Fundamentals of Programming 2

Doubly Linked Linear List

Arkadiusz Chrobot

Department of Computer Science

April 6, 2020

Outline

Introduction

The doubly linked linear list is another example of a linear list. The singly linked linear list and the doubly linked linear list are quite similar. From the user point of view the main difference between those two lists is that the doubly linked list may be easily traversed forward and backward. In this lecture the implementation of the doubly linked linear list in a form of a dynamically allocated data structure is presented.

Implementation

Just like the singly linked list, the doubly linked list is presented with the use of an example program, that uses it to store natural numbers in the ascending order. In other words, it is a sorted list. Similarly as in the case of singly linked list, first the base type is defined and the list pointer is declared. Next, the five basic operation for the list are implemented: creating the list, adding a single element to the list, removing a single element from the list, printing the content of the list on the screen and removing the list.

Base Type and List Pointer

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  struct list_node
5  {
6      int data;
7      struct list_node *previous, *next;
8  } *list_pointer;
```

Base Type

The base type of the doubly linked linear list, that is presented in the previous slide, differs from the base types of previously introduced data structures by having an additional pointer field. The field, which is called `previous` in the example base type is used for storing an address of a list element that precedes a given element. In the doubly linked linear list there can be only one element that stores the `NULL` value in that field. It is the first element in the list. There is a kind of the doubly linked linear list that has the same base data type as the singly linked linear list. It is called an XOR *linked list*. In the single pointer field is stored not a single address but a result of the bitwise xor operation performed on addresses of preceding and succeeding elements of a given element. The XOR list is more memory-effective than classical doubly linked linear list, but operations on the list are more time-consuming and complicated. The former list won't be described any more in the lecture. The presented base type can be modified or expanded provided that both pointer fields used for building the list are preserved.

List pointer

The list pointer (the `list_pointer` variable) is declared in the same slide where the base type is defined. Unlike in the case of the singly linked linear list it doesn't have to always point to the first element of the double linked linear list, indeed it may point to *any* element in the list. However, it is convenient if every operation on the list leaves the pointer pointing to the beginning of the list. This approach is applied in the demonstrated program.
In the slide that contains the definition of the base type and the declaration of the list pointer also the statements that include header files with the declarations of functions used in the program are presented.

Creating the List

Just like in the case of the singly linked linear list the operation of creating the list is equivalent to the operation of creating its first element and storing the address of the element in the list pointer. The operation is performed by the `create_list()` function, which definition is presented in the next slide.

Crating the List

```
1  struct list_node *create_list(int number)
2  {
3      struct list_node *first_node = (struct list_node *)
4                                     malloc(sizeof(struct list_node));
5      if(first_node) {
6          first_node->data = number;
7          first_node->previous = first_node->next = NULL;
8      }
9      return first_node;
10 }
```

Creating the List

The `create_list()` function definition for the doubly linked linear list is very similar to its counterpart for the singly linked linear list. Thus only the most important differences are described here. In the 7th line of the function's source code the `NULL` value is assigned not only to the `next` pointer field but also to the `previous` pointer field of the first element. The element is the first and in the same moment the last element of the list. The address returned by the function has to be stored in the list pointer.

Adding an Element to the List

Just like in the case of the singly linked linear list the operation of adding a single element to the doubly linked linear list has to be performed on a nonempty list. Let's assume that the list pointer should point to the first element of the list, after the operation is finished. If the operation fails the list should be in the same state as it was before it begun.

Adding an Element to the List

There are three cases that should be considered when implementing the operation of adding a single element to the doubly linked linear list:

1. the element is added at the front on the list and becomes its first element,
2. the element is added inside the list,
3. the element is added at the end of the list and becomes its last element.

The three cases are handled by separated helper function which are invoked by a single function responsible for the whole operation. First, the definitions of the helper functions are described.

Adding an Element to the List

Adding an Element to the List

```
1  struct list_node *add_at_front(struct list_node *list_pointer,
2                                 struct list_node *new_node)
3  {
4      new_node->next = list_pointer;
5      list_pointer->previous = new_node;
6      return new_node;
7  }
```

Adding an Element to the List

Adding an Element to the List

The `add_at_front()` function, unlike its counterpart for the singly linked linear list, has to take into consideration the `previous` pointer field in the currently first element of the list. Hence, in the 5th line the address of the new element is assigned to the aforementioned field of the currently first element of the list. The rest of the function is the same as in the case of the singly linked linear list.

Adding an Element to the List

Finding a spot

Handling of the two remaining cases requires traversing the list in search of a suitable spot. An address of the element **after** which a new one should be added to the list is an expected result of the operation. To locate such an element the same `find_spot()` function can be used as for the singly linked linear list. Its definition is presented in the next slide. Only one modification has been made to the function. The name of its second parameter is changed.

Adding an Element to the List

Finding a Spot

```
1  struct list_node *find_spot(struct list_node *list_pointer,
2                                                    int number)
3  {
4      struct list_node *previous = NULL;
5      while(list_pointer&&list_pointer->data<number) {
6          previous = list_pointer;
7          list_pointer = list_pointer->next;
8      }
9      return previous;
10 }
```

Adding an Element to the List

Adding Inside the List

```
1  void add_in_middle(struct list_node *node,
2                                      struct list_node *new_node)
3  {
4      new_node->previous = node;
5      new_node->next = node->next;
6      node->next->previous = new_node;
7      node->next = new_node;
8  }
```

Adding an Element to the List

Adding Inside the List

Adding a new element inside the doubly linked linear list is a little more complicated operation than adding a new element to the singly linked linear list, because of the additional pointer field. In the 4th line of the function presented in the previous slide, the address of the element that should precede in the list the new element is assigned to the `previous` field of the new element. The address of the element that should succeed the new element in the list is assigned to the `next` field of the new element in the 5th line. In the 6th line the address of the new element is assigned to the `previous` pointer field of succeeding element. The left side of the assignment statement is quite complex, but it means that the function uses the `next` field of the element that is pointed by the `node` parameter, to get to the element that should succeed the new one in the list and to modify its `previous` pointer field. In the result the new element is partially added to the list.

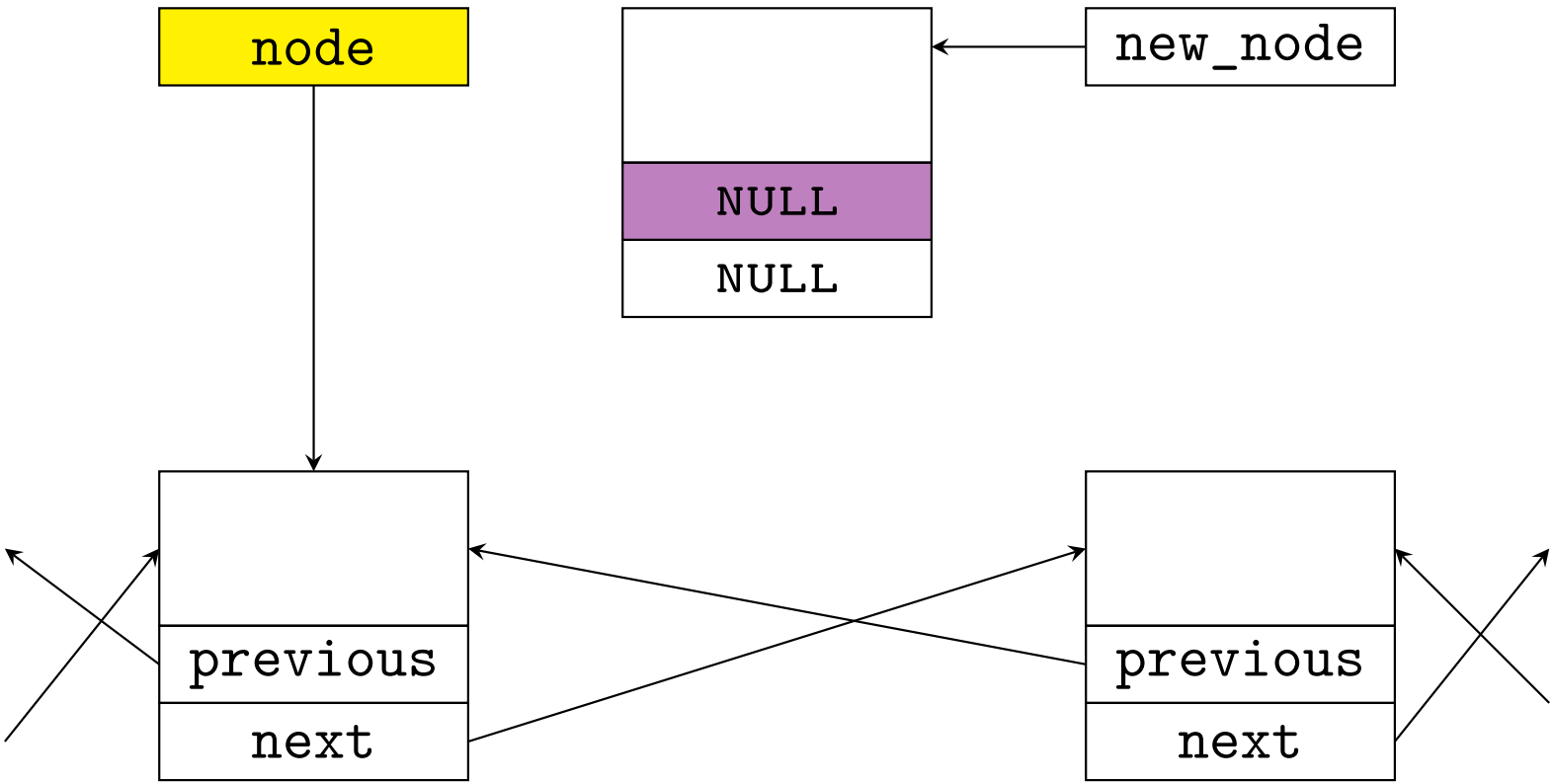
Adding an Element to the List

Adding Inside the List

In the 7th line of the `add_in_middle()` function, the address of the new element is assigned to the `next` pointer field of the element that should precede the new on the list. Please observe, that the lines no. 6 and 7 cannot switch their places. The next slides illustrate the behaviour of the `add_in_middle()` function. The fields that values are copied from are denoted by the yellow colour and the ones that the values are copied to are denoted by the violet colour.

Adding an Element to the List

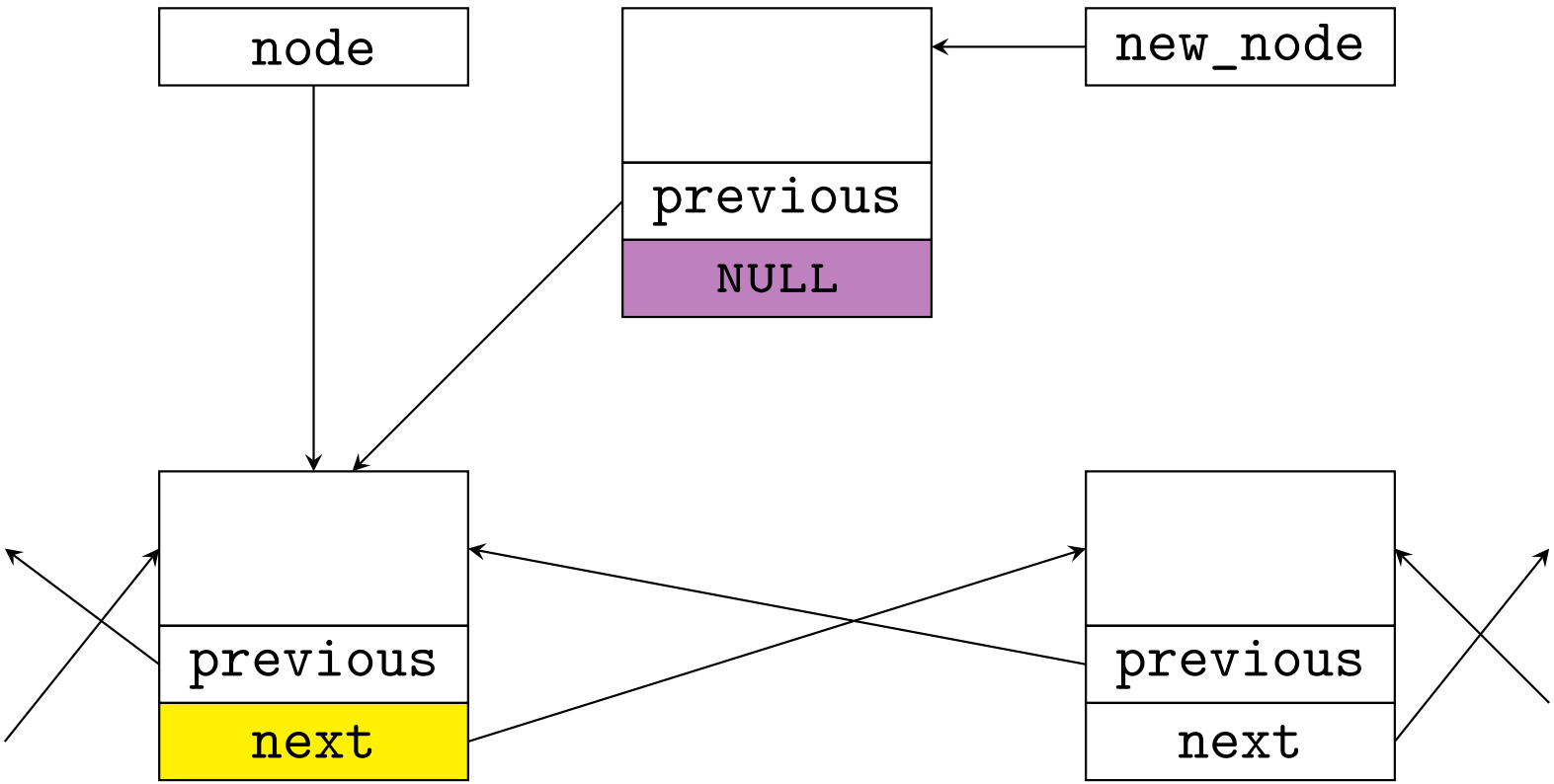
Adding Inside the List



Before performing the 4th line

Adding an Element to the List

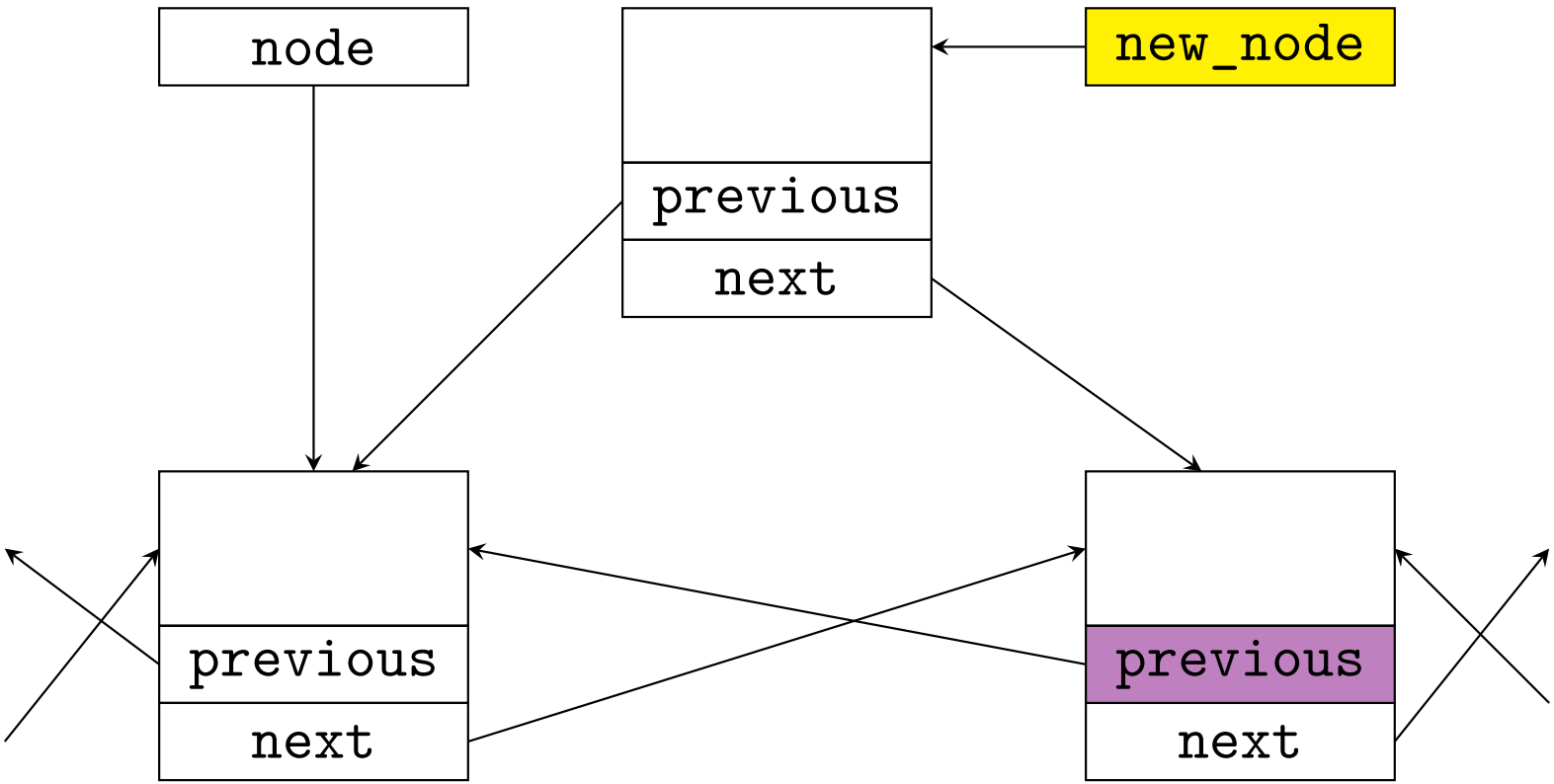
Adding Inside the List



After performing the 4th line

Adding an Element to the List

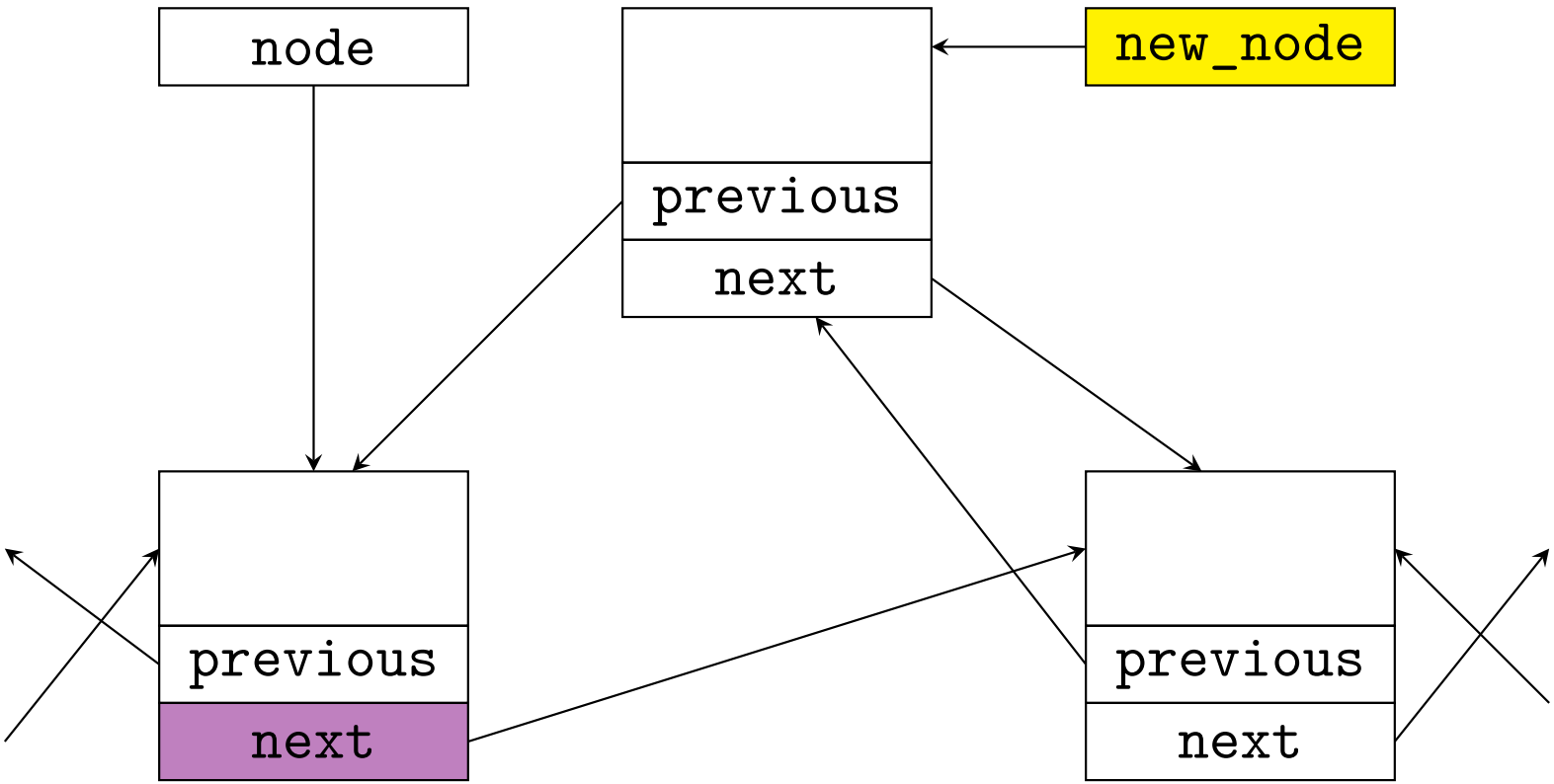
Adding Inside the List



After performing the 5th line

Adding an Element to the List

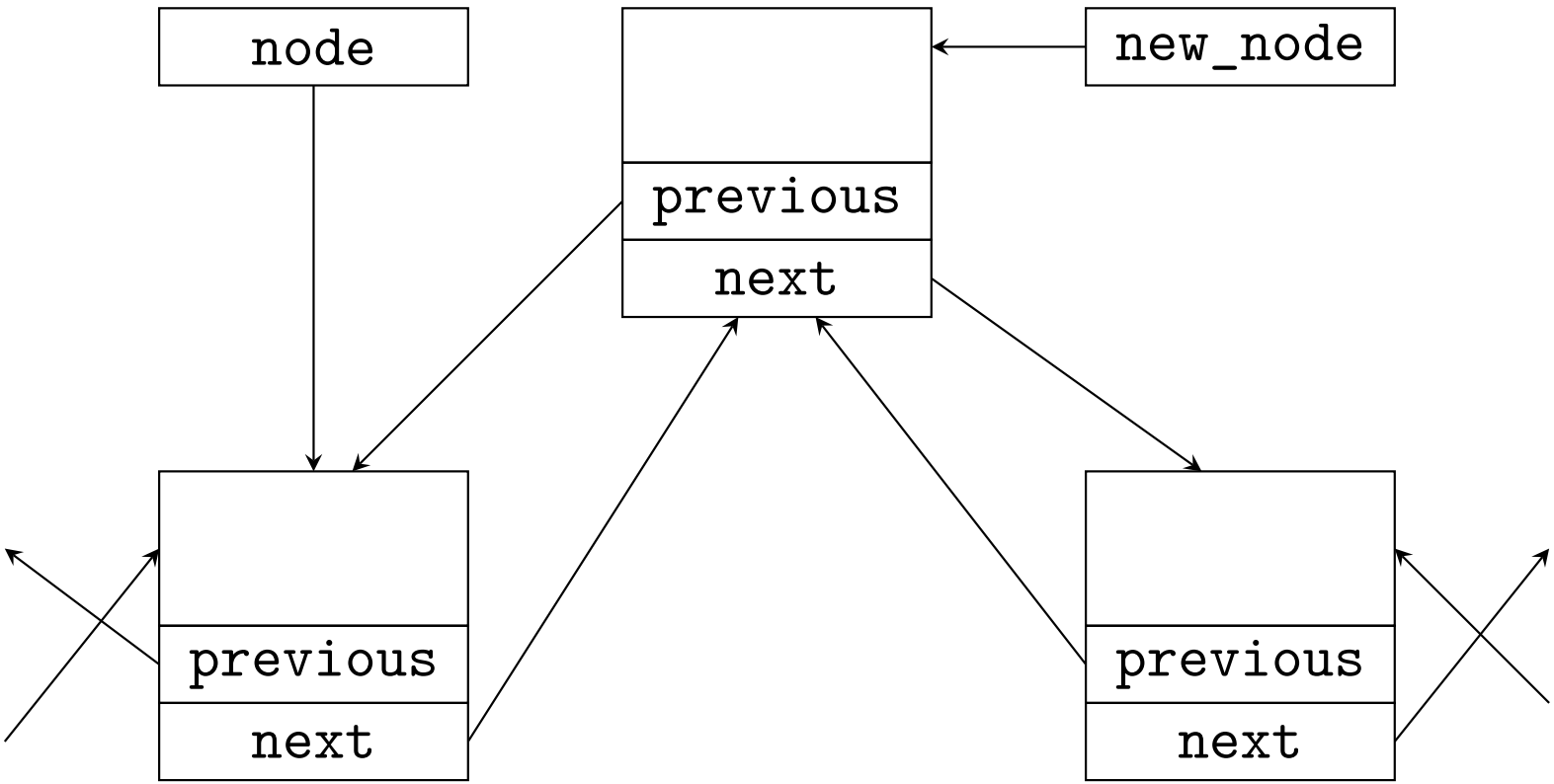
Adding Inside the List



After performing the 6th line

Adding an Element to the List

Adding Inside the List



After performing the 7th line

Adding an Element to the List

Adding at the End of the List

```
1  void add_at_back(struct list_node *last_node,
2                                     struct list_node *new_node)
3  {
4      last_node->next = new_node;
5      new_node->previous = last_node;
6  }
```

Adding an Element to the List

Adding at the End of the List

The `add_at_back()` function takes two pointers as arguments. The first one points to the currently last element of the list and the second one points to the new element that should be added at the end of the list. In the 4th line the address of the new element is assigned to the `next` field of the currently last element of the list. In the result the new element is partially linked to the list. In the 5th line the address of the element pointed by the `last_node` parameter is assigned to the `previous` field of the new element. After the assignment is completed, the new element becomes an integral part of the list, i.e. the last element of the list.

The add_node() Function

```
1   struct list_node *add_node(struct list_node *list_pointer, int number)
2   {
3       if(list_pointer) {
4           struct list_node *new_node = (struct list_node *)
5                                           malloc(sizeof(struct list_node));
6           if(new_node) {
7               new_node->data = number;
8               new_node->previous = new_node->next = NULL;
9               if(list_pointer->data>=number)
10                  return add_at_front(list_pointer,new_node);
11              else {
12                  struct list_node *node = find_spot(list_pointer, number);
13                  if(node->next)
14                      add_in_middle(node, new_node);
15                  else
16                      add_at_back(node, new_node);
17              }
18          }
19      }
20      return list_pointer;
21  }
```

The `add_node()` Function

The definition of the `add_node()` function is little more complicated then its counterpart for the singly linked linear list. The meaning of the parameters and returned value is however the same. The `add_node()` function for the doubly linked linear list first checks if the list pointer is valid (3th line), i.e. the list is not empty. If it is the function performs the statement in the 20th line and exits — the list stays empty. Otherwise the function tries to allocate memory for the new element (4th and 5th lines) and checks if the allocation is successful (6th line). If so, it initializes the fields of the new element (7th and 8th line) and recognizes which of the three cases of adding the new element to the list should be handled. Otherwise the function performs the statement in the 20th line and this time the list also stays unchanged. In the 9th line the function checks if it should add the new element at the beginning of the list. If so, it calls the `add_at_front()` helper function. When the latter finishes its job, the `add_node()` function exits.

The `add_node()` Function

If the new element should be added inside or at the back of the list, the `add_node()` function locates the element after which the new one should be added with the help of the `find_spot()` function. In the 13th line the former checks if the element found by the `find_spot()` function is the last in the list. If so, it calls the `add_at_back()` function and then exits. Otherwise it calls the `add_in_middle()` function and then also exits.

Removing and Element From the List

The operation of removing a single element from the doubly linked linear list is also performed differently than in the case of the singly linked linear list. Despite the similarities, the differences are so great that the implementation of the operation is more complicated. Just like in the case of the previously discussed list, it is required that if the operation is performed for a list with a single element then after it is completed the list should be empty. If the operation is performed for an empty list then the list should stay empty. If the operation is performed for a list with more than one element, then the list should be shorter by one element, provided that the list contains at least one element that should be removed. Finally, if the list doesn't contain any element to remove, it should stay unchanged.

Removing an Element From the List

Just like in the case of the singly linked linear list there are four cases that should be considered when implementing removing the element from the list:

1. removing the first element of the list,
2. removing an element from the inside of the list,
3. removing the last element of the list,
4. the list doesn't contain an element that should be removed.

The first three cases are handled by helper functions that are invoked by the `delete_node()` function. The helper functions are described in the next slides, before the latter function. The forth case doesn't require a separate subroutine to be defined.

Removing From the Front of the List

```
1  struct list_node *delete_at_front(struct list_node *list_pointer)
2  {
3      struct list_node *next = list_pointer->next;
4      if(next)
5          next->previous = NULL;
6      free(list_pointer);
7      return next;
8  }
```

Removing From the Front of the List

The `delete_at_front()` function unlike its counterpart for the singly linked linear list has to take into account the `previous` pointer field of the second element of the list. The element becomes the first element of the list as a result of the removal operation. In the 4th line the function checks if the next (i.e. second) element exists. If not, then the element to be removed is the first and only element of the list and the function performs the statement in the 6th line. Otherwise the `NULL` value is assigned to the `previous` field of the second element of the list (5th line), because this element will become the first element of the list. Only after the assignments are completed the memory for the formerly first element of the list is deallocated (6th line). The function returns the value of the local pointer named `node` and exits.

Searching for an Element

```
1  struct list_node *find_node(struct list_node *list_pointer,
2                                                  int number)
3  {
4      while(list_pointer&&list_pointer->data!=number)
5          list_pointer = list_pointer->next;
6      return list_pointer;
7  }
```

Searching for an Element

The `find_node()` function returns an address of an element of the list that stores in its `data` field the number passed to the function by the `number` parameter. The list pointer is passed to the function by its first parameter. In the `while` loop (4th and 5th lines) the function checks each element of the list if it contains the same number as the `number` parameter. The loop stops when the element is found or the list contains no more elements for verifying. The result of the search (the address of the element or the `NULL` value) is stored in the `list_pointer` parameter. The value of this pointer is returned by the function.

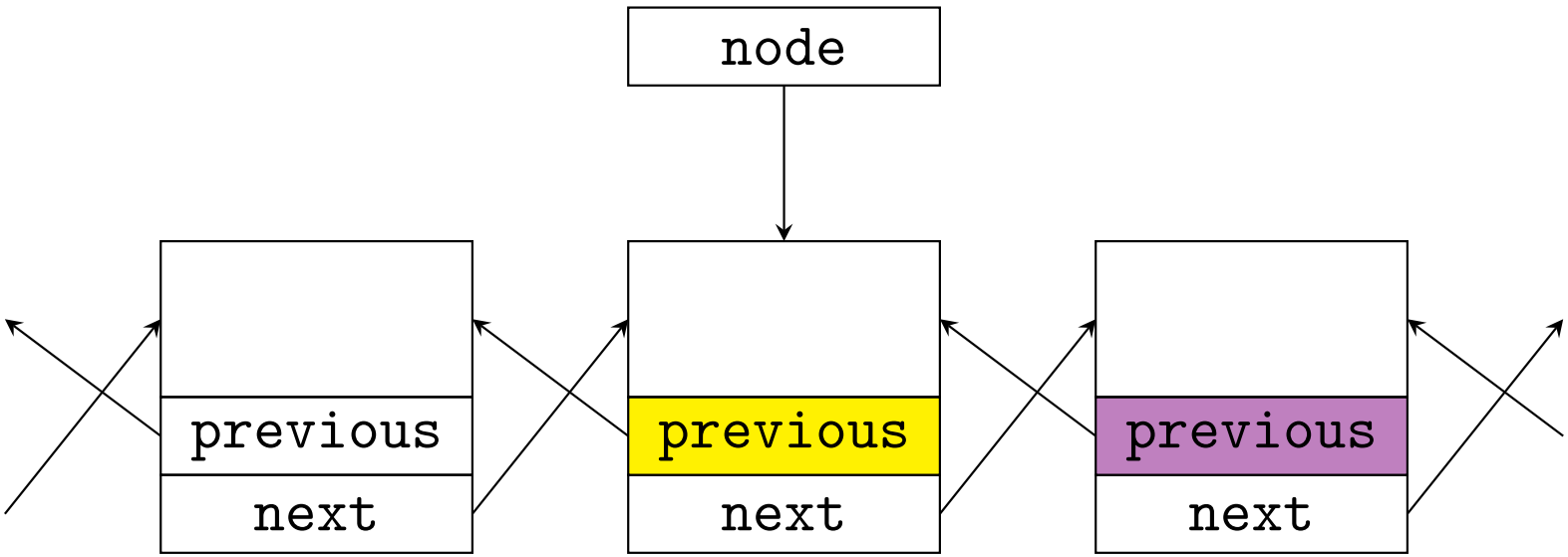
Removing From the Inside of the List

```
1  void delete_in_middle(struct list_node *node)
2  {
3      node->next->previous = node->previous;
4      node->previous->next = node->next;
5      free(node);
6  }
```

Removing From the Inside of the List

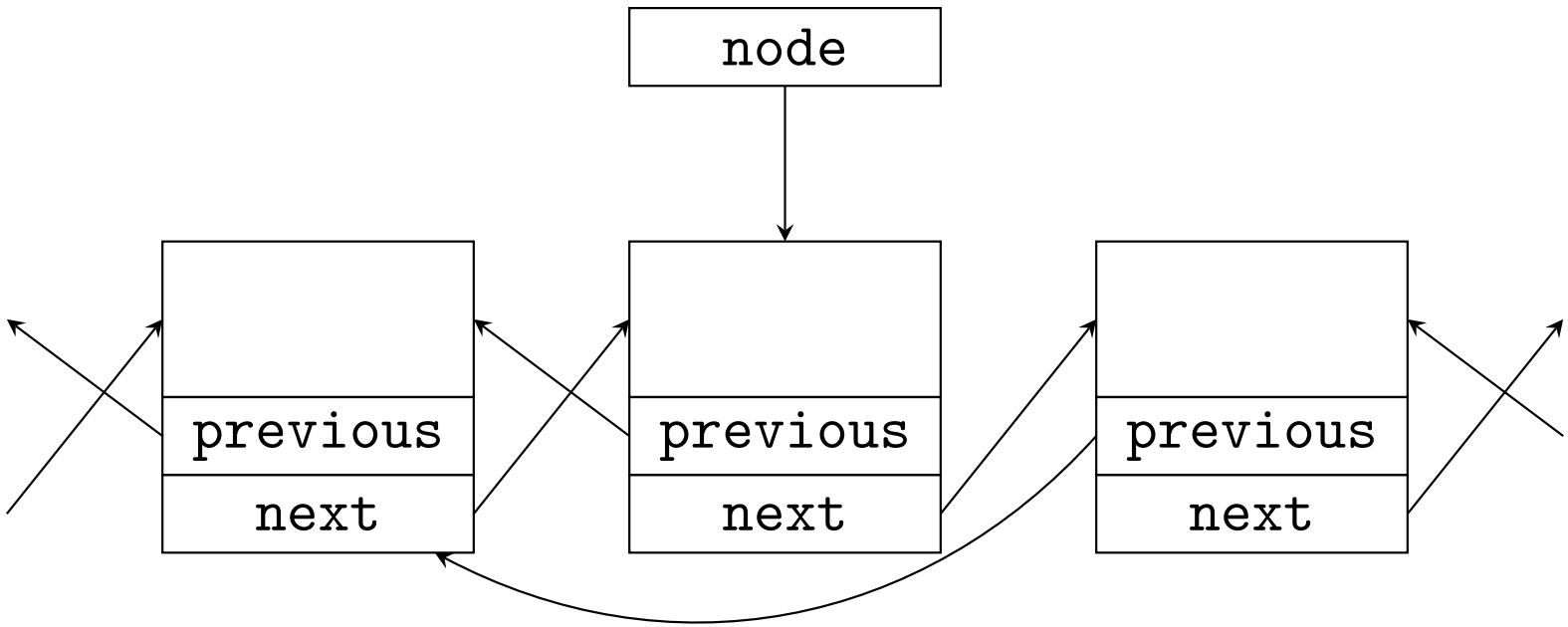
The presented in the previous slide function removes from the inside of the list the element which address is passed by the function's parameter. To this end it unlinks the element from the list in the 3rd and 4th lines and then it deallocates the memory for the element in the 5th line. In the 3rd line the function uses the `node` pointer, that points to the element to be removed, to assign to the `previous` field of the succeeding element the address of the preceding element. In the 4th line a similar work is performed, i.e. the function uses the same pointer to assign to the `next` field of the preceding element the address of the succeeding element. The next slides illustrates the behaviour of the described function. The meaning of the colours is the same as in the previous illustrations.

Removing From the Inside of the List



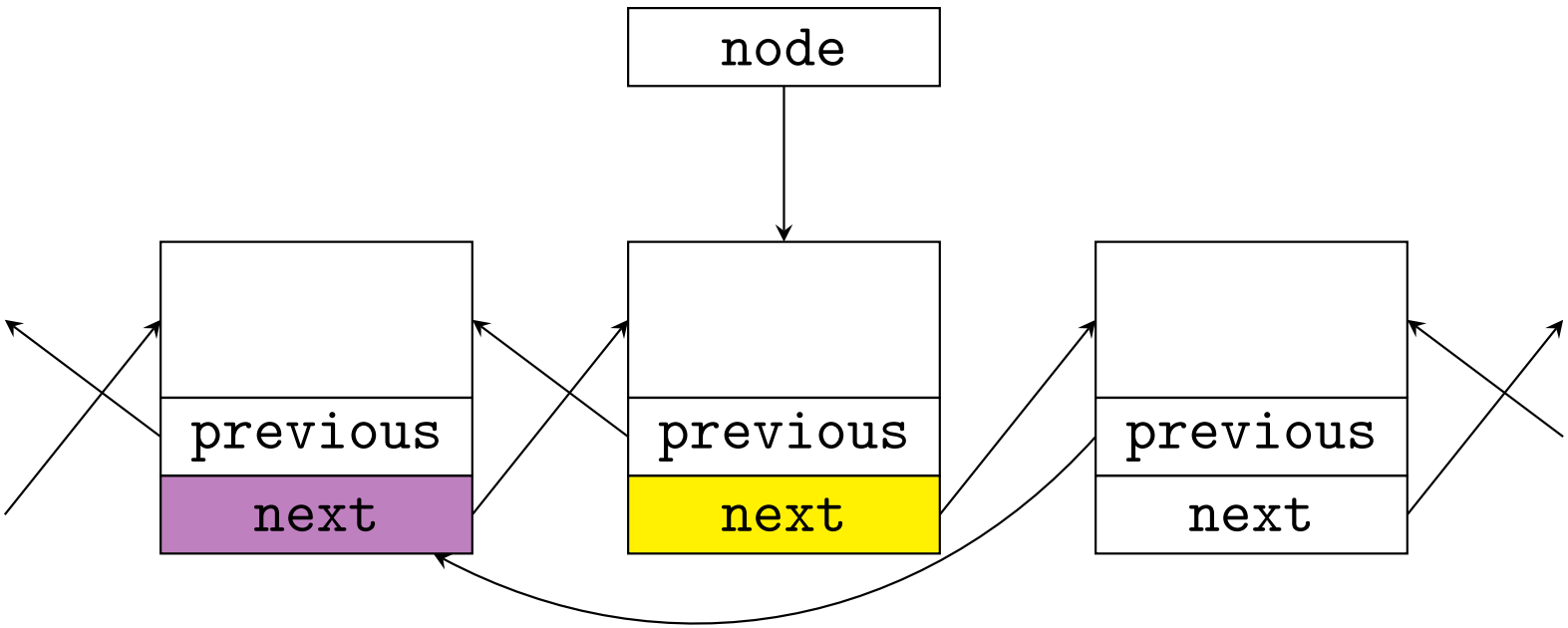
Before performing the 3rd line

Removing From the Inside of the List



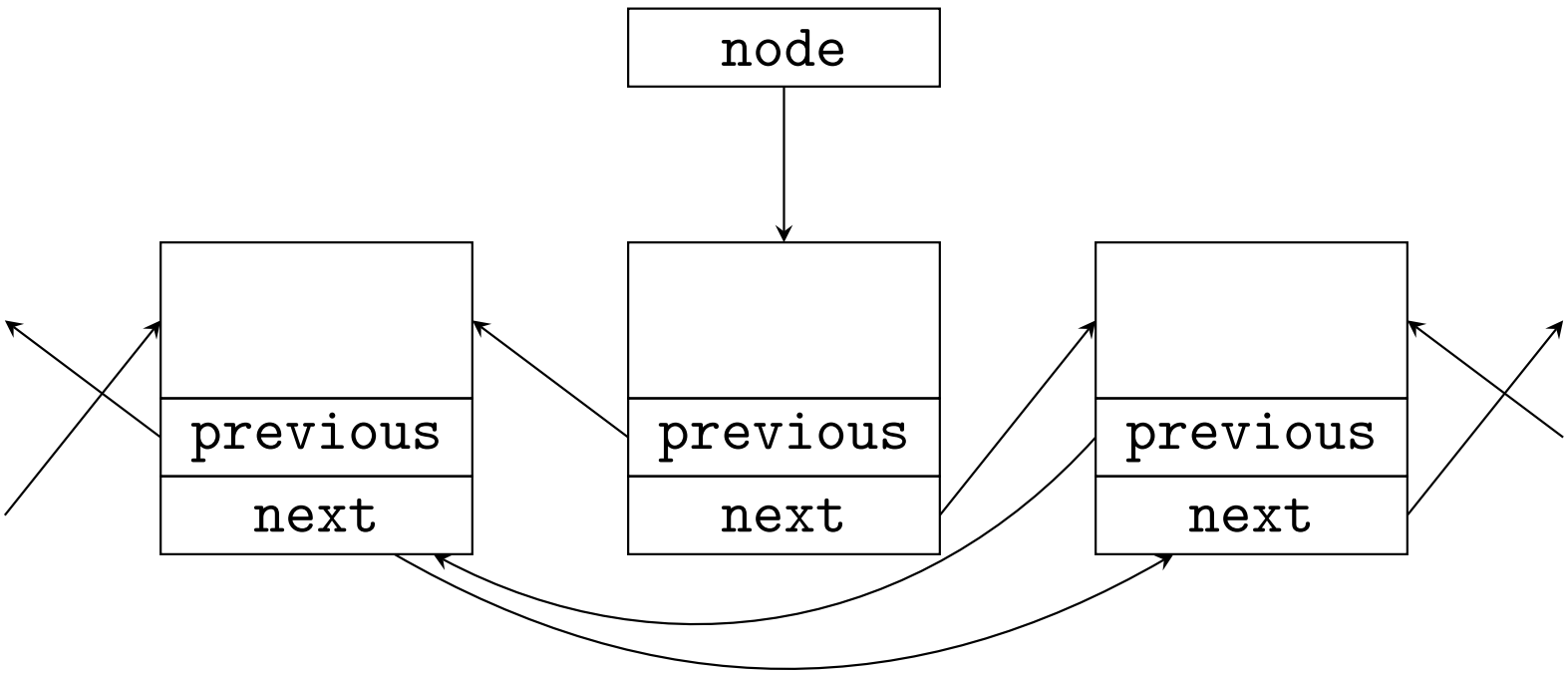
After performing the 3rd line

Removing From the Inside of the List



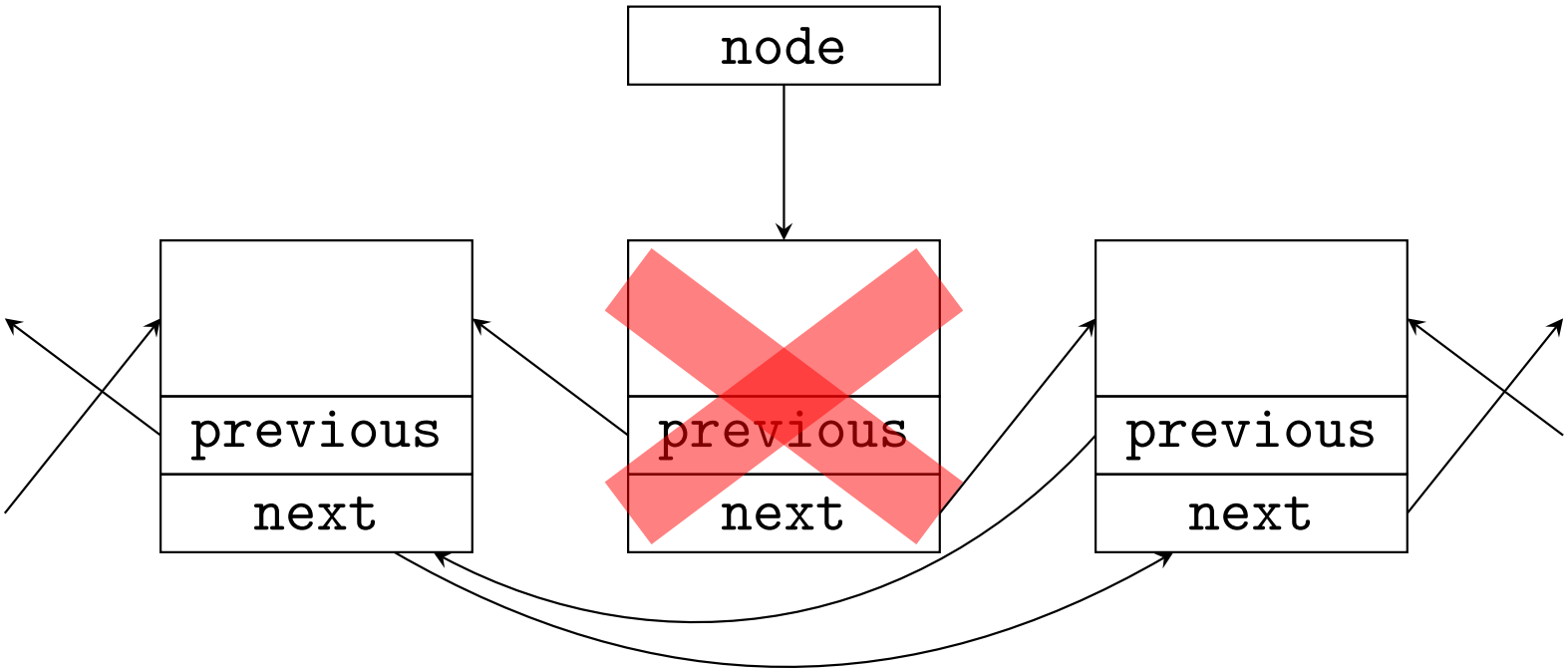
Before performing the 4th line

Removing From the Inside of the List



After performing the 4th line

Removing From the Inside of the List



After performing the 5th line

Removing From the End of the List

```
1  void delete_at_back(struct list_node *last_node)
2  {
3      last_node->previous->next = NULL;
4      free(last_node);
5  }
```

Removing From the End of the List

The `delete_at_back()` function is responsible for removing the last element of the list. The address of the element is passed to the function by its parameter. In the 3rd line the function uses the pointer to the last element to assign the `NULL` value to the `next` pointer field of the preceding element. After that, the latter becomes the last element of the list. In the 4th line the function deallocates the memory allocated for the element pointed by the `last_node` pointer.

Removing From the List

```
1  struct list_node *delete_node(struct list_node *list_pointer,
2                                int number)
3  {
4      if(list_pointer) {
5          if(list_pointer->data==number)
6              return delete_at_front(list_pointer);
7          else {
8              struct list_node *node = find_node(list_pointer,
9                                                 number);
10             if(node) {
11                 if(node->next)
12                     delete_in_middle(node);
13                 else
14                     delete_at_back(node);
15             }
16         }
17     }
18     return list_pointer;
19 }
```

Removing From the List

The `delete_node()` function just like `add_node()` function is more complicated than its counterpart for the singly linked linear list. The meaning of the parameters and the return value is however the same. In the 4th line the function checks if the list is not empty. If it is then it returns the `NULL` value stored in the `list_pointer` parameter. Otherwise the function checks if the first element of the list should be removed. If so, it invokes the `delete_at_front()` function, returns the value returned by the function and exits. If not, it tries to find the address of the element that should be removed from the list, by using the `find_node()` function. If the latter returns the `NULL` value, what is verified in the 10th line of the `delete_node()` function, it means that there is no element to remove in the whole list and the function performs the statement in the 18th line and exits.

Removing From the List

If however, the address returned by the `find_node()` function is not `NULL` then the `delete_node()` function checks in the 11th line if the element to be removed is in the inside of the list. If so, it calls the `delete_in_middle()` function. If not, then it means that the last element of the list should be removed and in the 14th line the `delete_at_back()` function is invoked. Regardless of which of the two helper functions has been called, after its job is completed, the `delete_node()` function returns the value of the `list_pointer` parameter and exits.

Printing the Content of the List

Printing Forwards and Backwards

```
1  void print_list_in_both_directions(struct list_node
2                                                  *list_pointer)
3  {
4      struct list_node *backward_pointer = NULL;
5      while(list_pointer) {
6          backward_pointer = list_pointer;
7          printf("%d ",list_pointer->data);
8          list_pointer = list_pointer->next;
9      }
10     puts("");
11     while(backward_pointer) {
12         printf("%d ",backward_pointer->data);
13         backward_pointer = backward_pointer->previous;
14     }
15     puts("");
16 }
```

Printing the Content of the List

The function defined in the previous slide prints the content of the list's elements forwards and backwards, i.e. starting from the first element of the list and starting from the last element of the list. The first case is performed in the first `while` loop (lines 5–9). In the loop the `backward_pointer` is used that, aside from the first and last iteration, points the element preceding the element pointed by the `list_pointer` parameter. After the first `while` loop stops the `backward_pointer` points the last element of the list. In the second `while` loop this pointer is used for traversing the list backwards. Its value in each of the iterations is replaced by the value stored in the `previous` field of the element currently pointed by this pointer itself.

Removing the List

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      while(*list_pointer) {
4          struct list_node *next = (*list_pointer)->next;
5          free(*list_pointer);
6          *list_pointer = next;
7      }
8  }
```

Removing the List

The function that removes the list from the computer memory is the same as the function that removes the singly linked linear list.

The main() Function

First Part

```
1  int main(void)
2  {
3      list_pointer = create_list(1);
4      int i;
5      for(i=2; i<5; i++)
6          list_pointer = add_node(list_pointer,i);
7      for(i=6; i<10; i++)
8          list_pointer = add_node(list_pointer,i);
9      print_list_in_both_directions(list_pointer);
```

The `main()` Function

First Part

Just like in the case of the singly linked linear list, the double linked linear list is created by adding a single element that stores the number `1`. Then the elements storing the numbers ranging from `2` to `4` and from `6` to `9` are added to the list. Next, the content of the list is displayed on the screen forwards and backwards (9th line).

The `main()` Function

Second Part

```
1       list_pointer = add_node(list_pointer,0);
2       print_list_in_both_directions(list_pointer);
3       list_pointer = add_node(list_pointer,5);
4       print_list_in_both_directions(list_pointer);
5       list_pointer = add_node(list_pointer,7);
6       print_list_in_both_directions(list_pointer);
7       list_pointer = add_node(list_pointer,10);
8       print_list_in_both_directions(list_pointer);
```

The `main()` Function

Second Part

To test the `add_node()` function, in the `main()` function of the program elements are added to the list, that store the following numbers: `0` (added at the front of the list), `5` (added inside the list), `7` (added inside the list, before an element that stores the same value) and `10` (added at the end of the list). After each of the operations is finished the content of the list is displayed forwards and backwards on the screen.

The main() Function

Third Part

```
1       list_pointer = delete_node(list_pointer,0);
2       print_list_in_both_directions(list_pointer);
3       list_pointer = delete_node(list_pointer,1);
4       print_list_in_both_directions(list_pointer);
5       list_pointer = delete_node(list_pointer,1);
6       print_list_in_both_directions(list_pointer);
7       list_pointer = delete_node(list_pointer,5);
8       print_list_in_both_directions(list_pointer);
9       list_pointer = delete_node(list_pointer,7);
10      print_list_in_both_directions(list_pointer);
11      list_pointer = delete_node(list_pointer,10);
12      print_list_in_both_directions(list_pointer);
13      remove_list(&list_pointer);
14      return 0;
15  }
```

The `main()` Function

Third Part

Just like in the case of the `add_node()` function, to verify the behaviour of the `delete_node()` function, from the doubly linked linear list are removed elements of the following values: `0` (removed from the front of the list), `1` (once again removed from the front of the list), `1` (not removed, it doesn't exist now), `5` (removed from the inside of the list), `7` (the first element that stores such a number is removed) and `10` (removed at the end of the list). After each of the operations is completed, the content of the list is displayed forwards and backwards on the screen. Eventually the `main()` function removes the list from the computer memory (13th line) and exits.

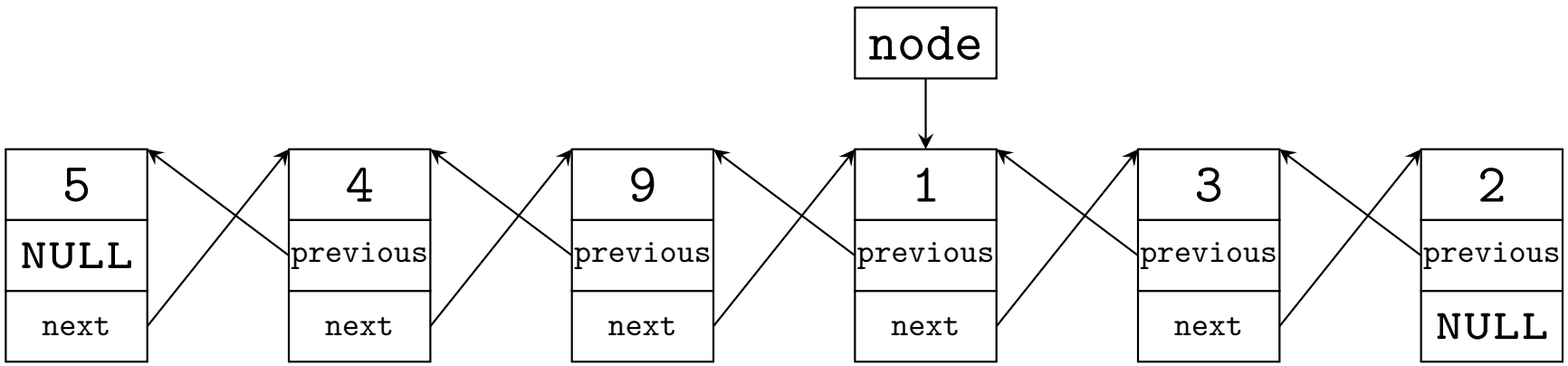
Summary

The presented implementation of the doubly linked linear list is not the only one that can be created. The list can be implemented with the use of a linear or multidimensional array, just like the singly linked linear list. There are also doubly linked linear lists with sentinels. The doubly linked linear list can be applied for building a stack or a queue. In some applications the doubly linked lists have an advantage over the singly linked ones — it is the double link between each of their elements. It is, for example, important in file systems where the lists represent files. In that case they are usually created not in the RAM of the computer but in an external storage, like a hard drive.

Summary

In the presented functions, like in the `delete_in_middle()` function or the `delete_at_back()` function, a complex expressions created with the use of the pointers are applied. The next slide shows even more complicated expressions of this kind. Those are related to the list in the upper part of the slide. The `node` pointer which is present at the beginning of every such an expression is also shown in the figure. Please try to evaluate each of the expressions.

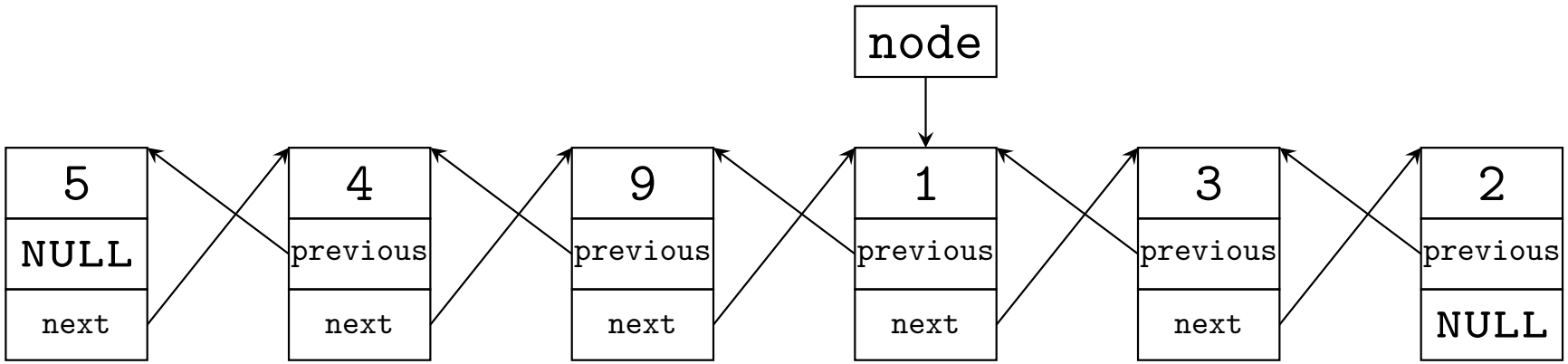
Summary



Expression no. 1

```
node->next->next->data
```

Summary



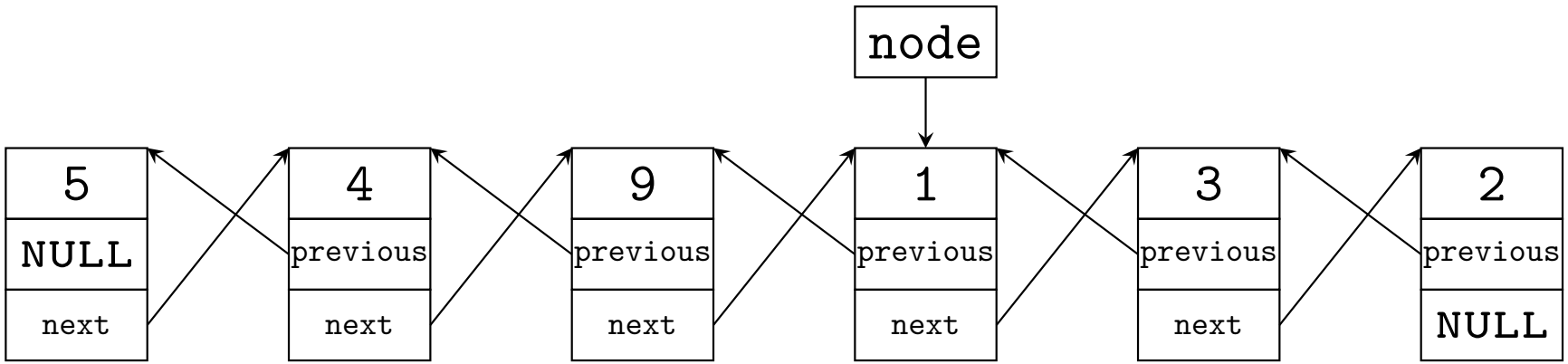
Expression no. 1

```
node->next->next->data
```

Answer no. 1

2

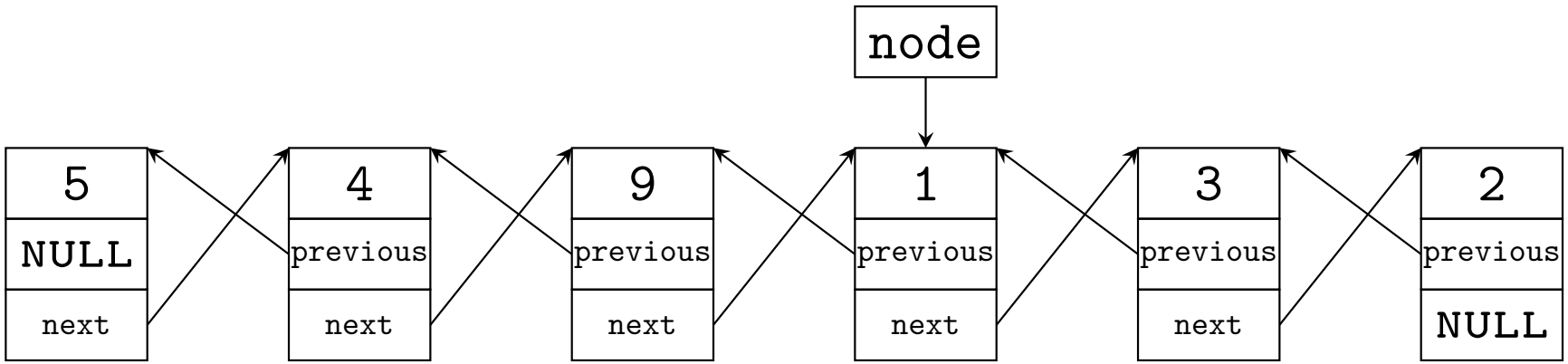
Summary



Expression no. 2

```
node->previous->previous->previous->data
```

Summary



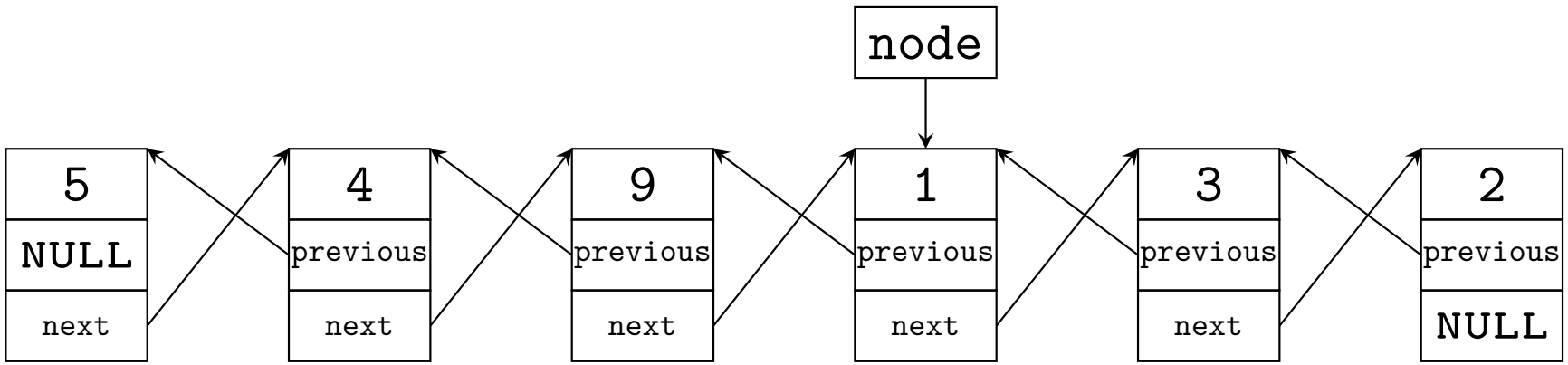
Expression no. 2

```
node->previous->previous->previous->data
```

Answer no. 2

5

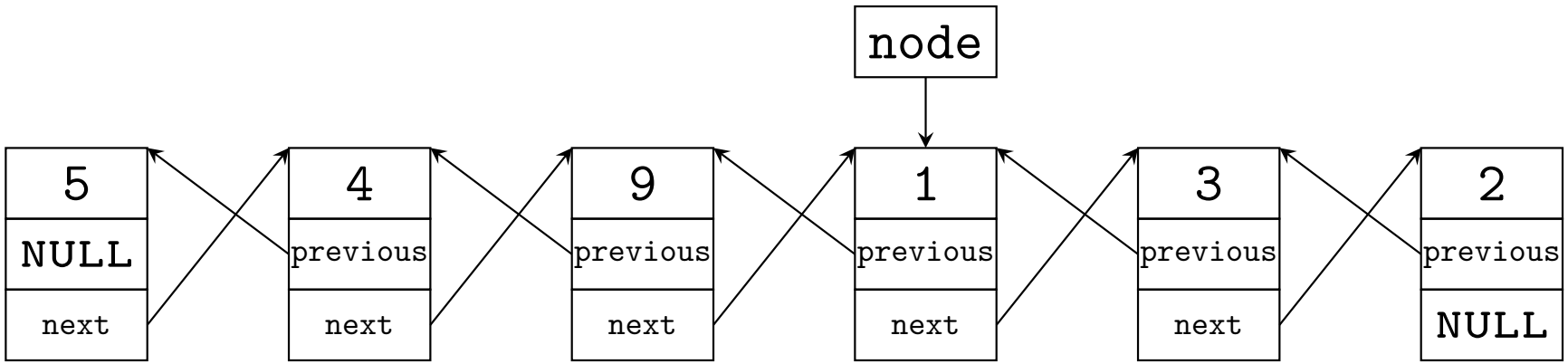
Summary



Expression no. 3

```
node->next->next->previous->previous->previous->previous->data
```

Summary



Expression no. 3

```
node->next->next->previous->previous->previous->previous->data
```

Answer no. 3

4

Summary

The rule for reading such expressions is quite simple — follow the pointers. It is worth to take a closer look at the last expression, where the `previous` and `next` pointers are used together. Those pointers "cancel out" each other, so the expression can be abbreviated to the `node->previous->previous->data` form.
The conclusion from studding such complex pointer expressions is as follows: Every programmer should know how to read such expressions and what they mean, but she or he should avoid using them in programs ☺.

Questions

?

The End

Thank You For Your Attention!