures

What is a structure template contains a pointer variable?

- It is fine… pointer variables are variables too, just that they store addresses, instead of values

What type of pointer variables can a structure have as member variables?

- It can have pointers to integers, to floating point numbers, to characters…

Self-referential Structures

What is a structure template contains a pointer variable?

- It is fine… pointer variables are variables too, just that they store addresses, instead of values

What type of pointer variables can a structure have as member variables?

- It can have pointers to integers, to floating point numbers, to characters…
- … as well as pointers to structure variables, including that of its own type; for example

```
struct node
{
        int value;
        struct node* next;
};
```

Self-referential Structures

What is a structure template contains a pointer variable?

- It is fine… pointer variables are variables too, just that they store addresses, instead of values

What type of pointer variables can a structure have as member variables?

- It can have pointers to integers, to floating point numbers, to characters…
- … as well as pointers to structure variables, including that of its own type; for example

```
struct node
{
        int value;
        struct node* next;
};
```

Such structures are also called *self-referential structures*…

- … because, a variable of such a structure, can contain a reference to another variable of the same kind

Self-referential Structures

What is a structure template contains a pointer variable?

- It is fine… pointer variables are variables too, just that they store addresses, instead of values

What type of pointer variables can a structure have as member variables?

- It can have pointers to integers, to floating point numbers, to characters…
- … as well as pointers to structure variables, including that of its own type; for example

```
struct node
{
        int value;
        struct node* next;
};
```

Such structures are also called *self-referential structures*…

- … because, a variable of such a structure, can contain a reference to another variable of the same kind
- Actually, you can add many such pointers as you want in the structure declaration

Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

- You create structure variables of a Self-referencing structure, e.g. `node`

Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

- You create structure variables of a Self-referencing structure, e.g. `node`
- You “chain” them together, by storing the address of the next node, in the previous node

Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

- You create structure variables of a Self-referencing structure, e.g. `node`
- You "chain" them together, by storing the address of the next node, in the previous node
- A special pointer, called `head`, points to the first node in the list

Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

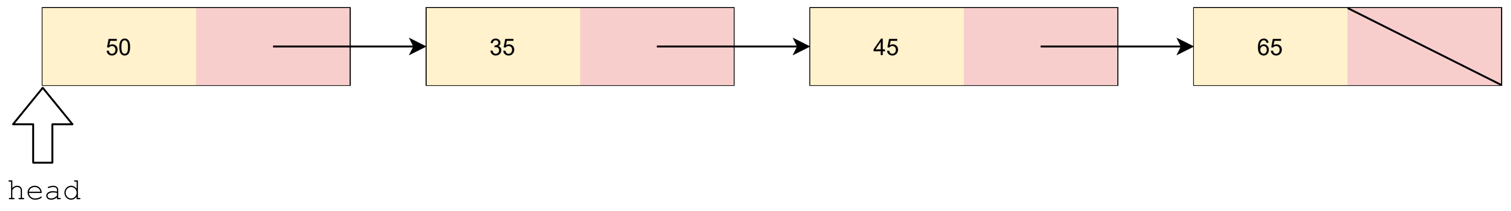
- You create structure variables of a Self-referencing structure, e.g. `node`
- You "chain" them together, by storing the address of the next node, in the previous node
- A special pointer, called `head`, points to the first node in the list
- The last node in the list, have the constant `NULL` stored in it pointer member variable

Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

- You create structure variables of a Self-referencing structure, e.g. `node`
- You "chain" them together, by storing the address of the next node, in the previous node
- A special pointer, called `head`, points to the first node in the list
- The last node in the list, have the constant `NULL` stored in it pointer member variable
- Something like this:

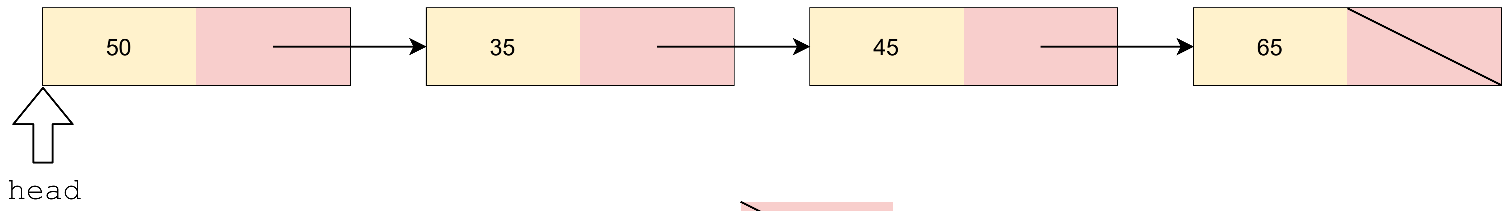


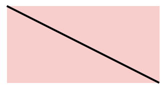
Example of Self-Referencing – Linked List

There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

- You create structure variables of a Self-referencing structure, e.g. `node`
- You "chain" them together, by storing the address of the next node, in the previous node
- A special pointer, called `head`, points to the first node in the list
- The last node in the list, have the constant `NULL` stored in it pointer member variable
- Something like this:



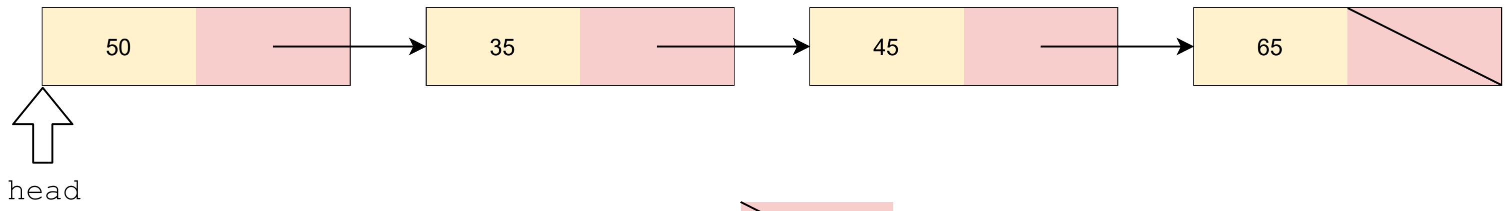
The  signifies `NULL`

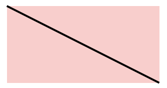
Example of Self-Referencing – Linked List

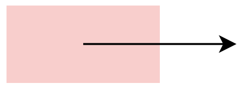
There is an important data structure, that relies on the concept of self-referencing

The idea can be summarised as below:

- You create structure variables of a Self-referencing structure, e.g. `node`
- You "chain" them together, by storing the address of the next node, in the previous node
- A special pointer, called `head`, points to the first node in the list
- The last node in the list, have the constant `NULL` stored in it pointer member variable
- Something like this:



The  signifies `NULL`

The  signifies a pointer to the next structure node

```
head = NULL
```

```
insert_at_head(50)
```

In the beginning, the `head` of the linked list points to `NULL`

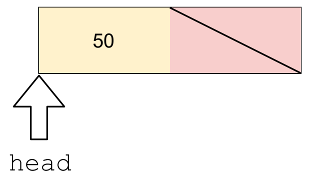
```
head = NULL
                 insert_at_head(50)
```

In the beginning, the `head` of the linked list points to `NULL`

The first insertion, needs to be done at the `head` of the list

```
head = NULL
```

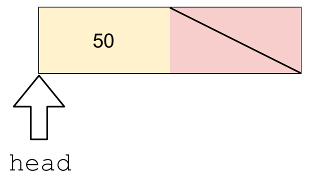
`insert_at_head(50)`



A structure variable is created, and its data elements are filled with the requested data

```
head = NULL
```

```
insert_at_head(50)
```

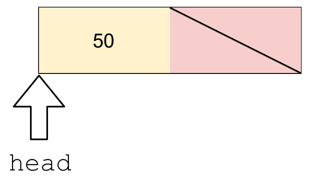


A structure variable is created, and its data elements are filled with the requested data

The `next` pointer (a member variable of the structure) is set to `NULL`

```
head = NULL
```

`insert_at_head(50)`



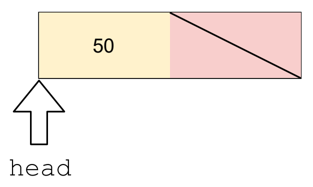
A structure variable is created, and its data elements are filled with the requested data

The `next` pointer (a member variable of the structure) is set to `NULL`

Since it is the first node in the list, we make the `head` variable point to it

```
head = NULL
```

`insert_at_head(50)`



A structure variable is created, and its data elements are filled with the requested data

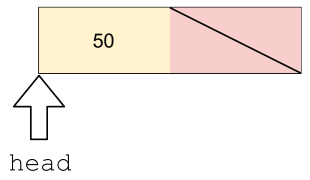
The `next` pointer (a member variable of the structure) is set to `NULL`

Since it is the first node in the list, we make the `head` variable point to it

Also, usually we allocate the space for the structure variable dynamically, using `malloc()`

```
head = NULL
```

`insert_at_head(50)`



A structure variable is created, and its data elements are filled with the requested data

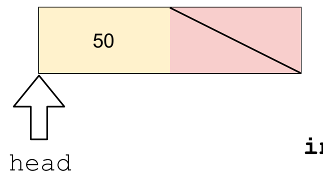
The `next` pointer (a member variable of the structure) is set to `NULL`

Since it is the first node in the list, we make the `head` variable point to it

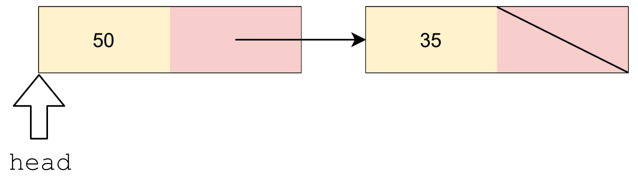
Also, usually we allocate the space for the structure variable dynamically, using `malloc()`

This saves us the hassle of managing scopes `head = NULL`

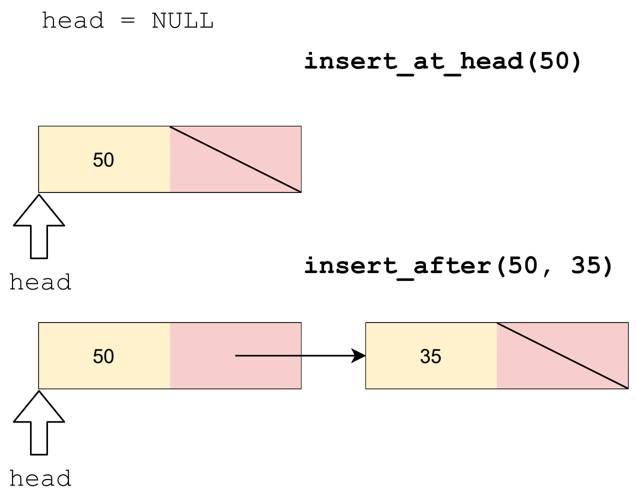
`insert_at_head(50)`



`insert_after(50, 35)`

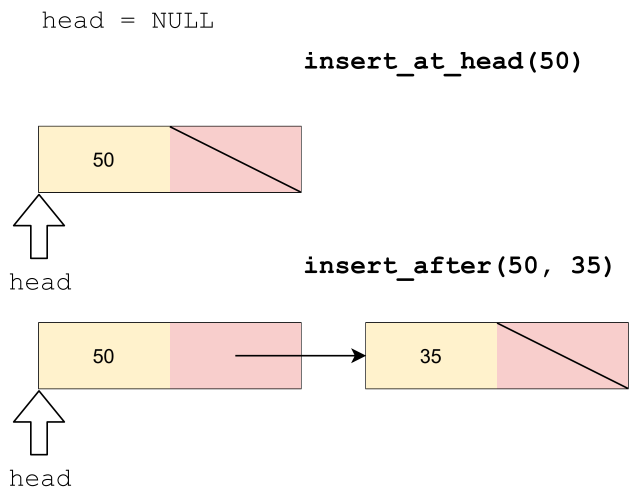


The subsequent insertions expect that the user provides us with the value of the ”previous” node



The subsequent insertions expect that the user provides us with the value of the ”previous” node

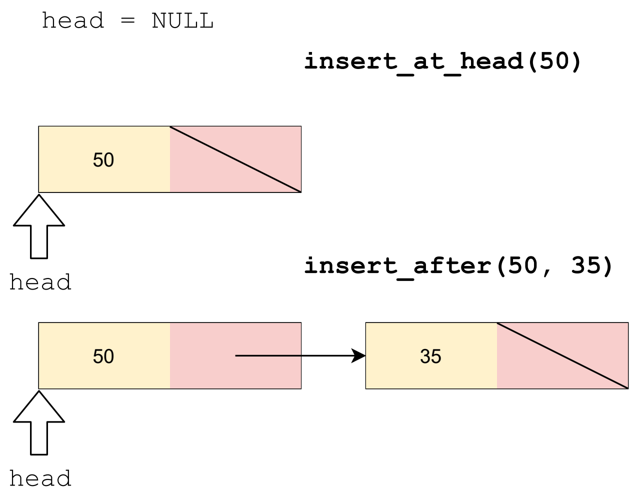
For example, here, the value `35`, is supposed to be inserted next to the value `50`



The subsequent insertions expect that the user provides us with the value of the ”previous” node

For example, here, the value `35`, is supposed to be inserted next to the value `50`

To do the insertion, we start checking the data of the nodes, starting from the `head`…

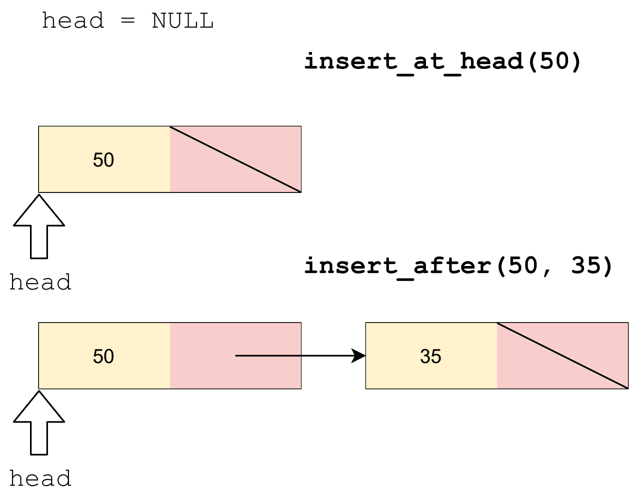


The subsequent insertions expect that the user provides us with the value of the "previous" node

For example, here, the value `35`, is supposed to be inserted next to the value `50`

To do the insertion, we start checking the data of the nodes, starting from the `head`…

… and we follow the `next` pointers to go from one node to another, till we find the right data



The subsequent insertions expect that the user provides us with the value of the ”previous” node

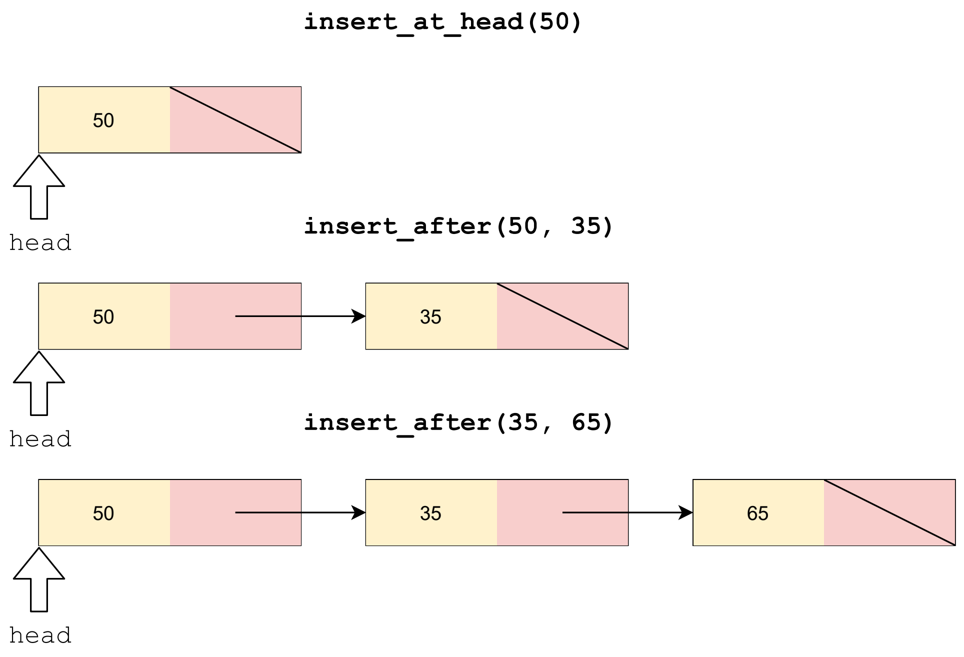
For example, here, the value `35`, is supposed to be inserted next to the value `50`

To do the insertion, we start checking the data of the nodes, starting from the `head`…

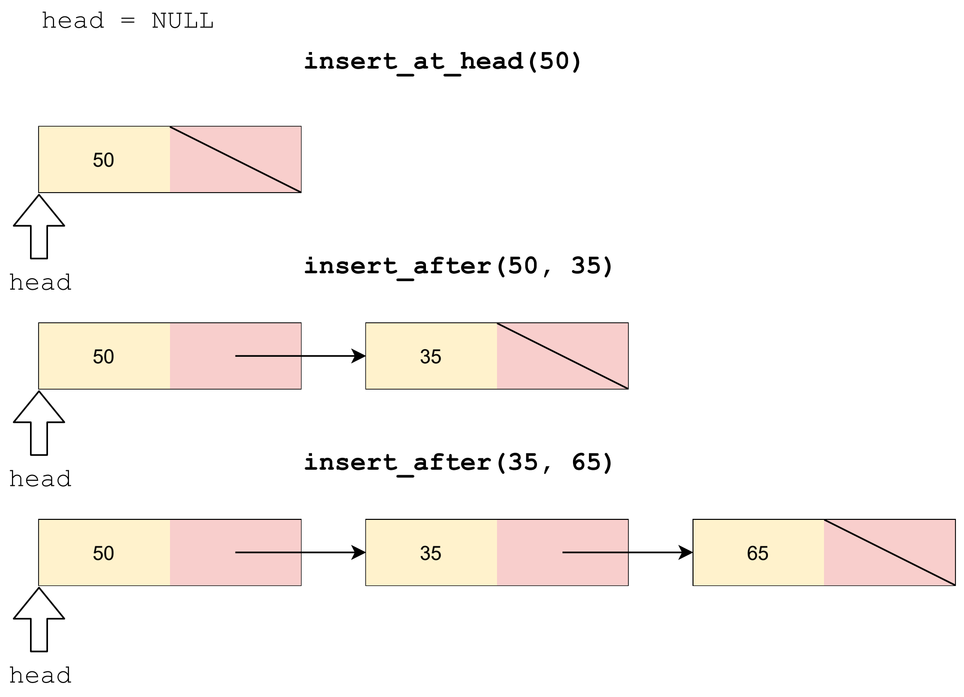
… and we follow the `next` pointers to go from one node to another, till we find the right data

If we cannot find a suitable previous node, we may signal that the insertion has failed

```
head = NULL
```

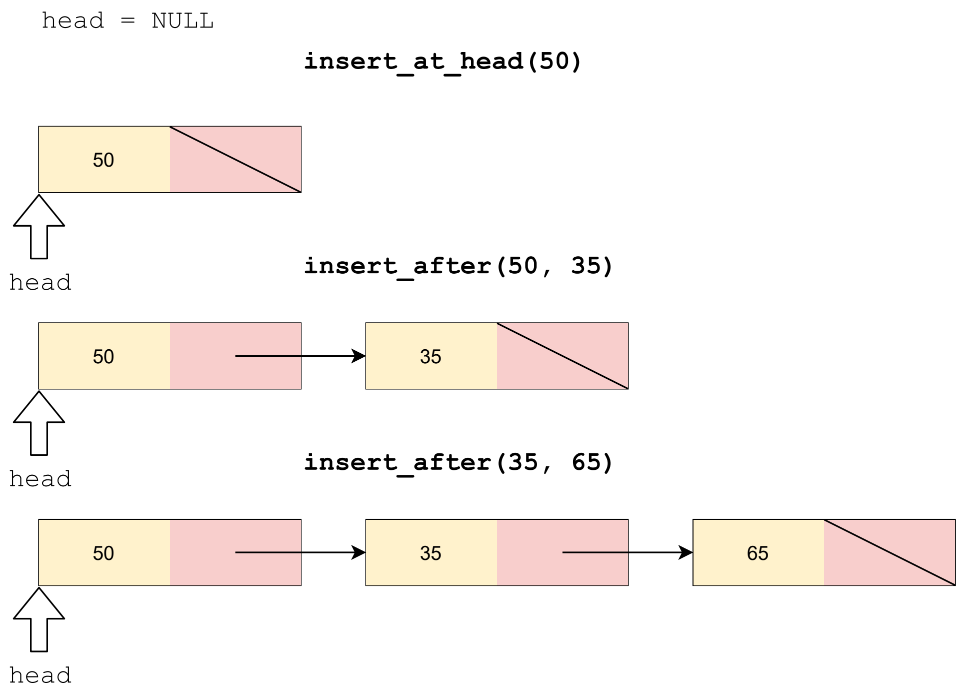


Another example of insertion… this time we wish to insert `65` after `35` in the linked list



Another example of insertion… this time we wish to insert `65` after `35` in the linked list

Remember that if we expect multiple nodes for the same value, we need to have an insertion policy

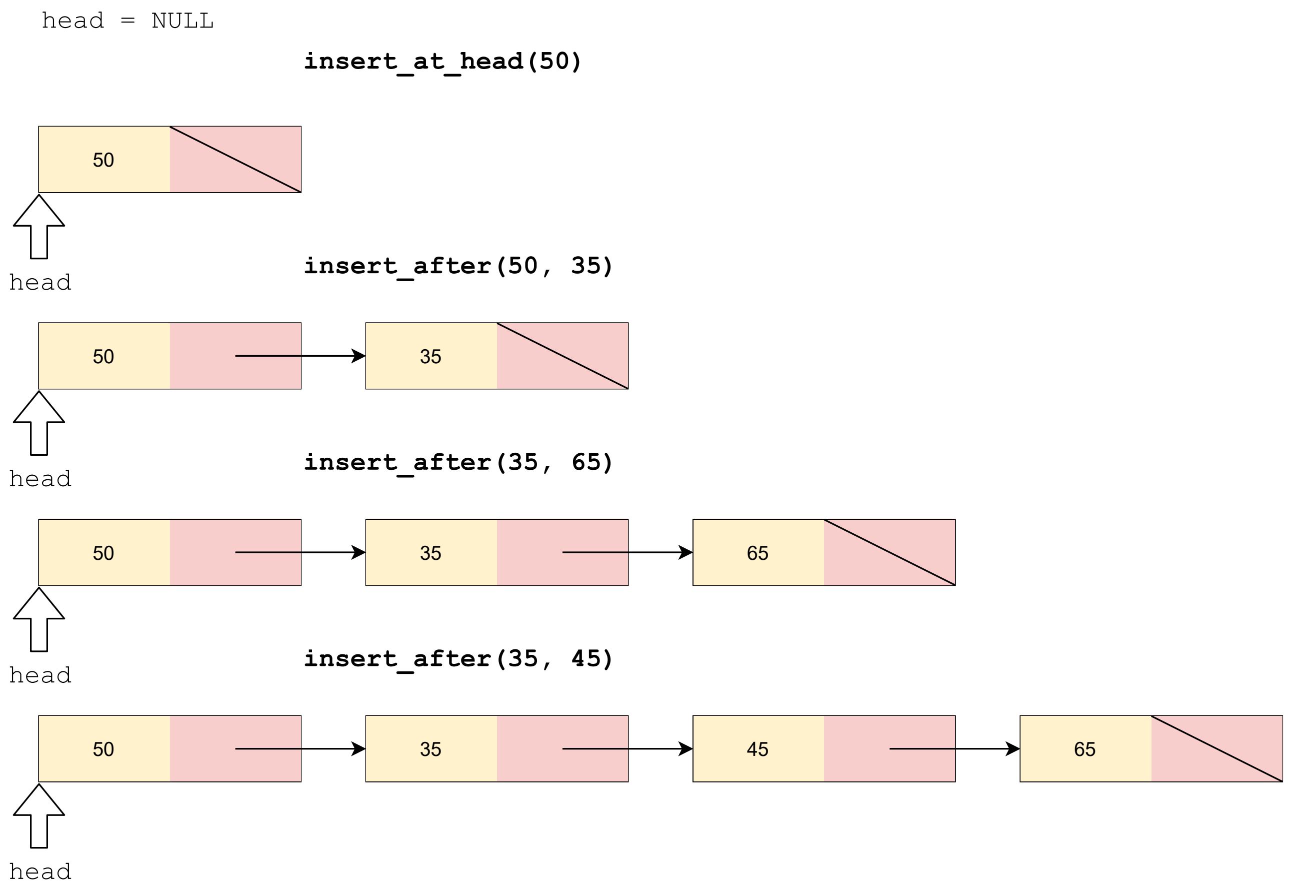


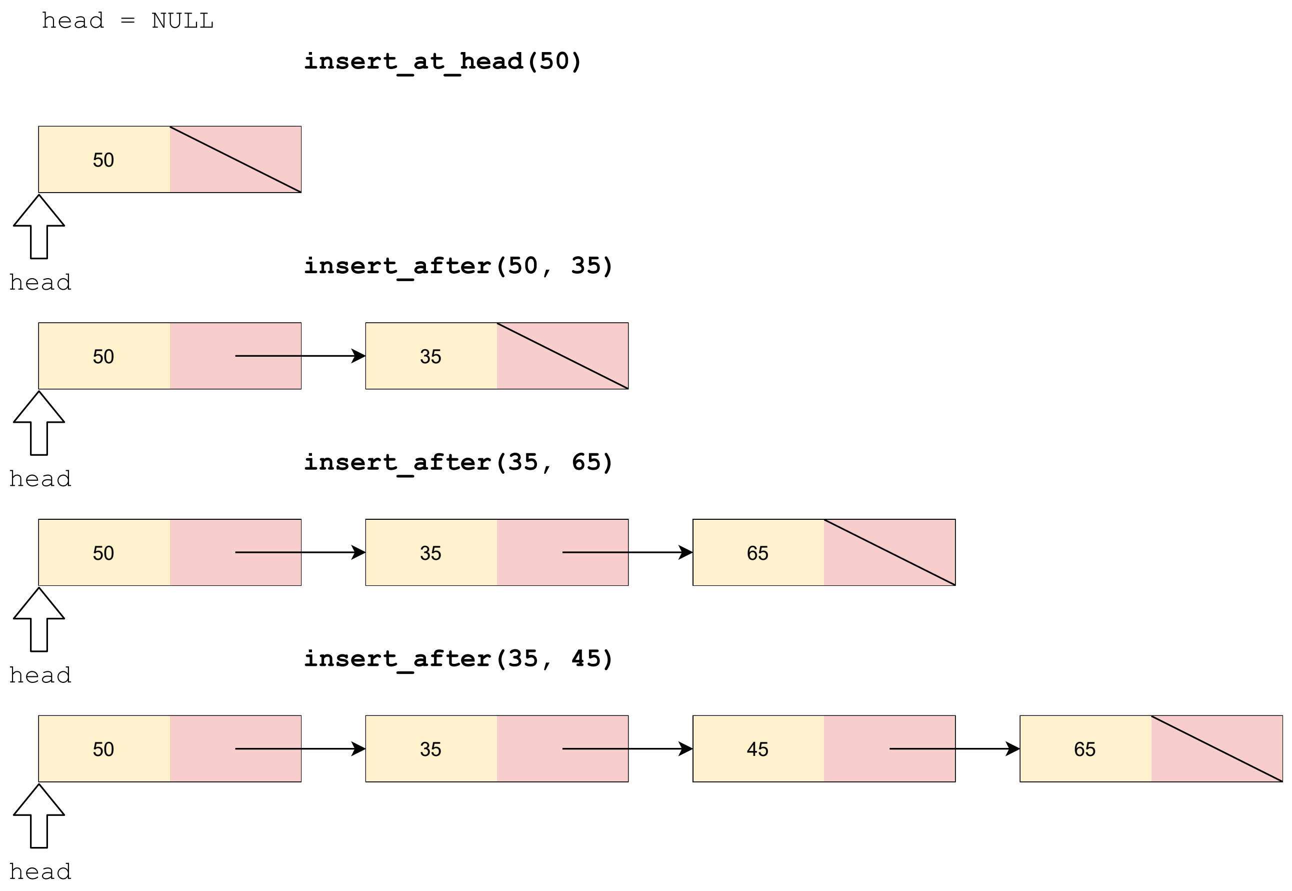
Another example of insertion… this time we wish to insert 65 after 35 in the linked list

Remember that if we expect multiple nodes for the same value, we need to have an insertion policy

For example, the simplest policy is to insert it after the first encountered instance (starting from head)

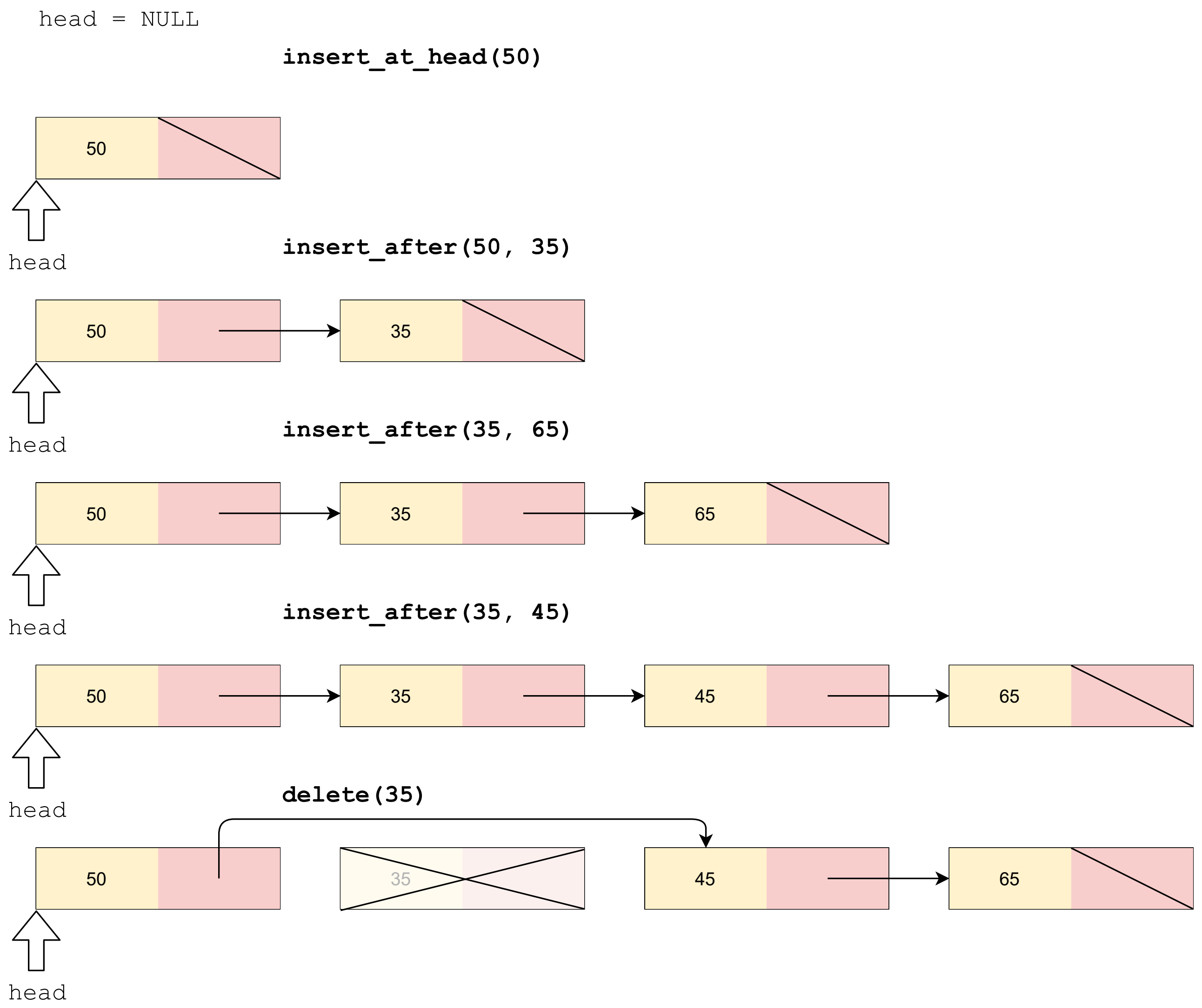
... and one more example of insertion



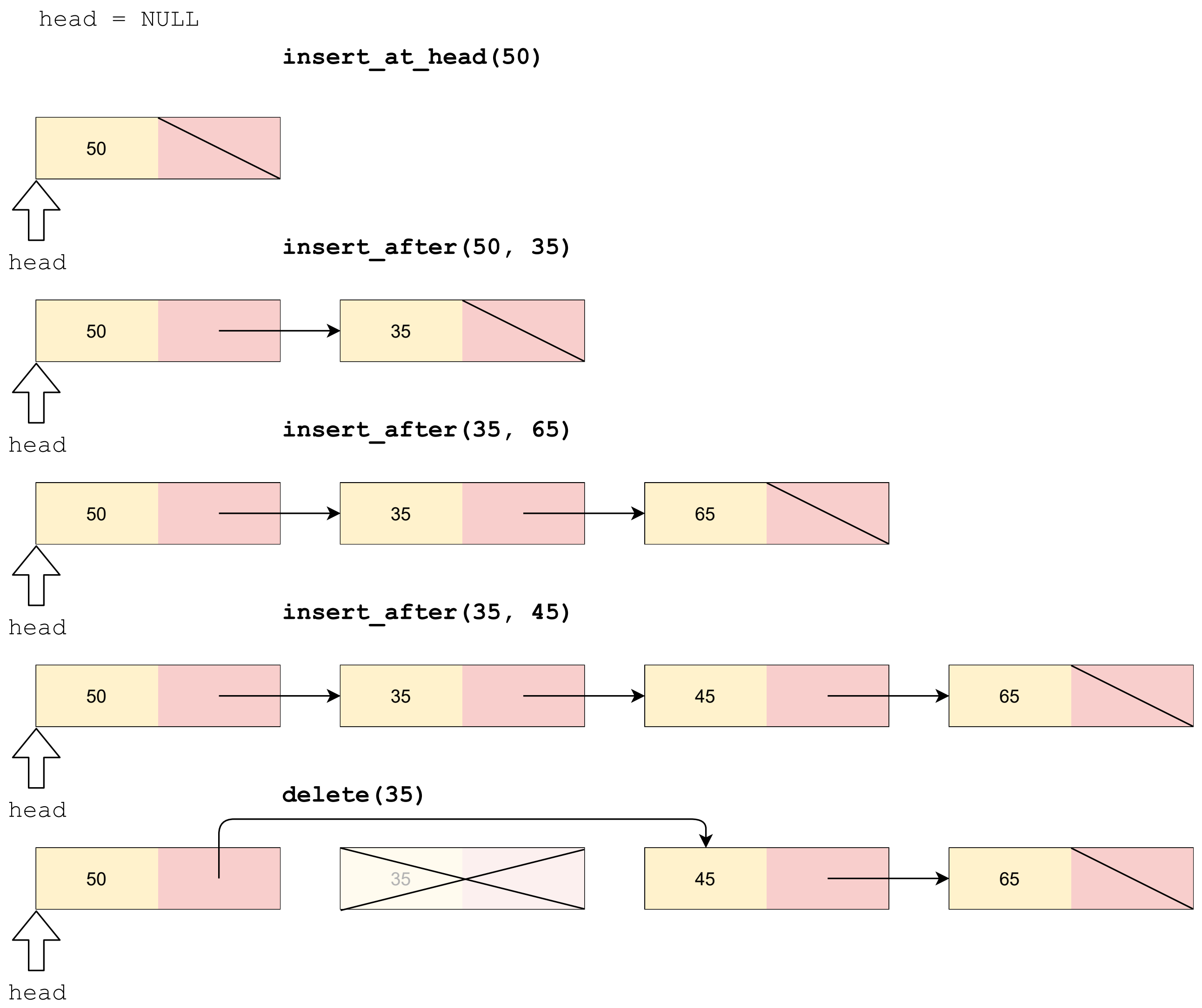


… and one more example of insertion

Convince yourself that you understand how the insertion operation works in a linked list

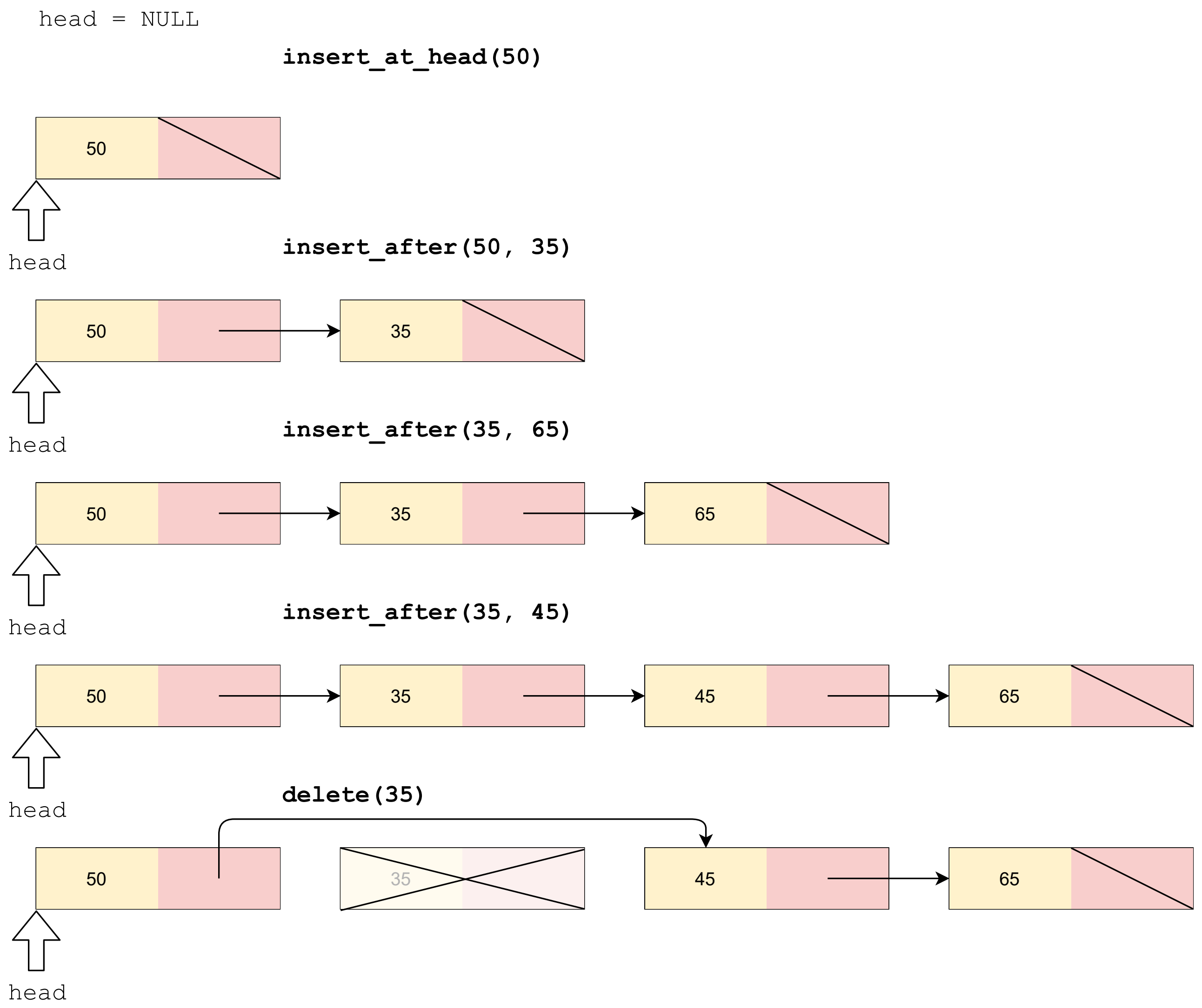


During the deletion, the user provides the value which should be deleted



During the deletion, the user provides the value which should be deleted

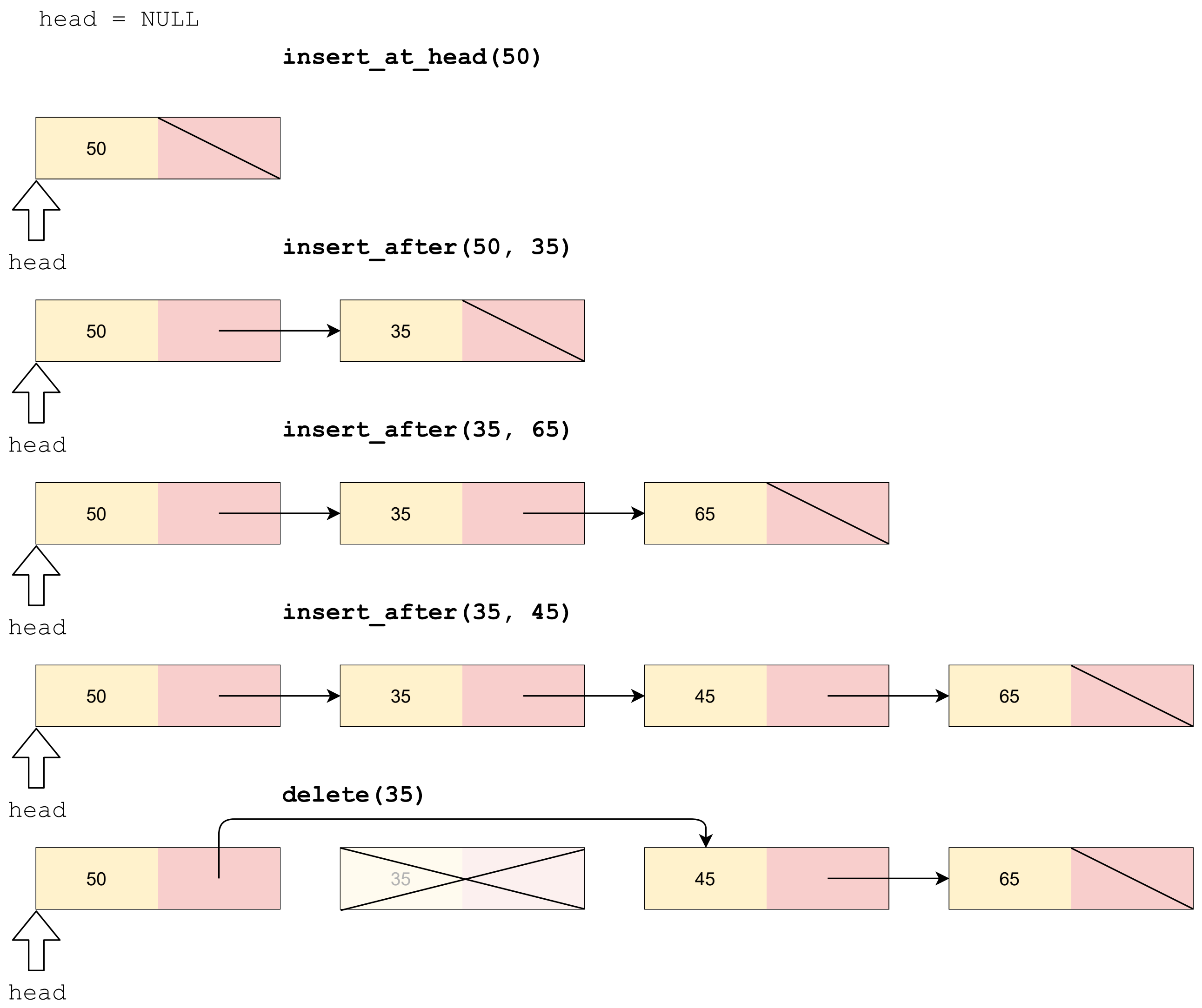
Similar to the insertion operation, the first step is to browse the list, starting from `head`…



During the deletion, the user provides the value which should be deleted

Similar to the insertion operation, the first step is to browse the list, starting from `head`…

… and following the `next` pointers, get to the node "previous" to the node to de deleted

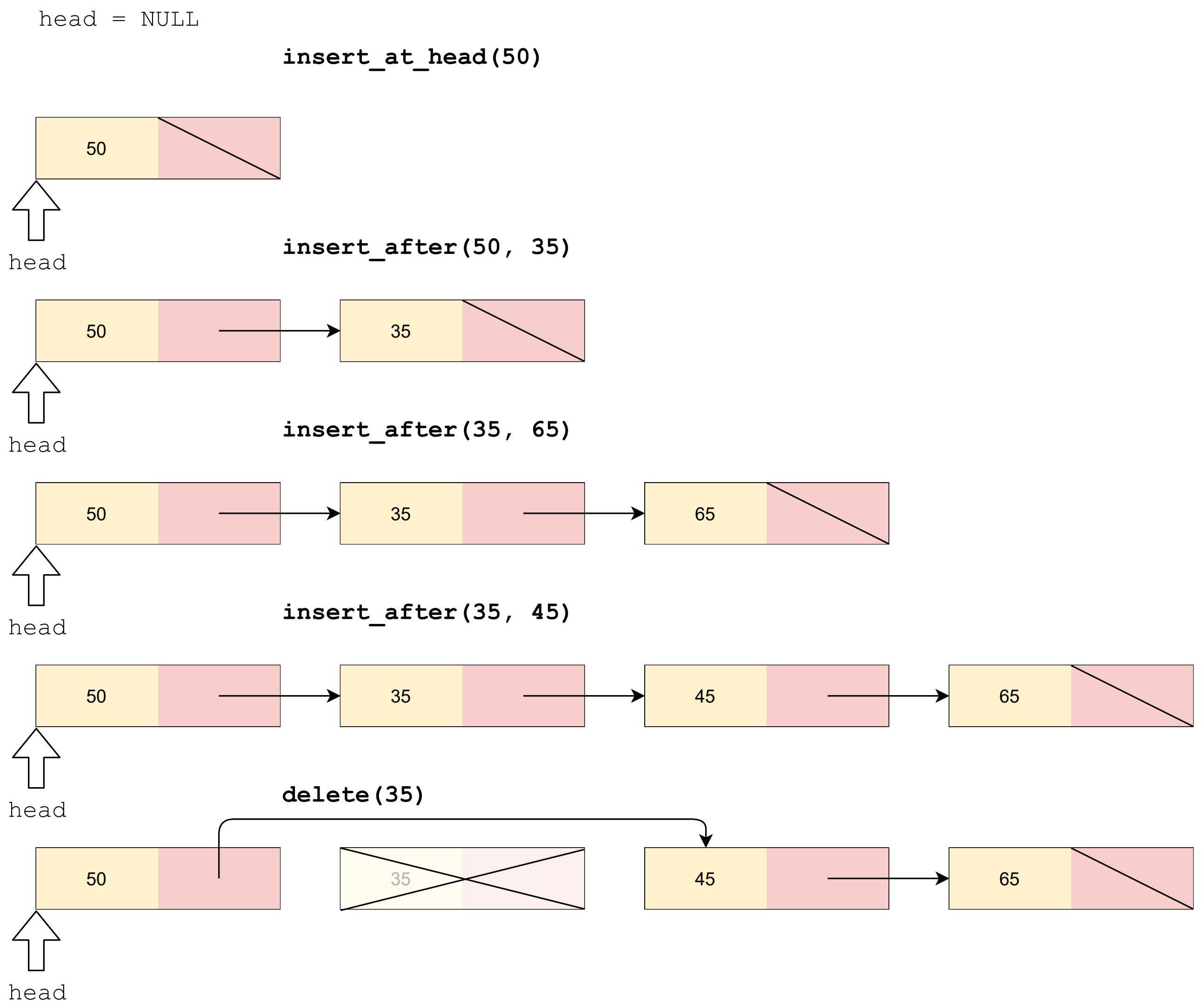


During the deletion, the user provides the value which should be deleted

Similar to the insertion operation, the first step is to browse the list, starting from `head`…

… and following the `next` pointers, get to the node "previous" to the node to de deleted

Here, that node is the one with value `50`, whose next pointer is changed to point to to `45`



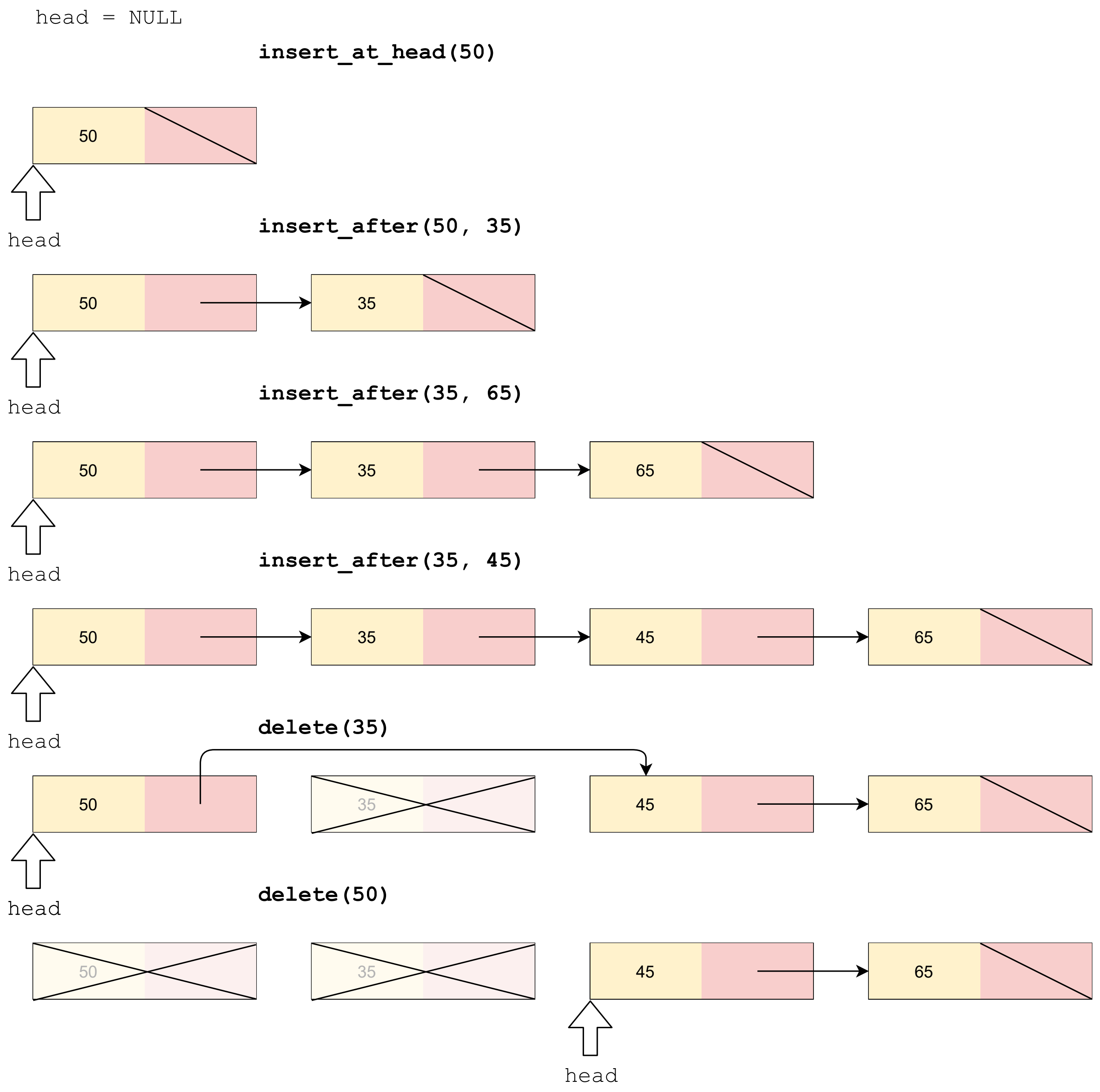
During the deletion, the user provides the value which should be deleted

Similar to the insertion operation, the first step is to browse the list, starting from `head`…

… and following the `next` pointers, get to the node "previous" to the node to de deleted

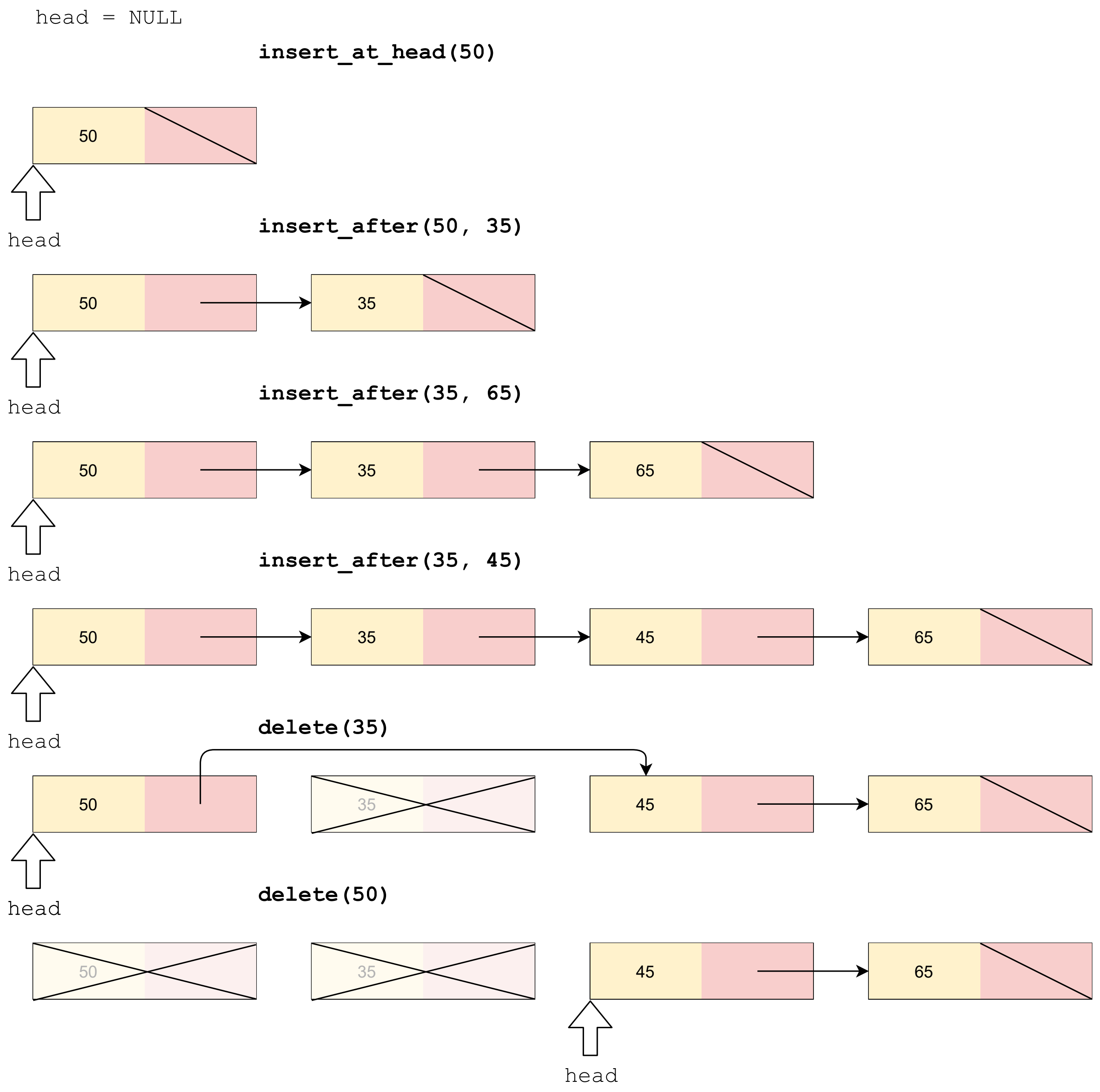
Here, that node is the one with value `50`, whose next pointer is changed to point to to `45`

Finally, the node with value 35 is deallocated, say by calling `free()` (if it was allocated dynamically)



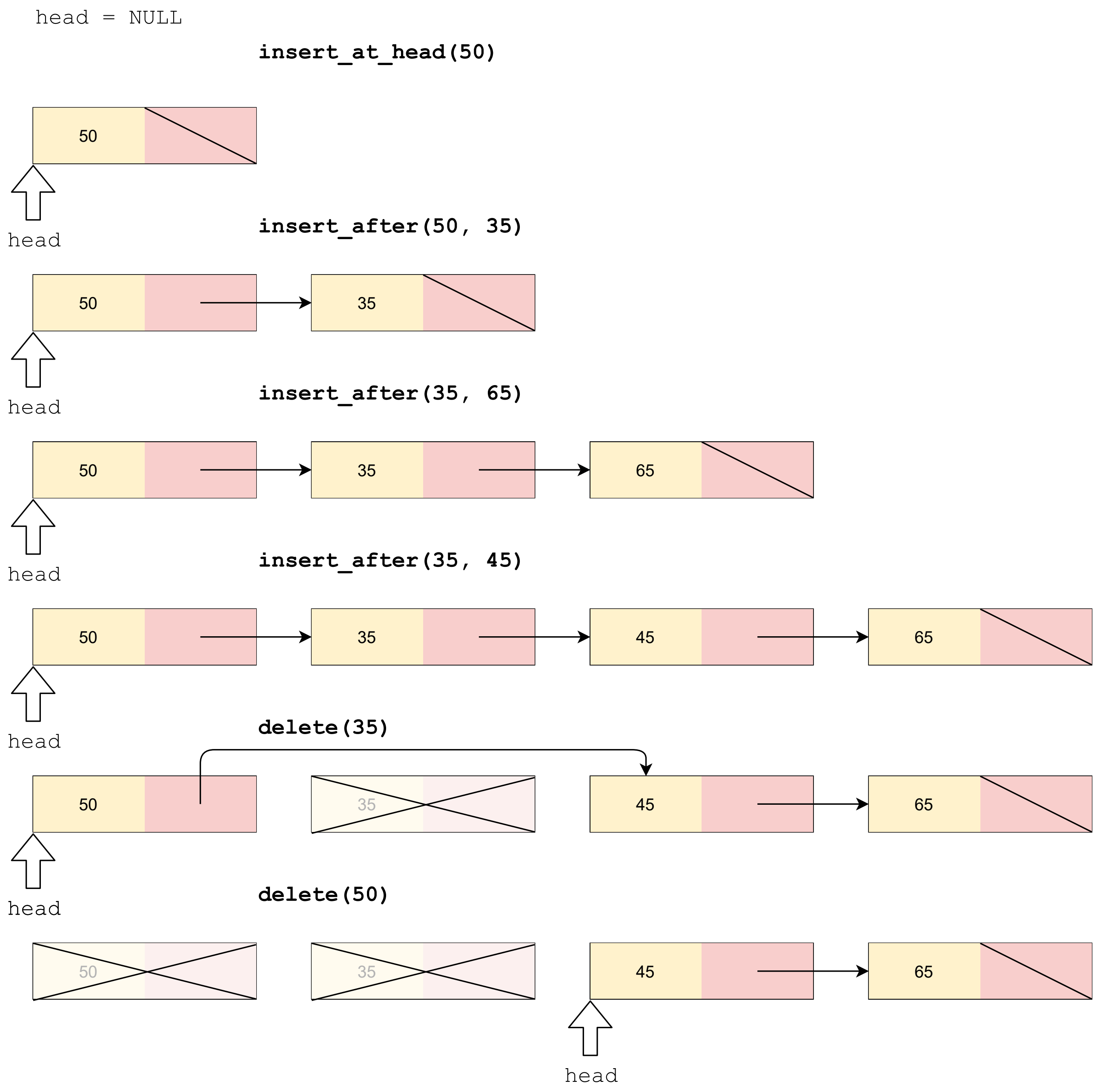
```
head = NULL
insert_at_head(50)
insert_after(50, 35)
insert_after(35, 65)
insert_after(35, 45)
delete(35)
delete(50)
```

… another deletion example, this time there is no "previous" node though, since it is the first node



… another deletion example, this time there is no "previous" node though, since it is the first node

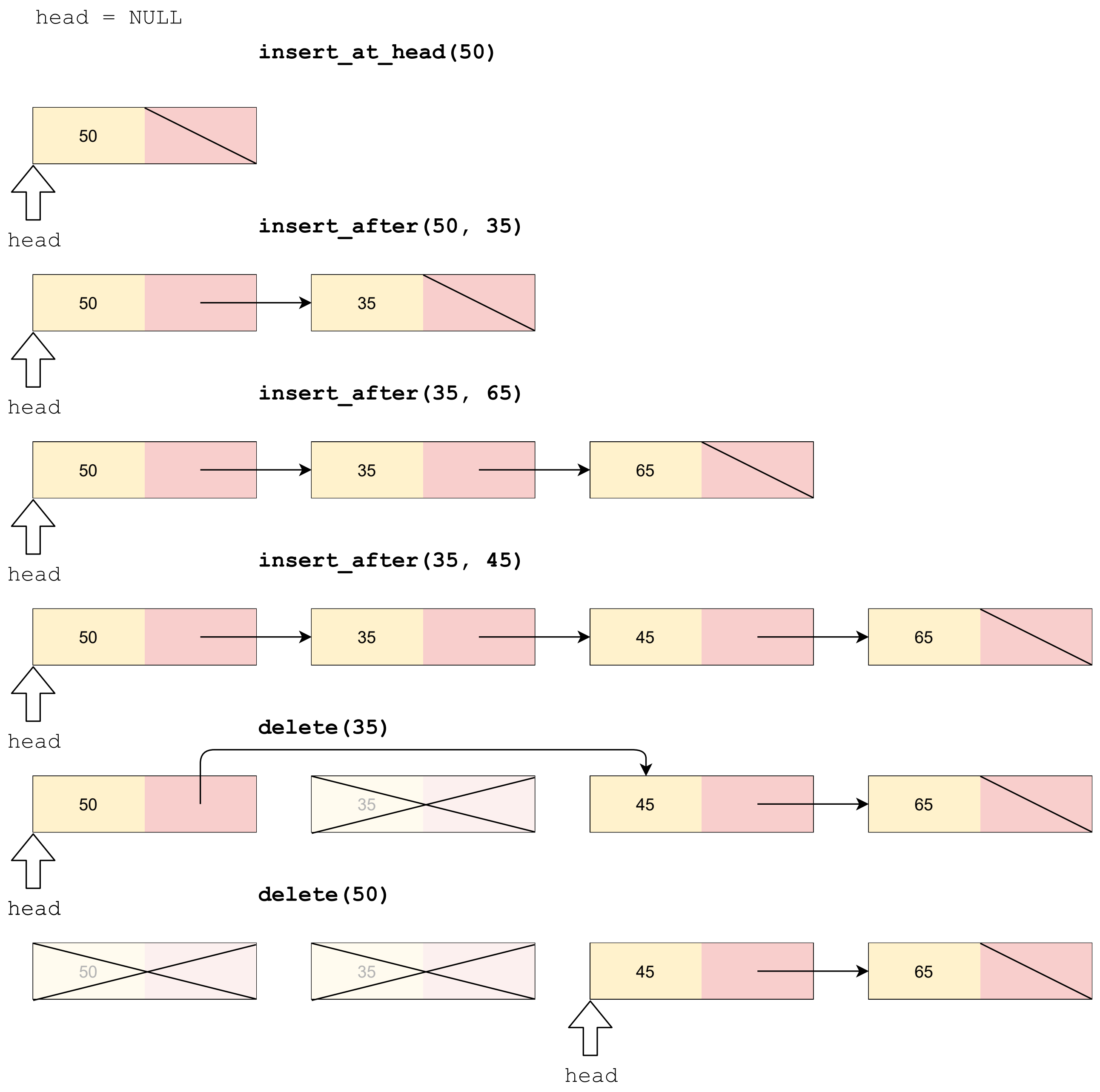
Here, we simply point the `head` to the next node in the list, and free the current `head` node



… another deletion example, this time there is no "previous" node though, since it is the first node

Here, we simply point the `head` to the next node in the list, and free the current `head` node

Similar to insertion, we need to set a policy for handling multiple instances of the same value



… another deletion example, this time there is no "previous" node though, since it is the first node

Here, we simply point the `head` to the next node in the list, and free the current `head` node

Similar to insertion, we need to set a policy for handling multiple instances of the same value

Again, the simplest policy is to delete the first instance encountered while browsing from `head`

Homework !!

I have shared a simple implementation for a linked list in the Google Drive folder

- It is called `SimpleLinkedList.c`
- Read the code and run the program to understand its working
- (if you find a bug, send me an email, I'll fix it; I wrote the code in a hurry, so couldn't test it satisfactorily)

Additional Reading

There is an annoying problem with the linked list that we discussed, also called *singly linked list*

- We can only move in one direction, i.e. starting from the head, and moving forward

There is another version of linked lists, called the *doubly linked list*

- Doubly linked lists have two pointers in each node, one for the next, and one for the previous node

Read more about doubly linked lists

- You may start here: https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/